Securing REST Web Services on Payara Server with Jakarta Security and OIDC

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

User Guide

Contents

Guide Updated: **January 2024**

The Representational State Transfer or REST architecture, originally propounded by Dr. Roy Fielding, has become the most widely used application architecture. Its simplicity, stateless nature and use of the ubiquitous HTTP standard methods makes it ideal for all kinds of applications on all programming platforms. The Jakarta EE Platform allows you to develop RESTful web services through the Jakarta REST specification. Jakarta REST provides all the core features of REST and along with other Jakarta specifications like CDI (Context and Dependency Injection) and Security, allow for creating modern, secure, and maintainable applications.

The ubiquitousness of HTTP has naturally resulted in many security risks to applications that are exposed to or deployed on the internet. RESTful web services are particularly at higher risk as most are directly exposed to the public facing internet. You can, however, secure your Jakarta REST applications on the Payara Platform through the Jakarta Security specification. This quick guide shows you how you can secure your RESTful applications on the Jakarta EE Platform on Payara Server with Jakarta Security. You should have a basic understanding of application security, OIDC and Keycloak. You can check out this other guide for a quick overview of these concepts.

We start by looking at how to configure it. We then proceed to discuss securing application resources with the relevant constructs, and finally see how to propagate security information across the application. By the end of this guide, you will be able to get started with the Jakarta Security specification securing your Jakarta EE applications in general and RESTful web services in particular.

Jakarta Security and OIDC

The Jakarta Security 3.0 specification, released in Jakarta EE 10, is the standard security API on the Jakarta EE Platform. This release came with native support for the OpenID Connect protocol. As OpenId Connect, or OIDC is a protocol with many implementing vendors out there, Jakarta Security now provides a standard, vendor agnostic way to secure your applications with OIDC. This means you can, at least in theory, migrate from one OIDC provider to the other without a significant refactoring of your application. You can read more about OIDC support in Jakarta Security on the spec page.

Configuring OIDC In Jakarta Security

The root entry to securing RESTful applications with OIDC and Jakarta Security is through the `@OpenIdAuthenticationMechanismDefinition()` annotation. This annotation has several parameters for configuring everything for communicating with a given OIDC vendor or server. The `OpenIDSecurityConfiguration` class below shows a sample configuration for connecting to an OIDC server.

```
@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")
    public String clientID;

    @Inject
    @ConfigProperty(name = "fish.payara.demos.conference.security.openid.
clientSecret")
    public String clientSecret;

    @Inject
    @ConfigProperty(name = "fish.payara.demos.conference.security.openid.
provider.uri")
    public String providerUri;

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

}
```

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.
- **jwksReadTimeout** and jwksConnectTimeout: 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 is what determines which server your application is connecting to. As mentioned above, there are several OIDC vendors out there that all offer OIDC services with minor variations. However, for learning and testing, you can use the freely available and open source Keycloak identity provider. Follow this doc to set up a Keycloak instance in docker.

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 distinct 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 from docker with exposed port 5050 to the host.

```
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
```

Securing REST Resource Methods

With the configuration in place, you are ready to secure your REST resource methods. You can declare a resource method as secure using 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")
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed("CAN_VOTE")
public class SessionVoteResource {

    @GET
    @Path("/session/{session}")
    public List<SessionRating> getRatingsForSession(@PathParam("session")
String sessionId) {
        return ratingService.getRatingsFor(sessionId);
    }
}
```

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>
```

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.

Propagating Security Context

Securing the application is one end. A lot of the time you will need information about the currently executing user to make decisions during application runtime. The Jakarta Security specification provides the `jakarta.security.enterprise.identitystore.openid.OpenIdContext` interface specifically for such situations when using it with OIDC. You can inject this bean into any Jakarta CDI bean and use its methods to get information pertaining to the currently executing user. The `SessionVoteResource` class is reproduced below to show the injection of the `OpenIdContext` bean.

OpenIdContext

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

    @Inject
    OpenIdContext openIdContext;

    @GET
    @Path("/session/{session}")
    public List<SessionRating> getRatingsForSession(@PathParam("session")
String sessionId) {
        return ratingService.getRatingsFor(sessionId);
    }

        @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response rate(SessionRating rating) {

        var email = openIdContext.getClaims().getEmail();
        Attendee currentUser = attendeeService.getByEmail(email.orElse(""))
                .orElseThrow(() -> new BadRequestException("Invalid JWT
token"));
        return Response.ok(ratingService.addRating(rating, currentUser)).
build();
    }
}
```

The updated resource class declares a dependency on `OpenIdContext` with the `@Inject annotation`. The class then calls the `getClaims().getEmail()` method to get the email of the currently executing user. The whole POST method will be executed only when the user is authenticated. So, we know at this point the user exists. However, depending on how the user account is set up with the OIDC vendor or provider, they may not have an email. In that case, we default to an empty string.

The `OpenIdContext` has many utility methods for getting different information about the authenticated user. It is important to note that even though the `OpenIdContext` bean will never be null, even if the user is not authenticated, you can get a null return value on some methods if the executing user is not authenticated. So, if you will be using user information in a context that is not automatically constrained to only logged in users like this `SessionVoteResource` class, then you should first do a null check to avoid running into null related problems.

SecurityContext

The `SecurityContext` is another Jakarta Security bean that you can inject into classes to programmatically check security related information. It has methods to get the `java.security.Principal` object, and another to check if the currently executing user has a specific role, among others. This bean is also guaranteed to not be null but could return null values for some situations if the user is not authenticated. It is important to always do a null check when used outside of an authenticated user requirement context.

Summary

The Jakarta Security API provides an easy, straightforward, and standard way to configure and secure your Jakarta REST resources with the OIDC protocol. This API now provides a standard, portable way to secure your applications on the Jakarta EE Platform. This guide took a quick look securing your Jakarta REST resources with Jakarta Security and OIDC. We looked at configuring and securing REST resources through annotation and xml. We also looked at getting security information with built-in Jakarta constructs, namely the `OpenIdContext` and Security Context. The principles discussed here can be expanded to the most non-trivial of applications. Do check out the Jakarta Security specification for comprehensive usage information.

sales@payara.fish

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

www.payara.fish