Keycloak OAuth 2 & OpenID with Spring Boot 3 and React Vite — Step-by-Step Guide

Codeching · Follow
8 min read · Jan 31, 2025

2

In this tutorial I'll show you that how you can integrate the Keycloak authentication provider (using Oauth 2.0 and OpenID) with a Spring boot 3 java back-end and a React (Vite) front-end application step by step. We'll use the Spring Security framework on the back-end side to secure our REST API endpoint.

You can view my video tutorial of this topic here as well:

https://www.youtube.com/watch?v=sU-hcMH-EQw

Run and set up Keycloak

First you need to run Keycloak on your local machine somehow. The easiest way is to run the official Keycloak Docker image.

Install docker desktop and create a docker compose file like this one (you can find more information about Keycloak setup here: https://www.keycloak.org/getting-started/getting-started-docker) :

```
#Setting up the keycloak see: https://www.keycloak.org/getting-started/getting-s
version: "3.1"


services:
 keycloak-db:
   image: postgres:11.4
   command:
     ["postgres", "-c", "log_statement=all", "-c", "log_destination=stderr"]
   environment:
     POSTGRES_DB: keycloak
     POSTGRES_USER: keycloak
     POSTGRES_PASSWORD: keycloak
   ports:
     - "14092:5432"
   volumes:
     - keycloak-db:/var/lib/postgresql/data


 keycloak:
   image: quay.io/keycloak/keycloak:26.1.0
   environment:
     KC_DB: postgres
     KC_DB_URL_HOST: keycloak-db
     KC_DB_USERNAME: keycloak
     KC_DB_PASSWORD: keycloak
     KC_DB_URL_PORT: 5432

     KC_BOOTSTRAP_ADMIN_USERNAME: keycloak
     KC_BOOTSTRAP_ADMIN_PASSWORD: keycloak


   command: start-dev
   ports:
     - "14082:8080"
   depends_on:
     - keycloak-db
   restart: unless-stopped


volumes:
 keycloak-db:
```

Run the following command in your terminal where you have your docker compose file:

```
docker compose up
```

```
docker compose up
```

First time this will download the missing docker images and after that it will run the containers. We are using postgres db here to store the Keycloak settings.

Realm

Once Keycloak is up and running, open the http://localhost:14082 and log in to the admin interface using the keycloak — keycloak username and password, then create a new realm (we set this in the docker compose). Name the realm `springboot-test`.

Client

Create a new client named `react-app` with the following settings:

- Root URL: http://localhost:5173
- Home URL: http://localhost:5173
- Valid redirect URIs: http://localhost:5173/*
- Web origins (for testing purposes): *
- Under the Roles tab, add 2 **privileges** (these are roles but I call this privilege): `basic-privilege`, `view-cars`

Realm Role

Create two **realm roles** and assign one privilege/client role to each:

- `basic-role` -> `basic-privilege`
- `can-view-cars` -> `view-cars`

Create User

Create a new user in the user menu and set a password for them and then assign the two realm roles to this new user.

Creating the React front-end application

This client is necessary because it will redirect the user to the login page and forward the received token to our backend application.

```
npm create vite@latest
```

Choose React and TypeScript!

Install dependencies

```
npm i
npm install --save keycloak-js @react-keycloak/web react-router-dom
```

Code

main.tsx

Next, in `main.tsx`, **remove the Strict mode as the Keycloak JS adapter does not like it!**

Lets create keycloak.ts file with this content:

```
import Keycloak from "keycloak-js";

const keycloak = new Keycloak({
  url: "http://localhost:14082/",
```

```
  url: "http://localhost:14082/",
  realm: "springboot-test",
  clientId: "react-app",
});

export default keycloak;
```

App.tsx

```
import { useEffect } from "react";
import "./App.css";
import Cars from "./Cars";
import keycloak from "./keycloak";
import { ReactKeycloakProvider, useKeycloak } from "@react-keycloak/web";

function App() {
  return (
    <ReactKeycloakProvider authClient={keycloak}>
      <SecuredContent />
    </ReactKeycloakProvider>
  );
}
const SecuredContent = () => {
  const { keycloak } = useKeycloak();
  const isLoggedIn = keycloak.authenticated;
  useEffect(() => {
    if (isLoggedIn === false) keycloak?.login();
  }, [isLoggedIn, keycloak]);
  if (!isLoggedIn) return <div>Not logged in</div>;
  return (
    <div>
      <h2>Springboot app using Keycloak authentication provider</h2>
      <Cars />
    </div>
  );
};
export default App;
```

Cars.tsx

Create another file Cars.tsx, and populate it with this code. This is a very simple test application so here we're not using any global state like redux store or any service layer or anything, just a local state and fetch API.

```
import { useKeycloak } from "@react-keycloak/web";
import { useEffect, useState } from "react";

type Car = {
  name: string;
  color: string;
  price: number;
};
const Cars = () => {
  const { keycloak } = useKeycloak();
  const [cars, setCars] = useState<Car[]>([]);
  useEffect(() => {
    const getData = async () => {
      try {
        if (keycloak && keycloak.authenticated) {
          await keycloak?.updateToken();
          const req = await fetch("http://localhost:1291/cars", {
            headers: {
              ["Authorization"]: `Bearer ${keycloak.token}`,
            },
          });
          setCars(await req.json());
        }
      } catch (e) {
        console.log("ERROR", e);
      }
    };
    getData();
  }, [keycloak]);
  return (
    <>
      <div style={{ marginTop: "20px" }}>
        {cars.map((car) => (
          <div key={car.name} style={{ padding: "10px", marginBottom: "20px" }}>
            <span>
              {car.name} - {car.color} | price: {car.price}
            </span>
          </div>
        ))}
      </div>
      <button
        type="button"
        className="text-blue-800"
        onClick={() => keycloak.logout()}
      >
        Logout ({keycloak?.tokenParsed?.preferred_username})
      </button>
    </>
  );
};
export default Cars;
```

As you can see, the `Cars` component already attempts to call the `/cars` endpoint of our Spring Boot application. Let's create that as well!

Spring Boot back-end application

Use Spring Initializr to create a Maven-based application with dependencies for Spring Web and Spring Security.

Create a Basic REST API

We can disable security temporarily like this until we're testing our REST API that we'll create:

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

@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exce
       http
               .authorizeHttpRequests(auth -> auth
                       .anyRequest().permitAll() // Permit all requests
               )
               .csrf(csrf -> csrf.disable()) // Disable CSRF protection for POST
               .formLogin(form -> form.disable()) // Disable login screen
               .httpBasic(basic -> basic.disable()); // Disable HTTP Basic Auth

       return http.build();
   }
}
```

Enable CORS for Local Communication

For the React app to communicate with the Spring Boot application running on a different port locally, enable CORS:

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration corsConfiguration = new CorsConfiguration();
    corsConfiguration.setAllowedOrigins(List.of("http://localhost:5173"));
    corsConfiguration.setAllowedMethods(List.of("GET", "POST"));
    corsConfiguration.setAllowCredentials(true);
    corsConfiguration.setAllowedHeaders(List.of("*"));
    corsConfiguration.setMaxAge(3600L);
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource
    source.registerCorsConfiguration("/**", corsConfiguration);
    return source;
}
```

Modify the filter chain to include CORS, so now we have this SecurityConfig class:

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

@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exce
         http.authorizeHttpRequests(auth -> auth
                   .anyRequest().permitAll() // Permit all requests
             )
             .csrf(csrf -> csrf.disable()) // Disable CSRF protection
             .cors(cors -> cors.configurationSource(corsConfigurationSource()))
             .formLogin(form -> form.disable()) // Disable login screen
             .httpBasic(basic -> basic.disable()); // Disable HTTP Basic Auth

   return http.build();
  }

  @Bean
  public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration corsConfiguration = new CorsConfiguration();
    corsConfiguration.setAllowedOrigins(List.of("http://localhost:5173"));
    corsConfiguration.setAllowedMethods(List.of("GET", "POST"));
    corsConfiguration.setAllowCredentials(true);
    corsConfiguration.setAllowedHeaders(List.of("*"));
    corsConfiguration.setMaxAge(3600L);
```

```
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource
        source.registerCorsConfiguration("/**", corsConfiguration);
        return source;
    }
}
```

Set the Port

Configure the port in `application.properties`:

```
server.port=1291
```

Create the Entity class

We have no JPA or anything like this, so just for testing purpose lets create a simple class.

```
package com.codeching.keycloak.keycloak_example;

public class CarEntity {
   private String name;
   private String color;
   private Long price;

   public CarEntity(String name, String color, Long price) {
       this.name = name;
       this.color = color;
       this.price = price;
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }

   public String getColor() {
       return color;
   }

   public void setColor(String color) {
       this.color = color;
   }

   public Long getPrice() {
       return price;
   }

   public void setPrice(Long price) {
       this.price = price;
   }
}
```

Create the REST Controller

Write the API and test it in the browser. You should see the JSON list of `CarEntity`.

```
package com.codeching.keycloak.keycloak_example;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.Arrays;
import java.util.List;

@RestController
public class MyRestController {

   @RequestMapping("/cars")
   public List<CarEntity> getCars(){
       return Arrays.asList(
               new CarEntity("Audi", "white", 22000000L),
               new CarEntity("BMW", "black", 24500000L),
               new CarEntity("Mercedes", "silver", 45000000L));
   }
}
```

Test the endpoint by calling `http://localhost:1291/cars`.

Setup OpenID with Keycloak in Spring Boot

Once the endpoint is working, set up OpenID 2.0 authentication. The application will require the user to have the `basic-role` realm role and the `view-cars` client privilege for the `/cars` endpoint.

Modify the `SecurityConfig` as follows. The `realm_access` provides the realm roles, and `resource_access` provides the privileges per client.

Here's a sample JWT token:


```
{
  "exp": 1738318021,
  "iat": 1738317721,
  "auth_time": 1738317721,
  "jti": "b629e5b9-6626-4a35-8b68-b1b2c8200539",
  "iss": "http://localhost:14082/realms/springboot-test",
  "aud": "account",
  "sub": "dd1d6639-9028-4fd4-92d4-b0f78b02722d",
  "typ": "Bearer",
  "azp": "react-app",
  "sid": "81076354-ae19-44fb-8065-18fcfe445679",
  "acr": "1",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "can-view-cars",
      "basic-role",
      "offline_access",
      "default-roles-springboot-test",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "react-app": {
      "roles": [
        "view-cars",
        "basic-privilege"
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid profile email",
  "email_verified": false,
  "name": "Laszlo Leber",
  "preferred_username": "stylerhun",
  "given_name": "Laszlo",
  "family_name": "Leber",
  "email": "stylerhun@gmail.com"
}
```


Add the Required Dependency

You will need the following dependency:

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

Update the `SecurityConfig`

Modify the `SecurityConfig` as follows to enable authentication with JWT:

```
package com.codeching.keycloak.keycloak_example;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.oauth2.jwt.Jwt;
import org.springframework.security.oauth2.server.resource.authentication.JwtAut
import org.springframework.security.oauth2.server.resource.authentication.JwtGra
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exc
        http.authorizeHttpRequests(auth -> auth
                requestMatchers("/cars").hasAuthority("ROLE_basic-role")
```

```
                .anyRequest().authenticated()
        )
                .oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> jwt.jwtAuthent
                .csrf(csrf -> csrf.disable())
                .cors(cors -> cors.configurationSource(corsConfigurationSource()

        return http.build();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.setAllowedOrigins(List.of("http://localhost:5173"));
        corsConfiguration.setAllowedMethods(List.of("GET", "POST"));
        corsConfiguration.setAllowCredentials(true);
        corsConfiguration.setAllowedHeaders(List.of("*"));
        corsConfiguration.setMaxAge(3600L);
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSo
        source.registerCorsConfiguration("/**", corsConfiguration);
        return source;
    }

    @Bean
    public JwtAuthenticationConverter jwtAuthenticationConverter() {
        JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthentic
        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(jwt -> {
            Collection<GrantedAuthority> authorities = new ArrayList<>();

            extractRolesFromClaim(jwt, "realm_access", authorities);
            extractRolesFromClaim(jwt, "resource_access", authorities, "react-ap

            return authorities;
        });

        return jwtAuthenticationConverter;
    }

    private void extractRolesFromClaim(Jwt jwt, String claimName, Collection<Gra
        extractRolesFromClaim(jwt, claimName, authorities, null);
    }

    private void extractRolesFromClaim(Jwt jwt, String claimName, Collection<Gra
        Map<String, Object> claim = jwt.getClaim(claimName);
        if (claim != null && claim.containsKey("roles")) {
            List<String> roles = (List<String>) claim.get("roles");
            for (String role : roles) {
                authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
            }
        }

        if (resource != null) {
            Map<String, Object> resourceAccess = jwt.getClaim(claimName);
            if (resourceAccess != null && resourceAccess.containsKey(resource))
                List<String> resourceRoles = (List<String>) ((Map<String, Object
                if (resourceRoles != null) {
                    for (String role : resourceRoles) {
                        authorities.add(new SimpleGrantedAuthority("ROLE_" + rol
                    }
                }
            }
        }
    }
}
```

Update the Controller with Authorization

In `MyRestController.java`, tie the `/cars` endpoint to the `view-cars` privilege:

```
package com.codeching.keycloak.keycloak_example;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;
import java.util.List;

@RestController
public class MyRestController {

   @RequestMapping("/cars")
   @PreAuthorize("hasAuthority('ROLE_view-cars')")
   public List<CarEntity> getCars() {
       return Arrays.asList(
               new CarEntity("Audi", "white", 22000000L),
               new CarEntity("BMW", "black", 24500000L),
               new CarEntity("Mercedes", "silver", 45000000L));
   }
}
```

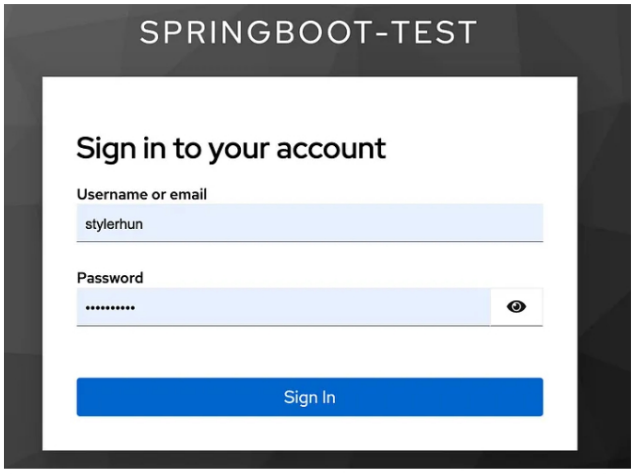
The last thing you should do is to convigure the keycloak URL in your application.properties file:

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:14082/real
```

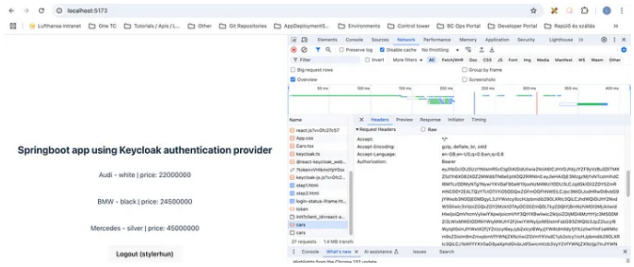
You can now test the `/cars` endpoint with a valid JWT from Keycloak. Run your Spring boot application with the spring-boot:run command. It will start and your front-end is ready to call the http://localhost:1291/cars endpoint.

If you've done everything correctly, then you can see this if you try out your application in your browser (http://localhost:5173):

First you need to log-in:



After log-in you can check your /cars request where you should see your token in the request headers:



And you should get back your repsonse.

Thank you for reading this!

#React #Spring Boot #Keycloak #Oauth 2.0 #Spring Security

Spring Boot | Spring Security | React | Keycloak | Oauth2

2

Written by Codeching

4 Followers · 1 Following

Follow

Codeching is my tech youtube channel with many interesting topics. Check it out: https://www.youtube.com/@codeching Homepage: https://lebersoftware.hu

No responses yet

Write a response

What are your thoughts?

More from Codeching

Codeching

Dockerizing an Angular application with Nodejs Postgres and NginX |...

In this tutorial we are going to create a multicontainer application which will be...

May 30, 2024 6

Codeching

Dockerizing a React application with Nodejs Postgres and NginX |...

In this tutorial we are going to create a multicontainer application which will be...

May 29, 2024

Codeching

Implement server-side rendering (SSR) in React 18 using Pipeable...

In this tutorial I show you that how you can use the SSR (server-side rendering)...

May 30, 2024

Codeching

How to use Docker to dockerize a Spring Boot application, React ap...

In this step by step tutorial we are going to dockerize the Spring Boot Java application...

May 29, 2024

See all from Codeching

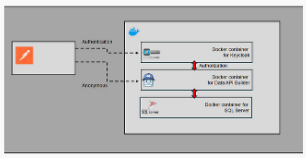
Recommended from Medium

Balian's techologies and innovation lab

Securing Spring Boot Microservices with OAuth2, JWT ...

The Challenge

Apr 15 8



Alessio Tugnoli

API Authentication and Authorization with Keycloak and...

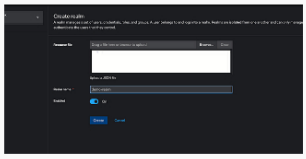
Introduction

Apr 15

In ITNEXT by Ivan Franchin

Keycloak Clustering Made Easy For Development

Exploring the new JDBC-PING default transport stack with Docker Compose for...



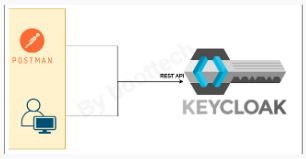
In Keycloak by Abhishek koserwal

Exploring Keycloak 26: Introducing the Organization Feature for Mult...

Introduction

Apr 15 18

Dec 2, 2024 20



Eric Anicet

Manage Keycloak using Admin REST API

This story will explain how to interact with the Keycloak server using REST API without any...

Apr 14 5 2

In hexaDefence by Nishada Liyanage

Keycloak Authorization Code Flow

Beginner's guide for OpenID Connect Authorization Code flow with Keycloak

Nov 13, 2024 5

See more recommendations