Securing Java Applications with OAuth 2.0 and OpenID Connect (OIDC)

In today's digital landscape, identity federation is crucial for authenticating users across multiple applications within an organization. Two key protocols in identity federation—OAuth 2.0 and OpenID Connect (OIDC)—provide a framework for secure and seamless access. This article demystifies these protocols, explains their differences, and outlines their applications in Java environments using tools like Spring Security and Keycloak.

1. Introduction to OAuth 2.0: The Authorization Standard

OAuth 2.0 is an authorization framework that allows third-party applications to access a user's resources without exposing their credentials. Through the use of access tokens, OAuth 2.0 establishes a secure, token-based access control mechanism. It's widely used for securing APIs and enables granular permissions management while safeguarding user privacy.

Key OAuth 2.0 Concepts:

- ***Access Tokens:*** Tokens that authorize limited access to resources, ensuring that sensitive information is protected.
- ***Scopes*:** Define what permissions the access token grants, allowing granular control.
- ***Authorization Grant Types:*** OAuth 2.0 defines multiple flows—such as Authorization Code and Client Credentials—that cater to different scenarios (e.g., web applications, machine-to-machine communication).

2. Understanding OpenID Connect: Authentication Layer Over OAuth 2.0

OIDC is an identity layer built on top of OAuth 2.0, designed to provide secure user authentication. While OAuth 2.0 primarily handles authorization, OIDC adds authentication capabilities, allowing applications to verify user identity through ID tokens and access their profile information.

OIDC Essentials:

- ***ID Tokens:*** Tokens containing claims about the user's identity, issued by the identity provider (IdP).
- ***UserInfo Endpoint:*** Allows the application to retrieve additional user profile information.

- ***OIDC Flows:*** Inherits OAuth 2.0's authorization flows, with added capabilities for user identity verification, such as Authorization Code Flow with PKCE.

3. Key Benefits of OAuth 2.0 and OIDC

These protocols offer several advantages in securing Java applications:

- ***Enhanced Security:*** Protect user data and credentials, reducing the risk of breaches.
- ***Single Sign-On (SSO):*** Simplify user access across multiple applications with centralized authentication.
- ***Interoperability***: Supports a wide range of identity providers (IdPs), including Google, Facebook, and Keycloak.
- **Scalability:** Ideal for web applications, APIs, and microservices.

4. Integrating OAuth 2.0 and OIDC in Java with Spring Security

Spring Security provides a robust framework for implementing OAuth 2.0 and OIDC in Java applications. Together with tools like Keycloak, this integration becomes straightforward, enabling developers to secure applications effectively.

Setting Up OAuth 2.0 and OIDC in Spring Security

- ***Dependency Configuration:*** Add the necessary dependencies to your project's **pom.xml** or **build.gradle**.
- ***Authorization Configuration:*** Configure WebSecurityConfigurerAdapter to set up client registration, security filters, and token management.
- ***Keycloak as an IdP:*** Keycloak is a popular open-source identity provider that supports both OAuth 2.0 and OIDC, providing a convenient interface for managing authentication and authorization.

Example Configuration with Spring Security and Keycloak:

1. Add Dependencies: Include the Keycloak and Spring Security dependencies in your pom.xml file.

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

2. Configure Keycloak Properties: In your application.yml or application.properties, configure Keycloak settings to connect with your Keycloak server.

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          keycloak:
            client-id: your-client-id
            client-secret: your-client-secret
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
        provider:
          keycloak:
            issuer-uri: http://localhost:8080/realms/your-realm
            user-info-uri: http://localhost:8080/realms/your-realm/protocol/openid-connect/userinfo
            token-uri: http://localhost:8080/realms/your-realm/protocol/openid-connect/token
            authorization-uri: http://localhost:8080/realms/your-realm/protocol/openid-connect/auth
```

3. Define Security Configuration: Configuring Keycloak as the OAuth 2.0 provider by putting the *SecurityFilterChain Bean*

```java
@Bean
SecurityFilterChain clientSecurityFilterChain(
      HttpSecurity http,
      ClientRegistrationRepository clientRegistrationRepository) throws Exception {
  http.oauth2Login(Customizer.withDefaults());
  http.logout((logout) -> {
    var logoutSuccessHandler =
        new OidcClientInitiatedLogoutSuccessHandler(clientRegistrationRepository);
    logoutSuccessHandler.setPostLogoutRedirectUri("{baseUrl}/");
    logout.logoutSuccessHandler(logoutSuccessHandler);
  });

  http.authorizeHttpRequests(requests -> {
    requests.requestMatchers("/", "/favicon.ico").permitAll();
    requests.requestMatchers("/nice").hasAuthority("NICE");
    requests.anyRequest().denyAll();
  });

  return http.build();
}
```

4. Role-Based Authorization (Optional): If your application uses role-based access, configure Keycloak roles within your SecurityConfig.

```java
interface AuthoritiesConverter extends Converter<Map<String, Object>,
Collection<GrantedAuthority>> {}

@Bean
AuthoritiesConverter realmRolesAuthoritiesConverter() {
  return claims -> {
    var realmAccess = Optional.ofNullable((Map<String, Object>)
claims.get("realm_access"));
    var roles = realmAccess.flatMap(map -> Optional.ofNullable((List<String>)
map.get("roles")));
    return roles.map(List::stream)
      .orElse(Stream.empty())
      .map(SimpleGrantedAuthority::new)
      .map(GrantedAuthority.class::cast)
      .toList();
  };
}
```

Because of generics type erasure in the JVM and because there could be many Converter beans with different inputs and outputs in an application context, this ***AuthoritiesConverter interface*** *can be a useful*

*tip for the bean factory when it searches for a **Converter<Map<String, Object>, Collection<GrantedAuthority>>** bean.*

```
@Bean
GrantedAuthoritiesMapper authenticationConverter(
  Converter<Map<String, Object>, Collection<GrantedAuthority>> authoritiesConverter) {
    return (authorities) -> authorities.stream()
      .filter(authority -> authority instanceof OidcUserAuthority)
      .map(OidcUserAuthority.class::cast)
      .map(OidcUserAuthority::getIdToken)
      .map(OidcIdToken::getClaims)
      .map(authoritiesConverter::convert)
      .flatMap(roles -> roles.stream())
      .collect(Collectors.toSet());
}
```

This setup enables authentication and role-based authorization through Keycloak using OAuth 2.0 and OIDC in Spring Boot.

OAuth 2.0 offers several authorization flows to meet various requirements:

1. ***Authorization Code Flow:*** Ideal for web applications, where the user authorizes the application to access their resources on a server.

2. ***Implicit Flow:*** Primarily used in single-page applications (SPAs), but is being phased out due to security concerns.

3. ***Client Credentials Flow:*** Best suited for machine-to-machine interactions, allowing a service to access resources on behalf of itself.

4. ***Resource Owner Password Credentials Flow:*** Deprecated due to security risks; only used in trusted environments.

6. Leveraging JWT for Secure Session Management

Both OAuth 2.0 and OIDC commonly use JSON Web Tokens (JWT) to manage sessions securely. JWTs provide a self-contained way to transmit claims between parties, enhancing security and reducing server-side storage requirements.

Key Benefits of Using JWT with OAuth 2.0 and OIDC:

- ***Stateless***: JWTs are stateless, allowing for distributed systems and microservices to handle authentication efficiently.
- ***Signed and Encrypted:*** JWTs can be signed (for authenticity) and encrypted (for confidentiality).
- ***Role-Based Access Control:*** Encodes user roles and permissions within the token, enabling fine-grained access control.

Example JWT Payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "roles": ["USER", "ADMIN"],
  "exp": 1678541200
}
```

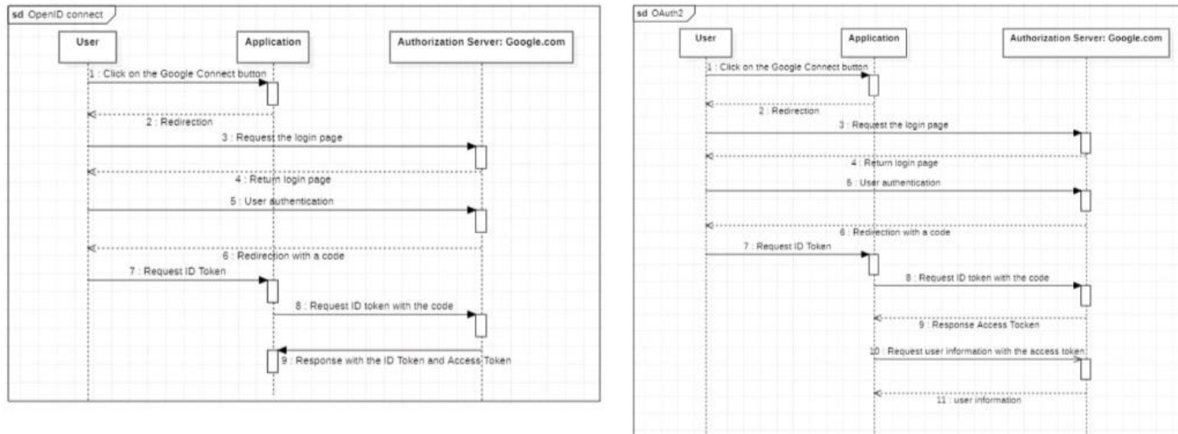
7. OAuth 2.0 vs. OIDC: Key Differences

While both OAuth 2.0 and OIDC provide secure access, they have different focuses:

- ***OAuth 2.0:*** Manages authorization, enabling applications to access resources on behalf of a user.
- ***OIDC:*** Adds authentication, allowing applications to verify a user's identity and access profile information.

When to Use OAuth 2.0 vs. OIDC

- ***OAuth 2.0:*** Use for cases where access control is required without in-depth user authentication.
- ***OIDC***: Use when verifying a user's identity and accessing profile data are essential.



---

8. Practical Use Cases for OAuth 2.0 and OIDC

These protocols are essential in applications where secure, federated identity is required. Common use cases include:

- ***Single Sign-On (SSO):*** Centralized access for multiple applications.
- ***API Security:*** Secure APIs by restricting access based on user roles and permissions.
- ***Cross-Platform Authentication:*** Allow users to log in using their identity on platforms like Google and Facebook.

9. Conclusion

OAuth 2.0 and OIDC are foundational for modern authentication and authorization, offering secure access to resources and applications. In Java, leveraging Spring Security and tools like Keycloak simplifies implementing these protocols, enabling efficient and secure identity management. By adopting OAuth 2.0 and OIDC, developers can enhance security, ensure scalability, and deliver a seamless user experience.