Eclipse MicroProfile Interoperable JWT RBAC

Scott Stark; David Blevins; Pedro Igor Silva; Sergey Beryozkin; Rudy De Busscher; Roberto Cortez

2.2.1, October 22, 2026: Final

Table of Contents

Copyright . . . 2
Eclipse Foundation Specification License . . . 2
Disclaimers . . . 2
Introduction . . . 4
Motivation . . . 5
Token Based Authentication . . . 5
Using JWT Bearer Tokens to Protect Services . . . 7
Recommendations for Interoperability . . . 8
Required and Recommended MP-JWT Headers and Claims . . . 8
Required MP-JWT Headers . . . 8
Recommended MP-JWT headers . . . 9
Required MP-JWT claims . . . 9
Recommended MP-JWT claims . . . 9
Additional Claims . . . 13
The `Claims` Enumeration Utility Class, and the Set of Claim Value Types . . . 14
Service Specific Authorization Claims . . . 16
Marking a JAX-RS Application as Requiring MP-JWT Access Control . . . 17
Requirements for Rejecting MP-JWT Tokens . . . 18
Mapping MP-JWT Tokens to Jakarta EE Container APIs . . . 19
CDI Injection Requirements . . . 19
Injection of `JsonWebToken` . . . 19
Injection of `JsonWebToken` claims via Raw Type, `ClaimValue`, `jakarta.enterprise.inject.Instance` and JSON-P Types . . . 19
Handling of Non-RequestScoped Injection of Claim Values . . . 23
JAX-RS Container API Integration . . . 24
`jakarta.ws.rs.core.SecurityContext.getUserPrincipal()` . . . 24
`jakarta.ws.rs.core.SecurityContext#isUserInRole(String)` . . . 24
Using the Common Security Annotations for the Java Platform (JSR-250) . . . 24
Mapping the `@RolesAllowed` to the MP-JWT group claim . . . 24
Mapping MP-JWT Token to Other Container APIs . . . 25
Signed JWT tokens . . . 26
Obtaining the Public Key . . . 26
Supported Signature Algorithms . . . 26
Supported Public Key Formats . . . 27
PKCS#8 . . . 27
JSON Web Key (JWK) . . . 28
JSON Web Key Set (JWKS) . . . 28
Signature Verification Configuration Parameters . . . 30

mp.jwt.verify.publickey . . . 30
mp.jwt.verify.publickey.location . . . 30
Relative Path . . . 30
file: URL Scheme . . . 31
http: URL Scheme . . . 31
Other URL Schemes . . . 31
mp.jwt.verify.publickey.algorithm . . . 32
Encrypted JWT claims and nested tokens . . . 33
Decryption Configuration Parameters . . . 34
mp.jwt.decrypt.key.location . . . 34
mp.jwt.decrypt.key.algorithm . . . 34
Verification of JWT token claims . . . 35
mp.jwt.verify.issuer . . . 35
mp.jwt.verify.audiences . . . 35
mp.jwt.verify.token.age . . . 35
mp.jwt.verify.clock.skew . . . 35
Requirements for accepting signed and encrypted tokens . . . 36
JWT and HTTP headers . . . 37
Configuration Properties . . . 37
mp.jwt.token.header . . . 37
mp.jwt.token.cookie . . . 37
How to provide Configuration Parameters . . . 38
Mapping Configuration Parameters to Environment Variables . . . 38
Future Directions . . . 40
resource_access claim . . . 40
roles claim . . . 40
aud claim . . . 40
Enabling/Disabling Claim Requirements . . . 40
classpath: URL Scheme . . . 41
Expiration tolerance . . . 41
Passing JWTs as Cookies . . . 41
Sample Implementations . . . 43
General Java EE/SE based Implementations . . . 43
Wildfly Swarm Implementations . . . 43
Release Notes for MicroProfile JWT 2.2 . . . 44
Spec Changes . . . 44
Release Notes for MicroProfile JWT 2.1 . . . 45
Spec Changes . . . 45
Release Notes for MicroProfile JWT 2.0 . . . 46
Incompatible Changes . . . 46
API/SPI Changes . . . 46

Other Changes . . . 46
Release Notes for MicroProfile JWT 1.2 . . . 47
API Changes . . . 47
Spec Changes . . . 47
Other Changes . . . 47
Release Notes for MicroProfile JWT 1.1.1 . . . 48
Changes in 1.1.1-RC2 . . . 48
Closed Issues in 1.1.1-RC2 . . . 48
Changes in 1.1.1-RC1 . . . 48
Closed Issues in 1.1.1-RC1 . . . 48
Release Notes for MicroProfile JWT 1.1 . . . 49

```
Specification: Eclipse MicroProfile Interoperable JWT RBAC

Version: 2.2.1

Status: Final

Release: October 22, 2026
```

Copyright

Copyright (c) 2017, 2026 Eclipse Foundation.

Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. <<url to this license>>"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) [$date-of-document] Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE

DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders. :sectnums:

Introduction

This specification outlines a proposal for using OpenID Connect (OIDC) based JSON Web Tokens (JWT) for Role Based Access Control (RBAC) of microservice endpoints.

Motivation

MicroProfile is a baseline platform definition that optimizes Enterprise Java for a microservices architecture and delivers application portability across multiple MicroProfile runtimes. While Java EE is a very feature rich platform and is like a toolbox that can be used to address a wide variety of application architectures, MicroProfile focuses on defining a small and a minimum set of Java EE standards that can be used to deliver applications based on a microservice architecture, they are:

- JAX-RS
- CDI
- JSON-P

The security requirements that involve microservice architectures are strongly related with RESTful Security. In a RESTful architecture style, services are usually stateless and any security state associated with a client is sent to the target service on every request in order to allow services to re-create a security context for the caller and perform both authentication and authorization checks.

One of the main strategies to propagate the security state from clients to services or even from services to services involves the use of security tokens. In fact, the main security protocols in use today are based on security tokens such as OAuth2, OpenID Connect, SAML, WS-Trust, WS-Federation and others. While some of these standards are more related with identity federation, they share a common concept regarding security tokens and token based authentication.

For RESTful based microservices, security tokens offer a very lightweight and interoperable way to propagate identities across different services, where:

- Services don't need to store any state about clients or users
- Services can verify the token validity if token follows a well known format. Otherwise, services may invoke a separated service.
- Services can identify the caller by introspecting the token. If the token follows a well known format, services are capable to introspect the token by themselves, locally. Otherwise, services may invoke a separated service.
- Services can enforce authorization policies based on any information within a security token
- Support for both delegation and impersonation of identities

Today, the most common solutions involving RESTful and microservices security are based on OAuth2, OpenID Connect (OIDC) and JSON Web Tokens (JWT) standards.

Token Based Authentication

Token Based Authentication mechanisms allow systems to authenticate, authorize and verify identities based on a security token. Usually, the following entities are involved:

- Issuer
    - Responsible for issuing security tokens as a result of successfully asserting an identity

(authentication). Issuers are usually related with Identity Providers.

- Client
  - Represented by an application to which the token was issued for. Clients are usually related with Service Providers. A client may also act as an intermediary between a subject and a target service (delegation).
- Subject
  - The entity to which the information in a token refers to.
- Resource Server
  - Represented by an application that is going to actually consume the token in order to check if a token gives access or not to a protected resource.

Independent of the token format or protocol in use, from a service perspective, token based authentication is based on the following steps:

- Extract security token from the request
  - For RESTful services, this is usually achieved by obtaining the token from the Authorization header.
- Perform validation checks against the token
  - This step usually depends on the token format and security protocol in use. The objective is make sure the token is valid and can be consumed by the application. It may involve signature, encryption and expiration checks.
- Introspect the token and extract information about the subject
  - This step usually depends on the token format and security protocol in use. The objective is to obtain all the necessary information about the subject from the token.
- Create a security context for the subject
  - Based on the information extracted from the token, the application creates a security context for the subject in order to use the information wherever necessary when serving protected resources.

Using JWT Bearer Tokens to Protect Services

For now, use cases are based on a scenario where services belong to the same security domain. This is an important note in order to avoid dealing with all complexities when you need to access services across different security domains. With that in mind, we assume that any information carried along with a token could be understood and processed (without any security breaches) by the different services involved.

The use case can be described as follows:

A client sends a HTTP request to Service A including the JWT as a bearer token:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

On the server, a token-based authentication mechanism in front of Service A perform all steps described on the Token Based Authentication section. As part of the security context creation, the server establishes role and group mappings for the subject based on the JWT claims. The role to group mapping is fully configurable by the server along the lines of the Java EE RBAC security model.

JWT tokens follow a well defined and known standard that is becoming the most common token format to protect services. It not only provides a token format but additional security aspects like signature and encryption based on another set of standards like JSON Web Signature (JWS), JSON Web Encryption (JWE) and others.

There are few reasons why JWT is becoming so widely adopted:

- Token validation doesn't require an additional trip and can be validated locally by each service
- Given its JSON nature, it is solely based on claims or attributes to carry authentication and authorization information about a subject.
- Makes easier to support different types of access control mechanisms such as ABAC, RBAC, Context-Based Access Control, etc.
- Message-level security using signature and encryption as defined by both JWS and JWE standards
- Given its JSON nature, processing JWT tokens becomes trivial and lightweight. Especially if considering Java EE standards such as JSON-P or the different third-party libraries out there such as Nimbus, Jackson, etc.
- Parties can easily agree on a specific set of claims in order to exchange both authentication and authorization information. Defining this along with the Java API and mapping to JAX-RS APIs are the primary tasks of the MP-JWT specification.
- Widely adopted by different Single Sign-On solutions and well known standards such as OpenID Connect given its small overhead and ability to be used across different security domains (federation)

Recommendations for Interoperability

The maximum utility of the MicroProfile JWT(MP-JWT) as a token format depends on the agreement between both identity providers and service providers. This means identity providers - responsible for issuing tokens - should be able to issue tokens using the MP-JWT format in a way that service providers can understand in order to introspect the token and gather information about a subject. To that end, the requirements for the MicroProfile JWT are:

1. Be usable as an authentication token.
2. Be usable as an authorization token that contains Jakarta EE application level roles indirectly granted via a groups claim.
3. Can be mapped to IdentityStore in JSR375.
4. Can support additional standard claims described in IANA JWT Assignments as well as non-standard claims.

To meet those requirements, we introduce 2 new claims to the MP-JWT:

- "upn": A human readable claim that uniquely identifies the subject or user principal of the token, across the MicroProfile services the token will be accessed with.
- "groups": A list of group names to be assigned to the principal of the MP-JWT. This typically will require a mapping at the application container level to application roles. A one-to-one mapping between group names and application role names is done by default when no custom mapping is defined in an implementation specific way.

Required and Recommended MP-JWT Headers and Claims

Required MP-JWT Headers

alg

This JOSE header parameter identifies the cryptographic algorithm used to secure the JWT.

- RSASSA-PKCS1-v1_5 SHA-256 or ECDSA using P-256 and SHA-256 algorithm are required when the claims have to be signed and must be specified as either "RS256", RFC7518, Section 3.3 or "ES256", RFC7515, Section 3.4.
- Support for RSAES using Optimal Asymmetric Encryption Padding algorithm and RSAES OAEP using SHA-256 and MGF1 with SHA-256 is required when the claims or nested JWT tokens have to be encrypted. These algorithms are used for encrypting the content encryption key and must be specified as either `RSA-OAEP` or `RSA-OAEP-256`, RFC7518, Section 4.3. Support for the other key management key algorithms such as `RSA-OAEP-384`, `RSA-OAEP-512` and others is optional.

enc

This JOSE header parameter is only required when the claims or nested JWT tokens have to be encrypted. It identifies the cryptographic algorithm used to encrypt the claims or nested JWT

tokens. AES in Galois/Counter Mode (GCM) algorithm is required and must be specified as "A256GCM", RFC7518, Section 5.3

Recommended MP-JWT headers

typ

This JOSE header parameter identifies the token as an RFC7519 and should be "JWT". RFC7519, Section 5.1

kid

This JOSE header parameter is a hint indicating which key was used to secure the JWT. RFC7515, Section-4.1.4. It may need to be set if the verification key is in the JWK format.

Required MP-JWT claims

iss

Identifies the token issuer

iat

Identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value MUST be a number containing a NumericDate value. RFC7519, Section 4.1.6

exp

Identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. The processing of the "exp" claim requires that the current date/time MUST be before the expiration date/time listed in the "exp" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. RFC7519, Section 4.1.4

upn (or preferred_username or sub)

MP-JWT "upn" claim is the user principal name in the java.security.Principal interface, and is the caller principal name in jakarta.security.enterprise.identitystore.IdentityStore. If this claim is missing, fallback to the "preferred_username", OIDC Section 5.1 should be attempted, and if that claim is missing, fallback to the "sub" claim should be used.

Recommended MP-JWT claims

sub

Uniquely identifies the principal that is the subject of the JWT. See the "upn" claim for how this relates to the container java.security.Principal. RFC7519, Section 4.1.2.

jti

Provides a unique identifier for the JWT. The identifier value MUST be assigned in a manner that ensures that there is a negligible probability that the same value will be accidentally assigned to a different data object; if the application uses multiple issuers, collisions MUST be prevented among values produced by different issuers as well. The "jti" claim can be used to prevent the JWT from being replayed. The "jti" value is a case-sensitive string. RFC7519, Section 4.1.7

aud

Identifies the MP JWT endpoint(s) which can be accessed by JWT. RFC7519, Section 4.1.3

[NOTE] MP JWT implementations may enforce that JWT tokens contain all the recommended headers and claims. The recommended headers and claims may become required in the future versions of the MP JWT specification.

[NOTE] It is recommended that JWT tokens have a `groups` claim if the endpoint requires Authorization but MP JWT implementations can map the groups from the other claims if the tokens have been issued by OpenId Connect and other providers which currently do not support MP JWT.

If no groups information can be extracted directly from the `groups` claim or via the custom mappers from other custom claims in a given token then this token can be accepted if the endpoint requires Authentication only.

NOTE	NumericDate used by `exp`, `iat`, and other date related claims is a JSON numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the specified UTC date/time, ignoring leap seconds

An example minimal MP-JWT in JSON would be:

```
{
     "typ": "JWT",
    "alg": "RS256",
    "kid": "abc-1234567890"
}
{
       "iss": "https://server.example.com",
       "jti": "a-123",
       "exp": 1311281970,
       "iat": 1311280970,
       "sub": "24400320",
       "upn": "jdoe@server.example.com",
       "groups": ["red-group", "green-group", "admin-group", "admin"],
}
```

This specification defines a JsonWebToken java.security.Principal interface extension that makes this set of required claims available via get style accessors. The JsonWebToken interface definition is:

```
package org.eclipse.microprofile.jwt;
public interface JsonWebToken extends Principal {

    /**
     * Returns the unique name of this principal. This either comes from the upn
     * claim, or if that is missing, the preferred_username claim. Note that for
     * guaranteed interoperability a upn claim should be used.
     *
```

```
     * @return the unique name of this principal.
     */
    @Override
    String getName();

    /**
     * Get the raw bearer token string originally passed in the authentication
     * header
     * @return raw bear token string
     */
    default String getRawToken() {
        return getClaim(Claims.raw_token.name());
    }

    /**
     * The iss(Issuer) claim identifies the principal that issued the JWT
     * @return the iss claim.
     */
    default String getIssuer() {
        return getClaim(Claims.iss.name());
    }

    /**
     * The aud(Audience) claim identifies the recipients that the JWT is
     * intended for.
     * @return the aud claim.
     */
    default Set<String> getAudience() {
        return getClaim(Claims.aud.name());
    }

    /**
     * The sub(Subject) claim identifies the principal that is the subject of
     * the JWT. This is the token issuing IDP subject.
     *
     * @return the sub claim.
     */
    default String getSubject() {
        return getClaim(Claims.sub.name());
    }

    /**
     * The jti(JWT ID) claim provides a unique identifier for the JWT.
     The identifier value MUST be assigned in a manner that ensures that
     there is a negligible probability that the same value will be
     accidentally assigned to a different data object; if the application
     uses multiple issuers, collisions MUST be prevented among values
     produced by different issuers as well.  The "jti" claim can be used
     to prevent the JWT from being replayed.
     * @return the jti claim.
     */
```

```
    default String getTokenID() {
        return getClaim(Claims.jti.name());
    }

    /**
     * The exp (Expiration time) claim identifies the expiration time on or
     * after which the JWT MUST NOT be accepted
     * for processing in seconds since 1970-01-01T00:00:00Z UTC
     * @return the exp claim.
     */
    default long getExpirationTime() {
        return getClaim(Claims.exp.name());
    }

    /**
     * The iat(Issued at time) claim identifies the time at which the JWT was
     * issued in seconds since 1970-01-01T00:00:00Z UTC
     * @return the iat claim
     */
    default long getIssuedAtTime() {
        return getClaim(Claims.iat.name());
    }

    /**
     * The groups claim provides the group names the JWT principal has been
     * granted.
     *
     * This is a MicroProfile specific claim.
     * @return a possibly empty set of group names.
     */
    default Set<String> getGroups() {
        return getClaim(Claims.groups.name());
    }

    /**
     * Access the names of all claims are associated with this token.
     * @return non-standard claim names in the token
     */
    Set<String> getClaimNames();

    /**
     * Verify is a given claim exists
     * @param claimName - the name of the claim
     * @return true if the JsonWebToken contains the claim, false otherwise
     */
    default boolean containsClaim(String claimName) {
        return claim(claimName).isPresent();
    }

    /**
     * Access the value of the indicated claim.
```

```
     * @param claimName - the name of the claim
     * @return the value of the indicated claim if it exists, null otherwise.
     */
    <T> T getClaim(String claimName);

    /**
     * A utility method to access a claim value in an {@linkplain Optional}
     * wrapper
     * @param claimName - the name of the claim
     * @param <T> - the type of the claim value to return
     * @return an Optional wrapper of the claim value
     */
    default <T> Optional<T> claim(String claimName) {
        return Optional.ofNullable(getClaim(claimName));
    }

    /**
     * A utility method to access a claim value in an {@linkplain Optional}
     * wrapper
     * @param claim - the claim
     * @param <T> - the type of the claim value to return
     * @return an Optional wrapper of the claim value
     */
    default <T> Optional<T> claim(Claims claim) {
        return claim(claim.name());
    }
}
```

Additional Claims

The JWT can contain any number of other custom and standard claims, and these are made available from the JsonWebToken getClaim(String) method. An example MP-JWT that contains additional "auth_time", "preferred_username", "acr", "nbf", "aud" and "roles" claims is:

```
{
     "typ": "JWT",
    "alg": "RS256",
    "kid": "abc-1234567890"
}
{
   "iss": "https://server.example.com",
   "aud": ["s6BhdRkqt3"],
   "exp": 1311281970,
   "iat": 1311280970,
   "sub": "24400320",
   "upn": "jdoe@server.example.com",
   "groups: ["red-group", "green-group", "admin-group"],
   "roles": ["auditor", "administrator"],
   "jti": "a-123",
```

```
   "auth_time": 1311280969,
   "preferred_username": "jdoe",
   "acr": "phr",
   "nbf":  1311288970
}
```

The `Claims` Enumeration Utility Class, and the Set of Claim Value Types

The `org.eclipse.microprofile.jwt.Claims` utility class encapsulate an enumeration of all the standard JWT related claims along with a description and the required Java type for the claim as returned from the `JsonWebToken#getClaim(String)` method.

```
public enum Claims {
    // The base set of required claims that MUST have non-null values in the
JsonWebToken
    iss("Issuer", String.class),
    sub("Subject", String.class),
    exp("Expiration Time", Long.class),
    iat("Issued At Time", Long.class),
    jti("JWT ID", String.class),
    upn("MP-JWT specific unique principal name", String.class),
    groups("MP-JWT specific groups permission grant", Set.class),
    raw_token("MP-JWT specific original bearer token", String.class),

    // The IANA registered, but MP-JWT optional claims
    aud("Audience", Set.class),
    nbf("Not Before", Long.class),
    auth_time("Time when the authentication occurred", Long.class),
    updated_at("Time the information was last updated", Long.class),
    azp("Authorized party - the party to which the ID Token was issued", String.class
),
    nonce("Value used to associate a Client session with an ID Token", String.class),
    at_hash("Access Token hash value", Long.class),
    c_hash("Code hash value", Long.class),

    full_name("Full name", String.class),
    family_name("Surname(s) or last name(s)", String.class),
    middle_name("Middle name(s)", String.class),
    nickname("Casual name", String.class),
    given_name("Given name(s) or first name(s)", String.class),
    preferred_username("Shorthand name by which the End-User wishes to be referred to
", String.class),
    email("Preferred e-mail address", String.class),
    email_verified("True if the e-mail address has been verified; otherwise false",
Boolean.class),

    gender("Gender", String.class),
```

```
    birthdate("Birthday", String.class),
    zoneinfo("Time zone", String.class),
    locale("Locale", String.class),
    phone_number("Preferred telephone number", String.class),
    phone_number_verified("True if the phone number has been verified; otherwise
false", Boolean.class),
    address("Preferred postal address", JsonObject.class),
    acr("Authentication Context Class Reference", String.class),
    amr("Authentication Methods References", String.class),
    sub_jwk("Public key used to check the signature of an ID Token", JsonObject.class
),
    cnf("Confirmation", String.class),
    sip_from_tag("SIP From tag header field parameter value", String.class),
    sip_date("SIP Date header field value", String.class),
    sip_callid("SIP Call-Id header field value", String.class),
    sip_cseq_num("SIP CSeq numeric header field parameter value", String.class),
    sip_via_branch("SIP Via branch header field parameter value", String.class),
    orig("Originating Identity String", String.class),
    dest("Destination Identity String", String.class),
    mky("Media Key Fingerprint String", String.class),

    jwk("JSON Web Key Representing Public Key", JsonObject.class),
    jwe("Encrypted JSON Web Key", String.class),
    kid("Key identifier", String.class),
    jku("JWK Set URL", String.class),

    UNKNOWN("A catch all for any unknown claim", Object.class)
    ;
...
    /**
     * @return A desccription for the claim
     */
    public String getDescription() {
        return description;
    }

    /**
     * The required type of the claim
     * @return type of the claim
     */
    public Class<?> getType() {
        return type;
    }
}
```

Note that the `groups` and `aud` claims should only be injected into a Set of String, not as a comma delimited string.

The current complete set of valid claim types is therefore, (excluding the invalid Claims.UNKNOWN Void type):

- java.lang.String
- java.lang.Long and long
- java.lang.Boolean and boolean
- java.util.Set<java.lang.String>
- jakarta.json.JsonValue.TRUE/FALSE
- jakarta.json.JsonString
- jakarta.json.JsonNumber
- jakarta.json.JsonArray
- jakarta.json.JsonObject

Custom claims not handled by the Claims enum must be any of the types defined in the previous acceptable claim types list.

Service Specific Authorization Claims

An extended form of authorization on a per service basis using a "resource_access" claim has been postponed to a future release. See Future Directions for more information.

Marking a JAX-RS Application as Requiring MP-JWT Access Control

Since the MicroProfile does not specify a deployment format, and currently does not rely on servlet metadata descriptors, we have added an `org.eclipse.microprofile.auth.LoginConfig` annotation that provides the same information as the web.xml login-config element. It's intended usage is to mark a JAX-RS `Application` as requiring MicroProfile JWT RBAC as shown in the following sample:

```
import org.eclipse.microprofile.auth.LoginConfig;

import jakarta.ws.rs.ApplicationPath;
import jakarta.ws.rs.core.Application;

@LoginConfig(authMethod = "MP-JWT", realmName = "TCK-MP-JWT")
@ApplicationPath("/")
public class TCKApplication extends Application {
}
```

The MicroProfile JWT implementation is responsible for either directly processing this annotation, or mapping it to an equivalent form of metadata for the underlying implementation container.

Requirements for Rejecting MP-JWT Tokens

The MP-JWT specification requires that an MP-JWT implementation reject a JWT token as an invalid MP-JWT token if any of the following conditions are not met:

1. The JWT must have a JOSE "alg" header that indicates the token was signed using the RS256 or ES256 algorithm when the service endpoint expects signed tokens.
2. The JWT must have the JOSE "alg" and "enc" headers that indicate that the token was encrypted using the `RSA-OAEP` or `RSA-OAEP-256` key management algorithms (or other key management algorithms supported by the implementation) and `A256GCM` content encryption algorithm when the service endpoint expects encrypted tokens. If the encrypted content is a nested signed JWT token then it must meet the condition 1.
3. The JWT must have an "iss" claim representing the token issuer that maps to an MP-JWT implementation container runtime configured value. Any issuer other than those issuers that have been whitelisted by the container configuration must be rejected with an HTTP_UNAUTHENTICATED(401) error.
4. The JWT must have an "iat" claim representing the token issuance time. It is impossible to calculate how long the token may have been used for without the "iat" claim so such tokens must be rejected with an HTTP_UNAUTHENTICATED(401) error.
5. The JWT must have an "exp" claim representing the token expiration time. Tokens without the "exp" claim have an unlimited lifetime and must be rejected with an HTTP_UNAUTHENTICATED(401) error.
6. The token must have at least one of "upn" or "preferred_username" or "sub" claim for Java Principal to have a name. Tokens from which no Principal name can be deduced must be rejected with an HTTP_UNAUTHENTICATED(401) error.
7. The signed JWT token verification and encrypted JWT token decryption failures must lead to an HTTP_UNAUTHENTICATED(401) error.

Mapping MP-JWT Tokens to Jakarta EE Container APIs

The requirements of how a JWT should be exposed via the various Jakarta EE container APIs is discussed in this section. For the 1.0 release, the only mandatory container integration is with the JAX-RS container, and injection of the MP-JWT types.

CDI Injection Requirements

This section describes the requirements for MP-JWT implementations with regard to the injection of MP-JWT tokens and their associated claim values.

Injection of `JsonWebToken`

An MP-JWT implementation must support the injection of the currently authenticated caller as a `JsonWebToken` with `@RequestScoped` scoping which must work even if the outer bean is `@ApplicationScoped`:

```
@Path("/endp")
@DenyAll
@ApplicationScoped
public class RolesEndpoint {

    @Inject
    private JsonWebToken callerPrincipal;
```

If there is no JWT in the request, an empty `JsonWebToken` is injected, which means all method calls to this token return `null`. Note that MP JWT will still perform Authentication and Authorization if the endpoint requires these verifications. Effectively, the injected empty token is only visible on unauthenticated and unauthorized endpoints.

If a JWT is sent to an endpoint that does not require Authentication and/or Authorization then it still must be verified before it can be accessed via `JsonWebToken` interface.

Endpoints which need to control the authentication process themselves can check if a token is available by calling a `JsonWebToken.getRawToken()` method.

Injection of `JsonWebToken` claims via Raw Type, `ClaimValue`, `jakarta.enterprise.inject.Instance` and JSON-P Types

This specification requires support for injection of claims from the current `JsonWebToken` using the `org.eclipse.microprofile.jwt.Claim` qualifier:

```
/**
 * Annotation used to signify an injection point for a {@link ClaimValue} from
 * a {@link JsonWebToken}
```

```
 */
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType
.TYPE})
public @interface Claim {
    /**
     * The value specifies the id name the claim to inject
     * @return the claim name
     * @see JsonWebToken#getClaim(String)
     */
    @Nonbinding
    String value() default "";

    /**
     * An alternate way of specifying a claim name using the {@linkplain Claims}
     * enum
     * @return the claim enum
     */
    @Nonbinding
    Claims standard() default Claims.UNKNOWN;
}
```

with `@Dependent` scoping.

MP-JWT implementations are required to throw a `DeploymentException` when detecting the ambiguous use of a `@Claim` qualifier that includes inconsistent non-default values for both the value and standard elements as is the case shown here:

```
@ApplicationScoped
public class MyEndpoint {
    @Inject
    @Claim(value="exp", standard=Claims.iat)
    private Long timeClaim;
...
}
```

The current complete set of valid claim types is:

- java.lang.String
- java.lang.Long and long
- java.lang.Boolean and boolean
- java.util.Set<java.lang.String>
- jakarta.json.JsonValue.TRUE/FALSE
- jakarta.json.JsonString
- jakarta.json.JsonNumber

- jakarta.json.JsonArray
- jakarta.json.JsonObject
- java.util.Optional wrapper of the above types.
- org.eclipse.microprofile.jwt.ClaimValue wrapper of the above types.

MP-JWT implementations are required to support injection of the claim values using any of these types. The claims are automatically converted to the type used in the injection point where the type must be any of the types defined in the previous acceptable claim types list.

The `org.eclipse.microprofile.jwt.ClaimValue` interface is:

```
/**
 * A representation of a claim in a {@link JsonWebToken}
 * @param <T> the expected type of the claim
 */
public interface ClaimValue<T> extends Principal {

    /**
     * Access the name of the claim.
     * @return The name of the claim as seen in the JsonWebToken content
     */
    @Override
    public String getName();

    /**
     * Access the value of the claim.
     * @return the value of the claim.
     */
    public T getValue();
}
```

The following example code fragment illustrates various examples of injecting different types of claims using a range of generic forms of the `ClaimValue`, `JsonValue` as well as the raw claim types:

```
import org.eclipse.microprofile.jwt.Claim;
import org.eclipse.microprofile.jwt.ClaimValue;
import org.eclipse.microprofile.jwt.Claims;

@Path("/endp")
@DenyAll
@RequestScoped
public class RolesEndpoint {
...

    // Raw types
    @Inject
    @Claim(standard = Claims.raw_token)
    private String rawToken;
```

```
    @Inject ①
    @Claim(standard=Claims.iat)
    private Long issuedAt;

    // ClaimValue wrappers
    @Inject ②
    @Claim(standard = Claims.raw_token)
    private ClaimValue<String> rawTokenCV;
    @Inject
    @Claim(standard = Claims.iss)
    private ClaimValue<String> issuer;
    @Inject
    @Claim(standard = Claims.jti)
    private ClaimValue<String> jti;
    @Inject ③
    @Claim("jti")
    private ClaimValue<Optional<String>> optJTI;
    @Inject
    @Claim("jti")
    private ClaimValue objJTI;
    @Inject
    @Claim("groups")
    private ClaimValue<Set<String>> groups;
    @Inject ④
    @Claim(standard=Claims.iat)
    private ClaimValue<Long> issuedAtCV;
    @Inject
    @Claim("iat")
    private ClaimValue<Long> dupIssuedAt;
    @Inject
    @Claim("sub")
    private ClaimValue<Optional<String>> optSubject;
    @Inject
    @Claim("auth_time")
    private ClaimValue<Optional<Long>> authTime;
    @Inject ⑤
    @Claim("custom-missing")
    private ClaimValue<Optional<Long>> custom;
    //
    @Inject
    @Claim(standard = Claims.jti)
    private Instance<String> providerJTI;
    @Inject ⑥
    @Claim(standard = Claims.iat)
    private Instance<Long> providerIAT;
    @Inject
    @Claim("groups")
    private Instance<Set<String>> providerGroups;
    //
    @Inject
    @Claim(standard = Claims.jti)
```

```
    private JsonString jsonJTI;
    @Inject
    @Claim(standard = Claims.iat)
    private JsonNumber jsonIAT;
    @Inject ⑦
    @Claim("roles")
    private JsonArray jsonRoles;
    @Inject
    @Claim("customObject")
    private JsonObject jsonCustomObject;
```

① Injection of a non-proxyable raw type like java.lang.Long must happen in a RequestScoped bean as the producer will have dependendent scope.

② Injection of the raw MP-JWT token string.

③ Injection of the jti token id as an `Optional<String>` wrapper.

④ Injection of the issued at time claim using an `@Claim` that references the claim name using the Claims.iat enum value.

⑤ Injection of a custom claim that does exist will result in an Optional<Long> value for which isPresent() will return false.

⑥ Another injection of a non-proxyable raw type like java.lang.Long, but the use of the jakarta.enterprise.inject.Instance interface allows for injection to occur in non-RequestScoped contexts.

⑦ Injection of a JsonArray of role names via a custom "roles" claim.

The example shows that one may specify the name of the claim using a string or a `Claims` enum value. The string form would allow for specifying non-standard claims while the `Claims` enum approach guards against typos.

Handling of Non-RequestScoped Injection of Claim Values

MP-JWT implementations are required to support a claim value injection into @ApplicationScoped scoped beans. A warning may be issued when the injection point is not an `org.eclipse.microprofile.jwt.ClaimValue` or `jakarta.inject.Provider` interface.

MP-JWT implementations are required to generate a `jakarta.enterprise.inject.spi.DeploymentException` for a claim value injection into Passivation capable beans, for example, @SessionScoped.

MP JWT implementations may issue a warning for any other context with a lifetime greater than @RequestScoped.

NOTE	If one needs to inject a claim value into a scope with a lifetime greater than @RequestScoped, such as @ApplicationScoped or @SessionScoped, one can also use the jakarta.enterprise.inject.Instance interface to do so.

JAX-RS Container API Integration

The behavior of the following JAX-RS security related methods is required for MP-JWT implementations.

`jakarta.ws.rs.core.SecurityContext.getUserPrincipal()`

The `java.security.Principal` returned from these methods MUST be an instance of `org.eclipse.microprofile.jwt.JsonWebToken`.

`jakarta.ws.rs.core.SecurityContext#isUserInRole(String)`

This method MUST return true for any name that is included in the MP-JWT "groups" claim, as well as for any role name that has been mapped to a group name in the MP-JWT "groups" claim.

Using the Common Security Annotations for the Java Platform (JSR-250)

The expectations for use of the various security annotations described in sections 2.9 - 2.12 of JSR-250 (@RolesAllowed, @PermitAll, @DenyAll), is that MP-JWT containers support the behavior as described in those sections. In particular, the interaction between the annotations should be as described in section 2.12 of JSR-250.

Mapping the `@RolesAllowed` to the MP-JWT group claim

In terms of mapping between the MP-JWT claims and role names used in @RolesAllowed, the role names that have been mapped to group names in the MP-JWT "groups" claim, MUST result in an allowing authorization decision wherever the security constraint has been applied.

Mapping MP-JWT Token to Other Container APIs

For non-Jakarta EE containers that provide access to some form of `java.security.Principal` representation of an authenticated caller, the caller principal MUST be compatible with the `org.eclipse.microprofile.jwt.JsonWebToken` interface.

Signed JWT tokens

In many cases, Json Web Tokens (JWT) are created by signing a JSON representation of the token claims by following the steps described in the JSON Web Signature(JWS) specification.

The signed JWT token itself can also be encrypted (thus becoming an inner nested token). In this case it will need to be decrypted first. Please see Encrypted JWT claims and nested tokens for more information.

Verification of JWT passed to the Microservice in HTTP requests at runtime is done with the Public Key corresponding to the Private Key held by the JWT Issuer.

At the time of JWT creation, the Issuer will sign the JWT with its Private Key before passing it to the user. Upon receiving the JWT in future HTTP requests, Microservices can then use the matching Public Key to verify the JWT and trust the user information (claims) it contains.

The goal of this chapter is to detail means of passing the Public Key from the JWT Issuer to the MicroProfile JWT implementation.

Obtaining the Public Key

In practice, the Public Key is often obtained manually from the JWT Issuer and stored in or passed to the binary of the Microservice. If your public Keys do not rotate frequently, then storing them in the binary image or on disk is a realistic option for many environments. For reference, SSL/TLS Certificates to support HTTPS, which are also Public Key based, are usually configured in the JVM itself and last for up to two years.

Alternatively, Public Keys may be obtained by the Microservice at runtime, directly from the JWT Issuer via HTTPS request. MicroProfile JWT implementations are required to support this method of fetching the Public Key from the JWT Issuer via means defined here. It should be noted, however, not all JWT Issuers support downloading of the Public Key via HTTPS request.

Supported Signature Algorithms

Support for RSA RS256 and Elliptic Curve Digital Signature Algorithm (ECDSA) ES256 is required. RSA keys used for creating and verifying RS256 signatures must be of 1024 or 2048 bits in length. Other RSA key sizes are allowed, but should be considered vendor-specific.

[NOTE] Support for RSA keys of 1024 bits in length is deprecated and will become optional in the next major version of the MP JWT specification.

Other asymmetric signature algorithms are allowed, but should be considered vendor-specific. This includes Digital Signature Algorithm (DSA), Diffie-Hellman (DS), Edwards-curve Digital Signature Algorithm (EdDSA, aka ed25519).

NOTE	Symmetrically signed JWTs such as HMAC-SHA256 (hs256) are explicitly not supported, deemed insecure for a distributed Microservice architecture where JWTs are expected to be passed around freely. Use of symmetric signatures would

> require all microservices to share a secret, eliminating the ability to determine who created the JWT.

Supported Public Key Formats

RSA and ECDSA Public Keys may be formatted in any of the following formats, specified in order of precedence:

- Public Key Cryptography Standards #8 (PKCS#8) PEM
- JSON Web Key (JWK)
- JSON Web Key Set (JWKS)
- JSON Web Key (JWK) Base64 URL encoded
- JSON Web Key Set (JWKS) Base64 URL encoded

Attempts to parse the Public Key text will proceed in the order specified above until a valid Public Key can be derived.

Support for other Public Key formats such as PKCS#1, SSH2, or OpenSSH Public Key format is considered optional.

MicroProfile JWT implementations are required to throw a `DeploymentException` when given a public key that cannot be parsed using either the standardly supported or vendor-specific key formats.

MicroProfile JWT implementations are required to throw a `DeploymentException` when given a Private Key in any format.

PKCS#8

Public Key Cryptography Standards #8 (PKCS#8) PEM format is a plain text format and is the default format for OpenSSL, many public/private key tools and is natively supported in Java.

The format consists of a Base64 URL encoded value wrapped in a standard `-----BEGIN PUBLIC KEY-----` header and footer. The Base64 URL encoded data can be decoded and the resulting byte array passed directly to `java.security.spec.PKCS8EncodedKeySpec`.

The following is an example of a valid RSA 2048 bit Public Key in PKCS#8 PEM format.

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0440JtmhlywtkMvR6tTM
s0U6e9Ja4xXj5+q+joWdT2xCHt91Ck9+5C5WOaRTco4CPFMBxoUPi1jktW5c+Oyk
nOIACXu6grXexarFQLjsREE+dkDVrMu75f7Gb9/lC7mrVM73118wnMP2u5MOQIoX
OqqC1y1gaoJaLp/OjTiJGCm4uxzubzUPN5IDAFaTfK+QErhtcGeBDwWjvikGfUfX
+WVq74DOoggLiGbB4jsT8iVXEm53JcoEY8nVr2ygr92TuU1+xLAGisjRSYJVe7V1
tpdRG1CiyCIkqhDFfFBGhFnWlu4gKMiT0KToA9GJfOuCz67XZEAhQYizcXbn1uxa
OQIDAQAB
-----END PUBLIC KEY-----
```

MicroProfile JWT implementations must inspect the supplied Public Key body for the `-----BEGIN PUBLIC KEY-----` header and parse the key as PKCS#8 if found.

Support for the legacy PKCS#1 format is not required and should be considered vendor-specific. PKCS#1 formatted keys can be identified by the use of the `-----BEGIN RSA PUBLIC KEY-----`.

MicroProfile JWT implementations are required to throw a `DeploymentException` if a Private Key is supplied.

JSON Web Key (JWK)

JSON Web Key (JWK) allows for a Public Key to be formatted in json and optionally Base64 encoded.

At minimum JWK formatted RSA Public Keys must contain the `kty` field set to "RSA" as well as the `n` and `e` fields. At minimum JWK formatted ECDSA Public Keys must contain the `kty` field set to "EC", `crv` field set to "P-256" as well as the `x` and `y` fields.

The following example is the previously shown PKCS#8 PEM formatted Public Key converted to JWK format.

```
{
  "kty": "RSA",
  "n": "sszbq1NfZap2IceUCO9rCF9ZYfHE3oU5m6Avgyxu1LmlB6rNPejO-
eB7T9iIhxXCEKsGDcx4Cpo5nxnW5PSQZM_wzXg1bAOZ3O6k57EoFC108cB0hdvOiCXXKOZGrGiZuF7q5Zt1ftq
Ik7oK2gbItSdB7dDrR4CSJSGhsSu5mP0",
  "e": "AQAB"
}
```

The next example shows the ECDSA Public Key:

```
{
  "kty":"EC",
  "crv":"P-256",
  "x":"w4HohvwOj21FBQE1PrJOAlPRQMyWimmXH9rIHa7YMTU",
  "y":"osZEjUhZa79-kClcGm79eX0q_QFLlrA99MhkzNy6MtI"
}
```

MicroProfile JWT implementations are required to throw a `DeploymentException` if the JWK `kty` field is missing or JSON text is found, but does not follow either JWK or JWKS format.

The JWK may be supplied in plain JSON or Base64 URL encoded JSON format.

See RFC-7517 for further details on JWK format and optional fields.

JSON Web Key Set (JWKS)

The JSON Web Key Set (JWKS) format allows for multiple keys to supplied, which can be useful for either key rotation or supporting environments that have multiple JWT Issuers and therefore

multiple Public Keys.

An example of a valid JWKS:

```
{
  "keys": [
    {
      "kid": "orange-1234",
      "kty": "RSA",
      "n": "sszbq1NfZap2IceUCO9rCF9ZYfHE3oU5m6Avgyxu1LmlB6rNPejO-
eB7T9iIhxXCEKsGDcx4Cpo5nxnW5PSQZM_wzXg1bAOZ3O6k57EoFC108cB0hdvOiCXXKOZGrGiZuF7q5Zt1ftq
Ik7oK2gbItSdB7dDrR4CSJSGhsSu5mP0",
      "e": "AQAB"
    },
    {
      "kid": "orange-5678",
      "kty": "RSA",
      "n":
"xC7RfPpTo7362rzATBu45Jv0updEZcr3IqymjbZRkpgTR8B19b_rS4dIficnyyU0plefkE2nJJyJbeW3Fon9B
Le4_srfXtqiBKcyqINeg0GrzIqoztZBmmmdo13lELSrGP91oHL-
UtCd1u5C1HoJc4bLpjUYxqOrJI4mmRC3Ksk5DV2OS1L5P4nBWIcR1oi6RQaFXy3zam3j1TbCD5urkE1CfUATFw
fXfFSPTGo7shNqsgaWgy6B2051l5Lq5UmMUBG0prK79ymjJemODwrB445z-lk3CTtlMN7bcQ3nC8xh-
Mb2XmRB0uoU4K3kHTsofXG4dUHWJ8wGXEXgJNOPzOQ",
      "e": "AQAB"
    }
  ]
}
```

If the incoming JWT uses the `kid` header field and there is a key in the supplied JWK set with the same `kid`, only that key is considered for verification of the JWT's digital signature.

For example, the following decoded JWT would involve a check on only the `orange-5678` key.

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "orange-5678"
}.
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

The JWKS may be supplied in plain JSON or Base64 URL encoded JSON format.

Signature Verification Configuration Parameters

See the Verification of JWT token claims section how to verify the token claims once its signature has been verified.

mp.jwt.verify.publickey

The `mp.jwt.verify.publickey` configuration property allows the Public Verification Key text itself to be supplied as a string. The Public Key will be parsed from the supplied string in the order defined in section Supported Public Key Formats.

The following example shows a Base 64 URL encoded JWK passed via system property.

```
java -jar movieservice.jar -Dmp.jwt.verify.publickey=eyJrdHkiOiJSU0EiLCJuI\
joieEM3UmZQcFRvNzM2MnJ6QVRCdTQ1SnYwdXBkRVpjcjNJcXltamJaUmtwZ1RSOEIxOWJfclM\
0ZElmaWNueXlVMHBsZWZrRTJuSkp5SmJlVzNGb245QkxlNF9zcmZYdHFpQktjeXFJTmVnMEdye\
klxb3p0WkJtbW1kbzEzbEVMU3JHUDkxb0hMLVV0Q2QxdTVDMUhvSmM0YkxwalVZeHFPckpJNG1\
tUkMzS3NrNURWMk9TMUw1UDRuQldJY1Ixb2k2UlFhRlh5M3phbTNqMVRiQ0Q1dXJrRTFDZlVBV\
EZ3ZlhmRlNQVEdvN3NoTnFzZ2FXZ3k2QjIwNWw1THE1VW1NVUJHMHBySzc5eW1qSmVtT0R3ckI\
0NDV6LWxrM0NUdGxNTjdiY1EzbkM4eGgtTWIyWG1SQjB1b1U0SzNrSFRzb2ZYRzRkVUhXSjh3R\
1hFWGdKTk9Qek9RIiwiZSI6IkFRQUIifQo
```

When supplied, `mp.jwt.verify.publickey` will override other standard means to supply the Public Key such as `mp.jwt.verify.publickey.location`. Vendor-specific options for supplying the key will always take precedence.

If neither the `mp.jwt.verify.publickey` nor `mp.jwt.verify.publickey.location` are supplied configuration are supplied, the MP-JWT signer configuration will default to a vendor specific behavior as was the case for MP-JWT 1.0.

MicroProfile JWT implementations are required to throw a `DeploymentException` if both `mp.jwt.verify.publickey` and `mp.jwt.verify.publickey.location` are supplied.

mp.jwt.verify.publickey.location

The `mp.jwt.verify.publickey.location` configuration property allows for an external or internal location of Public Verification Key to be specified. The value may be a relative path or a URL.

MicroProfile JWT implementations are required to check the path at startup or deploy time. Reloading the Public Key from the location at runtime as well as the frequency of any such reloading is beyond the scope of this specification and any such feature should be considered vendor-specific.

Relative Path

Relative or non-URL paths supplied as the location are resolved in the following order:

- `new File(location)`
- `Thread.currentThread().getContextClassLoader().getResource(location)`

The following example shows the file `orange.pem` supplied as either a file in the Microservice's binary or locally on disk.

```
java -jar movieservice.jar -Dmp.jwt.verify.publickey.location=orange.pem
```

Any non-URL is treated identically and may be a path inside or outside the archive.

```
java -jar movieservice.jar -Dmp.jwt.verify.publickey.location=/META-INF/orange.pem
```

Parsing of the file contents occurs as defined in Supported Public Key Formats

`file:` URL Scheme

File URL paths supplied as the location allow for explicit externalization of the file via full url.

```
java -jar movieservice.jar
-Dmp.jwt.verify.publickey.location=file:///opt/keys/orange.pem
```

Parsing of the file contents occurs as defined in Supported Public Key Formats

`http:` URL Scheme

HTTP and HTTPS URL paths allow for the Public Key to be fetched from a remote host, which may be the JWT Issuer or some other trusted internet or intranet location.

The location supplied must respond to an HTTP GET. Parsing of the HTTP message body occurs as defined in Supported Public Key Formats

```
java -jar movieservice.jar
-Dmp.jwt.verify.publickey.location=https://location.dev/widget/issuer
```

Other forms of HTTP requests and responses may be supported, but should be considered vendor-specific.

Other URL Schemes

All other locations containing a colon will be considered as URLs and be resolved using the following method:

- `new URL(location).openStream()`

Thus additional vendor-specific or user-defined options can easily be added.

Example custom "smb:" location

```
java -jar movieservice.jar -Dmp.jwt.verify.publickey.location=smb://Host/orange.pem
```

```
-Djava.protocol.handler.pkgs=org.foo
```

Example stub for custom "smb:" URL Handler

```
package org.foo.smb;

import java.io.IOException;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLStreamHandler;

/**
 * The smb: URL protocol handler
 */
public class Handler extends URLStreamHandler {
    @Override
    protected URLConnection openConnection(URL u) throws IOException {
        return // your URLConnection implementation
    }
}
```

See java.net.URL javadoc for more details.

Parsing of the `InputStream` occurs as defined in Supported Public Key Formats and must return Public Key text in one of the supported formats.

mp.jwt.verify.publickey.algorithm

The `mp.jwt.verify.publickey.algorithm` configuration property allows for specifying which Public Key Signature Algorithm is supported by the MP JWT endpoint. This property can be be set to either `RS256` or `ES256`. If `mp.jwt.verify.publickey.algorithm` is not set then both `RS256` and `ES256` must be accepted.

Support for the other asymmetric signature algorithms such as `RS512`, `ES512` and others is optional.

`mp.jwt.verify.publickey.algorithm` will provide an additional hint how to read the Public Key in the PKCS#8 PEM format as both RSA and EC Public Keys in the PKCS#8 PEM format may only have a standard `-----BEGIN PUBLIC KEY-----` header and footer.

It is also recommended to use this property to whitelist the token signature algorithm. For example, MP JWT implementations should only allow an `mp.jwt.verify.publickey.algorithm` algorithm instead of both `RS256` and `ES256` when verifying a token signature.

Encrypted JWT claims and nested tokens

Some claims may contain a sensitive information. For example, a user https://openid.net/specs/openid-connect-core-1_0.html#AddressClaim [Address claim] can be easily viewed if signed JWT token containing such a claim is intercepted or leaked in the logs.

In such cases, when the confidentiality of the claims is critical, the claims can be encrypted or the signed JWT can be encrypted, producing a JWT token by following the steps in the JSON Web Encryption(JWE) specification.

Decrypting the encrypted claims requires a single property, `mp.jwt.decrypt.key.location`, pointing to a private key which can be used to decrypt the token. All the location options supported by the `mp.jwt.verify.publickey.location` property are also supported by `mp.jwt.decrypt.key.location`.

An extra care is required to ensure the private keys are not leaked, particularly, making them available at the insecure HTTP locations or as part of the application archive is not recommended. This is also why the inlined private keys are not supported.

Note that two types of keys are required to implement a JWE encryption scheme:

- Content encryption key - typically a generated secret key which is used to encrypt a plaintext such as a JSON representation of the token claims.
- Key management key - public RSA key which is used to encrypt a content encryption key. `mp.jwt.decrypt.key.location` must point to a private RSA key matching this key.

Key management key algorithms which must be supported are RSA-OAEP (RSAES using Optimal Asymmetric Encryption Padding and SHA-1) and RSA-OAEP-256 (RSAES using Optimal Asymmetric Encryption Padding and SHA-256) with a public RSA key length 2048 bits or higher.

Content encryption algorithm which must be supported is A256GCM (AES in Galois/Counter Mode (GCM)).

Similarly to the signature verification keys, key management keys may be formatted in any of the following formats, specified in order of precedence:

- Private Key Cryptography Standards #8 (PKCS#8) PEM
- JSON Web Key (JWK)
- JSON Web Key Set (JWKS)
- JSON Web Key (JWK) Base64 URL encoded
- JSON Web Key Set (JWKS) Base64 URL encoded

The rules about matching a token `kid` header and JWK `kid` property for selecting the verification keys apply when a key management key is formatted as JWK.

If the claims have been immediately encrypted, without being signed first, then the application endpoints will have no guarantee that a token came from a trusted issuer. To have this proof the claims will need to be signed first and the resulting nested JWT token - encrypted next.

If the encrypted content is an inner nested JWT then the implementations must check that the `cty` (content type) JWE header is set to `JWT` and verify the signature of the nested JWT by configuring the verification key as described in the Signed JWT tokens section.

See the Verification of JWT token claims section how to verify the token claims once the token has been decrypted and the signature of its nested token (if present) verified.

Decryption Configuration Parameters

`mp.jwt.decrypt.key.location`

The `mp.jwt.decrypt.key.location` config property allows for an external or internal location of Private Decryption Key to be specified. The value may be a relative path or a URL. Please see mp.jwt.verify.publickey.location for all the information about the supported locations and Encrypted JWT claims and nested tokens section for the additional recommendations.

`mp.jwt.decrypt.key.algorithm`

The `mp.jwt.decrypt.key.algorithm` configuration property allows for specifying which key management key algorithm is supported by the MP JWT endpoint. Algorithms which must be supported are either `RSA-OAEP` or `RSA-OAEP-256`. If `mp.jwt.decryption.algorithm` is not set then both `RSA-OAEP` and `RSA-OAEP-256` must be accepted.

Support for the other key management key algorithms such as `RSA-OAEP-384`, `RSA-OAEP-512` and others is optional.

Verification of JWT token claims

MP JWT specification currently supports the verification of the token `iss` issuer, `aud` audience, and `iat` issued at claims which is done after the token signature has been verified or the token has been decrypted.

`mp.jwt.verify.issuer`

The `mp.jwt.verify.issuer` config property allows for the expected value of the `iss` claim to be specified. A MicroProfile JWT implementation must verify the `iss` claim of incoming JWTs is present and matches the configured value of `mp.jwt.verify.issuer`.

Note that since this property verifies the `iss` claim value, it will be effective irrespectively of how the token claims have been protected (signed or encrypted or signed first and encrypted next).

`mp.jwt.verify.audiences`

The `mp.jwt.verify.audiences` config property is a comma delimited list of allowable values for the `aud` claim. If specified, a MicroProfile JWT implementation must verify the `aud` claim of incoming JWTs is present and at least one value in the claim matches one of the configured values of `mp.jwt.verify.audiences`.

`mp.jwt.verify.token.age`

The `mp.jwt.verify.token.age` config property allows for the number of seconds since `iat` to be specified. A MicroProfile JWT implementation must verify the `iat` claim of incoming JWTs is present and the configured value of `mp.jwt.verify.token.age` since `iat` has not elapsed. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew.

`mp.jwt.verify.clock.skew`

The `mp.jwt.verify.clock.skew` config property allows for the clock skew in seconds used during the token expiry and age verification to be specified.

Requirements for accepting signed and encrypted tokens

MP JWT specification currently requires that an MP JWT application accepts only signed or only encrypted or only signed and encrypted tokens as it expected that many endpoints will have the requirements to accept a single token type only.

If only `mp.jwt.verify.publickey.location` or `mp.jwt.verify.publickey` properties are set then only the tokens containing the signed claims can be accepted. Support for such tokens is `required`.

If `mp.jwt.decrypt.key.location` and either `mp.jwt.verify.publickey.location` or `mp.jwt.verify.publickey` are set then only the tokens which contain the claims signed first and then encrypted can be accepted. Support for such tokens is `required`.

If only `mp.jwt.decrypt.key.location` property is set then only the tokens containing the encrypted claims can be accepted. Support for such tokens is `optional` however it is `recommended` that the encrypted-only tokens are supported if they are provided to the MP JWT endpoints as cookies.

JWT and HTTP headers

Configuration Properties

`mp.jwt.token.header`

The `mp.jwt.token.header` configuration property allows to set up the header which is expected to contain a JWT token.

MP JWT implementations are required to support `Authorization` (default) or `Cookie` configuration values.

Support for other headers or alternative authentication schemes is optional.

`mp.jwt.token.cookie`

The `mp.jwt.token.cookie` configuration property allows to set up the Cookie name (default is `Bearer`) which is expected to contain a JWT token.

This configuration will be ignored unless `mp.jwt.token.header` is set to `Cookie`.

Providing the recommendations on how to secure a JWT token as a Cookie is out of scope for the MP JWT specification. Generally one should avoid putting sensitive user information into a signed JWT token.

How to provide Configuration Parameters

MicroProfile JWT leverages the MicroProfile Config specification to provide a consistent means of passing all supported configuration options.

Prior to MicroProfile JWT 1.1 all configuration options for the Public Key and claim verification were vendor-specific. Any equivalent vendor-specific methods of configuration are still valid and shall be considered to override any standard configuration mechanisms.

MP JWT specification allows at minimum configuration options to be specified in the microservice binary itself or via command-line via -D properties as follows:

```
java -jar movieservice.jar -Dmp.jwt.verify.publickey.location=orange.pem
```

By convention of the MicroProfile JWT specification, property names are always lowercase and begin with `mp.jwt.`

Mapping Configuration Parameters to Environment Variables

When using environment variables to specify the MP-JWT configuration properties defined in this section, note that some operating systems allow only alphabetic characters and underscores in environment variables. Since characters such as '.' may be disallowed, in order to set a value for a config property such as `mp.jwt.verify.publickey` using an environment variable, the following mapping rules from the MP configuration spec are relevant:

When searching environment variables for configuration properties, the following transformation is applied to attempt to find a match:

- Exact match (i.e. mp.jwt.verify.publickey)
- Replace nonalphanumeric characters with '_' (i.e. mp_jwt_verify_publickey)
- Replace nonalphanumeric characters with '_' and convert to uppercase (i.e. MP_JWT_VERIFY_PUBLICKEY)

With these rules, the matching portable environment variables names for the current MP-JWT verification properties are:

`mp.jwt.verify.publickey`

: mp_jwt_verify_publickey or MP_JWT_VERIFY_PUBLICKEY

`mp.jwt.verify.publickey.location`

: mp_jwt_verify_publickey_location or MP_JWT_VERIFY_PUBLICKEY_LOCATION

`mp.jwt.verify.publickey.algorithm`

: mp_jwt_verify_publickey_algorithm or MP_JWT_VERIFY_PUBLICKEY_ALGORITHM

`mp.jwt.verify.issuer`

mp_jwt_verify_issuer or MP_JWT_VERIFY_ISSUER

`mp.jwt.verify.audiences`

mp_jwt_verify_audiences or MP_JWT_VERIFY_AUDIENCES

`mp.jwt.token.header`

mp_jwt_token_header or MP_JWT_TOKEN_HEADER

`mp.jwt.token.cookie`

mp_jwet_token_cookie or MP_JWT_TOKEN_COOKIE

`mp.jwt.decrypt.key.location`

mp_jwt_decrypt_key_location or MP_JWT_DECRYPT_KEY_LOCATION

Future Directions

Not all considerations discussed during the specification process make it into the specification. This section serves as an abridged version for the purposes of soliciting feedback and interest. By convention we will leave items in the Future Direction for at most two revisions.

`resource_access` claim

In future versions of the API we would like to address service specific group claims. The `resource_access` claim originally targeted for the 1.0 release of the specification has been postponed as additional work to determine the format of the claim key as well as how these claims would be surface through the standard Java EE APIs needs more work than the 1.0 release timeline will allow.

For reference, a somewhat related extension to the OAuth 2.0 spec Resource Indicators for OAuth 2.0 has not gained much traction. The key point it makes that seems relevant to our investigation is that the service specific grants most likely need to be specified using URIs for the service endpoints. How these endpoints map to deployment virtual hosts and partial, wildcard, etc. URIs needs to be determined.

`roles` claim

A `roles` claim was considered in addition to the `groups` claim that made it into the final specification. The `groups` claim is intended to be mapped to specific roles on the target resource server. The `roles` claim was intended to explicitly state roles inside the JWT that would not be subject to any mappings and are made available to directly to the application for `@RolesAllowed` and similar RBAC checks. The roles in the JWT should be exposed to the app in addition to any roles that result from group-to-role mapping provided by the target resource server. The intended JWT datatype for `roles` should be a JSON string array.

Though a `roles` claim is not required, implementations that support it and applications that use it should do so as detailed in this section to ensure alignment for any future standardization.

`aud` claim

The `aud` claim defined in RFC 7519 section 4.1.3 was considered for addition. The intended JWT datatype for `aud` should be a JSON string array or a single string as defined in RFC 7519.

Though an `aud` claim is not required, implementations that support it and applications that use it should do so as detailed in this section to ensure alignment for any future standardization.

Enabling/Disabling Claim Requirements

We discussed adding configuration settings to disable/enable the requirements for claims such as the `iss` claim, but could not decide on the format for such configuration given the time constraints of the 1.1 release. We will revisit this issue in the next release.

`classpath:` URL Scheme

The option to have a built-in `classpath:` URL Scheme was discussed with the intended benefit of providing some way to explicitly state a Public Key file is inside the archive and to remove potential a similarly named file existed on disk.

For the moment this was deemed an edge-case that could be solved with a custom URL Scheme and consensus that this would add to the complexity of the specification. In practice, this may be very useful so those who find themselves with this scenario are encouraged to contact the MicroProfile discussion lists.

Expiration tolerance

Relaxing or potentially ignoring the expiration time of a JWT was discussed and deemed an attractive option for future standardization. It was omitted in efforts to keep the first revision of the configuration as simple as possible.

Users who find themselves with this need are encouraged to both request support from their respective implementation and to detail their use case the MicroProfile discussion lists, so any future standardization work accounts for all scenarios.

Passing JWTs as Cookies

Semantics for passing JWTs via HTTP `Cookie` headers instead of the HTTP `Authorization` header were discussed and deemed a valuable addition for interoperability purposes.

It is likely to be a future recommendation that MicroProfile JWT implementation supporting the transport of MP-JWT tokens using cookies SHOULD recognize the cookie name `Bearer`, and the cookie value format being the JWS Compact Serialization as described in section 7.1 of JSON Web Signature (RFC-7515).

There are a number of security considerations when using cookies to transfer security information such as tokens in cookies. Issues related to CSRF, XSS, and storage need to be considered. The following links present material on these issues:

- Cookie Storage
- XSS
- CSRF

The motivation for using `Bearer` as the Cookie name is to establish a pattern of using the `Authorization` scheme as the cookie name. Bearer token example would look as follows:

- `Authorization: Bearer as14efgscd31qrewtadg`
- `Cookie: Bearer=as14efgscd31qrewtadg`

Basic auth would look as follows:

- `Authorization: Basic gasdqwe198abg313ffd`

- `Cookie: Basic=gasdqwe198abg313ffd`

Sample Implementations

This sections references known sample implementations of the Eclise MicroProfile JWT RBAC authorization specification.

General Java EE/SE based Implementations

A baseline sample implementation that is available under the Apache License, Version 2.0 can be found at https://github.com/MicroProfileJWT/microprofile-jwt-auth-prototype. The purpose of this prototype is to offer reusable code for integration of the JWT RBAC authenticaiton and authorization spec in various container environments. This particular implementation contains:

- a default implementation of the JsonWebToken interface
- a JAX-RS ContainerRequestFilter prototype
- JSR-375 IdentityStore and Credential prototypes

Wildfly Swarm Implementations

A sample implementation for a custom auth-method of MP-JWT with the Wildfly/Wildfly-Swarm Undertow web container, as a Wildfly-Swarm fraction, available under the Apache License, Version 2.0 can be found at: https://github.com/MicroProfileJWT/wfswarm-jwt-auth-fraction.

Release Notes for MicroProfile JWT 2.2

A full list of changes delivered in the 2.2 release can be found at MicroProfile JWT 2.2 Milestone.

Spec Changes

- Support both RS256 and ES256 signature algorithms if no signature algorithm has been configured (340)

Release Notes for MicroProfile JWT 2.1

A full list of changes delivered in the 2.1 release can be found at MicroProfile JWT 2.1 Milestone.

Spec Changes

- `mp.jwt.decrypt.key.algorithm` property for supporting an `RSA-OAEP-256` key management algorithm has been introduced (292).
- `mp.jwt.verify.token.age` property for restricting a token age has been introduced (284).
- `mp.jwt.verify.clock.skew` property for configuring a leeway for the token expiry and age verification has been introduced (284).
- Remaining Java EE references have been replaced with Jakarta EE references (281).

Release Notes for MicroProfile JWT 2.0

Incompatible Changes

This release aligns with Jakarta EE 9.1 (246), so it won't work with earlier versions of Jakarta or Java EE.

API/SPI Changes

There are no functional changes introduced in this release, except the dependency updating from javax to jakarta.

Other Changes

- TCK Jose4j dependency has been bumped to 0.7.9 to support running TCK with Java 17 (251)
- TCK ApplicationScopedInjectionTest has been fixed (250)

Release Notes for MicroProfile JWT 1.2

A full list of changes delivered in the 1.2 release can be found at MicroProfile JWT 1.2 Milestone.

API Changes

- A convenience method has been added to allow retrieving claims from `JsonWebToken` by using the `Claims` enum (#154)

Spec Changes

- Support for verifying JWT tokens which have been signed using Elliptic Curve `ES256` signature algorithm (#161)
- Support for decrypting JWT tokens which have been encrypted using `RSA-OAEP` and `A256GCM` algorithms and contain the claims or inner-signed JWT tokens (#58)
- Support for JWT audience `aud` claim (#121)
- Support for JWT token cookies (#93)
- JWT token `groups` claim is now optional (#129)
- Better specification of the injection point (#116, #127), scope (#45, #183) and required claims (#128) requirements
- Support for RSA keys of 1024 bit length has been deprecated (#197)

Other Changes

- Update to Jakarta APIs for MP 4.0 (#143)
- New TCK tests
- TCK tests now use `Jose4J` to sign and encrypt the tokens.

Release Notes for MicroProfile JWT 1.1.1

Changes in 1.1.1-RC2

This candidate patch release provides one additional TCK test fix, and information on how to use the MP-JWT configuration properties as environment variables on platforms that don't allow dots '.' in their names.

There are no API changes.

Closed Issues in 1.1.1-RC2

https://github.com/eclipse/microprofile-jwt-auth/issues/104 https://github.com/eclipse/microprofile-jwt-auth/issues/107

Changes in 1.1.1-RC1

This is a candidate patch release that provides fixes to the TCK tests that were seen to have issues in different implementations. The TCK tests WARs now include a META-INF/MPJWTTESTVERSION resource that contains the major/minor version string enum for the MP-JWT version the test WAR is targeting. Currently the versions enums are:

```
public enum MpJwtTestVersion {
    MPJWT_V_1_0,
    MPJWT_V_1_1
    ;
}
```

There are no API changes.

Closed Issues in 1.1.1-RC1

- https://github.com/eclipse/microprofile-jwt-auth/issues/104
- https://github.com/eclipse/microprofile-jwt-auth/issues/103
- https://github.com/eclipse/microprofile-jwt-auth/issues/98

Release Notes for MicroProfile JWT 1.1

The focus of this release was to add support for configuring the public key and issuer needed for verification of the MP-JWT using MicroProfile Config. The new MicroProfile Config properties are:

mp.jwt.verify.publickey

The embedded key material of the public key for the MP-JWT signer in PKCS8 PEM or JWK(S) format. If not found the mp.jwt.verify.publickey.location needs to be checked.

mp.jwt.verify.publickey.location

The relative path or full URL of the public key. All relative paths will be resolved within the archive using ClassLoader.getResource. If the value is a URL it will be resolved using new URL("…").openStream()

mp.jwt.verify.issuer

The expected iss claim value to validate against an MP-JWT.