

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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

9 JUL 2017 • 8 MIN READ



Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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. Examples will be shown both for Spring Boot 1 and Spring Boot 2.

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.

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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 <tomcat>
(RETURN if same as keystore password):
```

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

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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. We want to put it in the *resources* folder or the root folder.

Then, we configure the server to use our brand new keystore and enable https. Let's go through the steps both for Spring Boot 1 and Spring Boot 2.

Enable HTTPS in Spring Boot 1

Let's open our *application.properties* file (or *application.yml*) and define the following properties:

```
server.port=8443

server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=password
server.ssl.key-alias=tomcat
```


Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

application.properties (Spring Boot 1)

Enable HTTPS in Spring Boot 2

To enable HTTPS for our Spring Boot 2 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: tomcat
    key-password: password
  port: 8443
```

application.yml (Spring Boot 2)

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

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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.

Configure Spring Security to require HTTPS requests

When using Spring Security, we can configure it to require automatically block any request coming from a non-secure HTTP channel.

In a Spring Boot 1 application, we can achieve that by setting the `security.require-ssl` property to `true`, without explicitly touching our Spring Security configuration class.

To achieve the same result in a Spring Boot 2 application, we need to extend the `WebSecurityConfigurerAdapter` class, since the `security.require-ssl` property has been deprecated.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```


Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

Configuring Tomcat for Spring Boot 1

```
@Configuration
public class ServerConfig {

    @Bean
    public EmbeddedServletContainerFactory servletContainer() {
        TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFactory() {
            @Override
            protected void postProcessContext(Context context) {
                SecurityConstraint securityConstraint = new SecurityConstraint();
                securityConstraint.setUserConstraint("CONFIDENTIAL");
                SecurityCollection collection = new SecurityCollection();
                collection.addPattern("/*");
                securityConstraint.addCollection(collection);
                context.addConstraint(securityConstraint);
            }
        };
        tomcat.addAdditionalTomcatConnectors(getHttpConnector());
        return tomcat;
    }

    private Connector getHttpConnector() {
        Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setScheme("http");
        connector.setPort(8080);
        connector.setSecure(false);
        connector.setRedirectPort(8443);
    }
}
```

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

ServerConfig.java (Spring Boot 1)

Configuring Tomcat for Spring Boot 2

```
@Configuration
public class ServerConfig {

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

    private Connector getHttpConnector() {
```

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

```
        connector.setSecure(false);  
        connector.setRedirectPort(8443);  
        return connector;  
    }  
}
```

ServerConfig.java (Spring Boot 2)

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 keystore.jks -alias tomcat -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*).

Thomas Vitale – How to enable HTTPS in a Spring Boot Java 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, Chrome and Opera 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 Chrome, we can write the following URL in the search bar: `chrome://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.

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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-8.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 tomcat -keystore $JDK_HOME/jre/lib/security/cacert
```

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.

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

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 Boot Docs - SSL Configuration](#)

Last update: 15/12/2019

[Login](#)

Add a comment

M ↓ MARKDOWN

ADD COMMENT

[Upvotes](#) [Newest](#) [Oldest](#)



Sean Riley

0 points · 8 months ago

I really appreciate your tutorial here. I did experience a problem when calling my app from SoapUI and Postman though.

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

After much searching, I came across a StackOverflow post that offered the only thing that worked.

<https://stackoverflow.com/questions/50486314/how-to-solve-403-error-in-spring-boot-post-request/52662997#52662997>

It has to do with Cross-site request forgery or csrf. I had to change your SecurityConfig class to the following:

```
@EnableWebSecurity public class SecurityConfig extends WebSecurityConfigurerAdapter{

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

    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("*"));
        configuration.setAllowedMethods(Arrays.asList("*"));
        configuration.setAllowedHeaders(Arrays.asList("*"));
        configuration.setAllowCredentials(true);
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

0



Incredibly useful tutorial! Perfect! Thank you!!!



Hi,

Brian Loco

Hi,

Thomas Vitale MODERATOR

Hi Brian. When you have a project with Spring Security, by default all endpoints of your application are secured and a user is created. The username is "user", the password is printed out in the console. You can find more information here: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-security>

<https://www.thomasvitale.com/https-spring-boot-ssl-certificate/>

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

I am hitting `https://localhost:8443(POST)` and it works fine. But when I am hitting `http://localhost:8080(POST)`, it is hitting the server with GET request method. Any idea why POST is converting into GET? I am trying these APIs from Postman.

Thanks...



Gopinath K

0 points · 3 months ago

Very useful article. I have a question, there is SSL certificate that issued for a different web application which is a wildcard certificate. can I use this certificate to import it in my other azure VM that is running my spring boot app?



Sreekanth Tangirala

0 points · 5 days ago

I had an issue with the `keystore.jks` which was unable to read the file. I am using IntelliJ IDEA. I have created the certificate also but it did not work. I had to add a few VM params in IntelliJ as below.

1) keep `cacerts` in `C:\xx\java.cacerts` Note the VM args : `-Djavax.net.ssl.trustStore=C:\xx\java.cacerts -Djavax.net.ssl.trustStorePassword=xxxxxxx` 2)

You should pass vm args as mentioned below in 2 places a) `File>Settings>.....>Maven>Importing>Vm Options` for importer

b) `File>Settings>.....>Maven>Runner>VM Options`

Not sure the reason behind this but this works for me apart from all that is said in the blog. Thanks for the blog.



Amit Sharma

0 points · 36 days ago

Great article, to the point information. Good Work.



RK Raju Khunt

0 points · 19 days ago

Hello

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

```
org.springframework.boot.web.servlet.context.WebServerStartStopLifecycle.start(WebServerStartStopLifecycle.java:43) ~[spring-boot-2.3.3.RELEASE.jar:2.3.3.RELEASE] at org.springframework.context.support.DefaultLifecycleProcessor.doStart(DefaultLifecycleProcessor.java:182) ~[spring-context-5.2.8.RELEASE.jar:5.2.8.RELEASE] ... 19 common frames omitted Caused by: java.lang.IllegalArgumentException: standardService.connector.startFailed at org.apache.catalina.core.StandardService.addConnector(StandardService.java:231) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.springframework.boot.web.embedded.tomcat.TomcatWebServer.addPreviouslyRemovedConnectors(TomcatWebServer.java:282) ~[spring-boot-2.3.3.RELEASE.jar:2.3.3.RELEASE] at org.springframework.boot.web.embedded.tomcat.TomcatWebServer.start(TomcatWebServer.java:213) ~[spring-boot-2.3.3.RELEASE.jar:2.3.3.RELEASE] ... 21 common frames omitted Caused by: org.apache.catalina.LifecycleException: Protocol handler start failed at org.apache.catalina.connector.Connector.startInternal(Connector.java:1067) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:183) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.catalina.core.StandardService.addConnector(StandardService.java:227) ~[tomcat-embed-core-9.0.37.jar:9.0.37] ... 23 common frames omitted Caused by: java.lang.IllegalArgumentException: Alias name [ebrk] does not identify a key entry at org.apache.tomcat.util.net.AbstractJsseEndpoint.createSSLContext(AbstractJsseEndpoint.java:99) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.AbstractJsseEndpoint.initialiseSsl(AbstractJsseEndpoint.java:71) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.NioEndpoint.bind(NioEndpoint.java:216) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.AbstractEndpoint.bindWithCleanup(AbstractEndpoint.java:1141) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.AbstractEndpoint.start(AbstractEndpoint.java:1227) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.coyote.AbstractProtocol.start(AbstractProtocol.java:592) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.catalina.connector.Connector.startInternal(Connector.java:1064) ~[tomcat-embed-core-9.0.37.jar:9.0.37] ... 25 common frames omitted Caused by: java.io.IOException: Alias name [ebrk] does not identify a key entry at org.apache.tomcat.util.net.SSLUtilBase.getKeyManagers(SSLUtilBase.java:326) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.SSLUtilBase.createSSLContext(SSLUtilBase.java:246) ~[tomcat-embed-core-9.0.37.jar:9.0.37] at org.apache.tomcat.util.net.AbstractJsseEndpoint.createSSLContext(AbstractJsseEndpoint.java:97) ~[tomcat-embed-core-9.0.37.jar:9.0.37] ... 31 common frames omitted
```


Thomas Vitale – How to enable HTTPS in a Spring Boot Java application

MORE IN **SPRING**

New Book: Cloud Native Spring in Action - With Spring Boot and Kubernetes

26 Aug 2020 – 2 min read

Centralized Configuration with Spring Cloud Config

12 May 2020 – 10 min read

Spring Security and Keycloak to Secure a Spring Boot Application - A First Look

14 Nov 2019 – 11 min read

[See all 6 posts →](#)



WORDPRESS

My First 2 Years as WordPress Contributor

Exactly two years ago, at this same time, I was coming home from Milan after attending the first Italian WordPress Contributor Day. I didn't know then what it would have meant to me, but it was the beginning of something awesome.



THOMAS VITALE

7 NOV 2017 • 3 MIN READ



SPRING

Spring Data JPA using Hibernate and Java Configuration with Annotations

Spring Data JPA makes it very easy to implement JPA-based repositories. In this tutorial, I'll show you how to use it to integrate a relational database (PostgreSQL in my example) with a Spring Boot application.



THOMAS VITALE

15 MAY 2017 • 6 MIN READ

Thomas Vitale – How to enable HTTPS in a Spring Boot Java application