Spring Boot Keycloak example

spring boot with keycloak authentication and autotization example code from scratch, give complete code in step by step

Here's a **complete working example** of **Spring Boot with Keycloak Authentication and Authorization** from scratch using **Keycloak 24+ (latest)** with **Spring Boot 3.2+**, and **Jakarta Security**.

✅ Prerequisites

- Java 17+
- Spring Boot 3.2+
- Maven
- Keycloak 24+ running locally (or Docker)
- Keycloak Realm, Client, Roles, and User configured

🔧 Step 1: Setup Keycloak

Start Keycloak using Docker:

```bash
docker run -p 8080:8080 \
-e KEYCLOAK_ADMIN=admin \
-e KEYCLOAK_ADMIN_PASSWORD=admin \
quay.io/keycloak/keycloak:24.0.1 start-dev
```

Create realm, client, roles, and users:

1. **Login**: `http://localhost:8080/admin`
2. **Realm**: Create `springboot-realm`
3. **Client**:
   - Create client ID: `springboot-client`
   - Client Protocol: `openid-connect`

- Access Type: `public`
- Valid Redirect URIs: `http://localhost:8081/*`
- Enable `Standard Flow`

4. **Roles**: Create roles: `USER`, `ADMIN`
5. **Users**:
   - Create user: `john`
   - Set password: `password`
   - Assign role `USER`
   - Create user: `admin`
   - Set password: `adminpass`
   - Assign role `ADMIN`

🧩 Step 2: Create Spring Boot Project

Project Structure

```
css

springboot-keycloak-auth/
├── src/main/java/com/example/demo
│   ├── controller
│   │   └── HelloController.java
│   ├── config
│   │   └── SecurityConfig.java
│   └── DemoApplication.java
├── src/main/resources
│   └── application.yml
└── pom.xml
```

📝 Step 3: `pom.xml`

```
xml

<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.example</groupId>
 <artifactId>springboot-keycloak-auth</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>springboot-keycloak-auth</name>

<properties>
 <java.version>17</java.version>
 <spring.boot.version>3.2.4</spring.boot.version>
</properties>

<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>

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

 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
 </dependency>

 <dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-jose</artifactId>
 </dependency>

 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
 </dependency>

 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
 </dependency>
</dependencies>
```

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

⚙️ Step 4: `application.yml`

yaml

```
server:
  port: 8081

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/springboot-realm
```

🔐 Step 5: Security Config

`SecurityConfig.java`

java

```
package com.example.demo.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.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;
```

```java
@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public").permitAll()
                .requestMatchers("/user").hasRole("USER")
                .requestMatchers("/admin").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2.jwt(Customizer.withDefaults()));
        return http.build();
    }
}
```

👋 Step 6: Create Controller

`HelloController.java`

```java
package com.example.demo.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/public")
    public String publicEndpoint() {
        return "Hello from public endpoint";
    }

    @GetMapping("/user")
    public String userEndpoint() {
        return "Hello USER";
```

```
    }

    @GetMapping("/admin")
    public String adminEndpoint() {
        return "Hello ADMIN";
    }
}
```

🚀 Step 7: Main Class

`DemoApplication.java`

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
  public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

🧪 Step 8: Test With Postman

1. **Get token** from Keycloak:
   - URL: `http://localhost:8080/realms/springboot-realm/protocol/openid-connect/token`
   - Method: `POST`
   - Body: (x-www-form-urlencoded)

```txt
grant_type=password
client_id=springboot-client
```

```
username=john
password=password
```

2. **Use access_token** in Postman:
   - Set `Authorization: Bearer <token>`
   - Try:
     - GET `http://localhost:8081/public` → ✅ no auth
     - GET `http://localhost:8081/user` → ✅ for `john`
     - GET `http://localhost:8081/admin` → ❌ 403 for `john`, ✅ for `admin`

✅ Summary

- Secured endpoints with roles
- Role-based access using `hasRole()`
- Keycloak 24+ with Spring Boot 3.2+
- JWT token validation handled by Spring Security

If you'd like this project in a **GitHub repo or ZIP file**, or want to add **custom claims, user info endpoints, or refresh tokens**, let me know!

SecurityFilterChain check csrf, cors and jwt issuerURi

Sure! Here's how you can **configure** `SecurityFilterChain` in Spring Boot (3.2+) with:

1. ✅ **CSRF disabled**
2. ✅ **CORS enabled**
3. ✅ **JWT validation with Keycloak's** `issuer-uri`

✅ Full `SecurityConfig.java` with `csrf`, `cors`, and `jwt` issuerUri

java

```java
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
```

```java
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
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.List;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // Disable CSRF for API usage
            .cors(Customizer.withDefaults()) // Enable CORS
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public").permitAll()
                .requestMatchers("/user").hasRole("USER")
                .requestMatchers("/admin").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(Customizer.withDefaults()) // JWT validation will use issuer-uri from application.yml
            );

        return http.build();
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();

        configuration.setAllowedOrigins(List.of("*")); // Change to specific domain(s) in prod
        configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
        configuration.setAllowedHeaders(List.of("Authorization", "Cache-Control", "Content-
Type"));
        configuration.setAllowCredentials(true);
```

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

🔐 `application.yml` – Configure JWT `issuer-uri`

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/springboot-realm
```

> ✅ This tells Spring Security to fetch the Keycloak realm config and validate tokens against it.

✅ Notes

- `csrf().disable()` – needed for stateless REST APIs.
- `cors().configurationSource(...)` – enables cross-origin requests for frontend clients.
- `issuer-uri` – used by Spring Security to auto-discover the public JWKS for validating JWTs.

Would you like to enable CORS only for specific origins (e.g., your frontend app)? I can provide a stricter config if needed.

SecurityConfig.java validate issuer-uri and enable csrf and enable cors also

Here's the **correct and complete version of** `SecurityConfig.java` that:

✅ Validates JWT using `issuer-uri` from `application.yml`

✅ Enables **CSRF** protection (for form submissions or browser clients)

✅ Enables and configures **CORS** (for APIs accessed from frontend apps)

✅ `SecurityConfig.java`

java

```java
package com.example.demo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
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.List;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .cors(cors -> cors.configurationSource(corsConfigurationSource())) // ✅ Enable and
configure CORS
            .csrf(csrf -> csrf.enable()) // ✅ CSRF enabled (default in Spring Security)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public").permitAll()
                .requestMatchers("/user").hasRole("USER")
                .requestMatchers("/admin").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt
                    // ✅ Validate JWT using issuer-uri from application.yml
```

```
                .jwkSetUri(null) // Let Spring resolve using issuer-uri
            )
        );

    return http.build();
  }

  @Bean
  public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();

    config.setAllowedOrigins(List.of("http://localhost:3000")); // frontend origin
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
    config.setAllowCredentials(true); // If you send cookies with requests

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

✅ `application.yml` for JWT validation

yaml

```
spring:
 security:
  oauth2:
   resourceserver:
    jwt:
     issuer-uri: http://localhost:8080/realms/springboot-realm
```

🔎 Explanation

Feature	Status	Notes
✅ CORS	Enabled	Allows frontend (e.g., React) to call the API

Feature	Status	Notes
✅ CSRF	Enabled	Useful if using forms or cookie auth (optional for stateless APIs)
✅ JWT Token	Validated using `issuer-uri`	Automatically fetches JWKS from `/.well-known/openid-configuration`

⚠ Important Notes

- If you're using JWT from frontend with **Authorization header**, **CSRF token is not required**.
- If you're building a **pure REST API**, it's **common to disable CSRF** (`csrf().disable()`).
- But if you **keep CSRF enabled**, your frontend must fetch and send CSRF tokens correctly.

Would you like me to add **CSRF token support for a frontend app (like React or Angular)**?

validate issuer-uri add below code

```
@Value("${spring.security.oauth2.resourceserver. jwt.
issuer-uri}"}
String issuerURI;


.oauth2ResourceServer(oauth2 -> oauth2
            .jwt(jwt -> jwt
.decoder(JwtDecoders.fromIssuerLocation(issuerURI)
            )
```

Great — you're looking to **manually configure JWT decoder using** `issuer-uri` from `application.yml` using:

java

```
JwtDecoders.fromIssuerLocation(issuerURI)
```

Below is the **updated and clean version** of `SecurityConfig.java` that:

✅ Uses `@Value` to inject `issuer-uri`
✅ Explicitly sets the JWT decoder using `JwtDecoders.fromIssuerLocation(...)`
✅ Enables **CORS** and **CSRF**

✅ `SecurityConfig.java` (With Custom Issuer Validation)

java

```
package com.example.demo.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtDecoders;
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.List;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    // ✅ Inject issuer URI from application.yml
    @Value("${spring.security.oauth2.resourceserver.jwt.issuer-uri}")
    private String issuerURI;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
```

```
            .cors(cors -> cors.configurationSource(corsConfigurationSource())) // ✅ Enable CORS
            .csrf(csrf -> csrf.enable()) // ✅ Enable CSRF (default behavior)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public").permitAll()
                .requestMatchers("/user").hasRole("USER")
                .requestMatchers("/admin").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt.decoder(jwtDecoder())) // ✅ Custom decoder using issuer URI
            );

        return http.build();
    }

    // ✅ Use JwtDecoders to create a decoder from issuer-uri
    @Bean
    public JwtDecoder jwtDecoder() {
        return JwtDecoders.fromIssuerLocation(issuerURI);
    }

    // ✅ CORS Configuration
    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration config = new CorsConfiguration();

        config.setAllowedOrigins(List.of("http://localhost:3000")); // frontend origin
        config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
        config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
        config.setAllowCredentials(true);

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

✅ `application.yml`

yaml

```
spring:
 security:
  oauth2:
   resourceserver:
    jwt:
     issuer-uri: http://localhost:8080/realms/springboot-realm
```

🧪 Optional Validation

If you're seeing token decoding issues, it usually means:

- Issuer URI is incorrect
- Token does not match realm
- Token is expired or signed with the wrong key

You can log the value by adding:

```java
@PostConstruct
public void logIssuerUri() {
    System.out.println("Using issuer URI: " + issuerURI);
}
```

Would you like to log token claims or roles from the JWT? I can show how to extract roles from the `realm_access` claim in Keycloak.

some times .requestMatchers("/user").hasRole("USER") need to give another role

Yes — in Spring Security, if you want to **allow multiple roles to access the same endpoint**, you can do this using either:

✅ Option 1: Multiple `.hasRole()` with `.orMatchers()` (not ideal)

```java
```

```
.requestMatchers("/user").hasAnyRole("USER", "ADMIN")
```

✅ This is the **cleanest and preferred** way to allow **either USER or ADMIN**.

✅ Example inside `.authorizeHttpRequests()`

java

```
http
  .authorizeHttpRequests(auth -> auth
    .requestMatchers("/public").permitAll()
    .requestMatchers("/user").hasAnyRole("USER", "ADMIN") // 👉 Allow both roles
    .requestMatchers("/admin").hasRole("ADMIN")
    .anyRequest().authenticated()
  )
```

✅ Option 2: Use @PreAuthorize on controller methods

If you want more flexibility, use method-level annotations with `@PreAuthorize`.

java

```
@GetMapping("/user")
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
public String userEndpoint() {
  return "Hello USER or ADMIN";
}
```

> 🛡️ Make sure to enable method security:

java

```
@EnableMethodSecurity
```

🚀 Bonus: Use expression for more control

java

```
@PreAuthorize("hasAnyRole('USER','ADMIN') and hasAuthority('SOME_PERMISSION')")
```

Let me know if you want to build role-based routing with custom claims like `realm_access`, or map roles from token manually!

then give detailed full configuraton for .requestMatchers

Here's a complete and **detailed Spring Security configuration** using `requestMatchers(...)` for **multiple endpoints and multiple roles**, supporting:

- ✅ `public` – No auth required
- ✅ `/user` – Accessible by `ROLE_USER`, `ROLE_ADMIN`
- ✅ `/admin` – Only by `ROLE_ADMIN`
- ✅ `/manager` – Only by `ROLE_MANAGER`
- ✅ All other endpoints – Authenticated users only
- ✅ CORS enabled
- ✅ CSRF enabled
- ✅ JWT validation using `issuer-uri`

✅ `SecurityConfig.java` (Full config with `.requestMatchers`)

java

```
package com.example.demo.config;

import jakarta.annotation.PostConstruct;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtDecoders;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.cors.CorsConfiguration;
```

```java
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.List;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {

    @Value("${spring.security.oauth2.resourceserver.jwt.issuer-uri}")
    private String issuerURI;

    @PostConstruct
    public void logIssuerUri() {
        System.out.println("Using issuer URI: " + issuerURI);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .cors(cors -> cors.configurationSource(corsConfigurationSource())) // ✅ Enable CORS
            .csrf(csrf -> csrf.enable()) // ✅ Enable CSRF
            .authorizeHttpRequests(auth -> auth
                // Public endpoint – no authentication required
                .requestMatchers("/public", "/actuator/**").permitAll()

                // /user endpoint – accessible by USER or ADMIN roles
                .requestMatchers("/user").hasAnyRole("USER", "ADMIN")

                // /admin endpoint – only ADMIN role
                .requestMatchers("/admin").hasRole("ADMIN")

                // /manager endpoint – only MANAGER role
                .requestMatchers("/manager").hasRole("MANAGER")

                // All other requests – require authentication
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(jwt -> jwt.decoder(jwtDecoder())) // ✅ Use custom decoder with issuer-uri
            );
```

```
        return http.build();
    }

    // ✅ Configure JWT decoder with issuer-uri
    @Bean
    public JwtDecoder jwtDecoder() {
        return JwtDecoders.fromIssuerLocation(issuerURI);
    }

    // ✅ CORS configuration for cross-origin frontend apps
    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(List.of("http://localhost:3000")); // frontend origin
        config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
        config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
        config.setAllowCredentials(true); // send cookies if needed

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

✅ `application.yml`

yaml

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/springboot-realm
```

✅ Example Controller

java

```java
@RestController
public class RoleBasedController {

    @GetMapping("/public")
    public String publicEndpoint() {
        return "Public access";
    }

    @GetMapping("/user")
    public String userEndpoint() {
        return "Accessible by USER or ADMIN";
    }

    @GetMapping("/admin")
    public String adminEndpoint() {
        return "Accessible by ADMIN only";
    }

    @GetMapping("/manager")
    public String managerEndpoint() {
        return "Accessible by MANAGER only";
    }
}
```

✅ Key Points

Endpoint	Access Roles
`/public`	Anyone (no login)
`/user`	`ROLE_USER`, `ROLE_ADMIN`
`/admin`	`ROLE_ADMIN`
`/manager`	`ROLE_MANAGER`
`/**` others	Any authenticated

> Note: Spring automatically prefixes roles from JWT with `ROLE_`. So `hasRole("USER")` maps to claim `realm_access.roles: ["USER"]`.

Optional: Extract roles from a different claim?

If your roles are not in `realm_access.roles`, let me know — I can show how to use a **custom** `JwtAuthenticationConverter`.

Would you like to include that too?