Spring Security Workshop

Table of Contents

An always updated version of this document is available here as a PDF e-book.

A collection of Spring Security 5.7+ challenges; Part of a Spring Security blog series, and closely linked to the Spring Security Samples repository.

The project is divided into separate submodules, each of which demonstrates a single feature in isolation. While submodules can be combined to form a larger solution, we thought separating the functionality would make it easier to comprehend and extend.

All feedback much appreciated!

1. Requirements

In order to participate, you will need the following:

- Basic Spring knowledge
- Java 17+
- IDE of your choice
- Lombok IDE plugin
- `curl` or `httpie` or `IntelliJ HTTP Client`
- `docker-compose`, for Keycloak (optional)

Use https://sdkman.io/ or https://scoop-docs.vercel.app/ to easily install Java.

2. Getting started

1. Clone this Git repository.

```
git clone https://github.com/jdriven/spring-security-workshop.git
```

2. Verify the tests for the `adding-spring-security` module (only).

```
./mvnw -B verify --file adding-spring-security/pom.xml
```

3. The other modules will have compilation failures or failing tests, as their implementation is (intentionally) incomplete. It's up to you to implement the functionality to fix the tests!
4. If you are new to Spring Security, start with Adding Spring Security!
5. Once you have completed the basics, continue with any of the other modules.

3. Adding Spring Security

Adding Spring Security to an existing application can be quite a daunting prospect. Merely adding the required dependencies to your project sets off a chain of events which can break your application and tests.

Maybe you're suddenly shown a login prompt which expects a generated password logged on startup.
Maybe your tests now get the dreaded `401 Unauthorized`, or subsequently a `403 Forbidden`.
Maybe you get a `ClassCastException` when trying to use your `Authentication#getPrincipal()`.
So this should be fun!

We will walk you through adding Spring Security to an existing application, by explaining what happens when you first add the dependencies, what to do next, and how to fix your tests.

3.1. Context: Leave application

Imagine a Human Resources department that has a small application to track and approve/deny leave requests. Initially this application was only used from within the HR department, who received requests via phone or email. You can see the application in this state in leaveapp-initial, along with the current set of application tests.

Now we want to open up this application to all employees, so they can view and file (only their own) leave requests themselves. Anyone from the HR department can then either approve or deny a leave request, and view all requests. Up to you to implement these requirements for this small application.

3.2. 0. Verify initial application

Before we make any changes, it's good to get familiar with the leave application that we will work on. Look through the code to see there's a single controller, backed by a service, which connects to a repository, to store LeaveRequests.

3.2.1. Running the tests

In terms of tests, there's a single test for the service, and three different ways to test the controller:

1. once through @WebMvcTest,
2. once more through @SpringBootTest(webEnvironment = WebEnvironment.MOCK),
3. and finally through @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT).

That way, ideally, at least one of these methods will look familiar to what you know from your own projects.

All these tests should already pass on your machine as well, so go ahead and give that a try, either through your IDE, or using the below command as run from the workshop repository root.

Run the tests

```
# ~/workspace/spring-security-workshop $
./mvnw verify --file adding-spring-security/leaveapp-initial/pom.xml
```

3.2.2. Running the application

You should also be able to start the application through `LeaveRequestApplication` without any further changes, either through your IDE, or again, using the below command as run from the workshop repository root.

Run the application

```
# ~/workspace/spring-security-workshop $
./mvnw spring-boot:run --file adding-spring-security/leaveapp-initial/pom.xml
```

Once running you can immediately see an empty array of leave requests using this url: http://localhost:8080/view/all.

To populate your application with leave requests, run any of the below `curl` or `HTTPie` commands.

▼ *HTTPie commands to create, view and approve/deny leave requests.*

```
# Create a leave request for a specific user and time window
http POST ':8080/request/alice?from=2022-08-21&to=2022-09-11'

# View leave requests for employee
http :8080/view/employee/alice

# Approve leave request
http POST :8080/approve/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# Deny leave request
http POST :8080/deny/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# View leave request
http :8080/view/request/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# View all leave requests
http :8080/view/all
```

▼ *Curl commands to create, view and approve/deny leave requests.*

```
# Create a leave request for a specific user and time window
curl -v -X POST 'http://localhost:8080/request/alice?from=2022-08-21&to=2022-09-11'

# View leave requests for employee
curl -v http://localhost:8080/view/employee/alice

# Approve leave request
curl -v -X POST http://localhost:8080/approve/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# Deny leave request
curl -v -X POST http://localhost:8080/deny/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# View leave request
curl -v http://localhost:8080/view/request/2a37e1b6-d7e3-45fd-8b50-59357425d62e

# View all leave requests
curl -v http://localhost:8080/view/all
```

All of the above should just work on your machine. If it did, you're ready to start making changes.

3.3. 1. Adding Spring Security dependency

The first step in adding security to our application is picking the right dependency to add to our project. However, even figuring out which dependency to add can be difficult these days!

3.3.1. Which dependency to add?

Looking at start.spring.io we can see there are already five different dependencies related to Spring Security and/or OAuth2. In part this is down to a restructuring of OAuth2 support, with OAuth2 resource server and client support now moved into Spring Security 5.2+. As of June 1st 2022 the Spring Security OAuth project has reached End-of-Life.

In short we now advise against using the Spring Cloud Starter dependencies, and push towards using Spring Security support for OAuth2.

If your service will act as an OAuth2 resource server, by accepting JSON Web Tokens passed in from a gateway, you can use `spring-boot-starter-oauth2-resource-server`. We expect this to be the most common form within a micro-services landscape, where a central gateway assumes the OAuth2 client role. It is also what we will use throughout this blog post, although much of the details below apply to other forms as well.

If your service will act as an OAuth2 client to acquire JSON Web Tokens, you'll most likely want to use `spring-boot-starter-oauth2-client`.

Both starters will provide you with any transitive dependencies you might need for the most common security aspects.

3.3.2. What happens when you add the dependency?

Add the below dependency to leaveapp-initial/pom.xml.

oauth2-resource-server dependency snippet.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

When you rerun the tests, you will notice all web related tests now fail with either a HTTP `401 Unauthorized` or `403 Forbidden` response, depending on whether it's a `GET` or `POST` request.

When you run `LeaveRequestApplication` as before, and again open http://localhost:8080/view/all in the browser, you are now presented with a login dialog.

Please sign in

[Please sign in]

When you open the same endpoint from the commandline you immediately get a `HTTP/1.1 401` response.

We turn to the application logs to find out what happened in our application. As it turns out, there's a curious new logline from `UserDetailsServiceAutoConfiguration`:

Using generated security password warning

```
WARN UserDetailsServiceAutoConfiguration :

Using generated security password: 9c991bee-bf35-4970-92ed-e5458d561a73

This generated password is for development use only. Your security configuration must
be updated before running your application in production.
```

This auto configuration triggers when no other security configuration has been provided. It sets up our application with a default user and generated password, as a fallback of sorts. After all, if you're adding Spring Security to your class path you will want some form of security. At the very least the log line and dialog serve as a reminder to configure exactly what you want in your application.

3.4. 2. Configure OAuth2 resource server

Since we wish to configure our application to function as an OAuth2 resource server, we can provide the required configuration to make the generated security password go away. As indicated in the documentation, configuration takes the form of:

src/main/resources/application.properties snippet to configure oauth2 resource server.

```
spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://localhost:8090/auth/realms/spring-cloud-gateway-realm
```

Add the above snippet to application.properties.

Once added, the application will call out to the configured `issuer-uri` during startup, to configure the `JwtDecoder` through the OpenID Provider Configuration Information endpoint at: http://localhost:8090/auth/realms/spring-cloud-gateway-realm/.well-known/openid-configuration

But before we launch our application, we first need to ensure the issuer-uri is available, by running Keycloak.

3.4.1. Running Keycloak (or WireMock stubs)

During development we will either use Keycloak or WireMock to serve the configured `issuer-uri` endpoint. WireMock is the easiest one to get working quickly, as it does not require Docker compose; Keycloak takes some more effort, but can be used for your own projects as well.

Go ahead and run either Keycloak, or the WireMock stubs using the linked instructions.

The existing JSON Web Token provided below will only work with the WireMock stubs.

3.4.2. Authenticated requests

Next, launch the `LeaveRequestApplication` again, as you did before.

Now, when you open http://localhost:8080/view/all in the browser, you are no longer presented with a login dialog. Instead, you immediately get a `401 Unauthorized` error response. *"Progress"*!

The `UserDetailsServiceAutoConfiguration` warning about a `generated security password` has disappeared from the logs. The `UserDetailsServiceAutoConfiguration` from before has instead been replaced by `OAuth2ResourceServerJwtConfiguration`, which has provided us with a `JwtDecoder` to handle incoming tokens. Unless you provide your own `SecurityFilterChain`, this sets up your application to require a JSON Web Token for each request.

The reason we're now getting `40x` responses, is because our requests lack an `Authorization` header with `Bearer eyJhbGciOiJ⋯` JSON Web Token. To solve this, we have to actually pass a Bearer token along with our requests. Here's a token value to get your requests through with a `HTTP/1.1 20x` response:

▼ *Show Alice's JSON Web Token valid through August 23th 2032.*

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJUOUpHbFdTc244UXFoUkZlX19LWU5kRGVQ
Wm1xLVU1c2RZRWppQjFwNEpVIn0.eyJleHAiOjE5NzY4NzYzMzIsImlhdCI6MTY2MTUxNjMzMiwiYXV0aF90
aW1lIjoxNjYxNTE2MzMyLCJqdGkiOiJiNjc1YWJiMS05OTU5LTRhMDUtYTgxZS1iY2U1MWVjYjlkYzgiLCJp
c3MiOiJodHRwOi8vbG9jYWxob3N0OjgwOTAvYXV0aC9yZWFsbXMvc3ByaW5nLWNsb3VkLWdhdGV3YXktcmVh
```

```
bG0iLCJhdWQiOiJhY2NvdW50Iiwic3ViIjoiMTUzYmJkODYtNWQ1Yy00MmFkLTgyNjEtN2E5MzlmZjFjZWIy
IiwidHlwIjoiQmVhcmVyIiwiYXpwIjoic3ByaW5nLWNsb3VkLWdhdGV3YXktY2xpZW50Iiwibm9uY2UiOiJy
WHpXNVItdUhnRHJYTGRZUFZ3TDdmQUtwNGZNQ0ZuRmJDcl9RUllaREhVIiwic2Vzc2lvbl9zdGF0ZSI6ImMw
ZjNmNzYzLTc2NzQtNDZmNi1iMWNmLWQ5OWZiMDNmNDZhZiIsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJv
ZmZsaW5lX2FjY2VzcyIsImRlZmF1bHQtcm9sZXMtc3ByaW5nLWNsb3VkLWdhdGV3YXktcmVhbG0iLCJ1bWFf
YXV0aG9yaXphdGlvbiJdfSwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdl
LWFjY291bnQiLCJtYW5hZ2UtYWNjb3VudC1saW5rcyIsInZpZXctcHJvZmlsZSJdfX0sInNjb3BlIjoib3Bl
bmlkIGVtYWlsIHByb2ZpbGUiLCJzaWQiOiJjMGYzZjc2My03Njc0LTQ2ZjYtYjFjZi1kOTlmYjAzZjQ2YWYi
LCJlbWFpbF92ZXJpZmllZCI6ZmFsc2UsInByZWZlcnJlZF91c2VybmFtZSI6ImFsaWNlIn0.V9sQJ_8yP4qH
XMFq5fNQUG8KddHWVO5aiK6k08liiHXPDJw4sZwOHHSANS-7esxZOBcuvoxPOMYFEjY8k33-
PhuAbswoZaFSGSf19ksgC5s2dRlskFW1Z6QQNGEOtitSTk0O3xZ8606ZBsRR1asx-
X6MMXejwY5wtCl53mwcBlCB_Vs32UsXk8E7QbsVSCfl-
Inpab2w4reDb635n8wOo2RGLhK_8kxdR_8p7FEuKLUrzI12eU9IvWZf0XQvNC-
W_Niw52W1DIQ_SSnlMt17jIKvRRBOmBiSDq1gXz56oYEWGaDPVZG_u_ooWD8WqyOXq9gP4EsXK_J0LvLOCp8
5Vg
```


▼ *Show HR JSON Web Token valid through August 23th 2032.*


```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJUOUpHbFdTc244UXFoUkZlX19LWU5kRGVQ
Wm1xLVU1c2RZRWppQjFwNEpVIn0.eyJleHAiOjE5NzY4NzY0MjcsImlhdCI6MTY2MTUxNjQyNywiYXV0aF90
aW1lIjoxNjYxNTE2NDI3LCJqdGkiOiI4NTc1MzBmMS02YWNkLTQ4MGEtOGYwNi0yNTIzZjk1ZTllMGIiLCJp
c3MiOiJodHRwOi8vbG9jYWxob3N0OjgwOTAvYXV0aC9yZWFsbXMvc3ByaW5nLWNsb3VkLWdhdGV3YXktcmVh
bG0iLCJhdWQiOiJhY2NvdW50Iiwic3ViIjoiYTJmYjE3NmQtMWYwMy00ZWM1LWJiMGQtZTI3Y2VmYTUwY2Fj
IiwidHlwIjoiQmVhcmVyIiwiYXpwIjoic3ByaW5nLWNsb3VkLWdhdGV3YXktY2xpZW50Iiwibm9uY2UiOiJs
b0FuTlF5Q1Ntc1NjSlZnSjNubTdEN3hUVWhpc2Z0YlY1bjF5NWY5blp3Iiwic2Vzc2lvbl9zdGF0ZSI6ImRl
MjkzZjVlLTc0NmUtNGRmNi1iMjFjLWQ2Nzc5ZTE5YjBkNyIsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJv
ZmZsaW5lX2FjY2VzcyIsImRlZmF1bHQtcm9sZXMtc3ByaW5nLWNsb3VkLWdhdGV3YXktcmVhbG0iLCJIUiIs
InVtYV9hdXRob3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xlcyI6WyJt
YW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwic2NvcGUi
OiJvcGVuaWQgZW1haWwgcHJvZmlsZSIsInNpZCI6ImRlMjkzZjVlLTc0NmUtNGRmNi1iMjFjLWQ2Nzc5ZTE5
YjBkNyIsImVtYWlsX3ZlcmlmaWVkIjpmYWxzZSwibmFtZSI6ImJvYiIsInByZWZlcnJlZF91c2VybmFtZSI6
ImJvYiIsImdpdmVuX25hbWUiOiJib2IifQ.YesLamdNivBSa1eqxroBM6aKEaTeTf75c0Wfdsfwz8hWtjqQg
Nf9GglS7-kWK_lWc0PfH5zyU2X0slYM0ZRCUo-
8j6oM5v3OLkc77Qlc5uszUVrVcGOAcHe4jjSnqLfqsavPmruRXtzbCJncqvYuR0LW5Nfu1UWxyT28UPiwXVC
9juobgXraStmOlxaDXIP8JZWdEPqqpqrOjlLVuwMsXQi5g-
3L89idU7Bfaz0ujHMt9JX83gKxzQDf6I825fmGPQNabruI8gPOWMxTT-XzDoSfUgUOZ-
jXSa0bCADKrjAEpw5Q9KOnFzncoO6u6nQuUpwYvWfqRafACm0oy0P6Xw
```


If you're curious what information is contained within this token, you can easily decode the token into it's three parts using https://jwt.io.

▼ *Show decoded JSON Web Token payload.*


```
{
  "exp": 1976876332,
  "iat": 1661516332,
  "auth_time": 1661516332,
  "jti": "b675abb1-9959-4a05-a81e-bce51ecb9dc8",
  "iss": "http://localhost:8090/auth/realms/spring-cloud-gateway-realm",
  "aud": "account",
```

```
  "sub": "153bbd86-5d5c-42ad-8261-7a939ff1ceb2",
  "typ": "Bearer",
  "azp": "spring-cloud-gateway-client",
  "nonce": "rXzW5R-uHgDrXLdYPVwL7fAKp4fMCFnFbCr_QRYZDHU",
  "session_state": "c0f3f763-7674-46f6-b1cf-d99fb03f46af",
  "realm_access": {
    "roles": [
      "offline_access",
      "default-roles-spring-cloud-gateway-realm",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid email profile",
  "sid": "c0f3f763-7674-46f6-b1cf-d99fb03f46af",
  "email_verified": false,
  "preferred_username": "alice"
}
```

The below `HTTPie` and `curl` commands should all produce `20x` responses when both Keycloak (stubs) and `LeaveRequestApplication` are running.

Requests authenticated through a JSON Web Token.

```
# Store full token
export token=eyJhbGciOiJ...

# Create a leave request for a specific user and time window
http POST ':8080/request/alice?from=2022-08-21&to=2022-09-11' "Authorization: Bearer
${token}"
curl -v -X POST -H "Authorization: Bearer ${token}"
'http://localhost:8080/request/alice?from=2022-08-21&to=2022-09-11'

# View all leave requests
http :8080/view/all "Authorization: Bearer ${token}"
curl -v -H "Authorization: Bearer ${token}" http://localhost:8080/view/all
```

Perfect; our application now at the very least requires a valid JWT for any request. This covers our authentication needs for now; we will get to authorization at a later stage.

3.5. 3. Fixing the web tests

Now we have an application that requires an OpenID Connect provider on startup, and a valid JWT for any request. Neither plays well with the tests we have, so we're going to have to fix each of the different test flavors we have.

The leave application has four different types of tests which each use a partial or full Spring application context, to simulate what you might find in an existing application. They are identified below using the annotation and argument that bootstraps the test application context. We'll go over each test and the changes needed to make them pass again.

3.5.1. Tests using @SpringBootTest(webEnvironment = WebEnvironment.NONE)

Tests not using any web request are not (yet) broken. But don't worry, we will break these as soon as we add authorization.

3.5.2. Tests using @WebMvcTest(controllers = LeaveRequestController.class)

Next up, we want to make `LeaveRequestControllerWebMvcTest` pass again; We're testing the controller in isolation here, using `@WebMvcTest` together with `MockMvc`. Running the tests we see `GET` requests now get a `401 Unauthorized` response, while `POST` requests get a `403 Forbidden` response.

- The `401 Unauthorized` response on `GET` requests we get because we're not yet passing an `Authorization: Bearer eyJhbGciOiJ⋯` header in our tests.
- The `403 Forbidden` response takes a little more diving into; Debug logging through `logging.level.org.springframework.security: DEBUG` points us in the right direction:

```
DEBUG --- [main] o.s.security.web.csrf.CsrfFilter         : Invalid CSRF token found
for http://localhost/request/alice
DEBUG --- [main] o.s.s.w.access.AccessDeniedHandlerImpl   : Responding with 403 status
code
```

By default Spring Security adds Cross Site Request Forgery protection for `POST` requests. This protects our resource server from malicious requests; and it's best not to disable this. One way to get around this requirement for tests is to add `csrf` tokens to our POST requests, through `SecurityMockMvcRequestPostProcessors#csrf()` from `spring-security-test`. But if we only do that, we would then get a `401 Unauthorized` response!

So let's look into a proper fix; First, add the `spring-security-test` dependency to the leaveapp-initial/pom.xml file.

spring-security-test dependency snippet.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
```

```
    <scope>test</scope> ①
</dependency>
```

① Notice how we use the `test` scope, as this library is only needed on the test classpath.

This library provides us with `SecurityMockMvcRequestPostProcessors#jwt()`, among others, which we add to our test methods to have them pass a valid JWT along with each request. The JWT can be configured in a number of different ways. Update all the tests in `LeaveRequestControllerWebMvcTest` with the `jwt()` addition as seen below.

```
import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProces
sors.jwt; ①

...

@Test
void testRequest() throws Exception {
    when(service.request(anyString(), any(), any()))
        .thenReturn(new LeaveRequest("alice", of(2022, 11, 30), of(2022, 12, 3),
PENDING));
    mockmvc.perform(post("/request/{employee}", "alice")
        .param("from", "2022-11-30")
        .param("to", "2022-12-03")
        .with(jwt())) ②
        .andExpectAll(
            status().isAccepted(),
            content().contentType(MediaType.APPLICATION_JSON),
            jsonPath("$.employee").value("alice"),
            jsonPath("$.status").value("PENDING"));
}
```

① To keep the tests readable, we will use a static import.

② For now, we will use a plain `jwt()` with a `sub` (subject) claim of value `user`.
Later on, we will set the subject name to `alice` to match our requests.

Coincidentally, this also already resolves the CSRF issue with POST requests, as the `CsrfFilter` is skipped through the `JwtRequestPostProcessor`. If you repeat this pattern for all test methods and rerun the tests, you'll find all tests within `LeaveRequestControllerWebMvcTest` pass again!

3.5.3. Tests using @SpringBootTest(webEnvironment = WebEnvironment.MOCK)

In short, for now, these tests need the same treatment as we saw with `@WebMvcMock`. So go ahead and add `jwt()` to the web requests to get these tests working again as well.

3.5.4. Tests using @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)

The most challenging tests to adapt, are the ones using `TestRestTemplate`, mostly as the API makes it cumbersome to add headers. And even once you manage to add the `Authorization` header successfully to each request, you still have to mock the token handling.

So let's have a look at a complete example, to give you an idea of what's involved. Here's an unaltered test not yet passing authorization headers.

Unaltered test to POST new leave request for user Alice.

```
@Test
void testRequest() {
    LocalDate from = of(2022, 11, 30);
    LocalDate to = of(2022, 12, 3);
    ResponseEntity<LeaveRequestDTO> response = restTemplate.postForEntity(
        "/request/{employee}?from={from}&to={to}",
        null, LeaveRequestDTO.class, "alice", from, to);
    assertThat(response.getStatusCode()).isEqualByComparingTo(ACCEPTED);

assertThat(response.getHeaders().getContentType()).isEqualByComparingTo(APPLICATION_JS
ON);
    assertThat(response.getBody().getEmployee()).isEqualTo("alice");
    assertThat(response.getBody().getFromDate()).isEqualTo(from);
    assertThat(response.getBody().getToDate()).isEqualTo(to);
    assertThat(response.getBody().getStatus()).isEqualByComparingTo(PENDING);
}
```

Next, here's that same test, but now with all the bits and pieces to pass an authorization header, to trigger the JWT handling, which we mock to decode the token.

Test to POST new leave request for user `alice`, with `Authorization` header.

```
@MockBean
private JwtDecoder jwtDecoder; ①

@Nested
class AuthorizeUser {

    @BeforeEach
    void beforeEach() {
        when(jwtDecoder.decode(anyString())) ②
            .thenReturn(Jwt.withTokenValue("token")
                .subject("alice")
                .header("alg", "none")
                .build());
    }

    @Test
```

```
    void testRequest() throws Exception {
        HttpHeaders headers = new HttpHeaders();
        headers.setBearerAuth("some.random.token"); ③
        HttpEntity<?> httpEntity = new HttpEntity<>(headers); ④

        LocalDate from = of(2022, 11, 30);
        LocalDate to = of(2022, 12, 3);
        ResponseEntity<LeaveRequestDTO> response = restTemplate.exchange( ⑤
            "/request/{employee}?from={from}&to={to}",
            HttpMethod.POST, httpEntity, LeaveRequestDTO.class, "alice", from, to);
        assertThat(response.getStatusCode()).isEqualByComparingTo(ACCEPTED);

assertThat(response.getHeaders().getContentType()).isEqualByComparingTo(APPLICATION_JS
ON);
        assertThat(response.getBody().getEmployee()).isEqualTo("alice");
        assertThat(response.getBody().getFromDate()).isEqualTo(from);
        assertThat(response.getBody().getToDate()).isEqualTo(to);
        assertThat(response.getBody().getStatus()).isEqualByComparingTo(PENDING);
    }

    // ...

}
```

① We provide a `@MockBean JwtDecoder` to prevent the deferred JWT decoder initialization from triggering once a JWT is passed in.
If we do not add the mock bean, the application will call out to the `issuer-uri` upon receiving the first `Authorization` header. And hosting a OIDC server for your tests is just not feasible or desirable.

② We mock the `JwtDecoder.decode(String)` to always return the same `Jwt` value.
The response should match your actually decoded tokens, which can be hard to keep in sync.

③ We provide a dummy value for the `Authorization: Bearer` header.
This ensures the JWT handling is triggered, to decode the dummy value.

④ We wrap the headers into a `HttpEntity`.

⑤ We switch from `TestRestTemplate.postForEntity(…)` to `TestRestTemplate.exchange(…)`, as needed to pass the header.

As you can see, this quickly becomes cumbersome, even when common elements are extracted out into methods. Have a look at the reference implementation to see the complete converted test.

There's little value in converting all these tests by hand; just know that it can be done, but try to limit or avoid these types of tests. You might want to get some practice if these types of tests are common in your application, but otherwise feel free to delete the test, or copy the complete sample.

3.6. 4. Adding authorization

Our application now requires a JWT with every request, and decodes that into an Authentication object. While this is all fine and needed; it does not yet achieve much in terms of security; anyone

can authenticate, file and view leave requests and approve or deny them as they see fit. We need to configure our application with some common sense roles and restrictions. For instance:

- we want users to only submit and view requests for themselves;
- only HR employees can approve or deny requests, and view all requests.

3.6.1. Configuring method security

Since all requests pass through `LeaveRequestService`, this seems like the perfect place to add our security restrictions. We'll add a variety of security annotation and expressions, all of which are documented extensively in the Authorization chapter of Spring Security.

The security annotation handling needs to be enabled through `@EnableMethodSecurity(jsr250Enabled = true)`, as without this annotation the security annotations will not enforce anything!

Previously one might have used `@EnableGlobalMethodSecurity`, but this has been simplified as of Spring Security 5.6.

Add the annotation to a new configuration class. The class itself does not have to be annotated with `@Configuration`, as `@EnableMethodSecurity` is already meta-annotated.

Enable method security through a new configuration class.

```
@EnableMethodSecurity(jsr250Enabled = true)
public class MethodSecurityConfig {
    // ...
}
```

3.6.2. Require HR role

The easiest methods to secure are the ones that are only accessible to users with the `HR` role. For this we will use the `@javax.annotation.security.RolesAllowed` annotation, with an argument value of `HR`.

Go ahead and add these annotations to `LeaveRequestService` now.

```
@RolesAllowed("HR")
public Optional<LeaveRequest> approve(UUID id) {
    Optional<LeaveRequest> found = repo.findById(id);
    found.ifPresent(lr -> lr.setStatus(Status.APPROVED));
    return found;
}

@RolesAllowed("HR")
public Optional<LeaveRequest> deny(UUID id) {
    Optional<LeaveRequest> found = repo.findById(id);
    found.ifPresent(lr -> lr.setStatus(Status.DENIED));
    return found;
}
```

```
@RolesAllowed("HR")
public List<LeaveRequest> retrieveAll() {
    return repo.findAll();
}
```

If you rerun all tests, you will immediately notice any tests from classes annotated with `@SpringBootTest` using methods will fail. `LeaveRequestControllerWebMvcTest` is unaffected, as it uses a `@MockBean LeaveRequestService`, that does not trigger method security.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.NONE)

Our `LeaveRequestServiceTest` methods now need an active user, where this was not needed before. We use the `@WithMockUser` annotation from `spring-security-test`, with arguments added to match the required `HR` role.

Add mock user with HR role to all `AuthorizeRole` tests.

```
@Nested
@WithMockUser(roles = "HR")
class AuthorizeRole {
    ...
}
```

This should immediately fix all test failures in `LeaveRequestServiceTest.AuthorizeRole`.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.MOCK)

Previously we got away with passing in just any `jwt()` to our web controller tests using `@SpringBootTest`. Now, with the addition of `RolesAllowed`, tests such as `LeaveRequestControllerSpringBootWebEnvMockTest.AuthorizeRole` also need to pass a JSON Web Token with the `HR` role. Luckily that's easy enough; Add the following to each of the authorized role test methods.

MockMvc test to approve a leave request as a user with HR role.

```
@Test
void testApprove() throws Exception {
    LeaveRequest saved = repository
        .save(new LeaveRequest("alice", of(2022, 11, 30), of(2022, 12, 3), PENDING));
    mockmvc.perform(post("/approve/{id}", saved.getId())
        .with(jwt().authorities(new SimpleGrantedAuthority("ROLE_HR")))) ①
        .andExpectAll(
            status().isAccepted(),
            content().contentType(MediaType.APPLICATION_JSON),
            jsonPath("$.employee").value("alice"),
            jsonPath("$.status").value("APPROVED"));
}
```

① The `jwt()` call from before, now adds a known authority to pass the authorization requirement

on the service method.

With these roles added to each of the authorized roles tests, these should again pass.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)

These tests now also fail, but before we fix these we first need to introduce a few more components. Check back after we discussed converting JWT claims, and ignore these failing tests for now.

3.6.3. Require user *or* HR role

The more challenging methods to secure are the methods that are accessible both to individual users, as well as users with the HR role. Users should only be able to submit and view their own leave requests (only), while users with HR role have no such restrictions.

To protect these methods we will use a mix of pre- and post authorization annotations. These annotations take a Spring-EL expression as argument, which has access to the method arguments and returned object, as well as the authenticated user and built-in expressions.

Add the below annotations and security expressions to `LeaveRequestService`.

Pre- and PostAuthorizate annotations to allow both individual users and users with HR role.

```
@PreAuthorize("#employee == authentication.name or hasRole('HR')") ①
public LeaveRequest request(String employee, LocalDate from, LocalDate to) {
    LeaveRequest leaveRequest = LeaveRequest.builder()
        .employee(employee)
        .fromDate(from)
        .toDate(to)
        .build();
    return repo.save(leaveRequest);
}

@PreAuthorize("#employee == authentication.name or hasRole('HR')") ②
public List<LeaveRequest> retrieveFor(String employee) {
    return repo.findByEmployee(employee);
}

@PostAuthorize("returnObject.orElse(null)?.employee == authentication.name or
hasRole('HR')") ③
public Optional<LeaveRequest> retrieve(UUID id) {
    return repo.findById(id);
}
```

① Notice how `#employee` refers to the method argument name. For simplicity we use the name rather than an identifier.

② If you frequently use the exact same security expression, consider creating your own meta annotations.

③ We *post* authorize on the `returnObject`, after making the call out to the repository, as the `UUID` argument contains insufficient information for our security expression.

If you rerun the tests after adding these new annotations, you will see new `AuthorizeUser` test failures pop up.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.NONE)

The `LeaveRequestServiceTest.AuthorizeUser` tests will now fail, as "an Authentication object was not found in the SecurityContext". We can resolve these failures in much the same way as we saw before with `@WithMockUser`, this time using the `username` argument value to match our request employee.

Add mock user with username `alice` to all `AuthorizeUser` tests.

```
@Nested
@WithMockUser(username = "alice")
class AuthorizeUser {
    ....
}
```

There's one interesting test failure that persists, even after adding `@WithMockUser`, and it revolves around retrieving a non existing leave request. `testRetrieveByIdMissing` calls out to `LeaveRequestService.retrieve(UUID)`, which is protected through `@PostAuthorize`.

Think for a moment how you would protect an empty `Optional` response, when there's no `employee` field to use in the Spring Expression. Also consider what a potential attacker could learn about the existence of leave requests if we would not block requests to non existing identifiers.

To be safe, we wrote our security expression to deny access when the corresponding leave request can not be found, through `returnObject.orElse(null)?.employee == authentication.name`. The implication is that we now have to handle these cases differently in our tests as well.

Rewrite assertions around retrieving an unknown leave request.

```
@Test
void testRetrieveByIdMissing() {
    UUID randomUUID = UUID.randomUUID();
-   Optional<LeaveRequest> retrieved = service.retrieve(randomUUID); ①
+   assertThrows(AccessDeniedException.class, () -> service.retrieve(randomUUID)); ②
    verify(repository).findById(randomUUID); ③
-   assertThat(retrieved).isEmpty(); ④
}
```

① We no longer get an `Optional` response to verify.

② Instead we assert the method call is blocked with a Spring Security Exception.

③ Notice how the method still calls out to the repository as before.

④ The empty `Optional` assertion is removed, as exception handling takes over.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.MOCK)

We see a similar pattern in test failures for `LeaveRequestControllerSpringBootWebEnvMockTest.AuthorizeUser` that we previously saw for `AuthorizeRole` in the same test file. Where we used to be able to pass in just any `jwt()`, we not have to provide a token with matching characteristics, in this case a username `alice`.

Repeat the below JWT change for each of the unit tests in `LeaveRequestControllerSpringBootWebEnvMockTest.AuthorizeUser`.

Set subject claim to `alice` to match authorization expressions.

```
@Test
void testRequest() throws Exception {
    mockmvc.perform(post("/request/{employee}", "alice")
        .param("from", "2022-11-30")
        .param("to", "2022-12-03")
        .with(jwt().jwt(builder -> builder.subject("alice")))) ①
        .andExpectAll(
            status().isAccepted(),
            content().contentType(MediaType.APPLICATION_JSON),
            jsonPath("$.employee").value("alice"),
            jsonPath("$.status").value("PENDING"));
}
```

① Set the subject to `alice`, to match the POST request URL value.

Tests using @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)

If you already adopted the suggested approach of mocking the `JwtDecoder` response for `LeaveRequestControllerSpringBootWebEnvRandomPortTest.AuthorizeUser` above, then the tests should already pass. Here is the relevant section from that approach again; scroll up for the full details.

Mock the JwtDecoder response to return a `Jwt` with username alice.

```
@MockBean
private JwtDecoder jwtDecoder;

@Nested
class AuthorizeUser {

    @BeforeEach
    void beforeEach() {
        when(jwtDecoder.decode(anyString()))
            .thenReturn(Jwt.withTokenValue("token")
                .subject("alice") ①
                .header("alg", "none")
                .build());
    }

    // ...
```

```
}
```

① The subject claim will be turned into the `Authentication` object `name`, as used in our security expressions.

That should only leave `LeaveRequestControllerSpringBootWebEnvRandomPortTest.AuthorizeRole` with test failures, which we will finally fix in the next section, by introducing claim mapping.

3.7. 5. Convert JWT claims

One final and challenging part of adding Spring Security, is aligning the JSON Web Token claims returned by your token provider with the Authentication objects used throughout your application for authorization. These details will differ from provider to provider, and thus need fine tuning whenever you switch between providers. And while there are some common claims, such as `subject` for the user name, you might still want to map other claims to get the user's roles or preferred name.

Let's take another closer look at the JSON Web Token returned by Keycloak for Bob from HR.

Decoded JSON Web Token payload for Bob from HR.

```
{
  "exp": 1976876427,
  "iat": 1661516427,
  "auth_time": 1661516427,
  "jti": "857530f1-6acd-480a-8f06-2523f95e9e0b",
  "iss": "http://localhost:8090/auth/realms/spring-cloud-gateway-realm",
  "aud": "account",
  "sub": "a2fb176d-1f03-4ec5-bb0d-e27cefa50cac", ①
  "typ": "Bearer",
  "azp": "spring-cloud-gateway-client",
  "nonce": "loAnNQyCSmsScJVgJ3nm7D7xTUhisftbV5n1y5f9nZw",
  "session_state": "de293f5e-746e-4df6-b21c-d6779e19b0d7",
  "realm_access": {
    "roles": [
      "offline_access",
      "default-roles-spring-cloud-gateway-realm",
      "HR", ②
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
```

```
  "scope": "openid email profile",
  "sid": "de293f5e-746e-4df6-b21c-d6779e19b0d7",
  "email_verified": false,
  "name": "bob",
  "preferred_username": "bob", ③
  "given_name": "bob"
}
```

① Notice how the `sub` claim is a generated UUID.

② Bob's `HR` role is nested under `$.realm_access.roles`.

③ Bob's preferred username is `bob`.

Up to now we had set the `sub` or subject claim in our tests explicitly to `alice`, or assigned the `HR` role directly. For our application to work with Keycloak JSON Web Tokens, we have to properly convert the JWT claims.

3.7.1. Preferred name

Spring Security assumes the JWT subject as authentication name, which is what we use in our method security expressions. So to ensure the authentication name matches the `#employee` username passed into our web requests, we have to rename the `sub` claim.

```
@EnableWebSecurity ①
@ConditionalOnWebApplication ②
class WebSecurityConfig {

    @Bean
    @ConditionalOnProperty(name = "spring.security.oauth2.resourceserver.jwt.issuer-
uri") ③
    JwtDecoder jwtDecoderByIssuerUri(OAuth2ResourceServerProperties properties) {
        String issuerUri = properties.getJwt().getIssuerUri();
        NimbusJwtDecoder jwtDecoder = (NimbusJwtDecoder)
JwtDecoders.fromIssuerLocation(issuerUri);
        // Use preferred_username from claims as authentication name, instead of UUID
subject
        jwtDecoder.setClaimSetConverter(new UsernameSubClaimAdapter()); ④
        return jwtDecoder;
    }

}

class UsernameSubClaimAdapter implements Converter<Map<String, Object>, Map<String,
Object>> {

    private final MappedJwtClaimSetConverter delegate =
MappedJwtClaimSetConverter.withDefaults(Collections.emptyMap());

    @Override
    public Map<String, Object> convert(Map<String, Object> claims) {
```

```
        Map<String, Object> convertedClaims = this.delegate.convert(claims);
        String username = (String) convertedClaims.get("preferred_username"); ⑤
        convertedClaims.put("sub", username);
        return convertedClaims;
    }

}
```

① We introduce a new `WebSecurityConfig`, annotated with `@EnableWebSecurity`.

② By adding `@ConditionalOnWebApplication`, the presence of this class will not interfere with `LeaveRequestServiceTest`.

③ Our JwtDecoder requires the `issuer-uri`, so we only conditionally load this bean.

④ We wire up our customized `JwtDecoder`, as indicated in the documentation.

⑤ Finally, we extract the `preferred_username` claim, and override the `sub` claim.

With this converter in place, our Authentication object should now user the user's preferred name as name attribute.

There's just a minor inconvenience, as the default property value in `src/main/resources/application.yml`, triggers the conditional `JwtDecoder` bean in `LeaveRequestControllerSpringBootWebEnvMockTest`, which breaks our application context initialization.. To get around this, we set the property value to `false` in `src/test/resources/application.yml`, to make the auto-configuration back off.

src/test/resources/application.yml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: false
```

3.7.2. Granted authorities

Both Keycloak and the WireMock stubbed responses wrap the assigned roles inside a claim called `realm_access` by default. Within that claim there's a `roles` field, that holds a collection of roles.

`realm_access` *claim taken from decoded Keycloak JSON Web Token payload.*

```
"realm_access": {
  "roles": [
    "HR"
  ]
},
```

Because of the nested structure of these claims, we have to manually extract the authorities. The

process is a little involved, and hard to introduce gradually, so instead, here's the full annotated class.

SecurityFilterChain for custom JSON Web Token claim conversion.

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception { ①
    return http
        .authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated())
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(jwt -> jwt.jwtAuthenticationConverter(new
KeycloakRealmRoleConverter()))) ①
        .build();
}
```

① We add a `SecurityFilterChain` to our `WebSecurityConfig`.

② We wire up our custom authentication converter, as indicated in the documentation.

KeycloakRealmRoleConverter to unpack realm_access roles into GrantedAuthorities.

```
class KeycloakRealmRoleConverter implements Converter<Jwt, JwtAuthenticationToken> {

    @Override
    @SuppressWarnings("unchecked")
    public JwtAuthenticationToken convert(Jwt jwt) {
        Map<String, Object> realmAccess = (Map<String, Object>)
jwt.getClaims().getOrDefault("realm_access", Collections.emptyMap());
        List<String> roles = (List<String>) realmAccess.getOrDefault("roles",
Collections.emptyList()); ①
        List<GrantedAuthority> authorities = roles.stream()
            .map(roleName -> "ROLE_" + roleName)
            .map(SimpleGrantedAuthority::new) ②
            .collect(Collectors.toList());
        return new JwtAuthenticationToken(jwt, authorities); ③
    }

}
```

① Defensively, we extract first the `realm_access` claim, and then the `roles` contained within.

② Any roles are prefixed with `ROLE_` and converted into granted authorities.

③ Finally, we return a complete Authentication object for use in our application.

When you add this final piece to your application code, all we need to do is pass in the claim in our tests.

In the `LeaveRequestControllerSpringBootWebEnvRandomPortTest.AuthorizeRole` add the claim `realm_access` with the correct "HR" role.

JwtDecoder mocked response of JWT with HR role.

```
@Nested
class AuthorizeRole {

    @BeforeEach
    void beforeEach() {
        when(jwtDecoder.decode(anyString()))
            .thenReturn(Jwt.withTokenValue("token")
                .subject("bob") ①
                .header("alg", "none")
                .claim("realm_access", Collections.singletonMap("roles",
Collections.singletonList("HR")))
                .build());
    }

    // ...
}
```

The `LeaveRequestControllerSpringBootWebEnvRandomPortTest.AuthorizeRole` tests that we postponed previously now pass. These are the only tests that check the token claim mapping as part of their execution, so there is some value in these tests after all.

3.8. Verification

At this stage you can choose to revisit the JWTs shown previously for both Alice and Bob from HR, as well as the HTTPie / curl commands from before. All web requests should now be working as expected, provided you pass in the correct Bearer token.

3.9. Conclusion

We have now added Spring Security to our application, to allow any user to authenticate, while only granting some users special privileges. We worked through the challenges of picking the right dependency, and iteratively solving our tests failures as we first added authentication, and then authorization. We saw how some test approaches require less or more effort once you start addition security to your application, so take this into account when designing your tests.

If you got stuck at any point, leaveapp-complete shows the application in a final form, with `src/main` and `src/test` updated to the above specification.

Your implementation could of course differ from ours; It'll be interesting to compare your approach with ours!

3.10. What's next?

Congratulations, you solved the first challenge!

You can now choose to:

- automatically track who modifies an entry, and when
- limit your query results to the active user
- restrict which users can access what objects
- separate read and write permissions on objects
- route requests through a gateway

4. Auditing Spring Data Entities

The goal of this workshop is to have JPA link BlogPosts to the correct Author, based on the active user.

A first draft of the entities and repositories has been provided; Up to you to wire up auditing!

4.1. Getting things done

- Make sure the entities are auditable with the `AuditingEntityListener`
- Add proper configuration to authorize the author to post a blog
- Verify your implementation by running the tests

4.2. References

- Spring Data JPA Reference on Auditing

4.3. Solution

https://github.com/jdriven/spring-security-samples/tree/main/audit-spring-data-entities

5. Securing Spring Data methods

The goal of this workshop is to integrate Spring Data with Spring Security, such that we do not have to explicitly pass our active user as argument when retrieving user preferences.

A PreferencesRepository has been provided with annotated methods containing placeholder security expressions; Up to you to implement these and have them checked by Spring Security.

5.1. Getting things done

- PreAuthorize, PostAuthorize and configure the application enabling the user to save and find preferences by id
- Now make sure only the preferences are returned for the authorized user
- Run the tests to verify your implementation

5.2. References

- Method Security Expressions
- Spring Data & Spring Security Configuration

5.3. Solution

https://github.com/jdriven/spring-security-samples/tree/main/limit-spring-data-queries

6. Custom Access Decision Voter

The goal of this workshop is to restrict access to spreadsheets based on the active user.

The access permissions are stored in a repository; Up to you to verify access through a voter!

6.1. Getting things done

- First we are going to add a `SpreadsheetAccessDecisionVoter` to the application as described here. We'll complete the implementation of the `hasSpreadsheetAccess` method in step 3.
- Now we have to configure the application properly, so the voter will be picked up by the configuration as described here
- Now we go back to our `SpreadsheetAccessDecisionVoter` and implement the `hasSpreadsheetAccess` method so our test will succeed.Don't forget to make use of the access store to verify the access. Some hints can be found here

6.2. References

- AuthorizationManager

6.3. Solution

https://github.com/jdriven/spring-security-samples/tree/main/access-decision-voter

include::permission-evaluator/README.adoc[leveloffset=+1 :leveloffset: +1

Spring Cloud Gateway with OpenID Connect and Token Relay

The goal of this workshop is to secure a gateway with OpenID, and have JSON Web tokens relayed to backend services. An OpenID provider, gateway and two backend services have been provided; Up to you to connect the gateway and backend services to Keycloak, and have the Gateway propagate tokens to the backend.

This workshop requires an OpenID Connect provider, client and user to complete. We cover Keycloak setup in our example.

1. Getting things done

- Start a local Keycloak instance with Docker
- Update the `SecurityConfig` in the `travel-gateway` module
- Update the `application.yml` with the proper keycloak settings and properly configure the gateway to replay the tokens
- Now we finished the configuration for the `travel-gateway`, continue with the hotels and flights services
- Start the flights, hotels and gateway applications and open the webpage. You should see a login screen and once logged in you can navigate through the hotels and flights pages.

2. References

- Spring Cloud Gateway
- OAuth 2.0 Login
- OAuth 2.0 Resource Server

3. Solution

https://github.com/jdriven/spring-security-samples/tree/main/spring-cloud-gateway-oidc-tokenrelay