

# SSL communication with Java and Tomcat

This document describes briefly how to enable Tomcat's SSL capabilities and shows how to make use of it out of a Java application. We review some issues and provide example code, demonstrating basic things to be done to get a working environment. Make sure you have Java Version 1.4 installed on your system since JSSE (Java Secure Socket Extension) is already included. If you are using an older version you need to install the JSSE package, which is separately available from Sun. JSSE contains the SSL (Secure Socket Layer) that provides all functionality to make use of secure Internet communication paths.

This document is independent from Casabac GUI Server - but is a summary of experiences we made when connecting our Casabac SWT Client (written in Java) to a server using http-ssl.

## Configuring Tomcat

### SSL Support

To enable SSL support, open the server's configuration file

```
"SCATALINA_HOME/conf/server.xml"
```

and just uncomment the element:

```
<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->
```

All settings may be left on the defaults. Tomcat will listen on port 8443 for HTTPS requests.

### Java Keytool

Java provides a command line tool to create an SSL certificate, which is requested by Tomcat when it tries to initialise. If you already got an officially signed certificate, you don't need this step. More below we describe how you can import it into the server's keystore. However, for testing purpose, or during development it's sufficient to use a private and unsigned one.

With the following command you create a new certificate in your OS default keystore:

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

If your OS default keystore (it contains the certificates) doesn't exist, a new one is created in your home folder and the new certificate becomes added. The keystore file is named ".keystore". Tomcat will automatically find it next time the server gets started (as long as Tomcat runs under your user name). You can rename/copy the keystore file to any location. Then, you have to adjust the server configuration file to let Tomcat know where to look up:

- Add a new attribute the new location to the "SSL HTTP/1.1 Connector" entry in server.xml:

```
KeystoreFile="C:\dir\myKeystoreFile"
```

- If the password is different from "changeit", add the password attribute as well:

```
KeystorePass="changed"  
Signed Certificates
```

If you purchased a signed certificate from one of the global certificate authorities (i.e. VeriSign Inc.) you may not need to create your own. Simply import it into your server side keystore file:

```
%JAVA_HOME%\bin\keytool -import -file <certificate> -keystore changeit -keystore <keystore>
```

More detailed information on Tomcat can be found on <http://jakarta.apache.org/tomcat/>.

## SSL With Internet Explorer

After restarting Tomcat you should try first if you can access your pages either via HTTP or HTTPS, i.e.:

<https://localhost:8443/MyApplication/index.html>

<http://localhost:8080/MyApplication/index.html>

## SSL With Java

Now try to access the same page from your Java code. A primitive example on how that could be done is given below. If your server has a signed certificate installed in its keystore, it should work from scratch and you should see the source code of your web page on the OS console. However, if the certificate is not signed and hence unknown by the underlying Java SSL framework, some other steps need to be done to make it work (see next sections).

```
public void loadMyPage()  
{  
    try  
    {  
        java.net.URLConnection urlc = new java.net.URL("https://localhost:8443/index.html").openConnection();  
        java.io.InputStreamReader isr = new java.io.InputStreamReader(urlc.getInputStream(), "UTF8");  
        java.io.BufferedReader br = new java.io.BufferedReader(isr);  
        StringBuffer sb = new StringBuffer();  
  
        while (true)  
        {  
            String s = br.readLine();  
            if (s == null) break;  
            sb.append(s + "\n");  
        }  
  
        br.close();  
        System.out.print(sb.toString());  
    }  
    catch (Exception e)  
    {  
        e.printStackTrace();  
    }  
}
```

## Dealing With Client Side Authentication Issues

Trying to connect to Tomcat via Java and HTTPS may result in the following exception:

```
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: No trusted  
certificate found
```

When Tomcat does not respond with a signed certificate the client side SSL layer does not accept it. This means, the JRE could not authenticate the server's certificate by means of its local keystore and therefore the server might not be the one it claims to be. Each JRE contains a pre configured standard keystore (or truststore) for authenticating signed certificates. If you are interested in, have a look at:

```
%JAVA_HOME%\jre\lib\security\cacerts
```

## Replacing The Trust Manager

One way to overcome the SSL handshake exception is to provide and register a custom-made trust manager in your Java application. What the code below does, it accepts all kinds of server provided SSL certificates regardless whether signed or unsigned. Just run it once before you attempt to initiate the URL connection:

```
private void registerMyTrustManager()
{
    TrustManager[] trustAll = new javax.net.ssl.TrustManager[]
    {
        new javax.net.ssl.X509TrustManager()
        {
            public java.security.cert.X509Certificate[] getAcceptedIssuers(){ return null; }
            public void checkClientTrusted(java.security.cert.X509Certificate[] certs, String authType){}
            public void checkServerTrusted(java.security.cert.X509Certificate[] certs, String authType){}
        }
    };

    try
    {
        javax.net.ssl.SSLContext sc = javax.net.ssl.SSLContext.getInstance("SSL");
        sc.init(null, trustAll, new java.security.SecureRandom());
        javax.net.ssl.HttpURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

Now all certificates (signed and unsigned) become accepted and the exception disappears. Of course this is not advisable for a production system but quite useful for testing.

## Providing a Custom Key Store

A more laborious but recommended procedure to avoid the SSL handshake exception is done by letting the JRE address a keystore containing server's real certificate. For testing purpose you could simply use the one you have created during Tomcat configuration. Just pass the keystore location and password as JVM parameters when you start your application. Inside your Java code, it would be the same to set the two parameters as system properties.

```
-Djavax.net.ssl.trustStore=C:/myFolder/myKeyStoreFile
-Djavax.net.ssl.trustStorePassword=changeit
```

One step ahead it's maybe required to move an unsigned Tomcat certificate from Tomcat's keystore into your own, or even the JRE keystore. Thereby all required certificates could accumulate in one repository.

a) Export:

```
%JAVA_HOME%\bin\keytool -export -alias tomcat -keypass changeit -file C:\myCerti
```

b) Import:

```
%JAVA_HOME%\bin\keytool -import -file C:\myCerti -keypass changeit -keystore JAVA_HOME%\jre\lib\security\cacerts
```

## Patching Hostname Verification

If you did one of the previously described approaches and re-run your code you will immediately notice the story is not over now, because your client throws an IOException:

```
java.io.IOException: HTTPS hostname wrong:  should be <localhost>
```

This error message is quite misleading; it means your URL doesn't match the name/IP of your server, which is an important security check but unfounded in our arbitrary environment. However, in real life it's considerably important to ensure that the URL hostname and the server's identification hostname match.

To circumvent that misbehaviour you need to do one last step. The following code snippet figures out how to set a special hostname verifier that is called to confirm the host name; it always confirms any:

```
private void registerMyHostnameVerifier()
{
    javax.net.ssl.HostnameVerifier myHv = new javax.net.ssl.HostnameVerifier()
    {
        public boolean verify(String hostName, javax.net.ssl.SSLSession session)
        {
            return true;
        }
    }
    javax.net.ssl.HttpURLConnection.setDefaultHostnameVerifier(myHv);
}
```

## Summary

Now you're done and the initially provided example should work. A summarized cooking recipe would look like this:

- Create a keystore for Tomcat: use Java key tool
- Configure Tomcat: uncomment the HTTPS section, restart, make a smoke test with Internet Explorer
- Call the methods to install trust manager and hostname verifier
- Create a HTTPS URL connection and load your data