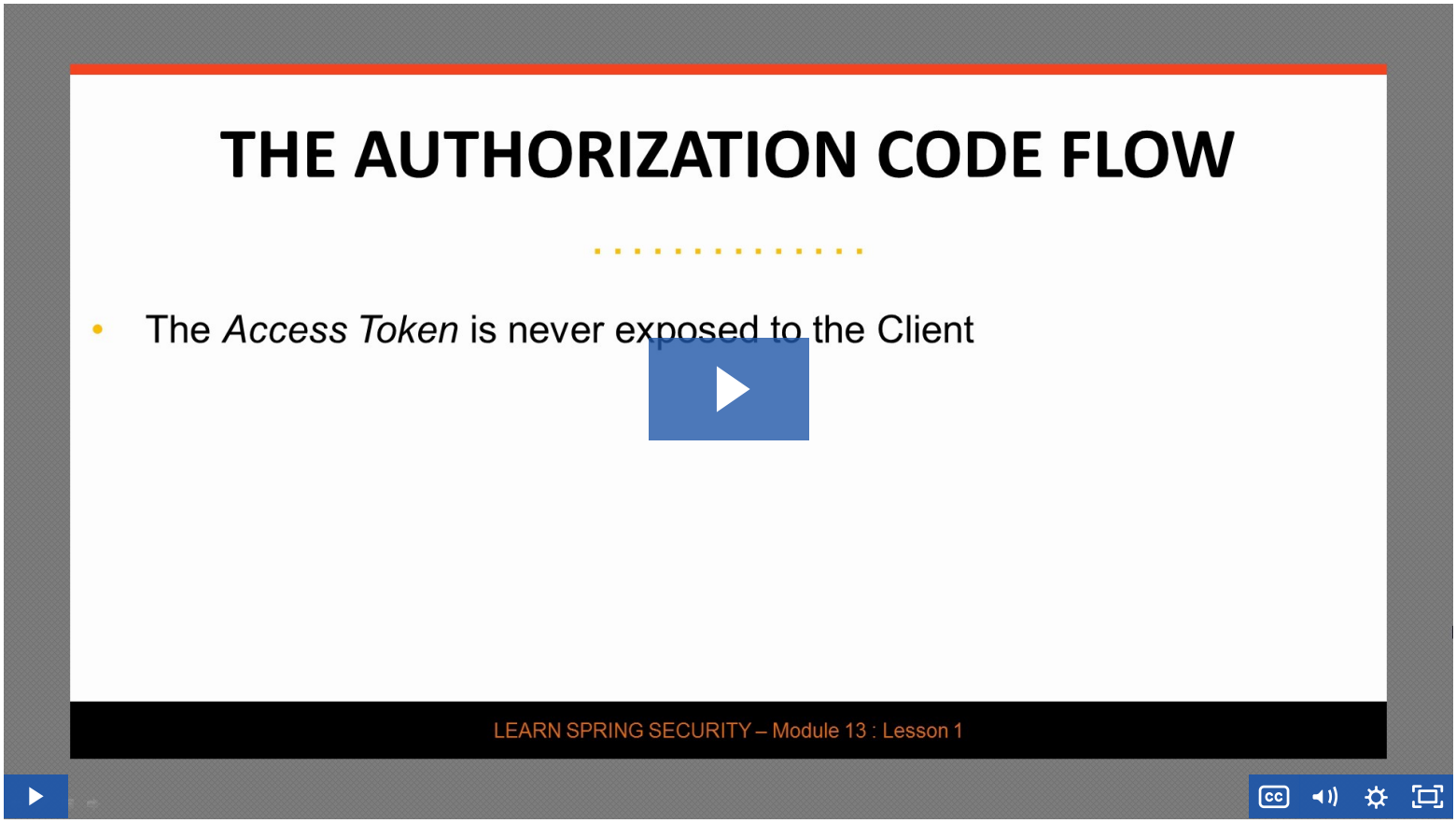
Lesson 1: The OAuth2 Implicit Flow and the Authorization Code Flow



1. Goal

The simple goal of this lesson is to introduce you to the redirect based flows in OAuth2.

2. Lesson Notes

This lesson is theoretical and doesn't have/need accompanying code.

2.1. High Level View of the Redirect Flows

Let's take a step back - and talk about what OAuth2 tries to achieve with its redirect flows.

First, OAuth2 needs to enable a flow that involves a redirect URI but doesn't run over HTTPS (and is still safe).

Second - it needs to deal with both:

- clients that have a server side component
- and clients which are client-side only and have no server side.

Which is why **we have 2 flows that involve redirects available**.

We have the Authorization Code Flow - to allow non-HTTPS scenarios and works with Clients with a server side component.

And we have the **Implicit Flow** - to allow Clients with no server side.

At a high level, these flow can be similar - up to a point:

- the user will be redirected in a browser to the Authorisation Server
- sign in
- authorize the request

But, in the Implicit Flow - instead of being returned to the client with an *Authentication Code* - they're redirected with an *Access Token* straight away.

2.2. The Authorization Code Flow

This flow is well suited for Clients that has a server side component.

The Access Token is never exposed to the Client / Resource Owner.

This is partly why this flow allows non-HTTPS scenarios.

At a high level, an intermediary, one-time-use *Authorization Code* is provided (instead of the *Access Token*).

And what's critical to understand is that only a legitimate receiver will be able to use this code - because you need the client secret.

That code will be useless to potential hackers intercepting requests over unencrypted transactions (because they don't know the client secret).

2.3. The Implicit Flow

This flow is well suited for for public, JS enabled Clients with no server-side component.

These Clients generally run on the device of the Resource Owner.

Because everything runs on the user's client, there's **no need for a Client authentication**; so there's no client secret - because it would be exposed anyways.

Now, because in this flow, the Access Token does reach the client side - the flow is somewhat less secure.

And because of that fact, it only grants Access Tokens (not Refresh Tokens).

2.4. Access Tokens and Fragments

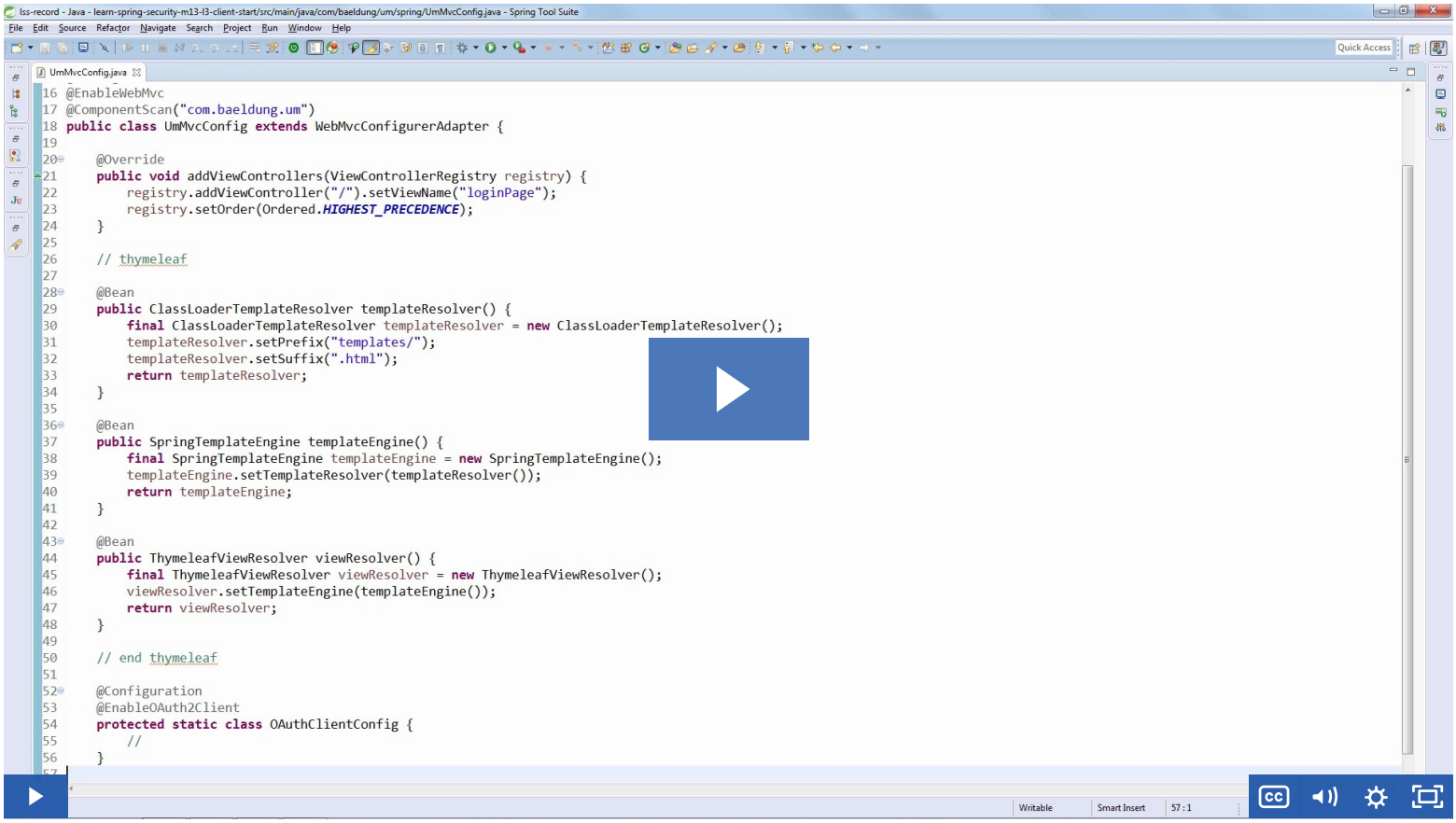
The *Access Token* is obtained through the authorization request - not with a separate request as in the Authorization Code Flow.

The *Access Token* is encoded in the redirection URI; it's **passed directly as a hash fragment** - not as an URL parameter.

That's important because, fragment are only accessible on the client side; they're not passed to the Server.

This makes it possible to pass an *Access Token* directly to the Client, without risking it being intercepted by intermediary Servers.

Lesson 2: Using the Authorization Code Flow in OAuth2



1. Goal

In this lesson we're going to go through an OAuth2 implementation using the Authorization Code Flow.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m13-lesson2-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m13-lesson2

New: An alternative version of the codebase for this lessons, where the OAuth2 Authorization Server and the OAuth2 Resource Server are independent is available here.

Let's start with a high level look at the flow:

- the User authenticates and a code is returned to the Client
- the Client exchanges this code for an Access Token - authenticating itself with a client id and client secret
- the Client then can call the API with the Access Token

2.1. The Server Impl

Let's start with the Authorization Server configuration - and more specifically - with defining a new Client:

```
@Override
public void configure(final ClientDetailsServiceConfigurer clients) throws Exception {// @formatter:off
    clients.inMemory()
        .withClient("lssClient")
        .secret("lssSecret")
        .authorizedGrantTypes("authorization_code", "refresh_token")
        .scopes("read","write");
}// @formatter:on
```

So, a straightforward Client implementation using the Authorization Code flow.

Next, let's touch on the API we're exposing. We're going to use the same, simple Users REST API we had and used in the last lesson as well.

And, next - let's also configure the Resource Server:

```
@Override
public void configure(HttpSecurity http) throws Exception {// @formatter:off
    http
    .requestMatchers().antMatchers("/api/users/**")
    .and()
    .authorizeRequests()
    .antMatchers(HttpMethod.GET,"/api/users/**").access("#oauth2.hasScope('read')")
    .antMatchers(HttpMethod.POST,"/api/users/**").access("#oauth2.hasScope('write')")
    .antMatchers(HttpMethod.DELETE,"/api/users/**").access("#oauth2.hasScope('write')");
}// @formatter:on
```

And finally, we're going to configure our web security.

We're starting from a bare-bones configuration - *UmSecurityConfig* - and defining a simple in memory user details (because we don't need anything else):

```
@Autowired
private UserDetailsService userDetailsService;
...
@Autowired
public void globalUserDetails(final AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService);
}
```

We're also defining the form based auth:

```
@Override
protected void configure(final HttpSecurity http) throws Exception { // @formatter:off
    http.authorizeRequests()
        .anyRequest().authenticated()
        .and().formLogin().permitAll()
        .and().csrf().disable()
        ;
} // @formatter:on
```

So, basically, the server configuration isn't very different than what we've done and seen before.

Finally, let's remove the security disable config from *application.properties*:

```
# security.basic.enabled=false
```

2.2. The Client Impl

Now, let's start working on the Client.

First, as you can see, we're now defining the Client in a different module - that's simply to keep things entirely separate and actually simulate a real-world scenario.

We're going to take advantage of yet another super useful Spring Security annotation - *@EnableOAuth2Client*.
This was built specifically for Client applications that want to use Spring Security and that are interacting with an Authorization Server using the Authorization Code flow.

So let's first use that and define a bare-bones config:

```
@Configuration
@EnableOAuth2Client
protected static class OAuthClientConfig {
    //
}
```

Now, because our Client application is separate from the API, we're going to have to consume the secured API from the Client - more specifically from the Server component of the Client Application.

We're going to need a simple HTTP Client that's also OAuth aware:

```
@Bean
public OAuth2RestTemplate redditRestTemplate(final OAuth2ClientContext clientContext) {
    return new OAuth2RestTemplate(resourceDetails(), clientContext);
}
```

And now let's define these resource details:

Again we're using a very focused class to represent that - a class that's specifically built for the Authorization Code flow - *AuthorizationCodeResourceDetails*:

```
@Bean
public OAuth2ProtectedResourceDetails resourceDetails() {
    AuthorizationCodeResourceDetails details = new AuthorizationCodeResourceDetails();
    details.setClientId("lssClient");
    details.setClientSecret("lssSecret");
    details.setAccessTokenUri("http://localhost:8081/um-webapp/oauth/token");
    details.setUserAuthorizationUri("http://localhost:8081/um-webapp/oauth/authorize");
    details.setScope(Arrays.asList("read", "write"));
    details.setGrantType("authorization_code");
    details.setUseCurrentUri(true);
    return details;
}
```

Notice a few things here:

- we're first defining the client credentials of course
- we're then defining 2 URIs - the token URI and the URI to which the user is to be redirected to authorize an Access Token

And now, with everything set up here - we're going to use the new HTTP Client and we're going to basically try to consume the secured API.

So, this is how everything starts:

- we're going to hit /users from the login page
- and the entire flow is going to start

Finally, let's have a look at the Servlet configuration (notice we're using an initializer here - ServletInitializer):

```
registerProxyFilter(servletContext, "oauth2ClientContextFilter");
```

2.3. Demo

Let's run and go through the full flow:

http://localhost:8082/um-webapp-client/

We start the authentication process by using *Login* (which is going to hit */user*).

We get redirected to the authorization URI (with a redirect uri param back to the client).

And because we're not yet authenticated in the Authorization Server - we're promoted to authenticate (form login).

We authenticate - and we get redirected back to the authorization URI.

And now, because we're already authenticated - we get redirected to the initial redirect uri parameter.

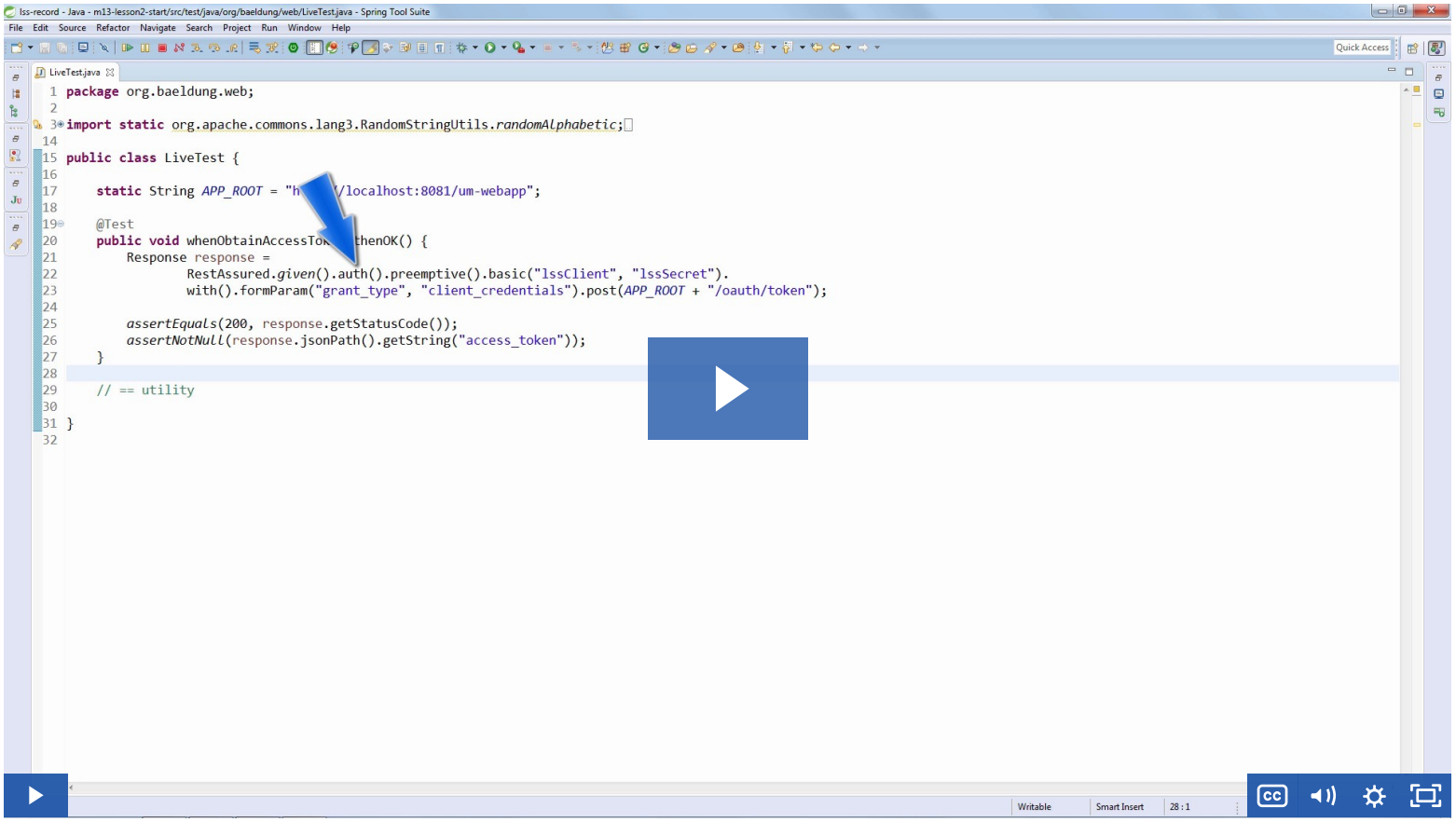
So, we get back to the Client application.

2.4. Ent-to-end Test

Finally, a good way to really understand the flow is to see a full live test going through the entire process.

The test will obtain the Authorization Code, then the Access Code, and then use the Access Code to consume the API.

Lesson 3: Confidential Clients and the Client Credentials Flow



1. Goal

The simple goal of this lesson is to go through an implementation of the Client Credentials flow with Spring Security.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m13-lesson3-start

If you want to skip and see the complete implementation, feel free to jump ahead and import: m13-lesson3

2.1. The Impl

We're going to start from a simple OAuth skeleton configuration.

And we're first going to define a client - using the Client Credentials flow:

```
clients
  .inMemory().withClient("lssClient")
  .secret("lssSecret")
  .authorizedGrantTypes("client_credentials")
  .scopes("read","write")
;
```

Notice how this is of course very similar to all the other definitions of clients we had in previous lessons.

Let's now have a look at the Resource Server side of the configuration - and let's see what kind of access we're asking for:

```
http.authorizeRequests()
  .antMatchers(HttpMethod.GET,"/api/user/**").access("#oauth2.hasScope('read')")
  .antMatchers(HttpMethod.POST,"/api/user/**").access("#oauth2.hasScope('write')")
  .antMatchers(HttpMethod.DELETE,"/api/user/**").access("#oauth2.hasScope('write')")
;
```

So, we're basically asking for the *read* scope on a GET operation and for the *write* scope on a POST or a DELETE.

2.2. The Tests

Now, we have our standard */users* API - that we've been using throughout the course, and we're going to write a few live tests to basically consume the API and of course, more importantly - go through the process of obtaining the token:

```
@Test
public void whenObtainAccessToken_thenOK() {
    Response response = RestAssured.given()
      .auth().preemptive().basic("lssClient", "lssSecret")
      .with().formParam("grant_type", "client_credentials").post(APP_ROOT + "/oauth/token");
    assertEquals(200, response.getStatusCode());
    assertNotNull(response.jsonPath().getString("access_token"));
}
```

So, what are we doing here exactly?

We're hitting the */oauth/token* endpoint - which is of course secured with Basic Auth.

We are specifying the type of flow we're using, but - what's important is that we're not passing ANY user credentials here - which is essentially all due to the flow we're using.

Alright, let's now write a test that's actually consuming the API, not just getting a token:

```
@Test
public void givenReadAndWriteScope_whenAccessUsers_thenOK() {
    String token = obtainAccessToken("lssClient", "lssSecret");
    Response readResponse = RestAssured.given().
      header("Authorization", "Bearer " + token).get(APP_ROOT + "/api/user");
    assertEquals(200, readResponse.getStatusCode());
```

```
    Map<String, String> params = createRandomUser();
    Response writeResponse = RestAssured.given().
      header("Authorization", "Bearer " + token).
      formParameters(params).post(APP_ROOT + "/api/user");
    assertEquals(201, writeResponse.getStatusCode());
}
```

And the utility methods:

```
private String obtainAccessToken(String clientId, String secret) {
    Response response = RestAssured.given()
      .auth().preemptive().basic(clientId, secret)
      .with().formParam("grant_type", "client_credentials").post(APP_ROOT + "/oauth/token");
    return response.jsonPath().getString("access_token");
}
private Map<String, String> createRandomUser() {
    Map<String, String> params = new HashMap<String, String>();
    params.put("email", randomAlphabetic(4) + "@test.com");
    String password = randomAlphabetic(8);
    params.put("password", password);
    params.put("passwordConfirmation", password);
    return params;
}
```

So, we're doing a few things here.

First, we're of course obtaining the token - just like we did in the previous test. Notice of course that we've abstracted the operation away - just for simplicity.

Then, we're doing a get and retrieving users - basically we want to check that we have read access - remember that GET was requiring the read scope.

Then, we're also doing a write, to make sure we have that kind of access as well. We're essentially creating a user.

And there we go - we did configure our client with both read and write access - so the test should pass just fine.

2.3. More Clients

Alright, let's now actually define a couple more clients in the Auth Server config.

We're going to do a read client and a write client. And then we're going to built out some tests around these specific clients, to make sure that they actually work as we expected them to.

Here we go:

```
@Test
public void givenReadScope_whenAccessUsers_thenCanReadOnly() {
    String token = obtainAccessToken("lssReadOnly", "lssReadSecret");
    Response readResponse = RestAssured.given()
      .header("Authorization", "Bearer " + token).get(APP_ROOT + "/api/users");
    assertEquals(200, readResponse.getStatusCode());
    Map<String, String> params = createRandomUser();
    Response writeResponse = RestAssured.given()
      .header("Authorization", "Bearer " + token).formParameters(params).post(APP_ROOT + "/api/users");
    assertEquals(403, writeResponse.getStatusCode());
}
```

3. Resources

- Spring Boot and OAuth2