https://bcho.tistory.com/953 http://egloos.zum.com/tiger5net/v/5202221

REST API 의 이해와 설계

#1-개념 소개

1. 웹의 장점을 최대한 활용할 수 있는 **네트워크 기반의 아키텍쳐**
   A. REST 구조는 웹 본연의 특성과 장점을 최대화 하기 위해서 브라우저와 웹서버, 웹 어플리케이션, 캐쉬등이 어떻게 구성되고 동작해야 하는 지를 기술하고 있다.
2. 분산된 하이퍼미디어 시스템을 위한 아키텍쳐 스타일 ?
3. Rob Fielding 발표
   A. HTTP 스펙을 만든 사람 중 한 명
   B. REST 는 매우 체계적으로 잘 정리된 구조 라고 품평
4. Representational safe transfer (REST) 의 의미
   A. 브라우저와 같은 웹 클라이언트는 URL link 와 같은 resource identifer를 통해 resource 에 대해 어떤 요청을 하고 그 결과를 받게 된다. 이러한 요청 결과는 브라우저 화면에 새로운 내용이 디스플레이 되는 등의 방식으로 표현된다. 즉, 이 과정을 통해서 브라우저 표현 상태(Representational State)의 변경(Transfer)을 유발하게된다. REST는 이러한 특성을 표현한 용어이다.

REST의 기본 요소

이름이 Terry인 사용자를 생성한다

1. Resource 사용자, http://myweb/users URL로 표현
   A. 웹이 제공하는 모든 정보/데이터
   B. 파일, 이미지, '나의 구매 책 리스트'
   C. resource 에 대해서는 그에 대한 링크인 resource identifier가 있다.
2. Method 생성한다, HTTP POST 메서드
3. Message 이름이 Terry인 사용자, 생성하고자 하는 디테일한 내용은 JSON 문서를 이용하여 표현됨.

4. HTTP POST, http://myweb/users[resource]/

{

“users”: {

“name”: “terry”

}

}

REST의 리소스

리소스 지향 아키텍처(Resource Oriented Architecture, ROA)

REST한 아키텍쳐를 구현하기 위한 한가지 방법.

1. 웹 서비스에서 제공할 데이터를 특정한다.
2. 데이터를 리소스로 나눈다.
3. 리소스에 URI로 이름을 부여한다.
4. 클라이언트에 제공할 리소스의 표현을 설계한다.
5. 링크와 폼을 이용해 리소스와 리소스를 연결한다.
6. 이벤트의 표준적인 코스를 검토한다.
7. 에러에 대해 검토한다.

출처: https://starplatina.tistory.com/entry/리소스-지향-아키텍처-설계-어프로치 [Clean Code that Works.]

REST는 리소스 지향 아키텍쳐 스타일이라는 정의 답게

모든 것을 리소스 즉 명사로 표현을 하며, 각 세부 리소스에는 id를 붙인다.

즉 사용자라는 리소스 타입을 http://myweb/users라고 정의했다면,

terry라는 id를 갖는 리소스는 http://myweb/users/terry 라는 형태로 정의한다.

다음은 http://myweb/users 라는 리소스를 이름은 terry, 주소는 seoul 이라는 내용(메시지)로 HTTP Post를 이용해서 생성하는 정의이다.

HTTP Post, http://myweb/users/

{

"name":"terry",

"address":"seoul"

}

조회

다음은 생성된 리소스 중에서 http://myweb/users 라는 사용자 리소스중에, id가 terry 인 사용자 정보를 조회해오는 방식이다. 조회이기 때문에, HTTP Get을 사용한다.

HTTP Get, http://myweb/users/terry

업데이트

다음은 http://myweb/users 라는 사용자 리소스중에, id가 terry 인 사용자 정보에 대해서, 주소를 “suwon”으로 수정하는 방식이다. 수정은 HTTP 메서드 중에 PUT을 사용한다.

HTTP PUT, http://myweb/users/terry

```
{
    "name":"terry",
    "address":"suwon"
}
```

HTTP 메서드

1. REST의 행위에 대한 method는, HTTP 메서드를 그대로 사용한다. 그 중 CRUD만 사용

메서드	의미	Idempotent
POST	Create	N
GET	SELECT	Y
PUT	Update	Y
DELETE	Delete	Y

2. Idempotent 여러 번 수행을 해도 결과가 같은 경우
   A. GET의 경우 게시물의 조회수 카운트를 늘려준다던가 하는 기능을 같이 수행했을 때 Idempotent 하지 않은 메서드로 정의
      i. 조회하다가 실패하였을 때는 올라간 조회수를 다시 -1로 빼줘야 한다
      ii. Idempotent 하지 않은 메서드에 대해서는 트렌젝션에 대한 처리가 별다른 주의가 필요하다
   B. 각 개별 API를 상태 없이 수행하는 REST의 특성

      해당 REST API를 다른 API와 함께 호출하다가 실패하였을 경우, 트렌젝션 복구를 위해서 다시 실행해야 한다

      1. Idempotent 하지 않은 메서드
         A. 기존 상태를 저장했다가 다시 원복
      2. Idempotent 한 메서드

A. 반복적으로 다시 메서드를 수행해주면 된다.

REST 구조의 특징

Uniform-Layerd-Client-Cache-Stateless-Server

Client, Server 구조

resource 에 대한 요청/제공을 위한 구조

1. resource에 대한 interaction 은 stateless 해야 한다.
   브라우저나, 서버가 세션 상태등을 저장하고 있어서는 안되며, 필요하다면 이러한 상태 역시 resource로 표현되고 접근 가능해야 한다.

2. resource에 대한 요청 결과는 cache 할 수 있다.
   client는 cacheable이라고 표시된 요청 결과들을 나중을 위해 저장해 둘 수 있다.

3. client는 서버에 resource를 요청하기 위해서 uniform interface를 사용한다.
   각 application 마다 어떤 동작(action)을 위해 별도로 정의한 operation을 사용하지 않는다"는 정도로 이해할 수 있다. 예를 들어 SOAP 방식에서는 각 application에서 정의한 XML 스키마 속에 데이터와 opertion 을 함께 지정하는 것이 일반적인데 이 것은 RESTful 하지 않다고 볼 수 있다.

4. Rails 1.2 에서 도입된 ActiveResource 의 경우 get, post, put, delete 와 같은 웹 표준 opertion만을 이용하는데 이것이 RESTful 에 더 적합한 방식이다. Client Server 구조는 여러 단계의 Layered 형태로 구성될 수 있다

5. client 가 server에 직접 연결되지 않고 중간에 하나 이상의 노드가 존재한다. 이들 **중간 노드는 client와 server의 역할을 동시에** 수행한다.

   위에 설명한 특징들 중에서 1, 3, 5번의 경우에는 기존의 웹에서 이미 적용되고 있거나 혹은 적용에 있어 별 이슈가 없는 내용들이다. 그런데 2, 4번의 경우에는 얘기가 좀 달라진다.

   오늘날 대부분의 웹 어플리케이션들은 사용자 인증 또는 상태 정보를 유지하기 위해 **쿠키 또는 HttpSession 등을 사용하고 있는데 이것은 REST의 원칙에 명백히 위배되는 방식**이다. 또, 웹 서버상의 데이터를 조회하는 것은 물론이고 변경이나 삭제 심지어는 생성을 위해서도 GET method 를 사용하는 경우도 매우 많은 것이 사실이다. (서버 사이드의 변경을 유발하는 요청에 대해서는 POST를 사용하는 것이 바람직하며, ActiveResource 에서는 생성을 위해서는 PUT, 삭제를 위해서는 DELETE를 사용한다.)

   하지만 현실적으로 여러 사용자에 대한 동적인 정보를 다루는 웹사이트를 쿠키나 세션없이 개발하는 것은 쉬운 일이 아니다. 또 PUT과 DELETE 같은 method는 HTTP 스펙에는 존재하지만 실제 이를 지원하는 브라우저는 많지 않다. (참고로 XMLHttpRequest 에서는 PUT과 DELETE를 지원한다. ? Using REST with Ajax)

REST 의 현황

1. RESTful 이란

   A. REST의 개념에 충실하게 구현된 웹 application과 구조를 말하며,

2. RESTafarian 이란

   A. REST 신봉자를 일컫는 말이다. (REST 토론 그룹)

   그러나 모든 웹 application들이 Stateless 와 Uniform Interface 라는 특성을 충분히 따르기에는 현실적인 어려움이 많으므로 당분간은 순수한 REST가 일반적인 웹 개발의 패러다임으로 자리잡지는 못할 것으로 보인다.

   그런데 최근 관심이 높아지고 있는 분야들인 Open API, Ajax, Rails 와 같은 곳에서 REST라는 용어를 자주 접하게 되는 것은 반가운 현상이다. 앞에서도 언급했지만, DHH는 RESTful 한 ActiveResource를 통해 새로운 가능성을 제시하고 있다. 또 Amazone, eBay, Yahoo 와 같은 주요 웹 업체들은 대부분 REST 방식의 OPEN API를 제공하고 있으며 (Amazone 이 제공하는 SOAP, REST 두 가지 방식의 API 중에서 REST API의 사용율이 85%라는 정보도 소개 된 바 있다) 최근에 Google 이 기존의 SOAP 방식 API의 지원을 중단 하면서 Ajax를 이용한 API를 새로 제공하기 시작한 것도 눈여겨 볼만한 대목이다.

REST의 특성

1. 유니폼 인터페이스(Uniform Interface)

REST는 HTTP 표준에만 따른 다면, 어떠한 기술이라던지 사용이 가능한 인터페이스 스타일

HTTP + JSON으로 REST API를 정의했다면, 안드로이드 플랫폼이건, iOS 플랫폼이건, 또는 C나 Java/Python이건 특정 언어나 기술에 종속 받지 않고 HTTP와 JSON을 사용할 수 있는 모든 플랫폼에 사용이 **가능한 느슨한 결합(Loosely coupling)** 형태의 구조이다.

※ 흔히들 근래에 REST를 이야기 하면, HTTP + JSON을 쉽게 떠올리는데, JSON은 하나의 옵션일뿐, 메시지 포맷을 꼭 JSON으로 적용해야할 필요는 없다. 자바스크립트가 유행하기전에만 해도 XML 형태를 많이 사용했으며, 근래에 들어서 사용의 편리성 때문에 JSON을 많이 사용하고 있지만, XML을 사용할 경우, XPath,XSL등 다양한 XML 프레임웍을 사용할 수 있을뿐만 아니라 메시지 구조를 명시적으로 정의할 수 있는 XML Scheme나 DTD등을 사용할 수 있기 때문에, 복잡도는 올라가더라도, 메시지 정의의 명확성을 더할 수 있다.

2. 무상태성/스테이트리스(Stateless)

REST는 REpresentational State Transfer 의 약어로 Stateless (상태 유지하지 않음)이 특징 중의 하나이다.

- 상태가 있다 없다는 의미

사용자나 클라이언트의 컨택스트를 서버쪽에 유지하지 않는다

쉽게 표현하면 HTTP Session과 같은 컨텍스트 저장소에 상태 정보를 저장하지 않는 형태를 의미

상태 정보를 저장하지 않으면 각 API 서버는 들어오는 요청만을 들어오는 메시지로만 처리하면 되며, 세션과 같은 컨텍스트 정보를 신경쓸 필요가 없기 때문에 구현이 단순해진다.

3. 캐슁 가능(Cacheable)

HTTP 프로토콜 기반의 로드 밸런서나 SSL은 물론이고, **HTTP가 가진 가장 강력한 특징중의 하나인 캐슁 기능을 적용할 수 있다.**

일반적인 서비스 시스템에서 60%에서 많게는 80%가량의 트렌젝션이 Select와 같은 조회성 트렌젝션인 것을 감안하면, HTTP의 리소스들을 웹캐쉬 서버등에 캐슁하는 것은 용량이나 성능 면에서 많은 장점을 가지고 올 수 있다.

구현은 HTTP 프로토콜 표준에서 사용하는 "**Last-Modified**" 태그나 **E-Tag**를 이용하면 캐슁을 구현할 수 있다.

아래와 같이 Client가 HTTP GET을 "Last-Modified" 값과 함께 보냈을 때, 컨텐츠가 변화가 없으면 REST 컴포넌트는 "304 Not Modified"를 리턴하면 Client는 자체 캐쉬에 저장된 값을 사용하게 된다.

네트웍 응답시간 뿐만 아니라, REST 컴포넌트가 위치한 서버에 트렌젝션을 발생시키지 않기 때문에, 전체 응답시간과 성능 그리고 서버의 자원 사용률을 비약적으로 향상 시킬 수 있다.

4. 자체 표현 구조(Self-descriptiveness)

REST API 자체가 매우 쉬워서 API 메시지 자체만 보고도 API를 이해할 수 있는 Self-descriptiveness 구조를 갖는 다는 것이다.

리소스와 메서드를 이용해서 어떤 메서드에 무슨 행위를 하는지를 알 수 있으며, 또한 메시지 포맷 역시 JSON을 이용해서 직관적으로 이해가 가능한 구조이다.

대부분의 REST 기반의 OPEN API들이 API 문서를 별도로 제공하고 있지만, 디자인 사상은 최소한의 문서의 도움만으로도 API 자체를 이해할 수 있어야 한다.

5. 클라이언트 서버 구조 (Client-Server 구조)

근래에 들면서 재 정립되고 있는 특징 중의 하나는 REST가 클라이언트 서버 구조라는 것이다.

REST 서버는 API를 제공하고, 제공된 API를 이용해서 비즈니스 로직 처리 및 저장을 책임진다.

클라이언트의 경우 사용자 인증이나 컨택스트(세션,로그인 정보)등을 직접 관리하고 책임 지는 구조로 역할이 나뉘어 지고 있다. 이렇게 역할이 각각 확실하게 구분되면서, 개발 관점에서 클라이언트와 서버에서 개발해야 할 내용들이 명확하게 되고 서로의 개발에 있어서 의존성이 줄어들게 된다.

6. 계층형 구조 (Layered System)

계층형 아키텍쳐 구조 역시 근래에 들어서 주목받기 시작하는 구조인데, 클라이언트 입장에서는 REST API 서버만 호출한다.

그러나 서버는 다중 계층으로 구성될 수 있다. 순수 비즈니스 로직을 수행하는 API 서버와 그 앞단에 사용자 인증(Authentication), 암호화 (SSL), 로드밸런싱등을 하는 계층을 추가해서 구조상의 유연성을 둘 수 있는데, 이는 근래에 들어서 앞에서 언급한 마이크로 서비스 아키텍쳐의 api gateway나, 간단한 기능의 경우에는 HA Proxy나 Apache와 같은 Reverse Proxy를 이용해서 구현하는 경우가 많다.

REST 안티 패턴

1. GET/POST를 이용한 터널링

A. GET을 이용한 터널링

i. http://myweb/users?method=update&id=terry

ii. 메서드의 실제 동작은 리소스를 업데이트 하는 내용인데, HTTP PUT을 사용하지 않고, GET에 쿼리 패러미터로 method=update라고 넘겨서, 이 메서드가 수정 메세드임을 명시했다.

iii. HTTP 메서드 사상을 따르지 않았기 때문에, REST라고 부를 수도 없고, 또한 웹 캐쉬 인프라등도 사용이 불가능하다.

B. POST를 이용한 터널링

i. Insert(Create)성 오퍼러이션이 아닌데도 불구하고, JSON 바디에 오퍼레이션 명을 넘기는 형태인데 예를 들어 특정 사용자 정보를 가지고 오는 API를 아래와 같이 POST를 이용해서 만든 경우이다.

ii. HTTP POST, http://myweb/users/

```
{
  "getuser":{
```

```
        "id":"terry",
      }
  }
```

2. Self-descriptiveness 속성을 사용하지 않음

자기 서술성을 깨먹는 가장 대표적인 사례가 앞서 언급한 GET이나 POST를 이용한 터널링을 이용한 구조가 된다.

3. HTTP Response code를 사용하지 않음

Http Response code를 충실하게 따르지 않고, 성공은 200, 실패는 500 과 같이 1~2개의 HTTP response code만 사용하는 경우이다.

심한 경우에는 에러도 HTTP Response code 200으로 정의한후 별도의 에러 메시지를 200 response code와 함께 보내는 경우인데, 이는 REST 디자인 사상에도 어긋남은 물론이고 자기 서술성에도 어긋난다.

REST API 디자인 가이드

- REST URI는 심플&직관
- URI에 리소스명은 동사보다는 명사를 사용한다.
- REST API는 리소스에 대해서 행동을 정의하는 형태를 사용한다.

  POST /dogs

  /dogs라는 리소스를 생성하라는 의미

  URL은 HTTP Method에 의해 CRUD (생성,읽기,수정,삭제)의 대상이 되는 개체(명사)라야 한다.

  HTTP Get : /dogs

  HTTP Post : /dogs/{puppy}/owner/{terry}
- 복수형 명사(/dogs)를 사용하는 것이 의미상 표현하기가 더 좋다.

	POST	GET	PUT	DELETE
	create	read	update	delete
/dogs	새로운 dogs 등록	dogs 목록을 리턴	Bulk로 여러 dogs 정보를 업데이트	모든 dogs 정보를 삭제
/dogs/baduk	에러	baduk 이라는 이름의 dogs 정보를 리턴	baduk 이라는 이름의 dogs 정보를 업데이트	baduk 이라는 이름의 dogs 정보를 삭제

리소스간의 관계를 표현하는 방법

- 사용자가 소유하고 있는 디바이스 목록 / 사용자-디바이스
- 사용자가 가지고 있는 강아지들 / 사용자-강아지들

  각각의 리소스간의 관계를 표현하는 방법은 여러가지가 있다.

사용자가 가지고 있는 핸드폰 디바이스 목록

Option 1. 서브 리소스로 표현하는 방법

- **/"리소스명"/"리소스 id"/"관계가 있는 다른 리소스명" 형태**
- **소유 "has"의 관계를** 묵시적으로 표현할 때 좋다

HTTP Get : /users/{userid}/devices

예) /users/terry/devices

과 같이 /terry라는 사용자가 가지고 있는 디바이스 목록을 리턴하는 방법이 있고

Option 2. 서브 리소스에 관계를 명시하는 방법

- **관계의 명이 복잡하다면 관계명을 명시적으로 표현**
- 관계의 명이 애매하거나 구체적인 표현이 필요할 때

사용자가 "좋아하는" 디바이스 목록을 표현해보면

HTTP Get : /users/{userid}/likes/devices

예) /users/terry/likes/devices

는 terry라는 사용자가 좋아하는 디바이스 목록을 리턴하는 방식이다.

에러처리

에러처리의 기본은 **HTTP Response Code**를 사용한 후, **Response body**에 **error detail을 서술**

구글의 gdata 서비스의 경우 10개

200 201 304 400 401 403 404 409 410 500

넷플릭스의 경우 9개

200 201 304 400 401 403 404 412 500

Digg의 경우 8개의 Response Code를 사용한다.

200 400 401 403 404 410 500 503

여러 개의 response code를 사용

1. 명시적이긴 하지만, 코드 관리가 어렵기 때문에 몇 가지 response code만을 권장한다.

- 200 성공
- 400 Bad Request - field validation 실패시
- 401 Unauthorized - API 인증, 인가 실패
- 404 Not found ? 해당 리소스가 없음
- 500 Internal Server Error - 서버 에러

추가적인 HTTP response code에 대한 사용이 필요하면 http response code 정의
http://en.wikipedia.org/wiki/Http_error_codes 문서를 참고하기 바란다.

2. **에러 내용에 대한 디테일 내용을 http body에 정의해서, 상세한 에러의 원인을 전달하는 것이 디버깅에 유리**하다.

Twillo의 Error Message 형식의 경우

HTTP Status Code : 401

{"**status":"401","message":"Authenticate**","**code":200003**,"**more info":"http://www.twillo.com/docs/errors/20003"}**

3. **에러 코드 번호와 해당 에러 코드 번호에 대한 Error dictionary link**를 제공한다.

비단 API 뿐 아니라, 잘 정의된 소프트웨어 제품의 경우에는 **별도의 Error 번호 에 대한 Dictionary 를 제공하는데**, Oracle의 WebLogic의 경우에도
http://docs.oracle.com/cd/E24329_01/doc.1211/e26117/chapter_bea_messages.htm#sthref7 와 같이 Error 번호, 이에 대한 자세한 설명과, 조치 방법등을 설명한다. 이는 개발자나 Trouble Shooting하는 사람에게 많은 정보를 제공해서, 조금 더 디버깅을 손쉽게 한다. (가급적이면 Error Code 번호를 제공하는 것이 좋다.)

다음으로 에러 발생시에, 선택적으로 에러에 대한 스택 정보를 포함 시킬 수 있다.

에러메세지에서 Error Stack 정보를 출력하는 것은 대단히 위험한 일이다. 내부적인 코드 구조와 프레임웍 구조를 외부에 노출함으로써, 해커들에게, 해킹을 할 수 있는 정보를 제공하기 때문이다. **일반적인 서비스 구조에서는 아래와 같은 에러 스택정보를 API 에러 메세지에 포함 시키지 않는 것이 바람직 하다.**

```
log4j:ERROR setFile(null,true) call failed.
java.io.FileNotFoundException: stacktrace.log (Permission denied)
at java.io.FileOutputStream.openAppend(Native Method)
...
```

4. **내부 개발중이거나 디버깅 시에는 매우 유용**
   A. API 서비스를 개발시, 서버의 모드를 production과 dev 모드로 분리
   B. 옵션에 따라 dev 모드등으로 기동시, REST API의 에러 응답 메세지에 에러 스택 정보를 포함해서 리턴하도록 하면, 디버깅에 매우 유용하게 사용할 수 있다.

API 버전 관리

API 정의에서 중요한 것중의 하나는 버전 관리이다.

이미 배포된 API 의 경우에는 계속해서 서비스를 제공하면서, **새로운 기능이 들어간 새로운 API를 배포할때는 하위 호환성을 보장하면서 서비스를 제공**해야 하기 때문에, 같은 API라도 **버전에 따라서 다른 기능을 제공하도록 하는 것이 필요**하다.

API의 버전을 정의하는 방법에는 여러가지가 있는데,

Facebook ?v=2.0

salesforce.com /services/data/v20.0/sobjects/Account

필자의 경우에는

{servicename}/{version}/{REST URL}

example) api.server.com/account/v2.0/groups

형태로 정의 하는 것을 권장한다.

이는 서비스의 배포 모델과 관계가 있는데, **자바 애플리케이션의** 경우, account.v1.0.war, account.v2.0.war와 같이 다른 **war로 각각 배포하여 버전별로 배포 바이너리를 관리**할 수 있고, 앞단에 서비스 명을 별도의 URL로 떼어 놓는 것은 향후 서비스가 확장되었을 경우에, account 서비스만 별도의 서버로 분리해서 배포하는 경우를 생각할 수 있다.

외부로 제공되는 URL은 api.server.com/account/v2.0/groups로 하나의 서버를 가르키지만, 내부적으로, HAProxy등의 reverse proxy를 이용해서 이런 URL을 맵핑할 수 있는데, api.server.com/account/v2.0/groups를 내부적으로 account.server.com/v2.0/groups 로 맵핑 하도록 하면, 외부에 노출되는 URL 변경이 없이 향후 확장되었을때 서버를 물리적으로 분리해내기가 편리하다.

페이징

큰 사이즈의 리스트 형태의 응답을 처리하기 위해서는 페이징 처리와 partial response 처리가 필요하다.

리턴되는 리스트 내용이 1,000,000개인데, 이를 하나의 HTTP Response로 처리하는 것은 서버 성능, 네트워크 비용도 문제지만 무엇보다 비현실적이다. 그래서, 페이징을 고려하는 것이 중요하다.

페이징을 처리하기 위해서는 여러가지 디자인이 있다.

예를 들어 100번째 레코드부터 125번째 레코드까지 받는 API를 정의하면

Facebook API 스타일 : /record?**offset=100&limit=25** 100번째 레코드에서부터 25개의 레코드를 출력한다.

Twitter API 스타일 : /record**?page=5&rpp=25** (RPP는 Record per page로 페이지당 레코드수로 RPP=25이면 페이지 5는 100~125 레코드가 된다.)

LikedIn API 스타일 : /record?start=50&count=25

> 구현 관점에서 보면 , 페이스북 API가 **조금 더 직관적이기** 때문에, **페이스북 스타일을 사용하는 것을 권장**한다.

Partial Response 처리

- 리소스에 대한 응답 메세지에 대해서 굳이 모든 필드를 포함할 필요가 없는 케이스가 있다.

페이스북 FEED - 사용자 ID, 이름, 글 내용, 날짜, 좋아요 카운트, 댓글, 사용자 사진 등,

API를 요청하는 Client의 용도에 따라 선별적으로 몇가지 필드만이 필요

- 필드를 제한

  전체 응답의 양을 줄여서 네트워크 대역폭(특히 모바일에서) 절약

  응답 메세지를 간소화하여 파싱등을 간략화

Linked in : /people:(id,first-name,last-name,industry)

Facebook : /terry/friends?fields=id,name

Google : ?fields=title,media:group(media:thumnail)

Linked in 스타일의 경우 가독성은 높지만 :()로 구별하기 때문에, HTTP 프레임웍으로 파싱하기가 어렵다. 전체를 하나의 URL로 인식하고, :(부분을 별도의 Parameter로 구별하지 않기 때문이다.

Facebook과 Google은 비슷한 접근 방법을 사용하는데, 특히 Google의 스타일은 더 재미있는데, group(media:thumnail) 와 같이 **JSON의 Sub-Object 개념을** 지원한다.

Partial Response는 **Facebook 스타일이 구현하기가 간단**하기 때문에, 필자의 경우는 Facebook 스타일의 partial response를 사용하는 것을 권장한다.

검색 (전역검색과 지역검색)

- HTTP GET에서 Query String에 검색 조건을 정의

  검색조건이 다른 Query String과 섞여 버릴 수 있다.

  /users?name=cho&region=seoul

  + 페이징 처리

  /users?name=cho&region=seoul&offset=20&limit=10

  페이징 처리에 정의된 offset과 limit가 검색 조건인지 아니면 페이징 조건인지 분간이 안간다.

- 쿼리 조건은 하나의 Query String으로 정의 **Deleminator**

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

검색 조건을 URLEncode를 써서 "q=name**%3D**cho,region**%3D**=seoul" 처럼

(실제로는 q= name=cho,region=seoul)표현,

검색 조건은 서버에 의해서 토큰 단위로 파싱되어야 한다.

- 검색의 범위

  시스템에 user,dogs,cars와 같은 리소스가 정의되어 있을때

  - 전역 검색

    전체 리소스에 대한 검색 정의

    /search?q=id%3Dterry

  - 리소스에 대한 검색

    특정 리소스에 대한 검색을 정의

    /users?q=id%3Dterry

HATEOS를 이용한 링크 처리(Hypermedia as the engine of application state)

하이퍼미디어의 특징을 이용하여 HTTP Response에 다음 Action이나 관계되는 리소스에 대한 HTTP Link를 함께 리턴하는 것이다.

- 예를 들어 페이징 처리의 경우, 리턴시, 전후페이지에 대한 링크를 제공한다거나

```
{
    [
        {
            "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"
        }
    ]
}
```

- 연관된 리소스에 대한 디테일한 링크를 표시

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

- **Self-Descriptive** 특성이 증대
  - API에 대한 가독성이 증가
  - 응답 메세지가 다른 리소스 URI에 대한 의존성을 가지기 때문에, 구현이 다소 까다롭다

단일 API 엔드포인트 활용

1. API 서버가 물리적으로 분리된 여러개의 서버에서 동작

   user.apiserver.com, car.apiserver.com과 같이 API 서비스마다 URL이 분리되어 있으면 개발자가 사용하기 불편하다.

   매번 다른 서버로 연결을 해야하거니와 중간에 방화벽이라도 있으면, 일일이 방화벽을 해제해야 한다.

2. API 서비스는 물리적으로 서버가 분리되어 있더라도 단일 URL을 사용하는 것이 좋다
   1. 방법

      HAProxy, nginx와 같은 reverse proxy를 사용

      HAProxy를 앞에 새우고 api.apiserver.com이라는 단일 URL을 구축한후에

      HAProxy 설정에서

api.apiserver.com/user는 user.apiserver.com 로 라우팅하게 하고

api.apiserver.com/car 는 car.apiserver.com으로 라우팅 하도록 구현하면 된다.

3. 뒷단에 API 서버들이 확장
    1. API를 사용하는 클라이언트 입장에서는 단일 엔드포인트를 보면 된다.
    2. 관리 관점에서도 단일 엔드포인트를 통해서 부하 분산 및 로그를 통한 Audit(감사)등을 할 수 있기 때문에 편리
    3. API에 대한 라우팅을 reverse proxy를 이용해서 함으로써 조금 더 유연한 운용이 가능하다.

REST에 문제점

1. JSON+HTTP 를 쓰면 REST인가?

    REST에 대한 속성을 제대로 이해하고 디자인해야 제대로된 REST 스타일이라고 볼 수 있다.

    4. 리소스를 제대로 정의
    5. CRUD를 HTTP 메서드인 POST/PUT/GET/DELETE에 대해서 맞춰 사용
    6. 에러코드에 대해서 HTTP Response code를 사용

2. 표준 규약이 없다

    그래서 관리가 어렵다.

    SOAP 기반의 웹 서비스와 같이 메시지 구조를 정의하는 WSDL도 없고, UDDI와 같은 서비스 관리체계도 없다. WS-I이나 WS-*와 같은 메시지 규약도 없다.

    REST가 최근 부각되는 이유 자체가 WebService의 복잡성과 표준의 난이도 때문에 Non Enterprise 진영(Google, Yahoo, Amazone)을 중심으로 집중적으로 소개된 것이다. 데이터에 대한 의미 자체가 어떤 비즈니스 요건처럼 Mission Critical한 요건이 아니기 때문에, 서로 데이터를 전송할 수 있는 정도의 상호 이해 수준의 표준만이 필요했지 Enterprise 수준의 표준이 필요하지도 않았고, 벤더들처럼 이를 주도하는 회사도 없었다.

    단순히 많이 사용하고 암묵적으로 암암리에 생겨난 표준 비슷한 것이 있을 뿐이다(이런 것을 **Defactor 표준**이라고 부른다).

    그런데 문제는 정확한 표준이 없다 보니, 개발에 있어 이를 관리하기가 어렵다는 것이다. 표준을 따르면 몇 가지 스펙에 맞춰서 맞춰 개발 프로세스나 패턴을 만들 수 있는데, REST에는 표준이 없으니 REST 기반으로 시스템을 설계하자면 사용할 REST에 대한 자체 표준을 정해야 하고, 어떤 경우에는 REST에 대한 잘못된 이해로 잘못된 REST 아키텍처에 '이건 REST다'는 딱지를 붙이기도 한다. 실제로 WEB 2.0의 대표 주자격인 Flickr.com도 REST의 특성을 살리지 못하면서 RPC 스타일로 디자인한 API를 HTTP + XML을 사용했다는 이유로 Hybrid REST라는 이름을 붙여 REST 아키텍쳐아키텍처에 대한 혼란을 초래했다.

    근래에 들어서 YAML등과 같이 REST 에 대한 표준을 만들고자 하는 움직임은 있으나, JSON의 자유도를 제약하는 방향이고 Learning curve가 다소 높기 때문에, 그다지 확산이 되지 않고 있다.

이런 비표준에서 오는 관리의 문제점은, **제대로된 REST API 표준 가이드와, API 개발 전후로 API 문서(Spec)을 제대로 만들어서 리뷰하는 프로세스를 갖추는 방법으로 해결하는 방법이 좋다.**

3. 기존의 전통적인 RDBMS에 적용시키기에 쉽지 않다.
   A. 리소스가 DB의 하나의 Row가 되는 경우
   B. DB의 경우는 Primary Key가 복합 Key 형태로 존재하는 경우
      - 여러 개의 컬럼이 묶여서 하나의 PK가 되는 경우
      - DB에서는 유효한 설계일지 몰라도, HTTP URI는 /에 따라서 계층 구조를 가지기 때문에, 이에 대한 표현이 매우 부자연스러워진다.

      예를 들어 DB의 PK가 "세대주의 주민번호"+"사는 지역"+"본인 이름일 때" DB에서는 이렇게 표현하는 것이 하나 이상할 것이 없으나, REST에서 이를 userinfo/{세대주 주민번호}/{사는 지역}/{본인 이름} 식으로 표현하게 되면 다소 이상한 의미가 부여될 수 있다.

      이외에도 resource에 대한 Unique한 Key를 부여하는 것에 여러가지 애로점이 있는데, 이를 해결하는 대안으로는 Alternative Key (AK)를 사용하는 방법이 있다. 의미를 가지지 않은 Unique Value를 Key로 잡아서 DB Table에 AK라는 필드로 잡아서 사용 하는 방법인데. 이미 Google 의 REST도 이러한 AK를 사용하는 아키텍쳐를 채택하고 있다.

      그러나 DB에 AK 필드를 추가하는 것은 전체적인 DB설계에 대한 변경을 의미하고 이는 즉 REST를 위해서 전체 시스템의 아키텍쳐에 변화를 준다는 점에서 REST 사용시 아키텍쳐적인 접근의 필요성을 의미한다.

      그래서 근래에 나온 **mongoDB**나 CouchDB,Riak등의 **Document based NoSQL**의 경우 **JSON Document를 그대로 넣을 수 있는 구조를 갖추는데, 하나의 도큐먼트를 하나의 REST 리소스로 취급하면 되기 때문에 REST의 리소스 구조에 맵핑 하기가 수월**하다.

#3 API 보안

REST API 보안

- 대부분의 통신이 이 API들을 이용해서 이루어진다.

  앱-서버 통신 / 자바스크립트 웹 클라이언트 - 서버간의 통신 등

  한번 보안이 뚫려 버리면 개인 정보가 탈취되는것 뿐만 아니라 많은 큰 문제를 야기할 수 있다.

'

REST API 보안 관점

1. 인증 (Authentication)

누가 서비스를 사용하는지를 확인하는 절차

> 웹 사이트에 사용자 아이디와 비밀 번호를 넣어서, 사용자를 확인하는 과정이 인증이다.

- API 인증

  API를 호출하는 대상 (단말, 다른 서버, 사용자...)을 확인하는 절차

2. 인가 (Authorization)

인가는 해당 리소스에 대해서, 사용자가 그 리소스를 사용할 권한이 있는지 체크하는 **권한 체크** 과정

> /users라는 리소스가 있을 때, **일반 사용자 권한**으로는 내 사용자 정보만 볼 수 있지만, **관리자 권한**으로는 다른 사용자 정보를 볼 수 있는 것과 같은 **권한의 차이를** 의미한다.

3. 네트워크 레벨 암호화

인증과 인가 과정이 끝나서 API를 호출하게 되면, 네트워크를 통해서 데이터가 통신 중, 해커등이 중간에 이 네트워크 통신을 낚아 채서(감청) 데이터를 볼 수 없게 할 필요가 있다.

이를 **네트워크 프로토콜단에서 처리**하는 것을 **네트워크 레벨의 암호화**라고 하는데, **HTTP에서의 네트워크 레벨 암호화는 일반적으로 HTTPS 기반의 보안 프로토콜**을 사용한다.

4. 메시지 무결성 보장

메시지가 중간에 해커와 같은 외부 요인에 의해서 **변조가 되지 않게 방지**하는 것

- **메시지에 대한 Signature를 생성**해서 **메시지와 같이 보낸 후에 검증**하는 방식

  메시지 문자열에 대한 해쉬코드를 생성해서, 문자열과 함께 보낸 후, 수신쪽에서 받은 문자열과 이 받은 문자열로 생성한 해쉬 코드를 문자열과 함께 온 해쉬코드와 비교하는 방법이 있다. 만약에 문자열이 중간에 변조되었으면, 원래 문자열과 함께 전송된 해쉬코드와 맞지 않기 때문에 메시지가 중간에 변조가되었는지 확인할 수 있다.

네트워크 레벨의 암호화를 완벽하게 사용한다면 외부적인 요인(해커)등에 의해서 메시지를 해석당할 염려가 없기 때문에 사용할 필요가 없다.

5. 메시지 본문 암호화

메시지 자체를 암호화하는 방법

네트워크 레벨의 암호화를 사용/신뢰할 수 없는 상황

- 애플리케이션 단에서 구현
  - 전체 메시지를 암호화
    - 암호화에 소요되는 비용이 크다
    - 중간에 API Gateway등을 통해서 메시지를 열어볼 수 있다.
    - 메시지 기반으로 라우팅 변환하는 작업등이 어렵다.
  - 특정 필드만 암호화
    - 일반적으로 사용한다.

인증 (Authentication)

1. API Key 방식

가장 기초적인 방법

- API Key

  특정 사용자만 알 수 있는 일종의 문자열

  개발자는 API 제공사의 포탈 페이등 등에서 API Key를 발급 받고, API를 호출할 때 API Key를 메시지 안에 넣어 호출한다. 서버는 메시지 안에서 API Key를 읽어 이 API가 누가 호출한 API인지를 인증하는 흐름이다.

모든 클라이언트들이 같은 API Key를 공유하기 때문에 한번 API Key가 **노출이 되면 전체 API가 뚫려 버리는 문제**가 있기 때문에 **높은 보안 인증을 요구하는 경우에는 권장**하지 않는다.

2. API Token 방식

사용자 ID, PASSWD등으로 사용자를 인증한 후에, 그 사용자가 API 호출에 사용할 기간이 유효한 API Token을 발급해서 API Token으로 사용자를 인증하는 방식이다.

- ID, PASSWD대신 API Token을 사용하는 이유
  - PASSWD는 주기적으로 바뀔 수 있다

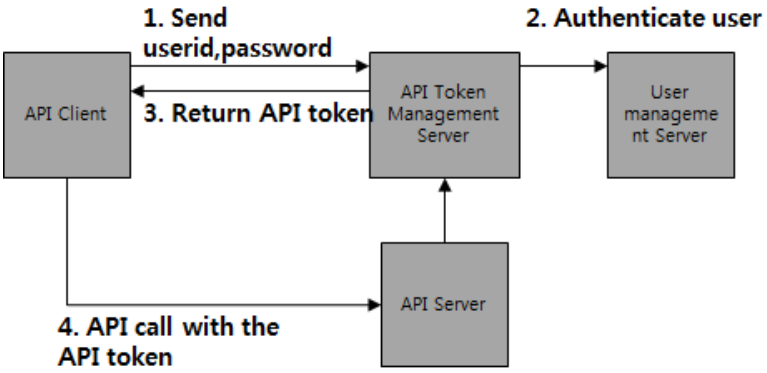
■ 매번 네트워크를 통해서 사용자 ID와 PASSWD를 보내는 것은 보안적으로 사용자 계정 정보를 탈취당할 가능성이 높다.

◆ API Token의 탈취

API를 호출가능

● ID와 PASSWD의 탈취

일반적으로 사용자들은 다른 서비스에도 같은 PASSWD를 사용하는 경우가 많기 때문에 연쇄적으로 다른 서비스에 대해서도 공격을 당할 수 있는 가능성이 높다.



1. API Client가 사용자 ID, PASSWD를 보내서 API호출을 위한 API Token을 요청한다.
2. API 인증 서버는 사용자 ID, PASSWD를 가지고, 사용자 정보를 바탕으로 사용자를 인증한다.
3. 인증된 사용자에 대해서 API Token을 발급한다. (유효 기간을 가지고 있다.)
4. API Client는 이 API Token으로 API를 호출한다. API Server는 API Token이 유효한지를 API Token 관리 서버에 문의하고, API Token이 유효하면 API 호출을 받아들인다.

1단계의 사용자 인증 단계에서는 보안 수준에 따라서 여러가지 방식을 사용할 수 있다.

1. HTTP Basic Auth

가장 기본적이고 단순한 형태의 인증 방식

사용자 ID와 PASSWD를 HTTP Header에 Base64 인코딩 형태로 넣어서 인증을 요청한다.

ID가 terry이고 PASSWD가 hello world일 때, 다음과 같이 HTTP 헤더에 "terry:hello world"라는 문자열을 Base64 인코딩을 해서 "Authorization"이라는 이름의 헤더로 서버에 전송하여 인증을 요청한다.

Authorization: Basic VGVycnk6aGVsbG8gd29ybGQ=

중간에 패킷을 가로채서 이 헤더를 Base64로 디코딩하면 사용자 ID와 PASSWD가 그대로 노출되기 때문에 반드시 HTTPS 프로토콜을 사용해야 한다.

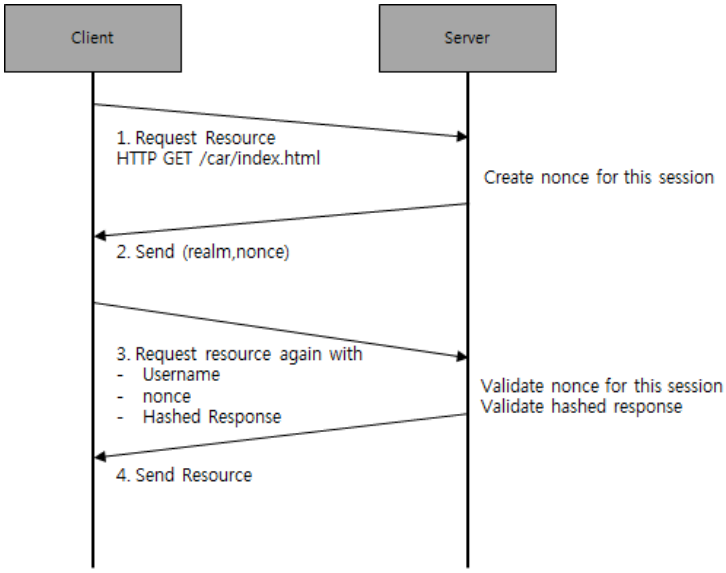
2. Digest access Authentication

HTTP Basic Auth 단점을 보강

i. 클라이언트가 인증을 요청할 때, 클라이언트가 서버로부터 nonce 라는 일종의 난수값을 받은 후에, (서버와 클라이언트는 이 난수 값을 서로 알고 있음)

ii. 사용자 ID와 PASSWD를 이 난수값을 이용해서 HASH화하여 서버로 전송하는 방식이다.

- 평문 형태로 날아가지 않기 때문에, 해커가 탈취할 수 없다.
- HASH 알고리즘을 알고 있다고 하더라도, HASH된 값에서 반대로 PASSWD를 추출하기가 어렵다



1. 클라이언트가 서버에 특정리소스 /car/index.html 을 요청한다.
2. 서버는 해당 세션에 대한 nonce값을 생성하여 저장한 후에, 클라이언트에 리턴한다.

   이때 realm을 같이 리턴하는데, realm은 인증의 범위로, 예를 들어 하나의 웹 서버에 car.war, market.war가 각각 http://myweb/car , http://myweb/market 이라는 URL로 배포가 되었다고 하면, 이 웹사이트는 각각 애플리케이션 car.war와 market.war에 대해서 서로 다른 인증realm을 갖는다.

※ **해당 session에 대해서 nonce 값을 유지 저장해야 하기 때문에, 서버 쪽에서는 상태 유지에 대한 부담**이 생긴다. **HTTP Session을 사용하거나 또는 서버간에 공유 메모리(memcached나 redis등)을 넣어서 서버간에 상태 정보를 유지할 수 있는 설계가 필요**하다.

3. 클라이언트는 앞에서 서버로부터 받은 realm과 nonce값으로 Hash 값을 생성

   HA1 = MD5(사용자이름:realm:비밀번호)
   HA2 = MD5(HTTP method:HTTP URL)
   response hash = MD5(HA1:nonce:HA2)
   를 통해서 response hash 값을 생성한다.

예를 들어서 /car/index.html 페이지를 접근하려고 했다고 하자, 서버에서 nonce값을 dcd98b7102dd2f0e8b11d0f600bfb0c093를 리턴하였고, realm은 car_realm@myweb.com 이라고 하자. 그리고 사용자 이름이 terry, 비밀 번호가 hello world하면

HA1 = MD5(terry:car_realm@myweb.com:hello world)로 7f052c45acf53fa508741fcf68b5c860 값이 생성되고

HA2 = MD5(GET:/car/index.html) 으로 0c9f8cf299f5fc5c38d5a68198f27247 값이 생성된다.

Response Hash는MD5(7f052c45acf53fa508741fcf68b5c860:
dcd98b7102dd2f0e8b11d0f600bfb0c093:0c9f8cf299f5fc5c38d5a68198f27247) 로 결과는
95b0497f435dcc9019c335253791762f 된다.

클라이언트는 사용자 이름인 "terry"와 앞서 받은 nonce값인 dcd98b7102dd2f0e8b11d0f600bfb0c093와 계산된 hash값인 **95b0497f435dcc9019c335253791762f 값을 서버에게 전송**한다.

4. 서버는 먼저 3에서 전달된 nonce값이 이 세션을 위해서 서버에 저장된 **nonce 값과 같은지 비교**를 한후, 전달된 사용자 이름인 terry와nonce값 그리고 서버에 저장된 사용자 비밀 번호를 이용해서 3번과 같은 방식으로 **response hash 값을 계산**하여 클라이언트에서 전달된 hash값과 같은지 **비교를 하고 같으면 해당 리소스를 (/car/index.html 파일)을 리턴한다.**

간단한 기본 메커니즘만 설명한것이며, **사실 digest access authentication은 qop (quality of protection)이라는 레벨에 따라서 여러가지 변종(추가적인 보안)을 지원**한다. 언뜻 보면 복잡해서 보안 레벨이 높아보이지만 사실 Hash 알고리즘으로 MD5를 사용하는데, 이 **MD5는 보안 레벨이 낮기 때문에** 미정부 보안 인증 규격인 FIPS인증 (http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf) 에서 인증하고 있지 않다. **FIPS 인증에서는 최소한 SHA-1,SHA1-244,SHA1-256 이상의 해쉬 알고리즘을 사용하도록 권장**하고 있다.

- MD5 해쉬

  Dictionary Attack에 취약

  - Dictionary Attack

    Hash된 값과 원래 값을 Dictionary (사전) 데이터 베이스로 유지해놓고, Hash 값으로 원본 메시지를 검색하는 방식

HA1 = MD5("Mufasa:testrealm@host.com:Circle Of Life") = 939e7578ed9e3c518a452acee763bce9

의 MD5 해쉬 값인 939e7578ed9e3c518a452acee763bce9 값을 가지고, MD5 Dictionary 사이트인
http://md5.gromweb.com/?md5=939e7578ed9e3c518a452acee763bce9 에서 검색해보면, Hash 값으로 원본 메시지인 Mufasa:testrealm@host.com:Circle Of Life 값이 Decrypt 되는 것을 확인할 수 있다.

그래서 반드시 추가적인 보안 (HTTPS) 로직등을 겸비해서 사용하기를 바라며, 더 높은 보안 레벨이 필요한 경우 다른 인증 메커니즘을 사용하는 것이 좋다.

- **FIPS 인증 수준의 보안 인증 프로토콜로**

  SHA-1 알고리즘을 사용하는 SRP6a 등

  높은 수준의 보안이 필요할 경우에는 아래 링크를 참고하기 바란다.
  http://en.wikipedia.org/wiki/Secure_Remote_Password_protocol

3. 클라이언트 인증 추가

추가적인 보안 강화

페이스북 - API Token을 발급받기 위해서, 사용자 ID, PASSWD 뿐만 아니라 client Id와 Client Secret을 입력 받도록 한다.

Client Id - 특정 앱에 대한 등록 Id

Client Secret - 특정 앱에 대한 비밀 번호로 페이스북 개발자 포털에서 앱을 등록하면 앱 별로 발급 되는 일종의 비밀 번호이다.

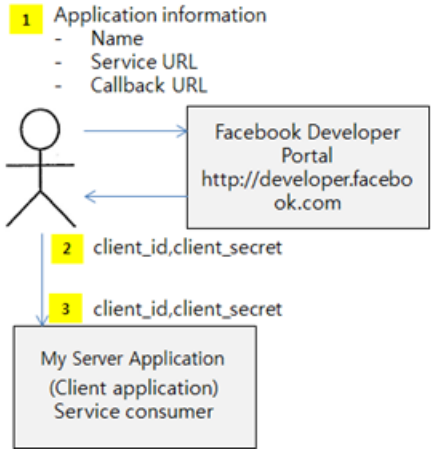
4. 제3자 인증 방식

페이스북이나 트위터와 같은 API 서비스 제공자들이 **파트너 애플리케이션에 많이 적용하는 방**법

My Server Application라는 서비스를 Facebook 계정을 이용하여 인증을 하는 경우이다.

- 서비스 My Server Application에 대해서 해당 사용자가 페이스북 사용자임을 인증
- 서비스 My Server Application는 사용자의 비밀번호를 받지 않고, 페이스북이 사용자를 인증하고 서비스 My Server Application에게 알려주는 방식이다.

파트너 서비스에 페이스북 사용자의 비밀번호가 노출되지 않는 방식이다.

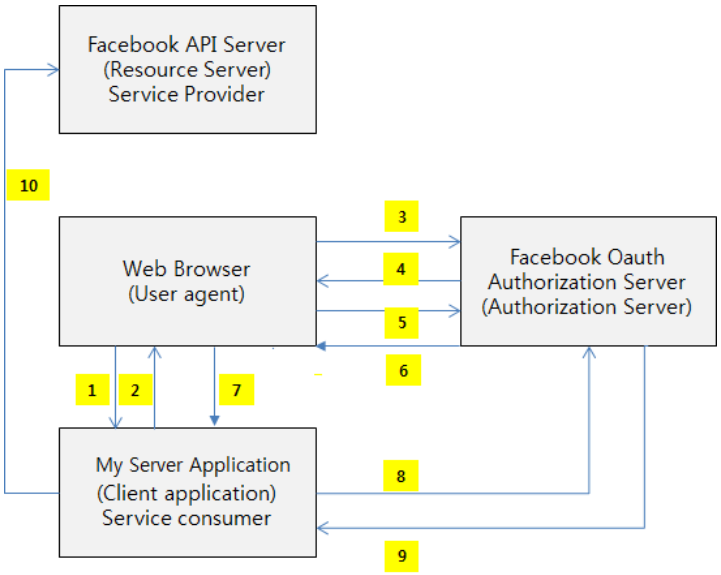


1. 먼저 페이스북의 Developer Portal에 접속을 하여, 페이스북 인증을 사용하고자 하는 애플리케이션 정보를 등록한다. (**서비스 명**, **서비스 URL**, 그리고 인증이 성공했을 때 **인증 성공 정보를 받을 CallBack URL**)

2. 페이스북 Developer Portal은 등록된 정보를 기준으로 해당 애플리케이션에 대한 **client_id와 client_secret을 발급**한다. 이 값은 앞에서 설명한 **클라이언트 인증**에 사용된다.

3. 다음으로, **개발하고자 하는 애플리케이션에, 이 client_id와 client_secret등을 넣고, 페이스북 인증 페이지 정보 등을 넣어서 애플리케이션을 개발**한다.

애플리케이션이 개발되서 실행이 되면, 아래와 같은 흐름에 따라서 사용자 인증을 수행하게 된다.



1. 웹브라우져에서 사용자가 My Server Application 서비스를 접근하려고 요청한다.

2. My Server Application은 사용자가 인증이되어 있지 않기 때문에, 페이스북 로그인 페이지 URL을 HTTP Redirection으로 URL을 브라우져에게 보낸다. 이때 이 URL에 페이스북에 **이 로그인 요청이 My Server Application에 대한 사용자 인증 요청임을 알려주기 위해서, client_id등의 추가 정보와 함께, 페이스북의 정보 접근 권한 (사용자 정보, 그룹 정보등)을 scope라는 필드를 통해서 요청한다.**

3. 브라우져는 페이스북 로그인 페이지로 이동하여, 2단계에서 받은 추가적인 정보와 함께 로그인을 요청한다.

4. 페이스북은 사용자에게 로그인 창을 보낸다.

5. 사용자는 로그인창에 ID/PASSWD를 입력한다.

6. 페이스북은 사용자를 인증하고, 인증 관련 정보과 함께 브라우져로 전달하면서, My Server Application의 로그인 완료 페이지로 Redirection을 요청한다.

7. My Server Application을 6에서 온 인증 관련 정보를 받는다.

8. My Server Application은 이 정보를 가지고, 페이스북에, 이 사용자가 제대로 인증을 받은 사용자인지를 문의한다.

9. 페이스북은 해당 정보를 보고 사용자가 제대로 인증된 사용자임을 확인해주고, API Access Token을 발급한다.

10. My Server Application은 9에서 받은 API Access Token으로 페이스북 API 서비스에 접근한다.

앞에서 설명했듯이, 이러한 방식은 자사가 아닌 파트너 서비스에게 자사 서비스 사용자의 인증을 거쳐서 API의 접근 권한을 전달하는 방식이다.

이러한 인증 방식의 대표적인 구현체는 **OAuth 2.0**으로, 이와 같은 제3자 인증뿐만 아니라, 직접 자사의 애플리케이션을 인증하기 위해서, 클라이언트로부터 직접 ID/PASSWD를 입력 받는 등.

클라이언트 타입(웹,서버,모바일 애플리케이션)에 대한 다양한 시나리오를 제공한다.

※ OAuth 2.0에 대한 자세한 설명은 PACKT 출판사의 OAuth 2.0 Identity and Access Management Patterns (by Martin Spasovski) 책을 참고하기를 추천한다.

이러한 3자 인증 방식은 일반적인 서비스에서는 필요하지 않지만, **자사의 API를 파트너등 외부 시스템에 제공하면서 사용자의 ID/PASSWD를 보호하는데는 필요한 서비스이기 때문에, API를 외부에 적용하는 경우에는 고려를 해야 한다.**

5. IP White List을 이용한 터널링

- API를 호출하는 클라이언트의 API가 일정할 때
- 서버간의 통신이나 타사 서버와 자사 서버간의 통신 같은 경우에, API 서버는

특정 API URL에 대해서 들어오는 IP 주소를 White List로 유지하는 방법

- API 서버 앞단에, HAProxy나 Apache와 같은 웹서버를 배치하여서 특정 URL로 들어올 수 있는 IP List를 제한
- 전체 API가 특정 서버와의 통신에만 사용된다면 아예, 하드웨어 방화벽 자체에 들어올 수 있는 IP List를 제한할 수 있다.

설정만으로 가능한 방법이기 때문에, 서버간의 통신이 있는 경우에는 적용하는 것을 권장한다.

6. Bi-diretional Certification (Mutual SSL)

가장 높은 수준의 인증 방식을 제공

- HTTPS 통신을 사용할 때 **서버에 공인 인증서**를 놓고 **단방향**으로 SSL을 제공한다.
- Bi-Directional Certification (**양방향** 인증서 방식) 방식은 **클라이언트에도 인증서**를 놓고 양방향으로 SSL을 제공하면서, **API 호출에 대한 인증을 클라이언트의 인증서를 이용**하는 방식이다.

구현 방법이 가장 복잡

공인 기관에서 발행된 인증서를 사용한다면 API를 호출하는 쪽의 신원을 확실하게 할 수 있고, 메시지까지 암호화되기 때문에, 가장 높은 수준의 인증을 제공한다.

권한 인가 (Authorization)

사용자가 인증을 받고 로그인을 했다해더라도 해당 API를 호출할 수 있는 권한이 있는 가를 체크해야 한다.

API 인가 방식

1. RBAC (Role Based Access Control)

   가장 일반적

   사용자의 역할(ROLE)을 기반

   정해진 ROLE에 권한을 연결해놓고, 이 ROLE을 가지고 있는 사용자에게 해당 권한을 부여

   일반 관리자 - 사용자 관리, 게시물 관리, 회원 가입 승인

   마스터 관리자 - 까페 게시판 게시판 관리, 메뉴 관리, 사용자 관리, 게시물 관리, 회원 가입 승인

   와 같은 권한을 만든후,

   Terry에 "마스터 관리자"라는 ROLE을 부여하면, 사용자 Terry는 "까페 게시판 게시판 관리, 메뉴 관리, 사용자 관리, 게시물 관리, 회원 가입 승인" 등의 권한을 가지게 된다.

   - Object

     권한 부여의 대상이 되는 사용자나 그룹

   - Permission

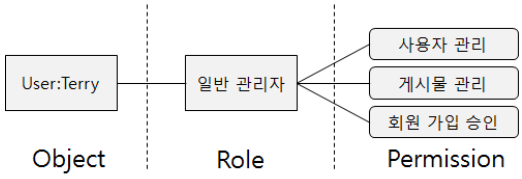
     각 개별 권한

   - Role

     사용자의 역할

   RBAC는 이 Role에 권한을 맵핑 한 다음 Object에 이 Role을 부여하는 방식

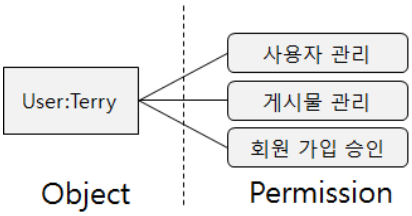
   많은 권한 인가는 사용자 역할을 기반으로 하기 때문에, 사용하기가 용이하다.



2. ACL (Access Control List)

   - RBAC 방식 - 권한을 ROLE이라는 중간 매개체를 통해서 사용자에게 부여
   - ACL 방식 - 사용자(또는 그룹과 같은 권한의 부여 대상) 에게 **직접 권한을 부여**

사용자 Terry에 직접 "까페 게시판 게시판 관리, 메뉴 관리, 사용자 관리, 게시물 관리, 회원 가입 승인" 권한부여



인증 (Authentication) -> 인가에 사용된 api accesstoken을 이용하여 사용자 정보를 조회

-> 사용자 정보에 연관된 권한 정보 (Permission이나 Role정보)를 받아서 이 권한 정보를 기반으로 API 사용 권한을 인가

사용자 정보 조회 "HTTP GET /users/{id}"라는 API 가 있다고 가정하자.
일반 사용자 - 자신의 id에 대해서만 사용자 정보 조회가 가능
관리자 - 다른 사용자의 id도 조회가 가능하도록 차등하여 권한을 부여할 수 있다.

API 권한 인가 처리 위치

1. 클라이언트 - API를 호출
2. API 서버 – API 실행

   가장 일반적인 처리 위치
3. gateway - API에 대한 중간 길목

1. 클라이언트에 의한 API 권한 인가 처리

- 클라이언트가 신뢰할 수 있는 경우에만 사용할 수 있다.

웹 UX 로직이 서버에 배치되어 있는 형태 (Struts나 Spring MVC와 같은 웹 레이어가 있는 경우)에 주로 사용했다.

- 웹 애플리케이션에서, **사용자 로그인 정보(세션 정보와 같은)**를 **보고 사용자 권한을 조회한 후에, API를 호출하는 방식**이다.

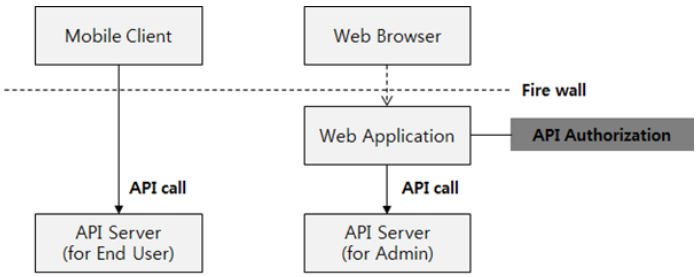
사용자 세션에, 사용자 ID와 ROLE을 본 후에, ROLE이 일반 사용자일 경우, 세션내의 사용자 ID와 조회하고자 하는 사용자 ID가 일치하는 경우에만 API를 호출하는 방식이다.

모바일 디바이스 등에 제공하는 API는 사용자 ROLE을 갖는 API와 같이 별도의 권한 인가가 필요 없는 API를 호출하는 구조를 갖는다.

- Mobile Client - 일반 사용자만 사용, 웹 애플리케이션 - 일반 사용자, 관리자 모두 사용
- 일반 사용자의 Mobile Client를 위한 API Server

  사용자 인증(Authentication)만 되면 모든 API 호출을 허용
- 웹 애플리케이션의 경우에는 일반 사용자냐, 관리자냐에 따라서 권한 인가가 필요

  Web Application에서, API를 호출하기 전에 사용자의 id와 권한에 따라서 API 호출 여부를 결정하는 API 권한 인가(Authorization) 처리

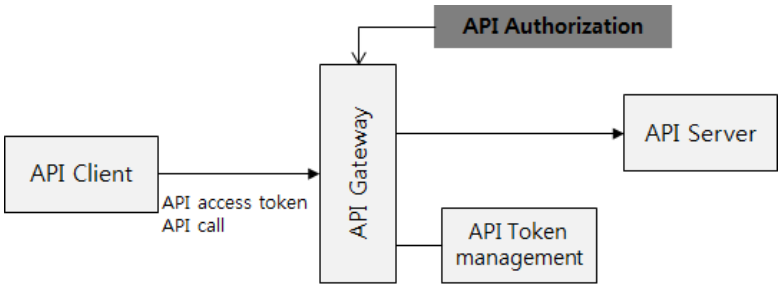


2. Gateway에 의한 권한 인가 처리

권한 인가는 모바일 클라이언트, 자바스크립트 기반의 웹 클라이언트등 다양한 클라이언트가 지원됨에 따라 **점차 서버쪽으로 이동**

- 자바 스크립트 클라이언트
  - 클라이언트에서 권한에 대한 인가는 의미가 없다

    자바 스크립트의 경우 브라우져의 디버거등으로 코드 수정이 가능하기 때문에 권한 처리 로직을 우회할 수도 있고 또한 API 포맷만 안다면 직접 API를 서버로 호출해서 권한 인가 없이 API를 사용할 수 있다.
  - 서버 쪽에서 권한 인가 처리
    - API gateway에 의한 권한 처리는 구현이 쉽지 않다.
    - API 서버에서 권한 처리를 하는 것이 일반적



API gateway에서 권한 인가를 처리하는 방법인데, API 호출이 들어오면, API access Token을 사용자 정보와 권한 정보로 API token management 정보를 이용해서 변환한 후에, 접근하고자 하는 API에 대해서 권한 인가 처리를 한다.

이는 API 별로 API를 접근하고자 하는데 필요한 권한을 체크해야 하는데, HTTP GET /users/{id}의 API를 예로 들어보면, 이 URL에 대한 API를 호출하기 위해서는 일반 사용자 권한을 가지고 있는 사용자의 경우에는 호출하는 사용자 id와 URL상의 {id}가 일치할 때 호출을 허용하고, 같지 않을 때는 호출을 불허해야 한다.

만약 사용자가 관리자 권한을 가지고 있을 경우에는 호출하는 사용자 id와 URL상의 {id}가 일치하지 않더라도 호출을 허용해야 한다.

그러나 이러한

- **api gateway에서의 권한 인가 어렵다.**

  /users/{id} API의 경우에는 사용자 id가 URL에 들어가 있기 때문에, API access token과 맵핑되는 사용자 ID와 그에 대한 권한을 통해서 API 접근 권한을 통제할 수 있지만

  - API에 따라서 사용자 id나 권한 인증에 필요한 정보가 HTTP Body에 json 형태나 HTTP Header 등에 들어가 있는 경우
    - 일일이 메세지 포맷에 따라서 별도의 권한 통제 로직을 gateway 단에서 구현해야 하는 부담
    - 권한 통제를 위해서 HTTP 메세지 전체를 일일이 파싱해야 하는 오버로드가 발생

3. 서버에 의한 API 권한 인가 처리

API 요청을 처리하는 API 서버의 비지니스 로직단에서 권한 처리를 하는 방식

- **각 비지니스 로직에서 API 메세지를 각각 파싱**

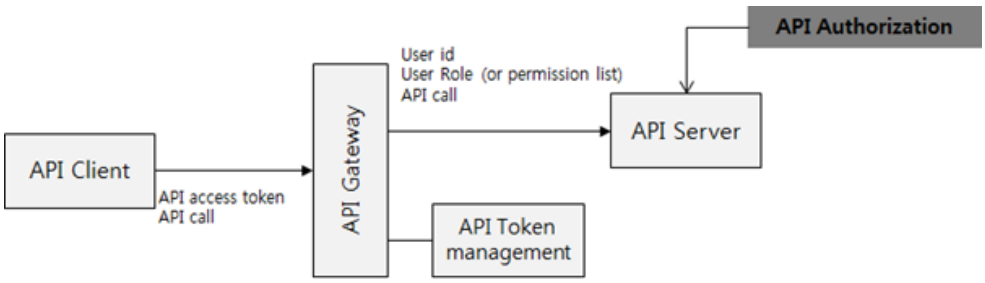
  API 별로 권한 인가 로직을 구현하기가 용이

- **권한 인가에 필요한 필드들을 api gateway에서 변환해서 API 서버로 전달해줌으로써 구현을 간략화**

I. API 클라이언트가 api access token을 이용해서 API를 호출

II. api gateway가 이 access token을 권한 인가에 필요한 사용자 id, role등으로 변환해서 API 서버에 전달

III. 각 비지니스 로직은 API 권한 인가에 필요한 사용자 정보등을 별도로 데이타 베이스를 뒤지지 않고 이 헤더의 내용만을 이용해서 API 권한 인가 처리



4. 네트워크 (전송) 레벨 암호화

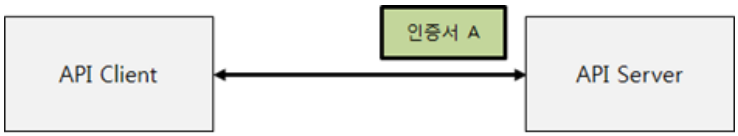
- 가장 기본적이고 필수적인 REST API 보안 방법
- 네트워크 전송 프로토콜에서 HTTPS 보안 프로토콜을 사용

  메시지 자체를 암호화해서 전송하기 때문에, 해킹으로 인한 메시지 누출 위협을 해소
- Man-in-The-Middle-Attack

  HTTPS를 사용하더라도, 메시지를 낚아채거나 변조하는 방법

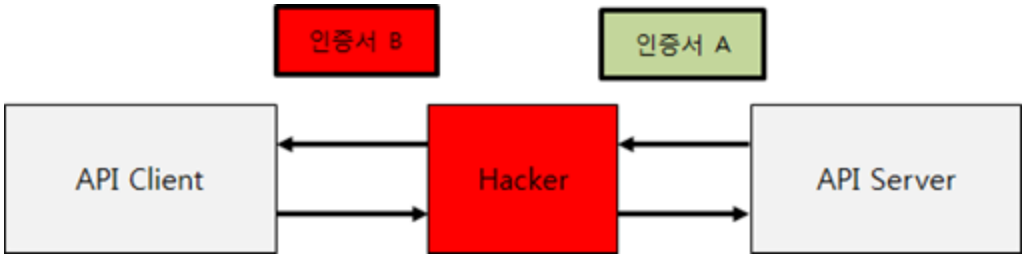
1. 정상적인 HTTPS 통신

서버에서 제공하는 인증서 A를 이용하여 API 클라이언트와 서버 상화간에 암호화된 신뢰된 네트워크 연결을 만든다.



2. Man in the middle attack

신뢰된 연결을 만들려고 할 때, 해커가 API 클라이언트와 서버 사이에 끼어 들어온다.



신뢰된 연결을 만들기 위해서 서버가 인증서 A를 클라이언트에게 내릴 때 해커가 이 인증서가 아닌 인증서 B를 클라이언트에 내리고, 인증서 B를 이용해서 API Client와 Hacker간에 HTTPS SSL 연결을 만든다. 그리고는 서버에게서 받은 인증서 A를 이용해서 해커와 API 서버간의 HTTP SSL 서버를 만든다.

이렇게 되면, 해커는 중간에서 API 클라이언트와 서버 사이에 메시지를 모두 열어 보고 변조도 가능하게 된다.

종종 대기업이나 공공 기관에서 웹브라우져를 사용하다 보면, 인증서가 바뀌었다는 메시지를 볼 수 가 있는데 (특히 파이어폭스 브라우져를 사용하면 인증서 변경을 잘 잡아낸다.) 이는 회사의

보안 정책상 HTTPS 프로토콜의 내용을 보고, 이를 감사하기 위한 목적으로 사용된다.

- Man in middle attack을 방지하는 방법

  가장 손쉬운 방법은 **공인된 인증서**를 사용하고 **인증서를 체크**하는 것이다.

해커가 인증서를 바꿔 치려면, 인증서를 발급해야 하는데, 공인 인증서는 Verisign과 같은 기간에서 인증서에 대한 공인 인증을 해준다. 즉, 이 **인증서를 발급한 사람이 누구이고 이에 대한 신원 정보를 가지고 있다. 이를 공인 인증서**라고 하는데, 공인 **인증서는 인증 기관의 Signature로 싸인이 되어 있다. (공인 인증기관이 인증했다는 정보가 암호화 되서 들어간다**.)

만약 해커가 공인 인증서를 사용하려면 인증 기관에 가서 개인 정보를 등록해야 하기 때문에, 공인 인증서를 사용하기 어렵고 보통 자체 발급한 비공인 인증서를 사용하기 때문에, 이를 이용해서 체크가 가능하고, 특히 인증서안에는 인증서를 발급한 기관의 정보와 인증서에 대한 고유 Serial 번호가 들어가 있기 때문에, 클라이언트에서 이 값을 체크해서 내가 발급하고 인증 받은 공인 인증서인지를 체크하도록 하면 된다.

5. 메시지 본문 암호화

간단하게 암호화가 필요한 특정 필드만 애플리케이션단에서 암호화하여 보내는 방법이 있다.

- 클라이언트와 서버가 암호화 키를 가진다

  대칭키와, 비대칭키 알고리즘

1. 비대칭키 알고리즘

   암호화를 시키는 키와, 암호를 푸는 복호화 키가 다른 경우

   - 암호화 시키는키 Public Key (공개키)

     공개키는 암호화는 할 수 있지만 반대로 암호화된 메시지를 풀 수 가 없기 때문에 누출이 되더라도 안전하다.

   - 암호화를 푸는키를 Private Key(비밀키)

   - RSA

     HTTPS의 경우에도 이 RSA 알고리즘을 사용한다.

   - 클라이언트에서 서버로 보내는 단방향 메시지에 대해서는 암호화하여 사용할 수 있지만 반대로 서버에서 클라이언트로 내려오는 응답 메시지등에는 적용하기가 어렵다.

   - 클라이언트가 서버에 등록될 때, 위와 반대 방법으로 클라이언트에서 비공개키와 공

개키 쌍을 생성한 후에, 서버로 공개키를 보내서 향후 서버에서 클라이언트로의 통신을 그 공개키를 사용하도록 해도 된다.

◆ 클라이언트-서버, 서버-클라이언트간의 키 쌍 두개를 관리해야 하기 때문에 복잡할 수 있는데, 이런 경우에는 대칭키 알고리즘을 고려해볼 수 있다.

2. 대칭키 알고리즘은 암호화와 복호화키가 같은 알고리즘이다.

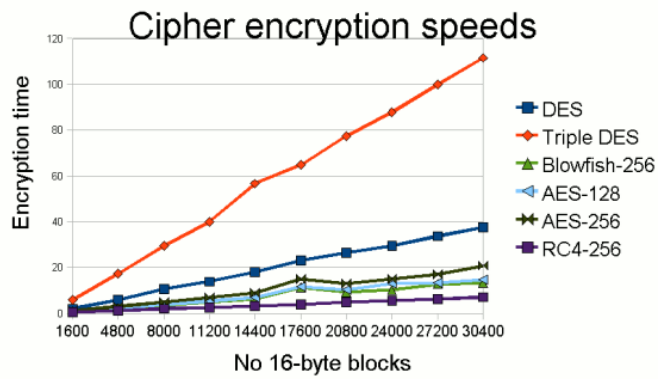
■ API 클라이언트와 서버가 같은 키를 알고 있어야 한다.

■ 네트워크를 통해서 보낼 경우 중간에 해커에 의해서 낚아채질 염려가 있기 때문에, 양쪽에 안전하게 키를 전송하는 방법이 필요하다.

1. 서버에서 공개키KA1와 비공개키KA2 쌍을 생성한다.
2. 클라이언트에게 공개키 KA1을 네트워크를 통해서 내려보낸다.
3. 클라이언트는 새로운 비공개 대칭키 KB를 생성한후 KA1을 이용해서 암호화하여 서버로 전송한다.
4. 서버는 전송된 암호화 메시지를 KA2로 복화화 하여, 그 안에 있는 비공개키 KB를 꺼낸다.
5. 향후 클라이언트와 서버는 상호 API통신시 비공개 대칭키 KB를 이용하여 암호화와 복호화를 진행한다.

■ AES256을 사용하면 빠른 암호화 속도와 높은 보안성을 보장받을 수 있다.

■ 대칭키 기반의 암호화 알고리즘 속도 비교



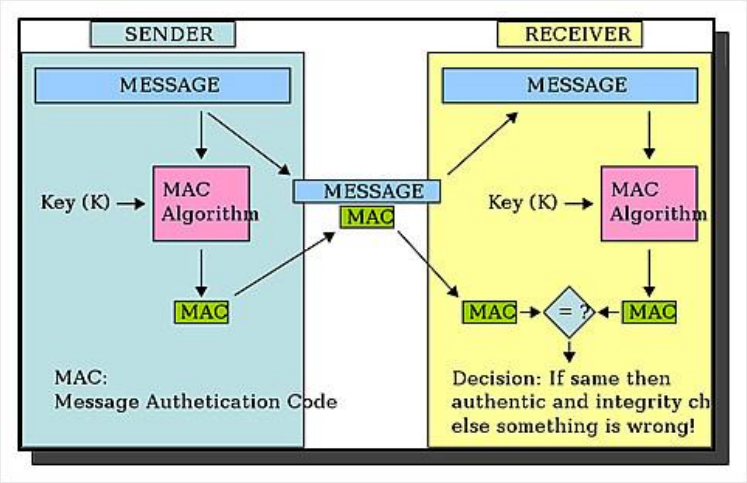
6. 메시지 무결성 보장

- 무결성
  - 서버 입장에서 API 호출을 받았을 때 이 호출이 신뢰할 수 있는 호출인지 아닌지를 구별하는 방법이다.
  - 해커가 중간에서 메시지를 가로챈 후, 내용을 변조하여 서버에 보냈을 때, 내용이 변조되었는지 여부를 판단하는 방법인
  - 일반적으로 HMAC을 이용한 방식이 널리 사용된다

- 전제

  RestAPI를 호출하는 클라이언트와 서버 간에는 **대칭키** 기반의 암호화 키 'Key'를 가지고 있다

이 키는 API access Token을 사용할 수도 있고, 앞의 메시지 본문 암호화에서 나온 방법을 이용해서 서로 대칭키를 교환하여 사용할 수도 있다.



1. 먼저 클라이언트는 호출하고자 하는 REST API의 URL을 앞에서 정의한 Key를 이용하여 HMAC 알고리즘을 사용하여 Hash 값을 추출한다.
   - 중요: 여기서는 편의상 URL을 가지고 HMAC 해시를 생성하였했는데, 전체 메시지에 대한 무결성을 보장하려면 URL이 아니라 메시지 전문 자체에 대해서 대해 Hash를 추출해야 한다.
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 변조하였했을 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.

- replay attack

  만약 메시지를 변경하지 않고 Hacker가 동일한 요청을 계속 보낸다면 메시지를 변조하지 않았기 때문에 서버는 이를 유효한 호출로 인식할 수 있다. 이를 방지하기 위해서 time stamp를 사용하는 방법이 있다.

  이 방법은 HMAC을 생성할 때, 메시지를 이용해서만 Hash 값을 생성하는 것이 아니라 timestamp를 포함하여 메시지를 생성하는 것이다.

HMAC (Key, (메시지 데이터+timestamp))

그리고 API를 호출할 때, timestamp 값을 같이 실어 보낸다.

http://service.myapi.com/restapiservice?xxxxx&hmac={hashvalue}&timestamp={호출시간}

이렇게 하면 서버는 메시지가 호출된 시간을 알 수 있고, 호출된 시간 +-10분(아니면 개발자가 정한 시간폭)만큼의 호출만 정상적인 호출로 인식하고 시간이 지난 호출의 메시지는 비정상적인 호출로 무시하면 된다.

* 참고 : : Hacker가 timestamp URL등 등을 통해서 통해 볼 수 있다고 하더라도, Key 값을 모르기 때문에 timestamp를 변조할 수 없다. timestamp를 변조할 경우에는 원본 Hash가 원본 timestamp로 생성되었기 때문에, timestamp가 변조된 경우 hash 값이 맞지 않게 된다.

HMAC을 구현하는 해시 알고리즘에는 MD5, SHA1등 등이 있는데, 보통 SHA-1 256bit 알고리즘을 널리 사용한다.

API보안에서는 최소한 HTTPS를 이용한 네트워크 보안과 함께, API Token등의 인증 방식을 반드시 사용하기를 권장한다.

보안 처리는 하지 않아도, API의 작동이나 사용에는 문제가 없다. 그러나 보안이라는 것은 한번 뚫려버리면 많은 정보가 누출이 되는 것은 물론이고 시스템이 심각한 손상까지 입을 수 있기 때문에, 시간이 걸리더라도 반드시 신경써서 설계 및 구현하는 것을 권장한다.

자바스크립트 클라이언트 지원

- **SPA (Sigle Page Application)**
  - 자바스크립트 기술이 발전하면서 **유행**
  - 브라우져에서 페이지간의 이동없이 자바스크립트를 이용해서 동적으로 페이지를 변경할 수 있는 방식이다.
  - 페이지 reloading이 없기 때문에 반응성이 좋아서 많이 사용
  - SPA의 경우 **서버와의 통신**을 **자바스크립가 직접 XMLHTTPRequest 객체를 이용해서 API 호출을 바로 하는 형태**이다.

- API 보안의 새로운 요구사항

  1. 자바스크립트 클라이언트는 코드 자체가 노출

     자바스크립트 코드는 브라우져로 로딩되서 수행되기 때문에 사용자 또는 해커가 클라이언트 코드를 볼 수 있다.

     그래서 보안 로직등이 들어가 있다고 하더라도 로직 자체는 탈취당할 수 있다.

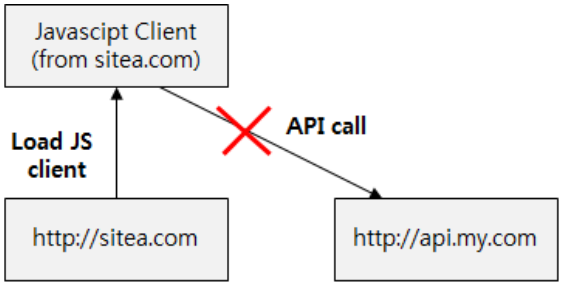
  2. 자바스크립트는 실행중에 브라우져의 디버거를 이용해서 변수 값을 보거나 또는 변수값을 변경하거나 비즈니스 로직을 변경하는 등의 행위가 가능하다.

1. Same Origin Policy에 대한 처리

먼저 자바스크립트의 API에 대한 호출은 **same origin policy(동일 출처 정책)의 제약**을 받는다.

- Same origin policy
  - **API 서버와 Javascript가 호스팅 되는 서버의 URL이 다르기 때문에 발생**

  자바스크립트와 같이 웹 브라우져에서 동작하는 프로그래밍 언어에서, 웹 브라우져에서 동작하는 프로그램은 해당 프로그램이 로딩된 위치에 있는 리소스만 접근이 가능하다. (http냐 https냐 와 같은 프로토콜과 호출 포트도 정확하게 일치해야 한다.)



웹사이트 sitea.com에서 자바스크립트를 로딩한 후에, 이 스크립트에서 api.my.com에 있는 API를 XMLHTTPRequest를 통해서 호출하고자 하면, Same origin policy에 의해서 호출 에러가 난다.

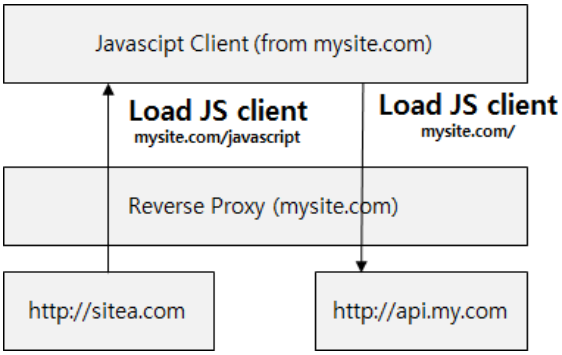
- 해결하는 방법
  - 인프라 측면에서 **Proxy를 사용**
  - **JSONP와 CORS** (Cross Origin Resource Sharing)

A. Proxy를 이용하는 방식

Same origin policy의 문제는 **API 서버와 Javascript가 호스팅 되는 서버의 URL이 다르기 때문에 발생**하는 문제

- 앞단에 Reverse Proxy등을 넣어서, 전체 URL을 같게 만들어 준다.

1. sitea.com과 api.my.com 앞에 reverse proxy를 넣고, reverse proxy의 주소를 http://mysite.com 으로 세팅한다.
2. mysite.com/javascript로 들어오는 요청은 sitea.com으로 라우팅 하고, mysite.com/의 다른 URL로 들어오는 요청은 api.my.com으로 라우팅한다.



javascript 가 로딩된 사이트도 mysite.com이 되고, javascript에서 호출하고자 하는 api URL도 mysite.com 이 되기 때문에, Same Origin Policy에 위배되지 않는다.

- **자사의 웹사이트를 서비스하는 경우에만 가능하다**. (타사의 사이트를 Reverse Proxy뒤에 놓기는 쉽지 않다.) 그래서 자사의 서비스용 API를 만드는데는 괜찮지만, 파트너사나 일반 개발자에게 자바스크립

트용 REST API를 오픈하는 경우에는 적절하지 않다.

B. 특정 사이트에 대한 접근 허용 방식

i. CORS 방식중, 가장 간단하고 쉬운 방식

ii. API 서버의 설정에서 모든 소스에서 들어오는 API 호출을 허용하도록 한다

iii. api.my.com 이라는 API 서비스를 제공할 때, 이 API를 어느 사이트에서라도 로딩된 자바스크립트라도 호출이 가능하게 하는 것이다.

iv. HTTP로 API를 호출하였을 경우 HTTP Response에 응답을 주면서 HTTP Header에 Request Origin (요청을 처리해줄 수 있는 출처)를 명시하는 방식이다.

v. **api.my.com에서 응답 헤더에 Access-Control-Allow-Origin: sitea.com 와 같이 명시해주면 sitea.com에 의해서 로딩된 자바스크립트 클라이언트 요청에 대해서만 api.my.com가 요청을 처리해준다**.

vi. * 로 해주면, request origin에 상관없이 사이트에서 로딩된 자바스크립트 요청에 대해서 처리를 해준다. Access-Control-Allow-Origin: *

C. Pre-flight를 이용한 세세한 CORS 통제

i. REST 리소스 (URL)당 섬세한 CORS 통제가 필요한 경우

ii. 이 방식은 REST 리소스를 호출하기 전에, 웹 브라우져가 HTTP OPTIONS 요청을 보내면 해당 REST Resource에 대해서 가능한 CORS 정보를 보내준다. (접근이 허용된 사이트, 접근이 허용된 메서드 등)

iii. 웹브라우져에서는 XMLHTTPRequest를 특정 URL로 요청하기 전에 먼저 HTTP Options를 호출한다.

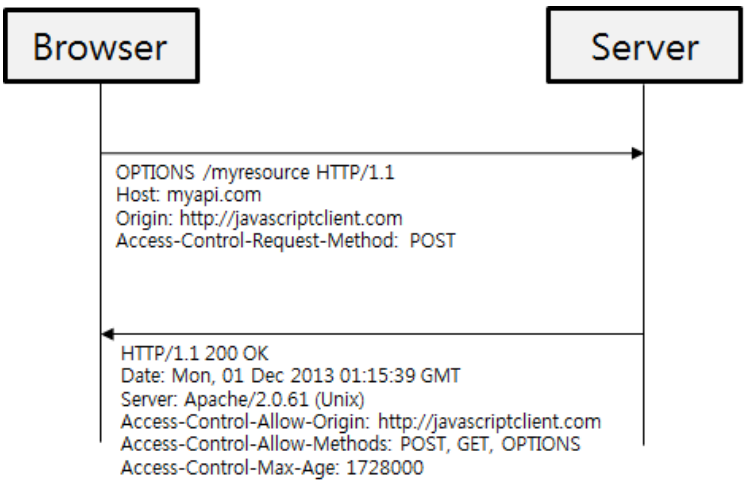
iv. 서버는 해당 URL을 접근할 수 있는 Origin URL과 HTTP Method를 리턴해준다. 이를 pre-flight 호출이라고 하는데, 이 정보를 기반으로 브라우져는 해당 URL에 XMLHTTPRequest를 보낼 수 있다.

2. 브라우져는 http://javascriptclient.com에서 로딩된 자바스크립트로 REST 호출을 하려고 한다.

   1. HTTP OPTION 메서드로 아래 첫번째 화살표와 같이 /myresource URL에 대해서 pre-flight 호출을 보낸다. 여기에는 Origin Site URL과 허가를 요청하는 HTTP 메서드등을 명시한다.

   2. 서버는 이 URL에 대한 접근 권한을 리턴하는데, 두번째 화살표와 같이 CORS접근이 가능한 Origin 사이트를 http://javascriptclient.com으로 리턴하고 사용할 수 있는 메서드는 POST,GET,OPTIONS 3개로 정의해서 리턴한다.

3. 이 pre-flight 호출은 Access-Control-Max-Age에 정의된 1728000초 동안 유효하다.



이러한 CORS 설정은 API 호출 코드에서 직접 구현할 수도 있지만, 그 보다는 앞단에서 로드 밸런서 역할을 하는 HA Proxy나 nginx와 같은 reverse proxy에서 설정을 통해서 간단하게 처리가 가능하다. 만약에 API단에서 구현이 필요하다하더라도 HTTP Header를 직접 건드리지 말고, Spring 등의 프레임웍에서 이미 CORS 구현을 지원하고 있으니 프레임웍을 통해서 간단하게 구현하는 것을 권장한다.

2. API access Token에 대한 인증 처리

자바스크립트 클라이언트는 모바일 앱이나, 서버와 같은 다른 API 클라이언트와 비교해서 **api access token을 안전하게 저장할 수 있는 방법이 없기** 때문에, 이 API access token에 대해서 다른 관리 방식이 필요하다.

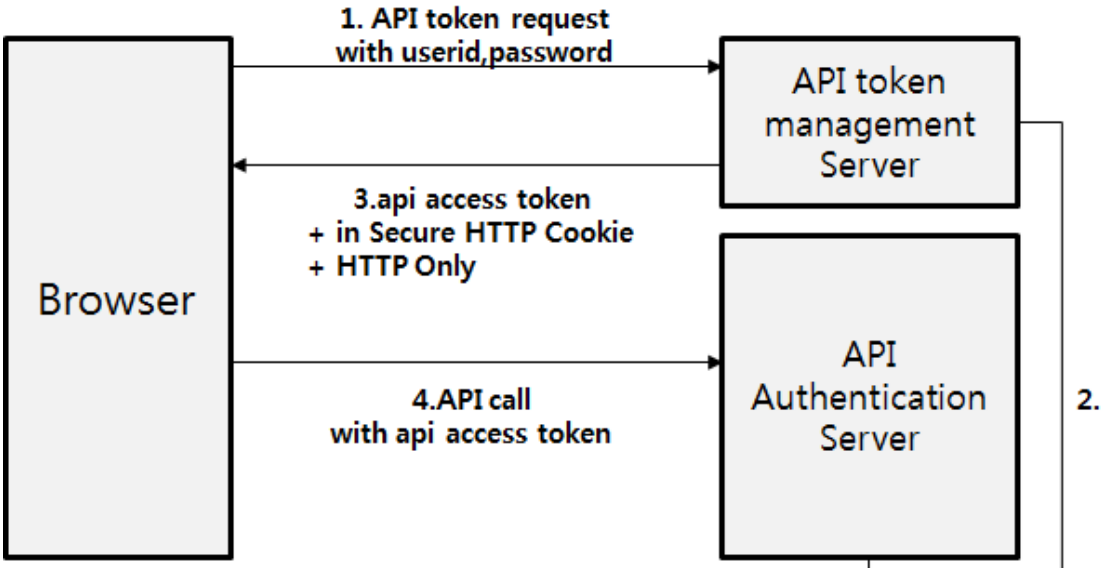
A. api access token을 Secure Cookie를 통해서 주고받는다.

- api access token을 서버에서 발급하여 자바스크립트 클라이언트로 리턴할 때, HTTP body에 리턴하는 것이 아니라 Secure Cookie에 넣어서 리턴한다.
- Secure Cookie
  - HTTP 프로토콜을 통해서는 전송이 불가능하고 항상 HTTPS를 통해서만 전송이 가능하다.
  - HTTP_ONLY라는 옵션을 쿠키에 추가
  - Cookie를 자바스크립트를 통해서 읽거나 조작할 수 없다. 단지 브라우저가 서버로 요청을 보낼 때, 브라우저에 의해서 자동으로 Cookie가 전송된다.

이 두 가지 방법을 쓰면 최소한 자바스크립트 소스코드 분석이나 네트워크 프로토콜 감청을 통한 api access token을 방어할 수 있다

B. api access token은 해당 세션에서만 유효하도록 한다.

- HTTP Session과 같이 특정 IP와 시간내에서만 api access token이 유효하도록 하는 방식이다.
- Access token을 발급할 때, access token을 요청한 클라이언트의 IP와 클라이언트의 Origin을 같이 저장해놓고, 발급할때 유효시간(Expire time)을 정해놓는다. (20분 등으로).
- 다음 access token을 이용해서 API가 호출 될 때 마다 IP와 Origin을 확인하고, access token이 유효시간 (Expire time) 이내에 호출을 하면 다시 연장해준다.(+20분을 다시 추가해준다) 만약에 브라우져에서 일정 시간동안 (20분) API를 호출하지 않았으면 API access token은 폐기되고 다시 access token을 발급 받도록 한다.



API access Token Table

Access Token	Origin URL	Client IP	Expire time
9axqo1921	javascriptclient.com	210.129.98.xx	2013-09-01 12:20:00
223as21o2	javascriptclient.com	223.62.182.xx	2013-09-01 12:12:20
:	:	:	:

- 모든 통신을 HTTPS를 이용한다.

1. 자바스크립트 클라이언트가 user id와 password를 보내서 사용자 인증과 함께, API access token을 요청한다.
   - ◆ HTTPS를 사용한다하더라도 Man in middle attack에 의해서 password가 노출 될 수 있기 때문에, 앞에서 언급한 Digest access Authentication 등의 인증 메커니

즘을 활용하여 가급적이면 password를 직접 보내지 않고 인증을 하는 것이 좋다.

2. 서버에서 사용자 인증이 끝나면 api access token을 발급하고 이를 내부 token store에 저장한다.
   - ◆ 앞에서 설명한 origin url, ip, expire time등을 저장한다.
   - ◆ 실제 시스템 디자인은 웹 클라이언트용과 일반 서버/모바일 앱등을 위한 api access token 정보도 같이 저장해서 두가지 타입을 access token에 대해서 지원하도록 하는 것이 좋다.
3. 생성된 토큰은 Secure Cookie와 HTTP Only 옵션을 통해서 브라우져에게로 전달된다.
4. 브라우져의 자바스크립트 클라이언트에서는 API를 호출할 때 이 api access token이 secure cookie를 통해서 자동으로 서버에 전송되고, 서버는 이 api access token을 통해서 접근 인증 처리를 하고 api server로 요청을 전달하여 처리하도록 한다.