조대협의 서버 사이드

REST API 디자인

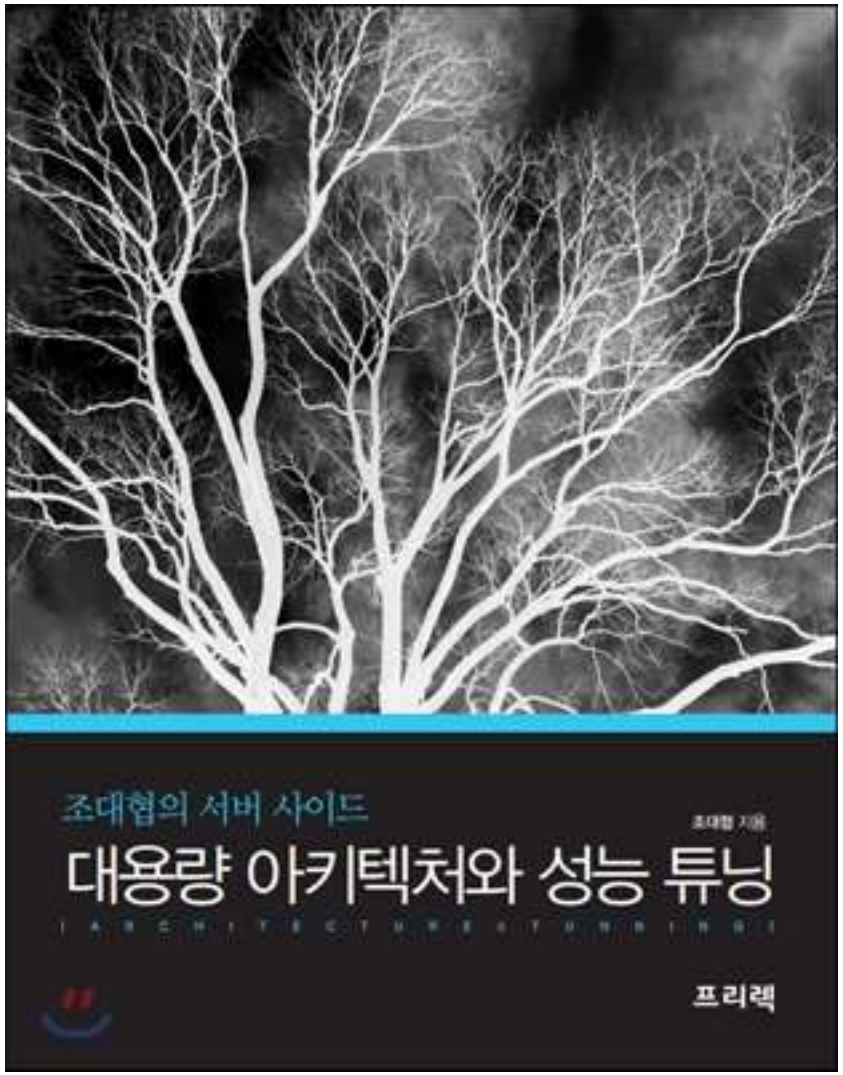
발표자 소개

조 대 협 본명: 조병욱

nhn Microsoft bea ORACLE SAMSUNG Pikicast

- 회원 13만명 온라인 개발자 커뮤니티 자바스터디(www.javastudy.co.kr) 운영자.. (기억의 저편..)
- 한국 자바 개발자 협회 부회장,서버사이드 아키텍트 그룹 운영자
- 벤쳐 개발자
- BEA 웹로직 기술 지원 엔지니어
- 장애 진단, 성능 튜닝
- NHN 잠깐
- 오라클 컨설턴트 (SOA,EAI,ALM,Enterprise 2.0,대용량 분산 시스템)
- MS APAC 클라우드 수석 아키텍트
- 프리렌서 (잘나가는 사장님)
- 삼성전자 무선 사업부 B2B팀 Chief Architect
- 전 피키캐스트 CTO로

이 문서의 내용은



조대협의 서버사이드

“대용량 아키텍처와 성능 튜닝” 중

4장 REST API의 이해와 설계를 정리한 내용

#목표

"대부분의 백앤드 시스템에 적용되는 REST API에 대한 올바른 디자인 방법에 대해서 알아보자.

#1

기본 개념 잡기

REST 아키텍쳐

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

REST 구성

구성 요소	내용	표현 방법
Resource	자원을 정의	HTTP URI
Verb	자원에 대한 행위를 정의	HTTP Method
Representations	자원에 대한 행위의 내용을 정의	HTTP Message Pay Load

REST의 기본

"사용자라는 Resource (//myweb/users) 이름이 "Terry"인 Representation으로 새로운 사용자를 생성 (HTTP POST) 한다. "

```
HTTP POST , http://myweb/users/
{
   "users":{
      "name":"terry"
   }
}
```

- HTTP 메서드

HTTP 메서드	의미	Idempotent
POST	생성	NO
GET	조회	YES
DELETE	삭제	YES
PUT	업데이트	YES

REST 예제

기능	HTTP 메서드	HTTP URL & BODY
모든 회원 정보 조회	HTTP GET	http://www.javastudy.co.kr/users
특정 회원 정보 조회	HTTP GET	http://www.javastudy.co.kr/users/terry
회원 정보 검색	HTTP GET	http://www.javastudy.co.kr/users?query=xxx

회원 등록	HTTP POST	http://www.javastudy.co.kr/users
		{
		"name":"terry",
		:
		}

회원 삭제	HTTP DELETE	http://www.javastudy.co.kr/users/terry

해당 회원 정보 변경	HTTP PUT	http://www.javastudy.co.kr/users/terry
		{
		"name":"terry",
		"address":"seoul"
		}

리소스

- 모든 개체를 “리소스” 라는 단위로 표현
- 보통 //{리소스 그룹명}/{리소스 ID} 로 표현
  - EX) //users/terry (사용자 중에서 테리)
- REST는 이 리소스에 대한 CRUD 행위로 표현
  - PUSH 메세지를 보낸다 → PUSH 메세지를 생성한다.

REST의 특성

- 유니폼 인터페이스
  - HTTP 표준만 따르면 어떠한 기술이든 사용가능 (HTTP/JSON, HTTP XML)
- 무상태성 (STATELESS)
- 캐싱 가능
  - 기존 웹 인프라를 그대로 사용 가능
  - 웹캐쉬, CDN을 이용한 캐싱
- 자체 표현 구조 (SELF-DESCRIPTIVENESS)
  - API 내용만 보고도 별도의 문서 없이도 쉽게 이해가 가능하다.

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)

안티 패턴

- GET/POST를 이용한 터널링

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

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

- SELF-DESCRIPTIVENSS 속성을 사용하지 않음 (이해하기 쉽게 만들라는 이야기)
- HTTP RESPONSE CODE를 제대로 사용하지 않음

#2

설계 패턴

URL을 심플하고 직관적으로

- API URL 만 보고도 무슨 API인지 이해하기 쉽게 직관적으로 (사실이게 제일 어려움)
- URL을 길게 만들지 말고 2 depth 정도만
  - /dogs/
  - /dogs/uki

동사 보다는 명사를 사용

- REST API는 대상에 대한 행동(CRUD)를 정의하는 개념이다.
- 대상은 명사가 되야 한다.
  - POST /dogs/

```
HTTP POST : /getDogs
HTTP POST : /setDogsOwer
```

나쁜 사용예

```
HTTP GET: /dogs
HTTP POST : /dogs/{puppy}/owner/{terry}
```

좋은 사용예

단수 보다는 복수형을 사용

- 리소스는 복수형 명사를 사용
- 매우 흔히 하는 실수
  - /dog (X)
  - /dogs

리소스간의 관계 표현방법

- Option A. 서브 URL을 사용하는 방법
  - GET /owners/{terry}/dogs
- Option B. 관계를 표현하는 리소스를 별도 정의 하는 방법
  - /resource/identifier/relation/other-related-resoulrce
  - GET /people/terry/ownsdog/puppy
- 리소스간 관계가 많지 않고 단순한 경우는 Option A, 관계가 많고 복잡한 경우는 Option B.

에러 처리

- HTTP Response code
  - 총 70개가 있음. 이중에서 선택
  - Google Gdata 10개, Netflix 9개, Digg의 경우 8개
  - 권장 Response code
    - 200 : 성공
    - 400 : Bad Request – Field validation 실패
    - 401 : 인증,인가 실패
    - 404 : 해당 리소스 찾을 수 없음
    - 500 : Internal Error – 서버 에러

에러 처리

- HTTP Response body
  - 상세 내용은 HTTP body에 표현하는 것이 좋음

```
{
   “error”          : 20003,
   “message”        : “Autorization failed”,
   “detail”         : “User doesn’t have permission to update user profile”,
   “moreinfo”       : http://myapi.com/docs/20003
}
```

  - Error code 범위를 미리 정해놓는게 좋은 2000번대 인증 관련, 30000번대 주문 관련
  - Error code 별 메세지와, Detail등은 별도의 파일로 관리하는 것이 다국어 지원에 유리함

에러 처리

- Error Stack 처리
  - Error Stack은 response 메세지에 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)

버전 관리

- 가급적이면 하위 호환성을 보장하는 것이 좋음 (JSON은 스키마리스)
- API 버전 정의 방법
  - Facebook : ?v=2.0
  - Salesforce : /service/data/v2.0/soobject/Account
- 추천 방법
  - api.server.com/{서비스명}/{버전}/{리소스}
  - ex) api.server.com/account/v2.0/groups
  - account1.war, account2.war로 새 버전을 별도의 배포 패키지로 관리하기가 편함

페이징

- 많은 도큐먼트를 리턴할때는 잘라서 리턴하는 페이징 처리가 필요
  - 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)
- 페이스북 API 스타일이 가장 직관적 나머지는 여러분의 선택

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:thumnail) 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
  - 가독성이 높아짐

검색

- 보통 HTTP GET에 쿼리 스트링을 사용함
  - *users?name=cho&region=seoul&offset=20&limit=10*
- 다른 쿼리 스트링 (페이징과 섞여서 헷갈릴 수 있음)
  - 검색 조건을 별도의 필드로 뽑아내는게 좋음
  - */user?q=name%3Dcho,region%3Dseoul&offset=20&limit=10*

검색

- 전역 검색과 리소스 검색
  - 전역 검색
    - 모든 리소스에 대한 검색
    - */search?q=id%3Dterry*
  - 지역 검색
    - 특정 리소스에 대한 검색
    - */users?q=id%3Dterry*

HATEOS

- Hypermedia as the engine of application state 약어
- 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/users?offset=11&limit=5'}
]
}
```

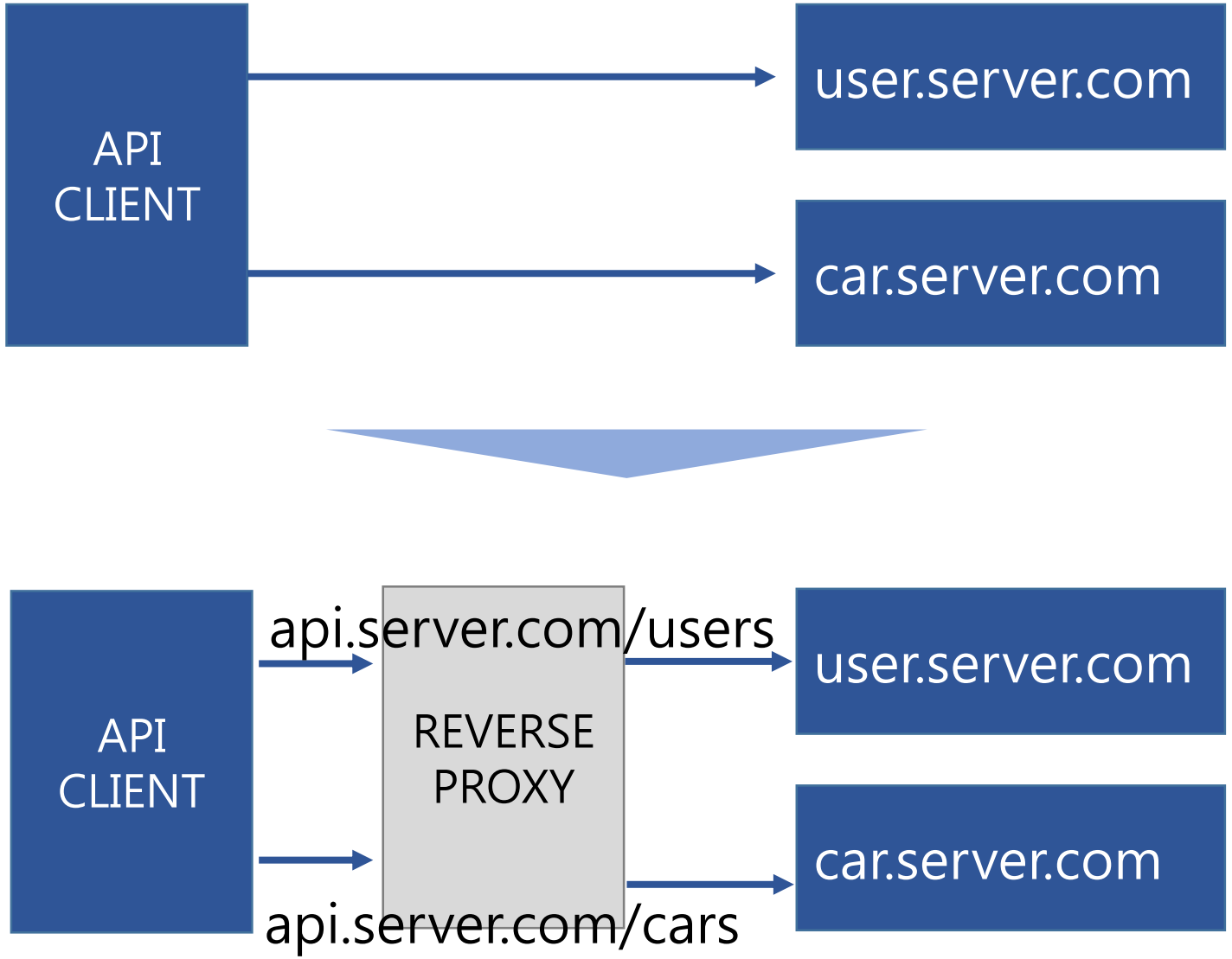
페이징에서 HATEOS로 이전,다음 페이지 링크 표현

```
HTTP GET users/terry
{
'id':'terry'
'links':[{
'rel':'friends',
'href':'http://xxx/users/terry/friends'
}]
}
```

사용자 정보에서 친구 정보에 대한 리소스를 HATEOS로 표현

단일 API URL & CORS

- API 서버를 분리해서 개발할 경우, 각 API 별 서버 URL이 다름
  - 개발자가 호출하는데 불편함
  - CORS 문제 발생
- REVERSE PROXY를 통한 라우팅으로 해결 가능



#3

API 보안

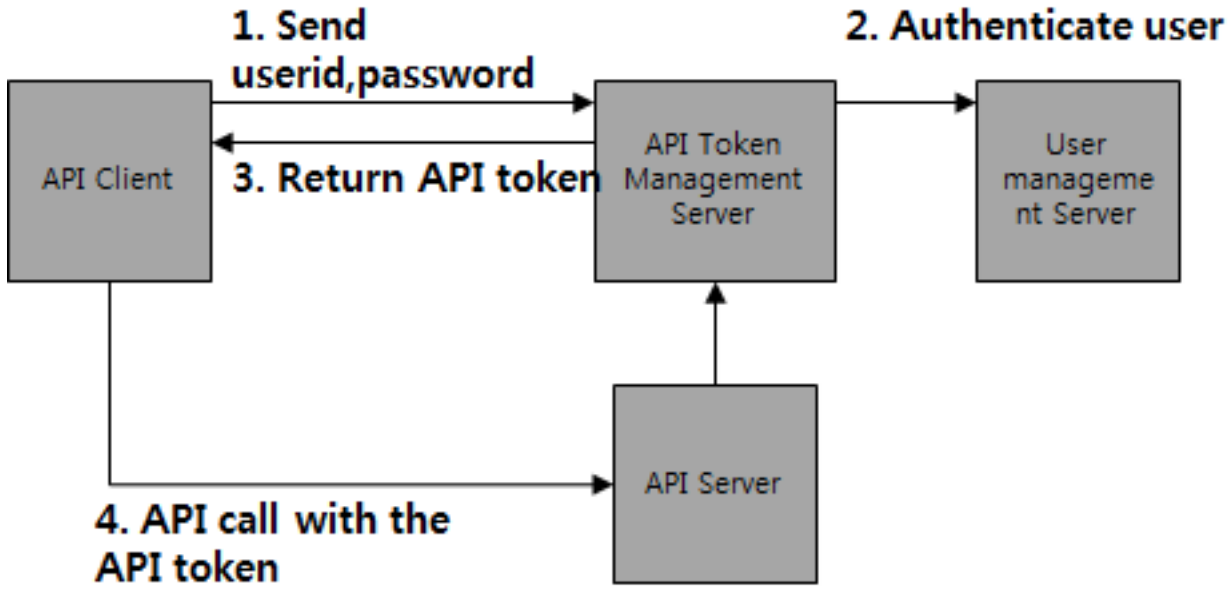
API 보안

- 인증 (Authentication)
- 인가 (Authorization)
- 네트워크 레벨 전송 암호화
- 메세지 무결성 보장
- 메시지 본문 암호화

http://bcho.tistory.com/955

API 인증 방식

- API 키 방식
  - 가장 기초적인 인증 방식.
  - API를 호출할때, API KEY를 보내는 방식 (모든 클라이언트가 같은 키를 공유)
  - 구현은 쉬우나 한번뚤리면 끝
- API 토큰 방식
  - 가장 널리 사용되는 방식
  - ID,PASSWORD를 넣으면, 일정 기간 유효한 API 토큰을 리턴하고, 매 호출마다 이 토큰을 사용



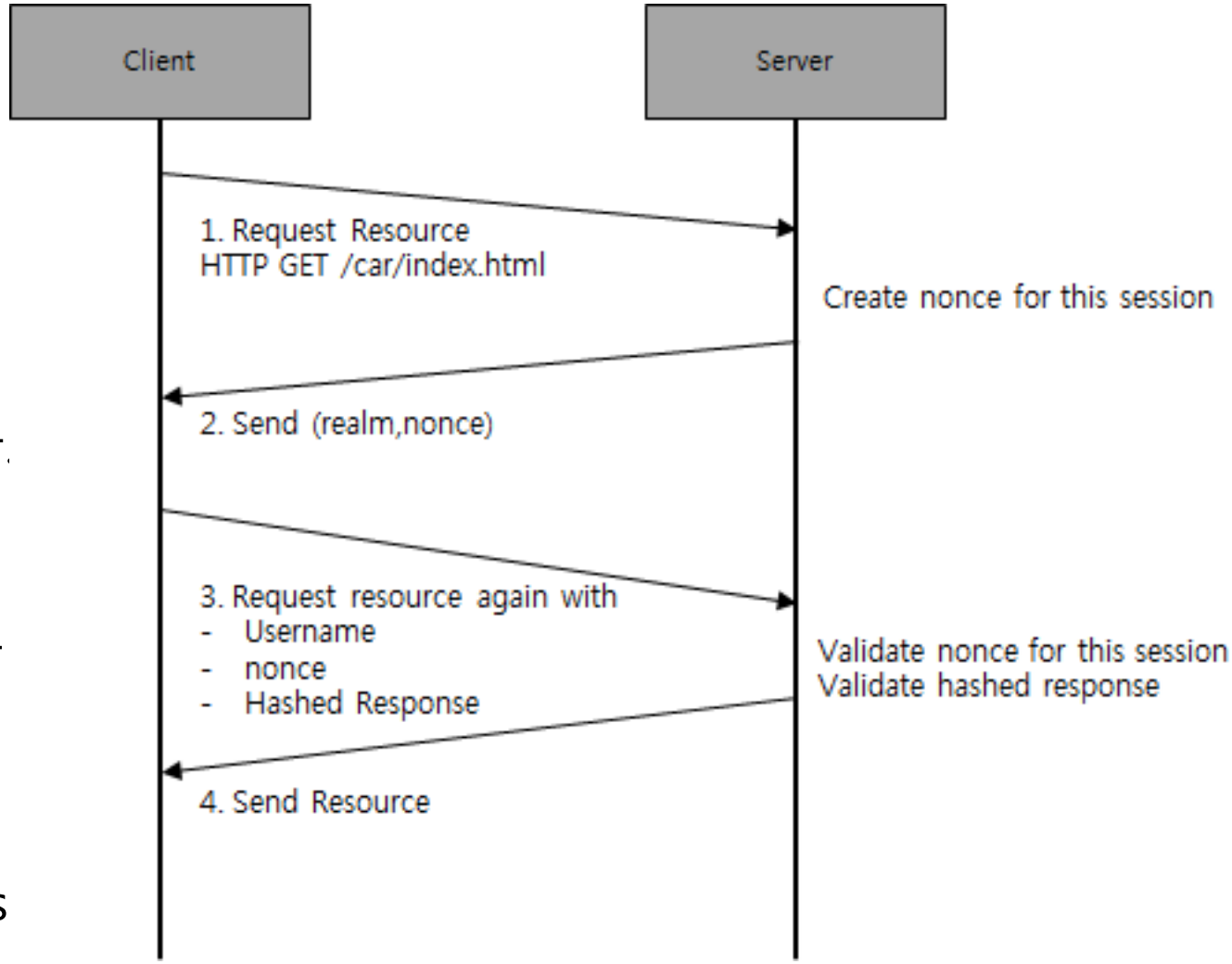
API 토큰 기반 API 인증 흐름

API 인증시 사용자 인증 방식

- HTTP BASIC AUTH
  - 가장 쉬운 방식으로 ID,PASSWORD를 직접 보낸다.
- DIGEST ACCESS AUTHENTICATION
  - 네트웍을 통해 패스워드를 보내지 않는다.
  - 클라이언트와 서버에서 공통적인 키를 이용해서 패스워드에 대한 해쉬를 생성하고 이를 비교하는 방식

    참고 :

    http://en.wikipedia.org/wiki/Digest_access_authentication



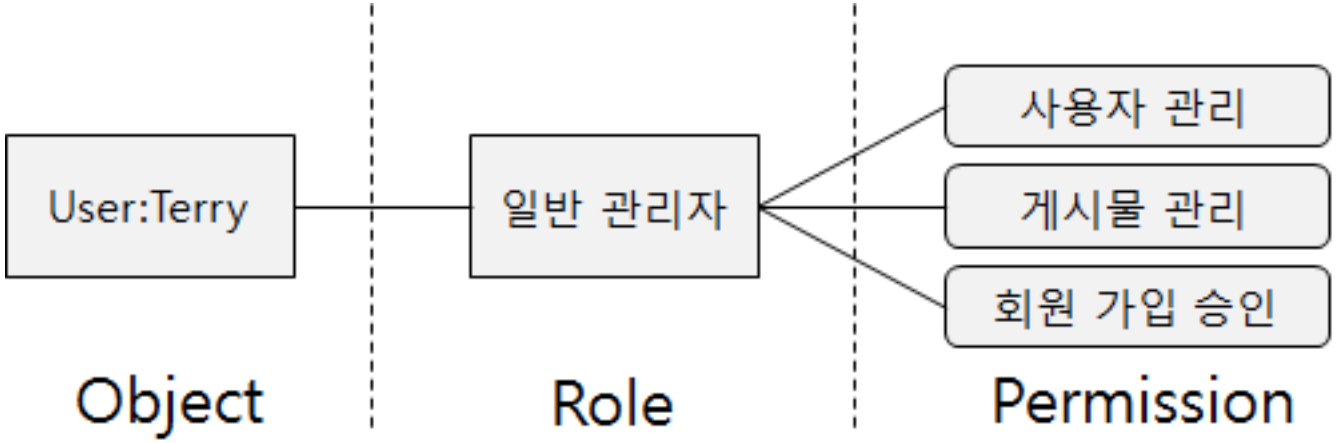
Nonce를 유지해야하기 때문에, 별도의 In memory 컴포넌트를 유지해야 하는 부담이 있음

API 인증시 사용자 인증 방식

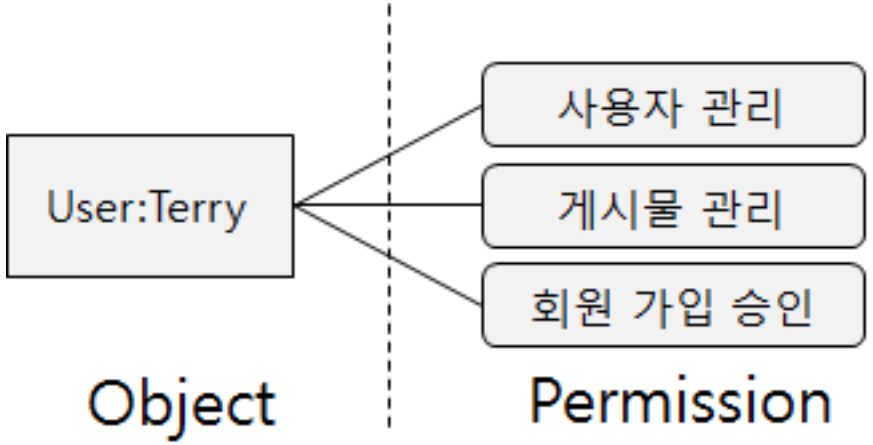
- 3자 인증 방식
  - 계정 정보를 내가 가지고 있는 게 아니라, 다른 서비스가 가지고 있는 경우
  - 페이스북 로그인, 구글 로그인등이 대표적인 사례
  - 보통 Oauth 2.0 을 이용해서 구현
  - 하나의 회사에 여러 서비스가 있을때 유용하게 사용할 수 있음
- 클라이언트 인증
  - 서비스가 여러 클라이언트에 의해 사용될때, 클라이언트별 접근 토큰을 발급하여 인증을 강화할 수 있음 (페이스북)
  - Clientid & secret (AppID, AppSecret)

API 호출 권한 처리

- 권한 처리 방식
  - RBAC (Role Based Authorization Control) : 사용자에 롤(사용자, 관리자)를 부여하여 통제
  - ACL (Access Control List) : 사용자별로 각각 세부 권한을 부여하는 방식



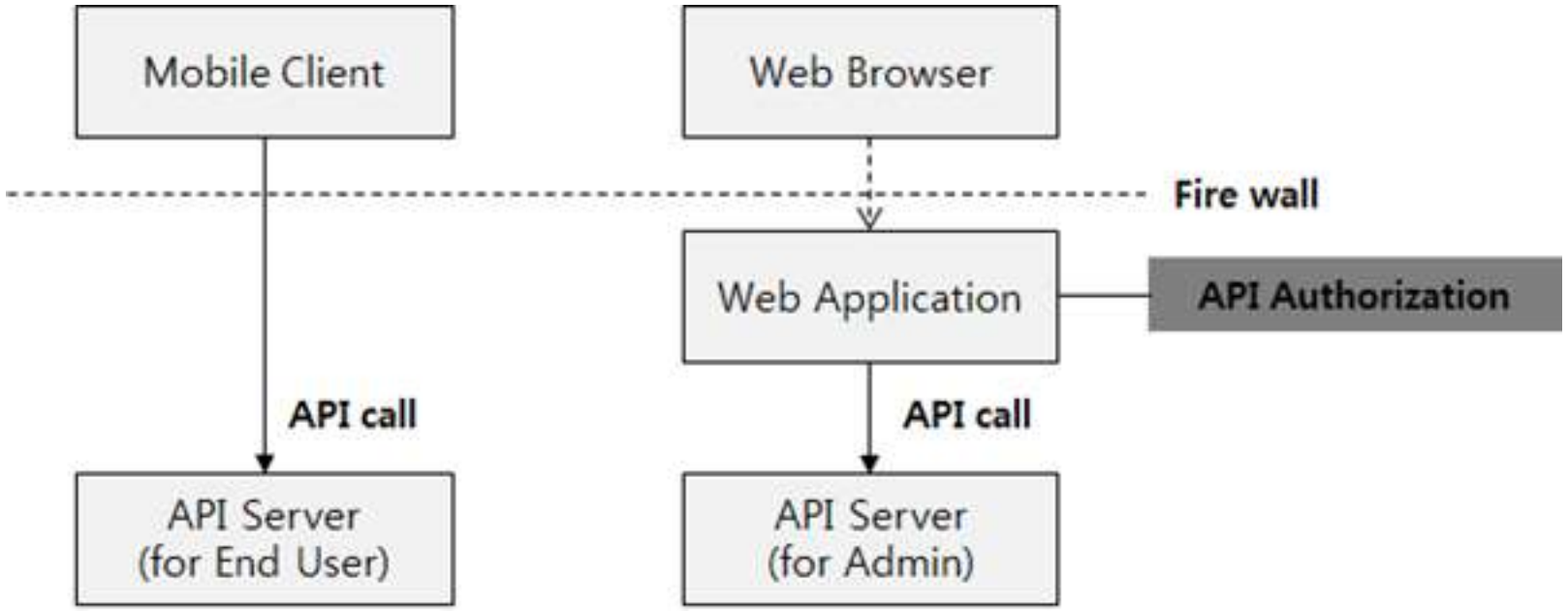
RBAC 방식의 권한 구조



ACL 방식의 권한 구조

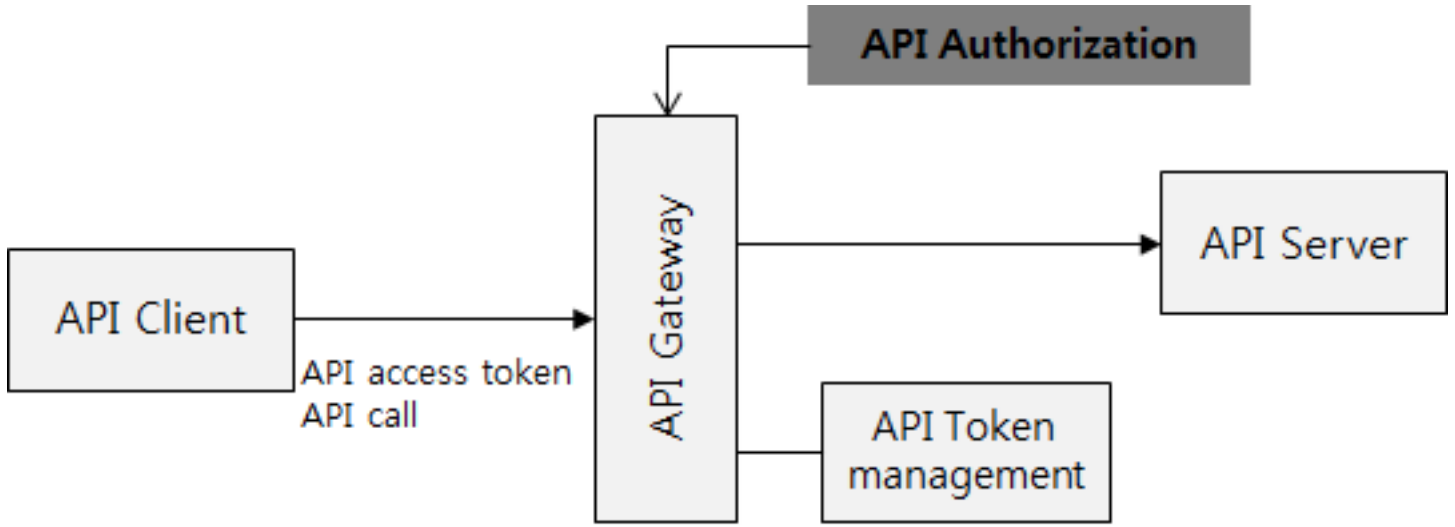
API 호출 권한 처리

- API 권한 처리를 하는 위치에 따라서 3가지 방식으로 구현이 가능함 (좀 어려움 ㅜㅜ)
- 클라이언트에서의 권한 처리
  - API 클라이언트에서 권한에 따라 적절한 API를 호출
  - 클라이언트가 신뢰할 수 있는 클라이언트만 가능 (내부 서버 또는 MVC 기반의 웹 애플리케이션등)



API 호출 권한 처리

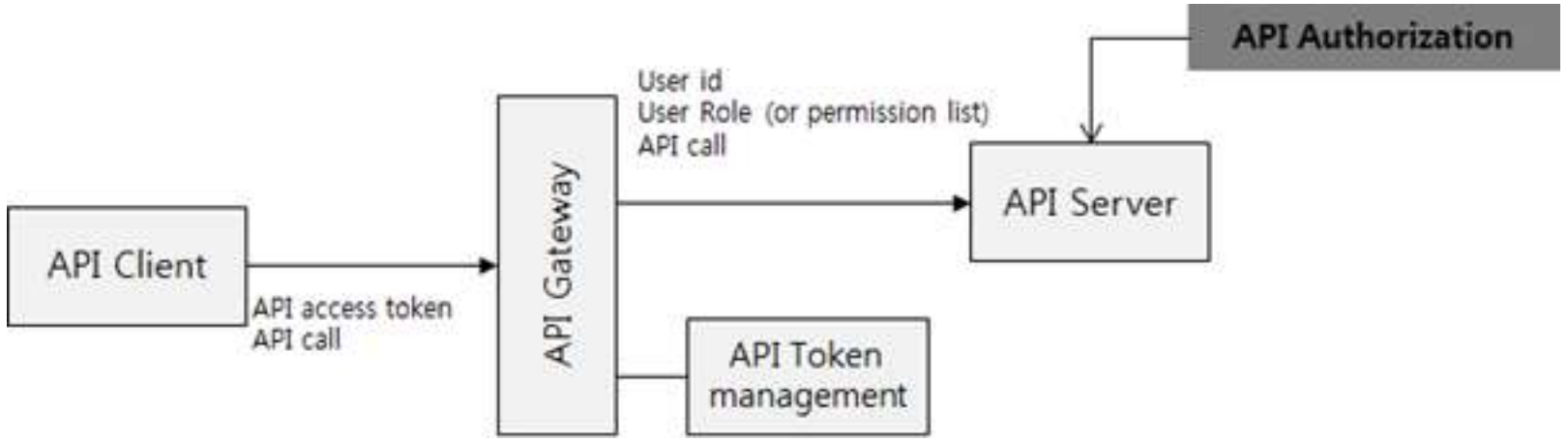
- 공통 필터에 의한 권한 처리 (API 게이트웨이, 서블릿 필터등)
  - 일반 사용자 GET /users/otheruser (x) 호출 안됨
  - 관리자 GET/users/otherusers (O) 관리자 권한으로 호출 가능
- 일반 사용자용 API와 관리자용 API가 잘 나눠져 있을 경우 효과적
- 권한 처리 로직이 복잡할 수 있고 설계가 어려움.
- 또한 권한 처리에 관련된 정보가 HTTP BODY에 들어가 있으면 처리가 어려움



API 호출 권한 처리

- 각 API 서버에서 권한 처리
  - 필터(게이트웨이)에 의한 권한 처리는 권한 정보가 HTTP BODY에 들어있으면 처리가 어려움
  - 공통 권한 정보를 HTTP HEADER에 넣어서, 개별 서버에서 처리하는 방식

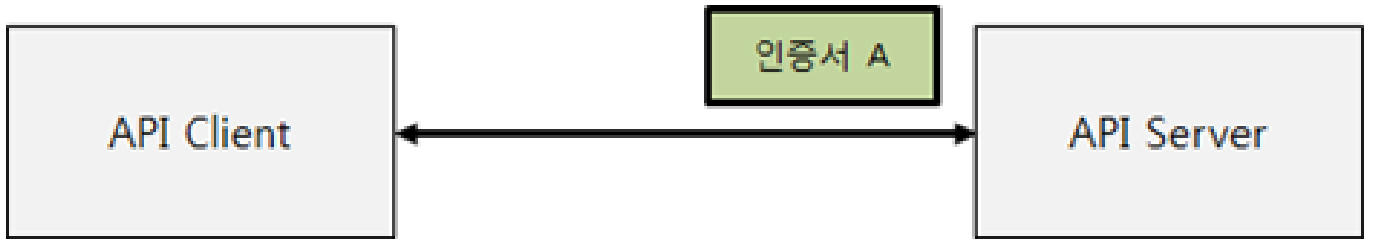
API 토큰을 게이트웨이에서 USER
ROLE, Permission등을 헤더에 추가



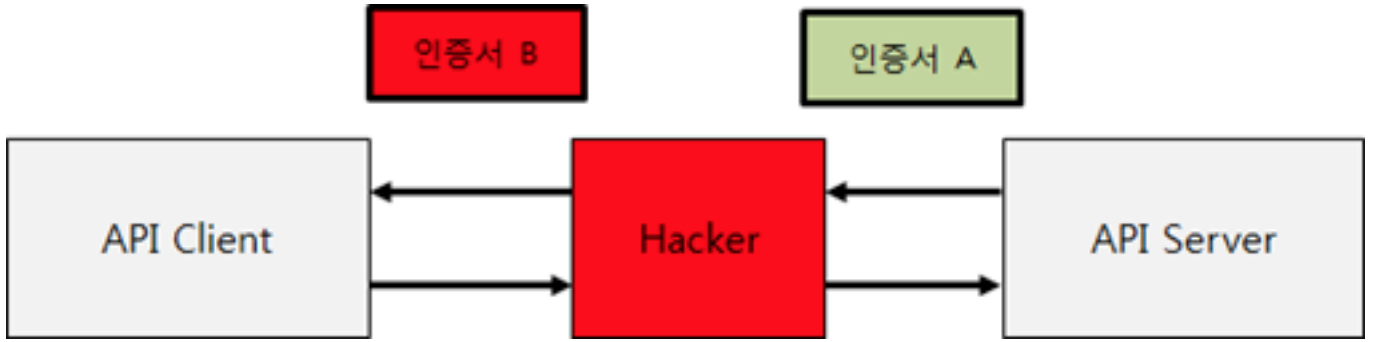
  - 필터에서 API토큰으로 ROLE, Permission등 인증 관련 정보를 찾아서 HTTP 헤더에 삽입하여 각 서버에서 이 정보를 기반으로 권한 처리
  - 또는 API 토큰 자체에 인증 관련 정보를 넣는 방식 (JWT 토큰)

API 네트워크 레벨 보안

- SSL (단방향, 양방향)
  - 단방향의 경우 Man in the middle attack 가능
  - 인증서의 발급자를 확인하는 로직을 추가하여 방어 가능



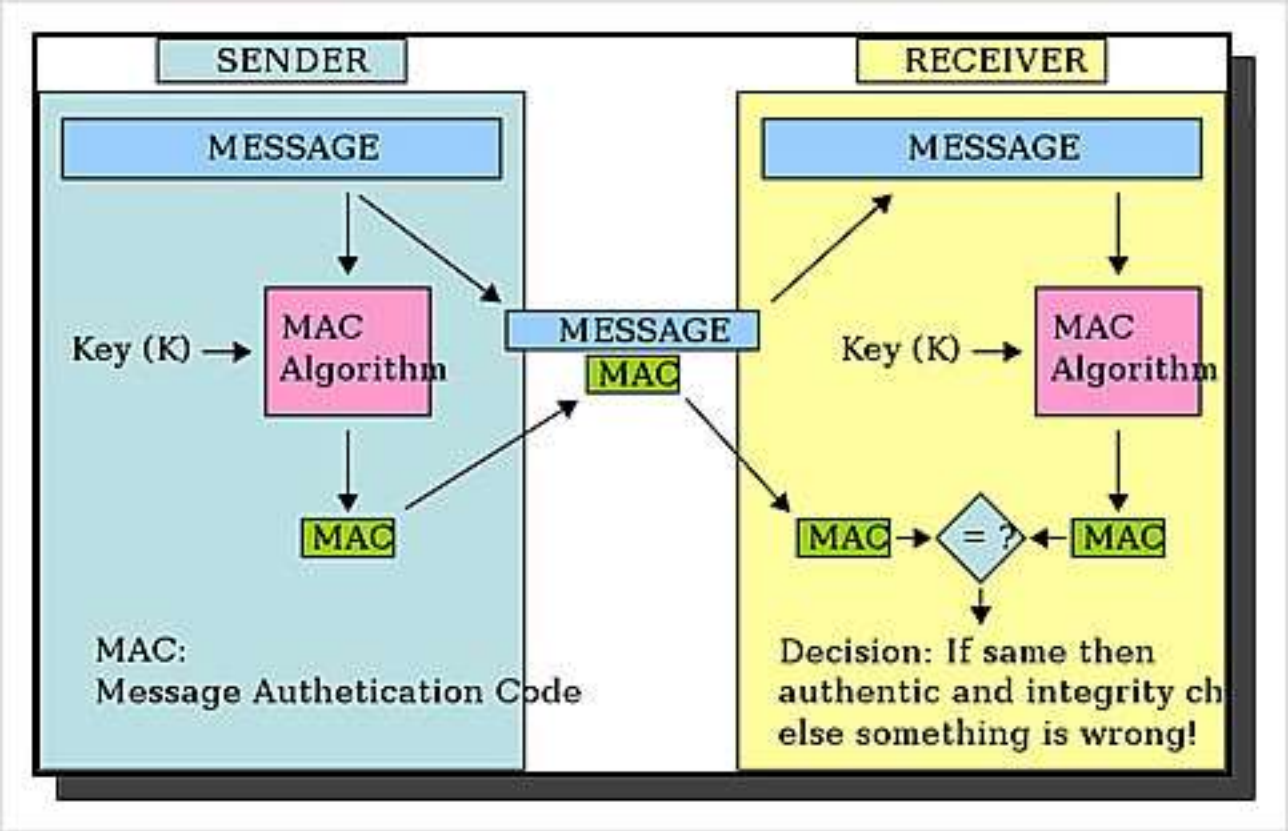
정상적인 SSL 호출



Man in the middle attack 을 이용해서 인증서를 바꿔 치는 방식

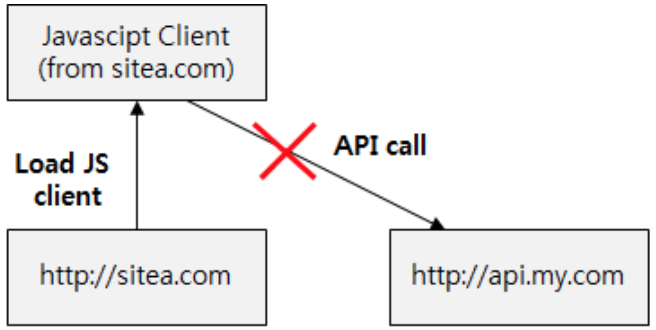
추가적인 보안 강화 방법

- 메세지 암호화
  - 대칭키 알고리즘을 이용하여 특정 필드를 암호화
- HMAC을 이용한 메세지 무결성 보장 (변조 방지)
  - 메세지 전송시, 메세지 끝에 본문의 해쉬를 이용한 시그니처를 생성하여 붙이는 방식으로, 본문이 변경되었는지를 확인할 수 있음.
  - 높은 보안을 필요로 할 때 사용

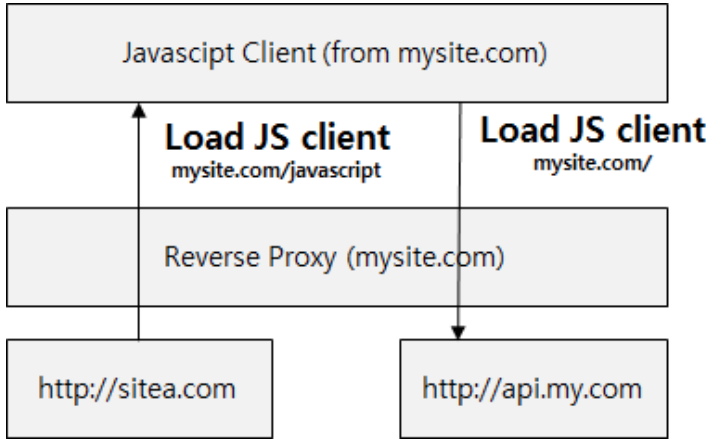


자바스크립트 클라이언트 지원

- 자바스크립트는 구조상 소스코드가 노출되기 때문에 추가적인 보안 조치를 하는 것이 좋음
- Same Origin Policy
  - 자바스크립트를 이용한 API호출은 자바스크립트를 다운 받은 URL에서만 가능
  - PROXY를 이용한 우회
  - Access-Control-Allow-Origin: * 로 전체 요청을 푸는 방식
  - HTTP OPTION 호출을 통한 Pre-flight



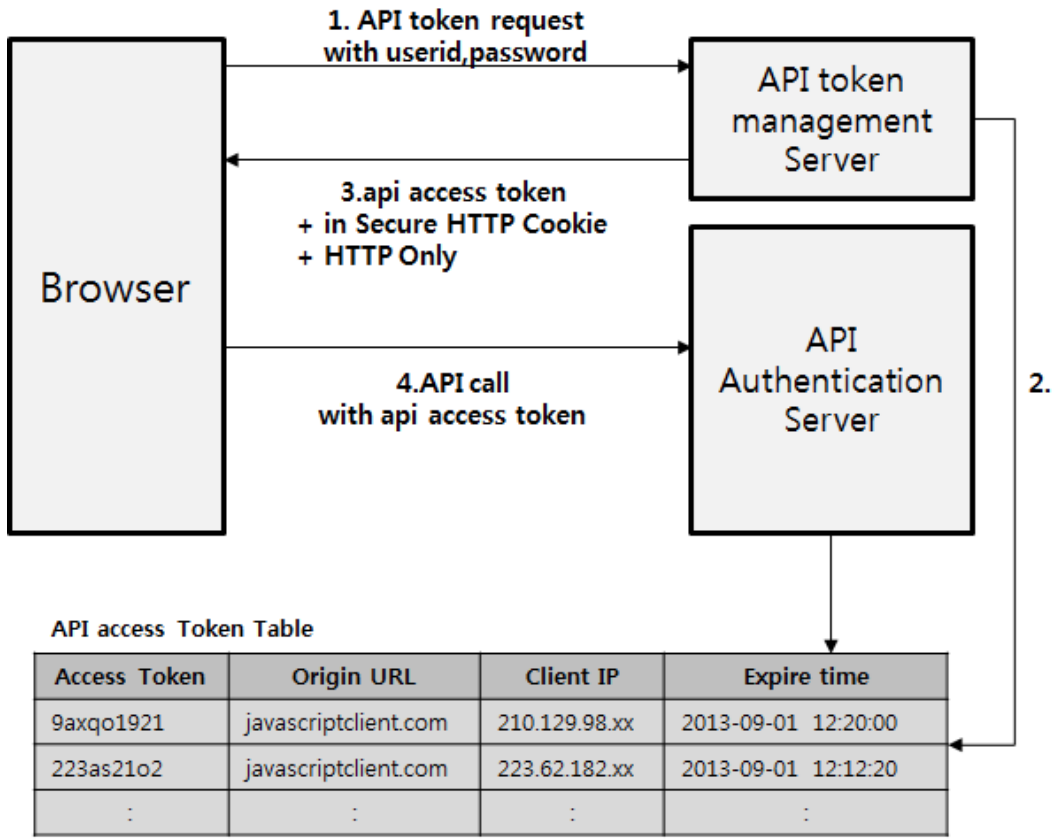
Same origin policy



PROXY를 이용한 우회

자바스크립트 클라이언트 지원

- API access token의 추가 보안 처리
  - 자바 스크립트 클라이언트에서는 access token 이 노출될 수 있음. 이에 대한 보강안
    - Secure Cookie : HTTPS를 통해서만 전송이 가능한 쿠키
    - HTTP ONLY 옵션 : 자바스크립트를 통해서 해당 쿠키는 접근이 불가. (브라우져에 의해서 자동으로 전송만됨)
    - 해당 세션에만 유효하도록 API 토큰을 통제



API access token을 특정 IP 주소에, 특정 시간에만 유효하게 하는 구조

#5

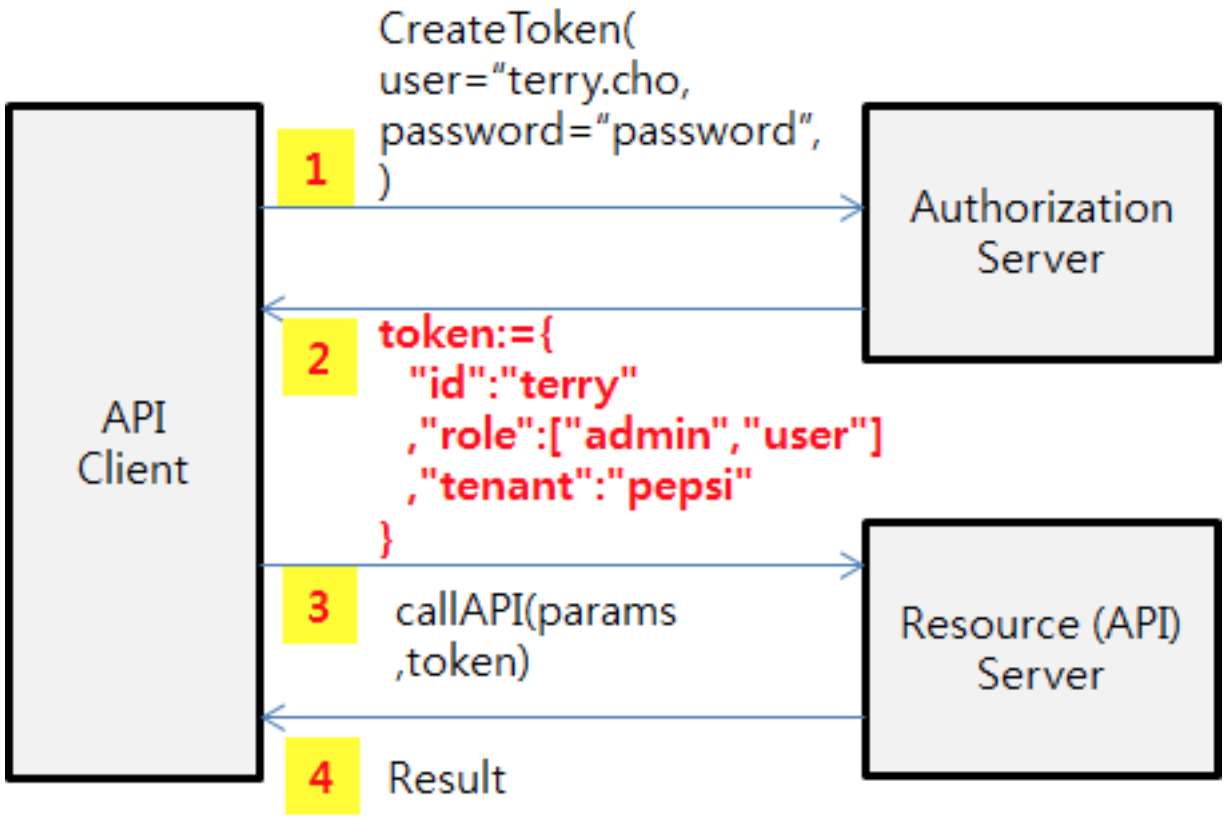
JWT 토큰

JWT 토큰

- JSON Web Token
- 클레임 기반 토큰
  - API access token 처럼 단순 키 문자열이 아니라, JSON 처럼 여러 정보를 담은 클레임 자체를 토큰으로 사용

```
{
  "id":"terry"
  ,"role":["admin","user"]
  ,"company":"pepsi"
}
```

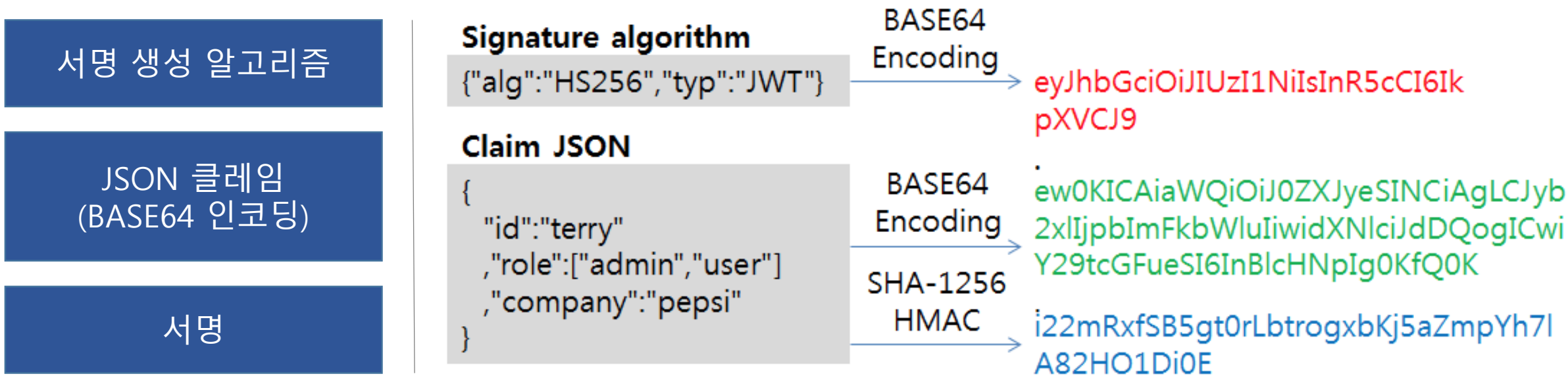
클레임 기반의 JSON 토큰



클레임 기반의 JSON 토큰
발급 및 인증 절차

JWT 토큰

- JSON 기반의 토큰의 문제점
  - 클라이언트단에서 변조가 가능
  - 변조를 방지 하는 방법 : JSON 토큰 뒤에 변조 방지를 위한 서명을 생성하여 붙임
- JWT 토큰 구조



JWT 토큰

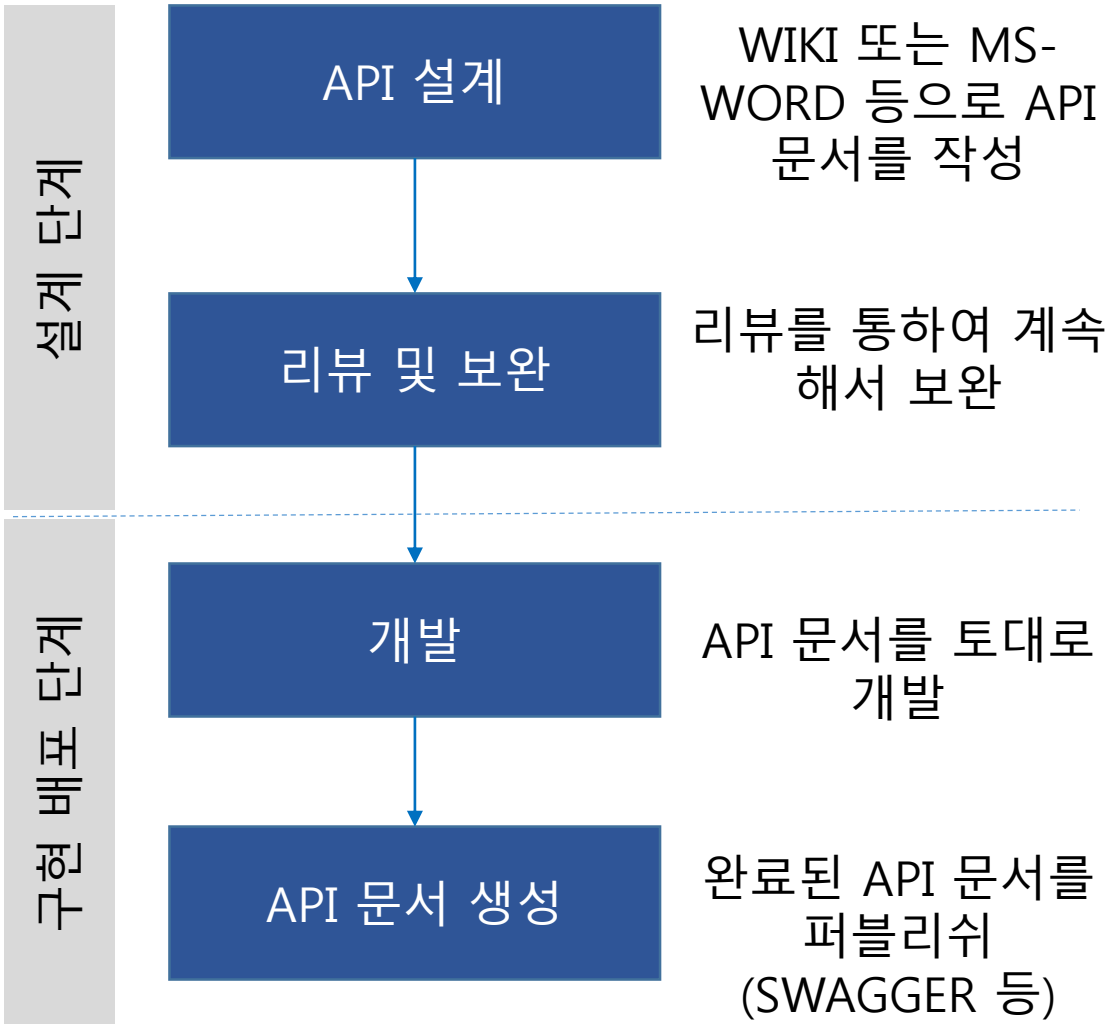
- 활용 방안
  - JSON 클레임안에, 상태 정보를 넣어서 서버쪽의 상태 정보를 다루는 부담을 덜 수 있음
  - 권한 정보를 넣음으로써, 권한 처리에 유리

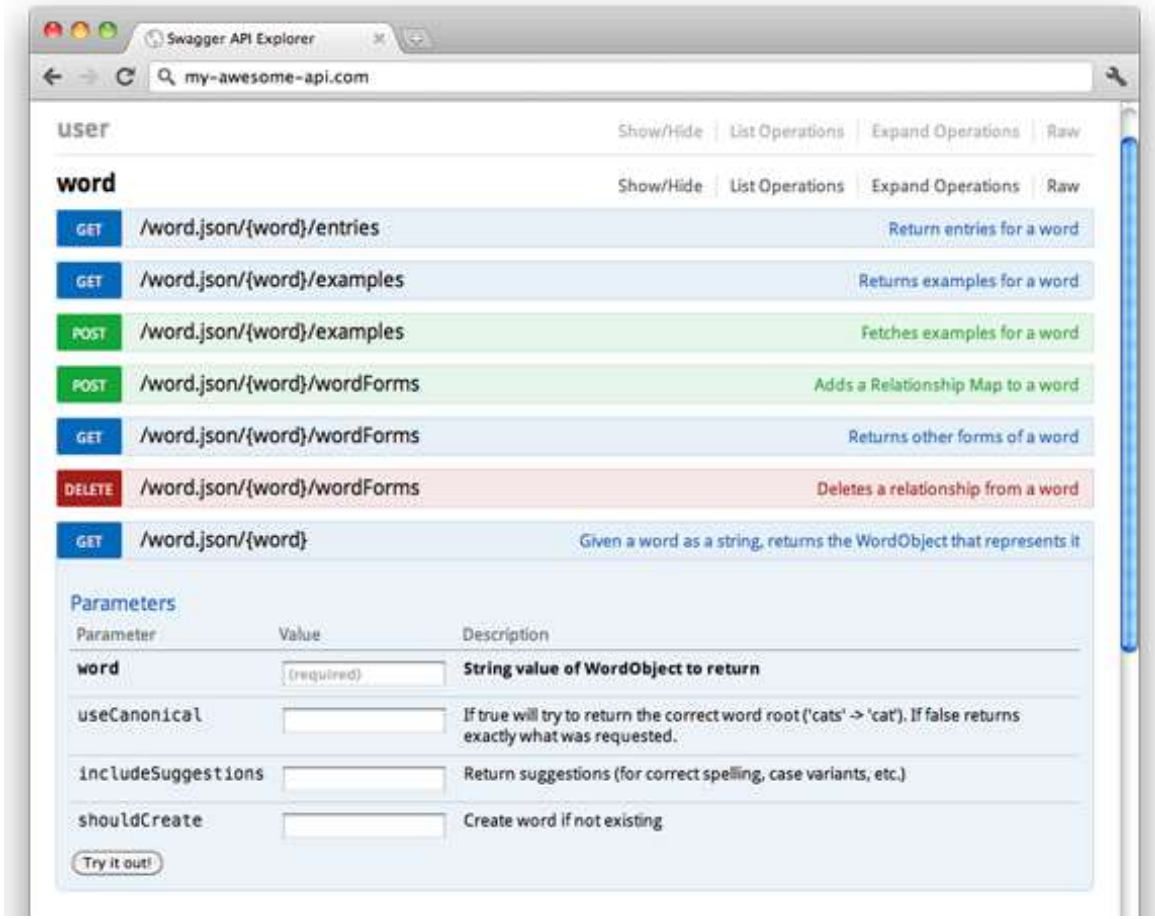
#5

API 설계 단계

API 문서화

- API 문서화는 중복이 발생하더라도 2단계(설계용,배포용)로 진행하는 것이 좋음 (리뷰)





SWAGGER를 이용한 API 문서

End of document