Interoperable JWT RBAC for Microprofile

Scott Stark; Pedro Igor Silva

1.0-RC10, 2017-09-05

Table of Contents

Specification: Interoperable JWT RBAC for Microprofile

Version: 1.0-RC10

Status: Proposed Final Draft

Release: 2017-09-05

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 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 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 Java 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": The token subject's group memberships that will be mapped to Java EE style application level roles in the MicroProfile service container.

Minimum MP-JWT Required Claims

The required minimum set of MP-JWT claims is then:

typ

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

alg

This JOSE header parameter identifies the cryptographic algorithm used to secure the JWT. MP-JWT requires the use of the RSASSA-PKCS1-v1_5 SHA-256 algorithm and must be specified as "RS256", RFC7515, Section 4.1.1

kid

This JOSE header parameter is a hint indicating which key was used to secure the JWT. RFC7515, Section-4.1.4

iss

The MP-JWT issuer. RFC7519, Section 4.1.1

sub

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

aud

Identifies the recipients that the JWT is intended for. RFC7519, Section 4.1.3

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

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

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

upn

This MP-JWT custom claim is the user principal name in the java.security.Principal interface, and is the caller principal name in javax.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.

groups

This MP-JWT custom claim is the list of group names that have been assigned to the principal of the MP-JWT. This typically will required a mapping at the application container level to application deployment roles, but a a one-to-one between group names and application role names is required to be performed in addition to any other mapping.

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",
       "aud": "s6BhdRkqt3",
       "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, not the
     *
     * @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));
    }
}
```

Additional Claims

The JWT can contain any number of other custom and standard claims, and these are made available from the JsonWebToken getOtherClaim(String) method. An example MP-JWT that contains additional "auth_time", "preferred_username", "acr", "nbf" 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),
    aud("Audience", Set.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
    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;
    }
}
```

Custom claims not handled by the Claims enum are required to be valid JSON-P `javax.json.JsonValue` subtypes. The current complete set of valid claim types is therefore, (excluding the invalid Claims.UNKNOWN Void type): * java.lang.String * java.lang.Long * java.lang.Boolean * java.util.Set<java.lang.String> * javax.json.JsonValue.TRUE/FALSE * javax.json.JsonString * javax.json.JsonNumber * javax.json.JsonArray * javax.json.JsonObject

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.jwt.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.annotation.LoginConfig;

import javax.ws.rs.ApplicationPath;
import javax.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 bearer token as an invalid MP-JWT token if any of the following conditions are not met:

1. The JWT must have a JOSE header that indicates the token was signed using the RS256 algorithm.
2. 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_UNUATHENTICATED(401) error.
3. The JWT signer must have a public key that that maps to an MP-JWT implementation container runtime configured value. Any public key other than those that have been whitelisted by the container configuration must be rejected with an HTTP_UNUATHENTICATED(401) error.

NOTE	A future MP-JWT specification may define how an MP-JWT implementation makes use of the MicroProfile Config specification to allow for configuration of the issuer, issuer public key and expiration clock skew. Additional requirements for validation of the required MP-JWT claims may also be defined.

Mapping MP-JWT Tokens to Java EE Container APIs

The requirements of how a JWT should be exposed via the various Java 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:

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

    @Inject
    private JsonWebToken callerPrincipal;
```

Injection of `JsonWebToken` claims via Raw Type, `ClaimValue`, `javax.inject.Provider` 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 implementation 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 set of types one my use for a claim value injection is:

- java.lang.String
- java.util.Set<java.lang.String>
- java.lang.Long
- java.lang.Boolean
- javax.json.JsonValue subtypes

- 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 `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 ④
    @Claim("aud")
    private ClaimValue<Set<String>> aud;
    @Inject
    @Claim("groups")
    private ClaimValue<Set<String>> groups;
    @Inject ⑤
    @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 ⑥
    @Claim("custom-missing")
    private ClaimValue<Optional<Long>> custom;
    //
    @Inject
    @Claim(standard = Claims.jti)
    private Provider<String> providerJTI;
    @Inject ⑦
    @Claim(standard = Claims.iat)
    private Provider<Long> providerIAT;
    @Inject
    @Claim("groups")
    private Provider<Set<String>> providerGroups;
    //
    @Inject
    @Claim(standard = Claims.jti)
    private JsonString jsonJTI;
```

```
    @Inject
    @Claim(standard = Claims.iat)
    private JsonNumber jsonIAT;
    @Inject ⑧
    @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>` wapper.

④ Injection of the aud audience claim as a Set<String>. This is the required type as seen by looking at the Claims.aud enum value's Java type member.

⑤ Injection of the issued at time claim using an @Claim that references the claim name using the Claims.iat enum value.

⑥ Injection of a custom claim that does exist will result in an Optional<Long> value for which isPresent() will return false.

⑦ Another injection of a non-proxyable raw type like java.lang.Long, but the use of the javax.inject.Provider interface allows for injection to occur in non-RequestScoped contexts.

⑧ 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 validate the use of claim value injection into contexts that do not match @RequestScoped. When the target context has @ApplicationScoped or @SessionScoped scope, implementations are required to generate a `javax.enterprise.inject.spi.DeploymentException`. For any other context, implementations should issue a warning, that may be suppressed by an implementation specific configuration setting.

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 use the javax.inject.Provider or javax.inject.Instance interfaces to do so.

JAX-RS Container API Integration

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

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

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

`javax.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.

Mapping from `@RolesAllowed`

Any role names used in @RolesAllowed or equivalent security constraint metadata that match names in 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.

Recommendations for Optional Container Integration

This section describes the expected behaviors for Java EE container APIs other than JAX-RS.

javax.security.enterprise.identitystore.IdentityStore.getCallerGroups(CredentialValidationResult)

This method should return the set of names found in the "groups" claim in the JWT if it exists, an empty set otherwise.

javax.ejb.SessionContext.getCallerPrincipal()

The java.security.Principal returned from this method MUST be an instance of org.eclipse.microprofile.jwt.JsonWebToken.

javax.ejb.SessionContext#isCallerInRole(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.

Overriding @LoginConfig from web.xml login-config

If a deployment with a web.xml descriptor contains a login-config element, an MP-JWT implementation should view the web.xml metadata as an override to the deployment annotation.

javax.servlet.http.HttpServletRequest.getUserPrincipal()

The java.security.Principal returned from this method MUST be an instance of org.eclipse.microprofile.jwt.JsonWebToken.

javax.servlet.http.HttpServletRequest#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.

javax.security.jacc.PolicyContext.getContext("javax.security.auth.Subject.container")

The javax.security.auth.Subject returned by the PolicyContext.getContext(String key) method with the standard "javax.security.auth.Subject.container" key, MUST return a Subject that has a `java.security.Principal` of type `org.eclipse.microprofile.jwt.JsonWebToken` amongst it's set of `Principal`s returned by `getPrincipals()`. Similarly, `Subject#getPrincipals(JsonWebToken.class)` must return a set with at least one value. This means that following code snipet must not throw an AssertionError:

```
    Subject subject = (Subject) PolicyContext.getContext(
"javax.security.auth.Subject.container");
    Set<? extends Principal> principalSet = subject.getPrincipals(JsonWebToken.class);
    assert principalSet.size() > 0;
```

Mapping MP-JWT Token to Other Container APIs

For non-Java 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.

Future Directions

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 addition 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.

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.