The JBoss 4 Application Server Web Developer Reference

JBoss AS 4.0.5

Release 2

Copyright © 2006 JBoss, Inc.

Table of Contents

1. The Tomcat Service 1
2. The server.xml file 3
    2.1. The Connector element 3
    2.2. The Engine element 4
    2.3. The Host element 5
    2.4. The Valve element 5
3. The context.xml file 7
4. Using HTTPS 8
5. Using DIGEST Authentication 11
6. Setting the context root of a web application 13
7. Setting up Virtual Hosts 15
8. Serving Static Content 17
9. Using Apache with Tomcat 18
10. Using JavaServer Faces 19

1

The Tomcat Service

Tomcat 5.5, the latest release of the Apache Java servlet container, supports the Servlet 2.4 and JSP 2.0 specifications. Tomcat is distributed as a deployable service in `jbossweb-tomcat-55.sar` in the deploy directory. It is shipped in exploded directory form, so it's easy to inspect and update the configuration of the embedded Tomcat instance.

The main service file is `META-INF/jboss-service.xml`. It configures the `org.jboss.web.tomcat.tc5.Tomcat5` MBean which controls Tomcat. Its configurable attributes include:

- **DefaultSecurityDomain**: This specifies the JAAS security domain to use in the absence of an explicit `security-domain` specification in the `jboss-web.xml` of a WAR file.
- **Java2ClassLoadingCompliance**: This enables the standard Java2 parent delegation class loading model rather than the servlet model which loads from the WAR first. It is true by default, otherwise loading from WARs that include client JARs with classes used by EJBs causes class loading conflicts. If you enable the servlet class loading model by setting this flag to false, you need to organize yodeployment package to avoid having duplicate classes in the deployment.
- **UseJBossWebLoader**: This flag indicates that Tomcat should use a JBoss unified class loader as the web application class loader. The default is true, which means that the classes inside of the `WEB-INF/classes` and `WEB-INF/lib` directories of the WAR file are incorporated into the default shared class loader repository. This allows classes and resources to be shared between web applications. If this is not what you want, you can disable this behaviour by setting this attribute to false.
- **LenientEjbLink**: This flag indicates that `ejb-link` errors should be ignored in favor of trying the `jndi-name` in the `jboss-web.xml`. The default is true.
- **ManagerClass**: This is the class to use as the session manager for replicating the state of web applications marked as distributable. The only provided implementation session manager is `org.jboss.web.tomcat.tc5.session.JBossCacheManager`, which uses JBossCache to track the distributed state.
- **SubjectAttributeName**: If set, this represents the request attribute name under which the JAAS subject will be stored. There is no default value, meaning that the subject is not set in the request.
- **SessionIdAlphabet**: This is the set of characters used to create a session IDs. It must be made up of exactly 65 unique characters.
- **SnapshotMode**: This sets the snapshot mode in a clustered environment. This must be one of `instant` or `interval`. In instant mode changes to a clustered session are instantly propagated whenever a modification is made. In interval mode all modifications are periodically propagated according to the `SnapshotInterval`.

- **SnapshotInterval**: This sets the snapshot interval in milliseconds for the interval snapshot mode. The default is 1000ms, which is 1 second.
- **UseLocalCache**: This is a flag that indicates whether the local HTTP session value should be used if it exists. When it is true, the existing local HTTP session values are used and updates are replicated, but updates to the same session on other nodes do not update the local session value. This mode is only useful for failover. When it is false, the session value is obtained from the distributed cache. This mode can be used with load balancing. The default is true.
- **UseJK**: This specifies that you are using MOD_JK(2) for load balancing with sticky session combined with `JvmRoute`. If set to true, it will insert a `JvmRouteFilter` to intercept every request and replace the `JvmRoute` if it detects a failover. This additionally requires the `JvmRoute` to be set inside the engine definition in the Tomcat `server.xml` file. The default is false.
- **Domain**: This is the JMX domain under which Tomcat will register additional MBeans. The default domain is `jboss.web`.
- **SecurityMangerService**: This is a reference to the JAAS security manager for Tomcat to use. It defaults to `jboss.security:service=JaasSecurityManager`.

2

The server.xml file

While the `jboss-service.xml` file controls the Tomcat integration service, Tomcat itself has its own configuration file which guides its operation. This is the `server.xml` descriptor that you will find in the `jbossweb-tomcat55.sar` directory.

The `server.xml` file doesn't have a formal DTD or schema definition, so we'll just cover the major configurable elements available. The top-level element is the `Server` element. It should contain a `Service` element representing the entire web subsystem. The supported attributes are:

- **name**: A unique name by which the service is known.
- **className**: The name of the class that provides the service implementation.

2.1. The Connector element

A `Service` element should have one or more connectors under it. A connector configures a transport mechanism that allows clients to send requests and receive responses from the `Service` it is associated with. Connectors forward requests to the engine and return the results to the requesting client. Each connector is configured using `Connector` element. Connectors support these attributes:

- **className**: the fully qualified name of the class of the connector implementation. The class must implement the `org.apache.catalina.Connector` interface. The embedded service defaults to the `org.apache.catalina.connector.http.HttpConnector`, which is the HTTP connector implementation.
- **acceptCount**: This is the maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused. The default value is 10.
- **address**. For servers with more than one IP address, this attribute specifies which address will be used for listening on the specified port. By default, this port will be used on all IP addresses associated with the server.
- **bufferSize**: This is the size (in bytes) of the buffer to be provided for input streams created by this connector. By default, buffers of 2048 bytes will be provided.
- **connectionTimeout**: This is the number of milliseconds this connector will wait, after accepting a connection, for the request URI line to be presented. The default value is 60000 (i.e. 60 seconds).
- **debug**: This is the debugging detail level of log messages generated by this component, with higher numbers creating more detailed output. If not specified, this attribute is set to zero (0). Whether or not this shows up in the log further depends on the log4j category `org.jboss.web.tomcat.tc5.Tomcat5` threshold.

- **enableLookups**: This is a flag that enables DNS resolution of the client hostname, as accessed via the `ServletRequest.getRemoteHost` method. This flag defaults to false.
- **maxThreads**: This is the maximum number of request processing threads to be created by this connector, which therefore determines the maximum number of simultaneous requests that can be handled. If not specified, this attribute is set to 200.
- **maxSpareThreads**: This is the maximum number of unused request processing threads that will be allowed to exist until the thread pool starts stopping the unnecessary threads. The default value is 50.
- **minSpareThreads**: This is the number of request processing threads that will be created when this connector is first started. The connector will also make sure it has the specified number of idle processing threads available. This attribute should be set to a value smaller than that set for maxThreads. The default value is 4.
- **port**: This is the TCP port number on which this connector will create a server socket and await incoming connections. Only one server application can listen to a particular port number on a particular IP address at a time.
- **proxyName**: If this connector is being used in a proxy configuration, this attribute specifies the server name to be returned for calls to `request.getServerName()`.
- **proxyPort**: If this connector is being used in a proxy configuration, this attribute specifies the server port to be returned for calls to `request.getServerPort()`.
- **redirectPort**: This is the port that non-SSL requests will be redirected to when a request for content secured under a transport confidentiality or integrity constraint is received. This defaults to the standard HTTPS port of 443.
- **secure**: This sets the `ServletRequest.isSecure` method value flag to indicate whether or not the transport channel is secure. This flag defaults to false.
- **scheme**: This sets the protocol name as accessed by the `ServletRequest.getScheme` method. The scheme defaults to `http`.
- **tcpNoDelay**: If this attribute is set to true, the `TCP_NO_DELAY` option will be set on the server socket, which improves performance under most circumstances. This is set to true by default.

You can find attribute descriptions in the Tomcat documentation at http://jakarta.apache.org/tomcat/tomcat-5.5-doc/config/http.html.

2.2. The Engine element

Each `Service` must have a single `Engine` configuration. An engine handles the requests submitted to a service via the configured connectors. The child elements supported by the embedded service include `Host`, `Logger`, `Valve` and `Listener`. The supported attributes include:

- **className**: This is the fully qualified class name of the `org.apache.catalina.Engine` interface implementation to use. If not specifies this defaults to `org.apache.catalina.core.StandardEngine`.
- **defaultHost**: This is he name of a `Host` configured under the `Engine` that will handle requests with host names

that do not match a `Host` configuration.

- **name**: This is a logical name assigned to the `Engine`. It is used in log messages produced by the `Engine`.

You can find additional information on the `Engine` element in the Tomcat documentation at http://jakarta.apache.org/tomcat/tomcat-5.5-doc/config/engine.html.

2.3. The Host element

A `Host` element represents a virtual host configuration. It is a container for web applications with a specified DNS hostname. The child elements supported by the embedded service include `Alias`, `Valve` and `Listener`. The supported attributes include:

- **className**: This is the fully qualified class name of the `org.apache.catalina.Host` interface implementation to use. If not specifies this defaults to `org.apache.catalina.core.StandardHost`.
- **name**: This is the DNS name of the virtual host. At least one `Host` element must be configured with a name that corresponds to the `defaultHost` value of the containing `Engine`.

The `Alias` element is an optional child element of the `Host` element. Each `Alias` specifies an alternate DNS name for the enclosing `Host`.

You can find additional information on the `Host` element in the Tomcat documentation at http://jakarta.apache.org/tomcat/tomcat-5.5-doc/config/host.html.

2.4. The Valve element

A `Valve` element configures a hook into the request processing pipeline for the web container. Valves must implement the `org.apache.catalina.Valve` interface. There is only one required configuration attribute:

- **className**: This is the fully qualified class name of the `org.apache.catalina.Valve` interface implementation.

The most commonly used valve is the `AccessLogValve`, which keeps a standard HTTP access log of incoming requests. The `className` for the access log value is `org.jboss.web.catalina.valves.AccessLogValue`. The additional attributes its supports include:

- **directory**: This is the directory path into which the access log files will be created.
- **pattern**: This is a pattern specifier that defines the format of the log messages. It defaults to `common`.
- **prefix**: This is the prefix to add to each log file name. It defaults to `access_log`.
- **suffix**: This is the suffix to add to each log file name. It defaults to an empty string, meaning that no suffix will be added.

You can find additional information on the `Valve` element and the available valve implementations in the Tomcat

documentation at http://jakarta.apache.org/tomcat/tomcat-5.5-doc/config/valve.html.

3

The context.xml file

The `context.xml` file contains the default Context element used for all web applications in the system. The supported attributes include:

- **cookies**: This is a flag indicating if sessions will be tracked using cookies. The default is true.
- **crossContext**: This is a flag indicating whether the `ServletContext.getContext(String path)` method should return contexts for other web applications deployed in the calling web application's virtual host.

You can find additional information on the `Context` element in the Tomcat documentation at http://jakarta.apache.org/tomcat/tomcat-5.5-doc/config/context.html.

4

Using HTTPS

There are a few ways you can configure HTTP over SSL for the embedded Tomcat servlet container depending on whether or not you use the JBoss specific connector socket factory, which allows you to obtain the JSSE server certificate information from a JBossSX `SecurityDomain`. This requires establishing a `SecurityDomain` using the `org.jboss.security.plugins.JaasSecurityDomain` MBean. A `server.xml` configuration file that illustrates the setup of only an SSL connector via this approach is given below.

```
<Server>
    <Service name="jboss.web" className="org.jboss.web.tomcat.tc5.StandardService">

        <Connector port="8080" address="${jboss.bind.address}" maxThreads="150"
            minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
            redirectPort="443" acceptCount="100" connectionTimeout="20000"
            disableUploadTimeout="true"/>

        <Connector port="443" address="${jboss.bind.address}" maxThreads="100"
            minSpareThreads="5" maxSpareThreads="15" scheme="https"
            secure="true" clientAuth="false"
            keystoreFile="${jboss.server.home.dir}/conf/chap8.keystore"
            keystorePass="rmi+ssl" sslProtocol="TLS"/>

        <Engine name="jboss.web" defaultHost="localhost">
            <Realm
                className="org.jboss.web.tomcat.security.JBossSecurityMgrRealm"
                certificatePrincipal="org.jboss.securia.Log4jLogger"
                verbosityLevel="WARNING" category="org.jboss.web.localhost.Engine"/>
            <Host name="localhost" autoDeploy="false" deployOnStartup="false"
                  deployXML="false">
                <DefaultContext cookies="true" crossContext="true" override="true"/>
            </Host>
        </Engine>
    </Service>
            </Server>
```

This configuration includes a `JaasSecurityDomain`, but since the descriptor is not being deployed as part of a SAR that includes the `chap8.keystore`, you need to copy the `chap8.keystore` to the `server/default/conf` directory. You can test this configuration by accessing the JMX console web application over HTTPS using this URL: https://localhost/jmx-console.

Note: if you are running on a system that requires special permissions to open ports below 1024, it might be easier need to change the port number to one above 1024. Port 8443 is commonly used because of this.

The configurable attributes are as follows:

- **algorithm**: This is the certificate encoding algorithm to be used. If not specified, the default value is SunX509.
- **className**: This is the fully qualified class name of the SSL server socket factory implementation class. You must specify org.apache.coyote.tomcat4.CoyoteServerSocketFactory here. Using any other socket factory will

not cause an error, but the server socket will not be using SSL.

- **clientAuth**: This attribute should be set to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. A false value, which is the default, will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.
- **keystoreFile**: This is the pathname of the keystore file where you have stored the server certificate to be loaded. By default, the pathname is the file `.keystore` in the operating system home directory of the user that is running Tomcat.
- **keystorePass**: This is the password used to access the server certificate from the specified keystore file. The default value is `changeit`.
- **keystoreType**: The type of keystore file to be used for the server certificate. If not specified, the default value is `JKS`.
- **protocol**: The version of the SSL protocol to use. If not specified, the default is `TLS`.

Note that if you try to test this configuration using the self-signed certificate and attempt to access content over an HTTPS connection, your browser should display a warning dialog indicating that it does not trust the certificate authority that signed the certificate of the server you are connecting to. For example, when we tested the first configuration example, IE 5.5 showed the initial security alert dialog listed in Figure 4.1. Figure 4.2 shows the server certificate details. This warning is important because anyone can generate a self-signed certificate with any information desired. Your only way to verify that the system on the other side really represents the party it claim to is by verifying that it is signed by a trusted third party.

Figure 4.1. The Internet Explorer 5.5 security alert dialog.

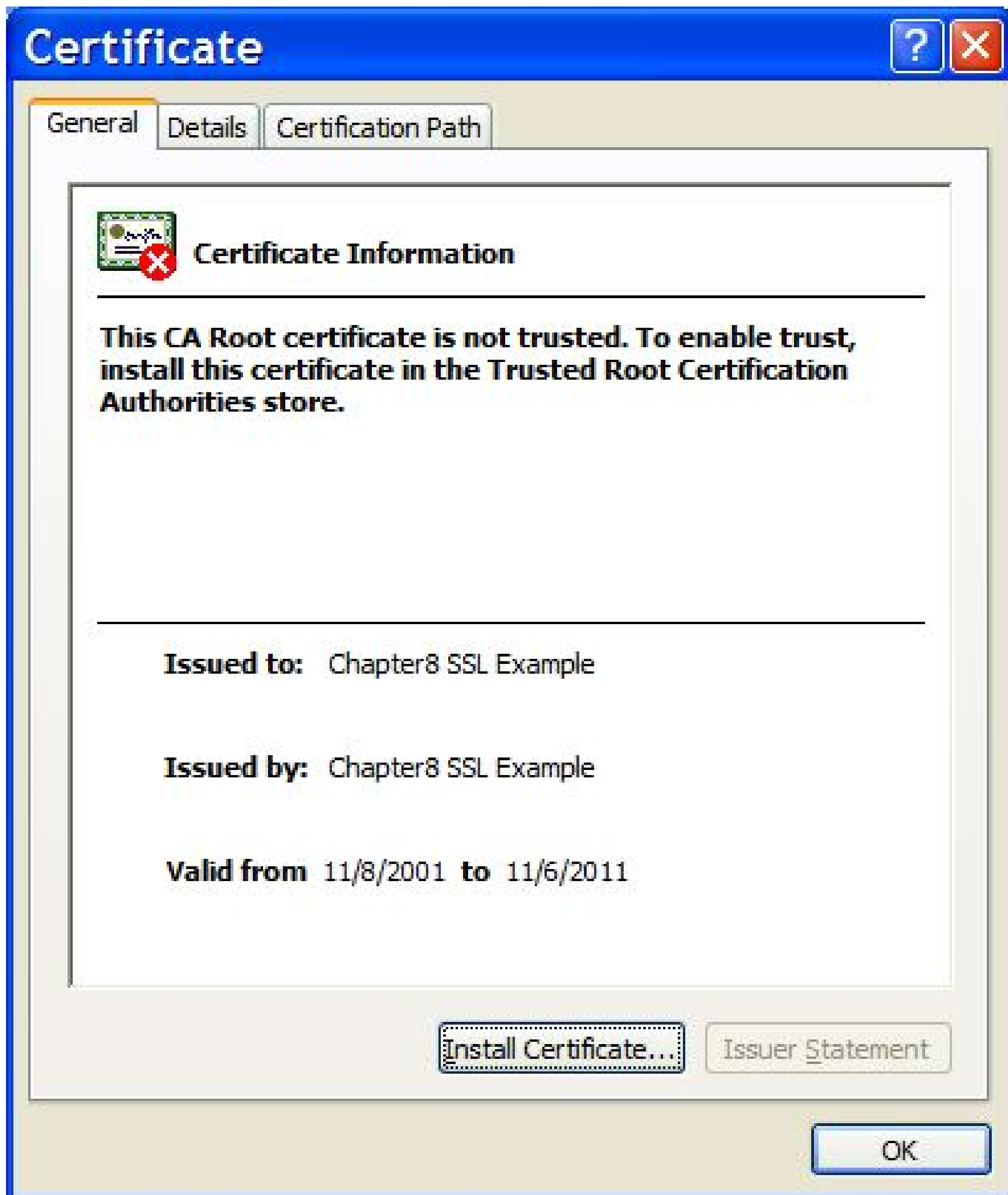

Figure 4.2. The Internet Explorer 5.5 SSL certificate details dialog.

5

Using DIGEST Authentication

When using BASIC and FORM web authentications, the users password is sent in the clear as part of the HTTP requests. As we saw in the last section, it is possible to encrypt the entire session using HTTPS, keeping the password private over the wire. However, this still requires the password to exist on in plain text form on the server, at least temporarily in memory if not in password store.

Digest authentication employs a challenge-response mechanism, whereby the server sends a unique challenge to the client. The client responds with a hashed value that the server compares against it's own hashed value. At no point does the client ever send the the actual password text to the server.

Web applications request digest authentication by setting the `auth-method` to `DIGEST` in the `web.xml` deployment descriptor. The following example shows what this would look like, omitting the application-specific `security-constraint` and `security-role` declarations.

```
<login-config>
    <auth-method>DIGEST</auth-method>
    <realm-name>My Aplication</realm-name>
</login-config>
```

To complete the configuration, we'll create a special digest-friendly security domain and link it to the application. For this example, we'll create a security domain under the name `java:/jaas/digest`. The application would link to it in the `jboss-web.xml` file.

```
<jboss-web>
    <security-domain>java:/jaas/digest</security-domain>
</jboss-web>
```

Now we need to create the security domain definition. We'll use the `UsersRolesLoginModule` in this example, though any login module that supports password hashing can be used. The following examples shows a complete configuration.

```
<application-policy name="digest">
    <authentication>
        <login-module code="org.jboss.security.auth.spi.UsersRolesLoginModule"
                      flag="required">
            <module-option name="usersProperties">digest-users.properties</module-option>
            <module-option name="rolesProperties">digest-roles.properties</module-option>
            <module-option name="hashAlgorithm">MD5</module-option>
            <module-option name="hashEncoding">rfc2617</module-option>
            <module-option name="hashUserPassword">false</module-option>
            <module-option name="hashStorePassword">true</module-option>
            <module-option name="passwordIsA1Hash">true</module-option>
            <module-option name="storeDigestCallback">
                org.jboss.security.auth.spi.RFC2617Digest
            </module-option>
        </login-module>
    </authentication>
</application-policy>
```

The first two module options configure the locations of the user and roles properties file. The remaining six complete the configuration for digest authentication. To enable digest authentication in your application, copy these last 6 options into your login module configuration.

At this point, all the is required is to create the password hashes to be stored in your user store, which is the `digest-users.properties` file in this example. Digest hashes hash the username, the password, and the realm name together. The realm name comes from the realm name in `web.xml` file. In this example it is `My Application`.

JBoss provides a helper class to create digest hashes. It can be invoked from the bin directory as shown here:

```
[bin]$ java -cp ../server/default/lib/jbosssx.jar \
org.jboss.security.auth.spi.RFC2617Digest username "My Application" password
RFC2617 A1 hash: 9b47ec6f03603dd49863e7d58c4c49ea
```

The three arguments are the username, the realm name and the password. The digested password should be stored in the user management store. In the example here, it would go in the `digest-users.properties` file.

```
user=9b47ec6f03603dd49863e7d58c4c49ea
```

You would still need to define the application roles and configure them in the login module to complete the security configuration. However, none of this differs with digest authentication.

6

Setting the context root of a web application

The context root of a web application determines which URLs Tomcat will delegate to your web application. If your application's context root is `myapp` then any request for `/myapp` or `/myapp/*` will be handled by your application unless a more specific context root exists. If a second web application were assigned the context root `myapp/help`, a request for `/myapp/help/help.jsp` would be handled by the second web application, not the first.

This relationship also holds when the context root is set to `/`, which is known as the root context. When an application is assigned to the root context, it will respond to all requests no handled by a more specific context root.

The context root for an application is determined by how it is deployed. When a web application is deployed inside an EAR file, the context root is specified in the `application.xml` file of the EAR, using a `context-root` element inside of a web module. In the following example, the context root of the `web-client.war` application is set to `bank`.

```
<application xmlns="http://java.sun.com/xml/ns/j2ee" version="1.4"
    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/application_1_4.xsd">
    <display-name>JBossDukesBank</display-name>

    <module>
        <ejb>bank-ejb.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>web-client.war</web-uri>
            <context-root>bank</context-root>
        </web>
    </module>

</application>
```

For web applications that are deployed outside an EAR file, the context root can be specified in two ways. First, the context root can be specified in the `WEB-INF/jboss-web.xml` file. The following example shows what the `jboss-web.xml` file would look like if it weren't bundled inside of an EAR file.

```
<jboss-web>
    <context-root>bank</context-root>
</jboss-web>
```

Finally, if no context root specification exists, the context root will be the base name of the WAR file. For `web-client.war`, the context root would default to `web-client`. The only special case to this naming special name `ROOT`. To deploy an application under the root context, you simply name it `ROOT.war`. JBoss already contains a `ROOT.war` web application in the `jbossweb-tomcat55.sar` directory. You will need to remove or rename that one to create your own root application.

Naming your WAR file after the context root they are intended to handle is a very good practice. Not only does it

reduce the number of configuration settings to manage, but it improves the maintainability of the application by making the intended function of the web application clear.

7

Setting up Virtual Hosts

Virtual hosts allow you to group web applications according to the various DNS names by which the machine running JBoss is known. As an example, consider the `server.xml` configuration file given in Example 7.1. This configuration defines a default host named `vhost1.mydot.com` and a second host named `vhost2.mydot.com`, which also has the alias `www.mydot.com` associated with it.

Example 7.1. A virtual host configuration.

```
<Server>
   <Service name="jboss.web"
      className="org.jboss.web.tomcat.tc5.StandardService">

      <!-- A HTTP/1.1 Connector on port 8080 -->
      <Connector port="8080" address="${jboss.bind.address}"
                 maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
                 enableLookups="false" redirectPort="8443" acceptCount="100"
                 connectionTimeout="20000" disableUploadTimeout="true"/>

      <Engine name="jboss.web" defaultHost="vhost1">
         <Realm className="org.jboss.web.tomcat.security.JBossSecurityMgrRealm"
                certificatePrincipal="org.jboss.security.auth.certs.SubjectDNMapping"
            />
         <Logger className="org.jboss.web.tomcat.Log4jLogger"
                 verbosityLevel="WARNING"
                 category="org.jboss.web.localhost.Engine"/>

            <Host name="vhost1" autoDeploy="false"
                  deployOnStartup="false" deployXML="false">
                <Alias>vhost1.mydot.com</Alias>
                <Valve className="org.apache.catalina.valves.AccessLogValve"
                       prefix="vhost1" suffix=".log" pattern="common"
                       directory="${jboss.server.home.dir}/log"/>


                <DefaultContext cookies="true" crossContext="true" override="true"/>
            </Host>
            <Host name="vhost2" autoDeploy="false"
                  deployOnStartup="false" deployXML="false">
                <Alias>vhost2.mydot.com</Alias>
                <Alias>www.mydot.com</Alias>

                <Valve className="org.apache.catalina.valves.AccessLogValve"
                       prefix="vhost2" suffix=".log" pattern="common"
                       directory="${jboss.server.home.dir}/log"/>

                <DefaultContext cookies="true" crossContext="true" override="true"/>
            </Host>
      </Engine>
   </Service>
</Server>
```

When a WAR file is deployed, it is associated by default with the virtual host whose name matches the `defaultHost` attribute of the containing `Engine`. To deploy a WAR to a specific virtual host you need to specify an appropriate `virtual-host` definition in your `jboss-web.xml` descriptor. The following `jboss-web.xml` descriptor demonstrates how to deploy a WAR to the virtual host `www.mydot.com`. Note that you can use either the virtual host name in the config file or the actual host name.

```
<jboss-web>
    <context-root>/</context-root>
    <virtual-host>www.mydot.com</virtual-host>
</jboss-web>
```

8

Serving Static Content

JBoss provides a default application that serves content for the *root* application context. This default context is the `ROOT.war` application in the `jbossweb-tomcat55.sar` directory. You can serve static files not associated with any other application by adding that content to the `ROOT.war` directory. For example, if you want to have a shared image directory you could create an `image` subdirectory in `ROOT.war` and place the images there. You could then access an image named `myimage.jpg` at http://localhost:8080/images/myimage.jpg.

9

Using Apache with Tomcat

In some architectures, it is useful to put an Apache web server in front of the JBoss server. External web clients talk to an Apache instance, which in turn speaks to the Tomcat instance on behalf of the clients. Apache needs to be configured to use the mod_jk module which speaks the AJP protocol to an AJP connector running in Tomcat. The provided `server.xml` file comes with this AJP connector enabled.

```
<Connector port="8009" address="${jboss.bind.address}"
           enableLookups="false" redirectPort="8443" debug="0"
           protocol="AJP/1.3" />
```

You'll need to consult the Apache and mod_jk documentation for complete installation instructions. Assuming you have a properly configured Apache instance, the following configuration fragment shows an example of how to connect with a WAR deployed with a context root of `/jbosstest`.

```
...
LoadModule jk_module libexec/mod_jk.so
AddModule mod_jk.c

<IfModule mod_jk.c>
    JkWorkersFile /tmp/workers.properties
    JkLogFile /tmp/mod_jk.log
    JkLogLevel debug
    JkMount /jbosstest/* ajp13
</IfModule>
```

The `workers.properties` file contains the details of how to contact the JBoss instance. For more details on how to front Apache for JBoss, especially for a cluster of JBoss servers, please refer to JBoss 4 Clustering Guide.

10

Using JavaServer Faces

Starting with version 4.0.3, the JBoss Application Server features built-in JavaServer Faces support. The implementation used is Apache MyFaces; however, the MyFaces extensions are not included with the JBoss distribution at this time. JBoss allows you to install your JSF application without putting lots of extra JSF implementation JAR files in your `WEB-INF/lib` directory.

To make use of JSF, you need to declare the Faces Servlet and the servlet mapping in the `web.xml` file:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

In order to initialize the JSF environment, you also need to add the MyFaces listener to `web.xml` file:

```
<listener>
    <listener-class>org.apache.myfaces.webapp.StartupServletContextListener</listener-class>
</listener>
```

There are several additional context parameters that can be set to control the configuration of JavaServer Faces for a particular application:

```
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
        /WEB-INF/examples-config.xml
    </param-value>
    <description>
        Comma separated list of URIs of (additional) faces config files.
        (e.g. /WEB-INF/my-config.xml)
        See JSF 1.0 PRD2, 10.3.2
    </description>
</context-param>

<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
    <description>
        State saving method: "client" or "server" (= default)
        See JSF Specification 2.5.2
    </description>
</context-param>

<context-param>
    <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-name>
```

```
    <param-value>true</param-value>
    <description>
        This parameter tells MyFaces if javascript code should be allowed in the
        rendered HTML output.
        If javascript is allowed, command_link anchors will have javascript code
        that submits the corresponding form.
        If javascript is not allowed, the state saving info and nested parameters
        will be added as url parameters.
        Default: "true"
    </description>
</context-param>

<context-param>
    <param-name>org.apache.myfaces.DETECT_JAVASCRIPT</param-name>
    <param-value>false</param-value>
    <description>
        This parameter tells MyFaces if javascript code should be allowed in the
        rendered HTML output.
        If javascript is allowed, command_link anchors will have javascript code
        that submits the corresponding form.
        If javascript is not allowed, the state saving info and nested parameters
        will be added as url parameters.
        Default: "false"

        Setting this param to true should be combined with STATE_SAVING_METHOD "server" for
        best results.

        This is an EXPERIMENTAL feature. You also have to enable the detector
        filter/filter mapping below to get JavaScript detection working.
    </description>
</context-param>

<context-param>
    <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
    <param-value>true</param-value>
    <description>
        If true, rendered HTML code will be formatted, so that it is "human readable".
        i.e. additional line separators and whitespace will be written, that do not
        influence the HTML code.
        Default: "true"
    </description>
</context-param>

<context-param>
    <param-name>org.apache.myfaces.AUTO_SCROLL</param-name>
    <param-value>true</param-value>
    <description>
        If true, a javascript function will be rendered that is able to restore the
        former vertical scroll on every request. Convenient feature if you have pages
        with long lists and you do not want the browser page to always jump to the top
        if you trigger a link or button action that stays on the same page.
        Default: "false"
    </description>
</context-param>
```

To use another JSF implementation, such as the reference implementation, instead of the bundled MyFaces implementation, simply delete the

```
jbossweb-tomcat55.sar/jsf-lib
```

directory. You will need to include the the implementation JAR files in your own

```
WEB-INF/lib
```

directory for each web application making use of JSF.