大家好，今天要來跟大家說明一個知道誰在敲打你家窗戶的方法！？我們在網路的世界中可以透過 API (Application Programming Interface) 與各種應用程式互相交流、溝通、傳遞各種訊息，而如何驗證身份就變成一件重要的事情，即我們不應該將所有的資料都裸奔，讓任何人都可以存取，所以在設計 Web API 的時候常常會需要考慮到授權或驗證的部分，那我們要如何確認誰有權限呢？要如何安全地傳遞訊息呢？接下來跟大家說明其中一種方式。

JSON Web Token (JWT)

JWT 的全名是 `JSON Web Token`，是一種基於 JSON 的開放標準(RFC 7519)，它定義了一種簡潔(compact)且自包含(self-contained)的方式，用於在雙方之間安全地將訊息作為 JSON 物件傳輸。而這個訊息是經過數位簽章(Digital Signature)，因此可以被驗證及信任。可以使用 `密碼` (經過 **HMAC** 演算法) 或用一對 `公鑰/私鑰` (經過 **RSA** 或 **ECDSA** 演算法) 來對 JWT 進行簽章。

> 註：
>
> - RFC：記錄網際網路規範、協定、過程等的標準檔案。
> - 簡潔(compact)：體積非常的小，可放在 URL 、 POST 參數或 HTTP Header 內發送請求，體積小意味著傳輸速度快。
> - 自包含(self-contained)：payload 裡面就有所需要的資訊，不需要再重新 query database 的資料。
> - 數位簽章(Digital Signature)：
> - 演算法：HMAC、RSA、ECDSA

什麼情況適合使用 JWT

- **授權(Authorization)**：這是很常見 JWT 的使用方式，例如使用者從 Client 端登入後，該使用者再次對 Server 端發送請求的時候，會夾帶著 JWT，允許使用者存取該 token 有權限的資源。單一登錄(Single Sign On)是當今廣泛使用 JWT 的功能之一，因為它的成本較小並且可以在不同的網域(domain)中輕鬆使用。

- **訊息交換(Information Exchange)**：JWT 可以透過公鑰/私鑰來做簽章，讓我們可以知道是誰發送這個 JWT，此外，由於簽章是使用 header 和 payload 計算的，因此還可以驗證內容是否遭到篡改。

JWT 的組成

1. **header**
2. **payload**
3. **signature/encryption data**

其中前兩個元素 header 和 payload 是特定結構的 JSON 物件，第三個部分取決於演算法是用來作簽章還是加密，如果是未加密的 JWT，則將其省略。JWT 可以被編碼成 JWS/JWE 簡潔的表現形式(Compact Serialization)。JWS 和 JWE 規範中定義了另一種序列化格式，稱為 JSON 序列化，這是一種非簡潔的表示形式，允許在同一 JWT 中使用多個簽章或接收者。

簡潔的序列化(The compact serialization)是對前兩個 UTF-8 字節的 JSON 元素（header 和 payload）以及進行簽章或加密的 data(不是 JSON 物件本身) 做 Base64 URL 安全編碼。三部分分別用一個 `.` 隔開，所以最後的結果會像這樣：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  # header
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.  # payload
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ  # signature
```

Header

每個 JWT 都有一個 header(又稱為 JOSE header)，是一種對自我的聲明，無論 JWT 是簽章的還是加密的，這些聲明都表示了其所使用的演算法，通常也會表示要如何解析 JWT 的其餘部分。

根據 JWT 的類型，header 中可能必須包含更多字段。例如，加密的 JWT 攜帶有關用於密鑰加密和內容加密的加密演算法的訊息。對於未加密的 JWT，這些字段則不會存在。

header 內含：

1. 必要欄位：

- **alg**：對此 JWT 進行簽章和(或)解密的主要演算法。對於未加密的 JWT，此聲明必須設置為 `none`。

1. 非必要欄位：

- **typ**：JWT 本身的媒體類型。此參數僅助於將 JWT 與帶有 JOSE header 的其他對象混合使用的情況。實際上，這種情況很少發生。如果存在，則此聲明應設置為值 `JWT`。
- **cty**：內容類型。大多數 JWT 攜帶特定的聲明以及任意數據作為其 payload 的一部分，在這種情況下，不得設置內容類型聲明。對於 payload 本身是 JWT 自己（巢狀 JWT）的實例，此聲明必須存在並帶有值 `JWT`，用來表示需要進一步處理巢狀的 JWT。而巢狀 JWT 很少見，因此 cty 聲明很少出現在 header 中。

所以說如果一個未加密的 JWT，其 header 會是這樣：

```
{
  "alg": "none"
}
```

經過編碼後為：`eyJhbGciOiJub25lIn0`

Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

通常所有使用者有興趣的資訊都會被放在 payload 裡，另外，某些規範中定義的權利要求也可能存在。就像 header 一樣，payload 是一個 JSON 物件。儘管特定的權利要求具有明確的含義，但沒有權利要求是強制性的。JWT 規範指出應該忽略在實踐中無法理解的聲明。具有所附特定含義的權利要求被稱為 registered claims。

Registered Claims 包含：

- **iss**：issuer 的簡稱，用字串(case-sensitive) 或 URI 表示這個 JWT 的唯一識別的發行方。
- **sub**：subject 的簡稱，用字串(case-sensitive) 或 URI 表示這個 JWT 所夾帶的唯一識別訊息。

換句話說，此 JWT 中包含的聲明是關於對象的聲明。JWT 規範規定，此聲明在發行方的上下文中必須是唯一的，或者在不可能的情況下必須是全局唯一的。處理此聲明是特定於應用程序的。

- **aud**：audience 的簡稱，用單字串(case-sensitive) 或 URI 或陣列表示這個 JWT 唯一識別的預期接收者。換句話說，當此聲明存在，則讀取此 JWT 中的數據的一方必須在 aud 中找到自己，或者無視 JWT 中包含的數據。與 iss 和 sub 要求的情況一樣，該權利要求是專用的。
- **exp**：expiration(time) 的簡稱，一個用來表示特定日期和時間的數字，格式為 POSIX 定義的 `自紀元以來的秒數`，即 UNIX時間。此聲明設置了該 JWT 被視為無效的確切時間。一些實踐可能允許時間存在一定的偏差(考慮此 JWT 在到期日期後的幾分鐘內有效)。
- **nbf**：not before (time) 的簡稱，`exp` 的相反，格式同 `exp`，當前時間和日期必須等於或晚於該日期和時間。一些實踐可能允許一定的偏差。
- **iat**：issued at (time) 的簡稱，一個用來表示特定日期和時間的數字(格式同 `exp` 和 `nbf`)，即該 JWT 發行的時間。
- **jti**：JWT ID 的簡稱，一個字串表示這個唯一識別的 JWT。此聲明可用於區分具有其他相似內容的 JWT (例如，防止重放)。取決於實現以確保唯一性。

所有的聲明，只要不在 registered claims 裡的，不是 private claims 就是 public claims。

Private Claims：

是由 JWT 的使用者(消費者和生產者)定義的那些。換句話說，這些是用於特定情況的臨時聲明。因此，必須注意防止衝突(collisions)。

Public Claims:

在 IANA JSON Web Token 聲明註冊表上註冊的聲明(用戶可以註冊其聲明，以防止衝突)，或者是使用抗衝突名稱命名的聲明(例如，在名稱前添加 namespace)。

JSON Web Signatures(JWS)

JSON Web Signatures(JWS) 大概是 JWT 最廣泛使用的功能。通過將簡單的數據格式與定義明確的簽章演算法系列相結合，JWT 迅速成為在客戶端和中介之間安全共享數據的理想格式。

簽章的目的是允許一個或多個參與方建立 JWT 的真實性(authenticity)。真實性意味著 JWT 中包含的數據未被篡改。換句話說，可以執行簽章檢查的任何一方都可以依靠 JWT 提供的內容。需要強調的是，簽章不會阻止其他方讀取 JWT 中的內容。

檢查 JWT 簽章的過程稱為驗證(validation)或 token 驗證(validating a token)。當滿足 header 和 payload 中指定的所有限制時，token 被視為有效。這是 JWT 的一個非常重要的方面：要求實現檢查 JWT 其 header 和 payload(以及用戶要求的任何內容) 到指定的點。因此，即使 JWT 缺少簽章(如果 header 將 `alg` 聲明設置為 `none`)，它也可以被視為有效。另外，即使 JWT 具有有效的簽章，也可能由於其他原因而被視為無效(例如，根據 `exp claim`，它可能已經過期)。對具有簽章的 JWT 常見的攻擊手法，依賴於剝離其簽章，然後更改 header 以使其成為不安全的 JWT。用戶有責任確保根據自己的要求對 JWT 進行驗證。

> 註：具簽章的 JWT 定義於 JSON Web Signature spec, RFC 7515。

JWS 的組成：

我們在前面有提到 JWT 的組成，這邊複習一下，並關注於結構中簽章(signature)的部分。 一個具簽章的 JWT 有三個元素：`header`、`payload`、`signature`

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

前兩個 header 和 payload 經過 Base64 decode 後結果如下：

```
{
  "alg": "HS256",
  "typ": "JWT"
}

{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

具簽章的 JWT 攜帶第三部分：signature，在簡潔序列化形式(compact serialization form)中的最後一個 `.` 之後。 根據 JWS 規範，有幾種可用的簽章演算法，因此這些八位位組的解釋方式各不相同。JWS 規範要求所有符合標準的實現都支持單一演算法：

- HMAC 使用 SHA-256，在 JWA 規範中稱為 HS256。

該規範還定義了一系列推薦演算法：

- 使用 SHA-256 的 RSASSA PKCS1 v1.5，在 JWA 規範中稱為 RS256。
- 使用 P-256 和 SHA-256 的 ECDSA，在 JWA 規範中稱為 ES256。

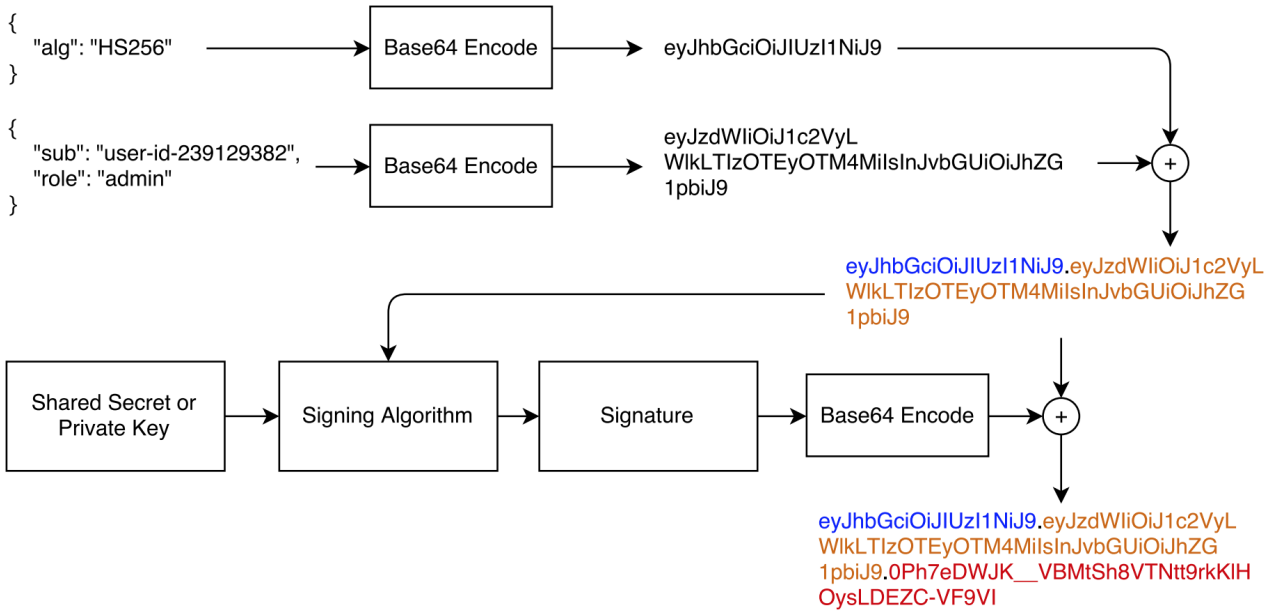
> 註：JWA 是 JSON Web Algorithms 規範 RFC 7518

其他演算法：

- HS384, HS512: SHA-384 and SHA-512 variations of the HS256 algorithm.
- RS384, RS512: SHA-384 and SHA-512 variations of the RS256 algorithm.
- ES384, ES512: SHA-384 and SHA-512 variations of the ES256 algorithm.
- PS256, PS384, PS512: RSASSA-PSS + MGF1 with SHA256/384/512 variants.

> 由於演算法很多種這裡就不詳細說明了，大家有興趣可以從參考資料下載相關文件研究。

JWS 整個完整的產生流程 (JWS Compact Serialization)：



從流程圖可以看出 header 和 payload 跟先前提到的一樣就是 JSON 物件經過 Base64 編碼後的結果，而 signature 則是可以透過 secret 或 private key 對編碼後的 header 加上 payload 經過演算法完成簽章，再將簽章用 Base64 編碼後的結果。

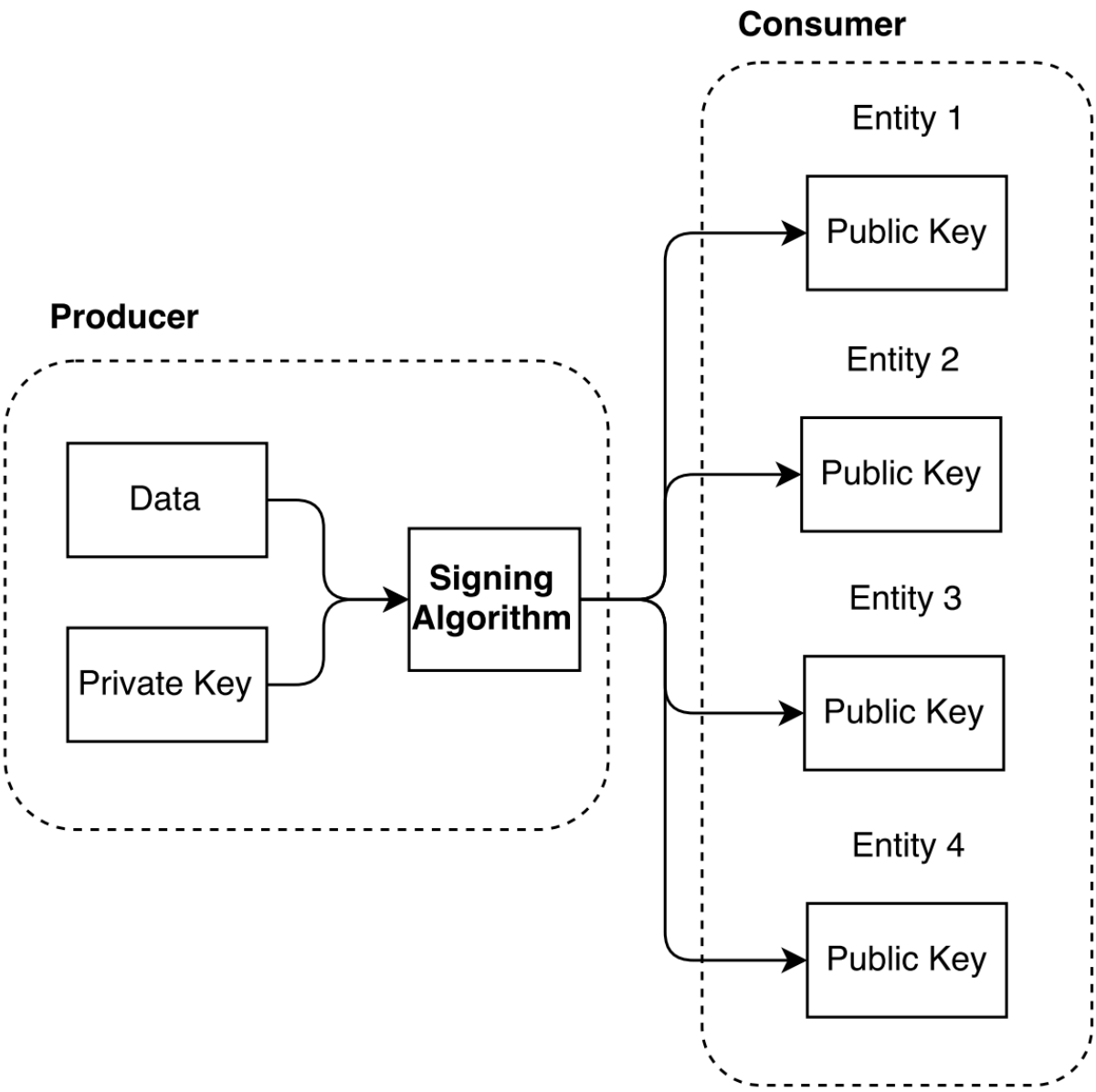
接著讓我們動手用 ruby 簡易實作 JWS 看看：

```ruby
require 'json'
require 'base64'
require 'openssl'

header = {
  "alg": "HS256",
  "typ": "JWT"
```

```
}

payload = {
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}

encoded_header = Base64.urlsafe_encode64(JSON(header)) # Base64 編碼 header
encoded_payload = Base64.urlsafe_encode64(JSON(payload)) # Base64 編碼 payload
digest = OpenSSL::Digest::SHA256.new # 使用 SHA256 演算法
data = "#{encoded_header}.#{encoded_payload}"

# For HMAC-based signing algorithms:
signature = OpenSSL::HMAC.digest(digest, 'secret', data) # 透過 HMAC 演算法簽章
encoded_signature = Base64.urlsafe_encode64(signature).gsub('=', '') # Base64 編碼 signature
puts "HMAC-based JWT: #{encoded_header}.#{encoded_payload}.#{encoded_signature}"

# For public-key signing algorithms:
key = OpenSSL::PKey::RSA.new(2048) # 產生 RSA 公鑰/私鑰
signature = key.sign(digest, data) # 用 key 進行簽章
encoded_signature = Base64.urlsafe_encode64(signature).gsub('=', '') # Base64 編碼 signature
puts "public-key JWT: #{encoded_header}.#{encoded_payload}.#{encoded_signature}"
```

註：有興趣深入研究的朋友可以詳讀 ruby-jwt 的 source code。

另外，如果是具有公私鑰的簽章方式(例如，RSA 演算法)，私鑰可用於創建簽章訊息並驗證其真實性。相反，公鑰只能用於驗證簽章訊息的真實性。因此，該方案允許**一對多**的安全分發訊息。接收方可以通過保留與訊息關聯的公共密鑰的副本來驗證訊息的真實性，但是他們不能使用它創建新訊息，如下圖：



而像是 `HMAC` 演算法這樣共享秘密(`secret`)簽章的方式，使用 HMAC + SHA-256，可以驗證訊息的任何一方也可以創建新訊息。如果合法用戶變為惡意用戶，則他可以修改訊息而無需其他方注意。用公鑰方案，變成惡意的用戶只能擁有他的公鑰，因此無法使用它創建新的簽章訊息。

JSON Web Encryption(JWE)

JWS 提供了一種驗證數據的方法，而 **JSON Web Encryption(JWE)** 提供了一種使數據對第三方不透明的方法。在這種情況下，不透明意味著不可讀。 加密的 token 不能由第三方檢查。

儘管看起來加密提供了與驗證相同的保證，並具有使數據無法讀取的附加功能，但情況並非總是如此。要理解為什麼，首先需要注意的是，就像在 JWS 中一樣，JWE 本質上提供了兩種方案：共享秘密方案和公鑰/私鑰方案。

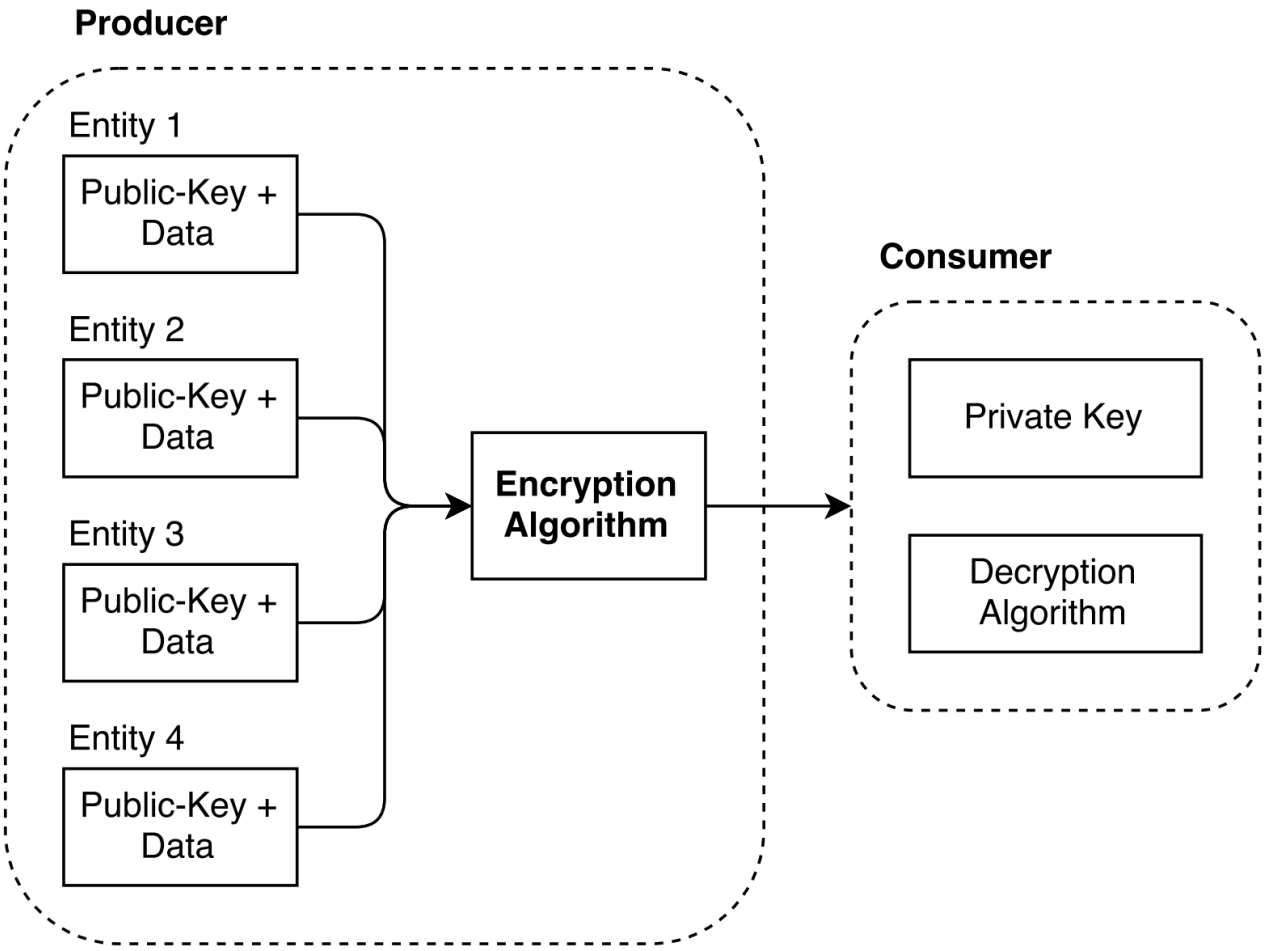
共享秘密方案通過讓各方知道共享秘密來起作用。擁有共享秘密的各方都可以加密和解密訊息。這類似於 JWS 中共享秘密的情況：擁有秘密的各方可以驗證並生成簽章 token。

但是，公鑰/私鑰方案的工作原理有所不同。在 JWS 中，擁有私鑰的一方可以簽章和驗證 token，而擁有公鑰的各方只能驗證那些 token，而在 JWE 中，擁有私鑰的一方是唯一可以解密 token 的一方。換句話說，公鑰持有者可以加密數據，但是只有持有私鑰的一方才能解密(和加密)該數據。實際上，這意味著在 JWE 中，持有公鑰的各方可以將新數據引入到交換中。相反，在 JWS 中，持有公鑰的各方只能驗證數據，而不能引入新數據。簡而言之，JWE 不能提供與 JWS 相同的保證，因此，它不能代替 JWS 在 token 交換中的作用。當使用公鑰/私鑰方案時，JWS 和 JWE 是互補的。

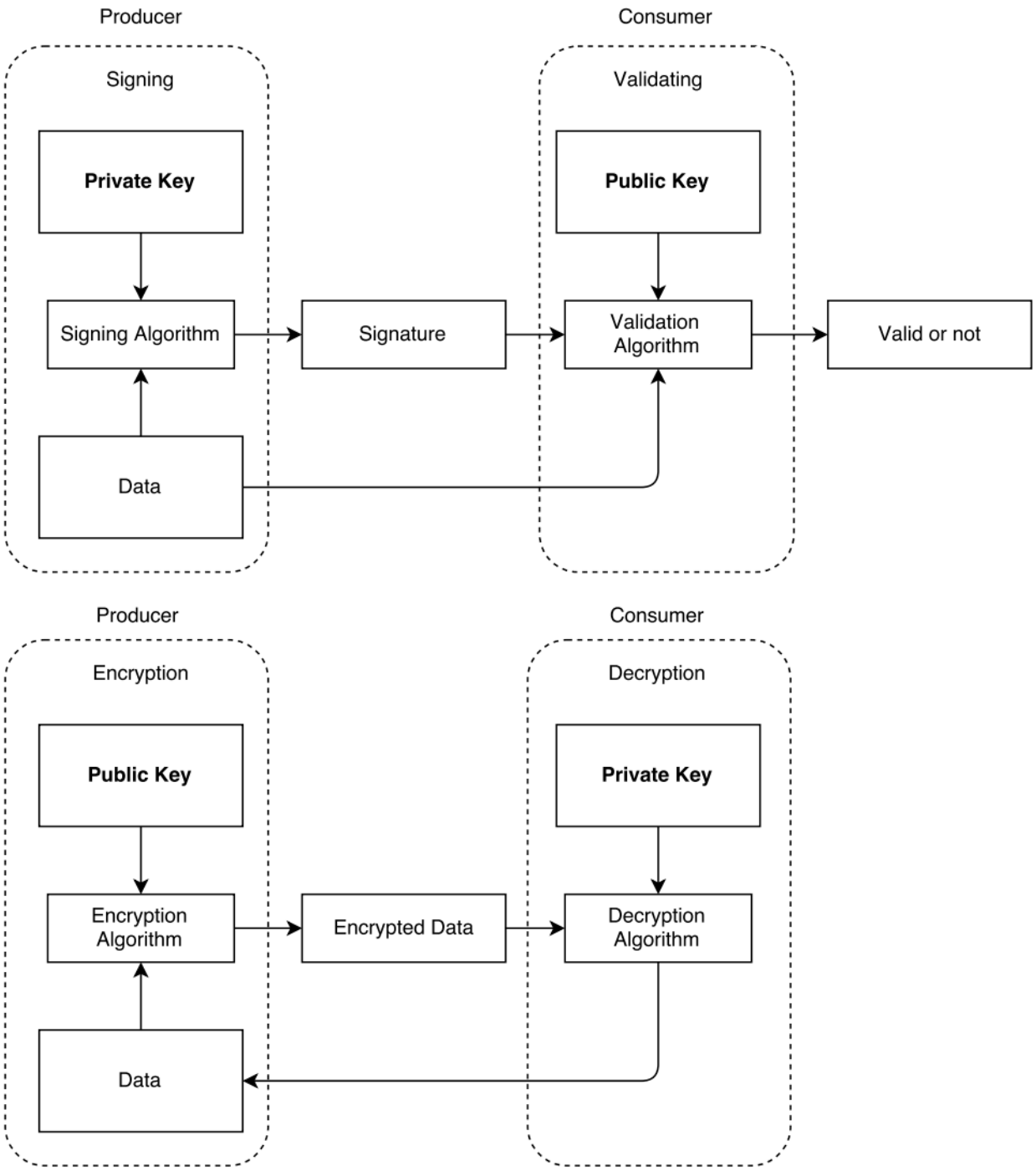
一種更簡單的理解方法是從生產者和消費者的角度進行思考。生產者可以對數據進行簽名或加密，因此消費者可以對其進行驗證或解密。對於 JWS，專用密鑰用於 JWS，而公用密鑰可用於對其進行驗證。生產者持有私鑰，而消費者持有公鑰。數據只能流動從私鑰持有者到公鑰持有者。相反，對於 JWE，使用公鑰對數據進行加密，而使用私鑰對數據進行解密。在這種情況下，數據只能從公鑰持有者流向私鑰持有者，即公鑰持有者是生產者，而私鑰持有者是消費者，比較表如下：

	JWS	**JWE**
Producer	Private-key	Public-key
Consumer	Public-key	Private-key

即 JWE 與 JWS 使用相同 RSA 演算法，JWE 可以使用公鑰加密訊息，而這些訊息只能使用私鑰解密。這允許構建**多對一**的安全通信通道，如下圖：



簽章和加密的流程比較圖如下：



註：加密的 JWT 定義於 JSON Web Encryption spec, RFC 7516。

JWE 的組成：

與具簽章和不安全的 JWT 相比，加密的 JWT 具有不同的表示形式：

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-
kFm1NJn8LE9XShH59_
i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7PcHALUzoOegEI-
8E66jX2E4zyJKxYxzZIItRzC5hlRirb6Y5Cl_p-
ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8Otv
zlV7elprCbuPhcCdZ6XDP0_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-
ljQTPcFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.
9hH0vgRfYgPnAHOd8stkvw
```

JWE 簡潔序列化(JWE Compact Serialization)包含五個元素，與 JWS 一樣，這些元素由點(`.`)分隔，並且包含在其中的數據是經過 Base64 編碼。

五個元素按照順序說明如下：

1. **The protected header**：類似 JWS 的 header。
2. **The encrypted key**：用於加密密文和其他加密數據的對稱密鑰。該密鑰是由用戶指定的實際加密密鑰加密後產生的。
3. **The initialization vector**：一些加密演算法需要的其他數據(通常是隨機的數據)。
4. **The encrypted data (ciphertext)**：加密的實際數據。
5. **The authentication tag**：該演算法產生的其他數據可用於驗證密文內容是否遭到篡改。

與簡潔序列化中的 JWS 及單一簽章一樣，JWE 支持簡潔形式的單一加密密鑰。

使用非對稱加密(公鑰/私鑰 加密)時，通常使用對稱密鑰執行實際的加密過程。而非對稱加密演算法通常具有很高的計算複雜度，因此對較長的數據序列(密文)進行加密並不是最理想的方式。想同時運用對稱(更快)和非對稱加密好處的一種方法是為對稱加密演算法生成一個隨機密鑰，然後使用非對稱演算算法對該密鑰進行加密。而這就是上面提到的第二個元素 `The encrypted key`。

一些加密演算法可以處理傳遞給它們的任何數據。如果密文被修改(即使沒有被解密)，演算法仍然可以對其進行處理。身份驗證標籤(`The authentication tag`)可以用來防止這種情況，本質上可以充當簽章。

> 註：關於使用加密演算法組成 JWE 的過程相對較複雜，這邊就不詳細說明了，有興趣的朋友可以詳細閱讀文末的參考資料。 JWK 是 JSON Web Keys 規範 RFC 7517，統一密鑰的表示格式允許輕鬆共享，並使密鑰有別於其他密鑰交換格式的複雜性。

總結

- JWS 和 JWE 是 JWT 的實作方式。
- JWA 和 JWK 是 JWT 中關於演算法和密鑰格式的規範。
- 目前普遍在用的 JWT 其實大多是 JWS，簡單說就是 Base64 Encode 過的 JSON + Signature。
- JWS 支援的 Signature 有單向跟雙向的，一般要用雙向(RS256)把公鑰跟私鑰分開，做驗證的時候才會相對安全，即發 Token 的 Server 跟 Client 用的是不同的 Key，兩邊都要驗證 JWT Signature 才能使用。
- JWS 只是有做簽章，其實是明碼傳輸，所以要注意不要放入太敏感的資料在其中，JWE 才是真正有作加密的方式。

參考資料

- Introduction to JSON Web Tokens
- JWT handbook