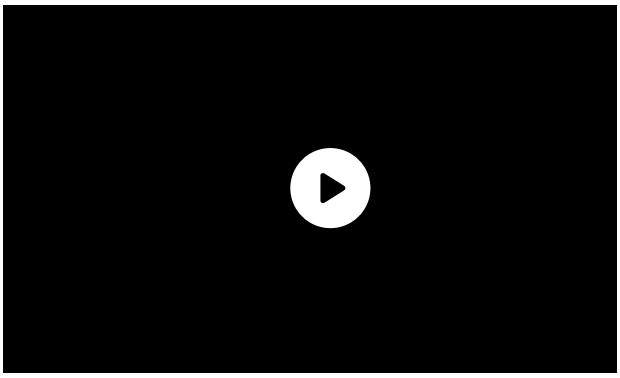
(/)

A Quick Guide to Using Keycloak With Spring Boot



Last updated: January 30, 2024

Written by: Michael Good (https://www.baeldung.com/author/michael-good)

Reviewed by: Michal Aibin (https://www.baeldung.com/editor/michal-author)

Security (https://www.baeldung.com/category/security)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot)

Keycloak (https://www.baeldung.com/tag/keycloak) **reference**

1. Overview (/)

In this tutorial, we'll discuss the basics of setting up a Keycloak server and connecting a Spring Boot application to it using Spring Security OAuth2.0.

Further reading:

Spring Security and OpenID Connect (/spring-security-openid-connect)

Learn how to set up OpenID Connect (from Google) with a simple Spring Security application.

Read more (/spring-security-openid-connect) →

Simple Single Sign-On with Spring Security OAuth2 (/sso-spring-security-oauth2)

A simple SSO implementation using Spring Security 5 and Boot.

Read more (/sso-spring-security-oauth2) →

CAS SSO With Spring Security (/spring-security-cas-sso)

Learn how to integrate the Central Authentication Service (CAS) with Spring Security.

Read more (/spring-security-cas-sso) →

2. What Is Keycloak?

Keycloak (http://www.keycloak.org/) is an open-source Identity and Access Management (/cs/iam-security) solution targeted towards modern applications and services.

Keycloak offers features such as Single-Sign-On (SSO), Identity Brokering and Social Login, User Federation, Client Adapters, an Admin Console, and an Account Management Console.

In our tutorial, we'll use the Admin Console of Keycloak for setting up and connecting to Spring Boot using the Spring Security OAuth2.0.

3. Setting Up a Keycloak Server

In this section, we will set up and configure the Keycloak server.

3.1. Downloading and Installing Keycloak

There are several distributions to choose from. However, in this tutorial, we'll be using the standalone version.

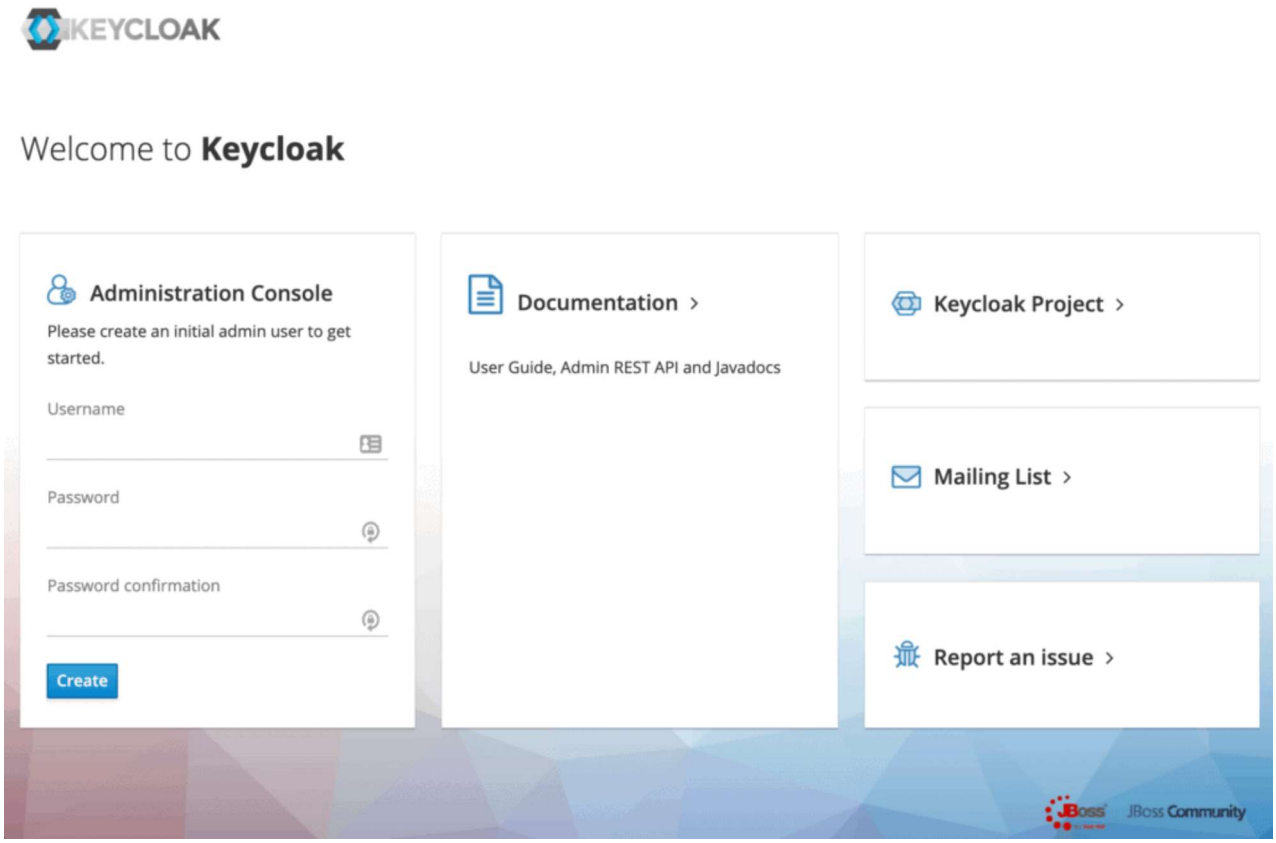
Let's download the Keycloak-22.0.3 Standalone server distribution (https://www.keycloak.org/archive/downloads-22.0.3.html) from the official source.

Once we've downloaded the Standalone server distribution, we can unzip and start Keycloak from the terminal:

```
$ unzip keycloak-22.0.3.zip
$ cd keycloak-22.0.3
$ bin/kc.sh start-dev
```

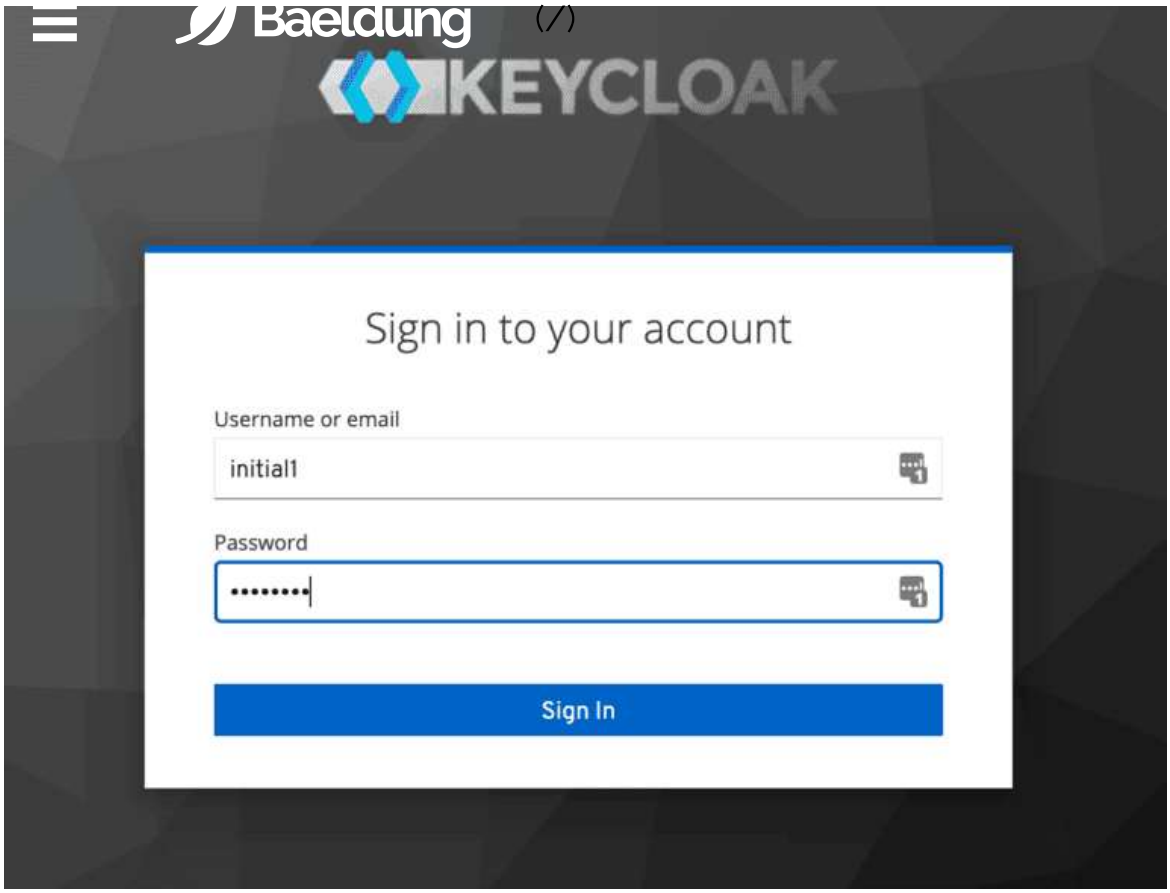
After running these commands, Keycloak will be starting its services. Once we see a line containing *Keycloak 22.0.3 [...] started*, we'll know its start-up is complete.

Now let's open a browser and visit *http://localhost:8080 (http://localhost:8080)*. **We'll be redirected to *http://localhost:8080/auth* to create an administrative login:**



(/wp-content/uploads/2017/11/keycloak1.png)
Let's create an initial admin user named *initial1* with the password *zaq1!QAZ*. Upon clicking *Create*, we'll see the message *User Created*.

We can now proceed to the Administrative Console. On the login page, we'll enter the initial admin user credentials:



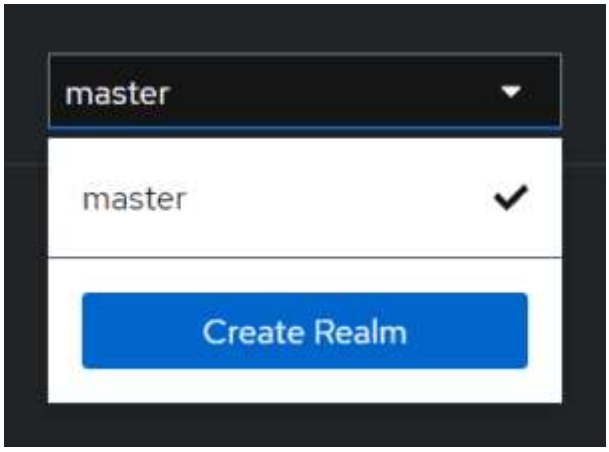
(/wp-content/uploads/2021/06/1_keycloak_admin_console.png)

3.2. Creating a Realm

A successful login will take us to the console and open up the default *Master* realm for us.

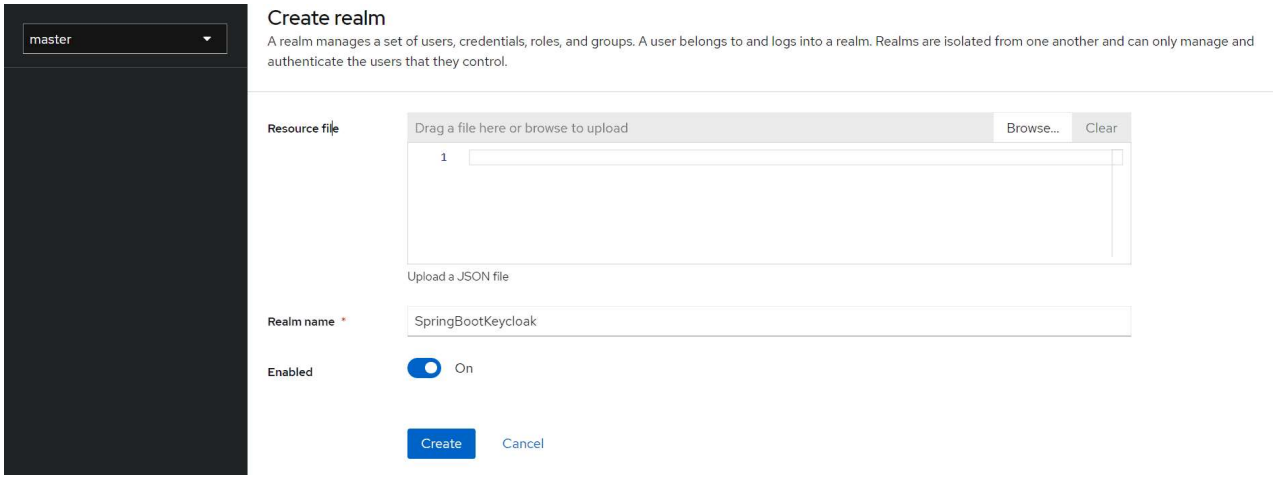
Here we'll focus on creating a custom realm.

Let's **navigate to the upper left corner to discover the *Create realm* button:**



(/wp-content/uploads/2017/11/create-realm.png)

On the next screen, **let's add a new realm called *SpringBootKeycloak*:**



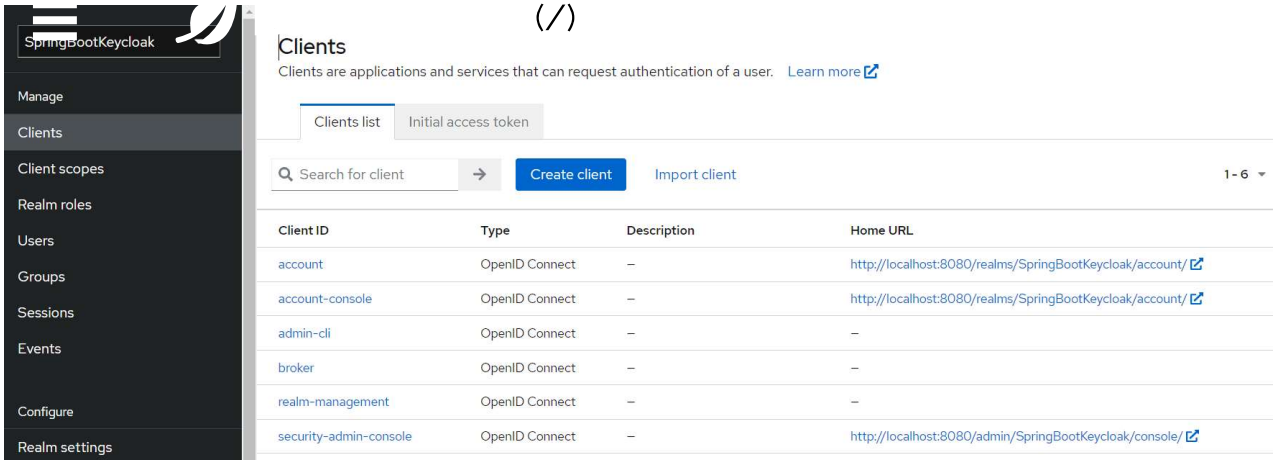
(/wp-content/uploads/2017/11/create-realm-name.png)

After clicking the *Create* button, a new realm will be created and we'll be redirected to it. All the operations in the next sections will be performed in this new *SpringBootKeycloak* realm.

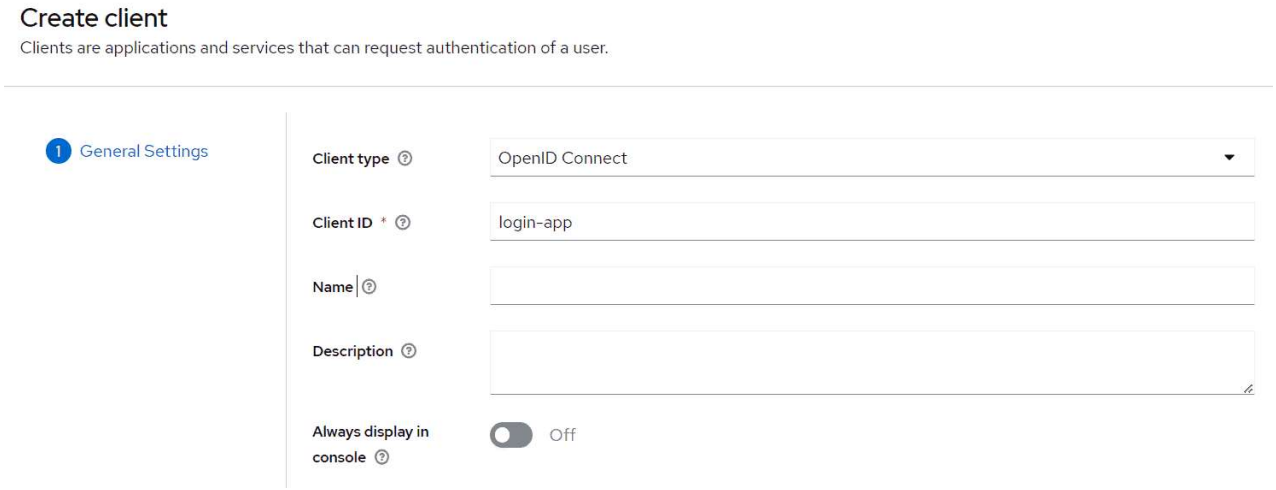
3.3. Creating a Client

Now we'll navigate to the Clients page. As we can see in the image below, **Keycloak comes with Clients that are already built-in**:

(/)



(/wp-content/uploads/2017/11/keycloak-clients-1.png)
We still need to add a new client to our application, so we'll click *Create*.
We'll call the new Client *login-app:*



(/wp-content/uploads/2017/11/create-client.png)
In the next screen, for the purpose of this tutorial, we'll leave all the defaults except **the *Valid Redirect URIs* field. This field should contain the application URL(s) that will use this client for authentication**:

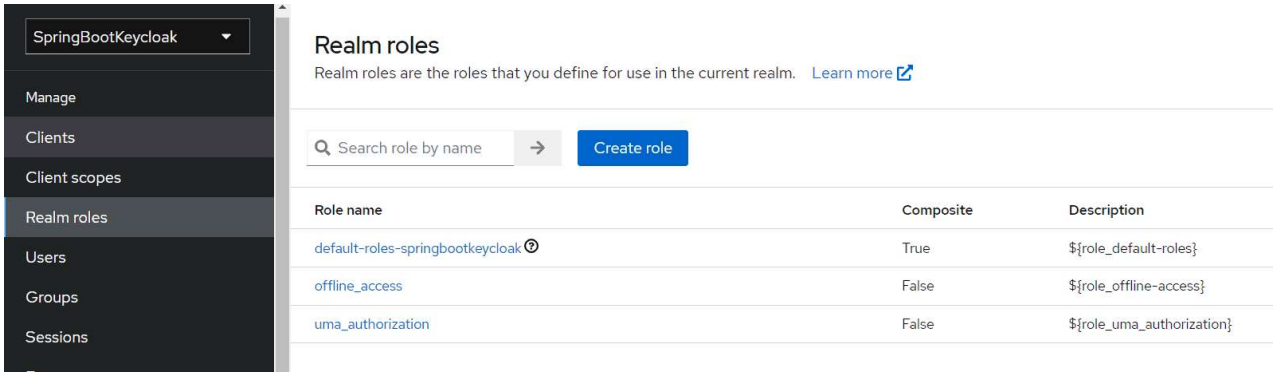
(/wp-content/uploads/2017/11/keycloak-redirect-uri.png)

Later on, we'll be creating a Spring Boot Application running at port 8081 that'll use this client. Hence we've used a redirect URL of *http://localhost:8081/** above.

3.4. Creating a Role and a User

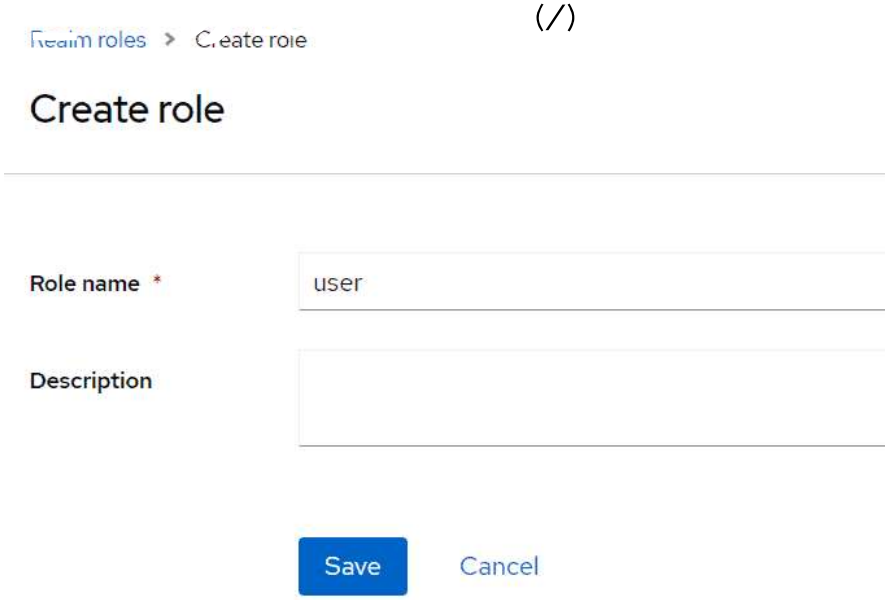
Keycloak uses Role-Based Access; therefore, each user must have a role.

To do that, we need to navigate to the *Realm Roles* page:



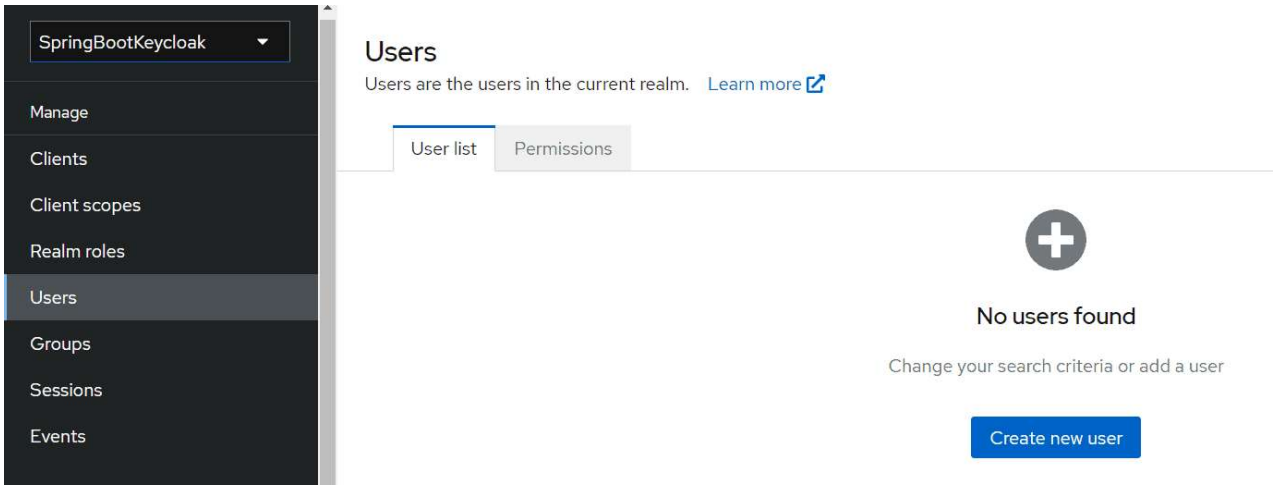
(/wp-content/uploads/2017/11/realm-roles.png)
Then we'll add the *user* role:

(/)



(/wp-content/uploads/2017/11/create-role.png)

Now we have a role that can be assigned to users, but as there are no users yet, **let's go to the *Users* page and add one:**

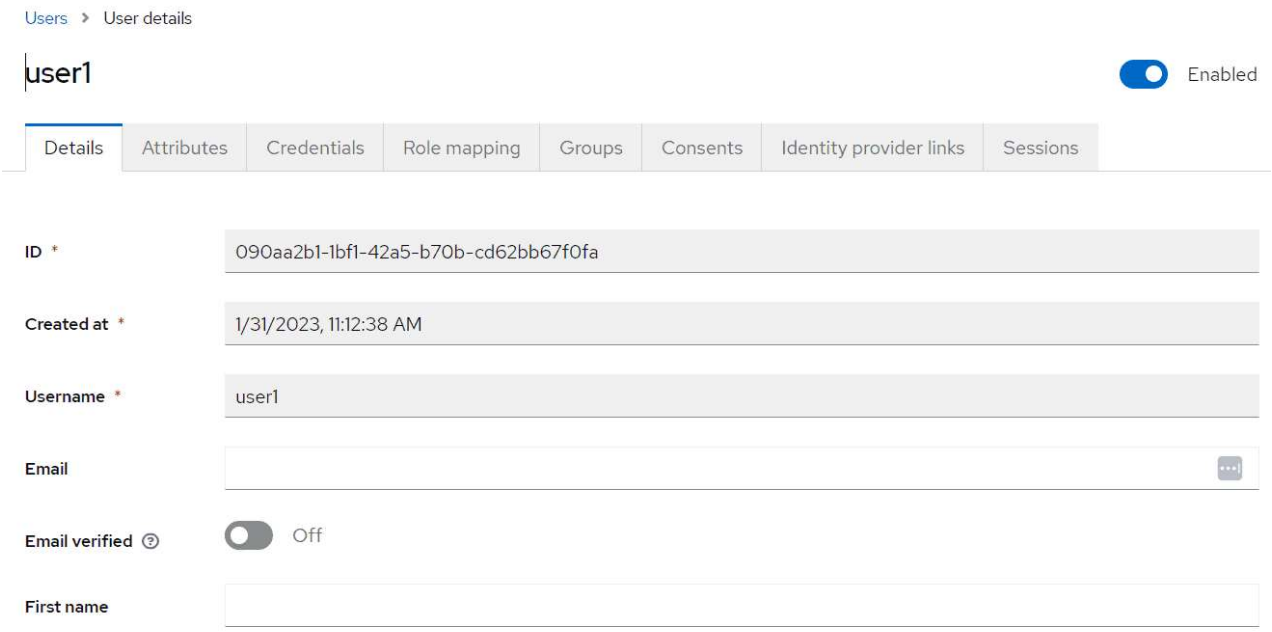


(/wp-content/uploads/2017/11/create-user.png)

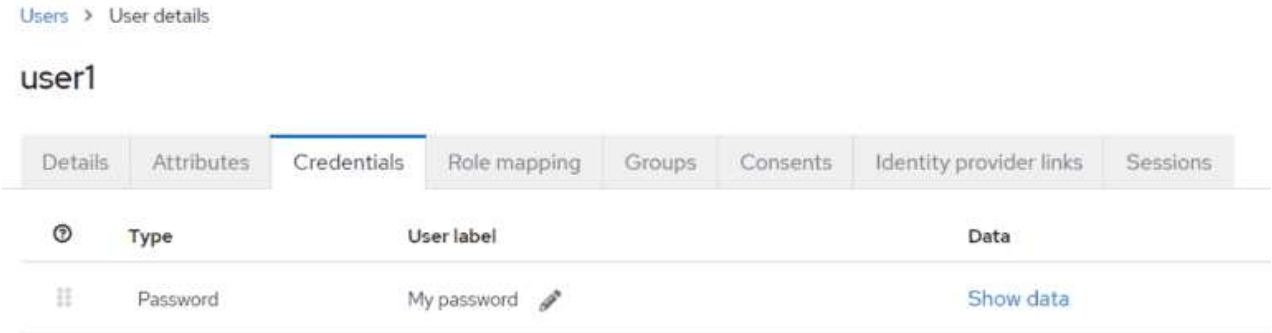
We'll add a user named *user1:*

(/)

(/wp-content/uploads/2017/11/create-user-2.png)
Once the user is created, a page with its details will be displayed:

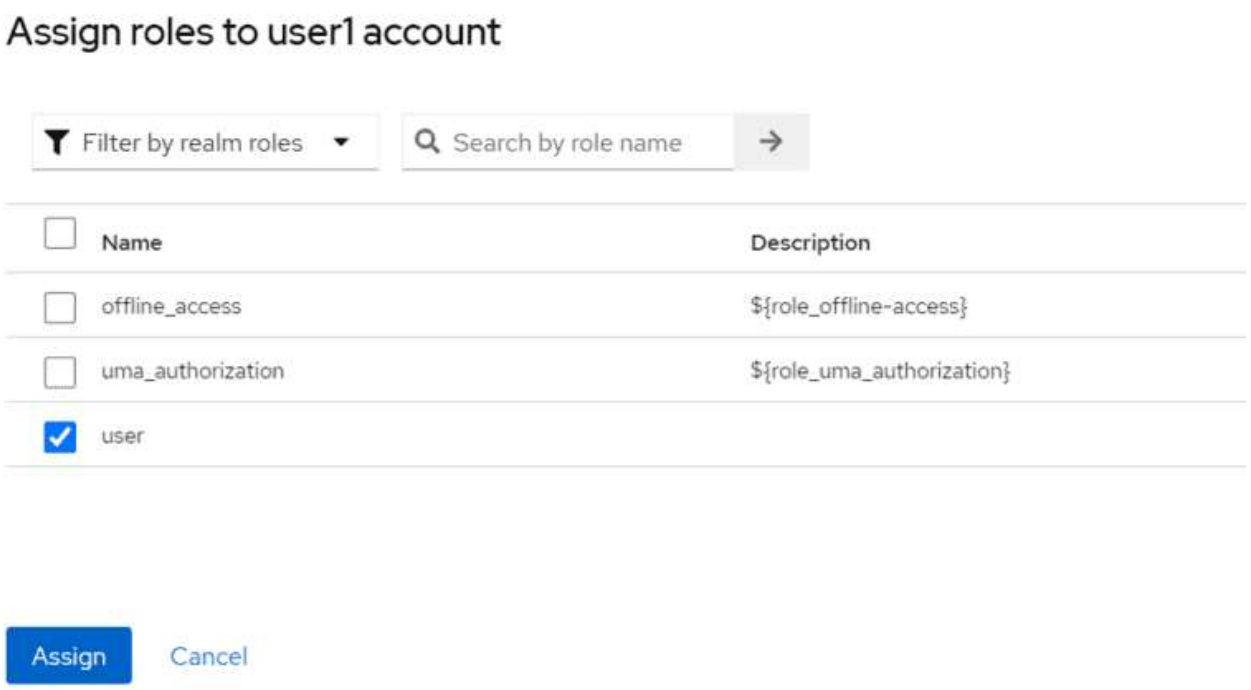


(/wp-content/uploads/2017/11/user-details.png)
We can now go to the *Credentials* tab. We'll be setting the initial password to *xsw2@WS:*



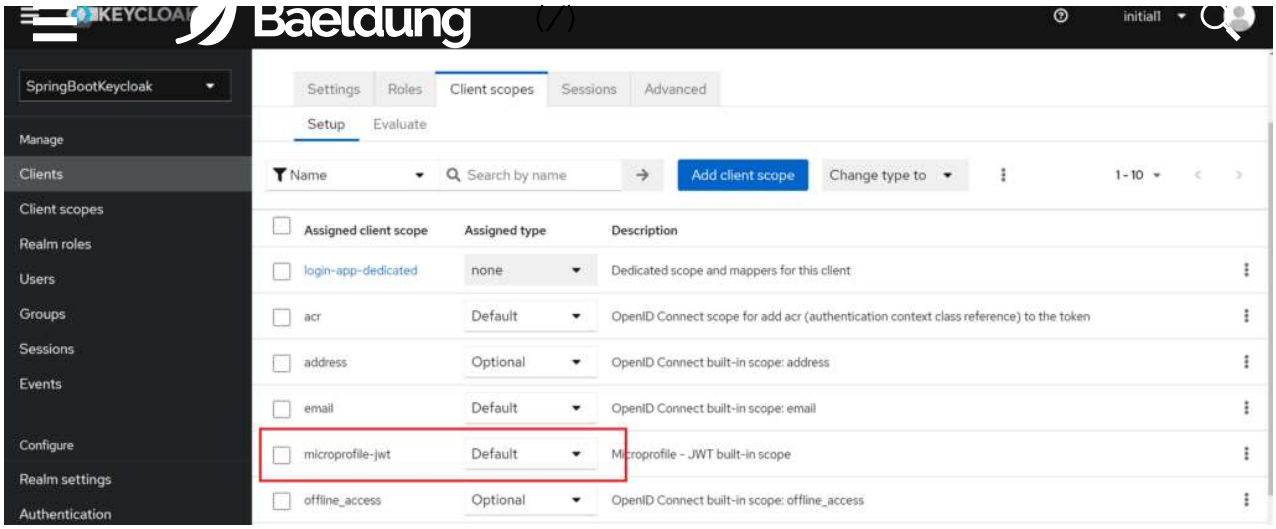
(/wp-content/uploads/2017/11/user-pass.png)

Finally, we'll navigate to the *Role Mappings* tab. We'll be assigning the *user* role to our *user1*:



(/wp-content/uploads/2017/11/assign-role.png)

Lastly, we need to set the ***Client Scopes*** properly so that KeyCloak passes all the roles for the authenticating user to the token. So we need to navigate to the *Client Scopes* page and then set *microprofile-jwt* to *"default"*, as shown in the picture below.



(/wp-content/uploads/2017/11/KeyCloak-Client-Scope.png)

4. Generating Access Tokens With Keycloak's API

Keycloak provides a REST API for generating and refreshing access tokens. We can easily use this API to create our own login page.

First, we need to acquire an access token from Keycloak by sending a POST request to this URL:

```
http://localhost:8080/realms/SpringBootKeycloak/protocol/openid-connect/token
```

The request should have this body in a *x-www-form-urlencoded* format:

```
client_id:<your_client_id>
username:<your_username>
password:<your_password>
grant_type:password
```

In response, we'll get an *access_token* and a *refresh_token*.

The access token should be used in every request to a Keycloak-protected resource by simply placing it in the *Authorization* header:

```
headers: {
    'Authorization': 'Bearer' + access_token
}
```

Once the access token has expired, we can refresh it by sending a POST request to the same URL as above, but containing the refresh token instead of username and password:

```
{
    'client_id': 'your_client_id',
    'refresh_token': refresh_token_from_previous_request,
    'grant_type': 'refresh_token'
}
```

Keycloak will respond to this with a new *access_token* and *refresh_token*.

5. Creating and Configuring a Spring Boot Application

In this section, we'll create a Spring Boot application and configure it as an OAuth Client to interact with the Keycloak server.

5.1. Dependencies

We use the Spring Security OAuth2.0 Client to connect to the Keycloak server.

Let's start by declaring *spring-boot-starter-oauth2-client* (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-oauth2-client) dependency in a Spring Boot application in the *pom.xml*:

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

Also, as we need to use Spring Security with Spring Boot, we must add this dependency (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security):

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

In order to delegate the identification control to a Keycloak server, we'll use the spring-boot-starter-oauth2-resource-server (https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-oauth2-resource-server) library. It will allow us to validate a JWT token with the Keycloak server. Hence, let's add it to our pom:

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

Now, the Spring Boot application can interact with Keycloak.

5.2. Keycloak Configuration

We consider the Keycloak client as an OAuth Client. So, we need to configure the Spring Boot application to use the OAuth Client.

The *ClientRegistration* class holds all of the basic information about the client. Spring auto-configuration looks for properties with the schema *spring.security.oauth2.client.registration.[registrationId]* and registers a client with OAuth 2.0 or OpenID Connect (OIDC) (/spring-security-openid-connect#introduction).

Let's configure the client registration configuration:

```
spring.security.oauth2.client.registration.keycloak.client-id=login-app
spring.security.oauth2.client.registration.keycloak.authorization-grant-type=authorization_code
spring.security.oauth2.client.registration.keycloak.scope=openid
```

The value we specify in *client-id* matches the client we named in the admin console.

The Spring Boot application needs to interact with an OAuth 2.0 or OIDC provider to handle the actual request logic for different grant types. So, we need to configure the OIDC provider. It can be auto-configured based on property values with the schema *spring.security.oauth2.client.provider.[provider name]*.

Let's configure the OIDC provider configuration:

```
spring.security.oauth2.client.provider.keycloak.issuer-
uri=http://localhost:8080/realms/SpringBootKeycloak
spring.security.oauth2.client.provider.keycloak.user-name-
attribute=preferred_username
```

As we can recall, we started Keycloak on port *8080*, hence the path specified in *issuer-uri*. This property identifies the base URI for the authorization server. We enter the realm name we created in the Keycloak admin console. Additionally, we can define *user-name-attribute* as *preferred_username* so as to populate our controller's *Principal* with a proper user.

Finally, let's add the configuration needed for validating JWT token against our Keycloak server:

```
spring.security.oauth2.resourceserver.jwt.issuer-
uri=http://localhost:8080/realms/SpringBootKeycloak
```

5.3. Configuration Class

We configure *HttpSecurity* by creating a *SecurityFilterChain bean*. Also, we need to enable OAuth2 login using *http.oauth2Login()*.

Let's go through the steps required for creating the security configuration. We'll configure the security settings for the application with the following objectives:

> Grant access exclusively to individuals with the role *USER* for URLs commencing with *customers/**. Allow authenticated users unrestricted access to all other URLs, excluding those under *customers/** if they lack the *USER* role.

The following code implements the mentioned security requirements:

(/)

```
@Bean
public SecurityFilterChain resourceServerFilterChain(HttpSecurity http)
throws Exception {
    http.authorizeHttpRequests(auth -> auth
       .requestMatchers(new AntPathRequestMatcher("/customers*",
HttpMethod.OPTIONS.name()))
       .permitAll()
       .requestMatchers(new AntPathRequestMatcher("/customers*"))
       .hasRole("user")
       .requestMatchers(new AntPathRequestMatcher("/"))
       .permitAll()
       .anyRequest()
       .authenticated());
    http.oauth2ResourceServer((oauth2) -> oauth2
       .jwt(Customizer.withDefaults()));
    http.oauth2Login(Customizer.withDefaults())
       .logout(logout ->
logout.addLogoutHandler(keycloakLogoutHandler).logoutSuccessUrl("/"));
    return http.build();
}
```

In the code above, **the *oauth2Login()* method adds *OAuth2LoginAuthenticationFilter* (https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/client/web/OAuth2LoginAuthenticationFilter.html) to the filter chain. The filter intercepts requests and applies the needed logic for OAuth 2**

authentication. The *oauth2ResourceServer* method validates the bound of JWT token against our Keycloak server. It also enforces the constraints discussed earlier.

Keycloak returns a token containing all pertinent information. To enable Spring Security to make decisions based on user-assigned roles, we must parse the token and extract the relevant details. However Spring Security generally adds *"ROLES_"* prefix to each role name, whereas Keycloak sends plain role names across. To fix this issue, we create a helper method which adds *"ROLE_"* prefix to each role retrieved from Keycloak.

```
@Bean
Collection generateAuthoritiesFromClaim(Collection roles) {
    return roles.stream().map(role -> new
SimpleGrantedAuthority("ROLE_" + role)).collect(
            Collectors.toList());
}
```

Now we can proceed to parse tokens. First, we have to check if the token is the instance of OidcUserAuthority or OAuth2UserAuthority. Since Keycloak tokens can be of either type, we need to implement a parsing logic. The code below checks for the type of token and decides on parsing mechanism.

```
boolean isOidc = authority instanceof OidcUserAuthority;
    if (isOidc) {
        /// Parsing code here
    }
```

By default, the token is an instance of OidcUserAuthority.

If the token is an instance OidcUserAuthority, it can be configured to contain the roles under both Groups or realm access. Hence we have to check both to extract the roles as shown in the code below,

```
if (userInfo.hasClaim(REALM_ACCESS_CLAIM)) {
    var realmAccess = userInfo.getClaimAsMap(REALM_ACCESS_CLAIM);
    var roles = (Collection) realmAccess.get(ROLES_CLAIM);
    mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
} else if (userInfo.hasClaim(GROUPS)) {
    Collection roles = (Collection) userInfo.getClaim(GROUPS);
    mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
}
```

If however the token is an instance OAuth2UserAuthority, we need to parse it as follows:

```
var oauth2UserAuthority = (OAuth2UserAuthority) authority;
Map<String, Object> userAttributes =
oauth2UserAuthority.getAttributes();
if (userAttributes.containsKey(REALM_ACCESS_CLAIM)) {
    Map<String, Object> realmAccess = (Map<String, Object>)
userAttributes.get(REALM_ACCESS_CLAIM);
    Collection roles = (Collection) realmAccess.get(ROLES_CLAIM);
    mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
}
```

The complete code is given below for your reference,

```
@Configuration
@EnableWebSecurity
class SecurityConfig {

    private static final String GROUPS = "groups";
    private static final String REALM_ACCESS_CLAIM = "realm_access";
    private static final String ROLES_CLAIM = "roles";

    private final KeycloakLogoutHandler keycloakLogoutHandler;

    SecurityConfig(KeycloakLogoutHandler keycloakLogoutHandler) {
        this.keycloakLogoutHandler = keycloakLogoutHandler;
    }

    @Bean
    public SessionRegistry sessionRegistry() {
        return new SessionRegistryImpl();
    }

    @Bean
    protected SessionAuthenticationStrategy
sessionAuthenticationStrategy() {
        return new
RegisterSessionAuthenticationStrategy(sessionRegistry());
    }

    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }

    @Bean
    public SecurityFilterChain resourceServerFilterChain(HttpSecurity
http) throws Exception {
        http.authorizeHttpRequests(auth -> auth
            .requestMatchers(new AntPathRequestMatcher("/customers*"))
            .hasRole("user")
            .requestMatchers(new AntPathRequestMatcher("/"))
            .permitAll()
            .anyRequest()
            .authenticated());
        http.oauth2ResourceServer((oauth2) -> oauth2
            .jwt(Customizer.withDefaults()));
        http.oauth2Login(Customizer.withDefaults())
            .logout(logout ->
logout.addLogoutHandler(keycloakLogoutHandler).logoutSuccessUrl("/"));
        return http.build();
    }

    @Bean
    public GrantedAuthoritiesMapper userAuthoritiesMapperForKeycloak()
```

```
{
        return authorities -> {
            Set<GrantedAuthority> mappedAuthorities = new HashSet<>();
            var authority = authorities.iterator().next();
            boolean isOidc = authority instanceof OidcUserAuthority;

            if (isOidc) {
                var oidcUserAuthority = (OidcUserAuthority) authority;
                var userInfo = oidcUserAuthority.getUserInfo();

                // Tokens can be configured to return roles under
                // Groups or REALM ACCESS hence have to check both
                if (userInfo.hasClaim(REALM_ACCESS_CLAIM)) {
                    var realmAccess =
userInfo.getClaimAsMap(REALM_ACCESS_CLAIM);
                    var roles = (Collection<String>)
realmAccess.get(ROLES_CLAIM);

mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
                } else if (userInfo.hasClaim(GROUPS)) {
                    Collection<String> roles = (Collection<String>)
userInfo.getClaim(
                        GROUPS);

mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
                }
            } else {
                var oauth2UserAuthority = (OAuth2UserAuthority)
authority;
                Map<String, Object> userAttributes =
oauth2UserAuthority.getAttributes();

                if (userAttributes.containsKey(REALM_ACCESS_CLAIM)) {
                    Map<String, Object> realmAccess = (Map<String,
Object>) userAttributes.get(
                        REALM_ACCESS_CLAIM);
                    Collection<String> roles = (Collection<String>)
realmAccess.get(ROLES_CLAIM);

mappedAuthorities.addAll(generateAuthoritiesFromClaim(roles));
                }
            }
            return mappedAuthorities;
        };
    }

    Collection<GrantedAuthority>
generateAuthoritiesFromClaim(Collection<String> roles) {
        return roles.stream().map(role -> new
SimpleGrantedAuthority("ROLE_" + role)).collect(
            Collectors.toList());
```

```
    }
}
```

(/)

Finally, we need to handle logout from Keycloak. To do that, we add the *KeycloakLogoutHandler* class:

```
@Componen                    (/)
public class KeycloakLogoutHandler implements LogoutHandler {

    private static final Logger logger =
LoggerFactory.getLogger(KeycloakLogoutHandler.class);
    private final RestTemplate restTemplate;

    public KeycloakLogoutHandler(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @Override
    public void logout(HttpServletRequest request, HttpServletResponse
response,
      Authentication auth) {
        logoutFromKeycloak((OidcUser) auth.getPrincipal());
    }

    private void logoutFromKeycloak(OidcUser user) {
        String endSessionEndpoint = user.getIssuer() +
"/protocol/openid-connect/logout";
        UriComponentsBuilder builder = UriComponentsBuilder
          .fromUriString(endSessionEndpoint)
          .queryParam("id_token_hint",
user.getIdToken().getTokenValue());

        ResponseEntity<String> logoutResponse =
restTemplate.getForEntity(
        builder.toUriString(), String.class);
        if (logoutResponse.getStatusCode().is2xxSuccessful()) {
            logger.info("Successfulley logged out from Keycloak");
        } else {
            logger.error("Could not propagate logout to Keycloak");
        }
    }
}
```

The *KeycloakLogoutHandler* class implements *LogoutHandler* class and sends a logout request to the Keycloak.

Now, after we authenticate, we'll be able to access the internal customers page.

5.4. Thymeleaf Web Pages

We're using Thymeleaf for our web pages.

We've got three pages:

- *external.html* – an externally facing web page for the public
- *customers.html* – an internally facing page that will have its access restricted to only authenticated users with the role *user*
- *layout.html* – a simple layout, consisting of two fragments, that are used for both the externally facing page and the internally facing page

The code for the Thymeleaf templates is available on Github (https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-keycloak/src/main/resources/templates).

5.5. Controller

The web controller maps the internal and external URLs to the appropriate Thymeleaf templates:

```
@GetMapping(path = "/")
public String index() {
    return "external";
}

@GetMapping(path = "/customers")
public String customers(Principal principal, Model model) {
    addCustomers();
    model.addAttribute("customers", customerDAO.findAll());
    model.addAttribute("username", principal.getName());
    return "customers";
}
```

For the path */customers*, we're retrieving all customers from a repository and adding the result as an attribute to the *Model*. Later on, we iterate through the results in Thymeleaf.

To be able to display a username, we're injecting the *Principal* as well.

(/)

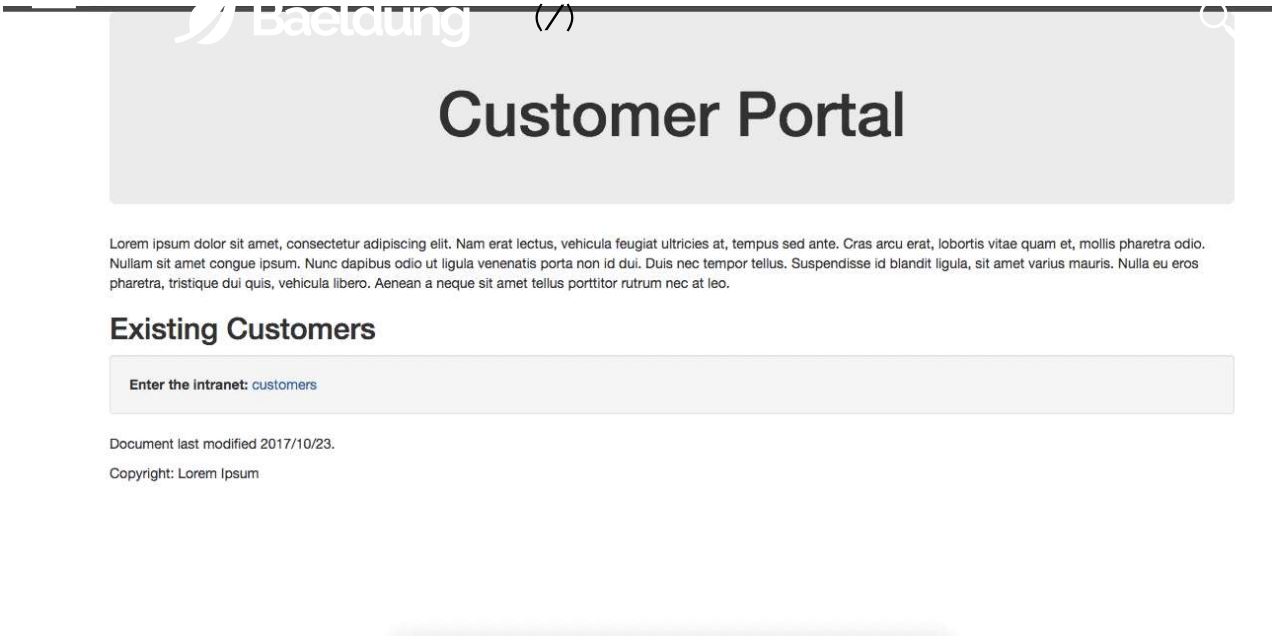
We should note that we're using customers here just as raw data to display, and nothing more.

6. Demonstration

Now we're ready to test our application. To run a Spring Boot application, we can start it easily through an IDE, like Spring Tool Suite (STS), or run this command in the terminal:

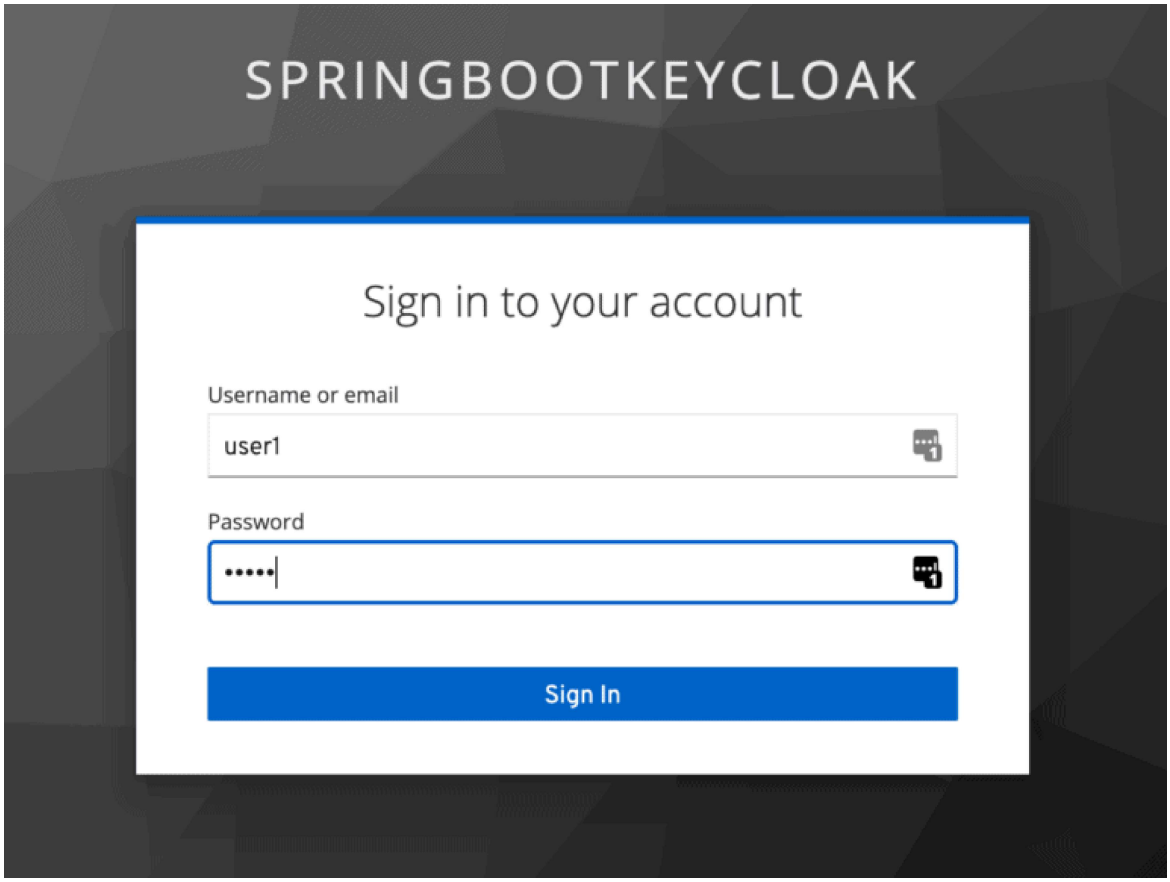
```
mvn clean spring-boot:run
```

On visiting *http://localhost:8081 (http://localhost:8081)* we see:



(/wp-content/uploads/2017/11/externalFacingKeycloakPage.png)
Now we click *customers* to enter the intranet, which is the location of sensitive information.

Note that we've been redirected to authenticate through Keycloak to see if we're authorized to view this content:



(/wp-content/uploads/2017/11/6_keycloak_userlogin.png)
Once we log in as *user1*, Keycloak will verify our authorization that we have the *user* role, and we'll be redirected to the restricted *customers* page:

(/wp-content/uploads/2017/11/customers-page.png)
Now we've finished the setup of connecting Spring Boot with Keycloak and demonstrating how it works.

As we can see, **Spring Boot seamlessly handled the entire process of calling the Keycloak Authorization Server**. We did not have to call the Keycloak API to generate the Access Token ourselves, or even send the Authorization header explicitly in our request for protected resources.

7. Conclusion

In this article, we configured a Keycloak server and used it with a Spring Boot Application.

We also learned how to set up Spring Security and use it in conjunction with Keycloak. A working version of the code shown in this article is available over on Github (https://github.com/eugenp/tutorials/tree/master/spring-boot-modules/spring-boot-keycloak).

(/)

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
ALL BULK COURSES (/COURSES/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/COURSES/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)
JAVA ARRAY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/JAVA-ARRAY)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)