Enable HTTPS in Spring Boot

This weekend I answered a question about enabling HTTPS in JHipster on StackOverflow (http://stackoverflow.com/questions/29522114/how-to-add-self-signed-ssl-certificate-to-jhipster-sample-app/29582178#29582178) that caught a lot of interest on Twitter so I decided to put a short post on it with some more useful details.

JHipster is a Spring Boot application with a lot of neat features and other frameworks completely integrated. The configuration is exactly the same like any other Spring Boot application, including the SSL settings. If you are interested to get a quick introduction on JHipster, feel free to take a look at my *Start a modern Java web application with JHipster* (http://www.drissamri.be/blog/technology/starting-modern-java-project-with-jhipster/)

If you are using Spring Boot and want to enable SSL (https) for your application on the embedded Tomcat there a few short steps you will need to take.

1. **Get yourself a SSL certificate:** generate a self-signed certifcate or get one from a Certificate Authority
2. **Enable HTTPS in Spring Boot**
3. **Redirect HTTP to HTTPS** *(optional)*

Step 1: Get a SSL certificate

If you want to use SSL and serve your Spring Boot application over HTTPS you will need to get a certificate.

You have two options to get one. You can generate a self-signed certificate, which will most likely be what you'll want to do in development since it's the easiest option. This usually isn't a good option in production since it will display a warning to the user that your certificate is not trusted.

The other (production) option is to request one from a Certificate Authority. I've heard good things about SSLMate (https://sslmate.com/) to buy your certificate for a reasonable price with excellent support. There are some providers that are able to give out free certificates but usually you'll have problems down the line if you have any issues or problems (revocations).

Since we are developers, let's generate a self-signed certificate to get started quickly with development of our application. Every Java Runtime Environment (JRE) comes bundled with a certificate management utility, keytool (https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html). This can be used to generate our self-signed certificate. Let's have a look:

```
 keytool -genkey -alias tomcat
 -storetype PKCS12 -keyalg RSA -keysize 2048
 -keystore keystore.p12 -validity 3650

Enter keystore password:
 Re-enter new password:
 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=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
 [no]: yes
```

This will generate a PKCS12 keystore called keystore.p12 with your newly generate certificate in it, with certificate alias `tomcat`. You will need to reference keystore in a minute when we start to configure Spring Boot.

Step 2: Enable HTTPS in Spring Boot

By default your Spring Boot embedded Tomcat container will have HTTP on port 8080 enabled. Spring Boot lets you configure HTTP or HTTPS in the application.properties, but not both at once. If you want to enable both you will need to configure at least one programmatically. The Spring Boot reference documentation (http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#howto-configure-ssl) recommends configuring HTTPS in the application.properties since it's the more complicated than HTTP.

> *Using configuration like the example above means the application will no longer support plain HTTP connector at port 8080. Spring Boot doesn't support the configuration of both an HTTP connector and an HTTPS connector via **application.properties**. If you want to have both then you'll need to configure one of them programmatically. It's recommended to use **application.properties** to configure HTTPS as the HTTP connector is the easier of the two to configure programmatically. See the spring-boot-sample-tomcat-multi-connectors (http://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-tomcat-multi-connectors) sample project for an example.*

Funny enough despite their recommendation to configure HTTPS in the application.properties, their example does the exact opposite.

Let's configure HTTPS in the default `application.properties` file under `src/main/resources` of your Spring Boot application:

```
server.port: 8443
server.ssl.key-store: keystore.p12
server.ssl.key-store-password: mypassword
server.ssl.keyStoreType: PKCS12
server.ssl.keyAlias: tomcat
```

That's all you need to do to make your application accessible over HTTPS on `https://localhost:8443`, pretty easy right?

Step 3: Redirect HTTP to HTTPS (optional)

In some cases it might be a good idea to make your application accessible over HTTP too, but redirect all traffic to HTTPS. To achieve this we'll need to add a second Tomcat connector, but currently it is not possible to configure two connector in the application.properties like mentioned before. Because of this we'll add the HTTP connector programmatically and make sure it redirects all traffic to our HTTPS connector.

For this we will need to add the **TomcatEmbeddedServletContainerFactory** bean to one of our **@Configuration** classes.

```
@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(initiateHttpConnector());
  return tomcat;
}

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

  return connector;
}
```

ContainerConfiguration.java view raw (https://gist.github.com/drissamri/8def1ce9322caab47e8e/raw/f1ae6ff109e71944afa244a6e8b7004da96bf0a5/ContainerConfiguration.java) (https://gist.github.com/drissamri/8def1ce9322caab47e8e#file-containerconfiguration-java) hosted with ♡ by GitHub (https://github.com)

That's all you need to do to make sure your application is always used over HTTPS!