

The background of the image is a wide-angle photograph of a rural landscape. It features rolling hills covered in green and brown agricultural fields. Several white wind turbines are scattered across the hills, with one prominent one on the left side. The sky is a clear, vibrant blue. In the bottom right corner, there's a small, light blue rounded rectangular overlay containing the text.

RESTful Web Services

# 목 차

---

1. RESTful in SW Architecture
2. REST 아키텍처 스타일
3. 리소스 설계
4. RESTful API 설계
5. JAX-RS API
6. JAX-RS API 고급 기능
7. 안전한 RESTful 서비스
8. RESTful 서비스 서술과 찾기



# 1. RESTful in SW Architecture

---

- 1.1 SW 아키텍처 정의
- 1.2 SW 아키텍처 이해
- 1.3 연결 방식
- 1.4 연결 설계
- 1.5 사례로 보는 연결의 문제와 해결 방안
- 1.6 컴포넌트
- 1.7 컴포넌트 플랫폼

## 1.1 SW 아키텍처 정의

- ✓ SW [intensive] 시스템의 아키텍처 설계에 대한 정의는 IEEE 1471-2000 버전이 표준입니다.
- ✓ 이 정의로부터 아키텍처 설계 대상을 추려 보면, “컴포넌트”와 “컴포넌트 간의 연결”임을 알 수 있습니다.
- ✓ 환경, 원칙 등은 주요 대상인 컴포넌트 설계와 연결 설계를 할 때 고려사항이거나 준수할 내용입니다.
- ✓ 아키텍처 도메인에서는 이 두 가지를 컴포넌트 설계, 커넥터 설계라고 표현합니다.

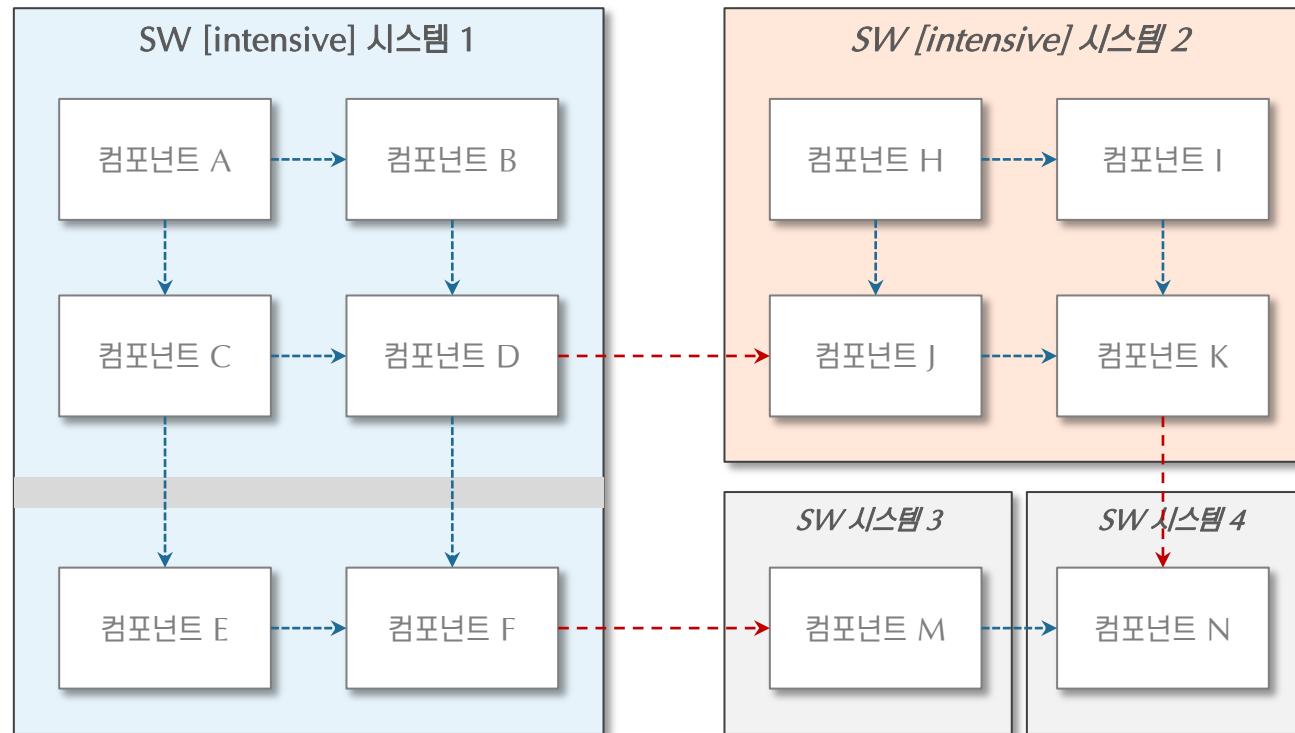
아키텍처는 시스템의 근간을 이루는 틀(fundamental organization)로써 시스템을 구성하는 컴포넌트, 컴포넌트 간의 [관계], 컴포넌트와 환경 간의 [관계], 그리고 설계와 개발을 다루는 원칙을 포함한다.

The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

IEEE 1471-2000

## 1.1 SW 아키텍처 정의

- ✓ SW 시스템에서 환경이란 다른 SW 시스템이나, HW 시스템, 더 나아가 운영 환경을 의미합니다.
- ✓ 앞에서 살펴 본 SW 아키텍처 정의를 바탕으로 아키텍처 설계를 표현해 보면 다음과 같습니다.
- ✓ 그것이 무엇인 컴포넌트를 정하여 획득(개발, 구매, 오픈소스)하고 다른 컴포넌트와 연결하는 것입니다.
- ✓ 연결은 지역(local) 연결이거나 원격(remote) 연결입니다.



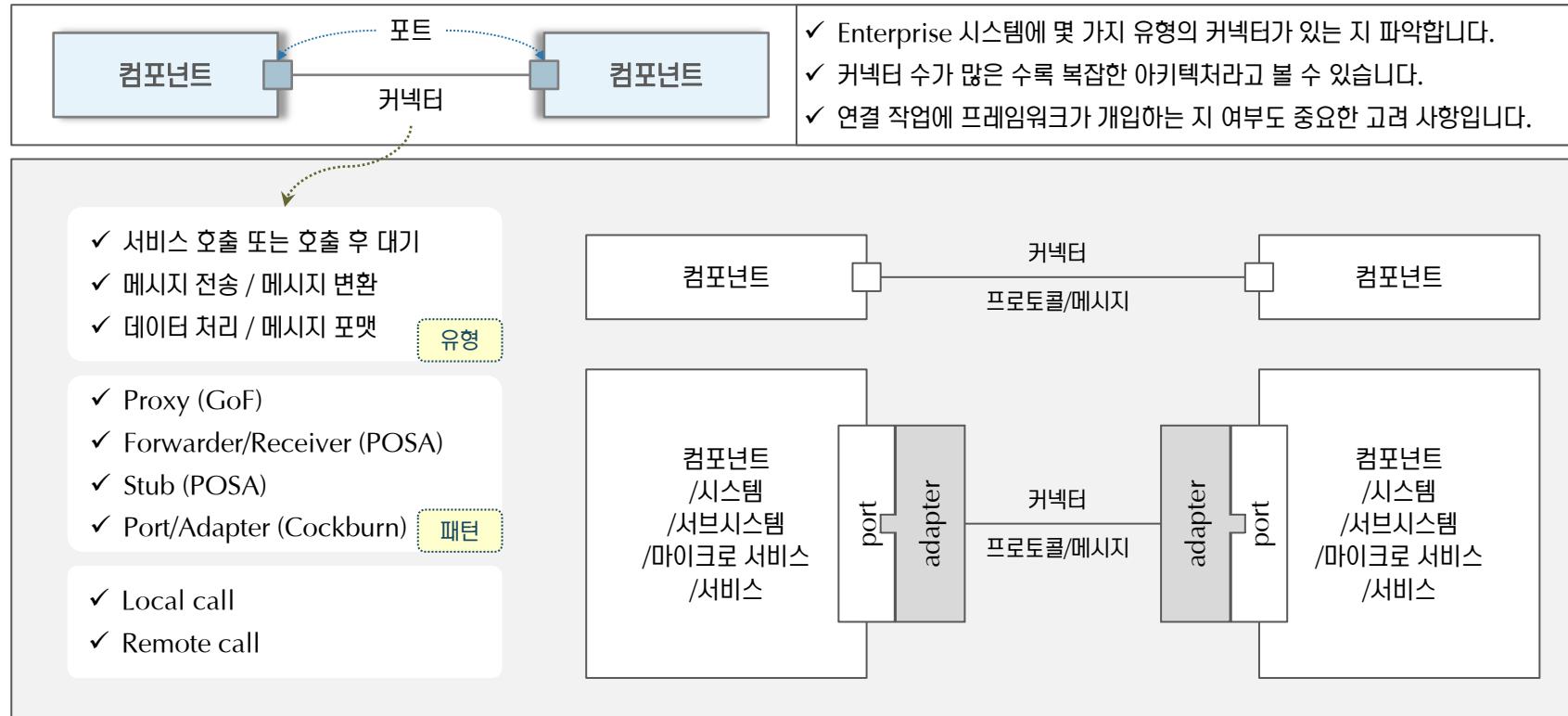
## 1.2 SW 아키텍처 이해: 컴포넌트

- ✓ IEEE 1471 SW 아키텍처 정의에서 언급한 (넓은 의미의) 컴포넌트를 이해하는 것이 중요합니다.
- ✓ 원래 CMU의 SEI 정의에서는 컴포넌트 대신에 “아키텍처 요소”라는 용어를 사용했습니다.
- ✓ 컴포넌트라는 용어는 혼란을 가져올 수 있습니다. 아키텍처 요소가 좀 더 이해하기 쉽고 명확합니다.
- ✓ 그럼 아키텍처 요소(또는 넓은 의미의 컴포넌트)에는 어떤 것들이 알아봅니다.

아키텍처 요소 타입 / 획득방법		자체개발	오픈소스	솔루션 벤더
비즈니스 컴포넌트	가입설계, 계약 컴포넌트	OMG Party컴포넌트		
	지연처리 프레임워크, 예외처리	Struts, myBATIS	T사 프레임워크	
	Mini-EAI서버, 상품 팩토리	Artifactory, SVN	T사의 암호화서버, IM/AM서버	
플랫폼 소프트웨어	운영체제	Linux	HP-UX, Sun Solaris	
	개발 언어	Physon, Groovy	Java, C, C++	
	실행 플랫폼	JBoss, Spring, JVM	WebLogic, PaaS	
	데이터 플랫폼	MariaDB, MongoDB, Redis	Oracle RDBMS	
	프로세스 플랫폼	OSWorkflow, uEngine	BPM	
	비즈니스 플랫폼	Compiere	SAP, Oracle ERP	
관리용 소프트웨어 (제외)		컴포넌트 관리, 배포관리		Tivoli, OpenView

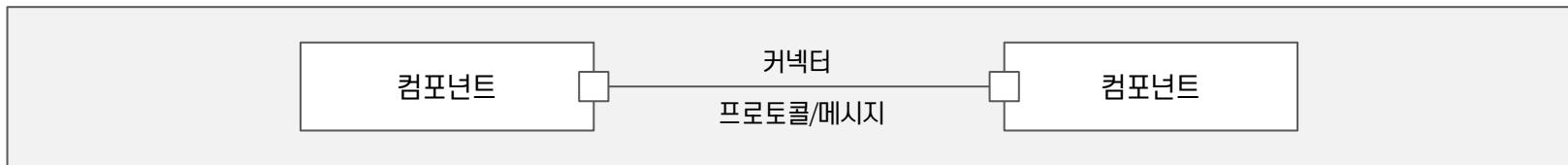
## 1.2 SW 아키텍처 이해: 연결(connector)

- ✓ 현대의 SW 시스템은 복잡성을 다루기 위해 시스템 협업을 하며, 여기서 커넥터는 중요한 컴포넌트입니다.
- ✓ 커넥터는 시스템(또는 서비스) 간의 연결 방식(프로토콜, 메시지, 프레임워크 등)을 결정합니다.
- ✓ Enterprise 시스템 설계에서, 커넥터는 검증된 기술을 사용하고 필요한 최소한의 유형을 갖도록 설계합니다.
- ✓ 컴포넌트, 포트, 커넥터, 어댑터는 모두 SW 아키텍처 도메인에서 사용하는 패턴(또는 스타일) 용어들입니다.



## 1.2 SW 아키텍처 이해: 연결(connector)

- ✓ 호출을 위한 연결은 지역(local) 호출과 원격(remote) 호출이 있습니다.
- ✓ 지역호출은 대상 컴포넌트가 제공하는 인터페이스를 호출합니다.
- ✓ 원격 호출은 다양한 방식이 있으며, 호출의 성격과 대상 컴포넌트의 특성, 개발환경 등에 따라 선택합니다.
- ✓ TCP 소켓 기반의 호출은 모든 호출의 근간이 됩니다. 관련 기술 확보가 중요합니다.



지역(local) 호출	원격(Remote) 호출						
	프로토콜 기반 연결	브로커 기반 연결					
<ul style="list-style-type: none"><li>✓ 함수/메소드 호출</li><li>✓ 라이브러리 호출</li><li>✓ 컴포넌트 인터페이스 호출</li><li>✓ 프레임워크 인터페이스 호출</li><li>✓ 플랫폼 인터페이스 호출</li><li>✓ ...</li></ul>	<ul style="list-style-type: none"><li>✓ TCP 기반 연결</li><li>✓ UDP 기반 연결</li><li>✓ HTTP 기반 연결</li><li>✓ FTP/SMTP 기반 연결</li></ul>	<table border="1"><tr><td>라이브러리 기반 연결</td></tr><tr><td>✓ Netty</td></tr><tr><td>프레임워크 기반 연결</td></tr><tr><td>✓ RESTful</td></tr></table>	라이브러리 기반 연결	✓ Netty	프레임워크 기반 연결	✓ RESTful	<ul style="list-style-type: none"><li>✓ CORBA (IDL 방식)</li><li>✓ EAI 솔루션 (메시지)</li><li>✓ MQ (메시지)</li><li>✓ 웹 서비스 (IDL + 메시지)</li></ul>
라이브러리 기반 연결							
✓ Netty							
프레임워크 기반 연결							
✓ RESTful							

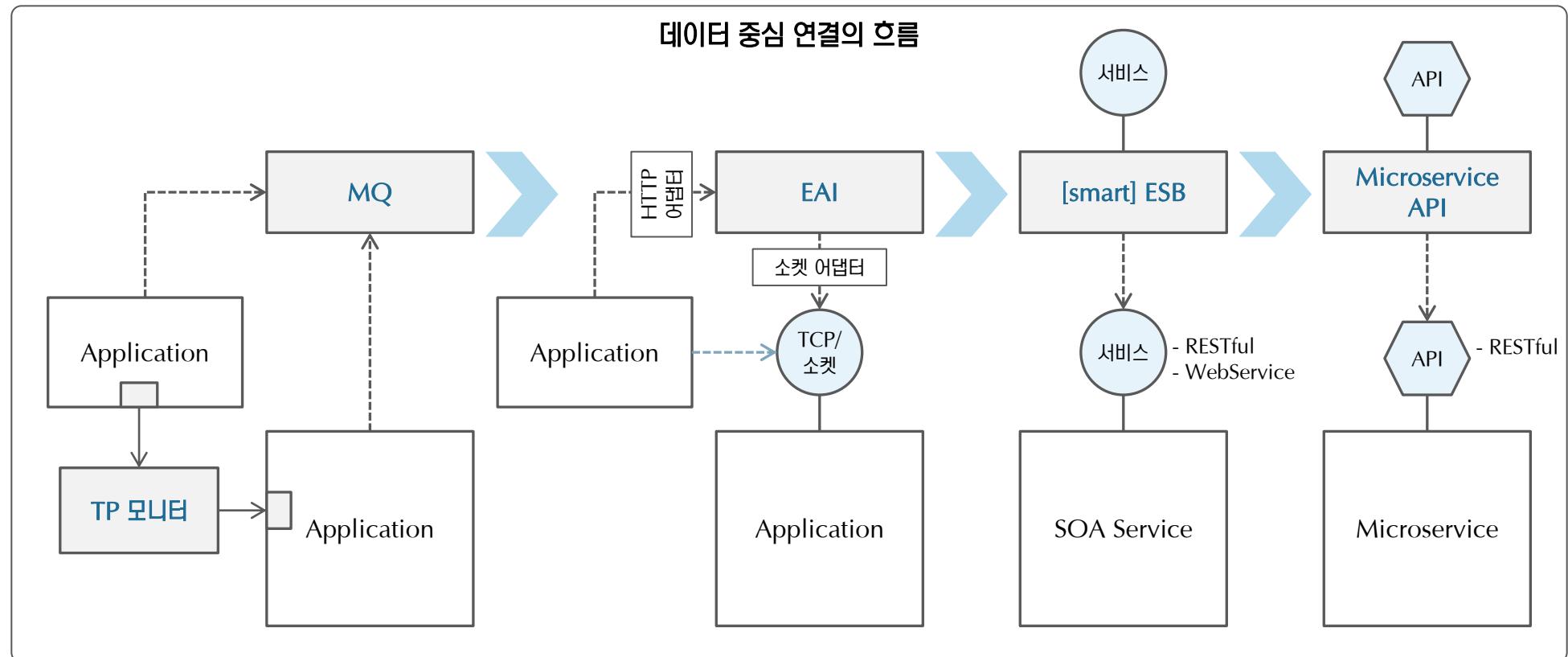
## 1.3 연결 방식: 다양한 원격 호출

- ✓ 원격 호출을 위한 커넥터를 구현하는 방법은 기술의 흐름을 따라 발전해 왔습니다.
- ✓ 1990년대에는 분산 컴퓨팅이 커다란 이슈가 되었으며, 원격 호출을 위한 기술이 발전하기 시작했습니다.
- ✓ 최근에는 컴포넌트 + 연결에 대한 기술이 Microservices에 이르렀으며 관련 기술이 주목을 받고 있습니다.
- ✓ 그런 관점에서 웹 + RESTful + MQ 기술이 연결의 중심을 차지하고 있습니다.

원격 서비스 호출 방법	프로토콜	결합도	성능	구현 난이도	메시지 구성	사용 빈도	환경 구성	구현 수단
TCP 소켓	TCP	강함	아주 빠름(1)	높음	--	높음	흔함	프로그래밍 언어
브로커(CORBA)	솔루션 의존(TCP, UDP)	강함	빠름(2)	높음	표준	거의 없음	구성 어려움	솔루션
TP 모니터	솔루션 의존(TCP, UDP)	강함	빠름(2)	중간	--	거의 없음	구성 어려움	솔루션
MQ	솔루션 의존	아주 느슨함	느림(4)	낮음	표준(JMS)	높음	흔함	오픈소스/솔루션
EAI	솔루션 의존(TCP/HTTP)	중간	빠름/중간(3)	중간	--	줄어듦	구성 어려움	솔루션
웹 호출	HTTP	느슨함	빠름(2)	낮음	--	높음	흔함	오픈소스/솔루션
웹 서비스	SOAP	강함	느림(3)	높음	표준	적음	구성 어려움	솔루션/오픈소스
RESTful	HTTP	느슨함	빠름(2)	낮음	표준	높아짐	흔함	오픈소스

## 1.3 연결 방식: 기술 흐름

- ✓ 기업 시스템은 여러 개의 애플리케이션으로 구성되어 있으며, 서로 간에 연결을 통해 협업을 합니다.
- ✓ 과거의 MQ나 TPM으로부터 현재의 Microservice API에 이르기까지 기술이 발전해 왔습니다.
- ✓ MQ나 EAI는 여전히 많이 쓰고 있는 기술이지만, 전체 흐름은 REST 서비스 쪽으로 흘러가고 있습니다.
- ✓ 서비스는 Smart 서비스에서 micro 서비스로 흐름이 형성되고 있습니다.



# 1.3 연결 방식: Heterogeneousness view

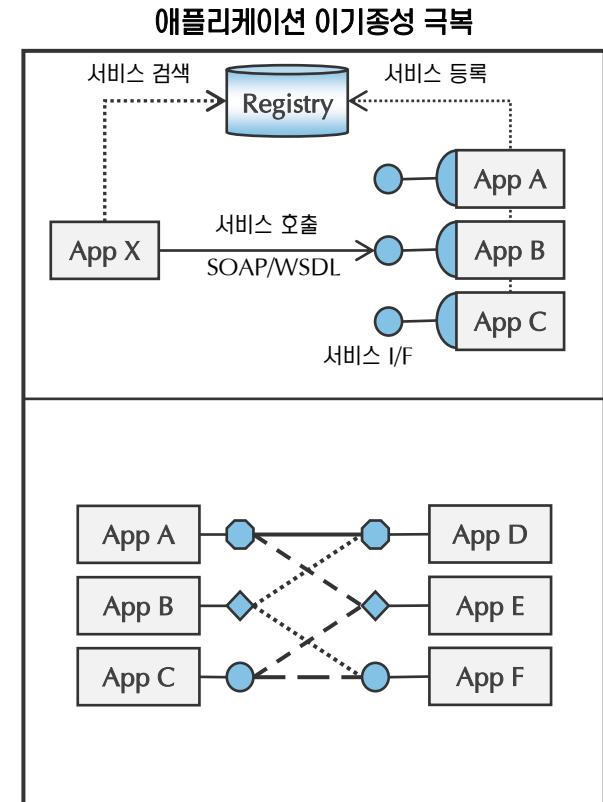
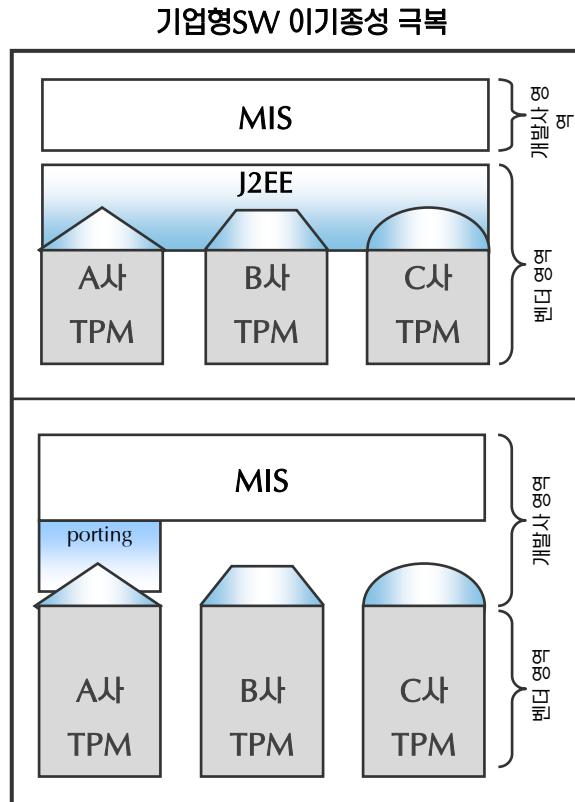
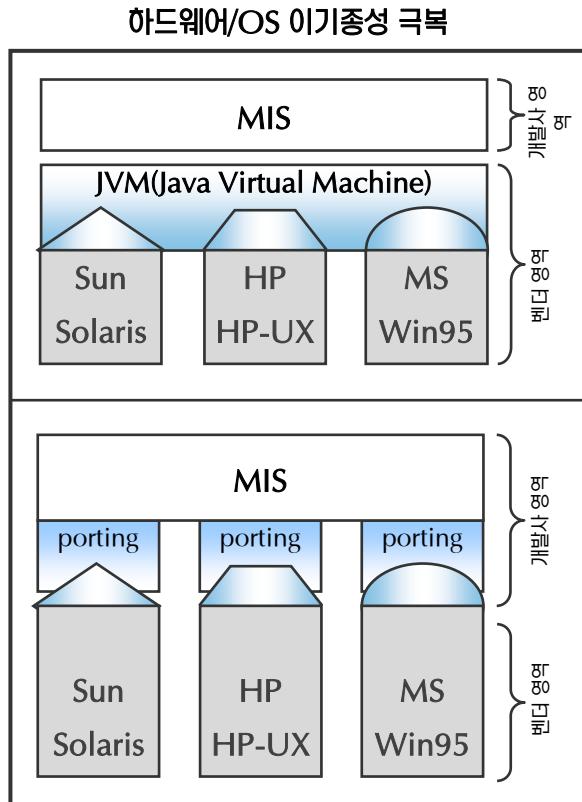
✓ 이기종성(Heterogeneousness) 극복은 새로 기술 탄생의 동력이 됩니다.

✓ 하드웨어/운영체제 이기종성 극복은 운영체제 의존성(dependency)을 제거하였고,

✓ 기업형 SW이기종성 극복은 벤더 의존성을 줄여주었고,

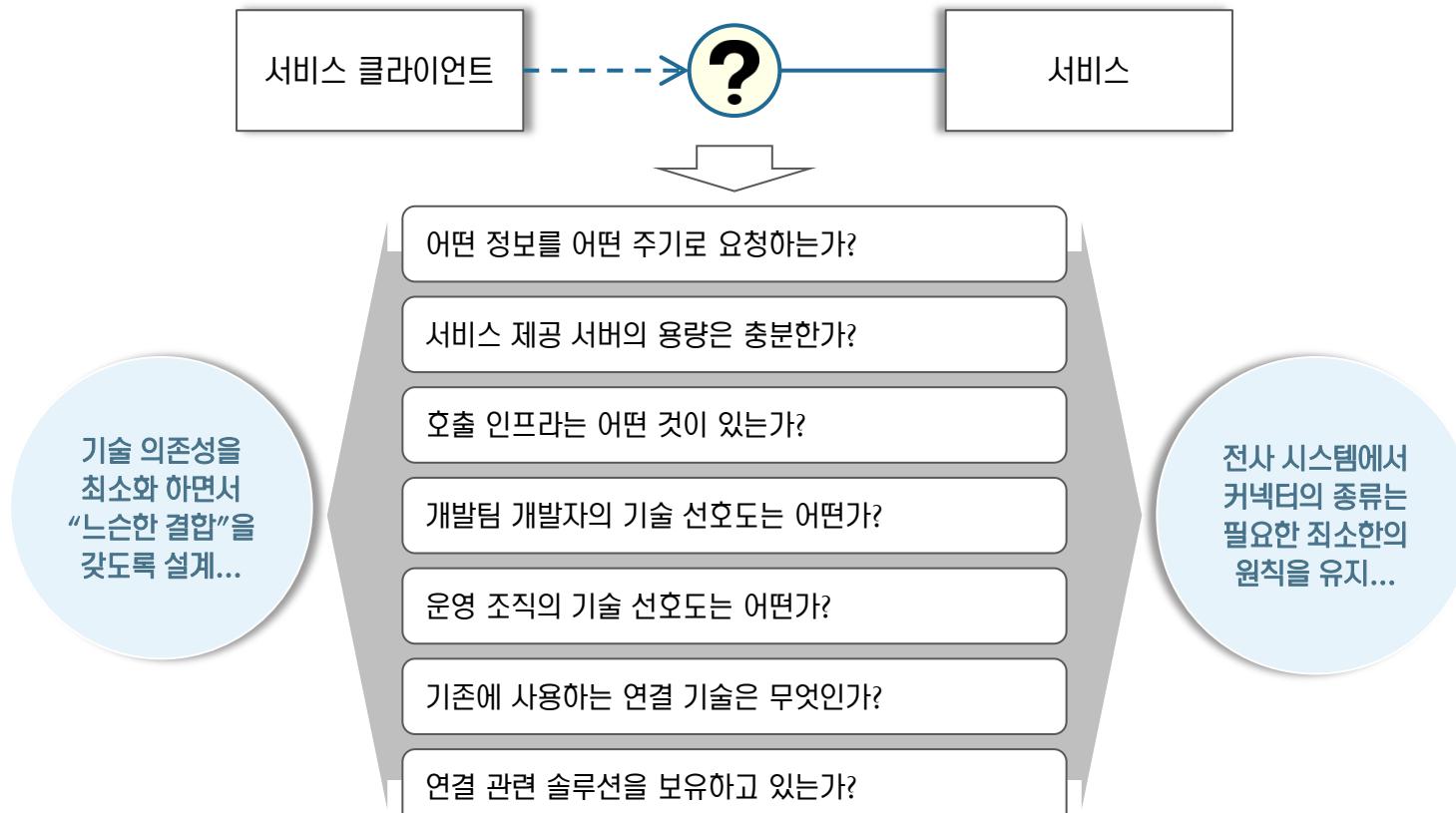
✓ App 인터페이스 이기종성 극복은 인터페이스 기술 의존성과 애플리케이션 간 의존성을 줄여주었습니다.

Write once, run anywhere.



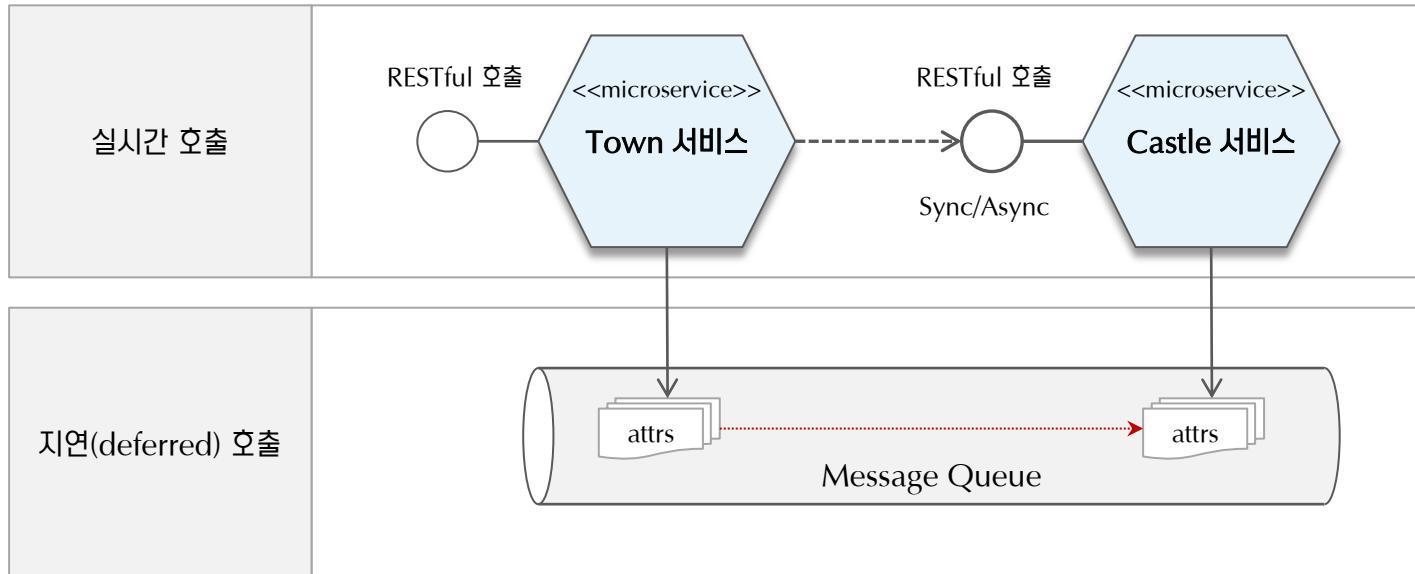
## 1.4 연결 설계: 고려사항

- ✓ 두 컴포넌트(아키텍처 요소) 사이를 “서비스 호출” 형식으로 연결할 때 고려할 내용이 여러 가지 있습니다.
- ✓ 유사한 시스템의 이전에 설계한 방식을 따라서 연결 설계를 결정하는 경우가 많습니다.
- ✓ 서비스 호출 연결 컨텍스트는 매우 다릅니다. 그 컨텍스트를 정확히 이해한 후 설계를 결정하여야 합니다.
- ✓ 어떤 경우든 “느슨한 결합”을 지향하고, 커넥터는 필요한 최소한의 원칙을 유지해야 합니다.



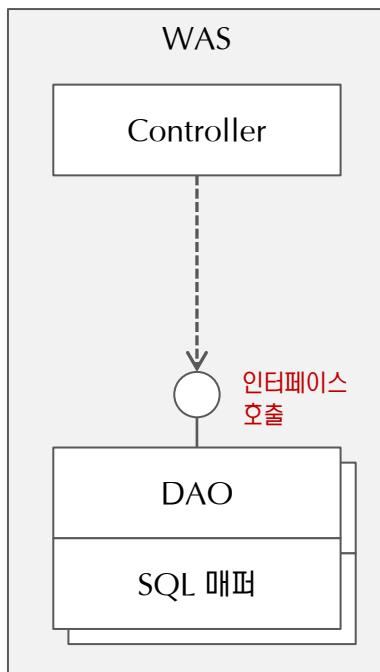
## 1.4 연결 설계: Two way connections

- ✓ ACID 트랜잭션 기반으로 서비스를 호출할 때, 호출 경로를 하나로 설계하는 것이 일반적이었습니다.
- ✓ 하지만, 빅 데이터 시대에 그런 단순한 호출 방식으로는 시스템 목표를 달성하기 어렵습니다.
- ✓ 실시간 호출도 필요에 따라 동기/비동기 호출 방식으로 나누어야 하고, 데이터가 많거나 시급하지 않은 것들은 지연 호출 경로를 준비하고 사용하여야 합니다

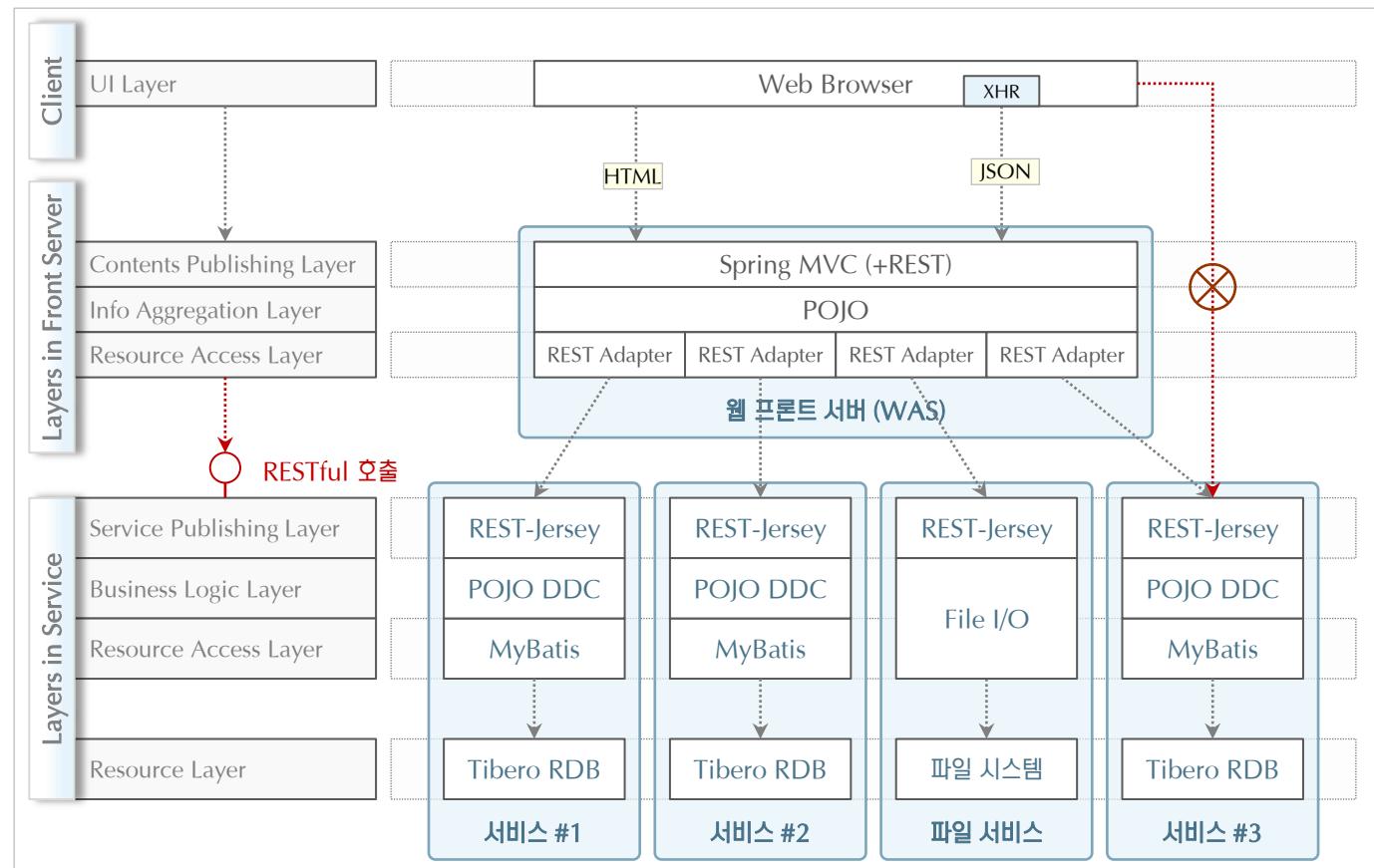


## 1.4 연결 설계: 사례 – RESTful 서비스

- ✓ 서비스 별 확장, 배포가 가능한 구조여야 하며, 궁극적으로 Microservices 기반 설계를 한 사례입니다.
- ✓ 고객 요구사항에서 따라 무늬만 컴포넌트가 아닌 “완전한 비즈니스 컴포넌트” 구조로 설계하였습니다.
- ✓ 비즈니스 로직 레이어를 제대로 갖추었으며, 프론트로 RESTful 서비스 발행(publishing)을 했습니다.
- ✓ DBMS는 단순한 데이터 저장소 역할을 하며, 모든 로직은 비즈니스 로직 레이어의 컴포넌트에 두었습니다.

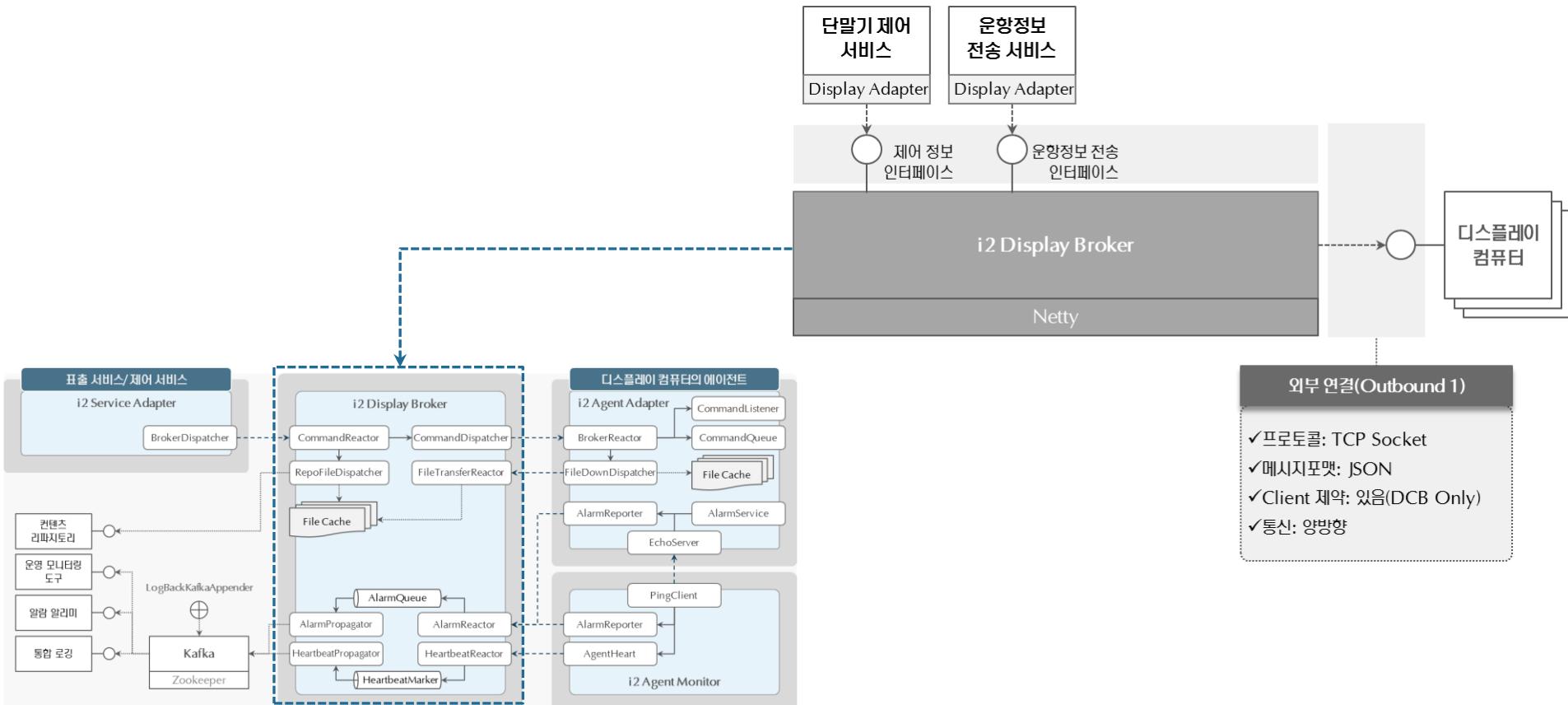


VS



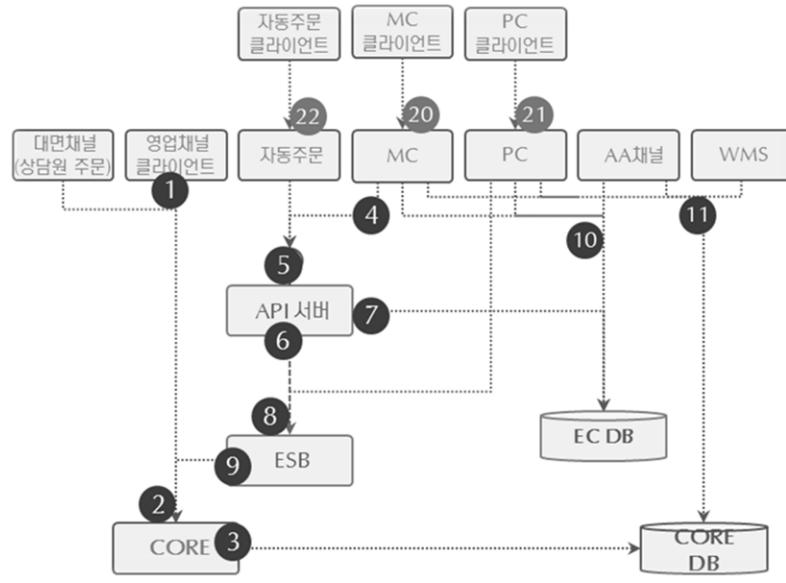
## 1.4 연결 설계: 사례 - 브로커

- ✓ 서버로부터 디스플레이 클라이언트로 정보를 보내거나 수집하는 시스템입니다.
- ✓ 디스플레이 제어 컴퓨터는 오래되었거나, 낮은 스펙인 경우도 있으며, TCP 소켓만을 지원하는 것도 있었습니다.
- ✓ 표준 메시지가 아니라 개발자가 정의해야 하는 경우와 제조사의 메시지를 따라야 하는 경우도 있었습니다.
- ✓ 따라서, TCP 기반으로 디스플레이 컴퓨터를 제어할 수 있도록 브로커-에이전트 구조로 설계를 했습니다.



# 1.5 사례로 보는 연결의 문제와 해결방안

- ✓ 복잡성 문제로 성능, 대응 지연 등의 문제를 해결하기 위해 e-commerce 시스템을 분석했습니다.
- ✓ 국내 시스템이 갖고 있는 전형적인 문제를 대부분 안고 있는 사례였습니다.
- ✓ 사례를 살펴보고, 3일 간의 학습 후에, 마지막 날은 재설계하는 시간을 갖겠습니다.

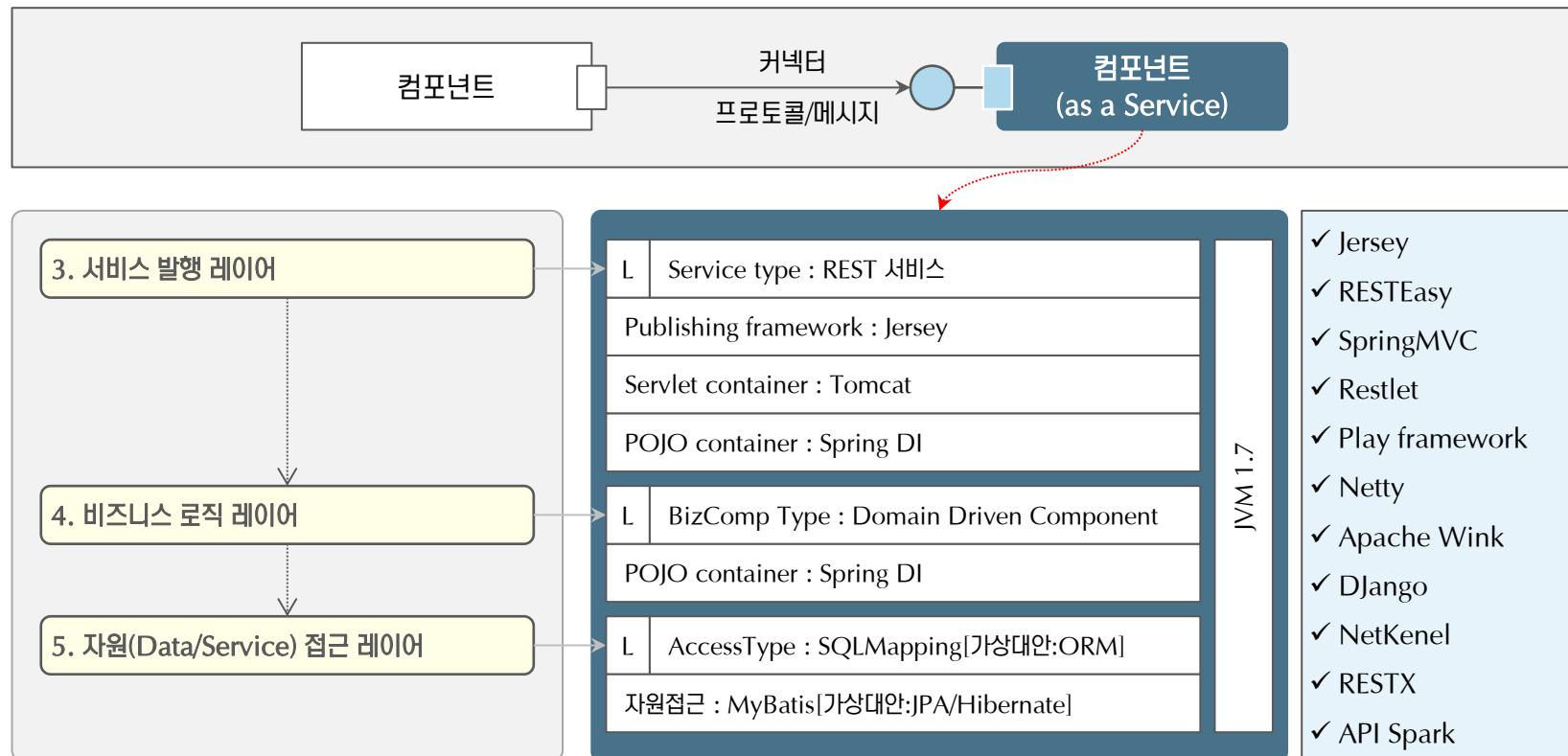


	연결 이름	연결 유형	라이브러리/프레임워크
1	CORE 호출	서비스 호출	MiPlatform, HTTP/XML
2	CORE 서비스 제공	서비스 제공	Spring MVC, HTML/XML, JSON, dev.story, MiPlatform 라이브러리
3	CORE DB 접근	DB 접근	dev.story, XML 기반 질의어
4	API 호출	서비스 호출	iceberg, HTTP/JSON (not RESTful)
5	API 제공	서비스 제공	Spring MVC, HTTP/JSON
6	ESB 호출	서비스 호출	dev.story, HTTP/JSON
7	EC DB 접근	DB 접근	MyBatis, SQL 매핑
8	ESB 서비스 제공	서비스 제공	Glassfish, HTTP/JSON
9	CORE 호출	서비스 호출	HTTP/JSON
10	EC DB 접근	DB 접근	MyBatis, SQL 매핑
11	CORE DB 접근	DB 접근	dev.story, XML 기반 질의어

[사례연구 보기](#)

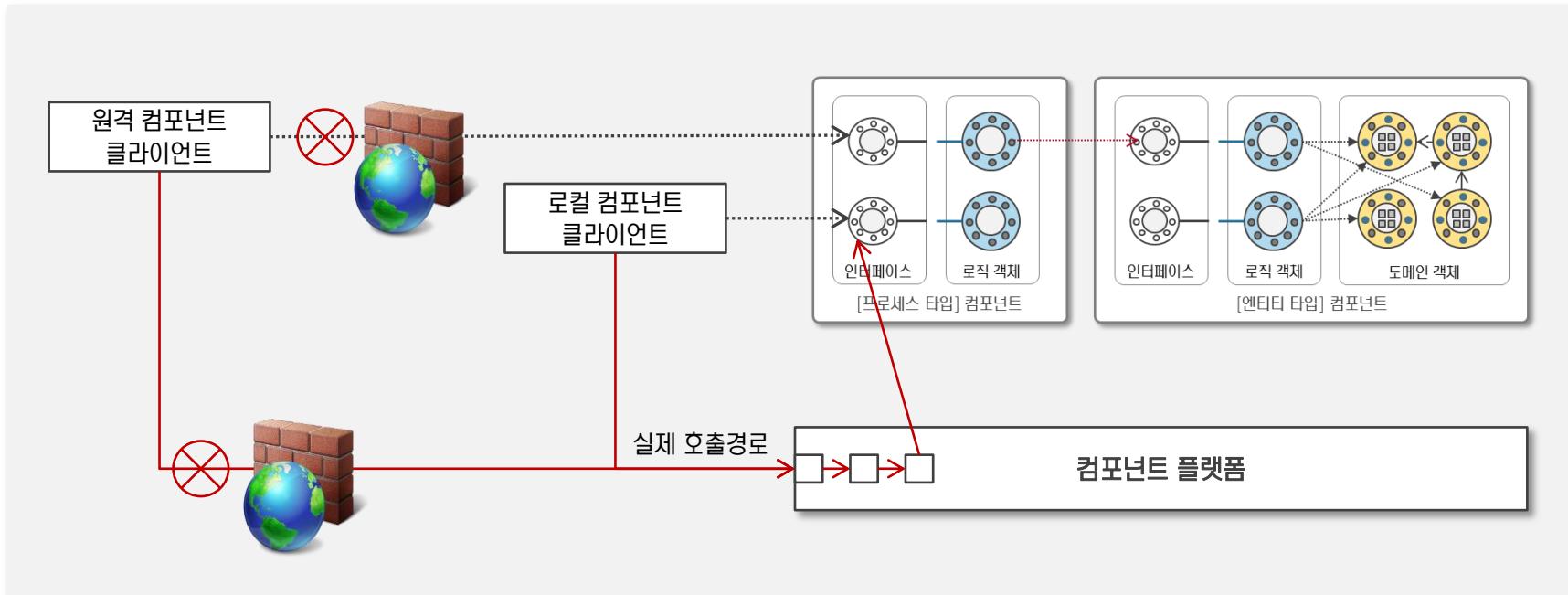
# 1.6 컴포넌트: 서비스 발행

- ✓ 커넥터의 실체는 두 컴포넌트 사이에 존재하는 것이 아니라 각 컴포넌트에 존재합니다.
- ✓ 서비스 컴포넌트(아키텍처 요소)의 경우, RESTful 서비스를 위한 장치가 서비스 컴포넌트 앞쪽에 있습니다.
- ✓ 커넥터의 특성은 컴포넌트가 연결을 위해 어떤 장치를 사용하는 가에 따라 다릅니다.
- ✓ RESTful 서비스 제공을 위해 사용할 수 있는 프레임워크는 매우 다양합니다.



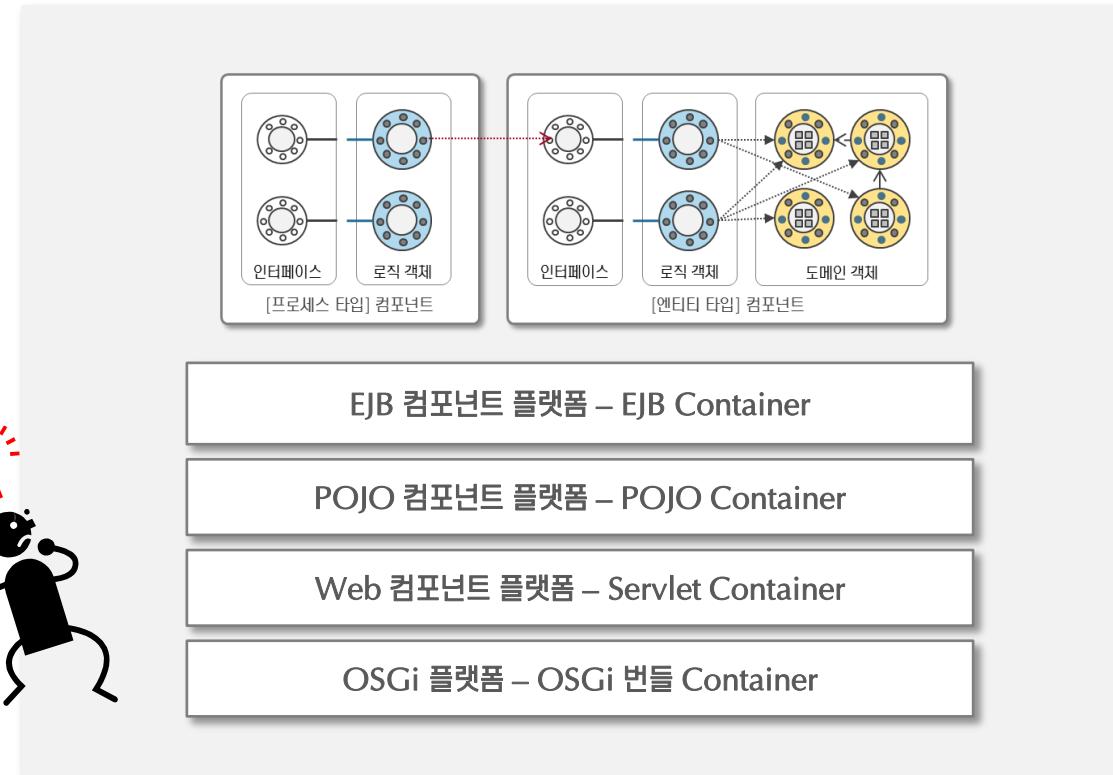
## 1.7 컴포넌트 플랫폼 (1/4) – 호출거리

- ✓ 로컬 클라이언트는 컴포넌트 플랫폼에 접근할 수 있지만, 원격 클라이언트는 그렇지 못합니다.
- ✓ 컴포넌트플랫폼으로의 접근을 보다 쉽게 도와주는 어떤 인프라가 필요합니다. ← 서비스 발행 플랫폼의 역할
- ✓ 원격 클라이언트가 로컬의 컴포넌트 플랫폼으로의 접근을 도와주는 인프라가 필요합니다.



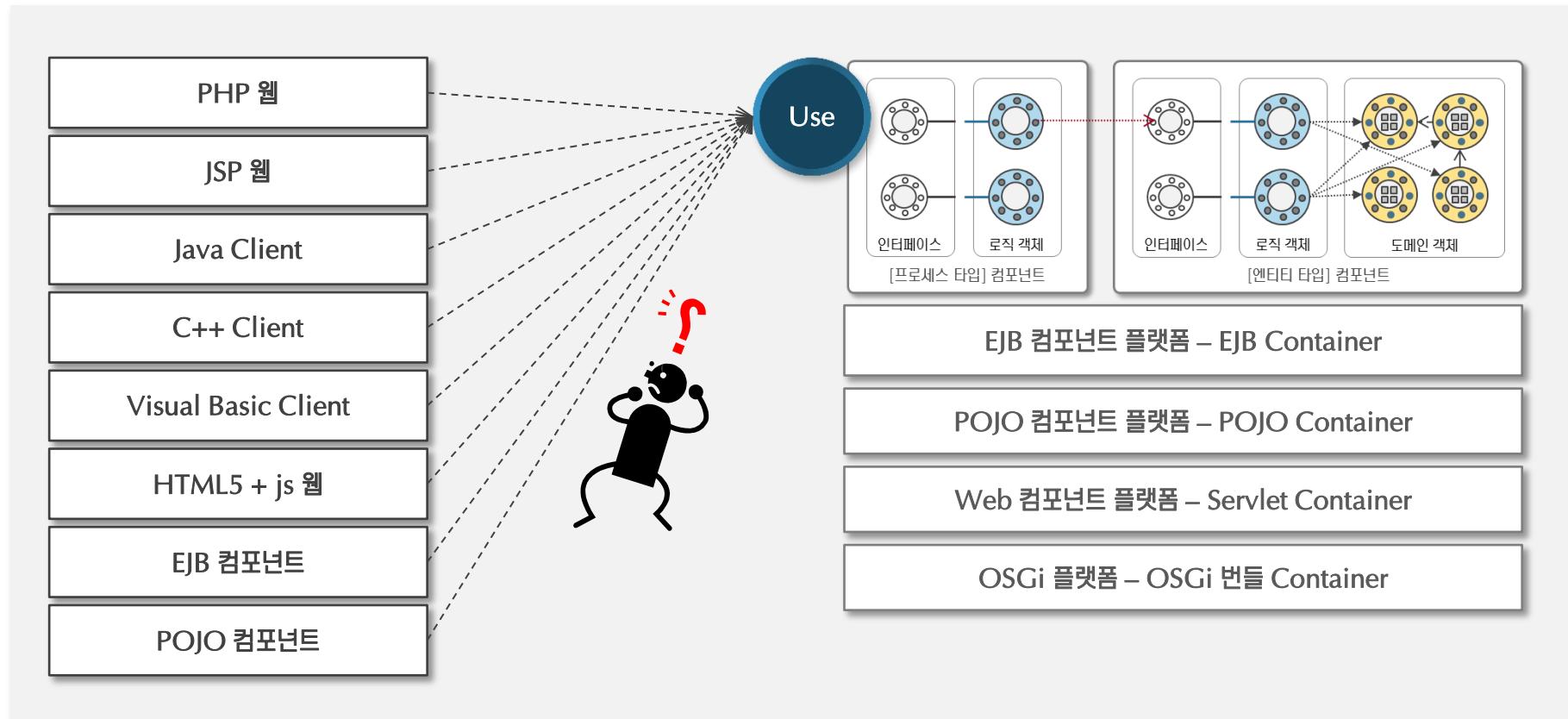
## 1.7 컴포넌트 플랫폼 [2/4] – 다양한 컴포넌트 모델

- ✓ 다양한 컴포넌트 모델이 존재하므로 목표 시스템의 특성을 잘 반영한 모델을 선택할 수 있는 장점이 있습니다.
- ✓ 하지만, 이러한 다양성은 클라이언트 입장에서 보면 혼란과 어려움을 의미합니다.
- ✓ 서버 측의 다양한 컴포넌트에 한 가지 방식으로 접근할 수 있어야 합니다. ← 서비스 발행 플랫폼의 역할



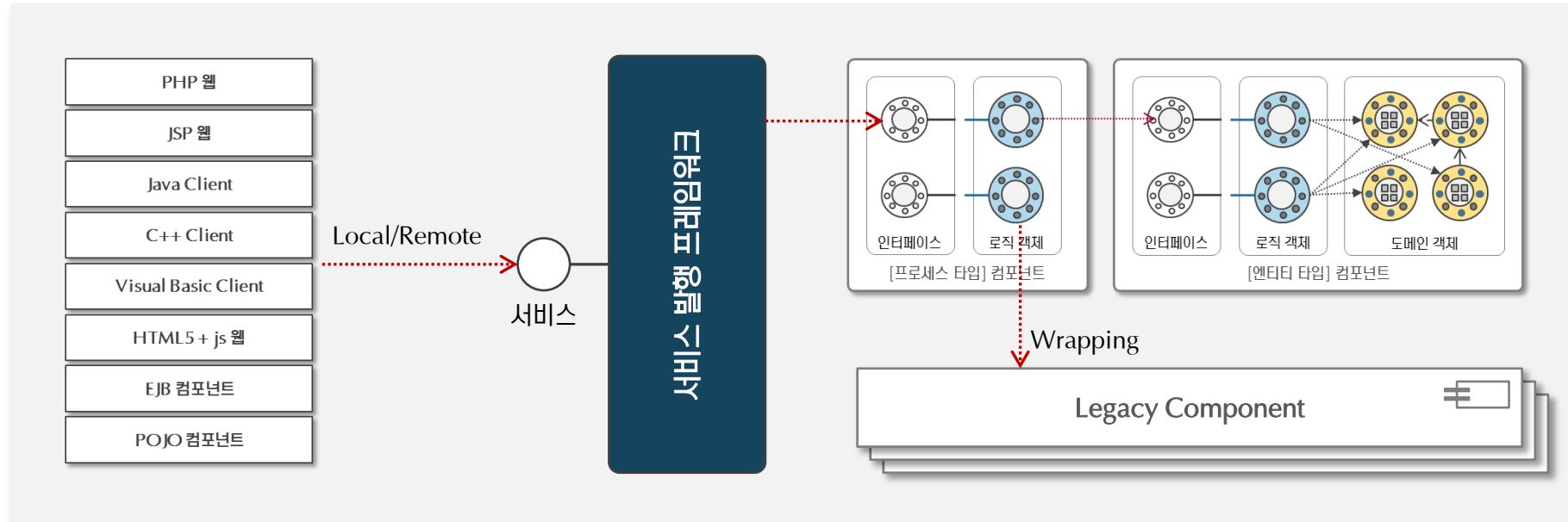
## 1.7 컴포넌트 플랫폼 [3/4] – 사용/재사용의 어려움

- ✓ 원격 접근의 어려움 외에도, 컴포넌트 인터페이스의 사용 또는 재사용의 어려움이 존재합니다.
- ✓ 서버 측에서 잘 구축된 컴포넌트를 쉽게 활용할 수 있다면 개발 생산성 및 자원 효율성을 높일 수 있습니다.
- ✓ 모든 클라이언트들이 가장 잘 알고 쉽게 접근하는 방법이 있어야 합니다. ← 서비스 발행 플랫폼의 역할



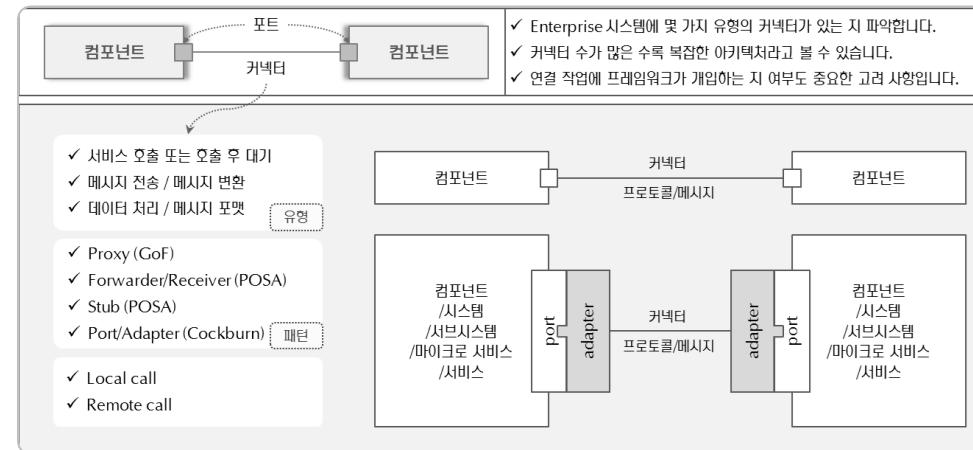
## 1.7 컴포넌트 플랫폼 (4/4) – SOA를 통한 극복

- ✓ 서비스 발행 프레임워크은 컴포넌트의 제약 조건을 극복할 수 있도록 도와주는 인프라입니다.
- ✓ 사용하기 쉬운 방식으로 표준 서비스 인터페이스를 외부로 노출(publishing) 하여 줍니다.
- ✓ 순수 웹을 기반으로 하는 REST 방식과 표준을 기반으로 하는 WS\*(WSDL, SOAP, UDDI) 방식이 있습니다.



# 요약

- ✓ 아키텍처 설계는 컴포넌트를 정의하고 내부 구조를 설계하고, 컴포넌트 사이를 연결하는 활동입니다.
- ✓ RESTful 서비스는 마이크로서비스를 서로 연결하는 수단으로, 연결을 위한 최선의 방법이 되었습니다.
- ✓ REST에 대한 올바른 이해와 정확한 활용은 현대의 Enterprise 시스템 아키텍처의 핵심이 되었습니다.
- ✓ RESTful 서비스를 널리 사용되는 웹 인프라를 그대로 활용할 수 있다는 장점이 있습니다.





## 2. REST 아키텍처 스타일

- 
- 2.1 웹 서비스
  - 2.2 RESTful 아키텍처 스타일
  - 2.3 성숙도 모델
  - 2.4 Safety와 역등
  - 2.5 HTTP 동사
  - 2.6 JAX-RS 2.0
  - 2.7 클라이언트 도구 - Postman

## 2.1 웹 서비스[1/2]

- ✓ 웹 서비스를 구현할 때 두 가지 기술을 사용할 수 있습니다. → 웹서비스와 RESTful 서비스
- ✓ 웹 서비스 기술은 호출 당사자 사이에 엄격한 계약이 있을 경우, 적절한 선택이 될 수 있습니다.
- ✓ 개방형 서비스를 위한 API를 제공하려면 HTTP 기반인 JAX-RS를 사용해야 합니다.
- ✓ 관공서 등에서는 웹 서비스를 , 인터넷 제공 업체 등에서는 RESTful 서비스를 주로 사용합니다.

Web Service	RESTful 서비스
JAX-WS	JAX-RS
Java API for XML Web Services	Java API for RESTful Web Services
SOAP/WSDL/UDDI	HTTP/JSON
금융감독원	Google, T map, Naver

*HTTP 1.1은 분산, 하이퍼미디어 정보 시스템의 협업에 사용합니다. HTTP는  
Representational State Transfer(REST) 개념을 바탕으로 설계하였습니다.*

## 2.1 웹 서비스[2/2]

- ✓ 정부의 공공 데이터 포털인 <http://www.data.go.kr> 을 보면 오픈 API를 볼 수 있습니다.
- ✓ RESTful API가 압도적으로 많음을 알 수 있습니다.

The screenshot shows the data.go.kr open API portal. At the top, there are tabs for '전체(21,197)', '파일데이터(19,047)', '오픈API(2,123) [selected]', and '표준데이터(27)'. Below this, a main title says '오픈API 2,123건을 찾았습니다.' with a search bar labeled '기관별 검색'. On the left, there's a section for '자전거 정보' with details: 조회수 : 242, 활용신청건수 : 4, 수정일 : 2016.11.25, 기관 : 제주특별자치도 제주시, 서비스유형 : REST. Below it is a 'XML' button. To the right of this section are three orange arrows pointing towards the sidebar, each labeled '보통필터' (General Filter). In the center, there's a section for '검진기관별 정보제공' with details: 조회수 : 30, 활용신청건수 : 50, 수정일 : 2016.11.23, 기관 : 국민건강보험공단, 서비스유형 : REST. Below it is a 'XML' button. To its right is another set of three orange arrows labeled '보건의료' (Healthcare). At the bottom, there's a section for '검진기관 찾기' with details: 조회수 : 32, 활용신청건수 : 60, 수정일 : 2016.11.23, 기관 : 국민건강보험공단, 서비스유형 : REST. Below it is a 'XML' button. To its right is another set of three orange arrows labeled '보건의료'. On the right side, there is a sidebar titled '인기검색어' containing links for 범죄, 날씨, 축산물, 유동인구, 미세먼지. Below this is a '국가중점데이터' dropdown. Under '서비스유형필터(OPENAPI)', there is a list with SOAP(74), REST(1340) [highlighted with a red arrow], RSS/ATOM(1), and LINK(739). Further down are '제공기관필터', '분류체계필터', and '이용허락범위필터' dropdowns.

## 2.2 RESTful 아키텍처 스타일(1/6)

- ✓ Representational State Transfer는 아키텍처 스타일로써 HTTP 동사와 URI를 준수합니다.
- ✓ REST는 다음 네 가지 원칙을 준수합니다.
  - [원칙 1] 모든 리소스는 URI로 식별한다.
  - [원칙 2] 모든 리소스는 다중 표현(multiple representation)을 가질 수 있다.
  - [원칙 3] 모든 리소스는 표준 HTTP 메소드로 접근/변경/생성/삭제 할 수 있다.
  - [원칙 4] 서버는 상태 정보를 갖지 않는다.

*Web storage provides far greater storage capacity (5 MB per origin in Google Chrome, Mozilla Firefox, and Opera; 10 MB per storage area in Internet Explorer; 25MB per origin on BlackBerry 10 devices) compared to 4 kB (around 1000 times less space) available to cookies.*

## 2.2 RESTful 아키텍처 스타일(2/6)

✓ 다음 제약 조건을 만족할 때, 그 서비스 또는 애플리케이션은 RESTful 서비스로 간주합니다.

- 클라이언트-서버

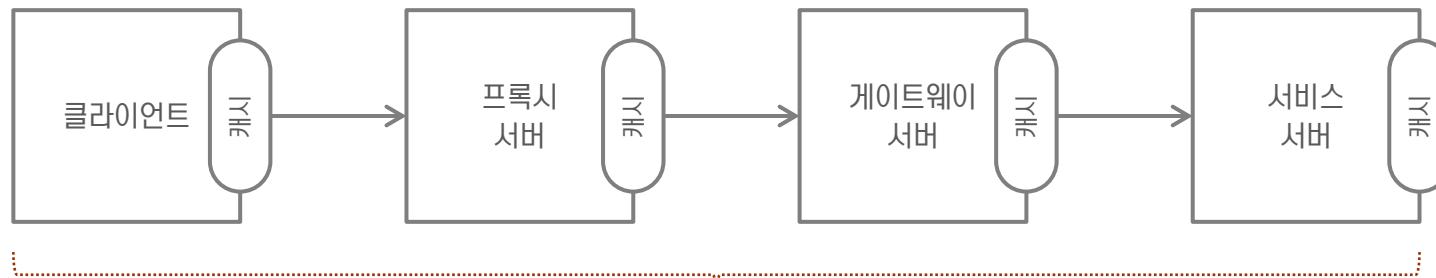
- 인터페이스를 사이에 두고 UI를 다루는 클라이언트와 데이터를 다루는 서버로 역할을 명확하게 나누어야 한다.
- Separation of concerns, portability, scalable(서버는 사용자 상태를 관리하지 않음)
- HTML5의 Local storage (5MB+ limit)와 클라이언트 서버 역할의 변화는 있는가?

- 캐시가능(cacheable)

- 응답(response)은 캐시가능 하여야 하며, 상태가 변경되어 못쓰게 된 응답을 재사용하는 경우가 없어야 한다.
- 잘 관리한 캐시는 클라이언트-서버 호출을 없애, 수행 성능과 범위성(scability)을 높여준다.

- 레이어드(layered)

- 요청은 중개(intermediary) 서버를 경유하여, 마지막 서버로 전달되어야 한다.
- 중개 서버는 로드-밸런싱과 공유 캐싱을 통해 시스템의 범위성을 높일 수 있도록 설계하여야 합니다.



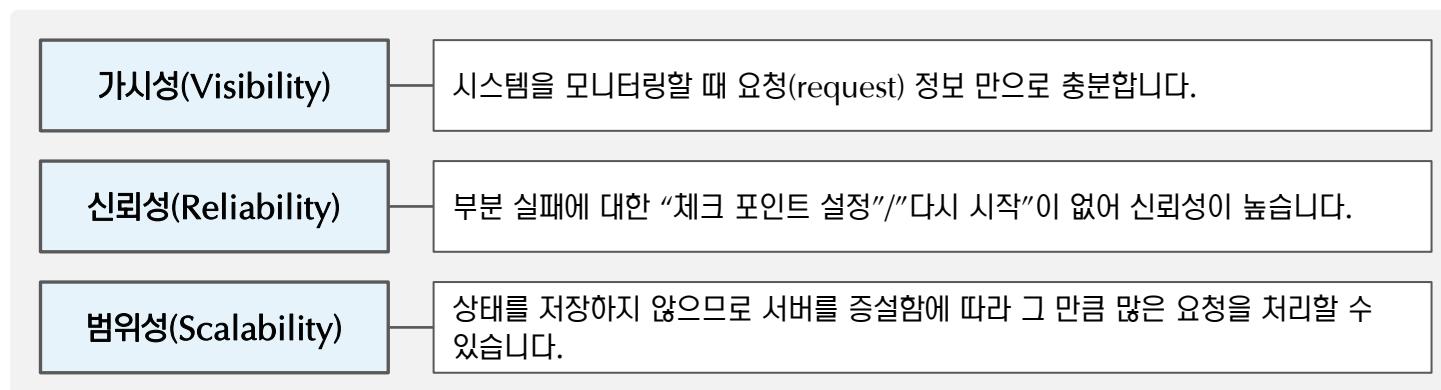
*Cacheable and Layered constraint*

## 2.2 RESTful 아키텍처 스타일(3/6)

### ✓ (계속)

- 상태 없음(stateless)

- 요청과 요청 사이에 클라이언트 컨텍스트를 서버에 저장하지 않습니다. 각 요청에 필요한 모든 정보를 담는다.
- 클라이언트 상태는 클라이언트가 갖고 있다.
- 인증 등의 정보를 포함한 세션 상태는 서버에 의해 데이터베이스 같은 다른 서비스로 이전할 수 있다.
- 이런 원칙은 요청에 대한 가시성(visibility), 신뢰성(reliability), 범위성(scalability)을 높여 줍니다.
- Roy Fielding의 글([http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)) 참조



## 2.2 RESTful 아키텍처 스타일(4/6)

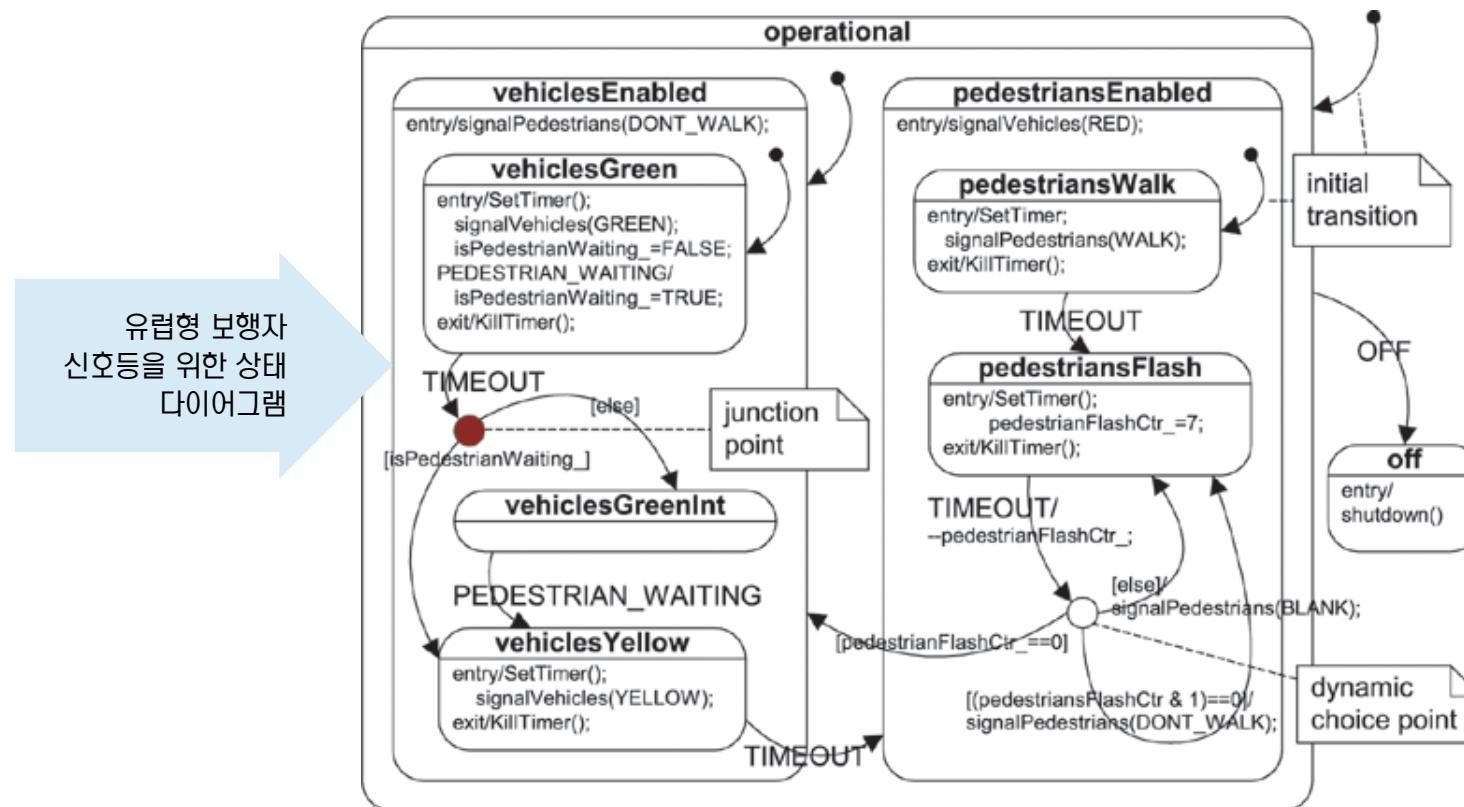
### ✓ (계속)

- Code on demand(Optional)
  - 실행 가능한 코드(Java applet, JavaScript)를 보냄으로써 클라이언트 기능을 확장하거나 조정할 수 있어야 한다.
  - REST 아키텍처의 유일한 선택적인 제약 조건임
- Uniform interface
  - REST 서비스 설계의 근간이 되는 제약 조건으로, 결합도를 낮추어서 각자 독립적으로 변화할 수 있도록 한다.
  - [리소스 식별 제약조건]
    - . 개별 리소스를 URI를 이용하여 요청에서 식별한다.
    - . 리소스 자체는 JSON, XML 등의 표현과는 개념적으로 분리되어 있다.
  - [표현을 통한 리소스 조작 제약조건]
    - . 클라이언트가 리소스에 대한 표현을 얻었을 때(첨부된 메타데이터 포함하여), 리소스를 삭제하거나 변경할 충분한 정보를 갖고 있다.
  - [자기 서술적 메시지]
    - . 각 메시지는 다루는 방법에 대한 충분한 정보를 포함한다. MIME 타입에 따라 어떤 파서를 사용할 지 알 수 있다.

## 2.2 RESTful 아키텍처 스타일(5/6)

### ✓ (계속)

- [HATEOAS-Hypermedia as the Engine of Application State] 애플리케이션 상태에 대한 엔진으로써 하이퍼미디어
  - . 클라이언트는 서버에 의해 하이퍼미디어 안에 동적으로 담아 둔 액션을 통해서만 상태를 변경해야 한다.
  - . 클라이언트는 바로 전에 서버로부터 전해 받은 표현 안에 서술된 것을 넘어서는 (특정 리소스에) 액션이 있다고 생각하지 않는다.



[출처] [http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2003/0306/cuj0306samek/cuj0306samek\\_f2.htm](http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2003/0306/cuj0306samek/cuj0306samek_f2.htm)

## 2.2 RESTful 아키텍처 스타일(6/6)

### ✓ (계속)

#### - [HATEOAS] (계속)

- . 애플리케이션의 모델은 자원 표현(모델)의 현재 집합 안의 하이퍼링크를 가로 지르면서 한 상태에서 다른 상태로 변화한다.
- . RESTful 시스템에서 CORBA나 자바 RMI와 같은 전통적인 클라이언트-서버 통신 모델에서 볼 수 있는 것과 같은 클라이언트와 서버 간에 고정된 인터페이스는 없다.
- . REST에서 클라이언트는 응답 바디 안에 있는 링크를 다루는 방법만 알면 된다. 클라이언트-서버 상호작용을 매우 다이나믹하게 하고 다른 네트워크 아키텍처와의 차별화 한다.

```
http://www.packtpub.com/resources/departments/IT
```

## 2.2 RESTful 아키텍처 스타일(6/6)

✓ (계속)

- [HATEOAS] (계속)

packtpub.com, RESTful Java Web Services

```
{ "departmentId" :10,
  "departmentName" :" IT"
  "manager" :" John Chen",
  "links" : [ {
    "rel" :" employees",
    "href" :http://packtpub.com/resources/departments/IT/employees
  } ] }
```

```
[{ "employeeId" :100,
  "firstName" :" Steven",
  "lastName" :" King",
  "links" : [ {
    "rel" : "self"
    "href" : "http://www.packtpub.com/resources/employees/100"
  }]
},
{ "employeeId" :101,
  "firstName" :" Neena",
  "lastName" :" Kochhar",
  "links" : [ {
    "rel" : "self"
    "href" : "http://www.packtpub.com/resources/employees/101"
  }]
}]
```

## 2.2 RESTful 아키텍처 스타일(6/6)

✓ (계속)

GET /account/12345 HTTP/1.1

Host: somebank.org

Accept: application/xml

...

HTTP/1.1 200 OK

Content-Type: application/xml

Content-Length: ...

```
<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="http://somebank.org/account/12345/deposit" />
  <link rel="withdraw" href="http://somebank.org/account/12345/withdraw" />
  <link rel="transfer" href="http://somebank.org/account/12345/transfer" />
  <link rel="close" href="http://somebank.org/account/12345/close" />
</account>
```

<https://en.wikipedia.org/wiki/HATEOAS>

HTTP/1.1 200 OK

Content-Type: application/xml

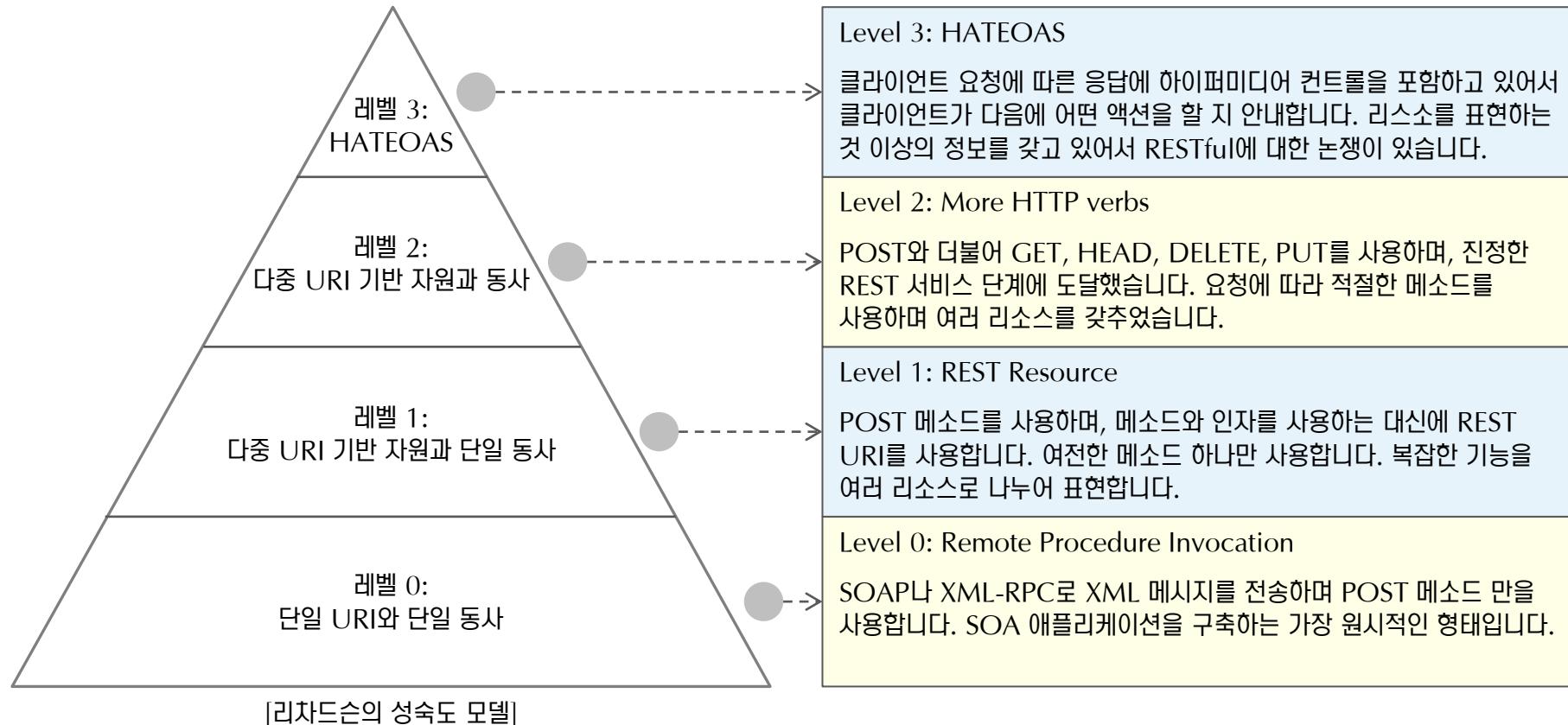
Content-Length: ...

```
<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">-25.00</balance>
  <link rel="deposit" href="http://somebank.org/account/12345/deposit" />
</account>
```

가능한 액션을 리소스의 상태에 따라 다양합니다.

## 2.3 성숙도 모델

- ✓ Leonard Richardson은 REST 서비스 성숙도 모델을 고안해 냈습니다.
- ✓ 레벨 0: 단일 URI와 단일 동사로부터 레벨 3: HATEOAS까지 총 네 단계로 구성되어 있습니다.
- ✓ 레벨 2부터 진정한 RESTful 서비스 수준에 이르렀다고 할 수 있습니다.



## 2.4 Safety와 멱등(idempotent)

---

### ✓ Safety

- 안전한(safe) 메소드는 여러 차례 호출에서 서버에서 자원의 상태를 변화시키지 않습니다.
  - GET /v1.1/coffees/orders/1234
- 안전한 메소드의 응답은 캐시에 둘 수 있습니다.
- GET과 HEAD는 안전한 메소드입니다.

### ✓ Idempotent(멱등) methods

- 여러 차례 불러도 동일한 결과를 보여주는 메소드를 멱등(idempotent) 메소드라고 합니다.
- GET 메소드는 멱등(idempotent) 메소드이며, 여러 차례 호출해도 결과는 같습니다.
- PUT 메소드는 멱등(idempotent) 메소드이며, 여러 차례 업데이트 해도 결과는 같습니다.
- POST 메소드는 멱등이 아닙니다. 여러 차례 호출은 매번 다른 결과를 가져옵니다.
- DELETE 메소드는 멱등입니다. 여러 차례 삭제를 해도 결과는 매번 동일합니다.

## 2.5 HTTP 메소드(1/9) – 개요 1

✓ HTTP 동사는 URL에 포함하여 보낸 데이터로 무엇을 해야 할지 서버에게 알려 줍니다.

### ✓ GET 메소드

- 가장 간단한 동사이며, 안전하고, 멱등이며, 따라서 응답 결과를 캐시에 둘 수 있다.
- `http://api.nextree.com/v1.1/users/1234?active=true`

### ✓ POST 메소드

- 자원을 생성할 때 사용하며, 안전하지 않으며, 멱등도 아니다. 캐시 항목이 있다면 무효가 된다.
- 쿼리 매개 변수를 사용하지 않는다.
- `curl -X POST -d`{"name":"John", "username":"john", "phone":"222-1234-1234"}`  
http://api.nextree.com/v1.1/users`

### ✓ DELETE 메소드

- 멱등이지만 안전하지 않다. 여러 차례 호출이나 한 차례 호출이나 결과는 같다.
- `curl -X DELETE http://api.nextree.com/v1.1/users/1234`

### ✓ HEAD 메소드

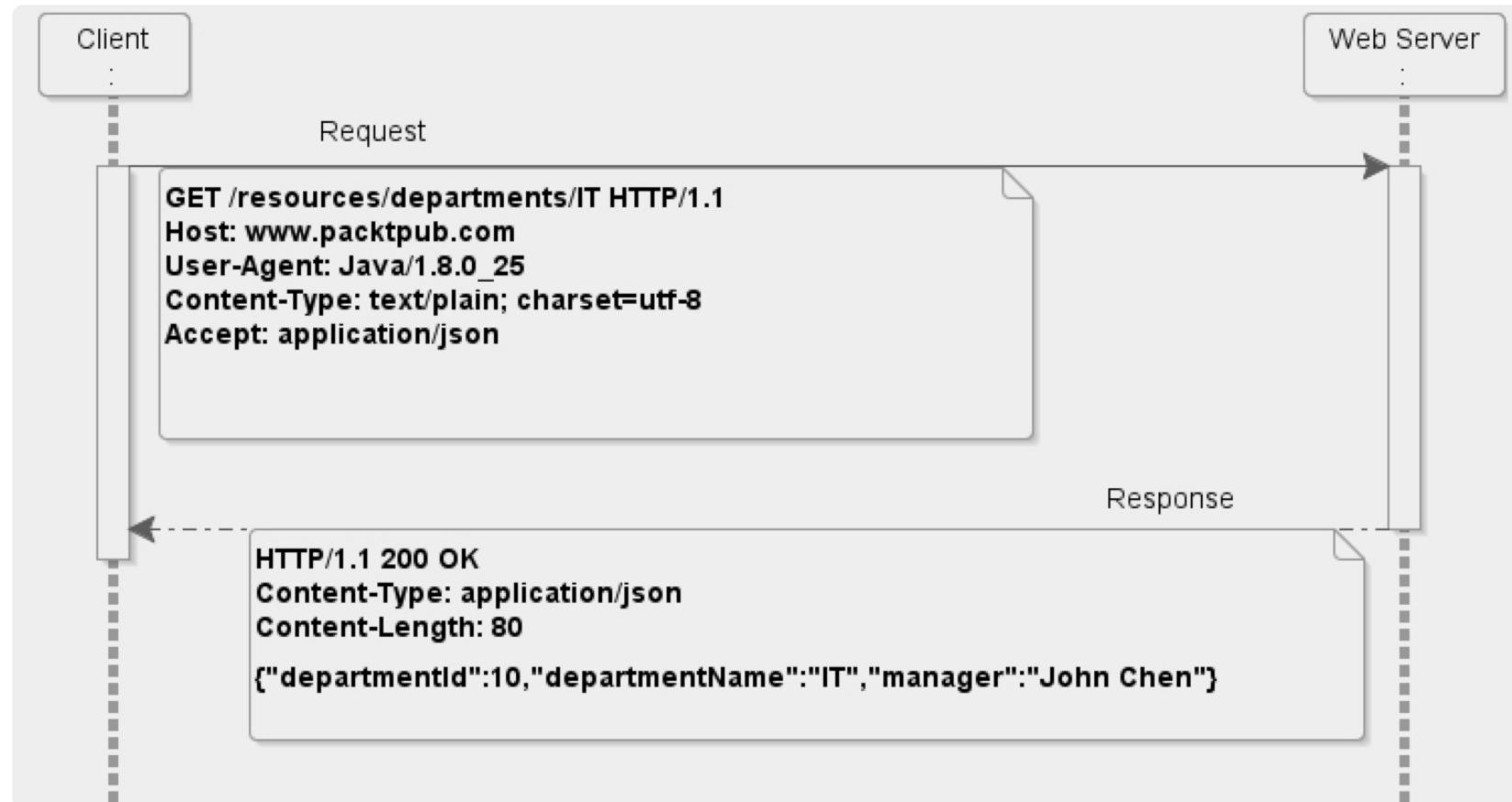
- GET과 동일한데, HTTP 헤더만 리턴한다. 멱등이며 안전하다.
- `curl -X http://api.nextree.com/v1.1/users`

## 2.5 HTTP 메소드(2/9) – 개요 2

- ✓ 앞으로 HTTP 메소드를 설명하는 사용할 부서(department) 정보에 대한 설명입니다.
- ✓ 부서의 예: JSON 표현(representation)은 다음과 같습니다.
  - { "department Id" :10 , "departmentName" : “ IT”, “manager” : “John Chen”}
- ✓ JSON으로 표현한 부서의 리스트는 다음과 같습니다.
  - [ { “department Id” :10 , “departmentName” : “IT”, “manager” : “John Chen”} ,  
▪ { “departmentId” : 20, “departmentName” : “Marketing”, “manager” : “Ameya Jn” } ,  
▪ { “departmentId” : 30 , “departmentName” : “HR”, “manager” : “Pat Fay”} ]
- ✓ 자원에 접근하는 URI를 정의합니다.
  - 부서(자원) 접근: <http://www.packtpub.com/resources/departments>
  - 이름으로 부서(자원) 접근: <http://www.packtpub.com/resources/departments/{name}>

## 2.5 HTTP 메소드(3/9) – GET 1

- ✓ [GET]은 자원 검색에 사용합니다. 컨텍스트 안에 어떤 자원이 있고, 어떤 표현이 가능한지 알아야 합니다.
- ✓ GET은 “안전하고(safety), 멱등인(idempotent)” 메소드입니다.
- ✓ 앞에서 살펴본 자원(resource)을 URI를 지정하여 가져오면(GET) 아래와 같은 결과를 리턴합니다.  
→ **GET** `http://www.packtpub.com/resources/departments/IT`



## 2.5 HTTP 메소드(4/9) – GET 2

- ✓ Java 클라이언트가 “IT”를 부서 id로 하여 GET 메소드 요청을 합니다.
- ✓ 클라이언트는 요청 헤더 필드의 “Accept”를 이용하여 표현(representation) 유형을 설정합니다. 이 요청 메시지는 자기 서술적(self-descriptive)이어서 내용을 보면 바로 이해할 수 있습니다.
  - 컨텐츠를 검색하기 위해 표준 메소드를 사용합니다. → GET
  - 컨텐츠 타입은 잘 알려진 미디어 타입(text/plain)으로 설정합니다. → text/plain
  - 받을 수 있는 응답 포맷을 선언합니다. → application/json
- ✓ 웹서버는 “GET” 요청을 받으면 RESTful 프레임워크로 요청을 넘깁니다.
- ✓ RESTful 프레임워크는 자원을 검색하지 않습니다. 자원 검색은 도메인 서비스 봇입니다.
- ✓ 서버의 뒷쪽에 있는 서비스는 URI에 정해진 조건에 따라 DB나 파일 등으로부터 자원을 검색합니다.
- ✓ 자원을 찾으면 클라이언트가 요청한 포맷(여기서는 JSON)으로 변환합니다.
- ✓ 서버는 200이라는 코드와 함께 JSON으로 변환된 자원을 response 메시지에 담아서 보냅니다. 응답 메시지 역시 자기 서술적이어야 내용을 보면 바로 이해할 수 있습니다.
- ✓ 서버에서 자원을 찾을 수 없을 경우 404, 예외가 발생했을 경우 500 등의 에러 코드를 리턴합니다.
- ✓ GET 요청을 위한 모든 호출은 표준 웹 호출 방식을 따릅니다.

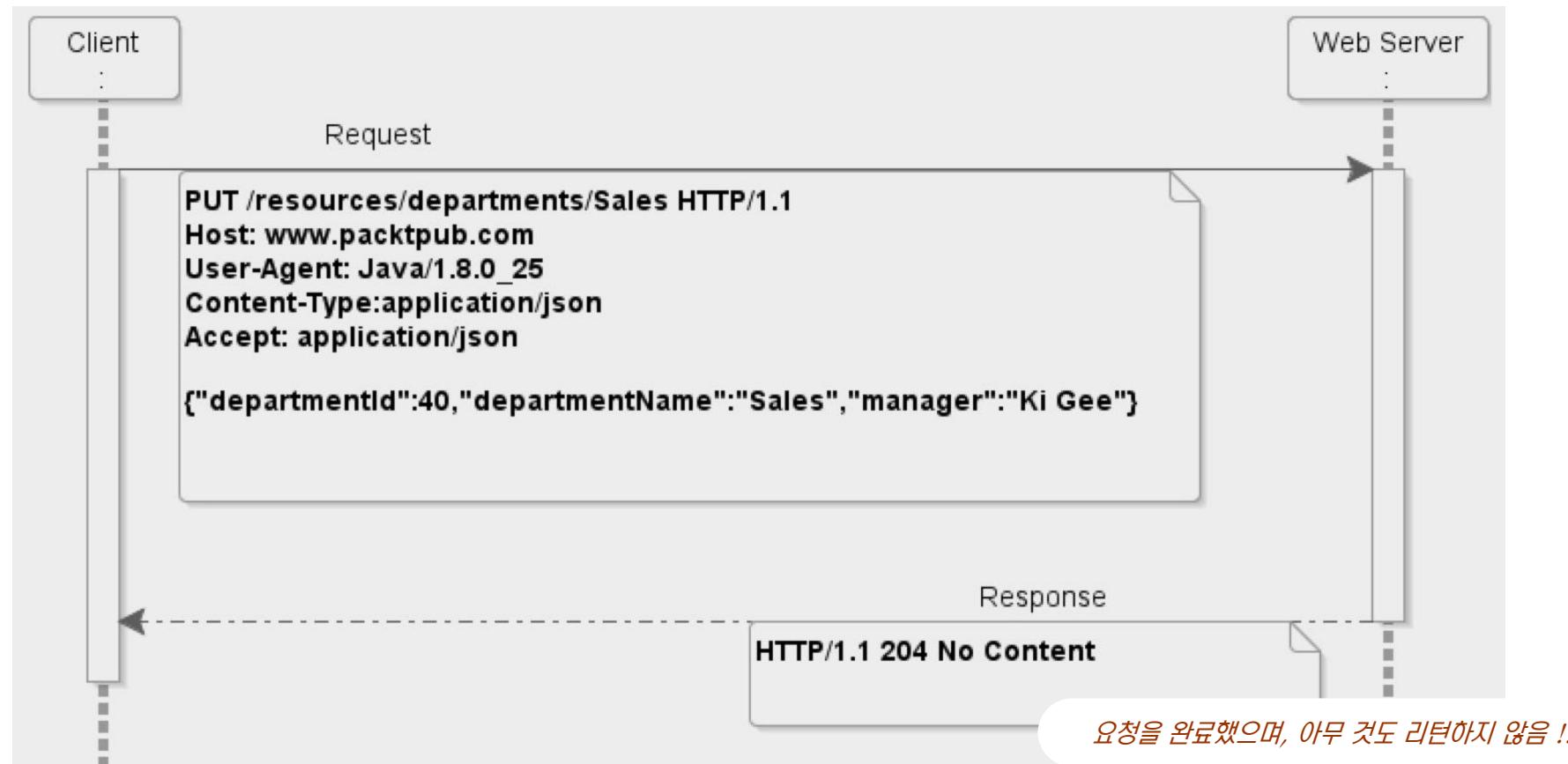
## 2.5 HTTP 메소드(5/9) – POST

- ✓ POST 메소드는 자원(resource)을 생성할 때 사용합니다.
- ✓ 부서(department)를 생성하기 위한 요청을 만들어 보면 아래와 같습니다.
- ✓ 리턴 값으로 생성된 리소스의 URI인 /resources/departments/40을 얻을 수 있습니다.



## 2.5 HTTP 메소드(6/9) – PUT

- ✓ PUT 메소드는 자원을 변경(update)할 때 사용합니다. PUT은 안전하지 않지만, 멱등입니다.
- ✓ 자원 URI를 지정하고, 페이로드로 변경할 [엔티티]값을 보냅니다.
- ✓ 변경 내역: {"departmentName":"Sales", "manager":"Tony Greig"} → : {"departmentName":"Sales", "manager":"Ki Gee"}



## 2.5 HTTP 메소드(7/9) – PATCH

- ✓ PATCH 메소드는 자원을 [부분적으로] 변경(update)할 때 사용합니다.
- ✓ 자원 URI를 지정하고, 페이로드로 변경할 [엔티티값 중에 필요한 값만] 을 보냅니다.
- ✓ 변경 내역: { "manager": "Ki Gee"}

[출처] <https://www.mnot.net/blog/2012/09/05/patch>

```
{  
  "name": "abc123",  
  "colour": "blue",  
  "count": 4  
}
```

PATCH /widgets/abc123 HTTP/1.1  
Host: api.example.com  
Content-Length: ...  
Content-Type: application/json-patch

```
[  
  {"replace": "/count", "value": 5}  
]
```

HTTP/1.1 200 OK  
Content-Type: text/plain  
Connection: close

Your patch

## 2.5 HTTP 메소드(8/9) – DELETE

- ✓ DELETE는 메소드는 자원을 삭제할 때 사용합니다. URI는 다른 메소드와 같습니다.
- ✓ URI는 다른 메소드와 같습니다.
- ✓ 자원이 성공적으로 삭제되면 204 코드를 리턴합니다.



## 2.5 HTTP 메소드(9/9) – HEAD

- ✓ HEAD는 GET 메소드와 비슷합니다. 컨텐츠를 리턴하지 않고 헤더 값만 리턴합니다.
- ✓ HEAD 요청은 GET 요청 처럼 안전하며, 멱등입니다.
- ✓ 서비스 가능 여부를 체크할 때 사용할 수 있습니다.
- ✓ 대량 데이터를 가져올 경우, GET 요청 전에 HEAD 요청을 이용하여 자원 변경 여부를 체크할 수 있습니다.
  - LastModified 체크
  - ContentLength 체크

```
curl -X HEAD http://api.nextree.co.kr/v1/users
```



### 3. 리소스 설계

- 
- 3.1 리소스 설계 개요
  - 3.2 REST 요청응답 패턴
  - 3.3 컨텐츠 조정
  - 3.4 도메인 객체와 리소스

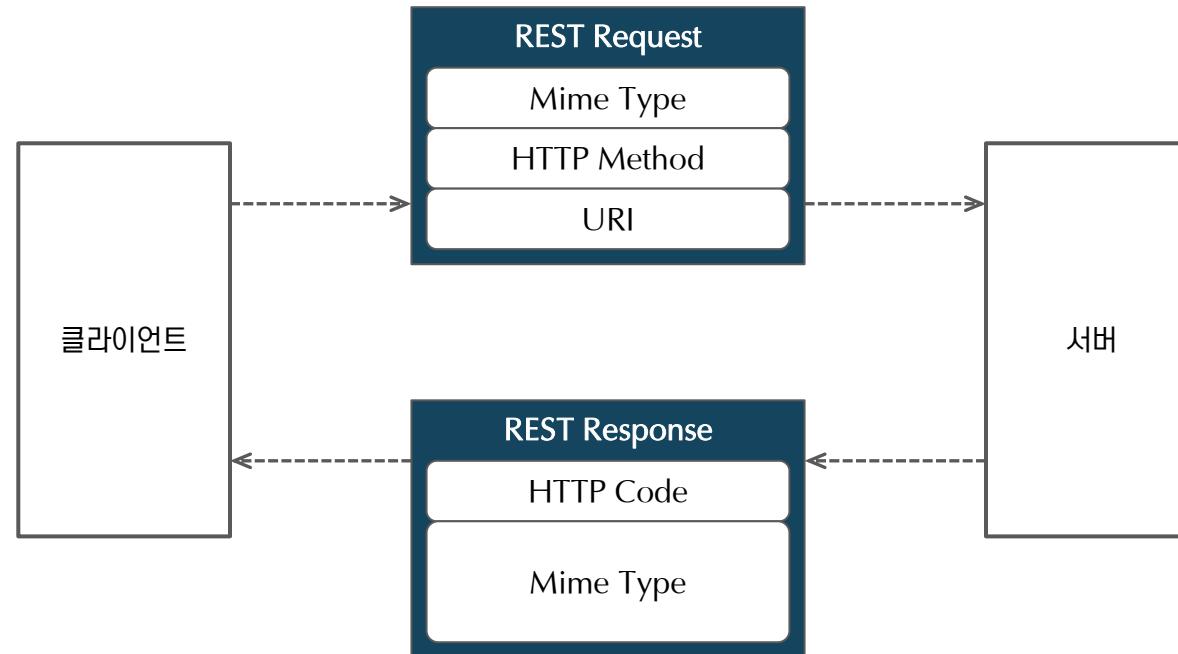
## 3.1 리소스 설계 개요

---

- ✓ RESTful 리소스 설계의 기본이 되는 REST 요청 및 응답 패턴을 이해합니다.
- ✓ 동일한 내용의 리소스이지만 클라이언트에 따라 표현 방법이 다른 경우 리소스의 여러 표현 방안을 알아봅니다.
- ✓ API는 꾸준히 변화하고 성장합니다. 이러한 변경에 대응하는 API 버전 전략에 대해 알아봅니다.
- ✓ REST 응답의 표준 HTTP 코드 사용 방안을 이해합니다.

## 3.2 REST 요청 응답 패턴

- ✓ REST 요청 및 응답의 기본적인 패턴은 클라이언트가 요청을 보내고 서버가 응답하는 방식입니다.
- ✓ 클라이언트는 URI, 표준 HTTP 메소드, MIME 타입으로 구성된 REST 요청을 보냅니다.
- ✓ 서버는 요청을 처리 후 표준 HTTP 응답 코드와 MIME 타입으로 조합하여 요청자에게 응답을 되돌려 줍니다.



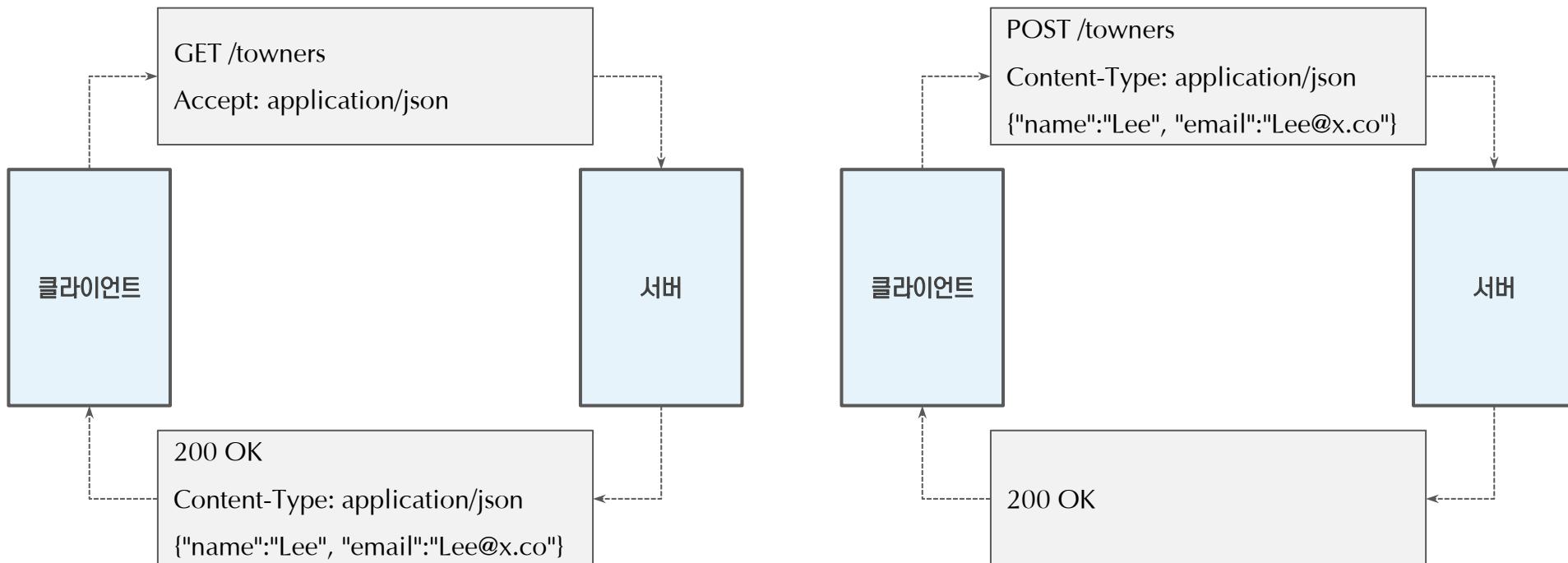
### 3.3 컨텐츠 조정(Content negotiation)

- ✓ 컨텐츠 조정은 동일한 URI로 리소스를 다르게 표현할 경우 사용합니다.
- ✓ 서버가 동일한 리소스에 대하여 여러 표현형태를 제공하는 경우 클라이언트는 적합한 표현형태를 선택합니다.
- ✓ 주로 HTTP 헤더를 사용하거나 URL 패턴을 사용하여 컨텐츠 조정이 일어납니다.
- ✓ HTTP 헤더를 통한 방법이 비즈니스로 부터 기술적 관심사를 분리하기 쉬우므로 이 방법을 선호하고 있습니다.

HTTP 헤더 사용	URL 패턴 사용
GET /towners HTTP/1.1 Accept: application/json	GET /towners.json HTTP/1.1
GET /towners HTTP/1.1 Accept: application/xml	GET /towners.xml HTTP/1.1

### 3.3 컨텐츠 조정(1/3) – HTTP 헤더 사용

- ✓ 동일한 리소스 요청(동일한 URL, Method)이지만 HTTP 헤더의 내용에 따라 요청/응답 컨텐츠가 조정됩니다.
- ✓ 즉, 헤더 정보로 동일한 내용을 해당 표현방식으로 해석하거나 표현합니다.(예: json, xml)
- ✓ 클라이언트 요청 정보에 "Accept" HTTP 헤더가 있으면 서버는 응답 컨텐츠를 해당 타입으로 표현합니다.
- ✓ POST, PUT 메소드와 같이 리소스를 변경할 때 "Content-Type" 헤더로 요청 컨텐츠를 해석할 수 있습니다.



### 3.3 컨텐츠 조정(2/3) – 지원 Annotation 1

- ✓ JAX-RS 명세는 표준 Annotation 형태로 컨텐츠 조정을 지원합니다.
- ✓ 대표적으로 javax.ws.rs.Produces와 javax.ws.rs.Consumes Annotation을 주로 사용합니다.
- ✓ Produces Annotation은 리소스를 어떤 MIME 타입으로 응답할지 식별하는데 사용합니다.
- ✓ 클라이언트의 Accept 헤더에 매치되는 유형으로 변환하며 매치되지 않은 경우 406 응답코드를 반환합니다.

Produces Annotation	HTTP 요청	HTTP 응답
<pre>@GET @Path("towners") @Produces(MediaType.APPLICATION_JSON) public List&lt;Towner&gt; findAllTowners() {     //     return townerService.findAll(); }</pre>	GET /namooclub/towners HTTP/1.1	HTTP/1.1 200 OK Content-Type: application/json [{"name":"Lee", "email":"lee@nextree.co.kr", "password":"1234"}, ...]
	GET /namooclub/towners HTTP/1.1 Accept: application/xml	HTTP/1.1 406 Not Acceptable

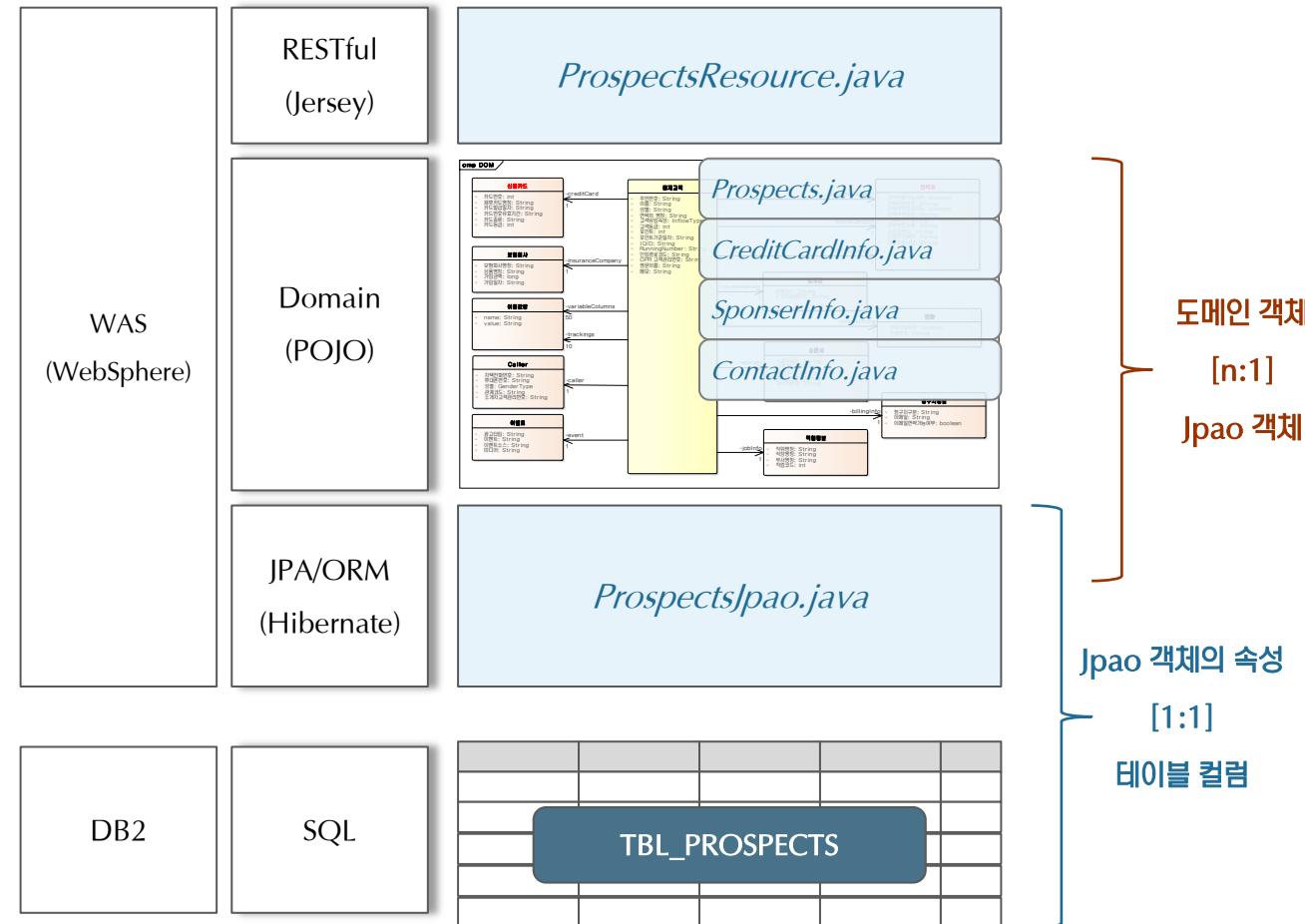
### 3.3 컨텐츠 조정(3/3) – 지원 Annotation 2

- ✓ Consumes Annotation은 어떤 Media 타입의 요청 데이터를 수용할 수 있는지 명시합니다.
- ✓ 따라서 Consumes는 주로 리소스의 변경과 관련되어 사용되며 Produces와 같이 사용되기도 합니다.
- ✓ 클라이언트의 Content-Type 헤더와 매치되는 Media 타입으로 요청 Body를 수용합니다.
- ✓ 매치되는 Media 타입이 식별되지 않는 경우 415(Unsupported Media Type) 응답코드를 반환합니다.

Consumes Annotation	HTTP 요청	HTTP 응답
<pre>@POST @Path("towners") @Consumes(MediaType.APPLICATION_JSON) @Produces(MediaType.APPLICATION_JSON) public Response addTowner(Towner towner) {     townerService.addTowner(towner);     return     Response.ok(townerService.findAll()).build(); }</pre>	<p>POST /namooclub/towners HTTP/1.1 Content-Type: application/json {"name":"hong", "email":"hong@nextree.co.kr", "password":"1234"}</p>	<p>HTTP/1.1 200 OK Content-Type: application/json [{"name":"Lee", "email":"lee@nextree.co.kr", "password":"1234"}, ...]</p>
	<p>POST /namooclub/towners HTTP/1.1 Content-Type: application/xml {"name":"hong", "email":"hong@nextree.co.kr", "password":"1234"}</p>	<p>HTTP/1.1 415 Unsupported Media Type</p>

## 3.4 도메인 객체와 리소스

- ✓ 테이블, 도메인 객체, 자원 객체 간의 관계 설정을 어떻게 해야 하는가?
- ✓ 도메인 객체에서 엔티티 객체와 값 객체에 대한 구분이 필요한가?



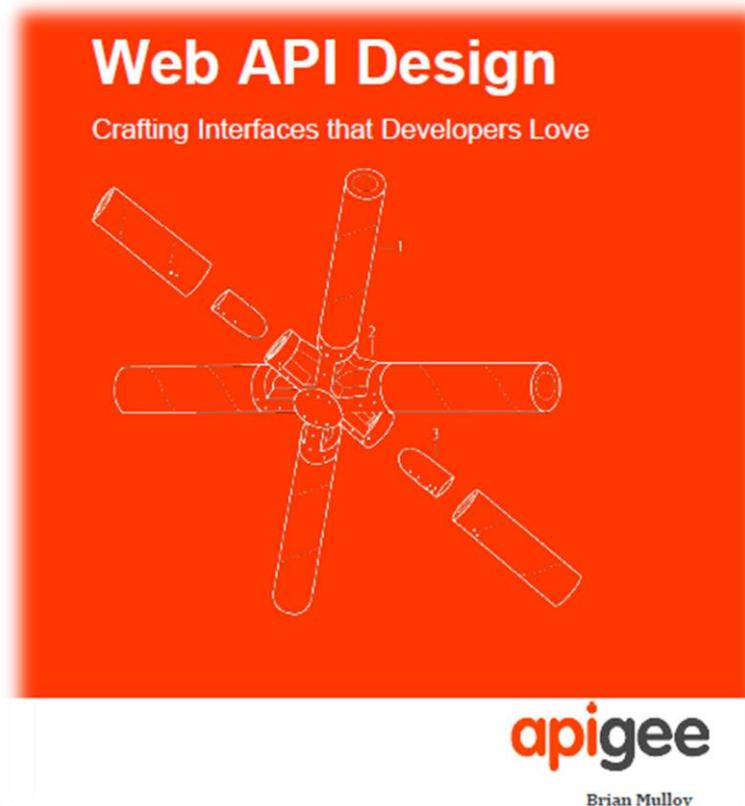


## 4. RESTful API 설계

- 
- 4.1 실용주의 REST API
  - 4.2 RESTful 서비스 설계 원칙
  - 4.3 명사와 동사
  - 4.4 연관 관계 단순화
  - 4.5 에러 처리
  - 4.6 버전 텁
  - 4.7 페이지 처리와 부분 응답
  - 4.8 자원이 없는 API
  - 4.9 API 호출 예제

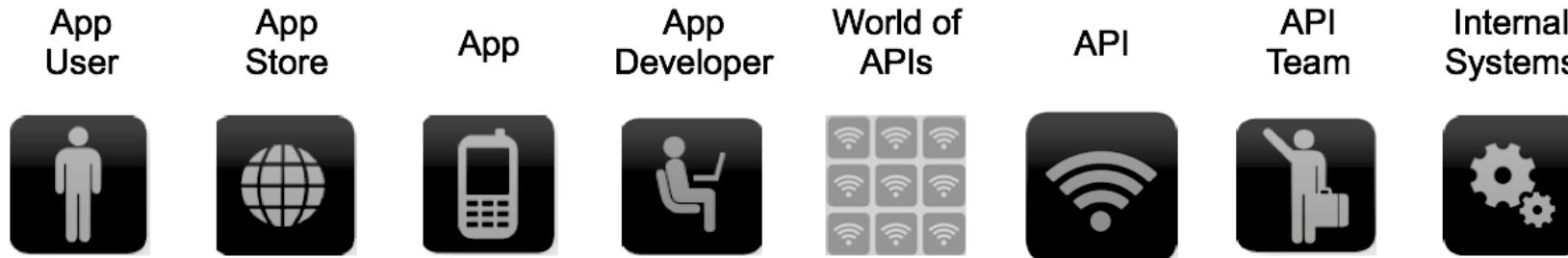
## 4.1 실용주의 REST API (1/3)

- ✓ 이 장은 apigee([www.apigee.com](http://www.apigee.com)) 사의 eBook을 기반으로 진행합니다. ← 무료 다운로드 가능함
- ✓ REST API 설계는 아키텍처 스타일이지 표준이 아닙니다. 적용에 유연한 입장을 취할 수 있습니다.
- ✓ API 설계의 핵심은 실용주의 관점을 유지하는 REST입니다.
- ✓ API 설계는 얼마나 많은 개발자들이 빨리 여러분의 API를 이용하느냐로 증명됩니다.



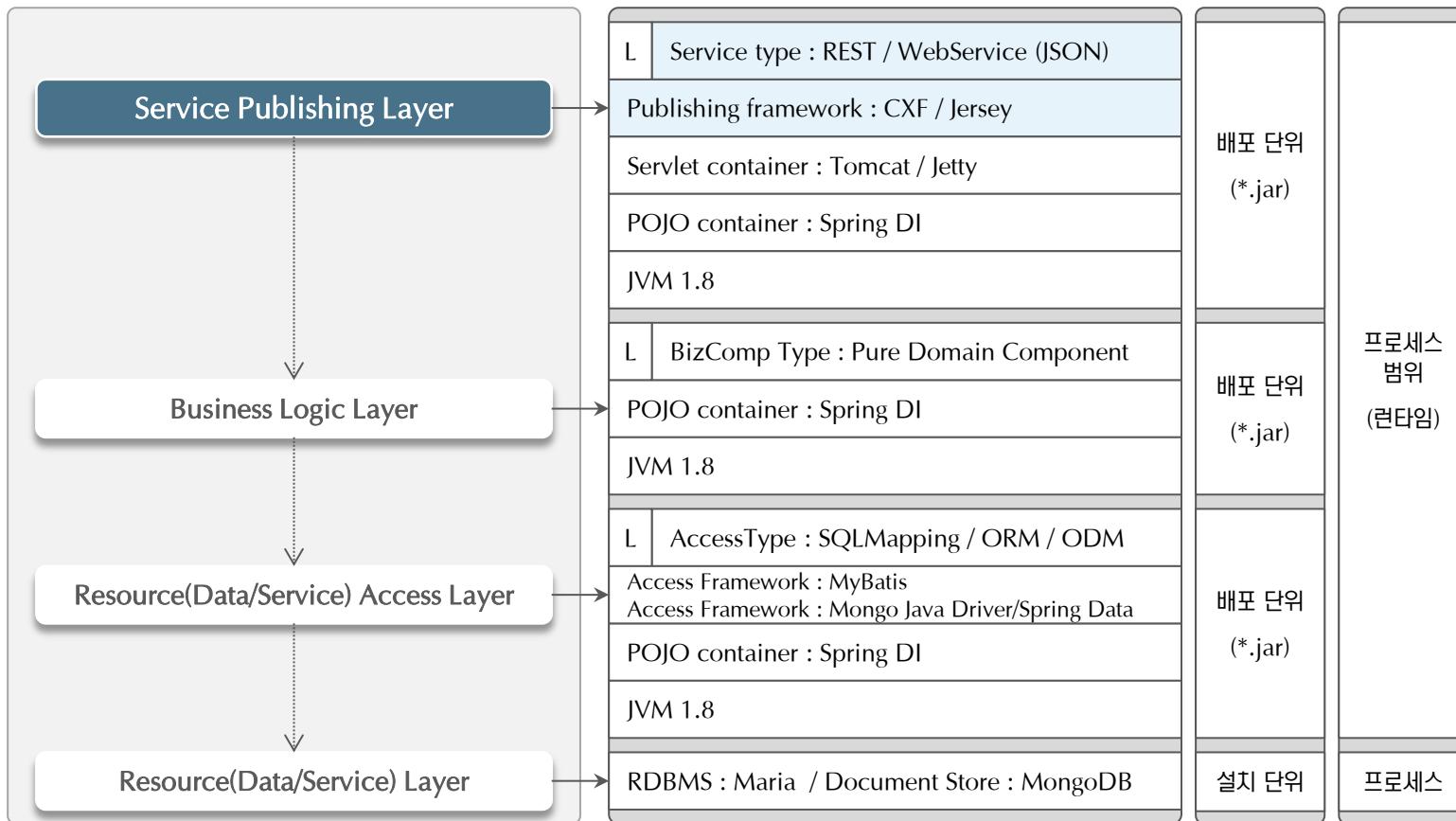
## 4.1 실용주의 REST API (2/3)

- ✓ API의 목표는 API를 사용하는 개발자의 성공을 돋는 것입니다.
- ✓ 따라서 API 설계의 첫번째 원칙은 개발자의 생산성과 성공을 극대화 하는 것입니다. ← 실용주의 REST
- ✓ 개발자의 관점에서 보면 많은 팁과 경험을 포함하는 훌륭한 가이드가 가장 중요합니다.



## 4.1 실용주의 REST API (3/3)

- ✓ RESTful API는 애플리케이션 레이어에서 보면 서비스 발행 레이어에 놓입니다.
- ✓ 아키텍처 설계 - 서비스 발행 플랫폼(또는 프레임워크)를 선택하고 RI를 개발한 후, API 가이드를 만듭니다.
- ✓ 도메인 모델(또는 서비스 명세) – API 가이드에 따라 외부로 서비스 API를 정의한 후 발행합니다.



## 4.2 RESTful 서비스 설계 원칙

---

- ✓ 자원의 URI를 식별한다. ← 어떤 명사로 자원을 표현할지 결정한다.
- ✓ 자원이 지원하는 메소드를 식별한다. ← CRUD 오퍼레이션을 위한 다양한 HTTP 메소드를 식별한다.
- ✓ 자원이 지원하는 서로 다른 표현을 식별한다. ← 표현을 어떻게 할지 결정한다. JSON, XML, HTML
- ✓ JAX-RS API를 이용하여 RESTful 서비스를 구현한다. ← JAX-RS 명세를 준수한다.
- ✓ RESTful 서비스를 배포한다. ← 컨테이너를 결정하고, war 파일을 구성한다.
- ✓ RESTful 서비스를 테스트한다. ← 서비스를 테스트하기 위한 클라이언트를 구현한다.

## 4.2 RESTful 서비스 설계 원칙

- ✓ 자원의 URI를 식별한다. ← 어떤 명사로 자원을 표현할지 결정한다.

/v1.1/library/books	도서관의 책(book) 자원을 표현한다.
/v1.1/library/books/isbn/123456	ISBN 번호가 “123456”인 어떤 책을 표현한다.
/v1.1/coffees	카페에서 판매한 모든 커피를 표현합니다.
/v1.1/coffees/orders	주문된 모든 커피를 표현합니다.
/v1.1/coffees/orders/123	주문번호가 “123”인 커피 주문을 표현합니다.
/v1.1/users/123	아이디가 “123”인 사용자를 표현합니다.
/v1.1/users/345/books	아이디가 “345”인 사용자를 위한 모든 책을 표현합니다.

## 4.2 RESTful 서비스 설계 원칙

- ✓ 자원이 지원하는 메소드를 식별한다. ← CRUD 오퍼레이션을 위한 다양한 HTTP 메소드를 식별한다.

GET	/v1.1/library/books	도서관의 책(book) 목록을 가져옵니다.
GET	/v1.1/library/books/isbn/123456	ISBN 번호가 “123456”인 어떤 책을 가져옵니다.
POST	/v1.1/library/books/orders	새로운 책 주문을 합니다.
DELETE	/v1.1/library/books/isbn/123456	ISBN 번호가 “123456”인 책을 삭제합니다.
PUT	/v1.1/library/books/isbn/123456	ISBN 번호가 “123456”인 책의 정보를 갱신합니다.
PATCH	/v1.1/library/books/isbn/123456	ISBN 번호가 “123456”인 책의 정보 중에 일정 부분만 갱신합니다.

## 4.3 명사와 동사 (1/3)

- ✓ 실용적인 RESTful 디자인의 첫번째 원칙은 간결하고 직관적인 기준 URL을 유지하는 것입니다.
- ✓ 기준 URL 설계는 API에서 가장 중요한 설계 행동유도성(affordance)입니다.
- ✓ 행동유도성은 가이드 문서가 필요없는 설계 재산입니다. ← 직관적인 설계



설계 행동유도성(affordance)과 문서 간의 충돌 !!

## 4.3 명사와 동사 (2/3)

- ✓ 자원(resource) 별로 두 개의 기준 url을 사용합니다.
- ✓ 기준 url에는 동사를 두지 않습니다.
- ✓ 컬렉션이나 요소들을 다룰 때는 HTTP 동사(??)를 사용합니다.

// 목록을 위한 URL

/dogs

// 목록 중 특정 개체를 위한 URL

/dogs/1234

Resource	POST(create)	GET(read))	PUT(update)	DELETE(delete)
/dogs	새로운 dog 생성	dogs 목록	dogs에 대한 대량 업데이트	모든 dogs 삭제
/dogs/1234	예러	특정 dog 보기	있으면 업데이트, 없으면 예러	삭제

## 4.3 명사와 동사 (3/3) – 복수 명사와 구체적인 이름

- ✓ 개발자가 API를 이용할 때, 예측하거나 추측할 수 있도록 일관성을 유지해야 합니다.
- ✓ 직관적인 API는 단수 명사 보다는 복수를 사용합니다.
- ✓ 좋은 API는 추상적인 이름이 아닌 구체적인 이름을 사용합니다.
- ✓ /items이나 /assets 보다는 /blogs나 /videos라는 식으로 구체적인 이름을 사용하는 것이 좋습니다.

Foursquare	GroupOn	Zappos
/checkins	/deals	/Product

## 4.4 연관관계 단순화

- ✓ URL을 5내지 6레벨까지 구성하는 것은 읽기 어려울 뿐만 아니라 바람직하지 않습니다.
- ✓ 자원 간의 연관관계를 단순하게 유지하는 방법으로 API의 직관성을 유지해야 합니다.
- ✓ HTTP 물음표(?) 아래 여러 패러미터를 두고 복잡성을 감주어야 합니다.

```
GET /dogs?color=red&state=running&location=park
```

## 4.5 에러 처리 (1/3)

- ✓ 실용주의 REST API에서는 에러를 어떻게 처리하는가?
- ✓ Facebook은 상태 코드가 200과 #803 에러가 났다고 하지만 그 에러가 무엇인지 알려주지 않습니다.
- ✓ Twilio는 상태코드가 401이며, 에러와 에러에 해당된 내용을 확인할 수 있는 문서정보까지 제공합니다.
- ✓ SimpleGeo는 상태코드가 401이며, 에러코드 외에 다른 정보는 제공하지 않습니다.

### Facebook

HTTP Status Code: 200

```
{"type": "OAuthException", "message": "#803 Some of the aliases you requested do not exist: foo.bar"}
```

### Twilio

HTTP Status Code: 401

```
{"status": "401", "message": "Authenticate", "code": 20003, "more info": "http://www.twilio.com/docs/errors/20003"}
```

### SimpleGeo

HTTP Status Code: 401

```
{"code": 401, "message": "Authentication Required"}
```

## 4.5 에러 처리 (2/3)

- ✓ HTTP 상태 코드를 사용합니다.
- ✓ 실제 상태 코드는 세 가지 정도만 제공하면 되고 필요하면 더 추가할 수 있습니다.
- ✓ 더 추가할 경우, 최대 8개를 넘지 않아야 합니다.

### 추천 기본 상태

200 - OK

400 - Bad Request

500 - Internal Server Error

### 추가상태

201 - Created

304 - Not Modified

404 - Not Found

401 - Unauthorized

403 - Forbidden

## 4.5 에러 처리 (3/3)

- ✓ 상태코드를 사용하더라도 사용자에게 가능한 자세한 메시지를 제공하여야 합니다.

### 코드를 위한 코드

200 - OK 401 - Unauthorized

### 사용자를 위한 메시지

```
{"developerMessage": "Verbose, plain language description of the problem for the app developer with hints about how to fix it.", "userMessage": "Pass this message on to the app user if needed.", "errorCode": 12345, "more info": "http://dev.teachdogrest.com/errors/12345"}
```

## 4.6 버전 팀 (1/2)

- ✓ 버전 없이 API를 릴리즈하면 안됩니다. 그리고 버전은 옵션이 아닌 필수입니다.
- ✓ 다른 서비스에서 사용하는 versioning 예제입니다.
  - Twilio URL에 timestamp를 사용합니다.
  - Salesforce.com은 URL의 중간 지점에 v20.0과 같은 버전 정보를 둡니다.
  - Facebook은 v. 와 같은 버전 표기법을 사용하지만 버전 정보는 선택가능한 패러미터입니다.

**Twilio**

/2010-04-01/Accounts/

**salesforce.com**

/services/data/v20.0/sobjects/Account

**Facebook**

?v=1.0

## 4.6 버전 팀 (2/2)

---

### ✓ 실용주의적인 REST에서 버전 번호를 어떻게 처리해야 하는가?

- 단순한 서수를 사용합니다.
- 적어도 하나의 버전을 유지합니다.
- 버전을 폐기하기 전에 개발자들이 대응할 수 있는 시간을 충분히(한 사이클 정도) 주어야 합니다.

## 4.7 페이지 처리와 부분 응답 (1/2)

- ✓ “부분 응답”은 개발자들이 필요로 하는 것만 제공하는 것입니다.
- ✓ 아래 정보는 3가지 서비스에서 제공하고 있는 “부분 응답” 예제입니다.
- ✓ 다음은 각 서비스에서 쉼표로 구분한 옵션 필드 등을 이용하여 제공하는 “부분 응답” API 예제입니다.

### LinkedIn

/people:(id,first-name,last-name,industry)

### Facebook

/joe.smith/friends?fields=id,name,picture

### Google

?fields=title,media:group(media:thumbnail)

## 4.7 페이지 처리와 부분 응답 [2/2]

- ✓ 개발자가 페이지 처리를 하기 쉽게 offset과 limit을 제공하도록 합니다.
- ✓ 다음은 offset과 limit 값의 예입니다.

**Facebook**

offset 50 and limit 25

**Twitter**

page 3 and rpp 25 (페이지 당 레코드 개수)

**LinkedIn**

start 50 and count 25

## 4.8 자원이 없는 API

- ✓ DB에 저장된 자원(resource)과 관련이 없는 응답을 처리할 때는 다음과 같이 하는 것을 추천합니다.
  - 명사가 아닌 동사를 사용합니다.
  - 비자원 처리 API는 다른 시나리오를 사용함으로 명확하게 서술합니다.
- ✓ 아래는 100유로를 중국 yen으로 바꿀 때의 예제입니다.

```
/convert?from=EUR&to=CNY&amount=100
```

## 4.9 API 호출 예제 (1/3)

- ✓ 이름이 AI인 갈색 강아지 객체를 생성합니다.
- ✓ AI라는 이름을 가진 강아지의 이름을 Rover로 이름을 변경합니다.

### Request

```
POST /dogs  
name=AI&furColor=brown ← 바디로 전달
```

### Response

```
200 OK  
{ "dog": { "id": "1234", "name": "AI", "furColor": "brown" } }
```

### Request

```
PUT /dogs/1234  
name=Rover ← 바디로 전달
```

### Response

```
200 OK  
{ "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }
```

## 4.9 API 호출 예제 (2/3)

- ✓ 특정 강아지 정보를 조회합니다.
- ✓ 모든 강아지 정보를 가져옵니다.

**Request**

```
GET /dogs/1234
```

**Response**

```
200 OK
```

```
{ "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }
```

**Request**

```
GET /dogs
```

**Response**

```
200 OK
```

```
{ "dogs": [  
    { "dog": { "id": "1233", "name": "Fido", "furColor": "white" } },  
    { "dog": { "id": "1234", "name": "Rover", "furColor": "brown" } }  
],  
"_metadata": [ { "totalCount": 327, "limit": 25, "offset": 100 } ]  
}
```

## 4.9 API 호출 예제 (3/3)

- ✓ 1234라는 아이디를 가진(여기서는 Rover라는 이름을 가진) 강아지 정보를 삭제합니다.

### Request

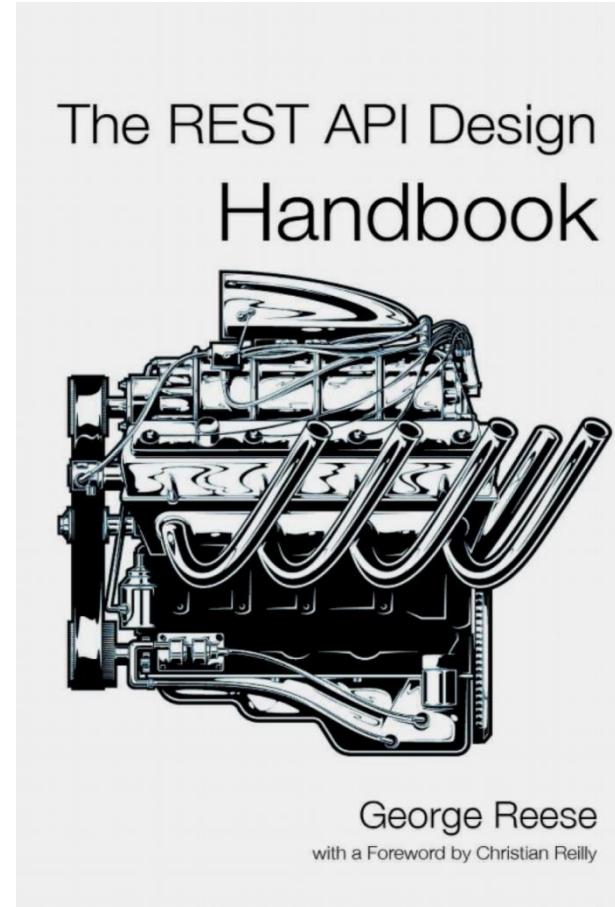
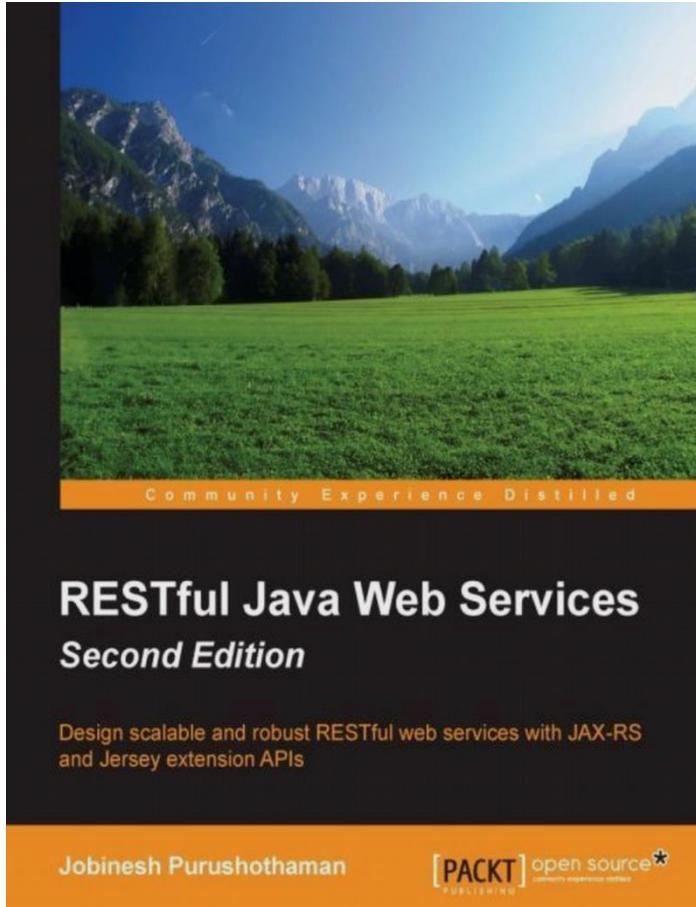
```
DELETE /dogs/1234
```

### Response

```
200 OK
```

# [참조] 추천 도서

✓ API 명세를 위한 추천 도서와 구현을 위한 추천 도서입니다.





## 5. JAX-RS API 소개

- 
- 5.1 JAX-RS 개요
  - 5.2 JAX-RS 어노테이션
  - 5.3 추가 메타-데이터 리턴
  - 5.4 RESTful 웹 서비스 만들기 (실습)
  - 5.5 RESTful 웹 서비스 만들기 (실습) - 클라이언트 API
  - 5.6 Summary

## 5.1 JAX-RS 개요

- ✓ RESTful 웹 서비스 발행을 위한 여러 프레임워크가 있습니다.
- ✓ 그래서 다양한 기술을 통합하고자 표준화(standardization)를 진행하였습니다.
- ✓ 그 결과 Java EE6에서 JAX-RS(JSR 311) 표준을 제정하였습니다.
- ✓ 현재는 JAX-RS2(JSR 339)가 RESTful 웹 서비스의 최신 표준입니다.

Jersey	오픈소스, JAX-RS 참조 구현체, <a href="https://jersey.java.net">https://jersey.java.net</a>
Apache CXF	오픈소스, JAX-RS, JAX-WS 모두 지원, <a href="http://cxf.apache.org">http://cxf.apache.org</a>
RESTEasy	오픈소스(from JBoss), <a href="http://resteasy.jboss.org">http://resteasy.jboss.org</a>
Restlet	오픈소스, 경량 프레임워크, 모바일 플랫폼에 적합, <a href="http://restlet.com">http://restlet.com</a>

- ✓ *RESTful web services*를 위한 코드를 작성할 때 JAX-RS 표준 API를 사용하는 것이 좋습니다.
- ✓ 표준을 바탕으로 개발하면 벤더로부터 자유롭습니다.
- ✓ 언제든지 구현체를 교체할 수 있으며, 소스 코드에 미치는 영향을 최소화 할 수 있습니다.

## 5.2 JAX-RS 어노테이션(1/23)

- ✓ JAX-RS 명세를 사용하면 쉽게 RESTful 웹 서비스를 개발하고, 발행(publishing)할 수 있습니다.
- ✓ JAX-RS는 Java EE 표준 스펙의 일부입니다.
- ✓ 따라서 Java EE 스펙을 구현한 서버(WAS)에 쉽게 이식, 교체가 가능합니다.
- ✓ JAX-RS 명세는 Java annotation 으로 작성합니다.

### JAX-RS API 메이븐 의존성

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0.1</version><!-- set the tight version -->
    <scope>provided</scope><!-- compile time dependency -->
</dependency>
```

## 5.2 JAX-RS 어노테이션(2/23) –@Path

- ✓ 클래스 이름 → javax.ws.rs.Path
- ✓ REST 자원의 URI 경로를 기술합니다. class와 method 레벨로 작성할 수 있습니다.
- ✓ URI 패턴: http:// host:port/<context-path>/<application-path>
- ✓ JAX-RS를 이용하여 application-path 부분을 작성합니다.

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("departments")
public class DepartmentService {

    @GET
    @Path("count")
    public Integer getTotalDepartments() {
        return findTotalRecordCount();
    }
    //Rest of the code goes here
}
```

getTotalDepartments() 메소드를  
http://host:port/<context-path>/departments/count로 발행합니다.

## 5.2 JAX-RS 어노테이션(3/23) –@Path

- ✓ URI Path 템플릿을 이용하여 RESTful URI를 작성합니다.
- ✓ URI Path로부터 실행 시점에 변수를 받을 수 있습니다.
- ✓ 클라이언트는 웹 서비스 요청 시 RESTful URI 설계에 맞게 요청해야 합니다.

```
import javax.ws.rs.Path;
import javax.ws.rs.DELETE;

@Path("departments")
public class DepartmentService {

    @DELETE
    @Path("{id}")
    public void removeDepartment(@PathParam("id") short id) {
        removeDepartmentEntity(id);
    }
    //Other methods removed for brevity
}
```

removeDepartment() 메소드를  
http://host:port/<context-path>/departments/10 와 같이 접근합니다.  
예제는 HttpMethod.DELETE인 경우에 호출됩니다.

@Path("{id}") 와 같이 식별자를 정의하고,  
@PathParam ("{id}") 와 같이 식별자 이용하여 변수를 메소드의 parameter로 받을  
수 있습니다.

## 5.2 JAX-RS 어노테이션(4/23) –@Path

- ✓ URI Path의 경로를 정규식(regular expressions)으로 제한할 수 있습니다.
- ✓ 정규식에 맞지 않은 요청은 REST resource를 찾지 못합니다.(404)
- ✓ 정규식에 대한 설명은 김재훈 님의 블로그 글(<http://www.nextree.co.kr/p4327>)을 참조합니다.

부서를 부서 이름으로 삭제하는 예제입니다.  
부서이름은 반드시 영문숫자 조합이어야 URI로 요청할 수 있습니다.

```
@DELETE  
@Path("{name: [a-zA-Z][a-zA-Z_0-9]}")  
public void removeDepartmentByName(@PathParam("name") String deptName) {  
    //Method implementation goes here  
}
```

## 5.2 JAX-RS 어노테이션(5/23) – 미디어 타입

---

- ✓ HTTP 프로토콜 헤더의 Content-Type은 Request body의 content type을 나타냅니다.
- ✓ Content-Type은 표준 인터넷 Media type을 표현합니다.
- ✓ RESTful 웹 서비스는 HTTP body의 타입을 표현하기 위해 HTTP 프로토콜에 정의한 이 속성을 활용합니다.
- ✓ JAX-RS에서는 @javax.ws.rs.Produces, @javax.ws.rs.Consumes 어노테이션으로 기술합니다.

## 5.2 JAX-RS 어노테이션(6/23) – 미디어 타입 – @Produces

- ✓ 어노테이션 클래스 → @javax.ws.rs.Produces
- ✓ REST 자원 클래스의 메소드가 리턴하는 인터넷 미디어 타입을 기술합니다.
- ✓ 클래스 레벨에 기술하면 전체 메소드의 기준값으로 적용됩니다.
- ✓ 메소드 레벨에 기술하면 클래스 레벨의 설정을 덮어 씁니다.

Produces 인터넷 미디어 타입
application/atom+xml
application/json
application/octet-stream
application/svg+xml
application/xhtml+xml
application/xml
text/html
text/plain
text/xml

```
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("departments")
@Produces(MediaType.APPLICATION_JSON)
public class DepartmentService{

    //Class implementation goes here...
}
```

예제는 classs 레벨에 적용된 @Produces 을 보여줍니다.  
default로 이 class의 REST resource는 http응답을  
JSON으로 보내게 됩니다.

## 5.2 JAX-RS 어노테이션(7/23) – 미디어 타입 – @Consumes

- ✓ 어노테이션 클래스 → `@javax.ws.rs.Consumes`
- ✓ REST 자원이 받을 수 있는 인터넷 미디어 타입을 기술합니다.
- ✓ `@Produces`와 마찬가지로 클래스, 메소드 레벨로 기술할 수 있으며, 적용 방식은 동일합니다.

Consumes 인터넷 미디어 타입
application/atom+xml
application/json
application/octet-stream
application/svg+xml
application/xhtml+xml
application/xml
text/html
text/plain
text/xml
multipart/form-data
application/x-www-form-urlencoded

예제는 method 레벨에 적용된 `@Consumes`을 보여줍니다.  
이 REST resource는 JSON 타입의 HTTP request body를  
받을 수 있고, 이를 객체로 변환하여 method의 파라미터로 받게  
됩니다.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.POST;

@POST
@Consumes(MediaType.APPLICATION_JSON)
public void createDepartment(Department entity) {

    //Method implementation goes here...
}
```

## 5.2 JAX-RS 어노테이션(8/23) – HTTP 메소드 처리

---

- ✓ RESTful 웹 서비스는 HTTP 프로토콜 표준 메소드 타입을 사용합니다.
- ✓ HTTP 메소드 → GET, PUT, POST, DELETE, HEAD, OPTIONS
- ✓ JAX-RS 어노테이션으로 REST 자원의 HTTP 메소드 타입을 기술합니다.
- ✓ RESTful URI 설계를 가능하게 합니다.

## 5.2 JAX-RS 어노테이션(9/23) – HTTP 메소드 처리 – @GET

- ✓ 어노테이션 클래스 → @javax.ws.rs.GET
- ✓ REST 자원을 조회할 때 사용합니다.
- ✓ HTTP의 GET 메소드입니다.

```
//imports removed for brevity
@Path("departments")
public class DepartmentService {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Department> findAllDepartments() {
        //Find all departments from the data store
        List<Department> departments = findAllDepartmentsFromDB();
        return departments;
    }
    //Other methods removed for brevity
}
```

예제는 전체 부서 목록을 조회하는 REST resource입니다.  
http://host:port/<context-path>/departments  
@GET annotation으로 HTTP method를 기술하고 있습니다.

## 5.2 JAX-RS 어노테이션(10/23) – HTTP 메소드 처리 – @PUT

- ✓ 어노테이션 클래스 → @javax.ws.rs.PUT
- ✓ REST resource를 update하거나 create 할 때 사용합니다. ← 주의 생성할 때만 사용해야 함

예제는 부서 정보를 update 하는 REST resource입니다.

http://host:port/<context-path>/departments/{id}

@PUT annotation으로 HTTP method를 기술하고 있습니다.

```
@PUT
@Path("{id}")
@Consumes(MediaType.APPLICATION_JSON)
public void editDepartment(@PathParam("id") Short id, Department department) {

    //Updates department entity to data store
    departmentService.modifyDepartment (id, department);

}
```

## 5.2 JAX-RS 어노테이션(11/23) – HTTP 메소드 처리 – @POST

- ✓ 어노테이션 클래스 → @javax.ws.rs.POST
- ✓ REST 자원을 생성할 때 사용합니다.

예제는 부서를 create 하는 REST resource입니다.

http://host:port/<context-path>/departments

@POST annotation으로 HTTP method를 기술하고 있습니다.

```
@POST
public void registerDepartment(Department department) {
    //Create department entity in data store
    departmentService.registerDepartment(department);
}
```

## 5.2 JAX-RS 어노테이션(12/23) – HTTP 메소드 처리 - @DELETE

- ✓ 어노테이션 클래스 → @javax.ws.rs.DELETE
- ✓ REST 자원을 삭제할 때 사용합니다.

예제는 부서를 delete하는 REST resource입니다.

http://host:port/<context-path>/departments/{id}

@DELETEannotation으로 HTTP method를 기술하고 있습니다.

```
@DELETE
@Path("{id}")
public void removeDepartment(@PathParam("id") Short id) {

    //remove department entity from data store
    departmentService.removeDepartment (id);

}
```

## 5.2 JAX-RS 어노테이션(13/23) – HTTP 메소드 처리 – @HEAD, @OPTIONS

---

### ✓ 어노테이션 클래스 → @javax.ws.rs.HEAD

- REST 자원의 HTTP 헤더 정보를 가져올 때 사용합니다.
- REST 자원의 상태를 체크하거나, 미디어 타입, Content-Length, LastModified 등의 메타 정보를 확인합니다.
- 직접 구현하지 않고, JAX-RS의 프레임워크에서 기본으로 제공합니다.
- @GET 요청의 헤더를 리턴합니다.

### ✓ @javax.ws.rs.OPTIONS

- REST 자원이 제공하는 HTTP 메소드 목록을 가져올 때 사용합니다.
- 직접 구현하지 않고, JAX-RS의 프레임워크에서 기본으로 제공합니다.

## 5.2 JAX-RS 어노테이션(14/23) – 요청 패러미터 접근

---

- ✓ JAX-RS 는 Request로부터 패러미터를 추출하는 방법을 어노테이션으로 제공합니다.
- ✓ Request Parameter의 종류는 **query, uri path, form, cookie, header, matrix** 가 있습니다.
- ✓ 대부분 @GET, @POST, @PUT, @DELETE과 함께 사용합니다.

## 5.2 JAX-RS 어노테이션(15/23) – 요청 패러미터 접근 – @PathParam

- ✓ 어노테이션 클래스 → @javax.ws.rsPathParam
- ✓ 일반적으로, URI path 템플릿은 자원을 식별할 수 있는 값을 포함하고 있습니다.
- ✓ URI path에 포함되어 있는 식별 값을 Parameter로 추출하는 경우에 사용합니다.

```
//Other imports removed for brevity
javax.ws.rsPathParam
@Path("departments")
public class DepartmentService {

    @DELETE
    @Path("{id}")
    public void removeDepartment(@PathParam("id") Short deptId) {
        removeDepartmentEntity(deptId);
    }
    //Other methods removed for brevity
}
```

예제에서 ID가 10인 부서를 삭제하고자 하는 경우 아래와 같이 요청합니다.  
DELETE http://host:port/<context-path>/departments/10

## 5.2 JAX-RS 어노테이션(16/23) – 요청 패러미터 접근 – @PathParam

- ✓ URI path에 여러 Parameter를 사용하는 것도 가능합니다.

예제는 Country의 특정 City에 존재하는 전체 부서 목록을 조회하는 것을 보여줍니다.  
GET http://host:port/<context-path>/departments/{countryCode}/{cityCode}

```
@Produces(MediaType.APPLICATION_JSON)
@Path("{country}/{city}")
public List<Department> findAllDepartments(
    @PathParam("country") String countryCode,
    @PathParam("city") String cityCode) {
    //Find all departments from the data store for a country
    //and city
    List<Department> departments = findAllMatchingDepartmentEntities(countryCode, cityCode );
    return departments;
}
```

## 5.2 JAX-RS 어노테이션(17/23) – 요청 패러미터 접근 – @QueryParam

- ✓ 어노테이션 클래스 → `@javax.ws.rs.QueryParam`
- ✓ HTTP query parameter를 가져올 때 사용합니다.

예제는 조회시 부서 이름을 HTTP query parameter로 받고 있습니다.  
GET `http://host:port/<context-path>/departments?name=IT`

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public List<Department> findAllDepartmentsByName(  
    @QueryParam("name") String deptName) {  
  
    List<Department> depts= findAllMatchingDepartmentEntities(deptName);  
    return depts;  
}
```

## 5.2 JAX-RS 어노테이션(18/23) – 요청 패러미터 접근 – @MatrixParam

- ✓ 어노테이션 클래스 → @javax.ws.rs.MatrixParam
- ✓ Parameter를 정의하는 또 다른 방식으로 이름-값 쌍 형태로 semicolon(;)으로 구분하여 작성합니다.
- ✓ Matrix Parameter는 자주 사용하지는 않습니다.
- ✓ 하지만, RESTful URI 설계시 복잡한 Parameter가 필요한 경우에 사용할 수 있습니다.

예제는 조회시 부서 이름과 City를 HTTP matrix parameter로 받고 있습니다.  
GET http://host:port/<context-path>/departments;name=IT;city=Bangalore

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
@Path("matrix")  
public List<Department> findAllDepartmentsByNameWithMatrix(  
    @MatrixParam("name") String deptName,  
    @MatrixParam("city") String locationCode) {  
    List<Department> depts=findAllDepartmentsFromDB(deptName, city);  
    return depts;  
}
```

## 5.2 JAX-RS 어노테이션(19/23) – 요청 패러미터 접근 – @HeaderParam

- ✓ 어노테이션 클래스 → `@javax.ws.rs.HeaderParam`
- ✓ HTTP 헤더 정보로부터 특정한 값을 사용하고자 할 때 사용합니다.
- ✓ 애플리케이션에서 필요한 정보를 헤더에 추가하여 사용할 수 있습니다.
- ✓ 요청 헤더에는 16가지 정보(필드)가 들어 있습니다.

예제는 부서를 생성하면서 요청을 생성한 page 정보를 기록하고자 HTTP Header로 부터 referer 값을 가져오는 것을 보여줍니다.

```
@POST
public void createDepartment(
    @HeaderParam("Referer") String referer,
    Department entity) {

    logSource(referer);
    createDepartmentInDB(department);
}
```

## 5.2 JAX-RS 어노테이션(20/23) – 요청 패러미터 접근 – @CookieParam

- ✓ 어노테이션 클래스 → @javax.ws.rs.CookieParam
- ✓ Cookie 정보를 가져올 때 사용합니다.

예제는 쿠키에 저장된 정보로 default 부서를 조회하고 있습니다.

```
@GET  
@Path("cook")  
@Produces(MediaType.APPLICATION_JSON)  
public Department getDefaultDepartment(@CookieParam("Default-Dept") short departmentId) {  
    Department dept=findDepartmentById(departmentId);  
    return dept;  
}
```

## 5.2 JAX-RS 어노테이션(21/23) – 요청 패러미터 접근 – @FormParam

- ✓ 어노테이션 클래스 → @javax.ws.rs.FormParam
- ✓ HTML form을 Request body에 담아 요청할 때 form 패러미터에 접근합니다.
- ✓ content type(media type)은 application/x-www-form-urlencoded

```
<!DOCTYPE html>
<html>
<head>
<title>Create Department</title>
</head>
<body>
<form method="POST"
action="/resources/departments">
    Department Id:
    <input type="text" name="departmentId">
    <br>
    Department Name:
    <input type="text" name="departmentName">
    <br>
    <input type="submit" value="Add Department" />
</form>
</body>
</html>
```

HTML

REST resource

예제는 HTML form으로 부터 submit된  
파라미터를 받아서 부서를 생성하고 있습니다.

```
@Path("departments")
public class DepartmentService {
    @POST
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void createDepartment(
        @FormParam("departmentId") short departmentId,
        @FormParam("departmentName") String departmentName) {
        createDepartmentEntity(departmentId, departmentName);
    }
}
```

## 5.2 JAX-RS 어노테이션(22/23) – 요청 패러미터 접근 – @DefaultValue

- ✓ 어노테이션 클래스 → `@javax.ws.rs.DefaultValue`
- ✓ Request 패러미터의 default값을 설정할 때 사용합니다.
- ✓ `@PathParam`, `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@FormParam`, `@HeaderParam`과 함께 사용합니다. 매칭되는 Parameter가 없는 경우의 default값을 지정합니다.

예제는 parameter의 default 값을 설정하는 방법을 보여줍니다.

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public List<Department> findAllDepartmentsInRange(  
    @DefaultValue("0") @QueryParam("from") Integer from,  
    @DefaultValue("100") @QueryParam("to") Integer to) {  
    findAllDepartmentEntitiesInRange(from, to);  
}
```

## 5.2 JAX-RS 어노테이션(23/23) – 요청 패러미터 접근 – @BeanParam

- ✓ 어노테이션 클래스 → `@javax.ws.rs.BeanParam`
- ✓ 패러미터를 객체로 매칭할 때 사용합니다.
- ✓ `@PathParam`, `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@FormParam`, `@HeaderParam` 등과 함께 사용합니다.

Bean

```
public class DepartmentBean {  
    @FormParam("departmentId")  
    private short departmentId;  
  
    @FormParam("departmentName")  
    private String departmentName;  
  
    //getter and setter for the above fields  
    //are not shown here to save space  
}
```

REST resource

```
@POST  
public void createDepartment(  
    @BeanParam DepartmentBean deptBean){  
  
    createDepartmentEntity(deptBean.getDepartmentId(),  
        deptBean.getDepartmentName());  
}
```

예제는 요청 parameter를 bean(객체)로 받는 방법을 보여줍니다.

## 5.3 추가 메타데이터 리턴

- ✓ javax.ws.rs.core.Response 객체는 HTTP body와, HTTP header의 wrapper 클래스입니다.
- ✓ HTTP Response에 추가 META 정보를 담기 위해 Response 객체를 사용합니다.

```
//Other imports removed for brevity
import javax.ws.rs.core.CacheControl;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;

@GET
@Produces(MediaType.APPLICATION_JSON)
public Response findAllDepartmentsByName(@QueryParam("name") String deptName) {
    List<Department> depts= findAllMatchingDepartmentEntities(deptName);
    //Sets cache control directive to the response
    CacheControl cacheControl = new CacheControl();
    //Cache the result for a day
    cacheControl.setMaxAge(86400);
    return Response.ok().cacheControl(cacheControl).entity(depts).build();
}
```

Response.ok() : HTTP 200  
.cacheControl(cacheControl) : Cache 보관 기간  
.entity(depts) : 응답 body

Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	Cache-Control: max-age=3600	Permanent
---------------	--	-----------------------------	-----------

## 5.4 RESTful 웹 서비스 만들기 [실습]

---

- ✓ 실습 자료를 별도로 설명 후 안내를 합니다.
- ✓ 실습 가이드를 참조합니다.

RESTful 서비스 예제 시스템 설명과 시연

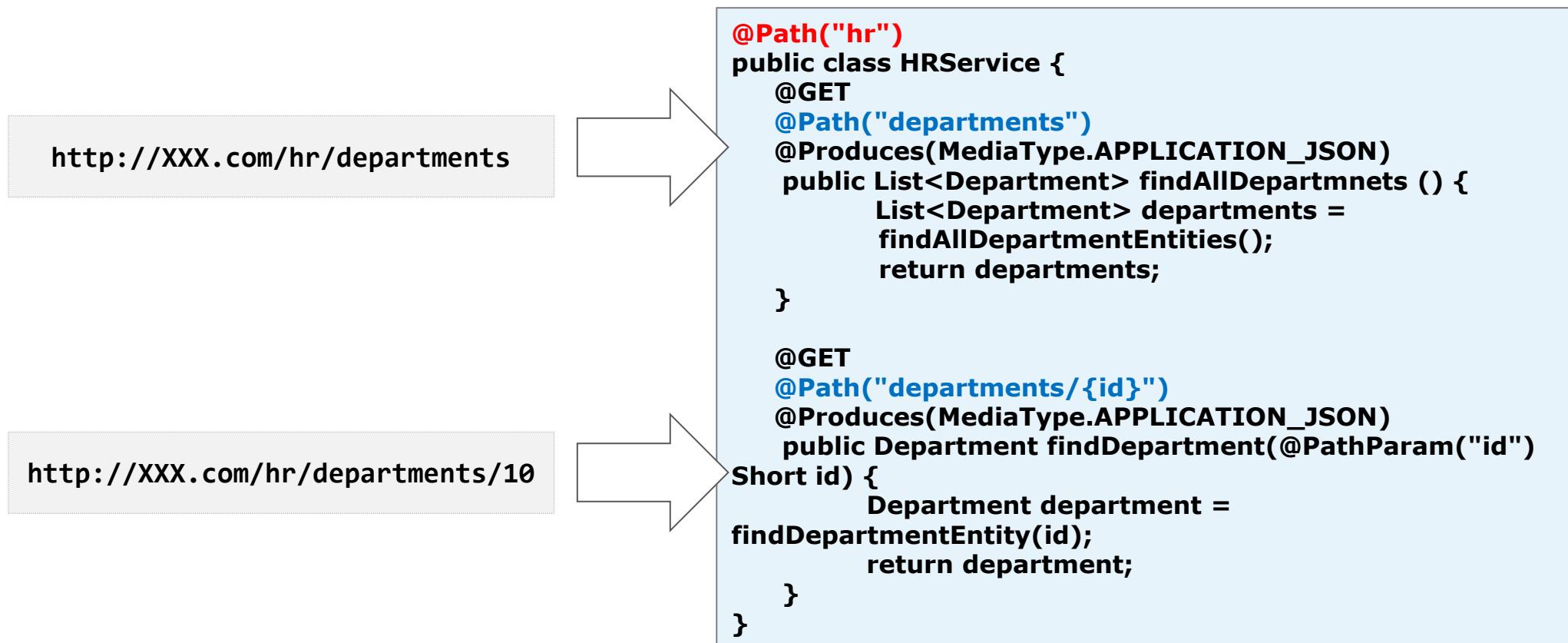


## 6. JAX-RS API 고급 기능

- 6.1 JAX-RS 서브 자원과 서브 자원 탐지기
- 6.2 예외 처리 방법
- 6.3 JAX-RS 데이터 검증
- 6.4 요청/응답 메시지 확장
- 6.5 비동기 RESTful
- 6.6 RESTful 서비스의 HTTP 캐시 관리
- 6.7 Filter와 Interceptor
- 6.8 JAX-RS 라이프 사이클
- 6.9 RESTful 고급 기능 (실습)

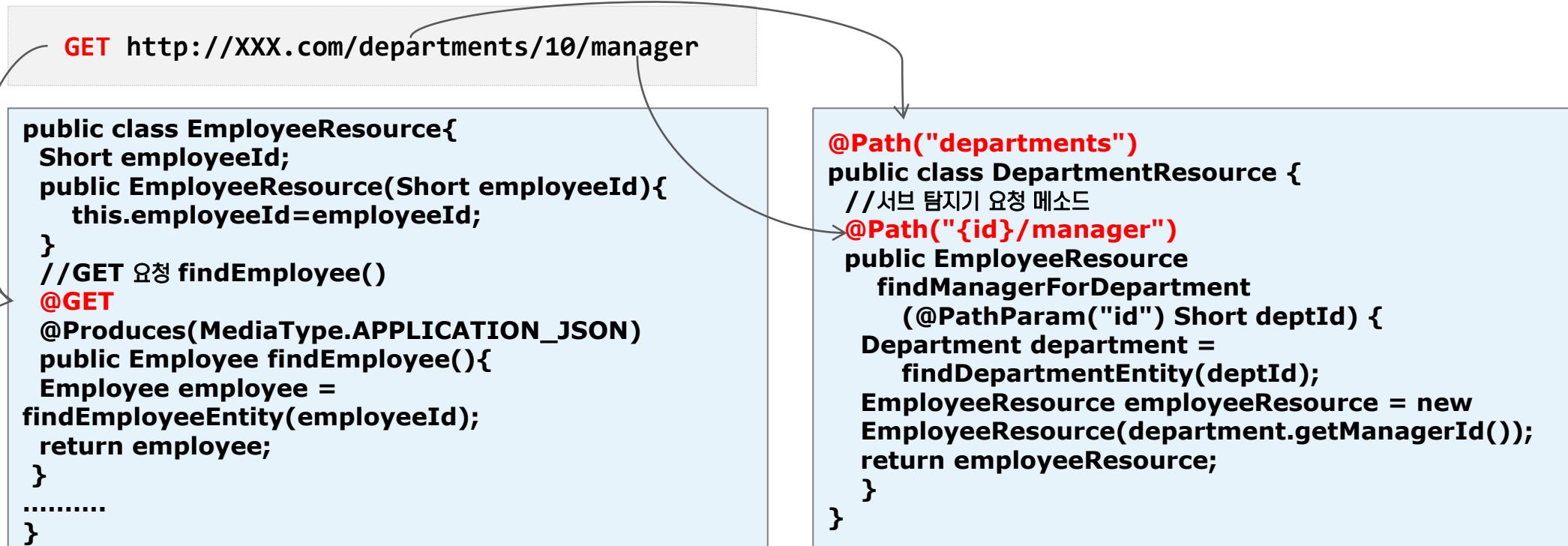
## 6.1 JAX-RS 서브자원과 서브자원 탐지기 (1/2)

- ✓ RESTful 서비스는 식별 가능한 고유 이름을 가지고 있으며 @Path 어노테이션으로 이름을 부여 합니다.
- ✓ @Path는 클래스와 서비스 처리를 위한 메소드에 사용할 수 있습니다.
- ✓ 클래스에 부여된 @Path 은 루트 자원, 메소드에 부여된 것은 서브자원(subresource)이라 합니다.
  - 서브 자원은 루트 자원에 종속됩니다.



## 6.1 JAX-RS 서브자원과 서브자원 탐지기 (1/2)

- ✓ 동일한 URI라도 @GET, @POST, @PUT과 같은 HTTP요청에 따라 다른 응답을 보낼 수 있습니다.
- ✓ HTTP요청에 따라 다른 응답을 보내는 서브 자원을 서브 탐지기라 합니다.
- ✓ 서브 탐지기는 어플리케이션을 보다 유연하게 만들어 줍니다.



## 6.2 예외 처리 방법 (1/4)

---

- ✓ 실용주의 REST는 에러가 발생 시 사용자에게 에러 정보를 제공해야 합니다.
- ✓ JAX-RS에서는 에러 정보를 ResponseBuilder와 WebApplicationException을 통해 제공합니다.
- ✓ 에러 정보는 HTTP 상태 뿐만 아니라 사용자가 식별할 수 있는 에러 정보도 포함되어야 합니다.

## 6.2 예외 처리 방법 (2/4) –ResponseBuilder

- ✓ ResponseBuilder는 HTTP 상태 코드를 활용하여 예외 정보를 제공합니다.
- ✓ 구현상 쉽게 예외 정보를 제공할 수 있지만 현실의 다양한 상황을 처리하기에는 미흡합니다.
  - 소스코드 내 Exception 발생 시 처리 미흡 -> 예외 발생 가능한 곳에서 프로그램적 처리 필요
  - 리턴 객체 필수 사용

예제는 ResponseBuilder를 이용한 예외 처리 예시입니다.

```
@DELETE
@Path("departments/{id}")
public Response remove(@PathParam("id") Short id) {
    Department department = entityManager.find(Department.class,id);
    if(department == null){
        //Department to be removed is not found in data store
        return Response.status(Response.Status.NOT_FOUND).entity
            ("Entity not found for : " + id).build();
    }
    entityManager.remove(entityManager.merge(department));
    return Response.status(Response.Status.OK).build();
}
```

## 6.2 예외 처리 방법 (3/4) –WebApplicationException

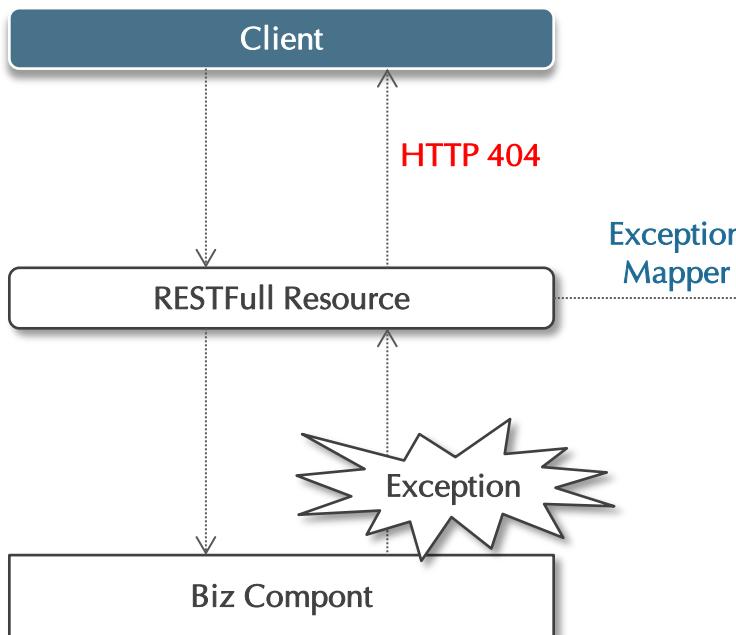
- ✓ WebApplicationException은 HTTP 상태 코드뿐만 아니라 다양한 에러 메시지 전달이 가능합니다.
- ✓ 서비스 중 예상치 못한 에러 발생 시에도 대응을 할 수 있습니다.
  - 서비스 중 발생되는 모든 Exception에 대해 throw를 통해 처리합니다.
  - WebApplicationException는 RuntimeException을 상속받아 서비스 중 발생하는 예외에 대응합니다.

예제는 WebApplicationException 처리 예시입니다.

```
@DELETE  
@Path("departments/{id}")  
public void removeDepartment(@PathParam("id") Short id) {  
    Department department = findDepartmentEntity(id);  
    //throw exception if department to be deleted is not found  
    if(department == null){  
        throw new  
            WebApplicationException(Response.Status.NOT_FOUND);  
    }  
    removeDepartmentEntity(department);  
}
```

## 6.2 예외 처리 방법 (4/4) –Exception Mapper Class

- ✓ RESTful API는 다른 컴포넌트를 통해 제공 받아 외부로 서비스를 제공하는 통로입니다.
- ✓ 때문에 다른 컴포넌트에서 발생한 예외에 대해서도 대응하여 고객에게 예외 정보를 제공 할 수 있어야 합니다.
- ✓ JAX-RS는 Exception Mapper Class를 통해 다양한 예외에도 대응할 수 있습니다.



예제는 서비스 처리 시 업무 예외가 발생한 경우 해당 예외를 처리하는 예시입니다.

```
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class DepartmentNotFoundExceptionMapper implements
    ExceptionMapper<DepartmentNotFoundException> {

    @Override
    public Response toResponse(DepartmentNotFoundException
        exception) {
        return Response.status(Response.Status.NOT_FOUND).
            entity(exception.getMessage()).build();
    }
}
```

## 6.3 JAX-RS 데이터 검증 (1/5)

- ✓ JAX-RS는 사용자가 요청한 데이터에 대한 검증 기능을 제공합니다.
- ✓ 입력데이터의 Null, Size, Type, 등에 대한 검증 기능을 어노테이션을 통해 사용합니다.
- ✓ 비즈니스 특성에 따라 추가적인 검증이 필요한 경우는 검증 모듈을 직접 구현해야 합니다.

어노테이션	JAVA Type	검증 설명
@NotNull	All JAVA Type	Must Not Null
@Null	All JAVA Type	Must Null
@AssertFalse	Java.lang.Boolean, Boolean	Must false
@AssertTrue	Java.lang.Boolean, Boolean	Must true
@DecimalMax	java.math.BigDecimal java.math.BigInteger java.lang.String byte, short, int, long, and their wrapper types	최대 사이즈
@DecimalMin	java.math.BigDecimal java.math.BigInteger java.lang.String byte, short, int, long, and their wrapper types	최소,동일 사이즈

## 6.3 JAX-RS 데이터 검증 (2/5)

### ✓ JAX-RS 제공 데이터 검증 유형(계속)

어노테이션	JAVA Type	검증 설명
@Digits	java.math.BigDecimal java.math.BigInteger java.lang.String byte, short, int, long, and their wrapper types	숫자 타입만 허용
@Max	java.math.BigDecimal java.math.BigInteger java.lang.String byte, short, int, long, and their wrapper types	최대 값(숫자)
@Min	java.math.BigDecimal java.math.BigInteger java.lang.String byte, short, int, long, and their wrapper types	최소 값(숫자)

## 6.3 JAX-RS 데이터 검증 (3/5)

### ✓ JAX-RS 제공 데이터 검증 유형(계속)

어노테이션	JAVA Type	검증 설명
@Future	java.util.Date java.util.Calendar	현재 기준 미래 일자
@Past	java.util.Date java.util.Calendar	현재 기준 과거 일자
@Pattern	Java.lang.String	정규식
@Size	java.lang.String (string length is evaluated) jav.util.Collection (collection size is evaluated) java.util.Map (map size is evaluated) java.lang.Array (array length is evaluated)	길이

## 6.3 JAX-RS 데이터 검증 (4/5)

### ✓ JAX-RS 제공 데이터 검증 유형 사용 예

- 검증에 실패할 경우 예외가 발생되며 예외 메시지는 정의된 메시지로 제공됩니다. -> HTTP 상태 코드 : 500

예제는 서비스 요청 시 데이터 검증 예시로 최소 입력 값에 대한 검증을 수행합니다.

```
@GET  
@Path("departments/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public Department findDepartment(@PathParam("id")  
    @Min(value = 0, message = "Department Id must be a positive value")  
    Short id, @Context Request request) {  
    Department department = findDepartmentEntity(id);  
    return department;  
}
```

## 6.3 JAX-RS 데이터 검증 (5/5)

- ✓ JAX-RS는 제공되는 데이터 검증 외 사용자 정의 검증도 지원합니다.
- ✓ 사용자 정의 검증을 사용하기 위해 검증기능, 사용자 정의 검증 어노테이션, 메시지 정의가 필요합니다.

```
@POST  
@Path("departments")  
@Consumes(MediaType.APPLICATION_JSON)  
public void create(@ValidDepartment Department entity) {  
    createDepartmentEntity(entity);  
}
```

RESTFull Resource

```
@Constraint(validatedBy = {ValidDepartmentValidator.class})  
@Target({ElementType.FIELD, ElementType.PARAMETER})  
@Retention(value = RetentionPolicy.RUNTIME)  
public @interface ValidDepartment {  
    //검증메시지 Key  
    String message() default  
        "{com.packtpub.rest.validation.deprule}";  
    Class<?>[] groups() default {};  
    Class<? extends Payload>[] payload() default {};
```

검증 어노테이션

```
com.packtpub.rest.validation.deprule=Department  
already exist
```

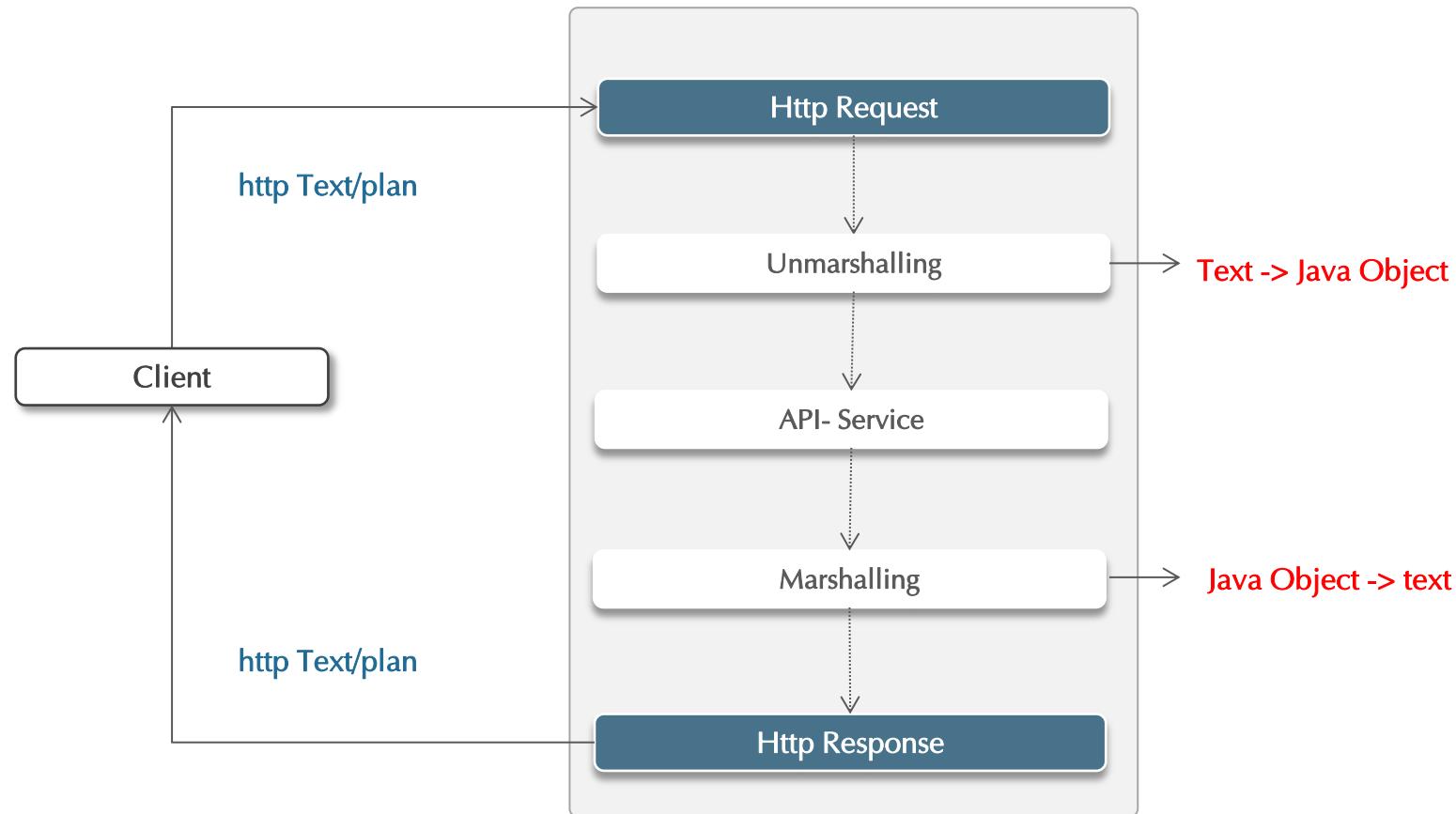
메시지 프로퍼티

```
public class ValidDepartmentValidator implements  
ConstraintValidator<ValidDepartment, Department> {  
    @Override  
    public void initialize(ValidDepartment  
constraintAnnotation) {  
    }  
    @Override  
    public boolean isValid(Department department,  
        ConstraintValidatorContext context) {  
        //검증 기능 구현  
        if(isDeptExistsForLoc(department.getDepartmentId(),  
            department.getDepartmentName(),  
            department.getLocationId())) {  
            return false;  
        }  
        return true;  
    }  
    ....  
}
```

검증 기능

## 6.4 요청/응답 메시지 확장 [1/5]

- ✓ RESTfull은 서비스 요청/응답 시 HTTP 데이터와 Java 객체간 변환 기능이 필요합니다. → Marshall/UnMarshall



## 6.4 요청/응답 메시지 확장 [2/5]

- ✓ JAX-RS는 기본적인 Http 데이터 유형과 Java 타입간 변환 매핑을 제공합니다.

Java 데이터 타입	Internet media type
byte[]	/*
java.lang.String	/*
java.io.Reader	/*
java.io.File	/*
javax.activation.DataSource	/*
javax.ws.rs.core.StreamingOutput	/*
All primitive types	text/plain
java.lang.Boolean	text/plain
java.lang.Character	text/plain
java.lang.Number	text/plain
javax.xml.transform.Source	text/xml, application/xml, and application/*+xml
javax.xml.bind.JAXBElement	text/xml, application/xml, and application/*+xml
Application supplied JAXB annotated objects (types annotated with @XmlRootElement or @XmlType)	text/xml, application/xml, and application/*+xml
javax.ws.rs.core.MultivaluedMap<String, String>	text/xml, application/xml, and application/*+xml

## 6.4 요청/응답 메시지 확장 [3/5]

- ✓ 기본 매핑외 추가 타입에 대한 매핑이 필요한 경우를 위해 확장 기능의 구현을 통해 사용할 수 있습니다.
- ✓ 요청과 응답에 대해 각각 별도의 확장 기능을 구현할 수 있습니다.
- ✓ 확장된 클래스에 @Provider추가 , 또는 JAX-RS 환경 설정에 추가하여 사용 합니다.- > 미 사용시 사용불가

구분	확장 클래스	주요 기능	설명
응답	javax.ws.rs.ext.MessageBodyWriter<T>	isWriteable()	Java 타입을 http 타입으로 변환 여부 체크
		isWriteable()	JAX-RS 2.0에서 삭제
		writeTo()	Java 타입을 Http Message로 변환
요청	javax.ws.rs.ext.MessageBodyReader<T>	isReadable()	Java 타입 변환여부 체크
		readFrom()	Input Stream을 통한 데이터 Read

## 6.4 요청/응답 메시지 확장 [4/5] – 서비스 응답

- ✓ 응답 시 javax.ws.rs.ext.MessageBodyWriter 인터페이스 구현을 통해 확장할 수 있습니다.
- ✓ 확장된 클래스에 @Provider 추가, 또는 JAX-RS 환경 설정에 추가하여 사용합니다. -> 미 사용시 사용불가
- ✓ @Produces에 Http 타입을 설정합니다.

```
.....  
import org.supercsv.prefs.CsvPreference;  
@Provider  
@Produces("application/csv")  
public class CSVMessageBodyWriter  
implements  
MessageBodyWriter<List<Department>> {  
....
```

예제는 응답 시 HTTP 헤더에 HTTP 미디어 타입을 설정합니다.

:  
GET /api/hr/departments HTTP/1.1  
Host: localhost:8080  
Accept: application/csv  
departmentId,departmentName,locationId,ma  
nagerId  
1001,"Finance",1700,101  
1002,"Office Administration",1500,205

예제는 응답 결과에 대한 HTTP 헤더 정보입니다.

## 6.4 요청/응답 메시지 확장 [5/5] – 서비스 요청

- ✓ 요청 시 javax.ws.rs.ext.MessageBodyReader interface의 구현을 통해 확장할 수 있습니다.
- ✓ 확장된 클래스에 @Provider 추가, 또는 JAX-RS 환경 설정에 추가하여 사용 → 미 사용시 사용불가
- ✓ @Comsumes 에 Http 타입 설정 합니다.

```
.....  
@Provider  
@Consumes("application/csv")  
public class CSVMessageBodyReader  
implements MessageBodyReader<List<D  
epartment>> {
```

예제는 요청 시 서비스가 전달 받을 HTTP 미디어 타입을 정의 합니다.

```
POST /api/hr/departments HTTP/1.1  
Host: localhost:8080  
Content-Type: application/csv  
departmentId,departmentName,locationId,ma  
nagerId  
1001,"Finance",1700,101  
1002,"Office Administration",1500,205
```

예제는 서비스 요청 시 HTTP헤더에 요청 미디어 타입을 설정합니다.

## 6.5 비동기 RESTful (1/3)

- ✓ RESTful 서비스는 요청 후 응답처리 까지 하나의 스레드가 필요합니다.
  - 서비스 처리 중인 스레드는 다른 서비스를 제공할 수 없습니다.
  - 대부분 RESTful 서비스의 스레드는 그 수가 한정되어 있습니다.
- ✓ 요청/응답 시간이 긴 서비스는 서비스 자원을 독점하여 서비스 성능에 영향을 줄 수 있습니다.
- ✓ 비동기 RESTful은 응답시간이 긴 서비스에 대해 자원 독점을 방지 할 수 있는 기능을 제공합니다.

```
@GET  
@Path("departments")  
@Produces(MediaType.APPLICATION_JSON)  
public void findAllDepartments(@Suspended AsyncResponse asyncResponse)
```

## 6.5 비동기 RESTful (2/3)

- ✓ @Suspended로 생성된 AsyncResponse 를 이용하여 비동기 RESTful을 구현합니다.

```
@Path("hr/async")
public class HRAsynchService {
    private final ExecutorService executorService = Executors.newCachedThreadPool();

    @GET
    @Path("departments")
    @Produces(MediaType.APPLICATION_JSON)
    public void findAllDepartments(@Suspended final AsyncResponse asyncResponse) {
        //Set time out for the request
        asyncResponse.setTimeout(10, TimeUnit.SECONDS);
        Runnable longRunningDeptQuery = new Runnable(){
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("HPPersistenceUnit");
            EntityManager entityManagerLocal = emf.createEntityManager();

            public void run() {
                CriteriaQuery cq = entityManagerLocal.getCriteriaBuilder().createQuery();
                cq.select(cq.from(Department.class));
                List<Department> depts = entityManagerLocal.createQuery(cq).getResultList();
                GenericEntity<List<Department>> entity = new GenericEntity<List<Department>>(depts)
            };
            asyncResponse.resume(Response.ok().entity(entity).build());
        };
        executorService.execute(longRunningDeptQuery);
    }
}
```

예제는 @Suspended를 이용한 서버영역의 비동기 호출 예시입니다.

## 6.5 비동기 RESTful (3/3)

- ✓ 비동기 RESTful 서비스는 Client영역도 비동기 호출에 대한 구현이 필요합니다.
- ✓ javax.ws.rs.client.AsyncInvoker를 이용하여 비동기 호출을 합니다.
- ✓ 비동기 호출을 위해 서버로 부터 결과 이벤트를 전달 받을 객체도 사용가능 합니다. → callback객체
  - Callback객체는 정상응답 처리(completed)와 예외 발생 시 처리(failed)가 가능합니다.

```
String BASE_URI =
"http://localhost:8080/hr-services/webresources";
Client client = ClientBuilder.newClient();
WebTarget webTarget =
client.target(BASE_URI).path("hr").path("departments");
AsyncInvoker asyncInvoker = webTarget.request(APPLICATION_JSON).async();
Future<Response> responseFuture = asyncInvoker.get(
    new InvocationCallback<List<Department>>() { // 콜백객체 등록
        @Override
        public void completed(List<Department> response) {
            //정상 응답 시 처리
            client.close();
        }
        @Override
        public void failed(Throwable throwable) {
            // 예외 발생 시 처리
            log(throwable);
        }
    });
});
```

예제는 클라이언트 영역에서 비동기 사용에 대한 예시입니다..

## 6.6 RESTful 서비스의 HTTP 캐시 관리 (1/6)

---

- ✓ 웹 기반 시스템은 항상 처리 속도에 대한 향상을 염두에 두어야 합니다.
- ✓ HTTP/1.1 프로토콜은 캐시를 통해 처리 속도 향상을 위한 명세를 제공합니다.
- ✓ JAX-RS API는 HTTP캐시 명세를 활용하여 서비스 속도를 향상 시킬 수 있는 기능을 제공합니다.

## 6.6 RESTful 서비스의 HTTP 캐시 관리 [2/6] –Expires header

- ✓ HTTP 헤더의 Expires 기능을 이용하여 데이터를 캐싱합니다.
- ✓ 제공되는 데이터는 유효일자(Expires) 까지 유효함을 보장해야 합니다.
- ✓ 공휴일 정보 등과 같이 변경 불가한 정보에 주로 사용합니다.

```
@GET
```

```
@Path("departments/{id}/holidays")
@Produces(MediaType.APPLICATION_JSON)
```

```
public Response getHolidayListForCurrentYear(@PathParam("id") Short
deptId) {
```

```
List<Date> holidayList = getHolidayListForDepartment(deptId);
Response.ResponseBuilder response = Response.ok(holidayList).
type(MediaType.APPLICATION_JSON);
int currentYear = getCurrentYear();
Calendar expirationDate = new GregorianCalendar
(currentYear,12, 31);
response.expires(expirationDate.getTime());
return response.build();
}
```

예제는 http Header에 데이터 유효일을 설정합니다.

Server: GlassFish Server Open Source Edition 4.1

Expires: Sat, 31 Dec 2015 00:00:00 GMT

Content-Type: application/json

Date: Mon, 02 Mar 2015 05:24:58 GMT

Content-Length: 20

Http Header에 데이터 유효일이 포함됩니다.

## 6.6 RESTful 서비스의 HTTP 캐시 관리 (3/6) –Cache–Control

- ✓ Expires보다 유연하게 캐시의 관리가 가능합니다.
- ✓ Cache-Control 항목을 통해 캐시의 상태를 관리 합니다.

```
@GET  
@Path("departments/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public Response findDepartmentById(@PathParam("id") Short deptId)  
{  
  
    Department department = findDepartmentEntityById(deptId);  
    //캐시 정보를 설정하기 위한 객체 생성  
    CacheControl cc = new CacheControl();  
    //캐시 유효기간 설정 (86400 sec)  
    cc.setMaxAge(86400);  
    cc.setPrivate(true);  
    ResponseBuilder builder = Response.ok(myBook);  
    //응답에 캐시 정보 설정  
    builder.cacheControl(cc);  
    return builder.build();  
}
```

예제는 http Header에 캐시정보를 설정합니다.

Server: GlassFish Server Open Source Edition 4.1  
**Cache-Control: private, no-transform, max-age=86400**  
Content-Type: application/json  
Date: Mon, 02 Mar 2015 05:56:29 GMT  
Content-Length: 82

Http Header에 캐시 정보가 포함됩니다.

## 6.6 RESTful 서비스의 HTTP 캐시 관리 (4/6) –Last-Modified

- ✓ HTTP 헤더의 Last-Midified 항목을 이용해 캐시 정보를 관리 합니다.
- ✓ HTTP 헤더의 If-Modified-Since 또는 If-Unmodified-Since 항목과 같이 사용될 수 있습니다.
- ✓ 사용자가 If-Modified-Since항목에 특정일자 입력 시 해당일 이후 변경 여부를 판단하여 응답이 가능합니다.
  - If-Modified-Since의 일자 이후 변경 되지 않은 경우 HTTP 304 상태코드 리턴

```
@GET  
@Path("departments/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public Response findDepartmentWithCacheValidationWithLastModifiedDate(  
    @PathParam("id") Short id, @Context Request request) {  
  
    //해당 객체 조회  
    Department department = entityManager.find(Department.class,id);  
    //최종 변경일 조회  
    Date latModifiedDate = department.getModifiedDate();  
    // If-Modified-Since항목의 데이터와 최종 변경일 비교  
    // javax.ws.rs.core.Request.evaluatePreconditions()를 이용한 비교  
    // If-Modified-Since일자 이후 변경된 경우 라면 null 리턴 미 변경된 경우 304 리턴  
    ResponseBuilder builder = request.evaluatePreconditions(latModifiedDate);  
    //cached resource did change; send new one  
    if (builder == null) {  
        builder = Response.ok(department);  
        builder.lastModified(latModifiedDate);  
    }  
    return builder.build();  
}
```

예제는 http Header의 Last-modified를 이용해 캐시정보를 확인 합니다.

## 6.6 RESTful 서비스의 HTTP 캐시 관리 (5/6) – ETag

- ✓ HTTP 헤더의 ETag 항목을 이용해 서버 정보와 클라이언트 정보를 비교하여 캐시를 관리 합니다.
- ✓ 서버에서 tag정보 생성하여 사용자에게 전송하고 사용자는 이 값을 다시 전송하여 캐시여부를 판단하기 때문에 보다 정밀한 캐시관리가 가능합니다.

```
@GET  
@Path("departments/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
  
public Response findDepartmentWithCacheValidation(@PathParam("id") Long deptId,  
@Context Request request) {  
    //데이터 조회 및 현재 태그 조회  
    Department department = findDepartmentEntity(deptId);  
    EntityTag etag = new EntityTag(Integer.toString(department.hashCode()));  
    //사용자가 요청 시 전송한 태그값과 비교. 데이터가 변경된 경우 데이터를 전송한다.  
    ResponseBuilder builder = request.evaluatePreconditions(etag);  
    if (builder == null) {  
        builder = Response.ok(department);  
        builder.tag(etag);  
    }  
    return builder.build();  
}
```

예제는 http Header의 Etag을 이용해 캐시정보를 설정합니다.

Server: GlassFish Server Open Source Edition 4.1  
ETag: "03cb35ca667706c68c0aad4cb04c7a211"  
Content-Type: application/json  
Date: Mon, 02 Mar 2015 05:30:11 GMT  
Content-Length: 82

Http Header에 캐시 정보가 포함됩니다.

## 6.6 RESTful 서비스의 HTTP 캐시 관리 [6/6] –data update

- ✓ POST 또는 PUT 처리 시 Last-Modified, Etag를 이용하여 데이터 저장 및 변경 여부를 판단하여 데이터 베이스 사용등과 같은 리소스 사용을 줄일 수 있습니다.

```
@PUT
@Path("etag/departments/{id}")
@Consumes(MediaType.APPLICATION_JSON)
public Response edit(@PathParam("id") Short deptId, @Context Request request, Department entity)
{
    //데이터 조회
    Department detEntityInDB = finDepartmentEntity(deptId);
    //Etag 조회
    EntityTag etag = new EntityTag(Integer.toString(detEntityInDB.hashCode()));
    //데이터 변경 여부 검증
    Response.ResponseBuilder builder =
request.evaluatePreconditions(detEntityInDB.getModifiedDate(), etag);
    // 데이터 미 변경시 Exception 발생 http 상태 코드 412
    if (builder != null) {
        return builder.status(
            Response.Status.PRECONDITION_FAILED).build();
    }
    updateDepartmentEntity(entity);
    EntityTag newEtag = new EntityTag(Integer.toString(entity.hashCode()));
    builder = Response.noContent();
    builder.lastModified(entity.getModifiedDate());
    builder.tag(newEtag);
    return builder.build();
}
```

예시는 Http Header의 캐시 정보를 이용하여 데이터의 업데이트 여부를 결정합니다.

## 6.7 Filter와 Interceptor (1/11)

---

- ✓ RESTful 서비스 중 인증, encoding, 사용자 정보 캐싱과 같은 공통 기능들이 존재 합니다.
- ✓ 각 기능을 서비스 별로 구현 하기 보다 JAX-RS의 filter와 interceptor를 이용해 쉽게 구현할 수 있습니다.
- ✓ filter와 interceptor는 유사한 기능이지만 일반적으로 filter는 Http Header, interceptor는 메시지 관련된 기능에 사용합니다.
- ✓ JAX-RS API는 클라이언트와 서버 모두 filter 기능을 제공을 하고 있습니다.

## 6.7 Filter와 Interceptor (2/11)

### ✓ 서버 영역 filter – 서비스 요청

- 요청에 대한 filter는 Postmatching, Prematching 영역으로 구성됩니다.
- Prematching 필터는 서비스 요청 직 후 바로 수행되며 Postmatching 필터는 서비스 요청 직 후 서비스에 해당하는 java resoce(메소드)가 매핑 된 이후 수행됩니다.(메소드는 수행 전)

구분	호출 시점	설명
Prematching	서비스 요청 접수 직 후	@PreMatching 선언 필요 서비스 요청 정보(Header) 변경 가능
Postmatching	서비스 요청 접수 수 서비스에 해당 하는 리소스(메소드) 매핑 직후(메소드 호출 전)	@PostMatching 선언 불필요 서비스 요청 정보(Header) 불가

## 6.7 Filter와 Interceptor (3/11) –서버 영역 filter–Prematching

✓ Prematching필터는 서비스 요청 직 후 바로 수행됩니다.

- 서비스에 필요한 리소스 매핑 전 이기 때문에 Http 정보 변경 가능 (POST→PUT)

```
import javax.ws.rs.container.PreMatching;  
  
@Provider  
@PreMatching  
public class JAXRSContainerPrematchingRequestFilter implements  
ContainerRequestFilter {  
  
    @Override  
    public void filter(ContainerRequestContext requestContext) throws IOException {  
        //Modify the method type from POST to PUT  
        if (requestContext.getMethod().equals("POST")) {  
            requestContext.setMethod("PUT");  
        }  
    }  
}
```

예제는 Http 헤더의 메소드 정보를 변경합니다.

## 6.7 Filter와 Interceptor (4/11) –서버 영역 filter –Postmatching

- ✓ Postmatcing필터는 Prematcing필터 이후에 수행됩니다.
- ✓ 요청한 서비스 URI에 대응되는 서비스가 로드된 이후 호출 됩니다.
- ✓ 호출될 서비스가 매핑된 이후 이기 때문에 HTTP 관련 정보를 변경할 수 없습니다.

```
import javax.ws.rs.ext.Provider;
```

예제는 권한정보를 확인 하는 필터입니다.

```
@Provider
public class AuthorizationRequestFilter implements ContainerRequestFilter {

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        //Get the URI for current request
        UriInfo uriInfo = requestContext.getUriInfo();
        String uri = uriInfo.getRequestUri().toString();
        int index = uri.indexOf("/config/");
        boolean isSettingsService = (index != -1);
        if (isSettingsService) {
            SecurityContext securityContext = requestContext.getSecurityContext();
            if (securityContext == null || !securityContext.isUserInRole("ADMIN")) {
                requestContext.abortWith(Response.status(Response.Status.UNAUTHORIZED).entity("Unauthorized
access.").build());
            }
        }
    }
}
```

## 6.7 Filter와 Interceptor (5/11) – 서버영역 – 응답필터

- ✓ 서비스 응답 필터는 REST API 호출 후 수행됩니다.
- ✓ 일반적으로 응답 Header 정보에 대한 설정관련 작업을 수행합니다.

```
import javax.ws.rs.container.ContainerResponseFilter;
@Provider
public class CORSFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
    ContainerResponseContext cres) throws IOException {
        //Specify CORS headers: * represents allow all values
        cres.getHeaders().add("Access-Control-Allow-Origin", "*");
        cres.getHeaders().add("Access-Control-Allow-Headers", "*");
        cres.getHeaders().add("Access-Control-Allow-Credentials", "true");
        cres.getHeaders().add("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS, HEAD");
        cres.getHeaders().add("Access-Control-Max-Age", "1209600");
    }
}
```

예제는 서비스 이후 응답 시 Header 정보에 필요한 값을 설정하는 필터입니다.

## 6.7 Filter와 Interceptor (6/11) -클라이언트 영역 -서비스 요청 필터

- ✓ 서비스 요청 시 클라이언트에서 filter 사용은 필수 입력 값 설정 여부 등 공통 기능에 사용할 수 있습니다.
- ✓ 서버와 연결 시 Header 정보 값 설정 등에 사용합니다.

예제는 사용자측에서 서비스 호출 시 Header에 특정 값이 없을 때의 예시입니다.

```
import javax.ws.rs.client.ClientRequestFilter;

@Provider
public class JAXRSClientRequestFilter implements ClientRequestFilter {

    @Override
    public void filter(ClientRequestContext requestContext) throws IOException {
        if(requestContext.getHeaders().get("Client-Application") == null) {
            requestContext.abortWith(Response.status(Response.Status.BAD_REQUEST)
                .entity("Client-Application header is required.").build());
        }
    }
}
```

## 6.7 Filter와 Interceptor (7/11) -클라이언트 영역 -서비스 응답

- ✓ 서비스 응답 시 클라이언트에서 filter 사용은 응답 값(HTTP 상태 코드) 정상여부 판단에 주로 사용됩니다.

```
import javax.ws.rs.client.ClientResponseFilter;

public class JAXRSClientResponseFilter implements ClientResponseFilter {

    @Override
    public void filter(ClientRequestContext reqContext, ClientResponseContext respContext) throws
    IOException {

        if (respContext.getStatus() == 200) {
            return;
        }else{
            logError(respContext);
        }
    }

    private void logError(ClientResponseContext respContext) {
        //에러 처리
    }
}
```

예제는 서비스 응답 시 HTTP 상태 코드 값을 확인하여 에러 여부를 판단하는 예제입니다.

## 6.7 Filter와 Interceptor (8/11) –서버 영역 interceptor– 서비스 요청

- ✓ javax.ws.rs.ext.ReaderInterceptor를 구현하여 사용합니다.
- ✓ interceptor는 메시지 관련된 정보 변환 등에 주로 사용합니다. → 한글 인코딩, 암호화 등

```
import javax.ws.rs.ext.ReaderInterceptor;

@Provider
public class JAXRSReaderInterceptor implements ReaderInterceptor {

    @Override
    public Object aroundReadFrom(ReaderInterceptorContext context) throws IOException,
    WebApplicationException {
        List<String> header = context.getHeaders().get("Content-Encoding");

        // decompress gzip stream only
        if (header != null && header.contains("gzip")) {
            InputStream originalInputStream = context.getInputStream();
            context.setInputStream(new GZIPInputStream(originalInputStream));
        }
        return context.proceed();
    }
}
```

예제는 서비스 요청 시 Header의 media type에 따라 GZIP inputStream을 사용하는 예제입니다.

## 6.7 Filter와 Interceptor (9/11) –서버 영역 interceptor– 서비스 응답

- ✓ javax.ws.rs.ext.JAXRSWriterInterceptor를 구현하여 사용합니다.
- ✓ 결과 메시지 관련 작업을 주로 수행 합니다.

```
import javax.ws.rs.ext.WriterInterceptor;

@Provider
public class JAXRSWriterInterceptor implements WriterInterceptor {

    @Override
    public void aroundWriteTo(WriterInterceptorContext context) throws IOException,
WebApplicationException {
        MultivaluedMap<String, Object> headers = context.getHeaders();
        headers.add("Content-Encoding", "gzip");
        OutputStream outputStream = context.getOutputStream();
        context.setOutputStream(new GZIPOutputStream(outputStream));

        context.proceed();
    }
}
```

예제는 서비스 응답 시 GZIP outputStream을 사용하는 예제입니다.

## 6.7 Filter와 Interceptor (10/11) – @NameBinding

- ✓ Filter나 interceptor에 선택적으로 적용할 때 사용 합니다.
- ✓ @javax.ws.rs.NameBinding을 이용하여 구현 합니다.

```
@NameBinding  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.METHOD, ElementType.T  
YPE})  
public @interface RequestLogger {  
}
```

어노테이션 정의

```
@RequestLogger  
public class RequestLoggerFilter implements  
ContainerRequestFilter {  
@Override  
public void filter(ContainerRequestContext  
requestContext)  
throws IOException {  
// Method implementation is not shown in this  
// example for brevity  
}}
```

RequestLogger Filter 정의

```
// imports are omitted for brevity  
@Stateless  
@Path("hr")  
public class HRService {  
  
    @GET  
    @Path("departments/{id}")  
    @Produces(MediaType.APPLICATION_JSON)  
    @RequestLogger  
    public Department findDepartment(@PathParam("id")  
Short id)  
    {  
        findDepartmentsEntity(id);  
        return department;  
    }  
}
```

RequestLogger Filter 적용

## 6.7 Filter와 Interceptor (11/11) –DynamicFeature

- ✓ Filter나 interceptor 를 비즈니스 상황에 따라 선택적으로 선별하여 적용할 때 사용합니다.
- ✓ @NameBinding보다 좀 더 세밀하게 선택적으로 적용할 수 있습니다.
- ✓ javax.ws.rs.container.DynamicFeature 을 이용하여 구현 합니다.

```
import javax.ws.rs.container.DynamicFeature;

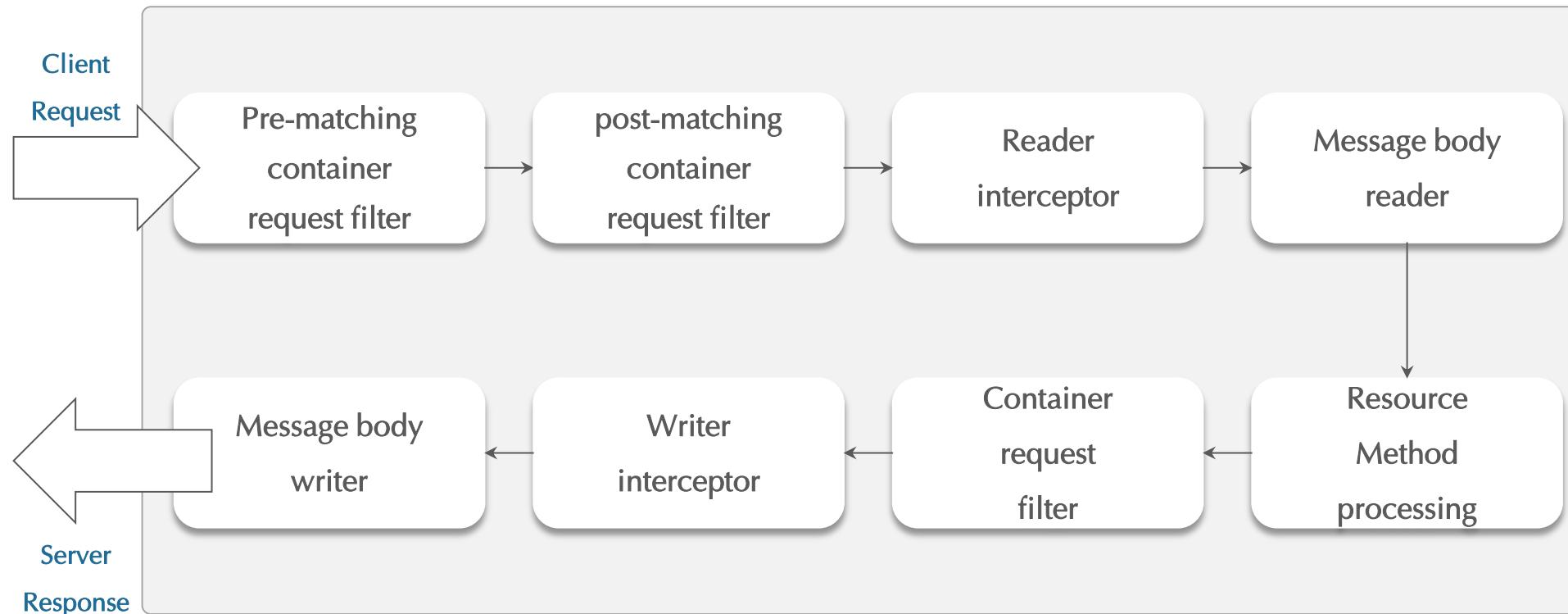
@Provider
public class DynamicFeatureRegister implements DynamicFeature {

    @Override
    public void configure(ResourceInfo resourceInfo,
        FeatureContext context) {
        if (resourceInfo.getResourceMethod().isAnnotationPresent (RequestLogger.class)) {
            context.register(RequestLoggerFilter.class);
        }
    }
}
```

예제는 RequestLogger.class 어노테이션이 적용된 경우  
RequestLoggerFilte를 추가 합니다.

## 6.8 JAX-RS 자원의 라이프 사이클

- ✓ 서비스 요청 부터 응답 까지 라이프 사이클은 다음과 같습니다.
- ✓ 서비스 중 예외가 발생한 경우 서비스는 중단 되고 ExceptionMapper에 정의된 예외 응답이 전달 됩니다.



## 6.9 RESTful 고급 기능 (실습)

---

- ✓ 실습 자료를 별도로 설명 후 안내를 합니다.
- ✓ 실습 가이드를 참조합니다.

RESTful 서비스 고급 기능 예제 시스템 설명과 시연

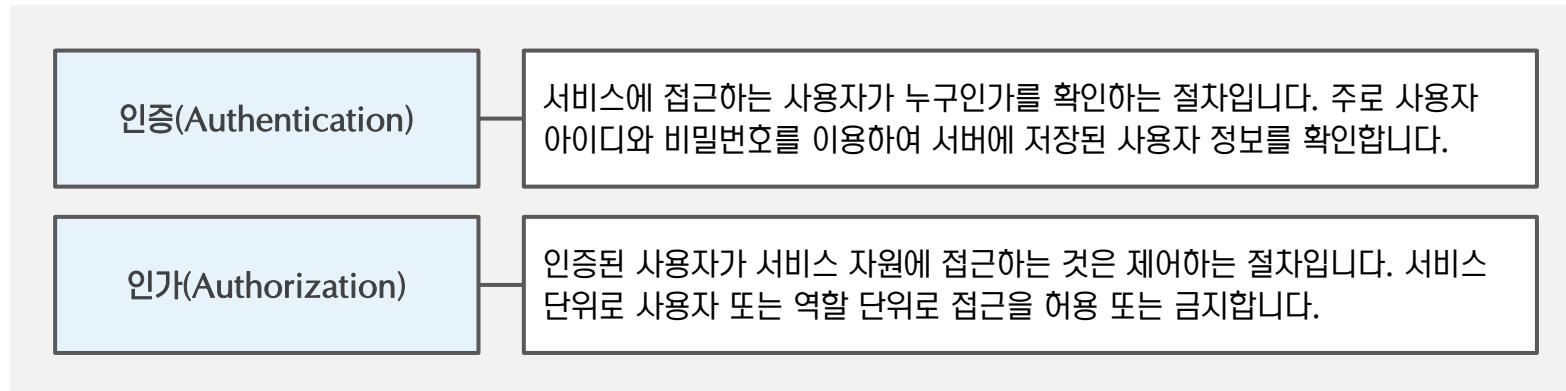


## 7. 안전한 RESTful 서비스

- 
- 7.1 서비스 보안
  - 7.2 HTTP 기본 인증
  - 7.3 OAuth 1.0
  - 7.4 OAuth 2.0

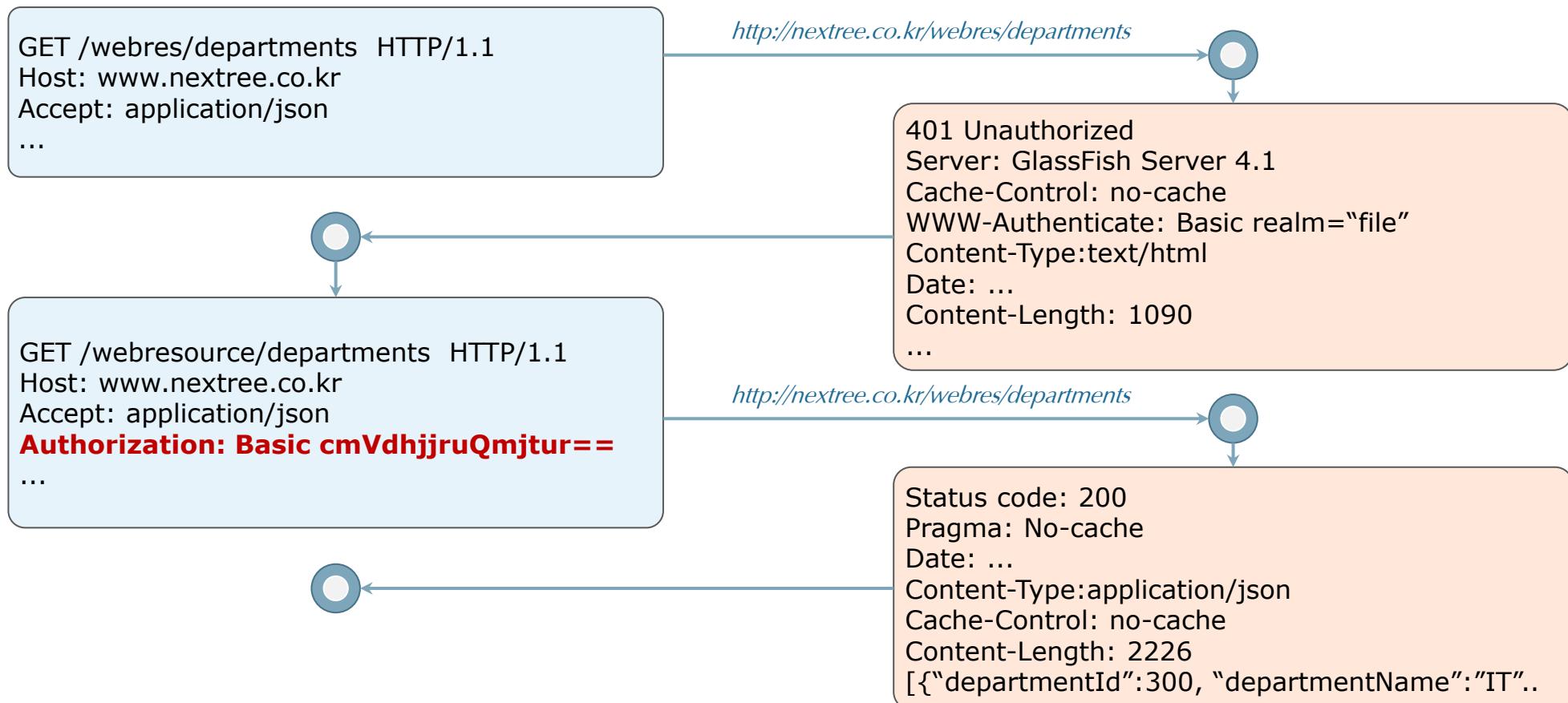
## 7.1 서비스 보안

- ✓ 대부분의 웹 서비스를 공개적으로 제공하지만, 그래도 정보 접근과 처리량에 대한 제어는 필요합니다.
- ✓ 사용자 계정을 통해서 이 두 가지를 제어할 수 있습니다. ← 예, 일일 조회 건수 제한
- ✓ 대부분 API 제공 업체는 비슷한 방식으로 서비스 자원으로의 접근을 관리하고 있습니다.
- ✓ 보안의 여러 요소 중에 RESTful 서비스와 관련된 것은 “인증”과 “인가”입니다.



## 7.2 HTTP 기본 인증(1/7) – 시나리오

- ✓ HTTP 기본 인증은 Base64로 인코딩된 사용자 이름과 비밀번호를 보내는 방식을 사용합니다.
- ✓ 요청할 때마다, 헤더의 “Authorization” 항목에 이 증명서(credential)을 설정하여 보냅니다.
- ✓ 증명서(credential)는 웹 브라우저의 캐시에 두어, 사용자가 매번 입력하지 않아도 됩니다.
- ✓ 증명서 탈취나 Base64 인코딩의 해독 가능성이 낮은 수준의 보안을 유지합니다.



## 7.2 HTTP 기본 인증(2/7) – 클라이언트

- ✓ Java REST 클라이언트에서 HTTP 기본 인증을 위해 ClientRequestFilter 인터페이스를 구현합니다.

```
@Provider
public class ClientAuthenticationFilter implements ClientRequestFilter {
    private final String user;
    private final String password;

    public ClientAuthenticationFilter(String user, String password) {
        this.user = user;
        this.password = password;
    }

    public void filter(ClientRequestContext requestContext) throws IOException {
        MultivaluedMap<String, Object> headers = requestContext.getHeaders();
        final String basicAuthentication = getBasicAuthentication();
        headers.add("Authorization", basicAuthentication);
    }

    private String getBasicAuthentication() {
        String token = this.user + ":" + this.password;
        try {
            byte[] encoded = Base64.getEncoder().encode(token.getBytes("UTF-8"));
            return "BASIC " + new String(encoded);
        } catch (UnsupportedEncodingException ex) {
            throw new IllegalArgumentException("Cannot encode with UTF-8", ex);
        }
    }
}
```

## 7.2 HTTP 기본 인증(3/7) – 클라이언트

- ✓ 클라이언트에서 WebTarget 객체를 만들고, 필터를 등록(webTarget.register(filter))합니다.
- ✓ 요청이 전송(transport) 계층에 도달하기 전에 등록된 모든 filter를 실행합니다.
- ✓ 클라이언트는 서버가 요구하는 인증 방식을 알고 서비스를 요청해야 합니다.

```
public class HRServiceAsynchJAXRSClient {  
  
    private static final String BASE_URI = "http://localhost:8080/hrapp";  
  
    ...  
  
    public Integer totalDepartments(String username, String password) {  
        //  
        Client client = javax.ws.rs.client.ClientBuilder.newClient();  
        WebTarget webTarget = client.target(BASE_URI).path("departments");  
  
        // 클라이언트에 필터를 등록합니다.  
        ClientAuthenticationFilter filter = new ClientAuthenticationFilter(username, password);  
        webTarget.register(filter);  
  
        Integer count = webTarget.path("count").request(javax.ws.rs.core.MediaType.TEXT_PLAIN).get(Integer.class);  
        client.close();  
  
        return count;  
    }  
}
```

## 7.2 HTTP 기본 인증(4/7) – 서비스의 web.xml

- ✓ J2EE 컨테이너의 web.xml 파일에 서비스 인증 정보를 설정합니다.
- ✓ 필요할 경우, 컨테이너 고유의 배치 서술자(예, glassfish-web.xml)에 필요한 정보를 설정합니다.
- ✓ 아래는 모든 GET 요청에 인증 제약조건을 설정하고 있습니다.

```
<web-app>
    <security-constraint>
        <web-resource-collection>
            <web-resource-name> Protected resource </web-resource-name>
            <url-pattern>/*</url-pattern>
            <http-method>GET</http-method>
        </web-resource-collection>
        <auth-constraint>
            <role-name>APIUser</role-name>
        </auth-constraint>
        <user-data-constraint>
            <transport-guarantee> CONFIDENTIAL </transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
        <!-- realm name used in GlassFish -->
        <realm-name>file</realm-name>
    </login-config>
    <security-role>
        <role-name>APIUser</role-name>
    </security-role>
</web-app>
```

web.xml

## 7.2 HTTP 기본 인증(5/7) – 컨테니어의 web.xml

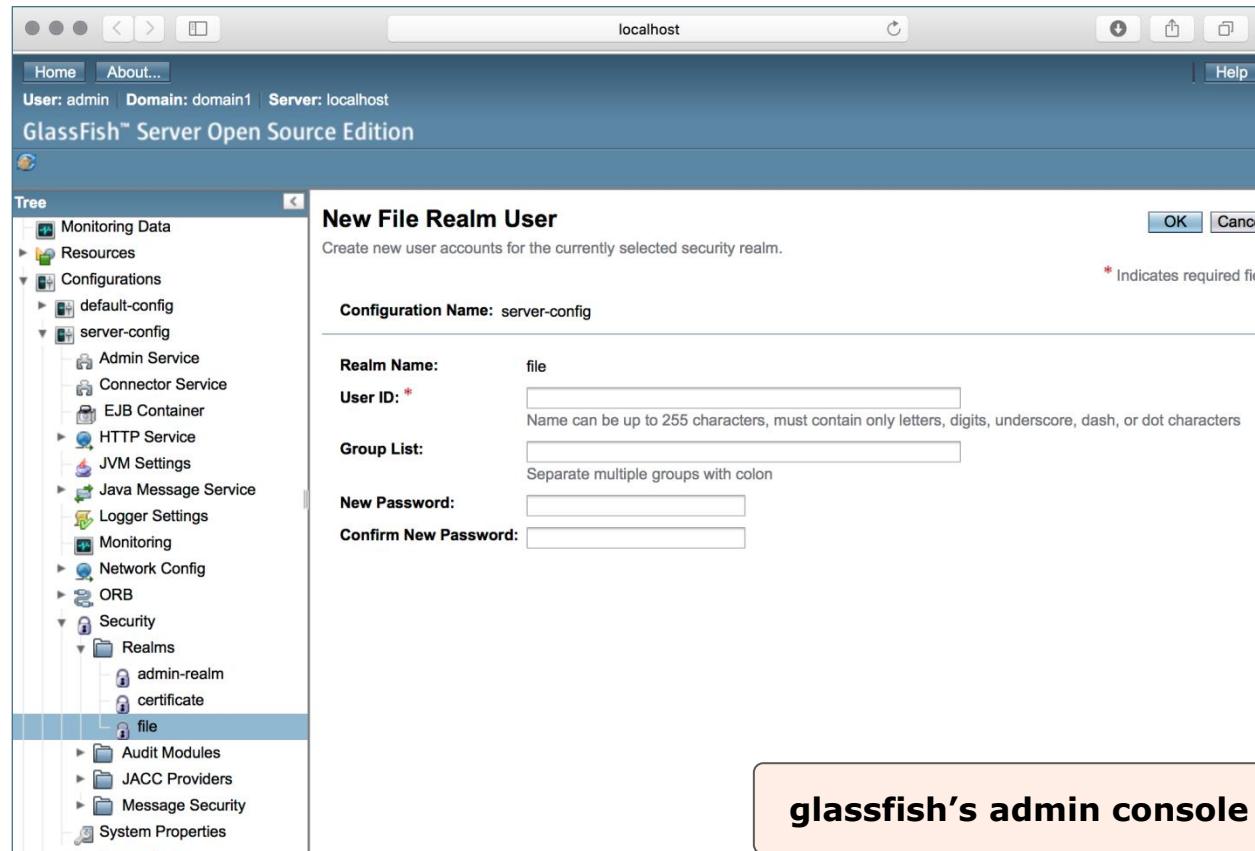
- ✓ 서비스의 역할 이름(여기서는 APIUser)을 애플리케이션의 사용자 그룹(여기서는 Users)과 매핑하여 줍니다.
- ✓ 애플리케이션의 사용자와 그룹은 애플리케이션의 관리자 기능을 이용하여 추가합니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<glassfish-web-app error-url="">
    <!-- Other entries go here -->
    <security-role-mapping>
        <!-- maps "Users" group to APIUser -->
        <b><role-name>APIUser</role-name>
<group-name>Users</group-name></b>
    </security-role-mapping>
</glassfish-web-app>
```

glassfish-web.xml

## 7.2 HTTP 기본 인증(6/7) – 사용자 관리

- ✓ GlassFish를 예로 들어 봅니다.
- ✓ Admin 콘솔에서 Configuration > server-config > Security > Realms > File을 선택합니다.
- ✓ 사용자 정보를 저장할 곳을 file로 선택합니다. ← file / DB / LDAP
- ✓ 사용자/비밀 번호를 등록하고, 그룹을 설정합니다.



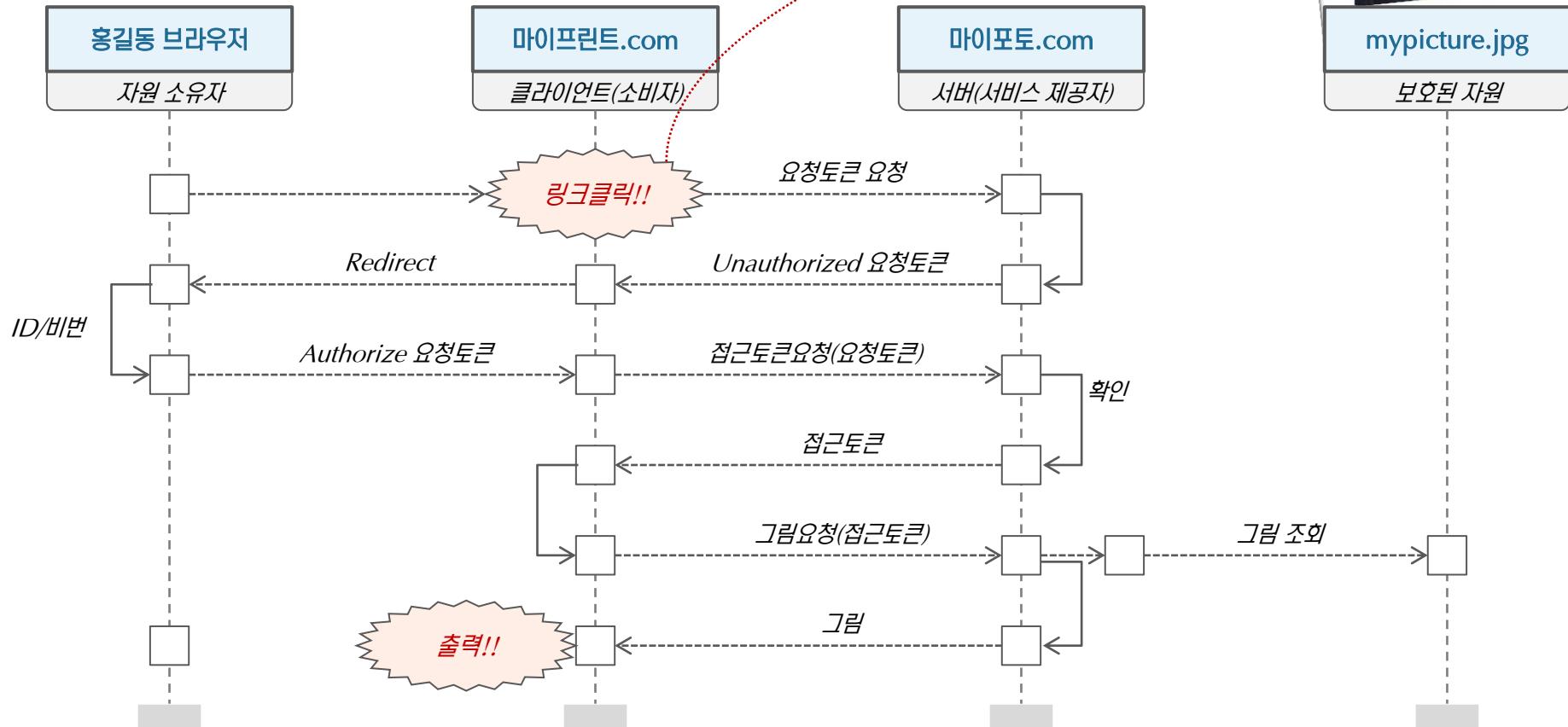
## 7.2 HTTP 기본 인증(7/7) – 요약

---

- ✓ 기본 인증 방식의 보안 수준은 근본적으로 낮을 수 밖에 없습니다.
- ✓ 증명서(credentials)를 일반 텍스트로 보냅니다. 해킹 가능성이 매우 높습니다.
- ✓ 이를 방지하기 위해서는 Secure Socket Layer(SSL)이나 Transport Layer Security(TLS) 프로토콜을 사용해야 합니다.
- ✓ 애플리케이션 서버에서 SSL/TLS 를 켜면 URI는 <https://<REST-RESOURCE-URI>> 포맷입니다.

## 7.3 OAuth 1.0 – 인가 흐름

- ✓ 마이프린트.com 사와 마이포토.com 사는 OAuth 1.0 프로토콜을 사용합니다.
- ✓ 마이프린트 사이트 사진 목록에서 사진을 선택하여 출력하려 합니다.
- ✓ 마이포토.com 사이트로 요청을 하려하는데 Request Token이 유효하지 않습니다.
- ✓ client 증명서(ID/비번), 임시 증명서(요청 토큰), 토큰 증명서(접근 토큰)



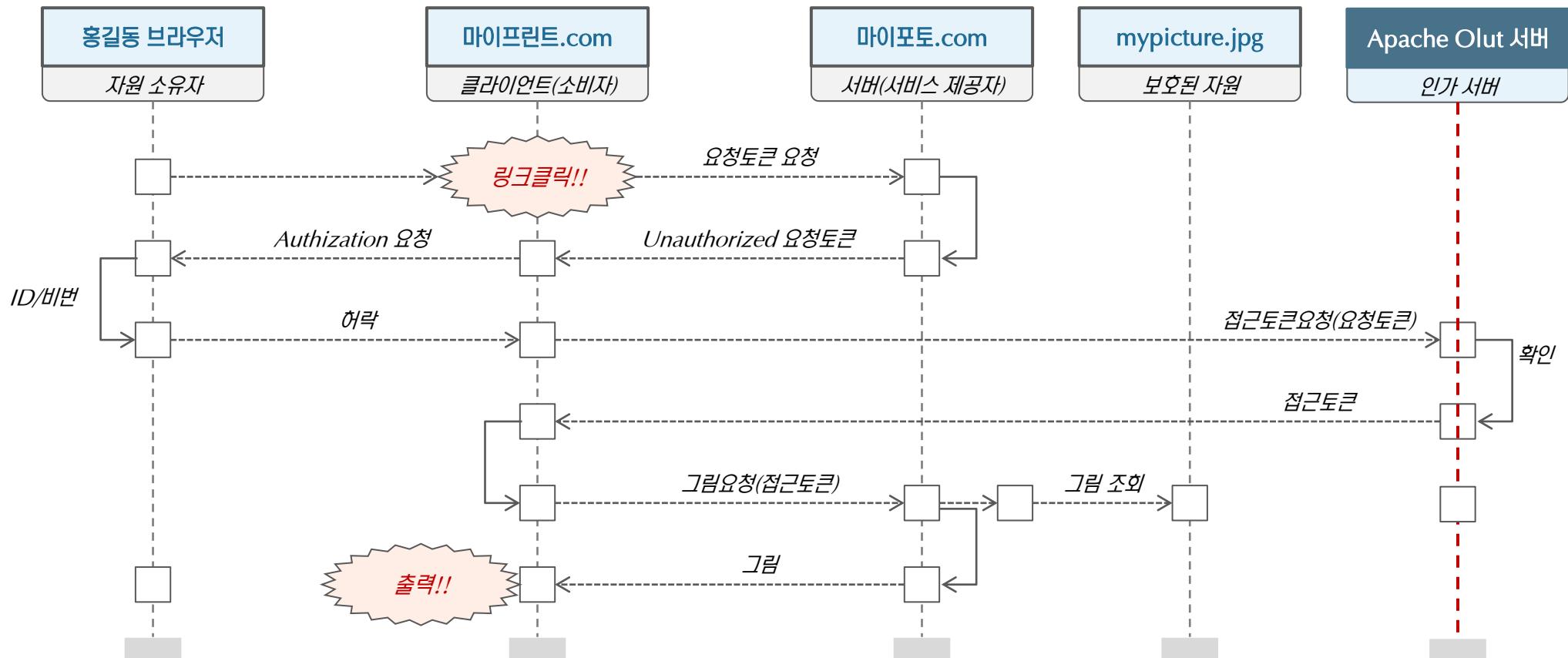
## 7.4 OAuth 2.0

---

- ✓ OAuth2.0은 최신 버전으로 클라이언트 개발을 쉽게 하는데 초점을 두었습니다.
- ✓ OAuth 2.0은 완전히 새로운 프로토콜로써 OAuth 1.0과 하위 호환성을 보장하지 않습니다.
- ✓ 다양한 클라이언트(웹, 데스크 탑, 모바일 폰, IoT 디바이스, 등)의 인가 흐름을 지원합니다.
- ✓ OAuth 2.0 개선 내용
  - 클라이언트가 API를 호출할 때마다, 요청 토큰의 비밀번호를 이용하여 서명(signature)을 생성해서 보내야 하고, 서버는 동일한 서명을 생성해서 비교해서 일치하면 접근을 허용하였습니다. OAuth 2.0은 SSL/TLS를 사용하므로 서명을 생성할 필요가 없습니다.
  - OAuth 1.0은 웹 클라이언트의 inbound와 outbound를 위해 설계하였습니다. OAuth 2.0은 다양한 클라이언트의 인가 흐름을 지원하여 매우 유연한 프로토콜로 발전했습니다.
  - 인가 흐름에 참여하는 당사자(party)를 명확하게 정의하였습니다. 클라이언트, 자원 소유자, 자원 서버, 인가 서버.
  - 접근 토큰에 유효 시간을 설정할 수 있습니다. 보안을 강화하여 부적절한 접근 기회를 줄였습니다.
  - Refresh 토큰 – 현재 토큰이 만료되면 전체 인가 절차를 밟지 않아도 새로운 토큰을 얻을 수 있습니다.

## 7.4 OAuth 2.0 – 인가 흐름

- ✓ 클라이언트는 보호된 자원에 접근하기 위해 소유자( 또는 인가 서버)에게 인가를 요청합니다.
- ✓ 클라이언트는 허락에 포함된 정보(ID, 비밀번호, 허가 타입)를 인가서버에 넘겨주고 접근 토큰을 요청합니다.
- ✓ 인가 서버는 클라이언트를 확인하고 인가 허락을 검증한 후에 접근 토큰을 발행합니다.





## 8. RESTful 서비스 서술과 찾기

8.1 WADL

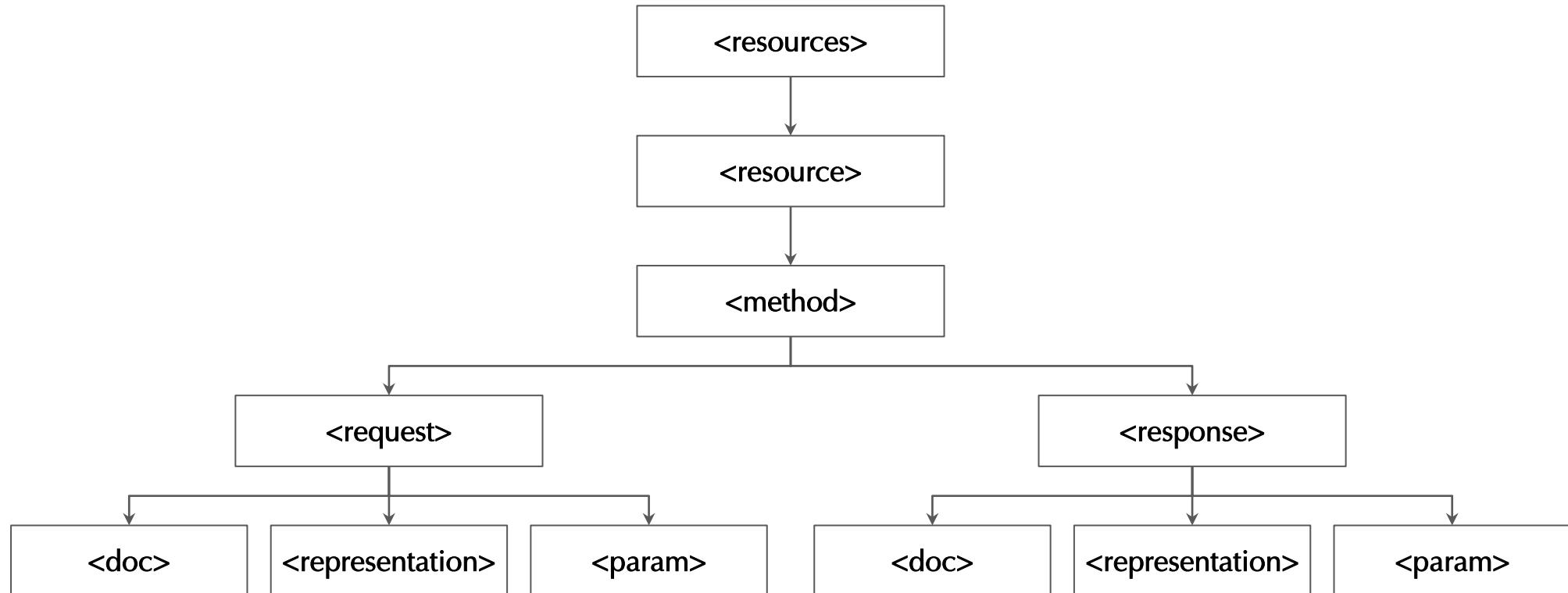
8.2 WAML

8.3 Swagger

8.4 요약

## 8.1 WADL[1/3] – 개요

- ✓ WADL(Web Application Description Language)은 HTTP 기반 앱을 위한 서술 언어입니다.
- ✓ 2009년 캔마이크로시스템 사에서 W3C에 표준안으로 제출했으나 아직은 표준이 아닙니다.
- ✓ WADL은 RESTful 웹 서비스가 제공하는 자원을 자원과 자원 간의 관계 모델로 표현합니다.
- ✓ WADL 명세서는 [www.w3.org/Submission/wadl](http://www.w3.org/Submission/wadl) 을 참조합니다.



## 8.1 WADL(2/3) – 예제

✓ 자원의 메소드를 WADL로 표현하면 다음과 같습니다.

```
@GET  
@Path("{id}")  
@Produces("application/json")  
public Department findDepartment(  
    @PathParam("id") Short id){  
    //  
    // 처리 로직...  
}
```

```
<application xmlns="http://wadl.dev.java.net/2009/02">  
    <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.10.4 2014-08-08 15:09:00" />  
    <grammars />  
    <resources base="http://localhost:8080/hrapp/webresources/">  
        <resource path="{id}">  
            <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id" style="template" type="xs:short" />  
            <method id="findDepartment" name="GET">  
                <response>  
                    <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02"  
                        xmlns="" element="department" mediaType="application/json" />  
                </response>  
            </method>  
        </resource>  
    </resources>  
</application>
```

## 8.1 WADL(3/3) – 요약

---

- ✓ Jersey 프레임워크는 RESTful 자원으로부터 WADL을 생성하는 플러그-인을 갖고 있습니다.
- ✓ WADL을 기본으로 생성하며, URI/application.wadl 을 호출하면 WADL 설명을 볼 수 있습니다.
- ✓ WADL로부터 Java 클라이언트를 생성할 수 있습니다. →<https://wadl.java.net>에서 wadl2java.bat
- ✓ 여러 API 벤더가 지원을 하고는 있지만, XML 표현에 거부감을 느끼는 경우도 많습니다.

## 8.2 RAML[1/2] – 개요

- ✓ RAML(RESTful API Modeling Language)은 사람은 읽을 수 있고, 기계는 처리할 수 있는 언어입니다.
- ✓ HTTP 구성요소를 명확하게 문서화 합니다. → 자원, 메소드, 패러미터, 응답, 미디어 타입 등
- ✓ 2013년에 RAML 워크그룹에서 제안을 했으며, 뮬소프트, 페이팔, 인튜이트, 시스코 등이 참여하고 있습니다.
- ✓ YAML(Yaml Ain't Markup Language) 표준을 바탕으로 하여 정의하였습니다.
- ✓ YAML 구조
  - 기본 정보
  - 보안
  - 자원과 포함된 자원
  - HTTP 메소드
    - Query 패러미터
    - 요청 데이터
    - 응답

## 8.2 RAML(2/2) – 예제

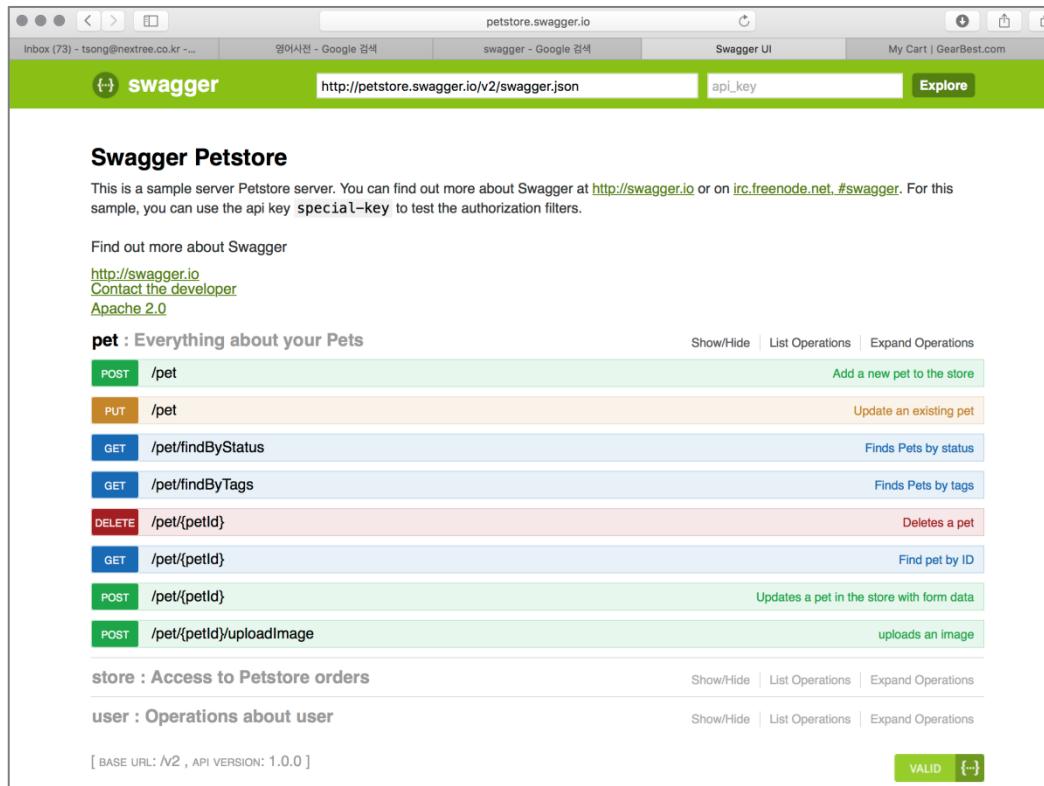
✓ RAML(RESTful API Modeling Language)은 사람은 읽고 기계는 처리할 수 있는 언어입니다.

```
@GET  
@Path("{id}")  
@Produces("application/json")  
public Department findDepartment(  
    @PathParam("id") Short id){  
    //  
    // 처리 로직...  
}
```

```
#%RAML 0.8  
title: department resource  
version: 1.0  
baseUri: "http://localhost:8080/hrapp/webresources"  
schemas:  
    - department: !include schemas/department.xsd  
    - department-jsonschema: !include schemas/department-jsonschema.json  
/departments:  
get: /{id}:  
    uriParameters:  
        id:  
            type: integer  
get:  
    responses:  
        200:  
            body:  
                application/json:  
                    schema: department-jsonschema  
                    example: !include examples/department.json
```

## 8.3 Swagger

- ✓ Swagger는 RESTful 웹 서비스를 서술하고, 생성하고, 소비하고, 가시화하는 프레임워크입니다.
- ✓ Java, Scala, Groovy, JavaScript, .Net 등을 지원합니다. → [www.swagger.io](http://www.swagger.io)
- ✓ Wordnik에서 내부 사용 목적으로 개발했으며, 현재 버전은 2.0이고, 100 퍼센트 오픈소스입니다.
- ✓ PayPal, Apigee, 3scale 등과 같은 벤더에서 적극적으로 참여하고 있습니다.
- ✓ 세 개의 주요 컴포넌트로 구성되어 있습니다. → 서버, 클라이언트, 사용자 인터페이스



## 8.4 요약

- ✓ RESTful 웹 서비스 명세를 서술하고 관리하기 위한 세 가지 방안이 있습니다.
- ✓ 세 가지 어느 것도 표준은 아니지만 벤더 사에서 대부분 지원을 하고 있습니다.
- ✓ 대중성, 사용성 면에서는 Swagger가 가장 앞에 서 있습니다.

특징	WADL	RAML	Swagger
발표일	2009	2013	2011
파일 포맷	XML	RAML	JSON/YAML
오픈소스 여부	예	예	예
상용 제품 여부	아니오	예	예
지원 언어	Java	js, Java, Node, PHP, Python, Ruby	Clojure, Go, js, Java, .Net, Node, PHP, Scala
인증	No	Basic, Digest, OAuth1, OAuth2	Basic, API Key, OAuth2
API 콘솔	No	Yes	Yes
서버 코드 생성(Java)	No	Yes	Yes
클라이언트 코드 생성(Java)	Yes	Yes	Yes

✓ 토론

## 감사합니다...

- ❖ 송태국 ([tsong@nextree.co.kr](mailto:tsong@nextree.co.kr))
- ❖ 넥스트리컨설팅(주) 대표이사
- ❖ 넥스트리소프트(주) 부사장
- ❖ [www.nextree.co.kr](http://www.nextree.co.kr)