Spring

How to enable HTTPS in a Spring Boot Java application

Setting up HTTPS for Spring Boot requires two steps: getting an SSL certificate and configuring SSL in Spring Boot. Whether you're going to generate a self-signed certificate or you have already got one by a CA, I'll show you how to enable HTTPS in a Spring Boot application.

Thomas Vitale
Jul 9, 2017 • 8 min read

Setting up HTTPS for Spring Boot requires two steps:

1. Getting an SSL certificate;
2. Configuring SSL in Spring Boot.

We can generate an SSL certificate ourselves (self-signed certificate). Its use is intended just for development and testing purposes. In production, we should use a certificate issued by a trusted Certificate Authority (CA).

In either case, we're going to see how to enable HTTPS in a Spring Boot application.

Introduction

In this tutorial, we're going to:

1. Get an SSL certificate
   - Generate a self-signed SSL certificate
   - Use an existing SSL certificate
2. Enable HTTPS in Spring Boot
3. Redirect HTTP requests to HTTPS
4. Distribute the SSL certificate to clients.

If you don't already have a certificate, follow the step 1a. If you have already got an SSL certificate, you can follow the step 1b.

Throughout this tutorial, I'll use the following technologies and tools:

- Java 11+
- Spring Boot 2.5+
- keytool

Keytool is a certificate management utility provided together with the JDK, so if you have the JDK installed, you should already have keytool available. To check it, try running the command `keytool --help` from your Terminal prompt. Note that if you are on Windows, you might need to launch it from the *\bin* folder. For more information about this utility, you can read the official documentation.
On GitHub, you can find the source code for the application we are building in this tutorial.

1a. Generate a self-signed SSL certificate

First of all, we need to generate a pair of cryptographic keys, use them to produce an SSL certificate and store it in a keystore. The keytool documentation defines a keystore as a database of "cryptographic keys, X.509 certificate chains, and trusted certificates".
To enable HTTPS, we'll provide a Spring Boot application with this keystore containing the SSL certificate.

The two most common formats used for keystores are JKS, a proprietary format specific for Java, and PKCS12, an industry-standard format. JKS used to be the default choice, but since Java 9 it's PKCS12 the recommended format. We're going to see how to use both.

Generate an SSL certificate in a keystore

Let's open our Terminal prompt and write the following command to create a **JKS keystore**:

```
keytool -genkeypair -alias springboot -keyalg RSA -keysize 4096 -storetype JKS -keystore springboot.jks -validity 3650
-storepass password
```

To create a **PKCS12 keystore**, and we should, the command is the following:

```
keytool -genkeypair -alias springboot -keyalg RSA -keysize 4096 -storetype PKCS12 -keystore springboot.p12 -validity 3650
-storepass password
```

Let's have a closer look at the command we just run:

- `genkeypair`: generates a key pair;
- `alias`: the alias name for the item we are generating;
- `keyalg`: the cryptographic algorithm to generate the key pair;
- `keysize`: the size of the key;
- `storetype`: the type of keystore;
- `keystore`: the name of the keystore;
- `validity`: validity number of days;

- `storepass`: a password for the keystore.

When running the previous command, we will be asked to input some information, but we are free to skip all of it (just press *Return* to skip an option). When asked if the information is correct, we should type *yes*. Finally, we hit return to use the keystore password as key password as well.

```
What is your first and last name?
    [Unknown]:
What is the name of your organizational unit?
    [Unknown]:
What is the name of your organization?
    [Unknown]:
What is the name of your City or Locality?
    [Unknown]:
What is the name of your State or Province?
    [Unknown]:
What is the two-letter country code for this unit?
    [Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
    [no]: yes

Enter key password for <springboot>
    (RETURN if same as keystore password):
```

At the end of this operation, we'll get a keystore containing a brand new SSL certificate.

Verify the keystore content

To check the content of the keystore following the **JKS** format, we can use keytool again:

```
keytool -list -v -keystore springboot.jks
```

To test the content of a keystore following the **PKCS12** format:

```
keytool -list -v -keystore springboot.p12
```

Convert a JKS keystore into PKCS12

Should we have already a JKS keystore, we have the option to migrate it to PKCS12; keytool has a convenient command for that:

```
keytool -importkeystore -srckeystore springboot.jks -destkeystore springboot.p12 -deststoretype pkcs12
```

1b. Use an existing SSL certificate

In case we have already got an SSL certificate, for example, one issued by Let's Encrypt, we can import it into a keystore and use it to enable HTTPS in a Spring Boot application.
We can use keytool to import our certificate in a new keystore.

```
keytool -import -alias springboot -file myCertificate.crt -keystore springboot.p12 -storepass password
```

To get more information about the keystore and its format, please refer to the previous section.

2. Enable HTTPS in Spring Boot

Whether our keystore contains a self-signed certificate or one issued by a trusted Certificate Authority, we can now set up Spring Boot to accept requests over HTTPS instead of HTTP by using that certificate.
The first thing to do is placing the keystore file inside the Spring Boot project. For testing purposes, we want to put it in the *resources* folder or the root folder. In production, you probably want to use a secret management solution to handle the keystore.
Then, we configure the server to use our brand new keystore and enable HTTPS.

Enable HTTPS in Spring Boot

To enable HTTPS for our Spring Boot application, let's open our *application.yml* file (or *application.properties*) and define the following properties:

```
server:
  ssl:
    key-store: classpath:keystore.p12
    key-store-password: password
    key-store-type: pkcs12
    key-alias: springboot
    key-password: password
  port: 8443
```

application.yml

Configuring SSL in Spring Boot

Let's have a closer look at the SSL configuration we have just defined in our Spring Boot application properties.

- `server.port`: the port on which the server is listening. We have used `8443` rather than the default `8080` port.
- `server.ssl.key-store`: the path to the key store that contains the SSL certificate. In our example, we want Spring Boot to look for it in the classpath.
- `server.ssl.key-store-password`: the password used to access the key store.
- `server.ssl.key-store-type`: the type of the key store (JKS or PKCS12).
- `server.ssl.key-alias`: the alias that identifies the key in the key store.
- `server.ssl.key-password`: the password used to access the key in the key store.

Redirect to HTTPS with Spring Security

When using Spring Security, we can configure it to automatically block any request coming from a non-secure HTTP channel and redirect them to HTTPS.
Let's create a *SecurityConfig* class to gather the security policies and configure the application to require a secure channel for all requests. We can also temporarily authorize all requests so we can focus on testing the HTTPS behaviour rather than the user authentication strategy.

```
@Configuration
public class SecurityConfig {


  @Bean
  SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
      .requiresChannel(channel ->
          channel.anyRequest().requiresSecure())
      .authorizeRequests(authorize ->
          authorize.anyRequest().permitAll())
      .build();
    }
}
```

SecurityConfig.java

For more information about how to configure SSL in Spring Boot, you can have a look at the Reference Guide. If you want to find out which properties are available to configure SSL, you can refer to the definition in the code-base.
Congratulations! You have successfully enabled HTTPS in your Spring Boot application! Give it a try: run the application, open your browser, visit *https://localhost:8443*, and check if everything works as it should.
If you're using a self-signed certificate, you will probably get a security warning from the browser and need to authorize it to open the web page anyway.

3. Multiple connectors for HTTP and HTTPS

Spring allows defining just one network connector in *application.properties* (or *application.yml*). We used it for HTTPS and relied on Spring Security to redirect all HTTP traffic to HTTPS.
What if we need both HTTP and HTTPS connectors in Tomcat and redirect all requests to the second one? We can keep the HTTPS configuration in the *application.yml* file and we set up the HTTP connector programmatically.

```
@Configuration
public class ServerConfig {
  @Bean
  public ServletWebServerFactory servletContainer() {
    TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory() {
      @Override
      protected void postProcessContext(Context context) {
        var securityConstraint = new SecurityConstraint();
        securityConstraint.setUserConstraint("CONFIDENTIAL");
        var collection = new SecurityCollection();
        collection.addPattern("/*");
        securityConstraint.addCollection(collection);
        context.addConstraint(securityConstraint);
      }
    };
```

```
    tomcat.addAdditionalTomcatConnectors(getHttpConnector());
    return tomcat;
  }

  private Connector getHttpConnector() {
    var connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
    connector.setScheme("http");
    connector.setPort(8080);
    connector.setSecure(false);
    connector.setRedirectPort(8443);
    return connector;
  }

}
```

ServerConfig.java

4. Distribute the SSL certificate to clients

When using a self-signed SSL certificate, our browser won't trust our application and will warn the user that it's not secure. And that'll be the same with any other client. It's possible to make a client trust our application by providing it with our certificate.

Extract an SSL certificate from a keystore

We have stored our certificate inside a keystore, so we need to extract it. Again, keytool supports us very well:

```
keytool -export -keystore springboot.p12 -alias springboot -file myCertificate.crt
```

The keystore can be in JKS or PKCS12 format. During the execution of this command, keytool will ask us for the keystore password that we set at the beginning of this tutorial (the extremely secure *password*).
Now we can import our certificate into our client. Later, we'll see how to import the certificate into the JRE in case we need it to trust our application.

Make a browser trust an SSL certificate

When using a keystore in the industry-standard PKCS12 format, we should be able to use it directly without extracting the certificate.
I suggest you check the official guide on how to import a PKCS12 file into your specific client. On macOS, for example, we can directly import a certificate into the Keychain Access (which browsers like Safari, Firefox and Brave rely on to manage certificates).

If deploying the application on *localhost*, we may need to do a further step from our browser: enabling insecure connections with *localhost*.
In Firefox, we are shown an alert message. To access the application, we need to explicitly define an exception for it and make Firefox trust the certificate.

In Brave, we can write the following URL in the search bar: `brave://flags/#allow-insecure-localhost` and activate the relative option.

Import an SSL certificate inside the JRE keystore

To make the JRE trust our certificate, we need to import it inside *cacerts*: the JRE trust store in charge of holding all certificates that can be trusted.
First, we need to know the path to our JDK home. A quick way to find it, if we are using Eclipse or STS as our IDE, is by going to *Preferences* > *Java* > *Installed JREs*.
If using IntelliJ IDEA, we can access this information by going to *Project Structure* > *SDKs* and look at the value of the *JDK home path* field.
On macOS, it could be something like */Library/Java/JavaVirtualMachines/adoptopenjdk-16.jdk/Contents/Home.* In the following, we'll refer to this location by using the placeholder `$JDK_HOME`.
Then, from our Terminal prompt, let's insert the following command (we might need to run it with administrator privileges by prefixing it with `sudo`):

```
keytool -importcert -file myCertificate.crt -alias springboot -keystore $JDK_HOME/jre/lib/security/cacerts
```

We'll be asked to input the JRE keystore password. If you have never changed it, it should be the default one: *changeit* or *changeme*, depending on the operating system.
Finally, keytool will ask if you want to trust this certificate: let's say *yes*.
If everything went right, we'd see the message *Certificate was added to keystore*. Great!

Conclusion

In this tutorial, we have seen how to generate a self-signed SSL certificate, how to import an existing certificate into a keystore, how to use it to enable HTTPS inside a Spring Boot application, how to redirect HTTP to HTTPS and how to extract and distribute the certificate to clients.
On GitHub, you can find the source code for the application we have built in this tutorial.
If you want to protect the access to some resources of your application, consider using Keycloak for the authentication and authorization of the users visiting your Spring Boot or Spring Security application.

References

- Spring Boot Docs - Configure SSL
- Spring Security Docs - Redirect to HTTPS

Last update: 26/06/2021
