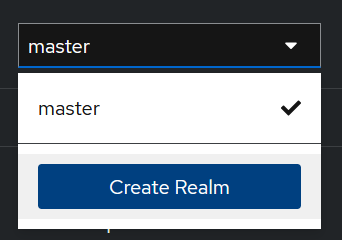
# Keycloak Server Configuration

Core concepts and terms description: <https://www.keycloak.org/docs/latest/server_admin/index.html#core-concepts-and-terms>

1. Creating a realm

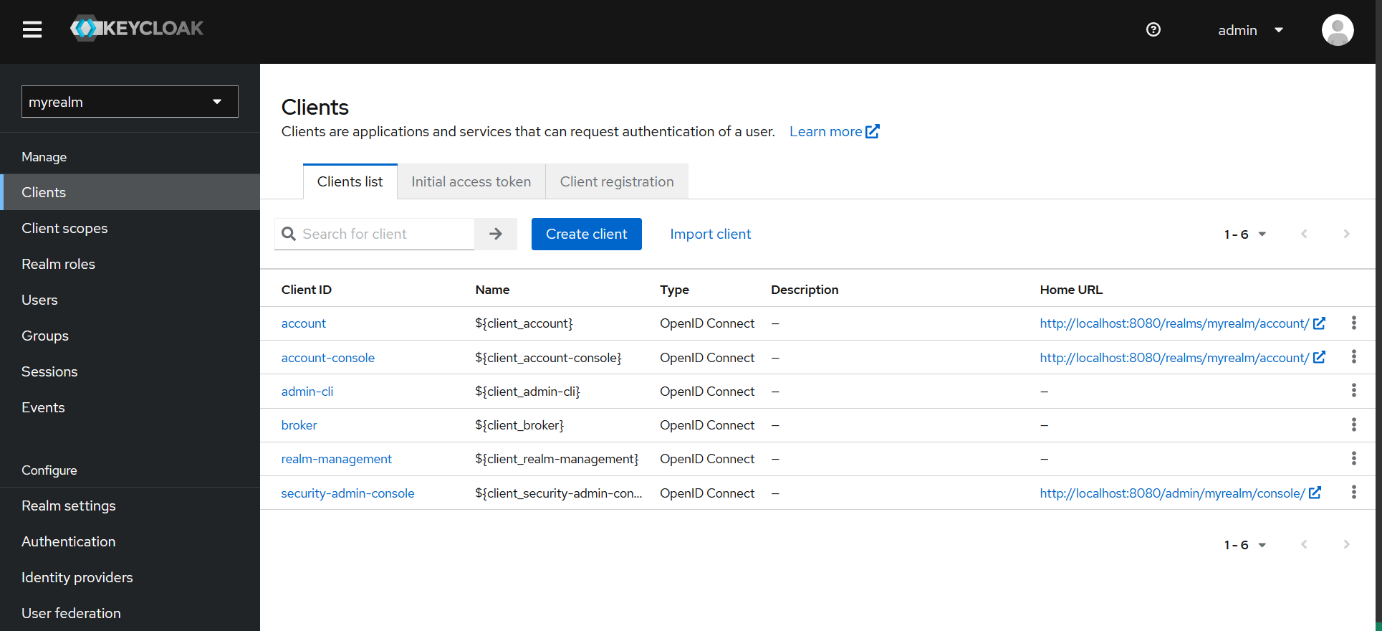
A realm in Keycloak is essentially a **tenant**. Realms are isolated from each other. The master realm can be seen as the “admin” tenant, used to manage Keycloak. When you are logged in to the admin console within Keycloak, navigate to the top left where it says “master”. Click the **create realm** button.



You can setup a realm using a configuration file (.json), or create a default one by providing a name.

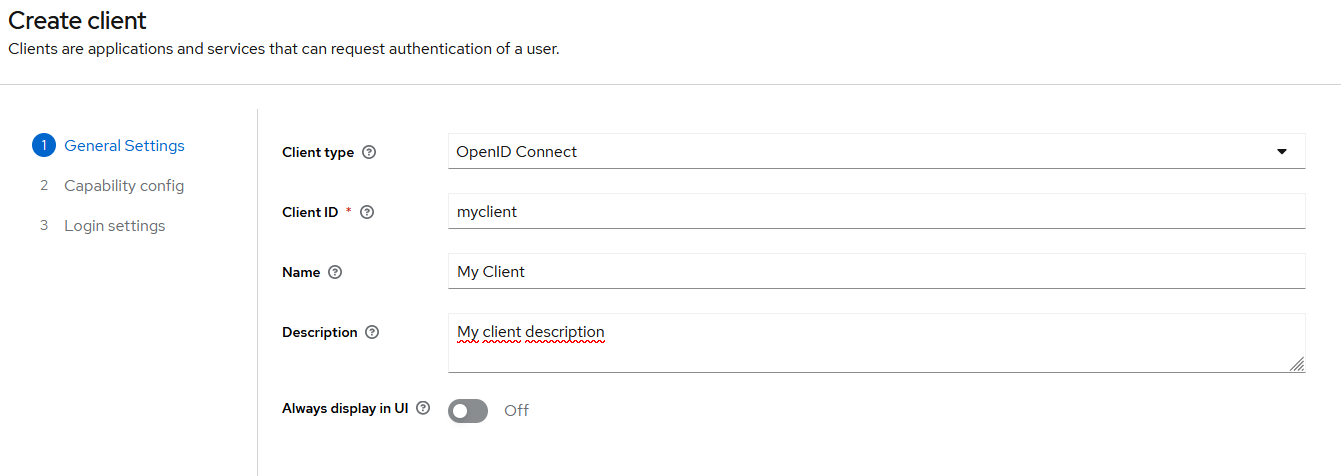
1. Creating a client

Clients are applications that can request authentication of a user. To create a client, click the **clients** tab on the sidebar. Then click the **create client** button.

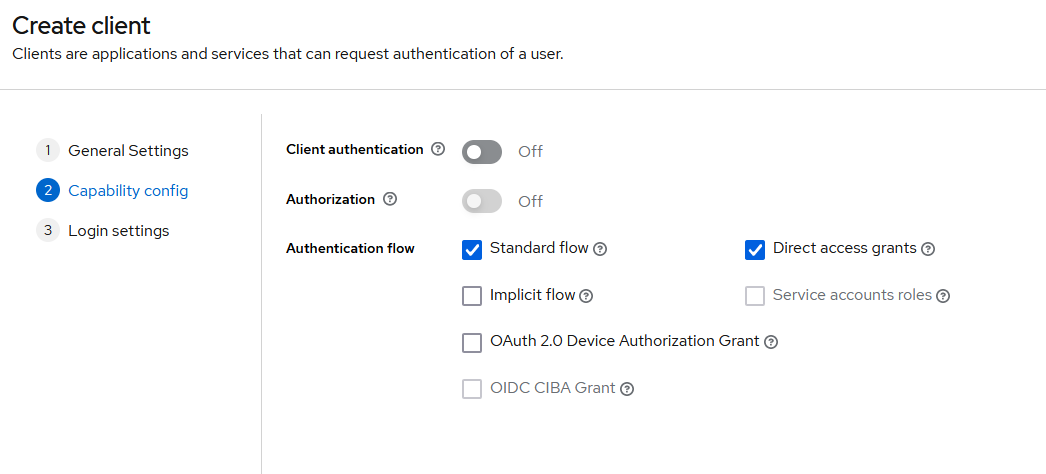


The following settings need to be configured:

* Client type: You can choose between OpenID Connect and SAML. Generally you want to use OpenID Connect for this.
* Client ID: This ID will be referenced in the URI and tokens.
* Name: Friendly display name.
* Description: A description of what the client is.

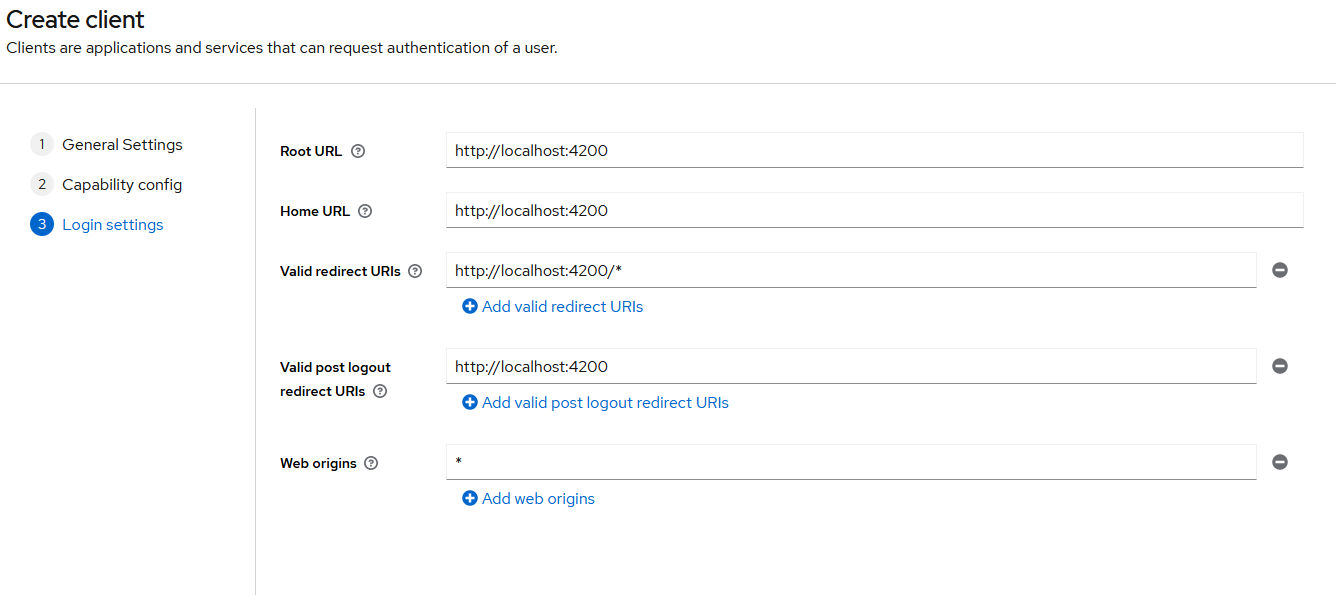


For now, we use the standard configuration for a client. We wil be using the standard authorization flow that uses an authorization code that can later be exchanged for an access token.



Last we have to configure the login settings.

* Root URL: The base URL of the client application
* Home URL: The default URL when the authorization server needs to redirect back to the client.
* Valid redirect URIs: Allows redirection to preconfigured URI's. In this case a wildcard is used, which means it can be redirected to anything as long as its localhost:4200
* Valid post logout redirect URIs: URI to redirect to when a user logs out.
* Web origins: Used for CORS. In this case we permit all origins using the \* wildcard.

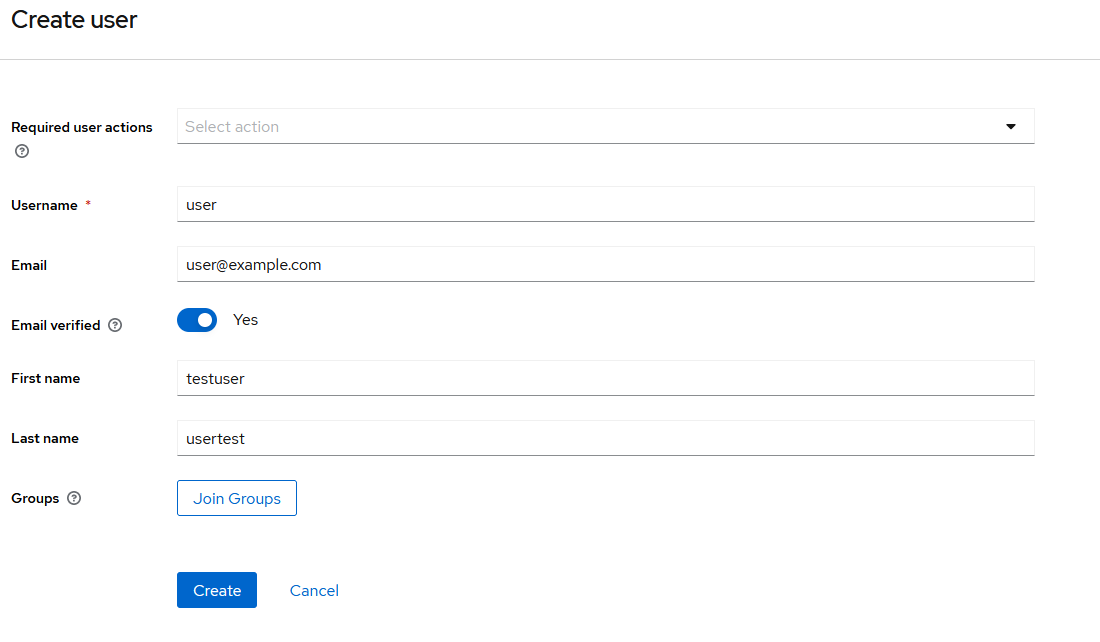
The client should now be visible under the **clients** tab:



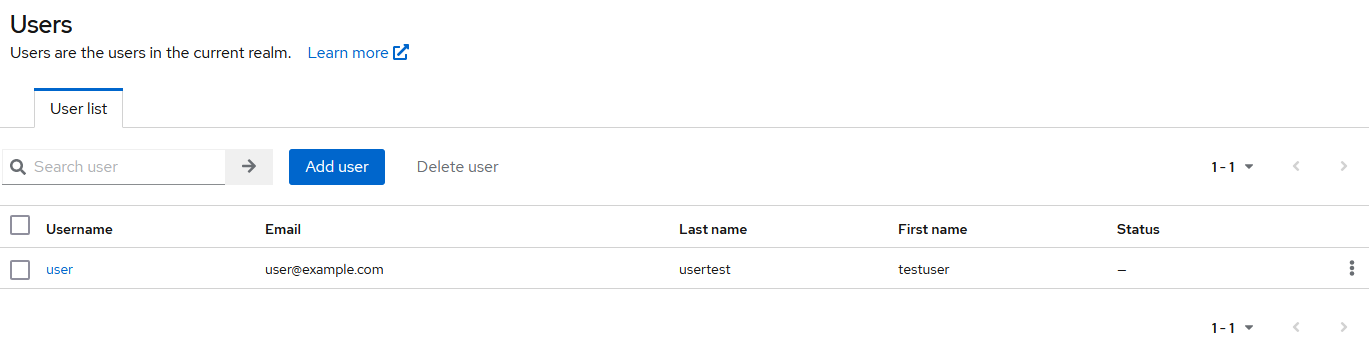
1. Adding a user

To add a user manually in the currently selected realm, click the **users** tab. Then click the **create new user** button. The following can be configured:

* Required user actions: These are actions that are required when a user logs in. This way, you could force a user to update their profile or change their password.
* Username: The name of the user.
* Email: The email of the user.
* Email verified: Sets if the email has been verified.
* First name: Real first name of the user.
* Last name: Real last name of the user.
* Groups: A user can be part of a group, which can have certain permissions or roles. These will be automatically inherited by the user.

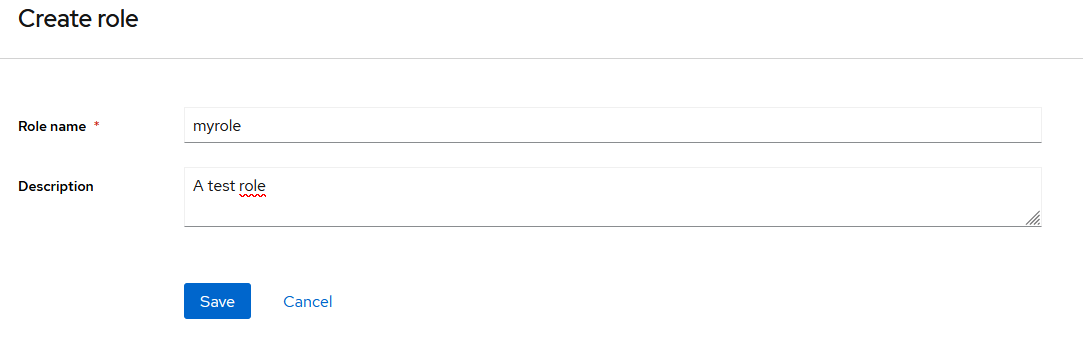


Since we did not specify any user actions, we have to change the password of the user manually. When you click the **create** button, you will be directed to the user details. Click the **credentials** tab, and then click the **set password** button. In this example the user credentials are user / user. The user should now be visible:



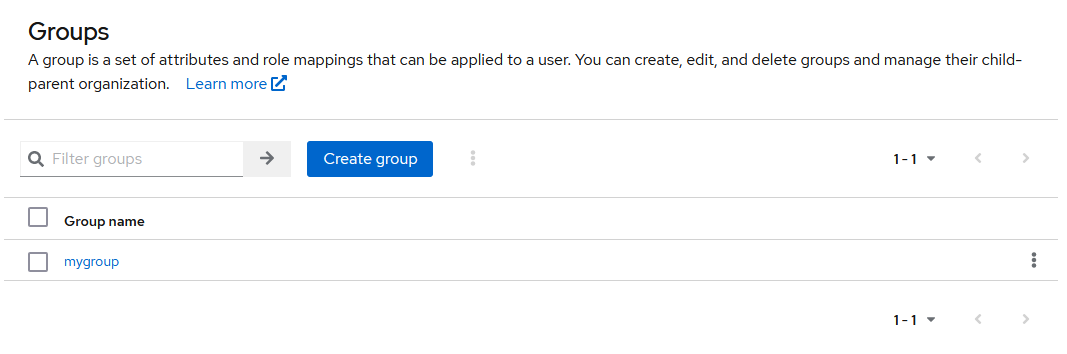
1. Adding roles

Click the **realm roles** tab, then click the **create role** button. Give the role a name and description and save.

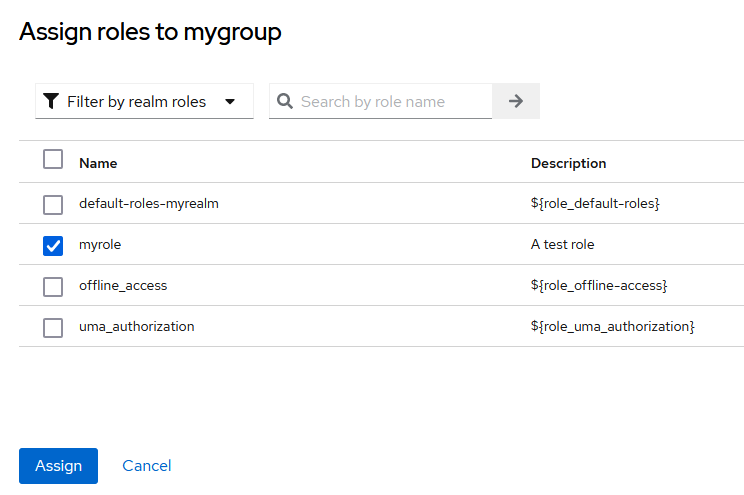
It is now possible to add a user to this role directly, however it is better to create a group first to have more control.

1. Adding groups

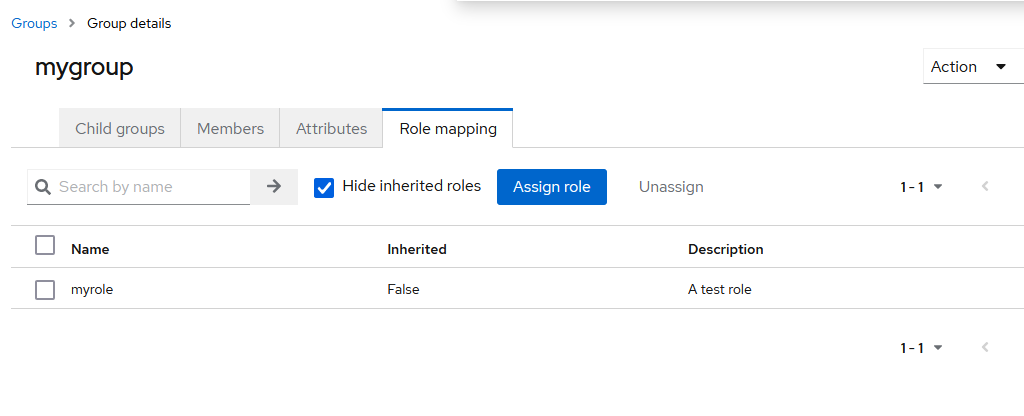
Click the **groups** tab, then click the **create group** button. Give the group a name and click **save**. A group is now created:



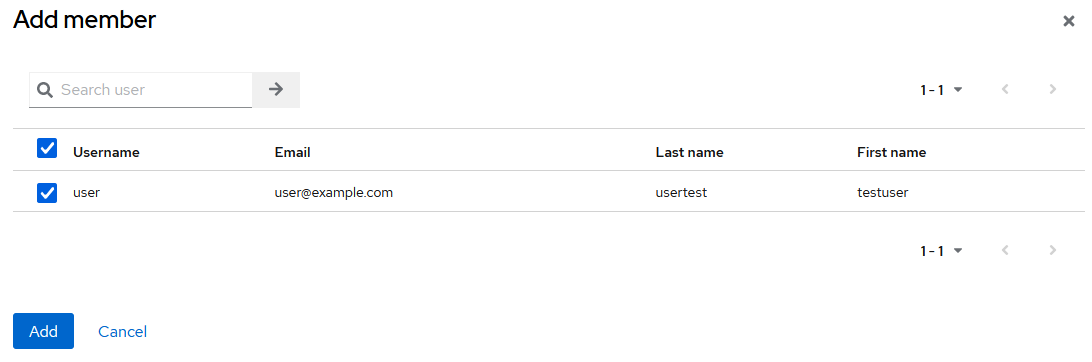
To add a role to a group, click the **role mapping** tab and click the **assign role** button. If you have created a role before, you can select this role using the **checkbox**. Then click the **assign** button.



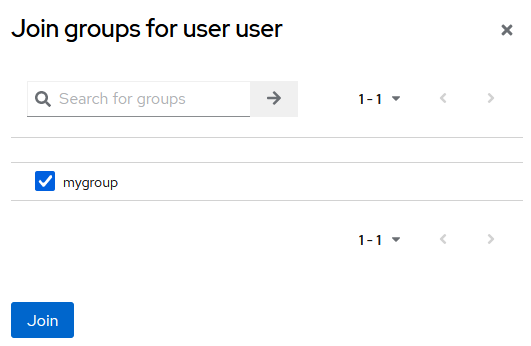
Your group should now have a role assigned to it. Every member of this group will now automatically inherit the role that is added.



Adding a user to a group can be done in two ways. You can use the groups or users tab. If you are using the **groups** tab, navigate to the **members** tab and select your user as follows:



If you are using the **users** tab, select a **user** and go to the **groups** tab. Then click the **join group** button and select the group as follows:

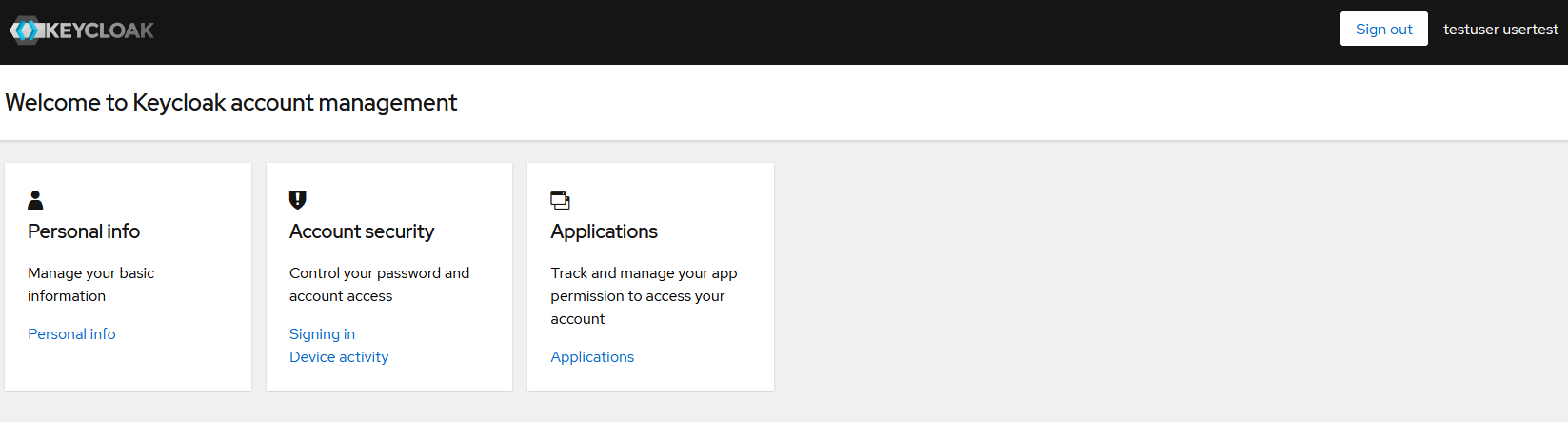


# Keycloak Account Console

A user can login to the account console with their own user credentials. The URL for this is:

<http://localhost:8080/realms/myrealm/account/>

When you login with the user account, this looks as follows:



# Keycloak URLs

The URL to the OpenID configuration is: <http://localhost:8080/realms/myrealm/.well-known/openid-configuration>

This endpoint exposes URLs, that will be used by other applications like your frontend client. Some of the important ones are:

* Authorization\_endpoint: This endpoint is used to initiate the OAuth 2.0 authorization flow. Clients can direct users to this endpoint to authenticate users so they can access protected resources using an access token.
* Token\_endpoint: Used by clients to exchange authorization codes for refresh / access or ID tokens. This endpoint is also used to refresh tokens.
* End\_session\_endpoint: Used to log out users from the Keycloak server. Any session that the user has will be invalidated. The user will usually be redirected to a logout URL.
* Userinfo\_endpoint: Used to obtain user profile information after authentication and authorization has been completed. An access token needs to be used to get this information.
* Jwks\_uri: Used by clients to obtain publis keys. These keys are used to verify JSON Web tokens issued by the OpenID Connect provider. When a user logs in and authorizes with a client, it receives an ID token containing information about that user.

It also displays the different kind of “grant types” that are supported. By default these are the following:

* Authorization\_code: Used by web apps that allow the client to get an access token on behalf of the user. The authorization server returns an authorization code when a user logs in, which will be used to exchange for an access token.
* Implicit: Also used by web apps, but does not involve an authorization code. The access token is returned as part of the redirection URL. It is considered less secure than the authorization code grant type, because the access token is exposed to the client.
* Password:
* Client\_credentials: Allows users to exchange credentials for an access token. This is typically used in mobile or desktop applications, where the client is owned by the same organization.
* Refresh\_token: Used by clients to obtain a new access token after the initial one has expired.

# Keycloak Angular Configuration

Our demo application demonstrates the usage of the keycloak-angular library with the keycloak docker container that is running. The following pages will be created:

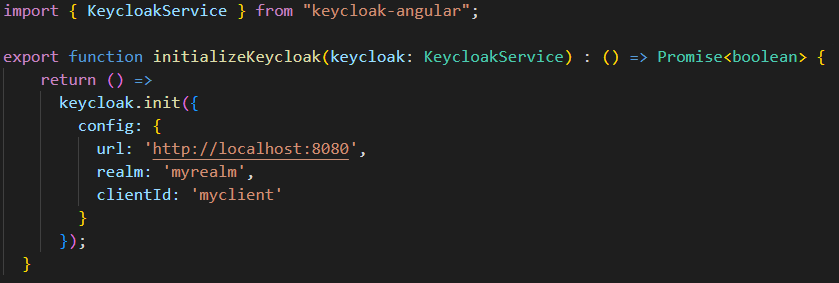
* A home landing page, visible to anyone
* A protected profile page, only visible to registered users
* A protected information page, only visible to users in a certain role
* Eventually a protected page that makes calls to a protected resource server (Java API)

The API reference for keycloak-js: <https://www.keycloak.org/docs/latest/securing_apps/#javascript-adapter-reference>

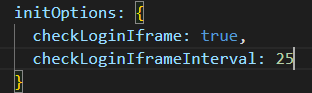
The API reference for keycloak-angular: <https://www.npmjs.com/package/keycloak-angular?activeTab=code>

1. Initializing Keycloak

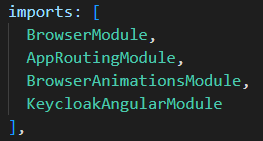
Create a folder in the app directory of your Angular application. Create a new typescript file such as **app.init.ts**. Add an exported function called initializeKeycloak. The default boilerplate for this function is as follows:

Here the keycloak service will be initialized with the settings of the keycloak docker instance that is configured. Note that the URL, realm and clientId are all matching.

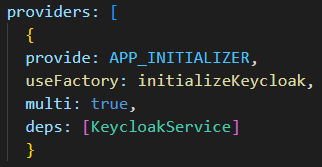
Optionally, we can pass an initOptions object to make sure that the token can be automatically refreshed.



Next we need to import the Keycloak module inside the app.module.ts file. Add the KeycloakAngularModule to the imports array:



Next we need to add the provider which will initialize keycloak by calling the initializeKeycloak function:



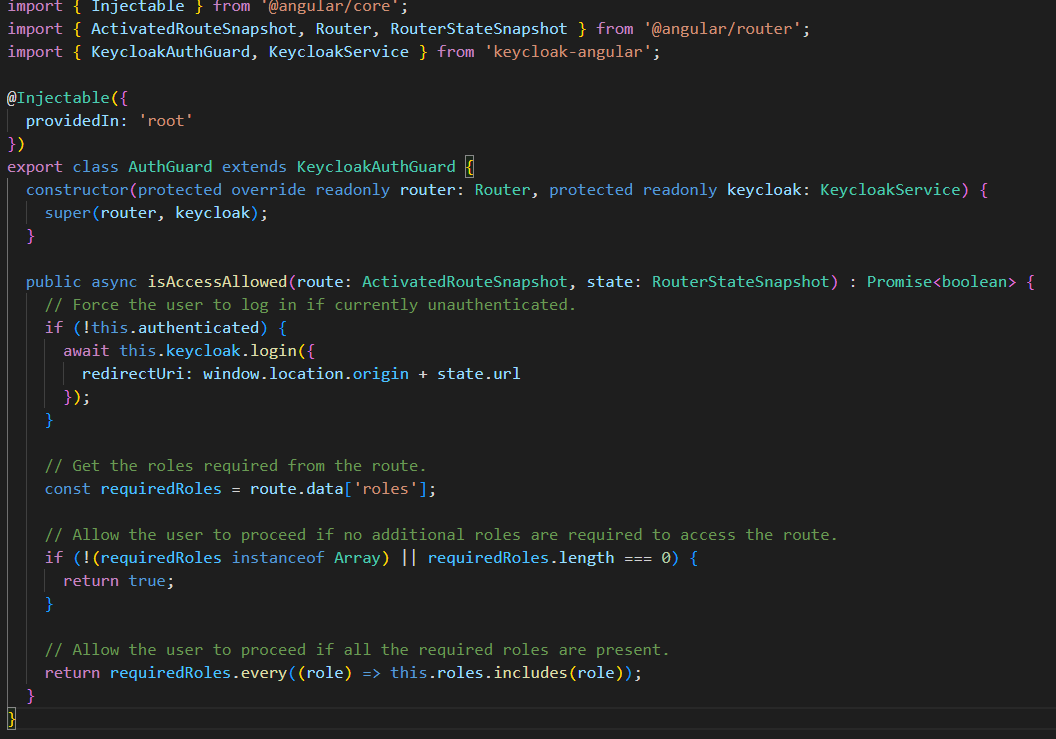
The following imports are needed:





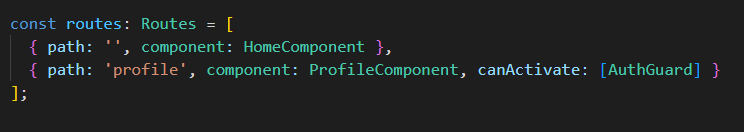
1. Adding an AuthGuard

With an AuthGuard, we can run some code before a component is getting loaded. In this case, we will check if the user is logged in when the profile component is loaded that displays the profile page. To use the default AuthGuard with Keycloak, you will have to extend from the provided KeycloakAuthGuard class. This class has one method “isAccessAllowed” that returns a boolean wether or not a user is allowed to proceed to a route.

As you can see, the KeycloakService is injected into the constructor, which will be used to redirect the user to the login page when he is not authenticated yet.

1. Adding routing

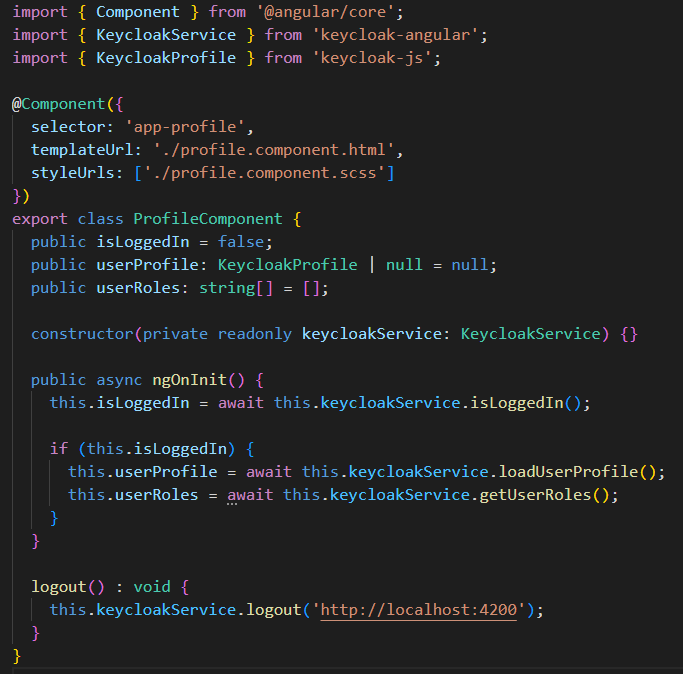
In order to use the authentication guard on a route, you need to define the canActivate property on that route with a reference to the AuthGuard class. In this demo, the following routes are present:

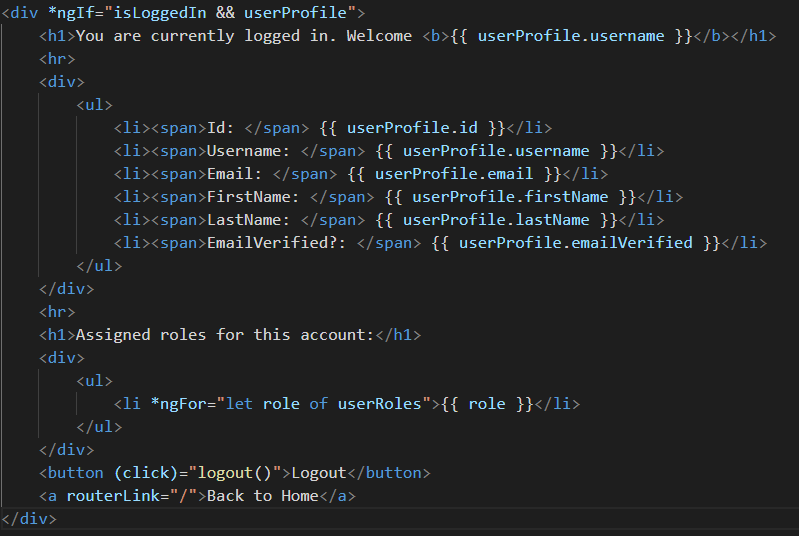


* The ‘ ‘ path is the default URL, and redirects the user to the content of the HomeComponent. This component is accessible to any user.
* The profile path is similar to the home path, but has the AuthGuard attached to it that was created earlier. This guard controls access to the profile route based on a condition. This AuthGuard will return true or false depending if a user is authenticated with keycloak.

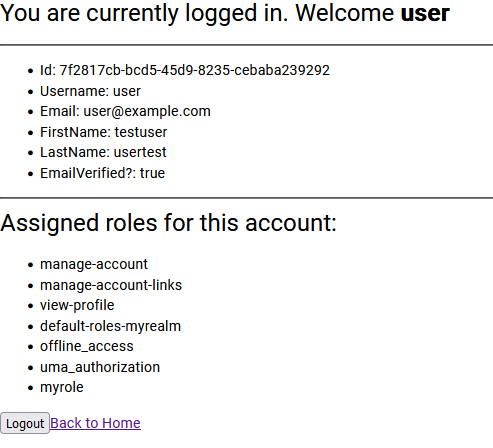
The ProfileComponent injects the KeycloakService and is initialized with data that will be used in that protected component. This component does the following:

* Gets if the user is currently logged in
* Gets the user profile
* Gets the user roles if the user is logged in
* Displays the roles in the html page
* Adds a logout button that can be clicked by the user

The HTML page looks like this:



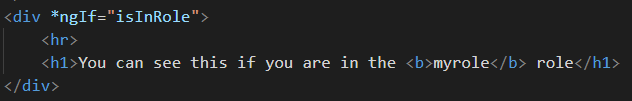
Which results in the following view after logging in:

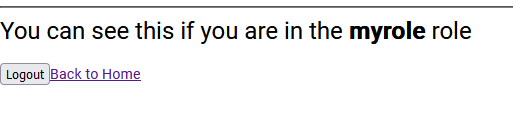


1. Adding role-based protected routes

Note: From the perspective of the user, it might be better to not display options that the user cant access. To do this, you can check if the user has a certain role:

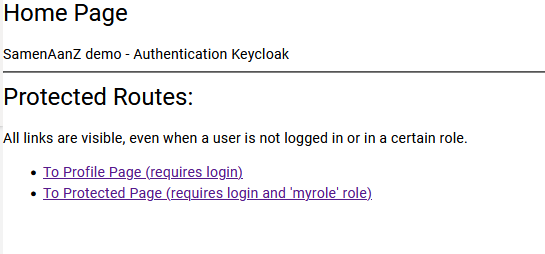






It is also possible to directly require a role when defining your routes. To do this, i've added a new component called ‘protected’. This is then added to the routes array:

The page currently looks like this:



When the user is not logged in and tries to access the protected role, the login prompt will show. When the user is in the ‘myrole’ role, he is able to access the protected page. If the user is not in the ‘myrole’ role, the link will not work.