{ Rest API }

Representation State Transfer API

REST

- HTTP 통신 기반의 아키텍처 REST
  - REST는 Representational State Transfer의 약어로 하나의 URI는 하나의 고유한 리소스를 대표하도록 설계된다는 개념.
  - REST 방식은 특정한 URI는 반드시 그에 상응하는 데이터 자체라는 것을 의미.

    예를 들면
    /post/123은 해당 페이지의 번호의 123번이라는 고유한 의미를 가지고 설계하고, 이에 대한 처리는 GET, POST방식과 같이 추가적인 정보를 통해서 결정된다.
  - REST API는 외부에서 위와 같은 방식으로 특정 URI를 통해서 사용자가 원하는 정보를 제공하는 방식.
  - REST방식의 서비스 제공이 가능한 것을 Restful 하다고 표현.
  - 스프링 3부터 @ResponseBody 애노테이션을 지원하면서 REST방식의 처리를 지원하고 있었으며 스프링 4에 들어와서 @RestController가 본격적으로 적용.

  - 참고 : http://haah.kr/2017/05/22/rest-the-beginning/

REST 의 구성

- REST API는 다음의 구성으로 이루어져있습니다.
  - 자원(RESOURCE) - URI
  - 행위(Verb) - HTTP METHOD (GET, POST, DELETE, PUT)
  - 표현(Representations)

REST API 규칙

- URI는 정보의 자원을 표현해야 한다. (리소스명은 동사보다는 명사를 사용)

  GET /members/delete/1

- 위와 같은 방식은 REST를 제대로 적용하지 않은 URI. URI는 자원을 표현하는데 중점을 두어야 한다.
- delete와 같은 행위에 대한 표현이 들어가서는 안된다.
- 자원에 대한 행위는 HTTP Method(GET, POST, PUT, DELETE 등)로 표현
- 앞에서 본 잘못 된 URI를 HTTP Method를 통해 수정하면 아래와 같다.

  DELETE /members/1

REST API 규칙

- 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 않음.

    POST /members/2 (o)
  - HTTP METHOD의 알맞은 역할
  - POST, GET, PUT, DELETE 이 4가지의 Method를 가지고 CRUD를 할 수 있다.

REST API 규칙

- 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 수정합니다.
  - DELETE DELETE를 통해 리소스를 삭제합니다.

- 정리

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

URI 설계 시 주의할 점

- 슬래시 구분자(/)는 계층 관계를 나타내는 데 사용
  - http://restapi.example.com/houses/apartments
  - http://restapi.example.com/animals/mammals/whales
- URI 마지막 문자로 슬래시(/)를 포함하지 않는다.
  - http://restapi.example.com/houses/apartments/ (X)
  - http://restapi.example.com/houses/apartments (0)
- 하이픈(-)은 URI 가독성을 높이는데 사용
- 밑줄(_)은 URI에 사용하지 않는다.
- URI 경로에는 소문자가 적합하다.
  - URI 경로에 대문자 사용은 피하도록 해야 합니다.
- 파일 확장자는 URI에 포함시키지 않는다.
  - GET / members/soccer/345/photo HTTP/1.1 Host: restapi.example.com Accept: image/jpg

리소스 간의 관계 표현

- REST 리소스 간에는 연관 관계가 있는 경우 다음과 같이 표현

  /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 표현.
  예를 들어 사용자가 '좋아하는' 디바이스 목록을 표현해야 할 경우

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

자원을 표현하는 Colllection과 Document

- DOCUMENT는 단순히 문서로 이해해도 되고, 한 객체라고 이해.
- 컬렉션은 문서들의 집합, 객체들의 집합이라고 의미 부여 하면 됨.
- 컬렉션과 도큐먼트는 모두 리소스라고 표현하고 URI에 표현됨.

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

sports라는 컬렉션과 soccer라는 도큐먼트로 표현

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

sports, players 컬렉션과 soccer, 13(13번인 선수)를 의미하는 도큐먼트로 URI가 이루어짐.

컬렉션은 복수로 사용하고 있다.

좀 더 직관적인 REST API를 위해서는 컬렉션과 도큐먼트를 사용할 때 단수 복수도 지켜주면 좀 더 이해하기 쉬운 URI를 설계할 수 있다.

Spring Framework RESTful

@PathVariable을 이용한 경로 변수 처리

- ID 가 10인 회원의 정보를 조회하기 위해한 URL을 구성할 때 다음과 같은 방법으로 요청 URL 경로에 포함시킬 수 있다.

http://localhost/open/member/mypage/10

- 경로의 특정 위치 값이 고정되지 않고 달라질 때 사용하는 것이 @PathVariable

```
@RequestMapping("/mypage/mypage/{id}“)
public String mypageView(@PathVariable(“id”) Long memID, Model model) {
        Member member = dao.selectById(id);
        if ( memebr == null ) {
                throw new MemberNotFoundException();
        }
        return “mypage/mypage";
}
```

XML/JSON 변환 처리

- HTTP 기반 API 형태로 제공 하는 서비스
  - SNS 로그인 API
  - 공공데이터 포털에서 데이터 제공
  - SNS 에서 목록 (포스트, 친구 등) 제공
- 이들 API의 특징 중 하나가 **XML 이나 JSON 형식을 사용**
- Spring MVC 를 사용할 때 XML/JSON 응답 생성을 위한 뷰 클래스를 사용하거나 response 객체를 이용해서 응답 생성
- **@RequestBody** 애노테이션과 **@ResponseBody** 애노테이션을 사용하면 보다 쉽게 구현

@RequestBody / @ResponseBody , HttpMessageConverter

- @RequestBody 과 @ResponseBody는 요청몸체와 응답 몸체 구현
  - **@RequestBody**
    - JSON 형식의 요청 몸체를 자바 객체로 변환
  - **@ResponseBody**
    - 자바 객체를 JSON 이나 XML 형식의 문자열로 변환
  - Spring Framework는 HttpMessageConverter 를 이용해서 자바 객체와 HTTP 요청/응답 몸체 사이의 변환 처리

@RequestBody / @ResponseBody , HttpMessageConverter

SimpleConverterController.java

```java
package com.bitcamp.test.controller;


@Controller 
@RequestMapping("/mc/simple")
public class SimpleConverterController {
	@RequestMapping(method = RequestMethod.GET)
		public String simpleForm() {
		return "mc/simple";
	}
	@RequestMapping(method = RequestMethod.POST)
	// 메서드의 리턴 값을 HTTP의 응답 데이터로 사용
	@ResponseBody
	public String simple(@RequestBody String body) {
	// @RequestBody : 요청 HTTP 데이터를 String body 로 전달
		return body;
	}
}
```

@RequestBody / @ResponseBody , HttpMessageConverter

views/mc/simple.jsp

```
<%@ page contentType="text/html; charset=utf-8" %>
<!DOCTYPE html>
<html>
<head><title>단순 입력 폼</title></head>
<body>
<form method="POST">
    이름: <input type="text" name="name" /> <br/>
    나이: <input type="text" name="age" />
    <input type="submit" />
</form>

</body>
</html>
```

@RequestBody / @ResponseBody , HttpMessageConverter

- **JAXB2 를 이용한 XML 처리**
  - **JAXB2 API는 자바 객체와 XML 사이의 변환 처리를 해주는 API**
  - Jax2RootElementHttpMessageConverter 는 JAXB2 API를 이용해서 자바 객체를 XML 응답으로 변환 하거나 XML 요청 몸체를 자바 객체로 변환
  - **JAVA6 이후 버전 부터 기본으로 포함 : Maven 의존에 추가할 필요 없음**
  - Jax2RootElementHttpMessageConverter 는 아래와 같은 변환 처리 지원
    XML → @XmlRootElement 객체 또는 @XmlType 객체로 읽기
    @XmlRootElement 적용 객체 → XML로 쓰기

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessageList.java

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "message-list")
public class GuestMessageList {
        @XmlElement(name = "message") // 반환 타입 지정
        private List<GuestMessage> messages;
        public GuestMessageList() {
        }
        public GuestMessageList(List<GuestMessage> messages) {
        this.messages = messages;
        }
        public List<GuestMessage> getMessages() {
        return messages;
        }
}
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessage.java

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = { "id", "message", "creationTime" })
public class GuestMessage {

        private Integer id;
        private String message;
        private Date creationTime;

        public GuestMessage() {
        }
        public GuestMessage(Integer id, String message, Date creationTime) {
                this.id = id;
                this.message = message;
                this.creationTime = creationTime;
        }
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessage.java

```
    public Integer getId() {
        return id;
    }
    public String getMessage() {
        return message;
    }
    public Date getCreationTime() {
        return creationTime;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public void setCreationTime(Date creationTime) {
        this.creationTime = creationTime;
    }
}
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessageController.java

```
@Controller
public class GuestMessageController {
        @RequestMapping(value = "/guestmessage/list.xml")
        @ResponseBody
        public GuestMessageList listXml() {
                return getMessageList();
        }

        @RequestMapping(value = "/guestmessage/post.xml",
                        method = RequestMethod.POST)
        @ResponseBody
        public GuestMessageList postXml(
                        @RequestBody GuestMessageList messageList) {
                return messageList;
        }
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessageController.java

```
        private GuestMessageList getMessageList() {
                List<GuestMessage> messages = Arrays.asList(
                new GuestMessage(1, "메시지", new Date()),
                new GuestMessage(2, "메시지2", new Date())  );

                return new GuestMessageList(messages);
        }
}
```

@RequestBody / @ResponseBody , HttpMessageConverter

- Jackson2를 이용한 JSON 처리
  - Jackson2 API는 자바 객체와 JSON 사이의 변환 처리를 해주는 API
  - MappingJackson2HttpMessageConverter 는 Jackson2 API를 이용해서 자바 객체를 JSON 응답으로 변환 하거나 JSON 요청 몸체를 자바 객체로 변환
  - Maven에 Jackson2 의존을 추가

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.11.0</version>
</dependency>
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessageController.java

```
@Controller
public class GuestMessageController {
        …
        @RequestMapping(value = "/guestmessage/list.json")
        @ResponseBody
        public GuestMessageList2 listJson() {
                return getMessageList2();
        }
        private GuestMessageList2 getMessageList2() {
                List<GuestMessage> messages = Arrays.asList(
                new GuestMessage(1, "메시지", new Date()),
                new GuestMessage(2, "메시지2", new Date()));
                return new GuestMessageList2(messages);
        }
}
```

@RequestBody / @ResponseBody , HttpMessageConverter

GuestMessageList2.java

```
import java.util.List;


public class GuestMessageList2 {


        private List<GuestMessage> messages;


        public GuestMessageList2(List<GuestMessage> messages) {
                this.messages = messages;
        }


        public List<GuestMessage> getMessages() {
                return messages;
        }
}
```

1. @RestController 소개

- 스프링 4 부터 @RestController 애노테이션의 경우 기존의 특정한 JSP와 같은 뷰를 만들어 내는 것이 아닌 REST방식의 데이터 자체를 서비스하는 것을 말한다.
- 스프링 3에는 해당 메소드의 리턴 타입에 @ResponseBody 애노테이션을 추가하는 형태로 작성.
- 기능은 달라진 것이 없지만, 컨트롤러 자체의 용도를 지정한다는 점에서 변화.
  - URI가 원하는 리소스를 의미한다.(복수형으로 작성)
  - URI에는 식별할 수 있는 데이터를 같이 전달하는 것이 일반적이다.

```
// 컨트롤러 위에 어노테이션을 붙혀서 사용한다.
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/sample")
public class RestControllerExampleController {
}
```

- @RestController는 JSP와 같은 뷰를 만들어 내지 않는 대신에 데이터 자체를 반환하는데, 이때 주로 사용되는 것은 **단순 문자열과 JSON, XML**, 등으로 나누어 볼 수 있다.

1. @RestController 소개

1) 단순문자열인 경우

1. 문자열 데이터는 기본적으로 브라우저에는 'text/html' 타입으로 처리된다.

```
@RestController
@RequestMapping("/sample")
public class RestControllerExampleController {
   public String sayHello() {
     return "Hello World";
   }
}
```

-> 해당 컨트롤러의 모든 뷰 처리가 JSP가 아니라는 것을 의미한다. 일반 문자열이 반환된다.

1. @RestController 소개

2) 객체를 JSON으로 반환하는 경우

1. 정보를 담을 VO객체 생성

```
public class MemberVO {

        private Integer id;
        private String name;
        private String email;
        private String photo;

        public Integer getId() {
                return id;
        }
}
```

1. @RestController 소개

2) 객체를 JSON으로 반환하는 경우

2. controller에 추가

```
@RestController
@RequestMapping("/sample")
public class RestControllerExampleController {
        @RequestMapping("/hello")
        public MemberVO sayHello() {
                MemberVO memberVO = new MemberVO();
                memberVO.setId(“1");
                memberVO.setName(“cooler");
                memberVO.setEmail(“cool@naver.com");
                memberVO.setPhoto(“cool.jpg”);
                return memberVO;
        }
}
```

2) 객체를 JSON으로 반환하는 경우

3. 라이브러리 추가

pom.xml에 스프링에서 객체를 json값으로 변경해 주는 라이브러리를 추가.

```
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.5.4</version>
</dependency>
```

1. @RestController 소개

2) 객체를 JSON으로 반환하는 경우

4. 결과 확인

```
{
        "id":1,
        "name":"cooler",
        "email":"cool@naver.com",
        "photo":"cool.jpg"
}
```

1. @RestController 소개

3) 컬렉션 타입의 객체를 반환하는 경우

- **LIST**

```
@RequestMapping("/hello")
public MemberVO sayHello() {
    List<MemberVO> list = new ArrayList<MemberVO>();
    list.add(new MemberVO (1, “cool1", “cool1@naver.com", “p1"));
    list.add(new MemberVO (2, "cool2", "cool2@naver.com", “p2"));
    list.add(new MemberVO (3, "cool3", "cool3@naver.com", “p3"));
    return sampleVO;
}
```

- **결과**

```
[
  {“id”:1, ”name”:”cool1”, “email”:”cool1@naver.com”, ”photo”:”p1”},
  {“id”:2, ”name”:”cool2”, “email”:”cool2@naver.com”, ”photo”:”p2”},
  {“id”:3, ”name”:”cool3”, “email”:”cool3@naver.com”, ”photo”:”p3”},
]
```

3) 컬렉션 타입의 객체를 반환하는 경우

- **MAP**

```
@RequestMapping("/hello")
public Map<Integer, MemberVO> sayHello() {
    List<MemberVO> list = new ArrayList<MemberVO>();
    list.add(new MemberVO(1, "cool1", "cool1@naver.com", "p1"));
    list.add(new MemberVO(2, "cool2", "cool2@naver.com", "p2"));
    list.add(new MemberVO(3, "cool3", "cool3@naver.com", "p3"));
    //Map파일생성
    Map<Integer, MemberVO> map = new HashMap<Integer, MemberVO>();
                for (int i = 0; i < list.size(); i ++) {
                            map.put(i, list.get(i));
                }
    return map;
}
```

- **결과**

```
{
  "0": {"id":1, "name":"cool1", "email":"cool1@naver.com", "photo":"p1"},
  "1": {"id":2, "name":"cool2", "email":"cool2@naver.com", "photo":"p2"},
  "2": {"id":3, "name":"cool3", "email":"cool3@naver.com", "photo":"p3"},
}
```

1. @RestController 소개

- @JasonIgnore : 응답 결과 제외
- @JsonFormat : 날짜 형식 변환

  (shape = Shape.STRING) //ISO-8601

  (yyyyMMddHHmmss)

2. @RequestBody로 JSON 요청 처리

```
@PostMapping("/api/members")
public void insertMember(
        @RequestBody  MemberVO member
) {
        ...
}
```

```
var arr = {id:1, name:'hot', email:'hot@naver.com', phpto:'photo.jpg'};
$.ajax({
    url: 'Ajax.do',
    type: 'POST',
    data: JSON.stringify(arr),
    contentType: 'application/json; charset=utf-8',
    dataType: 'json',
    async: false,
    success: function(msg) {
        alert(msg);
    }
});
```

마이크로 서비스 아키텍쳐의 이해

마이크로 서비스 아키텍쳐 (MSA의 이해)

조대협(http://bcho.tistory.com)

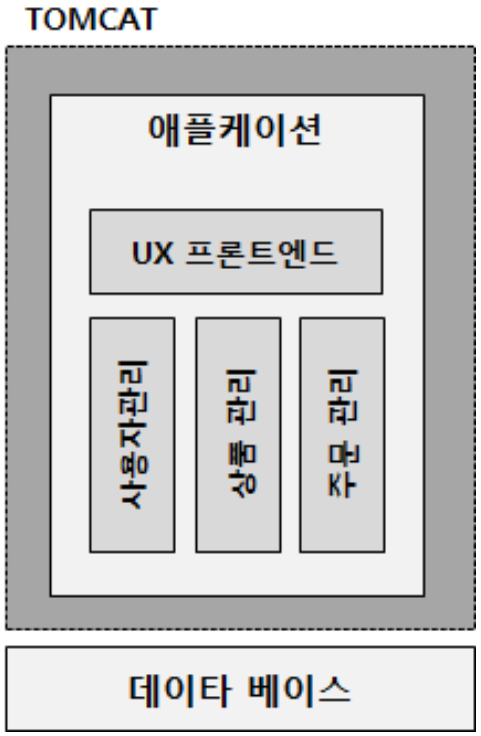
http://bcho.tistory.com/948

모노리틱 아키텍쳐(Monolithic Architecture)

- **모노리틱 아키텍쳐 스타일은 기존의 전통적인 웹 시스템 개발 스타일로, 하나의 애플리케이션 내에 모든 로직 들이 모두 들어 가 있는 "통짜 구조" 이다.**
  - 예를 들어, 온라인 쇼핑몰 애플리케이션이 있을 때, 톰캣 서버에서 도는 WAR 파일(웹 애플리케이션 패키징 파일)내에, 사용자 관리,상품,주문 관리 모든 컴포넌트들이 들어 있고 이를 처리하는 UX 로직 까지 하나로 포장되어서 들어가 있는 구조이다.

- 각 컴포넌트들은 상호 호출을 함수를 이용한 call-by-reference 구조를 취한다.

- 전체 애플리케이션을 하나로 처리하기 때문에, 개발 툴 등에서 하나의 애플리케이션만 개발하면 되고, 배포 역시 간편하며 테스트도 하나의 애플리케이션만 수행하면 되기 때문에 편리하다.



모노리틱 아키텍쳐(Monolithic Architecture)

- 문제점
  - **모노리틱 구조의 경우 작은 크기의 애플리케이션에서는 용이 하지만, 규모가 큰 애플리케이션에서는 불리한 점이 많다.**
    - 크기가 크기 때문에,
      빌드 및 배포 시간, 서버의 기동 시간이 오래 걸리며
    - 프로젝트를 진행하는 관점에서도,
      한 두 사람의 실수는 전체 시스템의 빌드 실패를 유발하기 때문에, 프로젝트가 커질 수록, 여러 사람들이 협업 개발하기가 쉽지 않다
    - 시스템 컴포넌트들이 서로 로컬 콜 (call-by-reference)기반으로 타이트하게 연결되어 있기 때문에,
      전체 시스템의 구조를 제대로 파악하지 않고 개발을 진행하면, 특정 컴포넌트나 모듈에서의 성능 문제나 장애가 다른 컴포넌트에까지 영향을 주게 되며, 이런 문제를 예방하기 위해서는 개발자가 대략적인 전체 시스템의 구조 등을 이해 해야 하는데, **시스템의 구조가 커질 수록, 개인이 전체 시스템의 구조와 특성을 이해하는 것은 어려워진다.**

모노리틱 아키텍쳐(Monolithic Architecture)

- 문제점
  - 특정 컴포넌트를 수정하고자 했을 때,
    컴포넌트 재 배포 시 수정된 컴포넌트만재 배포 하는 것이 아니라 전체 애플리케이션을 재 컴파일 하여 전체를 다시 통으로 재 배포 해야 하기 때문에 **잦은 배포가 있는 시스템의 경우 불리하며,**
  - **컴포넌트 별로, 기능/비 기능적 특성에 맞춰서 다른 기술을 도입하고자 할 때 유연하지 않다.**
    예를 들어서, 전체 애플리케이션을 자바로 개발했다고 했을 때, 파일 업로드/다운 로드와 같이 IO 작업이 많은 컴포넌트의 경우 node.js를 사용하는 것이 좋을 수 있으나, 애플리케이션이 자바로 개발되었기 때문에 다른 기술을 집어 넣기가 매우 어렵다.
  - ※ 모노리틱 아키텍쳐가 꼭 나쁘다는 것이 아니다.

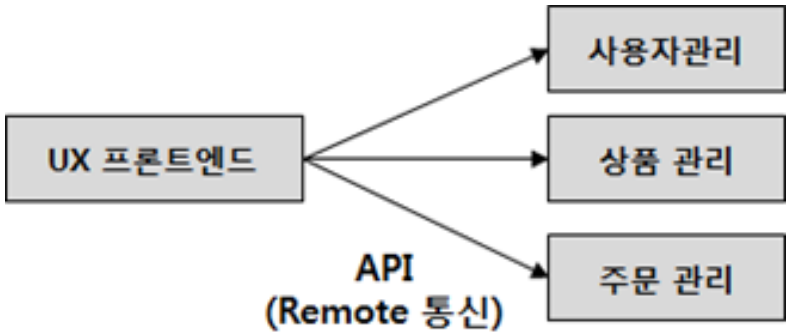
    규모가 작은 애플리케이션에서는 배포가 용이하고, 규모가 크더라도, call-by-reference call에 의해서 컴포넌트간 호출 시 **성능에 제약이 덜하며, 운영 관리가 용이하다.**
    또한 하나의 구조로 되어 있기 때문에, 트렌젝션 관리 등이 용이하다는 장점이 있다.

마이크로 서비스 아키텍쳐 (MSA)

- 마이크로 서비스 아키텍쳐는 SOA 사상에 근간을 두고, 대용량 웹 서비스 개발에 맞는 구조로 사상이 경량화 되고, 대규모개발팀의 조직 구조에 맞도록 변형된 아키텍쳐이다.
- 서비스
  - 마이크로 서비스 아키텍쳐에서는 각 컴포넌트를 서비스라는 개념으로 정의한다.
    서비스는 데이타에서 부터 비지니스 로직까지 독립적으로 상호 컴포넌트간의 의존성이 없이 개발된 컴포넌트로 **REST API**와 같은 표준 인터페이스로 그 기능을 외부로 제공한다.
  - 서비스 경계는 구문 또는 도메인(업무)의 경계를 따른다.
    예를 들어 사용자 관리, 상품 관리, 주문 관리와 같은 각 업무 별로 서비스를 나눠서 정의한다.
    사용자/상품 관리 처럼 여러 개의 업무를 동시에 하나의 서비스로 섞어서 정의하지 않는다.
  - REST API에서 /users,/products와 같이 주요 URI도 하나의 서비스 정의의 범위로 좋은 예가 된다.

마이크로 서비스 아키텍쳐 (MSA)

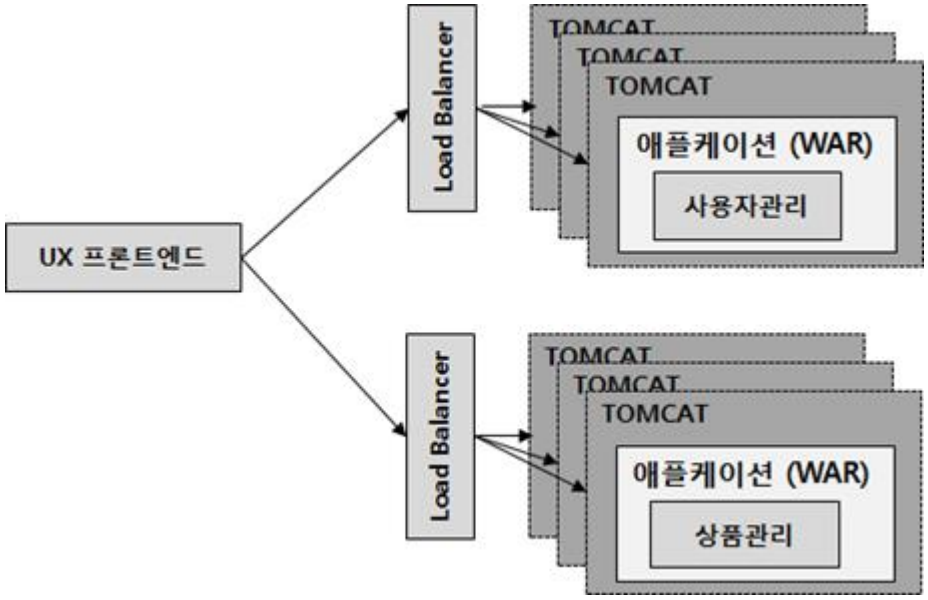
- 마이크로 서비스 아키텍쳐의 구조
  - 마이크로 서비스 아키텍쳐의 구조는 다음과 같은 모양을 따른다.



  - 각 컴포넌트는 서비스라는 형태로 구현되고 API를 이용하여 타 서비스와 통신을 한다.

마이크로 서비스 아키텍쳐 (MSA)

- 마이크로 서비스 아키텍쳐의 구조
  - 배포 구조관점에서도 **각 서비스는 독립된 서버로 타 컴포넌트와의 의존성이 없이 독립적으로 배포** 된다.
  - **예를 들어 사용자 관리 서비스는 독립적인 war파일로 개발되어, 독립된 톰캣 인스턴스에 배치된다.**
    확장을 위해서 서비스가 배치된 톰캣 인스턴스는 횡적으로 스케일 (인스턴스 수를 더함으로써)이 가능하고, 앞 단에 로드 밸런서를 배치하여 서비스간의 로드를 분산 시킨다.
  - 가장 큰 특징이, **애플리케이션 로직을 분리해서 여러 개의 애플리케이션으로 나눠서 서비스화하고, 각 서비스별로 톰캣을 분산 배치한 것이 핵심이다.**



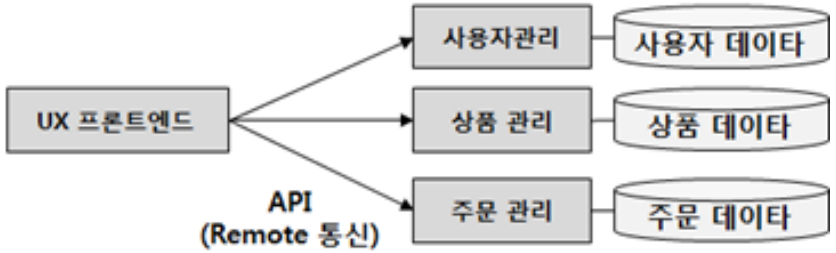
마이크로 서비스 아키텍쳐 (MSA)

- 데이터 분리
  - 데이타 저장 관점에서는 중앙 집중화된 하나의 통 데이타 베이스를 사용하는 것이 아니라 **서비스 별로 별도의 데이터베이스를 사용한다.**
  - 보통 모노리틱 서비스의 경우에는 하나의 통 데이타 베이스 (보통 RDBMS를 사용) 하는 경우가 일반적이지만, **마이크로 서비스 아키텍쳐의 경우, 서비스가 API에서 부터 데이타 베이스까지 분리되는 수직 분할 원칙 (Vertical Slicing)에 따라서 독립된 데이터베이스를 갖는다.**
  - **데이터베이스의 종류 자체를 다른 데이터베이스를 사용할 수 도 있지만, 같은 데이터베이스를 사용하더라도 데이터베이스를 나누는 방법을 사용한다.**
  - 이 경우, 다른 서비스 컴포넌트에 대한 의존성이 없이 서비스를 독립적으로 개발 및 배포/운영할 수 있다는 장점을 가지고 있으나,
    다른 컴포넌트의 데이터를 API 통신을 통해서만 가지고 와야 하기 때문에 성능상 문제를 야기할 수 있고, 또한 다른 벤더 종류의 데이터베이스간의 트렌젝션을 묶을 수 없는 문제점을 가지고 있다.

마이크로 서비스 아키텍쳐 (MSA)

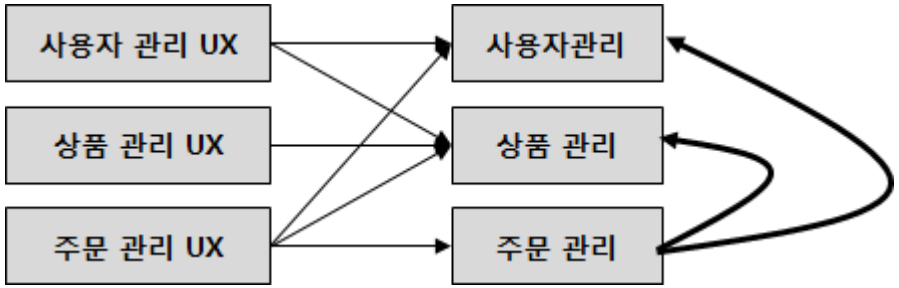
- 데이터 분리

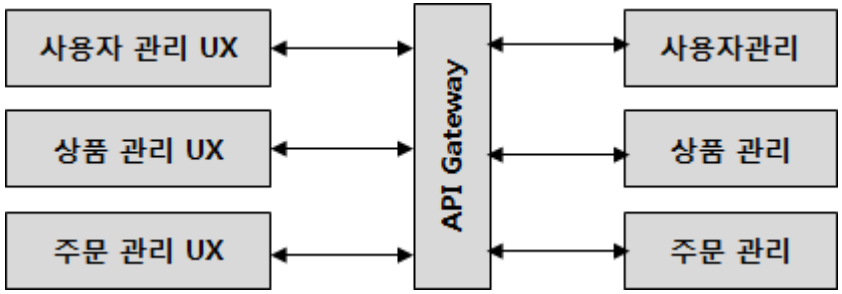
모노리틱 구조에서
데이타 저장 방식



마이크로 서비스 아키텍쳐에서
데이타 저장 방식

- API Gateway


사용자 관리 UX
상품 관리 UX
주문 관리 UX
사용자관리
상품 관리
주문 관리


사용자 관리 UX
상품 관리 UX
주문 관리 UX
API Gateway
사용자관리
상품 관리
주문 관리

마이크로 서비스 아키텍쳐 (MSA)

- 배포
  - 마이크로 서비스 아키텍쳐의 가장 큰 장점 중의 하나가 유연한 배포 모델이다.
    각 서비스가 다른 서비스와 물리적으로 완벽하게 분리되기 때문에 변경이 있는 서비스 부분만 부분 배포가 가능하다
    예를 들어서, 사용자 관리 서비스 로직이 변경되었을 때, 모노리틱 아키텍쳐의 경우에는 전체 시스템을 재 배포해야 하지만, 마이크로 서비스 아키텍쳐의 경우에는 변경이 있는 사용자 관리 서비스부분만 재 배포 하면 되기 때문에, 빠르고 전체 시스템의 영향도를 최소화한 수준에서 배포를 진행할 수 있다.

- **확장성**
  - 서비스 별로 독립된 배포 구조는 확장성에 있어서도 많은 장점을 가지고 오는데,부하가 많은 특정 서비스에 대해서만 확장이 가능하여 조금 더 유연한 확장 모델을 가질 수 있다.
    모노리틱 아키텍쳐의 경우에는 특정 서비스의 부하가 많아서 성능 확장이 필요할 때, 전체 서버의 수를 늘리거나 각 서버의 CPU 수를 늘려줘야 하지만, 마이크로 서비스 아키텍쳐의 경우에는 부하를 많이 받는 서비스 컴포넌트 만 확장을 해주면 된다.

마이크로 서비스 아키텍쳐 (MSA)

- **마이크로 서비스 아키텍쳐의 문제점**
  - **성능**
    - 마이크로 서비스 아키텍쳐는 서비스간의 호출을 API 통신을 이용하기 때문에 값을 json이나 xml에서 프로그래밍에서 사용하는 데이터 모델 (java object등)으로 변환하는 marsharing 오버헤드가 발생하고 호출을 위해서 이 메세지들이 네트워크를 통해서 전송되기 때문에 그만한 **시간이 더 추가로 소요된다.**
  - **메모리**
    - **마이크로 서비스 아키텍쳐는 각 서비스를 독립된 서버에 분할 배치하기 때문에, 중복되는 모듈에 대해서 그만큼 메모리 사용량이 늘어난다.**
    - 현대의 컴퓨팅 파워 자체가 워낙 발달하였고, 네트워크 인프라 역시 기존에 비해서 많은 성능상 발전이 있었다. 또한 메모리 역시 비용이 많이 낮춰지고 가용메모리 용량이 크게 늘어나서 큰 문제는 되지 않는다.
  - **테스팅이 더 어려움**
    - 마이크로 서비스 아키텍쳐의 경우 서비스들이 각각 분리가 되어 있고, 다른 서비스에 대한 종속성을 가지고 있기 때문에, 특정 사용자 시나리오나 기능을 테스트하고자 할 경우 여러 서비스에 걸쳐서 테스트를 진행해야 하는데 **테스트 환경 구축이나 문제 발생시 분리된 여러 개의 시스템을 동시에 봐야 하기 때문에 테스팅의 복잡도가 올라간다.**

마이크로 서비스 아키텍쳐 (MSA)

- 마이크로 서비스 아키텍쳐의 문제점
  - **운영 관점의 문제**
    - 운영 관점에서는 서비스 별로 서로 다른 기술을 사용할 수 있으며, 시스템이 아주 잘게 서비스 단위로 쪼게 지기 때문에 운영을 해야 할 대상 시스템의 개수가 늘어나고, 필요한 기술의수도 늘어나게 된다.
  - **서비스간 트렌젝션 처리**
    - 구현상의 가장 어려운 점 중의 하나가, 트렌젝션 처리이다. API 기반의 여러 서비스를 하나의 트렌젝션으로 묶는 것은 불가능하다.
    - 이러한 문제를 해결하기 위해서 몇 가지 방안이 있는데,
    - 그 첫 번째 방법으로는 아예 **애플리케이션 디자인 단계에서 여러 개의 API를 하나의 트렌젝션으로 묶는 분산 트렌젝션 시나리오 자체를 없애는 방안이다.**
      분산 트렌젝션이 아주 꼭 필요할 경우에는 차라리 모노리틱 아키텍쳐로 접근하는 것이 맞는 방법이다.
    - 그럼에도 불구하고, 트렌젝션 처리가 필요할 경우, **에러가 났을 경우에, 명시적으로, 데이터를 복구하는 에러처리 로직을 구현해야 한다.**

- **결론**
  - **마이크로 서비스 아키텍쳐는 대용량 웹시스템에 맞춰 개발된 API 기반의아키텍쳐 스타일이다.**
    대규모 웹서비스를 하는 많은 기업들이 이와 유사한 아키텍쳐 설계를 가지고 있지만, 마이크로 서비스 아키텍쳐가 무조건 정답은 아니다.
    하나의 설계에 대한 레퍼런스 모델이고, 각 업무나 비지니스에 대한 특성 그리고 팀에 대한 성숙도와 가지고 있는 시간과 돈과 같은 자원에 따라서 적절한 아키텍쳐 스타일이 선택되어야 하며, 또한 아키텍쳐는 처음 부터 완벽한 그림을 그리기 보다는 상황에 맞게 점진적으로 진화 시켜 나가는 모델이 바람직하다.