Anyframe Spring REST Plugin

Version 1.1.0

I.Introduction

Spring REST Plugin은 Spring 3부터 제공되는 Spring MVC 기반의 RESTful 웹서비스 구현 기능을 활용하는 방법을 가이드하기 위한 샘플 코드와 필요한 참조 라이브러리들로 구성되어 있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 springrest plugin을 설치한다.

```
mvn anyframe:install -Dname=springrest
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Plugin Name	Version Range
Query [http://dev.anyframejava.org/docs/anyframe/plugin/optional/query/1.6.0/reference/htmlsingle/query.html]	2.0.0 > * > 1.4.0

II.What is REST?

설치한 Spring REST plugin의 샘플 코드를 이용해서 Spring에서 제공하는 REST 지원 기능들을 살펴보기 전에 먼저 REST라는 개념이 무엇인지에 대해서 간단히 살펴보도록 하자.

REST는 **REpresentational State Transfer**의 약자로, 통식 규약이나 표준 또는 스펙이 아니라 분산 하이퍼미디어 시스템을 위한 www같은 소프트웨어 아키텍처의 한 형식이다. REST라는 용어는 2000년 로이필딩(Roy Fielding)의 박사학위 논문에서 처음 소개된 것으로 **네트워크 상에서 클라이언트와 서버 사이의 통신 방식**에 대해서 서술하고 있다.

1.REST 아키텍처

REST 아키텍처는 다음과 같은 요소들로 구성된다.

- **Resource**

  REST에서 가장 중요한 개념은 바로 유일한 ID를 가지는 Resource가 서버에 존재하고, 클라이언트는 각 Resource의 상태를 조작하기 위해 요청을 보낸다는 것이다. 일반적으로 Resource는 Movie, Student, Product 등과 같은 명사형의 단어이고, HTTP에서 이러한 **Resource를 구별하기 위한 ID는 '/moviefinder/movies/MV-00001'와 같은 URI이다.**

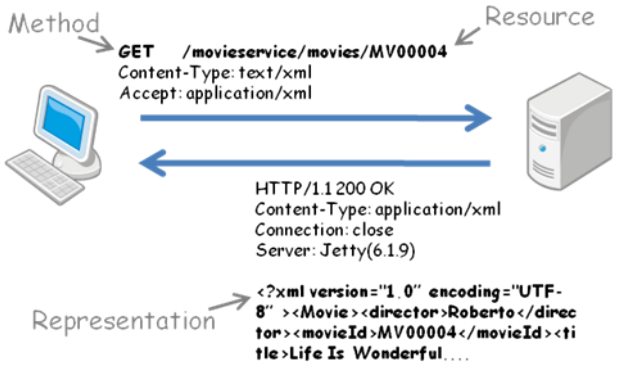
- **Method**

  GET, DELETE 등과 같이 Resource를 조작할 수 있는 동사형의 단어를 Method라고 한다. **클라이언트는 URI를 이용해서 Resource를 지정하고 해당 Resource를 조작하기 위해서 Method를 사용**한다. HTTP에서는 GET, POST, PUT, DELETE 등의 Method를 제공한다.

- **Representation of Resource**

  클라이언트가 서버로 요청을 보냈을 때, 서버가 응답으로 보내주는 **Resource의 상태를 Representation**이라고 한다. REST에서 하나의 Resource는 여러 형태의 Representation으로 나타내어 질 수 있다. 이를 **Content Negotiation**이라고 하는데, 뒤에서 자세히 설명할 것이다.

위의 구성 요소들을 바탕으로 REST 아키텍처에서 클라이언트가 'http://example.com/movies/MV00004'라는 URI를 가진 Movie Resource를 조회하는 과정을 그림으로 표현하면 다음과 같다.



2.Key Principles of REST

REST는 네트워크 아키텍처 원칙의 모음이다. 여기서 네트워크 아키텍처 원칙이란 Resource를 정의하고 Resource에 대한 ID(URI)를 지정하는 방법에 대한 개괄을 말한다. 간단한 의미로는, 도메인 지향 데이터를 HTTP위에서 전송하기 위한 아주 간단한 인터페이스를 설명한 것이라고 할 수 있다. REST의 핵심 원칙는 아래와 같이 5가지 정도로 요약할 수 있다. (출처 : http://www.infoq.com/articles/rest-introduction)

2.1.Give every "thing" an ID

위에서 설명했듯이 모든 Resource에는 URI라고 하는 유일한 ID를 부여한다. 클라이언트는 URI를 이용해서 수많은 Resource를 식별하므로 이 URI 설계를 위한 다음과 같은 Design Rule이 RESTful Web Services [http://oreilly.com/catalog/9780596529260]라는 책에서 소개되고 있다. 이는 많은 사람들이 그동안 RESTful 아키텍처를 적용하면서 축적된 경험을 바탕으로 만들어진 URI 설계 가이드이다.

- URI는 직관적으로 Resource를 인식할 수 있는 단어들로 구성할 것

  '/movies', '/products' 등과 같이 직관적으로 어떤 정보를 제공하는지 알 수 있도록 URI를 구성할 것을 가이드하고 있다.

- URI는 계층구조로 구성할 것

  '/hotels/hayatt/bookings/20101128'와 같이 URI path가 계층적인 구조를 가지도록 구성하는 것이 좋다.

- URI의 상위 path는 하위 path의 집합을 의미하는 단어로 구성할 것

  '/hotels/hayatt/bookings/20101128'와 같이 'hotels'는 'hayatt'의 집합이므로 '/hotels' 만으로도 호텔목록이라는 정보를 제공할 수 있는 유효한 URI가 된다.

이 외에도 여러가지 가이드들이 존재하지만 특징적인 것들만 나열하였다.

위와 같은 가이드에 맞춰 URI를 만들면 '/hotels/hilton', '/hotels/hayatt' 처럼 비슷한 패턴의 URI가 많이 생성된다. 이런 URI를 쉽게 관리할 수 있도록 URI를 추상화할 수 있도록 도와주는 것이 URI Template이다. **URI Template**은 '/movies/{movieId}'와 같이 **하나 이상의 변수를 포함하고 있는 URI 형식의 문자열**이다. URI Template에 대한 자세한 내용은 proposed RFC [http://tools.ietf.org/html/draft-gregorio-uritemplate-04]를 참조하기 바란다.

2.2.Link things together

하나의 Resource는 여러 개의 다른 Resource 정보를 포함할 수 있다. 아래 예에서 보는 것 처럼 Order는 Product와 Customer를 포함하고 있어서 Order정보 조회 요청에 대한 응답으로 전달된 Representation에 Product와 Customer에 대한 link가 포함되어있다. Representation이 다른 Resource에 대한 URI를 link로 포함하기 때문에 필요에 따라 클라이언트가 추가적인 정보를 조회할 수 있다. 이 개념은 'HATEOAS(Hypermedia As The Engine Of Application State)라는' 용어로도 많이 표현된다.

```
<order self='http://example.com/customers/1234' >
    <amount>23</amount>
    <product ref='http://example.com/products/4554' />
    <customer ref='http://example.com/customers/1234' />
</order>
```

클라이언트는 'Order'라는 Resource에 대한 Representation을 전달받았고, 필요에 따라 'Product'나 'Customer'의 정보를 다시 요청하면 된다. 즉, 서버에서는 또 다른 State로 전환할 수 있는 Resource의 link를 전달하기만 하고, 전환되어야 할 State의 순서를 지정하지는 않는다.

2.3.Use standard methods

Resource에 대한 CRUD 조작을 위해서 HTTP에서 제공하는 standard method를 사용할 것을 권장한다. 클라이언트가 서버의 Movie를 삭제하기 위해서 기존에는 '/movies.do?id=MV-00001&method=delete'와 같은 방식으로 요청했다면, REST에서는 '/movies/MV-00001'라는 URI와 HTTP의 DELETE method의 조합으로 요청할 수 있다.

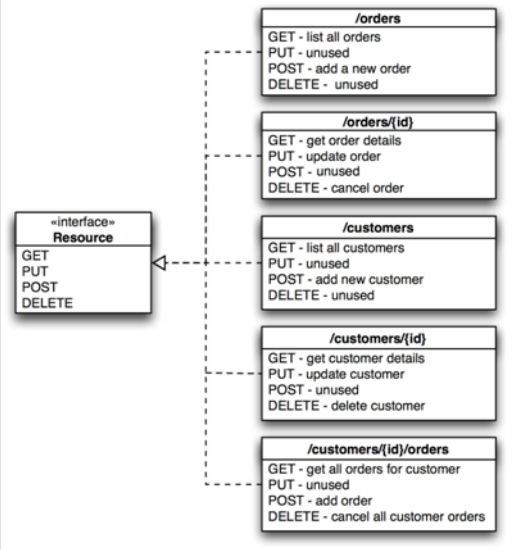
일반적으로 대부분의 브라우저에서는 GET, POST만 지원하기 때문에 REST 구현을 위한 Spring이나 Apache CXF같은 프레임워크들에서는 모든 HTTP method를 지원하기 위한 방안을 제공하고 있다.

Not support in most web browser

HTTP Methods	GET	POST	PUT	DELETE
Meaning	GET to retrieve information	POST to add new information	PUT to update information	Remove (logical) an entity
Example	GET /store/customers/123456	POST /store/customers	PUT /store/customers/123456	DELETE /store/customers/123456

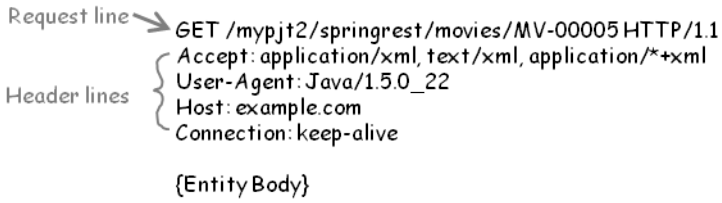
HTTP에서 제공하는 Method에 대한 자세한 내용은 HTTP/1.1 RFC [http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html]에 정의되어 있다.

예를 들어, 상품 주문을 관리하는 어플리케이션에서 RESTful 웹 서비스를 제공한다고 할 때, URI와 HTTP method의 조합은 아래의 그림처럼 정리할 수 있다.

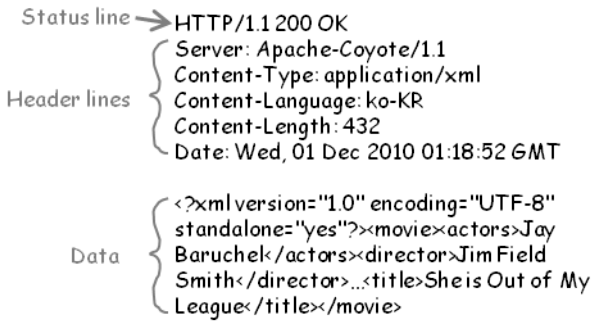


2.4.Resources with multiple representations

HTTP 기반의 REST에서 클라이언트는 자신이 처리할 수 있는 Format으로 Representation을 달라고 서버에게 요청할 수 있다. Request message의 **Accept header**에 클라이언트가 처리할 수 있는 Format을 명시하여 서버로 요청을 보내면 된다. 예를 들어, 아래의 HTTP Request는 **"'MV-00005'라는 ID를 가진 영화의 상세 정보를 XML 형태로 줘"**라는 의미가 된다.



위의 요청을 받은 서버는 응답으로 다음과 같은 Response Message를 전달할 것이다.



Accept header에 다른 Format을 명시하면 서버는 다른 형태의 응답을 전달할 것이다.

이와 같이 하나의 Resource는 여러개의 Representation을 가질 수 있다. 이를 **Content Negotiation [http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html#sec12]**이라고 한다.

여기서 한 가지 문제점은 일반적인 브라우저에서는 Accept Header 값을 고정하여 전송하기 때문에, Accept Header 값을 기반으로 한 Content Negotiation이 불가능하다는 것이다. 그래서 이에 대한 대안으로 **URL path에 확장자를 붙여, 확장자를 통해 클라이언트가 원하는 Representation을 표시하는 방법**을 사용한다. 예를 들어, '/myapp/movies.pdf' 라는 요청이 들어오면 서버는 영화목록을 찾아서 PDF View로 클라이언트에게 전달하는 것이다.

2.5.Communicate statelessly

REST에서 서버는 클라이언트로 부터 들어오는 각 요청에 대한 상태를 저장하지 않도록 권장한다. 요청이 처리되기 위해서 필요한 모든 정보는 반드시 요청에 포함하도록 해야한다. 서버는 클라이언트 관련 정보를 저장할 필요가 없으므로 클라이언트의 수의 증가에도 시스템이 유연하게 대응할 수 있다.

III.Spring REST Supports

이제 위에서 설명한 REST 아키텍처를 적용한 서비스를 구현하기 위해서 Spring 3에서는 어떤 기능을 추가적으로 지원하는지 springrest plugin의 소스 코드와 함께 하나씩 자세히 살펴보도록 하자.

Spring의 REST를 위한 기능은 모두 Spring MVC를 기반으로 지원된다. 다양한 Annotation과 HTTP Request/Response Body 메세지 처리를 위한 HttpMessageConverter, Content Negotiation 지원을 위한 ViewResolver, 모든 HTTP method 사용을 위한 Filter, 그리고 REST 클라이언트 어플리케이션 개발에 도움을 주는 RestTemplate 등이 있다.

3.Request Mapping

위에서 언급했듯이, Spring에서 제공하는 REST 지원 기능들은 모두 Spring MVC 기반으로 되어 있다. REST 방식으로 노출되는 서비스는 곧 Controller의 메소드이기 때문에 기존에 웹 어플리케이션을 개발하던 방식과 크게 다르지 않다.

Resource의 ID인 URI를 Controller 클래스나 메소드에 매핑하기 위해서는 @RequestMapping을 사용한다. @RequestMapping이 URI Template을 지원하기 때문에 아래 샘플코드와 같이 사용할 수 있다.

```
@Controller
@RequestMapping("/movies")
public class MovieController {
    // ...
    @RequestMapping(value = "/{movieId}", method = RequestMethod.GET)
    public String get(@PathVariable String movieId, Model model) throws Exception {
        // ...
    }
}
```

또한 REST 아키텍처에서 가이드하고 있는 원칙 중 하나인, 모든 HTTP method 사용을 위해서 @RequestMapping에서 'method' 속성을 제공한다. 따라서, '/movies/MV-00001'이라는 URI가 GET으로 요청이 들어올 경우 위의 get() 메소드가 매핑될 것이다.

DispatcherServlet URL 매핑

기존에 Spring MVC를 기반으로 개발된 웹 어플리케이션에서는 'xxx.do'라는 형태의 URL을 사용했지만, 위에서 설명했듯이 REST 스타일의 URL은 '/movies', '/movies/MV-00001' 처럼 계층 구조로 사용가능하도록 설계되었다. 따라서 web.xml에 DispatcherServlet을 정의하고 매핑할 URL 패턴을 '/'로 지정해야한다.

이 경우 css 나 이미지 등의 static 리소스 URL도 DispatcherServlet을 통하게 되어 화면이 정상적으로 동작하지 않는 문제가 있다. 그래서 Spring에서는 **<mvc:default-servlet-handler/>**를 제공하고 있다. 이 태그의 역할은 내부적으로 **DefaultServletHttpRequestHandler**를 등록해주는 것이다. 이 핸들러는 가장 낮은 우선순위를 가지고 있고, /**로 매핑되어 있다. 따라서 다른 handler mapping을 다 거친 후에 실패한 URL만 넘어오게 된다. DefaultServletHttpRequestHandler는 최종적으로 넘어온 요청을 처리하기 위해서 직접 static 리소스를 핸들링하는 것이 아니라 원래 서버가 제공하는 디폴트 서블릿으로 전달한다. 그래서 URLRewriteFilter 같은 것을 사용하지 않아도 간단하게 '/'를 DispatcherServlet에 매핑시킬 수 있게 된다.

그러나 springrest plugin의 경우 core plugin 등 다른 plugin들과 함께 섞여서 동작해야하기 때문에 <mvc:default-servlet-handler/>를 사용하지 않고, 기존에 정의된 DispatcherServlet에 아래와 같이 매핑만 추가하도록 설정했다.

```
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/springrest/*</url-pattern>
</servlet-mapping>
```

또한, Spring에서는 URI Template에 포함된 변수 값을 추출할 수 있도록 **@PathVariable**이라는 새로운 Annotation을 추가했다.

다음은 @PathVariable을 사용한 예이다.

```
@RequestMapping(value = "/{movieId}", method = RequestMethod.GET)
public String get(@PathVariable String movieId, Model model)
      throws Exception {
    Movie movie = this.movieService.get(movieId);
```

```
    // 중략
    return "springrestViewMovie";
}
```

'/movies/MV-00001'와 같은 URI로 요청이 들어왔을 때, 위의 get 메소드가 처리하게 되고 'MV-00001' 값은 'movieId' 입력 인자로 바인딩된다.

아래와 같이 변수명을 지정하여 사용하거나 여러개의 변수를 사용할 수도 있다.

```
@RequestMapping(value = "/movies/{movie}/posters/{poster}", method = RequestMethod.GET)
public String get(@PathVariable("movie") String movieId, @PathVariable("poster") String
 posterId, Model model)
      throws Exception {
    // 중략
    return "springrestViewMovie";
}
```

'/movies/*/posters/{posterId}'와 같이 Ant-style의 경로에도 사용할 수 있고, URI Template의 변수를 String이 아닌 다른 타입의 입력 인자로도 바인딩 가능하다.

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
}

@RequestMapping("/plans/{date}")
public void get(@PathVariable Date date) {
    // 중략
}
```

예로 '/plans/2010-09-05' URI로 들어온 요청은 위의 메소드가 처리할 것이고, '2010-09-05'는 date 입력 인자에 Date 타입으로 바인딩 될 것이다.

@RequestMapping에서 사용할 수 있는 속성은 core plugin 매뉴얼의 컨트롤러 구현 중 @RequestMapping [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.6.0/reference/htmlsingle/core.html#core_springmvc_controller_implementation_requestmapping] 내용을 참조하기 바란다.

4.URI Support Utils

Spring에서는 REST 방식에서 사용되는 URI 형식의 문자열을 생성하기 위해 UriTemplate, UriComponents 등의 util class를 제공한다.

4.1.UriTemplate

UriTemplate class는 중괄호({, })로 묶여진 변수를 포함하는 문자열을 실제 URI 형식으로 만들어주기 위해 Spring에서 제공하는 Class이다.

아래는 UriTemplate을 사용하여 URI 문자열을 만들어내는 사용 예제이다.

```
UriTemplate template = new UriTemplate("http://example.com/hotels/{hotel}/bookings/
{booking}");
System.out.println(template.expand("1", "42"));
```

```
UriTemplate template = new UriTemplate("http://example.com/hotels/{hotel}/bookings/
{booking}");
 Map<String, String> uriVariables = new HashMap<String, String>();
 uriVariables.put("booking", "42");
 uriVariables.put("hotel", "1");
 System.out.println(template.expand(uriVariables));
```

출력 결과 - http://example.com/hotels/1/bookings/42

위의 두 예제 실행 결과를 살펴보면 {hotel}, {booking} 대신에 "1", "42"로 값이 변경되었음을 확인 할 수 있다.

4.2.UriComponents

UriComponents class는 URI 형식의 문자열을 쉽게 작성하기 위해 Spring 3.1에서 새롭게 지원하는 class 이다. URI의 각 구성요소를 설정 할 수 있는 API를 제공한다.

```
UriComponents redirectUri = UriComponentsBuilder.fromPath(
  "/springrest/test/{movieId}").build()
  .expand(movie.getMovieId()).encode(); // movieId = "MV-000001"

  System.out.println(redirectUri.toUriString());
```

출력 결과 - "/springrest/test/MV-000001"

위의 예제에서 확인 할 수 있는 것처럼, UriComponentsBuilder class를 이용하여 UriComponents를 생성한 후, 중괄호({, })로 표시된 movieId에 값을 치환해서 URI문자열을 만들 수 있다. 그 밖에 host, port 등 URI에서 제공할 수 있는 정보를 API를 통해 제공하고 있다.

5.Redirection

Spring 3.1에서는 REST 환경에서 Redirect 요청을 처리할 때 속성값을 쉽게 전달할 수 있도록 FlashMap과 RedirectAttributes class를 제공한다.

5.1.RedirectAttributes

RedirectAttributes는 Redirect 동작을 수행 할 때 속성값을 URI variable, 혹은 parameter 형태로 쉽게 전달하기 위해 Spring에서 제공하는 Class이다.

```
@RequestMapping(method = RequestMethod.POST, consumes = "application/x-www-form-urlencoded")
public String create(@Valid Movie movie, BindingResult results,
  RedirectAttributes redirectAttr) throws Exception {

 this.movieService.create(movie);
 redirectAttr.addAttribute("movieId", movie.getMovieId())
   .addAttribute("title", movie.getTitle())
   .addFlashAttribute("message", "save successed!");

 return "redirect:/springrest/test/{movieId}";
}
```

위의 예제에서 RedirectAttributes를 이용하여 Redirect URI의 {movieId} 값을 설정하는 내용을 확인할 수 있다. URI에 포함 되지 않은 attribute에 대해서는 request의 parameter형태로 전송 한다. 또한 FlashMap에 특정 값을 추가 하기 위한 API도 제공 하고 있다.

5.2.FlashMap

FlashMap은 하나의 URL에서 Redirect 요청을 수행 할 때, 특정 값을 다른 URL에서 사용 하기 위해 제공되는 Class이다. Redirect 동작을 수행 하기 전 session에 값을 저장한 후, 한 번 사용 후 제거되는 특징을 가지고 있다.

FlashMap은 크게 두 가지 방법으로 사용 가능하다.

첫 번째 방법은 RedirectAttributes의 addFlashAttribute 메소드를 사용하는 방법이다.

두 번째 방법은 RequestContextUtils의 getOutputFlashMap 메소드를 사용하는 방법이다. RequestContextUtils class는 DispatcherServlet에서 설정된 request의 특정 값으로 접근을 도와주기위해 Spring에서 제공하는 Util class이다.

FlashMap에 저장한 값을 꺼내서 사용하기 위해서는 RequestContextUtils의 getInputFlashMap 메소드를 사용한다.

아래는 FlashMap을 사용하는 예제 코드이다.

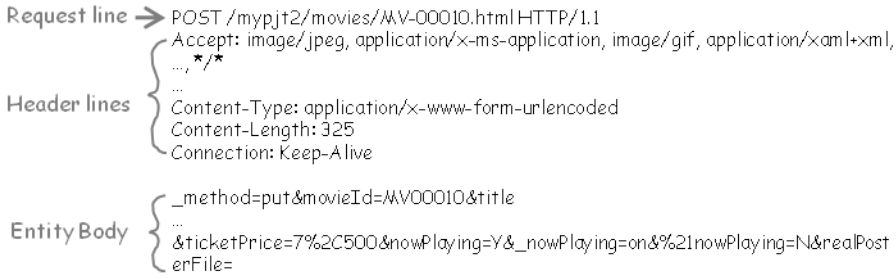
```
@RequestMapping(method = RequestMethod.POST, consumes = "application/x-www-form-urlencoded")
public String create(@Valid Movie movie, BindingResult results,
  RedirectAttributes redirectAttr) throws Exception {

 this.movieService.create(movie);

 // RedirectAttribute를 이용하여 FlashMap에 "message" 값을 저장
 redirectAttr.addAttribute("movieId", movie.getMovieId())
   .addFlashAttribute("message", "save successed!");
 return "redirect:/springrest/test/{movieId}";
```

```
}

@RequestMapping(method = RequestMethod.PUT, consumes = "application/x-www-form-urlencoded")
public String update(@Valid Movie movie, BindingResult results, HttpServletRequest request)
  throws Exception {

 this.movieService.update(movie);

 // RequestContextUtils를 이용하여 FlashMap에 "message" 값을 저장
 FlashMap fm = RequestContextUtils.getOutputFlashMap(request);
 fm.put("message", "update successed");
 return "redirect:" + redirectUri.toUriString();
}


@RequestMapping(value = "/{movieId}", method = RequestMethod.GET)
public String get(@PathVariable String movieId, Model model,
  HttpServletRequest request) throws Exception {

 // FlashMap에 저장된 message 값을 꺼내옴
 Map<String, ?> fm = RequestContextUtils.getInputFlashMap(request);
 if (fm != null) {
  String message = (String) fm.get("message");
  System.out.println(message);
 }
}
```

6.Multiple Representation

앞 장에서 설명했듯이, RESTful 아키텍처에서 하나의 Resource는 여러 형태의 Represenation을 가질 수 있다. 즉, 클라이언트가 서버에 생성하거나 수정하기 위해 전달하는 데이터의 형태도 다양할 수 있고, 서버가 클라이언트의 요청을 처리하고 전달하는 응답도 다양한 형태를 가질 수 있다. 이러한 Content Negotiation을 지원하기 위해서 Spring에서 제공하는 기능에 대해서 살펴보도록 하자.

기존의 웹 어플리케이션은 웹 페이지에서 form submit을 통해 저장 또는 수정하고자 하는 데이터들이 전달되었다. 그렇게 submit된 데이터는 아래와 같은 모습의 HTTP Request message로 서버에 들어온다.



일반적인 웹 어플리케이션에서 Controller의 메소드는 위 HTTP message body 부분의 정보를 Command 객체에 바인딩해서 사용한다.

```
@RequestMapping(value = "/{movieId}", method = RequestMethod.POST)
public void update(Movie updateMovie) throws Exception {
 this.movieService.update(updateMovie);
}
```

그러나 RESTful 웹 서비스로 노출하는 메소드는 아래의 그림 처럼 xml, json 등 다양한 형식으로 요청 데이터가 들어올 수 있다.

그래서 Spring에서는 다양한 형태의 HTTP Request Message를 직접 처리할 수 있도록 **@RequestBody**를 제공한다. 또한, 클라이언트로 다양한 형태의 HTTP Response Message를 직접 리턴할 수 있도록 **@ResponseBody**를 제공한다.

```
@RequestMapping(value = "/{movieId}", method = RequestMethod.PUT)
@ResponseBody
public void update(@RequestBody Movie updateMovie) throws Exception {
 this.movieService.update(updateMovie);
}
```

@RequestBody와 @ResponseBody가 각각 Request/Reponse message를 처리할 때, **message와 Java 객체간의 변환은 HttpMessageConverter [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html]**가 담당한다. Spring에서는 미디어 타입(예: html, xml, json 등)에 따라 Jaxb2RootElementHttpMessageConverter, StringHttpMessageConverter, MappingJacksonHttpMessageConverter 등 여러가지

HttpMessageConverter 구현체를 제공하고 있다. 자세한 내용은 본 매뉴얼 HTTP Message Conversion을 참조하기 바란다. @RequestBody를 적용하여 **Request message 처리시 Content-Type header 값에 따라** 적절한 HttpMessageConverter가 사용된다. 마찬가지로, @ResponseBody를 사용하여 **Response message 생성시 Request로 들어온 Accept header 값에 따라** 적절한 HttpMessageConverter가 사용된다.

클라이언트로 전달할 Response를 좀 더 상세하게 구성하고자 하는 경우에는 **ResponseEntity<?>**를 사용할 수 있다.

```
@RequestMapping(method = RequestMethod.POST)
public ResponseEntity<String> create(@RequestBody Movie movie) throws Exception {

    this.movieService.create(movie);

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("Location", "http://localhost:8080/mypjt2/movies/" +
 movie.getMovieId());

    // 201 CREATED, Location header
    return new ResponseEntity<String>("Created resource " + movie.getMovieId(),
 responseHeaders, HttpStatus.CREATED);
}
```

위 코드 예제에서는 @ResponseBody 대신 ResponseEntity를 사용해서 Location header와 '201 CREATED'라는 status code로 Response를 구성했다. 일반적으로 POST method의 경우 요청에 의해 Resource가 생성되었다면 '201 CREATED' status code와 새로 생성된 Resource를 조회할 수 있는 정보를 기술한 Location header를 리턴한다.

HTTP Response Status Code

HTTP Methods	GET	POST	PUT	DELETE
요청처리를 성공했을 때의 Status Code	'200 OK' + 결과로 돌려줄 내용	'201 Created' + Location 헤더 (새로 생성된 Resource의 정보)	리턴 값이 있으면 '200 OK', 없으면 '204 No Content'	리턴 값이 있으면 '200 OK', 없으면 '204 No Content'

자세한 내용은 HTTP Status Code Definitions [http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html]와 Method Definitions [http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html]를 참조하기 바란다.

@ResponseBody나 ResponseEntity를 사용하지 않고 기존의 웹 어플리케이션에서 처럼 View의 이름을 리턴하는 경우에도 Content Negotiation이 가능하도록 Spring에서는 **ContentNegotiatingViewResolver**를 제공한다.

ContentNegotiatingViewResolver는 자기가 직접 View를 구성하는 것이 아니라, 등록된 다른 모든 View Resolver에게로 View를 찾는 것을 위임한다. 다른 View Resolver들이 리턴한 View의 Content-Type과 HTTP Request의 Accept 헤더 값 또는 파일 확장자로 기술된 미디어 타입(Content-Type값)을 비교하여 클라이언트가 요청한 Content-Type에 가장 적합한 View를 선택하여 응답을 돌려준다.

이렇듯 ContentNegotiatingViewResolver는 다른 View Resolver들과 반드시 함께 사용되어야 하므로 View Resolver 설정 시 반드시 order를 정의해야 한다. 당연히 ContentNegotiatingViewResolver가 가장 높은 우선순위(가장 작은숫자)를 가져야 한다.

파일 확장자 기반의 Content Negotiation을 처리하기 위해서는 ContentNegotiatingViewResolver의 **mediaTypes** 속성에 파일 확장자와 미디어 타입을 매핑시켜 정의한다.

다음은 ContentNegotiatingViewResolver를 사용하기 위한 설정 예이다.

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
```

```
    <property name="mediaTypes">
        <map>
            <entry key="html" value="text/html"/>
            <entry key="xml" value="application/xml" />
        </map>
    </property>
    <property name="order" value="0"/>
</bean>
```

ContentNegotiatingViewResolver는 기본적으로 WebApplicationContext에 등록된 View Resolver들을 자동으로 찾아서 처리하지만, 아래와 같이 **viewResolvers** 속성을 이용해서 다른 View Resolver들을 명시적으로 지정할 수도 있다. 또한, **defaultViews** 속성에 View를 명시해두면, View Resolver 체인에서 클라이언트가 요청한 Content-Type을 지원하는 View를 찾지 못한 경우에 디폴트 View로 사용된다.

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="html" value="text/html"/>
            <entry key="atom" value="application/atom+xml"/>
            <entry key="json" value="application/json"/>
        </map>
    </property>

     <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
    </property>

    <property name="defaultViews">
        <list>
            <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>
        </list>
    </property>
</bean>
```

요청 처리후 돌려줄 적절한 View를 선택하기 위해서, ContentNegotiatingViewResolver는 클라이언트로부터 요청된 미디어 타입을 가지고 매칭시키는데, 이 미디어 타입을 알아내는 작업은 다음과 같은 과정으로 이루어진다.

1. favorPathExtension 속성 값이 true(디폴트값이 true이다)이고, Request path에 파일 확장자가 포함되어 있다면, ContentNegotiatingViewResolver의 mediaTypes 속성에 정의된 매핑 정보를 사용한다. 적절한 미디어 타입을 찾지 못했을 때, 만약 Java Activation Framework가 classpath에 존재한다면, FileTypeMap.getContentType(String filename) 메소드의 리턴 값을 미디어타입으로 사용한다.

2. favorParameter 속성 값이 true(디폴트값은 false이다)이고, Request에 미디어 타입을 정의하는 파라미터가 포함되어 있다면, ContentNegotiatingViewResolver의 mediaTypes 속성에 정의된 매핑 정보를 사용한다. 디폴트 파라미터 명은 'format'이고 이것은 parameterName이라는 속성으로 변경가능하다.

3. 위의 과정으로도 미디어 타입을 찾지 못했을 때, ContentNegotiatingViewResolver의 ignoreAcceptHeader가 false로 지정되어 있으면 Request의 Accept 헤더 값을 사용한다.

4. 위의 모든 과정을 거치고도 미디어 타입을 찾지 못한 경우, 최종적으로 ContentNegotiatingViewResolver의 defaultContentType이 정의되어 있다면 그 값을 클라이언트에서 요청한 미디어 타입으로 간주한다.

일단 클라이언트가 요청한 미디어 타입을 찾아내면 다른 View Resolver들에게 View를 요청하고, View Resolver들이 리턴한 View의 Content-Type과 요청들어온 미디어 타입의 매칭여부를 확인해서 가장 적합한 View를 찾아서 클라이언트로 응답한다.

Response Status Code와 에러 페이지

Controller의 메소드에서 Exception이 발생했을 때, Response를 직접 리턴하는 경우에는 메소드 내부에서 Exception을 catch 하고, Response에 Error Status Code를 설정하여 리턴을 하면 된다. 그러나 Response를 직접 리턴하지 않는 경우에는 발생한 Exception을 exceptionResolver가 처리하도록 되어 있다.

현재 Anyframe의 core plugin에 의해 SimpleMappingExceptionResolver가 설정되고 defaultErrorView로 error.jsp가 렌더링되어 에러가 발생했음에도 불구하고 REST 클라이언트에게는 '200 OK'라는 Response Status와 함께 error 페이지 HTML 내용이 리턴되는 문제가 있다.

SimpleMediaTypeViewResolver

위에서 설명한 대로 ContentNegotiatingViewResolver는 주어진 요청에 "가장 가까운" View를 찾아서 반환하도록 설계되었다. 하지만 이 때문에 설정이 복잡하고 Spring의 ViewResolve 로직을 이해하여야만 하는 직관적이지 못한 설정방법을 제공하고 있다.

이에 Anyframe에서는 Core 플러그인 1.0.4 버전부터 "Negotiation"은 제외하고 요청받은 View를 직관적으로 서비스할 수 있는 간략화된 형태의 ViewResolver를 별도로 제공하고 있다. SimpleMediaTypeViewResolver이다.

다음은 SimpleMediaTypeViewResolver를 사용할 경우의 설정 예이다.

```
<bean id="jstlViewResolver"
 class="org.springframework.web.servlet.view.UrlBasedViewResolver">
 <property name="viewClass"
 value="org.springframework.web.servlet.view.JstlView" />
 <property name="prefix" value="/WEB-INF/jsp/"/>
 <property name="suffix" value=".jsp"/>
</bean>

<bean class="org.anyframe.spring.web.servlet.view.SimpleMediaTypeViewResolver">
 <property name="mediaTypeMappings">
  <map>
   <entry key="html" value-ref="jstlViewResolver"/>
   <entry key="xml" value-ref="xmlView"/>
   <entry key="xmls" value-ref="xmlListView"/>
   <entry key="json" value-ref="jsonView"/>
  </map>
 </property>
 <property name="defaultMapping" ref="jstlViewResolver"/>
 <property name="order" value="0"/>
</bean>

<bean name="xmlView"
 class="org.springframework.web.servlet.view.xml.MarshallingView">
 <property name="marshaller" ref="marshaller" />
</bean>

<bean name="xmlListView"
 class="org.springframework.web.servlet.view.xml.MarshallingView">
 <property name="modelKey" value="resultPage" />
 <property name="marshaller" ref="marshaller" />
</bean>
```

```
<bean name="jsonView"
 class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>

<bean
 class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver">
 <property name="order" value="0" />
</bean>

<oxm:jaxb2-marshaller id="marshaller">
 <oxm:class-to-be-bound
 name="org.anyframe.springrest.example.springrest.domain.Movie" />
 <oxm:class-to-be-bound name="org.anyframe.pagination.Page" />
</oxm:jaxb2-marshaller>

<bean id="castorMarshaller"
 class="org.springframework.oxm.castor.CastorMarshaller" />
```

SimpleMediaTypeViewResolver는 요청받은 request uri의 media type(=extension)을 서비스할 ViewResolver 또는 View를 직접 지정하는 mapping 기반의 설정 방식을 통해 직관적이고 비교적 간단한 설정으로 Multiple Representation을 제공하도록 설계되었다.

위에서 언급한 SimpleMediaTypeViewResolver에 관한 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-springrest [http://dev.anyframejava.org/docs/anyframe/plugin/optional/springrest/1.1.0/reference/sample/anyframe-sample-springrest.zip]를 통해 다운로드받을 수 있다.

7.Views

Spring 3부터 Spring MVC에는 웹 어플리케이션에서 하나의 리소스, 즉 하나의 서비스에 대한 여러 형태의 응답을 지원하기 위해 다음과 같은 새로운 View들이 추가되었다..

- **AbstractAtomFeedView / AbstractRssFeedView : Atom이나 RSS 피드를 보여줄 수 있는 View**

  AbstractAtomFeedView와 AbstractRssFeedView는 AbstractFeedView의 하위클래스로 java.net의 ROME [https://rome.dev.java.net] 프로젝트를 기반으로 만들어져있다. Feed View를 구성하려면 AbstractAtomFeedView나 AbstractRssFeedView를 상속받은 클래스에서 각각에서 오버라이드 요구하는 메소드를 구현하여 사용한다.

```
public class SampleContentAtomView extends AbstractAtomFeedView {
    @Override
    protected List<Entry> buildFeedEntries(Map<String, Object> model,
                                           HttpServletRequest request,
 HttpServletResponse response) throws Exception {
        // 중략
    }
}
```

```
public class SampleContentRssView extends AbstractRssFeedView {
    @Override
    protected List<Item> buildFeedItems(Map<String, Object> model,
                                        HttpServletRequest request,
 HttpServletResponse response) throws Exception {
        // 중략
    }
}
```

  구현한 Feed View를 사용하기 위해서 Bean 정의 파일을 작성해야한다.

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="atom" value="application/atom+xml"/>
            <entry key="html" value="text/html"/>
        </map>
    </property>
    <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean
 class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
    </property>
</bean>

<bean id="movies" class="anyframe.sample.moviefinder.feed.MoviesAtomView"/>
```

- **MarshallingView : XML로 응답을 전달할 수 있는 View**

  MarshallingView는 클라이언트에게 XML 응답을 돌려주기 위해서 Spring OXM의 Marshaller를 사용한다. 기본적으로 컨트롤러가 리턴한 모든 Model을 XML로 변환하지만, **modelKey**라는 속성에 Model의 이름을 지정함으로써 Marshalling되어 클라이언트로 전달될 Model을 필터링 할 수 있다.

다음은 springrest plugin 설치로 추가된 springrest-servlet.xml의 일부이다. 먼저 설치된 core plugin에서 정의한 View Resolver들이 있으므로 MarshallingView를 위해서 BeanNameViewResolver만 추가하였다.

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="html" value="text/html" />
            <entry key="xml" value="application/xml" />
        </map>
    </property>
    <property name="order" value="0" />
</bean>

<bean id="springrestViewMovie"
 class="org.springframework.web.servlet.view.xml.MarshallingView">
    <property name="marshaller" ref="marshaller" />
</bean>

<bean id="springrestListMovie"
 class="org.springframework.web.servlet.view.xml.MarshallingView">
    <property name="marshaller" ref="marshaller" />
</bean>

<bean id="error" class="test003.springrest.moviefinder.web.view.MarshallingViewForError">
    <property name="marshaller" ref="marshaller" />
</bean>

<oxm:jaxb2-marshaller id="marshaller">
    <oxm:class-to-be-bound name="test003.springrest.domain.Movie" />
    <oxm:class-to-be-bound name="test003.springrest.domain.ResultPage" />
    <oxm:class-to-be-bound name="org.anyframe.pagination.Page" />
</oxm:jaxb2-marshaller>
```

- **MappingJacksonJsonView : JSON으로 응답을 전달할 수 있는 View**

  MappingJacksonJsonView는 클라이언트에게 JSON 응답을 돌려주기 위해서 Jackson 라이브러리의 ObjectMapper를 사용한다. 디폴트로 Model 객체 모두의 내용을 JSON으로 보내도록 되어있지만, **renderedAttributes** 속성을 이용해서 JSON으로 변환할 Model을 필터링 할 수 있다. ObjectMapper 확장이 필요한 경우 **objectMapper** 속성을 이용해서 확장한 ObjectMapper를 정의해 준다.

  MappingJacksonJsonView는 Anyframe의 Simpleweb Plugin에 적용되어 있으므로, 사용 예는 매뉴얼 Simpleweb Plugin의 JSON View 설정 [http://dev.anyframejava.org/docs/anyframe/plugin/optional/simpleweb/1.1.0/reference/htmlsingle/simpleweb.html#simpleweb_configuration_json]을 참조하기 바란다.

8.Exception Handling

Spring MVC에서는 컨트롤러의 메소드에서 특정 Exception이 발생했을 경우에 해당 Exception을 처리할 수 있도록 **HandlerExceptionResolver** 인터페이스를 제공한다. HandlerExceptionResolver 인터페이스의 resolveException(Exception, Handler) 메소드를 구현하여 DispatcherServlet에 등록하면 해당 Exception이 발생했을 때 구현한 메소드가 호출된다. 기존에 web.xml에서 <error-page>를 이용해서 에러를 보여주는 페이지를 정의하는 방식과 비슷하지만 좀 더 유연한 기능들을 제공한다.

Spring MVC에서는 몇가지 HandlerExceptionResolver 구현체를 제공하고 있다. core plugin 샘플에서 볼 수 있는 **SimpleMappingExceptionResolver**가 그 중 하나이다. SimpleMappingExceptionResolver의 exceptionMappings속성에 Exception 클래스와 해당 Exception이 발생했을 때 보여줄 View를 매핑하면 된다. core plugin에서는 모든 Exception에 대해서 error라는 이름의 View로 화면을 렌더링하도록 설정되어있다.

그 밖에 Exception 처리 방법에는 어떤 것들이 있는지 알아보자.

8.1.@ExceptionHandler

HandlerExceptionResolver 인터페이스를 직접 구현하지 않고 @ExceptionHandler를 이용할 수도 있다. 컨트롤러 메소드에 @ExceptionHandler를 붙이면 지정한 Exception이 발생했을 때 해당 예외를 처리하게 할 수 있다.

```
@Controller
@RequestMapping("/movies")
public class MovieController {
    // ...

    @ExceptionHandler(NotFoundException.class)
    public void handleNotFoundException(NotFoundException ex) {
        // ...
    }
}
```

위 컨트롤러에서 NotFoundException이 발생하면 handleNotFoundException() 메소드가 호출될 것이다.

다음과 같이 적절한 Response Status Code를 전달하기 위해서 @ResponseStatus와 함께 사용할 수도 있다.

```
@Controller
@RequestMapping("/movies")
public class MovieController {
    // ...

    @ExceptionHandler(NotFoundException.class)
    @ResponseStatus(value=HttpStatus.NOT_FOUND)
    public void handleNotFoundException(NotFoundException ex) {
        // ...
    }
}
```

위의 Annotation을 이용한 Exception 핸들링이 정상적으로 동작하기 위해서는, 반드시 org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver가 등록되어 있어야만 한다.

8.2.@ResponseStatus

@ResponseStatus를 사용하면 컨트롤러 메소드나 Exception 클래스가 Status Code를 리턴하도록 정의할 수 있다.

```
@ResponseStatus(value=HttpStatus.NOT_FOUND)
public class NotFoundException extends BaseException {
        // ...
}
```

위의 같이 정의한 경우, NotFoundException이 발생하면 클라이언트로 '404 Not Found' Status Code가 전달된다.

8.3.DefaultHandlerExceptionResolver

DispatcherServlet이 디폴트로 등록하는 HandlerExceptionResolver로 **DefaultHandlerExceptionResolver [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/mvc/support/DefaultHandlerExceptionResolver.html]**가 있다. DefaultHandlerExceptionResolver는 Spring에서 내부적으로 발생하는 주요 Exception들을 적절한 Response Status Code로 전환해 준다.

예를 들면, Request로 들어온 데이터를 처리하다가 타입이 맞지 않으면 TypeMismatchException이 발생하는데, 이것을 '400 Bad Request' Status Code로 리턴한다. DefaultHandlerExceptionResolver는 디폴트로 등록되지만 다른 HandlerExceptionResolver를 등록할 경우에는 명시적으로 등록하는 것이 좋다.

Exception	HTTP Status Code
ConversionNotSupportedException	500 (Internal Server Error)
HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
HttpMessageNotReadableException	400 (Bad Request)
HttpMessageNotWritableException	500 (Internal Server Error)
HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
MissingServletRequestParameterException	400 (Bad Request)
NoSuchRequestHandlingMethodException	404 (Not Found)
TypeMismatchException	400 (Bad Request)

Response Status Code와 HandlerExceptionResolver

기존의 웹 어플리케이션에서는 Error가 발생했을 때 에러의 정보를 보여주는 페이지로 이동하였다. 이 경우 REST 클라이언트에서는 에러가 발생했음에도 불구하고 '200 OK'라는 Response Status와 함께 error 페이지 HTML 내용이 리턴되는 문제가 있다. 반대로 REST 클라이언트에게 Error Status Code를 리턴하도록 설정을 바꿀 경우 기존 웹 어플리케이션에서는 에러를 위한 페이지를 사용할 수 없다.

이와 같은 경우 ExceptionHandlerExceptionResolver를 컨테이너에 등록하고, 다음과 같이 @ExceptionHandler와 @ResponseStatus를 이용하여 REST 클라이언트와 웹어플리케이션을 동시에 만족시킬 수 있다.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
@ExceptionHandler(NotFoundException.class)
public ModelAndView handleException(NotFoundException ex) {
```

```
    ModelMap model = new ModelMap();
    model.addAttribute("class", ClassUtils.getShortName(ex.getClass()));
    model.addAttribute("message", ex.getMessage());
    model.addAttribute("exception", ex);

    return new ModelAndView("error", model);
}
```

9.HTTP Method Conversion

앞서 설명했듯이, REST 아키텍처에서는 HTTP에서 정의하고 있는 모든 method를 사용할 것을 권장하고 있지만, 브라우저 기반의 HTML에서는 이 중 단 2가지, GET과 POST만을 지원한다. JavaScript를 이용해서 PUT과 DELETE를 사용할 수도 있겠지만 번거로운 코딩 작업이 추가되어야 하기 때문에, 일반적으로 HTML에는 POST를 사용하고 실제 HTTP Method를 지정하는 hidden 타입의 입력 필드를 추가해서 사용하는 경우가 많다.

Spring 3에서는 **HiddenHttpMethodFilter**를 제공하여 **실제 HTTP Method를 지정하는 hidden 타입의 입력 파라미터를 찾아내서 HTTP Method를 변환**하는 작업을 지원해준다. **web.xml에 HiddenHttpMethodFilter 설정을 추가하면, HTTP Method가 POST이고 _method라는 파라미터가 존재하는 경우 HTTP의 Method를 _method 값으로 바꾼다.** '_method'가 아닌 다른 파라미터명을 사용하려면 methodParam 속성을 이용해서 지정해준다.

또한 Spring에서는 <form:form>에서 실제 HTTP Method를 지정하는 hidden 타입의 입력 필드를 자동으로 추가해주기 때문에 훨씬 더 편리하게 사용할 수 있다.

```
<form:form method="delete">
    <input type="submit" value="Delete Movie"/>
</form:form>
```

JSP에 위와 같이 작성하면, 내부적으로는 POST 방식으로 "_method=delete"가 전달되는 것이다.

HiddenHttpMethodFilter 사용 시 유의 사항

HiddenHttpMethodFilter를 사용할 때 한가지 주의할 점은, 파일 업로드를 위해 form의 enctype 속성을 'multipart/form-data'로 지정하는 경우 HiddenHttpMethodFilter가 정상적으로 동작하기 않기 때문에 기존에 파일 업로드를 위해서 사용했던 MultipartResolver 설정 방식을 변경해야 한다는 것이다.

web.xml에다가 MultipartFilter [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/multipart/support/MultipartFilter.html]를 HiddenHttpMethodFilter 앞에 정의하고, MultipartResolver를 Spring의 root Application Context에 'filterMultipartResolver'라는 Bean 이름으로 설정해 주어야 HiddenHttpMethodFilter가 정상적으로 동작할 수 있다.

다음은 web.xml에 MultipartFilter와 HiddenHttpMethodFilter를 정의한 모습이다.

```
<filter>
    <filter-name>multipartFilter</filter-name>
    <filter-class>org.springframework.web.multipart.support.MultipartFilter</
filter-class>
</filter>
<filter-mapping>
    <filter-name>multipartFilter</filter-name>
    <url-pattern>/springrest/*</url-pattern>
</filter-mapping>
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</
filter-class>
</filter>
<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <url-pattern>/springrest/*</url-pattern>
</filter-mapping>
```

다음은 context-springrest-multipart.xml에 정의한 MultipartResolver 설정이다.

```
<bean id="filterMultipartResolver"
 class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="maxUploadSize">
        <value>10000000</value>
    </property>
</bean>
```

MultipartResolver Bean을 'filterMultipartResolver'가 아닌 다른 이름으로 정의할 경우, web.xml에서 MultipartFilter 정의 시에 <init-param>을 이용해서 'multipartResolverBeanName'을 설정해준다.

10.Implementing REST Client

지금까지 위에서 설명한 내용들은 모두 서버측 구현과 관련된 내용이었다. RestTemplate/AsyncRestTemplate은 REST 아키텍처에서 클라이언트 구현과 관련된 내용이다.

RestTemplate/AsyncRestTemplate은 Spring에서 제공하고 있는 JdbcTemplate이나, JmsTemplate과 같은 맥락의 Template으로, RESTful Service 호출과 관련된 여러 메소드를 제공하여 REST 클라이언트를 쉽게 개발할 수 있도록 도와주는 것이다. **RestTemplate/AsyncRestTemplate에서 Java 객체를 HTTP Request로 변환하거나 서버로 부터 전달된 HTTP Response를 다시 Java 객체로 변환할 때 HttpMessageConverter가 사용된다.** Spring에서 제공하는 주요 타입에 대한 HttpMessageConverter들은 RestTemplate에 디폴트로 등록된다. 그 외에 추가가 필요한 경우, RestTemplate/AsyncRestTemplate을 정의할 때 **messageConverters**라는 속성을 이용한다.

10.1.Configuration

RestTemplate/AsyncRestTemplate 역시 Spring 컨테이너에 Bean으로 정의하고, 참조할 클래스에서 Injection 받아 사용한다. 다음은 springrest plugin의 src/test/resources/context-restclient.xml파일의 일부이다.

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate" />
```

```
<bean id="asyncRestTemplate" class="org.springframework.web.client.AsyncRestTemplate" />
```

RestTemplate/AsyncRestTemplate에서도 HTTP Request 메세지를 구성하거나, Response 메세지를 파싱할 때 HttpMessageConverter를 사용한다. 디폴트로 등록된 HttpMessageConverter를 변경하거나 새로운 HttpMessageConverter를 추가하려면 **messageConverters** 속성을 사용한다.

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
    <property name="messageConverters">
        <list>
            <bean class="my.custom.MarshallingHttpMessageConverter">
                <property name="unmarshaller" ref="marshaller" />
                <property name="marshaller" ref="marshaller"/>
            </bean>
        </list>
    </property>
</bean><bean id="asyncRestTemplate"
 class="org.springframework.web.client.AsyncRestTemplate">
    <property name="messageConverters">
        <list>
            <bean class="my.custom.MarshallingHttpMessageConverter">
                <property name="unmarshaller" ref="marshaller" />
                <property name="marshaller" ref="marshaller"/>
            </bean>
        </list>
    </property>
</bean>
```

10.2.RestTemplate

RestTemplate은 GET, POST 등 모든 HTTP Method를 사용하여 쉽고 간편하게 RESTful 웹 서비스를 호출할 수 있도록 다음과 같은 메소드를 제공하고 있다.

HTTP	Method
DELETE	delete(java.lang.String, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#delete(java.lang.String, java.lang.Object...)]
GET	getForObject(java.lang.String, java.lang.Class, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#getForObject(java.lang.String, java.lang.Class, java.lang.Object...)]
	getForEntity(java.lang.String, java.lang.Class, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#getForEntity(java.lang.String, java.lang.Class, java.lang.Object...)]
HEAD	headForHeaders(java.lang.String, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#headForHeaders(String, String...)]
OPTIONS	optionsForAllow(java.lang.String, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#optionsForAllow(String, String...)]
POST	postForLocation(java.lang.String, java.lang.Object, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#postForLocation(String, Object, String...)]
	postForObject(java.lang.String, java.lang.Object, java.lang.Class, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#postForObject(java.lang.String, java.lang.Object, java.lang.Class, java.lang.Object...)]
PUT	put(java.lang.String, java.lang.Object, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#put(String, Object, String...)]
any	exchange(java.lang.String, org.springframework.http.HttpMethod, org.springframework.http.HttpEntity, java.lang.Class, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#exchange(java.lang.String, org.springframework.http.HttpMethod, org.springframework.http.HttpEntity, java.lang.Class, java.lang.Object...)]
	execute(java.lang.String, org.springframework.http.HttpMethod, org.springframework.web.client.RequestCallback, org.springframework.web.client.ResponseExtractor, java.lang.Object...) [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/client/RestTemplate.html#execute(java.lang.String, org.springframework.http.HttpMethod,

HTTP	Method
	org.springframework.web.client.RequestCallback, org.springframework.web.client.ResponseExtractor, java.lang.Object...)]

RestTemplate에서 제공하는 getForObject() 메소드를 사용하면 서버로부터 어떤 리소스를 조회하는 기능을 구현할 수 있고, postForLocation() 메소드를 사용하면 서버 측에 리소스를 생성하거나 수정하는 기능을 구현할 수 있다.

다음은 springrest plugin의 TestCase에서 RestTemplate을 사용한 예이다. 코드에서 볼 수 있듯이 RESTful 서비스를 호출하여 결과를 받아오는 것이 한 줄의 코드로 처리가 가능하다.

```
@Inject
@Named("restTemplate")
private RestTemplate restTemplate;

@Test
public void findMovie() {
    String movieId = "MV-00005";
    String movieSearchUrl = "http://localhost:8080/mypjt2/movies/{movieId}";

    Movie movie = restTemplate.getForObject(movieSearchUrl, Movie.class,
        movieId);

    assertThat(movie.getMovieId(), is(movieId));
}
```

RestTemplate의 메소드들은 모두 URI Template을 사용하여 요청 URI를 명시할 수 있다.

```
String result = restTemplate.getForObject("http://localhost:8080/testrest/springrest/movies/
{movieId}/edit.xml", Movie.class, "MV-00005");
```

아래와 같이 URI Template의 변수를 Map으로 처리할 수도 있다.

```
Map<String, String> vars = new HashMap<String, String>();
vars.put("movieId", "MV-00005");
String result = restTemplate.getForObject("http://localhost:8080/testrest/springrest/movies/
{movieId}/edit.xml", Movie.class, vars);
```

또한, exchange 메소드를 이용하여 HTTP Response의 header와 body 정보를 자유롭게 사용할 수도 있다.

```
@Test
public void createMovie() throws Exception {
    String movieCreateUrl = "http://localhost:8080/mypjt2/movies";

    Movie movie = makeMovie();

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_ATOM_XML);
    HttpEntity<Movie> requestEntity = new HttpEntity<Movie>(movie, headers);

    ResponseEntity<String> response = restTemplate.exchange(movieCreateUrl,
        HttpMethod.POST, requestEntity, String.class);
    assertThat(response.getStatusCode().toString(), is("201"));

    String movieSearchUrl = response.getHeaders().getLocation().toURL()
        .toString();
    movie = restTemplate.getForObject(movieSearchUrl, Movie.class);

    assertThat(movie, notNullValue());
```

```
    assertThat(movieSearchUrl,
        is("http://localhost:8080/mypjt2/movies/"
            + movie.getMovieId()));
    assertThat(movie.getTitle(), is("괴물"));

    System.out.println("New movie is registered.");
    System.out.println("1.MOVIE ID : " + movieSearchUrl);
    System.out.println("2.MOVIE Object : " + movie);
}
```

Error Status Code

RestTemplate으로 RESTful 웹 서비스를 호출했을 때, 서버로부터 404나 500 등의 Error Status Code를 리턴을 받게되면 Exception이 발생한다.

10.3.AsyncRestTemplate

AsyncRestTemplate 도 RestTemplate과 마찬가지로 GET, POST 등 모든 HTTP Method를 사용하여 쉽고 간편하게 RESTful 웹 서비스를 async 호출할 수 있도록 다음과 같은 메소드를 제공하고 있다. RestTemplate 과 유사한 Method들이 존재하나, RestTemplate과는 다르게 AsyncRestTemplate 은 ListenableFuture 형식을 return 한다.

HTTP	Method
DELETE	delete(java.lang.String, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#delete-java.lang.String-java.lang.Object...-]
GET	getForEntity(java.lang.String, java.lang.Class, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#getForEntity-java.lang.String-java.lang.Class-java.lang.Object...-]
HEAD	headForHeaders(java.lang.String, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#headForHeaders-java.lang.String-java.lang.Object...-]
OPTIONS	optionsForAllow(java.lang.String, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#optionsForAllow-java.lang.String-java.lang.Object...-]
POST	postForLocation(java.lang.String, java.lang.Object, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#postForLocation-java.lang.String-org.springframework.http.HttpEntity-java.lang.Object...-]
	postForEntity(java.lang.String, java.lang.Object, java.lang.Class, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#postForEntity-java.lang.String-org.springframework.http.HttpEntity-java.lang.Class-java.lang.Object...-]
PUT	put(java.lang.String, java.lang.Object, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/

HTTP	Method
	springframework/web/client/AsyncRestTemplate.html#put-java.lang.String-org.springframework.http.HttpEntity-java.lang.Object...-]
any	exchange(java.lang.String, org.springframework.http.HttpMethod, org.springframework.http.HttpEntity, java.lang.Class, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#exchange-java.lang.String-org.springframework.http.HttpMethod-org.springframework.http.HttpEntity-java.lang.Class-java.lang.Object...-]
	execute(java.lang.String, org.springframework.http.HttpMethod, org.springframework.web.client.RequestCallback, org.springframework.web.client.ResponseExtractor, java.lang.Object...) [http://docs.spring.io/spring/docs/4.0.0.RELEASE/javadoc-api/org/springframework/web/client/AsyncRestTemplate.html#execute-java.lang.String-org.springframework.http.HttpMethod-org.springframework.web.client.AsyncRequestCallback-org.springframework.web.client.ResponseExtractor-java.lang.Object...-]

AsyncRestTemplate에서 제공하는 getForEntity() 메소드를 사용하면 서버로부터 어떤 리소스를 조회하는 기능을 구현할 수 있고, postForLocation() 메소드를 사용하면 서버 측에 리소스를 생성하거나 수정하는 기능을 구현할 수 있다.

다음은 springrest plugin의 TestCase에서 AsyncRestTemplate을 사용한 예이다. 코드에서 볼 수 있듯이 RESTful 서비스를 async호출이 가능하여, 결과를 받아올 때는 async 호출 이후에 ListenableFuture.get() 이나 ListenableFuture.addCallback(...) 을 사용한다.

```
@Inject
 @Named("asyncRestTemplate")
 private AsyncRestTemplate asyncRestTemplate;

public void getMovie(String movieId) throws Exception {

  String movieSearchUrl = "http://localhost:8080/anyframe-sample-springrest/springrest/
moviesasync/{movieId}";

  ListenableFuture<ResponseEntity<Movie>> futureEntity =
 asyncRestTemplate.getForEntity(movieSearchUrl, Movie.class, movieId);

  ResponseEntity<Movie> movie = futureEntity.get();
  System.out.println("--> movie id : " + movie.getBody().getMovieId());
  System.out.println("--> movie title : " + movie.getBody().getTitle());

 }
```

AsyncRestTemplate의 메소드들은 모두 URI Template을 사용하여 요청 URI를 명시할 수 있다.

```
String result = asyncRestTemplate.getForEntity("http://localhost:8080/testrest/springrest/
movies/{movieId}/edit.xml", Movie.class, "MV-00005");
```

아래와 같이 URI Template의 변수를 Map으로 처리할 수도 있다.

```
Map<String, String> vars = new HashMap<String, String>();
vars.put("movieId", "MV-00005");
String result = asyncRestTemplate.getForEntity("http://localhost:8080/testrest/springrest/
movies/{movieId}/edit.xml", Movie.class, vars);
```

또한, exchange 메소드를 이용하여 HTTP Response의 header와 body 정보를 자유롭게 사용할 수도 있다.

```
    public String createMovie(Movie movie) throws Exception {
  String movieCreateUrl = "http://localhost:8080/anyframe-sample-springrest/springrest/
moviesasync";

  HttpHeaders headers = new HttpHeaders();
  headers.setContentType(MediaType.APPLICATION_ATOM_XML);
  HttpEntity<Movie> requestEntity = new HttpEntity<Movie>(movie, headers);

  ListenableFuture<ResponseEntity<String>> futureEntity =
    asyncRestTemplate.exchange(movieCreateUrl, HttpMethod.POST, requestEntity,
 String.class);

  futureEntity.addCallback(new ListenableFutureCallback<ResponseEntity<String>>() {

   public void onFailure(Throwable arg0) {
    System.out.println("--> Fail!!!");
   }

   public void onSuccess(ResponseEntity<String> arg0) {
    System.out.println("--> Success!!!");

    String movieSearchUrl = "";
    try {
     movieSearchUrl = arg0.getHeaders().getLocation().toURL().toString();
    } catch (MalformedURLException e) {
     e.printStackTrace();
    }
    System.out.println("result movieSearchUrl : " + movieSearchUrl);
   }
  });

  return "";
 }
```

asyncRestTemplate 를 사용한 async 호출시 유의사항

asyncRestTemplate 를 사용하여 async 호출을 할 경우 ListenableFuture 타입을 return 한다고 위에서 명시했다. 결과값을 얻기 위해서는 ListenableFuture.get() 이나 ListenableFuture.addCallback(...) 응 사용하는데, 해당 Method 를 사용하는 경우 기존의 javaScript 에서 서비스를 async 호출하는 것과 같이 별도의 Thread가 생성되어 종료시까지 모니터링하는 형식이 아니고, get이나 addCallback 을 한 시점에서 server 쪽의 작업이 종료되길 기다렸다가 결과값을 받아오게 된다. 이점을 유의하여, asyncRestTemplate을 통한 async 호출은 한꺼번에 여러가지 일을 Server 쪽에 요청한 후 이후에 한꺼번에 결과값을 받아오는 것이 가능한 부분에 사용하는것이 적합할 것으로 보인다.

Error Status Code

AsyncRestTemplate으로 RESTful 웹 서비스를 호출했을 때, 서버로부터 404나 500 등의 Error Status Code를 리턴을 받게되면 Exception이 발생한다.

11.HTTP Message Conversion

RestTemplate이나 @Controller에서 Java 객체를 HTTP Request로 변환하거나 서버로 부터 전달된 HTTP Response를 다시 Java 객체로 변환할 때 HttpMessageConverter가 사용된다. Spring에서 제공하는 주요 타입에 대한 HttpMessageConverter들은 RestTemplate(클라이언트측)과 AnnotationMethodHandlerAdapter(서버측)에 디폴트로 등록되어 내부적으로 변환에 사용된다.

HttpMessageConverter 인터페이스는 아래와 같은 모습이다. 정의된 메소드들을 보면 제공하는 기능을 알 수 있다.

```
public interface HttpMessageConverter<T> {
    // 입력된 클래스와 미디어 타입이 이 HttpMessageConverter에서 Read 가능한지 여부를 확인.
    boolean canRead(Class<?> clazz, MediaType mediaType);

    // 입력된 클래스와 미디어 타입이 이 HttpMessageConverter에서 Write 가능한지 여부를 확인.
    boolean canWrite(Class<?> clazz, MediaType mediaType);

    // 이 HttpMessageConverter에서 지원하는 미디어 타입 목록을 리턴.
    List<MediaType> getSupportedMediaTypes();

    // 입력된 Message를 읽어 입력된 타입 형태로 변환하여 리턴
    T read(Class<T> clazz, HttpInputMessage inputMessage) throws IOException,
                                            HttpMessageNotReadableException;

    // 입력된 객체를 입력된 OutputMessage로 전송
    void write(T t, HttpOutputMessage outputMessage) throws IOException,
                                            HttpMessageNotWritableException;
}
```

Spring에서 제공하는 HttpMessageConverter 인터페이스 구현체들을 하나씩 살펴보자.

- **StringHttpMessageConverter**

  HTTP Request나 Response와 String간의 변환을 수행한다. 디폴트로 모든 text 미디어 타입('text/*')을 지원한다.

- **FormHttpMessageConverter**

  HTTP Request나 Response와 Form 데이터(MultiValueMap<String, String>) 간의 변환을 수행한다. 디폴트로 'application/x-www-form-urlencoded' 미디어 타입을 지원한다.

- **ByteArrayMessageConverter**

  HTTP Request나 Response와 byte 배열 간의 변환을 수행한다. 디폴트로 모든 미디어 타입('*/*')을 지원한다.

- **MarshallingHttpMessageConverter**

  HTTP Request나 Response를 Spring OXM의 Marshaller/Unmarshaller를 사용하여 XML로 변환한다. 디폴트로 'text/xml', 'application/xml' 미디어 타입을 지원한다.

- **MappingJacksonHttpMessageConverter**

  HTTP Request나 Response를 Jackson 라이브러리의 ObjectMapper를 사용하여 XML로 변환한다. 디폴트로 'application/json' 미디어 타입을 지원한다.

- **SourceHttpMessageConverter**

  HTTP Request나 Response와 javax.xml.transform.Source(DOMSource, SAXSource, StreamSource만 지원) 간의 변환을 수행한다. 디폴트로 지원하는 미디어 타입은 'text/xml', 'application/xml'이다.

- **BufferedImageHttpMessageConverter**

  HTTP Request나 Response와 java.awt.image.BufferedImage 간의 변환을 수행한다. Java I/O API에서 지원하는 모든 미디어 타입에 대해서 변환을 지원한다.

12.OXM (Object/XML Mapping)

OXM은 Spring에서 Object와 XML간의 변환을 위해서 JAXB, Castor, JiBX 같은 XML Marshalling 기술을 추상화한 기능으로 원래는 Spring Web Service 프로젝트에 포함되어 있던 모듈이 분리되어 Spring 3에서 Core 영역에 포함되었다. REST Feature 범위는 아니지만 MarshallingView 및 MarshallingHttpMessageConverter와 연관지어 이 장에서 설명하도록 하겠다.

Spring OXM은 다음과 같은 특징을 가진다.

- 간편한 설정

  Marshaller를 일반 빈과 동일하게 정의한다. 또한 'oxm' 네임스페이스를 제공하여 JAXB2, XmlBeans, JiBX 등을 사용한 Marshaller를 손쉽게 정의할 수 있게 해준다.

- 일관된 인터페이스

  Marshaller/Unmarshaller라는 두가지 인터페이스로 동작하기 때문에 OX Mapping Framework를 설정만으로 쉽게 변경할 수 있다. 또한 OX Mapping Framework을 섞어서(mix and match) 사용할 수도 있다.

- 일관된 예외 계층

  Mapping(Serialization)하다 발생한 Exception 처리를 위해서 XmlMappingException이라는 Root Exception을 제공한다.

Spring OXM에서 Marshaller와 Unmarshaller 인터페이스는 구분되어 있지만 Spring에서 제공하고 있는 실제 구현체들은 하나의 클래스에서 두 개의 인터페이스 모두를 구현해서 제공하고 있다. 그래서 구현 클래스 하나만 Bean으로 등록하면 Marshaller로 사용할 수도 있고 Unmarshaller로 사용할 수도 있다.

12.1.Programmatic Using

XML 변환을 위한 Marshaller는 아래와 같이 Bean으로 정의한 다음 클래스에서 Injection 받아서 사용할 수 있다. 예제에서는 Castor를 사용하고 있지만, JAXB, XMLBeans, JiBX, XStream 등도 Marshaller로 사용할 수 있다. 앞서 언급했듯이 CastorMarshaller는 Marshaller와 Unmarshaller 인터페이스를 모두 구현하였기 때문에 두 가지 용도로 참조할 수 있다.

```
<beans>
    <bean id="sample" class="SampleClass">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>

    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
</beans>
```

다음은 클래스에서 Marshaller를 사용하는 예제이다.

```
public class SampleClass {
    @Inject
    private Marshaller marshaller;
    @Inject
    private Unmarshaller unmarshaller;

    // 중략
    public void save() throws IOException {
        FileOutputStream os = null;
        try {
            os = new FileOutputStream(FILE_NAME);
            this.marshaller.marshal(movies, new StreamResult(os));
```

```
        } finally {
            if (os != null) {
                os.close();
            }
        }
    }
    // 중략
}
```

12.2.Declarative Using

Spring에서 제공하는 'oxm' namespace를 이용하면 Marshaller 설정을 간편하게 추가할 수 있다.이를 위해서는 XML 상단에 아래의 스키마 정의를 추가해야 한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oxm="http://www.springframework.org/schema/oxm"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm-3.1.xsd">
```

현재 제공하고 있는 태그들은 다음과 같다.

- jaxb2-marshaller
- xmlbeans-marshaller
- jibx-marshaller

상세한 설정 방법은 각각의 Marshaller 설명에서 더 자세히 살펴보도록 하겠다.

12.3.JAXB

JAXB는 W3C XML 스키마를 지원하는 Object/XML 매핑 프레임워크로 Spring에서는 JAXB 2.0 API를 사용한 Jaxb2Marshaller를 제공하고 있다.

Jaxb2Marshaller를 사용하기 위한 설정은 다음과 같다.

```
<beans>
    <bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>myapp.springrest.domain.Movie</value>
            </list>
        </property>
    </bean>
</beans>
```

스키마 Validation이 필요한 경우 'schema' 속성을 추가하여 스키마 파일을 지정해 줄 수 있다.

'oxm' namespace를 이용해서 아래와 같이 간편하게 설정할 수도 있다.

```
<oxm:jaxb2-marshaller id="marshaller" contextPath="myapp.springrest.domain"/>
```

다음은 springrest plugin의 src/test/resources/context-restclient.xml파일의 일부이다. <oxm:class-to-be-bound>를 이용하여 변환할 클래스 목록을 정의하였다.

```
<oxm:jaxb2-marshaller id="marshaller">
```

```
    <oxm:class-to-be-bound name="myapp.springrest.domain.Movie"/>
</oxm:jaxb2-marshaller>
```

12.4.Castor

Castor는 오픈 소스 XML 바인딩 프레임워크로, Java 객체와 XML간의 변환에 대해서 Castor에서 사용하는 디폴트 규칙을 그대로 따른다면 Spring에서는 제공하는 CastorMarshaller를 추가 설정 없이 간단하게 Bean으로 정의할 수 있다.

CastorMarshaller를 사용하기 위한 설정은 다음과 같다.

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />
</beans>
```

Castor의 디폴트 변환 양식을 변경하고자 하는 경우 Castor 매핑 파일을 작성하여 아래 예와 같이 mappingLocation 속성으로 정의해준다. Castor 매핑 파일을 작성방법에 대해서는 Castor XML Mapping [http://castor.org/xml-mapping.html]을 참조한다.

```
<beans>
    <bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller">
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

12.5.XMLBeans

XMLBeans는 Full XML 스키마를 지원하는 XML 바인딩 프레임워크로, 자세한 내용은 XMLBeans 웹사이트 [http://xmlbeans.apache.org/]를 참조하기 바란다. Spring에서 제공하는 Marshaller/Unmarshaller 구현체는 XmlBeansMarshaller이다.

XmlBeansMarshaller를 사용하기 위한 설정은 다음과 같다.

```
<beans>
    <bean id="xmlBeansMarshaller"
 class="org.springframework.oxm.xmlbeans.XmlBeansMarshaller" />
</beans>
```

단, XmlBeansMarshaller는 모든 java.lang.Object가 아닌 XmlObject 타입의 객체만 변환할 수 있다는 것을 주의해야한다.

'oxm' namespace를 이용해서 아래와 같이 간편하게 설정할 수도 있다.

```
<oxm:xmlbeans-marshaller id="marshaller"/>
```

12.6.JiBX

JiBX는 XML 데이터를 Java 오브젝트에 바인딩하는 데 사용되는 도구로, 자세한 내용은 JiBX 웹사이트 [http://jibx.sourceforge.net/]를 참조하기 바란다. Spring에서 제공하는 Marshaller/Unmarshaller 구현체는 JibxMarshaller이다.

JibxMarshaller를 사용하기 위한 설정은 다음과 같다.

```
<beans>
    <bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">anyframe.sample.domain.Movie</property>
```

```
    </bean>
</beans>
```

위의 예에서는 하나의 JibxMarshaller만 정의하였지만, 여러 클래스를 변환하는 경우 targetClass 속성을 다르게 정의한 여러 개의 JibxMarshaller가 정의되어야 한다.

'oxm' namespace를 이용해서 아래와 같이 간편하게 설정할 수도 있다.

```
<oxm:jibx-marshaller id="marshaller" target-class="anyframe.sample.domain.Movie"/>
```