Part 7

REST API 설계

마이크로 서비스 구현의 핵심이 되는, REST API 에 대한 개념과 설계 방법과, 보안에 대해서 알아보고, 요즘 화두가 되고 있는 GraphQL에 대해서 알아본다.

REST 아키텍처

- 웹(HTTP)의 공동 창시자 Roy Fielding의 2000년 박사 논문에 소개 됨.
- 기존의 웹이 HTTP의 장점을 100% 활용하지 못하고 있음
- 네트워크 아키텍처 (Not a protocol)
- De facto Standard
- 오픈 진영(Google ,Amazon) 에 의해서 주도됨

REST의 구성 요소

01	리소스(Resource)	• 자원을 정의	HTTP URI
02	동사 (Verb)	• 자원에 대한 행위를 정의	HTTP Method
03	표현 (representation)	• 자원에 대한 행위의 내용을 정의	HTTP Body

REST API 예시

John Doe 라는 사용자를 생성하는 예시

 # POST /users

```
{
        "firstName": "John",
        "lastName": "Doe",
        "email": "john@example.com",
        "birthDate": "1990-01-01"
}
```

응답

```
{
        "id": 123,
        "firstName": "John",
        "lastName": "Doe",
        "email": "john@example.com",
        "birthDate": "1990-01-01",
        "createdAt": "2024-10-23T14:00:00Z",
        "updatedAt": "2024-10-23T14:00:00Z"
}
```

HTTP 메서드 설계

1	**GET**	리소스 조회	GET /users (모든 사용자 조회), GET /users/{userId} (특정 사용자 조회)
2	**POST**	리소스 생성	POST /users (새 사용자 생성)
3	**PUT**	리소스 전체 수정	PUT /users/{userId} (특정 사용자의 정보 전체 수정)
4	**PATCH**	리소스 일부 수정	PATCH /users/{userId} (특정 사용자의 일부 정보 수정)
5	**DELETE**	리소스 삭제	DELETE /users/{userId} (특정 사용자 삭제)

리소스

모든 개체를 **리소스라는** 단위로 표현

- /{그룹명}/{리소스 ID}식으로 표현
- 예: /users/terry

REST API 설계 원칙

01	명확한 Resource 식별	REST API는 리소스를 고유한 URI로 식별하며, 리소스를 나타내는 URI는 명사로 표현된다. URI는 계층적 구조를 따르고, 동작이 아닌 리소스 중심으로 설계되어야 한다. 이를 통해 명확하고 일관성 있는 API를 제공한다.
02	HTTP 메서드 기반 리소스 조작	REST는 HTTP 메서드(GET, POST, PUT, PATCH, DELETE)를 사용하여 리소스를 조작한다. 각 메서드는 특정 작업을 수행하며, 이들을 적절히 사용해 리소스에 대한 CRUD 작업을 일관성 있게 처리한다.
03	Stateless (무상태성)	REST API는 무상태로 동작하며, 각 요청은 독립적으로 처리된다. 서버는 클라이언트의 상태를 저장하지 않고, 요청 시 필요한 모든 정보를 함께 전송해야 한다. 이를 통해 확장성과 성능이 향상된다.
04	표준화된 응답 코드 사용	REST API는 HTTP 상태 코드를 사용해 요청 결과를 알린다.
05	캐시 가능	REST 응답은 캐시 가능 여부를 명시해야 한다.

REST 캐싱

REST도 HTTP이기 때문에 응답을 캐싱 가능

- CDN에 응답을 캐싱
- 웹케시 솔루션 (nginx 캐시, Varnish 캐시)를 사용

REST API 단점

De-facto 표준

- 명시적인 표준이 없다.
- 관리가 어렵다.
- 좋은 디자인을 가이드 하기가 어렵다.

RESTful 한 설계가 필요

- RDBMS는 관계형으로 리소스를 표현하지 않음. REST한 테이블 구조 설계가 필요
- 그래서 JSON을 그대로 저장하는 도큐먼트 기반의 NoSQL이 잘 맞음 (ex. MongoDB)

REST API 설계

REST API 설계팁

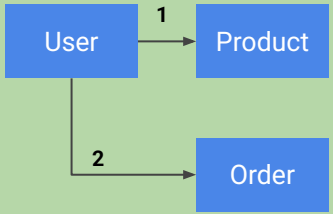
1. User Story

Feature (Lv 1)	Feature (Lv 2)	Details
사용자 가입 및 로그인	사용자는 구글 계정으로 로그인 한다.	사용자는 구글 계정으로 로그인한다.
	사용자는 페이스북 계정으로 로그인한다.	사용자는 페이스북 계정으로 로그인한다
비디오 피드 리스트 보기	사용자는 최근 업데이트된 순서대로 비디오 피드를 본다	사용자는 상하로 스크롤되는 비디오 피드를 볼 수 있다. 비디오 피드는 1초간 스크롤하지 않으면 현재 화면에 있는 비디오가 미리보기로 플레이된다.
비디오 촬영 및 업로드	사용자는 앱에있는 비디오를 업로드 할 수 있다.	사용자는 앱에 등록하는 비디오를 앱에서 촬영할 수 있다.
	사용자는 촬영한 비디오를 업로드 할 수 있다.	사용자는 촬영한 비디오를 촬영이 끝나자 마자 업로드하거나 또는 비디오 갤러리에서 이미 촬영된 비디오를 선택하여 업로드한다. 사용자는 비디오 업로드시 제목과,설명을 추가하여 업로드한다. 업로드시에 위치 정보를 같이 업로드 한다. GPS 정보를 이용하여 위치를 업로드 하거나 또는 근처에 이미 등록된 업체정보한것이 있을 경우, 업체정보한 이름을 목록으로 보여주고 선택할 수 있게 한다.
	사용자는 비디오를 업로드할때 해쉬태그를 추가할 수 있다.	사용자는 비디오 등록시, 설명 부분에 해쉬태그를 추가할 수 있다.
	사용자는 비디오 등록시 업체등반장을 등록할 수 있다.	사용자는 비디오 등록시 업체등반장을 GPS 정보를 이용하여 등록할 수 있다. 업체등반장은 이름과 위치로 등록하며, 이미 다른 사용이 등록했을 경우 유사한 이름의 등반장은 허나로 한다.
비디오 보기	사용자는 피드에서 비디오를 선택해서 플레이할 수 있다.	사용자가 비디오 피드에서 비디오를 클릭하면 비디오가 재생된다.
	사용자는 비디오를 플레이중에 비디오에 대한 댓글을 달 수 있다.	사용자는 비디오를 재생하는 화면에서 댓글을 달아서 자신의 의견을 등록할 수 있다.
	사용자는 비디오의 댓글중에 부적절한 댓글을 신고할 수 있다.	사용자는 비디오에 적절하지 않은 댓글이 있을 경우 신고하기 버튼을 이용하여 관리자에게 신고하고 삭제를 요청할 수 있다. 신고된 댓글은 관리자의 판단이 있을때까지 내용을 표시하지 않고 대신, "신고된 댓글입니다."메모 표시한다.
	사용자는 부적절한 비디오를 신고할 수 있다.	사용자는 비디오 재생하기나 또는 목록보기에서 적절하지 않은 비디오를 신고할 수 있으며, 신고에 대한 내용을 간단한 메모로 작성할 수 있다.
	사용자는 비디오를 추천할 수 있다.	사용자는 비디오를 재생하던지 마음에 드는 비디오에 "추천하기" 버튼을 누를 수 있다.
비디오 검색	사용자는 비디오 검색을 위해서 사진을 촬영할 수 있다.	사용자는 유사한 업체등반 코스를 검색하기 위해서, 실내 업체등반 코스를 사진을 촬영할 수 있다.
	사용자는 촬영한 사진을 이용해서 유사한 업체의 비디오를 검색할 수 있다.	사용자는 이미 갤러리에 저장된 실내 업체등반 코스나 또는 방금 촬영한 업체등산 코스 사진을 이용하여, 유사한 코스의 비디오를 검색할 수 있다. 일반 사용자는 일주일에 10개의 비디오를 사진 검색을 통해서 검색할 수 있으며, 유료 사용자는 무제한으로 검색할 할 수 있다.
	사용자는 비디오를 검색할때, 현재 위치와 유사한 위치에 있는 비디오를 검색할 수 있다.	사용자가 사진 기반으로 검색할때, 유사한 비디오를 사진뿐만 아니라, 사용자가 GPS를 활성화한 경우, GPS 정보를 이용하여, 반경 2KM-20KM 사이에 촬영한 영상을 우선으로 검색한다.
	사용자는 비디오를 검색할때 검색어를 추가하여 검색할 수 있다.	사용자는 검색시, 필수적인 검색 키워드를 추가하여 검색할 수 있다. 이 검색 키워드는 비디오 설명에 있는 텍스트와 해쉬 태그를 검색한다.
저장하기	유료 사용자는 자신이 마음에 드는 비디오를 북마크식으로 저장해놓을 수 있다.	유료 사용자는 자신에게 유용한 비디오를 북마크를 통해서 저장할 수 있다.
	유료 사용자는 북마크에 저장한 비디오의 목록을 볼 수 있다.	유료 사용자는 자신이 북마크한 비디오 목록을 볼 수 있다.

2. Architecture (Application/Dynamic)

Dynamic architecture

Use case 2.
As a user I want to purchase products



1. User requests product detail
2. User create order with (productId,quantity,price)

3. REST API Design Doc (recommend : Excel first)

```
Create Order
POST /orders/

{
  productid: 1232
  quantity: 3
  price : 5000
}
```

#	Description	method	URL	body	Note
2	Create Model	POST	/api/v1/models/	{ "url":"http://s3.~~~", "name":"Titanic", "library":"tensorflow" }'	model의 input/output parameter

3	Create deployment	POST	/api/v1/deployments	{
				"name": "TitanicServing",
				"modelid": 22,
				"version": "gnk43j",
				"strategy": "Rolling",
				"affinity":"gpu",
				"node":{
				"spec":{
				"cpu":"1",
				"memory":"1Gb",
				"gpu":"4",
				}
					node --> pod
					GPU --> gpu 종류 (T4, P100,V100)
					cf. google cloud endpoint
					pod min, max

5	Inference	POST	api/v1/deployments/:id/predict	{
				input:{
				:
				}
				}	

4	Get Deployment status	GET			현재 pod 수 리턴
1	User sign up	POST	/api/v1/auth/sighup		

한눈에 비교할 수 있게 엑셀(구글시트) 최소한 URI 까지

REST API 설계 가이드

- Microsoft REST API design guide : https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
- Google REST API design guide https://cloud.google.com/apis/design

리소스는 명사로 표현

URI는 동작이나 행동보다는 **리소스**(대상)를 식별하는 데 초점을 맞춰야 한다. **명사**를 사용하여 리소스를 표현하는 것이 좋다.

- 올바른 예: /users, /orders, /products
- 잘못된 예: /getUsers, /createOrder, /deleteProduct [안티패턴/터널링]

리소스는 복수형으로 사용

리소스 이름은 일반적으로 **복수형**을 사용한다. 이는 리소스 컬렉션을 나타내며 RESTful API의 통일성을 높인다.

- 올바른 예: /users, /products
- 잘못된 예: /user, /product

URI 경로는 계층 구조를 반영

URI는 리소스 간의 관계를 표현할 수 있어야 하며, 경로는 계층 구조를 따르는 것이 좋다.

- 예: /users/{userId}/orders/{orderId} (특정 사용자의 특정 주문)

슬래시는 URI의 경로를 구분하는 데 사용되며, 계층적 관계를 표현하는 역할을 한다.
트레일링 슬래시(URI의 끝에 슬래시를 붙이는 것)는 혼란을 줄 수 있으므로 일관성을 유지하는 것이 중요하다.

하이픈(-)과 밑줄(_) 사용

하이픈(-)은 여러 단어로 된 URI를 구분할 때 사용하고, **밑줄(_)**은 가독성을 떨어뜨릴 수 있으므로 지양한다.

- 올바른 예: /order-items
- 잘못된 예: /order_items

페이징, 정렬, 필터링

페이지

많은 양의 데이터를 한 번에 반환하지 않고, 필요한 만큼 나눠서 제공한다. 클라이언트는 필요한 페이지를 요청할 수 있어야 한다.

- 예: /users?page=2&limit=50
- Facebook 스타일 : /records?offset=100&limit=25
  (100번째 레코드부터 25개 출력)
- Twitter 스타일 : /records?page=5&rpp=25
  (페이지당 25개일때, 5페이지 출력)
- Linked in 스타일 : /records?start=50&count=25
  (50번째 레코드에서 25개 출력)

```
{
        "data": [
                {
                        "id": 1,
                        "name": "User 1"
                },
                {
                        "id": 2,
                        "name": "User 2"
                }
        ],
        "pagination": {
                "currentPage": 1,
                "totalPages": 10,
                "totalItems": 198,
                "itemsPerPage": 20
        }
}
```

검색

- 보통 HTTP GET에 쿼리 스트링을 사용함

  예)*users?name=cho&region=seoul&offset=20&limit=10*

- 다른 쿼리 스트링 (페이징과 섞여서 헷갈릴 수 있음) 검색 조건을 별도의 필드로 뽑아내는게 좋음

  예) */user?q=name%3Dcho,region%3Dseoul&offset=20&limit=10*

- 전역 검색

  예) 모든 리소스에 대한 검색 : /search?q=id%3Dterry

- 지역 검색

  예)특정 리소스에 대한 검색 : /users?q=id%3Dterry

부분 응답 (Partial Response)

REST API 응답중 일부만 응답 받는 방식

- Linked in : /people:(id,first-name,last-name,industry) 파싱하기 어려움
- Facebook : /terry/friends?fields=id,name 직관적
- Google : ?fields=title,media:group(media:thumbnail) subobject를 지원하기 때문에 유리함
- 또는 : ?field=title,media.address.city

많이 쓰지는 않지만 종종 유용함

- 코드내에 JOIN(REFERENCE)가 있을때, 부하를 줄일 수 있음
- 전체 패킷양을 줄일 수 있음
- **컬럼 데이터 베이스 사용시 유용함**

HATEOS

- Hypermedia as the engine of application state
- HTML LINK 개념으로 다른 리소스 또는 다른 연관된 행위에 대한 링크를 제공함으로써 SELF-DESCRIPTIVENESS를 극대화함
- 좋기는 한데, 의존성때문에 업데이트 하기가 쉽지 않음

```
HTTP GET users?offset=10&limit=5
{
        [
                {‘id’:’user1’,’name’:’terry’},
                {‘id’:’user2’,’name’:’carry’}
        ]
        ,”links“:[
                {rel’:’pre_page’,‘href’:’http://xxx/users
                ?offset=6&limit=5‘},
                {‘rel’:’next_page’,‘href’:’http://xxx/use
                rs?offset=11&limit=5‘}
        ]

}
```

응답 코드

성공한 응답 코드

- **200 OK**: 요청이 성공했으며, 응답 본문에 리소스가 포함된다.
- **201 Created**: 새로운 리소스가 성공적으로 생성되었음을 나타낸다. 이때 Location 헤더에 새로 생성된 리소스의 URI를 포함해야 한다.
- **204 No Content**: 요청은 성공했으나 응답 본문에 데이터가 없을 때 사용한다. (예: DELETE 요청 후)

클라이언트 오류 상태

- **400 Bad Request**: 잘못된 요청으로 인해 서버가 요청을 처리할 수 없을 때 사용한다.
- **401 Unauthorized**: 인증이 필요하지만 제공되지 않았거나 유효하지 않을 때 사용한다.
- **403 Forbidden**: 인증은 되었으나, 요청된 작업에 대한 권한이 없을 때 사용한다.
- **404 Not Found**: 요청한 리소스가 존재하지 않음을 나타낸다.

서버 상태 오류

- **500 Internal Server Error**: 서버에서 예기치 않은 오류가 발생했을 때 사용된다.
- **503 Service Unavailable**: 서버가 일시적으로 과부하 상태이거나 유지보수 중일 때 사용한다.

에러 처리

에러가 발생했을 때는 클라이언트가 원인을 명확히 알 수 있도록 **표준화된 에러 메시지 형식**을 제공해야 한다. 일반적으로 JSON 형식을 사용한다.

반환하는 HTTP 상태 코드와 에러 메시지는 일관성을 가져야 한다. 예를 들어, 404 상태 코드를 반환할 때는 "리소스를 찾을 수 없음"이라는 메시지가 적절하다.

```
HTTP/1.1 404 Not Found
Content-Type: application/json
{
        "error": {
                "code": 400,
                "message": "Invalid input data",
                "details": "The 'email' field must be a valid
                        email address."
        }
}
```

에러 처리

- Error Stack은 response 메시지에 포함 시키지 않는게 좋음
- 어떤 기술 스택을 사용하는지, 파일 위치등 해킹 가능한 자료가 유출될 수 있음
- 로그 레벨 옵션을 둬서, 개발이나 QA계에서만 출력하도록 하는게 좋음

```
log4j:ERROR setFile(null,true) call failed.
java.io.FileNotFoundException: stacktrace.log (Permission denied)
at java.io.FileOutputStream.openAppend(Native Method)
at java.io.FileOutputStream.(FileOutputStream.java:177)
at java.io.FileOutputStream.(FileOutputStream.java:102)
at org.apache.log4j.FileAppender.setFile(FileAppender.java:290)
at org.apache.log4j.FileAppender.activateOptions(FileAppender.java:164)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

버전관리

URI에 포함

```
GET /v1/resource
GET /v2/resource
```

- 명확하고 직관적, 테스트가 편함
- REST 디자인 원칙에서 다소 벗어남 (URL이 API 리소스를 식별)
- **Google, 가장 일반적**

Sub domain에 포함

```
GET https://v1.api.example.com/resource
```

- API 서버를 물리적으로 분리하여, 독립 운영 가능
- DNS, 네트워크 설정 필요

Header에 포함

```
GET /resource Headers: Accept:
application/vnd.myapi.v1+json
```

- 클라이언트 추가 작업이 필요하고, 브라우저에서 테스트하기 어려움

Query Parameter에 포함

```
GET /resource?version=1
```

- 간단한 버전 관리
- 캐싱이 복잡해짐
- **Microsoft**

안티패턴

- GET/POST를 이용한 터널링
- SELF-DESCRIPTIVENSS 속성을 사용하지 않음 (이해하기 쉽게 만들라는 이야기)
- HTTP RESPONSE CODE를 제대로 사용하지 않음

```
HTTP POST, http://myweb/users/

{
  "getuser":{
    "id":"terry",
  }
}
```

REST API 보안

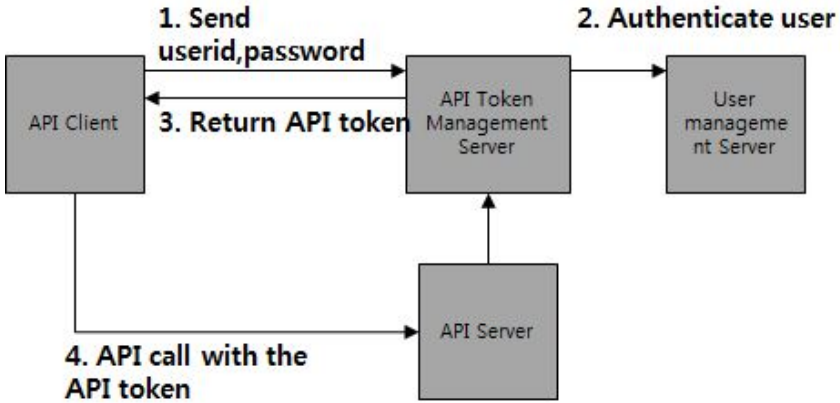
인증 (Authentication)

API 키 방식

- 가장 기초적인 인증 방식.
- API를 호출할때, API KEY를 보내는 방식 (모든 클라이언트가 같은 키를 공유)
- 구현은 쉬우나 한번 KEY가 노출되면 모든 ACCESS가 가능해짐

API 토큰 방식

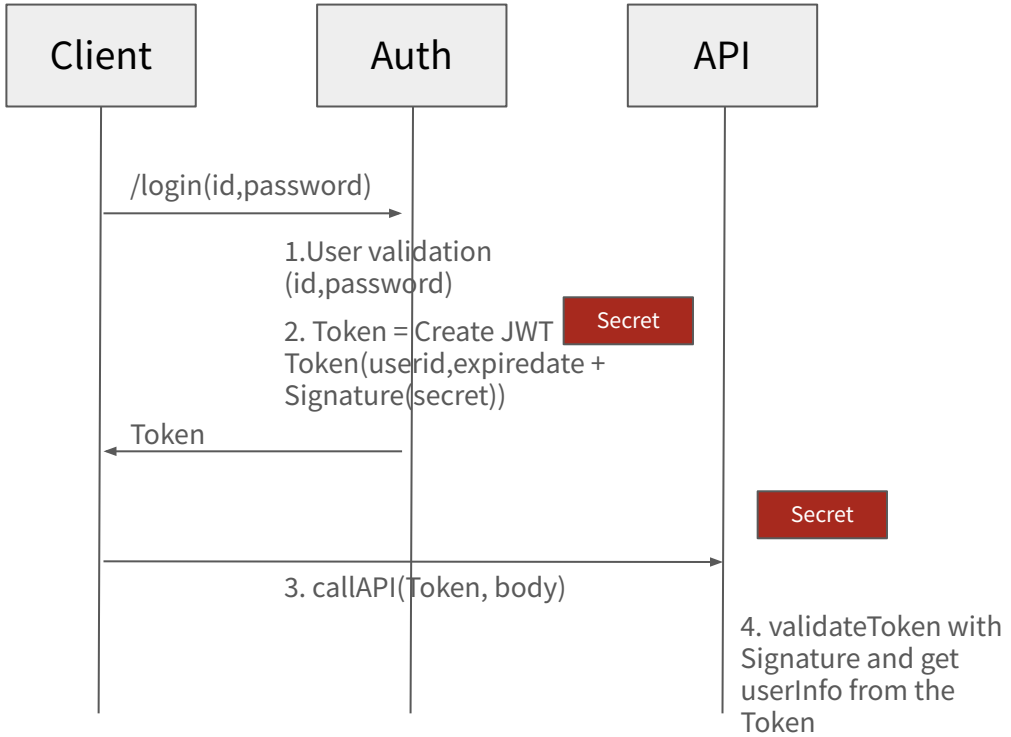
- 가장 널리 사용되는 방식
- ID,PASSWORD를 넣으면, 일정 기간 유효한 API 토큰을 리턴하고, 매 호출마다 이 토큰을 사용



JWT 토큰

1. 사용자가 인증된 후, 서버에서는 비밀키로 서명된 토큰을 발급 (BASE 64로 인코딩됨)
2. 클라이언트에서는 토큰을 이용하여 API를 호출
3. 서버에서는 비밀키를 이용하여 토큰을 검증하고, 토큰에 있는 정보 (사용자, 유효기간)으로 사용자 정보를 받아서 사용함

- 암호화는 HS.256 등 대칭키 방식과, PS256과 같은 비대칭키 방식 모두 지원



JWT Token 예시

Header (서명 알고리즘)

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

Payload (데이터)

```
{
  "sub": "1234567890",              // subject, 사용자 ID
  "name": "John Doe",               // 사용자 이름
  "admin": true,                    // 관리자인지 여부
  "iat": 1516239022,                // issued at, 토큰이 발행된 시간 (Unix timestamp)
  "exp": 1516242622,                // expiration, 토큰 만료 시간 (Unix timestamp)
  "aud": "your-app.com",            // audience, 토큰을 받을 대상 (예: 애플리케이션)
  "iss": "your-auth-server.com"     // issuer, 토큰을 발행한 주체
}
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMiwiZXhwIjoxNTE2MjQyNjIyLCJhdWQiOiJ5b3VyLWFwcC5jb20iLCJpc3MiOiJ5b3VyLWF1dGgtc2VydmVyLmNvbSJ9.WGtbsXouGJnA8VV5MxWvrSmGwMzdnPLV-wGSkTQEAUQ
```

Signature(서명)

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

JWT 토큰의 장점

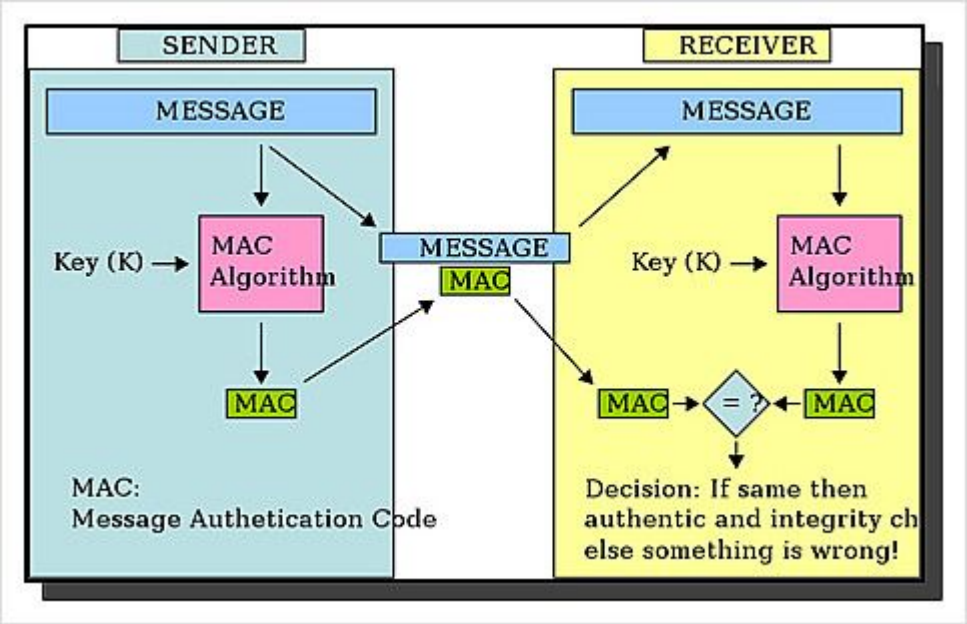
- **무상태성**: 서버는 토큰을 기억할 필요가 없으므로 무상태(stateless)로 작동한다.
- **확장성**: JWT는 여러 서비스 간에 쉽게 전파할 수 있다.
- **유연성**: 클라이언트의 다양한 데이터 (예: 유저 권한 정보 등)를 토큰에 담을 수 있다.

HMAC을 이용한 유효성 검증

메시지 무결성 보장 (변조 방지)

1. 클라이언트는 호출하고자 하는 REST API의 BODY를 대칭 키를 이용하여 HMAC 알고리즘을 사용하여 Hash 값을 추출한다.
2. API를 호출할 때, 메시지(또는 URL)에 추출한 HMAC을 포함해서 호출한다.
3. 서버는 호출된 URL을 보고 HMAC을 제외한 나머지 URL을 미리 정의된 Key를 이용해서, HMAC 알고리즘으로 Hash 값을 추출한다.
4. 서버는 3에서 생성된 HMAC 값과 API 호출 시 같이 넘어온 HMAC 값을 비교해서, 값이 같으면 이 호출을 유효한 호출이라고 판단한다.

만약에 만약 해커가 메시지를 중간에서 가로채어 변조하여 했을 경우, 서버에서 Hash를 생성하면 변조된 메시지에 대한 Hash가 생성되기 때문에 클라이언트에서 변조 전에 보낸 Hash 값과 다르게 된다. 이를 통해서 통해 메시지가 변조되었는지 여부를 판단할 수 있다.



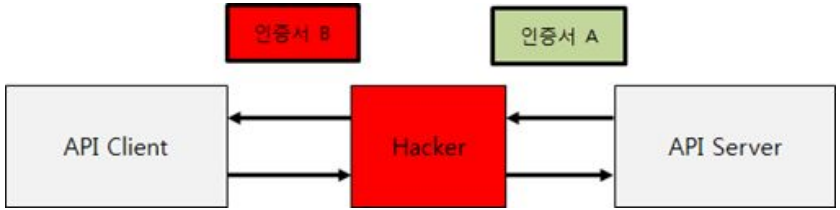
출처 : Wikipedia https://en.m.wikipedia.org/wiki/File:MAC.svg

네트워크 레벨 보안

- SSL 인증서를 이용하여 네트워크 레벨 보안 가능
- Man In the Middle Attack에 취약할 수 있음
- 공인 인증서를 사용하도록 하고, 인증서 발급자를 확인하는 로직을 추가하여 Man In the Middle Attack 방어.

정상적인 SSL 호출

Man in the middle attack

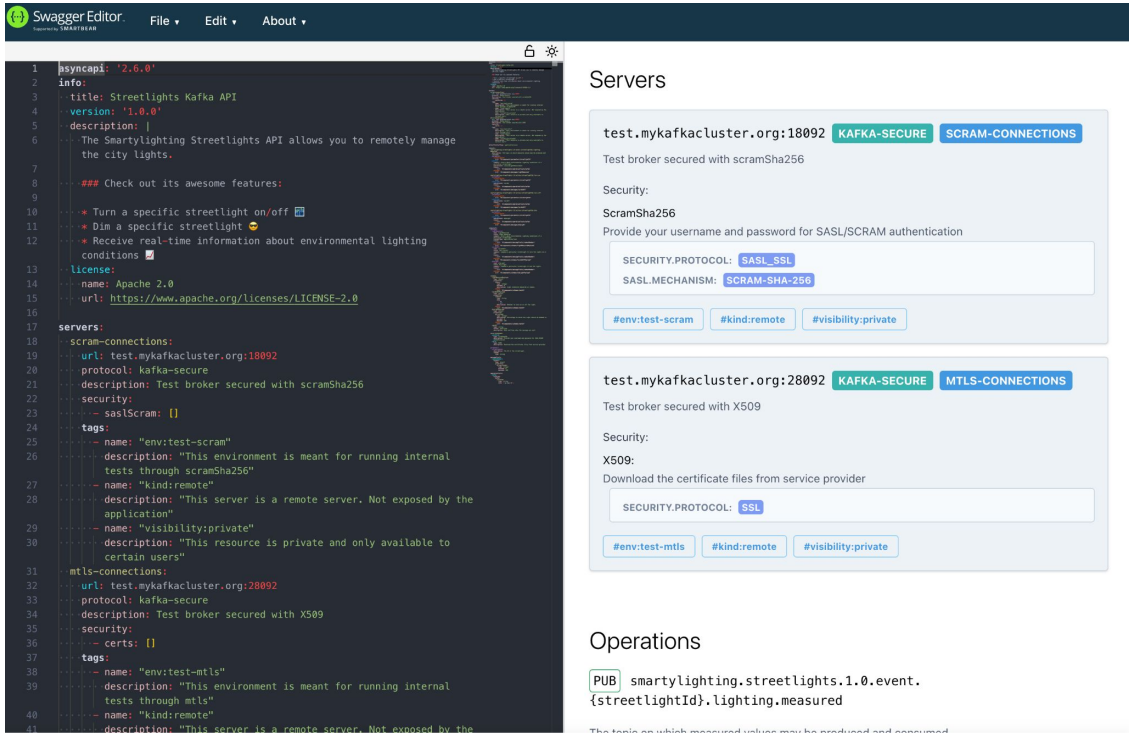


REST API 문서화

Swagger

- REST API 설계, 문서화, 테스트 그리고 디버깅을 지원하는 OSS
- OpenAPI Spec (OAS)를 중심으로 개발됨
- 구성
  - Swagger UI : API 문서를 UI로 제공하고 직접 호출할 수 있음
  - Swagger Editor : 웹 기반의 편집기로 OpenAPI 문서를 YAML또는 JSON으로 편집 가능
  - Swagger Codegen : OpenAPI Spec 파일에서 클라이언트 SDK, 서버 스켈레톤 코드, 문서를 자동 생성 가능

Swagger Editor



- 웹상에서 YAML,JSON을 이용하여 REST API SPEC 문서 생성 가능
- 초기 디자인시 손쉽게 사용 가능

https://editor-next.swagger.io/

Swagger in the Code

```
import io.swagger.v3.oas.annotations.media.Schema;

@Schema(description = "주문 상품 정보")
public class OrderItem {

    @Schema(description = "상품 ID", example = "201")
    private Long productId;

    @Schema(description = "상품 이름", example = "노트북")
    private String productName;

    @Schema(description = "구매 수량", example = "1")
    private int quantity;

    public OrderItem() {}
    public OrderItem(Long productId, String productName, int
quantity) {
        this.productId = productId;
        this.productName = productName;
        this.quantity = quantity;
    }
```

- YAML이나 JSON 처럼 별도 파일이 아니라, 코드 상에서 annotation을 사용하여, code + API spec을 함께 작성 가능
- 코드와 Spec의 일관성
- 코드 가독성이 떨어질 수 있음

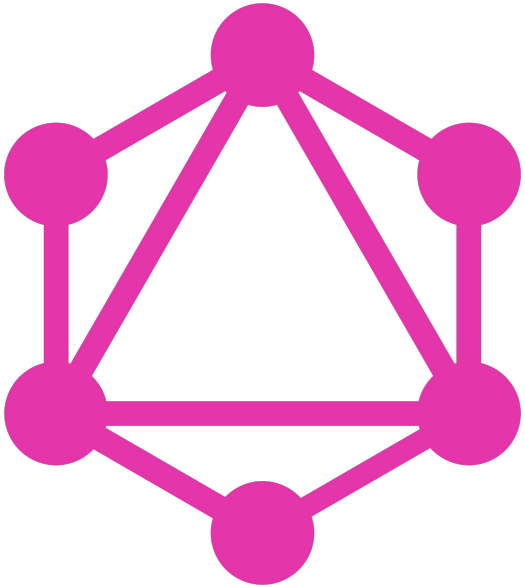
Swagger in the Code

```java
RestController
@RequestMapping("/api/orders")
public class OrderController {

    @Operation(summary = "주문 조회", description = "ID로 주문을 조회합니다.", security = @SecurityRequirement(name = "bearerAuth"))
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "성공", content = @Content(schema = @Schema(implementation = Order.class))),
            @ApiResponse(responseCode = "404", description = "주문을 찾을 수 없음")
    })
    @GetMapping("/{id}")
    public ResponseEntity<Order> getOrder(@Parameter(description = "주문 ID", required = true) @PathVariable Long id) {
        User user = new User(1L, "홍길동", 30);
        List<OrderItem> items = Arrays.asList( new OrderItem(201L, "노트북", 1), new OrderItem(202L, "마우스", 1));
        Order order = new Order(id, LocalDateTime.now(), user, items);

        return new ResponseEntity<>(order, HttpStatus.OK);
    }
```

GraphQL

GraphQL이란?

- 데이터 액세스(쿼리)에 최적화된 기술
- HTTP로 Query를 보내서 데이터를 CRUD하는 프로토콜
- REST가 여러개의 엔드포인트를 사용하는데 반해, GraphQL은 하나의 엔드포인트에 다른 쿼리를 전달해서 결과를 받는 형태
- 2015년 페이스북에 의해서 공개되고, Twitter, GitHub등에서 사용됨
- SOA의 Data Centric Service 와 유사한 개념 (데이터 접근만을 지원)



GraphQL SDL (Schema Definition Laguage)

```
// index.js (node.js ApolloServer example)
const { ApolloServer } = require('@apollo/server');
const { startStandaloneServer } = require('@apollo/server/standalone');
const { users, posts } = require('./data');

// GraphQL 스키마 정의 (typeDefs)
const typeDefs = `#graphql
  type User {
    id: ID!
    name: String!
    email: String!
    posts: [Post!]!
  }

  type Post {
    id: ID!
    title: String!
    content: String!
    author: User!
  }

  type Query {
    user(id: ID!): User
    users: [User!]!
    post(id: ID!): Post
    posts: [Post!]!
  }

  type Mutation {
    createUser(name: String!, email: String!): User!
    createPost(title: String!, content: String!, authorId: ID!): Post!
  }
```

DSL : GraphQL에 저장되는 데이터 타입, CRUD 인터페이스를 정의함

- type: 데이터 타입 정의
- query : 쿼리 구조 정의
- mutation : Update 쿼리 정의
- 이외에 : interface,union,input,directive 등 다양한 요소가 있음

GraphQL 쿼리 예제

http://localhost:4000/graphql

```
query {
  user(id: "1") {
    id
    name
    email
    posts {
      id
      title
    }
  }
}
```



```
{
  "data": {
    "user": {
      "id": "1",
      "name": "John Doe",
      "email": "john.doe@example.com",
      "posts": [
        {
          "id": "1",
          "title": "First Post"
        },
        {
          "id": "2",
          "title": "Second Post"
        }
      ]
    }
  }
}
```

프레임웍

- **Node.Js : Apollo Server** **(가장 인기 있음. 44,000 star in github)**
- Spring : Spring for GraphQL, DGS Framework (Netflix 개발)
- Python : Graphene python (7800 star)

  :

GraphQL

- HTTP 인터페이스를 통하여 데이터 엑세스가 편리함
- REST와는 완전히 다른 개념
- 장점
  - 한 쿼리에서 필요한 데이터를 모두 쿼리가 가능 (트래픽 감소)
  - 불필요한 필드를 제외하고 필요한 필드만 쿼리 가능
  - 단일 엔드포인트로 관리가 쉽고, 개발이 편리해짐
- 단점
  - 개념은 좋지만 높은 수준의 디자인이 필요.
  - 상대적으로 높은 러닝 커브
  - 캐슁하기 어려움 (단일 Endpoint)
  - 보안적인 위험 요소가 많음
    - 쿼리를 깊게 작성함으로써 무한 중첩 쿼리를 통한 공격이 가능 (User → Post →Comment → User → Post …)
    - 단일 endpoint이기 때문에, 권한 검사가 어려움
    - 메시지에 너무 많은 정보가 포함됨
  - 아직 eco system이 넓지 않음

감사합니다.