Spring REST API

[REST 란?]

REST는 "REpresentational State Transfer"의 약어로, 하나의 URI는
하나의 고유한 리소스(Resource)를 대표하도록 설계된다는 개념이다.
즉, 자원을 이름으로 구분하여 해당 자원의 상태(정보)를 주고받는 모든 것을 의미한다.

REST(Representational State Transfer)는 월드 와이드 웹과 같은 분산 하이퍼미디어 시스템을 위한 소프트웨어 아키텍처의 한 형식이다. 이 용어는 로이 필딩(Roy Fielding)의 2000년 박사학위 논문에서 소개되었다. 필딩은 HTTP의 주요 저자 중 한 사람이다. 로이 필딩은 HTTP의 주요 저자 중 한 사람으로 그 당시 웹(HTTP) 설계의 우수성에 비해 제대로 사용되지 못하는 모습에 안타까워하며 웹의 장점을 최대한 활용할 수 있는 아키텍처로써 REST를 발표했다고 한다.

REST는 네트워크 아키텍처 원리의 모음이다. 여기서 '네트워크 아키텍처 원리'란 자원을 정의하고 자원에 대한 주소를 지정하는 방법 전반을 일컫는다. 필딩의 REST 아키텍처 형식을 따르면 HTTP나 WWW가 아닌 아주 커다란 소프트웨어 시스템을 설계하는 것도 가능하다. 또한, 리모트 프로시저 콜 대신에 간단한 XML과 HTTP 인터페이스를 이용해 설계하는 것도 가능하다. 필딩의 REST 원리를 따르는 시스템은 종종 RESTful이란 용어로 지칭된다.

REST 는 월드 와이드 웹(WWW)과 같은 분산 하이퍼미디어 시스템을 위한 소프트웨어 개발 아키텍처의 한 형식으로서 기본적으로 웹의 기존 기술과 HTTP 프로토콜을 그대로 활용하기 때문에 웹의 장점을 최대한 활용할 수 있는 아키텍처 스타일이고 네트워크 상에서 Client와 Server 사이의 통신 방식 중 하나이다.

REST 는 HTTP URI(Uniform Resource Identifier)를 통해 자원(Resource)을 명명하고, HTTP Method(POST, GET, PUT, DELETE)를 통해 해당 자원에 대한 CRUD 작업(Operation)을 적용한다.

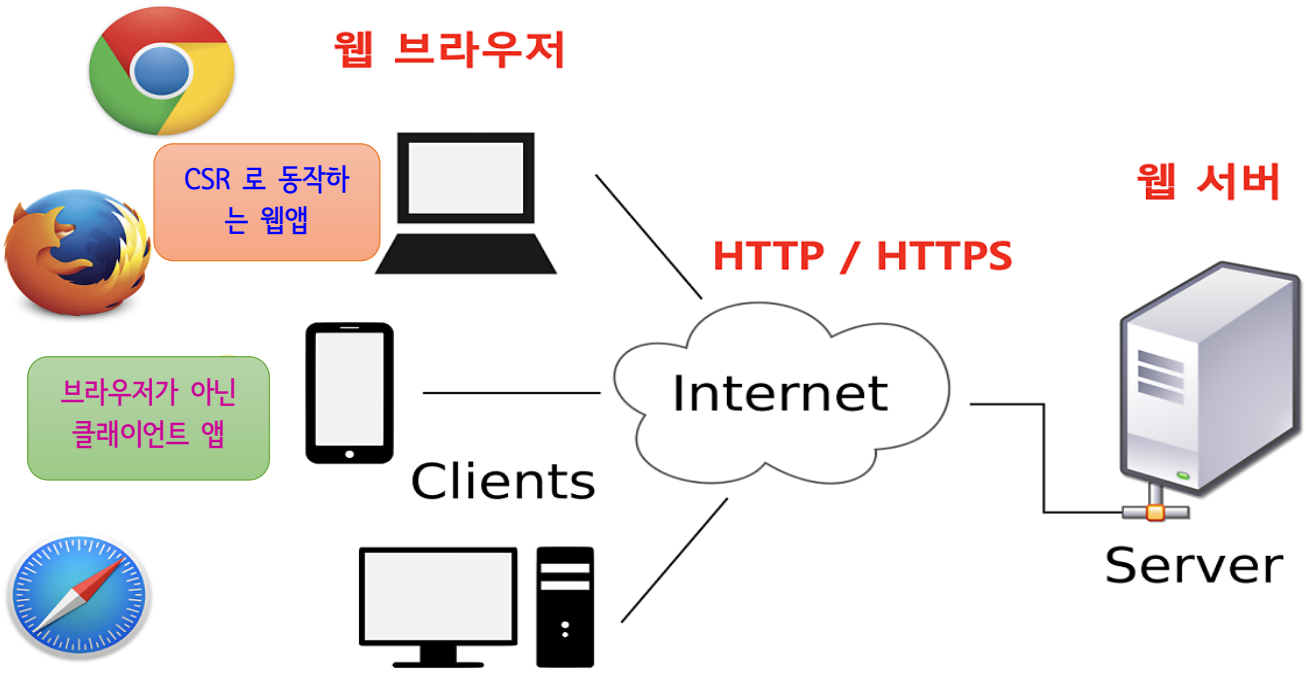
즉, REST는 자원 기반의 구조(ROA, Resource Oriented Architecture) 설계의 중심에 Resource가 있고 HTTP Method를 통해 Resource를 처리하도록 설계된 아키텍쳐를 의미한다. 웹 사이트의 이미지, 텍스트, DB 내용 등의 모든 자원에 고유한 ID인 HTTP URI를 부여한다.

CRUD	Operation
Create	생성(POST)
Read	조회(GET)
Update	수정(PUT)
Partial Update	부분수정(PATCH)

Delete 삭제(DELETE)
HEAD header 정보 조회(HEAD)

REST -> 자원을 이름(자원의 표현)으로 구분하여 해당 자원의 상태(정)를 주고 받는 모든 것

[REST가 필요한 이유]
- 애플리케이션 분리 및 통합
- Web 을 기반으로 하는 C&S 환경의 다양한 프로그램 개발
- 다양한 클라이언트의 등장



[REST 특징]

(1) Server-Client(서버-클라이언트 구조)

자원이 있는 쪽이 Server, 자원을 요청하는 쪽이 Client가 된다.

REST Server: API를 제공하고 비즈니스 로직 처리 및 저장을 책임진다.

Client: 사용자 인증이나 context(세션, 로그인 정보) 등을 직접 관리하고 책임진다.

서로 간 의존성이 줄어든다.

(2) Stateless(무상태)

HTTP 프로토콜은 Stateless Protocol이므로 REST 역시 무상태성을 갖는다.

Client의 context를 Server에 저장하지 않는다.

즉, 세션과 쿠키와 같은 context 정보를 신경쓰지 않아도 되므로 구현이 단순해진다.

Server는 각각의 요청을 완전히 별개의 것으로 인식하고 처리한다.

각 API 서버는 Client의 요청만을 단순 처리한다.

즉, 이전 요청이 다음 요청의 처리에 연관되어서는 안된다.

물론 이전 요청이 DB를 수정하여 DB에 의해 바뀌는 것은 허용한다.

Server의 처리 방식에 일관성을 부여하고 부담이 줄어들며, 서비스의 자유도가 높아진다.

(3) Cacheable(캐시 처리 가능)

웹 표준 HTTP 프로토콜을 그대로 사용하므로 웹에서 사용하는 기존의 인프라를 그대로 활용할 수 있다.

즉, HTTP가 가진 가장 강력한 특징 중 하나인 캐싱 기능을 적용할 수 있다.

HTTP 프로토콜 표준에서 사용하는 Last-Modified 태그나 E-Tag를 이용하면 캐싱 구현이 가능하다.

대량의 요청을 효율적으로 처리하기 위해 캐시가 요구된다.

캐시 사용을 통해 응답시간이 빨라지고 REST Server 트랜잭션이 발생하지 않기 때문에 전체 응답시간, 성능, 서버의 자원 이용률을 향상시킬 수 있다.

(4) Layered System(계층화)

Client는 REST API Server만 호출한다.

REST Server는 다중 계층으로 구성될 수 있다.

API Server는 순수 비즈니스 로직을 수행하고 그 앞단에 보안, 로드밸런싱, 암호화, 사용자 인증 등을 추가하여 구조상의 유연성을 줄 수 있다.

또한 로드밸런싱, 공유 캐시 등을 통해 확장성과 보안성을 향상시킬 수 있다.

PROXY, 게이트웨이 같은 네트워크 기반의 중간 매체를 사용할 수 있다.

(5) Code-On-Demand(optional)

Server로부터 스크립트를 받아서 Client에서 실행한다.

반드시 충족할 필요는 없다.

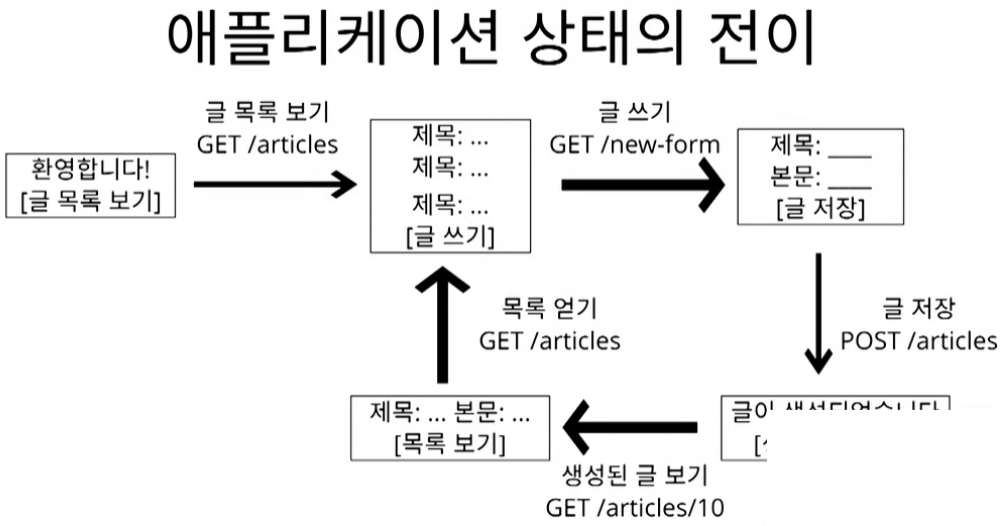
(6) Uniform Interface(인터페이스 일관성)

URI로 지정한 Resource에 대한 조작을 통일되고 한정적인 인터페이스로 수행한다.

HTTP 표준 프로토콜에 따르는 모든 플랫폼에서 사용이 가능하다.

특정 언어나 기술에 종속되지 않는다.

1) 리소스가 URI 로 식별되게 한다.
2) 리소스를 만들거나 업데이트 하거나 삭제하거나 할 때 REST Message 에 표현을 담아 전송하고 처리한다.(리프젠테이션 전송을 통한 리소스 조작)
3) Self-descriptive message (자체 표현 구조)
   메시지 스스로 자신을 설명한다.
   응답 메세지만 봐도 해당 메세지가 무엇을 제공하는 것인지 알 수 있어야 한다는 것이다.
   메시지를 봤을 때 메시지의 내용만으로 해석이 가능해야 한다!!!
4) Hypermedia As The Engine of Application State(HATEOAS)
   HAL 규약(application/han+json)으로 작성되는 응답 메시지를 통해 애플리케이션의 상태 전이는 응답시 함께 제공되는Hyperlink 를 이용한다.

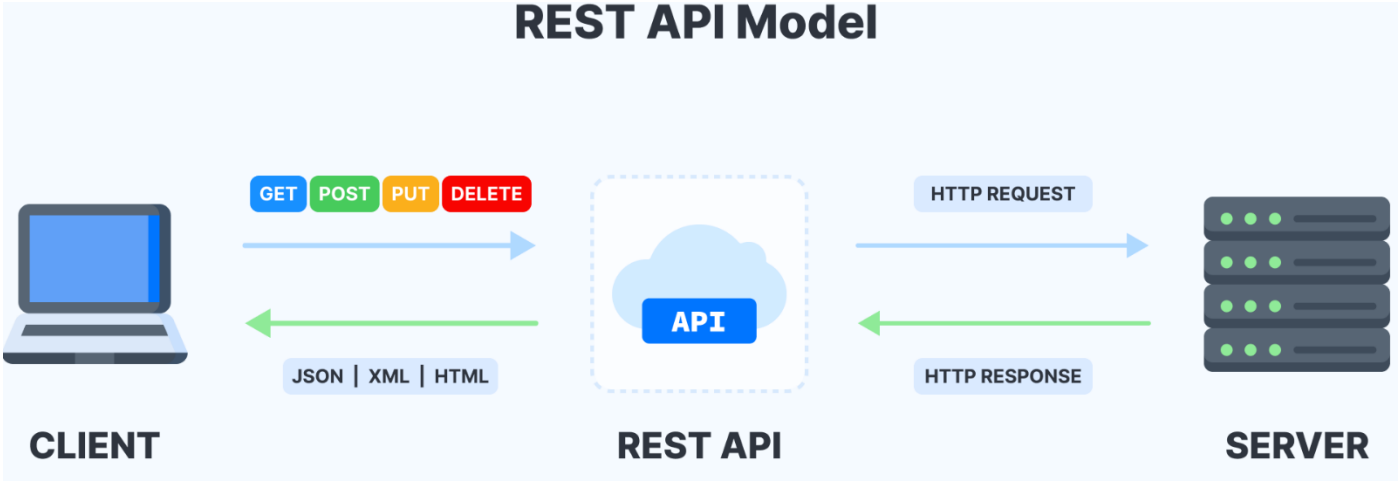


유니폼 인터페이스(Uniform Interface)를 만족해야 하는 이유

----> 서버와 클라이언트의 독립적인 진화를 위해서

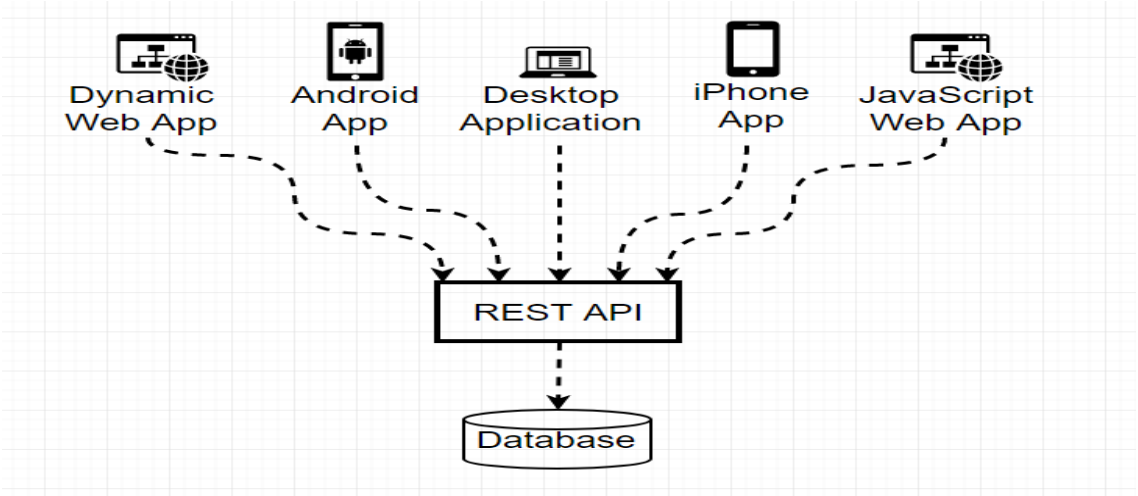
서버와 클라이언트는 독립적으로 진화할 수 있다.
서버의 기능이 변경되어도 클라이언트를 업데이트할 필요가 없다.

[REST API 란]



REST API 라는 것은 REST 아키텍처 스타일을 따르는 API 로서 분산 하이퍼미디어 시스템(웹)을 위한 아키텍처 스타일(제약조건 집합)이다.

스마트폰과 태블릿 등 서버에 접근하는 디바이스의 종류가 다양해지고 있기에 디바이스의 종류에 상관없이 공통으로 데이터를 처리할 수 있도록 하는 방식을 REST라고 한다. REST API는 사용자가 어떠한 요청을 했을 때 HTML을 리턴하지 않고, 사용자가 필요로 하는 결과(데이터)만을 리턴해주는 방식(JSON, XML)이다.



REST API는 REST 기반으로 서비스 API를 구현한 것으로 최근 OpenAPI(누구나 사용할 수 있도록 공개된 API: 구글 맵, 공공 데이터 등), 마이크로 서비스(하나의 큰 애플리케이션을 여러 개의 작은 애플리케이션으로 쪼개어 변경과 조합이 가능하도록 만든 아키텍처) 등을 제공하는 업체 대부분은 REST API를 개발하고 활용한다.

[REST API의 특징]

- 사내 시스템들도 REST 기반으로 시스템을 분산해 확장성과 재사용성을 높여 유지보수 및 운용을 편리하게 할 수 있다.
- REST는 HTTP 표준을 기반으로 구현하므로, HTTP를 지원하는 프로그램 언어로 클라이언트, 서버를 구현할 수 있다. 즉, REST API를 제작하면 자바스크립트, 자바, 파이썬, C# 등을 이용해 클라이언트를 개발할 수 있다.
- Self-decriptive -> 확장 가능한 커뮤니케이션
  서버나 클라이언트가 변경되더라도 오고가는 메시지가 self-descriptive 하므로 언제나 해석 가능하다.
- HATEOAS -> 애플리케이션 상태 전이의 late binding
  어디서 어디로 전이가 가능한지 미리 결정되지 않는다. 어떤 상태로 전이가 완료되고 나서야 그 다음 전이될 수 있는 상태가 결정된다. 즉 링크는 동적으로 변경 가능하다.

[REST API 설계 기본 규칙]

1. URI는 정보의 자원을 표현해야 한다.
   resource는 동사보다는 명사를, 대문자보다는 소문자를 사용한다.
   resource의 도큐먼트 이름으로는 단수 명사를 사용해야 한다.
   resource의 컬렉션 이름으로는 복수 명사를 사용해야 한다.
   resource의 스토어 이름으로는 복수 명사를 사용해야 한다.
   Ex) GET /Member/1 -> GET /members/1

2. 자원에 대한 행위는 HTTP Method(GET, PUT, POST, DELETE 등)로 표현한다.
   URI에 HTTP Method가 들어가면 안된다.
   Ex) GET /members/delete/1 -> DELETE /members/1
   URI에 행위에 대한 동사 표현이 들어가면 안된다.(즉, CRUD 기능을 나타내는 것은 URI에 사용하지 않는다.)
   Ex) GET /members/show/1 -> GET /members/1
   Ex) GET /members/insert/2 -> POST /members/2
   경로 부분 중 변하는 부분은 유일한 값으로 대체한다.(즉, :id는 하나의 특정 resource를 나타내는 고유 값이다.)
   Ex) student를 생성하는 route: POST /students
   Ex) id=12인 student를 삭제하는 route: DELETE /students/12

[REST API 디자인 가이드]

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

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

URI는 정보의 자원을 표현해야 한다. (리소스명은 동사보다는 명사를 사용) 다음에 제시된 URI 예는 REST를 제대로 적용하지 않은 URI이다.

GET /members/delete/1

URI는 자원을 표현하는데 중점을 두어야 한다. delete와 같은 행위에 대한 표현이 들어가서는 안된다. 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)로 표현한다. 위의 잘못 된 URI를 HTTP Method를 통해 수정해 보면 다음과 같다.

DELETE /members/1

또한 회원정보를 가져올 때는 GET, 회원 추가 시의 행위를 표현하고자 할 때는 POST METHOD를 사용하여 표현한다.

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 않다.
POST /members/2 (o)

[참고]HTTP METHOD의 알맞은 역할
POST, GET, PUT, DELETE 이 4가지의 Method를 가지고 CRUD를 할 수 있다.

METHOD	역할
POST	POST를 통해 해당 URI를 요청하면 리소스를 생성한다.
GET	GET를 통해 해당 리소스를 조회한다. 리소스를 조회하고 해당 도큐먼트에 대한 자세한 정보를 가져온다.
PUT	PUT를 통해 해당 리소스를 수정한다.(PATCH도 있음)
DELETE	DELETE를 통해 리소스를 삭제한다.

URI는 자원을 표현하는 데에 집중하고 행위에 대한 정의는 HTTP METHOD를 통해 하는 것이 REST한 API를 설계하는 중심 규칙이다.

[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 한다. REST API는 분명한 URI를 만들어 통신을 해야 하기 때문에 혼동을 주지 않도록 URI 경로의 마지막에는 슬래시(/)를 사용하지 않는다.
   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 있다.

4) 밑줄(_)은 URI에 사용하지 않는다.
   글꼴에 따라 다르긴 하지만 밑줄은 보기 어렵거나 밑줄 때문에 문자가 가려지기도 한다. 이런 문제를 피하기 위해 밑줄 대신 하이픈(-)을 사용하는 것이 좋다.(가독성)

5) URI 경로에는 소문자가 적합하다.
   URI 경로에 대문자 사용은 피하도록 해야 한다. 대소문자에 따라 다른 리소스로 인식하게 되기 때문이다. RFC 3986(URI 문법 형식)은 URI 스키마와 호스트를 제외하고는 대소문자를 구별하도록 규정하기 때문이다.
   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 않는다. 대신 Accept 헤더를 사용한다.

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

[리소스 간의 관계를 표현하는 방법]

REST 리소스 간에는 연관 관계가 있을 수 있고, 이런 경우 다음과 같은 표현방법으로 사용한다.

/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 있다.
예를 들어사용자가 '좋아하는' 디바이스 목록을 표현해야 할 경우 다음과 같은 형태로
사용될 수 있다.

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

[자원을 표현하는 Colllection과 Document]

컬렉션과 도큐먼트에 대해 알면 URI 설계가 한 층 더 쉬워진다. 도큐먼트는 문서 한 개 또는 한 개의 객체라고 할 수 있으며 컬렉션은 문서들의 집합, 객체들의 집합이라고 할 수 있다. 컬렉션과 도큐먼트는 모두 리소스라고 표현할 수 있으며 URI에 표현된다.

다음에 제시된 URL 은 sports라는 컬렉션과 soccer라는 도큐먼트로 표현되고 있다.

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

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

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

[REST API 설계 예시]

프론트엔드 웹에서 서버에 데이터를 요청하거나 배달 앱에서 서버에 주문을 넣거나 할 때 오늘날 널리 사용되는 것이 REST 란 형식의 API 이다. 과거의 SOAP (Simple Object Access Protocol)이라는 복잡한 형식을 대체한 것이다. REST 의 가장 중요한 특성은 각 요청이 어떤 동작이나 정보를 위한 것인지를 그 요청의 모습 자체로 추론 가능하다는 것이다. 어떤 학원에서 반과 학생들에 대한 API 를 만들 때

https://(사이트도메인)/classes
https://(사이트도메인)/classes/2
https://(사이트도메인)/classes/2/students
https://(사이트도메인)/classes/2/students/15
https://(사이트도메인)/classes/2/students?sex=male
https://(사이트도메인)/classes/2/students?page=2&count=10

요청을 보낼 때 사용하는 주소만으로도 대략 이게 뭘 하는 요청인지 파악이 가능하다.
또한 요청을 보낼 때 HTTP 요청 메서드를 사용해서 정보를 조회하는 것뿐만 아니라 생성, 수정 삭제 등도 처리할 수 있다.

GET - 조회
POST - 생성(바디에 새로운 학생 정보를 담는다)
PUT - 전체 변경(바디에 변경할 학생 정보를 담는다)
PATCH - 일부 변경(바디에 변경할 학생 일부 정보를 담는다)
DELETE - 삭제

CRUD	HTTP verbs	Route
resource들의 목록을 표시	GET	/resource
resource 하나의 내용을 표시	GET	/resource/:id
resource를 생성	POST	/resource
resource를 수정	PUT	/resource/:id
resource를 삭제	DELETE	/resource/:id

[HTTP 응답 상태 코드]

잘 설계된 REST API는 URI만 잘 설계된 것이 아닌 그 리소스에 대한 응답을 잘 내어주는 것까지 포함되어야 한다. 정확한 응답의 상태코드만으로도 많은 정보를 전달할 수가 있기 때문에 응답의 상태코드 값을 명확히 돌려주는 것은 생각보다 중요한 일이 될 수도 있다.

- 상태코드
200 클라이언트의 요청을 정상적으로 수행함

201 클라이언트가 어떠한 리소스 생성을 요청, 해당 리소스가 성공적으로 생성됨
(POST를 통한 리소스 생성 작업 시)
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 인증상태와 관계 없이 응답하고 싶지 않은 리소스를 클라이언트가 요청했을 때 사용하는 응답 코드(403 보다는 400이나 404를 사용할 것을 권고. 403 자체가 리소스가 존재한다는 뜻이기 때문에)
405 클라이언트가 요청한 리소스에서는 사용 불가능한 Method를 이용했을 경우 사용하는 응답 코드
301 클라이언트가 요청한 리소스에 대한 URI가 변경 되었을 때 사용하는 응답 코드
(응답 시 Location header에 변경된 URI를 적어 줘야 한다.)
500 서버에 문제가 있을 경우 사용하는 응답 코드

[웹 페이지의 REST 와 API 의 REST 비교]

웹 페이지는 REST 가 잘 되는데 API 는 REST가 잘 되지 않는다.

	웹 페이지	HTTP API
프로토콜	HTTP	HTTP
커뮤니케이션	사람-기계	기계-기계
미디어타입	**HTML**	**JSON**

	HTML	JSON
Hyperlink	가능	정의되어 있지 않음
Self-descriptive	가능(HTML 명세)	불완전

문법 해석은 가능하지만 의미를 해석하려면 별도로 문서가 필요하다.

Self-descriptive 한 REST API 설계를 하기 위한 방법

방법 1 : Media type
1. 미디어 타입을 하나 정의한다.
2. 미디어 타입 문서를 작성한다. 이 문서에 id 가 어떤 역할인지 또 title 이 어떤 역할인지 정의한다.
3. IANA에 미디어 타입을 등록한다. 만든 미디어 타입 문서를 미디어 타입의 명세로 등록한다.
4. 이제 이 메시지를 보는 사람은 명세를 찾아갈 수 있으므로 이 메시지의 의미를 온전히 해석할 수 있다

----------> 단점 : 매번 미디어 타입을 정의해야 한다.

방법 2 : Profile
1. id 가 어떤 역할인지 또 title 이 어떤 역할인지 정의한 명세를 작성한다.
2. Link 헤더에 profile relation으로 해당 명세를 링크한다.
3. 메시지를 보는 사람은 명세를 찾아갈 수 있으므로 이 문서의 의미를 온전히 해석할 수 있다.
----------> 단점 : 클라이언트가 Link 헤더(RFC 5988)와 profile(RFC6909)을 이해해야 한다.
content negotiation을 할 수 없다.

[HATEOAS]

방법 1 : data 로
data에 다양한 방법으로 하이퍼링크를 표현한다.
JSON 으로 하이퍼링크를 표현하는 방법을 정의한 명세들을 활용한다.
- JSON API
- HAL
- UBER
- Siren
- Collection+json

단점 : 기존 API를 많이 고쳐야 한다.

방법 2 : HTTP header 로
Link, Location 등의 헤더로 링크를 표현한다.

단점 : 정의된 relation 만 활용한다면 표현에 한계가 있다.

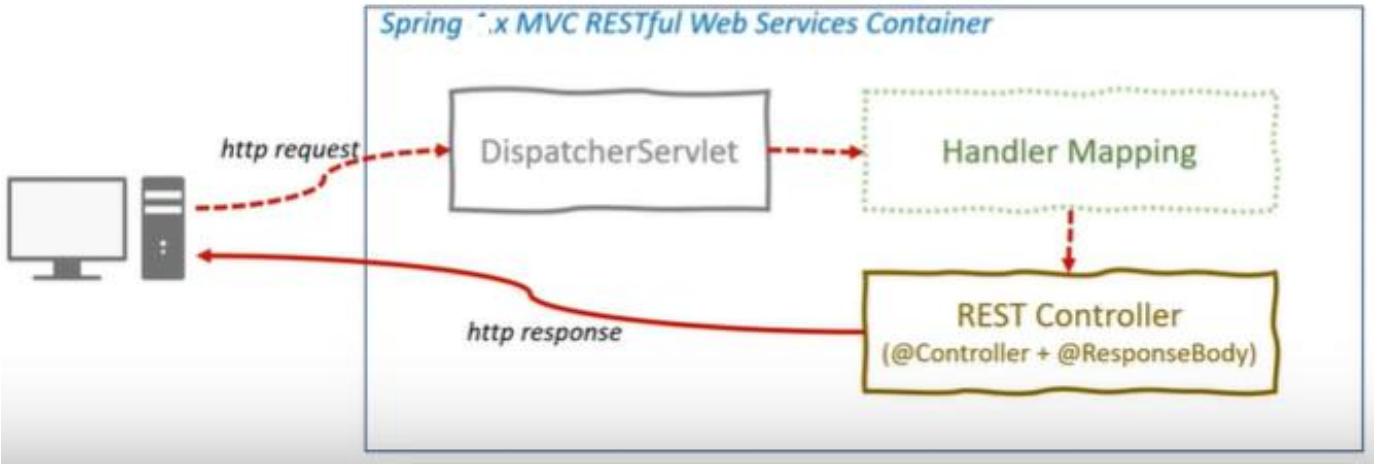
--> data, header 를 모두 사용하는 것도 좋음

[오늘날의 REST API 는…]

- 오늘날의 대부분의 “REST API”는 사실 REST를 따르지 않고 있다.
- REST의 제약조건 중에서 특히 Self-descriptive와 HATEOAS를 잘 만족하지 못한다.
- REST는 긴 시간에 걸쳐(수십년) 진화하는 웹 애플리케이션을 위한 것이다.
- REST를 따를 것인지는 API를 설계하는 이들이 스스로 판단하여 결정해야 한다.
- REST를 따르겠다면 Self-descriptive와 HATEOAS를 만족시켜야 한다.
  . Self-descriptive 는 custom media type이나 profile link relation 등으로 만족시킬 수 있다.
  . HATEOAS는 HTTP 헤더나 본문에 링크를 담아 만족시킬 수 있다.
- REST를 따르지 않겠다면 “REST를 만족하지 않는 REST API”를 뭐라고 부를지 결정해야 할 것이다.
  . **HTTP API** 라고 부를 수도 있고
  . 그냥 이대로 REST API라고 부를 수도 있다.

Spring 의 Rest API 개발 지원

Spring Rest
Spring HATEOAS



Spring Boot Auto Configuration
- DispatcherServletAutoConfiguration
- ErrorMvcAutoConfiguration
- HttpMessageConvertersAutoConfiguration -> JSON convert

DispatcherServlet -> '/'
- 클라이언트의 모든 요청을 한곳으로 받아서 처리
- 요청에 맞는 Handler로 요청을 전달
- Handler의 실행 결과를 Http Response 형태로 만들어서 반환

RestController
- SpringBoot 에서는 별도의 xml파일 설정 없이 어노테이션으로 등록
- Spring4부터 @RestController 어노테이션 지원
- RsetController = Controller + ResponseBody
- **View를 갖지 않는 REST Data(JSON/XML)를 반환 (데이터 포맷으로 반환)**

[사용 가능한 애노테이션]

사용한 어노테이션

어노테이션	설명	사용예시
@RestController	해당 class는 REST API 처리하는 Controller	
@Controller	return값이 String인 경우 text가 아닌 해당 String값의 html파일을 찾아서 리턴함	
@RequestMapping	URI를 지정해주는 어노테이션으로 get/post/put/delete 다 작동함. method속성을 이용하여 한 가지방식을 선택할 수 있음, 옛날방식	@RequestMapping(path="/hi", method= RequestMethod.GET)
@GetMapping, @PostMapping, @DeleteMapping, @PutMapping	@RequestMapping의 최근방식	@GetMapping("/hello"), @PostMapping(""), @DeleteMapping("/{userId}"), @PutMapping("")
@RequestBody	post보낼때 필수	@RequestBody postDTO dto
@JsonProperty	1개의 변수를 다른	@JsonProperty("phone_number")
@JsonNaming	DTO 전체 JSON형태를 지정함SnakeCaseStrategy, UpperCamelCaseStrategy,	@JsonNaming(value = PropertyNamingStrategy.SnakeCaseStrategy.clas

[POST 요청과 Content-Type 과의 관계]

@RequestBody vs @RequestParam vs @PathVariable

구분	설명	주 사용처	URL	ContentType
@PathVariable	URL경로에 변수를 넣는 것	주로 Rest api 에서 사용	http://localhost:8080/page/2	
@RequestParam	URL 파라미터로 값을 넘기는 방식	GET방식에서 주로 게시판 등에서 페이지 및 검색 정보를 함께 전달할 때 사용	http://localhost:8080/page?page=2&pageIdx=77	application/x-www-form-urlencoded; charset=UTF-8
@RequestBody	Body 자체를 넘기는 방식으로 POST에서만 사용 가능함. JSON형태로 전달된 데이터를 해당 파라미터 타입에 자동으로 저장하기때문에 변수명 상관없음	POST방식으로 주로 객체단위로 사용	http://localhost:8080/page	application/json; UTF-8;

[Spring Rest API 구현 관련 애노테이션]

@RestController
@ResponseBody
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping

[@GetMapping]

- @PathVariable을 이용한 GET

1. 매개변수와 변수이름 동일

```
@GetMapping(value = "/variable1/{variable}")
public String getVariable1(@PathVariable String variable) {
    log.info("@PathVariable을 통해 들어온 값 : {}", variable);
    return variable;
}
```

2. 매개변수와 변수이름 다름

```
@GetMapping(value = "/variable2/{variable}")
public String getVariable2(@PathVariable("variable") String var) {
    return var;
}
```

- @RequestParam을 이용한 GET

```
@GetMapping(value = "/request1")
public String getRequestParam1(
        @RequestParam String name,
        @RequestParam String email,
        @RequestParam String organization) {
    return name + " " + email + " " + organization;
}
```

- Map객체 활용

```
@GetMapping(value = "/request2")
public String getRequestParam2(@RequestParam Map<String, String> param) {
    StringBuilder sb = new StringBuilder();
    param.entrySet().forEach(map -> {
        sb.append(map.getKey() + " : " + map.getValue() + "₩n");
    });
    return sb.toString();
}
```

[@PostMapping]

RequestBody를 이용한 POST
  Post 요청은 클라이언트가 서버에 리소스를 저장하는데 사용한다.
  그리고 그 요청에는 리소스를 담기 위해 HTTP Body에 값을 넣어서 전송한다.
  일반적으로 Body에 작성되는 값은 JSON 형식으로 전송된다.
  JSON 형식으로 전달된 내용을 각각의 매개변수, DTO 객체 또는 MAP 객체로 전달받을 수 있다.
  어떤 값이 들어올지 모르는 POST요청의 경우에는 Map객체를 사용해서 body를 받는다.

```
@PostMapping(value = "/member")
    public String postMember(@RequestBody Map<String, Object> postData)
```

[@PutMapping]

PUT 메서드는 보통은 DTO가 모두 정해진 상태로 들어오므로 Map을 잘 사용하지는 않는다.

```
    // http://localhost:8080/api/v1/put-api/member1
    @PutMapping(value = "/member1")
    public String postMemberDto1(@RequestBody MemberDto memberDto) {
        return memberDto.toString();
    }

    // http://localhost:8080/api/v1/put-api/member2
    @PutMapping(value = "/member2")
    public MemberDto postMemberDto2(@RequestBody MemberDto memberDto) {
        return memberDto;
    }
```

DTO안에서 만들어 놓은 toString 메서드를 사용해서 결과값을 변환해서 출력할수도 있다. DTO 객체를 리턴하면 헤더영역의 Content-Type이 'text/plain'에서 'application/json'으로 변경된다.

@RestController 어노테이션이 지정된 클래스에서는 @ResponseBody를 생략할 수 있는데 이 @ResponseBody가 자동으로 값을 JSON형식으로 변환해서 전달하는 역할을 수행한다.

[@DeleteMapping]

1. @PathVariable을 이용한 DELETE

```
@DeleteMapping(value = "/{variable}")
public String DeleteVariable(@PathVariable String variable) {
    return variable;
}
```

DELETE API는 서버를 거쳐 데이터베이스에 저장된 리소스를 삭제할 때 사용한다.
이 때 컨트롤러를 통해 값을 받는 단계에서는 간단한 값만 받기 때문에 GET과 같이 URI에 값을 넣어 요청을 받은 형식으로 구현된다.

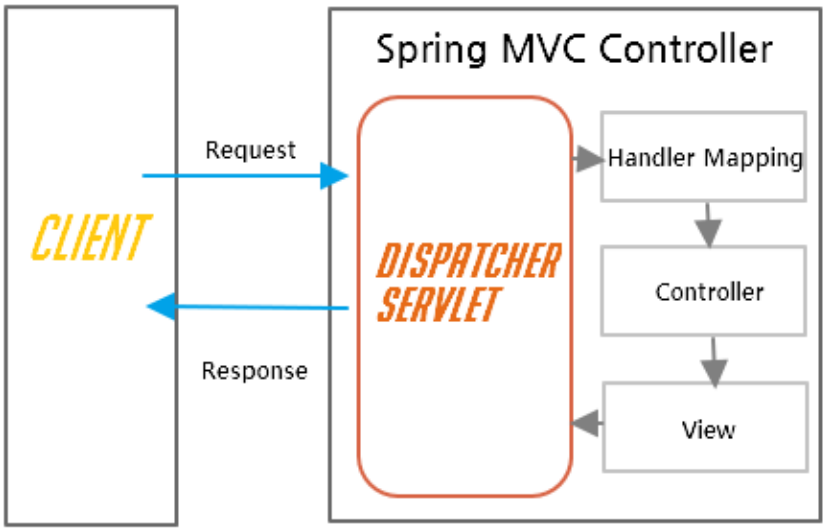
2. RequestParam을 이용한 DELETE

```
@DeleteMapping(value = "/request1")
public String getRequestParam1(@RequestParam String email) {
    ..................................;
}
```

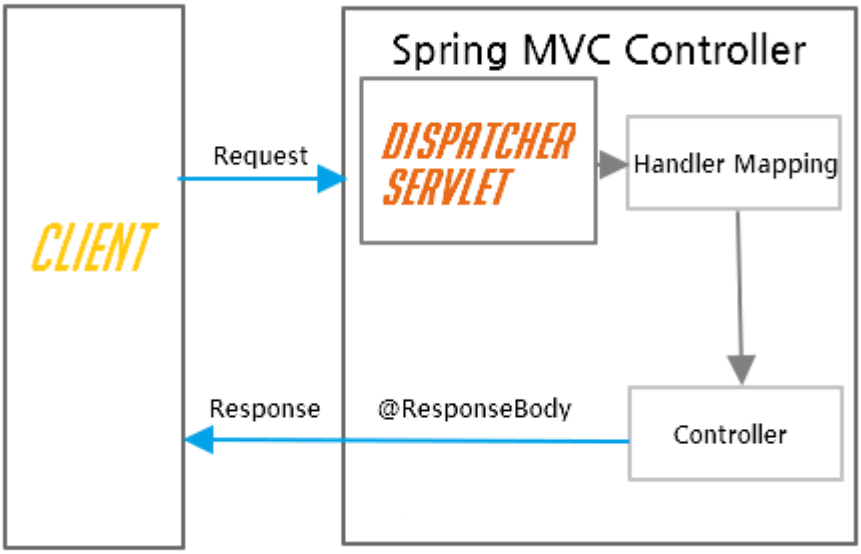
이렇게 URI에 포함된 값이나 Query String에 포함된 값을 전달받아 DELET 기능을 수행한다.

[@ResponseBody 와 @ResponseEntity]

- 템플릿을 통한 응답



- @ResponseBody 를 통한 응답



- @ResponseBody

@ResponseBody 는 HTTP 규격에 맞는 응답을 만들어주기 위한 애노테이션이다. HTTP 요청을 객체로 변환하거나, 객체를 응답으로 변환하는 HttpMessageConverter 를 사용한다. HTTPMessageConverter 는 해당 Annotation 이 붙은 대상을 response body 에 직렬화를 하는 방식으로 작동된다. 따라서 Controller 에서 반환할 객체나 Method 에 @ResponseBody 를 붙히는 것만으로 HTTP 규격에 맞는 값을 반환할 수 있다.

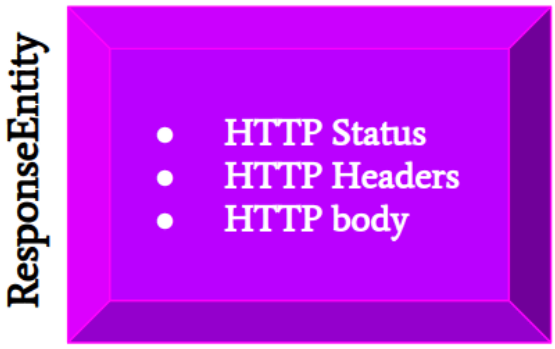
애노테이션을 추가하는 것으로 간단하게 처리를 할 수 있다는 점이 @ResponseBody 의 장점이다. 또한, 만약 해당 메서드를 가진 Controller 에 @RestController 가 붙으면 @ResponseBody 를 생략하여 더 간결하게 코드를 작성할 수 있다. 이미 해당 Annotation 에 명시가 되어있 때문에 생략해도 된다.

하지만 단점으로는 HTTP 규격 구성 요소 중 하나인 Header 에 대해서 유연하게 설정을 할 수 없다는 점이다. 또한 Status 도 메서드 밖에서 애노테이션을 사용하여 따로 설정을 해주어야 한다는 점이 있다. 이는 @ResponseBody 만 사용시에 별도의 뷰를 제공하지 않고, 데이터만 전송하는 형식이기 때문이다. 이와 같은 점들을 해결해 줄 수 있는 것이 바로 ResponseEntity 객체이다.

ResponseEntity 객체

ResponseEntity 도 마찬가지로 HTTP 응답을 빠르게 만들어주기 위한 객체이다. @ResponseBody 와 달리 Annotation 이 아닌 객체로 사용이 된다. 즉, 응답으로 변환될 정보를 모두 담은 요소들을 객체로 만들어서 반환한다. 객체의 구성요소에서 HttpMessageConverter 는 응답이 되는 본문을 처리해준 뒤에, RESTTemplate 에 나머지 구성 요소인 Status 를 넘겨주게 된다.

먼저, ResponseEntity 의 구조를 보게 되면, 다음과 같이 Status 만 필드 값으로 가지고 있다. ResponseEntity 에서 직접적으로 Status Code 를 지정할 수 있다는 것을 의미한다.




```
@GetMapping("/example")
public ResponseEntity<String> example(){
        String msg = "{ ₩" name ₩": ₩" 홍길동 ₩" }";

         HttpHeaders header = new HttpHeaders();
        header.add("Content-Type", "application/json; charset=UTF-8");

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

- Constructor보다는 Builder 패턴 사용을 권장

ResponseEntity 객체를 사용할 때, Constructor 를 사용하기보다는 Builder 를 활용하는 것을 권장하고 있다. 그 이유는 숫자로 된 상태 코드를 넣을 때, 잘못된 숫자를 넣을 수 있는 실수 때문이다. 따라서, Builder Pattern 를 활용하면 다음과 같이 코드를 변경할 수 있다.

```
return new ResponseEntity<MoveResponseDto>(moveResponseDto, headers,
                                    HttpStatus.valueOf(200));
```

```
return ResponseEntity.ok()
            .headers(headers)
            .body(moveResponseDto);
```

[추가]

HAL (Hypertext Application Language)은 JSON 또는 XML 코드 내의 외부 리소스 에 대한 링크와 같은 하이퍼미디어를 정의하기 위한 Internet Draft ("진행 중인 작업") 표준 규칙 이다. 이 표준은 2012년 6월에 특히 JSON과 함께 사용하기 위해 처음 제안되었으며 이후 JSON과 XML의 두 가지 변형으로 제공되었다. 두 개의 연관된 MIME 유형은 미디어 유형: application/hal+xml 및 미디어 유형: application/hal+json이다.

HAL을 채택한 API는 오픈 소스 라이브러리 의 사용을 단순화 하고 JSON 또는 XML을 사용하여 API 와 상호 작용할 수 있도록 한다.

Hypermedia as the Engine of Application State (HATEOAS)는 다른 네트워크 응용 프로그램 아키텍처 와 구별되는 REST 응용 프로그램 아키텍처의 제약 조건이다. HATEOAS를 통해 클라이언트는 응용 프로그램 서버가 하이퍼미디어를 통해 정보를 동적으로 제공하는 네트워크 응용 프로그램과 상호 작용한다 . REST 클라이언트는 하이퍼미디어에 대한 일반적인 이해 이상으로 애플리케이션 또는 서버와 상호 작용하는 방법에 대한 사전 지식이 거의 또는 전혀 필요하지 않다.

사용자 에이전트는 진입점 URL을 통해 REST API에 HTTP 요청을 한다. 클라이언트는 표현 내의 링크에서 선택하거나 미디어 유형이 제공하는 다른 방식으로 표현을 조작하여 애플리케이션 상태를 전환합니다. 이러한 방식으로 RESTful 상호 작용은 대역 외 정보가 아닌 하이퍼미디어에 의해 구동됩니다.

서버의 정보를 하이퍼미디어를 이용해서 동적으로 주고 받는다. 서버가 응답될 때 보내주는 관련 링크 정보(relation)를 가지고 서버와 소통한다. 애플리케이션의 상태는 상태의 변화에 따라서 링크 정보가 변경되어야 한다. 바로 이런 것을 HATEOAS 하는데 HATEOAS 를 쉽게 개발하도록 지원되는 API 가 Spring HATEOAS이다.

[Spring HATEOAS 구현 설명]

```
@GetMapping("/h_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));
    }
    EntityModel<User> model = EntityModel.of(user);
    WebMvcLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());
    model.add(linkTo.withRel("all-users"));
    return model;
}
```

- EntityModel<T>

RepresentationModel을 extend
도메인 객체를 감싸고, 그에 링크를 추가하는 객체
add( ) : 링크 추가 메소드
EntityModel객체.add(new Link("https://myhost/people/42")); 처럼 링크 인스턴스 추가 가능

- WebMvcLinkBuilder의 linkTo( )

컨트롤러 클래스를 가리키는 WebMvcLinkBuilder 객체를 반환

- WebMvcLinkBuilder의 methodOn( )

타겟 메소드(sample())의 가짜 메소드 콜이 있는 컨트롤러 프록시 클래스를 생성
프록시 클래스를 통해서 메서드를 호출하면 WebMvcLinkBuilder 객체가 호출된 메서드의 Link 객체를 포함하게 됨
직접 메소드 객체를 만드는 것보다 유연한 표현이 가능

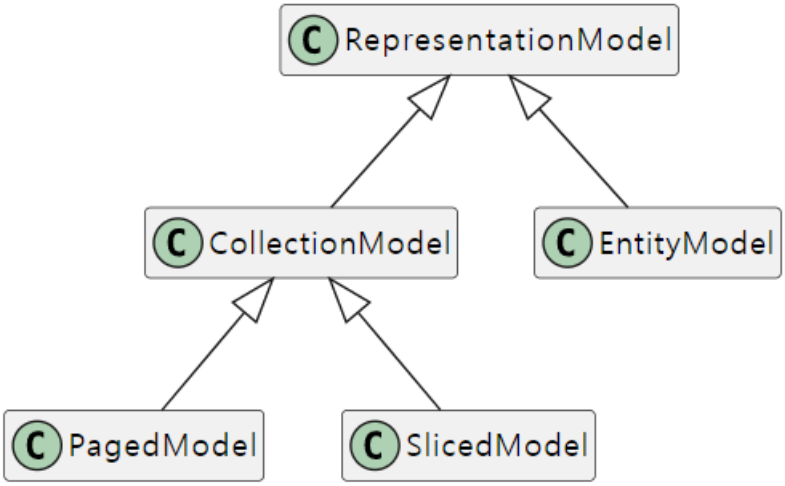
- LinkBuilderSuppoert의 withSelfRel( )
    빌더의 build( ) 역할
    현재 빌더 인스턴스를 self relationship 으로 하여 링크 객체 생성

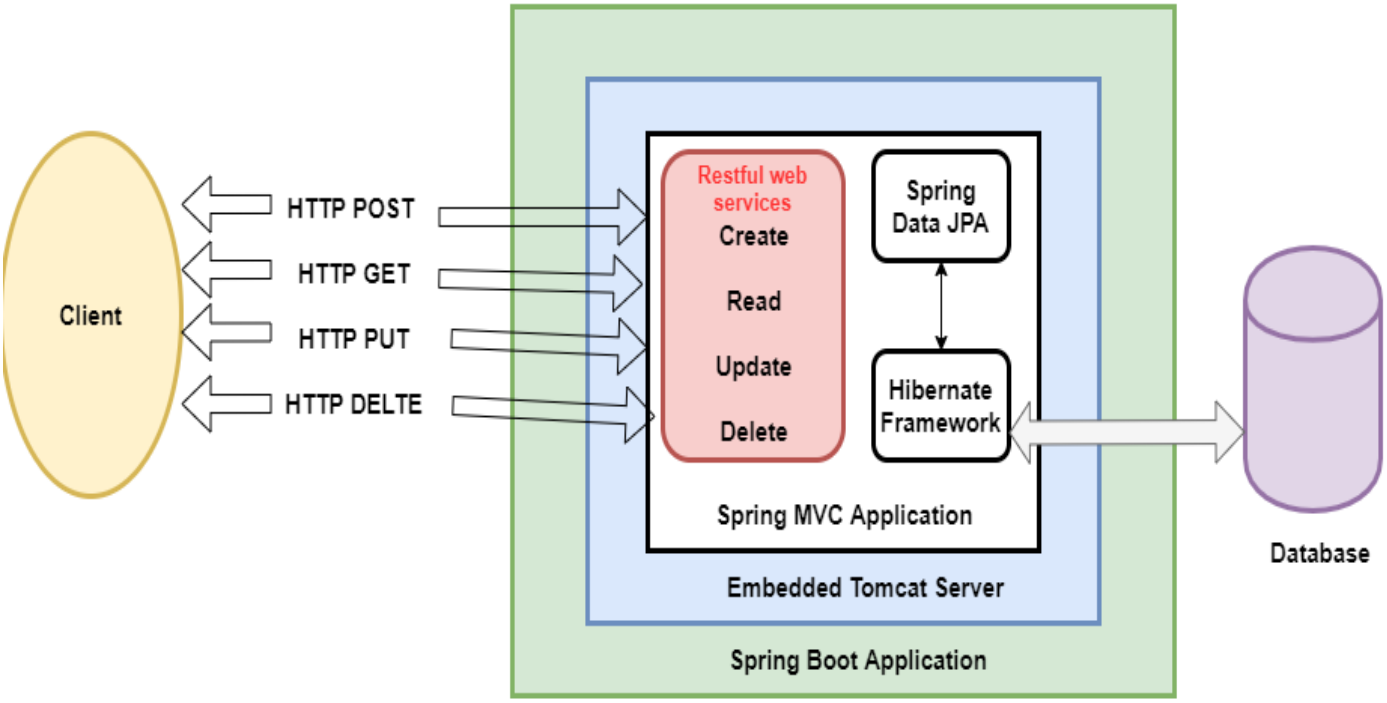
```
return new ResponseEntity<MoveResponseDto>(moveResponseDto, headers,
                HttpStatus.valueOf(200));

return ResponseEntity.ok()
        .headers(headers)
        .body(moveResponseDto);
```

[버전에 따른 API 변화(스프링 부트 2.2 부터)]

ResourceSupport → RepresentationModel
Resource → EntityModel
Resources → CollectionModel
PagedResources → PagedModel




Client
HTTP POST
HTTP GET
HTTP PUT
HTTP DELTE
Restful web services
Create
Read
Update
Delete
Spring Data JPA
Hibernate Framework
Spring MVC Application
Embedded Tomcat Server
Spring Boot Application
Database