Red Hat Fuse 7.11

Apache CXF 개발 가이드

Apache CXF 웹 서비스를 사용하여 애플리케이션 개발

Last Updated: 2023-09-29

Red Hat Fuse 7.11 Apache CXF 개발 가이드

Apache CXF 웹 서비스를 사용하여 애플리케이션 개발

법적 공지

초록

Apache CXF를 사용한 웹 서비스 개발 가이드.

차례

보다 포괄적 수용을 위한 오픈 소스 용어 교체

Red Hat은 코드, 문서, 웹 속성에서 문제가 있는 용어를 교체하기 위해 최선을 다하고 있습니다. 먼저 마스터(master), 슬레이브(slave), 블랙리스트(blacklist), 화이트리스트(whitelist) 등 네 가지 용어를 교체하고 있습니다. 이러한 변경 작업은 작업 범위가 크므로 향후 여러 릴리스에 걸쳐 점차 구현할 예정입니다. 자세한 내용은 CTO Chris Wright의 메시지 에서 참조하십시오.

I 부. WSDL 계약 작성

이 부분에서는 WSDL을 사용하여 웹 서비스 인터페이스를 정의하는 방법을 설명합니다.

1장. WSDL 계약 소개

초록

WSDL 문서는 웹 서비스 설명 언어와 여러 가지 가능한 확장을 사용하여 서비스를 정의합니다. 문서에는 논리적 부분이 있고 구체적인 부분이 있습니다. 계약의 추상 부분은 구현 중립 데이터 유형 및 메시지의 측면에서 서비스를 정의합니다. 문서의 구체적인 부분은 서비스를 구현하는 엔드포인트가 외부 세계와 상호작용하는 방식을 정의합니다.

서비스를 설계하는 데 권장되는 접근 방식은 코드를 작성하기 전에 WSDL 및 XML 스키마에서 서비스를 정의하는 것입니다. WSDL 문서를 직접 편집할 때는 문서와 올바른 정보가 올바른지 확인해야 합니다. 이 작업을 수행하려면 WSDL에 대해 어느 정도 익숙해야 합니다. W3C 웹 사이트 www.w3.org 에서 표준을 찾을 수 있습니다.

1.1. WSDL 문서의 구조

1.1.1. 개요

WSDL 문서는 루트 **정의** 요소에 포함된 요소의 컬렉션을 간단히 설명합니다. 이러한 요소는 서비스와 해당 서비스를 구현하는 엔드포인트가 액세스하는 방법을 설명합니다.

WSDL 문서에는 다음 두 가지 부분이 있습니다.

- 구현 중립적 용어로 서비스를 정의하는 논리 부분
- 서비스 를 구현하는 엔드포인트가 네트워크에 노출되는 방식을 정의하는 구체적인 부분

1.1.2. 논리 부분

WSDL 문서의 논리적 부분에는 유형 , **메시지**, **portType** 요소가 포함되어 있습니다. **서비스의 인터페이스와 서비스에서 교환된 메시지를 설명합니다. 형식 요소 내에서 XML 스키마는 메시지를 구성하는 데이터의 구조를 정의하는 데 사용됩니다.Within the types element, XML Schema is used to define the structure of the data that make up the messages. 많은 메시지 요소는 서비스에서 사용하는 메시지의 구조를 정의하는 데 사용됩니다. portType 요소에는 서비스에서 노출하는 작업에서 보낸 메시지를 정의하는 하나 이상의 작업 요소가 포함되어 있습니다.**

1.1.3. 세부 사항

WSDL 문서의 구체적인 부분에는 바인딩 및 서비스 요소가 포함되어 있습니다. 서비스를 구현하는 끝점이 외부 세계에 연결하는 방법을 설명합니다. 바인딩 요소는 메시지 요소에 의해 설명된 데이터 단위가 SOAP와 같은 구체적인 유선 데이터 형식으로 매핑되는 방법을 설명합니다. 서비스 요소에는 서비스를 구현하는 엔드포인트를 정의하는 하나 이상의 포트 요소가 포함됩니다.

1.2. WSDL 요소

WSDL 문서는 다음과 같은 요소로 구성되어 있습니다.

-

definitions - WSDL 문서의 루트 요소입니다. 이 요소의 특성은 WSDL 문서의 이름, 문서의 대상 네임스페이스, WSDL 문서에서 참조되는 네임스페이스의 단축 정의를 지정합니다.

- **Type -** 서비스에서 사용하는 메시지의 빌딩 블록을 구성하는 데이터 단위의 XML 스키마 정의입니다. 데이터 유형 정의에 대한 자세한 내용은 *2장. 논리 데이터 단위 정의* 을 참조하십시오.

- **Message -** 서비스 작업을 호출하는 동안 교환된 메시지에 대한 설명입니다. 이러한 요소는 서비스를 구성하는 작업의 인수를 정의합니다. 메시지 정의에 대한 자세한 내용은 *3장. 서비스에서 사용하는 논리 메시지 정의* 을 참조하십시오.

- **portType -** 서비스의 논리 인터페이스를 설명하는 작업 요소의 컬렉션입니다. 포트 유형 정의에 대한 자세한 내용은 *4장. 논리 인터페이스 정의* 을 참조하십시오.

- 작업 - 서비스에서 수행한 작업에 대한 설명입니다. 작업은 작업을 호출할 때 두 끝점 간에 전달되는 메시지에 의해 정의됩니다. 작업 정의에 대한 자세한 내용은 "작업" 을 참조하십시오.

- 바인딩 - 끝점에 대한 구체적인 데이터 형식 사양입니다. 바인딩 요소는 추상 메시지가 끝점에서 사용하는 구체적인 데이터 형식으로 매핑되는 방법을 정의합니다. 이 요소는 매개 변수 순서 및 반환 값과 같은 특정 값을 지정합니다.

- **Service -** 관련 포트 요소의 컬렉션입니다. 이러한 요소는 끝점 정의를 구성하는 리포지토리입니다.

- **port -** 바인딩 및 물리적 주소로 정의된 끝점입니다. 이러한 요소는 모든 추상 정의와 함께 전송 세부 정보를 가져오고, 서비스가 노출되는 물리적 끝점을 정의합니다.

1.3. 계약 설계

서비스에 대한 WSDL 계약을 설계하려면 다음 단계를 수행해야 합니다.

1. 서비스에서 사용하는 데이터 유형을 정의합니다.

2. 서비스에서 사용하는 메시지를 정의합니다.

3. **서비스의 인터페이스를 정의합니다.**

4. **각 인터페이스에서 사용하는 메시지 간 바인딩과 유선 데이터의 구체적인 표시를 정의합니다.**

5. **각 서비스에 대한 전송 세부 정보를 정의합니다.**

2장. 논리 데이터 단위 정의

초록

WSDL 계약 복잡한 데이터 형식의 서비스를 설명하는 경우 XML 스키마를 사용하여 논리 단위로 정의됩니다.

2.1. 논리 데이터 단위 소개

서비스를 정의할 때 가장 먼저 고려해야 할 것은 노출된 작업에 대한 매개 변수로 사용되는 데이터가 표현되는 방법입니다. 고정 데이터 구조를 사용하는 프로그래밍 언어로 작성된 애플리케이션과 달리 서비스는 여러 애플리케이션에서 사용할 수 있는 논리 단위로 데이터를 정의해야 합니다. 여기에는 두 단계가 포함됩니다.

1. 데이터를 서비스의 물리적 구현에서 사용하는 데이터 형식으로 매핑할 수 있는 논리 단위로 분할
2. 작업을 수행하기 위해 논리 단위를 엔드포인트 간에 전달되는 메시지로 결합

이 장에서는 첫 번째 단계에 대해 설명합니다. 3장. *서비스에서 사용하는 논리 메시지 정의* 두 번째 단계에 대해 설명합니다.

2.2. 데이터를 논리 데이터 단위로 매핑

2.2.1. 개요

서비스를 구현하는 데 사용되는 인터페이스는 작업 매개 변수를 XML 문서로서 나타내는 데이터를 정의합니다. 이미 구현된 서비스에 대한 인터페이스를 정의하는 경우 구현된 작업의 데이터 형식을 메시지로 어셈블할 수 있는 XML 요소를 구분하여 변환해야 합니다. 처음부터 시작하는 경우 메시지가 빌드된 빌딩 블록을 결정해야 구현 관점에서 이해할 수 있습니다.

2.2.2. 서비스 데이터 단위를 정의하는 데 사용 가능한 유형 시스템

WSDL 사양에 따르면 WSDL 계약에서 데이터 유형을 정의하기 위해 선택한 모든 유형 시스템을 사용할 수 있습니다. 그러나 W3C 사양은 XML 스키마가 WSDL 문서의 기본 표준 형식 시스템이라고 명시되어 있습니다. 따라서 XML 스키마는 Apache CXF의 내장 유형 시스템입니다.

2.2.3. 형식 시스템으로 XML 스키마

XML 스키마는 XML 문서를 구성하는 방법을 정의하는 데 사용됩니다. 이 작업은 문서를 구성하는 요소를 정의하여 수행됩니다. 이러한 요소는 xsd:int 와 같은 네이티브 XML 스키마 유형을 사용하거나 사용자가 정의한 형식을 사용할 수 있습니다. 사용자 정의 형식은 XML 요소의 조합을 사용하여 빌드되거나 기존 형식을 제한하여 정의됩니다.User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. 형식 정의와 요소 정의를 결합하면 복잡한 데이터를 포함할 수 있는 복잡한 XML 문서를 만들 수 있습니다.

WSDL XML 스키마에서 사용되는 경우 서비스와 상호 작용하는 데 사용되는 데이터를 보유하는 XML 문서의 구조를 정의합니다. 서비스에서 사용하는 데이터 단위를 정의할 때 메시지 파트의 구조를 지정하는 유형으로 정의할 수 있습니다. 또한 데이터 단위를 메시지 부분을 구성하는 요소로 정의할 수도 있습니다.

2.2.4. 데이터 단위 생성에 대한 고려 사항

서비스를 구현할 때 사용하는 유형을 직접 매핑하는 논리 데이터 단위를 생성하는 것이 좋습니다. 이 접근 방식이 효과가 있고 RPC 스타일 애플리케이션을 빌드하는 모델을 면밀하게 따르지만, 서비스 지향 아키텍처의 일부를 구축하는 데 반드시 이상적인 것은 아닙니다.

웹 서비스 상호 운용성 조직의 WS-I 기본 프로필은 데이터 단위를 정의하는 데 필요한 여러 지침을 제공하며 http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES 에서 액세스할 수 있습니다. 또한 W3C에서는 XML 스키마를 사용하여 WSDL 문서의 데이터 유형을 표현하기 위한 다음 지침도 제공합니다.

- 속성이 아닌 요소를 사용합니다.
- 프로토콜별 유형을 기본 유형으로 사용하지 마십시오.

2.3. 계약에 데이터 단위 추가

2.3.1. 개요

WSDL 계약을 만드는 방법에 따라 새로운 데이터 정의를 만들려면 다양한 양의 지식이 필요합니다. Apache CXF GUI 도구는 XML 스키마를 사용하여 데이터 유형을 설명하는 여러 가지 도움말을 제공합니다. 다른 XML 편집기는 다양한 수준의 지원을 제공합니다. 어떤 편집기를 선택하든 결과 계약이 어떻게 보이는지에 대한 지식을 갖는 것이 좋습니다.

2.3.2. 절차

WSDL 계약에 사용되는 데이터를 정의하려면 다음 단계가 포함됩니다.

1. 계약에서 설명하는 인터페이스에서 사용되는 모든 데이터 단위를 결정합니다.

2. 계약에서 형식 요소를 만듭니다.**Create a types element in your contract.**

3. 예 2.1. "WSDL 계약의 스키마 항목" 에 표시된 스키마 요소를 **type** 요소의 자식으로 생성합니다.

   targetNamespace 속성은 새 데이터 형식이 정의된 네임스페이스를 지정합니다. 가장 좋은 방법은 대상 네임스페이스에 대한 액세스를 제공하는 네임스페이스도 정의하는 것입니다. 나머지 항목은 변경하지 않아야 합니다.

   예 2.1. WSDL 계약의 스키마 항목

   ```
   <schema targetNamespace="http://schemas.iona.com/bank.idl"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:xsd1="http://schemas.iona.com/bank.idl"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   ```

4. 요소의 컬렉션인 각 복잡한 형식에 대해 **complexType** 요소를 사용하여 데이터 형식을 정의합니다. 2.5.1절. "데이터 구조 정의" 을 참조하십시오.

5. 각 배열에 대해 **complexType** 요소를 사용하여 데이터 유형을 정의합니다. 2.5.2절. "배열 정의" 을 참조하십시오.

6. 단순 형식에서 파생되는 각 복잡한 형식에 대해 **simpleType** 요소를 사용하여 데이터 형식을 정의합니다.**For each complex type that is derived from a simple type, define the data type using a simpleType element.** 2.5.4절. "제한으로 유형 정의" 을 참조하십시오.

7. 열거된 각 형식에 대해 **simpleType** 요소를 사용하여 데이터 형식을 정의합니다. 2.5.5절. "열거된 유형 정의" 을 참조하십시오.

8. 각 요소에 대해 요소 요소 를 사용하여 정의합니다. 2.6절. "요소 정의" 을 참조하십시오.

2.4. XML 스키마 간단한 유형

2.4.1. 개요

메시지 부분이 간단한 유형이 될 경우 이를 위한 유형 정의를 생성할 필요가 없습니다. 그러나 계약에 정의된 인터페이스에서 사용하는 복잡한 유형은 간단한 유형을 사용하여 정의됩니다.

2.4.2. 간단한 유형 입력

XML Schema 단순 유형은 주로 계약의 유형 섹션에 사용되는 요소 요소에 배치됩니다. 또한 제한 요소 및 확장 요소의 기본 특성에 사용됩니다.

간단한 유형은 **xsd** 접두사를 사용하여 항상 입력합니다. 예를 들어 요소가 **int** 임을 지정하려면 예 2.2. "간단한 유형으로 요소 정의" 에 표시된 대로 **type** 속성에 **xsd:int** 를 입력합니다.

예 2.2. 간단한 유형으로 요소 정의

```
<element name="simpleInt" type="xsd:int" />
```

2.4.3. 지원되는 XSD 간단한 유형

Apache CXF는 다음과 같은 **XML** 스키마 간단한 유형을 지원합니다.

- **xsd:string**
- **xsd:normalizedString**
- **xsd:int**
- **xsd:unsignedInt**
- **xsd:long**

- **xsd:unsignedLong**
- **xsd:short**
- **xsd:unsignedShort**
- **xsd:float**
- **xsd:double**
- **xsd:boolean**
- **xsd:byte**
- **xsd:unsignedByte**
- **xsd:integer**
- **xsd:positiveInteger**
- **xsd:negativeInteger**
- **xsd:nonPositiveInteger**
- **xsd:nonNegativeInteger**
- **xsd:decimal**

- **xsd:dateTime**
- **xsd:time**
- **xsd:date**
- **xsd:QName**
- **xsd:base64Binary**
- **xsd:hexBinary**
- **xsd:ID**
- **xsd:token**
- **xsd:language**
- **xsd:Name**
- **xsd:NCName**
- **xsd:NMTOKEN**
- **xsd:anySimpleType**
- **xsd:anyURI**

- xsd:gYear
- xsd:gMonth
- xsd:gDay
- xsd:gYearMonth
- xsd:gMonthDay

2.5. 복잡한 데이터 유형 정의

초록

XML 스키마는 단순 데이터 형식에서 복잡한 데이터 구조를 구축하기 위한 유연하고 강력한 메커니즘을 제공합니다. 요소 및 속성 시퀀스를 만들어 데이터 구조를 만들 수 있습니다.**You can create data structures by creating a sequence of elements and attributes.** 정의된 유형을 확장하여 더욱 복잡한 유형을 생성할 수도 있습니다.

복잡한 데이터 구조를 빌드하는 것 외에도 열거된 형식, 특정 범위의 값이 있는 데이터 형식 또는 기본 형식을 확장하거나 제한하여 특정 패턴을 따라야 하는 데이터 형식도 설명할 수 있습니다.**In addition to building complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.**

2.5.1. 데이터 구조 정의

2.5.1.1. 개요

XML 스키마에서 데이터 필드 컬렉션인 데이터 단위는 **complexType** 요소를 사용하여 정의됩니다. 복잡한 유형을 지정하려면 다음 세 가지 정보가 필요합니다.

1. 정의된 형식의 이름은 **complexType** 요소의 **name** 특성에 지정됩니다.

2. **complexType** 의 첫 번째 자식 요소는 유선에 배치될 때 구조 필드의 동작을 설명합니다. "복잡한 유형 종류" 을 참조하십시오.

3. 정의된 구조의 각 필드는 **complexType** 요소 의 손자인 요소에 정의됩니다. "구조의 부분 정의" 을 참조하십시오.

예를 들어 예 2.3. "간단한 구조" 에 표시된 구조는 두 개의 요소가 있는 복잡한 유형으로 **XML** 스키마로 정의됩니다.

예 2.3. 간단한 구조

```
struct personalInfo
{
  string name;
  int age;
};
```

예 2.4. "복잡한 유형" 예 2.3. "간단한 구조" 에 정의된 구조의 사용 가능한 **XML** 스키마 매핑을 보여줍니다. 예 2.4. "복잡한 유형" 로 정의된 구조는 **name** 및 **age** 의 두 가지 요소를 포함하는 메시지를 생성합니다.

.

예 2.4. 복잡한 유형

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

2.5.1.2. 복잡한 유형 종류

XML 스키마에는 **XML** 문서를 표시하고 유선으로 전달될 때 복잡한 형식의 필드를 구성하는 방법을 설명하는 세 가지 방법이 있습니다. **complexType** 요소의 첫 번째 자식 요소는 사용 중인 복잡한 유형을 결정합니다. 표 2.1. "복잡한 유형 설명자 요소" 복잡한 유형 동작을 정의하는 데 사용되는 요소를 보여줍니다.

표 2.1. 복잡한 유형 설명자 요소

요소	복잡한 유형 동작
순서	모든 복잡한 유형의 필드가 있을 수 있으며 유형 정의에 지정된 순서에 있어야 합니다.
all	모든 복잡한 유형의 필드는 존재할 수 있지만 임의의 순서로 있을 수 있습니다.
choice	구조의 요소 중 하나만 메시지에 배치할 수 있습니다.

예 2.5. "간단한 복잡한 선택 유형" 에 표시된 대로 선택 요소를 사용하여 구조가 정의된 경우 name 요소 또는 age 요소와 함께 메시지를 생성합니다.

예 2.5. 간단한 복잡한 선택 유형

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

2.5.1.3. 구조의 부분 정의

요소 요소 를 사용하여 구조를 구성하는 데이터 필드를 정의합니다. 모든 complexType 요소에는 하나 이상의 요소 가 포함되어야 합니다. complexType 요소 의 각 요소는 정의된 데이터 구조의 필드를 나타냅니다.

데이터 구조의 필드를 완전히 설명하기 위해 요소 요소에 는 다음 두 가지 필수 특성이 있습니다.

- **name 속성은 데이터 필드의 이름을 지정하고 정의된 복잡한 유형 내에서 고유해야 합니다.**
- **type 속성은 필드에 저장된 데이터 유형을 지정합니다. 형식은 XML Schema 단순 유형 중 하나이거나 계약에 정의된 이름이 지정된 복잡한 유형일 수 있습니다.**

name 및 type 외에도요소 요소에 는 minOcurrs 및 maxOccurs 라는 두 가지 일반적으로 사용되는 선택적 속성이 있습니다. 이러한 속성은 구조에서 필드가 발생하는 횟수에 바인딩됩니다. 기본적으로 각 필드는 복잡한 유형에서 한 번만 수행됩니다. 이러한 특성을 사용하여 필드에 필요한 횟수 또는 구조에 나타날 수 있는 횟수를 변경할 수 있습니다. 예를 들어 예 2.6. "발생 제약 조건을 사용하는 간단한 복합 유

형" 과 같이 최소 세 번 이상 발생해야 하는 필드인 **previous Job**을 정의할 수 있으며, **7**번 이상은 지정할 수 없습니다.

예 2.6. 발생 제약 조건을 사용하는 간단한 복합 유형

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
    <element name="previousJobs" type="xsd:string:
          minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

minOccurs 를 사용하면 예 2.7. "minOccurs를 0으로 설정한 간단한 복합 유형" 에 표시된 대로 **minOccurs** 를 **0**으로 설정하여 **age** 필드를 선택적으로 만들 수도 있습니다. 이 경우 사용 기간 을 생략할 수 있으며 데이터는 계속 유효합니다.

예 2.7. minOccurs를 **0**으로 설정한 간단한 복합 유형

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

2.5.1.4. 속성 정의

XML 문서에서 특성은 요소의 태그에 포함됩니다. 예를 들어 아래 코드의 **complexType** 요소에서 **name** 은 특성입니다. 복잡한 유형의 특성을 지정하려면 **complexType** 요소 정의에 특성 요소를 정의합니다. 특성 요소는 모든 , 시퀀스 또는 선택 요소 이후에만 나타날 수 있습니다.**An attribute element can appear only after the all,sequence, or choice element.** 각 복잡한 유형의 특성에 대해 하나의 특성 요소를 지정합니다. 모든 특성 요소는 **complexType** 요소의 직접 자식이어야 합니다.

예 2.8. 속성이 있는 복합 유형

```
<complexType name="personalInfo">
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
          minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="required" />
</complexType>
```

이전 코드에서 특성 요소는 **personalInfo** 복잡한 형식에 **age** 속성이 있음을 지정합니다.**In the previous code, the attribute element specifies that the personalInfo complex type has an age attribute. attribute** 요소에는 다음과 같은 속성이 있습니다.

- **name -** 특성을 식별하는 문자열을 지정하는 필수 특성입니다.
- **type -** 필드에 저장된 데이터의 유형을 지정합니다. 형식은 **XML Schema** 단순 유형 중 하나일 수 있습니다.
- 이 특성 을 갖는 데 복잡한 유형이 필요한지 여부를 지정하는 선택적 특성입니다. 유효한 값은 필수 또는 선택적 입니다. 기본값은 속성이 선택 사항입니다.

특성 요소에서 특성 의 기본값을 지정할 수 있는 선택적 **default** 특성을 지정할 수 있습니다.

2.5.2. 배열 정의

2.5.2.1. 개요

Apache CXF는 계약에서 배열을 정의하는 두 가지 방법을 지원합니다. 첫 번째는 **maxOccurs** 특성에 값이 **1**보다 큰 단일 요소를 사용하여 복잡한 형식을 정의합니다. 두 번째는 **SOAP** 배열을 사용하는 것입니다. **SOAP** 배열은 다차원 배열을 쉽게 정의하고 스파스로 채워진 배열을 전송할 수 있는 기능과 같은 추가 기능을 제공합니다.

2.5.2.2. 복합 형식 배열

복합 형식 배열은 시퀀스 복잡한 유형의 특수한 경우입니다. 단일 요소로 복잡한 유형을 정의하고 **maxOccurs** 특성에 대한 값을 지정하기만 하면 됩니다. 예를 들어, **20**개의 부동 소수점 숫자로 이루어진 배열을 정의하려면 예 2.9. “복잡한 유형 배열” 에 표시된 것과 유사한 복잡한 유형을 사용합니다.

예 2.9. 복잡한 유형 배열

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

minOccurs 특성의 값을 지정할 수도 있습니다.**You can also specify a value for the minOccurs attribute.**

2.5.2.3. SOAP 배열

SOAP 배열은 **wsdl:arrayType** 요소를 사용하여 **SOAP-ENC:Array** 기본 형식에서 파생하여 정의됩니다. 이에 대한 구문은 예 2.10. “wsdl:arrayType을 사용하여 파생되는 SOAP 배열의 구문” 에 표시되어 있습니다. **definitions** 요소가 **xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"** 를 선언하는지 확인합니다.

예 2.10. wsdl:arrayType을 사용하여 파생되는 SOAP 배열의 구문

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

TypeName 은 이 구문을 사용하여 새로 정의된 배열 유형의 이름을 지정합니다. ***elementType*** 은 배열의 요소 형식을 지정합니다. ***ArrayBounds*** 배열의 차원 수를 지정합니다. 단일 차원 배열을 지정하려면 **[].2**를 사용하여 **2** 차원 배열을 지정하려면 **[][]** 또는 **[,]** 를 사용합니다.

예를 들어 예 2.11. “SOAP 배열의 정의” 에 표시된 **SOAP** 배열인 **SOAPStrings**는 **1**차원 문자열 배열을 정의합니다. **wsdl:arrayType** 속성은 배열 요소, **xsd:string** 및 차원의 수를 지정합니다. **[]** 은 하나의 차원을 의미합니다.

예 2.11. SOAP 배열의 정의

```
<complexType name="SOAPStrings">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

SOAP 1.1 사양에 설명된 대로 간단한 요소를 사용하여 **SOAP Array**를 설명할 수도 있습니다. 이에 대한 구문은 예 2.12. “요소를 사용하여 파생되는 SOAP 배열의 구문” 에 표시되어 있습니다.

예 2.12. 요소를 사용하여 파생되는 SOAP 배열의 구문

```
<complexType name="TypeName">
  <complexContent>
```

```
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
              maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

이 구문을 사용하는 경우 요소의 **maxOccurs** 특성은 항상 **unbounded** 로 설정해야 합니다.

2.5.3. 확장으로 유형 정의

대부분의 주요 코딩 언어와 마찬가지로 **XML** 스키마를 사용하면 다른 데이터 형식에서 일부 요소를 상속하는 데이터 형식을 만들 수 있습니다. 이를 확장 기능으로 유형을 정의라고 합니다. 예를 들어, **planet** 이라는 새 요소를 추가하여 예 2.4. “복잡한 유형” 에 정의된 **personalInfo** 구조를 확장하는**anInfo**라는 새 유형을 만들 수 있습니다.

확장으로 정의된 유형에는 네 가지 부분이 있습니다.

1. 형식 이름은 **complexType** 요소의 **name** 특성으로 정의됩니다.

2. **complexContent** 요소는 새 형식에 둘 이상의 요소를 갖도록 지정합니다.

참고

복잡한 형식에 새 특성만 추가하는 경우 간단한**Content** 요소를 사용할 수 있습니다.

3. 기본 형식이라고 하는 새 형식을 파생 되는 형식은 확장 요소의 기본 특성에 지정됩니다.**The type from which the new type is derived, called the base type, is specified in the base attribute of the extension element.**

4. 새 유형의 요소 및 특성은 일반 복잡한 유형의 경우와 동일하게 확장 요소에 정의됩니다.

예를 들어, 외계인**Info** 는 예 2.13. “확장 기능으로 정의된 유형” 과 같이 정의됩니다.

예 2.13. 확장 기능으로 정의된 유형

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="xsd1:personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

2.5.4. 제한으로 유형 정의

2.5.4.1. 개요

XML 스키마를 사용하면 **XML** 스키마 단순 형식의 가능한 값을 제한하여 새 형식을 만들 수 있습니다.**XML Schema allows you to create new types by restricting the possible values of an XML Schema simple type.** 예를 들어 **SSN**은 정확히 **9**자로 구성된 간단한 유형 **SSN** 을 정의할 수 있습니다.**For example, you can define a simple type, SSN , which is a string of exactly 9 characters.** 단순 형식을 제한하여 정의한 새 형식은 **simpleType** 요소를 사용하여 정의됩니다.

제한으로 유형을 정의하려면 다음 세 가지 사항이 필요합니다.

1. 새 형식의 이름은 **simpleType** 요소의 **name** 특성으로 지정됩니다.
2. 기본 형식 이라고 하는 새 형식을 파생 되는 단순 형식은 제한 요소에 지정 됩니다.**The simple type from which the new type is derived, called the base type, is specified in the restriction element.** “기본 유형 지정” 을 참조하십시오.
3. 기본 유형에 배치된 제한을 정의하는 **facets** 라는 규칙은 제한 요소의 자식으로 정의됩니다. “제한 정의” 을 참조하십시오.

2.5.4.2. 기본 유형 지정

기본 유형은 새 유형을 정의하도록 제한되는 유형입니다. 제한 요소를 사용하여 지정됩니다. **restriction** 요소는 **simpleType** 요소의 유일한 자식이며 기본 유형을 지정하는 하나의 속성 **base** 를 갖습니다. 기본 유형은 **XML Schema** 간단한 유형 중 하나일 수 있습니다.

예를 들어, **xsd:int** 의 값을 제한하여 새 유형을 정의하려면 예 2.14. “기본 유형으로 int 사용” 에 표시

된 정의를 사용합니다.

예 2.14. 기본 유형으로 int 사용

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

2.5.4.3. 제한 정의

기본 유형에 따라 제한 사항을 정의하는 규칙을 ***facets*** 라고 합니다. **facet**는 **facet**가 적용되는 방법을 정의하는 하나의 속성 값이 있는 요소입니다. 사용 가능한 **facet** 및 유효한 값 설정은 기본 유형에 따라 다릅니다. 예를 들어 **xsd:string** 은 다음을 포함하여 **6**개의 **facet**를 지원합니다.

- 길이
- **minLength**
- **maxLength**
- 패턴
- **whitespace**
- **enumeration**

각 **facet** 요소는 제한 요소의 자식입니다.

2.5.4.4. 예제

예 2.15. “SSN 단순 유형 설명” 는 소셜 보안 번호를 나타내는 간단한 유형의 **SSN** 의 예를 보여줍니다. 결과 유형은 **xxx-xx-xxxx. <SSN>032-43-9876<SSN>**은 이 유형의 요소에 유효한 값이지만 **<SSN>032439876</SSN>**은 그렇지 않습니다.

예 2.15. SSN 단순 유형 설명

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

2.5.5. 열거된 유형 정의

2.5.5.1. 개요

XML 스키마의 열거된 형식은 제한에 따라 정의의 특별한 사례입니다. 이러한 항목은 모든 **XML** 스키마 기본 형식에서 지원하는 열거 형 **facet**를 사용하여 설명합니다. 대부분의 최신 프로그래밍 언어에서 열거된 형식과 마찬가지로, 이 유형의 변수는 지정된 값 중 하나만 가질 수 있습니다.

2.5.5.2. XML 스키마에서 열거형 정의

열거형 정의의 구문은 **예 2.16. "열거형 구문"** 에 표시되어 있습니다.

예 2.16. 열거형 구문

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

열거형 형식의 이름 을 지정합니다.**Specifies the name of the enumeration type. *EnumType*** 은 **case** 값의 형식을 지정합니다. **caseNValue** . 여기서 ***N*** 은 열거의 각 특정 사례에 대한 값을 지정합니다. ***CaseNValue*, where N is any number one or greater, specifies the value for each specific case of the enumeration.** 열거된 형식은 임의의 수의 대/소문자 값을 가질 수 있지만 간단한 형식에서 파생되기 때문에 한 번에 하나의 케이스 값만 유효합니다.**An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.**

2.5.5.3. 예제

예를 들어 **예 2.17. "widgetSize enumeration"** 에 표시된 열거형 위젯**Size**에 정의된 요소가 포함된 **XML** 문서는 **<wid getSize >big>big <widgetSize>**가 포함된 경우 유효하지만

<widgetSize>big,mungo</widgetSize>가 포함된 경우 유효하지 않습니다.

예 2.17. widgetSize enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
  </restriction>
</simpleType>
```

2.6. 요소 정의

XML 스키마의 요소는 스키마에서 생성된 **XML** 문서에 있는 요소의 인스턴스를 나타냅니다. 가장 기본적인 요소는 단일 요소로 구성됩니다. 복잡한 형식의 멤버를 정의하는 데 사용되는 요소 요소와 마찬가지로 다음 세 가지 특성이 있습니다.

- **XML** 문서에 표시되는 요소의 이름 을 지정하는 필수 특성입니다.**A required attribute that specifies the name of the element as it appears in an XML document.**
- **type -** 요소의 유형을 지정합니다. 형식은 모든 **XML** 스키마 기본 유형 또는 계약에 정의된 이름이 지정된 복잡한 유형일 수 있습니다. 형식에 인라인 정의가 있는 경우 이 속성을 생략할 수 있습니다.
- **nillable -** 문서에서 요소를 완전히 생략할 수 있는지 여부를 지정합니다. **nillable** 을 **true** 로 설정하면 스키마를 사용하여 생성된 문서에서 요소를 생략할 수 있습니다.

한 요소에는 인라인 유형 정의가 있을 수도 있습니다. 인라인 형식은 **complexType** 요소 또는 **simpleType** 요소를 사용하여 지정됩니다. 데이터 유형이 복잡하고 간단한지 지정하면 각 데이터 유형에 사용 가능한 도구를 사용하여 필요한 모든 유형의 데이터를 정의할 수 있습니다. 인라인 유형 정의는 재사용할 수 없으므로 권장되지 않습니다.

3장. 서비스에서 사용하는 논리 메시지 정의

초록

서비스는 해당 작업이 호출될 때 교환된 메시지에 의해 정의됩니다. **WSDL** 계약에서 이러한 메시지는 **message** 요소를 사용하여 정의됩니다. 메시지는 부분 요소를 사용하여 정의된 하나 이상의 부분 으로 구성됩니다.

3.1. 개요

서비스의 작업은 작업이 호출될 때 교환되는 논리 메시지를 지정하여 정의합니다. 이러한 논리 메시지는 네트워크를 **XML** 문서로 전달되는 데이터를 정의합니다. 메서드 호출의 일부인 모든 매개 변수가 포함되어 있습니다. 논리 메시지는 계약의 **message** 요소를 사용하여 정의됩니다. 각 논리 메시지는 부분 요소에 정의된 하나 이상의 부분 으로 구성됩니다.

메시지에서 각 매개 변수를 별도의 부분으로 나열할 수 있지만 권장 방법은 작업에 필요한 데이터를 캡슐화하는 단일 부분만 사용하는 것입니다.

3.2. MESSAGES 및 PARAMETER 목록

서비스에서 노출하는 각 작업에는 하나의 입력 메시지와 하나의 출력 메시지만 있을 수 있습니다. 입력 메시지는 작업이 호출될 때 서비스에서 수신하는 모든 정보를 정의합니다. 출력 메시지는 작업이 완료될 때 서비스에서 반환하는 모든 데이터를 정의합니다. 오류 메시지는 오류가 발생할 때 서비스에서 반환하는 데이터를 정의합니다.

또한 각 작업에는 여러 개의 오류 메시지가 있을 수 있습니다. 오류 메시지는 서비스가 오류가 발생할 때 반환되는 데이터를 정의합니다. 이러한 메시지는 일반적으로 소비자가 오류를 이해하기 위해 충분한 정보를 제공하는 한 부분만 있습니다.

3.3. 레거시 시스템과의 통합을 위한 메시지 설계

기존 애플리케이션을 서비스로 정의하는 경우 작업을 구현하는 메서드에서 사용하는 각 매개 변수가 메시지에 표시되어 있는지 확인해야 합니다. 또한 반환 값이 작업의 출력 메시지에 포함되어 있는지 확인해야 합니다.

메시지를 정의하는 한 가지 방법은 **RPC** 스타일입니다. **RPC** 스타일을 사용할 때 메서드의 매개 변수 목록에서 각 매개변수에 대해 하나의 부분을 사용하여 메시지를 정의합니다. 각 메시지 부분은 계약의 형식 요소에 정의된 유형을 기반으로 합니다. 입력 메시지에는 메서드의 각 입력 매개변수에 대해 하나의 부분이 포함됩니다. 출력 메시지에는 각 출력 매개 변수에 대한 한 부분이 포함되어 있으며 필요한 경우 반

환 값을 나타내는 파트가 포함되어 있습니다. 매개변수가 입력 및 출력 매개변수 둘 다이면 입력 메시지와 출력 메시지의 일부로 나열됩니다.

RPC 스타일 메시지 정의는 서비스가 Tibco 또는 CORBA와 같은 전송을 사용하는 레거시 시스템을 활성화하는 경우 유용합니다. 이러한 시스템은 절차 및 방법을 중심으로 설계되었습니다. 따라서 호출되는 작업의 매개 변수 목록과 유사한 메시지를 사용하여 모델링할 수 있습니다. RPC 스타일은 또한 노출 중인 서비스와 애플리케이션을 더 깔끔하게 매핑합니다.

3.4. SOAP 서비스에 대한 메시지 설계

RPC 스타일은 기존 시스템을 모델링하는 데 유용하지만 서비스의 커뮤니티는 래핑된 문서 스타일을 적극 권장합니다. 포장된 문서 스타일로, 각 메시지에는 하나의 부분이 있습니다. 메시지의 부분은 계약의 형식 요소에 정의된 래퍼 요소를 참조합니다. 래퍼 요소에는 다음과 같은 특징이 있습니다.

- 요소 시퀀스를 포함하는 복합 형식입니다.A complex type containing a sequence of elements. 자세한 내용은 2.5절. “복잡한 데이터 유형 정의” 에서 참조하십시오.
- 입력 메시지의 래퍼인 경우:
  - 각 메서드의 입력 매개 변수에 대해 하나의 요소가 있습니다.
  - 해당 이름은 연결된 작업의 이름과 동일합니다.
- 출력 메시지의 래퍼인 경우:
  - 각 메서드의 출력 매개 변수와 메서드의 inout 매개 변수에 대해 하나의 요소가 있습니다.
  - 첫 번째 요소는 메서드의 return 매개 변수를 나타냅니다.
  - 해당 이름은 래퍼가 연결된 작업의 이름에 Response 를 추가하여 생성됩니다.

3.5. 메시지 이름 지정

계약의 각 메시지에는 해당 네임 스페이스 내에서 고유한 이름이 있어야 합니다. 다음 명명 규칙을 사용하는 것이 좋습니다.

- 메시지는 단일 작업에서만 사용해야 합니다.
- 입력 메시지 이름은 작업 이름에 요청 을 추가하여 구성됩니다.
- 출력 메시지 이름은 작업 이름에 **Response** 를 추가하여 구성됩니다.
- 오류 메시지 이름은 오류 이유를 나타냅니다.

3.6. 메시지 부분

메시지 부분은 논리 메시지의 공식적인 데이터 단위입니다. 각 부분은 부분 요소를 사용하여 정의되며 **name** 속성 및 해당 데이터 유형을 지정하는 **type** 속성 또는 요소 특성으로 식별됩니다. 데이터 유형 속성은 표 3.1. “파트 데이터 유형 속성” 에 나열됩니다.

표 3.1. 파트 데이터 유형 속성

속성	설명
element="*elem_name*"	이 부분의 데이터 유형은 *elem_name* 이라는 요소에 의해 정의됩니다.
type="*type_name*"	이 부분의 데이터 유형은 *type_name* 이라고 하는 유형으로 정의됩니다.

메시지는 부분 이름을 재사용할 수 있습니다. 예를 들어 메서드에 참조로 전달되는 매개 변수가 있는 **foo** 이거나 **in/out**인 경우 예 3.1. “재사용된 부분” 과 같이 요청 메시지와 응답 메시지 모두에 포함될 수 있습니다.

예 3.1. 재사용된 부분

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

3.7. 예제

예를 들어, 개인 정보를 저장하고 직원의 ID 번호를 기반으로 직원의 데이터를 반환하는 방법을 제공하는 서버가 있다고 가정해 보겠습니다. 데이터를 조회하기 위한 메서드 서명은 예 3.2. “personalInfo lookup method” 과 유사합니다.

예 3.2. personalInfo lookup method

```
personalInfo lookup(long empId)
```

이 메서드 서명은 예 3.3. “RPC WSDL 메시지 정의” 에 표시된 RPC 스타일 WSDL 조각에 매핑할 수 있습니다.

예 3.3. RPC WSDL 메시지 정의

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
<message/>
<message name="personalLookupResponse>
  <part name="return" element="xsd1:personalInfo"/>
<message/>
```

또한 예 3.4. “문서 WSDL 메시지 정의 줄 바꿈” 에 표시된 래핑된 문서 스타일 WSDL 조각에 매핑할 수도 있습니다.

예 3.4. 문서 WSDL 메시지 정의 줄 바꿈

```
<wsdl:types>
  <xsd:schema ... >
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="empID" type="xsd:int" />
      </sequence>
    </complexType>
  </element>
  <element name="personalLookupResponse">
    <complexType>
      <sequence>
        <element name="return" type="personalInfo" />
      </sequence>
    </complexType>
  </element>
  </schema>
```

```
</types>
<wsdl:message name="personalLookupRequest">
  <wsdl:part name="empId" element="xsd1:personalLookup"/>
<message/>
<wsdl:message name="personalLookupResponse">
  <wsdl:part name="return" element="xsd1:personalLookupResponse"/>
<message/>
```

4장. 논리 인터페이스 정의

초록

논리 서비스 인터페이스는 **portType** 요소를 사용하여 정의합니다.

4.1. 개요

논리 서비스 인터페이스는 **WSDL portType** 요소를 사용하여 정의합니다. **portType** 요소는 추상 작업 정의의 컬렉션입니다. 각 작업은 작업에서 나타내는 트랜잭션을 완료하는 데 사용되는 입력, 출력 및 오류 메시지에 의해 정의됩니다.**Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. portType** 요소에서 정의한 서비스 인터페이스를 구현하기 위해 코드가 생성되면 각 작업은 계약에 지정된 입력, 출력 및 오류 메시지에 정의된 매개 변수를 포함하는 메서드로 변환됩니다.

4.2. 프로세스

WSDL 계약에서 논리 인터페이스를 정의하려면 다음을 수행해야합니다.

1. **interface** 정의를 포함하도록 **portType** 요소를 만들고 이를 고유한 이름을 지정합니다. “포트 유형”을 참조하십시오.
2. 인터페이스에 정의된 각 작업에 대해 작업 요소를 생성합니다. “작업”을 참조하십시오.
3. 각 작업에 대해 작업의 매개 변수 목록, 반환 유형 및 예외를 나타내는 데 사용되는 메시지를 지정합니다. “작업 메시지”을 참조하십시오.

4.3. 포트 유형

WSDL portType 요소는 논리 인터페이스 정의의 루트 요소입니다. 많은 웹 서비스 구현에서는 **portType** 요소를 생성된 구현 오브젝트에 직접 매핑하지만, 논리 인터페이스 정의는 구현된 서비스에서 제공하는 정확한 기능을 지정하지 않습니다. 예를 들어, **ticketSystem**이라는 논리적 인터페이스는 콘서트 티켓을 판매하거나 주차 티켓을 발행하는 구현을 초래할 수 있습니다.

portType 요소는 정의된 서비스를 노출하는 엔드포인트에서 사용하는 물리적 데이터를 정의하기 위해 바인딩에 매핑되는 **WSDL** 문서의 단위입니다.

WSDL 문서의 각 portType 요소에는 name 특성을 사용하여 지정된 고유한 이름이 있어야 하며 작업 요소에 설명된 작업 컬렉션으로 구성되어 있습니다. WSDL 문서는 모든 수의 포트 유형을 설명할 수 있습니다.

4.4. 작업

WSDL 작업 요소를 사용하여 정의하는 논리적 작업 에서는 두 끝점 간의 상호 작용을 정의합니다. 예를 들어, 검사 계정 균형을 위한 요청과 위젯의 그라스에 대한 주문은 모두 동작으로 정의될 수 있습니다.

portType 요소 내에 정의된 각 작업에는 name 특성을 사용하여 지정된 고유한 이름이 있어야 합니다. 작업을 정의하는 데 name 속성이 필요합니다.

4.5. 작업 메시지

논리 작업은 작업을 실행하기 위해 끝점 간에 전달되는 논리 메시지를 나타내는 요소 집합으로 구성됩니다. 작업을 설명할 수 있는 요소는 표 4.1. "작업 메시지 요소" 에 나열되어 있습니다.

표 4.1. 작업 메시지 요소

요소	설명
input	요청이 수행될 때 클라이언트 끝점에서 서비스 공급자로 보내는 메시지를 지정합니다.Specifies the message the client endpoint sends to the service provider when a request is made. 이 메시지의 일부는 작업의 입력 매개변수에 해당합니다.
출력	서비스 공급자가 요청에 대한 응답으로 클라이언트 끝점으로 보내는 메시지를 지정합니다. 이 메시지의 일부는 참조로 전달된 값과 같이 서비스 공급자가 변경할 수 있는 모든 작업 매개변수에 해당합니다. 여기에는 작업의 반환 값이 포함됩니다.
fault	끝점 간에 오류 조건을 전달하는 데 사용되는 메시지를 지정합니다.

하나 이상의 입력 또는 하나의 출력 요소가 있어야 하는 작업입니다.An operation is required to have at least one input or one output element. 작업에는 입력 및 출력 요소가 둘 다 있을 수 있지만 각 요소 중 하나만 있을 수 있습니다. 작업에 오류 가 있는 요소가 필요하지 않지만 필요한 경우 여러 개의 오류 요소가 있을 수 있습니다.

요소에는 표 4.2. “입력 및 출력 요소의 특성” 에 나열된 두 가지 속성이 있습니다.

표 4.2. 입력 및 출력 요소의 특성

속성	설명
name	작업을 구체적인 데이터 형식으로 매핑할 때 참조할 수 있도록 메시지를 식별합니다.Identifies the message so it can be referenced when mapping the operation to a specific data format. 이름은 포함된 포트 유형 내에서 고유해야 합니다.
message	전송 또는 수신 중인 데이터를 설명하는 추상 메시지를 지정합니다. **message** 속성 값은 WSDL 문서에 정의된 추상 메시지 중 하나의 **name** 속성에 해당해야 합니다.

모든 입력 및 출력 요소에 대한 **name** 속성을 지정할 필요는 없습니다. **WSDL**은 포함된 작업의 이름을 기반으로 기본 이름 지정 스키마를 제공합니다. 작업에 하나의 요소만 사용되는 경우 요소 이름은 기본적으로 작업 이름으로 설정됩니다. 입력 및 출력 요소를 모두 사용하는 경우 요소 이름은 각각 이름에 **Request** 또는 **Response** 가 포함된 작업의 이름으로 설정됩니다.

4.6. 반환 값

작업 요소는 작업 중에 전달된 데이터에 대한 추상 정의이므로 **WSDL**은 작업에 대해 지정된 반환 값을 제공하지 않습니다. 메서드가 값을 반환하면 출력 요소에 해당 메시지의 마지막 부분으로 매핑됩니다.

4.7. 예제

예를 들어 예 4.1. “personalInfo 조회 인터페이스” 에 표시된 인터페이스와 유사한 인터페이스가 있을 수 있습니다.

예 4.1. personalInfo 조회 인터페이스

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

이 인터페이스는 예 4.2. “personalInfo 조회 포트 유형” 의 포트 유형에 매핑할 수 있습니다.

예 4.2. personalInfo 조회 포트 유형

```
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="tns:personalLookupRequest"/>
    <output name="return" message="tns:personalLookupResponse"/>
    <fault name="exception" message="tns:idNotFoundException"/>
  </operation>
</portType>
```

II 부. 웹 서비스 바인딩

이 부분에서는 **Apache CXF** 바인딩을 **WSDL** 문서에 추가하는 방법을 설명합니다.

5장. WSDL의 바인딩 이해

초록

바인딩은 서비스를 정의하는 데 사용되는 논리 메시지를 엔드포인트에서 전송 및 수신할 수 있는 구체적인 페이로드 형식으로 매핑합니다.

5.1. 개요

바인딩은 서비스에서 사용하는 논리 메시지 사이의 브리지를 물리적 세계에서 사용하는 구체적인 데이터 형식으로 제공합니다. 이는 논리 메시지가 끝점에 의해 유선에서 사용되는 페이로드 형식으로 매핑되는 방법을 설명합니다. 매개 변수 순서, 구체적인 데이터 형식 및 반환 값 같은 세부 정보가 지정된 바인딩 내에 있습니다.**It is within the bindings that details such as parameter order, specific data types, and return values are specified.** 예를 들어 메시지의 일부는 **RPC** 호출에 필요한 순서를 반영하도록 바인딩에서 다시 정렬할 수 있습니다.**For example, the parts of a message can be reordered in a binding to reflect the order required by an RPC call.** 바인딩 유형에 따라 메서드의 반환 유형을 나타내는 메시지 부분（_ 있는 경우）을 식별할 수도 있습니다.**Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.**

5.2. 포트 유형 및 바인딩

포트 유형 및 바인딩은 직접 관련이 있습니다. 포트 유형은 두 논리 서비스 간의 상호 작용 집합에 대한 추상 정의입니다. 바인딩은 논리 서비스를 구현하는 데 사용되는 메시지가 물리적 세계에서 인스턴스화되는 방법에 대한 구체적인 정의입니다. 그러면 각 바인딩이 포트 유형에서 정의한 논리 서비스를 노출하는 끝점의 정의를 완료하는 네트워크 세부 정보 집합과 연결됩니다.

endpoint가 단일 서비스만 정의하도록 하기 위해 **WSDL**은 바인딩이 단일 포트 유형만 나타낼 수 있어야 합니다. 예를 들어 두 포트 유형과 계약을 체결한 경우 두 포트 유형을 모두 매핑하는 단일 바인딩을 구체적인 데이터 형식으로 작성할 수 없었습니다. 두 개의 바인딩이 필요합니다.

그러나 **WSDL**은 포트 유형을 여러 바인딩에 매핑할 수 있습니다. 예를 들어 계약에 단일 포트 유형이 있는 경우 둘 이상의 바인딩에 매핑할 수 있습니다. 각 바인딩은 메시지의 일부가 매핑되는 방식을 변경하거나 메시지에 대해 완전히 다른 페이로드 형식을 지정할 수 있습니다.

5.3. WSDL 요소

바인딩은 **WSDL** 바인딩 요소를 사용하여 계약에 정의됩니다. 바인딩 요소는 **PortType**에 대한 참조를 제공하는 바인딩 및 형식에 대한 고유 이름을 지정하는 등의 특성으로 구성됩니다. 이 속성의 값은 **4장.** *논리 인터페이스 정의* 에서 설명한 대로 바인딩을 엔드포인트와 연결하는 데 사용됩니다.

실제 매핑은 바인딩 요소의 자식에서 정의됩니다. 이러한 요소는 사용하기로 결정한 페이로드 형식의 유형에 따라 달라집니다. 다양한 페이로드 형식과 매핑을 지정하는 데 사용되는 요소는 다음 장에서 설명합니다.

5.4. 계약에 추가

Apache CXF는 사전 정의된 서비스 인터페이스에 대한 바인딩을 생성할 수 있는 명령줄 도구를 제공합니다.

이 도구는 귀하의 계약에 적절한 요소를 추가합니다. 그러나 다양한 유형의 바인딩 작동 방식에 대한 지식이 있는 것이 좋습니다.

텍스트 편집기를 사용하여 계약에 바인딩을 추가할 수도 있습니다. 직접 계약을 편집하면 계약이 유효한지 확인할 책임이 있습니다.

5.5. 지원되는 바인딩

Apache CXF는 다음 바인딩을 지원합니다.

- SOAP 1.1
- SOAP 1.2
- CORBA
- 순수 XML

6장. SOAP 1.1 메시지 사용

초록

Apache CXF는 SOAP 헤더를 사용하지 않는 SOAP 1.1 바인딩을 생성하는 도구를 제공합니다. 그러나 텍스트 또는 XML 편집기를 사용하여 바인딩에 SOAP 헤더를 추가할 수 있습니다.However, you can add SOAP headers to your binding using any text or XML editor.

6.1. SOAP 1.1 바인딩 추가

6.1.1. wsdl2soap 사용

wsdl2soap 을 사용하여 SOAP 1.1 바인딩을 생성하려면 다음 명령을 사용합니다. wsdl2soap-i*port-type-name*-b*binding-name*-d*output-directory*-o *output*-namespace -n*soap-body-namespace*-style (ral/rpc)-use (ral/encoded)-verb-oselll

참고

***wsdl2soap** 을 사용하려면 **Apache CXF** 배포판을 다운로드해야 합니다.*

명령에는 다음 옵션이 있습니다.

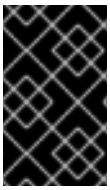
옵션	해석
-i *port-type-name*	바인딩이 생성되는 **portType** 요소를 지정합니다.
wsdlurl	**portType** 요소 정의를 포함하는 WSDL 파일의 경로와 이름입니다.

툴에는 다음과 같은 선택적 인수가 있습니다.

옵션	해석
-b *binding-name*	생성된 SOAP 바인딩의 이름을 지정합니다.
-d *output-directory*	생성된 WSDL 파일을 배치할 디렉터리를 지정합니다.
-o *output-file*	생성된 WSDL 파일의 이름을 지정합니다.

옵션	해석
-n *soap-body-namespace*	스타일이 RPC일 때 SOAP body 네임스페이스를 지정합니다.
-style (document/rpc)	SOAP 바인딩에서 사용할 인코딩 스타일(document 또는 RPC)을 지정합니다.Specifies the encoding style (document or RPC) to use in the SOAP binding. 기본값은 document입니다.
-use (literal/encoded)	SOAP 바인딩에 사용할 바인딩 사용(encoded 또는 literal)을 지정합니다.Specifies the binding use (encoded or literal) to use in the SOAP binding. 기본값은 literal입니다.
-v	도구의 버전 번호를 표시합니다.
-verbose	코드 생성 프로세스 중 주석을 표시합니다.
-quiet	코드 생성 프로세스 중 주석을 비활성화합니다.

***-iport-type-name** 및 **wsdlurl** 인수가 필요합니다. **-style rpc** 인수가 지정된 경우 **-nsoap-body-namspace** 인수도 필요합니다. 다른 모든 인수는 선택 사항이며 순서에 따라 나열될 수 있습니다.*



중요

***wsdl2soap** 은 문서 / 인코딩된 **SOAP** 바인딩 생성을 지원하지 않습니다.*

6.1.2. 예제

*시스템에 주문을 처리하고 주문을 처리하는 단일 작업을 제공하는 인터페이스가 있는 경우 예 6.1. “시스템 인터페이스 순서” 에 표시된 것과 유사한 **WSDL** 조각에 정의되어 있습니다.*

예 6.1. 시스템 인터페이스 순서

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

orderWidgets 에 대해 생성된 **SOAP** 바인딩은 예 6.2. “orderWidgets에 대한 SOAP 1.1 바인딩” 에 표시됩니다.

예 6.2. orderWidgets에 대한 SOAP 1.1 바인딩

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
```

이 바인딩은 문서/ 전문 메시지 스타일을 사용하여 메시지를 전송하도록 지정합니다.

6.2. SOAP 1.1 바인딩에 SOAP 헤더 추가

6.2.1. 개요

SOAP 헤더는 soap:header 요소를 기본 SOAP 1.1 바인딩에 추가하여 정의됩니다. soap:header 요소는 바인딩의 입력,출력 및 fault 요소의 선택적 자식입니다. SOAP 헤더는 상위 메시지의 일부가 됩니다. SOAP 헤더는 메시지 및 메시지 부분을 지정하여 정의됩니다. 각 SOAP 헤더에는 하나의 메시지 부분만 포함할 수 있지만 필요한 만큼 SOAP 헤더를 삽입할 수 있습니다.

6.2.2. 구문

SOAP 헤더 정의의 구문은 예 6.3. “Header Syntax” 에 표시됩니다. soap:header 의 message 속성은 헤더에 삽입되는 메시지의 정규화된 이름입니다. part 속성은 SOAP 헤더에 삽입된 메시지 부분의 이름입니다. SOAP 헤더는 항상 문서 스타일이므로 SOAP 헤더에 삽입된 WSDL 메시지 부분은 요소를 사용하여 정의해야 합니다. 메시지 와 부분 속성을 함께 사용하면 SOAP 헤더에 삽입할 데이터를 완전히 설명합니다.

예 6.3. Header Syntax

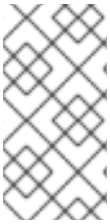
```
<binding name="headwig">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap:operation soapAction="" style="document"/>
      <input name="grain">
        <soap:body ... />
        <soap:header message="QName" part="partName"/>
      </input>
...
</binding>
```

필수 메시지 및 부분 속성 외에도 soap:header 는 네임스페이스, 사용, encodingStyle 속성도 지원합니다. 이 속성은 soap:header 에 대해 동일한 기능을 합니다. soap:body.

6.2.3. 본문과 헤더 간 메시지 분할

SOAP 헤더에 삽입된 메시지 부분은 계약의 유효한 메시지 부분일 수 있습니다. SOAP 본문으로 사용되는 상위 메시지의 일부일 수도 있습니다. 동일한 메시지에서 정보를 두 번 보낼 가능성이 낮기 때문에 SOAP 바인딩은 SOAP 본문에 삽입되는 메시지 부분을 지정하기 위한 수단을 제공합니다.

soap:body 요소에는 공백으로 구분된 부분 이름 목록을 사용하는 선택적 속성인 파트가 있습니다. 일부 가 정의되면 나열된 메시지 부분만 SOAP 본문에 삽입됩니다. 그런 다음 나머지 부분을 SOAP 헤더에 삽입할 수 있습니다.



참고

상위 메시지의 일부를 사용하여 **SOAP** 헤더를 정의할 때 **Apache CXF**는 자동으로 **SOAP** 헤더를 채웁니다.

6.2.4. 예제

예 6.4. “SOAP 헤더와의 SOAP 1.1 바인딩” 예 6.1. “시스템 인터페이스 순서” 에 표시된 **orderWidgets** 서비스의 수정된 버전을 표시합니다. 각 주문에 요청 및 응답의 **SOAP** 헤더에 배치된 **xsd:base64binary** 값이 있도록 이 버전이 수정되었습니다. **SOAP** 헤더는 **widgetKey** 메시지의 **keyVal** 파트로 정의됩니다. 이 경우 입력 또는 출력 메시지의 일부가 아니기 때문에 애플리케이션 논리에 **SOAP** 헤더를 추가해야 합니다.

예 6.4. SOAP 헤더와의 SOAP 1.1 바인딩

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
```

```
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap:body use="literal"/>
        <soap:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

헤더 값이 입력 및 출력 메시지의 일부임을 예 6.4. "SOAP 헤더와의 SOAP 1.1 바인딩" 로 수정할 수도 있습니다.

7장. SOAP 1.2 메시지 사용

초록

Apache CXF는 SOAP 헤더를 사용하지 않는 SOAP 1.2 바인딩을 생성하는 도구를 제공합니다. 텍스트 또는 XML 편집기를 사용하여 바인딩에 SOAP 헤더를 추가할 수 있습니다.

7.1. WSDL 문서에 SOAP 1.2 바인딩 추가

7.1.1. wsdl2soap 사용

참고

wsdl2soap을 사용하려면 Apache CXF 배포판을 다운로드해야 합니다.

wsdl2soap 을 사용하여 SOAP 1.2 바인딩을 생성하려면 다음 명령을 사용합니다. wsdl2soap-iport-type-name-bbinding-name-soap12-doutput-directory-ooutput-file-nsoap-body-namespace-style (document/rpc)-use(literal/encoded)-v-verbose-quietwsdlurl 툴에는 다음과 같은 필수 인수가 있습니다.

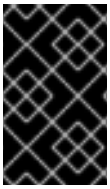
옵션	해석
-i *port-type-name*	바인딩이 생성되는 **portType** 요소를 지정합니다.
-soap12	생성된 바인딩에서 SOAP 1.2를 사용하도록 지정합니다.
wsdlurl	**portType** 요소 정의를 포함하는 WSDL 파일의 경로와 이름입니다.

툴에는 다음과 같은 선택적 인수가 있습니다.

옵션	해석
-b *binding-name*	생성된 SOAP 바인딩의 이름을 지정합니다.
-soap12	생성된 바인딩에서 SOAP 1.2를 사용하도록 지정합니다.
-d *output-directory*	생성된 WSDL 파일을 배치할 디렉터리를 지정합니다.

옵션	해석
-o *output-file*	생성된 WSDL 파일의 이름을 지정합니다.
-n *soap-body-namespace*	스타일이 RPC일 때 SOAP body 네임스페이스를 지정합니다.
-style (document/rpc)	SOAP 바인딩에서 사용할 인코딩 스타일(document 또는 RPC)을 지정합니다.Specifies the encoding style (document or RPC) to use in the SOAP binding. 기본값은 document입니다.
-use (literal/encoded)	SOAP 바인딩에 사용할 바인딩 사용(encoded 또는 literal)을 지정합니다.Specifies the binding use (encoded or literal) to use in the SOAP binding. 기본값은 literal입니다.
-v	도구의 버전 번호를 표시합니다.
-verbose	코드 생성 프로세스 중 주석을 표시합니다.
-quiet	코드 생성 프로세스 중 주석을 비활성화합니다.

***-i port-type-name** 및 **wsdlurl** 인수가 필요합니다. **-style rpc** 인수를 지정하면 **-n soap-body-namspace** 인수도 필요합니다. 다른 모든 인수는 선택 사항이며 순서에 따라 나열할 수 있습니다.*



중요

***wsdl2soap** 은 문서/ 로깅 **SOAP 1.2** 바인딩 생성을 지원하지 않습니다.*

7.1.2. 예제

*시스템에 주문을 처리하고 주문을 처리하는 단일 작업을 제공하는 인터페이스가 있는 경우 예 7.1. “시스템 인터페이스 순서” 에 표시된 것과 유사한 **WSDL** 조각에 정의되어 있습니다.*

예 7.1. 시스템 인터페이스 순서

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

orderWidgets에 대해 생성된 SOAP 바인딩은 예 7.2. "orderWidgets에 대한 SOAP 1.2 Bindings"에 표시됩니다.

예 7.2. orderWidgets에 대한 SOAP 1.2 Bindings

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
      </input>
      <output name="bill">
        <wsoap12:body use="literal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
```

이 바인딩은 문서/ 전문 메시지 스타일을 사용하여 메시지를 전송하도록 지정합니다.

7.2. SOAP 1.2 메시지에 헤더 추가

7.2.1. 개요

SOAP 메시지 헤더는 **soap12:header** 요소를 **SOAP 1.2** 메시지에 추가하여 정의됩니다. **soap12:header** 요소는 바인딩의 입력,출력 및 **fault** 요소의 선택적 자식입니다. **SOAP** 헤더는 상위 메시지의 일부가 됩니다. **SOAP** 헤더는 메시지 및 메시지 부분을 지정하여 정의됩니다. 각 **SOAP** 헤더는 하나의 메시지 부분만 포함할 수 있지만 필요한 만큼의 헤더를 삽입할 수 있습니다.

7.2.2. 구문

SOAP 헤더 정의의 구문은 예 7.3. “Header Syntax” 에 표시됩니다.

예 7.3. Header Syntax

```
<binding name="headwig">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap12:operation soapAction="" style="documment"/>
      <input name="grain">
        <soap12:body ... />
        <soap12:header message="QName" part="partName"
                       use="literal|encoded"
                       encodingStyle="encodingURI"
                       namespace="namespaceURI" />
      </input>
...
</binding>
```

soap12:header 요소의 속성은 표 7.1. “soap12:header 특성s” 에 설명되어 있습니다.

표 7.1. soap12:header 특성s

속성	설명
message	헤더에 삽입되는 메시지의 정규화된 이름을 지정하는 필수 속성입니다.
part	SOAP 헤더에 삽입된 메시지 부분의 이름을 지정하는 필수 속성입니다.
Use	인코딩 규칙을 사용하여 메시지 파트를 인코딩할지 여부를 지정합니다. **인코딩** 으로 설정된 경우 message parts은 **encodingStyle** 속성 값으로 지정된 인코딩 규칙을 사용하여 인코딩됩니다. **리터럴** 로 설정된 경우 메시지 부분은 참조되는 스키마 유형에서 정의됩니다.

속성	설명
encodingStyle	메시지를 구성하는 데 사용되는 인코딩 규칙을 지정합니다.
namespace	**use="encoded"** 로 직렬화된 헤더 요소에 할당할 네임스페이스를 정의합니다.

7.2.3. 본문과 헤더 간 메시지 분할

***SOAP** 헤더에 삽입된 메시지 부분은 계약의 유효한 메시지 부분일 수 있습니다. **SOAP** 본문으로 사용되는 상위 메시지의 일부일 수도 있습니다. 동일한 메시지에서 정보를 두 번 보낼 가능성이 낮기 때문에 **SOAP 1.2** 바인딩은 **SOAP** 본문에 삽입되는 메시지 부분을 지정할 수 있는 수단을 제공합니다.*

***soap12:body** 요소에는 공백으로 구분된 부분 이름 목록을 사용하는 선택적 속성인 파트가 있습니다. 부분 도 정의되면 나열된 메시지 부분만 **SOAP 1.2** 메시지의 본문에 삽입됩니다. 그런 다음 나머지 부분을 메시지의 헤더에 삽입할 수 있습니다.*

참고

*상위 메시지의 일부를 사용하여 **SOAP** 헤더를 정의할 때 **Apache CXF**는 자동으로 **SOAP** 헤더를 채웁니다.*

7.2.4. 예제

*예 7.4. “SOAP 1.2 Binding with a SOAP Header” 예 7.1. “시스템 인터페이스 순서” 에 표시된 **orderWidgets** 서비스의 수정된 버전을 표시합니다. 이 버전은 각 주문에 요청 헤더와 응답에 **xsd:base64binary** 값이 배치되도록 수정되었습니다. 헤더는 **widgetKey** 메시지의 **keyVal** 파트로 정의됩니다. 이 경우 입력 또는 출력 메시지의 일부가 아니기 때문에 헤더를 생성하는 애플리케이션 논리를 추가합니다.*

예 7.4. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

    <types>
```

```
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal"/>
        <soap12:header message="tns:widgetKey" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

*예 7.4. “SOAP 1.2 Binding with a SOAP Header” 에 표시된 대로 헤더 값이 입력 및 출력 메시지의 일부임을 알 수 있습니다. 예 7.5. “SOAP Header을 사용한 orderWidgets에 대한 SOAP 1.2 Binding” 이 경우 **keyVal** 은 입력 및 출력 메시지의 일부입니다. **soap12:body** 요소에서 **parts** 속성이 **keyVal** 을 본문에 삽입하지 않도록 지정합니다. 그러나 헤더에 삽입됩니다.*

예 7.5. SOAP Header을 사용한 orderWidgets에 대한 SOAP 1.2 Binding

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap12:operation soapAction="" style="document"/>
      <input name="order">
        <soap12:body use="literal" parts="numOrdered"/>
        <soap12:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
        <soap12:body use="literal" parts="bill"/>
        <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
      </output>
      <fault name="sizeFault">
        <soap12:body use="literal"/>
      </fault>
  </operation>
```

```
</binding>
...
</definitions>
```

8장. 첨부 파일과 함께 SOAP를 사용하여 바이너리 데이터 전송

초록

SOAP 첨부 파일은 SOAP 메시지의 일부로 바이너리 데이터를 전송하는 메커니즘을 제공합니다. 첨부 파일과 함께 SOAP를 사용하려면 SOAP 메시지를 MIME 다중 파트 메시지로 정의해야 합니다.

8.1. 개요

SOAP 메시지는 일반적으로 바이너리 데이터를 포함하지 않습니다. 그러나 W3C SOAP 1.1 사양을 사용하면 MIME 다중 파트/관련 메시지를 사용하여 SOAP 메시지에서 바이너리 데이터를 보낼 수 있습니다. 이 기술을 첨부 파일과 함께 SOAP을 사용합니다. SOAP 첨부 파일은 첨부파일을 사용하여 W3C의 SOAP 메시지에 정의되어 있습니다.

8.2. 네임스페이스

MIME 다중 파트/ 관련 메시지를 정의하는 데 사용되는 WSDL 확장 기능은 네임스페이스 http://schemas.xmlsoap.org/wsdl/mime/ 에 정의되어 있습니다.

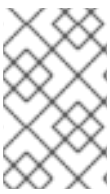
다음 설명에서는 이 네임스페이스가 mime 이 앞에 있다고 가정합니다. 이를 설정하는 WSDL 정의 요소의 항목은 예 8.1. “계약의 MIME 네임스페이스 사양” 에 표시됩니다.

예 8.1. 계약의 MIME 네임스페이스 사양

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

8.3. 메시지 바인딩 변경

기본 SOAP 바인딩에서 입력,출력 및 오류 요소의 첫 번째 하위 요소는 데이터를 나타내는 SOAP 메시지의 본문을 설명하는 soap:body 요소입니다. 첨부 파일과 함께 SOAP를 사용하면 soap:body 요소가 mime:multipartRelated 요소로 교체됩니다.



참고

WSDL은 오류 메시지에 대한 mime:multipartRelated 사용을 지원하지 않습니다.

mime:multipartRelated 요소는 Apache CXF에 메시지 본문이 바이너리 데이터를 포함하는 다중 부분

메시지임을 알립니다. 요소의 내용은 메시지 및 해당 콘텐츠의 부분을 정의합니다.
***MIME:multipartRelated** 요소에는 메시지의 개별 부분을 설명하는 하나 이상의 **mime:part** 요소가 포함되어 있습니다.*

*첫 번째 **mime:part** 요소에는 일반적으로 기본 **SOAP** 바인딩에 표시되는 **soap:body** 요소가 포함되어야 합니다. 나머지 **mime:part** 요소는 메시지에서 전송되는 첨부 파일을 정의합니다.*

8.4. MIME 다중 부분 메시지 설명

***MIME** 다중 파트 메시지는 여러 **mime:part** 요소를 포함하는 **mime:multipartRelated** 요소를 사용하여 설명합니다. **MIME** 다중 파트 메시지를 완전히 설명하려면 다음을 수행해야 합니다.*

1. ***MIME** 다중 부분으로 보내는 입력 또는 출력 메시지 내에 **mime:mulipartRelated** 요소를 **enclosing** 메시지의 첫 번째 하위 요소로 추가합니다.*

2. ***mime:part** 하위 요소를 **mime:multipartRelated** 요소에 추가하고 **name** 속성을 고유한 문자열로 설정합니다.*

3. ***soap:body** 요소를 **mime:part** 요소의 자식으로 추가하고 해당 속성을 적절하게 설정합니다.*

참고

*계약에 기본 **SOAP** 바인딩이 있는 경우 기본 바인딩에서 **MIME multipart** 메시지로 해당 메시지의 **soap:body** 요소를 복사할 수 있습니다.*

4. ***mime:part** 하위 요소를 **mime:multipartReleated** 요소에 추가하고 **name** 속성을 고유한 문자열로 설정합니다.*

5. ***mime:content** 하위 요소를 **mime:part** 요소에 추가하여 메시지의 이 부분의 내용을 설명합니다.*

***MIME** 메시지의 내용을 완전히 설명하기 위해 **mime:content** 요소에는 다음과 같은 속성이 있습니다.*

*표 **8.1. MIME:content** 속성*

속성	description +
part	바인딩에 배치되는 MIME 다중 **파트** 메시지의 콘텐츠로 사용되는 상위 메시지 정의에서 WSDL 메시지 파트의 이름을 지정합니다. +
type	이 메시지 부분에 있는 데이터의 MIME 유형입니다. MIME 유형은 구문 유형/하위 유형을 사용하여 type 및*sub type* 으로 정의됩니다. + **image/jpeg** 및 **text/plain** 과 같은 사전 정의된 MIME 유형이 많이 있습니다. MIME 유형은 IANA(Internet Assigned Numbers Authority)에 의해 유지 관리되며, MIME(Multipurpose Internet Mail Extensions)에 자세히 설명되어 있습니다. one: Internet Message Bodies 및 Multipurpose Internet Mail Extensions (MIME) Part two: 미디어 유형. +

6. *추가 **MIME** 부분의 경우 **[i303819]** 및 **[i303821]** 단계를 반복합니다.*

8.5. 예제

*예 8.2. "첨부파일과 함께 SOAP를 사용한 계약" 에서는 **JPEG** 형식으로 **X-ray**를 저장하는 서비스를 정의하는 **WSDL** 조각을 보여줍니다. 이미지 데이터인 **xRay** 는 **xsd:base64binary** 로 저장되며 **MIME multipart** 메시지의 두 번째 파트인 **imageData** 에 포장되어 있습니다. 입력 메시지의 나머지 두 부분인 **patientName** 및 **patientNumber** 는 **SOAP** 본문의 일부로 **MIME** 다중 파트 이미지의 첫 번째 부분으로 전송됩니다.*

예 8.2. 첨부파일과 함께 SOAP를 사용한 계약

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
```

```
  <part name="xRay" type="xsd:base64Binary"/>
 </message>
 <message name="storResponse">
  <part name="success" type="xsd:boolean"/>
 </message>

 <portType name="xRayStorage">
  <operation name="store">
   <input message="tns:storRequest" name="storRequest"/>
   <output message="tns:storResponse" name="storResponse"/>
  </operation>
 </portType>

 <binding name="xRayStorageBinding" type="tns:xRayStorage">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="store">
   <soap:operation soapAction="" style="document"/>
   <input name="storRequest">
    <mime:multipartRelated>
     <mime:part name="bodyPart">
      <soap:body use="literal"/>
     </mime:part>
     <mime:part name="imageData">
      <mime:content part="xRay" type="image/jpeg"/>
     </mime:part>
    </mime:multipartRelated>
   </input>
   <output name="storResponse">
    <soap:body use="literal"/>
   </output>
  </operation>
 </binding>

 <service name="xRayStorageService">
  <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
   <soap:address location="http://localhost:9000"/>
  </port>
 </service>
</definitions>
```

9장. SOAP MTOM을 사용하여 바이너리 데이터 전송

초록

SOAP Message Transmission Optimization Mechanism(MTOM)은 SOAP를 XML 메시지의 일부로 바이너리 데이터를 전송하는 메커니즘으로 교체합니다. Apache CXF와 함께 MTOM을 사용하려면 서비스의 계약에 올바른 스키마 유형을 추가하고 MTOM 최적화를 활성화해야 합니다.

9.1. MTOM 개요

SOAP Message Transmission Optimization Mechanism(MTOM)은 SOAP 메시지의 일부로 바이너리 데이터를 전송하기 위한 최적화된 방법을 지정합니다. 첨부 파일의 SOAP와 달리 MTOM은 바이너리 데이터를 전송하기 위해 XML-binary Optimized Packaging (XOP) 패키지를 사용해야 합니다. MTOM을 사용하여 바이너리 데이터를 전송하면 MIME Multipart/Related 메시지를 SOAP 바인딩의 일부로 완전히 정의할 필요가 없습니다. 그러나 다음과 같은 작업을 수행해야 합니다.

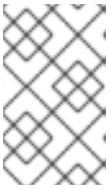
1. 첨부 파일로 보낼 데이터에 주석 을 답니다.

   WSDL 또는 데이터를 구현하는 Java 클래스에 주석을 달 수 있습니다.

2. 런타임의 MTOM 지원을 활성화합니다.

   이는 프로그래밍 방식으로 또는 구성을 통해 수행할 수 있습니다.

3. 첨부 파일로 전달되는 데이터에 대한 DataHandler 를 개발합니다.



참고

DataHandlerS 개발은 이 책의 범위를 벗어납니다.

9.2. MTOM을 사용하도록 데이터 유형 주석

9.2.1. 개요

WSDL에서 이미지 파일 또는 사운드 파일과 같은 바이너리 데이터 블록을 전달하는 데이터 유형을 정의할 때 xsd:base64Binary 유형의 데이터가 될 요소를 정의합니다. 기본적으로 xsd:base64Binary 형식의 모든 요소가 MTOM을 사용하여 직렬화될 수 있는 byte[] 가 생성됩니다. 그러나 코드 생성기의 기본 동

작은 직렬화를 최대한 활용하지 않습니다.**However, the default behavior of the code generators does not take full advantage of the serialization.**

MTOM을 완전히 활용하려면 서비스의 **WSDL** 문서 또는 바이너리 데이터 구조를 구현하는 **JAXB** 클래스에 주석을 추가해야 합니다. **WSDL** 문서에 주석을 추가하면 코드 생성기가 바이너리 데이터에 대한 스트리밍 데이터 처리기를 생성하도록 강제 적용합니다. **JAXB** 클래스에 주석을 달려면 적절한 콘텐츠 유형을 지정해야 하며 바이너리 데이터가 포함된 필드의 유형 사양을 변경해야 할 수도 있습니다.

9.2.2. WSDL 첫 번째

예 9.1. "MTOM에 대한 메시지" 하나의 문자열 필드, 하나의 정수 필드 및 바이너리 필드를 포함하는 메시지를 사용하는 웹 서비스의 **WSDL** 문서를 보여줍니다. 바이너리 필드는 큰 이미지 파일을 전송하기 위한 것이므로 일반 **SOAP** 메시지의 일부로 보내는 것은 적절하지 않습니다.

예 9.1. MTOM에 대한 메시지

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:xsd1="http://mediStor.org/types/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <schema targetNamespace="http://mediStor.org/types/"
          xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary" />
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>

  <message name="storRequest">
    <part name="record" element="xsd1:xRay"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>

  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
```

```
    </operation>
  </portType>

  <binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
    <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
      <soap12:operation soapAction="" style="document"/>
      <input name="storRequest">
        <soap12:body use="literal"/>
      </input>
      <output name="storResponse">
        <soap12:body use="literal"/>
      </output>
    </operation>
  </binding>
  ...
</definitions>
```

MTOM을 사용하여 메시지의 바이너리 부분을 최적화된 첨부 파일로 보내려면 바이너리 데이터가 포함된 요소에 **xmime:expectedContentTypes** 속성을 추가해야 합니다. 이 속성은 **http://www.w3.org/2005/05/xmlmime** 네임스페이스에 정의되어 있으며 요소가 포함할 것으로 예상되는 **MIME** 유형을 지정합니다. **MIME** 유형의 쉼표로 구분된 목록을 지정할 수 있습니다. 이 속성의 설정은 코드 생성기가 데이터에 대한 **JAXB** 클래스를 만드는 방법을 변경합니다. 대부분의 **MIME** 유형의 경우 코드 생성기는 **DataHandler**를 생성합니다. 이미지용과 같은 일부 **MIME** 유형에는 매핑이 정의되어 있습니다.

참고

MIME 유형은 **IANA(Internet Assigned Numbers Authority)**에 의해 유지 관리되며, **MIME(Multipurpose Internet Mail Extensions)**에 자세히 설명되어 있습니다. **part One: Internet Message Bodies** 및 **Multipurpose Internet Mail Extensions (MIME) Part two: 미디어 유형 2.**

대부분의 경우 **application/octet-stream** 을 지정합니다.

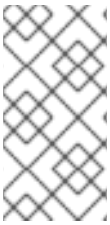
예 9.2. "MTOM용 바이너리 데이터" **MTOM** 사용을 위해 **예 9.1. "MTOM에 대한 메시지"** 에서 **xRayType** 을 수정하는 방법을 보여줍니다.

예 9.2. MTOM용 바이너리 데이터

```
...
  <types>
    <schema targetNamespace="http://mediStor.org/types/"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
      <complexType name="xRayType">
```

```
      <sequence>
        <element name="patientName" type="xsd:string" />
        <element name="patientNumber" type="xsd:int" />
        <element name="imageData" type="xsd:base64Binary"
              xmime:expectedContentTypes="application/octet-stream"/>
      </sequence>
    </complexType>
    <element name="xRay" type="xsd1:xRayType" />
  </schema>
 </types>
...
```

xRayType 에 대해 생성된 **JAXB** 클래스에는 더 이상 바이트[] 가 포함되어 있지 않습니다. 대신 코드 생성기가 **xmime:expectedContentTypes** 특성을 확인하고 **imageData** 필드에 대한 **DataHandler**를 생성합니다.



참고

MTOM을 사용하기 위해 바인딩 요소를 변경할 필요가 없습니다. 런타임은 데이터가 전송될 때 적절한 변경을 수행합니다.

9.2.3. Java 첫 번째

Java 첫 개발을 수행하는 경우 다음을 수행하여 **JAXB** 클래스 **MTOM**을 준비시킬 수 있습니다.

1. 바이너리 데이터를 보유한 필드가 **DataHandler**인지 확인합니다.

2. 스트리밍하려는 데이터가 **MTOM** 첨부 파일로 포함된 필드에 **@XmlMimeType()** 주석을 추가합니다.

예 9.3. “MTOM용 JAXB 클래스” **MTOM** 사용에 대한 주석이 있는 **JAXB** 클래스를 표시합니다.

예 9.3. MTOM용 JAXB 클래스

```
@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
```

```
    protected DataHandler imageData;
  ...
}
```

9.3. MTOM 활성화

기본적으로 **Apache CXF** 런타임은 **MTOM** 지원을 활성화하지 않습니다. 모든 바이너리 데이터를 일반 **SOAP** 메시지의 일부로 전송하거나 최적화되지 않은 첨부 파일로 보냅니다. 프로그래밍 방식으로 또는 구성을 통해 **MTOM** 지원을 활성화할 수 있습니다.

9.3.1. JAX-WS API 사용

9.3.1.1. 개요

서비스 공급자와 소비자 모두 **MTOM** 최적화를 활성화해야 합니다. **JAX-WS API**는 각 엔드포인트 유형에 대해 서로 다른 메커니즘을 제공합니다.

9.3.1.2. 서비스 공급자

JAX-WS API를 사용하여 서비스 공급자를 게시한 경우 다음과 같이 런타임의 **MTOM** 지원을 활성화합니다.

1. 게시된 서비스의 **Endpoint** 오브젝트에 액세스합니다.

   Endpoint 오브젝트에 액세스하는 가장 쉬운 방법은 끝점을 게시하는 것입니다. 자세한 내용은 **31장. 서비스 게시** 에서 참조하십시오.

2. 예 9.4. "끝점에서 SOAP 바인딩 가져오기" 과 같이 **getBinding()** 메서드를 사용하여 끝점에서 **SOAP** 바인딩을 가져옵니다.

   예 9.4. 끝점에서 SOAP 바인딩 가져오기

   ```
   // Endpoint ep is declared previously
   SOAPBinding binding = (SOAPBinding)ep.getBinding();
   ```

   반환된 바인딩 개체를 **MTOM** 속성에 액세스하려면 **SOAPBinding** 개체에 캐스팅해야 합니다.

3. 바인딩의 **MTOM enabled** 속성을 *예 9.5. "서비스 공급자의 MTOM 사용 속성 설정"* 와 같이 **setMTOMEnabled()** 메서드를 사용하여 **true** 로 설정합니다.

예 9.5. 서비스 공급자의 MTOM 사용 속성 설정

```
binding.setMTOMEnabled(true);
```

9.3.1.3. 소비자

MTOM으로 **JAX-WS** 소비자를 활성화하려면 다음을 수행해야 합니다.

1. 소비자의 프록시를 **BindingProvider** 개체로 캐스팅합니다.

소비자 프록시를 가져오는 방법에 대한 자세한 내용은 *25장. WSDL 계약 없이 소비자 개발* 또는 *28장. WSDL 계약에서 소비자 개발* 을 참조하십시오.

2. *예 9.6. "BindingProvider에서 SOAP 바인딩 가져오기"* 과 같이 **getBinding()** 메서드를 사용하여 **BindingProvider** 에서 **SOAP** 바인딩을 가져옵니다.

예 9.6. BindingProvider에서 SOAP 바인딩 가져오기

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. 바인딩의 **setMTOMEnabled()** 메서드를 사용하여 *예 9.7. "소비자의 MTOM 사용 속성 설정"* 에 표시된 대로 **MTOM enabled** 속성을 **true** 로 설정합니다.

예 9.7. 소비자의 MTOM 사용 속성 설정

```
binding.setMTOMEnabled(true);
```

9.3.2. 구성 사용

9.3.2.1. 개요

*컨테이너에 배포할 때와 같이 **XML**을 사용하여 서비스를 게시하는 경우 엔드포인트 구성 파일에서 끝점의 **MTOM** 지원을 활성화할 수 있습니다. 엔드 포인트 구성에 대한 자세한 내용은 IV 부. 웹 서비스 엔드 포인트 구성 을 참조하십시오.*

9.3.2.2. 절차

***MTOM** 속성은 엔드포인트의 **jaxws:endpoint** 요소 내에 설정됩니다. **MTOM**을 활성화하려면 다음을 수행합니다.*

1. ***jaxws:property** 하위 요소를 엔드포인트의 **jaxws:endpoint** 요소에 추가합니다.*
2. ***jaxws:property** 요소에 항목 하위 요소를 추가합니다.*
3. ***entry** 요소의 **key** 속성을 **mtom-enabled** 로 설정합니다.*
4. ***entry** 요소의 **value** 속성을 **true** 로 설정합니다.*

9.3.2.3. 예제

*예 9.8. “MTOM 활성화를 위한 구성” **MTOM**이 활성화된 끝점을 보여줍니다.*

*예 9.8. **MTOM** 활성화를 위한 구성*

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                  http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">

  <jaxws:endpoint id="xRayStorage"
                implementor="demo.spring.xRayStorImpl"
                address="http://localhost/xRayStorage">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```

10장. XML 문서 사용

초록

순수 XML 페이로드 형식은 SOAP의 오버헤드 없이 간단한 XML 문서를 사용하여 데이터를 교환할 수 있도록 하여 SOAP 바인딩에 대한 대안을 제공합니다.

10.1. XML 바인딩 네임스페이스

XML 형식 바인딩을 설명하는 데 사용되는 확장 기능은 네임스페이스 http://cxf.apache.org/bindings/xformat 에 정의되어 있습니다. Apache CXF 툴은 xformat 접두사를 사용하여 XML 바인딩 확장을 나타냅니다. 계약에 다음 줄을 추가합니다.Add the following line to your contract:

```
xmlns:xformat="http://cxf.apache.org/bindings/xformat"
```

10.2. 수동 편집

인터페이스를 순수 XML 페이로드 형식으로 매핑하려면 다음을 수행합니다.

1. 네임스페이스 선언을 추가하여 XML 바인딩을 정의하는 확장을 포함합니다. "XML 바인딩 네임스페이스" 을 참조하십시오.

2. 표준 WSDL 바인딩 요소를 계약에 추가하여 XML 바인딩을 보유하고, 바인딩에 고유한 이름을 지정한 다음, 바인딩되는 인터페이스를 나타내는 WSDL portType 요소의 이름을 지정합니다.

3. xformat:binding 하위 요소를 바인딩 요소에 추가하여 메시지가 SOAP 봉투 없이 순수 XML 문서로 처리되는지 확인합니다.

4. 필요한 경우 xformat:binding 요소의 rootNode 속성을 유효한 QName으로 설정합니다. rootNode 특성의 영향에 대한 자세한 내용은 "글의 XML 메시지" 을 참조하십시오.

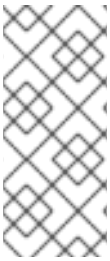
5. 바인딩된 인터페이스에 정의된 각 작업에 대해 작업 메시지에 대한 바인딩 정보를 보유하는 표준 WSDL 작업 요소를 추가합니다.

6. 바인딩에 추가된 각 작업에 대해 입력 , 출력 및 오류 자식 요소를 추가하여 작업에서 사용하

는 메시지를 나타냅니다.**For each operation added to the binding, add the input,output, and fault children elements to represent the messages used by the operation.**

이러한 요소는 논리 작업의 인터페이스 정의에 정의된 메시지에 해당합니다.

7. 선택적으로 유효한 **rootNode** 속성이 있는 **xformat:body** 요소를 추가된 입력,출력, **fault** 요소에 추가하여 바인딩 수준에서 설정된 **rootNode** 의 값을 재정의합니다.



참고

메시지에 **void**를 반환하는 작업에 대한 출력 메시지와 같이 어떤 메시지가 아무런 부분도 없는 경우, 전선에 작성된 메시지가 유효한지 확인하도록 메시지의 **rootNode** 특성을 설정해야 합니다.

10.3. 글의 XML 메시지

인터페이스의 메시지가 **SOAP** 봉투 없이 **XML** 문서로 전달되도록 지정하는 경우 메시지가 전선에 기록될 때 유효한 **XML** 문서를 형성하도록 해야 합니다. 또한 **XML** 문서를 수신하는 **Apache CXF** 이외의 사용자가 **Apache CXF**가 생성한 메시지를 이해하고 있는지 확인해야 합니다.

두 문제를 해결하는 간단한 방법은 글로벌 **xformat:binding** 요소 또는 개별 메시지의 **xformat:body** 요소에서 선택적 **rootNode** 특성을 사용하는 것입니다. **rootNode** 속성은 **Apache CXF**에서 생성된 **XML** 문서에 대한 루트 노드 역할을 하는 요소의 **QName**을 지정합니다. **rootNode** 속성이 설정되지 않은 경우 **Apache CXF**는 **doc** 스타일 메시지를 사용할 때 메시지 부분의 루트 요소를 루트 요소로 사용하거나 **rpc** 스타일 메시지를 사용할 때 메시지 파트 이름을 루트 요소로 사용합니다.

예를 들어 **rootNode** 속성이 *예 10.1. “유효한 XML 바인딩 메시지”* 에 정의된 메시지를 설정하지 않으면 루트 요소 **lineNumber** 를 사용하여 **XML** 문서를 생성합니다.

예 10.1. 유효한 XML 바인딩 메시지

```
<type ... >
  ...
  <element name="operatorID" type="xsd:int"/>
  ...
</types>
<message name="operator">
  <part name="lineNumber" element="ns1:operatorID"/>
</message>
```

한 부분이 있는 메시지의 경우, **Apache CXF**는 **rootNode** 속성이 설정되지 않은 경우에도 항상 유효한 **XML** 문서를 생성합니다. 그러나 *예 10.2. "잘못된 XML 바인딩 메시지"* 의 메시지는 유효하지 않은 **XML** 문서를 생성합니다.

예 10.2. 잘못된 XML 바인딩 메시지

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>

<message name="matildas">
  <part name="dancing" element="ns1:pairName"/>
  <part name="number" element="ns1:entryNum"/>
</message>
```

XML 바인딩에 지정된 **rootNode** 특성이 없으면 **Apache CXF**는 *예 10.2. "잘못된 XML 바인딩 메시지"* 로 정의된 메시지에 대해 *예 10.3. "잘못된 XML 문서"* 와 유사한 **XML** 문서를 생성합니다. 생성된 **XML** 문서는 **pairName** 및 **entryNum** 의 두 가지 루트 요소가 있기 때문에 유효하지 않습니다.

예 10.3. 잘못된 XML 문서

```
<pairName>
  Fred&Linda
</pairName>
<entryNum>
  123
</entryNum>
```

rootNode 특성을 설정하면 *예 10.4. "rootNode 세트가 있는 XML 바인딩"* **Apache CXF**에서 지정된 루트 요소에 요소를 래핑합니다. 이 예제에서 **rootNode** 특성은 전체 바인딩에 대해 정의되며 루트 요소 이름이 **entrants**로 지정되도록 지정합니다.

예 10.4. rootNode 세트가 있는 XML 바인딩

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
  </operation>
</portType>

<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
```

```
    <input name="contestant"/>
    <output name="entered"/>
  </binding>
```

입력 메시지에서 생성된 **XML** 문서는 *예 10.5. "rootNode 특성을 사용하여 생성된 XML 문서"* 과 유사합니다. **XML** 문서에는 이제 하나의 루트 요소만 있습니다.

예 10.5. rootNode 특성을 사용하여 생성된 XML 문서

```
<entrants>
  <pairName>
    Fred&Linda
  <entryNum>
    123
  </entryNum>
</entrants>
```

10.4. 바인딩의 ROOTNODE 특성 설정 덮어쓰기

각 개별 메시지에 대해 **rootNode** 특성을 설정하거나 메시지 바인딩 내부의 **xformat:body** 요소를 사용하여 특정 메시지에 대한 전역 설정을 덮어쓸 수도 있습니다. 예를 들어 *예 10.4. "rootNode 세트가 있는 XML 바인딩"* 에 정의된 출력 메시지가 입력 메시지와 다른 루트 요소를 갖도록 하려면 *예 10.6. "xformat:body사용"* 에 표시된 대로 바인딩의 루트 요소를 재정의할 수 있습니다.

예 10.6. xformat:body사용

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant"/>
    <output name="entered">
      <xformat:body rootNode="entryStatus" />
    </output>
  </operation>
</binding>
```

III 부. 웹 서비스 전송

*이 부분에서는 **Apache CXF** 전송을 **WSDL** 문서에 추가하는 방법을 설명합니다.*

11장. ENDPOINTS가 WSDL에서 정의되는 방법 이해

초록

끝점은 인스턴스화된 서비스를 나타냅니다. 바인딩과 끝점을 노출하는 데 사용되는 네트워킹 세부 정보를 결합하여 정의합니다.

11.1. 개요

엔드포인트는 서비스의 물리적 표현으로 간주될 수 있습니다. 서비스에서 사용하는 논리 데이터의 물리적 표현과 다른 엔드포인트에서 서비스를 연결하는 데 사용되는 물리적 연결 세부 정보를 정의하는 네트워킹 세부 정보 집합을 지정하는 바인딩을 결합합니다.

참고

CXF 공급자는 클라이언트에 해당하는 **CXF** 소비자의 서버입니다. **CXF(camel-cxf)** 구성 요소를 경로에서 시작 끝점으로 사용하는 경우 엔드포인트는 **Camel** 소비자와 **CXF** 공급자입니다. **Camel CXF** 구성 요소를 경로에서 종료 끝점으로 사용하는 경우 엔드포인트는 **Camel** 생산자와 **CXF** 소비자입니다.

11.2. 엔드 포인트 및 서비스

바인딩이 단일 인터페이스만 매핑할 수 있는 것과 동일한 방식으로 끝점은 단일 서비스에만 매핑할 수 있습니다. 그러나 서비스를 여러 끝점으로 표시할 수 있습니다. 예를 들어, 4개의 엔드포인트에서 표시되는 티켓 판매 서비스를 정의할 수 있습니다. 그러나 티켓 판매 서비스와 위젯 판매 서비스를 모두 나타내는 단일 엔드 포인트가 없었습니다.

11.3. WSDL 요소

엔드포인트는 **WSDL** 서비스 요소와 **WSDL** 포트 요소의 조합을 사용하여 계약에 정의됩니다. 서비스 요소는 관련 포트 요소의 컬렉션입니다. 포트 요소는 실제 엔드포인트를 정의합니다.

WSDL 서비스 요소에는 고유한 이름을 지정하는 단일 속성 이름 가 있습니다. 서비스 요소는 관련 포트 요소의 컬렉션의 부모 요소로 사용됩니다. **WSDL**은 포트 요소가 어떻게 관련되어 있는지에 대한 사양을 만들지 않습니다. 필요에 따라 포트 요소를 연결할 수 있습니다.**You can associate the port elements in any manner you see fit.**

WSDL 포트 요소에는 바인딩 특성이 있습니다. 이는 끝점에서 사용하는 바인딩을 지정하고 **wsdl:binding** 요소에 대한 참조입니다. 또한 모든 포트 간에 고유한 이름 을 제공하는 필수 속성인 **name**

*속성도 포함합니다. **port** 요소는 끝점에서 사용하는 실제 전송 세부 정보를 지정하는 요소의 부모 요소입니다. 전송 세부 정보를 지정하는 데 사용되는 요소는 다음 섹션에서 설명합니다.*

11.4. 계약에 끝점 추가

__Apache CXF__는 사전 정의된 서비스 인터페이스 및 바인딩 조합에 대해 끝점을 생성할 수 있는 명령줄 도구를 제공합니다.

이 도구는 귀하의 계약에 적절한 요소를 추가합니다. 그러나 끝점 작업을 정의하는 데 사용되는 다양한 전송 방법을 알고 있는 것이 좋습니다.

텍스트 편집기를 사용하여 계약에 끝점을 추가할 수도 있습니다. 계약을 직접 편집하면 계약이 유효한지 확인할 책임이 있습니다.

11.5. 지원되는 전송

엔드포인트 정의는 각 전송 __Apache CXF__에 대해 정의된 확장을 사용하여 빌드됩니다. 여기에는 다음 전송이 포함됩니다.

- *__HTTP__*
- *__CORBA__*
- *__Java Messaging Service__*

12장. HTTP 사용

초록

HTTP는 웹의 기본 전송입니다. 엔드포인트 간에 통신할 수 있도록 표준화되고 강력하고 유연한 플랫폼을 제공합니다. 이러한 요인으로 인해 대부분의 WS-* 사양에 대한 전송이 가정되고 RESTful 아키텍처에 통합되어 있습니다.

12.1. 기본 HTTP 끝점 추가

12.1.1. 대체 HTTP 런타임

Apache CXF는 다음과 같은 대체 HTTP 런타임 구현을 지원합니다.

- Undertow.에 대한 자세한 내용은 12.4절. “Undertow 런타임 구성” 에 자세히 설명되어 있습니다.
- Netty.NET은 12.5절. “Netty 런타임 구성” 에 자세히 설명되어 있습니다.

12.1.2. Netty HTTP URL

일반적으로 HTTP 엔드포인트는 클래스 경로에 포함된 모든 HTTP 런타임(Undertow 또는 Netty)을 사용합니다. Undertow 런타임과 Netty 런타임이 모두 classpath에 포함된 경우, 기본적으로 Undertow 런타임이 사용되므로 Netty 런타임을 사용할 때 명시적으로 지정해야 합니다.

classpath에서 두 개 이상의 HTTP 런타임을 사용할 수 있는 경우 다음 형식을 갖도록 끝점 URL을 지정하여 Undertow 런타임을 선택할 수 있습니다.

```
netty://http://RestOfURL
```

12.1.3. 페이로드 유형

사용 중인 페이로드 형식에 따라 HTTP 끝점의 주소를 지정하는 세 가지 방법이 있습니다.

- SOAP 1.1은 표준화된 soap:address 요소를 사용합니다.
- SOAP 1.2는 soap12:address 요소를 사용합니다.
- 다른 모든 페이로드 형식은 http:address 요소를 사용합니다.

참고

Camel 2.16.0 릴리스에서 Apache Camel CXF Payload는 즉시 스트림 캐시를 지원합니다.

12.1.4. SOAP 1.1

HTTP를 통해 SOAP 1.1 메시지를 전송하는 경우 SOAP 1.1 주소 요소를 사용하여 끝점의 주소를 지정해야 합니다. 여기에는 끝점의 주소를 URL로 지정하는 하나의 속성 위치 가 있습니다. SOAP 1.1 address 요소는 네임스페이스 http://schemas.xmlsoap.org/wsdl/soap/ 에 정의되어 있습니다.

예 12.1. “SOAP 1.1 포트 요소” HTTP를 통해 SOAP 1.1 메시지를 보내는 데 사용되는 포트 요소를 표시합니다.

예 12.1. SOAP 1.1 포트 요소

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

12.1.5. SOAP 1.2

HTTP를 통해 SOAP 1.2 메시지를 전송하는 경우 SOAP 1.2 주소 요소를 사용하여 끝점의 주소를 지정해야 합니다. 여기에는 끝점의 주소를 URL로 지정하는 하나의 속성 위치 가 있습니다. SOAP 1.2 address 요소는 네임스페이스 http://schemas.xmlsoap.org/wsdl/soap12/ 에 정의되어 있습니다.

예 12.2. “SOAP 1.2 포트 요소” HTTP를 통해 SOAP 1.2 메시지를 보내는 데 사용되는 포트 요소를 표시합니다.

예 12.2. SOAP 1.2 포트 요소

```
<definitions ...
            xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ... >
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

12.1.6. 기타 메시지 유형

메시지가 SOAP 이외의 페이로드 형식으로 매핑되면 HTTP 주소 요소를 사용하여 끝점 주소 를 지정해야 합니다. 여기에는 끝점의 주소를 URL로 지정하는 하나의 속성 위치 가 있습니다. HTTP 주소 요소는 네임스페이스 http://schemas.xmlsoap.org/wsdl/http/ 에 정의되어 있습니다.

예 12.3. “HTTP 포트 요소” XML 메시지를 보내는 데 사용되는 포트 요소를 표시합니다.

예 12.3. HTTP 포트 요소

```
<definitions ...
            xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

12.2. 소비자 구성

12.2.1. HTTP Consumer Endpoints 관련 메커니즘

HTTP 소비자 끝점은 끝점에서 자동으로 리디렉션 응답을 수락하는지 여부, 청크를 사용할 수 있는지 여부, 엔드포인트에서 keep-alive를 요청할지 여부 및 엔드포인트가 프록시와 상호 작용하는 방식을 포함

하여 여러 HTTP 연결 속성을 지정할 수 있습니다. HTTP 연결 속성 외에도 HTTP 소비자 끝점은 보안 방법을 지정할 수 있습니다.

소비자 끝점은 다음 두 가지 메커니즘을 사용하여 구성할 수 있습니다.

- 설정
- WSDL

12.2.2. 구성 사용

12.2.2.1. 네임스페이스

HTTP 소비자 끝점을 구성하는 데 사용되는 요소는 네임스페이스 http://cxf.apache.org/transports/http/configuration 에 정의되어 있습니다. 일반적으로 접두사 http-conf 를 사용합니다. HTTP 구성 요소를 사용하려면 예 12.4. "HTTP Consumer 구성 네임스페이스" 에 표시된 줄을 끝점 구성 파일의 빈 요소에 추가해야 합니다. 또한 구성 요소의 네임스페이스를 xsi:schemaLocation 속성에 추가해야 합니다.

예 12.4. HTTP Consumer 구성 네임스페이스

```
<beans ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/http/configuration
                            http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          ...">
```

12.2.2.2. Undertow 런타임 또는 Netty 런타임

http-conf 네임스페이스의 요소를 사용하여 Undertow 런타임 또는 Netty 런타임을 구성할 수 있습니다.

12.2.2.3. 구성 요소

http-conf:conduit 요소와 해당 하위 항목을 사용하여 HTTP 소비자 끝점을 구성합니다. http-

conf:conduit 요소는 엔드포인트에 해당하는 WSDL 포트 요소를 지정하는 단일 속성인 name 을 사용합니다. name 속성의 값은 portQName'.http-conduit' 형식을 사용합니다. 예 12.5. "HTTP-conf:conduit Element" 는 끝점의 대상 네임스페이스가 http://widgets.widgetvendor.net인 경우의 WSDL 조각 < port binding="widgetSOAPBinding" name="widgetSOAPPort >에 의해 지정된 엔드포인트에 대한 구성을 추가하는 데 사용되는 http-conf:conduit 요소를 표시합니다.

예 12.5. HTTP-conf:conduit Element

```
...
  <http-conf:conduit name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-conduit">
    ...
  </http-conf:conduit>
...
```

http-conf:conduit 요소에는 구성 정보를 지정하는 하위 요소가 있습니다. 자세한 내용은 표 12.1. "HTTP Consumer Endpoint를 구성하는 데 사용되는 요소" 에 설명되어 있습니다.

표 12.1. HTTP Consumer Endpoint를 구성하는 데 사용되는 요소

요소	설명
http-conf:client	시간 초과, 유지 요청, 콘텐츠 유형 등과 같은 HTTP 연결 속성을 지정합니다.Specifies the HTTP connection properties such as timeouts, keep-alive requests, content types, etc. "클라이언트 요소" 을 참조하십시오.
http-conf:authorization	끝점에서 선점하여 사용하는 기본 인증 방법을 구성하기 위한 매개변수를 지정합니다. 가장 선호되는 방법은 **http-conf:basicAuthSupplier** 오브젝트를 제공하는 것입니다.
http-conf:proxyAuthorization	발신 HTTP 프록시 서버에 대한 기본 인증을 구성하는 매개변수를 지정합니다.
http-conf:tlsClientParameters	SSL/TLS를 구성하는 데 사용되는 매개변수를 지정합니다.
http-conf:basicAuthSupplier	끝점에서 사용하는 기본 인증 정보를 제공하는 오브젝트의 빈 참조 또는 클래스 이름을 선점적으로 또는 401 HTTP 챌린지에 대한 응답으로 지정합니다.
http-conf:trustDecider	정보를 전송하기 전에 HTTP(S) **URLConnection** 오브젝트를 검사하여 HTTPS 서비스 공급자와의 연결을 설정하는 오브젝트의 빈 참조 또는 클래스 이름을 지정합니다.

12.2.2.4. 클라이언트 요소

http-conf:client 요소는 소비자 엔드포인트의 HTTP 연결의 비보안 속성을 구성하는 데 사용됩니다. 표 12.2. "HTTP 소비자 구성 속성" 에 설명된 해당 속성은 연결의 속성을 지정합니다.

표 12.2. HTTP 소비자 구성 속성

속성	설명
ConnectionTimeout	시간이 초과되기 전에 소비자가 연결을 설정하려고 시도하는 시간(밀리초)을 지정합니다. 기본값은 **30000** 입니다. **0** 은 소비자가 요청을 무기한 계속 보내도록 지정합니다.
ReceiveTimeout	시간이 초과되기 전에 소비자가 응답을 대기할 시간(밀리초)을 지정합니다. 기본값은 **30000** 입니다. **0** 은 소비자가 무기한 기다리도록 지정합니다.
AutoRedirect	소비자가 발급한 서버를 자동으로 따르는지 여부를 지정합니다. 기본값은 **false**입니다.
MaxRetransmits	소비자가 리디렉션을 충족하기 위해 요청을 다시 전송할 최대 횟수를 지정합니다. 기본값은 **-1** 이며 무제한 재전송이 허용되도록 지정합니다.
AllowChunking	사용자가 청크를 사용하여 요청을 보낼지 여부를 지정합니다. 기본값은 사용자가 요청을 보낼 때 청크를 사용하도록 지정하는 **true** 입니다. 다음 중 하나라도 해당하는 경우 청크를 사용할 수 없습니다. ● **HTTP-conf:basicAuthSupplier** 는 인증 정보를 사전에 제공하도록 구성되어 있습니다. ● **AutoRedirect** 는 **true** 로 설정됩니다. 두 경우 모두 **AllowChunking** 값이 무시되고 청크는 허용되지 않습니다.
accept	소비자가 처리할 준비가 된 미디어 유형을 지정합니다. 이 값은 HTTP **Accept** 속성의 값으로 사용됩니다. 특성 값은MIME(Multipurpose Internet mail extensions) 유형을 사용하여 지정됩니다.

속성	설명
AcceptLanguage	소비자가 응답을 받기 위해 선호하는 언어(예: 미국 영어)를 지정합니다. 이 값은 HTTP **AcceptLanguage** 속성 값으로 사용됩니다. 언어 태그는 International Organization for Standards(ISO)에 의해 규제되며 일반적으로 ISO-639 표준에 따라 결정된 언어 코드와 하이픈으로 구분된 ISO-3166 표준에 따라 국가 코드를 결합하여 구성됩니다. 예를 들어 en-US는 미국어를 나타냅니다.
AcceptEncoding	소비자가 처리할 준비가 된 콘텐츠 인코딩을 지정합니다. 콘텐츠 인코딩 레이블은 IANA(Internet Assigned Numbers Authority)에 의해 규제됩니다. 이 값은 HTTP **AcceptEncoding** 속성 값으로 사용됩니다.
ContentType	메시지의 본문에 전송되는 데이터의 미디어 유형을 지정합니다.Specifies the media type of the data being sent in the body of a message. 미디어 유형은 다목적 인터넷 메일 확장 (MIME) 유형을 사용하여 지정됩니다. 값은 HTTP **ContentType** 속성 값으로 사용됩니다. 기본값은 **text/xml** 입니다. 웹 서비스의 경우 이 값을 **text/xml** 로 설정해야 합니다. 클라이언트가 CGI 스크립트로 HTML 양식 데이터를 전송하는 경우 이를 **application/x-www-form-urlencoded** 로 설정해야 합니다. HTTP POST 요청이 SOAP와 반대되는 고정 페이로드 형식에 바인딩된 경우 일반적으로 콘텐츠 유형이 **애플리케이션/octet-stream** 으로 설정됩니다.
호스트	요청이 호출되는 리소스의 인터넷 호스트 및 포트 번호를 지정합니다. 이 값은 HTTP **Host** 속성 값으로 사용됩니다. 이 속성은 일반적으로 필요하지 않습니다. 이는 특정 DNS 시나리오 또는 애플리케이션 설계에만 필요합니다. 예를 들어, 동일한 인터넷 프로토콜(IP) 주소에 매핑되는 가상 서버의 경우 클라이언트가 클러스터를 선호하는 호스트를 나타냅니다.

속성	설명
연결	각 요청/응답 대화 상자 이후 특정 연결을 열린 상태로 유지할지 여부를 지정합니다.Specifies whether a particular connection is to be kept open or closed after each request/response dialog. 두 가지 유효한 값이 있습니다. • **keep-Alive** - 소비자가 초기 요청/응답 시퀀스 후 연결을 열린 상태로 설정하도록 지정합니다. 서버가 이를 수락하면 소비자가 이를 닫을 때까지 연결이 열려 있습니다. • **닫기**(기본값) - 요청/응답 시퀀스 후에 서버에 대한 연결이 종료되도록 지정합니다.
CacheControl	소비자의 서비스 공급자 요청을 포함하는 체인과 관련된 캐시에 의해 준수해야 하는 동작에 대한 지시문을 지정합니다.Specifies directives about the behavior that must be adhered to by caches involved in the chain consisting a request from a consumer to a service provider. 12.2.4절. "소비자 캐시 제어 지침"을 참조하십시오.
쿠키	모든 요청과 함께 보낼 정적 쿠키를 지정합니다.
BrowserType	요청이 시작되는 브라우저에 대한 정보를 지정합니다. 월드 와이드 웹 컨소시엄 (W3C)의 HTTP 사양에서 이것을 **user-agent** 라고도 합니다. 일부 서버는 요청을 전송하는 클라이언트에 따라 최적화됩니다.
Referer	소비자가 특정 서비스에 대한 요청을 수행하도록 지시하는 리소스의 URL을 지정합니다. 값은 HTTP **Referer** 속성의 값으로 사용됩니다. 이 HTTP 속성은 요청이 URL을 입력하는 대신 하이퍼링크를 클릭하는 브라우저 사용자의 결과인 경우에 사용됩니다. 이를 통해 서버는 이전 작업 흐름에 따라 처리를 최적화하고, 로깅, 최적화된 캐싱, 사용되지 않는 링크 추적 등의 목적으로 리소스에 백 링크 목록을 생성할 수 있습니다. 그러나 일반적으로 웹 서비스 애플리케이션에서는 사용되지 않습니다. **AutoRedirect** 특성이 **true** 로 설정되고 요청이 리디렉션되면 **Referer** 속성에 지정된 모든 값이 재정의됩니다. HTTP Referer 속성의 값은 소비자의 원래 요청을 리디렉션하는 서비스의 URL로 설정됩니다.

속성	설명
DecoupledEndpoint	별도의 공급자를 통해 응답을 받기 위한 분리된 엔드포인트의 URL을 지정합니다.Specifies the URL of a decoupled endpoint for the received of responses over a separate providerconsumer connection. 분리된 엔드포인트 사용에 대한 자세한 내용은 12.6절. "분리된 모드에서 HTTP 전송 사용" 에서 참조하십시오. 분리된 끝점에 대해 WS-Addressing을 사용하도록 소비자 끝점과 서비스 공급자 끝점을 모두 구성해야 합니다.
ProxyServer	라우팅되는 요청을 통과하는 프록시 서버의 URL을 지정합니다.
ProxyServerPort	라우팅되는 요청을 통한 프록시 서버의 포트 번호를 지정합니다.
ProxyServerType	요청을 라우팅하는 데 사용되는 프록시 서버의 유형을 지정합니다. 유효한 값은 다음과 같습니다. • **HTTP**(기본값) • **SOCKS**

12.2.2.5. 예제

*예 12.6. "HTTP Consumer Endpoint 구성" 는 호출당 한 번만 요청을 다시 전송하며 청크 스트림을 사용할 수 없는 요청 간에 공급자에 대한 연결을 열어 두고자 하는 **HTTP** 소비자 끝점의 구성을 보여줍니다.*

예 12.6. HTTP Consumer Endpoint 구성

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                          http://cxf.apache.org/schemas/configuration/http-conf.xsd
                         http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit name="{http://apache.org/hello_world_soap_http}SoapPort.http-conduit">
    <http-conf:client Connection="Keep-Alive"
                      MaxRetransmits="1"
                      AllowChunking="false" />
  </http-conf:conduit>
</beans>
```

12.2.2.6. 더 알아보기

HTTP 구성에 대한 자세한 내용은 16장. 구성 요소 에서 참조하십시오.

12.2.3. WSDL 사용

12.2.3.1. 네임스페이스

HTTP 소비자 끝점을 구성하는 데 사용되는 WSDL 확장 요소는 네임스페이스 http://cxf.apache.org/transports/http/configuration 에 정의되어 있습니다. 일반적으로 접두사 http-conf 를 사용합니다. HTTP 구성 요소를 사용하려면 예 12.7. “HTTP Consumer WSDL Element's Namespace” 에 표시된 줄을 끝점의 WSDL 문서의 정의 요소에 추가해야 합니다.

예 12.7. HTTP Consumer WSDL Element's Namespace

```
<definitions ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

12.2.3.2. Undertow 런타임 또는 Netty 런타임

http-conf 네임스페이스의 요소를 사용하여 Undertow 런타임 또는 Netty 런타임을 구성할 수 있습니다.

12.2.3.3. 클라이언트 요소

http-conf:client 요소는 WSDL 문서에서 HTTP 소비자의 연결 속성을 지정하는 데 사용됩니다. http-conf:client 요소는 WSDL 포트 요소의 자식입니다. 구성 파일에서 사용되는 클라이언트 요소와 동일한 속성이 있습니다. 속성은 표 12.2. “HTTP 소비자 구성 속성” 에서 확인할 수 있습니다.

12.2.3.4. 예제

예 12.8. “HTTP Consumer Endpoint 구성” 에는 캐시와 상호 작용하지 않도록 지정하는 HTTP 소비자 엔드포인트를 구성하는 WSDL 조각을 보여줍니다.

예 12.8. HTTP Consumer Endpoint 구성

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

12.2.4. 소비자 캐시 제어 지침

표 12.3. "HTTP-conf:client 캐시 제어 지침" **HTTP 소비자가 지원하는 캐시 제어 지시문을 나열합니다.**

표 12.3. HTTP-conf:client 캐시 제어 지침

directive	동작
no-cache	캐시는 먼저 서버와의 응답을 재검증하지 않고 후속 요청을 충족하기 위해 특정 응답을 사용할 수 없습니다. 특정 응답 헤더 필드가 이 값으로 지정된 경우 제한은 응답 내의 해당 헤더 필드에만 적용됩니다. 응답 헤더 필드를 지정하지 않으면 제한이 전체 응답에 적용됩니다.
no-store	캐시는 응답의 일부 또는 이를 호출한 요청의 일부를 저장해야 합니다.
max-age	소비자는 지정된 시간(초)보다 긴 응답을 허용할 수 있습니다.
max-stale	소비자는 만료 시간을 초과한 응답을 수락할 수 있습니다. 값이 max-stale에 할당되면 소비자가 해당 응답을 허용할 수 있는 응답의 만료 시간 이외의 초 수를 나타냅니다. 값이 할당되지 않은 경우 소비자는 모든 연령의 오래된 응답을 수락할 수 있습니다.
min-fresh	소비자는 최소한 지정된 시간(초) 동안 새로운 응답을 원하는 것입니다.
no-transform	캐시는 공급자와 소비자 간 응답으로 콘텐츠의 미디어 유형 또는 위치를 수정하지 않아야 합니다.
only-if-cached	캐시는 현재 캐시에 저장된 응답만 반환하고, 다시 로드하거나 재검토해야 하는 응답이 아닙니다.
cache-extension	다른 캐시 지시문에 대한 추가 확장을 지정합니다. 확장 기능은 정보 또는 동작일 수 있습니다. 확장된 지시문은 표준 지시문의 컨텍스트에 지정되므로 확장 지시문을 이해하지 못하는 애플리케이션이 standard 지시문에서 요구하는 동작을 따를 수 있습니다.

12.3. 서비스 공급자 구성

12.3.1. HTTP 서비스 공급자의 메커니즘

HTTP 서비스 공급자 끝점은 활성 요청을 유지할지 여부, 캐시와 상호 작용하는 방법, 소비자와 통신하는 데 오류의 내결함성을 포함하여 여러 HTTP 연결 특성을 지정할 수 있습니다.

서비스 공급자 끝점은 다음 두 가지 메커니즘을 사용하여 구성할 수 있습니다.

- 설정
- WSDL

12.3.2. 구성 사용

12.3.2.1. 네임스페이스

HTTP 공급자 끝점을 구성하는 데 사용되는 요소는 네임스페이스 http://cxf.apache.org/transports/http/configuration 에 정의되어 있습니다. 일반적으로 접두사 http-conf 를 사용합니다. HTTP 구성 요소를 사용하려면 예 12.9. “HTTP 공급자 구성 네임스페이스” 에 표시된 줄을 끝점 구성 파일의 빈 요소에 추가해야 합니다. 또한 구성 요소의 네임스페이스를 xsi:schemaLocation 속성에 추가해야 합니다.

예 12.9. HTTP 공급자 구성 네임스페이스

```
<beans ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      ...
      xsi:schemaLocation="...
                          http://cxf.apache.org/transports/http/configuration
                            http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          ...">
```

12.3.2.2. Undertow 런타임 또는 Netty 런타임

http-conf 네임스페이스의 요소를 사용하여 Undertow 런타임 또는 Netty 런타임을 구성할 수 있습니다.

12.3.2.3. 대상 요소

http-conf:destination 요소와 해당 하위 항목을 사용하여 HTTP 서비스 공급자 끝점을 구성합니다. http-conf:destination 요소는 엔드포인트에 해당하는 WSDL 포트 요소를 지정하는 단일 속성인 name 을 사용합니다. name 속성의 값은 portQName'.http-destination' 형식을 사용합니다. 예 12.10. "HTTP-conf:destination Element" 는 끝점의 대상 네임스페이스가 http://widgets.widgetvendor.net인 경우 WSDL 조각 < port binding="widgetSOAPBinding" name="widgetSOAPPort >에 의해 지정된 엔드포인트에 대한 구성을 추가하는 데 사용되는 http-conf:destination 요소를 표시합니다.

예 12.10. HTTP-conf:destination Element

```
...
  <http-conf:destination name="{http://widgets/widgetvendor.net}widgetSOAPPort.http-
destination">
    ...
  </http-conf:destination>
...
```

http-conf:destination 요소에는 구성 정보를 지정하는 여러 하위 요소가 있습니다. 자세한 내용은 표 12.4. "HTTP 서비스 공급자 끝점을 구성하는 데 사용되는 요소" 에 설명되어 있습니다.

표 12.4. HTTP 서비스 공급자 끝점을 구성하는 데 사용되는 요소

요소	설명
http-conf:server	HTTP 연결 속성을 지정합니다. "서버 요소" 을 참조하십시오.
http-conf:contextMatchStrategy	HTTP 요청을 처리하기 위한 컨텍스트 일치 전략을 구성하는 매개변수를 지정합니다.
http-conf:fixedParameterOrder	이 대상에서 처리하는 HTTP 요청의 매개변수 순서가 고정되는지 여부를 지정합니다.

12.3.2.4. 서버 요소

http-conf:server 요소는 서비스 공급자 끝점의 HTTP 연결 속성을 구성하는 데 사용됩니다. 표 12.5. "HTTP 서비스 공급자 구성 속성" 에 설명된 해당 속성은 연결의 속성을 지정합니다.

표 12.5. HTTP 서비스 공급자 구성 속성

속성	설명

속성	설명
ReceiveTimeout	연결 시간이 초과되기 전에 서비스 공급자가 요청을 수신하려고 하는 시간(밀리초)을 설정합니다. 기본값은 **30000** 입니다. **0** 공급자는 공급자가 시간 초과되지 않도록 지정합니다.
SuppressClientSendErrors	요청 수신 시 오류가 발생할 때 예외가 throw되는지 여부를 지정합니다.Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. 기본값은 **false** 입니다. 오류 발생 시 예외가 발생합니다.
SuppressClientReceiveErrors	소비자에게 응답을 보내는 데 오류가 발생할 때 예외가 throw되는지 여부를 지정합니다.Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. 기본값은 **false** 입니다. 오류 발생 시 예외가 발생합니다.
HonorKeepAlive	응답이 전송된 후에도 서비스 공급자가 연결 요청을 열린 상태로 유지할지 여부를 지정합니다. 기본값은 **false** 입니다. 유지 요청은 무시됩니다.
RedirectURL	클라이언트 요청에 지정된 URL이 요청된 리소스에 더 이상 적합하지 않은 경우 클라이언트 요청을 리디렉션해야 하는 URL을 지정합니다. 이 경우 서버 응답의 첫 번째 줄에 상태 코드가 자동으로 설정되지 않으면 상태 코드가 302로 설정되고 상태 설명이 Object Moved로 설정됩니다. 값은 HTTP RedirectURL 속성 값으로 사용됩니다.
CacheControl	서비스 공급자의 응답을 소비자로 구성된 체인과 관련된 캐시에 의해 준수해야 하는 동작에 대한 지시문을 지정합니다. 12.3.4절. "서비스 공급자 캐시 제어 지침"을 참조하십시오.
ContentLocation	응답에 전송되는 리소스가 있는 URL을 설정합니다.
ContentType	응답으로 전송되는 정보의 미디어 유형을 지정합니다. 미디어 유형은 다목적 인터넷 메일 확장 (MIME) 유형을 사용하여 지정됩니다. 값은 HTTP ContentType 위치 값으로 사용됩니다.

속성	설명
ContentEncoding	서비스 공급자가 전송하는 정보에 적용된 추가 콘텐츠 인코딩을 지정합니다. 콘텐츠 인코딩 레이블은 IANA(Internet Assigned Numbers Authority)에 의해 규제됩니다. 가능한 콘텐츠 인코딩 값에는 zip, gzip, compress, deflate, identity가 포함됩니다. 이 값은 HTTP ContentEncoding 속성 값으로 사용됩니다. 콘텐츠 인코딩의 주된 사용은 zip 또는 gzip과 같은 일부 인코딩 메커니즘을 사용하여 문서를 압축하도록 허용하는 것입니다. Apache CXF는 콘텐츠 코딩에 대한 유효성 검사를 수행하지 않습니다. 지정된 콘텐츠 코딩이 애플리케이션 수준에서 지원되도록 해야 합니다.
ServerType	응답을 보내는 서버의 유형을 지정합니다. 값은 ***program-name*/*version*** (예: **Apache/1.2.5)** 형식을 사용합니다.

12.3.2.5. 예제

*예 12.11. "HTTP 서비스 공급자 끝점 구성" **keep-alive** 요청을 준수하고 모든 통신 오류를 방지하는 **HTTP** 서비스 공급자 끝점의 구성을 보여줍니다.*

예 12.11. HTTP 서비스 공급자 끝점 구성

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                          http://cxf.apache.org/schemas/configuration/http-conf.xsd
                         http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:destination name="{http://apache.org/hello_world_soap_http}SoapPort.http-
destination">
    <http-conf:server SuppressClientSendErrors="true"
                      SuppressClientReceiveErrors="true"
                      HonorKeepAlive="true" />
  </http-conf:destination>
</beans>
```

12.3.3. WSDL 사용

12.3.3.1. 네임스페이스

HTTP 공급자 끝점을 구성하는 데 사용되는 WSDL 확장 요소는 네임스페이스 http://cxf.apache.org/transports/http/configuration 에 정의되어 있습니다. 일반적으로 접두사 http-conf 를 사용합니다. HTTP 구성 요소를 사용하려면 예 12.12. “HTTP Provider WSDL Element's Namespace” 에 표시된 줄을 끝점의 WSDL 문서의 정의 요소에 추가해야 합니다.

예 12.12. HTTP Provider WSDL Element's Namespace

```
<definitions ...
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
```

12.3.3.2. Undertow 런타임 또는 Netty 런타임

http-conf 네임스페이스의 요소를 사용하여 Undertow 런타임 또는 Netty 런타임을 구성할 수 있습니다.

12.3.3.3. 서버 요소

http-conf:server 요소는 WSDL 문서에서 HTTP 서비스 공급자의 연결 속성을 지정하는 데 사용됩니다. http-conf:server 요소는 WSDL 포트 요소의 자식입니다. 구성 파일에 사용된 server 요소와 동일한 속성이 있습니다. 속성은 표 12.5. “HTTP 서비스 공급자 구성 속성” 에서 확인할 수 있습니다.

12.3.3.4. 예제

예 12.13. “HTTP 서비스 공급자 엔드 포인트 구성” 에는 캐시와 상호 작용하지 않도록 지정하는 HTTP 서비스 공급자 엔드포인트를 구성하는 WSDL 조각을 보여줍니다.

예 12.13. HTTP 서비스 공급자 엔드 포인트 구성

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

12.3.4. 서비스 공급자 캐시 제어 지침

표 12.6. “http-conf:server Cache Control Directives” HTTP 서비스 공급자가 지원하는 캐시 제어 지시문을 나열합니다.

표 12.6. http-conf:server Cache Control Directives

directive	동작
no-cache	캐시는 먼저 서버와의 응답을 재검증하지 않고 후속 요청을 충족하기 위해 특정 응답을 사용할 수 없습니다. 특정 응답 헤더 필드가 이 값으로 지정된 경우 제한은 응답 내의 해당 헤더 필드에만 적용됩니다. 응답 헤더 필드를 지정하지 않으면 제한이 전체 응답에 적용됩니다.
public	모든 캐시는 응답을 저장할 수 있습니다.
private	공용(**공유**) 캐시는 응답이 단일 사용자를 대상으로 하므로 응답을 저장할 수 없습니다. 특정 응답 헤더 필드가 이 값으로 지정된 경우 제한은 응답 내의 해당 헤더 필드에만 적용됩니다. 응답 헤더 필드를 지정하지 않으면 제한이 전체 응답에 적용됩니다.
no-store	캐시는 응답의 일부 또는 이를 호출한 요청의 일부를 저장해야 합니다.
no-transform	캐시는 서버와 클라이언트 간의 응답으로 콘텐츠의 미디어 유형 또는 위치를 수정하지 않아야 합니다.
must-revalidate	캐시는 응답과 관련된 만료된 항목을 다시 시작해야 후속 응답에 해당 항목을 사용할 수 있습니다.
proxy-revalidate	공유 캐시에만 적용할 수 있고 공유되지 않은 비공개 캐시에서 무시할 수 있다는 점을 제외하고 must-revalidate와 동일합니다. 이 지시문을 사용하는 경우 public cache 지시문도 사용해야 합니다.
max-age	클라이언트는 지정된 시간(초)보다 오래된 응답을 수락할 수 있습니다.
s-max-age	공유 캐시에서만 시행할 수 있으며 개인 비공유 캐시에서 무시된다는 점을 제외하고 max-age와 동일한 작업을 수행합니다. s-max-age에서 지정한 기간은 max-age에서 지정한 기간을 재정의합니다. 이 지시문을 사용하는 경우 proxy-revalidate 지시문도 사용해야 합니다.
cache-extension	다른 캐시 지시문에 대한 추가 확장을 지정합니다. 확장 기능은 정보 또는 동작일 수 있습니다. 확장된 지시문은 표준 지시문의 컨텍스트에 지정되므로 확장 지시문을 이해하지 못하는 애플리케이션이 standard 지시문에서 요구하는 동작을 따를 수 있습니다.

12.4. UNDERTOW 런타임 구성

12.4.1. 개요

Undertow 런타임은 분리된 끝점을 사용하여 **HTTP** 서비스 공급자 및 **HTTP** 소비자가 사용합니다. 런타임의 스레드 풀을 구성할 수 있으며, **Undertow** 런타임을 통해 **HTTP** 서비스 공급자에 대한 여러 보안 설정도 설정할 수 있습니다.

12.4.2. Maven 종속성

Apache Maven을 빌드 시스템으로 사용하는 경우 프로젝트의 **pom.xml** 파일에 다음 종속성을 포함하여 프로젝트에 **Undertow** 런타임을 추가할 수 있습니다.

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-transports-http-undertow</artifactId>
   <version>${cxf-version}</version>
</dependency>
```

12.4.3. 네임스페이스

Undertow 런타임을 구성하는 데 사용되는 요소는 네임스페이스 **http://cxf.apache.org/transports/http-undertow/configuration** 에 정의되어 있습니다. **Undertow** 구성 요소를 사용하려면 *예 12.14. "Undertow 런타임 구성 네임스페이스"* 에 표시된 줄을 끝점 구성 파일의 빈 요소에 추가해야 합니다. 이 예에서 네임스페이스에는 **httpu** 접두사가 할당됩니다. 또한 구성 요소의 네임스페이스를 **xsi:schemaLocation** 속성에 추가해야 합니다.

예 12.14. Undertow 런타임 구성 네임스페이스

```
<beans ...
      xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration"
      ...
      xsi:schemaLocation="...
                  http://cxf.apache.org/transports/http-undertow/configuration
                    http://cxf.apache.org/schemas/configuration/http-undertow.xsd
                  ...">
```

12.4.4. engine-factory 요소

httpu:engine-factory 요소는 애플리케이션에서 사용하는 **Undertow** 런타임을 구성하는 데 사용되는 루트 요소입니다. 구성 중인 **Undertow** 인스턴스를 관리하는 버스 의 이름이 값인 단일 필수 속성 버스가 있습니다.

참고

*값은 일반적으로 기본 버스 인스턴스의 이름인 **cxf** 입니다.*

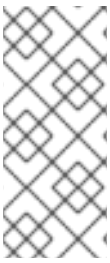
http:engine-factory 요소에는 Undertow 런타임 팩토리에서 인스턴스화한 HTTP 포트를 구성하는 데 사용되는 정보가 포함된 세 개의 자식이 있습니다. 어린이는 표 12.7. "Undertow Runtime Factory 구성을 위한 요소" 에 설명되어 있습니다.

표 12.7. Undertow Runtime Factory 구성을 위한 요소

요소	설명
httpu:engine	특정 Undertow 런타임 인스턴스에 대한 구성을 지정합니다. "엔진 요소" 을 참조하십시오.
httpu:identifiedTLSServerParameters	HTTP 서비스 공급자를 보호하기 위해 재사용 가능한 속성 세트를 지정합니다. 속성 집합을 참조할 수 있는 고유 식별자를 지정하는 단일 속성 **id** 가 있습니다.
httpu:identifiedThreadingParameters	Undertow 인스턴스의 스레드 풀을 제어하기 위해 재사용 가능한 속성 세트를 지정합니다. 속성 집합을 참조할 수 있는 고유 식별자를 지정하는 단일 속성 **id** 가 있습니다. "스레드 풀 구성" 을 참조하십시오.

12.4.5. 엔진 요소

httpu:engine 요소는 Undertow 런타임의 특정 인스턴스를 구성하는 데 사용됩니다. 여기에는 두 개의 속성인 host 가 포함되어 있는 undertow 및 port 가 포함된 글로벌 IP 주소를 지정합니다. 이 속성은 Undertow 인스턴스에서 관리하는 포트의 수를 지정합니다.



참고

port 특성에 값을 0 으로 지정할 수 있습니다. 포트 특성이 0 으로 설정된 httpu:engine 요소에 지정된 모든 스레드 속성은 명시적으로 구성되지 않은 모든 Undertow 리스너의 구성으로 사용됩니다.

각 httpu:engine 요소에는 두 개의 하위 항목이 있을 수 있습니다. 하나는 보안 속성을 구성하고 다른 하나는 Undertow 인스턴스의 스레드 풀을 구성하는 데 사용됩니다. 각 유형의 구성에 대해 구성 정보를 직접 제공하거나 상위 httpu:engine-factory 요소에 정의된 구성 속성 집합에 대한 참조를 제공할 수 있습니다.

구성 속성을 제공하는 데 사용되는 하위 요소는 표 12.8. "Undertow 런타임 인스턴스 구성을 위한 요소" 에 설명되어 있습니다.

표 12.8. Undertow 런타임 인스턴스 구성을 위한 요소

요소	설명
httpu:tlsServerParameters	특정 Undertow 인스턴스에 사용되는 보안을 구성하기 위한 속성 세트를 지정합니다.
httpu:tlsServerParametersRef	**identifiedTLSServerParameters** 요소에 의해 정의된 보안 속성 집합을 나타냅니다. **id** 속성은 참조된 **identifiedTLSServerParameters** 요소의 id를 제공합니다.
httpu:threadingParameters	특정 Undertow 인스턴스에서 사용하는 스레드 풀의 크기를 지정합니다. "스레드 풀 구성" 을 참조하십시오.
httpu:threadingParametersRef	**확인된ThreadingParameters** 요소에 의해 정의된 속성 집합을 나타냅니다. **id** 속성은 참조 **된 ThreadingParameters** 요소의 id를 제공합니다.

12.4.6. 스레드 풀 구성

다음 중 하나를 통해 Undertow 인스턴스의 스레드 풀 크기를 구성할 수 있습니다.

- *engine-factory 요소에서 확인된ThreadingParameters 요소를 사용하여 스레드 풀의 크기를 지정합니다. 그런 다음 threadingParametersRef 요소를 사용하여 요소를 참조합니다.*
- *threadingParameters 요소를 사용하여 스레드 풀의 크기를 직접 지정합니다.*

threadingParameters 에는 스레드 풀의 크기를 지정하는 두 개의 속성이 있습니다. 속성은 표 12.9. "Undertow 스레드 풀 구성을 위한 속성" 에서 확인할 수 있습니다.

참고

httpu:identifiedThreadingParameters 요소에는 단일 자 식 스레드 매개 변수 요소가 있습니다.

표 12.9. Undertow 스레드 풀 구성을 위한 속성

속성	설명
workerIOThreads	작업자에 대해 생성할 I/O 스레드 수를 지정합니다. 지정하지 않으면 dafult 값이 선택됩니다. 기본값은 CPU 코어당 하나의 I/O 스레드입니다.
minThreads	요청을 처리하는 데 Undertow 인스턴스에서 사용할 수 있는 최소 스레드 수를 지정합니다.
maxThreads	요청을 처리하는 데 Undertow 인스턴스에서 사용할 수 있는 최대 스레드 수를 지정합니다.

12.4.7. 예제

예 12.15. "Undertow 인스턴스 구성" 포트 번호 9001에서 Undertow 인스턴스를 구성하는 구성 조각을 보여줍니다.

예 12.15. Undertow 인스턴스 구성

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpu="http://cxf.apache.org/transports/http-undertow/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="http://cxf.apache.org/configuration/security
        http://cxf.apache.org/schemas/configuration/security.xsd
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://cxf.apache.org/transports/http-undertow/configuration
          http://cxf.apache.org/schemas/configuration/http-undertow.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  ...

  <httpu:engine-factory bus="cxf">
    <httpu:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpu:identifiedTLSServerParameters>

    <httpu:engine port="9001">
      <httpu:tlsServerParametersRef id="secure" />
      <httpu:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpu:engine>
  </httpu:engine-factory>
</beans>
```

12.4.8. 동시 요청 및 대기열 크기 제한

Undertow 서버 인스턴스에서 처리할 수 있는 최대 동시 연결 요청 수 및 대기열 크기에 대한 제한을 설정하는 **Request Limiting Handler**를 구성할 수 있습니다. 이 구성의 예는 다음과 같습니다. 예 12.16. "연결 요청 및 대기열 크기 제한"

표 12.10. 요청 제한 핸들러 구성을 위한 속성

속성	설명
maximumConcurrentRequests	Undertow 인스턴스에서 처리할 수 있는 최대 동시 요청 수를 지정합니다. 요청 수가 이 제한을 초과하면 요청이 대기됩니다.
queueSize	Undertow 인스턴스에서 처리하기 위해 대기열에 있을 수 있는 총 요청 수를 지정합니다. 요청 수가 이 제한을 초과하면 요청이 거부됩니다.

예 12.16. 연결 요청 및 대기열 크기 제한

```
<httpu:engine-factory>
    <httpu:engine port="8282">
        <httpu:handlers>
            <bean class="org.jboss.fuse.quickstarts.cxf.soap.CxfRequestLimitingHandler">
                <property name="maximumConcurrentRequests" value="1" />
                <property name="queueSize" value="1"/>
            </bean>
        </httpu:handlers>
    </httpu:engine>
</httpu:engine-factory>
```

12.5. NETTY 런타임 구성

12.5.1. 개요

Netty 런타임은 분리된 끝점을 사용하여 **HTTP** 서비스 공급자 및 **HTTP** 소비자가 사용합니다. 런타임의 스레드 풀을 구성할 수 있으며 **Netty** 런타임을 통해 **HTTP** 서비스 공급자에 대한 여러 보안 설정을 설정할 수도 있습니다.

12.5.2. Maven 종속 항목

Apache Maven을 빌드 시스템으로 사용하는 경우 프로젝트의 **pom.xml** 파일에 다음 종속성을 포함하여 **Netty** 런타임(웹 서비스 엔드포인트 정의)의 서버 측 구현을 프로젝트에 추가할 수 있습니다.

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-transports-http-netty-server</artifactId>
   <version>${cxf-version}</version>
</dependency>
```

프로젝트의 **pom.xml** 파일에 다음 종속성을 포함하여 **Netty** 런타임(웹 서비스 클라이언트 정의)의 클라이언트 측 구현을 프로젝트에 추가할 수 있습니다.

```
<dependency>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-rt-transports-http-netty-client</artifactId>
   <version>${cxf-version}</version>
</dependency>
```

12.5.3. 네임스페이스

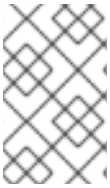
Netty 런타임을 구성하는 데 사용되는 요소는 네임스페이스 **http://cxf.apache.org/transports/http-netty-server/configuration** 에 정의되어 있습니다. 일반적으로 **httpn** 접두사를 사용합니다. **Netty** 구성 요소를 사용하려면 *예 12.17. "Netty Runtime 구성 네임스페이스"* 에 표시된 줄을 끝점 구성 파일의 빈 요소에 추가해야 합니다. 또한 구성 요소의 네임스페이스를 **xsi:schemaLocation** 속성에 추가해야 합니다.

예 12.17. Netty Runtime 구성 네임스페이스

```
<beans ...
      xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
      ...
      xsi:schemaLocation="...
            http://cxf.apache.org/transports/http-netty-server/configuration
          http://cxf.apache.org/schemas/configuration/http-netty-server.xsd
            ...">
```

12.5.4. engine-factory 요소

httpn:engine-factory 요소는 애플리케이션에서 사용하는 **Netty** 런타임을 구성하는 데 사용되는 루트 요소입니다. 구성되는 **Netty** 인스턴스를 관리하는 버스 의 이름이 값인 단일 필수 속성인 버스 가 있습니다.



참고

값은 일반적으로 기본 버스 인스턴스의 이름인 **cxf** 입니다.

httpn:engine-factory 요소에는 Netty 런타임 팩토리에서 인스턴스화한 HTTP 포트를 구성하는 데 사용되는 정보가 포함된 세 개의 자식이 있습니다. 어린이는 표 12.11. "Netty Runtime Factory 구성을 위한 요소" 에 설명되어 있습니다.

표 12.11. Netty Runtime Factory 구성을 위한 요소

요소	설명
httpn:engine	특정 Netty 런타임 인스턴스에 대한 구성을 지정합니다. "엔진 요소" 을 참조하십시오.
httpn:identifiedTLSServerParameters	HTTP 서비스 공급자를 보호하기 위해 재사용 가능한 속성 세트를 지정합니다. 속성 집합을 참조할 수 있는 고유 식별자를 지정하는 단일 속성 **id** 가 있습니다.
httpn:identifiedThreadingParameters	Netty 인스턴스의 스레드 풀을 제어하기 위해 재사용 가능한 속성 세트를 지정합니다. 속성 집합을 참조할 수 있는 고유 식별자를 지정하는 단일 속성 **id** 가 있습니다. "스레드 풀 구성" 을 참조하십시오.

12.5.5. 엔진 요소

httpn:engine 요소는 Netty 런타임의 특정 인스턴스를 구성하는 데 사용됩니다. 표 12.12. "Netty Runtime Instance 구성을 위한 속성" httpn:engine 요소에서 지원하는 특성을 보여줍니다.

표 12.12. Netty Runtime Instance 구성을 위한 속성

속성	설명
port	Netty HTTP 서버 인스턴스에서 사용하는 포트를 지정합니다. port 특성에 값을 **0** 으로 지정할 수 있습니다. 포트 특성을 **0** 으로 설정하여 engine 요소에 지정된 모든 스레드 속성은 명시적으로 구성되지 않은 모든 Netty 리스너의 구성으로 사용됩니다.
host	Netty HTTP 서버 인스턴스에서 사용하는 수신 주소를 지정합니다. 값은 호스트 이름 또는 IP 주소일 수 있습니다. 지정되지 않은 경우 Netty HTTP 서버는 모든 로컬 주소에서 수신 대기합니다.
readIdleTime	Netty 연결의 최대 읽기 유휴 시간을 지정합니다. 기본 스트림에 읽기 작업이 있을 때마다 타이머가 재설정됩니다.

속성	설명
writeIdleTime	Netty 연결의 최대 쓰기 유휴 시간을 지정합니다. 기본 스트림에 쓰기 작업이 있을 때마다 타이머가 재설정됩니다.
maxChunkContentSize	Netty 연결에 대해 집계된 최대 콘텐츠 크기를 지정합니다. 기본값은 10MB입니다.

***httpn:engine** 요소에는 **Netty** 인스턴스의 스레드 풀을 구성하기 위한 하나의 하위 요소와 보안 속성을 구성하는 하위 요소가 있습니다. 각 유형의 구성에 대해 구성 정보를 직접 제공하거나 상위 **httpn:engine-factory** 요소에 정의된 구성 속성 집합에 대한 참조를 제공할 수 있습니다.*

***httpn:engine** 의 지원되는 하위 요소는 표 12.13. "Netty Runtime Instance 구성을 위한 요소" 에 표시됩니다.*

표 12.13. Netty Runtime Instance 구성을 위한 요소

요소	설명
httpn:tlsServerParameters	특정 Netty 인스턴스에 사용되는 보안을 구성하기 위한 속성 세트를 지정합니다.
httpn:tlsServerParametersRef	**identifiedTLSServerParameters** 요소에 의해 정의된 보안 속성 집합을 나타냅니다. **id** 속성은 참조된 **identifiedTLSServerParameters** 요소의 id를 제공합니다.
httpn:threadingParameters	특정 Netty 인스턴스에서 사용하는 스레드 풀의 크기를 지정합니다. "스레드 풀 구성" 을 참조하십시오.
httpn:threadingParametersRef	**확인된ThreadingParameters** 요소에 의해 정의된 속성 집합을 나타냅니다. **id** 속성은 참조 **된 ThreadingParameters** 요소의 id를 제공합니다.
httpn:sessionSupport	**true** 인 경우 HTTP 세션에 대한 지원을 활성화합니다. 기본값은 **false** 입니다.
httpn:reuseAddress	**ReuseAddress** TCP 소켓 옵션을 설정하는 부울 값을 지정합니다. 기본값은 **false** 입니다.

12.5.6. 스레드 풀 구성

*다음 중 하나를 통해 **Netty** 인스턴스의 스레드 풀 크기를 구성할 수 있습니다.*

- *engine-factory 요소에서 확인된ThreadingParameters 요소를 사용하여 스레드 풀의 크기를 지정합니다. 그런 다음 threadingParametersRef 요소를 사용하여 요소를 참조합니다.*

- *threadingParameters 요소를 사용하여 스레드 풀의 크기를 직접 지정합니다.*

threadingParameters 요소에는 표 12.14. "Netty Thread Pool 구성을 위한 속성" 에 설명된 대로 스레드 풀의 크기를 지정하는 하나의 속성이 있습니다.

참고

httpn:identifiedThreadingParameters 요소에는 단일 자 식 스레드 매개 변수 요소가 있습니다.

표 12.14. Netty Thread Pool 구성을 위한 속성

속성	설명
threadPoolSize	요청을 처리하는 데 Netty 인스턴스에서 사용할 수 있는 스레드 수를 지정합니다.

12.5.7. 예제

예 12.18. "Netty 인스턴스 구성" 는 다양한 Netty 포트를 구성하는 구성 조각을 보여줍니다.

예 12.18. Netty 인스턴스 구성

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:h="http://cxf.apache.org/transports/http/configuration"
      xmlns:httpn="http://cxf.apache.org/transports/http-netty-server/configuration"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xsi:schemaLocation="
       http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
       http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
       http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
       http://cxf.apache.org/transports/http-netty-server/configuration
          http://cxf.apache.org/schemas/configuration/http-netty-server.xsd"
>
```

```
    ...
    <httpn:engine-factory bus="cxf">
      <httpn:identifiedTLSServerParameters id="sample1">
        <httpn:tlsServerParameters jsseProvider="SUN" secureSocketProtocol="TLS">
          <sec:clientAuthentication want="false" required="false"/>
        </httpn:tlsServerParameters>
      </httpn:identifiedTLSServerParameters>

      <httpn:identifiedThreadingParameters id="sampleThreading1">
        <httpn:threadingParameters threadPoolSize="120"/>
      </httpn:identifiedThreadingParameters>

      <httpn:engine port="9000" readIdleTime="30000" writeIdleTime="90000">
        <httpn:threadingParametersRef id="sampleThreading1"/>
      </httpn:engine>

      <httpn:engine port="0">
        <httpn:threadingParameters threadPoolSize="400"/>
      </httpn:engine>

      <httpn:engine port="9001" readIdleTime="40000" maxChunkContentSize="10000">
        <httpn:threadingParameters threadPoolSize="99" />
        <httpn:sessionSupport>true</httpn:sessionSupport>
      </httpn:engine>

      <httpn:engine port="9002">
        <httpn:tlsServerParameters>
          <sec:clientAuthentication want="true" required="true"/>
        </httpn:tlsServerParameters>
      </httpn:engine>

      <httpn:engine port="9003">
        <httpn:tlsServerParametersRef id="sample1"/>
      </httpn:engine>

    </httpn:engine-factory>
  </beans>
```

12.6. 분리된 모드에서 HTTP 전송 사용

12.6.1. 개요

*일반적인 **HTTP** 요청/응답 시나리오에서는 요청 및 응답이 동일한 **HTTP** 연결을 사용하여 전송됩니다. 서비스 공급자는 요청을 처리하고 적절한 **HTTP** 상태 코드 및 응답 콘텐츠를 포함하는 응답으로 응답합니다. 요청이 성공하면 **HTTP** 상태 코드가 **200**으로 설정됩니다.*

***WS-RM**을 사용하거나 요청이 실행하는 데 시간이 오래 걸리는 경우와 같이 일부 경우 요청 및 응답 메시지를 분리하는 것이 좋습니다. 이 경우 서비스 제공 업체는 요청을 수신한 **HTTP** 연결의 백채널을 통해 소비자에게 **202 Accepted** 응답을 보냅니다. 그런 다음 요청을 처리하고 새로운 분리 된 서버를 사용하여*

소비자로 응답을 다시 보냅니다. 소비자 런타임은 들어오는 응답을 수신하고 애플리케이션 코드로 반환하기 전에 적절한 요청과 관련이 있습니다.

12.6.2. 분리된 상호 작용 구성

분리된 모드에서 **HTTP** 전송을 사용하려면 다음을 수행해야 합니다.

1. **WS-Addressing**을 사용하도록 소비자를 구성합니다.

   "WS-Addressing을 사용하도록 엔드포인트 구성" 을 참조하십시오.

2. 분리된 엔드포인트를 사용하도록 소비자를 구성합니다.

   "소비자 구성" 을 참조하십시오.

3. 사용자가 **WS-Addressing**을 사용하도록 소비자가 상호 작용하는 서비스 공급자를 구성합니다.

   "WS-Addressing을 사용하도록 엔드포인트 구성" 을 참조하십시오.

12.6.3. WS-Addressing을 사용하도록 엔드포인트 구성

소비자가 **WS-Addressing**을 사용하는 소비자 및 모든 서비스 공급자를 지정합니다.

끝점이 다음 두 가지 방법 중 하나로 **WS-Addressing**을 사용하도록 지정할 수 있습니다.

- 예 12.19. "WSDL을 사용하여 WS 주소 지정 활성화" 과 같이 **wswa:UsingAddressing** 요소를 엔드포인트의 **WSDL** 포트 요소에 추가합니다.

  예 12.19. WSDL을 사용하여 WS 주소 지정 활성화

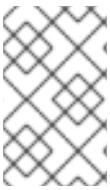
  ```
  ...
  <service name="WidgetSOAPService">
    <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
      <soap:address="http://widgetvendor.net/widgetSeller" />
      <wswa:UsingAddressing xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
  ```

```
  </port>
</service>
...
```

- *예 12.20. "정책을 사용하여 WS 연결 활성화" 과 같이 끝점의 WSDL 포트 요소에 WS-Addressing 정책 추가*

예 12.20. 정책을 사용하여 WS 연결 활성화

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy"> <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata"> <wsp:Policy/>
</wsam:Addressing> </wsp:Policy>
  </port>
</service>
...
```



참고

WS-Addressing 정책은 wswa:UsingAddressing WSDL 요소를 대체합니다.

12.6.4. 소비자 구성

http-conf:conduit 요소의 DecoupledEndpoint 특성을 사용하여 분리된 엔드포인트를 사용하도록 소비자 엔드포인트를 구성합니다.

예 12.21. "분리된 HTTP 끝점을 사용하도록 소비자 구성" 분리된 끝점을 사용하도록 예 12.19. "WSDL을 사용하여 WS 주소 지정 활성화" 에 정의된 엔드포인트를 설정하는 구성을 보여줍니다. 이제 소비자는 http://widgetvendor.net/widgetSellerInbox 에서 모든 응답을 받습니다.

예 12.21. 분리된 HTTP 끝점을 사용하도록 소비자 구성

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
                          http://cxf.apache.org/schemas/configuration/http-conf.xsd
                          http://www.springframework.org/schema/beans
                          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http:conduit name="{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
```

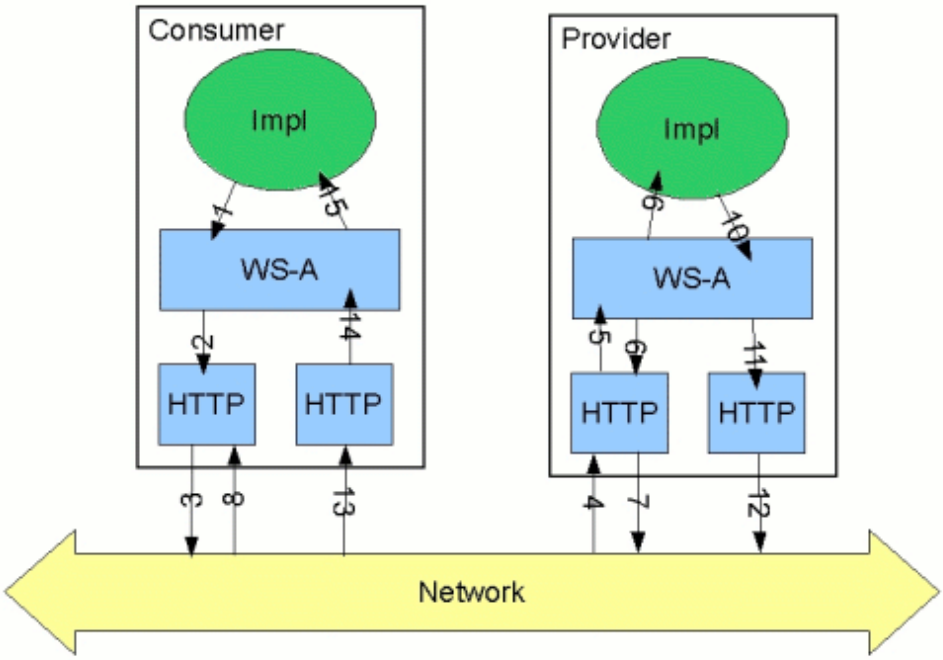
```
    <http:client DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

12.6.5. 메시지를 처리하는 방법

분리된 모드에서 **HTTP** 전송을 사용하면 **HTTP** 메시지 처리에 더 복잡한 계층을 추가합니다. 애플리케이션의 구현 수준 코드에 더 복잡한 복잡성이 투명하지만, 디버깅 이유를 위해 발생하는 상황을 이해하는 것이 중요할 수 있습니다.

그림 12.1. "분리된 HTTP 전송을 위한 메시지 흐름" 는 **HTTP**를 분리된 모드로 사용할 때 메시지 흐름을 표시합니다.

그림 12.1. 분리된 HTTP 전송을 위한 메시지 흐름



요청은 다음 프로세스를 시작합니다.

1. 소비자 구현은 작업을 호출하고 요청 메시지가 생성됩니다.

2. *WS-Addressing 계층은 WS-A 헤더를 메시지에 추가합니다.*

   분리된 끝점이 소비자 구성에 지정되면 decoupled 엔드포인트의 주소가 WS-A ReplyTo 헤더에 배치됩니다.

3. *메시지가 서비스 공급자로 전송됩니다.*

4. *서비스 공급자가 메시지를 수신합니다.*

5. *소비자의 요청 메시지는 공급자의 WS-A 계층으로 디스패치됩니다.*

6. *WS-A ReplyTo 헤더가 anonymous로 설정되지 않았기 때문에 공급자는 HTTP 상태 코드가 202로 설정된 메시지를 다시 보냅니다. 요청이 수신되었음을 인정합니다.*

7. *HTTP 계층은 원래 연결의 백채널을 사용하여 202 수락된 메시지를 소비자에게 다시 보냅니다.*

8. *소비자는 원본 메시지를 보내는 데 사용되는 HTTP 연결의 백채널에 202 Accepted 응답을 수신합니다.*

   소비자가 202 Accepted 응답을 받으면 HTTP 연결이 종료됩니다.

9. *요청은 요청이 처리되는 서비스 공급자의 구현에 전달됩니다.*

10. *응답이 준비되면 WS-A 계층으로 디스패치됩니다.*

11. *WS-A 계층은 WS-Addressing 헤더를 응답 메시지에 추가합니다.*

12. *HTTP 전송은 고객의 분리된 엔드포인트에 응답을 보냅니다.*

13. *소비자의 분리된 엔드포인트는 서비스 공급자로부터 응답을 수신합니다.*

14. *이 응답은 **WS-A RelatesTo** 헤더를 사용하여 적절한 요청과 상관 관계가 있는 소비자의 **WS-A** 계층으로 디스패치됩니다.*

15. *상관 관계가 클라이언트 구현으로 반환되고 호출은 차단 해제됩니다.*

13장. SOAP OVER JMS 사용

초록

Apache CXF는 W3C 표준 SOAP/JMS 전송을 구현합니다. 이 표준은 SOAP/HTTP 서비스에 대한 보다 강력한 대안을 제공하기 위한 것입니다. 이 전송을 사용하는 Apache CXF 애플리케이션은 SOAP/JMS 표준도 구현하는 애플리케이션과 상호 운용할 수 있어야 합니다. 전송은 끝점의 WSDL에서 직접 구성됩니다.

참고: CXF 3.0에서는 JMS 1.0.2 API에 대한 지원이 제거되었습니다. RedHat JBoss Fuse 6.2 이상을 사용하는 경우(CXF 3.0 포함) JMS 공급자는 JMS 1.1 API를 지원해야 합니다.

13.1. 기본 설정

13.1.1. 개요

JMS 프로토콜을 통한 SOAP 는 대부분의 서비스에서 사용하는 사용자 지정 SOAP/HTTP 프로토콜에 보다 안정적인 전송 계층을 제공하는 방식으로 W3C(World Wide Web Consortium)에 의해 정의됩니다. Apache CXF 구현은 사양을 완전히 준수하므로 호환 가능한 모든 프레임워크와 호환되어야 합니다.

이 전송에서는 JNDI를 사용하여 JMS 대상을 찾습니다. 작업이 호출되면 요청은 SOAP 메시지로 패키지되고 JMS 메시지 본문에서 지정된 대상에 전송됩니다.

SOAP/JMS 전송을 사용하려면 다음을 수행합니다.

1. 전송 유형을 SOAP/JMS로 지정합니다.
2. JMS URI를 사용하여 대상 대상을 지정합니다.
3. 필요한 경우 JNDI 연결을 구성합니다.
4. 선택적으로 추가 JMS 구성을 추가합니다.

13.1.2. JMS 전송 유형 지정

WSDL 바인딩을 지정할 때 JMS 전송을 사용하도록 SOAP 바인딩을 구성합니다. soap:binding 요소의 transport 속성을 http://www.w3.org/2010/soapjms/ 로 설정합니다. 예 13.1. "JMS 바인딩 사양을 통한 SOAP" SOAP/JMS를 사용하는 WSDL 바인딩을 보여줍니다.

예 13.1. JMS 바인딩 사양을 통한 SOAP

```
<wsdl:binding ... >
  <soap:binding style="document"
                transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

13.1.3. 대상 대상 지정

엔드포인트에 대한 WSDL 포트를 지정할 때 JMS 대상의 주소를 지정합니다. SOAP/JMS 엔드포인트의 address 사양은 SOAP/HTTP 끝점과 동일한 soap:address 요소 및 속성을 사용합니다. 차이점은 주소 사양입니다. JMS 엔드포인트는 JMS 1.0의 URI Scheme에 정의된 대로 JMS URI를 사용합니다. 예 13.2. "JMS URI 구문" 는 JMS URI의 구문을 표시합니다.

예 13.2. JMS URI 구문

```
jms:variant:destination?options
```

표 13.1. "JMS URI 변형" JMS URI에 사용 가능한 변형에 대해 설명합니다.

표 13.1. JMS URI 변형

변형	설명
jndi	대상 이름이 JNDI 대기열 이름임을 지정합니다. 이 변형을 사용하는 경우 JNDI 공급자에 액세스하기 위한 구성을 제공해야 합니다.
jndi-topic	대상 이름이 JNDI 주제 이름임을 지정합니다. 이 변형을 사용하는 경우 JNDI 공급자에 액세스하기 위한 구성을 제공해야 합니다.
queue	대상이 JMS를 사용하여 확인된 대기열 이름임을 지정합니다. 제공된 문자열은 대상의 표현을 생성하기 위해 **Session.createQueue()** 로 전달됩니다.
주제	대상이 JMS를 사용하여 확인된 주제 이름임을 지정합니다. 제공된 문자열은 대상 표현을 생성하기 위해 **Session.createTopic()** 로 전달됩니다.

JMS URI의 옵션 부분은 전송을 구성하는 데 사용되며 13.2절. “JMS URI” 에서 설명합니다.

예 13.3. “SOAP/JMS 엔드포인트 주소” 대상 대상이 JNDI를 사용하여 조회되는 SOAP/JMS 엔드포인트에 대한 WSDL 포트 항목을 표시합니다.

예 13.3. SOAP/JMS 엔드포인트 주소

```
<wsdl:port ... >
  ...
  <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

13.1.4. JNDI 및 JMS 전송 구성

SOAP/JMS는 JNDI 연결 및 JMS 전송을 구성하는 여러 가지 방법을 제공합니다.

- 13.2절. “JMS URI”
- 13.3절. “WSDL 확장”

13.2. JMS URI

13.2.1. 개요

SOAP/JMS를 사용하는 경우 JMS URI를 사용하여 끝점의 대상 대상을 지정합니다. URI에 하나 이상의 옵션을 추가하여 JMS 연결을 구성하는 데 JMS URI를 사용할 수도 있습니다. 이러한 옵션은 IETF 표준, Java Message Service 1.0용 URI Scheme에 자세히 설명되어 있습니다. JNDI 시스템, 응답 대상, 사용할 전달 모드 및 기타 JMS 속성을 구성하는 데 사용할 수 있습니다.

13.2.2. 구문

예 13.4. “JMS URI 옵션의 구문” 에 표시된 대로 대상의 주소에서 물음표(?)로 구분하여 JMS URI 끝에 하나 이상의 옵션을 추가할 수 있습니다. 여러 옵션은 앰퍼샌드(&)로 구분됩니다. 예 13.4. “JMS URI 옵션의 구문” 는 JMS URI에서 여러 옵션을 사용하는 구문을 보여줍니다.

예 13.4. JMS URI 옵션의 구문

> *jms:variant:jmsAddress?option1=value1&option2=value2&_optionN_=valueN*

13.2.3. JMS 속성

표 13.2. "URI 옵션으로 설정된 JMS 속성" 는 JMS 전송 계층에 영향을 주는 URI 옵션을 보여줍니다.

표 13.2. URI 옵션으로 설정된 JMS 속성

속성	기본값	설명
conduitIdSelectorPrefix		[선택 사항] conduit이 생성하는 모든 상관 관계 ID 앞에 있는 문자열 값입니다. 선택기는 이를 사용하여 응답을 수신할 수 있습니다.
deliveryMode	**PERSISTENT**	JMS **PERSISTENT** 또는 **NON_PERSISTENT** 메시지 의미 체계를 사용할지 여부를 지정합니다. **PERSISTENT** 전달 모드의 경우 JMS 브로커는 메시지를 인증하기 전에 영구 스토리지에 저장합니다. 반면 **NON_PERSISTENT** 메시지는 메모리에만 유지됩니다.
durableSubscriptionClientID		[선택 사항] 연결의 클라이언트 식별자를 지정합니다. 이 속성은 공급자가 클라이언트를 대신하여 유지 관리하는 상태와 연결을 연결하는 데 사용됩니다.This property is used to associate a connection with a state that the provider maintains on behalf of the client. 이를 통해 후속 ID를 가진 구독자는 이전 구독자가 남겨 두었던 상태에서 서브스크립션을 재개할 수 있습니다.
durableSubscriptionName		[선택 사항] 서브스크립션의 이름을 지정합니다.
messageType	**byte**	CXF에서 사용하는 JMS 메시지 유형을 지정합니다. 유효한 값은 다음과 같습니다. • byte • text • Binary

속성	기본값	설명
암호		[선택 사항] 연결 생성을 위한 암호를 지정합니다. URI에 이 속성을 추가하는 것은 권장되지 않습니다.
priority	**4**	0(최저)에서 9(최고) 사이의 범위인 JMS 메시지 우선 순위를 지정합니다.
receiveTimout	**60000**	요청/복원 교환이 사용될 때 클라이언트가 응답을 기다리는 시간을 지정합니다.Specifies the time, in milliseconds, the client will wait for a reply when request/reply exchange are used.
reconnectOnException	**true**	[CXF 3.0에서 사용 중지됨] 예외가 발생할 때 전송이 다시 연결되는지 여부를 지정합니다. 3.0에서는 예외가 발생할 때 항상 전송이 다시 연결됩니다.
replyToName		[선택 사항] 큐 메시지의 응답 대상을 지정합니다. 응답 대상은 **JMSReplyTo** 헤더에 나타납니다. JMS 공급자가 지정되지 않은 경우 임시 응답 큐를 할당하므로 request-reply 의미가 있는 애플리케이션에 이 속성을 설정하는 것이 좋습니다. 이 속성의 값은 JMS URI에 지정된 변형에 따라 해석됩니다. ● **JNDI** 변형 - JNDI로 확인된 대상 대기열의 이름입니다. ● **queue** variant- JMS를 사용하여 확인된 대상 대기열의 이름입니다.
sessionTransacted	**false**	트랜잭션 유형을 지정합니다. 유효한 값은 다음과 같습니다. ● **true**-resource 로컬 트랜잭션 ● **False**-JTA 트랜잭션

속성	기본값	설명
timeToLive	**0**	JMS 공급자가 메시지를 삭제할 시간(밀리초)을 지정합니다. 값 0은 무한한 수명을 나타냅니다.A value of **0** indicates an infinite lifetime.
topicReplyToName		[선택 사항] 주제 메시지의 응답 대상을 지정합니다. 이 속성의 값은 JMS URI에 지정된 변형에 따라 해석됩니다. ● **JNDI-topic -**JNDI로 확인된 대상 항목의 이름입니다. ● **topic**- JMS에서 해결한 대상 항목의 이름입니다.
useConduitIdSelector	**true**	conduit의 UUID가 모든 상관 관계 ID의 접두사로 사용되는지 여부를 지정합니다. 모든 conduits에는 고유한 UUID가 할당되므로 이 속성을 **true** 로 설정하면 여러 끝점이 JMS 대기열 또는 주제를 공유할 수 있습니다.
사용자 이름		[선택 사항] 연결을 만드는 데 사용할 사용자 이름을 지정합니다.

13.2.4. JNDI 속성

표 13.3. "URI 옵션으로 설정된 JNDI 속성" 는 이 엔드포인트에 대해 JNDI를 구성하는 데 사용할 수 있는 URI 옵션을 보여줍니다.

표 13.3. URI 옵션으로 설정된 JNDI 속성

속성	설명
jndiConnectionFactoryName	JMS 연결 팩토리의 JNDI 이름을 지정합니다.
jndiInitialContextFactory	JNDI 공급자의 정규화된 Java 클래스 이름을 지정합니다(Java **x.jms.InitialContextFactory** 유형 여야 함). **java.naming.factory.initial** Java 시스템 속성을 설정하는 것과 동일합니다.

속성	설명
jndiTransactionManagerName	Spring, Blueprint 또는 JNDI에서 검색할 JTA 트랜잭션 관리자의 이름을 지정합니다. 트랜잭션 관리자가 발견되면 JTA 트랜잭션이 활성화됩니다. **sessionTransacted** JMS 속성을 참조하십시오.
jndiURL	JNDI 공급자를 초기화하는 URL을 지정합니다. **java.naming.provider.url** Java 시스템 속성을 설정하는 것과 동일합니다.

13.2.5. 추가 JNDI 속성

java.naming.factory.initial *및* ***java.naming.provider.url*** *의 속성은* ***JNDI*** *공급자를 초기화하는 데 필요한 표준 속성입니다. 그러나* ***JNDI*** *공급자는 표준 속성 외에도 사용자 정의 속성을 지원할 수도 있습니다. 이 경우* ***jndi-PropertyName*** *형식의* ***URI*** *옵션을 설정하여 임의의* ***JNDI*** *속성을 설정할 수 있습니다.*

예를 들어, ***JNDI****의* ***SUN****의* ***LDAP*** *구현을 사용하는 경우 예 13.5. "JMS URI에서 JNDI 속성 설정" 과 같이* ***JMS URI****에서* ***JNDI*** *속성* ***java.naming.factory.control*** *을 설정할 수 있습니다.*

> *예 13.5. JMS URI에서 JNDI 속성 설정*
>
> ```
> jms:queue:FOO.BAR?jndi-
> java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
> ```

13.2.6. 예제

JMS *공급자가 아직 구성되지 않은 경우 옵션을 사용하여* ***URI****에 필수* ***JNDI*** *구성 세부 정보를 제공할 수 있습니다(표 13.3. "URI 옵션으로 설정된 JNDI 속성"참조). 예를 들어* ***Apache ActiveMQ JMS*** *공급자를 사용하도록 엔드포인트를 구성하고* ***test.cxf.jmstransport.queue*** *라는 큐에 연결하려면 예 13.6. "JNDI 연결을 구성하는 JMS URI" 에 표시된* ***URI****를 사용합니다.*

> *예 13.6. JNDI 연결을 구성하는 JMS URI*
>
> ```
> jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
> ?jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContextFactory
> &jndiConnectionFactoryName=ConnectionFactory
> &jndiURL=tcp://localhost:61616
> ```

13.2.7. 서비스 게시

JAX-WS 표준 publish() 메서드를 사용하여 SOAP/JMS 서비스를 게시할 수 없습니다. 대신 예 13.7. "SOAP/JMS 서비스 게시" 과 같이 Apache CXF의 JaxWsServerFactoryBean 클래스를 사용해야 합니다.

예 13.7. SOAP/JMS 서비스 게시

```
String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
    + "?jndiInitialContextFactory"
    + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
    + "&jndiConnectionFactoryName=ConnectionFactory"
    + "&jndiURL=tcp://localhost:61500";
Hello implementor = new HelloImpl();
JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
svrFactory.setServiceClass(Hello.class);
svrFactory.setAddress(address);
svrFactory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
svrFactory.setServiceBean(implementor);
svrFactory.create();
```

예 13.7. "SOAP/JMS 서비스 게시" 의 코드는 다음을 수행합니다.

끝점의 주소를 나타내는 JMS URI를 생성합니다.

JaxWsServerFactoryBean 을 인스턴스화하여 서비스를 게시합니다.

서비스의 JMS URI를 사용하여 팩토리 빈의 address 필드를 설정합니다.

팩토리에 의해 생성된 서비스가 SOAP/JMS 전송을 사용하도록 지정합니다.

13.2.8. 서비스 사용

표준 JAX-WS API는 SOAP/JMS 서비스를 사용하는 데 사용할 수 없습니다. 대신 예 13.8. "SOAP/JMS 서비스 사용" 과 같이 Apache CXF의 JaxWsProxyFactoryBean 클래스를 사용해야 합니다.

예 13.8. SOAP/JMS 서비스 사용

```
// Java
public void invoke() throws Exception {
```

```
        String address = "jms:jndi:dynamicQueues/test.cxf.jmstransport.queue3"
            + "?jndiInitialContextFactory"
            + "=org.apache.activemq.jndi.ActiveMQInitialContextFactory"
            + "&jndiConnectionFactoryName=ConnectionFactory&jndiURL=tcp://localhost:61500";
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
        factory.setAddress(address);
        factory.setTransportId(JMSSpecConstants.SOAP_JMS_SPECIFICIATION_TRANSPORTID);
        factory.setServiceClass(Hello.class);
        Hello client = (Hello)factory.create();
        String reply = client.sayHi(" HI");
        System.out.println(reply);
    }
```

예 13.8. "SOAP/JMS 서비스 사용" 의 코드는 다음을 수행합니다.

*끝점의 주소를 나타내는 **JMS URI**를 생성합니다.*

***JaxWsProxyFactoryBean** 을 인스턴스화하여 프록시를 생성합니다.*

*서비스의 **JMS URI**를 사용하여 팩토리 빈의 **address** 필드를 설정합니다.*

*팩토리에서 생성한 프록시가 **SOAP/JMS** 전송을 사용하도록 지정합니다.*

13.3. WSDL 확장

13.3.1. 개요

*바인딩 범위, 서비스 범위 또는 포트 범위에서 **WSDL** 확장 요소를 계약에 삽입하여 **JMS** 전송의 기본 구성을 지정할 수 있습니다. **WSDL** 확장 기능을 사용하면 **JNDI InitialContext** 를 부트스트래핑하는 속성을 지정할 수 있습니다. 그러면 **JMS** 대상을 조회하는 데 사용할 수 있습니다. **JMS** 전송 계층의 동작에 영향을 미치는 일부 속성을 설정할 수도 있습니다.*

13.3.2. SOAP/JMS 네임스페이스

***SOAP/JMS WSDL** 확장은 http://www.w3.org/2010/soapjms/ 네임스페이스에 정의되어 있습니다. **WSDL** 계약에서 이를 사용하려면 다음과 같은 설정을 **wsdl:definitions** 요소에 추가합니다.*

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

13.3.3. WSDL 확장 요소

표 13.4. "SOAP/JMS WSDL 확장 요소" JMS 전송을 구성하는 데 사용할 수 있는 모든 WSDL 확장 요소를 보여줍니다.

표 13.4. SOAP/JMS WSDL 확장 요소

요소	기본값	설명
soapjms:jndiInitialContextFactory		JNDI 공급자의 정규화된 Java 클래스 이름을 지정합니다. **java.naming.factory.initial** Java 시스템 속성을 설정하는 것과 동일합니다.
soapjms:jndiURL		JNDI 공급자를 초기화하는 URL을 지정합니다. **java.naming.provider.url** Java 시스템 속성을 설정하는 것과 동일합니다.
soapjms:jndiContextParameter		JNDI **InitialContext** 를 생성하기 위한 추가 속성을 지정합니다. **name** 및 **value** 속성을 사용하여 속성을 지정합니다.
soapjms:jndiConnectionFactoryName		JMS 연결 팩토리의 JNDI 이름을 지정합니다.
soapjms:deliveryMode	**PERSISTENT**	JMS **PERSISTENT** 또는 **NON_PERSISTENT** 메시지 의미 체계를 사용할지 여부를 지정합니다. **PERSISTENT** 전달 모드의 경우 JMS 브로커는 메시지를 인증하기 전에 영구 스토리지에 저장합니다. 반면 **NON_PERSISTENT** 메시지는 메모리에만 유지됩니다.

요소	기본값	설명
soapjms:replyToName		[선택 사항] 큐 메시지의 응답 대상을 지정합니다. 응답 대상은 **JMSReplyTo** 헤더에 나타납니다. JMS 공급자가 지정되지 않은 경우 임시 응답 큐를 할당하므로 request-reply 의미가 있는 애플리케이션에 이 속성을 설정하는 것이 좋습니다. 이 속성의 값은 JMS URI에 지정된 변형에 따라 해석됩니다. ● **JNDI** 변형 - JNDI로 확인된 대상 대기열의 이름입니다. ● **queue** variant- JMS를 사용하여 확인된 대상 대기열의 이름입니다.
soapjms:priority	**4**	0(최저)에서 9(최고) 사이의 범위인 JMS 메시지 우선 순위를 지정합니다.
soapjms:timeToLive	**0**	시간(밀리초)은 JMS 공급자가 메시지를 삭제합니다. 값 0은 무한한 수명을 나타냅니다.A value of **0** represents an infinite lifetime.

13.3.4. 구성 범위

***WSDL** 계약을 배치하는 **WSDL** 요소는 계약에 정의된 끝점에 대한 구성 변경 범위에 영향을 미칩니다. **SOAP/JMS WSDL** 요소는 **wsdl:binding** 요소, **wsdl:service** 요소 또는 **wsdl:port** 요소의 자식으로 배치할 수 있습니다. **SOAP/JMS** 요소의 상위는 다음 중 구성이 배치되는 범위를 결정합니다.*

바인딩 범위

***wsdl: binding** 요소 내에 확장 요소를 배치하여 바인딩 범위 에서 **JMS** 전송을 구성할 수 있습니다. 이 범위의 요소는 이 바인딩을 사용하는 모든 엔드포인트에 대한 기본 구성을 정의합니다. 바인딩 범위의 설정은 서비스 범위 또는 포트 범위에서 재정의할 수 있습니다.*

서비스 범위

***wsdl: service** 요소 내에 확장 요소를 배치하여 서비스 범위 에서 **JMS** 전송을 구성할 수 있습니다. 이 범위의 요소는 이 서비스의 모든 엔드포인트에 대한 기본 구성을 정의합니다. 서비스 범위의 모든 설정은 포트 범위에서 재정의할 수 있습니다.*

포트 범위

wsdl:port 요소 내에 확장 요소를 배치하여 포트 범위 에서 **JMS** 전송을 구성할 수 있습니다. 포트 범위의 요소는 이 포트에 대한 구성을 정의합니다. 서비스 범위 또는 바인딩 범위에서 정의된 동일한 확장 요소의 기본값을 재정의합니다.

13.3.5. 예제

예 13.9. "WSDL과 SOAP/JMS 구성의 계약" **SOAP/JMS** 서비스에 대한 **WSDL** 계약을 보여줍니다. 바인딩 범위, 서비스 범위에서 메시지 전달 세부 정보 및 포트 범위의 응답 대상에서 **JNDI** 계층을 구성합니다.

예 13.9. WSDL과 SOAP/JMS 구성의 계약

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
  ...
  <wsdl:binding name="JMSGreeterPortBinding" type="tns:JMSGreeterPortType">
    ...
    <soapjms:jndiInitialContextFactory>
      org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    ...
  </wsdl:binding>
  ...
  <wsdl:service name="JMSGreeterService">
    ...
    <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
    <soapjms:timeToLive>60000</soapjms:timeToLive>
    ...
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
      <soap:address location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
      <soapjms:replyToName>
        dynamicQueues/greeterReply.queue
      </soapjms:replyToName>
      ...
    </wsdl:port>
    ...
  </wsdl:service>
  ...
</wsdl:definitions>
```

예 13.9. "WSDL과 SOAP/JMS 구성의 계약" 의 **WSDL**은 다음을 수행합니다.

SOAP/JMS 확장의 네임스페이스를 선언합니다.

바인딩 범위에서 JNDI 연결을 구성합니다.

JMS 전달 스타일을 비영구로 설정하고 각 메시지를 1분 동안 활성으로 설정합니다.

대상 대상을 지정합니다.

응답 메시지가 greeterReply.queue 큐에 전송되도록 JMS 전송을 구성합니다.

14장. 일반 JMS 사용

초록

Apache CXF는 JMS 전송의 일반적인 구현을 제공합니다. 일반 JMS 전송은 SOAP 메시지 사용으로 제한되지 않으며 JMS를 사용하는 모든 애플리케이션에 연결할 수 있습니다.

참고: CXF 3.0에서는 JMS 1.0.2 API에 대한 지원이 제거되었습니다. RedHat JBoss Fuse 6.2 이상을 사용하는 경우(CXF 3.0 포함) JMS 공급자는 JMS 1.1 API를 지원해야 합니다.

14.1. JMS 구성 방법

Apache CXF 일반 JMS 전송은 모든 JMS 공급자에 연결하고 JMS 메시지를 TextMessage 또는 ByteMessage 의 본문과 교환하는 애플리케이션으로 작업할 수 있습니다.

JMS 전송을 활성화하고 구성하는 방법에는 두 가지가 있습니다.

- 14.2절. “JMS 구성 빈 사용”
- 14.5절. “WSDL을 사용하여 JMS 구성”

14.2. JMS 구성 빈 사용

14.2.1. 개요

JMS 구성을 단순화하고 더 강력하게 만들기 위해 Apache CXF는 단일 JMS 구성 빈을 사용하여 JMS 끝점을 구성합니다. 빈은 org.apache.cxf.transport.jms.JMSConfiguration 클래스에서 구현됩니다. 끝점을 직접 구성하거나 JMS 구성 및 대상을 구성하는 데 사용할 수 있습니다.

14.2.2. 구성 네임스페이스

JMS 구성 빈에서는 Spring p-namespace 를 사용하여 구성을 최대한 간단하게 만듭니다. 이 네임스페이스를 사용하려면 예 14.1. “Spring p-namespace 선언” 과 같이 구성의 루트 요소에 선언해야 합니다.

예 14.1. Spring p-namespace 선언

```
-
```

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ... >
  ...
</beans>
```

14.2.3. 설정 지정

org.apache.cxf.transport.jms.JMSConfiguration 클래스의 빈을 정의하여 **JMS** 구성을 지정합니다. 빈의 속성은 전송에 대한 구성 설정을 제공합니다.

중요

CXF 3.0에서는 **JMS** 전송에 더 이상 **Spring JMS**에 종속되지 않으므로 일부 **Spring JMS** 관련 옵션이 제거되었습니다.

표 14.1. "일반 JMS 구성 속성" 공급자와 소비자에게 공통된 속성을 나열합니다.

표 14.1. 일반 JMS 구성 속성

속성	기본값	설명
connectionFactory		[필수] JMS ConnectionFactory를 정의하는 빈에 대한 참조를 지정합니다.
wrapInSingleConnectionFactory	**true** [pre v3.0]	**CXF 3.0에서 삭제** CXF 3.0 **pre CXF 3.0** 은 Spring **SingleConnectionFactory** 로 ConnectionFactory를 래핑할지 여부를 지정합니다. 연결을 풀링하지 않는 ConnectionFactory를 사용할 때 JMS 전송의 성능이 향상되므로 이 속성을 활성화합니다. JMS 전송에서 각 메시지에 대한 새 연결을 생성하고 연결을 캐시하려면 **SingleConnectionFactory** 가 필요하므로 재사용할 수 있습니다.

속성	기본값	설명
reconnectOnException	**false**	**CXF 3.0에서 더 이상 사용되지 않는** CXF는 예외가 발생할 때 항상 다시 연결합니다. **pre CXF 3.0** 예외 발생 시 새 연결을 만들지 여부를 지정합니다. Spring **SingleConnectionFactory** 를 사용하여 ConnectionFactory를 래핑하는 경우: ● **True** easingon an exception, create a new connection PooledConnectionFactory를 사용할 때는 이 옵션을 활성화하지 마십시오. 이 옵션은 pooled 연결만 반환하지만 다시 연결하지 않기 때문입니다. ● 허 **위** - 예외로, 다시 연결을 시도하지 마십시오.
targetDestination		대상의 JNDI 이름 또는 공급자별 이름을 지정합니다.
replyDestination		응답이 전송되는 JMS 대상의 JMS 이름을 지정합니다. 이 속성을 사용하면 응답에 사용자 정의 대상을 사용할 수 있습니다. 자세한 내용은 14.6절. "이름이 지정된 Reply Destination 사용" 에서 참조하십시오.
destinationResolver	DynamicDestinationResolver	Spring **DestinationResolver** 에 대한 참조를 지정합니다. 이 속성을 사용하면 대상 이름이 JMS 대상으로 확인되는 방법을 정의할 수 있습니다. 유효한 값은 다음과 같습니다. ● **DynamicDestinationResolver** JMS 공급자의 기능을 사용하여 대상 이름 확인. ● **JndiDestinationResolver** JNDI를 사용하여 대상 이름을 확인.

속성	기본값	설명
transactionManager		Spring 트랜잭션 관리자에 대한 참조를 지정합니다. 이를 통해 서비스는 JTA 트랜잭션에 참여할 수 있습니다.
taskExecutor	**SimpleAsyncTaskExecutor**	**CXF 3.0에서 삭제** CXF 3.0 **preCXF 3.0** Spring TaskExecutor에 대한 참조를 지정합니다. 이는 리스너에서 수신되는 메시지를 처리하는 방법을 결정하는 데 사용됩니다.
useJms11	**false**	**CXF 3.0에서 제거된** CXF 3.0에서는 JMS 1.1 기능만 지원합니다. **pre CXF 3.0** 에서는 JMS 1.1 기능이 사용되는지 여부를 지정합니다. 유효한 값은 다음과 같습니다. ● 진 **정한** JMS 1.1 기능 ● **false** – JMS 1.0.2 features
messageIdEnabled	**true**	**CXF 3.0에서 삭제** **pre CXF 3.0** 은 JMS 전송에서 JMS 브로커가 메시지 ID를 제공할지 여부를 지정합니다. 유효한 값은 다음과 같습니다. ● **True** databind-broker는 메시지 ID를 제공해야 합니다. ● **false** ECDHE-ECDHEbroker에 메시지 ID를 제공할 필요가 없습니다. 이 경우 끝점은 메시지 생산자의 **setDisableMessageID()** 메서드를 **true** 로 호출합니다. 그러면 브로커에 메시지 ID를 생성하거나 엔드포인트의 메시지에 추가할 필요가 없다는 힌트가 제공됩니다. 브로커는 힌트를 수락하거나 무시합니다.

속성	기본값	설명
messageTimestampEnabled	**true**	**CXF 3.0에서 삭제** CXF 3.0 **pre CXF 3.0** 은 JMS 전송에서 JMS 브로커가 메시지 타임스탬프를 제공할지 여부를 지정합니다. 유효한 값은 다음과 같습니다. ● **True** ep-Progressbroker는 메시지 타임스탬프를 제공해야 합니다. ● **false** databind-ECDHEbroker는 메시지 타임스탬프를 제공할 필요가 없습니다. 이 경우 끝점은 메시지 생산자의 **setDisableMessageTimestamp()** 메서드를 **true** 로 호출합니다. 그러면 브로커가 타임스탬프를 생성하거나 엔드포인트의 메시지에 추가할 필요가 없다는 힌트가 제공됩니다. 브로커는 힌트를 수락하거나 무시합니다.
cacheLevel	**-1(** 기능 비활성화)	**CXF 3.0에서 삭제** **pre CXF 3.0** JMS 리스너 컨테이너가 적용할 수 있는 캐싱 수준을 지정합니다. 유효한 값은 다음과 같습니다. ● **0** – CACHE_NONE ● **1** – CACHE_CONNECTION ● **2** – CACHE_SESSION ● **3** – CACHE_CONSUMER ● **4** – CACHE_AUTO 자세한 내용은 Class DefaultMessageListenerContainer를 참조하십시오.

속성	기본값	설명
pubSubNoLocal	**false**	주제를 사용할 때 자체 메시지를 수신할지 여부를 지정합니다. ● **True** databind-Do not receive your own messages ● **거** 짓 - 자신의 메시지를 받을 수 있습니다.
receiveTimeout	**60000**	응답 메시지를 대기하는 시간(밀리초)을 지정합니다.
explicitQosEnabled	**false**	각 메시지(**true**)에 대해 QoS 설정(예: 우선 순위, 지속성, 시간)이 명시적으로 설정되는지 또는 기본값(**false**)을 사용할지 여부를 지정합니다.
deliveryMode	**2**	메시지가 영구적인지 여부를 지정합니다. 유효한 값은 다음과 같습니다. ● **1** (NON_PERSISTENT)-messages는 메모리만 유지 ● **2** (PERSISTENT) - 디스크에 유지됨
priority	**4**	메시지 우선 순위를 지정합니다. JMS 우선 순위 값의 범위는 **0** (최저)에서 **9** (최고)입니다. 자세한 내용은 JMS 공급자 설명서를 참조하십시오.
timeToLive	**0** (Indefinitely)	전송된 메시지가 삭제되기 전의 시간(밀리초)을 지정합니다.
sessionTransacted	**false**	JMS 트랜잭션이 사용되는지 여부를 지정합니다.
concurrentConsumers	**1**	**CXF 3.0에서 삭제** **pre CXF 3.0** 은 리스너에 대한 최소 동시 소비자 수를 지정합니다.

속성	기본값	설명
maxConcurrentConsumers	**1**	CXF 3.0에서 삭제 pre CXF 3.0 은 리스너에 대한 최대 동시 소비자 수를 지정합니다.
messageSelector		들어오는 메시지를 필터링하는 데 사용되는 선택기의 문자열 값을 지정합니다. 이 속성을 사용하면 여러 개의 연결이 큐를 공유할 수 있습니다. 메시지 선택기를 지정하는 데 사용되는 구문에 대한 자세한 내용은 JMS 1.1 사양 을 참조하십시오.
subscriptionDurable	**false**	서버가 내구성 있는 서브스크립션을 사용하는지 여부를 지정합니다.
durableSubscriptionName		내구성 서브스크립션을 등록하는 데 사용되는 이름(문자열)을 지정합니다.
messageType	**text**	메시지 데이터를 JMS 메시지로 패키징하는 방법을 지정합니다. 유효한 값은 다음과 같습니다. ● **텍스트**: 데이터가 **TextMessage**로 패키징됨을 나타냅니다. ● 바이트 배열(**byte** [])으로 데이터가 패키징됨을 나타냅니다.Indicates that the data will be packaged as an array of bytes (**byte[]**) ● **바이너리** - 데이터가 **ByteMessage**로 패키지됨을 나타냅니다.
pubSubDomain	**false**	대상 대상이 항목 또는 큐인지 여부를 지정합니다.Specifies whether the target destination is a topic or a queue. 유효한 값은 다음과 같습니다. ● **true** additionaltopic ● **false** – queue

속성	기본값	설명
jmsProviderTibcoEms	**false**	JMS 공급자가 Tibco EMS인지 여부를 지정합니다. **true** 로 설정하면 보안 컨텍스트의 보안 주체가 **JMS_TIBCO_SENDER** 헤더에서 채워집니다.
useMessageIDAsCorrelationID	**false**	**CXF 3.0에서 삭제** JMS에서 메시지 ID를 사용하여 메시지의 상관 관계를 유지할지 여부를 지정합니다. 클라이언트는 **true** 로 설정하면 클라이언트는 생성된 상관관계 ID를 설정합니다.
maxSuspendedContinuations	**-1(** 기능 비활성화)	CXF 3.0 JMS 대상이 보유할 수 있는 최대 중단된 연속 수를 지정합니다. 현재 숫자가 지정된 최대값을 초과하면 JMSListenerContainer가 중지됩니다.
reconnectPercentOfMax	**70**	CXF 3.0 은 **maxSuspendedContinuations** 를 초과하기 위해 JMSListenerContainer를 다시 시작할 시기를 지정합니다. 현재 일시 중단된 연속 수가 **(maxSuspendedContinuations * reconnectPercentOfMax/100)** 이하인 경우 리스너 컨테이너가 재시작됩니다.

예 14.2. "JMS 구성 빈" 에 표시된 대로 빈의 속성은 빈 요소에 대한 속성으로 지정됩니다. 모두 Spring p 네임스페이스에 선언됩니다.

예 14.2. JMS 구성 빈

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
      p:connectionFactory="jmsConnectionFactory"
      p:targetDestination="dynamicQueues/greeter.request.queue"
      p:pubSubDomain="false" />
```

14.2.4. 끝점에 구성 적용

JMSConfiguration 빈은 **Apache CXF** 기능 메커니즘을 사용하여 서버 및 클라이언트 끝점 모두에 직접 적용할 수 있습니다. 이렇게 하려면 다음을 수행합니다.

1. 끝점의 **address** 속성을 **jms://** 로 설정합니다.
2. **jaxws:feature** 요소를 엔드포인트의 구성에 추가합니다.
3. 유형 **org.apache.cxf.transport.jms.JMSConfigFeature** 의 빈을 기능에 추가합니다.
4. 빈 요소의 **p:jmsConfig-ref** 속성을 **JMSConfiguration** 빈의 **ID**로 설정합니다.

예 14.3. “JAX-WS 클라이언트에 JMS 구성 추가” 는 예 14.2. “JMS 구성 빈” 의 **JMS** 구성을 사용하는 **JAX-WS** 클라이언트를 표시합니다.

예 14.3. JAX-WS 클라이언트에 JMS 구성 추가

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"
              serviceClass="com.example.customerservice.CustomerService">
  <jaxws:features>
    <bean xmlns="http://www.springframework.org/schema/beans"
          class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

14.2.5. 전송에 구성 적용

JMSConfiguration 빈은 **jms:jmsConfig-ref** 요소를 사용하여 **JMS** 구성 및 **JMS** 대상에 적용할 수 있습니다. **jms:jmsConfig-ref** 요소의 값은 **JMSConfiguration** 빈의 **ID**입니다.

예 14.4. “JMS 구성에 JMS 구성 추가” 는 예 14.2. “JMS 구성 빈” 의 **JMS** 구성을 사용하는 **JMS** 구성을 보여줍니다.

예 14.4. JMS 구성에 JMS 구성 추가

```
<jms:conduit name="{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-conduit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

14.3. 클라이언트-SIDE JMS 성능 최적화

14.3.1. 개요

두 가지 주요 설정은 풀링 및 동기 수신이라는 클라이언트의 JMS 성능에 영향을 미칩니다.

14.3.2. 풀링

클라이언트 측에서 CXF는 각 메시지에 대해 새로운 JMS 세션 및 JMS 생산자를 생성합니다. 이는 세션과 생산자 오브젝트가 모두 스레드 안전하지 않기 때문입니다. 생산자 생성은 서버와 통신해야 하므로 특히 시간이 많이 사용됩니다.

연결 팩토리를 풀링하면 연결, 세션 및 생산자를 캐시하여 성능이 향상됩니다.

ActiveMQ의 경우 풀링을 구성하는 것이 간단합니다. 예를 들면 다음과 같습니다.

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.pool.PooledConnectionFactory;

ConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
PooledConnectionFactory pcf = new PooledConnectionFactory();

//Set expiry timeout because the default (0) prevents reconnection on failure
pcf.setExpiryTimeout(5000);
pcf.setConnectionFactory(cf);

JMSConfiguration jmsConfig = new JMSConfiguration();

jmsConfig.setConnectionFactory(pdf);
```

풀링에 대한 자세한 내용은 Red Hat JBoss Fuse 트랜잭션 가이드의 "Appendix A Optimizing Performance of JMS Single- and Multiple-Resource Transactions"를 참조하십시오.

14.3.3. 동기 수신 방지

요청/복합의 경우 JMS 전송에서 요청을 보낸 다음 응답을 기다립니다. 가능한 경우 요청/응답 메시징은 JMS MessageListener 를 사용하여 비동기적으로 구현됩니다.

그러나 엔드포인트 간에 큐를 공유해야 하는 경우 CXF는 동기 소비자.receive() 메서드를 사용해야 합니다. 이 시나리오에서는 메시지 선택기를 사용하여 메시지를 필터링하기 위해 MessageListener 가 필요합니다. 메시지 선택기를 사전에 알고 있어야 하므로 MessageListener 는 한 번만 열립니다.

메시지 선택기를 사전에 알 수 없는 두 가지 경우는 피해야 합니다.

- JMSMessageID 가 JMSCorrelationID로 사용되는 경우

  JMS 속성이 ConduitIdSelector 및 conduitSelectorPrefix 를 사용하는 경우 클라이언트는 JMSCorrelationId 를 설정하지 않습니다. 이로 인해 서버에서 요청 메시지의 JMSMessageId 를 JMSCorrelationId 로 사용합니다. JMSMessageID 를 사전에 알 수 없기 때문에 클라이언트는 synchronous Consumer.receive() 메서드를 사용해야 합니다.

  IBM JMS 엔드포인트(기본값)에서 consumers.receive() 메서드를 사용해야 합니다.

- 사용자는 요청 메시지에 JMStype 을 설정한 다음 사용자 지정 JMSCorrelationID 를 설정합니다.

  사용자 지정 JMSCorrelationID 를 사전에 알 수 없기 때문에 클라이언트는 동기 소비자.receive() 메서드를 사용해야 합니다.

따라서 일반적인 규칙은 동기 수신을 사용해야 하는 설정을 사용하지 않는 것입니다.

14.4. JMS 트랜잭션 구성

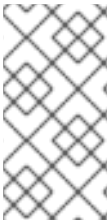
14.4.1. 개요

CXF 3.0은 단방향 메시징을 사용할 때 CXF 끝점에서 로컬 JMS 트랜잭션과 JTA 트랜잭션을 모두 지원합니다.

14.4.2. 로컬 트랜잭션

로컬 리소스를 사용하는 트랜잭션은 예외가 발생하는 경우에만 **JMS** 메시지를 롤백합니다. 데이터베이스 트랜잭션과 같은 다른 리소스를 직접 조정하지 않습니다.

로컬 트랜잭션을 설정하려면 일반적으로 끝점을 구성하고 속성 **sessionTrasnsacted** 를 **true** 로 설정합니다.



참고

트랜잭션 및 풀링에 대한 자세한 내용은 **Red Hat JBoss Fuse** 트랜잭션 가이드를 참조하십시오.

14.4.3. JTA 트랜잭션

JTA 트랜잭션을 사용하면 모든 **XA** 리소스를 조정할 수 있습니다. **CXF** 끝점이 **JTA** 트랜잭션용으로 구성된 경우 서비스 구현을 호출하기 전에 트랜잭션을 시작합니다. 예외가 발생하지 않으면 트랜잭션이 커밋됩니다. 그렇지 않으면 롤백됩니다.

JTA 트랜잭션에서는 **JMS** 메시지가 사용되고 데이터베이스에 기록된 데이터를 사용합니다. 예외가 발생하면 두 리소스가 모두 롤백되므로 메시지가 소비되고 데이터가 데이터베이스에 기록되거나 메시지가 롤백되고 데이터가 데이터베이스에 기록되지 않습니다.

JTA 트랜잭션을 구성하려면 다음 두 단계가 필요합니다.

1. 트랜잭션 관리자 정의
   - 빈 메서드
     - 트랜잭션 관리자 정의

```
<bean id="transactionManager"
class="org.apache.geronimo.transaction.manager.GeronimoTransactionManager"/>
```

     - **JMS URI**에서 트랜잭션 관리자의 이름을 설정합니다.

```
jms:queue:myqueue?jndiTransactionManager=TransactionManager
```

*이 예에서는 **ID TransactionManager** 가 있는 빈을 찾습니다.*

- ***OSGi** 참조 방법*
  - ***Blueprint**를 사용하여 **OSGi** 서비스로 트랜잭션 관리자를 조회합니다.*

```
<reference id="TransactionManager"
interface="javax.transaction.TransactionManager"/>
```

  - ***JMS URI**에서 트랜잭션 관리자의 이름을 설정합니다.*

```
jms:jndi:myqueue?jndiTransactionManager=java:comp/env/TransactionManager
```

*이 예에서는 **JNDI**에서 트랜잭션 관리자를 조회합니다.*

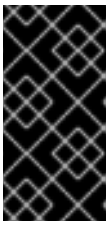
2. ***JCA** 풀링 연결 팩토리 구성*

***Spring**을 사용하여 **JCA** 풀링 연결 팩토리를 정의합니다.*

```
<bean id="xacf" class="org.apache.activemq.ActiveMQXAConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="ConnectionFactory"
class="org.apache.activemq.jms.pool.JcaPooledConnectionFactory">
  <property name="transactionManager" ref="transactionManager" />
  <property name="connectionFactory" ref="xacf" />
</bean>
```

*이 예에서 첫 번째 빈은 **JcaPooledConnectionFactory** 에 제공되는 **ActiveMQ XA** 연결 팩토리를 정의합니다. 그런 다음 **JcaPooledConnectionFactory** 가 **id ConnectionFactory** 와 기본 빈으로 제공됩니다.*

***JcaPooledConnectionFactory** 는 일반 **ConnectionFactory**처럼 보입니다. 그러나 새 연결 및 세션이 열리면 **XA** 트랜잭션을 확인하고 있는 경우 **JMS** 세션을 **XA** 리소스로 자동으로 등록합니다. 이를 통해 **JMS** 세션이 **JMS** 트랜잭션에 참여할 수 있습니다.*



중요

JMS 전송에서 XA ConnectionFactory를 직접 설정하는 것은 작동하지 않습니다!

14.5. WSDL을 사용하여 JMS 구성

14.5.1. JMSWS Extension Namespance

JMS 엔드포인트를 정의하는 WSDL 확장 기능은 네임스페이스 http://cxf.apache.org/transports/jms 에 정의되어 있습니다. JMS 확장을 사용하려면 예 14.5. “JMS WSDL 확장 네임스페이스” 에 표시된 줄을 계약의 정의 요소에 추가해야 합니다.

예 14.5. JMS WSDL 확장 네임스페이스

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

14.5.2. 기본 JMS 구성

14.5.2.1. 개요

JMS 주소 정보는 jms:address 요소 및 해당 하위인 jms:JMSNamingProperties 요소를 사용하여 제공됩니다. jms:address 요소의 속성은 JMS 브로커와 대상을 식별하는 데 필요한 정보를 지정합니다. jms:JMSNamingProperties 요소는 JNDI 서비스에 연결하는 데 사용되는 Java 속성을 지정합니다.

중요

JMS 기능을 사용하여 지정된 정보는 끝점의 WSDL 파일의 정보를 재정의합니다.

14.5.2.2. JMS 주소 지정

JMS 엔드포인트에 대한 기본 구성은 jms:address 요소를 서비스 포트 요소의 자식으로 사용하여 수행됩니다. WSDL에 사용된 jms:address 요소는 구성 파일에서 사용된 것과 동일합니다. 해당 속성은 표 14.2. “JMS 끝점 속성” 에 나열됩니다.

표 14.2. JMS 끝점 속성

속성	설명

속성	설명
destinationStyle	JMS 대상이 JMS 대기열 또는 JMS 주제인지 여부를 지정합니다.
jndiConnectionFactoryName	JMS 대상에 연결할 때 사용할 JMS 연결 팩토리에 바인딩된 JNDI 이름을 지정합니다.
jmsDestinationName	전송되는 JMS 대상의 JMS 이름을 지정합니다.
jmsReplyDestinationName	응답이 전송되는 JMS 대상의 JMS 이름을 지정합니다. 이 특성을 사용하면 응답에 대해 사용자 정의 대상을 사용할 수 있습니다. 자세한 내용은 14.6절. "이름이 지정된 Reply Destination 사용" 에서 참조하십시오.
jndiDestinationName	전송되는 JMS 대상에 바인딩된 JNDI 이름을 지정합니다.
jndiReplyDestinationName	응답이 전송되는 JMS 대상에 바인딩된 JNDI 이름을 지정합니다. 이 특성을 사용하면 응답에 대해 사용자 정의 대상을 사용할 수 있습니다. 자세한 내용은 14.6절. "이름이 지정된 Reply Destination 사용" 에서 참조하십시오.
connectionUserName	JMS 브로커에 연결할 때 사용할 사용자 이름을 지정합니다.
connectionPassword	JMS 브로커에 연결할 때 사용할 암호를 지정합니다.

jms:address WSDL 요소는 jms:JMSNamingProperties 하위 요소를 사용하여 JNDI 공급자에 연결하는 데 필요한 추가 정보를 지정합니다.

14.5.2.3. JNDI 속성 지정

JMS 및 JNDI 공급자와의 상호 운용성을 높이기 위해 jms:address 요소에는 JNDI 공급자에 연결할 때 사용되는 속성을 채우는 데 사용되는 값을 지정할 수 있는 하위 요소 jms:JMSNamingProperties 가 있습니다. jms:JMSNamingProperties 요소에는 name 과 value 의 두 가지 속성이 있습니다. name 은 설정할 속성의 이름을 지정합니다. value 속성은 지정된 속성의 값을 지정합니다.
JMS:JMSNamingProperties 요소는 공급자별 속성의 사양에도 사용할 수 있습니다.

다음은 설정할 수 있는 공통 JNDI 속성 목록입니다.

1. ***java.naming.factory.initial***

2. ***java.naming.provider.url***

3. ***java.naming.factory.object***

4. ***java.naming.factory.state***

5. ***java.naming.factory.url.pkgs***

6. ***java.naming.dns.url***

7. ***java.naming.authoritative***

8. ***java.naming.batchsize***

9. ***java.naming.referral***

10. ***java.naming.security.protocol***

11. ***java.naming.security.authentication***

12. ***java.naming.security.principal***

13. ***java.naming.security.credentials***

14. ***java.naming.language***

15. ***java.naming.applet***

이러한 속성에서 사용할 정보에 대한 자세한 내용은 JNDI 공급자의 설명서를 확인하고 Java API 참조 자료를 참조하십시오.

14.5.2.4. 예제

예 14.6. "JMS WSDL 포트 사양" 는 JMS WSDL 포트 사양의 예를 보여줍니다.

예 14.6. JMS WSDL 포트 사양

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

14.5.3. JMS 클라이언트 구성

14.5.3.1. 개요

JMS 소비자 엔드포인트는 사용하는 메시지 유형을 지정합니다. JMS 소비자 끝점은 JMS ByteMessage 또는 JMS TextMessage 를 사용할 수 있습니다.

ByteMessage 를 사용할 때 소비자 엔드포인트는 JMS 메시지 본문에 데이터를 저장하고 데이터를 검색하는 방법으로 byte[] 를 사용합니다. 메시지가 전송될 때, 포맷된 정보를 포함하는 메시지 데이터는 바이트[] 로 패키지되고, 그 메시지가 전선에 배치되기 전에 메시지 본문에 배치된다. 메시지가 수신되면 소비자 끝점은 바이트[] 에 패키징된 것처럼 메시지 본문에 저장된 데이터를 매끄럽게 표시하려고 합니다.

TextMessage 를 사용하는 경우 소비자 엔드포인트는 메시지 본문에서 데이터를 저장하고 검색하는 방법으로 문자열을 사용합니다. 메시지가 전송되면 모든 형식별 정보를 포함하는 메시지 정보가 문자열로 변환되어 JMS 메시지 본문에 배치됩니다. 메시지가 수신되면 소비자 끝점은 JMS 메시지 본문에 저장된 데이터를 문자열로 패딩하려고 합니다.

네이티브 JMS 애플리케이션이 Apache CXF 소비자와 상호 작용할 때 JMS 애플리케이션은 메시지 및 형식 정보를 해석해야 합니다. 예를 들어 Apache CXF 계약이 JMS 끝점에 사용되는 바인딩이 SOAP 임을 지정하고 메시지가 TextMessage 로 패키징된 경우 수신 JMS 애플리케이션은 모든 SOAP 봉투 정보가 포함된 텍스트 메시지를 받습니다.

14.5.3.2. 메시지 유형 지정

JMS 소비자 끝점에서 허용하는 메시지 유형은 선택적 **jms:client** 요소를 사용하여 구성됩니다. **jms:client** 요소는 **WSDL** 포트 요소의 자식이며 하나의 특성을 갖습니다.

표 14.3. JMS Client WSDL Extensions

messageType
메시지 데이터를 JMS 메시지로 패키징하는 방법을 지정합니다. **text** 는 데이터가 **TextMessage** 로 패키징되도록 지정합니다. **바이너리** 는 데이터가 **ByteMessage** 로 패키징되도록 지정합니다.

14.5.3.3. 예제

예 14.7. "JMS 소비자 엔드 포인트의 WSDL" 는 JMS 소비자 엔드 포인트를 구성하기 위한 WSDL을 보여줍니다.

예 14.7. JMS 소비자 엔드 포인트의 WSDL

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:client messageType="binary" />
  </port>
</service>
```

14.5.4. JMS 공급자 구성

14.5.4.1. 개요

JMS 공급자 끝점에는 구성 가능한 여러 동작이 있습니다. 여기에는 다음이 포함됩니다.

- 메시지가 상호 작용하는 방법
- 내구성 서브스크립션 사용

- *서비스가 로컬 JMS 트랜잭션을 사용하는 경우*
- *끝점에서 사용하는 메시지 선택기*

14.5.4.2. 설정 지정

*공급자 끝점 동작은 선택적 **jms:server** 요소를 사용하여 구성됩니다. **jms:server** 요소는 WSDL **wsdl:port** 요소의 하위이며 다음 속성이 있습니다.*

표 14.4. JMS 공급자 끝점 WSDL 확장

속성	설명
useMessageIDAsCorrealationID	JMS에서 메시지 ID를 사용하여 메시지의 상관 관계를 유지할지 여부를 지정합니다. 기본값은 **false**입니다.
durableSubscriberName	내구성 서브스크립션을 등록하는 데 사용되는 이름을 지정합니다.
messageSelector	사용할 메시지 선택기의 문자열 값을 지정합니다. 메시지 선택기를 지정하는 데 사용되는 구문에 대한 자세한 내용은 JMS 1.1 사양을 참조하십시오.
transactional	로컬 JMS 브로커가 메시지 처리와 관련된 트랜잭션을 생성할지 여부를 지정합니다. 기본값은 **false**입니다.[a]

[a] 현재 **트랜잭션** 특성을 **true** 로 설정하는 것은 런타임에서 지원되지 않습니다.

14.5.4.3. 예제

예 14.8. "JMS 공급자 끝점의 WSDL" 는 JMS 공급자 끝점 구성을 위한 WSDL을 보여줍니다.

예 14.8. JMS 공급자 끝점의 WSDL

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
```

```
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

14.6. 이름이 지정된 REPLY DESTINATION 사용

14.6.1. 개요

기본적으로 **JMS**를 사용하는 **Apache CXF** 엔드포인트는 응답을 백으로 전송하기 위한 임시 대기열을 생성합니다. 명명된 대기열을 사용하려면 엔드포인트의 **JMS** 구성의 일부로 응답을 보내는 데 사용되는 큐를 구성할 수 있습니다.

14.6.2. 응답 대상 이름 설정

끝점의 **JMS** 구성에서 **jmsReplyDestinationName** 속성 또는 **jndiReplyDestinationName** 속성을 사용하여 응답 대상을 지정합니다. 클라이언트 끝점은 지정된 대상에 대한 응답을 수신 대기하며 나가는 모든 요청의 **ReplyTo** 필드에 속성 값을 지정합니다. 서비스 끝점은 요청의 **ReplyTo** 필드에 지정된 대상이 없는 경우 응답 배치 위치로 **jndiReplyDestinationName** 속성 값을 사용합니다.

14.6.3. 예제

예 14.9. “명명된 응답 대기열을 사용하여 JMS 소비자 사양” 는 **JMS** 클라이언트 끝점의 구성을 표시합니다.

예 14.9. 명명된 응답 대기열을 사용하여 JMS 소비자 사양

```
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
  <jms:address destinationStyle="queue"
               jndiConnectionFactoryName="myConnectionFactory"
               jndiDestinationName="myDestination"
               jndiReplyDestinationName="myReplyDestination" >
    <jms:JMSNamingProperty name="java.naming.factory.initial"
                           value="org.apache.cxf.transport.jms.MyInitialContextFactory" />
    <jms:JMSNamingProperty name="java.naming.provider.url"
                           value="tcp://localhost:61616" />
  </jms:address>
</jms:conduit>
```

15장. APACHE ACTIVEMQ와 통합

15.1. 개요

Apache ActiveMQ를 JMS 공급자로 사용하는 경우 대상의 JNDI 이름을 큐 또는 토픽에 대해 동적으로 JNDI 바인딩을 생성하는 특수 형식으로 지정할 수 있습니다. 즉, 대기열 또는 토픽에 대한 JNDI 바인딩을 사용하여 JMS 공급자를 미리 구성할 필요가 없습니다.

15.2. 초기 컨텍스트 팩토리

Apache ActiveMQ를 JNDI와 통합하는 핵심은 ActiveMQInitialContextFactory 클래스입니다. 이 클래스는 JNDI InitialContext 인스턴스를 생성하는 데 사용되며 JMS 브로커의 JMS 대상에 액세스할 수 있습니다.

예 15.1. “Apache ActiveMQ에 연결하는 SOAP/JMS WSDL” 는 Apache ActiveMQ와 통합된 JNDI InitialContext 를 생성하는 SOAP/JMS WSDL 확장 기능을 보여줍니다.

> 예 15.1. Apache ActiveMQ에 연결하는 SOAP/JMS WSDL
>
> ```
> <soapjms:jndiInitialContextFactory>
>  org.apache.activemq.jndi.ActiveMQInitialContextFactory
> </soapjms:jndiInitialContextFactory>
> <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
> ```

예 15.1. “Apache ActiveMQ에 연결하는 SOAP/JMS WSDL” 에서 Apache ActiveMQ 클라이언트는 tcp://localhost:61616 에 있는 브로커 포트에 연결합니다.

15.3. 연결 팩토리 검색

JNDI InitialContext 인스턴스를 생성하고 javax.jms.ConnectionFactory 인스턴스에 바인딩된 JNDI 이름을 지정해야 합니다. Apache ActiveMQ의 경우 JNDI 이름 ConnectionFactory 를 ActiveMQConnectionFactory 인스턴스에 매핑하는 InitialContext 인스턴스에 사전 정의된 바인딩이 있습니다. 예 15.2. “Apache ActiveMQ 연결 팩토리 지정을 위한 SOAP/JMS WSDL” Apache ActiveMQ 연결 팩토리를 지정하기 위한 SOAP/JMS 확장 요소.

> 예 15.2. Apache ActiveMQ 연결 팩토리 지정을 위한 SOAP/JMS WSDL
>
> ```
> <soapjms:jndiConnectionFactoryName>
>  ConnectionFactory
> </soapjms:jndiConnectionFactoryName>
> ```

15.4. 동적 대상의 구문

대기열 또는 토픽에 동적으로 액세스하려면 대상의 **JNDI** 이름을 다음 형식 중 하나로 **JNDI** 복합 이름으로 지정합니다.

```
dynamicQueues/QueueName
dynamicTopics/TopicName
```

QueueName 및 **TopicName** 은 **Apache ActiveMQ** 브로커가 사용하는 이름입니다. **JNDI** 이름은 추상화 되지 않습니다.

예 15.3. "동적으로 생성된 큐를 사용한 WSDL 포트 사양" 동적으로 생성된 큐를 사용하는 **WSDL** 포트를 표시합니다.

예 15.3. 동적으로 생성된 큐를 사용한 **WSDL** 포트 사양

```
<service name="JMSService">
  <port binding="tns:GreeterBinding" name="JMSPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"
                 jndiDestinationName="dynamicQueues/greeter.request.queue" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
                             value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

애플리케이션이 **JMS** 연결을 열려고 하면 **Apache ActiveMQ**는 **JNDI** 이름 **greeter.request.queue** 가 있는지 확인합니다. 존재하지 않는 경우 새 큐를 생성하고 **JNDI** 이름 **greeter.request.queue** 에 바인딩합니다.

16장. 구성 요소

초록

conduits는 아웃바운드 연결을 구현하는 데 사용되는 낮은 수준의 전송 아키텍처입니다. 해당 동작 및 라이프사이클은 시스템 성능 및 처리 로드에 영향을 미칠 수 있습니다.

16.1. 개요

Conduits는 **Apache CXF** 런타임의 클라이언트 측 또는 아웃바운드 정보를 관리합니다. 포트 열기, 아웃바운드 연결 설정, 메시지 전송, 애플리케이션 및 단일 외부 끝점 간의 모든 응답을 수신 대기해야 합니다. 애플리케이션이 여러 끝점에 연결하는 경우 각 끝점에 대해 하나의 구성 인스턴스가 있습니다.

각 전송 유형은 **Conduit** 인터페이스를 사용하여 자체 **conduit**을 구현합니다. 이를 통해 애플리케이션 수준 기능과 전송 간의 표준화된 인터페이스가 가능합니다.

일반적으로 클라이언트 측 전송 세부 정보를 구성할 때 애플리케이션에서 사용 중인 문제에 대해 우려할 필요가 있습니다. 런타임에서 유추를 처리하는 방식의 근본적인 의미는 일반적으로 개발자가 걱정해야 하는 것이 아닙니다.

그러나 영영을 이해하는 것이 도움이 될 수있는 경우가 있습니다.

- 사용자 정의 전송 구현
- 제한된 리소스 관리를 위한 고급 애플리케이션 튜닝

16.2. CONDUIT 라이프 사이클

Conduits는 클라이언트 구현 개체에 의해 관리됩니다. 만든 후 클라이언트 구현 개체의 기간 동안 제한이 있습니다.**Once created, a conduit lives for the duration of the client implementation object.** 클러스터의 라이프 사이클은 다음과 같습니다.

1. 클라이언트 구현 개체가 생성되면 **ConduitSelector** 오브젝트에 대한 참조가 제공됩니다.
2. 클라이언트가 메시지를 전송해야 하는 경우, 요청의 요청자(**Conduit selector**)의 구성 요소에

대한 참조입니다.

*메시지가 새 끝점에 대한 경우 구성 선택기는 새 **conduit**을 생성하여 클라이언트 구현에 전달합니다. 그렇지 않으면 대상 끝점에 대한 참조를 클라이언트에 전달합니다.*

3. *필요한 경우 구성 요소는 메시지를 보냅니다.*

4. *클라이언트 구현 개체를 삭제하면 연결된 모든 구성 요소가 삭제됩니다.*

16.3. 유도 가중치

***conduit** 오브젝트의 가중치는 전송 구현에 따라 다릅니다. **HTTP** 고무는 매우 가벼운 무게입니다. **JMS**는 **JMS Session** 오브젝트 및 하나 이상의 **JMSListenerContainer** 오브젝트와 연관되므로 많은 경우가 많습니다.*

IV 부. 웹 서비스 엔드 포인트 구성

*이 가이드에서는 **Red Hat Fuse**에서 **Apache CXF** 엔드포인트를 생성하는 방법을 설명합니다.*

17장. JAX-WS 엔드포인트 구성

초록

JAX-WS 엔드포인트는 3개의 Spring 구성 요소 중 하나를 사용하여 구성됩니다. 올바른 요소는 구성할 끝점 유형 및 사용하려는 기능에 따라 달라집니다. 소비자의 경우 jaxws:client 요소를 사용합니다. 서비스 공급자의 경우 jaxws:endpoint 요소 또는 jaxws:server 요소를 사용할 수 있습니다.

끝점을 정의하는 데 사용되는 정보는 일반적으로 끝점의 계약에 정의되어 있습니다. 구성 요소의 를 사용하여 계약의 정보를 재정의할 수 있습니다. 구성 요소를 사용하여 계약에 제공되지 않는 정보를 제공할 수도 있습니다.

구성 요소를 사용하여 WS-RM과 같은 고급 기능을 활성화해야 합니다. 이 작업은 하위 요소를 엔드포인트의 구성 요소에 제공하여 수행됩니다. Java 우선 접근 방식을 사용하여 끝점을 사용하여 개발된 경우 끝점 계약 역할을 하는 SEI에 사용할 바인딩 및 전송 유형에 대한 정보가 부족할 수 있습니다.

17.1. 서비스 공급자 구성

17.1.1. 서비스 공급자 구성을 위한 요소

Apache CXF에는 서비스 공급자를 구성하는 데 사용할 수 있는 두 가지 요소가 있습니다.

- 17.1.2절. “jaxws:endpoint Element 사용”
- 17.1.3절. “jaxws:server 요소 사용”

두 요소 간의 차이점은 대부분 런타임 내부에 있습니다. jaxws:endpoint 요소는 서비스 끝점을 지원하기 위해 생성된 org.apache.cxf.jaxws.EndpointImpl 오브젝트에 속성을 주입합니다. jaxws:server 요소는 끝점을 지원하기 위해 생성된 org.apache.cxf.jaxws.support.JaxWsServerFactoryBean 오브젝트에 속성을 삽입합니다. EndpointImpl 오브젝트는 구성 데이터를 JaxWsServerFactoryBean 오브젝트에 전달합니다. JaxWsServerFactoryBean 오브젝트는 실제 서비스 오브젝트를 생성하는 데 사용됩니다. 두 구성 요소는 서비스 끝점을 구성하기 때문에 원하는 구문에 따라 선택할 수 있습니다.

17.1.2. jaxws:endpoint Element 사용

17.1.2.1. 개요

jaxws:endpoint 요소는 JAX-WS 서비스 공급자를 구성하기 위한 기본 요소입니다. 특성 및 자식은

서비스 공급자를 인스턴스화하는 데 필요한 모든 정보를 지정합니다. 많은 속성은 서비스 계약의 정보에 매핑됩니다. 하위 항목은 인터셉터 및 기타 고급 기능을 구성하는 데 사용됩니다.

17.1.2.2. 구성되는 끝점 식별

런타임에서 적절한 서비스 공급자에 구성을 적용하려면 해당 구성을 식별할 수 있어야 합니다. 서비스 공급자를 식별하는 기본 방법은 끝점을 구현하는 클래스를 지정하는 것입니다. 이 작업은 **jaxws:endpoint** 요소의 구현자를 사용하여 수행됩니다.

서로 다른 엔드포인트가 공통 구현을 공유하는 인스턴스의 경우 각 엔드포인트에 대해 서로 다른 구성을 제공할 수 있습니다. 구성에서 특정 끝점을 구분하는 방법은 다음 두 가지입니다.

- **serviceName** 속성 및 **endpointName** 속성의 조합

  serviceName 속성은 서비스의 엔드포인트를 정의하는 **wsdl:service** 요소를 지정합니다. **endpointName** 속성은 서비스의 엔드포인트를 정의하는 특정 **wsdl:port** 요소를 지정합니다. 두 속성 모두 **ns:name** 형식을 사용하여 **QNames**로 지정됩니다. **NS** 는 요소의 네임스페이스이고 **name** 은 요소의 **name** 특성 값입니다.

  참고

  wsdl:service 요소에 하나의 **wsdl:port** 요소만 있는 경우 **endpointName** 속성을 생략할 수 있습니다.

- **name** 속성

  name 속성은 서비스의 엔드포인트를 정의하는 특정 **wsdl:port** 요소의 **QName**을 지정합니다. **QName**은 **{ns}localPart** 형식으로 제공됩니다. **NS** 는 **wsdl:port** 요소의 네임스페이스이고 **localPart** 는 **wsdl:port** 요소의 **name** 속성 값입니다.

17.1.2.3. 속성

jaxws:endpoint 요소의 속성은 끝점의 기본 속성을 구성합니다. 이러한 속성에는 끝점 주소, 엔드포인트를 구현하는 클래스 및 엔드포인트를 호스팅하는 버스 가 포함됩니다.

표 17.1. "jaxws:endpoint Element를 사용하여 JAX-WS 서비스 공급자 구성 특성" **jaxws:endpoint** 요소의 속성을 설명합니다.

표 17.1. jaxws:endpoint Element를 사용하여 JAX-WS 서비스 공급자 구성 특성

속성	설명
id	다른 구성 요소에서 끝점을 참조하는 데 사용할 수 있는 고유 식별자를 지정합니다.Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
구현자	서비스를 구현하는 클래스를 지정합니다. 구현 클래스를 구성하는 Spring 빈에 대한 클래스 이름 또는 ID 참조를 사용하여 구현 클래스를 지정할 수 있습니다. 이 클래스는 classpath에 있어야 합니다.
implementorClass	서비스를 구현하는 클래스를 지정합니다. 이 속성은 **구현** 자 속성에 제공된 값이 Spring AOP를 사용하여 래핑된 빈에 대한 참조인 경우에 유용합니다.
주소	HTTP 끝점의 주소를 지정합니다. 이 값은 서비스 계약에 지정된 값을 재정의합니다.
wsdlLocation	끝점의 WSDL 계약의 위치를 지정합니다. WSDL 계약의 위치는 서비스가 배포되는 폴더를 기준으로 합니다.
endpointName	서비스의 **wsdl:port** 요소의 **name** 특성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:port** 요소의 네임스페이스입니다.
serviceName	서비스의 **wsdl:service** 요소의 **name** 특성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:service** 요소의 네임스페이스입니다.
publish	서비스가 자동으로 게시되어야 하는지 여부를 지정합니다. 이 값이 **false** 로 설정되면 개발자는 31장. *서비스 게시* 에 설명된 엔드포인트를 명시적으로 게시해야 합니다.
bus	서비스 엔드포인트를 관리하는 데 사용되는 버스를 구성하는 Spring 빈의 ID를 지정합니다. 이 기능은 공통 기능 세트를 사용하도록 여러 끝점을 구성할 때 유용합니다.
bindingUri	서비스에서 사용하는 메시지 바인딩의 ID를 지정합니다. 유효한 바인딩 ID 목록은 23장. *Apache CXF 바인딩 ID* 에서 제공됩니다.

속성	설명
name	서비스의 **wsdl:port** 요소의 stringified QName을 지정합니다. ***{ns}localPart*** 형식을 사용하여 QName으로 지정됩니다. *NS* 는 **wsdl:port** 요소의 네임스페이스이고 *localPart* 는 **wsdl:port** 요소의 **name** 속성 값입니다.
abstract	빈이 추상 빈인지 여부를 지정합니다. 추상화 빈은 구체적인 빈 정의를 위해 부모 역할을 하며 인스턴스화되지 않습니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 빈 공장이 빈을 인스턴스화하지 않도록 지시합니다.
depends-on	끝점이 인스턴스화되기 전에 종속되는 빈 목록을 지정합니다.
createdFromAPI	**Endpoint.publish()** 또는 **Service.getPort()** 와 같은 Apache CXF API를 사용하여 빈을 만들도록 지정합니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 다음이 수행됩니다. ● ID에 **.jaxws-endpoint** 를 추가하여 빈의 내부 이름을 변경합니다. ● 빈을 추상적으로 만듭니다.
publishedEndpointUrl	생성된 WSDL의 **address** 요소에 배치된 URL입니다. 이 값을 지정하지 않으면 **address** 특성 값이 사용됩니다. 이 속성은 "public" URL이 서비스가 배포된 URL과 같지 않은 경우 유용합니다.

*표 17.1. "jaxws:endpoint Element를 사용하여 JAX-WS 서비스 공급자 구성 특성" 에 나열된 속성 외에도 여러 **xmlns:shortName**속성을 사용하여 **endpointName** 및 **serviceName** 속성에 사용되는 네임스페이스를 선언해야 할 수 있습니다.*

17.1.2.4. 예제

*예 17.1. "Simple JAX-WS Endpoint 구성" 엔드포인트가 게시되는 주소를 지정하는 **JAX-WS** 끝점의 구성을 보여줍니다. 이 예제에서는 다른 모든 값에 기본값을 사용하거나 구현에서 주석에 값을 지정했다고 가정합니다.*

> ***예 17.1. Simple JAX-WS Endpoint 구성***
>
> ```
> <beans ...
> ```

```
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="example"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo" />
</beans>
```

예 17.2. "서비스 이름을 사용한 JAX-WS Endpoint 구성" 계약에 두 개의 서비스 정의가 포함된 JAX-WS 엔드포인트의 구성을 보여줍니다. 이 경우 serviceName 특성을 사용하여 인스턴스화할 서비스 정의를 지정해야 합니다.

예 17.2. 서비스 이름을 사용한 JAX-WS Endpoint 구성

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">

  <jaxws:endpoint id="example2"
                  implementor="org.apache.cxf.example.DemoImpl"
                  serviceName="samp:demoService2"
                  xmlns:samp="http://org.apache.cxf/wsdl/example" />

</beans>
```

xmlns:samp 속성은 WSDL 서비스 요소가 정의된 네임스페이스를 지정합니다.

예 17.3. "HTTP/2가 활성화된 JAX-WS Endpoint Configuration" HTTP/2가 활성화된 주소를 지정하는 JAX-WS 끝점의 구성을 보여줍니다.

Apache CXF의 HTTP/2 구성

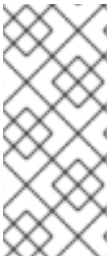
Apache Karaf에서 독립 실행형 Apache CXF Undertow 전송(http-undertow)을 사용하는 경우 HTTP/2가 지원됩니다. HTTP/2 프로토콜을 활성화하려면 jaxws:endpoint 요소의 address 속성을 절대 URL로 설정하고 org.apache.cxf.transports.http_undertow.EnableHttp2 속성을 true 로 설정해야 합니다.

참고

이 **HTTP/2** 구현은 일반 **HTTP** 또는 **HTTPS**와 함께 서버 측 **HTTP/2** 전송만 지원합니다.

예 17.3. HTTP/2가 활성화된 JAX-WS Endpoint Configuration

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
  http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
  ...">

  <cxf:bus>
    <cxf:properties>
       <entry key="org.apache.cxf.transports.http_undertow.EnableHttp2" value="true"/>
    </cxf:properties>
  </cxf:bus>

  <jaxws:endpoint id="example3"
              implementor="org.apache.cxf.example.DemoImpl"
              address="http://localhost:8080/demo" />
  </jaxws:endpoint>

</beans>
```



참고

성능 향상을 위해 **Red Hat**은 **Apache Karaf(pax-web-undertow)**의 서블릿 전송을 사용하여 웹 컨테이너의 중앙 집중식 구성 및 튜닝을 지원하지만 **pax-web-undertow** 는 **HTTP/2** 전송 프로토콜을 지원하지 않습니다.

17.1.3. jaxws:server 요소 사용

17.1.3.1. 개요

jaxws:server 요소는 **JAX-WS** 서비스 공급자를 구성하기 위한 요소입니다. 구성 정보를 **org.apache.cxf.jaxws.support.JaxWsServerFactoryBean** 에 삽입합니다. **Apache CXF** 특정 오브젝트입니다. 서비스를 구축하기 위해 순수 **Spring** 접근법을 사용하는 경우 **Apache CXF** 특정 **API**를 사용하여 서비스와 상호 작용할 필요가 없습니다.

jaxws:server 요소의 속성과 하위 요소는 서비스 공급자를 인스턴스화하는 데 필요한 모든 정보를 지정합니다. 속성은 엔드포인트를 인스턴스화하는 데 필요한 정보를 지정합니다. 하위 항목은 인터셉터 및

기타 고급 기능을 구성하는 데 사용됩니다.

17.1.3.2. 구성되는 끝점 식별

런타임에서 적절한 서비스 공급자에 구성을 적용하려면 해당 구성을 식별할 수 있어야 합니다. 서비스 공급자를 식별하는 기본 방법은 끝점을 구현하는 클래스를 지정하는 것입니다. 이 작업은 **jaxws:server** 요소의 **serviceBean** 속성을 사용하여 수행됩니다.

서로 다른 엔드포인트가 공통 구현을 공유하는 인스턴스의 경우 각 엔드포인트에 대해 서로 다른 구성을 제공할 수 있습니다. 구성에서 특정 끝점을 구분하는 방법은 다음 두 가지입니다.

- **serviceName** 속성 및 **endpointName** 속성의 조합

  serviceName 속성은 서비스의 엔드포인트를 정의하는 **wsdl:service** 요소를 지정합니다. **endpointName** 속성은 서비스의 엔드포인트를 정의하는 특정 **wsdl:port** 요소를 지정합니다. 두 속성 모두 **ns:name** 형식을 사용하여 **QNames**로 지정됩니다. **NS** 는 요소의 네임스페이스이고 **name** 은 요소의 **name** 특성 값입니다.

  참고

  wsdl:service 요소에 하나의 **wsdl:port** 요소만 있는 경우 **endpointName** 속성을 생략할 수 있습니다.

- **name** 속성

  name 속성은 서비스의 엔드포인트를 정의하는 특정 **wsdl:port** 요소의 **QName**을 지정합니다. **QName**은 **{ns}localPart** 형식으로 제공됩니다. **NS** 는 **wsdl:port** 요소의 네임스페이스이고 **localPart** 는 **wsdl:port** 요소의 **name** 속성 값입니다.

17.1.3.3. 속성

jaxws:server 요소의 속성은 끝점의 기본 속성을 구성합니다. 이러한 속성에는 끝점 주소, 엔드포인트를 구현하는 클래스 및 엔드포인트를 호스팅하는 버스 가 포함됩니다.

표 17.2. “jaxws:server Element를 사용하여 JAX-WS 서비스 공급자 구성 특성” **jaxws:server** 요소의 속성을 설명합니다.

표 17.2. jaxws:server Element를 사용하여 JAX-WS 서비스 공급자 구성 특성

속성	설명
id	다른 구성 요소에서 끝점을 참조하는 데 사용할 수 있는 고유 식별자를 지정합니다.Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
serviceBean	서비스를 구현하는 클래스를 지정합니다. 구현 클래스를 구성하는 Spring 빈에 대한 클래스 이름 또는 ID 참조를 사용하여 구현 클래스를 지정할 수 있습니다. 이 클래스는 classpath에 있어야 합니다.
serviceClass	서비스를 구현하는 클래스를 지정합니다. 이 속성은 **구현** 자 속성에 제공된 값이 Spring AOP를 사용하여 래핑된 빈에 대한 참조인 경우에 유용합니다.
주소	HTTP 끝점의 주소를 지정합니다. 이 값은 서비스 계약에 지정된 값을 재정의합니다.
wsdlLocation	끝점의 WSDL 계약의 위치를 지정합니다. WSDL 계약의 위치는 서비스가 배포되는 폴더를 기준으로 합니다.
endpointName	서비스의 **wsdl:port** 요소의 **name** 특성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:port** 요소의 네임스페이스입니다.
serviceName	서비스의 **wsdl:service** 요소의 **name** 특성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:service** 요소의 네임스페이스입니다.
publish	서비스가 자동으로 게시되어야 하는지 여부를 지정합니다. 이 값이 **false** 로 설정되면 개발자는 31장. *서비스 게시* 에 설명된 엔드포인트를 명시적으로 게시해야 합니다.
bus	서비스 엔드포인트를 관리하는 데 사용되는 버스를 구성하는 Spring 빈의 ID를 지정합니다. 이 기능은 공통 기능 세트를 사용하도록 여러 끝점을 구성할 때 유용합니다.
bindingId	서비스에서 사용하는 메시지 바인딩의 ID를 지정합니다. 유효한 바인딩 ID 목록은 23장. *Apache CXF 바인딩 ID* 에서 제공됩니다.
name	서비스의 **wsdl:port** 요소의 stringified QName을 지정합니다. ***{ns}localPart*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:port** 요소의 네임스페이스이고 *localPart* 는 **wsdl:port** 요소의 **name** 속성 값입니다.

속성	설명
abstract	빈이 추상 빈인지 여부를 지정합니다. 추상화 빈은 구체적인 빈 정의를 위해 부모 역할을 하며 인스턴스화되지 않습니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 빈 공장이 빈을 인스턴스화하지 않도록 지시합니다.
depends-on	끝점을 인스턴스화하기 전에 끝점이 인스턴스화되는 데 의존하는 빈 목록을 지정합니다.
createdFromAPI	**Endpoint.publish()** 또는 **Service.getPort()** 와 같은 Apache CXF API를 사용하여 빈을 만들도록 지정합니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 다음이 수행됩니다. ● ID에 **.jaxws-endpoint** 를 추가하여 빈의 내부 이름을 변경합니다. ● 빈을 추상적으로 만듭니다.

표 17.2. "jaxws:server Element를 사용하여 JAX-WS 서비스 공급자 구성 특성" 에 나열된 속성 외에도 여러 xmlns:shortName속성을 사용하여 endpointName 및 serviceName 속성에 사용되는 네임스페이스를 선언해야 할 수 있습니다.

17.1.3.4. 예제

예 17.4. "simple JAX-WS Server 구성" 엔드포인트가 게시되는 주소를 지정하는 JAX-WS 끝점의 구성을 보여줍니다.

예 17.4. simple JAX-WS Server 구성

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:server id="exampleServer"
                serviceBean="org.apache.cxf.example.DemoImpl"
                address="http://localhost:8080/demo" />
</beans>
```

17.1.4. 서비스 공급자에 기능 추가

17.1.4.1. 개요

jaxws:endpoint 및 **jaxws:server** 요소는 서비스 공급자를 인스턴스화하는 데 필요한 기본 구성 정보를 제공합니다. 서비스 공급자에 기능을 추가하거나 고급 구성을 수행하려면 구성에 하위 요소를 추가해야 합니다.

자식 요소를 사용하면 다음을 수행할 수 있습니다.

- 19장. Apache CXF Logging
- 59장. Interceptors를 사용하도록 끝점 구성
- 20장. WS-Addressing 배포
- 21장. 신뢰할 수 있는 메시징 활성화
- 17.1.5절. "JAX-WS 끝점에서 스키마 유효성 검사 활성화"

17.1.4.2. 요소

표 17.3. "JAX-WS 서비스 공급자 구성에 사용되는 요소" **jaxws:endpoint** 에서 지원하는 하위 요소에 대해 설명합니다.

표 17.3. JAX-WS 서비스 공급자 구성에 사용되는 요소

요소	설명
jaxws:handlers	메시지 처리를 위한 JAX-WS Handler 구현 목록을 지정합니다. JAX-WS Handler 구현에 대한 자세한 내용은 43장. 핸들러 작성 을 참조하십시오.
jaxws:inInterceptors	인바운드 요청을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:inFaultInterceptors	인바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.

요소	설명
jaxws:outInterceptors	아웃바운드 응답을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:outFaultInterceptors	아웃바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:binding	끝점에서 사용하는 메시지 바인딩을 구성하는 빈을 지정합니다. 메시지 바인딩은 **org.apache.cxf.binding.BindingFactory** 인터페이스의 구현을 사용하여 구성됩니다.[a]
jaxws:dataBinding [b]	끝점에서 사용하는 데이터 바인딩을 구현하는 클래스를 지정합니다.Specifies the class implementing the data binding used by the endpoint. 이는 포함된 빈 정의를 사용하여 지정합니다.
jaxws:executor	서비스에 사용되는 Java executor를 지정합니다. 이는 포함된 빈 정의를 사용하여 지정합니다.
jaxws:features	Apache CXF의 고급 기능을 구성하는 빈 목록을 지정합니다. 빈 참조 목록 또는 포함된 빈 목록을 제공할 수 있습니다.
jaxws:invoker	서비스에서 사용하는 org.apache.cxf.service. invoker 인터페이스의 구현을 지정합니다.[c]
jaxws:properties	끝점에 전달되는 속성의 Spring 맵을 지정합니다. 이러한 속성은 MTOM 지원 활성화와 같은 기능을 제어하는 데 사용할 수 있습니다.
jaxws:serviceFactory	서비스를 인스턴스화하는 데 사용되는 **JaxWsServiceFactoryBean** 오브젝트를 구성하는 빈을 지정합니다.

[a] SOAP 바인딩은 **soap:soapBinding** 빈을 사용하여 구성됩니다.

[b] **jaxws:endpoint** 요소는 **jaxws:dataBinding** 요소를 지원하지 않습니다.

[c] invoker 구현은 서비스가 호출되는 방법을 제어합니다. 예를 들어 서비스 구현의 새 인스턴스에서 각 요청을 처리했는지 또는 상태가 호출 간에 보존되는지 여부를 제어합니다.

17.1.5. JAX-WS 끝점에서 스키마 유효성 검사 활성화

17.1.5.1. 개요

schema-validation-enabled 속성을 설정하여 **jaxws:endpoint** 요소 또는 **jaxws:server** 요소에서 스키마 유효성 검사를 활성화할 수 있습니다. 스키마 유효성 검사가 활성화되면 클라이언트와 서버 간에 전송된 메시지가 스키마 준수 여부를 확인합니다. 기본적으로 스키마 유효성 검사는 성능에 큰 영향을 미치기 때문에 해제됩니다.**By default, schema validation is turned off, because it has a significant impact on performance.**

17.1.5.2. 예제

JAX-WS 끝점에서 스키마 유효성 검사를 활성화하려면 **jaxws:endpoint** 요소 또는 **jaxws:server** 요소의 **jaxws:properties** 하위 요소에서 **schema-validation-enabled** 속성을 설정합니다. 예를 들어 **jaxws:endpoint** 요소에 스키마 검증을 활성화하려면 다음을 수행합니다.

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
    wsdlLocation="wsdl/hello_world.wsdl"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:endpoint>
```

schema-validation-enabled 속성의 허용되는 값 목록은 **24.3.4.7절. “스키마 유효성 검사 유형 값”**을 참조하십시오.

17.2. 소비자 엔드 포인트 구성

17.2.1. 개요

JAX-WS 소비자 엔드포인트는 **jaxws:client** 요소를 사용하여 구성됩니다. 요소의 특성은 소비자를 생성하는 데 필요한 기본 정보를 제공합니다.

WS-RM과 같은 다른 기능을 사용자에게 **jaxws:client** 요소에 자식을 추가합니다. 하위 요소는 또한 끝점의 로깅 동작을 구성하고 엔드포인트의 구현에 다른 속성을 삽입하는 데 사용됩니다.

17.2.2. 기본 설정 속성

표 17.4. “JAX-WS Consumer를 구성하는 데 사용되는 속성” 에 설명된 속성은 **JAX-WS** 소비자를 구성하는 데 필요한 기본 정보를 제공합니다. 구성하려는 특정 속성에 대한 값만 제공해야 합니다. 대부분의 속성은 합리적인 기본값이 있거나, 끝점의 계약에서 제공하는 정보에 의존합니다.

표 17.4. JAX-WS Consumer를 구성하는 데 사용되는 속성

속성	설명
주소	소비자가 요청할 끝점의 HTTP 주소를 지정합니다. 이 값은 계약에 설정된 값을 재정의합니다.
bindingId	소비자가 사용하는 메시지 바인딩의 ID를 지정합니다. 유효한 바인딩 ID 목록은 23장. *Apache CXF 바인딩 ID* 에서 제공됩니다.
bus	엔드포인트를 관리하는 버스를 구성하는 Spring 빈의 ID를 지정합니다.
endpointName	소비자가 요청하는 서비스에 대한 **wsdl:port** 요소의 **name** 속성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:port** 요소의 네임스페이스입니다.
serviceName	소비자가 요청하는 서비스에 대한 **wsdl:service** 요소의 **name** 속성 값을 지정합니다. ***ns*:*name*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:service** 요소의 네임스페이스입니다.
사용자 이름	간단한 사용자 이름/암호 인증에 사용되는 사용자 이름을 지정합니다.
암호	간단한 사용자 이름/암호 인증에 사용되는 암호를 지정합니다.
serviceClass	서비스 엔드포인트 인터페이스(SEI)의 이름을 지정합니다.
wsdlLocation	끝점의 WSDL 계약의 위치를 지정합니다. WSDL 계약의 위치는 클라이언트가 배포된 폴더를 기준으로 합니다.
name	소비자가 요청하는 서비스에 대한 **wsdl:port** 요소의 stringified QName을 지정합니다. ***{ns}localPart*** 형식을 사용하여 QName으로 지정됩니다. 여기서 *ns* 는 **wsdl:port** 요소의 네임스페이스이고 *localPart* 는 **wsdl:port** 요소의 **name** 속성 값입니다.
abstract	빈이 추상 빈인지 여부를 지정합니다. 추상화 빈은 구체적인 빈 정의를 위해 부모 역할을 하며 인스턴스화되지 않습니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 빈 공장이 빈을 인스턴스화하지 않도록 지시합니다.
depends-on	끝점이 인스턴스화되기 전에 종속되는 빈 목록을 지정합니다.

속성	설명
createdFromAPI	**Service.getPort()** 와 같은 Apache CXF API를 사용하여 빈을 만들도록 지정합니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 다음이 수행됩니다. • **.jaxws-client** 를 해당 ID에 추가하여 빈의 내부 이름을 변경합니다. • 빈을 추상적으로 만듭니다.

*표 17.4. "JAX-WS Consumer를 구성하는 데 사용되는 속성" 에 나열된 속성 외에도 여러 **xmlns:shortName**특성을 사용하여 **endpointName** 및 **serviceName** 속성에 사용되는 네임스페이스를 선언해야 할 수 있습니다.*

17.2.3. 기능 추가

사용자에게 기능을 추가하거나 고급 구성을 수행하려면 구성에 하위 요소를 추가해야 합니다.

자식 요소를 사용하면 다음을 수행할 수 있습니다.

- *19장. Apache CXF Logging*
- *59장. Interceptors를 사용하도록 끝점 구성*
- *20장. WS-Addressing 배포*
- *21장. 신뢰할 수 있는 메시징 활성화*
- *"JAX-WS 소비자에 대한 스키마 유효성 검사 활성화"*

*표 17.5. "Consumer Endpoint 구성을 위한 요소" **JAX-WS** 소비자를 구성하는 데 사용할 수 있는 자식 요소에 대해 설명합니다.*

표 17.5. Consumer Endpoint 구성을 위한 요소

요소	설명
jaxws:binding	끝점에서 사용하는 메시지 바인딩을 구성하는 빈을 지정합니다. 메시지 바인딩은 **org.apache.cxf.binding.BindingFactory** 인터페이스의 구현을 사용하여 구성됩니다.[a]
jaxws:dataBinding	끝점에서 사용하는 데이터 바인딩을 구현하는 클래스를 지정합니다.Specifies the class implementing the data binding used by the endpoint. 포함된 빈 정의를 사용하여 이 값을 지정합니다. JAXB 데이터 바인딩을 구현하는 클래스는 **org.apache.cxf.jaxb.JAXBDataBinding** 입니다.
jaxws:features	Apache CXF의 고급 기능을 구성하는 빈 목록을 지정합니다. 빈 참조 목록 또는 포함된 빈 목록을 제공할 수 있습니다.
jaxws:handlers	메시지 처리를 위한 JAX-WS Handler 구현 목록을 지정합니다. JAX-WS Handler 구현 방법에 대한 자세한 내용은 43장. *핸들러 작성* 을 참조하십시오.
jaxws:inInterceptors	인바운드 응답을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:inFaultInterceptors	인바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:outInterceptors	아웃바운드 요청을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:outFaultInterceptors	아웃바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxws:properties	끝점으로 전달되는 속성의 맵을 지정합니다.
jaxws:conduitSelector	클라이언트가 사용할 org.apache.cxf.endpoint.ConduitSelector 구현을 지정합니다. ConduitSelector 구현은 아웃바운드 요청을 처리하는 데 사용되는 **Conduit** Selector 개체를 선택하는 데 사용되는 기본 프로세스를 재정의합니다.
[a] SOAP 바인딩은 **soap:soapBinding** 빈을 사용하여 구성됩니다.	

17.2.4. 예제

예 17.5. "간단한 소비자 구성" 간단한 소비자 구성을 보여줍니다.

예 17.5. 간단한 소비자 구성

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookClientImpl"
                address="http://localhost:8080/books"/>
  ...
</beans>
```

17.2.5. JAX-WS 소비자에 대한 스키마 유효성 검사 활성화

JAX-WS 소비자에서 스키마 유효성 검사를 활성화하려면 jaxws:client 요소(예:)의 jaxws:properties 하위 요소에서 schema-validation-enabled 속성을 설정합니다.

```
<jaxws:client name="{http://apache.org/hello_world_soap_http}SoapPort"
    createdFromAPI="true">
    <jaxws:properties>
        <entry key="schema-validation-enabled" value="BOTH" />
    </jaxws:properties>
</jaxws:client>
```

schema-validation-enabled 속성의 허용되는 값 목록은 24.3.4.7절. "스키마 유효성 검사 유형 값" 을 참조하십시오.

18장. JAX-RS 엔드포인트 구성

초록

이 장에서는 Blueprint XML과 Spring XML에서 JAX-RS 서버 엔드포인트를 인스턴스화하고 구성하는 방법과 XML에서 JAX-RS 클라이언트 엔드포인트(클라이언트 프록시 빈)를 인스턴스화하고 구성하는 방법도 설명합니다.

18.1. JAX-RS SERVER 엔드 포인트 구성

18.1.1. JAX-RS Server Endpoint 정의

18.1.1.1. 기본 서버 끝점 정의

XML에서 JAX-RS 서버 끝점을 정의하려면 최소한 다음을 지정해야 합니다.

1. XML에서 끝점을 정의하는 데 사용되는 jaxrs:server 요소. jaxrs: 네임스페이스 접두사는 Blueprint 및 Spring의 다양 한 네임스페이스에 각각 매핑됩니다.

2. jaxrs:server 요소의 address 특성을 사용하여 JAX-RS 서비스의 기본 URL입니다. 끝점 배포 방법에 영향을 주는 주소 URL을 지정하는 두 가지 다른 방법이 있습니다.

   - 상대 URL- 예를 들어 /customers. 이 경우 끝점은 기본 HTTP 컨테이너에 배포되며 엔드포인트의 기본 URL은 CXF 서블릿 기본 URL을 지정된 상대 URL과 결합하여 암시적으로 가져옵니다.

     예를 들어, JAX-RS 엔드포인트를 Fuse 컨테이너에 배포하는 경우 지정된 /customers URL이 URL로 확인됩니다. http://Hostname:8181/cxf/customers(컨테이너가 기본 8181 포트를 사용하고 있다고 가정함).

   - 절대 URL인 경우 (예: http://0.0.0.0:8200/cxf/customers) 이 경우 JAX-RS 엔드포인트에 대해 새 HTTP 리스너 포트가 열립니다(아직 열려 있지 않은 경우). 예를 들어 Fuse의 컨텍스트에서 JAX-RS 엔드포인트를 호스트하기 위해 새 Undertow 컨테이너가 암시적으로 생성됩니다. 특수 IP 주소 0.0.0.0 은 와일드카드 역할을 하며 현재 호스트에 할당된 호스트 이름(Multi-homed 호스트 시스템에서 유용할 수 있음)과 일치합니다.

3. JAX-RS 서비스의 구현을 제공하는 하나 이상의 JAX-RS 루트 리소스 클래스입니다. 리소스 클래스를 지정하는 가장 간단한 방법은 jaxrs:serviceBeans 요소 내에 나열하는 것입니다.

18.1.1.2. Blueprint 예

다음 **Blueprint XML** 예제에서는 상대 주소를 지정하는 **JAX-RS** 엔드포인트를 정의하는 방법을 보여줍니다. **/customers** (기본 **HTTP** 컨테이너로 배포 가능)는 **service.CustomerService** 리소스 클래스에 의해 구현됩니다.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    xsi:schemaLocation="
http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
http://cxf.apache.org/blueprint/jaxrs http://cxf.apache.org/schemas/blueprint/jaxrs.xsd
http://cxf.apache.org/blueprint/core http://cxf.apache.org/schemas/blueprint/core.xsd
">

    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>

     <jaxrs:server id="customerService" address="/customers">
        <jaxrs:serviceBeans>
            <ref component-id="serviceBean" />
        </jaxrs:serviceBeans>
     </jaxrs:server>

     <bean id="serviceBean" class="service.CustomerService"/>
</blueprint>
```

18.1.1.3. Blueprint XML 네임스페이스

Blueprint에서 **JAX-RS** 엔드포인트를 정의하려면 일반적으로 다음과 같은 **XML** 네임스페이스가 필요합니다.

접두사	네임스페이스
(기본값)	http://www.osgi.org/xmlns/blueprint/v1.0.0
cxf	http://cxf.apache.org/blueprint/core
jaxrs	http://cxf.apache.org/blueprint/jaxrs

18.1.1.4. Spring 예

다음 **Spring XML** 예제에서는 상대 주소 **/customers** (기본 **HTTP** 컨테이너에 배포)를 지정하고

service.CustomerService 리소스 클래스에 의해 구현되는 JAX-RS 엔드포인트를 정의하는 방법을 보여줍니다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxrs="http://cxf.apache.org/jaxrs"
      xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

    <jaxrs:server id="customerService" address="/customers">
      <jaxrs:serviceBeans>
        <ref bean="serviceBean"/>
      </jaxrs:serviceBeans>
    </jaxrs:server>

    <bean id="serviceBean" class="service.CustomerService"/>
</beans>
```

18.1.1.5. XML 네임스페이스 스프링

Spring에서 JAX-RS 엔드포인트를 정의하려면 일반적으로 다음과 같은 XML 네임스페이스가 필요합니다.

접두사	네임스페이스
(기본값)	http://www.springframework.org/schema/beans
cxf	http://cxf.apache.org/core
jaxrs	http://cxf.apache.org/jaxrs

18.1.1.6. Spring XML 자동 검색

(spring only) JAX-RS 루트 리소스 클래스를 명시적으로 지정하는 대신 Spring XML을 사용하면 자동 검색을 구성할 수 있으므로 특정 Java 패키지가 리소스 클래스(@Path에서 주석이 달린 클래스) 및 모든 검색된 리소스 클래스가 자동으로 엔드포인트에 연결됩니다. 이 경우 jaxrs:server 요소에서 address 속성과 basePackages 속성만 지정해야 합니다.

예를 들어 a.b.c Java 패키지에서 모든 JAX-RS 리소스 클래스를 사용하는 JAX-RS 엔드포인트를 정의하려면 다음과 같이 Spring XML에서 엔드포인트를 정의할 수 있습니다.

```
<jaxrs:server address="/customers" basePackages="a.b.c"/>
```

자동 검색 메커니즘은 지정된 Java 패키지에서 찾은 모든 JAX-RS 공급자 클래스도 엔드포인트에 검색하고 설치합니다.

18.1.1.7. Spring XML의 라이프사이클 관리

(spring only) Spring XML을 사용하면 빈 요소에서 scope 속성을 설정하여 빈의 라이프사이클을 제어할 수 있습니다. Spring에서 다음 범위 값을 지원합니다.

singleton

(기본값) 모든 곳에서 사용되며 Spring 컨테이너의 전체 수명 동안 지속되는 단일 빈 인스턴스를 생성합니다.

prototype

빈이 다른 빈에 삽입될 때마다 또는 빈 레지스트리에서 getBean() 을 호출하여 빈을 가져올 때 새 빈 인스턴스를 생성합니다.

요청

(웹 인식 컨테이너에서만 사용 가능) 빈에서 호출되는 모든 요청에 대해 새 빈 인스턴스를 생성합니다.

session

(웹 인식 컨테이너에서만 사용 가능) 단일 HTTP 세션의 수명 동안 새 빈을 생성합니다.

globalSession

(웹 인식 컨테이너에서만 사용 가능) 포 틀릿 간에 공유되는 단일 HTTP 세션의 수명 동안 새 빈을 생성합니다.

Spring 범위에 대한 자세한 내용은 빈 범위에 대한 Spring 프레임워크 설명서 를 참조하십시오.

jaxrs:serviceBeans 요소를 통해 JAX-RS 리소스 빈을 지정하는 경우 Spring 범위가 제 대로 작동하지 않습니다. 이 경우 리소스 빈에 scope 속성을 지정하면 scope 속성이 효과적으로 무시됩니다.

빈 범위가 JAX-RS 서버 끝점 내에서 제대로 작동하도록 하려면 서비스 팩토리에서 제공하는 간접 수준이 필요합니다. 빈 범위를 구성하는 가장 간단한 방법은 다음과 같이 jaxrs:server 요소의 bean 특성을 사용하여 리소스 빈을 지정하는 것입니다.

```
<beans ... >
```

```
  <jaxrs:server id="customerService" address="/service1"
    beanNames="customerBean1 customerBean2"/>

  <bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
scope="prototype"/>
  <bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
scope="prototype"/>
</beans>
```

*위 예제에서는 두 개의 리소스 빈, **customerBean1** 및 **customerBean2** 를 구성합니다. **beanNames** 특성은 리소스 빈 **ID**의 공백으로 구분된 목록으로 지정됩니다.*

*궁극적인 수준의 유연성을 위해 **jaxrs:serviceFactories** 요소를 사용하여 **JAX-RS** 서버 엔드포인트를 구성할 때 서비스 팩토리 개체를 명시적으로 정의할 수 있습니다. 이러한 보다 자세한 접근 방식은 기본 서비스 팩토리 구현을 사용자 지정 구현으로 교체할 수 있다는 이점이 있으므로 빈 라이프사이클을 궁극적으로 제어할 수 있습니다. 다음 예제에서는 다음 방법을 사용하여 두 리소스 빈인 **customerBean1** 및 **customerBean2** 를 구성하는 방법을 보여줍니다.*

```
<beans ... >
  <jaxrs:server id="customerService" address="/service1">
    <jaxrs:serviceFactories>
      <ref bean="sfactory1" />
      <ref bean="sfactory2" />
    </jaxrs:serviceFactories>
  </jaxrs:server>

  <bean id="sfactory1" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
    <property name="beanId" value="customerBean1"/>
  </bean>
  <bean id="sfactory2" class="org.apache.cxf.jaxrs.spring.SpringResourceFactory">
    <property name="beanId" value="customerBean2"/>
  </bean>

  <bean id="customerBean1" class="demo.jaxrs.server.CustomerRootResource1"
scope="prototype"/>
  <bean id="customerBean2" class="demo.jaxrs.server.CustomerRootResource2"
scope="prototype"/>
</beans>
```

참고

*단일턴 라이프사이클이 아닌 경우 **org.apache.cxf.service. invoker bean**을 구현하고 등록하는 것이 좋습니다(**jaxrs:server/jaxrs:invoker** 요소에서 참조하여 인스턴스를 등록할 수 있음).*

18.1.1.8. WADL 문서 연결

***jaxrs:server** 요소의 **docLocation** 속성을 사용하여 **WADL** 문서를 **JAX-RS** 서버 끝점과 선택적으로*

연결할 수 있습니다. 예를 들면 다음과 같습니다.

```
<jaxrs:server address="/rest" docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
</jaxrs:server>
```

18.1.1.9. 스키마 검증

JAX-B 형식의 메시지 콘텐츠를 설명하는 외부 **XML** 스키마가 있는 경우 **jaxrs:schemaLocations** 요소를 통해 이러한 외부 스키마를 **JAX-RS** 서버 끝점과 연결할 수 있습니다.

예를 들어 서버 끝점과 **WADL** 문서를 연결하고 수신되는 메시지에 대해 스키마 유효성 검사를 활성화하려면 다음과 같이 연결된 **XML** 스키마 파일을 지정할 수 있습니다.

```
<jaxrs:server address="/rest"
              docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/a.xsd</jaxrs:schemaLocation>
    <jaxrs:schemaLocation>classpath:/schemas/b.xsd</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

또는 지정된 디렉토리에 모든 스키마 파일 ***.xsd** 를 포함하려는 경우 다음과 같이 디렉터리 이름만 지정할 수 있습니다.

```
<jaxrs:server address="/rest"
              docLocation="wadl/bookStore.wadl">
  <jaxrs:serviceBeans>
    <bean class="org.bar.generated.BookStore"/>
  </jaxrs:serviceBeans>
  <jaxrs:schemaLocations>
    <jaxrs:schemaLocation>classpath:/schemas/</jaxrs:schemaLocation>
  </jaxrs:schemaLocations>
</jaxrs:server>
```

이러한 방식으로 스키마를 지정하는 것은 일반적으로 **JAX-B** 스키마에 액세스해야 하는 모든 종류의 기능에 유용합니다.

18.1.1.10. 데이터 바인딩 지정

jaxrs:dataBinding 요소를 사용하여 메시지 본문을 요청 및 응답 메시지에서 인코딩하는 데이터 바인딩을 지정할 수 있습니다. 예를 들어 JAX-B 데이터 바인딩을 지정하려면 다음과 같이 JAX-RS 끝점을 구성할 수 있습니다.

```
<jaxrs:server id="jaxbbook" address="/jaxb">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.jaxb.JAXBDataBinding"/>
  </jaxrs:dataBinding>
</jaxrs:server>>
```

Aegis 데이터 바인딩을 지정하려면 다음과 같이 JAX-RS 끝점을 구성할 수 있습니다.

```
<jaxrs:server id="aegisbook" address="/aegis">
  <jaxrs:serviceBeans>
    <ref bean="serviceBean" />
  </jaxrs:serviceBeans>
  <jaxrs:dataBinding>
    <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding">
      <property name="aegisContext">
        <bean class="org.apache.cxf.aegis.AegisContext">
          <property name="writeXsiTypes" value="true"/>
        </bean>
      </property>
    </bean>
  </jaxrs:dataBinding>
</jaxrs:server>
```

18.1.1.11. JMS 전송 사용

HTTP 대신 JMS 메시징 라이브러리를 전송 프로토콜로 사용하도록 JAX-RS를 구성할 수 있습니다. JMS 자체는 전송 프로토콜이 아니 므로 실제 메시징 프로토콜은 사용자가 구성하는 특정 JMS 구현에 따라 다릅니다.

예를 들어 다음 Spring XML 예제에서는 JMS 전송 프로토콜을 사용하도록 JAX-RS 서버 엔드포인트를 구성하는 방법을 보여줍니다.

```
<?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:jms="http://cxf.apache.org/transports/jms"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
http://cxf.apache.org/transports/jms http://cxf.apache.org/schemas/configuration/jms.xsd
```

```
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>
    <bean id="ConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL"
value="tcp://localhost:${testutil.ports.EmbeddedJMSBrokerLauncher}" />
    </bean>

    <jaxrs:server xmlns:s="http://books.com"
     serviceName="s:BookService"
     transportId= "http://cxf.apache.org/transports/jms"
     address="jms:queue:test.jmstransport.text?replyToName=test.jmstransport.response">
        <jaxrs:serviceBeans>
            <bean class="org.apache.cxf.systest.jaxrs.JMSBookStore"/>
        </jaxrs:serviceBeans>
    </jaxrs:server>

</beans>
```

이전 예제에 대한 다음 사항에 유의하십시오.

- ***JMS** 구현- **JMS** 구현은 **Apache ActiveMQ** 연결 팩토리 개체를 인스턴스화하는 **ConnectionFactory** 빈에서 제공합니다. 연결 팩토리를 인스턴스화한 후에는 자동으로 기본 **JMS** 구현 계층으로 설치됩니다.*

- ***JMS** 구성 또는 대상 개체-**Apache CXF**는 **JMS** 대상 개체(**JMS** 소비자를 나타내는) 또는 **JMS** 대상 오브젝트를 암시적으로 인스턴스화합니다. 이 오브젝트는 **setttings xmlns:s="http://books.com"** 속성(네임 스페이스 접두사 정의) 및 **serviceName="s:BookService" (QName** 정의)를 통해 정의된 **QName**으로 고유하게 식별되어야 합니다.*

- *전송 **ID**-에서 **JMS** 전송을 선택하려면 **transportId** 특성을 **http://cxf.apache.org/transports/jms** 로 설정해야 합니다.*

- ***JMS address-the jaxrs:server/@address** 속성은 표준화된 구문을 사용하여 보낼 **JMS** 대기열 또는 **JMS** 주제를 지정합니다. 이 구문에 대한 자세한 내용은 **https://tools.ietf.org/id/draft-merrick-jms-uri-06.txt** 에서 참조하십시오.*

18.1.1.12. 확장 매핑 및 언어 매핑

***JAX-RS** 서버 끝점은 파일 접미사(**URL**에서 표시됨)를 **MIME** 콘텐츠 유형 헤더에 자동으로 매핑하도록 구성하고 언어 접미사를 언어 유형 헤더에 매핑할 수 있습니다. 예를 들어 다음 형식의 **HTTP** 요청을 고*

려하십시오.**For example, consider a HTTP request of the following form:**

```
GET /resource.xml
```

다음과 같이 **.xml** 접미사를 자동으로 매핑하도록 **JAX-RS** 서버 끝점을 구성할 수 있습니다.

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:extensionMappings>
    <entry key="json" value="application/json"/>
    <entry key="xml" value="application/xml"/>
  </jaxrs:extensionMappings>
</jaxrs:server>
```

이전 서버 끝점에서 **HTTP** 요청을 수신하면 **type, application/xml** 의 새 콘텐츠 유형 헤더를 자동으로 생성하고 리소스 **URL**에서 **.xml** 접미사를 제거합니다.

언어 매핑의 경우 다음 형식의 **HTTP** 요청을 고려하십시오.**For the language mapping, consider a HTTP request of the following form:**

```
GET /resource.en
```

다음과 같이 **JAX-RS** 서버 끝점을 구성하여 **.en** 접미사가 자동으로 매핑할 수 있습니다.

```
<jaxrs:server id="customerService" address="/">
  <jaxrs:serviceBeans>
    <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
  </jaxrs:serviceBeans>
  <jaxrs:languageMappings>
     <entry key="en" value="en-gb"/>
  </jaxrs:languageMappings>
</jaxrs:server>
```

이전 서버 끝점에서 **HTTP** 요청을 수신하면 값이 **en-gb** 인 새 허용 언어 헤더를 자동으로 생성하고 리소스 **URL**에서 **.en** 접미사를 제거합니다.

18.1.2. jaxrs:server 속성

18.1.2.1. 속성

표 18.1. "JAX-RS Server Endpoint Attributes" jaxrs:server 요소에서 사용 가능한 특성을 설명합니다.

표 18.1. JAX-RS Server Endpoint Attributes

속성	설명
id	다른 구성 요소에서 끝점을 참조하는 데 사용할 수 있는 고유 식별자를 지정합니다.Specifies a unique identifier that other configuration elements can use to refer to the endpoint.
주소	HTTP 끝점의 주소를 지정합니다. 이 값은 서비스 계약에 지정된 값을 재정의합니다.
basePackages	**(spring only)** JAX-RS 루트 리소스 클래스 및/또는 JAX-RS 공급자 클래스를 검색하도록 검색되는 Java 패키지 목록을 쉼표로 구분하여 지정하여 자동 검색을 활성화합니다.
beanNames	JAX-RS 루트 리소스 빈의 공백으로 구분된 빈 ID 목록을 지정합니다. Spring XML의 컨텍스트에서 루트 리소스 빈 요소에 **scope** 속성을 설정하여 루트 리소스**빈의 라이프사이클을 정의할 수 있습니다**.
bindingId	서비스에서 사용하는 메시지 바인딩의 ID를 지정합니다. 유효한 바인딩 ID 목록은 23장. *Apache CXF 바인딩 ID* 에서 제공됩니다.
bus	서비스 엔드포인트를 관리하는 데 사용되는 버스를 구성하는 Spring 빈의 ID를 지정합니다. 이 기능은 공통 기능 세트를 사용하도록 여러 끝점을 구성할 때 유용합니다.
docLocation	외부 WADL 문서의 위치를 지정합니다.
modelRef	모델 스키마를 classpath 리소스로 지정합니다(예: **classpath:/path/to/model.xml**). JAX-RS 모델 스키마를 정의하는 방법에 대한 자세한 내용은 18.3절. "모델 스키마를 사용하여 REST 서비스 정의" 을 참조하십시오.
publish	서비스가 자동으로 게시되어야 하는지 여부를 지정합니다. **false** 로 설정하면 개발자가 엔드포인트를 명시적으로 게시해야 합니다.
publishedEndpointUrl	자동 생성된 WADL 인터페이스의 **wadl:resources/@base** 속성에 삽입되는 URL 기본 주소를 지정합니다.

속성	설명
serviceAnnotation	(spring only) Spring 에서 자동 검색에 대한 서비스 주석 클래스 이름을 지정합니다. 이 옵션을 **basePackages** 속성과 함께 사용하면 이 주석 유형에 따라 주석이 달린 클래스**만** 포함하도록 자동 검색 클래스 컬렉션을 제한합니다. 이 맞습니까?
serviceClass	JAX-RS 서비스를 구현하는 JAX-RS 루트 리소스 클래스의 이름을 지정합니다. 이 경우 클래스는 Blueprint 또는 Spring이 **아닌** Apache CXF에 의해 인스턴스화됩니다. Blueprint 또는 Spring에서 클래스를 인스턴스화하려면 대신 **jaxrs:serviceBeans** 하위 요소를 사용하십시오.
serviceName	JMS 전송이 사용되는 특수한 경우 JAX-RS 엔드포인트에 대한 서비스 QName(***ns:이름***형식 사용)을 지정합니다. 자세한 내용은 "JMS 전송 사용"의 내용을 참조하십시오.
staticSubresourceResolution	**true** 인 경우 정적 하위 리소스의 동적 확인을 비활성화합니다. 기본값은 **false** 입니다.
transportId	비표준 전송 계층(HTTP 대신)을 선택하기 위한 것입니다. 특히 이 속성을 **http://cxf.apache.org/transports/jms** 로 설정하여 JMS 전송을 선택할 수 있습니다. 자세한 내용은 "JMS 전송 사용"의 내용을 참조하십시오.
abstract	(spring only) 빈이 추상 빈인지 여부를 지정합니다. 추상화 빈은 구체적인 빈 정의를 위해 부모 역할을 하며 인스턴스화되지 않습니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 빈 공장이 빈을 인스턴스화하지 않도록 지시합니다.
depends-on	(spring only) 끝점을 인스턴스화하기 전에 끝점을 인스턴스화하는 데 의존하는 빈 목록을 지정합니다.

18.1.3. jaxrs:server 하위 요소

18.1.3.1. 하위 요소

*표 18.2. "JAX-RS Server Endpoint Child Elements" **jaxrs:server** 요소의 하위 요소에 대해 설명합니다.*

표 18.2. JAX-RS Server Endpoint Child Elements

요소	설명
jaxrs:executor	서비스에 사용되는 Java **Executor** (스레드 풀 구현)를 지정합니다. 이는 포함된 빈 정의를 사용하여 지정합니다.
jaxrs:features	Apache CXF의 고급 기능을 구성하는 빈 목록을 지정합니다. 빈 참조 목록 또는 포함된 빈 목록을 제공할 수 있습니다.
jaxrs:binding	**사용되지 않음.**
jaxrs:dataBinding	끝점에서 사용하는 데이터 바인딩을 구현하는 클래스를 지정합니다.Specifies the class implementing the data binding used by the endpoint. 이는 포함된 빈 정의를 사용하여 지정합니다. 자세한 내용은 "데이터 바인딩 지정" 에서 참조하십시오.
jaxrs:inInterceptors	인바운드 요청을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:inFaultInterceptors	인바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:outInterceptors	아웃바운드 응답을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:outFaultInterceptors	아웃바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:invoker	서비스에서 사용하는 org.apache.cxf.service. invoker 인터페이스의 구현을 지정합니다.[a]
jaxrs:serviceFactories	이 끝점과 연결된 JAX-RS 루트 리소스의 라이프사이클에 대한 최대 제어 수준을 제공합니다. 이 요소의 하위 항목(**org.apache.cxf.jaxrs.lifecycle.ResourceProvider** 유형의 인스턴스)은 JAX-RS 루트 리소스 인스턴스를 생성하는 데 사용됩니다.
jaxrs:properties	끝점에 전달되는 속성의 Spring 맵을 지정합니다. 이러한 속성은 MTOM 지원 활성화와 같은 기능을 제어하는 데 사용할 수 있습니다.

요소	설명
jaxrs:serviceBeans	이 요소의 하위 항목은 **빈** 요소의 인스턴스 또는 (**ref** 요소) JAX-RS 루트 리소스에 대한 참조입니다. 이 경우 **빈** 요소에 있는 경우 **scope** 속성 (Spring만 해당) 은 무시됩니다.
jaxrs:modelBeans	리소스 모델의 기본 요소(**jaxrs:resource** 요소에 해당)인 하나 이상의 **org.apache.cxf.jaxrs.model.UserResource** bean에 대한 참조 목록으로 구성됩니다. 자세한 내용은 18.3절. "모델 스키마를 사용하여 REST 서비스 정의"의 내용을 참조하십시오.
jaxrs:model	이 엔드포인트에서 직접 리소스 모델을 정의합니다(즉, 이 **jaxrs:model** 요소는 하나 이상의 **jaxrs:resource** 요소를 포함할 수 있음). 자세한 내용은 18.3절. "모델 스키마를 사용하여 REST 서비스 정의"의 내용을 참조하십시오.
jaxrs:providers	이 끝점에 하나 이상의 사용자 지정 JAX-RS 공급자를 등록할 수 있습니다. 이 요소의 자식은 **빈** 요소의 인스턴스 또는 (**ref** 요소) JAX-RS 공급자에 대한 참조입니다.
jaxrs:extensionMappings	REST 호출의 URL이 파일 확장으로 종료되면 이 요소를 사용하여 특정 콘텐츠 유형과 자동으로 연결할 수 있습니다. 예를 들어 **.xml** 파일 확장자는 **application/xml** 콘텐츠 유형과 연관될 수 있습니다. 자세한 내용은 "확장 매핑 및 언어 매핑"의 내용을 참조하십시오.
jaxrs:languageMappings	REST 호출의 URL이 언어 접미사로 끝나는 경우 이 요소를 사용하여 특정 언어에 매핑할 수 있습니다. 예를 들어 **.en** 언어 접미사는 **en-GB** 언어와 연결할 수 있습니다. 자세한 내용은 "확장 매핑 및 언어 매핑"의 내용을 참조하십시오.
jaxrs:schemaLocations	XML 메시지 콘텐츠 유효성 검사에 사용되는 하나 이상의 XML 스키마를 지정합니다.Specifies one or more XML schemas used for validating XML message content. 이 요소에는 하나 이상의 **jaxrs:schemaLocation** 요소가 포함될 수 있으며, 각 요소는 XML 스키마 파일의 위치를 지정합니다(일반적으로 **classpath** URL). 자세한 내용은 "스키마 검증"의 내용을 참조하십시오.
jaxrs:resourceComparator	들어오는 URL 경로를 특정 리소스 클래스 또는 메서드에 일치시키는 데 사용되는 알고리즘을 구현하는 사용자 지정 리소스 비교기를 등록할 수 있습니다.

요소	설명
jaxrs:resourceClasses	클래스 이름에서 여러 리소스를 만들려면 **jaxrs:server/@serviceClass** 속성 대신 사용할 수 있습니다. **jaxrs:resourceClasses** 의 하위 항목은 리소스 클래스 이름으로 설정된 **name** 속성이 있는 클래스 요소여야 합니다. 이 경우 클래스는 Blueprint 또는 Spring이 아닌 Apache CXF에 의해 인스턴스화됩니다.
[a] invoker 구현은 서비스가 호출되는 방법을 제어합니다. 예를 들어 서비스 구현의 새 인스턴스에서 각 요청을 처리했는지 또는 상태가 호출 간에 보존되는지 여부를 제어합니다.	

18.2. JAX-RS 클라이언트 엔드 포인트 구성

18.2.1. JAX-RS Client Endpoint 정의

18.2.1.1. 클라이언트 프록시 삽입

***XML** 언어(**Blueprint XML** 또는 **Spring XML**)에서 클라이언트 프록시 빈을 인스턴스화하는 기본 방법은 클라이언트 프록시를 사용하여 **REST** 서비스를 호출할 수 있는 다른 빈에 삽입하는 것입니다. **XML**에서 클라이언트 프록시 빈을 생성하려면 **jaxrs:client** 요소를 사용합니다.*

18.2.1.2. 네임스페이스

***JAX-RS** 클라이언트 끝점은 서버 끝점의 다른 **XML** 네임스페이스를 사용하여 정의합니다. 다음 표에서는 어떤 **XML** 언어에 사용할 네임스페이스를 보여줍니다.*

XML 언어	클라이언트 끝점의 네임스페이스
Blueprint	**http://cxf.apache.org/blueprint/jaxrs-client**
Spring	**http://cxf.apache.org/jaxrs-client**

18.2.1.3. 기본 클라이언트 끝점 정의

*다음 예제에서는 **Blueprint XML** 또는 **Spring XML**에서 클라이언트 프록시 빈을 생성하는 방법을 보여줍니다.*

```
<jaxrs:client id="restClient"
    address="http://localhost:8080/test/services/rest"
    serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws"/>
```

-

기본 클라이언트 끝점을 정의하려면 다음 속성을 설정해야 하는 위치입니다.

id

클라이언트 프록시의 빈 **ID**를 사용하여 **XML** 구성의 다른 빈에 클라이언트 프록시를 삽입할 수 있습니다.

주소

address 속성은 **REST** 호출의 기본 **URL**을 지정합니다.

serviceClass

serviceClass 속성은 루트 리소스 클래스를 지정하여 **REST** 서비스에 대한 설명을 제공합니다(**@Path**에서 주석 처리됨). 실제로 이 클래스는 서버 클래스이지만 클라이언트에서 직접 사용하지는 않습니다. 지정된 클래스는 클라이언트 프록시를 동적으로 구성하는 데 사용되는 해당 메타데이터(**Java** 리플렉션 및 **JAX-RS** 주석을 통해)에만 사용됩니다.

18.2.1.4. 헤더 지정

다음과 같이 **jaxrs:headers** 하위 요소를 사용하여 클라이언트 프록시의 호출에 **HTTP** 헤더를 추가할 수 있습니다.

```
<jaxrs:client id="restClient"
    address="http://localhost:8080/test/services/rest"
    serviceClass="org.apache.cxf.systest.jaxrs.BookStoreJaxrsJaxws"
    inheritHeaders="true">
    <jaxrs:headers>
       <entry key="Accept" value="text/xml"/>
    </jaxrs:headers>
</jaxrs:client>
```

18.2.2. jaxrs:client 속성

18.2.2.1. 속성

표 18.3. “JAX-RS 클라이언트 끝점 속성” **jaxrs:client** 요소에서 사용 가능한 특성을 설명합니다.

표 **18.3. JAX-RS** 클라이언트 끝점 속성

속성	설명

속성	설명
주소	소비자가 요청할 끝점의 HTTP 주소를 지정합니다. 이 값은 계약에 설정된 값을 재정의합니다.
bindingId	소비자가 사용하는 메시지 바인딩의 ID를 지정합니다. 유효한 바인딩 ID 목록은 23장. *Apache CXF 바인딩 ID* 에서 제공됩니다.
bus	엔드포인트를 관리하는 버스를 구성하는 Spring 빈의 ID를 지정합니다.
inheritHeaders	이 프록시에서 하위 리소스 프록시가 생성되는 경우 이 프록시에 설정된 헤더가 상속되는지 여부를 지정합니다. 기본값은 **false** 입니다.
사용자 이름	간단한 사용자 이름/암호 인증에 사용되는 사용자 이름을 지정합니다.
암호	간단한 사용자 이름/암호 인증에 사용되는 암호를 지정합니다.
modelRef	모델 스키마를 classpath 리소스로 지정합니다(예: **classpath:/path/to/model.xml**). JAX-RS 모델 스키마를 정의하는 방법에 대한 자세한 내용은 18.3절. "모델 스키마를 사용하여 REST 서비스 정의" 을 참조하십시오.
serviceClass	서비스 인터페이스 또는 리소스 클래스(**@PATH**로 주석이 추가됨)의 이름을 지정하고 JAX-RS 서버 구현에서 다시 사용합니다. 이 경우 지정된 클래스가 직접 호출 **되지 않습니다** (실제 서버 클래스). 지정된 클래스는 클라이언트 프록시를 동적으로 구성하는 데 사용되는 해당 메타데이터(Java 리플렉션 및 JAX-RS 주석을 통해)에만 사용됩니다.
serviceName	JMS 전송이 사용되는 특수한 경우 JAX-RS 엔드포인트에 대한 서비스 QName(***ns:이름***형식 사용)을 지정합니다. 자세한 내용은 "JMS 전송 사용"의 내용을 참조하십시오.
threadSafe	클라이언트 프록시가 스레드로부터 안전한지 여부를 지정합니다.Specifies whether the client proxy is thread-safe. 기본값은 **false** 입니다.
transportId	비표준 전송 계층(HTTP 대신)을 선택하기 위한 것입니다. 특히 이 속성을 **http://cxf.apache.org/transports/jms** 로 설정하여 JMS 전송을 선택할 수 있습니다. 자세한 내용은 "JMS 전송 사용"의 내용을 참조하십시오.

속성	설명
abstract	**(spring only)** 빈이 추상 빈인지 여부를 지정합니다. 추상화 빈은 구체적인 빈 정의를 위해 부모 역할을 하며 인스턴스화되지 않습니다. 기본값은 **false**입니다. 이 값을 **true** 로 설정하면 빈 공장이 빈을 인스턴스화하지 않도록 지시합니다.
depends-on	**(spring only)** 끝점이 인스턴스화되기 전에 의존하는 빈 목록을 지정합니다.

18.2.3. jaxrs: 클라이언트 하위 요소

18.2.3.1. 하위 요소

*표 18.4. "JAX-RS 클라이언트 끝점 하위 요소" **jaxrs:client 요소의 하위 요소에 대해 설명합니다.***

표 18.4. JAX-RS 클라이언트 끝점 하위 요소

요소	설명
jaxrs:executor	
jaxrs:features	Apache CXF의 고급 기능을 구성하는 빈 목록을 지정합니다. 빈 참조 목록 또는 포함된 빈 목록을 제공할 수 있습니다.
jaxrs:binding	**사용되지 않음.**
jaxrs:dataBinding	끝점에서 사용하는 데이터 바인딩을 구현하는 클래스를 지정합니다.Specifies the class implementing the data binding used by the endpoint. 이는 포함된 빈 정의를 사용하여 지정합니다. 자세한 내용은 "데이터 바인딩 지정" 에서 참조하십시오.
jaxrs:inInterceptors	인바운드 응답을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:inFaultInterceptors	인바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:outInterceptors	아웃바운드 요청을 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.

요소	설명
jaxrs:outFaultInterceptors	아웃바운드 오류 메시지를 처리하는 인터셉터 목록을 지정합니다. 자세한 내용은 VII 부. Apache CXF 인터셉터 개발 에서 참조하십시오.
jaxrs:properties	끝점으로 전달되는 속성의 맵을 지정합니다.
jaxrs:providers	이 끝점에 하나 이상의 사용자 지정 JAX-RS 공급자를 등록할 수 있습니다. 이 요소의 자식은**빈** 요소의 인스턴스 또는 (**ref** 요소) JAX-RS 공급자에 대한 참조입니다.
jaxrs:modelBeans	리소스 모델의 기본 요소(**jaxrs:resource** 요소에 해당)인 하나 이상의 **org.apache.cxf.jaxrs.model.UserResource** bean에 대한 참조 목록으로 구성됩니다. 자세한 내용은 18.3절. "모델 스키마를 사용하여 REST 서비스 정의"의 내용을 참조하십시오.
jaxrs:model	이 엔드포인트에서 직접 리소스 모델을 정의합니다(즉, **jaxrs:resource** 요소가 하나 이상 포함된 **jaxrs:model** 요소). 자세한 내용은18.3절. "모델 스키마를 사용하여 REST 서비스 정의"의 내용을 참조하십시오.
jaxrs:headers	발신 메시지에 헤더를 설정하는 데 사용됩니다. 자세한 내용은 "헤더 지정"의 내용을 참조하십시오.
jaxrs:schemaLocations	XML 메시지 콘텐츠 유효성 검사에 사용되는 하나 이상의 XML 스키마를 지정합니다.Specifies one or more XML schemas used for validating XML message content. 이 요소에는 하나 이상의 **jaxrs:schemaLocation** 요소가 포함될 수 있으며, 각 요소는 XML 스키마 파일의 위치를 지정합니다(일반적으로 **classpath** URL). 자세한 내용은"스키마 검증"의 내용을 참조하십시오.

18.3. 모델 스키마를 사용하여 REST 서비스 정의

18.3.1. 주석이 없는 RESTful 서비스

***JAX-RS** 모델 스키마를 사용하면 **Java** 클래스에 주석을 달지 않고 **RESTful** 서비스를 정의할 수 있습니다. 즉, **@Path , @Path Param,@Consumes, @Consumes ,@ Consumes , @tentSourcePolicy** 등과 같은 주석을 **Java** 클래스(또는 인터페이스)에 직접 추가하는 대신, 모델 스키마를 사용하여 모든 관련 **REST** 메타데이터를 별도의 **XML** 파일에 제공할 수 있습니다. 예를 들어 서비스를 구현하는 **Java** 소스를 수정할 수 없는 경우 유용합니다.*

18.3.2. 모델 스키마 예

예 18.1. "샘플 JAX-RS 모델 스키마" BookStoreNoAnnotations 루트 리소스 클래스에 대한 서비스 메타데이터를 정의하는 모델 스키마의 예를 보여줍니다.

예 18.1. 샘플 JAX-RS 모델 스키마

```
<model xmlns="http://cxf.apache.org/jaxrs">
  <resource name="org.apache.cxf.systest.jaxrs.BookStoreNoAnnotations" path="bookstore"
    produces="application/json" consumes="application/json">
    <operation name="getBook" verb="GET" path="/books/{id}" produces="application/xml">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="getBookChapter" path="/books/{id}/chapter">
      <param name="id" type="PATH"/>
    </operation>
    <operation name="updateBook" verb="PUT">
      <param name="book" type="REQUEST_BODY"/>
    </operation>
  </resource>
  <resource name="org.apache.cxf.systest.jaxrs.ChapterNoAnnotations">
    <operation name="getItself" verb="GET"/>
    <operation name="updateChapter" verb="PUT" consumes="application/xml">
       <param name="content" type="REQUEST_BODY"/>
    </operation>
  </resource>
</model>
```

18.3.3. 네임스페이스

모델 스키마를 정의하는 데 사용하는 XML 네임스페이스는 Blueprint XML에서 해당 JAX-RS 엔드포인트를 정의하는지 또는 Spring XML에서 정의하는지에 따라 달라집니다. 다음 표에서는 어떤 XML 언어에 사용할 네임스페이스를 보여줍니다.

XML 언어	네임스페이스
Blueprint	http://cxf.apache.org/blueprint/jaxrs
Spring	http://cxf.apache.org/jaxrs

18.3.4. 끝점에 모델 스키마를 연결하는 방법

모델 스키마를 끝점에 정의하고 연결하려면 다음 단계를 수행합니다.

1.

선택한 주입 플랫폼(**Blueprint XML** 또는 **Spring XML**)에 적합한 **XML** 네임스페이스를 사용하여 모델 스키마를 정의합니다.

2. 모델 스키마 파일을 프로젝트의 리소스에 추가하여 스키마 파일을 최종 패키지(**JAR, WAR** 또는 **OSGi** 번들 파일)의 **classpath**에서 사용할 수 있도록 합니다.

참고

또는 엔드포인트의 **jaxrs:model** 하위 요소를 사용하여 모델 스키마를 **JAX-RS** 엔드포인트에 직접 포함할 수도 있습니다.

3. 엔드포인트의 **model** 스키마를 **classpath**의 모델 스키마 위치로 설정하여 모델 스키마를 사용하도록 엔드포인트를 구성합니다(**classpath URL** 사용).

4. 필요한 경우 **jaxrs:serviceBeans** 요소를 사용하여 루트 리소스를 명시적으로 인스턴스화합니다. 모델 스키마에서 루트 리소스 클래스를 직접 참조하는 경우(기본 인터페이스 참조 대신) 이 단계를 건너뛸 수 있습니다.

18.3.5. 클래스를 참조하는 모델 스키마 구성

모델 스키마가 루트 리소스 클래스에 직접 적용되는 경우 모델 스키마가 루트 리소스 빈을 자동으로 인스턴스화하므로 **jaxrs:serviceBeans** 요소를 사용하여 루트 리소스 빈을 정의할 필요가 없습니다.

예를 들어 **customer-resources.xml** 이 메타데이터를 고객 리소스 클래스와 연결하는 모델 스키마로 설정된 경우 다음과 같이 **customerService** 서비스 끝점을 인스턴스화할 수 있습니다.

```
<jaxrs:server id="customerService"
        address="/customers"
        modelRef="classpath:/org/example/schemas/customer-resources.xml" />
```

18.3.6. 인터페이스를 참조하는 모델 스키마 구성

모델 스키마가 루트 리소스의 기본 인터페이스인 **Java** 인터페이스에 적용되는 경우 엔드포인트에서 **jaxrs:serviceBeans** 요소를 사용하여 루트 리소스 클래스를 인스턴스화해야 합니다.

예를 들어 **customer-interfaces.xml** 이 메타데이터를 고객 인터페이스와 연결하는 모델 스키마라고 가정하면 다음과 같이 **customerService** 서비스 엔드 포인트를 인스턴스화할 수 있습니다.

```
<jaxrs:server id="customerService"
              address="/customers"
              modelRef="classpath:/org/example/schemas/customer-interfaces.xml">
    <jaxrs:serviceBeans>
      <ref component-id="serviceBean" />
    </jaxrs:serviceBeans>
</jaxrs:server>

<bean id="serviceBean" class="service.CustomerService"/>
```

18.3.7. 모델 스키마 참조

*모델 스키마는 다음 **XML** 요소를 사용하여 정의됩니다.**A model schema is defined using the following XML elements:***

model

*모델 스키마의 루트 요소입니다. **model** 스키마를 참조해야 하는 경우(예: **modelRef** 특성을 사용하는 **JAX-RS** 끝점에서 이 요소에 대한 **id** 특성을 설정해야 합니다.*

model/resource

리소스 요소는 메타데이터를 특정 루트 리소스 클래스(또는 해당 인터페이스와 연결)하는 데 사용됩니다. 리소스 요소에서 다음 속성을 정의할 수 있습니다.

속성	description +
name	이 리소스 모델이 적용되는 리소스 클래스(또는 해당 인터페이스)의 이름입니다. +
path	이 리소스에 매핑되는 REST URL 경로의 구성 요소입니다. +
consumes	이 리소스에서 사용하는 콘텐츠 유형(인터넷 미디어 유형)을 지정합니다(예: **application/xml** 또는 **application/json**). +
produces	이 리소스에서 생성한 콘텐츠 유형(인터넷 미디어 유형)을 지정합니다(예: **application/xml** 또는 **application/json**). +

model/resource/operation

*작업 요소는 메타데이터를 **Java** 메서드와 연결하는 데 사용됩니다. 작업 요소에서 다음 특성을 정의할 수 있습니다.*

속성	description +
name	이 요소가 적용되는 Java 메서드의 이름입니다. +
path	이 메서드에 매핑되는 REST URL 경로의 구성 요소입니다. 이 특성 값에는 매개변수 참조(예: **path="/books/{id}/chapter"**)가 포함될 수 있으며, 여기서 **{id}** 는 경로에서 **id** 매개 변수 값을 추출합니다. +
verb	이 메서드에 매핑되는 HTTP 동사를 지정합니다. 일반적으로 **GET**,**POST**,**PUT** 또는 **DELETE** 중 하나입니다. HTTP 동사를 **지정하지** 않으면 Java 메서드가 **하위 리소스 오브젝트에 대한 참조를 반환하는 하위 리소스 locater** 이라고 가정합니다(sub-**resource** 클래스는 리소스 요소를 사용하는 메타데이터도 제공해야 함). +
consumes	이 작업에서 사용하는 콘텐츠 유형(인터넷 미디어 유형)을 지정합니다(예: **application/xml** 또는 **application/json**). +
produces	이 작업에서 생성한 콘텐츠 유형(인터넷 미디어 유형)을 지정합니다(예: **application/xml** 또는 **application/json**). +
oneWay	**true** 인 경우 작업을**단방향** 으로 구성합니다. 즉 응답 메시지가 필요하지 않습니다. 기본값은 **false**입니다. +

model/resource/operation/param

***param** 요소는 **REST URL**에서 값을 추출하여 메서드 매개변수 중 하나에 삽입합니다. **param** 요소에서 다음 속성을 정의할 수 있습니다.*

속성	description +
name	이 요소가 적용되는 Java 메서드 매개 변수의 이름입니다. +
type	REST URL 또는 메시지에서 매개변수 값을 추출하는 방법을 지정합니다. **PATH**,**QUERY**,**MATRIX**,**HEADER**,**COOKIE**,**FORM**,**CONTEXT**,**REQUEST_BODY** 값 중 하나로 설정할 수 있습니다. +
defaultValue	REST URL 또는 메시지에서 값을 추출할 기본값입니다. +
인코딩	**true** 인 경우 매개 변수 값이 URI 인코딩 형식으로 삽입됩니다(즉, **%nn** 인코딩 사용). 기본값은 **false** 입니다. 예를 들어, URL 경로에서 매개 변수를 추출하는 경우, 인코딩이 **true** 로 설정된 **/name/Joe%20Bloggs** 에서 매개 변수를 추출할 때 매개 변수가 **Joe%20Bloggs** 로 삽입되고, 그렇지 않으면 매개 변수가 **Joe Bloggs** 로 삽입됩니다. +

19장. APACHE CXF LOGGING

초록

이 장에서는 **Apache CXF** 런타임에서 로깅을 구성하는 방법을 설명합니다.

19.1. APACHE CXF 로깅 개요

19.1.1. 개요

Apache CXF는 **Java** 로깅 유틸리티인 **java.util.logging** 을 사용합니다. 로깅은 표준 **java.util.Properties** 형식을 사용하여 작성된 로깅 구성 파일로 구성됩니다. 애플리케이션에서 로깅을 실행하려면 프로그래밍 방식으로 로깅을 지정하거나 애플리케이션을 시작할 때 로깅 구성 파일을 가리키는 명령에서 속성을 정의하여 지정할 수 있습니다.

19.1.2. 기본 속성 파일

Apache CXF에는 **InstallDir/etc** 디렉토리에 있는 기본 **logging.properties** 파일이 제공됩니다. 이 파일은 로그 메시지의 출력 대상과 게시된 메시지 수준을 모두 구성합니다. 기본 구성은 **WARNING** 수준으로 플래그가 지정된 로거를 콘솔에 인쇄하도록 설정합니다. 구성 설정을 변경하지 않고 기본 파일을 사용하거나 특정 애플리케이션에 맞게 구성 설정을 변경할 수 있습니다.

19.1.3. 로깅 기능

Apache CXF에는 로깅을 활성화하기 위해 클라이언트 또는 서비스에 연결할 수 있는 로깅 기능이 포함되어 있습니다. 예 19.1. “로깅 활성화를 위한 구성” 로깅 기능을 활성화하는 구성을 보여줍니다.

예 19.1. 로깅 활성화를 위한 구성

```
<jaxws:endpoint...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

자세한 내용은 19.6절. “메시지 콘텐츠 로깅”의 내용을 참조하십시오.

19.1.4. 어디서 시작할 수 있습니까?

간단한 로깅 예제를 실행하려면 19.2절. "로깅 사용의 간단한 예" 에 설명된 지침을 따릅니다.

Apache CXF에서 로깅이 작동하는 방법에 대한 자세한 내용은 이 전체 장을 참조하십시오.

19.1.5. java.util.logging에 대한 추가 정보

java.util.logging 유틸리티는 가장 널리 사용되는 Java 로깅 프레임워크 중 하나입니다. 이 프레임워크를 사용하고 확장하는 방법을 설명하는 온라인으로 사용 가능한 많은 정보가 있습니다. 그러나 다음 문서는 java.util.logging 에 대한 좋은 개요를 제공합니다.

- http://download.oracle.com/javase/1.5.0/docs/guide/logging/overview.html
- http://download.oracle.com/javase/1.5.0/docs/api/java/util/logging/package-summary.html

19.2. 로깅 사용의 간단한 예

19.2.1. 로그 수준 및 출력 대상 변경

wsdl_first 샘플 애플리케이션에서 로그 메시지의 로그 수준 및 출력 대상을 변경하려면 다음 단계를 완료합니다.

1. InstallDir/samples/wsdl_first 디렉터리에 있는 README.txt 파일의 java 섹션을 사용하여 데모 실행에 설명된 대로 샘플 서버를 실행합니다. server start 명령은 다음과 같이 기본 logging.properties 파일을 지정합니다.

플랫폼	명령 +
Windows	**Java -Djava.util.config.file=%CXF_HOME%\etc\logging.properties demo.hw.server.Server** 를 시작합니다. +

플랫폼	명령 +
UNIX	**java -Djava.util.logging.config.file=$CXF_HOME/etc/logging.properties demo.hw.server.Server &** +

*기본 **logging.properties** 파일은 **InstallDir/etc** 디렉터리에 있습니다. **Apache CXF** 로거를 구성하여 **WARNING** 수준 로그 메시지를 콘솔에 인쇄합니다. 따라서 콘솔에 거의 인쇄되지 않습니다.*

2. *README.txt 파일에 설명된 대로 서버를 중지합니다.*

3. *기본 **logging.properties** 파일의 사본을 만들고 **mylogging.properties** 파일의 이름을 지정하고 기본 **logging.properties** 파일과 동일한 디렉토리에 저장합니다.*

4. *다음 구성 행을 편집하여 글로벌 로깅 수준 및 **mylogging.properties** 파일의 콘솔 로깅 수준을 **INFO** 로 변경합니다.*

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

5. *다음 명령을 사용하여 서버를 다시 시작합니다.*

플랫폼	명령 +
Windows	**Java -Djava.util.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server**를 시작합니다. +
UNIX	**Java -Djava.util.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &** +

수준 INFO 의 메시지를 기록하도록 글로벌 로깅 및 콘솔 로거를 구성했기 때문에 콘솔에 인쇄된 더 많은 로그 메시지가 표시됩니다.

19.3. 기본 로깅 구성 파일

19.3.1. 로깅 구성 개요

기본 로깅 구성 파일 **logging.properties** 는 **InstallDir/etc** 디렉터리에 있습니다. **Apache CXF** 로거를 구성하여 경고 수준 메시지를 콘솔에 인쇄합니다. 이 수준의 로깅이 애플리케이션에 적합한 경우 사용하기 전에 파일을 변경할 필요가 없습니다. 그러나 로그 메시지의 세부 수준을 변경할 수 있습니다. 예를 들어 로그 메시지가 콘솔로 전송되는지, 파일 또는 둘 다로 전송되는지 여부를 변경할 수 있습니다. 또한 개별 패키지 수준에서 로깅을 지정할 수 있습니다.

참고

이 섹션에서는 기본 **logging.properties** 파일에 표시되는 구성 속성에 대해 설명합니다. 그러나 설정할 수 있는 다른 많은 **java.util.logging** 구성 속성이 있습니다. **java.util.logging API**에 대한 자세한 내용은 **java.util.logging javadoc at:** http://download.oracle.com/javase/1.5/docs/api/java/util/logging/package-summary.html 을 참조하십시오.

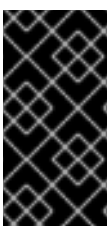
19.3.2. 로깅 출력 구성

19.3.2.1. 개요

Java 로깅 유틸리티인 **java.util.logging** 은 핸들러 클래스를 사용하여 로그 메시지를 출력합니다. 표 19.1. “java.util.logging Handler 클래스” 는 기본 **logging.properties** 파일에 구성된 핸들러를 보여줍니다.

표 19.1. java.util.logging Handler 클래스

처리기 클래스	다음에 대한 출력
ConsoleHandler	콘솔에 로그 메시지 출력
FileHandler	파일에 로그 메시지 출력



중요

처리기 클래스는 시작될 때 **Java VM**에 의해 설치되려면 시스템 클래스 경로에 있어야 합니다. 이 작업은 **Apache CXF** 환경을 설정할 때 수행됩니다.

19.3.2.2. 콘솔 핸들러 구성

예 19.2. "콘솔 핸들러 구성" 콘솔 로거를 구성하는 코드를 보여줍니다.

예 19.2. 콘솔 핸들러 구성

```
handlers= java.util.logging.ConsoleHandler
```

콘솔 처리기는 예 19.3. "콘솔 핸들러 속성" 에 표시된 구성 속성도 지원합니다.

예 19.3. 콘솔 핸들러 속성

```
java.util.logging.ConsoleHandler.level = WARNING
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

예 19.3. "콘솔 핸들러 속성" 에 표시된 구성 속성은 다음과 같이 설명할 수 있습니다.

콘솔 핸들러는 별도의 로그 수준 구성 속성을 지원합니다. 이를 통해 글로벌 로깅 설정이 다를 수 있는 동안 콘솔에 인쇄된 로그 메시지를 제한할 수 있습니다(19.3.3절. "로깅 수준 구성"참조). 기본 설정은 **WARNING** 입니다.

콘솔 처리기 클래스에서 로그 메시지를 포맷하는 데 사용하는 **java.util.logging formatter** 클래스를 지정합니다. 기본 설정은 **java.util.logging.SimpleFormatter** 입니다.

19.3.2.3. 파일 핸들러 구성

예 19.4. "파일 핸들러 구성" 파일 핸들러를 구성하는 코드를 보여줍니다.

예 19.4. 파일 핸들러 구성

```
handlers= java.util.logging.FileHandler
```

파일 핸들러는 예 19.5. "파일 핸들러 구성 속성" 에 표시된 구성 속성도 지원합니다.

예 19.5. 파일 핸들러 구성 속성

```
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

예 19.5. "파일 핸들러 구성 속성" 에 표시된 구성 속성은 다음과 같이 설명할 수 있습니다.

출력 파일의 위치 및 패턴을 지정합니다. 기본 설정은 홈 디렉토리입니다.

*로거가 하나의 파일에 쓰는 최대 양을 바이트 단위로 지정합니다. 기본 설정은 **50000** 입니다. 이를 **0** 으로 설정하면 로거가 하나의 파일에 쓰는 양에 제한이 없습니다.*

*사이클할 출력 파일 수를 지정합니다. 기본 설정은 **1** 입니다.*

*파일 처리기 클래스에서 로그 메시지를 포맷하는 데 사용하는 **java.util.logging formatter** 클래스를 지정합니다. 기본 설정은 **java.util.logging.XMLFormatter** 입니다.*

19.3.2.4. 콘솔 핸들러와 파일 핸들러 모두 구성

*콘솔 처리기와 파일 핸들러를 콘솔 로깅 및 파일 모두 구성 와 같이 쉼표로 구분하여 지정하여 콘솔 및 파일에 로그 메시지를 출력하도록 **logging** 유틸리티를 설정할 수 있습니다.*

콘솔 로깅 및 파일 모두 구성

```
Logging
```

```
handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```

19.3.3. 로깅 수준 구성

19.3.3.1. 로깅 수준

java.util.logging 프레임워크는 최소 세부 정보에서 가장 자세한 정보까지 다음과 같은 로깅 수준을 지원합니다.

- **SEVERE**
- **WARNING**
- **INFO**
- **CONFIG**
- **FINE**
- **FINER**
- **FINEST**

19.3.3.2. 글로벌 로깅 수준 구성

모든 로거에 기록된 이벤트 유형을 구성하려면 예 19.6. "글로벌 로깅 수준 구성" 에 표시된 대로 글로벌 로깅 수준을 구성합니다.

> *예 19.6. 글로벌 로깅 수준 구성*
>
> ```
> .level= WARNING
> ```

19.3.3.3. 개별 패키지에서 로깅 구성

> **level**

java.util.logging 프레임워크는 개별 패키지 수준에서 로깅 구성을 지원합니다. 예를 들어 예 19.7. "패키지 수준에서 로깅 구성" 에 표시된 코드 줄은 com.xyz.foo 패키지의 클래스의 SEVERE 수준에서 로깅을 구성합니다.

예 19.7. 패키지 수준에서 로깅 구성

```
com.xyz.foo.level = SEVERE
```

19.4. 명령줄에서 로깅 활성화

19.4.1. 개요

애플리케이션을 시작할 때 **java.util.logging.config.file** 속성을 정의하여 애플리케이션에서 로깅 유틸리티를 실행할 수 있습니다. 기본 **logging.properties** 파일 또는 해당 애플리케이션에 고유한 **logging.properties** 파일을 지정할 수 있습니다.

19.4.2. 애플리케이션에서 로그 구성 파일 지정

start-up

애플리케이션 시작 시 로깅을 지정하려면 애플리케이션을 시작할 때 예 19.8. "명령줄에서 로깅 시작 플래그"에 표시된 플래그를 추가합니다.

예 19.8. 명령줄에서 로깅 시작 플래그

```
-Djava.util.logging.config.file=myfile
```

19.5. 하위 시스템 및 서비스에 대한 로깅

19.5.1. 개요

"개별 패키지에서 로깅 구성"에 설명된 **com.xyz.foo.level** 구성 속성을 사용하여 지정된 **Apache CXF** 로깅 하위 시스템에 대한 세분화된 로깅을 설정할 수 있습니다.

19.5.2. Apache CXF 로깅 하위 시스템

표 19.2. "Apache CXF 로깅 하위 시스템" 사용 가능한 **Apache CXF** 로깅 하위 시스템 목록을 표시합니다.

표 19.2. Apache CXF 로깅 하위 시스템

하위 시스템	설명
org.apache.cxf.aegis	Aegis 바인딩
org.apache.cxf.binding.coloc	공동 배치 바인딩
org.apache.cxf.binding.http	HTTP 바인딩
org.apache.cxf.binding.jbi	JBI 바인딩
org.apache.cxf.binding.object	Java 오브젝트 바인딩
org.apache.cxf.binding.soap	SOAP 바인딩
org.apache.cxf.binding.xml	XML 바인딩
org.apache.cxf.bus	Apache CXF 버스
org.apache.cxf.configuration	구성 프레임워크
org.apache.cxf.endpoint	서버 및 클라이언트 끝점
org.apache.cxf.interceptor	인터셉터
org.apache.cxf.jaxws	JAX-WS 스타일 메시지 교환, JAX-WS 핸들러 처리, JAX-WS 및 구성과 관련된 인터셉터
org.apache.cxf.jbi	JBI 컨테이너 통합 클래스
org.apache.cxf.jca	JCA 컨테이너 통합 클래스
org.apache.cxf.js	JavaScript 프론트 엔드
org.apache.cxf.transport.http	HTTP 전송
org.apache.cxf.transport.https	HTTPS를 사용하여 HTTP 전송의 보안 버전
org.apache.cxf.transport.jbi	JBI 전송
org.apache.cxf.transport.jms	JMS 전송
org.apache.cxf.transport.local	로컬 파일 시스템을 사용하는 전송 구현
org.apache.cxf.transport.servlet	JAX-WS 엔드포인트를 서블릿 컨테이너에 로드하기 위한 HTTP 전송 및 서블릿 구현
org.apache.cxf.ws.addressing	WS-Addressing 구현

하위 시스템	설명
org.apache.cxf.ws.policy	WS-Policy 구현
org.apache.cxf.ws.rm	WS-ReliableMessaging (WS-RM) 구현
org.apache.cxf.ws.security.wss4j	WSS4J 보안 구현

19.5.3. 예제

***WS-Addressing** 샘플은 **InstallDir/samples/ws_addressing** 디렉터리에 포함되어 있습니다. 로깅은 해당 디렉터리에 있는 **logging.properties** 파일에서 구성됩니다. 관련 구성 행은 예 19.9. "WS-Addressing에 대한 로깅 구성" 에 표시되어 있습니다.*

예 19.9. WS-Addressing에 대한 로깅 구성

```
java.util.logging.ConsoleHandler.formatter = demos.ws_addressing.common.ConciseFormatter
...
org.apache.cxf.ws.addressing.soap.MAPCodec.level = INFO
```

*예 19.9. "WS-Addressing에 대한 로깅 구성" 의 구성은 **WS-Addressing** 헤더와 관련된 로그 메시지를 스누핑할 수 있으며 콘솔에 간결한 형식으로 표시할 수 있습니다.*

*이 샘플 실행에 대한 자세한 내용은 **InstallDir/samples/ws_addressing** 디렉터리에 있는 **README.txt** 파일을 참조하십시오.*

19.6. 메시지 콘텐츠 로깅

19.6.1. 개요

*서비스와 소비자 간에 전송되는 메시지의 콘텐츠를 기록할 수 있습니다. 예를 들어 서비스와 소비자 간에 전송되는 **SOAP** 메시지의 콘텐츠를 로깅할 수 있습니다.**For example, you might want to log the contents of SOAP messages that are being sent between a service and a consumer.***

19.6.2. 메시지 콘텐츠 로깅 구성

서비스와 소비자 간에 전송되는 메시지를 로깅하고 그 반대의 경우도 마찬가지이면 다음 단계를 완료합니다.

1. 끝점의 구성에 로깅 기능을 추가합니다.

2. 사용자의 구성에 로깅 기능을 추가합니다.

3. 로깅 시스템 로그 **INFO** 수준 메시지를 구성합니다.

19.6.3. 끝점에 로깅 기능 추가

예 19.10. "끝점 구성에 로깅 추가" 에 표시된 대로 엔드 포인트의 구성에 로깅 기능을 추가합니다.

예 19.10. 끝점 구성에 로깅 추가

```
<jaxws:endpoint ...>
  <jaxws:features>
    <bean class="org.apache.cxf.feature.LoggingFeature"/>
  </jaxws:features>
</jaxws:endpoint>
```

예 19.10. "끝점 구성에 로깅 추가" 에 표시된 **XML** 예제를 사용하면 **SOAP** 메시지 로깅을 사용할 수 있습니다.

19.6.4. 소비자에게 로깅 기능 추가

클라이언트 구성이 예 19.11. "클라이언트 구성에 로깅 추가" 와 같이 로깅 기능을 추가합니다.

예 19.11. 클라이언트 구성에 로깅 추가

```
<jaxws:client ...>
   <jaxws:features>
     <bean class="org.apache.cxf.feature.LoggingFeature"/>
    </jaxws:features>
</jaxws:client>
```

예 19.11. "클라이언트 구성에 로깅 추가" 에 표시된 **XML** 예제를 사용하면 **SOAP** 메시지 로깅을 사용할 수 있습니다.

19.6.5. 로깅을 log INFO 수준 메시지로 설정

서비스와 관련된 **logging.properties** 파일이 [예 19.12. "로깅 수준을 INFO로 설정"](#) 와 같이 **INFO** 수준 메시지를 기록하도록 구성되어 있는지 확인합니다.

예 19.12. 로깅 수준을 **INFO**로 설정

```
.level= INFO
java.util.logging.ConsoleHandler.level = INFO
```

19.6.6. SOAP 메시지 로깅

SOAP 메시지의 로깅을 보려면 **InstallDir/samples/wsdl_first** 디렉터리에 있는 **wsdl_first** 샘플 애플리케이션을 다음과 같이 수정합니다.

1. [예 19.13. "로깅에 대한 엔드 포인트 구성 SOAP 메시지"](#) 에 표시된 **jaxws:features** 요소를 **wsdl_first** 샘플 디렉터리에 있는 **cxf.xml** 구성 파일에 추가합니다.

   예 19.13. 로깅에 대한 엔드 포인트 구성 **SOAP** 메시지

   ```
   <jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                   createdFromAPI="true">
     <jaxws:properties>
       <entry key="schema-validation-enabled" value="true" />
     </jaxws:properties>
     <jaxws:features>
       <bean class="org.apache.cxf.feature.LoggingFeature"/>
     </jaxws:features>
   </jaxws:endpoint>
   ```

2. 샘플은 **InstallDir/etc** 디렉터리에 있는 기본 **logging.properties** 파일을 사용합니다. 이 파일의 사본을 만들고 이름을 **mylogging.properties** 로 지정합니다.

3. **mylogging.properties** 파일에서 **.level** 및 **java.util.logging.ConsoleHandler .level** 구성 속성을 다음과 같이 편집하여 로깅 수준을 **INFO** 로 변경합니다.

   ```
   .level= INFO
   java.util.logging.ConsoleHandler.level = INFO
   ```

4. 다음과 같이 **cxf.xml** 파일과 **mylogging.properties** 파일의 새 구성 설정을 사용하여 서버를 시작합니다.

플랫폼	명령 +
Windows	**Java -Djava.util.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.server.Server**를 시작합니다. +
UNIX	**Java -Djava.util.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.server.Server &** +

5. *다음 명령을 사용하여 **hello world** 클라이언트를 시작합니다.*

플랫폼	명령 +
Windows	**Java -Djava.util.config.file=%CXF_HOME%\etc\mylogging.properties demo.hw.client.Client .\wsdl\hello_world.wsdl** +
UNIX	**Java -Djava.util.config.file=$CXF_HOME/etc/mylogging.properties demo.hw.client.Client ./wsdl/hello_world.wsdl** +

***SOAP** 메시지는 콘솔에 기록됩니다.*

20장. WS-ADDRESSING 배포

초록

Apache CXF는 JAX-WS 애플리케이션에 대해 WS-Addressing을 지원합니다. 이 장에서는 Apache CXF 런타임 환경에 WS-Addressing을 배포하는 방법을 설명합니다.

20.1. WS-ADDRESSING 소개

20.1.1. 개요

WS-Addressing은 서비스가 전송 중립적인 방식으로 주소 지정 정보를 통신할 수 있도록 하는 사양입니다. 그것은 두 부분으로 구성되어 있습니다:

- 웹 서비스 끝점에 참조를 전달하는 구조
- 특정 메시지와 주소 지정 정보를 연결하는 MAP(Message Addressing Properties) 세트

20.1.2. 지원되는 사양

Apache CXF는 WS-Addressing 2004/08 사양과 WS-Addressing 2005/03 사양을 모두 지원합니다.

20.1.3. 더 많은 정보

WS-Addressing에 대한 자세한 내용은 2004/08 제출 (http://www.w3.org/Submission/ws-addressing/ 참조하십시오.

20.2. WS-ADDRESSING INTERCEPTORS

20.2.1. 개요

Apache CXF에서는 WS-Addressing 기능이 인터셉터로 구현됩니다. Apache CXF 런타임은 인터셉터를 사용하여 전송 및 수신되는 원시 메시지를 가로채고 작업합니다. 전송에서 메시지를 수신하면 메시지 오브젝트를 생성하고 인터셉터 체인을 통해 해당 메시지를 전송합니다. 애플리케이션 인터셉터 체인에 WS-Addressing 인터셉터가 추가되면 메시지에 포함된 WS-Addressing 정보가 처리됩니다.

20.2.2. WS-Addressing Interceptors

WS-Addressing 구현은 표 20.1. “WS-Addressing Interceptors” 에 설명된 대로 두 개의 인터셉터로 구성됩니다.

표 20.1. WS-Addressing Interceptors

인터셉터	설명
org.apache.cxf.ws.addressing.MAPAggregator	발신 메시지의MAP(Message Addressing Properties)를 집계하는 논리 인터셉터입니다.
org.apache.cxf.ws.addressing.soap.MAPCodec	메시지 주소 지정 속성(MAP)을 SOAP 헤더로 인코딩하고 디코딩해야 하는 프로토콜별 인터셉터입니다.

20.3. WS-ADDRESSING 활성화

20.3.1. 개요

WS-Addressing 인터셉터를 활성화하려면 인바운드 및 아웃바운드 인터셉터 체인에 WS-Addressing 인터셉터를 추가해야 합니다. 이 작업은 다음 방법 중 하나로 수행됩니다.

- *Apache CXF 기능*
- *RMAssertion 및 WS-Policy Framework*
- *WS-Addressing 기능에서 Policy assertionion 사용*

20.3.2. WS-Addressing을 기능으로 추가

WS-Addressing은 각각 예 20.1. “client.xml 및 클라이언트 구성에 WS-Addressing 기능 추가” 및 예 20.2. “server.xml 및 서버 구성에 WS-Addressing 기능 추가” 과 같이 클라이언트 및 서버 구성에 WS-Addressing 기능을 추가하여 활성화할 수 있습니다.

예 20.1. client.xml 및 클라이언트 구성에 WS-Addressing 기능 추가

```
<?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:jaxws="http://cxf.apache.org/jaxws"
      xmlns:wsa="http://cxf.apache.org/ws/addressing"
      xsi:schemaLocation="
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/ws/addressing
        http://cxf.apache.org/schemas/ws-addr-conf.xsd">

    <jaxws:client ...>
      <jaxws:features>
        <wsa:addressing/>
      </jaxws:features>
    </jaxws:client>
</beans>
```

예 20.2. server.xml 및 서버 구성에 WS-Addressing 기능 추가

```
<?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:jaxws="http://cxf.apache.org/jaxws"
      xmlns:wsa="http://cxf.apache.org/ws/addressing"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
      <jaxws:features>
        <wsa:addressing/>
      </jaxws:features>
    </jaxws:endpoint>
</beans>
```

20.4. WS-ADDRESSING ATTRIBUTES 구성

20.4.1. 개요

Apache CXF WS-Addressing 기능 요소는 네임스페이스 http://cxf.apache.org/ws/addressing. 표 20.2. “WS-Addressing Attributes” 에 설명된 두 가지 속성을 지원합니다.

표 20.2. WS-Addressing Attributes

특성 이름	값
allowDuplicates	중복 MessageID가 허용되는지 여부를 결정하는 부울입니다. 기본 설정은 **true** 입니다.
usingAddressingAdvisory	WSDL에 **UsingAddressing** 요소의 존재 여부를 나타내는 부울은 권고만 합니다. 즉, WS-Addressing 헤더의 인코딩을 방지하지 않습니다.

20.4.2. WS-Addressing 속성 구성

서버 또는 클라이언트 구성 파일의 **WS-Addressing** 기능으로 설정하려는 속성 및 값을 추가하여 **WS-Addressing** 특성을 구성합니다. 예를 들어 다음 구성 추출에서는 서버 끝점에서 **allowDuplicates** 특성을 **false** 로 설정합니다.

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
   <jaxws:endpoint ...>
      <jaxws:features>
         <wsa:addressing allowDuplicates="false"/>
      </jaxws:features>
   </jaxws:endpoint>
</beans>
```

20.4.3. 기능에 포함된 WS-Policy 어설션 사용

예 20.3. “정책을 사용하여 WS-Addressing 구성” 에서 비익명 응답을 활성화하는 주소 지정 정책 어설션이 정책 요소에 포함됩니다.

예 20.3. 정책을 사용하여 WS-Addressing 구성

```
<?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:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

   <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
                   createdFromAPI="true">
      <jaxws:features>
         <policy:policies>
            <wsp:Policy xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
               <wsam:Addressing>
                  <wsp:Policy>
                     <wsam:NonAnonymousResponses/>
                  </wsp:Policy>
               </wsam:Addressing>
            </wsp:Policy>
         <policy:policies>
```

```
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

21장. 신뢰할 수 있는 메시징 활성화

초록

Apache CXF는 **WS-Reliable Messaging(WS-RM)**을 지원합니다. 이 장에서는 **Apache CXF**에서 **WS-RM**을 활성화하고 구성하는 방법을 설명합니다.

21.1. WS-RM 소개

21.1.1. 개요

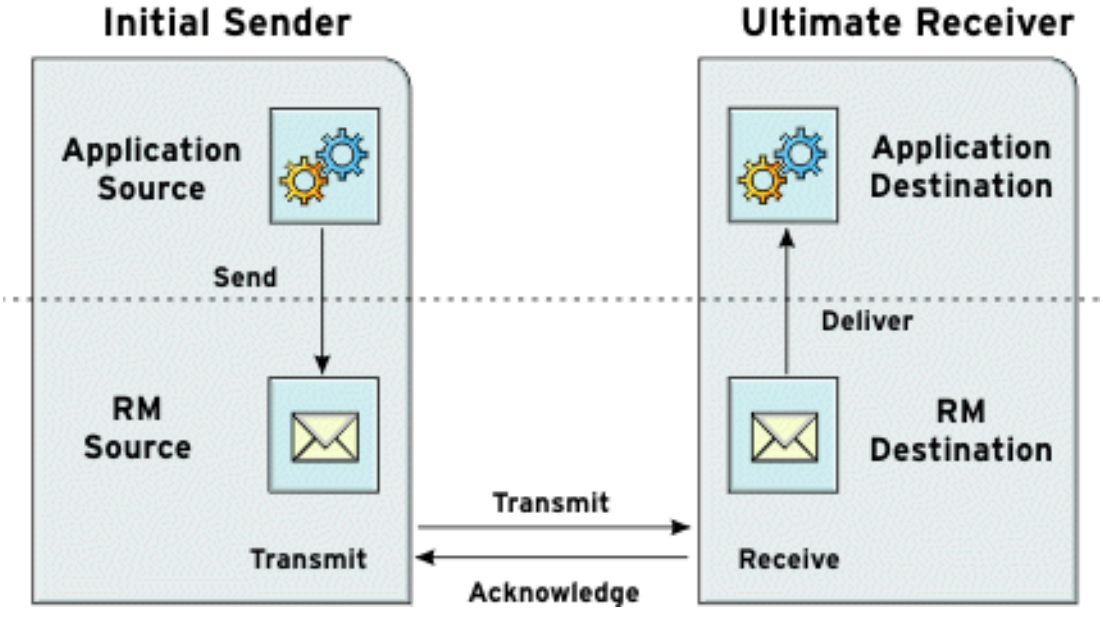
WS-ReliableMessaging (WS-RM)은 분산된 환경에서 메시지의 안정적인 전달을 보장하는 프로토콜입니다. 이를 통해 소프트웨어, 시스템 또는 네트워크 오류가 발생할 경우 분산 애플리케이션 간에 메시지를 안정적으로 전달할 수 있습니다.

예를 들어 **WS-RM**을 사용하면 올바른 메시지가 정확하게 한 번 올바르게 네트워크에 전달되었는지 확인할 수 있습니다.

21.1.2. WS-RM의 작동 방식

WS-RM은 소스와 대상 끝점 간 메시지의 안정적인 전달을 보장합니다. 소스는 메시지의 초기 발신자이고 대상은 그림 21.1. "웹 서비스 신뢰할 수 있는 메시징"에 표시된 대로 궁극적인 수신자입니다.

그림 21.1. 웹 서비스 신뢰할 수 있는 메시징



WS-RM 메시지 흐름은 다음과 같이 설명될 수 있습니다.

1. *RM 소스는 CreateSequence 프로토콜 메시지를 RM 대상으로 보냅니다. 여기에는 승인을 받는 끝점에 대한 참조(wsrm:AcksTo 끝점)가 포함되어 있습니다.*

2. *RM 대상은 CreateSequenceResponse 프로토콜 메시지를 다시 RM 소스로 보냅니다. 이 메시지에는 RM 시퀀스 세션의 시퀀스 ID가 포함되어 있습니다.*

3. *RM 소스는 애플리케이션 소스에서 보낸 각 메시지에 RM Sequence 헤더를 추가합니다. 이 헤더에는 시퀀스 ID와 고유한 메시지 ID가 포함됩니다.*

4. *RM 소스는 각 메시지를 RM 대상으로 전송합니다.*

5. *RM 대상은 RM SequenceAcknowledgement 헤더가 포함된 메시지를 전송하여 RM 소스에서 메시지 수신을 확인합니다.*

6. *RM 대상은 정확히 순서 방식으로 애플리케이션 대상에 메시지를 전달합니다.*

7. *RM 소스는 아직 승인을 받지 못한 메시지를 다시 전송합니다.*

   첫 번째 재전송 시도는 기본 재전송 간격 후에 수행됩니다. 연속 재전송 시도는 기본적으로 기하급수 백오프 간격 또는 고정된 간격으로 수행됩니다. 자세한 내용은 21.5절. “WS-RM 구성” 에서 참조하십시오.

이 전체 프로세스는 요청 및 응답 메시지에 대해 대칭적으로 발생합니다. 즉, 응답 메시지의 경우 서버는 RM 소스의 역할을 하고 클라이언트는 RM 대상의 역할을 합니다.

21.1.3. WS-RM 제공 보장

WS-RM은 사용된 전송 프로토콜에 관계없이 분산된 환경에서 안정적인 메시지 전달을 보장합니다. 신뢰할 수 있는 전달을 보장할 수 없는 경우 소스 또는 대상 끝점에서 오류를 기록합니다.

21.1.4. 지원되는 사양

Apache CXF는 다음 버전의 WS-RM 사양을 지원합니다.

WS-ReliableMessaging 1.0

(기본값) 2005년 2월의 후보 접수 버전을 종료했으며, 현재 출시되지 않았습니다. 그러나 이전 버전과의 호환성을 이유로 이 버전은 기본값으로 사용됩니다.

WS-RM 버전 1.0에서는 다음 네임스페이스를 사용합니다.

> http://schemas.xmlsoap.org/ws/2005/02/rm/

WS-RM의 이 버전은 다음 WS-Addressing 버전 중 하나와 함께 사용할 수 있습니다.

> http://schemas.xmlsoap.org/ws/2004/08/addressing (default)
> http://www.w3.org/2005/08/addressing

WS-RM의 2005년 2월 릴리스 버전을 준수하기 위해 이러한 WS-Addressing 버전(Apache CXF의 기본값) 중 첫 번째 버전을 사용해야 합니다. 그러나 대부분의 다른 웹 서비스 구현에서는 최근에 WS-Addressing 사양으로 전환되어 Apache CXF를 사용하면 WS-A 버전을 선택하여 상호 운용성을 원활하게 수행할 수 있습니다. 21.4절. “런타임 제어”

WS-ReliableMessaging 1.1/1.2

1.1/1.2 Web Services Reliable Messaging 사양에 해당합니다.

WS-RM 버전 1.1 및 1.2에서는 다음 네임스페이스를 사용합니다.

> http://docs.oasis-open.org/ws-rx/wsrm/200702

WS-RM의 1.1 및 1.2 버전에서는 다음 WS-Addressing 버전을 사용합니다.

> http://www.w3.org/2005/08/addressing

21.1.5. WS-RM 버전 선택

다음과 같이 사용할 WS-RM 사양 버전을 선택할 수 있습니다.

서버 측

공급자 측면에서 Apache CXF는 클라이언트에서 WS-ReliableMessaging을 사용하는 모든 버전에 적응하고 적절하게 응답합니다.

고객 측면

*클라이언트 측에서 **WS-RM** 버전은 클라이언트 구성에서 사용하는 네임스페이스에 의해 결정되거나(21.5절. "WS-RM 구성"참조) 런타임 제어 옵션을 사용하여 런타임에 **WS-RM** 버전을 재정의하여 결정됩니다(21.4절. "런타임 제어"참조).*

21.2. WS-RM 인터셉터

21.2.1. 개요

***Apache CXF**에서는 **WS-RM** 기능이 인터셉터로 구현됩니다. **Apache CXF** 런타임은 인터셉터를 사용하여 전송 및 수신되는 원시 메시지를 가로채고 작업합니다. 전송에서 메시지를 수신하면 메시지 오브젝트를 생성하고 인터셉터 체인을 통해 해당 메시지를 전송합니다. 애플리케이션의 인터셉터 체인에 **WS-RM** 인터셉터가 포함된 경우 애플리케이션은 신뢰할 수 있는 메시징 세션에 참여할 수 있습니다. **WS-RM** 인터셉터는 메시지 청크의 수집 및 집계를 처리합니다. 또한 모든 승인 및 재전송 논리를 처리합니다.*

21.2.2. Apache CXF WS-RM 인터셉터

***Apache CXF WS-RM** 구현은 4개의 인터셉터로 구성되어 있으며, 이는 표 21.1. "Apache CXF WS-ReliableMessaging Interceptors" 에 설명되어 있습니다.*

표 21.1. Apache CXF WS-ReliableMessaging Interceptors

인터셉터	설명
org.apache.cxf.ws.rm.RMOutInterceptor	발신 메시지에 대한 신뢰성 보장을 제공하는 논리적인 측면을 다룹니다. **CreateSequence** 요청을 보내고 **CreateSequenceResponse** 응답을 기다립니다. 애플리케이션 메시지의 시퀀스 속성-ID 및 메시지 번호 집계를 담당합니다.Also responsible for agggregating the sequence properties-ID and message number- for an application message.
org.apache.cxf.ws.rm.RMInInterceptor	애플리케이션 메시지에서 금지된 RM 프로토콜 메시지 및 **시퀀스Acknowledgement** 메시지를 가로채기 및 처리할 책임이 있습니다.
org.apache.cxf.ws.rm.RMCaptureInInterceptor	영구 스토리지의 수신 메시지를 캐싱합니다.
org.apache.cxf.ws.rm.RMDeliveryInterceptor	애플리케이션에 대한 InOrder 메시지 제공.
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	신뢰성 속성을 SOAP 헤더로 인코딩 및 디코딩해야 합니다.

인터셉터	설명
org.apache.cxf.ws.rm.RetransmissionInterceptor	나중에 다시 전송할 수 있도록 애플리케이션 메시지의 복사본을 생성합니다.

21.2.3. WS-RM 활성화

*인터셉터 체인에 **WS-RM** 인터셉터가 있으면 필요한 경우 **WS-RM** 프로토콜 메시지가 교환됩니다. 예를 들어 아웃바운드 인터셉터 체인에서 첫 번째 애플리케이션 메시지를 가로채는 경우 **RMOutInterceptor** 는 **CreateSequence** 요청을 보내고 **CreateSequenceResponse** 응답을 수신할 때까지 원래 애플리케이션 메시지를 처리하도록 기다립니다. 또한 **WS-RM** 인터셉터는 애플리케이션 메시지에 시퀀스 헤더를 추가하고 대상 측에서 메시지를 추출합니다. 메시지를 안정적으로 교환하기 위해 애플리케이션 코드를 변경할 필요가 없습니다.*

***WS-RM**을 활성화하는 방법에 대한 자세한 내용은 21.3절. “WS-RM 활성화” 을 참조하십시오.*

21.2.4. WS-RM 속성 구성

*구성을 통해 시퀀스 분리 및 신뢰할 수 있는 교환의 다른 측면을 제어합니다. 예를 들어 기본적으로 **Apache CXF**는 시퀀스의 수명을 최대화하려고 하므로 대역 외 **WS-RM** 프로토콜 메시지로 인한 오버헤드가 감소합니다. 애플리케이션 메시지당 별도의 시퀀스를 사용하도록 하려면 **WS-RM** 소스의 시퀀스 종료 정책(최대 시퀀스 길이를 **1**로 설정)을 구성합니다.*

***WS-RM** 동작 구성에 대한 자세한 내용은 21.5절. “WS-RM 구성” 을 참조하십시오.*

21.3. WS-RM 활성화

21.3.1. 개요

*안정적인 메시징을 활성화하려면 인바운드 및 아웃바운드 메시지 및 결함 모두에 대해 인터셉터 체인에 **WS-RM** 인터셉터를 추가해야 합니다. **WS-RM** 인터셉터는 **WS-Addressing** 인터셉터를 사용하므로 인터셉터 체인에 **WS-Addressing** 인터셉터도 있어야 합니다.*

다음 두 가지 방법 중 하나로 이러한 인터셉터가 있는지 확인할 수 있습니다.

- 명시적으로 **Spring** 빈을 사용하여 디스패치 체인에 추가하여

- 암시적 으로 **WS-Policy** 어설션을 사용하여 **Apache CXF** 런타임이 사용자를 대신하여 인터셉터를 투명하게 추가합니다.

21.3.2. Spring bean: 인터셉터를 명시적으로 추가합니다.

WS-RM을 활성화하려면 **Apache CXF** 버스 또는 **Spring** 빈 구성을 사용하여 소비자 또는 서비스 엔드포인트에 **WS-RM** 및 **WS-Addressing** 인터셉터를 추가합니다. 이 방법은 **InstallDir/samples/ws_rm** 디렉터리에 있는 **WS-RM** 샘플에서 가져온 접근 방식입니다. 구성 파일 **ws-rm.cxf** 는 **Spring** 빈으로 일대일로 추가된 **WS-RM** 및 **WS-Addressing** 인터셉터를 보여줍니다(에 21.1. “Spring Bean을 사용하여 WS-RM 활성화”참조).

에 21.1. Spring Bean을 사용하여 WS-RM 활성화

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/
   beans http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="mapAggregator" class="org.apache.cxf.ws.addressing.MAPAggregator"/>
   <bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
   <bean id="rmLogicalOut" class="org.apache.cxf.ws.rm.RMOutInterceptor">
      <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
      <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmCodec" class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
   <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
      <property name="inInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalIn"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
      <property name="inFaultInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalIn"/>
            <ref bean="rmCodec"/>
         </list>
      </property>
      <property name="outInterceptors">
         <list>
            <ref bean="mapAggregator"/>
            <ref bean="mapCodec"/>
```

```
            <ref bean="rmLogicalOut"/>
            <ref bean="rmCodec"/>
          </list>
        </property>
        <property name="outFaultInterceptors">
          <list>
            <ref bean="mapAggregator">
            <ref bean="mapCodec"/>
            <ref bean="rmLogicalOut"/>
            <ref bean="rmCodec"/>
          </list>
        </property>
      </bean>
    </beans>
```

예 21.1. “Spring Bean을 사용하여 WS-RM 활성화” 에 표시된 코드는 다음과 같이 설명될 수 있습니다.

Apache CXF 구성 파일은 Spring XML 파일입니다. 빈 요소에서 캡슐화된 하위 요소 의 네임스페이스 및 스키마 파일을 선언하는 여는 Spring 빈 요소를 포함해야 합니다.

각 WS-Addressing 인터셉터-MAPAggregator 및 MAPCodec 를 구성합니다. WS-Addressing에 대한 자세한 내용은 20장. WS-Addressing 배포 을 참조하십시오.

각 WS-RM 인터셉터-RMOutInterceptor,RMInInterceptor 및 RMSoapInterceptor 를 설정합니다.

인바운드 메시지의 인터셉터 체인에 WS-Addressing 및 WS-RM 인터셉터를 추가합니다.

인바운드 결함을 위해 WS-Addressing 및 WS-RM 인터셉터를 인터셉터 체인에 추가합니다.

아웃바운드 메시지의 인터셉터 체인에 WS-Addressing 및 WS-RM 인터셉터를 추가합니다.

아웃바운드 결함의 인터셉터 체인에 WS-Addressing 및 WS-RM 인터셉터를 추가합니다.

21.3.3. WS-Policy 프레임워크: 암시적으로 인터셉터 추가

WS-Policy 프레임워크는 WS-Policy를 사용할 수 있는 인프라 및 API를 제공합니다. 이는 2006년 11월 Web Services Policy 1.5-Framework 및 Web Services Policy 1.5-Attachment 사양의 초안 발행물

과 호환됩니다.

Apache CXF WS-Policy 프레임워크를 사용하여 WS-RM을 활성화하려면 다음을 수행합니다.

1. 정책 기능을 클라이언트 및 서버 엔드포인트에 추가합니다. 예 21.2. "WS-Policy를 사용하여 WS-RM 구성" **jaxws:feature** 요소에 중첩된 참조 빈을 표시합니다. 참조 빈에서는 동일한 구성 파일 내의 별도의 요소로 정의된 **AddressingPolicy** 를 지정합니다.

예 21.2. WS-Policy를 사용하여 WS-RM 구성

```
<jaxws:client>
   <jaxws:features>
     <ref bean="AddressingPolicy"/>
   </jaxws:features>
</jaxws:client>
<wsp:Policy wsu:Id="AddressingPolicy"
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
   <wsam:Addressing>
     <wsp:Policy>
       <wsam:NonAnonymousResponses/>
     </wsp:Policy>
   </wsam:Addressing>
</wsp:Policy>
```

2. 예 21.3. "귀하의 WSDL 파일에 RM 정책 추가" 에 표시된 대로 정책 또는 정책 참조 요소의 연결 지점으로 사용할 수 있는 **wsdl:service** 요소 또는 기타 **WSDL** 요소에 안정적인 메시징 정책을 추가합니다.

예 21.3. 귀하의 WSDL 파일에 RM 정책 추가

```
<wsp:Policy wsu:Id="RM"
  xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">
   <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
      <wsp:Policy/>
   </wsam:Addressing>
   <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
      <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
   </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
   <wsdl:port binding="tns:GreeterSOAPBinding" name="GreeterPort">
      <soap:address location="http://localhost:9020/SoapContext/GreeterPort"/>
      <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ws-
```

```
policy"/>
    </wsdl:port>
</wsdl:service>
```

21.4. 런타임 제어

21.4.1. 개요

*런타임 시 **WS-RM**을 제어하기 위해 클라이언트 코드에서 여러 메시지 컨텍스트 속성 값을 설정할 수 있으며, **org.apache.cxf.ws.rm.RMManager** 클래스에 공용 상수로 정의된 키 값을 사용할 수 있습니다.*

21.4.2. 런타임 제어 옵션

*다음 표에는 **org.apache.cxf.ws.rm.RMManager** 클래스에서 정의된 키가 나열되어 있습니다.*

키	설명
WSRM_VERSION_PROPERTY	string WS-RM 버전 네임스페이스 (**http://schemas.xmlsoap.org/ws/2005/02/rm/** 또는 **http://docs.oasis-open.org/ws-rx/wsrm/200702**).
WSRM_WSA_VERSION_PROPERTY	문자열 WS-Addressing 버전 네임스페이스 (**http://schemas.xmlsoap.org/ws/2004/08/addressing** 또는 **http://www.w3.org/2005/08/addressing**) - **http://schemas.xmlsoap.org/ws/2005/02/rm/** RM 네임스페이스를 사용하지 않는 한 이 속성은 무시됩니다.
WSRM_LAST_MESSAGE_PROPERTY	WS-RM 코드에서 마지막 메시지가 전송되고 있음을 나타내기 위해 **true** 로 설정하면 코드가 WS-RM 시퀀스 및 릴리스 리소스를 종료할 수 있습니다(CXF의 3.0.0 버전인 경우 WS-RM은 클라이언트를 닫을 때 기본적으로 RM 시퀀스를 종료합니다.)
WSRM_INACTIVITY_TIMEOUT_PROPERTY	비활성 기간(밀리초)입니다.
WSRM_RETRANSMISSION_INTERVAL_PROPERTY	긴 기본 재전송 간격(밀리초)입니다.
WSRM_EXPONENTIAL_BACKOFF_PROPERTY	부울 지수 백오프 플래그입니다.

키	설명
WSRM_ACKNOWLEDGEMENT_INTERVAL_PROPERTY	장기 승인 간격(밀리초)입니다.

21.4.3. JMX를 통한 WS-RM 제어

Apache CXF의 JMX 관리 기능을 사용하여 WS-RM의 여러 측면을 모니터링하고 제어할 수도 있습니다. JMX 작업의 전체 목록은 org.apache.cxf.ws.ManagedRMManager 및 org.apache.cxf.ws.rm.ManagedRMEndpoint 에 의해 정의되지만 이러한 작업에는 현재 RM 상태를 개별 메시지 수준으로 보는 작업이 포함됩니다. JXM을 사용하여 WS-RM 시퀀스를 종료하거나 종료하고 이전에 전송된 메시지가 원격 RM 끝점에 의해 승인될 때 알림을 받을 수도 있습니다.

21.4.4. JMX 제어의 예

*예를 들어 클라이언트 구성에 **JMX** 서버가 활성화된 경우 다음 코드를 사용하여 수신된 마지막 승인 번호를 추적할 수 있습니다.*

```
// Java
private static class AcknowledgementListener implements NotificationListener {
    private volatile long lastAcknowledgement;

    @Override
    public void handleNotification(Notification notification, Object handback) {
        if (notification instanceof AcknowledgementNotification) {
            AcknowledgementNotification ack = (AcknowledgementNotification)notification;
            lastAcknowledgement = ack.getMessageNumber();
        }
    }

    // initialize client
...
    // attach to JMX bean for notifications
    //  NOTE: you must have sent at least one message to initialize RM before executing this code
    Endpoint ep = ClientProxy.getClient(client).getEndpoint();
    InstrumentationManager im = bus.getExtension(InstrumentationManager.class);
    MBeanServer mbs = im.getMBeanServer();
    RMManager clientManager = bus.getExtension(RMManager.class);
    ObjectName name = RMUtils.getManagedObjectName(clientManager, ep);
    System.out.println("Looking for endpoint name " + name);
    AcknowledgementListener listener = new AcknowledgementListener();
    mbs.addNotificationListener(name, listener, null, null);

    // send messages using RM with acknowledgement status reported to listener
...
```

21.5. WS-RM 구성

21.5.1. Apache CXF-Specific WS-RM 속성 구성

21.5.1.1. 개요

Apache CXF별 속성을 구성하려면 rmManager Spring 빈을 사용합니다. 설정 파일에 다음을 추가합니다.

- 네임스페이스 목록에 대한 **http://cxf.apache.org/ws/rm/manager** 네임스페이스입니다.
- 구성하려는 특정 속성에 대한 **rmManager Spring** 빈입니다.

예 21.4. "Apache CXF-Specific WS-RM 속성 구성" 간단한 예를 보여줍니다.

예 21.4. Apache CXF-Specific WS-RM 속성 구성

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager http://cxf.apache.org/schemas/configuration/wsrm-
manager.xsd"&gt;
...
&lt;wsrm-mgr:rmManager&gt;
&lt;!--
  ...Your configuration goes here
--&gt;
&lt;/wsrm-mgr:rmManager&gt;
```

21.5.1.2. rmManager Spring 빈의 하위 항목

표 21.2. "rmManager Spring Bean의 하위" http://cxf.apache.org/ws/rm/manager 네임스페이스에 정의된 **rmManager Spring** 빈의 하위 요소를 보여줍니다.

표 21.2. rmManager Spring Bean의 하위

요소	설명
RMAsertion	**RMAssertion**유형의 요소
deliveryAssurance	적용할 전달 보장을 설명하는 Type **DeliveryAssuranceType** 의 요소
sourcePolicy	RM 소스의 세부 정보를 구성할 수 있는 **SourcePolicyType** 유형의 요소
destinationPolicy	RM 대상에 대한 세부 정보를 구성할 수 있는 **DestinationPolicyType** 유형의 요소

21.5.1.3. 예제

예를 들어 "확인되지 않은 최대 메시지 임계값" 에서 참조하십시오.

21.5.2. 표준 WS-RM 정책 속성 구성

21.5.2.1. 개요

다음 방법 중 하나로 표준 WS-RM 정책 속성을 구성할 수 있습니다.

- *"rmManager Spring bean의 RMAsertion"*
- *"기능 내에서 정책"*
- *"WSDL 파일"*
- *"외부 첨부 파일"*

21.5.2.2. WS-Policy RMAsertion Children

표 21.3. "WS-Policy RMAsertion Element의 하위 항목" http://schemas.xmlsoap.org/ws/2005/02/rm/policy 네임스페이스에 정의된 요소를 표시합니다.

표 21.3. WS-Policy RMAsertion Element의 하위 항목

이름	설명
InactivityTimeout	비활성으로 인해 RM 시퀀스를 종료하기 전에 메시지를 수신하지 않고 전달해야 하는 시간을 지정합니다.
BaseRetransmissionInterval	지정된 메시지에 대해 RM Source가 승인을 받아야 하는 간격을 설정합니다. **BaseRetransmissionInterval** 에 의해 설정된 시간 내에 승인이 수신되지 않으면 RM Source가 메시지를 다시 전송합니다.
ExponentialBackoff	일반적으로 알려진 지수 백오프 알고리즘(Tanenbaum)을 사용하여 재전송 간격이 조정됨을 나타냅니다. 자세한 내용은 **Computer Networks**, Andrew S. Tanenbaum, Prentice Hall PTR, 2003을 참조하십시오.
AcknowledgementInterval	WS-RM에서 승인은 반환 메시지로 전송되거나 독립형으로 전송됩니다. 반환 메시지를 사용하여 승인을 보낼 수 없는 경우 RM Destination은 독립형 승인을 보내기 전에 승인 간격까지 기다릴 수 있습니다. 확인되지 않은 메시지가 없으면 RM Destination은 승인 메시지를 보내지 않도록 선택할 수 있습니다.

21.5.2.3. 자세한 참조 정보

각 요소의 하위 요소 및 특성에 대한 설명을 포함한 자세한 참조 정보는 http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd 에서 참조하십시오.

21.5.2.4. rmManager Spring bean의 RMAsertion

Apache CXF rmManager Spring bean에 RMAssertion 을 추가하여 표준 WS-RM 정책 속성을 구성할 수 있습니다. 동일한 구성 파일에 모든 WS-RM 구성을 유지하려면 이 방법이 가장 좋습니다. 즉, 동일한 파일에서 Apache CXF 특정 속성과 표준 WS-RM 정책 속성을 구성하려면 가장 좋은 방법입니다.

예를 들어 예 21.5. "rmManager Spring Bean에서 RMAssertion을 사용하여 WS-RM 속성 구성" 의 구성은 다음과 같습니다.

- ***rmManager Spring 빈 내에서 RMAssertion 을 사용하여 구성된 표준 WS-RM 정책 속성 BaseRetransmissionInterval.***

- *동일한 구성 파일에 구성된 **Apache CXF**별 **RM** 속성인 **intraMessageThreshold.***

예 21.5. rmManager Spring Bean에서 RMAssertion을 사용하여 WS-RM 속성 구성

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
      xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
   <wsrm-policy:RMAssertion>
      <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
   </wsrm-policy:RMAssertion>
   <wsrm-mgr:destinationPolicy>
      <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
   </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

21.5.2.5. 기능 내에서 정책

*예 21.6. “WS-RM 특성을 기능 내에서 정책으로 구성” 과 같이 기능 내에서 표준 **WS-RM** 정책 특성을 구성할 수 있습니다.*

예 21.6. WS-RM 특성을 기능 내에서 정책으로 구성

```
<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:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-policy.xsd
http://cxf.apache.org/ws/addressing http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <jaxws:endpoint name="{http://cxf.apache.org/greeter_control}GreeterPort"
createdFromAPI="true">
       <jaxws:features>
           <wsp:Policy>
              <wsrm:RMAssertion
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                <wsrm:AcknowledgementInterval Milliseconds="200" />
              </wsrm:RMAssertion>
              <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                 <wsp:Policy>
                    <wsam:NonAnonymousResponses/>
                 </wsp:Policy>
```

```
            </wsam:Addressing>
        </wsp:Policy>
    </jaxws:features>
  </jaxws:endpoint>
</beans>
```

21.5.2.6. WSDL 파일

WS-Policy 프레임워크를 사용하여 **WS-RM**을 활성화하는 경우 **WSDL** 파일에서 표준 **WS-RM** 정책 속성을 구성할 수 있습니다. 이 방법은 서비스에서 기타 정책 인식 웹 서비스 스택에 배포된 소비자와 함께 **WS-RM**을 원활하게 사용하고자 하는 경우에 적합합니다.

예를 들어, "WS-Policy 프레임워크: 암시적으로 인터셉터 추가" 기본 재전송 간격이 **WSDL** 파일에 구성된 위치를 참조하십시오.

21.5.2.7. 외부 첨부 파일

외부 연결 파일에서 표준 **WS-RM** 정책 속성을 구성할 수 있습니다. 이 방법은 **WSDL** 파일을 변경할 수 없거나 원하지 않는 경우 좋은 방법입니다.

예 21.7. "외부 연결에서 WS-RM 구성" 특정 **EPR**에 대해 **WS-A** 및 **WS-RM**(기본 재전송 간격 **30**초)을 둘 다 활성화하는 외부 연결을 보여줍니다.

예 21.7. 외부 연결에서 WS-RM 구성

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <wsp:PolicyAttachment>
    <wsp:AppliesTo>
      <wsa:EndpointReference>
          <wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
       </wsa:EndpointReference>
    </wsp:AppliesTo>
    <wsp:Policy>
      <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
      <wsrmp:RMAssertion xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="30000"/>
      </wsrmp:RMAssertion>
    </wsp:Policy>
  </wsp:PolicyAttachment>
</attachments>/
```

21.5.3. WS-RM 구성 사용 사례

21.5.3.1. 개요

이 하위 섹션에서는 사용 사례 관점에서 **WS-RM** 속성을 구성하는 데 중점을 둡니다. 속성이 **http://schemas.xmlsoap.org/ws/2005/02/rm/policy/** 네임스페이스에 정의된 표준 **WS-RM** 정책 특성인 경우 **rmManager Spring** 빈 내의 **RMAssertion** 에서 설정하는 예제만 표시됩니다. 기능 내에서 정책과 같은 속성을 설정하는 방법에 대한 자세한 내용은 **WSDL** 파일 또는 외부 연결에서 참조하십시오. **21.5.2** 절. "표준 **WS-RM** 정책 속성 구성"

다음 사용 사례에 대해 설명합니다.

- "기본 재전송 간격"
- "다시 전송 시 지수 백오프"
- "승인 간격"
- "확인되지 않은 최대 메시지 임계값"
- "**RM** 시퀀스의 최대 길이"
- "메시지 전달 보장 정책"

21.5.3.2. 기본 재전송 간격

BaseRetransmissionInterval 요소는 **RM** 소스가 아직 확인되지 않은 메시지를 다시 전송하는 간격을 지정합니다. **http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd** 스키마 파일에 정의되어 있습니다. 기본값은 **3000**밀리초입니다.

예 **21.8.** "**WS-RM** 기본 재전송 간격 설정" **WS-RM** 기본 재전송 간격을 설정하는 방법을 보여줍니다.

예 21.8. WS-RM 기본 재전송 간격 설정

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
```

```
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

21.5.3.3. 다시 전송 시 지수 백오프

ExponentialBackoff 요소는 검증되지 않은 메시지에 대한 연속 재전송 시도가 지수 간격으로 수행되는지 여부를 결정합니다.

ExponentialBackoff 요소의 존재로 인해 이 기능을 사용할 수 있습니다. 기본적으로 **2** 의 지수 백오프 비율이 사용됩니다. **ExponentialBackoff** 는 플래그입니다. 요소가 있으면 **exponential** 백오프가 활성화됩니다. 요소가 없으면 **exponential** 백오프가 비활성화됩니다. 값이 필요하지 않습니다.

예 21.9. "WS-RM Exponential Backoff 속성 설정" 는 재전송을 위해 **WS-RM exponential** 백오프를 설정하는 방법을 보여줍니다.

예 21.9. WS-RM Exponential Backoff 속성 설정

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

21.5.3.4. 승인 간격

AcknowledgementInterval 요소는 **WS-RM** 대상이 비동기 승인을 보내는 간격을 지정합니다. 이는 수신 메시지의 수신 시 보내는 동기 확인에도 추가됩니다. 기본 비동기 승인 간격은 **0** 밀리초입니다. 즉, **AcknowledgementInterval** 이 특정 값으로 구성되지 않은 경우 승인은 즉시 전송됩니다(즉, 첫 번째 사용 가능한 기회에서).

비동기 승인은 다음 두 조건이 모두 충족되는 경우에만 **RM** 대상에 의해 전송됩니다.

- *RM 대상은 비익명 wsrm:acksTo 끝점을 사용합니다.*

- *승인 간격이 만료되기 전에 응답 메시지에 대한 승인이 발생하지 않습니다.*

예 21.10. "WS-RM 인식 간격 설정" 는 *WS-RM* 승인 간격을 설정하는 방법을 보여줍니다.

예 21.10. WS-RM 인식 간격 설정

```
<beans xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

21.5.3.5. 확인되지 않은 최대 메시지 임계값

maxUnacknowledged 특성은 시퀀스가 종료되기 전에 시퀀스당 누적할 수 있는 최대 메시지 수를 설정합니다.

예 21.11. "WS-RM Maximum Unackledged Message Threshold 설정" *WS-RM* 최대 확인되지 않은 메시지 임계값을 설정하는 방법을 보여줍니다.

예 21.11. WS-RM Maximum Unackledged Message Threshold 설정

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

21.5.3.6. RM 시퀀스의 최대 길이

maxLength 속성은 *WS-RM* 시퀀스의 최대 길이를 설정합니다. 기본값은 *0* 이며, 이는 *WS-RM* 시퀀스의 길이가 바인딩되지 않음을 의미합니다.

이 속성이 설정되면 **RM** 끝점이 제한에 도달하면 새 **RM** 시퀀스를 생성하고 이전에 보낸 메시지에 대한 모든 승인을 받습니다. 새 메시지는 **newsequence**를 사용하여 전송됩니다.

예 21.12. "WS-RM 메시지 시퀀스의 최대 길이 설정" **RM** 시퀀스의 최대 길이를 설정하는 방법을 보여줍니다.

예 21.12. WS-RM 메시지 시퀀스의 최대 길이 설정

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
   <wsrm-mgr:sourcePolicy>
      <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
   </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

21.5.3.7. 메시지 전달 보장 정책

다음과 같은 전달 보장 정책을 사용하도록 **RM** 대상을 구성할 수 있습니다.

- **At mostOnce - RM** 대상은 한 번만 애플리케이션 목적지로 메시지를 전달합니다. 메시지가 두 번 이상 발생하는 경우 오류가 발생합니다. 시퀀스의 일부 메시지가 전달되지 않을 수 있습니다.
- **AtLeastOnce - RM** 대상은 적어도 한 번 이상 애플리케이션 대상으로 메시지를 전달합니다. 전송된 모든 메시지가 전송되거나 오류가 발생합니다. 일부 메시지는 한 번 이상 배달될 수 있습니다.
- **InOrder - RM** 대상은 전송된 순서대로 애플리케이션 대상에 메시지를 전달합니다. 이 전달 보장은 **At most Once** 또는 **AtLeastOnce** 보증과 결합할 수 있습니다.

예 21.13. "WS-RM 메시지 전달 보장 정책 설정" 는 **WS-RM** 메시지 전달 보증을 설정하는 방법을 보여줍니다.

예 21.13. WS-RM 메시지 전달 보장 정책 설정

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
```

```
    <wsrm-mgr:deliveryAssurance>
      <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
  </wsrm-mgr:reliableMessaging>
  </beans>
```

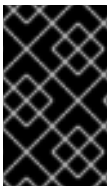
21.6. WS-RM 지속성 구성

21.6.1. 개요

이 장에 이미 설명된 **Apache CXF WS-RM** 기능은 네트워크 실패와 같은 경우에 대한 신뢰성을 제공합니다. **WS-RM** 지속성은 **RM** 소스 또는 **RM** 대상 충돌과 같은 다른 유형의 실패에 걸쳐 안정성을 제공합니다.

WS-RM 지속성에는 다양한 **RM** 엔드포인트의 상태를 영구 스토리지에 저장하는 작업이 포함됩니다. 이를 통해 엔드포인트가 다시 시작될 때 계속 메시지를 보내고 받을 수 있습니다.

Apache CXF는 구성 파일에서 **WS-RM** 지속성을 활성화합니다. 기본 **WS-RM** 지속성 저장소는 **JDBC** 기반입니다. 편의를 위해 **Apache CXF**에는 기본 제공 배포를 위한 **Derby**가 포함되어 있습니다. 또한 **Java API**를 사용하여 영구 저장소도 노출됩니다. 자체 지속성 메커니즘을 구현하기 위해 기본 **DB**와 함께 이 **API**를 사용하여 하나를 구현할 수 있습니다.



중요

WS-RM 지속성은 **oneway** 호출에만 지원되며 기본적으로 비활성화되어 있습니다.

21.6.2. 작동 방식

Apache CXF WS-RM 지속성은 다음과 같이 작동합니다.

- **RM** 소스 끝점에서 전송 전에 발신 메시지가 유지됩니다. 승인이 수신된 후 영구 저장소에서 제거됩니다.
- 충돌에서 복구한 후 모든 메시지가 승인될 때까지 지속된 메시지를 복구하고 다시 전송합니다. 이 시점에서 **RM** 시퀀스가 닫힙니다.
- **RM** 대상 끝점에서 들어오는 메시지가 유지되고 성공적으로 저장되면 승인이 전송됩니다. 메시지를 성공적으로 디스패치하면 영구 저장소에서 제거됩니다.

- 충돌에서 복구한 후 지속적인 메시지를 복구하고 디스패치합니다. 또한 새 메시지가 수락, 승인 및 전달되는 상태로 RM 시퀀스를 제공합니다.

21.6.3. WS-persistence 활성화

WS-RM 지속성을 활성화하려면 WS-RM용 영구 저장소를 구현하는 오브젝트를 지정해야 합니다. 직접 개발하거나 Apache CXF와 함께 제공되는 JDBC 기반 저장소를 사용할 수 있습니다.

예 21.14. "기본 WS-RM Persistence 저장소에 대한 구성" 에 표시된 구성은 Apache CXF와 함께 제공되는 JDBC 기반 저장소를 활성화합니다.

예 21.14. 기본 WS-RM Persistence 저장소에 대한 구성

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

21.6.4. WS-persistence 구성

Apache CXF와 함께 제공되는 JDBC 기반 저장소는 표 21.4. "JDBC 저장소 속성" 에 표시된 속성을 지원합니다.

표 21.4. JDBC 저장소 속성

특성 이름	유형	기본 설정
driverClassName	문자열	**org.apache.derby.jdbc.EmbeddedDriver**
userName	문자열	null
passWord	문자열	null
url	문자열	jdbc:derby:rmdb;create=true

예 21.15. "WS-RM Persistence용 JDBC 저장소 구성" 에 표시된 설정을 사용하면 Apache CXF와 함께 제공되는 JDBC 기반 저장소를 드라이버 ClassName 및 url 을 기본값이 아닌 값으로 설정할 수 있습니다.

예 21.15. WS-RM Persistence용 JDBC 저장소 구성

```
<bean id="RMTxStore" class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

22장. 고가용성 활성화

초록

이 장에서는 Apache CXF 런타임에서 고가용성을 활성화하고 구성하는 방법을 설명합니다.

22.1. 고가용성 소개

22.1.1. 개요

확장 가능하고 안정적인 애플리케이션은 분산 시스템에서 단일 장애 지점을 방지하기 위해 고가용성이 필요합니다. 를 사용하여 시스템을 단일 장애 지점으로부터 보호할 수 있습니다. 복제된 서비스.

복제된 서비스는 동일한 서비스의 여러 인스턴스 또는 복제본 으로 구성됩니다. 이러한 기능을 함께 단일 논리 서비스로 작동합니다. 클라이언트는 복제된 서비스에서 요청을 호출하고 Apache CXF는 멤버 복제본 중 하나에 요청을 제공합니다. 복제본 라우팅은 클라이언트에 투명합니다.

22.1.2. 고정 페일오버가 있는 HA

Apache CXF는 복제본 세부 정보가 service WSDL 파일에 인코딩되는 정적 페일오버와 함께 HA(고가용성)를 지원합니다. WSDL 파일에는 여러 포트가 포함되어 있으며 동일한 서비스에 대해 여러 호스트를 포함할 수 있습니다. WSDL 파일이 변경되지 않는 한 클러스터의 복제본 수는 정적으로 유지됩니다. 클러스터 크기를 변경하려면 WSDL 파일을 편집해야 합니다.

22.2. 고정 장애 조치(STIC FAILOVER)를 사용하여 HA 활성화

22.2.1. 개요

정적 페일오버를 사용하여 HA를 활성화하려면 다음을 수행해야 합니다.

1. “서비스 WSDL 파일에서 복제본 세부 정보 인코딩”
2. “클라이언트 구성에 클러스터링 기능 추가”

22.2.2. 서비스 WSDL 파일에서 복제본 세부 정보 인코딩

서비스 **WSDL** 파일에서 클러스터의 복제본 세부 정보를 인코딩해야 합니다. 예 22.1. "고정 장애 조치 (Failover)를 사용하여 HA 활성화: WSDL 파일" 는 세 개의 복제본의 서비스 클러스터를 정의하는 **WSDL** 파일 추출을 보여줍니다.

예 22.1. 고정 장애 조치 (Failover)를 사용하여 HA 활성화: WSDL 파일

```
<wsdl:service name="ClusteredService">
   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica1">
      <soap:address location="http://localhost:9001/SoapContext/Replica1"/>
   </wsdl:port>

   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica2">
      <soap:address location="http://localhost:9002/SoapContext/Replica2"/>
   </wsdl:port>

   <wsdl:port binding="tns:Greeter_SOAPBinding" name="Replica3">
      <soap:address location="http://localhost:9003/SoapContext/Replica3"/>
   </wsdl:port>

</wsdl:service>
```

예 22.1. "고정 장애 조치 (Failover)를 사용하여 HA 활성화: WSDL 파일" 에 표시된 **WSDL** 추출은 다음과 같이 설명할 수 있습니다.

세 개의 포트에 노출되는 서비스 **ClusterService** 를 정의합니다.

1. **Replica1**
2. **Replica2**
3. **Replica3**

포트 **9001** 에서 **HTTP** 끝점을 통해 **SOAP**로 **ClusterService** 를 노출하도록 **Replica1** 을 정의합니다.

포트 **9002** 에서 **ClusterService** 를 **HTTP** 끝점을 통해 **SOAP**로 노출하도록 **Replica2** 를 정의합니다.

포트 **9003** 에서 **HTTP** 끝점을 통해 **SOAP**로 **ClusterService** 를 노출하도록 **Replica3** 를 정의합니다.

22.2.3. 클라이언트 구성에 클러스터링 기능 추가

클라이언트 구성 파일에서 예 22.2. "정적 장애 조치(Failover)를 사용하여 HA 활성화: 클라이언트 구성" 과 같이 클러스터링 기능을 추가합니다.

예 22.2. 정적 장애 조치(Failover)를 사용하여 HA 활성화: 클라이언트 구성

```
<?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:jaxws="http://cxf.apache.org/jaxws"
      xmlns:clustering="http://cxf.apache.org/clustering"
        xsi:schemaLocation="http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica1"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica2"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

    <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
                  createdFromAPI="true">
        <jaxws:features>
            <clustering:failover/>
        </jaxws:features>
    </jaxws:client>

</beans>
```

22.3. 정적 장애 조치(FAILOVER)를 사용하여 HA 구성

22.3.1. 개요

기본적으로 정적 페일오버를 사용하는 HA는 클라이언트가 통신할 원본 서비스를 사용할 수 없거나 실패하는 경우 복제본 서비스를 선택할 때 순차적 전략을 사용합니다. 순차적 전략은 사용할 때마다 동일한 순차순으로 복제본 서비스를 선택합니다. 선택 사항은 Apache CXF의 내부 서비스 모델에 따라 결정되며, 그 결과 결정적인 페일오버 패턴을 제공합니다.

22.3.2. 임의의 전략 구성

복제본을 선택할 때 순차적 전략 대신 임의의 전략을 사용하도록 정적 장애 조치로 **HA**를 구성할 수 있습니다. **random** 전략은 서비스를 사용할 수 없게 되거나 실패할 때마다 임의의 복제본 서비스를 선택합니다. 클러스터의 생존 멤버에서 페일오버 대상을 선택하는 것은 완전히 무작위로 선택됩니다.

임의의 전략을 구성하려면 예 22.3. "정적 장애 조치(Failover)를 위한 Random 전략 구성" 에 표시된 구성을 클라이언트 구성 파일에 추가합니다.

예 22.3. 정적 장애 조치(Failover)를 위한 Random 전략 구성

```
<beans ...>
  <bean id="Random" class="org.apache.cxf.clustering.RandomStrategy"/>

  <jaxws:client name="{http://apache.org/hello_world_soap_http}Replica3"
                createdFromAPI="true">
    <jaxws:features>
      <clustering:failover>
        <clustering:strategy>
          <ref bean="Random"/>
        </clustering:strategy>
      </clustering:failover>
    </jaxws:features>
  </jaxws:client>
</beans>
```

예 22.3. "정적 장애 조치(Failover)를 위한 Random 전략 구성" 에 표시된 구성은 다음과 같이 설명할 수 있습니다.

임의의 전략을 구현하는 **Random** 빈 및 구현 클래스를 정의합니다.

복제본을 선택할 때 임의의 전략이 사용되도록 지정합니다.

23장. APACHE CXF 바인딩 ID

23.1. 바인딩 ID 테이블

표 23.1. Message Bindings의 바인딩 ID

바인딩	ID
CORBA	http://cxf.apache.org/bindings/corba
HTTP/REST	http://apache.org/cxf/binding/http
SOAP 1.1	http://schemas.xmlsoap.org/wsdl/soap/http
SOAP 1.1 w/ MTOM	http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true
SOAP 1.2	http://www.w3.org/2003/05/soap/bindings/HTTP/
SOAP 1.2 w/ MTOM	http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true
XML	http://cxf.apache.org/bindings/xformat

부록 A. MAVEN OSGI 툴 사용

초록

대규모 프로젝트의 번들 또는 번들 컬렉션을 수동으로 생성하는 것은 번거로울 수 있습니다. Maven 번들 플러그인을 사용하면 프로세스를 자동화하고 번들 매니페스트의 콘텐츠를 지정하기 위한 여러 바로 가기를 제공하여 작업을 더 쉽게 수행할 수 있습니다.

A.1. MAVEN BUNDLE 플러그인

Red Hat Fuse OSGi 툴링은 Apache Felix의 Maven 번들 플러그인 을 사용합니다. bundle 플러그인은 Peter Kriens의 B nd 툴을 기반으로 합니다. 번들에 패키지되는 클래스의 콘텐츠를 검사하여 OSGi 번들 매니페스트의 구성을 자동화합니다. 번들에 포함된 클래스에 대한 지식을 사용하여 플러그인은 번들 매니페스트에서 Import-Packages 및 Export-Package 속성을 채우기 위해 적절한 값을 계산할 수 있습니다. 플러그인에는 번들 매니페스트의 다른 필수 속성에 사용되는 기본값도 있습니다.

bundle 플러그인을 사용하려면 다음을 수행합니다.

1. A.2절. “Red Hat Fuse OSGi 프로젝트 설정” 프로젝트의 POM 파일에 대한 번들 플러그인입니다.
2. A.3절. “번들 플러그인 구성” 번들 매니페스트를 올바르게 채우는 플러그인입니다.

A.2. RED HAT FUSE OSGI 프로젝트 설정

A.2.1. 개요

OSGi 번들을 빌드하기 위한 Maven 프로젝트는 간단한 단일 수준 프로젝트일 수 있습니다. 하위 프로젝트는 필요하지 않습니다. 그러나 다음을 수행해야 합니다.However, it does that you do the following:

1. POM에 bundle 플러그인을 추가 합니다.
2. Maven에서 결과를 OSGi 번들로 패키징하도록 지시 합니다.

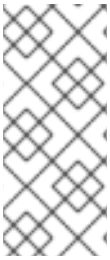
참고

적절한 설정으로 프로젝트를 설정하는 데 사용할 수 있는 여러 **Maven archetypes**이 있습니다.

A.2.2. 디렉토리 구조

OSGi 번들을 구성하는 프로젝트는 단일 수준 프로젝트일 수 있습니다. 최상위 **POM** 파일과 **src** 폴더가 있어야만 합니다. 모든 **Maven** 프로젝트에서와 마찬가지로 **src/java** 폴더에 모든 **Java** 소스 코드를 배치하고 **src/resources** 폴더에 **Java**가 아닌 리소스를 배치합니다.

Java가 아닌 리소스에는 **Spring** 구성 파일, **JBI** 엔드포인트 구성 파일, **WSDL** 계약이 포함됩니다.



참고

Apache CXF, Apache Camel 또는 다른 **Spring** 구성된 빈을 사용하는 **Red Hat Fuse OSGi** 프로젝트에는 **src/resources/META-INF/spring** 폴더에 있는 **beans.xml** 파일도 포함되어 있습니다.

A.2.3. 번들 플러그인 추가

번들 플러그인을 사용하려면 먼저 **Apache Felix**에 종속 항목을 추가해야 합니다. 종속성을 추가한 후 **POM**의 플러그인 부분에 **bundle** 플러그인을 추가할 수 있습니다.

예 A.1. “POM에 OSGi 번들 플러그인 추가” 프로젝트에 번들 플러그인을 추가하는 데 필요한 **POM** 항목을 표시합니다.

예 A.1. POM에 OSGi 번들 플러그인 추가

```
...
<dependencies>
  <dependency>
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin>
```

```
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
          <Import-Package>*,org.apache.camel.osgi</Import-Package>
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

예 A.1. "POM에 OSGi 번들 플러그인 추가" 의 항목은 다음을 수행합니다.

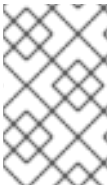
***Apache Felix**에 대한 종속성 추가*

*프로젝트에 **bundle** 플러그인 추가*

*프로젝트의 아티팩트 **ID**를 번들의 기호 이름으로 사용하도록 플러그인을 구성합니다.*

*번들 클래스에서 가져온 모든 **Java** 패키지를 포함하도록 플러그인을 구성합니다. 또한 **org.apache.camel.osgi** 패키지도 가져옵니다.*

나열된 클래스를 번들로 플러그인을 구성하지만 내보낸 패키지 목록에 포함하지 않도록 구성합니다.



참고

프로젝트의 요구 사항을 충족하도록 구성을 편집합니다.

*번들 플러그인 구성에 대한 자세한 내용은 **A.3**절. "번들 플러그인 구성" 을 참조하십시오.*

A.2.4. 번들 플러그인 활성화

***Maven**이 **bundle** 플러그인을 사용하도록 하려면 프로젝트의 결과를 번들로 패키징하도록 지시합니다. **POM** 파일의 **packaging** 요소를 번들로 설정하여 이 작업을 수행합니다.*

A.2.5. 유용한 Maven archetypes

bundle 플러그인을 사용하도록 사전 구성된 프로젝트를 생성하는 데 사용할 수 있는 여러 **Maven archetypes**가 있습니다.

- "Spring OSGi archetype"
- "Apache CXF 코드 우선 archetype"
- "Apache CXF wsdl-first archetype"
- "Apache Camel archetype"

A.2.6. Spring OSGi archetype

Spring OSGi archetype은 다음과 같이 **Spring DM**을 사용하여 **OSGi** 프로젝트를 빌드하기 위한 일반 프로젝트를 생성합니다.

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

다음 명령을 사용하여 **archetype**을 호출합니다.

```
mvn archetype:generate -DarchetypeGroupId=org.springframework.osgi -
DarchetypeArtifactId=spring-osgi-bundle-archetype -DarchetypeVersion=1.1.2 -DgroupId=groupId -
DartifactId=artifactId -Dversion=version
```

A.2.7. Apache CXF 코드 우선 archetype

Apache CXF 코드 우선 **archetype**은 다음과 같이 **Java**에서 서비스를 빌드하는 프로젝트를 생성합니다.

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2010.02.0-fuse-02-00
```

다음 명령을 사용하여 **archetype**을 호출합니다.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-cxf-code-first-archetype -DarchetypeVersion=2010.02.0-fuse-
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

A.2.8. Apache CXF wsdl-first archetype

Apache CXF wsdl-first archetype은 다음과 같이 **WSDL**에서 서비스를 생성하는 프로젝트를 생성합니다.

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2010.02.0-fuse-02-00
```

다음 명령을 사용하여 **archetype**을 호출합니다.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype -DarchetypeVersion=2010.02.0-fuse-
02-00 -DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

A.2.9. Apache Camel archetype

Apache Camel archetype은 다음과 같이 **Red Hat Fuse**에 배포된 경로를 구축하기 위한 프로젝트를 생성합니다.

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2010.02.0-fuse-02-00
```

다음 명령을 사용하여 **archetype**을 호출합니다.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.servicemix.tooling -
DarchetypeArtifactId=servicemix-osgi-camel-archetype -DarchetypeVersion=2010.02.0-fuse-02-00 -
DgroupId=groupId -DartifactId=artifactId -Dversion=version
```

A.3. 번들 플러그인 구성

A.3.1. 개요

번들 플러그인에는 기능하기 위해 정보가 거의 필요하지 않습니다. 모든 필수 속성은 기본 설정을 사용하여 유효한 **OSGi** 번들을 생성합니다.

기본값만 사용하여 유효한 번들을 생성할 수 있지만 일부 값을 수정할 수도 있습니다. 플러그인의 **instructions** 요소 내부에서 대부분의 속성을 지정할 수 있습니다.

A.3.2. 구성 속성

일반적으로 사용되는 구성 속성 중 일부는 다음과 같습니다.

- **Bundle-SymbolicName**
- **bundle-Name**
- **bundle-Version**
- **export-Package**
- **private-Package**
- **import-Package**

A.3.3. 번들의 심볼릭 이름 설정

기본적으로 **bundle** 플러그인은 **Bundle-SymbolicName** 속성 값을 **groupId + "." + artifactId** 로 설정합니다.

- **groupId** 에 하나의 섹션만 있는 경우 클래스가 있는 첫 번째 패키지 이름이 반환됩니다.

  예를 들어 **Id** 그룹이 **commons-logging:commons-logging** 이면 번들의 심볼릭 이름은 **org.apache.commons.logging** 입니다.

- **artifactId** 가 **groupId** 의 마지막 섹션과 같은 경우 **groupId** 가 사용됩니다.

  예를 들어 **POM**에서 그룹 **ID**와 아티팩트 **ID**를 **org.apache.maven:maven** 로 지정하는 경우 번들의 심볼릭 이름은 **org.apache.maven** 입니다.

- ***artifactId*** *가* ***groupId*** *의 마지막 섹션으로 시작되면 해당 부분이 제거됩니다.*

  *예를 들어 **POM**에서 그룹 **ID**와 아티팩트 **ID**를* ***org.apache.maven:maven-core*** *로 지정하는 경우 번들의 심볼릭 이름은* ***org.apache.maven.core*** *입니다.*

번들의 심볼릭 이름에 대한 자체 값을 지정하려면 예 A.2. "번들의 심볼릭 이름 설정" 과 같이 플러그인의 ***instructions*** *요소에* ***Bundle-SymbolicName*** *하위 항목을 추가합니다.*

예 ***A.2.*** *번들의 심볼릭 이름 설정*

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

A.3.4. 번들 이름 설정

기본적으로 번들 이름은 ***${project.name}*** *으로 설정됩니다.*

번들 이름에 대한 자체 값을 지정하려면 예 A.3. "번들 이름 설정" 과 같이 플러그인의 ***instructions*** *요소에* ***Bundle-Name*** *하위를 추가합니다.*

예 ***A.3.*** *번들 이름 설정*

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

A.3.5. 번들 버전 설정

기본적으로 번들 버전은 **${project.version}** 으로 설정됩니다. 모든 대시(-)는 점(.)으로 교체되고 숫자는 최대 4자리까지 채워집니다. 예를 들어 **4.2-SNAPSHOT** 는 **4.2.0.SNAPSHOT** 가 됩니다.

번들 버전에 대한 자체 값을 지정하려면 예 A.4. “번들 버전 설정” 에서와 같이 번들 버전 하위를 플러그인의 **instructions** 요소에 추가합니다.

예 A.4. 번들 버전 설정

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

A.3.6. 내보낸 패키지 지정

기본적으로 **OSGi** 매니페스트의 **Export-Package** 목록은 기본 패키지, **..** 및 **.internal** 이 포함된 패키지를 제외하고 로컬 **Java** 소스 코드의 모든 패키지로 채워집니다.

중요

플러그인 구성에서 **Private-Package** 요소를 사용하고 내보낼 패키지 목록을 지정하지 않으면 기본 동작은 번들의 **Private-Package** 요소에 나열된 패키지만 포함합니다. 내보낸 패키지가 없습니다.

기본 동작은 매우 큰 패키지로 발생하며 비공개로 유지해야 하는 패키지를 내보낼 수 있습니다. 내보낸 패키지 목록을 변경하려면 플러그인의 **instructions** 요소에 **Export-Package** 하위를 추가할 수 있습니다.

Export-Package 요소는 번들에 포함할 패키지 목록을 지정하고 내보낼 패키지 목록을 지정합니다. 패키지 이름은 * 와일드카드 기호를 사용하여 지정할 수 있습니다. 예를 들어 **com.fuse.demo.*** 항목에 **com.fuse.demo.***로 시작하는 프로젝트의 **classpath**의 모든 패키지가 포함되어 있습니다.

항목을 접두사로 지정할 패키지를 지정할 수 있습니다 **!**. 예를 들어 **!com.fuse.demo.private** 항목은

com.fuse.demo.private 패키지를 제외합니다.

패키지를 제외할 때 목록의 항목 순서가 중요합니다. 목록은 처음부터 순서대로 처리되며 이후의 충돌 항목은 무시됩니다.

예를 들어 **com.fuse.demo.private** 패키지를 제외한 **com.fuse.demo.demo**로 시작하는 모든 패키지를 포함하려면 다음을 사용하여 패키지를 나열하십시오.

```
!com.fuse.demo.private,com.fuse.demo.*
```

그러나 **com.fuse.demo.*,!com.fuse.demo.private** 를 사용하여 패키지를 나열하는 경우 첫 번째 패턴과 일치하므로 번들에 **com.fuse.demo.private** 이 포함됩니다.

A.3.7. 개인 패키지 지정

내보내 지 않고 번들에 포함할 패키지 목록을 지정하려면 번들 플러그인 구성에 **Private-Package** 명령을 추가할 수 있습니다. 기본적으로 **Private-Package** 명령을 지정하지 않으면 로컬 **Java** 소스의 모든 패키지가 번들에 포함됩니다.

중요

패키지가 **Private-Package** 요소 및 **Export-Package** 요소 둘 다의 항목과 일치하는 경우 **Export-Package** 요소가 우선합니다. 패키지가 번들에 추가되고 내보내집니다.

Private-Package 요소는 번들에 포함할 패키지 목록을 지정하는 데 사용되는 **Export-Package** 요소와 유사하게 작동합니다. **bundle** 플러그인은 목록을 사용하여 번들에 포함할 프로젝트의 **classpath**에서 모든 클래스를 찾습니다. 이러한 패키지는 번들에 패키지되지만 내보내지지 않습니다(**Export-Package** 명령어에서도 선택하지 않는 한).

예 A.5. “번들에 개인 패키지 포함” 번들에 개인 패키지를 포함하는 구성을 보여줍니다.

예 A.5. 번들에 개인 패키지 포함

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
```

```
    ...
   </instructions>
  </configuration>
</plugin>
```

A.3.8. 가져온 패키지 지정

기본적으로 번들 플러그인은 **OSGi** 매니페스트의 **Import-Package** 속성을 번들 콘텐츠로 참조하는 모든 패키지 목록으로 채웁니다.

기본 동작은 일반적으로 대부분의 프로젝트에 충분하지만 목록에 자동으로 추가되지 않는 패키지를 가져오려는 인스턴스를 찾을 수 있습니다. 기본 동작으로 인해 원치 않는 패키지를 가져올 수도 있습니다.

번들에서 가져올 패키지 목록을 지정하려면 플러그인의 **instructions** 요소에 **Import-Package** 하위 항목을 추가합니다. 패키지 목록의 구문은 **Export-Package** 요소 및 **Private-Package** 요소에 대해 동일합니다.

중요

Import-Package 요소를 사용하는 경우 플러그인은 번들의 콘텐츠를 자동으로 검사하여 필요한 가져오기가 있는지 여부를 확인하지 않습니다. 번들의 콘텐츠를 스캔하려면 패키지 목록에 * 를 마지막 항목으로 배치해야 합니다.

예 A.6. “번들에서 가져온 패키지 지정” 번들에서 가져온 패키지를 지정하기 위한 구성을 표시합니다.

예 A.6. 번들에서 가져온 패키지 지정

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws, javax.wsdl, org.apache.cxf.bus, org.apache.cxf.bus.spring,
org.apache.cxf.bus.resource, org.apache.cxf.configuration.spring, org.apache.cxf.resource,
org.springframework.beans.factory.config, * </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

A.3.9. 더 알아보기

번들 플러그인 구성에 대한 자세한 내용은 다음을 참조하십시오.

- *olink:OsgiDependencies/OsgiDependencies*
- *Apache Felix 문서*
- *Peter Kriens의 AQute Software Consultancy 웹 사이트*

V 부. JAX-WS를 사용하여 애플리케이션 개발

*이 가이드에서는 표준 **JAX-WS API**를 사용하여 웹 서비스를 개발하는 방법을 설명합니다.*

24장. 하이 업 서비스 개발

초록

Java 코드가 이미 서비스 지향 애플리케이션의 일부로 노출하려는 기능 집합을 구현하는 많은 인스턴스가 있습니다. 인터페이스를 정의하기 위해 WSDL을 사용하지 않도록 할 수도 있습니다. JAX-WS 주석을 사용하면 Java 클래스를 활성화하는 데 필요한 정보를 추가할 수 있습니다. 또한 WSDL 계약 대신 사용할 수 있는 Service Endpoint Interface (SEI)를 만들 수도 있습니다. WSDL 계약을 원하는 경우 Apache CXF는 주석이 달린 Java 코드에서 계약을 생성하는 도구를 제공합니다.

24.1. JAX-WS 서비스 개발 소개

Java에서 시작하는 서비스를 생성하려면 다음을 수행해야 합니다.

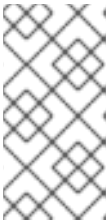
1. 24.2절. “SEI 생성” 서비스로 노출하려는 방법을 정의하는 SEI(Service Endpoint Interface) 입니다.

참고

Java 클래스에서 직접 작업할 수 있지만 인터페이스에서 작업하는 것이 좋습니다. 인터페이스는 서비스를 사용하는 애플리케이션 개발을 담당하는 개발자와 공유하는 데 더 적합합니다. 인터페이스는 더 작으며 서비스의 구현 세부 정보를 제공하지 않습니다.

2. 24.3절. “코드 주석 달기” 코드에 필요한 주석입니다.

3. 24.4절. “WSDL 생성” 서비스에 대한 WSDL 계약입니다.



참고

SEI를 서비스의 계약으로 사용하려는 경우 WSDL 계약을 생성할 필요가 없습니다.

4. 31장. 서비스 게시 서비스는 서비스 공급자입니다.

24.2. SEI 생성

24.2.1. 개요

서비스 엔드포인트 인터페이스 (SEI)는 서비스 구현과 해당 서비스에 대한 요청을 수행하는 소비자 간에 공유되는 Java 코드의 일부입니다. SEI는 서비스에서 구현하는 방법을 정의하고 서비스를 엔드포인트로 노출하는 방법에 대한 세부 정보를 제공합니다. WSDL 계약을 시작할 때 SEI는 코드 생성기에 의해 생성됩니다. 그러나 Java에서 시작할 때 개발자는 SEI를 생성해야 할 책임이 있습니다. SEI를 생성하기 위한 두 가지 기본 패턴이 있습니다.

- 녹색 필드 개발 - 이 패턴에서는 기존 Java 코드 또는 WSDL 없이 새로운 서비스를 개발하고 있습니다. SEI를 만들어 시작하는 것이 가장 좋습니다. 그런 다음 SEI를 사용하는 서비스 공급자 및 소비자를 구현하는 데 책임이 있는 모든 개발자에게 SEI를 배포할 수 있습니다.

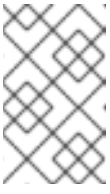
참고

녹색 필드 서비스 개발을 수행하는 권장 방법은 서비스 및 해당 인터페이스를 정의하는 WSDL 계약을 생성하여 시작하는 것입니다. 26장. Starting Point WSDL 계약 을 참조하십시오.

- 서비스 활성화 - 이 패턴에서는 일반적으로 Java 클래스로 구현되는 기존 기능 세트가 있으며 서비스 활성화를 원합니다. 즉, 두 가지를 해야 합니다.

a. 서비스의 일부로 노출되는 작업 만 포함하는 SEI를 만듭니다.

b. SEI를 구현하도록 기존 Java 클래스를 수정합니다.



참고

JAX-WS 주석을 Java 클래스에 추가할 수 있지만 권장되지는 않습니다.

24.2.2. 인터페이스 작성

SEI는 표준 Java 인터페이스입니다. 클래스에서 구현하는 메서드 집합을 정의합니다. 구현 클래스에서 액세스 권한이 있는 여러 멤버 필드와 상수를 정의할 수도 있습니다.It can also define a number of member fields and constants to which the implementing class has access.

SEI의 경우 정의된 메서드는 서비스에서 노출하는 작업에 매핑되도록 설계되었습니다. SEI는 wsdl:portType 요소에 해당합니다. SEI에서 정의한 방법은 wsdl:portType 요소의 wsdl:operation 요소에 해당합니다.

참고

JAX-WS는 서비스의 일부로 노출되지 않는 메서드를 지정할 수 있는 주석을 정의합니다. 그러나 이러한 방법을 SEI에서 벗어나는 것이 가장 좋습니다.

예 24.1. "간단한 SEI" 는 주식 업데이트 서비스에 대한 간단한 SEI를 보여줍니다.

예 24.1. 간단한 SEI

```
package com.fusesource.demo;

public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

24.2.3. 인터페이스 구현

SEI는 표준 Java 인터페이스이므로 구현하는 클래스는 표준 Java 클래스입니다. Java 클래스로 시작하는 경우 인터페이스를 구현하도록 수정해야 합니다. SEI로 시작하는 경우 구현 클래스는 SEI를 구현합니다.

예 24.2. "간단한 구현 클래스" 는 예 24.1. "간단한 SEI" 에서 인터페이스를 구현하는 클래스를 보여줍니다.

예 24.2. 간단한 구현 클래스

```
package com.fusesource.demo;

import java.util.*;

public class stockQuoteReporter implements quoteReporter
{
  ...
public Quote getQuote(String ticker)
  {
    Quote retVal = new Quote();
    retVal.setID(ticker);
```

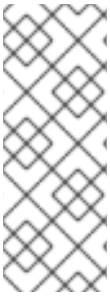
```
    retVal.setVal(Board.check(ticker));[1]
    Date retDate = new Date();
    retVal.setTime(retDate.toString());
    return(retVal);
  }
 }
```

24.3. 코드 주석 달기

24.3.1. JAX-WS 주석 개요

JAX-WS 주석은 **SEI**를 완전히 지정된 서비스 정의에 매핑하는 데 사용되는 메타데이터를 지정합니다. 주석에 제공된 정보 중 하나는 다음과 같습니다.

- 서비스의 대상 네임스페이스입니다.
- 요청 메시지를 보유하는 데 사용되는 클래스의 이름입니다.
- 응답 메시지를 보유하는 데 사용되는 클래스의 이름입니다.
- 작업을 한 가지 방법으로 수행 하는 경우**If an operation is a one way**
- 서비스에서 사용하는 바인딩 스타일
- 사용자 지정 예외에 사용되는 클래스의 이름입니다.
- 서비스에서 사용하는 유형이 정의된 네임스페이스



참고

대부분의 주석에는 적절한 기본값이 있으며 해당 주석에 값을 제공할 필요가 없습니다. 그러나 주석에 제공하는 정보가 많을수록 서비스 정의가 더 잘 지정됩니다. 잘 지정되어 있는 서비스 정의는 분산 애플리케이션의 모든 부분이 함께 작동할 가능성이 높아집니다.

24.3.2. 필수 주석

24.3.2.1. 개요

Java 코드에서 서비스를 생성하려면 코드에 주석을 하나만 추가해야 합니다. SEI 및 구현 클래스 모두에 @WebService 주석을 추가해야 합니다.

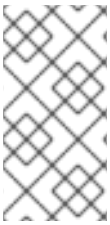
24.3.2.2. @WebService 주석

@WebService 주석은 javax.jws.WebService 인터페이스에서 정의되며 인터페이스 또는 서비스로 사용할 클래스에 배치됩니다. @WebService 에 설명 된 속성이 있습니다. 표 24.1. "@WebService 속성"

표 24.1. @WebService 속성

속성	설명
name	서비스 인터페이스의 이름을 지정합니다. 이 속성은 WSDL 계약에서 서비스 인터페이스를 정의하는 **wsdl:portType** 요소의 **name** 속성에 매핑됩니다. 기본값은 구현 클래스 이름에 **PortType** 을 추가하는 것입니다.[a]
targetNamespace	서비스가 정의된 대상 네임스페이스를 지정합니다. 이 속성을 지정하지 않으면 대상 네임스페이스는 패키지 이름에서 파생됩니다.If this property is not specified, the target namespace is derived from the package name.
serviceName	게시된 서비스의 이름을 지정합니다. 이 속성은 게시된 서비스를 정의하는 **wsdl:service** 요소의 **name** 속성에 매핑됩니다. 기본값은 서비스 구현 클래스의 이름을 사용하는 것입니다.
wsdlLocation	서비스의 WSDL 계약이 저장되는 URL을 지정합니다. 상대 URL을 사용하여 지정해야 합니다. 기본값은 서비스가 배포되는 URL입니다.
endpointInterface	구현 클래스에서 구현하는 SEI의 전체 이름을 지정합니다. 이 속성은 서비스 구현 클래스에서 특성이 사용되는 경우에만 지정됩니다.
portName	서비스가 게시되는 끝점의 이름을 지정합니다. 이 속성은 게시된 서비스의 끝점 세부 정보를 지정하는 **wsdl:port** 요소의 **name** 속성에 매핑됩니다. 기본값은 **서비스** 구현 클래스 이름에 추가 포트입니다.

[a] SEI에서 WSDL을 생성할 때 구현 클래스 이름 대신 인터페이스의 이름이 사용됩니다.



참고

@WebService 주석 속성의 값을 제공할 필요가 없습니다. 그러나 가능한 한 많은 정보를 제공하는 것이 좋습니다.

24.3.2.3. SEI에 주석 달기

SEI에서는 **@WebService** 주석을 추가해야 합니다. **SEI**는 서비스를 정의하는 계약이므로 **@WebService** 주석 속성에서 서비스에 대해 가능한 한 자세히 지정해야 합니다.

예 24.3. "@WebService 주석과 인터페이스" 는 **@WebService** 주석을 사용하여 예 24.1. "간단한 SEI" 에 정의된 인터페이스를 표시합니다.

예 24.3. @WebService 주석과 인터페이스

```
package com.fusesource.demo;

import javax.jws.*;

@WebService(name="quoteUpdater",
            targetNamespace="http://demos.redhat.com",
           serviceName="updateQuoteService",
            wsdlLocation="http://demos.redhat.com/quoteExampleService?wsdl",
            portName="updateQuotePort")
public interface quoteReporter
{
  public Quote getQuote(String ticker);
}
```

예 24.3. "@WebService 주석과 인터페이스" 의 **@WebService** 주석은 다음과 같습니다.

서비스 인터페이스를 정의하는 **wsdl:portType** 요소의 **name** 속성 값이 **quoteUpdater** 임을 지정합니다.

서비스의 대상 네임스페이스가 **http://demos.redhat.com** 임을 지정합니다.

게시된 서비스를 정의하는 **wsdl:service** 요소의 값이 **updateQuoteService** 임을 지정합니다.

서비스가 **http://demos.redhat.com/quoteExampleService?wsdl** 에서 **WSDL** 계약을 게시하도록 지

정합니다.

서비스를 노출하는 엔드포인트를 정의하는 **wsdl:port** 요소의 **name** 속성 값이 **updateQuotePort** 임을 지정합니다.

24.3.2.4. 서비스 구현에 주석 달기

@WebService 주석으로 **SEI**에 주석을 추가하는 것 외에도 **@WebService** 주석을 사용하여 서비스 구현 클래스에 주석을 달아야 합니다. 서비스 구현 클래스에 주석을 추가할 때 **endpointInterface** 속성만 지정해야 합니다. 예 24.4. "주석이 달린 서비스 구현 클래스" 에 표시된 대로 속성은 **SEI**의 전체 이름으로 설정해야 합니다.

예 24.4. 주석이 달린 서비스 구현 클래스

```
package org.eric.demo;

import javax.jws.*;

@WebService(endpointInterface="com.fusesource.demo.quoteReporter")
public class stockQuoteReporter implements quoteReporter
{
public Quote getQuote(String ticker)
  {
  ...
  }
}
```

24.3.3. 선택적 주석

초록

@WebService 주석은 **Java** 인터페이스 또는 **Java** 클래스를 활성화하는 서비스에 충분하지만 서비스 공급자로 서비스가 노출되는 방법을 완전히 설명하지는 않습니다. **JAX-WS** 프로그래밍 모델은 사용하는 바인딩과 같이 서비스에 대한 세부 정보를 **Java** 코드에 추가하는 데 다양한 선택적 주석을 사용합니다. 이러한 주석을 서비스의 **SEI**에 추가합니다.

SEI에 제공하는 세부 정보가 많을수록 개발자가 정의하는 기능을 사용할 수 있는 애플리케이션을 더 쉽게 구현할 수 있습니다. 또한 도구에서 생성된 **WSDL** 문서를 보다 구체적으로 만듭니다.

24.3.3.1. 개요

주석을 사용하여 바인딩 속성 정의

*서비스에 **SOAP** 바인딩을 사용하는 경우 **JAX-WS** 주석을 사용하여 여러 바인딩 속성을 지정할 수 있습니다. 이러한 속성은 서비스의 **WSDL** 계약에 지정할 수 있는 속성에 직접 해당합니다. 매개 변수 스타일과 같은 일부 설정은 메서드를 구현하는 방법을 제한할 수 있습니다. 이러한 설정은 메서드 매개 변수에 주석을 달 때 사용할 수 있는 주석에도 영향을 미칠 수 있습니다.*

24.3.3.2. @SOAPBinding 주석

*`@SOAPBinding` 주석은 **javax.jws.soap.SOAPBinding** 인터페이스에서 정의됩니다. 배포 시 서비스에서 사용하는 **SOAP** 바인딩에 대한 세부 정보를 제공합니다. `@SOAPBinding` 주석을 지정하지 않으면 래핑된 **doc/literal SOAP** 바인딩을 사용하여 서비스가 게시됩니다.*

SEI 및 SEI의 메서드에 `@SOAPBinding` 주석을 넣을 수 있습니다. 메서드에서 사용하는 경우 메서드의 `@SOAPBinding` 주석이 설정됩니다.

표 24.2. "@SOAPBinding 속성" `@SOAPBinding` 주석의 속성을 표시합니다.

표 24.2. @SOAPBinding 속성

속성	값	설명
style	style.DOCUMENT(기본값) Style.RPC	SOAP 메시지의 스타일을 지정합니다. RPC 스타일을 지정하면 SOAP 본문 내의 각 메시지 부분이 매개변수 또는 반환 값이며 **soap:body** 요소 내의 래퍼 요소 내에 표시됩니다. 래퍼 요소 내의 메시지 부분은 작업 매개 변수에 해당하며 작업의 매개변수와 동일한 순서로 표시되어야 합니다. DOCUMENT 스타일을 지정 하는 경우 SOAP 본문의 내용은 유효한 XML 문서 여야 하지만 폼이 엄격한 제한 되지 않습니다.If DOCUMENT style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.
Use	use.LITERAL(기본값) Use.ENCODED[a]	SOAP 메시지의 데이터를 스트리밍하는 방법을 지정합니다.Specifies how the data of the SOAP message is streamed.

속성	값	설명
parameterStyle [b]	ParameterStyle.BARE ParameterStyle.WRAPPED (기본값)	WSDL 계약의 메시지 부분에 해당하는 메서드 매개 변수가 SOAP 메시지 본문에 배치되는 방법을 지정합니다.Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. BARE를 지정하면 각 매개 변수가 메시지 본문에 메시지 루트의 하위 요소로 배치됩니다. WRAPPED를 지정하면 모든 입력 매개 변수가 요청 메시지의 단일 요소로 래핑되고 모든 출력 매개 변수는 응답 메시지의 단일 요소로 래핑됩니다.

[a] Use.ENCODED는 현재 지원되지 않습니다.

[b] **스타일** 을 RPC로 설정하는 경우 WRAPPED 매개변수 스타일을 사용해야 합니다.

24.3.3.3. 문서 베어 스타일 매개변수

*문서 베어 스타일은 **Java** 코드 간의 가장 직접적인 매핑이며 서비스의 결과 **XML** 표현입니다. 이 스타일을 사용하면 작업의 매개 변수 목록에 정의된 입력 및 출력 매개변수에서 스키마 유형이 직접 생성됩니다.*

***style** 속성이 **Style.DOCUMENT**로 설정된 **@SOAPBinding** 주석을 사용하여 베어 문서**literal** 스타일을 사용하고 해당 **parameterStyle** 속성을 **ParameterStyle.BARE**로 설정하도록 지정합니다.*

베어 매개 변수를 사용할 때 문서 스타일 사용 제한 사항을 위반하지 않도록 하려면 작업이 다음 조건을 준수해야 합니다.

- *작업에는 입력 또는 입력/출력 매개변수가 두 개 이상 있어야 합니다.*
- *작업에 **void** 이외의 반환 유형이 있는 경우 출력 또는 입력/출력 매개변수가 없어야 합니다.*
- *작업에 **void** 의 반환 유형이 있는 경우 출력 또는 입력/출력 매개 변수가 없어야 합니다.*

참고

@WebParam 주석 또는 **@WebResult** 주석을 사용하여 **SOAP** 헤더에 배치된 모든 매개변수는 허용된 매개 변수 수에 대해 계산되지 않습니다.

24.3.3.4. 래핑된 매개변수 문서

문서 래핑 스타일을 사용하면 **Java** 코드 간 매핑과 서비스의 결과 **XML** 표현과 같은 더 많은 **RPC**를 사용할 수 있습니다. 이 스타일을 사용하면 메서드의 매개 변수 목록의 매개 변수가 바인딩에 의해 단일 요소로 래핑됩니다. 이 문제의 단점은 **Java** 구현 간의 간접 계층과 메시지가 유선에 배치되는 방식을 도입한다는 것입니다.

래핑된 **document\literal** 스타일을 사용하도록 지정하려면 **style** 속성이 **Style.DOCUMENT**로 설정된 **@SOAPBinding** 주석을 사용하고 해당 **parameterStyle** 속성을 **ParameterStyle.WRAPPED**로 설정합니다.

"**@RequestWrapper** 주석" 주석 및 "**@ResponseWrapper** 주석" 주석을 사용하여 래퍼를 생성하는 방법을 제어할 수 있습니다.

24.3.3.5. 예제

예 24.5. "**SOAP Binding Annotation**을 사용하여 문서 **Bare SOAP** 바인딩 지정" 문서 베어 **SOAP** 메시지를 사용하는 **SEI**를 보여줍니다.

예 24.5. SOAP Binding Annotation을 사용하여 문서 Bare SOAP 바인딩 지정

```
package org.eric.demo;

import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="quoteReporter")
@SOAPBinding(parameterStyle=ParameterStyle.BARE)
public interface quoteReporter
{
  ...
}
```

24.3.3.6. 개요

주석을 사용하여 작업 속성 정의

*런타임이 **Java** 메서드 정의를 **XML** 작업 정의에 매핑하면 다음과 같은 세부 정보를 제공합니다.*

- *교환된 메시지는 **XML**에서 어떻게 보이는지*
- *메시지를 한 가지 방법으로 최적화할 수 있는 경우*
- *메시지가 정의된 네임스페이스*

24.3.3.7. @WebMethod 주석

*`@WebMethod` 주석은 **javax.jws.WebMethod** 인터페이스에서 정의됩니다. **SEI**의 메서드에 배치됩니다. `@WebMethod` 주석은 메서드가 연결된 작업을 설명하는 **wsdl:operation** 요소에 일반적으로 표시되는 정보를 제공합니다.*

표 24.3. “@WebMethod Properties” `@WebMethod` 주석의 속성을 설명합니다.

표 24.3. @WebMethod Properties

속성	설명
operationName	관련 **wsdl:operation** 요소의 **이름** 의 값을 지정합니다. 기본값은 메서드의 이름입니다.
작업	메서드에 대해 생성된 **soap:operation** 요소의 **soapAction** 특성 값을 지정합니다. 기본값은 빈 문자열입니다.
exclude	메서드를 서비스 인터페이스에서 제외해야 하는지 여부를 지정합니다. 기본값은 false입니다.

24.3.3.8. @RequestWrapper 주석

*`@RequestWrapper` 주석은 **javax.xml.ws.RequestWrapper** 인터페이스에서 정의됩니다. **SEI**의 메서드에 배치됩니다. `@RequestWrapper` 주석은 메시지 교환을 시작하는 요청 메시지의 메서드 매개변수에 래퍼 빈을 구현하는 **Java** 클래스를 지정합니다. 또한 요청 메시지를 마샬링 및 해제할 때 런타임에서 사용하는 요소 이름과 네임스페이스를 지정합니다.*

표 24.4. “@RequestWrapper 속성” `@RequestWrapper` 주석의 속성을 설명합니다.

표 24.4. @RequestWrapper 속성

속성	설명
localName	요청 메시지의 XML 표현에 래퍼 요소의 로컬 이름을 지정합니다.Specifies the local name of the wrapper element in the XML representation of the request message. 기본값은 메서드의 이름 또는 "@WebMethod 주석" 주석의 **operationName** 속성 값입니다.
targetNamespace	XML 래퍼 요소가 정의된 네임스페이스를 지정합니다. 기본값은 SEI의 대상 네임스페이스입니다.
className	래퍼 요소를 구현하는 Java 클래스의 전체 이름을 지정합니다.

참고

***className** 속성만 필요합니다.*

중요

*메서드에 **@SOAPBinding** 주석을 추가하고 해당 **parameterStyle** 속성이 **ParameterStyle.BARE** 로 설정된 경우 이 주석은 무시됩니다.*

24.3.3.9. @ResponseWrapper 주석

***@ResponseWrapper** 주석은 **javax.xml.ws.ResponseWrapper** 인터페이스에서 정의됩니다. **SEI**의 메서드에 배치됩니다. **@ResponseWrapper** 는 메시지 교환의 응답 메시지에 메서드 매개 변수에 대한 래퍼 빈을 구현하는 **Java** 클래스를 지정합니다. 또한 응답 메시지를 마샬링 및 해제할 때 런타임에서 사용하는 요소 이름 및 네임스페이스를 지정합니다.*

*표 24.5. "@ResponseWrapper 속성" **@ResponseWrapper** 주석의 속성을 설명합니다.*

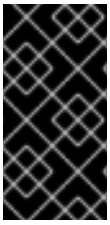
표 24.5. @ResponseWrapper 속성

속성	설명

속성	설명
localName	응답 메시지의 XML 표현에 래퍼 요소의 로컬 이름을 지정합니다.Specifies the local name of the wrapper element in the XML representation of the response message. 기본값은 Response가 추가된 메서드의 이름 또는 Response가 추가된 "@WebMethod 주석" 주석의 **operationName** 속성 값입니다.
targetNamespace	XML 래퍼 요소가 정의된 네임스페이스를 지정합니다. 기본값은 SEI의 대상 네임스페이스입니다.
className	래퍼 요소를 구현하는 Java 클래스의 전체 이름을 지정합니다.

참고

***className** 속성만 필요합니다.*



중요

***@SOAPBinding** 주석에도 메서드에 주석이 추가되고 해당 **parameterStyle** 속성이 **ParameterStyle.BARE** 로 설정된 경우 이 주석은 무시됩니다.*

24.3.3.10. @WebFault 주석

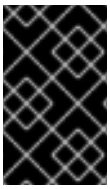
***@WebFault** 주석은 **javax.xml.ws.WebFault** 인터페이스에서 정의됩니다. **SEI**에서 **throw**되는 예외에 적용됩니다. **@WebFault** 주석은 **Java** 예외를 **wsdl:fault** 요소에 매핑하는 데 사용됩니다. 이 정보는 서비스 및 소비자 둘 다 처리할 수 있는 표현으로 예외를 마샬링하는 데 사용됩니다.*

*표 24.6. "@WebFault Properties" **@WebFault** 주석의 속성을 설명합니다.*

표 24.6. @WebFault Properties

속성	설명
name	fault 요소의 로컬 이름을 지정합니다.

속성	설명
targetNamespace	fault 요소가 정의된 네임스페이스를 지정합니다.Specifies the namespace under which the fault element is defined. 기본값은 SEI의 대상 네임스페이스입니다.
faultName	예외를 구현하는 Java 클래스의 전체 이름을 지정합니다.



중요

***name** 속성이 필요합니다.*

24.3.3.11. @Oneway 주석

***@Oneway** 주석은 **javax.jws.Oneway** 인터페이스에서 정의됩니다. 서비스의 응답이 필요하지 않은 **SEI**의 메서드에 배치됩니다. **@Oneway** 주석은 응답을 기다리지 않고 응답을 처리하는 리소스를 예약하지 않고 메서드 실행을 최적화할 수 있음을 런타임에 지시합니다.*

이 주석은 다음 기준을 충족하는 방법에서만 사용할 수 있습니다.

- ***void**를 반환합니다.*
- ***holder** 인터페이스를 구현하는 매개 변수가 없습니다.*
- *소비자로 다시 전달할 수 있는 예외는 **throw**되지 않습니다.*

24.3.3.12. 예제

*예 24.6. "Annotated methods가 있는 SEI" 주석이 달린 메서드가 포함된 **SEI**를 표시합니다.*

예 24.6. Annotated methods가 있는 SEI

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;
```

```
@WebService(name="quoteReporter")
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

24.3.3.13. 개요

주석을 사용하여 매개 변수 속성 정의

SEI의 메서드 매개 변수는 **wsdl:message** 요소와 **wsdl:part** 요소에 해당합니다. **JAX-WS**는 메서드 매개 변수에 대해 생성된 **wsdl:part** 요소를 설명할 수 있는 주석을 제공합니다.

24.3.3.14. @WebParam 주석

@WebParam 주석은 **javax.jws.WebParam** 인터페이스에서 정의됩니다. **SEI**에 정의된 메서드의 매개 변수에 배치됩니다. **@WebParam** 주석을 사용하면 매개 변수가 **SOAP** 헤더에 배치될 경우 매개 변수의 방향을 지정하고 생성된 **wsdl:part** 의 기타 속성을 지정할 수 있습니다.

표 24.7. “@WebParam 속성” **@WebParam** 주석의 속성을 설명합니다.

표 24.7. @WebParam 속성

속성	값	설명
name		생성된 WSDL 문서에 표시되는 매개 변수의 이름을 지정합니다. RPC 바인딩의 경우 매개 변수를 나타내는 **wsdl:part** 의 이름입니다. 문서 바인딩의 경우 매개 변수를 나타내는 XML 요소의 로컬 이름입니다.For document bindings, this is the local name of the XML element representing the parameter. JAX-WS 사양에 따라 기본값은 **arg*N*** 입니다. 여기서 *N* 은 0 기반 인수 인덱스(예: arg0, arg1, 등)로 대체됩니다.

속성	값	설명
targetNamespace		매개변수의 네임스페이스를 지정합니다. 매개 변수가 XML 요소에 매핑되는 문서 바인딩에서만 사용됩니다. 기본값은 서비스의 네임스페이스를 사용하는 것입니다.
mode	mode.IN (기본값)[a] Mode.OUT Mode.INOUT	매개 변수의 방향을 지정합니다.
header	false(기본값) true	매개 변수가 SOAP 헤더의 일부로 전달되는지 여부를 지정합니다.
partName		매개 변수에 대한 **wsdl:part** 요소의 **name** 속성 값을 지정합니다. 이 속성은 문서 스타일 SOAP 바인딩에 사용됩니다.

[a] Holder 인터페이스를 구현하는 모든 매개변수는 기본적으로 Mode.INOUT에 매핑됩니다.

24.3.3.15. @WebResult 주석

***@WebResult** 주석은 **javax.jws.WebResult** 인터페이스에서 정의됩니다. **SEI**에 정의된 메서드에 배치됩니다. **@WebResult** 주석을 사용하면 메서드의 반환 값에 대해 생성된 **wsdl:part** 의 속성을 지정할 수 있습니다.*

*표 24.8. "@WebResult 속성" **@WebResult** 주석의 속성을 설명합니다.*

표 24.8. @WebResult 속성

속성	설명
name	생성된 WSDL 문서에 표시되는 반환 값의 이름을 지정합니다. RPC 바인딩의 경우 이는 반환 값을 나타내는 **wsdl:part** 의 이름입니다. 문서 바인딩의 경우 이는 반환 값을 나타내는 XML 요소의 로컬 이름입니다.For document bindings, this is the local name of the XML element representing the return value. 기본값은 return입니다.

속성	설명
targetNamespace	반환 값의 네임스페이스를 지정합니다. 반환 값이 XML 요소에 매핑되는 문서 바인딩에서만 사용됩니다. 기본 값은 서비스의 네임스페이스를 사용하는 것입니다.
header	반환 값이 SOAP 헤더의 일부로 전달되는지 여부를 지정합니다.Specifies if the return value is passed as part of the SOAP header.
partName	반환 값에 대한 **wsdl:part** 요소의 **name** 속성 값을 지정합니다. 이 속성은 문서 스타일 SOAP 바인딩에 사용됩니다.

24.3.3.16. 예제

예 24.7. "완전히 주석 처리 SEI" 주석이 완전히 추가된 SEI를 표시합니다.

예 24.7. 완전히 주석 처리 SEI

```
package com.fusesource.demo;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(targetNamespace="http://demo.redhat.com",
            name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface quoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.redhat.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.redhat.com/types",
                   className="org.eric.demo.Quote")
  @WebResult(targetNamespace="http://demo.redhat.com/types",
             name="updatedQuote")
  public Quote getQuote(
                     @WebParam(targetNamespace="http://demo.redhat.com/types",
                               name="stockTicker",
                               mode=Mode.IN)
                     String ticker
  );
}
```

24.3.4. Apache CXF 주석

24.3.4.1. WSDL 문서

24.3.4.1.1. @WSDLDocumentation 주석

@WSDLDocumentation 주석은 org.apache.cxf.annotations.WSDLDocumentation 인터페이스에서 정의됩니다. SEI 또는 SEI 방법에 설치할 수 있습니다.

이 주석을 사용하면 SEI를 WSDL으로 변환한 후 wsdl:documentation 요소에 표시되는 문서를 추가할 수 있습니다. 기본적으로 설명서 요소는 포트 유형 내에 표시되지만 placement 속성을 지정하여 documentation in the WSDL 파일의 다른 위치에 표시되도록 할 수 있습니다. 24.3.4.2절. "@WSDLDocumentation 속성" @WSDLDocumentation 주석에서 지원하는 속성을 보여줍니다.

24.3.4.2. @WSDLDocumentation 속성

속성	설명
value	**(필수)** 문서 텍스트를 포함하는 문자열입니다.
배치	**(선택 사항)** WSDL 파일에서 이 문서가 표시되는 위치를 지정합니다. 가능한 배치 값 목록은 "WSDL 계약 포함" 에서 참조하십시오.
faultClass	**(선택 사항)** 배치가 **FAULT_MESSAGE**,**PORT_TYPE_OPERATION_FAULT** 또는 **BINDING_OPERATION_FAULT** 로 설정된 경우 이 속성을 오류를 나타내는 Java 클래스로 설정해야 합니다.

24.3.4.2.1. @WSDLDocumentationCollection 주석

@WSDLDocumentationCollection 주석은 org.apache.cxf.annotations.WSDLDocumentationCollection 인터페이스에서 정의됩니다. SEI 또는 SEI 방법에 설치할 수 있습니다.

이 주석은 단일 배치 위치 또는 다양한 배치 위치에 여러 문서 요소를 삽입하는 데 사용됩니다.

24.3.4.2.2. WSDL 계약 포함

WSDL 계약에 문서가 표시될 위치를 지정하려면 WSDLDocumentation.Placement 유형의 placement 속성을 지정할 수 있습니다. 배치에는 다음 값 중 하나가 있을 수 있습니다.

- *WSDLDocumentation.Placement.BINDING*
- *WSDLDocumentation.Placement.BINDING_OPERATION*
- *WSDLDocumentation.Placement.BINDING_OPERATION_FAULT*
- *WSDLDocumentation.Placement.BINDING_OPERATION_INPUT*
- *WSDLDocumentation.Placement.BINDING_OPERATION_OUTPUT*
- *WSDLDocumentation.Placement.DEFAULT*
- *WSDLDocumentation.Placement.FAULT_MESSAGE*
- *WSDLDocumentation.Placement.INPUT_MESSAGE*
- *WSDLDocumentation.Placement.OUTPUT_MESSAGE*
- *WSDLDocumentation.Placement.PORT_TYPE*
- *WSDLDocumentation.Placement.PORT_TYPE_OPERATION*
- *WSDLDocumentation.Placement.PORT_TYPE_OPERATION_FAULT*
- *WSDLDocumentation.Placement.PORT_TYPE_OPERATION_INPUT*

- ***WSDLDocumentation.Placement.PORT_TYPE_OPERATION_OUTPUT***
- ***WSDLDocumentation.Placement.SERVICE***
- ***WSDLDocumentation.Placement.SERVICE_PORT***
- ***WSDLDocumentation.Placement.TOP***

24.3.4.2.3. @WSDLDocumentation의 예

24.3.4.3절. "@WSDLDocumentation 사용" SEI 및 해당 방법 중 하나에 @WSDLDocumentation 주석을 추가하는 방법을 보여줍니다.

24.3.4.3. @WSDLDocumentation 사용

```
@WebService
@WSDLDocumentation("A very simple example of an SEI")
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.4절. "문서를 사용하여 생성된 WSDL" 에 표시된 WSDL은 24.3.4.3절. "@WSDLDocumentation 사용" SEI에서 생성되는 경우 문서 요소의 기본 배치는 각각 PORT_TYPE 및 PORT_TYPE_OPERATION 입니다.

24.3.4.4. 문서를 사용하여 생성된 WSDL

```
<wsdl:definitions ... >
  ...
  <wsdl:portType name="HelloWorld">
    <wsdl:documentation>A very simple example of an SEI</wsdl:documentation>
    <wsdl:operation name="sayHi">
      <wsdl:documentation>A traditional form of greeting</wsdl:documentation>
      <wsdl:input name="sayHi" message="tns:sayHi">
    </wsdl:input>
      <wsdl:output name="sayHiResponse" message="tns:sayHiResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

24.3.4.4.1. @WSDLDocumentationCollection의 예

24.3.4.5절. "@WSDLDocumentationCollection 사용" SEI에 @WSDLDocumentationCollection 주석을 추가하는 방법을 보여줍니다.

24.3.4.5. @WSDLDocumentationCollection 사용

```
@WebService
@WSDLDocumentationCollection(
    {
        @WSDLDocumentation("A very simple example of an SEI"),
        @WSDLDocumentation(value = "My top level documentation",
                           placement = WSDLDocumentation.Placement.TOP),
        @WSDLDocumentation(value = "Binding documentation",
                           placement = WSDLDocumentation.Placement.BINDING)
    }
)
public interface HelloWorld {
    @WSDLDocumentation("A traditional form of Geeky greeting")
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.6. 메시지 스키마 유효성 검사

24.3.4.6.1. @SchemaValidation 주석

@SchemaValidation 주석은 org.apache.cxf.annotations.SchemaValidation 인터페이스에서 정의됩니다. SEI 및 개별 SEI 방법에 설치할 수 있습니다.

이 주석은 이 끝점으로 전송된 XML 메시지의 스키마 유효성 검사를 설정합니다. 이 기능은 들어오는 XML 메시지 형식에 문제가 있다고 의심되는 경우 테스트 목적에 유용할 수 있습니다. 성능에 큰 영향을 미치기 때문에 유효성 검사가 기본적으로 비활성화되어 있습니다.

24.3.4.6.2. 스키마 검증 유형

스키마 유효성 검사 동작은 값이 org.apache.cxf.annotations.SchemaValidation.SchemaValidationType 유형의 열거형인 type 매개 변수에 의해 제어됩니다. 24.3.4.7절. "스키마 유효성 검사 유형 값" 사용 가능한 유효성 검사 유형 목록을 표시합니다.

24.3.4.7. 스키마 유효성 검사 유형 값

유형	설명
IN	클라이언트 및 서버에서 들어오는 메시지에 스키마 유효성 검사를 적용합니다.
OUT	클라이언트 및 서버에서 나가는 메시지에 스키마 유효성 검사를 적용합니다.
BOTH	클라이언트와 서버에서 수신 및 발신 메시지에 스키마 유효성 검사를 적용합니다.
NONE	모든 스키마 유효성 검사가 비활성화됩니다.
REQUEST	스키마 검증을 적용하여 메시지 요청(즉, 발신 클라이언트 메시지 및 수신 서버 메시지에 유효성 검사가 적용됩니다.
RESPONSE	응답 메시지에 스키마 유효성 검사를 적용합니다. 즉, 들어오는 클라이언트 메시지 및 발신 서버 메시지에 유효성 검사가 적용됩니다.

24.3.4.7.1. 예제

*다음 예제에서는 **MyService SEI**를 기반으로 끝점에 대한 메시지의 스키마 유효성 검사를 활성화하는 방법을 보여줍니다. 주석을 **SEI** 전체에 적용하는 방법과 **SEI**의 개별 메서드에 주석을 적용하는 방법을 확인합니다.*

```
@WebService
@SchemaValidation(type = SchemaValidationType.BOTH)
public interface MyService {
    Foo validateBoth(Bar data);

    @SchemaValidation(type = SchemaValidationType.NONE)
    Foo validateNone(Bar data);

    @SchemaValidation(type = SchemaValidationType.IN)
    Foo validateIn(Bar data);

    @SchemaValidation(type = SchemaValidationType.OUT)
    Foo validateOut(Bar data);

    @SchemaValidation(type = SchemaValidationType.REQUEST)
    Foo validateRequest(Bar data);

    @SchemaValidation(type = SchemaValidationType.RESPONSE)
    Foo validateResponse(Bar data);
}
```

24.3.4.8. 데이터 바인딩 지정

24.3.4.8.1. @DataBinding 주석

@DataBinding 주석은 org.apache.cxf.annotations.DataBinding 인터페이스에서 정의됩니다. 그것은 SEI에 배치되어 있습니다.

이 주석은 데이터 바인딩을 SEI와 연결하고 기본 JAXB 데이터 바인딩을 대체하는 데 사용됩니다. @DataBinding 주석의 값은 데이터 바인딩인ClassName .class 여야 합니다.

24.3.4.8.2. 지원되는 데이터 바인딩

현재 Apache CXF에서 지원되는 데이터 바인딩은 다음과 같습니다.

- org.apache.cxf.jaxb.JAXBDataBinding

  (기본값) 표준 JAXB 데이터 바인딩입니다.

- org.apache.cxf.sdo.SDODataBinding

  Service Data Objects(SDO) 데이터 바인딩은 Apache Tuscany SDO 구현을 기반으로 합니다. Maven 빌드 컨텍스트에서 이 데이터 바인딩을 사용하려면 cxf-rt-databinding-sdo 아티팩트에 종속성을 추가해야 합니다.

- org.apache.cxf.aegis.databinding.AegisDatabinding

  Maven 빌드 컨텍스트에서 이 데이터 바인딩을 사용하려면 cxf-rt-databinding-aegis 아티팩트에 종속성을 추가해야 합니다.

- org.apache.cxf.xmlbeans.XmlBeansDataBinding

  Maven 빌드 컨텍스트에서 이 데이터 바인딩을 사용하려면 cxf-rt-databinding-xmlbeans 아티팩트에 종속성을 추가해야 합니다.

- ***org.apache.cxf.databinding.source.SourceDataBinding***

  *이 데이터 바인딩은 **Apache CXF** 코어에 속합니다.*

- ***org.apache.cxf.databinding.stax.StaxDataBinding***

  *이 데이터 바인딩은 **Apache CXF** 코어에 속합니다.*

24.3.4.8.3. 예제

*24.3.4.9절. "데이터 바인딩 설정" **HelloWorld SEI**와 **SDO** 바인딩을 연결하는 방법을 보여줍니다.*

24.3.4.9. 데이터 바인딩 설정

```
@WebService
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.10. 메시지 압축

24.3.4.10.1. @GZIP 주석

*@GZIP 주석은 **org.apache.cxf.annotations.GZIP** 인터페이스에서 정의됩니다. 그것은 **SEI**에 배치되어 있습니다.*

*메시지의 **GZIP** 압축을 활성화합니다. **GZIP**는 협상된 개선 사항입니다. 즉, 클라이언트의 초기 요청은 **gzip**으로 지정되지 않지만 **Accept** 헤더가 추가되고 서버가 **GZIP** 압축을 지원하는 경우 응답이 **gzip**으로 표시되고 후속 요청도 수행됩니다.*

*24.3.4.11절. "@GZIP 속성" **@GZIP** 주석에서 지원하는 선택적 속성을 보여줍니다.*

24.3.4.11. @GZIP 속성

속성	설명

속성	설명
threshold	이 속성에서 지정한 크기보다 작은 메시지는 gzip으로 **표시되지 않습니다**. 기본값은 -1(제한 없음)입니다.

24.3.4.11.1. @FastInfoset

@FastInfoset 주석은 org.apache.cxf.annotations.FastInfoset 인터페이스에서 정의됩니다. 그것은 SEI에 배치되어 있습니다.

메시지에 FastInfoset 형식을 사용할 수 있습니다. FastInfoset은 XML 메시지의 이진 인코딩 형식이며, 이는 메시지 크기와 XML 메시지의 처리 성능을 모두 최적화하는 것을 목표로 합니다. 자세한 내용은 Fast Infoset 에 대한 다음 Sun 문서를 참조하십시오.

FastInfoset은 협상된 개선 사항입니다. 즉, 클라이언트의 초기 요청은 FastInfoset 형식이 아니지만 Accept 헤더가 추가되고 서버가 FastInfoset을 지원하는 경우 응답은 FastInfoset에 있으며 이후 요청도 포함됩니다.

24.3.4.12절. "@FastInfoset Properties" @FastInfoset 주석에서 지원하는 선택적 속성을 보여줍니다.

24.3.4.12. @FastInfoset Properties

속성	설명
force	협상 대신 FastInfoset 형식을 강제로 사용하는 부울 속성입니다. **true** 인 경우 fastInfoset 형식을 강제로 사용합니다. 그렇지 않으면 협상합니다. 기본값은 **false** 입니다.

24.3.4.12.1. @GZIP의 예

24.3.4.13절. "GZIP 활성화" HelloWorld SEI에 대해 GZIP 압축을 활성화하는 방법을 보여줍니다.

24.3.4.13. GZIP 활성화

```
@WebService
@GZIP
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.13.1. @FastInfoset 시험

24.3.4.14절. "FastInfoset 활성화" HelloWorld SEI에 대해 FastInfoset 형식을 활성화하는 방법을 보여줍니다.

24.3.4.14. FastInfoset 활성화

```
@WebService
@FastInfoset
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.15. 끝점에서 로깅 활성화

24.3.4.15.1. @logging 주석

@Logging 주석은 org.apache.cxf.annotations.Logging 인터페이스에서 정의됩니다. 그것은 SEI에 배치되어 있습니다.

이 주석을 사용하면 SEI와 연결된 모든 끝점에 로깅할 수 있습니다. 24.3.4.16절. "@logging 속성" 이 주석에서 설정할 수 있는 선택적 속성을 표시합니다.

24.3.4.16. @logging 속성

속성	설명
제한	로그에서 메시지가 잘리는 것 이상으로 크기 제한을 지정합니다. 기본값은 64K입니다.
inLocation	수신 메시지를 로깅할 위치를 지정합니다. < stderr >, < stdout > , < logger > 또는 파일 이름이 될 수 있습니다. 기본값은 &lt ;logger> 입니다.
outLocation	나가는 메시지를 로깅할 위치를 지정합니다. < stderr >, < stdout > , < logger > 또는 파일 이름이 될 수 있습니다. 기본값은 &lt ;logger> 입니다.

24.3.4.16.1. 예제

24.3.4.17절. "주석을 사용한 로깅 구성" 는 들어오는 메시지가 < stdout >로 전송되고 발신 메시지가

< logger >로 전송되는 HelloWorld SEI에 대한 로깅을 활성화하는 방법을 보여줍니다.

24.3.4.17. 주석을 사용한 로깅 구성

```
@WebService
@Logging(limit=16000, inLocation="<stdout>")
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.18. 끝점에 속성 및 정책 추가

초록

속성 및 정책을 모두 사용하여 구성 데이터를 끝점과 연결할 수 있습니다. 이러한 문제의 근본적인 차이점은 속성이 Apache CXF 특정 구성 메커니즘인 반면 정책 은 표준 WSDL 구성 메커니즘입니다. 일반적으로 정책은 WS 사양 및 표준에서 시작되며 일반적으로 WSDL 계약에 표시되는 wsdl:policy 요소를 정의하여 설정됩니다. 반대로 속성은 Apache CXF 관련이며 일반적으로 Apache CXF Spring 구성 파일에서 jaxws:properties 요소를 정의하여 설정됩니다.

그러나 여기에 설명된 대로 주석을 사용하여 Java에서 속성 설정 및 WSDL 정책 설정을 정의할 수도 있습니다.

24.3.4.19. 속성 추가

24.3.4.19.1. @EndpointProperty 주석

@EndpointProperty 주석은 org.apache.cxf.annotations.EndpointProperty 인터페이스에서 정의됩니다. 그것은 SEI에 배치되어 있습니다.

이 주석은 Apache CXF 관련 구성 설정을 엔드포인트에 추가합니다. 엔드포인트 속성은 Spring 구성 파일에 지정할 수도 있습니다. 예를 들어 끝점에서 WS-Security를 구성하려면 Spring 구성 파일에서 jaxws:properties 요소를 사용하여 끝점 속성을 추가할 수 있습니다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ... >

  <jaxws:endpoint
    id="MyService"
    address="https://localhost:9001/MyService"
    serviceName="interop:MyService"
    endpointName="interop:MyServiceEndpoint"
```

```
    implementor="com.foo.MyService">

    <jaxws:properties>
      <entry key="ws-security.callback-handler" value="interop.client.UTPasswordCallback"/>
      <entry key="ws-security.signature.properties" value="etc/keystore.properties"/>
      <entry key="ws-security.encryption.properties" value="etc/truststore.properties"/>
      <entry key="ws-security.encryption.username" value="useReqSigCert"/>
    </jaxws:properties>

  </jaxws:endpoint>
</beans>
```

또는 24.3.4.20절. “@EndpointProperty Annotations를 사용하여 WS-Security 구성” 와 같이 **SEI** 에 **@EndpointProperty** 주석을 추가하여 **Java**에서 이전 구성 설정을 지정할 수 있습니다.

24.3.4.20. @EndpointProperty Annotations를 사용하여 WS-Security 구성

```
@WebService
@EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback")
@EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties")
@EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties")
@EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
public interface HelloWorld {
   String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.20.1. @EndpointProperties 주석

@EndpointProperties 주석은 **org.apache.cxf.annotations.EndpointProperties** 인터페이스에서 정의됩니다. 그것은 **SEI**에 배치되어 있습니다.

이 주석은 여러 **@EndpointProperty** 주석을 목록으로 그룹화하는 방법을 제공합니다. **@EndpointProperties** 를 사용하면 24.3.4.21절. “@EndpointProperties 주석을 사용하여 WS-Security 구성” 와 같이 24.3.4.20절. “@EndpointProperty Annotations를 사용하여 WS-Security 구성” 을 다시 쓸 수 있습니다.

24.3.4.21. @EndpointProperties 주석을 사용하여 WS-Security 구성

```
@WebService
@EndpointProperties(
  {
  @EndpointProperty(name="ws-security.callback-handler"
value="interop.client.UTPasswordCallback"),
  @EndpointProperty(name="ws-security.signature.properties" value="etc/keystore.properties"),
  @EndpointProperty(name="ws-security.encryption.properties" value="etc/truststore.properties"),
  @EndpointProperty(name="ws-security.encryption.username" value="useReqSigCert")
})
```

```
public interface HelloWorld {
    String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.22. 정책 추가

24.3.4.22.1. @policy 주석

@Policy 주석은 org.apache.cxf.annotations.Policy 인터페이스에서 정의됩니다. SEI 또는 SEI 방법에 설치할 수 있습니다.

이 주석은 WSDL 정책을 SEI 또는 SEI 방법과 연결하는 데 사용됩니다. 이 정책은 표준 wsdl:policy 요소가 포함된 XML 파일을 참조하는 URI를 제공하여 지정합니다. WSDL 계약이 SEI에서 생성되는 경우 (예: java2ws 명령줄 툴 사용) 이 정책을 WSDL에 포함할지 여부를 지정할 수 있습니다.

24.3.4.23절. "@policy 속성" @Policy 주석에서 지원하는 속성을 표시합니다.

24.3.4.23. @policy 속성

속성	설명
uri	**(필수)** 정책 정의가 포함된 파일의 위치입니다.
includeInWSDL	**(선택 사항)** WSDL을 생성할 때 생성된 계약에 정책을 포함할지 여부입니다. 기본값은 **true** 입니다.
배치	**(선택 사항)** WSDL 파일에서 이 문서가 표시되는 위치를 지정합니다. 가능한 배치 값 목록은 "WSDL 계약 포함" 에서 참조하십시오.
faultClass	**(선택 사항)** 배치가 **BINDING_OPERATION_FAULT** 또는 **PORT_TYPE_OPERATION_FAULT** 로 설정된 경우 이 속성도 설정하여 이 정책이 적용되는 오류를 지정해야 합니다. 값은 오류를 나타내는 Java 클래스입니다.

24.3.4.23.1. @policies 주석

@Policies 주석은 org.apache.cxf.annotations.Policies 인터페이스에서 정의됩니다. SEI 또는 thse SEI 방법에 배치할 수 있습니다.

*이 주석은 여러 **@Policy** 주석을 목록으로 그룹화하는 방법을 제공합니다.*

24.3.4.23.2. WSDL 계약 포함

__WSDL__ 계약에 정책이 표시될 위치를 지정하려면 __Policy.Placement__ 유형인 __placement__ 속성을 지정할 수 있습니다. 배치에는 다음 값 중 하나가 있을 수 있습니다.

```
Policy.Placement.BINDING
Policy.Placement.BINDING_OPERATION
Policy.Placement.BINDING_OPERATION_FAULT
Policy.Placement.BINDING_OPERATION_INPUT
Policy.Placement.BINDING_OPERATION_OUTPUT
Policy.Placement.DEFAULT
Policy.Placement.PORT_TYPE
Policy.Placement.PORT_TYPE_OPERATION
Policy.Placement.PORT_TYPE_OPERATION_FAULT
Policy.Placement.PORT_TYPE_OPERATION_INPUT
Policy.Placement.PORT_TYPE_OPERATION_OUTPUT
Policy.Placement.SERVICE
Policy.Placement.SERVICE_PORT
```

24.3.4.23.3. @Policy의 예

다음 예제에서는 __WSDL__ 정책을 __HelloWorld SEI__와 연결하는 방법과 정책을 __sayHi__ 메서드와 연결하는 방법을 보여줍니다. 정책은 주석 디렉터리에 있는 파일 시스템의 __XML__ 파일에 저장됩니다.

```
@WebService
@Policy(uri = "annotationpolicies/TestImplPolicy.xml",
      placement = Policy.Placement.SERVICE_PORT),
@Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
      placement = Policy.Placement.PORT_TYPE)
public interface HelloWorld {
   @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
         placement = Policy.Placement.PORT_TYPE_OPERATION),
   String sayHi(@WebParam(name = "text") String text);
}
```

24.3.4.23.4. @Policies의 예

__@Policies__ 주석을 사용하여 다음 예와 같이 여러 __@Policy__ 주석을 목록에 그룹화할 수 있습니다.

```
@WebService
@Policies({
   @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
         placement = Policy.Placement.SERVICE_PORT),
   @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
```

```
        placement = Policy.Placement.PORT_TYPE)
})
public interface HelloWorld {
    @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
        placement = Policy.Placement.PORT_TYPE_OPERATION),
    String sayHi(@WebParam(name = "text") String text);
}
```

24.4. WSDL 생성

24.4.1. Maven 사용

코드에 주석이 추가되면 **java2ws Maven** 플러그인의 **-wsdl** 옵션을 사용하여 서비스에 대한 **WSDL** 계약을 생성할 수 있습니다. **java2ws Maven** 플러그인의 자세한 옵션 목록은 **44.3절. "java2ws"** 에서 참조하십시오.

예 24.8. "Java에서 WSDL 생성" 에서는 **java2ws Maven** 플러그인을 설정하여 **WSDL**을 생성하는 방법을 보여줍니다.

예 24.8. Java에서 WSDL 생성

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <genWsdl>true</genWsdl>
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

참고

className 값을 정규화된 **className**으로 바꿉니다.

24.4.2. 예제

예 24.9. "SEI에서 생성된 WSDL" 는 예 24.7. "완전히 주석 처리 SEI" 에 표시된 **SEI**에 대해 생성되는 **WSDL** 계약을 보여줍니다.

예 24.9. SEI에서 생성된 WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://demo.eric.org/"
        xmlns:tns="http://demo.eric.org/"
    xmlns:ns1=""
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns2="http://demo.eric.org/types"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema>
      <xs:complexType name="quote">
        <xs:sequence>
          <xs:element name="ID" type="xs:string" minOccurs="0"/>
          <xs:element name="time" type="xs:string" minOccurs="0"/>
          <xs:element name="val" type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getStockQuote">
    <wsdl:part name="stockTicker" type="xsd:string">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="getStockQuoteResponse">
    <wsdl:part name="updatedQuote" type="tns:quote">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="quoteReporter">
    <wsdl:operation name="getStockQuote">
      <wsdl:input name="getQuote" message="tns:getStockQuote">
    </wsdl:input>
      <wsdl:output name="getQuoteResponse" message="tns:getStockQuoteResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="quoteReporterBinding" type="tns:quoteReporter">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getStockQuote">
      <soap:operation style="rpc" />
      <wsdl:input name="getQuote">
```

```
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output name="getQuoteResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="quoteReporterService">
  <wsdl:port name="quoteReporterPort" binding="tns:quoteReporterBinding">
    <soap:address location="http://localhost:9000/quoteReporterService" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

---

[1]
Board is an assumed class whose implementation is left to the reader.

25장. WSDL 계약 없이 소비자 개발

초록

서비스 소비자를 개발하기 위해 WSDL 계약이 필요하지 않습니다. 주석이 달린 SEI에서 서비스 소비자를 생성할 수 있습니다. 서비스를 노출하는 엔드포인트가 게시되는 주소, 서비스를 노출하는 엔드포인트를 정의하는 서비스 요소의 QName 및 소비자가 요청하는 끝점을 정의하는 포트 요소의 QName을 알아야 합니다. 이 정보는 SEI의 주석에 지정하거나 별도로 제공할 수 있습니다.

25.1. JAVA-FIRST CONSUMER DEVELOPMENT

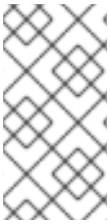
WSDL 계약 없이 소비자를 생성하려면 다음을 수행해야 합니다.

1. 소비자 가 작업을 호출할 서비스에 대한 Service 오브젝트를 생성합니다.
2. Service 오브젝트에 포트를 추가 합니다.
3. Service 오브젝트의 getPort() 메서드를 사용하여 서비스의 프록시를 가져옵니다.
4. 소비자의 비즈니스 로직을 구현합니다.

25.2. 서비스 오브젝트 생성

25.2.1. 개요

javax.xml.ws.Service 클래스는 서비스를 노출하는 모든 엔드포인트의 정의를 포함하는 wsdl:service 요소를 나타냅니다. 따라서 서비스에서 원격 호출을 수행하기 위한 프록시인 wsdl:port 요소에 의해 정의된 엔드포인트를 가져올 수 있는 메서드를 제공합니다.



참고

Service 클래스는 XML 문서 작업과 달리 클라이언트 코드가 Java 유형에서 작업할 수 있도록 하는 추상화를 제공합니다.

25.2.2. create() 메서드

Service 클래스에는 새 Service 오브젝트를 생성하는 데 사용할 수 있는 두 가지 static create() 메서드가 있습니다. 예 25.1. “Service create() 메서드” 에 표시된 대로, 두 create() 메서드 모두 Service 오브젝트가 나타내는 wsdl:service 요소의 QName을 사용하고, 하나는 WSDL 계약의 위치를 지정하는 URI를 취합니다.

참고

모든 서비스는 WSDL 계약을 게시합니다. SOAP/HTTP 서비스의 경우 URI는 일반적으로 ?wsdl 과 함께 추가된 서비스의 URI입니다.

> 예 25.1. Service create() 메서드
>
> public staticServicecreateURLwsdlLocationQNameserviceNameWebServiceExceptioncreateQNameserviceNameWebServiceException

serviceName 매개변수 값은 QName입니다. 네임스페이스 부분의 값은 서비스의 대상 네임스페이스입니다. 서비스의 대상 네임스페이스는 @WebService 주석의 targetNamespace 속성에 지정됩니다. QName의 로컬 부분의 값은 wsdl:service 요소의 name 속성 값입니다. 다음 방법 중 하나로 이 값을 확인할 수 있습니다. . @WebService 주석의 serviceName 속성에 지정됩니다.

1. @Web Service 주석의 name 속성 값에 Service를 추가합니다.
2. SEI 이름에 Service 를 추가합니다.

중요

OSGi 환경에 배포된 프로그래밍 방식으로 생성된 CXF 소비자는 Incurring ClassNotFoundExceptions의 가능성을 피하기 위해 특별한 처리가 필요합니다. 프로그래밍 방식으로 생성된 CXF 소비자가 포함된 각 번들에 대해 Singleton CXF 기본 버스를 생성하고 모든 번들의 CXF 소비자가 이를 사용하는지 확인해야 합니다. 이 보호 장치가 없으면 하나의 번들에서 생성된 CXF 기본 버스가 할당될 수 있으므로 상속 번들이 실패할 수 있습니다.

예를 들어, 번들 A가 CXF 기본 버스를 명시적으로 설정하지 않고 번들 B에 생성된 CXF 기본 버스가 할당되었다고 가정합니다. 번들 A에서 CXF 버스가 추가 기능(예: SSL 또는 WS-Security)으로 구성되어야 하는 경우 또는 번들 A의 애플리케이션에서 특정 클래스 또는 리소스를 로드하는데 필요한 경우 실패합니다. 이는 CXF 버스 인스턴스가 스레드 컨텍스트 클래스 로더(TCCL)를 생성한 번들의 번들 클래스 로더로 설정했기 때문입니다(이 경우 번들 B). 또한 ws4j(CXF에서 WS-Security 구현)와 같은 특정 프레임워크는 TCCL을 사용하여 번들 내부에서 calback 처리기 클래스 또는 기타 속성 파일과 같은 리소스를 로드합니다. 번들 A에는 번들 B의 기본 CXF 버스와 TCCL이 할당되기 때문에 ws4j 계층에서는 ClassNotFoundException 오류가 발생하는 번들 A에서 필요한 리소스를 로드할 수 없습니다.

Singleton CXF 기본 버스를 생성하려면 "예제" 에서와 같이 서비스 오브젝트를 생성하는 기본 메서드 시작 부분에 이 코드를 삽입합니다.

```
BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
```

25.2.3. 예제

예 25.2. "서비스 오브젝트 생성" 예 24.7. "완전히 주석 처리 SEI" 에 표시된 SEI의 Service 오브젝트를 생성하는 코드를 보여줍니다.

예 25.2. 서비스 오브젝트 생성

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    BusFactory.setThreadDefaultBus(BusFactory.newInstance().createBus());
    QName serviceName = new QName("http://demo.redhat.com", "stockQuoteReporter");
    Service s = Service.create(serviceName);
   ...
  }
}
```

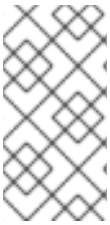
-

예 25.2. "서비스 오브젝트 생성" 의 코드는 다음을 수행합니다.

서비스의 모든 CXF 소비자가 사용할 수 있는 Singleton CXF 기본 버스를 생성합니다.

targetNamespace 속성과 @WebService 주석의 name 속성을 사용하여 서비스의 QName을 빌드합니다.

단일 매개변수 create() 메서드를 호출하여 새 Service 오브젝트를 생성합니다.



참고

단일 매개변수 create() 를 사용하면 WSDL 계약에 액세스하는 데 종속성을 확보할 수 없습니다.

25.3. 서비스에 포트 추가

25.3.1. 개요

서비스에 대한 엔드포인트 정보는 wsdl:port 요소에 정의되어 있으며 Service 오브젝트는 WSDL 계약에 정의된 각 엔드포인트에 대해 프록시 인스턴스를 생성합니다. Service 개체를 만들 때 WSDL 계약을 지정하지 않으면 서비스 오브젝트에 서비스를 구현하는 끝점에 대한 정보가 없으므로 프록시 인스턴스를 생성할 수 없습니다. 이 경우, addPort() 메서드를 사용하여 wsdl:port 요소를 나타내는 데 필요한 정보를 Service 오브젝트에 제공해야 합니다.

25.3.2. addPort() 메서드

Service 클래스는 예 25.3. "addPort() 메서드" 에 표시된 addPort() 메서드를 정의합니다. 이는 소비자 구현에서 사용할 수 있는 WSDL 계약이 없는 경우 사용됩니다. addPort() 메서드를 사용하면 Service 오브젝트에 정보를 제공할 수 있으며, 이는 일반적으로 서비스 구현에 대한 프록시를 생성하는 데 필요한 wsdl:port 요소에 저장됩니다.

예 25.3. addPort() 메서드

```
addPortQNameportNameStringbindingIdStringendpointAddressWebServiceException
```

portName 의 값은 **QName**입니다. 네임스페이스 부분의 값은 서비스의 대상 네임스페이스입니다. 서비스의 대상 네임스페이스는 **@WebService** 주석의 **targetNamespace** 속성에 지정됩니다. **QName**의 로컬 부분의 값은 **wsdl:port** 요소의 **name** 속성 값입니다. 다음 방법 중 하나로 이 값을 확인할 수 있습니다.

1. **@WebService** 주석의 **portName** 속성에 지정합니다.
2. **@WebService** 주석의 **name** 속성 값에 **Port** 를 추가합니다.
3. **SEI** 이름에 포트 를 추가합니다.

bindingId 매개변수 값은 끝점에서 사용하는 바인딩 유형을 고유하게 식별하는 문자열입니다. **SOAP** 바인딩의 경우 표준 **SOAP** 네임스페이스: **http://schemas.xmlsoap.org/soap/** 을 사용합니다. 끝점에서 **SOAP** 바인딩을 사용하지 않는 경우 **bindingId** 매개 변수의 값은 바인딩 개발자에 의해 결정됩니다. **endpointAddress** 매개변수 값은 끝점이 게시되는 주소입니다. **SOAP/HTTP** 엔드포인트의 경우 주소는 **HTTP** 주소입니다. **HTTP** 이외의 전송은 다른 주소 체계를 사용합니다.

25.3.3. 예제

예 25.4. "서비스 오브젝트에 포트 추가" **예 25.2. "서비스 오브젝트 생성"** 에서 생성된 **Service** 오브젝트에 포트를 추가하는 코드를 보여줍니다.

예 25.4. 서비스 오브젝트에 포트 추가

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
    QName portName = new QName("http://demo.redhat.com", "stockQuoteReporterPort");
    s.addPort(portName,
            "http://schemas.xmlsoap.org/soap/",
            "http://localhost:9000/StockQuote");
```

```
    ...
  }
}
```

예 25.4. "서비스 오브젝트에 포트 추가" 의 코드는 다음을 수행합니다.

portName 매개 변수의 QName을 생성합니다.

addPort() 메서드를 호출합니다.

끝점에서 SOAP 바인딩을 사용하도록 지정합니다.

끝점이 게시되는 주소를 지정합니다.

25.4. 끝점을 위한 프록시 가져오기

25.4.1. 개요

서비스 프록시는 원격 서비스에서 노출하는 모든 메서드를 제공하고 원격 호출에 필요한 모든 세부 정보를 처리하는 오브젝트입니다. Service 오브젝트는 getPort() 메서드를 통해 인식하는 모든 엔드포인트에 대해 서비스 프록시를 제공합니다. 서비스 프록시가 있으면 해당 메서드를 호출할 수 있습니다. 프록시는 서비스 계약에 지정된 연결 세부 정보를 사용하여 호출을 원격 서비스 엔드포인트로 전달합니다.

25.4.2. getPort() 메서드

예 25.5. "getPort() 메서드" 에 표시된 getPort() 메서드는 지정된 엔드포인트에 대한 서비스 프록시를 반환합니다. 반환된 프록시는 SEI와 동일한 클래스입니다.

예 25.5. getPort() 메서드

```
    public &lt;T&gt;
TgetPortQNameportNameClass<T>serviceEndpointInterfaceWebServiceException
```

portName 매개변수 값은 프록시가 생성되는 끝점을 정의하는 **wsdl:port** 요소를 식별하는 **QName**입니다. **serviceEndpointInterface** 매개 변수의 값은 **SEI**의 정규화된 이름입니다.

참고

WSDL 계약 없이 작동하는 경우 **portName** 매개 변수의 값은 일반적으로 **addPort()** 를 호출할 때 **portName** 매개변수에 사용되는 값과 동일합니다.

25.4.3. 예제

예 25.6. "서비스 프록시 가져오기" 예 25.4. "서비스 오브젝트에 포트 추가" 에 추가된 엔드포인트에 대한 서비스 프록시를 받기 위한 코드를 보여줍니다.

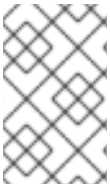
예 25.6. 서비스 프록시 가져오기

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    ...
    quoteReporter proxy = s.getPort(portName, quoteReporter.class);
    ...
  }
}
```

25.5. 소비자의 비즈니스 로직 구현

25.5.1. 개요

원격 끝점의 서비스 프록시를 인스턴스화하면 해당 메서드를 로컬 오브젝트인 것처럼 호출할 수 있습니다. 원격 메서드가 완료될 때까지 호출 블록입니다.



참고

메서드에 **@OneWay** 주석이 추가되면 호출이 즉시 반환됩니다.

25.5.2. 예제

예 25.7. "WSDL 계약 없이 고객 구현" 예 24.7. "완전히 주석 처리 SEI" 에 정의된 서비스의 소비자를 표시합니다.

예 25.7. WSDL 계약 없이 고객 구현

```
package com.fusesource.demo;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
  Service s = Service.create(serviceName);

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
  s.addPort(portName, "http://schemas.xmlsoap.org/soap/",
"http://localhost:9000/EricStockQuote");

  quoteReporter proxy = s.getPort(portName, quoteReporter.class);

  Quote quote = proxy.getQuote("ALPHA");
    System.out.println("Stock "+quote.getID()+" is worth "+quote.getVal()+" as of
"+quote.getTime());
  }
}
```

예 25.7. "WSDL 계약 없이 고객 구현" 의 코드는 다음을 수행합니다.

Service 오브젝트를 생성합니다.

Service 오브젝트에 끝점 정의를 추가합니다.

***Service** 오브젝트에서 서비스 프록시를 가져옵니다.*

서비스 프록시에서 작업을 호출합니다.

26장. STARTING POINT WSDL 계약

26.1. 샘플 WSDL 계약

예 26.1. "helloworld WSDL 계약" **HelloWorld WSDL** 계약을 보여줍니다. 이 계약은 **wsdl:portType** 요소에 단일 인터페이스인 **Greeter**를 정의합니다. 또한 계약은 **wsdl:port** 요소에서 서비스를 구현할 끝점을 정의합니다.

예 26.1. helloworld WSDL 계약

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified">
      <element name="sayHiResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMe">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeResponse">
        <complexType>
          <sequence>
            <element name="responseType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="greetMeOneWay">
        <complexType>
          <sequence>
            <element name="requestType" type="string"/>
          </sequence>
        </complexType>
      </element>
      <element name="pingMe">
```

```
      <complexType/>
    </element>
    <element name="pingMeResponse">
      <complexType/>
    </element>
    <element name="faultDetail">
      <complexType>
        <sequence>
          <element name="minor" type="short"/>
          <element name="major" type="short"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>

<wsdl:message name="sayHiRequest">
  <wsdl:part element="x1:sayHi" name="in"/>
</wsdl:message>
<wsdl:message name="sayHiResponse">
  <wsdl:part element="x1:sayHiResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeRequest">
  <wsdl:part element="x1:greetMe" name="in"/>
</wsdl:message>
<wsdl:message name="greetMeResponse">
  <wsdl:part element="x1:greetMeResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeOneWayRequest">
  <wsdl:part element="x1:greetMeOneWay" name="in"/>
</wsdl:message>
<wsdl:message name="pingMeRequest">
  <wsdl:part name="in" element="x1:pingMe"/>
</wsdl:message>
<wsdl:message name="pingMeResponse">
  <wsdl:part name="out" element="x1:pingMeResponse"/>
</wsdl:message>
<wsdl:message name="pingMeFault">
  <wsdl:part name="faultDetail" element="x1:faultDetail"/>
</wsdl:message>

<wsdl:portType name="Greeter">
  <wsdl:operation name="sayHi">
    <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
    <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
  </wsdl:operation>

  <wsdl:operation name="greetMe">
    <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
    <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
  </wsdl:operation>

  <wsdl:operation name="greetMeOneWay">
    <wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>
  </wsdl:operation>
```

```
    <wsdl:operation name="pingMe">
      <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
      <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
      <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
    ...
  </wsdl:binding>

  <wsdl:service name="SOAPService">
    <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
      <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

*예 26.1. “helloworld WSDL 계약” 에 정의된 **Greeter** 인터페이스는 다음 작업을 정의합니다.*

***sayHi - xsd:string** 의 단일 출력 매개 변수가 있습니다.*

***greetMe - xsd:string** 및 출력 매개 변수 **xsd:string** 의 입력 매개 변수를 포함합니다.*

***greetMeOneWay - xsd:string** 의 단일 입력 매개 변수를 포함합니다. 이 작업에는 출력 매개 변수가 없으므로 단방향 호출로 최적화되어 있습니다(즉, 소비자는 서버의 응답을 기다리지 않습니다).*

***PingMe -** 입력 매개변수가 없고 출력 매개변수가 없지만 오류 예외가 발생할 수 있습니다.*

27장. TOP-DOWN 서비스 개발

초록

서비스 공급자를 개발하는 최상위 방법에서는 서비스 공급자가 구현할 작업 및 방법을 정의하는 WSDL 문서에서 시작합니다. WSDL 문서를 사용하여 서비스 공급자에 대한 시작점 코드를 생성합니다. 생성된 코드에 비즈니스 논리를 추가하는 것은 일반적인 Java 프로그래밍 API를 사용하여 수행됩니다.

27.1. JAX-WS 서비스 공급자 개발 개요

WSDL 문서가 있으면 다음과 같이 JAX-WS 서비스 공급자를 개발하는 프로세스는 다음과 같습니다.

1. 27.2절. "시작 지점 코드 생성" 포인트 코드 시작.
2. 서비스 공급자의 작업을 구현 합니다.
3. 31장. 서비스 게시 구현된 서비스.

27.2. 시작 지점 코드 생성

27.2.1. 개요

JAX-WS는 WSDL에 정의된 서비스에서 해당 서비스를 서비스 공급자로 구현할 Java 클래스에 대한 자세한 매핑을 지정합니다. wsdl:portType 요소에서 정의하는 논리 인터페이스는 서비스 끝점 인터페이스(SEI)에 매핑됩니다. WSDL에 정의된 모든 복잡한 유형은 JAXB(Java Architecture for XML Binding) 사양에 정의된 매핑에 따라 Java 클래스에 매핑됩니다. wsdl:service 요소에서 정의한 끝점도 소비자가 서비스를 구현하는 서비스 프로바이더에 액세스하는 데 사용하는 Java 클래스에 생성됩니다.

cxf-codegen-plugin Maven 플러그인이 이 코드를 생성합니다. 또한 구현을 위한 시작점 코드를 생성하는 옵션을 제공합니다. 코드 생성기는 생성된 코드를 제어하기 위한 다양한 옵션을 제공합니다.

27.2.2. 코드 생성기 실행

예 27.1. "서비스 코드 생성" 코드 생성기를 사용하여 서비스에 대한 시작점 코드를 생성하는 방법을 보여줍니다.

예 27.1. 서비스 코드 생성

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>wsdl</wsdl>
            <extraargs>
              <extraarg>-server</extraarg>
              <extraarg>-impl</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

이 작업은 다음을 수행합니다.

- ***-impl** 옵션은 **WSDL** 계약의 각 **wsdl:portType** 요소에 대해 셸 구현 클래스를 생성합니다.*
- *서버 옵션은 서비스 공급자를 독립 실행형 애플리케이션으로 실행하는 간단한 **main()** 을 생성합니다.*
- ***sourceRoot** 는 생성된 코드가 **outputDir** 이라는 디렉터리에 작성되도록 지정합니다.*
- ***WSDL** 요소는 코드가 생성되는 **WSDL** 계약을 지정합니다.*

코드 생성기에 대한 전체 옵션 목록은 44.2절. “cxf-codegen-plugin” 에서 참조하십시오.

27.2.3. 생성된 코드

표 27.1. "서비스 공급자를 위한 생성된 클래스" 서비스 공급자를 생성하기 위해 생성된 파일을 설명합니다.

표 27.1. 서비스 공급자를 위한 생성된 클래스

파일	설명
***portTypeName*.java**	SEI입니다. 이 파일에는 서비스 공급자가 구현하는 인터페이스가 포함되어 있습니다. 이 파일을 편집해서는 안 됩니다.
***serviceName*.java**	엔드포인트입니다. 이 파일에는 소비자가 서비스에 요청하는 데 사용하는 Java 클래스가 포함되어 있습니다.
***portTypeName*Impl.java**	skeleton 구현 클래스입니다. 이 파일을 수정하여 서비스 공급자를 빌드합니다.
***portTypeName*Server.java**	서비스 공급자를 독립형 프로세스로 배포할 수 있는 기본 서버 메인 라인입니다. 자세한 내용은 31장. *서비스 게시* 에서 참조하십시오.

또한 코드 생성기는 **WSDL** 계약에 정의된 모든 형식에 대해 **Java** 클래스를 생성합니다.

27.2.4. 생성된 패키지

생성된 코드는 **WSDL** 계약에 사용되는 네임스페이스를 기반으로 패키지에 배치됩니다. 서비스를 지원하도록 생성된 클래스(**wsdl:portType** 요소, **wsdl:service** 요소 및 **wsdl:port** 요소)는 **WSDL** 계약의 대상 네임스페이스를 기반으로 하는 패키지에 배치됩니다. 계약의 형식 요소에 정의된 형식을 구현하기 위해 생성된 클래스는 **types** 요소의 **targetNamespace** 특성을 기반으로 패키지에 배치됩니다.

매핑 알고리즘은 다음과 같습니다.

1. 주요 **http://** 또는 **urn://** 는 네임스페이스를 제거합니다.
2. 네임스페이스의 첫 번째 문자열이 유효한 인터넷 도메인인 경우 예를 들어 **.com** 또는 **.gov** **.gov** 로 끝나는 경우 선행 **www.** 는 문자열에서 제거되며 나머지 두 구성 요소는 리플링됩니다.
3. 네임스페이스의 마지막 문자열이 **.xxx** 또는 **.xx** 패턴의 파일 확장자로 끝나는 경우 확장은 제

거됩니다.

4. 네임스페이스의 나머지 문자열이 결과 문자열에 추가되고 점으로 구분됩니다.

5. 모든 편지는 소문자로 되어 있습니다.

27.3. 서비스 공급자 구현

27.3.1. 구현 코드 생성

코드 생성기 **-impl** 플래그를 사용하여 서비스 공급자를 빌드하는 데 사용되는 구현 클래스를 생성합니다.

참고

서비스의 계약에 **XML** 스키마에 정의된 사용자 지정 형식이 포함된 경우 해당 유형의 클래스가 생성되어 사용 가능한지 확인해야 합니다.

코드 생성기 사용에 대한 자세한 내용은 44.2절. “cxf-codegen-plugin” 을 참조하십시오.

27.3.2. 생성된 코드

구현 코드는 두 개의 파일로 구성됩니다.

- **portTypeName.java -** 서비스의 서비스 인터페이스(**SEI**)입니다.
- **portTypeNameImpl.java -** 서비스에서 정의한 작업을 구현하는 데 사용할 클래스입니다.

27.3.3. 작업의 논리 구현

서비스 작업에 대한 비즈니스 논리를 제공하려면 **portTypeNameImpl.java** 의 스텁 메서드를 완료합니다. 일반적으로 표준 **Java**를 사용하여 비즈니스 논리를 구현합니다. 서비스에서 사용자 지정 **XML** 스키마 유형을 사용하는 경우 각 형식에 대해 생성된 클래스를 사용하여 조작해야 합니다. 일부 고급 기능에 액세스하는 데 사용할 수 있는 일부 **Apache CXF** 특정 **API**도 있습니다.

27.3.4. 예제

예를 들어 예 26.1. “helloworld WSDL 계약” 에 정의된 서비스의 구현 클래스는 예 27.2. “Greeter 서비스 구현” 로 표시될 수 있습니다. 굵게 표시된 코드 부분만 프로그래머가 삽입해야 합니다.

예 27.2. Greeter 서비스 구현

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")

public class GreeterImpl implements Greeter {

    public String greetMe(String me) {
        System.out.println("Executing operation greetMe"); System.out.println("Message received: " +
me + "\n"); return "Hello " + me;
    }

    public void greetMeOneWay(String me) {
        System.out.println("Executing operation greetMeOneWay\n"); System.out.println("Hello there
" + me);
    }

    public String sayHi() {
        System.out.println("Executing operation sayHi\n"); return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        FaultDetail faultDetail = new FaultDetail(); faultDetail.setMajor((short)2);
faultDetail.setMinor((short)1); System.out.println("Executing operation pingMe, throwing
PingMeFault exception\n"); throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
}
```

28장. WSDL 계약에서 소비자 개발

초록

소비자를 생성하는 한 가지 방법은 **WSDL** 계약에서 시작하는 것입니다. 계약은 소비자가 요청하는 서비스의 운영, 메시지 및 전송 세부 사항을 정의합니다. 소비자의 시작점 코드는 **WSDL** 계약에서 생성됩니다. 소비자에 필요한 기능이 생성된 코드에 추가됩니다.

28.1. STUB 코드 생성

28.1.1. 개요

cxf-codegen-plugin Maven 플러그인은 **WSDL** 계약에서 스텁 코드를 생성합니다. 스텁 코드는 원격 서비스에서 작업을 호출하는 데 필요한 지원 코드를 제공합니다.

소비자의 경우 **cxf-codegen-plugin Maven** 플러그인이 다음과 같은 유형의 코드를 생성합니다.

- 스텁 코드 **-** 소비자 구현을 위해 파일을 지원합니다.
- 포인트 코드 시작 **-** 원격 서비스에 연결하고 원격 서비스에서 모든 작업을 호출하는 샘플 코드입니다.

28.1.2. 소비자 코드 생성

소비자 코드를 생성하려면 **cxf-codegen-plugin Maven** 플러그인을 사용합니다. 예 28.1. “소비자 코드 생성” 코드 생성기를 사용하여 소비자 코드를 생성하는 방법을 보여 줍니다.**Shows how to use the code generator to generate consumer code.**

> 예 28.1. 소비자 코드 생성
>
> ```
> <plugin>
>   <groupId>org.apache.cxf</groupId>
>   <artifactId>cxf-codegen-plugin</artifactId>
>   <version>${cxf.version}</version>
>   <executions>
>     <execution>
>       <id>generate-sources</id>
> ```

```
        <phase>generate-sources</phase>
        <configuration>
          <sourceRoot>outputDir</sourceRoot>
          <wsdlOptions>
            <wsdlOption>
              <wsdl>wsdl</wsdl>
              <extraargs>
                <extraarg>-client</extraarg>
              </extraargs>
            </wsdlOption>
          </wsdlOptions>
        </configuration>
        <goals>
          <goal>wsdl2java</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

*여기서 **outputDir** 은 생성된 파일이 배치되는 디렉터리의 위치이며 **wsdl** 은 **WSDL** 계약의 위치를 지정합니다. **-client** 옵션은 소비자의 **main()** 메서드에 대한 시작점 코드를 생성합니다.*

***cxf-codegen-plugin Maven** 플러그인에 사용할 수 있는 전체 인수 목록은 44.2절. “cxf-codegen-plugin” 에서 참조하십시오.*

28.1.3. 생성된 코드

*코드 생성 플러그인은 예 26.1. “helloworld WSDL 계약” 에 표시된 계약에 대해 다음 **Java** 패키지를 생성합니다.*

- ***org.apache.hello_world_soap_http** - 이 패키지는 **http://apache.org/hello_world_soap_http** 대상 네임스페이스에서 생성됩니다. 이 네임스페이스에 정의된 모든 **WSDL** 엔티티(예: **Greeter** 포트 유형 및 **SOAPService** 서비스)는 이 **Java** 클래스에 매핑됩니다.*
- ***org.apache.hello_world_soap_http.types** - 이 패키지는 **http://apache.org/hello_world_soap_http/types** 대상 네임스페이스에서 생성됩니다. 이 네임스페이스에 정의된 모든 **XML** 유형(즉, **HelloWorld** 계약의 **wsdl:types** 요소에 정의된 모든 항목)은 이 **Java** 패키지의 **Java** 클래스에 매핑됩니다.*

***cxf-codegen-plugin Maven** 플러그인에서 생성된 스텁 파일은 다음 카테고리로 대체됩니다.*

- *org.apache.hello_world_soap_http 패키지의 WSDL 엔터티를 나타내는 클래스입니다. 다음 클래스는 WSDL 엔터티를 표현하기 위해 생성됩니다.*
  - *greeter - Greeter wsdl:portType 요소를 나타내는 Java 인터페이스입니다. JAX-WS 용어에서 이 Java 인터페이스는 서비스 엔드포인트 인터페이스(SEI)입니다.*
  - *SOAPService - SOAPService wsdl:service 요소를 나타내는 Java 서비스 클래스 (extending javax.xml.ws.Service)입니다.*
  - *PingMeFault - pingMeFault wsdl:fault 요소를 나타내는 Java 예외 클래스 (extending java.lang.Exception)입니다.*
- *org.objectweb.hello_world_soap_http.types 패키지에서 XML 유형을 나타내는 클래스입니다. HelloWorld 예에서 생성된 유일한 유형은 요청 및 응답 메시지에 대한 다양한 래퍼입니다. 이러한 데이터 유형 중 일부는 비동기 호출 모델에 유용합니다.*

28.2. 소비자 구현

28.2.1. 개요

WSDL 계약에서 시작할 때 소비자를 구현하려면 다음 스텁을 사용해야 합니다.

- *서비스 클래스*
- *SEI*

이러한 스텁을 사용하여 소비자 코드는 서비스 프록시를 인스턴스화하여 원격 서비스에 요청합니다. 또한 소비자의 비즈니스 논리를 구현합니다.

28.2.2. 생성된 서비스 클래스

예 28.2. "생성된 서비스 클래스 개요" 생성된 서비스 클래스의 일반적인 개요를 보여줍니다, ServiceName_Service[2]javax.xml.ws.Service 기본 클래스를 확장하는 .

예 28.2. 생성된 서비스 클래스 개요

```
@WebServiceClient(name="..." targetNamespace="..."
                  wsdlLocation="...")
public class ServiceName extends javax.xml.ws.Service
{
  ...
  public ServiceName(URL wsdlLocation, QName serviceName) { }

  public ServiceName() { }

  // Available only if you specify '-fe cxf' option in wsdl2java
  public ServiceName(Bus bus) { }

  @WebEndpoint(name="...")
  public SEI getPortName() { }
  .
  .
  .
}
```

예 28.2. "생성된 서비스 클래스 개요" 의 **ServiceName** 클래스는 다음 메서드를 정의합니다.

- **ServiceName (URL wsdlLocation, QName serviceName) - wsdl:service** 요소에 따라 **wsdlLocation** 서비스에서 얻을 수 있는 **WSDL** 계약과 함께 서비스 오브젝트를 구성합니다.

- **ServiceName() -** 기본 생성자입니다. 스텁 코드가 생성된 시간(예: **wsdl2java** 툴을 실행하는 경우)에 제공된 서비스 이름 및 **WSDL** 계약을 기반으로 서비스 개체를 구성합니다. 이 생성자를 사용하면 **WSDL** 계약이 지정된 위치에서 계속 사용 가능함을 전제로 합니다.

- **ServiceName(Bus bus) - (CXF** 특정) 서비스를 구성하는 데 사용되는 버스 인스턴스를 지정할 수 있는 추가 생성자입니다. 이 기능은 다중 버스 인스턴스를 서로 다른 스레드와 연결할 수 있는 다중 스레드 애플리케이션의 컨텍스트에서 유용할 수 있습니다. 이 생성자는 사용자가 지정하는 버스가 이 서비스에 사용되는지 확인하는 간단한 방법을 제공합니다. **wsdl2java** 툴을 호출할 때 **-fe cxf** 옵션을 지정하는 경우에만 사용 가능합니다.

- **getPortName() - PortName** 과 같은 **name** 속성이 있는 **wsdl:port** 요소에서 정의한 엔드포인트에 대한 프록시를 반환합니다. **getter** 메서드는 **ServiceName** 서비스에서 정의한 모든 **wsdl:port** 요소에 대해 생성됩니다. 여러 끝점 정의가 포함된 **wsdl:service** 요소에는 여러 **getPortName()** 메서드가 있는 생성된 서비스 클래스가 생성됩니다.

28.2.3. 서비스 엔드 포인트 인터페이스

원래의 **WSDL** 계약에 정의된 모든 인터페이스의 경우 해당 **SEI**를 생성할 수 있습니다. 서비스 엔드포인트 인터페이스는 **wsdl:portType** 요소의 **Java** 매핑입니다. 원래 **wsdl:portType** 요소에 정의된 각 작업은 **SEI**의 해당 메서드에 매핑됩니다. 작업의 매개변수는 다음과 같이 매핑됩니다. . 입력 매개변수는 메서드 인수에 매핑됩니다.

1. 첫 번째 출력 매개 변수는 반환 값에 매핑됩니다.

2. 출력 매개 변수가 두 개 이상 있는 경우 두 번째 및 후속 출력 매개 변수는 메서드 인수에 매핑됩니다(더 이상 이러한 인수의 값은 **holder** 유형을 사용하여 전달해야 함).

예를 들어 예 28.3. “Greeter Service Endpoint Interface” 은 예 26.1. “helloworld WSDL 계약” 에 정의된 **wsdl:portType** 요소에서 생성된 **Greeter SEI**를 표시합니다. 간단히 하기 위해 예 28.3. “Greeter Service Endpoint Interface” 는 표준 **JAXB** 및 **JAX-WS** 주석을 생략합니다.

예 28.3. Greeter Service Endpoint Interface

```
package org.apache.hello_world_soap_http;
  ...
public interface Greeter
{
  public String sayHi();
  public String greetMe(String requestType);
  public void greetMeOneWay(String requestType);
  public void pingMe() throws PingMeFault;
}
```

28.2.4. 소비자 주요 기능

예 28.4. “소비자 구현 코드” **HelloWorld** 소비자를 구현하는 코드를 보여줍니다. 소비자는 **SOAPService** 서비스의 **SoapPort** 포트에 연결한 다음 **Greeter** 포트 유형에서 지원하는 각 작업을 계속 호출합니다.

예 28.4. 소비자 구현 코드

```
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.SOAPService;

public final class Client {
```

```
 private static final QName SERVICE_NAME =
 new QName("http://apache.org/hello_world_soap_http",
          "SOAPService");

 private Client()
 {
 }

 public static void main(String args[]) throws Exception
 {
if (args.length == 0)
  {
    System.out.println("please specify wsdl");
    System.exit(1);
  }

URL wsdlURL;
  File wsdlFile = new File(args[0]);
  if (wsdlFile.exists())
  {
    wsdlURL = wsdlFile.toURL();
  }
  else
  {
    wsdlURL = new URL(args[0]);
  }

  System.out.println(wsdlURL);
SOAPService ss = new SOAPService(wsdlURL,SERVICE_NAME);
Greeter port = ss.getSoapPort();
  String resp;

System.out.println("Invoking sayHi...");
  resp = port.sayHi();
  System.out.println("Server responded with: " + resp);
  System.out.println();

  System.out.println("Invoking greetMe...");
  resp = port.greetMe(System.getProperty("user.name"));
  System.out.println("Server responded with: " + resp);
  System.out.println();

  System.out.println("Invoking greetMeOneWay...");
  port.greetMeOneWay(System.getProperty("user.name"));
  System.out.println("No response from server as method is OneWay");
  System.out.println();

try {
    System.out.println("Invoking pingMe, expecting exception...");
    port.pingMe();
  } catch (PingMeFault ex) {
    System.out.println("Expected exception: PingMeFault has occurred.");
    System.out.println(ex.toString());
  }
```

```
        System.exit(0);
    }
}
```

예 28.4. "소비자 구현 코드" 의 client.main() 방법은 다음과 같이 진행됩니다.

Apache CXF 런타임 클래스가 클래스 경로에 있는 경우 런타임은 암시적으로 초기화됩니다. Apache CXF를 초기화하기 위해 특수 함수를 호출할 필요가 없습니다.

소비자는 HelloWorld에 대한 WSDL 계약의 위치를 제공하는 단일 문자열 인수가 필요합니다. WSDL 계약의 위치는 wsdlURL 에 저장됩니다.

WSDL 계약의 위치 및 서비스 이름이 필요한 생성자를 사용하여 서비스 오브젝트를 생성합니다. 적절한 getPortName() 메서드를 호출하여 필요한 포트의 인스턴스를 가져옵니다. 이 경우 SOAPService 서비스는 Greeter 서비스 엔드포인트 인터페이스를 구현하는 SoapPort 포트만 지원합니다.

소비자는 Greeter 서비스 엔드포인트 인터페이스에서 지원하는 각 메서드를 호출합니다.

pingMe() 메서드의 경우 예제 코드는 PingMeFault fault 예외를 catch하는 방법을 보여줍니다.

28.2.5. -fe cxf 옵션으로 생성된 클라이언트 프록시

wsdl2java에서 -fe cxf 옵션을 지정하여 클라이언트 프록시를 생성하는 경우(cxf frontend를 선택하여) 생성된 클라이언트 프록시 코드가 Java 7과 더 잘 통합됩니다. 이 경우 getServiceNamePort() 메서드를 호출하면 SEI의 하위 인터페이스인 형식을 반환하고 다음과 같은 추가 인터페이스를 구현합니다.

- java.lang.AutoCloseable
- javax.xml.ws.BindingProvider (JAX-WS 2.0)
- org.apache.cxf.endpoint.Client

이를 통해 클라이언트 프록시 작업을 단순화하는 방법을 보려면 표준 JAX-WS 프록시 오브젝트를 사용하여 작성된 다음 Java 코드 샘플을 고려하십시오.

```
// Programming with standard JAX-WS proxy object
//
(ServiceNamePortType port = service.getServiceNamePort();
((BindingProvider)port).getRequestContext()
        .put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
port.serviceMethod(...);
((Closeable)port).close();
```

*또한 **cxf frontend**에서 생성된 코드를 사용하여 작성한 다음 동등한 코드 샘플과 위 코드를 비교합니다.*

```
// Programming with proxy generated using '-fe cxf' option
//
try (ServiceNamePortTypeProxy port = service.getServiceNamePort()) {
    port.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, address);
    port.serviceMethod(...);
}
```

---

[2]
***wsdl:service** 요소의 **name** 속성이 **Service**에서 종료되면 **_Service**가 사용되지 않습니다.*

29장. 런타임 시 WSDL 찾기

초록

WSDL 문서의 위치를 애플리케이션으로 하드 코딩하는 것은 확장할 수 없습니다. 실제 배포 환경에서는 런타임에 **WSDL** 문서의 위치가 해결되도록 해야 합니다. **Apache CXF**는 이를 가능하게 하는 다양한 도구를 제공합니다.

29.1. WSDL 문서를 찾는 메커니즘

JAX-WS API를 사용하여 소비자를 개발할 때는 서비스를 정의하는 **WSDL** 문서에 대한 하드 코딩된 경로를 제공해야 합니다. 이는 소규모 환경에서 **OK**이지만 하드 코딩된 경로를 사용하면 엔터프라이즈 배포에서 제대로 작동하지 않습니다.

이 문제를 해결하기 위해 **Apache CXF**는 하드 코딩된 경로를 사용하는 요구 사항을 제거하기 위한 세 가지 메커니즘을 제공합니다.

- 29.2절. “주입을 통해 프록시 인스턴스화”
- 29.3절. “JAX-WS Catalog 사용”
- 29.4절. “계약 확인 방법 사용”

참고

구현 코드에 프록시를 삽입하는 것이 가장 쉬운 방법이므로 일반적으로 가장 좋은 옵션입니다. 서비스 프록시를 삽입하고 인스턴스화하기 위해서는 클라이언트 끝점과 구성 파일만 필요합니다.

29.2. 주입을 통해 프록시 인스턴스화

29.2.1. 개요

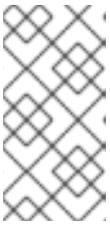
Apache CXF의 **Spring Framework**를 사용하면 **JAX-WS API**를 사용하여 서비스 프록시를 생성하는 번거로움을 피할 수 있습니다. 구성 파일에 클라이언트 끝점을 정의한 다음 구현 코드에 프록시를 직접 삽입할 수 있습니다. 런타임이 구현 오브젝트를 인스턴스화할 때 구성에 따라 외부 서비스에 대한 프록시를 인스턴스화합니다. 구현은 인스턴스화된 프록시에 대한 참조로 제공됩니다.

프록시는 구성 파일의 정보를 사용하여 인스턴스화되므로 **WSDL** 위치는 하드 코딩할 필요가 없습니다. 배포 시 변경할 수 있습니다. 런타임이 애플리케이션의 **classpath**에서 **WSDL**을 검색하도록 지정할 수도 있습니다.

29.2.2. 절차

외부 서비스의 프록시를 서비스 공급자의 구현에 삽입하려면 다음을 수행합니다.

1. 애플리케이션의 모든 부분에서 액세스할 수 있는 잘 알려진 위치에 필요한 **WSDL** 문서를 배포합니다.



참고

애플리케이션을 **WAR** 파일로 배포하는 경우 **WAR**의 **WEB-INF/wsdl** 폴더에 모든 **WSDL** 문서 및 **XML** 스키마 문서를 배치하는 것이 좋습니다.

참고

애플리케이션을 **JAR** 파일로 배포하는 경우 **JAR**의 **META-INF/wsdl** 폴더에 모든 **WSDL** 문서 및 **XML** 스키마 문서를 배치하는 것이 좋습니다.

2. 삽입되는 프록시에 대해 **JAX-WS** 클라이언트 끝점을 구성합니다.

3. **@Resource** 주석을 사용하여 제공하는 서비스에 프록시를 삽입합니다. **???**

29.2.3. 프록시 구성

애플리케이션의 구성 파일에서 **jaxws:client** 요소를 사용하여 **JAX-WS** 클라이언트 엔드포인트를 구성합니다. 이렇게 하면 런타임에 **org.apache.cxf.jaxws.JaxWsClientProxy** 개체를 지정된 속성을 사용하여 인스턴스화합니다. 이 오브젝트는 서비스 공급자에 삽입될 프록시입니다.

최소한 다음 속성에 대한 값을 제공해야 합니다.

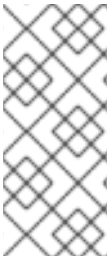
- **id-** 삽입할 클라이언트를 식별하는 데 사용되는 **ID**를 지정합니다.

- **serviceClass**는 프록시가 요청하는 서비스의 **SEI**를 지정합니다.

예 29.1. "프록시가 서비스 구현에 삽입될 수 있는 구성" **JAX-WS** 클라이언트 끝점에 대한 구성을 보여줍니다.

예 29.1. 프록시가 서비스 구현에 삽입될 수 있는 구성

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:client id="bookClient"
                serviceClass="org.apache.cxf.demo.BookService"
                wsdlLocation="classpath:books.wsdl"/>
  ...
</beans>
```



참고

예 29.1. "프록시가 서비스 구현에 삽입될 수 있는 구성" 에서 **wsdlLocation** 특성은 런타임이 **classpath**에서 **WSDL**을 로드하도록 지시합니다. **books.wsdl** 이 **classpath**에 있는 경우 런타임에서 해당 경로를 찾을 수 있습니다.

JAX-WS 클라이언트 구성에 대한 자세한 내용은 *17.2절. "소비자 엔드 포인트 구성"* 을 참조하십시오.

29.2.4. 공급자 구현 코딩

예 29.2. "서비스 구현에 프록시 삽입" 에 표시된 대로 **@Resource** 를 사용하여 서비스 구현에 구성된 프록시를 서비스 구현에 리소스로 삽입합니다.

예 29.2. 서비스 구현에 프록시 삽입

```
package demo.hw.server;

import org.apache.hello_world_soap_http.Greeter;

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")
public class StoreImpl implements Store {
```

```
@Resource(name="bookClient") private BookService proxy;

}
```

주석의 **name** 속성은 **JAX-WS** 클라이언트의 **id** 속성 값에 해당합니다. 구성된 프록시는 주석 직후 선언된 **BookService** 오브젝트에 삽입됩니다. 이 오브젝트를 사용하여 프록시의 외부 서비스에서 호출을 수행할 수 있습니다.

29.3. JAX-WS CATALOG 사용

29.3.1. 개요

JAX-WS 사양은 모든 구현이 다음을 지원해야 합니다.

웹 서비스, 특히 **WSDL** 및 **XML** 스키마 문서의 설명에 포함된 웹 서비스 문서를 해결할 때 사용할 표준 카탈로그 기능입니다.

이 카탈로그 기능은 **OASIS**에서 지정한 **XML** 카탈로그 기능을 사용합니다. **WSDL URI**를 사용하는 모든 **JAX-WS API** 및 주석은 이 카탈로그를 사용하여 **WSDL** 문서의 위치를 확인합니다.

즉, **WSDL** 문서의 위치를 특정 배포 환경에 맞게 다시 작성하는 **XML** 카탈로그 파일을 제공할 수 있습니다.

29.3.2. 카탈로그 작성

JAX-WS 카탈로그는 **OASIS XML Catalogs 1.1** 사양에 정의된 표준 **XML** 카탈로그입니다. 매핑을 지정할 수 있습니다.

- **URI**에 대한 문서의 공개 식별자 및/또는 시스템 식별자입니다.
- 다른 **URI**에 대한 리소스의 **URI**입니다.

표 **29.1. "일반 JAX-WS 카탈로그 요소"** **WSDL** 위치 확인에 사용되는 몇 가지 일반적인 요소를 나열합니다.

표 29.1. 일반 JAX-WS 카탈로그 요소

요소	설명
uri	URI를 대체 URI에 매핑합니다.
rewriteURI	URI의 시작을 다시 작성합니다. 예를 들어 이 요소를 사용하면 http://cxf.apache.org 로 시작하는 모든 URI를 classpath로 시작하는 URI에 매핑할 수 있습니다.
uriSuffix	원래 URI의 접미사에 따라 URI를 대체 URI에 매핑합니다. 예를 들어 foo.xsd로 끝나는 모든 URI를 **classpath:foo.xsd** 에 매핑할 수 있습니다.

29.3.3. 카탈로그 패키지

JAX-WS 사양은 **WSDL** 및 **XML** 스키마 문서를 해결하는 데 사용되는 카탈로그가 **META-INF/jax-ws-catalog.xml** 이라는 사용 가능한 모든 리소스를 사용하여 어셈블해야 합니다. 애플리케이션이 단일 **JAR** 또는 **WAR**에 패키지된 경우 카탈로그를 단일 파일에 배치할 수 있습니다.

애플리케이션이 여러 **JAR**로 패키징된 경우 카탈로그를 여러 파일로 분할할 수 있습니다. 각 카탈로그 파일은 특정 **JAR**의 코드에서 액세스하는 **WSDL**만 처리하도록 모듈화할 수 있었습니다.

29.4. 계약 확인 방법 사용

29.4.1. 개요

런타임에 **WSDL** 문서 위치를 해결하기 위한 가장 관련된 메커니즘은 사용자 정의 계약 확인자를 구현하는 것입니다. 이를 위해서는 **Apache CXF** 특정 **ServiceContractResolver** 인터페이스를 구현해야 합니다. 또한 버스에 사용자 정의 리졸버를 등록해야 합니다.

적절하게 등록하면 사용자 정의 계약 확인자는 필요한 **WSDL** 및 스키마 문서의 위치를 해결하는 데 사용됩니다.

29.4.2. 계약 확인 방법 구현

계약 확인자는 **org.apache.cxf.endpoint.ServiceContractResolver** 인터페이스의 구현입니다. *예 29.3. "ServiceContractResolver Interface"* 에 표시된 대로 이 인터페이스에는 구현해야 하는 단일 메서드 **getContractLocation()** 이 있습니다. **getContractLocation()** 은 서비스의 **QName**을 가져와 서비스의 **WSDL** 계약 **URI**를 반환합니다.

예 29.3. ServiceContractResolver Interface

```
public interface ServiceContractResolver
{
  URI getContractLocation(QName qname);
}
```

WSDL 계약의 위치를 해결하는 데 사용되는 논리는 애플리케이션별입니다. **UDDI** 레지스트리, 데이터베이스, 파일 시스템의 사용자 지정 위치 또는 선택한 기타 메커니즘에서 계약 위치를 확인하는 논리를 추가할 수 있습니다.

29.4.3. 계약 확인 프로그램을 프로그래밍 방식으로 등록

Apache CXF 런타임에서 계약 확인자를 사용하기 전에 계약 확인자 레지스트리에 등록해야 합니다. 계약 확인자 레지스트리는 **org.apache.cxf.endpoint.ServiceContractResolverRegistry** 인터페이스를 구현합니다. 그러나 자체 레지스트리를 구현할 필요는 없습니다. **Apache CXF**는 **org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl** 클래스에서 기본 구현을 제공합니다.

기본 레지스트리에 계약 확인자를 등록하려면 다음을 수행합니다.

1. 기본 버스 오브젝트에 대한 참조를 가져옵니다.
2. 버스의 **getExtension()** 메서드를 사용하여 버스에서 서비스 계약 레지스트리를 가져옵니다.
3. 계약 확인자의 인스턴스를 만듭니다.
4. 레지스트리의 **register()** 메서드를 사용하여 레지스트리에 계약 확인자를 등록합니다.

예 29.4. “계약 해결 관리자 등록” 은 계약 확인자를 기본 레지스트리로 등록하는 코드를 보여줍니다.

예 29.4. 계약 해결 관리자 등록

```
BusFactory bf=BusFactory.newInstance();
Bus bus=bf.createBus();

ServiceContractResolverRegistry registry = bus.getExtension(ServiceContractResolverRegistry);

JarServiceContractResolver resolver = new JarServiceContractResolver();

registry.register(resolver);
```

-

예 29.4. "계약 해결 관리자 등록"의 코드는 다음을 수행합니다.

버스 인스턴스를 가져옵니다.

버스 계약 확인자 레지스트리를 가져옵니다.

계약 확인자의 인스턴스를 생성합니다.

레지스트리에 계약 확인자를 등록합니다.

29.4.4. 구성을 사용하여 계약 확인자 등록

*구성을 통해 클라이언트에 추가할 수 있도록 계약 확인자를 구현할 수도 있습니다. 계약 확인자는 런타임에서 구성을 읽고 확인자 자체를 인스턴스화하는 방식으로 구현됩니다. 런타임에서 초기화를 처리하므로 클라이언트가 계약 확인자를 사용해야 하는 경우 런타임에 결정할 수 있습니다.***Because the runtime handles the initialization, you can decide at runtime if a client needs to use the contract resolver.**

구성을 통해 클라이언트에 추가할 수 있도록 계약 확인자를 구현하려면 다음을 수행합니다.

1. *계약 확인자 구현에 **init()** 메서드를 추가합니다.*
2. *예 29.4. "계약 해결 관리자 등록"과 같이 계약 확인자 레지스트리를 사용하여 계약 확인자를 등록하는 **init()** 메서드에 논리를 추가합니다.*
3. ***@PostConstruct** 주석을 사용하여 **init()** 메서드를 장식합니다.*

예 29.5. "구성을 사용하여 등록할 수 있는 서비스 계약 해결" 구성을 사용하여 클라이언트에 추가할 수 있는 계약 확인자 구현을 보여줍니다.

예 29.5. 구성을 사용하여 등록할 수 있는 서비스 계약 해결

```
import javax.annotation.PostConstruct;
```

```
import javax.annotation.Resource;
import javax.xml.namespace.QName;

import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;

public class UddiResolver implements ServiceContractResolver
{
  private Bus bus;
  ...

  @PostConstruct
  public void init()
  {
    BusFactory bf=BusFactory.newInstance();
    Bus bus=bf.createBus();
    if (null != bus)
    {
      ServiceContractResolverRegistry resolverRegistry =
bus.getExtension(ServiceContractResolverRegistry.class);
      if (resolverRegistry != null)
      {
        resolverRegistry.register(this);
      }
    }
  }

  public URI getContractLocation(QName serviceName)
  {
    ...
  }
}
```

클라이언트에 계약 확인자를 등록하려면 빈 요소를 클라이언트의 구성에 추가해야 합니다. 빈 요소의 클래스 특성은 계약 확인자를 구현하는 클래스의 이름입니다.

*예 29.6. "계약 Resolver 구성" **org.apache.cxf.demos.myContractResolver** 클래스에서 구현된 구성 확인자를 추가하기 위한 빈을 보여줍니다.*

*예 29.6. 계약 **Resolver** 구성*

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  ...
  <bean id="myResolver" class="org.apache.cxf.demos.myContractResolver" />
  ...
</beans>
```

29.4.5. 계약 해결 순서

*새 프록시가 생성되면 런타임에서 계약 레지스트리 확인자를 사용하여 원격 서비스의 **WSDL** 계약을 찾습니다. 계약 확인자 레지스트리는 확인자가 등록된 순서대로 각 계약 확인자의 **getContractLocation()** 메서드를 호출합니다. 등록된 계약 확인자 중 하나에서 반환된 첫 번째 **URI**를 반환합니다.*

*잘 알려진 공유 파일 시스템에서 **WSDL** 계약을 해결하려고 시도한 계약 확인자를 등록했다면 사용된 유일한 계약 확인자입니다. 그러나 **UDDI** 레지스트리를 사용하여**DL** 위치를 해결한 계약 확인자를 등록한 경우, 레지스트리는 두 확인자를 사용하여 서비스의 **WSDL** 계약을 찾을 수 있습니다. 레지스트리는 먼저 공유 파일 시스템 계약 확인자를 사용하여 계약을 찾습니다. 해당 계약 확인자가 실패한 경우 레지스트리는 **UDDI** 계약 확인자를 사용하여 찾습니다.*

30장. 일반 FAULT HANDLING

초록

JAX-WS 사양은 두 가지 유형의 오류를 정의합니다. 하나는 일반 **JAX-WS** 런타임 예외입니다. 다른 하나는 메시지 처리 중에 발생하는 예외의 프로토콜 특정 클래스입니다.

30.1. 런타임 FAULTS

30.1.1. 개요

대부분의 **JAX-WS API**는 일반 **javax.xml.ws.WebServiceException** 예외를 **throw**합니다.

30.1.2. WebServiceException을 throw하는 API

표 30.1. "WebServiceException을 지원하는 API" 일반 **WebServiceException** 예외를 **throw**할 수 있는 **JAX-WS API** 중 일부를 나열합니다.

표 30.1. WebServiceException을 지원하는 API

API	이유
Binding.setHandlerChain()	처리기 체인 구성에 오류가 있습니다.
BindingProvider.getEndpointReference()	지정된 클래스는 **W3CEndpointReference** 에서 할당되지 않습니다.
Dispatch.invoke()	Dispatch 인스턴스의 구성에 오류가 있거나 서비스와 통신하는 동안 오류가 발생했습니다.
Dispatch.invokeAsync()	Dispatch 인스턴스의 구성에 오류가 있습니다.
Dispatch.invokeOneWay()	Dispatch 인스턴스의 구성에 오류가 있거나 서비스와 통신하는 동안 오류가 발생했습니다.
LogicalMessage.getPayload()	제공된 **JAXBContext** 를 사용하여 페이로드를 분리 해제할 때 오류가 발생했습니다. WebServiceException의 **원인** 필드에는 원래 JAXBException이 포함되어 있습니다.

API	이유
LogicalMessage.setPayload()	메시지 페이로드를 설정할 때 오류가 발생했습니다. **JAXBContext** 를 사용할 때 예외가 throw되는 경우 WebServiceException의 **원인** 필드에는 원래 JAXBException이 포함되어 있습니다.
WebServiceContext.getEndpointReference()	지정된 클래스는 **W3CEndpointReference** 에서 할당되지 않습니다.

30.2. 프로토콜 FAULTS

30.2.1. 개요

요청 처리 중에 오류가 발생하면 프로토콜 예외가 발생합니다. 모든 동기 원격 호출은 프로토콜 예외를 throw할 수 있습니다. 근본적인 원인은 소비자의 메시지 처리 체인 또는 서비스 공급자에서 발생합니다.

JAX-WS 사양은 일반 프로토콜 예외를 정의합니다. 또한 SOAP 관련 프로토콜 예외 및 HTTP 특정 프로토콜 예외를 지정합니다.

30.2.2. 프로토콜 예외 유형

JAX-WS 사양은 세 가지 유형의 프로토콜 예외를 정의합니다. catch하는 예외는 애플리케이션에서 사용하는 전송 및 바인딩에 따라 다릅니다.

표 30.2. "일반 프로토콜 예외의 유형" 세 가지 유형의 프로토콜 예외와 이러한 예외가 throw되는 경우에 대해 설명합니다.Describes the three types of protocol exception and when they are thrown.

표 30.2. 일반 프로토콜 예외의 유형

예외 클래스	Thrown
javax.xml.ws.ProtocolException	이 예외는 일반적인 프로토콜 예외입니다. 사용 중인 프로토콜에 관계없이 잡을 수 있습니다. SOAP 바인딩 또는 HTTP 바인딩을 사용하는 경우 특정 오류 유형으로 캐스팅할 수 있습니다. HTTP 또는 JMS 전송과 함께 XML 바인딩을 사용하는 경우 일반 프로토콜 예외를 보다 구체적인 오류 유형으로 캐스팅할 수 없습니다.

예외 클래스	Thrown
javax.xml.ws.soap.SOAPFaultException	이 예외는 SOAP 바인딩을 사용할 때 원격 호출에 의해 발생합니다.This exception is thrown by remote invocations when using the SOAP binding. 자세한 내용은 "SOAP 프로토콜 예외 사용" 에서 참조하십시오.
javax.xml.ws.http.HTTPException	이 예외는 Apache CXF HTTP 바인딩을 사용하여 RESTful 웹 서비스를 개발할 때 throw됩니다. 자세한 내용은 VI 부. RESTful 웹 서비스 개발에서 참조하십시오.

30.2.3. SOAP 프로토콜 예외 사용

***SOAPFaultException** 예외는 **SOAP** 오류를 래핑합니다. 기본 **SOAP** 오류는 **fault** 필드에 **javax.xml.soap.SOAPFault** 오브젝트로 저장됩니다.*

*서비스 구현에서 애플리케이션에 대해 생성된 사용자 지정 예외에 맞지 않는 예외를 **throw**해야 하는 경우 예외 작성자를 사용하여 **SOAPFaultException**에서 오류를 래핑하고 사용자에게 다시 **throw**할 수 있습니다. 예 30.1. "SOAP 프로토콜 예외 발생" 메서드가 잘못된 매개 변수를 전달하는 경우 **SOAPFaultException**을 만들고 **throw**하기 위한 코드를 표시합니다.**Shows code for creating and throwing a SOAPFaultException if the method is passed an invalid parameter.***

예 30.1. SOAP 프로토콜 예외 발생

```
public Quote getQuote(String ticker)
{
  ...
  if(tickers.length()<3)
  {
    SOAPFault fault = SOAPFactory.newInstance().createFault();
    fault.setFaultString("Ticker too short");
    throw new SOAPFaultException(fault);
  }
  ...
}
```

*소비자가 **SOAPFaultException** 예외를 **catch**하면 래핑된 **SOAPFault** 예외를 검사하여 예외의 기본 원인을 검색할 수 있습니다. 예 30.2. "SOAP 프로토콜 예외에서 Fault를 받기" 와 같이 **SOAPFault** 예외는 **SOAPFaultException** 예외의 **getFault()** 메서드를 사용하여 검색됩니다.*

예 30.2. SOAP 프로토콜 예외에서 Fault를 받기

```
...
try
```

```
{
  proxy.getQuote(ticker);
}
catch (SOAPFaultException sfe)
{
  SOAPFault fault = sfe.getFault();
  ...
}
```

31장. 서비스 게시

초록

JAX-WS 서비스를 독립 실행형 Java 애플리케이션으로 배포하려면 서비스 공급자를 게시하는 코드를 명시적으로 구현해야 합니다.

31.1. 서비스를 게시하는 경우WHEN TO PUBLISH A SERVICE

Apache CXF는 서비스를 서비스 공급자로 게시하는 다양한 방법을 제공합니다. 서비스를 게시하는 방법은 사용 중인 배포 환경에 따라 다릅니다. Apache CXF에서 지원하는 대부분의 컨테이너에는 끝점 게시 논리가 필요하지 않습니다. 두 가지 예외가 있습니다.

- 독립 실행형 Java 애플리케이션으로 서버 배포
- Blueprint 없이 OSGi 컨테이너에 서버 배포

지원되는 컨테이너에 애플리케이션을 배포하는 방법에 대한 자세한 내용은 IV 부. 웹 서비스 엔드 포인트 구성 을 참조하십시오.

31.2. 서비스를 게시하는 데 사용되는 API

31.2.1. 개요

javax.xml.ws.Enddpoint 클래스는 JAX-WS 서비스 공급자를 게시하는 작업을 수행합니다. 끝점을 게시하려면 다음을 수행합니다.

1. 서비스 공급자에 대한 Endpoint 오브젝트를 생성합니다.
2. 엔드포인트를 게시합니다.
3. 애플리케이션이 종료되면 끝점을 중지합니다.

Endpoint 클래스는 서비스 공급자를 만들고 게시하는 방법을 제공합니다. 또한 단일 메서드 호출에서

서비스 공급자를 생성하고 게시할 수 있는 메서드를 제공합니다.It also provides a method that can create and publish a service provider in a single method call.

31.2.2. 서비스 공급자 인스턴스화

서비스 공급자는 Endpoint 오브젝트를 사용하여 인스턴스화됩니다. 다음 방법 중 하나를 사용하여 서비스 공급자의 끝점 오브젝트를 인스턴스화합니다.

- 정적끝점생성Objectimplementor This create() 메서드는 지정된 서비스 구현에 대한 끝점을 반환합니다. Endpoint 오브젝트는 구현 클래스의 javax.xml.ws.BindingType 주석에서 제공하는 정보를 사용하여 생성됩니다. 주석이 없으면 끝점에서 기본 SOAP 1.1/HTTP 바인딩을 사용합니다.

- 정적EndpointcreateURIbindingIDObjectimplementor This create() 메서드는 지정된 바인딩을 사용하여 지정된 구현 개체에 대한 Endpoint 오브젝트를 반환합니다. 이 메서드는 javax.xml.ws.BindingType 주석에서 제공하는 바인딩 정보가 있는 경우 덮어씁니다. bindingID 를 확인할 수 없거나 null 인 경우 javax.xml.ws.BindingType 에 지정된 바인딩을 사용하여 끝점을 만듭니다. bindingID 또는 javax.xml.ws.BindingType 을 모두 사용할 수 없는 경우 끝점 은 기본 SOAP 1.1/HTTP 바인딩을 사용하여 생성됩니다.

- 정적끝점게시문자열주소Objectimplementor publish() 메서드는 지정된 구현에 대한 Endpoint 개체를 만들고 게시합니다. Endpoint 오브젝트에 사용되는 바인딩은 제공된 주소 의 URL 체계에 따라 결정됩니다. 구현에서 사용할 수 있는 바인딩 목록은 URL 스키마를 지원하는 바인딩에 대해 검색됩니다. If one is found, the Endpoint object is created and published 찾을 수 없는 경우 메서드가 실패합니다.

  publish() 를 사용하는 것은 create() 메서드 중 하나를 호출한 다음 ???TITLE??? 에 사용된 publish() 메서드를 호출하는 것과 동일합니다.

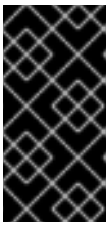
중요

모든 끝점 생성 메서드에 전달된 구현 오브젝트는 javax.jws.WebService 로 주석이 달린 클래스의 인스턴스여야 하며 SEI 구현 요구 사항을 충족해야 합니다. 그러지 않으면 javax.xml.ws.WebServiceProvider 로 주석이 달린 클래스의 인스턴스여야 합니다.

31.2.3. 서비스 공급자 게시

다음 Endpoint 방법 중 하나를 사용하여 서비스 공급자를 게시할 수 있습니다.

- 게시문자열주소 이 **publish()** 메서드는 지정된 주소에 서비스 공급자를 게시합니다.



중요

주소 의 **URL** 스키마는 서비스 공급자의 바인딩 중 하나와 호환되어야 합니다.

- **PublishObjectserverContext** 이 **publish()** 메서드는 지정된 서버 컨텍스트에 제공된 정보를 기반으로 서비스 공급자를 게시합니다. 서버 컨텍스트는 끝점에 대한 주소를 정의해야 하며 컨텍스트도 서비스 공급자의 사용 가능한 바인딩 중 하나와 호환되어야 합니다.

31.2.4. 게시된 서비스 공급자 중지

서비스 공급자가 더 이상 필요하지 않은 경우 해당 **stop()** 메서드 사용을 중지해야 합니다. 예 31.1. "게시된 끝점 중지 방법" 에 표시된 **stop()** 메서드는 끝점을 종료하고 사용 중인 리소스를 정리합니다.

예 31.1. 게시된 끝점 중지 방법

```
중지
```

중요

끝점이 중지되면 다시 게시될 수 없습니다.

31.3. 일반 JAVA 애플리케이션에 서비스 게시

31.3.1. 개요

애플리케이션을 일반 **java** 애플리케이션으로 배포하려면 애플리케이션의 **main()** 메서드에 끝점을 게시하는 논리를 구현해야 합니다. **Apache CXF**는 애플리케이션의 **main()** 메서드를 작성하는 두 가지 옵션을 제공합니다.

- **wsdl2java** 툴에서 생성한 **main()** 메서드를 사용합니다.

- 끝점을 게시하는 사용자 정의 **main()** 메서드 작성

31.3.2. 서버 Mainline 생성

코드 생성기 **-server** 플래그를 사용하면 도구가 간단한 서버 메인라인을 생성할 수 있습니다. 예 31.2. "생성된 서버 메인 라인" 와 같이 생성된 서버 메인 라인은 지정된 **WSDL** 계약의 각 포트 요소에 대해 하나의 서비스 공급자를 게시합니다.

자세한 내용은 **44.2절. "cxf-codegen-plugin"** 에서 참조하십시오.

예 31.2. "생성된 서버 메인 라인" 생성된 서버 메인라인을 표시합니다.

예 31.2. 생성된 서버 메인 라인

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer {

    protected GreeterServer() throws Exception {
        System.out.println("Starting Server");
        Object implementor = new GreeterImpl();
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new GreeterServer();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

예 31.2. "생성된 서버 메인 라인" 의 코드는 다음을 수행합니다.

서비스 구현 오브젝트의 사본을 인스턴스화합니다.

끝점의 계약에 있는 **wsdl:port** 요소의 **address** 하위 항목 콘텐츠를 기반으로 끝점의 주소를 생성합니다.

엔드포인트를 게시합니다.

31.3.3. 서버 Mainline 작성

Java 첫 번째 개발 모델을 사용하거나 생성된 서버 메인 라인을 사용하지 않으려면 직접 작성할 수 있습니다. 서버 메인라인을 작성하려면 다음을 수행해야 합니다.

1. "서비스 공급자 인스턴스화" 서비스 공급자의 **javax.xml.ws.Endpoint** 오브젝트입니다.
2. 서비스 공급자를 게시할 때 사용할 선택적 서버 컨텍스트를 생성합니다.
3. "서비스 공급자 게시" **publish()** 방법 중 하나를 사용하는 서비스 공급자입니다.
4. 애플리케이션을 종료할 준비가 되면 서비스 공급자를 중지합니다.

예 31.3. "사용자 정의 서버 Mainline" 서비스 공급자 게시를 위한 코드를 보여줍니다.

예 31.3. 사용자 정의 서버 Mainline

```
package org.apache.hello_world_soap_http;

import javax.xml.ws.Endpoint;

public class GreeterServer
{
  protected GreeterServer() throws Exception
  {
  }

  public static void main(String args[]) throws Exception
  {
    GreeterImpl impl = new GreeterImpl();
    Endpoint endpt.create(impl);
    endpt.publish("http://localhost:9000/SoapContext/SoapPort");

    boolean done = false;
   while(!done)
    {
```

```
        ...
      }

      endpt.stop();
       System.exit(0);
     }
   }
```

예 31.3. "사용자 정의 서버 Mainline" 의 코드는 다음을 수행합니다.

서비스 구현 개체의 사본을 인스턴스화합니다.

서비스 구현을 위해 게시되지 않은 끝점을 생성합니다.

http://localhost:9000/SoapContext/SoapPort 에서 서비스 공급자를 게시합니다.

서버가 종료될 때까지 반복합니다.

게시된 끝점을 중지합니다.

31.4. OSGI 컨테이너에 서비스 게시

31.4.1. 개요

OSGi 컨테이너에 배포되는 애플리케이션을 개발할 때 패키지된 번들 라이프 사이클과 끝점의 게시 및 중지를 조정해야 합니다. 번들이 시작될 때 끝점을 게시하고 번들이 중지되면 끝점을 중지하려고 합니다.

OSGi bundle activator를 구현하여 엔드포인트 라이프사이클을 번들의 라이프 사이클에 연결합니다. bundle activator는 OSGi 컨테이너에서 시작 시 번들에 대한 리소스를 생성하는 데 사용됩니다. 컨테이너는 또한 bundle activator를 사용하여 중지 시 번들 리소스를 정리합니다.

31.4.2. bundle activator 인터페이스

org.osgi.framework.BundleActivator 인터페이스를 구현하여 애플리케이션에 대한 번들 활성화기를 생성합니다. 예 31.4. "Bundle Activator Interface" 에 표시된 BundleActivator 인터페이스에는 구현

해야 하는 두 가지 방법이 있습니다.

예 31.4. Bundle Activator Interface

```
interface BundleActivator
{
  public void start(BundleContext context)
  throws java.lang.Exception;

  public void stop(BundleContext context)
  throws java.lang.Exception;
}
```

start() 메서드는 번들을 시작할 때 컨테이너에서 호출됩니다. 끝점을 인스턴스화하고 게시하는 곳입니다.

stop() 메서드는 번들이 중지될 때 컨테이너에서 호출됩니다. 여기서 엔드포인트를 중지합니다.

31.4.3. 시작 방법 구현

bundle activator의 시작 방법은 끝점을 게시하는 위치입니다. 끝점을 게시하려면 시작 방법을 수행해야 합니다.

1. "서비스 공급자 인스턴스화" 서비스 공급자의 **javax.xml.ws.Endpoint** 오브젝트입니다.
2. 서비스 공급자를 게시할 때 사용할 선택적 서버 컨텍스트를 생성합니다.
3. "서비스 공급자 게시" **publish()** 방법 중 하나를 사용하는 서비스 공급자입니다.

예 31.5. "Bundle Activator Start Method for Publishing an Endpoint" 서비스 공급자 게시를 위한 코드를 보여줍니다.

예 31.5. Bundle Activator Start Method for Publishing an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void start(BundleContext context)
  {
    WidgetOrderImpl impl = new WidgetOrderImpl();
    endpt = Endpoint.create(impl);
    endpt.publish("http://localhost:9000/SoapContext/SoapPort");
  }

  ...

}
```

예 31.5. “Bundle Activator Start Method for Publishing an Endpoint” 의 코드는 다음을 수행합니다.

서비스 구현 개체의 사본을 인스턴스화합니다.

서비스 구현을 위해 게시되지 않은 끝점을 생성합니다.

http://localhost:9000/SoapContext/SoapPort 에서 서비스 공급자를 게시합니다.

31.4.4. 중지 방법 구현

***bundle activator*의 *stop* 방법은 애플리케이션에서 사용하는 리소스를 정리하는 곳입니다. 구현에는 애플리케이션에서 게시한 모든 엔드포인트를 중지할 수 있는 논리가 포함되어야 합니다.**

예 31.6. “Bundle Activator Stop Method for Stopping an Endpoint” 는 게시된 엔드포인트를 중지하는 중지 방법을 보여줍니다.

예 31.6. Bundle Activator Stop Method for Stopping an Endpoint

```
package com.widgetvendor.osgi;

import javax.xml.ws.Endpoint;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
```

```
public class widgetActivator implements BundleActivator
{
  private Endpoint endpt;
  ...

  public void stop(BundleContext context)
  {
    endpt.stop();
  }

  ...

}
```

31.4.5. 컨테이너 정보

애플리케이션 번들에 **bundle actator**가 포함되어 있음을 컨테이너에 알려야 합니다. **Bundle-Activator** 속성을 번들 매니페스트에 추가하여 이 작업을 수행합니다. 이 속성은 번들을 활성화할 때 사용할 번들의 클래스에 대해 지시합니다. 해당 값은 번들 활성화기를 구현하는 클래스의 정규화된 이름입니다.

예 31.7. “Bundle Activator Manifest Entry” **com.widgetvendor.osgi.widgetActivator** 클래스에서 구현되는 번들의 매니페스트 항목을 표시합니다.

예 31.7. Bundle Activator Manifest Entry

```
Bundle-Activator: com.widgetvendor.osgi.widgetActivator
```

32장. 기본 데이터 바인딩 개념

초록

Apache CXF에서 유형 매핑을 처리하는 방법에 적용되는 여러 일반 항목이 있습니다.

32.1. 스키마 정의 포함 및 가져오기

32.1.1. 개요

Apache CXF는 **include** 및 **import schema** 태그를 사용하여 스키마 정의 포함 및 가져오기를 지원합니다. 이러한 태그를 사용하면 외부 파일 또는 리소스의 정의를 스키마 요소의 범위에 삽입할 수 있습니다. 포함 및 가져오기의 필수 차이점은 다음과 같습니다.

- 포함에는 포함된 스키마 요소와 동일한 대상 네임스페이스에 속하는 정의가 포함됩니다.
- 가져오기는 포함된 스키마 요소의 다른 대상 네임스페이스에 속하는 정의를 가져옵니다.

32.1.2. xsd:include syntax

include 지시문에는 다음 구문이 있습니다.

```
<include schemaLocation="anyURI" />
```

URI 에서 지정하는 참조 스키마는 포함된 스키마와 동일한 대상 네임스페이스에 속하거나 대상 네임스페이스에 전혀 속하지 않아야 합니다. 참조된 스키마가 대상 네임스페이스에 속하지 않는 경우 포함된 스키마의 네임스페이스로 자동으로 사용됩니다.**If the referenced schema does not belong to any target namespace, it is automatically adopted into the enclosing schema's namespace when it is included.**

예 32.1. “다른 스키마를 포함하는 스키마의 예” 다른 **XML** 스키마 문서를 포함하는 **XML** 스키마 문서의 예를 보여줍니다.

예 32.1. 다른 스키마를 포함하는 스키마의 예

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

```
  <types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
          xmlns="http://www.w3.org/2001/XMLSchema">
      <include schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="tns:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  ...
</definitions>
```

예 32.2. "포함된 스키마의 예" 포함된 스키마 파일의 내용을 표시합니다.

예 32.2. 포함된 스키마의 예

```
<schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
      xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

32.1.3. XSD:import 구문

import 지시문의 구문은 다음과 같습니다.

```
<import namespace="namespaceAnyURI"
      schemaLocation="schemaAnyURI" />
```

가져온 정의는 네임스페이스**AnyURI** 대상 네임스페이스에 속해야 합니다. **namespaceAnyURI** 가 비어 있거나 지정되지 않은 경우 가져온 스키마 정의가 정규화되지 않습니다.

*예 32.3. "다른 스키마를 가져오는 스키마의 예" 다른 **XML** 스키마를 가져오는 **XML** 스키마의 예를 보여줍니다.*

예 32.3. 다른 스키마를 가져오는 스키마의 예

```
<definitions targetNamespace="http://schemas.redhat.com/tests/schema_parser"
             xmlns:tns="http://schemas.redhat.com/tests/schema_parser"
             xmlns:imp="http://schemas.redhat.com/tests/imported_types"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://schemas.redhat.com/tests/schema_parser"
          xmlns="http://www.w3.org/2001/XMLSchema">
      <import namespace="http://schemas.redhat.com/tests/imported_types"
           schemaLocation="included.xsd"/>
      <complexType name="IncludingSequence">
        <sequence>
          <element name="includedSeq" type="imp:IncludedSequence"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  ...
</definitions>
```

예 32.4. "가져온 스키마의 예" 가져온 스키마 파일의 내용을 표시합니다.

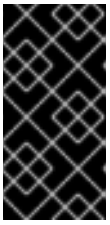
예 32.4. 가져온 스키마의 예

```
<schema targetNamespace="http://schemas.redhat.com/tests/imported_types"
        xmlns="http://www.w3.org/2001/XMLSchema">
  <!-- Included type definitions -->
  <complexType name="IncludedSequence">
    <sequence>
      <element name="varInt" type="int"/>
      <element name="varString" type="string"/>
    </sequence>
  </complexType>
</schema>
```

32.1.4. 비참조 스키마 문서 사용

서비스의 **WSDL** 문서에서 참조되지 않는 스키마 문서에 정의된 유형을 사용하는 것은 세 가지 단계 프로세스입니다.

1. **xsd2wsdl** 도구를 사용하여 스키마 문서를 **WSDL** 문서로 변환합니다.
2. 생성된 **WSDL** 문서에서 **wsdl2java** 도구를 사용하여 유형에 대한 **Java**를 생성합니다.



중요

wsdl2java 툴에서 **WSDL** 문서가 서비스를 정의하지 않음을 알리는 경고를 받게 됩니다. 이 경고를 무시할 수 있습니다.

3. 생성된 클래스를 **classpath**에 추가합니다.

32.2. XML 네임스페이스 매핑

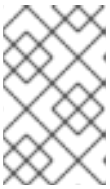
32.2.1. 개요

XML Schema 유형, 그룹 및 요소 정의는 네임스페이스를 사용하여 범위가 지정됩니다. 네임스페이스는 동일한 이름을 사용하는 엔터티 간에 충돌할 수 있는 기능을 방지합니다. **Java** 패키지는 비슷한 목적을 제공합니다. 따라서 **Apache CXF**는 스키마 문서에 정의된 구조를 구현하는 데 필요한 클래스가 포함된 패키지에 스키마 문서의 대상 네임스페이스를 매핑합니다.

32.2.2. 패키지 이름 지정

생성된 패키지의 이름은 다음 알고리즘을 사용하여 스키마의 대상 네임스페이스에서 파생됩니다.

1. **URI** 스키마가 있는 경우 제거됩니다.



참고

Apache CXF는 **http:**, **https:**, 및 **urn:** 스키마만 제거합니다.

예를 들어 네임스페이스 **http:\\www.widgetvendor.com\types\widgetTypes.xsd** 는 **\\widgetvendor.com\types\widgetTypes.xsd** 가 됩니다.

2. 파일 형식 식별자가 있는 경우 제거됩니다.**The trailing file type identifier, if present is removed.**

예를 들어 **\\www.widgetvendor.com\types\widgetTypes.xsd** 는 **\\widgetTypes.com\types\widgetTypes** 가 됩니다.

3.

결과 문자열은 / 및 : 구분 기호를 사용하여 문자열 목록으로 나뉩니다.

따라서 \\www.widgetvendor.com\types\widgetTypes 는 목록 {"www.widegetvendor.com", "types", "widgetTypes"} 가 됩니다.

4.

목록의 첫 번째 문자열이 인터넷 도메인 이름인 경우 다음과 같이 분해됩니다.

a.

선행 www. 는 제거됩니다.

b.

나머지 문자열은 . 을 구분 기호로 사용하여 해당 구성 요소로 나뉩니다.

c.

목록의 순서는 역순으로 정렬됩니다.

따라서 {"www.widegetvendor.com, "types", "widgetTypes "}는 {"com", "widegetvendor", "types", "widgetTypes"}}가 됩니다.

참고

인터넷 도메인 이름은 .com,.net,.edu,.org,.gov 또는 두 문자 국가 코드 중 하나로 끝납니다.

5.

문자열이 모든 소문자로 변환됩니다.

따라서 {"com", "widegetvendor", "types", "widgetTypes"} 는 {"com", "getvendor", "types", "widgettypes "}이 됩니다.

6.

문자열은 다음과 같이 유효한 Java 패키지 이름 구성 요소로 정규화됩니다.

a.

문자열에 특수 문자가 포함된 경우 특수 문자는 밑줄(_)으로 변환됩니다.

b.

문자열이 Java 키워드인 경우 키워드 앞에 밑줄(_)이 붙습니다.

c.

문자열이 **numeral**로 시작하는 경우 문자열 앞에 밑줄(_)이 붙습니다.

7.

문자열은 구분 기호로 **.** 을 사용하여 연결됩니다.

따라서 **{"com", "widegetvendor", "types", "widgettypes"}** 는 패키지 이름 **com.widgetvendor.types.widgettypes** 가 됩니다.

네임스페이스 **http:\\www.widgetvendor.com\types\widgetTypes.xsd** 네임스페이스에 정의된 **XML** 스키마 구성은 **Java** 패키지 **com.widgetvendor.types.widgettypes** 에 매핑됩니다.

32.2.3. 패키지 콘텐츠

JAXB 생성 패키지에는 다음이 포함됩니다.

- 스키마에 정의된 각 복잡한 유형을 구현하는 클래스

  복잡한 유형 매핑에 대한 자세한 내용은 **35장. 복합 유형 사용** 을 참조하십시오.

- 열거형 **facet**를 사용하여 정의된 모든 단순 형식에 대한 열거형 유형

  열거형이 매핑되는 방법에 대한 자세한 내용은 **34.3절. “열거형”** 을 참조하십시오.

- 스키마에서 개체를 인스턴스화하는 메서드를 포함하는 공용 **ObjectFactory** 클래스입니다.**A public ObjectFactory class that contains methods for instantiating objects from the schema.**

  ObjectFactory 클래스에 대한 자세한 내용은 **32.3절. “Object Factory”** 을 참조하십시오.

- **package-info.java** 파일은 패키지의 클래스에 대한 메타데이터를 제공합니다.

32.3. OBJECT FACTORY

32.3.1. 개요

JAXB는 개체 팩토리를 사용하여 JAXB 생성 생성자의 인스턴스를 인스턴스화하는 메커니즘을 제공합니다. 개체 팩토리에는 패키지의 범위에서 모든 XML 스키마 정의 구문을 인스턴스화하는 메서드가 포함됩니다. 유일한 예외는 개체 팩토리에서 생성 메서드를 가져오지 않는다는 것입니다.The only exception is that enumerations do not get a creation method in the object factory.

32.3.2. 복잡한 유형 팩토리 방법

XML 스키마 복잡한 형식을 구현하기 위해 생성된 각 Java 클래스에 대해 개체 팩토리에는 클래스 인스턴스를 만드는 메서드가 포함됩니다. 이 메서드는 다음과 같은 형식을 취합니다.This method takes the form:

```
typeName createtypeName();
```

예를 들어 스키마에 widgetType 이라는 복잡한 유형이 포함된 경우 Apache CXF는 이를 구현하기 위해 widget Type 이라는 클래스를 생성합니다. 예 32.5. “복잡한 Type Object Factory Entry” 개체 팩토리에서 생성된 생성 방법을 보여줍니다.

예 32.5. 복잡한 Type Object Factory Entry

```
public class ObjectFactory
{
  ...
  WidgetType createWidgetType()
  {
    return new WidgetType();
  }
  ...
}
```

32.3.3. 요소 팩토리 방법

스키마의 전역 범위에 선언되는 요소의 경우 Apache CXF는 개체 팩토리에 팩토리 메서드를 삽입합니다. 33장. XML Elements 사용 에서 설명한 것처럼 XML 스키마 요소는 JAXBElement<T> 오브젝트에 매핑 됩니다. 생성 방법은 다음과 같은 형식을 사용합니다.

```
public JAXBElement<elementType> createelementName(elementType value);
```

예를 들어 xsd:string 의 주석 이라는 요소가 있는 경우 Apache CXF는 에 표시된 개체 팩토리 메서드를 생성합니다. 예 32.6. “element Object Factory Entry”

예 32.6. element Object Factory Entry

```
public class ObjectFactory
```

```
{
    ...
    @XmlElementDecl(namespace = "...", name = "comment")
    public JAXBElement<String> createComment(String value) {
        return new JAXBElement<String>(_Comment_QNAME, String.class, null, value);
    }
    ...
}
```

32.4. 런타임 MARSHALLER에 클래스 추가

32.4.1. 개요

Apache CXF 런타임이 **XML** 데이터를 읽고 쓸 때 **XML** 스키마 유형을 대표 **Java** 유형과 연결하는 맵을 사용합니다. 기본적으로 맵에는 **WSDL** 계약 스키마 요소의 대상 네임스페이스에 정의된 모든 유형이 포함되어 있습니다. 또한 **WSDL** 계약으로 가져온 스키마의 네임스페이스에서 생성된 모든 형식을 포함합니다.

애플리케이션의 스키마 요소에서 사용하는 스키마 네임스페이스 이외의 네임스페이스에서 형식을 추가하면 **@XmlSee also** 주석을 사용하여 수행할 수 있습니다. 애플리케이션에서 애플리케이션 **WSDL** 문서 범위 외부에서 생성된 유형으로 작업해야 하는 경우 **@XmlSee**도 주석을 편집하여 **JAXB** 맵에 추가할 수 있습니다.

32.4.2. @XmlSee also 주석 사용

@XmlSee also 주석을 서비스의 **SEI**에 추가할 수 있습니다. **JAXB** 컨텍스트에 포함할 쉼표로 구분된 클래스 목록을 포함합니다. 예 32.7. “JAXB Context에 클래스 추가” **@XmlSeeAlso** 주석을 사용하는 구문을 보여줍니다.

예 32.7. JAXB Context에 클래스 추가

```
import javax.xml.bind.annotation.XmlSeeAlso;
    @WebService()
    @XmlSeeAlso({Class1.class, Class2.class, ..., ClassN.class})
    public class GeneratedSEI {
        ...
    }
```

JAXB 생성 클래스에 대한 액세스 권한이 있는 경우 필요한 유형을 지원하는 데 생성된 **ObjectFactory** 클래스를 사용하는 것이 더 효율적입니다. **ObjectFactory** 클래스를 포함 하는 것은 개체 팩토리에 알려진 모든 클래스를 포함 합니다.

32.4.3. 예제

예 32.8. "JAXB Context에 클래스 추가" 에는 **@XmlSee**에도 주석이 달린 **SEI**가 표시됩니다.

예 32.8. JAXB Context에 클래스 추가

```
...
import javax.xml.bind.annotation.XmlSeeAlso;
...
    @WebService()
    @XmlSeeAlso({org.apache.schemas.types.test.ObjectFactory.class,
org.apache.schemas.tests.group_test.ObjectFactory.class})
    public interface Foo {
        ...
    }
```

33장. XML ELEMENTS 사용

초록

XML 스키마 요소는 XML 문서에서 요소의 인스턴스를 정의하는 데 사용됩니다.XML Schema elements are used to define an instance of an element in an XML document. 요소는 XML 스키마 문서의 전역 범위에서 정의되거나 복잡한 형식의 멤버로 정의됩니다. 전역 범위에 정의되면 Apache CXF는 이를 더 쉽게 조작할 수 있는 JAXB 요소 클래스에 매핑합니다.

33.1. 개요

XML 문서의 요소 인스턴스는 XML Schema 문서의 전역 범위에 있는 XML Schema 요소 요소에 의해 정의되며, Java 개발자가 요소를 더 쉽게 작업할 수 있도록 합니다. Apache CXF는 전역 범위 요소를 특수 JAXB 요소 클래스 또는 해당 콘텐츠 형식과 일치하도록 생성된 Java 클래스에 전역 범위 요소를 매핑합니다.

요소가 매핑되는 방법은 type 속성에서 참조하는 명명된 유형을 사용하여 요소가 정의되었는지 또는 요소가 인라인 유형 정의를 사용하여 정의되었는지에 따라 달라집니다. 인라인 유형 정의로 정의된 요소는 Java 클래스에 매핑됩니다.

인라인 유형은 스키마의 다른 요소에서 재사용할 수 없기 때문에 명명된 유형을 사용하여 요소를 정의하는 것이 좋습니다.

33.2. XML 스키마 매핑

XML Schema 요소는 요소 요소 를 사용하여 정의됩니다. 요소 요소에 는 하나의 필수 특성이 있습니다. name 은 XML 문서에 표시되는 요소의 이름을 지정합니다.

name 속성 요소 요소 외에도 표 33.1. “요소를 정의하는 데 사용되는 특성” 에 나열된 선택적 속성이 있습니다.

표 33.1. 요소를 정의하는 데 사용되는 특성

속성	설명
type	요소의 형식을 지정합니다. 형식은 모든 XML 스키마 기본 유형 또는 계약에 정의된 이름이 지정된 복잡한 유형일 수 있습니다. 이 속성을 지정하지 않으면 인라인 유형 정의를 포함해야 합니다.

속성	설명
nillable	요소가 문서에서 완전히 벗어날 수 있는지 여부를 지정합니다.Specifies if an element can be left out of a document entirely. **nillable** 을 **true** 로 설정하면 스키마를 사용하여 생성된 문서에서 요소를 생략할 수 있습니다.
abstract	인스턴스 문서에서 요소를 사용할 수 있는지 여부를 지정합니다.Specifies if an element can be used in an instance document. **true** 는 요소가 인스턴스 문서에 나타나지 않음을 나타냅니다. 대신 **substitutionGroup** 속성에 이 요소의 QName이 포함된 다른 요소가 이 요소의 위치에 표시되어야 합니다. 이 특성의 효과 코드 생성에 대한 자세한 내용은 "추상 요소의 Java 매핑" 을 참조하십시오.
substitutionGroup	이 요소로 대체할 수 있는 요소의 이름을 지정합니다. 유형 대체 사용에 대한 자세한 내용은 37장. *element Substitution* 을 참조하십시오.
default	요소의 기본값을 지정합니다. 이 특성의 효과 코드 생성에 대한 자세한 내용은 "Java 요소의 기본값을 사용하여 매핑" 을 참조하십시오.
fixed	요소에 대한 고정 값을 지정합니다.Specifies a fixed value for the element.

예 33.1. "간단한 XML 스키마 요소 정의" 간단한 요소 정의를 보여줍니다.

> *예 33.1. 간단한 XML 스키마 요소 정의*
>
> ```
> <element name="joeFred" type="xsd:string" />
> ```

*요소는 또한 인라인 유형 정의를 사용하여 자체 유형을 정의할 수 있습니다. 인라인 형식은 **complexType** 요소 또는 **simpleType** 요소를 사용하여 지정됩니다. 데이터 유형이 복잡하고 간단한지 여부를 지정하면 각 유형의 데이터에 사용할 수 있는 도구를 사용하여 필요한 모든 유형의 데이터를 정의할 수 있습니다.*

예 33.2. "인라인 유형을 사용한 XML 스키마 요소 정의" 줄 형식 정의를 사용하여 요소 정의를 표시합니다.Shows an element definition with an in-line type definition.

예 33.2. 인라인 유형을 사용한 XML 스키마 요소 정의

```
<element name="skate">
  <complexType>
    <sequence>
      <element name="numWheels" type="xsd:int" />
      <element name="brand" type="xsd:string" />
    </sequence>
  </complexType>
</element>
```

33.3. 명명된 유형을 사용하여 요소의 매핑

기본적으로 전역적으로 정의된 요소는 요소 요소의 **type** 속성 값으로 템플릿 클래스가 결정되는 **JAXBElement<T>** 개체에 매핑됩니다. 기본 유형의 경우 템플릿 클래스는 "래퍼 클래스"에 설명된 래퍼 클래스 매핑을 사용하여 파생됩니다. 복잡한 유형의 경우 복잡한 유형을 지원하기 위해 생성된 **Java** 클래스가 템플릿 클래스로 사용됩니다.

매핑을 지원하고 요소의 **QName**에 대한 불필요한 걱정의 개발자를 완화하기 위해 예 33.3. "Globally Scoped Element에 대한 Object Factory Method"과 같이 전역적으로 정의된 각 요소에 대해 개체 팩토리 메서드가 생성됩니다.

예 33.3. Globally Scoped Element에 대한 Object Factory Method

```
public class ObjectFactory {

    private final static QName _name_QNAME = new QName("targetNamespace", "localName");

    ...

    @XmlElementDecl(namespace = "targetNamespace", name = "localName")
    public JAXBElement<type> createname(type value);

}
```

예를 들어 예 33.1. "간단한 XML 스키마 요소 정의"에 정의된 요소는 예 33.4. "간단한 요소용 개체 팩토리"로 표시된 오브젝트 팩토리 메서드가 생성됩니다.

예 33.4. 간단한 요소용 개체 팩토리

```
public class ObjectFactory {

    private final static QName _JoeFred_QNAME = new QName("...", "joeFred");

    ...
```

```
    @XmlElementDecl(namespace = "...", name = "joeFred")
    public JAXBElement<String> createJoeFred(String value);

}
```

예 33.5. "글로벌 범위 요소 사용" Java에서 전역 범위가 지정된 요소를 사용하는 예를 보여줍니다.

예 33.5. 글로벌 범위 요소 사용

```
JAXBElement<String> element = createJoeFred("Green");
String color = element.getValue();
```

33.4. WSDL에서 명명된 유형에서 요소 사용

전역 범위 요소가 메시지 부분을 정의하는 데 사용되는 경우 생성된 **Java** 매개 변수는 **JAXBElement<T >**의 인스턴스가 아닙니다. 대신 일반 **Java** 유형 또는 클래스에 매핑됩니다.

예 33.6. "Element를 메시지 파트로 Using an item as a message part" 에 표시된 WSDL 조각이 있는 경우 결과 메서드에는 문자열 유형의 매개 변수가 있습니다.

예 33.6. Element를 메시지 파트로 Using an item as a message part

```
<?xml version="1.0" encoding=";UTF-8"?>
<wsdl:definitions name="HelloWorld"
                  targetNamespace="http://apache.org/hello_world_soap_http"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://apache.org/hello_world_soap_http"
                  xmlns:x1="http://apache.org/hello_world_soap_http/types"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_soap_http/types"
          xmlns="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"><element name="sayHi">
      <element name="sayHi" type="string"/>
      <element name="sayHiResponse" type="string"/>
    </schema>
  </wsdl:types>

  <wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
  </wsdl:message>
  <wsdl:message name="sayHiResponse">
```

```
  <wsdl:part element="x1:sayHiResponse" name="out"/>
 </wsdl:message>

 <wsdl:portType name="Greeter">
  <wsdl:operation name="sayHi">
   <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
   <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
  </wsdl:operation>
 </wsdl:portType>
 ...
</wsdl:definitions>
```

예 33.7. "글로벌 요소를 파트로 사용하는 Java Method" sayHi 작업에 대해 생성된 메서드 서명을 보여줍니다.

예 33.7. 글로벌 요소를 파트로 사용하는 Java Method

```
stringsayHistringin
```

33.5. 인라인 유형을 사용하여 요소의 매핑

인라인 유형을 사용하여 요소가 정의되면 다른 유형을 Java에 매핑하는 데 사용되는 동일한 규칙에 따라 Java에 매핑됩니다. 간단한 유형의 규칙은 34장. 간단한 유형 사용 에 설명되어 있습니다. 복잡한 유형의 규칙은 35장. 복합 유형 사용 에 설명되어 있습니다.

인라인 유형 정의가 있는 요소에 대해 Java 클래스가 생성되면 생성된 클래스는 @XmlRootElement 주석으로 장식됩니다. @XmlRootElement 주석에는 name 및 namespace 의 두 가지 유용한 속성이 있습니다. 이러한 속성은 표 33.2. "@XmlRootElement 주석에 대한 속성" 에 설명되어 있습니다.

표 33.2. @XmlRootElement 주석에 대한 속성

속성	설명
name	XML Schema **요소** 의 **name** 특성 값을 지정합니다.
namespace	요소가 정의된 네임스페이스를 지정합니다. 대상 네임스페이스에 이 요소가 정의되면 속성이 지정되지 않습니다.If this element is defined in the target namespace, the property is not specified.

요소가 다음 조건 중 하나 이상을 충족하는 경우 @XmlRootElement 주석이 사용되지 않습니다.

- 요소의 **nillable** 속성이 **true**로 설정됩니다.

- 요소는 대체 그룹의 헤드 요소입니다.

  대체 그룹에 대한 자세한 내용은 37장. element Substitution 을 참조하십시오.

33.6. 추상 요소의 JAVA 매핑

요소의 **abstract** 속성을 **true**로 설정하면 형식의 인스턴스를 인스턴스화하는 데 필요한 개체 팩토리 메서드가 생성되지 않습니다.**When the element's abstract attribute is set to true the object factory method for instantiating instances of the type is not generated.** 인라인 유형을 사용하여 요소가 정의된 경우 인라인 유형을 지원하는 **Java** 클래스가 생성됩니다.

33.7. JAVA 요소의 기본값을 사용하여 매핑

요소의 기본 특성이 사용되는 경우 **defaultValue** 속성이 생성된 **@XmlElementDecl** 주석에 추가됩니다. 예를 들어 예 33.8. “XML Schema Element with a Default Value” 에 정의된 요소는 예 33.9. “기본 값이 있는 Element에 대한 Object Factory Method” 로 표시된 오브젝트 팩토리 메서드가 생성됩니다.

예 33.8. XML Schema Element with a Default Value

```
<element name="size" type="xsd:int" default="7"/>
```

예 33.9. 기본 값이 있는 Element에 대한 Object Factory Method

```
@XmlElementDecl(namespace = "...", name = "size", defaultValue = "7")
public JAXBElement<Integer> createUnionJoe(Integer value) {
    return new JAXBElement<Integer>(_Size_QNAME, Integer.class, null, value);
}
```

34장. 간단한 유형 사용

초록

XML 스키마 단순 유형은 **xsd:int** 와 같은 **XML** 스키마 기본 유형이거나 **simpleType** 요소를 사용하여 정의됩니다. 하위 또는 속성을 포함하지 않는 요소를 지정하는 데 사용됩니다. 일반적으로 네이티브 **Java** 구문에 매핑되며 구현하기 위해 특수 클래스 생성이 필요하지 않습니다. 열거된 단순 형식은 **Java** 열거형 형식에 매핑되므로 생성된 코드를 생성하지 않습니다.

34.1. 기본 유형

34.1.1. 개요

메시지 파트가 **XML** 스키마 기본 유형 중 하나를 사용하여 정의되면 생성된 매개 변수의 유형이 해당 **Java** 네이티브 유형에 매핑됩니다. 동일한 패턴은 복잡한 형식의 범위 내에 정의된 요소를 매핑할 때 사용됩니다. 결과 필드는 해당 **Java** 네이티브 유형입니다.

34.1.2. 매핑

표 34.1. “XML Schema Primitive Type to Java Native Type Mapping” **XML** 스키마 기본 유형과 **Java** 네이티브 유형 간의 매핑을 나열합니다.

표 34.1. XML Schema Primitive Type to Java Native Type Mapping

XML 스키마 유형	Java Type
xsd:string	문자열
xsd:integer	**BigInteger**
xsd:int	**int**
xsd:long	**long**
xsd:short	**short**
xsd:decimal	**BigDecimal**
xsd:float	**float**
xsd:double	**double**
xsd:boolean	**boolean**

XML 스키마 유형	Java Type
xsd:byte	**byte**
xsd:QName	**QName**
xsd:dateTime	**XMLGregorianCalendar**
xsd:base64Binary	**byte[]**
xsd:hexBinary	**byte[]**
xsd:unsignedInt	**long**
xsd:unsignedShort	**int**
xsd:unsignedByte	**short**
xsd:time	**XMLGregorianCalendar**
xsd:date	**XMLGregorianCalendar**
xsd:g	**XMLGregorianCalendar**
xsd:anySimpleType [a]	개체
xsd:anySimpleType [b]	문자열
xsd:duration	기간
xsd:NOTATION	**QName**

[a] 이 유형의 요소에 대해.

[b] 이 유형의 속성의 경우.

34.1.3. 래퍼 클래스

***XML Schema** 기본 유형을 **Java** 기본 유형으로 매핑하면 가능한 모든 **XML** 스키마 구문에서 작동하지 않습니다. **XML** 스키마 기본 유형이 **Java** 기본 유형의 해당 래퍼 유형에 매핑되어야 하는 몇 가지 경우가 있습니다. 이러한 경우는 다음과 같습니다.*

- *다음과 같이 **nillable** 특성이 있는 요소는 **true** 로 설정됩니다.*

```
<element name="finned" type="xsd:boolean"
        nillable="true" />
```

- ***minOccurs** 특성을 **0** 으로 설정하고 **maxOccurs** 특성을 **1** 로 설정하거나 **maxOccurs** 속성이 지정되지 않은 경우:*

```
<element name="plane" type="xsd:string" minOccurs="0" />
```

- ***use** 특성 을 선택적, 지정하지 않거나 지정하지 않고 기본 속성과 고정 특성이 지정되지 않은 특성 요소는 다음과 같습니다.*

```
<element name="date">
  <complexType>
    <sequence/>
    <attribute name="calType" type="xsd:string"
            use="optional" />
  </complexType>
</element>
```

*표 34.2. “Java Wrapper 클래스 매핑에 대한 기본 스키마 유형” 이러한 경우 **XML** 스키마 기본 유형이 **Java** 래퍼 클래스에 매핑되는 방법을 보여줍니다.*

*표 **34.2. Java Wrapper** 클래스 매핑에 대한 기본 스키마 유형*

스키마 유형	Java Type
xsd:int	**java.lang.Integer**
xsd:long	**java.lang.Long**
xsd:short	**java.lang.Short**
xsd:float	**java.lang.Float**
xsd:double	**java.lang.Double**
xsd:boolean	**java.lang.Boolean**
xsd:byte	**java.lang.Byte**
xsd:unsignedByte	**java.lang.Short**
xsd:unsignedShort	**java.lang.Integer**

스키마 유형	Java Type
xsd:unsignedInt	**java.lang.Long**
xsd:unsignedLong	**java.math.BigInteger**
xsd:duration	**java.lang.String**

34.2. 제한 사항을 정의하는 간단한 유형

34.2.1. 개요

***XML** 스키마를 사용하면 다른 기본 형식 또는 단순 형식에서 새 형식을 파생하여 간단한 형식을 만들 수 있습니다.**XML Schema allows you to create simple types by deriving a new type from another primitive type or simple type.** 간단한 형식은 **simpleType** 요소를 사용하여 설명합니다.*

*새 유형은 기본 유형 을 하나 이상의 **facet**로 제한하여 설명합니다. 이러한 **facets**는 새 유형에 저장될 수 있는 유효한 값을 제한합니다. 예를 들어 정확히 **9**자인 **SSN**이라는 간단한 유형을 정의할 수 있습니다.**For example, you could define a simple type, SSN, which is a string of exactly 9 characters.***

*각 기본 **XML** 스키마 유형에는 고유한 선택적 **facet** 세트가 있습니다.*

34.2.2. 절차

*사용자 고유의 간단한 유형을 정의 하려면 다음을 수행 합니다.**To define your own simple type do the following:***

1. *새 단순 유형의 기본 유형을 결정합니다.*
2. *선택한 기본 유형에 대해 사용 가능한 **facet**에 따라 새 유형을 정의하는 제한 사항을 결정합니다.*
3. *이 섹션에 표시된 구문을 사용하여 적절한 **simpleType** 요소를 계약의 **types** 섹션에 입력합니다.*

34.2.3. XML 스키마로 간단한 유형 정의

예 34.1. "간단한 유형 구문" 는 간단한 유형을 설명하는 구문을 보여줍니다.

예 34.1. 간단한 유형 구문

```
<simpleType name="typeName">
  <restriction base="baseType">
    <facet value="value" />
    <facet value="value" />
    ...
  </restriction>
</simpleType>
```

형식 설명은 **simpleType** 요소에 묶여 있으며 **name** 특성의 값으로 식별됩니다. 새 단순 유형이 정의되는 기본 유형은 **xsd:restriction** 요소의 기본 특성으로 지정됩니다. 각 **facet** 요소는 제한 요소 내에서 지정됩니다. 사용 가능한 **facet** 및 유효한 설정은 기본 유형에 따라 다릅니다. 예를 들어 **xsd:string** 에는 다음을 포함한 여러 **facet**가 있습니다.

- 길이
- **minLength**
- **maxLength**
- 패턴
- **whitespace**

예 34.2. "우편 코드 간단한 유형" 미국 상태에 사용되는 두 문자 우편 코드를 나타내는 간단한 유형에 대한 정의를 보여줍니다. 대문자는 두 개만 포함할 수 있습니다. **TX** 는 유효한 값이지만 **tx** 또는 **tX** 는 유효한 값이 아닙니다.

예 34.2. 우편 코드 간단한 유형

```
<xsd:simpleType name="postalCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}" />
  </xsd:restriction>
</xsd:simpleType>
```

34.2.4. Java로의 매핑

Apache CXF는 간단한 유형의 기본 유형의 **Java** 유형에 사용자 정의 간단한 유형을 매핑합니다. 따라서 예 34.2. "우편 코드 간단한 유형" 에 표시된 간단한 유형의 **postalCode** 를 사용하는 모든 메시지는 **postalCode** 의 기본 유형이 **xsd:string** 이므로 **String** 에 매핑됩니다. 예를 들어 예 34.3. "간단한 유형의 크레딧 요청" 에 표시된 **WSDL** 조각은 문자열 의 매개 변수인 **postalCode** 를 사용하는 **Java** 메서드 **state()** 가 생성됩니다.

예 34.3. 간단한 유형의 크레딧 요청

```
<message name="stateRequest">
  <part name="postalCode" type="postalCode" />
</message>
...
<portType name="postalSupport">
  <operation name="state">
    <input message="tns:stateRequest" name="stateRec" />
    <output message="tns:stateResponse" name="credResp" />
  </operation>
</portType>
```

34.2.5. facet 강제

기본적으로 **Apache CXF**는 간단한 유형을 제한하는 데 사용되는 **facet**를 적용하지 않습니다. 그러나 스키마 유효성 검사를 활성화하여 **facet**를 적용하도록 **Apache CXF** 엔드포인트를 구성할 수 있습니다.

스키마 유효성 검사를 사용하도록 **Apache CXF** 끝점을 구성하려면 **schema-validation-enabled** 속성을 **true** 로 설정합니다. 예 34.4. "서비스 공급자 구성 스키마 유효성 검사 사용" 스키마 유효성 검사를 사용하는 서비스 공급자의 구성을 보여줍니다.

예 34.4. 서비스 공급자 구성 스키마 유효성 검사 사용

```
<jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                wsdlLocation="wsdl/hello_world.wsdl"
                createdFromAPI="true">
  <jaxws:properties>
    <entry key="schema-validation-enabled" value="BOTH" />
  </jaxws:properties>
</jaxws:endpoint>
```

스키마 유효성 검사 구성에 대한 자세한 내용은 24.3.4.7절. "스키마 유효성 검사 유형 값" 을 참조하십시오.

34.3. 열거형

34.3.1. 개요

XML 스키마에서 열거된 형식은 xsd:enumeration facet를 사용하여 정의된 간단한 유형입니다. atomic 단순 형식과 달리 Java 열거 형에 매핑됩니다.

34.3.2. XML 스키마에서 열거된 유형 정의

열거형은 xsd:enumeration facet를 사용하는 간단한 유형입니다. 각 xsd:enumeration facet는 열거된 형식에 대해 가능한 하나의 값을 정의합니다.

예 34.5. "XML 스키마 정의 Enumeration" 열거된 형식에 대한 정의를 표시합니다.Shows the definition for an enumerated type. 다음과 같은 가능한 값이 있습니다.

- big
- large
- mungo
- gargantuan

예 34.5. XML 스키마 정의 Enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
    <enumeration value="gargantuan"/>
  </restriction>
```

34.3.3. Java로의 매핑

기본 형식이 **xsd:string** 인 **XML** 스키마 열거형은 **Java** 열거형 형식에 자동으로 매핑됩니다. 38.4절. "Enumeration 매핑 사용자 정의" 에 설명된 사용자 지정을 사용하여 다른 기본 유형과 함께 열거를 매핑하도록 코드 생성기에 지시할 수 있습니다.

enum 형식은 다음과 같이 생성됩니다.

1. 형식의 이름은 단순 형식 정의의 **name** 특성에서 가져와 **Java** 식별자로 변환됩니다.

   일반적으로 이는 **XML** 스키마 이름의 첫 번째 문자를 대문자로 변환하는 것을 의미합니다. **XML** 스키마 이름의 첫 번째 문자가 유효하지 않은 문자인 경우 아래에 있는_) 앞에 이름이 붙습니다.

2. 각 열거형 **facet**에 대해 열거형 상수는 **value** 특성의 값에 따라 생성됩니다.**For each enumeration facet, an enum constant is generated based on the value of the value attribute.**

   상수 이름은 값의 모든 소문자를 대문자로 변환하여 파생됩니다.

3. 열거형의 기본 형식에서 매핑된 **Java** 형식을 사용하는 생성자가 생성됩니다.**A constructor is generated that takes the Java type mapped from the enumeration's base type.**

4. **value()** 라는 공용 메서드는 형식의 인스턴스로 표시되는 **facet** 값에 액세스하기 위해 생성됩니다.**A public method called value() is generated to access the facet value that is represented by an instance of the type.**

   value() 메서드의 반환 유형은 **XML** 스키마 유형의 기본 유형입니다.

5. **fromValue()** 라는 공용 메서드가 생성되어 **facet** 값을 기반으로 **enum** 형식의 인스턴스를 만듭니다.

   value() 메서드의 매개 변수 유형은 **XML Schema** 유형의 기본 유형입니다.

6. 클래스는 **@XmlEnum** 주석을 사용하여 장식됩니다.

예 34.5. "XML 스키마 정의 Enumeration" 에 정의된 열거된 유형은 예 34.6. "String Bases XML Schema Enumeration에 대해 생성된 Enumerated Type" 로 표시된 **enum** 유형에 매핑됩니다.

예 34.6. String Bases XML Schema Enumeration에 대해 생성된 Enumerated Type

```
@XmlType(name = "widgetSize")
@XmlEnum
public enum WidgetSize {

    @XmlEnumValue("big")
    BIG("big"),
    @XmlEnumValue("large")
    LARGE("large"),
    @XmlEnumValue("mungo")
    MUNGO("mungo"),
    @XmlEnumValue("gargantuan")
    GARGANTUAN("gargantuan");
    private final String value;

    WidgetSize(String v) {
        value = v;
    }

    public String value() {
        return value;
    }

    public static WidgetSize fromValue(String v) {
        for (WidgetSize c: WidgetSize.values()) {
            if (c.value.equals(v)) {
                return c;
            }
        }
        throw new IllegalArgumentException(v);
    }

}
```

34.4. LISTS

34.4.1. 개요

XML 스키마는 공백으로 구분된 단순 형식 목록인 데이터 형식을 정의하는 메커니즘을 지원합니다. 목록 유형을 사용하는 요소 **primeList** 의 예는 예 34.7. “목록 유형 예” 에 표시되어 있습니다.

예 34.7. 목록 유형 예

```
<primeList>1 3 5 7 9 11 13<\primeList>
```

XML Schema 목록 유형은 일반적으로 Java List<T> 오브젝트에 매핑됩니다. 이 패턴과 유일한 변형은 메시지 부분이 XML Schema 목록 유형의 인스턴스에 직접 매핑되는 경우입니다.

34.4.2. XML 스키마에서 목록 유형 정의

XML Schema 목록 유형은 간단한 유형이며, 이러한 유형은 simpleType 요소를 사용하여 정의됩니다. 목록 유형을 정의하는 데 사용되는 가장 일반적인 구문은 예 34.8. "XML 스키마 목록 형식 구문" 에 표시되어 있습니다.

예 34.8. XML 스키마 목록 형식 구문

```
<simpleType name="listType">
  <list itemType="atomicType">
    <facet value="value" />
    <facet value="value" />
    ...
  </list>
</simpleType>
```

atomicType 에 대해 제공되는 값은 목록에 있는 요소의 유형을 정의합니다. xsd:int 또는 xsd:string 또는 목록이 아닌 사용자 정의 단순 형식처럼 XML 스키마 원자성 유형으로만 빌드될 수 있습니다.It can only be one of the built in XML Schema atomic types, such as xsd:int or xsd:string, or a user-defined simple type that is not a list.

목록 유형에 나열된 요소 유형을 정의하는 것 외에도 facets를 사용하여 목록 유형의 속성을 추가로 제한할 수도 있습니다. 표 34.3. "유형 Facets 나열" 목록 유형에서 사용하는 facet를 보여줍니다.

표 34.3. 유형 Facets 나열

facet	효과
길이	목록 형식의 인스턴스에 있는 요소 수를 정의합니다.Defines the number of elements in an instance of the list type.
minLength	목록 형식의 인스턴스에 허용되는 최소 요소 수를 정의합니다.
maxLength	목록 형식의 인스턴스에 허용되는 최대 요소 수를 정의합니다.Defines the maximum number of elements allowed in an instance of the list type.
enumeration	목록 형식의 인스턴스에 있는 요소에 허용되는 값을 정의합니다.Defines the allowable values for elements in an instance of the list type.

facet	효과
패턴	목록 형식의 인스턴스에 있는 요소의 어휘 형식을 정의합니다.Defines the lexical form of the elements in an instance of the list type. 패턴은 정규식을 사용하여 정의됩니다.

*예를 들어 예 34.7. “목록 유형 예” 에 표시된 **simpleList** 요소에 대한 정의는 예 34.9. “목록 유형 정의” 로 표시됩니다.*

예 34.9. 목록 유형 정의

```
<simpleType name="primeListType">
  <list itemType="int"/>
</simpleType>
<element name="primeList" type="primeListType"/>
```

예 34.8. “XML 스키마 목록 형식 구문” 에 표시된 구문 외에도 예 34.10. “목록 유형의 대체 구문” 에 표시된 덜 일반적인 구문을 사용하여 목록 유형을 정의할 수도 있습니다.

예 34.10. 목록 유형의 대체 구문

```
<simpleType name="listType">
  <list>
    <simpleType>
      <restriction base="atomicType">
        <facet value="value"/>
        <facet value="value"/>
        ...
      </restriction>
    </simpleType>
  </list>
  </simpleType>
```

34.4.3. 목록 유형 요소를 Java에 매핑

*요소가 목록 유형을 정의하면 목록 유형이 컬렉션 속성에 매핑됩니다. 컬렉션 속성은 **Java List<T>** 개체입니다. **List<T>** 에서 사용하는 템플릿 클래스는 목록의 기본 유형에서 매핑되는 래퍼 클래스입니다. 예를 들어 예 34.9. “목록 유형 정의” 에 정의된 목록 유형은 **List<Integer>** 에 매핑됩니다.*

래퍼 유형 매핑에 대한 자세한 내용은 "래퍼 클래스" 을 참조하십시오.

34.4.4. Java에 목록 유형 매개변수 매핑

메시지 파트가 목록 유형으로 정의되거나 목록 유형의 요소에 매핑되는 경우 결과 메서드 매개 변수는 **List<T>** 개체 대신 배열에 매핑됩니다.**When a message part is defined as a list type, or is mapped to an element of a list type, the resulting method parameter is mapped to an array instead of a List<T > object.** 배열의 기본 유형은 목록 유형의 기본 클래스의 래퍼 클래스입니다.

예를 들어 예 34.11. "목록 유형 메시지 파트가 있는 WSDL" 의 **WSDL** 조각은 예 34.12. "목록 유형 매개 변수가 있는 Java 메서드" 로 표시된 메서드 서명이 생성됩니다.

예 34.11. 목록 유형 메시지 파트가 있는 WSDL

```
<definitions ...>
  ...
  <types ...>
    <schema ... >
      <simpleType name="primeListType">
        <list itemType="int"/>
      </simpleType>
      <element name="primeList" type="primeListType"/>
    </schemas>
  </types>
  <message name="numRequest"> <part name="inputData" element="xsd1:primeList" />
</message>
  <message name="numResponse">;
    <part name="outputData" type="xsd:int">
  ...
  <portType name="numberService">
    <operation name="primeProcessor">
      <input name="numRequest" message="tns:numRequest" />
      <output name="numResponse" message="tns:numResponse" />
    </operation>
    ...
  </portType>
  ...
</definitions>
```

예 34.12. 목록 유형 매개 변수가 있는 Java 메서드

```
public interface NumberService {

    @XmlList
    @WebResult(name = "outputData", targetNamespace = "", partName = "outputData")
    @WebMethod
    public int primeProcessor(
```

```
        @WebParam(partName = "inputData", name = "primeList", targetNamespace = "...")
    java.lang.Integer[] inputData
    );
}
```

34.5. UNIONS

34.5.1. 개요

XML 스키마에서 공용 구조체는 여러 단순 형식 중 하나일 수 있는 형식을 설명할 수 있는 구문입니다.**In XML Schema, a union is a construct that allows you to describe a type whose data can be one of a number of simple types.** 예를 들어 값이 정수 **1**이거나 첫 번째 문자열에 해당하는 형식을 정의할 수 있습니다.**For example, you can define a type whose value is either the integer 1 or the string first.** 공용 구조체는 **Java Strings**에 매핑됩니다.

34.5.2. XML 스키마로 정의

XML Schema unions는 **simpleType** 요소를 사용하여 정의됩니다. **Union**의 멤버 유형을 정의하는 최소 하나의 **union** 요소를 포함합니다. 멤버 형식 공용 구조체는 공용 구조체의 인스턴스에 저장할 수 있는 유효한 데이터 형식입니다.**The member types of the union are the valid types of data that can be stored in an instance of the union. union** 요소의 **memberTypes** 특성을 사용하여 정의됩니다. **memberTypes** 특성 값은 하나 이상의 정의된 단순 형식 이름 목록을 포함합니다. **예 34.13. "간단한 통합 유형"** 정수 또는 문자열을 저장할 수 있는 공용 구조체의 정의를 보여 줍니다.**Shows the definition of a union that can store either an integer or a string.**

예 34.13. 간단한 통합 유형

```
<simpleType name="orderNumUnion">
  <union memberTypes="xsd:string xsd:int" />
</simpleType>
```

명명된 형식을 결합의 멤버 유형으로 지정하는 것 외에도 무명 간단한 유형을 **union**의 멤버 유형으로 정의할 수도 있습니다.**In addition to specifying named types as a member type of a union, you can also define an anonymous simple type as a member type of a union.** 이는 **union** 요소 내에 **anonymous** 유형 정의를 추가하여 수행됩니다. **예 34.14. "익명 회원 유형(Anonymous Member type)"** 유효한 정수의 가능한 값을 **1**에서 **10**까지 제한하는 익명 멤버 형식이 포함된 통합의 예를 보여줍니다.

예 34.14. 익명 회원 유형**(Anonymous Member type)**

```
<simpleType name="restrictedOrderNumUnion">
  <union memberTypes="xsd:string">
    <simpleType>
      <restriction base="xsd:int">
        <minInclusive value="1" />
```

```
      <maxInclusive value="10" />
    </restriction>
  </simpleType>
</union>
</simpleType>
```

34.5.3. Java로의 매핑

XML Schema union 유형은 **Java String** 개체에 매핑됩니다. 기본적으로 **Apache CXF**는 생성된 오브젝트의 내용을 확인하지 않습니다. **Apache CXF**의 유효성을 검사하려면 **"facet 강제"** 에 설명된 대로 스키마 유효성 검사를 사용하도록 런타임을 구성해야 합니다.

34.6. 간단한 유형 대체

34.6.1. 개요

XML은 **xsi:type** 특성을 사용하여 호환되는 유형 간 간단한 유형 대체를 허용합니다. 그러나 간단한 유형의 기본 매핑을 **Java** 기본 유형으로 매핑하지만 간단한 유형 대체를 완전히 지원하지는 않습니다. 런타임에서는 기본 단순 형식 대체를 처리할 수 있지만 정보가 손실됩니다. 코드 생성기는 무손실 간단한 유형 대체를 용이하게하는 **Java** 클래스를 생성하도록 사용자 지정할 수 있습니다.

34.6.2. 기본 매핑 및 마샬링

Java 기본 유형은 유형 대체를 지원하지 않기 때문에 **Java** 프리미티브 유형으로 간단한 유형의 기본 매핑은 간단한 유형 대체를 지원하는 데 문제가 있습니다. 유형을 정의하는 스키마에서 허용하는 경우에도 **int** 가 예상되는 변수로 단축 을 수행하려는 경우 **Java** 가상 머신은 **balk**합니다.

Java 유형 시스템에서 적용되는 제한 사항을 충족하기 위해 **Apache CXF**에서는 요소의 **xsi:type** 속성 값이 다음 조건 중 하나를 충족할 때 간단한 유형 대체를 허용합니다.

- 요소의 스키마 형식과 호환되는 기본 형식을 지정합니다.**Specifies a primitive type that is compatible with the element's schema type.**
- 요소의 스키마 형식에서 제한으로 파생되는 형식을 지정합니다.**Specifies a type that derives by restriction from the element's schema type.**
- 요소의 스키마 형식에서 확장으로 파생되는 복잡한 형식을 지정합니다.**Specifies a complex type that derives by extension from the element's schema type.**

런타임이 유형 대체를 수행하는 경우 요소의 **xsi:type** 속성에 지정된 유형에 대한 지식이 유지되지 않습니다. 형식 대체가 복합 형식에서 간단한 형식으로 된 경우 단순 유형과 직접 관련된 값만 유지됩니다.**If the type substitution is from a complex type to a simple type, only the value directly related to the simple type is preserved.** 확장에 의해 추가된 다른 모든 요소 및 특성은 손실됩니다.

34.6.3. 무손실 유형 대체 지원

간단한 유형의 생성을 사용자 지정하여 다음과 같은 방법으로 간단한 유형 대체의 무손실 지원을 용이하게 할 수 있습니다.

- **globalBindings** 사용자 지정 요소의 **mapSimpleTypeDef** 를 **true** 로 설정합니다.

  이를 통해 글로벌 범위에 정의된 모든 단순 형식에 대해 **Java** 값 클래스를 생성하도록 코드 생성기가 지시합니다.

  자세한 내용은 **38.3절. "Simple Types을 위한 Java 클래스 생성"** 에서 참조하십시오.

- **javaType** 요소를 전역**Bindings** 사용자 지정 요소에 추가합니다.

  이렇게 하면 코드 생성기가 **XML** 스키마 기본 형식의 모든 인스턴스를 특정 클래스의 특정 클래스에 매핑하도록 지시합니다.**This instructs the code generators to map all instances of an XML Schema primitive type to s specific class of object.**

  자세한 내용은 **38.2절. "XML 스키마의 Java 클래스 지정"** 에서 참조하십시오.

- 사용자 정의 하려는 특정 요소에 **baseType** 사용자 지정 요소를 추가 합니다.**Add a baseType custom element to the specific elements you want to customize.**

  baseType 사용자 지정 요소를 사용하면 속성을 나타내기 위해 생성된 **Java** 유형을 지정할 수 있습니다. 단순 형식 대체에 가장 적합한 호환성을 확인하려면 **java.lang.Object** 를 기본 유형으로 사용합니다.

  자세한 내용은 **38.6절. "Element 또는 Attribute의 기본 유형 지정"** 에서 참조하십시오.

35장. 복합 유형 사용

초록

복잡한 형식은 여러 요소를 포함할 수 있으며 특성이 있을 수 있습니다. 유형 정의로 표시되는 데이터를 저장할 수 있는 **Java** 클래스에 매핑됩니다. 일반적으로 콘텐츠 모델의 요소와 속성을 나타내는 속성 세트가 있는 빈에 매핑됩니다.

35.1. 기본 COMPLEX 유형 매핑

35.1.1. 개요

XML 스키마 복잡한 형식은 단순 형식보다 복잡한 정보를 포함하는 구문을 정의합니다. 가장 간단한 복잡한 유형은 특성을 사용하여 빈 요소를 정의합니다. 복잡한 더 복잡한 유형은 요소 컬렉션으로 구성됩니다.

기본적으로 **XML** 스키마 복잡한 유형은 **XML** 스키마 정의에 나열된 각 요소 및 특성을 나타내는 멤버 변수와 함께 **Java** 클래스에 매핑됩니다. 클래스에는 각 멤버 변수에 대해 **setter** 및 **getter**가 있습니다.

35.1.2. XML 스키마로 정의

XML 스키마 복합 유형은 **complexType** 요소를 사용하여 정의됩니다. **complexType** 요소는 데이터의 구조를 정의하는 데 사용되는 나머지 요소를 래핑합니다. 명명된 형식 정의의 부모 요소 또는 요소에 저장된 정보의 구조를 익명으로 정의하는 요소의 자식으로 나타날 수 있습니다.**It can appear either as the parent element of a named type definition, or as the child of an element element anonymous defining the structure of the information stored in the element. complexType** 요소를 사용하여 명명된 유형을 정의할 때 **name** 특성을 사용해야 합니다. **name** 속성은 유형을 참조할 수 있는 고유 식별자를 지정합니다.

하나 이상의 요소를 포함하는 복잡한 유형 정의에는 표 35.1. “Complex Type에서 How Elements Appear를 정의하는 데 필요한 요소” 에 설명된 하위 요소 중 하나가 있습니다. 이러한 요소는 형식의 인스턴스에 지정된 요소가 표시되는 방식을 결정합니다.

표 35.1. Complex Type에서 How Elements Appear를 정의하는 데 필요한 요소

요소	설명
all	복잡한 형식의 일부로 정의된 모든 요소가 형식의 인스턴스에 표시되어야 합니다.All of the elements defined as part of the complex type must appear in an instance of the type. 그러나 그들은 어떤 순서로든 나타날 수 있습니다.

요소	설명
choice	복잡한 형식의 일부로 정의된 요소 중 하나만 형식의 인스턴스에 나타날 수 있습니다.Only one of the elements defined as part of the complex type can appear in an instance of the type.
순서	복잡한 형식의 일부로 정의된 모든 요소는 형식의 인스턴스에 표시되어야 하며 형식 정의에 지정된 순서로 표시되어야 합니다.All of the elements defined as part of the complex type must appear in an instance of the type, and they must also appear in the order specified in the type definition.

참고

복잡한 유형 정의가 특성만 사용하는 경우 표 35.1. "Complex Type에서 How Elements Appear를 정의하는 데 필요한 요소" 에 설명된 요소 중 하나가 필요하지 않습니다.

*요소를 표시할 방법을 결정한 후 정의에 하나 이상의 요소 자식을 추가하여 요소를 정의합니다.**After deciding how the elements will appear, you define the elements by adding one or more element children to the definition.***

*예 35.1. "XML Schema Complex Type" **XML** 스키마에서 복잡한 유형 정의를 보여줍니다.*

예 35.1. XML Schema Complex Type

```
<complexType name="sequence">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="street" type="xsd:short" />
    <element name="city" type="xsd:string" />
    <element name="state" type="xsd:string" />
    <element name="zipCode" type="xsd:string" />
  </sequence>
</complexType>
```

35.1.3. Java로의 매핑

***XML** 스키마 복잡한 유형은 **Java** 클래스에 매핑됩니다. 복잡한 형식 정의의 각 요소는 **Java** 클래스의 멤버 변수에 매핑됩니다. **getter** 및 **setter** 메서드도 복잡한 유형의 각 요소에 대해 생성됩니다.*

생성된 모든 **Java** 클래스는 **@XmlType** 주석으로 장식됩니다. 이름이 지정된 복잡한 유형의 매핑이 있는 경우 주석 이름은 **complexType** 요소의 **name** 속성 값으로 설정됩니다. 복잡한 유형이 요소 정의의 일부로 정의되면 **@XmlType** 주석의 **name** 속성 값은 요소 요소 의 **name** 속성 값입니다.

참고

"인라인 유형을 사용하여 요소의 매핑" 에서 설명한 것처럼, 요소 정의의 일부로 정의된 복잡한 형식에 대해 생성되는 경우 생성된 클래스는 **@XmlRootElement** 주석으로 장식됩니다.

XML 스키마 복잡한 유형의 요소를 처리하는 방법을 나타내는 지침을 제공하기 위해 코드 생성기는 클래스 및 해당 멤버 변수를 장식하는 데 사용되는 주석을 변경합니다.

모든 복잡성 유형

모든 복잡한 유형은 **all** 요소를 사용하여 정의됩니다. 주석이 다음과 같이 표시됩니다.

- **@XmlType** 주석의 **propOrder** 속성이 비어 있습니다.
- 각 요소는 **@XmlElement** 주석으로 장식됩니다.
- **@XmlElement** 주석의 필수 속성이 **true** 로 설정됩니다.

예 35.2. "모든 복잡성 유형의 매핑" 두 개의 요소가 있는 모든 복잡한 유형의 매핑을 표시합니다.

예 35.2. 모든 복잡성 유형의 매핑

```
@XmlType(name = "all", propOrder = {

})
public class All {
    @XmlElement(required = true)
    protected BigDecimal amount;
    @XmlElement(required = true)
    protected String type;

    public BigDecimal getAmount() {
        return amount;
    }
```

```
    public void setAmount(BigDecimal value) {
        this.amount = value;
    }

    public String getType() {
        return type;
    }

    public void setType(String value) {
        this.type = value;
    }
}
```

선택 복잡성 유형

선택 가능한 복잡한 유형은 **choice** 요소를 사용하여 정의됩니다. 주석이 다음과 같이 표시됩니다.

- **@XmlType** 주석의 **propOrder** 속성은 **XML** 스키마 정의에 표시되는 순서대로 요소의 이름을 나열합니다.
- 멤버 변수에는 주석이 없습니다.

예 35.3. “선택 복잡성 유형의 매핑” 두 개의 요소가 있는 선택 복잡한 유형의 매핑을 표시합니다.

예 35.3. 선택 복잡성 유형의 매핑

```
@XmlType(name = "choice", propOrder = {
    "address",
    "floater"
})
public class Choice {

    protected Sequence address;
    protected Float floater;

    public Sequence getAddress() {
        return address;
    }

    public void setAddress(Sequence value) {
        this.address = value;
    }

    public Float getFloater() {
        return floater;
    }
```

```
    public void setFloater(Float value) {
        this.floater = value;
    }

}
```

시퀀스 복잡성 유형

*시퀀스 복합 유형은 **sequence** 요소를 사용하여 정의됩니다. 다음과 같이 주석이 추가됩니다.*

- *@XmlType 주석의 **propOrder** 속성은 **XML** 스키마 정의에 표시되는 순서대로 요소의 이름을 나열합니다.*
- *각 요소는 **@XmlElement** 주석으로 장식됩니다.*
- *@XmlElement 주석의 필수 속성이 **true** 로 설정됩니다.*

예 35.4. “시퀀스 Complex 유형의 매핑” 예 35.1. “XML Schema Complex Type” 에 정의된 복잡한 유형의 매핑을 보여줍니다.

예 35.4. 시퀀스 Complex 유형의 매핑

```
@XmlType(name = "sequence", propOrder = {
    "name",
    "street",
    "city",
    "state",
    "zipCode"
})
public class Sequence {

    @XmlElement(required = true)
    protected String name;
    protected short street;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String zipCode;

    public String getName() {
        return name;
    }
```

```
    public void setName(String value) {
        this.name = value;
    }

    public short getStreet() {
        return street;
    }

    public void setStreet(short value) {
        this.street = value;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String value) {
        this.city = value;
    }

    public String getState() {
        return state;
    }

    public void setState(String value) {
        this.state = value;
    }

    public String getZipCode() {
        return zipCode;
    }

    public void setZipCode(String value) {
        this.zipCode = value;
    }
}
```

35.2. 속성

35.2.1. 개요

Apache CXF는 복잡한 **Type** 요소 범위 내에서 특성 요소 및 **attribute Group** 요소의 사용을 지원합니다. **XML** 문서 특성 선언에 대한 구조를 정의할 때 태그가 포함된 값이 아니라 태그 내에 지정된 정보를 추가할 수 있는 수단을 제공합니다. 예를 들어 **XML** 스키마의 **XML** 요소 **<값 currency ="euro">410<\value>**을 설명할 때 통화 속성은 예 35.5. “XML Schema Defining and Attribute” 과 같이 특성 요소를 사용하여 설명됩니다.

attributeGroup 요소를 사용하면 스키마에서 정의한 모든 복잡한 유형에서 참조할 수 있는 재사용 가능한 속성 그룹을 정의할 수 있습니다. 예를 들어 특성 카테고리 와 **pubDate** 를 사용하는 일련의 요소를

정의하는 경우 이러한 속성을 사용하여 특성 그룹을 정의하고 이를 사용하는 모든 요소에서 해당 요소를 참조할 수 있습니다. 이는 예 35.7. "특성 그룹 정의" 에 표시됩니다.

애플리케이션 논리 개발에 사용할 데이터 유형을 설명하는 경우, 속성을 선택적 또는 필수 로 설정된 특성은 구조의 요소로 취급됩니다. 복잡한 형식 설명에 포함된 각 특성 선언에 대해 요소(element)는 적절한 getter 및 setter 메서드와 함께 특성 클래스에 생성됩니다.

35.2.2. XML 스키마에서 속성 정의

XML Schema 특성 요소에는 특성을 식별하는 데 사용되는 필수 특성이 한 개 있습니다.An XML Schema attribute element has one required attribute, name, that is used to identify the attribute. 또한 표 35.2. "선택 사항 Attributes XML 스키마에서 특성을 정의하는 데 사용됨" 에서 설명하는 네 가지 선택적 속성도 있습니다.

표 35.2. 선택 사항 Attributes XML 스키마에서 특성을 정의하는 데 사용됨

속성	설명
Use	특성이 필요한지 여부를 지정합니다. 유효한 값은 **필수**,**선택 사항 또는 금지** 됩니다. **선택** 사항이 기본값입니다.
type	특성에서 사용할 수 있는 값의 형식을 지정합니다.Specifies the type of value the attribute can take. 사용되지 않는 경우 특성의 스키마 유형을 인라인으로 정의해야 합니다.
default	특성에 사용할 기본값을 지정합니다. **특성 요소의 use 특성** 이 **선택 옵션으로** 설정된 경우에만 사용됩니다.
fixed	특성에 사용할 고정 값을 지정합니다.Specifies a fixed value to use for the attribute. **특성 요소의 use 특성** 이 **선택 옵션으로** 설정된 경우에만 사용됩니다.

예 35.5. "XML Schema Defining and Attribute" 값이 문자열인 특성을 정의하는 특성 요소를 표시합니다.Shows an attribute element defining an attribute, currency, whose value is a string.

> *예 35.5. XML Schema Defining and Attribute*
>
> ```
> <element name="value">
>   <complexType>
>     <xsd:simpleContent>
>       <xsd:extension base="xsd:integer">
>         <xsd:attribute name="currency" type="xsd:string"
>                        use="required"/>
> ```

```
    </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>
</xsd:element>
```

특성 요소에서 **type** 속성 을 생략하면 데이터 형식을 줄 바꿈으로 설명해야 합니다. 예 35.6. "In-Line Data Description의 특성" 특성, 카테고리, 값을 **autobiography,non-fiction** 또는 **fiction** 을 사용할 수 있는 속성 요소를 표시합니다.

예 35.6. In-Line Data Description의 특성

```
<attribute name="category" use="required">
 <simpleType>
  <restriction base="xsd:string">
   <enumeration value="autobiography"/>
   <enumeration value="non-fiction"/>
   <enumeration value="fiction"/>
  </restriction>
 </simpleType>
</attribute>
```

35.2.3. XML 스키마에서 특성 그룹 사용

복잡한 유형 정의에서 특성 그룹을 사용하는 것은 두 가지 단계 프로세스입니다.

1. 특성 그룹을 정의합니다.

   특성 그룹은 여러 특성 하위 요소가 있는 **attribute Group** 요소를 사용하여 정의합니다. **attributeGroup** 에는 특성 그룹을 참조하는 데 사용되는 문자열을 정의하는 **name** 속성이 필요합니다. 특성 요소는 특성 그룹의 멤버를 정의하고 "XML 스키마에서 속성 정의" 와 같이 지정됩니다. 예 35.7. "특성 그룹 정의" 특성 그룹 **catalogIndecies** 에 대한 설명을 보여줍니다. 속성 그룹에는 두 개의 멤버가 있습니다. 범주 는 선택 사항이며 **pubDate** 는 필수 항목입니다.

   예 35.7. 특성 그룹 정의

   ```
   <attributeGroup name="catalogIndices">
     <attribute name="category" type="catagoryType" />
     <attribute name="pubDate" type="dateTime"
                use="required" />
   </attributeGroup>
   ```

2. 복합 유형의 정의에서 특성 그룹을 사용합니다.

ref 속성과 함께 **attributeGroup** 요소를 사용하여 복잡한 유형 정의에서 특성 그룹을 사용합니다. **ref** 특성의 값은 유형 정의의 일부로 사용하려는 특성 그룹이 지정된 이름입니다. 예를 들어 복잡한 유형 **dvdType** 에서 특성 **group catalogIndecies** 를 사용하려면 *예 35.8. "특성 그룹을 사용하는 복합 유형"* 에 표시된 대로 **<attributeGroup ref="catalogIndecies" />**를 사용합니다.

예 35.8. 특성 그룹을 사용하는 복합 유형

```
<complexType name="dvdType">
  <sequence>
    <element name="title" type="xsd:string" />
    <element name="director" type="xsd:string" />
    <element name="numCopies" type="xsd:int" />
  </sequence>
  <attributeGroup ref="catalogIndices" />
</complexType>
```

35.2.4. Java에 속성 매핑

특성은 멤버 요소가 **Java**에 매핑되는 방식과 동일하게 **Java**에 매핑됩니다. 필수 속성 및 선택적 특성은 생성된 **Java** 클래스의 멤버 변수에 매핑됩니다. 멤버 변수는 **@XmlAttribute** 주석으로 장식됩니다. 속성이 필요한 경우 **@XmlAttribute** 주석의 **required** 속성이 **true** 로 설정됩니다.

예 35.9. "techDoc Description" 에 정의된 복잡한 유형은 *예 35.10. "techDoc Java Class"* 에 표시된 **Java** 클래스에 매핑됩니다.

예 35.9. techDoc Description

```
<complexType name="techDoc">
  <all>
    <element name="product" type="xsd:string" />
    <element name="version" type="xsd:short" />
  </all>
  <attribute name="usefullness" type="xsd:float"
             use="optional" default="0.01" />
</complexType>
```

예 35.10. techDoc Java Class

```
@XmlType(name = "techDoc", propOrder = {

})
public class TechDoc {
```

```
    @XmlElement(required = true)
    protected String product;
    protected short version;
    @XmlAttribute protected Float usefullness;

    public String getProduct() {
        return product;
    }

    public void setProduct(String value) {
        this.product = value;
    }

    public short getVersion() {
        return version;
    }

    public void setVersion(short value) {
        this.version = value;
    }

    public float getUsefullness() { if (usefullness == null) { return 0.01F; } else { return usefullness; }
}

    public void setUsefullness(Float value) {
        this.usefullness = value;
    }
}
```

*예 35.10. "techDoc Java Class" 에 표시된 대로 **default** 속성 및 **fixed** 특성은 특성에 대해 생성된 **getter** 메서드에 코드를 추가하도록 코드 생성기에 지시합니다. 이 추가 코드는 값이 설정되지 않은 경우 지정된 값이 반환되도록 합니다.*

중요

*고정 속성은 **default** 특성과 동일하게 처리됩니다. 고정 특성을 **Java** 상수로 처리하려면 **38.5절. "Fixed Value Attribute Mapping"** 에 설명된 사용자 지정을 사용할 수 있습니다.*

35.2.5. 속성 그룹을 Java로 매핑

*특성 그룹은 그룹 멤버가 형식 정의에서 명시적으로 사용되는 것처럼 **Java**에 매핑됩니다. 특성 그룹에 세 개의 멤버가 있고 복잡한 형식으로 사용되는 경우 해당 형식에 대해 생성된 클래스에는 특성 그룹의 각 멤버에 대해 **getter** 및 **setter** 메서드와 함께 멤버 변수가 포함됩니다. 예를 들어 예 35.8. "특성 그룹을 사용하는 복합 유형" 에 정의된 복잡한 유형인 **Apache CXF**는 예 35.11. "dvdType Java Class" 에 표시된 대로 특성 그룹의 멤버를 지원하기 위해 멤버 변수 카테고리 및 **pubDate** 를 포함하는 클래스를 생성합니다.*

예 35.11. dvdType Java Class

```
@XmlType(name = "dvdType", propOrder = {
    "title",
    "director",
    "numCopies"
})
public class DvdType {

    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String director;
    protected int numCopies;
    @XmlAttribute protected CatagoryType category; @XmlAttribute(required = true)
@XmlSchemaType(name = "dateTime") protected XMLGregorianCalendar pubDate;

    public String getTitle() {
        return title;
    }

    public void setTitle(String value) {
        this.title = value;
    }

    public String getDirector() {
        return director;
    }

    public void setDirector(String value) {
        this.director = value;
    }

    public int getNumCopies() {
        return numCopies;
    }

    public void setNumCopies(int value) {
        this.numCopies = value;
    }

    public CatagoryType getCatagory() {
        return catagory;
    }

    public void setCatagory(CatagoryType value) {
        this.catagory = value;
    }

    public XMLGregorianCalendar getPubDate() {
        return pubDate;
    }

    public void setPubDate(XMLGregorianCalendar value) {
        this.pubDate = value;
```

```
    }
}
```

35.3. 간단한 형식에서 복합 유형 파생

35.3.1. 개요

Apache CXF는 간단한 유형에서 복잡한 유형의 파생을 지원합니다. 간단한 유형은 **sub-elements** 및 **attributes** 둘 다 정의로 구성됩니다. 따라서 간단한 유형에서 복잡한 유형을 파생하는 주된 이유 중 하나는 간단한 유형에 특성을 추가하는 것입니다.

간단한 유형에서 복잡한 유형을 파생하는 두 가지 방법이 있습니다.

- 확장 기능 사용
- 제한 사항별

35.3.2. 확장의 파생

예 35.12. "Simple Type by Extension에서 Complex 유형 파생" 통화 특성을 포함하는 **xsd:decimal** **primitive** 유형에서 확장에 의해 파생되는 복잡한 **type**인 **International Price** 의 예를 보여줍니다.

예 35.12. Simple Type by Extension에서 Complex 유형 파생

```
<complexType name="internationalPrice">
  <simpleContent>
    <extension base="xsd:decimal">
      <attribute name="currency" type="xsd:string"/>
    </extension>
  </simpleContent>
  </complexType>
```

simpleContent 요소는 새 형식에 하위 요소가 포함되어 있지 않음을 나타냅니다. **extension** 요소는 새 유형이 **xsd:decimal** 을 확장하도록 지정합니다.

35.3.3. 제한별 유도

예 35.13. "단순 유형에서 제한으로 복합 유형 파생" xsd:string 에서 제한으로 파생되는 복잡한 유형 idType 의 예를 보여줍니다. 정의된 형식은 xsd:string의 가능한 값을 10자 길이의 값으로 제한합니다. 또한 형식에 특성을 추가합니다.

예 35.13. 단순 유형에서 제한으로 복합 유형 파생

```
<complexType name="idType">
  <simpleContent>
    <restriction base="xsd:string">
     <length value="10" />
     <attribute name="expires" type="xsd:dateTime" />
    </restriction>
  </simpleContent>
</complexType>
```

예 35.12. "Simple Type by Extension에서 Complex 유형 파생" 에서와 같이 새 유형에 자식이 포함되지 않은 simpleContent 요소 신호입니다. 이 예제에서는 제한 요소를 사용하여 새 형식에 사용되는 가능한 값을 제한합니다. 특성 요소는 새 형식에 요소를 추가합니다.The attribute element adds the element to the new type.

35.3.4. Java로의 매핑

간단한 유형에서 파생된 복잡한 유형은 @XmlType 주석으로 장식되는 Java 클래스에 매핑됩니다. 생성된 클래스에는 복잡한 형식이 파생되는 단순 형식의 멤버 변수 값이 포함됩니다.The generated class contains a member variable, value, of the simple type from which the complex type is derived. 멤버 변수는 @XmlValue 주석으로 장식됩니다. 클래스에도 getValue() 메서드와 setValue() 메서드가 있습니다. 또한 생성된 클래스에는 단순 유형을 확장하는 각 속성에 대해 멤버 변수와 연결된 getter 및 setter 메서드가 있습니다.

예 35.14. "idType Java Class" 는 예 35.13. "단순 유형에서 제한으로 복합 유형 파생" 에 정의된 idType 유형에 대해 생성된 Java 클래스를 보여줍니다.

예 35.14. idType Java Class

```
@XmlType(name = "idType", propOrder = {
    "value"
})
public class IdType {

    @XmlValue
    protected String value;
    @XmlAttribute
    @XmlSchemaType(name = "dateTime")
    protected XMLGregorianCalendar expires;
```

```
    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public XMLGregorianCalendar getExpires() {
        return expires;
    }

    public void setExpires(XMLGregorianCalendar value) {
        this.expires = value;
    }

}
```

35.4. COMPLEX TYPES에서 복합 유형 파생

35.4.1. 개요

XML 스키마를 사용하면 복잡한 요소를 사용하여 다른 복잡한 유형을 확장하거나 제한하여 새 복합 형식을 파생할 수 있습니다.**Using XML Schema, you can derive new complex types by either extending or restricting other complex types using the complexContent element.** 파생된 복잡한 유형을 나타내는 **Java** 클래스를 생성할 때 **Apache CXF**는 기본 유형의 클래스를 확장합니다. 이러한 방식으로 생성된 **Java** 코드는 **XML** 스키마에서 의도한 상속 계층을 유지합니다.

35.4.2. 스키마 구문

complexContent 요소 및 확장 요소 또는 제한 요소를 사용하여 다른 복잡한 형식에서 복잡한 형식을 파생시킵니다. **complexContent** 요소는 포함된 데이터 설명에 둘 이상의 필드가 포함되도록 지정합니다. 확장 요소 및 복합**Content** 요소의 자식인 제한 요소는 새 유형을 생성하도록 수정되는 기본 유형을 지정합니다. 기본 유형은 **base** 특성으로 지정됩니다.

35.4.3. 복잡한 유형 확장

복잡한 유형을 확장하려면 확장 요소를 사용하여 새 유형을 구성하는 추가 요소 및 특성을 정의합니다. 복잡한 유형 설명에서 허용되는 모든 요소는 새 유형의 정의의 일부로 허용될 수 있습니다. 예를 들어 새 형식에 익명 열거형을 추가하거나 **choice** 요소를 사용하여 새 필드 중 하나만 유효할 수 있습니다.**For example, you can add an anonymous enumeration to the new type, or you can use the choice element to specify that only one of the new fields can be valid at a time.**

예 35.15. “Extension으로 Complex 유형 파생” 두 가지 복잡한 유형, **widgetOrderInfo** 및 위젯 **OrderBillInfo** 를 정의하는 **XML** 스키마 조각을 보여줍니다. **widgetOrderBillInfo** 는 **widgetOrderInfo** 를

*확장하여 **orderNumber** 및 **amtDue** 의 두 가지 새로운 요소를 포함합니다.*

*예 35.15. **Extension**으로 **Complex** 유형 파생*

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element name="order_date" type="xsd:dateTime"/>
    <element name="type" type="xsd1:widgetSize"/>
    <element name="shippingAddress" type="xsd1:Address"/>
  </sequence>
  <attribute name="rush" type="xsd:boolean" use="optional" />
</complexType>
<complexType name="widgetOrderBillInfo">
  <complexContent>
    <extension base="xsd1:widgetOrderInfo">
      <sequence>
        <element name="amtDue" type="xsd:decimal"/>
        <element name="orderNumber" type="xsd:string"/>
      </sequence>
      <attribute name="paid" type="xsd:boolean"
                 default="false" />
    </extension>
  </complexContent>
</complexType>
```

35.4.4. 복잡한 유형 제한

*복잡한 유형을 제한하려면 **restriction** 요소를 사용하여 기본 유형의 요소 또는 속성의 가능한 값을 제한합니다. 복잡한 유형을 제한하는 경우 기본 형식의 모든 요소와 특성을 나열해야 합니다. 각 요소에 대해 정의에 제한적인 속성을 추가할 수 있습니다. 예를 들어 요소에 **maxOccurs** 특성을 추가하여 발생할 수 있는 횟수를 제한할 수 있습니다.**For example, you can add a maxOccurs attribute to an element to limit the number of times it can occur.** 고정 특성을 사용하여 요소 중 하나 이상이 미리 설정된 값을 갖도록 할 수도 있습니다.*

*예 35.16. "제한으로 복잡한 유형 정의" 다른 복잡한 유형을 제한하여 복잡한 유형을 정의하는 예를 보여줍니다. 제한된 유형인 **wallawallaAddress** 는 도시 요소, 상태 요소 및 **zipCode** 요소가 고정되기 때문에 워싱턴의 주소에만 사용할 수 있습니다.*

예 35.16. 제한으로 복잡한 유형 정의

```
<complexType name="Address">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="street" type="xsd:short" maxOccurs="3"/>
    <element name="city" type="xsd:string"/>
    <element name="state" type="xsd:string"/>
    <element name="zipCode" type="xsd:string"/>
```

```
  </sequence>
 </complexType>
 <complexType name="wallawallaAddress">
  <complexContent>
   <restriction base="xsd1:Address">
    <sequence>
     <element name="name" type="xsd:string"/>
     <element name="street" type="xsd:short"
            maxOccurs="3"/>
     <element name="city" type="xsd:string"
            fixed="WallaWalla"/>
     <element name="state" type="xsd:string"
            fixed="WA" />
     <element name="zipCode" type="xsd:string"
            fixed="99362" />
    </sequence>
   </restriction>
  </complexContent>
 </complexType>
```

35.4.5. Java로의 매핑

모든 복잡한 형식과 마찬가지로 **Apache CXF**는 다른 복잡한 유형에서 파생된 복잡한 유형을 나타내는 클래스를 생성합니다. 파생 복합 형식에 대해 생성된 **Java** 클래스는 기본 복잡한 유형을 지원하기 위해 생성된 **Java** 클래스를 확장합니다. 기본 **Java** 클래스도 **@XmlSeeAlso** 주석을 포함하도록 수정되었습니다. 기본 클래스의 **@XmlSee also** 주석 은 기본 클래스를 확장하는 모든 클래스를 나열합니다.

확장 프로그램에서 새 복합 형식을 파생하면 생성된 클래스에 추가된 모든 요소 및 특성에 대한 멤버 변수가 포함됩니다.**When the new complex type is derived by extension, the generated class will include member variables for all of the added elements and attributes.** 새 멤버 변수는 다른 모든 요소와 동일한 매핑에 따라 생성됩니다.

제한으로 새 복합 형식을 파생하면 생성된 클래스에 새 멤버 변수가 없습니다.**When the new complex type is derived by restriction, the generated class will have no new member variables.** 생성된 클래스는 단순히 추가 기능을 제공하지 않는 셸이 됩니다. **XML** 스키마에 정의된 제한이 적용되도록 전적으로 사용자에게 제공됩니다.**It is entirely up to you to ensure that the restrictions defined in the XML Schema are enforced.**

예를 들어 예 35.15. “Extension으로 Complex 유형 파생” 의 스키마는 두 개의 **Java** 클래스, 즉 **widget Ord erInfo** 및 위젯 **BillOrderInfo**가 생성됩니다. **widgetOrderBillInfo** 는 **widget Ord erInfo**에서 확장 가능하므로 **widgetOrderInfo** 를 확장합니다. 예 35.17. “WidgetOrderBillInfo” 위젯 **OrderBillInfo**에 대해 생성된 클래스를 표시합니다.

예 35.17. WidgetOrderBillInfo

```
@XmlType(name = "widgetOrderBillInfo", propOrder = {
```

```
    "amtDue",
    "orderNumber"
})
public class WidgetOrderBillInfo
    extends WidgetOrderInfo
{
    @XmlElement(required = true)
    protected BigDecimal amtDue;
    @XmlElement(required = true)
    protected String orderNumber;
    @XmlAttribute
    protected Boolean paid;

    public BigDecimal getAmtDue() {
        return amtDue;
    }

    public void setAmtDue(BigDecimal value) {
        this.amtDue = value;
    }

    public String getOrderNumber() {
        return orderNumber;
    }

    public void setOrderNumber(String value) {
        this.orderNumber = value;
    }

    public boolean isPaid() {
        if (paid == null) {
            return false;
        } else {
            return paid;
        }
    }

    public void setPaid(Boolean value) {
        this.paid = value;
    }
}
```

35.5. 발생 제한

35.5.1. Occurrence 제약 조건을 지원하는 스키마 요소

XML 스키마를 사용하면 복잡한 형식 정의를 구성하는 네 가지 XML 스키마 요소에서 발생 제약 조건을 지정할 수 있습니다.XML Schema allows you to specify the occurrence constraints on four of the XML Schema elements that make up a complex type definition:

35.5.2. 모든 요소의 발생 제약 조건

35.5.2.1. XML 스키마

모든 요소로 정의된 복잡한 형식은 모든 요소에서 정의한 구조의 여러 번 발생을 허용하지 않습니다. 그러나 **minOccurs** 특성을 **0** 으로 설정하여 모든 요소 선택 사항으로 정의된 구조를 만들 수 있습니다.

35.5.2.2. Java로의 매핑

모든 요소의 **minOccurs** 속성을 **0** 으로 설정하면 생성된 **Java** 클래스에는 영향을 미치지 않습니다.

35.5.3. 선택 요소에서의 발생 제약 조건

35.5.3.1. 개요

기본적으로 선택 요소 결과는 복잡한 형식의 인스턴스에 한 번만 나타날 수 있습니다.**By default, the results of a choice element can only appear once in an instance of a complex type.** 선택 요소에서 정의한 구조를 나타내기 위해 선택 한 요소가 **minOccurs** 속성 및 해당 **mxOccurs** 특성을 사용하여 표시할 수 있는 횟수를 변경할 수 있습니다. 이러한 특성을 사용하면 선택 형식이 복잡한 유형의 인스턴스에서 무제한 시간을 무제한으로 발생할 수 있도록 지정할 수 있습니다.**Using these attributes you can specify that the choice type can occur zero to an unlimited number of times in an instance of a complex type.** 선택 유형에 대해 선택한 요소는 유형마다 동일할 필요가 없습니다.

35.5.3.2. XML 스키마로 사용

minOccurs 속성은 선택 유형이 표시되어야 하는 최소 횟수를 지정합니다. 값은 모든 양의 정수가 될 수 있습니다. **minOccurs** 속성을 **0** 으로 설정하면 선택 유형이 복잡한 유형의 인스턴스 내에 표시되지 않아도 됩니다.

maxOccurs 속성은 선택 형식이 표시될 수 있는 최대 횟수를 지정합니다. 값은 0이 아닌, 양의 정수 또는 바인딩되지 않은 값일 수 있습니다. maxOccurs 속성을 unbounded 로 설정하면 선택 형식이 무한한 횟수가 표시될 수 있습니다.

예 35.18. "Occurrence 제약 조건 선택" 선택 유형인 ClubEvent 의 정의를 선택 발생 제약 조건으로 표시합니다. 선택 유형 전체적으로 0에서 바인딩되지 않은 시간을 반복할 수 있습니다.

예 35.18. Occurrence 제약 조건 선택

```
<complexType name="ClubEvent">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="MemberName" type="xsd:string"/>
    <element name="GuestName" type="xsd:string"/>
  </choice>
</complexType>
```

35.5.3.3. Java로의 매핑

단일 인스턴스 선택 구조와는 달리 여러 번 발생할 수 있는 XML 스키마 선택 구조는 단일 멤버 변수를 사용하여 Java 클래스에 매핑됩니다. 이 단일 멤버 변수는 시퀀스의 여러 발생에 대한 모든 데이터를 보유하는 List<T> 개체입니다.This single member variable is a List <T> object that holds all of the data for the multiple occurrences of the sequence. 예를 들어 예 35.18. "Occurrence 제약 조건 선택" 에 정의된 시퀀스가 두 번 발생한 경우 목록에 두 개의 항목이 있습니다.

Java 클래스의 멤버 변수의 이름은 멤버 요소의 이름을 연결하여 파생됩니다. 요소 이름은 Or 로 구분되며 변수 이름의 첫 번째 문자는 소문자로 변환됩니다. 예를 들어 예 35.18. "Occurrence 제약 조건 선택" 에서 생성된 멤버 변수의 이름은 memberNameOrGuestName 입니다.

목록에 저장된 개체 유형은 멤버 요소의 유형 간의 관계에 따라 달라집니다. 예를 들면 다음과 같습니다.

- *멤버 요소가 동일한 유형의 경우 생성된 목록에 JAXBElement<T > 개체가 포함됩니다. JAXBElement<T > 개체의 기본 유형은 멤버 요소 형식의 일반 매핑에 의해 결정됩니다.*
- *멤버 요소가 다른 형식이며 Java 표현이 공통 인터페이스를 구현하는 경우 목록에 공통 인터페이스의 개체가 포함됩니다.*
- *멤버 요소가 다른 형식이며 해당 Java 표현이 공통 기본 클래스를 확장하면 목록에 공통 기본 클래스의 개체가 포함됩니다.*

- 다른 조건이 충족되지 않으면 목록에 **Object** 오브젝트가 포함됩니다.

생성된 **Java** 클래스는 멤버 변수에 대한 **getter** 메서드만 갖습니다. **getter** 메서드는 라이브 목록에 대한 참조를 반환합니다. 반환된 목록에 대한 수정 사항은 실제 오브젝트에 적용됩니다.

Java 클래스는 **@XmlType** 주석으로 장식됩니다. 주석의 **name** 속성은 **XML Schema** 정의의 부모 요소에서 **name** 속성 값으로 설정됩니다. 주석의 **propOrder** 속성에는 시퀀스의 요소를 나타내는 단일 멤버 변수가 포함되어 있습니다.

선택한 구조의 요소를 나타내는 멤버 변수는 **@XmlElements** 주석으로 장식됩니다. **@XmlElements** 주석에는 쉼표로 구분된 **@XmlElement** 주석 목록이 포함되어 있습니다. 목록에는 유형의 **XML** 스키마 정의에 정의된 각 멤버 요소에 대해 하나의 **@XmlElement** 주석이 있습니다. 목록의 **@XmlElement** 주석에는 **XML Schema** 요소 요소 의 **name** 속성으로 설정된 **name** 속성이 있으며 **XML Schema** 요소 요소 유형 의 매핑에서 생성된 **Java** 클래스로 설정된 해당 **type** 속성이 있습니다.

예 35.19. “Occurrence Constraint를 사용한 Choice 구조의 Java 설명” 예 35.18. “Occurrence 제약 조건 선택” 에 정의된 **XML** 스키마 선택 구조의 **Java** 매핑을 보여줍니다.

예 35.19. Occurrence Constraint를 사용한 Choice 구조의 Java 설명

```
@XmlType(name = "ClubEvent", propOrder = {
    "memberNameOrGuestName"
})
public class ClubEvent {

    @XmlElementRefs({
        @XmlElementRef(name = "GuestName", type = JAXBElement.class),
        @XmlElementRef(name = "MemberName", type = JAXBElement.class)
    })
    protected List<JAXBElement<String>> memberNameOrGuestName;

    public List<JAXBElement<String>> getMemberNameOrGuestName() {
        if (memberNameOrGuestName == null) {
            memberNameOrGuestName = new ArrayList<JAXBElement<String>>();
        }
        return this.memberNameOrGuestName;
    }

}
```

35.5.3.4. minOccurs가 0으로 설정

minOccurs 요소만 지정되고 해당 값이 **0** 이면 **minOccurs** 속성이 설정되지 않은 것처럼 코드 생성기

가 Java 클래스를 생성합니다.

35.5.4. Elements에서의 발생 제약 조건

35.5.4.1. 개요

요소 요소의 minOccurs 속성 및 maxOccurs 속성을 사용하여 복잡한 요소에서 특정 요소가 표시되는 횟수를 지정할 수 있습니다.You can specify how many times a specific element in a complex type appears using the element's minOccurs attribute and maxOccurs attribute. 두 속성의 기본값은 1 입니다.

35.5.4.2. minOccurs가 0으로 설정

복잡한 유형의 멤버 요소 minOccurs 속성 중 하나를 0 으로 설정하면 해당 Java 멤버 변수를 구성하는 @XmlElement 주석이 변경됩니다. 필수 속성을 true 로 설정하는 대신 @XmlElement 주석의 필수 속성이 false 로 설정됩니다.

35.5.4.3. mindays가 1보다 큰 값으로 설정

XML 스키마에서는 요소 요소의 minOccurs 속성을 둘 이상의 값으로 설정하여 형식의 인스턴스에서 요소가 두 번 이상 발생하도록 지정할 수 있습니다.In XML Schema you can specify that an element must occur more than once in an instance of the type by setting the element's minOccurs attribute to a value greater than one. 그러나 생성된 Java 클래스는 XML 스키마 제약 조건을 지원하지 않습니다. Apache CXF는 minOccurs 속성이 설정되지 않은 것처럼 지원되는 Java 멤버 변수를 생성합니다.

35.5.4.4. maxOccurs set이 있는 요소

멤버 요소가 복잡한 형식의 인스턴스에 여러 번 표시되도록 하려면 요소의 maxOccurs 속성을 1보다 큰 값으로 설정합니다.When you want a member element to appear multiple times in an instance of a complex type, you set the element's maxOccurs attribute to a value greater than 1. maxOccurs 특성의 값을 unbounded 로 설정하여 멤버 요소가 무제한으로 표시될 수 있음을 지정할 수 있습니다.

코드 생성기는 maxOccurs 특성이 있는 멤버 요소를 값 1보다 큰 값으로 매핑합니다.The code generators map a member element with the maxOccurs attribute set to a value greater than 1 to a Java member variable. 목록의 기본 클래스는 요소의 유형을 Java에 매핑하여 결정됩니다. XML 스키마 기본 유형의 경우 래퍼 클래스는 "래퍼 클래스" 에 설명된 대로 사용됩니다. 예를 들어 멤버 요소가 xsd:int 형식의 경우 생성된 멤버 변수는 List<Integer> 오브젝트입니다.For example, if the member element is of type xsd:int the generated member variable is a List<Integer > object.

35.5.5. 시퀀스의 발생 제한

35.5.5.1. 개요

기본적으로 시퀀스 요소의 내용은 복잡한 형식의 인스턴스에 한 번만 나타날 수 있습니다.**By default, the contents of a sequence element can only appear once in an instance of a complex type.** 시퀀스 요소에서 정의한 요소 시퀀스가 **minOccurs** 속성 및 해당 **maxOccurs** 특성을 사용하여 표시할 수 있는 횟수를 변경할 수 있습니다.**You can change the number of times the sequence of elements defined by a sequence element is allowed to appear using its minOccurs attribute and its maxOccurs attribute.** 이러한 특성을 사용하면 시퀀스 유형이 복잡한 유형의 인스턴스에서 무제한 횟수로 0이 발생할 수 있습니다.**Using these attributes you can specify that the sequence type can occur zero to an unlimited number of times in an instance of a complex type.**

35.5.5.2. XML 스키마 사용

minOccurs 속성은 정의된 복합 유형의 인스턴스에서 시퀀스가 발생해야 하는 최소 횟수를 지정합니다. 값은 모든 양의 정수가 될 수 있습니다. **minOccurs** 특성을 **0** 으로 설정하면 시퀀스가 복잡한 형식의 인스턴스 내에 나타나지 않도록 지정합니다.**Setting the minOccurs attribute to 0 specifies that the sequence does not need to appear inside an instance of the complex type.**

maxOccurs 속성은 정의된 복합 유형의 인스턴스에서 시퀀스가 발생할 수 있는 횟수에 대한 상한을 지정합니다. 값은 **0**이 아닌, 양의 정수 또는 바인딩되지 않은 값일 수 있습니다. **maxOccurs** 속성을 **unbounded** 로 설정하면 시퀀스가 무한한 횟수로 표시될 수 있습니다.

예 35.20. “Occurrence Constraints이 있는 순서” 시퀀스 발생 제약 조건을 사용하여 시퀀스 형식 **CultureInfo**의 정의를 표시합니다.**Shows the definition of a sequence type, CultureInfo, with sequence occurrence constraints.** 시퀀스는 **0**에서 **2**번 반복할 수 있습니다.

예 35.20. Occurrence Constraints이 있는 순서

```
<complexType name="CultureInfo">
  <sequence minOccurs="0" maxOccurs="2">
    <element name="Name" type="string"/>
    <element name="Lcid" type="int"/>
  </sequence>
</complexType>
```

35.5.5.3. Java로의 매핑

단일 인스턴스 시퀀스와 달리 여러 번 발생할 수 있는 **XML** 스키마 시퀀스는 단일 멤버 변수를 사용하여 **Java** 클래스에 매핑됩니다.**Unlike single instance sequences, XML Schema sequences that can occur multiple times are mapped to a Java class with a single member variable.** 이 단일 멤버 변수는 시퀀스의 여러 발생에 대한 모든 데이터를 보유하는 **List<T>** 개체입니다.**This single member variable is a List <T> object that holds all of the data for the multiple occurrences of the sequence.** 예를 들어 예 35.20. “Occurrence Constraints이 있는 순서” 에 정의된 시퀀스가 두 번 발생한 경우 목록에 네 개의 항목이 있습니다.

Java 클래스의 멤버 변수의 이름은 멤버 요소의 이름을 연결하여 파생됩니다. 요소 이름은 및 로 구분되며 변수 이름의 첫 번째 문자는 소문자로 변환됩니다. 예를 들어 예 35.20. “Occurrence Constraints이 있는 순서” 에서 생성된 멤버 변수의 이름은 nameAndLcid 입니다.

목록에 저장된 개체 유형은 멤버 요소의 유형 간의 관계에 따라 달라집니다. 예를 들면 다음과 같습니다.

- 멤버 요소가 동일한 유형의 경우 생성된 목록에 JAXBElement<T> 개체가 포함됩니다. JAXBElement<T> 개체의 기본 유형은 멤버 요소 형식의 일반 매핑에 의해 결정됩니다.
- 멤버 요소가 다른 형식이며 Java 표현이 공통 인터페이스를 구현하는 경우 목록에 공통 인터페이스의 개체가 포함됩니다.
- 멤버 요소가 다른 형식이며 해당 Java 표현이 공통 기본 클래스를 확장하면 목록에 공통 기본 클래스의 개체가 포함됩니다.
- 다른 조건이 충족되지 않으면 목록에 Object 오브젝트가 포함됩니다.

생성된 Java 클래스에는 멤버 변수에 대한 getter 메서드만 있습니다. getter 메서드는 라이브 목록에 대한 참조를 반환합니다. 반환된 목록에 대한 수정은 실제 오브젝트에 영향을 미칩니다.

Java 클래스는 @XmlType 주석으로 장식됩니다. 주석의 name 속성은 XML Schema 정의의 부모 요소에서 name 속성 값으로 설정됩니다. 주석의 propOrder 속성에는 시퀀스의 요소를 나타내는 단일 멤버 변수가 포함되어 있습니다.

시퀀스의 요소를 나타내는 멤버 변수는 @XmlElements 주석으로 장식됩니다. @XmlElements 주석에는 쉼표로 구분된 @XmlElement 주석 목록이 포함되어 있습니다. 목록에는 유형의 XML 스키마 정의에 정의된 각 멤버 요소에 대해 하나의 @XmlElement 주석이 있습니다. 목록의 @XmlElement 주석에는 XML Schema 요소 요소 의 name 속성으로 설정된 name 속성이 있으며 XML Schema 요소 요소 유형의 매핑에서 생성된 Java 클래스로 설정된 해당 type 속성이 있습니다.

예 35.21. “Occurrence Constraint가 포함된 시퀀스를 나타냅니다.Represents an Sequence with an Occurrence Constraint” 예 35.20. “Occurrence Constraints이 있는 순서” 에 정의된 XML 스키마 시퀀스의 Java 매핑을 보여줍니다.

예 35.21. Occurrence Constraint가 포함된 시퀀스를 나타냅니다.Represents an Sequence with an Occurrence Constraint

```
@XmlType(name = "CultureInfo", propOrder = {
    "nameAndLcid"
})
public class CultureInfo {

    @XmlElements({
        @XmlElement(name = "Name", type = String.class),
        @XmlElement(name = "Lcid", type = Integer.class)
    })
    protected List<Serializable> nameAndLcid;

    public List<Serializable> getNameAndLcid() {
        if (nameAndLcid == null) {
            nameAndLcid = new ArrayList<Serializable>();
        }
        return this.nameAndLcid;
    }

}
```

35.5.5.4. minOccurs가 0으로 설정

minOccurs 요소만 지정되고 해당 값이 **0** 이면 **minOccurs** 속성이 설정되지 않은 것처럼 코드 생성기가 **Java** 클래스를 생성합니다.

35.6. 모델 그룹 사용

35.6.1. 개요

XML 스키마 모델 그룹은 사용자 정의 복합 형식에서 요소 그룹을 참조할 수 있는 편리한 바로 가기입니다.**For example, you can define a group of elements that are common to several types in your application and then reference the group again.** 모델 그룹은 그룹 요소를 사용하여 정의되며 복잡한 유형 정의와 유사합니다. 모델 그룹을 **Java**로 매핑하는 것도 복잡한 유형의 매핑과 유사합니다.

35.6.2. XML 스키마에서 모델 그룹 정의

name 속성이 있는 **group** 요소를 사용하여 **XML Schema**에서 모델 그룹 을 정의합니다. **name** 특성의 값은 스키마 전체에서 그룹을 참조하는 데 사용되는 문자열입니다. **complexType** 요소와 같은 **group** 요소에는 시퀀스 요소, 모든 요소 또는 **choice** 요소가 **immediate** 자식으로 포함될 수 있습니다.

자식 요소 내에서는 요소 요소를 사용하여 그룹의 멤버를 정의합니다.**Inside the child element, you define the members of the group using element elements.** 그룹의 각 멤버에 대해 하나의 요소 요소

*를 지정합니다. 그룹 멤버는 **minOccurs** 및 **maxOccurs** 를 포함하여 요소 요소에 대한 표준 속성을 사용할 수 있습니다.**Group members can use any of the standard attributes for the element element including minOccurs and maxOccurs.** 따라서 그룹에 세 개의 요소가 있고 그 중 하나가 최대 **3** 번 발생할 수 있는 경우 세 가지 요소 가 있는 그룹을 정의하면 **maxOccurs="3"**을 사용하는 그룹을 정의합니다. 예 35.22. "XML 스키마 모델 그룹" 세 가지 요소가 있는 모델 그룹을 보여 줍니다.*

예 35.22. XML 스키마 모델 그룹

```
<group name="passenger">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="clubNum" type="xsd:long" />
    <element name="seatPref" type="xsd:string"
          maxOccurs="3" />
  </sequence>
</group>
```

35.6.3. 형식 정의에서 모델 그룹 사용Using a model group in a type definition

*모델 그룹을 정의한 후에는 복잡한 형식 정의의 일부로 사용할 수 있습니다.**Once a model group has been defined, it can be used as part of a complex type definition.** 복잡한 유형 정의에서 모델 그룹을 사용하려면 **ref** 속성이 있는 **group** 요소를 사용합니다. **ref** 특성의 값은 해당 특성이 정의되었을 때 그룹에 지정된 이름입니다. 예를 들어 예 35.22. "XML 스키마 모델 그룹" 에 정의된 그룹을 사용하려면 예 35.23. "모델 그룹이 있는 복합 유형" 와 같이 **{ref="tns:passenger" >** 을 사용합니다.*

예 35.23. 모델 그룹이 있는 복합 유형

```
<complexType name="reservation">
  <sequence>
    <group ref="tns:passenger" />
    <element name="origin" type="xsd:string" />
    <element name="destination" type="xsd:string" />
    <element name="fltNum" type="xsd:long" />
  </sequence>
</complexType>
```

*형식 정의에서 모델 그룹을 사용하면 그룹이 형식의 멤버가 됩니다.**When a model group is used in a type definition, the group becomes a member of the type.** 따라서 예약 인스턴스에는 네 개의 멤버 요소가 있습니다. 첫 번째 요소는 **passenger** 요소이며 예 35.22. "XML 스키마 모델 그룹" 에 표시된 그룹에서 정의한 멤버 요소를 포함합니다. 예약 인스턴스 예는 예 35.24. "모델 그룹이 있는 **Type**의 인스턴스" 에 표시되어 있습니다.*

예 35.24. 모델 그룹이 있는 Type의 인스턴스

```
<reservation>
  <passenger> <name>A. Smart</name> <clubNum>99</clubNum> <seatPref>isle1</seatPref>
```

```
</passenger>
  <origin>LAX</origin>
  <destination>FRA</destination>
  <fltNum>34567</fltNum>
</reservation>
```

35.6.4. Java로의 매핑

기본적으로 모델 그룹은 복잡한 유형 정의에 포함된 경우에만 **Java** 아티팩트에 매핑됩니다. 모델 그룹을 포함하는 복잡한 유형의 코드를 생성할 때 **Apache CXF**는 모델 그룹의 멤버 변수를 해당 유형에 대해 생성된 **Java** 클래스에 간단하게 포함합니다. 모델 그룹을 나타내는 멤버 변수는 모델 그룹의 정의에 따라 주석이 추가됩니다.

예 35.25. “그룹이 포함된 유형” 는 예 35.23. “모델 그룹이 있는 복합 유형” 에 정의된 복잡한 유형에 대해 생성된 **Java** 클래스를 보여줍니다.

예 35.25. 그룹이 포함된 유형

```
@XmlType(name = "reservation", propOrder = {
    "name",
    "clubNum",
    "seatPref",
    "origin",
    "destination",
    "fltNum"
})
public class Reservation {

    @XmlElement(required = true)
    protected String name;
    protected long clubNum;
    @XmlElement(required = true)
    protected List<String> seatPref;
    @XmlElement(required = true)
    protected String origin;
    @XmlElement(required = true)
    protected String destination;
    protected long fltNum;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public long getClubNum() {
        return clubNum;
    }
```

```
public void setClubNum(long value) {
    this.clubNum = value;
}

public List<String> getSeatPref() {
    if (seatPref == null) {
        seatPref = new ArrayList<String>();
    }
    return this.seatPref;
}

public String getOrigin() {
    return origin;
}

public void setOrigin(String value) {
    this.origin = value;
}

public String getDestination() {
    return destination;
}

public void setDestination(String value) {
    this.destination = value;
}

public long getFltNum() {
    return fltNum;
}

public void setFltNum(long value) {
    this.fltNum = value;
}
```

35.6.5. 여러 발생

그룹 요소의 **maxOccurs** 속성을 둘 이상의 값으로 설정하여 모델 그룹이 두 번 이상 표시되도록 지정할 수 있습니다.**You can specify that the model group appears more than once by setting the group's maxOccurs attribute to a value greater than one.** 모델 그룹 **Apache CXF**의 여러 발생을 허용하기 위해 모델 그룹을 **List<T>** 오브젝트에 매핑 합니다. **List<T >** 오브젝트는 그룹의 첫 번째 하위 규칙에 따라 생성됩니다.

- 시퀀스 요소를 사용하여 그룹을 정의한 경우 **35.5.5**절. “시퀀스의 발생 제한” 을 참조하십시오.
- 선택 요소를 사용하여 그룹을 정의하는 경우 **35.5.3**절. “선택 요소에서의 발생 제약 조건” 을 참조하십시오.

36장. 와일드 카드 유형 사용

초록

스키마 작성자가 바인딩 요소 또는 특성을 정의된 형식으로 연기하려는 경우가 있습니다.**There are instances when a schema author want to defer binding elements or attributes to a defined type.** 이러한 경우 **XML** 스키마는 와일드카드 위치 소유자를 지정하기 위한 세 가지 메커니즘을 제공합니다. 이러한 모든 것은 **XML** 스키마 기능을 유지하는 방식으로 **Java**에 매핑됩니다.

36.1. 모든 요소 사용

36.1.1. 개요

XML Schema any 요소는 복잡한 유형 정의에서 와일드 카드 장소 홀더를 만드는 데 사용됩니다. **XML** 스키마에 대해 **XML** 요소를 인스턴스화하는 경우 모든 유효한 **XML** 요소가 될 수 있습니다.**When an XML element is instantiated for an XML Schema any element, it can be any valid XML element.** 모든 요소에는 인스턴스화된 **XML** 요소의 콘텐츠 또는 이름에 제한이 없습니다.

예를 들어 예 36.1. “XML 스키마 유형 : Any Element” 에 정의된 복잡한 유형이 제공되면 예 36.2. “모든 요소가 포함된 XML 문서” 에 표시된 **XML** 요소 중 하나를 인스턴스화할 수 있습니다.

예 36.1. XML 스키마 유형 : Any Element

```
<element name="FlyBoy">
  <complexType>
    <sequence>
      <any />
      <element name="rank" type="xsd:int" />
    </sequence>
  </complexType>
</element>
```

예 36.2. 모든 요소가 포함된 XML 문서

```
<FlyBoy>
  <learJet>CL-215</learJet>
  <rank>2</rank>
</element>
<FlyBoy>
  <viper>Mark II</viper>
  <rank>1</rank>
</element>
```

XML 스키마 모든 요소는 Java Object 개체 또는 Java org.w3c.dom.Element 개체에 매핑됩니다.

36.1.2. XML 스키마 지정

모든 요소는 시퀀스 복잡한 유형을 정의하고 복잡한 유형을 선택할 때 사용할 수 있습니다. 대부분의 경우 모든 요소는 빈 요소입니다. 그러나 주석 요소를 자식으로 사용할 수 있습니다.

표 36.1. "XML Schema Any Element의 특성" 모든 요소의 속성을 설명합니다.

표 36.1. XML Schema Any Element의 특성

속성	설명
namespace	XML 문서의 요소를 인스턴스화하는 데 사용할 수 있는 요소의 네임스페이스를 지정합니다.Specifies the namespace of the elements that can be used to instantiate the element in an XML document. 유효한 값은 다음과 같습니다. **##any** 네임스페이스의 요소를 사용할 수 있도록 지정합니다. 이는 기본값입니다. **##기타** **부모 요소의 네임스페이스 이외의 모든 네임스페이스의 요소** 를 사용할 수 있도록 지정합니다. **##local** 네임스페이스가 없는 요소를 사용해야 합니다. **##targetNamespace** 부모 요소의 네임스페이스의 요소를 사용할 수 있도록 지정합니다. **공백으로 구분된 URI 목록 #\#local 및 \#targetNamespace** 나열된 네임스페이스의 요소를 사용할 수 있도록 지정합니다.
maxOccurs	요소의 인스턴스가 부모 요소에 표시될 수 있는 최대 횟수를 지정합니다.Specifies the maximum number of times an instance of the element can appear in the parent element. 기본값은 **1** 입니다. 요소의 인스턴스가 무제한으로 표시될 수 있도록 지정하려면 특성 값을 **unbounded** 로 설정할 수 있습니다.
minOccurs	요소의 인스턴스가 부모 요소에 나타날 수 있는 최소 횟수를 지정합니다.Specifies the minimum number of times an instance of the element can appear in the parent element. 기본값은 **1** 입니다.

속성	설명
processContents	요소를 인스턴스화하는 데 사용되는 요소를 유효성 검사해야 하는 방법을 지정합니다.Specifies how the element used to instantiate the any element should be validated. 유효한 값은 다음과 같습니다. **strict** 적절한 스키마에 대해 요소의 유효성을 검사하도록 지정합니다.Specifies that the element must be validated against the proper schema. 이는 기본값입니다. **lax** 적절한 스키마에 대해 요소를 검증하도록 지정합니다. 유효성을 검사할 수 없는 경우 오류가 발생하지 않습니다. **skip** 요소를 유효성 검사를 수행할 수 없도록 지정합니다.Specifies that the element should not be validated.

예 36.3. "모든 요소를 사용하여 정의되는 복합 유형" 모든 요소를 사용하여 정의된 복잡한 형식을 보여줍니다.

예 36.3. 모든 요소를 사용하여 정의되는 복합 유형

```
<complexType name="surprisePackage">
  <sequence>
    <any processContents="lax" />
    <element name="to" type="xsd:string" />
    <element name="from" type="xsd:string" />
  </sequence>
</complexType>
```

36.1.3. Java로의 매핑

XML 스키마를 사용하면 모든 요소(element)라는 Java 속성이 생성됩니다. 속성에는 getter 및 setter 메서드가 연결되어 있습니다. 결과 속성의 유형은 요소의 processContents 특성 값에 따라 다릅니다. 모든 요소의 processContents 특성이 건너뛰도록 설정된 경우 해당 요소는 org.w3c.dom.Element 개체에 매핑됩니다.If the element's processContents attribute is set to skip, the element is mapped to a org.w3c.dom.Element object. processContents 특성의 다른 모든 값에 대해 모든 요소 가 Java Object 오브젝트에 매핑됩니다.

생성된 속성은 @XmlAnyElement 주석으로 장식됩니다. 이 주석에는 데이터를 마샬링할 때 런타임에 수행할 작업을 지시하는 선택적 lax 속성이 있습니다. 기본값은 false 이며 런타임에 자동으로 데이터를

org.w3c.dom.Element 개체로 마샬링하도록 지시합니다. **lax** 를 **true** 로 설정하면 런타임에 데이터를 **JAXB** 유형으로 마샬링하려고 합니다. 모든 요소의 **processContents** 특성이 건너뛰 도록 설정되면 **Lax** 속성이 기본값으로 설정됩니다. **processContents** 특성의 다른 모든 값의 경우 **lax** 는 **true** 로 설정됩니다.

예 36.4. "모든 요소가 있는 Java 클래스" 예 36.3. "모든 요소를 사용하여 정의되는 복합 유형" 에 정의된 복잡한 유형이 **Java** 클래스에 매핑되는 방법을 보여줍니다.

예 36.4. 모든 요소가 있는 **Java** 클래스

```
public class SurprisePackage {

    @XmlAnyElement(lax = true) protected Object any;
    @XmlElement(required = true)
    protected String to;
    @XmlElement(required = true)
    protected String from;

    public Object getAny() { return any; }

    public void setAny(Object value) { this.any = value; }

    public String getTo() {
        return to;
    }

    public void setTo(String value) {
        this.to = value;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String value) {
        this.from = value;
    }

}
```

36.1.4. 마샬링

모든 요소에 대한 **Java** 속성에 **La x** 가 **false** 로 설정되어 있거나 속성이 지정되지 않은 경우 런타임에서 **XML** 데이터를 **JAXB** 개체에 구문 분석하지 않습니다. 데이터는 항상 **DOM Element** 객체에 저장됩니다.

모든 요소에 대한 **Java** 속성에 **lax** 가 **true** 로 설정된 경우 런타임은 **XML** 데이터를 적절한 **JAXB** 개체에 마샬링하려고 합니다. 런타임은 다음 절차를 사용하여 적절한 **JAXB** 클래스를 식별하려고 시도합니다.

1. XML 요소의 요소 태그를 런타임에 알려진 요소 목록에 대해 확인합니다. 일치하는 항목이 있으면 런타임은 XML 데이터를 요소에 대한 적절한 JAXB 클래스로 마샬링합니다.

2. XML 요소의 xsi:type 속성을 확인합니다. 일치하는 항목을 찾으면 런타임에서 XML 요소를 해당 유형의 적절한 JAXB 클래스로 마샬링합니다.

3. 일치하는 항목을 찾을 수 없는 경우 XML 데이터를 DOM Element 개체로 마샬링합니다.

일반적으로 애플리케이션의 런타임은 스키마에서 생성된 모든 유형의 계약에 대해 알고 있습니다. 여기에는 계약의 wsdl:types 요소에 정의된 유형, 포함을 통해 계약에 추가된 모든 데이터 유형, 다른 스키마를 가져와서 계약에 추가된 모든 유형이 포함됩니다. 32.4절. "런타임 Marshaller에 클래스 추가" 에서 설명하는 @XmlSeeAlso 주석을 사용하여 런타임에서 추가 유형을 인식할 수도 있습니다.

36.1.5. unmarshalling

모든 요소에 대한 Java 속성이 false 로 설정되어 있거나 속성이 지정되지 않은 경우 런타임은 DOM Element 객체만 허용합니다. 다른 유형의 개체를 사용하려고 하면 마샬링 오류가 발생합니다.

모든 요소의 Java 속성에 lax 가 true 로 설정된 경우 런타임은 Java 데이터 형식과 XML Schema 구문 간의 내부 맵을 사용하여 유선에 쓸 XML 구조를 결정합니다. 런타임에서 클래스를 알고 XML 스키마 구문에 매핑할 수 있는 경우 데이터를 작성하고 xsi:type 속성을 삽입하여 요소에 포함된 데이터 유형을 식별합니다.If the runtime knows the class and can map it to an XML Schema construct, it writes out the data and inserts an xsi:type attribute to identify the type of data the element contains.

런타임에서 Java 개체를 알려진 XML 스키마 구성에 매핑할 수 없는 경우 마샬링 예외가 발생합니다. 32.4절. "런타임 Marshaller에 클래스 추가" 에서 설명하는 @XmlSeeAlso 주석을 사용하여 런타임 맵에 유형을 추가할 수 있습니다.

36.2. XML 스키마 ANYTYPE 유형 사용

36.2.1. 개요

XML Schema 유형 xsd:anyType 은 모든 XML 스키마 유형의 루트 유형입니다. 모든 프리미티브는 사용자 정의 복잡한 유형과 마찬가지로 이 유형의 파생물입니다. 결과적으로 xsd:anyType 으로 정의된 요소는 스키마 문서에 정의된 복잡한 유형뿐만 아니라 XML 스키마 기본 형식 형식으로 데이터를 포함할 수 있습니다.As a result, elements defined as being of xsd:anyType can contain data in the form of any of the XML Schema primitives as well as any complex type defined in a schema document.

Java에서 가장 가까운 유형은 Object 클래스입니다. 다른 모든 Java 클래스가 하위 형식의 클래스입니다.

36.2.2. XML 스키마로 사용

다른 XML 스키마 복잡한 형식과 마찬가지로 xsd:anyType 을 사용합니다. 요소 요소의 형식 값으로 사용할 수 있습니다.It can be used as the value of an element 's type element. 다른 유형이 정의된 기본 유형으로도 사용할 수 있습니다.

예 36.5. "와일드 카드 요소가 있는 복합 유형" xsd:anyType 형식의 요소를 포함하는 복잡한 형식의 예를 보여줍니다.

예 36.5. 와일드 카드 요소가 있는 복합 유형

```
<complexType name="wildStar">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="ship" type="xsd:anyType" />
  </sequence>
</complexType>
```

36.2.3. Java로의 매핑

xsd:anyType 유형의 요소는 Object 오브젝트에 매핑됩니다. 예 36.6. "와일드 카드 요소 설명 Java Representation of a Wild Card Element" 에서 예 36.5. "와일드 카드 요소가 있는 복합 유형" 의 매핑을 Java 클래스에 표시합니다.

예 36.6. 와일드 카드 요소 설명 Java Representation of a Wild Card Element

```
public class WildStar {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true) protected Object ship;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public Object getShip() { return ship; }
```

```
    public void setShip(Object value) { this.ship = value; }
}
```

이 매핑을 사용하면 와일드카드 요소를 나타내는 속성에 데이터를 배치할 수 있습니다. Apache CXF 런타임은 데이터의 마샬링 및 사용 가능한 Java 표현으로 데이터 마샬링을 처리합니다.

36.2.4. 마샬링

Apache CXF가 XML 데이터를 Java 형식으로 마샬링할 때 모든Type 요소를 알려진 JAXB 개체로 마샬링하려고 합니다. anyType 요소를 JAXB 생성된 개체로 마샬링할 수 있는지 확인하려면 런타임에서 요소의 xsi:type 속성을 검사하여 요소에서 데이터를 구성하는 데 사용되는 실제 유형을 결정합니다. xsi:type 속성이 없는 경우 런타임은 인트로스펙션을 통해 요소의 실제 데이터 유형을 식별합니다. 요소의 실제 데이터 형식이 애플리케이션의 JAXB 컨텍스트에서 알려진 유형 중 하나로 확인되면 요소가 적절한 유형의 JAXB 오브젝트로 마샬링됩니다.

런타임이 요소의 실제 데이터 형식을 결정할 수 없거나 요소의 실제 데이터 형식이 알려진 형식이 아닌 경우 런타임은 콘텐츠를 org.w3c.dom.Element 개체로 마샬링합니다.If the runtime cannot determine the actual data type of the element, or the actual data type of the element is not a known type, the runtime marshals the content into a org.w3c.dom.Element object. 그런 다음 DOM APIs를 사용하여 요소의 콘텐츠를 작업해야합니다.

애플리케이션 런타임은 일반적으로 계약 내에 포함된 스키마에서 생성된 모든 유형에 대해 알고 있습니다. 여기에는 계약의 wsdl:types 요소에 정의된 유형, 포함을 통해 계약에 추가된 모든 데이터 유형, 다른 스키마 문서를 가져와서 계약에 추가된 모든 유형이 포함됩니다. 32.4절. “런타임 Marshaller에 클래스 추가” 에서 설명하는 @XmlSeeAlso 주석을 사용하여 런타임에서 추가 유형을 인식할 수도 있습니다.

36.2.5. unmarshalling

Apache CXF가 Java 유형을 XML 데이터로 분리할 때 Java 데이터 유형과 XML 스키마 구문 간의 내부 맵을 사용하여 유선에 쓸 XML 구조를 결정합니다. 런타임에서 클래스를 알고 클래스를 XML 스키마 구문에 매핑할 수 있는 경우 데이터를 작성하고 xsi:type 속성을 삽입하여 요소에 포함된 데이터 유형을 식별합니다.If the runtime knows the class and can map the class to an XML Schema construct, it writes out the data and inserts an xsi:type attribute to identify the type of data the element contains. 데이터가 org.w3c.dom.Element 개체에 저장된 경우 런타임은 개체에서 나타내는 XML 구조를 기록하지만 xsi:type 속성은 포함하지 않습니다.

런타임에서 Java 개체를 알려진 XML 스키마 구문에 매핑할 수 없는 경우 마샬링 예외가 발생합니다. 32.4절. “런타임 Marshaller에 클래스 추가” 에서 설명하는 @XmlSeeAlso 주석을 사용하여 런타임 맵에 유형을 추가할 수 있습니다.

36.3. 바인딩되지 않은 특성 사용

36.3.1. 개요

XML 스키마에는 복잡한 형식 정의에서 임의의 속성에 대한 자리 표시자를 남겨 둘 수 있는 메커니즘이 있습니다. 이 메커니즘을 사용하여 모든 특성이 있을 수 있는 복잡한 유형을 정의할 수 있습니다. 예를 들어 세 가지 속성을 지정하지 않고 **<robot name="epsilon" />**, **<robot age="10000" />** 또는 **<robot type="weevil" />** 요소를 정의하는 유형을 생성할 수 있습니다. 이는 데이터의 유연성이 필요할 때 특히 유용할 수 있습니다.

36.3.2. XML 스키마로 정의

선언되지 않은 특성은 **anyAttribute** 요소를 사용하여 **XML** 스키마에서 정의됩니다. 특성 요소를 사용할 수 있는 위치에서 사용할 수 있습니다.**It can be used wherever an attribute element can be used.** **anyAttribute** 요소에는 예 36.7. “Undeclared Attribute를 사용하는 복합 유형” 과 같이 속성이 없습니다.

예 36.7. Undeclared Attribute를 사용하는 복합 유형

```
<complexType name="arbitter">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="rate" type="xsd:float" />
  </sequence>
  <anyAttribute />
</complexType>
```

정의된 유형인 **arbitter** 에는 두 개의 요소가 있으며 모든 유형의 하나의 특성을 가질 수 있습니다. 예 36.8. “와일드 카드 특성을 사용하여 정의된 요소의 예” 에 표시된 세 가지 요소는 모두 복잡한 유형 중재에서 생성될 수 있습니다.

예 36.8. 와일드 카드 특성을 사용하여 정의된 요소의 예

```
<officer rank="12"><name>...</name><rate>...</rate></officer>
<lawyer type="divorce"><name>...</name><rate>...</rate></lawyer>
<judge><name>...</name><rate>...</rate></judge>
```

36.3.3. Java로의 매핑

anyAttribute 요소를 포함하는 복잡한 형식이 **Java**에 매핑되면 코드 생성기는 **otherAttributes** 라는 멤버를 생성된 클래스에 추가합니다. **otherAttributes** 는 **java.util.Map<QName, String >** 유형이며 맵의 라이브 인스턴스를 반환하는 **getter** 메서드가 있습니다. **getter**에서 반환된 맵이 실시간이므로 맵에 대한 수정 사항이 자동으로 적용됩니다. 예 36.9. “Undeclared Attribute를 사용하여 Complex Type의 클래

스" 예 36.7. "Undeclared Attribute를 사용하는 복합 유형" 에 정의된 복잡한 유형에 대해 생성된 클래스를 보여줍니다.

예 36.9. Undeclared Attribute를 사용하여 Complex Type의 클래스

```
public class Arbitter {

    @XmlElement(required = true)
    protected String name;
    protected float rate;

    @XmlAnyAttribute private Map<QName, String> otherAttributes = new HashMap<QName,
String>();

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public float getRate() {
        return rate;
    }

    public void setRate(float value) {
        this.rate = value;
    }

    public Map<QName, String> getOtherAttributes() { return otherAttributes; }

}
```

36.3.4. 선언되지 않은 속성으로 작업

생성된 클래스의 **otherAttributes** 멤버는 **Map** 오브젝트로 채워집니다. 맵은 **QNames** 를 사용하여 키가 지정됩니다. 맵이 표시되면 오브젝트에 설정된 모든 속성에 액세스하고 오브젝트에 대한 새 특성을 설정할 수 있습니다.

예 36.10. "Undeclared Attributes로 작업" 선언되지 않은 특성을 사용하기 위한 샘플 코드를 보여줍니다.

예 36.10. Undeclared Attributes로 작업

```
Arbitter judge = new Arbitter();
Map<QName, String> otherAtts = judge.getOtherAttributes();
```

```
QName at1 = new QName("test.apache.org", "house");
QName at2 = new QName("test.apache.org", "veteran");

otherAtts.put(at1, "Cape");
otherAtts.put(at2, "false");

String vetStatus = otherAtts.get(at2);
```

예 36.10. "Undeclared Attributes로 작업" 의 코드는 다음을 수행합니다.

선언되지 않은 속성이 포함된 맵을 가져옵니다.

*특성을 사용할 **QNames**를 생성합니다.*

속성 값을 맵으로 설정합니다.

특성 중 하나에 대한 값을 검색합니다.

37장. ELEMENT SUBSTITUTION

초록

XML 스키마 대체 그룹을 사용하면 최상위 수준 또는 헤드 요소를 대체할 수 있는 요소 그룹을 정의할 수 있습니다.**XML Schema substitution groups allow you to define a group of elements that can replace a top level, or head, element.** 이는 공통 기본 형식을 공유하거나 교환이 가능해야 하는 요소를 공유하는 여러 요소가 있는 경우에 유용합니다.

37.1. XML 스키마의 그룹 대체

37.1.1. 개요

대체 그룹은 해당 스키마에서 생성된 문서에서 다른 요소를 교체할 수 있는 요소를 지정할 수 있는 **XML** 스키마의 기능입니다. **replaceable** 요소는 **head** 요소라고 하며 스키마의 전역 범위에 정의해야 합니다. 대체 그룹의 요소는 **head** 요소 또는 **head** 요소의 형식에서 파생되는 형식과 동일한 형식이어야 합니다.

대체 그룹을 사용하면 제네릭 요소를 사용하여 지정할 수 있는 요소 컬렉션을 빌드할 수 있습니다.**A substitution group allows you to build a collection of elements that can be specified using a generic element.** 예를 들어 **3**가지 유형의 위젯을 판매하는 회사의 주문 시스템을 구축하는 경우 세 가지 위젯 유형에 대한 공통 데이터 세트가 포함된 일반 위젯 요소를 정의할 수 있습니다. 그런 다음 각 위젯 유형에 대한 보다 구체적인 데이터 세트가 포함된 대체 그룹을 정의할 수 있습니다. 그러면 각 위젯 유형에 대한 특정 순서 작업을 정의하는 대신 일반 위젯 요소를 메시지 파트로 지정할 수 있습니다. 실제 메시지가 작성되면 메시지에 대체 그룹의 요소가 포함될 수 있습니다.

37.1.2. 구문

대체 그룹은 **XML Schema** 요소 요소 의 **substitutionGroup** 특성을 사용하여 정의됩니다. **substitutionGroup** 특성의 값은 정의되는 요소가 대체되는 요소의 이름입니다. 예를 들어, 헤드 요소 가 위젯 이면, 목재**Group="widget"**이라는 요소에 특성 **substitutionGroup=" widget "**을 추가 하면 위젯 요소가 사용되는 모든 곳에서는 목차를 대체할 수 있습니다. 이는 예 37.1. “대체 그룹 사용” 에 표시됩니다.

예 37.1. 대체 그룹 사용

```
<element name="widget" type="xsd:string" />
<element name="woodWidget" type="xsd:string"
        substitutionGroup="widget" />
```

37.1.3. 유형 제한 사항

대체 그룹의 요소는 **head** 요소 또는 **head** 요소의 형식에서 파생된 형식의 형식과 동일해야 합니다. 예를 들어 **head** 요소가 **xsd:int**인 경우 대체 그룹의 모든 멤버가 **xsd:int** 형식 또는 **xsd:int**에서 파생되는 형식이어야 합니다.**For example, if the head element is of type xsd:int all members of the substitution group must be of type xsd:int or of a type derived from xsd:int.** 대체 그룹의 요소가 **head** 요소의 유형에서 파생되는 유형인 예 37.2. “Complex Types을 사용하는 대체 그룹” 에 표시된 대체 그룹을 정의할 수도 있습니다.

예 37.2. Complex Types을 사용하는 대체 그룹

```
<complexType name="widgetType">
  <sequence>
    <element name="shape" type="xsd:string" />
    <element name="color" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="woodWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="woodType" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="plasticWidgetType">
  <complexContent>
    <extension base="widgetType">
      <sequence>
        <element name="moldProcess" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<element name="widget" type="widgetType" />
<element name="woodWidget" type="woodWidgetType"
         substitutionGroup="widget" />
<element name="plasticWidget" type="plasticWidgetType"
         substitutionGroup="widget" />
<complexType name="partType">
  <sequence>
    <element ref="widget" />
  </sequence>
</complexType>
<element name="part" type="partType" />
```

대체 그룹인 위젯 의 헤드 요소는 **widget Type** 유형으로 정의됩니다. 대체 그룹의 각 요소는 **widgetType** 을 확장하여 해당 유형의 위젯 주문과 관련된 데이터를 포함합니다.

예 37.2. “Complex Types을 사용하는 대체 그룹” 의 스키마에 따라 예 37.3. “대체 그룹을 사용하는 XML 문서” 의 파트 요소는 유효합니다.

예 37.3. 대체 그룹을 사용하는 XML 문서

```
<part>
  <widget>
    <shape>round</shape>
    <color>blue</color>
  </widget>
</part>
<part>
  <plasticWidget>
    <shape>round</shape>
    <color>blue</color>
    <moldProcess>sandCast</moldProcess>
  </plasticWidget>
</part>
<part>
  <woodWidget>
    <shape>round</shape>
    <color>blue</color>
    <woodType>elm</woodType>
  </woodWidget>
</part>
```

37.1.4. 추상 헤드 요소

스키마를 사용하여 생성된 문서에 절대 나타날 수 없는 추상 헤드 요소를 정의할 수 있습니다. 추상 헤드 요소는 제네릭 클래스의 보다 구체적인 구현을 정의하기 위한 기반으로 사용되기 때문에 Java의 추상 클래스와 유사합니다. 추상 헤드는 최종 제품에서 일반 요소를 사용하지 않도록 합니다.

예 37.4. “추상 헤드 정의” 와 같이 요소 요소의 **abstract** 속성을 **true** 로 설정하여 추상 헤드 요소 를 선언합니다. 이 스키마를 사용하면 유효한 **review** 요소에 **positiveComment** 요소가 포함될 수 있지만 주석 요소가 포함될 수 없습니다.

예 37.4. 추상 헤드 정의

```
<element name="comment" type="xsd:string" abstract="true" />
<element name="positiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="negtiveComment" type="xsd:string"
         substitutionGroup="comment" />
<element name="review">
  <complexContent>
    <all>
      <element name="custName" type="xsd:string" />
      <element name="impression" ref="comment" />
```

```
    </all>
  </complexContent>
</element>
```

37.2. JAVA에서 그룹 대체

37.2.1. 개요

JAXB 사양에 지정된 대로 Apache CXF는 와일드카드 정의에 대한 JAXBElement 클래스 지원 기능과 함께 Java의 네이티브 클래스 계층 구조를 사용하는 대체 그룹을 지원합니다. 대체 그룹의 멤버가 모두 공통 기본 형식을 공유해야 하므로 요소의 유형을 지원하기 위해 생성된 클래스도 공통 기본 유형을 공유합니다. 또한 Apache CXF는 헤드 요소의 인스턴스를 JAXBElement<? 확장 T > 속성에 매핑합니다.

37.2.2. 생성된 개체 팩토리 방법

대체 그룹을 포함하는 패키지를 지원하기 위해 생성된 개체 팩토리에는 대체 그룹의 각 요소에 대한 메서드가 있습니다. head 요소를 제외하고 대체 그룹의 각 멤버마다 표 37.1. “JAXB 요소 선언을 위한 속성은 Substitution 그룹의 멤버입니다.” 에 설명된 대로 오브젝트 팩토리 메서드를 장식하는 @XmlElementDecl 주석에는 두 가지 추가 속성이 포함됩니다.

표 37.1. JAXB 요소 선언을 위한 속성은 Substitution 그룹의 멤버입니다.

속성	설명
substitutionHeadNamespace	head 요소가 정의된 네임스페이스를 지정합니다.
substitutionHeadName	head 요소의 **name** 특성의 값을 지정합니다.

대체 그룹의 @XmlElementDecl 대체 요소의 헤드 요소에 대한 개체 팩토리 메서드에는 기본 네임스페이스 속성 및 기본 이름 속성만 포함됩니다.

요소 인스턴스화 방법 외에도 개체 팩토리에는 head 요소를 나타내는 개체를 인스턴스화하는 메서드가 포함됩니다.In addition to the element instantiation methods, the object factory contains a method for instantiating an object representing the head element. 대체 그룹의 멤버가 모두 복잡한 형식인 경우 개체 팩토리에는 사용되는 각 복잡한 유형의 인스턴스를 인스턴스화하는 방법도 포함됩니다.

예 37.5. “Substitution 그룹에 대한 Object Factory Method” 예 37.2. “Complex Types을 사용하는 대체 그룹” 에 정의된 대체 그룹의 오브젝트 팩토리 메서드를 보여줍니다.

예 37.5. Substitution 그룹에 대한 Object Factory Method

```
public class ObjectFactory {

    private final static QName _Widget_QNAME = new QName(...);
    private final static QName _PlasticWidget_QNAME = new QName(...);
    private final static QName _WoodWidget_QNAME = new QName(...);

    public ObjectFactory() {
    }

    public WidgetType createWidgetType() {
        return new WidgetType();
    }

    public PlasticWidgetType createPlasticWidgetType() {
        return new PlasticWidgetType();
    }

    public WoodWidgetType createWoodWidgetType() {
        return new WoodWidgetType();
    }

    @XmlElementDecl(namespace="...", name = "widget")
    public JAXBElement<WidgetType> createWidget(WidgetType value) {
        return new JAXBElement<WidgetType>(_Widget_QNAME, WidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "plasticWidget", substitutionHeadNamespace =
"...", substitutionHeadName = "widget")
    public JAXBElement<PlasticWidgetType> createPlasticWidget(PlasticWidgetType value) {
        return new JAXBElement<PlasticWidgetType>(_PlasticWidget_QNAME,
PlasticWidgetType.class, null, value);
    }

    @XmlElementDecl(namespace = "...", name = "woodWidget", substitutionHeadNamespace =
"...", substitutionHeadName = "widget")
    public JAXBElement<WoodWidgetType> createWoodWidget(WoodWidgetType value) {
        return new JAXBElement<WoodWidgetType>(_WoodWidget_QNAME,
WoodWidgetType.class, null, value);
    }

}
```

37.2.3. 인터페이스의 대체 그룹

*대체 그룹의 헤드 요소가 작업 메시지 중 하나에서 메시지 파트로 사용되는 경우 결과 메서드 매개 변수는 해당 요소를 지원하기 위해 생성된 클래스의 객체가 됩니다. **JAXBElement<? extends T>** 클래스의 인스턴스일 필요는 없습니다. 런타임은 **Java**의 네이티브 유형 계층 구조를 사용하여 유형 대체를 지원하고 **Java**는 지원되지 않는 유형을 사용하려고 합니다.*

*런타임에서 요소 대체를 지원하는 데 필요한 모든 클래스를 알고 있도록 **SEI**는 **@XmlSee also** 주석으로 장식됩니다. 이 주석은 마샬링에 런타임에 필요한 클래스 목록을 지정합니다. **@XmlSeeAlso** 주석을*

사용하는 방법에 대한 자세한 내용은 32.4절. "런타임 Marshaller에 클래스 추가" 을 참조하십시오.

예 37.7. "Substitution 그룹을 사용하여 생성된 인터페이스" 는 예 37.6. "Substitution 그룹을 사용하는 WSDL 인터페이스" 에 표시된 인터페이스에 대해 생성된 SEI를 표시합니다. 이 인터페이스에서는 예 37.2. "Complex Types을 사용하는 대체 그룹" 에 정의된 대체 그룹을 사용합니다.

예 37.6. Substitution 그룹을 사용하는 WSDL 인터페이스

```
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order" />
    <output message="tns:widgetOrderBill" name="bill" />
    <fault message="tns:badSize" name="sizeFault" />
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

예 37.7. Substitution 그룹을 사용하여 생성된 인터페이스

```
@WebService(targetNamespace = "...", name = "orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

  @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
  @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
  @WebMethod
  public int checkWidgets(
    @WebParam(partName = "widgetPart", name = "widget", targetNamespace = "...")
    com.widgetvendor.types.widgettypes.WidgetType widgetPart
  );
}
```

예 37.7. "Substitution 그룹을 사용하여 생성된 인터페이스" 에 표시된 SEI는 @XmlSeeAlso 주석의 오브젝트 팩토리를 나열합니다. 네임스페이스의 개체 팩토리를 나열하면 해당 네임스페이스에 대해 생성된 모든 클래스에 액세스할 수 있습니다.

37.2.4. 복잡한 유형의 대체 그룹

대체 그룹의 헤드 요소가 복잡한 형식의 요소로 사용될 때 코드 생성기는 요소를 **JAXBElement<? extends T>** 속성에 매핑합니다.**When the head element of a substitution group is used as an element in a complex type, the code generator maps the element to a JAXBElement<? extends T > property.** 대체 그룹을 지원하기 위해 생성된 생성된 클래스의 인스턴스가 포함된 속성에 매핑되지 않습니다.**It does not map it to a property containing an instance of the generated class generated to support the substitution group.**

예를 들어 예 37.8. “대체 그룹을 사용하는 복잡한 유형” 에 정의된 복잡한 유형은 예 37.9. “대체 그룹을 사용하여 복잡한 유형의 **Java** 클래스” 로 표시되는 **Java** 클래스가 생성됩니다. 복잡한 유형에서는 예 37.2. “**Complex Types**을 사용하는 대체 그룹” 에 정의된 대체 그룹을 사용합니다.

예 37.8. 대체 그룹을 사용하는 복잡한 유형

```
<complexType name="widgetOrderInfo">
  <sequence>
    <element name="amount" type="xsd:int"/>
    <element ref="xsd1:widget"/>
  </sequence>
</complexType>
```

예 37.9. 대체 그룹을 사용하여 복잡한 유형의 Java 클래스

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "widgetOrderInfo", propOrder = {"amount","widget",})
public class WidgetOrderInfo {

    protected int amount;
    @XmlElementRef(name = "widget", namespace = "...", type = JAXBElement.class) protected
JAXBElement<? extends WidgetType> widget;
    public int getAmount() {
        return amount;
    }

    public void setAmount(int value) {
        this.amount = value;
    }

    public JAXBElement<? extends WidgetType> getWidget() { return widget; }

    public void setWidget(JAXBElement<? extends WidgetType> value) { this.widget =
((JAXBElement<? extends WidgetType> ) value); }

}
```

37.2.5. 대체 그룹 속성 설정

대체 그룹으로 작업하는 방법은 코드 생성기가 그룹을 직각 **Java** 클래스 또는 **JAXBElement<? extends T >** 클래스에 매핑했는지에 따라 달라집니다. 요소가 생성된 값 클래스의 개체에 간단히 매핑되면 형식 계층 구조의 일부인 다른 **Java** 객체와 동일한 방식으로 오브젝트로 작업합니다. 부모 클래스에 대한 하위 클래스 중 하나를 대체할 수 있습니다. 개체를 검사하여 정확한 클래스를 확인하고 적절하게 캐스팅할 수 있습니다.

JAXB 사양은 생성된 클래스의 오브젝트를 인스턴스화하는 데 오브젝트 팩토리 메서드를 사용하는 것이 좋습니다.

코드 생성기에서 **JAXBElement<? 확장 T>** 개체를 확장하여 대체 그룹의 인스턴스를 보관하도록 확장하는 경우 요소의 값을 **JAXBElement<? extends T >** 개체에 래핑해야 합니다.**When the code generators create a JAXBElement<? extends T> object to hold instances of a substitution group, you must wrap the element's value in a JAXBElement<? extends T > object.** 이를 수행하는 가장 좋은 방법은 개체 팩토리에서 제공하는 요소 생성 방법을 사용하는 것입니다. 해당 값을 기반으로 요소를 만들기 위한 쉬운 방법을 제공합니다.

예 37.10. “하위 그룹의 멤버 설정” 대체 그룹의 인스턴스를 설정하기 위한 코드를 보여줍니다.

예 37.10. 하위 그룹의 멤버 설정

```
ObjectFactory of = new ObjectFactory();
PlasticWidgetType pWidget = of.createPlasticWidgetType();
pWidget.setShape = "round';
pWidget.setColor = "green";
pWidget.setMoldProcess = "injection";

JAXBElement<PlasticWidgetType> widget = of.createPlasticWidget(pWidget);

WidgetOrderInfo order = of.createWidgetOrderInfo();
order.setWidget(widget);
```

예 37.10. “하위 그룹의 멤버 설정” 의 코드는 다음을 수행합니다.

개체 팩토리를 인스턴스화합니다.

plastic WidgetType 개체를 인스턴스화 합니다.

JAXBElement<PlasticWidgetType > 개체를 인스턴스화하여 플라스틱 위젯 요소를 보유합니다.

위젯 OrderInfo 오브젝트를 인스턴스화 합니다.

위젯 OrderInfo 개체의 위젯을 plastic widget 요소를 포함하는 JAXBElement 개체로 설정합니다.

37.2.6. 대체 그룹 속성의 값 가져오기

개체 팩토리 메서드는 요소의 값을 JAXBElement<? 확장 T > 개체에서 추출할 때 도움이 되지 않습니다. JAXBElement<? 확장 T >의 getValue() 메서드를 사용해야 합니다. 다음 옵션은 getValue() 메서드에서 반환된 오브젝트 유형을 결정합니다.

- 모든 가능한 클래스의 isInstance() 메서드를 사용하여 요소의 value 개체의 클래스를 결정합니다.

- JAXBElement<? extends T > 개체의 getName() 메서드를 사용하여 요소의 이름을 결정합니다.

  getName() 메서드는 QName을 반환합니다. 요소의 로컬 이름을 사용하여 값 개체에 대한 적절한 클래스를 확인할 수 있습니다.

- JAXBElement<? extends T > 개체의 getDeclaredType() 메서드를 사용하여 value 개체의 클래스를 확인합니다.

  getDeclaredType() 메서드는 요소의 value 개체의 Class 오브젝트를 반환합니다.

주의

getDeclaredType() 메서드가 value 개체의 실제 클래스에 관계없이 head 요소의 기본 클래스를 반환할 가능성이 있습니다.

예 37.11. “Substitution 그룹의 멤버 값 가져오기Get the value of a Member of the Substitution Group” 대체 그룹에서 값을 검색하는 코드를 표시합니다. 요소 값 개체의 적절한 클래스를 결정하려면 예제에서는 요소의 getName() 메서드를 사용합니다.

예 37.11. Substitution 그룹의 멤버 값 가져오기Get the value of a Member of the Substitution Group

```
String elementName = order.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget")
{
  WoodWidgetType widget=order.getWidget().getValue();
}
else if (elementName.equals("plasticWidget")
{
  PlasticWidgetType widget=order.getWidget().getValue();
}
else
{
  WidgetType widget=order.getWidget().getValue();
}
```

37.3. 위젯 벤더 예

37.3.1. 위젯 명령 인터페이스

이 섹션에서는 **Apache CXF**에서 실제 애플리케이션을 해결하기 위해 사용되는 대체 그룹의 예를 보여줍니다. 서비스 및 소비자는 **예 37.2. "Complex Types을 사용하는 대체 그룹"** 에 정의된 위젯 대체 그룹을 사용하여 개발됩니다. 이 서비스는 **checkWidgets** 및 **placeWidgetOrder** 라는 두 가지 작업을 제공합니다. **예 37.12. "위젯 명령 인터페이스"** 주문 서비스의 인터페이스를 표시합니다.

예 37.12. 위젯 명령 인터페이스

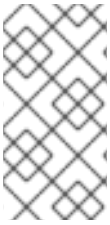
```
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation"
        type="xsd1:widgetOrderBillInfo"/>
</message>
<message name="widgetMessage">
  <part name="widgetPart" element="xsd1:widget" />
</message>
<message name="numWidgets">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
  <operation name="checkWidgets">
    <input message="tns:widgetMessage" name="request" />
```

```
    <output message="tns:numWidgets" name="response" />
  </operation>
</portType>
```

예 37.13. "위젯 주문 SEI" 에는 인터페이스에 대해 생성된 **Java SEI**가 표시됩니다.

예 37.13. 위젯 주문 SEI

```
@WebService(targetNamespace = "http://widgetVendor.com/widgetOrderForm", name =
"orderWidgets")
@XmlSeeAlso({com.widgetvendor.types.widgettypes.ObjectFactory.class})
public interface OrderWidgets {

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "numInventory", targetNamespace = "", partName = "numInventory")
    @WebMethod
    public int checkWidgets(
        @WebParam(partName = "widgetPart", name = "widget", targetNamespace =
"http://widgetVendor.com/types/widgetTypes")
        com.widgetvendor.types.widgettypes.WidgetType widgetPart
    );

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "widgetOrderConformation", targetNamespace = "", partName =
"widgetOrderConformation")
    @WebMethod
    public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo placeWidgetOrder(
        @WebParam(partName = "widgetOrderForm", name = "widgetOrderForm",
targetNamespace = "")
        com.widgetvendor.types.widgettypes.WidgetOrderInfo widgetOrderForm
    ) throws BadSize;
}
```



참고

예제에서는 대체 그룹의 사용만 보여 주므로 일부 비즈니스 논리는 표시되지 않습니다.

37.3.2. CheckWidgets 작업

37.3.2.1. 개요

CheckWidgets 는 대체 그룹의 헤드 멤버인 매개 변수가 있는 간단한 작업입니다. 이 작업에서는 대체 그룹의 멤버인 개별 매개 변수를 처리하는 방법을 보여줍니다. 소비자는 매개변수가 대체 그룹의 유효

한 멤버인지 확인해야 합니다. 서비스에서는 요청에 전송된 대체 그룹의 멤버를 올바르게 결정해야 합니다.

37.3.2.2. 소비자 구현

생성된 메서드 서명은 대체 그룹의 **head** 요소 유형을 지원하는 **Java** 클래스를 사용합니다. 대체 그룹의 멤버 요소는 **head** 요소 또는 **head** 요소의 유형에서 파생 된 형식의 동일한 유형 중 하나이므로 대체 그룹의 멤버를 지원하기 위해 생성된 **Java** 클래스는 **head** 요소를 지원하기 위해 생성된 **Java** 클래스에서 상속됩니다. **Java**의 유형 계층 구조는 기본적으로 상위 클래스 대신 하위 클래스를 사용할 수 있도록 지원합니다.

Apache CXF가 대체 그룹 및 **Java** 유형 계층 구조에 대한 유형을 생성하는 방법 때문에 클라이언트는 특수 코드를 사용하지 않고 **checkWidgets()** 를 호출할 수 있습니다. **checkWidgets()** 를 호출할 논리를 개발할 때 위젯 대체 그룹을 지원하기 위해 생성된 클래스 중 하나의 개체로 전달할 수 있습니다.

예 37.14. “consumer Invoking checkWidgets()” **checkWidgets()** 를 호출하는 소비자를 표시합니다.

예 37.14. consumer Invoking checkWidgets()

```
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
    WidgetType widget = new WidgetType();
    ...
    break;
  }
  case '2':
  {
    WoodWidgetType widget = new WoodWidgetType();
    ...
    break;
  }
  case '3':
  {
    PlasticWidgetType widget = new PlasticWidgetType();
    ...
    break;
  }
```

```
  default :
    System.out.println("Invaid Widget Selection!!");
}

proxy.checkWidgets(widgets);
```

37.3.2.3. 서비스 구현

checkWidgets() 서비스의 구현은 위젯 **Type** 오브젝트로 위젯 설명을 가져오고, 위젯의 인벤토리를 확인하고, 주식의 위젯 수를 반환합니다. 대체 그룹을 구현하는 데 사용되는 모든 클래스는 동일한 기본 클래스에서 상속되므로 **JAXB** 특정 **API**를 사용하지 않고 **checkWidgets()** 를 구현할 수 있습니다.

위젯에 대한 대체 그룹의 멤버를 지원하기 위해 생성된 모든 클래스는 위젯 **Type** 클래스 를 확장합니다. 이 사실 때문에 **instanceof** 를 사용하여 전달된 위젯 유형을 확인하고, 필요한 경우 **widgetPart** 오브젝트를 보다 제한적인 유형으로 캐스팅할 수 있습니다. 적절한 유형의 오브젝트가 있으면 위젯의 올바른 종류를 확인할 수 있습니다.

예 37.15. “CheckWidgets()의 서비스 구현()” 가능한 구현을 보여줍니다.

예 37.15. CheckWidgets()의 서비스 구현()

```
public int checkWidgets(WidgetType widgetPart)
{
  if (widgetPart instanceof WidgetType)
  {
    return checkWidgetInventory(widgetType);
  }
  else if (widgetPart instanceof WoodWidgetType)
  {
    WoodWidgetType widget = (WoodWidgetType)widgetPart;
    return checkWoodWidgetInventory(widget);
  }
  else if (widgetPart instanceof PlasticWidgetType)
  {
    PlasticWidgetType widget = (PlasticWidgetType)widgetPart;
    return checkPlasticWidgetInventory(widget);
  }
}
```

37.3.3. placeWidgetOrder 작업

37.3.3.1. 개요

placeWidgetOrder 는 대체 그룹을 포함하는 두 가지 복잡한 유형을 사용합니다. 이 작업은 **Java** 구현

에서 이러한 구조를 사용하는 방법을 보여줍니다. 소비자와 서비스 둘 다 대체 그룹의 멤버를 가져오고 설정해야 합니다.

37.3.3.2. 소비자 구현

***placeWidgetOrder()** 를 호출하려면 소비자가 위젯 대체 그룹의 하나의 요소가 포함된 위젯 순서를 구성해야 합니다. 주문에 위젯을 추가할 때 소비자는 대체 그룹의 각 요소에 대해 생성된 개체 팩토리 메서드를 사용해야 합니다. 이렇게 하면 런타임 및 서비스가 순서를 올바르게 처리할 수 있습니다. 예를 들어, 플라스틱 위젯에 대한 순서가 배치되는 경우 **ObjectFactory.createPlasticWidget()** 메서드는 순서에 추가하기 전에 요소를 만드는 데 사용됩니다.*

*예 37.16. "하위 그룹 멤버 설정" 위젯 **OrderInfo** 개체의 위젯 속성 을 설정하기 위 한 소비자 코드를 표시합니다.*

예 37.16. 하위 그룹 멤버 설정

```
ObjectFactory of = new ObjectFactory();

WidgetOrderInfo order = new of.createWidgetOrderInfo();
...
System.out.println();
System.out.println("What color widgets do you want to order?");
String color = reader.readLine();
System.out.println();
System.out.println("What shape widgets do you want to order?");
String shape = reader.readLine();
System.out.println();
System.out.println("What type of widgets do you want to order?");
System.out.println("1 - Normal");
System.out.println("2 - Wood");
System.out.println("3 - Plastic");
System.out.println("Selection [1-3]");
String selection = reader.readLine();
String trimmed = selection.trim();
char widgetType = trimmed.charAt(0);
switch (widgetType)
{
  case '1':
  {
    WidgetType widget = of.createWidgetType();
    widget.setColor(color);
    widget.setShape(shape);
    JAXB<WidgetType> widgetElement = of.createWidget(widget);
order.setWidget(widgetElement);
    break;
  }
  case '2':
  {
    WoodWidgetType woodWidget = of.createWoodWidgetType();
    woodWidget.setColor(color);
```

```
    woodWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of wood are your widgets?");
    String wood = reader.readLine();
    woodWidget.setWoodType(wood);
    JAXB<WoodWidgetType> widgetElement = of.createWoodWidget(woodWidget);
order.setWoodWidget(widgetElement);
    break;
   }
   case '3':
   {
    PlasticWidgetType plasticWidget = of.createPlasticWidgetType();
    plasticWidget.setColor(color);
    plasticWidget.setShape(shape);
    System.out.println();
    System.out.println("What type of mold to use for your
                widgets?");
    String mold = reader.readLine();
    plasticWidget.setMoldProcess(mold);
    JAXB<WidgetType> widgetElement = of.createPlasticWidget(plasticWidget);
order.setPlasticWidget(widgetElement);
    break;
   }
   default :
    System.out.println("Invaid Widget Selection!!");
    }
```

37.3.3.3. 서비스 구현

placeWidgetOrder() 메서드는 **WidgetOrderInfo** 개체 형태로 주문을 수신하고, 주문을 처리하며, 위젯 **OrderBillInfo** 개체 형태로 소비자에게 청구서를 반환합니다. 주문은 일반 위젯, 플라스틱 위젯 또는 목재 위젯일 수 있습니다. 주문한 위젯 유형은 **widget OrderForm** 오브젝트의 위젯 속성에 저장되는 오브젝트 유형에 따라 결정됩니다. 위젯 속성은 대체 그룹이며 위젯 요소, **woodWid get** 요소 또는 **plasticWidget** 요소를 포함할 수 있습니다.

구현은 순서에 따라 가능한 요소 중 어느 것이 저장되는지 결정해야합니다. 이 작업은 **JAXBElement<? 확장 T >** 개체의 **getName()** 메서드를 사용하여 요소의 **QName**을 결정합니다. 그런 다음 **QName**을 사용하여 대체 그룹의 요소를 순서대로 결정할 수 있습니다. 청구서에 포함된 요소가 알려지면 해당 값을 적절한 유형의 개체로 추출할 수 있습니다.

예 37.17. “placeWidgetOrder()구현” 가능한 구현을 보여줍니다.

예 37.17. placeWidgetOrder()구현

```
public com.widgetvendor.types.widgettypes.WidgetOrderBillInfo
placeWidgetOrder(WidgetOrderInfo widgetOrderForm)
{
  ObjectFactory of = new ObjectFactory();
```

```
WidgetOrderBillInfo bill = new WidgetOrderBillInfo()

 // Copy the shipping address and the number of widgets
 // ordered from widgetOrderForm to bill
 ...

int numOrdered = widgetOrderForm.getAmount();

String elementName = widgetOrderForm.getWidget().getName().getLocalPart();
if (elementName.equals("woodWidget")
{
  WoodWidgetType widget=order.getWidget().getValue();
  buildWoodWidget(widget, numOrdered);

  // Add the widget info to bill
  JAXBElement<WoodWidgetType> widgetElement = of.createWoodWidget(widget);
  bill.setWidget(widgetElement);

  float amtDue = numOrdered * 0.75;
  bill.setAmountDue(amtDue);
}
else if (elementName.equals("plasticWidget")
{
  PlasticWidgetType widget=order.getWidget().getValue();
  buildPlasticWidget(widget, numOrdered);

  // Add the widget info to bill
  JAXBElement<PlasticWidgetType> widgetElement = of.createPlasticWidget(widget);
  bill.setWidget(widgetElement);

  float amtDue = numOrdered * 0.90;
  bill.setAmountDue(amtDue);
}
else
{
  WidgetType widget=order.getWidget().getValue();
  buildWidget(widget, numOrdered);

  // Add the widget info to bill
  JAXBElement<WidgetType> widgetElement = of.createWidget(widget);
  bill.setWidget(widgetElement);

  float amtDue = numOrdered * 0.30;
  bill.setAmountDue(amtDue);
}

 return(bill);
}
```

예 37.17. "placeWidgetOrder()구현"의 코드는 다음을 수행합니다.

요소를 만들기 위해 개체 팩토리를 인스턴스화합니다.Sertss an object factory to create elements.

위젯 OrderBillInfo 개체를 인스턴스화하여 청구서를 보유합니다.

정렬된 위젯 수를 가져옵니다.Gets the number of widgets ordered.

순서에 저장된 요소의 로컬 이름을 가져옵니다.Gets the local name of the element stored in the order.

요소가 woodWidget 요소인지 확인합니다.

요소의 값을 순서에서 적절한 유형의 개체로 추출합니다.

bill에 있는 JAXBElement<T > 개체를 만듭니다.

bill 개체의 위젯 속성 을 설정합니다.

bill 개체의 amountDue 속성을 설정합니다.

38장. 생성 방법 사용자 정의

초록

기본 JAXB 매핑은 XML 스키마를 사용하여 Java 애플리케이션에 대한 개체를 정의할 때 발생하는 대부분의 경우를 해결합니다. 기본 매핑이 충분하지 않은 인스턴스의 경우 JAXB는 광범위한 사용자 지정 메커니즘을 제공합니다.

38.1. 유형 매핑 사용자 정의 기본

38.1.1. 개요

JAXB 사양은 Java 유형이 XML 스키마 구성에 매핑되는 방법을 사용자 지정하는 여러 XML 요소를 정의합니다. 이러한 요소는 XML 스키마 구문으로 인라인으로 지정할 수 있습니다.**These elements can be specified in-line with XML Schema constructs. XML** 스키마 정의를 수정할 수 없거나 사용하지 않으려는 경우 외부 바인딩 문서에서 사용자 정의를 지정할 수 있습니다.**If you cannot, or do not want to, modify the XML Schema definitions, you can specify the customizations in external binding document.**

38.1.2. 네임스페이스

JAXB 데이터 바인딩을 사용자 지정하는 데 사용되는 요소는 네임스페이스 http://java.sun.com/xml/ns/jaxb 에 정의되어 있습니다. 예 38.1. “JAXB 사용자 지정 네임스페이스” 에 표시된 것과 유사한 네임스페이스 선언을 추가해야 합니다. 이는 JAXB 사용자 지정을 정의하는 모든 XML 문서의 루트 요소에 추가됩니다.

> 예 38.1. JAXB 사용자 지정 네임스페이스
>
> ```
> xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
> ```

38.1.3. 버전 선언

JAXB 사용자 지정을 사용하는 경우 사용 중인 JAXB 버전을 표시해야 합니다. 이 작업은 외부 바인딩 선언의 루트 요소에 jaxb:version 특성을 추가하여 수행됩니다. 인라인 사용자 지정을 사용하는 경우 사용자 지정이 포함된 스키마 요소에 jaxb:version 속성을 포함해야 합니다. 특성 값은 항상 2.0 입니다.

예 38.2. “JAXB 사용자 지정 버전 지정” 스키마 요소에 사용된 jaxb:version 특성의 예를 보여줍니다.

> 예 38.2. JAXB 사용자 지정 버전 지정

```
< schema ...
      jaxb:version="2.0">
```

38.1.4. 인라인 사용자 정의 사용

코드 생성기가 **XML** 스키마 구문을 **Java** 구조에 매핑하는 방법을 사용자 지정하는 가장 직접적인 방법은 사용자 지정 요소를 **XML** 스키마 정의에 직접 추가하는 것입니다. **JAXB** 사용자 지정 요소는 수정 중인 **XML** 스키마 구문의 **xsd:appinfo** 요소에 배치됩니다.

예 38.3. “사용자 지정 XML 스키마” 는 **in-line JAXB** 사용자 지정을 포함하는 스키마의 예를 보여줍니다.

예 38.3. 사용자 지정 XML 스키마

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
      jaxb:version="2.0">
  <complexType name="size">
    <annotation> <appinfo> <jaxb:class name="widgetSize" /> </appinfo> </annotation>
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

38.1.5. 외부 바인딩 선언 사용

형식을 정의하는 **XML** 스키마 문서를 변경할 수 없거나 외부 바인딩 선언을 사용하여 사용자 지정을 지정할 수 있습니다.**When you cannot, or do not want to, make changes to the XML Schema document that defines your type, you can specify the customizations using an external binding declaration.** 외부 바인딩 선언은 여러 중첩된 **jaxb:bindings** 요소로 구성됩니다. 예 38.4. “JAXB External Binding Declaration Syntax” 외부 바인딩 선언의 구문을 보여줍니다.

예 38.4. JAXB External Binding Declaration Syntax

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            jaxb:version="2.0">
  <jaxb:bindings [schemaLocation="schemaUri" | wsdlLocation="wsdlUri">
    <jaxb:bindings node="nodeXPath">
      binding declaration
    </jaxb:bindings>
```

```
    ...
  </jaxb:bindings>
<jaxb:bindings>
```

schemaLocation 속성 및 **wsdlLocation** 특성은 수정 사항이 적용되는 스키마 문서를 식별하는 데 사용됩니다. 스키마 문서에서 코드를 생성하는 경우 **schemaLocation** 속성을 사용합니다. **WSDL** 문서에서 코드를 생성하는 경우 **wsdlLocation** 속성을 사용합니다.

node 속성은 수정할 특정 **XML** 스키마 구성을 식별하는 데 사용됩니다. **XML** 스키마 요소로 확인되는 **XPath** 문입니다.**It is an XPath statement that resolves to an XML Schema element.**

스키마 문서 **widgetSchema.xsd** 가 예 38.5. “XML 스키마 파일” 에 표시된 경우 예 38.6. “외부 바인딩 선언” 에 표시된 외부 바인딩 선언은 복잡한 유형 크기 의 생성을 수정합니다.

예 38.5. XML 스키마 파일

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        version="1.0">
  <complexType name="size">
    <sequence>
      <element name="longSize" type="xsd:string" />
      <element name="numberSize" type="xsd:int" />
    </sequence>
  </complexType>
<schema>
```

예 38.6. 외부 바인딩 선언

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="wsdlSchema.xsd">
    <jaxb:bindings node="xsd:complexType[@name='size']">
       <jaxb:class name="widgetSize" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

코드 생성기가 외부 **bing** 선언을 사용하도록 지시하려면 다음과 같이 **wsdl2java** 도구의 **-b binding-file** 옵션을 사용합니다.

```
wsdl2java -b widgetBinding.xml widget.wsdl
```

38.2. XML 스키마의 JAVA 클래스 지정

38.2.1. 개요

기본적으로 **XML** 스키마 유형은 **Java** 기본 유형에 매핑됩니다. **XML** 스키마와 **Java** 간의 가장 논리적인 매핑이지만 애플리케이션 개발자의 요구 사항을 항상 충족하는 것은 아닙니다. **XML** 스키마 기본 유형을 추가 정보를 보유할 수 있는 **Java** 클래스에 매핑하거나 간단한 유형 대체를 허용하는 클래스에 **XML** 프리미티브 유형을 매핑해야 할 수 있습니다.

JAXB javaType 사용자 지정 요소를 사용하면 **XML** 스키마 기본 유형과 **Java** 기본 유형 간의 매핑을 사용자 지정할 수 있습니다. 글로벌 수준 및 개별 인스턴스 수준에서 매핑을 사용자 지정하는 데 사용할 수 있습니다. **javaType** 요소를 단순 형식 정의의 일부로 사용하거나 복잡한 형식 정의의 일부로 사용할 수 있습니다.

javaType 사용자 지정 요소를 사용하는 경우 기본 형식의 **XML** 표현을 대상 **Java** 클래스로 변환하기 위한 메서드를 지정해야 합니다. 일부 매핑에는 기본 변환 방법이 있습니다. 기본 매핑이 없는 인스턴스의 경우 **Apache CXF**는 필요한 메서드 개발을 쉽게 수행하기 위해 **JAXB** 메서드를 제공합니다.

38.2.2. 구문

javaType 사용자 지정 요소는 표 38.1. “XML 스키마 유형에 대한 Java 클래스 생성을 사용자 정의하는 특성”에 설명된 대로 네 가지 속성을 사용합니다.

표 38.1. XML 스키마 유형에 대한 Java 클래스 생성을 사용자 정의하는 특성

속성	필수 항목	설명
name	있음	XML Schema 기본 유형이 매핑되는 Java 클래스의 이름을 지정합니다. 유효한 Java 클래스 이름 또는 Java 기본 유형의 이름이어야 합니다. 이 클래스가 존재하고 애플리케이션에 액세스할 수 있는지 확인해야 합니다. 코드 생성기는 이 클래스를 확인하지 않습니다.
xmlType	없음	사용자 지정할 XML 스키마 기본 유형을 지정합니다.Specifies the XML Schema primitive type that is being customized. 이 특성은 **javaType** 요소가 **globalBindings** 요소의 자식으로 사용되는 경우에만 사용됩니다.

속성	필수 항목	설명
parseMethod	없음	데이터의 문자열 기반 XML 표현을 Java 클래스의 인스턴스로 구문 분석하는 메서드를 지정합니다.Specifies the method responsible for parsing the string-based XML representation of the data into an instance of the Java class. 자세한 내용은 "변환기 지정" 에서 참조하십시오.
printMethod	없음	Java 개체를 데이터의 문자열 기반 XML 표현으로 변환하는 메서드를 지정합니다. 자세한 내용은 "변환기 지정" 에서 참조하십시오.

***javaType** 사용자 지정 요소는 다음 세 가지 방법으로 사용할 수 있습니다.*

- ***XML** 스키마 기본 형식의 모든 인스턴스를 수정하려면 **javaType** 요소가 전역 **Bindings** 사용자 지정 요소의 자식으로 사용될 때 스키마 문서에서 **XML Schema** 유형의 모든 인스턴스를 수정합니다. 이러한 방식으로 사용되는 경우 수정되는 **XML Schema** 기본 유형을 식별하는 **xmlType** 특성 값을 지정해야 합니다.**When it is used in this manner, you must specify a value for the xmlType attribute that identifies the XML Schema primitive type being modified.***

*예 38.7. "글로벌 **Primitive Type** 사용자 정의" 스키마의 모든 인스턴스에 **java.lang.Integer** 를 사용하도록 코드 생성기에 지시하는 인라인 글로벌 사용자 지정을 보여줍니다.*

*예 **38.7.** 글로벌 **Primitive Type** 사용자 정의*

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings ...>
        <jaxb:javaType name="java.lang.Integer"
                       xmlType="xsd:short" />
```

```
    </globalBindings
  </appinfo>
 </annotation>
 ...
</schema>
```

- 간단한 형식 정의를 수정하려면 **javaType** 요소는 이름이 지정된 단순 형식 정의에 적용될 때 **XML** 단순 형식의 모든 인스턴스에 대해 생성된 클래스를 수정합니다. **javaType** 요소를 사용하여 간단한 유형 정의를 수정할 때 **xmlType** 특성을 사용하지 마십시오.

예 38.8. "단순 유형 사용자 지정을 위한 파일 바인딩" **zipCode** 라는 간단한 형식의 생성을 수정하는 외부 바인딩 파일을 보여줍니다.

예 38.8. 단순 유형 사용자 지정을 위한 파일 바인딩

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings wsdlLocation="widgets.wsdl">
    <jaxb:bindings node="xsd:simpleType[@name='zipCode']">
      <jaxb:javaType name="com.widgetVendor.widgetTypes.zipCodeType"
                     parseMethod="com.widgetVendor.widgetTypes.support.parseZipCode"
                     printMethod="com.widgetVendor.widgetTypes.support.printZipCode" />
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

- 복잡한 유형 정의의 요소 또는 특성을 수정하려면 **javaType** 을 **JAXB** 속성 사용자 지정의 일부로 포함하여 복잡한 형식 정의의 개별 부분에 적용할 수 있습니다. **javaType** 요소는 속성의 **baseType** 요소에 자식으로 배치됩니다. **javaType** 요소를 사용하여 복잡한 유형 정의의 특정 부분을 수정하는 경우 **xmlType** 특성을 사용하지 마십시오.

예 38.9. "Complex 유형에서 요소 사용자 지정을 위한 파일 바인딩" 복합 형식의 요소를 수정하는 바인딩 파일을 표시합니다.**Shows a binding file that modifies an element of a complex type.**

예 38.9. Complex 유형에서 요소 사용자 지정을 위한 파일 바인딩

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='cost']">
        <jaxb:property>
          <jaxb:baseType>
```

```
        <jaxb:javaType name="com.widgetVendor.widgetTypes.costType"
                       parseMethod="parseCost"
                       printMethod="printCost" >
      </jaxb:baseType>
    </jaxb:property>
  </jaxb:bindings>
 </jaxb:bindings>
</jaxb:bindings>
```

baseType 요소 사용에 대한 자세한 내용은 38.6절. “Element 또는 Attribute의 기본 유형 지정” 을 참조하십시오.

38.2.3. 변환기 지정

Apache CXF는 **XML** 스키마 기본 유형을 임의의 **Java** 클래스로 변환할 수 없습니다. **javaType** 요소를 사용하여 **XML** 스키마 기본 유형의 매핑을 사용자 지정할 때 코드 생성기는 사용자 지정 **XML** 스키마 기본 유형을 마샬링하고 마샬링하는 데 사용되는 어댑터 클래스를 만듭니다. 샘플 어댑터 클래스는 예 38.10. “JAXB Adapter 클래스” 에 표시되어 있습니다.

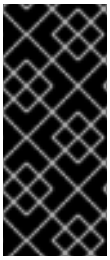
예 38.10. JAXB Adapter 클래스

```
public class Adapter1 extends XmlAdapter<String, javaType>
{
  public javaType unmarshal(String value)
  {
    return(parseMethod(value));
  }

  public String marshal(javaType value)
  {
    return(printMethod(value));
  }
}
```

parseMethod 및 **printMethod** 는 해당 **parseMethod** 특성 및 **printMethod** 속성 값으로 대체됩니다. 값은 유효한 **Java** 메서드를 식별해야 합니다. 다음 두 가지 방법 중 하나로 메서드의 이름을 지정할 수 있습니다.

- 패키지 이름 형식의 정규화된 **Java** 메서드 이름. **ClassName .methodName**
- **methodName**형식의 간단한 메서드 이름입니다.

간단한 메서드 이름만 제공하면 코드 생성기는 **javaType** 요소의 **name** 특성에 지정된 클래스에 메서드가 있다고 가정합니다.



중요

코드 생성기는 **parse** 또는 **print** 메서드를 생성하지 않습니다. 당신은 그들을 공급 할 책임이 있습니다. 구문 분석 및 인쇄 방법을 개발하는 방법에 대한 자세한 내용은 "변환기 구현" 을 참조하십시오.

parseMethod 특성의 값이 제공되지 않으면 코드 생성기는 **name** 특성으로 지정된 **Java** 클래스에 첫 번째 매개 변수가 **Java String** 오브젝트인 생성자가 있다고 가정합니다. 생성된 어댑터의 **unmarshal()** 메서드는 가정 생성자를 사용하여 **Java** 개체를 **XML** 데이터로 채웁니다.

printMethod 속성 값이 제공되지 않으면 코드 생성기는 **name** 속성에 지정된 **Java** 클래스에 **toString()** 메서드가 있다고 가정합니다. 생성된 어댑터의 **marshal()** 메서드는 **Java** 개체를 **XML** 데이터로 변환하는 데 **assumed toString()** 메서드를 사용합니다.

javaType 요소의 **name** 특성이 **Java** 기본 유형을 지정하거나 **Java** 기본의 래퍼 유형 중 하나를 지정하는 경우 코드 생성기는 기본 변환기를 사용합니다. 기본 변환기에 대한 자세한 내용은 "기본 기본 형식 변환기" 을 참조하십시오.

38.2.4. 생성된 내용

"변환기 지정" 에서 언급한 바와 같이 **javaType** 사용자 지정 요소를 사용하면 **XML Schema** 기본 유형의 사용자 지정마다 하나의 어댑터 클래스를 생성합니다. 어댑터의 이름은 패턴 어댑터**N** 을 사용하여 순서대로 지정됩니다. 두 가지 기본 유형 사용자 지정을 지정하는 경우 코드 생성기는 **Adapter1** 및 **Adapter2** 의 두 가지 어댑터 클래스를 생성합니다.

XML 스키마 구성에 대해 생성된 코드는 전역적으로 정의된 요소인지 아니면 복잡한 형식의 일부로 정의되는지에 따라 달라집니다.

XML Schema construct가 전역적으로 정의된 요소인 경우 형식에 대해 생성된 개체 팩토리 메서드는 다음과 같이 기본 메서드에서 수정됩니다.**When the XML Schema construct is a globally defined element, the object factory method generated for the type is modified from the default method as follows:**

- 이 메서드는 **@XmlJavaTypeAdapter** 주석으로 장식됩니다.

주석은 이 요소의 인스턴스를 처리할 때 사용할 어댑터 클래스에 지시합니다. 어댑터 클래스는 클래스 오브젝트로 지정됩니다.

- 기본 유형은 **javaType** 요소의 **name** 특성으로 지정된 클래스로 교체됩니다.

예 38.11. "Global Element에 대한 사용자 정의 Object Factory Method" 예 38.7. "글로벌 Primitive Type 사용자 정의" 에 표시된 사용자 지정의 영향을 받는 요소의 오브젝트 팩토리 메서드를 보여줍니다.

예 38.11. Global Element에 대한 사용자 정의 Object Factory Method

```
@XmlElementDecl(namespace = "http://widgetVendor.com/types/widgetTypes", name = "shorty")
@XmlJavaTypeAdapter(org.w3._2001.xmlschema.Adapter1.class)
public JAXBElement<Integer> createShorty(Integer value) {
    return new JAXBElement<Integer>(_Shorty_QNAME, Integer.class, null, value);
}
```

XML 스키마 구성이 복잡한 유형의 일부로 정의되면 생성된 **Java** 속성은 다음과 같이 수정됩니다.

- 속성은 **@XmlJavaTypeAdapter** 주석으로 장식됩니다.

  주석은 이 요소의 인스턴스를 처리할 때 사용할 어댑터 클래스에 지시합니다. 어댑터 클래스는 클래스 오브젝트로 지정됩니다.

- 속성의 **@XmlElement** 에는 형식 속성이 포함되어 있습니다.

  type 속성의 값은 생성된 개체의 기본 유형을 나타내는 클래스 오브젝트입니다. **XML** 스키마 기본 유형의 경우 클래스는 **String** 입니다.

- 속성은 **@XmlSchemaType** 주석으로 장식됩니다.

  주석은 구성의 **XML** 스키마 기본 유형을 식별합니다.

- 기본 유형은 **javaType** 요소의 **name** 특성으로 지정된 클래스로 교체됩니다.

예 38.12. "사용자 정의 복잡성 유형" 예 38.7. "글로벌 Primitive Type 사용자 정의" 에 표시된 사용자 지정의 영향을 받는 요소의 오브젝트 팩토리 메서드를 보여줍니다.

예 38.12. 사용자 정의 복잡성 유형

```
public class NumInventory {

    @XmlElement(required = true, type = String.class) @XmlJavaTypeAdapter(Adapter1.class)
@XmlSchemaType(name = "short") protected Integer numLeft;
    @XmlElement(required = true)
    protected String size;

    public Integer getNumLeft() {
        return numLeft;
    }

    public void setNumLeft(Integer value) {
        this.numLeft = value;
    }

    public String getSize() {
        return size;
    }

    public void setSize(String value) {
        this.size = value;
    }

}
```

38.2.5. 변환기 구현

Apache CXF 런타임에서는 javaType 요소에서 지정한 Java 클래스로 XML 기본 형식을 변환하는 방법을 알지 못합니다. 단, parseMethod 속성 및 printMethod 속성에 의해 지정된 메서드를 호출해야 합니다. 런타임에서 호출하는 메서드의 구현을 제공해야 합니다. 구현된 메서드는 XML 기본 형식의 어휘 구조로 작업할 수 있어야 합니다.The implemented methods must be able to working with the lexical structures of the XML primitive type.

데이터 변환 방법의 구현을 단순화하기 위해 Apache CXF는 javax.xml.bind.DatatypeConverter 클래스를 제공합니다. 이 클래스는 모든 XML 스키마 기본 유형을 구문 분석하고 인쇄하는 메서드를 제공합니다. 구문 분석 메서드는 XML 데이터를 문자열 표현하고 표 34.1. "XML Schema Primitive Type to Java Native Type Mapping" 에 정의된 기본 유형의 인스턴스를 반환합니다. 인쇄 메서드는 기본 형식의 인스턴스를 사용 하 고 XML 데이터의 문자열 표현을 반환 합니다.The print methods take an instance of the default type and they return a string representation of the XML data.

DatatypeConverter 클래스에 대한 Java 문서는 https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/DatatypeConverter.html 에서 확인할 수

있습니다.

38.2.6. 기본 기본 형식 변환기

Java 기본 유형 또는 Java 기본 유형 Wrapper 클래스 중 하나를 지정할 때 javaType 요소의 name 속성에는 parseMethod 속성 또는 printMethod 속성에 대한 값을 지정할 필요가 없습니다. Apache CXF 런타임은 값이 제공되지 않는 경우 기본 변환기를 대체합니다.

기본 데이터 변환기에서는 JAXB DatatypeConverter 클래스를 사용하여 XML 데이터를 구문 분석합니다. 기본 변환기는 변환 작업을 수행하는 데 필요한 모든 형식 캐스팅도 제공합니다.

38.3. SIMPLE TYPES을 위한 JAVA 클래스 생성

38.3.1. 개요

기본적으로 명명된 단순 형식은 열거형이 아닌 한 생성된 형식을 가져오지 않습니다.**By default, named simple types do not result in generated types unless they are enumerations.** 간단한 유형을 사용하여 정의된 요소는 Java 기본 유형의 속성에 매핑됩니다.

Java 클래스에 간단한 유형이 생성되어야 하는 경우가 있습니다(예: 유형 대체를 사용하려는 경우).

전역적으로 정의된 모든 간단한 유형에 대한 클래스를 생성하도록 코드 생성기에 지시하려면 globalBindings 사용자 지정 요소의 mapSimpleTypeDef 를 true 로 설정합니다.

38.3.2. 사용자 정의 추가

코드 생성기가 간단한 유형의 Java 클래스를 생성하도록 지시하기 위해 globalBinding 요소의 mapSimpleTypeDef 특성을 추가하고 해당 값을 true 로 설정합니다.

예 38.13. “SimpleTypes에 대한 Java 클래스 생성을 위한 인라인 사용자 지정” 코드 생성기가 이름이 지정된 간단한 유형에 대해 Java 클래스를 생성하도록 강제 적용하는 인 줄 사용자 지정을 보여줍니다.

예 38.13. SimpleTypes에 대한 Java 클래스 생성을 위한 인라인 사용자 지정

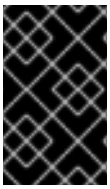
```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
```

```
  <annotation>
    <appinfo>
      <jaxb:globalBindings mapSimpleTypeDef="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

예 38.14. "파일의 강제 적용에 파일 바인딩" 간단한 형식의 생성을 사용자 지정하는 외부 바인딩 파일을 보여줍니다.

예 38.14. 파일의 강제 적용에 파일 바인딩

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings mapSimpleTypeDef="true" />
  <jaxb:bindings>
<jaxb:bindings>
```



중요

이 사용자 지정은 전역 범위에 정의된 간단한 유형만 영향을 미 칩니다.

38.3.3. 생성된 클래스

*단순 형식에 대해 생성된 클래스에는 **value** 라는 하나의 속성이 있습니다. **value** 속성은 34.1절. "기본 유형" 의 매핑에 의해 정의된 **Java** 유형 중 하나입니다. 생성된 클래스에는 **value** 속성의 **getter** 및 **setter** 가 있습니다.*

*예 38.16. "간단한 유형의 사용자 지정 매핑" 는 예 38.15. "사용자 지정 매핑을 위한 간단한 유형" 에 정의된 간단한 유형에 대해 생성된 **Java** 클래스를 보여줍니다.*

예 38.15. 사용자 지정 매핑을 위한 간단한 유형

```
<simpleType name="simpleton">
  <restriction base="xsd:string">
    <maxLength value="10"/>
  </restriction>
</simpleType>
```

예 38.16. 간단한 유형의 사용자 지정 매핑

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "simpleton", propOrder = {"value"})
public class Simpleton {

    @XmlValue
    protected String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

}
```

38.4. ENUMERATION 매핑 사용자 정의

38.4.1. 개요

xsd:string 이외의 스키마 형식을 기반으로 열거된 형식을 사용하려면 코드 생성기에 매핑하도록 지시합니다.**If you want enumerated types that are based on a schema type other than xsd:string, you must instruct the code generator to map it.** 생성된 열거형 상수의 이름을 제어할 수도 있습니다.**You can also control the name of the generated enumeration constants.**

사용자 지정은 **jaxb:typesafeEnumClass** 요소와 함께 하나 이상의 **jaxb:typesafeEnumMember** 요소를 사용하여 수행됩니다.

코드 생성기에 대한 기본 설정이 모든 열거 멤버에 대해 유효한 **Java** 식별자를 만들 수 없는 인스턴스도 있을 수 있습니다. 전역**Bindings** 사용자 지정의 특성을 사용하여 코드 생성기에서 이 문제를 처리하는 방법을 사용자 지정할 수 있습니다.**You can customize how the code generators handle this by using an attribute of the globalBindings custom.**

38.4.2. 멤버 이름 사용자 정의기

코드 생성기의 멤버를 생성할 때 이름 지정 충돌이 발생하거나 열거형 멤버에 유효한 **Java ID**를 만들 수 없는 경우 코드 생성기는 기본적으로 경고를 생성하고 열거형에 대한 **Java** 열거형 유형을 생성하지 않습니다.**If the code generator encounters a naming collision when generating the members of an enumeration or if it cannot create a valid Java identifier for a member of the enumeration, the**

code generator, by default, generates a warning and does not generate a Java enum type for the enumeration.

전역Binding 요소의 typesafeEnumMemberName 속성을 추가하여 이 동작을 변경할 수 있습니다. typesafeEnumMemberName 속성 값은 표 38.2. "Customizing Enumeration Member Name Generation의 값" 에 설명되어 있습니다.

표 38.2. Customizing Enumeration Member Name Generation의 값

값	설명
skipGeneration(default)	Java **enum** 유형이 생성되지 않고 경고를 생성하도록 지정합니다.
generateName	멤버 이름이 **VALUE_*N*** 패턴 뒤에 생성되도록 지정합니다. n은 1에서 시작하여 열거형의 각 멤버에 대해 증가됩니다. *N* starts off at one, and is incremented for each member of the enumeration.
generateError	열거형을 Java enum 형식에 매핑할 수 없는 경우 코드 생성기가 오류를 생성하도록 지정합니다.Specifies that the code generator generates an error when it cannot map an enumeration to a Java **enum** type.

예 38.17. "사용자 정의 유형 유형 보안 구성원 이름을 강제 적용" 코드 생성기가 형식 안전한 멤버 이름을 생성하도록 강제 적용하는 인라인 사용자 지정을 보여줍니다.

예 38.17. 사용자 정의 유형 유형 보안 구성원 이름을 강제 적용

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings typesafeEnumMemberName="generateName" />
    </appinfo>
  </annotation>
  ...
</schema>
```

38.4.3. 클래스 사용자 정의 프로그램

jaxb:typesafeEnumClass 요소는 XML 스키마 열거형이 Java enum 형식에 매핑되도록 지정합니다.

*여기에는 표 38.3. "생성된 Enumeration 클래스 사용자 정의 특성" 에 설명된 두 가지 속성이 있습니다. **jaxb:typesafeEnumClass** 요소가 줄에서 지정되면 수정되는 간단한 유형의 **xsd:annotation** 요소에 배치해야 합니다.*

표 38.3. 생성된 Enumeration 클래스 사용자 정의 특성

속성	설명
name	생성된 Java 열거형 유형의 이름 을 지정합니다. 이 값은 유효한 Java 식별자여야 합니다.
map	열거형을 Java enum 형식에 매핑해야 하는지 여부를 지정합니다.Specifies if the enumeration should be mapped to a Java **enum** type. 기본값은 **true** 입니다.

38.4.4. 멤버 customizer

***jaxb:typesafeEnumMember** 요소는 **XML** 스키마 열거형과 **Java enum** 형식 상 수 간의 매핑을 지정합니다. 사용자 지정할 각 열거형 **facet**에 대해 하나의 **jaxb:typesafeEnumMember** 요소를 사용해야 합니다.*

인라인 사용자 지정을 사용하는 경우 이 요소를 다음 두 가지 방법 중 하나로 사용할 수 있습니다.

- *수정되는 열거형 **facet**의 **xsd:annotation** 요소 내에 배치할 수 있습니다.*
- *모두 열거형을 사용자 지정하는 데 사용되는 **jaxb:typesafeEnumClass** 요소의 자식으로 배치할 수 있습니다.*

***jaxb:typesafeEnumMember** 요소에는 필요한 **name** 속성이 있습니다. **name** 속성은 생성된 **Java enum** 형식 상수의 이름을 지정합니다. 값은 유효한 **Java** 식별자여야 합니다.*

***jaxb:typesafeEnumMember** 요소에도 **value** 속성이 있습니다. 이 값은 열거 형 **facet**를 적절한 **jaxb:typesafeEnumMember** 요소와 연결하는 데 사용됩니다. **value** 속성 값은 열거 형 **facets' value** 속성의 값 중 하나와 일치해야 합니다. 이 속성은 형식 생성을 사용자 지정하기 위해 외부 바인딩 사양을 사용하거나 **jaxb:typesafeEnumMember** 요소를 **jaxb:typesafeEnumClass** 요소의 하위 항목으로 그룹화할 때 필요합니다.*

38.4.5. 예제

예 38.18. "Enumerated Type의 줄 사용자 지정" 인라인 사용자 지정을 사용하고 열거형의 멤버를 별

*도로 사용자 지정하는 열거된 유형을 표시합니다.**Shows an enumerated type that uses in-line custom and has the enumeration's members customized separately.***

예 38.18. Enumerated Type의 줄 사용자 지정

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass />
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="one" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="2">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="two" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="3">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="three" />
          </appinfo>
        </annotation>
      </enumeration>
      <enumeration value="4">
        <annotation>
          <appinfo>
            <jaxb:typesafeEnumMember name="four" />
          </appinfo>
        </annotation>
      </enumeration>
    </restriction>
  </simpleType>
<schema>
```

예 38.19. “Combined Mapping을 사용하여 Enumerated Type의 줄 사용자 지정” 는 인라인 사용자 지정을 사용하고 클래스 사용자 지정에서 멤버의 사용자 지정을 결합하는 열거된 유형을 보여줍니다.

예 38.19. Combined Mapping을 사용하여 Enumerated Type의 줄 사용자 지정

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <simpleType name="widgetInteger">
    <annotation>
      <appinfo>
        <jaxb:typesafeEnumClass>
          <jaxb:typesafeEnumMember value="1" name="one" />
          <jaxb:typesafeEnumMember value="2" name="two" />
          <jaxb:typesafeEnumMember value="3" name="three" />
          <jaxb:typesafeEnumMember value="4" name="four" />
        </jaxb:typesafeEnumClass>
      </appinfo>
    </annotation>
    <restriction base="xsd:int">
      <enumeration value="1" />
      <enumeration value="2" />
      <enumeration value="3" />
      <enumeration value="4" >
    </restriction>
  </simpleType>
<schema>
```

예 38.20. “Enumeration을 사용자 정의하는 바인딩 파일” 열거된 형식을 사용자 지정하는 외부 바인딩 파일을 표시합니다.

예 38.20. Enumeration을 사용자 정의하는 바인딩 파일

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:simpleType[@name='widgetInteger']">
      <jaxb:typesafeEnumClass>
        <jaxb:typesafeEnumMember value="1" name="one" />
        <jaxb:typesafeEnumMember value="2" name="two" />
        <jaxb:typesafeEnumMember value="3" name="three" />
        <jaxb:typesafeEnumMember value="4" name="four" />
      </jaxb:typesafeEnumClass>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

38.5. FIXED VALUE ATTRIBUTE MAPPING

38.5.1. 개요

기본적으로 코드 생성기는 고정 값을 일반 속성에 맞게 사용하는 것으로 정의된 특성을 매핑합니다.**By default, the code generators map attributes defined as having a fixed value to normal properties.** 스키마 유효성 검사를 사용하는 경우 **Apache CXF**는 스키마 정의를 적용할 수 있습니다(24.3.4.7절. “스키마 유효성 검사 유형 값”참조). 그러나 스키마 유효성 검사를 사용하면 메시지 처리 시간이 증가합니다.

고정 값이 있는 속성을 **Java**에 매핑하는 또 다른 방법은 **Java** 상수에 매핑하는 것입니다. **globalBindings** 사용자 지정 요소를 사용하여 고정 값 속성을 **Java** 상수에 매핑하도록 코드 생성기에 지시할 수 있습니다. **property** 요소를 사용하여 보다 지역화된 수준에서 **Java** 상수에 고정 값 속성의 매핑을 사용자 지정할 수도 있습니다.

38.5.2. 글로벌 사용자 정의

전역**Binding** 요소의 **fixedAttributeAsConstantProperty** 속성을 추가하여 이 동작을 변경할 수 있습니다. 이 속성을 **true** 로 설정하면 고정 특성을 사용하여 정의된 모든 속성을 **Java** 상수에 매핑하도록 코드 생성기가 지시합니다.

예 38.21. “in-Line Customization to Force Generation of Constants” 코드 생성기가 고정 값을 사용하여 속성에 대해 상수를 생성하도록 강제 적용하는 인라인 사용자 지정을 보여줍니다.

예 38.21. in-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <annotation>
    <appinfo>
      <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
    </appinfo>
  </annotation>
  ...
</schema>
```

예 38.22. “파일의 강제 적용에 파일 바인딩” 고정 속성의 생성을 사용자 지정하는 외부 바인딩 파일을 보여줍니다.

예 38.22. 파일의 강제 적용에 파일 바인딩

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:globalBindings fixedAttributeAsConstantProperty="true" />
  <jaxb:bindings>
<jaxb:bindings>
```

38.5.3. 로컬 매핑

속성 요소의 **fixedAttributeAsConstantProperty** 속성 을 사용하여 특성 매핑을 특성별로 사용자 지정할 수 있습니다. 이 속성을 **true** 로 설정하면 고정 특성을 사용하여 정의된 모든 속성을 **Java** 상수에 매핑하도록 코드 생성기가 지시합니다.

예 38.23. "in-Line Customization to Force Generation of Constants" 코드 생성기가 고정된 값으로 단일 속성에 대해 상수를 생성하도록 강제 적용하는 인라인 사용자 지정을 보여줍니다.

예 38.23. in-Line Customization to Force Generation of Constants

```
<schema targetNamespace="http://widget.com/types/widgetTypes"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">
  <complexType name="widgetAttr">
    <sequence>
      ...
    </sequence>
    <attribute name="fixer" type="xsd:int" fixed="7">
      <annotation> <appinfo> <jaxb:property fixedAttributeAsConstantProperty="true" /> </appinfo>
</annotation>
    </attribute>
  </complexType>
  ...
</schema>
```

예 38.24. "파일의 강제 적용에 파일 바인딩" 고정 속성의 생성을 사용자 지정하는 외부 바인딩 파일을 표시합니다.

예 38.24. 파일의 강제 적용에 파일 바인딩

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
            jaxb:version="2.0">
  <jaxb:bindings schemaLocation="types.xsd">
    <jaxb:bindings node="xsd:complexType[@name='widgetAttr']">
      <jaxb:bindings node="xsd:attribute[@name='fixer']">
          <jaxb:property fixedAttributeAsConstantProperty="true" />
        </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

38.5.4. Java 매핑

기본 매핑에서 모든 속성은 **getter** 및 **setter** 메서드를 사용하여 표준 **Java** 속성에 매핑됩니다. 이 사용자 지정이 고정 특성을 사용하여 정의된 속성에 적용되는 경우 예 38.25. "고정 값 속성을 Java Constant에 매핑" 에서와 같이 속성이 **Java** 상수에 매핑됩니다.

예 38.25. 고정 값 속성을 Java Constant에 매핑

```
@XmlAttribute
public final static type NAME = value;
```

유형 은 34.1절. "기본 유형" 에서 설명하는 매핑을 사용하여 특성의 기본 유형을 **Java** 유형에 매핑하여 결정됩니다.

NAME 은 특성 요소의 **name** 속성 값을 모든 대문자로 변환하여 결정됩니다.

값은 특성 요소의 고정 특성 값에 따라 결정됩니다.

예를 들어 예 38.23. "in-Line Customization to Force Generation of Constants" 에 정의된 속성은 예 38.26. "고정 값 특성을 Java Constant로 매핑" 로 매핑됩니다.

예 38.26. 고정 값 특성을 Java Constant로 매핑

```
@XmlRootElement(name = "widgetAttr")
public class WidgetAttr {

  ...

  @XmlAttribute
  public final static int FIXER = 7;
```

```
    ...
}
```

38.6. ELEMENT 또는 ATTRIBUTE의 기본 유형 지정

38.6.1. 개요

경우에 따라 요소에 대해 생성된 개체의 클래스를 사용자 지정하거나 **XML** 스키마 복잡한 유형의 일부로 정의된 특성을 사용자 지정해야 합니다. 예를 들어 간단한 형식 대체를 허용하도록 보다 일반적인 개체 클래스를 사용할 수 있습니다.**For example, you might want to use a more generalized class of object to allow for simple type substitution.**

이를 수행하는 한 가지 방법은 **JAXB** 기본 유형 사용자 지정을 사용하는 것입니다. 경우에 따라 개발자가 요소 또는 특성을 나타내기 위해 생성된 오브젝트의 클래스를 지정할 수 있습니다. 기본 유형 사용자 지정을 사용하면 **XML** 스키마 구문과 생성된 **Java** 오브젝트 간의 대체 매핑을 지정할 수 있습니다. 이 대체 매핑은 간단한 전문 분야 또는 기본 기본 클래스의 일반화일 수 있습니다. **XML** 스키마 기본 유형을 **Java** 클래스에 매핑할 수도 있습니다.

38.6.2. 사용자 정의 사용

XML 스키마 구문에 **JAXB** 기본 유형 속성을 적용하려면 **JAXB baseType** 사용자 지정 요소를 사용합니다. **baseType** 사용자 지정 요소는 **JAXB** 속성 요소의 자식이므로 적절히 중첩해야 합니다.

XML 스키마 구문의 매핑을 **Java** 오브젝트에 사용자 지정하는 방법에 따라 **baseType** 사용자 지정 요소의 **name** 속성 또는 **javaType** 하위 요소를 추가합니다. **name** 특성은 생성된 오브젝트의 기본 클래스를 동일한 클래스 계층 구조 내의 다른 클래스에 매핑하는 데 사용됩니다. **javaType** 요소는 **XML** 스키마 기본 유형을 **Java** 클래스에 매핑하려는 경우 사용됩니다.

중요

동일한 **baseType** 사용자 지정 요소에서 **name** 특성과 **javaType** 하위 요소를 둘 다 사용할 수 없습니다.

38.6.3. 기본 매핑 특수화 또는 일반화

baseType 사용자 지정 요소의 **name** 특성은 동일한 **Java** 클래스 계층 구조 내의 클래스에 생성된 개체의 클래스를 재정의하는 데 사용됩니다. 속성은 **XML Schema** 구문을 매핑할 **Java** 클래스의 정규화된 이름을 지정합니다. 지정된 **Java** 클래스는 코드 생성기가 일반적으로 **XML Schema** 구문에 대해 생성하

*는 **Java** 클래스의 슈퍼 클래스 또는 하위 클래스 여야 합니다. **Java** 기본 유형에 매핑되는 **XML** 스키마 기본 형식의 경우 래퍼 클래스는 사용자 지정을 위해 기본 기본 클래스로 사용됩니다.*

*예를 들어 **xsd:int** 로 정의된 요소는 **java.lang.Integer** 를 기본 기본 클래스로 사용합니다. **name** 특성의 값은 **Number** 또는 **Object** 와 같은 **Integer** 의 모든 슈퍼 클래스를 지정할 수 있습니다.*

*간단한 유형 대체의 경우 가장 일반적인 사용자 지정은 기본 유형을 **Object** 오브젝트에 매핑하는 것입니다.*

*예 38.27. "기본 유형의 인라인 사용자 지정" 복잡한 유형의 한 요소를 **Java Object** 오브젝트에 매핑하는 인라인 사용자 지정을 보여줍니다.*

예 38.27. 기본 유형의 인라인 사용자 지정

```
<complexType name="widgetOrderInfo">
  <all>
    <element name="amount" type="xsd:int" />
     <element name="shippingAdress" type="Address">
      <annotation> <appinfo> <jaxb:property> <jaxb:baseType name="java.lang.Object" />
</jaxb:property> </appinfo> </annotation>
    </element>
    <element name="type" type="xsd:string"/>
  </all>
</complexType>
```

예 38.28. "기본 유형 사용자 지정을 위한 외부 바인딩 파일" 는 예 38.27. "기본 유형의 인라인 사용자 지정" 에 표시된 사용자 지정의 외부 바인딩 파일을 표시합니다.

예 38.28. 기본 유형 사용자 지정을 위한 외부 바인딩 파일

```
<jaxb:bindings xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               jaxb:version="2.0">
  <jaxb:bindings schemaLocation="enumMap.xsd">
    <jaxb:bindings node="xsd:ComplexType[@name='widgetOrderInfo']">
      <jaxb:bindings node="xsd:element[@name='shippingAddress']">
        <jaxb:property>
          <jaxb:baseType name="java.lang.Object" />
        </jaxb:property>
      </jaxb:bindings>
    </jaxb:bindings>
  </jaxb:bindings>
<jaxb:bindings>
```

결과 Java 오브젝트의 @XmlElement 주석에는 type 속성이 포함됩니다. type 속성의 값은 생성된 개체의 기본 유형을 나타내는 클래스 오브젝트입니다. XML 스키마 기본 형식의 경우 클래스는 해당 Java 기본 유형의 래퍼 클래스입니다.

예 38.29. "수정 기본 클래스가 있는 Java 클래스" 예 38.28. "기본 유형 사용자 지정을 위한 외부 바인딩 파일" 의 스키마 정의에 따라 생성된 클래스를 보여줍니다.

예 38.29. 수정 기본 클래스가 있는 Java 클래스

```
public class WidgetOrderInfo {

    protected int amount;
    @XmlElement(required = true)
    protected String type;
    @XmlElement(required = true, type = Address.class) protected Object shippingAddress;

    ...
    public Object getShippingAddress() {
        return shippingAddress;
    }

    public void setShippingAddress(Object value) {
        this.shippingAddress = value;
    }

}
```

38.6.4. javaType 사용

javaType 요소는 XML 스키마 기본 유형을 사용하여 정의된 요소 및 특성이 Java 개체에 매핑되는 방법을 사용자 지정하는 데 사용할 수 있습니다. javaType 요소를 사용하면 baseType 요소의 name 속성을 사용하는 것보다 훨씬 더 유연하게 사용할 수 있습니다. javaType 요소를 사용하면 기본 유형을 오브젝트의 모든 클래스에 매핑할 수 있습니다.

javaType 요소 사용에 대한 자세한 설명은 38.2절. "XML 스키마의 Java 클래스 지정" 을 참조하십시오.

39장. A JAXBCONTEXT 오브젝트 사용

초록

JAXBContext 개체를 사용하면 **Apache CXF**의 런타임이 **XML** 요소와 **Java** 개체 간에 데이터를 변환할 수 있습니다. 애플리케이션 개발자는 메시지 처리기에서 **JAXB** 개체를 사용하고 원시 **XML** 메시지로 작업하는 소비자를 구현할 때 **JAXBContext** 개체를 인스턴스화해야 합니다.

39.1. 개요

JAXBContext 개체는 런타임에서 사용하는 하위 수준 오브젝트입니다. 런타임은 **XML** 요소와 해당 **Java** 표현을 변환할 수 있습니다. 애플리케이션 개발자는 일반적으로 **JAXBContext** 개체를 사용할 필요가 없습니다. **XML** 데이터의 마샬링 및 분리는 일반적으로 **JAX-WS** 애플리케이션의 전송 및 바인딩 계층에서 처리됩니다.

그러나 애플리케이션이 **XML** 메시지 콘텐츠를 직접 조작해야 하는 경우가 있습니다. 다음 두 가지 경우 중 하나를 수행합니다.

- 41.1절. “소비자에서 XML 사용”
- 43장. 핸들러 작성

사용 가능한 **JAXBContext.newInstance()** 메서드 중 하나를 사용하여 **JAXBContext** 개체를 인스턴스화해야 합니다.

39.2. 모범 사례

jaxbContext 오브젝트는 인스턴스화하는 데 리소스 집약적입니다. 애플리케이션에서 가능한 적은 수의 인스턴스를 생성하는 것이 좋습니다. 이를 수행하는 한 가지 방법은 애플리케이션에서 사용하는 모든 **JAXB** 개체를 관리하고 가능한 한 많은 애플리케이션 부분 간에 공유할 수 있는 단일 **JAXBContext** 개체를 만드는 것입니다.

jaxbContext 개체는 스레드로부터 안전 합니다.

39.3. 개체 팩토리를 사용하여 JAXBCONTEXT 개체 가져오기

JAXBContext 클래스는 JAXB 개체를 구현하는 클래스 목록을 사용하는 예 39.1. "클래스를 사용하여 JAXB 컨텍스트 가져오기" 에 표시된 newInstance() 메서드를 제공합니다.

> 예 39.1. 클래스를 사용하여 JAXB 컨텍스트 가져오기
>
> staticJAXBContextnewInstanceClass...classesToBeBoundJAXBException

반환된 JAXBObject 오브젝트는 메서드에 전달된 클래스에서 구현하는 JAXB 오브젝트에 대해 마샬링 및 unmarshal 데이터를 마샬링할 수 있습니다. 또한 메서드에 전달된 모든 클래스에서 정적으로 참조되는 모든 클래스를 사용할 수 있습니다.

애플리케이션에서 사용하는 모든 JAXB 클래스의 이름을 newInstance() 메서드에 전달할 수 있지만 효율적이지 않습니다. 동일한 목표를 달성하는 보다 효율적인 방법은 애플리케이션에 생성된 오브젝트 팩토리 또는 개체 팩토리를 전달하는 것입니다. 결과 JAXBContext 개체는 지정된 오브젝트 팩토리를 인스턴스화할 수 있는 JAXB 클래스를 관리할 수 있습니다.

39.4. 패키지 이름을 사용하여 JAXBCONTEXT 개체 가져오기

JAXBContext 클래스는 예 39.2. "클래스를 사용하여 JAXB 컨텍스트 가져오기" 에 표시된 newInstance() 메서드를 제공하며, 여기에는 콜론(:) 별도의 패키지 이름 목록을 사용합니다. 지정된 패키지에는 XML 스키마에서 파생된 JAXB 개체가 포함되어야 합니다.

> 예 39.2. 클래스를 사용하여 JAXB 컨텍스트 가져오기
>
> staticJAXBContextnewInstanceStringcontextPathJAXBException

반환된 JAXBContext 오브젝트는 지정된 패키지의 클래스에서 구현하는 모든 JAXB 오브젝트에 대해 마샬링 및 unmarshal 데이터를 마샬링할 수 있습니다.

40장. 비동기 애플리케이션 개발

초록

JAX-WS는 서비스에 비동기적으로 액세스하기 위한 쉬운 메커니즘을 제공합니다. **SEI**는 서비스에 비동기적으로 액세스하는 데 사용할 수 있는 추가 방법을 지정할 수 있습니다. **Apache CXF** 코드 생성기는 추가 방법을 생성합니다. 비즈니스 논리를 추가하기만 하면 됩니다.

40.1. 비동기 종료의 유형

일반적인 동기 호출 모드 외에도 **Apache CXF**는 두 가지 비동기 호출을 지원합니다.

- 폴링 접근 방식을 사용하여 원격 작업을 호출하려면 출력 매개 변수가 없지만 **javax.xml.ws.Response** 오브젝트를 반환하는 메서드를 호출합니다. **Response** 오브젝트(**javax.util.concurrency.Future** 인터페이스에서 상속됨)를 폴링하여 응답 메시지가 도착했는지 여부를 확인할 수 있습니다.

- 콜백 접근 방식 - 콜백 접근 방식을 사용하여 원격 작업을 호출하려면 콜백 오브젝트(**Java x.xml.ws.AsyncHandler** 유형)에 대한 참조를 해당 매개 변수 중 하나로 사용하는 메서드를 호출합니다. 응답 메시지가 클라이언트에 도착하면 런타임은 **AsyncHandler** 개체에서 다시 호출하고 응답 메시지의 콘텐츠를 제공합니다.

40.2. 비동기 예에 대한 WSDL

예 40.1. "비동기에 대한 WSDL 계약 예" 비동기 예제에 사용되는 **WSDL** 계약을 표시합니다. 계약은 단일 인터페이스인 **GreeterAsync**를 정의하며 단일 작업인 **greetMeSometime**을 포함합니다.

예 40.1. 비동기에 대한 WSDL 계약 예

```
<?xml version="1.0" encoding="UTF-8"?><wsdl:definitions
xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://apache.org/hello_world_async_soap_http"
             xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://apache.org/hello_world_async_soap_http"
             name="HelloWorld">
  <wsdl:types>
    <schema targetNamespace="http://apache.org/hello_world_async_soap_http/types"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:x1="http://apache.org/hello_world_async_soap_http/types"
        elementFormDefault="qualified">
```

```
    <element name="greetMeSometime">
     <complexType>
      <sequence>
       <element name="requestType" type="xsd:string"/>
      </sequence>
     </complexType>
    </element>
    <element name="greetMeSometimeResponse">
     <complexType>
      <sequence>
       <element name="responseType"
              type="xsd:string"/>
      </sequence>
     </complexType>
    </element>
  </schema>
</wsdl:types>

<wsdl:message name="greetMeSometimeRequest">
  <wsdl:part name="in" element="x1:greetMeSometime"/>
</wsdl:message>
<wsdl:message name="greetMeSometimeResponse">
  <wsdl:part name="out"
          element="x1:greetMeSometimeResponse"/>
</wsdl:message>

<wsdl:portType name="GreeterAsync">
  <wsdl:operation name="greetMeSometime">
    <wsdl:input name="greetMeSometimeRequest"
             message="tns:greetMeSometimeRequest"/>
    <wsdl:output name="greetMeSometimeResponse"
              message="tns:greetMeSometimeResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="GreeterAsync_SOAPBinding"
            type="tns:GreeterAsync">
  ...
</wsdl:binding>

<wsdl:service name="SOAPService">
  <wsdl:port name="SoapPort"
          binding="tns:GreeterAsync_SOAPBinding">
    <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

40.3. STUB 코드 생성

40.3.1. 개요

비동기 호출 스타일에는 **SEI**에 정의된 전용 비동기 메서드에 대한 추가 스텁 코드가 필요합니다. 이 특수 스텁 코드는 기본적으로 생성되지 않습니다. 비동기 기능을 전환하고 필수 스텁 코드를 생성하려면 **WSDL 2.0** 사양의 매핑 사용자 지정 기능을 사용해야 합니다.

사용자 지정을 사용하면 **Maven** 코드 생성 플러그인에서 스텁 코드를 생성하는 방식을 수정할 수 있습니다. 특히 **WSDL-to-Java** 매핑을 수정하고 특정 기능을 전환할 수 있습니다. 여기에서 사용자 지정은 비동기 호출 기능을 전환하는 데 사용됩니다. 사용자 지정은 **jaxws:bindings** 태그를 사용하여 정의한 바인딩 선언을 사용하여 지정됩니다(**jaxws** 접두사가 **http://java.sun.com/xml/ns/jaxws** 네임스페이스와 연결되어 있음). 바인딩 선언을 지정하는 방법은 다음 두 가지가 있습니다.

외부 바인딩 선언

외부 바인딩 선언을 사용할 때 **jaxws:bindings** 요소는 **WSDL** 계약과 별도로 파일에 정의됩니다. 스텁 코드를 생성할 때 코드 생성기에 바인딩 선언 파일의 위치를 지정합니다.**You specify the location of the binding declaration file to code generator when you generate the stub code.**

포함된 바인딩 선언

포함된 바인딩 선언을 사용하는 경우 **jaxws:bindings** 요소를 **WSDL** 계약에서 직접 포함시켜 **WSDL** 확장 프로그램으로 처리합니다. 이 경우 **jaxws:bindings** 의 설정은 바로 부모 요소에만 적용됩니다.

40.3.2. 외부 바인딩 선언 사용

비동기 호출에서 전환하는 바인딩 선언 파일에 대한 템플릿은 예 40.2. “비동기 바인딩 선언에 대한 템플릿” 에 표시됩니다.

예 40.2. 비동기 바인딩 선언에 대한 템플릿

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          wsdlLocation="AffectedWSDL"
          xmlns="http://java.sun.com/xml/ns/jaxws">
  <bindings node="AffectedNode">
    <enableAsyncMapping>true</enableAsyncMapping>
  </bindings>
</bindings>
```

여기서 **AffectedWSDL** 은 이 바인딩 선언의 영향을 받는 **WSDL** 계약의 **URL**을 지정합니다. **AffectedNode** 는 **WSDL** 계약의 노드(또는 노드)가 이 바인딩 선언의 영향을 받는 노드(또는 노드)를 지정하는 **XPath** 값입니다. 전체 **WSDL** 계약을 사용하려면 **AffectedNode** 를 **wsdl:definitions** 로 설정할

수 있습니다. 비동기 호출 기능을 활성화하려면 **jaxws:enableAsyncMapping** 요소가 **true** 로 설정됩니다.

예를 들어 **GreeterAsync** 인터페이스에 대해서만 비동기 메서드를 생성하려는 경우 이전 바인딩 선언에서 **<bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']">**을 지정할 수 있습니다.

바인딩 선언이 파일에 저장된다고 가정하면 예 40.3. “소비자 코드 생성” 에 표시된 대로 **POM**을 설정합니다.

예 40.3. 소비자 코드 생성

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${cxf.version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>outputDir</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>hello_world.wsdl</wsdl>
            <extraargs>
              <extraarg>-client</extraarg>
              <extraarg>-b async_binding.xml</extraarg>
            </extraargs>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

b 옵션은 코드 생성기에 외부 바인딩 파일을 찾을 위치를 알려줍니다.

코드 생성기에 대한 자세한 내용은 44.2절. “cxf-codegen-plugin” 에서 참조하십시오.

40.3.3. 포함된 바인딩 선언 사용

jaxws:bindings 요소와 연결된 jaxws:enableAsynchMapping 하위를 WSDL에 직접 배치하여 서비스를 정의하는 WSDL 문서에 직접 바인딩 사용자 지정을 포함할 수도 있습니다. jaxws 접두사에 대한 네임스페이스 선언도 추가해야 합니다.

예 40.4. "비동기 매핑을 위한 임베디드 바인딩 선언이 있는 WSDL" 작업에 대한 비동기 매핑을 활성화하는 포함 바인딩 선언이 있는 WSDL 파일을 표시합니다.

예 40.4. 비동기 매핑을 위한 임베디드 바인딩 선언이 있는 WSDL

```
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
                  ...
                  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
                  ...>
  ...
  <wsdl:portType name="GreeterAsync">
    <wsdl:operation name="greetMeSometime">
      <jaxws:bindings> <jaxws:enableAsyncMapping>true</jaxws:enableAsyncMapping>
</jaxws:bindings>
      <wsdl:input name="greetMeSometimeRequest"
                  message="tns:greetMeSometimeRequest"/>
      <wsdl:output name="greetMeSometimeResponse"
                   message="tns:greetMeSometimeResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

WSDL 문서에 바인딩 선언을 포함할 때 선언의 영향을 받는 범위를 제어할 수 있습니다.When including the binding declaration into the WSDL document, you can control the scope of the declaration by changing where you place the declaration. 선언이 wsdl:definitions 요소의 자식으로 배치되면 코드 생성기는 WSDL 문서에 정의된 모든 작업에 대해 비동기 메서드를 만듭니다. 이 요소가 wsdl:portType 요소의 자식으로 배치되면 코드 생성기는 인터페이스에 정의된 모든 작업에 대해 비동기 메서드를 만듭니다.If it is placed as a child of a wsdl:portType element the code generator creates asynchronous methods for all of the operations defined in the interface. 이 요소가 wsdl:operation 요소의 자식으로 배치되면 코드 생성기는 해당 작업에 대해서만 비동기 메서드를 만듭니다.If it is placed as a child of a wsdl:operation element the code generator creates asynchronous methods for only that operation.

포함된 선언을 사용할 때 코드 생성기에 특별한 옵션을 전달할 필요가 없습니다.It is not necessary to pass any special options to the code generator when using embedded declarations. 코드 생성기가 이를 인식하고 그에 따라 조치를 취할 것입니다.

40.3.4. 생성된 인터페이스

이러한 방식으로 스텁 코드를 생성한 후 GreeterAsync SEI(파일 GreeterAsync.java)는 예 40.5.

"Asynchronous Invocations를 위한 메서드의 서비스 엔드포인트 인터페이스" 와 같이 정의됩니다.

예 40.5. Asynchronous Invocations를 위한 메서드의 서비스 엔드포인트 인터페이스

```
package org.apache.hello_world_async_soap_http;

import org.apache.hello_world_async_soap_http.types.GreetMeSometimeResponse;
...

public interface GreeterAsync
{
  public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<GreetMeSometimeResponse> asyncHandler
    );

    public Response<GreetMeSometimeResponse> greetMeSometimeAsync(
        java.lang.String requestType
    );

    public java.lang.String greetMeSometime(
        java.lang.String requestType
    );
}
```

일반적인 동기 방법 외에도 **greetMeSometime()** 은 **greetMeSometime** 작업에 대해 두 가지 비동기 메서드도 생성됩니다.

- 콜백 접근 방법 **public future<?>greetMeSomtimeAsyncjava.lang.StringrequestTypeAsyncHandler<GreetMeSomtimeResponse>asyncHandler**
- 폴링 접근 방식 공개응답 **<GreetMeSomeTimeResponse>greetMeSometimeAsyncjava.lang.StringrequestType**

40.4. POLLING APPROACH를 사용하여 비동기 클라이언트 구현

40.4.1. 개요

폴링 접근 방식은 비동기 애플리케이션 개발을 위한 두 가지 방법 중 가장 간단한 방법입니다. 클라이언트는 **OperationNameAsync()** 라는 비동기 메서드를 호출하고 응답에 대해 폴링하는 **Response<T >**

개체를 반환합니다. 응답을 기다리는 동안 클라이언트가 수행하는 작업은 애플리케이션의 요구 사항에 따라 달라집니다. 폴링을 처리하기 위한 두 가지 기본 패턴이 있습니다.

- 비차단 폴링 - 비차단 응답<T>.isDone() 메서드를 호출하여 결과가 준비되었는지 여부를 주기적으로 확인합니다. 결과가 준비되면 클라이언트가 이를 처리합니다. 그렇지 않은 경우 고객은 다른 작업을 계속합니다.

- 폴링 차단- 바로 Response<T>.get() 을 호출하고 응답이 도달할 때까지 차단됩니다(선택적으로 시간 초과 지정).

40.4.2. non-blocking 패턴 사용

예 40.6. "비동기 작업 호출에 대한 블록되지 않은 Polling Approach" 에서는 예 40.1. "비동기에 대한 WSDL 계약 예" 정의된 greetMeSometime 작업에서 비동기 호출을 수행하기 위해 비차단 폴링을 사용하는 방법을 보여줍니다. 클라이언트는 비동기 작업을 호출하고 주기적으로 검사하여 결과가 반환되는지 확인합니다.

예 40.6. 비동기 작업 호출에 대한 블록되지 않은 Polling Approach

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));

      while (!greetMeSomeTimeResp.isDone()) {
      // client does some work
      }
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
```

```
        System.exit(0);
    }
}
```

예 40.6. "비동기 작업 호출에 대한 블록되지 않은 Polling Approach" 의 코드는 다음을 수행합니다.

프록시에서 **greetMeSometimeAsync()** 를 호출합니다.

메서드 호출은 **Response<GreetMeSometimeResponse >** 개체를 클라이언트에 즉시 반환합니다. **Apache CXF** 런타임은 원격 끝점에서 응답을 수신하고 **Response< GreetMeSometimeResponse >** 오브젝트를 채우는 세부 정보를 처리합니다.

참고

런타임은 원격 끝점의 **greetMeSometime()** 메서드로 요청을 전송하고 호출의 비동기 특성 세부 정보를 투명하게 처리합니다. 엔드 포인트, 따라서 서비스 구현에서는 클라이언트가 응답을 기다리는 방법에 대한 세부 정보를 염려하지 않습니다.

반환된 **Response** 오브젝트의 **isDone()** 를 확인하여 응답이 도착했는지 확인합니다.

응답이 도착하지 않은 경우 클라이언트는 다시 확인하기 전에 계속 작업을 진행합니다.

응답이 도착하면 클라이언트는 **get()** 메서드를 사용하여 **Response** 개체에서 이를 검색합니다.

40.4.3. 차단 패턴 사용

블록 폴링 패턴을 사용하는 경우 **Response** 오브젝트의 **isDone()** 은 절대 호출되지 않습니다. 대신 **Response** 오브젝트의 **get()** 메서드는 원격 작업을 호출한 직후에 호출됩니다. 응답을 사용할 수 있을 때까지 **get()** 가 차단됩니다.

또한 **get()** 메서드에 시간 제한을 전달할 수도 있습니다.

예 40.7. "비동기 작동 호출을 위한 Polling Approach 차단" 는 차단 폴링을 사용하는 클라이언트를 보여줍니다.

*예 40.7. 비동기 작동 호출을 위한 **Polling Approach** 차단*

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception {

    // set up the proxy for the client

    Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
      System.exit(0);
  }
}
```

40.5. CALLBACK APPROACH를 사용하여 비동기 클라이언트 구현

40.5.1. 개요

비동기 작업 호출을 수행하는 다른 방법은 콜백 클래스를 구현하는 것입니다. 그런 다음 콜백 오브젝트를 매개 변수로 사용하는 비동기 원격 메서드를 호출합니다. 런타임은 콜백 오브젝트에 대한 응답을 반환합니다.

콜백을 사용하는 애플리케이션을 구현하려면 다음을 수행합니다.

1. **AsyncHandler** 인터페이스를 구현하는 콜백 클래스를 생성합니다.

참고

콜백 오브젝트는 애플리케이션에 필요한 모든 응답 처리를 수행할 수 있습니다.

2. 콜백 개체를 매개 변수로 사용하고 **future< ?>** 개체를 반환하는 **operationNameAsync()** 를 사용하여 원격 호출을 만듭니다.

3. 클라이언트가 응답 데이터에 액세스해야 하는 경우 반환된 **future< ? > isDone()** 메서드를 폴링하여 원격 끝점이 응답을 전송했는지 확인할 수 있습니다.

콜백 개체가 모든 응답 처리를 수행하는 경우 응답이 도착했는지 확인할 필요가 없습니다.

40.5.2. 콜백 구현

콜백 클래스는 **javax.xml.ws.AsyncHandler** 인터페이스를 구현해야 합니다. 인터페이스는 단일 방법을 정의합니다. **handleResponseResponse<T>**는 **Apache CXF** 런타임에서 **handleResponse()** 메서드를 호출하여 응답이 도착했음을 알립니다. 예 40.8. “javax.xml.ws.AsyncHandler 인터페이스” 구현 해야 하는 **AsyncHandler** 인터페이스의 개요를 보여 줍니다.**Shows an outline of the AsyncHandler interface that you must implement.**

예 40.8. javax.xml.ws.AsyncHandler 인터페이스

```
public interface javax.xml.ws.AsyncHandler
{
  void handleResponse(Response<T> res)
}
```

예 40.9. “콜백 구현 클래스” 예 40.1. “비동기에 대한 WSDL 계약 예” 에 정의된 **greetMeSometime** 작업의 콜백 클래스를 표시합니다.

예 40.9. 콜백 구현 클래스

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
```

```
import org.apache.hello_world_async_soap_http.types.*;

public class GreeterAsyncHandler implements AsyncHandler<GreetMeSometimeResponse>
{
  private GreetMeSometimeResponse reply;

  public void handleResponse(Response<GreetMeSometimeResponse>
                             response)
  {
    try
    {
      reply = response.get();
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }

  public String getResponse()
  {
    return reply.getResponseType();
  }
}
```

예 40.9. "콜백 구현 클래스"에 표시된 콜백 구현은 다음을 수행합니다.

*원격 끝점에서 반환된 응답 을 보유하는 멤버 변수 **response**를 정의합니다.*

***handleResponse()** 를 구현합니다.*

이 구현에서는 단순히 응답을 추출하여 멤버 변수 회신 에 할당합니다.

***getResponse()** 라는 추가 메서드를 구현합니다.*

이 방법은 응답 에서 데이터를 추출하고 반환하는 편의 방법입니다.

40.5.3. 소비자 구현

*예 40.10. "비동기 작업 호출을 위한 콜백 접근 방식" 다음 예제에서는 콜백 접근 방식을 사용하여 예 40.1. "비동기에 대한 WSDL 계약 예"에 정의된 **GreetMeSometime** 작업에 대한 비동기 호출을 만드*

는 클라이언트를 보여줍니다.

예 40.10. 비동기 작업 호출을 위한 콜백 접근 방식

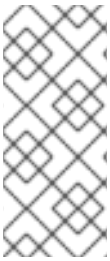
```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client {
  ...

  public static void main(String args[]) throws Exception
  {
    ...
    // Callback approach
    GreeterAsyncHandler callback = new GreeterAsyncHandler();

    Future<?> response =
      port.greetMeSometimeAsync(System.getProperty("user.name"),
                                callback);
    while (!response.isDone())
    {
      // Do some work
    }
    resp = callback.getResponse();
    ...
    System.exit(0);
  }
}
```

예 40.10. "비동기 작업 호출을 위한 콜백 접근 방식" 의 코드는 다음을 수행합니다.

콜백 오브젝트를 인스턴스화합니다.

프록시에서 콜백 오브젝트를 사용하는 **greetMeSometimeAsync()** 를 호출합니다.

method 호출은 **future< ?>** 개체를 즉시 클라이언트에 반환합니다. **Apache CXF** 런타임은 원격 끝점에서 응답을 수신하는 세부 정보를 처리하고 콜백 오브젝트의 **handleResponse()** 메서드를 호출하고 **Response< GreetMeSometimeResponse >** 오브젝트를 채웁니다.



참고

런타임은 원격 끝점의 **greetMeSometime()** 메서드로 요청을 전송하고 원격 끝점의 지식 없이 호출의 비동기 특성에 대한 세부 정보를 처리합니다. 따라서 서비스 구현 엔드포인트는 클라이언트가 응답을 기다리는 방법에 대해 걱정할 필요가 없습니다.

반환된 **future< ? >** 개체의 **isDone()** 메서드를 사용하여 응답이 원격 끝점에서 도착했는지 확인합니다.

콜백 오브젝트의 **getResponse()** 메서드를 호출하여 응답 데이터를 가져옵니다.

40.6. 원격 서비스에서 반환한 예외 CATCHING EXCEPTIONS RETURNED FROM A REMOTE SERVICE

40.6.1. 개요

비동기 요청을 수행하는 소비자는 동기 요청을 할 때 반환된 것과 동일한 예외를 받지 않습니다. 소비자에게 비동기적으로 반환되는 모든 예외는 **ExecutionException** 예외로 래핑됩니다. 서비스에서 **throw**한 실제 예외는 **ExecutionException** 예외의 **cause** 필드에 저장됩니다.

40.6.2. 예외를 포착합니다.

원격 서비스에서 생성한 예외는 소비자의 비즈니스 논리에 응답을 전달하는 메서드에서 로컬로 **throw**됩니다. 소비자가 동기 요청을 하면 원격 호출을 수행하는 메서드에서 예외가 발생합니다. 소비자가 비동기 요청을 수행할 때 **Response<T>** 개체의 **get()** 메서드는 예외를 **throw**합니다. 소비자는 응답 메시지 검색을 시도할 때까지 요청을 처리하는 동안 오류가 발생했음을 감지하지 않습니다.

JAX-WS 프레임워크에서 생성된 메서드와 달리 **Response<T>** 개체의 **get()** 메서드는 사용자 모델링 예외 및 일반 **JAX-WS** 예외가 발생하지 않습니다. 대신 **java.util.concurrent.ExecutionException** 예외가 **throw**됩니다.

40.6.3. 예외 세부 정보 가져오기

프레임워크는 원격 서비스에서 반환된 예외를 **ExecutionException** 예외의 원인 필드에 저장합니다. 원인 필드의 값을 가져오고 저장된 예외를 검사하여 원격 예외에 대한 세부 정보가 추출됩니다. 저장된 예외는 사용자 정의 예외 또는 일반 **JAX-WS** 예외 중 하나일 수 있습니다.

40.6.4. 예제

예 40.11. "Polling Approach을 사용하여 예외 관리" 은 폴링 방식을 사용하여 예외를 catch하는 예를 보여줍니다.

예 40.11. Polling Approach을 사용하여 예외 관리

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.apache.hello_world_async_soap_http.*;

public final class Client
{
  private static final QName SERVICE_NAME
    = new QName("http://apache.org/hello_world_async_soap_http",
                "SOAPService");

  private Client() {}

  public static void main(String args[]) throws Exception
  {
    ...
    // port is a previously established proxy object.
    Response<GreetMeSometimeResponse> resp =
      port.greetMeSometimeAsync(System.getProperty("user.name"));

    while (!resp.isDone())
    {
      // client does some work
    }

    try
    {
      GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
      // process the response
    }
    catch (ExecutionException ee)
    {
       Throwable cause = ee.getCause();
       System.out.println("Exception "+cause.getClass().getName()+" thrown by the remote
service.");
    }
  }
}
```

예 40.11. "Polling Approach을 사용하여 예외 관리" 의 코드는 다음을 수행합니다.

Response<T> 개체의 get() 메서드로 호출을 try/catch 블록으로 래핑합니다.

ExecutionException 예외를 catch합니다.

예외에서 cause 필드를 추출합니다.

소비자가 콜백 접근 방식을 사용하는 경우 예외를 catch하는 데 사용되는 코드가 서비스 응답을 추출하는 콜백 오브젝트에 배치됩니다.

41장. 원시 XML 메시지 사용

초록

상위 수준 JAX-WS API는 데이터를 JAXB 오브젝트로 마샬링하여 개발자가 네이티브 XML 메시지를 사용하지 못하도록 보호합니다. 그러나 전선을 통과하는 원시 XML 메시지 데이터에 직접 액세스하는 것이 더 나은 경우가 있습니다. JAX-WS API는 원시 XML에 대한 액세스를 제공하는 두 개의 인터페이스를 제공합니다. Dispatch 인터페이스는 클라이언트 측 인터페이스이고 공급자 인터페이스는 서버 측 인터페이스입니다.

41.1. 소비자에서 XML 사용

초록

Dispatch 인터페이스는 원시 메시지에서 직접 작업할 수 있는 하위 수준 JAX-WS API입니다. DOM 개체, SOAP 메시지 및 JAXB 개체를 포함한 여러 유형의 메시지를 수락하고 반환합니다. 낮은 수준의 API이므로 Dispatch 인터페이스는 더 높은 수준의 JAX-WS API가 수행하는 메시지 준비를 수행하지 않습니다. Dispatch 오브젝트에 전달하는 메시지 또는 페이로드가 제대로 구성되어 있고 원격 작업이 호출되는지 확인해야 합니다.

41.1.1. 사용량 모드

41.1.1.1. 개요

디스패치 오브젝트에는 두 가지 사용 모드 가 있습니다.

- 메시지 모드
- 메시지 페이로드 모드(Payload 모드)

Dispatch 오브젝트에 지정하는 사용 모드는 사용자 수준 코드로 전달되는 세부 정보 양을 결정합니다.

41.1.1.2. 메시지 모드

메시지 모드에서 Dispatch 오브젝트는 전체 메시지와 함께 작동합니다. 전체 메시지에는 바인딩 특정 헤더 및 래퍼가 포함됩니다. 예를 들어 SOAP 메시지가 필요한 서비스와 상호 작용하는 소비자는 Dispatch 오브젝트의 invoke() 메서드를 완전히 지정된 SOAP 메시지를 제공해야 합니다. invoke() 메서

드도 완전히 지정된 SOAP 메시지를 반환합니다. 소비자 코드는 SOAP 메시지의 헤더와 SOAP 메시지의 봉투 정보를 작성하고 읽습니다.

JAXB 오브젝트로 작업할 때 메시지 모드가 적합하지 않습니다.

Dispatch 오브젝트가 메시지 모드를 사용하도록 지정하려면 Dispatch 오브젝트를 생성할 때 java.xml.ws.Service.Mode.MESSAGE 값을 제공합니다. Dispatch 오브젝트 생성에 대한 자세한 내용은 "Dispatch 오브젝트 생성" 을 참조하십시오.

41.1.1.3. 페이로드 모드

페이로드 모드에서 는 메시지 페이로드 모드라고도 하며 Dispatch 오브젝트는 메시지의 페이로드에서만 작동합니다. 예를 들어 페이로드 모드에서 작동하는 Dispatch 오브젝트는 SOAP 메시지의 본문에서만 작동합니다. 바인딩 계층은 바인딩 수준 래퍼 및 헤더를 처리합니다. result가 invoke() 메서드에서 반환되면 바인딩 수준 래퍼 및 헤더가 이미 제거되며 메시지의 본문만 남아 있습니다.

Apache CXF XML 바인딩, 페이로드 모드 및 메시지 모드와 같은 특수 래퍼를 사용하지 않는 바인딩으로 작업하는 경우 동일한 결과를 제공합니다.

Dispatch 오브젝트가 페이로드 모드를 사용하도록 지정하려면 Dispatch 오브젝트를 생성할 때 java.xml.ws.Service.Mode.PAYLOAD 값을 제공합니다. Dispatch 오브젝트 생성에 대한 자세한 내용은 "Dispatch 오브젝트 생성" 을 참조하십시오.

41.1.2. 데이터 유형

41.1.2.1. 개요

Dispatch 개체는 하위 수준 오브젝트이므로 높은 수준의 소비자 API로 동일한 JAXB 생성 유형을 사용하도록 최적화되지 않습니다. 디스패치 오브젝트는 다음 유형의 오브젝트를 사용하여 작동합니다.

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource

- *"JAXB 오브젝트 사용"*

41.1.2.2. 소스 오브젝트 사용

Dispatch 개체는 javax.xml.transform.Source 인터페이스에서 파생된 개체를 수락하고 반환합니다. 소스 오브젝트는 바인딩 및 메시지 모드 또는 페이로드 모드에서 지원됩니다.

소스 개체는 XML 문서를 보유하는 낮은 수준의 개체입니다.Source objects are low level objects that hold XML documents. 각 소스 구현은 저장된 XML 문서에 액세스한 다음 해당 콘텐츠를 조작하는 메서드를 제공합니다. 다음 오브젝트는 소스 인터페이스를 구현합니다.

DOMSource

XML 메시지를 Document Object Model(DOM) 트리로 보관합니다. XML 메시지는 getNode() 메서드를 사용하여 액세스하는 Node 오브젝트 세트로 저장됩니다. 노드는 setNode() 메서드를 사용하여 DOM 트리에 업데이트되거나 추가될 수 있습니다.

saXSource

XML 메시지를SAX(Simple API for XML) 개체로 보유하고 있습니다. SAX 오브젝트에는 원시 데이터와 원시 데이터를 구문 분석하는 XMLReader 개체를 보유하는 InputSource 개체가 포함되어 있습니다.

StreamSource

XML 메시지를 데이터 스트림으로 보관합니다. 데이터 스트림은 다른 데이터 스트림과 동일하게 조작될 수 있습니다.

Dispatch 개체를 만들어 일반 소스 개체를 사용하도록 하는 경우 Apache CXF는 메시지를 SAXSource 개체로 반환합니다.

이 동작은 끝점의 source-preferred-format 속성을 사용하여 변경할 수 있습니다. Apache CXF 런타임 구성에 대한 자세한 내용은 IV 부. 웹 서비스 엔드 포인트 구성 을 참조하십시오.

41.1.2.3. SOAPMessage 오브젝트 사용

디스패치 오브젝트는 다음 조건이 true인 경우 javax.xml.soap.SOAPMessage 오브젝트를 사용할 수 있습니다.

- *Dispatch 개체는 SOAP 바인딩을 사용 합니다.The Dispatch object is using the SOAP*

binding

- *Dispatch 개체는 메시지 모드를 사용합니다.*

SOAPMessage 오브젝트에는 SOAP 메시지가 있습니다. 하나의 SOAPPart 오브젝트와 1개 이상의 AttachmentPart 오브젝트가 포함됩니다. SOAPPart 오브젝트에는 SOAP 봉투, 모든 SOAP 헤더 및 SOAP 메시지 본문을 포함한 SOAP 메시지의 SOAP 특정 부분이 포함되어 있습니다. AttachmentPart 오브젝트에는 첨부 파일로 전달되는 바이너리 데이터가 포함되어 있습니다.

41.1.2.4. DataSource 개체 사용

디스패치 오브젝트는 다음 조건이 true인 경우 javax.activation.DataSource 인터페이스를 구현하는 개체를 사용할 수 있습니다.

- *Dispatch 오브젝트는 HTTP 바인딩을 사용하고 있습니다.*
- *Dispatch 개체는 메시지 모드를 사용합니다.*

데이터 소스 오브젝트는 URL, 파일, 바이트 배열을 포함하여 다양한 소스의 MIME 형식 데이터를 사용하기 위한 메커니즘을 제공합니다.

41.1.2.5. JAXB 오브젝트 사용

Dispatch 개체는 원시 메시지로 작업할 수 있는 낮은 수준의 API를 대상으로 하지만 JAXB 개체로 작업할 수도 있습니다. JAXB 개체를 사용하려면 Dispatch 오브젝트가 사용 중인 JAXB 오브젝트를 마샬링하고 마샬링 해제할 수 있는 JAXBContext 를 전달해야 합니다. Dispatch 오브젝트를 만들 때 JAXBContext 가 전달됩니다.

JAXBContext 개체에서 설명하는 모든 JAXB 오브젝트를 invoke() 메서드에 매개 변수로 전달할 수 있습니다. 반환된 메시지를 JAXBContext 개체에서 이해하는 모든 JAXB 개체로 캐스팅할 수도 있습니다.

JAXBContext 오브젝트 생성에 대한 자세한 내용은 39장. A JAXBContext 오브젝트 사용 을 참조하십시오.

41.1.3. Dispatch Objects 사용

41.1.3.1. 절차

Dispatch 오브젝트를 사용하여 원격 서비스를 호출하려면 다음 시퀀스를 따라야 합니다.

1. **Dispatch** 오브젝트를 생성합니다.
2. 요청 메시지를 구성합니다. **???**
3. 적절한 **invoke()** 메서드를 호출합니다.
4. 응답 메시지를 구문 분석합니다.

41.1.3.2. Dispatch 오브젝트 생성

Dispatch 오브젝트를 생성하려면 다음을 수행합니다.

1. **Dispatch** 오브젝트가 호출될 서비스 를 정의하는 **wsdl:service** 요소를 나타내는 **Service** 오브젝트를 생성합니다. **25.2절. “서비스 오브젝트 생성”** 을 참조하십시오.
2. **예 41.1. “createDispatch() 메서드”** 에 표시된 **Service** 오브젝트의 **createDispatch()** 메서드를 사용하여 **Dispatch** 오브젝트를 생성합니다.

예 41.1. createDispatch() 메서드

```
publicDispatch<T>createDispatchQNameportNamejava.lang.Class<T>typeService.ModemodeWebServiceException
```

참고

JAXB 개체를 사용하는 경우 **createDispatch()** 의 메서드 서명이 있습니다. **publicDispatch<T>createDispatchQNameportNamejavax.xml.bind.JAXBContext**컨텍스트**Service.ModemodeWebServiceException**

표 41.1. "createDispatch()매개변수" createDispatch() 메서드의 매개변수를 설명합니다.

표 41.1. createDispatch()매개변수

매개변수	설명
portName	Dispatch 오브젝트가 호출될 서비스 공급자를 나타내는 **wsdl:port** 요소의 QName을 지정합니다.
type	Dispatch 개체에서 사용하는 개체의 데이터 형식을 지정합니다.Specifies the data type of the objects used by the Dispatch object. 41.1.2절. "데이터 유형" 을 참조하십시오. JAXB 개체를 사용할 때 이 매개 변수는 JAXB 오브젝트를 마샬링 및 unmarshal하는 데 사용되는 **JAXBContext** 오브젝트를 지정합니다.
mode	Dispatch 개체의 사용 모드를 지정합니다.Specifies the usage mode for the Dispatch object. 41.1.1절. "사용량 모드" 을 참조하십시오.

예 41.2. "Dispatch 오브젝트 생성" 페이로드 모드에서 DOMSource 개체로 작동하는 Dispatch 개체를 만드는 코드를 보여 줍니다.Shows the code for creating a Dispatch object that works with DOMSource objects in payload mode.

예 41.2. Dispatch 오브젝트 생성

```
package com.fusesource.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client
{
public static void main(String args[])
 {
   QName serviceName = new QName("http://org.apache.cxf", "stockQuoteReporter");
   Service s = Service.create(serviceName);

   QName portName = new QName("http://org.apache.cxf", "stockQuoteReporterPort");
   Dispatch<DOMSource> dispatch = s.createDispatch(portName,
                                    DOMSource.class,
                                    Service.Mode.PAYLOAD);
   ...
```

41.1.3.3. 요청 메시지 구성

Dispatch 오브젝트를 사용할 때는 처음부터 요청을 빌드해야 합니다. 개발자는 **Dispatch** 오브젝트에 전달된 메시지가 대상 서비스 공급자가 처리할 수 있는 요청과 일치하는지 확인합니다. 이를 위해서는 서비스 공급자가 사용하는 메시지와 필요한 헤더 정보에 대한 정확한 지식이 필요합니다.

이 정보는 **WSDL** 문서 또는 메시지를 정의하는 **XML** 스키마 문서에서 제공할 수 있습니다. 서비스 제공 업체는 크게 다르지만 다음과 같은 몇 가지 지침을 따라야 합니다.

- 요청의 루트 요소는 호출되는 작업에 해당하는 **wsdl:operation** 요소의 **name** 속성 값을 기반으로 합니다.

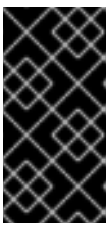
  

  주의

  호출 중인 서비스가 **doc/literal** 베어 메시지를 사용하는 경우 요청의 루트 요소는 **wsdl:operation** 요소에서 참조하는 **wsdl:part** 요소의 **name** 속성 값을 기반으로 합니다.

- 요청의 루트 요소는 네임스페이스 자격입니다.

- 호출 중인 서비스가 **rpc/literal** 메시지를 사용하는 경우 요청의 최상위 요소인 네임스페이스를 사용할 수 없습니다.

  중요

  최상위 수준 요소의 하위 항목은 네임스페이스를 정규화할 수 있습니다. 확인하려면 스키마 정의를 확인해야 합니다.

- 호출 중인 서비스가 **rpc/literal** 메시지를 사용하는 경우 최상위 요소 중 어느 것도 **null**일 수 없습니다.

- 호출 중인 서비스가 **doc/literal** 메시지를 사용하는 경우 메시지의 스키마 정의에서 요소에 네임스페이스가 있는지 여부를 결정합니다.

XML 메시지를 사용하는 서비스에 대한 자세한 내용은 WS-I 기본 프로필 를 참조하십시오.

41.1.3.4. 동기 호출

응답을 생성하는 동기 호출을 만드는 소비자의 경우 예 41.3. “Dispatch.invoke() 메서드” 에 표시된 Dispatch 오브젝트의 invoke() 메서드를 사용합니다.

예 41.3. Dispatch.invoke() 메서드

TinvokeTmsgWebServiceException

호출() 메서드에 전달된 응답 및 요청의 유형은 둘 다 Dispatch 오브젝트를 생성할 때 결정됩니다. 예를 들어 createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE) 를 사용하여 Dispatch 오브젝트를 생성하는 경우 응답 및 요청 모두 SOAPMessage 오브젝트입니다.

참고

JAXB 개체를 사용하는 경우 응답 및 요청은 제공된 JAXBContext 개체가 마샬링 및 해상할 수 있는 모든 유형일 수 있습니다. 또한 응답 및 요청은 다른 JAXB 오브젝트일 수 있습니다.

예 41.4. “Dispatch 오브젝트를 사용하여 Synchronous Invocation 만들기” DOMSource 개체를 사용하여 원격 서비스에서 동기 호출을 수행하는 코드를 보여 줍니다.Shows code for making a synchronous invocation on a remote service using a DOMSource object.

예 41.4. Dispatch 오브젝트를 사용하여 Synchronous Invocation 만들기

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document requestDoc = db.newDocument();
Element root = requestDoc.createElementNS("http://org.apache.cxf/stockExample",
                                           "getStockPrice");
root.setNodeValue("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

41.1.3.5. 비동기 호출

디스패치 오브젝트도 비동기 호출을 지원합니다. 40장. 비동기 애플리케이션 개발 에서 설명한 상위 수준 비동기 API와 마찬가지로 Dispatch 오브젝트는 폴링 접근 방식과 콜백 접근 방식을 모두 사용할 수 있습니다.

폴링 접근 방식을 사용할 때 invokeAsync() 메서드는 응답이 도착했는지 확인하기 위해 폴링할 수 있는 Response<t > 오브젝트를 반환합니다. 예 41.5. “Polling에 대한 Dispatch.invokeAsync() 메서드” 폴링 방식을 사용하여 비동기 호출을 수행하는 데 사용되는 메서드의 서명을 표시합니다.

예 41.5. Polling에 대한 Dispatch.invokeAsync() 메서드

```
Response <T>invokeAsyncTmsgWebServiceException
```

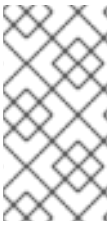
비동기 호출에 대한 폴링 접근 방식을 사용하는 방법에 대한 자세한 내용은 40.4절. “Polling Approach를 사용하여 비동기 클라이언트 구현” 을 참조하십시오.

콜백 방법을 사용할 때 invokeAsync() 메서드는 응답이 반환될 때 AsyncHandler 구현을 사용합니다. 예 41.6. “Dispatch.invokeAsync() 메서드 사용” 콜백 접근 방식을 사용하여 비동기 호출을 만드는 데 사용되는 메서드의 서명을 표시합니다.Shows the signature of the method used to make an asynchronous invocation using the callback approach.

예 41.6. Dispatch.invokeAsync() 메서드 사용

```
future<?>invokeAsyncTmsgAsyncHandler<T>처리기WebServiceException
```

비동기 호출에 대한 콜백 접근 방식을 사용하는 방법에 대한 자세한 내용은 40.5절. “Callback Approach를 사용하여 비동기 클라이언트 구현” 을 참조하십시오.



참고

동기 invoke() 메서드와 마찬가지로 응답 유형 및 요청 유형은 Dispatch 오브젝트를 생성할 때 결정됩니다.

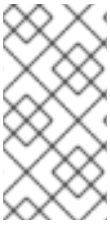
41.1.3.6. OneWay 호출

요청이 응답을 생성하지 않으면 Dispatch 오브젝트의 invokeOneWay() 를 사용하여 원격 호출을 수행합니다. 예 41.7. “The Dispatch.invokeOneWay() Method” 이 메서드의 서명을 표시합니다.

예 41.7. The Dispatch.invokeOneWay() Method

invokeOneWayTmsgWebServiceException

요청을 패키징하는 데 사용되는 개체의 유형은 **Dispatch** 오브젝트를 생성할 때 결정됩니다. 예를 들어, **Dispatch** 개체가 **createDispatch(portName, DOMSource.class, Service.Mode.PAYLOAD)** 를 사용하여 생성된 경우 요청은 **DOMSource** 개체에 패키지됩니다.



참고

JAXB 개체를 사용하는 경우 응답 및 요청은 제공된 **JAXBContext** 개체가 마샬링 및 **unmarshal**할 수 있는 모든 유형일 수 있습니다.

예 41.8. "Dispatch Object를 사용하여 One way Invocation을 만드는 방법" **JAXB** 오브젝트를 사용하여 원격 서비스에서 단방향 호출을 수행하는 코드를 보여줍니다.

예 41.8. Dispatch Object를 사용하여 One way Invocation을 만드는 방법

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc = JAXBContext.newInstance("org.apache.cxf.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);

// Dispatch disp created previously
disp.invokeOneWay(request);
```

41.2. 서비스 공급자의 XML 사용

초록

공급자 인터페이스는 메시지에서 직접 원시 **XML**로 직접 작동하는 서비스 공급자를 구현할 수 있는 하위 수준 **JAX-WS API**입니다. 메시지는 **Provider** 인터페이스를 구현하는 개체에 전달하기 전에 **JAXB** 개체에 패키징되지 않습니다.

41.2.1. 메시징 모드

41.2.1.1. 개요

공급자 인터페이스를 구현하는 오브젝트에는 두 가지 메시징 모드 가 있습니다.

- 메시지 모드
- 페이로드 모드

지정하는 메시징 모드는 구현에 전달되는 메시징 세부 수준을 결정합니다.

41.2.1.2. 메시지 모드

메시지 모드 를 사용할 때 공급자 구현은 완전한 메시지와 함께 작동합니다. 전체 메시지에는 바인딩 특정 헤더 및 래퍼가 포함됩니다. 예를 들어 **SOAP** 바인딩을 사용하는 공급자 구현은 완전히 지정된 **SOAP** 메시지로서 요청을 수신합니다. 구현에서 반환된 응답은 완전히 지정된 **SOAP** 메시지여야 합니다.

공급자 구현에서 예 41.9. "공급자 구현에서 메시지 모드 사용 지정" 와 같이 **java.xml.ws.Service.Mode.MESSAGE** 값을 **javax.xml.ws.ServiceMode** 주석에 대한 값으로 제공하여 메시지 모드를 사용하도록 지정하려면

예 41.9. 공급자 구현에서 메시지 모드 사용 지정

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

41.2.1.3. 페이로드 모드

페이로드 모드에서 공급자 구현은 메시지의 페이로드만 사용하여 작동합니다. 예를 들어 페이로드 모드에서 작동하는 공급자 구현은 **SOAP** 메시지의 본문에서만 작동합니다. 바인딩 계층은 바인딩 수준 래퍼 및 헤더를 처리합니다.

Apache CXF XML 바인딩, 페이로드 모드 및 메시지 모드와 같은 특수 래퍼를 사용하지 않는 바인딩으로 작업하는 경우 동일한 결과를 제공합니다.

공급자 구현에서 예 41.10. "공급자 구현에서 페이로드 모드를 사용하도록 지정" 와 같이 **java.xml.ws.Service.Mode.PAYLOAD** 값을 **javax.xml.ws.ServiceMode** 주석에 대한 값으로 제공하여 페이로드 모드를 사용하도록 지정하려면

예 41.10. 공급자 구현에서 페이로드 모드를 사용하도록 지정

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

@ServiceMode 주석 값을 지정하지 않으면 공급자 구현에서는 페이로드 모드를 사용합니다.

41.2.2. 데이터 유형

41.2.2.1. 개요

낮은 수준의 오브젝트이므로 공급자 구현에서는 상위 수준 소비자 **API**와 동일한 **JAXB** 생성 유형을 사용할 수 없습니다. 공급자 구현은 다음 유형의 오브젝트에서 작동합니다.

- **javax.xml.transform.Source**
- **javax.xml.soap.SOAPMessage**
- **javax.activation.DataSource**

41.2.2.2. 소스 오브젝트 사용

공급자 구현은 **javax.xml.transform.Source** 인터페이스에서 파생된 개체를 수락 및 반환할 수 있습니다. 소스 개체는 **XML** 문서를 보유하는 낮은 수준의 개체입니다.**Source objects are low level objects that hold XML documents.** 각 소스 구현은 저장된 **XML** 문서에 액세스하고 해당 콘텐츠를 조작하는 메서드를 제공합니다. 다음 오브젝트는 소스 인터페이스를 구현합니다.

DOMSource

XML 메시지를 **Document Object Model(DOM)** 트리로 보관합니다. **XML** 메시지는 **getNode()** 메서드를 사용하여 액세스하는 **Node** 오브젝트 세트로 저장됩니다. 노드는 **setNode()** 메서드를 사용

하여 DOM 트리에 업데이트되거나 추가될 수 있습니다.

saXSource

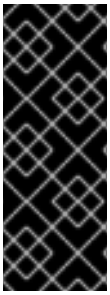
XML 메시지를SAX(Simple API for XML) 개체로 보유하고 있습니다. SAX 오브젝트에는 원시 데이터와 원시 데이터를 구문 분석하는 XMLReader 개체를 보유하는 InputSource 개체가 포함되어 있습니다.

StreamSource

XML 메시지를 데이터 스트림으로 보관합니다. 데이터 스트림은 다른 데이터 스트림과 동일하게 조작될 수 있습니다.

일반 소스 개체를 사용하도록 Provider 개체를 생성하는 경우 Apache CXF는 메시지를 SAXSource 개체로 반환합니다.

이 동작은 끝점의 source-preferred-format 속성을 사용하여 변경할 수 있습니다. Apache CXF 런타임 구성에 대한 자세한 내용은 IV 부. 웹 서비스 엔드 포인트 구성 을 참조하십시오.



중요

Source 개체를 사용할 때 개발자는 필요한 모든 바인딩 특정 래퍼가 메시지에 추가되도록 담당합니다. 예를 들어 SOAP 메시지가 필요한 서비스와 상호 작용할 때 개발자는 필요한 SOAP 봉투가 발신 요청에 추가되고 SOAP 봉투의 콘텐츠가 올바른지 확인해야 합니다.

41.2.2.3. SOAPMessage 오브젝트 사용

공급자 구현에서는 다음 조건이 true인 경우 javax.xml.soap.SOAPMessage 오브젝트를 사용할 수 있습니다.

- 공급자 구현에서는 SOAP 바인딩을 사용하고 있습니다.
- 공급자 구현에서는 메시지 모드를 사용합니다.

SOAPMessage 오브젝트에는 SOAP 메시지가 있습니다. 하나의 SOAPPart 오브젝트와 1개 이상의 AttachmentPart 오브젝트가 포함됩니다. SOAPPart 오브젝트에는 SOAP 봉투, 모든 SOAP 헤더 및 SOAP 메시지 본문을 포함한 SOAP 메시지의 SOAP 특정 부분이 포함되어 있습니다. AttachmentPart 오브젝트에는 첨부 파일로 전달되는 바이너리 데이터가 포함되어 있습니다.

41.2.2.4. DataSource 개체 사용

공급자 구현에서는 다음 조건이 **true**인 경우 **javax.activation.DataSource** 인터페이스를 구현하는 개체를 사용할 수 있습니다.

- 구현에서는 **HTTP** 바인딩 사용
- 구현은 메시지 모드를 사용합니다.

데이터 소스 오브젝트는 **URL**, 파일, 바이트 배열을 포함하여 다양한 소스의 **MIME** 형식 데이터를 사용하기 위한 메커니즘을 제공합니다.

41.2.3. 공급자 오브젝트 구현

41.2.3.1. 개요

공급자 인터페이스는 비교적 쉽게 구현할 수 있습니다. 구현되어야 하는 하나의 메서드인 **invoke()** 만 있습니다. 여기에는 세 가지 간단한 요구 사항이 있습니다.

- 구현에는 **@WebServiceProvider** 주석이 있어야 합니다.
- 구현에는 기본 **public** 생성자가 있어야 합니다.
- 구현에서는 입력한 공급자 버전의 공급자 인터페이스를 구현해야 합니다.

  즉, **Provider<T>** 인터페이스를 구현할 수 없습니다. 41.2.2절. “데이터 유형” 에 나열된 구체적인 데이터 유형을 사용하는 인터페이스 버전을 구현해야 합니다. 예를 들어 **Provider<SAXSource>** 인스턴스를 구현할 수 있습니다.

공급자 인터페이스 구현의 복잡성은 요청 메시지를 처리하고 적절한 응답을 빌드하는 논리에 있습니다.

41.2.3.2. 메시지 작업

상위 수준 **SEI** 기반 서비스 구현과 달리, 공급자 구현에서는 요청을 원시 **XML** 데이터로 수신하며 응답을 원시 **XML** 데이터로 보내야 합니다. 이를 위해서는 개발자가 구현 중인 서비스에서 사용하는 메시지에 대해 잘 알고 있어야 합니다. 이러한 세부 사항은 일반적으로 서비스를 설명하는 **WSDL** 문서에서 찾을 수 있습니다.

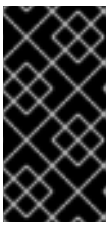
WS-I Basic Profile 은 다음을 포함하여 서비스에서 사용하는 메시지에 대한 지침을 제공합니다.

- 요청의 루트 요소는 호출되는 작업에 해당하는 **wsdl:operation** 요소의 **name** 속성 값을 기반으로 합니다.

주의

서비스에서 **doc/literal** 베어 메시지를 사용하는 경우 요청의 루트 요소는 **wsdl:operation** 요소에서 참조하는 **wsdl:part** 요소의 **name** 속성 값을 기반으로 합니다.

- 모든 메시지의 루트 요소는 네임스페이스 자격입니다.
- 서비스에서 **rpc/literal** 메시지를 사용하는 경우 메시지의 최상위 요소는 네임스페이스를 사용할 수 없습니다.



중요

최상위 수준의 요소 하위는 네임스페이스를 정규화할 수 있지만 스키마 정의를 확인해야 합니다.

- 서비스에서 **rpc/literal** 메시지를 사용하는 경우 최상위 요소 중 어느 것도 **null**일 수 없습니다.
- 서비스에서 **doc/literal** 메시지를 사용하는 경우 메시지의 스키마 정의는 요소 중 하나의 네임스페이스가 있는지 여부를 결정합니다.

41.2.3.3. @WebServiceProvider 주석

JAX-WS에서 서비스 구현으로 인식하려면 공급자 구현을 @WebServiceProvider 주석으로 장식해야 합니다.

표 41.2. "@WebServiceProvider Properties" @WebServiceProvider 주석에 대해 설정할 수 있는 속성을 설명합니다.

표 41.2. @WebServiceProvider Properties

속성	설명
portName	서비스의 엔드포인트를 정의하는 **wsdl:port** 요소의 **name** 속성 값을 지정합니다.
serviceName	서비스의 엔드포인트를 포함하는 **wsdl:service** 요소의 **name** 속성 값을 지정합니다.
targetNamespace	서비스 WSDL 정의의 targetname 공간을 지정합니다.
wsdlLocation	서비스를 정의하는 WSDL 문서의 URI를 지정합니다.

이러한 속성은 모두 선택 사항이며 기본적으로 비어 있습니다. 이를 비워 두면 Apache CXF는 구현 클래스의 정보를 사용하여 값을 생성합니다.

41.2.3.4. invoke() 메서드 구현

Provider 인터페이스에는 구현해야 하는 하나의 메서드인 invoke() 만 있습니다. invoke() 메서드는 구현되는 공급자 인터페이스 유형에서 선언된 오브젝트 유형으로 패키징된 들어오는 요청을 수신하고 동일한 유형의 개체에 패키지된 응답 메시지를 반환합니다. 예를 들어, Provider<SOAPMessage> 인터페이스의 구현은 요청을 SOAPMessage 오브젝트로 수신하고 response를 SOAPMessage 개체로 반환합니다.

공급자 구현에서 사용하는 메시징 모드는 요청 및 응답 메시지에 포함된 바인딩 특정 정보의 양을 결정합니다. 메시지 모드를 사용하는 구현에서는 요청과 함께 모든 바인딩 특정 래퍼 및 헤더를 수신합니다. 또한 모든 바인딩 특정 래퍼 및 헤더를 응답 메시지에 추가해야 합니다. 페이로드 모드를 사용하는 구현에서는 요청 본문만 받습니다. 페이로드 모드를 사용한 구현에서 반환된 XML 문서는 요청 메시지의 본문에 배치됩니다.

41.2.3.5. 예제

*예 41.11. "provider<SOAPMessage> 구현" 메시지 모드에서 **SOAPMessage** 개체와 함께 작동하는 **Provider** 구현을 보여줍니다.*

예 41.11. provider<SOAPMessage> 구현

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
                    serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.MESSAGE")
public class  stockQuoteReporterProvider implements Provider<SOAPMessage>
{
public stockQuoteReporterProvider()
  {
  }

public SOAPMessage invoke(SOAPMessage request)
  {
  SOAPBody requestBody = request.getSOAPBody();
  if(requestBody.getElementName.getLocalName.equals("getStockPrice"))
    {
    MessageFactory mf = MessageFactory.newInstance();
      SOAPFactory sf = SOAPFactory.newInstance();

    SOAPMessage response = mf.createMessage();
      SOAPBody respBody = response.getSOAPBody();
      Name bodyName = sf.createName("getStockPriceResponse");
      respBody.addBodyElement(bodyName);
      SOAPElement respContent = respBody.addChildElement("price");
      respContent.setValue("123.00");
      response.saveChanges();
    return response;
    }
    ...
  }
}
```

예 41.11. "provider<SOAPMessage> 구현" 의 코드는 다음을 수행합니다.

*다음 클래스는 **wsdl:service** 요소의 이름이 **stockQuoteReporter** 이고 **wsdl:port** 요소의 이름이 **stockQuoteReporterPort** 인 서비스를 구현하는 **Provider** 개체를 지정합니다.*

이 공급자 구현에서 메시지 모드를 사용하도록 지정합니다.

*필수 기본 **public** 생성자를 제공합니다.*

SOAPMessage** 개체를 사용하고 **SOAPMessage** 개체를 반환하는 **invoke()** 메서드의 구현을 제공합니다.**Provides an implementation of the invoke() method that takes a SOAPMessage object and returns a SOAPMessage object.

*들어오는 **SOAP** 메시지의 본문에서 요청 메시지를 추출합니다.*

요청 메시지의 루트 요소를 검사하여 요청을 처리하는 방법을 결정합니다.

응답을 빌드하는 데 필요한 공장을 생성합니다.

*응답에 대한 **SOAP** 메시지를 작성합니다.*

***response**를 **SOAPMessage** 오브젝트로 반환합니다.*

*예 41.12. “provider<DOMSource> 구현” 페이로드 모드에서 **DOMSource** 개체를 사용하는 공급자 구현의 예를 보여줍니다.*

예 41.12. provider<DOMSource> 구현

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements Provider<DOMSource>
public stockQuoteReporterProvider()
  {
  }

public DOMSource invoke(DOMSource request)
  {
    DOMSource response = new DOMSource();
    ...
```

```
    return response;
  }
}
```

예 41.12. “provider<DOMSource> 구현” 의 코드는 다음을 수행합니다.

*클래스에서 **wsdl:service** 요소의 이름이 **stockQuoteReporter** 이고 **wsdl:port** 요소의 이름이 **stockQuoteReporterPort** 인 서비스를 구현하는 **Provider** 개체를 지정합니다.*

이 공급자 구현에서 페이로드 모드를 사용하도록 지정합니다.

*필수 기본 **public** 생성자를 제공합니다.*

DOMSource** 개체를 가져와서 **DOMSource** 개체를 반환하는 **invoke()** 메서드의 구현을 제공합니다.**Provides an implementation of the invoke() method that takes a DOMSource object and returns a DOMSource object.

42장. 컨텍스트 작업

초록

JAX-WS는 컨텍스트를 사용하여 메시징 체인과 함께 메타데이터를 전달합니다. 이 메타데이터는 범위에 따라 구현 수준 코드에서 액세스할 수 있습니다. 구현 수준 아래의 메시지에서 작동하는 JAX-WS 핸들러에도 액세스할 수 있습니다.

42.1. 컨텍스트 이해

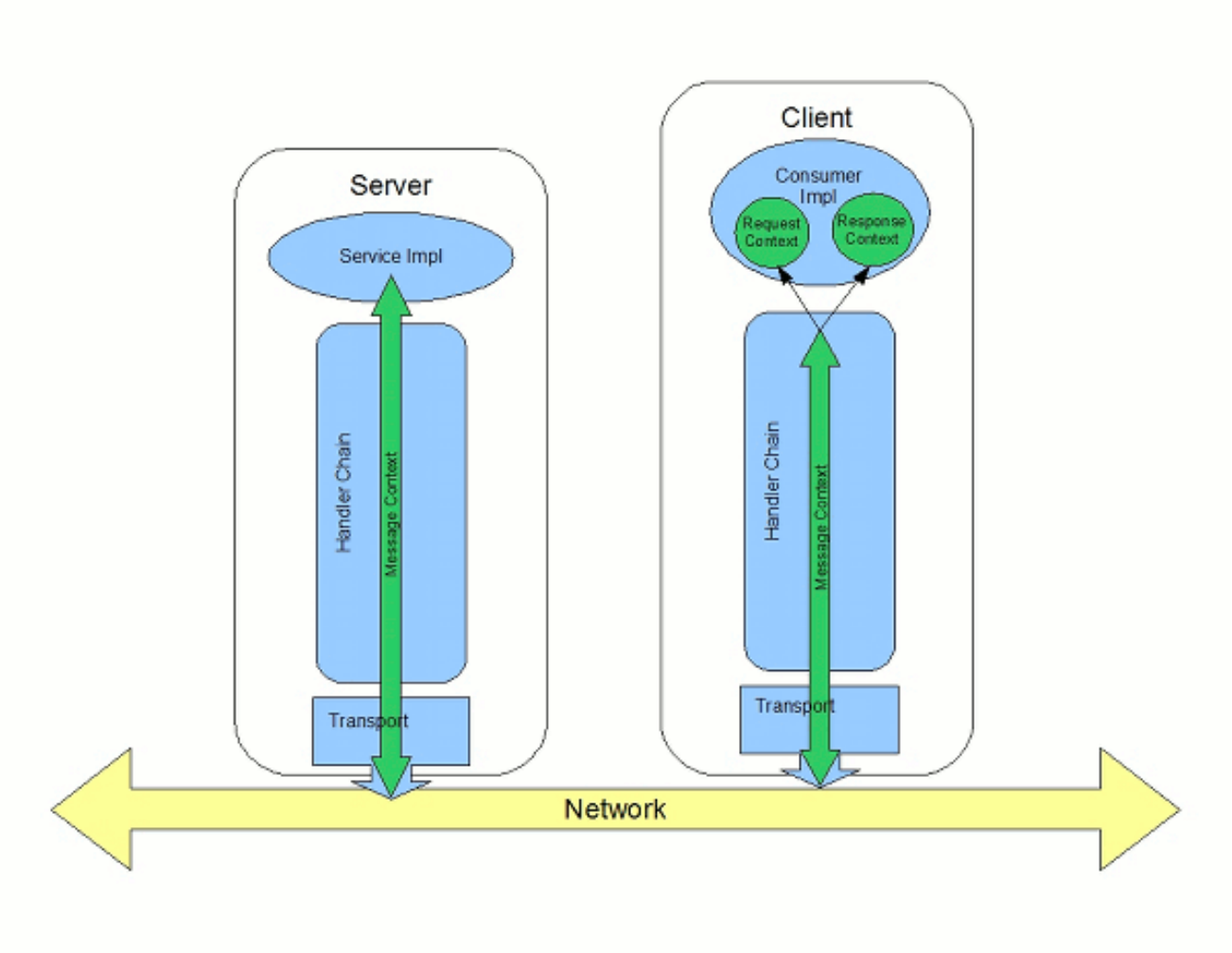
42.1.1. 개요

대부분의 경우 메시지에 대한 정보를 애플리케이션의 다른 부분에 전달해야 합니다. Apache CXF는 컨텍스트 메커니즘을 사용하여 이 작업을 수행합니다. 컨텍스트는 발신 또는 들어오는 메시지와 관련된 속성을 보유하는 맵입니다. 컨텍스트에 저장된 속성은 일반적으로 메시지에 대한 메타데이터이며 메시지를 전달하는 데 사용되는 기본 전송입니다. 예를 들어 HTTP 응답 코드 또는 JMS 상관 ID와 같은 메시지를 전송하는 데 사용되는 전송별 헤더는 JAX-WS 컨텍스트에 저장됩니다.

컨텍스트는 JAX-WS 애플리케이션의 모든 수준에서 사용할 수 있습니다. 그러나 메시지 처리 스택에서 컨텍스트에 액세스하는 위치에 따라 미묘한 방식으로 다릅니다. JAX-WS Handler 구현은 컨텍스트에 직접 액세스할 수 있으며 여기에 설정된 모든 속성에 액세스할 수 있습니다. 서비스 구현에서는 삽입된 컨텍스트를 사용하여 컨텍스트에 액세스하고 APPLICATION 범위에 설정된 속성만 액세스할 수 있습니다. 소비자 구현은 APPLICATION 범위에 설정된 속성만 액세스할 수 있습니다.

그림 42.1. “메시지 컨텍스트 및 메시지 처리 경로”는 컨텍스트 속성이 Apache CXF를 통과하는 방법을 보여줍니다. 메시지가 메시징 체인을 통과하면 관련 메시지 컨텍스트가 함께 전달됩니다.

그림 42.1. 메시지 컨텍스트 및 메시지 처리 경로



42.1.2. 속성을 컨텍스트에 저장하는 방법How properties are stored in a context

메시지 컨텍스트는 **javax.xml.ws.handler.MessageContext** 인터페이스의 모든 구현입니다. **MessageContext** 인터페이스는 **java.util.Map<String** 키, **Object value>** 인터페이스를 확장합니다. 맵 오브젝트는 키 값 쌍으로 정보를 저장합니다.

메시지 컨텍스트에서 속성은 이름/값 쌍으로 저장됩니다. 속성의 키는 속성을 식별하는 문자열 입니다. 속성 값은 **Java** 오브젝트에 저장된 모든 값일 수 있습니다. 메시지 컨텍스트에서 값을 반환하는 경우 애플리케이션에서 예상하고 캐스팅할 유형을 알아야 합니다. 예를 들어 속성 값이 **UserInfo** 개체에 저장된 경우 **UserInfo** 개체로 다시 캐스팅해야 하는 개체로 메시지 컨텍스트에서 계속 반환됩니다.**For example, if a property's value is stored in a UserInfo object it is still returned from a message context as an Object that must be cast back into a UserInfo object.**

메시지 컨텍스트의 속성도 범위가 있습니다. 범위는 메시지 처리 체인에서 속성에 액세스할 수 있는 위치를 결정합니다.

42.1.3. 속성 범위

메시지 컨텍스트의 속성은 범위가 지정됩니다. 속성은 다음 범위 중 하나일 수 있습니다.

애플리케이션

APPLICATION 으로 범위가 지정된 속성은 **JAX-WS Handler** 구현, 소비자 구현 코드 및 서비스 공급자 구현 코드에서 사용할 수 있습니다. 처리기가 서비스 공급자 구현에 속성을 전달해야 하는 경우 속성의 범위를 **APPLICATION**으로 설정합니다.**If a handler needs to pass a property to the service provider implementation, it sets the property's scope to APPLICATION.** 소비자 구현 또는 서비스 공급자 구현 컨텍스트에서 설정된 모든 속성은 자동으로 **APPLICATION** 으로 범위가 지정됩니다.

HANDLER

HANDLER 로 범위가 지정된 속성은 **JAX-WS Handler** 구현에서만 사용할 수 있습니다. 핸들러 구현에서 메시지 컨텍스트에 저장된 속성은 기본적으로 **HANDLER** 로 범위가 지정됩니다.

메시지 컨텍스트의 **setScope()** 메서드를 사용하여 속성 범위를 변경할 수 있습니다. 예 42.1. “The MessageContext.setScope() Method” 메서드의 서명을 표시합니다.

> **예 42.1. The MessageContext.setScope() Method**
>
> SetScopeStringkeyMessageContext.Scope범위java.lang.IllegalArgumentException

첫 번째 매개 변수는 속성의 키를 지정합니다. 두 번째 매개 변수는 속성의 새 범위를 지정합니다. 범위는 다음 중 하나일 수 있습니다.

- **MessageContext.Scope.APPLICATION**
- **MessageContext.Scope.HANDLER**

42.1.4. 핸들러의 컨텍스트 개요

JAX-WS Handler 인터페이스를 구현하는 클래스는 메시지의 컨텍스트 정보에 직접 액세스할 수 있습니다. 메시지의 컨텍스트 정보는 **Handler** 구현의 **handleMessage(), handleFault()** 및 **close()** 메서드로 전달됩니다.

핸들러 구현은 해당 범위에 관계없이 메시지 컨텍스트에 저장된 모든 속성에 액세스할 수 있습니다. 또한 논리 핸들러는 **LogicalMessageContext** 라는 특수 메시지 컨텍스트를 사용합니다.

LogicalMessageContext 개체에는 메시지 본문의 콘텐츠에 액세스하는 메서드가 있습니다.

42.1.5. 서비스 구현의 컨텍스트 개요

서비스 구현은 메시지 컨텍스트에서 APPLICATION 으로 범위가 지정된 속성에 액세스할 수 있습니다. 서비스 공급자의 구현 오브젝트는 WebServiceContext 오브젝트를 통해 메시지 컨텍스트에 액세스합니다.

자세한 내용은 42.2절. "서비스 구현에서 컨텍스트 작업" 에서 참조하십시오.

42.1.6. 소비자 구현의 컨텍스트 개요

소비자 구현은 메시지 컨텍스트의 콘텐츠에 대한 간접 액세스 권한을 갖습니다. 소비자 구현에는 두 개의 별도의 메시지 컨텍스트가 있습니다.

- *요청 컨텍스트 - 나가는 요청에 사용되는 속성 사본을 보유합니다.*
- *응답 컨텍스트 - 들어오는 응답에서 속성 사본을 보유합니다.*

디스패치 계층은 소비자 구현의 메시지 컨텍스트와 Handler 구현에서 사용하는 메시지 컨텍스트 간에 속성을 전송합니다.

소비자 구현에서 요청이 디스패치 계층에 전달되면 요청 컨텍스트의 콘텐츠가 디스패치 계층에서 사용하는 메시지 컨텍스트로 복사됩니다. 서비스에서 응답이 반환되면 디스패치 계층에서 메시지를 처리하고 적절한 속성을 메시지 컨텍스트로 설정합니다. 디스패치 계층에서 응답을 처리한 후 메시지 컨텍스트에서 APPLICATION 으로 범위가 지정된 모든 속성을 소비자 구현의 응답 컨텍스트에 복사합니다.

자세한 내용은 42.3절. "소비자 구현에서 컨텍스트 작업" 에서 참조하십시오.

42.2. 서비스 구현에서 컨텍스트 작업

42.2.1. 개요

컨텍스트 정보는 **WebServiceContext** 인터페이스를 사용하여 서비스 구현에 사용 가능하게 됩니다. **WebServiceContext** 개체에서 현재 요청의 컨텍스트 속성으로 채워지는 **MessageContext** 개체를 애플리케이션 범위에서 가져올 수 있습니다.**From the WebServiceContext object you can obtain a MessageContext object that is populated with the current request's context properties in the application scope.** 속성 값을 조작할 수 있으며 응답 체인을 통해 다시 전파됩니다.

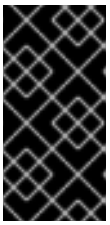
참고

MessageContext 인터페이스는 **java.util.Map** 인터페이스에서 상속됩니다. 해당 콘텐츠는 맵 인터페이스의 메서드를 사용하여 조작할 수 있습니다.

42.2.2. 컨텍스트 가져오기

서비스 구현에서 메시지 컨텍스트를 얻으려면 다음을 수행합니다.

1. **WebServiceContext** 유형의 변수를 선언합니다.
2. **javax.annotation.Resource** 주석으로 변수를 장식하여 컨텍스트 정보가 변수에 삽입됨을 나타냅니다.
3. **get MessageContext ()** 메서드를 사용하여 **WebServiceContext** 개체에서 **MessageContext** 개체를 가져옵니다.



중요

getMessageContext() 는 **@WebMethod** 주석으로 장식되는 메서드에서만 사용할 수 있습니다.

예 42.2. "서비스 구현에서 컨텍스트 오브젝트 가져오기" **context** 개체를 가져오기 위한 코드를 표시합니다.

예 42.2. 서비스 구현에서 컨텍스트 오브젝트 가져오기

```
import javax.xml.ws.*;
import javax.xml.ws.handler.*;
import javax.annotation.*;

@WebServiceProvider
```

```
public class WidgetServiceImpl
{
  @Resource
  WebServiceContext wsc;

  @WebMethod
  public String getColor(String itemNum)
  {
    MessageContext context = wsc.getMessageContext();
  }

  ...
}
```

42.2.3. 컨텍스트에서 속성을 읽습니다.Read a property from a context.

구현을 위한 **MessageContext** 개체를 얻은 후에는 예 42.3. “MessageContext.get() 메서드” 에 표시된 **get()** 메서드를 사용하여 여기에 저장된 속성에 액세스할 수 있습니다.

예 42.3. MessageContext.get() 메서드

vgetObject키

참고

이 **get()** 은 맵 인터페이스에서 상속됩니다.

key 매개변수는 컨텍스트에서 검색할 속성을 나타내는 문자열입니다. **get()** 는 속성에 대한 적절한 유형으로 캐스팅되어야 하는 개체를 반환합니다. 표 42.1. “서비스 구현 컨텍스트에서 사용 가능한 속성” 서비스 구현의 컨텍스트에서 사용할 수 있는 여러 속성을 나열합니다.**Lists a number of the properties that are available in a service implementation's context.**

중요

컨텍스트에서 반환된 개체의 값을 변경하면 컨텍스트의 속성 값도 변경됩니다.**Changing the values of the object returned from the context also changes the value of the property in the context.**

예 42.4. “서비스의 메시지 컨텍스트에서 속성 가져오기” 호출된 작업을 나타내는 **WSDL** 작업 요소의 이름을 가져오는 코드를 표시합니다.

예 42.4. 서비스의 메시지 컨텍스트에서 속성 가져오기

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

...
// MessageContext context retrieved in a previous example
QName wsdl_operation = (QName)context.get(Message.WSDL_OPERATION);
```

42.2.4. 컨텍스트에서 속성 설정

구현을 위한 **MessageContext** 개체를 얻은 후에는 예 42.5. “MessageContext.put() 메서드” 에 표시된 **put()** 메서드를 사용하여 속성을 설정하고 기존 속성을 변경할 수 있습니다.

예 42.5. MessageContext.put() 메서드

VputKkeyVvalueClassCastExceptionIllegalArgumentException

설정 중인 속성이 메시지 컨텍스트에 이미 존재하는 경우 **put()** 메서드는 기존 값을 새 값으로 교체하고 이전 값을 반환합니다. 속성이 메시지 컨텍스트에 없는 경우 **put()** 메서드는 속성을 설정하고 **null** 을 반환합니다.

예 42.6. “서비스의 메시지 컨텍스트에서 속성 설정” **HTTP** 요청에 대한 응답 코드를 설정하기 위한 코드를 표시합니다.

예 42.6. 서비스의 메시지 컨텍스트에서 속성 설정

```
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.message.Message;

...
// MessageContext context retrieved in a previous example
context.put(Message.RESPONSE_CODE, new Integer(404));
```

42.2.5. 지원되는 컨텍스트

표 42.1. “서비스 구현 컨텍스트에서 사용 가능한 속성” 서비스 구현 개체의 컨텍스트를 통해 액세스할 수 있는 속성을 나열합니다.**Lists the properties accessible through the context in a service**

implementation object.

표 42.1. 서비스 구현 컨텍스트에서 사용 가능한 속성

속성 이름	설명
org.apache.cxf.message.Message	
PROTOCOL_HEADERS[a]	전송별 헤더 정보를 지정합니다. 값은 **java.util.Map<String, List<String>>**.
RESPONSE_CODE	사용자에게 반환되는 응답 코드를 지정합니다.Specifies the response code returned to the consumer. 값은 **Integer** 개체로 저장됩니다.
ENDPOINT_ADDRESS	서비스 공급자의 주소를 지정합니다. 값은 문자열로 저장 **됩니다**.
HTTP_REQUEST_METHOD	요청과 함께 전송된 HTTP 동사를 지정합니다. 값은 문자열로 저장 **됩니다**.
PATH_INFO	요청 중인 리소스의 경로를 지정합니다. 값은 문자열로 저장 **됩니다**. 경로는 hostname 뒤 및 쿼리 문자열 앞에 있는 URI의 부분입니다. 예를 들어 엔드포인트 URI가 http://cxf.apache.org/demo/widgets 인 경우 경로는 **/demo/widgets** 입니다.
QUERY_STRING	요청을 호출하는 데 사용되는 URI에 연결된 쿼리(있는 경우)를 지정합니다. 값은 문자열로 저장 **됩니다**. 쿼리는 URI의 끝에 표시됩니까 **?**? . 예를 들어 http://cxf.apache.org/demo/widgets?color 에 대한 요청이 발생하면 쿼리는 **색상** 입니다.
MTOM_ENABLED	SOAP 첨부에 대해 서비스 공급자가 MTOM을 사용할 수 있는지 여부를 지정합니다. 값은 **부울** 로 저장됩니다.
SCHEMA_VALIDATION_ENABLED	서비스 공급자가 스키마에 대한 메시지의 유효성을 검사할지 여부를 지정합니다.Specifies whether the service provider validates messages against a schema. 값은 **부울** 로 저장됩니다.
FAULT_STACKTRACE_ENABLED	런타임에서 스택 추적을 오류 메시지와 함께 제공하는지를 지정합니다. 값은 **부울** 로 저장됩니다.
CONTENT_TYPE	메시지의 MIME 형식을 지정합니다. 값은 문자열로 저장 **됩니다**.

속성 이름	설명
BASE_PATH	요청 중인 리소스의 경로를 지정합니다. 이 값은 **java.net.URL** 으로 저장됩니다. 경로는 hostname 뒤 및 쿼리 문자열 앞에 있는 URI의 부분입니다. 예를 들어 끝점 URL이 http://cxf.apache.org/demo/widgets 인 경우 기본 경로는 **/demo/widgets** 입니다.
ENCODING	메시지의 인코딩을 지정합니다. 값은 문자열로 저장 **됩니다**.
FIXED_PARAMETER_ORDER	매개변수가 메시지에 특정 순서로 표시될지 여부를 지정합니다. 값은 **부울** 로 저장됩니다.
MAINTAIN_SESSION	소비자가 향후 요청에 대해 현재 세션을 유지 관리하도록 지정합니다. 값은 **부울** 로 저장됩니다.
WSDL_DESCRIPTION	구현 중인 서비스를 정의하는 WSDL 문서를 지정합니다. 값은 **org.xml.sax.InputSource** 오브젝트로 저장됩니다.
WSDL_SERVICE	구현 중인 서비스를 정의하는 **wsdl:service** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_PORT	서비스에 액세스하는 데 사용되는 엔드포인트를 정의하는 **wsdl:port** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_INTERFACE	구현 중인 서비스를 정의하는 **wsdl:portType** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_OPERATION	소비자가 호출한 작업에 해당하는 **wsdl:operation** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
javax.xml.ws.handler.MessageContext	
MESSAGE_OUTBOUND_PROPERTY	메시지가 아웃바운드인지 여부를 지정합니다. 값은 **부울** 로 저장됩니다. **true** 는 메시지가 아웃바운드로 지정되도록 지정합니다.

속성 이름	설명
INBOUND_MESSAGE_ATTACHMENTS	요청 메시지에 포함된 첨부 파일이 포함되어 있습니다. 값은 **java.util.Map<String, DataHandler>로 저장됩니다**. 맵의 키 값은 헤더의 MIME Content-ID입니다.
OUTBOUND_MESSAGE_ATTACHMENTS	응답 메시지에 대한 첨부 파일이 포함되어 있습니다. 값은 **java.util.Map<String, DataHandler>로 저장됩니다**. 맵의 키 값은 헤더의 MIME Content-ID입니다.
WSDL_DESCRIPTION	구현 중인 서비스를 정의하는 WSDL 문서를 지정합니다. 값은 **org.xml.sax.InputSource** 오브젝트로 저장됩니다.
WSDL_SERVICE	구현 중인 서비스를 정의하는 **wsdl:service** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_PORT	서비스에 액세스하는 데 사용되는 엔드포인트를 정의하는 **wsdl:port** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_INTERFACE	구현 중인 서비스를 정의하는 **wsdl:portType** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
WSDL_OPERATION	소비자가 호출한 작업에 해당하는 **wsdl:operation** 요소의 정규화된 이름을 지정합니다. 값은 **QName** 으로 저장됩니다.
HTTP_RESPONSE_CODE	사용자에게 반환되는 응답 코드를 지정합니다.Specifies the response code returned to the consumer. 값은 **Integer** 개체로 저장됩니다.
HTTP_REQUEST_HEADERS	요청에서 HTTP 헤더를 지정합니다. 값은 **java.util.Map<String, List<String>>**.
HTTP_RESPONSE_HEADERS	응답에 대한 HTTP 헤더를 지정합니다. 값은 **java.util.Map<String, List<String>>**.
HTTP_REQUEST_METHOD	요청과 함께 전송된 HTTP 동사를 지정합니다. 값은 문자열로 저장 **됩니다**.

속성 이름	설명
SERVLET_REQUEST	서블릿의 요청 오브젝트를 포함합니다. 이 값은 **javax.servlet.http.HttpServletRequest** 로 저장됩니다.
SERVLET_RESPONSE	서블릿의 response 오브젝트를 포함합니다. 이 값은 **javax.servlet.http.HttpResponse** 로 저장됩니다.
SERVLET_CONTEXT	서블릿의 context 오브젝트를 포함합니다. 이 값은 **javax.servlet.ServletContext** 로 저장됩니다.
PATH_INFO	요청 중인 리소스의 경로를 지정합니다. 값은 문자열로 저장 **됩니다**. 경로는 hostname 뒤 및 쿼리 문자열 앞에 있는 URI의 부분입니다. 예를 들어 엔드포인트 URL이 http://cxf.apache.org/demo/widgets 인 경우 경로는 **/demo/widgets** 입니다.
QUERY_STRING	요청을 호출하는 데 사용되는 URI에 연결된 쿼리(있는 경우)를 지정합니다. 값은 문자열로 저장 **됩니다**. 쿼리는 URI의 끝에 표시됩니까 **?**? . 예를 들어 http://cxf.apache.org/demo/widgets?color 에 대한 요청이 발생하면 쿼리 문자열이 **색상** 입니다.
REFERENCE_PARAMETERS	WS-Addressing 참조 매개 변수를 지정합니다. 여기에는 **wsa:IsReferenceParameter** 속성이 **true** 로 설정된 모든 SOAP 헤더가 포함됩니다. 값은 **java.util.List** 로 저장됩니다.
org.apache.cxf.transport.jms.JMSConstants	
JMS_SERVER_HEADERS	JMS 메시지 헤더를 포함합니다. 자세한 내용은 42.4절. "JMS Message Properties 사용" 에서 참조하십시오.
[a] HTTP를 사용하는 경우 이 속성은 표준 JAX-WS 정의 속성과 동일합니다.	

42.3. 소비자 구현에서 컨텍스트 작업

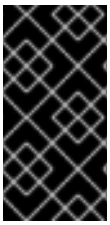
42.3.1. 개요

*소비자 구현은 **BindingProvider** 인터페이스를 통해 컨텍스트 정보에 액세스할 수 있습니다. **BindingProvider** 인스턴스는 두 개의 개별 컨텍스트에서 컨텍스트 정보를 보유합니다.The*

BindingProvider instance holds context information in two separate contexts:

- *요청 컨텍스트 요청 컨텍스트를 사용하면 아웃바운드 메시지에 영향을 주는 속성을 설정할 수 있습니다.The request context enables you to set properties that affect outbound messages. 요청 컨텍스트 속성은 특정 포트 인스턴스에 적용되며, 설정된 경우 속성은 해당 속성이 명시적으로 지워질 때까지 포트에서 수행된 모든 후속 작업 호출에 영향을 줍니다.Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such as a property is explicitly cleared. 예를 들어 요청 컨텍스트 속성을 사용하여 연결 제한 시간을 설정하거나 헤더에서 전송하기 위해 데이터를 초기화할 수 있습니다.*

- *응답 컨텍스트 응답 컨텍스트 를 사용하면 현재 스레드에서 만든 마지막 작업 호출에 대한 응답으로 설정된 속성 값을 읽을 수 있습니다.The response context enables you to read the property values set by the response to the last operation invocation made from the current thread. 응답 컨텍스트 속성은 작업을 호출할 때마다 재설정됩니다. 예를 들어 응답 컨텍스트 속성에 액세스하여 마지막 인바운드 메시지에서 수신된 헤더 정보를 읽을 수 있습니다.For example, you might access a response context property to read header information received from the last inbound message.*



중요

메시지 컨텍스트의 애플리케이션 범위에 배치된 정보만 소비자 구현에서 액세스할 수 있습니다.

42.3.2. 컨텍스트 가져오기

컨텍스트는 javax.xml.ws.BindingProvider 인터페이스를 사용하여 가져옵니다. BindingProvider 인터페이스에는 컨텍스트를 가져오는 두 가지 방법이 있습니다.

- *getRequestContext() 예 42.7. “getRequestContext() 메서드” 에 표시된 getRequestContext() 메서드는 요청 컨텍스트를 Map 오브젝트로 반환합니다. 반환된 Map 개체를 사용하여 컨텍스트의 콘텐츠를 직접 조작할 수 있습니다.*

  예 42.7. getRequestContext() 메서드

  ```
  Map<String, Object>getRequestContext
  ```

- *getResponseContext() 예 42.8. “getResponseContext() 메서드” 에 표시된 getResponseContext() 는 응답 컨텍스트를 Map 오브젝트로 반환합니다. 반환된 Map 오브젝트*

의 내용은 현재 스레드에서 이루어진 원격 서비스에 대한 가장 최근의 성공적인 요청에서 응답 컨텍스트 내용의 상태를 반영합니다.

예 42.8. getResponseContext() 메서드

Map<String, Object>getResponseContext

프록시 개체는 BindingProvider 인터페이스를 구현 하므로 프록시 개체를 캐스팅 하 여 BindingProvider 개체를 가져올 수 있습니다.Since proxy objects implement the BindingProvider interface, a BindingProvider object can be obtained by casting a proxy object. BindingProvider 오브젝트에서 얻은 컨텍스트는 이를 생성하는 데 사용되는 프록시 오브젝트에서 호출되는 작업에만 유효합니다.

예 42.9. "소비자의 요청 컨텍스트 가져오기" 프록시에 대한 요청 컨텍스트를 가져오는 코드를 보여줍니다.

예 42.9. 소비자의 요청 컨텍스트 가져오기

```
// Proxy widgetProxy obtained previously
BindingProvider bp = (BindingProvider)widgetProxy;
Map<String, Object> requestContext = bp.getRequestContext();
```

42.3.3. 컨텍스트에서 속성을 읽습니다.Read a property from a context.

소비자 컨텍스트는 java.util.Map<String, Object> 오브젝트에 저장 됩니다. 맵에는 임의의 오브젝트가 포함된 String 개체와 값이 있는 키가 있습니다. java.util.Map.get() 을 사용하여 응답 컨텍스트 속성 맵의 항목에 액세스합니다.

특정 컨텍스트 속성인 ContextPropertyName 을 검색하려면 예 42.10. "응답 컨텍스트 속성 읽기" 에 표시된 코드를 사용합니다.

예 42.10. 응답 컨텍스트 속성 읽기

```
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
  ((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType) responseContext.get(ContextPropertyName);
```

42.3.4. 컨텍스트에서 속성 설정

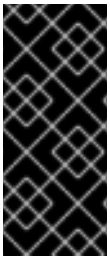
소비자 컨텍스트는 **java.util.Map<String, Object >** 오브젝트에 저장된 해시 맵입니다. 이 맵에는 임의의 오브젝트인 **String** 개체와 값이 있는 키가 있습니다. 컨텍스트에서 속성을 설정하려면 **java.util.Map.put()** 메서드를 사용합니다.

요청 컨텍스트와 응답 컨텍스트 모두에서 속성을 설정할 수 있지만 요청 컨텍스트에 대한 변경 사항만 메시지 처리에 영향을 미칩니다. 현재 스레드에서 각 원격 호출이 완료되면 응답 컨텍스트의 속성이 재설정됩니다.

예 42.11. “요청 컨텍스트 속성 설정” 에 표시된 코드는 **BindingProvider.ENDPOINT_ADDRESS_PROPERTY** 값을 설정하여 대상 서비스 공급자의 주소를 변경합니다.

예 42.11. 요청 컨텍스트 속성 설정

```
// Set request context property.
java.util.Map<String, Object> requestContext =
    ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://localhost:8080/widgets");

// Invoke an operation.
port.SomeOperation();
```



중요

속성이 요청 컨텍스트에서 설정되면 해당 값은 모든 후속 원격 호출에 사용됩니다.**Once a property is set in the request context its value is used for all subsequent remote invocations.** 값을 변경할 수 있으며 변경된 값이 사용됩니다.

42.3.5. 지원되는 컨텍스트

Apache CXF는 소비자 구현에서 다음 컨텍스트 속성을 지원합니다.

표 42.2. 소비자 컨텍스트 속성

속성 이름	설명
javax.xml.ws.BindingProvider	

속성 이름	설명
ENDPOINT_ADDRESS_PROPERTY	대상 서비스의 주소를 지정합니다. 값은 문자열로 저장됩니다.
USERNAME_PROPERTY[a]	HTTP 기본 인증에 사용되는 사용자 이름을 지정합니다. 값은 문자열로 저장 **됩니다**.
PASSWORD_PROPERTY[b]	HTTP 기본 인증에 사용되는 암호를 지정합니다. 값은 문자열로 저장 **됩니다**.
SESSION_MAINTAIN_PROPERTY[c]	클라이언트가 세션 정보를 유지할지 여부를 지정합니다. 값은 **부울** 오브젝트로 저장됩니다.
org.apache.cxf.ws.addressing.JAXWSAConstants	
CLIENT_ADDRESSING_PROPERTIES	소비자가 원하는 서비스 공급자에 연결하는 데 사용하는 WS-Addressing 정보를 지정합니다. 값은 **org.apache.cxf.ws.addressing.AddressingProperties** 로 저장됩니다.
org.apache.cxf.transports.jms.context.JMSConstants	
JMS_CLIENT_REQUEST_HEADERS	메시지에 대한 JMS 헤더를 포함합니다. 자세한 내용은 42.4절. "JMS Message Properties 사용" 에서 참조하십시오.

[a] 이 속성은 HTTP 보안 설정에 정의된 사용자 이름으로 재정의됩니다.

[b] 이 속성은 HTTP 보안 설정에 정의된 암호로 재정의됩니다.

[c] Apache CXF는 이 속성을 무시합니다.

42.4. JMS MESSAGE PROPERTIES 사용

초록

Apache CXF JMS 전송에는 JMS 메시지의 속성을 검사하는 데 사용할 수 있는 컨텍스트 메커니즘이 있습니다. 컨텍스트 메커니즘을 사용하여 JMS 메시지의 속성을 설정할 수도 있습니다.

42.4.1. JMS 메시지 헤더 검사

초록

소비자 및 서비스는 **JMS** 메시지 헤더 속성에 액세스하기 위해 다양한 컨텍스트 메커니즘을 사용합니다. 그러나 두 메커니즘 모두 헤더 속성을 **org.apache.cxf.transports.jms.context.JMSMessageHeadersType** 오브젝트로 반환합니다.

42.4.1.1. 서비스에서 JMS 메시지 헤더 가져오기

WebServiceContext 개체에서 **JMS** 메시지 헤더 속성을 가져오려면 다음을 수행합니다.

1. "컨텍스트 가져오기"에 설명된 대로 컨텍스트를 가져옵니다.
2. **org.apache.cxf.transports.jms.JMSConstants.JMS_SERVER_HEADERS** 매개 변수와 함께 메시지 컨텍스트의 **get()** 메서드를 사용하여 메시지 컨텍스트에서 메시지 헤더를 가져옵니다.

예 42.12. "서비스 구현에서 JMS 메시지 헤더 가져오기" 서비스의 메시지 컨텍스트에서 **JMS** 메시지 헤더를 가져오는 코드를 표시합니다.

예 42.12. 서비스 구현에서 JMS 메시지 헤더 가져오기

```
import org.apache.cxf.transport.jms.JMSConstants;
import org.apache.cxf.transports.jms.context.JMSMessageHeadersType;

@WebService(serviceName = "HelloWorldService",
                    portName = "HelloWorldPort",
                    endpointInterface = "org.apache.cxf.hello_world_jms.HelloWorldPortType",
                    targetNamespace = "http://cxf.apache.org/hello_world_jms")
 public class GreeterImplTwoWayJMS implements HelloWorldPortType
 {
  @Resource
  protected WebServiceContext wsContext;
  ...

  @WebMethod
  public String greetMe(String me)
  {
   MessageContext mc = wsContext.getMessageContext();
   JMSMessageHeadersType headers = (JMSMessageHeadersType)
mc.get(JMSConstants.JMS_SERVER_HEADERS);
    ...
  }
   ...
}
```

42.4.1.2. 소비자에서 JMS 메시지 헤더 속성 가져오기

메시지가 JMS 전송에서 성공적으로 검색되면 소비자의 응답 컨텍스트를 사용하여 JMS 헤더 속성을 검사할 수 있습니다. 또한 "client Receive Timeout" 에 설명된 대로 시간이 초과되기 전에 클라이언트가 대기하는 시간을 설정하거나 확인할 수 있습니다. 소비자의 응답 컨텍스트에서 JMS 메시지 헤더를 가져오려면 다음을 수행합니다.

1. "컨텍스트 가져오기" 에 설명된 대로 응답 컨텍스트를 가져옵니다.

2. org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_RESPONSE_HEADERS 를 매개 변수로 사용하여 응답 컨텍스트에서 JMS 메시지 헤더 속성을 가져옵니다.

예 42.13. "소비자 응답 헤더에서 JMS 헤더 가져오기" 소비자의 응답 컨텍스트에서 JMS 메시지 헤더 속성을 가져오는 코드를 표시합니다.

예 42.13. 소비자 응답 헤더에서 JMS 헤더 가져오기

```
import org.apache.cxf.transports.jms.context.*;
// Proxy greeter initialized previously
BindingProvider  bp = (BindingProvider)greeter;
Map<String, Object> responseContext = bp.getResponseContext();
JMSMessageHeadersType responseHdr = (JMSMessageHeadersType)
                    responseContext.get(JMSConstants.JMS_CLIENT_RESPONSE_HEADERS);
...
}
```

예 42.13. "소비자 응답 헤더에서 JMS 헤더 가져오기" 의 코드는 다음을 수행합니다.

프록시를 BindingProvider로 캐스팅합니다.

응답 컨텍스트를 가져옵니다.

응답 컨텍스트에서 JMS 메시지 헤더를 검색합니다.

42.4.2. 메시지 헤더 속성 검사

42.4.2.1. 표준 JMS 헤더 속성

표 42.3. "JMS 헤더 속성" 검사할 수 있는 JMS 헤더의 표준 속성을 나열합니다.

표 42.3. JMS 헤더 속성

속성 이름	속성 유형	getter Method
ID 상관 관계	**string**	**getJMSCorralationID()**
전송 모드	**int**	**getJMSDeliveryMode()**
메시지 만료	**long**	**getJMSExpiration()**
메시지 ID	**string**	**getJMSMessageID()**
우선 순위	**int**	**getJMSPriority()**
redelivered	**boolean**	**getJMSRedlivered()**
Time Stamp	**long**	**getJMSTimeStamp()**
유형	**string**	**getJMSType()**
라이브 시간	**long**	**getTimeToLive()**

42.4.2.2. 선택적 헤더 속성

*또한 **JMSMessageHeadersType.getProperty()** 를 사용하여 **JMS** 헤더에 저장된 선택적 속성을 검사할 수 있습니다. 선택적 속성은 **org.apache.cxf.transports.jms.context.JMSPropertyType** 목록으로 반환됩니다. 선택적 속성은 이름/값 쌍으로 저장됩니다.*

42.4.2.3. 예제

*예 42.14. "JMS 헤더 속성 읽기" 응답 컨텍스트를 사용하여 일부 **JMS** 속성을 검사하는 코드를 보여줍니다.*

예 42.14. JMS 헤더 속성 읽기

```
// JMSMessageHeadersType messageHdr retrieved previously
System.out.println("Correlation ID: "+messageHdr.getJMSCorrelationID());
System.out.println("Message Priority: "+messageHdr.getJMSPriority());
System.out.println("Redelivered: "+messageHdr.getRedelivered());

JMSPropertyType prop = null;
List<JMSPropertyType> optProps = messageHdr.getProperty();
Iterator<JMSPropertyType> iter = optProps.iterator();
```

```
while (iter.hasNext())
{
  prop = iter.next();
   System.out.println("Property name: "+prop.getName());
   System.out.println("Property value: "+prop.getValue());
}
```

예 42.14. "JMS 헤더 속성 읽기" 의 코드는 다음을 수행합니다.

메시지의 상관 관계 ID 값을 출력합니다.

메시지의 우선순위 속성 값을 출력합니다.

메시지의 재전송 속성 값을 출력합니다.

메시지의 선택적 헤더 속성 목록을 가져옵니다.Gets the list of the message's optional header properties.

속성 목록을 통과할 Iterator 를 가져옵니다.

선택적 속성 목록을 반복하고 이름과 값을 출력합니다.

42.4.3. JMS 속성 설정

초록

소비자 끝점에서 요청 컨텍스트를 사용하여 여러 JMS 메시지 헤더 속성 및 소비자 끝점의 제한 값을 설정할 수 있습니다. 이러한 속성은 단일 호출에 유효합니다. 서비스 프록시에서 작업을 호출할 때마다 다시 설정해야 합니다.

서비스에 헤더 속성을 설정할 수 없습니다.

42.4.3.1. JMS 헤더 속성

표 42.4. "settable JMS Header Properties" 소비자 엔드포인트의 요청 컨텍스트를 사용하여 설정할

수 있는 JMS 헤더의 속성을 나열합니다.

표 42.4. settable JMS Header Properties

속성 이름	속성 유형	setter Method
ID 상관 관계	**string**	**setJMSCorralationID()**
전송 모드	**int**	**setJMSDeliveryMode()**
우선 순위	**int**	**setJMSPriority()**
라이브 시간	**long**	**setTimeToLive()**

이러한 속성을 설정하려면 다음을 수행합니다.

1. org.apache.cxf.transports.jms.context.JMSMessageHeadersType 오브젝트를 생성합니다.
2. 표 42.4. "settable JMS Header Properties" 에 설명된 적절한 설정 방법을 사용하여 설정할 값을 채웁니다.
3. org.apache.cxf.transports.jms.JMSConstants.JMS_CLIENT_REQUEST_HEADERS를 첫 번째 인수로, 새 JMSMessageHeadersType 오브젝트를 두 번째 인수로 호출하여 요청 컨텍스트의 put() 메서드를 요청 컨텍스트로 설정합니다.

42.4.3.2. 선택적 JMS 헤더 속성

선택적 속성을 JMS 헤더로 설정할 수도 있습니다. 선택적 JMS 헤더 속성은 다른 JMS 헤더 속성을 설정하는 데 사용되는 JMSMessageHeadersType 오브젝트에 저장됩니다. 이러한 개체는 org.apache.cxf.transports.jms.context.JMSPropertyType 개체가 포함된 List 오브젝트로 저장됩니다. JMS 헤더에 선택적 속성을 추가하려면 다음을 수행합니다.

1. JMSPropertyType 오브젝트를 생성합니다.
2. setName() 을 사용하여 속성의 이름을 설정합니다.

3. ***setValue()*** *를 사용하여 속성의 값 필드를 설정합니다.*

4. ***JMSMessageHeadersType.getProperty().add(JMSPropertyType)*** *를 사용하여* ***JMS*** *메시지 헤더에 속성을 추가합니다.*

5. *모든 속성이 메시지 헤더에 추가될 때까지 절차를 반복합니다.*

42.4.3.3. client Receive Timeout

JMS *헤더 속성 외에도 소비자 엔드포인트가 시간 초과하기 전에 응답을 대기하는 시간을 설정할 수 있습니다.* ***org.apache.cxf.transports.jms.JMSConstants.JMSConstants.JMSConstants.JMS_CLIENT_RECEIVE_TIMEOUT****을 사용하여 요청 컨텍스트의* ***put()*** *메서드를 호출하고 소비자가 두 번째 인수로 대기하도록 시간(밀리초)을 나타내는 긴 값을 설정합니다.*

42.4.3.4. 예제

예 42.15. "요청 컨텍스트를 사용하여 JMS 속성 설정" 요청 컨텍스트를 사용하여 일부 ***JMS*** *속성을 설정하는 코드를 표시합니다.*

예 42.15. 요청 컨텍스트를 사용하여 JMS 속성 설정

```
import org.apache.cxf.transports.jms.context.*;
 // Proxy greeter initialized previously
InvocationHandler handler = Proxy.getInvocationHandler(greeter);

BindingProvider bp= null;
if (handler instanceof BindingProvider)
{
  bp = (BindingProvider)handler;
  Map<String, Object> requestContext = bp.getRequestContext();

  JMSMessageHeadersType requestHdr = new JMSMessageHeadersType();
  requestHdr.setJMSCorrelationID("WithBob");
  requestHdr.setJMSExpiration(3600000L);


  JMSPropertyType prop = new JMSPropertyType;
  prop.setName("MyProperty");
  prop.setValue("Bluebird");
  requestHdr.getProperty().add(prop);

  requestContext.put(JMSConstants.CLIENT_REQUEST_HEADERS, requestHdr);
```

```
    requestContext.put(JMSConstants.CLIENT_RECEIVE_TIMEOUT, new Long(1000));
  }
```

예 42.15. "요청 컨텍스트를 사용하여 JMS 속성 설정" 의 코드는 다음을 수행합니다.

변경하려는 JMS 속성이 있는 프록시의 InvocationHandler 를 가져옵니다.

InvocationHandler 가 BindingProvider 인지 확인합니다.

반환된 InvocationHandler 개체를 BindingProvider 개체로 캐스팅하여 요청 컨텍스트를 검색합니다. casts the returned InvocationHandler object into a BindingProvider object to retrieve the request context.

요청 컨텍스트를 가져옵니다.

새 메시지 헤더 값을 보유하는 JMSMessageHeadersType 오브젝트를 생성합니다.

Correlation ID를 설정합니다.

만료 속성을 60분으로 설정합니다.

새 JMSPropertyType 오브젝트를 생성합니다.

선택적 속성의 값을 설정합니다.

메시지 헤더에 선택적 속성을 추가합니다.

JMS 메시지 헤더 값을 요청 컨텍스트로 설정합니다.

클라이언트 수신 시간 초과 속성을 1초로 설정합니다.

43장. 핸들러 작성

초록

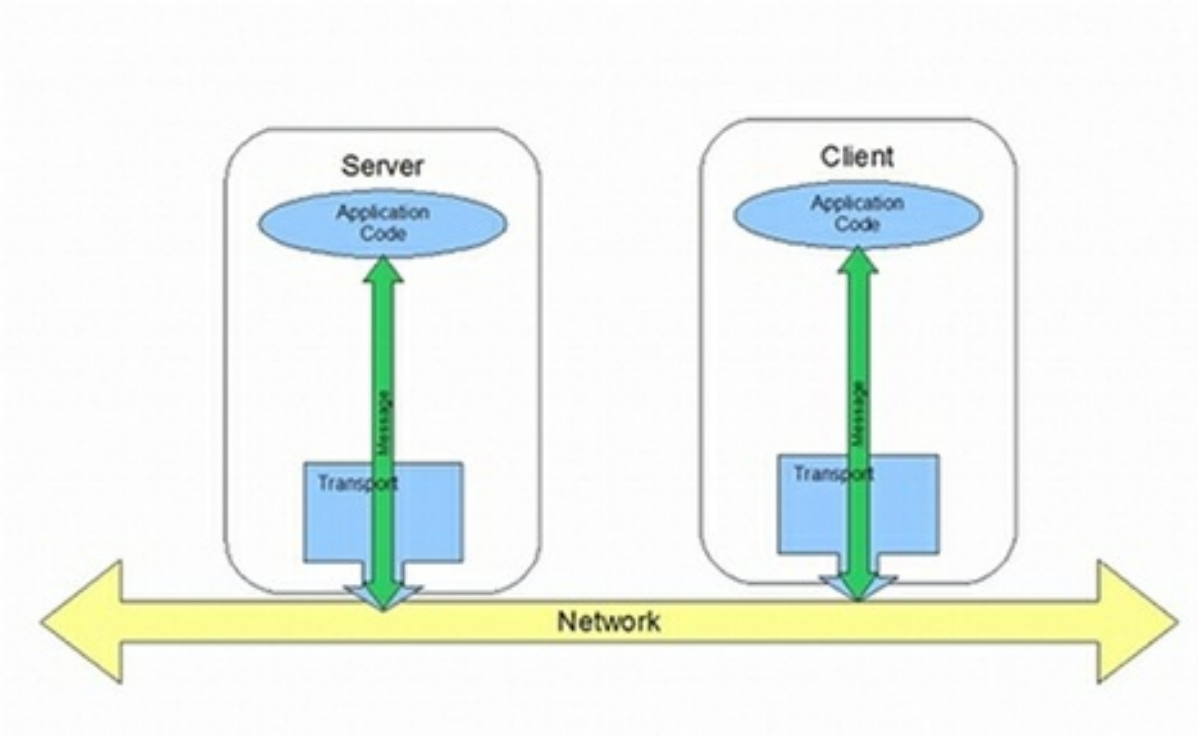
JAX-WS는 메시지 처리 모듈을 애플리케이션에 추가하기 위한 유연한 플러그인 프레임워크를 제공합니다. 핸들러라고 하는 이러한 모듈은 애플리케이션 수준 코드와 독립적이며 낮은 수준의 메시지 처리 기능을 제공할 수 있습니다.

43.1. 핸들러: 소개

43.1.1. 개요

서비스 프록시가 서비스에서 작업을 호출하면 작업의 매개 변수가 **Apache CXF**로 전달됩니다. 서비스에서 메시지를 수신한 경우 **Apache CXF**는 전선에서 메시지를 읽고 메시지를 재구성한 다음 작업 매개 변수를 작업을 수행하는 애플리케이션 코드로 전달합니다. 애플리케이션 코드가 요청 처리를 완료하면 응답 메시지에서 요청을 시작한 서비스 프록시로의 여행 시 유사한 이벤트 체인을 수행합니다. 이는 그림 43.1. "메시지 교환 경로"에 표시됩니다.

그림 43.1. 메시지 교환 경로

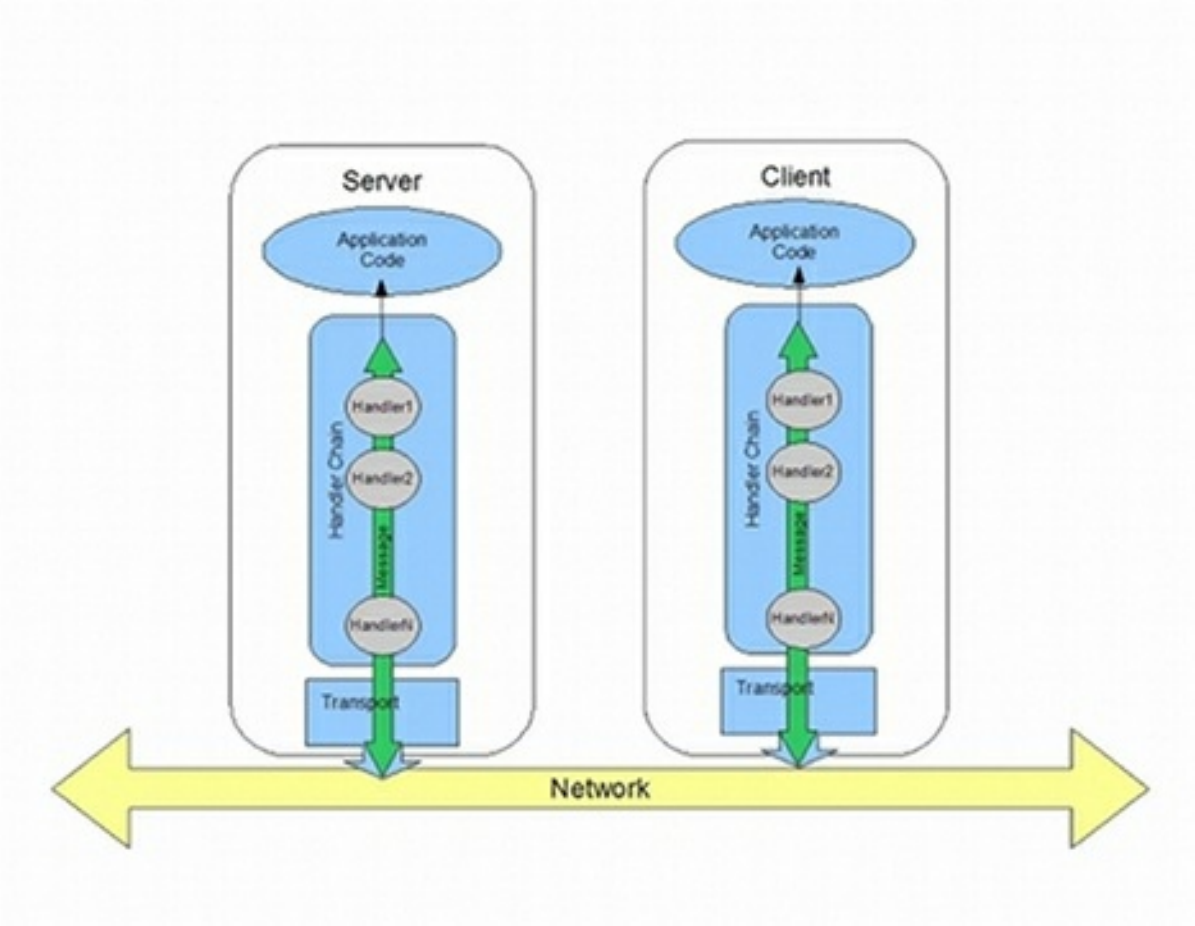


JAX-WS는 애플리케이션 수준 코드와 네트워크 간에 메시지 데이터를 조작하는 메커니즘을 정의합니다. 예를 들어 열린 네트워크를 통해 메시지 데이터를 전달하려는 경우 독점 암호화 메커니즘을 사용하여 암호화할 수 있습니다. 데이터를 암호화하고 해독하는 **JAX-WS** 핸들러를 작성할 수 있습니다. 그런 다음 모든 클라이언트 및 서버의 메시지 처리 체인에 처리기를 삽입할 수 있습니다.

그림 43.2. "핸들러를 사용한 메시지 교환 경로"에 표시된 대로 핸들러는 애플리케이션 수준 코드와

메시지를 네트워크에 배치하는 전송 코드 간에 이동하는 체인에 배치됩니다.

*그림 **43.2.** 핸들러를 사용한 메시지 교환 경로*



43.1.2. 처리기 유형

***JAX-WS** 사양은 다음 두 가지 기본 핸들러 유형을 정의합니다.*

- *논리 핸들러 논리 핸들러는 메시지 페이로드 및 메시지 컨텍스트에 저장된 속성을 처리할 수 있습니다. 예를 들어 애플리케이션에서 순수 **XML** 메시지를 사용하는 경우 논리 핸들러는 전체 메시지에 액세스할 수 있습니다. 애플리케이션에서 **SOAP** 메시지를 사용하는 경우 논리 핸들러는 **SOAP** 본문의 콘텐츠에 액세스할 수 있습니다. 메시지 컨텍스트에 배치되지 않는 한 **SOAP** 헤더 또는 첨부 파일에 액세스할 수 없습니다.*

  논리 핸들러는 처리기 체인의 애플리케이션 코드에 가장 가까운 위치에 배치됩니다. 즉, 메시지가 애플리케이션 코드에서 전송으로 전달될 때 먼저 실행됩니다. 메시지가 네트워크에서 수신되고 애플리케이션 코드로 다시 전달되면 논리 핸들러가 마지막에 실행됩니다.

- *프로토콜 핸들러 프로토콜 핸들러는 네트워크에서 수신한 전체 메시지와 메시지 컨텍스트에 저장된 속성을 처리할 수 있습니다. 예를 들어 애플리케이션에서 **SOAP** 메시지를 사용하는 경우 프로토콜 처리기는 **SOAP** 본문, **SOAP** 헤더 및 모든 연결의 콘텐츠에 액세스할 수 있습니다.**For***

example, if the application uses SOAP messages, the protocol handlers would have access to the contents of the SOAP body, the SOAP headers, and any attachments.

프로토콜 핸들러는 처리기 체인의 전송에 가장 가까운 위치에 배치됩니다. 즉, 메시지가 네트워크에서 수신될 때 먼저 실행됩니다. 메시지가 애플리케이션 코드에서 네트워크로 전송되면 프로토콜 처리기가 마지막으로 실행됩니다.

참고

Apache CXF에서 지원하는 유일한 프로토콜 핸들러는 **SOAP**에 따라 다릅니다.

43.1.3. 핸들러 구현

두 핸들러 유형의 차이점은 매우 미묘하며 공통 기본 인터페이스를 공유합니다. 논리 핸들러와 프로토콜 핸들러는 공통의 부모로 인해 구현되어야 하는 여러 메서드를 공유합니다.

- **handleMessage()** **handleMessage()** 메서드는 모든 처리기의 중앙 메서드입니다. 이 방법은 일반 메시지 처리를 담당합니다.

- **handleFault()** **handleFault()** 는 오류 메시지를 처리하는 방법입니다.

- **close()** **close()** 는 메시지가 체인의 끝에 도달할 때 핸들러 체인의 모든 실행된 핸들러에서 호출됩니다. 메시지 처리 중에 사용되는 리소스를 정리하는 데 사용됩니다.

논리 핸들러 구현과 프로토콜 핸들러 구현 간의 차이점은 다음과 같습니다.

- 구현되는 특정 인터페이스

  모든 핸들러는 **Handler** 인터페이스에서 파생되는 인터페이스를 구현합니다. 논리 핸들러는 **LogicalHandler** 인터페이스를 구현합니다. 프로토콜 핸들러는 **Handler** 인터페이스의 프로토콜별 확장을 구현합니다. 예를 들어 **SOAP** 핸들러는 **SOAPHandler** 인터페이스를 구현합니다.

- 처리기에서 사용할 수 있는 정보의 양입니다.

  프로토콜 핸들러는 메시지의 콘텐츠와 메시지 콘텐츠와 함께 패키지된 모든 프로토콜별 정보

에 액세스할 수 있습니다. 논리 핸들러는 메시지의 콘텐츠에만 액세스할 수 있습니다. 논리 핸들러는 프로토콜 세부 정보에 대한 지식이 없습니다.

43.1.4. 애플리케이션에 핸들러 추가

애플리케이션에 처리기를 추가하려면 다음을 수행해야 합니다.

1. *처리기를 서비스 공급자, 소비자 또는 둘 다에서 사용할지 여부를 결정합니다.*
2. *작업에 가장 적합한 핸들러 유형을 결정합니다.*
3. *올바른 인터페이스를 구현합니다.*

   *논리 핸들러를 구현하려면 **43.2절. "논리 핸들러 구현"** 에서 참조하십시오.*

   *프로토콜 핸들러를 구현하려면 **43.4절. "프로토콜 핸들러 구현"** 에서 참조하십시오.*
4. *핸들러를 사용하도록 끝점을 구성합니다. **43.10절. "핸들러를 사용하도록 끝점 구성"** 을 참조하십시오.*

43.2. 논리 핸들러 구현

43.2.1. 개요

*논리 핸들러는 **javax.xml.ws.handler.LogicalHandler** 인터페이스를 구현합니다. **예 43.1. "LogicalHandler Synopsis"** 에 표시된 **LogicalHandler** 인터페이스는 **LogicalMessageContext** 개체를 **handleMessage()** 메서드 및 **handleFault()** 메서드로 전달합니다. 컨텍스트 개체는 메시지의 본문 과 메시지 교환의 컨텍스트로 설정된 모든 속성에 대한 액세스를 제공합니다.*

예 43.1. LogicalHandler Synopsis

```
public interface LogicalHandler extends Handler
{
  boolean handleMessage(LogicalMessageContext context);
  boolean handleFault(LogicalMessageContext context);
  void close(LogicalMessageContext context);
}
```

–

43.2.2. 절차

논리 hander를 구현하려면 다음을 수행합니다.

1. 처리기에 필요한 모든 43.6절. "핸들러 초기화" 논리를 구현합니다.
2. 43.3절. "논리 핸들러에서 메시지 처리" 논리를 구현합니다.
3. 43.7절. "자주하는 질문" 논리를 구현합니다.
4. 완료되면 43.8절. "핸들러 종료" 처리기에 대한 논리를 구현합니다.
5. 43.9절. "핸들러 릴리스" 의 논리를 실행하기 전에 처리기의 리소스를 삭제합니다.

43.3. 논리 핸들러에서 메시지 처리

43.3.1. 개요

일반 메시지 처리는 handleMessage() 메서드에서 처리됩니다.

handleMessage() 메서드는 메시지 본문 및 메시지 컨텍스트에 저장된 모든 속성에 대한 액세스를 제공하는 LogicalMessageContext 개체를 수신합니다.

handleMessage() 메서드는 메시지 처리를 계속하는 방법에 따라 true 또는 false를 반환합니다. 또한 예외를 throw할 수 있습니다.

43.3.2. 메시지 데이터 가져오기Get the message data

논리 메시지 처리기로 전달된 LogicalMessageContext 개체를 사용하면 컨텍스트의 getMessage() 메서드를 사용하여 메시지 본문에 액세스할 수 있습니다. 예 43.2. "논리 핸들러에서 메시지 페이로드를

가져오는 방법" 에 표시된 **getMessage()** 메서드는 메시지 페이로드를 **LogicalMessage** 오브젝트로 반환합니다.

> *예 43.2. 논리 핸들러에서 메시지 페이로드를 가져오는 방법*
>
> **LogicalMessagegetMessage**

LogicalMessage 개체가 있으면 이를 사용하여 메시지 본문을 조작할 수 있습니다. *예 43.3. "논리 메시지 보유기"* 에 표시된 **LogicalMessage** 인터페이스에는 실제 메시지 본문을 사용하기 위한 **getter** 및 **setter**가 있습니다.

> *예 43.3. 논리 메시지 보유기*
>
> **LogicalMessageSourcegetPayloadObjectgetPayloadJAXBContext컨텍스트setPayloadObjectpayload**

중요

*메시지 페이로드의 내용은 사용 중인 바인딩 유형에 따라 결정됩니다. **SOAP** 바인딩은 메시지의 **SOAP** 본문에만 액세스할 수 있습니다. **XML** 바인딩은 전체 메시지 본문에 액세스할 수 있도록 합니다.*

43.3.3. 메시지 본문을 XML 개체로 작업

*논리 메시지 쌍은 메시지 페이로드를 **javax.xml.transform.dom.DOMSource** 개체로 사용하여 작업합니다.*

*매개 변수가 없는 **getPayload()** 메서드는 메시지 페이로드를 **DOMSource** 개체로 반환합니다. 반환된 오브젝트는 실제 메시지 페이로드입니다. 반환된 개체의 모든 변경 사항은 메시지 본문을 즉시 변경합니다.*

*메시지의 본문을 단일 **Source** 개체를 사용하는 **setPayload()** 메서드를 사용하여 **DOMSource** 개체로 교체할 수 있습니다.*

43.3.4. message body를 JAXB 오브젝트로 작업

다른 쌍의 **getter** 및 **setter**를 사용하면 메시지 페이로드를 **JAXB** 오브젝트로 작업할 수 있습니다. **JAXBContext** 개체를 사용하여 메시지 페이로드를 **JAXB** 오브젝트로 변환합니다.

JAXB 오브젝트를 사용하려면 다음을 수행합니다.

1. 메시지 본문에서 데이터 유형을 관리할 수 있는 **JAXBContext** 개체를 가져옵니다.

   JAXBContext 오브젝트 생성에 대한 자세한 내용은 **39장. A JAXBContext 오브젝트 사용**을 참조하십시오.

2. 예 **43.4. "Message Body를 JAXB 오브젝트로 가져오기"**에 표시된 대로 메시지 본문을 가져옵니다.

   예 43.4. Message Body를 JAXB 오브젝트로 가져오기

   ```
   JAXBContext jaxbc = JAXBContext(myObjectFactory.class);
   Object body = message.getPayload(jaxbc);
   ```

3. 반환된 개체를 적절한 형식으로 캐스팅합니다.

4. 메시지 본문을 필요에 따라 조작합니다.

5. 업데이트된 메시지 본문을 예 **43.5. "JAXB 오브젝트를 사용하여 메시지 본문 업데이트"**에 표시된 대로 컨텍스트로 다시 넣습니다.

   예 43.5. JAXB 오브젝트를 사용하여 메시지 본문 업데이트

   ```
   message.setPayload(body, jaxbc);
   ```

43.3.5. 컨텍스트 속성 작업

논리 처리기로 전달된 논리 메시지 컨텍스트는 애플리케이션의 메시지 컨텍스트 인스턴스이며 이 컨텍스트에 저장된 모든 속성에 액세스할 수 있습니다. 핸들러는 **APPLICATION** 범위 및 **HANDLER** 범위 모두에서 속성에 액세스할 수 있습니다.

애플리케이션의 메시지 컨텍스트와 마찬가지로 논리 메시지 컨텍스트는 **Java Map**의 하위 클래스입니다. 컨텍스트에 저장된 속성에 액세스하려면 **get()** 메서드 및 **put()** 메서드를 사용하여 맵 인터페이스에서

상속됩니다.

기본적으로 논리 처리기 내에서 메시지 컨텍스트에서 설정한 모든 속성에는 **HANDLER**의 범위가 할당됩니다.**By default, any properties you set in the message context from inside a logical handler are assigned a scope of HANDLER.** 애플리케이션 코드가 컨텍스트의 **setScope()** 메서드를 사용하여 속성 범위를 **APPLICATION**으로 명시적으로 설정하는 데 필요한 속성에 액세스할 수 있도록 하려면.

메시지 컨텍스트에서 속성 작업에 대한 자세한 내용은 **42.1**절. “컨텍스트 이해” 을 참조하십시오.

43.3.6. 메시지의 방향을 결정하는 방법How to determine the direction of the message

메시지가 처리기 체인을 통과하는 방향을 아는 것이 종종 중요합니다. 예를 들어 들어오는 요청에서 보안 토큰을 검색하고 보안 토큰을 나가는 응답에 연결하려고 합니다.

메시지의 방향은 메시지 컨텍스트의 아웃바운드 메시지 속성에 저장됩니다. 예 **43.6. “SOAP 메시지 컨텍스트에서 메시지 직접 가져오기”** 에 표시된 대로 **MessageContext.MESSAGE_OUTBOUND_PROPERTY** 키를 사용하여 메시지 컨텍스트에서 아웃바운드 메시지 속성을 검색합니다.

예 43.6. SOAP 메시지 컨텍스트에서 메시지 직접 가져오기

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

속성은 부울 오브젝트로 저장됩니다. 개체의 부울**Value()** 메서드를 사용하여 속성 값을 확인할 수 있습니다. 속성이 **true**로 설정되면 메시지는 아웃바운드입니다.**If the property is set to true, the message is outbound.** 속성이 **false**로 설정되면 메시지가 인바운드됩니다.**If the property is set to false, the message is inbound.**

43.3.7. 반환 값 확인

handleMessage() 메서드가 메시지 처리를 완료하는 방법은 메시지 처리가 진행되는 방식에 직접적인 영향을 미칩니다. 다음 작업 중 하나를 수행하여 완료할 수 있습니다.

1. 메시지 처리가 정상적으로 계속되어야 하는 **Apache CXF** 런타임으로 **true** 반환 **true** 신호를 반환합니다. 다음 처리기에 해당 **handleMessage()** 가 호출되었습니다.

2.

*일반적인 메시지 처리가 중지되어야 하는 **Apache CXF** 런타임으로 **false-Returning false** 신호를 반환합니다. 런타임이 진행되는 방식은 현재 메시지에 사용되는 메시지 교환 패턴에 따라 다릅니다.*

요청-응답 메시지 교환의 경우 다음과 같은 일이 발생합니다.

a. *메시지 처리의 방향은 반전됩니다.*

예를 들어 서비스 공급자가 요청을 처리하는 경우 메시지는 서비스의 구현 오브젝트로 진행되지 않습니다. 대신 요청을 시작한 소비자로의 반환을 위해 바인딩으로 다시 전송됩니다.

b. *새 처리 방향에서 처리기 체인을 따라 있는 모든 메시지 처리기에는 체인에 있는 순서대로 **handleMessage()** 메서드가 호출됩니다.*

c. *메시지가 처리기 체인의 끝에 도달하면 디스패치됩니다.*

단방향 메시지 교환의 경우 다음과 같은 일이 발생합니다.

d. *메시지 처리가 중지됩니다.*

e. *이전에 호출한 모든 메시지 핸들러에는 **close()** 메서드가 호출되었습니다.*

f. *메시지가 디스패치되어 있습니다.*

3. ***ProtocolException** 예외를 **throw - ProtocolException** 예외 또는 이 예외의 하위 클래스를 **throw**하면 오류 메시지 처리가 시작되는 **Apache CXF** 런타임이 시작됩니다. 런타임이 진행되는 방식은 현재 메시지에 사용되는 메시지 교환 패턴에 따라 다릅니다.*

요청-응답 메시지 교환의 경우 다음과 같은 일이 발생합니다.

a. *처리기가 오류 메시지를 아직 생성하지 않은 경우 런타임은 메시지를 오류 메시지로 래핑합니다.*

b.
메시지 처리의 방향은 반전됩니다.

예를 들어 서비스 공급자가 요청을 처리하는 경우 메시지는 서비스의 구현 오브젝트로 진행되지 않습니다. 대신 요청을 시작한 소비자로의 반환을 위해 바인딩으로 다시 전송됩니다.

c.
*새 처리 방향에서 처리기 체인을 따라 있는 모든 메시지 처리기에는 체인에 상주하는 순서대로 **handleFault()** 메서드가 호출됩니다.*

d.
오류 메시지가 처리기 체인의 끝에 도달하면 디스패치됩니다.

단방향 메시지 교환의 경우 다음과 같은 일이 발생합니다.

e.
처리기가 오류 메시지를 아직 생성하지 않은 경우 런타임은 메시지를 오류 메시지로 래핑합니다.

f.
메시지 처리가 중지됩니다.

g.
*이전에 호출한 모든 메시지 핸들러에는 **close()** 메서드가 호출되었습니다.*

h.
오류 메시지가 디스패치되어 있습니다.

4.
*다른 런타임 예외를 **throw - ProtocolException** 예외 이외의 런타임 예외는 메시지 처리가 중지되는 **Apache CXF** 런타임을 알리는 신호입니다. 이전에 호출한 모든 메시지 핸들러에는 **close()** 메서드가 호출되고 예외가 디스패치됩니다. 메시지가 요청 응답 메시지 교환의 일부인 경우 요청이 발생한 소비자에게 반환되도록 예외가 디스패치됩니다.*

43.3.8. 예제

*예 43.7. "논리 메시지 핸들러 메시지 처리" 서비스 소비자가 사용하는 논리 메시지 처리기에 대한 **handleMessage()** 메시지의 구현을 보여줍니다. 요청을 서비스 공급자에게 전송되기 전에 처리합니다.*

*예 **43.7.** 논리 메시지 핸들러 메시지 처리*

```
public class SmallNumberHandler implements LogicalHandler<LogicalMessageContext>
{
```

```
    public final boolean handleMessage(LogicalMessageContext messageContext)
    {
        try
        {
            boolean outbound =
(Boolean)messageContext.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

            if (outbound)
            {
                LogicalMessage msg = messageContext.getMessage();

                JAXBContext jaxbContext = JAXBContext.newInstance(ObjectFactory.class);
                Object payload = msg.getPayload(jaxbContext);
                if (payload instanceof JAXBElement)
                {
                    payload = ((JAXBElement)payload).getValue();
                }

                if (payload instanceof AddNumbers)
                {
                    AddNumbers req = (AddNumbers)payload;

                    int a = req.getArg0();
                    int b = req.getArg1();
                    int answer = a + b;

                    if (answer < 20)
                    {
                      AddNumbersResponse resp = new AddNumbersResponse();
                      resp.setReturn(answer);
                      msg.setPayload(new ObjectFactory().createAddNumbersResponse(resp),
                                     jaxbContext);

                      return false;
                    }
                }
                else
                {
                  throw new WebServiceException("Bad Request");
                }
            }
            return true;
        }
        catch (JAXBException ex)
        {
            throw new ProtocolException(ex);
        }
    }
...
}
```

예 43.7. "논리 메시지 핸들러 메시지 처리"의 코드는 다음을 수행합니다.

메시지가 아웃바운드 요청인지 확인합니다.

메시지가 아웃바운드 요청인 경우 처리기는 추가 메시지 처리를 수행합니다.

*메시지 컨텍스트에서 메시지 페이로드의 **LogicalMessage** 표현을 가져옵니다.*

***JAXB** 오브젝트로 실제 메시지 페이로드를 가져옵니다.*

요청이 올바른 유형인지 확인합니다.

이 경우 핸들러는 메시지를 계속 처리합니다.

합계의 값을 확인합니다.

*임계값 **20**보다 작으면 응답을 작성하고 클라이언트에 반환합니다.*

응답을 빌드합니다.

*메시지 처리를 중지하고 클라이언트에 대한 응답을 반환하는 **false**를 반환합니다.*

*메시지가 올바른 유형이 아닌 경우 런타임 예외를 **throw**합니다.*

이 예외는 클라이언트에 반환됩니다.

*메시지가 인바운드 응답이거나 합계가 임계값을 충족하지 않으면 **true**를 반환합니다.**Return true if the message is an inbound response or the sum does not meet the threshold.***

메시지 처리는 정상적으로 계속됩니다.

JAXB 마샬링 오류가 발생하는 경우 ProtocolException을 throw합니다.

현재 처리기와 클라이언트 간의 처리기의 handleFault() 메서드에서 처리한 후 예외는 클라이언트에 다시 전달됩니다.

43.4. 프로토콜 핸들러 구현

43.4.1. 개요

프로토콜 핸들러는 사용 중인 프로토콜에 따라 다릅니다. Apache CXF는 JAX-WS에 지정된 대로 SOAP 프로토콜 핸들러를 제공합니다. SOAP 프로토콜 핸들러는 javax.xml.ws.handler.soap.SOAPHandler 인터페이스를 구현합니다.

예 43.8. "SOAPHandler Synopsis" 에 표시된 SOAPHandler 인터페이스는 메시지에 대한 액세스를 SOAPMessage 개체로 제공하는 SOAP 특정 메시지 컨텍스트를 사용합니다. 또한 SOAP 헤더에 액세스할 수 있습니다.

예 43.8. SOAPHandler Synopsis

```
public interface SOAPHandler extends Handler
{
  boolean handleMessage(SOAPMessageContext context);
  boolean handleFault(SOAPMessageContext context);
  void close(SOAPMessageContext context);
  Set<QName> getHeaders()
}
```

SOAP 특정 메시지 컨텍스트를 사용하는 것 외에도 SOAP 프로토콜 핸들러에는 getHeaders() 라는 추가 메서드를 구현해야 합니다. 이 추가 방법은 처리기가 처리할 수 있는 헤더의 QNames를 반환합니다.

43.4.2. 절차

논리 핸드러를 구현하려면 다음을 수행합니다.

1. 처리기에 필요한 모든 43.6절. "핸들러 초기화" 논리를 구현합니다.

2. 43.5절. "SOAP 핸들러에서 메시지 처리" 논리를 구현합니다.

3. 43.7절. "자주하는 질문" 논리를 구현합니다.

4. getHeaders() 메서드를 구현합니다.

5. 완료되면 43.8절. "핸들러 종료" 처리기에 대한 논리를 구현합니다.

6. 43.9절. "핸들러 릴리스" 의 논리를 실행하기 전에 처리기의 리소스를 삭제합니다.

43.4.3. getHeaders() 메서드 구현

예 43.9. "SOAP.getHeaders() 메서드" 에 표시된 getHeaders() 는 Apache CXF 런타임에서 처리기가 처리할 책임이 있는 SOAP 헤더를 보여줍니다. 처리기에서 이해할 수 있는 각 SOAP 헤더의 외부 요소의 QNames를 반환합니다.Returns the QNames of the outer element of each SOAP header the handler understands.

> 예 43.9. SOAP.getHeaders() 메서드
>
> ```
> Set<QName>getHeaders
> ```

대부분의 경우 null을 반환하는 것만으로 충분합니다. 그러나 애플리케이션에서 SOAP 헤더의 mustUnderstand 특성을 사용하는 경우 애플리케이션의 SOAP 핸들러에서 이해하는 헤더를 지정하는 것이 중요합니다. 런타임은 등록된 모든 핸들러가 true 로 설정된 mustUnderstand 특성이 있는 헤더 목록에 대해 이해하는 SOAP 헤더 세트를 확인합니다. 플래그가 지정된 헤더 중 하나라도 이해된 헤더 목록에 없는 경우 런타임은 메시지를 거부하고 SOAP에서 예외를 이해해야 합니다.

43.5. SOAP 핸들러에서 메시지 처리

43.5.1. 개요

일반 메시지 처리는 **handleMessage()** 메서드에서 처리됩니다.

handleMessage() 메서드는 메시지 본문에 대한 액세스를 **SOAPMessage** 개체 및 메시지와 연결된 **SOAP** 헤더로 제공하는 **SOAPMessageContext** 개체를 수신합니다. 또한 컨텍스트는 메시지 컨텍스트에 저장된 모든 속성에 대한 액세스를 제공합니다.

handleMessage() 메서드는 메시지 처리를 계속하는 방법에 따라 **true** 또는 **false**를 반환합니다. 또한 예외를 **throw**할 수 있습니다.

43.5.2. 메시지 본문으로 작업

SOAP 메시지 컨텍스트의 **getMessage()** 메서드를 사용하여 **SOAP** 메시지를 가져올 수 있습니다. 메시지를 라이브 **SOAPMessage** 개체로 반환합니다. 처리기에서 메시지의 변경 사항은 컨텍스트에 저장된 메시지에 자동으로 반영됩니다.

기존 메시지를 새 메시지로 교체하려면 컨텍스트의 **setMessage()** 메서드를 사용할 수 있습니다. **setMessage()** 메서드는 **SOAPMessage** 개체를 사용합니다.

43.5.3. SOAP 헤더를 가져옵니다.Gets the SOAP headers

SOAPMessage 개체의 **getHeader()** 메서드를 사용하여 **SOAP** 메시지의 헤더에 액세스할 수 있습니다. 그러면 처리하려는 헤더 요소를 찾기 위해 검사해야 하는 **SOAPHeader** 개체로 **SOAP** 헤더가 반환됩니다.

SOAP 메시지 컨텍스트는 예 43.10. “SOAPMessageContext.getHeaders() 메서드”에 표시된 **getHeaders()** 메서드를 제공합니다. 이 메서드는 지정된 **SOAP** 헤더에 대해 **JAXB** 개체를 포함하는 배열을 반환합니다.

> 예 43.10. SOAPMessageContext.getHeaders() 메서드
>
> Ojbect[]getHeadersQName헤더JAXBContext컨텍스트부울allRoles

요소의 **QName**을 사용하여 헤더를 지정합니다. **allRoles** 매개변수를 **false**로 설정하여 반환된 헤더를 추가로 제한할 수 있습니다. 즉, 런타임에서 활성 **SOAP** 역할에 적용할 수 있는 **SOAP** 헤더만 반환하도록 지시합니다.

헤더가 없으면 메서드가 빈 배열을 반환합니다.

JAXBContext 오브젝트를 인스턴스화하는 방법에 대한 자세한 내용은 *39장. A JAXBContext 오브젝트 사용* 을 참조하십시오.

43.5.4. 컨텍스트 속성 작업

논리 처리기에 전달된 **SOAP** 메시지 컨텍스트는 애플리케이션의 메시지 컨텍스트 인스턴스이며 이 컨텍스트에 저장된 모든 속성에 액세스할 수 있습니다. 핸들러는 **APPLICATION** 범위와 **Handler** 범위 모두에서 속성에 액세스할 수 있습니다.

애플리케이션의 메시지 컨텍스트와 마찬가지로 **SOAP** 메시지 컨텍스트는 **Java Map**의 하위 클래스입니다. 컨텍스트에 저장된 속성에 액세스하려면 **get()** 메서드 및 **put()** 메서드를 사용하여 맵 인터페이스에서 상속됩니다.

기본적으로 논리 핸들러 내부에서 컨텍스트에서 설정한 모든 속성에는 **HANDLER** 의 범위가 할당됩니다. 애플리케이션 코드가 컨텍스트의 **setScope()** 메서드를 사용하여 속성 범위를 **APPLICATION**으로 명시적으로 설정하는 데 필요한 속성에 액세스할 수 있도록 하려면.

메시지 컨텍스트에서 속성 작업에 대한 자세한 내용은 *42.1절. “컨텍스트 이해”* 을 참조하십시오.

43.5.5. 메시지의 방향을 결정하는 방법How to determine the direction of the message

메시지가 처리기 체인을 통과하는 방향을 아는 것이 종종 중요합니다. 예를 들어 들어오는 메시지에서 나가는 메시지와 **strip** 헤더에 헤더를 추가하려고 합니다.

메시지의 방향은 메시지 컨텍스트의 아웃바운드 메시지 속성에 저장됩니다. *예 43.11. “SOAP 메시지 컨텍스트에서 메시지 직접 가져오기”* 에 표시된 대로 **MessageContext.MESSAGE_OUTBOUND_PROPERTY** 키를 사용하여 메시지 컨텍스트에서 아웃바운드 메시지 속성을 검색합니다.

예 43.11. SOAP 메시지 컨텍스트에서 메시지 직접 가져오기

```
Boolean outbound;
outbound = (Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
```

속성은 부울 오브젝트로 저장됩니다. 개체의 부울Value() 메서드를 사용하여 속성 값을 확인할 수 있습니다. 속성이 true로 설정되면 메시지는 아웃바운드입니다.If the property is set to true, the message is outbound. 속성이 false로 설정되면 메시지가 인바운드됩니다.If the property is set to false, the message is inbound.

43.5.6. 반환 값 확인

handleMessage() 메서드가 메시지 처리를 완료하는 방법은 메시지 처리가 진행되는 방식에 직접적인 영향을 미칩니다. 다음 작업 중 하나를 수행하여 완료할 수 있습니다.

1. *메시지 처리가 정상적으로 계속되어야 하는 Apache CXF 런타임으로 true 반환 true 신호를 반환합니다. 다음 처리기에 해당 handleMessage() 가 호출되었습니다.*

2. *일반적인 메시지 처리가 중지되는 Apache CXF 런타임으로 false-Returning false 신호를 반환합니다. 런타임이 진행되는 방식은 현재 메시지에 사용되는 메시지 교환 패턴에 따라 다릅니다.*

   요청-응답 메시지 교환의 경우 다음과 같은 일이 발생합니다.

   a. *메시지 처리의 방향은 반전됩니다.*

      예를 들어 서비스 공급자가 요청을 처리하는 경우 메시지가 서비스의 구현 오브젝트로 진행되지 않습니다. 대신 요청을 시작한 소비자로의 반환을 위해 바인딩으로 다시 전송됩니다.

   b. *새 처리 방향에서 처리기 체인을 따라 있는 모든 메시지 처리기에는 체인에 있는 순서대로 handleMessage() 메서드가 호출됩니다.*

   c. *메시지가 처리기 체인의 끝에 도달하면 디스패치됩니다.*

      단방향 메시지 교환의 경우 다음과 같은 일이 발생합니다.

   d. *메시지 처리가 중지됩니다.*

   e. *이전에 호출한 모든 메시지 핸들러에는 close() 메서드가 호출되었습니다.*

f.

메시지가 디스패치되어 있습니다.

3.

***ProtocolException** 예외- **protocolException** 예외 또는 이 예외의 하위 클래스를 **throw**하여 오류 메시지 처리가 시작되는 **Apache CXF** 런타임을 신호합니다. 런타임이 진행되는 방식은 현재 메시지에 사용되는 메시지 교환 패턴에 따라 다릅니다.*

요청-응답 메시지 교환의 경우 다음과 같은 일이 발생합니다.

a.

처리기가 오류 메시지를 아직 생성하지 않은 경우 런타임은 메시지를 오류 메시지로 래핑합니다.

b.

메시지 처리의 방향은 반전됩니다.

예를 들어 서비스 공급자가 요청을 처리하는 경우 메시지가 서비스의 구현 오브젝트로 진행되지 않습니다. 요청이 발생한 소비자의 반환을 위해 바인딩으로 다시 전송됩니다.

c.

*새 처리 방향에서 처리기 체인을 따라 있는 모든 메시지 처리기에는 체인에 상주하는 순서대로 **handleFault()** 메서드가 호출됩니다.*

d.

오류 메시지가 처리기 체인의 끝에 도달하면 디스패치됩니다.

단방향 메시지 교환의 경우 다음과 같은 일이 발생합니다.

e.

처리기가 오류 메시지를 아직 생성하지 않은 경우 런타임은 메시지를 오류 메시지로 래핑합니다.

f.

메시지 처리가 중지됩니다.

g.

*이전에 호출한 모든 메시지 핸들러에는 **close()** 메서드가 호출되었습니다.*

h.

오류 메시지가 디스패치되어 있습니다.

4.

*다른 런타임 예외를 **throw - ProtocolException** 예외 이외의 런타임 예외는 메시지 처리가*

중지되는 **Apache CXF** 런타임을 알리는 신호입니다. 이전에 호출한 모든 메시지 핸들러에는 **close()** 메서드가 호출되고 예외가 디스패치됩니다. 메시지가 요청-응답 메시지의 일부이면 요청을 시작한 소비자에게 반환되도록 예외를 디스패치합니다.

43.5.7. 예제

예 43.12. "SOAP 핸들러에서 메시지 처리" shows a handleMessage() implementation that prints the SOAP message to the screen.

예 43.12. SOAP 핸들러에서 메시지 처리

```
public boolean handleMessage(SOAPMessageContext smc)
{
  PrintStream out;

  Boolean outbound =
(Boolean)smc.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

  if (outbound.booleanValue())
  {
    out.println("\nOutbound message:");
  }
  else
  {
    out.println("\nInbound message:");
  }

  SOAPMessage message = smc.getMessage();

  message.writeTo(out);
  out.println();

  return true;
}
```

예 43.12. "SOAP 핸들러에서 메시지 처리" 의 코드는 다음을 수행합니다.

메시지 컨텍스트에서 아웃바운드 속성을 검색합니다.

메시지 방향을 테스트하고 적절한 메시지를 출력합니다.

컨텍스트에서 **SOAP** 메시지를 검색합니다.

콘솔에 메시지를 출력합니다.

43.6. 핸들러 초기화

43.6.1. 개요

런타임에서 처리기의 인스턴스를 생성하면 핸드너가 메시지를 처리하는 데 필요한 모든 리소스를 생성합니다. 핸들러의 생성자에 이 작업을 수행하기 위한 모든 논리를 배치할 수 있지만 가장 적절한 위치가 아닐 수도 있습니다. 처리기 프레임워크는 핸들러를 인스턴스화할 때 여러 선택적 단계를 수행합니다. 선택적 단계 중에 실행될 리소스 주입 및 기타 초기화 논리를 추가할 수 있습니다.

처리기에 대한 초기화 메서드를 제공할 필요가 없습니다.

43.6.2. 초기화 순서

Apache CXF 런타임은 다음과 같은 방식으로 핸들러를 초기화합니다.

1. 핸들러의 생성자를 호출합니다.
2. **@Resource** 주석에서 지정하는 모든 리소스가 삽입됩니다.
3. **@PostConstruct** 주석으로 데코딩된 메서드는 이름이 있는 경우 입니다.

참고

@PostConstruct 주석으로 장식되는 메서드에는 **void** 반환 유형이 있어야 하며 매개변수가 없어야 합니다.

4. 처리기는 **Ready** 상태에 있습니다.

43.7. 자주하는 질문

43.7.1. 개요

핸들러는 메시지 처리 중에 **ProtocolException** 예외가 **throw**될 때 오류 메시지를 처리하는 데 **handleFault()** 메서드를 사용합니다.

handleFault() 메서드는 처리기 유형에 따라 **LogicalMessageContext** 개체 또는 **SOAPMessageContext** 개체를 수신합니다. 수신된 컨텍스트는 핸들러의 메시지 페이로드에 대한 구현 액세스를 제공합니다.

handleFault() 메서드는 오류 메시지 처리 방법에 따라 **true** 또는 **false**를 반환합니다. 또한 예외를 **throw**할 수 있습니다.

43.7.2. 메시지 페이로드 가져오기

handleFault() 메서드에서 받은 컨텍스트 오브젝트는 **handleMessage()** 메서드에서 수신한 오브젝트와 유사합니다. 컨텍스트의 **getMessage()** 메서드를 사용하여 동일한 방식으로 메시지 페이로드에 액세스합니다. 유일한 차이점은 컨텍스트에 포함된 페이로드입니다.

LogicalMessageContext 사용에 대한 자세한 내용은 **43.3절. "논리 핸들러에서 메시지 처리"** 을 참조하십시오.

SOAPMessageContext 사용에 대한 자세한 내용은 **43.5절. "SOAP 핸들러에서 메시지 처리"** 을 참조하십시오.

43.7.3. 반환 값 확인

handleFault() 메서드가 메시지 처리를 완료하는 방법은 메시지 처리가 진행되는 방식에 직접적인 영향을 미칩니다. 다음 작업 중 하나를 수행하여 완료합니다.

return true

오류 처리가 정상적으로 계속되어야 한다는 실제 신호를 반환합니다. 체인에서 다음 처리기의 **handleFault()** 메서드가 호출됩니다.

return false

오류 처리가 중지되는 잘못된 신호를 반환합니다. 현재 메시지를 처리하는 데 호출된 핸들러의 **close()** 메서드가 호출되고 오류 메시지가 디스패치됩니다.

예외를 throw

예외를 **throw**하면 오류 메시지 처리가 중지됩니다. 현재 메시지를 처리하는 데 호출된 핸들러의 **close()** 메서드가 호출되고 예외가 디스패치됩니다.

43.7.4. 예제

예 43.13. "메시지 핸들러에서 Fault 처리" 는 메시지 본문을 화면에 출력하는 **handleFault()** 의 구현을 보여줍니다.

예 43.13. 메시지 핸들러에서 Fault 처리

```
public final boolean handleFault(LogicalMessageContext messageContext)
{
  System.out.println("handleFault() called with message:");

  LogicalMessage msg=messageContext.getMessage();
  System.out.println(msg.getPayload());

  return true;
}
```

43.8. 핸들러 종료

처리기 체인이 메시지 처리를 완료하면 런타임에서 실행된 각 핸들러의 **close()** 메서드를 호출합니다. 이는 메시지 처리 중에 처리기에서 사용한 리소스를 정리하거나 속성을 기본 상태로 재설정하는 데 적절한 위치입니다.

리소스가 단일 메시지 교환 이상으로 유지되어야 하는 경우 처리기의 **close()** 메서드 중에 해당 리소스를 정리하지 않아야 합니다.

43.9. 핸들러 릴리스

43.9.1. 개요

런타임은 처리기가 바인딩된 서비스 또는 서비스 프록시를 종료할 때 핸들러를 해제합니다. 런타임은 핸들러의 종료자를 호출하기 전에 선택적 **release** 메서드를 호출합니다. 이 선택적 릴리스 방법은 처리기에서 사용하는 리소스를 해제하거나 처리기의 종료자에 적합하지 않은 다른 작업을 수행하는 데 사용할 수 있습니다.

처리기에 대한 정리 메서드를 제공할 필요가 없습니다.

43.9.2. 릴리스 순서

처리기가 해제되면 다음이 수행됩니다.

1. 핸들러는 활성 메시지 처리를 완료합니다.
2. 런타임은 **@PreDestroy** 주석을 사용하여 데코딩 메서드를 호출합니다.

   이 방법은 핸들러에서 사용하는 리소스를 모두 정리해야 합니다.
3. 처리기의 종료자는 이라고 합니다.

43.10. 핸들러를 사용하도록 끝점 구성

43.10.1. 프로그래밍 방식 설정

43.10.1.1. 소비자에 핸들러 체인 추가

43.10.1.1.1. 개요

소비자에 처리기 체인을 추가하면 핸들러 체인을 명시적으로 빌드해야 합니다. 그런 다음 서비스 프록시의 **Binding** 오브젝트에 핸들러 체인을 직접 설정합니다.

중요

Spring 구성을 사용하여 구성된 모든 처리기 체인은 프로그래밍 방식으로 구성된 처리기 체인을 재정의합니다.

43.10.1.1.2. 절차

고객에게 처리기 체인을 추가하려면 다음을 수행합니다.

1. *처리기 체인을 보유 하는 **List<Handler >** 오브젝트를 생성합니다.*

2. *체인에 추가될 각 핸들러의 인스턴스를 생성합니다.*

3. *런타임에 각 인스턴스화된 처리기 개체를 목록에 추가합니다.**Add each of the instantiated handler objects to the list in the order they are to be invoked by the runtime.***

4. *서비스 프록시에서 바인딩 오브젝트를 가져옵니다.*

   ***Apache CXF**는 **org.apache.cxf.jaxws.binding.DefaultBindingImpl** 이라는 바인딩 인터페이스 구현을 제공합니다.*

5. ***Binding** 개체의 **setHandlerChain()** 메서드를 사용하여 프록시에 처리기 체인을 설정합니다.*

43.10.1.1.3. 예제

예 43.14. "소비자에 핸들러 체인 추가" 소비자에 처리기 체인을 추가하기 위한 코드를 보여줍니다.

예 43.14. 소비자에 핸들러 체인 추가

```
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;
import java.util.ArrayList;
import java.util.List;

import org.apache.cxf.jaxws.binding.DefaultBindingImpl;
...
SmallNumberHandler sh = new SmallNumberHandler();
List<Handler> handlerChain = new ArrayList<Handler>();
handlerChain.add(sh);

DefaultBindingImpl binding = ((BindingProvider)proxy).getBinding();
binding.getBinding().setHandlerChain(handlerChain);
```

예 43.14. "소비자에 핸들러 체인 추가" 의 코드는 다음을 수행합니다.

핸들러를 인스턴스화합니다.

***List** 오브젝트를 생성하여 체인을 유지합니다.*

체인에 처리기를 추가합니다.

*프록시에서 **DefaultBindingImpl** 오브젝트로 바인딩 오브젝트를 가져옵니다.*

처리기 체인을 프록시의 바인딩에 할당합니다.

43.10.1.2. 서비스 공급자에 핸들러 체인 추가

43.10.1.2.1. 개요

***SEI** 또는 구현 클래스를 **@HandlerChain** 주석으로 위임하여 서비스 공급자에 처리기 체인을 추가합니다. 주석은 서비스 공급자가 사용하는 핸들러 체인을 정의하는 메타 데이터 파일을 가리킵니다.*

43.10.1.2.2. 절차

서비스 공급자에 처리기 체인을 추가하려면 다음을 수행합니다.

1. ***@HandlerChain** 주석을 사용하여 공급자의 구현 클래스를 장식합니다.*
2. *처리기 체인을 정의하는 처리기 구성 파일을 생성합니다.*

43.10.1.2.3. @HandlerChain 주석

***javax.jws.HandlerChain** 주석은 서비스 공급자의 구현 클래스를 장식합니다. 런타임에서 파일 속성으로 지정된 처리기 체인 구성 파일 을 로드하도록 지시합니다.*

*주석의 **file** 속성은 로드할 처리기 구성 파일을 식별하는 두 가지 방법을 지원합니다.*

- *a URL*
- *상대 경로 이름*

예 43.15. "Handler Chain을 로드하는 서비스 구현" handlers.xml 이라는 파일에 정의된 핸들러 체인을 사용할 서비스 공급자 구현을 보여줍니다. handlers.xml 은 서비스 공급자가 실행되는 디렉터리에 있어야 합니다.

예 43.15. Handler Chain을 로드하는 서비스 구현

```
import javax.jws.HandlerChain;
import javax.jws.WebService;
...

@WebService(name = "AddNumbers",
            targetNamespace = "http://apache.org/handlers",
            portName = "AddNumbersPort",
            endpointInterface = "org.apache.handlers.AddNumbers",
            serviceName = "AddNumbersService")
@HandlerChain(file = "handlers.xml")
public class AddNumbersImpl implements AddNumbers
{
...
}
```

43.10.1.2.4. 처리기 구성 파일

처리기 구성 파일은 JSR 109(Java EE, 버전 1.2용 웹 서비스)의 XML 문법을 사용하여 처리기 체인을 정의합니다. 이 문법은 http://java.sun.com/xml/ns/javaee 에 정의되어 있습니다.

처리기 구성 파일의 루트 요소는 handler-chains 요소입니다. handler-chains 요소에는 하나 이상의 handler-chain 요소가 있습니다.

handler-chain 요소는 처리기 체인을 정의합니다. 표 43.1. "Server-Side Chain을 정의하는 데 사용되는 요소" handler-chain 요소의 하위 요소에 대해 설명합니다.

표 43.1. Server-Side Chain을 정의하는 데 사용되는 요소

요소	설명
handler	처리기를 설명하는 요소를 포함합니다.

요소	설명
service-name-pattern	처리기 체인이 바인딩된 서비스를 정의하는 WSDL **서비스** 요소의 QName을 지정합니다.Specifies the QName of the WSDL service element defining the service to which the handler chain is bound. QName을 정의할 때 * 를 와일드카드로 사용할 수 있습니다.
port-name-pattern	처리기 체인이 바인딩되는 끝점을 정의하는 WSDL **포트** 요소의 QName을 지정합니다. QName을 정의할 때 * 를 와일드카드로 사용할 수 있습니다.
protocol-binding	처리기 체인이 사용되는 메시지 바인딩을 지정합니다. 바인딩은 URI로 지정되거나 다음과 같은 별칭 중 하나로 지정됩니다. **#\#SOAP11_HTTP**,**\#SOAP11_HTTP_MTOM**,**\#SOAP12_HTTP_**MTOM ,**\## SOAP12_HTTP_MTOM** 또는 **\#\#XML_HTTP**. 메시지 바인딩 URI에 대한 자세한 내용은 23장. *Apache CXF 바인딩 ID* 을 참조하십시오.

***handler-chain** 요소는 단일 **handler** 요소를 자식으로 사용하는 경우에만 필요합니다. 그러나 전체 처리기 체인을 정의하는 데 필요한 만큼의 처리기 요소를 지원할 수 있습니다. 체인의 핸들러는 처리기 체인 정의에서 지정된 순서대로 실행됩니다.*

중요

실행의 최종 순서는 지정된 핸들러를 논리 처리기 및 프로토콜 처리기로 정렬하여 결정됩니다. 그룹화 내에서 구성에 지정된 순서가 사용됩니다.

***protocol-binding** 과 같은 다른 하위 항목은 정의된 처리기 체인의 범위를 제한하는 데 사용됩니다. 예를 들어 **service-name-pattern** 요소를 사용하는 경우 처리기 체인은 지정한 **WSDL** 서비스 요소의 하위인 서비스 공급자에만 연결됩니다. 처리기 요소에서 이러한 제한 자식 중 하나만 사용할 수 있습니다.*

***handler** 요소는 처리기 체인에서 개별 핸들러를 정의합니다. 해당 **handler-class** 하위 요소는 처리기를 구현하는 클래스의 정규화된 이름을 지정합니다. **handler** 요소에는 처리기의 고유한 이름을 지정하는 선택적 **handler-name** 요소가 있을 수도 있습니다.*

예 43.16. "처리기 구성 파일" 단일 처리기 체인을 정의하는 핸들러 구성 파일을 보여줍니다. 체인은 두 개의 핸들러로 구성됩니다.

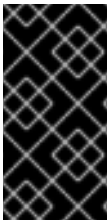
예 43.16. 처리기 구성 파일

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
    <handler>
      <handler-name>LoggingHandler</handler-name>
      <handler-class>demo.handlers.common.LoggingHandler</handler-class>
    </handler>
    <handler>
      <handler-name>AddHeaderHandler</handler-name>
      <handler-class>demo.handlers.common.AddHeaderHandler</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

43.10.2. Spring 설정

43.10.2.1. 개요

처리기 체인을 사용하도록 엔드포인트를 구성하는 가장 쉬운 방법은 끝점 구성에서 체인을 정의하는 것입니다. 이 작업은 **jaxwxs:handlers** 하위를 끝점을 구성하는 요소에 추가하여 수행됩니다.



중요

구성 파일을 통해 추가된 처리기 체인이 프로그래밍 방식으로 구성된 처리기 체인보다 우선합니다.

43.10.2.2. 절차

처리기 체인을 로드하도록 끝점을 구성하려면 다음을 수행합니다.

1. 끝점에 구성 요소가 없는 경우 하나를 추가합니다.

   Apache CXF 엔드포인트 구성에 대한 자세한 내용은 *17장. JAX-WS 엔드포인트 구성* 을 참조하십시오.

2. **jaxws:handlers** 하위 요소를 엔드포인트의 구성 요소에 추가합니다.

3. 체인의 각 처리기에 대해 처리기를 구현하는 클래스를 지정하는 빈 요소를 추가합니다.

   처리기 구현을 두 개 이상의 위치에서 사용하는 경우 **ref** 요소를 사용하여 빈 요소를 참조할 수 있습니다.

43.10.2.3. handlers 요소

jaxws:handlers 요소는 끝점의 구성에 처리기 체인을 정의합니다. 모든 **JAX-WS** 엔드포인트 구성 요소에 자식으로 표시될 수 있습니다. 다음은 다음과 같습니다.

- **jaxws:endpoint** 는 서비스 공급자를 구성합니다.
- **jaxws:server** 도 서비스 공급자를 구성합니다.
- **jaxws:client** 는 서비스 소비자를 구성합니다.

다음 두 가지 방법 중 하나로 처리기 체인에 핸들러를 추가합니다.

- 구현 클래스를 정의 하는 빈 요소 추가
- **ref** 요소를 사용하여 구성 파일의 다른 위치에서 이름이 지정 된 빈 요소를 참조합니다.

핸들러가 구성에 정의된 순서는 실행할 순서입니다. 논리 처리기와 프로토콜 처리기를 혼합하면 순서가 수정될 수 있습니다. 런타임은 구성에 지정된 기본 순서를 유지하면서 이를 적절한 순서로 정렬합니다.

43.10.2.4. 예제

예 43.17. “Handler Chain In Spring을 사용하도록 엔드 포인트 구성” 핸들러 체인을 로드하는 서비스 공급자의 구성을 보여줍니다.

예 43.17. Handler Chain In Spring을 사용하도록 엔드 포인트 구성

```
<beans ...
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  ...
```

```
  schemaLocation="...
    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
    ...">
  <jaxws:endpoint id="HandlerExample"
                  implementor="org.apache.cxf.example.DemoImpl"
                  address="http://localhost:8080/demo">
    <jaxws:handlers> <bean class="demo.handlers.common.LoggingHandler" /> <bean
class="demo.handlers.common.AddHeaderHandler" /> </jaxws:handlers>
  </jaws:endpoint>
</beans>
```

44장. MAVEN 툴 참조

44.1. 플러그인 설정

초록

Apache CXF 플러그인을 사용하려면 먼저 POM에 적절한 종속 항목과 리포지토리를 추가해야 합니다.

44.1.1. 종속 항목

프로젝트의 **POM**에 다음 종속 항목을 추가해야 합니다.

- **JAX-WS frontend**

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-frontend-jaxws</artifactId>
  <version>version</version>
</dependency>
```

- **HTTP 전송**

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http</artifactId>
  <version>version</version>
</dependency>
```

- **Undertow 전송**

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-undertow</artifactId>
  <version>version</version>
</dependency>
```

44.2. CXF-CODEGEN-PLUGIN

초록

WSDL 문서에서 JAX-WS 호환 Java 코드 생성

44.2.1. 개요

44.2.2. 기본 예

다음 POM 추출은 myService.wsdl WSDL 파일을 처리하기 위해 Maven cxf-codegen-plugin 을 구성하는 방법에 대한 간단한 예를 보여줍니다.

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>3.3.6.fuse-7_11_1-00015-redhat-00002</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>target/generated/src/main/java</sourceRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>src/main/resources/wsdl/myService.wsdl</wsdl>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

44.2.3. 기본 구성 설정

이전 예제에서는 다음 구성 설정을 사용자 지정할 수 있습니다.

configuration/sourceRoot

생성된 Java 파일이 저장될 디렉터리를 지정합니다. 기본값은 target/generated-sources/cxf 입니다.

configuration/wsdlOptions/wsdlOption/wsdl

WSDL 파일의 위치를 지정합니다.

44.2.4. 설명

wsdl2java 작업은 **WSDL** 문서를 가져와서 서비스를 구현할 완전히 주석이 달린 **Java** 코드를 생성합니다. **WSDL** 문서에는 유효한 **portType** 요소가 있어야 하지만 바인딩 요소 또는 서비스 요소를 포함할 필요는 없습니다. 선택적 인수를 사용하여 생성된 코드를 사용자 지정할 수 있습니다.

44.2.5. WSDL 옵션

플러그인을 구성하려면 하나 이상의 **wsdlOptions** 요소가 필요합니다. **wsdlOptions** 요소의 **wsdl** 자식이 필요하며 플러그인에서 처리할 **WSDL** 문서를 지정합니다. **wsdl** 요소 외에도 **wsdl Options** 요소는 **WSDL** 문서가 처리되는 방법을 사용자 지정할 수 있는 여러 개의 자식을 사용할 수 있습니다.

두 개 이상의 **wsdlOptions** 요소를 플러그인 구성에 나열할 수 있습니다. 각 요소는 처리를 위한 단일 **WSDL** 문서를 구성합니다.

44.2.6. 기본 옵션

defaultOptions 요소는 선택적 요소입니다. 지정된 **WSDL** 문서 모두에 사용되는 옵션을 설정하는 데 사용할 수 있습니다.

중요

옵션이 **wsdlOptions** 요소에서 중복되는 경우 **wsdlOptions** 요소의 값은 앞에 추가됩니다.

44.2.7. 코드 생성 옵션 지정

일반 코드 생성 옵션(**Apache CXF wsdl2java** 명령줄 툴에서 지원하는 스위치에 연결)을 지정하려면 추가**args** 요소를 **wsdlOption** 요소의 자식으로 추가할 수 있습니다. 예를 들어 다음과 같이 **-impl** 옵션과 **-verbose** 옵션을 추가할 수 있습니다.

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <!-- you can set the options of wsdl2java command by using the <extraargs> -->
      <extraargs>
        <extraarg>-impl</extraarg>
        <extraarg>-verbose</extraarg>
      </extraargs>
```

```
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

스위치에서 인수를 사용하는 경우 후속 **extraarg** 요소를 사용하여 이러한 인수를 지정할 수 있습니다. 예를 들어 **jibx** 데이터 바인딩을 지정하려면 다음과 같이 플러그인을 구성할 수 있습니다.

```
...
<configuration>
  <sourceRoot>target/generated/src/main/java</sourceRoot>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <extraargs>
        <extraarg>-databinding</extraarg>
        <extraarg>jibx</extraarg>
      </extraargs>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

44.2.8. 바인딩 파일 지정

하나 이상의 **JAX-WS** 바인딩 파일의 위치를 지정하려면 **bindingFiles** 요소를 **wsdlOption**의 자식으로 추가할 수 있습니다.

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <bindingFiles>
        <bindingFile>${basedir}/src/main/resources/wsdl/async_binding.xml</bindingFile>
      </bindingFiles>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

44.2.9. 특정 WSDL 서비스에 대한 코드 생성

코드를 생성할 **WSDL** 서비스의 이름을 지정하려면 **serviceName** 요소를 **wsdlOption** 의 자식(기본 동작은 **WSDL** 문서의 모든 서비스에 대한 코드를 생성하는 것입니다)으로 추가할 수 있습니다.

```
...
```

```
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
      <serviceName>MyWSDLService</serviceName>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

44.2.10. 여러 WSDL 파일에 대한 코드 생성

여러 **WSDL** 파일에 대한 코드를 생성하려면 **WSDL** 파일에 대한 **wsdlOption** 요소를 추가로 삽입하기만 하면 됩니다. 모든 **WSDL** 파일에 적용되는 몇 가지 일반적인 옵션을 지정하려면 공통 옵션을 **defaultOptions** 요소에 추가합니다.

```
<configuration>
  <defaultOptions>
     <bindingFiles>
        <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
     </bindingFiles>
     <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlOptions>
     <wsdlOption>
        <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
        <serviceName>MyWSDLService</serviceName>
     </wsdlOption>
     <wsdlOption>
        <wsdl>${basedir}/src/main/resources/wsdl/myOtherService.wsdl</wsdl>
        <serviceName>MyOtherWSDLService</serviceName>
     </wsdlOption>
  </wsdlOptions>
</configuration>
```

와일드카드 일치를 사용하여 여러 **WSDL** 파일을 지정할 수도 있습니다. 이 경우 **wsdlRoot** 요소를 사용하여 **WSDL** 파일이 포함된 디렉터리를 지정한 다음 * 문자와 함께 와일드카드를 지원하는 **include** 요소를 사용하여 필요한 **WSDL** 파일을 선택합니다. 예를 들어, **src/main/resources/wsdl** 루트 디렉토리에서 **Service.wsdl** 로 끝나는 모든 **WSDL** 파일을 선택하려면 다음과 같이 플러그인을 구성할 수 있습니다.

```
<configuration>
  <defaultOptions>
     <bindingFiles>
        <bindingFile>${basedir}/src/main/jaxb/bindings.xml</bindingFile>
     </bindingFiles>
     <noAddressBinding>true</noAddressBinding>
  </defaultOptions>
  <wsdlRoot>${basedir}/src/main/resources/wsdl</wsdlRoot>
  <includes>
```

```
      <include>*Service.wsdl</include>
    </includes>
  </configuration>
```

44.2.11. Maven 리포지토리에서 WSDL 다운로드

Maven 리포지토리에서 직접 **WSDL** 파일을 다운로드하려면 **wsdlArtifact** 요소를 **wsdlOption** 요소의 자식으로 추가하고 다음과 같이 **Maven** 아티팩트의 좌표를 지정합니다.

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdlArtifact>
        <groupId>org.apache.pizza</groupId>
        <artifactId>PizzaService</artifactId>
        <version>1.0.0</version>
      </wsdlArtifact>
    </wsdlOption>
  </wsdlOptions>
</configuration>
...
```

44.2.12. 인코딩

(requires JAXB 2.2) 생성된 **Java** 파일에 사용되는 문자 인코딩**(Charset)**을 지정하려면 다음과 같이 **encoding** 요소를 구성 요소의 자식으로 추가합니다.

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <encoding>UTF-8</encoding>
</configuration>
...
```

44.2.13. 별도의 프로세스 복제

fork 요소를 구성 요소의 자식으로 추가하여 코드 생성을 위해 별도의 **JVM**을 분기하도록 **codegen** 플러그인을 구성할 수 있습니다. **fork** 요소는 다음 값 중 하나로 설정할 수 있습니다.

한 번

codegen 플러그인 구성에 지정된 모든 WSDL 파일을 처리하도록 단일 새 JVM을 분기합니다.

always

코드gen 플러그인 구성에 지정된 각 WSDL 파일을 처리하도록 새 JVM을 포크합니다.

false

(기본값) 는 forking을 비활성화합니다.

코드gen 플러그인이 별도의 JVM을 포크하도록 구성된 경우(즉, fork 옵션이 false 이외의 값으로 설정되어 있음) additional JVM 인수를 additionalJvmArgs 요소를 통해 forked JVM에 지정할 수 있습니다. 예를 들어, 다음 조각은 로컬 파일 시스템에서만 XML 스키마에 액세스하도록 (javax.xml.accessExternalSchema 시스템 속성)에서 XML 스키마에 액세스하도록 제한되는 단일 JVM을 분기하도록 codegen 플러그인을 구성합니다.

```
...
<configuration>
  <wsdlOptions>
    <wsdlOption>
      <wsdl>${basedir}/src/main/resources/wsdl/myService.wsdl</wsdl>
    </wsdlOption>
  </wsdlOptions>
  <fork>once</fork>
  <additionalJvmArgs>-Djavax.xml.accessExternalSchema=jar:file,file</additionalJvmArgs>
</configuration>
...
```

44.2.14. 옵션 참조

코드 생성 프로세스를 관리하는 데 사용되는 옵션은 다음 표에서 검토합니다.

옵션	해석
-fe|-frontend *frontend*	코드 생성기에서 사용하는 프런트 엔드를 지정합니다.Specifies the front end used by the code generator. 가능한 값은 **jaxws**,**jaxws21** 및 **cxf** 입니다. **jaxws21** 프론트엔드는 JAX-WS 2.1 규정을 준수하는 데 사용됩니다. **jaxws** frontend 대신 선택적으로 사용할 수 있는 **cxf** frontend는 서비스 클래스에 대한 추가 생성자를 제공합니다. 이 생성자를 사용하면 서비스 구성을 위한 버스 인스턴스를 편리하게 지정할 수 있습니다. 기본값은 **jaxws** 입니다.

옵션	해석
-db|-databinding *databinding*	코드 생성기에서 사용하는 데이터 바인딩을 지정합니다.Specifies the data binding used by the code generator. 가능한 값은 **jaxb**,**xmlbeans**,**sdo -static** 및 **sdo-dynamic**), **jibx** 입니다. 기본값은 **jaxb** 입니다.
-wv *wsdlVersion*	도구에서 예상되는 WSDL 버전을 지정합니다. 기본값은 **1.1** 입니다.[a]
-p *wsdlNamespace=PackageName*	생성된 코드에 사용할 0개 이상의 패키지 이름을 지정합니다. 선택적으로 패키지 이름 매핑에 대한 WSDL 네임스페이스를 지정합니다.
-b *bindingName*	하나 이상의 JAXWS 또는 JAXB 바인딩 파일을 지정합니다. 각 바인딩 파일에 별도의 **-b** 플래그를 사용합니다.
-sn *serviceName*	코드를 생성할 WSDL 서비스의 이름을 지정합니다. 기본값은 WSDL 문서의 모든 서비스에 대한 코드를 생성하는 것입니다.
-reserveClass *classname*	**-autoNameResolution** 와 함께 사용되는 경우 클래스를 생성할 때 사용하지 **않는** wsdl-to-java의 클래스 이름을 정의합니다. 여러 클래스에 대해 이 옵션을 여러 번 사용하십시오.
-catalog *catalogUrl*	가져온 스키마 및 WSDL 문서를 확인하는 데 사용할 XML 카탈로그의 URL을 지정합니다.
-d *output-directory*	생성된 코드 파일이 작성되는 디렉터리를 지정합니다.
-compile	생성된 Java 파일을 컴파일합니다.
-classdir *complile-class-dir*	컴파일된 클래스 파일이 작성되는 디렉터리를 지정합니다.Specifies the directory into which the compiled class files are written.
-clientjar *jar-file-name*	모든 클라이언트 클래스 및 WSDL이 포함된 JAR 파일을 생성합니다. 이 옵션을 지정하면 지정된 **wsdlLocation** 이 작동하지 않습니다.
-client	클라이언트 메인 라인에 대한 시작점 코드를 생성합니다.
-server	서버 메인 라인에 대한 시작 지점 코드를 생성합니다.

옵션	해석
-impl	구현 개체의 시작 지점 코드를 생성합니다.Generates starting point code for an implementation object.
-all	모든 시작 지점 코드를 생성합니다. 유형, 서비스 프록시, 서비스 인터페이스, 서버 메인라인, 클라이언트 메인라인, 구현 오브젝트, Ant **build.xml** 파일.
-ant	Ant **build.xml** 파일을 생성합니다.
-autoNameResolution	바인딩 사용자 정의를 사용하지 않고도 이름 지정을 자동으로 해결합니다.
-defaultValues=*DefaultValueProvider*	생성된 클라이언트 및 생성된 구현에 대한 기본값을 생성하도록 툴에 지시합니다. 선택적으로 기본값을 생성하는 데 사용되는 클래스의 이름을 제공할 수도 있습니다. 기본적으로 **RandomValueProvider** 클래스가 사용됩니다.
-nexclude *schema-namespace=java-packagename*	코드를 생성할 때 지정된 WSDL 스키마 네임스페이스를 무시합니다. 이 옵션은 여러 번 지정할 수 있습니다. 또한 선택적으로 제외된 네임스페이스에 설명된 유형에서 사용하는 Java 패키지 이름을 지정합니다.
-exsh (**true**/**false**)	확장된 soap 헤더 메시지 바인딩 처리를 활성화하거나 비활성화합니다. 기본값은 false입니다.
-noTypes	생성 유형을 비활성화합니다.
-DNS (true/false)	기본 네임스페이스 패키지 이름 매핑을 활성화하거나 비활성화합니다. 기본값은 true입니다.
-DEX (true/false)	default excludes 네임스페이스 매핑을 활성화하거나 비활성화합니다. 기본값은 true입니다.
-xjc*args*	JAXB 데이터 바인딩을 사용하는 경우 XJC로 직접 전달할 쉼표로 구분된 인수 목록을 지정합니다. 가능한 모든 XJC 인수 목록을 가져오려면 **-xjc-X** 를 사용합니다.
-noAddressBinding	JAX-WS 2.1 호환 매핑 대신 Apache CXF 전용 WS-Addressing 유형을 사용하도록 툴에 지시합니다.
-validate [=all|basic|none]	코드를 생성하려고 시도하기 전에 WSDL 문서의 유효성을 검사하도록 도구를 지시합니다.
-keep	기존 파일을 덮어쓰지 않도록 도구에 지시합니다.
-wsdlLocation *wsdlLocation*	**@WebService** 주석의 **wsdlLocation** 속성 값을 지정합니다.

옵션	해석
-v	도구의 버전 번호를 표시합니다.
-verbose|-V	코드 생성 프로세스 중 주석을 표시합니다.
-quiet	코드 생성 프로세스 중 주석을 비활성화합니다.
-allowElementReferences[=true], **-aer[=true]**	**true** 인 경우 JAX-WS 2.2 사양의 2.3.1.2(v) 섹션에 제공된 규칙을 무시하면 래퍼 스타일 매핑을 사용할 때 요소 참조를 허용하지 않습니다. 기본값은 **false** 입니다.
-asyncMethods[=*method1*,*method2*,...]	JAX-WS 바인딩 파일에서 **enableAsyncMapping** 과 유사하게 클라이언트 측 비동기 호출을 허용하도록 생성된 Java 클래스 메서드 목록입니다.
-bareMethods[=*method1*,*method2*,...]	JAX-WS 바인딩 파일에서 **enableWrapperStyle** 과 유사하게 래퍼 스타일을 갖는 Java 클래스 메서드 목록입니다(아래 참조).
-mimeMethods[=*method1*,*method2*,...]	JAX-WS 바인딩 파일에서 **MIMEContent를 활성화하는 것과 유사하게 mime:content 매핑을 활성화하는** Java 클래스 메서드 목록입니다.
-faultSerialVersionUID *fault-serialVersionUID*	오류 예외의 suid를 생성하는 방법 가능한 값은 **NONE**,**TIMESTAMP**,**FQCN** 또는 특정 번호입니다. 기본값은 **None입니다**.
-encoding *인코딩*	Java 코드를 생성할 때 사용할 Charset 인코딩을 지정합니다.
-exceptionSuper	**wsdl:fault** 요소에서 생성된 오류 빈의 슈퍼 클래스(기본값: **java.lang.Exception**).
-seiSuper *interfaceName*	생성된 SEI 인터페이스의 기본 인터페이스를 지정합니다. 예를 들어 이 옵션을 사용하여 Java 7 **AutoCloseable** 인터페이스를 슈퍼 인터페이스로 추가할 수 있습니다.
-mark-generated	생성된 클래스에 @Generated 주석을 추가합니다.

[a] 현재 Apache CXF는 코드 생성기에 대한 WSDL 1.1 지원만 제공합니다.

44.3. JAVA2WS

초록

Java 코드에서 **WSDL** 문서를 생성

44.3.1. 개요

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-java2ws-plugin</artifactId>
  <version>version</version>
  <executions>
    <execution>
      <id>process-classes</id>
      <phase>process-classes</phase>
      <configuration>
        <className>className</className>
        <option>...</option>
        ...
      </configuration>
      <goals>
        <goal>java2ws</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

44.3.2. 설명

java2ws 작업은 서비스 엔드포인트 구현(**SEI**)을 사용하고 웹 서비스를 구현하는 데 사용되는 지원 파일을 생성합니다. 다음을 생성할 수 있습니다.

- **WSDL** 문서
- 서비스를 **POJO**로 배포하는 데 필요한 서버 코드
- 서비스에 액세스하기 위한 클라이언트 코드
- 래퍼 및 오류 빈

44.3.3. 필수 구성

*플러그인을 사용하려면 **className** 구성 요소가 있어야 합니다. 요소의 값은 처리할 **SEI**의 정규화된 이름입니다.*

44.3.4. 선택적 구성

*다음 표에 나열된 구성 요소의 를 사용하여 **WSDL** 생성을 미세 조정할 수 있습니다.*

요소	설명
frontend	SEI를 처리하고 지원 클래스를 생성하는 데 사용할 프론트엔드를 지정합니다. **jaxws** 가 기본값입니다. **simple** 도 지원됩니다.
DataBinding	SEI를 처리하고 지원 클래스를 생성하는 데 사용되는 데이터 바인딩을 지정합니다.Specifies the data binding used for processing the SEI and generating the support classes. JAX-WS 프런트 엔드를 사용할 때 기본값은 **jaxb** 입니다. 단순 프런트 엔드를 사용할 때 기본값은 **aegis** 입니다.
genWsdl	**true** 로 설정할 때 WSDL 문서를 생성하도록 도구를 지시합니다.
genWrapperbean	**true** 로 설정할 때 래퍼 빈과 오류 빈을 생성하도록 툴에 지시합니다.
genClient	**true** 로 설정할 때 클라이언트 코드를 생성하도록 도구를 지시합니다.
genServer	**true** 로 설정할 때 서버 코드를 생성하도록 도구를 지시합니다.
outputFile	생성된 WSDL 파일의 이름을 지정합니다.
classpath	SEI를 처리할 때 검색되는 classpath를 지정합니다.
soap12	**true** 로 설정할 때 생성된 WSDL 문서가 SOAP 1.2 바인딩을 포함하도록 지정합니다.
targetNamespace	생성된 WSDL 파일에 사용할 대상 네임스페이스를 지정합니다.
serviceName	생성된 **서비스** 요소의 **name** 특성 값을 지정합니다.

VI 부. RESTFUL 웹 서비스 개발

*이 가이드에서는 **JAX-RS API**를 사용하여 웹 서비스를 구현하는 방법을 설명합니다.*

45장. RESTFUL 웹 서비스 소개

초록

REST(Representational State Transfer)는 네 가지 기본 HTTP 동사만 사용하여 HTTP를 통한 데이터 전송을 중심으로 하는 소프트웨어 아키텍처 스타일입니다. 또한 SOAP 봉투 및 상태 데이터 사용과 같은 추가 래퍼를 사용하고 있습니다.

45.1. 개요

REST(Representational State Transfer)는 Roy Fielding이라는 연구자에 의해 의사 논문에서 처음으로 기술한 아키텍처 스타일입니다. RESTful 시스템에서 서버는 URI를 사용하여 리소스를 노출하고 클라이언트는 4개의 HTTP 동사를 사용하여 이러한 리소스에 액세스합니다. 클라이언트가 리소스 표시를 수신하면 상태가 됩니다. 일반적으로 링크를 따라 새 리소스에 액세스할 때 해당 상태를 변경하거나 전환합니다. REST는 작업을 수행하기 위해 광범위한 표준 grammar를 사용하여 리소스를 나타낼 수 있다고 가정합니다.

World Wide Web은 REST 원칙에 따라 설계된 시스템의 가장 유비쿼터스 예제입니다. 웹 브라우저는 웹 서버에서 호스팅되는 리소스에 액세스하는 클라이언트 역할을 합니다. 리소스는 모든 웹 브라우저에서 사용할 수 있는 HTML 또는 XML 문법을 사용하여 나타냅니다. 브라우저는 또한 쉽게 새로운 자원에 대한 링크를 따를 수 있습니다.

RESTful 시스템의 장점은 확장성이 높고 유연하다는 것입니다. 네 가지 HTTP 동사를 사용하여 리소스에 액세스하고 조작되므로 리소스가 URI를 사용하여 노출되고 리소스는 표준 grammars를 사용하여 표시되므로 클라이언트는 서버 변경에 의한 영향을 받지 않습니다. 또한 RESTful 시스템은 캐싱 및 프록시와 같은 HTTP의 확장성 기능을 최대한 활용할 수 있습니다.

45.2. 기본 REST 원칙

RESTful 아키텍처는 다음과 같은 기본 원칙을 준수합니다.

- 애플리케이션 상태 및 기능은 리소스로 나뉩니다.
- 리소스는 하이퍼 미디어 링크로 사용할 수 있는 표준 URI를 사용하여 해결할 수 있습니다.
- 모든 리소스는 4개의 HTTP 동사만 사용합니다.

- *DELETE*
- *GET*
- *POST*
- *PUT*

- *모든 리소스는 HTTP에서 지원하는 MIME 유형을 사용하여 정보를 제공합니다.*
- *프로토콜은 상태 비저장 방식입니다.*
- *응답은 캐시할 수 있습니다.*
- *프로토콜이 계층화되어 있습니다.*

45.3. 리소스

리소스 는 REST의 핵심입니다. 리소스는 URI를 사용하여 처리할 수 있는 정보의 소스입니다. 웹 초기에는 리소스가 주로 정적 문서였습니다. 최신 웹에서 리소스는 모든 정보 소스가 될 수 있습니다. 예를 들어 웹 서비스는 URI를 사용하여 액세스할 수 있는 경우 리소스일 수 있습니다.

RESTful 엔드포인트는 주소를 처리하는 리소스를 나타냅니다. 표현은 리소스에서 제공하는 데이터를 포함하는 문서입니다. 예를 들어 고객 레코드에 대한 액세스를 제공하는 웹 서비스의 방법은 리소스이며 서비스와 소비자 간에 교환된 고객 레코드의 사본은 리소스를 나타냅니다.

45.4. REST 모범 사례

RESTful 웹 서비스를 설계할 때 다음 사항을 염두에 두는 것이 좋습니다.

- *노출하려는 각 리소스에 대해 별도의 URI를 제공합니다.*

*예를 들어 구동 레코드를 다루는 시스템을 구축하는 경우 각 레코드에는 고유한 **URI**가 있어야 합니다. 또한 시스템이 주차 위반 및 벌금에 대한 정보를 제공하는 경우, 각 리소스 유형에는 고유한 기반도 있어야 합니다. 예를 들어, 운이 빠른 벌금은 **/speedingfines/driverID** 및 주차 위반은 **/parkingfines/driverID** 를 통해 액세스 할 수 있습니다.*

- *__URI__에 __nouns__를 사용합니다.*

  *__nouns__를 사용하면 리소스가 사물이며 작동하지 않음을 강조합니다. **/ordering** 과 같은 __URI__는 작업을 의미하지만 / 주문은 항목을 의미합니다.*

- *__GET__ 에 매핑되는 메서드는 데이터를 변경하지 않아야 합니다.*

- *귀하의 답변에 대한 링크를 사용하십시오.*

  응답에 다른 리소스에 대한 링크를 배치하면 클라이언트가 데이터 체인을 더 쉽게 추적할 수 있습니다. 예를 들어 서비스에서 리소스 컬렉션을 반환하는 경우 클라이언트가 제공된 링크를 사용하여 각 개별 리소스에 더 쉽게 액세스할 수 있습니다. 링크가 포함되지 않은 경우 클라이언트에는 특정 노드에 체인을 따르려면 추가 논리가 필요합니다.

- *서비스를 상태 비저장으로 설정합니다.*

  클라이언트 또는 서비스가 상태 정보를 유지하도록 요구하면 둘 사이에 긴밀한 결합이 필요합니다. 긴밀한 결합으로 업그레이드 및 마이그레이션은 더욱 어려워집니다. 상태를 유지 관리하면 통신 오류로부터 복구도 더 어려워질 수 있습니다.

45.5. RESTFUL 웹 서비스 설계

***RESTful** 웹 서비스를 구현하는 데 사용하는 프레임워크와 관계없이 여러 단계를 따라야 합니다.*

1. *서비스에서 노출할 리소스를 정의합니다.*

   일반적으로 서비스는 트리로 구성된 하나 이상의 리소스를 노출합니다. 예를 들어 운전 기록 서비스는 다음 세 가지 리소스로 구성할 수 있습니다.

   - ***/license/driverID***

- */license/driverID/speedingfines*
- */license/driverID/parkingfines*

2. *각 리소스에서 수행할 작업을 정의합니다.*

   예를 들어, 다이버의 주소를 업데이트하거나 운전자 기록에서 주차 티켓을 제거할 수 있습니다.

3. *작업을 적절한 **HTTP** 동사에 매핑합니다.*

*서비스를 정의한 후에는 **Apache CXF**를 사용하여 구현할 수 있습니다.*

45.6. APACHE CXF로 REST 구현

***Apache CXF**는 **RESTful Web Services(JAX-RS)**에 대한 **Java API**구현을 제공합니다. **JAX-RS**는 주석을 사용하여 **POJO**를 리소스에 매핑하는 표준화된 방법을 제공합니다.*

*추상 서비스 정의에서 **JAX-RS**를 사용하여 구현된 **RESTful** 웹 서비스로 이동할 때 다음을 수행해야 합니다.*

1. *서비스의 리소스 트리의 최상위를 나타내는 리소스에 대한 루트 리소스 클래스를 생성합니다.*

   ***46.3**절. "루트 리소스 클래스"*을 참조하십시오.

2. *서비스의 기타 리소스를 하위 리소스에 매핑합니다.*

   ***46.5**절. "하위 리소스 작업"*을 참조하십시오.

3. *각 리소스에서 사용하는 각 **HTTP** 동사를 구현하는 메서드를 생성합니다.*

   ***46.4**절. "리소스 메서드 작업"*을 참조하십시오.

참고

Apache CXF는 Java 인터페이스를 RESTful 웹 서비스에 매핑하는 이전 HTTP 바인딩을 계속 지원합니다. HTTP 바인딩은 기본 기능을 제공하며 여러 가지 제한 사항이 있습니다. 개발자는 JAX-RS를 사용하도록 애플리케이션을 업데이트하는 것이 좋습니다.

45.7. 데이터 바인딩

기본적으로 Apache CXF는 Java Architecture for XML Binding (JAXB) 개체를 사용하여 리소스와 해당 표현을 Java 오브젝트에 매핑합니다. Java 개체와 XML 요소 간의 깔끔하고 잘 정의된 매핑을 제공합니다.

Apache CXF JAX-RS 구현에서는 JSON(JavaScript Object Notation)을 사용하여 데이터 교환도 지원합니다. JSON은 Ajax 개발자가 사용하는 인기 있는 데이터 형식입니다. JSON과 JAXB 간 데이터 마샬링은 Apache CXF 런타임에서 처리합니다.

46장. 리소스 생성

초록

RESTful 웹 서비스에서 모든 요청은 리소스에서 처리합니다. **JAX-RS API**는 **Java** 클래스로 리소스를 구현합니다. 리소스 클래스는 하나 이상의 **JAX-RS** 주석이 추가된 **Java** 클래스입니다. **JAX-RS**를 사용하여 구현된 **RESTful** 웹 서비스의 코어는 루트 리소스 클래스입니다. **root** 리소스 클래스는 서비스에서 노출하는 리소스 트리의 진입점입니다. 모든 요청을 처리할 수도 있고 요청을 처리하는 하위 리소스에 대한 액세스를 제공할 수 있습니다.

46.1. 소개

46.1.1. 개요

JAX-RS API를 사용하여 구현된 **RESTful** 웹 서비스는 **Java** 클래스에서 구현하는 리소스의 표현으로 응답을 제공합니다. 리소스 클래스 는 **JAX-RS** 주석을 사용하여 리소스를 구현하는 클래스입니다. 대부분의 **RESTful** 웹 서비스의 경우 액세스해야 하는 리소스 컬렉션이 있습니다. 리소스 클래스의 주석은 리소스의 **URI**와 같은 정보와 각 작업에서 처리하는 **HTTP** 동사 정보를 제공합니다.

46.1.2. 리소스 유형

JAX-RS API를 사용하면 두 가지 기본 유형의 리소스를 만들 수 있습니다.

- 46.3절. “루트 리소스 클래스” 은 서비스의 리소스 트리를 가리키는 진입점입니다. 서비스에서 리소스의 기본 **URI**를 정의하기 위해 **@Path** 주석으로 장식됩니다.

- 46.5절. “하위 리소스 작업” **root** 리소스를 통해 액세스합니다. 이는 **@Path** 주석으로 장식되는 메서드에 의해 구현됩니다. 하위 리소스의 **@Path** 주석은 루트 리소스의 기본 **URI**를 기준으로 **URI**를 정의합니다.

46.1.3. 예제

예 46.1. “간단한 리소스 클래스” 간단한 리소스 클래스를 보여줍니다.

예 **46.1.** 간단한 리소스 클래스

```
package demo.jaxrs.server;

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

```
import javax.ws.rs.PathParam;

@Path("/customerservice")
public class CustomerService
{
  public CustomerService()
  {
  }

  @GET
  public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  ...
}
```

두 가지 항목은 예 46.1. “간단한 리소스 클래스” 에 정의된 클래스를 리소스 클래스로 설정합니다.

***@Path** 주석은 리소스의 기본 **URI**를 지정합니다.*

***@GET** 주석은 메서드가 리소스에 대한 **HTTP GET** 메서드를 구현하도록 지정합니다.*

46.2. 기본 JAX-RS 주석

46.2.1. 개요

***RESTful** 웹 서비스 구현에 필요한 가장 기본적인 정보는 다음과 같습니다.*

- *서비스 리소스의 **URI***
- *클래스의 메서드가 **HTTP** 동사에 매핑되는 방법*

***JAX-RS**는 이러한 기본 정보를 제공하는 주석 세트를 정의합니다. 모든 리소스 클래스에는 이러한 주석 중 하나 이상이 있어야 합니다.*

46.2.2. 경로 설정

@Path 주석은 리소스의 URI를 지정합니다. 주석은 javax.ws.rs.Path 인터페이스에서 정의되며 리소스 클래스 또는 리소스 메서드를 장식하는 데 사용할 수 있습니다. 문자열 값을 유일한 매개 변수로 사용합니다. string 값은 구현된 리소스의 위치를 지정하는 URI 템플릿입니다.

URI 템플릿은 리소스의 상대 위치를 지정합니다. 예 46.2. "URI 템플릿 구문" 에 표시된 대로 템플릿에는 다음이 포함될 수 있습니다.

- *처리되지 않은 경로 구성 요소*
- *{ }로 둘러싸인 매개 변수 식별자*

참고

매개 변수 식별자는 기본 경로 처리를 변경하는 정규식을 포함할 수 있습니다.

예 46.2. URI 템플릿 구문

```
@Path("resourceName/{param1}/../{paramN}")
```

예를 들어 URI 템플릿 위젯 /{color}/{number} 는 위젯s /blue/12 에 매핑됩니다. color 매개변수 값은 파란색에 할당됩니다. number 매개 변수의 값은 12 로 할당됩니다.

URI 템플릿이 전체 URI에 매핑되는 방법은 @Path 주석 장식에 따라 다릅니다. 루트 리소스 클래스에 배치되면 URI 템플릿은 트리의 모든 리소스의 루트 URI이며 서비스가 게시되는 URI에 직접 추가됩니다. 주석이 하위 리소스를 장식하는 경우 루트 리소스 URI를 기준으로 합니다.

46.2.3. HTTP 동사 지정

JAX-RS는 메서드에 사용할 HTTP 동사를 지정하기 위해 다섯 개의 주석을 사용합니다.

- *javax.ws.rs.DELETE는 메서드가 DELETE 에 매핑되도록 지정합니다.*

- *javax.ws.rs.GET 메서드가 GET 에 매핑되도록 지정합니다.*
- *javax.ws.rs.POST는 메서드가 POST 에 매핑되도록 지정합니다.*
- *javax.ws.rs.PUT는 메서드가 PUT 에 매핑되도록 지정합니다.*
- *javax.ws.rs.HEAD는 메서드가 HEAD 에 매핑되도록 지정합니다.*

메서드를 HTTP 동사에 매핑할 때 매핑이 의미가 있는지 확인해야 합니다. 예를 들어 구매 주문을 제출하려는 메서드를 매핑하는 경우 PUT 또는 POST 에 매핑합니다. GET 또는 DELETE 에 매핑하면 예기치 않은 동작이 발생합니다.

46.3. 루트 리소스 클래스

46.3.1. 개요

루트 리소스 클래스는 RESTful 웹 서비스를 구현하는 JAX-RS의 진입점입니다. 서비스에서 구현하는 리소스의 루트 URI를 지정하는 @Path 로 장식됩니다. 해당 메서드는 리소스에 대한 작업을 직접 구현하거나 하위 리소스에 대한 액세스를 제공합니다.

46.3.2. 요구 사항

클래스를 루트 리소스 클래스가 되기 위해서는 다음 기준을 충족해야 합니다.

- *클래스는 @Path 주석으로 장식되어야 합니다.*

  지정된 경로는 서비스에서 구현하는 모든 리소스의 루트 URI입니다. 루트 리소스 클래스가 해당 경로가 위젯임을 지정하고 해당 메서드 중 하나가 GET 동사를 구현하도록 지정하는 경우 위젯에서 GET 이 해당 메서드를 호출합니다. 하위 리소스에서 해당 URI가 {id} 로 지정되면 하위 리소스의 전체 URI 템플릿이 위젯 /{id}이며 위젯/ 12 및 위젯 / 42 와 같은 URI에 대한 요청을 처리합니다.

- *클래스에는 런타임이 호출할 공용 생성자가 있어야 합니다.*

  런타임은 모든 생성자의 매개 변수에 대한 값을 제공할 수 있어야 합니다. 생성자의 매개 변수에는 JAX-RS 매개 변수 주석으로 장식되는 매개 변수가 포함될 수 있습니다. 매개변수 주석에

대한 자세한 내용은 47장. 리소스 클래스 및 메서드에 정보 전달 에서 참조하십시오.

- 클래스 메서드 중 하나 이상이 **HTTP** 동사 주석 또는 **@Path** 주석으로 장식되어야 합니다.

46.3.3. 예제

예 46.3. "루트 리소스 클래스" 하위 리소스에 대한 액세스를 제공하는 루트 리소스 클래스를 표시합니다.

예 46.3. 루트 리소스 클래스

```
package demo.jaxrs.server;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/customerservice/")
public class CustomerService
{
  public CustomerService()
  {
    ...
  }

  @GET
  public Customer getCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @DELETE
  public Response deleteCustomer(@QueryParam("id") String id)
  {
    ...
  }

  @PUT
  public Response updateCustomer(Customer customer)
  {
    ...
  }

  @POST
  public Response addCustomer(Customer customer)
```

```
    {
      ...
    }

    @Path("/orders/{orderId}/")
    public Order getOrder(@PathParam("orderId") String orderId)
    {
      ...
    }

  }
```

예 46.3. "루트 리소스 클래스" 의 클래스는 루트 리소스 클래스에 대한 모든 요구 사항을 충족합니다.

클래스는 **@Path** 주석으로 장식됩니다. 서비스에서 노출하는 리소스의 루트 **URI**는 **customerservice** 입니다.

클래스에는 **public** 생성자가 있습니다. 이 경우 간단히 인수 생성자를 사용하지 않습니다.**In this case the no argument constructor is used for simplicity.**

이 클래스는 리소스에 대한 **4**개의 **HTTP** 동사를 각각 구현합니다.

이 클래스는 **getOrder()** 메서드를 통해 하위 리소스에 대한 액세스도 제공합니다. **@Path** 주석을 사용하여 지정된 대로 하위 리소스의 **URI**는 **customerservice/order/id** 입니다. 하위 리소스는 **Order** 클래스에서 구현됩니다.

하위 리소스 구현에 대한 자세한 내용은 **46.5**절. "하위 리소스 작업" 을 참조하십시오.

46.4. 리소스 메서드 작업

46.4.1. 개요

리소스 메서드에는 **JAX-RS** 주석을 사용하여 주석이 추가됩니다. 메서드 프로세스에 대한 요청 유형을 지정하는 **HTTP** 메서드 주석 중 하나가 있습니다. **JAX-RS**는 리소스 메서드에 여러 가지 제약 조건을 배치합니다.

46.4.2. 일반 제약 조건

모든 리소스 메서드는 다음 조건을 충족해야 합니다.**All resource methods must meet the following conditions:**

- 공개가 되어야 합니다.
- "HTTP 동사 지정" 에 설명된 **HTTP** 메서드 주석 중 하나로 장식해야 합니다.
- "매개 변수" 에 설명된 대로 엔터티 매개 변수가 두 개 이상 없어야 합니다.

46.4.3. 매개 변수

리소스 메서드 매개변수는 다음 두 가지 형식을 사용합니다.

- 엔터티 매개변수-**Entity** 매개변수는 주석이 없습니다. 해당 값은 요청 엔터티 본문에서 매핑됩니다. 엔터티 매개 변수는 애플리케이션에 엔터티 공급자가 있는 모든 유형일 수 있습니다. 일반적으로 이러한 개체는 **JAXB** 오브젝트입니다.

중요

리소스 메서드에는 하나의 엔터티 매개 변수만 있을 수 있습니다.

엔터티 공급자에 대한 자세한 내용은 **51**장. 엔터티 지원 을 참조하십시오.

- 주석이 달린 매개변수 -**Annotated** 매개변수는 요청에서 매개변수 값을 매핑하는 방법을 지정하는 **JAX-RS** 주석 중 하나를 사용합니다. 일반적으로 매개 변수의 값은 요청 **URI**의 일부에서 매핑됩니다.

요청 데이터를 메서드 매개 변수에 매핑하는 데 **JAX-RS** 주석을 사용하는 방법에 대한 자세한 내용은 **47**장. 리소스 클래스 및 메서드에 정보 전달 을 참조하십시오.

예 **46.4.** "유효한 매개변수 목록이 있는 리소스 메서드" 유효한 매개 변수 목록이 있는 리소스 메서드를 보여줍니다.

> 예 **46.4.** 유효한 매개변수 목록이 있는 리소스 메서드

```
@POST
@Path("disaster/monster/giant/{id}")
public void addDaikaiju(Kaiju kaiju,
                        @PathParam("id") String id)
{
  ...
}
```

예 46.5. "매개변수가 유효하지 않은 리소스 메서드" 매개 변수 목록이 유효하지 않은 리소스 메서드를 표시합니다. 주석이 없는 두 개의 매개 변수가 있습니다.

예 46.5. 매개변수가 유효하지 않은 리소스 메서드

```
@POST
@Path("disaster/monster/giant/")
public void addDaikaiju(Kaiju kaiju,
                        String id)
{
  ...
}
```

46.4.4. 반환 값

리소스 메서드는 다음 중 하나를 반환할 수 있습니다.

- ***void***
- *애플리케이션에 엔터티 공급자가 있는 **Java** 클래스*

  *엔터티 공급자에 대한 자세한 내용은 **51**장. 엔터티 지원 을 참조하십시오.*
- ***Response** 오브젝트*

  ***Response** 오브젝트에 대한 자세한 내용은 **48.3**절. "애플리케이션 응답 미세 조정" 에서 참조하십시오.*
- ***GenericEntity<T>** 객체*

GenericEntity<T> 오브젝트에 대한 자세 한 내용은 48.4절. “일반 유형 정보가 있는 엔터티 반환” 을 참조하십시오.

모든 리소스 메서드는 HTTP 상태 코드를 요청자에게 반환합니다. 메서드 반환 유형이 void 이거나 반환된 값이 null 이면 리소스 메서드는 HTTP 상태 코드를 204 로 설정합니다. 리소스 메서드가 null 이외의 값을 반환하는 경우 HTTP 상태 코드를 200 으로 설정합니다.

46.5. 하위 리소스 작업

46.5.1. 개요

둘 이상의 리소스에서 서비스를 처리해야 하는 경우가 많습니다. 예를 들어 주문 처리 서비스에서 모범 사례에서는 각 고객이 고유한 리소스로 처리된다고 제안합니다. 각 주문은 고유한 리소스로도 처리됩니다.

JAX-RS API를 사용하여 고객 리소스 및 주문 리소스를 하위 리소스로 구현합니다. 하위 리소스는 root 리소스 클래스를 통해 액세스하는 리소스입니다. 리소스 클래스의 메서드에 @Path 주석을 추가하여 정의합니다. 하위 리소스는 다음 두 가지 방법 중 하나로 구현할 수 있습니다.

- 하위 리소스 메서드- 하위 리소스에 대한 HTTP 동사를 직접 구현하고 “HTTP 동사 지정” 에 설명된 주석 중 하나로 장식됩니다.
- 하위 리소스 locator- 하위 리소스를 구현하는 클래스를 가리킵니다.

46.5.2. 하위 리소스 지정

하위 리소스는 @Path 주석으로 메서드를 장식하여 지정됩니다. 하위 리소스의 URI는 다음과 같이 구성됩니다.

1. 하위 리소스의 @Path 주석 값을 하위 리소스의 상위 리소스의 @Path 주석 값에 추가합니다.

   상위 리소스의 @Path 주석은 하위 리소스가 포함된 클래스의 오브젝트를 반환하는 리소스 클래스의 메서드에 있을 수 있습니다.

2. 루트 리소스에 도달할 때까지 이전 단계를 반복합니다.

3. 어셈블된 **URI**는 서비스가 배포되는 기본 **URI**에 추가됩니다.

예를 들어 예 46.6. "주문 하위 리소스" 에 표시된 하위 리소스의 **URI**는 **baseURI/customerservice/order/12** 일 수 있습니다.

예 **46.6.** 주문 하위 리소스

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }
}
```

46.5.3. 하위 리소스 방법

하위 리소스 방법은 **@Path** 주석과 **HTTP** 동사 주석 중 하나로 장식됩니다. 하위 리소스 메서드는 지정된 **HTTP** 동사를 사용하여 리소스에 대한 요청을 직접 처리합니다.

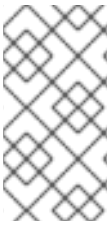
예 46.7. "하위 리소스 방법" 세 가지 하위 리소스 메서드가 있는 리소스 클래스를 표시합니다.

- **getOrder()** 는 **URI**가 **/customerservice/orders/{orderId}/** 와 일치하는 리소스에 대한 **HTTP GET** 요청을 처리합니다.
- **updateOrder()** 는 **URI**가 **/customerservice/orders/{orderId}/** 와 일치하는 리소스에 대한 **HTTP PUT** 요청을 처리합니다.
- **newOrder()** 는 **/customerservice/orders/** 에서 리소스에 대한 **HTTP POST** 요청을 처리합니다.

예 **46.7.** 하위 리소스 방법

```
...
@Path("/customerservice/")
```

```
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @Path("/orders/{orderId}/")
  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

  @Path("/orders/")
  @POST
  public Order newOrder(Order order)
  {
    ...
  }
}
```



참고

동일한 **URI** 템플릿을 사용하는 하위 리소스 메서드는 하위 리소스 로케이터에서 반환된 리소스 클래스와 동일합니다.

46.5.4. 하위 리소스 검색기

하위 리소스 로케이터는 **HTTP** 동사 주석 중 하나로 장식되지 않으며 직접 처리하지 않도록 하위 리소스에 대한 요청입니다. 대신 하위 리소스 로케이터는 요청을 처리할 수 있는 리소스 클래스의 인스턴스를 반환합니다.

HTTP 동사 주석이 없는 경우 하위 리소스 로케이터도 엔터티 매개 변수를 가질 수 없습니다. 하위 리소스 로케이터 메서드에서 사용하는 모든 매개변수는 **47장. 리소스 클래스 및 메서드에 정보 전달** 에 설명된 주석 중 하나를 사용해야 합니다.

예 46.8. "하위 리소스 검색기에서 특정 클래스를 반환" 에 표시된 대로 하위 리소스 로케이터를 사용하면 모든 메서드를 하나의 슈퍼 클래스에 두는 대신 리소스를 재사용 가능한 클래스로 캡슐화할 수 있습니다. **processOrder()** 메서드는 하위 리소스 로케이터입니다. **URI** 템플릿 **/orders/{orderId}/** 에 일치하는 **URI**에서 요청이 생성되면 **Order** 클래스의 인스턴스를 반환합니다. **Order** 클래스에는 **HTTP** 동사 주석으로 장식되는 메서드가 있습니다. **PUT** 요청은 **updateOrder()** 메서드에서 처리합니다.

예 46.8. 하위 리소스 검색기에서 특정 클래스를 반환

```
...
@Path("/customerservice/")
public class CustomerService
{
  ...
  @Path("/orders/{orderId}/")
  public Order processOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  ...
}

public class Order
{
  ...
  @GET
  public Order getOrder(@PathParam("orderId") String orderId)
  {
    ...
  }

  @PUT
  public Order updateOrder(@PathParam("orderId") String orderId,
                           Order order)
  {
    ...
  }

}
```

하위 리소스 로케이터는 런타임에 처리되어 다형성(**polymorphism**)을 지원할 수 있습니다. 하위 리소스 로케이터의 반환 값은 일반 오브젝트, 추상 클래스 또는 클래스 계층 구조의 최상위가 될 수 있습니다. 예를 들어, **PayPal** 주문 및 신용 카드 주문을 모두 처리하는 데 필요한 서비스가 있는 경우 예 46.8. "하위 리소스 검색기에서 특정 클래스를 반환" 의 **processOrder()** 방법은 변경되지 않을 수 있습니다. **ppOrder** 및 **ccOder** 라는 두 개의 클래스를 구현해야 합니다. 이 클래스는 **Order** 클래스를 확장했습니다. **processOrder()** 구현에서는 필요한 논리에 따라 필요한 하위 리소스의 구현을 인스턴스화합니다.

46.6. 리소스 선택 방법

46.6.1. 개요

지정된 **URI**를 하나 이상의 리소스 메서드에 매핑할 수 있습니다. 예를 들어 **URI** 고객 서비스/**12/ma** 는 템플릿 **@Path("customerservice/{id}")** 또는 **@Path("customerservice/{id}/{state}")** 와 일치할 수 있습니다. **JAX-RS**는 리소스 메서드를 요청과 일치하는 세부 알고리즘을 지정합니다. 알고리즘은 정규화된 **URI, HTTP** 동사 및 요청 및 응답 엔터티의 미디어 유형을 리소스 클래스의 주석과 비교합니다.

46.6.2. 기본 선택 알고리즘

JAX-RS 선택 알고리즘은 다음 세 단계로 나뉩니다.

1. 루트 리소스 클래스를 결정합니다.

   요청 URI는 @Path 주석으로 장식되는 모든 클래스와 일치합니다. @Path 주석이 요청 URI와 일치하는 클래스가 결정됩니다.

   리소스 클래스의 @Path 주석 값이 전체 요청 URI와 일치하는 경우 클래스의 메서드가 세 번째 단계에 대한 입력으로 사용됩니다.

2. 개체가 요청을 처리할지 결정합니다.

   요청 URI가 선택한 클래스의 @Path 주석 값보다 긴 경우 리소스 메서드의 @Path 주석 값이 요청을 처리할 수 있는 하위 리소스를 찾는 데 사용됩니다.

   하나 이상의 하위 리소스 메서드가 요청 URI와 일치하는 경우 이러한 메서드는 세 번째 단계의 입력으로 사용됩니다.

   요청 URI에 대한 유일한 일치 항목이 하위 리소스 locaters인 경우 하위 리소스 locater에서 생성한 오브젝트의 리소스 메서드가 요청 URI와 일치하도록 합니다. 이 단계는 하위 리소스 메서드가 요청 URI와 일치할 때까지 반복됩니다.

3. 요청을 처리할 리소스 메서드를 선택합니다.

   HTTP 동사 주석이 요청의 HTTP 동사와 일치하는 리소스 메서드입니다. 또한 선택한 리소스 방법은 요청 엔터티 본문의 미디어 유형을 수락해야 하며 요청에 지정된 미디어 유형을 준수하는 응답을 생성할 수 있어야 합니다.

46.6.3. 여러 리소스 클래스에서 선택

선택 알고리즘의 처음 두 단계에서는 요청을 처리할 리소스를 결정합니다. 경우에 따라 리소스는 리소스 클래스에 의해 구현됩니다.**In some cases the resource is implemented by a resource class.** 다른 경우에는 동일한 URI 템플릿을 사용하는 하나 이상의 하위 리소스에 의해 구현됩니다. 요청 URI와 일치하는 리소스가 여러 개 있는 경우 리소스 클래스가 하위 리소스보다 우선합니다.

*리소스 클래스와 하위 리소스 간의 정렬 후 두 개 이상의 리소스가 요청 **URI**와 일치하는 경우 다음 조건을 사용하여 단일 리소스를 선택합니다.*

1. *__URI__ 템플릿에서 가장 리터럴 문자를 사용하는 리소스를 선호합니다.*

   리터럴 문자는 템플릿 변수의 일부가 아닌 문자입니다. 예를 들어 __/widgets/{id}/{color}__ 에는 __10__ 개의 리터럴 문자가 있으며 __/widgets/1/{color}__ 는 __11__개의 리터럴 문자가 있습니다. 따라서 요청 __URI /widgets/1/red__ 는 __URI__ 템플릿으로 __/widgets/1/{color}__ 리소스와 일치합니다.

   참고

   후행 슬래시(__/__)는 리터럴 문자로 간주됩니다. 따라서 __/joefred/__ 가 __/joefred__ 에 우선합니다.

2. *__URI__ 템플릿에서 가장 많은 변수를 사용하는 리소스를 선호합니다.*

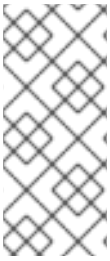
   요청 __URI /widgets/30/green__ 은 __/widgets/{id}/{color}__ 및 __/widgets/{amount}/__ 와 일치할 수 있습니다. 그러나 __URI__ 템플릿 __/widgets/{id}/{color}__ 가 있는 리소스는 두 개의 변수가 있으므로 선택됩니다.

3. *정규 표현식을 포함하는 대부분의 변수로 리소스를 선호합니다.*

   *요청 __URI /widgets/30/green__ 은 __/widgets/{number}/{color}__ 및 __/widgets/{id:}/{color}*__와 일치시킬 수 있습니다. 그러나 __URI__ 템플릿 __*/widgets/{id:.}/{color}__ 인 리소스는 정규 표현식을 포함하는 변수가 있기 때문에 선택됩니다.*

46.6.4. 여러 리소스 방법에서 선택

대부분의 경우 요청 __URI__와 일치하는 리소스를 선택하면 요청을 처리할 수 있는 단일 리소스 메서드가 생성됩니다. 이 방법은 요청에 지정된 __HTTP__ 동사와 리소스 메서드의 __HTTP__ 동사와 일치시켜 결정됩니다. 적절한 __HTTP__ 동사 주석 외에도 선택한 메서드는 요청에 포함된 요청 엔터티를 처리하고 요청의 메타데이터에 지정된 적절한 유형의 응답을 생성할 수 있어야 합니다.



참고

리소스 메서드에서 처리할 수 있는 요청 엔티티 유형은 **@Consumes** 주석으로 지정됩니다. 리소스 메서드에서 생성할 수 있는 응답 유형은 **@ inspector** 주석을 사용하여 지정합니다.

리소스를 선택할 때 요청을 처리할 수 있는 여러 메서드가 생성되면 다음 조건을 사용하여 요청을 처리할 리소스 메서드를 선택합니다.**When selecting a resource produces multiple methods that can handle a request the following criteria is used to select the resource method that will handle the request:**

1. 하위 리소스보다 리소스 메서드를 선호합니다.

2. 하위 리소스 **locaters**를 통한 하위 리소스 방법을 선호합니다.

3. **@Consumes** 주석 및 **@Produces** 주석에서 가장 구체적인 값을 사용하는 메서드를 선호합니다.

   예를 들어 **@Consumes("text/xml")** 주석이 있는 메서드는 **@Consumes("text/*")** 주석이 있는 메서드보다 선호됩니다. 두 방법 모두 **@Consumes** 주석 또는 주석 **@Consumes("*/*")** 가 없는 메서드보다 우선합니다.

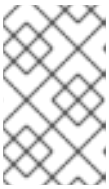
4. 요청 본문 엔티티의 콘텐츠 유형과 가장 근접한 메서드를 선호합니다.

   참고

   요청 본문 엔티티의 콘텐츠 유형은 **HTTP Content-Type** 속성에 지정됩니다.

5. 응답으로 허용되는 콘텐츠 유형과 가장 근접한 방법을 선호합니다.

   참고

   응답에서 허용되는 콘텐츠 유형은 **HTTP Accept** 속성에서 지정됩니다.

46.6.5. 선택 프로세스 사용자 정의

개발자가 여러 리소스 클래스를 선택하는 방식에서 알고리즘을 다소 제한적으로 보고한 경우도 있습니다. 예를 들어 지정된 리소스 클래스가 일치되고 이 클래스에 일치하는 리소스 메서드가 없는 경우 알고리즘이 실행을 중지합니다. 일치하는 나머지 리소스 클래스를 확인하지 않습니다.

Apache CXF는 **org.apache.cxf.jaxrs.ext.ResourceComparator** 인터페이스를 제공하며 런타임이 여러 일치하는 리소스 클래스를 처리하는 방법을 사용자 지정하는 데 사용할 수 있습니다. 예 46.9. "리소스 선택을 사용자 정의할 수 있는 인터페이스"에 표시된 **ResourceComparator** 인터페이스에는 구현해야 하는 방법이 있습니다. 하나는 두 리소스 클래스를 비교하고 다른 하나는 두 개의 리소스 메서드를 비교합니다.

예 46.9. 리소스 선택을 사용자 정의할 수 있는 인터페이스

```
package org.apache.cxf.jaxrs.ext;

import org.apache.cxf.jaxrs.model.ClassResourceInfo;
import org.apache.cxf.jaxrs.model.OperationResourceInfo;
import org.apache.cxf.message.Message;

public interface ResourceComparator
{
  int compare(ClassResourceInfo cri1,
              ClassResourceInfo cri2,
              Message message);

  int compare(OperationResourceInfo oper1,
              OperationResourceInfo oper2,
              Message message);
}
```

사용자 정의 구현은 다음과 같이 두 리소스 사이에서 선택합니다.

- 첫 번째 매개변수가 두 번째 매개변수보다 더 나은 경우 **1** 을 반환합니다.
- 두 번째 매개변수가 첫 번째 매개변수보다 더 나은 경우 **-1** 을 반환합니다.

0 이 반환되는 경우 런타임은 기본 선택 알고리즘을 사용합니다.

서비스의 **jaxrs:server** 요소에 **resourceComparator** 하위를 추가하여 사용자 정의 **ResourceComparator** 구현을 등록합니다.

47장. 리소스 클래스 및 메서드에 정보 전달

초록

JAX-RS는 개발자가 리소스로 전달되는 정보를 제어할 수 있는 여러 주석을 지정합니다. 주석은 URI의 매트릭스 매개변수와 같은 일반적인 HTTP 개념을 따릅니다. 표준 API를 사용하면 메서드 매개 변수, 빈 속성 및 리소스 클래스 필드에 주석을 사용할 수 있습니다. Apache CXF는 빈에 삽입할 매개 변수 시퀀스를 주입할 수 있는 확장을 제공합니다.

47.1. 데이터 삽입의 기본 사항

47.1.1. 개요

HTTP 요청 메시지의 데이터를 사용하여 초기화되는 매개변수, 필드 및 빈 속성에는 런타임을 통해 해당 값이 삽입됩니다. 삽입된 특정 데이터는 47.2절. “JAX-RS API 사용” 에 설명된 주석 세트로 지정됩니다.

JAX-RS 사양은 데이터가 삽입될 때 몇 가지 제한 사항을 적용합니다. 또한 요청 데이터를 삽입할 수 있는 오브젝트 유형에 대한 몇 가지 제한 사항을 배치합니다.

47.1.2. 데이터가 삽입되는 경우When data is injected

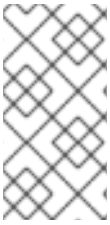
요청 데이터는 요청으로 인해 인스턴스화될 때 개체에 삽입됩니다. 즉, 리소스에 직접 해당하는 오브젝트만 삽입 주석을 사용할 수 있습니다. 46장. 리소스 생성 에서 설명한 것처럼 이러한 오브젝트는 @Path 주석으로 장식되는 루트 리소스 또는 하위 리소스 로케이터 메서드에서 반환된 오브젝트입니다.

47.1.3. 지원되는 데이터 유형

데이터가 삽입될 수 있는 특정 데이터 유형 세트는 삽입된 데이터의 소스를 지정하는 데 사용되는 주석에 따라 달라집니다. 그러나 모든 삽입 주석은 다음 데이터 유형 세트를 지원합니다.

- int,char 또는 long과 같은 프리미티브
- 단일 String 인수를 허용 하는 생성자가 있는 개체입니다.Objects that have a constructor that accepts a single String argument.
- 단일 String 인수를 허용 하는 정적 valueOf() 메서드가 있는 개체입니다.Objects that have a static valueOf() method that accepts a single String argument.

- *List<T>, Set<T> 또는 SortedSet< T > 목록에서 다른 조건을 충족하는 개체*



참고

삽입 주석에 지원되는 데이터 유형에 대한 요구 사항이 다른 경우 주석 설명에서 차이점이 강조 표시됩니다.

47.2. JAX-RS API 사용

47.2.1. JAX-RS 주석 유형

표준 JAX-RS API는 필드, 빈 속성 및 메서드 매개 변수에 값을 삽입하는 데 사용할 수 있는 주석을 지정합니다. 주석은 다음 세 가지 유형으로 나눌 수 있습니다.

- *47.2.2절. "요청 URI에서 데이터 삽입"*
- *47.2.3절. "HTTP 메시지 헤더에서 데이터 삽입"*
- *47.2.4절. "HTML 양식에서 데이터 삽입"*

47.2.2. 요청 URI에서 데이터 삽입

47.2.2.1. 개요

RESTful 웹 서비스를 설계하는 모범 사례 중 하나는 각 리소스에 고유한 URI가 있어야 한다는 것입니다. 개발자는 이 원칙을 사용하여 기본 리소스 구현에 많은 정보를 제공할 수 있습니다. 리소스에 대한 URI 템플릿을 설계할 때 개발자는 리소스 구현에 삽입할 수 있는 매개 변수 정보를 포함하도록 템플릿을 빌드할 수 있습니다. 개발자는 리소스 구현에 정보를 제공하는 데 쿼리 및 매트릭스 매개 변수를 활용할 수도 있습니다.

47.2.2.2. URI 경로에서 데이터 가져오기

리소스에 대한 정보를 얻는 가장 일반적인 메커니즘 중 하나는 리소스에 대한 URI 템플릿을 생성하는 데 사용되는 변수를 사용하는 것입니다. 이 작업은 javax.ws.rs.PathParam 주석을 사용하여 수행됩니다. @PathParam 주석에는 데이터가 삽입될 URI 템플릿 변수를 식별하는 단일 매개 변수가 있습니다.

예 47.1. "URI 템플릿 변수에서 데이터 삽입" 에서 **@PathParam** 주석은 **URI** 템플릿 변수 색상 의 값이 **itemColor** 필드에 삽입되도록 지정합니다.

예 47.1. URI 템플릿 변수에서 데이터 삽입

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

@Path("/boxes/{shape}/{color}")
class Box
{
  ...

  @PathParam("color")
  String itemColor;

  ...
}
```

@PathParam 주석에서 지원하는 데이터 유형은 *"지원되는 데이터 유형"* 에 설명된 것과 다릅니다. **@PathParam** 주석을 삽입하는 엔터티는 다음 유형 중 하나여야 합니다.

- **PathSegment**

  값은 일치하는 경로의 최종 세그먼트입니다.

- **List<PathSegment>**

  값은 **named template** 매개변수와 일치하는 경로 세그먼트에 해당하는 **PathSegment** 오브젝트 목록입니다.

- **int,char** 또는 **long**과 같은 프리미티브

- 단일 **String** 인수를 허용 하는 생성자가 있는 개체입니다.**Objects that have a constructor that accepts a single String argument.**

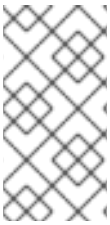
- 단일 **String** 인수를 허용 하는 정적 **valueOf()** 메서드가 있는 개체입니다.**Objects that have a static valueOf() method that accepts a single String argument.**

47.2.2.3. 쿼리 매개변수 사용

웹에서 정보를 전달하는 일반적인 방법은 **URI**에 쿼리 매개 변수 를 사용하는 것입니다. 쿼리 매개 변수는 **URI** 끝에 표시되며, 물음표(**?**)로 **URI**의 리소스 위치 부분과 구분됩니다. 이름과 값이 등호(**=**)로 구분된 하나 이상의 이름 값으로 구성됩니다. 두 개 이상의 쿼리 매개 변수를 지정하면 쌍은 세미콜론(**;**) 또는 앰퍼샌드(**&**)로 서로 구분됩니다. 예 47.2. "쿼리 문자열이 있는 URI" 쿼리 매개 변수가 있는 **URI** 구문을 보여줍니다.

예 47.2. 쿼리 문자열이 있는 URI

```
http://fusesource.org?name=value;name2=value2;...
```



참고

세미콜론 또는 앰퍼샌드를 사용하여 쿼리 매개 변수를 구분할 수 있지만 둘 다 사용할 수는 없습니다.

javax.ws.rs.QueryParam 주석은 쿼리 매개 변수의 값을 추출하여 **JAX-RS** 리소스에 삽입합니다. 주석은 값이 추출되고 지정된 필드, 빈 속성 또는 매개변수에 삽입되는 쿼리 매개변수의 이름을 식별하는 단일 매개 변수를 사용합니다. **@QueryParam** 주석은 "지원되는 데이터 유형" 에 설명된 유형을 지원합니다.

예 47.3. "쿼리 매개변수의 데이터를 사용하는 리소스 메서드" 메서드의 **id** 매개 변수에 쿼리 매개 변수 **id** 값을 삽입하는 리소스 메서드를 보여줍니다.

예 47.3. 쿼리 매개변수의 데이터를 사용하는 리소스 메서드

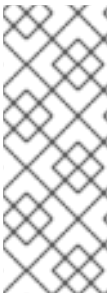
```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  @Path("/{type}")
  public void updateMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
```

```
    ...
  }
  ...
}
```

HTTP POST 를 **/monstersforhire/daikaiju?id=jonas updateMonster()** 메서드의 유형 은 **daikaiju** 로 설정하고 **id** 는 **jonas** 로 설정됩니다.

47.2.2.4. matrix 매개변수 사용

URI 쿼리 매개변수와 같은 **URI** 매트릭스 매개 변수는 리소스를 선택하는 추가 정보를 제공할 수 있는 이름/값 쌍입니다. 쿼리 매개 변수와는 달리 매트릭스 매개 변수는 **URI**의 아무 곳에나 나타날 수 있으며 세미콜론(;)을 사용하여 **URI**의 계층적 경로 세그먼트와 분리됩니다. **/mostersforhire/daikaiju;id=jonas** 에는 **id** 및 **/monstersforhire/japan;type=daikaiju/flying;wingspan=40** 이라는 두 개의 매트릭스 매개변수가 있습니다.**wingspan=40**에는 **type** 및 **wingspan** 이라는 두 개의 매트릭스 매개변수가 있습니다.



참고

리소스의 **URI**를 계산할 때 매트릭스 매개변수는 평가되지 않습니다. 따라서 요청 **URI** **/monstersforhire/japan;type=daikaiju/flying;wingspan=40** 은 **/monstersforhire/japan/flying** 을 처리하는 데 사용되는 적절한 리소스를 찾는 데 사용됩니다.

matrix 매개변수 값은 **javax.ws.rs.MatrixParam** 주석을 사용하여 필드, 매개 변수 또는 빈 속성에 삽입됩니다. 주석은 값이 추출되고 지정된 필드, 빈 속성 또는 매개변수에 삽입되는 **matrix** 매개변수의 이름을 식별하는 단일 매개 변수를 사용합니다. **@MatrixParam** 주석은 "지원되는 데이터 유형" 에 설명된 유형을 지원합니다.

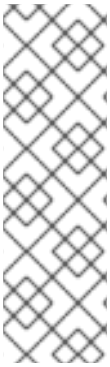
예 47.4. "매트릭스 매개변수의 데이터를 사용하는 리소스 메서드" 메서드의 매개 변수에 매트릭스 매개 변수 유형 및 **id** 값을 삽입하는 리소스 메서드를 보여줍니다.

예 47.4. 매트릭스 매개변수의 데이터를 사용하는 리소스 메서드

```
import javax.ws.rs.MatrixParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
```

```
public void updateMonster(@MatrixParam("type") String type,
                          @MatrixParam("id") String id)
{
  ...
}
...
}
```

HTTP POST 를 **/monstersforhire;type=daikaiju;id=whale the updateMonster()** 메서드의 유형 은 **daikaiju** 로 설정하고 **id** 는 **whale** 으로 설정됩니다.



참고

JAX-RS는 **URI**의 모든 매트릭스 매개 변수를 한 번에 평가하므로 **URI**에서 매트릭스 매개 변수 위치에 제약 조건을 적용할 수 없습니다. 예: **/monstersforhire/japan;type=daikaiju/flying;wingspan=40** , **/monstersforhire/flying;type=daikaiju;type=daikaiju;wingspan=40** **type=daikaiju;wingspan=40/flying** 은 모두 **JAX-RS API**를 사용하여 구현된 **RESTful** 웹 서비스에서 동등한 것으로 처리됩니다.

47.2.2.5. URI 디코딩 비활성화

기본적으로 모든 요청 **URI**가 디코딩됩니다. 따라서 **URI /monster/night%20stalker** 와 **URI /monster/night stalker** 는 동일합니다. 자동 **URI** 디코딩을 사용하면 **ASCII** 문자 집합 이외의 문자를 매개 변수로 쉽게 보낼 수 있습니다.

URI를 자동으로 디코딩하지 않으려면 **javax.ws.rs.Encoded** 주석을 사용하여 **URI** 디코딩을 비활성화할 수 있습니다. 주석은 다음 수준에서 **URI** 디코딩을 비활성화하는 데 사용할 수 있습니다.

- 클래스 수준 **- @Encoded** 주석으로 클래스를 분리하면 클래스의 모든 매개변수, 필드 및 빈 속성에 대한 **URI** 디코딩이 비활성화됩니다.
- 메서드 수준 **- @Encoded** 주석을 사용하여 메서드를 분리하면 클래스의 모든 매개변수에 대한 **URI** 디코딩이 비활성화됩니다.
- 매개 변수/필드 수준**- @Encoded** 주석을 사용하여 매개 변수 또는 필드를 분리하면 클래스의 모든 매개변수에 대한 **URI** 디코딩이 비활성화됩니다.

예 47.5. “URI 디코딩 비활성화” **getMonster()** 메서드가 **URI** 디코딩을 사용하지 않는 리소스를 보여

줍니다. addMonster() 메서드는 type 매개 변수에 대한 URI 디코딩만 비활성화합니다.

예 47.5. URI 디코딩 비활성화

```
@Path("/monstersforhire/")
public class MonsterService
{
  ...

  @GET
  @Encoded
  @Path("/{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @QueryParam("id") String id)
  {
    ...
  }

  @PUT
  @Path("/{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @QueryParam("id") String id)
  {
    ...
  }
  ...
}
```

47.2.2.6. 오류 처리

URI 주입 주석 중 하나를 사용하여 데이터를 삽입하려고 할 때 오류가 발생하면 WebApplicationException 예외를 래핑하는 원래 예외가 생성됩니다. WebApplicationException 예외 상태가 404 로 설정되어 있습니다.

47.2.3. HTTP 메시지 헤더에서 데이터 삽입

47.2.3.1. 개요

일반적으로 요청 메시지의 HTTP 헤더는 메시지에 대한 일반 정보, 전송 중 처리 방법, 예상 응답에 대한 세부 정보를 전달합니다. 일반적으로 몇 가지 표준 헤더가 인식되고 사용되는 반면 HTTP 사양에서는 이름/값 쌍을 HTTP 헤더로 사용할 수 있습니다. JAX-RS API는 HTTP 헤더 정보를 리소스 구현에 삽입하는 쉬운 메커니즘을 제공합니다.

가장 일반적으로 사용되는 HTTP 헤더 중 하나는 쿠키입니다. 쿠키를 통해 HTTP 클라이언트 및 서버는 여러 요청/응답 시퀀스에서 정적 정보를 공유할 수 있습니다. JAX-RS API는 리소스에서 리소스 구현에 직접 데이터를 삽입하는 주석을 제공합니다.

47.2.3.2. HTTP 헤더에서 정보 삽입

javax.ws.rs.HeaderParam 주석은 **HTTP** 헤더 필드의 데이터를 매개 변수, 필드 또는 빈 속성에 삽입하는 데 사용됩니다. 값을 추출하여 리소스 구현에 삽입되는 **HTTP** 헤더 필드의 이름을 지정하는 단일 매개 변수가 있습니다. 관련 매개변수, 필드 또는 빈 속성은 "지원되는 데이터 유형" 에 설명된 데이터 유형을 준수해야 합니다.

If-Modified-Since 헤더 삽입 **HTTP If-Modified-Since** 헤더의 값을 클래스의 **oldestDate** 필드에 삽입하는 코드를 표시합니다.

If-Modified-Since 헤더 삽입

```
import javax.ws.rs.HeaderParam;
...
class RecordKeeper
{
  ...
  @HeaderParam("If-Modified-Since")
  String oldestDate;
  ...
}
```

47.2.3.3. 쿠키에서 정보 삽입

쿠키는 특별한 유형의 **HTTP** 헤더입니다. 첫 번째 요청의 리소스 구현에 전달되는 하나 이상의 이름/값 쌍으로 구성됩니다. 첫 번째 요청 후 쿠키는 공급자와 각 메시지 간에 소비자 간에 전달됩니다. 요청을 생성하므로 소비자만 쿠키를 변경할 수 있습니다. 쿠키는 일반적으로 여러 요청/응답 시퀀스에서 세션을 유지하고, 사용자 설정 및 유지할 수 있는 기타 데이터를 저장하는 데 사용됩니다.

javax.ws.rs.CookieParam 주석은 쿠키 필드에서 값을 추출하여 리소스 구현에 삽입합니다. 값을 추출할 쿠키 필드의 이름을 지정하는 단일 매개 변수를 사용합니다. "지원되는 데이터 유형" 에 나열된 데이터 유형 외에도 **@ Cookie Param** 로 장식되는 엔티티도 쿠키 객체가 될 수 있습니다.

예 47.6. "쿠키 삽입" 처리 쿠키의 값을 **CB** 클래스의 필드에 삽입하는 코드를 표시합니다.

예 47.6. 쿠키 삽입

```
import javax.ws.rs.CookieParam;
...
```

```
class CB
{
  ...
  @CookieParam("handle")
  String handle;
  ...
}
```

47.2.3.4. 오류 처리

HTTP 메시지 삽입 주석 중 하나를 사용하여 데이터를 삽입하려고 할 때 오류가 발생하면 **WebApplicationException** 예외를 래핑하는 원래 예외가 생성됩니다. **WebApplicationException** 예외 상태가 **400** 으로 설정되어 있습니다.

47.2.4. HTML 양식에서 데이터 삽입

47.2.4.1. 개요

HTML 양식은 사용자로부터 정보를 얻는 쉬운 수단이며, 또한 쉽게 만들 수 있습니다. 양식 데이터는 **HTTP GET** 요청 및 **HTTP POST** 요청에 사용할 수 있습니다.

GET

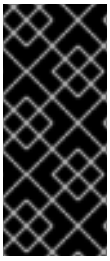
양식 데이터가 **HTTP GET** 요청의 일부로 전송되면 데이터는 쿼리 매개 변수 세트로 **URI**에 추가됩니다. 쿼리 매개변수에서 데이터를 삽입하는 방법은 "쿼리 매개변수 사용" 에서 참조하십시오.

POST

폼 데이터가 **HTTP POST** 요청의 일부로 전송되면 데이터는 **HTTP** 메시지 본문에 저장됩니다. 양식 데이터는 양식 데이터를 지원하는 일반 엔터티 매개 변수를 사용하여 처리할 수 있습니다. **@FormParam** 주석을 사용하여 데이터를 추출하고 리소스 메서드 매개 변수에 조각을 삽입하여 처리할 수도 있습니다.

47.2.4.2. @FormParam 주석을 사용하여 양식 데이터를 주입

javax.ws.rs.FormParam 주석은 양식 데이터에서 필드 값을 추출하고 값을 리소스 메서드 매개 변수에 삽입합니다. 주석은 값을 추출하는 필드의 키를 지정하는 단일 매개 변수를 사용합니다. 관련 매개변수는 "지원되는 데이터 유형" 에 설명된 데이터 유형을 준수해야 합니다.



중요

JAX-RS API Javadoc은 **@FormParam** 주석을 필드, 메서드 및 매개 변수에 배치할 수 있다고 설명합니다. 그러나 **@FormParam** 주석은 리소스 메서드 매개 변수에 배치된 경우에만 의미가 있습니다.

47.2.4.3. 예제

리소스 메서드 매개 변수에 양식 데이터 삽입 폼 데이터를 해당 매개 변수에 삽입하는 리소스 메서드를 표시합니다.**Shows a resource method that injects form data into its parameters.** 이 메서드는 클라이언트 양식에 문자열 데이터가 포함된 세 개의 필드이름,태그 및 본문이 포함되어 있다고 가정합니다.

리소스 메서드 매개 변수에 양식 데이터 삽입

```
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;

...
@POST
public boolean updatePost(@FormParam("title") String title,
                          @FormParam("tags") String tags,
                          @FormParam("body") String post)
{
  ...
}
```

47.2.5. 삽입할 기본값 지정

47.2.5.1. 개요

보다 강력한 서비스 구현을 제공하기 위해 선택적 매개 변수를 기본값으로 설정할 수 있습니다. 특히 긴 **URI** 문자열을 입력하는 경우 오류가 발생하기 때문에 쿼리 매개 변수 및 매트릭스 매개 변수에서 가져온 값에 특히 유용할 수 있습니다. 또한 쿠키에서 추출된 매개 변수의 기본값을 설정하는 것이 좋습니다. 요청 시스템에 모든 값으로 쿠키를 구성하는 적절한 정보가 없기 때문입니다.

javax.ws.rs.DefaultValue 주석은 다음 삽입 주석과 함께 사용할 수 있습니다.

- *@PathParam*
- *@QueryParam*
- *@MatrixParam*
- *@FormParam*
- *@HeaderParam*
- *@CookieParam*

@DefaultValue 주석은 삽입 주석에 해당하는 데이터가 요청에 없는 경우 사용할 기본값을 지정합니다.

47.2.5.2. 구문

매개변수의 기본값을 설정하는 구문 @DefaultValue 주석 사용에 대한 구문을 보여줍니다.

매개변수의 기본값을 설정하는 구문

```
import javax.ws.rs.DefaultValue;
 ...
 void resourceMethod(@MatrixParam("matrix")
                     @DefaultValue("value)
                     int someValue, ... )
 ...
```

주석은 매개변수, 빈 또는 필드 앞에 있어야 합니다. 첨부된 삽입 주석과 관련된 @DefaultValue 주석의 위치는 중요하지 않습니다.

@DefaultValue 주석은 단일 매개변수를 사용합니다. 이 매개변수는 삽입 주석을 기반으로 적절한 데이터를 추출할 수 없는 경우 필드에 삽입되는 값입니다. 값은 모든 문자열 값일 수 있습니다. 값은 연결된 필드의 유형과 호환되어야 합니다. 예를 들어 연결된 필드가 int 이면 기본값으로 파 란색으로 인해 예외가 발생합니다.

47.2.5.3. 목록 및 세트 처리

주석이 달린 매개 변수 유형, 빈 또는 필드가 List, Set 또는 SortedSet인 경우 결과 컬렉션에는 제공된 기본값에서 매핑된 단일 항목이 있습니다.

47.2.5.4. 예제

기본값 설정 에서는 @DefaultValue 를 사용하여 값이 삽입되는 필드의 기본값을 지정하는 두 가지 예를 보여줍니다.

기본값 설정

```
import javax.ws.rs.DefaultValue;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/monster")
public class MonsterService
{

  @Get
  public Monster getMonster(@QueryParam("id") @DefaultValue("42") int id,
                            @QueryParam("type") @DefaultValue("bogeyman") String type)
  {
    ...
  }

  ...
}
```

기본값 설정 의 getMonster() 메서드는 GET 요청이 baseURI/monster 로 전송될 때 호출됩니다. 이 메서드에는 두 개의 쿼리 매개 변수 id 와 type 가 필요하며 URI에 추가됩니다. URI URI/monster?id=1&type=fomóiri를 사용하는 GET 요청은 Fomóiri 와 함께 Fomóiri를 반환합니다.

@DefaultValue 주석은 두 매개 변수에 모두 배치되므로 쿼리 매개변수를 생략하면 getMonster() 메서드가 작동할 수 있습니다. baseURI/monster로 전송된 GET 요청은 URI baseURI /monster?id=42&type=bogeyman 을 사용하여 GET 요청과 동일합니다.

47.2.6. Java Bean에 매개 변수 삽입

47.2.6.1. 개요

REST를 통해 HTML 양식을 게시할 때 서버 측의 일반적인 패턴은 양식에서 수신한 모든 데이터를 캡슐화하기 위한 Java 빈(및 다른 매개 변수 및 HTML 헤더에서 데이터일 수 있음)을 생성하는 것입니다. 일반적으로 이 Java 빈을 생성하는 것은 두 단계 프로세스입니다. 리소스 메서드는 주입(예: @FormParam 주석을 메서드 매개 변수에 추가하여) 양식 값을 수신하며, 리소스 메서드는 빈의 생성자를 호출하여 양식 데이터를 전달합니다.

JAX-RS 2.0 @BeanParam 주석을 사용하면 단일 단계에서 이 패턴을 구현할 수 있습니다. 양식 데이터는 빈 클래스 필드에 직접 삽입할 수 있으며, JAX-RS 런타임에서 빈 자체는 자동으로 생성됩니다. 예를 들어 가장 쉽게 설명할 수 있습니다.

47.2.6.2. injectiontion 대상

@BeanParam 주석은 리소스 메서드 매개 변수, 리소스 필드 또는 빈 속성에 연결할 수 있습니다. 그러나 매개변수 대상은 모든 리소스 클래스 라이프사이클에 사용할 수 있는 유일한 종류의 대상입니다. 다른 종류의 대상은 요청별 라이프사이클로 제한됩니다. 이 상황은 표 47.1. "@BeanParam injection 대상" 에 요약되어 있습니다.

표 47.1. @BeanParam injection 대상

대상	리소스 클래스 라이프사이클
매개변수	All
FIELD	요청당 (기본값)
제품 상세 정보	요청당 (기본값)

47.2.6.3. BeanParam 주석이 없는 예

다음 예제에서는 @BeanParam을 사용하지 않고 기존 접근 방식을 사용하여 Java 빈에서 양식 데이터를 캡처하는 방법을 보여줍니다.

```
// Java
import javax.ws.rs.POST;
```

```
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Response;
...
@POST
public Response orderTable(@FormParam("orderId")  String orderId,
                           @FormParam("color")    String color,
                           @FormParam("quantity") String quantity,
                           @FormParam("price")    String price)
{
    ...
    TableOrder bean = new TableOrder(orderId, color, quantity, price);
    ...
    return Response.ok().build();
}
```

*이 예에서 **orderTable** 방법은 가구 웹 사이트에서 테이블 수량을 주문하는 데 사용되는 양식을 처리합니다. 주문 폼이 게시되면 양식 값이 **orderTable** 메서드의 매개 변수에 삽입되고 **orderTable** 메서드는 삽입된 양식 데이터를 사용하여 **TableOrder** 클래스의 인스턴스를 명시적으로 만듭니다.*

47.2.6.4. BeanParam 주석 사용 예

*위 예제를 리팩터링하여 **@BeanParam** 주석을 활용할 수 있습니다. **@BeanParam** 접근법을 사용하는 경우 양식 매개 변수를 빈 클래스의 필드에 직접 삽입할 수 있습니다. 실제로 **@PathParam,@QueryParam,@FormParam,@MatrixParam,@CookieParam, @HeaderParam , @HeaderParam** 등의 빈 클래스에서 표준 **JAX-RS** 매개변수 주석을 사용할 수 있습니다. 양식을 처리하는 코드는 다음과 같이 리팩토링될 수 있습니다.*

```
// Java
import javax.ws.rs.POST;
import javax.ws.rs.FormParam;
import javax.ws.rs.core.Response;
...
public class TableOrder {
    @FormParam("orderId")
    private String orderId;

    @FormParam("color")
    private String color;

    @FormParam("quantity")
    private String quantity;

    @FormParam("price")
    private String price;

    // Define public getter/setter methods
    // (Not shown)
    ...
}
...
@POST
```

```
public Response orderTable(@BeanParam TableOrder orderBean)
{
   ...
   // Do whatever you like with the 'orderBean' bean
   ...
   return Response.ok().build();
}
```

이제 양식 주석이 빈 클래스인 **TableOrder**에 추가되었으므로 리소스 메서드 서명의 모든 **@FormParam** 주석을 단일 **@BeanParam** 주석으로 교체할 수 있습니다. 이제 폼이 **orderTable** 리소스 메서드에 게시되면 **JAX-RS** 런타임에서 자동으로 **TableOrder** 인스턴스, **orderBean; orderBean** 를 만들고, 빈 클래스에 매개변수 주석에 의해 지정된 모든 데이터를 삽입합니다.

47.3. 매개변수 변환기

47.3.1. 개요

매개 변수 변환기를 사용하면 모든 유형의 필드, 빈 속성 또는 리소스 메서드 인수에 매개변수(문자열 유형)를 삽입할 수 있습니다. 적절한 매개 변수 변환기를 구현하고 바인딩하여 매개 변수 **String** 값을 대상 유형으로 변환할 수 있도록 **JAX-RS** 런타임을 확장할 수 있습니다.

47.3.2. 자동 변환

매개변수는 **String** 의 인스턴스로 수신되므로 항상 **String** 유형의 필드, 빈 속성 및 메서드 매개변수에 직접 삽입할 수 있습니다. 또한 **JAX-RS** 런타임에는 매개 변수 문자열을 다음 유형으로 자동 변환하는 기능이 있습니다.

1. 기본 유형.
2. 단일 **String** 인수를 수락하는 생성자가 있는 형식입니다.**Types that have a constructor that accepts a single String argument.**
3. 형식의 인스턴스를 반환하는 단일 **String** 인수가 있는 **valueOf** 또는 **fromString** 이라는 정적 메서드가 있는 **Type**입니다.**Types that have a static method named valueOf or fromString with a single String argument that returns an instance of the type.**
4. **List<T > , Set<T > ,** 또는 **SortedSet < T >, T** 가 **2** 또는 **3**에 설명된 유형 중 하나인 경우.

47.3.3. 매개 변수 변환기

자동 변환에서 다루지 않는 형식에 매개 변수를 삽입하려면 해당 형식에 대한 사용자 지정 매개 변수 변환기를 정의할 수 있습니다.**In order to inject a parameter into a type not covered by automatic conversion, you can define a custom parameter converter for the type.** 매개 변수 변환기는 **String** 에서 사용자 지정 형식으로 변환 정의 및 사용자 지정 형식에서 **String**으로의 변환을 정의할 수 있는 **JAX-RS** 확장입니다.**A parameter converter is a JAX-RS extension that enables you to define conversion from String to a custom type, and also in the reverse direction, from the custom type to a String.**

47.3.4. 팩토리 패턴

JAX-RS 매개 변수 변환기 메커니즘은 팩토리 패턴을 사용합니다. 따라서 매개 변수 변환기를 직접 등록하는 대신 요청 시 매개 변수 변환기(**type, javax.ws.rs.ext.ParamConverterProvider**)를 만드는 매개 변수 변환기(**type, javax.ws.rs.ext.ParamConverter**)를 생성해야 합니다.

47.3.5. ParamConverter 인터페이스

javax.ws.rs.ext.ParamConverter 인터페이스는 다음과 같이 정의됩니다.

```
// Java
package javax.ws.rs.ext;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.ws.rs.DefaultValue;

public interface ParamConverter<T> {

    @Target({ElementType.TYPE})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    public static @interface Lazy {}

    public T fromString(String value);

    public String toString(T value);
}
```

자체 **ParamConverter** 클래스를 구현하려면 **fromString** 메서드를 재정의하여 매개 변수 문자열을 대상 유형으로 변환하고 대상 유형을 문자열로 다시 변환 해야 합니다.

47.3.6. ParamConverterProvider 인터페이스

javax.ws.rs.ext.ParamConverterProvider 인터페이스는 다음과 같이 정의됩니다.

```
// Java
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

public interface ParamConverterProvider {
   public <T> ParamConverter<T> getConverter(Class<T> rawType, Type genericType, Annotation
annotations[]);
}
```

자체 ParamConverterProvider 클래스를 구현하려면 이 인터페이스를 구현해야 합니다. 이 인터페이스는 ParamConverter 인스턴스를 생성하는 팩토리 메서드인 getConverter 메서드를 재정의해야 합니다.

47.3.7. 매개 변수 변환기 공급자 바인딩

매개 변수 변환기 공급자를 JAX-RS 런타임(애플리케이션에서 사용할 수 있도록 하는)에 바인딩 하려면 다음과 같이 구현 클래스에 @Provider 주석을 추가해야 합니다.

```
// Java
...
import javax.ws.rs.ext.ParamConverterProvider;
import javax.ws.rs.ext.Provider;

@Provider
public class TargetTypeProvider implements ParamConverterProvider {
   ...
}
```

이 주석은 배포 단계에서 매개변수 변환기 공급자가 자동으로 등록되도록 합니다.

47.3.8. 예제

다음 예제에서는 매개 변수 문자열을 TargetType 형식으로 변환 하거나 변환할 수 있는 기능이 있는 ParamConverter 및 ParamConverter를 구현하는 방법을 보여줍니다.

```
// Java
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

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

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

@Provider
public class TargetTypeProvider implements ParamConverterProvider {

    @Override
    public <T> ParamConverter<T> getConverter(
        Class<T> rawType,
        Type genericType,
        Annotation[] annotations
    ) {
        if (rawType.getName().equals(TargetType.class.getName())) {
            return new ParamConverter<T>() {

                @Override
                public T fromString(String value) {
                    // Perform conversion of value
                    // ...
                    TargetType convertedValue = // ... ;
                    return convertedValue;
                }

                @Override
                public String toString(T value) {
                    if (value == null) { return null; }
                    // Assuming that TargetType.toString is defined
                    return value.toString();
                }
            };
        }
        return null;
    }

}
```

47.3.9. 매개 변수 변환기 사용

이제 **TargetType**에 대한 매개변수 변환기를 정의했으므로 다음과 같이 **TargetType** 필드 및 인수에 직접 매개변수를 삽입할 수 있습니다.

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
...
@POST
public Response updatePost(@FormParam("target") TargetType target)
{
  ...
}
```

47.3.10. 기본값의 지연 변환

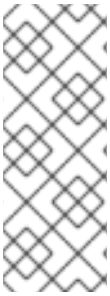
매개변수의 기본값(**@DefaultValue** 주석을 사용하여) 기본값을 지정하는 경우 기본값을 바로 대상 유형(기본 동작)으로 변환하거나 필요한 경우에만 기본값을 변환해야 하는지(**lazy** 변환)할지 여부를 선택할 수 있습니다. 지연 변환을 선택하려면 대상 유형에 **@ParamConverter.Lazy** 주석을 추가합니다. 예를 들면 다음과 같습니다.

```
// Java
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.ext.ParamConverter.Lazy;
...
@POST
public Response updatePost(
    @FormParam("target")
    @DefaultValue("default val")
    @ParamConverter.Lazy
    TargetType target)
{
  ...
}
```

47.4. APACHE CXF 확장 사용

47.4.1. 개요

Apache CXF는 개발자가 일련의 주입 주석을 단일 주석으로 교체할 수 있는 표준 **JAX-WS** 주입 메커니즘에 대한 확장을 제공합니다. 단일 주석은 주석을 사용하여 추출된 데이터의 필드가 포함된 빈에 배치됩니다. 예를 들어 리소스 방법에 **id,type, size** 라는 세 개의 쿼리 매개 변수를 포함하는 요청 **URI**가 필요한 경우 단일 **@QueryParam** 주석을 사용하여 모든 매개 변수를 해당 필드가 있는 빈에 삽입할 수 있습니다.



참고

대신 **@BeanParam** 주석을 사용하는 것이 좋습니다(**JAX-RS 2.0** 이후 사용 가능). 표준화된 **@BeanParam** 접근 방식은 독점형 **Apache CXF** 확장보다 유연하므로 권장되는 대안입니다. 자세한 내용은 47.2.6절. “Java Bean에 매개 변수 삽입”의 내용을 참조하십시오.

47.4.2. 지원되는 삽입 주석

이 확장은 모든 삽입 매개 변수를 지원하지 않습니다. 다음 항목만 지원합니다.

- **@PathParam**

- *@QueryParam*
- *@MatrixParam*
- *@FormParam*

47.4.3. 구문

주석이 빈에 직렬 주입을 사용함을 나타내기 위해 다음 두 가지 작업을 수행해야 합니다.

1. *주석의 매개변수를 빈 문자열로 지정합니다. 예를 들어 **@PathParam("")** 은 일련의 **URI** 템플릿 변수를 빈으로 직렬화하도록 지정합니다.*
2. *주석이 달린 매개변수가 삽입되는 값과 일치하는 필드가 있는 빈인지 확인합니다.*

47.4.4. 예제

*예 47.7. "빈에 쿼리 매개변수 삽입" 에서는 여러 쿼리 매개변수를 빈에 삽입하는 예를 보여줍니다. 리소스 메서드는 요청 **URI**가 두 개의 쿼리 매개 변수(**type** 및 **id**)를 포함할 것으로 예상합니다. 해당 값은 **Monster** 빈의 해당 필드에 삽입됩니다.*

예 47.7. 빈에 쿼리 매개변수 삽입

```
import javax.ws.rs.QueryParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
...

@Path("/monstersforhire/")
public class MonsterService
{
  ...
  @POST
  public void updateMonster(@QueryParam("") Monster bean)
  {
    ...
  }
  ...
}
```

```
public class Monster
{
  String type;
  String id;

  ...
}
```

48장. 소비자에게 정보 반환

초록

RESTful 요청은 최소한 HTTP 응답 코드를 소비자에게 반환해야 합니다. 대부분의 경우 일반 JAXB 오브젝트 또는 GenericEntity 오브젝트를 반환하여 요청을 충족할 수 있습니다. 리소스 메서드가 응답 엔티티와 함께 추가 메타데이터를 반환해야 하는 경우 JAX-RS 리소스 메서드에서 필요한 HTTP 헤더 또는 기타 메타데이터가 포함된 Response 개체를 반환할 수 있습니다.

48.1. 반환 유형

소비자에게 반환되는 정보는 리소스 메서드가 반환하는 정확한 유형의 개체를 결정합니다. 이는 명확하게 보일 수 있지만 Java 반환 오브젝트 간의 매핑과 RESTful 소비자로 반환되는 것은 일대일이 아닙니다. 최소한 RESTful 소비자는 응답 엔티티 본문 외에도 유효한 HTTP 반환 코드를 반환해야 합니다. Java 객체에 포함된 데이터를 응답 엔티티에 포함된 데이터의 매핑은 소비자가 기꺼이 받아들일 준비가 되어 있는 MIME 타입에 의해 영향을 받습니다.

Java 오브젝트를 RESTful 응답 메시지와 관련된 문제를 해결하기 위해 리소스 메서드는 다음과 같이 네 가지 유형의 Java 구문을 반환할 수 있습니다.

- 48.2절. "일반 Java 구문 반환" JAX-RS 런타임으로 결정되는 HTTP 반환 코드로 기본 정보를 반환합니다.
- 48.2절. "일반 Java 구문 반환" JAX-RS 런타임에 의해 결정된 HTTP 반환 코드로 복잡한 정보를 반환합니다.
- 48.3절. "애플리케이션 응답 미세 조정" 프로그래밍 방식으로 확인된 HTTP 반환 상태로 복잡한 정보를 반환합니다. Response 오브젝트를 사용하면 HTTP 헤더도 지정할 수 있습니다.
- 48.4절. "일반 유형 정보가 있는 엔터티 반환" JAX-RS 런타임에 의해 결정된 HTTP 반환 코드로 복잡한 정보를 반환합니다. GenericEnitity 오브젝트는 데이터를 serialize하는 런타임 구성 요소에 대한 자세한 정보를 제공합니다.

48.2. 일반 JAVA 구문 반환

48.2.1. 개요

대부분의 경우 리소스 클래스는 표준 Java 유형, JAXB 개체 또는 애플리케이션에 엔터티 공급자가 있

는 모든 개체를 반환할 수 있습니다. 이러한 경우 런타임은 반환되는 개체의 Java 클래스를 사용하여 MIME 유형 정보를 결정합니다. 런타임은 또한 소비자에게 보낼 적절한 HTTP 반환 코드를 결정합니다.

48.2.2. 반환 가능한 유형

리소스 메서드는 엔터티 작성기가 제공되는 모든 Java 유형 또는 void 를 반환할 수 있습니다. 기본적으로 런타임에는 다음에 대한 공급자가 있습니다.

- Java 기본 사항
- Java 프리미티브의 Number 표현
- JAXB 오브젝트

"기본적으로 지원되는 유형" 기본적으로 지원되는 모든 반환 유형을 나열합니다. "사용자 정의 작성자" 사용자 지정 엔터티 작성기를 구현하는 방법을 설명합니다.Describes how to implement a custom entity writer.

48.2.3. MIME 유형

런타임은 먼저 @Produces 주석의 리소스 메서드 및 리소스 클래스를 확인하여 반환된 엔터티의 MIME 유형을 결정합니다. 이 중 하나를 찾으면 주석에 지정된 MIME 유형을 사용합니다. 리소스 구현에서 지정한 항목을 찾지 못하면 엔터티 프로바이더에 따라 적절한 MIME 유형을 결정합니다.

기본적으로 런타임은 다음과 같이 MIME 유형을 할당합니다.

- Java 프리미티브 및 해당 Number 표현에는 MIME 유형의 application/octet-stream 이 할당됩니다.
- JAXB 오브젝트에는 application/xml 의 MIME 유형이 할당됩니다.

애플리케이션은 "사용자 정의 작성자" 에 설명된 대로 사용자 지정 엔터티 공급자를 구현하여 다른 매핑을 사용할 수 있습니다.

48.2.4. 응답 코드

리소스 메서드가 일반 Java 구문을 반환하는 경우 런타임은 예외를 throw하지 않고 리소스 메서드가 완료되면 응답의 상태 코드를 자동으로 설정합니다. 상태 코드는 다음과 같이 설정됩니다.

- 204(콘텐츠 없음) - 리소스의 메서드 반환 유형이 void입니다.
- 204(콘텐츠 없음) - 반환된 엔티티의 값이 null입니다.
- 200(OK) - 반환된 엔터티의 값이 null이 아닙니다.

리소스 메서드가 완료되기 전에 예외가 throw되는 경우 50장. 예외 처리 에 설명된 대로 반환 상태 코드가 설정됩니다.

48.3. 애플리케이션 응답 미세 조정

48.3.1. 응답 빌드의 기본 사항

48.3.1.1. 개요

RESTful 서비스는 리소스 메서드가 일반 Java 구문을 반환할 때 허용되는 것보다 소비자에 대해 반환된 응답을 보다 정확하게 제어해야 합니다. JAX-RS Response 클래스를 사용하면 리소스 메서드에서 소비자에게 전송된 반환 상태를 제어하고 HTTP 메시지 헤더와 쿠키를 응답으로 지정할 수 있습니다.

response 개체는 소비자에게 반환되는 엔터티를 나타내는 개체를 래핑합니다. Response objects wrap the object representing the entity that is returned to the consumer. 응답 오브젝트는 Response Builder 클래스를 팩토로 사용하여 인스턴스화됩니다.

ResponseBuilder 클래스에는 응답의 메타데이터를 조작하는 데 사용되는 많은 방법이 있습니다. 예를 들어 ResonseBuilder 클래스에는 HTTP 헤더 및 캐시 제어 지시문을 설정하는 메서드가 포함됩니다.

48.3.1.2. 응답 빌더와 응답 빌더 간 관계

Response 클래스에는 보호된 생성자가 있으므로 직접 인스턴스화할 수 없습니다. 이는 Response 클래스로 묶은 ResponseBuilder 클래스를 사용하여 생성됩니다. ResponseBuilder 클래스는 생성된 응

답으로 캡슐화될 모든 정보에 대한 소유자입니다. **ResponseBuilder** 클래스에는 메시지에서 **HTTP** 헤더 속성을 설정하는 방법도 있습니다.

Response 클래스는 적절한 응답 코드를 쉽게 설정하고 엔터티를 래핑하는 몇 가지 메서드를 제공합니다. 각 공통 응답 상태 코드에 대한 방법이 있습니다. 엔터티 본문 또는 필수 메타데이터를 포함하는 상태에 해당하는 메서드에는 관련 응답 빌더로 정보를 직접 설정할 수 있는 버전이 포함됩니다.

ResponseBuilder 클래스의 **build()** 메서드는 메서드가 호출될 때 응답 빌더에 저장된 정보를 포함하는 **response** 오브젝트를 반환합니다. 응답 오브젝트가 반환되면 응답 빌더가 정리 상태로 반환됩니다.

48.3.1.3. 응답 빌더 가져오기

응답 빌더를 가져오는 방법에는 두 가지가 있습니다.

- **Response** 클래스를 사용하여 응답 빌더 가져오기 에서와 같이 **Response** 클래스의 정적 메서드 사용

  Response 클래스를 사용하여 응답 빌더 가져오기

  ```
  import javax.ws.rs.core.Response;

  Response r = Response.ok().build();
  ```

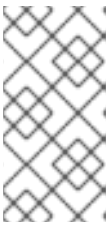
  이러한 방식으로 응답 빌더를 가져올 때 인스턴스에 액세스할 수 없는 경우 여러 단계에서 조작할 수 있습니다. 모든 작업을 단일 메서드 호출로 입력해야 합니다.

- **Apache CXF** 특정 **ResponseBuilderImpl** 클래스 사용. 이 클래스를 사용하면 응답 빌더에서 직접 작업할 수 있습니다. 그러나 모든 응답 빌더 정보를 수동으로 설정해야 합니다.

  예 48.1. “ResponseBuilderImpl 클래스를 사용하여 응답 빌더 가져오기” **ResponseBuilderImpl** 클래스를 사용하여 **Response** 클래스를 사용하여 응답 빌더 가져오기 을 다시 작성할 수 있는 방법을 보여줍니다.

  예 48.1. **ResponseBuilderImpl** 클래스를 사용하여 응답 빌더 가져오기

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(200);
Response r = builder.build();
```



참고

Response 클래스의 메서드에서 반환된 **ResponseBuilder** 를 **Response BuilderImpl** 오브젝트에 간단히 할당할 수도 있습니다.

48.3.1.4. 더 알아보기

Response 클래스에 대한 자세한 내용은 **Response** 클래스의 **Javadoc** 을 참조하십시오.

ResponseBuilder 클래스에 대한 자세한 내용은 **ResponseBuilder** 클래스의 **Javadoc** 을 참조하십시오.

Apache CXF ResponseBuilderImI 클래스에 대한 자세한 내용은 **ResponseBuilderImpl Javadoc** 을 참조하십시오.

48.3.2. 일반적인 사용 사례에 대한 응답 생성

48.3.2.1. 개요

Response 클래스는 **RESTful** 서비스에 필요한 더 일반적인 응답을 처리하기 위한 바로 가기 메서드를 제공합니다. 이러한 메서드는 제공된 값 또는 기본값을 사용하여 적절한 헤더 설정을 처리합니다. 또한 적절한 경우 엔티티 본문을 채우기를 처리합니다.

48.3.2.2. 성공적인 요청에 대한 응답 생성

요청이 성공적으로 처리되면 요청이 충족되었음을 인정하기 위해 응답을 보내야 합니다. 이 응답은 엔터티를 포함할 수 있습니다.

응답을 성공적으로 완료할 때 가장 일반적인 응답은 **OK** 입니다. **OK** 응답에는 일반적으로 요청에 해당하는 엔터티가 포함됩니다. **Response** 클래스에는 응답 상태를 **200** 으로 설정하고 첨부된 응답 빌더에

제공된 엔터티를 추가하는 오버로드된 **ok()** 메서드가 있습니다. **ok()** 메서드의 다섯 가지 버전이 있습니다. 가장 일반적으로 사용되는 변형은 다음과 같습니다.

- **response .ok()- 200** 상태 및 빈 엔티티 본문으로 응답을 만듭니다.
- **response .ok(java.lang.Object** 엔티티) **-** 응답이 **200** 인 응답을 만들고, 제공된 오브젝트를 응답 엔티티 본문에 저장하고, 오브젝트를 검사하여 엔티티 유형을 결정합니다.

200 응답을 사용하여 응답 생성 에서는 **OK** 상태로 응답을 생성하는 예를 보여줍니다.

200 응답을 사용하여 응답 생성

```
import javax.ws.rs.core.Response;
import demo.jaxrs.server.Customer;
...

Customer customer = new Customer("Jane", 12);

return Response.ok(customer).build();
```

요청자가 엔티티를 요구하지 않는 경우, **200 OK** 상태 대신 **204 No Content status**를 보내는 것이 더 적절할 수 있습니다. **Response.noContent()** 메서드는 적절한 응답 오브젝트를 생성합니다.

204 상태로 응답 생성 는 **204** 상태로 응답을 생성하는 예를 보여줍니다.

204 상태로 응답 생성

```
import javax.ws.rs.core.Response;

return Response.noContent().build();
```

48.3.2.3. 리디렉션에 대한 응답 생성

Response 클래스는 리디렉션 응답 상태 3개를 처리하는 메서드를 제공합니다.

303 다른 보기

303 기타 보기 상태는 요청된 리소스가 요청을 처리하도록 소비자를 새 리소스로 영구적으로 리디렉션해야 하는 경우에 유용합니다.

Response 클래스는 **Other ()** 메서드를 참조 하며 **303** 상태의 응답을 생성하고 새 리소스 **URI**를 메시지의 위치 필드에 배치합니다. **see Other()** 메서드는 새 **URI**를 **java.net.URI** 오브젝트로 지정하는 단일 매개 변수를 사용합니다.

304 수정되지 않음

304 Not Modified 상태는 요청 특성에 따라 다양한 용도로 사용될 수 있습니다. 이전 **GET** 요청 이후 요청된 리소스가 변경되지 않았음을 나타내는 데 사용할 수 있습니다. 또한 리소스를 수정하라는 요청을 통해 리소스가 변경되지 않았음을 나타낼 수도 있습니다.

Response 클래스 **notModified()** 메서드는 **304** 상태로 응답을 생성하고 **HTTP** 메시지에 수정된 날짜 속성을 설정합니다. **notModified()** 메서드의 세 가지 버전이 있습니다.

- **notModified**
- **notModifiedjavax.ws.rs.core.Entitytag**
- **notModifiedjava.lang.Stringtag**

307 임시 리디렉션

307 임시 리디렉션 상태는 요청된 리소스가 소비자를 새 리소스로 전달해야 하지만 소비자가 향후 요청을 처리하기 위해 이 리소스를 계속 사용하고자 하는 경우 유용합니다.

응답 클래스 **temporaryRedirect()** 메서드는 **307** 상태로 응답을 생성하고 새 리소스 **URI**를 메시지의 위치 필드에 배치합니다. **temporaryRedirect()** 메서드는 새 **URI**를 **java.net.URI** 오브젝트로 지정하는 단일 매개 변수를 사용합니다.

304 상태로 응답 생성 는 **304** 상태로 응답을 생성하는 예를 보여줍니다.

304 상태로 응답 생성

```
import javax.ws.rs.core.Response;

return Response.notModified().build();
```

48.3.2.4. 신호 오류에 대한 응답 생성

Response 클래스는 두 가지 기본 처리 오류에 대한 응답을 생성하는 메서드를 제공합니다.

- **serverError- 500 Internal Server Error** 상태로 응답을 만듭니다.
- **notAcceptablejava.util.List<javax.ws.rs.core.Variant>변형- 허용되는 리소스 유형 목록이 포함된 406 Not Acceptable** 상태 및 엔터티 본문으로 응답을 만듭니다.

500 상태의 응답 생성 는 **500** 상태의 응답을 생성하는 예를 보여줍니다.

500 상태의 응답 생성

```
import javax.ws.rs.core.Response;

return Response.serverError().build();
```

48.3.3. 더 많은 고급 응답 처리

48.3.3.1. 개요

Response 클래스 메소드는 일반적인 사례에 대한 응답을 생성하기 위한 짧은 컷을 제공합니다. 캐시

제어 지시문 지정, 사용자 지정 HTTP 헤더 추가 또는 Response 클래스에서 처리되지 않은 상태를 전송하는 등의 복잡한 경우를 처리해야 하는 경우 build() 메서드를 사용하여 응답 오브젝트를 생성하기 전에 ResponseBuilder 클래스 메서드를 사용하여 응답을 채워야 합니다.

"응답 빌더 가져오기" 에서 설명한 대로 Apache CXF ResponseBuilderImpl 클래스를 사용하여 직접 조작할 수 있는 응답 빌더 인스턴스를 생성할 수 있습니다.

48.3.3.2. 사용자 정의 헤더 추가

ResponseBuilder 클래스의 header() 메서드를 사용하여 사용자 정의 헤더가 응답에 추가됩니다. header() 메서드는 두 개의 매개변수를 사용합니다.

- name- 헤더 이름을 지정하는 문자열입니다.
- value- 헤더에 저장된 데이터를 포함하는 Java 오브젝트

header() 메서드를 반복적으로 호출하여 메시지에 여러 개의 헤더를 설정할 수 있습니다.

응답에 헤더 추가 응답에 헤더를 추가하는 코드를 표시합니다.

응답에 헤더 추가

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.header("username", "joe");
Response r = builder.build();
```

48.3.3.3. 쿠키 추가

ResponseBuilder 클래스의 cookie() 메서드를 사용하여 사용자 정의 헤더가 응답에 추가됩니다. cookie() 방법은 하나 이상의 쿠키를 사용합니다. 각 쿠키는 javax.ws.rs.core.NewCookie 개체에 저장됩니다. NewCookie 클래스의 contructors의 가장 쉬운 방법은 두 가지 매개 변수를 사용합니다.

- **name-** 쿠키 이름을 지정하는 문자열입니다.
- **value-** 쿠키 값을 지정하는 문자열입니다.

cookie() 메서드를 반복적으로 호출하여 여러 쿠키를 설정할 수 있습니다.

응답에 쿠키 추가 응답에 쿠키를 추가하는 코드를 보여줍니다.

응답에 쿠키 추가

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.NewCookie;

NewCookie cookie = new NewCookie("username", "joe");

Response r = Response.ok().cookie(cookie).build();
```

주의

null 매개 변수 목록을 사용하여 **cookie()** 메서드를 호출하면 이미 응답과 연결된 모든 쿠키가 지워집니다.

48.3.3.4. 응답 상태 설정

Response 클래스의 도우미 메서드에서 지원하는 상태 중 하나를 제외한 상태를 반환하려는 경우 **Response Builder** 클래스의 **status()** 메서드를 사용하여 응답의 상태 코드를 설정할 수 있습니다. **status()** 메서드에는 두 가지 변형이 있습니다. 응답 코드를 지정하는 **int** 를 가져옵니다.**Gets an int that specifies the response code.** 다른 하나는 **Response.Status** 오브젝트를 사용하여 응답 코드를 지정합니다.

Response.Status 클래스는 **Response** 클래스에 포함된 열거형입니다. 대부분의 정의된 **HTTP** 응답

코드에 대한 항목이 있습니다.

응답에 헤더 추가 응답 상태를 **404 Not Found** 로 설정하기 위한 코드를 보여줍니다.

응답에 헤더 추가

```
import javax.ws.rs.core.Response;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.status(404);
Response r = builder.build();
```

48.3.3.5. 캐시 제어 지시문 설정

ResponseBuilder 클래스의 **cacheControl()** 메서드를 사용하면 응답에서 캐시 제어 헤더를 설정할 수 있습니다. **cacheControl()** 메서드는 응답에 대한 캐시 제어 지시문을 지정하는 **javax.ws.rs.CacheControl** 개체를 사용합니다.

CacheControl 클래스에는 **HTTP** 사양에서 지원하는 모든 캐시 제어 지시문에 해당하는 메서드가 있습니다. 지시문은 **setter** 메서드에서 부울 값을 사용하는 간단한 **on** 또는 **off** 값입니다. 지시문에 **max-age** 지시문과 같은 숫자 값이 필요한 경우 **setter**는 **int** 값을 사용합니다.

응답에 헤더 추가 **no-store** 캐시 제어 지시문을 설정하기 위한 코드를 표시합니다.

응답에 헤더 추가

```
import javax.ws.rs.core.Response;
import javax.ws.rs.core.CacheControl;
import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;

CacheControl cache = new CacheControl();
cache.setNoCache(true);

ResponseBuilderImpl builder = new ResponseBuilderImpl();
builder.cacheControl(cache);
Response r = builder.build();
```

48.4. 일반 유형 정보가 있는 엔터티 반환

48.4.1. 개요

애플리케이션에서 반환된 개체의 MIME 유형 또는 응답을 serialize하는 데 사용되는 엔터티 공급자를 더 많이 제어해야 하는 경우가 있습니다. JAX-RS javax.ws.rs.core.GenericEntity<T > 클래스는 엔터티를 나타내는 개체의 일반 유형을 지정하기 위한 메커니즘을 제공하여 엔터티의 직렬화를 제어할 수 있습니다.

48.4.2. GenericEntity<T> 오브젝트 사용

응답을 serialize하는 엔터티 공급자를 선택하는 데 사용되는 기준 중 하나는 개체의 일반 유형입니다. 개체의 일반 유형은 개체의 Java 유형을 나타냅니다. 공용 Java 유형 또는 JAXB 개체가 반환되면 런타임에서 Java 리플렉션을 사용하여 제네릭 형식을 결정할 수 있습니다. 그러나 JAX-RS Response 개체가 반환되면 런타임에서 래핑된 엔터티의 일반 유형을 확인할 수 없으며 개체의 실제 Java 클래스가 Java 유형으로 사용됩니다.

엔터티 공급자에 올바른 일반 유형 정보가 제공되도록 하기 위해 엔터티를 GenericEntity<T > 개체에 래핑할 수 있습니다.

리소스 메서드는 GenericEntity<T> 개체를 직접 반환할 수도 있습니다.Resource methods can also directly return a GenericEntity<T > object. 실제로 이 방법은 거의 사용되지 않습니다. GenericEntity<T> 개체에서 래핑되지 않은 엔터티에 대해 저장된 일반 유형 정보와 일반적으로 GenericEntity<T> 개체에 래핑된 엔터티에 저장된 일반 유형 정보는 일반적으로 동일합니다.The generic type information determined by reflect of an unwrapped entity and the generic type information stored for an entity wrapped in a GenericEntity<T > object are typically the same.

48.4.3. GenericEntity<T> 오브젝트 생성

GenericEntity<T> 오브젝트를 생성하는 방법에는 두 가지가 있습니다.

1. 래핑되는 엔터티를 사용하여 GenericEntity<T > 클래스의 하위 클래스를 생성합니다. 서브클래스를 사용하여 GenericEntity<T> 오브젝트 생성 런타임 시 제네릭 형식을 사용할 수 있는 List<String> 형식의 엔터티가 포함된 GenericEntity<T > 개체를 만드는 방법을 보여 줍니

다.**Shows how to create a GenericEntity<T> object containing an entity of type List<String> whose generic type will be available at runtime.**

서브 클래스를 사용하여 **GenericEntity<T>** 오브젝트 생성

```
import javax.ws.rs.core.GenericEntity;

List<String> list = new ArrayList<String>();
...
GenericEntity<List<String>> entity =
  new GenericEntity<List<String>>(list) {};
Response response = Response.ok(entity).build();
```

GenericEntity<T >를 생성하는 데 사용되는 서브 클래스는 일반적으로 익명입니다.

2.

제네릭 유형 정보를 엔티티와 함께 제공하여 인스턴스를 직접 생성합니다. 예 **48.2. "GenericEntity<T> 개체를 직접 인스턴스화합니다."** **AtomicInteger** 유형의 엔티티가 포함된 응답을 만드는 방법을 보여줍니다.

예 48.2. GenericEntity<T> 개체를 직접 인스턴스화합니다.

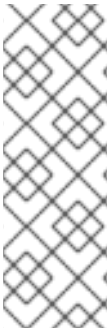
```
import javax.ws.rs.core.GenericEntity;

AtomicInteger result = new AtomicInteger(12);
GenericEntity<AtomicInteger> entity =
  new GenericEntity<AtomicInteger>(result,
                                   result.getClass().getGenericSuperclass());
Response response = Response.ok(entity).build();
```

48.5. 비동기 응답

48.5.1. 서버의 비동기 처리

48.5.1.1. 개요

서버 측에서 비동기 호출을 처리하는 목적은 스레드를 보다 효율적으로 사용할 수 있도록 하고 궁극적으로 서버의 요청 스레드가 모두 차단되기 때문에 클라이언트 연결 시도가 거부되는 시나리오를 방지하는 것입니다. 호출이 비동기적으로 처리되면 요청 스레드가 거의 즉시 해제됩니다.



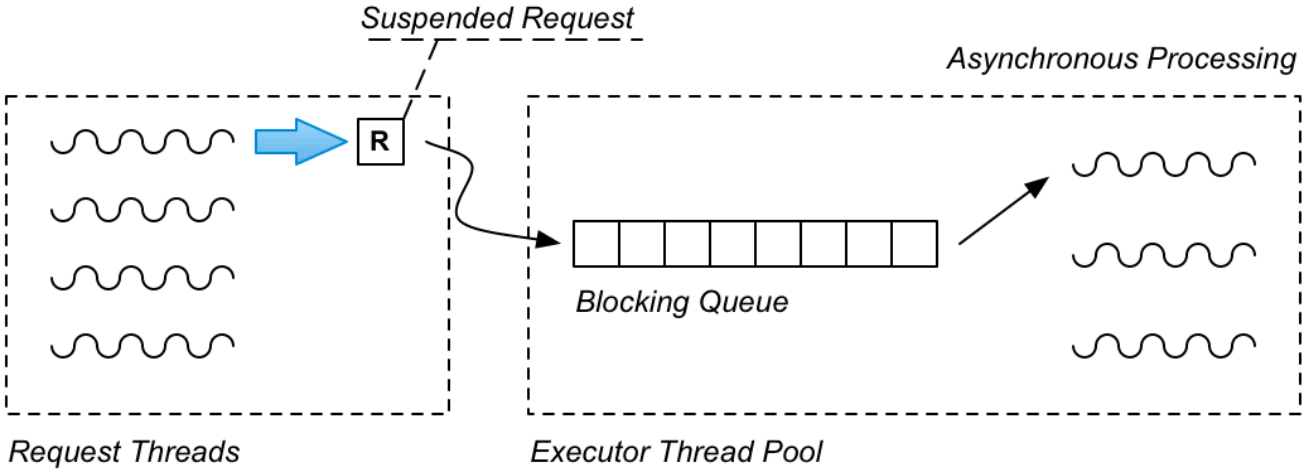
참고

서버 측에서 비동기 처리를 사용하도록 설정해도 서버에서 응답을 수신 할 때까지 클라이언트는 여전히 차단된 상태로 유지됩니다. 클라이언트 쪽에서 비동기 동작을 보려면 클라이언트 쪽 비동기 처리를 구현해야 합니다.**If you want to see asynchronous behavior on the client side, you must implement client-side asynchronous processing.** **49.6절. "클라이언트의 비동기 처리"** 을 참조하십시오.

48.5.1.2. 비동기 처리를 위한 기본 모델

그림 **48.1.** "비동기 처리를 위한 스레딩 모델" 서버 측에서 비동기 처리를 위한 기본 모델에 대한 개요를 보여줍니다.

그림 **48.1.** 비동기 처리를 위한 스레딩 모델



개요에서 요청은 비동기 모델에서 다음과 같이 처리됩니다.**In outline, a request is processed as follows in the asynchronous model:**

1. 비동기 리소스 메서드는 요청 스레드 내에서 호출되고 **AsyncResponse** 개체에 대한 참조를 받습니다.**An asynchronous resource method is invoked within a request thread (and receives a reference to an AsyncResponse object, which will be needed later to send the response).**

2. 리소스 메서드는 요청을 처리하는 데 필요한 모든 정보 및 처리 논리를 포함하는 실행 가능 개체에서 일시 중지된 요청을 캡슐화합니다.

3. 리소스 메서드는 실행 가능한 개체를 **executor** 스레드 풀의 차단 대기열로 푸시합니다.

4. 이제 리소스 메서드가 반환되어 요청 스레드가 해제될 수 있습니다.

5. 실행 가능 개체가 큐의 맨 위에 도달하면 **executor** 스레드 풀의 스레드 중 하나에 의해 처리됩니다.**When the Runnable object gets to the top of the queue, it is processed by one of the threads in the executor thread pool.** 그런 다음 캡슐화된 **AsyncResponse** 오브젝트를 사용하여 응답을 클라이언트에 다시 보냅니다.

48.5.1.3. Java executor를 사용한 스레드 풀 구현

java.util.concurrent API는 매우 쉽게 전체 스레드 풀 구현을 생성할 수 있는 강력한 **API**입니다. **Java** 동시성 **API**의 용어에서 스레드 풀을 **execut or**라고 합니다. 이는 작업 스레드 및 해당 스레드를 제공하는 차단 대기열을 포함하여 전체 작업 스레드 풀을 만들려면 단일 코드 줄만 필요합니다.

예를 들어 그림 48.1. “비동기 처리를 위한 스레딩 모델”에 표시된 **Executor Thread Pool** 과 같은 전체 작업 스레드 풀을 생성하려면 다음과 같이 **java.util.concurrent.Executor** 인스턴스를 만듭니다.

```
Executor executor = new ThreadPoolExecutor(
    5,                          // Core pool size
    5,                          // Maximum pool size
    0,                          // Keep-alive time
    TimeUnit.SECONDS,                   // Time unit
    new ArrayBlockingQueue<Runnable>(10)  // Blocking queue
);
```

이 생성자는 스레드 **5**개가 포함된 새 스레드 풀을 만들고, 단일 차단 대기열에 의해 제공되며 최대 **10**개의 **Runnable** 개체를 보유할 수 있습니다. 스레드 풀에 작업을 제출하려면 **executor.execute** 메서드를 호출하여 실행 가능한 개체(**asynchronous** 작업을 캡슐화하는 실행 가능 오브젝트)에 대한 참조를 전달합니다.

48.5.1.4. 비동기 리소스 메서드 정의

비동기적인 리소스 메서드를 정의하려면 **@Suspended** 주석을 사용하여 **javax.ws.rs.container.AsyncResponse** 유형의 인수를 주입하고 메서드가 **void** 를 반환하는지 확인합니다. 예를 들면 다음과 같습니다.

```
// Java
...
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
```

```
@Path("/bookstore")
public class BookContinuationStore {
   ...
   @GET
   @Path("{id}")
   public void handleRequestInPool(@PathParam("id") String id,
                                   @Suspended AsyncResponse response) {
      ...
   }
   ...
}
```

삽입된 **AsyncResponse** 오브젝트가 나중에 응답을 반환하는 데 사용되므로 리소스 메서드가 **void** 를 반환해야 합니다.

48.5.1.5. AsyncResponse 클래스

javax.ws.rs.container.AsyncResponse 클래스는 들어오는 클라이언트 연결에서 추상 핸들을 제공합니다. **AsyncResponse** 오브젝트가 리소스 메서드에 삽입되면 기본 **TCP** 클라이언트 연결이 초기에 일시 중지된 상태입니다. 나중에 응답을 반환할 준비가 되면 기본 **TCP** 클라이언트 연결을 다시 활성화하고 **AsyncResponse** 인스턴스에서 **resume** 를 호출하여 응답을 다시 전달할 수 있습니다. 또는 호출을 중단해야 하는 경우 **AsyncResponse** 인스턴스에서 취소 를 호출할 수 있습니다.

48.5.1.6. 일시 중단된 요청을 실행 가능으로 캡슐화

그림 48.1. "비동기 처리를 위한 스레딩 모델" 에 표시된 비동기 처리 시나리오에서는 일시 중단된 요청을 전용 스레드 풀에서 나중에 처리할 수 있는 큐로 푸시합니다. 그러나 이러한 접근 방식이 작동하려면 개체에서 일시 중단된 요청 을 캡슐화하는 방법이 필요합니다. 일시 중단된 요청 오브젝트는 다음 사항을 캡슐화해야 합니다.

- 들어오는 요청의 매개 변수(있는 경우).
- 들어오는 클라이언트 연결에 대한 핸들과 응답을 다시 보내는 방법을 제공하는 **AsyncResponse** 오브젝트입니다.
- 호출 논리입니다.

이러한 사항을 캡슐화하는 편리한 방법은 **Runnable** 클래스를 정의하여 일시 중지된 요청을 나타내는 것입니다. 여기서 **Runnable.run()** 메서드는 호출 논리를 캡슐화하는 것입니다. 가장 우아한 방법은 다음 예제와 같이 로컬 클래스로 **Runnable** 을 구현하는 것입니다.

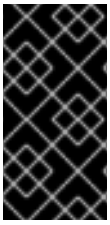
48.5.1.7. 비동기 처리 예

비동기 처리 시나리오를 구현하려면 리소스 메서드 구현에서 실행 가능한 실행 가능 오브젝트를 **executor** 스레드 풀에 전달해야 합니다.**To implement the asynchronous processing scenario, the implementation of the resource method must pass a Runnable object (representing the suspended request) to the executor thread pool. Java 7** 및 **8**에서는 다음 예제와 같이 일부 새로운 구문을 사용하여 **Runnable** 클래스를 로컬 클래스로 정의할 수 있습니다.

```
// Java
package org.apache.cxf.systest.jaxrs;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executor;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.CompletionCallback;
import javax.ws.rs.container.ConnectionCallback;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

import org.apache.cxf.phase.PhaseInterceptorChain;

@Path("/bookstore")
public class BookContinuationStore {

    private Map<String, String> books = new HashMap<String, String>();
    private Executor executor = new ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS,
                                new ArrayBlockingQueue<Runnable>(10));

    public BookContinuationStore() {
        init();
    }
    ...
    @GET
    @Path("{id}")
    public void handleRequestInPool(final @PathParam("id") String id,
                                    final @Suspended AsyncResponse response) {
        executor.execute(new Runnable() {
            public void run() {
                // Retrieve the book data for 'id'
                // which is presumed to be a very slow, blocking operation
                // ...
                bookdata = ...
                // Re-activate the client connection with 'resume'
```

```
            // and send the 'bookdata' object as the response
            response.resume(bookdata);
        }
      });
   }
   ...
}
```

*리소스 메서드 인수, **id** 및 **response** 가 **Runnable** 로컬 클래스 정의로 직접 전달되는 방법을 확인합니다. 이 특수 구문을 사용하면 로컬 클래스에서 해당 필드를 정의하지 않고도 **Runnable.run()** 메서드에서 직접 리소스 메서드 인수를 사용할 수 있습니다.*



중요

*이 특수 구문이 작동하려면 리소스 메서드 매개 변수를 **final** 로 선언 해야 합니다(즉, 메서드 구현에서 변경되지 않아야 함).*

48.5.2. 시간 초과 및 시간 제한 핸들러

48.5.2.1. 개요

*비동기 처리 모델은 **REST** 호출 시 시간 초과도 지원합니다. 기본적으로 시간 초과로 인해 **HTTP** 오류 응답이 클라이언트로 다시 전송됩니다. 그러나 시간 초과 처리기 콜백을 등록하는 옵션도 있으므로 시간 초과 이벤트에 대한 응답을 사용자 지정할 수 있습니다.**But you also have the option of registering a timeout handler callback, which enables you to customize the response to a timeout event.***

48.5.2.2. 처리기 없이 타임아웃 설정 예

*시간 초과를 지정하지 않고 간단한 호출 타임아웃을 정의하려면 다음 예제와 같이 **AsyncResponse** 오브젝트에서 **setTimeout** 메서드를 호출합니다.*

```
// Java
// Java
...
import java.util.concurrent.TimeUnit;
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
```

```
public class BookContinuationStore {
    ...
    @GET
    @Path("/books/defaulttimeout")
    public void getBookDescriptionWithTimeout(@Suspended AsyncResponse async) {
        async.setTimeout(2000, TimeUnit.MILLISECONDS);
        // Optionally, send request to executor queue for processing
        // ...
    }
    ...
}
```

java.util.concurrent.TimeUnit 클래스에서 언제든지 시간 단위를 사용하여 시간 초과 값을 지정할 수 있습니다. 위 예제는 요청을 **executor** 스레드 풀로 보내는 코드를 표시하지 않습니다. 시간 제한 동작을 테스트하려는 경우 리소스 메서드 본문에 **async.SetTimeout** 호출만 포함하고 모든 호출에 시간 초과가 트리거될 수 있습니다.

AsyncResponse.NO_TIMEOUT 값은 무한한 타임아웃을 나타냅니다.

48.5.2.3. 기본 타임아웃 동작

기본적으로 호출 타임아웃이 트리거되면 **JAX-RS** 런타임에서 **ServiceUnavailableException** 예외를 발생시키고 상태 **503** 을 사용하여 **HTTP** 오류 응답을 반환합니다.

48.5.2.4. TimeoutHandler 인터페이스

시간 제한 동작을 사용자 정의하려면 **TimeoutHandler** 인터페이스를 구현하여 시간 초과 처리기를 정의해야 합니다.

```
// Java
package javax.ws.rs.container;

public interface TimeoutHandler {
    public void handleTimeout(AsyncResponse asyncResponse);
}
```

구현 클래스에서 **handleTimeout** 메서드를 재정의하면 시간 제한을 처리하기 위해 다음 방법 중 하나를 선택할 수 있습니다.

- **asyncResponse.cancel** 메서드를 호출하여 응답을 취소합니다.

- ***response** 값을 사용하여 **asyncResponse.resume** 메서드를 호출하여 응답을 보냅니다.*

- ***asyncResponse.setTimeout** 메서드를 호출하여 대기 기간을 확장합니다. (예를 들어, **10**초 이상 기다리려면 **asyncResponse.setTimeout(10, TimeUnit.SECONDS)**을 호출할 수 있습니다.*

48.5.2.5. 처리기를 사용하여 시간 제한 설정 예

*시간제한 처리기를 사용하여 호출 시간을 정의하려면 다음 예제와 같이 **AsyncResponse** 개체에서 **setTimeout** 메서드 및 **setTimeoutHandler** 메서드를 모두 호출합니다.*

```
// Java
...
import javax.ws.rs.GET;
import javax.ws.rs.NotFoundException;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
   ...
   @GET
   @Path("/books/cancel")
   public void getBookDescriptionWithCancel(@PathParam("id") String id,
                                @Suspended AsyncResponse async) {
      async.setTimeout(2000, TimeUnit.MILLISECONDS);
      async.setTimeoutHandler(new CancelTimeoutHandlerImpl());
      // Optionally, send request to executor queue for processing
      // ...
   }
   ...
}
```

*여기서 이 예제에서는 호출 타임아웃을 처리하기 위해 **CancelTimeoutHandlerImpl** 타임아웃 핸들러의 인스턴스를 등록합니다.*

48.5.2.6. 시간 초과 처리기를 사용하여 응답을 취소

***CancelTimeoutHandlerImpl** 시간제한 핸들러는 다음과 같이 정의됩니다.*

```
// Java
...
```

```
import javax.ws.rs.container.AsyncResponse;
...
import javax.ws.rs.container.TimeoutHandler;

@Path("/bookstore")
public class BookContinuationStore {
    ...
    private class CancelTimeoutHandlerImpl implements TimeoutHandler {

        @Override
        public void handleTimeout(AsyncResponse asyncResponse) {
            asyncResponse.cancel();
        }

    }
    ...
}
```

AsyncResponse 오브젝트에 대한 호출 취소 효과는 **HTTP 503 (**서비스 사용할 수 없음**)** 오류 응답을 클라이언트에 보내는 것입니다. 응답 메시지에 **Retry-After: HTTP** 헤더를 설정하는 데 사용되는 **cancel** 메서드**(int** 또는 **java.util.Date** 값**)**에 대한 인수를 선택적으로 지정할 수 있습니다. 그러나 클라이언트는 종종 **Retry-After:** 헤더를 무시합니다.

48.5.2.7. 실행 가능한 인스턴스에서 취소된 응답 처리

실행 가능 스레드 풀에서 처리를 위해 대기 중인 실행 가능한 인스턴스로 캡슐화된 경우 스레드 풀이 요청을 처리하는 시점에 **AsyncResponse**가 취소되었음을 확인할 수 있습니다.**If you have encapsulated a suspended request as a Runnable instance, which is queued for processing in an executor thread pool, you might find that the AsyncResponse has been canceled by the time the thread pool gets around to processing the request.** 이러한 이유로 **Runnable** 인스턴스에 일부 코드를 추가하여 취소된 **AsyncResponse** 개체에 대처할 수 있습니다. 예를 들면 다음과 같습니다.

```
// Java
...
@Path("/bookstore")
public class BookContinuationStore {
    ...
    private void sendRequestToThreadPool(final String id, final AsyncResponse response) {

        executor.execute(new Runnable() {
            public void run() {
                if ( !response.isCancelled() ) {
                    // Process the suspended request ...
                    // ...
                }
            }
        });

    }
    ...
}
```

48.5.3. 삭제 연결 처리

48.5.3.1. 개요

클라이언트 연결이 끊어진 경우를 처리하기 위해 콜백을 추가할 수 있습니다.

48.5.3.2. ConnectionCallback 인터페이스

삭제된 연결에 대한 콜백을 추가하려면 다음과 같이 정의된 javax.ws.rs.container.ConnectionCallback 인터페이스를 구현해야 합니다.

```
// Java
package javax.ws.rs.container;

public interface ConnectionCallback {
    public void onDisconnect(AsyncResponse disconnected);
}
```

48.5.3.3. 연결 콜백 등록

연결 콜백을 구현한 후 레지스터 방법 중 하나를 호출하여 현재 **AsyncResponse** 오브젝트에 등록해야 합니다. 예를 들어, **type**의 연결 콜백을 등록하려면 **MyConnectionCallback:**

```
asyncResponse.register(new MyConnectionCallback());
```

48.5.3.4. 연결 콜백의 일반적인 시나리오

일반적으로 연결 콜백을 구현하는 주요 이유는 삭제된 클라이언트 연결과 관련된 리소스를 확보할 수 있습니다(여기서 **AsyncResponse** 인스턴스를 사용해야 하는 리소스를 식별하는 키로 사용할 수 있음).

48.5.4. 콜백 등록

48.5.4.1. 개요

호출이 완료되면 알림을 받기 위해 선택적으로 **AsyncResponse** 인스턴스에 콜백을 추가할 수 있습니다. 이 콜백을 호출할 수 있는 경우 처리에는 다음 두 가지 대체 지점이 있습니다.

- 요청 처리가 완료되면 응답이 이미 클라이언트로 다시 전송되었거나,

- *요청 처리가 완료되고 매핑되지 않은 **Throwable** 가 호스팅 **I/O** 컨테이너로 전파되었습니다.*

48.5.4.2. CompletionCallback 인터페이스

완료 콜백을 추가하려면 다음과 같이 정의된 javax.ws.rs.container.CompletionCallback 인터페이스를 구현해야 합니다.

```
// Java
package javax.ws.rs.container;

public interface CompletionCallback {
    public void onComplete(Throwable throwable);
}
```

*일반적으로 **throw** 가능한 인수는 **null** 입니다. 그러나 요청 처리가 매핑되지 않은 예외가 발생한 경우 **throw**할 수 없는 예외 인스턴스가 **throw**됩니다.**However, if the request processing resulted in an unmapped exception, throwable contains the unmapped exception instance.***

48.5.4.3. 완료 콜백 등록

*완료 콜백을 구현한 후 레지스터 방법 중 하나를 호출하여 현재 **AsyncResponse** 오브젝트에 등록해야 합니다. 예를 들어, **type** 완료 콜백을 등록하려면 **MyCompletionCallback:***

```
asyncResponse.register(new MyCompletionCallback());
```

49장. JAX-RS 2.0 클라이언트 API

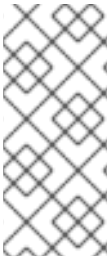
초록

JAX-RS 2.0은 REST 호출 또는 HTTP 클라이언트 호출에 사용할 수 있는 완전한 기능을 갖춘 클라이언트 API를 정의합니다. 여기에는 fluent API(요청 구축 간소화), 메시지 구문 분석을 위한 프레임워크(엔터 제공자라고도 함) 및 클라이언트 측에서 비동기 호출 지원이 포함됩니다.

49.1. JAX-RS 2.0 클라이언트 API 소개

49.1.1. 개요

JAX-RS 2.0은 JAX-RS 클라이언트의 fluent API를 정의하므로 HTTP 요청 단계별로 빌드한 다음 적절한 HTTP 동사(GET, POST, PUT 또는 DELETE)를 사용하여 요청을 호출할 수 있습니다.



참고

jaxrs:client 요소를 사용하여 Blueprint XML 또는 Spring XML에서 JAX-RS 클라이언트를 정의할 수도 있습니다. 이 방법에 대한 자세한 내용은 18.2절. “JAX-RS 클라이언트 엔드 포인트 구성” 을 참조하십시오.

49.1.2. 종속 항목

애플리케이션에서 JAX-RS 2.0 클라이언트 API를 사용하려면 프로젝트의 pom.xml 파일에 다음 Maven 종속성을 추가해야 합니다.

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-rs-client</artifactId>
  <version>3.3.6.fuse-7_11_1-00015-redhat-00002</version>
</dependency>
```

비동기 호출 기능을 사용하려는 경우(49.6절. “클라이언트의 비동기 처리”참조) 다음 Maven 종속성도 필요합니다.

```
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transports-http-hc</artifactId>
  <version>3.3.6.fuse-7_11_1-00015-redhat-00002</version>
</dependency>
```

49.1.3. 클라이언트 API 패키지

JAX-RS 2.0 클라이언트 인터페이스와 클래스는 다음 Java 패키지에 있습니다.

```
javax.ws.rs.client
```

JAX-RS 2.0 Java 클라이언트를 개발하는 경우 일반적으로 코어 패키지에서 클래스에 액세스해야 합니다.

```
javax.ws.rs.core
```

49.1.4. 간단한 클라이언트 요청의 예

다음 코드 조각은 간단한 예를 보여줍니다. JAX-RS 2.0 클라이언트 API는 http://example.org/bookstore JAX-RS 서비스에서 호출하고 GET HTTP 메서드로 호출하는 데 사용됩니다.

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
Response res = client.target("http://example.org/bookstore/books/123")
    .request("application/xml").get();
```

49.1.5. fluent API

JAX-RS 2.0 클라이언트 API는 fluent API (도메인별 언어라고도 함)로 설계되었습니다. fluent API에서는 Java 메서드가 간단한 언어의 명령처럼 보이는 방식으로 Java 메서드 체인이 단일 문으로 호출됩니다. JAX-RS 2.0에서 fluent API는 REST 요청을 빌드하고 호출하는 데 사용됩니다.

49.1.6. REST 호출을 수행하는 단계

JAX-RS 2.0 클라이언트 API를 사용하여 다음과 같이 일련의 단계에서 클라이언트 호출을 빌드하고 호출합니다.

1. 클라이언트를 부트스트랩합니다.

2. *대상을 구성합니다.*

3. *호출을 빌드하고 만듭니다.**Build and make the invocation.***

4. *응답을 구문 분석합니다.*

49.1.7. 클라이언트를 부트스트랩

*첫 번째 단계는 **javax.ws.rs.client.Client** 오브젝트를 생성하여 클라이언트를 부트스트랩하는 것입니다. 이 **Client** 인스턴스는 비교적 많은 가중치 오브젝트로, **JAX-RS** 클라이언트를 지원하는 데 필요한 기술 스택을 나타냅니다(인터셉터 및 추가 **CXF** 기능 포함). 이상적으로 새 오브젝트를 생성하는 대신 클라이언트 개체를 다시 사용할 수 있어야 합니다.*

*새 **Client** 오브젝트를 생성하려면 다음과 같이 **ClientBuilder** 클래스에서 **static** 메서드를 호출합니다.*

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
...
Client client = ClientBuilder.newClient();
...
```

49.1.8. 대상 구성

*대상을 구성하면 **REST** 호출에 사용할 **URI**를 효과적으로 정의합니다. 다음 예제에서는 **path(String)** 메서드를 사용하여 기본 **URI**, 기본, 경로 세그먼트를 기본 **URI**에 추가하는 방법을 보여줍니다.*

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
```

49.1.9. 빌드 및 호출

*이 단계는 실제로 두 단계로 구성됩니다. 먼저 **HTTP** 요청(헤더, 수락된 미디어 유형 포함)을 빌드하고, 두 번째는 관련 **HTTP** 메서드를 호출합니다(필요한 경우 요청 메시지 본문을 제공하는 옵션).*

예를 들어 **application/xml** 미디어 유형을 수락하는 요청을 생성하고 호출하려면 다음을 수행합니다.

```
// Java
import javax.ws.rs.core.Response;
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

49.1.10. 응답 구문 분석

마지막으로 이전 단계에서 얻은 **Respose** 를 구문 분석해야 합니다. 일반적으로 응답은 다른 **HTTP** 메타데이터 및 **HTTP** 메시지 본문(있는 경우)과 함께 **HTTP** 헤더를 캡슐화하는 **javax.ws.rs.core.Response** 오브젝트 형태로 반환됩니다.

String 형식으로 반환된 **HTTP** 메시지에 액세스하려면 다음과 같이 **String.class** 인수를 사용하여 **readEntity** 메서드를 호출하여 쉽게 수행할 수 있습니다.

```
// Java
...
String msg = resp.readEntity(String.class);
```

항상 **String.class** 를 **readEntity** 에 인수로 지정하여 응답의 메시지 본문에 **String**으로 액세스할 수 있습니다. 메시지 본문의 보다 일반적인 변환 또는 변환을 위해 엔터티 공급자를 제공하여 변환을 수행할 수 있습니다.**For more general transformations or conversions of the message body, you can provide an entity provider to perform the conversion.** 자세한 내용은 49.4절. “요청 및 응답 구문 분석” 에서 참조하십시오.

49.2. 클라이언트 대상 빌드

49.2.1. 개요

초기 클라이언트 인스턴스를 만든 후 다음 단계는 요청 **URI**를 빌드하는 것입니다. **WebTarget** 빌더 클래스를 사용하면 **URI** 경로 및 쿼리 매개변수를 포함하여 **URI**의 모든 측면을 구성할 수 있습니다.

49.2.2. WebTarget 빌더 클래스

javax.ws.rs.WebTarget 빌더 클래스는 **fluent API**의 일부를 제공하여 요청에 대한 **REST URI**를 빌드할 수 있도록 합니다.

49.2.3. 클라이언트 대상 생성

WebTarget 인스턴스를 생성하려면 **javax.ws.rs.client.Client** 인스턴스에서 대상 메서드 중 하나를 호출합니다. 예를 들면 다음과 같습니다.

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
```

49.2.4. 기본 경로 및 경로 세그먼트

대상 메서드를 사용하여 하나의 **go**에서 전체 경로를 지정할 수 있습니다. 또는 기본 경로를 지정한 다음 대상 메서드와 경로 메서드의 조합을 사용하여 조각별로 경로 세그먼트 조각을 추가할 수 있습니다. 기본 경로를 경로 세그먼트와 결합할 때의 이점은 약간 다른 대상에서 여러 호출에 기본 경로 **WebTarget** 오브젝트를 쉽게 다시 사용할 수 있다는 점입니다. 예를 들면 다음과 같습니다.

```
// Java
import javax.ws.rs.client.WebTarget;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget headers     = base.path("bookheaders");
// Now make some invocations on the 'headers' target...
...
WebTarget collections = base.path("collections");
// Now make some invocations on the 'collections' target...
...
```

49.2.5. URI 템플릿 매개변수

대상 경로의 구문은 **URI** 템플릿 매개 변수도 지원합니다. 즉, 템플릿 매개변수 **{param}** 을 사용하여 경로 세그먼트를 초기화할 수 있으며, 그러면 나중에 지정된 값으로 해석됩니다. 예를 들면 다음과 같습니다.

```
// Java
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
WebTarget base = client.target("http://example.org/bookstore/");
WebTarget books = base.path("books").path("{id}");
...
Response resp = books.resolveTemplate("id", "123").request("application/xml").get();
```

resolveTemplate 메서드가 경로 세그먼트인 **{id}** 를 값 **123** 로 바꿉니다.

49.2.6. 쿼리 매개변수 정의

쿼리 매개 변수는 단일 ? 문자로 쿼리 매개 변수의 시작을 나타내는 URI 경로에 추가할 수 있습니다.Query parameters can be appended to the URI path, where the beginning of the query parameters is marked by a single ? character. 이 메커니즘을 사용하면 구문을 사용하여 일련의 이름/값 쌍을 설정할 수 있습니다. name1=value1&name2= value2 =value2&...

WebTarget 인스턴스를 사용하면 다음과 같이 queryParam 메서드를 사용하여 쿼리 매개 변수를 정의할 수 있습니다.A WebTarget instance enables you to define query parameters using the queryParam method, as follows:

```
// Java
WebTarget target = client.target("http://example.org/bookstore/")
                        .queryParam("userId","Agamemnon")
                        .queryParam("lang","gr");
```

49.2.7. 매트릭스 매개변수 정의

행렬 매개 변수는 쿼리 매개변수와 다소 유사하지만 널리 지원되지 않으며 다른 구문을 사용합니다. WebTarget 인스턴스에서 matrix 매개 변수를 정의하려면 matrixParam(String, Object) 메서드를 호출합니다.

49.3. 클라이언트 할당 빌드

49.3.1. 개요

WebTarget 빌더 클래스를 사용하여 대상 URI를 빌드한 후 다음 단계는 HTTP 헤더, 쿠키 등 요청의 다른 측면을 구성하는 것입니다. 호출을 빌드하는 마지막 단계는 적절한 HTTP 동사(GET, POST, PUT 또는 DELETE)를 호출하고 필요한 경우 메시지 본문을 제공하는 것입니다.

49.3.2. invocation.Builder 클래스

javax.ws.rs.client.Invocation.Builder 빌더 클래스는 HTTP 메시지의 내용을 빌드하고 HTTP 메서드를 호출할 수 있는 fluent API의 일부를 제공합니다.

49.3.3. 호출 빌더 생성

Invocation.Builder 인스턴스를 생성하려면 javax.ws.rs.client.WebTarget 인스턴스에서 요청 방법 중 하나를 호출합니다. 예를 들면 다음과 같습니다.

```
// Java
import javax.ws.rs.client.WebTarget;
```

```
import javax.ws.rs.client.Invocation.Builder;
...
WebTarget books = client.target("http://example.org/bookstore/books/123");
Invocation.Builder invbuilder = books.request();
```

49.3.4. HTTP 헤더 정의

다음과 같이 헤더 메서드를 사용하여 요청 메시지에 HTTP 헤더 를 추가할 수 있습니다.

```
Invocation.Builder invheader = invbuilder.header("From", "fionn@example.org");
```

49.3.5. 쿠키 정의

다음과 같이 쿠키 방법을 사용하여 요청 메시지에 쿠키 를 추가할 수 있습니다.

```
Invocation.Builder invcookie = invbuilder.cookie("myrestclient", "123xyz");
```

49.3.6. 속성 정의

다음과 같이 속성 메서드를 사용하여 이 요청의 컨텍스트에서 속성을 설정할 수 있습니다.**You can set a property in the context of this request using the property method, as follows:**

```
Invocation.Builder invproperty = invbuilder.property("Name", "Value");
```

49.3.7. 허용되는 미디어 유형, 언어 또는 인코딩 정의

다음과 같이 허용되는 미디어 유형, 언어 또는 인코딩을 정의할 수 있습니다.

```
Invocation.Builder invmedia = invbuilder.accept("application/xml")
                              .acceptLanguage("en-US")
                              .acceptEncoding("gzip");
```

49.3.8. HTTP 메서드 호출

REST 호출을 빌드하는 프로세스는 **HTTP** 호출을 수행하는 **HTTP** 메서드를 호출하여 종료됩니다. 다음 방법(**javax.ws.rs.client.Sync invoker** 기본 클래스에서 상속됨)을 호출할 수 있습니다.

```
get
post
delete
```

```
put
head
trace
options
```

*호출하려는 특정 **HTTP** 동사가 이 목록에 없는 경우 일반 메서드 메서드를 사용하여 모든 **HTTP** 메서드를 호출할 수 있습니다.**If the specific HTTP verb you want to invoke is not on this list, you can use the generic method method to invoke any HTTP method.***

49.3.9. 입력한 응답

*모든 **HTTP** 호출 방법은 형식화되지 않은 변형과 형식화된 변형(추가 인수를 취함)과 함께 제공됩니다. 기본 **get()** 메서드를 사용하여 요청을 호출하는 경우 호출에서 **javax.ws.rs.core.Response** 오브젝트가 반환됩니다. 예를 들면 다음과 같습니다.*

```
Response res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").get();
```

*그러나 **get(Class<T>)** 메서드를 사용하여 응답이 특정 유형으로 반환되도록 요청하는 것 도 가능합니다. 예를 들어 요청을 호출하고 응답을 **BookInfo** 개체로 반환하도록 요청하려면 다음을 수행합니다.*

```
BookInfo res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").get(BookInfo.class);
```

*그러나 이 기능이 작동하려면 응답 형식 **application/xml** 을 요청된 유형으로 매핑할 수 있는 **Client** 인스턴스에 적합한 엔터티 공급자 를 등록해야 합니다. 엔터티 공급자에 대한 자세한 내용은 **49.4**절. “요청 및 응답 구문 분석” 을 참조하십시오.*

49.3.10. 게시 또는 배치에서 발신 메시지 지정

*메시지 본문을 요청에 포함하는 **HTTP** 메서드 (예: **POST** 또는 **PUT**)의 경우 메시지 본문을 메서드의 첫 번째 인수로 지정해야 합니다.**For HTTP methods that include a message body in the request (such as POST or PUT), you must specify the message body as the first argument of the method.** 메시지 본문은 **javax.ws.rs.client.Entity** 오브젝트로 지정해야 합니다. 여기서 **Entity** 는 메시지 내용 및 관련 미디어 유형을 캡슐화합니다. 예를 들어 메시지 내용이 **String** 유형으로 제공되는 **POST** 메서드를 호출하려면 다음을 수행합니다.*

```
import javax.ws.rs.client.Entity;
...
Response res = client.target("http://example.org/bookstore/registerbook")
                .request("application/xml")
                .put(Entity.entity("Red Hat Install Guide", "text/plain"));
```

필요한 경우 **Entity.entity()** 생성자 메서드는 등록된 엔티티 공급자를 사용하여 제공된 메시지 인스턴스를 지정된 미디어 유형에 자동으로 매핑합니다. 메시지 본문을 간단한 문자열 유형으로 지정할 수 있습니다.

49.3.11. 지연된 호출

HTTP 요청을 바로 호출하는 대신(예: **get()** 메서드를 호출하여) 나중에 호출할 수 있는 **javax.ws.rs.client.Invocation** 오브젝트를 생성하는 옵션이 있습니다. **Invocation** 오브젝트는 **HTTP** 메서드를 포함하여 보류 중인 호출의 모든 세부 정보를 캡슐화합니다.

다음 방법을 사용하여 **Invocation** 오브젝트를 빌드할 수 있습니다.

```
buildGet
buildPost
buildDelete
buildPut
build
```

예를 들어 **GET Invocation** 오브젝트를 생성하고 나중에 호출하려면 다음과 같은 코드를 사용할 수 있습니다.

```
import javax.ws.rs.client.Invocation;
import javax.ws.rs.core.Response;
...
Invocation getBookInfo = client.target("http://example.org/bookstore/books/123")
                .request("application/xml").buildGet();
...
// Later on, in some other part of the application:
Response = getBookInfo.invoke();
```

49.3.12. 비동기 호출

JAX-RS 2.0 클라이언트 **API**는 클라이언트 측에서 비동기 호출을 지원합니다. 비동기 호출을 만들려면 **request()** 다음 메서드 체인에서 **async()** 메서드를 호출합니다. 예를 들면 다음과 같습니다.

```
Future<Response> res = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get();
```

비동기 호출을 수행할 때 반환된 값은 **java.util.concurrent.Future** 개체입니다. 비동기 호출에 대한 자세한 내용은 **49.6절. "클라이언트의 비동기 처리"** 을 참조하십시오.

49.4. 요청 및 응답 구문 분석

49.4.1. 개요

HTTP 호출을 수행하는 중요한 측면은 클라이언트가 발신 요청 메시지와 들어오는 응답을 구문 분석할 수 있어야 한다는 것입니다. **JAX-RS 2.0**에서 키 개념은 **Entity** 클래스이며, 미디어 유형으로 태그된 원시 메시지를 나타냅니다. 원시 메시지를 구문 분석하려면 미디어 유형을 특정 **Java** 유형으로 변환하거나 미디어 유형을 변환하는 기능이 있는 여러 엔터티 공급자 를 등록할 수 있습니다.

즉, **JAX-RS 2.0**의 컨텍스트에서 **Entity** 는 원시 메시지의 표현이며 엔터티 공급자는 원시 메시지를 구문 분석하는 기능을 제공하는 플러그인입니다(**media type** 기반).

49.4.2. 엔터티

Entity 는 메타데이터(미디어 유형, 언어 및 인코딩)에 의해 보강된 메시지 본문입니다. **Entity** 인스턴스는 메시지를 원시 형식으로 유지하며 특정 미디어 유형과 연결됩니다. **Entity** 개체의 콘텐츠를 **Java** 개체로 변환하려면 지정된 미디어 유형을 필수 **Java** 유형에 매핑할 수 있는 엔터티 공급자 가 필요합니다.

49.4.3. 변형

javax.ws.rs.core.Variant 오브젝트는 다음과 같이 엔터티 와 연결된 메타데이터를 캡슐화합니다.

- 미디어 유형
- 언어,
- 인코딩.

효과적으로는 엔터티 를 **HTTP** 메시지 콘텐츠로 구성하여 **Variant** 메타데이터로 보강할 수 있습니다.

49.4.4. 엔터티 공급자

엔터티 공급자는 미디어 유형과 **Java** 유형 간에 매핑 기능을 제공하는 클래스입니다. 엔터티 공급자를 특정 미디어 유형(또는 여러 미디어 유형)의 메시지를 구문 분석하는 기능을 제공하는 클래스로 생각할 수 있습니다. 엔터티 제공자의 두 가지 종류가 있습니다:

MessageBodyReader

미디어 유형에서 **Java** 유형으로 매핑할 수 있는 기능을 제공합니다.

MessageBodyWriter

Java 유형에서 미디어 유형으로 매핑할 수 있는 기능을 제공합니다.

49.4.5. 표준 엔터티 공급자

다음 **Java** 및 미디어 유형 조합에 대한 엔터티 공급자는 표준으로 제공됩니다.

byte[]

모든 미디어 유형(*/*).

java.lang.String

모든 미디어 유형(*/*).

java.io.InputStream

모든 미디어 유형(*/*).

java.io.Reader

모든 미디어 유형(*/*).

java.io.File

모든 미디어 유형(*/*).

javax.activation.DataSource

모든 미디어 유형(*/*).

javax.xml.transform.Source

XML 유형(text/xml,application/xml, 그리고 application/xml 형식)은 application/*+xml형식으로 되어 있습니다.

javax.xml.bind.JAXBElement 및 애플리케이션에서 제공하는 **JAXB** 클래스

XML 유형(text/xml,application/xml, 그리고 application/xml 형식)은 application/*+xml형식으로 되어 있습니다.

MultivaluedMap<String,String>

폼 콘텐츠(**application/x-www-form-urlencoded**).

StreamingOutput

모든 미디어 유형(***/***), **MessageBodyWriter** 만 해당됩니다.

java.lang.Boolean, java.lang.Character, java.lang.Number

text/plain 에만 해당한다. **boxing/unboxing** 변환을 통해 지원되는 해당하는 기본 형식입니다.

49.4.6. 응답 오브젝트

기본 반환 유형은 형식화되지 않은 응답을 나타내는 **javax.ws.rs.core.Response** 유형입니다. **Response** 오브젝트는 메시지 본문, **HTTP** 상태, **HTTP** 헤더, 미디어 유형 등을 포함하여 완전한 **HTTP** 응답에 대한 액세스를 제공합니다.

49.4.7. 응답 상태 액세스

getStatus 메서드(**HTTP** 상태 코드 반환)를 통해 응답 상태에 액세스할 수 있습니다.

```
int status = resp.getStatus();
```

또는 설명 문자열을 제공하는 **getStatusInfo** 메서드를 사용합니다.

```
String statusReason = resp.getStatusInfo().getReasonPhrase();
```

49.4.8. 반환된 헤더에 액세스

다음 방법 중 하나를 사용하여 **HTTP** 헤더에 액세스할 수 있습니다.

```
MultivaluedMap<String,Object>
getHeaders()

MultivaluedMap<String,String>
getStringHeaders()

String
getHeaderString(String name)
```

예를 들어 응답 의 날짜 헤더가 있음을 알고 있는 경우 다음과 같이 액세스할 수 있습니다.

```
String dateAsString = resp.getHeaderString("Date");
```

49.4.9. 반환된 쿠키 액세스

다음과 같이 **getCookies** 방법을 사용하여 응답에 설정된 새 쿠키에 액세스할 수 있습니다.

```
import javax.ws.rs.core.NewCookie;
...
java.util.Map<String,NewCookie> cookieMap = resp.getCookies();
java.util.Collection<NewCookie> cookieCollection = cookieMap.values();
```

49.4.10. 반환된 메시지 콘텐츠에 액세스

Response 개체에서 **readEntity** 메서드 중 하나를 호출하여 반환된 메시지 콘텐츠에 액세스할 수 있습니다. **readEntity** 메서드는 사용 가능한 엔터티 공급자를 자동으로 호출하여 메시지를 요청된 유형(**ReadEntity**의 첫 번째 인수로 지정)으로 변환합니다. 예를 들어, 메시지 콘텐츠에 **String** 유형으로 액세스하려면 다음을 수행합니다.

```
String messageBody = resp.readEntity(String.class);
```

49.4.11. 컬렉션 반환 값

Java 일반 유형(예: **List** 또는 **Collection** 유형)으로 반환된 메시지에 액세스해야 하는 경우 **javax.ws.rs.core.GenericType<T>** 구조를 사용하여 요청 메시지 유형 을 지정할 수 있습니다. 예를 들면 다음과 같습니다.

```
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import javax.ws.rs.core.GenericType;
import java.util.List;
...
GenericType<List<String>> stringListType = new GenericType<List<String>>() {};

Client client = ClientBuilder.newClient();
List<String> bookNames = client.target("http://example.org/bookstore/booknames")
                .request("text/plain")
                .get(stringListType);
```

49.5. 클라이언트 끝점 구성

49.5.1. 개요

기능 및 공급자를 등록하고 구성하여 기본 **javax.ws.rs.client.Client** 오브젝트의 기능을 보강할 수 있습니다.

49.5.2. 예제

다음 예제에서는 로깅 기능, 사용자 지정 엔터티 공급자 및 **prettyLogging** 속성을 **true**로 설정하도록 구성된 클라이언트를 보여줍니다.**The following example shows a client configured to have a logging feature, a custom entity provider, and to set the prettyLogging property to true:**

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import org.apache.cxf.feature.LoggingFeature;
...
Client client = ClientBuilder.newClient();
client.register(LoggingFeature.class)
      .register(MyCustomEntityProvider.class)
      .property("LoggingFeature.prettyLogging","true");
```

49.5.3. 오브젝트 등록을 위한 구성 가능한 API

Client 클래스는 **register** 메서드의 여러 변형을 제공하는 오브젝트 등록에 대한 **Configurable API**를 지원합니다. 대부분의 경우 다음 예와 같이 클래스 또는 개체 인스턴스를 등록합니다.

```
client.register(LoggingFeature.class)
client.register(new LoggingFeature())
```

레지스터 변형에 대한 자세한 내용은 구성 가능 에 대한 참조 설명서를 참조하십시오.

49.5.4. 고객에게 무엇을 설정할 수 있습니까?

클라이언트 끝점의 다음 측면을 구성할 수 있습니다.

- 기능
- 공급자
- 속성

- 필터
- 인터셉터

49.5.5. 기능

javax.ws.rs.core.Feature 는 실제로 JAX-RS 클라이언트에 추가 기능 또는 기능을 추가하는 플러그인입니다. 종종 기능은 필요한 기능을 제공하기 위해 하나 이상의 인터셉터를 설치합니다.

49.5.6. 공급자

공급자는 매핑 기능을 제공하는 특정 유형의 클라이언트 플러그인입니다. JAX-RS 2.0 사양은 다음과 같은 종류의 공급자를 정의합니다.

엔터티 공급자

엔터티 공급자 는 Java 유형에서 특정 미디어 유형 간 매핑 기능을 제공합니다. 자세한 내용은 49.4절. "요청 및 응답 구문 분석" 에서 참조하십시오.

예외 매핑 공급자

예외 매핑 공급자 는 확인된 런타임 예외를 응답 인스턴스에 매핑합니다.

컨텍스트 공급자

컨텍스트 공급자는 서버 측에서 리소스 클래스 및 기타 서비스 공급자에 컨텍스트를 제공하는 데 사용됩니다.

49.5.7. 필터

JAX-RS 2.0 필터는 메시지 처리 파이프라인의 다양한 지점(extension 포인트)에서 URI, 헤더 및 기타 컨텍스트 데이터에 액세스할 수 있는 플러그인입니다. 자세한 내용은 61장. JAX-RS 2.0 필터 및 인터셉터의 내용을 참조하십시오.

49.5.8. 인터셉터

JAX-RS 2.0 인터셉터는 요청 또는 응답의 메시지 본문에 읽거나 쓰는 대로 액세스할 수 있는 플러그인입니다. 자세한 내용은 61장. JAX-RS 2.0 필터 및 인터셉터의 내용을 참조하십시오.

49.5.9. 속성

클라이언트에 속성을 하나 이상 설정하여 등록된 기능 또는 등록된 공급자의 구성을 사용자 지정할 수 있습니다.**By setting one or more properties on the client, you can customize the configuration of a registered feature or a registered provider.**

49.5.10. 기타 구성 가능한 유형

javax.ws.rs.client.Client(및 **javax.ws.rs.client.Client Builder**) 오브젝트를 구성할 뿐만 아니라 **WebTarget** 오브젝트도 구성할 수 있습니다. **WebTarget** 개체의 구성을 변경하면 새 **WebTarget** 구성을 제공하기 위해 기본 클라이언트 구성이 깊이 복사됩니다. 따라서 원래 **Client** 개체의 구성을 변경하지 않고 **WebTarget** 오브젝트의 구성을 변경할 수 있습니다.

49.6. 클라이언트의 비동기 처리

49.6.1. 개요

JAX-RS 2.0은 클라이언트 측에서 호출의 비동기 처리를 지원합니다. **java.util.concurrent.Future<V >** 반환 값을 사용하거나 호출 콜백을 등록하여 비동기 처리 스타일이 지원됩니다.

49.6.2. future<V> 반환 값을 사용한 비동기 호출

future< V > 접근 방식을 비동기 처리에 사용하면 다음과 같이 클라이언트 요청을 비동기적으로 호출할 수 있습니다.**Using the future<V> approach to asynchronous processing, you can invoke a client request asynchronously, as follows:**

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get();
...
// At a later time, check (and wait) for the response:
Response resp = futureResp.get();
```

입력한 답변에 유사한 방법을 사용할 수 있습니다. 예를 들어, **type**에 대한 응답을 얻으려면 **BookInfo:**

```
Client client = ClientBuilder.newClient();
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(BookInfo.class);
...
// At a later time, check (and wait) for the response:
BookInfo resp = futureResp.get();
```

49.6.3. 호출 콜백을 사용하는 비동기 호출

future<V> 개체를 사용하여 반환 값에 액세스하는 대신 호출 콜백(**javax.ws.rs.client.InvocationCallback<RESPONSE>**)을 사용하여 호출을 정의할 수 있습니다.

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;
...
Client client = ClientBuilder.newClient();
Future<Response> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(
    new InvocationCallback<Response>() {
        @Override
        public void completed(final Response resp) {
            // Do something when invocation is complete
            ...
        }

        @Override
        public void failed(final Throwable throwable) {
            throwable.printStackTrace();
        }
    });
...
```

입력한 답변에 유사한 방법을 사용할 수 있습니다.

```
// Java
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Client;
import java.util.concurrent.Future;
import javax.ws.rs.core.Response;
import javax.ws.rs.client.InvocationCallback;
...
Client client = ClientBuilder.newClient();
```

```
Future<BookInfo> futureResp = client.target("http://example.org/bookstore/books/123")
                .request("application/xml")
                .async()
                .get(
    new InvocationCallback<BookInfo>() {
        @Override
        public void completed(final BookInfo resp) {
            // Do something when invocation is complete
            ...
        }

        @Override
        public void failed(final Throwable throwable) {
            throwable.printStackTrace();
        }
    });
...
```

50장. 예외 처리

초록

가능한 경우 리소스 메서드에서 catch한 예외로 인해 요청 소비자에게 유용한 오류가 반환되어야 합니다. JAX-RS 리소스 메서드는 WebApplicationException 예외를 throw할 수 있습니다. 적절한 응답에 예외를 매핑하도록 ExceptionMapper<E> 구현을 제공할 수도 있습니다.

50.1. JAX-RS 예외 클래스 개요

50.1.1. 개요

JAX-RS 1.x에서 사용 가능한 유일한 예외 클래스는 WebApplicationException 입니다. 그러나 JAX-WS 2.0부터 많은 추가 JAX-RS 예외 클래스가 정의되어 있습니다.

50.1.2. JAX-RS 런타임 수준 예외

다음 예외는 JAX-RS 런타임에서만 throw되어야 합니다(즉, 애플리케이션 수준 코드에서 이러한 예외를 throw해서는 안 됩니다).

ProcessingException

(JAX-RS 2.0만 해당) 요청 처리 중 또는 JAX-RS 런타임의 응답 처리 중에 javax.ws.rs.ProcessingException 을 throw할 수 있습니다. 예를 들어 필터 체인 또는 인터셉터 체인 처리의 오류로 인해 이 오류가 발생할 수 있습니다.

ResponseProcessingException

(JAX-RS 2.0만 해당) javax.ws.rs.client.ResponseProcessingException 은 클라이언트 측 의 JAX-RS 런타임에서 오류가 발생할 때 발생할 수 있는 ProcessingException 의 하위 클래스입니다.

50.1.3. JAX-RS 애플리케이션 수준 예외

다음 예외는 애플리케이션 수준 코드에서 throw(및 catch)되도록 설계되었습니다.The following exceptions are intended to be thrown (and caught) in your application level code:

WebApplicationException

javax.ws.rs.WebApplicationException 은 서버 측의 애플리케이션 코드에서 throw될 수 있는 일반적인 애플리케이션 수준 JAX-RS 예외입니다. 이 예외 유형은 HTTP 상태 코드, 오류 메시지 및

(선택적으로) 응답 메시지를 캡슐화할 수 있습니다. 자세한 내용은 50.2절. "WebApplicationException 예외를 보고서에 사용"의 내용을 참조하십시오.

ClientErrorException

(JAX-RS 2.0만 해당) javax.ws.rs.ClientErrorException 클래스는 WebApplicationException에서 상속하고 HTTP 4xx 상태 코드를 캡슐화하는 데 사용됩니다.

ServerErrorException

(JAX-RS 2.0만 해당) javax.ws.rs.ServerErrorException 클래스는 WebApplicationException에서 상속하고 HTTP 5xx 상태 코드를 캡슐화하는 데 사용됩니다.

RedirectionException

(JAX-RS 2.0만 해당) javax.ws.rs.RedirectionException 클래스는 WebApplicationException에서 상속하고 HTTP 3xx 상태 코드를 캡슐화하는 데 사용됩니다.

50.2. WEBAPPLICATIONEXCEPTION 예외를 보고서에 사용

> errors
> indexterm:[WebApplicationException]

50.2.1. 개요

JAX-RS API는 리소스 메서드가 RESTful 클라이언트가 사용할 수 있는 적절한 예외를 생성하는 쉬운 방법을 제공하기 위해 WebApplicationException 런타임 예외를 도입했습니다. WebApplicationException 예외는 요청의 originator로 반환할 엔티티 본문을 정의하는 Response 개체를 포함할 수 있습니다. 엔티티 본문이 제공되지 않는 경우 클라이언트에 반환되는 HTTP 상태 코드를 지정하는 메커니즘도 제공합니다.

50.2.2. 간단한 예외 생성

WebApplicationException 예외를 만드는 가장 쉬운 방법은 WebApplicationException 예외에서 원래 예외를 래핑하는 인수 생성자 또는 생성자를 사용하는 것입니다. 두 생성자 모두 빈 응답이 있는 WebApplicationException을 만듭니다.

이러한 생성자 중 하나에 의해 생성된 예외가 발생하면 런타임은 비어 있는 엔티티 본문과 상태 코드 500 Server Error 와 함께 응답을 반환합니다.

50.2.3. 클라이언트에 반환되는 상태 코드 설정

500 이외의 오류 코드를 반환하려는 경우 상태를 지정할 수 있는 4개의 WebApplicationException

생성자 중 하나를 사용할 수 있습니다. 예 50.1. “상태 코드를 사용하여 WebApplicationException 생성” 에 표시된 두 개의 생성자 중 두 개는 반환 상태를 정수로 사용합니다.

> 예 50.1. 상태 코드를 사용하여 WebApplicationException 생성
>
> WebApplicationException상태WebApplicationExceptionjava.lang.Throwablecauseintstatus

예 50.2. “상태 코드를 사용하여 WebApplicationException 생성” 에 표시된 나머지 두 개는 response .Status 인스턴스로 응답 상태를 사용합니다.

> 예 50.2. 상태 코드를 사용하여 WebApplicationException 생성
>
> WebApplicationExceptionjavax.ws.rs.core.Response.Status상태WebApplicationExceptionjava.lang.Throwablecausejavax.ws.rs.core.Response.Status상태

이러한 생성자 중 하나에서 만든 예외가 throw되면 런타임은 비어 있는 엔터티 본문 및 지정된 상태 코드와 함께 응답을 반환합니다.When an exception created by either of these constructors is thrown, the runtime returns a response with an empty entity body and the specified status code.

50.2.4. 엔터티 본문 제공

예외와 함께 메시지를 보내려면 Response 개체를 사용하는 WebApplicationException 생성자 중 하나를 사용할 수 있습니다. 런타임은 Response 오브젝트를 사용하여 클라이언트에 전송된 응답을 생성합니다. 응답에 저장된 엔터티는 메시지의 엔터티 본문에 매핑되고 응답의 상태 필드는 메시지의 HTTP 상태에 매핑됩니다.

예 50.3. “예외를 사용하여 메시지 전송” 예외 이유를 포함하는 클라이언트에 텍스트 메시지를 반환하는 코드를 표시하고 HTTP 메시지 상태를 409 Conflict 로 설정합니다.

> 예 50.3. 예외를 사용하여 메시지 전송
>
> ```
> import javax.ws.rs.core.Response;
> import javax.ws.rs.WebApplicationException;
> import org.apache.cxf.jaxrs.impl.ResponseBuilderImpl;
>
> ...
> ResponseBuilderImpl builder = new ResponseBuilderImpl();
> builder.status(Response.Status.CONFLICT);
> ```

```
builder.entity("The requested resource is conflicted.");
Response response = builder.build();
throw WebApplicationException(response);
```

50.2.5. 일반 예외 확장

WebApplicationException 예외를 확장할 수 있습니다. 이렇게 하면 사용자 정의 예외를 만들고 일부 보일러 플레이트 코드를 제거할 수 있습니다.

예 50.4. "WebApplicationException 확장" *예 50.3. "예외를 사용하여 메시지 전송"* 에서 코드와 유사한 응답을 생성하는 새로운 예외를 보여줍니다.

예 50.4. WebApplicationException 확장

```
public class ConflicteddException extends WebApplicationException
{
  public ConflictedException(String message)
  {
    ResponseBuilderImpl builder = new ResponseBuilderImpl();
    builder.status(Response.Status.CONFLICT);
    builder.entity(message);
    super(builder.build());
     }
}

...
throw ConflictedException("The requested resource is conflicted.");
```

50.3. JAX-RS 2.0 예외 유형

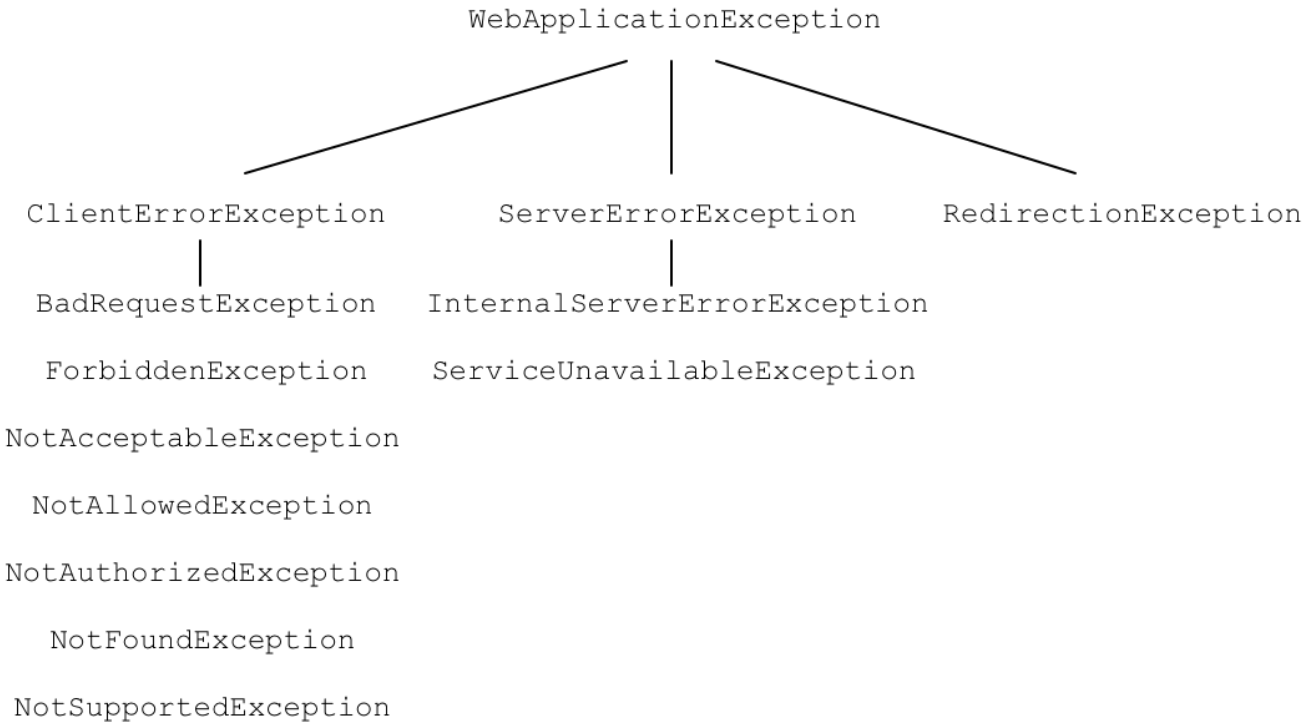
50.3.1. 개요

JAX-RS 2.0에는 애플리케이션 코드에서 **throw**(및 **catch**)할 수 있는 여러 특정 **HTTP** 예외 유형이 도입되었습니다(기존 **WebApplicationException** 예외 유형 포함). 이러한 예외 유형을 사용하여 **HTTP** 클라이언트 오류(**HTTP 4xx** 상태 코드) 또는 **HTTP** 서버 오류(**HTTP 5xx** 상태 코드)에 대해 표준 **HTTP** 상태 코드를 래핑할 수 있습니다.

50.3.2. 예외 계층 구조

그림 50.1. "JAX-RS 2.0 애플리케이션 예외 계층 구조" **JAX-RS 2.0**에서 지원되는 애플리케이션 수준 예외의 계층 구조를 보여줍니다.

그림 50.1. JAX-RS 2.0 애플리케이션 예외 계층 구조



50.3.3. WebApplicationException 클래스

javax.ws.rs.WebApplicationException 예외 클래스(Tht-RS 1.x 이후 사용 가능)는 JAX-RS 2.0 예외 계층의 기반에 있으며 50.2절. "WebApplicationException 예외를 보고서에 사용" 에 자세히 설명되어 있습니다.

50.3.4. ClientErrorException 클래스

javax.ws.rs.ClientErrorException 예외 클래스는 HTTP 클라이언트 오류(HTTP 4xx 상태 코드)를 캡슐화하는 데 사용됩니다. 애플리케이션 코드에서 이 예외 또는 해당 하위 클래스 중 하나를 throw할 수 있습니다.

50.3.5. ServerErrorException 클래스

javax.ws.rs.ServerErrorException 예외 클래스는 HTTP 서버 오류(HTTP 5xx 상태 코드)를 캡슐화하는 데 사용됩니다. 애플리케이션 코드에서 이 예외 또는 해당 하위 클래스 중 하나를 throw할 수 있습니다.

50.3.6. RedirectionException 클래스

javax.ws.rs.RedirectionException 예외 클래스는 HTTP 요청 리디렉션(HTTP 3xx 상태 코드)을 캡슐화하는 데 사용됩니다. 이 클래스의 생성자는 리디렉션 위치를 지정하는 URI 인수를 사용합니다. 리디렉

션 URI는 getLocation() 메서드를 통해 액세스할 수 있습니다. 일반적으로 HTTP 리디렉션은 클라이언트 측에서 투명합니다.

50.3.7. 클라이언트 예외 하위 클래스

JAX-RS 2.0 애플리케이션에서 다음 HTTP 클라이언트 예외(HTTP 4xx 상태 코드)를 발생시킬 수 있습니다.

BadRequestException

400 Bad Request HTTP 오류 상태를 캡슐화합니다.

ForbiddenException

403 Forbidden HTTP 오류 상태를 캡슐화합니다.

NotAcceptableException

406 Acceptable HTTP 오류 상태를 캡슐화합니다.

NotAllowedException

405 Method Not Allowed HTTP 오류 상태를 캡슐화합니다.

NotAuthorizedException

401 Unauthorized HTTP 오류 상태를 캡슐화합니다. 이 예외는 다음 경우 중 하나에서 발생할 수 있습니다.This exception could be raised in either of the following cases:

- 클라이언트가 필요한 자격 증명을 전송하지 않았거나 (HTTP Authorization 헤더에 있음)
- 클라이언트에서 자격 증명을 제공했지만 자격 증명이 유효하지 않았습니다.

NotFoundException

404 Not Found HTTP 오류 상태를 캡슐화합니다.

NotSupportedException

415 지원되지 않는 미디어 유형 HTTP 오류 상태를 캡슐화합니다.

50.3.8. 서버 예외 하위 클래스

JAX-RS 2.0 애플리케이션에서 다음 HTTP 서버 예외(HTTP 5xx 상태 코드)를 발생시킬 수 있습니다.

InternalServerErrorException

500 Internal Server Error HTTP 오류 상태를 캡슐화합니다.

ServiceUnavailableException

503 Service Unavailable HTTP 오류 상태를 캡슐화합니다.

50.4. 응답과 예외 매핑

50.4.1. 개요

WebApplicationException 예외를 throw하는 것이 비현실적이거나 불가능한 경우가 있습니다. 예를 들어 가능한 모든 예외를 catch하고 이를 위해 WebApplicationException을 만들 필요가 없습니다. 애플리케이션 코드에서 더 쉽게 작업할 수 있도록 하는 사용자 지정 예외를 사용할 수도 있습니다.

이러한 경우를 처리하기 위해 JAX-RS API를 사용하면 클라이언트로 보낼 Response 개체를 생성하는 사용자 지정 예외 공급자를 구현할 수 있습니다. 사용자 지정 예외 공급자는 ExceptionMapper<E> 인터페이스를 구현하여 생성됩니다. Apache CXF 런타임에 등록하면 E 유형의 예외가 throw될 때마다 사용자 지정 공급자가 사용됩니다.

50.4.2. 예외 매퍼를 선택하는 방법

예외 매퍼는 다음 두 가지 경우에 사용됩니다.

- 예외 또는 해당 하위 클래스 중 하나가 throw되면 런타임은 적절한 예외 매퍼를 확인합니다. 예외 매퍼는 throw된 특정 예외를 처리하면 선택됩니다. throw된 특정 예외에 대한 예외 매퍼가 없는 경우 예외의 가장 가까운 수퍼 클래스에 대한 예외 매퍼가 선택됩니다.**If there is not an exception mapper for the specific exception that was thrown, the exception mapper for the nearest superclass of the exception is selected.**

- 기본적으로 WebApplicationException은 기본 매퍼, WebApplicationExceptionMapper로 처리됩니다. 추가 사용자 지정 매퍼가 등록되어 있어도 WebApplicationException 예외(예: 사용자 지정 RuntimeException 매퍼)를 처리할 수 있는 경우 사용자 지정 매퍼가 사용되지 않고 WebApplicationExceptionMapper 가 대신 사용됩니다.

그러나 **Message** 개체에서 **default.wae.mapper.least.specific** 속성을 **true** 로 설정하여 이 동작을 변경할 수 있습니다. 이 옵션이 활성화되면 기본 **WebApplicationExceptionMapper** 가 가장 낮은 우선 순위로 리비스되어 사용자 지정 예외를 사용하여 **WebApplicationException** 예외를 처리할 수 있습니다. 예를 들어 이 옵션이 활성화된 경우 사용자 지정 **RuntimeException** 매퍼를 등록하여 **WebApplicationException** 예외를 **catch**할 수 있습니다.
"WebApplicationException에 대한 예외 매퍼 등록" 을 참조하십시오.

예외에 대한 예외 매퍼를 찾을 수 없는 경우 예외를 **ServletException** 예외로 래핑되고 컨테이너 런타임에 전달됩니다. 그러면 컨테이너 런타임에서 예외를 처리하는 방법을 결정합니다.

50.4.3. 예외 매퍼 구현

예외 매퍼는 **javax.ws.rs.ext.ExceptionMapper<E>** 인터페이스를 구현하여 생성됩니다. 예 50.5. "예외 매퍼 인터페이스" 에 표시된 대로 인터페이스에는 원래 예외를 매개 변수로 사용하고 **Response** 개체를 반환하는 단일 메서드인 **toResponse()** 가 있습니다.

예 50.5. 예외 매퍼 인터페이스

```
public interface ExceptionMapper<E extends java.lang.Throwable>
{
  public Response toResponse(E exception);
}
```

예외 매퍼에서 생성한 **Response** 오브젝트는 다른 **Response** 오브젝트와 마찬가지로 런타임에서 처리합니다. 소비자에 대한 결과 응답에는 **Response** 오브젝트에 캡슐화된 상태, 헤더 및 엔티티 본문이 포함됩니다.

예외 매퍼 구현은 런타임을 통해 공급자로 간주됩니다. 따라서 **@Provider** 주석으로 장식되어야 합니다.

예외 매퍼가 **Response** 개체를 빌드하는 동안 예외가 발생하면 런타임에서 **500 Server Error** 상태의 응답을 소비자에게 보냅니다.

예 50.6. "응답에 예외 매핑" **Spring AccessDeniedException** 예외를 가로채는 예외 매퍼를 표시하고 **403 Forbidden** 상태 및 빈 엔터티 본문을 사용하여 응답을 생성합니다.

예 50.6. 응답에 예외 매핑

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

```
import org.springframework.security.AccessDeniedException;

@Provider
public class SecurityExceptionMapper implements ExceptionMapper<AccessDeniedException>
{

  public Response toResponse(AccessDeniedException exception)
  {
    return Response.status(Response.Status.FORBIDDEN).build();
  }

}
```

런타임은 모든 **AccessDeniedException** 예외를 포착하고 엔티티 본문 및 상태 **403** 이 없는 **Response** 개체를 만듭니다. 그런 다음 런타임은 **Response** 오브젝트를 일반 응답과 마찬가지로 처리합니다. 그 결과 소비자가 상태가 **403** 인 **HTTP** 응답을 수신합니다.

50.4.4. 예외 매퍼 등록

JAX-RS 애플리케이션에서 예외 매퍼를 사용하려면 먼저 예외 매퍼를 런타임에 등록해야 합니다. 예외 매퍼는 애플리케이션의 구성 파일에 **jaxrs:providers** 요소를 사용하여 런타임에 등록됩니다.

jaxrs:providers 요소는 **jaxrs:server** 요소의 하위이며 빈 요소 목록 을 포함합니다. 각 빈 요소는 하나의 예외 매퍼를 정의합니다.

예 **50.7.** "런타임을 사용하여 예외 매퍼 등록" 사용자 지정 예외 매퍼, **SecurityExceptionMapper** 를 사용하도록 구성된 **JAX-RS** 서버를 보여줍니다.

예 50.7. 런타임을 사용하여 예외 매퍼 등록

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

50.4.5. WebApplicationException에 대한 예외 매퍼 등록

WebApplicationException 예외에 대한 예외 매퍼는 기본 **WebApplicationException Mapper** 에서

*자동으로 처리되기 때문에 특별한 경우입니다. 일반적으로 **WebApplicationException** 을 처리 할 것으로 예상되는 사용자 정의 매퍼를 등록하더라도 기본 **WebApplicationExceptionMapper** 로 계속 처리됩니다. 이 기본 동작을 변경하려면 **default.wae.mapper.least.specific** 속성을 **true** 로 설정해야 합니다.*

*예를 들어 다음 **XML** 코드는 **JAX-RS** 끝점에서 **default.wae.mapper.least.specific** 속성을 활성화하는 방법을 보여줍니다.*

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="securityException" class="com.bar.providers.SecurityExceptionMapper"/>
    </jaxrs:providers>
    <jaxrs:properties>
      <entry key="default.wae.mapper.least.specific" value="true"/>
    </jaxrs:properties>
  </jaxrs:server>
</beans>
```

*다음 예와 같이 인터셉터에서 **default.wae.mapper.least.specific** 속성을 설정할 수도 있습니다.*

```
// Java
public void handleMessage(Message message)
{
    m.put("default.wae.mapper.least.specific", true);
    ...
```

51장. 엔터티 지원

초록

Apache CXF 런타임은 MIME 유형과 Java 개체 중에서 상자에서 제한된 수의 매핑을 지원합니다. 개발자는 사용자 정의 독자와 작성자를 구현하여 매핑을 확장할 수 있습니다. 사용자 지정 독자 및 작성자는 시작 시 런타임에 등록됩니다.

51.1. 개요

런타임은 JAX-RS MessageBodyReader 및 MessageBodyWriter 구현을 사용하여 HTTP 메시지와 Java 메시지 간에 데이터를 직렬화하고 역직렬화합니다. 독자와 작성자는 처리할 수 있는 MIME 유형을 제한할 수 있습니다.

런타임에서는 여러 공통 매핑에 대한 독자 및 작성자를 제공합니다. 애플리케이션에 더 많은 고급 매핑이 필요한 경우 개발자는 MessageBodyReader 인터페이스 및/또는 MessageBodyWriter 인터페이스의 사용자 지정 구현을 제공할 수 있습니다. 사용자 지정 독자 및 작성자는 애플리케이션이 시작될 때 런타임으로 등록됩니다.

51.2. 기본적으로 지원되는 유형

표 51.1. “기본적으로 지원되는 엔터티 매핑” Apache CXF에서 제공하는 엔터티 매핑을 상자에서 나열합니다.

표 51.1. 기본적으로 지원되는 엔터티 매핑

Java Type	MIME 유형
기본 유형	**text/plain**
java.lang.Number	**text/plain**
byte[]	*/*
java.lang.String	*/*
java.io.InputStream	*/*
java.io.Reader	*/*
java.io.File	*/*

Java Type	MIME 유형
javax.activation.DataSource	***/***
javax.xml.transform.Source	**text/xml,application/xml,application/*+xml**
javax.xml.bind.JAXBElement	**text/xml,application/xml,application/*+xml**
JAXB 주석이 있는 오브젝트	**text/xml,application/xml,application/*+xml**
javax.ws.rs.core.MultivaluedMap<String, String>	**application/x-www-form-urlencoded** [a]
javax.ws.rs.core.StreamingOutput	***/*** [b]

[a] 이 매핑은 HTML 양식 데이터를 처리하는 데 사용됩니다.

[b] 이 매핑은 소비자에게 데이터를 반환하는 경우에만 지원됩니다.

51.3. 사용자 정의 리더

*사용자 지정 엔터티 리더는 들어오는 **HTTP** 요청을 서비스의 구현이 조작할 수 있는 **Java** 유형으로 매핑해야 합니다. **javax.ws.rs.ext.MessageBodyReader** 인터페이스를 구현합니다.*

예 51.1. "메시지 리더 인터페이스" 에 표시된 인터페이스에는 구현이 필요한 두 가지 방법이 있습니다.

예 51.1. 메시지 리더 인터페이스

```
package javax.ws.rs.ext;

public interface MessageBodyReader<T>
{
  public boolean isReadable(java.lang.Class<?> type,
                            java.lang.reflect.Type genericType,
                            java.lang.annotation.Annotation[] annotations,
                            javax.ws.rs.core.MediaType mediaType);

  public T readFrom(java.lang.Class<T> type,
                    java.lang.reflect.Type genericType,
                    java.lang.annotation.Annotation[] annotations,
                    javax.ws.rs.core.MediaType mediaType,
                    javax.ws.rs.core.MultivaluedMap<String, String> httpHeaders,
                    java.io.InputStream entityStream)
  throws java.io.IOException, WebApplicationException;
}
```

isReadable()

isReadable() 메서드는 reader가 데이터 스트림을 읽고 적절한 유형의 엔터티 표현을 생성할 수 있는지 여부를 결정합니다. 독자가 적절한 유형의 엔터티를 만들 수 있는 경우 메서드는 true를 반환합니다.If the reader can create the proper type of entity the method returns true.

표 51.2. "판독기가 엔터티를 생성할 수 있는지 확인하는 데 사용되는 매개변수" isReadable() 메서드의 매개 변수에 대해 설명합니다.

표 51.2. 판독기가 엔터티를 생성할 수 있는지 확인하는 데 사용되는 매개변수

매개변수	유형	설명
type	**Class<T>**	엔터티를 저장하는 데 사용되는 개체의 실제 Java 클래스를 지정합니다.
genericType	유형	엔터티를 저장하는 데 사용되는 개체의 Java 유형을 지정합니다. 예를 들어 메시지 본문을 메서드 매개 변수로 변환하려는 경우 값은 **Method.getGenericParameterTypes()** 메서드에서 반환된 메서드 매개 변수의 유형입니다.
annotations	annotation[]	엔터티를 저장하도록 생성된 오브젝트 선언에 대한 주석 목록을 지정합니다. 예를 들어 메시지 본문을 메서드 매개 변수로 변환하는 경우 **Method.getParameterAnnotations()** 메서드에서 반환된 해당 매개 변수의 주석이 됩니다.
mediaType	**MediatType**	HTTP 엔터티의 MIME 형식을 지정합니다.

readFrom()

readFrom() 메서드는 HTTP 엔터티를 읽고 원하는 Java 오브젝트에 포함합니다. 읽기가 성공하면 메서드가 엔터티를 포함하는 생성된 Java 개체를 반환합니다. 입력 스트림을 읽을 때 오류가 발생하면 메서드가 IOException 예외를 throw해야 합니다.If an error occurs when reading the input stream the method should throw an IOException exception. HTTP 오류 응답이 필요한 오류가 발생하면 HTTP 응답이 있는 WebApplicationException을 throw해야 합니다.

표 51.3. "엔터티를 읽는 데 사용되는 매개변수" readFrom() 메서드의 매개 변수에 대해 설명합니다.

표 51.3. 엔터티를 읽는 데 사용되는 매개변수

매개변수	유형	설명
type	**Class<T>**	엔터티를 저장하는 데 사용되는 개체의 실제 Java 클래스를 지정합니다.
genericType	유형	엔터티를 저장하는 데 사용되는 개체의 Java 유형을 지정합니다. 예를 들어 메시지 본문을 메서드 매개 변수로 변환하려는 경우 값은 **Method.getGenericParameterTypes()** 메서드에서 반환된 메서드 매개 변수의 유형입니다.
annotations	annotation[]	엔터티를 저장하도록 생성된 오브젝트 선언에 대한 주석 목록을 지정합니다. 예를 들어 메시지 본문을 메서드 매개 변수로 변환하는 경우 **Method.getParameterAnnotations()** 메서드에서 반환된 해당 매개 변수의 주석이 됩니다.
mediaType	**MediatType**	HTTP 엔터티의 MIME 형식을 지정합니다.
httpHeaders	MultivaluedMap<String, String>	엔터티와 연결된 HTTP 메시지 헤더를 지정합니다.
entityStream	**InputStream**	HTTP 엔터티를 포함하는 입력 스트림을 지정합니다.

중요

이 메서드는 입력 스트림을 닫지 않아야 합니다.

***MessageBodyReader** 구현을 엔터티 리더로 사용하려면 **javax.ws.rs.ext.Provider** 주석으로 장식해야 합니다. **@Provider** 주석은 제공된 구현에서 추가 기능을 제공하는 런타임을 경고합니다. 구현도 "독자 및 작성자 등록" 에 설명된 대로 런타임에 등록되어 있어야 합니다.*

*기본적으로 사용자 지정 엔터티 공급자는 모든 **MIME** 유형을 처리합니다. 사용자 지정 엔터티 리더가 **javax.ws.rs.Consumes** 주석을 사용하여 처리할 **MIME** 유형을 제한할 수 있습니다. **@Consumes** 주석*

은 사용자 정의 엔터티 공급자가 읽는 쉼표로 구분된 **MIME** 유형 목록을 지정합니다. 엔터티가 지정된 **MIME** 유형이 아닌 경우 엔터티 공급자가 가능한 리더로 선택되지 않습니다.

예 51.2. "XML 소스 엔터티 리더" 엔터티 리더는 **XML** 엔터티를 사용하여 소스 오브젝트에 저장합니다.

예 51.2. XML 소스 엔터티 리더

```
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Consumes;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyReader;
import javax.ws.rs.ext.Provider;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Consumes({"application/xml", "application/*+xml", "text/xml", "text/html" })
public class SourceProvider implements MessageBodyReader<Object>
{
  public boolean isReadable(Class<?> type,
                            Type genericType,
                            Annotation[] annotations,
                            MediaType mt)
  {
    return Source.class.isAssignableFrom(type) || XMLSource.class.isAssignableFrom(type);
  }

  public Object readFrom(Class<Object> source,
                         Type genericType,
                         Annotation[] annotations,
                         MediaType mediaType,
                         MultivaluedMap<String, String> httpHeaders,
                         InputStream is)
  throws IOException
  {
    if (DOMSource.class.isAssignableFrom(source))
    {
      Document doc = null;
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder;
      try
```

```
        {
          builder = factory.newDocumentBuilder();
          doc = builder.parse(is);
        }
        catch (Exception e)
        {
          IOException ioex = new IOException("Problem creating a Source object");
          ioex.setStackTrace(e.getStackTrace());
          throw ioex;
        }

        return new DOMSource(doc);
      }
      else if (StreamSource.class.isAssignableFrom(source) ||
    Source.class.isAssignableFrom(source))
      {
        return new StreamSource(is);
      }
      else if (XMLSource.class.isAssignableFrom(source))
      {
        return new XMLSource(is);
      }

      throw new IOException("Unrecognized source");
    }
  }
```

51.4. 사용자 정의 작성자

사용자 지정 엔티티 작성자는 **Java** 유형을 **HTTP** 엔티티에 매핑합니다. **javax.ws.rs.ext.MessageBodyWriter** 인터페이스를 구현합니다.

예 51.3. "메시지 작성기 인터페이스" 에 표시된 인터페이스에는 구현이 필요한 세 가지 방법이 있습니다.

예 51.3. 메시지 작성기 인터페이스

```
package javax.ws.rs.ext;

public interface MessageBodyWriter<T>
{
  public boolean isWriteable(java.lang.Class<?> type,
                             java.lang.reflect.Type genericType,
                             java.lang.annotation.Annotation[] annotations,
                             javax.ws.rs.core.MediaType mediaType);

  public long getSize(T t,
                      java.lang.Class<?> type,
                      java.lang.reflect.Type genericType,
```

```
                java.lang.annotation.Annotation[] annotations,
                javax.ws.rs.core.MediaType mediaType);

  public void writeTo(T t,
                java.lang.Class<?> type,
                java.lang.reflect.Type genericType,
                java.lang.annotation.Annotation[] annotations,
                javax.ws.rs.core.MediaType mediaType,
                javax.ws.rs.core.MultivaluedMap<String, Object> httpHeaders,
                java.io.OutputStream entityStream)
   throws java.io.IOException, WebApplicationException;
 }
```

isWriteable()

***isWriteable()** 메서드는 엔터티 작성자가 **Java** 유형을 적절한 엔터티 유형에 매핑할 수 있는지 여부를 결정합니다. 작성자가 매핑을 수행할 수 있는 경우 메서드는 **true** 를 반환합니다.*

*표 **51.4.** "엔터티를 읽는 데 사용되는 매개변수" **isWritable()** 메서드의 매개 변수에 대해 설명합니다.*

*표 **51.4.** 엔터티를 읽는 데 사용되는 매개변수*

매개변수	유형	설명
type	**Class<T>**	작성 중인 개체의 Java 클래스를 지정합니다.
genericType	유형	리소스 메서드 반환 유형의 리플렉션 또는 반환된 인스턴스의 검사를 통해 얻은 Java 유형의 개체를 지정합니다. 48.4절. "일반 유형 정보가 있는 엔터티 반환" 에 설명된 **GenericEntity** 클래스는 이 값을 제어하는 지원을 제공합니다.
annotations	annotation[]	엔터티를 반환하는 메서드의 주석 목록을 지정합니다.
mediaType	**MediatType**	HTTP 엔터티의 MIME 형식을 지정합니다.

getSize()

***getSize()** 메서드는 **writeTo()** 전에 호출됩니다. 기록되는 엔터티의 길이 (_ 바이트) 를 반환합니다.**Returns the length, in bytes, of the entity being written.** 양수 값이 반환되면 값은 **HTTP** 메시지의 **Content-Length** 헤더에 기록됩니다.*

표 51.5. "엔터티를 읽는 데 사용되는 매개변수" getSize() 메서드의 매개 변수에 대해 설명합니다.

표 51.5. 엔터티를 읽는 데 사용되는 매개변수

매개변수	유형	설명
t	generic	작성 중인 인스턴스를 지정합니다.
type	**Class<T>**	작성 중인 개체의 Java 클래스를 지정합니다.
genericType	유형	리소스 메서드 반환 유형의 리플렉션 또는 반환된 인스턴스의 검사를 통해 얻은 Java 유형의 개체를 지정합니다. 48.4절. "일반 유형 정보가 있는 엔터티 반환" 에 설명된 **GenericEntity** 클래스는 이 값을 제어하는 지원을 제공합니다.
annotations	annotation[]	엔터티를 반환하는 메서드의 주석 목록을 지정합니다.
mediaType	**MediatType**	HTTP 엔터티의 MIME 형식을 지정합니다.

writeTo()

***writeTo()** 메서드는 **Java** 개체를 원하는 엔터티 유형으로 변환하고 해당 엔터티를 출력 스트림에 씁니다. 출력 스트림에 엔터티를 작성할 때 오류가 발생하면 메서드가 **IOException** 예외를 **throw**해야 합니다.**If an error occurs when writing the entity to the output stream the method should throw an IOException exception. HTTP** 오류 응답이 필요한 오류가 발생하면 **HTTP** 응답이 있는 **WebApplicationException**을 **throw**해야 합니다.*

표 51.6. "엔터티를 읽는 데 사용되는 매개변수" writeTo() 메서드의 매개 변수에 대해 설명합니다.

표 51.6. 엔터티를 읽는 데 사용되는 매개변수

매개변수	유형	설명
t	generic	작성 중인 인스턴스를 지정합니다.
type	**Class<T>**	작성 중인 개체의 Java 클래스를 지정합니다.

매개변수	유형	설명
genericType	유형	리소스 메서드 반환 유형의 리플렉션 또는 반환된 인스턴스의 검사를 통해 얻은 Java 유형의 개체를 지정합니다. 48.4절. "일반 유형 정보가 있는 엔터티 반환" 에 설명된 **GenericEntity** 클래스는 이 값을 제어하는 지원을 제공합니다.
annotations	annotation[]	엔터티를 반환하는 메서드의 주석 목록을 지정합니다.
mediaType	**MediatType**	HTTP 엔터티의 MIME 형식을 지정합니다.
httpHeaders	MultivaluedMap<String, Object>	엔터티와 연결된 HTTP 응답 헤더를 지정합니다.
entityStream	**OutputStream**	엔터티가 작성되는 출력 스트림을 지정합니다.

***MessageBodyWriter** 구현을 엔터티 작성자로 사용하려면 **javax.ws.rs.ext.Provider** 주석으로 장식해야 합니다. **@Provider** 주석은 제공된 구현에서 추가 기능을 제공하는 런타임을 경고합니다. 구현도 "독자 및 작성자 등록" 에 설명된 대로 런타임에 등록되어 있어야 합니다.*

*기본적으로 사용자 지정 엔터티 공급자는 모든 **MIME** 유형을 처리합니다. 사용자 지정 엔터티 작성자가 **javax.ws.rs. inspector** 주석을 사용하여 처리할 **MIME** 유형을 제한할 수 있습니다. **@Produces** 주석은 사용자 지정 엔터티 공급자가 생성하는 **MIME** 유형의 쉼표로 구분된 목록을 지정합니다. 엔터티가 지정된 **MIME** 유형이 아닌 경우 엔터티 공급자가 가능한 작성기로 선택되지 않습니다.**If an entity is not of a specified MIME type, the entity provider will not be selected as a possible writer.***

*예 51.4. "XML 소스 엔터티 작성기" 소스 오브젝트를 가져와 **XML** 엔터티를 생성하는 엔터티 작성기를 표시합니다.*

예 51.4. XML 소스 엔터티 작성기

```
import java.io.IOException;
import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;

import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
```

```
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.ext.MessageBodyWriter;
import javax.ws.rs.ext.Provider;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;

import org.apache.cxf.jaxrs.ext.xml.XMLSource;

@Provider
@Produces({"application/xml", "application/*+xml", "text/xml" })
public class SourceProvider implements MessageBodyWriter<Source>
{

  public boolean isWriteable(Class<?> type,
                             Type genericType,
                             Annotation[] annotations,
                             MediaType mt)
  {
    return Source.class.isAssignableFrom(type);
  }

  public void writeTo(Source source,
                      Class<?> clazz,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mediatype,
                      MultivaluedMap<String, Object> httpHeaders,
                      OutputStream os)
  throws IOException
  {
    StreamResult result = new StreamResult(os);
    TransformerFactory tf = TransformerFactory.newInstance();
    try
    {
      Transformer t = tf.newTransformer();
      t.transform(source, result);
    }
    catch (TransformerException te)
    {
      te.printStackTrace();
      throw new WebApplicationException(te);
    }
  }

  public long getSize(Source source,
                      Class<?> type,
                      Type genericType,
                      Annotation[] annotations,
                      MediaType mt)
  {
```

```
    return -1;
  }
}
```

51.5. 독자 및 작성자 등록

JAX-RS 애플리케이션에서 사용자 지정 엔터티 공급자를 사용하려면 사용자 지정 공급자를 런타임으로 등록해야 합니다. 공급자는 애플리케이션 구성 파일의 **jaxrs:providers** 요소 또는 **JAXRSServerFactoryBean** 클래스를 사용하여 런타임에 등록됩니다.

jaxrs:providers 요소는 **jaxrs:server** 요소의 하위이며 빈 요소 목록 을 포함합니다. 각 빈 요소는 하나의 엔터티 공급자를 정의합니다.

예 51.5. “런타임을 사용하여 엔터티 공급자 등록” 사용자 지정 엔터티 공급자 집합을 사용하도록 구성된 **JAX-RS** 서버를 표시합니다.

예 51.5. 런타임을 사용하여 엔터티 공급자 등록

```
<beans ...>
  <jaxrs:server id="customerService" address="/">
    ...
    <jaxrs:providers>
      <bean id="isProvider" class="com.bar.providers.InputStreamProvider"/>
      <bean id="longProvider" class="com.bar.providers.LongProvider"/>
    </jaxrs:providers>
  </jaxrs:server>
</beans>
```

JAXRSServerFactoryBean 클래스는 구성 **API**에 대한 액세스를 제공하는 **Apache CXF** 확장입니다. 애플리케이션에 인스턴스화된 엔터티 공급자를 추가할 수 있는 **setProvider()** 메서드가 있습니다.
예 51.6. “엔터티 공급자를 프로그래밍 방식으로 등록” 엔터티 공급자를 프로그래밍 방식으로 등록하기 위한 코드를 보여줍니다.

예 51.6. 엔터티 공급자를 프로그래밍 방식으로 등록

```
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
...
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
...
SourceProvider provider = new SourceProvider();
sf.setProvider(provider);
...
```

52장. 컨텍스트 정보 가져오기 및 사용

초록

컨텍스트 정보에는 리소스의 URI, HTTP 헤더 및 기타 주입 주석을 사용하여 쉽게 사용할 수 없는 기타 세부 정보에 대한 세부 정보가 포함됩니다. Apache CXF는 가능한 모든 컨텍스트 정보를 단일 개체에 맞게 조정하는 특수 클래스를 제공합니다.

52.1. 컨텍스트 소개

52.1.1. 컨텍스트 주석

javax.ws.rs.core.Context 주석을 사용하여 컨텍스트 정보를 필드 또는 리소스 메서드 매개 변수에 삽입하도록 지정합니다. 컨텍스트 유형 중 하나의 필드 또는 매개 변수에 주석을 추가하면 런타임에서 주석이 달린 필드 또는 매개 변수에 적절한 컨텍스트 정보를 삽입하도록 지시합니다.

52.1.2. 컨텍스트 유형

표 52.1. "컨텍스트 유형" 삽입할 수 있는 컨텍스트 정보 유형과 해당 정보를 지원하는 오브젝트 유형을 나열합니다.

표 52.1. 컨텍스트 유형

개체	컨텍스트 정보
UriInfo	전체 요청 URI
HttpHeaders	HTTP 메시지 헤더
요청	최상의 표현 변형을 결정하거나 사전 조건 집합이 설정되었는지 여부를 결정하는 데 사용할 수 있는 정보
SecurityContext	사용 중인 인증 체계, 요청 채널이 안전한 경우, 요청 채널 및 사용자 원칙을 포함한 요청자의 보안에 대한 정보

52.1.3. 컨텍스트 정보를 사용할 수 있는 위치

컨텍스트 정보는 JAX-RS 애플리케이션의 다음 부분에서 사용할 수 있습니다.

- 리소스 클래스

- 리소스 메서드
- 엔터티 공급자
- 예외 매퍼

52.1.4. 범위

@Context 주석을 사용하여 삽입된 모든 컨텍스트 정보는 현재 요청과 관련이 있습니다. 이는 엔터티 공급자 및 예외 매퍼를 포함한 모든 경우에 해당합니다.

52.1.5. 컨텍스트 추가

JAX-RS 프레임워크를 사용하면 개발자가 컨텍스트 메커니즘을 사용하여 삽입할 수 있는 정보 유형을 확장할 수 있습니다. Context<T> 개체를 구현하고 런타임에 등록하여 사용자 지정 컨텍스트를 추가합니다.

52.2. 전체 요청 URI 작업

초록

요청 URI에는 상당한 양의 정보가 포함됩니다. 이 정보는 47.2.2절. “요청 URI에서 데이터 삽입” 설명된 대로 메서드 매개 변수를 사용하여 액세스할 수 있지만 매개 변수를 사용하면 URI 처리 방법에 대한 특정 제약 조건이 적용됩니다. URI 세그먼트에 액세스하는 데 매개 변수를 사용하면 전체 요청 URI에 대한 리소스 액세스도 제공하지 않습니다.

URI 컨텍스트를 리소스에 삽입하여 전체 요청 URI에 대한 액세스를 제공할 수 있습니다. URI는 UriInfo 개체로 제공됩니다. UriInfo 인터페이스는 여러 가지 방법으로 URI를 분해하는 기능을 제공합니다. URI를 클라이언트로 반환할 URI를 구성할 수 있는 UriBuilder 오브젝트로 URI를 제공할 수도 있습니다.

:experimental:

52.2.1. URI 정보 삽입

52.2.1.1. 개요

UriInfo 오브젝트인 클래스 필드 또는 메서드 매개 변수를 @Context 주석으로 장식하는 경우 현재 요청에 대한 URI 컨텍스트가 UriInfo 오브젝트에 삽입됩니다.

52.2.1.2. 예제

클래스 필드에 URI 컨텍스트 삽입 URI 컨텍스트를 삽입하여 채워진 필드가 있는 클래스를 표시합니다.

클래스 필드에 URI 컨텍스트 삽입

```
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.Path;
...
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo requestURI;
  ...
}
```

52.2.2. URI 작업

52.2.2.1. 개요

URI 컨텍스트를 사용할 때의 주요 이점 중 하나는 서비스의 기본 URI와 선택한 리소스에 대한 URI 경로 세그먼트에 대한 액세스를 제공한다는 것입니다. 이 정보는 URI에 따라 처리 결정을 내리거나 응답의 일부로 반환하도록 URI 계산과 같은 여러 목적에 유용할 수 있습니다. 예를 들어, 요청의 기본 URI에 .com 확장 기능이 포함되어 있는 경우 서비스가 US $를 사용할 수 있으며 기본 URI에 .co.uk 확장이 포함된 경우 영국 Pound를 결정할 수 있습니다.

UriInfo 인터페이스는 URI의 부분에 액세스하기 위한 메서드를 제공합니다.

- *기본 URI*
- *리소스 경로*
- *전체 URI*

52.2.2.2. 기본 URI 가져오기

기본 URI 는 서비스가 게시되는 루트 URI입니다. 서비스의 @Path 주석에 지정된 URI의 일부를 포함하지 않습니다. 예를 들어 예 47.5. "URI 디코딩 비활성화" 에 정의된 리소스를 구현하는 서비스가 http://fusesource.org 에 게시되고 http://fusesource.org/montersforhire/nightstalker?12 에서 요청이 작성된 경우 기본 URI는 http://fusesource.org입니다.

표 52.2. "리소스의 기본 URI에 액세스하는 방법" 기본 URI를 반환하는 메서드를 설명합니다.Describes the methods that return the base URI.

표 52.2. 리소스의 기본 URI에 액세스하는 방법

방법	Desription
URIgetBaseUri	서비스의 기본 URI를 **URI** 오브젝트로 반환합니다.
UriBuildergetBaseUriBuilder	기본 URI를 **javax.ws.rs.core.UriBuilder** 오브젝트로 반환합니다. **UriBuilder** 클래스는 서비스에서 구현하는 다른 리소스에 대한 URI를 생성하는 데 유용합니다.

52.2.2.3. 경로 가져오기Get the path

요청 URI의 경로 부분은 현재 리소스를 선택하는 데 사용된 URI 부분입니다. 기본 URI를 포함하지 않지만 URI에 포함된 URI 템플릿 변수와 매트릭스 매개 변수를 포함합니다.

경로 값은 선택한 리소스에 따라 다릅니다. 예를 들어 리소스의 경로 가져오기 에 정의된 리소스의 경로는 다음과 같습니다.

- *rootPath — /monstersforhire/*

- ***getterPath — /mostersforhire/nightstalker***

  ***GET** 요청은 **/monstersforhire/nightstalker** 에서 작성되었습니다.*

- ***putterPath — /mostersforhire/911***

  ***PUT** 요청은 **/monstersforhire/911** 에서 생성되었습니다.*

리소스의 경로 가져오기

```
@Path("/monstersforhire/")
public class MonsterService
{
  @Context
  UriInfo rootUri;

  ...

  @GET
  public List<Monster> getMonsters(@Context UriInfo getUri)
  {
    String rootPath = rootUri.getPath();
    ...
  }

  @GET
  @Path("/{type}")
  public Monster getMonster(@PathParam("type") String type,
                            @Context UriInfo getUri)
  {
    String getterPath = getUri.getPath();
    ...
  }

  @PUT
  @Path("/{id}")
  public void addMonster(@Encoded @PathParam("type") String type,
                         @Context UriInfo putUri)
  {
    String putterPath = putUri.getPath();
    ...
  }
  ...
}
```

표 52.3. "리소스 경로에 액세스하는 방법" 리소스 경로를 반환하는 메서드를 설명합니다.Describes the methods that return the resource path.

표 52.3. 리소스 경로에 액세스하는 방법

방법	Desription
StringgetPath	리소스의 경로를 디코딩된 URI로 반환합니다.
stringgetPath부울decode	리소스의 경로를 반환합니다. **false** 를 지정하면 URI 디코딩이 비활성화됩니다.
List<PathSegment>getPathSegments	디코딩된 경로를 javax.ws.rs.core.PathSegment 오브젝트 목록으로 반환합니다. 매트릭스 매개 변수를 포함한 경로의 각 부분은 목록의 고유한 항목에 배치됩니다. 예를 들어 리소스 경로 **상자/round# tall** 은 세 개의 항목, **박스,라운드**, 키가 있는 목록을 만듭니다.
List<PathSegment>getPathSegmentsbooleandecode	경로를 javax.ws.rs.core.PathSegment 오브젝트 목록으로 반환합니다. 매트릭스 매개 변수를 포함한 경로의 각 부분은 목록의 고유한 항목에 배치됩니다. **false** 를 지정하면 URI 디코딩이 비활성화됩니다. 예를 들어 리소스 경로 **box#tall /round** 를 사용하면 **박스**, 키가 넓고 **라운드** 의 세 항목이 있는 목록이 생성됩니다.

52.2.2.4. 전체 요청 URI 만들기

표 52.4. "전체 요청 URI에 액세스하는 방법" 전체 요청 URI를 반환하는 메서드를 설명합니다.Describes the methods that return the full request URI. 요청 URI를 반환하거나 리소스의 절대 경로를 반환하는 옵션이 있습니다. 차이점은 요청 URI에는 URI에 추가된 쿼리 매개 변수가 포함되어 있으며 절대 경로에 쿼리 매개 변수가 포함되지 않는다는 것입니다.

표 52.4. 전체 요청 URI에 액세스하는 방법

방법	Desription
URIgetRequestUri	쿼리 매개 변수와 매트릭스 매개 변수를 포함한 전체 요청 URI를 **java.net.URI** 오브젝트로 반환합니다.

방법	Desription
UriBuildergetRequestUriBuilder	**javax.ws.rs.UriBuilder** 오브젝트로 쿼리 매개변수 및 매트릭스 매개변수를 포함한 전체 요청 URI를 반환합니다. **UriBuilder** 클래스는 서비스에서 구현하는 다른 리소스에 대한 URI를 생성하는 데 유용합니다.
URIgetAbsolutePath	matrix 매개 변수를 포함한 전체 요청 URI를 **java.net.URI** 오브젝트로 반환합니다. 절대 경로에는 쿼리 매개변수가 포함되지 않습니다.
UriBuildergetAbsolutePathBuilder	matrix 매개 변수를 포함한 전체 요청 URI를 **javax.ws.rs.UriBuilder** 오브젝트로 반환합니다. 절대 경로에는 쿼리 매개변수가 포함되지 않습니다.

URI http://fusesource.org/montersforhire/nightstalker?12.을 사용하여 이루어진 요청의 경우 getRequestUri() 메서드가 http://fusesource.org/montersforhire/nightstalker?12 를 반환합니다. getAbsolutePath() 메서드는 http://fusesource.org/montersforhire/nightstalker 를 반환합니다.

52.2.3. URI 템플릿 변수의 값 가져오기

52.2.3.1. 개요

"경로 설정" 에서 설명한 대로 리소스 경로에는 동적으로 값에 바인딩되는 변수 세그먼트가 포함될 수 있습니다. 종종 이러한 변수 경로 세그먼트는 "URI 경로에서 데이터 가져오기" 에 설명된 대로 리소스 방법에 대한 매개 변수로 사용됩니다. 그러나 URI 컨텍스트를 통해 액세스할 수도 있습니다.

52.2.3.2. 경로 매개 변수를 가져오는 방법

***UriInfo** 인터페이스는 경로 매개변수 목록을 반환하는 예 52.1. "URI 컨텍스트에서 경로 매개 변수를 반환하는 방법" 에 표시된 두 가지 메서드를 제공합니다.*

> *예 52.1. URI 컨텍스트에서 경로 매개 변수를 반환하는 방법*
>
> ***MultivaluedMap<java.lang.String, java.lang.String>getPathParametersMultivaluedMap<java.lang.String, java.lang.String>getPathParameters부울decode***

매개 변수를 사용하지 않는 **getPathParameters()** 메서드는 경로 매개 변수를 자동으로 디코딩합니다. **URI** 디코딩을 비활성화하려면 **getPathParameters(false)** 를 사용합니다.

값은 해당 템플릿 식별자를 키로 사용하여 맵에 저장됩니다. 예를 들어 리소스에 대한 **URI** 템플릿이 **/{color}/box/{note}** 인 경우 반환된 맵에는 키가 **color** 인 두 개의 항목이 있고 유의 합니다.

52.2.3.3. 예제

예 52.2. "URI 컨텍스트에서 경로 매개변수 추출" **URI** 컨텍스트를 사용하여 경로 매개 변수를 검색하는 코드를 표시합니다.

예 52.2. URI 컨텍스트에서 경로 매개변수 추출

```
import javax.ws.rs.Path;
import javax.ws.rs.Get;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.core.MultivaluedMap;

@Path("/monstersforhire/")
public class MonsterService

  @GET
  @Path("/{type}/{size}")
  public Monster getMonster(@Context UriInfo uri)
  {
    MultivaluedMap paramMap = uri.getPathParameters();
    String type = paramMap.getFirst("type");
    String size = paramMap.getFirst("size");
  }
}
```

53장. 주석 INHERITANCE

초록

JAX-RS 주석은 주석이 있는 인터페이스를 구현하는 하위 클래스 및 클래스로 상속할 수 있습니다. 상속 메커니즘을 사용하면 하위 클래스와 구현 클래스가 부모로부터 상속된 주석을 재정의할 수 있습니다.

53.1. 개요

상속은 개발자가 특정 요구 사항을 충족하기 위해 특화될 수 있는 제네릭 개체를 만들 수 있기 때문에 Java에서 더 강력한 메커니즘 중 하나입니다. JAX-RS는 클래스를 매핑하는 데 사용되는 주석을 수퍼 클래스에서 상속하도록 허용하여 이 전원을 유지합니다.

JAX-RS 주석 상속은 인터페이스를 지원하기 위해 확장됩니다. 구현 클래스는 구현되는 인터페이스에서 사용되는 JAX-RS 주석을 상속합니다.

JAX-RS 상속 규칙은 상속된 주석을 재정의하는 메커니즘을 제공합니다. 그러나 super 클래스 또는 인터페이스에서 해당 주석을 상속하는 구문에서 JAX-RS 주석을 완전히 제거할 수는 없습니다.

53.2. 상속 규칙

리소스 클래스는 구현하는 인터페이스에서 JAX-RS 주석을 상속합니다. 리소스 클래스는 확장하는 모든 슈퍼 클래스에서 JAX-RS 주석도 상속합니다. 슈퍼 클래스에서 상속된 주석은 am 인터페이스에서 상속된 주석보다 우선합니다.

예 53.1. “주석 상속” 에 표시된 코드 샘플에서 Kaijin 클래스의 getMonster() 메서드는 Kaiju 인터페이스에서 @Path,@GET 및 @PathParam 주석을 상속합니다.

예 53.1. 주석 상속

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  public Monster getMonster(@PathParam("id") int id);
  ...
}
```

```
@Path("/kaijin")
public class Kaijin implements Kaiju
{
  public Monster getMonster(int id)
  {
    ...
  }
  ...
}
```

53.3. 상속된 주석 덮어쓰기

상속된 주석을 재정의하는 것은 새 주석을 제공하는 것만큼 쉽습니다. 하위 클래스 또는 구현 클래스에서 메서드에 대한 자체 **JAX-RS** 주석을 제공하는 경우 해당 메서드에 대한 모든 **JAX-RS** 주석은 무시됩니다.

예 53.2. “주석 상속 덮어쓰기” 에 표시된 코드 샘플에서 **Kaijin** 클래스의 **getMonster()** 메서드는 **Kaiju** 인터페이스에서 주석을 상속하지 않습니다. 구현 클래스는 **@inspector** 주석 을 재정의하여 인터페이스의 모든 주석이 무시됩니다.

예 53.2. 주석 상속 덮어쓰기

```
public interface Kaiju
{
  @GET
  @Path("/{id}")
  @Produces("text/xml");
  public Monster getMonster(@PathParam("id") int id);
  ...
}

@Path("/kaijin")
public class Kaijin implements Kaiju
{

  @GET
  @Path("/{id}")
  @Produces("application/octect-stream");
  public Monster getMonster(@PathParam("id") int id)
  {
    ...
  }
  ...
}
```

54장. OPENAPI 지원을 사용하여 JAX-RS 끝점 확장

초록

CXF OpenApiFeature(org.apache.cxf.jaxrs.openapi.OpenApiFeature)를 사용하면 간단한 구성으로 게시된 JAX-RS 서비스 끝점을 확장하여 OpenAPI 문서를 생성할 수 있습니다.

OpenApiFeature는 Spring Boot 및 Karaf 구현 모두에서 지원됩니다.

54.1. OPENAPIFEATURE 옵션

OpenApiFeature에서 다음 옵션을 사용할 수 있습니다.

표 54.1. OpenApiFeature 작업

이름	설명	기본값
configLocation	OpenAPI 구성 위치	null
contactEmail	연락처 이메일+	null
contactName	연락처 이름 +	null
contactUrl	연락처 링크 +	null
Customizer	customr 클래스 인스턴스	null
description	description+	null
filterClass	보안 필터 ++	null
ignoredRoutes	모든 리소스를 스캔할 때 특정 경로 제외 (**검사AllResources**++ 참조)	null
라이센스	라이센스 +	null
licenseUrl	라이센스 URL+	null
prettyPrint	openapi.json을 생성할 때 JSON 문서++를 크게 인쇄합니다.	true
propertiesLocation	속성 파일 위치	**/swagger.properties**

이름	설명	기본값
readAllResources	@Operation++ 없이 모든 작업을 읽어 보십시오.	true
resourceClasses	검색해야 하는 리소스 클래스 목록입니다.	null
resourcePackages	리소스를 스캔해야 하는 패키지 이름 목록 ++	null
runAsFilter	필터로 기능 실행	false
scan	모든 JAX-RS 리소스를 자동으로 스캔	true

scanKnownConfigLocations	다음과 같은 알려진 OpenAPI 구성 위치(classpath 또는 파일 시스템)를 스캔합니다.
	`openapi-configuration.yaml`
	`openapi-configuration.json`
	`openapi.yaml`
	`openapi.json`
		true

scannerClass	애플리케이션, 리소스 패키지, 리소스 클래스, 클래스 경로 스캔의 범위를 지정하는 데 사용되는 JAX-RS API 스캐너 클래스의 이름은 Resource Scanning 섹션을 참조하십시오.	null
securityDefinitions	보안 정의 목록+	null
supportSwaggerUi	SwaggerUI 지원	null(== true)
swaggerUiConfig	Swagger UI 구성	null
swaggerUiMavenGroupAndArtifact	SwaggerUI를 캡처하기 위한 Maven 아티팩트	null
swaggerUiVersion	SwaggerUI의 버전	null
termsOfServiceUrl	서비스 URL 추가	null
title	제목 +	null

이름	설명	기본값
useContextBasedConfig	설정된 경우 각 OpenApiContext 인스턴스에 대해 고유한 컨텍스트 Id가 생성됩니다(여러 서버 끝점 사용 참조). 또한 검사 속성을 false로 설정할 수 있습니다.	false
version	version+	null

+ 옵션은 OpenAPI 클래스에 정의되어 있습니다.

++ 옵션은 SwaggerConfiguration 클래스에 정의되어 있습니다.

54.2. KARAF IMPLEMENTATIONS

이 섹션에서는 JAR 파일 내에 정의된 REST 서비스를 사용하고 Karaf 컨테이너의 Fuse에 배포되는 OpenApiFeature를 사용하는 방법에 대해 설명합니다.

54.2.1. 빠른 시작 예

Fuse 소프트웨어 다운로드 페이지에서 Red Hat Fuse 빠른 시작 을 다운로드할 수 있습니다.

빠른 시작 zip 파일에는 CXF를 사용하여 RESTful(JAX-RS) 웹 서비스를 생성하는 방법과 OpenAPI를 활성화하고 JAX-RS 엔드포인트에 주석을 추가하는 방법을 설명하는 빠른 시작용 /cxf/rest/ 디렉터리가 포함되어 있습니다.

54.2.2. OpenAPI 활성화

OpenAPI를 활성화하려면 다음을 수행해야 합니다.

- *CXF 클래스(org.apache.cxf.jaxrs.openapi.OpenApiFeature)를 < jaxrs:server > 정의에 추가하여 CXF 서비스를 정의하는 XML 파일을 수정합니다.*

  예를 들어 55.4 예제 XML 파일 에서 참조하십시오.

- *REST 리소스 클래스에서 다음을 수행합니다.*
  - *서비스에 필요한 각 주석에 대해 OpenAPI 주석을 가져옵니다.*

    ```
    import io.swagger.annotations.*
    ```

    *여기서 * = Api Operation,ApiParam,ApiResponse,ApiResponses 등.*

    자세한 내용은 https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X 으로 이동합니다.

    예를 들어 55.5 예제 리소스 클래스 에서 참조하십시오.

  - *JAX-RS 주석에 OpenAPI 주석 추가(@PATH,@PUT,@ GET ,@GET,@ GET ,@Consumes,@DELETE,@PathParam 등).*

예를 들어 55.5 예제 리소스 클래스 에서 참조하십시오.

55.4 예제 XML 파일

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
   xmlns:cxf="http://cxf.apache.org/blueprint/core"
   xsi:schemaLocation="
     http://www.osgi.org/xmlns/blueprint/v1.0.0
https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
     http://cxf.apache.org/blueprint/jaxrs
http://cxf.apache.org/schemas/blueprint/jaxrs.xsd
     http://cxf.apache.org/blueprint/core
http://cxf.apache.org/schemas/blueprint/core.xsd">

   <jaxrs:server id="customerService" address="/crm">
      <jaxrs:serviceBeans>
         <ref component-id="customerSvc"/>
      </jaxrs:serviceBeans>
      <jaxrs:providers>
        <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
      </jaxrs:providers>
      <jaxrs:features>
```

```
        <bean class="org.apache.cxf.jaxrs.openapi.OpenApiFeature">
          <property name="title" value="Fuse:CXF:Quickstarts - Customer Service" />
          <property name="description" value="Sample REST-based Customer Service" />
          <property name="version" value="${project.version}" />
        </bean>
        </jaxrs:features>
    </jaxrs:server>

    <cxf:bus>
        <cxf:features>
          <cxf:logging />
        </cxf:features>
        <cxf:properties>
            <entry key="skip.default.json.provider.registration" value="true" />
        </cxf:properties>
    </cxf:bus>

    <bean id="customerSvc" class="org.jboss.fuse.quickstarts.cxf.rest.CustomerService"/>

</blueprint>
```

55.5 예제 리소스 클래스

```
.
.
.

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;

import io.swagger.annotations.Api;
import io.swagger.annotations.ApiOperation;
import io.swagger.annotations.ApiParam;
import io.swagger.annotations.ApiResponse;
import io.swagger.annotations.ApiResponses;

.
.
.

@Path("/customerservice/")
```

```
@Api(value = "/customerservice", description = "Operations about customerservice")
public class CustomerService {

    private static final Logger LOG =
        LoggerFactory.getLogger(CustomerService.class);

    private MessageContext jaxrsContext;
    private long currentId              = 123;
    private Map<Long, Customer> customers = new HashMap<>();
    private Map<Long, Order> orders       = new HashMap<>();

    public CustomerService() {
        init();
    }


    @GET
    @Path("/customers/{id}/")
    @Produces("application/xml")
    @ApiOperation(value = "Find Customer by ID", notes = "More notes about this
        method", response = Customer.class)
    @ApiResponses(value = {
            @ApiResponse(code = 500, message = "Invalid ID supplied"),
            @ApiResponse(code = 204, message = "Customer not found")
            })
    public Customer getCustomer(@ApiParam(value = "ID of Customer to fetch",
        required = true) @PathParam("id") String id) {
        LOG.info("Invoking getCustomer, Customer id is: {}", id);
        long idNumber = Long.parseLong(id);
        return customers.get(idNumber);
    }


    @PUT
    @Path("/customers/")
    @Consumes({ "application/xml", "application/json" })
    @ApiOperation(value = "Update an existing Customer")
    @ApiResponses(value = {
            @ApiResponse(code = 500, message = "Invalid ID supplied"),
            @ApiResponse(code = 204, message = "Customer not found")
            })
    public Response updateCustomer(@ApiParam(value = "Customer object that needs
        to be updated", required = true) Customer customer) {
        LOG.info("Invoking updateCustomer, Customer name is: {}", customer.getName());
        Customer c = customers.get(customer.getId());
        Response r;
        if (c != null) {
            customers.put(customer.getId(), customer);
            r = Response.ok().build();
        } else {
            r = Response.notModified().build();
        }

        return r;
    }
```

```
    @POST
    @Path("/customers/")
    @Consumes({ "application/xml", "application/json" })
    @ApiOperation(value = "Add a new Customer")
    @ApiResponses(value = { @ApiResponse(code = 500, message = "Invalid ID
        supplied"), })
    public Response addCustomer(@ApiParam(value = "Customer object that needs to
        be updated", required = true) Customer customer) {
        LOG.info("Invoking addCustomer, Customer name is: {}", customer.getName());
        customer.setId(++currentId);

        customers.put(customer.getId(), customer);
        if (jaxrsContext.getHttpHeaders().getMediaType().getSubtype().equals("json"))
    {
            return Response.ok().type("application/json").entity(customer).build();
        } else {
            return Response.ok().type("application/xml").entity(customer).build();
        }
    }

    @DELETE
    @Path("/customers/{id}/")
    @ApiOperation(value = "Delete Customer")
    @ApiResponses(value = {
            @ApiResponse(code = 500, message = "Invalid ID supplied"),
            @ApiResponse(code = 204, message = "Customer not found")
            })
    public Response deleteCustomer(@ApiParam(value = "ID of Customer to delete",
        required = true) @PathParam("id") String id) {
        LOG.info("Invoking deleteCustomer, Customer id is: {}", id);
        long idNumber = Long.parseLong(id);
        Customer c = customers.get(idNumber);

        Response r;
        if (c != null) {
            r = Response.ok().build();
            customers.remove(idNumber);
        } else {
            r = Response.notModified().build();
        }

        return r;
    }

.
.
.

}
```

54.3. SPRING BOOT IMPLEMENTATIONS

이 섹션에서는 Spring Boot에서 Swagger2Feature를 사용하는 방법에 대해 설명합니다.

OpenAPI 3 구현의 경우 OpenApiFeature(org.apache.cxf.jaxrs.openapi.OpenApiFeature)를 사용하십시오.

54.3.1. 빠른 시작 예

빠른 시작 예 (https://github.com/fabric8-quickstarts/spring-boot-cxf-jaxrs)는 Spring Boot와 함께 Apache CXF를 사용하는 방법을 보여줍니다. 빠른 시작에서는 Swagger가 활성화된 CXF JAX-RS 끝점을 포함하는 애플리케이션을 구성하는 Spring Boot를 사용합니다.

54.3.2. Swagger 활성화

Swagger를 활성화하는 데는 다음이 포함됩니다.

- REST 애플리케이션에서 다음을 수행합니다.
  - Swagger2feature 가져오기:

```
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
```

  - CXF 엔드포인트에 Swagger2Feature 추가:

```
endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
```

    예를 들어 55.1 예 REST 애플리케이션 예 에서 참조하십시오.

- Java 구현 파일에서 서비스에 필요한 각 주석에 대해 Swagger API 주석을 가져옵니다.

```
import io.swagger.annotations.*
```

  여기서 * = Api Operation,ApiParam,ApiResponse,ApiResponses 등.

  자세한 내용은 https://github.com/swagger-api/swagger-core/wiki/Annotations 에서 참

조하십시오.

예를 들어 55.2 예 Java 구현 파일 에서 참조하십시오.

- Java 파일에서 Swagger 주석을 JAX-RS 주석에 추가합니다(@PATH,@PUT,@ GET ,@GET,@ 509 ,@Consumes,@DELETE,@PathParam 등).

예를 들어 55.3 예 Java 파일 에서 참조하십시오.

55.1 예 REST 애플리케이션 예

```
package io.fabric8.quickstarts.cxf.jaxrs;

import java.util.Arrays;

import org.apache.cxf.Bus;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.swagger.Swagger2Feature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SampleRestApplication {

    @Autowired
    private Bus bus;

    public static void main(String[] args) {
        SpringApplication.run(SampleRestApplication.class, args);
    }

    @Bean
    public Server rsServer() {
        // setup CXF-RS
        JAXRSServerFactoryBean endpoint = new JAXRSServerFactoryBean();
        endpoint.setBus(bus);
        endpoint.setServiceBeans(Arrays.<Object>asList(new HelloServiceImpl()));
        endpoint.setAddress("/");
        endpoint.setFeatures(Arrays.asList(new Swagger2Feature()));
        return endpoint.create();
    }
}
```

55.2 예 Java 구현 파일

```
import io.swagger.annotations.Api;

@Api("/sayHello")
public class HelloServiceImpl implements HelloService {

    public String welcome() {
        return "Welcome to the CXF RS Spring Boot application, append /{name} to call the hello
service";
    }

    public String sayHello(String a) {
        return "Hello " + a + ", Welcome to CXF RS Spring Boot World!!!";
    }

}
```

55.3 예 Java 파일

```
package io.fabric8.quickstarts.cxf.jaxrs;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.springframework.stereotype.Service;

@Path("/sayHello")
@Service
public interface HelloService {

    @GET
    @Path("")
    @Produces(MediaType.TEXT_PLAIN)
    String welcome();

    @GET
    @Path("/{a}")
    @Produces(MediaType.TEXT_PLAIN)
```

```
    String sayHello(@PathParam("a") String a);

}
```

54.4. OPENAPI 문서 액세스

OpenApiFeature에서 OpenAPI를 활성화하면 서비스 끝점 위치 다음에 /openapi.json 또는 /openapi.yaml 을 사용하여 구성된 위치 URL에서 OpenAPI 문서를 사용할 수 있습니다.

예를 들어 http://host:port/context/services/ 에 게시된 JAX-RS 엔드포인트의 경우 context 가 웹 애플리케이션 컨텍스트이고 /services 는 서블릿 URL인 경우 OpenAPI 문서는 http://host:port/context/services/openapi.json 및 http://host:port/context/services/openapi.yaml 에서 사용할 수 있습니다.

OpenApiFeature가 활성화되어 있으면 CXF Services 페이지가 OpenAPI 문서에 연결됩니다.

위의 예에서는 http://host:port/context/services/services 로 이동한 다음 OpenAPI JSON 문서를 반환하는 링크를 따릅니다.

다른 호스트의 OpenAPI UI에서 정의에 액세스하는 데 CORS 지원이 필요한 경우 cxf-rt-rs-security-cors 에서 CrossOriginResourceSharingFilter 를 추가할 수 있습니다.

54.5. 역방향 프록시를 통해 OPENAPI에 액세스

역방향 프록시를 통해 OpenAPI JSON 문서 또는 OpenAPI UI에 액세스하려면 다음 옵션을 설정합니다.

- CXFServlet use-x-forwarded-headers init 매개변수를 true 로 설정합니다.
  - Spring Boot에서 매개 변수 이름 앞에 cxf.servlet.init 를 접두사로 지정합니다.

```
cxf.servlet.init.use-x-forwarded-headers=true
```

- **Karaf에서 installDir/etc/org.apache.cxf.osgi.cfg 구성 파일에 다음 행을 추가합니다.**

```
cxf.servlet.init.use-x-forwarded-headers=true
```

참고: etc 디렉터리에 org.apache.cxf.osgi.cfg 파일이 아직 없는 경우 하나를 생성할 수 있습니다.

- **OpenApiFeature basePath 옵션의 값을 지정하고 OpenAPI가 basePath 값을 캐싱하지 못하도록 하려면 OpenApiFeature usePathBasedConfig 옵션을 TRUE 로 설정합니다.**

```
<bean class="org.apache.cxf.jaxrs.openapi.OpenApiFeature">
   <property name="usePathBasedConfig" value="TRUE" />
</bean>
```

VII 부. APACHE CXF 인터셉터 개발

*이 가이드에서는 메시지에서 사전 및 사후 처리를 수행할 수 있는 **Apache CXF** 인터셉터를 작성하는 방법을 설명합니다.*

55장. APACHE CXF 런타임의 인터셉터

초록

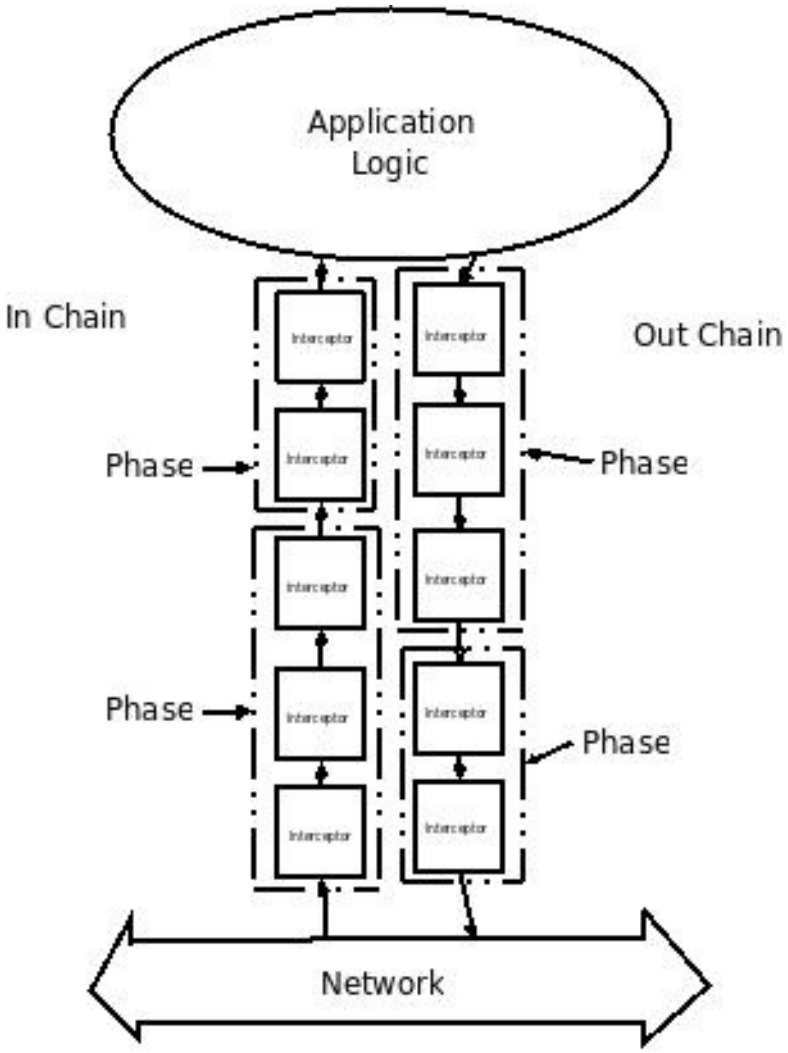
Apache CXF 런타임의 대부분의 기능은 인터셉터에 의해 구현됩니다. Apache CXF 런타임에서 생성한 모든 끝점에는 메시지 처리를 위한 3개의 잠재적인 인터셉터 체인이 있습니다. 이러한 체인의 인터셉터는 전선에서 전송되는 원시 데이터와 끝점의 구현 코드에서 처리하는 Java 오브젝트 간에 메시지를 변환하는 역할을 합니다. 인터셉터는 적절한 순서로 처리가 이루어지도록 단계별로 구성됩니다.

55.1. 개요

Apache CXF가 처리하는 주요 부분에는 메시지 처리 작업이 포함됩니다. 소비자가 원격 서비스에서 호출하면 런타임에서 데이터를 사용 가능한 메시지로 마샬링해야 합니다. 서비스 공급자는 메시지를 손상되지 않고 비즈니스 논리를 실행하고 적절한 메시지 형식으로 응답을 마샬링해야 합니다. 그런 다음 소비자는 응답 메시지를 미수하고 적절한 요청과 상호 작용한 후 소비자의 애플리케이션 코드로 다시 전달해야 합니다. 기본적인 마샬링 및 unmarshaling 외에도 Apache CXF 런타임은 메시지 데이터를 사용하여 여러 다른 작업을 수행할 수 있습니다. 예를 들어 WS-RM이 활성화된 경우 런타임은 메시지를 마샬링하고 구분 해제하기 전에 메시지 청크 및 승인 메시지를 처리해야 합니다. 보안이 활성화되면 런타임은 메시지 처리 순서의 일부로 메시지의 자격 증명을 검증해야 합니다.

그림 55.1. “Apache CXF 인터셉터 체인” 서비스 공급자가 수신할 때 요청 메시지가 걸리는 기본 경로를 표시합니다.

그림 55.1. Apache CXF 인터셉터 체인



55.2. APACHE CXF에서 메시지 처리

***Apache CXF** 개발 소비자가 원격 서비스를 호출하면 다음과 같은 메시지 처리 시퀀스가 시작됩니다.*

1. ***Apache CXF** 런타임은 요청을 처리하는 아웃바운드 인터셉터 체인을 생성합니다.*

2. *호출이 양방향 메시지 교환을 시작하면 런타임은 인바운드 인터셉터 체인과 오류 처리 인터셉터 체인을 생성합니다.*

3. *요청 메시지는 아웃바운드 인터셉터 체인을 통해 순차적으로 전달됩니다.*

   *체인의 각 인터셉터는 메시지에서 일부 처리를 수행합니다. 예를 들어 **Apache CXF**는 **SOAP** 인터셉터를 제공하고 **SOAP** 봉투로 메시지를 패키징했습니다.*

4.

*아웃바운드 체인의 인터셉터 중 하나라도 오류 조건을 생성하면 체인이 **unwound**이고 제어가 애플리케이션 수준 코드로 반환됩니다.*

*인터셉터 체인은 이전에 호출한 모든 인터셉터에서 **fault** 처리 방법을 호출하여 작동하지 않습니다.*

5. *요청은 적절한 서비스 공급자로 디스패치됩니다.*

6. *응답이 수신되면 인바운드 인터셉터 체인을 통해 순차적으로 전달됩니다.*

참고

응답이 오류 메시지이면 오류 처리 인터셉터 체인으로 전달됩니다.

7. *인바운드 체인의 인터셉터 중 하나라도 오류 조건을 생성하면 체인이 불필요합니다.*

8. *메시지가 인바운드 인터셉터 체인의 끝에 도달하면 애플리케이션 코드로 다시 전달됩니다.*

***Apache CXF** 개발 서비스 공급자가 소비자로부터 요청을 수신하면 비슷한 프로세스가 수행됩니다.*

1. ***Apache CXF** 런타임은 요청 메시지를 처리하는 인바운드 인터셉터 체인을 생성합니다.*

2. *요청이 양방향 메시지 교환의 일부인 경우 런타임은 아웃바운드 인터셉터 체인과 오류 처리 인터셉터 체인도 생성합니다.*

3. *요청은 인바운드 인터셉터 체인을 통해 순차적으로 전달됩니다.*

4. *인바운드 체인의 인터셉터 중 하나라도 오류 조건을 생성하면 체인이 **unwound**이고 오류가 소비자에게 디스패치됩니다.*

*인터셉터 체인은 이전에 호출한 모든 인터셉터에서 **fault** 처리 방법을 호출하여 작동하지 않습니다.*

5. *요청이 인바운드 인터셉터 체인의 끝에 도달하면 서비스 구현으로 전달됩니다.*

6. *응답이 준비되면 아웃바운드 인터셉터 체인을 통해 순차적으로 전달됩니다.*

참고

응답이 예외인 경우 오류 처리 인터셉터 체인을 통해 전달됩니다.

7. *아웃바운드 체인의 인터셉터 중 하나라도 오류 조건을 생성하면 체인이 불필요하고 오류 메시지가 디스패치됩니다.*

8. *요청이 아웃바운드 체인의 끝에 도달하면 소비자에게 디스패치됩니다.*

55.3. 인터셉터

***Apache CXF** 런타임의 모든 메시지 처리는 인터셉터에서 수행합니다. 인터셉터는 애플리케이션 계층에 전달되기 전에 메시지 데이터에 액세스할 수 있는 **POJO**입니다. 메시지를 변환하거나, 메시지에서 헤더를 제거하거나, 메시지 데이터를 검증하는 등 여러 가지 작업을 수행할 수 있습니다. 예를 들어 인터셉터는 메시지에서 보안 헤더를 읽고, 외부 보안 서비스에 대한 인증 정보를 확인하고, 메시지 처리를 계속할 수 있는지 결정할 수 있습니다.*

인터셉터에서 사용할 수 있는 메시지 데이터는 여러 요인으로 결정됩니다.

- *인터셉터 체인*
- *인터셉터의 단계*
- *체인에서 이전에 발생한 다른 인터셉터*

55.4. 단계

*인터셉터는 단계로 구성됩니다. 단계는 공통 기능이 있는 인터셉터의 논리적 그룹입니다. 각 단계는 특정 유형의 메시지 처리를 담당합니다. 예를 들어 애플리케이션 계층에 전달되는 마샬링 **Java** 개체를 처리*

하는 인터셉터는 모두 동일한 단계에서 수행됩니다.

55.5. 인터셉터 체인

단계는 인터셉터 체인에 집계됩니다. 인터셉터 체인은 메시지가 인바운드 또는 아웃바운드인지를 기반으로 정렬되는 인터셉터 단계 목록입니다.

Apache CXF를 사용하여 생성된 각 엔드포인트에는 세 개의 인터셉터 체인이 있습니다.

- 인바운드 메시지의 체인
- 아웃바운드 메시지의 체인
- 오류 메시지의 체인

인터셉터 체인은 주로 끝점에서 사용하는 바인딩 및 전송의 선택에 따라 구성됩니다. 보안 또는 로깅과 같은 다른 런타임 기능을 추가하면 체인에 인터셉터도 추가됩니다. 개발자는 구성을 사용하여 체인에 사용자 정의 인터셉터를 추가할 수도 있습니다.

55.6. 인터셉터 개발

기능에 관계없이 인터셉터를 개발하는 작업은 항상 동일한 기본 절차를 따릅니다.

1. **56장. Interceptor API**

   Apache CXF는 사용자 정의 인터셉터를 더 쉽게 개발할 수 있도록 다양한 추상 인터셉터를 제공합니다.

2. **57.2절. "인터셉터의 단계 지정"**

   인터셉터를 사용하려면 메시지의 특정 부분을 사용할 수 있어야 하며 데이터가 특정 형식이어야 합니다. 메시지의 콘텐츠와 데이터 형식은 인터셉터 단계에 따라 부분적으로 결정됩니다.

3. *57.3절. “단계에서 인터셉터 배치 구성”*

   일반적으로 단계 내의 인터셉터 순서는 중요하지 않습니다. 그러나 특정 상황에서는 인터셉터가 동일한 단계에서 다른 인터셉터 이전 또는 이후에 실행되는지 확인해야 할 수 있습니다.

4. *58.2절. “메시지 처리”*

5. *58.3절. “오류 후 정리 취소”*

   인터셉터가 실행된 후 활성 인터셉터 체인에서 오류가 발생하면 해당 오류 처리 논리가 호출됩니다.

6. *59장. Interceptors를 사용하도록 끝점 구성*

56장. INTERCEPTOR API

초록

인터셉터는 기본 인터셉터 인터페이스를 확장하는 **PhaseInterceptor** 인터페이스를 구현합니다. 이 인터페이스는 **Apache CXF** 런타임에서 인터셉터 실행을 제어하기 위해 사용하는 여러 방법을 정의하고 애플리케이션 개발자가 구현하는 데 적합하지 않습니다. 인터셉터 개발을 단순화하기 위해 **Apache CXF**는 확장 가능한 여러 추상 인터셉터 구현을 제공합니다.

56.1. 인터페이스

Apache CXF의 모든 인터셉터는 예 56.1. "기본 인터셉터 인터페이스" 에 표시된 기본 인터셉터 인터페이스를 구현합니다.

예 56.1. 기본 인터셉터 인터페이스

```
package org.apache.cxf.interceptor;

public interface Interceptor<T extends Message>
{

  void handleMessage(T message) throws Fault;

  void handleFault(T message);

}
```

Interceptor 인터페이스는 개발자가 사용자 지정 인터셉터를 위해 구현하는 데 필요한 두 가지 방법을 정의합니다.

handleMessage()

handleMessage() 메서드는 인터셉터에서 대부분의 작업을 수행합니다. 메시지 체인의 각 인터셉터에서 호출되고 처리 중인 메시지의 내용을 받습니다. 개발자는 이 방법의 인터셉터의 메시지 처리 로직을 구현합니다. **handleMessage()** 메서드 구현에 대한 자세한 내용은 58.2절. "메시지 처리" 을 참조하십시오.

handleFault()

일반 메시지 처리가 중단되면 **handleFault()** 메서드가 인터셉터에서 호출됩니다. 런타임은 인터셉터 체인의 회수를 해제하므로 각 인터셉터의 **handleFault()** 메서드를 역순으로 호출합니다. **handleFault()** 메서드 구현에 대한 자세한 내용은 58.3절. "오류 후 정리 취소" 을 참조하십시오.

대부분의 인터셉터는 **Interceptor** 인터페이스를 직접 구현하지 않습니다. 대신 예 56.2. "phase 인터셉터 인터페이스" 에 표시된 단계 **Interceptor** 인터페이스를 구현합니다. **phaseInterceptor** 인터페이스는 인터셉터 체인에 참여할 수 있는 네 가지 방법을 추가합니다.

예 56.2. phase 인터셉터 인터페이스

```
package org.apache.cxf.phase;
...

public interface PhaseInterceptor<T extends Message> extends Interceptor<T>
{

    Set<String> getAfter();

    Set<String> getBefore();

    String getId();

    String getPhase();

}
```

56.2. 추상 인터셉터 클래스

phaseInterceptor 인터페이스를 직접 구현하는 대신 개발자는 **AbstractPhaseInterceptor** 클래스를 확장해야합니다. 이 추상 클래스는 **PhaseInterceptor** 인터페이스의 단계 관리 방법에 대한 구현을 제공합니다. **AbstractPhaseInterceptor** 클래스는 **handleFault()** 메서드의 기본 구현도 제공합니다.

개발자는 **handleMessage()** 메서드의 구현을 제공해야 합니다. 또한 **handleFault()** 메서드에 대해 다른 구현을 제공할 수 있습니다. 개발자 제공 구현은 일반 **org.apache.cxf.message.Message** 인터페이스에서 제공하는 방법을 사용하여 메시지 데이터를 조작할 수 있습니다.

SOAP 메시지와 함께 작동하는 애플리케이션의 경우 **Apache CXF**는 **AbstractSoapInterceptor** 클래스를 제공합니다. 이 클래스를 확장하면 **handleMessage()** 메서드와 메시지 데이터에 대한 액세스 권한이 있는 **handleFault()** 메서드를 **org.apache.cxf.binding.soap.SoapMessage** 오브젝트로 제공합니다. **SoapMessage** 오브젝트에는 메시지에서 **SOAP** 헤더, **SOAP** 봉투 및 기타 **SOAP** 메타데이터를 검색하는 방법이 있습니다.

57장. INTERCEPTOR가 언제 (S) 인턴트가 될 때 확인

초록

인터셉터는 단계로 구성됩니다. 인터셉터가 실행되는 단계는 액세스할 수 있는 메시지 데이터의 일부를 결정합니다. 인터셉터는 동일한 단계에서 다른 인터셉터와 관련된 위치를 결정할 수 있습니다. 인터셉터 단계 및 단계 내의 위치는 인터셉터의 생성자 논리의 일부로 설정됩니다.

57.1. 인터셉터 위치 지정

사용자 지정 인터셉터를 개발할 때 고려해야 할 첫 번째 사항은 인터셉터가 속한 메시지 처리 체인의 위치입니다. 개발자는 다음 두 가지 방법 중 하나로 메시지 처리 체인에서 인터셉터의 위치를 제어할 수 있습니다.

- 인터셉터의 단계 지정
- 단계 내 인터셉터 위치에 대한 제약 조건 지정

일반적으로 인터셉터의 위치를 지정하는 코드는 인터셉터 생성자에 배치됩니다. 이렇게 하면 런타임에서 인터셉터를 인스턴스화하고 애플리케이션 수준 코드에서 명시적인 작업 없이 인터셉터 체인에 적절한 위치를 배치할 수 있습니다.

57.2. 인터셉터의 단계 지정

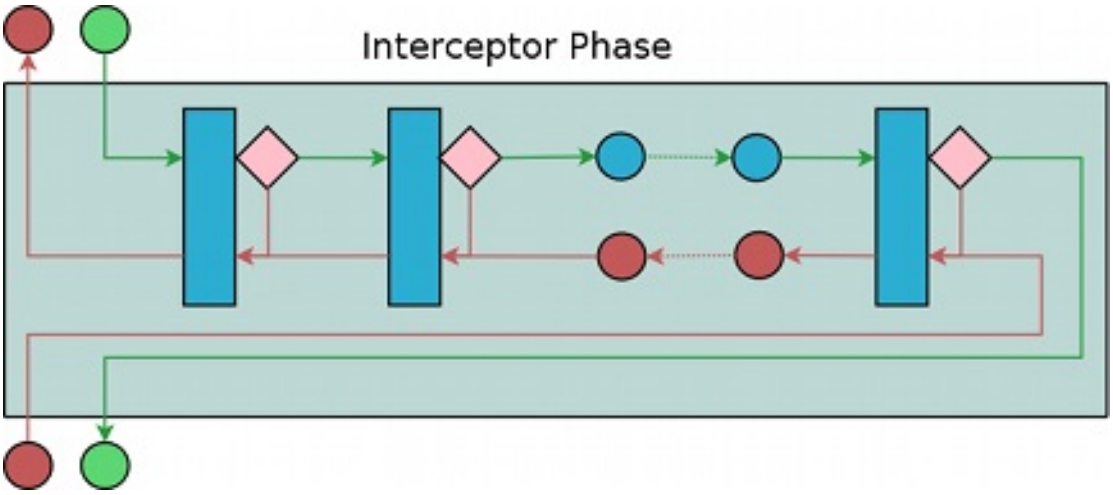
57.2.1. 개요

인터셉터는 단계로 구성됩니다. 인터셉터의 단계는 호출되는 메시지 처리 시퀀스의 시기를 결정합니다. 개발자는 인터셉터의 단계의 생성자를 지정합니다. 단계는 프레임워크에서 제공하는 상수 값을 사용하여 지정됩니다.

57.2.2. 단계

단계는 인터셉터의 논리 컬렉션입니다. 그림 57.1. “인터셉터 단계” 에 표시된 대로 단계 내의 인터셉터는 순차적으로 호출됩니다.

그림 57.1. 인터셉터 단계



단계는 주문 목록에 함께 연결되어 인터셉터 체인을 형성하고 메시지 처리 절차에 정의된 논리 단계를 제공합니다. 예를 들어 인바운드 인터셉터 체인의 **RECEIVE** 단계에 있는 인터셉터 그룹은 유선에서 선택한 원시 메시지 데이터를 사용하여 전송 수준 세부 정보를 처리합니다.

그러나 어떤 단계에서든 수행할 수 있는 작업은 적용되지 않습니다. 단계 내의 인터셉터는 단계의 정신에 있는 작업을 준수하는 것이 좋습니다.

Apache CXF에 의해 정의된 전체 단계 목록은 *62장. Apache CXF 메시지 처리 단계* 에서 확인할 수 있습니다.

57.2.3. 단계 지정

Apache CXF는 단계를 지정하는 데 사용할 **org.apache.cxf.Phase** 클래스를 제공합니다. 클래스는 상수의 컬렉션입니다.**The class is a collection of constants. Apache CXF**에 의해 정의된 각 단계는 단계 클래스에서 이에 해당하는 상수를 갖 습니다. 예를 들어 **RECEIVE** 단계는 **value Phase.RECEIVE**로 지정됩니다.

57.2.4. 단계 설정

인터셉터의 단계는 인터셉터 생성자에 설정됩니다. **AbstractPhaseInterceptor** 클래스는 인터셉터를 인스턴스화하기 위한 세 가지 생성자를 정의합니다.

- 공용 **AbstractPhaseInterceptor(String phase)**- 인터셉터의 단계를 지정된 단계로 설정하고 인터셉터의 **id**를 인터셉터의 클래스 이름으로 자동으로 설정합니다.

  이 생성자는 대부분의 사용 사례를 충족합니다.

- 공용 **AbstractPhaseInterceptor(문자열 id, String 단계)**- 인터셉터의 **id**를 첫 번째 매개 변수로 전달된 문자열로, 인터셉터의 단계를 두 번째 문자열로 설정합니다.

- 공용 **AbstractPhaseInterceptor(문자열 단계, 부울 uniqueId)**- 인터셉터가 생성된 고유 **id**를 사용해야 하는지 여부를 지정합니다. **uniqueId** 매개변수가 **true** 이면 인터셉터의 **ID**가 시스템에 의해 계산됩니다. **uniqueId** 매개변수가 **false** 이면 인터셉터의 **id**가 인터셉터의 클래스 이름으로 설정됩니다.

사용자 정의 인터셉터의 단계를 설정하는 권장 방법은 예 57.1. "인터셉터의 단계 설정" 에 표시된 **super()** 메서드를 사용하여 **AbstractPhaseInterceptor** 생성자에 단계를 전달하는 것입니다.

예 57.1. 인터셉터의 단계 설정

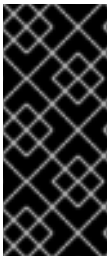
```
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
 {

    public StreamInterceptor()
   {
     super(Phase.PRE_STREAM);
    }
}
```

예 57.1. "인터셉터의 단계 설정" 에 표시된 **StreamInterceptor** 인터셉터는 **PRE_STREAM** 단계에 배치됩니다.

57.3. 단계에서 인터셉터 배치 구성

57.3.1. 개요

인터셉터를 단계에 배치하면 인터셉터가 제대로 작동하는지 확인하기 위해 배치를 충분히 제어할 수 없습니다. 예를 들어 **SAAJ API**를 사용하여 메시지의 **SOAP** 헤더를 검사해야 하는 인터셉터가 있는 경우 메시지를 **SAAJ** 오브젝트로 변환하는 인터셉터 다음에 실행해야 합니다. 한 인터셉터에서 다른 인터셉터에 필요한 메시지의 일부를 사용하는 경우가 있을 수 있습니다. 이 경우 개발자는 인터셉터 전에 실행해야 하는 인터셉터 목록을 제공할 수 있습니다. 개발자는 인터셉터 후 실행해야 하는 인터셉터 목록을 제공할 수도 있습니다.



중요

런타임은 인터셉터 단계 내에서 이러한 목록만 적용할 수 있습니다. 개발자가 현재 단계 이후에 실행해야 하는 인터셉터 목록에 이전 단계의 인터셉터를 배치하면 런타임에서 요청을 무시합니다.

57.3.2. 체인 앞에 추가

인터셉터를 개발할 때 발생하는 한 가지 문제는 인터셉터에 필요한 데이터가 항상 존재하지 않는다는 것입니다. 이 문제는 체인의 하나의 인터셉터가 이후 인터셉터에 필요한 메시지 데이터를 사용하는 경우 발생할 수 있습니다. 개발자는 인터셉터를 수정하여 사용자 정의 인터셉터를 제어하고 문제를 해결할 수 있습니다. 그러나 **Apache CXF**에서 여러 인터셉터를 사용하고 개발자가 수정할 수 없기 때문에 이 방법이 항상 가능한 것은 아닙니다.

대체 솔루션은 사용자 정의 인터셉터에 필요한 메시지 데이터를 사용할 인터셉터 전에 사용자 정의 인터셉터를 배치하는 것입니다. 가장 쉬운 방법은 이전 단계에서 배치하는 것이지만 항상 가능한 것은 아닙니다. **Apache CXF**의 **AbstractPhaseInterceptor** 클래스에서 두 개의 **addBefore()** 메서드를 하나 이상 제공하기 전에 인터셉터를 배치해야 하는 경우.

예 57.2. "다른 인터셉터 전에 인터셉터를 추가하는 방법" 에 표시된 대로 단일 인터셉터 **ID**를 사용하고 다른 하나는 인터셉터 **ID** 컬렉션을 가져옵니다. 여러 개의 호출을 수행하여 인터셉터를 계속 목록에 추가할 수 있습니다.

예 57.2. 다른 인터셉터 전에 인터셉터를 추가하는 방법

publicaddBeforeStringipublicaddBeforeCollection<String>i

예 57.3. "현재 인터셉터 후 실행해야 하는 인터셉터 목록 지정" 에 표시된 대로 개발자는 사용자 지정 인터셉터의 **constuctor**에서 **addBefore()** 메서드를 호출합니다.

예 57.3. 현재 인터셉터 후 실행해야 하는 인터셉터 목록 지정

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addBefore(HolderOutInterceptor.class.getName());
    }
```

```
...

}
```

대부분의 인터셉터는 인터셉터 ID에 클래스 이름을 사용합니다.

57.3.3. 다음으로 체인을 추가합니다.

인터셉터에 필요한 데이터가 존재하지 않는 또 다른 이유는 데이터가 메시지 오브젝트에 배치되지 않았기 때문입니다. 예를 들어 인터셉터는 메시지 데이터를 SOAP 메시지로 사용하여 작업하려고 할 수 있지만 메시지가 SOAP 메시지로 전환되기 전에 체인에 배치되면 작동하지 않습니다. 개발자는 인터셉터를 수정하여 사용자 정의 인터셉터를 제어하고 문제를 해결할 수 있습니다. 그러나 Apache CXF에서 여러 인터셉터를 사용하고 개발자가 수정할 수 없기 때문에 이 방법이 항상 가능한 것은 아닙니다.

대체 솔루션은 사용자 정의 인터셉터가 인터셉터 또는 인터셉터 뒤에 배치되었는지 확인하여 사용자 정의 인터셉터에 필요한 메시지를 생성하는 것입니다. 가장 쉬운 방법은 나중 단계에서 배치하는 것이지만 항상 가능한 것은 아닙니다. AbstractPhaseInterceptor 클래스는 인터셉터를 하나 이상의 다른 인터셉터 후에 배치해야 하는 경우를 위해 두 개의 addAfter() 메서드를 제공합니다.

예 57.4. "다른 인터셉터 후 인터셉터 추가 방법" 에 표시된 대로 한 가지 방법은 단일 인터셉터 ID를 사용하고 다른 방법은 인터셉터 ID 컬렉션을 가져옵니다. 여러 개의 호출을 수행하여 인터셉터를 계속 목록에 추가할 수 있습니다.

예 57.4. 다른 인터셉터 후 인터셉터 추가 방법

```
publicaddAfterStringipublicaddAfterCollection<String>i
```

예 57.5. "현재 인터셉터 전에 실행해야 하는 인터셉터 목록 지정" 에 표시된 대로 개발자는 사용자 지정 인터셉터의 constuctor에서 addAfter() 메서드를 호출합니다.

예 57.5. 현재 인터셉터 전에 실행해야 하는 인터셉터 목록 지정

```
public class MyPhasedOutInterceptor extends AbstractPhaseInterceptor
{

    public MyPhasedOutInterceptor() {
        super(Phase.PRE_LOGICAL);
        addAfter(StartingOutInterceptor.class.getName());
    }
```

```
...
}
```

대부분의 인터셉터는 인터셉터 ID에 클래스 이름을 사용합니다.

58장. 인터셉터 처리 논리 구현

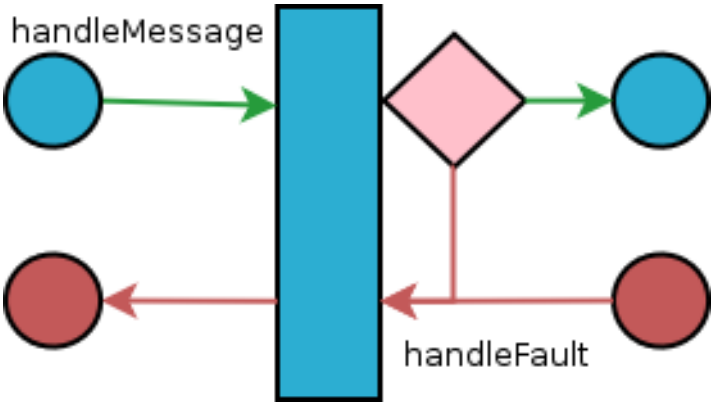
초록

인터셉터는 쉽게 구현할 수 있습니다. 대부분의 처리 논리는 **handleMessage()** 메서드에 있습니다. 이 방법은 메시지 데이터를 수신하고 필요에 따라 조작합니다. 개발자는 오류 처리 케이스를 처리하기 위해 몇 가지 특수 논리를 추가해야 할 수도 있습니다.

58.1. 인터셉터 흐름

그림 58.1. "인터셉터를 통한 흐름" 인터셉터를 통한 프로세스 흐름을 보여줍니다.

그림 58.1. 인터셉터를 통한 흐름



일반적인 메시지 처리에서는 **handleMessage()** 메서드만 호출됩니다. **handleMessage()** 메서드는 인터셉터의 메시지 처리 논리를 배치하는 곳입니다.

인터셉터의 **handleMessage()** 방법 또는 인터셉터 체인의 후속 인터셉터에서 오류가 발생하면 **handleFault()** 메서드가 호출됩니다. **handleFault()** 방법은 오류 발생 시 인터셉터 후에 정리하는 데 유용합니다. 또한 오류 메시지를 변경하는 데 사용할 수 있습니다.

58.2. 메시지 처리

58.2.1. 개요

일반적인 메시지 처리에서 인터셉터의 **handleMessage()** 메서드가 호출됩니다. 메시지 데이터를 **Message** 개체로 수신합니다. 메시지 개체는 메시지의 실제 내용과 함께 메시지 또는 메시지 처리 상태와

관련된 여러 속성을 포함할 수 있습니다. **Message** 오브젝트의 정확한 내용은 체인에서 현재 인터셉터 이전의 인터셉터에 따라 달라집니다.

58.2.2. 메시지 콘텐츠 가져오기

Message 인터페이스는 메시지 콘텐츠를 추출하는 데 사용할 수 있는 두 가지 방법을 제공합니다.

- **public <T> TgetContentjava.lang.Class<T > format getContent()** 메서드는 지정된 클래스의 개체에 메시지 내용을 반환합니다. 콘텐츠를 지정된 클래스의 인스턴스로 사용할 수 없는 경우 **null**이 반환됩니다.**If the contents are not available as an instance of the specified class, null is returned.** 사용 가능한 콘텐츠 유형 목록은 인터셉터 체인의 인터셉터 위치 및 인터셉터 체인의 방향에 따라 결정됩니다.

- **publicCollection<Attachment> getAttachments ()** 메서드는 메시지와 관련된 바이너리 첨부 파일이 포함된 **Java Collection** 오브젝트를 반환합니다. 첨부 파일은 **org.apache.cxf.message.Attachment** 오브젝트에 저장됩니다. **attachment** 개체는 바이너리 데이터를 관리하는 방법을 제공합니다.

중요

첨부 파일은 인터셉터 처리 후만 사용할 수 있습니다.

58.2.3. 메시지 방향 확인

메시지의 방향은 메시지 교환을 쿼리함으로써 결정될 수 있다. 메시지 교환은 인바운드 메시지와 아웃바운드 메시지를 별도의 속성에 저장합니다.[3]

메시지와 관련된 메시지 교환은 메시지의 **getExchange()** 메서드를 사용하여 검색됩니다. 예 58.1. “메시지 교환 받기” 에 표시된 대로 **getExchange()** 는 매개 변수를 사용하지 않고 메시지 교환을 **org.apache.cxf.message.Exchange** 오브젝트로 반환합니다.

> 예 58.1. 메시지 교환 받기
>
> **ExchangegetExchange**

Exchange 개체에는 교환과 관련된 메시지를 받기 위해 예 58.2. “메시지 교환에서 메시지 가져오기” 에 표시되는 네 가지 방법이 있습니다. 각 메서드는 메시지를 **org.apache.cxf.Message** 개체로 반환하거

나 메시지가 없는 경우 **null**을 반환합니다.

예 58.2. 메시지 교환에서 메시지 가져오기

getIn MessageMessagegetInFaultMessage메시지**getOutMessage**메시지**getOutFaultMessage**

예 58.3. "메시지 체인의 방향 확인" 현재 메시지가 아웃바운드인지 여부를 확인하는 코드를 표시합니다. 메서드는 메시지를 교환하고 현재 메시지가 교환의 아웃바운드 메시지와 동일한지 확인합니다. 또한 **exchange** 아웃바운드 **fault** 메시지와 아웃바운드 오류 메시지와 아웃바운드 **fault** 체인의 오류 메시지도 확인합니다.

예 58.3. 메시지 체인의 방향 확인

```
public static boolean isOutbound()
{
   Exchange exchange = message.getExchange();
       return message != null
             && exchange != null
             && (message == exchange.getOutMessage()
                || message == exchange.getOutFaultMessage());
}
```

58.2.4. 예제

예 58.4. "메시지 처리 방법 예" **zip** 압축 메시지를 처리하는 인터셉터의 코드를 표시합니다. 메시지의 방향을 확인한 다음 적절한 작업을 수행합니다.

예 58.4. 메시지 처리 방법 예

```
import java.io.IOException;
import java.io.InputStream;
import java.util.zip.GZIPInputStream;

import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class StreamInterceptor extends AbstractPhaseInterceptor<Message>
{

  ...

  public void handleMessage(Message message)
  {
```

```
        boolean isOutbound = false;
        isOutbound = message == message.getExchange().getOutMessage()
              || message == message.getExchange().getOutFaultMessage();

        if (!isOutbound)
        {
          try
          {
            InputStream is = message.getContent(InputStream.class);
            GZIPInputStream zipInput = new GZIPInputStream(is);
            message.setContent(InputStream.class, zipInput);
          }
          catch (IOException ioe)
          {
            ioe.printStackTrace();
          }
        }
         else
        {
         // zip the outbound message
        }
      }
    ...
  }
```

58.3. 오류 후 정리 취소

58.3.1. 개요

인터셉터 체인을 실행하는 동안 오류가 발생하면 런타임에서 인터셉터 체인을 트래버스하는 것을 중지하고 이미 실행된 체인의 인터셉터의 **handleFault()** 메서드를 호출하여 체인을 보류 해제합니다.

handleFault() 메서드를 사용하여 일반 메시지 처리 중에 인터셉터에서 사용하는 리소스를 정리할 수 있습니다. 또한 메시지 처리가 성공적으로 완료된 경우에만 대기해야 하는 작업을 롤백하는 데 사용할 수도 있습니다. 오류 메시지가 아웃바운드 오류 처리 인터셉터 체인에 전달될 경우 **handleFault()** 메서드를 사용하여 오류 메시지에 정보를 추가할 수도 있습니다.

58.3.2. 메시지 페이로드 가져오기

handleFault() 메서드는 일반 메시지 처리에 사용되는 **handleMessage()** 메서드와 동일한 **Message** 개체를 받습니다. **Message** 오브젝트에서 메시지 내용 가져오기는 "메시지 콘텐츠 가져오기" 에 설명되어 있습니다.

58.3.3. 예제

예 58.5. "보류 해제 인터셉터 체인 처리" 인터셉터 체인이 **unwound**인 경우 원래 **XML** 스트림이 메시지로 다시 배치되도록 하는 데 사용되는 코드를 표시합니다.

예 58.5. 보류 해제 인터셉터 체인 처리

```
@Override
public void handleFault(SoapMessage message)
{
  super.handleFault(message);
  XMLStreamWriter writer = (XMLStreamWriter)message.get(ORIGINAL_XML_WRITER);
  if (writer != null)
  {
    message.setContent(XMLStreamWriter.class, writer);
  }
}
```

---

[3] 또한 인바운드 및 아웃바운드 오류를 별도로 저장합니다.

59장. INTERCEPTORS를 사용하도록 끝점 구성

초록

인터셉터는 메시지 교환에 포함될 때 끝점에 추가됩니다. 끝점의 인터셉터 체인은 **Apache CXF** 런타임에 있는 여러 구성 요소의 인터셉터 체인에서 구성됩니다. 인터셉터는 끝점의 구성 또는 런타임 구성 요소 중 하나에 지정됩니다. 인터셉터는 구성 파일 또는 인터셉터 **API**를 사용하여 추가할 수 있습니다.

59.1. 인터셉터 연결 위치 결정

59.1.1. 개요

인터셉터 체인을 호스팅하는 여러 런타임 오브젝트가 있습니다. 여기에는 다음이 포함됩니다.

- **endpoint** 오브젝트
- **service** 오브젝트
- 프록시 오브젝트
- 끝점 또는 프록시를 생성하는 데 사용되는 팩토리 오브젝트
- 바인딩
- 중앙 버스 오브젝트

개발자는 자체 인터셉터를 이러한 오브젝트에 연결할 수 있습니다. 인터셉터를 연결하는 가장 일반적인 오브젝트는 버스와 개별 끝점입니다. 올바른 오브젝트를 선택하려면 끝점을 만들기 위해 이러한 런타임 오브젝트를 결합하는 방법을 이해해야 합니다. 디자인에 따라 각 **cxf** 관련 번들에는 자체 **cxf** 버스가 있습니다. 따라서 인터셉터가 버스에 구성되어 있고 동일한 블루프린트 컨텍스트의 서비스를 다른 번들로 가져오거나 생성하는 경우 인터셉터는 처리되지 않습니다. 대신 가져온 서비스의 **JAXWS** 클라이언트 또는 엔드포인트에 직접 인터셉터를 구성할 수 있습니다.

59.1.2. 끝점 및 프록시

끝점 또는 프록시에 인터셉터를 연결하는 것이 인터셉터를 배치하는 가장 정교한 방법입니다. 끝점 또는 프록시에 직접 연결된 인터셉터는 특정 끝점 또는 프록시에만 적용됩니다. 이는 특정 서비스 구현과 관련된 인터셉터를 연결하는 좋은 장소입니다. 예를 들어 개발자가 단위를 지표에서 **imperial**로 변환하는 서비스 인스턴스를 노출하려는 경우 인터셉터를 하나의 엔드포인트에 직접 연결할 수 있습니다.

59.1.3. 팩토리

Spring 구성을 사용하여 끝점 또는 프록시를 생성하는 데 사용되는 팩토리에 인터셉터를 연결하면 인터셉터를 끝점 또는 프록시에 직접 연결하는 것과 동일한 효과가 있습니다. 그러나 인터셉터가 공장에 프로그래밍 방식으로 연결된 경우 공장에 연결된 인터셉터는 공장에서 생성한 모든 엔드포인트 또는 프록시로 전파됩니다.

59.1.4. 바인딩

인터셉터를 바인딩에 연결하면 개발자가 바인딩을 사용하는 모든 끝점에 적용되는 인터셉터 세트를 지정할 수 있습니다. 예를 들어 개발자가 특수 **ID** 요소를 포함하도록 원시 **XML** 바인딩을 사용하는 모든 끝점을 강제 적용하려는 경우 요소를 **XML** 바인딩에 추가하는 인터셉터를 연결할 수 있습니다.

59.1.5. 버스

인터셉터를 연결하는 가장 일반적인 위치는 버스입니다. 인터셉터가 버스에 연결되면 인터셉터가 해당 버스에서 관리하는 모든 끝점으로 전파됩니다. 버스에 인터셉터를 연결하는 것은 유사한 인터셉터 세트를 공유하는 끝점을 여러 개 생성하는 애플리케이션에서 유용합니다.

59.1.6. 첨부 파일 지점 결합

끝점의 인터셉터 체인의 마지막 세트는 나열된 오브젝트에서 제공하는 인터셉터 체인의 모호함이므로 나열된 여러 오브젝트를 단일 엔드 포인트 구성으로 결합할 수 있습니다. 예를 들어 애플리케이션이 검증 토큰에 대해 확인된 인터셉터가 모두 필요한 여러 끝점을 생성하는 경우 해당 인터셉터가 애플리케이션의 버스에 연결됩니다. 해당 끝점 중 하나에 유로를 달러로 변환한 인터셉터도 필요한 경우 변환 인터셉터가 특정 엔드포인트에 직접 연결됩니다.

59.2. 구성을 사용하여 인터셉터 추가

59.2.1. 개요

인터셉터를 엔드포인트에 연결하는 가장 쉬운 방법은 구성 파일을 사용하는 것입니다. 끝점에 연결할 각 인터셉터는 표준 **Spring** 빈을 사용하여 구성됩니다. 그런 다음 인터셉터의 빈을 **Apache CXF** 구성 요

소를 사용하여 적절한 인터셉터 체인에 추가할 수 있습니다.

*연결된 인터셉터 체인이 있는 각 런타임 구성 요소는 특수 **Spring** 요소를 사용하여 구성할 수 있습니다. 각 구성 요소의 요소에는 인터셉터 체인을 지정하기 위한 표준 하위 집합이 있습니다. 구성 요소와 연결된 인터셉터 체인마다 하나의 자식이 있습니다. 하위 항목은 체인에 추가할 인터셉터의 빈을 나열합니다.*

59.2.2. 구성 요소

표 59.1. "인터셉터 체인 구성 요소" 인터셉터를 런타임 구성 요소에 연결하는 4가지 구성 요소를 설명합니다.

표 59.1. 인터셉터 체인 구성 요소

요소	설명
inInterceptors	끝점의 인바운드 인터셉터 체인에 추가할 인터셉터를 구성하는 빈 목록이 포함되어 있습니다.
outInterceptors	끝점의 아웃바운드 인터셉터 체인에 추가할 인터셉터를 구성하는 빈 목록이 포함되어 있습니다.
inFaultInterceptors	끝점의 인바운드 오류 처리 인터셉터 체인에 추가할 인터셉터를 구성하는 빈 목록이 포함되어 있습니다.
outFaultInterceptors	끝점의 아웃바운드 오류 처리 인터셉터 체인에 추가할 인터셉터를 구성하는 빈 목록이 포함되어 있습니다.

*모든 인터셉터 체인 구성 요소는 **list** 하위 요소를 사용합니다. **list** 요소에는 체인에 연결된 각 인터셉터에 대해 하나의 하위 항목이 있습니다. 인터셉터는 인터셉터를 직접 구성하는 빈 요소 또는 인터셉터를 구성하는 빈 요소를 참조하는 **ref** 요소를 사용하여 빈 요소를 사용하여 지정할 수 있습니다.*

59.2.3. 예제

예 59.1. "버스에 인터셉터 연결" 는 인터셉터를 버스의 인바운드 인터셉터 체인에 연결하는 구성을 보여줍니다.

예 59.1. 버스에 인터셉터 연결

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
```

```
        xsi:schemaLocation="
        http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"&gt;
  ...
  &lt;bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor"/&gt;

  &lt;cxf:bus&gt;
    *&lt;cxf:inInterceptors&gt;
      &lt;list&gt;
        &lt;ref bean="GZIPStream"/&gt;
      &lt;/list&gt;
    &lt;/cxf:inInterceptors&gt;*
  &lt;/cxf:bus&gt;
&lt;/beans&gt;
```

예 59.2. "JAX-WS 서비스 공급자에 인터셉터 연결" 는 인터셉터를 JAX-WS 서비스의 아웃바운드 인터셉터 체인에 연결하는 구성을 보여줍니다.

예 59.2. JAX-WS 서비스 공급자에 인터셉터 연결

```
&lt;beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jaxws="http://cxf.apache.org/jaxws"
        xmlns:wsa="http://cxf.apache.org/ws/addressing"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"&gt;

  &lt;jaxws:endpoint ...&gt;
    *&lt;jaxws:outInterceptors&gt;
      &lt;list&gt;
        &lt;bean id="GZIPStream" class="demo.stream.interceptor.StreamInterceptor" /&gt;
      &lt;/list&gt;
    &lt;/jaxws:outInterceptors&gt;*
  &lt;/jaxws:endpoint&gt;
&lt;/beans&gt;
```

59.2.4. 더 알아보기

Spring 구성을 사용하여 엔드포인트를 구성하는 방법에 대한 자세한 내용은 IV 부. 웹 서비스 엔드 포인트 구성 을 참조하십시오.

59.3. 프로그래밍 방식으로 인터셉터 추가

59.3.1. Interceptors 추가 방법

인터셉터는 다음 두 가지 방법 중 하나를 사용하여 프로그래밍 방식으로 끝점에 연결할 수 있습니다.

- **InterceptorProvider API**
- **Java** 주석

InterceptorProvider API를 사용하면 개발자가 인터셉터 체인이 있는 런타임 구성 요소에 인터셉터를 연결할 수 있지만 기본 **Apache CXF** 클래스로 작업해야 합니다. **Java** 주석은 서비스 인터페이스 또는 서비스 구현에만 추가할 수 있지만 개발자는 **JAX-WS API** 또는 **JAX-RS API** 내에 있을 수 있습니다.

59.3.2. 인터셉터 공급자 API 사용

59.3.2.1. 개요

인터셉터는 인터셉터 공급자 인터페이스 에 표시된 **InterceptorProvider** 인터페이스를 구현하는 모든 구성 요소로 등록할 수 있습니다.

인터셉터 공급자 인터페이스

```
package org.apache.cxf.interceptor;

import java.util.List;

public interface InterceptorProvider
{
    List<Interceptor<? extends Message>> getInInterceptors();

    List<Interceptor<? extends Message>> getOutInterceptors();

    List<Interceptor<? extends Message>> getInFaultInterceptors();

    List<Interceptor<? extends Message>> getOutFaultInterceptors();
}
```

*인터페이스의 네 가지 방법을 사용하면 각 엔드포인트의 인터셉터 체인을 **Java List** 오브젝트로 검색할 수 있습니다. 개발자는 **Java List** 오브젝트에서 제공하는 방법을 사용하여 모든 체인에 인터셉터를 추가하고 제거할 수 있습니다.*

59.3.2.2. 절차

***InterceptorProvider API**를 사용하여 런타임 구성 요소의 인터셉터 체인에 인터셉터를 연결하려면 다음을 수행해야 합니다.*

1. *인터셉터를 연결할 체인으로 런타임 구성 요소에 대한 액세스 권한을 가져옵니다.*

   *개발자는 **Apache CXF** 특정 **API**를 사용하여 표준 **Java** 애플리케이션 코드에서 런타임 구성 요소에 액세스해야 합니다. 일반적으로 런타임 구성 요소는 **JAX-WS** 또는 **JAX-RS** 아티팩트를 기본 **Apache CXF** 개체로 캐스팅하여 액세스할 수 있습니다.*

2. *인터셉터의 인스턴스를 생성합니다.*

3. *적절한 **get** 방법을 사용하여 원하는 인터셉터 체인을 검색합니다.*

4. ***List** 오브젝트의 **add()** 메서드를 사용하여 인터셉터를 인터셉터 체인에 연결합니다.*

   이 단계는 일반적으로 인터셉터 체인 검색과 결합됩니다.

59.3.2.3. 소비자에게 인터셉터 연결

*프로그래밍 방식으로 사용자에게 인터셉터 연결 **JAX-WS** 소비자의 인바운드 인터셉터 체인에 인터셉터를 연결하는 코드를 보여줍니다.*

프로그래밍 방식으로 사용자에게 인터셉터 연결

```
package com.fusesource.demo;

import java.io.File;
```

```
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.endpoint.Client;

public class Client
{
  public static void main(String args[])
  {
    QName serviceName = new QName("http://demo.eric.org", "stockQuoteReporter");
    Service s = Service.create(serviceName);

    QName portName = new QName("http://demo.eric.org", "stockQuoteReporterPort");
    s.addPort(portName, "http://schemas.xmlsoap.org/soap/", "http://localhost:9000/EricStockQuote");

    quoteReporter proxy = s.getPort(portName, quoteReporter.class);

    Client cxfClient = (Client) proxy;

    ValidateInterceptor validInterceptor = new ValidateInterceptor();
    cxfClient.getInInterceptor().add(validInterceptor);

    ...
  }
}
```

프로그래밍 방식으로 사용자에게 인터셉터 연결 의 코드는 다음을 수행합니다.

*소비자에 대한 **JAX-WS Service** 오브젝트를 만듭니다.*

*소비자의 대상 주소를 제공하는 **Service** 오브젝트에 포트를 추가합니다.*

서비스 공급자에서 메서드를 호출하는 데 사용되는 프록시를 생성합니다.

*프록시를 **org.apache.cxf.endpoint.Client** 유형으로 캐스팅합니다.*

인터셉터의 인스턴스를 생성합니다.

인바운드 인터셉터 체인에 인터셉터를 연결합니다.

59.3.2.4. 서비스 공급자에 인터셉터 연결

프로그래밍 방식으로 서비스 공급자에 인터셉터 연결 서비스 공급자의 아웃바운드 인터셉터 체인에 인터셉터를 연결하는 코드를 보여줍니다.

프로그래밍 방식으로 서비스 공급자에 인터셉터 연결

```
package com.fusesource.demo;
import java.util.*;

import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.frontend.EndpointImpl;

public class stockQuoteReporter implements quoteReporter
{
  ...
  public stockQuoteReporter()
  {
    ServerFactoryBean sfb = new ServerFactoryBean();
    Server server = sfb.create();
    EndpointImpl endpt = server.getEndpoint();

    AuthTokenInterceptor authInterceptor = new AuthTokenInterceptor();

    endpt.getOutInterceptor().add(authInterceptor);
  }
}
```

프로그래밍 방식으로 서비스 공급자에 인터셉터 연결 의 코드는 다음을 수행합니다.

기본 **Apache CXF** 개체에 대한 액세스를 제공할 **ServerFactoryBean** 개체를 만듭니다.

Apache CXF가 엔드포인트를 나타내는 데 사용하는 서버 개체를 가져옵니다.

*서비스 공급자의 **Apache CXF EndpointImpl** 오브젝트를 가져옵니다.*

인터셉터의 인스턴스를 생성합니다.

*인터셉터를 끝점에 연결합니다.**s** 아웃바운드 인터셉터 체인.*

59.3.2.5. 버스에 인터셉터 연결

버스에 인터셉터 연결 는 인터셉터를 버스의 인바운드 인터셉터 체인에 연결하는 코드를 보여줍니다.

버스에 인터셉터 연결

```
import org.apache.cxf.BusFactory;
org.apache.cxf.Bus;

...

Bus bus = BusFactory.getDefaultBus();

WatchInterceptor watchInterceptor = new WatchInterceptor();

bus..getInInterceptor().add(watchInterceptor);

...
```

버스에 인터셉터 연결 의 코드는 다음을 수행합니다.

런타임 인스턴스의 기본 버스를 가져옵니다.

인터셉터의 인스턴스를 생성합니다.

인바운드 인터셉터 체인에 인터셉터를 연결합니다.

WatchInterceptor는 런타임 인스턴스에서 생성된 모든 엔드포인트의 인바운드 인터셉터 체인에 연결됩니다.

59.3.3. Java 주석 사용

59.3.3.1. 개요

Apache CXF는 개발자가 엔드포인트에서 사용하는 인터셉터 체인을 지정할 수 있는 네 가지 Java 주석을 제공합니다. 인터셉터를 끝점에 연결하는 다른 방법과 달리 주석은 애플리케이션 수준 아티팩트에 연결됩니다. 사용되는 아티팩트는 주석의 적용 범위를 결정합니다.

59.3.3.2. 주석을 배치할 위치

주석은 다음 아티팩트에 배치할 수 있습니다.

- *끝점을 정의하는 서비스 끝점 인터페이스(SEI)*

  주석이 SEI에 배치되면 인터페이스를 구현하는 모든 서비스 공급자와 SEI를 사용하여 프록시를 생성하는 모든 소비자에 영향을 미칩니다.

- *서비스 구현 클래스*

  주석이 구현 클래스에 배치되면 구현 클래스를 사용하는 모든 서비스 공급자가 영향을 받습니다.

59.3.3.3. 주석

주석은 모두 org.apache.cxf.interceptor 패키지이며 표 59.2. "인터셉터 체인 주석" 에 설명되어 있습니다.

표 59.2. 인터셉터 체인 주석

주석	설명
InInterceptors	인바운드 인터셉터 체인의 인터셉터를 지정합니다.
OutInterceptors	아웃바운드 인터셉터 체인의 인터셉터를 지정합니다.

주석	설명
InFaultInterceptors	인바운드 fault 인터셉터 체인의 인터셉터를 지정합니다.
OutFaultInterceptors	아웃바운드 fault 인터셉터 체인의 인터셉터를 지정합니다.

59.3.3.4. 인터셉터 나열

인터셉터 목록은 체인 주석의 인터셉터 나열 구문 에 표시된 구문을 사용하여 정규화된 클래스 이름 목록으로 지정됩니다.

체인 주석의 인터셉터 나열 구문

```
interceptors={"interceptor1", "interceptor2", ..., "interceptorN"}
```

59.3.3.5. 예제

*서비스 구현에 인터셉터 연결 **SayHiImpl** 에서 제공하는 논리를 사용하는 엔드포인트의 인바운드 인터셉터 체인에 두 개의 인터셉터를 연결하는 주석을 표시합니다.*

서비스 구현에 인터셉터 연결

```
import org.apache.cxf.interceptor.InInterceptors;

@InInterceptors(interceptors={"com.sayhi.interceptors.FirstLast",
"com.sayhi.interceptors.LogName"})
public class SayHiImpl implements SayHi
{
  ...
}
```

60장. FLY에서 INTERCEPTOR CHAINS 조작

초록

인터셉터는 메시지 처리 논리의 일부로 끝점의 인터셉터 체인을 재구성할 수 있습니다. 새 인터셉터를 추가하고 인터셉터를 제거하고 인터셉터를 다시 정렬하고 인터셉터 체인을 일시 중단할 수도 있습니다. 모든 온-The-fly 조작은 호출 특정이므로 원래 체인은 끝점이 메시지 교환에 참여할 때마다 사용됩니다.

60.1. 개요

인터셉터 체인은 메시지가 생성을 스프링하는 동안만 살아 있습니다. 각 메시지에는 처리를 담당하는 인터셉터 체인에 대한 참조가 포함되어 있습니다. 개발자는 이 참조를 사용하여 메시지의 인터셉터 체인을 변경할 수 있습니다. 체인이 변경 당하므로 메시지의 인터셉터 체인에 대한 모든 변경 사항은 다른 메시지 교환에 영향을 미치지 않습니다.

60.2. 체인 라이프사이클

인터셉터 체인 및 체인의 인터셉터는 In-invocation 기준으로 인스턴스화됩니다. 메시지 교환에 참여하도록 끝점이 호출되면 필요한 인터셉터 체인이 인터셉터 인스턴스와 함께 인스턴스화됩니다. 인터셉터 체인 생성을 초래한 메시지 교환이 완료되면 체인과 해당 인터셉터 인스턴스가 삭제됩니다.

즉 인터셉터 체인 또는 인터셉터 필드의 변경 사항은 메시지 교환에서 유지되지 않습니다. 따라서 인터셉터가 활성 체인에 다른 인터셉터를 배치하면 활성 체인만 적용됩니다. 향후의 메시지 교환은 끝점 구성에 의해 결정된 대로 공수 상태에서 생성됩니다. 또한 개발자가 향후 메시지 처리를 변경하는 인터셉터에서 플래그를 설정할 수 없음을 의미합니다.

인터셉터에서 향후 인스턴스에 정보를 전달해야 하는 경우 메시지 컨텍스트에서 속성을 설정할 수 있습니다. 이 컨텍스트는 메시지 교환에 걸쳐 유지됩니다.

60.3. 인터셉터 체인 가져오기

메시지 인터셉터 체인 변경의 첫 번째 단계는 인터셉터 체인을 가져오는 것입니다. 이 작업은 예 60.1. “인터셉터 체인을 가져오는 방법” 에 표시된 Message.getInterceptorChain() 메서드를 사용하여 수행됩니다. 인터셉터 체인은 org.apache.cxf.interceptor.InterceptorChain 오브젝트로 반환됩니다.

> 예 60.1. 인터셉터 체인을 가져오는 방법
>
> InterceptorChaingetInterceptorChain

60.4. 인터셉터 추가

InterceptorChain 오브젝트에는 인터셉터 체인에 인터셉터를 추가하기 위해 예 60.2. "인터셉터 체인에 인터셉터 추가 방법" 에 표시되는 두 가지 방법이 있습니다. 하나는 단일 인터셉터를 추가할 수 있으며 다른 인터셉터를 사용하면 여러 인터셉터를 추가할 수 있습니다.

예 60.2. 인터셉터 체인에 인터셉터 추가 방법

```
Interceptor<? extends Message>i addCollection<Interceptor<? extends Message>>i
```

예 60.3. "on-the-fly 인터셉터 체인에 인터셉터 추가" 메시지의 인터셉터 체인에 단일 인터셉터를 추가하는 코드를 보여줍니다.

예 60.3. on-the-fly 인터셉터 체인에 인터셉터 추가

```
void handleMessage(Message message)
{
  ...
  AddledIntereptor addled = new AddledIntereptor();
  InterceptorChain chain = message.getInterceptorChain();
  chain.add(addled);
  ...
}
```

예 60.3. "on-the-fly 인터셉터 체인에 인터셉터 추가" 의 코드는 다음을 수행합니다.

체인에 추가할 인터셉터의 사본을 인스턴스화합니다.

중요

체인에 추가되는 인터셉터는 현재 인터셉터 또는 현재 인터셉터와 동일한 단계에 있어야 합니다.

현재 메시지의 인터셉터 체인을 가져옵니다.

체인에 새 인터셉터를 추가합니다.

60.5. 인터셉터 제거

InterceptorChain 오브젝트에는 인터셉터 체인에서 인터셉터를 제거하기 위해 예 60.4. "인터셉터 체인에서 인터셉터를 제거하는 방법"에 표시된 하나의 방법이 있습니다.

예 60.4. 인터셉터 체인에서 인터셉터를 제거하는 방법

```
Interceptor<? extends Message>i
```

예 60.5. "on-the-fly의 인터셉터 체인에서 인터셉터 제거" 메시지 인터셉터 체인에서 인터셉터를 제거하는 코드를 보여줍니다.

예 60.5. on-the-fly의 인터셉터 체인에서 인터셉터 제거

```
void handleMessage(Message message)
{
  ...
  Iterator<Interceptor<? extends Message>> iterator =
          message.getInterceptorChain().iterator();
  Interceptor<?> removeInterceptor =  null;
  for (; iterator.hasNext(); ) {
    Interceptor<?> interceptor = iterator.next();
    if (interceptor.getClass().getName().equals("InterceptorClassName")) {
      removeInterceptor = interceptor;
      break;
    }
  }

  if (removeInterceptor !=  null) {
    log.debug("Removing interceptor {}",removeInterceptor.getClass().getName());
    message.getInterceptorChain().remove(removeInterceptor);
  }
  ...
}
```

여기서 **InterceptorClassName** 은 체인에서 제거할 인터셉터의 클래스 이름입니다.

61장. JAX-RS 2.0 필터 및 인터셉터

초록

JAX-RS 2.0은 REST 호출을 위한 처리 파이프라인에 필터 및 인터셉터를 설치하기 위한 표준 API 및 의미 체계를 정의합니다. 필터 및 인터셉터는 일반적으로 로깅, 인증, 권한 부여, 메시지 압축, 메시지 암호화 등과 같은 기능을 제공하는 데 사용됩니다.

61.1. JAX-RS 필터 및 인터셉터 소개

61.1.1. 개요

이 섹션에서는 JAX-RS 필터 및 인터셉터의 처리 파이프라인에 대한 개요를 제공하여 필터 체인 또는 인터셉터 체인을 설치할 수 있는 확장 지점을 강조 표시합니다.

61.1.2. 필터

JAX-RS 2.0 필터 는 CXF 클라이언트 또는 서버를 통해 전달되는 모든 JAX-RS 메시지에 대한 개발자 액세스 권한을 부여하는 플러그인 유형입니다. 필터는 메시지(HTTP 헤더, 쿼리 매개 변수, 미디어 유형 및 기타 메타데이터)와 관련된 메타데이터를 처리하는 데 적합합니다. 필터는 메시지 호출을 중단할 수 있는 기능을 갖습니다(예: 보안 플러그인에 사용).

원하는 경우 각 확장 지점에 여러 개의 필터를 설치할 수 있습니다. 이 경우 필터가 체인에서 실행됩니다(단, 각 설치된 필터에 대한 우선순위 값을 지정하지 않는 한 실행 순서는 정의되지 않습니다).

61.1.3. 인터셉터

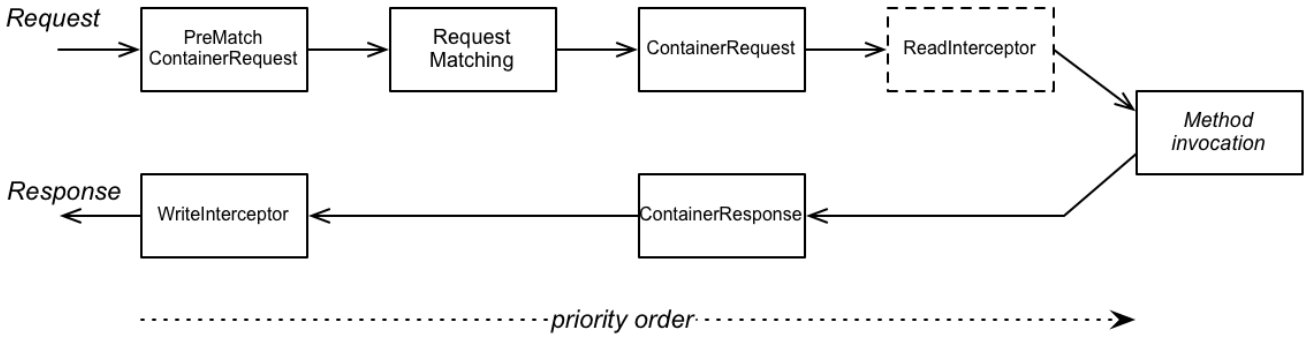
JAX-RS 2.0 인터셉터 는 메시지 본문에 대한 개발자 액세스 권한을 부여하는 플러그인 유형입니다. 인터셉터는 MessageBodyReader.readFrom 메서드 호출(reader 인터셉터의 경우) 또는 MessageBodyWriter.writeTo 메서드 호출(writer 인터셉터의 경우)을 중심으로 래핑됩니다.

원하는 경우 각 확장 포인트에 여러 인터셉터를 설치할 수 있습니다. 이 경우 인터셉터는 체인에서 실행됩니다(단, 각 설치된 인터셉터의 우선 순위 값을 지정하지 않는 한 실행 순서가 정의되지 않음).

61.1.4. 서버 처리 파이프라인

그림 61.1. “서버-Side Filter and Interceptor Extension Points” 서버 측에 설치된 JAX-RS 필터 및 인터셉터의 처리 파이프라인 개요를 보여줍니다.

그림 61.1. 서버-Side Filter and Interceptor Extension Points



61.1.5. 서버 확장 지점

서버 처리 파이프라인에서 다음 확장 포인트 중 하나에서 필터(또는 인터셉터)를 추가할 수 있습니다.

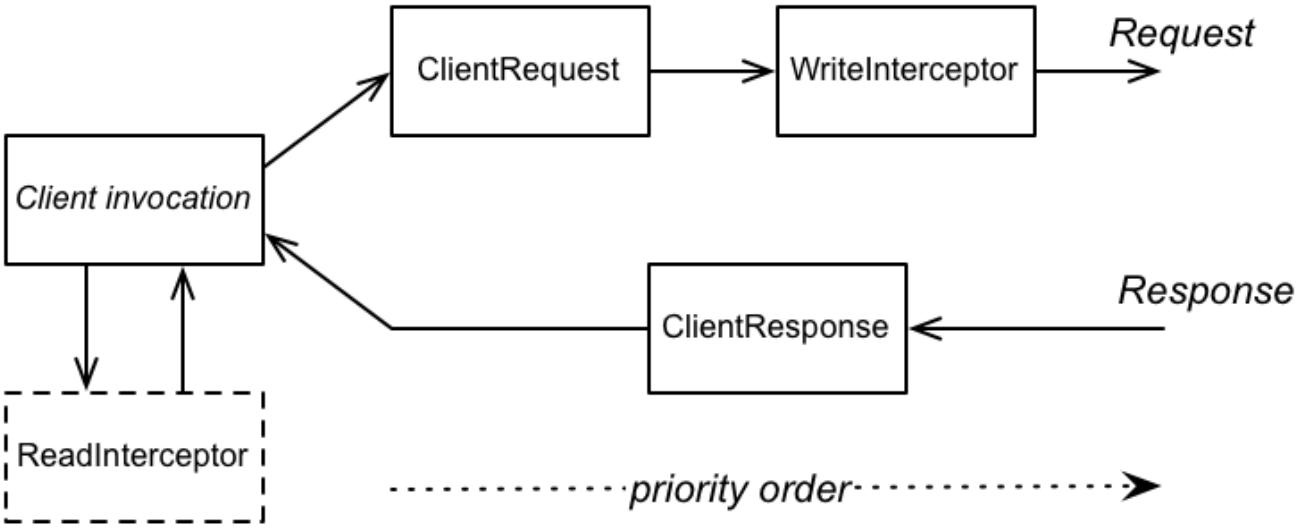
1. **PreMatchContainerRequest** 필터
2. **ContainerRequest** 필터
3. **ReadInterceptor**
4. **ContainerResponse** 필터
5. **WriteInterceptor**

리소스 일치가 발생 하기 전에 **PreMatchContainerRequest** 확장 지점에 도달하므로 이 시점에서 일부 컨텍스트 메타데이터를 사용할 수 없습니다.

61.1.6. 클라이언트 처리 파이프라인

그림 61.2. “클라이언트-Side Filter and Interceptor Extension Points” 클라이언트측에 설치된 **JAX-RS** 필터 및 인터셉터의 처리 파이프라인 개요를 보여줍니다.

그림 61.2. 클라이언트-Side Filter and Interceptor Extension Points



61.1.7. 클라이언트 확장 지점

클라이언트 처리 파이프라인에서 다음 확장 포인트 중 하나에서 필터(또는 인터셉터)를 추가할 수 있습니다.

1. **ClientRequest** 필터
2. **WriteInterceptor**
3. **ClientResponse** 필터
4. **ReadInterceptor**

61.1.8. 필터 및 인터셉터 순서

동일한 확장 지점에 여러 필터 또는 인터셉터를 설치하는 경우 필터 실행 순서는 할당된 우선 순위에 따라 달라집니다(**Java** 소스의 **@Priority** 주석 사용). 우선 순위는 정수 값으로 표시됩니다. 일반적으로 우선 순위가 높은 필터는 서버 측의 리소스 메서드 호출에 더 가까운 반면, 우선 순위가 더 낮은 필터는 클라이언트 호출에 더 가깝게 배치됩니다. 즉, 요청 메시지에 작동하는 필터 및 인터셉터는 우선순위 번호의 오름차순 으로 실행됩니다. 응답 메시지에 대한 필터 및 인터셉터는 우선 순위 번호의 내림차순으로 실행됩니다.

61.1.9. 필터 클래스

사용자 지정 **REST** 메시지 필터를 생성하기 위해 다음 **Java** 인터페이스를 구현할 수 있습니다.

- *javax.ws.rs.container.ContainerRequestFilter*
- *javax.ws.rs.container.ContainerResponseFilter*
- *javax.ws.rs.client.ClientRequestFilter*
- *javax.ws.rs.client.ClientResponseFilter*

61.1.10. 인터셉터 클래스

사용자 지정 **REST** 메시지 인터셉터를 생성하기 위해 다음 **Java** 인터페이스를 구현할 수 있습니다.

- *javax.ws.rs.ext.ReaderInterceptor*
- *javax.ws.rs.ext.WriterInterceptor*

61.2. 컨테이너 요청 필터

61.2.1. 개요

이 섹션에서는 서버(컨테이너) 측에서 들어오는 요청 메시지를 가로채는 데 사용되는 컨테이너 요청 필터 를 구현하고 등록하는 방법을 설명합니다. 컨테이너 요청 필터는 서버 측에서 헤더를 처리하는 데 자주 사용되며, 일반 요청 처리(즉, 호출된 특정 리소스 메서드와 독립적인 처리)에 사용할 수 있습니다.

또한 컨테이너 요청 필터는 두 개의 개별 확장 포인트인 **PreMatchContainerRequest** (리소스 일치 단계 이전) 및 **ContainerRequest** (리소스 일치 단계 이후)에 설치할 수 있기 때문에 특별한 케이스의 일부입니다.

61.2.2. ContainerRequestFilter 인터페이스

javax.ws.rs.container.ContainerRequestFilter 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerRequestFilter {
    public void filter(ContainerRequestContext requestContext) throws IOException;
}
```

ContainerRequestFilter 인터페이스를 구현하면 서버 측에서 다음 확장 포인트 중 하나에 대한 필터를 생성할 수 있습니다.

- **PreMatchContainerRequest**
- **ContainerRequest**

61.2.3. ContainerRequestContext interface

ContainerRequestFilter 의 필터 메서드는 들어오는 요청 메시지 및 관련 메타데이터에 액세스하는 데 사용할 수 있는 **javax.ws.rs.container.ContainerRequestContext** 의 단일 인수를 수신합니다. **ContainerRequestContext** 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.container;

import java.io.InputStream;
import java.net.URI;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.UriInfo;
```

```
public interface ContainerRequestContext {
    public Object getProperty(String name);
    public Collection getPropertyNames();
    public void setProperty(String name, Object object);
    public void removeProperty(String name);
    public UriInfo getUriInfo();
    public void setRequestUri(URI requestUri);
    public void setRequestUri(URI baseUri, URI requestUri);
    public Request getRequest();
    public String getMethod();
    public void setMethod(String method);
    public MultivaluedMap getHeaders();
    public String getHeaderString(String name);
    public Date getDate();
    public Locale getLanguage();
    public int getLength();
    public MediaType getMediaType();
    public List getAcceptableMediaTypes();
    public List getAcceptableLanguages();
    public Map getCookies();
    public boolean hasEntity();
    public InputStream getEntityStream();
    public void setEntityStream(InputStream input);
    public SecurityContext getSecurityContext();
    public void setSecurityContext(SecurityContext context);
    public void abortWith(Response response);
}
```

61.2.4. PreMatchContainerRequest 필터에 대한 샘플 구현

PreMatchContainerRequest 확장 지점에 대한 컨테이너 요청 필터(즉, 리소스 일치 전에 필터가 실행되는 경우) **ContainerRequestFilter** 인터페이스를 구현하는 클래스를 정의하려면 **@PreMatching** 주석(**PreMatchContainerRequest** 확장 지점 선택)으로 클래스에 주석을 답니다.

예를 들어 다음 코드는 우선순위가 **20**인 **PreMatchContainerContainerRequest** 확장 지점에 설치된 간단한 컨테이너 요청 필터의 예를 보여줍니다.

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.annotation.Priority;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SamplePreMatchContainerRequestFilter implements
  ContainerRequestFilter {

  public SamplePreMatchContainerRequestFilter() {
    System.out.println("SamplePreMatchContainerRequestFilter starting up");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    System.out.println("SamplePreMatchContainerRequestFilter.filter() invoked");
  }
}
```

61.2.5. ContainerRequest 필터에 대한 샘플 구현

ContainerRequest 확장 포인트(즉, 리소스 일치 후 필터가 실행되는 컨테이너 요청 필터)를 구현하려면 **@PreMatching** 주석 없이 **ContainerRequestFilter** 인터페이스 를 구현하는 클래스를 정의합니다.

예를 들어 다음 코드는 우선순위가 **30**인 **ContainerRequest** 확장 지점에 설치된 간단한 컨테이너 요청 필터의 예를 보여줍니다.

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;
```

```
@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

  public SampleContainerRequestFilter() {
    System.out.println("SampleContainerRequestFilter starting up");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    System.out.println("SampleContainerRequestFilter.filter() invoked");
  }
}
```

61.2.6. ResourceInfo 삽입

ContainerRequest 확장 포인트 (즉, 리소스 일치가 발생한 후) **ResourceInfo** 클래스를 삽입하여 일치하는 리소스 클래스 및 리소스 메서드에 액세스할 수 있습니다. 예를 들어 다음 코드는 **ResourceInfo** 클래스를 **ContainerRequestFilter** 클래스의 필드로 삽입하는 방법을 보여줍니다.

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;
import javax.ws.rs.core.Context;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {

  @Context
  private ResourceInfo resinfo;

  public SampleContainerRequestFilter() {
    ...
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    String resourceClass = resinfo.getResourceClass().getName();
    String methodName    = resinfo.getResourceMethod().getName();
    System.out.println("REST invocation bound to resource class: " + resourceClass);
    System.out.println("REST invocation bound to resource method: " + methodName);
  }
}
```

61.2.7. 호출 중지

컨테이너 요청 필터의 적합한 구현을 생성하여 서버 측 호출을 중단할 수 있습니다. 일반적으로 이 기능은 서버 측에서 보안 기능을 구현하는 데 유용합니다. 예를 들어 인증 기능 또는 권한 부여 기능을 구현합니다. 들어오는 요청이 성공적으로 인증되지 않으면 컨테이너 요청 필터 내에서 호출을 중단할 수 있습니다.

*예를 들어 다음 사전 일치 기능은 **URI**의 쿼리 매개 변수에서 사용자 이름과 암호를 추출하고 인증 방법을 호출하여 사용자 이름과 암호 자격 증명을 확인합니다. 인증에 실패하면 **ContainerRequestContext** 개체에서 **abortWith** 를 호출하여 호출이 중단되고 클라이언트에 반환되는 오류 응답을 전달합니다.*

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.ext.Provider;

@PreMatching
@Priority(value = 20)
@Provider
public class SampleAuthenticationRequestFilter implements
  ContainerRequestFilter {

  public SampleAuthenticationRequestFilter() {
    System.out.println("SampleAuthenticationRequestFilter starting up");
  }

  @Override
  public void filter(ContainerRequestContext requestContext) {
    ResponseBuilder responseBuilder = null;
    Response response = null;

    String userName = requestContext.getUriInfo().getQueryParameters().getFirst("UserName");
    String password = requestContext.getUriInfo().getQueryParameters().getFirst("Password");
    if (authenticate(userName, password) == false) {
      responseBuilder = Response.serverError();
      response = responseBuilder.status(Status.BAD_REQUEST).build();
      requestContext.abortWith(response);
    }
  }

  public boolean authenticate(String userName, String password) {
    // Perform authentication of 'user'
    ...
  }
}
```

61.2.8. 서버 요청 필터 바인딩

서버 요청 필터(즉, **Apache CXF** 런타임에 설치)를 바인딩 하려면 다음 단계를 수행합니다.

1. 다음 코드 조각에 표시된 대로 컨테이너 요청 필터 클래스에 **@Provider** 주석을 추가합니다.

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.ext.Provider;
import javax.annotation.Priority;

@Provider
@Priority(value = 30)
public class SampleContainerRequestFilter implements ContainerRequestFilter {
  ...
}
```

컨테이너 요청 필터 구현이 **Apache CXF** 런타임에 로드되면 **REST** 구현에서 로드된 클래스를 자동으로 검사하여 **@Provider** 주석(검색 단계)으로 표시된 클래스를 검색합니다.

2. **XML**에서 **JAX-RS** 서버 엔드포인트를 정의할 때(예: 18.1절. “JAX-RS Server 엔드 포인트 구성”참조), **jaxrs:providers** 요소의 공급자 목록에 **server request** 필터를 추가합니다.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
      <jaxrs:providers>
        <ref bean="filterProvider" />
      </jaxrs:providers>
      <bean id="filterProvider"
class="org.jboss.fuse.example.SampleContainerRequestFilter"/>

    </jaxrs:server>

</blueprint>
```

참고

이 단계는 **Apache CXF**의 비표준 요구 사항입니다. **JAX-RS** 표준에 따라 엄밀히 말하면 **@Provider** 주석은 필터를 바인딩하는 데 필요한 모든 것이어야 합니다. 그러나 실제로는 표준 접근 방식은 다소 무독하며 많은 라이브러리가 대규모 프로젝트에 포함될 때 번들링 공급자로 이어질 수 있습니다.

61.3. 컨테이너 응답 필터

61.3.1. 개요

이 섹션에서는 서버 측에서 나가는 응답 메시지를 가로채는 데 사용되는 컨테이너 응답 필터 를 구현하고 등록하는 방법을 설명합니다. 컨테이너 응답 필터는 응답 메시지에 헤더를 자동으로 채우는 데 사용할 수 있으며 일반적으로 모든 종류의 일반 응답 필터에 사용할 수 있습니다.

61.3.2. ContainerResponseFilter 인터페이스

javax.ws.rs.container.ContainerResponseFilter 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.container;

import java.io.IOException;

public interface ContainerResponseFilter {
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext)
            throws IOException;
}
```

ContainerResponseFilter 를 구현하면 서버 측의 **ContainerResponse** 확장 포인트에 대한 필터를 생성하여 호출 후 응답 메시지를 필터링할 수 있습니다.

참고

컨테이너 응답 필터는 요청 메시지(**requestContext** 인수를 통해) 및 응답 메시지(**responseContext** 메시지를 통해) 모두에 액세스할 수 있지만 이 단계에서 응답만 수정할 수 있습니다.

61.3.3. ContainerResponseContext 인터페이스

ContainerResponseFilter 의 필터 방법은 javax.ws.rs.container.ContainerRequestContext 유형의 인수와 유형 javax.ws.rs.container.ContainerResponseContext 의 인수이며, 발신 응답 메시지 및 관련 메타데이터에 액세스하는 데 사용할 수 있는 두 가지 인수를 수신합니다.
"ContainerRequestContext interface"

ContainerResponseContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.container;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ContainerResponseContext {

    public int getStatus();

    public void setStatus(int code);

    public Response.StatusType getStatusInfo();

    public void setStatusInfo(Response.StatusType statusInfo);

    public MultivaluedMap<String, Object> getHeaders();

    public abstract MultivaluedMap<String, String> getStringHeaders();

    public String getHeaderString(String name);

    public Set<String> getAllowedMethods();

    public Date getDate();

    public Locale getLanguage();

    public int getLength();

    public MediaType getMediaType();
```

```
    public Map<String, NewCookie> getCookies();

    public EntityTag getEntityTag();

    public Date getLastModified();

    public URI getLocation();

    public Set<Link> getLinks();

    boolean hasLink(String relation);

    public Link getLink(String relation);

    public Link.Builder getLinkBuilder(String relation);

    public boolean hasEntity();

    public Object getEntity();

    public Class<?> getEntityClass();

    public Type getEntityType();

    public void setEntity(final Object entity);

    public void setEntity(
            final Object entity,
            final Annotation[] annotations,
            final MediaType mediaType);

    public Annotation[] getEntityAnnotations();

    public OutputStream getEntityStream();

    public void setEntityStream(OutputStream outputStream);
}
```

61.3.4. 샘플 구현

ContainerResponse 확장 포인트(즉, 서버 측에서 호출이 실행된 후 필터가 실행되는 컨테이너 응답 필터)를 구현하려면 **ContainerResponseFilter** 인터페이스를 구현하는 클래스를 정의합니다.

예를 들어 다음 코드는 우선 순위가 **10**인 **ContainerResponse** 확장 지점에 설치된 간단한 컨테이너 응답 필터의 예를 보여줍니다.

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
```

```
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {

  public SampleContainerResponseFilter() {
    System.out.println("SampleContainerResponseFilter starting up");
  }

  @Override
  public void filter(
      ContainerRequestContext requestContext,
      ContainerResponseContext responseContext
  )
  {
    // This filter replaces the response message body with a fixed string
    if (responseContext.hasEntity()) {
      responseContext.setEntity("New message body!");
    }
  }
}
```

61.3.5. 서버 응답 필터 바인딩

서버 응답 필터(즉, **Apache CXF** 런타임에 설치)를 바인딩 하려면 다음 단계를 수행합니다.

1. 다음 코드 조각과 같이 컨테이너 응답 필터 클래스에 **@Provider** 주석을 추가합니다.

```
// Java
package org.jboss.fuse.example;

import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;

@Provider
@Priority(value = 10)
public class SampleContainerResponseFilter implements ContainerResponseFilter {
  ...
}
```

컨테이너 응답 필터 구현을 **Apache CXF** 런타임에 로드하면 **REST** 구현에서 로드된 클래스를 자동으로 검사하여 **@Provider** 주석(검색 단계)으로 표시된 클래스를 검색합니다.

2. *XML에서 JAX-RS 서버 엔드포인트를 정의할 때(예: 18.1절. "JAX-RS Server 엔드 포인트 구성"참조), jaxrs:providers 요소의 공급자 목록에 서버 응답 필터를 추가합니다.*

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
   xmlns:cxf="http://cxf.apache.org/blueprint/core"
   ...
>
   ...
   <jaxrs:server id="customerService" address="/customers">
     ...
     <jaxrs:providers>
       <ref bean="filterProvider" />
     </jaxrs:providers>
     <bean id="filterProvider"
class="org.jboss.fuse.example.SampleContainerResponseFilter"/>

   </jaxrs:server>

</blueprint>
```

참고

이 단계는 Apache CXF의 비표준 요구 사항입니다. JAX-RS 표준에 따라 엄밀히 말하면 @Provider 주석은 필터를 바인딩하는 데 필요한 모든 것이어야 합니다. 그러나 실제로는 표준 접근 방식은 다소 무독하며 많은 라이브러리가 대규모 프로젝트에 포함될 때 번들링 공급자로 이어질 수 있습니다.

61.4. 클라이언트 요청 필터

61.4.1. 개요

이 섹션에서는 클라이언트 측에서 나가는 요청 메시지를 가로채는 데 사용되는 클라이언트 요청 필터를 구현하고 등록하는 방법을 설명합니다. 클라이언트 요청 필터는 종종 헤더를 처리하는 데 사용되며 모든 종류의 일반 요청 처리에 사용할 수 있습니다.

61.4.2. ClientRequestFilter 인터페이스

javax.ws.rs.client.ClientRequestFilter 인터페이스는 다음과 같이 정의됩니다.

```
// Java
package javax.ws.rs.client;
...
import javax.ws.rs.client.ClientRequestFilter;
```

```
import javax.ws.rs.client.ClientRequestContext;
...
public interface ClientRequestFilter {
    void filter(ClientRequestContext requestContext) throws IOException;
}
```

ClientRequestFilter 를 구현하면 클라이언트 측에서 **ClientRequest** 확장 포인트에 대한 필터를 생성하여 메시지를 서버에 보내기 전에 요청 메시지를 필터링할 수 있습니다.

61.4.3. ClientRequestContext 인터페이스

ClientRequestFilter 의 필터 메서드는 나가는 요청 메시지 및 관련 메타데이터에 액세스하는 데 사용할 수 있는 **javax.ws.rs.client.ClientRequestContext** 의 단일 인수를 수신합니다.
ClientRequestContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.client;

import java.io.OutputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.net.URI;
import java.util.Collection;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.Map;

import javax.ws.rs.core.Configuration;
import javax.ws.rs.core.Cookie;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.MessageBodyWriter;

public interface ClientRequestContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();

    public void setProperty(String name, Object object);

    public void removeProperty(String name);

    public URI getUri();

    public void setUri(URI uri);

    public String getMethod();
```

```
    public void setMethod(String method);

    public MultivaluedMap<String, Object> getHeaders();

    public abstract MultivaluedMap<String, String> getStringHeaders();

    public String getHeaderString(String name);

    public Date getDate();

    public Locale getLanguage();

    public MediaType getMediaType();

    public List<MediaType> getAcceptableMediaTypes();

    public List<Locale> getAcceptableLanguages();

    public Map<String, Cookie> getCookies();

    public boolean hasEntity();

    public Object getEntity();

    public Class<?> getEntityClass();

    public Type getEntityType();

    public void setEntity(final Object entity);

    public void setEntity(
            final Object entity,
            final Annotation[] annotations,
            final MediaType mediaType);

    public Annotation[] getEntityAnnotations();

    public OutputStream getEntityStream();

    public void setEntityStream(OutputStream outputStream);

    public Client getClient();

    public Configuration getConfiguration();

    public void abortWith(Response response);
}
```

61.4.4. 샘플 구현

***ClientRequest** 확장 포인트(즉, 요청 메시지를 보내기 전에 필터가 실행되는) 클라이언트 요청 필터를 구현하려면 **ClientRequestFilter** 인터페이스를 구현하는 클래스를 정의합니다.*

*예를 들어 다음 코드는 우선 순위가 **20**인 **ClientRequest** 확장 지점에 설치된 간단한 클라이언트 요청 필터의 예를 보여줍니다.*

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.annotation.Priority;

@Priority(value = 20)
public class SampleClientRequestFilter implements ClientRequestFilter {

  public SampleClientRequestFilter() {
    System.out.println("SampleClientRequestFilter starting up");
  }

  @Override
  public void filter(ClientRequestContext requestContext) {
    System.out.println("ClientRequestFilter.filter() invoked");
  }
}
```

61.4.5. 호출 중지

적합한 클라이언트 요청 필터를 구현하여 클라이언트 측 호출을 중단할 수 있습니다. 예를 들어 클라이언트 측 필터를 구현하여 요청이 올바르게 포맷되는지 확인하고 필요한 경우 요청을 중단할 수 있습니다.

*다음 테스트 코드는 항상 요청을 중단하고 **BAD_REQUEST HTTP** 상태를 클라이언트 호출 코드로 반환합니다.*

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import javax.annotation.Priority;

@Priority(value = 10)
public class TestAbortClientRequestFilter implements ClientRequestFilter {

  public TestAbortClientRequestFilter() {
    System.out.println("TestAbortClientRequestFilter starting up");
  }

  @Override
  public void filter(ClientRequestContext requestContext) {
```

```
    // Test filter: aborts with BAD_REQUEST status
    requestContext.abortWith(Response.status(Status.BAD_REQUEST).build());
  }
}
```

61.4.6. 클라이언트 요청 필터 등록

JAX-RS 2.0 클라이언트 **API**를 사용하여 **javax.ws.rs.client.Client** 오브젝트 또는 **javax.ws.client.WebTarget** 오브젝트에 클라이언트 요청 필터를 직접 등록할 수 있습니다. 효과적으로는 클라이언트 요청 필터를 선택적으로 다른 범위에 적용할 수 있으므로 필터의 특정 **URI** 경로만 영향을 받습니다.

예를 들어 다음 코드는 **SampleClientRequestFilter** 필터를 등록하여 클라이언트 오브젝트를 사용하여 수행된 모든 호출에 적용되는 방법 및 **TestAbortClientRequestFilter** 필터를 등록하는 방법을 보여주므로 **rest/TestAbortClientRequest RequestRequestRequestRequest**의 하위 경로만 적용합니다.

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientRequestFilter());
WebTarget target = client
  .target("http://localhost:8001/rest/TestAbortClientRequest");
target.register(new TestAbortClientRequestFilter());
```

61.5. 클라이언트 응답 필터

61.5.1. 개요

이 섹션에서는 클라이언트 측에서 들어오는 응답 메시지를 가로채는 데 사용되는 클라이언트 응답 필터 를 구현하고 등록하는 방법을 설명합니다. 클라이언트 응답 필터는 클라이언트 측의 모든 종류의 일반 응답 처리에 사용할 수 있습니다.

61.5.2. ClientResponseFilter 인터페이스

javax.ws.rs.client.ClientResponseFilter 인터페이스는 다음과 같이 정의됩니다.

```
// Java
package javax.ws.rs.client;
...
import java.io.IOException;
```

```
public interface ClientResponseFilter {
    void filter(ClientRequestContext requestContext, ClientResponseContext responseContext)
            throws IOException;
}
```

ClientResponseFilter 를 구현하면 클라이언트 측에서 **ClientResponse** 확장 포인트에 대한 필터를 생성하여 서버에서 수신한 후 응답 메시지를 필터링할 수 있습니다.

61.5.3. ClientResponseContext 인터페이스

ClientResponseFilter 의 필터 방법은 **javax.ws.rs.client.ClientRequestContext** 유형의 인수와 **javax.ws.rs.client.ClientResponseContext** 형식의 인수와 발신 응답 메시지 및 관련 메타데이터에 액세스하는 데 사용할 수 있는 두 가지 인수를 수신합니다. "ClientRequestContext 인터페이스"

ClientResponseContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.client;

import java.io.InputStream;
import java.net.URI;
import java.util.Date;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import javax.ws.rs.core.EntityTag;
import javax.ws.rs.core.Link;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;

public interface ClientResponseContext {

    public int getStatus();

    public void setStatus(int code);

    public Response.StatusType getStatusInfo();

    public void setStatusInfo(Response.StatusType statusInfo);

    public MultivaluedMap<String, String> getHeaders();

    public String getHeaderString(String name);
```

```
    public Set<String> getAllowedMethods();

    public Date getDate();

    public Locale getLanguage();

    public int getLength();

    public MediaType getMediaType();

    public Map<String, NewCookie> getCookies();

    public EntityTag getEntityTag();

    public Date getLastModified();

    public URI getLocation();

    public Set<Link> getLinks();

    boolean hasLink(String relation);

    public Link getLink(String relation);

    public Link.Builder getLinkBuilder(String relation);

    public boolean hasEntity();

    public InputStream getEntityStream();

    public void setEntityStream(InputStream input);
}
```

61.5.4. 샘플 구현

ClientResponse 확장 포인트(즉, 서버에서 응답 메시지를 수신한 후 필터가 실행되는 클라이언트 응답 필터)를 구현하려면 **ClientResponseFilter** 인터페이스를 구현하는 클래스를 정의합니다.

예를 들어 다음 코드는 **priority**가 **20**인 **ClientResponse** 확장 지점에 설치된 간단한 클라이언트 응답 필터의 예를 보여줍니다.

```
// Java
package org.jboss.fuse.example;

import javax.ws.rs.client.ClientRequestContext;
import javax.ws.rs.client.ClientResponseContext;
import javax.ws.rs.client.ClientResponseFilter;
import javax.annotation.Priority;

@Priority(value = 20)
public class SampleClientResponseFilter implements ClientResponseFilter {
```

```
  public SampleClientResponseFilter() {
    System.out.println("SampleClientResponseFilter starting up");
  }

  @Override
  public void filter(
      ClientRequestContext requestContext,
      ClientResponseContext responseContext
  )
  {
    // Add an extra header on the response
    responseContext.getHeaders().putSingle("MyCustomHeader", "my custom data");
  }
}
```

61.5.5. 클라이언트 응답 필터 등록

JAX-RS 2.0 클라이언트 **API**를 사용하여 **javax.ws.rs.client.Client** 오브젝트 또는 **javax.ws.client.WebTarget** 오브젝트에 클라이언트 응답 필터를 직접 등록할 수 있습니다. 효과적으로는 클라이언트 요청 필터를 선택적으로 다른 범위에 적용할 수 있으므로 필터의 특정 **URI** 경로만 영향을 받습니다.

예를 들어 다음 코드는 **SampleClientResponseFilter** 필터를 등록하는 방법을 보여주므로 **client** 개체를 사용하여 수행된 모든 호출에 적용됩니다.

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(new SampleClientResponseFilter());
```

61.6. 엔터티 리더 인터셉터

61.6.1. 개요

이 섹션에서는 클라이언트 측 또는 서버 측에서 메시지 본문을 읽을 때 입력 스트림을 가로채는 데 사용할 수 있는 엔터티 리더 인터셉터 를 구현하고 등록하는 방법을 설명합니다. 이는 일반적으로 암호화 및 암호 해독 또는 압축 해제와 같은 요청 본문의 일반적인 변환에 유용합니다.

61.6.2. ReaderInterceptor 인터페이스

javax.ws.rs.ext.ReaderInterceptor 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.ext;

public interface ReaderInterceptor {
    public Object aroundReadFrom(ReaderInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

reader Interceptor 인터페이스를 구현하면 서버 측 또는 클라이언트 측에서 읽는 중에 메시지 본문 **(Entity object)**을 가로챌 수 있습니다. 다음 컨텍스트 중 하나에서 엔터티 리더 인터셉터를 사용할 수 있습니다.

- 서버 측인터셉터로 바인딩된 경우 엔터티 판독기 인터셉터는 애플리케이션 코드(일치된 리소스의)에서 액세스될 때 요청 메시지 본문을 인터셉트합니다. **REST** 요청의 의미 체계에 따라 일치된 리소스에서 메시지 본문에 액세스하지 못할 수 있습니다. 이 경우 **reader** 인터셉터는 호출되지 않습니다.

- 클라이언트 쪽인터셉터로 바인딩된 경우 엔터티 판독기 인터셉터는 클라이언트 코드에서 액세스될 때 응답 메시지 본문을 인터셉트합니다. 클라이언트 코드가 응답 메시지에 명시적으로 액세스하지 않는 경우(예: **Response.getEntity** 메서드를 호출하여) **reader** 인터셉터가 호출되지 않습니다.

61.6.3. ReaderInterceptorContext 인터페이스

readerInterceptor의 **aroundReadFrom** 방법은 메시지 본문**(Entity object)** 및 메시지 메타데이터 모두에 액세스하는 데 사용할 수 있는 **javax.ws.rs.ext. ReaderInterceptor Context** 유형의 하나의 인수를 수신합니다.

reader InterceptorContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;
```

```
public interface ReaderInterceptorContext extends InterceptorContext {

    public Object proceed() throws IOException, WebApplicationException;

    public InputStream getInputStream();

    public void setInputStream(InputStream is);

    public MultivaluedMap<String, String> getHeaders();
}
```

61.6.4. InterceptorContext 인터페이스

ReaderInterceptorContext 인터페이스는 기본 **InterceptorContext** 인터페이스에서 상속된 메서드도 지원합니다.

InterceptorContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.ext;

import java.lang.annotation.Annotation;
import java.lang.reflect.Type;
import java.util.Collection;

import javax.ws.rs.core.MediaType;

public interface InterceptorContext {

    public Object getProperty(String name);

    public Collection<String> getPropertyNames();

    public void setProperty(String name, Object object);

    public void removeProperty(String name);

    public Annotation[] getAnnotations();

    public void setAnnotations(Annotation[] annotations);

    Class<?> getType();

    public void setType(Class<?> type);

    Type getGenericType();

    public void setGenericType(Type genericType);

    public MediaType getMediaType();
```

```
    public void setMediaType(MediaType mediaType);
}
```

61.6.5. 클라이언트 측에서의 샘플 구현

클라이언트 측에 대해 엔터티 리더 인터셉터를 구현하려면 **ReaderInterceptor** 인터페이스를 구현하는 클래스를 정의합니다.

예를 들어 다음 코드는 클라이언트 측의 엔티티 리더 인터셉터의 예를 보여줍니다(우선 **10** 우선 순위 **10**임)는 들어오는 응답의 메시지 본문에서 **company _NAME** 의 모든 인스턴스를 **Red Hat** 으로 대체합니다.

```
// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
public class SampleClientReaderInterceptor implements ReaderInterceptor {

  @Override
  public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
        throws IOException, WebApplicationException
  {
    InputStream inputStream = interceptorContext.getInputStream();
    byte[] bytes = new byte[inputStream.available()];
    inputStream.read(bytes);
    String responseContent = new String(bytes);
    responseContent = responseContent.replaceAll("COMPANY_NAME", "Red Hat");
    interceptorContext.setInputStream(new ByteArrayInputStream(responseContent.getBytes()));

    return interceptorContext.proceed();
  }
}
```

61.6.6. 서버 측에서의 샘플 구현

서버 측에 엔터티 리더 인터셉터를 구현하려면 **ReaderInterceptor** 인터페이스를 구현하고 **@Provider** 주석으로 주석을 추가하는 클래스를 정의합니다.

*예를 들어 다음 코드는 들어오는 요청의 메시지 본문에서 **company _NAME** 의 모든 인스턴스를 **Red Hat** 으로 대체하는 서버 측의 엔터티 **reader** 인터셉터의 예를 보여줍니다.*

```
// Java
package org.jboss.fuse.example;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {

  @Override
  public Object aroundReadFrom(ReaderInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
    InputStream inputStream = interceptorContext.getInputStream();
    byte[] bytes = new byte[inputStream.available()];
    inputStream.read(bytes);
    String requestContent = new String(bytes);
    requestContent = requestContent.replaceAll("COMPANY_NAME", "Red Hat");
    interceptorContext.setInputStream(new ByteArrayInputStream(requestContent.getBytes()));

    return interceptorContext.proceed();
  }
}
```

61.6.7. 클라이언트 측에서 리더 인터셉터 바인딩

***JAX-RS 2.0** 클라이언트 **API**를 사용하여 **javax.ws.rs.client.Client** 오브젝트 또는 **javax.ws.client.WebTarget** 오브젝트에 엔터티 리더 인터셉터를 직접 등록할 수 있습니다. 즉, **reader** 인터셉터를 선택적으로 다른 범위에 적용할 수 있으므로 인터셉터의 특정 **URI** 경로만 영향을 받습니다.*

*예를 들어 다음 코드는 **client** 오브젝트를 사용하여 수행된 모든 호출에 적용되도록 **SampleClientReaderInterceptor** 인터셉터를 등록하는 방법을 보여줍니다.*

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
```

```
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

JAX-RS 2.0 클라이언트를 사용하여 인터셉터를 등록하는 방법에 대한 자세한 내용은 49.5절. "클라이언트 끝점 구성" 을 참조하십시오.

61.6.8. 서버 측에서 리더 인터셉터 바인딩

서버 측의 **reader** 인터셉터를 바인딩 하려면 (즉, **Apache CXF** 런타임에 설치) 다음 단계를 수행합니다.

1. 다음 코드 조각에 표시된 대로 **reader** 인터셉터 클래스에 **@Provider** 주석을 추가합니다.

```
// Java
package org.jboss.fuse.example;
...
import javax.annotation.Priority;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.ReaderInterceptorContext;

@Priority(value = 10)
@Provider
public class SampleServerReaderInterceptor implements ReaderInterceptor {
  ...
}
```

reader 인터셉터 구현이 **Apache CXF** 런타임에 로드되면 **REST** 구현에서 로드된 클래스를 자동으로 검사하여 **@Provider** 주석(스캔 단계)으로 표시된 클래스를 검색합니다.

2. **XML**에서 **JAX-RS** 서버 엔드포인트를 정의할 때(예: 18.1절. "JAX-RS Server 엔드 포인트 구성"참조), **jaxrs:providers** 요소의 공급자 목록에 **reader** 인터셉터를 추가합니다.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
```

```
    <jaxrs:providers>
      <ref bean="interceptorProvider" />
    </jaxrs:providers>
    <bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerReaderInterceptor"/>

  </jaxrs:server>

</blueprint>
```

참고

이 단계는 **Apache CXF**의 비표준 요구 사항입니다. **JAX-RS** 표준에 따르면 **@Provider** 주석은 인터셉터를 바인딩하는 데 필요한 모든 주석이어야 합니다. 그러나 실제로는 표준 접근 방식은 다소 무독하며 많은 라이브러리가 대규모 프로젝트에 포함될 때 번들링 공급자로 이어질 수 있습니다.

61.7. ENTITY WRITER INTERCEPTOR

61.7.1. 개요

이 섹션에서는 클라이언트 측 또는 서버 측에서 메시지 본문을 작성할 때 출력 스트림을 가로채는 데 사용할 수 있는 엔터티 작성기 인터셉터 를 구현하고 등록하는 방법을 설명합니다. 이는 일반적으로 암호화 및 암호 해독 또는 압축 해제와 같은 요청 본문의 일반적인 변환에 유용합니다.

61.7.2. WriterInterceptor 인터페이스

javax.ws.rs.ext.WriterInterceptor 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.ext;

public interface WriterInterceptor {
    void aroundWriteTo(WriterInterceptorContext context)
            throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

WriterInterceptor 인터페이스를 구현하면 서버 쪽 또는 클라이언트 측에서 기록되는 메시지 본문 **(Entity object)**을 가로챌 수 있습니다. 다음 컨텍스트 중 하나에서 엔터티 작성기 인터셉터를 사용할 수 있습니다.

- 서버 측인터셉터로 바인딩된 경우, 엔터티 작성자 인터셉터는 응답을 마샬링하고 클라이언트로 다시 전송하기 직전에 응답 메시지 본문을 인터셉트합니다.

- 클라이언트 쪽인터셉터로 바인딩된 경우 엔터티 작성자 인터셉터는 요청을 마샬링하고 서버로 전송하기 직전에 요청 메시지 본문을 인터셉트합니다.

61.7.3. WriterInterceptorContext 인터페이스

WriterInterceptor 의 **aroundWriteTo** 방법은 **javax.ws.rs.ext.WriterInterceptorContext** 유형의 하나의 인수를 수신하며 메시지 본문**(Entity object)** 및 메시지 메타데이터 둘 다에 액세스하는 데 사용할 수 있습니다.

WriterInterceptorContext 인터페이스는 다음과 같이 정의됩니다.

```
// Java
...
package javax.ws.rs.ext;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MultivaluedMap;

public interface WriterInterceptorContext extends InterceptorContext {

    void proceed() throws IOException, WebApplicationException;

    Object getEntity();

    void setEntity(Object entity);

    OutputStream getOutputStream();

    public void setOutputStream(OutputStream os);

    MultivaluedMap<String, Object> getHeaders();
}
```

61.7.4. InterceptorContext 인터페이스

WriterInterceptorContext 인터페이스는 기본 **InterceptorContext** 인터페이스에서 상속된 메서드도 지원합니다. **InterceptorContext** 에 대한 정의는 **"InterceptorContext 인터페이스"** 에서 참조하십시오.

61.7.5. 클라이언트 측에서의 샘플 구현

*클라이언트 측에 대해 엔터티 작성기 인터셉터를 구현하려면 **WriterInterceptor** 인터페이스를 구현하는 클래스를 정의합니다.*

*예를 들어 다음 코드는 클라이언트 쪽의 엔터티 작성기 인터셉터 예를 보여줍니다(우선 순위 **10**임)는 발신 요청의 메시지 본문에 추가 텍스트 행을 추가합니다.*

```
// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
public class SampleClientWriterInterceptor implements WriterInterceptor {

  @Override
  public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
    OutputStream outputStream = interceptorContext.getOutputStream();
    String appendedContent = "\nInterceptors always get the last word in.";
    outputStream.write(appendedContent.getBytes());
    interceptorContext.setOutputStream(outputStream);

    interceptorContext.proceed();
  }
}
```

61.7.6. 서버 측에서의 샘플 구현

*서버 측에 대해 엔터티 작성자 인터셉터를 구현하려면 **WriterInterceptor** 인터페이스를 구현하고 **@Provider** 주석으로 주석을 추가하는 클래스를 정의합니다.*

*예를 들어 다음 코드는 서버 측의 엔터티 작성기 인터셉터 예를 보여줍니다(우선 순위 **10**임)는 발신 요청의 메시지 본문에 추가 텍스트 행을 추가합니다.*

```
// Java
package org.jboss.fuse.example;

import java.io.IOException;
import java.io.OutputStream;

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

```
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {

  @Override
  public void aroundWriteTo(WriterInterceptorContext interceptorContext)
        throws IOException, WebApplicationException {
    OutputStream outputStream = interceptorContext.getOutputStream();
    String appendedContent = "\nInterceptors always get the last word in.";
    outputStream.write(appendedContent.getBytes());
    interceptorContext.setOutputStream(outputStream);

    interceptorContext.proceed();
  }
}
```

61.7.7. 클라이언트 측에서 작성자 인터셉터 바인딩

JAX-RS 2.0 클라이언트 **API**를 사용하여 **javax.ws.rs.client.Client** 오브젝트 또는 **javax.ws.client.WebTarget** 오브젝트에 엔터티 작성자 인터셉터를 직접 등록할 수 있습니다. 사실상, 이는 작성자 인터셉터를 선택적으로 다른 범위에 적용할 수 있으므로 인터셉터의 특정 **URI** 경로만 영향을 받습니다.

예를 들어 다음 코드는 **client** 오브젝트를 사용하여 수행된 모든 호출에 적용되도록 **SampleClientReaderInterceptor** 인터셉터를 등록하는 방법을 보여줍니다.

```
// Java
...
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.Response;
...
Client client = ClientBuilder.newClient();
client.register(SampleClientReaderInterceptor.class);
```

JAX-RS 2.0 클라이언트를 사용하여 인터셉터를 등록하는 방법에 대한 자세한 내용은 **49.5**절. “클라이언트 끝점 구성” 을 참조하십시오.

61.7.8. 서버 측에서 작성자 인터셉터 바인딩

서버 측의 작성자 인터셉터를 바인딩 하려면 (즉, **Apache CXF** 런타임에 설치) 다음 단계를 수행합니다.

1. *다음 코드 조각에 표시된 대로 **@Provider** 주석을 **writer** 인터셉터 클래스에 추가합니다.*

```
// Java
package org.jboss.fuse.example;
...
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.ext.Provider;
import javax.ws.rs.ext.WriterInterceptor;
import javax.ws.rs.ext.WriterInterceptorContext;
import javax.annotation.Priority;

@Priority(value = 10)
@Provider
public class SampleServerWriterInterceptor implements WriterInterceptor {
  ...
}
```

*작성자 인터셉터 구현이 **Apache CXF** 런타임에 로드되면 **REST** 구현에서 로드된 클래스를 자동으로 검사하여 **@Provider** 주석(스캔 단계)으로 표시된 클래스를 검색합니다.*

2. ***XML**에서 **JAX-RS** 서버 엔드포인트를 정의할 때(예: 18.1절. “JAX-RS Server 엔드 포인트 구성”참조), **jaxrs:providers** 요소의 공급자 목록에 **writer** 인터셉터를 추가합니다.*

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    ...
>
    ...
    <jaxrs:server id="customerService" address="/customers">
      ...
      <jaxrs:providers>
        <ref bean="interceptorProvider" />
      </jaxrs:providers>
      <bean id="interceptorProvider"
class="org.jboss.fuse.example.SampleServerWriterInterceptor"/>

    </jaxrs:server>

</blueprint>
```

참고

이 단계는 **Apache CXF**의 비표준 요구 사항입니다. **JAX-RS** 표준에 따르면 **@Provider** 주석은 인터셉터를 바인딩하는 데 필요한 모든 주석이어야 합니다. 그러나 실제로는 표준 접근 방식은 다소 무독하며 많은 라이브러리가 대규모 프로젝트에 포함될 때 번들링 공급자로 이어질 수 있습니다.

61.8. 동적 바인딩

61.8.1. 개요

컨테이너 필터 및 컨테이너 인터셉터를 리소스에 바인딩하는 표준 방법은 **@Provider** 주석을 사용하여 필터 및 인터셉터에 주석을 추가하는 것입니다. 이렇게 하면 바인딩이 **global:** 즉, 필터 및 인터셉터가 서버 측의 모든 리소스 클래스 및 리소스 메서드에 바인딩됩니다.

동적 바인딩은 서버 측에 바인딩하는 대체 방법입니다. 이를 통해 인터셉터 및 필터가 적용되는 리소스 방법을 선택하고 선택할 수 있습니다. 필터 및 인터셉터에 대한 동적 바인딩을 활성화하려면 여기에 설명된 대로 사용자 지정 **DynamicFeature** 인터페이스를 구현해야 합니다.

61.8.2. DynamicFeature 인터페이스

DynamicFeature 인터페이스는 **javax.ws.rx.container** 패키지에 다음과 같이 정의됩니다.

```
// Java
package javax.ws.rs.container;

import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.ReaderInterceptor;
import javax.ws.rs.ext.WriterInterceptor;

public interface DynamicFeature {
    public void configure(ResourceInfo resourceInfo, FeatureContext context);
}
```

61.8.3. 동적 기능 구현

다음과 같이 동적 기능을 구현합니다.

1. 이전에 설명한 대로 하나 이상의 컨테이너 필터 또는 컨테이너 인터셉터를 구현합니다. 그러나 **@Provider** 주석으로 주석을 달 지 마십시오 (기타적으로는 전역으로 바인딩되어 동적 기능이 효과적으로 관련이 없습니다).

2. *DynamicFeature 클래스를 구현하여 동적 기능을 만들고 configure 메서드를 재정의합니다.*

3. *구성 메서드에서 resourceInfo 인수를 사용하여 어떤 리소스 클래스 및 이 기능이 호출되는 리소스 메서드를 검색할 수 있습니다.In the configure method, you can use the resourceInfo argument to discover which resource class and which resource method this feature is being called for. 일부 필터 또는 인터셉터를 등록할지 여부를 결정하는 기준으로 이 정보를 사용할 수 있습니다.*

4. *현재 리소스 방법으로 필터 또는 인터셉터를 등록하도록 결정하는 경우 context.register 방법 중 하나를 호출하여 이를 수행할 수 있습니다.*

5. *동적 기능 클래스에 @Provider 주석에 주석을 달아 배포 단계 중에 선택하도록 해야 합니다.*

61.8.4. 동적 기능 예

다음 예제에서는 @GET 로 주석이 달린 MyResource 클래스(또는 하위 클래스)의 모든 메서드에 대해 LoggingFilter 필터를 등록하는 동적 기능을 정의하는 방법을 보여줍니다.

```
// Java
...
import javax.ws.rs.container.DynamicFeature;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.FeatureContext;
import javax.ws.rs.ext.Provider;

@Provider
public class DynamicLoggingFilterFeature implements DynamicFeature {
  @Override
  void configure(ResourceInfo resourceInfo, FeatureContext context) {
    if (MyResource.class.isAssignableFrom(resourceInfo.getResourceClass())
      && resourceInfo.getResourceMethod().isAnnotationPresent(GET.class)) {
      context.register(new LoggingFilter());
  }
}
```

61.8.5. 동적 바인딩 프로세스

JAX-RS 표준을 사용하려면 각 리소스 메서드에 대해 DynamicFeature.configure 메서드를 정확히 한 번 호출해야 합니다. 즉, 모든 리소스 방법에는 동적 기능에 의해 설치된 필터 또는 인터셉터가 잠재적으로 있을 수 있지만, 각 경우에 필터 또는 인터셉터를 등록할지 여부를 결정하는 동적 기능까지 있습니다. 즉 동적 기능에서 지원하는 바인딩의 세분화는 개별 리소스 메서드 수준에 있습니다.

61.8.6. FeatureContext 인터페이스

FeatureContext 인터페이스 (구성 방법에서 필터 및 인터셉터를 등록할 수 있음)는 다음과 같이 **Configurable <>** 의 하위 인터페이스로 정의됩니다.

```
// Java
package javax.ws.rs.core;

public interface FeatureContext extends Configurable<FeatureContext> {
}
```

Configurable <> 인터페이스는 다음과 같이 단일 리소스 방법에 필터 및 인터셉터를 등록하는 다양한 방법을 정의합니다.

```
// Java
...
package javax.ws.rs.core;

import java.util.Map;

public interface Configurable<C extends Configurable> {
    public Configuration getConfiguration();
    public C property(String name, Object value);
    public C register(Class<?> componentClass);
    public C register(Class<?> componentClass, int priority);
    public C register(Class<?> componentClass, Class<?>... contracts);
    public C register(Class<?> componentClass, Map<Class<?>, Integer> contracts);
    public C register(Object component);
    public C register(Object component, int priority);
    public C register(Object component, Class<?>... contracts);
    public C register(Object component, Map<Class<?>, Integer> contracts);
}
```

62장. APACHE CXF 메시지 처리 단계

62.1. 인바운드 단계

표 62.1. "인바운드 메시지 처리 단계" 인바운드 인터셉터 체인에서 사용할 수 있는 단계를 나열합니다.

표 62.1. 인바운드 메시지 처리 단계

단계	설명
RECEIVE	바이너리 첨부 파일의 MIME 경계 결정과 같은 전송 특정 처리를 수행합니다.
PRE_STREAM	전송에서 수신한 원시 데이터 스트림을 처리합니다.
USER_STREAM	
POST_STREAM	
READ	요청이 SOAP 또는 XML 메시지인지 여부를 확인하고 빌드에서 적절한 인터셉터를 추가합니다. 이 단계에서 SOAP 메시지 헤더도 처리됩니다.
PRE_PROTOCOL	프로토콜 수준 처리를 수행합니다. 여기에는 WS-* 헤더의 처리 및 SOAP 메시지 속성 처리가 포함됩니다.
USER_PROTOCOL	
POST_PROTOCOL	
UNMARSHAL	메시지 데이터를 애플리케이션 수준 코드에서 사용하는 개체로 분리합니다.
PRE_LOGICAL	Unmarshalled 메시지 데이터를 처리합니다.
USER_LOGICAL	
POST_LOGICAL	
PRE_INVOKE	
INVOKE	메시지를 애플리케이션 코드에 전달합니다. 서버 측에서 서비스 구현은 이 단계에서 호출됩니다. 클라이언트 측에서 응답이 애플리케이션으로 다시 전달됩니다.
POST_INVOKE	아웃바운드 인터셉터 체인을 호출합니다.

62.2. 아웃바운드 단계

표 62.2. "인바운드 메시지 처리 단계" 인바운드 인터셉터 체인에서 사용할 수 있는 단계를 나열합니다.

표 62.2. 인바운드 메시지 처리 단계

단계	설명
SETUP	체인의 이후 단계에서 필요한 모든 세트를 수행합니다.
PRE_LOGICAL	애플리케이션 수준에서 전달된 손상되지 않은 데이터에 대한 처리를 수행합니다.
USER_LOGICAL	
POST_LOGICAL	
PREPARE_SEND	유선 메시지를 쓰기 위한 연결을 엽니다.
PRE_STREAM	데이터 스트림에 입력을 위해 메시지를 준비하는 데 필요한 처리를 수행합니다.
PRE_PROTOCOL	처리 프로토콜별 정보 처리를 시작합니다.
WRITE	프로토콜 메시지를 작성합니다.
PRE_MARSHAL	메시지를 마샬링합니다.
MARSHAL	
POST_MARSHAL	
USER_PROTOCOL	프로토콜 메시지를 처리합니다.
POST_PROTOCOL	
USER_STREAM	바이트 수준 메시지를 처리합니다.
POST_STREAM	
SEND	메시지를 보내고 전송 스트림을 닫습니다.

중요

*아웃바운드 인터셉터 체인에는 이름이 **_ENDING** 으로 추가되는 미러 일련의 마지막 단계가 있습니다. 종료 단계는 데이터가 유선에 기록되기 전에 일부 터미널 작업이 필요한 인터셉터를 사용합니다.*

63장. APACHE CXF 제공 인터셉터

63.1. 핵심 APACHE CXF 인터셉터

63.1.1. 인바운드

표 63.1. "코어 인바운드 인터셉터" 모든 Apache CXF 엔드포인트에 추가된 핵심 인바운드 인터셉터를 나열합니다.

표 63.1. 코어 인바운드 인터셉터

class	단계	설명
ServiceInvokerInterceptor	**INVOKE**	서비스에서 적절한 메서드를 호출합니다.

63.1.2. 아웃 바운드

Apache CXF는 기본적으로 아웃바운드 인터셉터 체인에 코어 인터셉터를 추가하지 않습니다. 끝점의 아웃바운드 인터셉터 체인의 내용은 사용 중인 기능에 따라 다릅니다.

63.2. 프론트 엔드

63.2.1. JAX-WS

표 63.2. "인바운드 JAX-WS 인터셉터" JAX-WS 엔드포인트의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.2. 인바운드 JAX-WS 인터셉터

class	단계	설명
HolderInInterceptor	**PRE_INVOKE**	메시지의 out 또는 in/out 매개변수에 대해 홀더 오브젝트를 만듭니다.
WrapperClassInInterceptor	**POST_LOGICAL**	래핑된 doc/literal 메시지의 부분을 적절한 개체 배열로 래핑하지 않습니다.

class	단계	설명
LogicalHandlerInInterceptor	**PRE_PROTOCOL**	끝점에서 사용하는 JAX-WS 논리 핸들러로 메시지 처리를 전달합니다. JAX-WS 핸들러가 완료되면 메시지가 인바운드 체인의 다음 인터셉터로 전달됩니다.
SOAPHandlerInterceptor	**PRE_PROTOCOL**	엔드포인트에서 사용하는 JAX-WS SOAP 핸들러로 메시지 처리를 전달합니다. SOAP 핸들러가 메시지와 함께 완료되면 체인의 다음 인터셉터와 함께 메시지가 전달됩니다.

표 63.3. "outbound JAX-WS 인터셉터" JAX-WS 엔드포인트의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.3. outbound JAX-WS 인터셉터

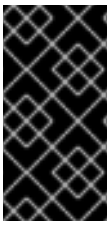
class	단계	설명
HolderOutInterceptor	**PRE_LOGICAL**	holder 개체에서 out 및 in/out 매개변수의 값을 제거하고 메시지의 매개 변수 목록에 값을 추가합니다.
WebFaultOutInterceptor	**PRE_PROTOCOL**	아웃바운드 오류 메시지를 처리합니다.
WrapperClassOutInterceptor	**PRE_LOGICAL**	메시지에 추가되기 전에 doc/literal 메시지와 rpc/literal 메시지가 올바르게 래핑되는지 확인합니다.
LogicalHandlerOutInterceptor	**PRE_MARSHAL**	끝점에서 사용하는 JAX-WS 논리 핸들러로 메시지 처리를 전달합니다. JAX-WS 핸들러가 완료되면 메시지는 아웃바운드 체인의 다음 인터셉터와 함께 전달됩니다.
SOAPHandlerInterceptor	**PRE_PROTOCOL**	엔드포인트에서 사용하는 JAX-WS SOAP 핸들러로 메시지 처리를 전달합니다. SOAP 핸들러가 메시지 처리를 완료하면 체인의 다음 인터셉터로 전달됩니다.
MessageSenderInterceptor	**PREPARE_SEND**	발신 전송을 준비하기 위해 Destination 오브젝트로 다시 호출하여 출력 스트림, 헤더 등을 설정합니다.

63.2.2. JAX-RS

표 63.4. "인바운드 JAX-RS 인터셉터" JAX-RS 엔드포인트의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.4. 인바운드 JAX-RS 인터셉터

class	단계	설명
JAXRSInInterceptor	**PRE_STREAM**	루트 리소스 클래스를 선택하고, 구성된 JAX-RS 요청 필터를 호출하고, 루트 리소스에서 호출할 메서드를 결정합니다.



중요

***JAX-RS** 엔드포인트의 인바운드 체인은 **Service invokerInInterceptor** 인터셉터로 직접 건너뜁니다. **JAXRSInterceptor** 이후에 다른 인터셉터는 호출되지 않습니다.*

표 63.5. "outbound JAX-RS 인터셉터" JAX-RS 엔드포인트의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.5. outbound JAX-RS 인터셉터

class	단계	설명
JAXRSOutInterceptor	**MARSHAL**	응답을 전송할 적절한 형식으로 마샬링합니다.

63.3. 메시지 바인딩

63.3.1. SOAP

표 63.6. "인바운드 SOAP 인터셉터" SOAP Binding을 사용할 때 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.6. 인바운드 SOAP 인터셉터

class	단계	설명

class	단계	설명
CheckFaultInterceptor	**POST_PROTOCOL**	메시지가 오류 메시지인지 확인합니다. 메시지가 오류 메시지인 경우 정상적인 처리가 중단되고 오류 처리가 시작됩니다.
MustUnderstandInterceptor	**PRE_PROTOCOL**	프로세스에서 헤더를 이해해야 합니다.
RPCInInterceptor	**UNMARSHAL**	Unmarshals rpc/literal 메시지입니다. 메시지가 베어 메탈인 경우 메시지 부분을 역직렬화하기 위해 메시지가 **BareInterceptor** 오브젝트에 전달됩니다.
ReadsHeadersInterceptor	**READ**	SOAP 헤더를 구문 분석하고 메시지 오브젝트에 저장합니다.
SoapActionInInterceptor	**READ**	SOAP 작업 헤더를 구문 분석하고 작업에 대한 고유한 작업을 찾으려고 합니다.
SoapHeaderInterceptor	**UNMARSHAL**	작업 매개 변수에 매핑되는 SOAP 헤더를 적절한 오브젝트에 바인딩합니다.
AttachmentInInterceptor	**RECEIVE**	mime 경계의 mime 헤더를 구문 분석하고, **루트** 부분을 찾아 입력 스트림을 재설정하고, 다른 부분을 **Attachment** 오브젝트 컬렉션에 저장합니다.
DocLiteralInInterceptor	**UNMARSHAL**	SOAP 본문에서 첫 번째 요소를 검사하여 적절한 작업을 결정하고 데이터 바인딩을 호출하여 데이터를 읽습니다.
StaxInInterceptor	**POST_STREAM**	메시지에서 **XMLStreamReader** 개체를 만듭니다.Creates an XMLStreamReader object from the message.
URIMappingInterceptor	**UNMARSHAL**	HTTP GET 메서드의 처리를 처리합니다.
SwAInInterceptor	**PRE_INVOKE**	바이너리 SOAP 첨부 파일에 필요한 MIME 핸들러를 생성하고 매개 변수 목록에 데이터를 추가합니다.

표 63.7. "outbound SOAP 인터셉터" SOAP Binding을 사용할 때 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.7. outbound SOAP 인터셉터

class	단계	설명
RPCOutInterceptor	**MARSHAL**	전송을 위한 rpc 스타일 메시지를 마샬링합니다.
SoapHeaderOutFilterInterceptor	**PRE_LOGICAL**	인바운드로만 표시된 모든 SOAP 헤더를 제거합니다.
SoapPreProtocolOutInterceptor	**POST_LOGICAL**	SOAP 버전 및 SOAP 작업 헤더를 설정합니다.
AttachmentOutInterceptor	**PRE_STREAM**	메시지에 포함될 수 있는 첨부 파일을 처리하는 데 필요한 첨부 파일 및 첨부 파일을 설정합니다.Sets the attachment marshalers and the mime things required to process any attachments that might be in the message.
BareOutInterceptor	**MARSHAL**	메시지 부분을 작성합니다.
StaxOutInterceptor	**PRE_STREAM**	메시지에서 **XMLStreamWriter** 오브젝트를 만듭니다.
WrappedOutInterceptor	**MARSHAL**	아웃바운드 메시지 매개 변수를 래핑합니다.
SoapOutInterceptor	**WRITE**	메시지에서 **soap:envelope** 요소와 헤더 블록에 대한 요소를 씁니다. 또한 나머지 인터셉터에 대해 빈 **soap:body** 요소를 씁니다.
SwAOutInterceptor	**PRE_LOGICAL**	SOAP 첨부 파일로 패키징된 바이너리 데이터를 제거하고 나중에 처리하기 위해 저장합니다.

63.3.2. XML

표 63.8. "인바운드 XML 인터셉터" XML Binding을 사용할 때 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.8. 인바운드 XML 인터셉터

class	단계	설명
AttachmentInInterceptor	**RECEIVE**	mime 경계의 mime 헤더를 구문 분석하고, **루트** 부분을 찾아서 입력 스트림을 재설정한 다음, **Attachment** 오브젝트 컬렉션의 다른 부분을 저장합니다.
DocLiteralInInterceptor	**UNMARSHAL**	메시지 본문의 첫 번째 요소를 검사하여 적절한 작업을 확인한 다음 데이터 바인딩을 호출하여 데이터를 읽습니다.
StaxInInterceptor	**POST_STREAM**	메시지에서 **XMLStreamReader** 개체를 만듭니다.Creates an XMLStreamReader object from the message.
URIMappingInterceptor	**UNMARSHAL**	HTTP GET 메서드의 처리를 처리합니다.
XMLMessageInInterceptor	**UNMARSHAL**	XML 메시지를 분리합니다.

표 63.9. "아웃바운드 XML 인터셉터" XML Binding을 사용할 때 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.9. 아웃바운드 XML 인터셉터

class	단계	설명
StaxOutInterceptor	**PRE_STREAM**	메시지에서 **XMLStreamWriter** 개체를 만듭니다.
WrappedOutInterceptor	**MARSHAL**	아웃바운드 메시지 매개 변수를 래핑합니다.
XMLMessageOutInterceptor	**MARSHAL**	전송을 위한 메시지를 마샬링합니다.

63.3.3. CORBA

표 63.10. "인바운드 CORBA 인터셉터" CORBA Binding을 사용할 때 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.10. 인바운드 CORBA 인터셉터

class	단계	설명
CorbaStreamInInterceptor	**PRE_STREAM**	CORBA 메시지를 역직렬합니다.
BareInInterceptor	**UNMARSHAL**	메시지 부분을 역직렬합니다.

표 63.11. "아웃바운드 CORBA 인터셉터" CORBA Binding을 사용할 때 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.11. 아웃바운드 CORBA 인터셉터

class	단계	설명
CorbaStreamOutInterceptor	**PRE_STREAM**	메시지를 serialize합니다.
BareOutInterceptor	**MARSHAL**	메시지 부분을 작성합니다.
CorbaStreamOutEndingInterceptor	**USER_STREAM**	메시지에 대한 스트림 가능한 오브젝트를 생성하고 메시지 컨텍스트에 저장합니다.

63.4. 기타 기능

63.4.1. 로깅

표 63.12. "인바운드 로깅 인터셉터" 로깅을 지원하기 위해 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.12. 인바운드 로깅 인터셉터

class	단계	설명
LoggingInInterceptor	**RECEIVE**	원시 메시지 데이터를 로깅 시스템에 씁니다.

표 63.13. "아웃바운드 로깅 인터셉터" 로깅을 지원하기 위해 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.13. 아웃바운드 로깅 인터셉터

class	단계	설명
LoggingOutInterceptor	**PRE_STREAM**	로깅 시스템에 아웃바운드 메시지를 씁니다.

로깅에 대한 자세한 내용은 19장. Apache CXF Logging 에서 참조하십시오.

63.4.2. WS-Addressing

표 63.14. "인바운드 WS-Addressing 인터셉터" WS-Addressing을 사용할 때 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.14. 인바운드 WS-Addressing 인터셉터

class	단계	설명
MAPCodec	**PRE_PROTOCOL**	메시지 주소 지정 속성을 디코딩합니다.

표 63.15. "아웃바운드 WS-Addressing 인터셉터" WS-Addressing을 사용할 때 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.15. 아웃바운드 WS-Addressing 인터셉터

class	단계	설명
MAPAggregator	**PRE_LOGICAL**	메시지의 주소 지정 속성을 집계합니다.
MAPCodec	**PRE_PROTOCOL**	메시지 주소 지정 속성을 인코딩합니다.

WS-Addressing에 대한 자세한 내용은 20장. WS-Addressing 배포 을 참조하십시오.

63.4.3. WS-RM

중요

WS-RM은 WS-Addressing을 사용하므로 모든 WS-Addressing 인터셉터도 인터셉터 체인에 추가됩니다.

표 63.16. "인바운드 WS-RM 인터셉터" WS-RM을 사용할 때 끝점의 인바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.16. 인바운드 WS-RM 인터셉터

class	단계	설명
RMInInterceptor	**PRE_LOGICAL**	메시지 부분 및 승인 메시지 집계를 처리합니다.
RMSoapInterceptor	**PRE_PROTOCOL**	메시지에서 WS-RM 속성을 인코딩하고 디코딩합니다.

표 63.17. "아웃바운드 WS-RM 인터셉터" WS-RM을 사용할 때 끝점의 아웃바운드 메시지 체인에 추가된 인터셉터를 나열합니다.

표 63.17. 아웃바운드 WS-RM 인터셉터

class	단계	설명
RMOutInterceptor	**PRE_LOGICAL**	메시지의 청크 및 청크 전송을 처리합니다. 또한 승인 및 재전송 요청 처리도 처리합니다.
RMSoapInterceptor	**PRE_PROTOCOL**	메시지에서 WS-RM 속성을 인코딩하고 디코딩합니다.

WS-RM에 대한 자세한 내용은 21장. 신뢰할 수 있는 메시징 활성화 에서 참조하십시오.

64장. 인터셉터 공급자

64.1. 개요

인터셉터 공급자는 인터셉터 체인이 연결된 **Apache CXF** 런타임의 오브젝트입니다. 모두 **org.apache.cxf.interceptor.InterceptorProvider** 인터페이스를 구현합니다. 개발자는 자체 인터셉터를 모든 인터셉터 공급자에 연결할 수 있습니다.

64.2. 공급자 목록

다음 오브젝트는 인터셉터 공급자입니다.

- **AddressingPolicyInterceptorProvider**
- **ClientFactoryBean**
- **ClientImpl**
- **ClientProxyFactoryBean**
- **CorbaBinding**
- **CXFBusImpl**
- **org.apache.cxf.jaxws.EndpointImpl**
- **org.apache.cxf.endpoint.EndpointImpl**
- **ExtensionManagerBus**
- **JAXRSClientFactoryBean**

- *JAXRSServerFactoryBean*
- *JAXRSServiceImpl*
- *JaxWsClientEndpointImpl*
- *JaxWsClientFactoryBean*
- *JaxWsEndpointImpl*
- *JaxWsProxyFactoryBean*
- *JaxWsServerFactoryBean*
- *JaxwsServiceBuilder*
- *MTOMPolicyInterceptorProvider*
- *NoOpPolicyInterceptorProvider*
- *ObjectBinding*
- *RMPolicyInterceptorProvider*
- *ServerFactoryBean*
- *ServiceImpl*

- ***SimpleServiceBuilder***
- ***SoapBinding***
- ***WrappedEndpoint***
- ***WrappedService***
- ***XMLBinding***

VIII 부. APACHE CXF 기능

*이 가이드에서는 **Apache CXF**의 다양한 고급 기능을 활성화하는 방법을 설명합니다.*

65장. 빈 유효성 검사

초록

빈 유효성 검사는 서비스 클래스 또는 인터페이스에 Java 주석을 추가하여 런타임 제약 조건을 정의할 수 있는 Java 표준입니다. Apache CXF는 인터셉터를 사용하여 이 기능을 웹 서비스 메서드 호출과 통합합니다.

65.1. 소개

65.1.1. 개요

빈 유효성 검사 1.1(JSR-349) - 원래 빈 유효성 검사 1.0(JSR-303) 표준으로 발전된 경우 Java 주석을 사용하여 런타임 시 확인할 수 있는 제약 조건을 선언할 수 있습니다. 주석을 사용하여 Java 코드의 다음 부분에서 제약 조건을 정의할 수 있습니다.

- 빈 클래스의 필드입니다.
- 메서드 및 생성자 매개 변수.
- 메서드 반환 값.

65.1.2. 주석이 달린 클래스의 예

다음 예제에서는 몇 가지 표준 빈 유효성 검사 제약 조건이 추가된 Java 클래스를 보여줍니다.

```
// Java
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Max;
import javax.validation.Valid;
...
public class Person {
    @NotNull private String firstName;
    @NotNull private String lastName;
    @Valid @NotNull private Person boss;

    public @NotNull String saveItem( @Valid @NotNull Person person, @Max( 23 ) BigDecimal age )
{
        // ...
    }
}
```

65.1.3. 빈 유효성 검사 또는 스키마 검증?

일부 면에서 빈 검증 및 스키마 검증은 매우 유사합니다. **XML** 스키마를 사용하여 끝점을 구성하는 것은 웹 서비스 끝점에서 런타임에 메시지를 확인하는 잘 설정된 방법입니다. **XML** 스키마는 들어오고 나가는 메시지에 대한 빈 유효성 검사와 동일한 제한 조건의 대부분을 확인할 수 있습니다. 그러나 빈 검증은 다음 이유 중 하나 이상에 유용한 대체가 될 수 있습니다.

- 빈 유효성 검사를 사용하면 **XML** 스키마와 독립적으로 제약 조건을 정의할 수 있습니다(예: 코드 우선 서비스 개발의 경우).
- 현재 **XML** 스키마가 너무 느릴 경우 빈 유효성 검사를 사용하여 엄격한 제약 조건을 정의할 수 있습니다.
- 빈 유효성 검사를 사용하면 **XML** 스키마 언어를 사용하여 정의할 수 없는 사용자 지정 제약 조건을 정의할 수 있습니다.

65.1.4. 종속 항목

Bean Validation 1.1(JSR-349) 표준에서는 구현이 아닌 **API**만 정의합니다. 따라서 종속성은 다음 두 부분으로 제공되어야 합니다.

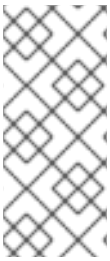
- 핵심 종속 항목**-** 빈 검증 **1.1 API, Java** 통합 표현식 언어 **API** 및 구현을 제공합니다.
- **Hibernate Validator** 종속성**-** 빈 검증 **1.1** 구현을 제공합니다.

65.1.5. 핵심 종속 항목

빈 유효성 검사를 사용하려면 프로젝트의 **Maven pom.xml** 파일에 다음 코어 종속 항목을 추가해야 합니다.

```
<dependency>
   <groupId>javax.validation</groupId>
   <artifactId>validation-api</artifactId>
   <version>1.1.0.Final</version>
</dependency>
<dependency>
   <groupId>javax.el</groupId>
   <artifactId>javax.el-api</artifactId>
   <!-- use 3.0-b02 version for Java 6 -->
```

```
    <version>3.0.0</version>
</dependency>
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <!-- use 3.0-b01 version for Java 6 -->
    <version>3.0.0</version>
</dependency>
```



참고

javax.el/javax.el-api 및 **org.glassfish/javax.el** 종속성은 **Java** 통합 표현식 언어의 **API** 및 구현을 제공합니다. 이 표현식 언어는 빈 유효성 검사를 통해 내부적으로 사용되지만 애플리케이션 프로그래밍 수준에서는 중요하지 않습니다.

65.1.6. Hibernate Validator 종속 항목

빈 유효성 검사의 **Hibernate Validator** 구현을 사용하려면 프로젝트의 **Maven pom.xml** 파일에 다음과 같은 추가 종속 항목을 추가해야 합니다.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.3.Final</version>
</dependency>
```

65.1.7. OSGi 환경에서 검증 공급자 해결

유효성 검사 공급자를 확인하는 기본 메커니즘에는 **classpath**를 검사하여 공급자 리소스를 찾아야 합니다. 그러나 **OSGi(Apache Karaf)** 환경의 경우 검증 공급자(예: **Hibernate validator)**가 별도의 번들에 패키지되어 애플리케이션 **classpath**에서 자동으로 사용할 수 없기 때문에 이 메커니즘이 작동하지 않습니다. **OSGi**의 컨텍스트에서 **Hibernate** 유효성 검증기를 애플리케이션 번들에 연결해야 하며, **OSGi**는 이를 성공적으로 수행하는 데 약간의 도움이 필요합니다.

65.1.8. OSGi에서 명시적으로 검증 공급자 구성

OSGi의 컨텍스트에서 자동 검색을 사용하지 않고 유효성 검사 공급자를 명시적으로 구성해야 합니다. 예를 들어 빈 유효성 검사를 활성화하기 위해 공통 검증 기능을 사용하는 경우 다음과 같이 검증 공급자를 사용하여 구성해야 합니다. “빈 유효성 검사 기능”

```
<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
```

```
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

여기서 **HibernateValidationProviderResolver** 는 **Hibernate** 검증 공급자를 래핑하는 사용자 정의 클래스입니다.

65.1.9. HibernateValidationProviderResolver 클래스의 예

다음 코드 예제에서는 **Hibernate** 유효성 검증기를 확인하는 사용자 지정 **HibernateValidationProviderResolver** 를 정의하는 방법을 보여줍니다.

```
// Java
package org.example;

import static java.util.Collections.singletonList;
import org.hibernate.validator.HibernateValidator;
import javax.validation.ValidationProviderResolver;
import java.util.List;

/**
 * OSGi-friendly implementation of {@code javax.validation.ValidationProviderResolver} returning
 * {@code org.hibernate.validator.HibernateValidator} instance.
 *
 */
public class HibernateValidationProviderResolver implements ValidationProviderResolver {

    @Override
    public List getValidationProviders() {
        return singletonList(new HibernateValidator());
    }
}
```

Maven 번들 플러그인을 사용하도록 구성된 **Maven** 빌드 시스템에서 이전 클래스를 빌드할 때 배포 시 애플리케이션이 **Hibernate** 검증기 번들에 연결됩니다(이미 **Hibernate** 유효성 검사기 번들을 이미 배포한 것으로 가정).

65.2. 빈 유효성 검사로 서비스 개발

65.2.1. 서비스 빈에 주석 달기

65.2.1.1. 개요

빈 유효성 검사를 사용하여 서비스를 개발하는 첫 번째 단계는 관련 검증 주석을 서비스를 나타내는 **Java** 클래스 또는 인터페이스에 적용하는 것입니다. 검증 주석을 사용하면 메서드 매개 변수, 반환 값 및

클래스 필드에 제약 조건을 적용할 수 있습니다. 그러면 서비스가 호출될 때마다 런타임에 확인됩니다.

65.2.1.2. 간단한 입력 매개변수 검증

매개변수가 간단한 **Java** 유형인 서비스 메서드의 매개변수를 검증하기 위해 빈 검증 **API(javax.validation.constraints** 패키지)에서 제약 조건 주석을 적용할 수 있습니다. 예를 들어 다음 코드 예제에서는 **null (@Not null** 주석)에 대한 매개 변수를 모두 테스트 합니다. **id** 문자열이 **\\d+** 정규식 **(@Pattern** 주석)과 일치하는지 여부 및 **name** 문자열의 길이가 **1 ~ 50** 범위에 있는지 여부를 테스트합니다.

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
@POST
@Path("/books")
public Response addBook(
      @NotNull @Pattern(regexp = "\\d+") @FormParam("id") String id,
      @NotNull @Size(min = 1, max = 50) @FormParam("name") String name) {
   // do some work
   return Response.created().build();
}
```

65.2.1.3. 복잡한 입력 매개변수 검증

복잡한 입력 매개변수(오브젝트 인스턴스)를 검증하려면 다음 예와 같이 **@Valid** 주석을 매개변수에 적용합니다.

```
import javax.validation.Valid;
...
@POST
@Path("/books")
public Response addBook( @Valid Book book ) {
   // do some work
   return Response.created().build();
}
```

@Valid 주석은 자체적으로 제약 조건을 지정하지 않습니다. **Book** 매개 변수에 **@Valid** 에 주석을 답니다. 유효성 검사 엔진에 주석을 달 때 **Book** 클래스**(recursively)**의 정의 내에서 조회할 유효성 검사 엔진에 효과적으로 알립니다. 이 예제에서 **Book** 클래스는 다음과 같이 **ID** 및 **name** 필드에 대한 유효성 검사 제약 조건으로 정의됩니다.

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
...
public class Book {
```

```
    @NotNull @Pattern(regexp = "\\d+") private String id;
    @NotNull @Size(min = 1, max = 50) private String name;

    // ...
}
```

65.2.1.4. 반환 값 검증 (non-Response)

일반 메서드 반환 값(non-Response)에 검증을 적용하려면 메서드 서명 앞에 주석을 추가합니다. 예를 들어 nullness(@Not null 주석)에 대한 반환 값을 테스트하고 검증 제약 조건을 재귀적으로 테스트하려면 다음과 같이 getBook 메서드에 주석을 달 수 있습니다.

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
...
@GET
@Path("/books/{bookId}")
@Override
@NotNull @Valid
public Book getBook(@PathParam("bookId") String id) {
    return new Book( id );
}
```

65.2.1.5. 반환 값 검증 (Response)

javax.ws.rs.core.Response 오브젝트를 반환하는 메서드에 검증을 적용하려면 non-Response 사례에서와 동일한 주석을 사용할 수 있습니다. 예를 들면 다음과 같습니다.

```
import javax.validation.constraints.NotNull;
import javax.validation.Valid;
import javax.ws.rs.core.Response;
...
@GET
@Path("/books/{bookId}")
@Valid @NotNull
public Response getBookResponse(@PathParam("bookId") String id) {
    return Response.ok( new Book( id ) ).build();
}
```

65.2.2. 표준 주석

65.2.2.1. 빈 검증 제약 조건

표 65.1. "빈 빈 유효성 검사에 대한 표준 주석" 빈 유효성 검사 사양에 정의된 표준 주석을 표시합니다. 이 사양은 필드 및 메서드 반환 값 및 매개변수에 대한 제약 조건을 정의하는 데 사용할 수 있습니다(클래스 수준에서 표준 주석 중 하나를 적용할 수 있음).

표 65.1. 빈 빈 유효성 검사에 대한 표준 주석

주석	적용 대상	설명
@AssertFalse	**부울**,**부울**	주석이 달린 요소가 **false** 인지 확인합니다.
@AssertTrue	**부울**,**부울**	주석이 달린 요소가 **true** 인지 확인합니다.
@DecimalMax(value=, inclusive=)	**BigInteger**,**BigInteger**,**CharSequence**,**바이트**,**짧은**,**int** 및 기본 형식 래퍼s	**inclusive=false** 인 경우 주석이 달린 값이 지정된 최대값보다 작은지 확인합니다. 그렇지 않으면 값이 지정된 최대값보다 작거나 같은지 확인합니다. **value** 매개 변수는 **BigDecimal** 문자열 형식의 최대값을 지정합니다.
@DecimalMin(value=, inclusive=)	**BigInteger**,**BigInteger**,**CharSequence**,**바이트**,**짧은**,**int** 및 기본 형식 래퍼s	**inclusive=false** 이면 주석이 달린 값이 지정된 최소값보다 큰지 확인합니다. 그렇지 않으면 값이 지정된 최소값보다 크거나 같은지 확인합니다. **value** 매개 변수는 **BigDecimal** 문자열 형식의 최소값을 지정합니다.
@Digits(integer=, fraction=)	**BigInteger**,**BigInteger**,**CharSequence**,**바이트**,**짧은**,**int** 및 기본 형식 래퍼s	주석이 있는 값이 최대 정수 숫자 및 소수 부분 소수 자릿수를 갖는 숫자인지 확인합니다.Checks whether the annotated value is a number having up to **integer** digits and **fraction** fractional digits.
@Future	**java.util.Date**, **java.util.Calendar**	주석이 달린 날짜가 나중에 있는지 확인합니다.
@Max(value=)	**BigInteger**,**BigInteger**,**CharSequence**,**바이트**,**짧은**,**int** 및 기본 형식 래퍼s	주석이 지정된 최대값보다 작거나 같은지 여부를 확인합니다.
@Min(value=)	**BigInteger**,**BigInteger**,**CharSequence**,**바이트**,**짧은**,**int** 및 기본 형식 래퍼s	주석이 지정된 최소값보다 크거나 같은지 여부를 확인합니다.Checks whether the annotated value is greater than or equal to the specified minimum.

주석	적용 대상	설명
@NotNull	모든 유형	주석이 달린 값이 **null** 이 아닌지 확인합니다.
@Null	모든 유형	주석이 달린 값이 **null** 인지 확인합니다.
@Past	**java.util.Date**, **java.util.Calendar**	주석이 달린 날짜가 과거인지 여부를 확인합니다.
@Pattern(regex=, flag=)	**CharSequence**	주석이 달린 문자열이 지정된 플래그 일치를 고려하여 정규 표현식 **regex** 와 일치하는지 확인합니다.
@Size(min=, max=)	**CharSequence**,**Collection**,**Map** 및 arrays	주석이 달린 컬렉션, 맵 또는 배열의 크기가 **min** 과 **max** (포함) 사이에 있는지 확인합니다.
@Valid	모든 비독성 유형	주석이 있는 오브젝트에 검증을 재귀적으로 수행합니다. 개체가 컬렉션 또는 배열인 경우 요소를 재귀적으로 유효성을 검사합니다.If the object is a collection or an array, the elements are validated recursively. 오브젝트가 맵인 경우 값 요소의 유효성을 재귀적으로 확인합니다.

65.2.3. 사용자 정의 주석

65.2.3.1. Hibernate에서 사용자 정의 제약 조건 정의

*빈 검증 **API**를 사용하여 자체 사용자 정의 제약 조건 주석을 정의할 수 있습니다. **Hibernate** 유효성 검사기 구현에서 이 작업을 수행하는 방법에 대한 자세한 내용은 **Hibernate** 검증기 참조 가이드의 사용자 지정 제한 조건 생성 장을 참조하십시오.*

65.3. 빈 빈 유효성 검사 구성

65.3.1. JAX-WS 구성

65.3.1.1. 개요

*이 섹션에서는 **Blueprint XML** 또는 **Spring XML**에서 정의된 **JAX-WS** 서비스 끝점에서 빈 유효성 검사를 활성화하는 방법을 설명합니다. 빈 유효성 검사를 수행하는 데 사용되는 인터셉터는 **JAX-WS** 끝점*

과 JAX-RS 1.1 끝점에 모두 일반적입니다(JAX-RS 2.0 엔드포인트는 다른 인터셉터 클래스를 사용합니다.

65.3.1.2. 네임스페이스

이 섹션에 표시된 XML 예제에서는 다음 표에 표시된 것처럼 jaxws 네임스페이스 접두사를 Blueprint 또는 Spring의 적절한 네임스페이스에 매핑해야 합니다.

XML 언어	네임스페이스
Blueprint	http://cxf.apache.org/blueprint/jaxws
Spring	http://cxf.apache.org/jaxws

65.3.1.3. 빈 유효성 검사 기능

JAX-WS 끝점에서 빈 유효성 검사를 활성화하는 가장 간단한 방법은 빈 유효성 검사 기능을 엔드포인트에 추가하는 것 입니다. 빈 유효성 검사 기능은 다음 클래스에서 구현됩니다.

org.apache.cxf.validation.BeanValidationFeature

이 기능 클래스의 인스턴스를 JAX-WS 끝점에 추가하여(Java API를 통해 또는 Jaxws:features:features의 jaxws:features 하위 요소 XML에서 jaxws:endpoint 를 통해) 끝점에서 빈 유효성 검사를 활성화할 수 있습니다. 이 기능은 들어오는 메시지 데이터를 확인하는 In 인터셉터와 반환 값을 검증하는 Out 인터셉터(기본 구성 매개 변수를 사용하여 인터셉터가 생성된 위치)를 설치합니다.

65.3.1.4. 빈 유효성 검사 기능을 사용한 샘플 JAX-WS 구성

다음 XML 예제에서는 commonValidationFeature bean을 JAX-WS 기능으로 추가하여 JAX-WS 끝점에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
                serviceName="s:BookWorld"
                endpointName="s:BookWorldPort"
                implementor="#bookWorldValidation"
                address="/bwsoap">
    <jaxws:features>
        <ref bean="commonValidationFeature" />
    </jaxws:features>
</jaxws:endpoint>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>
```

```
<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

HibernateValidationProviderResolver 클래스의 샘플 구현은 "HibernateValidationProviderResolver 클래스의 예" 을 참조하십시오. **OSGi** 환경(**Apache Karaf**)의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.

참고

컨텍스트에 따라 **jaxws** 접두사를 **Blueprint** 또는 **Spring**의 적절한 **XML** 네임스페이스에 매핑해야 합니다.

65.3.1.5. 일반적인 빈 유효성 검사 1.1 인터셉터

빈 유효성 검사 구성을 보다 세밀하게 제어하려면 빈 검증 기능을 사용하는 대신 인터셉터를 개별적으로 설치할 수 있습니다. 빈 유효성 검사 기능 대신 다음 인터셉터 중 하나 또는 둘 다를 구성할 수 있습니다.

org.apache.cxf.validation.BeanValidationInInterceptor

JAX-WS(또는 **JAX-RS 1.1**) 엔드포인트에 설치하는 경우 유효성 검사 제약 조건에 대해 리소스 메서드 매개 변수의 유효성을 검증합니다. 유효성 검사가 실패하면 **javax.validation.ConstraintViolationException** 예외를 발생시킵니다. 이 인터셉터를 설치하려면 **XML**의 **jaxws:inInterceptors** 하위 요소(또는 **XML**의 **jaxrs:inInterceptors** 하위 요소)를 통해 엔드포인트에 추가합니다.

org.apache.cxf.validation.BeanValidationOutInterceptor

JAX-WS(또는 **JAX-RS 1.1**) 엔드포인트에 설치하는 경우 유효성 검사 제약 조건에 대해 응답 값을 검증합니다. 유효성 검사가 실패하면 **javax.validation.ConstraintViolationException** 예외를 발생시킵니다. 이 인터셉터를 설치하려면 **XML**의 **jaxws:outInterceptors** 하위 요소(또는 **XML**의 **jaxrs:outInterceptors** 하위 요소)를 통해 엔드포인트에 추가합니다.

65.3.1.6. 빈 유효성 검사 인터셉터를 사용한 샘플 JAX-WS 구성

다음 **XML** 예제에서는 관련 **In** 인터셉터 빈 및 **Out** 인터셉터 빈을 엔드포인트에 명시적으로 추가하여 **JAX-WS** 끝점에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxws:endpoint xmlns:s="http://bookworld.com"
                serviceName="s:BookWorld"
                endpointName="s:BookWorldPort"
                implementor="#bookWorldValidation"
                address="/bwsoap">
    <jaxws:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxws:inInterceptors>

    <jaxws:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxws:outInterceptors>
</jaxws:endpoint>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>
<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

***HibernateValidationProviderResolver** 클래스의 샘플 구현은 "**HibernateValidationProviderResolver** 클래스의 예" 을 참조하십시오. **OSGi** 환경(**Apache Karaf**)의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.*

65.3.1.7. BeanValidationProvider 구성

***org.apache.cxf.validation.BeanValidationProvider** 는 빈 유효성 검사 구현(검증공급자)을 래핑하는 간단한 래퍼 클래스입니다. 기본 **BeanValidationProvider** 클래스를 재정의하여 빈 유효성 검사 구현을 사용자 지정할 수 있습니다. **BeanValidationProvider** 빈을 사용하면 다음 공급자 클래스 중 하나 이상을 덮어쓸 수 있습니다.*

javax.validation.ParameterNameProvider

*메서드 및 생성자 매개 변수의 이름을 제공합니다.**Provides names for method and constructor parameters. Java** 리플렉션 **API**에서 메서드 매개 변수 또는 생성자 매개 변수의 이름에 대한 액세스 권한을 부여 하지 않으므로 이 클래스가 필요합니다.*

javax.validation.spi.ValidationProvider<T>

*지정된 유형인 **T** 에 대한 빈 유효성 검사를 구현합니다. 고유한 **ValidationProvider** 클래스를 구*

현하면 고유한 클래스에 대한 사용자 지정 검증 규칙을 정의할 수 있습니다. 이 메커니즘을 효과적으로 사용하면 빈 검증 프레임워크를 확장할 수 있습니다.

javax.validation.ValidationProviderResolver

ValidationProvider 클래스를 검색하고 검색된 클래스 목록을 반환하는 메커니즘을 구현합니다. 기본 확인자는 **classpath**의 **META-INF/services/javax.validation.spi.ValidationProvider** 파일을 찾습니다. 여기에는 **ValidationProvider** 클래스 목록이 포함되어야 합니다.

javax.validation.ValidatorFactory

javax.validation.Validator 인스턴스를 반환하는 팩토리입니다.

org.apache.cxf.validation.ValidationConfiguration

유효성 검사 공급자 계층에서 더 많은 클래스를 재정의할 수 있는 **CXF** 래퍼 클래스입니다.

BeanValidationProvider 를 사용자 정의하려면 사용자 정의 **BeanValidationProvider** 인스턴스를 검증 **In** 인터셉터의 생성자에 전달하고 검증 아웃 인터셉터 생성자에 전달합니다. 예를 들면 다음과 같습니다.

```
<bean id="validationProvider" class="org.apache.cxf.validation.BeanValidationProvider" />

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
   <property name="provider" ref="validationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
   <property name="provider" ref="validationProvider" />
</bean>
```

65.3.2. JAX-RS 구성

65.3.2.1. 개요

이 섹션에서는 **Blueprint XML** 또는 **Spring XML**에서 정의된 **JAX-RS** 서비스 끝점에서 빈 유효성 검사를 활성화하는 방법을 설명합니다. 빈 유효성 검사를 수행하는 데 사용되는 인터셉터는 **JAX-WS** 끝점과 **JAX-RS 1.1** 끝점에 모두 일반적입니다(**JAX-RS 2.0** 엔드포인트는 다른 인터셉터 클래스를 사용합니다.

65.3.2.2. 네임스페이스

이 섹션에 표시된 **XML** 예제에서는 다음 표에 표시된 것처럼 **jaxws** 네임스페이스 접두사를 **Blueprint** 또는 **Spring**의 적절한 네임스페이스에 매핑해야 합니다.

XML 언어	네임스페이스
Blueprint	http://cxf.apache.org/blueprint/jaxws
Spring	http://cxf.apache.org/jaxws

65.3.2.3. 빈 유효성 검사 기능

JAX-RS 끝점에서 빈 유효성 검사를 활성화하는 가장 간단한 방법은 빈 유효성 검사 기능을 엔드포인트에 추가하는 것 입니다. 빈 유효성 검사 기능은 다음 클래스에서 구현됩니다.

org.apache.cxf.validation.BeanValidationFeature

이 기능 클래스의 인스턴스를 JAX-RS 끝점에 추가하여(Java API를 통해 또는 jaxrs:features 하위 요소 XML에서 jaxrs:server 를 통해) 엔드포인트에서 빈 유효성 검사를 활성화할 수 있습니다. 이 기능은 들어오는 메시지 데이터를 확인하는 In 인터셉터와 반환 값을 검증하는 Out 인터셉터(기본 구성 매개 변수를 사용하여 인터셉터가 생성된 위치)를 설치합니다.

65.3.2.4. 검증 예외 매퍼

또한 JAX-RS 엔드포인트를 사용하려면 유효성 검사 예외를 HTTP 오류 응답에 매핑해야 하는 유효성 검사 예외 매퍼 를 구성해야 합니다. 다음 클래스는 JAX-RS에 대한 유효성 검사 예외 매핑을 구현합니다.

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

JAX-RS 2.0 사양에 따라 검증 예외 매핑을 구현합니다. 모든 입력 매개변수 유효성 검증 위반은 HTTP 상태 코드 400 Bad Request 에 매핑되고 모든 반환 값 유효성 검사 위반(또는 내부 검증 위반)은 HTTP 상태 코드 500 Internal Server Error 에 매핑됩니다.

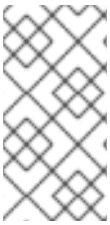
65.3.2.5. 샘플 JAX-RS 구성

다음 XML 예제에서는 commonValidationFeature bean을 JAX-RS 기능으로 추가하고 exceptionMapper 빈을 JAX-RS 공급자로 추가하여 JAX-RS 엔드포인트에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxrs:server address="/bwrest">
   <jaxrs:serviceBeans>
      <ref bean="bookWorldValidation"/>
   </jaxrs:serviceBeans>
   <jaxrs:providers>
      <ref bean="exceptionMapper"/>
   </jaxrs:providers>
   <jaxrs:features>
      <ref bean="commonValidationFeature" />
```

```
    </jaxrs:features>
</jaxrs:server>

<bean id="bookWorldValidation"
class="org.apache.cxf.systest.jaxrs.validation.spring.BookWorldImpl"/>
<beanid="exceptionMapper"class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="commonValidationFeature" class="org.apache.cxf.validation.BeanValidationFeature">
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

HibernateValidationProviderResolver 클래스의 샘플 구현은 **"HibernateValidationProviderResolver 클래스의 예"** 을 참조하십시오. **OSGi** 환경(**Apache Karaf**)의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.



참고

컨텍스트에 따라 **jaxrs** 접두사를 **Blueprint** 또는 **Spring**의 적절한 **XML** 네임스페이스에 매핑해야 합니다.

65.3.2.6. 일반적인 빈 유효성 검사 1.1 인터셉터

빈 유효성 검사 기능을 사용하는 대신 빈 유효성 검사 인터셉터를 설치하여 검증 구현을 보다 세밀하게 제어할 수 있습니다. **JAX-RS**는 이를 위해 **JAX-WS**와 동일한 인터셉터를 사용합니다. **"일반적인 빈 유효성 검사 1.1 인터셉터"**

65.3.2.7. 빈 유효성 검사 인터셉터를 사용한 샘플 JAX-RS 구성

다음 **XML** 예제에서는 관련 **In** 인터셉터 빈 및 **Out** 인터셉터 빈을 서버 엔드포인트에 명시적으로 추가하여 **JAX-RS** 엔드포인트에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxrs:server address="/">
    <jaxrs:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxrs:inInterceptors>

    <jaxrs:outInterceptors>
        <ref bean="validationOutInterceptor" />
    </jaxrs:outInterceptors>

    <jaxrs:serviceBeans>
```

```
    ...
    </jaxrs:serviceBeans>

    <jaxrs:providers>
        <ref bean="exceptionMapper"/>
    </jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor" class="org.apache.cxf.validation.BeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor" class="org.apache.cxf.validation.BeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

HibernateValidationProviderResolver 클래스의 샘플 구현은 **"HibernateValidationProviderResolver 클래스의 예"** 을 참조하십시오. **OSGi** 환경**(Apache Karaf)**의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.

65.3.2.8. BeanValidationProvider 구성

"BeanValidationProvider 구성" 에 설명된 대로 사용자 정의 **BeanValidationProvider** 인스턴스를 검증 인터셉터에 삽입할 수 있습니다.

65.3.3. JAX-RS 2.0 구성

65.3.3.1. 개요

JAX-RS 1.1(Aust-WS와 공통 검증 인터셉터를 공유함)과 달리 **JAX-RS 2.0** 구성은 **JAX-RS 2.0**에 고유한 전용 검증 인터셉터 클래스에 의존합니다.

65.3.3.2. 빈 유효성 검사 기능

JAX-RS 2.0의 경우 다음 클래스에서 구현되는 전용 빈 유효성 검사 기능이 있습니다.

org.apache.cxf.validation.JAXRSBeanValidationFeature

JAX-RS 엔드포인트(Java API를 통해 또는 XML에서 jaxrs:server 의 jaxrs:features 하위 요소)에 이 기능의 인스턴스를 추가하면 JAX-RS 2.0 서버 끝점에서 빈 유효성 검사를 활성화할 수 있습니다. 이 기능은 들어오는 메시지 데이터를 확인하는 In 인터셉터와 반환 값을 검증하는 Out 인터셉터(기본 구성 매개 변수를 사용하여 인터셉터가 생성된 위치)를 설치합니다.

65.3.3.3. 검증 예외 매퍼

JAX-RS 2.0은 JAX-RS 1.x와 동일한 검증 예외 매퍼 클래스를 사용합니다.

org.apache.cxf.jaxrs.validation.ValidationExceptionMapper

JAX-RS 2.0 사양에 따라 검증 예외 매핑을 구현합니다. 모든 입력 매개변수 유효성 검증 위반은 HTTP 상태 코드 400 Bad Request 에 매핑되고 모든 반환 값 유효성 검사 위반(또는 내부 검증 위반)은 HTTP 상태 코드 500 Internal Server Error 에 매핑됩니다.

65.3.3.4. 빈 유효성 검사 호출기

기본이 아닌 라이프사이클 정책(예: Spring 라이프사이클 관리 사용)을 사용하여 JAX-RS 서비스를 구성하는 경우(예: Spring 라이프사이클 관리 사용) org.apache.cxf.jaxrs.validation.JAXRSBeanValidation callr 인스턴스를 등록해야 하며, 엔드포인트 구성에서 jaxrs:invoker 요소를 사용하여 빈이 올바르게 호출되도록 해야 합니다.

JAX-RS 서비스 라이프사이클 관리에 대한 자세한 내용은 "Spring XML의 라이프사이클 관리" 을 참조하십시오.

65.3.3.5. 빈 유효성 검사 기능을 사용한 샘플 JAX-RS 2.0 구성

다음 XML 예제에서는 jaxrsValidationFeature bean을 JAX-RS 기능으로 추가하고 exceptionMapper 빈을 JAX-RS 공급자로 추가하여 JAX-RS 2.0 끝점에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxrs:server address="/">
   <jaxrs:serviceBeans>
   ...
   </jaxrs:serviceBeans>
   <jaxrs:providers>
      <ref bean="exceptionMapper"/>
   </jaxrs:providers>
   <jaxrs:features>
      <ref bean="jaxrsValidationFeature" />
   </jaxrs:features>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>
<bean id="jaxrsValidationFeature" class="org.apache.cxf.validation.JAXRSBeanValidationFeature">
```

```
    <property name="provider" ref="beanValidationProvider"/>
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

HibernateValidationProviderResolver 클래스의 샘플 구현은 *"HibernateValidationProviderResolver 클래스의 예"* 을 참조하십시오. **OSGi 환경(Apache Karaf)**의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.

참고

컨텍스트에 따라 **jaxrs** 접두사를 **Blueprint** 또는 **Spring**의 적절한 **XML** 네임스페이스에 매핑해야 합니다.

65.3.3.6. 일반적인 빈 유효성 검사 1.1 인터셉터

빈 유효성 검사의 구성을 보다 세밀하게 제어하려면 빈 유효성 검사 기능을 사용하는 대신 **JAX-RS** 인터셉터를 개별적으로 설치할 수 있습니다. 다음 **JAX-RS** 인터셉터 중 하나 또는 둘 다를 구성합니다.

org.apache.cxf.validation.JAXRSBeanValidationInInterceptor

JAX-RS 2.0 서버 끝점에 설치할 때 유효성 검사 제약 조건에 대해 리소스 메서드 매개 변수의 유효성을 검증합니다. 유효성 검사가 실패하면 **javax.validation.ConstraintViolationException** 예외를 발생시킵니다. 이 인터셉터를 설치하려면 **XML**의 **jaxrs:inInterceptors** 하위 요소를 통해 엔드포인트에 추가합니다.

org.apache.cxf.validation.JAXRSBeanValidationOutInterceptor

JAX-RS 2.0 끝점에 설치되는 경우 유효성 검사 제약 조건에 따라 응답 값을 검증합니다. 유효성 검사가 실패하면 **javax.validation.ConstraintViolationException** 예외를 발생시킵니다. 이 인터셉터를 설치하려면 **XML**의 **jaxrs:inInterceptors** 하위 요소를 통해 엔드포인트에 추가합니다.

65.3.3.7. 빈 유효성 검사 인터셉터를 사용한 샘플 JAX-RS 2.0 구성

다음 **XML** 예제에서는 관련 **In** 인터셉터 빈 및 **Out** 인터셉터 빈을 서버 엔드포인트에 명시적으로 추가하여 **JAX-RS 2.0** 끝점에서 빈 유효성 검사 기능을 활성화하는 방법을 보여줍니다.

```
<jaxrs:server address="/">
    <jaxrs:inInterceptors>
        <ref bean="validationInInterceptor" />
    </jaxrs:inInterceptors>
```

```
    <jaxrs:outInterceptors>
      <ref bean="validationOutInterceptor" />
    </jaxrs:outInterceptors>

    <jaxrs:serviceBeans>
    ...
    </jaxrs:serviceBeans>

    <jaxrs:providers>
      <ref bean="exceptionMapper"/>
    </jaxrs:providers>
</jaxrs:server>

<bean id="exceptionMapper" class="org.apache.cxf.jaxrs.validation.ValidationExceptionMapper"/>

<bean id="validationInInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationInInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="validationOutInterceptor"
class="org.apache.cxf.jaxrs.validation.JAXRSBeanValidationOutInterceptor">
    <property name="provider" ref="beanValidationProvider" />
</bean>

<bean id="beanValidationProvider" class="org.apache.cxf.validation.BeanValidationProvider">
    <constructor-arg ref="validationProviderResolver"/>
</bean>

<bean id="validationProviderResolver" class="org.example.HibernateValidationProviderResolver"/>
```

HibernateValidationProviderResolver 클래스의 샘플 구현은 **"HibernateValidationProviderResolver 클래스의 예"** 을 참조하십시오. **OSGi** 환경**(Apache Karaf)**의 컨텍스트에서 **beanValidationProvider** 를 구성하는 데만 필요합니다.

65.3.3.8. BeanValidationProvider 구성

"BeanValidationProvider 구성" 에 설명된 대로 사용자 정의 **BeanValidationProvider** 인스턴스를 검증 인터셉터에 삽입할 수 있습니다.

65.3.3.9. JAXRSParameterNameProvider 구성

org.apache.cxf.jaxrs.validation.JAXRSParameterNameProvider 클래스는 **JAX-RS 2.0** 엔드포인트 컨텍스트에서 메서드 및 생성자 매개변수 이름을 제공하는 데 사용할 수 있는 **javax.validation.ParameterNameProvider** 인터페이스의 구현입니다.