Spring Security
Lab 1

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 8 / 11 (note: labs are tested with JDK 8 but should work on JDK 11 with no or minimal changes)
- Spring Tool Suite (STS) IDE
- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files

2 Setting up TLS for one-way (server) authentication

SSL/TLS provides two modes of authentication, which happen during the early stages of the handshake process after the server returns the ServerHello. In the most common mode (one way or server authentication), the server sends it certificate to client who validates it using the certification path algorithm. In the mutual or client authentication mode, the client additionally returns its certificate to the server for authentication.

Client certificates are primarily used as a form of MFA to supplement the standard username/password SFA used to authenticate identity of client / user to server. Client certificates are

not used for any cryptographic purpose (signing / encryption) other than authentication: the subsequent symmetric session key established during key exchange is accomplished via the public key in the server certificate.

When setting up TLS on the server and/or client end, we will be using key stores and trust stores. A trust store is essentially a key store with a more specific purpose. The key store of an entity X (which can be either the client or server) would contain both private and public keys for certificates belonging to X. On the other hand, the trust store of entity X would contain only certificates (such as trusted root certificates and intermediate certificates) used to verify end-entity certificates received from other parties (for e.g. entity Y).

The JDK comes bundled with a truststore called `cacerts` which resides in the `%JAVA_HOME%/jre/lib/security` directory that is based on the Oracle Root program. By default, all Java apps (including Spring Boot apps) will use this trust store, unless a customized truststore is explicitly configured for the app to use.

When X communicates with Y and Y needs to authenticate itself to X, the trusted root certificates from entity Y's key store will need to be imported into the trust store of X (if they don't already exist there).

2.1 Server setup for one-way authentication

The main folder for this lab is `SSL-Demo`

Start up STS. Switch to the Java perspective.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `SSLServerDemo`
Group: `com.workshop.security`
Artifact: `SSLServerDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Configuration for a SSL Server`
Package: `com.workshop.security`

Add the following dependencies:
```
Web -> Spring Web
Developer Tools -> Lombok
Developer Tools -> Spring Boot DevTools
```

At the command prompt shell in any directory, create a new key store for the server with a single self-signed certificate:

```
keytool -genkeypair -alias server-keypair -keyalg RSA -keysize 2048 -sigalg SHA256withRSA -dname "cn=marvel.com,ou=marvel comics,o=marvel inc,l=NY city,ST=New York,C=US" -validity 365 -ext SAN=dns:localhost,ip:127.0.0.1 -ext KU=digitalSignature,keyCertSign -ext EKU=anyExtendedKeyUsage -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit
```

Here we add in additional extension fields for the Key Usage, which are relevant for a self-signed certificate that will be used in the fashion of a root CA certificate (i.e. to sign another intermediate or end-entity certificate). For more info on the options for these arguments, go to: https://docs.oracle.com/en/java/javase/13/docs/specs/man/keytool.html#supported-named-extensions

Copy this new key store `server-keystore.p12` into `src/main/resources`.

In `src/main/resources`, place the contents below in `application.properties`

```
server.port=8443
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:server-keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-alias=server-keypair
```

In `src/main/java`, place the files

```
Developer
DeveloperController
```

The properties in `application.properties` should be self-explanatory. The `server.ssl.key-alias` property specifies the alias of the certificate to return to a client that connects to this server during the TLS handshake process. Remember ONLY the certificate is returned, NOT the private key that is paired to that certificate. Private keys are ALWAYS kept private in the key store and not distributed to any party.

Another important point to note is that for high security environments, we should not explicitly specify the key store password directly this file (particularly when the key store contains a private key !!). Instead we can set the key store password as an environment variable from the command shell, and then read its property to initialize `server.ssl.key-store-password`.

While it might be safe to specify the key store password directly if the source code only ever remains on a trusted environment, the primary risk is that the source code for this app might be accidentally uploaded to public environment (e.g. GitHub) without removing the key store password. This presents a security loophole that can be used by a diligent hacker as step to compromise the system.

By default, TLS runs on port 443 on a live production server. In the embedded (and also standalone) Tomcat server, we typically use port 8443 instead.

The Developer class provides the model for a simple class to be serialized and returned as JSON, while the Controller class provides with some basic REST API methods (GET, POST, PUT, DELETE) to allow us to test the API over HTTPS.

Run the app in the usual manner from the Boot Dashboard.

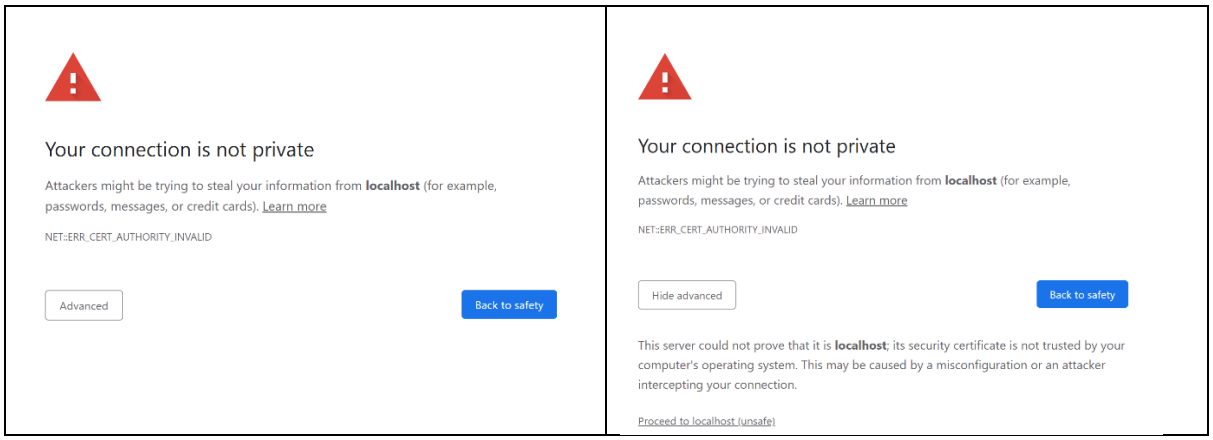
2.2 Browser setup for one-way authentication

Using a browser, connect to the app's deployment port using HTTPS

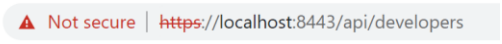
https://localhost:8443/api/developers

A security warning is flagged as the self-signed certificate submitted by the server (the REST API app) is not recognized as a valid root CA (i.e. this certificate is not present in the Trusted Root CA store for the current user or computer).

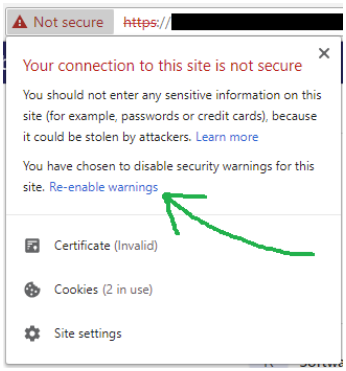
Click on Advanced and then Proceed to localhost (unsafe)



You should see the address bar indicating the URL as being insecure, with the JSON returned displayed in the browser as usual.



Now if you open a new browser tab and type in the same URL again, the previous security warning is no longer shown because this site is flagged as a security exception in the browser. If you want the browser to show the security warning again every time you visit this URL, you have to re-enable the warning.



We will now import this self-signed certificate that we created earlier into the Trusted Root CA store of the current user, which will effectively make it into a root certificate on our system.

Navigate in a command prompt to the directory where you initially created `server-keystore.p12`, and type

```
keytool -exportcert -rfc -alias server-keypair -file server.cer -
keystore server-keystore.p12 -storepass changeit -storetype PKCS12
```

Double click on `server.cer` to open the certificate, then click Install Certificate. This opens the Certificate Import Wizard. Select Current User for the store location, then in the certificate store section, select the option Place all certificates in the following store. Then click Browse and select Trusted Root Certification Authorities, select OK and select Next. Then click Finish.
A security warning should appear about installing the certificate. Click Yes.

Now open the certificate store for the current user, by typing `certmgr.msc` at the run command from the Windows Start icon.

You should now be able to see the `marvel` self-signed certificate installed there.

You can also view the Trusted Root CA section from the browser.

Chrome: Settings -> Privacy and Security -> Security (Safe Browsing) -> Manage Certificates
Firefox: Settings -> Privacy and security -> Security Certificates -> View Certificates

Close the browser, open a new tab and type in the same URL again:

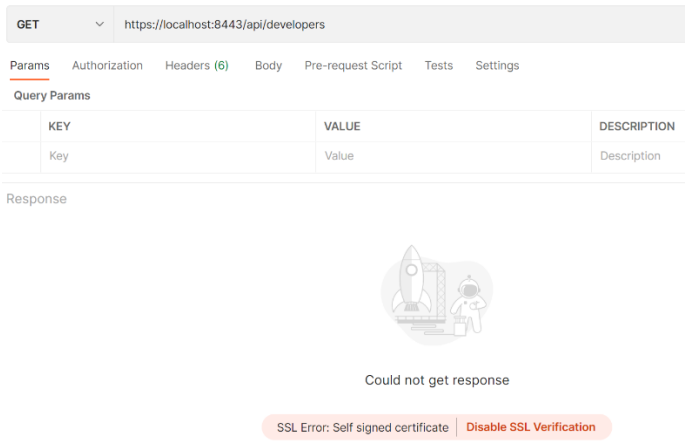
https://localhost:8443/api/developers

This time no errors appear since the self-signed certificate submitted by the server is identical to the one in the Trusted Root CA section, which validates it immediately. You can click on the padlock to view the self-signed certificate from the server in the usual manner.

2.3 Postman setup for one-way authentication

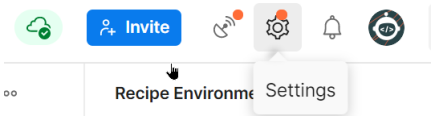
We will first attempt accessing the REST API app using Postman. Start up Postman and issues a GET to the same URL:
https://localhost:8443/api/developers

Notice that an error response is returned:

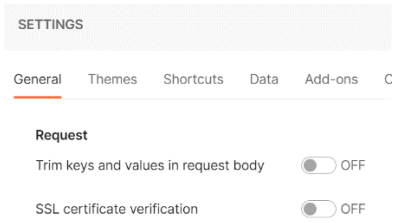


This is because Postman by default is configured not to accept self-signed certificates as a security measure. Remember that valid certificates are always signed by an intermediate CA, so a server should never return a self-signed certificate under normal circumstances.

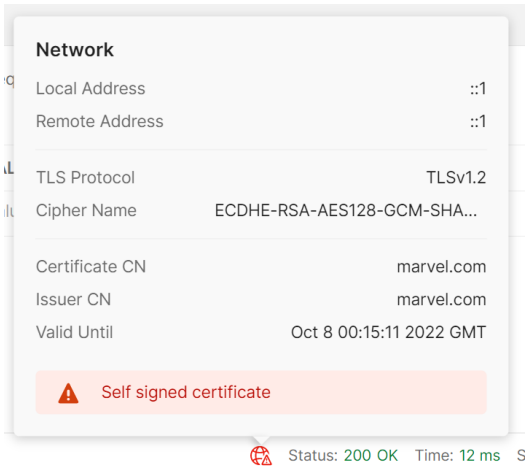
If we are doing in-house development using a self-signed certificate (such as in this case), we can disable SSL verification.
Select Settings.



In the General tab, turn off SSL verification.



Now if you send a GET again to the same URL, you should get back the expected response. In addition, you will see a warning regarding the self-signed certificate in the network icon in the response area. Notice as well that this provides info on the particular version of TLS being used as the cipher suite negotiated for the handshake process.



Next send a POST to the same URL

https://localhost:8443/api/developers

with this raw JSON content for the body

```
{
    "id": 2,
    "name": "Superman",
    "age": 44,
    "languages": [
        "JavaScript",
        "PHP"
    ],
    "married": true
}
```

and again verify the request was received correctly from the log output from the server.

Finally, send a PUT and DELETE to this URL

https://localhost:8443/api/developers/888

and verify again the request was received correctly from the log output from the server.

2.4 RestTemplate client setup for one-way authentication

Next, we will create a basic Rest Template client to communicate with this REST API.

Go to File -> New -> Spring Starter Project. Complete it with the following details:

Name: `SSLClientDemo`
Group: `com.workshop.security`
Artifact: `SSLClientDemo`
Version: `0.0.1-SNAPSHOT`
Description: `Demo Configuration for a SSL Client`
Package: `com.workshop.security`

Add the following dependencies:
`Web -> Spring Web`
`Developer Tools -> Lombok`
`Developer Tools -> Spring Boot DevTools`

In `src/main/resources`, place the contents below in `application.properties`

```
myrest.url=https://localhost:8443/api/developers
spring.main.web-application-type=none
```

In `src/main/java`, place the files

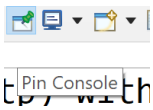
```
Developer
MainConfig
MyRestService
MyRunner
```

All of these provide a simple implementation of a RestTemplate client to invoke the basic REST API methods already provided in DeveloperController of `SSLServerDemo`

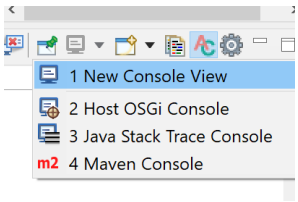
Before starting up `SSLClientDemo`, you will need to make some configuration changes so that you can see the log output from both this app and the already running `SSLServerDemo`. In order to see the Console output from these two applications, you will have to:

- pin the current console view showing output from `SSLServerDemo`
- open a new separate Console view
- ensure that the new Console view does not switch to a different application when standard output changes
- Run `SSLClientDemo` in the new Console view

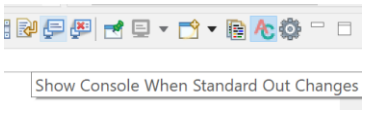
In the current Console view, select Pin Console from the icon at the upper right hand corner of the Console View



Then, select New Console View from the drop-down list of the icon at the upper right hand corner of the Console View.



Switch over to this new Console view. You will notice it is currently identical to the older one. Make sure the option Show Console When Standard Out changes is deselected from the icon at the upper right hand corner of the Console View



With this new console view active, select `SSLClientDemo` from the Boot Dashboard and run it. You should see the log output from `SSLClientDemo` appear in this new console view, while the old console view shows the latest log output from `SSLServerDemo`

At this point of time, `SSLClientDemo` will crash with various exceptions related to failure in the SSL Handshake process and the resulting inability to find a valid certification path to the requested target. This is because the required self-signed certificate `server.cer` is not available in the trust store used by the app. If no trust store is explicitly configured for the app, the default trust store that is used is `cacerts` which resides in the `%JAVA_HOME%/jre/lib/security` directory

There are three ways to configure the RestTemplate client application in order for it to work correctly:

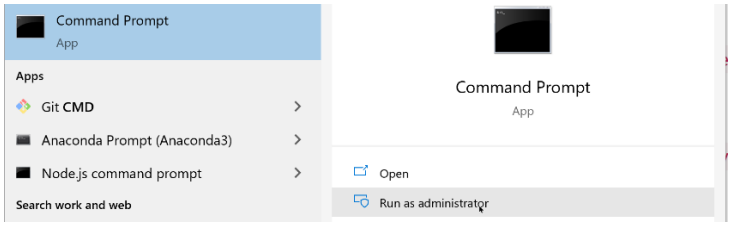
a) Import the self-signed or untrusted certificate into the default Java truststore `cacerts`.

b) Configure SSL in the HTTPClient used in the RestTemplate object to accept certificates of a certain type

c) Configure SSL in the HTTPClient to use a custom trust store containing the same self-signed certificate / untrusted certificate submitted by the REST service in the initial SSL/TLS handshake

2.4.1 Setting up by importing into cacerts

To import `server.cer` into `cacerts`, start up a command prompt shell with admin privilege:



Navigate to the directory containing `server.cer` that you created earlier, and type the following below (making sure you use the location for own JDK installation on your system):

```
keytool -importcert -trustcacerts -alias marvel -keystore "C:\Program Files\Java\jdk1.8.0_251\jre\lib\security\cacerts" -file server.cer -rfc -storepass changeit
```

and type yes to the prompt regarding importing the certificate.

When this is complete, double check that the certificate is in the trust store with:

```
keytool -list -alias marvel -keystore "C:\Program Files\Java\jdk1.8.0_251\jre\lib\security\cacerts" -storepass changeit -v
```

Now run `SSLClientDemo` from the Boot Dashboard, and this time it should work with the correct log output on the client and server side.

While this approach is reasonably fast to implement, it presents a potential security hole since we now have an inauthentic certificate in the `cacerts` file which will also be used by default by another Java apps running on the same machine. This can be problematic if the machine / server is also being used to deploy apps written by other developers who are not aware of this “workaround” solution used.

Before attempting the next approaches, let’s delete the `marvel` certificate from `cacerts`. This command should again be run from a command prompt with system privilege:

```
keytool -delete -alias marvel -keystore "C:\Program Files\Java\jdk1.8.0_251\jre\lib\security\cacerts" -storepass changeit -v
```

Now run `SSLClientDemo` from the Boot Dashboard, and it should fail again with the expected exceptions.

2.4.2 Setting up by configuring to trust all certificates

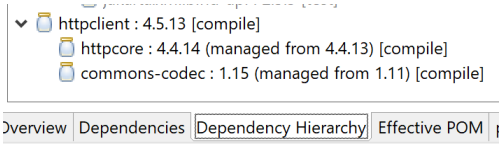
For the next two approaches, we will be using classes from HTTPClient.

Replace `pom.xml` in your project with `pom.xml` from `changes`.

This incorporates an additional managed dependency for HTTPClient:

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

We will be using classes from HTTPClient (such as SSLConnectionSocketFactory) as well as HttpCore (such as SSLContextBuilder) so ensure that you referring to the API documentation for this based on the version for the managed dependency in the project POM (typically this will be the latest version). You can double check on this in the dependency hierarchy.



The latest relevant classes are:

https://www.javadoc.io/doc/org.apache.httpcomponents/httpclient/latest/org/apache/http/conn/ssl/SSLConnectionSocketFactory.html
https://javadoc.io/doc/org.apache.httpcomponents/httpcore/latest/org/apache/http/ssl/SSLContextBuilder.html
https://www.javadoc.io/doc/org.apache.httpcomponents/httpclient/latest/org/apache/http/conn/ssl/NoopHostnameVerifier.html

The key focus here is on configuring the RestTemplate object properly in the factory @Bean method that produces it. The business logic that consumes this object is exactly the same as for a standard RestTemplate client, so we won't need to make any changes here:

For approach b), we can either:

- use the TrustAllStrategy, which essentially configures the TrustManager to validate all certificates
- use a TrustSelfSignedStrategy, which configures the TrustManager to validate all self-signed certificates
- load without a TrustStore (i.e. null), which results in no Trust Manager, and therefore no validation of certificates

In the package `com.workshop.security` in `src/main/java`, make the following changes

`MainConfig-v2`

Now run `SSLClientDemo` from the Boot Dashboard, and this time it should work with the correct log output on the client and server side.

This approach is now better than a) because we do not introduce a potential security loophole by putting an inauthentic self-signed certificate into `cacerts`.

2.4.3 Setting up by configuring to validate from a custom trust store

In certain situations however, we may wish to activate certificate validation as normal but validate in a customized manner (for e.g. validating against a custom trust store, rather than `cacerts`). This is essentially approach c) mentioned earlier.

We can create a new custom trust store that contains the self-signed marvel certificate that we produced earlier by typing:

```
keytool -importcert -trustcacerts -alias marvel -file server.cer -
storetype PKCS12 -keystore client-truststore.p12 -storepass changeit
```

Next we confirm for the presence of this certificate in the custom trust store:

```
keytool -list -storetype PKCS12 -keystore client-truststore.p12 -
storepass changeit -v
```

Notice that only the certificate is here, not the private key paired with the certificate, which is still in its original key store `server-keystore.p12`

Now, we copy `client-truststore.p12` and place in `src/main/resources` of `SSLClientDemo`

Set up the additional following properties in `application.properties` to configure the app to access the trust store:

```
myrest.url=https://localhost:8443/api/developers
spring.main.web-application-type=none

trust.store.location=classpath:client-truststore.p12
trust.store.password=changeit
```

In the package `com.workshop.security` in `src/main/java`, make the following changes

`MainConfig-v3`

We now configure the @Bean method to use the trust store that we have placed on our classpath in order to perform certificate validation in the usual manner.

Now run `SSLClientDemo` from the Boot Dashboard, and again it should work with the correct log output on the client and server side.

So far, we have been working with only a single self-signed certificate, which would be representative of a root certificate in a real-life scenario. We have seen that servers should be working off an end-entity certificate rather than a self-signed certificate, so let's simulate creating an end-entity certificate that is signed by the root certificate. We will then install this end-entity certificate onto our REST API app `SSLServerDemo`

Type the following commands in a command prompt in the directory containing `server-keystore.p12`

First we create another key pair and accompanying certificate with the alias `ironman`:

```
keytool -genkeypair -alias ironman -keyalg RSA -keysize 2048 -dname "cn=ironman.com,ou=supersuits,o=Stark Industries inc,l=Boston,ST=Massachusetts,C=US" -validity 365 -ext SAN=dns:localhost,ip:127.0.0.1 -ext KU=digitalSignature,keyCertSign -ext EKU=anyExtendedKeyUsage -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit
```

Then we generate a CSR for this certificate and use our original self-signed certificate to sign the CSR

```
keytool -certreq -alias ironman -file ironmanrequest.csr -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit

keytool -alias server-keypair -gencert -sigalg SHA256withRSA -ext san=dns:localhost,ip:127.0.0.1 -ext KU=digitalSignature,keyCertSign -ext EKU=anyExtendedKeyUsage -infile ironmanrequest.csr -outfile signedironman.cer -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit
```

Finally, we import back the signed certificate into the same alias and view it to ensure that we can see a certificate chain of length 2:

```
keytool -importcert -alias ironman -file signedironman.cer -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit

keytool -list -alias ironman -storetype PKCS12 -keystore server-keystore.p12 -storepass changeit -v
```

Stop the `SSLServerDemo` app if it is still running

Copy this latest version of `server-keystore.p12` to `src/main/resources` of `SSLServerDemo`

Modify `application.properties` of this project to use the newly created and signed `ironman` certificate when going through TLS handshake with a client:

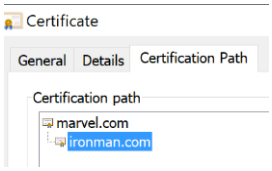
```
server.port=8443
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:server-keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-alias=ironman
```

Restart `SSLServerDemo`

Using a new browser tab, connect to the app's deployment port using HTTPS

https://localhost:8443/api/developers

There should not be any security warning flagged (assuming that the `marvel` self-signed certificate is still installed in the Trusted Root CA section for the current user). If you view the certificate in the browser, you should clearly see a certification path of length 2 starting from `marvel` and ending at `ironman`.



Restart `SSLClientDemo`

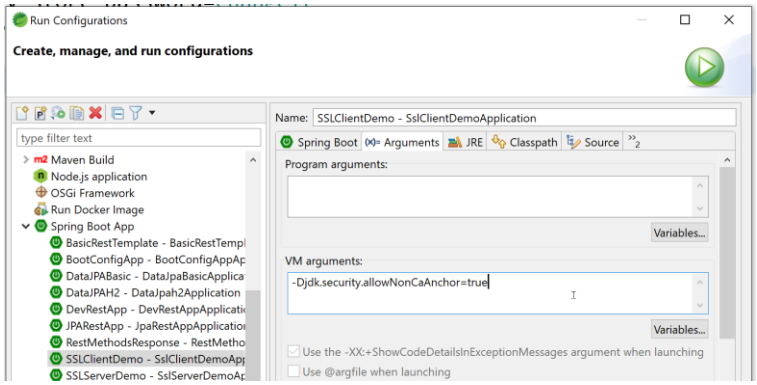
Notice that it will crash with this error:
`Caused by: sun.security.validator.ValidatorException: TrustAnchor with subject "CN=marvel.com, OU=marvel comics, O=marvel inc, L=NY city, ST=New York, C=US" is not a CA certificate`

This is because the validation algorithm is capable of recognizing the self-signed certificate `marvel` currently in `client-truststore.p12` is not a proper root CA certificate (i.e. it was generated using the keytool program instead).

To overcome this issue, we could reset the MainConfig class back to the previous version of `MainConfig-v2`, which simply recognizes all certificates. However, if we want to allow the client to construct a certification path that includes both ironman and marvel, we can set the following property in the VM arguments field of Run configuration for `SSLClientDemo`

`-Djdk.security.allowNonCaAnchor=true`

Right click on the Project, select Run As -> Run Configuration, then enter the property above in the VM arguments field and click Apply and then Run.



This time it should work.

For all of the above approaches, if we wish to see a complete trail of handshake messages in the setup of TLS, add the following VM arguments as well to the Run Configuration for `SSLClientDemo`.

```
-Dhttps.protocols=TLSv1.2,TLSv1.1,TLSv1
-Djavax.net.debug=ssl:handshake:verbose:keymanager:trustmanager
-Djava.security.debug=access:stack
```

Typical real life scenarios involve at least one intermediate certificate (resulting in a chain length of 3). However, we cannot simulate this here because of security restrictions in Windows and also Java that will not recognize any further certificates signed via the `ironman` key pair and certificate. However, if authentic root and intermediate certificates are used, this code implementation should work perfectly fine.

2.5 RestTemplate client setup for custom trust store with valid certificates

We will demonstrate by making calls to this public REST API:

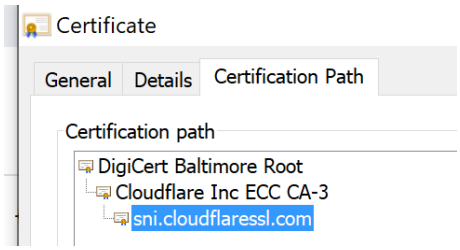
https://www.exchangerate-api.com/

To proceed, get a free key by entering your email address.

Make a sample call using this key to check the results on Postman or your browser and verify it is working properly:

https://v6.exchangerate-api.com/v6/YOUR-API-KEY/latest/USD

The server certificate for this site is in a certificate chain of 3 that links back to a trusted root CA in the local user store.



We will now modify `SSLClientDemo` to make a call to this API endpoint using the default trust store `cacerts`. As `cacerts` already has the DigiCert Baltimore Root CA installed in it, the calls should work fine without any problems.

Make the following changes to `application.properties`

```
myrest.url=https://v6.exchangerate-api.com/v6
my.key=put your api key here
spring.main.web-application-type=none
```

In the package `com.workshop.security` in `src/main/java`, place the following files:

`RemoteRestService`

In the package `com.workshop.security` in `src/main/java`, make the following changes

`MainConfig (switch back to the original version)`
`MyRunner-v2`

Remove this option from the VM arguments in the Run Configuration

`-Djdk.security.allowNonCaAnchor=true`

Then click Apply and Run. The client successfully makes a API call to the endpoint and logs the output result. This is because, as mentioned earlier, when there is no explicit configuration of a custom trust store to use, the default trust store `cacerts` is utilized, which already has the required root certificate to validate the certification path using the TLS certificate returned from the server.

Let's now explicitly configure to use a custom trust store in the manner that we have done earlier.

Make the following changes to `application.properties`

```
myrest.url=https://v6.exchangerate-api.com/v6
my.key=put your api key here
spring.main.web-application-type=none

trust.store.location=classpath:client-truststore.p12
trust.store.password=changeit
```

In the package `com.workshop.security` in `src/main/java`, make the following changes

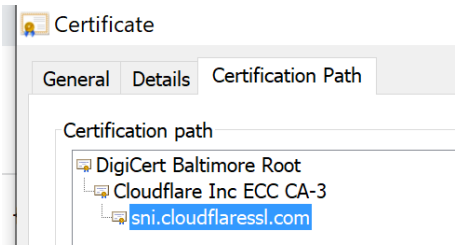
`MainConfig-v3`

Now run `SSLClientDemo` again. This time it fails, since we are using the custom trust store `client-truststore.p12` which does not contain the root and intermediate certificates required to validate the TLS certificate returned by the client.

Return to the browser and make a call using the URL below (if you haven't already done so earlier)

https://v6.exchangerate-api.com/v6/YOUR-API-KEY/latest/USD

and check the server certificate for this site. This lies in a certificate chain of 3 with the root CA being DigiCert Baltimore Root and the intermediate CA being Cloudflare Inc.



We are going to import the root CA and intermediate CA certificates one at a time and then import them into our custom trust store `client-truststore.p12`. There is a possibility of importing all the certificates as a certificate chain in P7B format, but this format is not recognized by keytool, so we will work instead at importing them one at a time.

Double click on DigiCert Baltimore Root in the Certification Path tab which will bring up a separate Certificate Dialog box for it. In this new box, go to Details tab and select Copy To File. This brings up the Certificate Export Wizard. Select the Base 64 encoded x509 (.CER) format and save it as `baltimoreroot.cer` in a suitable location.

Repeat this for the Cloudflare Inc ECC intermediate cert and save it as `cloudflareinc.cer` in a suitable location.

We do not need to save the final certificate (`sni.cloudflaressl.com`) as this certificate will be passed to the client during the TLS handshake.

Copy both these files to where `client-truststore.p12` is located at.

In a command prompt, type:

```
keytool -importcert -trustcacerts -alias baltimorerootcert -file baltimoreroot.cer -storetype PKCS12 -keystore client-truststore.p12 -storepass changeit
```

You will be given a warning to which you should response to with no.

```
Certificate already exists in system-wide CA keystore under alias <baltimorecybertrustca [jdk]>
Do you still want to add it to your own keystore? [no]:  no
Certificate was not added to keystore
```

This shows that the certificate validation algorithm will still consult the `cacerts` trust store to find root CA certificates, if it does not find them in the custom trust store. So in this case, we can forego the need to import `baltimoreroot.cer`

However, we will still need to import the intermediate certificate `cloudflareinc.cer` with:

```
keytool -importcert -trustcacerts -alias cloudflareinccert -file cloudflareinc.cer -storetype PKCS12 -keystore client-truststore.p12 -storepass changeit
```

Now check the contents of `client-truststore.p12` with:

```
keytool -list -storetype PKCS12 -keystore client-truststore.p12 -storepass changeit -v
```

There should only be 2 entries: our original self-signed certificate `marvel` as well as the newly imported `cloudflareinccert`

Finally, we can copy and paste `client-truststore.p12` into `src/main/resources` of `SSLClientDemo`

Now run `SSLClientDemo` again. This time it should succeed.

Here we have demonstrated how can import one intermediate certificate into our custom trust store in order to complete the certificate chain path for validating TLS certificates that we receive from a server. We can easily extend this process to cater for certificate chains that exceed the length of 3.

On the other hand, when working with self-signed certificates, while it is possible to create a certificate chain exceeding a length of 2, this chain will not be validated by the SSL library in HTTPClient for security reasons (to avoid keytool being used to orchestrate certificate-related attacks on the TLS protocol).

3 Setting up TLS for mutual (client) authentication

Before proceeding, we will modify `SSLClientDemo` back to its original implementation in order to make REST API calls successfully to `SSLServerDemo`

In the package `com.workshop.security` in `src/main/java`, make the following changes

`MyRunner (switch back to the original version)`

Delete `RemoteRestService`

Make the following changes to `application.properties`

```
myrest.url=https://localhost:8443/api/developers
spring.main.web-application-type=none

trust.store.location=classpath:client-truststore.p12
trust.store.password=changeit
```

Add this property back to the VM arguments in the Run configuration for `SSLClientDemo`

```
-Djdk.security.allowNonCaAnchor=true
```

If you wish, you can also remove the other properties that we placed in there earlier in order to log a complete trail of handshake messages in the setup of TLS

Start up `SSLServerDemo`

Run `SSLClientDemo` and verify that it runs successfully as we had demonstrated previously before.

3.1 Server setup for mutual authentication

The process involved here is simply extending the work we did earlier by generating a certificate for the client to authenticate itself and then placing this certificate in the client's key store as well as the server's trust store.

At the command prompt shell in any directory, create a new key store for the client with a single self-signed certificate:

```
keytool -genkeypair -alias client-keypair -keyalg RSA -keysize 2048 -
sigalg SHA256withRSA -dname "cn=dcuniverse.com,ou=DC comics,o=DC
inc,l=San Francisco,ST=California,C=US" -validity 365 -ext
KU=digitalSignature,keyCertSign -ext
EKU=clientAuth,serverAuth,anyExtendedKeyUsage -storetype PKCS12 -
keystore client-keystore.p12 -storepass changeit
```

Here we add in additional extension fields for the Key Usage in case we intend to use this certificate to sign another certificate in the way that we did for the server self-signed certificate earlier. Most importantly however, the Extended Key Usage must specify clientAuth so that we can use this certificate specifically for client authentication.

Next, we export this certificate and create a new trust store for the server by importing in this certificate:

```
keytool -exportcert -rfc -alias client-keypair -file client.cer -
keystore client-keystore.p12 -storepass changeit -storetype PKCS12

keytool -importcert -trustcacerts -alias dc -file client.cer -
storetype PKCS12 -keystore server-truststore.p12 -storepass changeit
```

Finally, we verify the contents of the newly created trust store.

```
keytool -list -storetype PKCS12 -keystore server-truststore.p12 -
storepass changeit -v
```

Place `server-truststore.p12` in `src/main/resources` for `SSLServerDemo`

Make the following modifications to `application.properties`

```
server.port=8443
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:server-keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-alias=ironman

server.ssl.client-auth=need

server.ssl.trust-store-type=PKCS12
server.ssl.trust-store=classpath:server-truststore.p12
server.ssl.trust-store-password=changeit
```

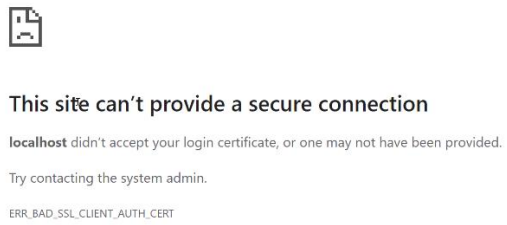
Start up `SSLServerDemo` as usual from the Boot Dashboard

3.2 Browser setup for mutual authentication

Using a browser, connect to the app's deployment port using HTTPS

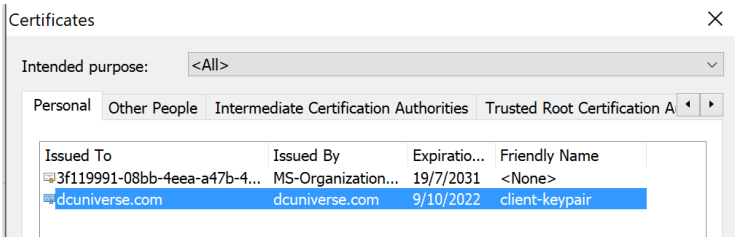
https://localhost:8443/api/developers

You should obtain an error message similar to the one below as the browser has not yet been configured to return a client certificate that the server expects for mutual authentication.



In order to get pass this error, you will need to install a client certificate for the browser to use in the Personal section of the browser / OS Certificates list.

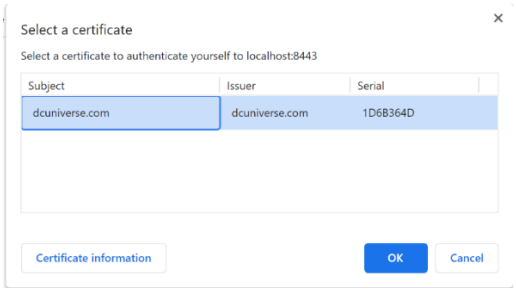
For e.g. in Chrome, go to
Settings -> Privacy and Security -> Security (Safe Browsing) -> Manage Certificates
In the Personal tab, select Import. Then select the PKCS12 keystore containing the private / public key pair for the client certificate you wish to use (in this example, it is: `client-keystore.p12`).
You will be prompted for the key store password (`changeit` in this example). Make sure you place the certificate in the Personal Certificate store section.
When you are done, you should be able to see the imported certificate in the Personal section.



Close and reopen the browser and in a new tab, go to:

https://localhost:8443/api/developers

You should now be prompted on a certificate to submit to the server.



Select the dcuniverse.com certificate and click on. This submits the installed client certificate for authentication to `SSLServerDemo` who authenticates using the same certificate that it has in its trust store (`server-truststore.p12`).

You should now receive back the expected results from the REST API service.

3.3 RestTemplate client setup for mutual authentication

Lets run `SSLClientDemo` without making any modifications.

As expected, the application fails with an exception regarding a failed handshake and a bad certificate. This is because we still have not configured it to submit a client certificate for authentication to the REST service.

Place `client-keystore.p12` in `src/main/resources` of `SSLClientDemo`

The `application.properties` can be further extended to use this key store as shown below:

```
myrest.url=https://localhost:8443/api/developers
spring.main.web-application-type=none

trust.store.location=classpath:client-truststore.p12
trust.store.password=changeit

key.store.location=classpath:client-keystore.p12
key.store.password=changeit
```

In the package `com.workshop.security` in `src/main/java`, make the following changes

`MainConfig-v2`

Run `SSLClientDemo`. This time it should succeed and produce the expected results.

4 Tidying up

As general good security practice, you should remove all self-signed / dummy certificates that you have placed in the current user / computer certificate stores after you have finished working with them. This is particularly important if you are working on a shared machine that will be used by other developers or users for other purposes.