4주차

Rest API

API란?

- Application Programming Interface의 줄임말
- 응용프로그램에서 사용할 수 있도록 다른 응용 프로그램을 제어할 수 있게 만든 인터페이스를 뜻함
- API를 사용하면 내부 구현 로직을 알지 못해도 정의되어 있는 기능을 쉽게 사용할 수 있다.

※인터페이스(interface)

- 어떤 장치간 정보를 교환하기 위한수단이나 방법을 의미함(ex. 마우스, 키보드, 터치패드 등)

REST란?

- Representational State Transfer의 줄임말
- 자원의 이름으로 구분하여 해당 자원의 상태를 교환하는 것을 의미
- Rest는 서버와 클라이언트의 통신 방식중 하나
- HTTP URL(Uniform Resource Identifier)를 통해 자원을 명시하고 HTTP Method를 통해 자원을 교환하는 것

※HTTP Method : CRUD(Create, Read, Update, Delete)

Rest의 특징(규칙)

- Server Client 구조
  자원이 있는 쪽이 server, 요청하는 쪽이 client
  클라이언트와 서버가 독립적으로(자원의 교집합 X, 각각의 서버 이용) 분리되어 있어야 함
- Stateless
  요청간에 클라이언트 정보가 서버에 저장되지 않음
  서버는 각각의 요청을 완전히 별개의 것으로 인식하고 처리

- Cacheable
  HTTP 프로토콜을 그대로 사용하기 때문에 HTTP의 특징인 캐싱 기능을 적용
  대량의 요청을 효율적으로 처리하기 위해 캐시를 사용
- 계층화(Layered System)
  클라이언트는 서버의 구성과 상관없이 Rest API 서버로 요청
  서버는 다중 계층으로 구성될 수 있음(로드밸런싱, 보안요소, 캐시 등)
- Code on Demand(Optional)
  요청을 받으면 서버에서 클라이언트로 코드 또는 스크립트(로직)을 전달하여 클라이언트 기능 확장
- 인터페이스 일관성(Uniform Interface)
  정보가 표준 형식으로 전송되기 위해 구성 요소간 통합 인터페이스를 제공
  HTTP 프로토콜을 따르는 플랫폼에서 사용 가능하게 설계

REST API란?

- REST 아키텍처의 조건을 준수하는 어플리케이션 프로그래밍 인터페이스를 뜻함
- 최근 많은 API가 REST API로 제공되고 있다.
- 일반적으로 REST 아키텍처를 구현하는 웹 서비스를 RESTFUL 하다고 표현한다.

REST API 특징

- REST기반으로 시스템을 분산하여 확장성과 재사용성을 높임
- HTTP 표준을 따르고 있어 여러 프로그래밍 언어로 구현할 수 있다.

REST API 장점

- HTTP 표준 프로토콜을 사용하는 모든 플랫폼에서 호환가능
- 서버와 클라이언트의 역할을 명확하게 분리
- 여러 서비스 설계에서 생길 수 있는 문제를 최소화

REST API 설계규칙

- 웹 기반의 REST API를 설계할경우에는 URL를 통해 자원을 표현해야 함
  http://thinkground.studio/member/589
  member → Resource / 589 → Resource id

- 자원에 대한 조작은 HTTP Method(CRUD)를 통해 표현해야함
  URI에 행위가 들어가면 안됨
  HEADER를 통해 CRUD를 표현해 동작을 요청해야함
- 메시지를 통한 리소스 조작
  HEADER를 통해 Content-Type을 지정하여 데이터를 전달
  대표적 형식으로 HTML, XML, JSON, TEXT가 있다.
- URI에는 소문자를 사용
- Resource의 이름이나 URI가 길어질 경우 하이폰(-)을 통해가족성을 높일 수 있다.
- 언더바(_)는 사용x
- 파일 확장자를 표현하지 않음

Hello World 응답하기

@RestController

- Spring Framework 4 버전부터 사용 가능한 어노테이션
- @Controller에 @Restponsebody가 결합된 어노테이션
- 컨트롤러 클래스 하우메소드에 @Restponsebody 어노테이션을 붙이지 않아도 문자열과 JSON 등을 전송할 수 있다.
- view를 거치지 않고 HTTP Restponsebody 에 직접 return 값을 담아 보내게 됨

@RequestMapping

- MVC의 핸들러 매핑(Handler Mapping)을 위해서 Default Annotation Handler Mapping을 사용
- Default Annotation Handler Mapping 매핑정보로 @Request Mapping 어노테이션 활용
- 클래스와 메소드의 RequestMapping을 통해 URL을 매핑하여 경로를 설정하여 해당 메소드에서 처리
  value : url 설정
  method : GET, POST, DELETE, PUT, PATCH 등
- 스프링 4.3 버전 부터 메소드를 지정하는 방식보다 간단하게 사용할수 있는 어노테이션을 사용할 수 있음

@GetMapping, @DeleteMapping, @PatchMapping, @PostMapping, @PutMapping

```
@RestController
public class HelloController{
    @GetMapping("Hello")
    pulbic string hello(){
        return "hello world";
    }
}
```

GET API

@ReuestMapping

- value와 method로 정의하여 API를 개발하는 방식
- 이제는 고전적인 방법으로 사용하지 않음

```
@RequestMapping(Value = "/hello", method = RequestMethode.GET)
Public String getHello(){return "hello world";}
```

@GetMapping(without param)

- 별도의 파라미터 없이 get api를 호출하는 경우 사용되는 방법

```
@GetMapping(Value = "/name")
Public String getName(){
return "flature";
}
```

@PathVariable

- GET 형식의 요청에서 파라미터를 전달하기 위해 URL에 값을 담아 요청하는 방법

- 아래 방식은 @GetMapping에서 사용된 {변수}의 이름과 메소드의 매개변수와 일치시켜야함

```
@GetMapping(Value = "/variable1/{variable}")//아래의 variable과 일치
Public String getVariable1(@PathVariable String variable){
    return variable;
}
```

- 아래 방식은 @GetMapping에서 사용된 {변수}의 이름과 메소드의 매개변수가 다를 경우 사용하는 방식
- 변수의 관리의 용이를 위해 사용하는 방식

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

- {variable} → variable → var

@RequestParam

- GET 형식의 요청에서 쿼리 문자열을 전달하기 위해 사용되는 방법
- “?”를 기준으로 우측에 {키} = {값}의 형태로 전달되면, 복수 형태로 전달할 경우 &를 사용함

```
@GetMapping(Value = "/request1")
Public String getRequestParam1(
    @Requestparam String name,
    @Requestparam String email,
    @Requestparam String organization
){
    return name + " " + email + " " + organization;
}
```

⇒ http://localhost:8080/api/v1/get-api/request/?name = flature & email = think.~com & organization = thinkgraound

@RequestParam

- GET 형식의 요청에서 쿼리 문자열을 전달하기 위해 사용되는 방법

- 아래 예시 코드는 어떤 요청값이 들어올지 모를 경우 사용하는 방식

```
@RequestMapping(Value = "/request2")
Public String getRequestParam2(@RequestParam Map < String, String > Param){
    Param.entrySet().forEach(map-> {
        String Builder sb = new String Builder();
        sb.append(map.getkey() + ":" + map.getValue()+"\n");
});
    return sb.toString();
}
```

⇒ http://~/v2/get-api/request2? name = flature & email= think~.com&organization = thinkgraound
map< String ⇒ key
String>Param ⇒ value

DTO 사용

- get형식의 요청에서 쿼리 문자열을 전달하기 위해 사용되는 방법
- key와 value가 정해져 있지만, 받아야할 파라미터가 많을 경우 DTO 객체를 사용한 방식

```
@GetMapping(Value = "/request3")
Public String getRequestParam3(MemberDTO memberDTO){
    return memeberDTO.toString();
}

Public class MemberDTO{
  private String name;
  private String email;
  private String organizatioin;
}
```

POST API

POST API

- 리소스를 추가하기 위해 사용되는 API
- @PostMapping
    - POST API를 제작하기 위해 사용되는 어노테이션

    - @RequestMapping + POST Method의 조합
- 일반적으로 추가하고자 하는 resource를 http body에 추가하여 서버에 요청
- 그렇기 때문에 @RequestBody를 이용하여 body에 담겨있는 값을 받아야 함

```
@PostMapping(Value = "/member")
Public String postMember(@RequestBody Map<String,Object>postData){
  String Builder sb = new String builder();
  postData.entryset().forEach(map->{
    sb.append(map.getkey() + ":" + map.getvalue()+"\n");
});
  return sb.toString();
}
```

DTO 사용

- key와 value가 정해져 있지만, 받아야할 파라미터가 많을 경우 DTO 객체를 사용한 방식

```
@PostMapping(Value = "/memeber2")
Public String postMemeberDTO(@RequestBody MemberDTO memberDTO){
  return memberDTO.toString();
}
```

DTO ← getset메소드가 있어야 한다.

PUT, DELETE API

PUT API

- 해당 리소스가 존재하면 갱신하고, 리소스가 없을 경우새로 생성해주는 API
- 업데이트를 위한 메소드, 기본적인 동작 방식은 POST API와 동일

DELETE API

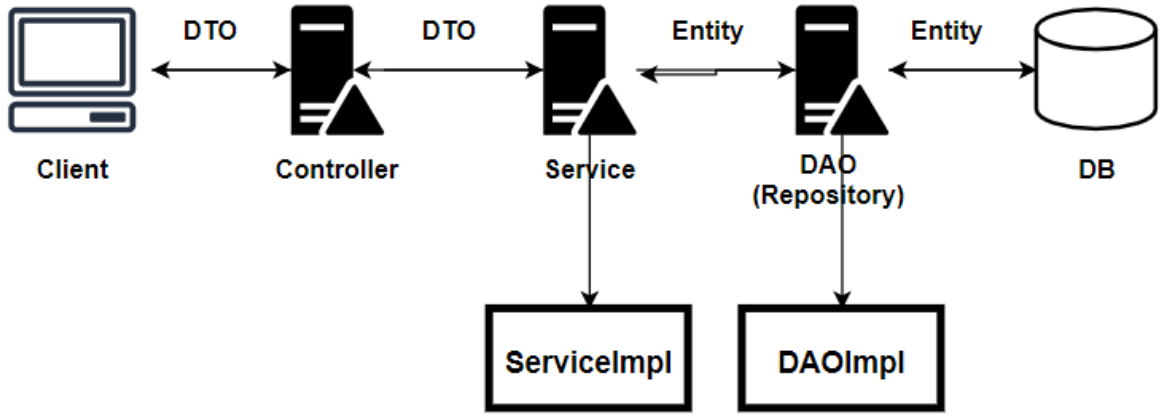
- 서버를 통해 리소스를 삭제하기 위해 사용되는 api
- 일반적으로 @PathVariable을 통해 리소스 id 등을 받아 처리

Response Entity

- Spring Framework에서 제공하는 클래스 중 Http Entity라는 클래스를 상속받아 사용하는 클래스
- 사용자의 http Request에 대한 응답 데이터를 포함
- 포함하는 클래스
  - Http Status
  - Http Header
  - Http Body

Entity, DAO, DTO, Repository

Spring Boot 서비스 구조



Entity(Domain)

- 데이터베이스에 쓰일 컬럼과여러 엔터티 간의 인과관계를 정의
- 데이터베이스의 테이블을 하나의 엔터티로 생각해도 무방함
- 실제 데이터베이스의 테이블과 1:1로 매핑됨
- 이 클래스의 필드는 각 테이블 내부의 컬럼(Column)을 의미

Repository

- Entity에 의해 생성된 데이터베이스에 접근하기 위한 인터페이스

- Service와 DB를 연결하는 고리 역할을 수행
- 데이터베이스에 적용하고자 하는 CRUD를 정의하는 영역

DAO(Data Access Object)

- Repository 활용해서 DB접근, 직접적으로 사용
- 데이터베이스에 접근하는 객체를 의미(Persistance Layer)
- Service가 DB에 연결할 수 있게 해주는 역할
- DB를 사용하여 데이터를 조회하거나 조작하는 기능을 전담

DTO(Data Transfer Object)

- DTO는 VO(Value Object)로 불리기도 하며, 계층간 데이터교환을 위한 객체를 의미
- VO의 경우 Read Only의 개념을 가지고 있다.
- Entity와 차이점은 Entity는 데이터베이스와 동일하게 만들어져 있는 클래스, DTO는 Entity와 같은 필드 값을 가질 수 있지만 서비스에서 추가 삭제하는 작업을 할 수 있기 때문에 DB의 컬럼에서 독립적이다

ORM과 JPA

ORM(Object Relational Mappin)

orm이란?

- 어플리케이션의 객체와 관계형 데이터베이스의 데이터를자동으로 매핑해주는 것을 의미
    - java의 데이터 클래스와 관계형 데이터베이스의 테이블 매핑
- 객체지향 프로그래밍과 관계형 데이터베이스의 차이로 발생하는 제약사항을 해결해주는 역할 수행
- 대표적으로 JPA, Hibernate 등이 있다 (Persistent API)

장점

- SQL 쿼리가 아닌 직관적인 코드로 데이터를 조작할 수 있다
  - 개발자가 보다 비즈니스 로직에 집중할 수 있다.
- 재사용 및 유지보수가 편리
  - ORM은 독립적으로 작성되어 있어 재사용이 가능
  - 매핑정보를 명확하게 설계하기 때문에 따로 데이터베이스를 볼 필요가 없다
- DBMS에 대한 종속서이 줄어듬
  - DBMS를교체하는 작업을 비교적 적은 리스크로 수행 가능

JPA(Java Persistance API)

jpa란?

- jap는 java Peristance API의 줄임말 이며, ORM과 관련된 인터페이스의 모음
- java 진영에서 표준 orm으로 채택됨
- ORM이 큰 캐념이라고 하면, jpa는 더 구체화 시킨 스팩을 포함하고 있다.

Hibernate

- ORM Framework 중 하나
- jpa의 실제 구현체 중 하나이며, 현재 jpa 구현체중 가장 많이 사용됨
- jpa
  - EclipseLink
  - Hibernate
  - DataNucreus

Spring Data JPA

- Spring Framework에서 JPA를 편리하게 사용할 수 있게 지원하는 라이브러리
  - CRUD 처리용 인터페이스 제공
  - Repository 개발 시 인터페이스만 작성하면 구현 객체를 동적으로 생성해서 주입
  - 데이터 접근 계층 개발시 인터페이스만 작성해도 됨

- Hibernate에서 자주 사용되는 기능을 좀 더 쉽게 사용할 수 있게 구현