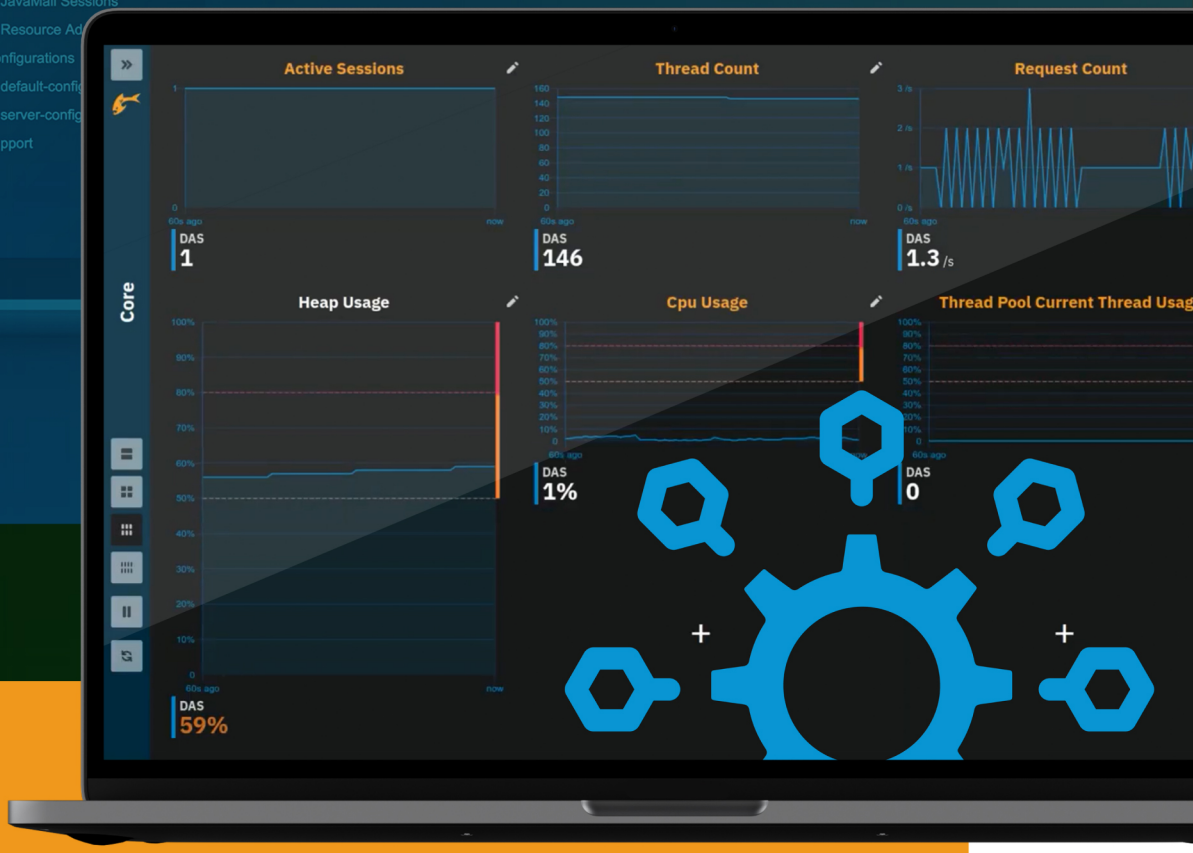
Securing Jakarta EE Applications with MicroProfile JWT



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

User Guide

Contents

Guide Updated: **December 2023**

Securing applications is a critical aspect of modern software development, ensuring that only authorised users can access sensitive functionalities and data. In the realm of Java enterprise development, one of the robust solutions for securing applications is the use of MicroProfile JWT (JSON Web Tokens). This approach combines the strengths of Jakarta EE, with the agility and portability of MicroProfile standards, particularly for microservices architectures.

MicroProfile JWT (MP JWT) is a specification within the MicroProfile project, designed to provide a common way of using JWTs for authentication and authorization in microservices. This specification aligns with the industry-standard JWT format, a compact, URL-safe means of representing claims to be transferred between two parties, enabling Single Sign-On (SSO) and simplified token-based authentication and authorization. Using this, Jakarta EE applications can leverage MP JWT to enhance their security, allowing seamless integration with modern authentication mechanisms like OAuth 2.0 and OpenID Connect.

This guide will show you how to secure Jakarta EE applications with MicroProfile JWT authentication. As security is a very complex topic, this guide uses the Keycloak project as the security provider. It is recommended to not roll out your own security infrastructure but defer to experts. Keycloak is an open source identity and access management (IAM) framework. It provides user federation, strong authentication, user management, fine-grained authorization, and more. This chapter delegates the process of creating users to Keycloak and focuses on using JWT to secure web resources.

The guide starts by looking at the anatomy of a JWT, the proceeds to using the MicroProfile JWT Authentication APIs to secure and get information from JWT tokens. By the end of this guide, you will be able to secure your Jakarta EE applications using the MicroProfile JWT API.

Token Based Authentication

RESTful web services are stateless by nature. This means every request is independent of the previous request. The server will need to be given the same security related information with each request. The most efficient way to achieve this form of stateless security is through the use of tokens. A token is a long string that can be verified to ascertain the veracity of the claim of a calling client. It allows systems to authenticate, authorise and verify the identity of a client. Token based security does not rely on the HTTP session, making it much more scalable, performant and reliable.

The token is mostly added to the request header. This will then be grabbed by the service and validated, introspective to extract information about the calling client and then create a security context that can then be propagated to all parts of the protected service. The MicroProfile JWT API is a token based authentication, authorization and Role Based Access Control (RBAC) mechanism that can be used for token based application security. Json Web Token or JWT is a token based standard that has emerged as the most popular standard for creating lightweight security tokens.

A JWT token is usually signed so the service can verify the token. The information contained in a token is called claims. The claims can also be encrypted so they are not passed as plain text in the header. A typical JWT is made up of the header, the claims (or body) and the signature.

The Header

The JWT section of a JWT typically contains the type of algorithm used to sign the JWT (RS256, ES256, HS256 etc), the type of the token (e.g. JWT), the key ID and other metadata depending on the issuer of the token. Signing the JWT only enables receiving service to verify that the information has not been tampered with. It does not make the token unreadable. For that, a token will need to be encrypted, in addition to the signing.

The Claims (or Body)

The claims or body of the JWT is all the information that the client would like to pass to the server. A lot of the time the body contains the roles that a given client is assigned. Roles determine the extent to which a client can carry out different operations in an application.

The Signature

The signature is optional by highly recommended information that allows the service to verify that the token has not changed or been tampered with along the way.

A JWT is a base64 encoded string of the header, claims and signature information, a sample of which is shown below.

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJEMkdxYmxJU3YzY1Jpdmh2Z
FR0YmpaVDg2VkFGeUFpT29wZGlvZHYtQllJIn0.eyJleHAiOjE2NjE2MzY5MDksImlhdCI6M
TY2MTYzNjYwOSwianRpIjoiOTAxOGVlZmEtZDc2Ni00ODI4LTgwMmUtYjFkNjMwOTQ5ODUxIiwiaX
NzIjoiaHR0cDovL2xvY2FsaG9zdDo1MDUwL3JlYWxtcy9qd2FsbGV0IiwiYXVkIjoiYWNjb3Vud
CIsInN1YiI6ImY2OTU0YTRkLTFmZmEtNGNkMC05YmRhLTM4NmVkOWQwNzQ3ZCIsInR5cCI6IkJ
lYXJlciIsImF6cCI6Imp3YWxsZXQtc2VydmljZSIsInNlc3Npb25fc3RhdGUiOiJlOTY
1ZWQ3YS00OTI3LTQ5MzItODQyOS1jNWNhMjNmYzQ5NzMiLCJhY3IiOiIxIiwicmVhbG1fYWNjZXN
zIjp7InJvbGVzIjpbIm9mZmxpbmVfYWNjZXNzIiwiZGVmYXVsdC1yb2xlcy1qd2FsbGV0IiwidGVs
bGVyIiwidW1hX2F1dGhvcml6YXRpb24iXX0sInJlc291cmNlX2FjY2VzcyI6eyJhY2NvdW50Ijp
7InJvbGVzIjpbIm1hbmFnZS1hY2NvdW50IiwibWFuYWdlLWFjY291bnQtbGlua3MiLCJ2aWV3LX
Byb2ZpbGUiXX19LCJzY29wZSI6InByb2ZpbGUgZW1haWwiLCJzaWQiOiJlOTY1ZWQ3YS00OT
I3LTQ5MzItODQyOS1jNWNhMjNmYzQ5NzMiLCJ1cG4iOiJtYXgiLCJlbWFpbF92ZXJpZmllZCI6Zm
```

```
Fsc2UsImdyb3VwcyI6WyJvZmZsaW5lX2FjY2VzcyIsImRlZmF1bHQtcm9sZXMtandhbGxldCIsIn
RlbGxlciIsInVtYV9hdXRob3JpemF0aW9uIl0sInByZWZlcnJlZF91c2VybmFtZSI6Im1heCJ9.lG
6zEr-BEMQwyEUswIgCM5InWAH4ljAu8JK6Zc0naHD8kcaGAssxsXHzf315n--cysYJzeAQBfD
mzsJr_Roe0io2nN4CZ61T6Vz4YnE_5_RRn9n6cl2rG_1oIsLknqKVseNQ4itZZjKfIGVUYxURxK
mO6gsCH8FCHN1hu7rmq4jwPC0aO7Ypvjg2IpExsIociprXk7_1iyfllZ4XnlHYBADOpkNOduJgMyo
jQwBgTz-xa-9sBb-eoKys1gBD3nbV8STNbKAGzOUiQaq28T4x3pIn6JNfw2JbBu6_cxKXhzghoPuk
boLhn2C_MYJt4OiycxuprsbDRKoxAIjus_9yoA
```

The same JWT is shown decoded below.

```
{
    "alg": "RS256",
    "typ": "JWT",
    "kid": "D2GqblISv3cRivhvdTtbjZT86VAFyAiOopdiodv-BYI"
  }
  {
    "exp": 1661636909,
    "iat": 1661636609,
    "jti": "9018eefa-d766-4828-802e-b1d630949851",
    "iss": "http://localhost:5050/realms/jwallet",
    "aud": "account",
    "sub": "f6954a4d-1ffa-4cd0-9bda-386ed9d0747d",
    "typ": "Bearer",
    "azp": "jwallet-service",
    "session_state": "e965ed7a-4927-4932-8429-c5ca23fc4973",
    "acr": "1",
    "realm_access": {
      "roles": [
        "offline_access",
        "default-roles-jwallet",
        "teller",
        "uma_authorization"
      ]
    },
    "resource_access": {
      "account": {
        "roles": [
          "manage-account",
          "manage-account-links",
```

```
          "view-profile"
        ]
      }
    },
    "scope": "profile email",
    "sid": "e965ed7a-4927-4932-8429-c5ca23fc4973",
    "upn": "max",
    "email_verified": false,
    "groups": [
      "offline_access",
      "default-roles-jwallet",
      "teller",
      "uma_authorization"
    ],
    "preferred_username": "max"
  }

 {
    "e": "AQAB",
    "kty": "RSA",
    "n": "3buUZWr0Fp9Rm85GPIAoYfiqNapj2DxW8gjMWOjPEgg4KUqg3tSC6G0kAzPJ70zMm
fe-b3CWBQvmNXWIUZCXPlnBuVKyWcowBQk7QRnEkOuw5vL6bYU5I2DA_1CPIn4v3g4ox0GmjOwD
flrlb2vlPw76BpHIKd-EUI-mkm-Yv6H60BZmo7iRyVF7TNGVYgIg0xrVFPIvTDV2yvZKO4qG6qY
Wk2khMPqXm-725QBp2wgWDhowibjuoOCW1S5BCr5qc3N-uaPej7Nbcub80v1XGJ7feHqg4TXtND
CSikTAGCa0geW3eP1wRoE_9QYzKlN45LI8LxhncUl58WpdulgblQ"
  }
```


The MicroProfile JWT

The MicroProfile JWT API provides a standard, mostly opinionated way of verifying a JWT token, authorising the client and propagating the resulting security context to the rest of the application. It achieves these objectives by requiring that JWTs be signed with RSA-based digital signature algorithms. All that the JWT runtime needs is the public key of the authorising server which creates the JWT to verify the token. It also mandates certain information to be available within the claims of the JWT. These are:

- exp - The expiration of the JWT. This ensures old, expired tokens are rejected
- iss - The issuer of the token
- iat - The issued at, identifies when the token was issued
- upn - This is the user principal or the preferred username of the currently logged-in user

It also has recommended claims as follows:

- sub - who is being represented by this token?
- jti - A unique identifier for this claim
- aud - Identifies the MP JWT endpoint(s) which can be accessed by JWT

These requirements allow a Jakarta EE application to be fully stateless. The sub claim will be mapped to a java.security.Principal instance. The groups will be used to determine if the Principal is in a given role required by a given resource method. The client will send the token with each request, the JWT runtime will validate and propagate the security context with each call.

MicroProfile JWT Security Process

MicroProfile JWT allows the creation of stateless microservices by delegating the session information about users to the client. Each call by the client contains all the information needed to construct a security context. This security context is then plugged into existing Jakarta EE infrastructure that is available to all parts of the application. There are three steps in the MicroProfile JWT security process.

Authenticate the Client

The MicroProfile JWT runtime automatically validates the passed JWT token on the request header using the available public key of the issuing server or authority. This is where Keycloak comes in in our context. We use Keycloak as the issuing authority to issue the JWT tokens on our behalf.

A typical enterprise application will have two tiers. The client or UI tier, which can be written in for instance Angular or Vue, and the server or resource tier. The client tier is the “user” or client of the server or resource tier. An end-user will typically interact with the UI tier. When the end-user clicks a button in the UI to perform an action, for instance make a currency conversion, the UI sends a request to the backend or server. This request by the UI to the server is what will contain the JWT token. But how does the UI get the token?

When a user clicks to sign up through the UI, that user creation process will be delegated to Keycloak. So the user credentials will be stored in Keycloak. When the user logs in through the UI, the UI redirects the user to Keycloak to log in. Keycloak authenticates the user and redirects them back to the UI. From this point, any operation made by the logged in user that entails calling the backend server, the UI can go to Keycloak for a JWT token on behalf of the user, which will be forwarded to the server with each request that the logged in user performs.

This architecture fully decouples the backend server, or in this case our Jakarta EE application from knowing anything about the user. Our resources simply require a valid token, and respective methods will require the currently executing user to have certain roles. All of this will be sent to the server with each call. This way, the UI tier and the backend server tier are fully decoupled and stateless. There is no state sharing between them. Every security information needed by the server is sent with each client request.

Providing the Public Key

The MicroProfile JWT runtime will validate and authenticate the passed JWT automatically by using the public key of the issuing server, in this case keycloak. The MicroProfile JWT spec requires the provision of the public key, or a URL for getting the public key. The specification defines two MicroProfile Config properties for passing the public key of the issuing authority or server to the MicroProfile JWT runtime. These properties are mp.jwt.verify.publickey and mp.jwt.verify.publickey.location.

The first property allows the passing of the public key text itself as the value of the property. The second property allows specifying a URL from which the public keys can be downloaded. Specifying both properties results in a deployment exception. The mp.jwt.verify.issuer property is also used to verify the value of the iss or issuer claim. The passed value in the JWT must match the passed value in the property.

The following shows an example of passing the Keycloak URL to the runtime, assuming running Keycloak locally on port 5050.

```
mp.jwt.verify.publickey.location=http://localhost:5050/realms/jwallet/protocol/
openid-connect/certs
 mp.jwt.verify.issuer=http://localhost:5050/realms/jee-gpt
```

A call to http://localhost:5050/realms/jee-gpt/protocol/openid-connect/certs returns the sample public keys shown below.


```
{
    "keys": [
    {
    "kid": "D2GqblISv3cRivhvdTtbjZT86VAFyAiOopdiodv-BYI",
    "kty": "RSA",
    "alg": "RS256",
    "use": "sig",
    "n": "3buUZWr0Fp9Rm85GPIAoYfiqNapj2DxW8gjMWOjPEgg4KUqg3tSC6G0kA
zPJ70zMmfe-b3CWBQvmNXWIUZCXPlnBuVKyWcowBQk7QRnEkOuw5vL6bYU5I2DA_1CPIn4v3g4ox
0GmjOwDflrlb2vlPw76BpHIKd-EUI-mkm-Yv6H60BZmo7iRyVF7TNGVYgIg0xrVFPIvTDV2yvZKO
4qG6qYWk2khMPqXm-725QBp2wgWDhowibjuoOCW1S5BCr5qc3N-uaPej7Nbcub80v1XGJ7feHqg4TX
tNDCSikTAGCa0geW3eP1wRoE_9QYzKlN45LI8LxhncUl58WpdulgblQ",
    "e": "AQAB",
    "x5c": [
    "MIICnTCCAYUCBgGC2NIn1TANBgkqhkiG9w0BAQsFADASMRAwDgYDVQQDDAdqd2FsbGV0MB4X
DTIyMDgyNjA2MjIwN1oXDTMyMDgyNjA2MjM0N1owEjEQMA4GA1UEAwwHandhbGxldDCCASIwDQY
JKoZIhvcNAQEBBQADggEPADCCAQoCggEBAN27lGVq9BafUZvORjyAKGH4qjWqY9g8VvIIzFjozxII
OClKoN7UguhtJAMzye9MzJn3vm9wlgUL5jV1iFGQlz5ZwblSslnKMAUJO0EZxJDrsOby+m2FOSNg
```

```
wP9QjyJ+L94OKMdBpozsA35a5W9r5T8O+gaRyCnfhFCPppJvmL+h+tAWZqO4kclRe0zRlWICINMa1R
TyL0w1dsr2SjuKhuqmFpNpITD6l5vu9uUAadsIFg4aMIm47qDgltUuQQq+anNzfrmj3o+zW3Lm/
NL9Vxie33h6oOE17TQwkopEwBgmtIHlt3j9cEaBP/UGMypTeOSyPC8YZ3FJefFqXbpYG5UCAwEAA
TANBgkqhkiG9w0BAQsFAAOCAQEAuuo6yFASkSWDLh9kyMTZhhcfV87ZEoC4kmkNn/2N2d0gypqxx
LWPDG7rk2QLqocNT0IP6wh+cr9TO/oAUdIc1vAJuLWsw/XiGbPjTVmxw9r1eg3tMphrw+wG4CrRN
sKTs1xgDDMnAyTA7zNZylSwAJ2YBy/7WAb1DzPdpwxuERY3Bn1U9pTV6Tth+SY3n0DVWYl9ik2eyv/
UoTCLdf9gInxHxoyl6moIH944UZXdVnZ9s+TONtVu278fQNUJMbvufN1+IQQkZLH5zehVd8IY/vX
1q9ROjFeaHjYEYSoggo3tCT93KVFYGA6UCm2ADQfluMxNwkIIdvfRVzqIuCKZuw==
    ],
    "x5t": "uej9O_JH-rlIpJVQMTBiD6kf-1k",
    "x5t#S256": "szj37ByL-T6Plrmzbgb7Bnq32KsYtGp_B_dDOYGEWcI"
    },
    {
    "kid": "B5uP6pwSkS-wvKmh31ZeEmeXNKU_UaO9EfTOeut00NY",
    "kty": "RSA",
    "alg": "RSA-OAEP",
    "use": "enc",
    "n": "sR3wZR9t35zM_sftCb2EbX6_tZo1boubrmnuj8YZ1utCOl2pQGX7v3uz_
YHGsAbp-3J3iCt94KliRKA2yYCVi2Ozqd-EEz6GAYxF1HzYghWO-j9jMrNu1N-O8LXVEp21wP
IDsIuCLkqYWZGgoOzY2yyWkVjiHcAu3_LokwufVu-BM9iRY2FZrvgYEHPGZOk5a8hTIHSIx6dX
UcU6oZ0etiGdidrIWH5GFXLSZtpcuya-w01_o1Khm97dCtGY3pVasnLgDK6Vguzul1MdpTQHw
FI7m3O7uQB0CRtqe8qPnNxVLal_Ow5b5N7YNw5fufEqvLk5Dz-au4TPuFwSSvVrw",
    "e": "AQAB",
    "x5c": [
    "MIICnTCCAYUCBgGC2NIoXjANBgkqhkiG9w0BAQsFADASMRAwDgY
DVQQDDAdqd2FsbGV0MB4XDTIyMDgyNjA2MjIwN1oXDTMyMDgyNjA2MjM
0N1owEjEQMA4GA1UEAwwHandhbGxldDCCASIwDQYJKoZIhvcNAQEBBQADg
gEPADCCAQoCggEBALEd8GUfbd+czP7H7Qm9hG1+v7WaNW6Lm65p7o/GGdbrQjpd
qUBl+797s/2BxrAG6ftyd4grfeCpYkSgNsmAlYtjs6nfhBM+hgGMRdR82IIVjvo/YzKzbt
TfjvC11RKdtcD3SA7CLgi5KmFmRoKDs2NsslpFY4h3ALt/y6JMLn1bvgTPYkWNhWa74GBB
zxmTpOWvIUyB0iMenV1HFOqGdHrYhnYnayFh+RhVy0mbaXLsmvsNNf6NSoZve3QrRmN6VWr
Jy4AyulYLs7pdTHaU0B8BSO5tzu7kAdAkbanvKj5zcVS2pfzsOW+Te2DcOX7nxKry5OQ8/
mruEz7hcEkr1a8CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAXW0wtyu6Tmv6i9rEvly5jvNcsNs
vHVpW9tSV0KVWFP0YgvQ0dIQoRhygpvVfZMrlXW5EQ0uKnvZCbt/BzEPMl3G6DfsaTssw1Gea
quMgw1osNqG7HCHGqQX8+EG5U3Neov4+YsSCtYYCWsYqO8OYK6lqp8TQ0+Ok6u9aTpFvH233G
d9ZmM72fN8omfR2X5dwlwL6uNAKiGc6rqavklV0qK3/TpSbYXG8oPGuAFmJMGmSJ9Xr1+x-
0gr3ncsYQnHf7jjTx7y3BG+AB2G7y3uez2fXDBktjBUiAV+31N4fh99WnbOPiVkhM2lWFMeuL
1jg+vDgL/+PDziAGo2R5tgJM1w==",
    ],
    "x5t": "rU2BclI4blvOQgymzyX0GHmm_Tk",
    "x5t#S256": "Vkv7YeizaqHS23BlvZCJwTSpjpvp55J1PCmE2LtREos"
    }
    ]
    }
```

With the provided keys, the MicroProfile JWT runtime will automatically validate and authenticate each REST call to all protected resources in the application. By default, the runtime will look for the key in the HTTP Authorization header. This can be configured to something else through the Config property mp.jwt.token.header. We recommend leaving the default unless you have a very specific architectural reason to change that.

Authorization

When the client is authenticated, the MicroProfile JWT then maps the passed groups in the claims of the JWT to client roles. For each protected resource, the runtime will automatically check if the currently executing client has the role declared by the server.

Security Context Propagation

With the token verified and client authenticated, the MicroProfile JWT API provides ways of getting the raw token which can then be passed to other backend resources in a microservice cluster. For instance, when a client authenticates with Service A, and Service A depends on Service B, if both are secured with with MicroProfile JWT, Service A can get the raw token passed by the client and use that to make a call to Service B, thereby propagating the JWT to Service B.

MicroProfile JWT usage

The first step to protecting resources with MicroProfile JWT is through the use of the @LoginConfig on a Jakarta REST jakarta.ws.rs.core.Application class. The RootResourceConfiguration class below, which extends Application, is annotated with @LoginConfig to activate JWT authentication.

```
@ApplicationPath("api")
 @LoginConfig(authMethod = "MP-JWT")
 public class RootResourceConfiguration extends Application {
 }
```

The @LoginConfig annotation tells the runtime that all resources in the application should be protected and thus all client requests will need to pass a valid JWT for verification and authentication. With the configuration in place, securing a web resource is as simple as annotating the resource class or methods with @RolesAllowed. The WalletResource below shows the use of the @RolesAllowed annotation to require only authenticated users with the role teller. User roles are completely managed in Keycloak. In enterprise applications, the product owner together with the business team will determine the requisite role for each user of an application. All of this can be done fully in Keycloak. Our backend service is simply consuming what is configured in the security framework.

```
@Consumes(MediaType.APPLICATION_JSON)
 @Produces(MediaType.APPLICATION_JSON)
 @Path("wallets")
 @RolesAllowed("teller")
 public class WalletResource {
 }
```

The @RolesAllowed annotation, from the jakarta.annotation.security.RolesAllowed package, will cause a 401 HTTP status code to be returned to clients that make calls to any resource in this class without a valid token. The following shows a sample response to the resource without the requisite token and role.

```
http://localhost:3001/wallet/api/wallets

HTTP/1.1 401 Unauthorized
 WWW-Authenticate: Bearer realm="MP-JWT", error="invalid_token"
 Content-Language: en-US
 Content-Length: 0
 Date: Sun, 28 Aug 2022 00:13:24 GMT
```

Getting Information From Tokens

When a user is successfully verified and authenticated, the application might want to get hold of some information contained in the claims of the token. The API provides two primary ways of accessing the claims in the body of the JWT. These are the JsonWebToken bean and the injection of claims value using the @Claim qualifier.

JsonWebToken

The JsonWebToken interface extends the java.security.Principal and contains methods for getting the claims in the JWT. It also has a method for getting the raw token. This can be used to get the raw token for onward propagation to other services. This is my recommended approach to consuming the claims of the JWT because the JsonWebToken instance does not require the bean into which it is injected to be @RequestScoped. The following shows the injection of the JsonWebToken into the @ApplicationScoped WalletService.

```
@ApplicationScoped
 public class WalletService {

    @Inject
    JsonWebToken jsonWebToken;

    public BalanceResponse createWallet(CreateWalletRequest request){

        Wallet wallet = new Wallet();
        wallet.setBalance(BigDecimal.ZERO);

        wallet.setCurrency(request.getCurrency());
        wallet.setAuthUserId(jsonWebToken.getSubject());
        wallet = walletRepository.save(wallet);

        return getBalance(wallet.getId());
    }
}
```

The WalletService is an application-scoped bean that has the JsonWebToken injected into it. The MicroProfile JWT runtime will provide an implementation of the interface for us. The injected JsonWebToken can be used to obtain the claims of the token. The createUser method calls the getSubject method on the jsonWebToken to get the subject of the token. The subject of the token returned by Keycloak is the IAM ID of the user.

This way, we link wallet database records to the users in Keycloak without both services knowing anything about each other. The WalletService is not a Jakarta REST artefact. It is a plain CDI bean but through the MicroProfile JWT API, we are able to access the claims in the token passed through the HTTP header in the service layer of our application. We believe this is the cleanest way to access claims from the JWT and thus recommend this approach. Even if an endpoint is not secured, resulting in no token passed, an empty instance of the JsonWebToken will be created. All method invocations on the empty instance will return null.

Injecting Into Raw Types and ClaimValue

JWT claims can also be injected into simple Java types through @Inject and the use of @Claims qualifier. We generally do not recommend this way of getting the tokens because some of the field types into which claims can be injected require the beans into which they're being injected to be @RequestScoped. Effectively to be able to fully utilise injecting claim values this way, the bean should be @RequestScoped. The following shows the injection of claims using these two constructs.

```
@Action
 public class WalletService {

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

    @Inject
    @Claim(standard = Claims.raw_token)
    private ClaimValue<String> rawToken;

}
```

The subject field is a simple String type annotated @Inject and the qualifier @Claim. The standard parameter of the @Claim qualifier is passed the name of the claim to inject into this field. The Claims enum is used to pass the sub or subject claim as the claim of interest. The org.eclipse.microprofile.jwt.ClaimValue is an interface that extends the Principal interface. It has two methods, getName and getvalue. Get name returns the name of the claim, and getValue returns the value of the claim in the token.

MicroProfile JWT and the SecurityContext

The jakarta.ws.rs.core.SecurityContext interface is a Jakarta REST artefact that can be injected into resource artefacts through the @Context (deprecated since Jakarta EE 10, to be fully replaced with @Inject in Jakarta EE 11) annotation. It has methods for getting the Principal, checking if the currently logged in user is in a given role, if the request was made through a HTTPS connection and the authentication scheme. In a MicroProfile JWT application, The isUserInRole method will map the role set to the passed groups of the claims in the token. The getPrincipal method will also return a Principal, which is also an instance of JsonWebToken. Effectively the MicroProfile JWT propagates all the information passed through the token to all parts of the application and existing security infrastructure within Jakarta EE. The following shows injecting the SecurityContext into the Wallet Resource.

```
@Consumes(MediaType.APPLICATION_JSON)
 @Produces(MediaType.APPLICATION_JSON)
 @Path("wallets")
 @RolesAllowed("teller")
 public class WalletResource {

    @Context
    SecurityContext securityContext;

}
```

Summary

This guide covered the use of the MIcroProfile JWT API for securing Jakarta EE REST resources. Given the complex nature of application security, this guide followed best practices by offloading the handling of user credentials to an external service, in this case Keycloak. You learned how to activate the MicroProfile JWT in an application with the @LoginConfig annotation, and how individual resources can be secured either at the class or resource method level through the use of @RolesAllowed.

You also learned about how to access claims in a token through the JsonWebToken interface, and that JWT claims are automatically propagated and available in existing Jakarta EE security infrastructure like the Principal and SecurityContext interfaces.

Security is a very complex and fast changing domain, with new exploits breaches everyday. As such, a hybrid security approach as discussed in this guide, where the more complicated aspect like user credential handling is delegated to a specialist application like Keycloak, leaving our application to only consume the token generated is one way to mitigate security risks.