Enabling HTTPS

jim stafford

Fall 2022 v2020-07-17: Built: 2022-12-07 06:16 EST

Table of Contents

Chapter 1. Introduction

In all the examples to date (and likely forward), we have been using the HTTP protocol. This has been very easy option to use, but I likely do not have to tell you that straight HTTP is **NOT secure** for use and especially **NOT appropriate** for use with credentials or any other authenticated information.

Hypertext Transfer Protocol Secure (HTTPS) — with trusted certificates — is the secure way to communicate using APIs in modern environments. We still will want the option of simple HTTP in development and most deployment environments provide an external HTTPS proxy that can take care of secure communications with the external clients. However, it will be good to take a short look at how we can enable HTTPS directly within our Spring Boot application.

1.1. Goals

You will learn:

- the basis of how HTTPS forms trusted, private communications
- the difference between self-signed certificates and those signed by a trusted authority
- how to enable HTTPS/TLS within our Spring Boot application

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. define the purpose of a public certificate and private key
2. generate a self-signed certificate for demonstration use
3. enable HTTPS/TLS within Spring Boot
4. optionally implement an HTTP to HTTPS redirect
5. implement a Maven Failsafe integration test using HTTPS

Chapter 2. HTTP Access

I have cloned the "noauth-security-example" to form the "https-hello-example" and left most of the insides intact. You may remember the ability to execute the following authenticated command.

Example Successful Authentication

```
$ curl -v -X GET http://localhost:8080/api/authn/hello?name=jim -u "user:password" ①
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpwYXNzd29yZA== ②
>
< HTTP/1.1 200
hello, jim
```

① curl supports credentials with `-u` option

② curl Base64 encodes credentials and adds `Authorization` header

We get rejected when no valid credentials are supplied.

Example Rejection of Anonymous

```
$ curl -X GET http://localhost:8080/api/authn/hello?name=jim
{"timestamp":"2020-07-18T14:43:39.670+00:00","status":401,
 "error":"Unauthorized","message":"Unauthorized","path":"/api/authn/hello"}
```

It works as we remember it, but the issue is that our slightly encoded (`dXNlcjpwYXNzd29yZA==`), plaintext password was issued in the clear. We can fix that by enabling HTTPS.

Chapter 3. HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is an extension of HTTP encrypted with Transport Layer Security (TLS) for secure communication between endpoints — offering privacy and integrity (i.e., hidden and unmodified). HTTPS formerly offered encryption with the now deprecated Secure Sockets Layer (SSL). Although the SSL name still sticks around, TLS is only supported today. [1] [2]

3.1. HTTPS/TLS

At the heart of HTTPS/TLS are X.509 certificates and the Public Key Infrastructure (PKI). Public keys are made available to describe the owner (subject), the issuer, and digital signatures that prove the contents have not been modified. If the receiver can verify the certificate and trusts the issuer — the communication can continue. [3]

With HTTPS/TLS, there is one-way and two-way option with one-way being the most common. In one-way TLS — only the server contains a certificate and the client is anonymous at the network level. Communications can continue if the client trusts the certificate presented by the server. In two-way TLS, the client also presents a signed certificate that can identify them to the server and form two-way authentication at the network level. Two-way is very secure but not as common except in closed environments (e.g., server-to-server environments with fixed/controlled communications). We will stick to one-way TLS in this lecture.

3.2. Keystores

A keystore is repository of security certificates - both private and public keys. There are two primary types: Public Key Cryptographic Standards (PKCS12) and Java KeyStore (JKS). PKCS12 is an industry standard and JKS is specific to Java. [4] They both have the ability to store multiple certificates and use an alias to identify them. Both use password protection.

There are typically two uses for keystores: your identity (keystore) and the identity of certificates you trust (truststore). The former is used by servers and must be well protected. The later is necessary for clients. The truststore can be shared but its contents need to be trusted.

3.3. Tools

There are two primary tools when working with certificates and keystores: keytool and openssl.

keytool comes with the JDK and can easily generate and manage certificates for Java applications. Keytool originally used the JKS format but since Java 9 switched over to PKCS12 format.

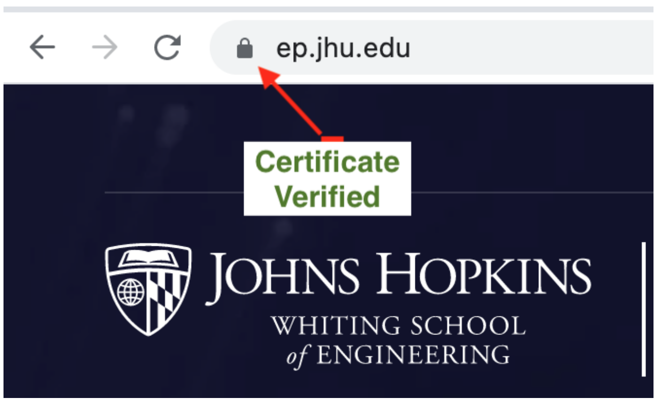
openssl is a standard, open source tool that is not specific to any environment. It is commonly used to generate and convert to/from all types of certificates/keys.

3.4. Self Signed Certificates

The words "trust" and "verify" were used a lot in the paragraphs above when describing certificates.

When we visit various web sites — that locked icon next to the "https" URL indicates the certificate presented by the server was verified and came from a trusted source.

Verified Server Certificate



Trusted certificates come from sources that are pre-registered in the browsers and Java JRE truststore and are obtained through purchase.

We can generate self-signed certificates that are not immediately trusted until we either ignore checks or enter them into our local browsers and/or truststore(s).

[1] *"HTTPS"*, Wikipedia

[2] *"Transport Layer Security"*, Wikipedia

[3] *"Public key certificate"*, Wikipedia

[4] *"Spring Boot HTTPS"*, ZetCode, July 2020

Chapter 4. Enable HTTPS/TLS in Spring Boot

To enable HTTPS/TLS in Spring Boot — we must do the following

1. obtain a digital certificate - we will generate a self-signed certificate without purchase or much fanfare
2. add TLS properties to the application
3. optionally add an HTTP to HTTPS redirect - useful in cases where clients forget to set the protocol to `https://` and use `http://` or use the wrong port number.

4.1. Generate Self-signed Certificate

The following example shows the creation of a self-signed certificate using keytool. Refer to the keytool reference page for details on the options. The following Java Keytool page provides examples of several use cases. I kept the values of the certificate extremely basic since there is little chance we will ever use this in a trusted environment.

Generate Self-signed RSA Certificate

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \①
-keystore keystore.p12  -alias https-hello \②
-storepass password
What is your first and last name?
  [Unknown]:  localhost
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
```

① specifying a valid date 10 years in the future

② assigning the alias `https-hello` to what is generated in the keystore

4.2. Place Keystore in Reference-able Location

The keytool command output a keystore file called `keystore.p12`. I placed that in the resources area of the application — which will be can be referenced at runtime using a classpath reference.

Place Keystore in Location to be Referenced

```
$ tree src/main/resources/
src/main/resources/
|-- application.properties
`-- keystore.p12
```

Incremental Learning Example Only: Don’t use Source Tree for Certs

This example is trying hard to be simple and using a classpath for the keystore to be portable. You should already know how to convert the classpath reference to a file or other reference to keep sensitive information protected and away from the code base. **Do not** store credentials or other sensitive information in the `src` tree in a real application as the `src` tree will be stored in CM.

4.3. Add TLS properties

The following shows a minimal set of properties needed to enable TLS. [1]

Example TLS properties

```
server.port=8443①
server.ssl.enabled=true
server.ssl.key-store=classpath:keystore.p12②
server.ssl.key-store-password=password③
server.ssl.key-alias=https-hello
```

① using an alternate port - optional

② referencing keystore in the classpath — could also use a file reference

③ think twice before placing credentials in a properties file

Do not place credentials in CM system

Do not place real credentials in files checked into CM. Have them resolved from a source provided at runtime.

Note the presence of the legacy "ssl" term in the property name even though "ssl" is deprecated and we are technically setting up "tls".

[1] *"HTTPS using Self-Signed Certificate in Spring Boot"*, Baeldung, June 2020

Chapter 5. Untrusted Certificate Error

Once we restart the server, we should be able to connect using HTTPS and port 8443. However, there will be a trust error. The following shows the error from curl.

Untrusted Certificate Error

```
$ curl https://localhost:8443/api/authn/hello?name=jim -u user:password
curl: (60) SSL certificate problem: self signed certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

Chapter 6. Accept Self-signed Certificates

curl and older browsers have the ability to accept self-signed certificates either by ignoring their inconsistencies or adding them to their truststore.

The following is an example of curl's `-insecure` option (`-k` abbreviation) that will allow us to communicate with a server presenting a certificate that fails validation.

Enable -insecure Option

```
$ curl -kv -X GET https://localhost:8443/api/authn/hello?name=jim -u "user:password"
* Connected to localhost (::1) port 8443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server did not agree to a protocol
* Server certificate:
*  subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  start date: Jul 18 13:46:35 2020 GMT
*  expire date: Jul 16 13:46:35 2030 GMT
*  issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
* Server auth using Basic with user 'user'
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8443
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
hello, jim
```

6.1. Optional Redirect

To handle clients that may address our application using the wrong protocol or port number — we can optionally setup a redirect to go from the common port to the TLS port. The following snippet was taken directly from a ZetCode article but I have seen this near exact snippet many times elsewhere.

HTTP:8080 ⇒ HTTPS:8443 Redirect

```
@Bean
public ServletWebServerFactory servletContainer() {
    var 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(redirectConnector());
    return tomcat;
}

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

6.2. HTTP:8080 ⇒ HTTPS:8443 Redirect Example

With the optional redirect in place, the following shows an example of the client being sent from their original `http://localhost:8080` call to `https://localhost:8443`.

```
$ curl -kv -X GET http://localhost:8080/api/authn/hello?name=jim -u "user:password"
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 302 ①
< Location: https://localhost:8443/api/authn/hello?name=jim ②
```

① HTTP 302/Redirect Returned

② Location header provides the full URL to invoke — including the protocol

6.3. Follow Redirects

Browsers automatically follow redirects and we can get curl to automatically follow redirects by adding the `--location` option (or `-L` abbreviated). The following command snippet shows curl being requested to connect to an HTTP port , receiving a 302/Redirect, and then completing the original command using the URL provided in the `Location` header of the redirect.

Example curl Follow Redirect

```
$ curl -kvL -X GET http://localhost:8080/api/authn/hello?name=jim -u "user:password"
①
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 302
< Location: https://localhost:8443/api/authn/hello?name=jim
<
* Issue another request to this URL: 'https://localhost:8443/api/authn/hello?name=jim'
...
* Server certificate:
*  subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  start date: Jul 18 13:46:35 2020 GMT
*  expire date: Jul 16 13:46:35 2030 GMT
*  issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  SSL certificate verify result: self signed certificate (18), continuing anyway.
> GET /api/authn/hello?name=jim HTTP/1.1
> Host: localhost:8443
> Authorization: Basic dXNlcjpwYXNzd29yZA==
>
< HTTP/1.1 200
hello, jim
```

① `-L` (--location) redirect option causes curl to follow the 302/Redirect response

6.4. Caution About Redirects

One note of caution I will give about redirects is the tendency for Intellij to leave orphan processes which seems to get worse with the Tomcat redirect in place. Since our targeted interfaces are for API clients — which should have a documented source of how to communicate with our server — there should be no need for the redirect. The redirect is primarily valuable for interfaces that switch between HTTP and HTTPS we are either all HTTP or all HTTPS and no need to be eclectic.

Eliminating the optional redirect also eliminates the need for the redirect code and reduces our required steps to obtaining the certificate and setting a few simple properties.

Chapter 7. Maven Integration Test

Since we are getting close to real deployments to test environments and we have hit unit integration tests pretty hard, I wanted to demonstrate a test of the HTTPS configuration using a true integration test and the Maven Failsafe plugin.

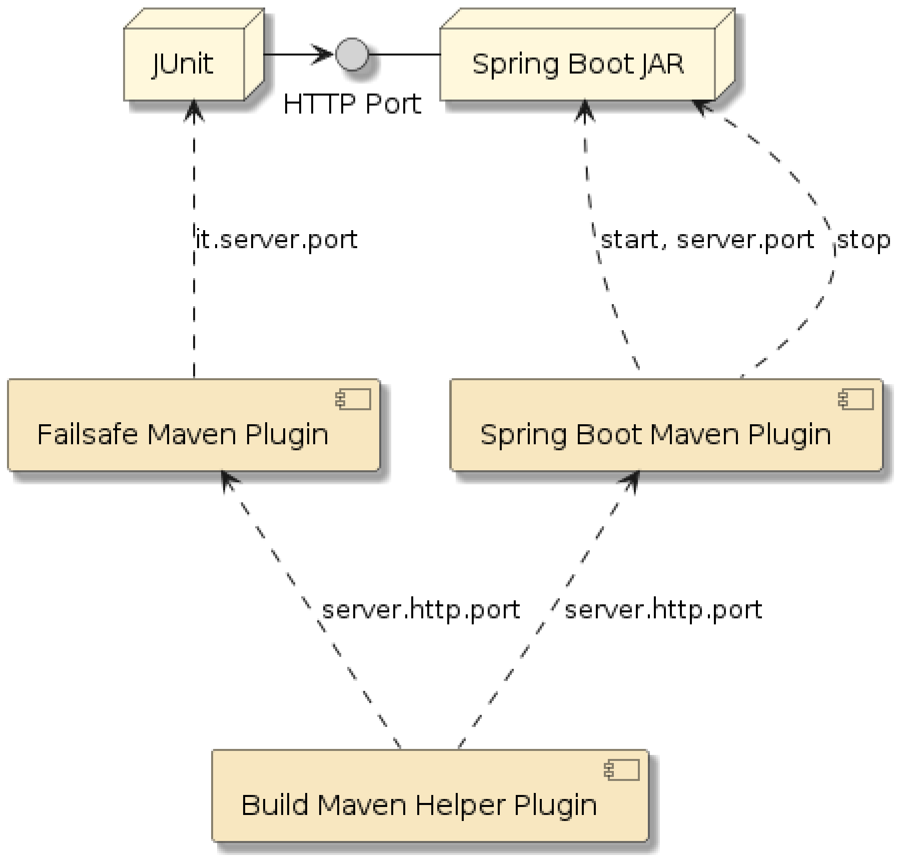


Figure 1. Maven Failsafe Integration Test

A Maven Failsafe integration test is very similar to the other Web API unit integration tests you are use to seeing in this course. The primary difference is that there are no server-side components in the JUnit Spring context. All the server-side components are in a separate executable. The following diagram shows the participants that directly help to implement the integration test.

This will be accomplished with the aid of the Maven Failsafe, Spring Boot, and Build Maven Helper plugins.

With that said, we will still want to be able to execute simple integration tests like this within the IDE. Therefore expect some setup aspects to support both IDE-based and Maven-based integration testing setup in the paragraphs that follow.

7.1. Maven Integration Test Phases

Maven executes integration tests using four (4) phases

- pre-integration-test - start resources
- integration-test - execute tests
- post-integration-test - stop resources
- verify - evaluate/assert test results

We will make use of three (3) plugins to perform that work within Maven. Each is also accompanied by *steps to mimic the Maven capability on a small scale with the IDE*:

- `spring-boot-maven-plugin` - used to start and stop the server-side Spring Boot process

    - *(use IDE, "java -jar" command, or "mvn springboot:run" command to manually start, restart, and stop the server)*
- `build-maven-helper-plugin` - used to allocate a random network port for server
    - *(within the IDE you will use a property file that uses a well-known port# used one-at-a-time)*
- `maven-failsafe-plugin` - used to run the JUnit JVM with the tests — passing in the port# — and verifying/asserting the results.
    - *(use IDE to run test following server-startup)*

7.2. Spring Boot Maven Plugin

The `spring-boot-maven-plugin` will be configured with at least 2 executions to support Maven integration testing.

spring-boot-maven-plugin Shell

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    ...
  </executions>
</plugin>
```

7.2.1. SpringBoot: pre-integration-test Phase (start)

The following snippet shows the plugin being used to `start` the server in the background (versus a blocking `run`). The execution is configured to supply a Spring Boot `server.port` property with the HTTP port to use. We will use a separate plugin to generate the port number and have that assigned to the Maven `server.http.port` property at build time. The client-side Spring Boot Test will also need this port value for the client(s) in the integration tests.

SpringBoot pre-integration-test Execution

```
<execution>
  <id>pre-integration-test</id> ①
  <phase>pre-integration-test</phase> ②
  <goals>
    <goal>start</goal> ③
  </goals>
  <configuration>
    <skip>${skipITs}</skip> ④
    <arguments> ⑤
      <argument>--server.port=${server.http.port}</argument>
    </arguments>
  </configuration>
</execution>
```

① each execution must have a unique ID

② this execution will be tied to the `pre-integration-test` phase

③ this execution will `start` the server in the background

④ `-DskipITs=true` will deactivate this execution

⑤ `--server.port` is being assigned at runtime and used by server for HTTP/S listen port

Failsafe is overriding the fixed value from `application-https.properties`.

src/main/resources/application-https.properties

```
server.port=8443
```

The above execution phase has the same impact as if we launched the JAR manually with `spring.profiles.active` and whether `server.port` was supplied on the command. This allows multiple IT tests to run concurrently without colliding on network port number. It also permits the use of a well-known/fixed value for use with IDE-based testing.

Manual Start Commands

```
$ java -jar target/https-hello-example-*-SNAPSHOT.jar --spring.profiles.active=https
Tomcat started on port(s): 8443 (https) with context path ''①

$ java -jar target/https-hello-example-*-SNAPSHOT.jar --spring.profiles.active=https
--server.port=7712 ②
Tomcat started on port(s): 7712 (http) with context path '' ②
```

① Spring Boot using well-known/fixed port# supplied in `application-https.properties`

② Spring Boot using runtime `server.port` property to override port to use

7.2.2. SpringBoot: post-integration-test Phase (stop)

The following snippet shows the Spring Boot plugin being used to `stop` a running server.

```
<execution>
  <id>post-integration-test</id> ①
  <phase>post-integration-test</phase> ②
  <goals>
    <goal>stop</goal> ③
  </goals>
  <configuration>
    <skip>${skipITs}</skip> ④
  </configuration>
</execution>
```

① each execution must have a unique ID

② this execution will be tied to the `post-integration-test` phase

③ this execution will stop the running server

④ -DskipITs=true will deactivate this execution

> *skipITs support*
>
> Most plugins offer a skip option to bypass a configured execution and sometimes map that to a Maven property that can be expressed on the command line. Failsafe maps their property to skipITs. By mapping the Maven skipITs property to the plugin's skip configuration element, we can inform related plugins to do nothing. This allows one to run the Maven install phase without requiring integration tests to run and pass.

7.3. Build Helper Maven Plugin

The build-helper-maven-plugin contains various utilities that are helpful to create a repeatable, portable build. We are using the reserve-network-port goal to select an available HTTP port at build-time. The allocated port number is assigned to the Maven server.http.port property. This was shown picked up by the Spring Boot Maven Plugin earlier.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>reserve-network-port</id>
      <phase>process-resources</phase> ①
      <goals>
        <goal>reserve-network-port</goal> ②
      </goals>
      <configuration>
        <portNames> ③
          <portName>server.http.port</portName>
        </portNames>
      </configuration>
    </execution>
  </executions>
</plugin>
```

① execute during the process-resources Maven phase — which is well before pre-integration-test

② execute the reserve-network-port goal of the plugin

③ assigned the identified port to the Maven server.http.port property

7.4. Failsafe Plugin

The Failsafe plugin has some default behavior, but once we start configuring it — we need to restate much of what it would have done automatically for us.

maven-failsafe-plugin Declaration

```
  <plugin>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        ...
    </executions>
  </plugin>
```

7.4.1. Failsafe: integration-test Phase

In the snippet below, we are primarily configuring Failsafe to launch the JUnit test with an `it.server.port` property. This will be read in by the `ServerConfig` `@ConfigurationProperties` class

integration-test Phase

```
<execution>
    <id>integration-test</id>
    <phase>integration-test</phase> ①
    <goals> ①
      <goal>integration-test</goal>
    </goals>
    <configuration>
      <includes> ①
        <include>**/*IT.java</include>
      </includes>
      <systemPropertyVariables> ②
        <it.server.port>${server.http.port}</it.server.port>
      </systemPropertyVariables>
      <additionalClasspathElements> ③
        <additionalClasspathElement>
${basedir}/target/classes</additionalClasspathElement>
      </additionalClasspathElements>
      <useModulePath>false</useModulePath> ④
    </configuration>
</execution>
```

① re-state some Failsafe default relative to phase, goal, includes

② add a `-Dit.server.port=${server.http.port}` system property to the execution

③ adding `target/classes` to classpath when JUnit test using classes from "src/main"

④ turning off some Java 9 module features

Full disclosure. I need to refresh my memory on exactly why default `additionalClasspathElements` and `useModulePath` did not work here.

7.4.2. Failsafe: verify Phase

The snippet below shows the final phase for Failsafe. After the integration resources have been taken down, the only thing left is to assert the results. This pass/fail assertion is delayed a few phases so that the build does not fail while integration resources are still running.

verify Phase

```
<execution>
  <id>verify</id>
  <phase>verify</phase>
  <goals>
    <goal>verify</goal>
  </goals>
</execution>
```

7.5. JUnit @SpringBootTest

With the Maven setup complete — that brings us back to a familiar looking JUnit test and `@SpringBootTest` However, there is no application or server-side resources in the Spring context,

```
@SpringBootTest(classes={ClientTestConfiguration.class}, ①
        webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
@ActiveProfiles({"its"}) ③
public class HttpsRestTemplateIT {
    @Autowired ④
    private RestTemplate authnUser;
    @Autowired ⑤
    private URI authnUrl;
```

① no application class in this integration test. Everything is server-side.

② have only a client-side web environment. No listen port necessary

③ activate `its` profile for scope of test case

④ inject `RestTemplate` configured with user credentials that can authenticate

⑤ inject URL to endpoint test will be calling

Since we have no `RANDOM_PORT` and a late `@LocalServerPort` injection, we can move `ServerConfig` to the configuration class and inject the baseURL product.

7.6. ClientTestConfiguration

This trimmed down `@Configuration` class is all that is needed for JUnit test to be a client of a remote process. The `@SpringBootTest` will demand to have a `@SpringBootConfiguration` and we technically do not have the `@SpringBootApplication` during the test.

```
@SpringBootConfiguration(proxyBeanMethods = false)
@EnableAutoConfiguration
@Slf4j
```

```
public class ClientTestConfiguration {
    ...
    @Bean
    @ConfigurationProperties("it.server")
    public ServerConfig itServerConfig() { ...
    @Bean
    public URI authnUrl(ServerConfig serverConfig) { ...①
    @Bean
    public RestTemplate authnUser(RestTemplateBuilder builder,...②
    ...
```

① baseUrl of remote server

② `RestTemplate` with authentication and HTTPS filters applied

7.7. application-its.properties

The following snippet shows the `its` profile-specific configuration file, complete with

- `it.server.scheme` (https)
- `it.server.port` (8443)
- trustStore properties pointing at the server-side identity keystore.

application-its.properties

```
it.server.scheme=https
#must match self-signed values in application-https.properties
it.server.trust-store=keystore.p12
it.server.trust-store-password=password
#used in IDE, overridden from command line during failsafe tests
it.server.port=8443 ①
```

① default port when working in IDE. Overridden by command line properties by Failsafe

The keystore/truststore used in this example is for learning and testing. Do not store operational certs in the source tree. Those files end up in the searchable CM system and the JARs with the certs end up in a Nexus repository.

7.8. username/password Credentials

The following shows the username and password credentials being injected using values from the properties. In this test's case — they should always be provided. Therefore, no default String is defined.

username/password Credentials

```
public class ClientTestConfiguration {
    @Value("${spring.security.user.name}")
    private String username;
    @Value("${spring.security.user.password}")
    private String password;
```

7.9. ServerConfig

The following shows the primary purpose for `ServerConfig` as a `@ConfigurationProperties` class with flexible prefix. In this particular case it is being instructed to read in all properties with prefix "it.server" and instantiate a ServerConfig.

```
@Bean
@ConfigurationProperties("it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}
```

From the property file earlier, you will notice that the URL scheme will be "https" and the port will be "8443" or whatever property override is supplied on the command line. The resulting value will be injected into the `@Configuration` class.

7.10. authnUrl URI

Since we don't have the late-injected `@LocalServerPort` for the web-server and our `ServerConfig` is now all property-based, we can now delegate baseUrls to injectable beans. The following shows the `baseUrl` from `ServerConfig` being used to construct a URL for "api/authn/hello".

Building baseUrl from Injected ServerConfig

```
@Bean
public URI authnUrl(ServerConfig serverConfig) {
    URI baseUrl = serverConfig.getBaseUrl();
    return UriComponentsBuilder.fromUri(baseUrl).path("/api/authn/hello").build()
.toUri();
}
```

7.11. authUser RestTemplate

By no surprise, `authnUser()` is adding a `BasicAuthenticationInterceptor` containing the injected username and password to a new `RestTemplate` for use in the test. The injected `ClientHttpRequestFactory` will take care of the HTTP/HTTPS details.

authnUser RestTemplate

```
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                              ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
            //used to read the streams twice -- so we can use the logging filter below
            ()->new BufferingClientHttpRequestFactory(requestFactory))
            .interceptors(new BasicAuthenticationInterceptor(username, password),
                    new RestTemplateLoggingFilter())
            .build();
    return restTemplate;
}
```

7.12. HTTPS ClientHttpRequestFactory

The HTTPS-based ClientHttpRequestFactory is built by following some excellent instructions and short article provided by Geoff Bourne. The following intermediate factory relies on the ability to construct an SSLContext.

```
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.http.client.ClientHttpRequestFactory;
/*
TLS configuration based on great/short article by Geoff Bourne
https://medium.com/@itzgeoff/using-a-custom-trust-store-with-resttemplate-in-spring-
boot-77b18f6a5c39
 */
@Bean
public ClientHttpRequestFactory httpsRequestFactory(SSLContext sslContext,
        ServerConfig serverConfig) {
    HttpClient httpsClient = HttpClientBuilder.create()
            .setSSLContext(serverConfig.isHttps() ? sslContext : null)
            .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

7.13. SSL Context

The SSLContext `@Bean` factory locates and loads the trustStore based on the properties within `ServerConfig`. If found, it uses the `SSLContextBuilder` from the apache HTTP libraries to create a `SSLContext`.

```
import org.apache.http.ssl.SSLContextBuilder;
import javax.net.ssl.SSLContext;
...
@Bean
public SSLContext sslContext(ServerConfig serverConfig)  {
    try {
        URL trustStoreUrl = null;
        if (serverConfig.getTrustStore()!=null) {
            trustStoreUrl = HttpsExampleApp.class.getResource("/" + serverConfig
.getTrustStore());
            if (null==trustStoreUrl) {
                throw new IllegalStateException("unable to locate truststore:/" +
serverConfig.getTrustStore());
            }
        }
        SSLContextBuilder builder = SSLContextBuilder.create()
                .setProtocol("TLSv1.2");
        if (trustStoreUrl!=null) {
            builder.loadTrustMaterial(trustStoreUrl, serverConfig
.getTrustStorePassword());
        }
        return builder.build();
    } catch (Exception ex) {
        throw new IllegalStateException("unable to establish SSL context", ex);
    }
}
```

7.14. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTPS/Authn-enabled `RestTemplate` and baseUrl injected. From here it is just a normal test, but activity is remote on the server side.

```
public class HttpsRestTemplateIT {
    @Autowired ①
    private RestTemplate authnUser;
    @Autowired ②
    private URI authnUrl;

    @Test
    public void user_can_call_authenticated() {
        //given a URL to an endpoint that accepts only authenticated calls
        URI url = UriComponentsBuilder.fromUri(authnUrl)
            .queryParam("name", "jim").build().toUri();

        //when called with an authenticated identity
        ResponseEntity<String> response = authnUser.getForEntity(url, String.class);

        //then expected results returned
        then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        then(response.getBody()).isEqualTo("hello, jim");
    }
}
```

① `RestTemplate` with authentication and HTTPS aspects addressed using filters

② `authnUrl` built from `ServerConfig` and injected into test

7.15. Maven Verify

When we execute `mvn verify` (with option to add `clean`), we see the port being determined and assigned to the `server.http.port` Maven property.

Starting Maven Build

```
$ mvn verify
...
- build-helper-maven-plugin:3.1.0:reserve-network-port (reserve-network-port)
Reserved port 52024 for server.http.port ①
...②
- maven-surefire-plugin:3.0.0-M5:test (default-test) @ https-hello-example ---
...
- spring-boot-maven-plugin:2.4.2:repackage (package) @ https-hello-example ---
Replacing main artifact with repackaged archive
③
- spring-boot-maven-plugin:2.4.2:start (pre-integration-test) @ https-hello-example
---
```

① the port identified by build-helper-maven-plugin as `52024`

② Surefire tests firing at an earlier `test` phase

③ server starting in the `pre-integration-test` phase

7.15.1. Server Output

When the server starts, we can see that the `https` profile is activate and Tomcat was assigned the `52024` port value from the build.

Server Output

```
HttpsExampleApp#logStartupProfileInfo:664 The following profiles are active: https ①
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 52024 (https) ②
TomcatWebServer#start:220 Tomcat started on port(s): 52024 (https) with context path
'' ②
```

① `https` profile has been activated on the server

② server HTTP(S) port assigned to `52024`

7.15.2. JUnit Client Output

When the JUnit client starts, we can see that SSL is enabled and the baseURL contains `https` and the dynamically assigned port `52024`.

JUnit Client Output

```
HttpsRestTemplateIT#logStartupProfileInfo:664 The following profiles are active: its
①
ClientTestConfiguration#authnUrl:64 baseUrl=https://localhost:52024 ②
```

```
ClientTestConfiguration#authnUser:107 enabling SSL requests ③
```

① `its` profile is active in JUnit client

② baseUrl is assigned `https` and port `52024`, with the latter dynamically assigned at build-time

③ SSL has been enabled on client

7.15.3. JUnit Test DEBUG

There is some DEBUG logged during the activity of the test(s).

Message Exchange

```
GET /api/authn/hello?name=jim, headers=[accept:"text/plain, application/json,
application/xml, application/*+json, text/xml, application/*+xml, */*",
authorization:"Basic dXNlcjpwYXNzd29yZA==", host:"localhost:52024", connection:"Keep-
Alive", user-agent:"masked", accept-encoding:"gzip,deflate"]]
```

7.15.4. Failsafe Test Results

Test results are reported.

Failsafe Test Results

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.409 s - in
info.ejava.examples.svc.https.HttpsRestTemplateIT
[INFO] Results:
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

7.15.5. Server is Stopped

Server is stopped.

Server is Stopped

```
[INFO] --- spring-boot-maven-plugin:2.4.2:stop (post-integration-test)
[INFO] Stopping application...
15:29:42.178 RMI TCP Connection(4)-127.0.0.1  INFO
XBeanRegistrar$SpringApplicationAdmin#shutdown:159 Application shutdown requested.
```

7.15.6. Test Results Asserted

Test results are asserted.

Overall Test Results

```
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (verify) @ https-hello-example ---
[INFO] ------------------------------------------------------------------------
```

```
[INFO] BUILD SUCCESS
```

Chapter 8. Summary

In this module we learned:

- the basis of how HTTPS forms trusted, private communications
- how to generate a self-signed certificate for demonstration use
- how to enable HTTPS/TLS within our Spring Boot application
- how to add an optional redirect and why it may not be necessary
- how to setup and run a Maven Failsafe Integration Test