Arun B Chandrasekaran

Follow 167 Followers About

Photo by MILKOVÍ on Unsplash

Securing REST API using Keycloak and Spring Oauth2

Arun B Chandrasekaran Nov 19, 2018 · 7 min read

Keycloak is Open Source Identity and Access Management Server, which is a OAuth2 and OpenID Connect(OIDC) protocol complaint. This article is to explain how Spring Boot REST APIs can be secured with Keycloak using Spring OAuth2 library.

Keycloak documentation suggest 3 ways to secure Spring based REST APIS.

1. Using Keycloak Spring Boot Adapter

2. Using keycloak Spring Security Adapter

Let us see how we can use Keycloak OIDC support and Spring OAuth2 library to **secure REST APIs**. Benefits Of Using Spring OAuth2 Over Keycloak Adapter is explained at the end of this article.

Let us explore how to setup Keycloak and interact with it using Spring OAuth2 library.

This is a lengthy article with step by step instructions, screenshots, code snippets. Complete code is available on github. I recommend reading this article before looking into the code.

Step 1: Getting Started With Keycloak

Refer Keycloak getting started documentation to run and setup keycloak admin user.

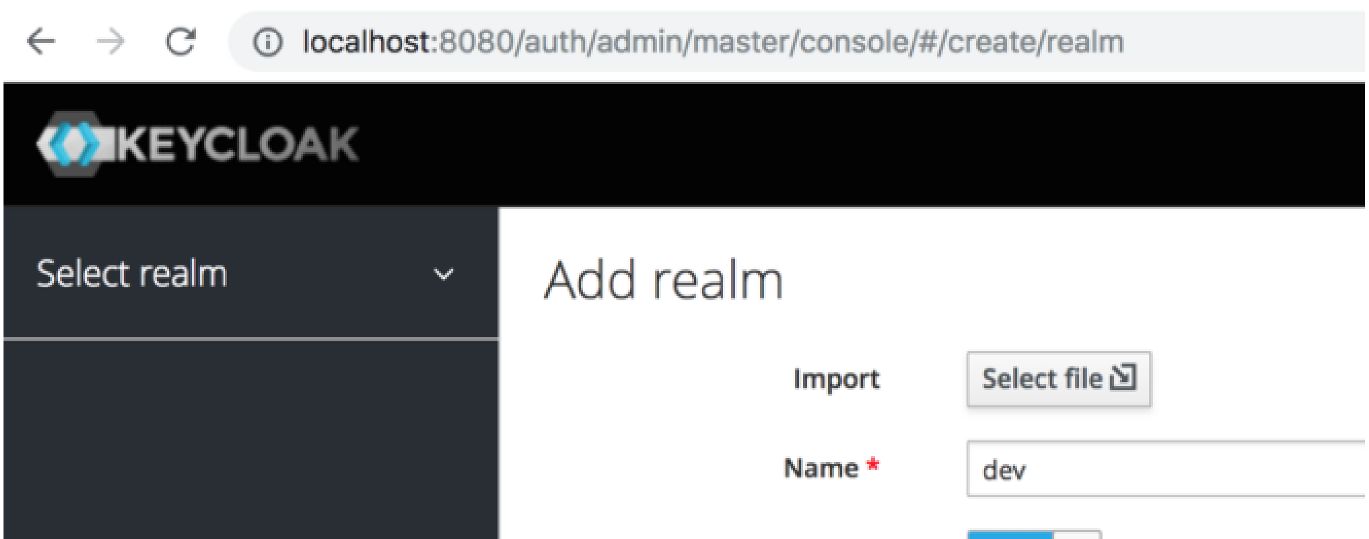
After running Keycloak, access keycloak admin console using http://localhost:8080/auth

Setup keycloak username=admin, password=admin.

Note: *Standalone Keycloak runs on Wildfly server. Don't worry about configuring a user to manage Wildfly server. We need a Keycloak admin user to create realm, client, user, role etc in Keycloak.*

Step 2: Create dev Realm

```
Name: dev
```


localhost:8080/auth/admin/master/console/#/create/realm

Add realm

Import: Select file

Name *: dev

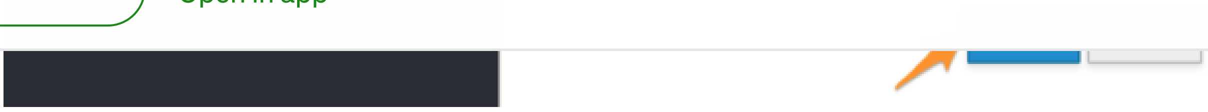
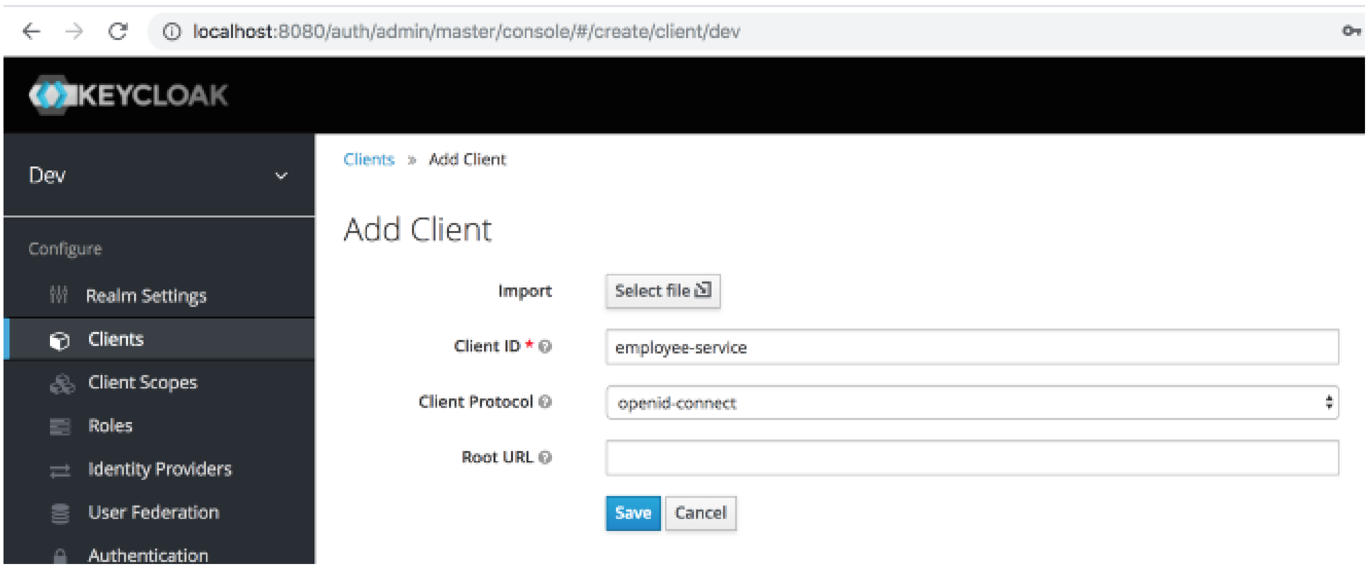


Figure 1: add dev realm

Step 3: Create a Client (Micro-Service)

```
Client ID       : employee-service
Client Protocol : openid-connect
```


localhost:8080/auth/admin/master/console/#/create/client/dev

Add Client

Import: Select file
Client ID *: employee-service
Client Protocol: openid-connect
Root URL:

Figure 2: Add client

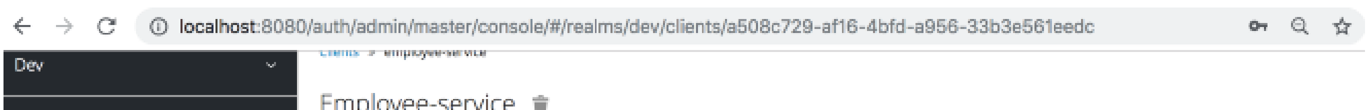
Step 4: Configure Client

If Keycloak runs on Port 8080, make sure your microservice runs on another port. In the example, micro-service is configured to run on 8085.

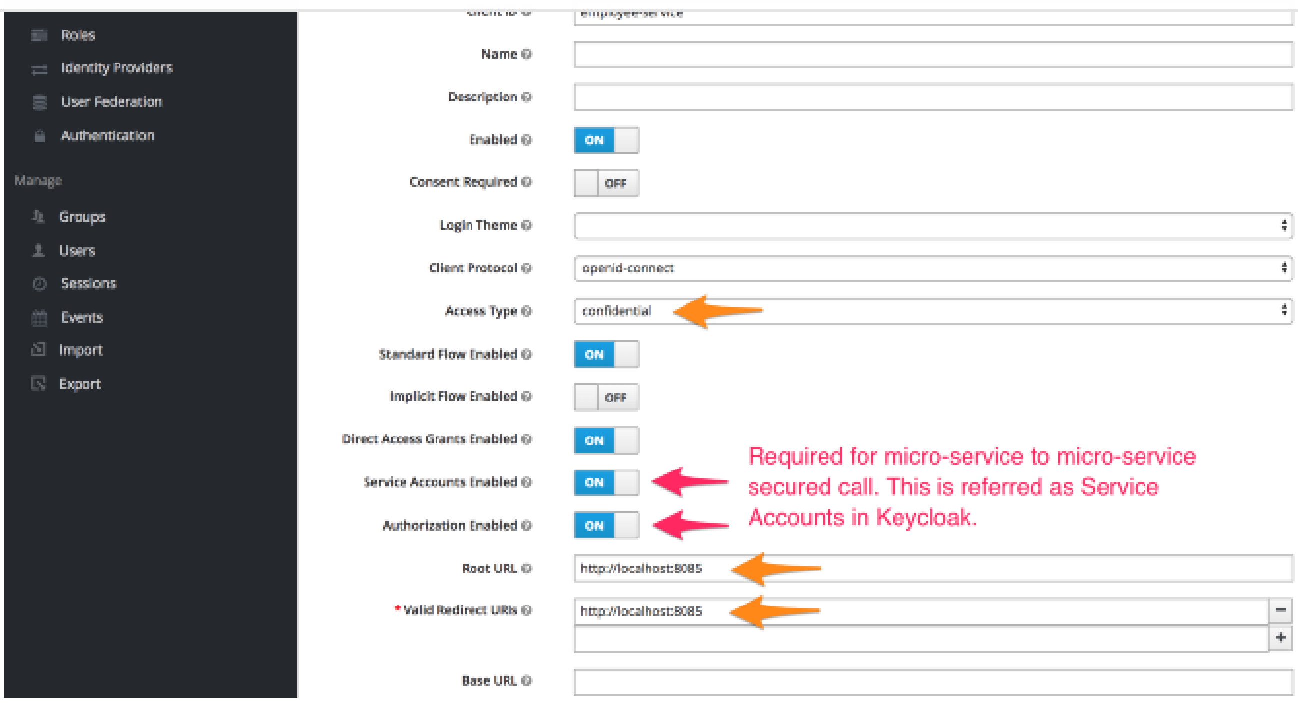
```
Access Type         : confidential
Valid Redirect URIs : http://localhost:8085

# Required for micro-service to micro-service secured calls
Service Accounts Enabled : On
Authorization Enabled : On
```

Note: *Access Type confidential supports getting access token using client credentials grant as well as authorization code grant. If a micro-service need to call another micro-service, caller will be 'confidential' and callee will be 'bearer-only'.*


localhost:8080/auth/admin/master/console/#/realms/dev/clients/a508c729-af16-4bfd-a956-33b3e561eedc

Employee-service



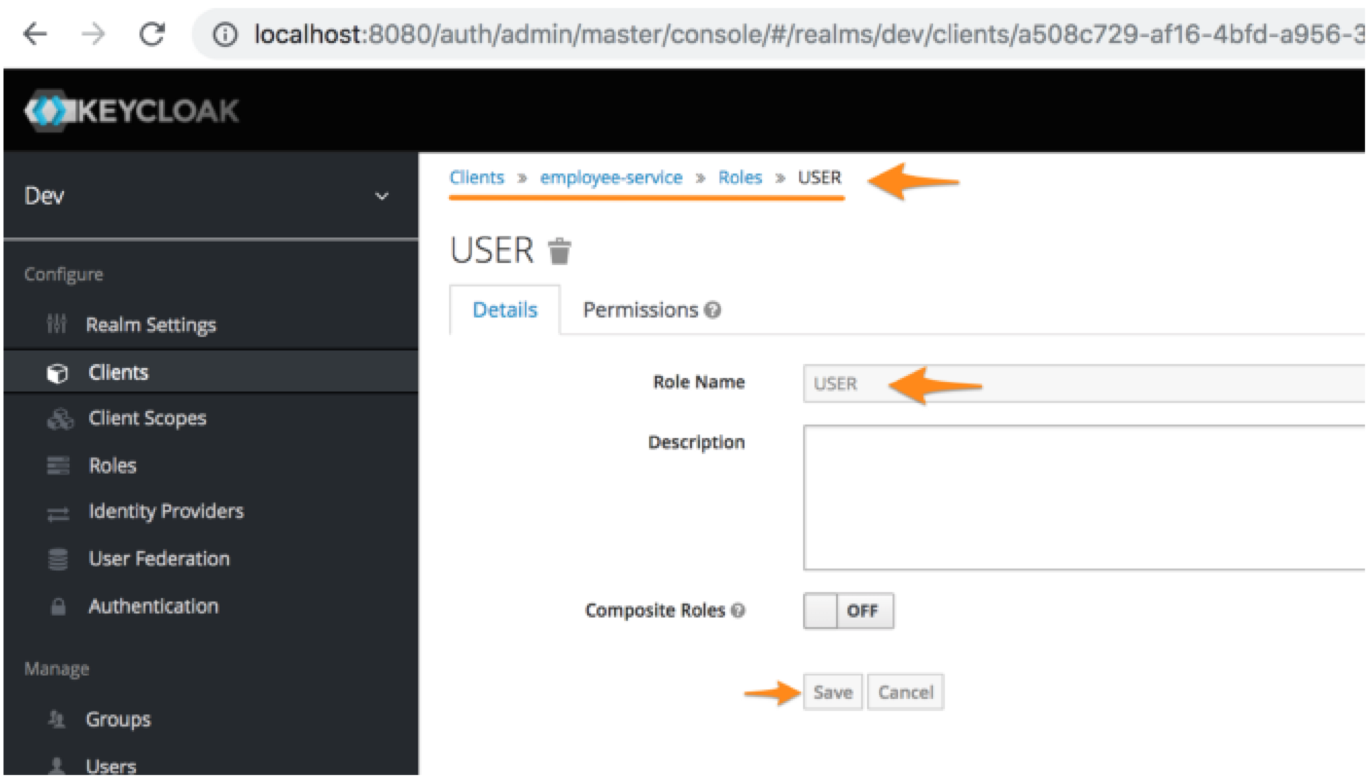
Client ID	employee-service
Name	
Description	
Enabled	ON
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	confidential
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON
Service Accounts Enabled	ON
Authorization Enabled	ON
Root URL	http://localhost:8085
* Valid Redirect URIs	http://localhost:8085
Base URL	

Required for micro-service to micro-service secured call. This is referred as Service Accounts in Keycloak.

Figure 2: Configure client

Step 5: Create Client Role

Create a role under the client. In this case, role USER is created under employee-service.



localhost:8080/auth/admin/master/console/#/realms/dev/clients/a508c729-af16-4bfd-a956-3

Clients » employee-service » Roles » USER

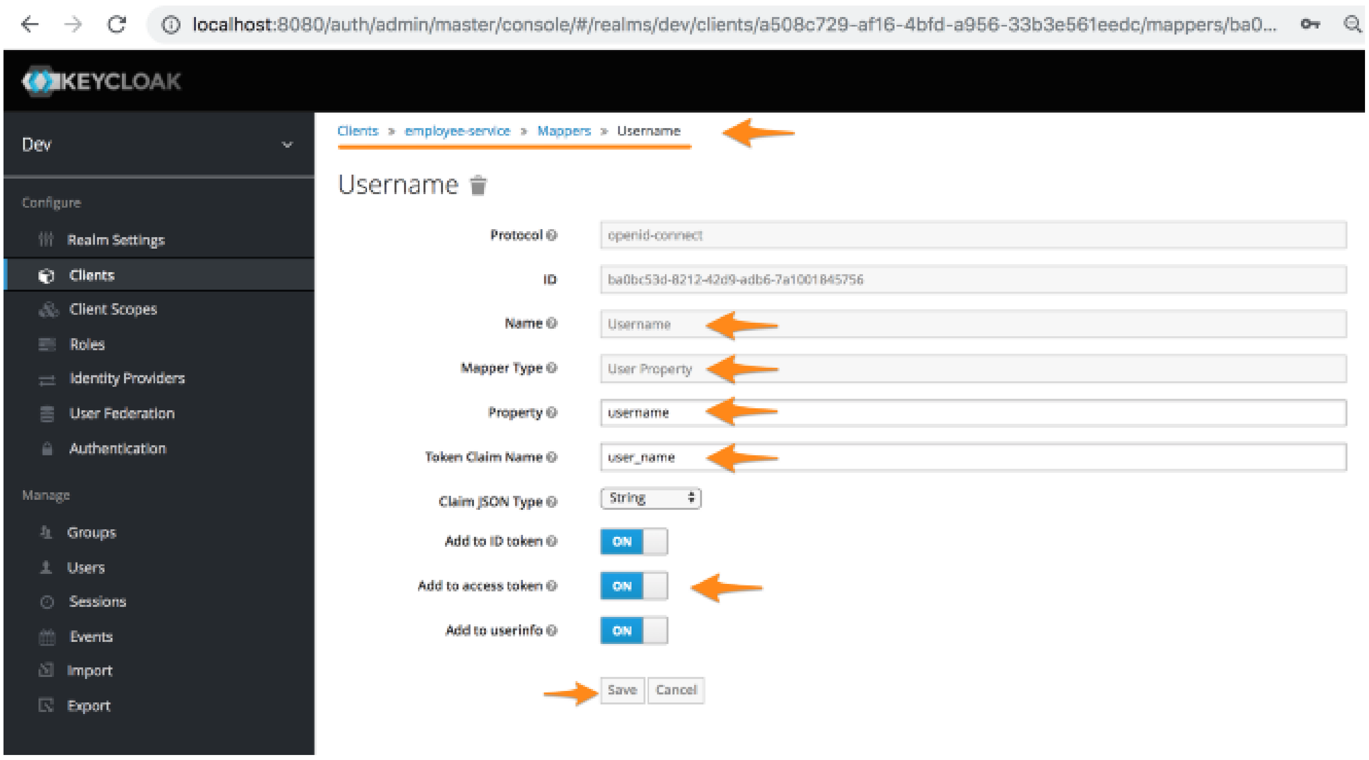
USER

Role Name	USER
Description	
Composite Roles	OFF

Figure 3: Create role

Step 6: Create a Mapper (To get user_name in access token)

Keycloak access token is a JWT. It is a JSON and each field in that JSON is called a claim. By default, logged in username is returned in a claim named “preferred_username” in access token. Spring Security OAuth2 Resource Server expects username in a claim

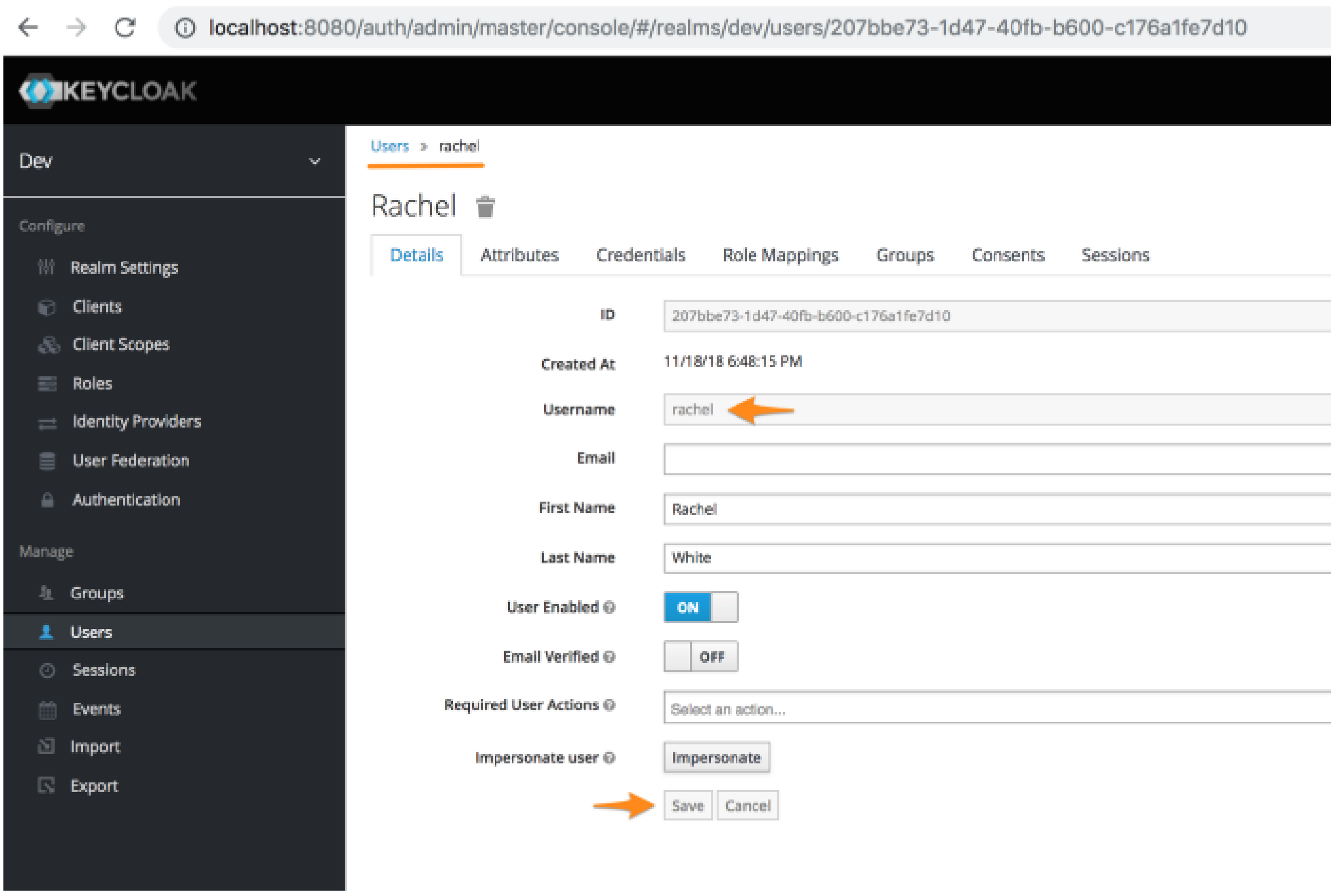

localhost:8080/auth/admin/master/console/#/realms/dev/clients/a508c729-af16-4bfd-a956-33b3e561eedc/mappers/ba0...

Username

Protocol	openid-connect
ID	ba0bc53d-8212-42d9-adb6-7a1001845756
Name	Username
Mapper Type	User Property
Property	username
Token Claim Name	user_name
Claim JSON Type	String
Add to ID token	ON
Add to access token	ON
Add to userinfo	ON

Figure 4: Create Mapper

Step 7: Create User


localhost:8080/auth/admin/master/console/#/realms/dev/users/207bbe73-1d47-40fb-b600-c176a1fe7d10

Rachel

ID	207bbe73-1d47-40fb-b600-c176a1fe7d10
Created At	11/18/18 6:48:15 PM
Username	rachel
Email	
First Name	Rachel
Last Name	White
User Enabled	ON
Email Verified	OFF
Required User Actions	Select an action...
Impersonate user	Impersonate

Figure 5: Create User

assigned/mapped to user.

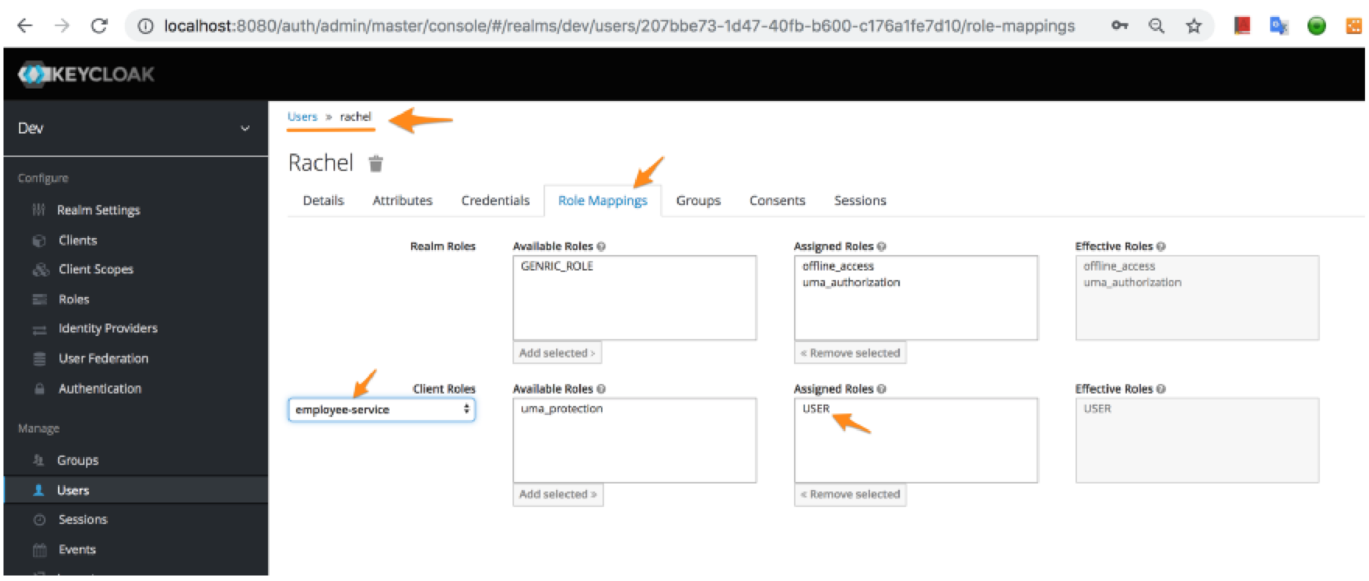


Figure 6: Assign role to user

Step 9: Get Configuration From OpenID Configuration Endpoint

Because Keycloak is OpenID Connect and OAuth2 complaint, below is OpenID Connection configuration URL to get details about all security endpoints,

GET **http://localhost:8080/auth/realms/dev/.well-known/openid-configuration**

Important URLS to be copied from response:

issuer : http://localhost:8080/auth/realms/dev

authorization_endpoint: ${**issuer**}/protocol/openid-connect/auth

token_endpoint: ${**issuer**}/protocol/openid-connect/token

token_introspection_endpoint: ${**issuer**}/protocol/openid-connect/token/introspect

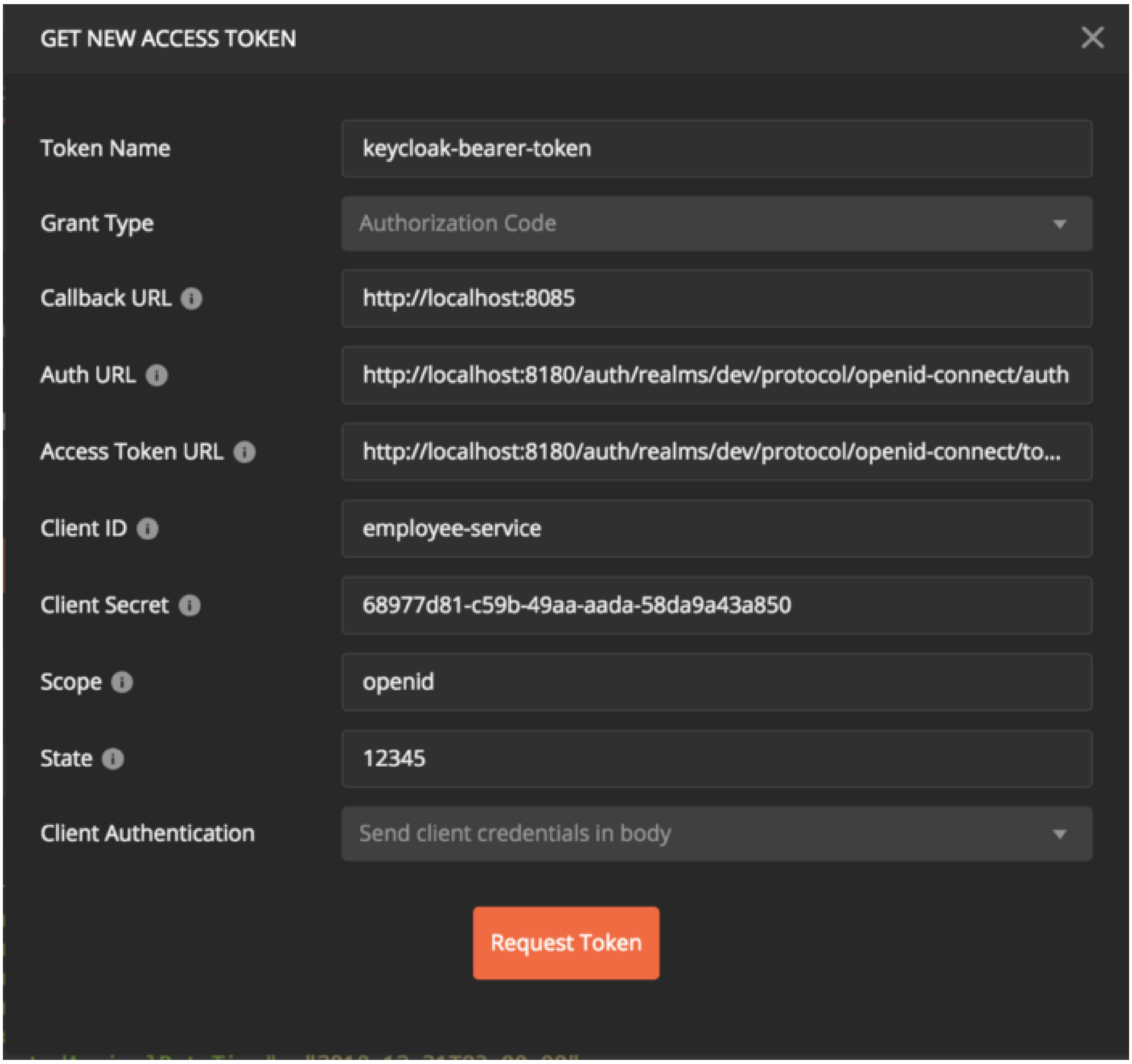
userinfo_endpoint: ${**issuer**}/protocol/openid-connect/userinfo

Response also contains grant types and scopes supported

grant_types_supported: ["client_credentials", …]

scopes_supported: ["openid", …]

To Get Access Token Using Postman (For Testing)



Token Name	keycloak-bearer-token
Grant Type	Authorization Code
Callback URL	http://localhost:8085
Auth URL	http://localhost:8180/auth/realms/dev/protocol/openid-connect/auth
Access Token URL	http://localhost:8180/auth/realms/dev/protocol/openid-connect/to...
Client ID	employee-service
Client Secret	68977d81-c59b-49aa-aada-58da9a43a850
Scope	openid
State	12345
Client Authentication	Send client credentials in body

Postman tool screenshot: To get access token from keycloak for a client

- Make sure you select client authentication as “Send client credentials in body” while requesting token.
- Callback URL is redirect URL configured in Keycloak.
- Client secret may be different for you, copy the one from client configuration in keycloak.
- You may also use https://jwt.io to inspect the contents of token received.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.security.oauth.boot</groupId>
  <artifactId>spring-security-oauth2-autoconfigure</artifactId>
  <version>2.0.1.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Step 11: Configure application.properties

General Security Properties

```
# Can be set to false to disable security during local development
rest.security.enabled=true

rest.security.api-matcher=/api/**
rest.security.cors.allowed-origins=*
rest.security.cors.allowed-headers=*
rest.security.cors.allowed-methods=GET,POST,PUT,PATCH,DELETE,OPTIONS
rest.security.cors.max-age=3600
```

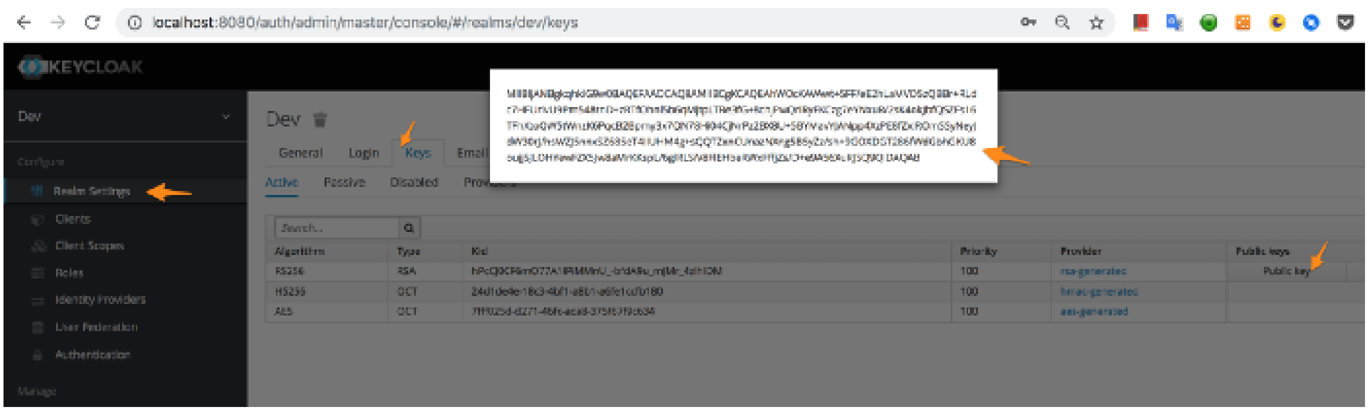
Properties to secure REST Endpoints using OAuth2 Resource Server

```
security.oauth2.resource.id=employee-service

security.oauth2.resource.token-info-uri=${rest.security.issuer-uri}/protocol/openid-connect/token/introspect

security.oauth2.resource.user-info-uri=${rest.security.issuer-uri}/protocol/openid-connect/userinfo

security.oauth2.resource.jwt.key-value=-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhWOcKAVAwt+5FF/eE2hLaMVD5zQBBr+RLdc7HFUrlvU9Pm548rnD+zRTfOhnl5b6qMjtpLTRe3fG+8chjPwQriRyFKCzg7eYNxuR/2sK4okJbfQSZFs16TFhXtoQW5tWnzK6PqcB2Bpmy3x7QN78Hi04CjNrPz2BX8U+5BYMavYJANpp4XzPE8fZxlROmSSyNeyJdW30rJ/hsWZJ5nnxSZ685eT4IIUHM4g+sQQTZxnCUnazNXng5B5yZz/sh+9GOXDGT286fWdGbhGKU8oujjSJLOHYewFZX5Jw8aMrKKspL/6glRLSiV8FlEHbeRWxFffjZs/D+e9A56XuRJSQ9QIDAQAB\n-----END PUBLIC KEY-----
```



Algorithm	Type	Kid	Priority	Provider	Public keys
RS256	RSA	hPcQ0CF6mO77A1IRMMnU_-b'dA8u_mjMr_4zHON	100	rsa-generated	Public key
HS256	OCT	24d1de4e-18c3-4bf1-a8b1-a6fe1cd1b180	100	hmac-generated	
AES	OCT	7ff025d-d271-46fc-aca8-375f57f9c634	100	aes-generated	

Figure 7: Copy jwt public key value

Note 1: *security.oauth2.resource.jwt.key-value property value can be copied from public key at realm level. This is very important and this property is what uses* ***JwtAccessTokenCustomizer*** *which we will see later.*

Note 2: *Property values will be different based on your configuration, care should be take to use correct values.*

Properties to call another micro-service (Service Accounts)

```
# If this micro-services that needs to call another
# secured micro-service
security.oauth2.client.client-id=employee-service

security.oauth2.client.client-secret=68977d81-c59b-49aa-aada-58da9a43a850

security.oauth2.client.user-authorization-uri=${rest.security.issuer-uri}/protocol/openid-connect/auth
security.oauth2.client.scope=openid
security.oauth2.client.grant-type=client_credentials
```

***Note:** Above properties are required for **OAuth2RestTemplate** that is used to make secure service account calls.*

Step 12: JWT Access Token Customizer

In order for Spring OAuth2 to parse and set SecurityConextHolder, it needs the roles or authorities from token. Also, in order to determine the list of clients/application/micro-service a user has access, it needs the list of client ids from token. This is the only setup that needs some special handling.

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.security.oauth2.resource.JwtAccessTokenConverterCo
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.oauth2.provider.OAuth2Authentication;
import org.springframework.security.oauth2.provider.OAuth2Request;
import org.springframework.security.oauth2.provider.token.DefaultAccessTokenConverter;
import org.springframework.security.oauth2.provider.token.store.JwtAccessTokenConverter;

public class JwtAccessTokenCustomizer extends DefaultAccessTokenConverter
                                      implements JwtAccessTokenConverterConfigurer {

  private static final Logger LOG = LoggerFactory.getLogger(JwtAccessTokenCustomizer.class);

  private static final String CLIENT_NAME_ELEMENT_IN_JWT = "resource_access";

  private static final String ROLE_ELEMENT_IN_JWT = "roles";

  private ObjectMapper mapper;

  public JwtAccessTokenCustomizer(ObjectMapper mapper) {
    this.mapper = mapper;
    LOG.info("Initialized {}", JwtAccessTokenCustomizer.class.getSimpleName());
```

```java
    public void configure(JwtAccessTokenConverter converter) {
      converter.setAccessTokenConverter(this);
      LOG.info("Configured {}", JwtAccessTokenConverter.class.getSimpleName());
    }

    /**
     * Spring oauth2 expects roles under authorities element in tokenMap,
     * but keycloak provides it under resource_access. Hence extractAuthentication
     * method is overriden to extract roles from resource_access.
     *
     * @return OAuth2Authentication with authorities for given application
     */
    @Override
    public OAuth2Authentication extractAuthentication(Map<String, ?> tokenMap) {
      LOG.debug("Begin extractAuthentication: tokenMap = {}", tokenMap);
      JsonNode token = mapper.convertValue(tokenMap, JsonNode.class);
      Set<String> audienceList = extractClients(token); // extracting client names
      List<GrantedAuthority> authorities = extractRoles(token); // extracting client roles

      OAuth2Authentication authentication = super.extractAuthentication(tokenMap);
      OAuth2Request oAuth2Request = authentication.getOAuth2Request();

      OAuth2Request request =
          new OAuth2Request(oAuth2Request.getRequestParameters(),
              oAuth2Request.getClientId(),
              authorities, true,
              oAuth2Request.getScope(),
              audienceList, null, null, null);

      Authentication usernamePasswordAuthentication =
              new UsernamePasswordAuthenticationToken(authentication.getPrincipal(),
              "N/A", authorities);

      LOG.debug("End extractAuthentication");
      return new OAuth2Authentication(request, usernamePasswordAuthentication);
    }

    private List<GrantedAuthority> extractRoles(JsonNode jwt) {
      LOG.debug("Begin extractRoles: jwt = {}", jwt);
      Set<String> rolesWithPrefix = new HashSet<>();

      jwt.path(CLIENT_NAME_ELEMENT_IN_JWT)
          .elements()
          .forEachRemaining(e -> e.path(ROLE_ELEMENT_IN_JWT)
              .elements()
              .forEachRemaining(r -> rolesWithPrefix.add("ROLE_" + r.asText())));

      final List<GrantedAuthority> authorityList =
      return authorityList;
    }

    private Set<String> extractClients(JsonNode jwt) {
      LOG.debug("Begin extractClients: jwt = {}", jwt);
      if (jwt.has(CLIENT_NAME_ELEMENT_IN_JWT)) {
        JsonNode resourceAccessJsonNode = jwt.path(CLIENT_NAME_ELEMENT_IN_JWT);
        final Set<String> clientNames = new HashSet<>();
        resourceAccessJsonNode.fieldNames()
            .forEachRemaining(clientNames::add);

        LOG.debug("End extractClients: clients = {}", clientNames);
        return clientNames;

      } else {
        throw new IllegalArgumentException("Expected element " +
                  CLIENT_NAME_ELEMENT_IN_JWT + " not found in token");
      }
    }
  }
```

securing-rest-api-using-keycloak_access-token-customizer.java hosted with ♡ by GitHub view raw

Step 13: OAuth2 Resource Server Configurer

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.boot.autoconfigure.security.oauth2.resource.ResourceServerProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSec
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.oauth2.client.OAuth2RestTemplate;
import org.springframework.security.oauth2.client.resource.OAuth2ProtectedResourceDetails;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceSer
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerCon
import org.springframework.security.oauth2.config.annotation.web.configurers.ResourceServerSecur
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

@Configuration
@EnableWebSecurity
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

```java
public class SecurityConfigurer extends ResourceServerConfigurerAdapter {

  @Autowired
  private ResourceServerProperties resourceServerProperties;

  @Autowired
  private SecurityProperties securityProperties;

  @Override
  public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
    resources.resourceId(resourceServerProperties.getResourceId());
  }


  @Override
  public void configure(final HttpSecurity http) throws Exception {

    http.cors()
        .configurationSource(corsConfigurationSource())
        .and()
        .headers()
        .frameOptions()
        .disable()
        .and()
        .csrf()
        .disable()
        .authorizeRequests()
        .antMatchers(securityProperties.getApiMatcher())
        .authenticated();

  }

  @Bean
  public CorsConfigurationSource corsConfigurationSource() {
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    if (null != securityProperties.getCorsConfiguration()) {
      source.registerCorsConfiguration("/**", securityProperties.getCorsConfiguration());
    }
    return source;
  }

  @Bean
  public JwtAccessTokenCustomizer jwtAccessTokenCustomizer(ObjectMapper mapper) {
    return new JwtAccessTokenCustomizer(mapper);
  }

  @Bean
  public OAuth2RestTemplate oauth2RestTemplate(OAuth2ProtectedResourceDetails details) {
    OAuth2RestTemplate oAuth2RestTemplate = new OAuth2RestTemplate(details);
```

```
        oAuth2RestTemplate.getAccessToken();
        return oAuth2RestTemplate;
    }
}
```

securing-rest-api-using-keycloak_security-configurer.java hosted with ♡ by GitHub view raw

Note: *OAuth2RestTemplate is required if this micro-service needs to call another micro-service.*

Step 14: Secure REST Endpoints

PreAuthorize annotation is use to secure REST endpoints with appropriate roles. Refer below example.

```
import java.util.Set;
import org.arun.springoauth.config.SecurityContextUtils;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/v1/employees")
public class EmployeeRestController {

  @GetMapping(path = "/username")
  @PreAuthorize("hasAnyAuthority('ROLE_USER')")
  public ResponseEntity<String> getAuthorizedUserName() {
    return ResponseEntity.ok(SecurityContextUtils.getUserName());
  }

  @GetMapping(path = "/roles")
  @PreAuthorize("hasAnyAuthority('ROLE_USER')")
  public ResponseEntity<Set<String>> getAuthorizedUserRoles() {
    return ResponseEntity.ok(SecurityContextUtils.getUserRoles());
  }
}
```

securing-rest-api-using-keycloak_role-rest-controller.java hosted with ♡ by GitHub view raw

Step 15: Disable Basic Auth if not required

In order to disable default security, **SecurityAutoConfiguration** and **UserDetailsServiceAutoConfiguration** can be excluded.

```java

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

securing-rest-api-using-keycloak_disable-basic-auth-security.java hosted with ♡ by GitHub view raw

For Complete Code

Refer: https://github.com/bcarun/spring-oauth2-keycloak-connector

Benefits Of Using Spring OAuth2 Over Keycloak Adapters:

1. Most of the time, Keycloak Server upgrade requires upgrading Keycloak adapter as well. If Keycloak adapters were used in 100s of micro-services, then all these micro-services needs to be upgraded to use newer version of Keycloak adapter. This requires regression testing of all micro-services and it is time consuming. This can be avoided by using Spring OAuth2 which integrates with Keycloak at protocol level. As these protocols don't change often and Keycloak Server upgrade will continue to support the respective protocol version, no change is required in micro-services when Keycloak Server is upgraded.

2. Sometimes, Spring Boot version upgrade requires Keycloak and Keycloak Spring Boot adapter or Keycloak Spring Security adapter to be upgraded. This can be avoided if we use Spring OAuth2.

3. If organization decides to migrate from Keycloak to another OAuth2 OpenID Authx Provider, all the micro-services that used Keycloak Spring Boot adapter and Keyclock Spring Security needs to be refactored. Using Spring OAuth2 + Spring Security can significantly simplify the migrations.

4. OAuth2RestTemplate class available in Spring OAuth2 takes care of refreshing on need and caching access tokens using OAuth2Context. This is a great benefit when there is a need to securely interact between micro-services.

To understand accessing secured micro-service from another micro-service using **OAuth2RestTemplate**, refer my other article stated below.

Accessing Secure REST API using Spring OAuth2RestTemplate

link.medium.com

References:

OAuth 2 Simplified

This post describes OAuth 2.0 in a simplified format to help developers and service providers implement the protocol...

aaronparecki.com

5 Easy Steps to Understanding JSON Web Tokens (JWT)

In this article, the fundamentals of what JSON Web Tokens (JWT) are, and why they are used will be explained. JWT are...

medium.com

How can one validate with a public key a JWT signature generated with a private key?

As I understand it, a JSON Web Token (JWT) consists of 3 parts: the header, specifying the hashing algorithm to use for...

crypto.stackexchange.com

Other OpenID Connect libraries | Keycloak Documentation

The Authorization Code flow redirects the user agent to Keycloak. Once the user has successfully authenticated with...

www.keycloak.org

Spring Security OAuth

This is the user guide for the support for . For OAuth 1.0, everything is different, so see its user guide. This user...

projects.spring.io