6. REST & Ajax

Objectives

- REST방식의 데이터 교환 방식의 이해
- Ajax를 통한 JSON 데이터 통신
- jQuery를 이용하는 Ajax 처리
- JavaScript의 모듈 패턴

1. REST방식으로 전환

웹의 과거와 현재

- 과거의 웹 서비스
  - 고정된 브라우저의 주소창
  - 특정한 확장자를 이용하는 모델 2 방식(ex> *.do)
  - 특정한 파라미터에 의한 분기 구조
- 현재의 웹 서비스
  - URI + 식별 데이터
  - GET/POST외에 PUT/DELETE 등의 다양한 전송 방식 사용
  - 서버에서는 순수한 데이터만을 서비스 하는 방식

REST방식

- REST는 'Representational State Transfer'의 약어로 하나의 URI는 하나의 고유한 리소스(Resource)를 대표하도록 설계된다는 개념에 전송방식을 결합해서 원하는 작업을 지정

- 스프링에서는 다양한 어노테이션과 기능을 통해서 REST방식의 서비스를 간편하게 구축할 수 있음

어노테이션	기능
@RestController	Controller가 REST 방식을 처리하기 위한 것임을 명시합니다.
@ResponseBody	일반적인 JSP와 같은 뷰로 전달되는 게 아니라 데이터 자체를 전달하기 위한 용도
@PathVariable	URL 경로에 있는 값을 파라미터로 추출하려고 할 때 사용
@CrossOrigin	Ajax의 크로스 도메인 문제를 해결해주는 어노테이션
@RequestBoby	JSON 데이터를 원하는 타입으로 바인딩 처리

@RestController

- 스프링 4에서부터는 @Controller 외에 @RestController라는 어노테이션을 추가해서 해당 Controller의 모든 메서드의 리턴 타입을 기존과 다르게 처리한다는 것을 명시
- @RestController는 메서드의 리턴 타입으로 사용자가 정의한 클래스 타입을 사용할 수 있고, 이를 JSON이나 XML로 자동으로 처리

예제프로젝트의 준비

- 데이터의 처리는 XML과 JSON을 이용할 것이므로 pom.xml을 변경
- jackson-databind와 jackson-mapper-asl : json을 리턴
- Jackson-dataformat-xml : xml로 리턴
- Java객체를 JSON으로 쉽게 변환할 수 있는 gson 라이브러리

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.16.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.codehaus.jackson/jackso
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.13</version>
</dependency>
```

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.13.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.9.0</version>
</dependency>
```

@RestController의 반환 타입

- @RestController를 사용하는 컨트롤러에서는 다음과 같은 반환 타입들을 사용한다.
  - String 혹은 Integer 등의 타입들
  - 사용자 정의 타입
  - ResponseEntity<> 타입
- 주로 ResponseEntity 타입을 이용하는 것이 일반적

SampleVO클래스와 SampleController

- JSON 혹은 XML로 변환될 데이터

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class SampleVO {
  private Integer mno;
  private String firstName;
  private String lastName;
}
```

- SampleVO를 서비스하는 SampleController

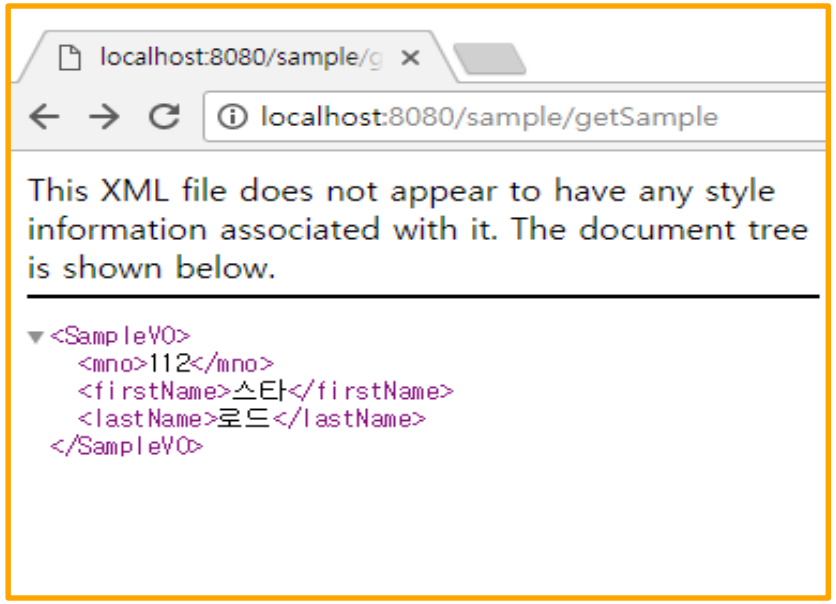
```
@RestController

@RequestMapping("/sample")

@Log4j

public class SampleController {

}
```

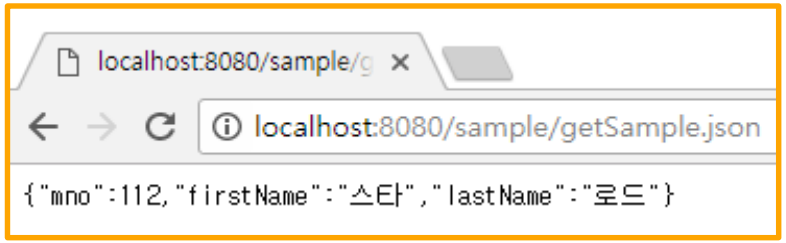
JSON/XML의 테스트

```
@Log
@Controller
@RequestMapping("/sample/**")
public class SampleController {

        @GetMapping(value="getSample")
        public @ResponseBody SampleVO getSample() {
          return new SampleVO(1,"hong","gidong");
        }
}
```

```
@Log
@RestController
@RequestMapping("/sample/**")
public class SampleController {

        @GetMapping(value="getSample")
        public  SampleVO getSample() {
          return new SampleVO(1,"hong","gidong");
        }
}
```

@RestController=@Controller+ @**ResponseBody**



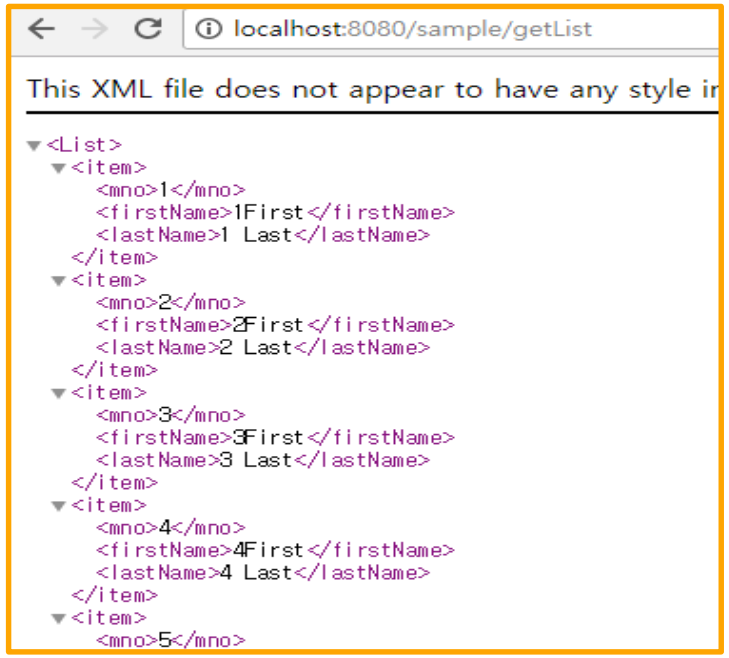


확장자에 따라 다른 타입으로 서비스

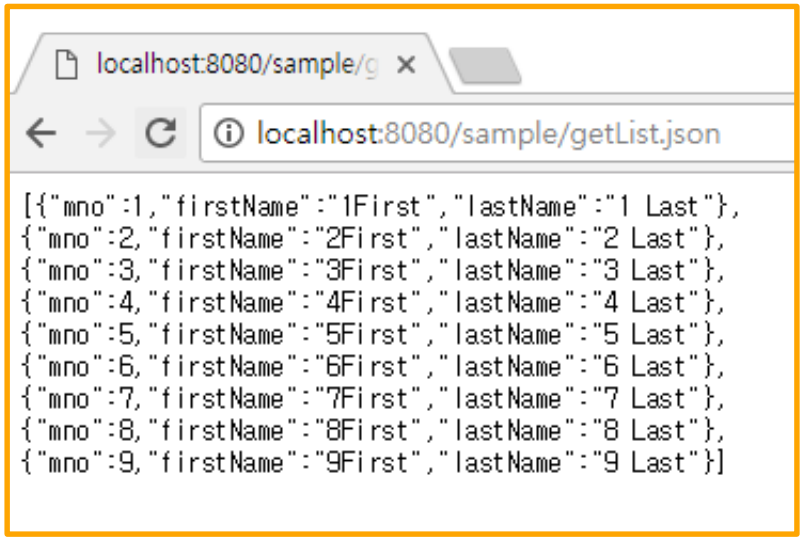
Collection타입의 객체 반환-List

```
@GetMapping(value="getList")
public List<SampleVO> getSampleList() {
        List<SampleVO> list=new ArrayList<SampleVO>();
        for(int i=1; i<=10;i++) {
                list.add(new SampleVO(i,"Frist_"+i,"Last_"+i));
        }
        return  list;
}
```

http://localhost:8080/sample/getList



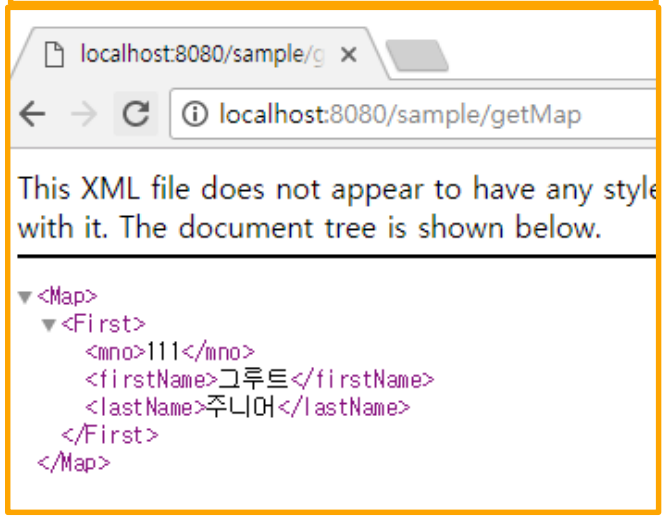
http://localhost:8080/sample/getList.json



Collection타입의 객체 반환-Map

```
@GetMapping(value="getMap")
public Map<String, SampleVO> getSampleMap() {
        Map<String, SampleVO> map=new HashMap<String,SampleVO>();
        map.put("first",new SampleVO(11,"그루트","주니어"));
        map.put("second",new SampleVO(11,"길동","홍"));

        return  map;
}
```

http://localhost:8080/sample/getMap



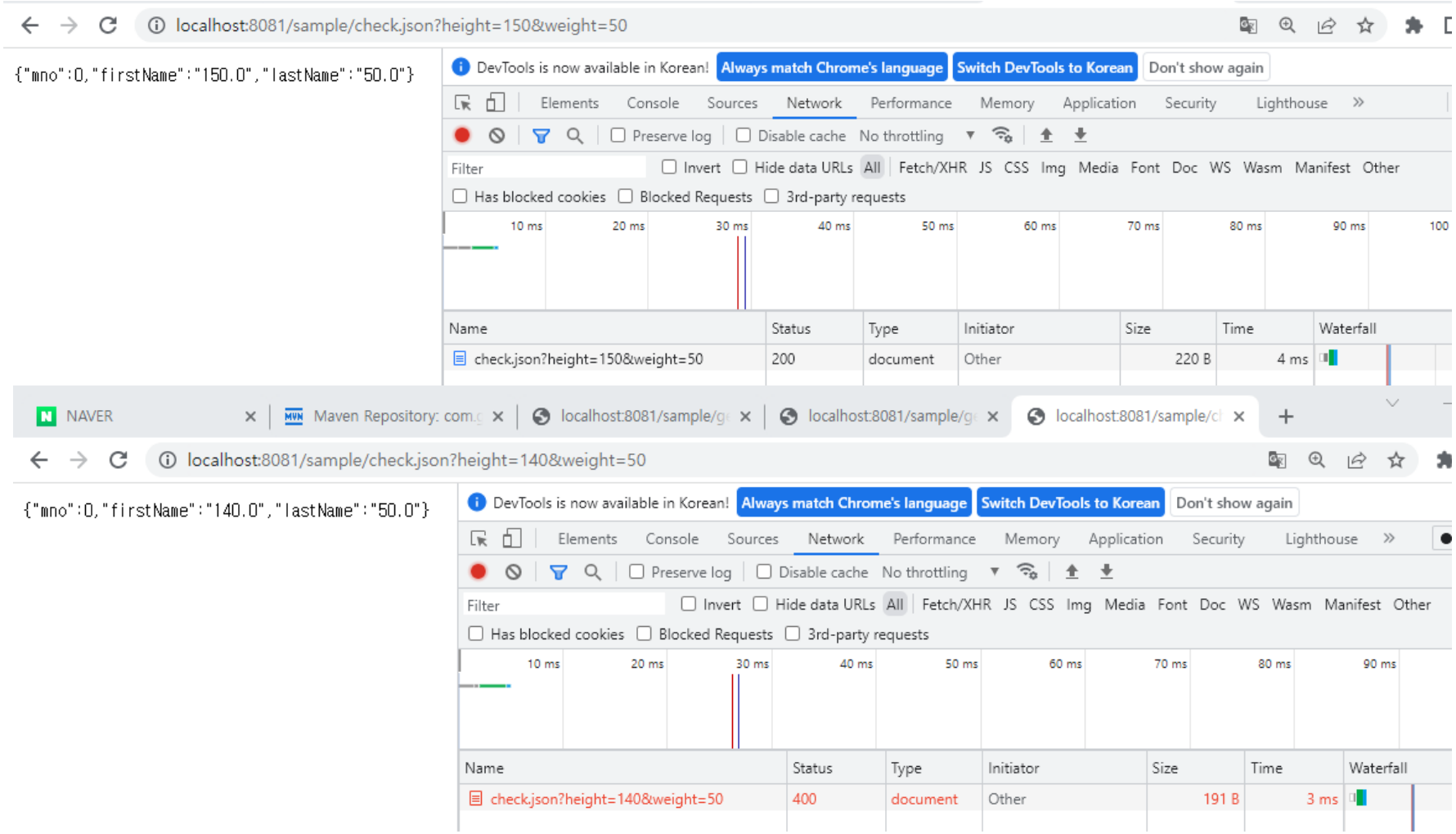
http://localhost:8080/sample/getMap.json

ResponseEnitity타입

- 데이터뿐만 아니라 브라우저에 HTTP상태 코드등 추가적인 데이터를 전달할 수 있다는 장점

```
@GetMapping(value = "/check", params = { "height", "weight" })
  public ResponseEntity<SampleVO> check(Double height, Double weight) {
    SampleVO vo = new SampleVO(000, "" + height, "" + weight);
    ResponseEntity<SampleVO> result = null;
    if (height < 150) {
      result = ResponseEntity.status(HttpStatus.BAD_GATEWAY).body(vo);
    } else {
      result = ResponseEntity.status(HttpStatus.OK).body(vo);
    }
    return result;
 }
```

실행화면에서 우클릭-> 검사 클릭->네트워크 크릭 후 새로고침 클릭 (height=140, height=150 비교



@RestController의 파라미터

- @PathVariable:
  - 일반 컨트롤러에서도 사용이 가능하지만 REST 방식에서 자주 사용됨.
  - URL 경로의 일부를 파라미터로 사용할 때 이용
- @RequestBody:
  - JSON 데이터를 원하는 타입의 객체로 변환해야 하는 경우에 주로 사용
  - 일반 <form>방식으로 처리된 데이터

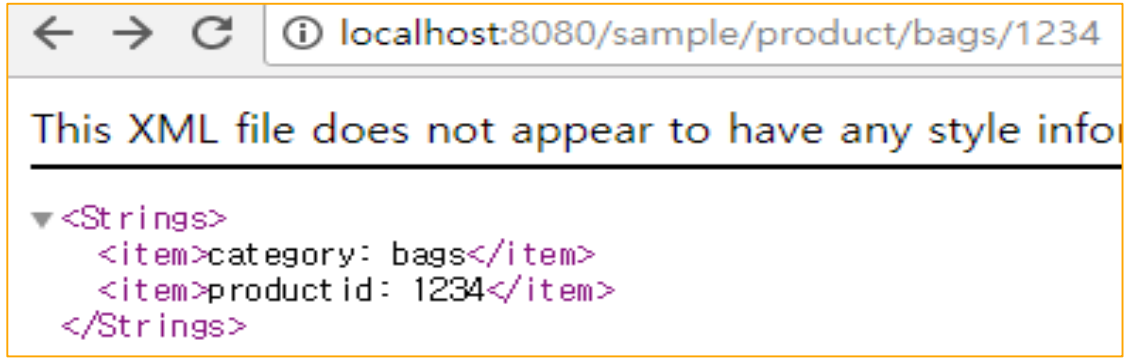
@PathVariable

- URI경로 중간에 들어간 값을 얻기 위해서 사용

http://localhost:8080/sample/{sno}

http://localhost:8080/sample/{sno}/{page}/{pno}

```
@GetMapping("product/{cat}/{pid}")
  public String[] getPath(
      @PathVariable("cat") String cat,
      @PathVariable("pid") Integer pid) {
    return new String[] { "category: " + cat, "productid: " + pid };
 }
```



@RequestBody

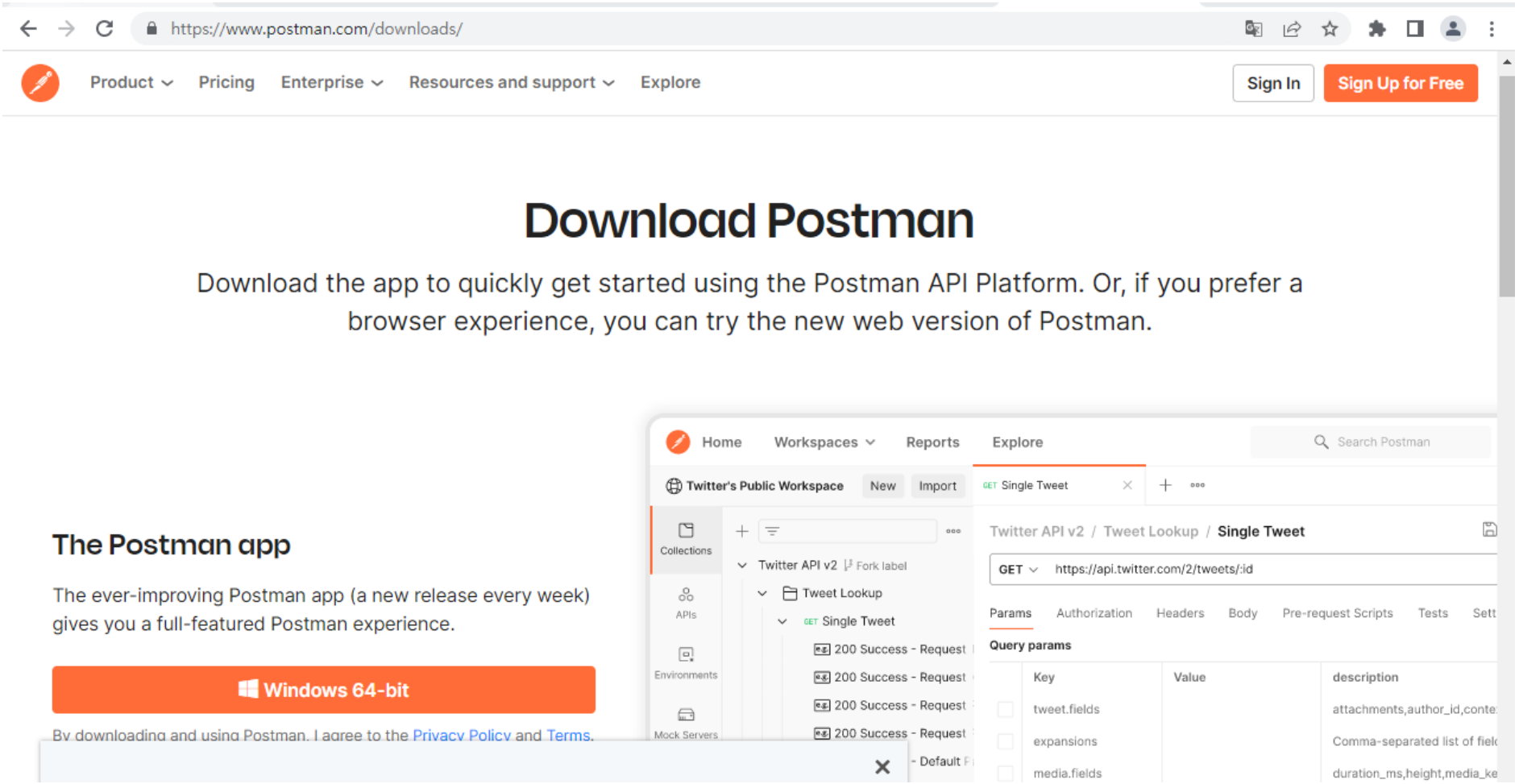
- 전송된 데이터가 JSON이고, 이를 컨트롤러에서는 사용자 정의 타입의 객체로 변환할때 사용

```
@PostMapping("convert")
public SampleVO convert(@RequestBody SampleVO vo) {
    log.info("covert.............sampleVO:"+vo);
    return vo;
}
```

일반적으로 브라우저에서는 JSON형태의 데이터를 전송할 수 없으므로
별도의 REST관련 도구(Postman)를 이용해서 테스트를 진행해야 함

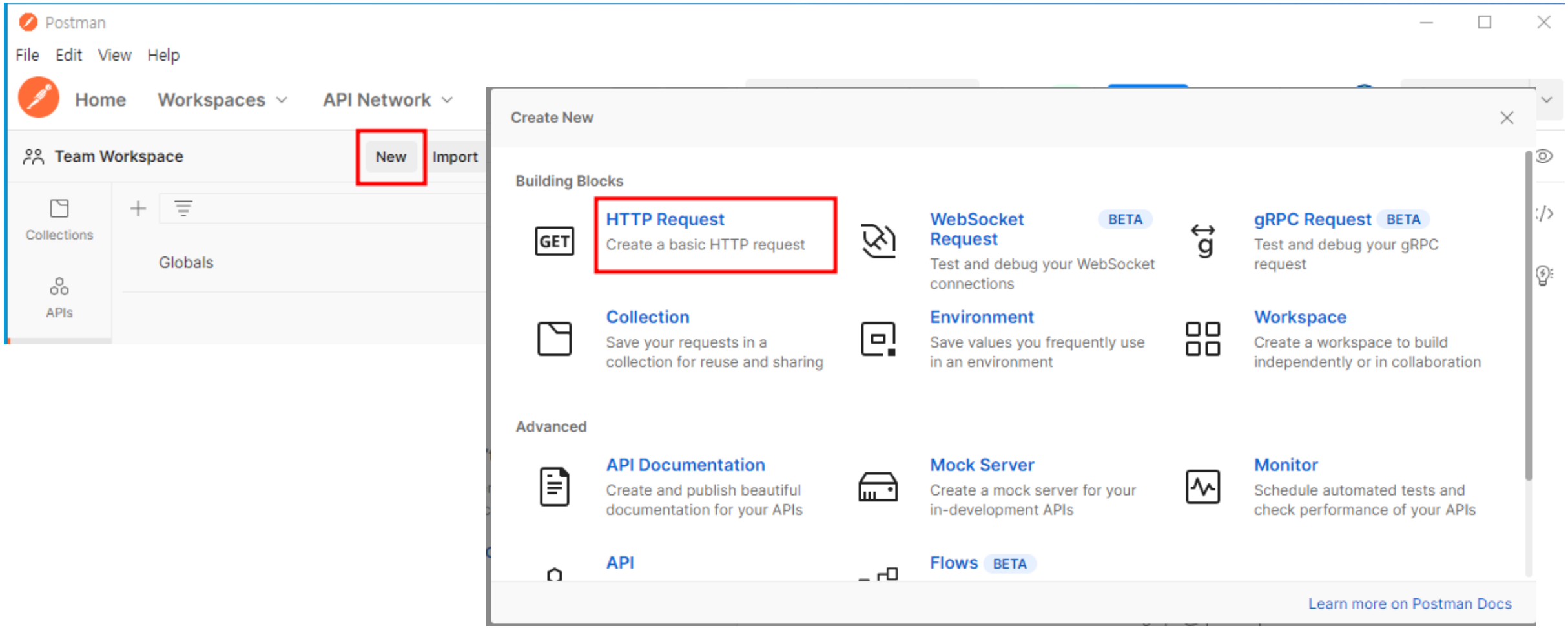
Postman 프로그램을 이용하는 테스트

- Postman 다운로드 : https://www.postman.com/downloads/



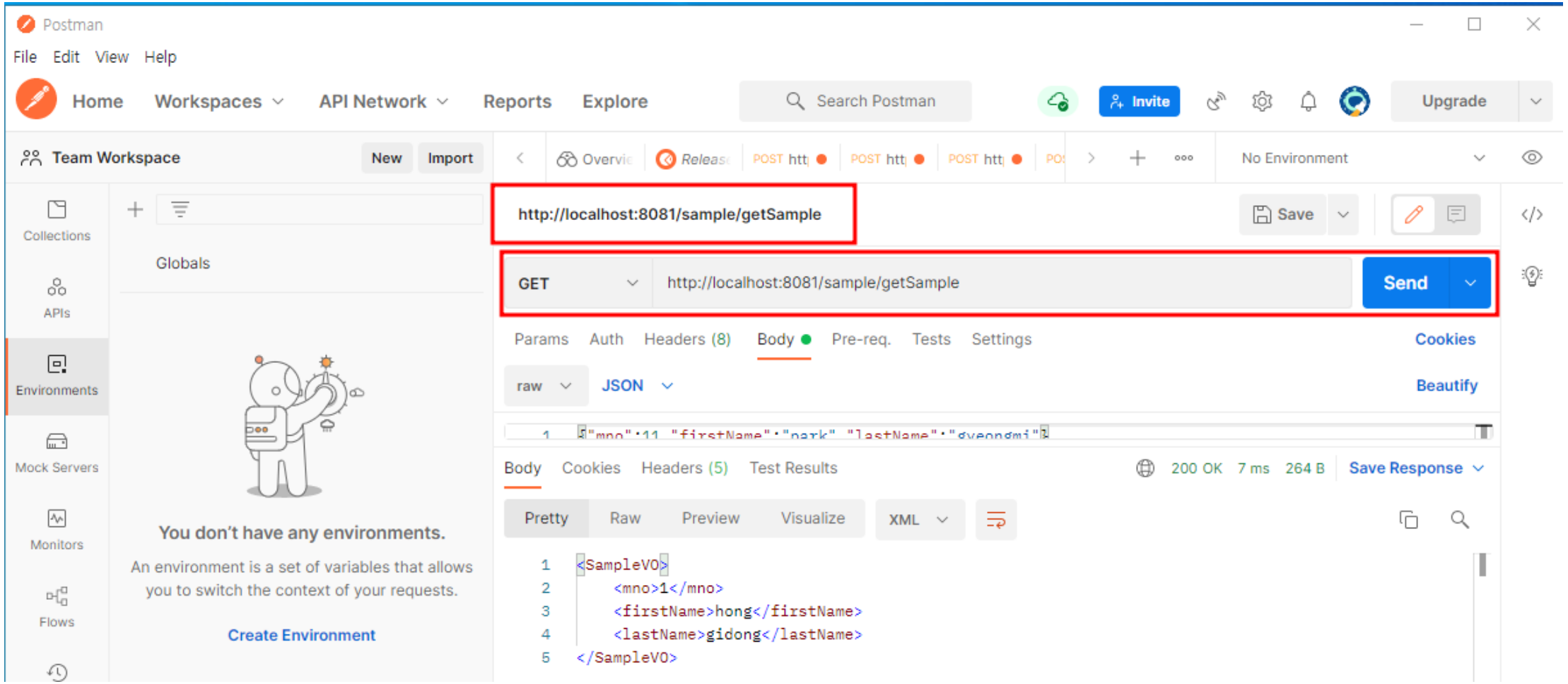
Postman을 이용한 테스트

- Postman 상세 사용방법 : https://binit.tistory.com/17
- New 클릭->HTTP Request 선택



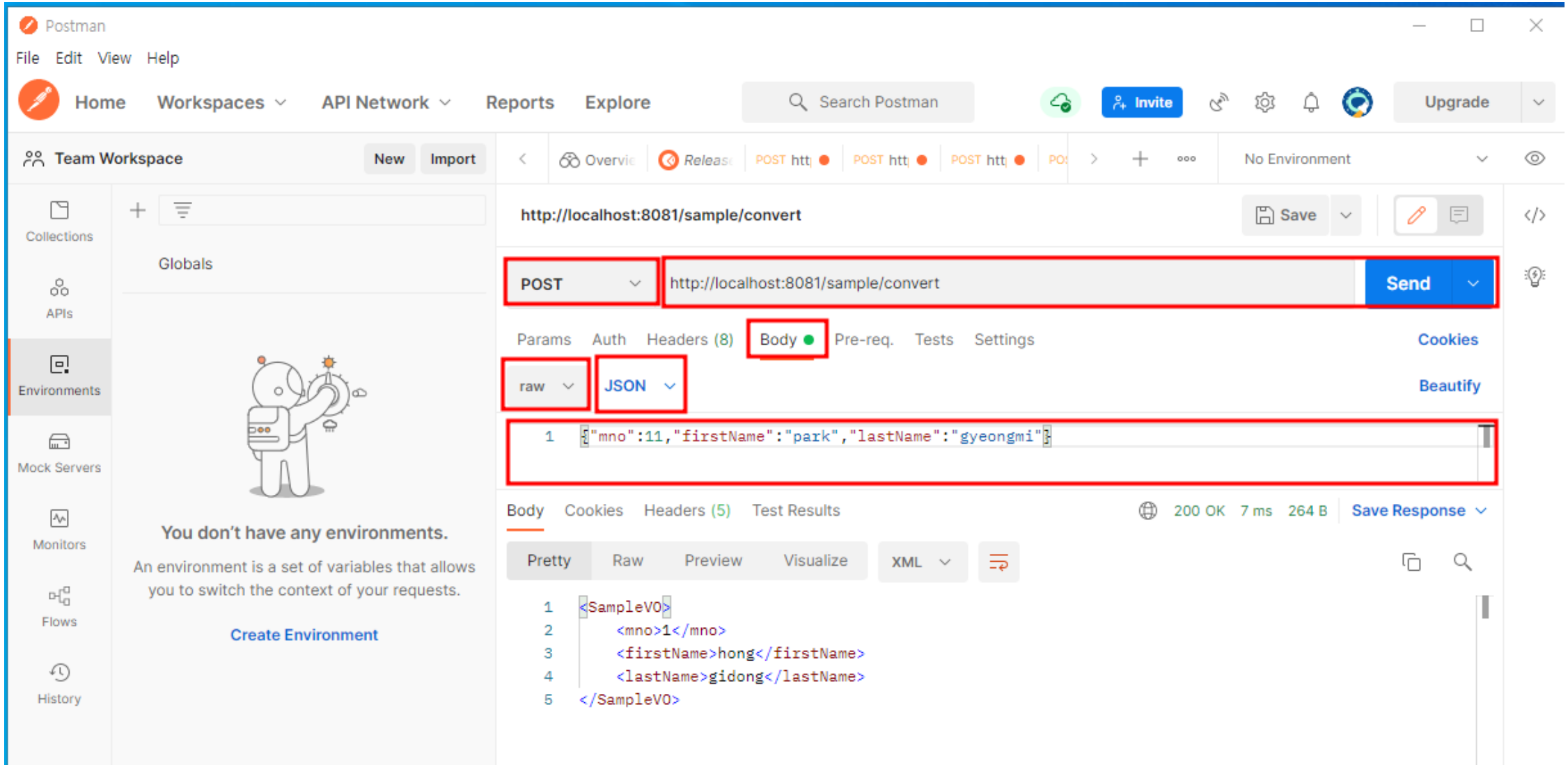
Postman을 이용한 테스트

- GET 메소드 사용 : 결과



Postman을 이용한 테스트

- POST : 테스트



다양한 전송방식과 URI설계

- REST 방식의 데이터 교환에서 가장 특이한 점은 기존의 GET/POST 외에 다양한 방식으로 데이터를 전달한다는 점

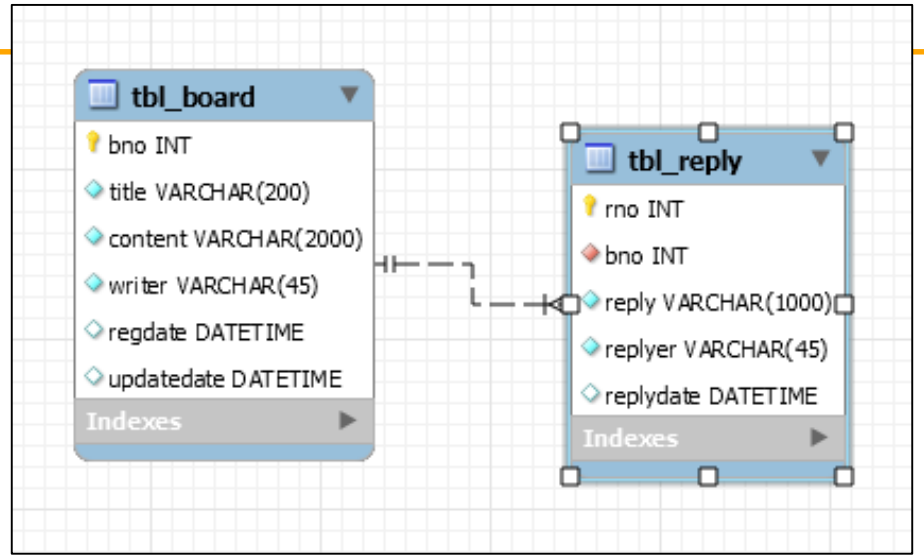
작업	전송방식
Create	POST
Read	GET
Update	PUT
Delete	DELETE

작업	전송방식	URI
등록	POST	/members/new
조회	GET	/members/{id}
수정	PUT	/members/{id} + body (json 데이터 등)
삭제	DELETE	/member/{id}

2. Ajax를 이용하는 댓글 처리

댓글 처리를 위한 테이블 설계

```
create table tbl_reply (
  rno number(10,0),
  bno number(10,0) not null,
  reply varchar2(1000) not null,
  replyer varchar2(50) not null,
  replyDate date default sysdate,
);

create sequence seq_reply;

alter table tbl_reply add constraint pk_reply primary key (rno);

alter table tbl_reply  add constraint fk_reply_board
foreign key (bno)  references  tbl_board (bno);
```



식별키(PK) 지정

외래키(FK) 지정

ReplyVO클래스의 추가/Mapper 준비

```java
@Data
public class ReplyVO {
  private Long rno;
  private Long bno;
  private String reply;
  private String replyer;
  private Date replyDate;
}
```

```java
public interface ReplyMapper {


}
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.pgm.webboard.mapper.ReplyMapper">


</mapper>
```

Mapper 작업

- ReppleyMapper interface

```
@Mapper
public interface ReplyMapper {
    public int insert(ReplyVO vo);
    public ReplyVO read(int rno);
    public int update(ReplyVO vo);
    public int delete(int bno);
    public List<ReplyVO> getReplyList(int bno);
}
```

Mapper 작업

- ReplyMapper.xml

```
<insert id="insert">
    insert into tbl_reply (bno, reply, replyer)  values (#{bno}, #{reply}, #{replyer})
</insert>
<select id="read" resultType="com.pgm.webboard.mapper.ReplyVO">
    select * from tbl_reply where rno=#{rno}
</select>
<select id="getReplyList" resultType="com.pgm.webboard.mapper.ReplyVO">
    select * from tbl_reply where bno=#{bno} order by rno desc
</select>
<update id="update">
    update tbl_reply set reply=#{reply},replyer=#{replyer} where rno=#{rno}
</update>
<delete id="delete">
    delete from tbl_reply where rno=#{rno}
</delete>
```

서비스영역과 컨트롤러 처리

- ReplyService 인터페이스와 ReplyServiceImpl 클래스 생성

```
public interface ReplyService {
  public int register(ReplyVO vo);
  public int remove(int rno);
  public List<ReplyVO> getList(int bno);
}
```

```
@Service
@Log
public class ReplyServiceImpl implements ReplyService {
  @Autowired
  private ReplyMapper mapper;
  @Override
  public int register(ReplyVO vo) {
    log.info("register....." + vo);
    return mapper.insert(vo);
  }
..생략...
```

ReplyController의 설계

작업	URL	HTTP 전송방식
등록	/replies/new	POST
조회	/replies/:rno	GET
삭제	/replies/:rno	DELETE
수정	/replies/:rno	PUT or PATCH
페이지	/replies/pages/:bno/:page	GET

```
@RequestMapping("/replies/")
@RestController
@Log
@AllArgsConstructor
public class ReplyController {
   private ReplyService service;
}
```

등록작업과 테스트

- 댓글 등록의 경우 브라우저에서는 JSON 타입으로 된 댓글 데이터를 전송하고, 서버에서는 댓글의 처리 결과가 정상적으로 되었는지 문자열로 결과를 알려 주는 방식으로 처리

```
@PostMapping(value = "/new", consumes = "application/json")
  public ResponseEntity<String> create(@RequestBody ReplyVO vo) {
    log.info("ReplyVO: " + vo);
    int insertCount = replyService.register(vo);
    log.info("Reply INSERT COUNT: " + insertCount);
    return insertCount == 1 ? new ResponseEntity<String>("success", HttpStatus.OK)
        : new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
  }
```

특정 게시물의 댓글 목록

```
@GetMapping(value = "/pages/{bno}")
  public ResponseEntity<List<ReplyVO>> getList(@PathVariable("bno") Long bno) {
    Log.info("getList.................");
    return new ResponseEntity<List<ReplyVO>>(replyService.getList(bno), HttpStatus.OK);
  }
```

댓글의 삭제/조회

```
@GetMapping(value = "/{rno}")
public ResponseEntity<ReplyVO> get(@PathVariable("rno") Long rno) {
   log.info("get: " + rno);
   return new ResponseEntity<ReplyVO>(replyService.get(rno), HttpStatus.OK);
 }


 @DeleteMapping(value= "/{rno}")
 public ResponseEntity<String> remove(@PathVariable("rno") Long rno) {
   log.info("remove: " + rno);
   return service.remove(rno) == 1 ? new ResponseEntity<String>("success", HttpStatus.OK)
       : new ResponseEntity<String>(HttpStatus.INTERNAL_SERVER_ERROR);
 }
```

댓글의 수정

```
@PutMapping(value = "/{rno}", consumes = "application/json")
 public ResponseEntity<String> modify(@RequestBody ReplyVO vo, @PathVariable("rno") int rno) {
   vo.setRno(rno);

   Log.info("rno: " + rno);
   Log.info("modify: " + vo);

   return replyService.modify(vo) == 1 ? new ResponseEntity<String>("success", HttpStatus.OK)
       : new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);

 }
```