Apache Tomcat 6.0

SSL Configuration HOW-TO

Quick Start

IMPORTANT NOTE: This Howto refers to usage of JSSE, that comes included with jdk 1.5 and higher. When using APR, Tomcat will use OpenSSL, which uses a different configuration.

The description below uses the variable name $CATALINA_BASE to refer the base directory against which most relative paths are resolved. If you have not configured Tomcat 6 for multiple instances by setting a CATALINA_BASE directory, then $CATALINA_BASE will be set to the value of $CATALINA_HOME, the directory into which you have installed Tomcat 6.

To install and configure SSL support on Tomcat 6, you need to follow these simple steps. For more information, read the rest of this HOW-TO.

1. Create a certificate keystore by executing the following command:

   Windows:

   ```
   %JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
   ```

   Unix:

   ```
   $JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
   ```

   and specify a password value of "changeit".

2. Uncomment the "SSL HTTP/1.1 Connector" entry in `$CATALINA_BASE/conf/server.xml` and tweak as necessary.

Introduction to SSL

SSL, or Secure Socket Layer, is a technology which allows web browsers and web servers to communicate over a secured connection. This means that the data being sent is encrypted by one side, transmitted, then decrypted by the other side before processing. This is a two-way process, meaning that both the server AND the browser encrypt all traffic before sending out data.

Another important aspect of the SSL protocol is Authentication. This means that during your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials, in the form of a "Certificate", as proof the site is who and what it claims to be. In certain cases, the server may also request a Certificate from your web browser, asking for proof that *you* are who you claim to be. This is known as "Client Authentication," although in practice this is used more for business-to-business (B2B) transactions than with individual users. Most SSL-enabled web servers do not request Client Authentication.

SSL and Tomcat

It is important to note that configuring Tomcat to take advantage of secure sockets is usually only necessary when running it as a stand-alone web server. When running Tomcat primarily as a Servlet/JSP container behind another web server, such as Apache or Microsoft IIS, it is usually necessary to configure the primary web server to handle the SSL connections from users. Typically, this server will negotiate all SSL-related functionality, then pass on any requests destined for the Tomcat container only after decrypting those requests. Likewise, Tomcat will return cleartext responses, that will be encrypted before being returned to the user's browser. In this environment, Tomcat knows that communications between the primary web server and the client are taking place over a secure connection (because your application needs to be able to ask about this), but it does not participate in the encryption or decryption itself.

Certificates

In order to implement SSL, a web server must have an associated Certificate for each external interface (IP address) that accepts secure connections. The theory behind this design is that a server should provide some kind of reasonable assurance that its owner is who you think it is, particularly before receiving any sensitive information. While a broader explanation of Certificates is beyond the scope of this document, think of a Certificate as a "digital driver's license" for an Internet address. It states what company the site is associated with, along with some basic contact information about the site owner or administrator.

This "driver's license" is cryptographically signed by its owner, and is therefore extremely difficult for anyone else to forge. For sites involved in e-commerce, or any other business transaction in which authentication of identity is important, a Certificate is typically purchased from a well-known *Certificate Authority* (CA) such as VeriSign or Thawte. Such certificates can be electronically verified -- in effect, the Certificate Authority will vouch for the authenticity of the certificates that it grants, so you can believe that that Certificate is valid if you trust the Certificate Authority that granted it.

In many cases, however, authentication is not really a concern. An administrator may simply want to ensure that the data being transmitted and received by the server is private and cannot be snooped by anyone who may be eavesdropping on the connection. Fortunately, Java provides a relatively simple command-line tool, called `keytool`, which can easily create a "self-signed" Certificate. Self-signed Certificates are simply user generated Certificates which have not been officially registered with any well-known CA, and are therefore not really guaranteed to be authentic at all. Again, this may or may not even be important, depending on your needs.

General Tips on Running SSL

The first time a user attempts to access a secured page on your site, he or she is typically presented with a dialog containing the details of the certificate (such as the company and contact name), and asked if he or she wishes to accept the Certificate as valid and continue with the transaction. Some browsers will provide an option for permanently accepting a given Certificate as valid, in which case the user will not be bothered with a prompt each time they visit your site. Other browsers do not provide this option. Once approved by the user, a Certificate will be considered valid for at least the entire browser session.

Also, while the SSL protocol was designed to be as efficient as securely possible, encryption/decryption is a computationally expensive process from a performance standpoint. It is not strictly necessary to run an entire web application over SSL, and indeed a developer can pick and choose which pages require a secure connection and which do not. For a reasonably busy site, it is customary to only run certain pages under SSL, namely those pages where sensitive information could possibly be exchanged. This would include things like login pages, personal information pages, and shopping cart checkouts, where credit card information could possibly be transmitted. Any page within an application can be requested over a secure socket by simply prefixing the address with `https:` instead of `http:`. Any pages which absolutely **require** a secure connection should check the protocol type associated with the page request and take the appropriate action if `https` is not specified.

Finally, using name-based virtual hosts on a secured connection can be problematic. This is a design limitation of the SSL protocol itself. The SSL handshake, where the client browser accepts the server certificate, must occur before the HTTP request is accessed. As a result, the request information containing the virtual host name cannot be determined prior to authentication, and it is therefore not possible to assign multiple certificates to a single IP address. If all virtual hosts on a single IP address need to authenticate against the same certificate, the addition of multiple virtual hosts should not interfere with normal SSL operations on the server. Be aware, however, that most client browsers will compare the server's domain name against the domain name listed in the certificate, if any (applicable primarily to official, CA-signed certificates). If the domain names do not match, these browsers will display a warning to the client user. In general, only address-based virtual hosts are commonly used with SSL in a production environment.

Configuration

Prepare the Certificate Keystore

Tomcat currently operates only on `JKS`, `PKCS11` or `PKCS12` format keystores. The `JKS` format is Java's standard "Java KeyStore" format, and is the format created by the `keytool` command-line utility. This tool is included in the JDK. The `PKCS12` format is an internet standard, and can be manipulated via (among other things) OpenSSL and Microsoft's Key-Manager.

Each entry in a keystore is identified by an alias string. Whilst many keystore implementations treat aliases in a case insensitive manner, case sensitive implementations are available. The

`PKCS11` specification, for example, requires that aliases are case sensitive. To avoid issues related to the case sensitivity of aliases, it is not recommended to use aliases that differ only in case.

To import an existing certificate into a JKS keystore, please read the documentation (in your JDK documentation package) about `keytool`. Note that OpenSSL often adds readable comments before the key, `keytool`does not support that, so remove the OpenSSL comments if they exist before importing the key using `keytool`.

To import an existing certificate signed by your own CA into a PKCS12 keystore using OpenSSL you would execute a command like:

```
openssl pkcs12 -export -in mycert.crt -inkey mykey.key \
                        -out mycert.p12 -name tomcat -CAfile myCA.crt \
                        -caname root -chain
```

For more advanced cases, consult the OpenSSL documentation.

To create a new keystore from scratch, containing a single self-signed Certificate, execute the following from a terminal command line:

Windows:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA
```

Unix:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

(The RSA algorithm should be preferred as a secure algorithm, and this also ensures general compatibility with other servers and components.)

This command will create a new file, in the home directory of the user under which you run it, named "`.keystore`". To specify a different location or filename, add the `-keystore` parameter, followed by the complete pathname to your keystore file, to the `keytool` command shown above. You will also need to reflect this new location in the `server.xml` configuration file, as described later. For example:

Windows:

```
%JAVA_HOME%\bin\keytool -genkey -alias tomcat -keyalg RSA \
  -keystore \path\to\my\keystore
```

Unix:

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA \
  -keystore /path/to/my/keystore
```

After executing this command, you will first be prompted for the keystore password. The default password used by Tomcat is "`changeit`" (all lower case), although you can specify a custom password if you like. You will also need to specify the custom password in the `server.xml` configuration file, as described later.

Next, you will be prompted for general information about this Certificate, such as company, contact name, and so on. This information will be displayed to users who attempt to access a secure page in your application, so make sure that the information provided here matches what they will expect.

Finally, you will be prompted for the *key password*, which is the password specifically for this Certificate (as opposed to any other Certificates stored in the same keystore file). You **MUST** use the same password here as was used for the keystore password itself. (Currently, the `keytool` prompt will tell you that pressing the ENTER key does this for you automatically.)

If everything was successful, you now have a keystore file with a Certificate that can be used by your server.

Note: your private key password and keystore password should be the same. If they differ, you will get an error along the lines of `java.io.IOException: Cannot recover key`, as documented in Bugzilla issue 38217, which contains further references for this issue.

Edit the Tomcat Configuration File

If you are using APR, you have the option of configuring an alternative engine to OpenSSL.

```
<Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="someengine" SSLRandomSeed="somedevice" />
```

The default value is

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

So to use SSL under APR, make sure the SSLEngine attribute is set to something other than `off`. The default value is `on` and if you specify another value, it has to be a valid engine name. If you haven't compiled in SSL support into your Tomcat Native library, then you can turn this initialization off

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

SSLRandomSeed allows to specify a source of entropy. Productive system needs a reliable source of entropy but entropy may need a lot of time to be collected therefore test systems could use no blocking entropy sources like "/dev/urandom" that will allow quicker starts of Tomcat.

The final step is to configure your secure socket in the `$CATALINA_BASE/conf/server.xml` file, where `$CATALINA_BASE` represents the base directory for the Tomcat 6 instance. An example `<Connector>` element for an SSL connector is included in the default `server.xml` file installed with Tomcat. It will look something like this:

```
<-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector
           port="8443" minSpareThreads="5" maxSpareThreads="75"
           enableLookups="true" disableUploadTimeout="true"
           acceptCount="100"  maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"/>
-->
```

The example above will throw an error if you have the APR and the Tomcat Native libraries in your path, as tomcat will try to autoload the APR connector. The APR connector uses different attributes for SSL keys and certificates. An example of such configuration would be

```
<-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector
           port="8443" minSpareThreads="5" maxSpareThreads="75"
           enableLookups="true" disableUploadTimeout="true"
           acceptCount="100"  maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           SSLCertificateFile="/usr/local/ssl/server.crt"
           SSLCertificateKeyFile="/usr/local/ssl/server.pem"
           clientAuth="false" sslProtocol="TLS"/>
-->
```

To avoid auto configuration you can define which connector to use by specifying a classname in the protocol attribute.
To define a Java connector, regardless if the APR library is loaded or not do:

```
<-- Define a blocking Java SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector protocol="org.apache.coyote.http11.Http11Protocol"
           port="8443" minSpareThreads="5" maxSpareThreads="75"
           enableLookups="true" disableUploadTimeout="true"
           acceptCount="100"  maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"/>
-->
<-- Define a non-blocking Java SSL Coyote HTTP/1.1 Connector on port 8443 -
->
```

```
<!--
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
           port="8443" minSpareThreads="5" maxSpareThreads="75"
           enableLookups="true" disableUploadTimeout="true"
           acceptCount="100"  maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           keystoreFile="${user.home}/.keystore" keystorePass="changeit"
           clientAuth="false" sslProtocol="TLS"/>
-->
```

and to specify an APR connector

```
<-- Define a APR SSL Coyote HTTP/1.1 Connector on port 8443 -->
<!--
<Connector protocol="org.apache.coyote.http11.Http11AprProtocol"
           port="8443" minSpareThreads="5" maxSpareThreads="75"
           enableLookups="true" disableUploadTimeout="true"
           acceptCount="100"  maxThreads="200"
           scheme="https" secure="true" SSLEnabled="true"
           SSLCertificateFile="/usr/local/ssl/server.crt"
           SSLCertificateKeyFile="/usr/local/ssl/server.pem"
           clientAuth="false" sslProtocol="TLS"/>
-->
```

You will note that the Connector element itself is commented out by default, so you will need to remove the comment tags around it. Then, you can customize the specified attributes as necessary. For detailed information about the various options, consult the Server Configuration Reference. The following discussion covers only those attributes of most interest when setting up SSL communication.

The `port` attribute (default value is 8443) is the TCP/IP port number on which Tomcat will listen for secure connections. You can change this to any port number you wish (such as to the default port for `https` communications, which is 443). However, special setup (outside the scope of this document) is necessary to run Tomcat on port numbers lower than 1024 on many operating systems.

If you change the port number here, you should also change the value specified for the `redirectPort` attribute on the non-SSL connector. This allows Tomcat to automatically redirect users who attempt to access a page with a security constraint specifying that SSL is required, as required by the Servlet 2.4 Specification.

There are additional options used to configure the SSL protocol. You may need to add or change the following attribute values, depending on how you configured your keystore earlier:

Attribute	Description
`clientAuth`	Set this value to `true` if you want Tomcat to require all SSL clients to present a client Certificate in order to use this socket. Set this value to `want` if you want Tomcat to request a client Certificate, but not fail if one isn't presented.

`SSLEnabled`	Use this attribute to enable SSL traffic on a connector. To turn on SSL handshake/encryption/decryption on a connector set this value to `true`. The default value is `false`. When turning this value `true` you will want to set the `scheme` and the `secure` attributes as well to pass the correct `request.getScheme()` and `request.isSecure()` values to the servlets
`keystoreFile`	Add this attribute if the keystore file you created is not in the default place that Tomcat expects (a file named `.keystore` in the user home directory under which Tomcat is running). You can specify an absolute pathname, or a relative pathname that is resolved against the `$CATALINA_BASE` environment variable.
`keystorePass`	Add this element if you used a different keystore (and Certificate) password than the one Tomcat expects (`changeit`).
`keystoreType`	Add this element if using a keystore type other than `JKS`. For example the *.p12 files from OpenSSL can be used using `PKCS12`.
`sslProtocol`	The encryption/decryption protocol to be used on this socket. It is not recommended to change this value if you are using Sun's JVM. It is reported that IBM's 1.4.1 implementation of the TLS protocol is not compatible with some popular browsers. In this case, use the value `SSL`.
`ciphers`	The comma separated list of encryption ciphers that this socket is allowed to use. By default, any available cipher is allowed.
`algorithm`	The `X509` algorithm to use. This defaults to the Sun implementation (`SunX509`). For IBM JVMs you should use the value `IbmX509`. For other vendors, consult the JVM documentation for the correct value.
`truststoreFile`	The TrustStore file to use to validate client certificates.
`truststorePass`	The password to access the TrustStore. This defaults to the value of `keystorePass`.
`truststoreType`	Add this element if your are using a different format for the TrustStore then you are using for the KeyStore.
`keyAlias`	Add this element if your have more than one key in the KeyStore. If the element is not present the first key read in the KeyStore will be used.

After completing these configuration changes, you must restart Tomcat as you normally do, and you should be in business. You should be able to access any web application supported by Tomcat via SSL. For example, try:

```
https://localhost:8443
```

and you should see the usual Tomcat splash page (unless you have modified the ROOT web application). If this does not work, the following section contains some troubleshooting tips.

Installing a Certificate from a Certificate Authority

To obtain and install a Certificate from a Certificate Authority (like verisign.com, thawte.com or trustcenter.de), read the previous section and then follow these instructions:

Create a local Certificate Signing Request (CSR)

In order to obtain a Certificate from the Certificate Authority of your choice you have to create a so called Certificate Signing Request (CSR). That CSR will be used by the Certificate Authority to create a Certificate that will identify your website as "secure". To create a CSR follow these steps:

- Create a local Certificate (as described in the previous section):

```
keytool -genkey -alias tomcat -keyalg RSA \
        -keystore <your_keystore_filename>
```

- Note: In some cases you will have to enter the domain of your website (i.e. `www.myside.org`) in the field "first- and lastname" in order to create a working Certificate.
- The CSR is then created with:

```
keytool -certreq -keyalg RSA -alias tomcat -file certreq.csr \
        -keystore <your_keystore_filename>
```

Now you have a file called `certreq.csr` that you can submit to the Certificate Authority (look at the documentation of the Certificate Authority website on how to do this). In return you get a Certificate.

Importing the Certificate

Now that you have your Certificate you can import it into you local keystore. First of all you have to import a so called Chain Certificate or Root Certificate into your keystore. After that you can proceed with importing your Certificate.

- Download a Chain Certificate from the Certificate Authority you obtained the Certificate from.
  For Verisign.com commercial certificates go to:
  http://www.verisign.com/support/install/intermediate.html
  For Verisign.com trial certificates go to:
  http://www.verisign.com/support/verisign-intermediate-ca/Trial_Secure_Server_Root/index.html
  For Trustcenter.de go to:
  http://www.trustcenter.de/certservices/cacerts/en/en.htm#server
  For Thawte.com go to: http://www.thawte.com/certs/trustmap.html
- Import the Chain Certificate into your keystore

```
keytool -import -alias root -keystore <your_keystore_filename> \
        -trustcacerts -file <filename_of_the_chain_certificate>
```

- And finally import your new Certificate

```
keytool -import -alias tomcat -keystore <your_keystore_filename> \
        -file <your_certificate_filename>
```

Troubleshooting

Here is a list of common problems that you may encounter when setting up SSL communications, and what to do about them.

- I get "java.security.NoSuchAlgorithmException" errors in my log files.

  The JVM cannot find the JSSE JAR files. Follow all of the directions to download and install JSSE.

- When Tomcat starts up, I get an exception like "java.io.FileNotFoundException: {some-directory}/{some-file} not found".

  A likely explanation is that Tomcat cannot find the keystore file where it is looking. By default, Tomcat expects the keystore file to be named `.keystore` in the user home directory under which Tomcat is running (which may or may not be the same as yours :-). If the keystore file is anywhere else, you will need to add a `keystoreFile` attribute to the `<Factory>` element in the Tomcat configuration file.

- When Tomcat starts up, I get an exception like "java.io.FileNotFoundException: Keystore was tampered with, or password was incorrect".

  Assuming that someone has not *actually* tampered with your keystore file, the most likely cause is that Tomcat is using a different password than the one you used when you created the keystore file. To fix this, you can either go back and recreate the keystore file, or you can add or update the `keystorePass` attribute on the `<Connector>` element in the Tomcat configuration file. **REMINDER** - Passwords are case sensitive!

- When Tomcat starts up, I get an exception like "java.net.SocketException: SSL handshake errorjavax.net.ssl.SSLException: No available certificate or key corresponds to the SSL cipher suites which are enabled."

  A likely explanation is that Tomcat cannot find the alias for the server key withinthe specified keystore. Check that the correct `keystoreFile` and `keyAlias` are specified in the `<Connector>` element in the Tomcat configuration file. **REMINDER** - `keyAlias` values may be case sensitive!

If you are still having problems, a good source of information is the **TOMCAT-USER** mailing list. You can find pointers to archives of previous messages on this list, as well as subscription and unsubscription information, at http://tomcat.apache.org/lists.html.

Miscellaneous Tips and Bits

To access the SSL session ID from the request, use:

```
String sslID =
(String)request.getAttribute("javax.servlet.request.ssl_session");
```

For additional discussion on this area, please see Bugzilla.