



[한국ICT인재개발원] 스프링 프레임워크

8. 스프링의 rest 방식 서버

前) 광고데이터 분석 1년

前) IT강의 경력 2년 6개월

前) 머신러닝을 활용한 데이터 분석 프로젝트반 운영 1년

前) 리그오브 레전드 데이터 분석 등...

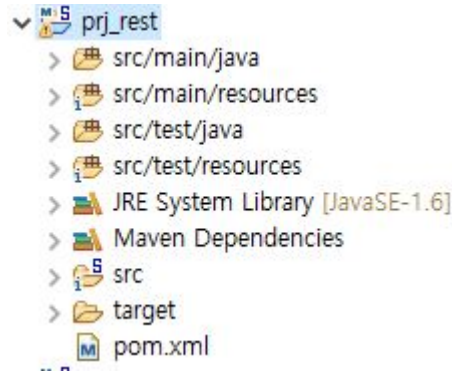
現) 국비반 강의 진행중

REST는 “Representational State Transfer의 약어로,
기존의 url에 파라미터를 붙여서 정보를 처리하는 대신
url자체가 1:1로 어떤 자원인지를 매칭해 나타내도록 설계하는 개념입니다.
원래 웹은 컴퓨터용 웹 브라우저 하나만을 고려해도 되었지만
스마트폰, 태블릿 등의 등장 이후로 어플리케이션, 브라우저와 더불어
여러 종류의 하드웨어까지 고려해야 하기 때문에 서버에서
모든 기기에서 통용될 수 있는 자료만 내주자는 흐름이 생겼습니다.



가령, url 자체를 구성할때 위와 같이 /단위로 나뉘었을때

/freedom은 자유게시판을 나타내고 /91055는 91055번 글을 나타내는 식입니다.



한국 ICT 인재개발원



[한국ICT인재개발원] 스프링 프레임워크
5. 스프링 CRUD와 테스트

위와같이 **Spring legacy project**를 생성해주세요.

프로젝트 이름은 **prj_rest**로 하겠습니다.

설정은 **05번** 교안을 따라서 해 주시면 되겠습니다.

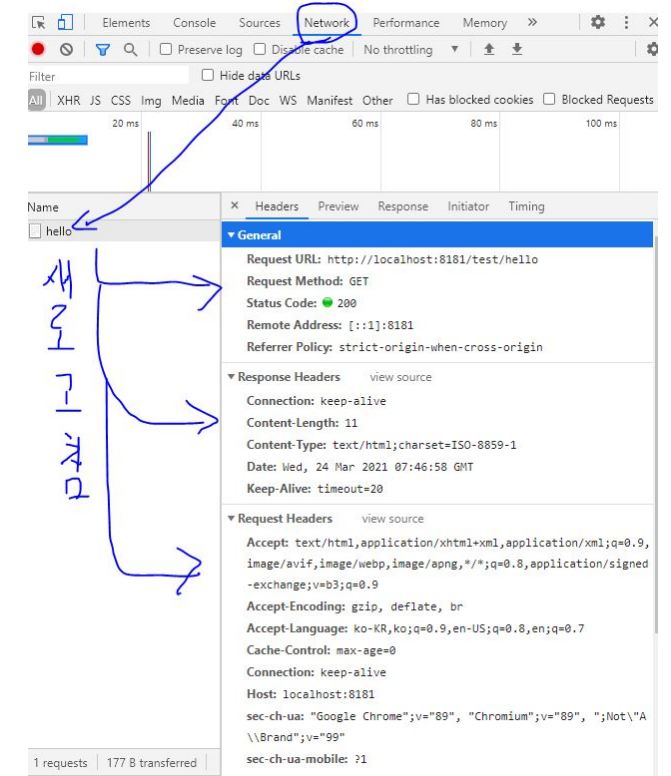
```
1 package org.ict.controller;
2
3 import org.springframework.web.bi
4 import org.springframework.web.bi
5
6 @RestController
7 @RequestMapping("/test")
8 public class TestController {
9
10 }
11
```

@RestController는 일반 컨트롤러와는 달리
이 컨트롤러가 rest방식으로 작동할 것이라는 선언을 하는 어노테이션입니다.
이제, 이 서버는 .jsp파일을 필요로 하지 않으며, 단순히 결과는
문자열, xml문서, json문서만을 화면에 표시합니다.

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping("/hello")
    public String sayHello() {
        return "Hello Hello";
    }
}
```

← → ↺ ⓘ localhost:8181/test/hello
Hello Hello



먼저, 단순 문자열을 다뤄보겠습니다.
메서드가 리턴을 String으로 하면 그 문자를 화면에 표출할 뿐입니다.
거기에 더해, F12를 눌러 개발자 도구를 열어 Network탭으로 이동한 다음
다시 새로고침을 하면 위와 같이 서버가 전송한 데이터에 대한 정보가
나와있습니다.

```
1 package org.ict.domain;
2
3 import lombok.Data;
4
5 @Data
6 public class TestVO {
7
8     private Integer mno;
9     private String Name;
10    private Integer age;
11 }
12
```

RestController에서 객체(vo등...)는 전부 json으로 변환됩니다.

먼저 테스트용 VO를 생성해보겠습니다.


```
@RequestMapping("/sendVO")  
public TestVO sendTestVO() {  
    TestVO testVO = new TestVO();  
  
    testVO.setName("채종훈");  
    testVO.setAge(21);  
    testVO.setMno(1);  
    return testVO;  
}
```

다음 컨트롤러에서 방금 생성한 TestVO를 리턴하는 메서드를 생성합니다.





1. Jackson Databind

com.fasterxml.jackson.core » [jackson-databind](https://com.fasterxml.jackson.core/jackson-databind)

General data-binding functionality for Jackson: works on core streaming API

Last Release on Mar 3, 2021



Jackson Databind » 2.12.2

General data-binding functionality for Jackson: works on core streaming API

License	Apache 2.0
Categories	JSON Libraries
HomePage	http://github.com/FasterXML/jackson
Date	(Mar 03, 2021)
Files	bundle (1.4 MB) View All
Repositories	Central
Used By	18,014 artifacts

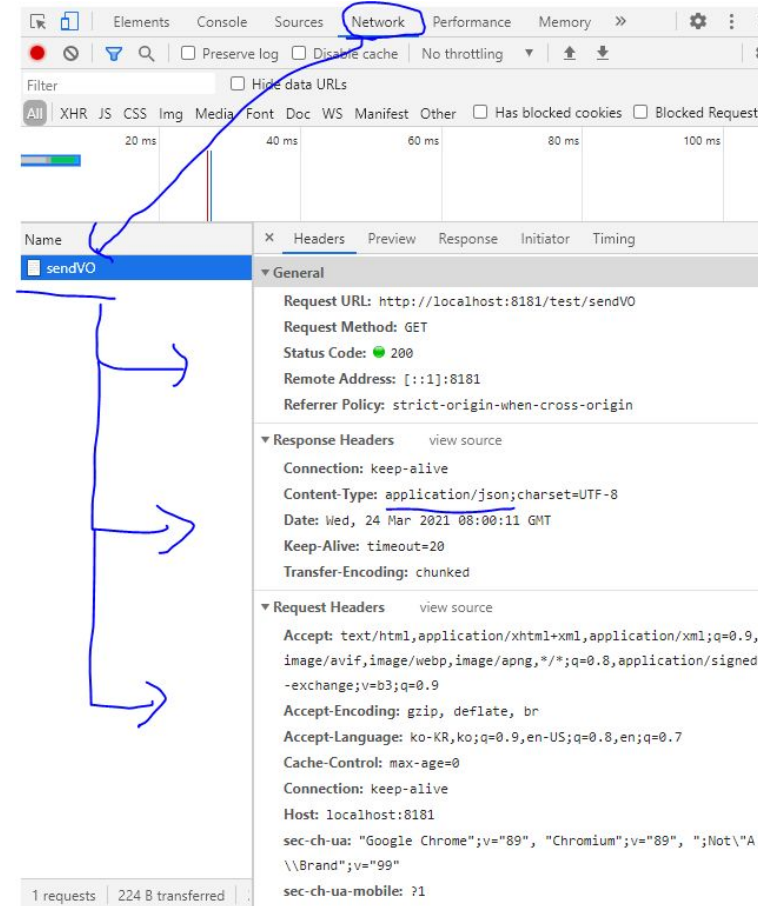
[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.12.2</version>
</dependency>
```

따라서, mvnrepository에서 jackson-databind 라이브러리를 새로 pom.xml에 추가해줍니다.

← → ↻ ⓘ localhost:8181/test/sendVO

```
{"mno":1,"age":21,"name":"채종훈"}
```



jackson-databind를 추가해 json데이터를 표출할 수 있게 처리했다면 다시 서버를 구동해주시고, 해당 주소에 접속합니다. 그러면 위와 같이 객체를 json형태로 바꿔서 출력해줍니다. 또한 아까의 text와는 달리 json타입을 리턴했음을 명시해주고 있습니다.

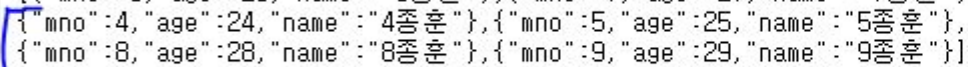
```
@RequestMapping("/sendVoList")
public List<TestVO> sendVoList() {

    List<TestVO> list = new ArrayList<>();
    for (int i=0; i<10; i++) {
        TestVO vo = new TestVO();
        vo.setMno(i);
        vo.setName(i + "종훈");
        vo.setAge(20 + i);
        list.add(vo);
    }
    return list;
}
```

이번에는 객체 리스트(List)는 어떻게 처리되는지 보겠습니다.

메서드 sendVoList()를 만듭니다.

```
[{"mno":0,"age":20,"name":"0종훈"}, {"mno":1,"age":21,"name":"1종훈"}, {"mno":2,"age":22,"name":"2종훈"}, {"mno":3,"age":23,"name":"3종훈"},  
{"mno":4,"age":24,"name":"4종훈"}, {"mno":5,"age":25,"name":"5종훈"}, {"mno":6,"age":26,"name":"6종훈"}, {"mno":7,"age":27,"name":"7종훈"},  
{"mno":8,"age":28,"name":"8종훈"}, {"mno":9,"age":29,"name":"9종훈"}]
```



이렇게 전달받은 객체 리스트는 위와 같이 []로 감싸진 json객체의 집합으로 표시됩니다.

```
@RequestMapping("/sendMap")
public Map<Integer, TestVO> sendMap() {

    Map<Integer, TestVO> map = new HashMap<>();

    for(int i=0; i<10; i++) {
        TestVO vo = new TestVO();
        vo.setName("채종훈");
        vo.setMno(i);
        vo.setAge(50+i);
        map.put(i, vo);
    }
    return map;
}
```

원래 json처럼 표현되는 Map자료형 역시 그대로 표현할 수 있습니다.

```
{
  "0": {"mno": 0, "age": 50, "name": "채종훈"},
  "1": {"mno": 1, "age": 51, "name": "채종훈"},
  "2": {"mno": 2, "age": 52, "name": "채종훈"},
  "3": {"mno": 3, "age": 53, "name": "채종훈"},
  "4": {"mno": 4, "age": 54, "name": "채종훈"},
  "5": {"mno": 5, "age": 55, "name": "채종훈"},
  "6": {"mno": 6, "age": 56, "name": "채종훈"},
  "7": {"mno": 7, "age": 57, "name": "채종훈"},
  "8": {"mno": 8, "age": 58, "name": "채종훈"},
  "9": {"mno": 9, "age": 59, "name": "채종훈"}
}
```

이 경우, 객체 내부의 객체로 표현됩니다.
[]로 동등하게 나열하는 형태가 아님에 유의해주세요.

응답 번호대	설명	대표적 번호
100번대	현재 데이터 처리중임을 나타냅니다.	100(처리중)
200번대	정상적으로 응답이 완료되었음을 의미합니다.	200(응답 처리 완료)
300번대	기본에 있던 URL이나 현재는 변경되었음을 나타냅니다.	301(리다이렉트)
400번대	요청 URL이 존재하지 않음을 나타냅니다.	404(url 없음)
500번대	요청은 정상이나 서버 내부 문제로 처리되지 못함을 나타냅니다.	500(내부 로직 에러)

서버는 요청을 처리하며, 결과로 코드를 함께 전송합니다.
그리고 **ResponseEntity**를 사용하면 개발자가 의도한 타이밍에 의도한
응답을 사용자에게 전달할 수 있습니다.

```
@RequestMapping("/sendErrorAuth")
public ResponseEntity<Void> sendListAuth() {

    return
        new ResponseEntity<>(HttpStatus.BAD_REQUEST);
}
```

이제 접속하는 족족 강제로 400에러를 발생시키는 ResponseEntity<>를 전송해보겠습니다.

HttpStatus.BAD_REQUEST 와 같이 HttpStatus객체를 ResponseEntity 객체의 생성자로 넣어 return구문에 제공하면 됩니다.

응답 타입을 조절하는 ResponseEntity

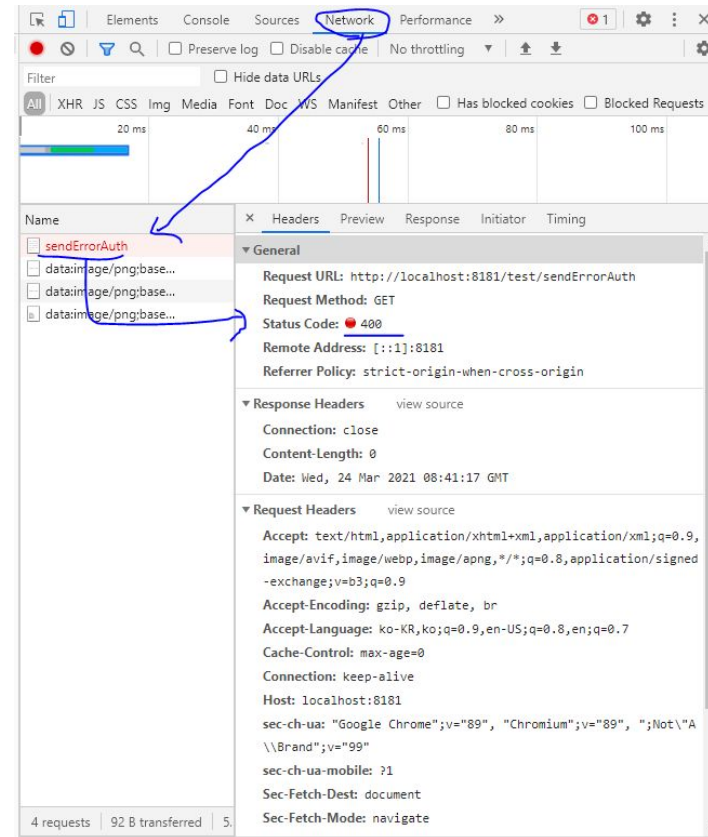


페이지가 작동하지 않습니다.

문제가 계속되면 사이트 소유자에게 문의하세요.

HTTP ERROR 400

새로고침



정확한 주소로 접속했음에도 불구하고 바로 400에러를 발생시키는것을 볼 수 있습니다.

마치 에러처리를 하듯 상황에 맞는 에러를 전달하면 됩니다.

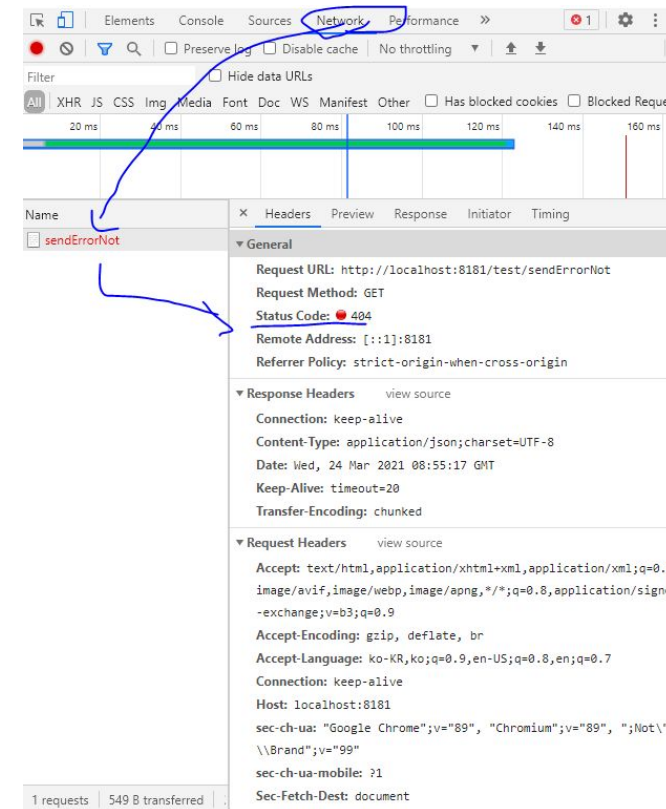
```
@RequestMapping("/sendErrorNot")
public ResponseEntity<List<TestVO>> sendListNot(){

    List<TestVO> list = new ArrayList<>();
    for (int i=0; i<10; i++) {
        TestVO vo = new TestVO();
        vo.setMno(i);
        vo.setName(i + "종훈");
        vo.setAge(20 + i);
        list.add(vo);
    }
    return
        new ResponseEntity<List<TestVO>>(
            list, HttpStatus.NOT_FOUND);
}
```

메세지만 전달할 수 있는것은 아닙니다.
요청 자료를 보내주면서도 동시에 에러를 발생시킬수도 있습니다.
ResponseEntity객체를 생성할 때 파라미터를 2개 넣어주면 됩니다.

응답 타입을 조절하는 ResponseEntity

```
[{"mno":0,"age":20,"name":"0종훈"}, {"mno":1,"age":21,"name":"1종훈"}, {"mno":2,"age":22,"name":"2종훈"}, {"mno":3,"age":23,"name":"3종훈"}, {"mno":4,"age":24,"name":"4종훈"}, {"mno":5,"age":25,"name":"5종훈"}, {"mno":6,"age":26,"name":"6종훈"}, {"mno":7,"age":27,"name":"7종훈"}, {"mno":8,"age":28,"name":"8종훈"}, {"mno":9,"age":29,"name":"9종훈"}]
```



위와 같이 데이터는 보여주며 404에러를 발생시킬수도 있습니다.

보통 호출한쪽에 에러의 원인과 함께 원인데이터를 제공할때 사용합니다.

방식	용도
get	데이터 조회
post	데이터 삽입
put	데이터 수정
delete	데이터의 삭제

원래 html에서는 **get, post**방식만 지정을 해서 데이터 처리에 사용할 수 있었지만, 사실 **http** 프로토콜은 크게 4가지 방식(**get, post, put, delete**) 요청을 처리할 수 있습니다.

URL부터 시작해 모든 부분을 **rest**방식으로 구성해 댓글서비스를 개발하겠습니다.

패턴	전송방식	설명
/replies/all/123	GET	게시물 123번의 모든 댓글
/replies/ + data	POST	댓글 등록
/replies/456 + 데이터	PUT/PATCH	456번 댓글 수정
/replies/456	DELETE	456번 댓글 삭제

원래 html에서는 get, post방식만 지정을 해서 데이터 처리에 사용할 수 있었지만, 사실 http 프로토콜은 약 4가지 방식(get, post, put/patch, delete) 요청을 처리할 수 있습니다.

URL부터 시작해 모든 부분을 rest방식으로 구성해 댓글서비스를 개발하겠습니다.

```
create table ictreply(  
  rno int not null auto_increment,  
  bno int not null default 0,  
  replytext varchar(1000) not null,  
  replyer varchar(100) not null,  
  regdate timestamp default now(),  
  updatedate timestamp default now(),  
  primary key(rno)  
);
```

```
alter table ictreply add constraint fk_board  
foreign key(bno) references ictboard(bno);
```

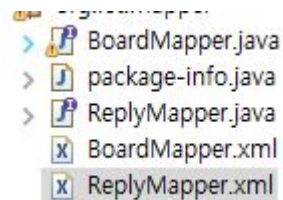
이제 댓글을 위한 테이블을 생성해야 합니다.

댓글은 글과 무조건 연동되기 때문에 이를 감안해, 외래 키까지 걸어주도록 하겠습니다.

위의 쿼리문을 실행해 테이블을 생성하고, ictboard와 연동시켜주세요.


```
1 package org.ict.domain;
2
3 import java.sql.Date;
4
5 import lombok.Data;
6
7 @Data
8 public class ReplyVO {
9
10     private int rno;
11     private int bno;
12     private String replytext;
13     private String replyer;
14
15     private Date regdate;
16     private Date updatedate;
17 }
18
```

다시 게시판 프로젝트로 돌아와주시고
domain에 이제 리플 관련 정보를 처리할 수 있는 VO를 생성합니다.



```
public interface ReplyMapper {  
    public List<ReplyVO> getList(int bno);  
    public void create(ReplyVO vo);  
    public void update(ReplyVO vo);  
    public void delete(int rno);  
}
```

```
1 <?xml version="1.0" encoding="UTF-8" ?>  
2 <!DOCTYPE mapper  
3   PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
4   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
5 <!-- 1~4번 라인은 xml 스키마 설정으로 굳이 외울 필요 없음. -->  
6  
7 <mapper namespace="org.ict.mapper.ReplyMapper">  
8  
9 </mapper>  
10
```

댓글 관련해서는 새로운 mapper, service를 다시 구성합니다.
따라서 ReplyMapper 인터페이스, 구현xml을 생성합니다.

```
<select id="getList"
  responseType="org.ict.domain.ReplyVO">
  SELECT * FROM ictreply
    WHERE bno = #{bno}
    ORDER BY rno DESC
</select>
```

```
<update id="update">
  UPDATE ictreply
    SET
      replytext = #{replytext}, updatedate = now()
    WHERE rno = #{rno}
</update>
```

```
<insert id="create">
  INSERT INTO ictreply (bno, replytext, replyer)
    VALUES (#{bno}, #{replytext}, #{replyer})
</insert>
```

```
<delete id="delete">
  DELETE FROM ictreply
    WHERE rno = #{rno}
</delete>
```

구현 메서드 내부 실행 쿼리문은 위와 같이 작성합니다.

```
public interface ReplyService {  
  
    public void addReply(ReplyVO vo);  
  
    public List<ReplyVO> listReply(int bno);  
  
    public void modifyReply(ReplyVO vo);  
  
    public void removeReply(int rno);  
}
```

```
public class ReplyServiceImpl implements ReplyService {  
  
    @Autowired  
    private ReplyMapper mapper;  
  
    @Override  
    public void addReply(ReplyVO vo) {  
        mapper.create(vo);  
    }  
  
    @Override  
    public List<ReplyVO> listReply(int bno) {  
        return mapper.getList(bno);  
    }  
  
    @Override  
    public void modifyReply(ReplyVO vo) {  
        mapper.update(vo);  
    }  
  
    @Override  
    public void removeReply(int rno) {  
        mapper.delete(rno);  
    }  
}
```

이번엔 서비스를 마저 구현합니다.

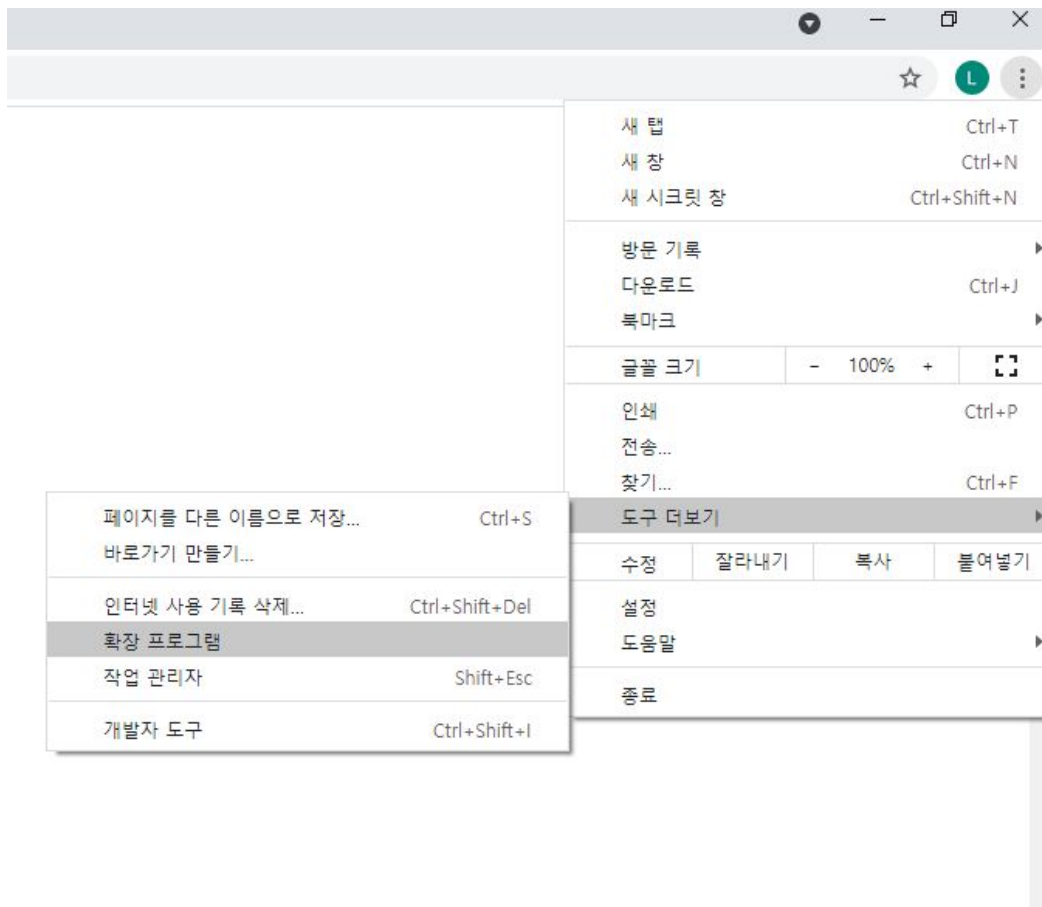
```
1 package org.ict.controller;
2
3 import org.springframework.web.bind.annotation.*;
4
5 @RestController
6 @RequestMapping("/replies")
7 public class ReplyController {
8     |
9 }
10
```

이제 본격적으로 **mapper**와 **service**를 제어할 수 있는 컨트롤러를 활용해보겠습니다. 컨트롤러를 생성해주세요.

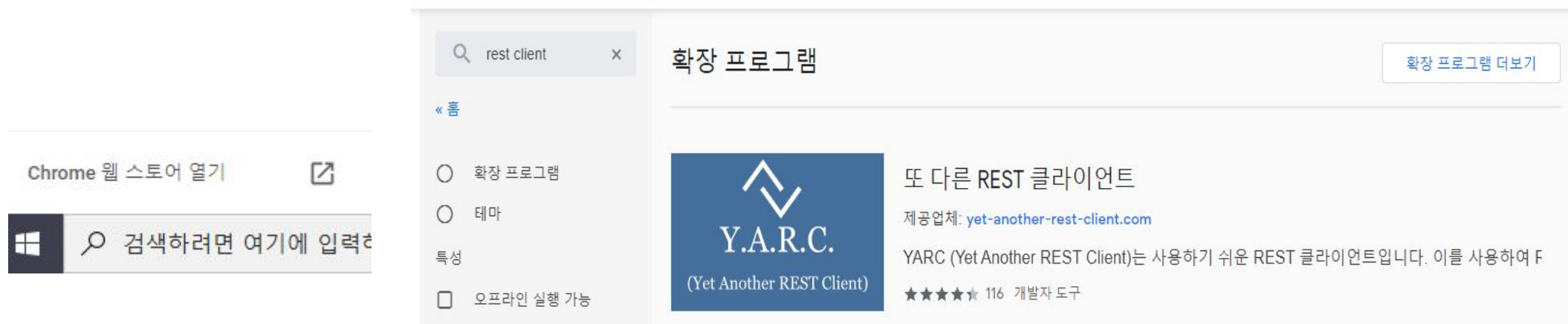
```
@RestController
@RequestMapping("/replies")
public class ReplyController {

    @Autowired
    private ReplyService service;
```

컨트롤러가 기능을 사용할 수 있도록 서비스 객체도 내부에 생성합니다.



컨트롤러에 대한 rest 방식 요청을 하기 위해 크롬창의 메뉴->도구 더보기 -> 확장프로그램에 들어갑니다.



메뉴바를 열어 좌측하단의 Chrome 웹 스토어를 열고
rest client 를 검색 후 Y.A.R.C를 클릭해 다운받습니다.



또 다른 REST 클라이언트

제공업체: yet-another-rest-client.com

★★★★★ 116 | [개발자 도구](#) | 사용자 50,000+명



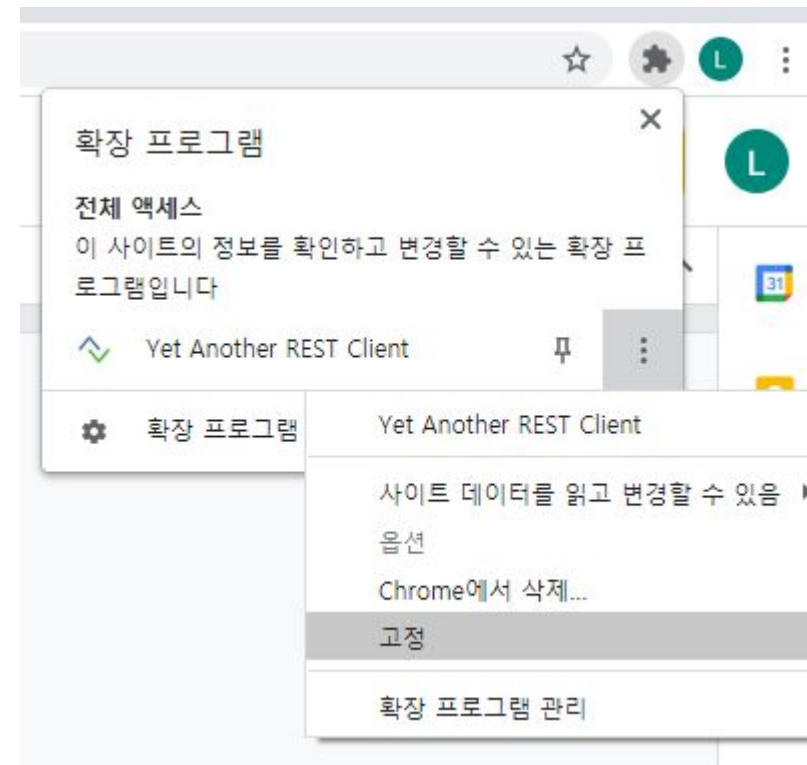
[개요](#)

[개인정보 보호관행](#)

[리뷰](#)

[지원](#)

[관련 프로그램](#)



Chrome에 추가 후, 우상단의 퍼즐조각 버튼을 클릭하고 **ARC cookie exchange**를 클릭해 실제로 사용할 수 있는 상태로 만듭니다.

YARC! v1.1.5 Yet Another REST Client

Main Favorites 0 History About Help

Request Settings

URL: ☆ http://www.example.com/resource/123 GET

Payload: application/json

Custom Headers

Authentication

Send Request

고정된 아이콘 클릭시 위와같이 어떤 주소에 어떤 메서드로 요청할지를 정할 수 있습니다.

이를 토대로 put/patch, delete 요청을 테스트할 수 있습니다.

```
@PostMapping(value="", consumes="application/json",
              produces= {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> register(
    @RequestBody ReplyVO vo){

    ResponseEntity<String> entity = null;
    try{
        service.addReply(vo);
        entity = new ResponseEntity<String>(
            "SUCCESS", HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<String>(
            e.getMessage(), HttpStatus.BAD_REQUEST);
    }
    return entity;
}
```

먼저 POST 방식의 글 추가 패턴부터 보겠습니다.

consumes는 전달할 데이터의 형태를 저장하는것으로 json인 경우 위와같이 적습니다.

produce 역시 json데이터 처리시 위와 같이 작성합니다.



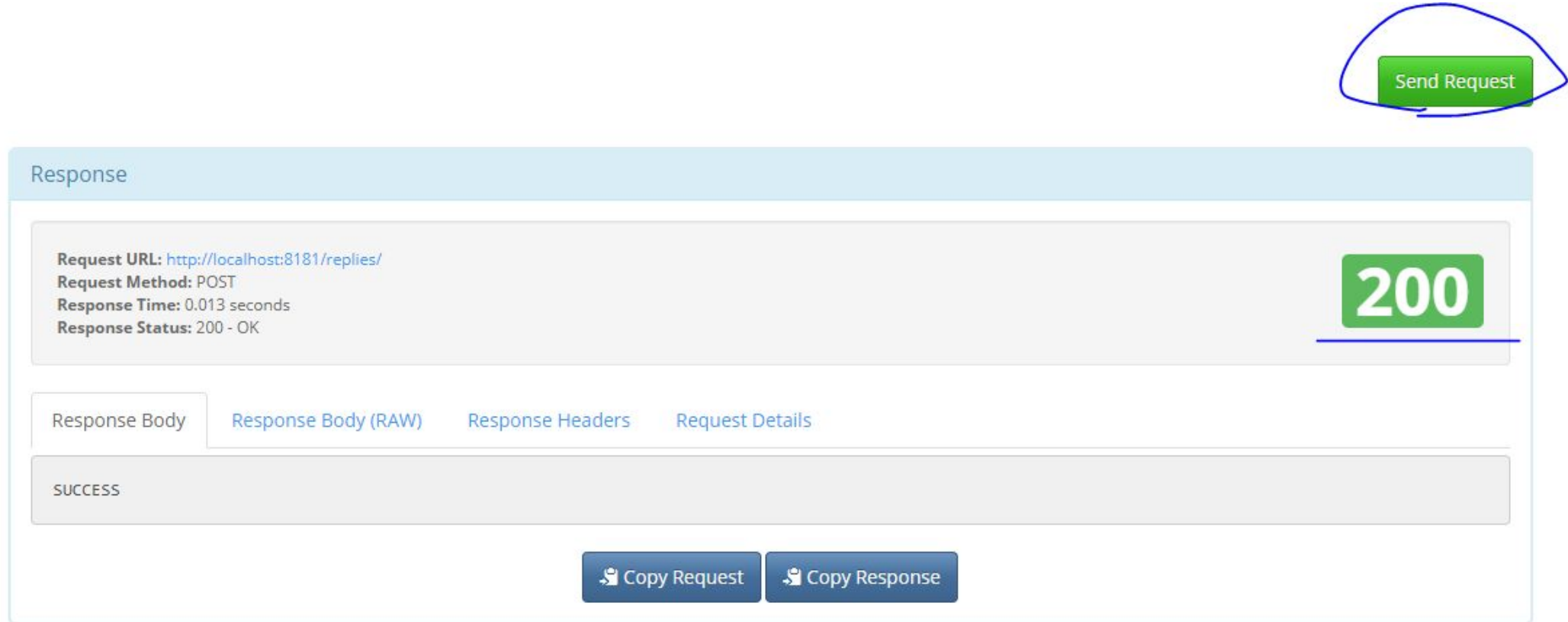
The screenshot shows a REST client window titled "Request Settings". It contains two main sections: "URL:" and "Payload:". The "URL:" field has a star icon and the text "http://localhost:8181/replies/". The "Payload:" field contains a JSON object: {"bno": "4", "replytext": "댓글인데요", "replyer": "댓글임"}. The "replytext" and "replyer" fields are underlined in red. To the right of the "URL:" field is a dropdown menu with "POST" selected and a downward arrow. A blue circle is drawn around the "POST" dropdown.

Request Settings

URL: ☆ http://localhost:8181/replies/ POST ▾

Payload: {"bno": "4", "replytext": "댓글인데요", "replyer": "댓글임"}

그리고 크롬창에서 위와 같이 글 추가 URL과 payload에는 추가할 댓글의 정보를 입력한 뒤, POST방식을 체크하면...
(bno는 반드시 ictboard에 있는 번호로 넣으며, 문자열로 처리합니다.)



위와 같이 200코드가 반환되며, mysql workbench에서 icreply 테이블에 대한 조회를 했을때 적은 내용이 삽입된것을 확인할 수 있습니다.

```
@PostMapping(value="/all/{bno}",
              produces = {MediaType.APPLICATION_XML_VALUE,
                           MediaType.APPLICATION_JSON_UTF8_VALUE})
public ResponseEntity<List<ReplyVO>> list(
    @PathVariable("bno") Integer bno){

    ResponseEntity<List<ReplyVO>> entity = null;

    try {
        entity = new ResponseEntity<>(
            service.listReply(bno), HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }

    return entity;
}
```

다음은 전체 글 목록을 요청하는 list 메서드를 생성합니다.

이 메서드의 파라미터에는 @PathVariable이라는 어노테이션이 있는데 이는 상단의 {bno}자리에 들어온 요소는 ?bno= 과 같이 간주한다는 의미입니다.

이번에는 produces를 확장해 XML, JSON을 모두 다루게 해보겠습니다.



Request Settings

URL: ☆ http://localhost:8181/replies/all/3 GET

Payload: application/json

Custom Headers

Headers: Accept: application/json Add Selected Header Add New Header

Header Name	Header Value	Actions
Content-Type	application/json	 

The above header(s) will be added to the next request.

Request URL: http://localhost:8181/replies/all/3
Request Method: GET
Response Time: 0.226 seconds
Response Status: 200 - OK

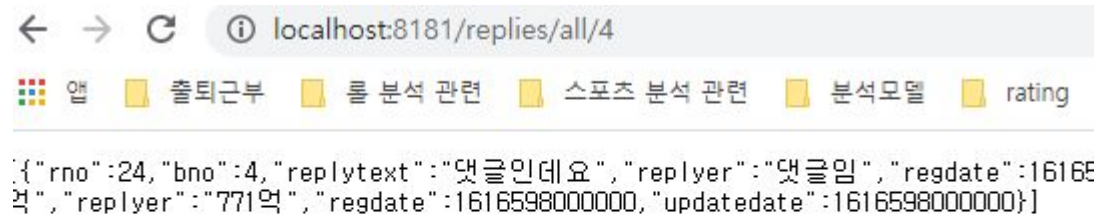
200

Response Body

Response Body (RAW) Response Headers Request Details

```
{
  {
    "rno": 5,
    "bno": 3,
    "replytext": "댓글시험",
    "replyer": "시험글쓰미",
    "regdate": 1616598000000,
    "update": 1616598000000
  },
  {
    "rno": 4,
    "bno": 3,
    "replytext": "댓글시험",
    "replyer": "시험글쓰미",
    "regdate": 1616598000000,
    "update": 1616598000000
  },
  {
    "rno": 3,
    "bno": 3,
    "replytext": "댓글시험",
    "replyer": "시험글쓰미",
    "regdate": 1616598000000,
    "update": 1616598000000
  },
  {
    "rno": 2,
    "bno": 3,
    "replytext": "댓글시험",
    "replyer": "시험글쓰미",
    "regdate": 1616598000000,
    "update": 1616598000000
  }
}
```

위와 같이 요청을 하면, 해당 글(3)에 있던 값이 bno로 전달되고, 전달된 값을 받아 200 응답을 하며 데이터가 출력됩니다.



단 get방식은 브라우저에서도 조회가 가능하기에 위와 같이 브라우저상에서 조회해서 확인할 수도 있습니다.

기본적으로는 xml로 나오지만, 만약 json타입으로 보고싶다면 url뒤에 .json을 추가합니다.


```
// 일반 방식이 아닌 rest방식에서는 삭제로직을 post가 아닌
// delete 방식으로 요청하기 때문에 @DeleteMapping
@DeleteMapping(value="/{rno}",
    produces = {MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> remove(
    @PathVariable("rno") int rno){
    ResponseEntity<String> entity = null;
    try {
        service.removeReply(rno);
        entity =
            new ResponseEntity<String>(
                "SUCCESS", HttpStatus.OK);
    } catch (Exception e ) {
        e.printStackTrace();
        entity =
            new ResponseEntity<>(
                e.getMessage(), HttpStatus.BAD_REQUEST);
    }
    return entity;
}
```

삭제로직은 위와 같이 작성하며, delete 요청을 처리하기 위해 @DeleteMapping을 걸어서 로직을 정의합니다.

Response

Request URL: <http://localhost:8181/replies/23>
Request Method: DELETE
Response Time: 0.118 seconds
Response Status: 200 - OK

200

Response Body

Response Body (RAW)

Response Headers

Request Details

SUCCESS

Copy Request

Copy Response

삭제로직 역시 DELETE를 받아 리플번호를 입력받으면 삭제하도록 합니다.
성공시 200코드와 SUCCESS를 보여줍니다.
삭제 후 실제로 삭제되었는지 DB를 체크해보세요.

```
@RequestMapping(method= {RequestMethod.PUT, RequestMethod.PATCH},
                  value="/{rno}",
                  consumes = "application/json",
                  produces = MediaType.TEXT_PLAIN_VALUE)
public ResponseEntity<String> modify(
    @RequestBody ReplyVO vo, @PathVariable("rno") int rno){

    ResponseEntity<String> entity = null;
    try {
        vo.setRno(rno);
        service.modifyReply(vo);

        entity = new ResponseEntity<String>("SUCCESS", HttpStatus.OK);
    } catch (Exception e) {
        e.printStackTrace();
        entity = new ResponseEntity<String>(
            e.getMessage(), HttpStatus.BAD_REQUEST);
    }
    return entity;
}
```

PUT, PATCH방식은 구분이 엄격하지 않기에 위와 같이 두 가지를 모두 포함시켜줍니다.

consumes에는 json을 입력받음을 명시하고, produces에는 문자열을 리턴한다고 명시합니다.

Payload:

```
{"replytext": "본문수정은 잘 됩니까?"}
```

Custom Headers >

Authentication >

Send Request

Response

Request URL: <http://localhost:8181/replies/24>
Request Method: PUT
Response Time: 0.194 seconds
Response Status: 200 - OK

200

Response Body

Response Body (RAW)

Response Headers

Request Details

SUCCESS

역시 수정이 되는지 확인해주시고, 200코드가 뜨면 DB쪽도 확인해주세요.