{ REST API }

Representational State Transfer API

Rest API ?

> **REST**(Representational State Transfer): 자원을 **이름**으로 구분하여 해당 자원의 **상태**를 주고받는 모든 것
> **API**(application programming interface): 애플리케이션 프로그래밍 인터페이스

즉, REST API는 컴퓨터와 컴퓨터, 서버와 클라이언트 등 다양한 애플레이케이션 연결 구조에서 프로그래밍 인터페이스 규격에 맞춰 자원의 이름으로 구분하여 자원의 상태를 주고받는 행위를 말합니다.

● 혹시 RESTFUL API라는 말을 들어보셨나요?
RESTFUL이란 REST의 원리를 따르는 시스템으로, REST를 사용했다 하여 모두가 RESTFUL 하지는 않습니다.
즉 REST API의 설계 규칙을 명확하게 지킨 시스템만이 RESTFUL 하다고 말할 수 있습니다.

1. REST API의 탄생

REST는 Representational State Transfer라는 용어의 약자로서 2000년도에 로이 필딩 (Roy Fielding)의 박사학위 논문에서 최초로 소개되었습니다. 로이 필딩은 HTTP의 주요 저자 중 한 사람으로 그 당시 웹(HTTP) 설계의 우수성에 비해 제대로 사용되어지지 못하는 모습에 안타까워하며 웹의 장점을 최대한 활용할 수 있는 아키텍처로써 REST를 발표했다고 합니다.

2. REST 구성

쉽게 말해 REST API는 다음의 구성으로 이루어져있습니다. 자세한 내용은 밑에서 설명하도록 하겠습니다.

- 자원(RESOURCE) - URI
- 행위(Verb) - HTTP METHOD
- 표현(Representations)

3. REST 의 특징

1) Uniform (유니폼 인터페이스)

Uniform Interface는 URI로 지정한 리소스에 대한 조작을 통일되고 한정적인 인터페이스로 수행하는 아키텍처 스타일을 말합니다.

2) Stateless (무상태성)

REST는 무상태성 성격을 갖습니다. 다시 말해 작업을 위한 상태정보를 따로 저장하고 관리하지 않습니다. 세션 정보나 쿠키정보를 별도로 저장하고 관리하지 않기 때문에 API 서버는 들어오는 요청만을 단순히 처리하면 됩니다. 때문에 서비스의 자유도가 높아지고 서버에서 불필요한 정보를 관리하지 않음으로써 구현이 단순해집니다.

3) Cacheable (캐시 가능)

REST의 가장 큰 특징 중 하나는 HTTP라는 기존 웹표준을 그대로 사용하기 때문에, 웹에서 사용하는 기존 인프라를 그대로 활용이 가능합니다. 따라서 HTTP가 가진 캐싱 기능이 적용 가능합니다.

4) Self-descriptiveness (자체 표현 구조)

REST의 또 다른 큰 특징 중 하나는 REST API 메시지만 보고도 이를 쉽게 이해 할 수 있는 자체 표현 구조로 되어 있다는 것입니다.

5) Client - Server 구조

REST 서버는 API 제공, 클라이언트는 사용자 인증이나 컨텍스트(세션, 로그인 정보)등을 직접 관리하는 구조로 각각의 역할이 확실히 구분되기 때문에 클라이언트와 서버에서 개발해야 할 내용이 명확해지고 서로간의 의존성이 줄어들게 됩니다.

6) 계층형 구조

REST 서버는 다중 계층으로 구성될 수 있으며 보안, 로드 밸런싱, 암호화 계층을 추가해 구조상의 유연성을 둘 수 있고 PROXY, 게이트웨이 같은 네트워크 기반의 중간매체를 사용할 수 있게 합니다.

4. REST API 디자인 가이드

REST API 설계 시 가장 중요한 항목은 다음의 2가지로 요약할 수 있습니다.

첫 번째, URI는 정보의 자원을 표현해야 한다.
두 번째, 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE)로 표현한다.

다른 것은 다 잊어도 위 내용은 꼭 기억하시길 바랍니다.

4-1. REST API 중심 규칙
1) URI는 정보의 자원을 표현해야 한다. (리소스명은 동사보다는 명사를 사용)

```
GET /members/delete/1
```

위와 같은 방식은 REST를 제대로 적용하지 않은 URI입니다. URI는 자원을 표현하는데 중점을 두어야 합니다. delete와 같은 행위에 대한 표현이 들어가서는 안됩니다.

2) 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)로 표현
위의 잘못 된 URI를 HTTP Method를 통해 수정해 보면

```
DELETE /members/1
```

으로 수정할 수 있겠습니다.
회원정보를 가져올 때는 GET, 회원 추가 시의 행위를 표현하고자 할 때는 POST METHOD를 사용하여 표현합니다.

회원정보를 가져오는 URI

```
GET /members/show/1      (x)
GET /members/1           (o)
```

회원을 추가할 때

```
GET /members/insert/2    (x)  - GET 메서드는 리소스 생성에 맞지 않습니다.
POST /members/2          (o)
```

[참고]HTTP METHOD의 알맞은 역할

POST, GET, PUT, DELETE 이 4가지의 Method를 가지고 CRUD를 할 수 있습니다.

METHOD	역할
POST	POST를 통해 해당 URI를 요청하면 리소스를 생성합니다.
GET	GET를 통해 해당 리소스를 조회합니다. 리소스를 조회하고 해당 도큐먼트에 대한 자세한 정보를 가져온다.
PUT	PUT를 통해 해당 리소스를 수정합니다.
DELETE	DELETE를 통해 리소스를 삭제합니다.

다음과 같은 식으로 URI는 자원을 표현하는 데에 집중하고 행위에 대한 정의는 HTTP METHOD를 통해 하는 것이 REST한 API를 설계하는 중심 규칙입니다.

4-2. URI 설계 시 주의할 점

1) 슬래시 구분자(/)는 계층 관계를 나타내는 데 사용

```
http://restapi.example.com/houses/apartments
http://restapi.example.com/animals/mammals/whales
```

2) URI 마지막 문자로 슬래시(/)를 포함하지 않는다.

URI에 포함되는 모든 글자는 리소스의 유일한 식별자로 사용되어야 하며 URI가 다르다는 것은 리소스가 다르다는 것이고, 역으로 리소스가 다르면 URI도 달라져야 합니다. REST API는 분명한 URI를 만들어 통신을 해야 하기 때문에 혼동을 주지 않도록 URI 경로의 마지막에는 슬래시(/)를 사용하지 않습니다.

```
http://restapi.example.com/houses/apartments/  (X)
http://restapi.example.com/houses/apartments   (0)
```

3) 하이픈(-)은 URI 가독성을 높이는데 사용

URI를 쉽게 읽고 해석하기 위해, 불가피하게 긴 URI경로를 사용하게 된다면 하이픈을 사용해 가독성을 높일 수 있습니다.

4) 밑줄(_)은 URI에 사용하지 않는다.

글꼴에 따라 다르긴 하지만 밑줄은 보기 어렵거나 밑줄 때문에 문자가 가려지기도 합니다.

이런 문제를 피하기 위해 밑줄 대신 하이픈(-)을 사용하는 것이 좋습니다.(가독성)

5) URI 경로에는 소문자가 적합하다.
URI 경로에 대문자 사용은 피하도록 해야 합니다. 대소문자에 따라 다른 리소스로 인식하게 되기 때문입니다. RFC 3986(URI 문법 형식)은 URI 스키마와 호스트를 제외하고는 대소문자를 구별하도록 규정하기 때문이지요.

RFC 3986 is the URI (Unified Resource Identifier) Syntax document

6) 파일 확장자는 URI에 포함시키지 않는다.

http://restapi.example.com/members/soccer/345/photo.jpg (X)

REST API에서는 메시지 바디 내용의 포맷을 나타내기 위한 파일 확장자를 URI 안에 포함시키지 않습니다. Accept header를 사용하도록 합시다.

GET / members/soccer/345/photo HTTP/1.1 Host: restapi.example.com Accept: image/jpg

4-3. 리소스 간의 관계를 표현하는 방법
REST 리소스 간에는 연관 관계가 있을 수 있고, 이런 경우 다음과 같은 표현방법으로 사용합니다.

/리소스명/리소스 ID/관계가 있는 다른 리소스명

ex) GET : /users/{userid}/devices (일반적으로 소유 'has'의 관계를 표현할 때)

만약에 관계명이 복잡하다면 이를 서브 리소스에 명시적으로 표현하는 방법이 있습니다. 예를 들어 사용자가 '좋아하는' 디바이스 목록을 표현해야 할 경우 다음과 같은 형태로 사용될 수 있습니다.

GET : /users/{userid}/likes/devices (관계명이 애매하거나 구체적 표현이 필요할 때)

4-4. 자원을 표현하는 Colllection과 Document

Collection과 Document에 대해 알면 URI 설계가 한 층 더 쉬워집니다. DOCUMENT는 단순히 문서로 이해해도 되고, 한 객체라고 이해하셔도 될 것 같습니다. 컬렉션은 문서들의 집합, 객체들의 집합이라고 생각하시면 이해하시는데 좀더 편하실 것 같습니다. 컬렉션과 도큐먼트는 모두 리소스라고 표현할 수 있으며 URI에 표현됩니다. 예를 살펴보도록 하겠습니다.

```
http://restapi.example.com/sports/soccer
```

위 URI를 보시면 sports라는 컬렉션과 soccer라는 도큐먼트로 표현되고 있다고 생각하면 됩니다. 좀 더 예를 들어보자면

```
http://restapi.example.com/sports/soccer/players/13
```

sports, players 컬렉션과 soccer, 13(13번인 선수)를 의미하는 도큐먼트로 URI가 이루어지게 됩니다. 여기서 중요한 점은 컬렉션은 복수로 사용하고 있다는 점입니다. 좀 더 직관적인 REST API를 위해서는 컬렉션과 도큐먼트를 사용할 때 단수 복수도 지켜준다면 좀 더 이해하기 쉬운 URI를 설계할 수 있습니다.

5. HTTP 응답 상태 코드

마지막으로 응답 상태코드를 간단히 살펴보도록 하겠습니다. 잘 설계된 REST API는 URI만 잘 설계된 것이 아닌 그 리소스에 대한 응답을 잘 내어주는 것까지 포함되어야 합니다. 정확한 응답의 상태코드만으로도 많은 정보를 전달할 수가 있기 때문에 응답의 상태코드 값을 명확히 돌려주는 것은 생각보다 중요한 일이 될 수도 있습니다. 혹시 200이나 4XX관련 특정 코드 정도만 사용하고 있다면 처리 상태에 대한 좀 더 명확한 상태코드 값을 사용할 수 있기를 권장하는 바입니다.

상태코드에 대해서는 몇 가지만 정리하도록 하겠습니다.

상태코드	
200	클라이언트의 요청을 정상적으로 수행함
201	클라이언트가 어떠한 리소스 생성을 요청, 해당 리소스가 성공적으로 생성됨(POST를 통한 리소스 생성 작업 시)

상태코드	
400	클라이언트의 요청이 부적절 할 경우 사용하는 응답 코드
401	클라이언트가 인증되지 않은 상태에서 보호된 리소스를 요청했을 때 사용하는 응답 코드
	(로그인 하지 않은 유저가 로그인 했을 때, 요청 가능한 리소스를 요청했을 때)
403	유저 인증 상태와 관계 없이 응답하고 싶지 않은 리소스를 클라이언트가 요청했을 때 사용하는 응답 코드
	(403 보다는 400이나 404를 사용할 것을 권고, 403체가 리소스가 존재한다는 뜻이기 때문에)
404(Not Found, 찾을 수 없음)	서버가 요청한 페이지(Resource)를 찾을 수 없다. 예를 들어 서버에 존재하지 않는 페이지에 대한 요청이 있을 경우 서버는 이 코드를 제공한다.
405	클라이 언트가 요청한 리소스에서는 사용 불가능한 Method를 이용했을 경우 사용하는 응답 코드

상태코드	
301	클라이언트가 요청한 리소스에 대한 URI가 변경 되었을 때 사용하는 응답 코드
	(응답 시 Location headder에 변경된 URI를 적어 줘야 합니다.)
500	서버에 문제가 있을 경우 사용하는 응답코드

■ 테스트

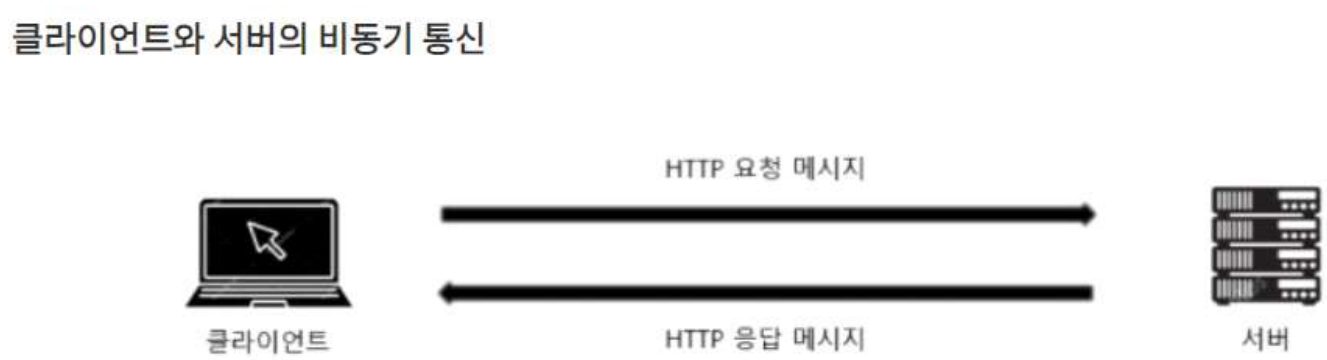
■ {JSON} Placeholder 사이트

https://jsonplaceholder.typicode.com

무료로 가짜 API를 사용해 무료로 각종 테스트를 진행 할 수 있는 서비스 제공.

6. 비동기 처리(여러가지 Annotation)
스프링에서 비동기 처리를 하는 경우 @RequestBody , @ResponseBody를 사용합니다.

비동기 처리를 위해 이 어노테이션들은 어떻게 작동할까요?



클라이언트에서 서버로 통신하는 메시지를 요청(request) 메시지라고 하며, 서버에서 클라이언트로 통신하는 메시지를 응답(response) 메시지라고 합니다.

웹에서 화면전환(새로고침) 없이 이루어지는 동작들은 대부분 비동기 통신으로 이루어집니다.

비동기통신을 하기위해서는 클라이언트에서 서버로 요청 메세지를 보낼 때, 본문에 데이터를 담아서 보내야 하고, 서버에서 클라이언트로 응답을 보낼때에도 본문에 데이터를 담아서 보내야 하는데, 이 본문이 바로 body입니다.

즉, 요청본문 requestBody, 응답본문 responseBody 을 담아서 보내야 하는 것이죠.

이때 본문에 담기는 데이터 형식은 여러가지 형태가 있겠지만 가장 대표적으로 사용되는 것이 JSON입니다.

즉, 비동기식 클라이언트-서버 통신을 위해 JSON 형식의 데이터를 주고받는 것입니다.

스프링 MVC에서도 클라이언트에서 전송한 xml데이터나 json 등등 데이터를 컨트롤러에서 DOM객체나 자바객체로 변환해서 송수신할 수 있습니다..

@RequestBody 어노테이션과 @ResponseBody 어노테이션이 각각 HTTP요청 바디를 자바객체로 변환하고 자바객체를 다시 HTTP 응답 바디로 변환해 줍니다.

@RestController는 @Controller와는 다르게 리턴값에 자동으로 @ResponseBody가 붙게되어 별도 어노테이션을 명시해주지 않아도 HTTP 응답데이터(body)에 자바 객체가 매핑되어 전달 됩니다.

@Controller인 경우에 바디를 자바객체로 받기 위해서는 @ResponseBody 어노테이션을 반드시 명시해 주어야 합니다.

@RequestBody / @ResponseBody 정리.
클라이언트에서 서버로 필요한 데이터를 요청하기 위해 JSON 데이터를 요청 본문에 담아서 서버로 보내면, 서버에서는 @RequestBody 어노테이션을 사용하여 HTTP 요청 본문에 담긴 값들을 자바객체로 변환시켜, 객체에 저장합니다.

서버에서 클라이언트로 응답 데이터를 전송하기 위해 @ResponseBody 어노테이션을 사용하여 자바 객체를 HTTP 응답 본문의 객체로 변환하여 클라이언트로 전송한다.

Spring boot로 Rest API 만들기

프로젝트 시작하기

○ 기존 게시판 서비스 중 CRUD를 REST API 로 생성

■ TestEntity

```
public class TestEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String test;
}
```

■ TestForm

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class TestForm {
    private Long id;
    private String memo;
}
```

■ Repository - ApiTestRepository

■ ApiTestController / ApiTestService 분리

```
@RestController
public class ApiTestController {

    @Autowired
    ApiTestRepository apiTestRepository;
```

```
    @GetMapping("/api/test")
    public List<TestEntity> test(){
        return apiTestRepository.findAll();
    }

    @GetMapping("/api/test/{id}")
    public ResponseEntity<TestEntity> getOne(@PathVariable Long id){
        TestEntity apiTest = apiTestRepository.findById(id).orElse(null);
        if(apiTest == null){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
        }
        return ResponseEntity.status(HttpStatus.OK).body(apiTest);
    }

    @PostMapping("/api/test")
    public TestEntity input(@RequestBody TestForm dto){
        TestEntity apiTest = new TestEntity();
        apiTest.setTest(dto.getTest());
        return apiTestRepository.save(apiTest);
    }
```

```
{
    "id" : 11
    "memo" : "신규자료"
}
```

```
    @PatchMapping("/api/test/{id}")
    private ResponseEntity<TestEntity> patch(@PathVariable Long id,
                                @RequestBody TestForm form){
        TestEntity inputEntity = new TestEntity();
        inputEntity.setId(form.getId());
        inputEntity.setTest(form.getTest());
```

```
        TestEntity apiTest = apiTestRepository.findById(id).orElse(null);
        if(apiTest==null || id != inputEntity.getId()){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
        }
        TestEntity updated = apiTestRepository.save(inputEntity);
        return ResponseEntity.status(HttpStatus.OK).body(updated);
    }


    @DeleteMapping("/api/test/{id}")
    public ResponseEntity<TestEntity> deleteOne(@PathVariable Long id){
        TestEntity apiTest = apiTestRepository.findById(id).orElse(null);
        if(apiTest == null){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
        }
        apiTestRepository.deleteById(id);
        return ResponseEntity.status(HttpStatus.OK).body(null);

    }
}
```