Apache Tomcat

SSL/TLS Configuration How-To

ntroduction to SSL/TLS

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are technologies which allow 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/TLS 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/TLS 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. Details can be found in the Security Considerations Document. 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.

Tomcat is able to use any of the the cryptographic protocols that are provided by the underlying environment. Java itself provides cryptographic capabilities through JCE/JCA and encrypted communications capabilities through JSSE. Any compliant cryptographic "provider" can provide cryptographic algorithms to Tomcat. The built-in provider (SunJCE) includes support for various SSL/TLS versions like SSLv3, TLSv1, TLSv1.1, and so on. Check the documentation for your version of Java for details on protocol and algorithm support.

If you use the optional tcnative library, you can use the OpenSSL cryptographic provider through JCA/JCE/JSSE which may provide a different selection of cryptographic algorithms and/or performance benefits relative to the SunJCE provider. Check the documentation for your version of OpenSSL for details on protocol and algorithm support.

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 passport" for an Internet address. It states which organisation the site is associated with, along with some basic contact information about the site owner or administrator.

This certificate is cryptographically signed by its owner, and is therefore extremely difficult for anyone else to forge. For the certificate to work in the visitors browsers without warnings, it needs to be signed by a trusted third party. These are called *Certificate Authorities* (CAs). To obtain a signed certificate, you need to choose a CA and follow the instructions your chosen CA provides to obtain your certificate. A range of CAs is available including some that offer certificates at no cost.

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 signed by a well-known CA and are, therefore, not really guaranteed to be authentic at all. While self-signed certificates can be useful for some testing scenarios, they are not suitable for any form of production use.

General Tips on Running SSL

When securing a website with SSL it's important to make sure that all assets that the site uses are served over SSL, so that an attacker can't bypass the security by injecting malicious content in a JavaScript file or similar. To further enhance the security of your website, you should evaluate to use the HSTS header. It allows you to communicate to the browser that your site should always be accessed over https.

Using name-based virtual hosts on a secured connection requires careful configuration of the names specified in a single certificate or Tomcat 8.5 onwards where Server Name Indication (SNI) support is available. SNI allows multiple certificates with different names to be associated with a single TLS connector.

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, but keytool does not support that. So if your certificate has comments before the key data, remove them before importing the certificate with 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 JKS 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). The keytool prompt will tell you that pressing the ENTER key automatically uses the same password for the key as the keystore. You are free to use the same password or to select a custom one. If you select a different password to the keystore password, you will also need to specify the custom password in the server.xml configuration file.

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

Edit the Tomcat Configuration File

Tomcat can use two different implementations of SSL:

- JSSE implementation provided as part of the Java runtime
- JSSE implementation that uses OpenSSL

The exact configuration details depend on which implementation is being used. If you configured Connector by specifying generic protocol="HTTP/1.1" then the implementation used by Tomcat is chosen automatically.

Auto-selection of implementation can be avoided if needed. It is done by

specifying a classname in the **protocol** attribute of the Connector.

To define a Java (JSSE) connector, regardless of whether the APR library is loaded or not, use one of the following:

<!-- Define an HTTP/1.1 Connector on port 8443, JSSE NIO implementation -->

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"

sslImplementationName="org.apache.tomcat.util.net.jsse.JSSEImplementation
"

port="8443" .../>

<!-- Define an HTTP/1.1 Connector on port 8443, JSSE NIO2 implementation -->

<Connector protocol="org.apache.coyote.http11.Http11Nio2Protocol"

sslImplementationName="org.apache.tomcat.util.net.jsse.JSSEImplementation
"

port="8443" .../>

The OpenSSL JSSE implementation can also be configured explicitly if needed. If the Tomcat Native library or Java 22 is installed, using
the sslImplementationName attribute allows enabling it. When using the OpenSSL JSSE implementation, the configuration can use either the JSSE attributes or the OpenSSL attributes, but must not mix attributes from both types in the same SSLHostConfig or Connector element.

With Tomcat Native:

<!-- Define an HTTP/1.1 Connector on port 8443, JSSE NIO implementation and OpenSSL -->

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
port="8443"

sslImplementationName="org.apache.tomcat.util.net.openssl.OpenSSLImplementation"

.../>

With Java 22 FFM API:

<!-- Define an HTTP/1.1 Connector on port 8443, JSSE NIO implementation and OpenSSL -->

<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
port="8443"

sslImplementationName="org.apache.tomcat.util.net.openssl.panama.OpenSSLImplementation"

.../>

Alternately a listener can be added to the Server to enable OpenSSL on all connectors without having to add the sslImplementationName attribute on each.

With Tomcat Native:

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

With Java 22 FFM API:

<Listener className="org.apache.catalina.core.OpenSSLLifecycleListener"/>

The SSLRandomSeed attribute of the listeners allows specifying 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 the Connector in the $CATALINA_BASE/conf/server.xml file, where $CATALINA_BASE represents the base directory for the Tomcat instance. An example <Connector> element for an SSL connector is included in the default server.xml file installed with Tomcat.

To configure an SSL connector that uses JSSE with the JSSE configuration style, you will need to remove the comments and edit it so it looks something like this:

```
<!-- Define an SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector
        protocol="org.apache.coyote.http11.Http11NioProtocol"
        port="8443"
        maxThreads="150"
        SSLEnabled="true">
    <SSLHostConfig>
        <Certificate
            certificateKeystoreFile="${user.home}/.keystore"
            certificateKeystorePassword="changeit"
            type="RSA"
            />
        </SSLHostConfig>
</Connector>
```

The OpenSSL configuration style uses different attributes for many SSL settings, particularly keys and certificates. An example of an APR configuration style is:

```
<!-- Define an SSL Coyote HTTP/1.1 Connector on port 8443 -->
<Connector
        protocol="org.apache.coyote.http11.Http11NioProtocol"
        port="8443"
        maxThreads="150"
        SSLEnabled="true" >
    <SSLHostConfig>
```

```
        <Certificate
                certificateKeyFile="conf/localhost-rsa-key.pem"
                certificateFile="conf/localhost-rsa-cert.pem"
                certificateChainFile="conf/localhost-rsa-chain.pem"
                type="RSA"
                />
    </SSLHostConfig>
</Connector>
```

The configuration options and information on which attributes are mandatory, are documented in the SSL Support section of the HTTP connector configuration reference. Tomcat supports either configuration style (JSSE or OpenSSL) with all TLS connectors.

The port attribute 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 Specification.

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 self-signed 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>

Each Certificate Authority tends to differ slightly from the others. They may require slightly different information and/or provide the certificate and associated certificate chain in different formats. Additionally, the rules that the Certificate Authorities use for issuing certificates change over time. As a result you may find that the commands given above may need to be modified. If you require assitance then help is available via the Apache Tomcat users mailing list.

Using OCSP Certificates

Support of the Online Certificate Status Protocol (OCSP) in Apache Tomcat uses OpenSSL. This can be used either through Tomcat Native or the FFM API on Java 22 and newer.

To use OCSP, you require the following:

- OCSP-enabled certificates
- Tomcat with an OpenSSL enabled connector

- Configured OCSP responder

Generating OCSP-Enabled Certificates

Apache Tomcat requires the OCSP-enabled certificate to have the OCSP responder location encoded in the certificate. The basic OCSP-related certificate authority settings in the openssl.cnf file could look as follows:

#... omitted for brevity

[x509]

x509_extensions = v3_issued

[v3_issued]

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid,issuer

The address of your responder

authorityInfoAccess = OCSP;URI:http://127.0.0.1:8088

keyUsage = critical,digitalSignature,nonRepudiation,keyEncipherment,dataEncipherment,keyAgreement,keyCertSign,cRLSign,encipherOnly,decipherOnly

basicConstraints=critical,CA:FALSE

nsComment="Testing OCSP Certificate"

#... omitted for brevity

The settings above encode the OCSP responder address 127.0.0.1:8088 into the certificate. Note that for the following steps, you must have openssl.cnf and other configuration of your CA ready. To generate an OCSP-enabled certificate:

- Create a private key:

openssl genrsa -aes256 -out ocsp-cert.key 4096

- Create a signing request (CSR):
- openssl req -config openssl.cnf -new -sha256 \

-key ocsp-cert.key -out ocsp-cert.csr

- Sign the CSR:
- openssl ca -openssl.cnf -extensions ocsp -days 375 -notext \

-md sha256 -in ocsp-cert.csr -out ocsp-cert.crt

- You may verify the certificate:

openssl x509 -noout -text -in ocsp-cert.crt

Troubleshooting

Additional information may be obtained about TLS handshake failures by configuring the dedicated TLS handshake logger to log debug level messages by adding the following to $CATALINA_BASE/conf/logging.properties:

org.apache.tomcat.util.net.NioEndpoint.handshake.level=FINE

or

org.apache.tomcat.util.net.Nio2Endpoint.handshake.level=FINE

depending on the **Connector** being used.

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

- 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 certificateKeystoreFile attribute to the <Certificate> 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 error javax.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 within the specified keystore. Check that the correct certificateKeystoreFile and certificateKeyAlias are specified in the <Certificate> 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 https://tomcat.apache.org/lists.html.

Using the SSL for session tracking in your application

This is a new feature in the Servlet 3.0 specification. Because it uses the SSL session ID associated with the physical client-server connection there are some limitations. They are:

- Tomcat must have a connector with the attribute **isSecure** set to true.
- If SSL connections are managed by a proxy or a hardware accelerator they must populate the SSL request headers (see the SSLValve) so that the SSL

session ID is visible to Tomcat.

- If Tomcat terminates the SSL connection, it will not be possible to use session replication as the SSL session IDs will be different on each node.

To enable SSL session tracking you need to use a context listener to set the tracking mode for the context to be just SSL (if any other tracking mode is enabled, it will be used in preference). It might look something like:

```
package org.apache.tomcat.example;


import java.util.EnumSet;


import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.SessionTrackingMode;


public class SessionTrackingModeListener implements ServletContextListener {


    @Override
    public void contextDestroyed(ServletContextEvent event) {
        // Do nothing
    }


    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext context = event.getServletContext();
```

```
            EnumSet<SessionTrackingMode> modes =
                EnumSet.of(SessionTrackingMode.SSL);

            context.setSessionTrackingModes(modes);
        }

}
```

Miscellaneous Tips and Bits

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

```
String sslID =
(String)request.getAttribute("jakarta.servlet.request.ssl_session_id");
```

For additional discussion on this area, please see Bugzilla.

To terminate an SSL session, use:

```
// Standard HTTP session invalidation
session.invalidate();

// Invalidate the SSL Session
org.apache.tomcat.util.net.SSLSessionManager mgr =
    (org.apache.tomcat.util.net.SSLSessionManager)
    request.getAttribute("jakarta.servlet.request.ssl_session_mgr");
mgr.invalidateSession();

// Close the connection since the SSL session will be active until the connection
// is closed
```

response.setHeader("Connection", "close");

SSI How To

Introduction

SSI (Server Side Includes) are directives that are placed in HTML pages, and evaluated on the server while the pages are being served. They let you add dynamically generated content to an existing HTML page, without having to serve the entire page via a CGI program, or other dynamic technology.

Within Tomcat SSI support can be added when using Tomcat as your HTTP server and you require SSI support. Typically this is done during development when you don't want to run a web server like Apache.

Tomcat SSI support implements the same SSI directives as Apache. See the Apache Introduction to SSI for information on using SSI directives.

SSI support is available as a servlet and as a filter. You should use one or the other to provide SSI support but not both.

Servlet based SSI support is implemented using the class org.apache.catalina.ssi.SSIServlet. Traditionally, this servlet is mapped to the URL pattern "*.shtml".

Filter based SSI support is implemented using the class org.apache.catalina.ssi.SSIFilter. Traditionally, this filter is mapped to the URL pattern "*.shtml", though it can be mapped to "*" as it will selectively enable/disable SSI processing based on mime types. The contentType init param allows you to apply SSI processing to JSP pages, JavaScript, or any other content you wish.

By default SSI support is disabled in Tomcat.

Installation

CAUTION - SSI directives can be used to execute programs external to the Tomcat JVM.

To use the SSI servlet, remove the XML comments from around the SSI servlet

and servlet-mapping configuration in $CATALINA_BASE/conf/web.xml.

To use the SSI filter, remove the XML comments from around the SSI filter and filter-mapping configuration in $CATALINA_BASE/conf/web.xml.

Only Contexts which are marked as privileged may use SSI features (see the privileged property of the Context element).

Servlet Configuration

There are several servlet init parameters which can be used to configure the behaviour of the SSI servlet.

- **buffered** - Should output from this servlet be buffered? (0=false, 1=true) Default 0 (false).
- **debug** - Debugging detail level for messages logged by this servlet. Default 0.
- **expires** - The number of seconds before a page with SSI directives will expire. Default behaviour is for all SSI directives to be evaluated for every request.
- **isVirtualWebappRelative** - Should "virtual" SSI directive paths be interpreted as relative to the context root, instead of the server root? Default false.
- **inputEncoding** - The encoding to be assumed for SSI resources if one cannot be determined from the resource itself. Default is the default platform encoding.
- **outputEncoding** - The encoding to be used for the result of the SSI processing. Default is UTF-8.
- **allowExec** - Is the exec command enabled? Default is false.

Filter Configuration

There are several filter init parameters which can be used to configure the behaviour of the SSI filter.

- **contentType** - A regex pattern that must be matched before SSI

processing is applied. When crafting your own pattern, don't forget that a mime content type may be followed by an optional character set in the form "mime/type; charset=set" that you must take into account. Default is "text/x-server-parsed-html(;.*)?".

- **debug** - Debugging detail level for messages logged by this servlet. Default 0.
- **expires** - The number of seconds before a page with SSI directives will expire. Default behaviour is for all SSI directives to be evaluated for every request.
- **isVirtualWebappRelative** - Should "virtual" SSI directive paths be interpreted as relative to the context root, instead of the server root? Default false.
- **allowExec** - Is the exec command enabled? Default is false.

Directives

Server Side Includes are invoked by embedding SSI directives in an HTML document whose type will be processed by the SSI servlet. The directives take the form of an HTML comment. The directive is replaced by the results of interpreting it before sending the page to the client. The general form of a directive is:

<!--#directive [param=value] -->

The directives are:

- **config** - <!--#config errmsg="Error occurred" sizefmt="abbrev" timefmt="%B %Y" --> Used to set SSI error message, the format of dates and file sizes processed by SSI.
  All are optional but at least one must be used. The available options are as follows:
  errmsg - error message used for SSI errors
  sizefmt - format used for sizes in the **fsize** directive
  timefmt - format used for timestamps in the **flastmod** directive

- **echo** - <!--#echo var="VARIABLE_NAME" encoding="entity" --> will be replaced by the value of the variable.
  The optional **encoding** parameter specifies the type of encoding to use. Valid values are **entity** (default), **url** or **none**. NOTE: Using an encoding other than **entity** can lead to security issues.
- **exec** - <!--#exec cmd="file-name" --> Used to run commands on the host system.
- **exec** - <!--#exec cgi="file-name" --> This acts the same as the **include virtual** directive, and doesn't actually execute any commands.
- **include** - <!--#include file="file-name" --> inserts the contents. The path is interpreted relative to the document where this directive is being used, and IS NOT a "virtual" path relative to either the context root or the server root.
- **include** - <!--#include virtual="file-name" --> inserts the contents. The path is interpreted as a "virtual" path which is relative to either the context root or the server root (depending on
  the **isVirtualWebappRelative** parameter).
- **flastmod** - <!--#flastmod file="filename.shtml" --> Returns the time that a file was last modified. The path is interpreted relative to the document where this directive is being used, and IS NOT a "virtual" path relative to either the context root or the server root.
- **flastmod** - <!--#flastmod virtual="filename.shtml" --> Returns the time that a file was last modified. The path is interpreted as a "virtual" path which is relative to either the context root or the server root (depending on the **isVirtualWebappRelative** parameter).
- **fsize** - <!--#fsize file="filename.shtml" --> Returns the size of a file. The path is interpreted relative to the document where this directive is being used, and IS NOT a "virtual" path relative to either the context root or the server root.
- **fsize** - <!--#fsize virtual="filename.shtml" --> Returns the size of a file. The

path is interpreted as a "virtual" path which is relative to either the context root or the server root (depending on the **isVirtualWebappRelative** parameter).

- **printenv** - <!--#printenv --> Returns the list of all the defined variables.
- **set** - <!--#set var="foo" value="Bar" --> is used to assign a value to a user-defined variable.
- **if elif endif else** - Used to create conditional sections. For example:
- <!--#config timefmt="%A" -->
- <!--#if expr="$DATE_LOCAL = /Monday/" -->
- Meeting at 10:00 on Mondays
- <!--#elif expr="$DATE_LOCAL = /Friday/" -->
- Turn in your time card
- <!--#else -->
- Yoga class at noon.

<!--#endif -->

See the Apache Introduction to SSI for more information on using SSI directives.

Variables

SSI variables are implemented via request attributes on the **jakarta.servlet.ServletRequest** object and are not limited to the SSI servlet. Variables starting with the names "java.", "javax.", "sun" or "org.apache.catalina.ssi.SSIMediator." are reserved and cannot be used.

The SSI servlet currently implements the following variables:

Variable Name	Description
AUTH_TYPE	The type of authentication used for this user: BASIC, FORM, etc.

CONTENT_LENGTH	The length of the data (in bytes or the number of characters) passed from a form.
CONTENT_TYPE	The MIME type of the query data, such as "text/html".
DATE_GMT	Current date and time in GMT
DATE_LOCAL	Current date and time in the local time zone
DOCUMENT_NAME	The current file
DOCUMENT_URI	Virtual path to the file
GATEWAY_INTERFACE	The revision of the Common Gateway Interface that the server uses if enabled: "CGI/1.1".
HTTP_ACCEPT	A list of the MIME types that the client can accept.
HTTP_ACCEPT_ENCODING	A list of the compression types that the client can accept.
HTTP_ACCEPT_LANGUAGE	A list of the languages that the client can accept.
HTTP_CONNECTION	The way that the connection from the client is being managed: "Close" or "Keep-Alive".
HTTP_HOST	The web site that the client requested.
HTTP_REFERER	The URL of the document that the client linked from.
HTTP_USER_AGENT	The browser the client is using to issue the

	request.
LAST_MODIFIED	Last modification date and time for current file
PATH_INFO	Extra path information passed to a servlet.
PATH_TRANSLATED	The translated version of the path given by the variable PATH_INFO.
QUERY_STRING	The query string that follows the "?" in the URL.
QUERY_STRING_UNESCAPED	Undecoded query string with all shell metacharacters escaped with "\"
REMOTE_ADDR	The remote IP address of the user making the request.
REMOTE_HOST	The remote hostname of the user making the request.
REMOTE_PORT	The port number at remote IP address of the user making the request.
REMOTE_USER	The authenticated name of the user.
REQUEST_METHOD	The method with which the information request was issued: "GET", "POST" etc.
REQUEST_URI	The web page originally requested by the client.
SCRIPT_FILENAME	The location of the current web page on the server.

SCRIPT_NAME	The name of the web page.
SERVER_ADDR	The server's IP address.
SERVER_NAME	The server's hostname or IP address.
SERVER_PORT	The port on which the server received the request.
SERVER_PROTOCOL	The protocol used by the server. E.g. "HTTP/1.1".
SERVER_SOFTWARE	The name and version of the server software that is answering the client request.
UNIQUE_ID	A token used to identify the current session if one has been established.

CGI How To

Introduction

The CGI (Common Gateway Interface) defines a way for a web server to interact with external content-generating programs, which are often referred to as CGI programs or CGI scripts.

Within Tomcat, CGI support can be added when you are using Tomcat as your HTTP server and require CGI support. Typically this is done during development when you don't want to run a web server like Apache httpd. Tomcat's CGI support is largely compatible with Apache httpd's, but there are some limitations (e.g., only one cgi-bin directory).

CGI support is implemented using the servlet class org.apache.catalina.servlets.CGIServlet. Traditionally, this servlet is mapped to the URL pattern "/cgi-bin/*".

By default CGI support is disabled in Tomcat.

Installation

CAUTION - CGI scripts are used to execute programs external to the Tomcat JVM.

To enable CGI support:

1. There are commented-out sample servlet and servlet-mapping elements for CGI servlet in the default $CATALINA_BASE/conf/web.xml file. To enable CGI support in your web application, copy that servlet and servlet-mapping declarations into WEB-INF/web.xml file of your web application.

Uncommenting the servlet and servlet-mapping
in $CATALINA_BASE/conf/web.xml file enables CGI for all installed web applications at once.

2. Set privileged="true" on the Context element for your web application.

Only Contexts which are marked as privileged are allowed to use the CGI servlet. Note that modifying the global $CATALINA_BASE/conf/context.xml file affects all web applications. See Context documentation for details.

Configuration

There are several servlet init parameters which can be used to configure the behaviour of the CGI servlet.

- **cgiMethods** - Comma separated list of HTTP methods. Requests using one of these methods will be passed to the CGI script for the script to generate the response. The default value is GET,POST. Use * for the script to handle all requests regardless of method. Unless over-ridden by the configuration of this parameter, requests using HEAD, OPTIONS or TRACE will have handled by the superclass.
- **cgiPathPrefix** - The CGI search path will start at the web application root directory + File.separator + this prefix. By default there is no value, which results in the web application root directory being used as the search path. The recommended value is WEB-INF/cgi
- **cmdLineArgumentsDecoded** - If command line arguments are enabled (via **enableCmdLineArguments**) and Tomcat is running on Windows then

each individual decoded command line argument must match this pattern else the request will be rejected. This is to protect against known issues passing command line arguments from Java to Windows. These issues can lead to remote code execution. For more information on these issues see Markus Wulftange's blog and this archived blog by Daniel Colascione.

- **cmdLineArgumentsEncoded** - If command line arguments are enabled (via **enableCmdLineArguments**) individual encoded command line argument must match this pattern else the request will be rejected. The default matches the allowed values defined by RFC3875 and is [\w\Q%;/?:@&,$-.!~*'()\E]+
- **enableCmdLineArguments** - Are command line arguments generated from the query string as per section 4.4 of 3875 RFC? The default is false.
- **environment-variable-** - An environment to be set for the execution environment of the CGI script. The name of variable is taken from the parameter name. To configure an environment variable named FOO, configure a parameter named environment-variable-FOO. The parameter value is used as the environment variable value. The default is no environment variables.
- **executable** - The name of the executable to be used to run the script. You may explicitly set this parameter to be an empty string if your script is itself executable (e.g. an exe file). Default is perl.
- **executable-arg-1**, **executable-arg-2**, and so on - additional arguments for the executable. These precede the CGI script name. By default there are no additional arguments.
- **envHttpHeaders** - A regular expression used to select the HTTP headers passed to the CGI process as environment variables. Note that headers are converted to upper case before matching and that the entire header name must match the pattern. Default is ACCEPT[-0-9A-Z]*|CACHE-CONTROL|COOKIE|HOST|IF-[-0-9A-Z]*|REFERER|USER-AGENT

- **parameterEncoding** - Name of the parameter encoding to be used with the CGI servlet. Default is System.getProperty("file.encoding","UTF-8"). That is the system default encoding, or UTF-8 if that system property is not available.
- **passShellEnvironment** - Should the shell environment variables from Tomcat process (if any) be passed to the CGI script? Default is false.
- **stderrTimeout** - The time (in milliseconds) to wait for the reading of stderr to complete before terminating the CGI process. Default is 2000.

The CGI script executed depends on the configuration of the CGI Servlet and how the request is mapped to the CGI Servlet. The CGI search path starts at the web application root directory + File.separator + cgiPathPrefix. The **pathInfo** is then searched unless it is null - in which case the **servletPath** is searched.

The search starts with the first path segment and expands one path segment at a time until no path segments are left (resulting in a 404) or a script is found. Any remaining path segments are passed to the script in the **PATH_INFO** environment variable.

Proxy Support How-To

Introduction

Using standard configurations of Tomcat, web applications can ask for the server name and port number to which the request was directed for processing. When Tomcat is running standalone with the HTTP/1.1 Connector, it will generally report the server name specified in the request, and the port number on which the **Connector** is listening. The servlet API calls of interest, for this purpose, are:

- ServletRequest.getServerName(): Returns the host name of the server to which the request was sent.
- ServletRequest.getServerPort(): Returns the port number of the server to which the request was sent.
- ServletRequest.getLocalName(): Returns the host name of the Internet Protocol (IP) interface on which the request was received.

- ServletRequest.getLocalPort(): Returns the Internet Protocol (IP) port number of the interface on which the request was received.

When you are running behind a proxy server (or a web server that is configured to behave like a proxy server), you will sometimes prefer to manage the values returned by these calls. In particular, you will generally want the port number to reflect that specified in the original request, not the one on which the **Connector** itself is listening. You can use the proxyName and proxyPort attributes on the <Connector> element to configure these values.

Proxy support can take many forms. The following sections describe proxy configurations for several common cases.

Apache httpd Proxy Support

Apache httpd 1.3 and later versions support an optional module (mod_proxy) that configures the web server to act as a proxy server. This can be used to forward requests for a particular web application to a Tomcat instance, without having to configure a web connector such as mod_jk. To accomplish this, you need to perform the following tasks:

1. Configure your copy of Apache so that it includes the mod_proxy module. If you are building from source, the easiest way to do this is to include the --enable-module=proxy directive on the ./configure command line.
2. If not already added for you, make sure that you are loading the mod_proxy module at Apache startup time, by using the following directives in your httpd.conf file:
3. LoadModule proxy_module {path-to-modules}/mod_proxy.so
4. Include two directives in your httpd.conf file for each web application that you wish to forward to Tomcat. For example, to forward an application at context path /myapp:
5. ProxyPass /myapp http://localhost:8081/myapp

ProxyPassReverse /myapp http://localhost:8081/myapp

which tells Apache to forward URLs of the form http://localhost/myapp/* to the Tomcat connector listening on port 8081.

6. Configure your copy of Tomcat to include a special <Connector> element, with appropriate proxy settings, for example:
7. <Connector port="8081" ...
8. proxyName="www.mycompany.com"

   proxyPort="80"/>

which will cause servlets inside this web application to think that all proxied requests were directed to www.mycompany.com on port 80.

9. It is legal to omit the proxyName attribute from the <Connector> element. If you do so, the value returned by request.getServerName() will by the host name on which Tomcat is running. In the example above, it would be localhost.
10. If you also have a <Connector> listening on port 8080 (nested within the same Service element), the requests to either port will share the same set of virtual hosts and web applications.
11. You might wish to use the IP filtering features of your operating system to restrict connections to port 8081 (in this example) to be allowed **only** from the server that is running Apache.
12. Alternatively, you can set up a series of web applications that are only available via proxying, as follows:
    - Configure another <Service> that contains only a <Connector> for the proxy port.
    - Configure appropriate Engine, Host, and Context elements for the virtual hosts and web applications accessible via proxying.
    - Optionally, protect port 8081 with IP filters as described earlier.
13. When requests are proxied by Apache, the web server will be recording these requests in its access log. Therefore, you will generally want to

disable any access logging performed by Tomcat itself.

When requests are proxied in this manner, **all** requests for the configured web applications will be processed by Tomcat (including requests for static content). You can improve performance by using the mod_jk web connector instead of mod_proxy. mod_jk can be configured so that the web server serves static content that is not processed by filters or security constraints defined within the web application's deployment descriptor (/WEB-INF/web.xml).