Securing REST APIs With Client Certificates

(/users/3408002/dzone-55.html) by Pavel Sklenar (/users/3408002/dzone-55.html)
MVB ·
Mar. 31, 19 · Integration Zone (/enterprise-integration-training-tools-news) · Tutorial

Like (35) Comment (9) Save Tweet

This post is about an example of securing a REST API (https://dzone.com/articles/developing-rest-apis) with a client certificate (a.k.a. X.509 certificate authentication).

In other words, a client verifies a server according to its certificate and the server identifies that client according to a client certificate (so-called mutual authentication).

In connection with Spring Security, we will be able to perform some additional authentication and authorization.

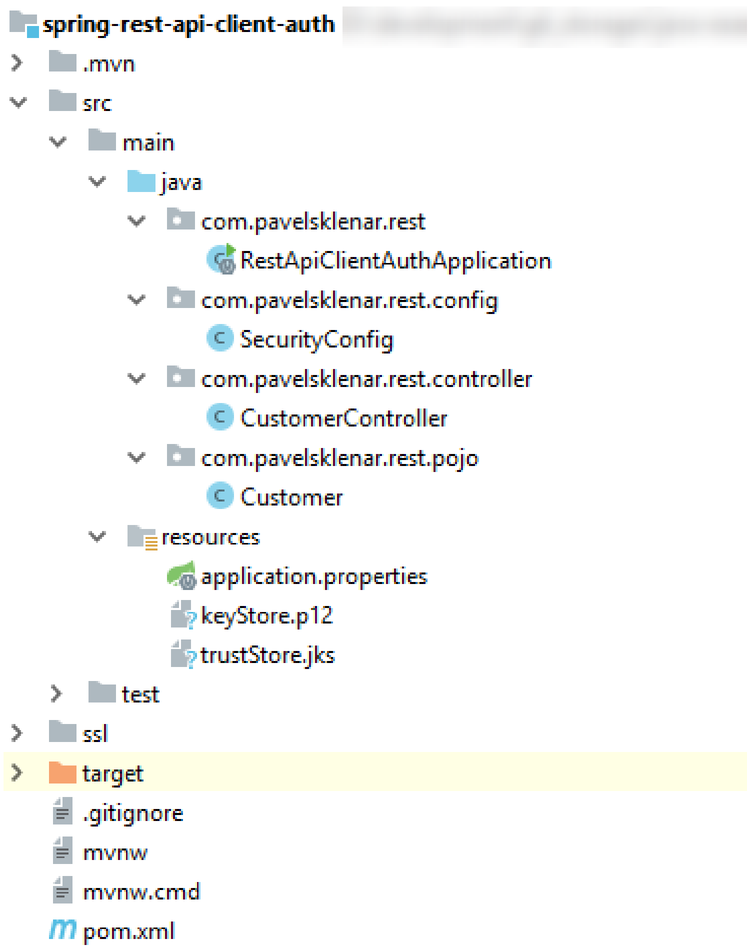
Technologies used:

1. Spring Boot 2.0.5.RELEASE
2. Spring Web + Security 5.0.8.RELEASE
3. Embedded Tomcat 8.5.34

Quick post overview:

- Create a simple REST API (service) (without any security)
- Create certificates for server and client
- Configure the server to serve HTTPS content
- Configure the server to require a client certificate
- Spring Security for further client authentication and authorization
- Test our secured REST API

Final Project Structure



```
spring-rest-api-client-auth
> .mvn
v src
  v main
    v java
      v com.pavelsklenar.rest
          RestApiClientAuthApplication
      v com.pavelsklenar.rest.config
          SecurityConfig
      v com.pavelsklenar.rest.controller
          CustomerController
      v com.pavelsklenar.rest.pojo
          Customer
    v resources
        application.properties
        keyStore.p12
        trustStore.jks
  > test
> ssl
> target
  .gitignore
  mvnw
  mvnw.cmd
  pom.xml
```

(https://blog.pavelsklenar.com/wp-content/uploads/2018/10/structure.png)

Creating a New Base Spring Boot Project

We will start with a new project generated by Spring Initializr (https://start.spring.io/). We just need two Spring dependencies, i.e. Spring Web + Spring Security.

All required dependencies are shown here:

```
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
    &lt;artifactId&gt;spring-boot-starter-security&lt;/artifactId&gt;
&lt;/dependency&gt;
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
    &lt;artifactId&gt;spring-boot-starter-web&lt;/artifactId&gt;
&lt;/dependency&gt;
```

Create a Simple REST API Service

Let's create a simple REST controller serving a detail about a customer using an HTTP GET method:

```
@RestController
@RequestMapping("/customer")
public class CustomerController {

    @GetMapping("/{id}")
    public Customer GetCustomer(@PathVariable Long id) {
        return new Customer(id, "Customer" + id);
    }
}
```

Displaying URL http://localhost:8080/customer/1 returns this JSON object:

```
{
    "id":1,
    "name":"Customer1"
}
```

Create Certificates for Server and Client

I want to stay focused on securing REST APIs so I will show you how to generate all required files in a very concise way. For more details about commands, visit my other blog post about creating a PKCS #12 key store. (https://blog.pavelsklenar.com/how-to-create-pkcs-12-for-your-application/)

```
#Create folders to generate all files (separated for client and server)
mkdir ssl &amp;&amp; cd ssl &amp;&amp; mkdir client &amp;&amp; mkdir server

## Server
# Generate server private key and self-signed certificate in one step
openssl req -x509 -newkey rsa:4096 -keyout server/serverPrivateKey.pem -out server/server.cr
# Create PKCS12 keystore containing private key and related self-sign certificate
openssl pkcs12 -export -out server/keyStore.p12 -inkey server/serverPrivateKey.pem -in serve
# Generate server trust store from server certificate
keytool -import -trustcacerts -alias root -file server/server.crt -keystore server/trustStor
```

```
## Client
# Generate client's private key and a certificate signing request (CSR)
openssl req -new -newkey rsa:4096 -out client/request.csr -keyout client/myPrivateKey.pem -r

## Server
# Sign client's CSR with server private key and a related certificate
openssl x509 -req -days 360 -in request.csr -CA server/server.crt -CAkey server/serverPrivat

## Client
# Verify client's certificate
openssl x509 -text -noout -in client/pavel.crt
# Create PKCS12 keystore containing client's private key and related self-sign certificate
openssl pkcs12 -export -out client/client_pavel.p12 -inkey client/myPrivateKey.pem -in clien
```

You can find the SSL folder with all generated files on the project's GitHub page (https://github.com/pajikos/java-examples/tree/master/spring-rest-api-client-auth).

We will use files in the server folder to configure our server.

The final client's file `client/client_pavel.p12` can be either imported into your browser or used in another client application.

On Windows, just open this file and import it into your system to test the REST API with any browser.

Configure the Server to Serve HTTPS Content

Basically, there are two options.

You can use any standalone server (e.g. Tomcat (https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html), WildFly (http://www.mastertheboss.com/jboss-server/jboss-security/complete-tutorial-for-configuring-ssl-https-on-wildfly), etc.) so the configuration would be specific to your choice. I prefer this choice for production environments.

Instead of configuring an application server, I will show you the second, simpler way of using an embedded Tomcat server inside Spring Boot.

The configuration is quite easy, we will change the port to 8443 and configure the server key store generated in the previous steps:

```
# Define a custom port (instead of the default 8080)
server.port=8443
# The format used for the keystore
server.ssl.key-store-type=PKCS12
# The path to the keystore containing the certificate
server.ssl.key-store=classpath:keyStore.p12
# The password used to generate the certificate
server.ssl.key-store-password=changeit
```

Configure the Server to Require a Client Certificate

The configuration of any server to require a client certificate (i.e. the mutual authentication (https://searchsecurity.techtarget.com/definition/mutual-authentication)) is very similar to the server side configuration except using words like a trust store instead of a key store.
So the embedded Tomcat (https://dzone.com/articles/a-step-by-step-guide-to-tomcat-performance-monitor) configuration seems like:

```
# Trust store that holds SSL certificates.
server.ssl.trust-store=classpath:trustStore.jks
# Password used to access the trust store.
server.ssl.trust-store-password=changeit
# Type of the trust store.
server.ssl.trust-store-type=JKS
# Whether client authentication is wanted ("want") or needed ("need").
server.ssl.client-auth=need
```

The embedded server now ensures (without any other configuration) that the clients with a valid certificate only are able to call our REST API.

Other clients will be declined by the server due to being unable to make correct SSL/TLS handshake (required by mutual authentication).

Please note that all configuration items starting server.* are related to an embedded Tomcat server only. You do not need it when using any standalone application server.

Spring Security for Further Client Authentication and Authorization

It would be fine to get an incoming client for our application as a logged user.

That gives us the possibility to perform some other authentications and authorizations using Spring Security (e.g. securing method calls to specific roles).

Until now, no Spring Security was needed, but all clients with any valid certificate may perform any call in our application without knowing who the caller is.

So we must configure Spring Security to create a logged user using a username from a client certificate (usually from the CN (https://www.ssl.com/faqs/common-name/) field, see the method call `subjectPrincipalRegex`):

field, see the method call subjectPrincipalRegex).

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .anyRequest().authenticated().and()
                .x509()
                    .subjectPrincipalRegex("CN=(.*?)(?:,|$)")
                    .userDetailsService(userDetailsService());
    }

    @Bean
    public UserDetailsService userDetailsService() {
        return (UserDetailsService) username -&amp;amp;amp;amp;amp;gt; {
            if (username.equals("pavel")) {
                return new User(username, "",
                        AuthorityUtils
                                .commaSeparatedStringToAuthorityList("ROLE_USER"));
            } else {
                throw new UsernameNotFoundException(String.format("User %s not found", userr
            }
        };
    }
}
```

Using the bean `UserDetailsService` is a kind of fake, but it shows an example of an additional authentication to accept only the username "pavel".

In other words, it accepts a client with a certificate containing the value "pavel" only in the certificate's CN field (as mentioned before, configured with `subjectPrincipalRegex`).

As you might have noticed, only the user "pavel" is a member of the role "user", so now we are able to restrict method calls to specific roles:

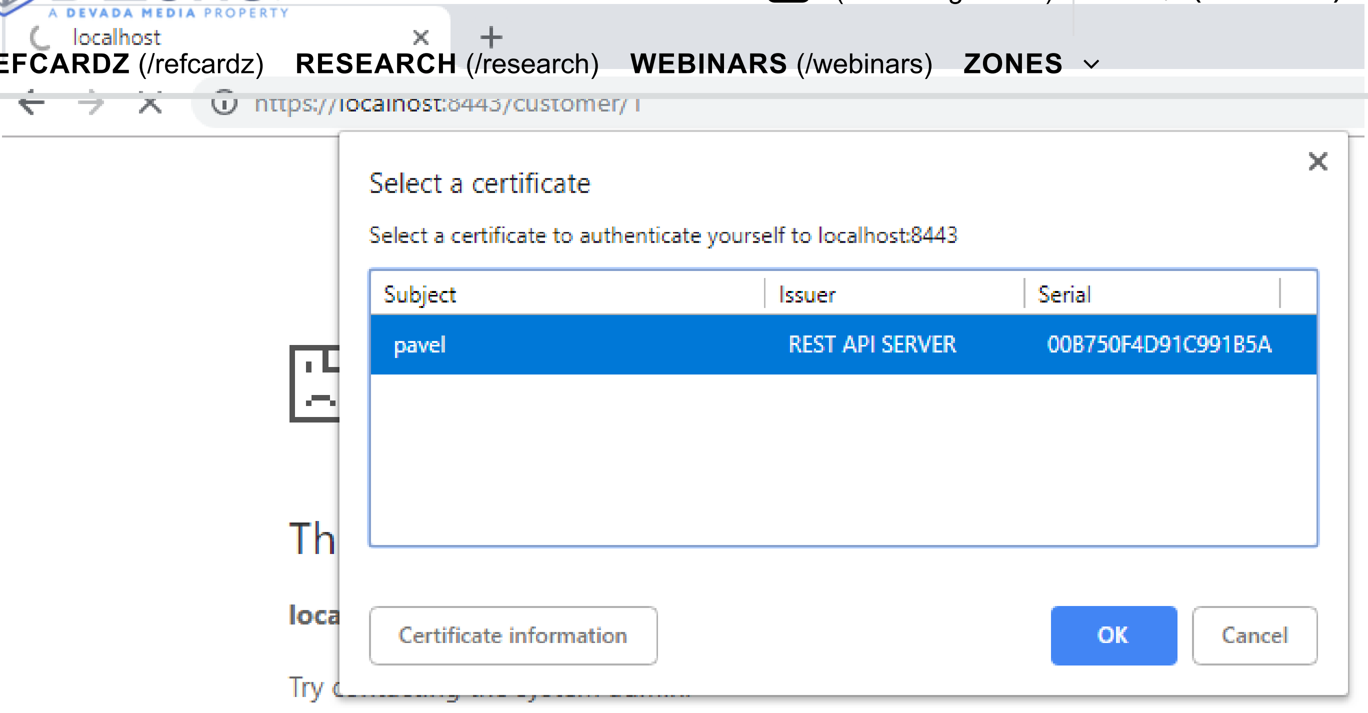
```
    @GetMapping("/{id}")
    @Secured("ROLE_USER")
    public Customer GetCustomer(@PathVariable Long id) {
        return new Customer(id, "Customer" + id);
    }
```

Test Secured REST API

When you successfully import `client/client_pavel.p12` into your system and the application runs, you can visit URL `https://localhost:8443/customer/1`.

The first access of this page displays a window to select the correct certificate to

The first access of this page displays a window to select the correct certificate to authenticate with the server:

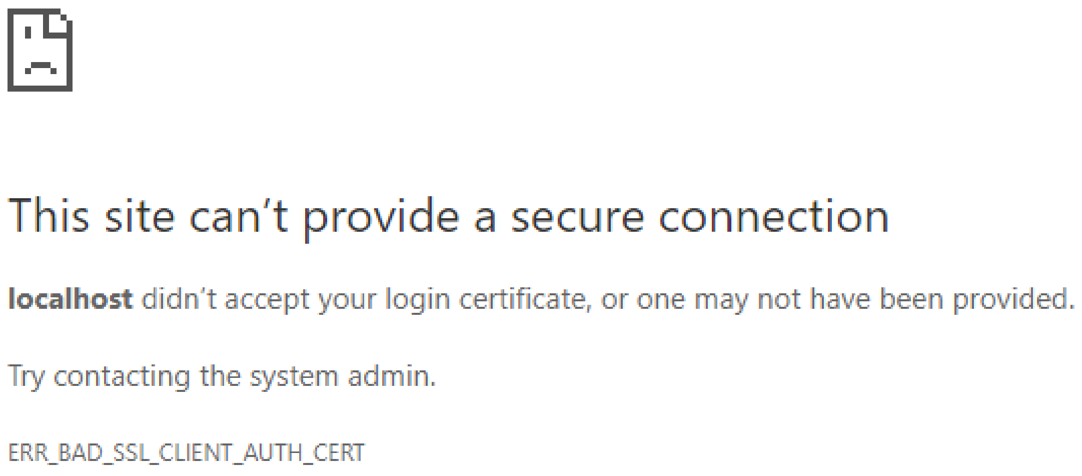

localhost
https://localhost:8443/customer/1

Select a certificate

Select a certificate to authenticate yourself to localhost:8443

Subject	Issuer	Serial
pavel	REST API SERVER	00B750F4D91C991B5A

When you submit an incorrect certificate, you will see the "access denied" page (otherwise JSON object returned):


This site can't provide a secure connection

localhost didn't accept your login certificate, or one may not have been provided.

Try contacting the system admin.

ERR_BAD_SSL_CLIENT_AUTH_CERT

(https://blog.pavelsklenar.com/wp-content/uploads/2018/10/access-denied.png)

That is all! You can find all my source code on my GitHub profile (https://github.com/pajikos/java-examples/tree/master/spring-rest-api-client-auth).

And a useful link for this topic:

- Spring Boot Official: Configure SSL (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-ssl)

Integration Partner Resources