Secure Spring REST With Spring Security and OAuth2

by Adam Zaręba MVB · **Feb. 23, 18 · Security Zone · Tutorial**

In this post, we are going to demonstrate Spring Security + OAuth2 for securing REST API endpoints on an example Spring Boot project. Clients and user credentials will be stored in a relational database (example configurations prepared for H2 and PostgreSQL database engines). To do it we will have to:

- Configure Spring Security + database.
- Create an Authorization Server.
- Create a Resource Server.
- Get an access token and a refresh token.
- Get a secured Resource using an access token.

To simplify the demonstration, we are going to combine the Authorization Server and Resource Server in the same project. As a grant type, we will use a password (we will use BCrypt to hash our passwords).

Before you start you should familiarize yourself with OAuth2 fundamentals.

Introduction

The OAuth 2.0 specification defines a delegation protocol that is useful for conveying authorization decisions across a network of web-enabled applications and APIs. OAuth is used in a wide variety of applications, including providing mechanisms for user authentication.

OAuth Roles

OAuth specifies four roles:

- **Resource owner (the User)** – an entity capable of granting access to a protected resource (for example end-user).
- **Resource server (the API server)** – the server hosting the protected resources, capable of accepting responding to protected resource requests using access tokens.
- **Client** – an application making protected resource requests on behalf of the resource owner and with its authorization.
- **Authorization server** – the server issuing access tokens to the client after successfully

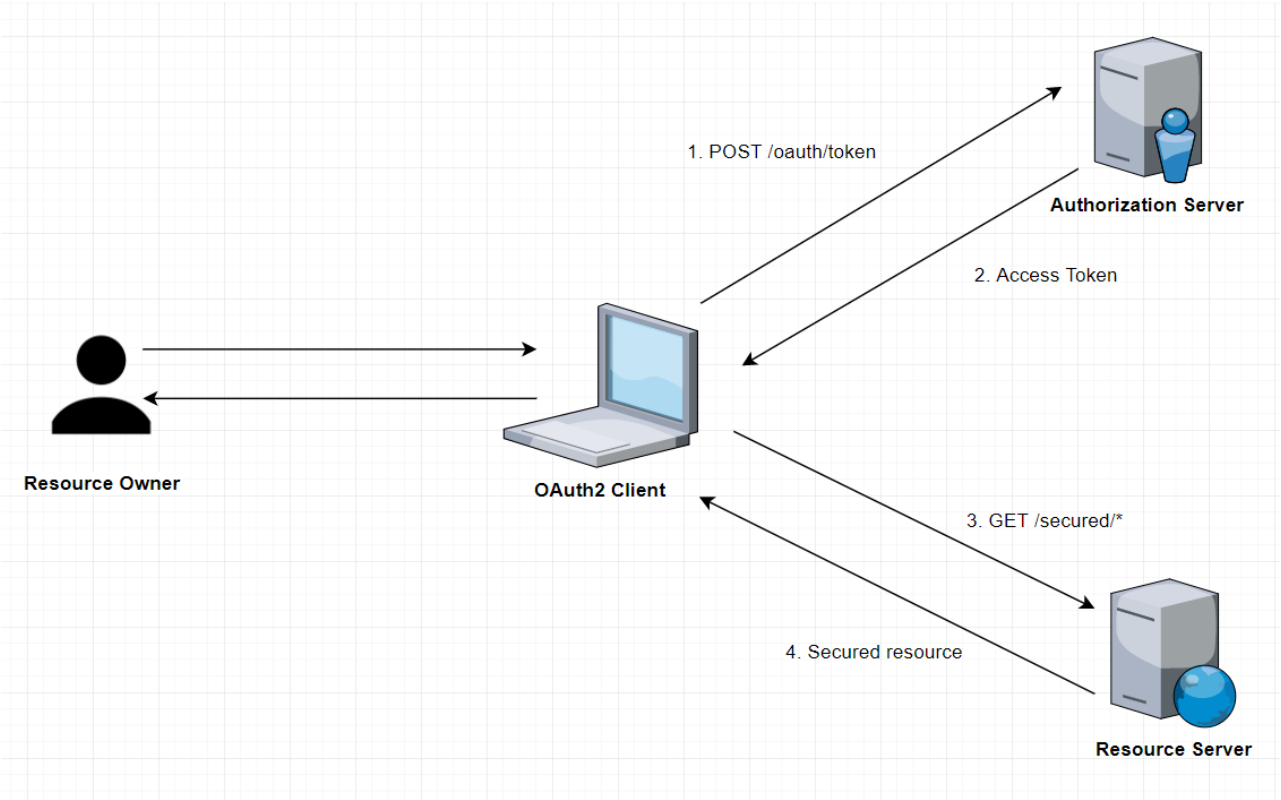
- **Authorization server** – the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Grant Types

OAuth 2 provides several "grant types" for different use cases. The grant types defined are:

- **Authorization Code**
- **Password**
- **Client credentials**
- **Implicit**

The overall flow of a Password Grant:

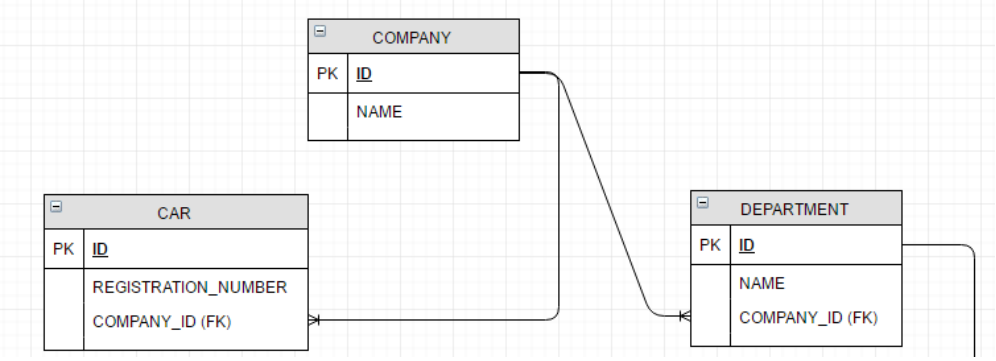


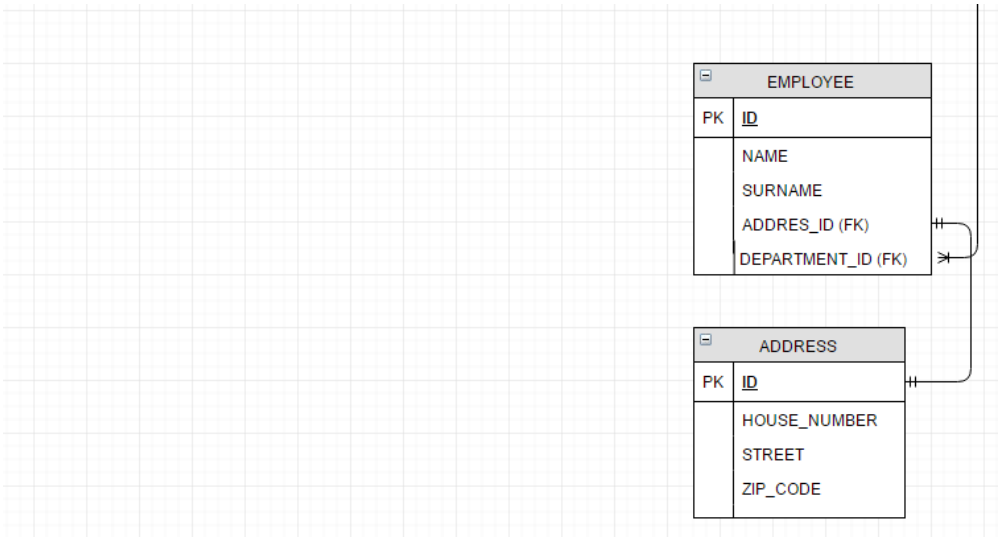
Application

Let's consider the database layer and application layer for our example application.

Business Data

Our main business object is `Company`:





Based on CRUD operations for Company and Department objects ,we want to define following access rules:

- COMPANY_CREATE
- COMPANY_READ
- COMPANY_UPDATE
- COMPANY_DELETE
- DEPARTMENT_CREATE
- DEPARTMENT_READ
- DEPARTMENT_UPDATE
- DEPARTMENT_DELETE

In addition, we want to create ROLE_COMPANY_READER role.

OAuth2 Client Setup

We need to create the following tables in the database (for internal purposes of OAuth2 implementation):

- OAUTH_CLIENT_DETAILS
- OAUTH_CLIENT_TOKEN
- OAUTH_ACCESS_TOKEN
- OAUTH_REFRESH_TOKEN
- OAUTH_CODE
- OAUTH_APPROVALS

Let's assume that we want to call a resource server like 'resource-server-rest-api.' For this server, we define two clients called:

- spring-security-oauth2-read-client (authorized grant types: read)
- spring-security-oauth2-read-write-client (authorized grant types: read, write)

```
INSERT INTO OAUTH_CLIENT_DETAILS(CLIENT_ID, RESOURCE_IDS, CLIENT_SECRET, SCOPE, AUTHORIZED_
VALUES ('spring-security-oauth2-read-client', 'resource-server-rest-api',
/*spring-security-oauth2-read-client-password1234*/'$2a$04$WGq2P9egiOYoOFemBRfsiO9qTcyJtNR
'read', 'password,authorization_code,refresh_token,implicit', 'USER', 10800, 2592000);

INSERT INTO OAUTH_CLIENT_DETAILS(CLIENT_ID, RESOURCE_IDS, CLIENT_SECRET, SCOPE, AUTHORIZED_
VALUES ('spring-security-oauth2-read-write-client', 'resource-server-rest-api',
/*spring-security-oauth2-read-write-client-password1234*/'$2a$04$soeOR.QFmClXeFIrhJVLWOQxf
'read,write', 'password,authorization_code,refresh_token,implicit', 'USER', 10800, 259200
```

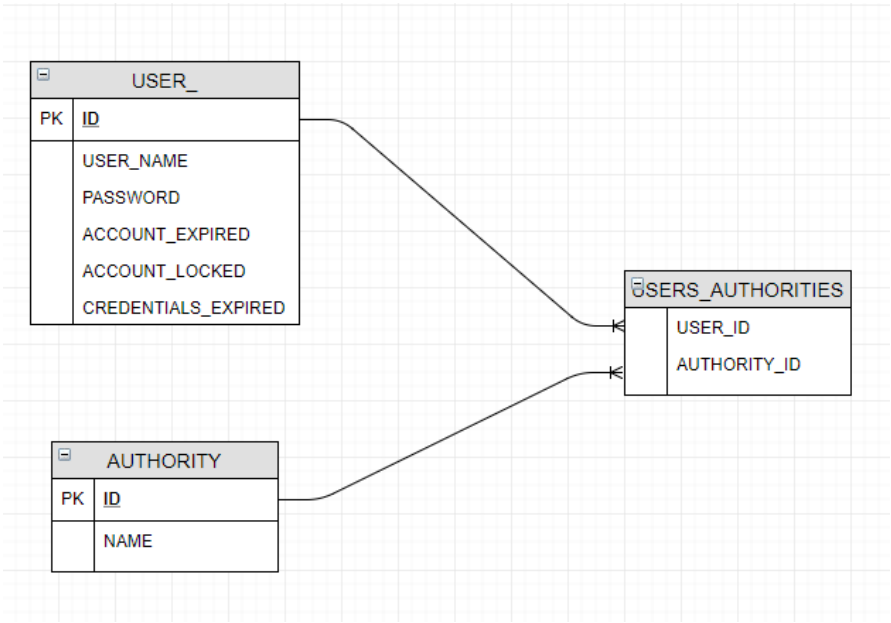
Note that password is hashed with BCrypt (4 rounds).

Authorities and Users Setup

Spring Security comes with two useful interfaces:

- UserDetails - provides core user information.
- GrantedAuthority - represents an authority granted to an Authentication object.

To store authorization data we will define following data model:



Because we want to come with some pre-loaded data, below is the script that will load all authorities:

```
INSERT INTO AUTHORITY(ID, NAME) VALUES (1, 'COMPANY_CREATE');
```

```
INSERT INTO AUTHORITY(ID, NAME) VALUES (2, 'COMPANY_READ');
INSERT INTO AUTHORITY(ID, NAME) VALUES (3, 'COMPANY_UPDATE');
INSERT INTO AUTHORITY(ID, NAME) VALUES (4, 'COMPANY_DELETE');

INSERT INTO AUTHORITY(ID, NAME) VALUES (5, 'DEPARTMENT_CREATE');
INSERT INTO AUTHORITY(ID, NAME) VALUES (6, 'DEPARTMENT_READ');
INSERT INTO AUTHORITY(ID, NAME) VALUES (7, 'DEPARTMENT_UPDATE');
INSERT INTO AUTHORITY(ID, NAME) VALUES (8, 'DEPARTMENT_DELETE');
```

Here is the script to load all users and assigned authorities:

```
INSERT INTO USER_(ID, USER_NAME, PASSWORD, ACCOUNT_EXPIRED, ACCOUNT_LOCKED, CREDENTIALS_EXP
  VALUES (1, 'admin', /*admin1234*/'$2a$08$qvrzQZ7jJ7oy2p/msL4M0.l83Cd0jNsX6AJUitbgRXGzge4j

INSERT INTO USER_(ID, USER_NAME, PASSWORD, ACCOUNT_EXPIRED, ACCOUNT_LOCKED, CREDENTIALS_EXP
  VALUES (2, 'reader', /*reader1234*/'$2a$08$dwYz8O.qtUXboGosJFsS4u19LHKW7aCQ0LXXuNlRfjjGKw

INSERT INTO USER_(ID, USER_NAME, PASSWORD, ACCOUNT_EXPIRED, ACCOUNT_LOCKED, CREDENTIALS_EXP
  VALUES (3, 'modifier', /*modifier1234*/'$2a$08$kPjzxewXRGNRiIuL4FtQH.mhMn7ZAFBYKB3ROz.J24

INSERT INTO USER_(ID, USER_NAME, PASSWORD, ACCOUNT_EXPIRED, ACCOUNT_LOCKED, CREDENTIALS_EXP
  VALUES (4, 'reader2', /*reader1234*/'$2a$08$vVXqh6S8TqfHMs1SlNTu/.J25iUCrpGBpyGExA.9yI.Il

INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 1);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 2);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 3);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 4);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 5);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 6);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 7);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 8);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (1, 9);

INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (2, 2);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (2, 6);

INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (3, 3);
INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (3, 7);

INSERT INTO USERS_AUTHORITIES(USER_ID, AUTHORITY_ID) VALUES (4, 9);
```

Note that the password is hashed with BCrypt (8 rounds).

Application Layer

The test application is developed in Spring boot + Hibernate + Flyway with an exposed REST API. To demonstrate data company operations, the following endpoints were created:

```
@RestController
@RequestMapping("/secured/company")
public class CompanyController {

    @Autowired
    private CompanyService companyService;

    @RequestMapping(method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    List<Company> getAll() {
        return companyService.getAll();
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET, produces = MediaType.APPLI
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    Company get(@PathVariable Long id) {
        return companyService.get(id);
    }

    @RequestMapping(value = "/filter", method = RequestMethod.GET, produces = MediaType.APP
    @ResponseStatus(value = HttpStatus.OK)
    public @ResponseBody
    Company get(@RequestParam String name) {
        return companyService.get(name);
    }

    @RequestMapping(method = RequestMethod.POST, produces = MediaType.APPLICATION_JSON_VALU
    @ResponseStatus(value = HttpStatus.OK)
    public ResponseEntity<?> create(@RequestBody Company company) {
        companyService.create(company);
        HttpHeaders headers = new HttpHeaders();
        ControllerLinkBuilder linkBuilder = linkTo(methodOn(CompanyController.class).get(cc
        headers.setLocation(linkBuilder.toUri());
        return new ResponseEntity<>(headers, HttpStatus.CREATED);
    }

    @RequestMapping(method = RequestMethod.PUT, produces = MediaType.APPLICATION_JSON_VALUE
```

```
    @ResponseStatus(value = HttpStatus.OK)
    public void update(@RequestBody Company company) {
        companyService.update(company);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE, produces = MediaType.AP
    @ResponseStatus(value = HttpStatus.OK)
    public void delete(@PathVariable Long id) {
        companyService.delete(id);
    }
}
```

PasswordEncoders

Since we are going to use different encryptions for OAuth2 client and user, we will define separate password encoders for encryption:

- OAuth2 client password – BCrypt (4 rounds)
- User password - BCrypt (8 rounds)

```
@Configuration
public class Encoders {

    @Bean
    public PasswordEncoder oauthClientPasswordEncoder() {
        return new BCryptPasswordEncoder(4);
    }

    @Bean
    public PasswordEncoder userPasswordEncoder() {
        return new BCryptPasswordEncoder(8);
```

Spring Security Configuration

Provide UserDetailsService

Because we want to get users and authorities from the database, we need to tell Spring Security how to get this data. To do it we have to provide an implementation of the UserDetailsService interface:

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional(readOnly = true)
```

```
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
        User user = userRepository.findByUsername(username);

        if (user != null) {
            return user;
        }

        throw new UsernameNotFoundException(username);
    }
}
```

To separate the service and repository layers we will create `UserRepository` with JPA Repository:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT DISTINCT user FROM User user " +
            "INNER JOIN FETCH user.authorities AS authorities " +
            "WHERE user.username = :username")
    User findByUsername(@Param("username") String username);
}
```

Setup Spring Security

The @EnableWebSecurity annotation and WebSecurityConfigurerAdapter work together to provide security to the application. The `@Order` annotation is used to specify which WebSecurityConfigurerAdapter should be considered first.

```
@Configuration
@EnableWebSecurity
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
@Import(Encoders.class)
public class ServerSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private PasswordEncoder userPasswordEncoder;

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(userPasswordEncoder);
```

```
    }
}
```

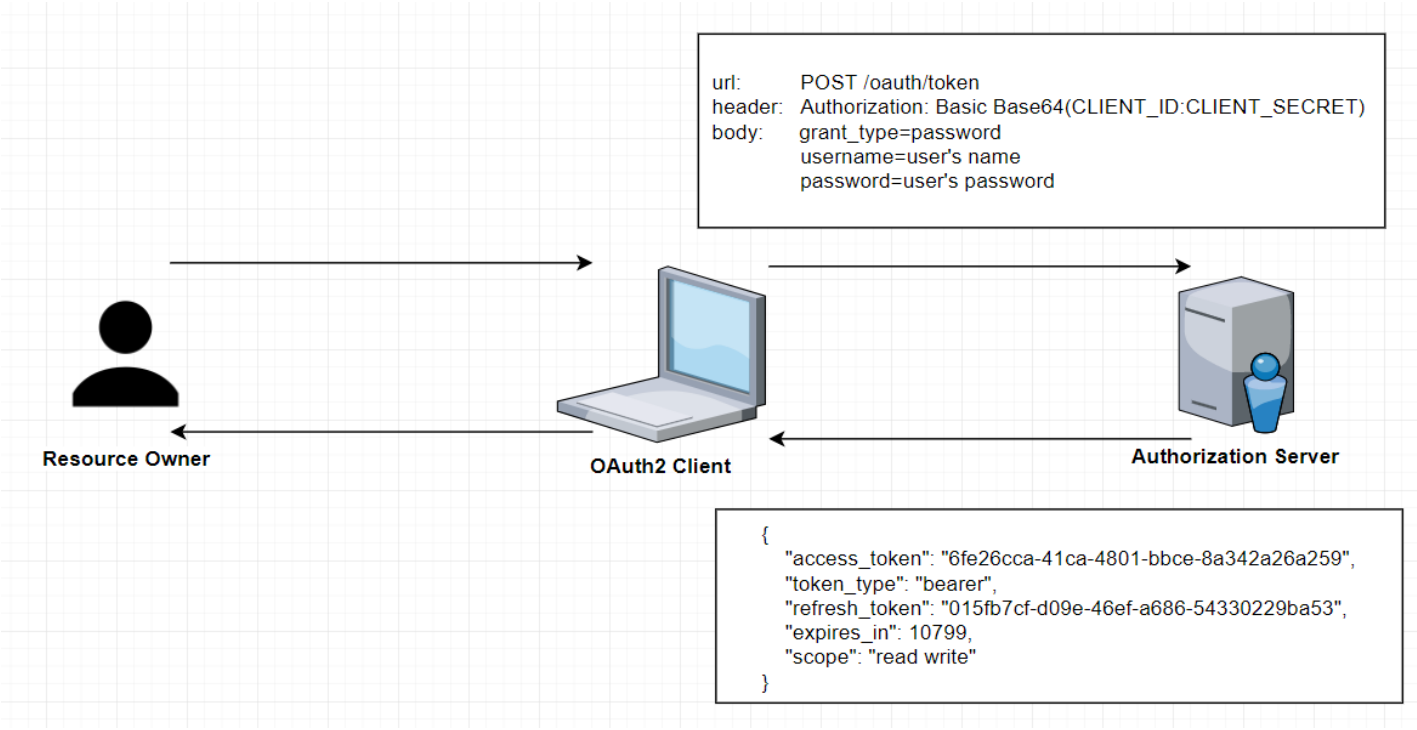
OAuth2 Configuration

First of all, we have to implement the following components:

- Authorization Server
- Resource Server

Authorization Server

The authorization server is responsible for the verification of user identity and providing the tokens.



Spring Security handles the Authentication and Spring Security OAuth2 handles the Authorization. To configure and enable the OAuth 2.0 Authorization Server we have to use @EnableAuthorizationServer annotation.

```
@Configuration
@EnableAuthorizationServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
@Import(ServerSecurityConfig.class)
public class AuthServerOAuth2Config extends AuthorizationServerConfigurerAdapter {

    @Autowired
    @Qualifier("dataSource")
    private DataSource dataSource;

    @Autowired
    private AuthenticationManager authenticationManager;
```

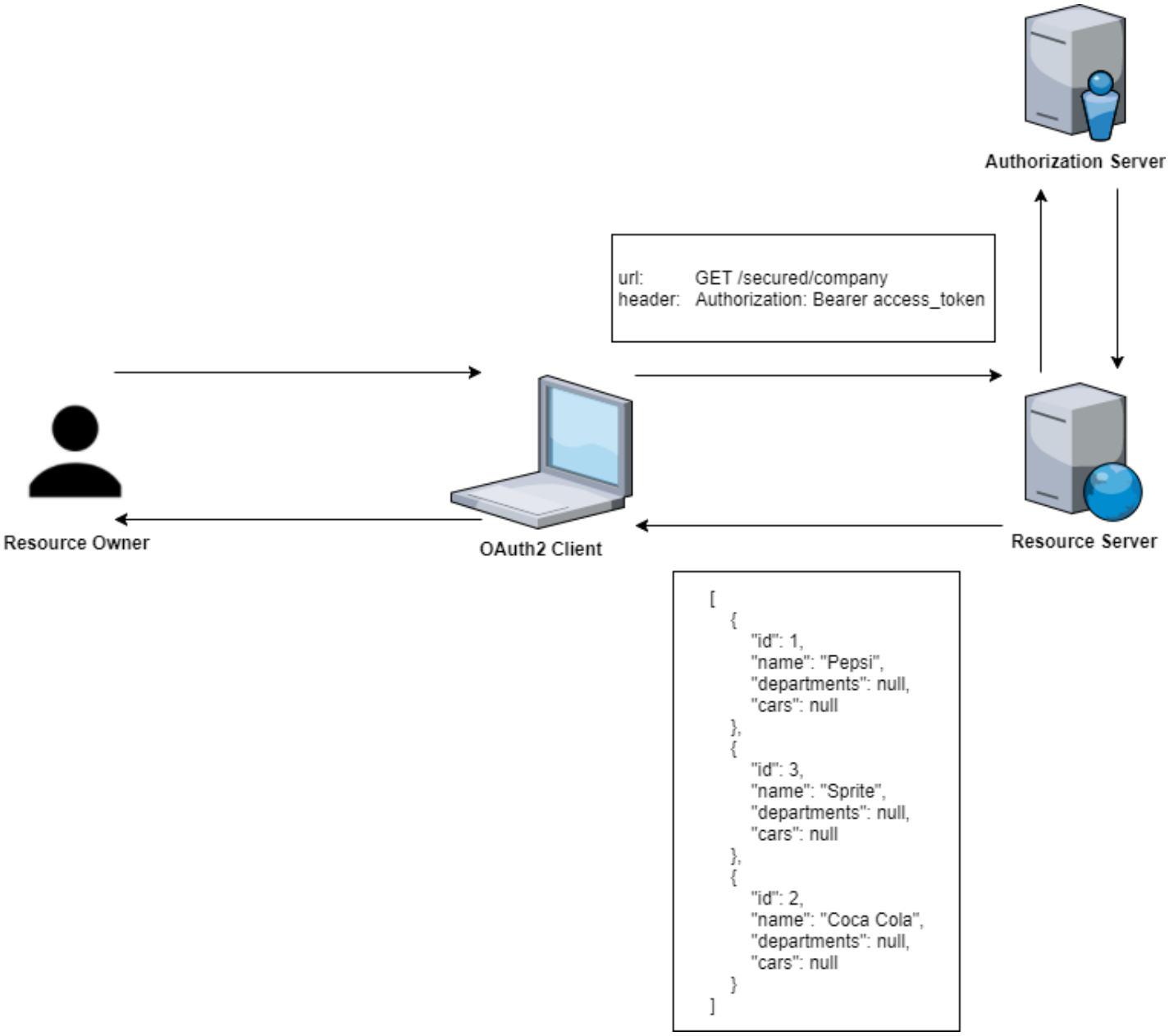
```
13
14      @Autowired
15      private UserDetailsService userDetailsService;
16
17      @Autowired
18      private PasswordEncoder oauthClientPasswordEncoder;
19
20      @Bean
21      public TokenStore tokenStore() {
22          return new JdbcTokenStore(dataSource);
23      }
24
25      @Bean
26      public OAuth2AccessDeniedHandler oauthAccessDeniedHandler() {
27          return new OAuth2AccessDeniedHandler();
28      }
29
30      @Override
31      public void configure(AuthorizationServerSecurityConfigurer oauthServer) {
            oauthServer.tokenKeyAccess("permitAll()").checkTokenAccess("isAuthenticated()").pas
32      }
33
34      @Override
35      public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
36          clients.jdbc(dataSource);
37      }
38
39      @Override
40      public void configure(AuthorizationServerEndpointsConfigurer endpoints) {
41          endpoints.tokenStore(tokenStore()).authenticationManager(authenticationManager).use
42      }
43  }
44
```

Some important points. We:

- Defined the `TokenStore` bean to let Spring know to use the database for token operations.
- Overrode the configure methods to use the custom `UserDetailsService` implementation, `AuthenticationManager` bean, and OAuth2 client's password encoder.
- Defined handler bean for authentication issues.
- Enabled two endpoints for checking tokens (/oauth/check_token and /oauth/token_key) by overriding the configure (`AuthorizationServerSecurityConfigureroauthServer`) method.

Resource Server

A Resource Server serves resources that are protected by the OAuth2 token.



Spring OAuth2 provides an authentication filter that handles protection. The @EnableResourceServer annotation enables a Spring Security filter that authenticates requests via an incoming OAuth2 token.

```
@Configuration
@EnableResourceServer
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {

    private static final String RESOURCE_ID = "resource-server-rest-api";
    private static final String SECURED_READ_SCOPE = "#oauth2.hasScope('read')";
    private static final String SECURED_WRITE_SCOPE = "#oauth2.hasScope('write')";
    private static final String SECURED_PATTERN = "/secured/**";

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.resourceId(RESOURCE_ID);
    }

    @Override
```

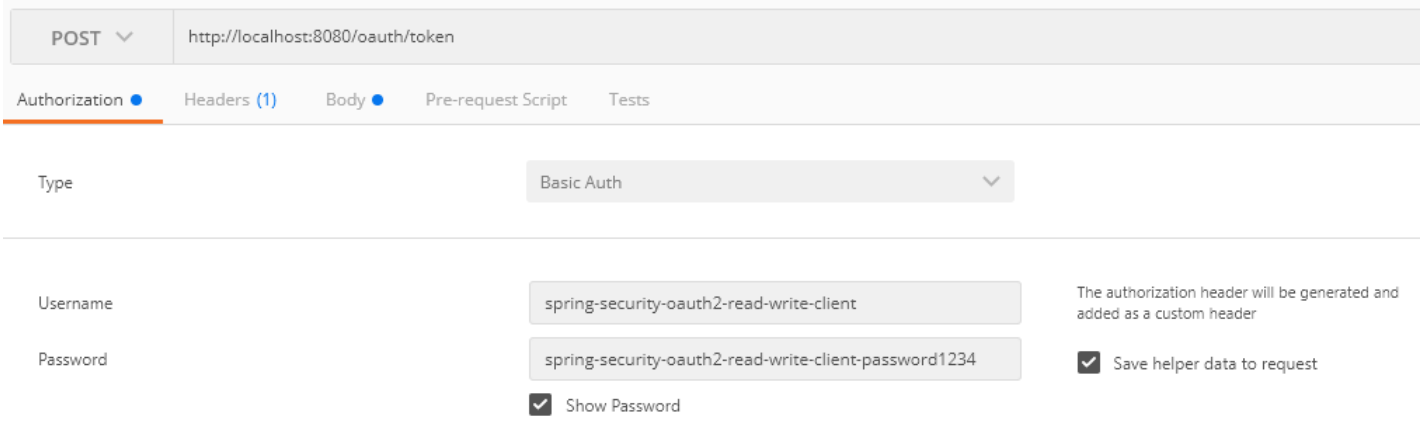
```
    public void configure(HttpSecurity http) throws Exception {
        http.requestMatchers()
                .antMatchers(SECURED_PATTERN).and().authorizeRequests()
                .antMatchers(HttpMethod.POST, SECURED_PATTERN).access(SECURED_WRITE_SCOPE)
                .anyRequest().access(SECURED_READ_SCOPE);
    }
}
```

The `configure(HttpSecurity http)` method configures the access rules and request matchers (path) for protected resources using the `HttpSecurity` class. We secure the URL paths `/secured/*`. It's worth noting that to invoke any POST method request, the 'write' scope is needed.

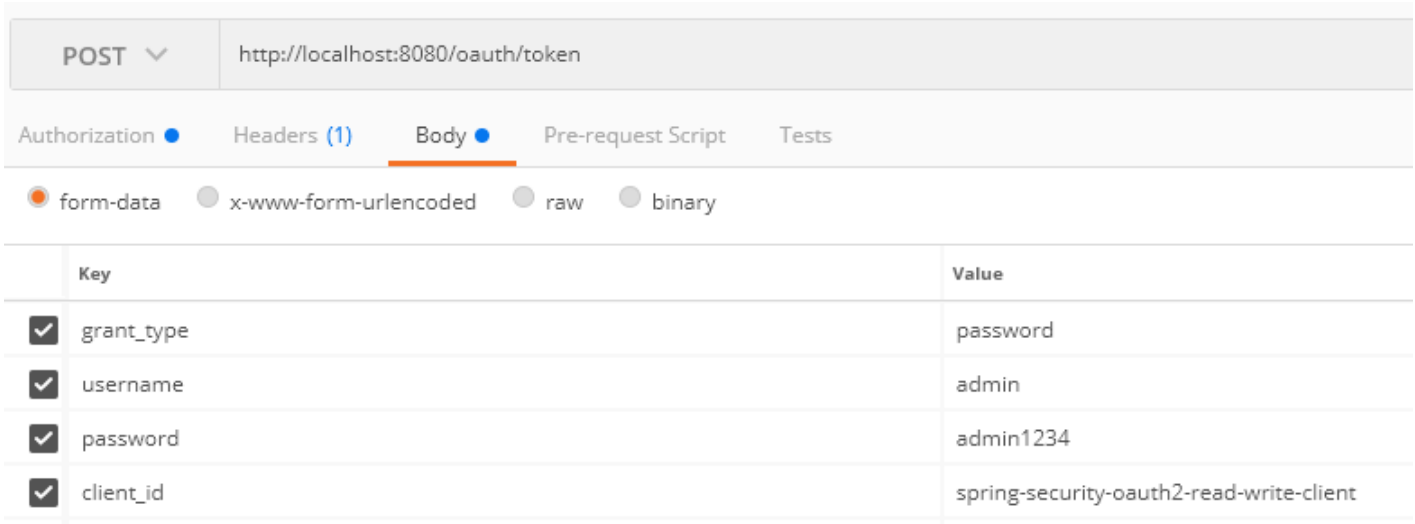
Let's check if our authentication endpoint is working – invoke:

```
curl -X POST \
  http://localhost:8080/oauth/token \
  -H 'authorization: Basic c3ByaW5nLXNlY3VyaXR5LW9hdXRoMi1yZWFkLXdyaXRlLWNsaWVudDpzcHJpbmct
  -F grant_type=password \
  -F username=admin \
  -F password=admin1234 \
  -F client_id=spring-security-oauth2-read-write-client
```

Below are screenshots from Postman:



and



You should get a response similar to the following:

```
{
    "access_token": "e6631caa-bcf9-433c-8e54-3511fa55816d",
    "token_type": "bearer",
    "refresh_token": "015fb7cf-d09e-46ef-a686-54330229ba53",
    "expires_in": 9472,
    "scope": "read write"
}
```

Access Rules Configuration

We decided to secure access to the Company and Department objects on the service layer. We have to use the @PreAuthorize annotation.

```
@Service
public class CompanyServiceImpl implements CompanyService {

    @Autowired
    private CompanyRepository companyRepository;

    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasAuthority('COMPANY_READ') and hasAuthority('DEPARTMENT_READ')")
    public Company get(Long id) {
        return companyRepository.find(id);
    }

    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasAuthority('COMPANY_READ') and hasAuthority('DEPARTMENT_READ')")
    public Company get(String name) {
        return companyRepository.find(name);
    }

    @Override
    @Transactional(readOnly = true)
    @PreAuthorize("hasRole('COMPANY_READER')")
    public List<Company> getAll() {
        return companyRepository.findAll();
    }

    @Override
    @Transactional
    @PreAuthorize("hasAuthority('COMPANY_CREATE')")
    public void create(Company company) {
        companyRepository.create(company);
    }
```

```
34
35      @Override
36      @Transactional
37      @PreAuthorize("hasAuthority('COMPANY_UPDATE')")
38      public Company update(Company company) {
39          return companyRepository.update(company);
40      }
41
42      @Override
43      @Transactional
44      @PreAuthorize("hasAuthority('COMPANY_DELETE')")
45      public void delete(Long id) {
46          companyRepository.delete(id);
47      }
48
49      @Override
50      @Transactional
51      @PreAuthorize("hasAuthority('COMPANY_DELETE')")
52      public void delete(Company company) {
53          companyRepository.delete(company);
54      }
55  }
```

Let's test if our endpoint is working fine:

```
curl -X GET \
  http://localhost:8080/secured/company/ \
  -H 'authorization: Bearer e6631caa-bcf9-433c-8e54-3511fa55816d'
```

Let's see what will happen if we authorize with it 'spring-security-oauth2-read-client' – this client has only the read scope defined.

```
curl -X POST \
  http://localhost:8080/oauth/token \
  -H 'authorization: Basic c3ByaW5nLXNlY3VyaXR5LW9hdXRoMi1yZWFkLWNsaWVudDpzcHJpbmctc2VjdXJp
  -F grant_type=password \
  -F username=admin \
  -F password=admin1234 \
  -F client_id=spring-security-oauth2-read-client
```

Then for the below request:

```
  http://localhost:8080/secured/company \
  -H 'authorization: Bearer f789c758-81a0-4754-8a4d-cbf6eea69222' \
  -H 'content-type: application/json' \
  -d '{
    "name": "TestCompany",
    "departments": null,
    "cars": null
``
```

```
}
```

We are getting the following error:

```
{
    "error": "insufficient_scope",
    "error_description": "Insufficient scope for this resource",
    "scope": "write"
}
```

Summary

In this blog post, we showed OAuth2 authentication with Spring. Access rights were defined straightforward – by establishing a direct connection between User and Authorities. To enhance this example we can add an additional entity – Role – to improve the structure of the access rights.

The source code for the above listings can be found in this GitHub project.

Like This Article? Read More From DZone

OAuth 2.0 Beginner's Guide

Developing RESTful APIs With Loopback, Part 2: Securing Your API

Getting Access Token for Microsoft Graph Using OAuth REST API, Part 1

Free DZone Refcard
REST API Security

Topics: SPRING SECRUITY , OAUTH 2.0 , SECURITY , REST API SECURITY , WEB APPLICATION SECURITY

Opinions expressed by DZone contributors are their own.

Security Partner Resources

Top 5 OWASP Resources No Developer Should Be Without
Checkmarx

The Go Language Guide: Web Application Secure Coding Practices
Checkmarx

What is the Deserialization vulnerability and what are the challenges in providing a solution
Waratek

Getting Access Token for Microsoft Graph Using OAuth REST API, Part 2

by Eran Hertz · Apr 13, 18 · Security Zone · Tutorial

Welcome back! If you missed Part 1, check it out here.

Getting the Access Token

After we registered our OAuth App, got its Client ID and Secret, and configured its permissions, we can finally use AAD Services in order to get the Access Token.

In OAuth, there are several different ways to achieve access tokens, each suited for different a scenario. Those ways are called "grant flows," and, according to the desired flow, a different message needs to be sent. Let's review our different flows.

Flow 1: Get an Access Token From Client Credentials (Client Credentials Grant)

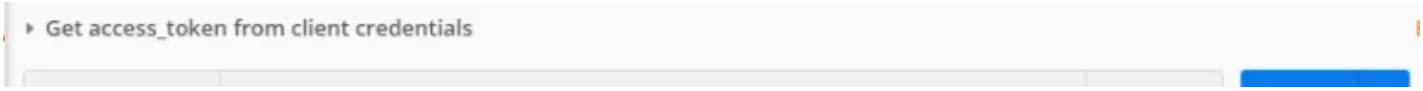
The most basic option is to use our Client ID and Secret in order to get an access token. For this, we need to send a POST message to our Azure Active Directory Authentication endpoint (which we talked about before) with following body parameters:

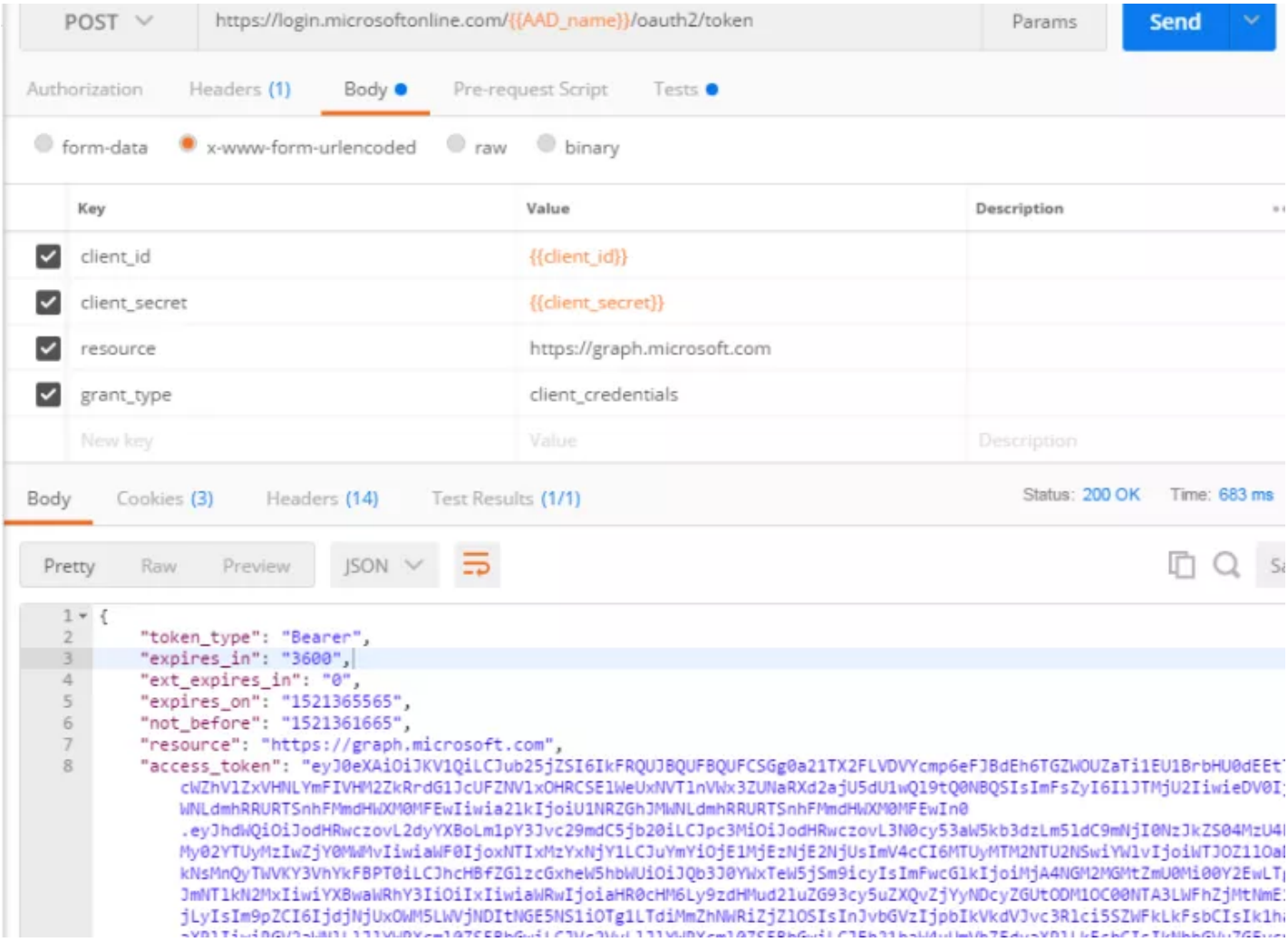
```
POST https://login.microsoftonline.com/<AAD_name>/oauth2/token
```

- `grant_type` : The flow we want to use, `client_credentials` in this case.
- `client_id` : The Client ID (Application ID) of the application we created in the previous step.
- `client_secret` : The Client Secret we created in the previous step.
- `resource` : The name of the resource we would like to get access, `https://graph.microsoft.com` in this case.

We will receive a response with a JSON object containing the following properties:

- `token_type` : The value `Bearer`
- `expires_on` : The token expire timestamp in Unix epoch time.
- `access_token` : The access token we needed to access the Graph API





This option is called **Client Credentials Grant Flow** and is suitable for machine-to-machine authentication where a specific user's permission to access data is not required.

To learn more about this flow, see: Service to service calls using client credentials (shared secret or certificate)

Flow 2 - Get Access Token From Client and User Credentials (Resource Owner Credentials Grant)

The first option, while it is the simplest of all (since it only requires the Application ID and Secret), doesn't always work. There are several Graph Methods for which just using the client credentials is not enough - they require user authorization as well. For example, in order to retrieve Group Events, we can see permission `ApplicationNot supported` , meaning getting access to that resource with just Client Credentials will not work. However, the first line is, `Delegated (work or school account): Group.Read.All` meaning, if we can get a "delegated permission" we can make this work.

So what does "delegated permission" mean, you ask? Well in simple terms, we need to show the API that not only have we come with an approved Client, we also have to carry a valid User authorization as well. Meaning that our access token needs to contain **both** a valid **Client** and **User** claims.

So how can we do that? There are a couple of ways to achieve that, in this option, we will look at the simplest way - the Resource Owner Credentials Grant.

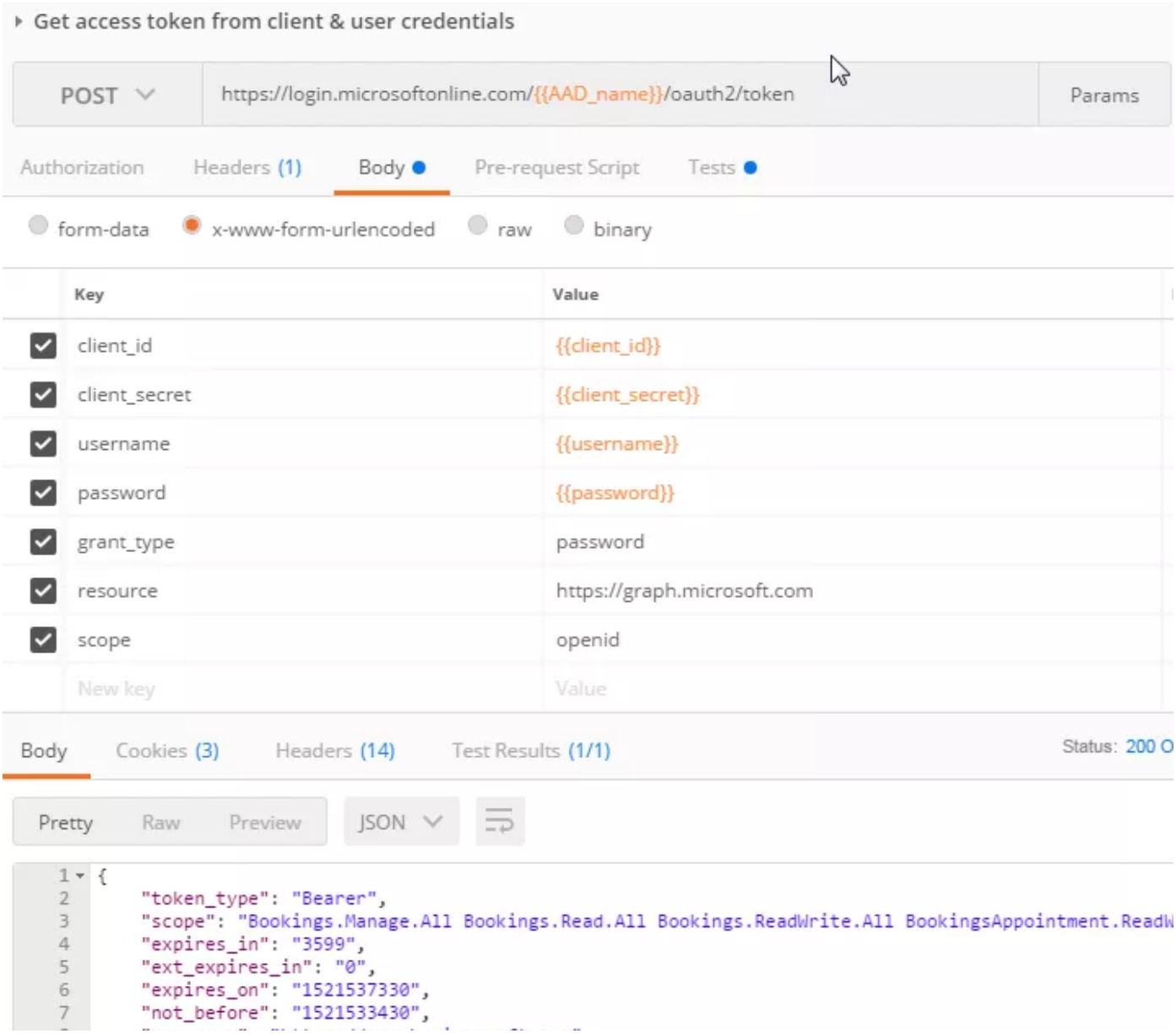
For this flow, we need to send the following POST message:

```
POST https://login.microsoftonline.com/{{AAD_name}}/oauth2/token
```

- `grant_type` : The grant flow we want to use, `password` in this case.
- `client_id` : The Client ID (Application ID) of the application we created in the previous step.
- `client_secret` : The Client Secret we created in the previous step.
- `resource` : The name of the resource to which we would like to get access, `https://graph.microsoft.com` in this case.
- `username` : Full username of the user, including the domain, for example, `john@contoso.onmicrosoft.com`
- `password` : User's plain-text password.

We will receive a response with a JSON object containing the following properties:

- `token_type` : The value `Bearer`
- `expires_on` : The token expire timestamp in Unix epoch time.
- `access_token` : The access token we needed to access the Graph API.
- `refresh_token` : A refresh token that can be used to acquire a new access token when the original expires.



```
    "resource": "https://graph.microsoft.com",
    "access_token": "eyJ0eXAiOiJKV1QiLCJub25jZSI6IkFRQUJBQUFBQUFCSGg0a21TX2FLVDVYcmp6eFJBdEh6LTRZSG9
    "refresh_token": "AQABAAAAAABHh4kmS_aKT5XrjzxRAtHz-_gaBu5XPSB4JMz2MtPxaBk0cr8aJXDLvcDruljFNJ12j3
    "id_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJhdWQiOiIyMDg0YzYwYy1mZTQyLTRjYTAtODBlNC05MTY
}
```


To learn more about this flow, see: Resource Owner Password Credentials Grant in Azure AD OAuth.

Besides the access token, we received two additional tokens - Refresh Token and ID Token. They were are not necessary for this flow, but they can be used in other grant flows and this is an example of how to get them. We automatically get the Refresh Token in this flow, and we can get an ID Token by adding to the request `scope` parameter with the value `openid`, as seen in the above Postman screenshot.

That's all for Part 2, tune in tomorrow for the final post of this series where cover the last two flow types you need!

Like This Article? Read More From DZone

OAuth Has Many Flaws But it Is the Best We Have at the Moment

OAuth 2.0 Beginner's Guide

Require ALL Platform Partners to Use APIs So There are Registered Applications

Free DZone Refcard
REST API Security

Topics: SECURITY, OAUTH, API SECURITY