User Guide

How to Secure JAX-RS Endpoints of a Stateless Application

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

Contents

Introduction

Securing your application is a very important aspect of the development of your application. You not only need to make sure that the application has the intended functionality but also that this functionality can only be executed by the appropriate people. You not only need to make sure that updates to data are restricted to the correct people, but it is also important that end users only see data they are allowed to see. And in case of sensitive data, this is even more important.

When your application is based on REST endpoints, securing the application is even more challenging. Those endpoints are in most cases called by some front end, written in all types of technologies ranging from browser-based to native mobile apps, and thus you can't ask for a user name and password when they are called. They are also stateless, they don't keep track of previous calls. So we need to provide them all the information about the user in a secure way, every time.

This User Guide will discuss the different aspects of securing the JAX-RS endpoints of your application using standards and common practices like OAuth2, OpenID Connect, JWT Tokens, and MicroProfile JWT authentication in combination with the Payara Platform.

Authentication and Authorization

When we talk about the security of your application, there are two important concepts you need to be aware of.

The first one is **authentication**, which is a question around the end-user who works with your application. There are various ways for identifying the user. The user name and password are the traditional technique, but there are other alternatives that are more common these days. When you maintain the identity data yourself, you need to make sure that they can't get exploited as most users use the same user name and password combination on many sites. Options like the 2 Factor Authentication (2FA using one-time tokens sent to the linked Mobile Phone for example) or 'outsourcing' the identification process using the OpenID Connect flow makes it much more secure.

The result is always the same, after authentication the application knows who the end-user is and can correlate it with a unique identification stored for him.

As mentioned in the introduction section, after a successful authentication by the front end application, this information needs to be passed on to the JAX-RS endpoints called by this front end. We will cover this aspect in the JWT Tokens and MicroProfile JWT Auth Specification section.

The second important aspect is the **authorization** of the end-user. Now that we know who (s)he is, we can determine what functionality is allowed by this user. For each user, we need to determine what functionality is allowed in the application. Can the end-user use certain screens, elements on

the screen like buttons, and is it allowed to send and or read that specific data set to and from the REST endpoints?

So, the authorization is the real heart of the security of your application. It's what defines whether or not the data of your application will be correctly used.

Groups vs Roles

Within the concept of authorization, there is always a lot of confusion about groups and roles, and what they exactly mean.

The **groups** concept is linked to the person. It gives the various classifications of that user. Like he is part of department X and works in Teams Y and Z.

A **role**, on the other hand, is linked to the application. You can have different roles in the application like end-user, administrator, and so on.

And for each application, you need to make a mapping between the group of your user and the role it receives in the application.

Note, however, that using roles in your application is not the most optimal solution. When you need to introduce an additional role within your application because you want more fine-grained control of what people can do, you need to adapt the entire application and perform intensive testing again.

A better solution is to use permissions within the code. For each feature of the application, you need a certain kind of permission. Reading the information about the orders, updating the orders that you have entered, or updating any order, etc ...

You can end up having hundreds of permissions and instead of assigning them individually to all your users, you group them into roles. But, it's important in this scenario that in the event you need to create another role, the application doesn't need to be adapted or re-tested. You only need to encode an additional mapping between the new role and the permissions.

How to Secure a Stateless App

So, how can we apply all this to our JAX-RS endpoints of a stateless application? Passing a user name and password with each request is not an option. This would mean that validation of credentials needs to be performed each time, just as retrieving the authorization.

Something like a session identification or any other opaque value is also not a solution as it requires the same amount of verification and data retrieval.

The use of a token containing all the required information is a workable and performant approach. In the next section, I describe the most commonly used token today.

JWT Tokens

The common way to propagate some user-related information from one service to another is by using the JSON Web Tokens (JWT). JWT is an open standard defined by IETF rfc7519 and is a way to securely transmit claims between parties as a JSON object. Where claims are a statement about the user like user name and roles, but they can contain any kind of information.

The information can be transferred securely because it is signed. This means that although the claims are readable, and thus should never contain sensitive data like medical information, they can't be altered as that would break the signature and such a change can be detected.

The structure of the compact format of the JWT token is:

- *header:* Containing information about the token like used signature algorithm, key identification, etc ..
- *payload:* Contain the claims.
- *signature:* The signature calculated over the header and payload with a (private) key.

These three parts are base64 encoded and concatenated with a '.' (dot).

Such a token is generated by a provider, like KeyCloak, OAuth0, Okta, Apereo CAS, ... and can be verified by the service which receives such a token when it has the (public) key to verify the signature. The advantage of this is that there is no central service required for the verification of such tokens which can become the bottleneck in the system and a target of a DDOS attack.

Those tokens are also safe because:

When header or claims are changed, the signature can detect such modifications and token can be rejected.

One of the claims is an 'expiration time' when the token should no longer be trusted. By keeping this relatively short, 15 minutes is a typical value, the token can only be used in a short period. When the token is captured by someone, it is rapidly unusable.

To keep the JWT tokens safe, the communication should use SSL so that they can't be spoofed from the network.

OAuth2 and OpenID Connect

Now that we know that we should have a JWT token to propagate the user information to our stateless REST endpoints, we will have a look at how they are created.

You can perform the authentication of the end-user based on his user name and password yourself and generate a token.

But this approach has many disadvantages:

- You need to make sure the passwords are stored safely and can't be leaked. Important since many users use the same password for many websites and applications.
- Not only do passwords need to be kept safe, but also the private keys that are used to sign the JWT tokens. And you probably want to implement some kind of rotating key schema to reduce the security risks even more.
- When performing the user management yourself, features like a single sign-on are much more difficult to implement.

These days, a lot of applications 'outsource' the user management to the OpenID Connect provider which uses a JWT token for the identification of the user. This means we can use this id Token for the authentication and authorization of our REST endpoints.

OpenID Connect is built on top of the OAuth2 specification, so let us have a brief look at that first. (I will only cover these topics at a high level, as describing this in detail would require too much space.)

OAuth2 is about the authorization part and is similar to the SAML protocol but uses JSON instead of XML. An example describes the usage scenario the easiest way:

When an application needs to access some files stored by a cloud drive service, the application will redirect you to the Cloud Service where it verifies the credentials the user specifies. But it also asks you which permissions you give to the application on your files.

The resulting opaque token (so it doesn't contain any meaning in it) can then be used by the application to request some files from the Cloud Service supplying this ticket. The Cloud provider then validates the request (are the correct permissions granted by the user when the token is created?) and replies with the requested data if the request is valid.

With OAuth2, the application doesn't know anything about the token. It can use it only to perform some actions against the provider who has created the token.

For more details about the different flows like password and authorization code grant flows, have a look at the specification or any resource explaining OAuth2 more in detail.

OpenID Connect is built on top of this where the token which is created, and is an idToken describing the user. It contains information about the user identity and roles and permissions it has (known by the OpenID Connect provider). This specification is more about the authentication of the user.

And as already mentioned, the id Token is a JWT token which can be used by anyone who receives it to inspect who made the call when it has access to the (public) key to verify the signature and determine the JWT token is valid.

Again, for more details on this, have a look at the specification or any resource explaining OpenID Connect more in detail.

When we create an application, we should ask the OpenID Connect provider to validate the user credentials and supply us with a token which can be used to call our stateless REST endpoints which have all the information they need regarding authentication and authorization within the token.

MicroProfile JWT Auth Specification

The JWT token described earlier is a good concept, but it's missing a few standard claims for what we need.

There is a subject defined (sub claim) but that is not standardized, so the contents are not always the user name we normally use in our application. And OpenID Connect has claims for the name and other fields of the end-user but not for the user name.

Also, since OpenID Connect is geared towards the authentication there is no claim which describes the roles or the groups the user is assigned to.

The MicroProfile JWT authentication specification defines these gaps. It starts from the JWT token as they are defined for the OpenID Connect specification but adds a few claims

The `upn` and `preferred_username` to pass on the user name information via the token.

A groups claim is also added containing all the group names assigned to the user. As discussed above, groups is the correct term to refer to the set of divisions a user belongs to.

The specification also defines that there is a one to one mapping to the application roles we can use within the application.

The MicroProfile specification defines also how we can access the information defined within the JWT token from within our code.

Just like all other MicroProfile specifications, it is based on CDI so the use of `@Inject` was a natural choice.

The following snippets will give you access to the preferred_username claim value.

```
@Inject
@Claim(standard = Claims.preferred_username)
private String subject;
```

If you need to have access to multiple claims, you can either inject all them separately or grab the entire token in a `JsonWebToken` instance

```
@Inject
private JsonWebToken callerPrincipal;
```

@RolesAllowed

Instead of verifying if the user has the correct application role to execute a certain method in a programmatic way the preferred way is to use an annotation for this.

The MicroProfile JWT Auth specification recommends that implementations support the `@RolesAllowed` defined in the Java EE / Jakarta EE specification. Within the Payara implementation, the JWT token information can be used in combination with the `@RolesAllowed` annotation. The MicroProfile JWT authentication and authorization are entirely integrated into the server which means that all subsystems can base their decisions on the contents of the token.

```
@RolesAllowed("app-role")
public Response performAction() {
      return ...
}
```

Example

In this example, we will set up Keycloak to provide us with a JWT token when we sign in and use this to secure some JAX-RS endpoints within Payara Server or Payara Micro.

Configure Keycloak

You can download and start Keycloak locally

- Download from the Keycloak download page.
- Unzip and start up the server `./bin/standalone.sh -Djboss.http.port=8888`
- • Open the main page of Keycloak in the browser http://localhost:8888/auth/ and create an administrative user with the name `payara`.

or you can use the Docker Image to get the server running and define the administrative user.

```
docker run -e KEYCLOAK_USER=payara -e KEYCLOAK_PASSWORD=payara -p 8888:8888 -d
jboss/keycloak:8.0.2
```

In the next steps, we are going the create a specific realm for our application with all the configuration we need to integrate it with the Payara Platform using the MicroProfile JWT authentication specification.

1. Open the Administration console of Keycloak at the URL `http://localhost:8888/auth/admin`
2. Login with the administrative user you have created (or defined as environment variables for the Docker Container) earlier on.
3. Hover over the `Master` label in the top left corner and click on the `new realm` button.
4. Enter the realm name `payara_jwt_demo` and click on the `Create` button.
5. Click on the `Roles` menu item in the left and create a new application role by clicking the `Add Role` button.
6. Define the role name as `app-role` and click the `Save` button.
7. In this step we create a user for our example and assign it the created role. In the menu on the left side, click the `Users` item, click the `Add user` button and specify the username `test` for example, and click on the `Save` button. Click on the `Credentials` tab and define the password for the user, switch of the `temporary password flag` and click on the `Set password` button. Click on the Role Mappings tab and assign the `app-role` to the user (select item and click on `Add selected`)
8. Click on the `Clients` menu item in the left and create a new client by clicking the `Create` button.

9. The settings page of the client is shown now. No changes need to be made for this example but when you integrate Keycloak as OpenID Connect provider within your JSF application for example, a few value are required as the redirect URI. For this example, the Direct Access Grants Enabled must be on (is the default in version 8.0.2) as we will call a Keycloak endpoint with the user name and password (which is just for demo purposes here as in production you should use the Authorization Code flow.
10. Click on the Mappers tab and the Add builtin button on the right of the screen.
11. Check the checkbox on the row groups and add it to the mappers. This will make sure the application roles are available with the idToken issued by Keycloak.

Testing Keycloak

You can test out this configuration by issuing a POST call to the Keycloak endpoint.

```
curl --data "realm=payara_jwt_demo&grant_type=password&client_id=mp-client&us
ername=test&password=test" http://localhost:8888/auth/realms/payara_jwt_demo/
protocol/openid-connect/token
```

Or you can use any other tool to submit the POST request like PostMan. The important things to know are

endpoint : http://localhost:8888/auth/realms/<realm-name>/protocol/openid-connect/token

Body parameters

- realm: The name of the realm you have created
- grant_type: fixed value password since we are using the password flow in this example.
- client_id: The client we have created, this corresponds with the application.
- username: The name of the user
- password: The password of the user.

Since the OpenID Connect providers should we accessed over a secure channel, (using https) it is not an issue that we have the password in plain text in the call. Remember also that the use of the password grant is not a good production practice and that with a Authorization Code flow, the password is exchanged bewteen browser and OpenID Connect provider immediately (also over SSL)

The response is something similar to

{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI3Wkp6VHgxTzB3ajdoc3hDbnpHNHc2Wkw4WS0wQ3dkNm1Xb1g2LXNVcHdNIn0.eyJqdGkiOiJhNmI3OGEzYi1lOTM4LTQ0ZjItYjlxNi1hOTYzMGVkZDU2OGQiLCJleHAiOjE1ODI1NTQxOTMsIm5iZiI6MCwiaWF0IjoxNTgyNTUzODkzLCJpc3MiOiJodHRwOi8vbG9jYWxob3N0Ojg4ODgvYXV0aC9yZWFsbXMvcGF5YXJhX2p3dF9kZW1vIiwiYXVkIjoiYWNjb3VudCIsInN1YiI6IjRiMzg0MTM5LWM3ZWEtNGMxOS05MzI3LThhNzc0MjEzNDA5NCIsInR5cCI6IkJlYXJlciIsImF6cCI6Im1wLWNsaWVudCIsImF1dGhfdGltZSI6MCwic2Vzc2lvbl9zdGF0ZSI6IjhiMjE2N2Y4LTUyMDAtNDFkMy1hMDNkLWI1YTdiMzFjZGQ5YSIsImFjciI6IjEiLCJyZWFsbV9hY2Nlc3MiOnsicm9sZXMiOlsib2ZmbGluZV9hY2Nlc3MiLCJhcHAtcm9maWxlLWFwaSIsInVtYV9hdXRob3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwic2NvcGUiOiJwcm9maWxlIGVtYWlsIiwiZW1haWxfdmVyaWZpZWQiOmZhbHNlLCJncm91cHMiOlsib2ZmbGluZV9hY2Nlc3MiLCJhcHAtcm9maWxlLWFwaSIsInVtYV9hdXRob3JpemF0aW9uIl0sInByZWZlcnJlZF91c2VybmFtZSI6InRlc3QifQ.Cw64qbCxRCC__qEDtNsuzq8nsam1B0j19347OCmDoBM2hFRfFegrQiq_Hu3XKZGtZhzrS0J25YhalciTZkr0HDgdxNd4RA1EtEa6q7vCCCkNnHqqeQnH7VJuZSycVRYYfWF3YTMJjqJ4EYKoYbFuyBHiYhJds6OocRtU67ly67J6xE6IZixWi8kUtKHJ1heVEtuJa5hc6G6mTQ5uJOHlFclhMPflAtmL9j46kfO2Q_e1hIF20CMEwKARDUzfxWiGI6EBPCr0IiE4Ns42wVQxwTm_cLI9tZ9oEDweQh3UGMTZd2ddw19JhG87OAv6nv3fqA0joJMzSmM_An34TFUI7A","expires_in":300,"refresh_expires_in":1800,"refresh_token":"eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJiODkyMjMzOC1mMDY2LTQyYWEtOTZhMy00YTI1ZmY4ZGE5ODAifQ.eyJqdGkiOiIwNWYxOTJjYi0wMjYyLTRlMTgtOTQ1MS03MDg1ZGEzODMzMDgiLCJleHAiOjE1ODI1NTU2OTMsIm5iZiI6MCwiaWF0IjoxNTgyNTUzODkzLCJpc3MiOiJodHRwOi8vbG9jYWxob3N0Ojg4ODgvYXV0aC9yZWFsbXMvcGF5YXJhX2p3dF9kZW1vIiwiYXVkIjoiaHR0cDovL2xvY2FsaG9zdDo4ODg4L2F1dGgvcmVhbG1zL3BheWFyYV9qd3RfZGVtbyIsInN1YiI6IjRiMzg0MTM5LWM3ZWEtNGMxOS05MzI3LThhNzc0MjEzNDA5NCIsInR5cCI6IlJlZnJlc2giLCJhenAiOiJtcC1jbGllbnQiLCJhdXRoX3RpbWUiOjAsInNlc3Npb25fc3RhdGUiOiI4YjIxNjdmOC01MjAwLTQxZDMtYTAzZC1iNWE3YjMxY2RkOWEiLCJyZWFsbV9hY2Nlc3MiOnsicm9sZXMiOlsib2ZmbGluZV9hY2Nlc3MiLCJhcHAtcm9maWxlLWFwaSIsInVtYV9hdXRob3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwic2NvcGUiOiJwcm9maWxlIGVtYWlsIn0.r0gif-Jc_UwIXUNqFLfNX-oF4bGwRkpxRVwFUz6wq90","token_type":"bearer","not-before-policy":0,"session_state":"8b2167f8-5200-41d3-a03d-b5a7b31cdd9a","scope":"profile email"}

The value of the access token property is the one we are interested in which is the JWT token. If you examine the contents by performing a BASE_64 decoding on the 3 parts, or you put it on a site like jwt.io, you can see the contents of the token.

Header


```
{
"alg": "RS256",
"typ": "JWT",
"kid": "7ZJzTx1O0wj7hsxCnzG4w6ZL8Y-0Cwd6mWoX6-sUpwM"
}
```


Body / Claims


```
{
"jti": "a6b78a3b-e938-44f2-b216-a9630edd568d",
"exp": 1582554193,
"nbf": 0,
"iat": 1582553893,
"iss": "http://localhost:8888/auth/realms/payara_jwt_demo",
"aud": "account",
"sub": "4b384139-c7ea-4c19-9327-8a7742134094",
"typ": "Bearer",
"azp": "mp-client",
"auth_time": 0,
"session_state": "8b2167f8-5200-41d3-a03d-b5a7b31cdd9a",
"acr": "1",
"realm_access": {
      "roles": ["offline_access","app-role","uma_authorization"]
      }
"resource_access": {
      "account": {
            "roles": ["manage-account","manage-account-links","view-profile"]
            }
      },
"scope": "profile email",
"email_verified": false,
"groups": ["offline_access","app-role","uma_authorization"],
"preferred_username": "test"
}
```

Application

In this next section of the demo, we are creating a simple application containing an endpoint which we need to protect. We can deploy this application to the Payara Server or run it with Payara Micro. Both systems support fully the MicroProfile specifications used in the example.

The endpoint will only be accessible when the request contains a JWT token issued by our Keycloak Server, and the user will need a specific role.

The steps are

1. Create a maven project from scratch, or use the MicroProfile Starter site (https://start.MicroProfile.io). That project is geared towards running Payara Micro as a fat jar. But you can use also a WAR which is deployed on Payara Server or running with Payara Micro.
2. Specify the MicroProfile dependencies for the project if not already defined in the `pom.xml` file.

```
<dependency>
        <groupId>org.eclipse.MicroProfile</groupId>
        <artifactId>MicroProfile</artifactId>
        <version>3.2</version>
        <type>pom</type>
        <scope>provided</scope>
</dependency>
```

3. Create or adapt the JAX-RS application Java class file to

```
@ApplicationPath("/data")
@ApplicationScoped

@LoginConfig(authMethod = "MP-JWT")
@DeclareRoles({"app-role"})
public class DemoRestApplication extends Application {
}
```

Where each of the annotation has the following purpose

- *@ApplicationPath*; defines the part of the URL hosting the JAX_RS endpoints
- *@ApplicationScoped*; Defines the class as CDI bean so that the following 2 annotations are picked up (since MicroProfile is CDI based, annotations are only discovered on CDI beans)
- *@LoginConfig(authMethod = "MP-JWT")*; Defines that the endpoints are protected by JWT tokens as defined by the MicroProfile JWT Authentication spec.
- *@DeclareRoles({"app-role"})*; Defines the roles which are used by the application.

4. Define the JAX_RS endpoint and restrict its usage

```
@Path("/protected")
@RequestScoped
public class ProtectedController {

	@Inject
	@Claim(standard = Claims.preferred_username)
	private String subject;

	@GET
	@RolesAllowed("app-role")
	public String getJWTBasedValue() {
		return "Protected Resource called by " + subject;
	}
}
```

- *@RequestScoped*; indicates it is a CDI bean and the MicroProfile JWT auth annotation are recognized.
- *@Claim()*; indicate we want to ave access to a specific claim (here the user name) defined within the JWT token
- *@RolesAllowed;* Defines the role we expect the user to have before the call can be executed.

5. Create a MicroProfile properties file for the configuration of the JWT token verification. The file `src/main/resources/META-INF/MicroProfile-config.properties` needs the following content

```
mp.jwt.verify.publickey.location=http://localhost:8888/auth/realms/
payara_jwt_demo/protocol/openid-connect/certs
mp.jwt.verify.issuer=http://localhost:8888/auth/realms/payara_jwt_
demo
```

   The first parameter defines the URL where the public key can be retrieved for the verification of the signature.

   The second one is to make sure that we only accept tokens which are issued by our Keycloak (the issuer is one of the claims in the token).

6. Build the artifact and deploy it on Payara Server or run it with Payara Micro.

Test the Application

The JWT token needs to be passed as a header value when making the request to the endpoint, just as you need to do when using OAuth2 and OpenID Connect services.

Authorization: Bearer `<access-token>`

So, the endpoint we have created in the example can be called:

```
curl --location --request GET 'http://localhost:8080/demo/data/protected'
--header 'Authorization: Bearer eyJhbGci...'
```

Be aware that the access token you have received from the Keycloak server is by default only 5 minutes valid. So make sure you have request a recent one when trying out the call.

Passing on JWT Token

Within a microservices architecture, it is very common that one microservice calls another one. Since each one is responsible for a specific domain, communication is required between them.

So, the first microservice request gets a request with the correct header, and verification succeeds. How can it be passed on to another microservice which is called?

Before answering that question, let us have a look at how you can easily call an endpoint with the help of MicroProfile Rest Client specification.

With the standard JAX-RS client, one can construct a WebTarget and retrieve the Response.

```
Client client = ClientBuilder.newClient();
WebTarget webTarget
      = client.target("http://<host>:<port>/path/to/other/endpoint");
Response response = webTarget.request(MediaType.APPLICATION_JSON).get();
```

The way you construct and execute such a call is rather low level.

With MicroProfile Rest Client, you can use some high-level constructs and calling a REST endpoint appears the same as calling any other service. The only difference, of course, is that it effectively uses a remote call.

You start by defining an interface which describes the endpoint:

```
@RegisterRestClient
@ApplicationScoped
public interface ProtectedService {

      @GET
      @Path("/protected")
      String getValue();
}
```

The important factor here is that you annotate the interface with @RegisterRestClient so that a proxy class will be created and can be used later on.

Of course, the example here is not very useful, but the method can return a POJO and with @Produces we can indicate that JSON, for instance, will be used for the communication. Automatic conversion to the Java instance of the return is performed by the system when we call it.

The return type and parameter of the method, like path parameter or query parameters, need to match just as the HTTP method, GET in this case with the remote endpoint definition.

The proxy can be used in the following way:

```
@Inject
@RestClient
private ProtectedService service;

@GET
@Path("/test")
@RolesAllowed("app-role")
public String doTest() {
      ...
      service.getValue();
      return ...
}
```

With the use of a ‘regular’ inject, we only need to supply the Qualifier @RestClient, we get a bean implementation that can perform the call and perform all the operations behind the scene like the JSON conversion.

How can we make sure that the authentication header is now passed on from one endpoint to the other one?

By simply annotating the interface with `@RegisterClientHeaders`, the generated code will make sure that all header data on the incoming request, so also the Authorization header with the JWT token, are passed on to the called endpoint.

Security Must be Addressed When Developing Applications

Security is one of the major aspects of the application that needs to be addressed. When using REST endpoints this imposes an additional challenge. How can user information be passed on in a secure way and in a format that the microservice can validate with additional remote calls?

By using JWT tokens that contain claims about the user identity and authorization, we can make sure that this information is passed in a way that rules out tampering. We also discussed the usage of an OpenID Connect provider to supply these tokens instead of generating them yourself.

In the second half of the guide, a detailed description of an application is given to use such JWT tokens and how these tokens can be passed on to other microservices.

sales@payara.fish **+44 207 754 0481** **www.payara.fish**