

# Spring Boot Microservices Tutorial - Part 4

In Part 4 of this **Spring Boot Microservices Tutorial** series, we will secure our API Gateway using Keycloak

## What is Keycloak?

Keycloak is an open-source Authorization Server that can be used to outsource the authentication and authorization from our application. Keycloak supports various authentication and authorization protocols like OAuth2, OpenID Connect, SAML, etc. It also offers features like Single Sign On (SSO), and Multi-Factor Authentication (MFA) out of the box.

If you want to learn more about OAuth2 and OIDC you can refer to the below documentation

<https://oauth.net/2/> and <https://openid.net/developers/how-connect-works/>

## Download Keycloak

To download Keycloak, we must create the docker-compose.yml file inside our API gateway project.

### docker-compose.yml

```
version: '3.8'
services:
  keycloak-mysql:
    container_name: keycloak-mysql
    image: mysql:8
    volumes:
      - ./volume-data/mysql_keycloak_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: keycloak
      MYSQL_USER: keycloak
      MYSQL_PASSWORD: password
  keycloak:
    container_name: keycloak
    image: quay.io/keycloak/keycloak:24.0.1
```

```
command: [ "start-dev", "--import-realm" ]
environment:
  DB_VENDOR: MYSQL
  DB_ADDR: mysql
  DB_DATABASE: keycloak
  DB_USER: keycloak
  DB_PASSWORD: password
  KEYCLOAK_ADMIN: admin
  KEYCLOAK_ADMIN_PASSWORD: admin
ports:
  - "8181:8080"
volumes:
  - ./docker/keycloak/realms/:/opt/keycloak/data/import/
depends_on:
  - keycloak-mysql
```

The above file sets up Keycloak along with a MySQL database to store the keycloak configuration. For now, we are starting Keycloak in the dev environment using the 'start-dev' argument provided through the command field of the docker-compose configuration.

Now you can run the below command to start the Keycloak docker container:

```
docker compose up -d
```

## Keycloak Configuration

After starting the Keycloak docker container, it's time to set up Keycloak, open the URL <http://localhost:8181> this should open the home page of keycloak, provide admin/admin as the credentials as we have configured it in the docker-compose file.

In Keycloak, all the clients, users, and roles related to a particular application (or) a group of applications reside inside something known as a realm. Realms are independent of each other, so if you create one client/user in one realm, you cannot use it from another realm.

In our project, we will be using the Client Credentials grant, to communicate with the API Gateway and fetch an Access Token, and this access token will be verified by the API Gateway against Keycloak.

To get started, log in to the Keycloak Admin page using the above-mentioned credentials and the first thing we are going to do is to create the realm.

- After logging in, click on the dropdown with the text "Keycloak" in the top left-side corner and click on the Create Realm button

The screenshot shows the Keycloak admin console at [localhost:8181/admin/master/console/](http://localhost:8181/admin/master/console/). The top navigation bar has a dropdown set to "Keycloak" which is expanded to show "master". Below it, there's a "Create realm" button. The main content area is titled "master realm" and contains the "Welcome to Keycloak" message. The sidebar on the left lists various management options like "Realm roles", "Users", "Groups", etc.

- Provide the name of the realm(eg: spring-microservices-realm) and click on the **Create** button
- Now the realm should be created successfully
- Next, click on the **Clients** link to the left sidebar, and click on **Create Client**

The screenshot shows the "Create client" form in the Keycloak admin console. The sidebar is set to "spring-microservices-se...". The "Clients" option is selected. The form itself is titled "Create client" and describes clients as applications and services that can request authentication of a user. It has three steps: 1. General settings (selected), 2. Capability config, and 3. Login settings. In the "General settings" step, the "Client type" is set to "OpenID Connect", "Client ID" is "spring-client-credentials-id", and both "Name" and "Description" fields are empty. A toggle switch for "Always display in UI" is set to "Off". At the bottom are "Next", "Back", and "Cancel" buttons.

- Provide any Client ID you like eg: test-client-id and click on **Next**
- Check Client Authentication as ON
- In the Authentication Flow, select **Service accounts roles** and unselect all other options, this makes sure that our client supports Client Credentials grant

The screenshot shows the Keycloak 'Create client' interface. On the left, a sidebar lists 'Clients' as the active section. The main area is titled 'Create client' with the subtitle 'Clients are applications and services that can request authentication of a user.' A navigation bar at the top includes 'Clients > Create client'. Step 2, 'Capability config', is selected. Under 'Client authentication', the 'On' toggle is selected. In the 'Authentication flow' section, the 'Service accounts roles' checkbox is checked. Other options like 'Standard flow', 'Implicit flow', 'OAuth 2.0 Device Authorization Grant', and 'OIDC CIBA Grant' are available but unchecked. At the bottom, there are 'Next', 'Back', and 'Cancel' buttons.

- Click on **Next** and then **Save**
- Finally, click on the Credentials tab, here you can view the client secret that is generated automatically and you can also regenerate a new client secret. Make sure to copy the client secret, we will use this in the later parts to request a token to access the API Gateway

## Configure Keycloak in API Gateway

Now let's configure keycloak in our api-gateway application, for that we need to add the **spring-boot-starter-oauth2-resource-server** dependency to our pom.xml file

Here is how our pom.xml file now looks like after adding the dependency.

**spring-boot-starter-oauth2-resource-server:** Use this for **securing APIs** when accessing via **Postman** or other REST clients. It validates the OAuth2 access tokens.

**spring-boot-starter-oauth2-login / spring-boot-starter-oauth2-client:** Use this for **OAuth2 login** and **authentication** when accessing through a **web browser**. It handles the authentication flow and provides tokens for user login.

As we are accessing via postman, we are using **spring-boot-starter-oauth2-resource-server**

### pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.4</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.programming.techie</groupId>
  <artifactId>api-gateway</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>api-gateway</name>
  <description>API Gateway</description>
  <properties>
    <java.version>21</java.version>
    <spring-cloud.version>2023.0.1</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-gateway-mvc</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <plugins>
```

```

<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>

</project>

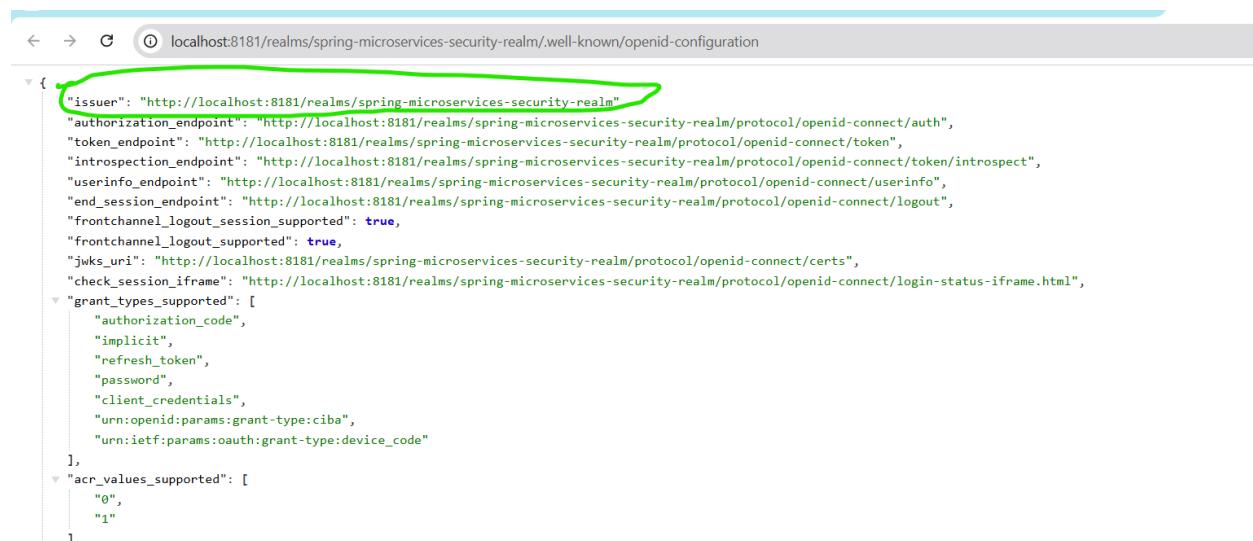
```

Now let's configure the application.properties with the Authorization Server details. For that first, we need to retrieve the Issuer URI of our authorization server. For keycloak, it's usually in the format:

<http://<key-cloak-url>/realm/<realm-name>>

For the realm we created in the previous step, the issuer uri will be:

<http://localhost:8181/realm/spring-microservices-realm>



Now let's go ahead and configure this inside our Spring Boot API Gateway application.

```
spring.security.oauth2.resource-server.jwt.issuer-uri=http://localhost:8181/realm/spring-microservices-realm
```

The next step is to create the Security Configuration class, let's create a package called config and create a class inside the package - SecurityConfig.class

```
package com.programming.techie.gateway.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws
Exception {
        return httpSecurity.authorizeHttpRequests(authorize -> authorize
            .anyRequest().authenticated()
            .oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()))
            .build();
    }
}
```

This is the basic Security Configuration, Spring Security already creates for us out of the box, you can also choose to ignore adding this file if you don't need to add any additional configuration.

Now let's run the ApiGatewayApplication.java class and test out our endpoints using postman.

- Open the Postman client and inside the request window, click on the Authorization tab and then select OAuth2.
- Select the Grant Type as Client Credentials
- Enter the Client ID and Client Secret of the client we created in the previous steps
- Leave the rest of the fields as it is, and click on the Get New Access Token button.

This will allow anyone with access to this request to view and use it.

### Configure New Token

Token Name	Token
Grant type	Client Credentials
Access Token URL ⓘ	http://localhost:8181/realm... spring-client-credentials-id
Client ID ⓘ	spring-client-credentials-id
Client Secret ⓘ	HexxeZfz2hf94eJqw3uZ4PABZDdKE4kE
Scope ⓘ	e.g. read:org
Client Authentication ⓘ	Send as Basic Auth header

Advanced

(i) You can add more specific customizations to your OAuth2 requests here. Learn more about [configuration](#) ↗

- This will make a call to the Token Endpoint and fetches us a new Access Token.
- Click on the Use Token method, to add the token to our Request Context Window.

Auth Type

OAuth 2.0

The authorization data will be automatically generated when you send the request. Learn more about [OAuth 2.0](#) authorization.

Add authorization data to Request Head... ↴

Current Token

(i) Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about [variables](#).

This token is only available to you. Sync the token to let collaborators on this request use it.

Token

Token

eyJhbGciOiJSUzI1NiIsInR5cC1gOjAiSl0Ulw...

Header Prefix ⓘ

Bearer

Auto-refresh Token

Your expired token will be auto-refreshed before sending a request.

- Select any request you want to make for example: call the Product Service Endpoint - GET <HTTP://localhost:9000/api/product> and click on Send

You should receive a successful response from the API Gateway.

The screenshot shows the Postman interface with a successful API call. The request method is POST, and the URL is <http://localhost:9000/api/order>. The Body tab is selected, showing the JSON payload:

```
1 {  
2   "skuCode": "iphone_15",  
3   "price": "1000",  
4   "quantity": "10"  
5 }
```

The response status is 201 Created, with a response time of 78 ms and a response size of 364 B. The response body is "Order Placed Successfully".

In the next part, we will learn how to implement Circuit Breaker Pattern using Resilience4J and Spring Cloud Circuit Breaker Project.