vFabric tc Server Administration

VMware vFabric Cloud Application Platform 5.0

VMware vFabric tc Server 2.6

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-000659-00

You can find the most up-to-date technical documentation on the VMware Web site at: https://www.vmware.com/support.

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to: docfeedback@vmware.com

VMware, Inc., 3401 Hillview Avenue, Palo Alto, CA 94304

www.vmware.com

Table of Contents

1. vFabric tc Server Administration

vFabric tc Server Administration describes how to perform the most common VMware® vFabric™ tc Server administration tasks. Read this documentation to learn how to configure, manage, and monitor tc Runtime instances with the VMware® vFabric™ Hyperic® management tool; configure instances manually with the tc Server command-line interface; and enable clustering for high availability.

Intended Audience

vFabric tc Server Administration is intended for anyone who wants to administer configure and administer tc Server in more advanced ways than described in *Getting Started with vFabric tc Server*.

2. Overview of tc Server Administration

This guide describes how to perform the most common VMware vFabric tc Server administration tasks:

- Configuring and Monitoring tc Runtime Using vFabric Hyperic™. Use the vFabric Hyperic user interface to update the configuration of a tc Runtime instance and to monitor its health and performance.
- Configuring a tc Runtime Instance Manually. Configure a single tc Runtime instance by manually updating its configuration files, such as `server.xml`.
- Using the tc Server Command Line Interface. Use the `tcsadmin` command-line interface and a command script to manage and configure a tc Runtime instance or group.
- Enabling Clustering for High Availability. Create a cluster of tc Runtime instances so as to enable session replication, cluster-wide deployment, and context replication. This section also describes how to enable load balancing.

In procedures that describe how to configure individual tc Runtime instances, it is assumed that you already have created at least one instance and that you now want to change the default configuration to take advantage of tc Server features as well as standard Apache Tomcat features. If you have not created a tc Runtime instance, see "Creating a New tc Runtime Instance" in *Getting Started with vFabric tc Server*.

The tc Server runtime component, tc Runtime, is a servlet container, based on Apache Tomcat, that is hardened for enterprise use, coupled with key operational capabilities and advanced diagnostics, and backed by mission-critical support.

3. Configuring and Monitoring tc Runtime Instances Using Hyperic

vFabric Hyperic is a comprehensive enterprise application management tool. It manages and monitors all instances of vFabric tc Server on any computer, all Spring-powered applications, and a variety of other non-SpringSource platforms and application servers such as Apache Tomcat. Hyperic provides a single console with powerful dashboards from which you can easily check the health of your applications. With Hyperic, you can:

- Manage the lifecycle of tc Runtime instances by starting, stopping, and restarting local or remote instances.
- Similarly manage the lifecycle of a *group* of tc Runtime instances that are distributed over a network of computers.
- Configure a single instance of tc Runtime. Configuration options include the various port numbers to which the tc Runtime instance listens, JVM options such as heap size and enabling debugging, default server values for JSPs and static content, JDBC datasources, various tc Runtime connectors, and so on.
- Configure a group of tc Runtime instances using the `tcsadmin` command.
- Deploy a Web application from an accessible file system, either local or remote. You can deploy to both a single tc Runtime instance or to a predefined group of servers.
- Manage the lifecycle of applications deployed to a single tc Runtime instance or group of instances. Application lifecycle operations include start, stop, redeploy, undeploy, and reload.

In addition to the preceding tc Runtime-related actions, Hyperic performs these standard tasks:

- Inventories the resources on your network.
- Monitors your resources.
- Alerts you to problems with resources.
- Controls the resources.

Getting Started with vFabric tc Server describes how to install and use Hyperic and provides a tutorial that demonstrates the most common tasks.

When using the Hyperic user interface, click on the **Help** link at the top of most pages for detailed online-help. You can also browse the *vFabric Hyperic* documentation for additional information.

User Permissions Required to Use the tc Server Hyperic Plug-in Features

For simplicity, it is often assumed in this documentation that you log in to the Hyperic user interface as the Hyperic super-user (`hqadmin`) when you want to manage a tc Runtime instance. This is not required, of course. You can also log in as a non-super user and still use the tc Server Hyperic plugin features, as long as the user has the correct permissions.

In Hyperic, users are assigned roles, which in turn are assigned permissions, such as **View** and **Control**. For general information about what each permission means with respect to server resources (such as a tc Runtime instance) in Hyperic, see "Create and Manage Roles in vFabric Hyperic" in *vFabric Hyperic Administration*. For general information about the default users in Hyperic and creating new ones, see Create and Manage User Accounts in *vFabric Hyperic Administration*.

The following table describes the *additional* effects that some of the Hyperic permissions have on the tc Server Hyperic plugin features. Use this table to determine which role you should assign a user that will be managing tc Runtime instances.

Table 3.1. Hyperic Permission Effects on tc Server Hyperic Plug-in Features

Permission	Additional Effect on tc Server Hyperic Plug-in Features
View	Allows the user to:

Permission	Additional Effect on tc Server Hyperic Plug-in Features
	• View the deployed Web applications in the **Views > Application Management** tab. • View the current configuration of a tc Runtime instance in the **Views > Server Configuration** tab.
Modify	Allows the user to: • Update the fields in the **Views > Server Configuration** tab and then push the data to the configuration files associated with the tc Runtime instance, such as `server.xml`. • Use the application lifecycle commands of the **Views > Application Management** tab to start, stop, reload, or undeploy a Web application.
Control	Allows the user to use the commands in the **Control** tab to start, stop, and restart a tc Runtime instance.

Managing tc Runtime-Related Hyperic Alerts

tc Server includes a full set of diagnostic features that make it easy to troubleshoot problems with tc Runtime instances and the applications that you deploy to them. For each diagnostic feature, the tc Server Hyperic plug-in has one or more corresponding preconfigured alerts.

After Hyperic triggers an alert associated with a diagnostic feature (because the associated condition has been met), Hyperic disables the alert until an administrator marks it as Fixed. You can use Hyperic to further configure this alert with additional control actions or even disable it, as described in the following sections:

- Viewing and Changing the Preconfigured Alerts
- Viewing and Changing the Metric Collection Interval
- Deadlock Detection
- Excessive Time in Garbage Collection
- Slow or Failed Requests
- JDBC Connection Monitoring

Viewing and Changing the Preconfigured Alerts

The preconfigured Hyperic alerts associated to the diagnostic features of tc Runtime work on one of two Hyperic resources: either the tc Runtime instance itself, or with a service of the tc Runtime instance. This information is important to know because it determines how you view, and optionally change, a particular alert.

The following table lists each preconfigured alert and the Hyperic resource type to which it is associated. The resource type `SpringSource tc Runtime 6.0` refers to the tc Runtime instance itself; the resource type `SpringSource tc Runtime 6.0` *`Service`*, such as `SpringSource tc Runtime 6.0 Thread Diagnostics`, refers to a service of the tc Runtime instance.

Note: The tc Runtime version is associated with the core version of Tomcat on which the runtime is based, rather than the version of the tc Server bundle.

The third column in the table indicates whether the alert is triggered by a metric condition or an event/log level condition. If the former, the name of the metric is displayed; if the latter, the specific string in the log (if any) that triggers the alert is displayed.

Table 3.2. Preconfigured tc Runtime Alerts

Alert Name	Associated Hyperic Resource Type	Metric or Events/Log Level Based?
Deadlocks Detected	SpringSource tc Runtime 6.0 and SpringSource tc Runtime 7.0	Metric (Deadlocks Detected)
Excessive Time Spent in Garbage Collection	SpringSource tc Runtime 6.0 and SpringSource tc Runtime 7.0	Metric (Percent Up Time in Garbage Collection)

Alert Name	Associated Hyperic Resource Type	Metric or Events/Log Level Based?
Slow or Failed Request	SpringSource tc Runtime 6.0 and SpringSource tc Server 7.0 Thread Diagnostics	Events/Logs Level.
JDBC Connection Abandoned	SpringSource tc Runtime 6.0 and SpringSource tc Server 7.0 Tomcat JDBC Connection Pool Global	Events/Logs Level (CONNECTION ABANDONED)
JDBC Connection Failed	SpringSource tc Runtime 6.0 and SpringSource tc Server 7.0 Tomcat JDBC Connection Pool Global	Events/Logs Level (CONNECTION FAILED)
JDBC Query Failed	SpringSource tc Runtime 6.0 and SpringSource tc Server 7.0 Tomcat JDBC Connection Pool Global	Events/Logs Level (FAILED QUERY)
Slow JDBC Query	SpringSource tc Runtime 6.0 and SpringSource tc Server 7.0 Tomcat JDBC Connection Pool Global	Events/Logs Level (SLOW QUERY)

The following procedure summarizes how to view and change preconfigured alerts. For a detailed tutorial that shows how to view and change the Deadlocks Detected alert, see "Tutorial: Using Hyperic to Configure and Manage tc Runtime Instances" in *Getting Started with vFabric tc Server*.

1. Browse to the resource to which the alert is associated, as described in the preceding table. See "Getting Started with the Hyperic User Interface" in *Getting Started with vFabric tc Server* for information about browsing to Hyperic resources.
2. Click the **Alert** tab.
3. Click the **Configure** button. A table of alerts currently configured for the resource is displayed.
4. Click the name of the alert. The Alert Definition page for the alert is displayed.

   The definition page has three sections: the top Alert Properties section provides general properties of the alert; the middle Condition Set section describes the conditions that trigger the alert; and a series of tabs at the bottom enable you to configure the particular control action that occurs if the alert is triggered, the escalation scheme, who should be notified if the alert is triggered, and so on.
5. If you want to change the general properties, conditions, control actions, and so on of the alert, click the appropriate **EDIT...** button, make your changes, then click **OK**.
6. To disable the alert, go back to the Alert Definitions table, select the name of the alert by checking the box to the left of its name, then select `No` for the **Set Active** drop-down list and click the arrow to the right.

The remainder of this chapter describes each alert in more detail, including any special instructions to enable the alert.

Viewing and Changing the Metric Collection Interval

As shown in the Preconfigured tc Runtime Alerts table, the two alerts associated with the tc Runtime instance itself use metrics in their condition to determine whether the alert should be triggered. The following procedure describes how you can view, and optionally change, the collection interval for Deadlock Detection and Excessive Time in Garbage Collection.

1. Click the **Administration** tab at the top of the Hyperic user interface.
2. In the Hyperic Server Settings section, click the **Monitoring Defaults** link.
3. Scroll down until you find the `SpringSource tc Runtime 6.0` or `SpringSource tc Runtime 7.0` entry in the Server Types table, and then click the **EDIT TEMPLATE METRIC** link to the right.

   A page shows all metrics associated with the tc Runtime instance. For example, under Utilization you will find the Deadlocks Detected metric. By default, the Collection Interval column shows that Hyperic Server collects information about this metric every 2 minutes.

4. To change the collection interval for a specific metric, select it by clicking the box to the left of its name.

5. Enter the new collection interval at the bottom of the page in the Collection Interval for Selected field, specify whether it is in minutes or hours, then click the arrow to the right.

Deadlock Detection

The tc Runtime automatically detects whether a thread deadlock occurs in a tc Runtime instance or an application deployed to the instance.

The out-of-the-box Hyperic alert is triggered if the Deadlocks Detected metric exceeds 0. Hyperic checks the metric every two minutes to see whether the condition is met. Hyperic applies this alert to all auto-discovered tc Runtime instances and enables it by default. This alert is associated with the tc Runtime instance itself.

For a detailed tutorial that shows how to view and change the Deadlocks Detected alert, see "Tutorial: Using Hyperic to Configure and Manage tc Runtime Instances" in *Getting Started with vFabric tc Server*.

Excessive Time in Garbage Collection

A Hyperic metric represents the percentage of process up time (0 -100) that the tc Runtime instance has spent in garbage collection.

The alert is triggered when the total garbage collection time is excessive (by default, 40% of process up time.) Hyperic checks this metric every 5 minutes to see if the condition has been met. Hyperic applies this alert to all auto-discovered tc Runtime instances and enables it by default.

Enabling the Slow or Failed Request Alert

When clients begin connecting and using a Web application deployed to a tc Runtime instance, they may encounter slow or failed requests. Although the tc Runtime instance logs these errors in the log files by default, it is often difficult to pinpoint the exact origin of the error and how to go about fixing it. By enabling thread diagnostics, tc Runtime provides additional information to help you troubleshoot the problem.

A *failed* request is one that simply did not execute; a *slow* request is a request that takes longer than a certain threshold. The default threshold is 500 milliseconds.

When you enable thread diagnostics, you can view the following contextual information about a slow or failed client request:

- Time and date of the slow or failed request.
- Exact URL invoked by the client that resulted in a slow or failed request.
- Exact error returned by the request.
- Database queries that were executed as part of the request and how long each one took.
- Whether any database connection failed or succeeded.
- Whether the database had any other connectivity problems.
- Whether the database connection pool ran out of connections.
- Whether any garbage collection occurred during the request, and if so, how long it took.

The associated Hyperic alert is triggered if a client request to tc Runtime is slow (over a configured threshold) or if it failed.

This alert is not enabled by default. Explicitly enable it as follows:

1. Browse to the **Views > Server Configuration** console page for the tc Runtime instance.

2. Click the **Services** tab.

3. In the table, click the service you want to configure; the default tc Runtime service is called `Catalina`.
4. In the **Thread Diagnostics** section, check the **Enable Thread Diagnostics** property.
5. At the bottom of the page, click **Save**.
6. Click the necessary links and buttons to push configuration changes to the tc Runtime instance and restart the instance.

Enabling JDBC Connection Monitoring

Hyperic includes a new service called `SpringSource tc Runtime 6.0 Tomcat JDBC Connection Pool Global` that represents any high-concurrency Tomcat JDBC datasources you might have configured for your tc Runtime instance. This service monitors the health of the datasource, such as whether its connection to the database has failed or was abandoned, and whether the JDBC queries that clients execute are taking too long. Hyperic creates this service when you create a new Tomcat JDBC datasource; one instance of a service exists per datasource.

Four Hyperic alerts are associated with this diagnostic feature; they are triggered as follows:

- **JDBC Connection Failed**: A particular high-concurrency JDBC connection that uses a configured datasource fails.
- **JDBC Connection Abandoned**: A particular high-concurrency JDBC connection that uses a configured datasource is abandoned by the database server.
- **JDBC Query Failed**: A high-concurrency JDBC query fails.
- **Slow JDBC Query**: A high-concurrency JDBC query takes too long to execute.

To receive monitoring information for the preceding JDBC alerts, enable log tracking for this service:

1. Browse to the `SpringSource tc Runtime 6.0 Tomcat JDBC Connection Pool Global` service associated with your JDBC datasource.
2. Click the **Inventory** tab.
3. In the **Configuration Properties** section, be sure that the **service.log_track.enable** property is checked. Checking this box subscribes Hyperic to JMX notifications sent from the tc Runtime instance, which then get displayed in Hyperic as log events.

Securing the Hyperic Server

The Hyperic Server includes a default self-signed SSL certificate in its keystore; the same certificate is shipped with every Hyperic Server. Because this certificate is so readily available, anyone who connects to your particular Hyperic Server (assuming an out-of-the-box configuration) using HTTPS cannot in reality trust the certificate. Although this is adequate in the testing phase of your application, in production you typically want to configure Hyperic Server more securely. This section describes the basics steps for securing the Hyperic Server when connecting to it over HTTPS.

It is assumed that you understand basic SSL concepts such as certificates, public and private keys, keystores, and truststores. It is also assumed that you know how to get a certificate from a trusted certificate authority or how to generate your own. The main focus in this section is how to update the Hyperic Server configuration so that the server uses your certificate.

1. Obtain a certificate from a trusted certificate authority (CA) such as Verisign or create your own.

   Use the keytool command-line tool, provided in the Sun JDK, to generate a certificate. The `keytool` link also tells you how to get a certificate from a CA.

2. Install the certificate into the Hyperic Server keystore. When you first install Hyperic Server, this keystore contains a self-signed certificate; replace the default certificate with your own certificate that you got from a CA or that you generated with a tool such as `keytool`.

Update the Hyperic Server's default keystore file:

- Keystore name: `hyperic.keystore`
- Keystore location: *`INSTALL_DIR`*`/server-4.6.X.X-EE/hq-engine/hq-server/conf` directory, where *`INSTALL_DIR`* refers to the directory in which you installed Hyperic Server, such as `/opt/vmware/hyperic`.
- Keystore password: `hyperic`.

As with generating your own certificate, you can also use the keytool command-line tool to update a keystore.

3. **Optional Step:** Change the default Hyperic Server keystore and truststore filename, location, and password by updating *`INSTALL_DIR`*`/server-4.6.X.X-EE/hq-engine/hq- server/conf/server.xml`. This step is unnecessary if you simply install your certificate in the default Hyperic Server keystore file, as described in the preceding step.

   In the `server.xml` file, edit the `<Connector>` element that corresponds to the SSL port, which is the port that Hyperic Server uses for HTTPS. The `keystoreFile`, `keystorePass`, `truststoreFile`, and `truststorePass` attributes identify the keystore and truststore files and their passwords. The following snippet shows the default `<Connector>` configuration:

```
<Connector port="${server.webapp.secure.port}"
         executor="tomcatThreadPool" maxHttpHeaderSize="8192"
         emptySessionPath="true" protocol="HTTP/1.1" SSLEnabled="true"
         scheme="https" secure="true" clientAuth="false"
         keystoreFile="${catalina.base}/conf/hyperic.keystore"
         keystorePass="hyperic"
         truststoreFile="${catalina.base}/conf/hyperic.keystore"
         truststorePass="hyperic"
         sslProtocol = "TLS" />
```

   In the preceding snippet, the variable `${catalina.base}` points to the *`INSTALL_DIR`*`/server-4.6.X.X-EE/hq-engine/hq-server` directory.

4. Restart the Hyperic Server for the changes to take effect.

The Hyperic Server then uses your certificate rather than the unsecure out-of-the-box certificate installed with Hyperic Server.

When you next use a browser to invoke the Hyperic user interface, the browser automatically trusts a certificate from a certificate authority (CA) such as VeriSign, or it asks whether you want to trust an unrecognized certificate, and updates its internal trust store accordingly.

If you are using the `tcsadmin` command-line interface to access the Hyperic Server, you need to update the truststore only on the corresponding client computer if the signing authority is not already trusted. If you do need to update the truststore, add the public key of the new certificate that you previously installed in the Hyperic Server's keystore. You do this by updating the default truststore in the client's JVM (either the *`JAVA_HOME`*`/lib/security/jssecacerts` or *`JAVA_HOME`*`/lib/security/cacerts` file) or by creating a new truststore and pointing to it using the `javax.net.ssl.trustStore` system property. For more information, see Customizing the Default Key and Trust Stores, Store Types, and Store Passwords.

4. Configuring a tc Runtime Instance Manually

When you first install tc Runtime, the `server.xml` file contains typical server configuration values that get you up and running immediately. However, as you use tc Runtime and go into production, you might require additional configuration. This chapter describes typical and additional configuration use cases.

Configuration Files and Templates

The tc Runtime configuration files are located in the *`CATALINA_BASE`*`/conf` directory, where *`CATALINA_BASE`* refers to the directory in which you have installed a tc Runtime instance. The main configuration files are:

- **`server.xml`**. Main configuration file for a tc Runtime instance. It configures the behavior of the servlet/JSP container. By default, the `server.xml` file for a tc Runtime instance uses variable substitution for configuration properties such as HTTP and JMX port numbers that must be unique across multiple server instances on the same computer. These variables take the form `${var}`. For example, the variable for the HTTP port that the tc Runtime instance listens to is `${http.port}`. The specific values for these variables for a particular server instance are stored in the `catalina.properties` file, in the same directory as the `server.xml` file.
- **`catalina.properties`**. Properties file that contains the server instance-specific values for variables in the `server.xml` file.

The `conf` directory also contains the following two files that configure common properties for *all* Web applications deployed to the tc Runtime instance:

- **`web.xml`**. Defines default values for all Web applications.
- **`context.xml`**. The contents of this file will be loaded for each Web application.

The tc Runtime installation also includes a set of configuration *templates* in the `INSTALL-DIR/springsource-tc-server-`*`edition`*`/templates` directory, where *`edition`* refers to the edition of vFabric tc Server that you are using, whether `developer` or `standard`. You can specify these templates when you create a new tc Runtime instance to automatically enable certain configuration features, such as SSL or clustering. Each template is a directory that contains new, modified, or fragments of files that the `tcruntime-instance` script uses to modify the default tc Runtime instance files. Many of the templates change the default `server.xml` file, so you can also look at the `server-fragment.xml` files in the various template directories for examples of configuring an existing tc Runtime instance. The `server-fragment.xml` files are fragments of the `server.xml` file that the `tcruntime-instance` script applies to the default tc Runtime configuration so as to enable a particular feature.

For details about the templates provided by tc Runtime, see "Creating a tc Runtime Instance Using a Template" in *Getting Started with vFabric tc Server*.

Simple tc Runtime Configuration

The following sample `server.xml` file shows a basic out-of-the-box configuration for a default tc Runtime instance included in tc Runtime. This configuration file uses typical values for a standard set of XML elements. Sample `server.xml` files in later sections of this documentation build on this file.

This `server.xml` file uses variable substitution for configuration properties, such as HTTP and JMX port numbers, that must be unique across multiple server instances on one computer. These variables take the form `${var}`. For example, the variable for the HTTP port that the tc Runtime instance listens to is `${http.port}`. The specific values for these variables for a particular server instance are stored in the `catalina.properties` file, located in the same directory as the `server.xml` file. A snippet of the default `catalina.properties` file is shown after the sample `server.xml` file.

See Description of the Basic server.xml File for information about the elements and attributes in this sample configuration file in case you need to change them to suit your own environment.

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="${shutdown.port}" shutdown="SHUTDOWN">
```

```
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />

  <Listener className="com.springsource.tcserver.serviceability.rmi.JmxSocketListener"
            port="${jmx.port}"
            bind="127.0.0.1"
            useSSL="false"
            passwordFile="${catalina.base}/conf/jmxremote.password"
            accessFile="${catalina.base}/conf/jmxremote.access"
            authenticate="true"/>

  <Listener className="com.springsource.tcserver.serviceability.deploy.TcContainerDeployer" />

  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
  </GlobalNamingResources>


  <Service name="Catalina">

    <Executor name="tomcatThreadPool" namePrefix="tomcat-http--" maxThreads="300" minSpareThreads="50"/>

    <Connector
              executor="tomcatThreadPool"
              port="${http.port}"
              protocol="HTTP/1.1"
              connectionTimeout="20000"
              redirectPort="8443"
              acceptCount="100"
              maxKeepAliveRequests="15"/>

    <Engine name="Catalina" defaultHost="localhost">

      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase"/>

      <Host name="localhost"  appBase="webapps"
            unpackWARs="true" autoDeploy="true" deployOnStartup="true" deployXML="true"
            xmlValidation="false" xmlNamespaceAware="false">
      </Host>
    </Engine>
  </Service>
</Server>
```

The following snippet of `catalina.properties` shows how to set values for the variables used in the preceding `server.xml` file.

```
base.shutdown.port=-1
base.jmx.port=6969
bio.http.port=8080
bio.https.port=8443
```

Description of the Basic server.xml File

Note the following components of the preceding sample `server.xml`:

- **`<Server>`**. Root element of the `server.xml` file. Its attributes represent the characteristics of the entire tc Runtime servlet container. The `shutdown` attribute specifies the command string that the shutdown port number receives through a TCP/IP connection in order to shut down the tc Runtime instance. The `port` attribute is the TCP/IP port number that listens for a shutdown message for this tc Runtime instance; note that in this `server.xml` file the variable is `${shutdown.port}`. By default, the `catalina.properties` file substitutes a value of `-1`, which disables the shutdown via TCP connection. Thus

the only way to stop the tc Runtime instance is to issue a `kill` command on the process ID (PID) of the tc Runtime instance. This is what the `tcruntime-ctl.sh` command does when you use it to stop a running tc Runtime instance.

- **`<Listener>`**. List of lifecycle listeners that monitor and manage the tc Runtime instance. Each listener listens to a specific component of the tc Runtime instance and has been programmed to do something at certain lifecycle events of the component, such as before starting up, after stopping, and so on.

  The first three `<Listener>` elements configure standard Tomcat lifecycle listeners.

  You can insert a `com.springsource.tcserver.properties.SystemProperties` listener before these standard listeners to set properties from external properties files. See Adding a System Properties Listener.

  The listener implemented by the `com.springsource.tcserver.serviceability.rmi.JmxSocketListener` class is specific to tc Server. This listener enables JMX management of tc Runtime; in particular, this is the JMX configuration that the HQ user interface uses to monitor tc Runtime instances. The `port` attribute specifies the port of the JMX server that monitoring products, such as Hyperic HQ, connect to. The variable `${jmx.port}` is set to `6969` in the default `catalina.properties` file. The `bind` attribute specifies the host of the JMX server; by default, this attribute is set to the localhost (`127.0.0.1`).

  Warning: The value of the `bind` attribute of `JmxSocketListener` overrides the value of the `java.rmi.server.hostname` Java system property. This directly affects how names are bound in the RMI registries; by default, the names will be bound to localhost (`127.0.0.1`). This means that RMI clients running on a different host from the one hosting the tc Runtime instance will be unable to access the RMI objects because, from their perspective, the host name is incorrect. This is because the host should be the name or IP address of the tc Runtime computer rather than `localhost`. When the tc Runtime instance starts, if it finds that the value of the `bind` attribute is different from or incompatible with the `java.rmi.server.hostname` Java system property, the instance will log a warning but will startup anyway and override the system property as described. If this causes problems in your particular environment, then you should change the value of the `bind` attribute to specify the actual hostname on which the tc Runtime runs.

  The monitoring application (such as Hyperic) that connects to the tc Runtime instance via JMX must specify a user and password to actually gain access. You configure these in the files pointed to by the `accessFile` and `passwordFile` attributes of the Listener. By default, the JMX user is `admin` with password `springsource`.

  By default, SSL is disabled; if you enable it by updating the `useSSL` attribute, you must then configure HQ with the trustStore and trustStorePassword. To set these values, add the following to the `agent.javaOpts` entry in each HQ Agent's `agent.properties` file:

```
agent.javaOpts=-Xmx128m -Djava.net.preferIPv4Stack=true -Dsun.net.inetaddr.ttl=60 \
-Djavax.net.ssl.trustStore=${full path to truststore} -Djavax.net.ssl.trustStorePassword=${password}
```

- **`<GlobalNamingResources>`**. Groups the global JNDI resources for this server instance that Web applications deployed to the server can use. In the preceding example, the `<Resource>` element defines the database used to load the users and roles from the *`CATALINA_BASE`*`/conf/tomcat-users.xml` file into an in-memory data structure. This resource is later referenced by the `<Engine>` XML element so that Web applications deployed to tc Runtime instances can query the database for the list of users and the roles to which the users are mapped, as well as update the file.

- **`<Service>`**. Groups one or more connectors, one or more executors, and a single engine. Connectors define a transport mechanism, such as HTTP, that clients use to send and receive messages to and from the associated service. A client can use many transports, which is why a `<Service>` element can have many `<Connector>` elements. The executors define thread pools that can be shared between components, such as connectors. The engine then defines how these requests and responses that the connector receives and sends are in turn handled by the tc Runtime instance; you can define only a single `<Engine>` element for any given `<Service>` element.

  The sample `server.xml` file above includes a single `<Connector>` for the HTTP transport, a single `<Executor>` that configures the thread pool used by the connector, and a single `<Engine>` as required.

- **`tomcatThreadPool`**. As defined by the `<Executor>` XML element, allows a maximum of 300 active threads. The minimum number of threads that are always kept alive is 50.

- **`<Connector>`**. Listens for HTTP requests at the 8080 TCP/IP port (as set by the `${bio.http.port}` variable in `catalina.properties`). The connector uses the thread pool defined by the `tomcatThreadPool` executor and ignores all other thread attributes. After accepting a connection from a client, the connector waits a maximum of 20000 milliseconds for a request URI, after which it times out. If this connector receives a request from the client that requires the SSL transport, the tc Runtime instance automatically redirects the request to port 8443. If the tc Runtime instance receives a connection request at a moment in time when all possible request processing threads are in use, the server puts the request on a queue; the `acceptCount` attribute specifies the maximum length of this queue (100) after which the server refuses all connection requests. Finally, the maximum number of HTTP requests that can be pipelined until the connection is closed by the server is 15, as specified by the `maxKeepAliveRequests` attribute.
- **`Catalina`**. Logical name of the engine. This name appears in all log and error messages so you can easily identify problems. The value of the `defaultHost` attribute is the name of a `<Host>` child element of `<Engine>`; this host processes requests directed to host names on this server.

  The `<Realm>` child element of `<Engine>` represents a database of users, passwords, and mapped roles used for authentication in this service. In the preceding sample, the realm simply references the `UserDatabase` resource, defined by the `<Resource>` child element of `<GlobalNamingResources>`.

  The `<Host>` child element represents a virtual host, which is an association of a network name for a server (such as `www.mycompany.com`) with the particular server on which Catalina is running. tc Runtime automatically deploys Web applications that are copied to the *`CATALINA_BASE`*`/webapps` directory while the tc Runtime instance is running and automatically deploys them when the server starts. The tc Runtime instance unpacks the Web applications into a directory hierarchy if they are deployed as WAR files. SpringSource tc Runtime parses any `context.xml` file contained in the `META-INF` directory of deployed applications. The `xmlValidation` attribute specifies that the tc Runtime instance does not validate XML files when parsing them, or in other words, it accepts invalid XML. The `xmlNamespaceAware` attribute specifies that tc Runtime does not take namespaces into account when reading XML files.

The preceding sample `server.xml` file contains typical elements and attribute values for a simple out-of-the-box tc Runtime configuration. However, you can configure many more elements and attributes in this file. For complete elements documentation about the tc Runtime `server.xml` file, see Apache Tomcat Configuration Reference.

Adding a System Properties Listener

tc Server includes a useful feature that allows you to configure tc Server and Java system properties through external properties files. Properties that you set using this method can be use to as replacement values in `server.xml`. The external properties files are also useful for setting application properties, instead of modifying the `setenv.sh` script to set them on the `java` command line with the `-D` flag. The properties are available to applications through `java.lang.System.getProperties()`.

The listener should be the first child of the `Server` element in the `server.xml` file, since XML is processed in the order it appears and properties must be set before they are referenced.

The following example specifies four properties files to be processed in sequence.

```
<Listener className="com.springsource.tcserver.properties.SystemProperties"
    file.1="${catalina.base}/conf/base.properties"
    file.3="${catalina.base}/conf/qa.properties"
    file.2="${catalina.base}/conf/dev.properties"
    file.4="${catalina.base}/conf/prod.properties"
    immutable="false"
    trigger="now"/>
```

There can be up to one hundred files, and they are processed in sequence by the numeric extension, not in the order they appear. In the example above, the `dev.properties` file is processed before the `qa.properties` file, even though they are not listed in that order.

The `immutable` attribute, `false` by default, determines if properties can be overridden. When `false`, the property value is set each time the key is encountered. If `immutable` is `true`, once a value is associated with a key it cannot be changed; later occurrences of the property are ignored. Whether `immutable` is set to `true` or `false`, a debug message is logged when an existing property is encountered.

If you specify a properties file that does not exist, a message is logged, but processing continues. This allows you to set up `system.xml` for different runtime environments by supplying only the appropriate properties files. In the example above, for example, if the `prod.properties` file is missing, the properties in the `base.properties`, `dev.properties`, and `qa.properties` files are processed, with any properties overridden in `qa.properties` taking precedence.

The presence of the `trigger` attribute causes the properties to be applied before parsing the remainder of the `server.xml` file. The value of the `trigger` attribute is ignored.

Setting Up a High-Concurrency JDBC Datasource

A datasource defines a pool of JDBC connections for a specific database using a URL, username, and so on. JDBC datasources make it easy for an application to access data in a database server.

Comparing the DBCP Datasource and the tc Runtime Datasource

In a tc Runtime instance, you can create the following two types of JDBC datasources:

- Database connection pool (DBCP) datasource
- tc Runtime datasource

The **DBCP datasource** is the standard datasource provided by tc Runtime; it uses the `org.apache.commons.dbcp` package. Although this datasource is adequate for simple applications, it is single-threaded, which means that in order to be thread-safe, the tc Runtime instance must lock the entire pool, even during query validation. Thus it is not suitable for highly concurrent environments. Additionally, it can be slow, which in turn can negatively affect the performance of Web applications.

The **tc Runtime datasource** includes all the functionality of the DBCP datasource, but adds additional features to support highly-concurrent environments and multiple core/cpu systems. The tc Runtime datasource typically performs much better than the DBCP datasource. Additional features include:

- Dynamic implementation of the interfaces, which means that the datasource supports the `java.sql` and `javax.sql` interfaces for your runtime environment (as long as your JDBC driver supports it), even when compiled with a lower version of the JDK.
- Validation intervals so that tc Runtime doesn't have to validate every single time the application uses the connection, which improves performance.
- Run-Once query, which is a configurable query that tc Runtime runs only once when the connection to the database is established. This is very useful to set up session settings that you want to exist during the entire time the connection is established.
- Ability to configure custom interceptors to enhance the functionality of the datasource. You can use interceptors to gather query stats, cache session states, reconnect the connection upon failures, retry queries, cache query results, and so on. The interceptors are dynamic and not tied to a JDK version of a `java.sql`/`javax.sql` interface.
- Asynchronous connection retrieval - you can queue your request for a connection and receive a Future<Connection> back.

Configuring the tc Runtime High-Concurrency JDBC Datasource

As with any tc Runtime resource, you configure the high-concurrency datasource (that is, the tc Runtime datasource) using a `<Resource>` child element of `<GlobalNamingResource>`. Most attributes are common to the standard DBCP and the tc Runtime datasources; however, the following new attributes apply only to the new tc Runtime datasource.

- initSQL
- jdbcInterceptors
- validationInterval
- jmxEnabled
- fairQueue

- useEquals

Use the `factory` attribute of the `<Resource>` element to specify the type of datasource:

- Set the `factory` attribute to `org.apache.tomcat.jdbc.pool.DataSourceFactory` to use the tc Runtime high-concurrency datasource. This is also the default value of the `factory` attribute for tc Runtime, so you will automatically use the high-concurrency datasource if you do not specify this attribute at all.
- Set the `factory` attribute to `org.apache.tomcat.dbcp.dbcp.BasicDataSourceFactory` to use the standard DBCP datasource.

IBM JVM USERS ONLY: If you are using an IBM JVM, see useEquals for important information.

The following table lists the attributes for configuring either the high-concurrency datasource or the standard DBCP datasource. Most attributes are valid for both of the datasources, but some are only valid for one datasource. These exceptions are noted in the table. The default values shown are for the high-concurrency datasource, which is the default datasource for tc Server. Default values for the DBCP datasource may be different. See the Apache DBCP documentation for details.

Table 4.1. Connection Pool Configuration Attributes

Attribute	Default	Description
username (*required*)		The username to pass to the JDBC driver to establish a connection with the database.
password (*required*)		The password to pass to the JDBC driver to establish a connection with the database.
url (*required*)		The connection URL to pass to the JDBC driver to establish a connection.
driverClassName (*required*)		The fully qualified Java class name of the JDBC driver to use. The driver must be accessible from the same classloader as `tomcat-jdbc.jar`
connectionProperties		Connection properties to send to the JDBC driver when establishing a new database connection. The syntax for this string is [propertyName=value;]* The "user" and "password" properties are passed explicitly, so do not include them here.
defaultAutoCommit	`true`	The default auto-commit state of connections created by this pool. If it is not set, the JDBC driver's default setting is active.
defaultReadOnly	driver default	The default read-only state of connections created by this pool. If not set, the `setReadOnly` method will not be called. (Some drivers do not support read only mode, for example Informix.)
defaultTransactionIsolation	driver default	The default TransactionIsolation state of connections created by this pool. One of the following: • `NONE` • `READ_COMMITTED` • `READ_UNCOMMITTED` • `REPEATABLE_READ`

Attribute	Default	Description
		• `SERIALIZABLE` (see Javadoc). If not set, the default is the JDBC driver's default.
defaultCatalog		The default catalog of connections created by this pool.
initialSize	`10`	The initial number of connections to create when the pool is started.
maxActive	`100`	The maximum number of active connections that can be allocated from this pool at the same time, or negative for no limit.
maxIdle	`maxActive` (100)	The maximum number of connections that should be kept in the pool at all times. Idle connections are checked periodically (if enabled) and connections that have been idle for longer than `minEvictableIdleTimeMillis` are released. See also `testWhileIdle`.
minIdle	`10`	The minimum number of established connections that should be kept in the pool at all times. The connection pool can shrink below this number if validation queries fail. The default value is derived from `initialSize`.
maxWait	`30000`	The maximum milliseconds a pool with no available connections will wait for a connection to be returned before throwing an exception, or `-1` to wait indefinitely.
validationQuery		The SQL query to use to validate connections from this pool before returning them to the caller. If specified, the query *must* be an SQL SELECT statement that returns at least one row.
testOnBorrow	`false`	Indicates whether objects are validated before borrowed from the pool. If the object fails to validate, it is dropped from the pool, and an attempt is made to borrow another. A `true` value has no effect unless the `validationQuery` parameter is set to a non-null string.
testOnReturn	`false`	Indicates if objects are validated before they are returned to the pool. A true value has no effect unless the `validationQuery` parameter is set to a non-null string.
testWhileIdle	`false`	Indicates whether objects are validated by the idle object evictor (if any). If an object fails to validate, it is dropped from the pool. A `true` value has no effect unless the `validationQuery` parameter is set to a non-null

Attribute	Default	Description
		string. This parameter must be set to activate the pool test/cleaner thread.
timeBetweenEvictionRunsMillis	`5000`	The number of milliseconds to sleep between runs of the idle object evictor thread. The thread checks for idle, abandoned connections and validates idle connections. The value should not be set below 1 second (1000).
numTestsPerEvictionRun		Not used by the Tomcat JDBC pool. The number of objects to examine during each run of the idle object evictor thread, if any.
minEvictableIdleTimeMillis	`60000`	The minimum amount of time an object may sit idle in the pool before it is eligible for eviction by the idle object evictor, if any.
connectionInitSqls	`null`	A Collection of SQL statements used to initialize physical connections when they are first created. These statements are executed only once, when the connection factory creates the connection. DBCP Versions 1.3 and 1.4 of incorrectly use "`initConnectionSqls`" as the name of this property for JNDI object factory configuration. Until 1.3.1/1.4.1 are released, "`initConnectionSqls`" must be used as the name for this property when using BasicDataSourceFactory to create BasicDataSource instances via JNDI.
poolPreparedStatements	`false`	This property is not used.
maxOpenPreparedStatements		This property is not used.
accessToUnderlyingConnectionAllowed		Not used. Access can be achieved by calling `unwrap` on the pooled connection. See `javax.sql.DataSource` interface, or call `getConnection` through reflection, or cast the object as `javax.sql.PooledConnection`.
removeAbandoned	`false`	Set to `true` to remove abandoned connections if they exceed the `removeAbandonedTimeout`. Setting this to `true` can recover database connections from poorly written applications that fail to close a connection. A connection is considered abandoned and eligible for removal if it has been idle longer than the `removeAbandonedTimeout`.
removeAbandonedTimeout	`60`	Timeout in seconds before an abandoned connection can be removed. The value should be set to the longest running query your applications might have.
logAbandoned	`false`	Set to `true` to log stack traces for application code that abandons a Connection. Logging an abandoned Connection adds overhead for every Connection open because a stack trace has to be generated.
initSQL (*high concurrency JDBC datasource only*)		Initial SQL statement that is run only when a connection is first created. Use this feature to set up session

Attribute	Default	Description
		settings that should exist during the entire time the connection is established.
jdbcInterceptors (*high concurrency JDBC datasource only*)	`null`	Semicolon-separated list of classnames extending `org.apache.tomcat.jdbc.pool.JdbcInterceptor`. tc Runtime inserts interceptors in the chain of operations on the `java.sql.Connection` object. **Warning**: Be sure you do not include any white space (such as spaces or tabs) in the value of this attribute, or the classes will not be found. Predefined interceptors: • `org.apache.tomcat.jdbc.pool.interceptor.ConnectionState` - keeps track of auto commit, read only, catalog and transaction isolation level. • `org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer` - keeps track of opened statements, and closes them when the connection is returned to the pool.
validationInterval (*high concurrency JDBC datasource only*)	`30000` (30 seconds)	Number of milliseconds tc Runtime waits before running a validation check to ensure that the JDBC connection is still valid. A connection that has been validated within this interval is not revalidated. Running validation checks too frequently can slow performance.
jmxEnabled (*high concurrency JDBC datasource only*)	`true`	Specifies whether the connection pool is registered with the JMX server.
fairQueue (*high concurrency JDBC datasource only*)	`true`	Specifies whether calls to `getConnection()` should be treated fairly in a true FIFO (first in, first out) fashion. This ensures that threads receive connections in the order they arrive. It uses the `org.apache.tomcat.jdbc.pool.FairBlockingQueue` implementation to manage the list of idle connections. This feature must be enabled (that is, set the attribute to `true`) to use the asynchronous connection retrieval feature, which is the ability to queue your connection request. **Note**: When `fairQueue=true` and the operating system is Linux, there is a very large performance difference in how locks and lock waiting is implemented. To disable this Linux-specific behavior and still use the fair queue, add the property `org.apache.tomcat.jdbc.pool.FairBlockingQueue.ignoreOS=true` to your system properties before the connection pool classes are loaded.
abandonWhenPercentageFull	`0`	Connections that have been abandoned (timed out) are not closed and reported up unless the number of connections in use is above the percentage defined

Attribute	Default	Description
		by this parameter. The value should be between 0 and 100. The default value is 0, which implies that connections are eligible for closure as soon as `removeAbandonedTimeout` has been reached.
maxAge	`0`	Time in milliseconds to keep this connection. When a connection is returned to the pool, the pool checks to see if the `now -time-when-connected > maxAge` has been reached, and if so, it closes the connection rather than returning it to the pool. The default value is 0, which implies that connections are left open and no age check is done upon returning the connection to the pool.
useEquals (*high concurrency JDBC datasource only*)	`false`	Specifies whether the ProxyConnection class should use String.equals() instead of "==" when comparing method names. Does not apply to added interceptors as those are configured individually. **NOTE FOR IBM JVM USERS**: If you are running tc Runtime on a platform that uses the IBM JVM (such as AIX), always set the `useEquals` attribute to `true` if you want a high-concurrency connection pool to work correctly. IBM JVMs do not use String literal pools for method names, which means you always want to use `String.equals()` when comparing method names in this case.
suspectTimeout	`0`	Timeout value in seconds. Similar to `removeAbandonedTimeout` but instead of treating the connection as abandoned and potentially closing the connection, this simply logs the warning if `logAbandoned` is set to true. If this value is equal or less than 0, no suspect checking will be performed. Suspect checking only takes place if the timeout value is larger than 0 and the connection was not abandoned or if abandon check is disabled. If a connection is suspect a WARN message is logged and a JMX notification is sent once.
alternateUsernameAllowed	`false`	For performance reasons, by default the JDBC pool ignores the `DataSource.getConnection(username, password)` call and returns a previously pooled connection established using the globally configured properties `username` and `password`. The pool can, however, be used with different credentials each time a connection is used. If you request a connection with the credentials user1/password1 and the connection was previously connected using user2/password2, the connection is closed, and reopened with the requested credentials. This way, the pool size is still managed on a global level, and not on a per schema level. To enable the functionality described in `DataSource.getConnection(username,password)`,

Attribute	Default	Description
		set the property `alternateUsernameAllowed` to `true`.

The following `server.xml` snippet shows how to configure the high-concurrency JDBC datasource for your tc Runtime instance. You can add this datasource to a tc Server Runtime instance by including the diagnostics template in the `tcruntime-instance create` command line. For an explanation of the following example, see Description of the High Concurrency JDBC Datasource.

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="-1" shutdown="SHUTDOWN">

 ...

  <GlobalNamingResources>

    <Resource name="jdbc/TestDB"
              auth="Container"
              type="javax.sql.DataSource"
              username="root"
              password="password"
              driverClassName="com.mysql.jdbc.Driver"
              url="jdbc:mysql://localhost:3306/mysql?autoReconnect=true"

              testWhileIdle="true"
              testOnBorrow="true"
              testOnReturn="false"
              validationQuery="SELECT 1"
              validationInterval="30000"
              timeBetweenEvictionRunsMillis="5000"
              maxActive="100"
              minIdle="10"
              maxWait="10000"
              initialSize="10"
              removeAbandonedTimeout="60"
              removeAbandoned="true"
              logAbandoned="true"
              minEvictableIdleTimeMillis="30000"
              jmxEnabled="true"
              jdbcInterceptors="org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;
                 org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer;
                 org.apache.tomcat.jdbc.pool.interceptor.SlowQueryReportJmx(threshold=10000)"/>

  </GlobalNamingResources>
  ...
  <Service name="Catalina">
  ...
  </Service>
</Server>
```

Description of the High Concurrency JDBC Datasource

In the preceding sample `server.xml`, the `<Resource>` element does not include a `factory` attribute, which means that the resource is using the default value, `org.apache.tomcat.jdbc.pool.DataSourceFactory`, the tc Runtime high-concurrency datasource. The `<Resource>` element attributes in the example function as follows:

- **`name`**. JNDI name of this JDBC resource is `jdbc/TestDB`.
- **`auth`**. The container signs on to the resource manager on behalf of the application.
- **`type`**. This resource is a JDBC datasource.
- **`username, password`**. Name and password of the database user who connects to the database.

- **`driverClassName`**. tc Runtime should use the `com.mysql.jdbc.Driver` JDBC driver to connect to the database, in this case a MySQL database.
- **`url`**. URL that the JDBC driver uses to connect to a MySQL database. The format of this URL is specified by JDBC.
- **`testXXX` attributes**. tc Runtime validates objects before it borrows them from the connection pool and those objects are validated by the idle object evictor, but that tc Runtime does *not* validate objects when it returns them to the pool.
- **`validationQuery`**. tc Runtime runs the very simple SQL query `SELECT 1` when it validates connections from the pool before returning a connection to a user upon request. Because this query should always return a value, if it returns an exception then tc Runtime knows there is a problem with the connection.
- **`validationInterval`**. tc Runtime waits at least 30 seconds before running a validation query.
- **`timeBetweenEvictionRunsMillis`**. tc Runtime sleeps 5000 milliseconds between runs of the idle connection validation/cleaner thread.
- **`maxActive`**. tc Runtime allocates a maximum of 100 active connections from this pool at the same time
- **`minIdle`**. tc Runtime keeps a minimum of 10 established connections in the pool at all times.
- **`maxWait`**. Where no connections are available, tc Runtime waits a maximum of 10,000 milliseconds for a connection to be returned before throwing an exception.
- **`initialSize`**. tc Runtime creates 10 connections when it initially starts the connection pool.
- **`removeAbandonedTimeout`**. tc Runtime waits 60 seconds before it removes an abandoned, but still in use, connection.
- **`removeAbandoned`**. tc Runtime removes abandoned connections after they have been idle for the `removeAbandonedTimeout` amount of time.
- **`logAbandoned`**. tc Runtime flags to log stack traces for application code that abandoned a Connection.
- **`minEvictableIdleTimeMillis`**. Minimum amount of time an object may sit idle in the pool before it is eligible for eviction on this tc Runtime is 30,000 milliseconds.
- **`jmxEnabled`**. This tc Runtime can be monitored using JMX. You must set this attribute to true if you want HQ to monitor the resource.
- **`jdbcInterceptors`**. List of interceptor classes associated with this datasource.

For complete documentation about the tc Runtime `server.xml` file and all the possible XML elements you can include, see Apache Tomcat Configuration Reference.

Configuring SSL

When you configure SSL (secure socket layer) for tc Runtime, use one of the following frameworks:

- The SSL framework provided by Java SE Security (JSSE), which is included in the JDK and available to you by default.
- OpenSSL, which is what tc Runtime uses when you use the Apache Portable Runtime (APR) library. APR libraries provide a predictable and consistent interface to underlying platform-specific implementations. Use of APR provides superior scalability, performance, and better integration with native server technologies. The APR libraries are usually installed by default on Unix versions of tc Runtime; you must download the libraries for other platforms.

tc Server includes templates that make it simple to configure a tc Runtime instance with SSL when you create the instance. Choose one of the SSL templates—`apr-ssl`, `bio-ssl`, or `nio-ssl`—based on the type of I/O you want to use. You can

also use the `jmx-ssl` template to configure SSL for the JMX connector. See "Creating a Runtime Instance with a Template" in *Getting Started with vFabric tc Server* for help using the templates.

The following snippet of a sample `server.xml` file is equivalent to using the `bio-ssl` template to create an instance. It builds on the simple out-of-the-box configuration file by adding SSL capabilities to tc Runtime so that users can make a secure connection to deployed applications over HTTPS. Add SSL to tc Runtime by adding a `<Connector>` child XML element to the `<Service>` element, alongside the existing connector that configures the non-SSL-enabled HTTP port. This new connector is configured for a different TCP/IP port than the regular non-SSL port; users who specify the SSL port enable SSL handshake, encryption, and decryption during their connection.

See Description of the SSL Connector for detailed information about this new `<Connector>` element. This XML snippet uses the SSL framework provided by JSSE; for an example of a connector that uses APR, see Using an APR Connector to Configure SSL.

```
<Connector
        executor="tomcatThreadPool"
        port="8443"
        protocol="HTTP/1.1"
        connectionTimeout="20000"
        redirectPort="8443"
        acceptCount="100"
        maxKeepAliveRequests="15"
        keystoreFile="${catalina.base}/conf/tcserver.keystore"
        keystorePass="changeme"
        keyAlias="tcserver"
        SSLEnabled="true"
        scheme="https"
        secure="true"/>
```

Description of the SSL Connector

The preceding snippet of `server.xml` describes a new SSL-enabled `<Connector>` that uses the JSSE libraries included in the JDK. The attribute values in the example function as follows:

- **`executor`**, **`protocol`**, **`connectionTimeout`**, **`maxKeepAliveRequests`**, **`acceptCount`**. Same attributes as those of the basic HTTP connector. Although this connector is used for HTTPS connections, you still set protocol to `HTTP/1.1`; other attributes specify an SSL-enabled connection.
- **`port`**. The TCP/IP port that users specify as the secure connection port is 8443. Set the value of the `redirectPort` attribute of your non-SSL connectors to this value to ensure that users who require a secure connection are redirected to the secure port, even if they initially start at the non-secure port.
- **`SSLEnabled`**. Specifies that SSL is enabled for this connector.
- **`secure`**. If set to `true`, ensures that a call to `request.isSecure()` from the connecting client always returns `true`. Default is `false`.
- **`scheme`**. If set to `https`, ensures that a call to `request.getScheme()` from the connecting client returns `https` when clients use this connector. The default value of this attribute is `http`.
- **`keystoreFile`**. Name of the file that contains the server's private key and public certificate used in the SSL handshake, encryption, and decryption. You use an alias and password to access this information. In the example, this file is called `tcserver.keystore` and is located in the same directory as the standard tc Runtime configuration files: *`CATALINA_BASE`*`/conf`.

  See Creating a Simple Keystore File for information about creating they keystore file.
- **`keyAlias` and `keystorePass`**. Alias and password to access the keystore specified by the `keystoreFile` attribute. In the example, the alias is `tcserver` and the password is `changeme`.

For complete documentation about configuring SSL for tc Runtime servers, see SSL Configuration HOW-TO.

For general documentation about the tc Runtime `server.xml` file and all the possible XML elements you can include, see Apache Tomcat Configuration Reference.

Using an APR Connector to Configure SSL

When you use an APR connector to specify a secure tc Runtime port, tc Runtime uses the OpenSSL framework, meaning that you will be using an SSL engine native to your platform rather than the one included in JSSE. Use the `apr-ssl` template with `tcruntime-instance` script to create a tc Runtime instance configured to use OpenSSL. This section describes configuration changes that are made for you when you use the `apr-ssl` template.

Before configuring the connector, add the APR listener to your `server.xml` file in the `<Listener>` element:

```
<Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
```

The preceding element initializes the native SSL engine. The `<Connector>` element enables the use of this engine in the connector with the `SSLEnabled` attribute, as shown in the following sample:

```
<Connector
           executor="tomcatThreadPool"
           port="8443"
           protocol="org.apache.coyote.http11.Http11AprProtocol"
           connectionTimeout="20000"
           redirectPort="8443"
           acceptCount="100"
           maxKeepAliveRequests="15"
           SSLCertificateFile="${catalina.base}/conf/tcserver.crt"
           SSLCertificateKeyFile="${catalina.base}/conf/tcserver.key"
           SSLPassword="changeme"
           SSLEnabled="true"
           scheme="https"
           secure="true"/>
```

This connector configuration is similar to the one that uses the JSSE SSL libraries, as described in Description of the SSL Connector, but with the following differences, mostly having to do with the configuration of OpenSSL:

- The value of the `protocol` attribute is `org.apache.coyote.http11.Http11AprProtocol`, rather than `HTTP/1.1`, to indicate that the connector is using the APR libraries.
- The `SSLCertificateFile` attribute specifies the name of the file that contains the server certificate. The format is PEM-encoded. In the example, this file is called `tcserver.crt`, and is located in the conf directory under the `CATALINA_BASE` directory in which your tc Runtime instance is installed.
- The `SSLCertificateKeyFile` attribute specifies the name of the file that contains the server private key. The format is PEM-encoded. In the example, the file is called `tcserver.key` and is located in the same directory as the certificate file.
- The `SSLPassword` attribute specifies the password for the encrypted private key in the file pointed to by the `SSLCertificateKeyFile` attribute.
- The preceding attributes are used instead of the `keystoreFile`, `keystorePass`, and `keyAlias` attributes of the JSSE secure connector.

See Apache Portable Runtime (APR) based native library for Tomcat for additional information about APR and how to configure an APR HTTPS connector.

Creating a Simple Keystore File For Both SSL and OpenSSL

Configuring SSL or OpenSSL for tc Runtime requires a keystore that contains certificates and public keys. The certificate identifies the company or organization and verifies the public key. Clients that connect to tc Runtime use the public key to encrypt and decrypt data transferred over the wire.

Warning: The tc Server installation includes sample keystores, key files, and certificates. Note that they are provided *only* to aid testing and debugging; they are not for production use. You must replace them with your own generated keystores and certificates before moving to a production system.

Your keystore can use self-signed certificates that, although they do not guarantee authenticity, can be used by both the clients and server to encrypt and decrypt data. Use the `keytool` JDK tool to create a keystore that contains self-signed certificates, as shown below. If you require an authentic, verified certificate, purchase one from a well-known Certificate Authority such as VeriSign. Then use the `keytool` tool to import the certificate into your keystore.

For complete documentation about creating keystores, in particular how to import a fully authentic certificate into an existing keystore, see SSL Configuration HOW-TO.

To use the `keytool` tool to create a keystore that contains self-signed certificates:

```
prompt> $JAVA_HOME/bin/keytool -genkey -alias alias -keyalg RSA -keystore keystore
```

Be sure that the value of the `-alias` option matches the value of the `keyAlias` attribute of the secure Connector you configured in the `server.xml` file, as described in the preceding section. Similarly, the value of the `-keystore` option should match the value of the `keystoreFile` attribute. For example:

```
prompt> $JAVA_HOME/bin/keytool -genkey -alias tcserver -keyalg RSA -keystore \
    /apache/tomcat6/conf/tcserver.keystore
```

In the example, `CATALINA_BASE` is assumed to be `/apache/tomcat6`.

A message asks for a keystore password; this password must match the `keystorePass` attribute of the `<Connector>` element that configures the secure port, as described in the preceding section. After prompts for information about your company, a message requests the password for the keystore alias; set this to the same value as the keystore password.

Using the Apache Portable Runtime (APR)

The Apache Portable Runtime (APR) is a set of libraries and APIs that map directly to your underlying operating system. tc Runtime can use APR to provide additional scalability and performance because of high-quality integration with native server technologies. APR provides access to advanced IO functionality (such as sendfile, epoll and OpenSSL), OS level functionality (random number generation, system status, etc.), and native process handling (shared memory, NT pipes and Unix sockets).

The APR libraries are automatically installed in most Unix platforms, although you need to compile the Java Native Interface (JNI) wrappers. For other platforms, such as Windows, you must download and install the libraries. See Apache Portable Runtime (APR) Native Library for Tomcat.

Add the APR libraries to the `LD_LIBRARY_PATH` (Unix) or `PATH` (Windows) environment variable used by the tc Runtime process so that tc Runtime can access the libraries when it runs.

The following sample `server.xml` file shows how to configure tc Runtime to use APR. The file builds on the simple out-of-the-box configuration described in Simple tc Runtime Configuration.

See Comparing the APR-Enabled server.xml File with Out-of-the-Box server.xml for information about how the two files differ.

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="-1" shutdown="SHUTDOWN">

  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <Listener className="org.apache.catalina.core.JasperListener" />
  <Listener className="org.apache.catalina.mbeans.ServerLifecycleListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />

  <GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
```

```
  </GlobalNamingResources>

  <Service name="Catalina">

    <Executor name="tomcatThreadPool" namePrefix="tomcat-http--" maxThreads="300" minSpareThreads="50"/>

    <Connector
               executor="tomcatThreadPool"
               port="8080"
               protocol="org.apache.coyote.http11.Http11AprProtocol"
               connectionTimeout="20000"
               redirectPort="8443"
               acceptCount="100"
               maxKeepAliveRequests="15"/>

    <Connector
               executor="tomcatThreadPool"
               port="8443"
               protocol="org.apache.coyote.http11.Http11AprProtocol"
               connectionTimeout="20000"
               redirectPort="8443"
               acceptCount="100"
               maxKeepAliveRequests="15"
               SSLCertificateFile="${catalina.base}/conf/tcserver.crt"
               SSLCertificateKeyFile="${catalina.base}/conf/tcserver.key"
               SSLPassword="changeme"
               SSLEnabled="true"
               scheme="https"
               secure="true"/>


    <Engine name="Catalina" defaultHost="localhost">

      <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
             resourceName="UserDatabase"/>

      <Host name="localhost"  appBase="webapps"
            unpackWARs="true" autoDeploy="true" deployOnStartup="true" deployXML="true"
            xmlValidation="false" xmlNamespaceAware="false">
      </Host>
    </Engine>
  </Service>
</Server>
```

Comparing the APR-Enabled server.xml File with Out-of-the-Box server.xml

In the preceding sample `server.xml`, most of the configuration is the same as the non-APR enabled `server.xml` file except for the following:

- The preceding `server.xml` file includes an additional APR-specific listener, implemented by the `org.apache.catalina.core.AprLifecycleListener` class. The `SSLEngine="on"` attribute enables the native SSL engine, rather than the JSEE engine provided by the JDK.
- The `protocol="org.apache.coyote.http11.Http11AprProtocol"` attribute of the `<Connector>` elements specify that the two HTTP connectors (with and without SSL enabled) both use the native HTTP protocol implementation.

See Configuring SSL for details about configuring the native SSL connector.

For complete documentation about the tc Runtime `server.xml` file and all the possible XML elements you can include, see Apache Tomcat Configuration Reference.

Configuring Logging for tc Runtime

As with standard Apache Tomcat, SpringSource tc Runtime uses Commons Logging throughout its internal code. This allows you to choose a logging configuration that suits your needs, such as `java.util.logging` (configured by default) or `log4j`.

Commons Logging provides tc Runtime with the ability to log hierarchically across various log levels without needing to rely on a particular logging implementation.

The sections that follow summarize the basic information in the standard Apache Tomcat logging documentation (see Logging in Tomcat). These sections also describe the *additional* logging features of tc Runtime as compared to the default logging in Apache Tomcat, such as asynchronous logging.

- Configuring the JULI Implementation of java.util.logging
- Logging Levels for java.util.logging
- Configuring Asynchronous Logging
- Configuring log4j
- Updating Logging Parameters Dynamically

Configuring the JULI Implementation of java.util.logging

SpringSource tc Runtime provides its own implementation of `java.util.logging` called JULI that addresses a major limitation of the JDK implementation: the inability to configure per-Web application logging. The JULI implementation is the default logging framework in tc Runtime.

Note: It is assumed that you are already familiar with the basic `java.util.logging` facility provided by the JDK. If you are not, see:

- Java Logging Overview
- Package java.util.logging

With the JULI implementation, you can configure logging at a variety of levels:

- Globally for the entire JVM used by tc Runtime by updating the standard `logging.properties` file of the JDK, typically located in the `JAVA_HOME/jre/lib` directory.
- Per-tc Runtime instance by updating the `logging.properties` file located in the `CATALINA_BASE/conf` directory of the tc Runtime instance.
- Per-Web application by adding a `logging.properties` file in the `WEB-INF/classes` directory of the Web application deployed to the tc Runtime instance.

At each level you use a `logging.properties` file to configure logging; the level that the file configures is based on the location of the file. You can also configure logging programmatically, although this document does not discuss this method. The `logging.properties` files for the tc Runtime instance or Web application, however, support extended constructs that allow more freedom to define handlers and assign them to loggers. The differences are described later in this section.

The default tc Runtime `logging.properties` file, located in `CATALINA_BASE/conf` of your server instance, specifies two types of handlers: `ConsoleHandler` for routing logging to *stdout* and `FileHandler` for writing long messages to a file. You can set the log level of each handler to standard `java.util.logging` levels, such as SEVERE or WARNING; see Logging Levels for java.util.logging for the full list.

The default log level setting in the JDK `logging.properties` file is set to INFO. You can also target specific packages from which to collect logging and specify the level of logging you want. For example, to set debugging from the entire tc Runtime instance, add the following to the `CATALINA_BASE/conf/logging.properties` file:

```
org.apache.catalina.level=FINEST
```

If you set the preceding log level, also set the `ConsoleHandler` level to collect this threshold, or in other words, be at a level higher than the overall tc Runtime level.

When you configure the `logging.properties` file for the tc Runtime instance or Web application, you use a similar configuration as that of the JDK `logging.properties` file. You can also specify additional extensions to allow better flexibility in assigning loggers. Use the following guidelines:

- As in Java 6.0, declare the list of handlers using `handlers`.
- As in Java 6.0, loggers define a list of handlers using the `loggerName.handlers` property.
- You define the set of handlers for the root logger using the `.handlers` property; note that there is no logger name.
- By default, loggers do not delegate to their parent if they have associated handlers. You can change this behavior for a particular logger using the `loggerName.useParentHandlers` property, which accepts a boolean value (`true` or `false`).
- As in Java 6.0, use the `handlerName.level` property to specify the level of logging you want for a given handler. See Logging Levels for java.util.logging for all the available log levels.
- You can add a prefix to handler names by specifying the `handlerName.prefix` property. In this case, tc Runtime can instantiate multiple handlers from a single class. A prefix is a String that starts with a digit and ends with '.'. For example, `22foobar.` is a valid prefix. The default prefix, if you do not specify one for a particular handler, is `juli.`.
- Similarly, you can also add a suffix to handler names with the `handlerName.suffix` property. The default suffix, if you do not specify one for a particular handler, is `.log`.
- Specify the directory to which a file handler writes its log files using the `handlerName.directory` property; the default value is `logs`. You can use the `${catalina.base}` variable to point to a `CATALINA_BASE` directory of your tc Runtime instance.
- A tc Runtime instance buffers logging using a default buffer size of 8192 bytes. If you want to reduce the disk access frequency and write larger chunks of data to a log each time, increase the buffer size of a handler by using the `handlerName.bufferSize` property.
- System property replacement for property values expressed using the format `${`*`systemPropertyName`*`}`.

The following example shows a `CATALINA_BASE/conf/logging.properties` file for a tc Runtime instance. It shows how to use the `level`, `prefix`, `directory`, and `bufferSize` properties for a variety of `FileHandlers`:

```
handlers = 1catalina.org.apache.juli.FileHandler, \
           2localhost.org.apache.juli.FileHandler, \
           3manager.org.apache.juli.FileHandler, \
           4admin.org.apache.juli.FileHandler, \
           java.util.logging.ConsoleHandler

.handlers = 1catalina.org.apache.juli.FileHandler, java.util.logging.ConsoleHandler

############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################

1catalina.org.apache.juli.FileHandler.level = FINE
1catalina.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
1catalina.org.apache.juli.FileHandler.prefix = catalina.

2localhost.org.apache.juli.FileHandler.level = FINE
2localhost.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
2localhost.org.apache.juli.FileHandler.prefix = localhost.

3manager.org.apache.juli.FileHandler.level = FINE
3manager.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
3manager.org.apache.juli.FileHandler.prefix = manager.
```

```
4admin.org.apache.juli.FileHandler.level = FINE
4admin.org.apache.juli.FileHandler.directory = ${catalina.base}/logs
4admin.org.apache.juli.FileHandler.prefix = admin.
4admin.org.apache.juli.FileHandler.bufferSize = 16384

java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter


############################################################
# Facility specific properties.
# Provides extra control for each logger.
############################################################

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].level = INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].handlers = \
   2localhost.org.apache.juli.FileHandler

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].level = INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/manager].handlers = \
   3manager.org.apache.juli.FileHandler

org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/admin].level = INFO
org.apache.catalina.core.ContainerBase.[Catalina].[localhost].[/admin].handlers = \
   4admin.org.apache.juli.FileHandler
```

The following example shows a `WEB-INF/classes/logging.properties` file for a specific Web application. The properties file configures a `ConsoleHandler` to route messages to *stdout*. It also configures a `FileHandler` that prints log messages at the FINE level to the `CATALINA_BASE/logs/servlet-examples.log` file:

```
handlers = org.apache.juli.FileHandler, java.util.logging.ConsoleHandler

############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################

org.apache.juli.FileHandler.level = FINE
org.apache.juli.FileHandler.directory = ${catalina.base}/logs
org.apache.juli.FileHandler.prefix = servlet-examples.

java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

Logging Levels for java.util.logging

The following table lists the standard log levels that you can set in the various `logging.properties` files, with the highest level listed first down to the lowest level (OFF). Enabling logging at a given level also enables logging at all higher levels. In general, the lower level of logging you enable, the more data that tc Runtime writes to the log files, so be careful when setting the logging level very low.

Table 4.2. Standard Log Levels

Level	Description
ALL	Logs all messages.
SEVERE	Logs messages indicating a serious failure. SEVERE messages describe events that prevent normal program execution. They should be completely intelligible to end users and to system administrators.
WARNING	Logs message indicating a potential problem. WARNING messages describe events that interest end users or system managers, or that indicate potential problems.

Level	Description
INFO	Logs informational messages. Typically, INFO messages are written to the console or its equivalent, which means that the INFO level should only be used for reasonably significant messages that will make sense to end users and system admins.
CONFIG	Logs static configuration messages. CONFIG messages provide a variety of static configuration information, to assist in debugging problems that may be associated with particular configurations; for example, the CPU type, the graphics depth, the GUI look-and-feel, and so on.
FINE	Logs relatively detailed tracing. FINE messages might include things like minor (recoverable) failures. Issues indicating potential performance problems are also worth logging as FINE. In general the FINE level should be used for information that will be broadly interesting to developers who do not have a specialized interest in the specific subsystem. The exact meaning of the three levels vary among subsystems, but in general, use FINEST for the most voluminous detailed output, FINER for somewhat less detailed output, and FINE for the lowest volume and most important messages.
FINER	See FINE for FINE, FINER, and FINEST descriptions. FINER indicates a fairly detailed tracing message. By default logging calls for entering, returning, or throwing an exception are traced at this level.
FINEST	See FINE for FINE, FINER, and FINEST descriptions. FINEST indicates a highly detailed tracing message.
OFF	Turns off logging.

Configuring Asynchronous Logging

By default, the tc Runtime thread that handles incoming Web requests is the same thread that writes to the log file, such as `catalina.out`. Thus if a resource issue causes the thread writing to the log file to block, the incoming Web request is also blocked until the thread is able to finish writing to the log file. Depending on your environment, this problem can affect the performance of incoming Web requests.

Asynchronous logging addresses this potential performance problem with a separate thread to write to the log file. The Web request thread does not have to wait for the write to the log file to complete, and incoming request from users (or Web services) are not affected by internal resource issues.

Another advantage of asynchronous logging is that you can configure a more verbose log level without affecting the performance of the incoming requests, because even though a lot of information is being written to the log file, it is being written by a different thread from the one handling the incoming requests.

Asynchronous logging is available only if your tc Runtime instance uses version 1.6 of the JDK/JRE. Also, asynchronous logging is available only with the `java.util.logging` logging configuration, and not with `log4j`.

To configure asynchronous logging for a tc Runtime instance:

1. Edit the *`CATALINA_BASE`*`/conf/logging.properties` file, where *`CATALINA_BASE`* refers to the root directory of your tc Runtime instance, such as `/var/opt/vmware/vfabric-tc-server-standard/myserver`. Change every instance of `FileHandler` in the file to `AsyncFileHandler`.

   The following snippet shows how the first few non-commented lines of the file will look after the substitution:

```
handlers = 1catalina.org.apache.juli.AsyncFileHandler, \
           2localhost.org.apache.juli.AsyncFileHandler, \
           3manager.org.apache.juli.AsyncFileHandler, \
           4host-manager.org.apache.juli.AsyncFileHandler, \
           java.util.logging.ConsoleHandler

.handlers = 1catalina.org.apache.juli.AsyncFileHandler
```

```
############################################################
# Handler specific properties.
# Describes specific configuration info for Handlers.
############################################################

1catalina.org.apache.juli.AsyncFileHandler.level = FINE
1catalina.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
1catalina.org.apache.juli.AsyncFileHandler.prefix = catalina.

2localhost.org.apache.juli.AsyncFileHandler.level = FINE
2localhost.org.apache.juli.AsyncFileHandler.directory = ${catalina.base}/logs
2localhost.org.apache.juli.AsyncFileHandler.prefix = localhost.
...
```

2. Optionally configure how asynchronous logging behaves by setting one or more of the system properties listed in the properties table. Each property has a default value so you only need to set them if their default values are not adequate.

   Set the properties in the `CATALINA_BASE/bin/setenv.sh` (Unix) or `CATALINA_BASE/bin/setenv.bat` (Windows) file by updating the `APPLICATION_OPTS` variable. Use the standard `-D` option for each system property you set. The following example shows how to set two of the properties on Unix:

   ```
   APPLICATION_OPTS=-Dorg.apache.juli.AsyncOverflowDropType=1 -Dorg.apache.juli.AsyncMaxRecordCount=10000
   ```

3. Restart your tc Runtime instance for the changes to take effect.

Asynchronous Logging System Properties

The following table lists the system properties you can set to configure the asynchronous logging feature of tc Runtime.

Table 4.3. Asynchronous Logging System Properties

Property Name	Description	Default Value
org.apache.juli.AsyncOverflowDropType	Specifies the action taken by tc Runtime when the memory limit of records has been reached. You can set this property to one of the following values: • `1` : tc Runtime drops, and does not log, the record that caused the overflow. • `2` : tc Runtime drops the record that is next in line to be logged to make room for the latest record on the queue. • `3` : tc Runtime suspends the thread while the queue empties out and flushes the entries to the write buffer. • `4` : tc Runtime drops the current log entry.	1
org.apache.juli.AsyncMaxRecordCount	Max number of log records that the asynchronous logger keeps in memory. When this limit is reached and a new record	10000

Property Name	Description	Default Value
	is being logged by the JULI framework, the system takes an action based on the value of the `org.apache.juli.AsyncOverflowDropType` property. This number represents the global number of records, not on a per handler basis.	
org.apache.juli.AsyncLoggerPollInterval	Poll interval (in milliseconds) of the asynchronous logger thread. If the log queue is empty, the asynchronous logging thread issues a `poll(`*`poll_interval`*`)` call in order to not wake up to often.	1000

Configuring log4j

The following steps describe how to configure basic `log4j`, rather than `java.util.logging`, as the logging implementation for a given tc Runtime instance. The text after the basic procedure describes how to further customize the log4j configuration.

1. Under the `CATALINA_BASE` directory, create the following directories if they do not already exist:
   - `CATALINA_BASE/lib`
   - `CATALINA_BASE/bin`
2. Create a file called `log4j.properties` in the `CATALINA_BASE/lib` directory of your tc Runtime instance.
3. Add the following properties to the `log4j.properties` file:

```
log4j.rootLogger=INFO, R
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=${catalina.base}/logs/tomcat.log
log4j.appender.R.MaxFileSize=10MB
log4j.appender.R.MaxBackupIndex=10
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

4. Download log4j (version 1.2 or later) and place the `log4j.jar` file in the `CATALINA_BASE/lib` directory of your tc Runtime instance.
5. Copy the `CATALINA_BASE/bin/extras/tomcat-juli.jar` file provided with tc Server to the `CATALINA_BASE/bin` directory of your tc Runtime instance.
6. Copy the `CATALINA_HOME/bin/extras/tomcat-juli-adpaters.jar` file provided with tc Server to the `CATALINA_BASE/lib` directory of your tc Runtime instance.
7. Delete the `CATALINA_BASE/conf/logging.properties` file to prevent `java.util.properties` from generating zero-length log files.

Specifying Included Packages With log4j Logging

SpringSource recommends that you configure the specific packages that you want to include in the logging. Because tc Runtime defines loggers by Engine and Host names, use these names in the `log4j.properties` file.

For example, if you want a more detailed Catalina localhost log, add the following lines to the end of the `log4j.properties` you created:

```
log4j.logger.org.apache.catalina.core.ContainerBase.[Catalina].[localhost]=DEBUG
log4j.logger.org.apache.catalina.core=DEBUG
log4j.logger.org.apache.catalina.session=DEBUG
```

Warning: A level of DEBUG produces megabytes of logging and will consequently slow the startup of tc Runtime. Be sure that you use this level sparingly, typically only when you need to debug internal tc Runtime operations.

For the full list of logging levels you can specify when configuring log4j, see Log Levels.

Configuring a Web Application with log4j Logging

You can configure your Web applications to use log4j for their own logging, which is in addition to the tc Runtime logging configuration described in the preceding sections.

The basic steps are as follows:

1. Create a `log4j.properties` file that is similar to the one described in Configuring log4j.
2. Update the `log4j.properties` file with logging information specific to your application. For example, if you want to specify that the `logger` in package `my.package` be at level DEBUG, add the following:

   ```
   log4j.logger.my.package=DEBUG
   ```

3. Put the `log4j-`*`version`*`.jar` file in the `WEB-INF/lib` directory of your Web application, where *`version`* refers to the version of the JAR file, such as `log4j-1.2.15.jar`.

See the log4j documentation for detailed information.

Updating Logging Parameters Dynamically

You can use JMX to modify logging levels and other logging parameters while tc Runtime is executing. The modifications you make using JMX are not persisted; when the server restarts, any changes you made are lost. You could use this feature to enable debugging messages to help troubleshoot an application problem while the problem is occurring. This is useful for problems that take time to develop after a reboot or are otherwise difficult to reproduce.

The JULI implementation of `java.util.logging` allows you to create separate loggers for each Web application by adding `logging.properties` files to your Web applications. This allows you to control logging at a very fine level.

Using JMX, you can list loggers, change the logging level for any single logger by name, and set a new handler (log file) for a logger. You can specify the logger you want to manage using a logger string defined in the `logging.properties` file, prefixed with "`jmx:`", for example `jmx:com.springsource.tcserver`.

Following is code for a JSP you can use to try out using JMX to manage loggers dynamically. It reports whether different logging levels are enabled and also displays the class loader and logger names. Add the JSP to a web application, deploy it, and call it before and after changing the logging level as described in the first example below.

```
<%@page import="org.apache.juli.logging.*"%>
<%
  Log log = LogFactory.getLog(this.getClass());
  String dmessage = "log.jsp log message[DEBUG] "+System.currentTimeMillis();
  String imessage = "log.jsp log message[INFO] "+System.currentTimeMillis();
  String wmessage = "log.jsp log message[WARN] "+System.currentTimeMillis();
  String emessage = "log.jsp log message[ERROR] "+System.currentTimeMillis();
  log.debug(dmessage);
  log.info(imessage);
  log.warn(wmessage);
```

```
  log.error(emessage);
%>
  Debug Enabled: <%=log.isDebugEnabled()%> <br/>
  Info  Enabled: <%=log.isInfoEnabled()%> <br/>
  Warn  Enabled: <%=log.isWarnEnabled()%> <br/>
  Error Enabled: <%=log.isErrorEnabled()%> <br/>

Class Loader: <%=this.getClass().getClassLoader().getParent().
    getClass().getName()%>#<%=System.identityHashCode(this.getClass().
    getClassLoader().getParent())%> <br/>
LoggerName: <%=this.getClass().getName()%>#<%=this.getClass().
    getClassLoader().getParent().getClass().
    getName()%>#<%=System.identityHashCode(this.getClass().
    getClassLoader().getParent())%> <br/>
```

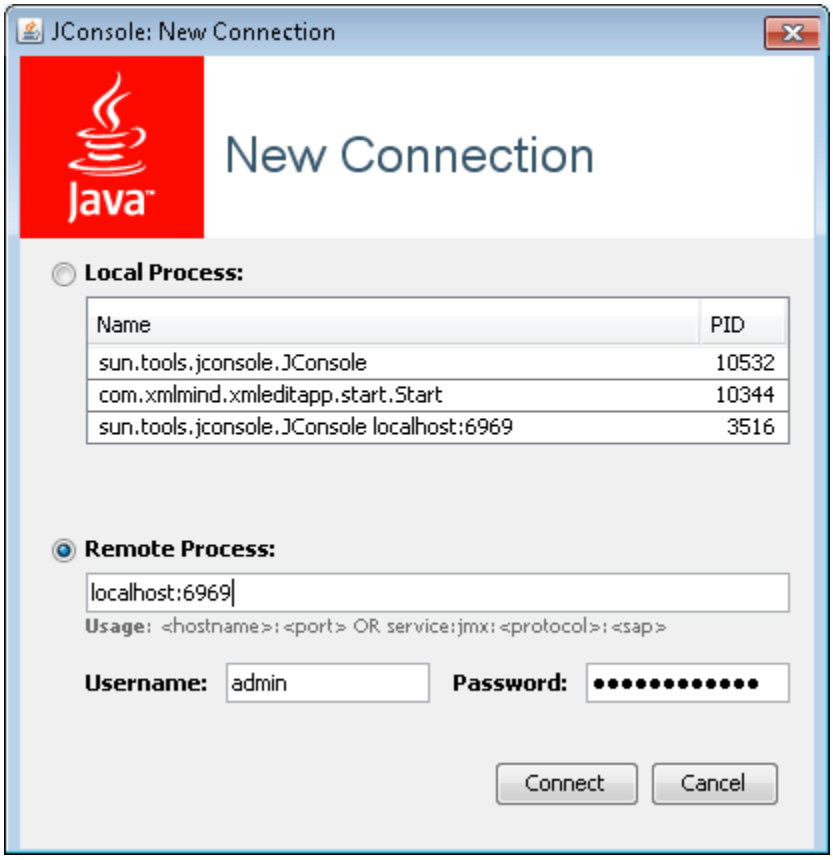
The following examples use JConsole, the Java Monitoring and Management Console included with the JDK, to manage loggers. There is a `jconsole` executable in the JDK `bin` directory that you can execute from a shell or Command Prompt. JConsole connects to a tc Runtime instance at the JMX port, 6969 by default. To verify your JMX port, check the `base.jmx.port` property in the `CATALINA_HOME/conf/catalina.properties` file.

Setting a New Logging Level for a Logger

This example shows how to use JMX to change the logging level for a logger without restarting the tc Runtime instance. A logger string from the `CATALINA_HOME/conf/logging.properties` file identifies the logger.

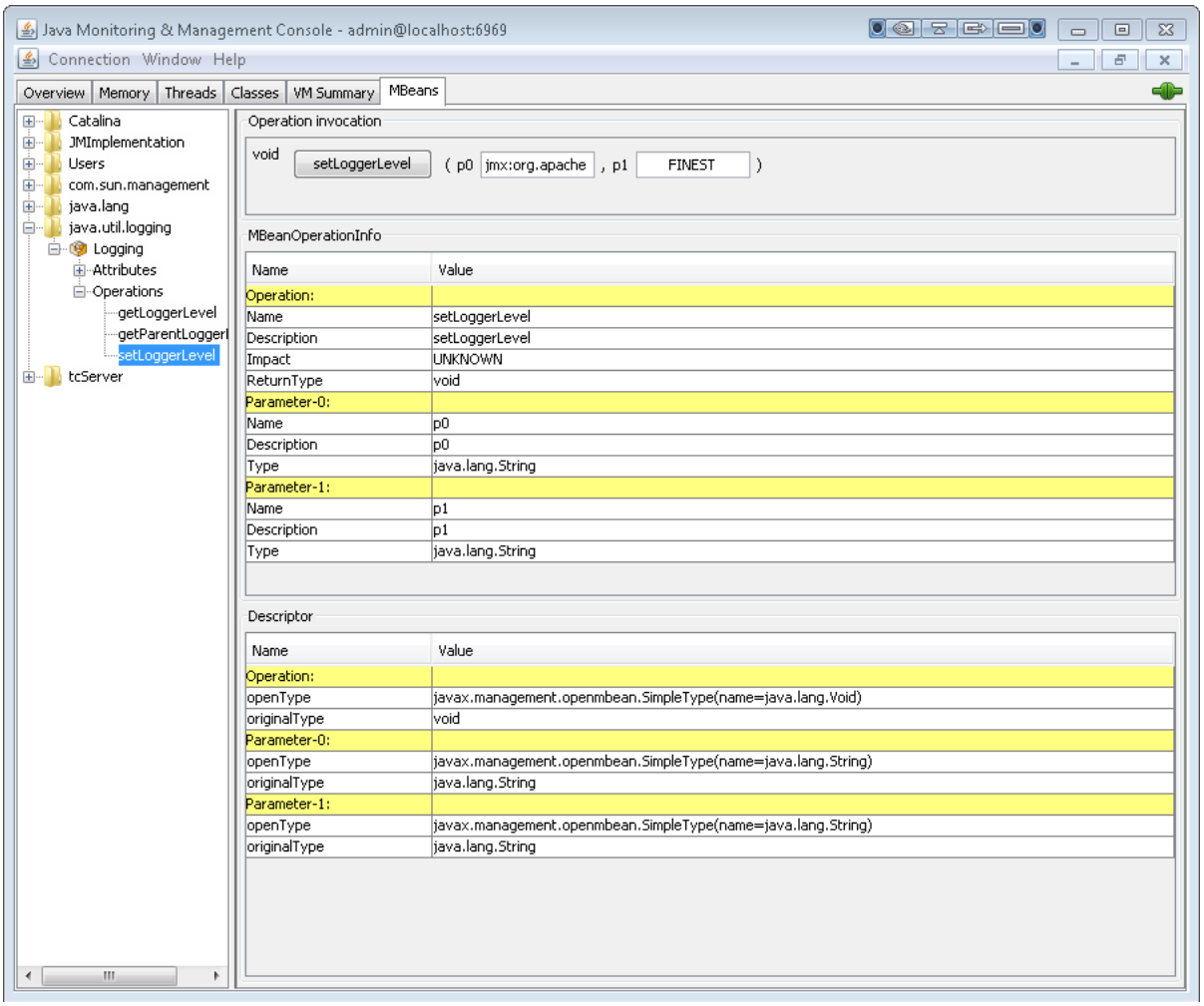
1. Start JConsole and connect to the tc Runtime instance.

   In the New Connection window **Remote Process** field, enter the host name or IP and JMX port for the tc Server, separated by a colon. Enter the user name and password (the defaults are admin/springsource) and click **Connect**.



2. Click the **MBeans** tab.

3. In the tree at the left, expand **java.util.logging** > **Logging** > **Operations**, and click the **setLoggerLevel** operation.



4. In the Operation invocation section, enter the logger name in the **p0** field and the new logging level in the **p1** field. Then click **setLoggerLevel**.

   The logger name, **p0**, can be one of the following:

   - Logger strings defined in `CATALINA_HOME/conf/logging.properties` prefixed with "`jmx:`". For example `jmx:com.springsource.tcserver`, `jmx:org.apache.catalina`, or `jmx:org.apache.tomcat`.
   - A fully qualified logger name, as described in the preceding section.

   If you are using the JSP code presented above to test this feature, copy the logger name from the page's output in your browser and paste it into the **p0** field. Be careful not to copy any trailing spaces into the field.

   The logging level, **p1**, is one of the logging levels described in Logging Levels for java.util.logging: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, OFF, or ALL.

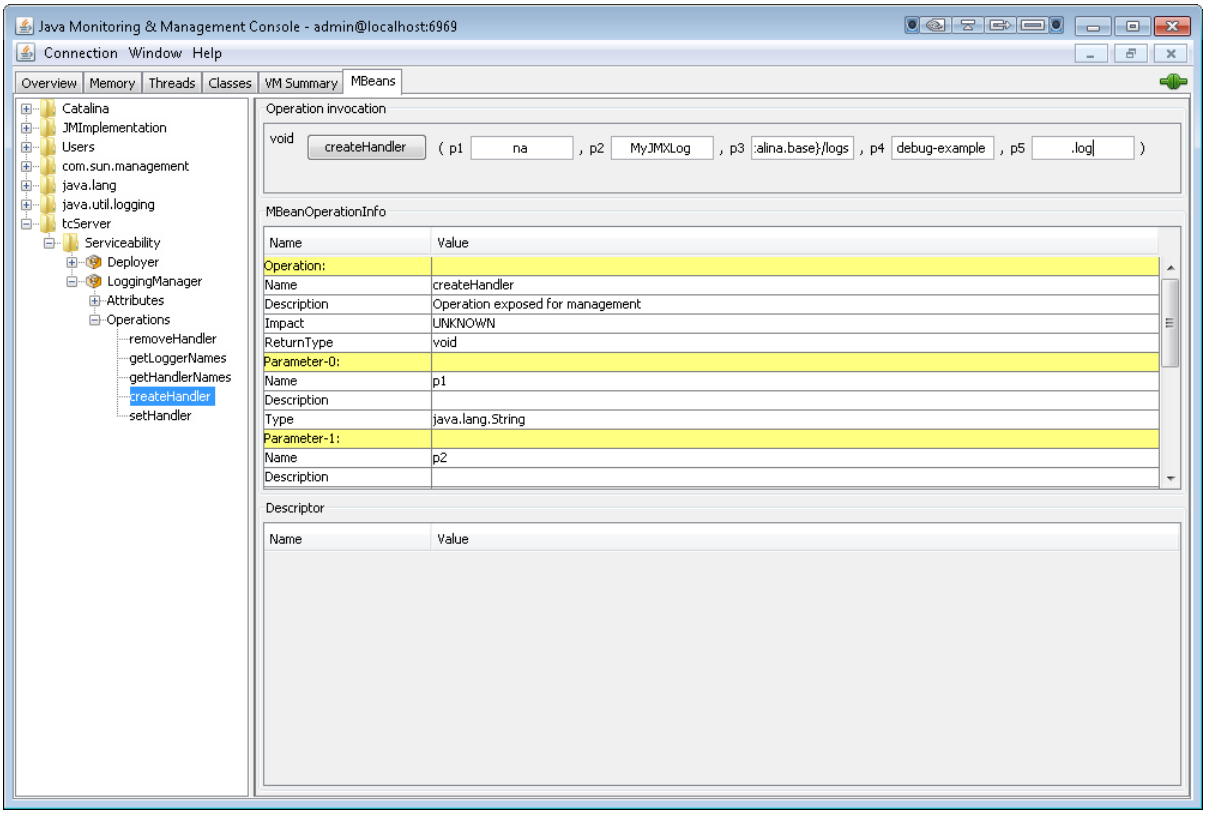
After you click **SetLoggerLevel**, the new logging level takes effect. If you are using the sample JSP code, reloading the page logs messages and updates the status of the logging levels.

Example: Create a New Log File and Redirect Debug Output To It

The following example creates a log handler (a log file), associates it with a logger, and sets the logging level for the logger. The result is a newly created log file with messages redirected into it.

1. Start JConsole and connect to the tc Server instance. (See previous example.)

2. Click the **MBeans** tab and then, in the tree at left, navigate to **tcServer** > **Serviceability** > **LoggingManager** > **Operations**.



3. Click the **createHandler** operation. You use this operation to create a log file. Complete the parameters as follows:

   - **p0**: empty. This parameter is ignored.
   - **p1**: The name of your handler, for example `MyJMXLog`.
   - **p2**: The location of the log file, for example `${catalina.base}/logs`.
   - **p3**: The prefix of the log file name, for example `debug-example`.
   - **p4**: The suffix for the log file name, for example `.log`.

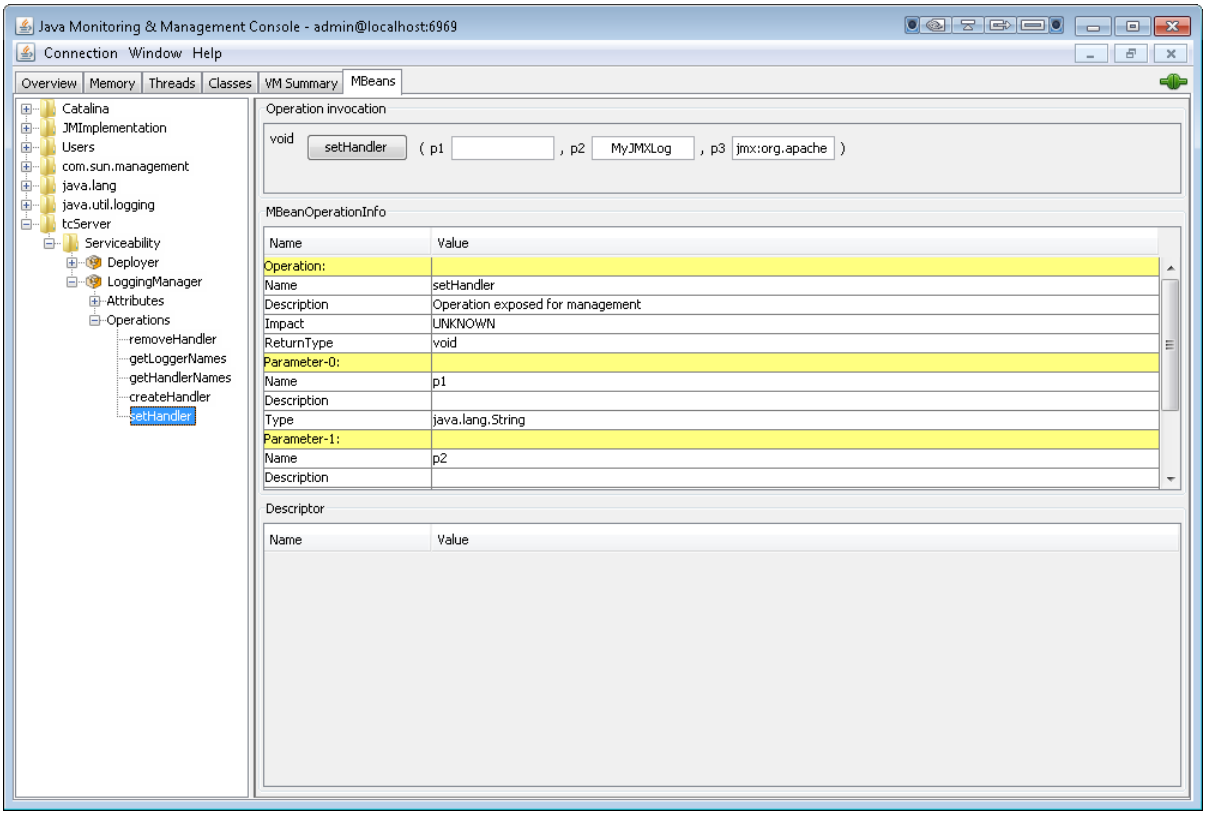
   Parameters **p2**, **p3**, and **p4** establish the location and name of the new log file. The file name is constructed from the prefix, a day timestamp, and the suffix, for example `debug-example.2011-11-11.log`. The **p2** parameter specifies the directory where the file is created, in this example `CATALINA_HOME/logs`.

4. Click **createHandler**.

   Now you can verify that the new log file has been created in the `CATALINA_HOME/logs` directory.

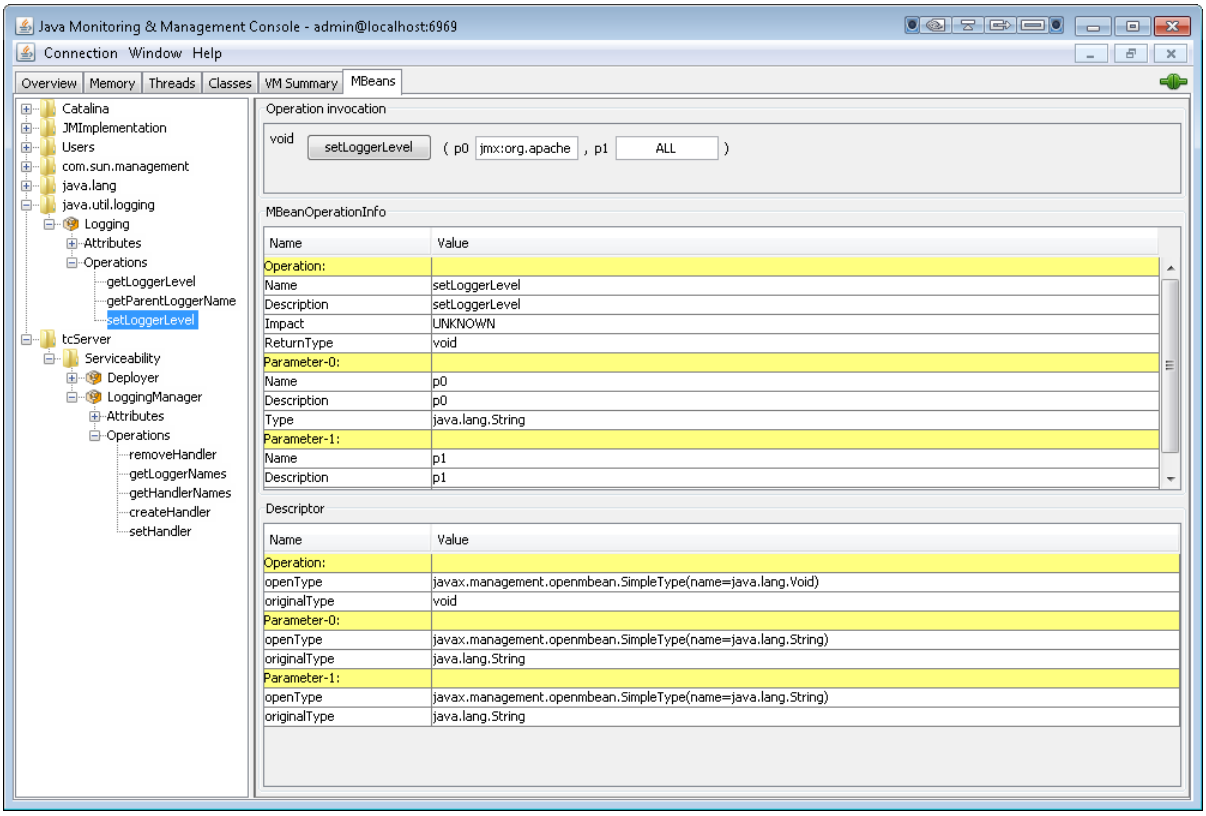
5. Click the **setHandler** operation. You use this operation to associate the log file you created with a logger. Complete the parameters as follows:

   - **p0**: empty.
   - **p1**: The name of your handler, for example `MyJMXLog`.
   - **p2**: The name of the logger, for example `jmx:org.apache`.



6. Navigate to **java.util.logging** > **Logging** > **Operations** and click the **setLoggerLevel** operation. Complete the parameters as follows:

   - **p0**: `jmx.org.apache`
   - **p1**: ALL



7. Click **setLoggerLevel**.

   Messages are now written to the new log file.

Remember that changes you make with JMX are lost when the server is rebooted. The changes are not written to the tc Server configuration files.

Obfuscating Passwords in tc Runtime Configuration Files

SpringSource tc Runtime stores its configuration files in the `CATALINA_BASE/conf` directory; these files include `server.xml`, `context.xml`, `web.xml`, and `jmxremote.password`. By default, passwords in these files are in cleartext. This is typically not a problem during development; however, when you move to production, you will probably want to protect these passwords for security reasons so that the actual password string does not show up in the configuration files.

Passwords appear in these configuration files in a variety of places. For example, as described in Configuring the High Concurrency JDBC Connection Pool, you use the `<Resource>` element of the `server.xml` file to configure a JDBC connection pool, and the element's `password` attribute specifies the password of the user who connects to the database server, as shown in the following sample snippet of the `server.xml` file (only relevant parts shown):

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="-1" shutdown="SHUTDOWN">

 ...

  <GlobalNamingResources>

  <Resource name="jdbc/TestDB"
            auth="Container"
            type="javax.sql.DataSource"
            username="root"
            password="mypassword"
            driverClassName="com.mysql.jdbc.Driver"
```

```
            url="jdbc:mysql://localhost:3306/mysql?autoReconnect=true"
      ...
  </GlobalNamingResources>
  ...
  <Service name="Catalina">
  ...
  </Service>
</Server>
```

Another example is the `jmxremote.password` file that contains the password for the JMX username/role that HQ uses to connect to the JMX server associated with the tc Runtime instance. By default, the password is in cleartext. The following example shows the out-of-the-box file in which the `admin` role has the password `springsource`:

```
# The "admin" role has password "springsource".
admin springsource
```

The remainder of this section describes how to protect the password text in any of the following tc Runtime configuration files located in the `CATALINA_BASE/conf` directory:

- `server.xml`
- `context.xml`
- `web.xml`
- `jmxremote.password`

The section is divided into four topics. The first topic provides the basic obfuscation procedure, and the next three topics provide additional security. All four topics use the `<Resource>` element of the `server.xml` as an example. Apply the same procedure to protect passwords in any of the other configuration files, except where noted.

- Using Base64 Encoding to Obfuscate Passwords
- Using a Passphrase to Encrypt Passwords
- Storing Passphrases and Encrypted Properties in Separate Files
- Being Prompted for the Passphrase When you Start the Instance

Note: This section does not apply to the `tomcat-users.xml` file. In this file, you should hash the password using MD5 or SHA1 to prevent it from being disclosed.

Using Base64 Encoding to Obfuscate Passwords

The following procedure describes the standard way to obfuscate passwords in the configuration files using base64 encoding. Base64 encoding is a scheme that encodes binary data by treating it numerically and translating it into a base 64 representation.

1. Open a terminal command window and change to the *`INSTALL_DIR`*`/vfabric-tc-server-`*`edition`* directory, where *`INSTALL_DIR`* refers to the directory in which you installed tc Runtime and *`edition`* refers to your tc Server edition, such as `developer` or `standard`. For example:

   ```
   prompt$ cd /opt/vmware/vfabric-tc-server-standard
   ```

2. Run the following `java` command on Unix; substitute *`version`* with the version of tc Runtime you are using, such as `6.0.32.A-RELEASE` and substitute `password` with the actual password text you want to obfuscate:

   ```
   prompt$ java -cp \
      tomcat-version/lib/tcServer.jar:tomcat-version/bin/tomcat-juli.jar:tomcat-version/lib/tomcat-coyote.jar \
      com.springsource.tcserver.security.PropertyDecoder -encode base64 password
   ```

   In a Windows command prompt, the syntax is slightly different in that you use back-slashes instead of forward-slashes and semi-colons instead of colons:

```
prompt> java -cp \
   tomcat-version\lib\tcServer.jar;tomcat-version\bin\tomcat-juli.jar;\
   tomcat-version\lib\tomcat-coyote.jar \
   com.springsource.tcserver.security.PropertyDecoder -encode base64 password
```

For example, if you are using version 6.0.32.A-RELEASE of tc Runtime and want to obfuscate the password `mypassword`, run the following command on Unix:

```
prompt$ java -cp \
   tomcat-6.0.32.A-RELEASE/lib/tcServer.jar:tomcat-6.0.32.A-RELEASE/bin/tomcat-juli.jar:\
   tomcat-6.0.32.A-RELEASE/lib/tomcat-coyote.jar \
   com.springsource.tcserver.security.PropertyDecoder -encode base64 mypassword
```

On Windows:

```
prompt> java -cp \
   tomcat-6.0.32.A-RELEASE\lib\tcServer.jar;tomcat-6.0.32.A-RELEASE\bin\tomcat-juli.jar;\
   tomcat-6.0.32.A-RELEASE\lib\tomcat-coyote.jar \
   com.springsource.tcserver.security.PropertyDecoder -encode base64 mypassword
```

The preceding commands are shown on multiple lines, but you should run the command on a single line.

The preceding `java` command outputs the encoded password; it will look something like `bXlwYXNzd29yZA==`.

3. Edit the relevant configuration file (`server.xml` in our example) and substitute the cleartext password with a variable; use the form `${variable_name}`. The following example shows how to substitute the password of the `<Resource>` element with the variable `${db.password}`:

```
<Resource name="jdbc/TestDB"
          auth="Container"
          type="javax.sql.DataSource"
          username="root"
          password="${db.password}"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/mysql?autoReconnect=true"
          ...
```

Important: Variable replacement as described in the preceding example is supported only in the XML files, specifically `server.xml`, global and application `context.xml` files, and global and application `web.xml` files. This means you cannot use variable replacement in the `jmxremote.password` file. For this file, you directly replace the cleartext password with the encoded password. For example:

```
# The "admin" role
admin s2enc://c3ByaW5nc291cmNl
```

4. Edit the `catalina.properties` file (in the case of obfuscating passwords in XML files) and add the following properties:

```
org.apache.tomcat.util.digester.PROPERTY_SOURCE=com.springsource.tcserver.security.PropertyDecoder
com.springsource.tcserver.security.PropertyDecoder.passphrase=base64
db.password=s2enc://bXlwYXNzd29yZA==
```

The first two properties specify how tc Runtime should decode the encoded password; the third property (`db.password`) is the variable you substituted in the `server.xml` file. The value of this property is the encoded password you generated from a previous step, preceded by the required text `s2enc://`.

When obfuscating passwords in tc Runtime configuration files, always precede the encoded text with the hard-coded string `s2enc://`.

5. Restart tc Runtime for the changes to take effect.

Using a Passphrase to Encrypt Passwords

Strengthen password protection by specifying your own passphrase when encrypting the password, rather than using base64. The steps for using passphrase encryption are similar to using base64 encoding but with the following changes:

- When generating the encrypted password, specify your own passphrase rather than the text `base64`. The following Unix example uses the passphrase `mypassphrase`; you can substitute any phrase you want:

```
prompt$ java -cp \
   tomcat-6.0.32.A-RELEASE/lib/tcServer.jar:tomcat-6.0.32.A-RELEASE/bin/tomcat-juli.jar:\
   tomcat-6.0.32.A-RELEASE/lib/tomcat-coyote.jar \
   com.springsource.tcserver.security.PropertyDecoder -encode mypassphrase mypassword
```

  On Windows:

```
prompt> java -cp \
   tomcat-6.0.32.A-RELEASE\lib\tcServer.jar;tomcat-6.0.32.A-RELEASE\bin\tomcat-juli.jar;\
   tomcat-6.0.32.A-RELEASE\lib\tomcat-coyote.jar \
   com.springsource.tcserver.security.PropertyDecoder -encode mypassphrase mypassword
```

  The command is shown on multiple lines, but you should run the command on a single line.

  The preceding `java` command outputs the encrypted password using passphrase encryption; it will look something like `koBC0uF1N200plwJgBfeQg==`.

- When you edit the `catalina.properties` file and add the properties, specify your passphrase instead of `base64` for the `com.springsource.tcserver.security.PropertyDecoder.passphrase` property and set your variable to the new encrypted password:

```
org.apache.tomcat.util.digester.PROPERTY_SOURCE=com.springsource.tcserver.security.PropertyDecoder
com.springsource.tcserver.security.PropertyDecoder.passphrase=mypassphrase
db.password=s2enc://koBC0uF1N200plwJgBfeQg==
```

In the preceding example, the string `mypassphrase` in the `catalina.properties` file is in cleartext. However, this still offers a degree of protection because the seed used for encryption is not readily available; it is available only to the tc Runtime instance. If, however, you do not want the `catalina.properties` file to include the passphrase, or even include the encrypted password itself, you can store them in separate files and simply point to them from `catalina.properties`, as described in the next section.

Storing Passphrases and Encrypted Properties in Separate Files

Although storing the passphrase (when using passphrase encryption) and encrypted passwords in the `catalina.properties` is reasonably secure, some users might prefer to store these values in separate files.

To store the passphrase in a separate file, replace the value of the `com.springsource.tcserver.security.PropertyDecoder.passphrase` property with the name of a file. You can use the `${catalina.base}` variable to specify a directory relative to the `CATALINA_BASE` of the tc Runtime instance.

In the following sample snippet of `catalina.properties`, the passphrase is stored in a file called `secure.file` in the `CATALINA_BASE/conf` directory of the tc Runtime instance:

```
org.apache.tomcat.util.digester.PROPERTY_SOURCE=com.springsource.tcserver.security.PropertyDecoder
com.springsource.tcserver.security.PropertyDecoder.passphrase=${catalina.base}/conf/secure.file
db.password=s2enc://koBC0uF1N200plwJgBfeQg==
```

Create the `secure.file` file: it should contain a single line with the passphrase. For example:

```
mypassphrase
```

Similarly, to store the actual encrypted password in a separate file, replace the password variable (`db.password` in our example) in the `catalina.properties` file with a property called `com.springsource.tcserver.security.PropertyDecoder.properties`. Set this property to the name of a file that contains the password variable.

In the following sample snippet of `catalina.properties`, the encrypted password is stored in a file called `application.properties` in the `CATALINA_BASE/conf` directory of the tc Runtime instance:

```
org.apache.tomcat.util.digester.PROPERTY_SOURCE=com.springsource.tcserver.security.PropertyDecoder
com.springsource.tcserver.security.PropertyDecoder.passphrase=${catalina.base}/conf/secure.file
com.springsource.tcserver.security.PropertyDecoder.properties=${catalina.base}/conf/application.properties
```

Create the `application.properties` file and add the original password variable. Following with our example, the file would include the following:

```
db.password=s2enc://koBC0uF1N200plwJgBfeQg==
```

Being Prompted for the Passphrase When you Start the Instance

Storing the passphrase and encrypted passwords in operating system files when using passphrase encryption is reasonably secure. However, some users may want to be *prompted* for the passphrase so that it does not appear in cleartext in any file at all. This section describes how to do this. The section builds on the information in Using a Passphrase to Encrypt Passwords, so it is assumed that you have already read that section and that you have generated the encrypted password based on a passphrase of your choice.

Warning: This feature requires that you start the tc Runtime instance as a foreground process using the `run` option of `tcruntime-ctl.sh|bat` on both Unix and Windows. On Unix, you can then put the process in the background. On Windows, however, this means that you cannot control the instance using the Windows Services console. For this reason, this feature is not practical for production use on Windows.

To be prompted for the passphrase when you start the tc Runtime instance, update the `catalina.properties` file and set the `com.springsource.tcserver.security.PropertyDecoder.passphrase` property to the value `console`. For example:

```
org.apache.tomcat.util.digester.PROPERTY_SOURCE=com.springsource.tcserver.security.PropertyDecoder
com.springsource.tcserver.security.PropertyDecoder.passphrase=console
db.password=s2enc://koBC0uF1N200plwJgBfeQg==
```

Subsequently, you must start the tc Runtime instance as a foreground process using the `run` option of `tcruntime-ctl.sh|bat`. For example, on Unix:

```
prompt$ cd /opt/vmware/vfabric-tc-server-standard
prompt$ ./tcruntime-ctl.sh myserver run
```

After the usual startup information, the script asks you for the passphrase. After entering it, the tc Runtime instance starts as usual.

On Unix, if you want to now put the tc Runtime instance process in the background, first suspend the process by using control-Z, and then enter `bg` at the Unix prompt to put the suspended process in the background.

General Security Best Practices

The preceding sections provide specific information about obfuscating and encrypting passwords in tc Runtime configuration files using a variety of methods. This section provides general best practices for securing your tc Runtime instances.

For additional security, SpringSource recommends that:

- On the computer on which you you have installed tc Server, create an operating system user whose *only* purpose is to run the tc Runtime process. In other words, this user would be the only user who starts/stops the tc Runtime instance, and this user would do nothing else but start/stop the tc Runtime process.

- Make it impossible for anyone to log on to the computer directly as this dedicated tc Server user.
- Set the permissions for all tc Runtime configuration files so that they are readable *only* by this dedicated tc Server user.

If you set up the preceding scenario, the only users who will be able read the passwords in the tc Runtime configuration files (whether they are in cleartext, are obfuscated, or encrypted) are users with `root` privileges.

To implement this scenario on Windows, you can use the `install` *`run-as-user`* option of `tcruntime-ctl.bat` to install the tc Runtime instance as a Windows service and specify that it should run as the dedicated tc Server user. See "tcruntime-ctl Command Reference" in *Getting Started with vFabric tc Server* for details.

On Unix, you can use the `boot.rc.template` script to customize your Unix boot process so that the tc Runtime instance starts automatically when your computer starts. Use the `TOMCAT_USER` variable in the script to specify the dedicated tc Server user that you want the tc Runtime instance to run as. You then use the boot script the same way you use any other Unix boot script on your computer. For example, you might copy it to the `/etc/init.d` directory, giving it a unique name such as `my-tc-runtime-instance`. Then you would link this script from `/etc/rc*.d` as appropriate, depending on when you want the tc Runtime instance to start during the Unix boot sequence.

Alternatively, if you do not want the tc Runtime instance to start automatically when the Unix computer boots, you can run the `my-tc-runtime-instance` file in the `/etc/init.d` directory as the `root` user, rather than start the tc Runtime process using the `tcruntime-ctl.sh` script.

For more information about the `boot.rc.template` script, see "Unix: Starting tc Runtime Instances Automatically at System Boot Time" in *Getting Started with vFabric tc Server*.

Configuring an Oracle DataSource With Proxied Usernames

When you configure a global shared JDBC datasource for a particular tc Runtime instance, by default all deployed applications that use the datasource connect to the configured database using the same username and password. This user is called a *proxy*, because the proxy user performs a database task on behalf of the user using the application deployed to tc Runtime. When an application user connects anonymously through a proxy, however, it is impossible to customize security for individual users or get a meaningful audit trail of the users that actually used the database.

For this reason it can be useful to configure the tc Runtime instance so that, while many applications share a particular global datasource, each application connects to the database using a *different* username and password via the proxy user, rather than directly through the proxy user that is configured for the datasource itself. SpringSource tc Runtime has implemented this feature using the Oracle proxy connection authentication.

NOTE: This feature applies only to Oracle datasources.

The following procedure describes how to configure tc Runtime, and your applications, to use a shared global Oracle datasource with the proxy connection authentication.

1. Configure a standard shared global Oracle datasource for your tc Runtime instance by adding a `<Resource>` child element of the `<GlobalNamingResource>` element in the `server.xml` file. The actual configuration depends on your Oracle database environment, but the following snippet provides an example (relevant sections shown in bold):

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="-1" shutdown="SHUTDOWN">

 ...

  <GlobalNamingResources>

    <Resource name="jdbc/TestDB" auth="Container"
      type="oracle.jdbc.pool.OracleDataSource"
      description="Oracle Datasource"
      factory="oracle.jdbc.pool.OracleDataSourceFactory"
      url="jdbc:oracle:thin:@//localhost:1521/orcl"
      user="default_user"
      password="password"
```

```
      connectionCachingEnabled="true"
      connectionCacheName="CXCACHE"
      connectionCacheProperties="{MaxStatementsLimit=5, MinLimit=1, MaxLimit=1, ValidateConnection=true}"/>

  </GlobalNamingResources>
  ...
  <Service name="Catalina">
  ...
  </Service>
</Server>
```

In the preceding `server.xml` snippet, by default the `jdbc/TestDB` datasource connects to the database as the user `default_user` with password `password`; this is the proxy user.

2. Use the `jdbc/TestDB` datasource in your servlet and JSPs as usual.

   The following snippet shows an example of using it in a JSP to get a connection to the database:

```
<%@ page import="java.sql.Connection,java.sql.ResultSet,java.sql.Statement,javax.naming.*,javax.sql.*"%>

    Context initContext = new InitialContext();
    Context envContext  = (Context)initContext.lookup("java:/comp/env");
    DataSource datasource = (DataSource)envContext.lookup("jdbc/TestDB");
    Connection con = datasource.getConnection();
    ...
```

3. For each application that uses the datasource and for which you want to configure a specific proxied-user, update the application's `META-INF/context.xml` file by adding a `<ResourceLink>` element that links the global Oracle datasource to the `com.springsource.tcserver.oracle.OracleProxyDataSourceFactory` factory. Use the `username` and `password` attributes of `<ResourceLink>` to configure the specific user you want this particular application to connect to the database as, via the proxy user. For example (relevant section shown in bold):

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <ResourceLink global="jdbc/TestDB" name="jdbc/TestDB"
           username="proxieduser" password="proxypassword"
           factory="com.springsource.tcserver.oracle.OracleProxyDataSourceFactory"/>
</Context>
```

   When the application described by this `context.xml` file uses the `jdbc/TestDB` datasource, it will connect to the database first as the proxy user (`default_user`) and then open a proxy connection as the `proxieduser` user, with password `proxypassword`.

   For this feature to work correctly, you must update the `context.xml` files for each relevant application, not the global `context.xml` file located in the `CATALINA_BASE/conf` directory.

4. For the changes to take effect, restart your tc Runtime instance, which in turn redeploys all relevant applications.

5. If you have not already done so, create all required Oracle database users that match the usernames you configured in the `context.xml` and `server.xml` files. For example:

```
create user default_user identified by password;
create user proxieduser identified by proxypassword;
grant dba to default_user;
grant dba to proxieduser;
ALTER USER proxieduser GRANT CONNECT THROUGH default_user AUTHENTICATED USING password;
```

   The preceding SQL statements show how the `proxieduser` connects to the Oracle database through `default_user`. These SQL statements are just examples; for complete descriptions of these statements, see your Oracle database documentation.

Reporting Status for a Deployed Application, Host, or Engine

By default, the error or status page that tc Runtime displays when it encounters a particular HTTP status or error code (such as `404` when tc Runtime does not find a requested resource) is hard-coded. However, you might want or need to change the displayed error, for simple customization reasons or because of your organization's security requirements that dictate how error pages should work and what they should look like. This section describes how to customize what tc Runtime displays when it encounters a particular HTTP status code.

The HTTP 1.1 specification defines the HTTP status codes. The following list describes some common codes:

- `403 Forbidden`: The server understood the request, but is refusing to fulfill it.
- `404 Not Found`: The server has not found anything matching the Request-URI.
- `500 Internal Server Error`: The server encountered an unexpected condition which prevented it from fulfilling the request.

To customize the way tc Runtime responds when it encounters one of these codes, you add a `Valve` element to the `server.xml` configuration file whose `className` attribute is `com.springsource.tcserver.security.StatusReportValve`. The `StatusReportValve` has a number of other attributes that describe its behavior, as described in Attributes of the StatusReportValve.

You can specify the `StatusReportValve` as a direct child element of either the `Host` or `Engine` element in the `server.xml` file, depending on the associated Catalina container for which you want to configure the Valve. If you specify that the `StatusReportValve` is a direct child element of `Engine`, then you must explicitly disable the Valve at the `Host` level, using the `Host` attribute `errorReportValveClass=""`.

You define how tc Runtime handles a particular HTTP status code by adding an attribute to the `StatusReportValve` whose name is `error.XXX`, where `XXX` is the numerical status code, such as `error.404`. Then set the value of this attribute in one of the following ways:

- `error.XXX=file://valid/file/path/URI`: Specifies that when tc Runtime encounters the XXX status code, it should display the specified URI. If the URI is not valid, the file doesn't exist, or it is not readable, tc Runtime ignores the status code.
- `error.XXX=/path/to/file`. Specifies that when tc Runtime encounters the XXX status code, it should display the specified file. If the path does not point to a file node, tc Runtime interprets the path as a message string. If the file node is a directory or not readable, tc Runtime ignores the status code.
- `error.XXX=message string`: Specifies that when tc Runtime encounters the XXX status code, it should display the specified message as the body of the status response.

If tc Runtime encounters a status code that you have not defined in `StatusReportValve` using an `error.XXX` attribute, then tc Runtime does not act upon the status code. Additionally, if your application has already responded to the status code, then the `StatusReportValve` does not act upon the status code.

Once you configure your tc Runtime instance with the `StatusReportValve` and you start the instance, you can dynamically change the attributes of the Valve using JMX.

The following `server.xml` snippet shows an example of configuring a `StatusReportValve` for the Catalina Engine; only relevant parts of `server.xml` are shown (in bold):

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="${shutdown.port}" shutdown="SHUTDOWN">

   ...
   <Service name="Catalina">
   ...

      <Engine name="Catalina" defaultHost="localhost">
```

```
      <Valve className="com.springsource.tcserver.security.StatusReportValve"
             fileEncoding="utf-8"
             contentType="text/html"
             characterEncoding="utf-8"
             zeroLengthContent="false"
             commitOnReport="true"
             cacheFiles="true"
             removeException="true"
             error.500="${catalina.base}/conf/500.html"
             error.404="${catalina.base}/conf/404.html"
             error.403="I am sorry, you do not have access"
      />

      ...
      <Host name="localhost"  appBase="webapps"
            unpackWARs="true" autoDeploy="true" deployOnStartup="true"
            deployXML="true" xmlValidation="false" xmlNamespaceAware="false"
            errorReportValveClass="" >
      </Host>
     </Engine>
   </Service>
</Server>
```

In the preceding example, the `StatusReportValve` can act upon three HTTP status codes: `404`, `500`, and `403`. When tc Runtime encounters the `404` status code, it displays the contents of the file `CATALINA_BASE/conf/404.html`. Similarly for status code `500`, although in this case it displays the file `CATALINA_BASE/conf/500.html`. If tc Runtime encounters the status code `500`, it displays the literal message `I am sorry, you do not have access`.

Note that, because the `StatusReportValve` is configured at the `Engine` level, the child `Host` element explicitly disables the Valve using the attribute `errorReportValveClass=""`.

The following table describes all the attributes of the `StatusReportValve`.

Table 4.4. Attributes of the StatusReportValve

Attribute	Description
`className`	Specify the `com.springsource.tcserver. security.StatusReportValve` class, or a class that extends the `StatusReportValve` class.
`fileEncoding`	Specifies the encoding of the displayed static files. If you do not specify this attribute, tc Runtime uses the default platform encoding.
`contentType`	Specifies the `Content-Type` header for the HTTP response. Default value is `text/html`. See MIME Media Types for the full list.
`characterEncoding`	Specifies the `charset` parameter of the `Content-Type` header for the HTTP response. Default value is `utf-8`. See Character Sets for the full set.
`zeroLengthContent`	If you have set this attribute to `true` and the response is not committed, the Valve returns with a 0 length body. Useful for `mod_jk` and reverse proxy where the Web server only overrides the body if it is of 0 length (effectively, it has no body.)
`commitOnReport`	If you have set this attribute to `true`, the StatusReportValve always tries to commit the response even with a 0 length body. If you set it to `false`, then Valves further up the chain may change the response.
`cacheFiles`	If you set this attribute to `true`, the Valve caches the content of the static pages as `java.lang.ref.WeakReference<String>`. Once cached, tc Runtime makes no attempt to read the file system unless the garbage collector clears the weak references.
`removeException`	If you set this attribute to `true`, the Valve removes the `Globals.EXCEPTION_ATTR` from the request attribute. Valves further up in the chain will no longer have access to the exception that caused the error.
`error.XXX`	Specifies that tc Runtime should act upon the `XXX` status code by displaying either the specified URI, file, or message string. See the previous discussion for details.

Enabling Thread Diagnostics

`ThreadDiagnosticsValve` collects diagnostic information from tc Runtime request threads. If the thread has JDBC activity on a DataSource, the collected diagnostics can include the JDBC query, depending on how you configure `ThreadDiagnosticsValve`. The collected information is exposed through JMX MBeans.

Hyperic Server, via the tc Server plug-in, uses `ThreadDiagnosticsValve` to enable and access thread diagnostics, as described in Enabling the Slow or Failed Request Alert.

The diagnostics collected for a thread include the following:

- The URI of the request
- The query portion of the request string
- Time the request began
- Time the request completed
- Total duration of the request
- The number of garbage collections that occurred during the request
- The time spent in garbage collection
- Number of successful connection requests
- Number of failed connection requests
- Time spent waiting for connections
- Text of each query executed
- Execution time for each query
- Status of each query
- Execution time for all queries
- Stack traces for failed queries

Setting Up ThreadDiagnosticsValve

Set up `ThreadDiagnosticsValve` by adding a `Valve` child element to the `Engine` or `Host` element in `conf/server.xml` and configuring a DataSource, if you want JDBC diagnostics.

If you include the `diagnostics` template in the `tcruntime-instance create` command, the configuration is done for you, including creating a DataSource whose activity will be included in the diagnostics. For example:

```
prompt$ ./tcruntime-instance.sh create -t diagnostics myInstance
```

When you create a tc Runtime instance using the `diagnostics` template, the following `Valve` element is inserted as a child of the `Engine` element in the `conf/server.xml` file of the new instance.

```
<Valve className="com.springsource.tcserver.serviceability.request.ThreadDiagnosticsValve"
       loggingInterval="10000"
       notificationInterval="60000"
       hreshold="10000"/>
```

You can, of course, add the `Valve` element manually. The following table describes the attributes you can set on the `Valve` element for `ThreadDiagnosticsValve`.

Table 4.5. Properties of ThreadDiagnosticsValve

Attribute	Description
`className`	The managed class: `com.springsource.tcserver.serviceability.request.ThreadDiagnosticsValve`. Required.
`threshold`	The minimum time (milliseconds) a request must last to be reported. A request must exceed this time to qualify. The default is `500`.
`history`	The number of qualified requests to keep in the history. The default is `1000`.
`loggingInterval`	The minimum number of milliseconds between logging requests, to prevent flooding. The default is `5000`.
`notificationInterval`	The minimum number of milliseconds between JMX notifications, to avoid flooding. The default is `5000`.
`logExtendedData`	If `true`, a detailed message is logged for the thread, including the thread name, priority, id, and stack traces. Default: `false`.

Configuring JDBC Diagnostics

The `ThreadDiagnosticsValve` monitors a DataSource if it is configured with the `ThreadQueryReport` jdbcInterceptor. Furthermore, the `ThreadQueryReport` interceptor is automatically added when the DataSource is created with `com.springsource.tcserver.serviceability.request.DataSourceFactory`. Therefore, if you do not want JDBC diagnostics, set the DataSource `factory` attribute to `org.apache.tomcat.jdbc.pool.DataSourceFactory` instead. Another option is to use `org.apache.tomcat.jdbc.pool.DataSourceFactory` and explicitly add `com.springsource.tcserver.serviceability.request.ThreadQueryReport` to the DataSource's `jdbcInterceptors` attribute in `server.xml`, which enables JDBC diagnostics.

The following example is the DataSource added to `server.xml` when you use the `diagnostics` template to create a tc Runtime instance:

```
<Resource auth="Container"
          driverClassName="com.mysql.jdbc.Driver"
           factory="com.springsource.tcserver.serviceability.request.DataSourceFactory"
           initialSize="10"
           jdbcInterceptors="ConnectionState;StatementFinalizer;SlowQueryReportJmx(threshold=10000)"
           jmxEnabled="true"
           logAbandoned="true"
           maxActive="100"
           maxWait="10000"
           minEvictableIdleTimeMillis="30000"
           minIdle="10"
           name="jdbc/TestDB"
           password="password"
           removeAbandoned="true"
           removeAbandonedTimeout="60"
           testOnBorrow="true"
           testOnReturn="false"
           testWhileIdle="true"
           timeBetweenEvictionRunsMillis="5000"
           type="javax.sql.DataSource"
           url="jdbc:mysql://localhost:3306/mysql?autoReconnect=true"
           username="root"
           validationInterval="30000"
           validationQuery="SELECT 1"/>
```

Even though the `jdbcInterceptors` attribute does not include `ThreadQueryReport`, diagnostics will be produced for this DataSource because it uses the `com.springsource.tcserver.serviceability.request.DataSourceFactory`.

5. Using the tc Server Command-Line Interface

This chapter documents `tcsadmin`, the vFabric tc Server command-line interface (CLI).

Overview of tcsadmin

The `tcsadmin` command-line interface (CLI) is the script version of the tc Server-related features of the HTML HQ user interface. Use `tcsadmin` to configure and manage both single and groups of tc Runtime instances.

To use the CLI, you must install the HQ Server. When you run the various CLI commands, you specify an HQ Server by its hostname and port.

You can perform the following tasks with `tcsadmin`; all tasks are within the context of a particular HQ Server:

- List the known tc Runtime instances, tc Runtime groups, and members of a group.
- Create a group of tc Runtime instances.
- List the applications deployed to a tc Runtime instance or tc Runtime group and the current status of the applications.
- Upload and deploy an application to a tc Runtime instance or group.
- Start, stop, reload, and undeploy an application to a tc Runtime instance or group.
- Modify the configuration of a tc Runtime instance or group.
- Upload a `server.xml` file to a tc Runtime instance or group.
- Download the configuration of a tc Runtime instance or group and write it to a file.
- Start, stop, or restart a tc Runtime instance or all members of a tc Runtime group.

In application management and configuration operations for a *group* of tc Runtime instances, the CLI performs the operations sequentially and synchronously. For example, if you deploy a Web application to a group of tc Runtime instances, the CLI deploys the application to one of the tc Runtime instances, waits for success, and only then deploys the application to another instance in the group.

Any user configured to access the HQ user interface can also use the `tcsadmin` CLI. Additionally, all role-based access is honored for all tc Runtime operations, including server configuration, application management, and server control.

HQ API Command-Line Interface (hqapi.sh)

Use the `tcsadmin` CLI to execute commands that are specific to tc Runtime, such as listing the applications deployed to a particular tc Runtime instance or modifying the specific configuration of a tc Runtime instance. You can also use the HQ API command-line tool `hqapi.sh` to perform operations that are more related to the HQ Server itself, such as accessing and updating HQ inventory and related configuration data and working with HQ platforms, servers, services, groups, metric collection settings, alerts, escalations, users, and roles. The two CLIs complement each other: `tcsadmin.sh` works directly with tc Runtime instances and `hqapi.sh` works with the higher-level HQ inventory.

You download the `hqapi.sh` CLI from the HTML-based HQ user interface. After logging in to the user interface, click the **Administration** tab, then click on the **HQ Web Services API** link, and download and unzip the `hqapi-`*`version`*`.tar.gz` file.

For detailed documentation about using `hqapi.sh`, see "HQApi Command-Line Tools" in *vFabric Hyperic Web Services API*.

Tips for Windows Users

In this documentation, it is assumed that you are running the `tcsadmin` command-line interface on a Unix platform. If you are running Windows, the following changes to the documentation apply:

- Execute the Windows-specific `tcsadmin.bat` command rather than the Unix `tcsadmin.sh` command.
- Use back-slashes (`\`) instead of forward-slashes when specifying a directory path, as well as the standard Windows method for indicating disks, such as `c:\home\springsource`.
- Use quotation marks to enclose directory names that contain spaces. For example, `--localpath="c:\home\My Web Apps\test.war"`.

Downloading the tcsadmin Command-Line Interface

Download the `tcsadmin` command-line interface binaries onto each computer on which you want to run the command:

1. On the computer on which you want to download and install the `tcsadmin` command-line interface, invoke the HQ user interface in a browser. See "Getting Started with the HQ User Interface" in *Getting Started with vFabric tc Server* for details.
2. Click the **Administration** tab at the top of the main HQ user interface page.
3. In the **Plugins** section, click the `tc Server Command-line Interface` link.
4. Click the `springsource-tcserver-scripting-client.zip` link and save the ZIP file to a temporary directory on your computer.
5. Unzip the file to the directory of your choice.
6. The `tcsadmin.sh` (Unix) and `tcsadmin.bat` (Windows) scripts are located in the `springsource-tcserver-scripting-client-`*`version`*`/bin` directory, where *`version`* refers to a version string, such as `2.1.0`.

General Syntax of the tcsadmin Command-Line Interface

Use the following syntax with the `tcsadmin` command-line interface:

```
tcsadmin.sh|bat command options connection-parameters
```

where:

- Unix users use the `tcsadmin.sh` command-line interface and Windows users use `tcsadmin.bat`.
- *`command`* refers to a specific command, such as `start-application`, which starts an application that is deployed to a tc Runtime instance or group, or `modify-configuration` to change the configuration of a tc Runtime instance or group. See List of Commands.
- *`options`* refers to one or more options for the specific command. These options take the form `--optionname=value`, such as `--groupname=MyGroup1`. The list of available options depends on the specified command.
- *`connection-parameters`* is the list of parameters that define the HQ Server to which the command-line interface connects. You can also specify the connection parameters in a file to avoid specifying the same parameters at the command-line. See Connection Parameters .

The following example shows how to start an application called `myApp` that is deployed to the tc Runtime group called `Group1`.

```
prompt$ ./tcsadmin.sh start-application --groupname=Group1 --application=myApp
```

In the example, the command is `start-application` and the options are `--groupname=Group1` and `--application=myApp`. Because no connection parameters are specified at the command-line, it is assumed that either the user running the command has created a *`user.home/`*`.hq/client.properties` file, or the default values of the connection parameters are adequate.

In general, the `tcsadmin` commands that list resources, such as `list-servers` or `list-applications`, output one line for each resource and separate each bit of information about the resource with the pipe (`|`) character. For example:

```
prompt$ ./tcsadmin.sh list-servers

10115|mydesktop tc Runtime myserver|/opt/vmware/vfabric-tc-server-standard/myserver|Running
```

The commands that perform an actual configuration task, such as `create-group`, output the success or failure of the command. For example:

```
prompt$ ./tcsadmin.sh create-group --name=Group1 --version="6.0" \
    --description="A group of tc Runtime 6.0 instances" --location="San Francisco"

Command create-group executed successfully
```

List of Commands

The following tables list `tcsadmin` commands, along with a brief description. A command is one of the following types:

- **Inventory**. Manage the tc Runtime resources in the HQ inventory, such as listing the known tc Runtime instances and groups and creating new groups.
- **Application Management**. Deploy applications to a tc Runtime instance or group and manage already-deployed applications.
- **Configuration**. Configure tc Runtime instances and the members of a tc Runtime group.
- **Control**. Start, stop, and restart tc Runtime instances or groups.

Click the name of each command for details, including its associated options and examples.

Table 5.1. tcsadmin Inventory Commands

Command	Description
list-servers	Lists all servers of type `SpringSource tc Runtime 6.0` and `SpringSouce tc Runtime 7.0` in the HQ inventory.

Command	Description
modify-server	Changes the name or description of an existing tc Runtime instance.
list-groups	Lists all tc Runtime groups in the HQ inventory.
create-group	Creates a new tc Runtime group.
add-server-to-group	Adds a tc Runtime instance to an existing group.
remove-server-from-group	Removes a tc Runtime instance from a group.
delete-group	Deletes an existing tc Runtime group.

Table 5.2. tcsadmin Application Management Commands

Command	Description
list-applications	Lists applications deployed to a tc Runtime instance or group and return their status.
deploy-application	Uploads and deploys an application to a tc Runtime instance or group from a local or remote file system.
start-application	Starts an application that was previously deployed to a tc Runtime instance or group.
stop-application	Stops an application that was previously started.
reload-application	Reloads an application that is currently deployed to a tc Runtime instance or group.
undeploy-application	Undeploys an application that is currently deployed to a tc Runtime instance or group.

Table 5.3. tcsadmin Configuration Commands

Command	Description
put-file	Modifies the configuration of a tc Runtime instance or each member of a tc Runtime group.
get-file	Downloads the configuration of a tc Runtime instance to a file.
list-jvm-options	Lists the JVM options for a single tc Runtime instance.
set-jvm-options	Sets the JVM options for a tc Runtime instance or group.

Table 5.4. tcsadmin Control Commands

Command	Description
start	Starts a tc Runtime instance or group.
stop	Stops a tc Runtime instance or group.
restart	Restarts a tc Runtime instance or group.

Connection Parameters

The `tcsadmin` command-line interface must connect to an HQ Server to perform the work associated with a particular command. This means that each time you use the CLI, you must specify connection parameters that indicate the particular HQ Server to which you want to connect.

Specify these parameters on the command-line, or create a *`user.home`*`/.hq/client.properties` file that lists the parameters and their values to avoid having to specify the connection parameters with every command. The variable *`user.home`* refers to the home directory of the user running the command-line interface.

The `.hq` directory name starts with a dot. On Windows, the home directory of a user is `c:\Documents and Settings\<user_name>`.

Parameters specified at the command-line override the client properties file. If the CLI does not find any value for a parameter, either at the command-line or in the client properties file, it uses the default value.

The following table lists these connection parameters and their default values:

Table 5.5. tcsadmin Connection Parameters

Connection Parameter	Description	Default Value
--host=*hostname*	Host name of the computer that hosts the HQ Server. If you execute the tc Server CLI on the same computer on which the HQ server is running, you can specify `localhost`.	`localhost`
--port=*portnumber*	TCP/IP port number to which the HQ Server listens.	`7080`
--user=*username*	Name of the user that the CLI uses to connect to the HQ Server. This user must be configured to use the HQ user interface. All role-based access is honored.	`hqadmin`
--password=*password*	The password of the user that the CLI uses to connect to the HQ Server.	`hqadmin`
--secure	Specifies that the CLI connects to the HQ Server over a secure connection (SSL).	Parameter does not take a value when specified at the command-line. If you do not specify the parameter, the CLI connects using a non-secure connection.

The following example uses these parameters on the command-line:

```
prompt$ ./tcsadmin.sh list-servers --host=localhost --port=7081 \
         --user=tc_user --password=super_secret --secure
```

In the example, the CLI connects to an HQ Server on the same computer on which the CLI is running. The HQ Server listens at the TCP/IP port `7081`. The username and password of the user connecting to the HQ Server are `tc_user` and `super_secret`, respectively. The CLI connects to the HQ Server through SSL.

If you want to specify the same parameters in a file to avoid specifying them at the command line, create a file called `client.properties` in the `.hq` sub-directory of the home directory of the user running the CLI. The following example shows the content of this file using the parameter values specified above:

```
host=localhost
port=7081
user=tc_user
password=super_secret
secure=true
```

Even though the `--secure` parameter does not take a value at the command line, it *does* take a Boolean value in the `client.properties` file: `true` to enable a secure connection, `false` otherwise.

The following example shows the previous command line example, but without the connection parameters; the CLI reads the values from the *`user.home`*`/.hq/client.properties` file:

```
prompt$ ./tcsadmin.sh list-servers
```

In the remainder of the documentation, it is assumed that you have created a `client.properties` file that contains the connection parameters; this is so sample usage of the server configuration and application management commands is easier to read.

Using Special Characters as Parameter Values

Sometimes it is necessary to use special characters, such as a space or a meta character, as part of a parameter value. The following guidelines describe how to use these special characters:

- Enclose parameter values that contain spaces in quotation marks. For example:

  ```
  prompt> ./tcsadmin list-servers --platform="My Machine"
  ```

- On Unix, escape meta-characters by preceding them with a backslash (\). Unix meta-characters include the following:

  ```
  \ . ! $ % ^ & * | { } [ ] " ' ` ~ ; .
  ```

  You must escape these characters on Unix platforms because they have special meaning in the shell. For example, to specify the name of an application that contains a pipe character:

  ```
  prompt > ./tcsadmin start-application --groupname=Group1 --application=my\|app
  ```

- Be particularly careful with the set-jvm-options command, because its `--options` argument takes a comma-delimited set of JVM options. If one of the options themselves includes a comma, then be sure to escape it with a backslash; otherwise, the `tcsadmin` script interprets it as a delimiter between a list of JVM options.

Group Command Behavior

You can run many `tcsadmin` commands against a single tc Runtime instance or a group of tc Runtime instances. The following guidelines describe the behavior of commands if you run then against a group and a failure occurs:

- Group application deployment and configuration commands, such as `deploy_application` or `put-file`, execute sequentially and synchronously. For example, if you use `tcsadmin` to deploy a Web application to a group, the script deploys the application to one of the tc Runtime instances in the group, waits for the deployment to succeed or fail, then deploys the application to another tc Runtime instance in the group, and so on until the script has attempted to deploy the application to all members of the group. You cannot specify the order of execution for the tc Runtime instances in a group.

  If the script encounters a failure when deploying to one tc Runtime instance in the group, it reports the failure but then continues to deploy the application to any remaining tc Runtime instances in the group.

- Group control commands, such as `start`, also happen sequentially and synchronously, and you cannot specify the order in which the tc Runtime instances in the group start, stop, and restart. However, if the script encounters a failure when starting, stopping, and restarting a tc Runtime instance, the script stops all execution and does *not* continue to execute the command against any remaining tc Runtime instances. Server control actions that have already executed are *not* rolled back.

  For example, assume a tc Runtime group includes five tc Runtime instances that are all currently stopped and you execute the `start` command against the group. Assume that two tc Runtime instances started successfully, but then the third encountered

a problem and returned a failure. The script stops all execution. At that point, the first two tc Runtime instances are still started, and the remaining three are still stopped.

Exit Codes

Each time you execute `tcsadmin`, the script returns an exit code reporting success or failure of the particular command you just ran. The script also outputs a descriptive status message to *stdout*.

The exit code for success is always `0`; the exit code for failure is a non-zero positive integer. Typically, the failure exit code is `1` if a single failure occurred. However, because some `tcsadmin` commands run against a tc Runtime group, the command might succeed for some tc Runtime instances in the group but fail for others. In this case, the exit code equals the number of failures.

The detailed reference information for each `tcsadmin` command includes the expected exit codes for that particular command.

Getting Help

Use the following commands to get help on a particular command or the list of available commands:

```
prompt$ ./tcsadmin.sh help
```

The `help` command lists all the available commands along with their parameters.

```
prompt$ ./tcsadmin.sh command --help
```

Use the `--help` parameter with any specific command to get detailed information about that command. The following example shows how to get detailed information about the `list-applications` command:

```
prompt$ ./tcsadmin.sh list-applications --help
```

Inventory Commands

This section documents the `tcsadmin` commands that you use to manage the HQ inventory. See also General Syntax of the tcsadmin Command-Line Interface.

- list-servers
- modify-server
- list-groups
- create-group
- add-server-to-group
- remove-server-from-group
- delete-group

Every time you run the `tcsadmin` script, provide connection parameters that specify the HQ Server to which the script connects. Specify these connection parameters at the command-line or in a *`user.home`*`/.hq/client.properties` file, where *`user.home`* refers to your home directory. In this section of the documentation, it is assumed that you have created a `client.properties` file, so that examples are not cluttered with the same connection parameters and are thus easier to read. See Connection Parameters for detailed information about specifying the connection parameters at the command-line or in a file.

list-servers

List all servers in the HQ inventory of type `SpringSource tc Runtime 6.0` and `SpringSource tc Runtime 7.0`.

The command outputs the following information for each tc Runtime instance, with each bit of information separated by a `|` character:

- The internal HQ ID of the tc Runtime instance. This ID is guaranteed to be unique across all resources in the HQ inventory. You can use this ID as a parameter value for other `tcsadmin` commands, such as remove-server-from-group, to uniquely identify a specific tc Runtime instance.
- The name of the tc Runtime instance.
- The description of the tc Runtime instance.
- The status of the tc Runtime instance, such as `Stopped` or `Running`.

Exit codes: Returns `0` if successful and `1` on failure to retrieve tc Runtime information.

Table 5.6. Options of the list-servers Command

Option	Description	Required?
--platformname	Particular platform for which you want the associated list of tc Runtime instances. If you do not specify this option, then the command returns *all* tc Runtime instances in the entire HQ inventory. To get the exact name of an HQ platform, invoke the HQ user interface in your browser, click the **Resources > Browse** link at the top, then click the **Platforms (X)** link; the platform names in the HQ inventory appear in the **Platform** column of the table.	No.
--groupname	Name of a tc Runtime `Compatible Group/Cluster` whose list of member tc Runtime instances you want to view. If you do not specify this option, then the command returns all tc Runtime instances associated with the specified platform or in the entire HQ inventory. To get the exact name of a tc Runtime group, see list-groups.	No.

This example uses `list-servers` to list all tc Runtime instances in the HQ inventory and shows sample output:

```
prompt$ ./tcsadmin.sh list-servers

10115|mydesktop tc Runtime myserver|/opt/vmware/vfabric-tc-server-standard/myserver|Running
```

This example lists the tc Runtime instances that are part of the group called `Group1`:

```
prompt$ ./tcsadmin.sh list-servers --groupname=Group1
```

modify-server

Change the name or description of a tc Runtime instance.

Exit codes: Returns `0` on success and `1` on failure to change the name or description of a tc Runtime instance.

Table 5.7. Options of the modify-server Command

Option	Description	Required?
--serverid	ID of the tc Runtime instance that you want to modify. See list-servers to get the internal IDs of the tc Runtime instances in the HQ inventory.	Yes.
--name	New name of the tc Runtime instance.	No.
--description	New description of the tc Runtime instance.	No.

This example changes the name and description of a tc Runtime instance with id `10007`:

```
prompt$ ./tcsadmin.sh modify-server --serverid=10007 --name="tc Runtime 1-XX" \
       --description="First tc Runtime running on XX computer"
```

Enter the command on one line; the example shows it on two lines for clarity.

list-groups

List all tc Runtime groups in the HQ inventory.

The groups are of type `Compatible Group/Cluster` that contain servers of type `SpringSource tc Runtime 6.0` and `SpringSource tc Runtime 7.0`.

The command outputs the following information for each tc Runtime group, with each bit of information separated by the `|` character:

- Internal HQ ID of the tc Runtime group. This ID is guaranteed to be unique across all resources in the HQ inventory.
- Name of the tc Runtime group.
- Description of the tc Runtime group.
- Location of the tc Runtime group.

Exit codes: Returns `0` on success and `1` on failure.

This command does not have any associated options.

This example lists all known tc Runtime groups, along with sample output:

```
prompt$ ./tcsadmin.sh list-groups

10001|tcs-group|A group of tc Runtime instances|San Francisco
10002|my-group|Another group of tc Runtime instances|Oakland
```

create-group

Create a `Compatible Group/Cluster` with resources of type `SpringSource tc Runtime 6.0` or `SpringSource tc Runtime 7.0`.

The `create-group` command creates an empty group; use add-server-to-group to add specific tc Runtime instances to the group.

Exit codes: Returns `0` on success and `1` on failure to create the group.

Table 5.8. Options of the list-groups Command

Option	Description	Required?
--name	Name of the tc Runtime group you want to create.	Yes.
--description	Description of the new group.	No.
--location	Physical location of the group's hardware. The location is for your own accounting purposes; it does not specify internal technical information for the HQ Server.	No.
--version	Allows only tc Runtime instances of the specified version to be added to the group.	No.

This example creates a new empty group of tc Runtime instances with the specified name, description, version, and location:

```
prompt$ ./tcsadmin.sh create-group --name=Group2 --description="A group of tc Runtime 7.0 instances" \
    --version=7.0 --location="Austin"
```

add-server-to-group

Add a tc Runtime instance to an existing tc Runtime group. The tc Runtime version must be compatible with the group version.

Exit codes: The command returns `0` on success and `1` on failure to add the tc Runtime instance to the group.

Table 5.9. Options of the add-server-to-group Command

Option	Description	Required?
--groupname	Name of the group to which you will add a new tc Runtime instance. Use list-groups to view the list of existing tc Runtime groups.	Yes.
--servername	Name of the tc Runtime instance you will add to the group. To get a list of tc Runtime instance names in the HQ inventory, use list-servers.	Specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance you will add to the group. To get a list of the tc Runtime instance names and IDs in the HQ inventory, use list-servers.	Specify `--servername` or `--serverid`, but not both.

This example adds the tc Runtime instance with name `example_server` to the existing tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin.sh add-server-to-group --groupname=Group1 --servername="example_server"
```

remove-server-from-group

Remove a tc Runtime instance from an existing group.

Exit codes: Returns `0` if successful and `1` on failure to remove a server from the group.

Table 5.10. Options of the remove-server-from-group Command

Option	Description	Required?
--groupname	Name of the group from which you will remove an existing tc Runtime instance member. Use list-groups to view the list of existing tc Runtime groups.	Yes.
--servername	Name of the tc Runtime instance you will remove from the group. Use list-servers to view the tc Runtime instance names and IDs in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance you will remove from the group. Use list-servers to view the tc Runtime instance names and IDs in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.

This example removes the tc Runtime instance with name `example_server` from the existing tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin.sh remove-server-from-group --groupname=Group1 --servername="example_server"
```

delete-group

Delete an existing tc Runtime group. Deleting a group does *not* delete the tc Runtime instances that make up the group.

Exit codes: Returns `0` on success and `1` on failure to delete the group.

Table 5.11. Options of the delete-group Command

Option	Description	Required?
--name	Name of the tc Runtime group that you want to delete.	Yes.

Option	Description	Required?
	Use list-groups to view the list of existing tc Runtime groups.	

This example deletes an existing tc Runtime group:

```
prompt$ ./tcsadmin.sh delete-group --name="Group1"
```

Application Management Commands

This section documents the `tcsadmin` commands that you use to manage applications deployed to a tc Runtime instance or group. See also General Syntax of the tcsadmin Command-Line Interface.

- list-applications
- deploy-application
- start-application
- stop-application
- reload-application
- undeploy-application

Every time you run the `tcsadmin` script, provide connection parameters that specify the HQ Server to which the script connects. Specify these connection parameters at the command-line or in a *`user.home`*`/.hq/client.properties` file, where *`user.home`* refers to your home directory. In this section of the documentation, it is assumed that you have created a `client.properties` file, so that examples are not cluttered with the same connection parameters and are thus easier to read. See Connection Parameters for detailed information about specifying the connection parameters at the command-line or in a file.

list-applications

List the applications deployed to a tc Runtime instance or group and get the current status of the applications.

Narrow the scope of the command by specifying a particular service (such as `Catalina`) or virtual host (such as `localhost`) of the tc Runtime instance or group. Use the names of the service or virtual host in the corresponding `server.xml` configuration file of the tc Runtime instance.

The `list-applications` command returns the following information for each deployed application, with each bit of information separated by the `|` character:

- Name of the service of the tc Runtime instance to which the application is deployed. This name corresponds to the service name in the `server.xml` file for the tc Runtime instance.
- Name of the virtual host of the tc Runtime instance to which the application is deployed. This name corresponds to the host name in the `server.xml` file for the tc Runtime instance.
- Name (context) of the deployed application.
- Version of the deployed application.
- Status of the application: `Running` or `Stopped`. If you are listing the deployed applications of a *group* of tc Runtime instances and the application is started on some tc Runtime instances and stopped on other instances, the status message is `Mixed`.
- Number of current active sessions for the application.

Exit codes: Returns `0` if it is completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to retrieve the information for the deployed applications. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure; if the command fails for

only some of the tc Runtime members of the group but succeeds for others, then the command returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.12. Options of the list-applications Command

Option	Description	Required?
--servername	Name of the tc Runtime instance for which you want to list the deployed applications. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When listing the applications deployed to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance for which you want to list the deployed applications. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When listing the applications deployed to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group for which you want to list the deployed applications. This option lists applications on each tc Runtime member of the group, even if the application is not consistently deployed to each member of the group. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When listing the applications deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group for which you want to list the deployed applications. This option lists applications on each tc Runtime member of the group, even if the application is not consistently deployed to each member of the group. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When listing the applications deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance for which you want to list the deployed applications. By default, tc Runtime instances are configured with the `Catalina` service. To get the names of the services of a tc Runtime instance, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, which lists the configured services. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements. If you do not specify this option, `list-applications` lists the applications in *all* services of the specified tc Runtime instance or group.	No.
--tchost	Name of the virtual host of the tc Runtime instance for which you want to list the deployed applications. By default, tc Runtime instances are configured with the `localhost` virtual host. To get the names of the virtual hosts of a tc Runtime instance, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements. If you do not specify this option, the `list-applications` command lists the applications in *all* virtual hosts of the specified tc Runtime instance or group.	No.
--application	Name of a Web application for which you want status information.	No.

This example lists the Web applications deployed to the a tc Runtime instance with name `example_server`, along with sample output. The command limits the scope of the list to the applications deployed to the `Catalina` service and `localhost` virtual host.

```
prompt$ ./tcsadmin list-applications --servername="example_server" --tcservice=Catalina --tchost=localhost
```

```
Catalina|localhost|ROOT|Running|0
Catalina|localhost|insight|Running|0
```

The next example shows how to list the Web applications deployed to all members of the `Group1` tc Runtime group. The output includes the applications that are deployed to all members of the group and applications deployed only to a single member of the group.

```
prompt$ ./tcsadmin list-applications --groupname=Group1
```

This example shows how to view the status of a specific Web application deployed to a single tc Runtime instance named `example_server`. If the application is not currently deployed to the server, you get an error.

```
prompt$ ./tcsadmin list-applications --application=swf-booking-mvc --servername="example_server"
```

deploy-application

Upload and deploy a Web application WAR file to a tc Runtime instance or group.

The WAR file can reside on the same computer on which you are running the `tcsadmin` command-line interface or on the remote computer on which the tc Runtime instance is running.

If you use this command to deploy a Web application to a group of tc Runtime instances and you specify a remote path for the WAR file, then it is assumed that the WAR file is located on *every* computer that hosts the members of the tc Runtime group and that the WAR file's directory pathname is identical on each computer. A good way to implement this is to mount a shared network drive on each computer that hosts a tc Runtime instance, then put the WAR file on the shared drive. Make sure that the name of the mounted shared drive is the same on each computer.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to deploy the application. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for only some of the tc Runtime members of the group but succeeds for others, then the command returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.13. Options of the deploy-application Command

Option	Description	Required?
--servername	Name of the tc Runtime instance to which you will deploy the Web application. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When deploying to a single tc Runtime instance, specify either `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance to which you will deploy the Web applications. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When deploying to a single tc Runtime instance, specify either `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group to which you will deploy the Web application. This option deploys the application to each member of the group. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When deploying an application to a tc Runtime group, specify either `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group to which you want to deploy the Web application. This option deploys the application to every single member of the group. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When deploying an application to a tc Runtime group, specify either `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance or group to which the Web application will be deployed. Default is `Catalina`.	No.

Option	Description	Required?
	To get the names of the services of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab which lists the configured services. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements.	
--tchost	Name of the virtual host of the tc Runtime instance or group to which the Web application will be deployed. Default value is `localhost`. To get the names of the virtual hosts of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements.	No.
--localpath	Full pathname of the Web application WAR file, local to the computer on which you are running the `tcsadmin` command-line interface.	Specify either `--localpath` or `--remotepath` but not both.
--remotepath	Full pathname of the Web application WAR file on the remote computer that hosts the tc Runtime instances (either single or members of a group.) If you are deploying the Web application to a group, it is assumed that the WAR file exists on each computer that hosts the members of the group and that the full pathname of the files is identical. You can put the WAR file on a shared network drive as long as the drive is mounted with the same name on each computer that hosts the tc Runtime instances.	Specify either `--localpath` or `--remotepath` but not both.
--contextpath	Context path of the Web application. The context path refers to the pathname used in the URL to invoke the Web application, relative to the tc Runtime instance's root URL. The default value of this option is the name of the WAR file without any file endings or pathnames. For example, if you deploy the local file `/home/myapps/myStore.war` then the default context path is `myStore`.	No.

The following example shows how to deploy an application whose WAR file is called `/home/myapps/swf-booking-mvc.war` and is located on the same computer on which you are running the `tcsadmin` command-line interface. The command deploys the application to all members of the tc Runtime group called `Group1`. Because no `--contextpath` option is specified, the context path used to invoke the deployed application will be `swf-booking-mvc`; the full URL will be something like `http://myhost:8080/swf-booking-mvc`.

```
prompt$ deploy-application --groupname=Group1 --localpath=/tmp/swf-booking-mvc.war
```

start-application

Start an application that is currently deployed to a tc Runtime instance or group.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to start the application. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime members of the group but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

The following table lists the options you can specify with this command.

Table 5.14. Options of the start-application Command

Option	Description	Required?
--servername	Name of the tc Runtime instance on which the Web application you want to start is deployed.	When starting an application deployed to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.

Option	Description	Required?
	Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	
--serverid	ID of the tc Runtime instance on which the Web application you want to start is deployed. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When starting an application deployed to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group on which the application you want to start is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When starting an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group on which the application you want to start is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When starting an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance or group where the application you want to start is deployed. Default value is `Catalina`. To get the names of the services of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, which lists the configured services. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements.	No.
--tchost	Name of the virtual host of the tc Runtime instance or group where the application you want to start is deployed. Default value is `localhost`. To get the names of the virtual hosts of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements.	No.
--application	Name of the Web application you want to start. It is assumed that the application has been previously deployed. If you do not specify this option, then `tcsadmin` starts all relevant applications, depending on the other options you have specified. Use list-applications to get the list of applications deployed to a tc Runtime instance or group.	No.
--revision	Specifies the particular revision of the application to start. This feature, available only on instances using Tomcat 7, allows you to deploy a new revision without interrupting the current version of the application. Existing user sessions continue using the current application version, but new sessions use the highest numbered revision that has been deployed.	No.

The example starts an application called `swf-booking-mvc` deployed to the tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin start-application --groupname=Group1 --application=swf-booking-mvc
```

stop-application

Stop an application that is currently deployed to a tc Runtime instance or group.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to stop the application. If you execute against a group of tc Runtime instances, the command

returns `1` for a general blanket failure; if the command fails for only some of the tc Runtime members of the group but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.15. Options of the stop-application Command

Option	Description	Required?
--servername	Name of the tc Runtime instance on which the Web application you want to stop is deployed. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When stopping an application deployed to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance on which the Web application you want to stop is deployed. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When stopping an application deployed to just a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group on which the application you want to stop is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When stopping an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group on which the application you want to stop is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When stopping an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance or group where the application you want to stop is deployed. Default is `Catalina`. To get the names of the services of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, which lists the configured services. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements.	No.
--tchost	Name of the virtual host of the tc Runtime instance or group where the application you want to stop is deployed. Default is `localhost`. To get the names of the virtual hosts of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements.	No.
--application	Name of the Web application you want to stop. It is assumed that the application has been previously deployed. If you do not specify this option, `tcsadmin` stops all relevant applications, depending on the other options you have specified. Use list-applications to get the list of applications deployed to a tc Runtime instance or group.	No.
--revision	Specifies the particular version of the application to stop. This feature is available only on instances using Tomcat 7. You can use the `stop-application` command with this option to stop an older version of the application after deploying the new version and ensuring that all sessions using the older version have ended.	No.

The example stops an application called `swf-booking-mvc` deployed to the tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin stop-application --groupname=Group1 --application=swf-booking-mvc
```

reload-application

Reload an application that is currently deployed to a tc Runtime instance or group.

Reloading refers to tc Runtime re-reading the WAR file into memory. Use this command if you have updated the original WAR file.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to reload the application. If you execute the command against a group of tc Runtime instances, it returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

You can execute the `reload-application` command on both started and stopped Web applications.

Table 5.16. Options of the reload-application Command

Option	Description	Required?
--servername	Name of the tc Runtime instance on which the Web application you want to reload is deployed. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When reloading an application deployed to just a single tc Runtime instance, you must specify either `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance on which the Web application you want to reload is deployed. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When reloading an application deployed to just a single tc Runtime instance, you must specify either `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group on which the application you want to reload is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When reloading an application deployed to a tc Runtime group, you must specify either `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group on which the application you want to reload is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When reloading an application deployed to a tc Runtime group, you must specify either `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance or group where the application you want to reload is deployed. Default is `Catalina`. To get the names of the services of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, which lists the configured services. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements.	No.
--tchost	Name of the virtual host of the tc Runtime instance or group where the application you want to reload is deployed. Default value is `localhost`. To get the names of the virtual hosts of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements.	No.
--application	Name of the Web application you want to reload. It is assumed that the application has been previously deployed. If you do not specify this option, `tcsadmin` reloads all relevant applications, depending on the other options you have specified.	No.

Option	Description	Required?
	Use list-applications to get the list of applications deployed to a tc Runtime instance or group.	
--revision	Specifies the particular version of the application to reload. This feature is available only on instances using Tomcat 7.	No.

The example reloads an application called `swf-booking-mvc` deployed to the tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin reload-application --groupname=Group1 --application=swf-booking-mvc
```

undeploy-application

Undeploy an application that is currently deployed to a tc Runtime instance or group.

When you undeploy an application, you essentially obliterate it from the tc Runtime environment. The only way to use the application again is by deploying it from scratch.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to undeploy the application. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If it fails for some tc Runtime members of the group but succeeds for others, the command returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.17. Options of the undeploy-application Command

Option	Description	Required?
--servername	Name of the tc Runtime instance on which the Web application you want to undeploy is currently deployed. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When undeploying an application deployed to just a single tc Runtime instance, you must specify either `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance on which the Web application you want to undeploy is currently deployed. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When undeploying an application deployed to just a single tc Runtime instance, you must specify either `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group on which the application you want to undeploy is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When undeploying an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group on which the application you want to undeploy is deployed. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When undeploying an application deployed to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--tcservice	Name of the service of the tc Runtime instance or group where the application you want to undeploy is deployed. Default is `Catalina`. To get the names of the services of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, and go to the **Views > Server Configuration > Services** tab, which lists the configured services. Or, in	No.

Option	Description	Required?
	the tc Runtime instance's `server.xml` configuration file, look for `<Service>` elements.	
--tchost	Name of the virtual host of the tc Runtime instance or group where the application you want to undeploy is deployed. Default is `localhost`. To get the names of the virtual hosts of a tc Runtime instance or group, invoke the HQ user interface in a browser, navigate to the tc Runtime instance, go to the **Views > Server Configuration > Services** tab, click on a service in the table, and then click on the **Hosts** link in the left. Or, in the tc Runtime instance's `server.xml` configuration file, look for `<Host>` elements.	No.
--application	Name of the Web application you want to undeploy. It is assumed that the application has been previously deployed. If you do not specify this option, `tcsadmin` undeploys all relevant applications, depending on the other options you have specified. Use list-applications to get the list of applications deployed to a tc Runtime instance or group.	No.
--revision	Specifies the particular version of the application to undeploy. This feature is available only on instances using Tomcat 7. You can use the `undeploy-application` command with this option to undeploy an older version of the application after deploying the new version and ensuring that all sessions using the older version have ended.	No.

The example undeploys an application called `swf-booking-mvc` currently deployed to the tc Runtime group called `Group1`:

```
prompt$ ./tcsadmin undeploy-application --groupname=Group1 --application=swf-booking-mvc
```

tc Runtime and Group Configuration Commands

This section documents the `tcsadmin` commands that you use to configure tc Runtime instances and groups. See also General Syntax of the tcsadmin Command-Line Interface.

- get-file
- put-file
- list-jvm-options
- set-jvm-options

Every time you run the `tcsadmin` script, provide connection parameters that specify the HQ Server to which the script connects. Specify these connection parameters at the command-line or in a *`user.home`*`/.hq/client.properties` file, where *`user.home`* refers to your home directory. In this section of the documentation, it is assumed that you have created a `client.properties` file, so that examples are not cluttered with the same connection parameters and are thus easier to read. See Connection Parameters for detailed information about specifying the connection parameters at the command-line or in a file.

get-file

Get a configuration file from a single tc Runtime instance and write it to a file on the local computer on which you are running the `tcsadmin` script.

Although you can get any configuration file, typically you work only with the following files, relative to the `CATALINA_BASE/conf` directory:

- `server.xml`
- `web.xml`
- `context.xml`

Exit codes: Returns `0` if successful and `1` on failure.

Table 5.18. Options of the get-file Command

Option	Description	Required?
--servername	Name of the tc Runtime Instance from which you want to get a configuration file. Use list-servers to get the names of the tc Runtime Instances in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance from which you want to get a configuration file. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.
--file	Name of the configuration file that you want to get from the tc Runtime instance, relative to its `CATALINA_BASE` directory. Typical values include `conf/server.xml`, `conf/catalina.properties`, `conf/web.xml`, or `conf/context.xml`. Use forward or backward slash to specify directories; the slashes are resolved on the corresponding computer platform on which the command is eventually run.	Yes.
--targetfile	Name of the file, local to the computer on which you are running the `tcsadmin` script, to which you want the retrieved configuration to be written. If you enter a relative pathname, it is local to the working directory of the script, which by default is the top-level installation directory of the script, such as `/usr/springsource-tcserver- scripting-client-2.1.X.RELEASE`.	Yes.

The example gets a `server.xml` configuration file from a tc Runtime instance with ID `10045` and writes it to a local file called `/tmp/group-server.xml`:

```
prompt$ ./tcsadmin get-file --serverid=10045 --targetfile=/tmp/group-server.xml --file=conf/server.xml
```

put-file

Push a single tc Runtime instance configuration file to a tc Runtime instance or group of servers.

In this context, the term *push* refers to copying any configuration file to the tc Runtime instance directory. Although you can push any configuration file, typically you work only with the following files, relative to the `CATALINA_BASE/conf` directory:

- `server.xml`
- `web.xml`
- `context.xml`

- `catalina.properties`

When you execute this command to push a new copy of a particular configuration file, the `tcsadmin` script creates a backup of the existing file in the corresponding `conf` directory with the name *`file_name`*`.yyyy-mm-dd.hh-mm-ss` to facilitate manual rollback. If an error occurs writing any of the files, the `tcsadmin` script rolls back all file writes. If you execute this command against a group, this file backup happens for each member of the group. You can disable the creation of the backup file with the `--nobackupfile` option.

When you push a configuration file to a group of tc Runtime instances, SpringSource highly recommends that you use templated files. This means that the file uses variables for configuration values that vary from server to server, such as the HTTP listen port; then, the `catalina.properties` file for each server contains the actual values which tc Runtime substitutes for the variables at runtime. The default `server.xml` file for tc Runtime instances uses templates, which means that you can use the get-file command to get a `server.xml` file for a particular tc Runtime instance and use it as a template for pushing configuration changes to a group, or even another tc Runtime instance. You can also use this command to push individual `catalina.properties` to each tc Runtime instance.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to push the configuration file. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, the command returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.19. Options of the put-file Command

Option	Description	Required?
--servername	Name of the tc Runtime instance to which you want to push a configuration file. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When pushing a configuration file to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance to which you want to push a configuration file. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When pushing a configuration file to a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Specifies the name of the tc Runtime group to which you want to push a configuration file. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When pushing a configuration file to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	Specifies the ID of the tc Runtime group to which you want to push a configuration file. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When pushing a configuration file to a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--targetfile	Name of the configuration file that you want to update. Typical values include `conf/server.xml`, `conf/catalina.properties`, `conf/web.xml`, or `conf/context.xml`. Use a forward or backward slash to specify directories; the slashes are resolved on the corresponding computer platform on which the command is eventually run.	Yes.
--file	Name of the file, local to the computer on which you are running the `tcsadmin` script, that you want to push to the tc Runtime instance or group. If you enter a relative pathname, it is local to the working directory of the script, which by default is the top-level installation directory of the script, such as `/usr/springsource-tcserver- scripting-client-2.1.X.RELEASE`.	Yes.
--nobackupfile	Specifies that the `tcsadmin` script should NOT create a backup of the existing configuration file before the script pushes the new file to the tc	No.

Option	Description	Required?
	Runtime instance. The default behavior if you do not specify this option is for the script to always create a backup of the file. This option is useful if, for example, you are pushing a file to the `webapps` directory. If the script were to create a backup copy of the file, the tc Runtime instance would then try to deploy both the backup and the new file.	

The example pushes a `server.xml` configuration file to all members of the tc Runtime group called `Group1`; the file that is pushed is located on the local computer and is called `/tmp/group-server.xml`:

```
prompt$ ./tcsadmin put-file --groupname=Group1 --file=/tmp/group-server.xml --targetfile=conf/server.xml
```

list-jvm-options

List all the Java virtual machine (JVM) options that are currently set for a single tc Runtime instance.

The command gets the currently set JVM options from the following locations:

- Unix: The `JVM_OPTS` variable set in the `bin/setenv.sh` file.
- Windows: The `wrapper.java.additional.x` property in the `conf/wrapper.conf` file.

The command returns each JVM option on a single line; for example:

```
prompt> ./tcsadmin list-jvm-options --servername="example_server"
-Xmx512m
-Xms128m
-Xss=192k
-Dmy.custom.prop=foo
```

Exit codes: Returns `0` if successful and `1` on failure.

Table 5.20. Options of the list-jvm-options Command

Option	Description	Required?
--servername	Name of the tc Runtime instance from which you want to list the currently-set JVM options. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.
--serverid	Specifies the ID of the tc Runtime instance from which you want to list the currently-set JVM options. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	Specify `--servername` or `--serverid`, but not both.

The example gets the JVM options that are currently set for a tc Runtime instance with ID `10045`:

```
prompt$ ./tcsadmin list-jvm-options --serverid=10045
```

set-jvm-options

Modify the JVM options for a tc Runtime instance or a group of tc Runtime instances.

The command sets the JVM options by updating the following files on the target tc Runtime instance or instances:

- Unix: The `JVM_OPTS` variable in the `bin/setenv.sh` file.
- Windows: The `wrapper.java.additional.x` property in the `conf/wrapper.conf` file.

Warning: The `set-jvm-options` command *overwrites* any existing JVM options; it does not add to existing options. For example, if you have previously set the `-Xmx512m` and `-Xss192k` JVM options for the tc Runtime instance, and then you execute the following `set-jvm-options` command:

```
prompt$ ./tcsadmin set-jvm-options --options=-Xms384m --serverid=10045
```

only the `-Xms384m` JVM option will be set; the `-Xss192k` option is no longer set.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to set the JVM options. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.21. Options of the set-jvm-options Command

Option	Description	Required?
--servername	Name of the tc Runtime instance whose JVM options you want to set. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When running this command against a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance whose JVM options you want to set. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When running this command against a single tc Runtime instance, specify`--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group whose JVM options you want to set. Each tc Runtime instance member will be updated. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When running this command against a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	Specifies the ID of the tc Runtime group whose JVM options you want to set. Each tc Runtime instance member will be updated. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When running this command against a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--options	Comma-separated list of JVM options that you want to set. If an option value itself contains a comma, make sure you escape the comma by preceding it with a backslash (\). For example, the option `--options="-Xmx512m, -Dsomeprop=value1\,value2, -Xms128m"` resolves into three options: `-Xmx512m`, `-Dsomeprop=value1,value2` and `-Xms128m`. To remove all JVM options, use the following syntax: `--option=""`.	Yes.

The example sets the initial Java heap size (using `-Xms`) and the maximum Java heap size (using `-Xmx`) for each tc Runtime instance in the group called `Group1`:

```
prompt$ ./tcsadmin set-jvm-options --groupname=Group1 --options=-Xms512m,-Xmx1024m
```

tc Runtime and Group Control Commands: Reference

This section documents the `tcsadmin` commands that you use to control tc Runtime instances and groups. See also General Syntax of the tcsadmin Command-Line Interface.

- start
- stop
- restart

Each section describes the command, provides the list of available options, and provides usage examples.

Every time you run the `tcsadmin` script, provide connection parameters that specify the HQ Server to which the script connects. Specify these connection parameters at the command-line or in a *`user.home`*`/.hq/client.properties` file, where *`user.home`* refers to your home directory. In this section of the documentation, it is assumed that you have created a `client.properties` file, so that examples are not cluttered with the same connection parameters and are thus easier to read. See Connection Parameters for detailed information about specifying the connection parameters at the command-line or in a file.

start

Start a tc Runtime instance or all the members of a tc Runtime group.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to start the server. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.22. Options of the start Command

Option	Description	Required?
--servername	Name of the tc Runtime instance that you want to start. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When starting a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance that you want to start. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When starting a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group that you want to start. When you specify a tc Runtime group for this command, *all* its members are started. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When starting the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group that you want to start. When you specify a tc Runtime group for this command, *all* its members are started. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When starting the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.

The example starts the tc Runtime instance named `example_server`.

```
prompt$ ./tcsadmin start --servername="example_server"
```

stop

Stop a tc Runtime instance or all the members of a tc Runtime group.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to start the server. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

Table 5.23. Options of the stop Command

Option	Description	Required?
--servername	Name of the tc Runtime instance that you want to stop. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When stopping a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.

Option	Description	Required?
--serverid	ID of the tc Runtime instance that you want to stop. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When stopping a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group that you want to stop. When you specify a tc Runtime group for this command, *all* its members are stopped. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When stopping the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group that you want to stop. When you specify a tc Runtime group for this command, *all* its members are stopped. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When stopping the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.

The following example shows how to stop the tc Runtime instance named `example_server`.

```
prompt$ ./tcsadmin stop --servername="example_server"
```

restart

Restart a tc Runtime instance or all the members of a tc Runtime group.

A restart is a shorthand way of stopping and starting the server or group.

Exit codes: Returns `0` if completely successful. If you execute the command against a single tc Runtime instance, the command returns an exit code of `1` if it failed to start the server. If you execute the command against a group of tc Runtime instances, the command returns `1` for a general blanket failure. If the command fails for some tc Runtime group members but succeeds for others, it returns an exit code of the number of failures and outputs to *stdout* one line for each failure, along with the reason.

The following table lists the options you can specify with this command.

Table 5.24. Options of the restart Command

Option	Description	Required?
--servername	Name of the tc Runtime instance that you want to restart. Use list-servers to get the names of the tc Runtime instances in the HQ inventory.	When restarting a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--serverid	ID of the tc Runtime instance that you want to restart. Use list-servers to get the IDs of the tc Runtime instances in the HQ inventory.	When restarting a single tc Runtime instance, specify `--servername` or `--serverid`, but not both.
--groupname	Name of the tc Runtime group that you want to restart. When you specify a tc Runtime group for this command, *all* its members are restarted. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When restarting the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.
--groupid	ID of the tc Runtime group that you want to restart. When you specify a tc Runtime group for this command, *all* its members are restarted. Use list-groups to get the names and IDs of the tc Runtime groups in the HQ inventory.	When restarting the servers in a tc Runtime group, specify `--groupname` or `--groupid`, but not both.

The example shows how to restart the tc Runtime instance named `example_server`:

```
prompt$ ./tcsadmin restart --servername="example_server"
```

6. Creating tc Runtime Templates

A template provides configuration information and files to support a feature or application on a tc Runtime instance. The built-in templates that ship with tc Server make it simple to configure tc Runtime features such as SSL or JMX or to add a management application to an instance at creation time, such as Spring Insight.

You can create your own templates by creating a subdirectory in the `templates` directory of your tc Server installation directory and populating it with files according to the instructions in this section. You could, for example, construct a template that allows creating a tc Runtime instance with a web application or set of web applications ready to deploy, with a custom configuration specified at the `tcruntime-instance create` command line or through interactive prompts.

A template is a directory containing files that the `tcruntime-instance create` command processes when it creates a new tc Runtime instance. Some files are copied directly to the new tc Runtime instance. Other files are applied to configuration files in the tc Runtime instance; that is, they are used to alter the content of standard configuration files, such as `conf/server.xml`.

Files you place in the template directory that are not interpreted specially by the instance creation scripts are copied into the new instance. For example, if your web application requires JAR libraries, you can create a `lib` subdirectory and place the JAR files there. If you have a WAR file to deploy, put it in a `webapps` subdirectory and it will be copied to the `webapps` subdirectory of the new tc Runtime instance.

The target platform (Windows or Unix) and the JVM (Sun HotSpot or IBM J9) are recognized at instance creation time and variables are handled accordingly, files omitted from the copy when appropriate. Your Linux tc Runtime instances will not have unneeded `.bat` or `.dll` files. Path names and environment variables are automatically handled with the correct syntax for the target platform.

Parts of a Template

A template directory contains at minimum a `README.txt` file. The other contents depend on the purpose of the template. The following sections describe the kinds of files that a template can have.

README.txt

A template must have a `README.txt` file in its root directory. This file is a synopsis of the configuration and content that the template provides to an instance. The file should not have the name of the template, but a version and build date are considered best practices. When in doubt, look at the examples provided by the templates packaged in tc Runtime.

When an instance is created, the content of the `README.txt` files in each template are combined into a single `README.txt` file that is placed in the root of the created instance. The combined `README.txt` file documents the templates' contributions to the newly created instance.

Following is the `README.txt` file that is the result of creating an instance using the `base`, `bio`, `bio-ssl`, and `elastic-memory` templates.

```
Operating System Family: unix
Virtual Machine Architecture: x64
Virtual Machine Name: hotspot
================================================================================================================================
```

```
Template: base
Version: 2.6.0.RELEASE
Build Date: 20110729092530

* Sets Xmx to 512M
* Sets Xss to 192K
* Adds a control script to the instance
* Adds the Windows service wrapper libraries
* Adds a default jmxremote configuration with a read/write user called 'admin' with a password of 'springsource'
* Adds a default JULI logging configuration
* Adds a default server configuration containing:
 * A JRE memory leak prevention listener
 * A tc Runtime Deployer listener
 * A JMX socket listener
 * A LockOutRealm to prevent attempts to guess user passwords via a brute-force attack
 * An in-memory user database
 * A threadpool that has up to 300 threads
 * A host that uses 'webapps' as its app base
 * An AccessLogValve
* Adds a default Tomcat user configuration that is empty
* Adds an init.d script configured to start the instance as a specific user
* Adds a root web application
================================================================================================================
Template: base-tomcat-7
Version: 2.6.0.RELEASE
Build Date: 20110729092530

* Adds Tomcat 7-specific ThreadLocalLeakPreventionListener
* Adds Tomcat 7-specific catalina.properties
* Adds Tomcat 7-specific default catalina.policy to be used when starting with the -security option
* Adds Tomcat 7-specific JspServlet configuration
* Adds Tomcat 7-specific web-app declaration
================================================================================================================
Template: bio
Version: 2.6.0.RELEASE
Build Date: 20110729092530

* Adds a Blocking IO (BIO) connector for HTTP
================================================================================================================
Template: bio-ssl
Version: 2.6.0.RELEASE
Build Date: 20110729092530

* Adds a Blocking IO (BIO) connector for HTTPS
* Adds sample certificate and key files that can be used to test the SSL configuration

NOTE: The sample certificate and key files are not suitable for production systems.
================================================================================================================
Template: elastic-memory
Version: 1.0.0.RELEASE
Build Date: 20110729095358

* Adds Elastic Memory for Java version 82906b25cc97bb5d799578d0114921a35c26e41b to launch configuration
* Sets Xmx to 1024M
* Sets up EM4J logging in conf/logging.properties
```

Environment

A template may contribute a `bin/setenv.properties` file containing platform-agnostic environmental configuration. This file is turned into `bin/setenv.sh` on Unix machines and `bin/setenv.bat` and `conf/wrapper.conf` files on Windows machines. The file may contain properties with any of the following well-known keys.

Table 6.1. setenv.properties Keys

Key	Description
`java.home`	The directory where the JVM is installed.

Key	Description
`java.agent.#`	The path to the Eleastic Memory for Java native library. It is added to the JVM command line with the `-agentpath` option.
`agent.path.#`	The path to the Elastic Memory for Java JAR file. I tis added to the JVM command line with the `-javaagent` option.
`class.path.#`	Adds a JAR to the Java class path.
`java.library.path.#`	The path to a native library. It is added to the `java.library.path` in the JVM command command line.
`java.opt.#`	A JVM option to be added to the JVM command line.

Except for `java.home`, each of these keys can be declared multiple times by incrementing its digit suffix. An example declaring two entries for `java.library.path` follows.

```
java.library.path.1=${catalina.base}/bin/amd64-linux
java.library.path.2=${vmware.tools.location:/usr/lib/vmware-tools}/lib/libvmGuestLib.so
```

Automatic Boilerplate Decoration

Entries for the `setenv.properties` keys do not need to have boilerplate text attached. When the template is processed, the values are processed to create command line options with the correct platform- and JVM-specific syntax. The following table describes what will be prepended to each entry.

Table 6.2. Automatic Boilerplate Decoration

Entry	Entry After Decoration
`java.agent.1=`*`value-1`* `java.agent.2=`*`value-2`*	`-javaagent:`*`value-1`* `-javaagent:`*`value-2`*
`agent.path.1=`*`value-1`* `agent.path.2=`*`value-2`*	`-agentpath:`*`value-1`* `-agentpath:`*`value-2`*
`class.path.1=`*`value-1`* `class.path.2=`*`value-2`*	*`value-1`*`:`*`value-2`*
`java.library.path.1=`*`value-1`* `java.library.path.2=`*`value-2`*	`-Djava.library.path=`*`value-1`*`:`*`value-2`*

Memory and Stack Size JAVA_OPTS Filtering

There are a few common properties that are regularly set to control memory and stack size of the VM. In cases where duplicate values for these are found due to the combination of templates, the largest value of each will be chosen. The list of these properties follows.

- `-Xmx`
- `-Xms`
- `-Xss`
- `-XX:MaxPermSize`

JVM Type Specific Properties

To ensure that a property is only used for a specific JVM type, the well-known keys can be qualified with values of the `vm.name` property. The value must be located between the base key and the incrementing digit, delimited by '`.`' characters. For example:

```
java.opt.hotspot.1=+XX:MaxPermSize=1024M
java.opt.j9.2=-Xaggressive
```

OS Family Specific Properties

To ensure that a property is only used for a specific operating system family, the well-known keys can be qualified with values of the `os.family` property. The value must be located between the base key and the incrementing digit, delimited by '`.`' characters. An example using the `os.family` property follows.

```
java.library.path.unix.1=${vmware.tools.location:/usr/lib/vmware-tools}/lib/libvmGuestLib.so
java.library.path.windows.2=${vmware.tools.location:C:\Program Files\VMware\VMware Tools}
```

VM Architecture Specific Properties

To ensure that a property is only used for a specific VM architecture, the well-known keys can be qualified with values of the `vm.arch` property. The value must be located between the base key and the incrementing digit, delimited by '`.`' characters. An example using the `vm.arch` property follows.

```
java.library.path.unix.x64.1=${catalina.base}/bin/amd64-linux
java.library.path.unix.x86.2=${catalina.base}/bin/x86-linux
```

Combining Values in Qualified Properties

The well-known keys can be qualified with values of any combination of the implicit properties. These values must be located between the base key and the incrementing digit, delimited by '`.`' characters, but can be in any order. An example using the `os.family`, `vm.arch`, and `vm.name` properties follows.

```
java.library.path.unix.x64.hotspot.1=${catalina.base}/bin/amd64-linux
```

XML Configuration Fragments

A template may contribute any of the following XML configuration files.

- `conf/server-fragment.xml`
- `conf/web-fragment.xml`
- `conf/context-fragment.xml`
- `conf/tomcat-users-fragment.xml`

These files contribute to the standard Tomcat configuration file of the same name, less the "`-fragment`" portion of the name. Inside the file is an XML fragment that describes what is to be added, removed, or updated in the respective configuration file. The XML fragment describes its contributions using the `add:` and `remove:` keywords on elements and attributes and the `update:` keyword, which can only be used on attributes. In addition, other XML elements are defined to describe a single XML element that the contributions should act upon. The XML elements that exist can be thought of as a direct example of an XPath expression. For example the XPath expression `//Server/Service[@name="Catalina"]` would be represented as follows.

```
<?xml version='1.0' encoding='utf-8'?>
<Server>
  <Service name="Catalina">
  </Service>
</Server>
```

A more complex example of the XPath expression `//Server/Service[@name="Catalina"]/Engine[@name="Catalaina"][@defaultHost="localhost"]` is represented as follows.

```
<?xml version='1.0' encoding='utf-8'?>
<Server>
```

```
  <Service name="Catalina">
    <Engine name="Catalina" defaultHost="localhost">
    </Engine>
  </Service>
</Server>
```

Once an element has been specified using an XML fragment, contributions can then be specified. They could be updates and additions of attributes, as illustrated in the following example.

```
<?xml version='1.0' encoding='utf-8'?>
<Server>
  <Listener className="com.springsource.tcserver.serviceability.rmi.JmxSocketListener"
      update:useSSL="true"
      add:useJdkClientFactory="true"
      passwordFile="${catalina.base}/conf/jmxremote.password"
      accessFile="${catalina.base}/conf/jmxremote.access"
      add:keystoreFile="${catalina.base}/conf/tcserver.keystore"
      add:keystorePass="changeme"
      add:truststoreFile="${catalina.base}/conf/tcserver.keystore"
      add:truststorePass="changeme"
      update:authenticate="false"/>
</Server>
```

When adding an element, once the element has been marked as `add:`, it is unnecessary to also mark the attributes of the new element. An example of adding an element without marking its attributes follows.

```
<?xml version='1.0' encoding='utf-8'?>
<Server>
  <Service name="Catalina">
    <add:Connector executor="tomcatThreadPool"
        port="${http.port:8080}"
        protocol="org.apache.coyote.http11.Http11Protocol"
        connectionTimeout="20000"
        redirectPort="${https.port:8443}"
        acceptCount="100"
        maxKeepAliveRequests="15"/>
  </Service>
</Server>
```

It is unnecessary to mark any sub-elements with `add:` when the parent element is marked. An example adding an element with sub-elements without marking its sub-elements follows.

```
<?xml version='1.0' encoding='utf-8'?>
<Server>
  <Service name="Catalina">
    <Engine name="Catalina" defaultHost="localhost" add:jvmRoute="${node.name:tc-runtime-1}">
      <add: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.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"/>
      </add:Cluster>
    </Engine>
  </Service>
</Server>
```

Logging Properties Fragment

A template may contribute a `conf/logging-fragment.properties` file. This file contributes to the standard Tomcat `conf/logging.properties` file. The properties fragment describes its contributions by prefixing property keys with the `add.` keyword, as shown in the following example.

```
############################################################
# For Elastic Memory for Java (EM4J) logging.
#
# Values for com.vmware.jem.level are:
# WARNING, INFO, CONFIG, FINE, FINER, FINEST
#
############################################################
add.com.vmware.jem.level=${loggingLevel:INFO}
add.com.vmware.jem.handlers=java.util.logging.ConsoleHandler
add.java.util.logging.ConsoleHandler.formatter=com.vmware.jem.BalloonLogFormatter
```

Other Files

Any other file in the template that is not specifically excluded (see Platform Specificity) is copied directly to the instance. Properties files have their content substituted when copied. If a file clashes with a file contributed by another template, a warning is displayed to the user and the later file will replace the earlier file. Ordering of template application is dependent on user input and may vary.

Property Substitution

Property substitution allows customizing tc Runtime instances by providing instance-specific values at creation time. The `tcruntime-instance` script scans for property place holders in files and substitutes a value that is derived from a default or another defined property, or supplied interactively by the user when the script is run with the `--interactive` option. Property substitution occurs in the `bin/setenv.properties` file, XML configuration fragments, the logging properties fragment, and all `.properties` files.

The syntax for a place holder is as follows:

```
${property-name[:default-value]}
```

Implicit Properties

Templates are provided a set of implicit properties, determined at instance creation time. They are generally specific to the platform where the instance is created and the `JAVA_HOME` the instance will use at runtime. The list of implicit properties and their possible values are shown in the following table.

Table 6.3. Implicit Properties

Property	Possible Values
`os.family`	• `unix` • `windows`
`vm.arch`	• `x64`

Property	Possible Values
	• x86
vm.name	• hotspot • j9
catalina.base	• $CATALINA_BASE • %CATALINA_BASE%
catalina.home	• $CATALINA_HOME • %CATALINA_HOME%

Configuration Prompts

When a user runs the instance creation script in interactive mode, the script prompts for any property not specified as part of the command. The standard prompt is `Please enter a value for '%s'. Default '%s':` when a default is provided and `Please enter a value for '%s':` when no default is provided. These prompts are generic and not good at helping the user select a useful value. You can provide more helpful custom prompt text. To do this, a template must contain a resource bundle called `configuration-prompts.properties` in the root of the template. This bundle contains the text to display when prompting for a value. In addition, the prompt can include the default value for the property by embedding the `${default}` place holder in the text. For example:

```
vmware.tools.location=Enter the path to the VMware tools installation. The default path is '${default}'\:
```

The template user accepts the default by pressing Enter without entering a value.

Configuration prompts can be localized for particular languages and countries. To do this, append language and country codes to the file name. For example, a resource bundle containing localized prompts for Spanish speakers would be called `configuration-prompts_es.properties`.

Platform Specificity

When a tc Runtime instance is created, some files are not created or copied to the instance because they are not required by the target platform. For example, there is no benefit to copying Windows `.bat` files to a Linux host. In addition, some files are used by the template, or to document the template, and are not copied into the instance.

Files Excluded on Unix

Files Excluded on Windows

Template Files Excluded from All Instances

Files Excluded on Unix

When a template is rendered on a Unix platform, Windows platform-specific files are not rendered in the instance. This includes files matched by the following patterns:

- `**/*.bat`
- `**/*.dll`
- `**/*.exe`
- `**/amd64-winnt/**`
- `**/x86-winnt/**`
- `**/win32/**`

- `**/winx86_64/**`

Files Excluded on Windows

When a template is rendered on a Windows platform, Unix platform-specific files are not rendered in the instance. This includes files matched by the following patterns:

- `**/*.sh`
- `**/*.so`
- `**/amd64-linux/**`
- `**/x86-linux/**`

Template Files Excluded

Files matching the following patterns are not copied directly into a tc Runtime instance:

- `README.txt`
- `bin/setenv.properties`
- `conf/*-fragment.properties`
- `conf/*-fragment.xml`
- `configuration-prompts(_([A-Za-z])+)?.properties`

Splitting a Template for Tomcat Versions

The `base` template is an example of a template that provides different options depending on whether the target instance uses Tomcat 6 or Tomcat 7. This is a generalized feature that you can use if you have different configuration options or file contributions for Tomcat 6 and Tomcat 7.

The `base` template has three parts:

- `base` – The files in this directory are processed for both Tomcat 6 and Tomcat 7 instances.
- `base-tomcat-6` – The files in this directory are processed only only if the target instance uses a Tomcat 6 runtime.
- `base-tomcat-7` – The files in this directory are processed only if the target instance uses a Tomcat 7 runtime.

You can create a custom template with different options for Tomcat 6 and Tomcat 7 by using the same directory naming convention.

7. Enabling Clustering for High Availability

Clustering refers to grouping one or more tc Runtime instances so that they appear to work as a single server. A cluster provides:

- **Session replication**. When a client, typically using a browser, connects to a tc Runtime instance, tc Runtime creates a session Object that it uses to manage all subsequent interaction between itself and that client. Depending on how the Web application was programmed, the session Object can contain a lot of useful information, such as user security credentials, current items in a user's shopping cart, and so on. If the tc Runtime instance is part of a cluster, the session is automatically copied to each member of the cluster group, and is updated each time the session is modified, such as when the user adds a new item to their shopping cart. This means that if the first tc Runtime instance crashes, any tc Runtime instance in the group can immediately take over the session without interruption, completely hiding the server crash from the client who continues to work as if nothing had happened. This capability greatly increases the usability of Web applications.

  You can use the VMware vFabric™ GemFire® HTTP Session Management Module to provide HTTP session management for a tc Server cluster. The module provides tc Server templates to configure GemFire session management in either a peer-to-peer configuration or client/server configuration. In the peer-to-peer configuration, each tc Runtime instance becomes a GemFire peer, using multicast to discover each other and replicating session data between them. In the client-server configuration, you run a GemFire cache server and tc Runtime instances replicate session data to the cache server. See the GemStone documentation for help obtaining the templates and configuring GemStone HTTP Session Management.

- **Context attribute replication**. A context represents a Web application that is deployed to a tc Runtime instance. In the same way that client sessions can be replicated, the Web application context itself can also be replicated to all members of a cluster group.

A tc Runtime cluster can be as small as two server instances hosted on the same computer to hundreds of tc Runtime instances hosted on many different computers of different operating systems.

Typically, you configure a tc Runtime cluster to use multicast for the communication between member servers. The cluster is then uniquely identified by the combination of its multicast IP address and port. Each member of the cluster must have the same multicast address and port configured so that the cluster can automatically discover each member and react appropriately if a member does not respond. You can create multiple clusters, such as one for testing and another for production, by configuring different multicast address/ports for each cluster.

In addition to creating a tc Runtime cluster, you might also want to configure a load balancer in front of the cluster so as to split up the incoming requests between multiple tc Runtime instances. Load balancing attempts to direct requests to the tc Runtime instance with the smallest load at that point in time. The load balancer can also detect when a tc Runtime instance has failed, in which case it stops directing requests to it until the tc Runtime instance restarts, adding to the high availability of tc Runtime. You can use VMware vFabric Web Server to provide load balancing for tc Server. See "Configuring Load Balancing Between Two or More tc Runtime Instances" in *vFabric Web Server Installation and Configuration* for instructions.

See High Level Steps for Creating and Using tc Runtime Clusters for the basic steps to get started with tc Runtime clusters.

Additional Cluster Documentation from Apache

For additional information about configuring tc Runtime clusters, see:

- Clustering/Session Replication HOW-TO
- Configuration Reference for the Cluster Object

High-Level Steps for Creating and Using tc Runtime Clusters

The following procedure outlines the main tasks you perform to create and configure a tc Runtime cluster from two or more tc Runtime instances.

It is assumed that you have already created the individual tc Runtime instances, on one computer or distributed over multiple computers. If this is not the case, see "Creating a New tc Runtime Instance" in *Getting Started with vFabric tc Server*.

1. Prepare your Web applications so they can be deployed to a cluster and take full advantage of the tc Runtime clustering features. See Web Application Requirements for Using Session Replication.
2. Be sure that you have correctly configured your network to enable multicast, which is the typical method of communication between cluster members. See Network Considerations.
3. Configure your tc Runtime instances into a simple cluster using the default values for most of the configuration options. See Configuring a Simple tc Runtime Cluster.
4. If the default configuration does not suit your needs, configure other cluster configuration options. See Advanced Cluster Configuration Options.
5. Start your cluster by starting all the tc Runtime instances that make up the cluster group. You can do this manually, as described in "Starting and Stopping tc Runtime Instances" in *Getting Started with vFabric tc Server*, or by using the HQ User Interface.

Web Application Requirements for Using Session Replication

In addition to configuring the cluster from a server administration point of view, make sure your Web application meets these requirements:

- All servlet and JSP session data must be serializable. In Java terms, this means that every field in the session object must either implement the `java.io.Serializable` interface or it must be `transient`.
- tc Runtime uses cookies to track session state, which means that the Web application URLs for a particular session always look the same. If they do not, the tc Runtime instance creates a new session each time a client sends a message, which essentially disables session replication for that Web application.
- Configure your Web application to be *distributable*, that is, suitable for running in a distributed environment such as a tc Runtime cluster. You can do this in one of two ways:
  - Add the `<distributable />` element to the *web.xml* deployment descriptor of your Web application; `<distributable />` is a child-element of the root `<web-app>` element. The `web.xml` file is located in the `WEB-INF` directory of your Web application. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <distributable />

    <display-name>HelloWorld Application</display-name>
    <servlet>
    ...
<web-app>
```

  - If you do not want to change the `web.xml` deployment descriptor file of your Web application, you can use the tc Runtime-specific `<Context distributable="true">` element to specify that one or all Web applications are distributable. You can specify this element in the `CATALINA_BASE/conf/context.xml` file if you want to make ALL Web applications of a particular tc Runtime instance distributable. For example:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<Context distributable="true" >
...
</Context >
```

    You can also add this element to specific context files to narrow its scope. For details, see The Context Container.

- To enable application context replication, specify that your application context use the `org.apache.catalina.ha.context.ReplicatedContext` context implementation, rather than the default (`org.apache.catalina.core.StandardContext`). As described in the preceding bullet, you can update the `CATALINA_BASE/conf/context.xml` file as shown:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Context distributable="true" className="org.apache.catalina.ha.context.ReplicatedContext" >
...
</Context >
```

Network Considerations

Be sure that multicast is working on each computer that hosts members of the tc Runtime cluster.

If the computers that host your tc Runtime cluster also host other applications that use multicast communications, be sure that the other applications *do not* use the same multicast address and port as the tc Runtime cluster. This precaution eliminates unnecessary processing of irrelevant messages by the tc Runtime cluster. In addition to overhead and decreased performance, unnecessary processing can delay cluster communications, causing a cluster member to be marked failed when in fact it is alive but broadcast of its heartbeat messages is taking too long.

Configuring a Simple tc Runtime Cluster

In this section you set up a simple tc Runtime cluster that uses default values for most configuration options. A description of this default cluster configuration follows the procedure.

1. For each tc Runtime instance that will be a member of the cluster, update its `CATALINA_BASE/conf/server.xml` by adding a `<Cluster>` child-element of the `<Engine>` element, as shown in the following example (only relevant sections shown):

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="-1" shutdown="SHUTDOWN">
  ...
  <Service name="Catalina">
    ...
    <Engine name="Catalina" defaultHost="localhost">
        <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/>
        ...
    </Engine>
  </Service>
</Server>
```

   The `server.xml` file for many tc Runtime instances already contains a commented-out `<Cluster>`; in which case, simply remove the comment tags.

   You can also add the `<Cluster>` element to the `<Host>` element of the `server.xml` file, thus enabling clustering in all virtual hosts of the tc Runtime instance. When you add the `<Cluster>` element inside the `<Engine>` element, the cluster appends the host name of each session manager to the manager's name so that two contexts that have the same name but are part of two different hosts are distinguishable.

2. If you will run more than one tc Runtime instance on the *same* computer, be sure the various TCP/IP listen ports for each tc Runtime instance are unique. You configure the listen ports using the `port` and `redirectPort` attributes of the `<Connector>` element in the `server.xml` file. See Simple tc Runtime Configuration.

3. If you will run more than one tc Runtime instance on the same computer, and you are using Hyperic HQ to monitor and configure the cluster and its individual members, be sure the JMX listen ports for each tc Runtime instance are unique. You configure the JMX listen port using the `port` attribute of the `<Listener classname="com.springsource.tcserver.serviceability.rmi.JmxSocketListener">` element in the `server.xml` file. See Simple tc Runtime Configuration.

4. If the cluster is hosted on more than one computer, time-synchronize the computers with the Network Time Protocol (NTP). See The Network Time Protocol.

The cluster that results from the preceding procedure has the following configuration:

- The cluster is enabled with all-to-all session replication, wherein a session on one member of the cluster that is modified by the client is replicated to *all* other members of the cluster, even members in which the application is not deployed. This is the recommended session replication scheme for small clusters, but as the cluster gains members, SpringSource recommends a primary-secondary replication scheme in which session data is replicated to a single backup member, and only to members in which the application is deployed. See Replicating a Session to a Single Backup Member.
- The multicast address is `228.0.0.4`.
- The multicast port is `45564`.
- The members of the cluster send out heartbeats (to broadcast that they are alive and well) every 500 milliseconds.
- If a heartbeat is not received from a member of the cluster after 3000 milliseconds, the cluster is notified and the member may be marked failed.
- The IP address that members of the cluster broadcast to the other members is the local value of `java.net.InetAddress.getLocalHost().getHostAddress()`.
- The TCP/IP port that members use to listen for replication messages is the first available server socket in range 4000-4100.

For additional detailed information about tc Runtime clusters and a description of the default cluster configuration, see Clustering /Session Replication HOW-TO.

Advanced Cluster Configuration Options

This section describes a small subset of the cluster configuration options that are more advanced than those described in Configuring a Simple tc Runtime Cluster, which describes how to set up a very simple cluster using mostly default values. Read this section if the default cluster values do not suit your needs.

In all cases the configuration requires you to add child elements or attributes to the basic `<Cluster>` element.

This section includes the following subsections:

- Changing the Default Multicast Address and Port
- Changing the Maximum Time After Which an Unresponsive Cluster Member is Dropped
- Replicating a Session to a Single Backup Member

tc Runtime clusters are highly configurable and this section describes only a few use cases. For more information, see Clustering/ Session Replication HOW-TO.

Changing the Default Multicast Address and Port

The default multicast address and port of a cluster are `228.0.0.4` and `45564`, respectively. Sometimes you need to change these values; for example, suppose you want to configure two clusters on the same computer, one for testing and one for production. The easiest way to set this up is to specify different multicast/port combinations for the two clusters.

To change the multicast address and port of a cluster, update the `server.xml` file for each tc Runtime instance that is a member of the cluster and add or update the `<Membership>` child element of the `<Channel>` element, which itself is a child member of the `<Cluster>` element.

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster">
   <Channel className="org.apache.catalina.tribes.group.GroupChannel">
```

```
      <Membership className="org.apache.catalina.tribes.membership.McastService"
                  address="228.1.0.4"
                  port="55564" />

   </Channel>

</Cluster>
```

Use the `address` and `port` attributes of the `<Membership>` element to set the multicast address and port; in the preceding example, the new values are `228.1.0.4` and `55564`, respectively.

For more information on the `<Membership>` element, its default behavior, and the attributes you can set to further configure it, see The Cluster Membership Object.

Changing the Maximum Time After Which an Unresponsive Cluster Member Is Dropped

The default implementation of the cluster group notification is built on multicast heartbeats sent using UDP packets to a multicast IP address. As described in the general cluster documentation, you group cluster members by specifying the same multicast address/port combination (either using the default values or custom values). Each member then sends out a heartbeat within a given interval (frequency); this heartbeat is used for dynamic discovery. The cluster membership listener listens for these heartbeats; if the membership listener does not receive a heartbeat from a node within a certain timeframe (droptime), the cluster considers the member suspect and notifies the channel to take appropriate action.

The default frequency at which members send heartbeats (500 milliseconds) is typically adequate. On high-latency networks, you might want to increase the default value of the drop time (3000 milliseconds) to protect against false positives.

To change the drop time, update the `server.xml` file for each tc Runtime instance that is a member of the cluster and add or update the `<Membership>` child element of the `<Channel>` element, which itself is a child member of the `<Cluster>` element.

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster">
   <Channel className="org.apache.catalina.tribes.group.GroupChannel">
      <Membership className="org.apache.catalina.tribes.membership.McastService"
                  dropTime="6000" />
   </Channel>

</Cluster>
```

Use the `dropTime` attribute of the `<Membership>` element to set the drop time; in the preceding example, the new drop time value is `6000` milliseconds.

For more information on the `<Membership>` element, its default behavior, and the attributes you can set to further configure it, see The Cluster Membership Object.

Replicating a Session to a Single Backup Member

The default cluster configuration uses the DeltaManager object to enable all-to-all session replication, which means that the cluster replicates the session information (typically session deltas) to *all* the other nodes in the cluster, including nodes in which the application is not even deployed. (In this context, a node refers to a tc Runtime instance that is a member of the cluster.) All-to-all replication works well for smaller clusters that are made up of just a few nodes. However, the DeltaManager requires that all nodes in the cluster be homogeneous and that all nodes must deploy the same applications and be exact replicas.

Therefore, if you have configured a large cluster with many nodes, or you find the requirements of the DeltaManager too limiting, SpringSource recommends that you configure the cluster so that it replicates to just a single backup node by using the BackupManager object. The cluster ensures that the node to which it replicates also has the application deployed. The location of the backup node is known to all nodes in the cluster. Finally, because the cluster is replicating to just one node, the cluster supports heterogeneous deployment.

To configure use of a single backup node for session replication, add or update `<Manager>` child element of the `<Cluster>` element in the `server.xml` files for all tc Runtime instances that are members of the cluster, as shown in the following snippet:

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

For additional information about the BackupManager, its default behavior, and the attributes you can set on the `<Manager>` element, see The ClusterManager Object.