Piotr's TechBlog

Part 2: Microservices security with OAuth2

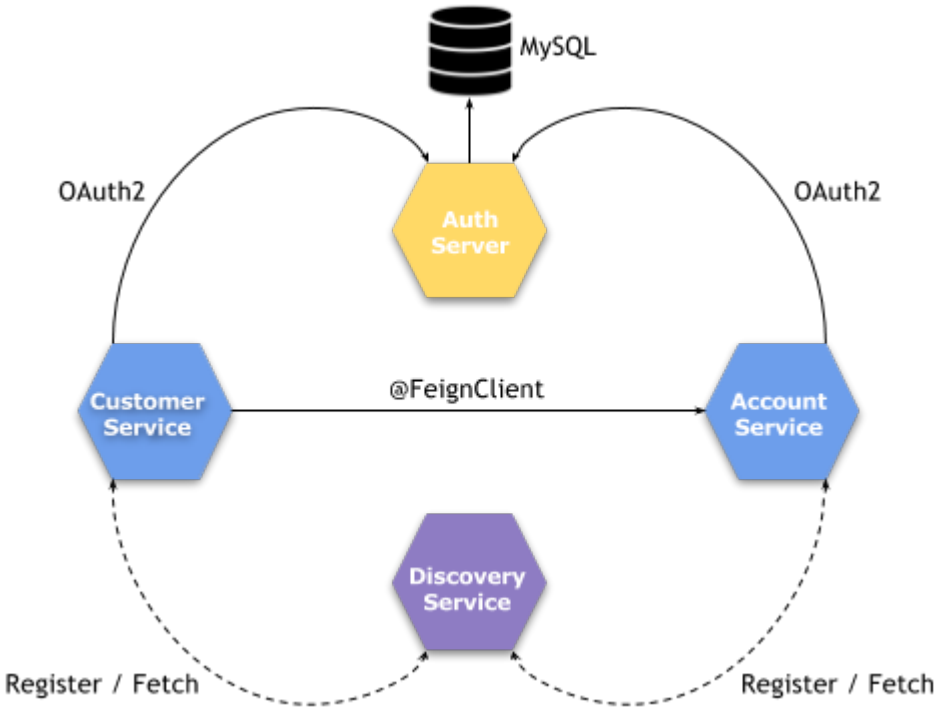
I have been writing about security with OAuth2 in some articles before. This article is the continuation of samples previously described in the following posts:

- Microservices security with Oauth2 (https://piotrminkowski.wordpress.com/2017/02/22/microservices-security-with-oauth2/)

- Advanced Microservices Security with OAuth2 (https://piotrminkowski.wordpress.com/2017/03/30/advanced-microservices-security-with-oauth2/)

Today I'm going to show you more advanced sample than before, where all authentication and OAuth2 data is stored on database. We also find out how to secure microservices, especially considering an inter-communication between them with Feign client. I hope this article will provide a guidance and help you with designing and implementing secure solutions with Spring Cloud. Let's begin.

There are four services running inside our sample system, what is visualized on the figure below. There is nothing unusual here. We have a discovery server where our sample microservices account-service and customer-service are registered. Those microservices are both protected with OAuth2 authorization. Authorization is managed by auth-server. It stores not only OAuth2 tokens, but also users authentication data. The whole process is implemented using Spring Security and Spring Cloud libraries.



1. Start database

All the authentication credentials and tokens are stored in MySQL database. So, the first step is to start MySQL. The most comfortable way to achieve it is through a Docker container. The command visible below in addition to starting database also creates schema and user oauth2.

```
docker run -d --name mysql -e MYSQL_DATABASE=oauth2 -e MYSQL_USER=oaut
```

2. Configure data source in application

MySQL is now available on port host *192.168.99.100* if you run Docker on Windows and port *33306*. Datasource properties should be set in `application.yml` of auth-server. Spring Boot is also able to run some SQL scripts on selected datasource after an application startup. It's good news for us, because we have to create some tables on the schema dedicated for OAuth2 process.

```
spring:
  application:
    name: auth-server
  datasource:
    url: jdbc:mysql://192.168.99.100:33306/oauth2?useSSL=false
    username: oauth2
    password: oauth2
    driver-class-name: com.mysql.jdbc.Driver
    schema: classpath:/script/schema.sql
    data: classpath:/script/data.sql
```

3. Create schema in MySQL

Despite appearances, it is not so simple to find the SQL script with tables that needs to be created when using Spring Security for OAuth2. Here's that script, which is available under `/src/main/resources/script/schema.sql` in auth-server module. We have to create six tables:

- oauth_client_details
- oauth_client_token
- oauth_access_token

- oauth_refresh_token
- oauth_code
- oauth_approvals

```
drop table if exists oauth_client_details;
create table oauth_client_details (
  client_id VARCHAR(255) PRIMARY KEY,
  resource_ids VARCHAR(255),
  client_secret VARCHAR(255),
  scope VARCHAR(255),
  authorized_grant_types VARCHAR(255),
  web_server_redirect_uri VARCHAR(255),
  authorities VARCHAR(255),
  access_token_validity INTEGER,
  refresh_token_validity INTEGER,
  additional_information VARCHAR(4096),
  autoapprove VARCHAR(255)
);
drop table if exists oauth_client_token;
create table oauth_client_token (
  token_id VARCHAR(255),
  token LONG VARBINARY,
  authentication_id VARCHAR(255) PRIMARY KEY,
  user_name VARCHAR(255),
  client_id VARCHAR(255)
);

drop table if exists oauth_access_token;
CREATE TABLE oauth_access_token (
  token_id VARCHAR(256) DEFAULT NULL,
  token BLOB,
  authentication_id VARCHAR(256) DEFAULT NULL,
  user_name VARCHAR(256) DEFAULT NULL,
  client_id VARCHAR(256) DEFAULT NULL,
  authentication BLOB,
  refresh_token VARCHAR(256) DEFAULT NULL
);

drop table if exists oauth_refresh_token;
CREATE TABLE oauth_refresh_token (
  token_id VARCHAR(256) DEFAULT NULL,
  token BLOB,
  authentication BLOB
);

drop table if exists oauth_code;
create table oauth_code (
  code VARCHAR(255), authentication LONG VARBINARY
);
drop table if exists oauth_approvals;
create table oauth_approvals (
    userId VARCHAR(255),
    clientId VARCHAR(255),
    scope VARCHAR(255),
    status VARCHAR(10),
```

```
    expiresAt DATETIME,
    lastModifiedAt DATETIME
);
```

4. Add some test data to database

There is also the second SQL script `/src/main/resources/script/data.sql` with some insert commands for the test purpose. The most important thing is to add some client id/client secret pairs.

```
INSERT INTO `oauth_client_details` (`client_id`, `client_secret`, `sco
INSERT INTO `oauth_client_details` (`client_id`, `client_secret`, `sco
INSERT INTO `oauth_client_details` (`client_id`, `client_secret`, `sco
```

5. Bulding Authorization Server

Now, the most important thing in this article – authorization server configuration. The configuration class should be annotated with `@EnableAuthorizationServer`. Then we need to overwrite some methods from extended `AuthorizationServerConfigurerAdapter` class. The first important thing here is to set the default token storage to a database by providing bean `JdbcTokenStore` with default data source as a parameter. Although all tokens are now stored in a database we still want to generate them in JWT format. That's why the second bean `JwtAccessTokenConverter` has to be provided in that class. By overriding different configure methods inherited from the base class we can set a default storage for OAuth2 client details and require authorization server to always verify the API key submitted in HTTP headers.

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Config extends AuthorizationServerConfigurerAdapte

    @Autowired
    private DataSource dataSource;
    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpo
```

```
12          endpoints.authenticationManager(this.authenticationManager).tok
13                .accessTokenConverter(accessTokenConverter());
14      }
15
16      @Override
17      public void configure(AuthorizationServerSecurityConfigurer oauthS
18          oauthServer.checkTokenAccess("permitAll()");
19      }
20
21      @Bean
22      public JwtAccessTokenConverter accessTokenConverter() {
23          return new JwtAccessTokenConverter();
24      }
25
26      @Override
27      public void configure(ClientDetailsServiceConfigurer clients) thro
28          clients.jdbc(dataSource);
29      }
30
31      @Bean
32      public JdbcTokenStore tokenStore() {
33          return new JdbcTokenStore(dataSource);
34      }
35
36  }
```

The main OAuth2 grant type, which is used in the current sample is Resource owner credentials grant. In that type of grant client application sends user login and password to authenticate against OAuth2 server. A POST request sent by the client contains the following parameters:

- grant_type – with the value 'password'
- client_id – with the client's ID
- client_secret – with the client's secret
- scope – with a space-delimited list of requested scope permissions
- username – with the user's username
- password – with the user's password

The authorization server will respond with a JSON object containing the following parameters:

- token_type – with the value 'Bearer'
- expires_in – with an integer representing the TTL of the access token
- access_token – the access token itself

- refresh_token – a refresh token that can be used to acquire a new access token when the original expires

Spring application provides a custom authentication mechanism by implementing `UserDetailsService` interface and overriding its method `loadUserByUsername`. In our sample application user credentials and authorities are also stored in the database, so we inject `UserRepository` bean to the custom `UserDatailsService` class.

```
@Component("userDetailsService")
public class UserDetailsServiceImpl implements UserDetailsService {

    private final Logger log = LoggerFactory.getLogger(UserDetailsSer

    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(final String login) {

        log.debug("Authenticating {}", login);
        String lowercaseLogin = login.toLowerCase();

        User userFromDatabase;
        if(lowercaseLogin.contains("@")) {
            userFromDatabase = userRepository.findByEmail(lowercaseLo
        } else {
            userFromDatabase = userRepository.findByUsernameCaseInsen
        }

        if (userFromDatabase == null) {
            throw new UsernameNotFoundException("User " + lowercaseLo
        } else if (!userFromDatabase.isActivated()) {
            throw new UserNotActivatedException("User " + lowercaseLo
        }

        Collection<GrantedAuthority> grantedAuthorities = new ArrayLi
        for (Authority authority : userFromDatabase.getAuthorities())
            GrantedAuthority grantedAuthority = new SimpleGrantedAuth
            grantedAuthorities.add(grantedAuthority);
        }

        return new org.springframework.security.core.userdetails.User
    }

}
```

That's practically all what should be written about auth-service module. Let's move on to the client microservices.

6. Bulding microservices

The REST API is very simple. It does nothing more than returning some data. However, there is one interesting thing in that implementation. That is preauthorization based on OAuth token scope, which is annotated on the API methods with `@PreAuthorize("#oauth2.hasScope('read')")`.

```
@RestController
public class AccountController {

   @GetMapping("/{id}")
   @PreAuthorize("#oauth2.hasScope('read')")
   public Account findAccount(@PathVariable("id") Integer id) {
      return new Account(id, 1, "123456789", 1234);
   }

   @GetMapping("/")
   @PreAuthorize("#oauth2.hasScope('read')")
   public List<Account> findAccounts() {
      return Arrays.asList(new Account(1, 1, "123456789", 1234), new
        new Account(3, 1, "123456781", 10000));
   }

}
```

Preauthorization is disabled by default. To enable it for API methods we should use `@EnableGlobalMethodSecurity` annotation. We should also declare that such a preauthorization would be based on OAuth2 token scope.

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class OAuth2ResourceServerConfig extends GlobalMethodSecurityC

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler
        return new OAuth2MethodSecurityExpressionHandler();
    }

}
```

7. Feign client with OAuth2

The API method findAccounts implemented in AccountController is invoked by customer-service through a Feign client.

```
@FeignClient(name = "account-service", configuration = AccountClientCo
public interface AccountClient {

   @GetMapping("/")
   List<Account> findAccounts();

}
```

If you call account service endpoint via Feign client you get the following exception.

```
feign.FeignException: status 401 reading AccountClient#findAccounts();
```

Why? Of course, account-service is protected with OAuth2 token authorization, but Feign client does not send an authorization token in the request header. That aproach may be customized by defining custom configuration class for Feign client. It allows to declare a request interceptor. In that case we can use an implementation for OAuth2 provided by `OAuth2FeignRequestInterceptor` from Spring Cloud OAuth2 library. We prefer password

```
public class AccountClientConfiguration {

   @Value("${security.oauth2.client.access-token-uri}")
   private String accessTokenUri;
   @Value("${security.oauth2.client.client-id}")
   private String clientId;
   @Value("${security.oauth2.client.client-secret}")
   private String clientSecret;
   @Value("${security.oauth2.client.scope}")
   private String scope;

   @Bean
   RequestInterceptor oauth2FeignRequestInterceptor() {
      return new OAuth2FeignRequestInterceptor(new DefaultOAuth2Clien
   }

   @Bean
   Logger.Level feignLoggerLevel() {
      return Logger.Level.FULL;
   }

   private OAuth2ProtectedResourceDetails resource() {
      ResourceOwnerPasswordResourceDetails resourceDetails = new Reso
      resourceDetails.setUsername("piomin");
      resourceDetails.setPassword("piot123");
```

```
        resourceDetails.setAccessTokenUri(accessTokenUri);
        resourceDetails.setClientId(clientId);
        resourceDetails.setClientSecret(clientSecret);
        resourceDetails.setGrantType("password");
        resourceDetails.setScope(Arrays.asList(scope));
        return resourceDetails;
    }

}
```

8. Testing

Finally, we may perform some tests. Let's build a sample project using `mvn clean install` command. If you run all the services with the default settings they would be available under addresses:

- **Config Server** – *http://localhost:9999/*
- **Discovery Server** – *http://localhost:8761/*
- **Account Service** – *http://localhost:8082/*
- **Customer Service** – *http://localhost:8083/*

The test method is visible below. We use `OAuth2RestTemplate` with `ResourceOwnerPasswordResourceDetails` to perform resource owner credentials grant operation and call `GET /{id}` API method from customer-service with OAuth2 token send in the request header.

```
@Test
public void testClient() {
    ResourceOwnerPasswordResourceDetails resourceDetails = new Resour
    resourceDetails.setUsername("piomin");
    resourceDetails.setPassword("piot123");
    resourceDetails.setAccessTokenUri("http://localhost:9999/oauth/to
    resourceDetails.setClientId("customer-service");
    resourceDetails.setClientSecret("secret");
    resourceDetails.setGrantType("password");
    resourceDetails.setScope(Arrays.asList("read"));
    DefaultOAuth2ClientContext clientContext = new DefaultOAuth2Clien
    OAuth2RestTemplate restTemplate = new OAuth2RestTemplate(resource
    restTemplate.setMessageConverters(Arrays.asList(new MappingJackso
    final Customer customer = restTemplate.getForObject("http://local
    System.out.println(customer);
}
```

Related

[Microservices security with Oauth2]

In "security"

[Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud]

In "microservices"

[Part 2: Creating microservices - monitoring with Spring Cloud Sleuth, ELK and Zipkin]

In "spring cloud"

Author: Piotr Mińkowski

IT Architect, Java Software Developer View all posts by Piotr Mińkowski

Piotr Mińkowski / December 1, 2017 / security / authorization, Feign, microservices, MySQL, Oauth2, security, spring cloud, spring-security

4 thoughts on “Part 2: Microservices security with OAuth2”

Vivek

July 18, 2018 at 7:27 am

hi, I have tried to implement in my project but unable to established as you mentioned. Please provide me the github link of above implementation. Is

★ Like

Vivek

August 16, 2018 at 9:02 am

Hi , I have implemented the password grant type as you define in blog. But My requirement is to implement the implicit resource . can you please help to achieve this. As I want implicit secutiry for intercommunication service.

★ Like

Ibrahima Kane

September 19, 2018 at 8:40 am

Thanks a lot Piotr for this awesome article.

I do like you schemas also. What drawing tool did you use?

★ Like

Piotr Mińkowski

September 19, 2018 at 10:00 am

Thanks 🙂 Google Drawings 🙂

★ Like