Enabling HTTPS

jim stafford

Fall 2024 v2020-07-17: Built: 2024-11-19 21:37 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

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 Base64 encoded (`dXNlcjpwYXNzd29yZA==`), plaintext credentials were issued in the clear.

Base64 Encoding is not Encryption

```
> Authorization: Basic dXNlcjpwYXNzd29yZA== ①
...
$ echo -n dXNlcjpwYXNzd29yZA== | base64 -d && echo ②
user:password
```

① credentials are base64 encoded

② base64 encoded credentials can be easily decoded

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 latter 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 websites — that locked icon next to the "https" URL indicates the certificate presented by the server was verified and came from a trusted source. Only a server with the associated private key could have generated an inspected value that matched the well-known public key.

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. However, that reference does not include coverage of the now necessary `-ext` x509 option for `-genkeypair` and Subject Alternative Name (SAN). The specification of the SAN has to be added to your command.

I kept the values of the certificate extremely basic since there is little chance we will ever use this in a trusted environment. Some key points:

- we request that an RSA certificate be generated
- the certificate should be valid for 10 years
- stored in a keystore ("keystore.p12") using alias ("https-hello")
- tracked by a DN ("CN=localhost,...")
- supplied with a SAN X.509 extension to include a DNS ("localhost") and IP ("127.0.0.1") value that the server can legally respond from to be accepted. SAN is a relatively new requirement and takes the place of validating the hostname using Common Name value ("CN=localhost") of the DN.

Generate Self-signed RSA Certificate

```
$ keytool -genkeypair -keyalg RSA -keysize 2048 -validity 3650 \①
-keystore keystore.p12 -storepass password -alias https-hello \②
-ext "SAN:c=DNS:localhost,IP:127.0.0.1" \③
-dname "CN=localhost,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown" ④
```

① specifying a valid date 10 years (3650 days) in the future

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

③ assigning Subject Alternative Names that host can legally answer with

④ assigning a Distinguished Name to uniquely track the certificate

Listing Contents of Keystore

```
$ keytool --list -keystore ./keystore.p12 -storepass password -v
```

```
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: https-hello
Creation date: Sep 8, 2024
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 6e1836ff31a4e3fb
Valid from: Sun Sep 08 06:32:35 EDT 2024 until: Wed Sep 06 06:32:35 EDT 2034
Certificate fingerprints:
         SHA1: A8:2A:93:56:E4:A0:51:CA:21:32:AF:94:F4:FC:05:10:A7:37:47:CE
         SHA256:
50:22:90:94:64:6E:27:B3:AA:27:53:A9:F4:D5:AD:39:D1:E2:97:D0:33:CF:7B:C1:FB:1E:D0:3E:CE
:FB:24:89
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.17 Criticality=true
SubjectAlternativeName [
  DNSName: localhost
  IPAddress: 127.0.0.1
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: C5 BD 54 7D 57 BB 6A B4   8D B0 EC B0 D4 98 36 86  ..T.W.j.......6.
0010: FD 38 81 C3                                        .8..
]
```

I was alerted to the X.509 feature as a new requirement/solution via the following Stackoverflow post. Prior versions of Spring Boot networking dependencies did not have that extension as a resolution requirement.

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.
```

5.1. Option: Supply Trusted Certificates

One option is to construct a file of trusted certificates that includes our self-signed certificate. Curl requires this to be in PEM format. We can build that with the openssl pkcs12 command.

Building Example Trusted Certificate Authority

```
openssl pkcs12 -in ./src/main/resources/keystore.p12 -passin pass:password \
-out ./target/certs.pem \
-clcerts -nokeys -nodes ①
```

① certs only, no private keys, not password protected

PEM File is Text File with Base64 Encoded Certificate Information

```
$ cat target/certs.pem
Bag Attributes
    friendlyName: https-hello
    localKeyID: 54 69 6D 65 20 31 37 32 35 37 39 31 35 35 35 32 31 34
subject=C=Unknown, ST=Unknown, L=Unknown, O=Unknown, OU=Unknown, CN=localhost
issuer=C=Unknown, ST=Unknown, L=Unknown, O=Unknown, OU=Unknown, CN=localhost
-----BEGIN CERTIFICATE-----
MIIDnjCCAoagAwIBAgIIbhg2/zGk4/swDQYJKoZIhvcNAQELBQAwbjEQMA4GA1UE
...
yy1zz1mKV0tzKdtW+PteNVDc
-----END CERTIFICATE-----
```

Adding the generated file as our CA trust source with `--cacert`, we are able to get curl to accept the certificate.

Curl Accepting the Server Certificate After Added to Trusted Sources

```
$ curl -v -X GET https://localhost:8443/api/authn/hello?name=jim -u user:password
```

```
--cacert ./target/certs.pem && echo
...
*  CAfile: ./target/trust.pem
...
* Server certificate:
*  subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
*  SSL certificate verify ok.
...
< HTTP/1.1 200
hello, jim
```

5.2. Option: Accept Self-signed Certificates

curl and 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
```

Chapter 6. Optional Redirect

To handle clients that may address our application using the wrong protocol or port number — we can optionally set up 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() {
    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(redirectConnector());
    return tomcat;
}

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

6.1. 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

6.2. Follow Redirects

Browsers automatically follow redirects, and we can get curl to automatically follow redirects by adding the `--location` option (or `-L` abbreviated). However, security reasons, this will not present the credentials to the redirected location.

Adding `--location-trusted` will perform the `--location` redirect and present the supplied credentials to the new location — with security risk that we did not know the location ahead of time.

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 -kv -X GET http://localhost:8080/api/authn/hello?name=jim -u user:password
--location-trusted ①
...
* Server auth using Basic with user 'user'
> 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
...
*  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 `--location-trusted` redirect option causes curl to follow the 302/Redirect response and present credentials

It is a security risk to (a) send credentials using an unencrypted HTTP connection and then (b) have the credentials issued to a location issued by the server. This was just an example of how to implement server-side redirects and perform a simple test/demonstration. This was not an example of how to securely implement the client-side.

6.3. 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.

Chapter 7. Maven Unit Integration Test

The approach to unit integration testing the application with HTTPS enabled is very similar to non-HTTPS techniques. The changes will be primarily in enhancing the `ClientHttpRequestFactory` that we have been defaulting to a simple HTTP version to date.

Former ClientHttpRequestFactory Bean Factory being Replaced

```
@Bean
ClientHttpRequestFactory requestFactory() {
    return new SimpleClientHttpRequestFactory();
}
```

We will be enhancing the `ClientHttpRequestFactory` `@Bean` factory with an optional SSLFactory to control what gets created. `@Autowired(required=false)` is used to make the injected component optional/nullable.

Enhanced ClientHttpRequestFactory Bean Factory with Optional SSLFactory

```
@Bean @Lazy
public ClientHttpRequestFactory httpsRequestFactory(
        @Autowired(required = false) SSLFactory sslFactory) { ...
```

We will support that method with a conditional `SSLFactory` `@Bean` factory that will be activated when `it.server.trust-store` property is supplied and non-empty. `@ConditionalOnExpression` is used with a Spring Expression Language (SpEL) expression designed to fail if the property is not provided or provided without a value.

Conditional SSLFactory

```
@Bean @Lazy
@ConditionalOnExpression(
        "!T(org.springframework.util.StringUtils).isEmpty('${it.server.trust-
store:}')")
public SSLFactory sslFactory(ResourceLoader resourceLoader, ServerConfig serverConfig)
    throws IOException { ...
```

7.1. JUnit @SpringBootTest

The `@SpringBootTest` will bring in the proper configuration and activate:

- the application's `https` profile to activate HTTPS on the server-side
- the tests `ntest` profile to add any properties needed by the `@SpringBootTest`

```
@SpringBootTest(classes= {ClientTestConfiguration.class}, ①
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles({"https", "ntest"}) ②
```

```
@Slf4j
public class HttpsRestTemplateNTest {
    @Autowired
    private RestTemplate authnUser;
    @Autowired
    private URI authnUrl;
```

① test configuration with test-specific components

② activate application `https` profile and testing `ntest` profile

It is important to note that using HTTPS is not an intrusive part of the application or test. If we eliminate the `https` and `ntest` profiles, the tests will complete using straight HTTP.

7.2. application-https.properties [REVIEW]

The following is a repeat from above of what the application has defined for the `https` profile.

application-https.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
```

7.3. application-ntest.properties

The following snippet shows the `ntest` profile-specific configuration file used to complete the test definition. It primarily supplies the fact that:

- the protocol scheme will be HTTPS
- trustStore properties need for the client-side of communication

For convenience and consistency, I have defined the `ntest` property values using the `https` property values.

application-ntest.properties

```
it.server.scheme=https
it.server.trust-store=${server.ssl.key-store} ①
it.server.trust-store-password=${server.ssl.key-store-password} ①
```

① directly referencing properties defined in `application.properties`

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.

7.4. ServerConfig

For brevity, I will not repeat the common constructs seen in most API and security test configurations. However, the first quiet change to the `@TestConfiguration` is the definition of the `ServerConfig`. It looks very much the same but know that the `it.server.scheme` is applied to the returned object and is not yet in-place within the `Bean` factory method.

```
@Bean @Lazy
@ConfigurationProperties("it.server") ①
public ServerConfig itServerConfig(@LocalServerPort int port) { ②
    return new ServerConfig().withPort(port);
}
```

① `@ConfigurationProperties` (scheme=https) are applied to what is returned

② `@LocalServerPort` is assigned at startup

Custom port is being applied before @ConfigurationProperties

`@ConfigurationProperties` are assigned to what is returned from the `@Bean` factory. That means that `it.server.port` property would override `@LocalServerPort` if the property was provided.

7.5. Maven Dependencies

Spring 6 updated the networking in `RestTemplate` to use `httpclient5` and a custom SSL Context library for HTTPS communications. `httpclient5` and its TLS extensions require two new dependencies for our test client to support this extra setup.

Spring 6 RestTemplate Required HTTPS Dependencies

```
<!-- needed to set up TLS for RestTemplate -->
<dependency>
    <groupId>org.apache.httpcomponents.client5</groupId>
    <artifactId>httpclient5</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.github.hakky54</groupId>
    <artifactId>sslcontext-kickstart-for-apache5</artifactId>
    <scope>test</scope>
</dependency>
```

7.6. HTTPS ClientHttpRequestFactory

The HTTPS-based ClientHttpRequestFactory is built by following basic instructions in the docs and tweaking to make SSL a runtime option. There are three main objects created:

1. connectionManager using the optional sslFactory
2. httpClient using the connectionManager
3. requestFactory using the httpClient

The requestFactory is returned.

ClientHttpRequestFactory @Bean Factory with Optional SSL Support

```
import nl.altindag.ssl.util.Apache5SslUtils;
import org.apache.hc.client5.http.classic.HttpClient;
import org.apache.hc.client5.http.impl.classic.HttpClients;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManager;
import org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;
/*
https://sslcontext-kickstart.com/client/apache5.html
https://github.com/Hakky54/sslcontext-kickstart#apache-5
 */
@Bean @Lazy
public ClientHttpRequestFactory httpsRequestFactory(
        @Autowired(required = false) SSLFactory sslFactory) { ①
    PoolingHttpClientConnectionManagerBuilder builder =
            PoolingHttpClientConnectionManagerBuilder.create();
    PoolingHttpClientConnectionManager connectionManager =
      Optional.ofNullable(sslFactory)
        .map(sf->builder.setSSLSocketFactory(Apache5SslUtils.toSocketFactory(sf))) ②
        .orElse(builder) ③
        .build();

    HttpClient httpsClient = HttpClients.custom()
            .setConnectionManager(connectionManager)
            .build();
    return new HttpComponentsClientHttpRequestFactory(httpsClient);
}
```

① `@Autowired.required=false` allows `sslFactory` to be null if not using SSL

② `Optional` processing path if `sslFactory` is present

③ `Optional` processing path if `sslFactory` is not present

7.7. SSL Factory

The `SSLFactory` `@Bean` factory is conditional based on the presence of a non-empty `it.server.trust-store` property. If that property does not exist or has an empty value — this `@Bean` factory will not be invoked and the `ClientHttpRequestFactory` `@Bean` factory will be invoked with a null for the `sslFactory`.

The trustStore is loaded with the help of the Spring `ResourceLoader`. By using this component, we can use `classpath:`, `file:`, and other resource location syntax in order to construct an `InputStream` to ingest the trustStore.

Conditional SSLFactory @Bean Factory based on Present Trust Store Property

```
import nl.altindag.ssl.SSLFactory;
import org.springframework.core.io.ResourceLoader;

@Bean @Lazy
@ConditionalOnExpression(
        "!T(org.springframework.util.StringUtils).isEmpty('${it.server.trust-
store:}')") ①
public SSLFactory sslFactory(ResourceLoader resourceLoader, ServerConfig serverConfig)
        throws IOException {
    try (InputStream trustStoreStream = resourceLoader
            .getResource(serverConfig.getTrustStore()).getInputStream()) {
        return SSLFactory.builder()
            .withProtocols("TLSv1.2")
            .withTrustMaterial(trustStoreStream, serverConfig.getTrustStorePassword())
            .build();
    } catch (FileNotFoundException ex) {
        throw new IllegalStateException("unable to locate truststore: " +
serverConfig.getTrustStore(), ex);
    }
}
```

① conditional activates factory bean when `it.server.trust-store` is not empty

7.8. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTPS/Authn-enabled `RestTemplate` and baseUrl injected.

Review: Standard RestTemplate Construction Used to Date

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

From here it is just a normal test, but activity is remote on the server side.

Review: Example Test

```
public class HttpsRestTemplateNTest {
    @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

Example HTTPS Test Execution

```
INFO  ClientTestConfiguration#authnUrl:52 baseUrl=https://localhost:60364
INFO  HttpsRestTemplateNTest#setUp:30 baseUrl=https://localhost:60364/api/authn/hello
DEBUG RestTemplate#debug:127 HTTP GET https://localhost:60364/api/authn/hello?name=jim
```

With the `https` and `ntest` profiles disabled, the test reverts to HTTP.

Example HTTPS Test Execution

```
INFO  ClientTestConfiguration#authnUrl:52 baseUrl=http://localhost:60587
INFO  HttpsRestTemplateNTest#setUp:29 baseUrl=http://localhost:60587/api/authn/hello
DEBUG RestTemplate#debug:127 HTTP GET http://localhost:60587/api/authn/hello?name=jim
```

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