Everything You Ever Wanted to Know About SSL (but Were Afraid to Ask)

> Or perhaps more accurately, "practical things I've learned about SSL". This post (and the companion Spring Boot application) will demonstrate using SSL certificates to validate and authenticate connections to secure endpoints over HTTPS for some common use cases (web servers, browser authentication, unit and integration testing). It shows how to configure Apache HTTP server for two-way SSL, unit testing SSL authentication with Apache's `HttpClient` and `HttpServer` (Java), and integration testing a REST API within a Spring Boot application running on an embedded Tomcat container.

There are lots of ways for a client to authenticate itself against a server, including basic authentication, form-based authentication, and OAuth.

To prevent exposing user credentials over the wire, the client communicates with the server over HTTPS, and the server's identify is confirmed by validating its SSL certificate. The server doesn't necessarily care who the client is, just as long as they have the correct credentials.

An even higher level of security can be gained with using SSL certificates for both the client and the server.

Two-way SSL authentication (also known as "mutual authentication", and "TLS/SSL with client certificates") refers to two parties authenticating each other through verifying provided digital certificates, so that both parties are assured of the other's identity.

Terminology

TLS vs SSL

TLS is the successor to SSL. It is a protocol that ensures privacy between communicating applications. Unless otherwise stated, in this document consider TLS and SSL as interchangable.

Certificate (cert)

The public half of a public/private key pair with some additional metadata about who issued it etc. It may be freely given to anyone.

Private Key

A private key can verify that its corresponding certificate/public key was used to encrypt data. It is never given out publicly.

Certificate Authority (CA)

A company that issues digital certificates. For SSL/TLS certificates, there are a small number of providers (e.g. Symantec/Versign/Thawte, Comodo, GoDaddy, LetsEncrypt) whose certificates are included by most browsers and Operating Systems. They serve the purpose of a "trusted third party".

Certificate Signing Request (CSR)

A file generated with a private key. A CSR can be sent to a CA to request to be signed. The CA uses its private key to digitally sign the CSR and create a signed cert. Browsers can then use the CA's cert to validate the new cert has been approved by the CA.

X.509

A specification governing the format and usage of certificates.

Authentication with SSL

SSL is the standard security technology for establishing an encrypted link between a web server and a browser. Normally when a browser (the client) establishes an SSL connection to a secure web site, only the server certificate is checked. The browser either relies on itself or the operating system providing a list of certs that have been designated as root certificates and to be trusted as CAs.

One-way SSL authentication (server -> client)

Client and server use 9 handshake messages to establish the encrypted channel prior to message exchanging:

1. Client sends `ClientHello` message proposing SSL options.
2. Server responds with `ServerHello` message selecting the SSL options.
3. Server sends `Certificate` message, which contains the server's certificate.
4. Server concludes its part of the negotiation with `ServerHelloDone` message.
5. Client sends session key information (encrypted with server's public key) in `ClientKeyExchange` message.
6. Client sends `ChangeCipherSpec` message to activate the negotiated options for all future messages it will send.
7. Client sends `Finished` message to let the server check the newly activated options.
8. Server sends `ChangeCipherSpec` message to activate the negotiated options for all future messages it will send.
9. Server sends `Finished` message to let the client check the newly activated options.

Two-way SSL authentication (server <-> client)

Client and server use 12 handshake messages to establish the encrypted channel prior to message exchanging:

1. Client sends `ClientHello` message proposing SSL options.
2. Server responds with `ServerHello` message selecting the SSL options.
3. Server sends `Certificate` message, which contains the server's certificate.
4. Server requests client's certificate in `CertificateRequest` message, so that the connection can be mutually authenticated.
5. Server concludes its part of the negotiation with `ServerHelloDone` message.
6. Client responds with `Certificate` message, which contains the client's certificate.
7. Client sends session key information (encrypted with server's public key) in `ClientKeyExchange` message.
8. Client sends a `CertificateVerify` message to let the server know it owns the sent certificate.

9. Client sends `ChangeCipherSpec` message to activate the negotiated options for all future messages it will send.
10. Client sends `Finished` message to let the server check the newly activated options.
11. Server sends `ChangeCipherSpec` message to activate the negotiated options for all future messages it will send.
12. Server sends `Finished` message to let the client check the newly activated options.

File Formats for Certs and Keys

Privacy-Enhanced Mail (PEM)

PEM is just Distinguished Encoding Rules (DER) that has been Base64 encoded. Used for keys and certificates.

PKCS12

PKCS12 is a password-protected format that can contain multiple certificates and keys.

Java KeyStore (JKS)

Java version of PKCS12 and also password protected. Entries in a JKS file must have an "alias" that is unique. If an alias is not specified, "mykey" is used by default. It's like a database for certs and keys.

Tools

OpenSSL

An open source toolkit implementing the SSL (v2/v3) and TLS (v1) protocols, as well as a full-strength general purpose cryptography library.

Keytool

Manages a Java KeyStore of cryptographic keys, X.509 certificate chains, and trusted certificates. Ships with the JDK.

XCA

A graphical tool to create and manage certificates.

PKI and the SSL Certificate Chain ("the Chain of Trust")

All SSL/TLS connections rely on a chain of trust called the SSL Certificate Chain. Part of PKI (Public Key Infrastructure), this chain of trust is established by certificate authorities (CAs) who serve as trust anchors that verify the validity of the systems being communicated with. Each client (browser, OS, etc.) ships with a list of trusted CAs.

CA-signed Certificates

Chain of Trust

In the above example, the wildcard certificate for "`*.elastic.snaplogic.com`" has been issued by the "Go Daddy Secure Certificate Authority - G2" intermediate CA, which in turn was issued by the "Go Daddy Root Certificate Authority - G2" root CA.

> Many organizations will create their own internal, self-signed root CA to be used to sign certificates for PKI use within that organization. Then, if each system trusts that CA, the certificates that are issued and signed by that CA will be trusted too.

To trust a system that presents a the above certificate at a particular domain (e.g. `https://elastic.snaplogic.com`), the client system must trust both the intermediate CA and the root CA (the public certs of those CAs must exist in the client system's trust/CA store), as well as verifying the chain is valid (signatures

match, domain names match, and other requirements of the X.509 standard).

Once a client trusts the intermediate and root CAs, all valid certificates signed by those CAs will be trusted by the client.

> If only particular certificates signed by a trusted CA should be trusted, then either limiting the certificates in the CA store, or checking for certain certificate fingerprints, etc. should be considered instead.

Self-signed Certificates

Self-signed certificates have a chain length of 1 - they are not signed by a CA but by the certificate creator itself. All root certificates are self-signed (a chain has to start somewhere).

For example, to create a self-signed certificate (plus private key) for `localhost`, the following OpenSSL command may be used:

```
| root@SL-MBP-RHOWLETT.local:/private/etc/apache2/ssl
| => openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout localhost_self-signed.key -out localhost_self-signed.pem
Generating a 2048 bit RSA private key
.....................................................+++
.....................+++
writing new private key to 'localhost_self-signed.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Colorado
Locality Name (eg, city) []:Boulder
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SnapLogic
Organizational Unit Name (eg, section) []:SnapTeam
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:
```

> The `"Common Name"` must match the domain that will be presenting the certificate e.g. `localhost`

To create a `.p12` file (that can be imported and trusted within OS X's Keychain application for example):

```
| root@SL-MBP-RHOWLETT.local:/private/etc/apache2/ssl
| => openssl pkcs12 -export -in /etc/apache2/ssl/localhost_self-signed.pem -inkey /etc/apache2/ssl/localhost_self-signed.key -name "SelfSignedServer" -ou
Enter Export Password:
Verifying - Enter Export Password:
```

When you want to specifically control a small number of certificates to use within an internal network you control, self-signed certificates can be very useful.

Create a local SSL server with Apache

Modified from: https://gist.github.com/jonathantneal/774e4b0b3d4d739cbc53

Configuring Apache

Switch to root:

```
1 sudo su
```

Within Terminal, start Apache:

```
1 apachectl start
```

In a web browser, visit http://localhost. You should see a message stating that **It works!**

Configuring Apache for HTTP: Setting up a port 80 Virtual Host

Within Terminal, edit the Apache Configuration:

```
1 vi /etc/apache2/httpd.conf
```

Enable SSL by uncommenting line 143:

```
1 LoadModule ssl_module libexec/apache2/mod_ssl.so
```

Within your editor, replace line 212 to suppress messages about the server's fully qualified domain name:

```
1 ServerName localhost
```

Next, uncomment line 160 and line 499 to enable Virtual Hosts.

```
1 LoadModule vhost_alias_module libexec/apache2/mod_vhost_alias.so
2
3 Include /private/etc/apache2/extra/httpd-vhosts.conf
```

Uncomment line 518 to include httpd-ssl.conf (to listen on port 443):

```
1 Include /private/etc/apache2/extra/httpd-ssl.conf
```

Optionally, uncomment line 169 to enable PHP.

```
1 LoadModule php5_module libexec/apache2/libphp5.so
```

Within Terminal, edit the Virtual Hosts

```
1 vi /etc/apache2/extra/httpd-vhosts.conf
```

Within your editor, replace the entire contents of this file with the following, replacing `rhowlett` with your user name.

```
1  <VirtualHost *:80>
2      ServerName localhost
3      DocumentRoot "/Users/rhowlett/Sites/localhost"
4
5      <Directory "/Users/rhowlett/Sites/localhost">
6          Options Indexes FollowSymLinks
7          AllowOverride All
8          Order allow,deny
9          Allow from all
10         Require all granted
11     </Directory>
12 </VirtualHost>
```

Within Terminal, restart Apache:

```
1 apachectl restart
```

Configuring Apache: Creating a Site

Within **Terminal**, Create a **Sites** directory, which will be the parent directory of many individual Site subdirectories:

```
1 mkdir ~/Sites
```

Next, create a **localhost** subdirectory within **Sites**, which will be our first site:

```
1 mkdir ~/Sites/localhost
```

Finally, create an HTML document within **localhost**:

```
1 echo "<h1>localhost works</h1>" > ~/Sites/localhost/index.html
```

Now, in a web browser, visit http://localhost. You should see a message stating that localhost works.

Configuring SSL

Note: I used `snaplogic` for all passwords below

Modified from: http://www.stefanocapitanio.com/configuring-two-way-authentication-ssl-with-apache/

Within **Terminal**, create a SSL directory:

```
1 mkdir /etc/apache2/ssl
2
3 cd /etc/apache2/ssl
4 mkdir certs private
```

Create the CA cert

Create a database to keep track of each certificate signed:

```
1 echo '100001' > serial
2 touch certindex.txt
```

Make a custom config file for openssl to use:

```
1 vi openssl.cnf
```

```
#
# OpenSSL configuration file.
#

# Establish working directory.

dir = .

[ ca ]
default_ca = CA_default

[ CA_default ]
serial = $dir/serial
database = $dir/certindex.txt
new_certs_dir = $dir/certs
certificate = $dir/cacert.pem
private_key = $dir/private/cakey.pem
default_days = 365
default_md = sha512
preserve = no
email_in_dn = no
nameopt = default_ca
certopt = default_ca
policy = policy_match
```

```
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
default_bits = 2048 # Size of keys
default_keyfile = key.pem # name of generated keys
default_md = sha512 # message digest algorithm
string_mask = nombstr # permitted characters
distinguished_name = req_distinguished_name
req_extensions = v3_req

[ req_distinguished_name ]
countryName = Country Name (2 letter code)
countryName_default = US
countryName_min = 2
countryName_max = 2

stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default = Colorado

localityName = Locality Name (eg, city)
localityName_default = Boulder

0.organizationName = Organization Name (eg, company)
0.organizationName_default = SnapLogic

# we can do this but it is not needed normally :-)
#1.organizationName = Second Organization Name (eg, company)
#1.organizationName_default = World Wide Web Pty Ltd

organizationalUnitName = Organizational Unit Name (eg, section)
organizationalUnitName_default = SnapTeam

commonName = Common Name (eg, YOUR name)
commonName_max = 64
commonName_default = localhost

emailAddress = Email Address
emailAddress_max = 64

# SET-ex3 = SET extension number 3

[ req_attributes ]
challengePassword = A challenge password
challengePassword_min = 4
challengePassword_max = 20

unstructuredName = An optional company name

[ req_distinguished_name ]
```

```
countryName = Country Name (2 letter code)
countryName_default = US
countryName_min = 2
countryName_max = 2

stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default = Colorado

localityName = Locality Name (eg, city)

0.organizationName = Organization Name (eg, company)
0.organizationName_default = SnapLogic

# we can do this but it is not needed normally :-)
#1.organizationName = Second Organization Name (eg, company)
#1.organizationName_default = World Wide Web Pty Ltd

organizationalUnitName = Organizational Unit Name (eg, section)
organizationalUnitName_default = SnapTeam

commonName = Common Name (eg, YOUR name)
commonName_max = 64
commonName_default = localhost

emailAddress = Email Address
emailAddress_max = 64

# SET-ex3 = SET extension number 3

[ req_attributes ]
challengePassword = A challenge password
challengePassword_min = 4
challengePassword_max = 20

unstructuredName = An optional company name
[ v3_ca ]
basicConstraints = CA:TRUE
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer:always

[ v3_req ]
basicConstraints = CA:FALSE
subjectKeyIdentifier = hash
```

Note that I've set `default_md` to `sha512` so that modern browsers won't complain about Weak Signature Algorithms.

Create a CA by creating a root certificate. This will create the private key (`private/cakey.pem`) and the public key (`cacert.pem`, a.k.a. the certificate) of the root CA. Use the default **localhost** for the common name:

```
| root@SL-MBP-RHOWLETT.local:/private/etc/apache2/ssl
| => openssl req -new -x509 -extensions v3_ca -keyout private/cakey.pem -out cacert.pem -days 365 -config ./openssl.cnf
Generating a 2048 bit RSA private key
..............................+++
............................+++
writing new private key to 'private/cakey.pem'
```

```
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
phrase is too short, needs to be at least 4 chars
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Colorado]:
Locality Name (eg, city) [Milan]:Boulder
Organization Name (eg, company) [Organization default]:SnapLogic
Organizational Unit Name (eg, section) [SnapLogic]:SnapTeam
Common Name (eg, YOUR name) [localhost]:
Email Address []:
```

Create the Server cert

Create a key and signing request for the server. This will create the CSR for the server (server-req.pem) and the server's private key (private/server-key.pem). Use the default localhost for the common name:

```
| root@SL-MBP-RHOWLETT.local:/etc/apache2/ssl
| => openssl req -new -nodes -out server-req.pem -keyout private/server-key.pem -days 365 -config openssl.cnf
Generating a 2048 bit RSA private key
......+++
.................................+++
writing new private key to 'private/server-key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Colorado]:
Locality Name (eg, city) [Boulder]:
Organization Name (eg, company) [SnapLogic]:
Organizational Unit Name (eg, section) [SnapTeam]:
Common Name (eg, YOUR name) [localhost]:
Email Address []:
```

Have the CA sign the server's CSR. This will create the server's public certificate (`server-cert.pem`):

```
| root@SL-MBP-RHOWLETT.local:/etc/apache2/ssl
| => openssl ca -out server-cert.pem -days 365 -config openssl.cnf -infiles server-req.pem
Using configuration from openssl.cnf
Enter pass phrase for ./private/cakey.pem:
Check that the request matches the signature
```

```
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'US'
stateOrProvinceName   :PRINTABLE:'Colorado'
localityName          :PRINTABLE:'Boulder'
organizationName      :PRINTABLE:'SnapLogic'
organizationalUnitName:PRINTABLE:'SnapTeam'
commonName            :PRINTABLE:'localhost'
Certificate is to be certified until Oct  4 16:30:23 2016 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Create the Client cert

Each client will create a key and signing request. We will just create one for now. You must use a different common name than the server/CA - here I'm using `client`. This will create the client's CSR (`client-req.pem`) and the client's private key (`private/client-key.pem`):

```
| root@SL-MBP-RHOWLETT.local:/etc/apache2/ssl
| => openssl req -new -nodes -out client-req.pem -keyout private/client-key.pem -days 365 -config openssl.cnf
Generating a 2048 bit RSA private key
.....................................................+++
...........+++
writing new private key to 'private/client-key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Colorado]:
Locality Name (eg, city) [Boulder]:
Organization Name (eg, company) [SnapLogic]:
Organizational Unit Name (eg, section) [SnapTeam]:
Common Name (eg, YOUR name) [localhost]:client
Email Address []:
```

Have the CA sign the client's CSR. This will create the client's public certificate (`client-cert.pem`):

```
| root@SL-MBP-RHOWLETT.local:/etc/apache2/ssl
| => openssl ca -out client-cert.pem -days 365 -config openssl.cnf -infiles client-req.pem
Using configuration from openssl.cnf
Enter pass phrase for ./private/cakey.pem:
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'US'
stateOrProvinceName   :PRINTABLE:'Colorado'
localityName          :PRINTABLE:'Boulder'
```

```
organizationName      :PRINTABLE:'SnapLogic'
organizationalUnitName:PRINTABLE:'SnapTeam'
commonName            :PRINTABLE:'client'
Certificate is to be certified until Oct  4 16:40:01 2016 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Finally, create the PKCS12 file using the client's private key, the client's public cert and the CA cert. This will create the (Mac-friendly) PKCS12 file (`client-cert.p12`):

```
| root@SL-MBP-RHOWLETT.local:/etc/apache2/ssl
| => openssl pkcs12 -export -in client-cert.pem -inkey private/client-key.pem -certfile cacert.pem -name "Client" -out client-cert.p12
Enter Export Password:
Verifying - Enter Export Password:
```

Configuring Apache for HTTPS and one-way SSL auth

As root:

```
vi /etc/apache2/extra/httpd-vhosts.conf
```

Add a Virtual Host for port 443 and enable SSL:

```
<VirtualHost *:80>
    ServerName localhost
    DocumentRoot "/Users/rhowlett/Sites/localhost"

    <Directory "/Users/rhowlett/Sites/localhost">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
        Require all granted
    </Directory>
</VirtualHost>

<VirtualHost *:443>
    ServerName localhost
    DocumentRoot "/Users/rhowlett/Sites/localhost"

    SSLCipherSuite HIGH:MEDIUM:!aNULL:!MD5
    SSLEngine on
    SSLCertificateFile /etc/apache2/ssl/server-cert.pem
    SSLCertificateKeyFile /etc/apache2/ssl/private/server-key.pem

    <Directory "/Users/rhowlett/Sites/localhost">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
        Require all granted
    </Directory>
</VirtualHost>
```

Restart Apache:

```
apachectl restart
```

(Mac) Install the CA cert into Keychain Access

Open `/etc/apache2/ssl` in Finder:

Finder

Open the CA cert (`cacert.pem`) by double-clicking it to install it to Keychain Access:

Install CA cert

Mark it as trusted:

Trust CA cert Trusted Trusted

Open your browser to https://localhost and you should see a successful secure connection:

Successful HTTPS

Configuring Apache for two-way SSL auth

As root:

```
vi /etc/apache2/extra/httpd-vhosts.conf
```

Add the `SSLVerifyClient`, `SSLCertificateFile`, and `SSLCACertificateFile` options:

```
<VirtualHost *:80>
    ServerName localhost
    DocumentRoot "/Users/rhowlett/Sites/localhost"

    <Directory "/Users/rhowlett/Sites/localhost">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
        Require all granted
    </Directory>
</VirtualHost>

<VirtualHost *:443>
    ServerName localhost
    DocumentRoot "/Users/rhowlett/Sites/localhost"

    SSLCipherSuite HIGH:MEDIUM:!aNULL:!MD5
    SSLEngine on
    SSLCertificateFile /etc/apache2/ssl/server-cert.pem
    SSLCertificateKeyFile /etc/apache2/ssl/private/server-key.pem

    SSLVerifyClient require
    SSLVerifyDepth 10
    SSLCACertificateFile /etc/apache2/ssl/cacert.pem
```

```
    <Directory "/Users/rhowlett/Sites/localhost">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
        Require all granted
    </Directory>
</VirtualHost>
```

Restart Apache:

```
apachectl restart
```

OpenSSL can now confirm that the two-way SSL handshake can be successfully completed:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/Downloads
| => openssl s_client -connect localhost:443 -tls1 -cert /etc/apache2/ssl/client-cert.pem -key /etc/apache2/ssl/private/client-key.pem
CONNECTED(00000003)
depth=1 /C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
verify error:num=19:self signed certificate in certificate chain
verify return:0
---
Certificate chain
 0 s:/C=US/ST=Colorado/O=SnapLogic/OU=SnapTeam/CN=localhost
   i:/C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
 1 s:/C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
   i:/C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDPjCCAiYCAxAAATANBgkqhkiG9w0BAQ0FADBtMQswCQYDVQQGEwJVUzERMA8G
A1UECBMIQ29sb3JhZG8xEDAOBgNVBAcTB0JvdWxkZXIxEjAQBgNVBAoTCVNuYXBM
b2dpYzERMA8GA1UECxMIU25hcFRlYW0xEjAQBgNVBAMTCWxvY2FsaG9zdDAeFw0x
NTEwMDYxOTE1MTNaFw0xNjEwMDUxOTE1MTNaMFsxCzAJBgNVBAYTAlVTMREwDwYD
VQQIEwhDb2xvcmFkbzESMBAGA1UEChMJU25hcExvZ2ljMREwDwYDVQQLEwhTbmFw
VGVhbTESMBAGA1UEAxMJbG9jYWxob3N0MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAyvia0x0Nd4tYyvoXEYtI3s/eLIQ3wFsOJIibNy70Plhp35gScQ69
MiIrVDYqIydVbInzyY5kuhttrUIrHCIiDwa5OqEiExJ+ollY9icnrMLrEXJqvv5C
/fduS5byC6StNg7xHQkYlYLUYMw8QQyCZFQVGXlxZeG6i086ffMYduFimkBAkNj5
/LkIwrOELpGnNcrOJxQEnLi8vmRI3oiCrgVc0ugrFBnoj3Tf6y3lx23fYgLbqf9c
bRCS6V3eppa/x9sezv9KQ+pDYly0bwKIcvJ9xLp7qPiO+smGGvS97Ec4NAif8y6v
pU92cPH32cv1p0AIDF0+GMOgVyAYZgSKQwIDAQABMA0GCSqGSIb3DQEBDQUAA4IB
AQBedsAvkB1yNLE2GCJWWQ19qEKOIBYCRQc2z29PgF/LAz5GVOIw/ZiN37C2vTob
jk1NnqfOx5aipQ5Pe5D2yfbarDl0kaqRn9MhBySi+oi3AgUZ5yL0x/nGF9O8jszJ
OM1FUC6qXKic5pR0qTrdXigONlKb0Au+l3z5dFMiqnNmDrNlI8kW1OXrwy/jvyPv
H1bHWAKYFTvHi2v7A0B96V1VvFBLbuQztckPQ3VpFDOwWhWLr2D90vxFd1Ea0SCi
3bysz4ax9XP0bmXJY+968nV31qQJMkk5/3rE5PWVZibsniccfdujgSQYl+yNA3sB
F5h6mCR6pAONZFo6+U3zARSb
-----END CERTIFICATE-----
subject=/C=US/ST=Colorado/O=SnapLogic/OU=SnapTeam/CN=localhost
issuer=/C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
---
Acceptable client certificate CA names
/C=US/ST=Colorado/L=Boulder/O=SnapLogic/OU=SnapTeam/CN=localhost
---
SSL handshake has read 4004 bytes and written 1539 bytes
```

```
---
New, TLSv1/SSLv3, Cipher is DHE-RSA-AES256-SHA
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : TLSv1
    Cipher    : DHE-RSA-AES256-SHA
    Session-ID: A1D9CE5273963BCF70503B499D7714ECE2B628CEE59CE554615743ACEEA8E281
    Session-ID-ctx:
    Master-Key: 0920CEE1491E9A116B2DF959430890D449D49DA990A178C0AC980DD5AF359B7E1CDD4B2D8237C8F81BAE186BC06E7BB0
    Key-Arg   : None
    TLS session ticket:
    0000 - 5c 8f 01 d5 5d c1 62 d5-65 d7 8f 05 5f 47 d2 82   \...].b.e..._G..
    0010 - f0 fd 2c 88 be 58 25 6c-9e 9a 1e 78 6a b4 66 c4   ..,..X%l...xj.f.
    0020 - 0d 3d 31 04 97 a2 5f e7-6f 3c 9f c9 b1 44 6a ab   .=1..._.o<...Dj.
    0030 - 84 89 76 e4 63 9b 81 b7-c3 28 e0 95 c6 c3 f5 89   ..v.c....(......
    0040 - d5 f9 7f da df fb 12 f7-de 2a ec e8 c2 01 59 07   .........*....Y.
    0050 - 9e ad 91 56 91 34 88 73-66 d1 ea c1 72 dc 56 ee   ...V.4.sf...r.V.
    0060 - ee 61 fe 5e 38 f0 aa d6-3a 7d ad ef e6 be 2a 15   .a.^8...:}....*.
    0070 - dc cc 9f 04 5e e8 f9 2b-07 21 6b 0f da 9f 08 2e   ....^..+.!k.....
    0080 - 88 af 96 41 98 f3 ff 8a-01 66 1a 1d 61 47 1b e5   ...A.....f..aG..
    0090 - ec ab b7 af 79 aa 7d 25-ca e0 fa f4 2b 2e 9a dd   ....y.}%....+...
    00a0 - 95 0c 4b 35 d8 96 8b f0-1e 20 c1 c3 47 fc 65 ed   ..K5..... ..G.e.
    00b0 - 21 e4 50 59 1e 33 6a 5c-c6 27 f1 65 be 5b 0f 35   !.PY.3j\.'.e.[.5
    00c0 - 1d ba ac bf f5 9c d9 b7-32 87 11 ae b7 87 9b 52   ........2......R
    00d0 - bb 00 6b 66 af e2 94 45-e3 8f fb e0 b4 c6 d7 5a   ..kf...E.......Z
    00e0 - f8 1d 7a af e3 ee bb 6b-93 ff 46 af ed 86 bc f8   ..z....k..F.....
    00f0 - 6d e2 c9 60 eb 61 8e b9-7e bd 4d bb 1e 01 95 d2   m..`.a..~.M.....
    0100 - f5 d5 ee 82 10 4a 1d 23-9e 94 d7 0b 46 e4 d4 32   .....J.#....F..2
    0110 - 11 92 76 4a 94 9e a3 61-21 9b 4c 49 6c df 7b 18   ..vJ...a!.LIl.{.
    0120 - b7 49 66 bd 48 0d eb 9a-ad e9 32 c7 b9 6d 70 1a   .If.H.....2..mp.
    0130 - c7 a1 25 21 b4 f1 03 5b-80 83 e9 da 8d 56 f1 d9   ..%!...[.....V..
    0140 - 8b c5 32 b7 3a 67 5b 9c-51 84 a0 09 04 4f 48 60   ..2.:g[.Q....OH`
    0150 - 27 c0 fe 1c 45 7a 3b b2-22 8d ed 65 72 23 8a bf   '...Ez;."..er#..
    0160 - e3 09 eb 78 98 ec 08 06-9d 37 02 1a 4b ae cd 3a   ...x.....7..K..:
    0170 - 9c a4 bd 5d 47 5e d3 d7-7b 89 7b 97 78 a6 4c 10   ...]G^..{.{.x.L.
    0180 - bf 3e ed 1f f4 fe e5 97-90 ee 31 58 5f ff c6 c2   .>........1X_...
    0190 - 61 b7 df 0a f5 27 c6 a8-ac 61 a3 d0 1e 3a 6a 42   a....'...a...:jB
    01a0 - a9 18 b4 fb 4b 25 87 62-97 26 48 35 d0 16 d1 06   ....K%.b.&H5....
    01b0 - 9d 82 b5 e2 7b f2 24 c5-83 a1 4b fe 8d 38 ae 30   ....{.$...K..8.0
    01c0 - 8e eb e1 ac 8b 48 fa 27-b0 e1 ce b3 17 62 69 f0   .....H.'.....bi.
    01d0 - 30 17 ae 31 9d bf 77 64-66 5b 13 8e a2 63 2e 58   0..1..wdf[...c.X
    01e0 - 02 10 26 e1 3b 0d 55 fc-3d 0f d5 08 2d 1e 28 0a   ..&.;.U.=...-.(.
    01f0 - c2 fd a2 f3 2a 40 25 ed-2b 06 2c 92 c3 78 a3 b3   ....*@%.+.,..x..
    0200 - 35 bc d9 6c 57 97 ca 93-0f f3 b8 e4 60 d8 99 b4   5..lW.......`...
    0210 - b8 ba ae b7 47 4a 59 84-5b f9 5e b2 11 44 42 bd   ....GJY.[.^..DB.
    0220 - e8 46 3d 1d 09 70 72 f6-23 df 89 f8 f7 b7 84 d2   .F=..pr.#.......
    0230 - 7d 42 0e 5d d7 76 c2 da-0b 61 f9 48 3c c9 5f ba   }B.].v...a.H<._.
    0240 - ab be 5f 82 2b 03 07 f1-83 12 69 ee 56 b5 7e 06   .._.+.....i.V.~.
    0250 - 03 d7 8e b3 70 7c 93 75-3d cd e0 a1 1b 8a 14 ef   ....p|.u=.......
    0260 - 91 c6 74 14 1e 16 4c 46-07 c5 62 04 70 a7 fd 5d   ..t...LF..b.p..]
    0270 - e5 67 d8 bf 43 bb 5e f3-7c 37 db 1a 66 cb ad 7d   .g..C.^.|7..f..}
    0280 - cc 30 e4 9b 35 30 b5 6c-d0 4b ba b2 8b 01 71 0e   .0..50.l.K....q.
```

```
    0290 - 0a af ec 4e 6a 1a f8 6f-b7 5e 2b b9 e9 ec b6 b6   ...Nj..o.^+.....
    02a0 - 38 1c 70 5c 86 bf ae a4-e6 41 9d c9 9f 40 e4 a0   8.p\.....A...@..
    02b0 - 4b 0d 3d ab 01 90 da 55-cb b8 c8 e6 94 8d 76 35   K.=....U......v5
    02c0 - 94 b5 e2 1a 7c 69 5c b3-ee 08 8b bd 3f 97 c4 31   ....|i\.....?..1
    02d0 - 72 8a 30 a8 c6 3e 74 74-dc 47 c1 d0 ce bd 0b 19   r.0..>tt.G......
    02e0 - f4 93 55 8c 1f 02 b3 6e-f3 4d 44 f1 cc f0 ef 2d   ..U....n.MD....-
    02f0 - 4d 16 92 a3 15 fe 69 db-cc b1 b5 6b d0 4a 49 fc   M.....i....k.JI.
    0300 - 67 9e 0c 47 96 08 0e f2-b2 5c 06 24 45 f3 6a 7d   g..G.....\.$E.j}
    0310 - 6e 1b 2b 9a 68 23 11 3a-43 79 8c 77 9e 98 be 38   n.+.h#.:Cy.w...8
    0320 - 9a 0e e1 a5 17 bd 0f 7b-e0 ac ca 94 ac 48 68 5c   .......{.....Hh\
    0330 - f1 2b 98 b5 8d 36 b6 4f-aa 6f e7 d4 4d a3 f0 4c   .+...6.O.o..M..L
    0340 - cb 09 92 91 01 b9 c2 f1-49 24 64 d3 14 2f a3 5f   ........I$d../._
    0350 - 74 6f c0 54 16 73 c8 40-33 bc 7e e9 3b d8 d5 7c   to.T.s.@3.~.;..|
    0360 - 78 49 5c 80 83 88 4e 4b-46 f2 7a 6b 62 c4 ca 42   xI\...NKF.zkb..B
    0370 - 18 b6 22 40 77 fc 26 0e-28 50 89 7a 14 49 ba b0   .."@w.&.(P.z.I..
    0380 - 2c d7 26 7a 30 f9 9b 90-ba 9a 1f 3b 80 1b 0b 25   ,.&z0......;...%
    0390 - f0 e7 83 83 55 1f 1e f0-71 5b 64 a4 1e 76 91 bb   ....U...q[d..v..
    03a0 - d9 19 f5 2d 2e 54 d7 3a-93 95 29 ae 44 09 e6 cd   ...-.T.:..).D...
    03b0 - ec 79 8d b6 3c 09 d5 05-8d fc 2b 79 88 37 25 92   .y..<.....+y.7%.
    03c0 - 73 ae e6 8a d6 0c 1a eb-7b b9 08 44 4e 81 67 36   s.......{..DN.g6
    03d0 - a6 3a 57 43 d0 ed dc 3e-bb 0f 87 02 f5 fe 80 bb   .:WC...>........
    03e0 - 28 17 6e 7e ad c4 d9 4c-0a 53 fa 41 d2 d2 7c 76   (.n~...L.S.A..|v
    03f0 - a4 95 10 26 1d 5b 7d 19-23 dd 28 a0 48 c1 96 d9   ...&.[}.#.(.H...

    Start Time: 1444189099
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
---
closed
```

(Mac) Install the client PKCS12 file into Keychain Access

Open `/etc/apache2/ssl` in Finder and double-click the client PKCS12 file (`client-cert.p12`) to install it to Keychain Access:

Enter client cert password

Client cert added

Open https://localhost in your browser again and select the client cert when prompted:

Select client cert

You should then once again see a successful secure connection:

Successful two-way SSL

cURL will also work:

```
| => curl -v --cert /etc/apache2/ssl/client-cert.p12:snaplogic https://localhost
* Rebuilt URL to: https://localhost/
*   Trying ::1...
* Connected to localhost (::1) port 443 (#0)
* WARNING: SSL: Certificate type not set, assuming PKCS#12 format.
* Client certificate: client
```

curl prompt

```
| => curl -v --cert /etc/apache2/ssl/client-cert.p12:snaplogic https://localhost
* Rebuilt URL to: https://localhost/
*   Trying ::1...
* Connected to localhost (::1) port 443 (#0)
* WARNING: SSL: Certificate type not set, assuming PKCS#12 format.
* Client certificate: client
* TLS 1.0 connection using TLS_DHE_RSA_WITH_AES_256_CBC_SHA
* Server certificate: localhost
* Server certificate: localhost
> GET / HTTP/1.1
> Host: localhost
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Tue, 05 Jan 2016 23:16:29 GMT
< Server: Apache/2.4.16 (Unix) PHP/5.5.29 OpenSSL/0.9.8zg
< Last-Modified: Fri, 02 Oct 2015 18:34:41 GMT
< ETag: "19-521236aaf5240"
< Accept-Ranges: bytes
< Content-Length: 25
< Content-Type: text/html
<
<h1>localhost works</h1>
* Connection #0 to host localhost left intact
```

Unit Testing SSL Authentication with Apache's HttpClient and HttpServer

Apache's HttpComponents provides HttpClient, "an efficient, up-to-date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations."

It also provides HttpCore, which includes an embedded `HttpServer`, which can be used for unit testing.

Generate a PKCS12 (`.p12`) file from the public `server-cert.pem`, the private `server-key.pem`, and the CA cert `cacert.pem` created above to be used by the local test `HttpServer` instance:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => openssl pkcs12 -export -in /etc/apache2/ssl/server-cert.pem -inkey /etc/apache2/ssl/private/server-key.pem -certfile /etc/apache2/ssl/cacert.pem -name "Serve
Enter Export Password:
Verifying - Enter Export Password:
```

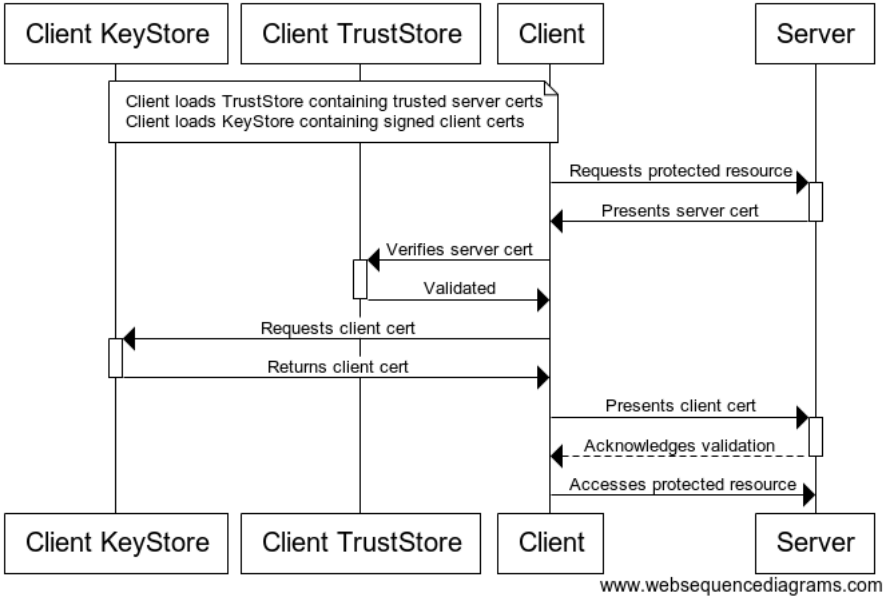
> If you see `unable to write 'random state'`, run `sudo rm ~/.rnd` and try again

Java KeyStores (JKS)

Java has its own version of PKCS12 called **Java KeyStore (JKS)**. It is also password protected. Entries in a JKS file must have an "alias" that is unique. If an alias is not specified, "mykey" is used by default. It's like a database for certs and keys.

For both the "KeyStore" and "TrustStore" fields in the REST SSL Account settings, we are going to use JKS files. The difference between them is for terminology reasons: KeyStores provide credentials, TrustStores verify credentials.

Clients will use certificates stored in their TrustStores to verify identities of servers. They will present certificates stored in their KeyStores to servers requiring them.



The JDK ships with a tool called **Keytool**. It manages a JKS of cryptographic keys, X.509 certificate chains, and trusted certificates.

Creating KeyStores and TrustStores with Keytool

Create the Server's KeyStore from the PKCS12 file:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -importkeystore -deststorepass snaplogic -destkeypass snaplogic -destkeystore server_keystore.jks -srckeystore server-cert.p12 -srcstoretype PKCS12
```

View the server keystore to confirm it now contains the server's cert:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -list -v -keystore server_keystore.jks
Enter keystore password:

Keystore type: JKS
```

```
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: server
Creation date: Jan 4, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: 100001
Valid from: Tue Oct 06 13:15:13 MDT 2015 until: Wed Oct 05 13:15:13 MDT 2016
Certificate fingerprints:
     MD5:  62:83:6B:84:1B:CB:DE:26:CA:E0:9D:E8:04:84:B6:C1
     SHA1: AD:D4:27:FF:9A:68:77:25:95:C3:A2:BE:F6:22:AD:82:5C:2B:AF:EB
     SHA256: 8D:8D:EA:E5:7C:7A:E9:42:C9:9E:71:2A:76:C7:BE:BE:34:CC:4A:CC:83:ED:FE:C8:8E:C6:06:D2:D8:89:59:4A
     Signature algorithm name: SHA512withRSA
     Version: 1
Certificate[2]:
Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: e4e00ed07233a969
Valid from: Tue Oct 06 13:14:51 MDT 2015 until: Wed Oct 05 13:14:51 MDT 2016
Certificate fingerprints:
     MD5:  F3:5E:28:E4:28:47:F2:EC:82:E2:BD:16:31:DC:90:02
     SHA1: 6F:0F:49:BA:A9:30:01:E9:4C:60:B3:A1:85:7D:BB:C6:79:1F:41:7B
     SHA256: A7:9D:25:E4:A6:34:8A:A3:5B:9A:CD:F3:62:D0:D8:2F:6A:A0:71:6A:6D:19:F3:04:A1:FD:BC:FB:21:40:DE:A1
     Signature algorithm name: SHA512withRSA
     Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
[CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US]
SerialNumber: [    e4e00ed0 7233a969]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
```

```
]


*****************************************
*****************************************
```

Create the client's truststore and import the server's public certificate:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -import -v -trustcacerts -keystore client_truststore.jks -storepass snaplogic -alias server -file /etc/apache2/ssl/server-cert.pem
Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: 100001
Valid from: Tue Oct 06 13:15:13 MDT 2015 until: Wed Oct 05 13:15:13 MDT 2016
Certificate fingerprints:
     MD5:  62:83:6B:84:1B:CB:DE:26:CA:E0:9D:E8:04:84:B6:C1
     SHA1: AD:D4:27:FF:9A:68:77:25:95:C3:A2:BE:F6:22:AD:82:5C:2B:AF:EB
     SHA256: 8D:8D:EA:E5:7C:7A:E9:42:C9:9E:71:2A:76:C7:BE:BE:34:CC:4A:CC:83:ED:FE:C8:8E:C6:06:D2:D8:89:59:4A
     Signature algorithm name: SHA512withRSA
     Version: 1
Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing client_truststore.jks]
```

View the client's truststore to confirm it contains the server's cert:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -list -v -keystore client_truststore.jks
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: server
Creation date: Jan 4, 2016
Entry type: trustedCertEntry

Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: 100001
Valid from: Tue Oct 06 13:15:13 MDT 2015 until: Wed Oct 05 13:15:13 MDT 2016
Certificate fingerprints:
     MD5:  62:83:6B:84:1B:CB:DE:26:CA:E0:9D:E8:04:84:B6:C1
     SHA1: AD:D4:27:FF:9A:68:77:25:95:C3:A2:BE:F6:22:AD:82:5C:2B:AF:EB
     SHA256: 8D:8D:EA:E5:7C:7A:E9:42:C9:9E:71:2A:76:C7:BE:BE:34:CC:4A:CC:83:ED:FE:C8:8E:C6:06:D2:D8:89:59:4A
     Signature algorithm name: SHA512withRSA
     Version: 1


*****************************************
*****************************************
```

One-Way SSL

At this point we have enough to demonstrate one-way SSL with the local test `HttpServer` instance. The `createLocalTestServer` method instantiates an embedded `HttpServer` instance with an (optional) `sslContext` (`null` meaning HTTP-only) and a `boolean` "forceSSLAuth" indicating if client certificates are required or not:

```
protected HttpServer createLocalTestServer(SSLContext sslContext, boolean forceSSLAuth)
        throws UnknownHostException {
    final HttpServer server = ServerBootstrap.bootstrap()
            .setLocalAddress(Inet4Address.getByName("localhost"))
            .setSslContext(sslContext)
            .setSslSetupHandler(socket -> socket.setNeedClientAuth(forceSSLAuth))
            .registerHandler("*",
                    (request, response, context) -> response.setStatusCode(HttpStatus.SC_OK))
            .create();

    return server;
}
```

The `getStore` method loads the JKS files from the classpath, and the `getKeyManagers` and `getTrustManagers` methods turn that store into the respective `Key-` or `TrustManager` arrays that are used to initialize an `SSLContext`:

```
private static final String JAVA_KEYSTORE = "jks";

/
 * KeyStores provide credentials, TrustStores verify credentials.

 * Server KeyStores stores the server's private keys, and certificates for corresponding public
 * keys. Used here for HTTPS connections over localhost.

 * Client TrustStores store servers' certificates.
 */
protected KeyStore getStore(final String storeFileName, final char[] password) throws
        KeyStoreException, IOException, CertificateException, NoSuchAlgorithmException {
    final KeyStore store = KeyStore.getInstance(JAVA_KEYSTORE);
    URL url = getClass().getClassLoader().getResource(storeFileName);
    InputStream inputStream = url.openStream();
    try {
        store.load(inputStream, password);
    } finally {
        inputStream.close();
    }

    return store;
}

/
 * KeyManagers decide which authentication credentials (e.g. certs) should be sent to the remote
 * host for authentication during the SSL handshake.

 * Server KeyManagers use their private keys during the key exchange algorithm and send
```

```
 * certificates corresponding to their public keys to the clients. The certificate comes from
 * the KeyStore.
 /
protected KeyManager[] getKeyManagers(KeyStore store, final char[] password) throws
        NoSuchAlgorithmException, UnrecoverableKeyException, KeyStoreException {
    KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(
            KeyManagerFactory.getDefaultAlgorithm());
    keyManagerFactory.init(store, password);

    return keyManagerFactory.getKeyManagers();
}

/
 * TrustManagers determine if the remote connection should be trusted or not.

 * Clients will use certificates stored in their TrustStores to verify identities of servers.
 * Servers will use certificates stored in their TrustStores to verify identities of clients.
 /
protected TrustManager[] getTrustManagers(KeyStore store) throws NoSuchAlgorithmException,
        KeyStoreException {
    TrustManagerFactory trustManagerFactory =
            TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
    trustManagerFactory.init(store);

    return trustManagerFactory.getTrustManagers();
}
```

The `SSLContext` is created and initialized like so:

```
/
Create an SSLContext for the server using the server's JKS. This instructs the server to
present its certificate when clients connect over HTTPS.
 /
protected SSLContext createServerSSLContext(final String storeFileName, final char[]
        password) throws CertificateException, NoSuchAlgorithmException, KeyStoreException,
        IOException, UnrecoverableKeyException, KeyManagementException {
    KeyStore serverKeyStore = getStore(storeFileName, password);
    KeyManager[] serverKeyManagers = getKeyManagers(serverKeyStore, password);
    TrustManager[] serverTrustManagers = getTrustManagers(serverKeyStore);

    SSLContext sslContext = SSLContexts.custom().useProtocol("TLS").build();
    sslContext.init(serverKeyManagers, serverTrustManagers, new SecureRandom());

    return sslContext;
}
```

The following unit test shows making a HTTPS request to the local test `HttpServer` instance and validating the server's public certificate with the client's truststore:

```
private static final boolean ONE_WAY_SSL = false; // no client certificates

private static final char[] KEYPASS_AND_STOREPASS_VALUE = "snaplogic".toCharArray();
```

```
4  private static final String SERVER_KEYSTORE = "ssl/server_keystore.jks";
5  private static final String CLIENT_TRUSTSTORE = "ssl/client_truststore.jks";
6
7  private CloseableHttpClient httpclient;
8
9  @Before
10 public void setUp() throws Exception {
11     httpclient = HttpClients.createDefault();
12 }
13
14 @Test
15 public void httpsRequest_With1WaySSLAndValidatingCertsAndClientTrustStore_Returns200OK()
16         throws Exception {
17     SSLContext serverSSLContext =
18             createServerSSLContext(SERVER_KEYSTORE, KEYPASS_AND_STOREPASS_VALUE);
19
20     final HttpServer server = createLocalTestServer(serverSSLContext, ONE_WAY_SSL);
21     server.start();
22
23     String baseUrl = getBaseUrl(server);
24
25     // The server certificate was imported into the client's TrustStore (using keytool -import)
26     KeyStore clientTrustStore = getStore(CLIENT_TRUSTSTORE, KEYPASS_AND_STOREPASS_VALUE);
27
28     SSLContext sslContext =
29             new SSLContextBuilder().loadTrustMaterial(
30                     clientTrustStore, new TrustSelfSignedStrategy()).build();
31
32     httpclient = HttpClients.custom().setSSLContext(sslContext).build();
33
34     /
35     The HTTP client will now validate the server's presented certificate using its TrustStore.
36      Since the cert was imported to the client's TrustStore explicitly (see above), the
37      certificate will validate and the request will succeed
38      /
39     try {
40         HttpResponse httpResponse = httpclient.execute(
41                 new HttpGet("https://&amp;quot; + baseUrl + "/echo/this"));
42
43         assertThat(httpResponse.getStatusLine().getStatusCode(), equalTo(200));
44     } finally {
45         server.stop();
46     }
47 }
48
49 protected String getBaseUrl(HttpServer server) {
50     return server.getInetAddress().getHostName() + ":" + server.getLocalPort();
51 }
```

The above unit test is included in the **everything-ssl** GitHub project, along with the following (which are useful to see the behavior when the SSL handshake fails, when server certificate validation is bypassed, malformed contexts etc.)

- `execute_WithNoScheme_ThrowsClientProtocolExceptionInvalidHostname`
- `httpRequest_Returns200OK`

- `httpsRequest_WithNoSSLContext_ThrowsSSLExceptionPlaintextConnection`
- `httpsRequest_With1WaySSLAndValidatingCertsButNoClientTrustStore_ThrowsSSLException`
- `httpsRequest_With1WaySSLAndTrustingAllCertsButNoClientTrustStore_Returns200OK`
- `httpsRequest_With1WaySSLAndValidatingCertsAndClientTrustStore_Returns200OK`

Two-Way SSL (Client Certificates)

To configure two-way SSL we have to create the server's truststore and create the client's keystore.

Since the client's certificate is signed by a CA we created ourselves, import the CA cert into the server truststore:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -import -v -trustcacerts -keystore server_truststore.jks -storepass snaplogic -file /etc/apache2/ssl/cacert.pem -alias cacert
Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: e4e00ed07233a969
Valid from: Tue Oct 06 15:14:51 EDT 2015 until: Wed Oct 05 15:14:51 EDT 2016
Certificate fingerprints:
     MD5:  F3:5E:28:E4:28:47:F2:EC:82:E2:BD:16:31:DC:90:02
     SHA1: 6F:0F:49:BA:A9:30:01:E9:4C:60:B3:A1:85:7D:BB:C6:79:1F:41:7B
     SHA256: A7:9D:25:E4:A6:34:8A:A3:5B:9A:CD:F3:62:D0:D8:2F:6A:A0:71:6A:6D:19:F3:04:A1:FD:BC:FB:21:40:DE:A1
     Signature algorithm name: SHA512withRSA
     Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
[CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US]
SerialNumber: [    e4e00ed0 7233a969]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing server_truststore.jks]
```

Viewing the server truststore will show the CA's certificate:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -list -v -keystore server_truststore.jks
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: cacert
Creation date: Jan 5, 2016
Entry type: trustedCertEntry

Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: e4e00ed07233a969
Valid from: Tue Oct 06 15:14:51 EDT 2015 until: Wed Oct 05 15:14:51 EDT 2016
Certificate fingerprints:
     MD5:  F3:5E:28:E4:28:47:F2:EC:82:E2:BD:16:31:DC:90:02
     SHA1: 6F:0F:49:BA:A9:30:01:E9:4C:60:B3:A1:85:7D:BB:C6:79:1F:41:7B
     SHA256: A7:9D:25:E4:A6:34:8A:A3:5B:9A:CD:F3:62:D0:D8:2F:6A:A0:71:6A:6D:19:F3:04:A1:FD:BC:FB:21:40:DE:A1
     Signature algorithm name: SHA512withRSA
     Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
[CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US]
SerialNumber: [    e4e00ed0 7233a969]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
]
```

```
*******************************************
*******************************************
```

Finally, the client keystore stores the client certificate that will presented to the server for SSL authentication. Import the cert from client's PKCS12 file (created above):

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -importkeystore -srckeystore /etc/apache2/ssl/client-cert.p12 -srcstoretype pkcs12 -destkeystore client_keystore.jks -deststoretype jks -deststorepas
Enter source keystore password:
Entry for alias client successfully imported.
Import command completed:  1 entries successfully imported, 0 entries failed or cancelled
```

Viewing the created `client_keystore.jks` file will show the `client` entry in the keystore:

```
| rhowlett@SL-MBP-RHOWLETT.local:~/dev/robinhowlett/github/everything-ssl/src/main/resources/ssl
| => keytool -list -v -keystore client_keystore.jks
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: client
Creation date: Jan 4, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=client, OU=SnapTeam, O=SnapLogic, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: 100002
Valid from: Tue Oct 06 13:15:41 MDT 2015 until: Wed Oct 05 13:15:41 MDT 2016
Certificate fingerprints:
     MD5:  F1:EF:60:64:48:DC:9B:C1:92:37:61:90:ED:48:01:1C
     SHA1: C5:4B:1C:EF:85:C1:8C:5A:AA:74:54:49:F0:B5:97:F1:EC:34:49:6F
     SHA256: B0:00:E4:C1:AE:03:92:95:9C:A2:BB:DB:13:3A:B6:38:BE:B4:BF:04:D0:72:41:6D:62:A6:93:D0:46:7E:3C:97
     Signature algorithm name: SHA512withRSA
     Version: 1
Certificate[2]:
Owner: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Issuer: CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US
Serial number: e4e00ed07233a969
Valid from: Tue Oct 06 13:14:51 MDT 2015 until: Wed Oct 05 13:14:51 MDT 2016
Certificate fingerprints:
     MD5:  F3:5E:28:E4:28:47:F2:EC:82:E2:BD:16:31:DC:90:02
     SHA1: 6F:0F:49:BA:A9:30:01:E9:4C:60:B3:A1:85:7D:BB:C6:79:1F:41:7B
     SHA256: A7:9D:25:E4:A6:34:8A:A3:5B:9A:CD:F3:62:D0:D8:2F:6A:A0:71:6A:6D:19:F3:04:A1:FD:BC:FB:21:40:DE:A1
     Signature algorithm name: SHA512withRSA
     Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
```

```
]
[CN=localhost, OU=SnapTeam, O=SnapLogic, L=Boulder, ST=Colorado, C=US]
SerialNumber: [    e4e00ed0 7233a969]
]

#2: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 03 09 12 6E 8B DD 7A 80   FB F5 21 AB 75 D9 B8 49  ...n..z...!.u..I
0010: 79 5B 61 1F                                        y[a.
]
]



*******************************************
*******************************************
```

> So, in summary, the server will present the certificate in its keystore to the client. The client will use its truststore to validate the server's certificate. The client will present its certificate in its keystore to the server, and the server will validate the client certificate's chain using the CA certificate in the server's truststore.

The `HttpServer` instance can now be created with the `forceSSLAuth` parameter set to `true` (see the `TWO_WAY_SSL` boolean) which will require client certificates. The client's `SSLContext` now has both the client truststore and keystore loaded:

```
1  private static final boolean TWO_WAY_SSL = true; // client certificates mandatory
2
3  @Test
4  public void httpsRequest_With2WaySSLAndHasValidKeyStoreAndTrustStore_Returns200OK()
5          throws Exception {
6      SSLContext serverSSLContext =
7              createServerSSLContext(SERVER_KEYSTORE, KEYPASS_AND_STOREPASS_VALUE);
8
9      final HttpServer server = createLocalTestServer(serverSSLContext, TWO_WAY_SSL);
10     server.start();
11
12     String baseUrl = getBaseUrl(server);
13
14     KeyStore clientTrustStore = getStore(CLIENT_TRUSTSTORE, KEYPASS_AND_STOREPASS_VALUE);
15     KeyStore clientKeyStore = getStore(CLIENT_KEYSTORE, KEYPASS_AND_STOREPASS_VALUE);
16
17     SSLContext sslContext =
18             new SSLContextBuilder()
19                     .loadTrustMaterial(clientTrustStore, new TrustSelfSignedStrategy())
20                     .loadKeyMaterial(clientKeyStore, KEYPASS_AND_STOREPASS_VALUE)
21                     .build();
22
23     httpclient = HttpClients.custom().setSSLContext(sslContext).build();
```

```
24
25    try {
26        CloseableHttpResponse httpResponse = httpclient.execute(
27                new HttpGet("https://&amp;quot; + baseUrl + "/echo/this"));
28
29        assertThat(httpResponse.getStatusLine().getStatusCode(), equalTo(200));
30    } finally {
31        server.stop();
32    }
33 }
```

Once again, the above unit test is included in the **everything-ssl** GitHub project, along with the following:

- `httpsRequest_With2WaySSLAndUnknownClientCert_ThrowsSSLExceptionBadCertificate`
- `httpsRequest_With2WaySSLButNoClientKeyStore_ThrowsSSLExceptionBadCertificate`

Two-Way SSL Authentication with Spring Boot, embedded Tomcat and RestTemplate

Spring Boot "takes an opinionated view of building production-ready Spring (Java) applications". Spring Boot's Starter POMs and auto-configuration make it quite easy to get going:

pom.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0&amp;quot; xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance&amp;quot;
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd&amp;quot;>
4    <modelVersion>4.0.0</modelVersion>
5
6      <parent>
7          <groupId>org.springframework.boot</groupId>
8          <artifactId>spring-boot-starter-parent</artifactId>
9          <version>1.3.1.RELEASE</version>
10         <relativePath/> <!– lookup parent from repository –>
11     </parent>
12
13   <groupId>com.robinhowlett</groupId>
14   <artifactId>everything-ssl</artifactId>
15   <version>0.0.1-SNAPSHOT</version>
16   <packaging>jar</packaging>
17
18   <name>everything-ssl</name>
19   <description>Spring Boot and SSL</description>
20
21   <properties>
22       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23       <java.version>1.8</java.version>
24
25       <commons-io-version>2.4</commons-io-version>
26   </properties>
27
28   <dependencies>
29       <dependency>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

      <dependency>
          <groupId>org.apache.httpcomponents</groupId>
          <artifactId>httpclient</artifactId>
      </dependency>
      <dependency>
          <groupId>commons-io</groupId>
          <artifactId>commons-io</artifactId>
          <version>${commons-io-version}</version>
      </dependency>


    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

  <build>
      <plugins>
          <plugin>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-failsafe-plugin</artifactId>
              <version>2.19</version>
              <executions>
                  <execution>
                      <goals>
                          <goal>integration-test</goal>
                          <goal>verify</goal>
                      </goals>
                  </execution>
              </executions>
          </plugin>
      </plugins>
  </build>

</project>
```

The Spring Boot documentation describes the properties required to configure SSL. I wanted however to support both HTTP and HTTPS for testing purposes, so an explicit SSL Connector for the embedded Tomcat container needed to be created:

Config.java

```
1  /*
2   * Configure embedded Tomcat and SSL connectors
3   /
4  @Configuration
5  public class Config {
6
7      @Autowired
8      private Environment env;
9
10     // Embedded Tomcat with HTTP and HTTPS support
11     @Bean
12     public EmbeddedServletContainerFactory servletContainer() {
13         TomcatEmbeddedServletContainerFactory tomcat = new
14                 TomcatEmbeddedServletContainerFactory();
15         tomcat.addAdditionalTomcatConnectors(createSSLConnector());
16         return tomcat;
17     }
18
19     // Creates an SSL connector, sets two-way SSL, key- and trust stores, passwords, ports etc.
20     protected Connector createSSLConnector() {
21         Connector connector = new Connector(Http11Protocol.class.getCanonicalName());
22         Http11Protocol protocol = (Http11Protocol) connector.getProtocolHandler();
23
24         File keyStore = null;
25         File trustStore = null;
26
27         try {
28             keyStore = getKeyStoreFile();
29         } catch (IOException e) {
30             throw new IllegalStateException("Cannot access keyStore: [" + keyStore + "] or " +
31                     "trustStore: [" + trustStore + "]", e);
32         }
33
34         trustStore = keyStore;
35
36         connector.setPort(env.getRequiredProperty("ssl.port", Integer.class));
37         connector.setScheme(env.getRequiredProperty("ssl.scheme"));
38         connector.setSecure(env.getRequiredProperty("ssl.secure", Boolean.class));
39
40         protocol.setClientAuth(env.getRequiredProperty("ssl.client-auth"));
41         protocol.setSSLEnabled(env.getRequiredProperty("ssl.enabled", Boolean.class));
42
43         protocol.setKeyPass(env.getRequiredProperty("ssl.key-password"));
44         protocol.setKeystoreFile(keyStore.getAbsolutePath());
45         protocol.setKeystorePass(env.getRequiredProperty("ssl.store-password"));
46         protocol.setTruststoreFile(trustStore.getAbsolutePath());
47         protocol.setTruststorePass(env.getRequiredProperty("ssl.store-password"));
48         protocol.setCiphers(env.getRequiredProperty("ssl.ciphers"));
49
50         return connector;
51     }
52
53     // support loading the JKS from the classpath (to get around Tomcat limitation)
54     private File getKeyStoreFile() throws IOException {
55         ClassPathResource resource = new ClassPathResource(env.getRequiredProperty("ssl.store"));
```

```
56
57          // Tomcat won't allow reading File from classpath so read as InputStream into temp File
58          File jks = File.createTempFile("server_keystore", ".jks");
59          InputStream inputStream = resource.getInputStream();
60          try {
61              FileUtils.copyInputStreamToFile(inputStream, jks);
62          } finally {
63              IOUtils.closeQuietly(inputStream);
64          }
65
66          return jks;
67      }
68  }
```

The `application.properties` file then defines the required SSL properties:

```
ssl.port=8443
ssl.scheme=https
ssl.secure=true
ssl.client-auth=true
ssl.enabled=true
ssl.key-password=snaplogic
ssl.store=ssl/server_keystore.jks
ssl.store-password=snaplogic
ssl.ciphers=TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_S
```

The `ssl.client-auth=true` property enforces two-way SSL.

> We are re-using the JKS files created earlier (above)

A simple REST interface is defined to return JSON representations of Greeting instances:

> GreetingController.java

```
1  /*
2   * Just says hello
3   /
4  @RestController
5  public class GreetingController {
6
7      private static final String template = "Hello, %s!";
8
9      @RequestMapping("/greeting")
10     public Greeting greet(
11             @RequestParam(value = "name", required = false, defaultValue = "World!") String name) {
12         return new Greeting(String.format(template, name));
13     }
14
15 }
```

Spring Security auto-configuration will enable basic authentication on the REST endpoint by default, so let's switch it off:

> HttpSecurityConfig.java

```
/*
 * Disable default-enabled basic auth
 /
@EnableWebSecurity
public class HttpSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic().disable();
    }

}
```

Integration Testing SSL Authentication with Spring's TestRestTemplate

Spring Boot really is great - it was quite straightforward to write an integration test to demonstrate two-way SSL authentication with the application running on the embedded Tomcat container.

First I created an `integration-test.properties` file to set the HTTPS port to be different than then main application (the HTTP port will be set to a random open port by an annotation on the test itself). The only contents of this file is the `ssl.port` property. All the other properties will come from the `application.properties` file detailed above:

```
ssl.port=54321
```

The integration test class has annotations that instruct it to run using the `SpringJUnit4ClassRunner`, the scan for `@Configuration` classes at the base package of the `EverythingSSLApplication` class, to choose a random available HTTP port with the `@WebIntegrationTest` annotation (which itself is a combination of the `@IntegrationTest` and `@WebAppConfiguration` annotations), and finally the `integration-test.properties` file is denoted as a `@TestPropertySource`.

ITEverythingSSL.java

```
/*
 * Integration test that uses the embedded Tomcat instance configured by this Spring Boot app
 /
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = EverythingSSLApplication.class)
@WebIntegrationTest(randomPort = true)
@TestPropertySource(locations = "classpath:integration-test.properties")
public class ITEverythingSSL {

    public static final String CLIENT_TRUSTSTORE = "ssl/client_truststore.jks";
    public static final String CLIENT_KEYSTORE = "ssl/client_keystore.jks";

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Value("${local.server.port}")
    private int port = 0;

    @Value("${ssl.port}")
    private int sslPort = 0;

    @Value("${ssl.store-password}")
    private String storePassword;
```

```
24
25     …
```

The first test confirms that plain HTTP is supported:

```
1  @Test
2  public void rest_OverPlainHttp_GetsExpectedResponse() throws Exception {
3      Greeting expected = new Greeting("Hello, Robin!");
4
5      RestTemplate template = new TestRestTemplate();
6
7      ResponseEntity<Greeting> responseEntity =
8              template.getForEntity("http://localhost:&amp;quot; + port + "/greeting?name={name}",
9                      Greeting.class, "Robin");
10
11     assertThat(responseEntity.getBody().getContent(), equalTo(expected.getContent()));
12 }
```

`TestRestTemplate` is "a convenience subclass of Spring's `RestTemplate` that is useful in integration tests"

The `getRestTemplateForHTTPS` method creates a `TestRestTemplate` instance with an `SSLContext` set to support the client's keystore and truststore:

```
1  private RestTemplate getRestTemplateForHTTPS(SSLContext sslContext) {
2      SSLConnectionSocketFactory connectionFactory = new SSLConnectionSocketFactory(sslContext,
3              new DefaultHostnameVerifier());
4
5      CloseableHttpClient closeableHttpClient =
6              HttpClientBuilder.create().setSSLSocketFactory(connectionFactory).build();
7
8      RestTemplate template = new TestRestTemplate();
9      HttpComponentsClientHttpRequestFactory httpRequestFactory =
10             (HttpComponentsClientHttpRequestFactory) template.getRequestFactory();
11     httpRequestFactory.setHttpClient(closeableHttpClient);
12     return template;
13 }
```

The test that then demonstrates two-way SSL is quite simple:

```
1  @Test
2  public void rest_WithTwoWaySSL_AuthenticatesAndGetsExpectedResponse() throws Exception {
3      Greeting expected = new Greeting("Hello, Robin!");
4
5      SSLContext sslContext = SSLContexts.custom()
6              .loadKeyMaterial(
7                      getStore(CLIENT_KEYSTORE, storePassword.toCharArray()),
8                      storePassword.toCharArray())
9              .loadTrustMaterial(
10                     getStore(CLIENT_TRUSTSTORE, storePassword.toCharArray()),
11                     new TrustSelfSignedStrategy())
```

```
12            .useProtocol("TLS").build();
13
14    RestTemplate template = getRestTemplateForHTTPS(sslContext);
15
16    ResponseEntity<Greeting> responseEntity =
17            template.getForEntity("https://localhost:&amp;quot; + sslPort + "/greeting?name={name}",
18                    Greeting.class, "Robin");
19
20    assertThat(responseEntity.getBody().getContent(), equalTo(expected.getContent()));
21 }
```

The following integration tests have also been included in the project:

- `rest_WithMissingClientCert_ThrowsSSLHandshakeExceptionBadCertificate`
- `rest_WithUntrustedServerCert_ThrowsSSLHandshakeExceptionUnableFindValidCertPath`

The companion Spring Boot application is available on GitHub

Two-Way SSL with SnapLogic's REST Snap

Naturally, SnapLogic's REST Snap makes this all very easy:

SnapLogic REST Snap

Thank Yous

Thank you to Ed Heneghan for correcting my initial mistakes.

Chandan Ray • 10 months ago

Amazing Article. It helped me a lot in understanding the SSL concepts. But I have one query - In both the cases of one-way ssl or the two-way ssl, in what format the server is presenting its certificate to the client or client is presenting its certificate to the server ? If it is p.12 then we are sending the private key along with the certificate. In that case it becomes security risk. Can you please elaborate.

1 ^ | v • Reply • Share ›

j.alvi ➚ Chandan Ray • 7 months ago

This is my understanding and may not be correct.

When we have to generate a certificate for the client, initially we create a private key file and the certificate request file. We then send over the certificate request file to the CA server which signs it and returns the client's public certificate. Now if the client was an external entity, the client would just send a certificate request file to the CA server and get the client's public certificate and the CA certificate.

However, an administrator may want to create the client's private key, client's certificate request file, sign the certificate request using the CA server and get the client's public certificate on his "own system". Afterwards, the administrator may package the client's public certificate, client's private key and CA cert into a package (.p12 file) and give it to the actual client machine to use it for SSL. The private key may be inside the .p12 file but the file is password protected and during the SSL handshake, the server will not be able to extract the private key out of the client's certificate.

Please let me know your thoughts or if you found something conclusive about putting the private key inside .p12 file.

@robinhowlett Great article. Can you please clarify this point ? Thanks.

^ | v • Reply • Share ›

mahesh Camath • 2 years ago

Very well written article. In general, any change to the certificates (like renew) would require web server restart. I've a Spring boot app'n with tomcat as its embedded container. I'm looking for a way to update my key store entry periodically, without restarting the JVM. Is it feasible without a reverse proxy? What would it take to achieve it. I'm wonder if writing own KeyManager extension is an acceptable practice?

1 ^ | v • Reply • Share ›

mahesh Camath ➔ mahesh Camath • 2 years ago

Oh ok found it. Need to write a EmbeddedServletContainerCustomizer i.e. in the servletContainer() method in this post, we can add customizers to TomcatEmbeddedServletContainerFactory,

```
webContainerFactory
.addConnectorCustomizers((TomcatConnectorCustomizer) (Connector connector) -> {
httpsConnector = connector;
httpsConnector.setAttribute("bindOnInit", false); // Needed with newer versions of embedded tomcat
//.....
});
```

Now that we have a handle to the connector instance (only one in my case), we just need to do

```
httpsConnector.stop();
// ******************************************************
// An changes towards the connector setting can be done here .
// For example: If we want to change alias, we can simply change it here
// i.e. get an handle to Http Protocol implementation httpsConnector.getProtocolHandler()
// and change the alias to keystore entry
// ******************************************************
httpsConnector.start();
```

1 ^ | v • Reply • Share ›

סמואל דוידוב • 12 days ago

Amazing article!! It helps to understand a lot of things. But still have some concerns regarding how to implement following thing (May be i missed in article) --- i have client and server 2 mutual authentication. Springboot applications. On the server i load from properties :

```
server.ssl.key-store=src\\main\\resources\\serverkeystore.p12
server.ssl.key-store-password=secret
server.ssl.trust-store=src\\main\\resources\\servertruststore.p12
server.ssl.trust-store-password=secret
```

And works everything fine. But i want to encrypt the password like this

```
server.ssl.key-store-password=PEgC8MYwNKVUryuVekqa/A==
```

and decrypt it within the code when the application starts up to actually "secret". Then to reload this property like this:

System.setProperty("javax.net.ssl.keyStorePassword", decryptedKSPassowrd);
But i always get a Application Failed Tomcat is already running.
I understand that the problem is tomcat already running and need to be restarted to reload this property again,

I tried so many variations to load with not reserved property names , initialising and reloading before the springboot context is starting up. But nothing helps.

May be there is some workaround in this case to handle with it?
Thanks in advance

Reply • Share ›

סמואל דוידוב ↩ **סמואל דוידוב** • 2 days ago

I don't know why, but it works with spring boot parent version 1.3.1...But for today it's pretty old and i use 1.5.1 . with this new one it doesn't work. May be someone insert new code and made it bad...or opposite fixed this as it's not good to restart even embedded tomcat for certificates...i don't know
Someone can help me?

Reply • Share ›

Måns Rolandi • 3 months ago

Thanks a lot for an excellent article. There's one thing that seems contradictory to me though. You describe the setup process thus:
1. Create self signed certificate and key: localhost_self-signed.pem and localhost_self-signed.key
2. Create openssl.cnf which refers to CA certificate and key: cacert.pem and cakey.pem
3. Create signing request for cacert.pem and cakey.pem: openssl req -new -x509 -extensions v3_ca -keyout private/cakey.pem -out cacert.pem -days 365 -config ./openssl.cnf

It seems to me that you would need the certificate/key referred to in (2) to be present in order to run the command in (3), which actually creates that cert/key pair. Am I missing something?

Reply • Share ›

Binh Thanh Nguyen • 5 months ago

Thanks, nice post

Reply • Share ›

Sagar Shah • a year ago

Thank you for sharing this! It's very useful for me to get an overall understanding of SSL flow.

One question -
Is the creation of .p12 files necessary? Is there an alternative to that? Like client-cert.p12 and server-cert.p12? What if, I just need JKS keystore to be created? Can that JKS keystore be created without .p12 file? I have client cert, private key and ca cert.

Please advise.

Reply • Share ›

will lou • a year ago

This post is really straightforward and easy to understand, thank you for putting this together, Robin

Reply • Share ›

Mario Guerrero • a year ago

The greatest post about SSL/TLS.

I like a lot and i used the springboot project.

Thanks.

Reply • Share ›

Albert Robert • a year ago

I tried to run the unit test from the everything-ssl project, all the tests pass except :
httpsRequest_With2WaySSLAndHasValidKeyStoreAndTrustStore_Returns200OK fails with:
javax.net.ssl.SSLHandshakeException: Received fatal alert: certificate_unknown
Am I missing some configuration? I tried to debug but did not find the cause and also 2 way SSL is pretty new for me.
Thanks

Reply • Share ›

Navy Chen • a year ago

what an amazing subject regarding to SSLi never find!!!

Reply • Share ›

Uwe Grünheid • a year ago

Hey Robin,

Is there a special reason why you use the server public key in the jks clients truststore and not the cacert.pem like you did in the server truststore?
As i could imagine both certificates are signed by the ca certificate so both ways should work if i got it right?

Cheers,
Uwe

▲ | ▼ • Reply • Share ›

j.alvi ➚ Uwe Grünheid • 7 months ago

I have the same question. The same CA was used to sign both client and server certificates. Hence, the trusted store on both ends (client and server) should have the CA cert inside it.

@robinhowlett Great article. It would be great if you can clarify this point.

▲ | ▼ • Reply • Share ›

j.alvi ➚ j.alvi • 6 months ago

From what I have found online is that there are 2 approaches to doing this:
1) you can store the CA's root cert into the trusted store or
2) you can store the other party's public key certificate in the trusted store

▲ | ▼ • Reply • Share ›

Tom Fennelly • 2 years ago

Hi Rob (fellow Irishman :)).

Really nice, well written and useful post. The thing I find most confusing about all the JSSE/JKS stuff is the creation of the keystores etc. You have explained it really nicely and logically here. I tried to code it myself, capturing the key creation in a shell script (https://github.com/tfennell... ... see genkeys.sh). I created a test "Main" class based on your code but I'm getting a "Exception in thread "main" javax.net.ssl.SSLHandshakeException: Received fatal alert: bad_certificate" exception ... seems like I didn't quite create the certs properly, but I can't see what I did wrong.

▲ | ▼ • Reply • Share ›

Dariusz Muszczak • 2 years ago

Hi magnificent article, thank you ! Now ssl isn't so scary, but i still have one problem. I started your spring boot app on my local PC, and then on my smartphone with android (both devices were in the same network). On smartphone I starting proxy every request to ip adres where spring boot app listen (my android had installed your keystore). Then i sent for example https://google.com request, which firstly go to your app but without any reaction in your app. Have you maybe any idea what i do wrong ? And of course sorry for my bad english.

Reply • Share ›

raj • 2 years ago

great post. one question though. my spring-boot application is both a server and a client to "secure" server. This secure server needs cert. so I want to enable only client-side cert. That is my server runs on HTTP. but I need to use client-cert to talk to secure server. how can I achieve this?

Reply • Share ›

robinhowlett Mod ➚ raj • 2 years ago

If I understand you correctly, you have a target server ("secure server"), and a client application ("client") running on a Spring Boot application ("server").

The "secure server" is the one that determines if two-way SSL (client certificates) are enforced, so, if that is a Spring Boot system also, then that would need to define the appropriate value for "server.ssl.client-auth"[1] and have the HTTPS port (e.g. 443) open.

The "client" application would then just need to include its certificate (in the keystore) in the HTTP client making the request over HTTPS[2]. The Spring Boot "server" hosting the "client" application is independent of that - it doesn't need to available over HTTPS (but why wouldn't you want that?).

[1] https://docs.spring.io/spri...
[2] https://github.com/robinhow...

Reply • Share ›

raj ➚ robinhowlett • 2 years ago

Thanks for reply. when you say "The "client" application would then just need to include its certificate (in the keystore) in the HTTP client making the request over HTTPS" do you mean truststore? In your github code linked, you do .loadTrustMaterial and .loadKeyMaterial. I would

need only loadTrustMaterial right? I guess I need this portion of the code. you mentioned in the comment that you need to import client cert into keystore using keytool.
https://github.com/robinhow...
▲ | ▼ • Reply • Share ›

Himalay Majumdar • 2 years ago
Landed here from your github code, this helped me solve my problem with spring boot loading resource file. One clarification I need however. I created a client-rest-api which calls a server-rest-api (one way call to server). My client-rest-api uses certificates provided by the server-rest-api. Does it come under one-way-ssl or two-way-ssl? Event though its just a one way call from client to server, I am thinking its two-way-ssl since here server validates that client is having the proper certificates.
▲ | ▼ • Reply • Share ›

robinhowlett Mod ➚ Himalay Majumdar • 2 years ago
If the server is hosted at an HTTPS address and it has a valid cert for the domain, then, yes, you are doing two-way SSL.

One-way SSL would be the above but not using/validating client certificates.
▲ | ▼ • Reply • Share ›

Zilev • 3 years ago
Great post, do you have an example using only one way ssl with spring? Regards
▲ | ▼ • Reply • Share ›

robinhowlett Mod ➚ Zilev • 3 years ago
It's very similar. I have unit tests on the GitHub repo[1] that demo one-way SSL.

In terms of Spring Boot configuration, you would just need to ensure client-auth[2] is false (this is my custom configuration). Then the server wouldn't prompt the clients for certificates and clients wouldn't need to use a keystore.

The default Spring Boot way is server.ssl.client-auth not being present (or "want" or "need") in the application properties[3].

[1] https://github.com/robinhow...

[2] https://github.com/robinhow...
[3] https://docs.spring.io/spri...
Reply • Share ›

Zilev ➚ robinhowlett • 3 years ago

It works!!! :D :D , thanks so much for help :D :D I have other questions please, if I want to use the same tests with an external https, how can I tests if works between a normal http?, And second question I've seen some people use this "SSLContexts.custom().loadTrustMaterial(null, new TrustSelfSignedStrategy()).build();" seems it is used for allows to trust certificates from additional KeyStores in addition to the default KeyStore, is it true? I've looked for documentation but I cann't find it :' (.Thanks a lot in advance

Reply • Share ›

robinhowlett Mod ➚ Zilev • 3 years ago

Normally, you'd enable HTTPS-only, but if you want both (like I did for demo purposes), you can (if you are using Tomcat) create a new Connector for SSL in addition to the default one[1].

The Boot documentation suggests using the application.properties configuration for HTTPS and configuring a HTTP Connector manually[2].

"SSLContexts.custom().loadTrustMaterial(null, new TrustSelfSignedStrategy()).build();": if this is being applied to the client configuration it, then it means that the client is being loaded with a null TrustStore. Unless you have set up multiple TrustManagers[2], it won't use the system's default trust store.

[1] https://github.com/robinhow...
[2] https://docs.spring.io/spri...
[3] http://stackoverflow.com/qu...
Reply • Share ›

Prateek Gupta • 3 years ago

This is amazing! I would love your feedback on this tutorial "Using HTTPs with Rails"
tutorials.pluralsight.com/r...
Reply • Share ›

robinhowlett Mod ➚ Prateek Gupta • 3 years ago

Don't know much about Rails but this looks comprehensive *thumbs up*

1 ^ | v • Reply • Share ›

ALSO ON **THE BOY WONDERS**

Spring Social Bootstrap: Create REST API SDKs and CLIs that can Record and Replay HTTP …

1 comment • 4 years ago

John Sheehan — You should check out our debugging proxy as well that allows for replay, etc: https://www.runscope.com/do...

Spring app migration: from XML to Java-based config

16 comments • 4 years ago

disqus_Z68BEtVUPC — Nice one Robin! Helped me tremendously. Hope you're well!

Building a Google Chrome Extension (including Keyboard Shortcuts and Copying to the …

1 comment • 2 years ago

Miguel Stevens — Great article! Thanks for this.

Supporting Multi-Step Commands with Spring Shell

3 comments • 4 years ago

Edward Beckett — Much Nicer :)