

Kecloak in Docker #7 – How to authorize requests via Postman

 8th February 2022 /  little_pinecone /  Tools

Postman comes with a wide variety of OAuth 2.0 compliant configuration options that allow us to authorize requests against a Keycloak protected API. The current standard recommendation is to use Authorization Code Flow with PKCE extension.

Prerequisites

- Minimum [Postman version 7.23](#).
- The sample collection I'm using for this article is in my public Postman workspace and is part of my **keycloak-spring-boot** application. If you want to run the project locally (it requires Docker for running a Keycloak instance), visit the [project repository on GitHub](#) and follow the directions in the README.md file.
- If this is your first attempt to run Keycloak in Docker, I recommend reading the post [Keycloak in Docker # 1 – How to run Keycloak in a Docker container](#), as I explained the basic configuration there.
- The solution that I'll show works for the following sample client configuration:



Client ID ?	<input type="text" value="spring-boot-example-app"/>
Name ?	<input type="text"/>
Description ?	<input type="text" value="Example client for a Spring Boot REST API"/>
Enabled ?	<input checked="" type="checkbox"/> ON
Always Display in Console ?	<input type="checkbox"/> OFF
Consent Required ?	<input type="checkbox"/> OFF
Login Theme ?	<input type="text"/>
Client Protocol ?	<input type="text" value="openid-connect"/>
Access Type ?	<input type="text" value="confidential"/>
Standard Flow Enabled ?	<input checked="" type="checkbox"/> ON
Implicit Flow Enabled ?	<input type="checkbox"/> OFF
Direct Access Grants Enabled ?	<input type="checkbox"/> OFF
Service Accounts Enabled ?	<input type="checkbox"/> OFF
OAuth 2.0 Device Authorization Grant Enabled ?	<input type="checkbox"/> OFF
OIDC CIBA Grant Enabled ?	<input type="checkbox"/> OFF
Authorization Enabled ?	<input type="checkbox"/> OFF
Front Channel Logout ?	<input type="checkbox"/> OFF
Root URL ?	<input type="text"/>
* Valid Redirect URIs ?	<input type="text" value="http://localhost:8080/*"/>

Why we should use the PKCE Grant Type to authorize Postman requests in Keycloak



As we can read in the **Authorization Code** Grant Type documentation:

It is recommended that all clients use the **PKCE** extension with this flow as well to provide better security.

<https://oauth.net/2/grant-types/authorization-code/>

Furthermore, the PKCE description states that:

PKCE (RFC 7636) is an extension to the Authorization Code flow to prevent CSRF and authorization code injection attacks.

<https://oauth.net/2/pkce/>

In addition, my example Keycloak client is configured with the **confidential** Access Type. Consequently, it **provides client secrets** when exchanging temporary codes for tokens. This configuration may appear to provide sufficient protection. However, the documentation advocates using the PKCE extension even when we already apply client secrets:

PKCE is *not* a replacement for a client secret, and PKCE is recommended even if a client is using a client secret.

<https://oauth.net/2/pkce/>

Provide the data required for authorization in Keycloak to the Postman environment

In order to execute the authorization flow from Postman, I will have to enter some confidential and environment-sensitive details, e.g. **Client Secret**, **Auth URL** etc.:



<input type="checkbox"/>	Authorize using browser
Auth URL ⓘ	<input type="text" value="https://example.com/login/oauth/authorize"/>
Access Token URL ⓘ	<input type="text" value="https://example.com/login/oauth/access_token"/>
Client ID ⓘ	<input type="text" value="Client ID"/>
Client Secret ⓘ	<input type="text" value="Client Secret"/>

I will store the sensitive data as **environment variables**. First, click the **Environment quick look** button (the eye icon) and click **Edit** for the selected environment. Then, you can start adding variables. Below you'll see what information we need for the authorization flow and where to find the values.

Callback URL

Postman uses this url to extract the access token after a successful authorization in Keycloak. This is the **Redirect URI** value that I specified for my client in Keycloak:





* Valid Redirect URIs ⓘ

I'm going to create the `{{redirectUri}}` variable with the `http://localhost://8080/*` value.

Auth URL and Access Token URL

I'm going to provide the **authorization** and **token** urls from the **Keycloak OIDC URI endpoint list**. To get the values, visit the general realm settings and click the **OpenID Endpoint Configuration**:



Display name	<input type="text" value="Keep growing"/>
HTML Display name	<input type="text"/>
Frontend URL 	<input type="text"/>
Enabled 	<input checked="" type="checkbox"/>
User-Managed Access 	<input type="checkbox"/>
Endpoints 	<div><div>OpenID Endpoint Configuration</div><div>SAML 2.0 Identity Provider Metadata</div></div>
	<div><input type="button" value="Save"/> <input type="button" value="Cancel"/></div>

We're going to see the **authorization** and **token** endpoints in the list below:

```
localhost:8024/auth/realms/keep-growing/.well-known/openid-configuration
{
  "issuer": "http://localhost:8024/auth/realms/keep-growing",
  "authorization_endpoint": "http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/token",
  "introspection_endpoint": "http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/logout",
  "frontchannel_logout_session_supported": true,
}
```

Make sure that the **host** is consistent with the security **properties in your API**. If you use **127.0.0.1** instead of **localhost** in the API security config, consequently use the **127.0.0.1** value for the urls in Postman. I'm going to create the following variables in Postman:

- `{{authUrl}}` with the `http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/auth` value
- `{{accessTokenUrl}}` with the `http://localhost:8024/auth/realms/keep-growing/protocol/openid-connect/token` value.

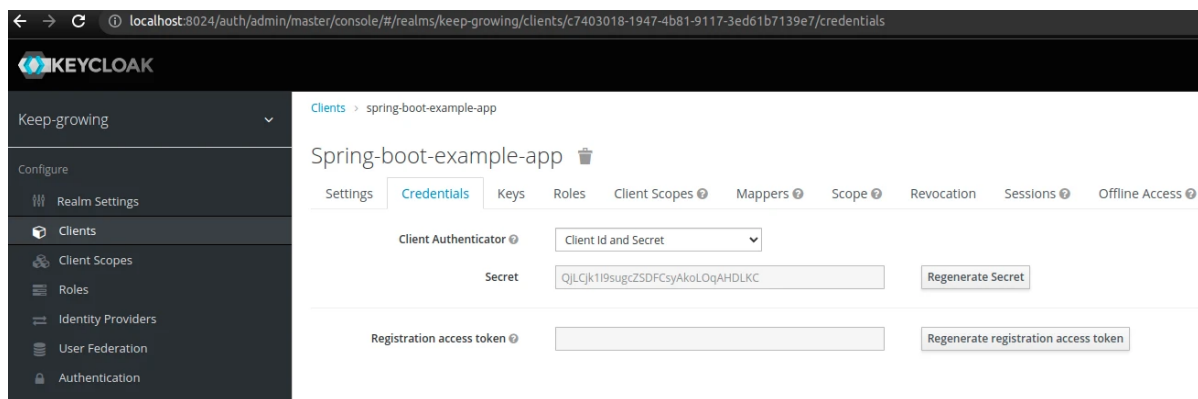
Client ID

It's the identifier for our client:



Client secret

As I mentioned before, my sample client has the **confidential** access type. Therefore, its configuration includes the **Credentials** tab, where I generated the following secret:



I'm going to copy that value and put it in the `{{clientSecret}}` variable.

Full variable list

To summarize, all the environment variables I have added to my Postman collection are below:



	VARIABLE	INITIAL VALUE ⓘ	CURRENT VALUE ⓘ	...	Persist All	Reset All
<input checked="" type="checkbox"/>	baseUrl	http://localhost:8080	http://localhost:8080			
<input checked="" type="checkbox"/>	accessTokenUrl	http://127.0.0.1:8024/...	http://127.0.0.1:8024/auth/realms/keep-growing/protocol...			
<input checked="" type="checkbox"/>	clientId	spring-boot-example-...	spring-boot-example-app			
<input checked="" type="checkbox"/>	clientSecret	QJLCjk1I9sugcZSDFCsy...	QJLCjk1I9sugcZSDFCsyAkoLOqAHDLC			
<input checked="" type="checkbox"/>	xsrftoken		d0111878-3386-4d0f-ac87-346ae13d41a4			
<input checked="" type="checkbox"/>	redirectUri	http://localhost:8080/*	http://localhost:8080/*			
<input checked="" type="checkbox"/>	authUrl	http://127.0.0.1:8024/...	http://127.0.0.1:8024/auth/realms/keep-growing/protocol...			
	Add a new variable					

ⓘ Use variables to reuse values in different places. Work with the current value of a variable to prevent sharing sensitive values with your team. [Learn more about variable values](#)

Cancel

Update

If your API uses csrf protection, read the [How to add X-XSRF-TOKEN header to Postman requests](#) article. Otherwise, you'll get 403 errors even after completing this tutorial.

Configuration required to authorize Postman requests in Keycloak

Now I will set up authorization for my Postman collection using the variables mentioned above and providing some additional information. First, I'll edit the collection:



Then, I'll and go the **Authorization** tab and select the **OAuth 2.0** authorization type to configure a new token. Below you'll see what additional information I'm going to provide.

Add authorization data to

I'm going to select the **Request Headers** option.

Token Name

This is an arbitrary value to distinguish this particular token from others you keep in Postman. I'm going to call it **spring-boot-example-app token**.

Grant Type

I'm going to select the **Authorization Code (With PKCE)** option.

Scope

Verify what scopes are available for your client.:

[JavaTools](#)[Taking care](#)[Angular](#)About   

Optional Client Scopes ⓘ	Available Client Scopes ⓘ	Assigned Optional Client Scopes ⓘ
	<div>Add selected »</div>	<div>« Remove selected</div>
	<div>Add selected »</div>	<div>« Remove selected</div>

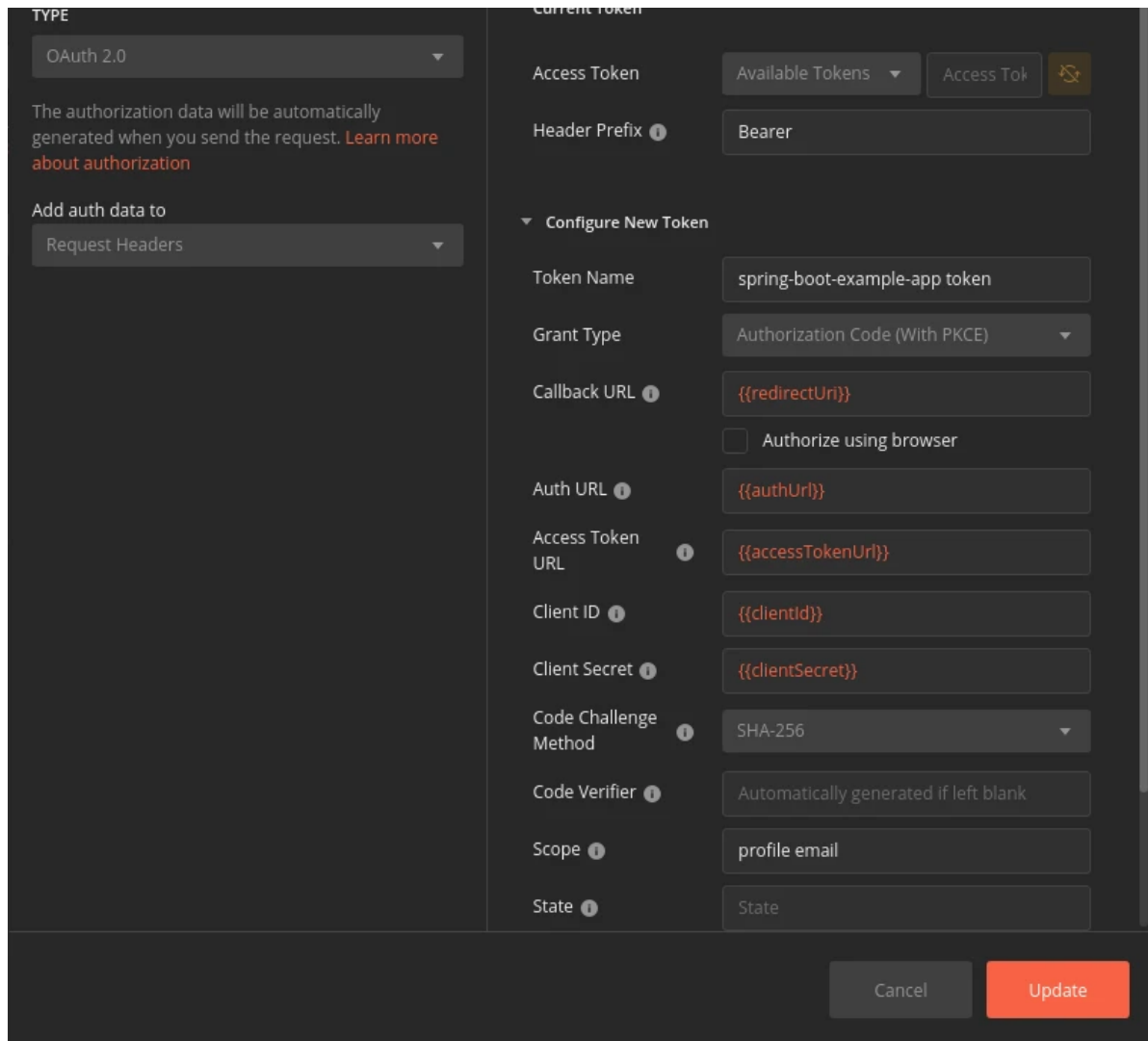
I'm going to provide the **profile** **email** value here.

Client Authentication

I'm going to select the **Send client credentials in body** option.

Full configuration

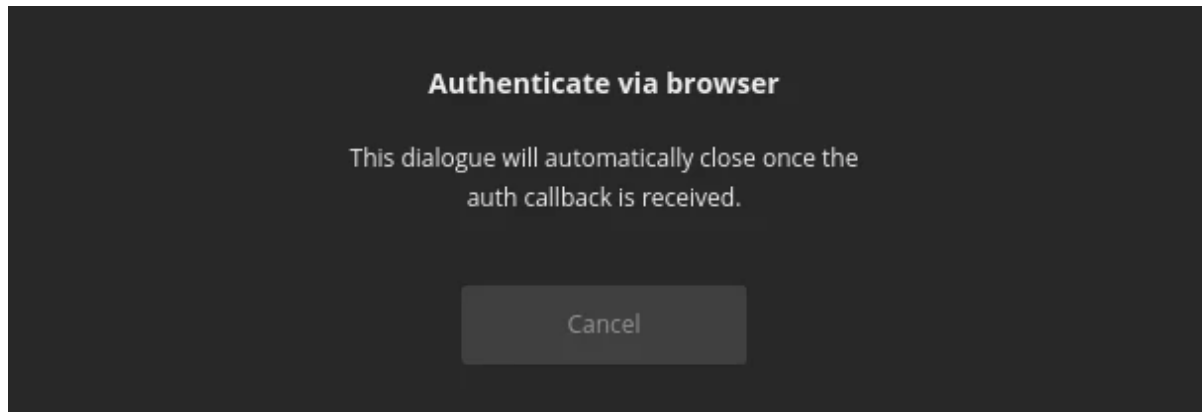
You can see the complete configuration in the screenshot below:



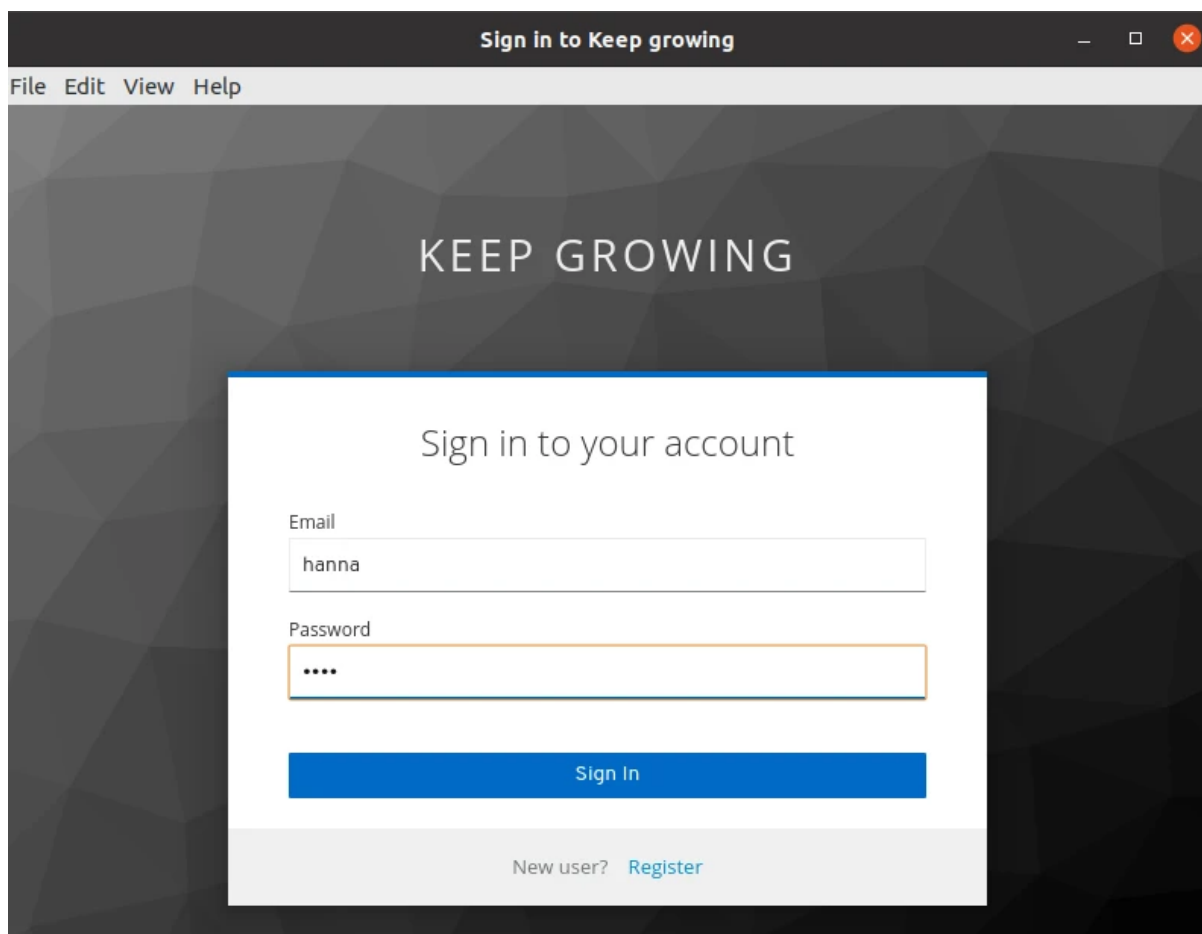
The image shows the Postman OAuth 2.0 configuration interface. On the left, under 'TYPE', 'OAuth 2.0' is selected. Below it, a note states: 'The authorization data will be automatically generated when you send the request. [Learn more about authorization](#)'. Under 'Add auth data to', 'Request Headers' is selected. On the right, under 'Current Token', there are buttons for 'Available Tokens', 'Access Tok', and a refresh icon. Below this, the 'Header Prefix' is set to 'Bearer'. The 'Configure New Token' section contains the following fields: 'Token Name' (spring-boot-example-app token), 'Grant Type' (Authorization Code (With PKCE)), 'Callback URL' ({{redirectUri}}), 'Auth URL' ({{authUri}}), 'Access Token URL' ({{accessTokenUri}}), 'Client ID' ({{clientId}}), 'Client Secret' ({{clientSecret}}), 'Code Challenge Method' (SHA-256), 'Code Verifier' (Automatically generated if left blank), 'Scope' (profile email), and 'State' (State). At the bottom right, there are 'Cancel' and 'Update' buttons.

Get a new Access Token

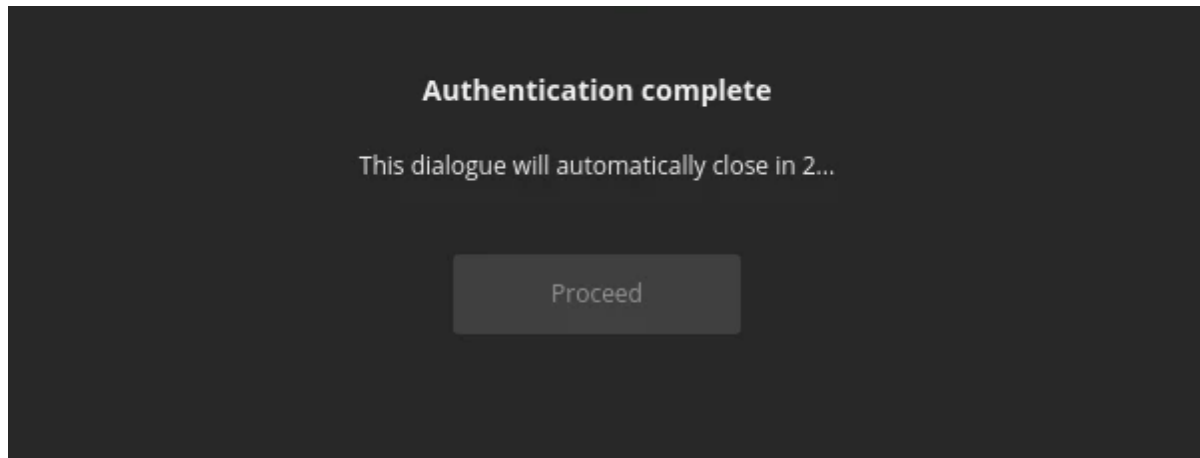
Finally, we can click the **Get new Access Token** button. We're going to see the following Postman screen informing us that authentication will be done through a browser:



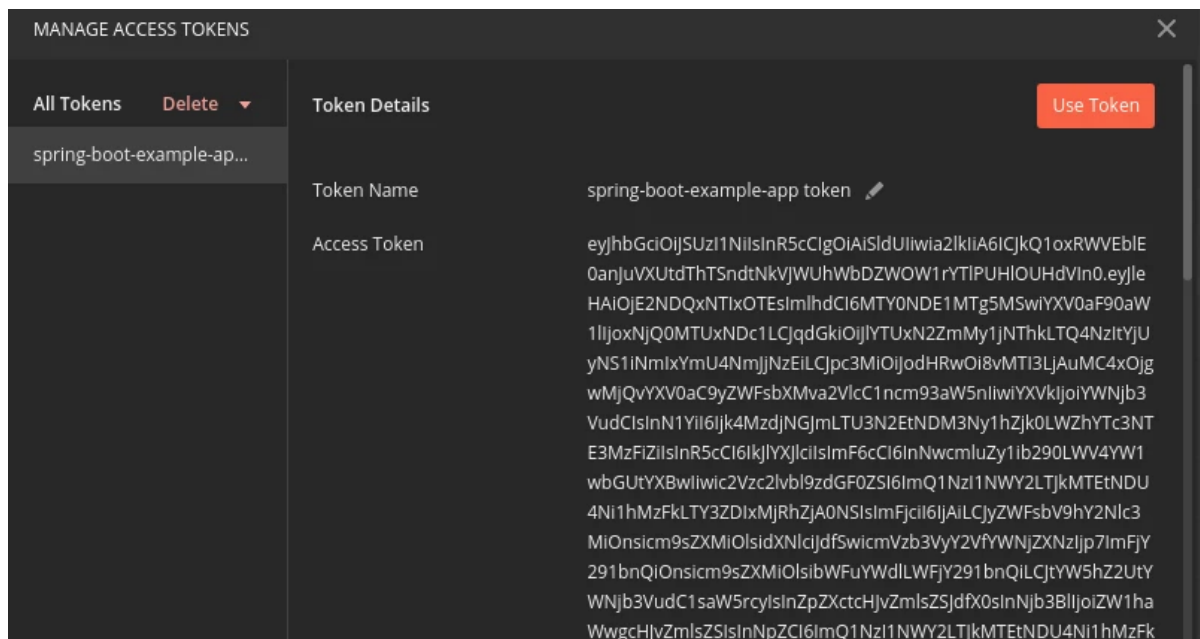
Meanwhile, our authorization server will provide the a login form. Below you can see my Keycloak login screen where I authenticate as an example user of my realm:



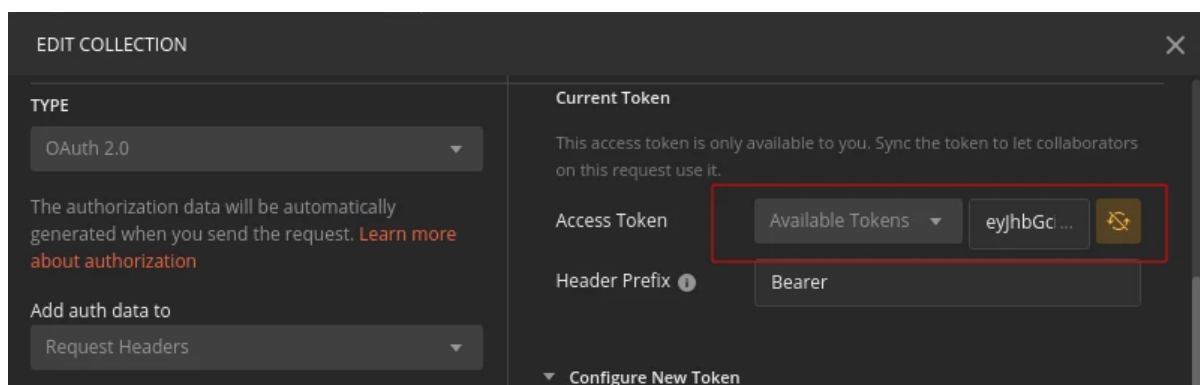
After successful authentication, we will see the following screen for a few seconds:



As a result, a new **spring-boot-example** token value is available in the **Manage Access Token** window. Click the **Use Token** button:



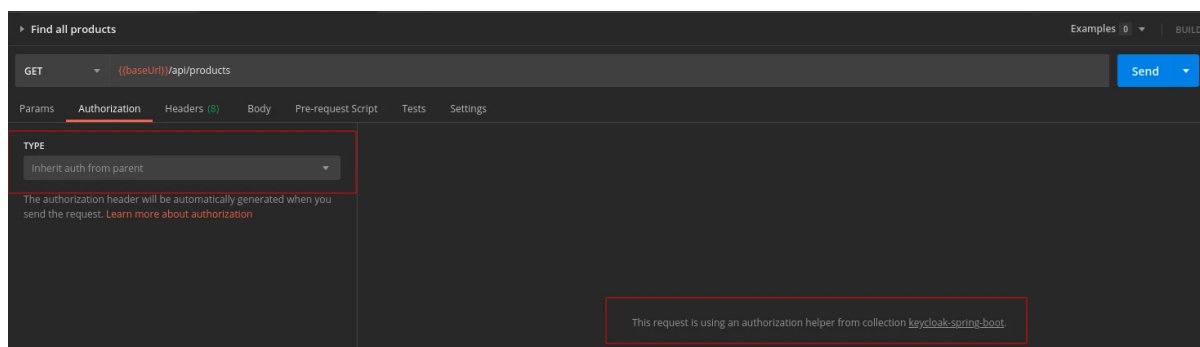
You'll see the token value in the **Current Token** section:



[JavaTools](#)[Taking care](#)[Angular](#)About    

Make the token available in all requests

To make this token accessible in all requests in my collection, I'm going to select the **Inherit auth from parent** option in the request **Authorization** tab:

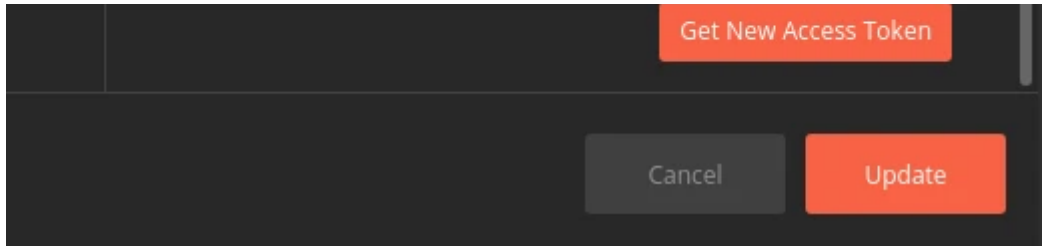


As a result, Keycloak will authorize all requests with this config from my Postman collection.

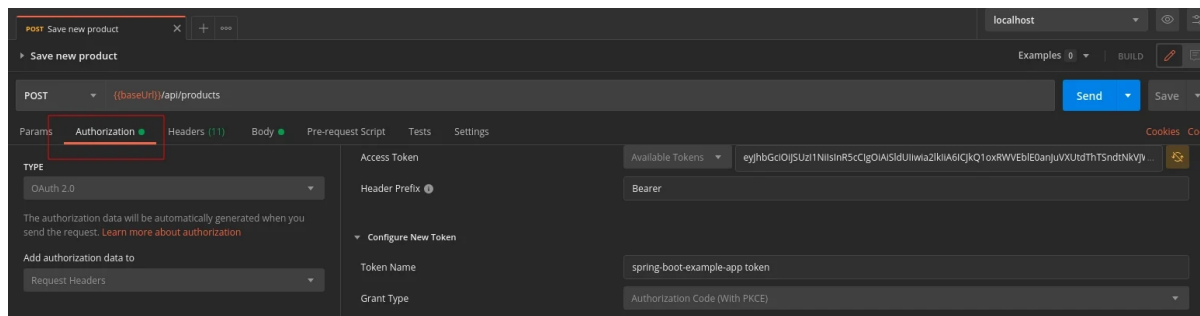
Update and manage Access Tokens

When the token expires, generate a new value. Just edit your collection, go to the **Authorization** tab, click the **Generate New Access Token** button, then click the **Use Token** button and finally click the **Update** button:

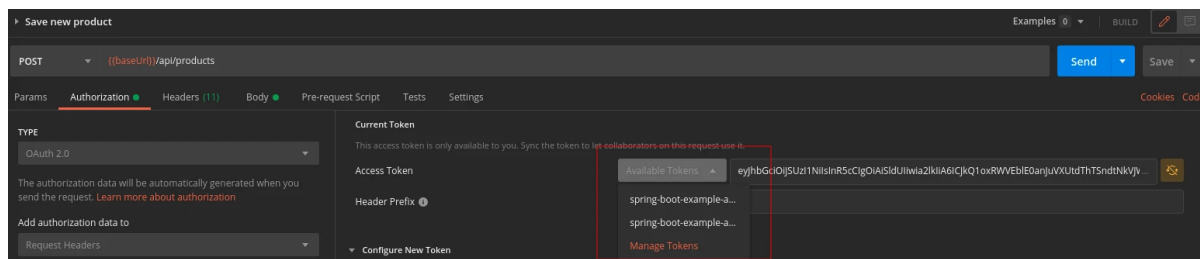
JavaToolsTaking careAngular

About Search ...

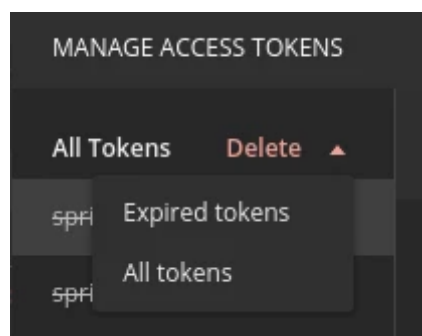
To update the token or to overwrite the collection config for only one request, go to the **Authorization** tab of that request and select the **OAuth 2.0** for the authorization type. You'll be able to generate a new token or redefine the configuration values:



Furthermore, you can select a specific token value from the **Available Tokens** dropdown:



Finally, you can remove some tokens by clicking the **Manage Tokens** option:





Why not Implicit flow

This is the easiest flow. Nonetheless, it comes with some severe security vulnerabilities:

The implicit grant (...) and other response types causing the authorization server to issue access tokens in the authorization response are vulnerable to access token leakage and access token replay (...).

Moreover, no viable mechanism exists to cryptographically bind access tokens issued in the authorization response to a certain client (...). This makes replay detection for such access tokens at resource servers impossible.

In order to avoid these issues, clients SHOULD NOT use the implicit grant (...), unless access token injection in the authorization response is prevented and the aforementioned token leakage vectors are mitigated.

<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics#section-2.1.2>

The vulnerabilities were described in depth in the **Implement the OAuth 2.0 Authorization Code with PKCE Flow** post on the Okta blog:

Notice that after you authenticate, the Authorization Server (like Google) responds directly with tokens. This means that the tokens are in your browser's address bar as a result of the redirect. That's problematic since Google can't definitively know that your browser (the intended recipient) actually received the response. It's also problematic because modern browsers can do browser history syncing and they support browser extensions that could be actively scanning for tokens in the browser address bar. Leaking tokens is a big security risk.

<https://developer.okta.com/blog/2019/08/22/okta-authjs-pkce#why-you-should-never-use-the-implicit-flow-again>

As a result of these security concerns, the Implicit Flow is officially deprecated:

Why not Password flow

Another convenient flow that we can configure in Postman, the **Password Credentials** Grant Type, also comes with an increased risk of attack:

The resource owner password credentials grant MUST NOT be used. This grant type insecurely exposes the credentials of the resource owner to the client. Even if the client is benign, this results in an increased attack surface (credentials can leak in more places than just the authorization server) and users are trained to enter their credentials in places other than the authorization server.

<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-13#section-3.4>

Because the client application has to collect the user's password and send it to the authorization server, it is not recommended that this grant be used at all anymore.

<https://oauth.net/2/grant-types/password/>

More on how to authorize Postman requests in Keycloak

- [Authorizing requests](#), [OAuth 2.0 authorization type](#), [Requesting an OAuth 2.0 token](#) in the Postman documentation and the example [collection that documents a few OAuth 2.0 authorization flows](#).
- [OAuth 2.0: Implicit Flow is Dead, Try PKCE Instead](#) and [OAuth 2 Simplified](#) articles.
- [Proof Key for Code Exchange by OAuth Public Clients](#)

Photo by [ELEVATE](#) from [Pexels](#)



Name *

Email *

Website

☐ By using this form you agree with the storage and handling of your data by this website. *

POST COMMENT

Created by [Marta Szymek](#)

[Privacy Policy](#)