**What is OpenID Connect, and How is it Different from OAuth?**

OpenID Connect (OIDC) is an authentication layer built on top of the OAuth 2.0 framework. While OAuth facilitates **authorization** through access tokens containing specific scopes, OpenID Connect adds **authentication** by introducing an ID token. The ID token includes user-specific information and claims for identity verification.

**Why is OpenID Connect Important?**

OAuth by itself only issues access tokens with scopes, which indicate the permissions granted but do not identify the end user. OpenID Connect addresses this gap by providing ID tokens, which include details about the end user or resource owner attempting to access information. This ensures that services can verify **who** the user is, not just **what** they are authorized to do.

**What is Keycloak?**

Keycloak is an Identity and Access Management (IAM) solution that simplifies setting up an authorization server for organizations. While large companies often build their own servers using OAuth and OpenID Connect, smaller organizations can leverage Keycloak to efficiently manage authentication and authorization.  
Other platforms that provide similar services include Okta, Amazon Cognito, and ForgeRock.

**Types of OAuth Grant Flows**

**Client Credentials Grant Type Flow**

The Client Credentials flow is used for machine-to-machine communication without user involvement.

**How it Works:**

1. **Request Access Token**
   * The client sends a request to the Authorization Server (Auth Server) to obtain an access token.
   * This request includes the client’s credentials (e.g., Client ID and secret).
2. **Issue Access Token**
   * The Auth Server validates the client’s credentials.
   * If valid, it issues an access token to the client.
3. **Request Protected Resources**
   * The client sends the access token to the Resource Server when requesting protected resources.
4. **Validate Token and Provide Resources**
   * The Resource Server validates the token with the Auth Server.
   * Upon successful validation, the Resource Server grants access to the requested resources.

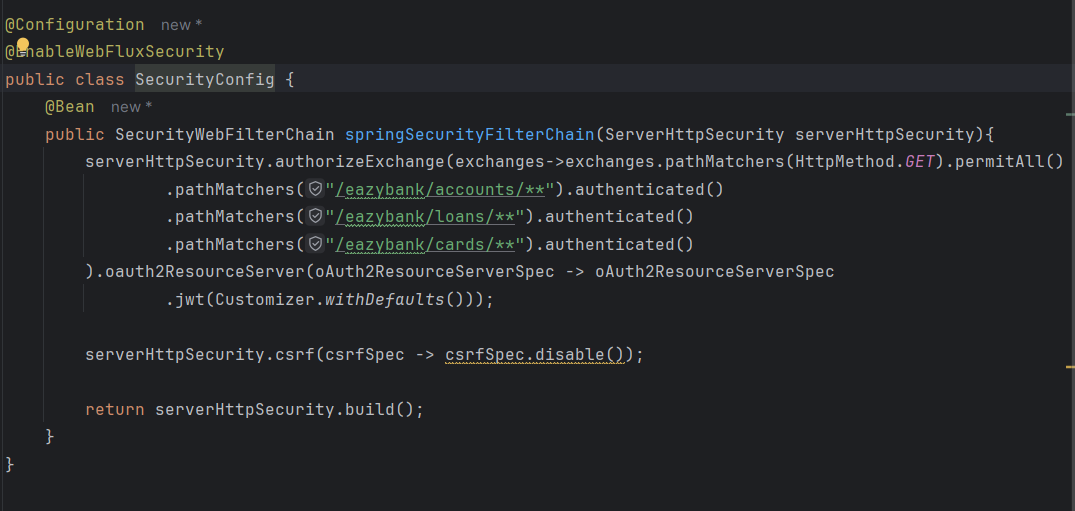
**Key Characteristics:**

* **No User Interaction:** Designed for server-to-server communication.
* **Security:** Client credentials must be securely stored and transmitted.
* **Use Case:** Ideal for APIs, microservices, or backend-to-backend communication.

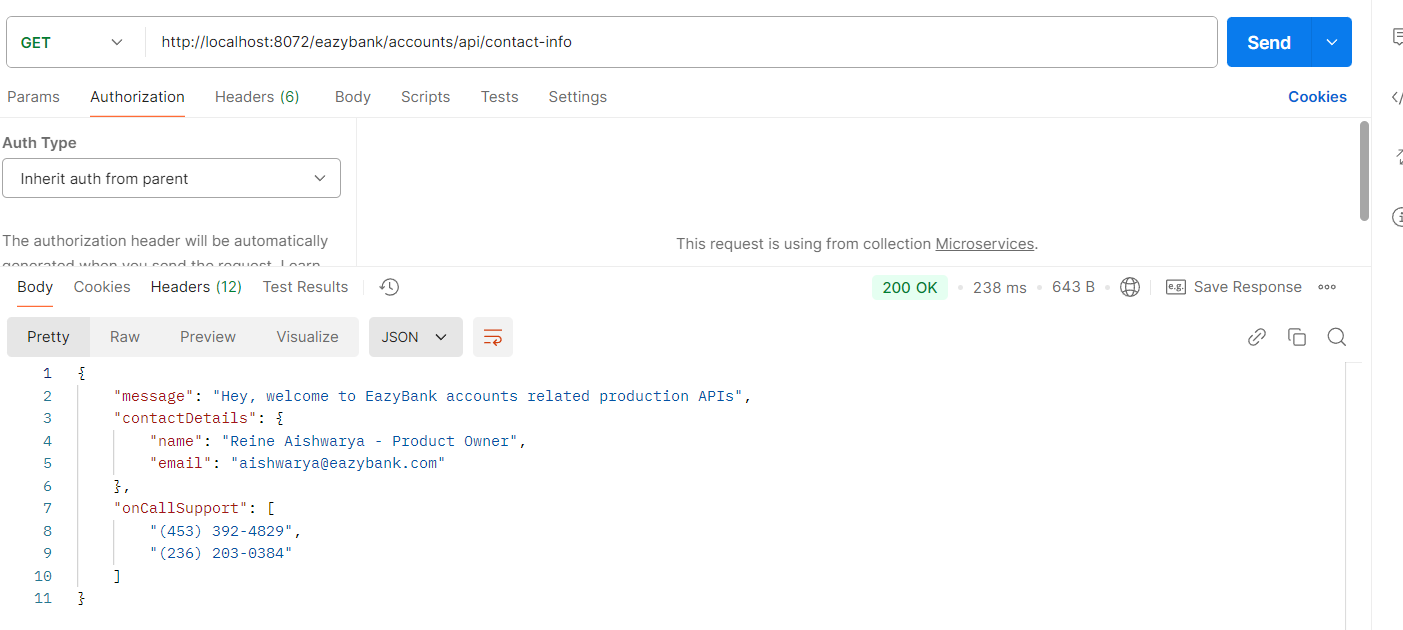
**Implementation**

In a gateway server, we configure security settings using a SecurityConfig class. This class specifies:

* HTTP methods allowed without authentication (e.g., GET requests).
* Methods requiring authentication (e.g., POST requests).

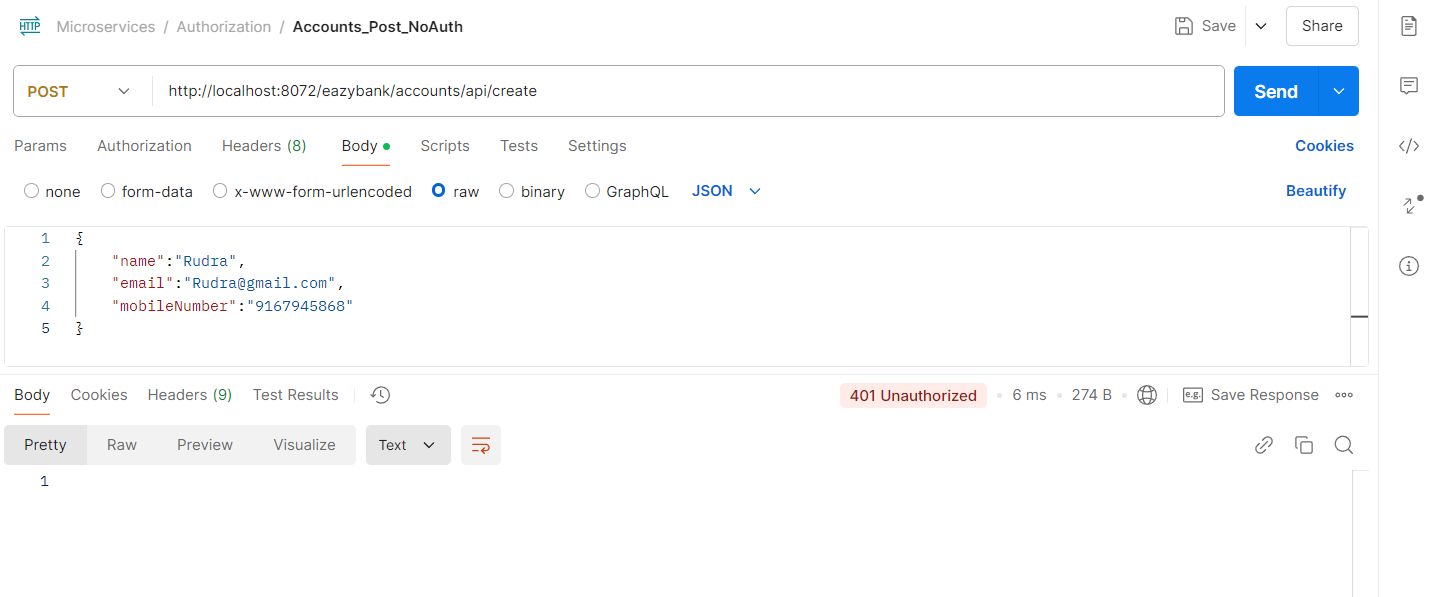


As shown in above code, Http Get Methods are allow to invoke without any authentication but in case of post method we’re asking our services to get authenticated.



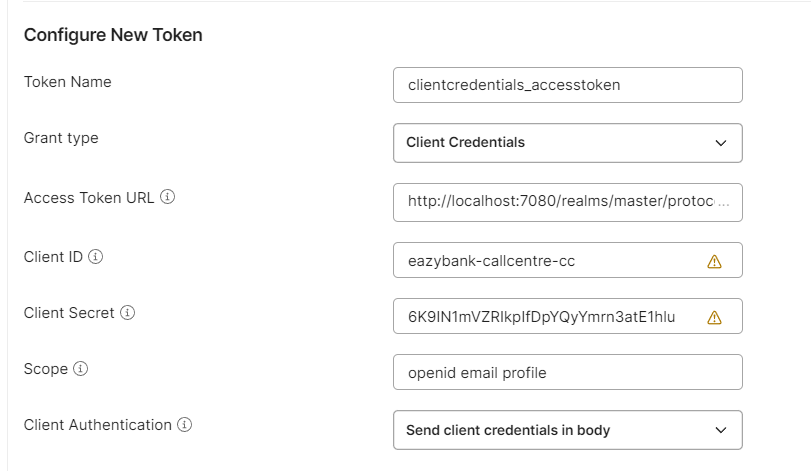
As demonstrated, we can fetch the contact information from the accounts microservice without any authentication. This is achieved by configuring the oauth2ResourceServer method, which specifies that our gateway server will act as an OAuth resource server and use the default configuration to handle JWT tokens.

However, when attempting to submit data via a POST request, the server returns a **401 Unauthorized Error**. This occurs because no access token has been generated or provided to authenticate the request.



**Generating and Using Access Tokens**

1. **Registering a Client in Keycloak:**
   * Log in to the Keycloak portal.
   * Navigate to the "Client" tab and create a client.
   * Select **OpenID Connect** as the client type and specify a Client ID (e.g., eazybank-callcentre-cc).
   * Save and note the Client Secret.
2. **Using Access Tokens:**
   * In tools like Postman, include the Client ID, Client Secret, and scope when generating an access token.
   * Use the token to authenticate requests (e.g., for creating accounts, loans, or cards).



This process ensures secure communication and proper authorization for services interacting with the gateway.

In an organization, specific roles may need access to invoke certain REST APIs. To enforce this, we modify the SecurityConfig file by replacing the authenticated() method with the hasRole() method, as shown below. This ensures that only users with the required roles can access the APIs.



Additionally, we must define these roles in the OAuth server (e.g., Keycloak). To do this:

1. Navigate to the **Roles** section in the Keycloak server and create the required roles.
2. Assign these roles to the respective client by editing the client configuration.

This setup ensures that the access control for APIs is role-based and aligned with the organization's security policies.

The KeycloakRoleConverter class implements the Converter interface and overrides its convert method. This method takes a JWT (JSON Web Token) as input and returns a **Collection of GrantedAuthority** (i.e., a collection of roles). The primary purpose of this class is to extract roles from the JWT and convert them into a format that Spring Security understands.

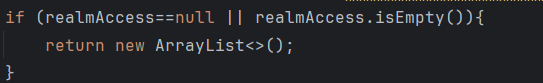
**Step-by-Step Breakdown:**

1. **Extracting realm\_access from the JWT:**
   * The JWT contains claims as a HashMap. To extract the roles, we first retrieve the realm\_access section of the token using the following command:



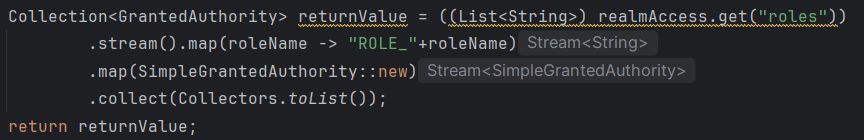
* + Here, realm\_access is expected to be a nested map within the JWT containing user roles.

1. **Checking for Null or Empty realm\_access:**
   * Before proceeding, we ensure that realm\_access is not null or empty:



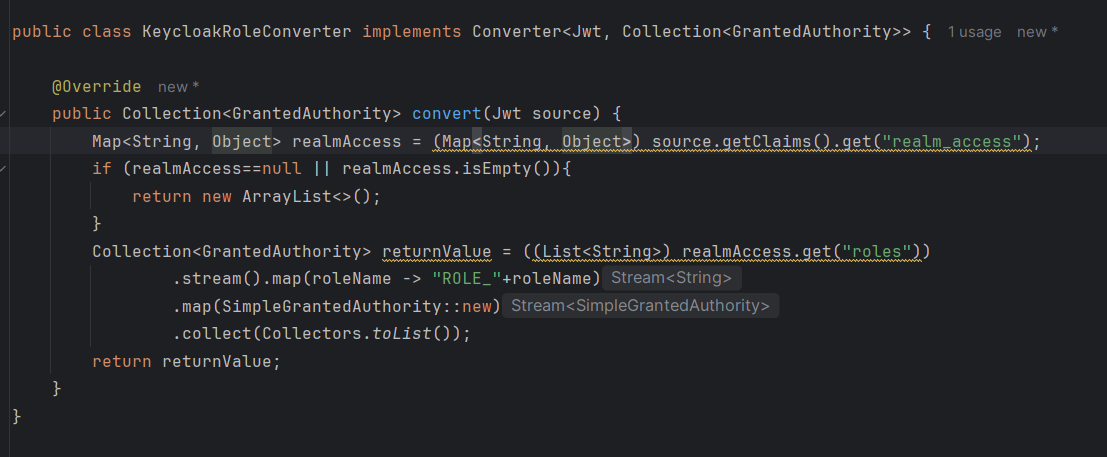
* + If no roles are found, an empty list is returned, ensuring no invalid data is processed.

1. **Extracting and Transforming Roles:**
   * The roles field within realm\_access contains a list of role names. We extract this list and use a lambda expression to:
     + Prefix each role with ROLE\_, as Spring Security expects role names in this format (e.g., admin becomes ROLE\_admin).
     + Convert each role into a SimpleGrantedAuthority object, which implements the GrantedAuthority interface in Spring Security.
   * The transformation is performed using the following code:



1. **Returning the Result:**
   * Finally, the method returns the list of GrantedAuthority objects, which can then be used by Spring Security for role-based access control.

Below is the completed code which we’ve to write to extract the roles from JWT which can be understand by spring framework:



Now once we write down the logic to convert JWT token. We’ve to mention same in SecurityConfig under “oAuth2ResourceServerSpec.jwt” method.

We’ll create separate method called “grantedAuthoritiesExtractor” which will return the Converter<Jwt, Mono<AbstractAuthenticationToken>>.

A **converter** is a functional interface that processes a JWT and returns a reactive AbstractAuthenticationToken, which contains the user's authentication details, including their roles or authorities.

**Code break down:**

Step 1: Creating JwtAuthenticationConverter



JwtAuthenticationConverter is a class provided by Spring Security.

It is used to convert a JWT into authentication details, including granted authorities (roles).

**Step 2: Configuring the KeycloakRoleConverter**

****

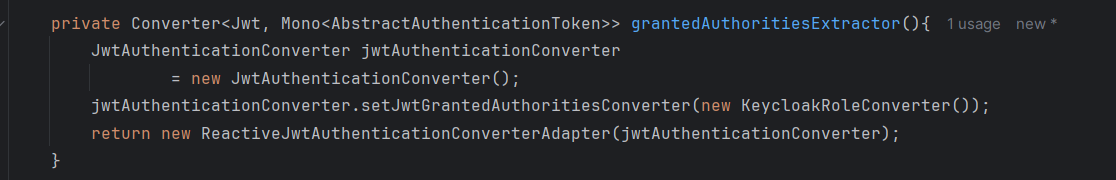
* The KeycloakRoleConverter is being set as the **JwtGrantedAuthoritiesConverter** for the JwtAuthenticationConverter.
  + - KeycloakRoleConverter extracts roles from the JWT (as described in the earlier explanation) and converts them into a collection of GrantedAuthority objects that Spring Security uses.
    - This ensures that the roles defined in the Keycloak JWT are correctly processed and understood by the Spring Security framework.

**Step 3: Wrapping with ReactiveJwtAuthenticationConverterAdapter**

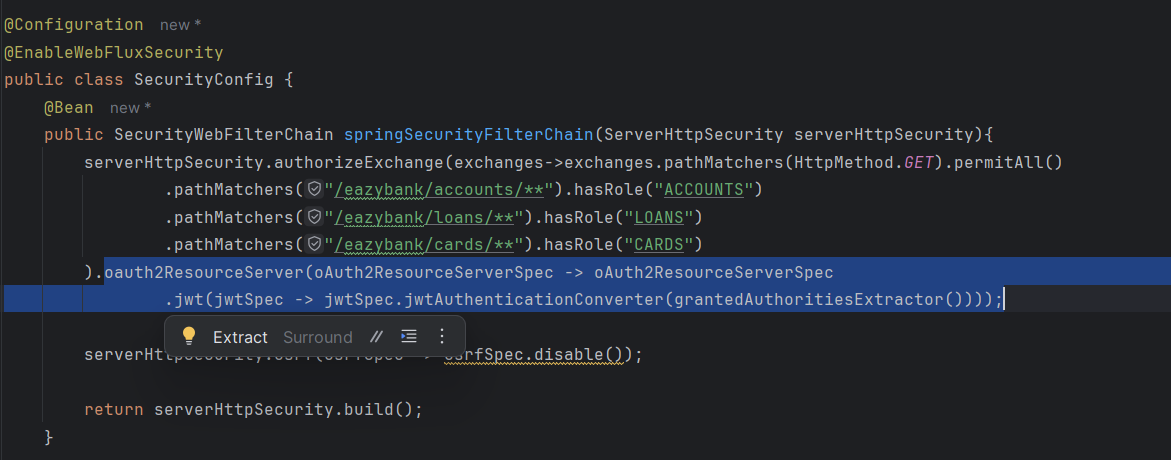
****

* The ReactiveJwtAuthenticationConverterAdapter is a wrapper class that adapts the synchronous JwtAuthenticationConverter for use in reactive applications.
* Reactive applications work asynchronously, and this adapter ensures that the JwtAuthenticationConverter can be used in such an environment.
* The return type of this method is Converter<Jwt, Mono<AbstractAuthenticationToken>>, which aligns with the reactive programming paradigm.

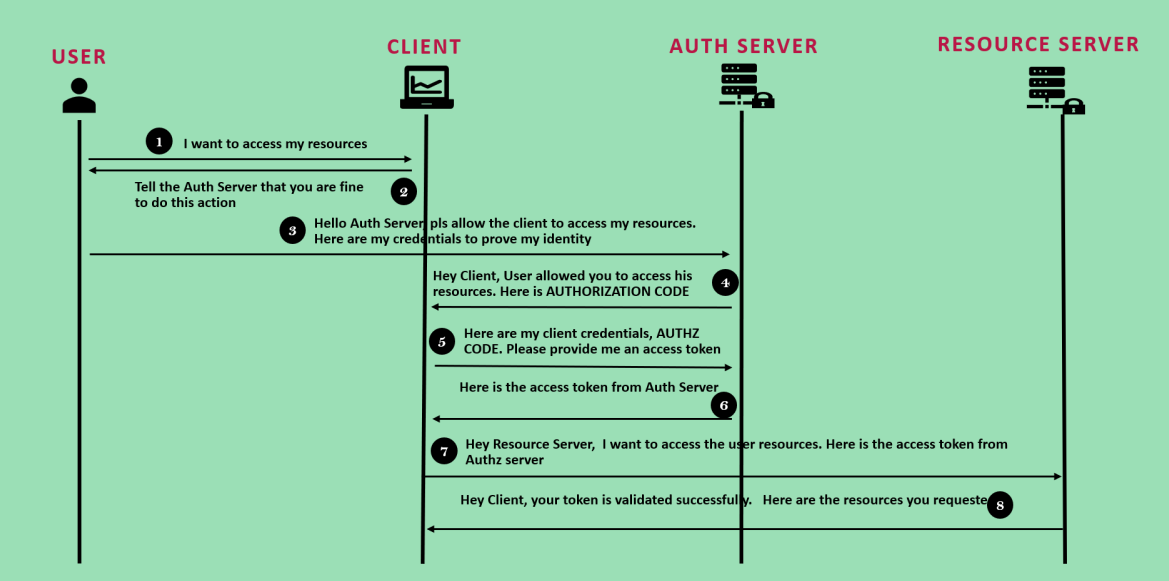
Below is the complete expected code:



After this, we’ve to mention same method name under “oAuth2ResourceServer.jwt” as shown below:



**2) Authorization code grant type flow**



Below are the steps by step which Authorization grant type flow will follow:

**1. User Requests Access**

* The user interacts with the client application (e.g., a mobile app or web app) and says:  
  *“I want to access my resources.”*

**2. Client Redirects User to Auth Server**

* The client redirects the user to the **Auth Server**, asking the user to approve this action.
* The Auth Server displays a prompt to the user saying something like:  
  *“This app wants permission to access your resources. Is that okay?”*

**3. User Approves Access**

* If the user agrees, they give their consent and authenticate themselves (e.g., by logging in).
* The Auth Server then says to the client:  
  *“The user has given their approval. Here’s the* ***Authorization Code*** *you can use to proceed.”*

**4. Client Receives Authorization Code**

* The **Authorization Code** is sent back to the client application. This code is short-lived and acts as proof that the user authorized the request.

**5. Client Exchanges Code for Access Token**

* The client sends the Authorization Code, along with its own credentials (like a client ID and secret), to the **Auth Server**.
* It says:  
  *“Here’s the Authorization Code and my credentials. Please give me an Access Token so I can access the user’s resources.”*

**6. Auth Server Issues Access Token**

* The Auth Server validates everything. If the Authorization Code and client credentials are valid, it responds:  
  *“Here’s your Access Token. Use this to access the protected resources.”*

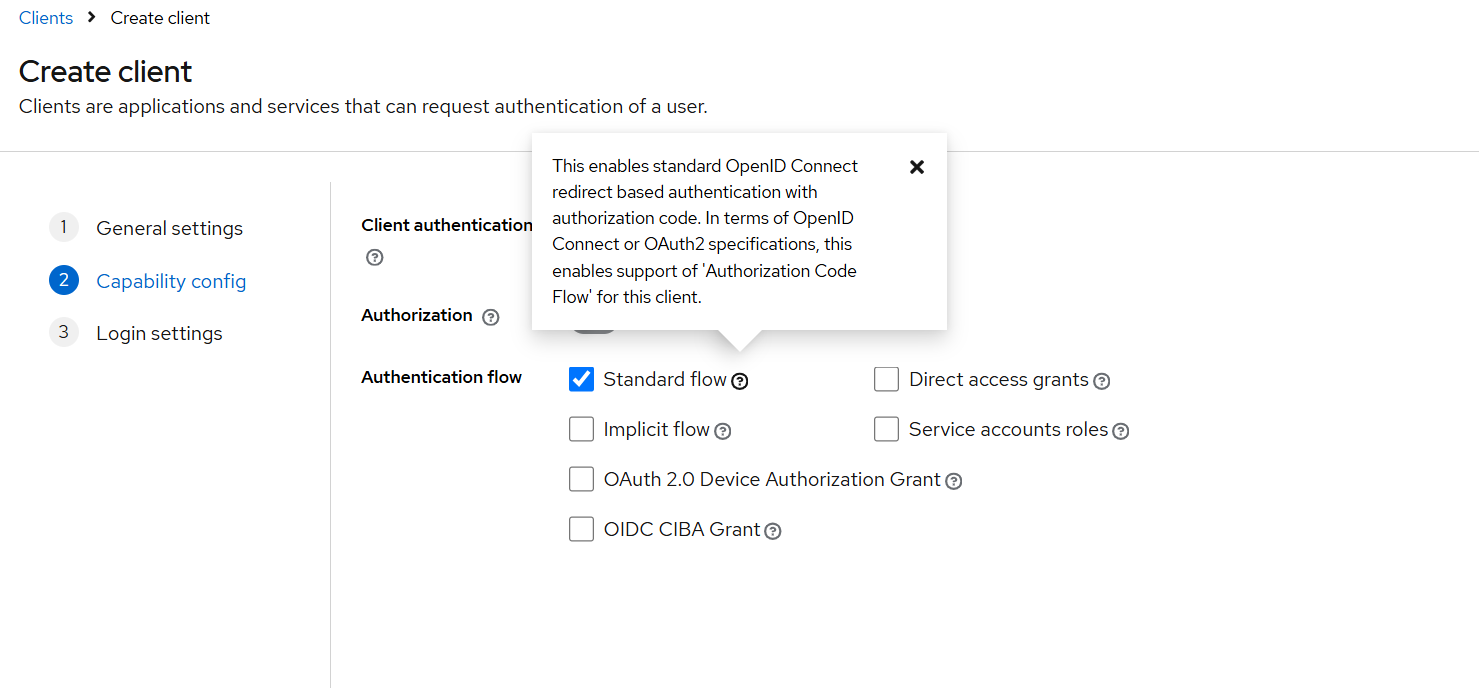
**7. Client Requests Resources**

* The client then sends a request to the **Resource Server**, including the Access Token.
* It says:  
  *“I’d like to access the user’s resources. Here’s my Access Token to prove I’m authorized.”*

**8. Resource Server Validates Token and Sends Resources**

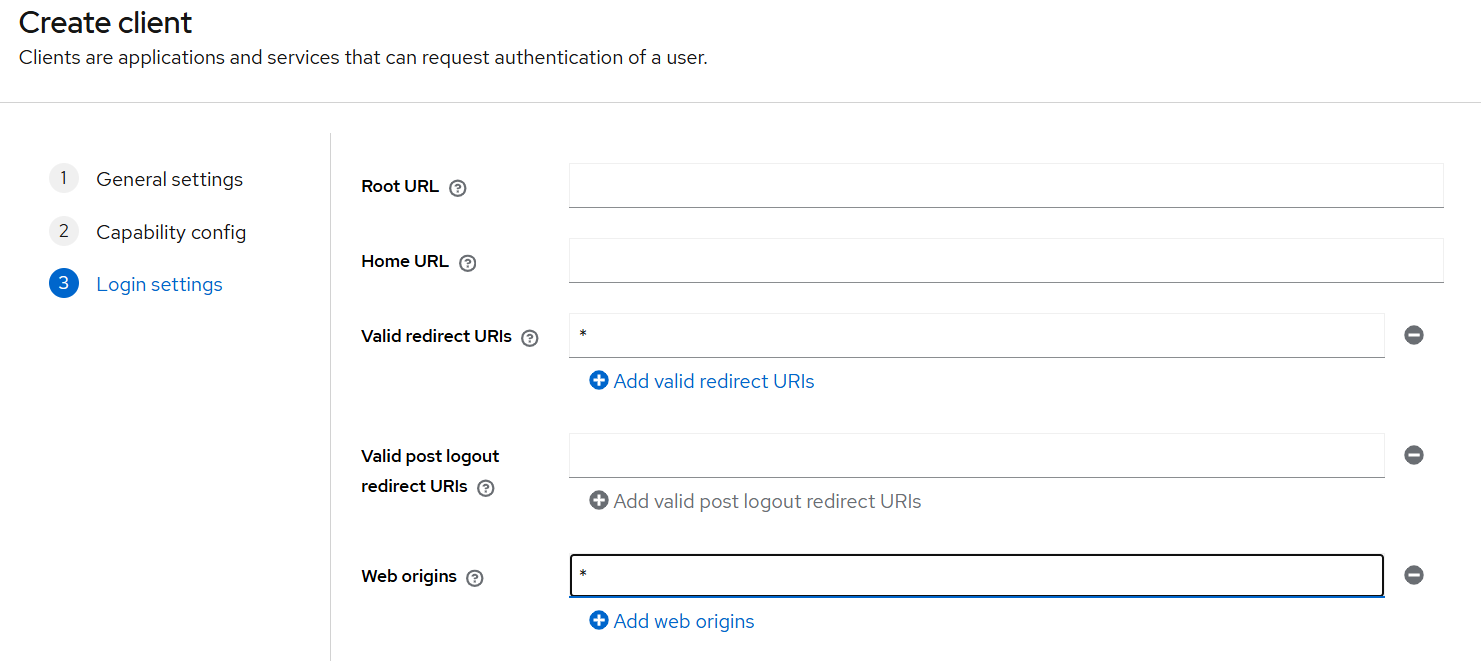
* The Resource Server validates the Access Token (by checking with the Auth Server if needed).
* Once validated, it provides the requested resources to the client, saying:  
  *“Your token is valid. Here are the resources the user authorized you to access.”*

Implementation:

To implement role-based authentication, we first need to register our client in the Keycloak server with the **Authorization Code** flow. This flow is ideal for scenarios involving user authentication.

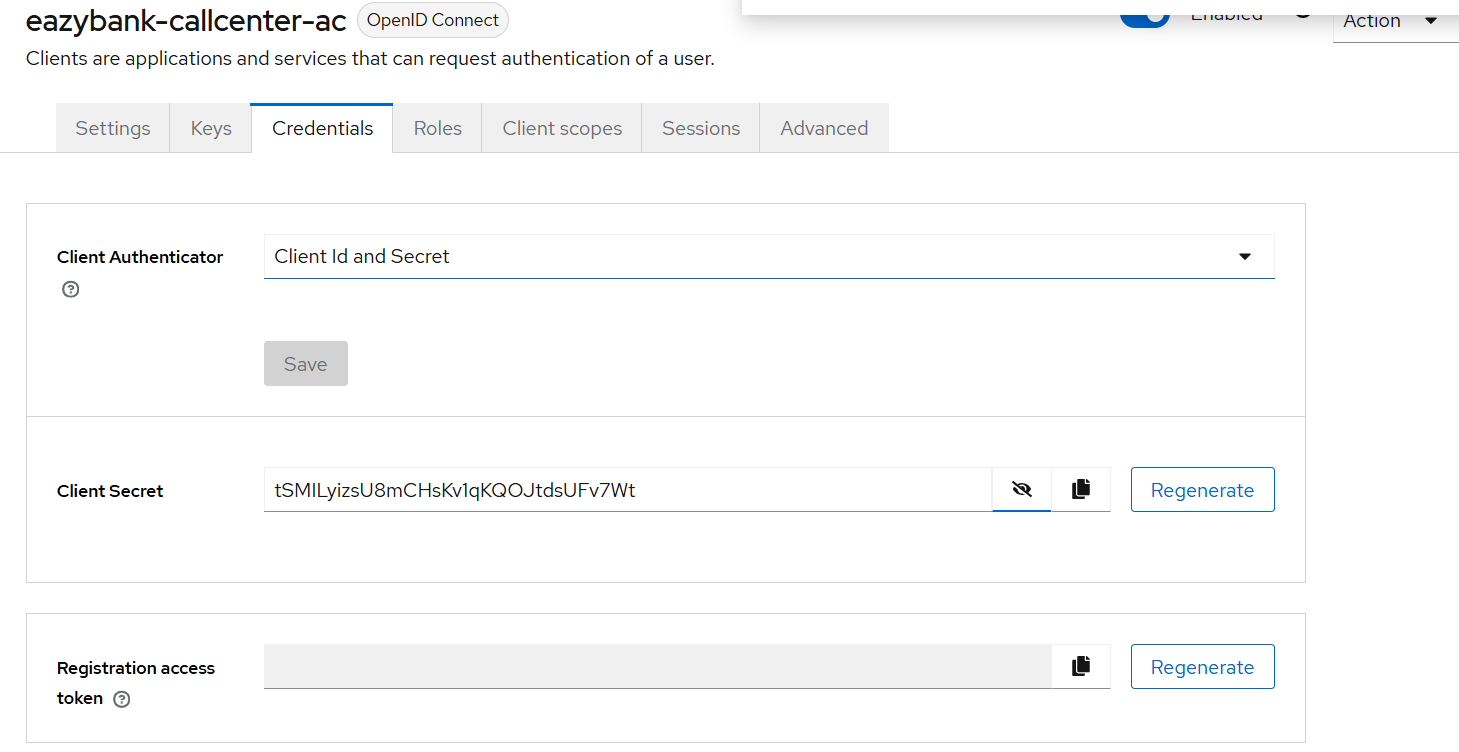
**Steps:**

1. **Client Registration:**
   * While creating the client, specify a **redirect URL**. This URL determines where the application will be redirected after successful authentication.
   * Since we lack a user interface in this case, we can set the redirect URL to \*, allowing redirection to any URL.
   * Similarly, under **Web Origins**, set \* to enable the authorization server to accept requests from different domains or servers, as our microservices might run on different ports.



1. **Store Client Secret:**

* After creating the client, note down the **client secret**, as it will be required for generating access tokens.



1. **User Creation:**

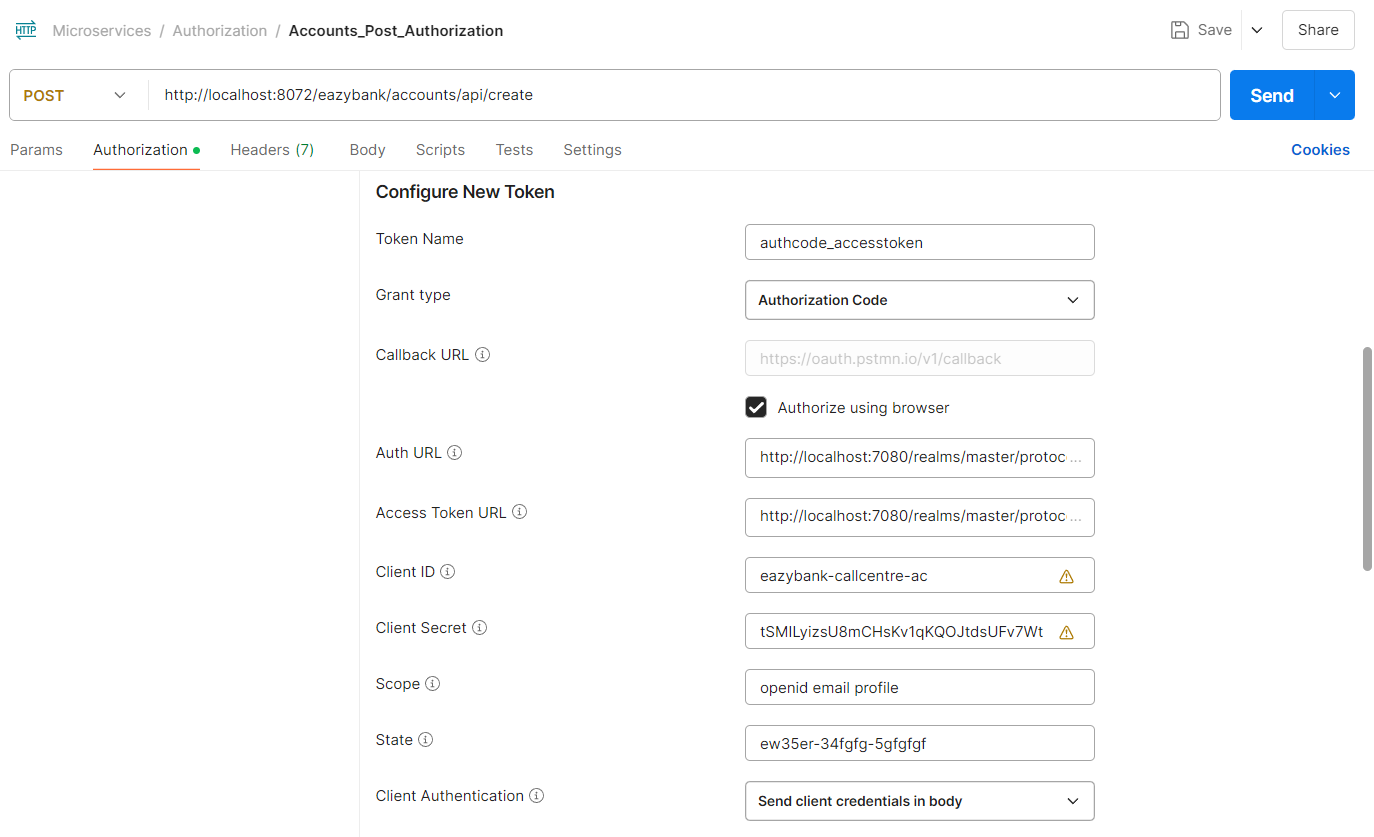
* Since the Authorization Code flow involves an end user, create a user in the Keycloak server.
* After creating the user, set up a password for the user to enable login.

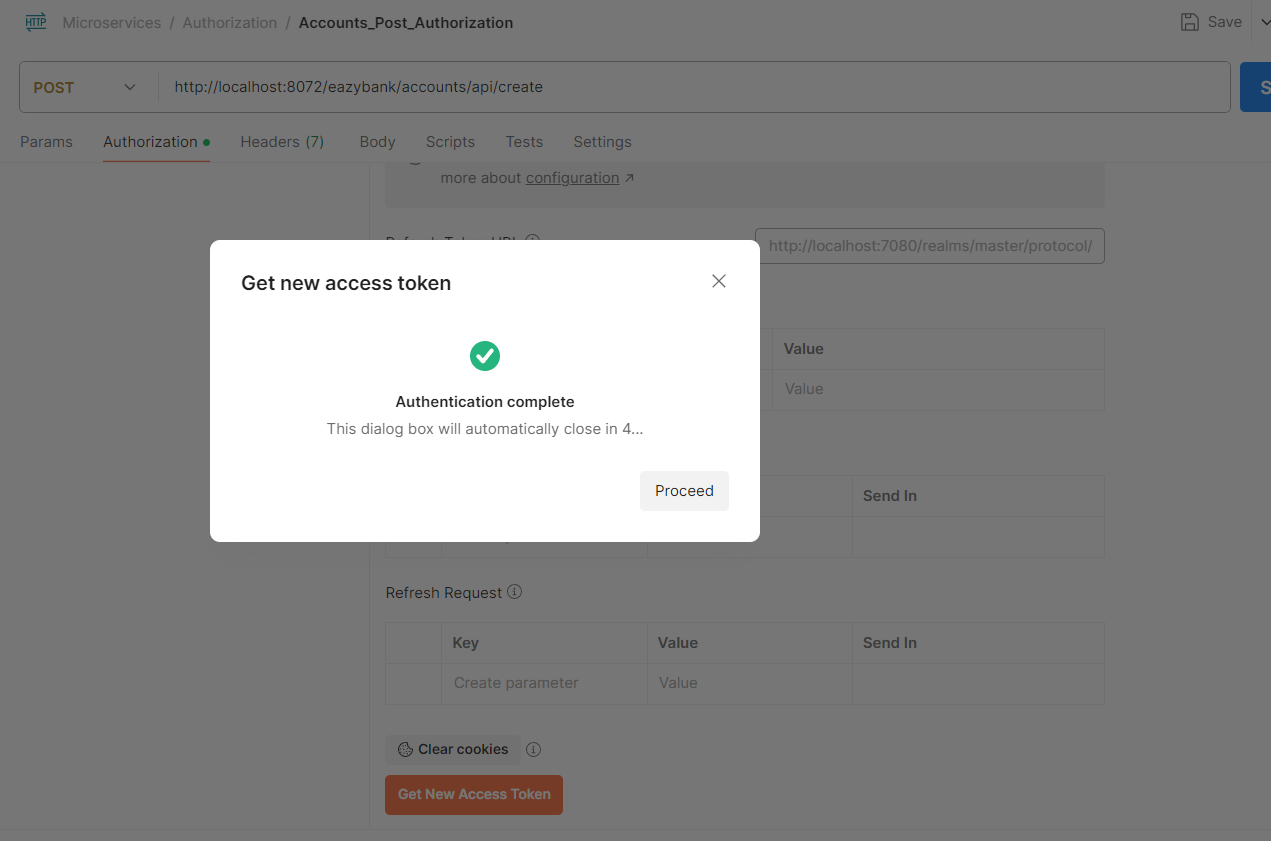
1. **Configuring Postman:**

* Before generating an access token, configure the following details in Postman:
  + **Auth URL**: The endpoint for authentication.
  + **Access Token URL**: The endpoint for token generation.
* These URLs can be obtained by running localhost:7080 if the Keycloak server is hosted on port 7080.

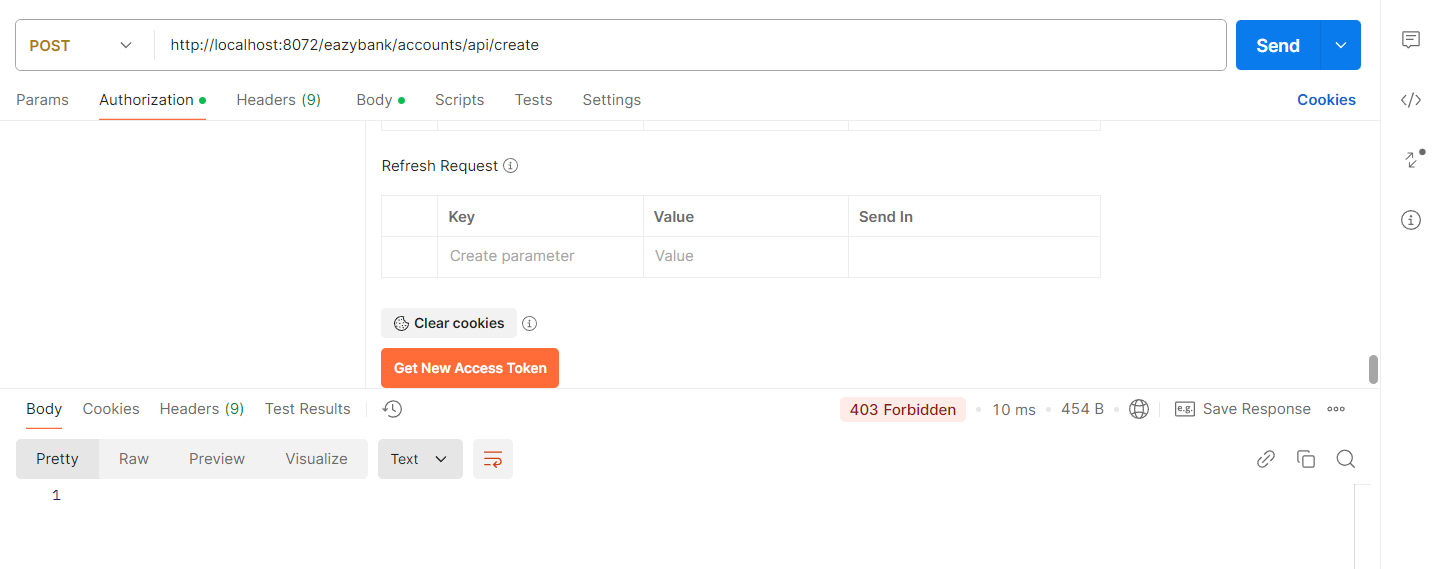
1. **Generating an Access Token:**

* Use Postman to initiate the process of generating a new access token.
* This action redirects you to the Keycloak portal for user login.
* Upon successful login, Keycloak confirms authentication and provides the access token.





This setup ensures secure role-based access for your REST APIs while allowing flexible cross-domain interactions for microservices.

Now if you see, I’m getting error as forbidden and that’s because I haven’t added roles at a user side. And after that we’re able to create new account.