Article
Elliott Grey · Mar 7, 2023 10m read

Reference for the JSON Web Classes

Foreword

InterSystems IRIS versions 2022.2 and newer feature a redesigned functionality for JSON web tokens (JWTs). Once housed under the %OAuth2 class package, the JWT class, along with other JSON web classes (JWCs), now live under %Net.JSON. This migration occured in order to modularize the JWCs. Before, they were closely intertwined with the implementation for the OAuth 2.0 framework. Now, they can be maintained and used separately from OAuth2.

Note: For backwards compatibility, the classes still exist under %OAuth2 package, but the codebase now uses %Net.JSON.

The goal of this article is to serve as a sort of cheat sheet for the JWCs because in my quest for writing documentation for them, I didn't find any one source with comprehensive information that covered all of them and how they related to each other. I aim for this article to be that source.

Prologue

What are JWCs?

JSON web classes are web protocols using JSON-based data structures. They are useful for authorization and information exchange, such as in OAuth 2.0 and OpenID Connect.

InterSystems IRIS 2022.2+ currently supports seven classes under %Net.JSON: (click to expand definition)

> JSON Object Signing and Encryption (JOSE)
A set of standards for signing and encrypting data using JSON-based data structures. Includes JWT, JWS, JWE, JWA, JWK, and JWKS.

> JSON Web Token (JWT)
A compact, URL-safe means of representing claims transferred between two parties that can be digitally signed, encrypted, or both.

> JSON Web Signature (JWS)
A JWS represents signed content using JSON-based data structures. The JWA defines the signing and verification algorithms for the JWS. AKA a signed JWT.

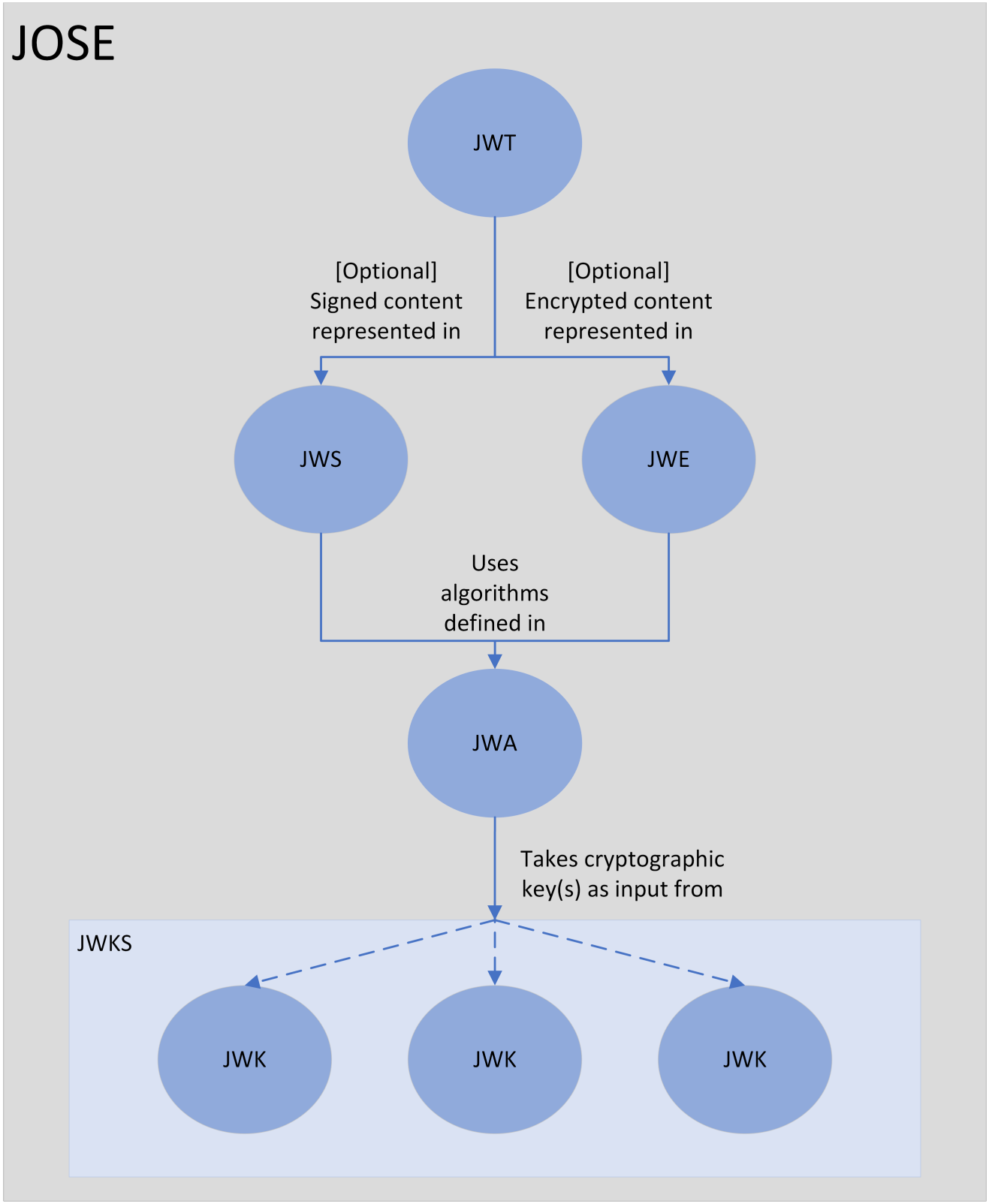
> JSON Web Encryption (JWE)
A JWE represents encrypted content using JSON-based data structures. The JWA defines the encryption and decryption algorithms for the JWE. AKA an encrypted JWT.

> JSON Web Algorithms (JWA)
A JWA defines a set of cryptographic algorithms and identifiers used with the JWS, JWE, and JWK classes.

> JSON Web Key (JWK)
A JWK represents a cryptographic key used as input for the algorithms defined in the JWA class.

> JSON Web Key Set (JWKS)
A JWKS is a set of JWKs

The following diagram demonstrates the relationship between the JWCs as defined above:



We'll break down the different pieces further in the next section.

A JWT Dissection

A JSON web token (JWT) is a compact, URL-safe means of representing claims transferred between two parties that can be digitally signed, encrypted, or both.

There are two types of JWTs:
- JWS
- JWE

A JWS is a signed JWT and a JWE is an encrypted JWT. Common parlance appears to be to just say "JWT" or "encrypted JWT." Notice that the default is for JWTs to be signed, although they *can* be unsigned (these are unsecured JWTs).

But let's back that up a second—what is a claim? A claim is just a piece of information represented in a key/value pair that a client is asserting as true. For example, it could be information about a client trying to log in from a certain location. The following JSON object contains three claims (username, location, and admin):

```
{
    "username": "persephone",
    "location": "underworld",
    "admin": "true"
}
```

So JWTs transfer claims like that between two parties, such as a client and a server. However, if they just transferred *just* that information, there would be no guarantee that someone didn't tamper with it or that no one but the intended recipient saw the contents. Our message would have no integrity or confidentiality. Fortunately, JWTs provide an optional way of guaranteeing both with the JSON web signature (JWS) and JSON web encryption (JWE) standards. Whether it's a JWS or a JWE, there are multiple parts to a JWT.

JSON Web Signature (JWS)

For a non-encrypted, signed JWT (henceforth called a JWS), there are three parts:

1. Header
2. Payload
3. Signature

Each part is a JSON object. If you Base64URL encode each part and concatenate them together with a period (.) between them, you have a JWT (header.payload.signature).

Let's dive into each part.

The Header

The header part of a JWS consists of metadata about the token type and, if specified, the JSON web algorithm (JWA) needed for signature validation of the token. It could look like the following:

```
{
  "alg": "HS256",
```

```
  "typ": "JWT"
}
```

Base64URL encode this (below) and you have the first part of a JWT!

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

So, quick recap: the header contains the JWA and token type.

The Payload

The second part of a JWS is the payload. It holds the claims. Using the previous example, one such payload could be:

```
{
    "username": "persephone",
    "location": "underworld",
    "admin": "true"
}
```

Then we Base64URL encode it to get:

```
eyJ1c2VybmFtZSI6InBlcnNlcGhvbmUiLCJsb2NhdGlvbiI6InVuZGVyd29ybGQiLCJhZG1pbiI6InRydWUif
Q
```

So now our JWS looks like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InBlcnNlcGhvbmUiLCJsb2NhdGlvbiI6
InVuZGVyd29ybGQiLCJhZG1pbiI6InRydWUifQ
```

So, quick recap: the payload contains the claims/information you want to transmit.

The Signature

The third part of a JWS is the signature. You take the JWT so far (the encoded header and the encoded payload), a secret (also known as a private key or a JSON web key (JWK)), the algorithm specified in the JWA, and sign that. Doing so might look something like this:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Using the secret/JWK value thecatsmeow, our final JWS would look like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InBlcnNlcGhvbmUiLCJsb2NhdGlvbiI6
```

```
InVuZGVyd29ybGQiLCJhZG1pbiI6InRydWUifQ.KAZcjC9tqRV4DunI3sSma3k6fvL5ntgLXe9Gl7hKg5o
```

So, quick recap: the signature contains the integrity validation.

Abstracting up a little bit, we have:

You Noticed I Said...

You may have noticed that I qualified our JWT as "*can* be digitally signed, encrypted, or both."

Let's talk about the signing part first then dive into the encryption in the next section. It is possible to not sign the JWS and just have a header and payload (so no longer a JWS but an unsigned, unencrypted JWT). This is possible if the JWA specified in the header is "none".

So it would look like:

```
{
  "alg": "none",
  "typ": "JWT"
}
```

Then the resulting JWT would just be the Base64URL encoded header + . + Base64URL encoded payload + .. The signature would be an empty string, so an example unsecured JWT might look like:

```
eyJhbGciOiJub25lIn0.eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtc
GxlLmNvbS9pc19yb290Ijp0cnVlfQ.
```

But please don't send unsecured JWTs because then anyone could tamper with them and you don't know if the claims are valid.

JSON Web Encryption (JWE)

To reiterate, a JWT can be digitally signed, encrypted, or both. We already discussed signatures with JWSs, so let's move on to encryption.

As noted before, a non-encrypted, signed JWT (JWS) has three parts: the header, payload, and signature.

As noted now, an encrypted JWT (henceforth called a JWE) has five parts:

1. Protected Header
2. Encrypted Key
3. Initialization Vector (IV)
4. Payload/Ciphertext
5. Authentication Tag

Protected Header

The protected header is the first part of a JWE. It is unencrypted because the recipient needs to know how to decrypt the rest of the JWE. To inform them of how, it contains information such as 1) the algorithm used to encrypt the content encryption key (CEK) and produce the encrypted key as well as 2) the algorithm used to encrypt the payload and produce the ciphertext and authentication tag.

The following is an example of a protected header:

```
{
  "alg":"RSA-OAEP",
  "enc":"A256GCM"
}
```

It is then Base64URL encoded to produce:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ
```

Encrypted Key

The second part of our JWE is the encrypted key. The encrypted key is the encrypted form of the CEK, which is a symmetric key used to encrypt the payload and produce the ciphertext and authentication tag. The CEK is encrypted using the algorithm specified in the "alg" value in the protected header and the recipient's public key.

With a CEK of thecatsmeowmeows and a randomly generated RSA 1024-bit public key, one such value of the *encrypted key* using the RSA-OAEP algorithm could be:

```
X6znPIKWHnO8MhHD2scnUv7PVAo8VfxHYxmZQR0J8/rqGOB+udq8DkXd93n7S2cS3LT1Inx4qQ5J8GquQyc2x
fS5n2INgKjSedYac4LBCkmpYRbRyNawK2eMEUDkcdBlqBE4NlWcAhl6X0H4AiNs7r+P8ffipvyztd51JdLoTl
w=
```

The encrypted key is then Base64URL encoded and concatenated to the protected header. So we have our JWE so far as:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.WDZ6blBJS1dIbk84TWhIRDJzY25VdjdQVkFvOF
ZmeEhZeG1aUVIwSjgvcnFHT0IrdWRxOERrWGQ5M243UzJjUzNMVDFJbng0cVE1SjhHcXVReWMyeGZTNW4ySU5
nS2pTZWRZYWM0TEJDa21wWVJiUnlOYXdLMmVNRVVEa2NkQmxxQkU0TmxXY0FobDZYMEg0QWlOczdyK1A4ZmZp
cHZ5enRkNTFKZExvVGx3PQ
```

Initialization Vector (IV)

The IV is the third part of the JWE. It is randomly generated and Base64URL encoded. It does not need to be secret, so it is not encrypted. An example IV could be catsarefantastic. We would then Base64URL encode this and concatenate with the encoded protected header and encrypted key parts to get:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.WDZ6blBJS1dIbk84TWhIRDJzY25VdjdQVkFvOF
ZmeEhZeG1aUVIwSjgvcnFHT0IrdWRxOERrWGQ5M243UzJjUzNMVDFJbng0cVE1SjhHcXVReWMyeGZTNW4ySU5
nS2pTZWRZYWM0TEJDa21wWVJiUnlOYXdLMmVNRVVEa2NkQmxxQkU0TmxXY0FobDZYMEg0QWlOczdyK1A4ZmZp
cHZ5enRkNTFKZExvVGx3PQ.Y2F0c2FyZWZhbnRhc3RpYw
```

Payload/Ciphertext

The fourth part of the JWE is the payload/ciphertext. This is where we nest the JWT. Everything so far has been about how do we protect and then later read this data once received, but now we get to talk about the data. Everything we talked about in the JWS section applies here. We have our three piece JWS with the header,

payload, and signature. Then using the CEK and IV, we encrypt the JWS using AES GCM and request a 128-bit authentication tag output.

An example of a possible Base64URL encoded ciphertext is:

```
NjI0YjZkZTlmMGEzNjk0MTgyMTYyNTc3MmEyMjM4ZWY0MTJhMzljMzJiOGVjNzVjMzU4MGIzNTVhMGUyN2M1M
WYyY2Y2OGIyYmNkODM2YmNiZjBkOGIzZjMzMmQyODBlZWZhNjBkYTQ5M2VlNjRhMTg4NmMzYTFlY2E2OGQ0Nj
kyOGQzNTFjOWFjODdhY2QzZDc0ZTY4OTc1MTA4NzQ0NTEyNTJhOGM5N2U3OGFkNjJhMmNmMWQwNzM5MmQwYzc
yM2EzMjg5MWI2YjlmMzRkNmYxMDU5YTVlMTljZThjMTNkNzFlMjgzZWY1ZGM0ZDdmZTNhMzk1YmM2MDE5NjRm
ZmMwYmZlMDM2YWY1MzZmYTdiYTYzNWU3NTJmMzk1OTBhY2Y2ZWM4YjlmZjBmYzY1ZTM0M2U5YzE4OTk0ZjAyY
TZlNDA0NjEzNDM1ZTVhMQ
```

So our JWE so far is:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.WDZ6blBJS1dIbk84TWhIRDJzY25VdjdQVkFvOF
ZmeEhZeG1aUVIwSjgvcnFHT0IrdWRxOERrWGQ5M243UzJjUzNMVDFJbng0cVE1SjhHcXVReWMyeGZTNW4ySU5
nS2pTZWRZYWM0TEJDa21wWVJiUnlOYXdLMmVNRVVEa2NkQmxxQkU0TmxXY0FobDZYMEg0QWlOczdyK1A4ZmZp
cHZ5enRkNTFKZExvVGx3PQ.Y2F0c2FyZWZhbnRhc3RpYw.NjI0YjZkZTlmMGEzNjk0MTgyMTYyNTc3MmEyMjM
4ZWY0MTJhMzljMzJiOGVjNzVjMzU4MGIzNTVhMGUyN2M1MWYyY2Y2OGIyYmNkODM2YmNiZjBkOGIzZjMzMmQy
ODBlZWZhNjBkYTQ5M2VlNjRhMTg4NmMzYTFlY2E2OGQ0NjkyOGQzNTFjOWFjODdhY2QzZDc0ZTY4OTc1MTA4N
zQ0NTEyNTJhOGM5N2U3OGFkNjJhMmNmMWQwNzM5MmQwYzcyM2EzMjg5MWI2YjlmMzRkNmYxMDU5YTVlMTljZT
hjMTNkNzFlMjgzZWY1ZGM0ZDdmZTNhMzk1YmM2MDE5NjRmZmMwYmZlMDM2YWY1MzZmYTdiYTYzNWU3NTJmMzk
1OTBhY2Y2ZWM4YjlmZjBmYzY1ZTM0M2U5YzE4OTk0ZjAyYTZlNDA0NjEzNDM1ZTVhMQ
```

Authentication Tag

The authentication tag is the final part of the JWE. It is an output of obtaining the ciphertext (encrypted nested JWS). The authentication tag received from encrpyting the ciphertext in the last section is:

```
9f19e30efeddf20f5232b76f07c755ac
```

So we Base64URL encode it and concatenate it on the JWE to get our finalizad JWE as:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.WDZ6blBJS1dIbk84TWhIRDJzY25VdjdQVkFvOF
ZmeEhZeG1aUVIwSjgvcnFHT0IrdWRxOERrWGQ5M243UzJjUzNMVDFJbng0cVE1SjhHcXVReWMyeGZTNW4ySU5
nS2pTZWRZYWM0TEJDa21wWVJiUnlOYXdLMmVNRVVEa2NkQmxxQkU0TmxXY0FobDZYMEg0QWlOczdyK1A4ZmZp
cHZ5enRkNTFKZExvVGx3PQ.Y2F0c2FyZWZhbnRhc3RpYw.NjI0YjZkZTlmMGEzNjk0MTgyMTYyNTc3MmEyMjM
4ZWY0MTJhMzljMzJiOGVjNzVjMzU4MGIzNTVhMGUyN2M1MWYyY2Y2OGIyYmNkODM2YmNiZjBkOGIzZjMzMmQy
ODBlZWZhNjBkYTQ5M2VlNjRhMTg4NmMzYTFlY2E2OGQ0NjkyOGQzNTFjOWFjODdhY2QzZDc0ZTY4OTc1MTA4N
zQ0NTEyNTJhOGM5N2U3OGFkNjJhMmNmMWQwNzM5MmQwYzcyM2EzMjg5MWI2YjlmMzRkNmYxMDU5YTVlMTljZT
hjMTNkNzFlMjgzZWY1ZGM0ZDdmZTNhMzk1YmM2MDE5NjRmZmMwYmZlMDM2YWY1MzZmYTdiYTYzNWU3NTJmMzk
1OTBhY2Y2ZWM4YjlmZjBmYzY1ZTM0M2U5YzE4OTk0ZjAyYTZlNDA0NjEzNDM1ZTVhMQ.OWYxOWUzMGVmZWRkZ
jIwZjUyMzJiNzZmMDdjNzU1YWM
```

Let's abstract away the Base64URL encoding and look at the high level overview of a JWE:

Notes on Terms

In striving for clarity, I dropped using some of the JSON web class names in favor of their descriptive names. Namely, JWK/JWKS in favor of public keys, cryptographic keys, encryption keys, etc. The key used for encryption

or signing is a JSON web key (JWK) and the set of them (so the symmetric key and the pairs of asymmetric keys) are a JSON web key set (JWKS).

I wanted to pause to mention this to bring all the terminology full circle. A JWT is either a JWS or a JWE. The algorithms used in a JWS/JWE are defined in the JWA. The keys used as input for the algorithms in the JWA are JWKs and stored as a set as a JWKS.

What about JOSE? JOSE is the collection of standards. Like we have a murder for crows or a clowder for cats, we have a JOSE for JSON web standards.

Conclusion

I hope this article serves as a good reference point for those looking to work with the JSON web classes (JWCs).

The JWCs have three different use cases:

1. Authentication
2. Authorization
3. Information exchange

In InterSystems IRIS 2022.2+, you can set the OAuth 2.0 configurations to use JWTs. This is described in OAuth 2.0 and OpenID Connect. Your custom code can utilize the JWCs as defined in %Net.JSON for these use cases as well.

If you found this article helpful, please let me know!

#JSON #OAuth2 #Security #InterSystems IRIS