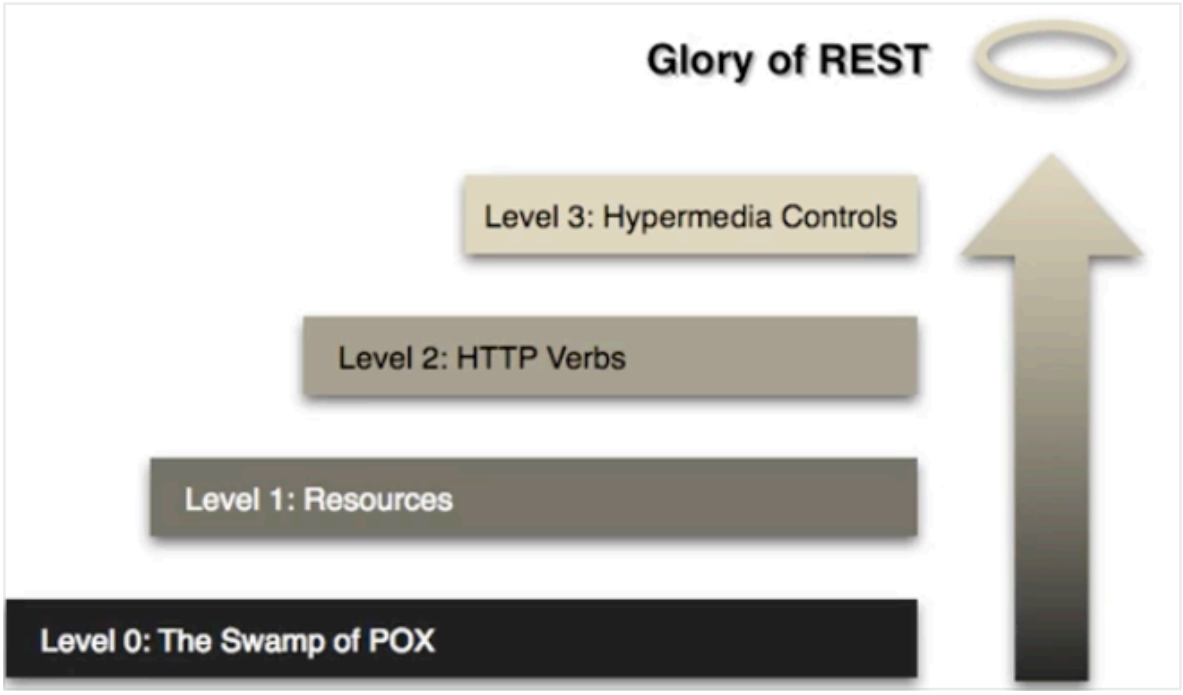

Richardson Maturity Model

Rest API는 Richardson의 성숙도 모델을 통해 단계를 구분할 수 있다.



Level 0

- 리소스를 웹 서비스 형태로 제공해서 단순히 URI만 맵핑한 형태
- HTTP 메소드를 통해 (GET / POST / PUT / DELETE) 목적을 구분할 수 있음에도 URI에 불필요한 동사가 함께 사용 되는 상태이다.
  - http://server/getPosts
  - http://server/deletePosts
  - http://server/doThis

Level 1

- 외부에 공개하고자하는 리소스에 대해 의미있고 적절한 URI로 표현된 형태
- 일정한 패턴을 가지고 작성되었지만 HTTP 메소드와 작업의 형태가 일치하지않은 상태이다.
  - http://server/accounts
  - http://server/accounts/10

Level 2

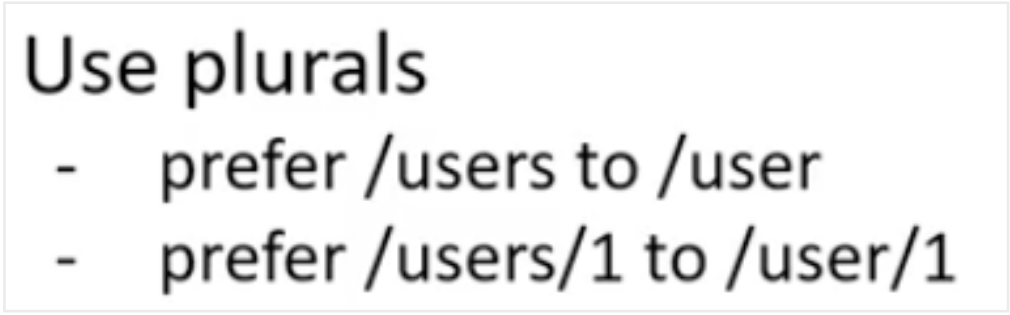
- Level 1 + HTTP Method
- 제공하고자 하는 리소스를 용도와 상태에 맞춰 HTTP Method를 사용한 형태
- CRUD 형태에 맞춰 적절하게 HTTP Method를 사용한 상태로 같은 URI를 통해 다른 서비스를 제공할 수 있다.

Level 3

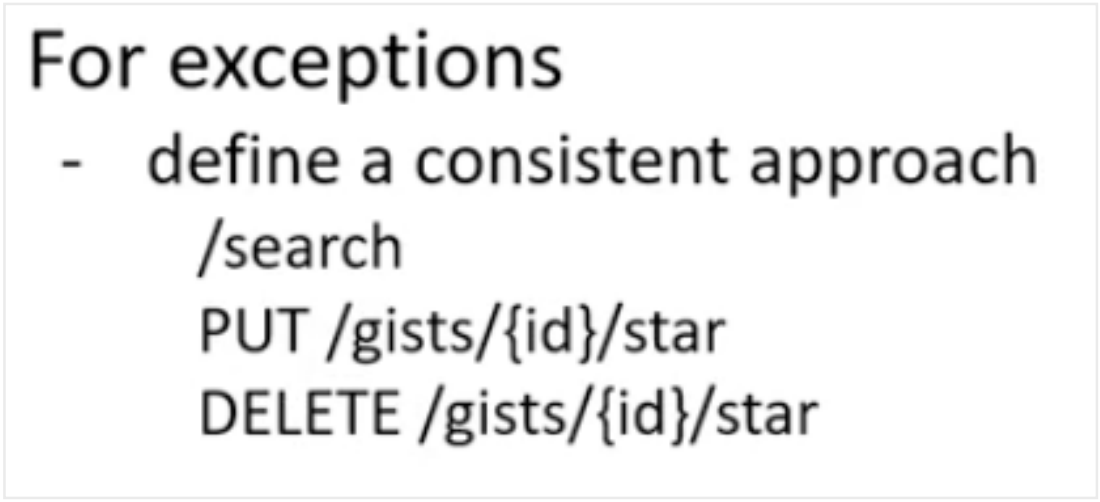
- Level 2 + HTTP HATEOAS
- 다음 작업으로 어떤것을 할 수 있는지, 그 작업을 하기 위해 다뤄져야할 리소스 URL에 어떤 것이 있는지, 어떤 액션이 있는지 알려주는 기능이 포함되어있다.
- 클라이언트측에서 서버가 제공하는 서비스를 일일이 찾지 않아도 된다.
- 최소한의 진입점 endpoint를 통해 서버의 다음 uri 값을 알 수 있다.

REST API 설계 고려 사항

1. 개발자 중심 설계 보다는 해당 API를 사용하는 소비자 입장에서 간단 명료하고 직관적인 API 형태로 설계 되어야한다.
2. HTTP 메소드와 request, response 타입 header 값과 같이 HTTP의 장점을 최대한 살려 설계 해야한다.
3. 최소한 레벨 2 모델이 가진 특성을 가지고 각 리소스 별로 적절한 HTTP의 메소드를 제공해야한다.
4. 개발자가 제공하는 URI는 보안과 관련된 정보를 포함해선 안된다. (스프링 시큐리티 .. 토큰 .. 헤더 값 ..)
5. 복수 형태의 URI를 사용한다. 특정 하고 싶다면 다음 덱스를 사용한다. 동사 형태가 아닌 명사 형태로 사용한다.



6. 일관된 접근 엔드 포인트를 사용해야한다.



LEVEL2 HTTP Verbs

도메인 클래스 생성

간단한 도메인을 생성하고 서비스 로직을 작성한다.
로직은 간단히 전체 회원 조회, 특정 회원 조회/저장/삭제/수정 기능이 존재한다.
(여기서 중요한건 로직이 아닌 흐름이기 때문에 자세한 서비스 코드는 생략)

GET HTTP Method

우리는 REST API를 구현할 예정이기 때문에 컨트롤러를 생성하고 @RestController 어노테이션을 붙여주자

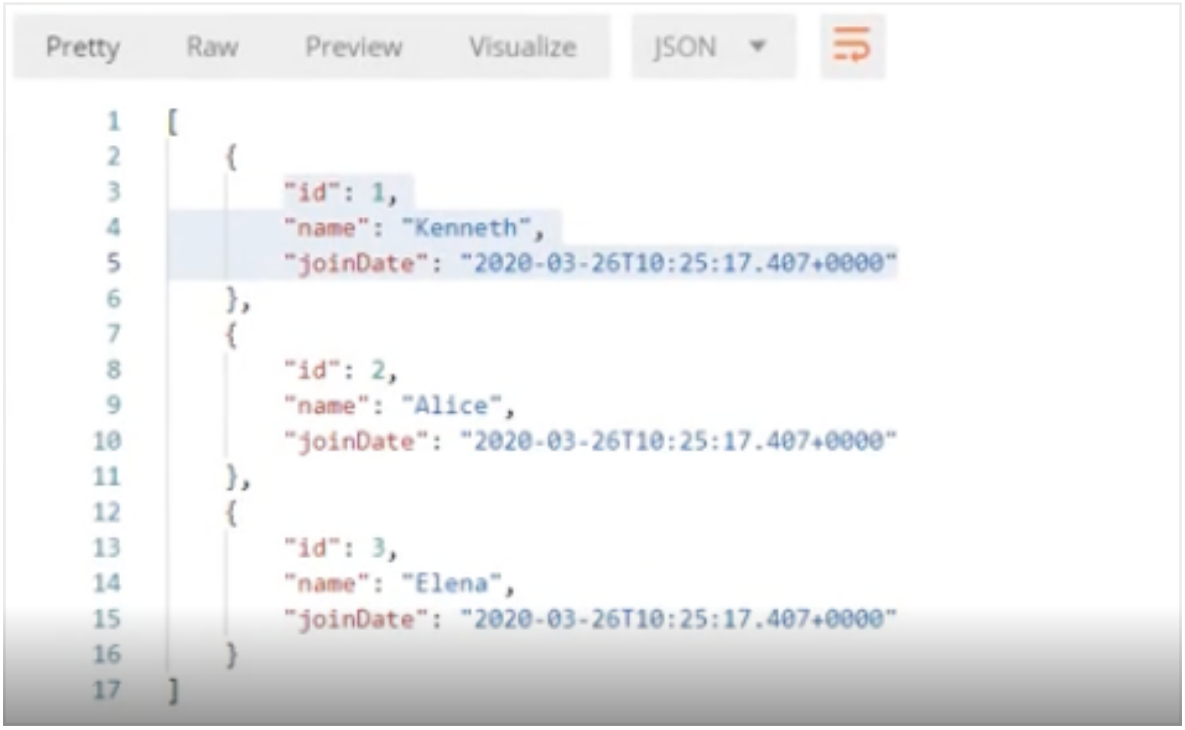
```java
@GetMapping("/users")
public List<User> retrieveAllUsers() {
    return service.findAll();
}

// GET /users/1 or /users/10 -> String
@GetMapping("/users/{id}")
public User retrieveUser(@PathVariable int id) {
    return service.findOne(id);
}
```

GetMapping을 통해 단순히 전체 유저 정보와 특정 유저 정보를 반환하는 URI를 설정했다.

postmen 조회 결과

http://localhost:8080/users



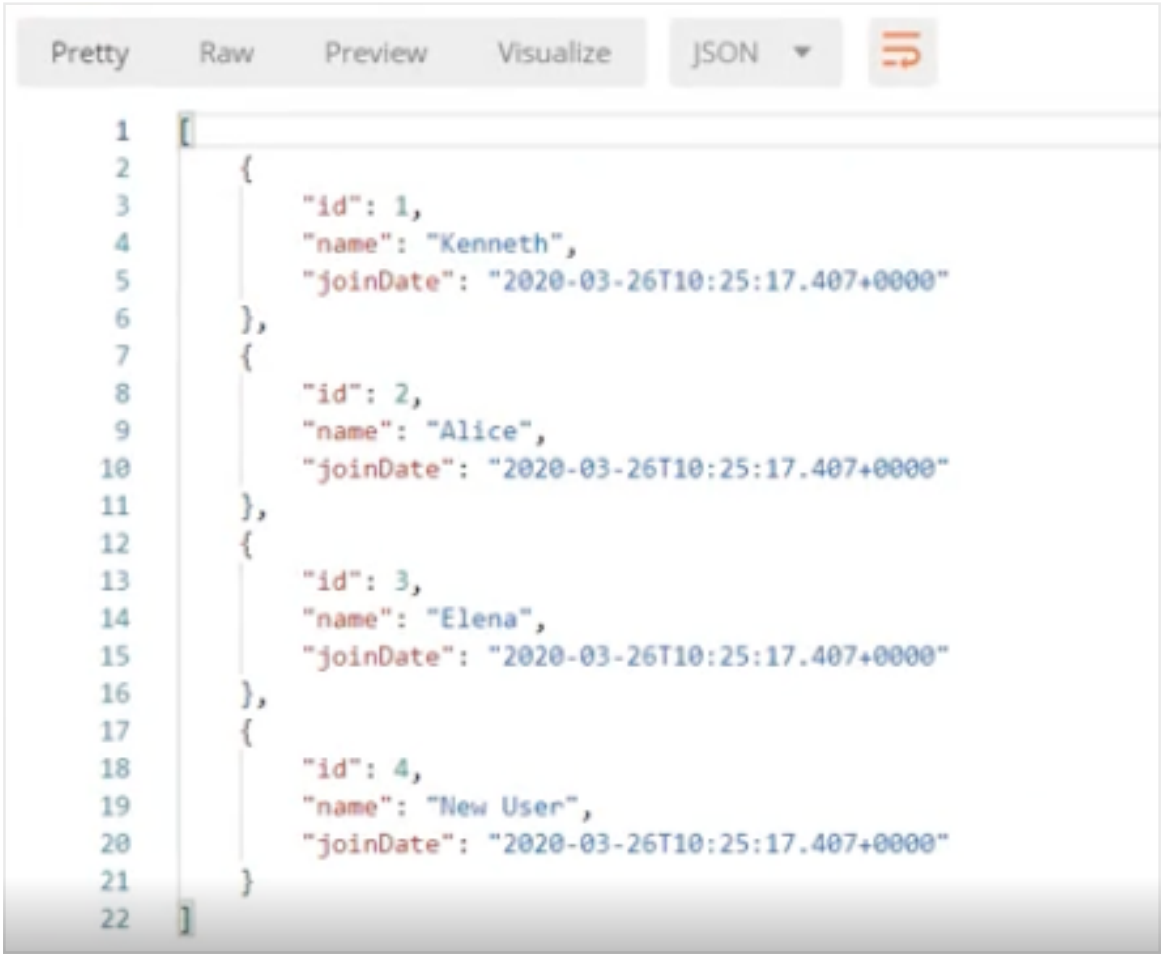
POST HTTP Method

```java
@PostMapping("/users")
public void createUser(@RequestBody User user) {
    User savedUser = service.save(user);
}
```

클라이언트로부터 POST와 PUT과 같은 HTTP 메소드를 통해 Json, XML 타입의 오브젝트 형태 데이터를 받는다면 매개 변수 타입에 @RequestBody를 선언해야한다.

postmen 조회 결과

http://localhost:8080/users



서버는 클라이언트에게 GET, POST 모두 동일한 URI를 통해 요청을 받았지만 전혀 다른 동작을 하는것을 확인할 수 있다.

문제점 - GET/POST 동일한 Status Code 반환

HTTP 메소드를 통해 동일한 URI를 사용하지만 다른 기능을 하도록 잘 구성 했지만 두 가지 요

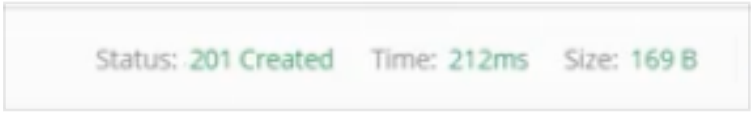
청 결과가 모두 200OK를 반환하고있다.

RESTful 스럽게 API를 작성하고싶다면 HTTP Status Code 역시 깔끔하게 정리해야한다.
POST 매핑을 통해 새로운 유저 등록을 성공할 경우 201 Created 를 반환 받을 수 있게 변경해 보자

```
@PostMapping("/users")
public ResponseEntity createUser(@Valid @RequestBody User user) {
    User savedUser = service.save(user);

    URI location = ServletUriComponentsBuilder.fromCurrentRequest() ServletUriComponentsBuilder
                    .path("/{id}") UriComponentsBuilder
                    .buildAndExpand(savedUser.getId()) UriComponents
                    .toUri();

    return ResponseEntity.created(location).build();
}
```



흐름
1. ServletUriComponentsBuilder 클래스의 fromCurrentRequest() 메소드를 사용해 현재 request 정보를 받아온다.
2. path() 메서드를 통해 URI 경로를 추가한다. (여기서는 POST 매핑을 통해 받은 새로운 유저 아이디가 인자 값으로 들어간다.)
3. toUri() 메서드를 통해 새로운 URI를 생성한다.
4. ResponseEntity에 미리 정의된 created(201) 값에 URI를 넣어 반환한다.

결과

원래 사용자 상세 보기 API를 호출하기 위해선 사용자의 id 값을 알고, URI에 포함을 시킨 후 전달해야하는데
생성된 id는 서버에서 제공하기 때문에 클라이언트에서 이를 알기 위해 서버에게 한 번 더 요청을 해야했다.

하지만 위 방법을 통해 POST 메소드의 실행 결과값을 전달 받게 되면 그만큼 네트워크 트래픽이 감소되고 효율적인 어플리케이션을 완성할 수 있다.

ResponseEntity<T>?

- ResponseEntity<T> 클래스는 스프링에서 HTTP 응답을 생성할 때 사용되는 클래스로 HttpEntity 클래스를 상속 받는다.

- ResponseEntity는 응답 본문과 HTTP 상태 코드, 응답 헤더 등을 포함하는 객체를 생성하여 클라이언트에 반환하는 역할을 한다.

- 컨트롤러 메서드에서 반환되는 객체를 HTTP 응답으로 변환할 때, 응답 본문과 HTTP 상태 코드, 응답 헤더 등을 더욱 세밀하게 제어할 수 있다.

ServletUriComponentsBuilder?
- ServletUriComponentsBuilder 는 현재 request의 URI를 생성하는데 사용되는 클래스로 UriComponentsBuilder 클래스를 상속 받는다.

- 서블릿 기반 애플리케이션에서 URI를 생성하는 데 사용할 수 있는 다양한 메서드를 제공한다.

- 현재 request의 스키마, 호스트, 포트, 경로, 쿼리 파라미터 등의 정보를 추출할 수 있다.

- URI를 생성하거나 새로운 URI를 생성하는 데 사용된다.

문제점 - 유효하지 않은 URI 값에도 200 OK 반환

현재 등록된 유저 정보는 POST를 통해 생성한 유저 한 명이지만 다음과 같은 URI에도 HTTP 상태 코드가 200 OK 값을 반환한다.

http://localhost8080/users/100

물론 클라이언트가 반환 받은 화면 역시 에러 페이지가 아닌 빈 화면이 렌더링 된다.
데이터베이스에 존재하지않은 데이터를 요청하는것은 서버의 오류가 아니기 때문에 요청 받은 데이터를 검색할 수 없지만
프로그램 실행에는 아무런 문제가 없어 발생하는 문제이다.

외부 클래스 생성 - Exception Handling

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

```java
@GetMapping("/users/{id}")
public User retrieveUser(@PathVariable int id) {
    User user = service.findOne(id);

    if (user == null) {
        throw new UserNotFoundException(String.format("ID[%s] not found", id));
    }

    return user;
}
```

로직은 간단하다 user가 null 값일 경우 직접 생성한 UserNotFoundException 이 호출되는데 이 때 @ResponseStatus를 통해 NOT_FOUND - 404를 반환한다.

만약 @ResponseStatus를 통해 상태 코드를 제어 하지 않는다면 500번 서버 오류를 반환 한다. 이는 알맞은 상태 코드가 아니므로 상황에 따라 적절한 상태 코드 제어가 필요하다.

```
**주의**

위 방법으로만 예외 처리를 하게 된다면 스프링 부트가 제공하는 기본 에러 코드가 모두 출력된다.
(timestamp, status, error, message, trace) 이는 클라이언트에게 불필요한 정보이고
예외 처리시 민감한 정보가 함께 반환될 수 있으므로 주의가 필요하다.

```
AOP - Exception Handling

> 스프링 AOP(Aspect-Oriented Programming)는 객체 지향 프로그래밍의 한계를 극복하기 위해 등장한 프로그래밍 패러다임 중 하나로
기능의 분리와 모듈화를 위한 방법으로 여러 모듈에서 공통적으로 사용되는 기능을 분리하여 구현하고 이를 필요한 곳에서 재사용할 수 있는 기능이다.
트랜잭션 처리, 보안 처리, 로깅 처리, 캐시 처리 등에서 AOP를 사용할 수 있다.

먼저 특정 예외가 아닌 모든 예외 상황에 대해 500번 서버 에러를 반환해주는
Exception Handler를 생성해보자

먼저 예외 처리시 클라이언트에게 반환할 데이터를 저장할 클래스를 생성한다.

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ExceptionResponse {
    private Date timestamp;
    private String message;
    private String details;
}
```

모든 예외 처리

```java
@RestControllerAdvice
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)
    public final ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request) {
        ExceptionResponse exceptionResponse =
                new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription( includeClientInfo: false));

        return new ResponseEntity(exceptionResponse, HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

```

**@RestControllerAdvice**

해당 어노테이션이 할당되어있다면 모든 컨트롤러가 실행될 때 반드시
@RestControllerAdvice 어노테이션을 가진 빈이 실행된다.

**@ExceptionHandler()**
적용된 메소드가 ExceptionHandler로 사용될 수 있음을 지칭하는 어노테이션으로
인자값으로 지정된 클래스를 기반으로 에러 처리를 진행한다.

```

특정 예외 처리

```java
@ExceptionHandler(UserNotFoundException.class)
public final ResponseEntity<Object> handleUserNotFoundExceptions(Exception ex, WebRequest request) {
    ExceptionResponse exceptionResponse =
            new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription( includeClientInfo: false));

    return new ResponseEntity(exceptionResponse, HttpStatus.NOT_FOUND);
}
```

모든 예외 처리와 동일한 로직을 가지지만, @ExceptionHandler() 어노테이션에 직접 생성한
예외 클래스를 넣어서 커스텀한 예외 메세지를 적용할 수 있다.

DELETE HTTP Method

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable int id) {
    User user = service.deleteById(id);

    if (user == null) {
        throw new UserNotFoundException(String.format("ID[%S] NOT FOUND", id));
    }
}
```

PUT HTTP Method

```
@PutMapping("/users/{id}")
public ResponseEntity<Object> updateEUser(@RequestBody User user, @PathVariable int id){
    Optional<User> userOptional = Optional.ofNullable(service.findOne(id));

    if (!userOptional.isPresent()) return ResponseEntity.notFound().build();

    user.setId(id);
    service.save(user);
    return ResponseEntity.noContent().build();

}
```

DELETE, PUT 메소드 역시 앞에서 살펴본 GET, POST와 큰 차이점은 없다.
다만 DELETE의 경우 200 OK를 내보내도 되지만 PUT의 경우 204 No Content를 반환하는 게 바람직하다.

PATCH Method에 대해

스프링은 GET, POST, PUT, DELETE 와 PATCH 어노테이션도 지원 하지만 RESTful API를 설계할 때,
클라이언트의 의도와 상관 없이 동일 요청이 여러 번 들어올 수 있다.
이런 상황은 때에 따라 치명적일 수 있는데 이를 한 번만 실행할 수 있도록 하는 것을 RESTful API에서는 멱등성(Idempotent)이라고 한다.

GET/PUT/DELETE/HEAD/OPTIONS/TRACE 와 같은 HTTP 메소드는 멱등성을 지원하지만

POST/PATCH는 이를 지원하지 않기 때문에 CRUD 작업을 진행 할 때 PATCH 어노테이션을 사용하는건 지양된다.

LEVEL3 Hypermedia Controls

HTTP 상태 코드와 적절한 URI name을 구성했다면 다음은 HATEOAS를 적용해 현재 리소스와 연관된 (호출 가능한) 자원 상태 정보를 제공해 보자

먼저 스프링에서 HATEOAS를 적용하기 위해 관련된 라이브러리를 추가한다.

```
xml
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-hateoas</artifactId>
        </dependency>

gradle

  dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-hateoas'
        ...
        ...
}
```

우리는 이미 LEVEL2 단계에서 전체 유저 조회와 특정 유저 조회가 가능한 URI를 설계했는데, 이를 수정해 3단계를 구현한다.

```java
@GetMapping("/users")
public List<User> retrieveAllUsers() {
    return service.findAll();
}

@GetMapping("/users/{id}")
public EntityModel<User> retrieveUser(@PathVariable int id) {
    User user = service.findOne(id);

    if (user == null) {
        throw new UserNotFoundException(String.format("ID[%S] NOT FOUND", id));
    }
    // HATEOAS
    // "all-users", SERVER_PATH + "/users"
    // retrieveAllUsers
    EntityModel<User> model = EntityModel.of(user);
    WebMvcLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());
    model.add(linkTo.withRel("all-users"));

    return model;
}
```

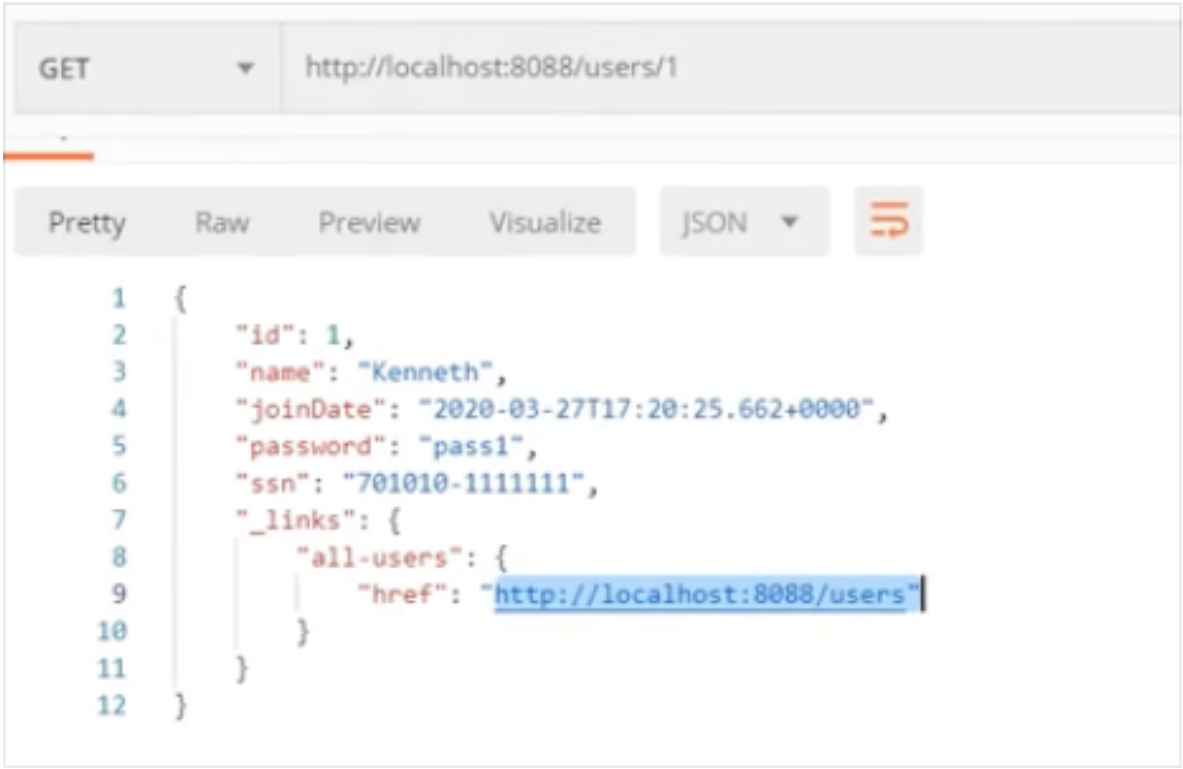
흐름

1. EntityModel에 user를 담는다.
EntityModel 클래스는 하이퍼미디어 링크를 포함하는 단일 리소스 모델 클래스로 클라이언트에게 보내는 단일 리소스를 나타내는 동시에 해당 리소스에 대한 링크를 제공해준다.

2. WebMvcLinkBuilder 클래스를 통해 링크를 생성한다.
WebMvcLinkBuilder는 단일 링크 혹은 링크 컬렉션을 생성하는 등의 다양한 메서드를 제공해주는 역할을 한다.(methodOn, linkTo)
(여기서는 전체 회원 조회 retrieveAllUsers( )와 관련된 링크를 생성한다.)

3. 추가로 생성된 link를 EntityModel에 추가한다.

결과



GET Mapping을 대표로 HATEOAS를 적용했다. 만약 다른 API도 HATEOAS를 적용해 구현하고싶다면 위 내용을 참고해 작성하자

마무리

레벨 2단계는 HTTP API 준수한 상태를 말하고, 3 단계는 REST API를 준수한 상태를 의미하지만,
현실적인 문제(너무 많은 정보 제공으로 트래픽 부하 ..)가 존재하기 때문에 Richardson은 최소한 HTTP Verbs를 준수한 2단계 수준의 API는 구현 하는게 바람직하다고 말한다.