**Self-Signed Certificate**

## SSL

SSL, a.k.a Secure Sockets Layer, is an internet security protocol that establishes security to data in transit between two computer devices.

SSL encrypts the data transmitted between multiple devices, making it nearly impossible to decrypt

**What is an SSL Certificate?**

SSL stands for *Secure Sockets Layer*. TLS (*Transport Layer Security*) is an updated, more secure version of SSL. This is a standard technology providing a secure connection between the server and a client.

SSL certificate is used to identify a server. Clients use this certificate to verify if the server is really who it claims to be.

A Certificate consists of a public and private key pair. The private key is stored on the server, and the public key is packaged with the certificate and shared with the client during the[SSL handshake](http://javarevisited.blogspot.sg/2013/07/how-ssl-https-and-certificates-works-in-java-web-application.html).

Clients use the public key from the certificate to encrypt data, and the server uses the private key to decrypt data.

## Keystore

In Java, a [KeyStore](https://www.java67.com/2012/09/keytool-command-examples-java-add-view-certificate-ssl.html)is used to store private keys and certificates with public keys. The format in which these digital keys and certificates are stored in a Keystore is called Keystore format.

Since Java 9, the default [Keystore](https://www.java67.com/2012/12/difference-between-truststore-vs.html) format is PKCS12. We will be using the PKCS12 Keystore format in this tutorial.

**Keystore and Truststore —**is used to store SSL certificates in **Java** but there is subtle **difference between** them. **truststore** is used to store public certificates while **keystore** is used to store private certificates of client or server

To enable SSL first you need a SSL certificate signed by a certification authority (CA). Either you can buy an SSL certificate or If you only need to configure HTTPS to test your application, you can generate a self-signed certificate using a tool like OpenSSL.

We'll use either of the following certificate formats:

* PKCS12: [**Public Key Cryptographic Standards**](https://en.wikipedia.org/wiki/PKCS_12) is a password protected format that can contain multiple certificates and keys; it's an industry-wide used format.
* JKS: [**Java KeyStore**](https://en.wikipedia.org/wiki/Keystore) is similar to PKCS12; it's a proprietary format and is limited to the Java environment.

We can use either keytool or OpenSSL tools to generate the certificates from the command line. [Keytool](https://docs.oracle.com/en/java/javase/11/tools/keytool.html) is shipped with Java Runtime Environment, and OpenSSL can be downloaded from [here](https://www.openssl.org/).

$ keytool -genkey -alias localhost\_ssl -keyalg RSA -keysize 2048 -validity 700 -keypass secret -storepass secret -keystore server.jks

server:  
 port: 443  
 servlet:  
 context-path: /  
 ssl:  
 enabled: true  
 key-alias: localhost\_ssl  
 key-store: classpath:server.jks  
 key-store-type: jks  
 key-password: secret  
 key-store-password: secret

$ **keytool -genkeypair -alias local\_ssl -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore local-ssl.p12 -validity 365 -ext san=dns:localhost**

server:  
 port: 443  
 servlet:  
 context-path: /  
 ssl:  
 enabled: true  
 key-alias: local\_ssl  
 key-store: classpath:local-ssl.p12  
 key-store-type: PKCS12  
 key-password: abcd123  
 key-store-password: abcd123

Let’s understand above command –

* -genkey – is the keytool command to generate the public/private key.
* -alias – indicates the alias of the certificate, which is used by SSL/TLS layer
* -keyalg RSA -keysize 2048 -validity 700 – are indicating the crypto algorithm, keysize and certificate validity.
* -keypass secret -storepass secret – are the passwords of our truststore and keystore
* -keystore server.jks – is the actual keystore where the certificate and public/private key will be stored. Here we are using [JKS](https://en.wikipedia.org/wiki/Keystore) fromat – *Java Key Store*, there are other formats as well for keystore.

server:

ssl:

key-store: classpath:keystore.p12

key-store-password: password

key-store-type: pkcs12

key-alias: springboot

key-password: password

server:

port: 8443 1

ssl:

key-alias: yourKeyAlias 2

key-store: path/to/keystore 3

key-store-password: yourKeyStorePassword 4

key-password: yourKeyPassword 5

trust-store: path/to/trust-store 6

trust-store-password: yourTrustStorePassword 7

**server.ssl.key-password —**the password used to access the key in the key store

**server.ssl.key-alias —**the alias identifies the key in the key store

**server.ssl.key-store-type —**the type of the key store

**server.ssl.key-store-password —**the password used to access the key store

**server.ssl.key-store —**the path to the key store that holds the SSL certificate

**server.port —**server HTTP port

<https://www.codejava.net/frameworks/spring-boot/configure-https-with-self-signed-certificate>

keytool -list -v -keystore springboot.jks

**Install Self-Signed Certificate**

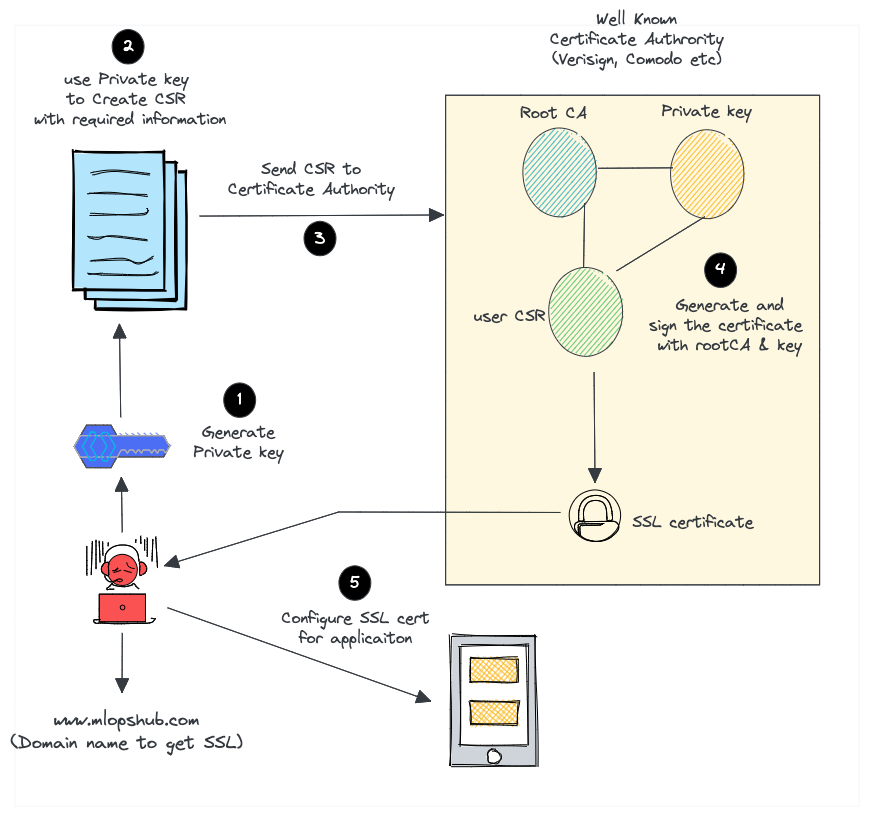
Now you have to generate a certificate file from the keystore file. Use the keytool program with this command:

**keytool -export -keystore local-ssl.p12 -alias local\_ssl -file local-cert.crt**

**What is a Self-Signed Certificate?**

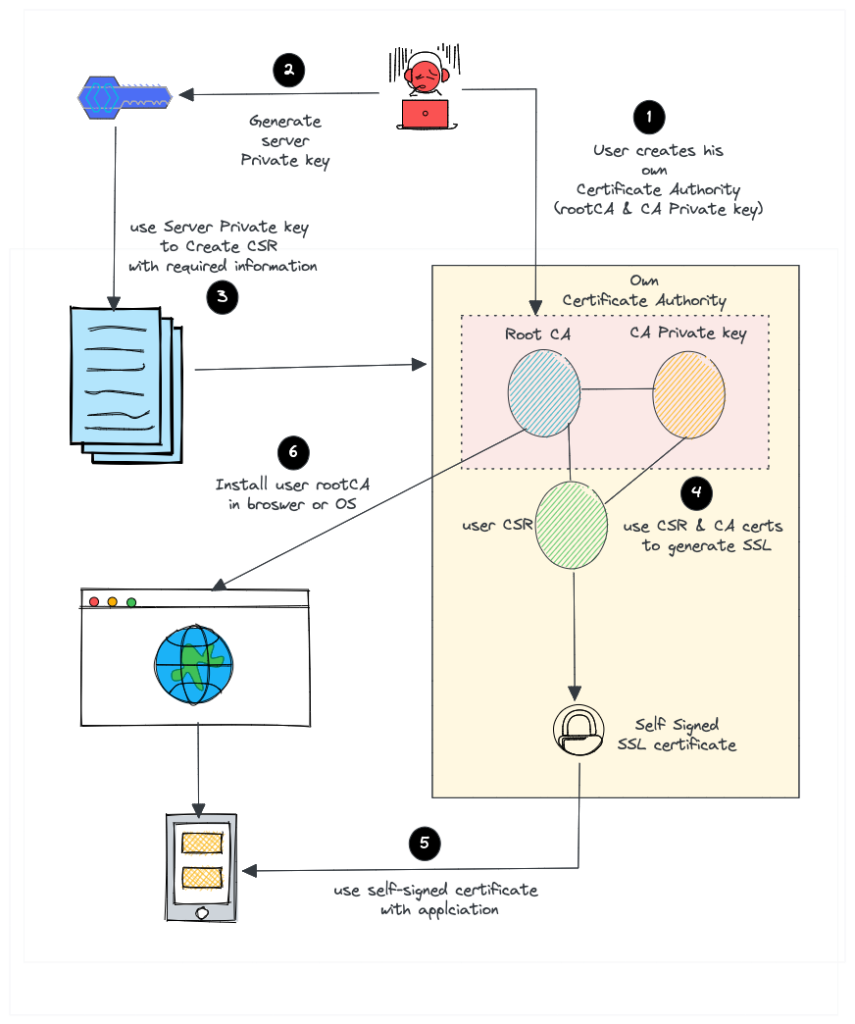
A self-signed certificate is an SSL/TSL certificate not signed by a public or private certificate authority. Instead, it is signed by the creator’s own personal or root CA certificate.

Here is what we do to request paid SSL/TLS certificate from a **well-known Certificate Authority** like Verisign or comodo.

[](https://devopscube.com/wp-content/uploads/2022/03/image-9.png)

1. Create a **certificate signing request (CSR)** with a private key. A CSR contains details about location, organization, and FQDN (Fully Qualified Domain Name).
2. Send the CSR to the trusted CA authority.
3. The CA authority will send you the SSL certificate signed by their root certificate authority and private key.
4. You can then validate and use the SSL certificate with your applications.

But **for a self-signed certificate**, here is what we do.

[](https://devopscube.com/wp-content/uploads/2022/03/image-8.png)

1. Create our own root CA certificate & CA private key (We act as a CA on our own)
2. Create a server private key to generate CSR
3. Create an SSL certificate with CSR using our root CA and CA private key.
4. Install the CA certificate in the browser or Operating system to avoid security warnings.

## Creating a Certificate Signing Request

**If we want our certificate signed, we need a certificate signing request (CSR)**. The CSR includes the public key and some additional information (such as organization and country).

Let’s create a CSR (domain.csr) from our existing private key:

## Creating a Private Key

First, we’ll create a private key. A private key helps to enable encryption, and is the most important component of our certificate.

openssl genrsa -des3 -out private.key 2048

## Creating a Certificate Signing Request

openssl req -key private.key -new -out domain.csr

**We can also create both the private key and CSR with a single command**:

openssl req -newkey rsa:2048 -keyout domain.key -out domain.csr

If we want our private key unencrypted, we can add the -nodes option:

openssl req -newkey rsa:2048 -nodes -keyout domain.key -out domain.csr

**Creating a Self-Signed Certificate**

A self-signed certificate is **a certificate that’s signed with its own private key**. It can be used to encrypt data just as well as CA-signed certificates, but our users will be shown a warning that says the certificate isn’t trusted.

Let’s create a self-signed certificate (*domain.crt*) with our existing private key and CSR:

openssl x509 -signkey private.key -in domain.csr -req -days 365 -out domain.crt

The *-days* option specifies the number of days that the certificate will be valid.

We can create a self-signed certificate with just a private key:

openssl req -key private.key -new -x509 -days 365 -out domain.crt

<https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs>

### [Generate a Private Key and a CSR](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-private-key-and-a-csr)

Use this method if you want to use HTTPS (HTTP over TLS) to secure your Apache HTTP or Nginx web server, and you want to use a Certificate Authority (CA) to issue the SSL certificate. The CSR that is generated can be sent to a CA to request the issuance of a CA-signed SSL certificate

This command creates a 2048-bit private key (domain.key) and a CSR (domain.csr) from scratch:

openssl req -newkey rsa:2048 -nodes -keyout private.key -out domain.csr

The -newkey rsa:2048 option specifies that the key should be 2048-bit, generated using the RSA algorithm. The -nodes option specifies that the private key should not be encrypted with a pass phrase. The -new option, which is not included here but implied, indicates that a CSR is being generated.

### [Generate a CSR from an Existing Private Key](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-csr-from-an-existing-private-key)

Use this method if you already have a private key that you would like to use to request a certificate from a CA.

This command creates a new CSR (domain.csr) based on an existing private key (domain.key):

openssl req -key domain.key -new -out domain.csr

Answer the CSR information prompt to complete the process.

The -key option specifies an existing private key (domain.key) that will be used to generate a new CSR. The -new option indicates that a CSR is being generated.

### [Generate a CSR from an Existing Certificate and Private Key](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-csr-from-an-existing-certificate-and-private-key)

Use this method if you want to renew an existing certificate but you or your CA do not have the original CSR for some reason. It basically saves you the trouble of re-entering the CSR information, as it extracts that information from the existing certificate.

This command creates a new CSR (domain.csr) based on an existing certificate (domain.crt) and private key (domain.key):

openssl x509 -in domain.crt -signkey domain.key -x509toreq -out domain.csr

## [Generating SSL Certificates](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generating-ssl-certificates)

If you would like to use an SSL certificate to secure a service but you do not require a CA-signed certificate, a valid (and free) solution is to sign your own certificates.

A common type of certificate that you can issue yourself is a self-signed certificate. A self-signed certificate is a certificate that is signed with its own private key. Self-signed certificates can be used to encrypt data just as well as CA-signed certificates, but your users will be displayed a warning that says that the certificate is not trusted by their computer or browser. Therefore, self-signed certificates should only be used if you do not need to prove your service’s identity to its users (e.g. non-production or non-public servers).

This section covers OpenSSL commands that are related to generating self-signed certificates.

### [Generate a Self-Signed Certificate](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-self-signed-certificate)

Use this method if you want to use HTTPS (HTTP over TLS) to secure your Apache HTTP or Nginx web server, and you do not require that your certificate is signed by a CA.

This command creates a 2048-bit private key (domain.key) and a self-signed certificate (domain.crt) from scratch:

openssl req -newkey rsa:2048 -nodes -keyout domain.key -x509 -days 365 -out domain.crt

nswer the CSR information prompt to complete the process.

The -x509 option tells req to create a self-signed certificate. The -days 365 option specifies that the certificate will be valid for 365 days. A temporary CSR is generated to gather information to associate with the certificate.

### [Generate a Self-Signed Certificate from an Existing Private Key](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-self-signed-certificate-from-an-existing-private-key)

Use this method if you already have a private key that you would like to generate a self-signed certificate with it.

This command creates a self-signed certificate (domain.crt) from an existing private key (domain.key):

openssl req -key domain.key -new -x509 -days 365 -out domain.crt

Answer the CSR information prompt to complete the process.

The -x509 option tells req to create a self-signed certificate. The -days 365 option specifies that the certificate will be valid for 365 days. The -new option enables the CSR information prompt.

### [Generate a Self-Signed Certificate from an Existing Private Key and CSR](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#generate-a-self-signed-certificate-from-an-existing-private-key-and-csr)

Use this method if you already have a private key and CSR, and you want to generate a self-signed certificate with them.

This command creates a self-signed certificate (domain.crt) from an existing private key (domain.key) and (domain.csr):

openssl x509 -signkey domain.key -in domain.csr -req -days 365 -out domain.crt

The -days 365 option specifies that the certificate will be valid for 365 days.

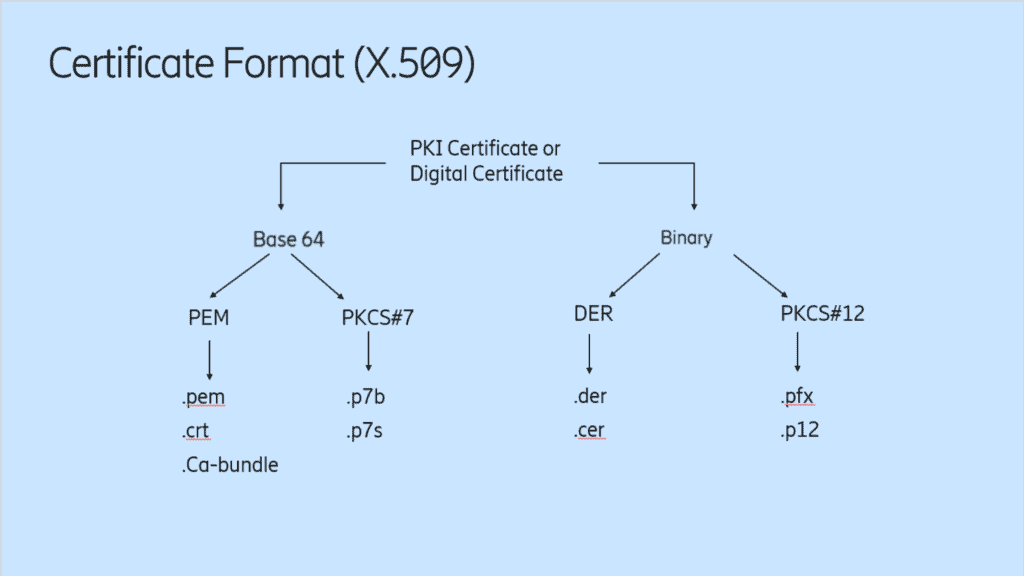
## [View Certificates](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#view-certificates)

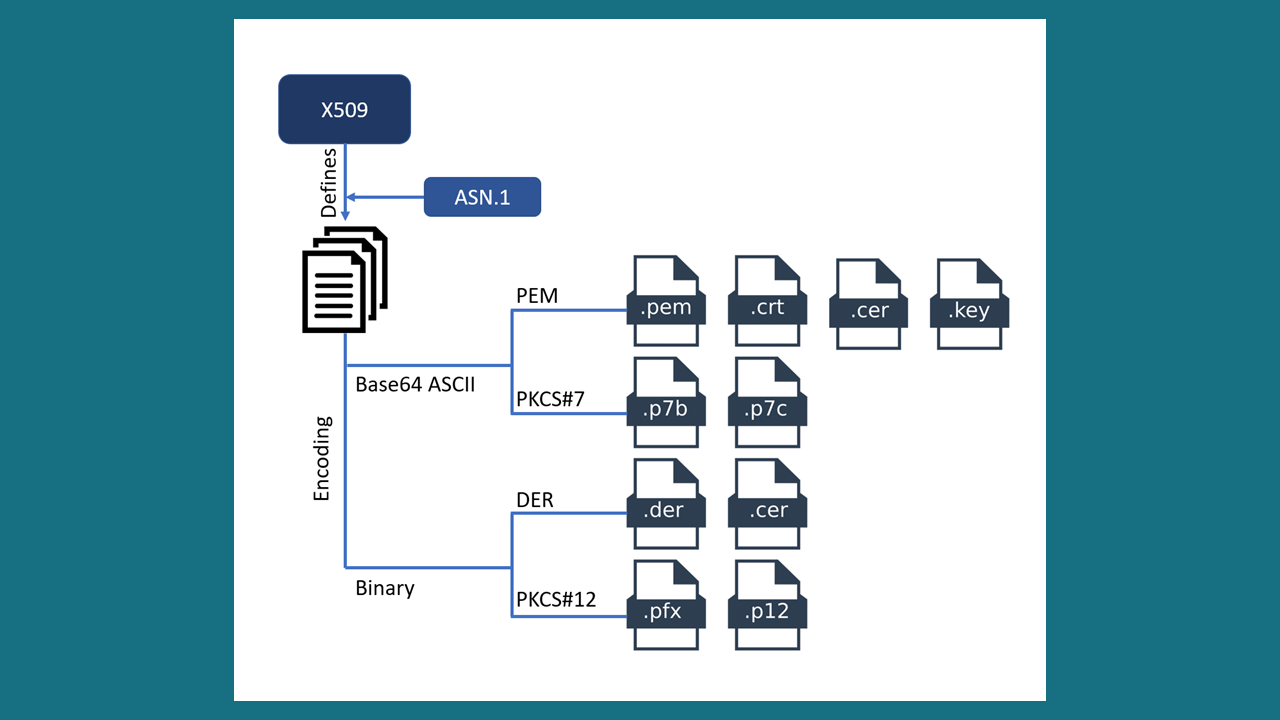
Certificate and CSR files are encoded in PEM format, which is not readily human-readable.

This section covers OpenSSL commands that will output the actual entries of PEM-encoded files.

### [View CSR Entries](https://www.digitalocean.com/community/tutorials/openssl-essentials-working-with-ssl-certificates-private-keys-and-csrs#view-csr-entries)

This command allows you to view and verify the contents of a CSR (domain.csr) in plain text:





a). Generate a private key.

openssl genrsa -out private.key 2048

b). Generate a public key.

openssl rsa -in private.key -pubout -out public.key

c). Generate a CSR file with the help of the private key.

openssl req -new -key private.key -out localcsr.csr

d). Generate a CRT file with the help of the private key & CSR file.

openssl x509 -in localcsr.csr -out localcrt.crt -req -signkey private.key -days 365

e). Generate .pfx file (finally) with the help of the private key & CRT file.

openssl pkcs12 -export -out localpfx.pfx -inkey private.key -in localcrt.crt

f). **p12**

openssl pkcs12 -export -out localp12.p12 -inkey private.key -in localcrt.crt

g) pfx -> cer

openssl pkcs12 -in localpfx.pfx -clcerts -nokeys -out localcer.cer

h) CRT -> PEM

***openssl x509 -in cert.crt -out cert.pem***

* **-in yourfile.pfx**: Specifies the input PFX file.
* **-clcerts**: Extracts only the client certificates (omit this option if you want to extract other certificates, such as CA certificates).
* **-nokeys**: Specifies not to include private keys in the output (as you're converting to CER, which only contains certificates).

If we get any error

While using cmd:

openssl x509 -outform der -in yourfile.pem -out yourfile.cer

The error message "Could not read certificate from private.pem" suggests that OpenSSL is attempting to read a private key file (**private.pem**) as if it were a certificate file. If you want to convert a PEM-encoded private key to CER format, you should use the **openssl rsa** command, not the **openssl x509** command.

Private key

openssl genrsa -out private.key 2048

Then a CSR file:

openssl req -new -key private.key -out localcsr.csr

CRt file

openssl x509 -in localcsr.csr -out localcrt.crt -req -signkey private.key -days 365 -extfile **csr.config -extensions v3\_ca**

**p12**

openssl pkcs12 -export -out localp12.p12 -inkey private.key -in localcrt.crt

**OR**Private key

openssl genrsa -out private.key 2048

Then a CSR file:

openssl req -new -key private.key -out localcsr.csr -subj "/C=GB/ST=London/L=London/O=Global\ Security/OU=IT\ Department/CN=example.com"

CRt file

openssl x509 -in localcsr.csr -out localcrt.crt -req -signkey private.key -days 365 -extfile **v3.ext**

**p12**

openssl pkcs12 -export -out localp12.p12 -inkey private.key -in localcrt.crt

**v3.ext**

**subjectKeyIdentifier = hash**

**authorityKeyIdentifier = keyid:always,issuer:always**

**basicConstraints = CA:TRUE**

**keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign**

**subjectAltName = @alt\_names**

**issuerAltName = issuer:copy**

**[alt\_names]**

**DNS.1 = localhost**

**IP.1= 127.0.0.1**

**csr.config**

[req]

days = 3650

serial = 1

distinguished\_name = req\_distinguished\_name

x509\_extensions = v3\_ca

[req\_distinguished\_name]

countryName = IN

stateOrProvinceName = MP

localityName = Indore

organizationName = US Bank

organizationalUnitName = ICS\_MM

commonName = localhost

emailAddress = rahul@usbank.com

[ v3\_ca ]

# The extentions to add to a self-signed cert

subjectKeyIdentifier = hash

authorityKeyIdentifier = keyid:always,issuer:always

# THIS IS VERY IMPORTANT IF YOU WANT TO USE THIS CERTIFICATION AS AN AUTHORITY!!!

basicConstraints = CA:TRUE

keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign

subjectAltName = @alt\_names

issuerAltName = issuer:copy

[alt\_names]

DNS.1 = localhost

IP.1= 127.0.0.1

**mTLS Root cert**

<https://medium.com/@salarai.de/how-to-enable-mutual-tls-in-a-sprint-boot-application-77144047940f>

## Generating Root Certificates

Steps to generate a self-signed root certificate. We will use it for signing client and server certificates later in the story.

1. This command generates a private key for the root certificate.

openssl genrsa -out rootCA.key 2048

1. Request a certificate from openssl using the key generated in the previous step

openssl req -x509 -new -key rootCA.key -sha256 -days 365 -out rootCA.pem

## Signing Server Certificate

1. Let’s create a private key and then a CSR for our server certificate.

openssl genrsa -out server.key 2048

1. Now request a CSR with the key as input key:

openssl req -new -sha256 -key server.key -out server.csr

1. let’s sign the server certificate with the given CSR.

openssl x509 -req -in server.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out server.pem -days 365 -sha256

## Signing Client Certificate

As mentioned in the background, mutual TLS is based on both parties authenticating each other. If it were to be one-way TLS, we would not need the client certificate, because server would not request it. In this case however, we’d like the client to present its certificate and we’d like the server to authenticate it.

1. Let’s create client certificates so we can use them to call the API.

openssl genrsa -out client.key 2048

1. Then create a CSR for the client in the same way

openssl req -new -sha256 -key client.key -out client.csr

1. Then we sign the client certificate also in the same way

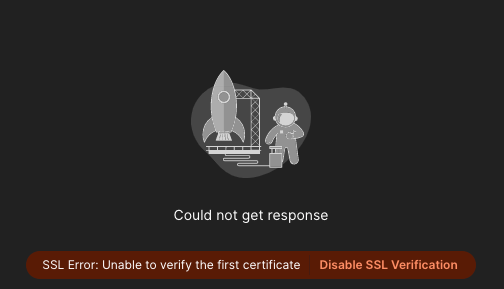
openssl x509 -req -in client.csr -CA rootCA.pem -CAkey rootCA.key -CAcreateserial -out client.pem -days 365 -sha256

1. Navigate to the directory where you have the certificates and run the following command to create a key store from server certificate and its private key.

openssl pkcs12 -export -in server.pem -out keystore.p12 -name server -inkey server.key

Put the following snippet in **application.yaml** file and restart the application.

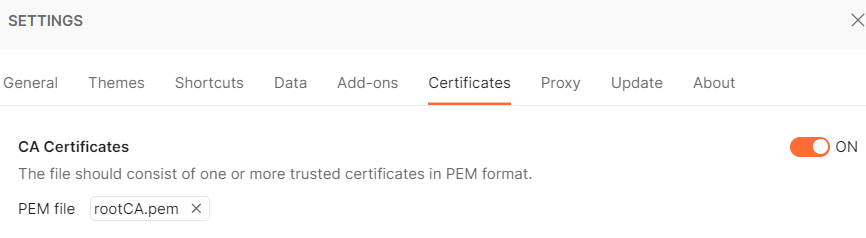
server:  
 ssl:  
 enabled: true  
 key-store: "classpath:keystore.p12"  
 key-store-password: changeit  
 key-store-type: PKCS12



Seems like Postman cannot verify our server’s certificate during the handshake which makes sense because Postman has no idea that we are a certificate authority. It already trusts other public CAs on internet, but somehow we need to get Postman to trust us as a root CA.

Navigate to Postman preferences and then certificates tab where you can add CA certificate. There’s a section at the top titled **CA Certificates** which provides you the ability to manually trust a CA.

select the **rootCA.pem** file so Postman can trust the CA (us). Re-run the request again and the connection should be verified now.

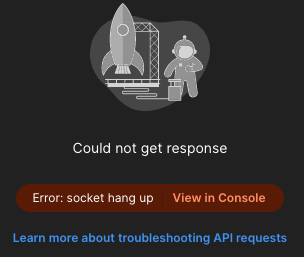


## Enabling mutual TLS on a Sample Spring Boot Application

However to get the mutual authentication between the client and the server, we need to change our configuration a little bit so that our server asks for the client certificate too. Navigate to **application.yaml** file and add the following properties:

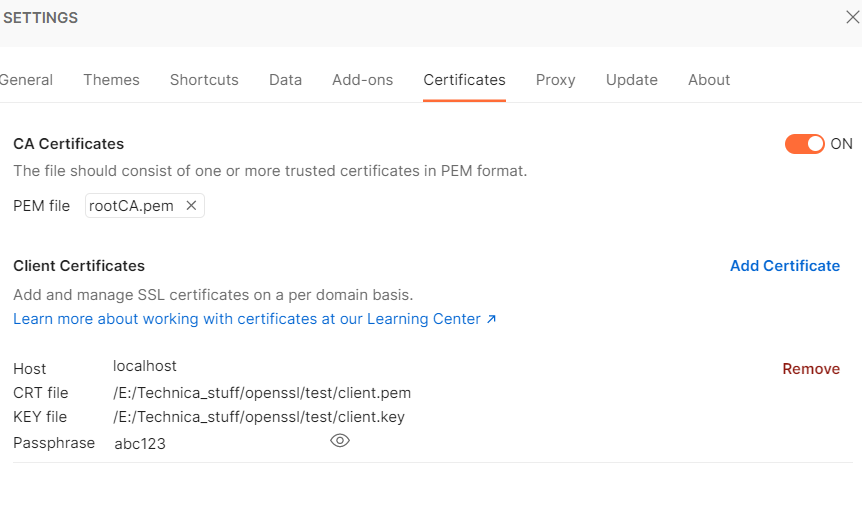
client-auth: need

This key allows us to configure whether we want client authentication (a.k.a mutual authentication) or not. It can be either NEED, WANT and NONE. NEED indicates that the server must validate client certificate while WANT also asks for the client certificate with the key difference that it keeps the connection if no authentication is provided. Using NONE, the client certificate is never requested. Restart the application and try the endpoint again.



Postman doesn’t seem to be happy about the result. In this case, the server could not verify the client because client was not presenting any sort of certificates and the connection was dropped during the handshake. What we need to do to get this sorted out is to send client certificate along with the request when calling the API.

Navigate to certificates tab in Postman preferences again and click on **Add Certificate**and add the following entry:  
**- host:**localhost  
**- port:** 443  
- **CRT File:**/path/to/client.pem  
**- KEY File:**/path/to/client.key

Resend the request again and Postman complains about the same error. Although both client and server present their certificates to each other, the connection still fails to be established. whenever the client calls the endpoint and somewhere in those lines, there’s a message saying **“unknown certificate”**. This exception gets thrown since our server is not able to verify the client certificate because it does not trust the root CA of the client certificate. It knows that it is a valid certificate but it has no idea where it got signed and hence rejects the request. (just like Postman).

It turns out that enabling the client certificate authentication requires trusting the other party’s root CA

Java does this trusting mechanism with object bundles called **Trust Store**. Just like key stores that hold server’s certificate, trust stores hold certificates that the application needs to trust in order to validate and authenticate clients. Java uses its own trust store by default which is shipped with all JRE(s), Let’s create a trust store holding the root CA that signed the client certificate. Navigate to where the certificates for the client are and run the following command:

keytool -import -file rootCA.pem -alias rootCA -keystore truststore.p12

Now let’s put the **truststore.p12** file inside **src/main/resources** folder and point to the file using following properties:

trust-store: "classpath:truststore.p12"  
trust-store-password: changeit  
trust-store-type: PKCS12

Excellent! That’s how we do mTLS in Spring! Although our sample application is just for testing purposes

curl -kv --cert client.pem --key client.key --pass "abc123" <https://localhost/hello>

# [**How to know if certificate is self-signed**](https://security.stackexchange.com/questions/93162/how-to-know-if-certificate-is-self-signed)

When certificate is self-signed, then issuer and subject field contains the same value.

openssl x509 -subject -issuer -noout -in localcrt.crt

I understand that the server offers the client its SSL certificate, which contains the signature of a CA. I also understand that the client will usually have a list of certificate authorities it trusts. But what if the SSL certificate is signed by a CA that the client is not aware of? How will the client validate the certificate then?

When the client is verifying a certificate, there are three possibilities:

* The certificate is signed by a CA that the client already trusts (and for which it knows the public key). In this case the client treats the certificate as valid.
* The certificate is signed by a CA about which the client has no knowledge at all. In this case the client treats the certificate as invalid (and the browser will likely display a warning message instead of loading the page).
* The certificate is signed by a CA that the client doesn't know, but which has a certificate that is signed by a CA that the client *does* know. (In this case the server must usually send both its own certificate, and the certificate of the CA - called the "intermediate CA" - that signed its certificate). Since the intermediate CA's certificate is signed by a CA that the client already trusts, it knows can trust it, and since the server's certificate is signed by the intermediate CA, the client knows it can trust it too.

Note that CA certificates are "special" - just because you have a certificate signed by a trusted CA, that doesn't mean you can then sign other certificates and have clients trust them - unless your certificate is marked as being valid for signing other certificates.

Typically keystores and trust stores are used when our applications need to communicate securely over SSL/TLS.

These keystores and trust stores are password-protected files that reside on the same file system as the application. The default and the most widely used format for these files are JKS (Java Keystore). At least that was the case until Java 8.

With Java 9, the default Keystore format was changed from JKS to PKCS12. The most noteworthy difference between JKS and PKCS12 is that while JKS was a format specific to Java, PKCS12 is a standardized and language-neutral way of storing encrypted private keys and certificates.

"A PKCS12(Public-Key Cryptography Standards) defines an archive file format for storing server certificates, intermediate certificate if any, and private key into a single encryptable file"

Pfx → JKS:

keytool -importkeystore -srckeystore CERTIFICATE.pfx -srcstoretype PKCS12 -deststoretype JKS -destkeystore keystore.jks

You can run the following command to list the content of your keystore file (and alias name):

keytool -v -list -keystore keystore.jks

### **Example of RSA Algorithm**

Let us take an example of this procedure to learn the concepts. For ease of reading, it can write the example values along with the algorithm steps.

* Choose two large prime numbers P and Q

Let P = 47, Q = 17

* Calculate N = P x Q

We have, N = 7 x 17 = 119.

* Choose the public key (i.e., the encryption key) E such that it is not an element of (P -1) x (Q – 1)
  + Let us find (7 - 1) x (17 -1) = 6 x 16 = 96
  + The factors of 96 are 2, 2, 2, 2, 2, and 3 (because 96 = 2 x 2 x 2 x 2 x 2 x 3).
  + Therefore, it can select E such that none of the factors of E is 2 and 3. We cannot choose E as 4 (because it has 2 as a factor), 15 (because it has 3 as a factor) and 6 (because it has 2 and 3 both as factors).
  + Let us choose E as 5 (it can have been any other number that does not its factors as 2 and 3).
* Choose the private key (i.e., the decryption key) D including the following equation is true:

(D x E) mod (P – 1) x (Q – 1) = 1

* + Let us substitute the values of E, P, and Q in the equation.
  + We have (D x 5) mod (7 – 1) x (17 – 1) = 1.
  + That is, (D x 5) mod (6) x (16) = 1.
  + That is, (D x 5) mod (96) = 1
  + After some calculations, let us take D = 77. Then the following is true: (77 x 5) mod (96) = 385 mod 96 = 1 which is what we wanted.
* For encryption, calculate the cipher text (CT) from the plain text (PT) as follows:

CT = PTE mod N

Let us assume that we want to encrypt plain text 10. Then, we have

CT = 105 mod 119 = 100000 mod 119 = 40.

* Send CT as the cipher text to the receiver.

Send 40 as the cipher text to the receiver.

* For decryption, calculate the plain text (PT) from the cipher text (CT) as follows:

PT = CTD mod N

It perform the following:

PT = CTDmod N

That is,

PT = 4077mod 119 = 10, which was the original plaintext of step5.

* Choose p = 3 and q = 11
* Compute n = p \* q = 3 \* 11 = 33
* Compute φ(n) = (p - 1) \* (q - 1) = 2 \* 10 = 20
* Choose e such that 1 < e < φ(n) and e and φ (n) are coprime. Let e = 7
* Compute a value for d such that (d \* e) % φ(n) = 1. One solution is d = 3 [(3 \* 7) % 20 = 1]
* Public key is (e, n) => (7, 33)
* Private key is (d, n) => (3, 33)
* The encryption of *m = 2* is *c = 27 % 33 = 29*
* The decryption of *c = 29* is *m = 293 % 33 = 2*

**“ASN1 (Abstract Syntax Notation 1) is a means of describing digital objects. If simply says ANS describes digital objects and their formats”**

An X.509 certificate includes the public key as well as information about the person or entity to whom the certificate is issued to, information about the certification authority issuing the certificate, and information about the certificate.

X.509 certificates comes with two formats

1. PEM format
2. DER format

## PEM format

It is the most common format that Certificate Authorities issue certificates in. It contains the **‘ — –BEGIN CERTIFICATE — –”** and **“ — –END CERTIFICATE — –”** statements

Following are some special attributes of PEM format certificates

1. They are **Base64** encoded **ASCII** files
2. They have extensions such as **.pem**,**.crt**,**.cer**,**.key**
3. Apache and similar servers uses PEM format certificates

PEM format certificates may contains

1. Private keys (RSA or DSA)
2. Public keys (RSA or DSA)
3. X.509 certificates

## DER format

It is a Binary form of ASCII PEM format certificate. All types of Certificates & Private Keys can be encoded in DER format.

Following are some special attributes of DER format certificates

1. They are Binary format files
2. They have extensions **.cer**&**.der**
3. DER is typically used in Java platform

DER format certificates may contains

1. Private keys (RSA or DSA)
2. Public keys (RSA or DSA)
3. X.509 certificates

openssl x509 -inform DER -in local-cert.crt -out certificate.pem