A Spring Boot service can use Keycloak for user authentication while managing roles within the Spring Boot application itself. This setup involves delegating the authentication process to Keycloak and handling role-based authorization logic in the application.

**Steps to Achieve This:**

1. **Use Keycloak for Authentication**:
   * Configure your Spring Boot application as a client in Keycloak.
   * Use Keycloak to issue JSON Web Tokens (JWTs) upon successful authentication.
2. **Configure Spring Security**:
   * Integrate Spring Security with Keycloak by adding the required dependencies (spring-boot-starter-security, spring-security-oauth2-resource-server, and keycloak-spring-boot-starter).
   * Configure Keycloak's public key for JWT validation in your application (spring.security.oauth2.resourceserver.jwt.issuer-uri or Keycloak adapter configuration).
3. **Extract User Information**:
   * After authentication, extract the user information (e.g., username or email) from the Keycloak-issued JWT.
4. **Manage Roles in Spring Boot**:
   * Create a database table or other storage mechanism to manage roles within your Spring Boot app.
   * Map roles to users based on the user information obtained from the JWT.
   * Implement a custom UserDetailsService or role management service to fetch user roles.
5. **Apply Authorization Logic**:
   * Use Spring Security annotations (@PreAuthorize, @RolesAllowed, etc.) to enforce role-based access control.
   * Optionally, implement a GrantedAuthoritiesMapper to map roles stored in your application to Spring Security GrantedAuthority objects.
6. **Optional: Sync Roles with Keycloak**:
   * If required, you can sync Keycloak roles with your application roles. For example, Keycloak roles can be fetched from the JWT and mapped to additional roles in your application.

**Sample Configuration:**

**application.properties:**

spring.security.oauth2.resourceserver.jwt.issuer-uri=https://<your-keycloak-domain>/realms/<realm-name>

spring.security.oauth2.resourceserver.jwt.jwk-set-uri=https://<your-keycloak-domain>/realms/<realm-name>/protocol/openid-connect/certs

**Security Configuration:**

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/public/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.oauth2ResourceServer().jwt();

}

}

**Custom Role Management:**

@Service

public class RoleService {

public Set<String> getUserRoles(String username) {

// Fetch roles from your database or another source based on the username

return Set.of("ROLE\_USER", "ROLE\_ADMIN");

}

}

@Component

public class CustomJwtAuthenticationConverter implements Converter<Jwt, AbstractAuthenticationToken> {

@Autowired

private RoleService roleService;

@Override

public AbstractAuthenticationToken convert(Jwt jwt) {

String username = jwt.getSubject();

Set<String> roles = roleService.getUserRoles(username);

List<GrantedAuthority> authorities = roles.stream()

.map(SimpleGrantedAuthority::new)

.collect(Collectors.toList());

return new JwtAuthenticationToken(jwt, authorities);

}

}

**Enable Custom Converter:**

@Configuration

public class JwtConfig {

@Bean

public JwtAuthenticationConverter jwtAuthenticationConverter(CustomJwtAuthenticationConverter customConverter) {

JwtAuthenticationConverter converter = new JwtAuthenticationConverter();

converter.setJwtGrantedAuthoritiesConverter(customConverter);

return converter;

}

}

**Key Benefits:**

* **Authentication offloading**: Keycloak handles user authentication securely.
* **Flexible role management**: The application can maintain its own role structure without relying on Keycloak's roles.
* **Separation of concerns**: Authentication and authorization responsibilities are decoupled.

This setup is particularly useful when the application's roles and permissions model is too complex to manage in Keycloak directly.