Piotr's TechBlog

Java, Spring, Kotlin, microservices, Kubernetes, containers

Spring Cloud Gateway OAuth2 with Keycloak

Security, Spring Cloud

Spring Cloud Gateway OAuth2 with Keycloak

By piotr.minkowski • October 9, 2020 • 22

Spring Cloud Gateway OAuth2 support is a key part of the microservices security process. Of course, the main reason for using an API gateway pattern is to hide services from the external client. However, when we set about hiding our services, we didn't secure them. In this article, I'm going to show you how to set up Spring Cloud Gateway OAuth2 with Spring Security and Keycloak.

> Spring Cloud Gateway is a very useful product. You may take an advantage of the many interesting features it provides. One of them is rate-limiting. You may read more about that in the article **Rate Limiting in Spring Cloud Gateway with Redis**. It is also worth learning about a circuit breaking and fault tolerance. You may find interesting pieces of information about it in the articles **Circuit Breaking in Spring Cloud Gateway with Resilience4j** and **Timeouts and Retries in Spring Cloud Gateway**.

Search

Search Go

Follow Blog via Email

Enter your email address to follow this blog and receive notifications of new posts by email.

Enter your email address

Follow

Categories

- Cloud (21)
- Containers (54)
- Continuous Integration (18)
- Data Grids (7)
- Kotlin (9)
- Kubernetes (43)
- Message Brokers (15)

Source code

If you would like to try it by yourself, you may always take a look at my source code. In order to do that you need to clone my repository **sample-spring-security-microservices**. Then you should go to the `gateway` directory, and just follow my instructions 🙂 If you are interested in more details about Spring Security you should read its **documentation**.

Enable OAuth2 in Spring Cloud Gateway

To enable OAuth2 support for the Spring Cloud Gateway application we need to add some dependencies. Of course, the `spring-cloud-starter-gateway` dependency is required to enable the gateway feature. We also need to include `spring-boot-starter-oauth2-client` enabling Spring Security's client support for OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0. Finally, we have to add `spring-cloud-starter-security` to activate the `TokenRelay` filter.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

In the next step, we need to provide the configuration settings for the OAuth2 client. Because we are integrating with Keycloak we should set the name of registrationId (`spring.security.oauth2.client.provider.[registrationId]`) to `keycloak`. Then we need to set the uris of `token`, `authorization` and `userinfo` endpoints. On the other hand, we can set a value for a single `issuer` endpoint. The last important property in that section is `user-name-attribute`. Keycloak is returning user login inside the `preferred_username` attribute.

We will define two different clients for authorization. The first of them `spring-cloud-gateway` contains the scope allowed by our test method, while the second `spring-cloud-gateway-2` does not.

- Micronaut (12)
- Microservices (54)
- Other (16)
- Performance (16)
- Quarkus (3)
- Security (15)
- Spring Boot (57)
- Spring Cloud (40)

Tags

Apache Kafka Consul Containers Continuous Delivery Continuous Integration Docker ELK Eureka Hibernate istio Java Jenkins Jib JPA Kotlin Kubernetes Maven Micronaut microservices Minikube OAuth2 openshift Performance quarkus Reactor REST Ribbon security service discovery service mesh Skaffold Spring Spring Boot Spring Boot Actuator Spring Boot Microservices Spring Cloud Spring Cloud Config Spring Cloud Gateway Spring Cloud Kubernetes Spring Data Spring Security Spring WebFlux Swagger testcontainers testing

```
spring:
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri:
http://localhost:8080/auth/realms/master/protocol/openid-
connect/token
            authorization-uri:
http://localhost:8080/auth/realms/master/protocol/openid-connect/auth
            userinfo-uri:
http://localhost:8080/auth/realms/master/protocol/openid-
connect/userinfo
            user-name-attribute: preferred_username
        registration:
          keycloak-with-test-scope:
            provider: keycloak
            client-id: spring-with-test-scope
            client-secret: c6480137-1526-4c3e-aed3-295aabcb7609
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
          keycloak-without-test-scope:
            provider: keycloak
            client-id: spring-without-test-scope
            client-secret: f6fc369d-49ce-4132-8282-5b5d413eba23
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
```

In the last step, we need to configure Spring Security. Since Spring Cloud Gateway is built on top of Spring WebFlux, we need to annotate the configuration bean with `@EnableWebFluxSecurity`. Inside the `springSecurityFilterChain` method we are going to enable authorization for all the exchanges. We will also set OAuth2 as a default login method and finally disable CSRF.

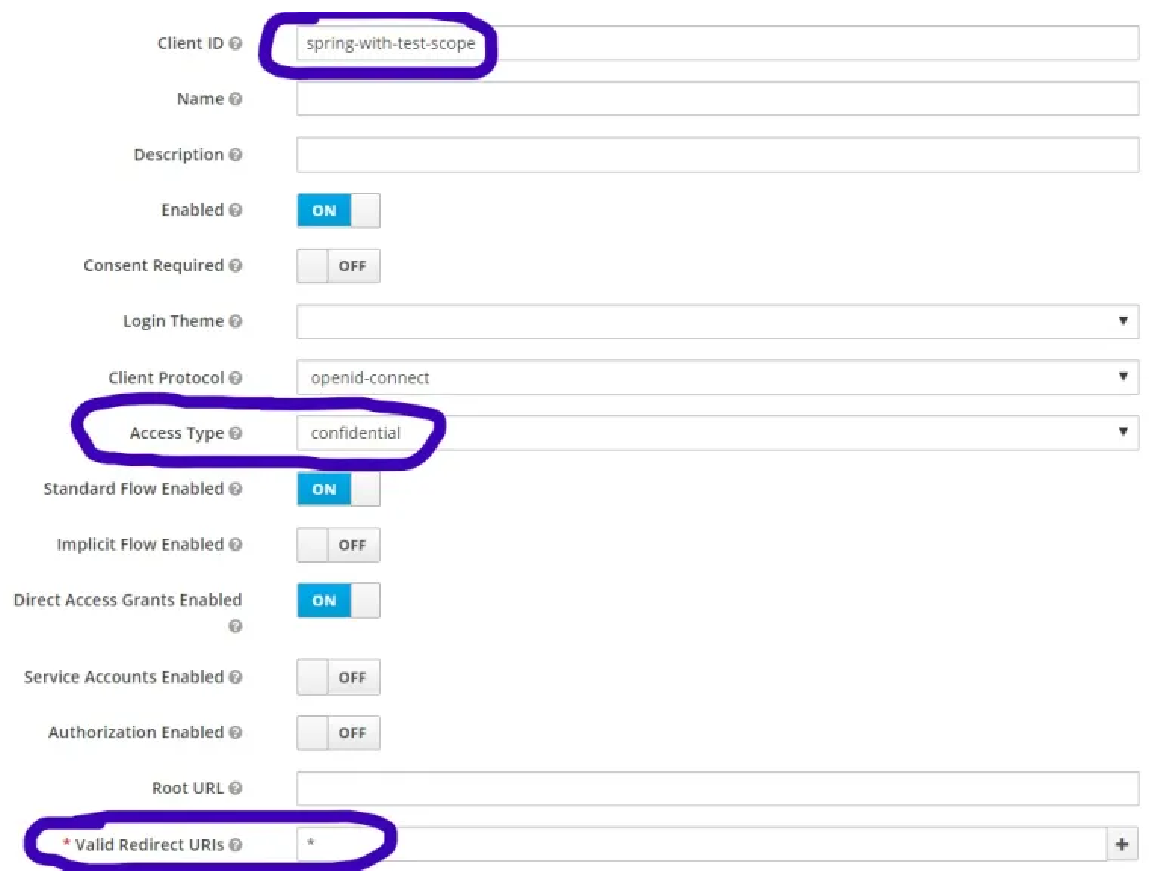
```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

   @Bean
   public SecurityWebFilterChain
springSecurityFilterChain(ServerHttpSecurity http) {
      http.authorizeExchange(exchanges ->
exchanges.anyExchange().authenticated())
         .oauth2Login(withDefaults());
      http.csrf().disable();
      return http.build();
   }

}
```

Run and configure Keycloak

We are running Keycloak on a Docker container. By default, Keycloak exposes API and a web console on port `8080`. However, that port number must be different than the Spring Cloud Gateway application port, so we are overriding it with `8888`. We also need to set a username and password to the admin console.

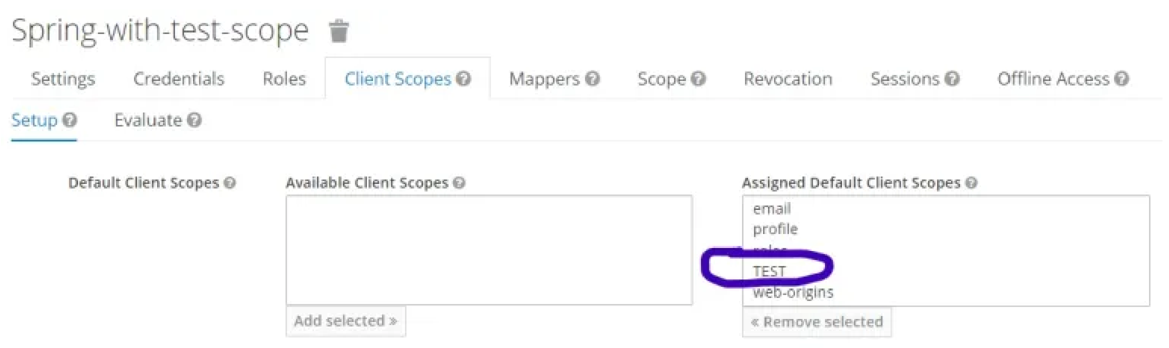
```
$ docker run -d --name keycloak -p 8888:8080 \
   -e KEYCLOAK_USER=spring \
   -e KEYCLOAK_PASSWORD=spring123 \
   jboss/keycloak
```

Then we need to create two clients with the same names as defined inside the gateway configuration. Both of them need to have `confidential` in the "Access Type" section, a valid redirection URI set. We may use a simple wildcard while setting the redirection address as shown below.



Client ID	spring-with-test-scope
Name	
Description	
Enabled	ON
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	confidential
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON
Service Accounts Enabled	OFF
Authorization Enabled	OFF
Root URL	
* Valid Redirect URIs	*

The client `spring-with-test-scope` will have the scope `TEST` assigned. In contrast, the second client `spring-without-test-scope` will not have the scope `TEST` assigned.



Spring-with-test-scope

Default Client Scopes	Available Client Scopes	Assigned Default Client Scopes
		email
		profile
		roles
		TEST
		web-origins
	Add selected »	« Remove selected

Enable OAuth2 Resourse in Spring Cloud Gateway

Now, we may proceed to the implementation of the downstream application. In order to run it, you need to switch to the `callme` directory in the source code. First, we need to include some Maven dependencies. The `spring-boot-starter-web` starter provides web support for Spring Boot application. With `spring-boot-starter-security` we enable Spring Security for our microservice. The `spring-security-oauth2-resource-server` contains Spring Security's support for OAuth 2.0 Resource Servers. It is also used to protect APIs via OAuth 2.0 Bearer Tokens. Finally, the `spring-security-oauth2-jose` module contains Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework provides a method to securely transfer claims between parties. It supports JWT and JWS (JSON Web Signature).

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

In the next step, we need to configure a connection to the authorization server. A resource server will use the property `spring.security.oauth2.resourceserver.jwt.issuer-uri` to discover the authorization server public keys and then validate incoming JWT tokens.

```
spring:
  application:
    name: callme
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/auth/realms/master
```

We should also provide a Spring Security configuration. First, we need to annotate the `Configuration` bean with `@EnableWebSecurity`. Then, we should enable annotation-based security for the controller methods. It allows simple role-based access with `@PreAuthorize` and `@PostAuthorize`. In order to enable a method security feature we need to use annotation `@EnableGlobalMethodSecurity`. Finally, we just need to configure Spring Security to authorize all the incoming requests and validate JWT tokens.

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

   protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests(authorize ->
authorize.anyRequest().authenticated())

.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
   }
}
```

Finally, let's take a look at the implementation of the REST controller class. It a single `ping` method. That method may be accessed only by the client with the `TEST` scope. It returns a list of assigned scopes from the `Authentication` bean.

```
@RestController
@RequestMapping("/callme")
public class CallmeController {

   @PreAuthorize("hasAuthority('SCOPE_TEST')")
   @GetMapping("/ping")
   public String ping() {
      SecurityContext context = SecurityContextHolder.getContext();
      Authentication authentication = context.getAuthentication();
      return "Scopes: " + authentication.getAuthorities();
   }
}
```

Configure routing on Spring Cloud Gateway

The last step before proceeding to the tests is to configure routing on the Spring Cloud Gateway application. Since the downstream service (`callme`) is running on port `8040` we need to set the `uri` to `http://127.0.0.1:8040`. In order to forward the access token to the `callme-service` we have to enable a global filter `TokenRelay`. Just to be sure that everything works as expected, we will remove the `Cookie` with the session id. The session id is generated on the gateway after performing `OAuth2Login`.

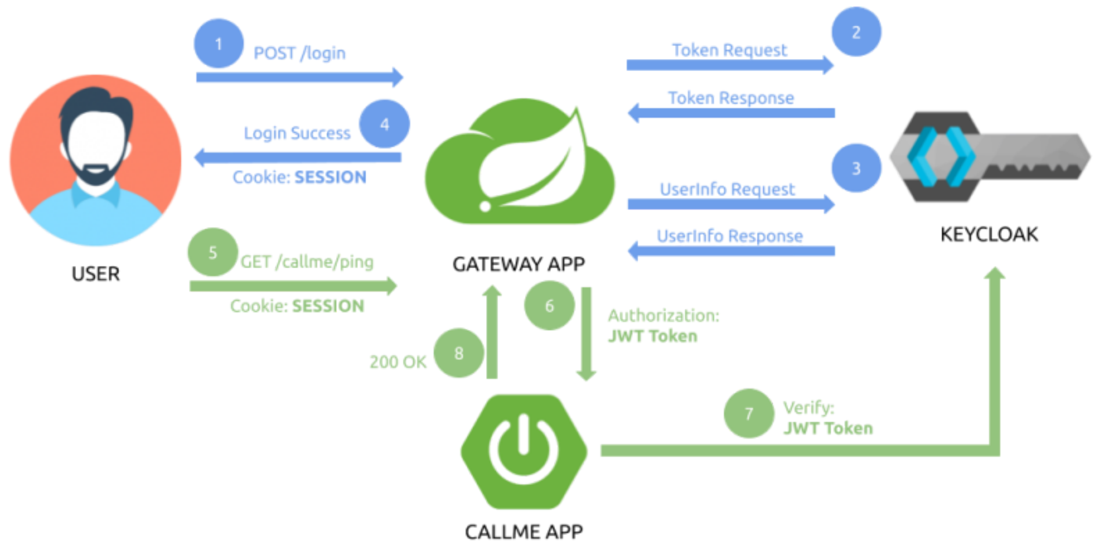
```
spring:
  application:
    name: gateway
  cloud:
    gateway:
      default-filters:
        - TokenRelay
      routes:
        - id: callme-service
          uri: http://127.0.0.1:8040
          predicates:
            - Path=/callme/**
          filters:
            - RemoveRequestHeader=Cookie
```

Finally, let's take a look at the gateway main class. I added there two useful endpoints. First of them `GET /` is returning the HTTP session id. The second of them `GET /token` will return the current JWT access token. After the successful Spring Cloud Gateway OAuth2 login, you will see the result from the `index` method.

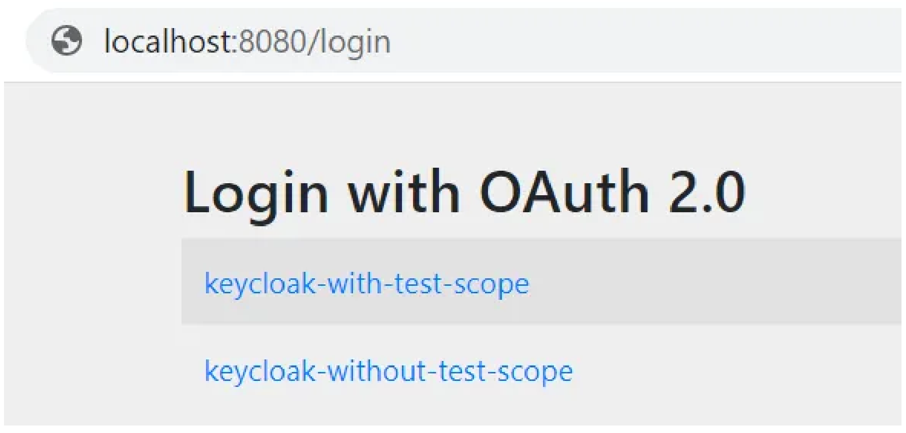
```
@SpringBootApplication
@RestController
public class GatewayApplication {

   private static final Logger LOGGER =
LoggerFactory.getLogger(GatewayApplication.class);

   public static void main(String[] args) {
      SpringApplication.run(GatewayApplication.class, args);
   }

   @GetMapping(value = "/token")
   public Mono<String> getHome(@RegisteredOAuth2AuthorizedClient
OAuth2AuthorizedClient authorizedClient) {
      return
Mono.just(authorizedClient.getAccessToken().getTokenValue());
   }

   @GetMapping("/")
   public Mono<String> index(WebSession session) {
      return Mono.just(session.getId());
   }

}
```

Spring Cloud Gateway OAuth2 testing scenario

First, let's take a look at the picture that illustrates our use case. We are calling `POST /login` endpoint on the gateway (1). After receiving the login request Spring Cloud Gateway try to obtain the access token from the authorization server (2). Then Keycloak is returning the JWT access token. As a result, Spring Cloud Gateway is calling the `userinfo` endpoint (3). After receiving the response it is creating a web session and `Authentication` bean. Finally, the gateway application is returning a session id to the external client (4). The external client is using a cookie with session-id to authorize requests. It calls `GET ping` from the `callme` application (5). The `gateway` application is forwarding the request to the downstream service (6). However, it removes the cookie and replaces it with a JWT access token. The `callme` application verifies an incoming token (7). Finally, it returns `200 OK` response if the client is allowed to call endpoint (8). Otherwise, it returns `403 Forbidded`.



We may start testing in the web browser. First, let's call the `login` endpoint. We have to available clients `keycloak-with-test-scope` and `keycloak-without-test-scope`. We will use the client `keycloak-with-test-scope`.

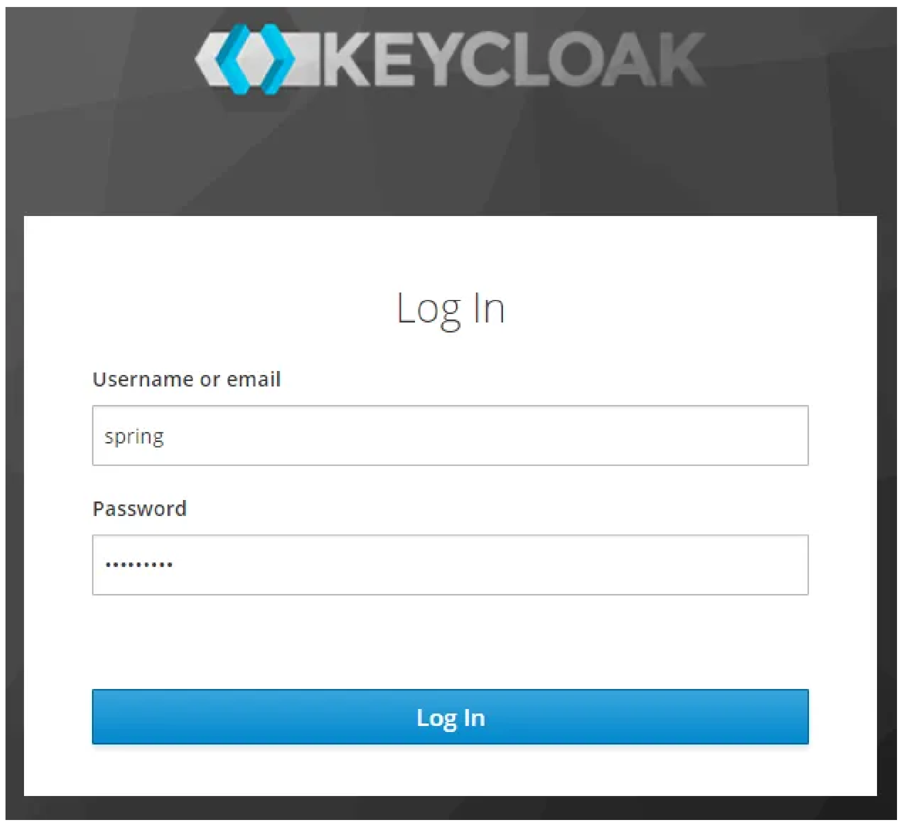

localhost:8080/login

Login with OAuth 2.0

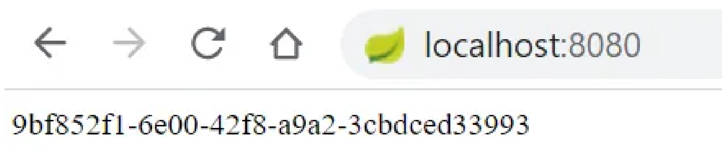
keycloak-with-test-scope

keycloak-without-test-scope

Then, the gateway redirects us to the Keycloak `login` page. We can use the credentials provided during the creation of the Keycloak container.



After a successful login, the gateway will perform the OAuth2 authorization procedure. Finally, it redirects us to the main page. The main page is just a method `index` inside the controller. It is returning the current session id.

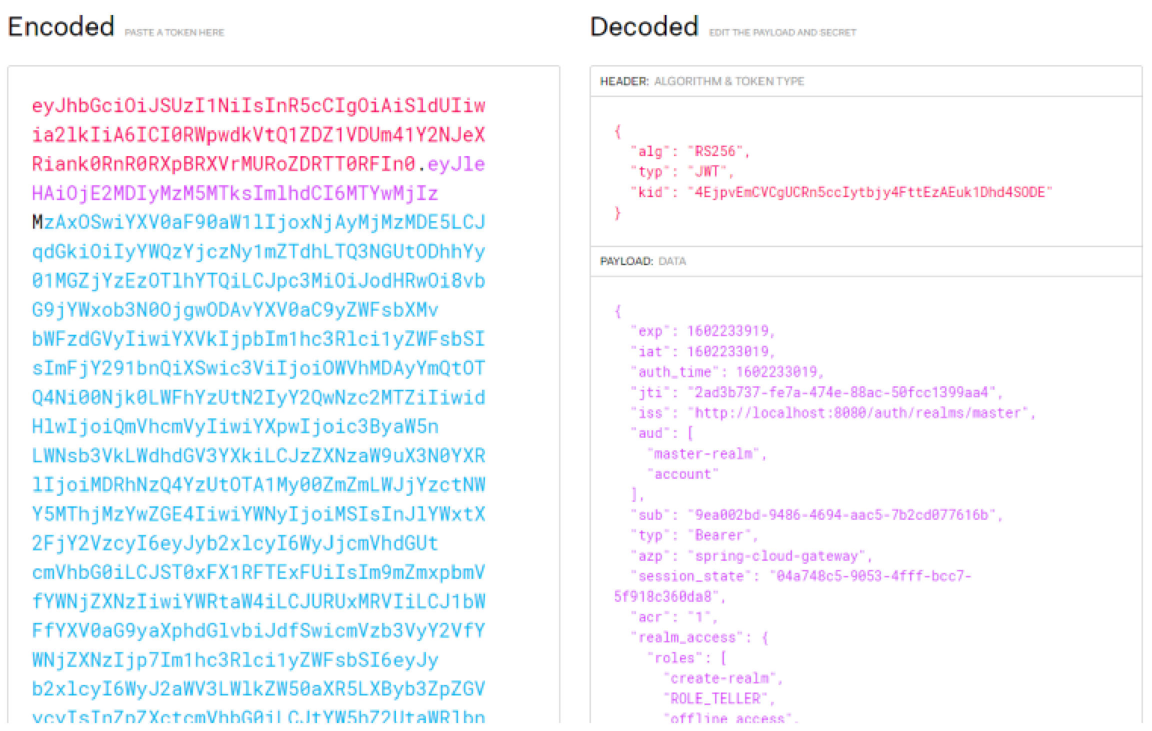

localhost:8080

```
9bf852f1-6e00-42f8-a9a2-3cbdced33993
```

We can also use another endpoint implemented on the gateway – `GET /token`. It is returning the current JWT access token.

```
$ curl http://localhost:8080/token -H "Cookie: SESSION=9bf852f1-6e00-
42f8-a9a2-3cbdced33993"
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI0RWpwdkVtQ1ZDZ1VDUm41Y

MzAxOSwiYXV0aF90aW1lIjoxNjAyMjMzMDE5LCJqdGkiOiIyYWQzYjczNy1mZTdhLTQ3NGUt

bWFzdGVyIiwiYXVkIjpbIm1hc3Rlci1yZWFsbSIsImFjY291bnQiXSwic3ViIjoiOWVhMDAy

LWNsb3VkLWdhdGV3YXkiLCJzZXNzaW9uX3N0YXRlIjoiMDRhNzQ4YzUtOTA1My00ZmZmLWJj

cmVhbG0iLCJST0xFX1RFTExFUiIsIm9mZmxpbmVfYWNjZXNzIiwiYWRtaW4iLCJURUxMRVIi

b2xlcyI6WyJ2aWV3LWlkZW50aXR5LXByb3ZpZGVycyIsInZpZXctcmVhbG0iLCJtYW5hZ2Ut

dXNlcnMiLCJxdWVyeS1yZWFsbXMiLCJ2aWV3LWF1dGhvcml6YXRpb24iLCJxdWVyeS1jbGll

dHMiLCJ2aWV3LXVzZXJzIiwidmlldy1jbGllbnRzIiwibWFuYWdlLWF1dGhvcml6YXRpb24i

Z2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwi

ZF91c2VybmFtZSI6InBpb21pbiJ9.X8XfIHiAiR1YMjiJza75aC294qLwi83RrUU2phorM7F
lx80Zu5xqTrMqMC1-RbHBnX-oUTbs4ViS3DziZlDvoRajdkrh6UTiK5oWgoRW-
4qsH5L4X1W
bRfoBZgyHFRSnhaCO4CLgjCyEgeLUR5A-JWY-
OMYQIOAxxHB2GwE3MNFfLWeqpmS1AWU8fL0giFFXFDfa1_XZEKgnqe1S75Ps_z8B1sfNfvNp
iPuFqDD
RT6DGU-Hing9LnGuOt3Yas-WYdN7PKBigvIZv0LyvRFcilRJQBjOdVfEddL3OQ0rmEg
```

Just to check, you can decode a JWT token on the **https://jwt.io** site.



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
ia2lkIiA6ICI0RWpwdkVtQ1ZDZ1VDUm41Y2NJeX
Riank0RnR0RXpBRXVrMURoZDRTT0RFIn0.eyJle
HAiOjE2MDIyMzM5MTksImlhdCI6MTYwMjIz
MzAxOSwiYXV0aF90aW1lIjoxNjAyMjMzMDE5LCJ
qdGkiOiIyYWQzYjczNy1mZTdhLTQ3NGUtODhhYy
01MGZjYzEzOTlhYTQiLCJpc3MiOiJodHRwOi8vb
G9jYWxob3N0OjgwODAvYXV0aC9yZWFsbXMv
bWFzdGVyIiwiYXVkIjpbIm1hc3Rlci1yZWFsbSI
sImFjY291bnQiXSwic3ViIjoiOWVhMDAyYmQtOT
Q4Ni00Njk0LWFhYzUtN2IyY2QwNzc2MTZiIiwid
HlwIjoiQmVhcmVyIiwiYXpwIjoic3ByaW5n
LWNsb3VkLWdhdGV3YXkiLCJzZXNzaW9uX3N0YXR
lIjoiMDRhNzQ4YzUtOTA1My00ZmZmLWJjYzctNW
Y5MThjMzYwZGE4IiwiYWNyIjoiMSIsInJlYWxtX
2FjY2VzcyI6eyJyb2xlcyI6WyJjcmVhdGUt
cmVhbG0iLCJST0xFX1RFTExFUiIsIm9mZmxpbmV
fYWNjZXNzIiwiYWRtaW4iLCJURUxMRVIiLCJ1bW
FfYXV0aG9yaXphdGlvbiJdfSwicmVzb3VyY2VfY
WNjZXNzIjp7Im1hc3Rlci1yZWFsbSI6eyJy
b2xlcyI6WyJ2aWV3LWlkZW50aXR5LXByb3ZpZGV
ycyIsInZpZXctcmVhbG0iLCJtYW5hZ2UtaWRlbn
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "4EjpvEmCVCgUCRn5ccIytbjy4FttEzAEuk1Dhd4SODE"
}
```

PAYLOAD: DATA

```
{
  "exp": 1602233919,
  "iat": 1602233019,
  "auth_time": 1602233019,
  "jti": "2ad3b737-fe7a-474e-88ac-50fcc1399aa4",
  "iss": "http://localhost:8080/auth/realms/master",
  "aud": [
    "master-realm",
    "account"
  ],
  "sub": "9ea002bd-9486-4694-aac5-7b2cd077616b",
  "typ": "Bearer",
  "azp": "spring-cloud-gateway",
  "session_state": "04a748c5-9053-4fff-bcc7-
5f918c360da8",
  "acr": "1",
  "realm_access": {
    "roles": [
      "create-realm",
      "ROLE_TELLER",
      "offline_access",
```

Finally, let's call the endpoint exposed by the `callme` application. We are setting session `Cookie` in the request header. The endpoint is returning a list of scopes assigned to the current user. Only user with scope `TEST` is allowed to call the method.

```
$ curl http://localhost:8080/callme/ping -H "Cookie:
SESSION=9bf852f1-6e00-42f8-a9a2-3cbdced33993"
Scopes: [SCOPE_profile, SCOPE_email, SCOPE_TEST]
```

Conclusion

In this article we were discussing important aspects related to microservices security. I showed you how to enable Spring Cloud Gateway OAuth2 support and integrate it with Keycloak. We were implementing such mechanisms like OAuth2 login, token relay, and OAuth2 resource server. The token relay mechanisms will be completely migrated from Spring Cloud Security to Spring Cloud Gateway. Enjoy 🙂

Share this:

Twitter Facebook

Like this:

Like

2 bloggers like this.

Related

Spring Microservices Security Best Practices
May 26, 2021
In "Microservices"

A New Era Of Spring Cloud
May 1, 2020
In "Kotlin"

Secure Rate Limiting with Spring Cloud Gateway
May 21, 2021
In "Microservices"

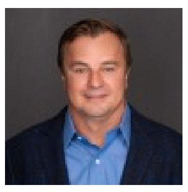
Keycloak microservices OAuth2 security Spring Cloud Spring Cloud Gateway Spring Security

Written by:

piotr.minkowski

View All Posts →

22 COMMENTS



andriykalashnykovgmailcom October 13, 2020 9:26 pm

Piotr, great article!

There are couple of thing you can automate – creation of Clients and Client Scops. You can either run Keycloak docker image and execute kcadm.sh commands against it:

# script – begin
docker run -d –rm –name keycloak -p 8888:8080 -e KEYCLOAK_USER=spring -e KEYCLOAK_PASSWORD=spring123 jboss/keycloak:11.0.2&

sleep 60
docker exec -it keycloak /opt/jboss/keycloak/bin/kcadm.sh config credentials –server http://localhost:8080/auth –realm master –user spring –password spring123 && \
docker exec -it keycloak /opt/jboss/keycloak/bin/kcadm.sh create -x "client-scopes" -r master -s name=TEST -s protocol=openid-connect && \
docker exec -it keycloak /opt/jboss/keycloak/bin/kcadm.sh create clients -r master -s clientId=spring-without-test-scope -s enabled=true -s clientAuthenticatorType=client-secret -s secret=79a93cb3-b460-40c8-8c96-c8c8bfe47d39 -s 'redirectUris=["*"]' && \
docker exec -it keycloak /opt/jboss/keycloak/bin/kcadm.sh create clients -r master -s clientId=spring-with-test-scope -s enabled=true -s clientAuthenticatorType=client-secret -s secret=b129f0c2-a46a-4bdb-a059-4eca03639767 -s 'redirectUris=["*"]' -s 'defaultClientScopes=["TEST", "web-origins", "profile", "roles", "email"]'
# script end

or you can create Client and Client Scopes with command line or from Keycloak web UI, export them into json and the run Keycloak image with imported realm. Though importing realm works fin for realm names other's then Master. Let's say you create realm Test and add your Clients and Client scope, but you need to modify application.yaml file where it references http://localhost:8888/auth/realms/master/ and use http://localhost:8888/auth/realms/test/ instead

# script – begin
# import test reaml
# docker run -d –rm –name keycloak -p 8888:8080 -e KEYCLOAK_USER=spring -e KEYCLOAK_PASSWORD=spring123 -e KEYCLOAK_MIGRATION_STRATEGY=OVERWRITE_EXISTING -e KEYCLOAK_IMPORT=/opt/jboss/keycloak/imports/realm-test.json -v $(pwd):/opt/jboss/keycloak/imports jboss/keycloak:11.0.2
# script – end

I've forked you repo and will provide scripts about soon (https://github.com/AndriyKalashnykov/sample-spring-security-microservices)

Loading...

REPLY

piotr.minkowski November 10, 2020 8:10 am

Ok. Thanks for the tips 🙂

Loading...

REPLY

Vishnu Prakash January 7, 2021 2:10 pm

how logout can be handled?

Loading...

REPLY

piotr.minkowski April 9, 2021 12:27 pm

You can configure it using `HttpSecurity` object

Loading...

REPLY

galih January 24, 2021 4:50 am

sir, i implemented this one and this works cool article but however i want to try to call the secured api from frontend, in this case i have reactjs spa separated from spring cloud gateway, how can i achieve this?

Loading...

REPLY

piotr.minkowski April 9, 2021 12:28 pm

Unfortunately, I'm not good at reactjs...

Loading...

REPLY

B. Taxi January 28, 2021 12:55 am

Thanks for the great article & videos! One (beginner) question ... why do the HTTP endpoint methods in the Gateway return Mono's, but the endpoint in the callme Controller returns a String? Thanks!

Loading...

REPLY

piotr.minkowski February 19, 2021 12:47 pm

Because the gateway is based on WebFlux and Reactor, while application on Spring MVC.

Loading...

REPLY

Gan March 1, 2021 10:39 am

Thank you for your article, although some parts are not well understood, I look forward to more keycloak tutorials

Loading...

REPLY

piotr.minkowski March 5, 2021 9:54 pm

Which of them is not well understood?

Loading...

REPLY

rokko11 March 19, 2021 8:24 am

Thank you, Piotr, for this great tutorial! Everything worked instantly! 🙂 (except keycloak port 8888 vs 8080 in application.yml)

Loading...

REPLY

Robert Vialonic March 24, 2021 11:32 am

Thank you for this great tutorial! Sadly something changed in Spring Boot version 2.4.x, but for the moment 2.3.x is ok for me 🙂

Another question: I changed the Authorization Annotation to

@PreAuthorize(value = "hasRole('TESTROLE')")

but it seems that the roles do not pass the gateway. Do you know how to solve this?

Loading...

REPLY

piotr.minkowski April 9, 2021 12:22 pm

Did add a new role in Keycloak?

Loading...

REPLY

Duy March 31, 2021 10:29 am

Thank you for your article.
Sorry my English is pretty bad, But I'm having trouble getting the HA keycloak on kubernetes, when I increase the number of GATEWAY's PODs the app can't log in, the same problem happened when I increased the number of PODs of other business SERIVCEs.

Loading...

REPLY

piotr.minkowski April 9, 2021 12:19 pm

Well, I didn't test with multiple pods. Do you any errors or warnings in the logs?

Loading...

REPLY

Ashish Kumar April 1, 2021 11:59 am

I am getting the below error:
Caused by: java.lang.IllegalArgumentException: Unable to find GatewayFilterFactory with name TokenRelay

Loading...

REPLY

piotr.minkowski April 9, 2021 12:18 pm

When do you get it? During startup?

Loading...

REPLY

springlearner June 3, 2021 3:50 am

How can we make the POST request bu sending OAuth2 token?

Loading...

REPLY

tigersoft November 2, 2021 5:40 am

Excellent article! Could you please extend it explaining how to consume this API via web application (i.e. Angular)? This may cause problems from configuration perspective to set up CORS settings and pass token obtained from public keycloak client to API gateway, to have all things work together. That would be great, thank you in advance.

Loading...

REPLY

piotr.minkowski November 24, 2021 8:18 am

Thanks. I'll try to do the in the near future. Thanks for the suggestion.

Loading...

REPLY

Hassen November 8, 2021 4:13 pm

Thank you for your interesting article.
i have an angular application running in a different domain when i try to call my spring gateway i have a cors problem knowing that if i remove keyclaok as well as spring-boot-starter-oauth2-client from pom.xml I can access my gateway without any cors problems.
thanks in advance for your help

Loading...

REPLY

piotr.minkowski November 24, 2021 8:17 am

You can change the CORS configuration with `CorsConfigurationSource` bean:

```
@Configuration @EnableWebFluxSecurity public
class SecurityConfig { @Bean public
SecurityWebFilterChain
springSecurityFilterChain(ServerHttpSecurity
http) { ... http.cors()...; return
http.build(); } @Bean CorsConfigurationSource
corsConfigurationSource() { CorsConfiguration
configuration = new CorsConfiguration();
configuration.setAllowedOrigins(Arrays.asList("
https://example.com"));
configuration.setAllowedMethods(Arrays.asList("
GET","POST")); UrlBasedCorsConfigurationSource
source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**",
configuration); return source; } }
```

Loading...

REPLY

Leave a Reply

Enter your comment here...

Contact info

If you would like to contact me in order you have any questions, thoughts or ideas (e.g. suggestions for future articles) contact me via email.

Top Posts & Pages

- Serverless Java Functions on OpenShift
- Kafka Streams with Spring Cloud Stream
- Guide to building Spring Boot library

Social Media

Warsaw, Poland
piotr.minkowski@gmail.com

- Kafka Streams with Quarkus
- An Advanced Guide to GraphQL with Spring Boot