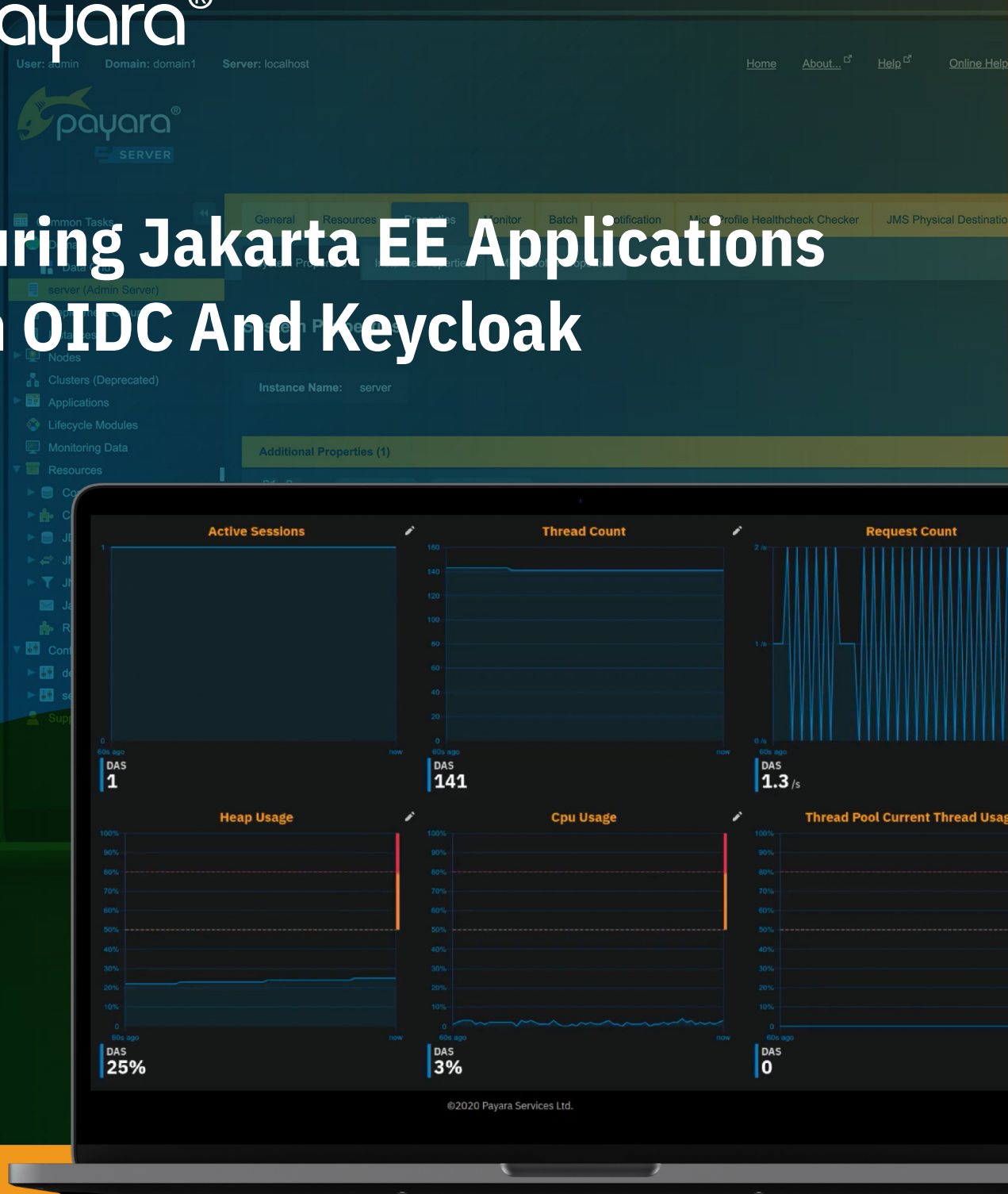




Securing Jakarta EE Applications With OIDC And Keycloak



The Payara® Platform - Production-Ready,
Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **October 2023**

OpenId and OpenID Connect	1
Advantages of OIDC	2
Simplified Authentication	2
Standardised Protocol	2
Enhanced Security	2
Single Sign-On (SSO)	3
Rich Identity Information	3
Identity Providers	3
OpenId Auth	4
OIDC Token Types	4
Grant Types	5
Keycloak As An Identity Provider	5
OIDC Auth With Keycloak	6
Implementing OIDC With Jakarta Security	7
Annotation Driven Configuration	8
@RolesAllowed	11
Using Web.xml	12
Getting User Authenticated Info	12
Summary	14

Security is one of the most important components of every application. More so in an era where almost all applications are delivered across the web. Ensuring that an application is secure for legitimate users and out of reach of bad actors is a very complex exercise. The constant stream of new malicious attackers seeking ways to compromise applications for different gains requires application developers and architects to be a step ahead at all times.

The need to simplify, harmonise and centralise security as much as possible led the industry to come up with the OpenId Authentication Protocol. OpenID is an open standard and decentralised authentication protocol, designed to make it easy for users to log in to different websites, webapps and online services using a single identity. This single identity eliminates the need for creating and remembering multiple usernames and passwords for different online platforms. Instead, users can log in using an identity provider that they trust, such as a social media account or an email service provider.

Traditionally on the Jakarta EE Platform, security had been implemented through disparate APIs that made it hard to implement robust, complete and maintainable security without resorting to third party libraries. The release of Jakarta EE 10 with the Jakarta Security 3.0 specification, ushered in an era of seamless and unified security implementation on the platform. This guide looks at securing Payara Micro Jakarta EE deployed applications using the new Jakarta Security API and its OpenID Connect capability.

We start by taking a brief look at what OpenID is, a look at Keycloak, a popular, open source identity provider, and finally how to secure a typical Jakarta EE application using Jakarta Security and Keycloak as an identity provider. By the end of this guide, you will have a foundational overview of how to start implementing security in your Jakarta EE applications.

OpenId and OpenID Connect

OpenID and OpenID Connect (OIDC) are protocols at the heart of modern web authentication, providing streamlined mechanisms for user identification across various services and platforms. Despite the similarity in their names and shared objective of enhancing authentication, they differ in their approaches and technology stacks. Understanding these protocols is essential in adopting them for your application security.

OpenID emerged as an open standard for authentication, enabling users to log in to multiple unrelated websites using a single digital identity, without each site having to own and manage its own identity system. The protocol allowed users to create an account with their preferred OpenID provider (such as Google or Yahoo) and use that account to sign in to any website that accepts OpenID authentication. Despite its innovative approach, OpenID had its set of limitations and did not see widespread adoption, partly due to its complexity and the rapidly evolving needs of web security.

Recognizing the limitations of OpenID and the advent of OAuth 2.0 - a protocol designed for authorization, the community developed OpenID Connect, shortened to OIDC. OIDC is a simple identity layer built on top of the OAuth 2.0 protocol. While OAuth 2.0 handles authorization, allowing third-party applications to access user data without sharing passwords, OIDC provides authentication services, ensuring that users are who they claim to be. In essence, OIDC is a modernised, simplified version of OpenID, retaining the latter's goals while leveraging the strengths of OAuth 2.0 for better security and simplicity.

Advantages of OIDC

Simplified Authentication

OpenID Connect (OIDC) streamlines the authentication process by employing a straightforward flow, which significantly reduces the development complexity traditionally associated with securing applications. By utilising modern web technologies and providing well-defined protocols, OIDC minimises the amount of custom code you need to write for handling authentication. This simplicity accelerates the development process, allowing for quicker deployments while ensuring a secure and reliable authentication mechanism.

Standardised Protocol

Being a standardised protocol, OIDC fosters interoperability across different systems and services. This standardisation makes it easier for you to integrate authentication services in a consistent and predictable manner, regardless of the underlying platforms or technologies being used. The commonality of the protocol also means that you can leverage a wide range of libraries and tools designed to work with OIDC, further simplifying the integration process and promoting best practices across the industry.

Enhanced Security

OIDC enhances security by building upon the robustness of OAuth 2.0, incorporating advanced security mechanisms like token-based authentication. This approach is a significant improvement over older methods, such as cookie-based sessions. Token-based authentication is more secure as tokens are self-contained, can be expired, and revoked easily, reducing the risk associated with token compromise. Additionally, OIDC supports various flows to cater to different use cases, each designed with a focus on securing user credentials and the applications utilising them. This makes it ideal for securing applications that employ different paradigms.

Single Sign-On (SSO)

One of the notable advantages of OIDC is its support for Single Sign-On (SSO), a feature that permits users to log in once and gain access to multiple applications without needing to authenticate again. SSO enhances user experience by providing a seamless transition between services while reducing the authentication overhead for both users and you as a developer. This not only simplifies the user's interaction with various services but also lowers the operational and maintenance costs associated with managing multiple authentication systems.

Rich Identity Information

Unlike OAuth 2.0, which is primarily an authorization protocol, OIDC provides rich identity information about the authenticated user through a standardised RESTful API known as the UserInfo Endpoint. This feature enables you to easily obtain user details such as name, email, and other attributes once authentication is successful, without requiring additional API calls to the identity provider. The availability of identity information is crucial for personalising user experiences and for making informed authorization decisions within applications. By providing this information in a standardised and easily accessible manner, OIDC greatly simplifies the task of managing user identities and authorization in modern applications.

Identity Providers

Identity Providers, often abbreviated as IdPs, play a crucial role in the realm of OpenID and OpenID Connect by serving as the backbone for user authentication and authorization. They are specialised services or servers tasked with managing user identities and providing authentication services to dependent applications, ensuring that only authorised individuals can access certain resources.

In the context of OpenID, an Identity Provider is responsible for verifying the user's identity and providing the relying party (the application needing authentication) with an assertion that proves the user's identity. This system decentralises the authentication mechanism, freeing the applications from handling passwords or other sensitive information, and placing the onus of secure authentication on the IdP.

OpenID Connect (OIDC), being an extension of OpenID, inherits this model but enriches it further. OIDC Identity Providers not only authenticate users but also provide additional information about the authenticated user, known as claims, in a standardised manner. This is achieved through a token-based system where, post-authentication, the IdP issues tokens that the application can use to obtain the user's information in a secure manner.

Utilising IdPs allows you to offload the significant burden of implementing and maintaining authentication systems, which are often complex and critical to the security posture of your applications. This frees you up to focus more on the core functionality of your applications, while also providing a better, and often more secure, user experience, since IdPs are implemented by people highly specialised in security.

Furthermore, the use of standard protocols like OpenID Connect ensures that you can easily switch between different Identity Providers if needed, without having to overhaul the authentication system within your applications. This fosters a more flexible and interoperable ecosystem where applications and services can securely interact with each other regardless of the underlying Identity Provider being used.

OpenId Auth

OpenId Connect authorization is about obtaining the necessary permissions to access protected resources and verify user identities. Authentication and authorization information in OIDC are exchanged through tokens.

OIDC Token Types

Tokens are the fundamental pieces of data used to grant access and convey authorization information between parties. There are a number of different token types within the OIDC auth workflow. Each token has a different role it plays in exchanging information between a user, an IdP and a dependent application or client. These are as follows.

- **Access Token** - An access token is a string representing an authorization issued to the client. It signifies the permissions and scope granted to the client by the user. The client can use this token to make API requests on behalf of the user.
- **Identity Token (JWT)** - Identity Token is a JSON Web Token (JWT) that contains claims about the identity of the authenticated user. It can be decoded and verified by the client to ensure that the user is who they claim to be.
- **Refresh Token** - A refresh token is used to obtain new access or identity tokens without requiring the user to authenticate again. It's helpful to maintain the session alive for a longer period, improving the user experience.

Grant Types

An OpenID Connect (OIDC) grant type is a mechanism that allows an application to obtain an access token and ID token from an OIDC provider. Access tokens are used to access protected resources, while ID tokens are used to verify the user's identity.

There are four standard OIDC grant types:

- **Authorization code grant type** - This is the most secure grant type and is the recommended grant type for most applications. In this grant type, the user is redirected to the OIDC provider's login page to authenticate. After the user has authenticated, the OIDC provider redirects the user back to the application with an authorization code. The application can then use the authorization code to obtain an access token and ID token from the OIDC provider.
- **Implicit grant type** - This grant type is less secure than the authorization code grant type, but it is simpler to implement. In this grant type, the application is redirected to the OIDC provider's login page to authenticate the user. After the user has authenticated, the OIDC provider redirects the user back to the application with an access token and ID token.
- **Client credentials grant type** - This grant type is used by applications to obtain an access token on their own behalf. In this grant type, the application provides its client ID and client secret to the OIDC provider. The OIDC provider then uses this information to verify the application's identity and issue an access token.
- **Device code grant type** - This grant type is used for devices that cannot easily display a web browser, such as smart TVs and IoT devices. In this grant type, the user visits a website on their device and enters a device code. The application then uses the device code to obtain an access token and ID token from the OIDC provider.

Which OIDC grant type you should use depends on the specific needs of your application. If you are developing a web application, the authorization code grant type is the recommended grant type. If you are developing a mobile application, you may want to consider using the implicit grant type. And if you are developing an application that needs to obtain an access token on its own behalf, you will need to use the client credentials grant type.

Keycloak As An Identity Provider

Keycloak is a popular Identity Provider (IdP) that offers a comprehensive suite of modern authentication and authorization features. It's an open-source project maintained by Red Hat, which provides an elegant solution for securing modern applications without requiring extensive setup or a deep understanding of security protocols. As an IdP, Keycloak supports both OpenID and OpenID Connect (OIDC), among other standard protocols like SAML 2.0 and OAuth 2.0. This makes it a flexible choice for integrating a robust authentication and authorization system into applications.

One of the strengths of Keycloak is its ease of use. It provides a user-friendly admin console that allows developers and administrators to configure realms, users, roles, and clients, making it easier to manage the security aspects of an application. This is particularly beneficial for developers who may not have extensive expertise in security, allowing them to leverage Keycloak's features to secure their applications effectively.

Keycloak also provides a variety of social login capabilities, multi-factor authentication, and user federation, enabling it to serve as a bridge between various user identity sources and applications. This functionality is crucial for developers working in environments with diverse identity sources or those looking to provide a seamless user experience across multiple platforms.

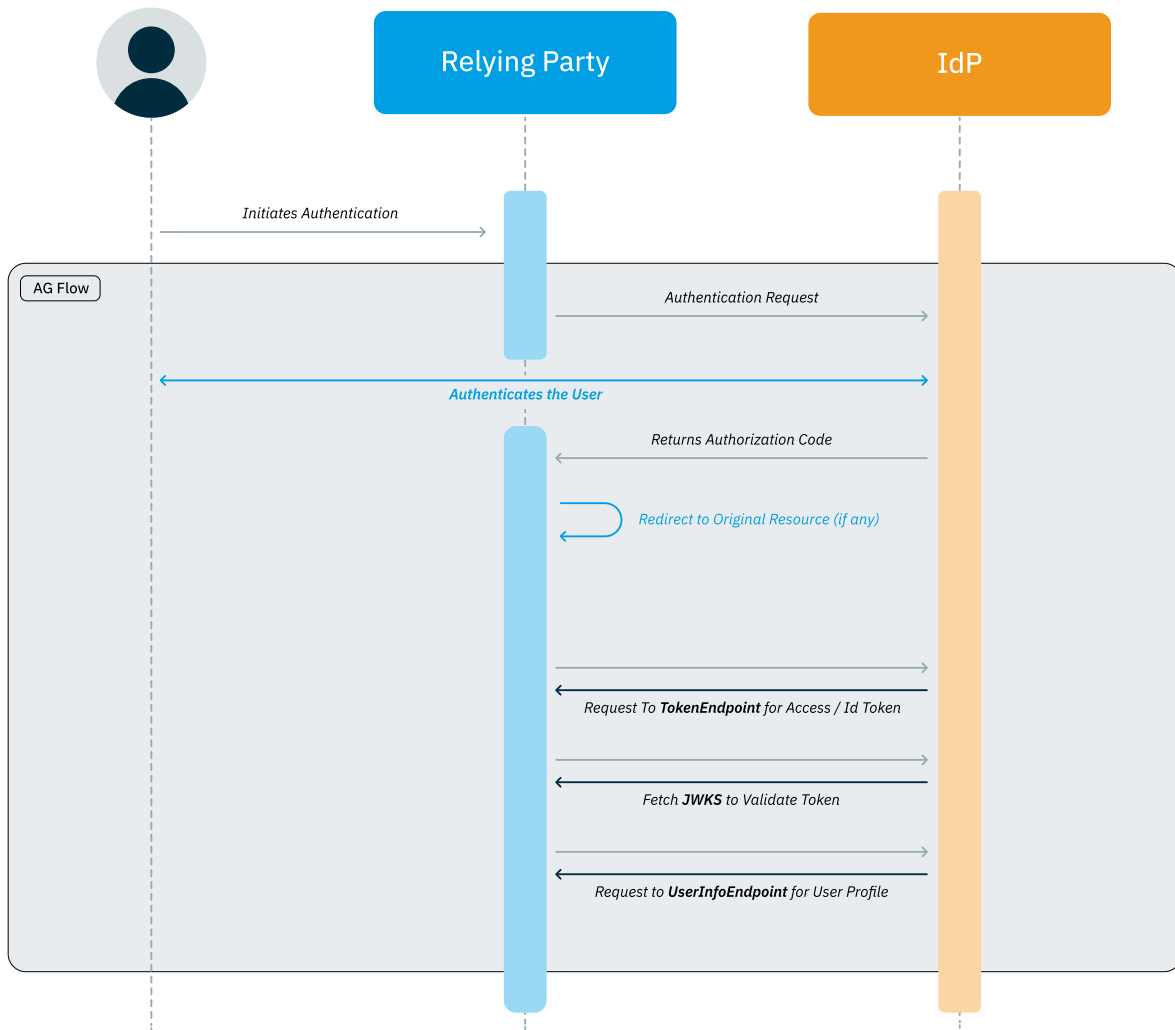
Additionally, Keycloak's support for single sign-on (SSO) means that users can authenticate once and gain access to multiple applications, enhancing user convenience and reducing the authentication overhead for developers.

Moreover, being an open-source project, Keycloak has a vibrant community that contributes to its continuous improvement and offers support through community forums and other platforms. This community-driven nature ensures that Keycloak stays updated with the latest security standards and features, making it a reliable choice for developers seeking a robust, modern identity provider for their applications. As an IdP, Keycloak is a great choice if you are looking for a full suite IdP that can get you up to securing your applications as fast as possible.

OIDC Auth With Keycloak

The typical authentication workflow using OIDC and Keycloak as an IdP is as follows:

1. The user visits your Jakarta EE application that supports OIDC and clicks on the login button. This initiates the OIDC authentication workflow.
2. The user is redirected to Keycloak's login page. Here, the user can enter their credentials and authenticate with Keycloak. The authentication could be username/password, or any configured third-party IdP like Google or Facebook.
3. Once the user has authenticated, Keycloak will redirect the user back to your web application with an authorization code.
4. Your application will then exchange the authorization code for an access token and ID token from Keycloak.
5. The application will then use the access token to access protected resources on behalf of the user. For example, using the access token to call a REST API that is protected by Keycloak.
6. The application can also use the ID token to verify the user's identity. For example, extract the user's name and email address from the ID token.



Implementing OIDC With Jakarta Security

The Jakarta Security 3.0 specification released with Jakarta EE 10 has built-in support for OIDC. This gives you a standardised, portable way of securing Jakarta EE applications using this single API. In the next sections, we take a look at how to secure a Jakarta EE endpoint with OIDC using the Jakarta Security API.

Annotation Driven Configuration

The first step to OIDC authentication and authorization is to configure your Jakarta EE application as such. The root configuration annotation from the Jakarta Security API is the `@OpenIdAuthenticationMechanismDefinition()`. A typical configuration is as follows.

```
@OpenIdAuthenticationMechanismDefinition(  
    clientId = "${oidConfig.clientID}",  
    clientSecret = "${oidConfig.clientSecret}",  
    redirectURI = "${baseURL}/index.xhtml",  
    scope = {"openid", "email", "profile", "offline_access"},  
    prompt = PromptType.LOGIN,  
    providerURI = "${oidConfig.providerUri}",  
    jwksReadTimeout = 10_000,  
    jwksConnectTimeout = 10_000,  
    claimsDefinition = @ClaimsDefinition(callerGroupsClaim = "${oidConfig.  
callerGroupsClaim}"),  
    extraParameters = "audience=https://api.payara.fish/",  
    logout = @LogoutDefinition(redirectURI = "${baseURL}/index.xhtml")  
)  
@Named("oidConfig")  
@ApplicationScoped  
public class OpenIDSecurityConfiguration {  
  
    private static final String ROLES_CLAIM = "/roles";  
  
    @Inject  
    @ConfigProperty(name = "fish.payara.demos.conference.security.openid.  
clientId")  
    private String clientID;  
  
    @Inject  
    @ConfigProperty(name = "fish.payara.demos.conference.security.openid.  
clientSecret")  
    private String clientSecret;  
  
    @Inject  
    @ConfigProperty(name = "fish.payara.demos.conference.security.openid.  
provider.uri")  
    private String providerUri;
```

```
@Inject
@ConfigProperty(name = "fish.payara.demos.conference.security.openid.
custom.namespace")
private String customNamespace;

public String getClientID() {
    return clientID;
}

public String getClientSecret() {
    return clientSecret;
}

public String getProviderUri() {
    return providerUri;
}

public String getCallerGroupsClaim() {
    return customNamespace + ROLES_CLAIM;
}
}
```

The `@OpenIdAuthenticationMechanismDefinition` annotation defines the OpenID Connect authentication mechanism configuration. Within it, we pass various parameters to configure the authentication. The first thing to note is the use of the Jakarta Expression Language interpolation string. Using `${}` allows us to externalise configuration of the various parameters. In this example, the `OpenIDSecurityConfiguration` class itself has `MicroProfile Config` fields into which the externalised parameters can be injected. The class is annotated with the `@Named` qualifier from Jakarta CDI, making it available for reference in the OIDC configuration annotation.

The parameters of the configuration annotation are as follows:

- **clientId** and **clientSecret**: These are credentials used by the application to authenticate with the OpenID Provider (OP).
- **redirectURI**: This is the URI where the OP will redirect the user after successful authentication.
- **scope**: These are the scopes requested from the OP. Scopes define the data and permissions available to the application.
- **prompt**: This sets the prompt behaviour when the user is redirected to the OP.
- **providerURI**: The URI of the OP.
- **jwtReadTimeout** and **jwtConnectTimeout**: These are timeouts (in milliseconds) for reading and connecting to the JWKS endpoint of the OP, respectively.
- **claimsDefinition**: This defines how claims from the OP are mapped in the application.
- **extraParameters**: Additional parameters sent in the authorization request to the OP.
- **logout**: This defines the logout behaviour and redirect URI after logout.

The provider URI can point to any IdP. However, since we are having this discussion using Keycloak as the IdP, the URI should point to a running Keycloak instance, with the client and client ID parameters resolving to a Keycloak client. You can use [this doc](#) from Keycloak to set up a client.

The configured MicroProfile Config properties referenced in the OpenIDSecurityConfiguration class can be defined in any valid MicroProfile Config source. Because your application will go through different stages - development, testing, staging, production - using MicroProfile Config to configure the OIDC information allows for setting different values in different environments automatically without needing to repackage or redeploy the application.

An example of the configured values in the application bundled microprofile-config.properties file is as follows, assuming you are running Keycloak locally on port 5050, with a realm named my-realm.

```
fish.payara.demos.conference.security.openid.clientId=my-conference-app
fish.payara.demos.conference.security.openid.clientSecret=b7ad4b57-ec73-47e9-abc4-50c7364129f3
fish.payara.demos.conference.security.openid.provider.uri=http://localhost:5050/auth/realms/my-realm
fish.payara.demos.conference.security.openid.custom.namespace=http://my-conference-app/claims
```

Declaring Secured Parts Of The Application

With that configuration in place, the application is now ready to authenticate with OIDC. However, only configuration OIDC is half the task. The other half is declaring which part of the application should be protected from unauthenticated users. This can be achieved either by annotating resources with the @RolesAllowed annotation or by setting the constrained paths in the web.xml.

@RolesAllowed

You can use the `@RolesAllowed` to configure specific roles that an authenticated user should have before being allowed to access the protected resource. The `SessionVoteResource` class below shows an example.

```
@Path("/rating")
@RequestScoped
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed("CAN_VOTE")
public class SessionVoteResource {

    @Inject
    AttendeeService attendeeService;

    @Inject
    SessionRatingService ratingService;

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response rate(SessionRating rating) {
        //TODO - Get email properly from logged in user via JWT
        var email = "";
        Attendee currentUser = attendeeService.getByEmail(email)
            .orElseThrow(() -> new BadRequestException("Invalid JWT
token"));
        return Response.ok(ratingService.addRating(rating, currentUser))
            .build();
    }
}
```

The class is annotated `@RolesAllowed`, passing in `CAN_VOTE`. This means the currently executing user must have the given role before access to any method invocation on the class will be allowed. Naturally, anonymous, unauthenticated users will not be allowed. This annotation allows for effortless implementation of Role Based Access Control (RBAC) in combination with the configured OIDC.

Using Web.xml

You can also declare protected resources in the web.xml file of the application. An example of such a declaration is shown below.

```
<security-constraint>
<web-resource-collection>
  <web-resource-name>Session Resources</web-resource-name>
  <url-pattern>/sessions/*</url-pattern>
</web-resource-collection>
<auth-constraint>
  <role-name>admin</role-name>
  <role-name>speaker</role-name>
  <role-name>attendee</role-name>
</auth-constraint>
</security-constraint>
<security-role>
<role-name>speaker</role-name>
</security-role>
<security-role>
<role-name>attendee</role-name>
</security-role>
<security-role>
<role-name>admin</role-name>
</security-role>
```

The traditional `<security-constraint>` element can also be used to declare restricted resources on a user role basis. This was the main way to declare restricted resources on the Jakarta Platform, before the popularity of annotation driven development.

Getting User Authenticated Info

Configuring OIDC and declaring protected resources either through annotation or the web.xml will be enough only for the most trivial of applications. In a much more complex application, there will be the need to have access to the authenticated information of the currently executing user. The Jakarta Security API provides the `OpenIdContext` bean you can inject to get access to the token information of the authenticated user. The `AuthController` below shows the use of the `OpenIdContext` bean.

```
@Named
@RequestScoped
public class AuthController implements Serializable {

    @Inject
    SecurityContext securityContext;

    @Inject
    OpenIdContext openIdContext;

    public boolean isAuthenticated() {
        return Optional.ofNullable(securityContext.getCallerPrincipal()).
isPresent();
    }

    public String getCurrentIdentityName() {
        var nameClaim = openIdContext.getClaims().getName();
        return nameClaim.orElse(null);
    }

    public boolean hasSpeakerRole() {
        return securityContext.isCallerInRole("speaker");
    }

    public boolean hasAttendeeRole() {
        return securityContext.isCallerInRole("attendee");
    }

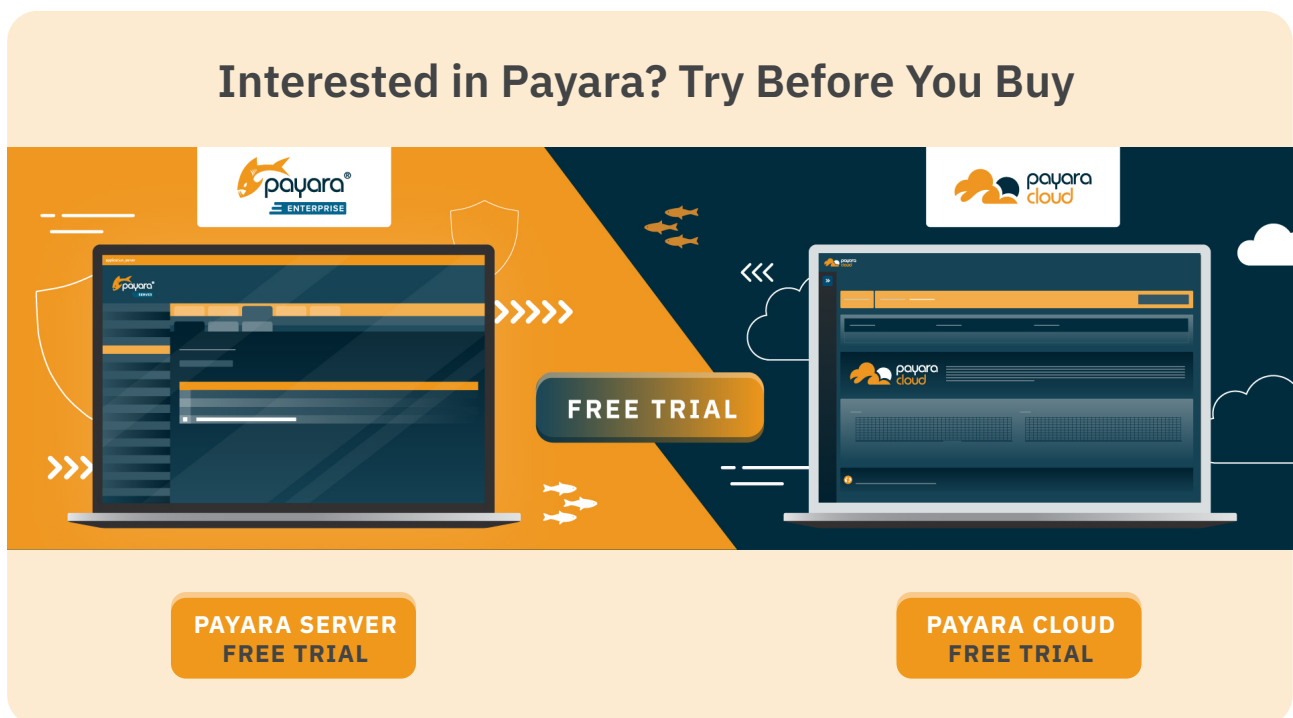
    public boolean hasAdminRole() {
        return securityContext.isCallerInRole("admin");
    }
}
```

The AuthController injects the OpenIdContext and SecurityContext beans. These two artefacts are provided to you by the Jakarta Security runtime. They are guaranteed to not be null. However, depending on what method you invoke, you could get null data returned depending on whether the currently executing user is authenticated with your configured IdP or not. In this example, we call the getClaims method to the name of the user. The getClaims method returns a bean instance of OpenIdClaims that has methods for getting all information about the authenticated user found in the data part of the token.

The SecurityContext is a bean that provides access to programmatic security. In this example, we call the `getCallerPrincipal` to return the `java.security.Principal` object, representing the authenticated user, or null if the user is not authenticated.

Summary

In this guide, we took a look at the basic theory of OIDC authentication, then looked at how to configure OIDC in a Jakarta EE application, using MicroProfile Config for utmost flexibility and configurability. We saw how to use the various constructs provided by Jakarta Security to effortlessly secure an application. The Jakarta Security and OIDC give you a streamlined, simplified and standard way to protect all kinds of applications. As security is a very complex and difficult thing to get right, it is important to use and adhere to industry tried, tested and accepted standards in order to secure your applications.



Interested in Payara? Try Before You Buy

The banner features two laptops. The left laptop displays the Payara Enterprise interface with the logo above it. The right laptop displays the Payara Cloud interface with the logo above it. A central orange button with white text reads "FREE TRIAL". Below each laptop is an orange button with white text: "PAYARA SERVER FREE TRIAL" and "PAYARA CLOUD FREE TRIAL". The background is a dark blue gradient with orange fish icons and white arrows.



sales@payara.fish



UK: +44 800 538 5490
Intl: +1 888 239 8941



www.payara.fish

Payara Services Ltd 2023 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ