Spring Cloud et keycloak

1 Préalable

Ce sujet est librement inspiré de cette documentation [1] (avec de nombreuses corrections).

2 Introduction

Spring Cloud Gateway est une bibliothèque qui nous permet de créer rapidement des passerelles API légères basées sur Spring Boot, que nous avons déjà abordé dans la séance précédente.

Cette fois-ci, nous allons montrer comment implémenter rapidement les modèles OAuth 2.0 au-dessus.

3 Récapitulatif rapide d'OAuth 2.0

OAuth 2.0 est une norme bien établie, utilisée sur tout Internet comme mécanisme de sécurité permettant aux utilisateurs et aux applications d'accéder aux ressources en toute sécurité.

Bien que la description détaillée de cette norme dépasse le cadre de ce sujet, commençons par un bref récapitulatif de quelques termes clés :

- **Ressource** : Toute information qui ne peut être récupérée que par des clients autorisés.
- **Client** : une application qui consomme une ressource, généralement via une API REST
- **Serveur de ressources** : Un service chargé de fournir une ressource aux clients autorisés
- **Propriétaire de la ressource** : entité (humaine ou application) qui possède une ressource et qui, en dernier ressort, est responsable d'en accorder l'accès à un client.
- **Jeton** : une information reçue par un client et envoyée à un serveur de ressources dans le cadre de la requête d'authentification.
- **Fournisseur d'identité (IdP)** : Valide les informations d'identification de l'utilisateur et délivre des jetons d'accès aux clients.
- **Flux d'authentification** : séquence d'étapes qu'un client doit suivre pour obtenir un jeton valide.

Pour une description complète de la norme, un bon point de départ est la documentation d'Auth0 sur ce sujet [2].

4 Spring Cloud Gateway en tant que client OAuth 2.0

Dans ce scénario, toute requête entrante non authentifiée déclenchera un flux de code d'autorisation . Une fois le jeton acquis par la passerelle, il sera utilisé lors de l'envoi de requêtes à un service `quotes` (gestion de devis).

- Un bon exemple de ce modèle en action est une application d'agrégation de flux de réseaux sociaux : pour chaque réseau pris en charge, la passerelle agirait comme un client OAuth 2.0.
- Par conséquent, l'interface utilisateur (généralement une application monopage développée avec Angular, React ou un framework similaire) peut accéder de manière transparente aux données de ces réseaux pour le compte de l'utilisateur final. Plus important encore : **elle peut le faire sans que l'utilisateur n'ait jamais à révéler ses identifiants à l'agrégateur**.

1. https://www.baeldung.com/spring-cloud-gateway-oauth2
2. https://auth0.com/docs/get-started/authentication-and-authorization-flow

5 Spring Cloud Gateway en tant que serveur de ressources OAuth 2.0

- **Ici, la passerelle joue le rôle de contrôleur, s'assurant que chaque requête possède un jeton d'accès valide avant de l'envoyer à un service** `quotes`. De plus, elle peut également vérifier si le jeton dispose des autorisations nécessaires pour accéder à une ressource donnée, en fonction des étendues associées.

- Il est important de noter que ce type de contrôle d'autorisation opère principalement à un niveau grossier. Le contrôle d'accès plus précis (par exemple, les autorisations au niveau des objets/champs) est généralement implémenté côté serveur à l'aide de la logique métier.
- Dans ce modèle, il convient de s'intéresser à la manière dont les services côté serveur authentifient et autorisent les requêtes transmises. Deux cas principaux se présentent :
  - ▷ **Propagation du jeton** : API Gateway transmet le jeton reçu au serveur dorsal tel quel.
  - ▷ **Remplacement de jeton** : API Gateway remplace le jeton entrant par un autre avant d'envoyer la requête.
- **Dans ce tutoriel, nous aborderons uniquement le cas de la propagation des jetons, car il s'agit du scénario le plus courant**. Le second cas est également possible, mais il nécessite une configuration et un codage supplémentaires qui nous détourneraient des points principaux que nous souhaitons présenter ici.

6 Aperçu d'un exemple de projet

- Pour illustrer l'utilisation de Spring Gateway avec les modèles OAuth décrits jusqu'ici, créons un exemple de projet exposant un seul point de terminaison : `/quotes/{symbol}` . L'accès à ce point de terminaison nécessite un jeton d'accès valide émis par le fournisseur d'identité configuré. Nous avons également prévu un end-point `/quotes` accessibles uniquement aux utilisateurs `gold`.
- Dans notre cas, nous utiliserons le fournisseur d'identité Keycloak intégré. Les seules modifications requises sont l'ajout d'une nouvelle application cliente et de quelques utilisateurs pour les tests.
- Pour pimenter un peu les choses, notre service `quotes` affichera un prix différent selon l'utilisateur associé à la demande. Les utilisateurs disposant du rôle `gold` bénéficieront d'un prix réduit, tandis que tous les autres paieront le prix normal.
- Nous allons utiliser Spring Cloud Gateway comme interface pour ce service et, en modifiant seulement quelques lignes de configuration, nous pourrons faire passer son rôle de client OAuth à celui de serveur de ressources.

7 Fournisseur d'identité Keycloak

- L'application Keycloak embarquée que nous utiliserons dans ce tutoriel est simplement une application SpringBoot classique que nous pouvons cloner depuis GitHub et compiler avec Maven :

```
## Coner tous les projets liés à la sécurité
git clone https://github.com/Baeldung/spring-security-oauth
## Récupérer le projet qui nous intéresse
mv -v spring-security-oauth/oauth-rest/oauth-authorization-server .
## Supprimer les autres
rm -vrf spring-security-oauth/
## Entrer dans le projet
cd oauth-authorization-server
mvn install
```

> **Note :** Ce projet cible actuellement Java 13 et versions ultérieures, mais il fonctionne également correctement avec Java 11. Il suffit dajouter `-Djava.version=11` à la commande Maven.

- Ensuite, nous remplacerons le fichier `src/main/resources/baeldung-domain.json` par cette version [3]. La version modifiée contient les mêmes configurations que l'originale, auxquelles s'ajoutent une application cliente supplémentaire (`quotes-client`), deux groupes d'utilisateurs (`golden_customers` et `silver_customers`) et deux rôles (`gold` et `silver`).
- Créer le répertoire pour les données de Keycloak :

```
mkdir -p /home/$USER/keycloakdb
```

- Changez dans les fichiers `src/main/resources/application.yml` et `src/main/resources/META-INF/keycloak-server.json` , l'URL JDBC par la valeur `jdbc:h2:/home/<votre-login>/keycloakdb` . Cette modification permet de pérenisser les modifications effectuées dans Keycloak.
- Nous pouvons maintenant démarrer le serveur à l'aide du plugin Maven `spring-boot:run` :

```
mvn spring-boot:run
```

- Une fois le serveur lancé, vous pouvez y accéder à l'adresse `http://localhost:8083/auth/admin/master/console/#/realms/baeldung` dans notre navigateur. Après vous être connectés avec les identifiants d'administrateur (`bael-admin` / `pass`), nous accéderons à l'écran de gestion du domaine.

- Pour finaliser la configuration du fournisseur d'identité, ajoutons deux utilisateurs. Le premier sera `maxwell.smart` , membre du groupe `golden_customer` . Le second sera `john.snow` , que nous n'ajouterons à aucun groupe. N'oubliez pas de donner un mot de passe à ces utilisateurs (*crédentials*) `snow` pour `john.snow` et `smart` pour `maxwell.smart` .
- En suivant le chemin `Clients` / `quotes-client` / `Settings` ajoutez les **Valid redirect URIs** ci-dessous :
  - `http://localhost:8080/*` notre backend `quotes`
  - `http://localhost:8900/*` notre passerelle
- En suivant le chemin `Clients` / `quotes-client` / `Credentials` , notez le secret du client.
- En utilisant la configuration fournie, les membres du groupe `golden_customers` assumeront automatiquement le rôle `gold` .

8 Service `quotes-api`

- Utilisez Spring Initializr [4] pour créer le projet `quotes-api` de gestion des devis. Adoptez les paramètres ci-dessous :
  - Java 21,
  - Group `fr` ,
  - Artifact `quotes` ,
  - Configuration `YAML` ,
  - Maven,

3. baeldung-realm.json
4. https://start.spring.io/

- ▷ Spring Boot 3.5.11,
- ▷ Dépendances : Spring Reactive Web, OAuth2 Resource Server, Spring Boot DevTools, Lombok

Les tickets opaques. Ces tickets sont une alternative aux JWT. Ils sont fabriqués par un serveur d'authentification et ne sont vérifiables et exploitables que par ce serveur. Ils ne contiennent pas de données (contrairement aux JWT) et l'extraction d'information passe par un processus d'introspection offert par le serveur d'authentification. (plus d'information ici [a]).

a. https://zitadel.com/blog/jwt-vs-opaque-tokens

- Enrichissez le fichier de configuration `application.yml` avec le code ci-dessous. Ces paramètres permettent de configurer un service de décodage des tokens (OpaqueToken [5]) qui va valider les tickets auprès de Keycloak.

```
server.port: 8080

spring.application.name: my_oauth_app

spring.security.oauth2.resourceserver.opaquetoken:
  introspection-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/token/introspect
  client-id: quotes-client
  client-secret: <CLIENT-SECRET>
```

- Ajoutez `@EnableWebFluxSecurity` à votre starter.
- Ajoutez `@EnableMethodSecurity(prePostEnabled = true,securedEnabled = true)` à votre starter.
- Ajoutez le contrôleur ci-dessous

5. https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/opaque-token.html

```java
@RestController
public class QuoteApi {

    @GetMapping("/quotes/{symbol}")
    public Mono<Quote> getQuote(@PathVariable("symbol") String symbol,
                                BearerTokenAuthentication auth ) {

        // Pour analyser les informations du token
        auth.getTokenAttributes().forEach((k,v)->{
            System.out.printf("Token key=%s, value=%s\n", k, v);
        });
        System.out.println("Roles : "+ extractRoles(auth));

        Quote q = new Quote();
        q.setSymbol(symbol);
        if (extractRoles(auth).contains("gold")) {
            q.setPrice(10.0);
        } else {
            q.setPrice(12.0);
        }
        return Mono.just(q);
    }

    @PreAuthorize("#auth.tokenAttributes['realm_access']['roles'].contains('gold')")
    @GetMapping("/quotes")
    public Flux<Quote> getQuotes(BearerTokenAuthentication auth ) {
        var q1 = new Quote("quote 1", 100.0);
        var q2 = new Quote("quote 2", 200.0);
        return Flux.just(q1,q2);
    }

    private List<String> extractRoles(BearerTokenAuthentication auth) {
        if (auth.getTokenAttributes().get("realm_access") instanceof Map<?,?> roles){
            if (roles.get("roles") instanceof List<?> listRoles){
                return listRoles.stream().map(role->role.toString()).toList();
            }
        }
        return List.of();
    }

}
```

- N'oubliez pas créer également la classe `Quote`.
- Vous pouvez maintenant lancer votre backend `quotes-api` :

```
mvn spring-boot:run -Pquotes-application
```

- Essayez de récupérer un devis (cela ne devrait pas fonctionner) :

```
curl -v http://localhost:8080/quotes/BAEL
```

9 Demander une authentification

- Tournons nous vers keycloak pour demander une authentification et récupérer un ticket :

```
# demander une authentification
curl -L -X POST \
  'http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/token' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'client_id=quotes-client' \
  --data-urlencode 'client_secret=<CLIENT-SECRET>' \
  --data-urlencode 'grant_type=password' \
  --data-urlencode 'scope=email roles profile' \
  --data-urlencode 'username=john.snow' \
  --data-urlencode 'password=snow'

{"access_token":"eyJhbGciOiJSUzI1...PWBtYi0vruw4aAQ",
"expires_in":300,"refresh_expires_in":1800,
"refresh_token":"eyJhbGciOiJIU...VqfKmq44Ky21Ww",
"token_type":"Bearer",
"not-before-policy":0,
"session_state":"eedc9e46-1b25-41d8-a7f0-50ffdffa8d93",
"scope":"profile email"}

# placer le token dans une variable
TOKEN="eyJhbGciOiJSUzI1...PWBtYi0vruw4aAQ"
```

- Vous pouvez maintenant utiliser le token dans l'application `quotes-api` :

```
curl --location --request GET 'http://localhost:8080/quotes/BAEL' \
--header 'Accept: application/json' \
--header "Authorization: Bearer $TOKEN"
```

- Vous devriez obtenir :

```
{"symbol":"BAEL","price":12.0}
```

- Si vous vous authentifiez avec l'utilisateur `Maxwell Smart` vous devriez avoir une baisse du prix.

10 Création de la passerelle

- Utilisez Spring Initializr[6] pour créer votre passerelle. Adoptez les paramètres ci-dessous :
  - ▷ Java 21,
  - ▷ Group `fr`,
  - ▷ Artifact `gateway`,
  - ▷ Configuration `YAML`,
  - ▷ Maven,
  - ▷ Spring Boot 3.5.11,
  - ▷ Dépendances : Reactive Gateway, Spring Reactive Web, OAuth2 Resource Server, OAuth2 Client, Spring Boot DevTools, Lombok
- Ajoutez `@EnableWebFluxSecurity` à votre starter.
- Préparez le fichier YAML `application.yml` :

6. https://start.spring.io/

```
server.port: 8900
spring.application.name: myGateway

spring.security.oauth2.resourceserver.opaquetoken:
  introspection-uri: http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/token/introspect
  client-id: quotes-client
  client-secret: <CLIENT-SECRET>

spring.cloud.gateway.server.webflux:
  routes:
    - id: quotes
      uri: http://localhost:8080
      predicates:
        - Path=/quotes/**
```

⏩ **Travail à faire :** Lancez la passerelle. Vous devriez pourvoir récupérer un devis (après authentification) via la passerelle (port `8900`) sans passer directement par l'application.

11 Spring Gateway as OAuth 2.0 Client

- Pour la classe de démarrage, nous utiliserons la même que celle déjà présente pour la version serveur de ressources. Cela nous permettra de souligner que le comportement de sécurité repose entièrement sur les bibliothèques et propriétés disponibles.
- En réalité, la seule différence notable entre les deux versions réside dans les propriétés de configuration. Il est nécessaire de configurer les détails du fournisseur soit via la propriété `issuer-uri`, soit via les paramètres individuels des différents points de terminaison (autorisation, jeton et introspection).
- Nous devons également définir les détails d'enregistrement de notre client d'application, notamment les étendues (`scope`) demandées. Ces étendues indiquent au fournisseur d'identité (IdP) quel ensemble d'informations sera accessible via le mécanisme d'introspection.
- Préparez une nouvelle version du fichier YAML `application.yml`. Clairement notre passerelle devient un client OAuth 2.0 et nous configurons les éléments qui permettent une fabrication et vérification du token d'authentification.

```
server.port: 8900
spring.application.name: myGateway

spring.security.oauth2.client:
  provider:
    keycloak:
      issuer-uri: http://localhost:8083/auth/realms/baeldung
  registration:
    quotes-client:
      provider: keycloak
      client-id: quotes-client
      client-secret: <CLIENT-SECRET>
      authorization-grant-type: authorization_code
      scope:
        - email
        - profile
        - roles
        - openid
```

- Nous ajoutons à `application.yml` la défintion des routes et l'utilisation du filtre spring cloud TokenRelay [7]

7. https://docs.spring.io/spring-cloud-gateway/reference/4.3/spring-cloud-gateway-server-webflux/gatewayfilter-factories/tokenrelay-factory.html

qui se charge de transmettre les tokens aux services.

```
spring.cloud.gateway.server.webflux:
  routes:
    - id: quotes
      uri: http://localhost:8080
      predicates:
        - Path=/quotes/**
      filters:
        - TokenRelay=
```

- Sinon, si nous voulons que toutes les routes déclenchent un flux d'autorisation, nous pouvons ajouter le filtre TokenRelay à la section `default-filters` :

```
spring.cloud.gateway.server.webflux:
  default-filters:
    - TokenRelay=
  routes:
    - id: quotes
      uri: http://localhost:8080
      predicates:
        - Path=/quotes/**
```

⏩ Nous allons tester la nouvelle configuration

- Lancez la passerelle : `mvn spring-boot:run`
- Dans un navigateur, tentez d'utiliser `quotes-api` via la passerelle avec l'adresse : `http://localhost:8900/quotes/doors`. Vous devriez être renvoyé vers la page d'authentification de Keycloak. Utilisez l'utilisateur `maxwell.smart` (pour avoir les droits `gold`).

- Après l'authentification, vérifiez que vous obtenez le bon résultat.
- Faites un nouvel essai avec `http://localhost:8900/quotes`.
- Faites les mêmes essais avec l'utilisation `john.snow`.
- Dans Keycloak, avec le chemin `Users` / `john.snow` / `Sessions` vous pouvez observer les sessions ouvertes et même les supprimer. Faites un essai et vérifiez l'impact.

⏩ Déconnexion

- Utilisez l'URL ci-dessous pour vous déconnecter :

`http://localhost:8083/auth/realms/baeldung/protocol/openid-connect/logout`

12 Une autre application

- Par duplication de `quotes-api`, créez une application `callme` qui va écouter sur le port `8090`.
- Le contrôleur sera

```
@RestController
public class CallmeControler {

    @GetMapping("/callme/ping")
    @PreAuthorize("#auth.tokenAttributes['realm_access']['roles'].contains('gold')")
    public Mono<String> ping(BearerTokenAuthentication auth) {
        return Mono.just("Im␣a␣running");
    }
}
```

- En suivant le chemin `Clients` / `quotes-client` / `Settings` ajoutez les **Valid redirect URIs** ci-dessous :
  - ▷ `http://localhost:8090/*` nouvelle application
- Ajuster la configuration de la passerelle pour prévoir ce service :

```
spring.cloud.gateway.server.webflux:
  default-filters:
    - TokenRelay=
  routes:
    - id: quotes
      uri: http://localhost:8080
      predicates:
        - Path=/quotes/**
    - id: callme
      uri: http://localhost:8090
      predicates:
        - Path=/callme/**
```

- Tester ce service via la passerelle `http://localhost:8900/callme/ping`.
- Ajouter à `quotes-api` la définition d'un client WEB :

```
@Bean
public WebClient webClient() {
    return WebClient.builder()
            .filter(new ServletBearerExchangeFilterFunction())
            .build();
}
```

- Ajouter à `quotes-api` une entrée qui va appeler l'application `callme` :

```
@Autowired
WebClient webClient;

@PreAuthorize("#auth.tokenAttributes['realm_access']['roles'].contains('gold')")
@GetMapping("/quotes/ping")
public Mono<String> ping(BearerTokenAuthentication auth) {
    var token = auth.getToken().getTokenValue();
    return webClient
            .get()
            .uri("http://localhost:8090/callme/ping")
            .header("Authorization","Bearer␣"+token)
            .retrieve()
            .bodyToMono(String.class)
            .map(v -> v + "␣(via␣quotes-ai)");
}
```

Moralité : nous avons maintenant deux services et le premier est capable d'appeler le second et lui passer le ticket d'authentification. Ils partagent donc le système d'authentification.