System Integration II: SOAP, WS-* Web Services & Spring-WS

PA165 Enterprise Java

Bruno Rossi

Webservices & WSDL

W3C Definition of Web Services

A Web service is a software system designed to support **interoperable machine-to-machine interaction over a network**. It has an interface described in a **machine processable format** (specifically **WSDL**). Other systems interact with the Web service in a manner prescribed by its description using **SOAP messages**, typically conveyed using **HTTP with an XML** serialization in conjunction with **other Web-related standards**.

Web Service Description Language (WSDL)

- The Web Service Description Language (WSDL) is a **technical description of a Web Service**
- It mentions all **interfaces available**, with the relevant information for the **invocation** (parameters, return type...)

It is possible to generate:

- the client code for accessing the Web Service
- A WSDL file from Java source code
- A Java source code skeleton from WSDL file

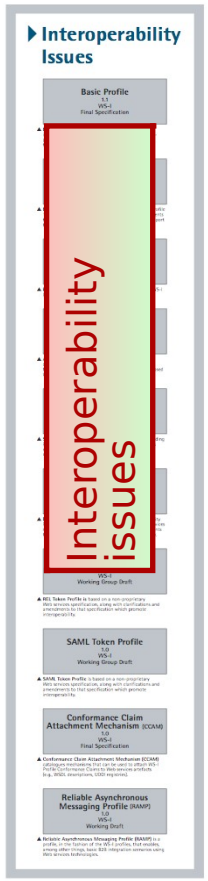
Thomas Erl's definition

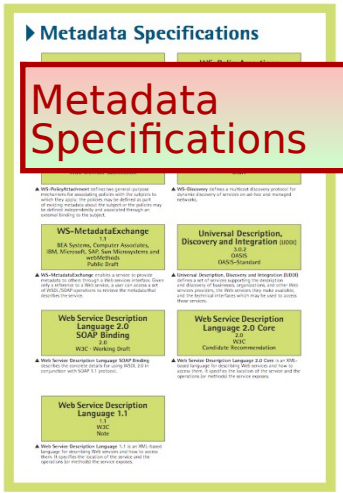
What are WS-* specifications

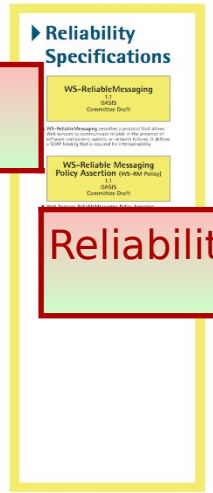
- The term "WS-*" has become a commonly used abbreviation that refers to the **second-generation Web services specifications**. These are **extensions to the basic Web services framework** established by first generation standards represented by WSDL, SOAP, and UDDI.
- The term "WS-*" became popular because the majority of titles given to second-generation Web services specifications have been prefixed with "WS-".

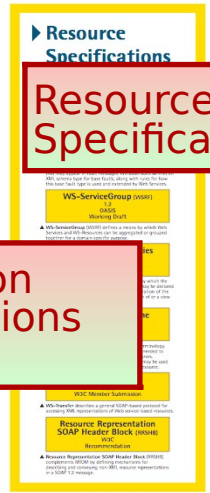
Thomas Erl's definition

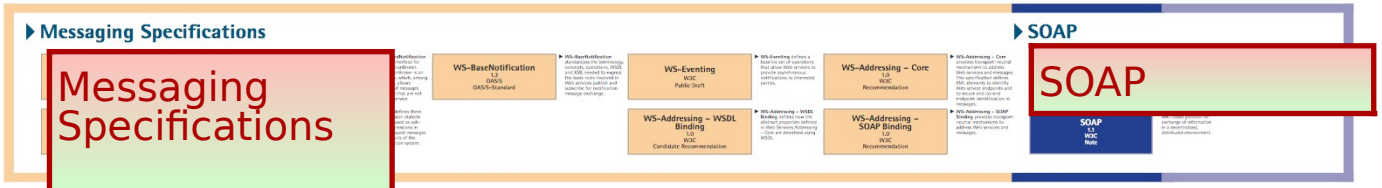
Web Services Standards Overview

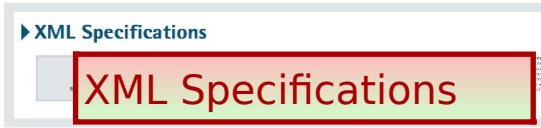


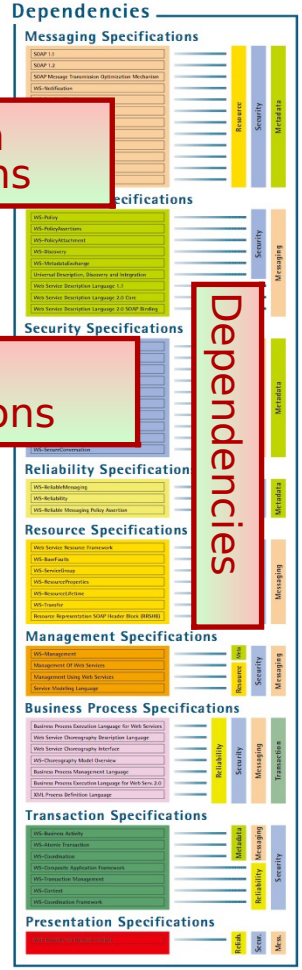












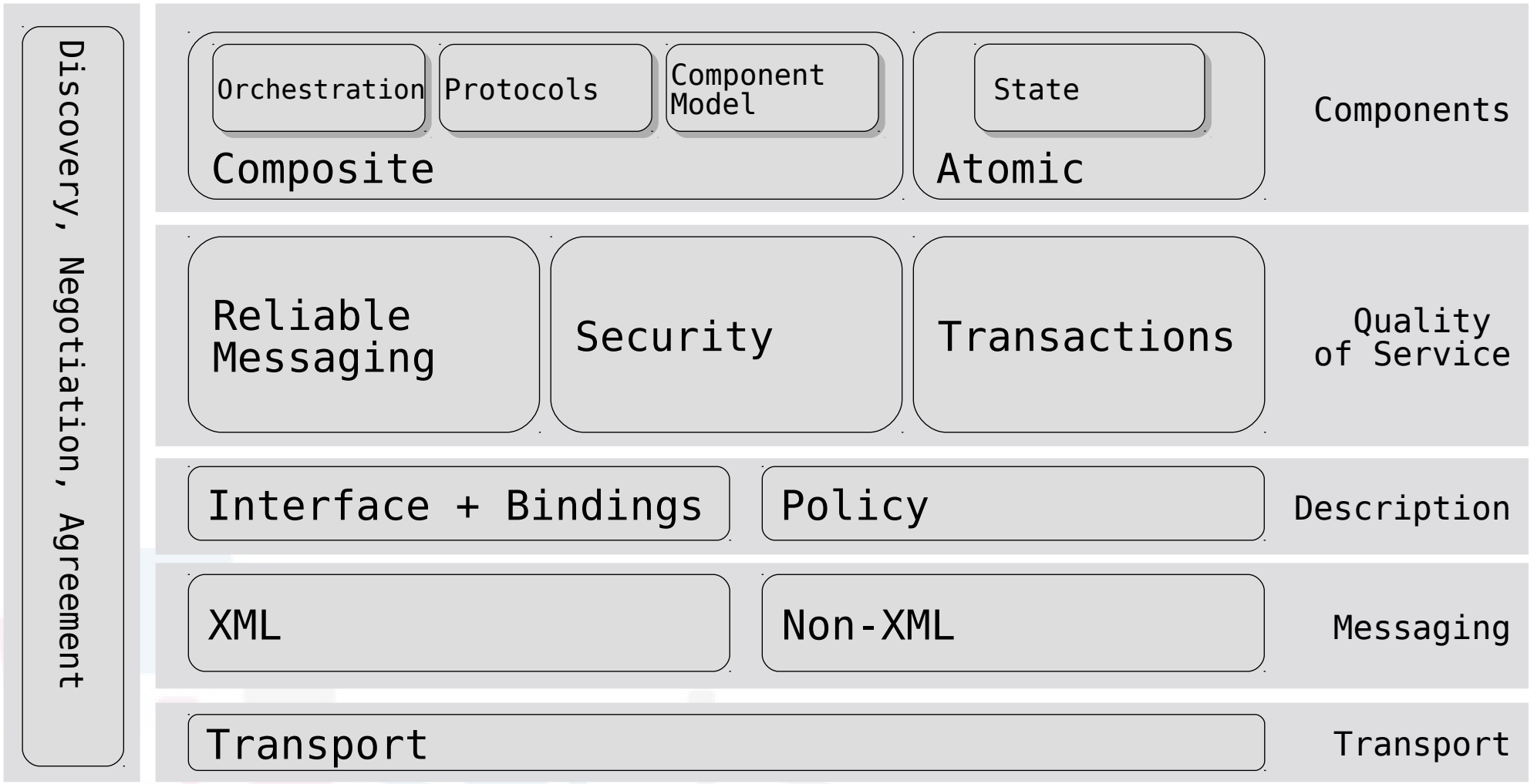
innoQ Deutschland GmbH
Halskestraße 17
D-40880 Ratingen
Phone +49 2102 77162-100
info@innoq.com · www.innoq.com

innoQ Schweiz GmbH
Gewerbestrasse 11
CH-6330 Cham
Phone +41 41 743 01 11

Version 3.0 · February 2007

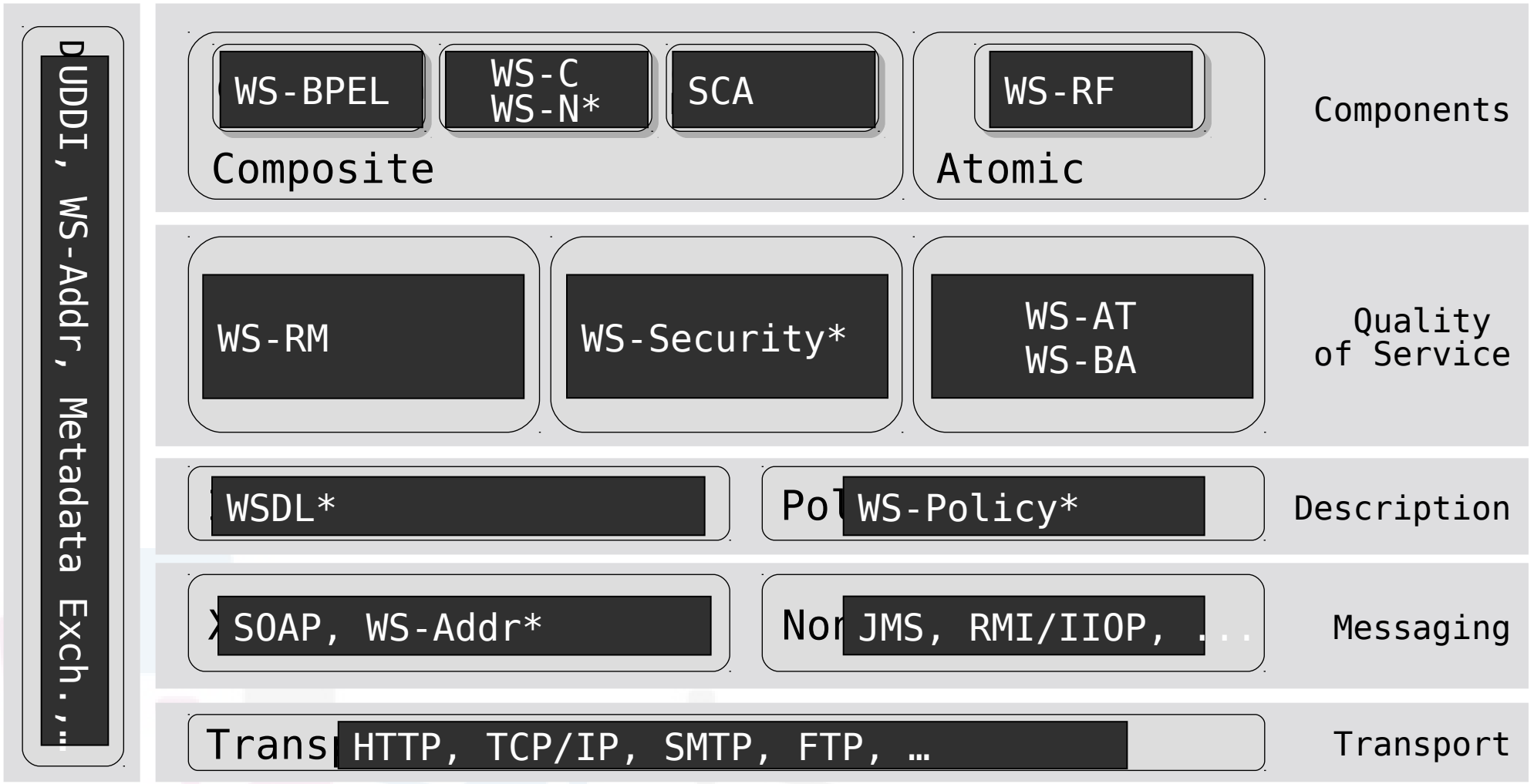
Web Services Standards for SOA

The Web Services Platform Architecture



Web Services Standards for SOA

The Web Services Platform Architecture



SOAP
(Simple Object Access Protocol)

SOAP, in general terms

- Acronym for **Simple Object Access Protocol**
- Nowadays it refers more to a specification, so it has lost the original meaning
- Provides a **communication protocol** for data transport for webservices
- Exchanges **complete documents** or **call a remote procedure**
- Is **platform, language, and protocol independent**

An historical overview:

https://kore.fi.muni.cz/wiki/index.php/PA165/WebServices_(English)

XML (Extensible Markup Language)

- **Sets of rules for encoding documents** to **structure**, **store**, and **transport** data in a convenient way
- **Human-readable** and **machine-readable** format
- XML 1.0 Specification produced by the W3C
- two current versions of XML.
  - XML 1.0, currently in its fifth edition, still recommended for general use
  - XML 1.1, not very widely implemented and is recommended for use only by those who need its unique features

Schema and Validation

- **Well-formed** (compliant to XML standard) vs **valid** (compliant to DTD)
  - Document contains a reference to DTD,
  - DTD declares elements and attributes, and specifies the grammatical rules
- XML processors
  - re validating or non-validating
  - If error discovered it is reported, but processing may continue normally
- schema languages constrain
  - the **set of elements in a document**,
  - **attributes that are applied** to them,
  - the **order** in which they appear,
  - the **allowable parent/child relationships**

XML Schema: **XSD (XML Schema Definition)**

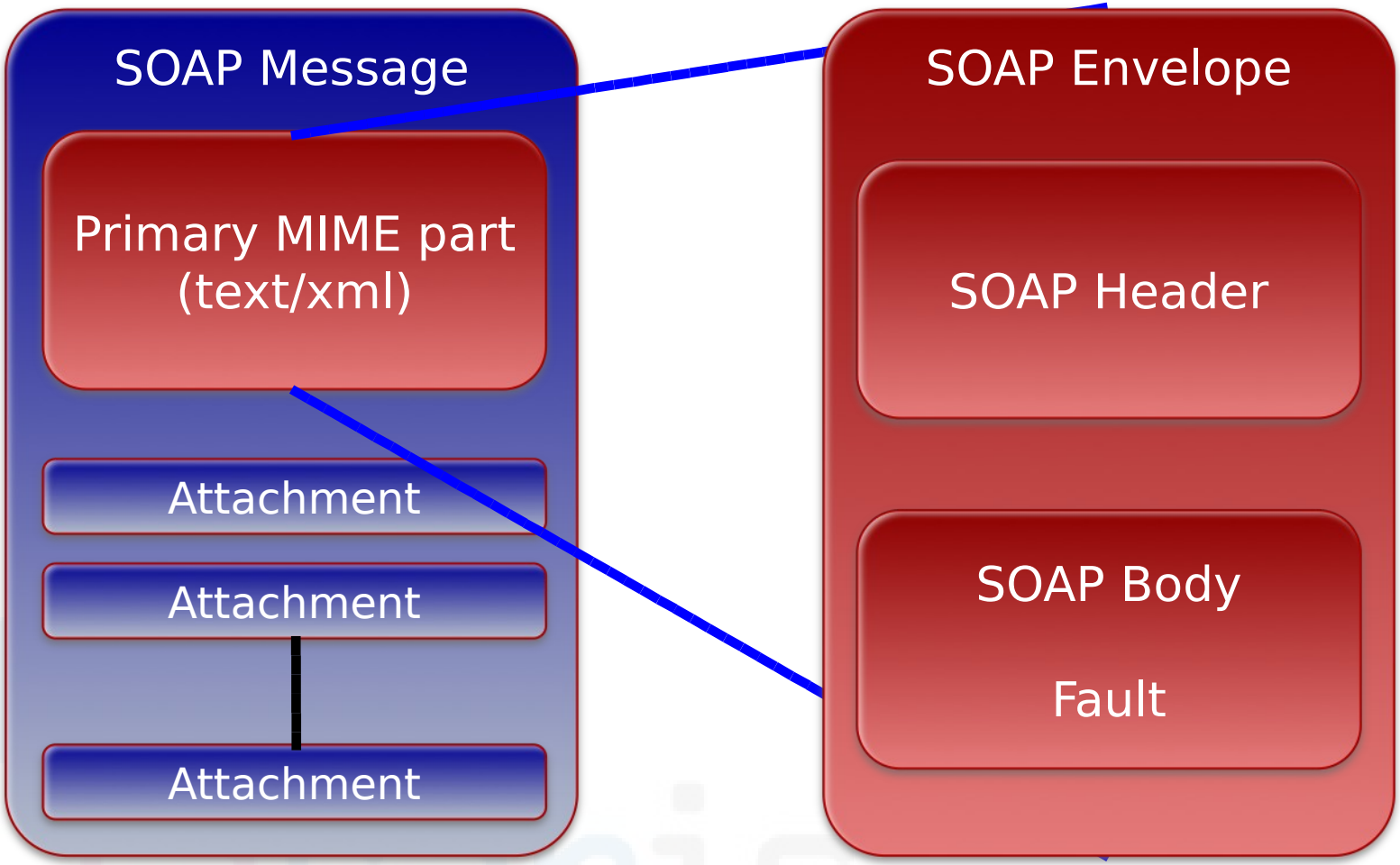
- schema language, described by the W3C
  - **(successor of DTD = Document Type Definition)**
  - XML schema is more powerful than DTDs

- XSDs use an XML-based format, so XML tools can be used process them.

XML Messaging

- SOAP 1.1 defined:
  - An **XML envelope for XML messaging**:
    - Headers + body.
  - An **HTTP binding for SOAP messaging**:
    - SOAP is “transport independent”.
- A **convention for doing RPC**
- An **XML serialization format for structured dat**a.
  - SOAP Attachments: How to carry and reference data attachments

SOAP Message



SOAP Message Envelope

Encoding information

Header

- Optional
- Contains context knowledge
  - Security
  - Transaction

Body

- Methods and parameters
- Contains application data

A SOAP Request

```
POST /temp HTTP/1.1
Host: www.somewhere.com
Content-Type: text/xml; charset="utf-8"
Content-Length: xxx
SOAPAction: "http://www…../temp"

<?xml version="1.0"?>
……..
```

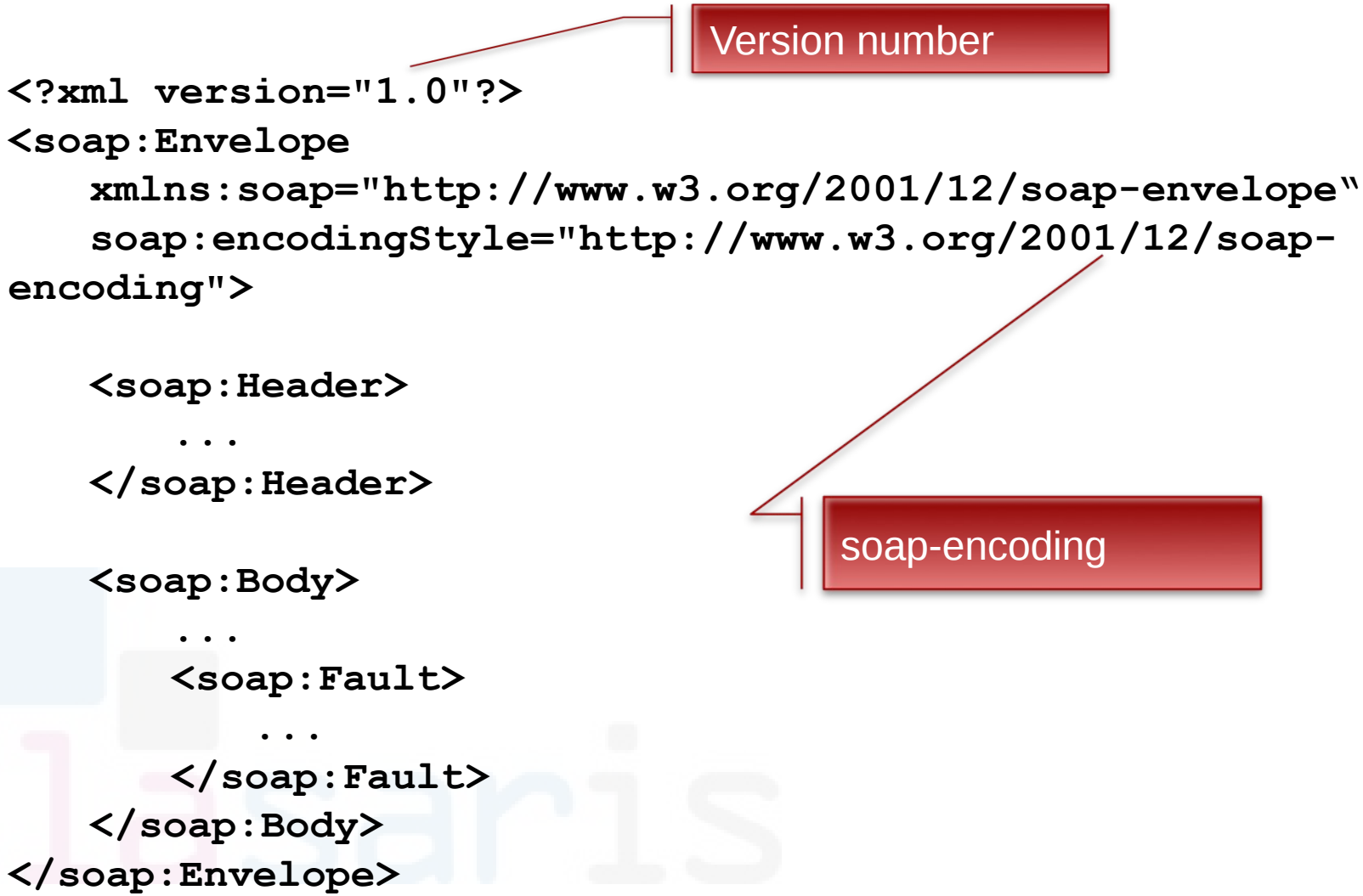
HTTP headers and the blank line

an XML document

"The SOAPAction HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client MUST use this header field when issuing a SOAP HTTP Request."
Note: in SOAP 1.2, the SOAPAction header has been replaced with the "action" attribute on the application/soap+xml media type (Content-Type: application/soap+xml; charset=utf-8). But it works almost exactly the same way as SOAPAction.

Source: Simple Object Access Protocol (SOAP) 1.1 specifications

XML message structure




```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-
encoding">

    <soap:Header>
        ...
    </soap:Header>

    <soap:Body>
        ...
        <soap:Fault>
            ...
        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```

SOAP Encoding

- When SOAP specification was written for the first time, **XMLSchema was not available, so a common way to describe messages was defined**.
- Now SOAP encoding defines it's own namespace as http://schemas.xmlsoap.org/soap/encoding/ and a set of rules to follow.
- Rules of expressing application-defined data types in XML
- Based on W3C XML Schema
- Simple values
  - `Built-in types from XML Schema, Part 2 (simple types, enumerations, arrays of bytes)`
- Compound values
  - `Structures, arrays, complex types`

SOAP Header (1/3)

- Allows to specify non-body related information
- For example if some node is the receiver, how intermediary nodes might deal with the message, etc...

```
<SOAP-ENV:Header>
        <ns:timeAlive value="3600" xmlns:ns="http://muni.fi.cz/pa165/ws"
</SOAP-ENV:Header>
```

SOAP Header (2/3)

- Some interesting attributes that are specified in SOAP specifications are **mustUnderstand** and **role**
- **MustUnderstand = "true"** means that if a node does not understand the header element with such attribute → must send a SOAPFault
- **Role**: only a node with the specified role can deal with the header element – other nodes do not need to process

```
<SOAP-ENV:Header>
        <ns:timeAlive value="3600" xmlns:ns="http://muni.fi.cz/pa165/ws"
         role="http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
</SOAP-ENV:Header>
```

See http://www.w3.org/TR/2003/REC-soap12-part1-20030624/#soaproles

SOAP Header (3/3)

- Relay: whether the element needs to be kept when the message is forwarded even if it has been processed by one node

```
<SOAP-ENV:Header>
        <ns:timeAlive value="3600" xmlns:ns="http://muni.fi.cz/pa165/ws"
         relay="true"
</SOAP-ENV:Header>
```

WS-Addressing (1/2)

→ WS-* specifications are inserted on top of SOAP messaging

→ For example, looking at SOAP, there is no knowledge about where the message is going, or how to return the response or where to post an error message → this can be problematic in case of asynchronous communication

→ WS-Addressing adds this information to the SOAP envelope

See https://jax-ws.java.net/nonav/jax-ws-21-ea2/docs/why-wsaddressing.html

WS-Addressing (2/2)

Example

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://www.w3.org/2004/12/addressing">
<soap:Header>
        <wsa:MessageID>
            UniqueMessageIdentifier
        </wsa:MessageID>
        <wsa:ReplyTo>
          <wsa:Address>http://somereceiving.client</wsa:Address>
        </wsa:ReplyTo>

        <wsa:FaultTo>
            <wsa:Address>http://somereceiving.server/ErrorHandler</wsa:Address>
        </wsa:FaultTo>

        <wsa:To>http://somereceiving.server/HandlerURI </wsa:To>
        <wsa:Action>
         http://somereceiving.server/ACTION
     </wsa:Action>
    </soap:Header>

    <soap:Body>
     <!-- SOAP Request as usual here -->
    </soap:Body>
```

Summing up

- SOAP, originally defined as **Simple Object Access Protocol**, is a protocol specification that is used to exchange information in a structured way – the protocol is used for implementation of **Web Services as it builds on top of an Application Layer protocol, Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP)**.
- SOAP is based on on Extensible Markup Language (XML) for its message format.
- SOAP is usually the **foundation layer of a web services protocol stack**, to provide a basic messaging framework.

Some Remarks

- SOAP is not *“what it used to be”*, the name remained, but the content has changed
- **SOAP term** is often used as **synonym** for **WS* web service architecture**, although **it is one element of it**
- **SOAP is not just one element of WS***, it is used in other context as well, even in parallel with REST web services.
- **SOAP is often hidden from the developer**, build into tools in such a way that developer does not have to deal with it at a detailed level.

SOAP vs REST (1/2)

REST

- **Easy** to develop
- **Using existing infrastructure** (and based on HTTP)
- **Little learning** required

SOAP

- More an **industry standard** than an architectural style
- **More overhead** but more **protocol independence**
- Based on a **specification**

SOAP vs REST (2/2)

Best used in these cases:

REST

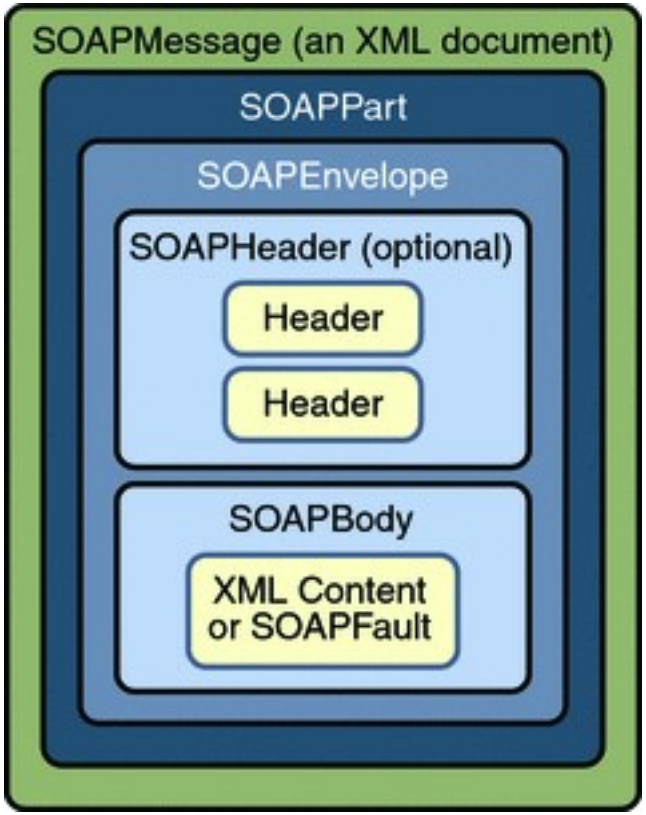
- **Limited bandwidth** and **resources**
- **Stateless** operations
- Opportunities for **caching**

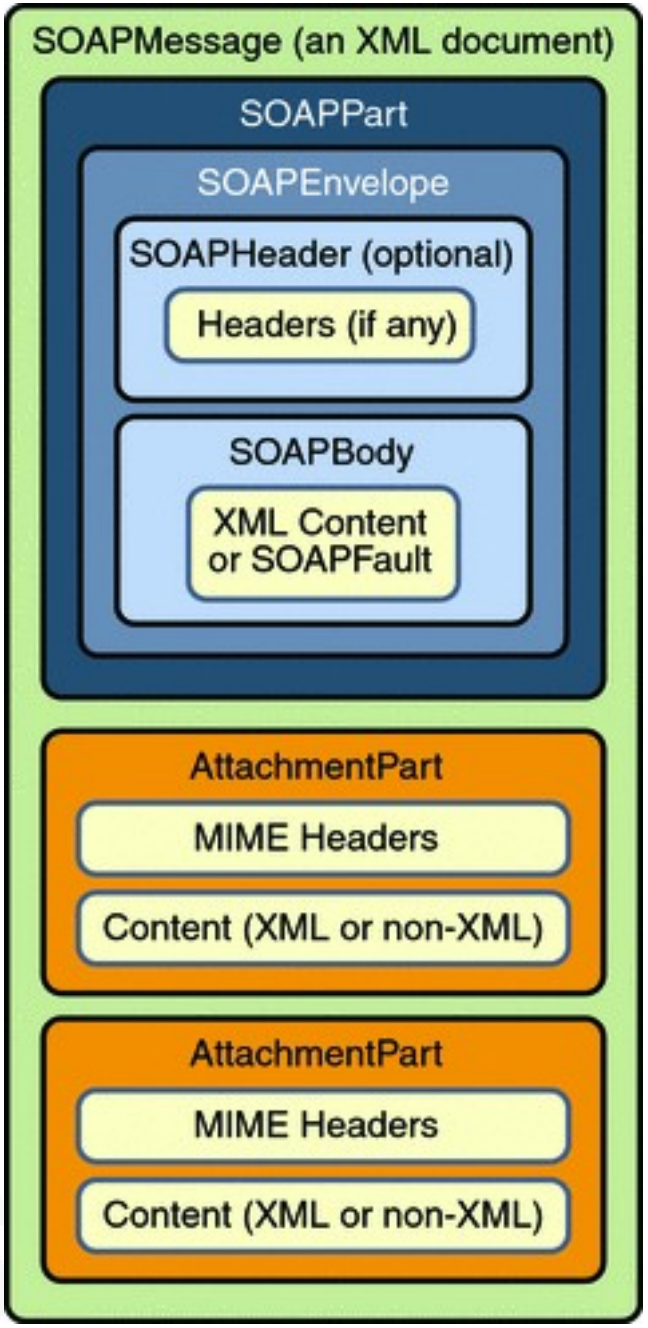
SOAP

- Asynchronous processing & invocation
- Need for **formal contracts**
- **Stateful** operations

SOAP with Attachments,
SOAP with Attachments API for Java (SAAJ)

- **SOAP with Attachments (SwA)** or MIME for Web Services refers to the method of using Web Services to send and receive files using a combination of SOAP and MIME, primarily over HTTP.
- Note that SwA is not a new specification, but rather a **mechanism for using the existing SOAP and MIME facilities to perfect the transmission of files using Web Services invocations**.
- The **SOAP with Attachments API for Java** or **SAAJ** provides a standard way to send XML documents over the Internet from the Java platform.
- **SAAJ** enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments.
- Developers can also use it to **write SOAP messaging applications directly instead of using JAX-RPC (obsolete) or JAX-WS**





SOAP with Attachments API for Java
The Java EE 5 Tutorial
http://docs.oracle.com/javaee/5/tutorial/doc/bnbhf.html

Creating a SOAP Connection

```
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.MessageFactory;
……..

public SimpleSAAJ {
    public static void main(String args[]) {
        try {
            //Create a SOAPConnection
            SOAPConnectionFactory factory =
                    SOAPConnectionFactory.newInstance();

            SOAPConnection connection =
                    factory.createConnection();

            ..................
            // Close the SOAPConnection
            connection.close();

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
  }
}
```

Creating a SOAP Message

```
.....
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPBody;
import java.net.URL;
.....
```

```
//Create a SOAPMessage
SOAPMessageFactory messageFactory =
MessageFactory.newInstance();
SOAPMessage message = messageFactory.createMessage();
SOAPPart  soapPart = message.getSOAPPart();
SOAPEnvelope envelope = soapPart.getEnvelope();
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
header.detachNode();
```

Populate a SOAP Message

```
//Create a SOAPBodyElement
 Name bodyName = envelope.createName("GetElement"
                 "n", "http://localhost");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);

//Insert Content
  Name name = envelope.createName("symbol");
  SOAPElement symbol = bodyElement.addChildElement(name);
  symbol.addTextNode("Smith");
```

This will produce the SOAP envelope:

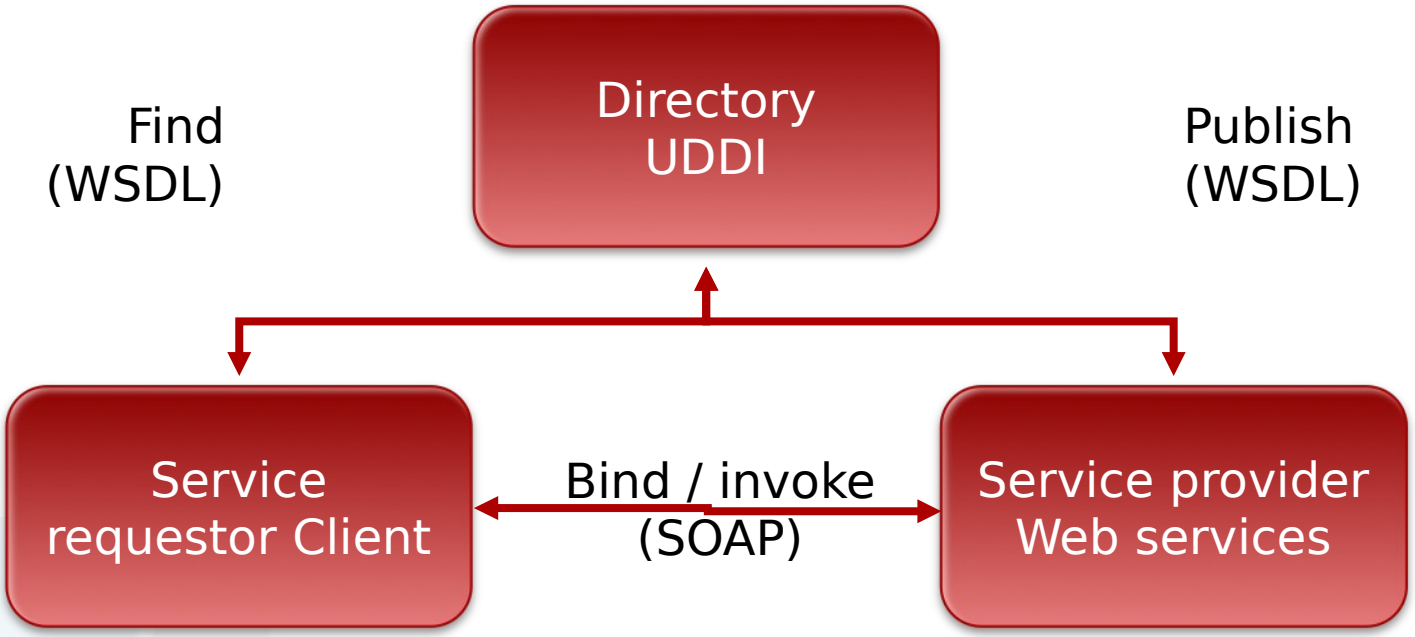
```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <n:GetElement xmlns:n="http://localhost">
      <symbol>Smith</symbol>
    </n:GetElement>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

That you can send with

```
java.net.URL endpoint = new URL("localhost/addr");
SOAPMessage response = connection.call(message, endpoint);
```

Invoking Webservices

Use of web services



SOAP, WSDL, UDDI, and XML in all of them

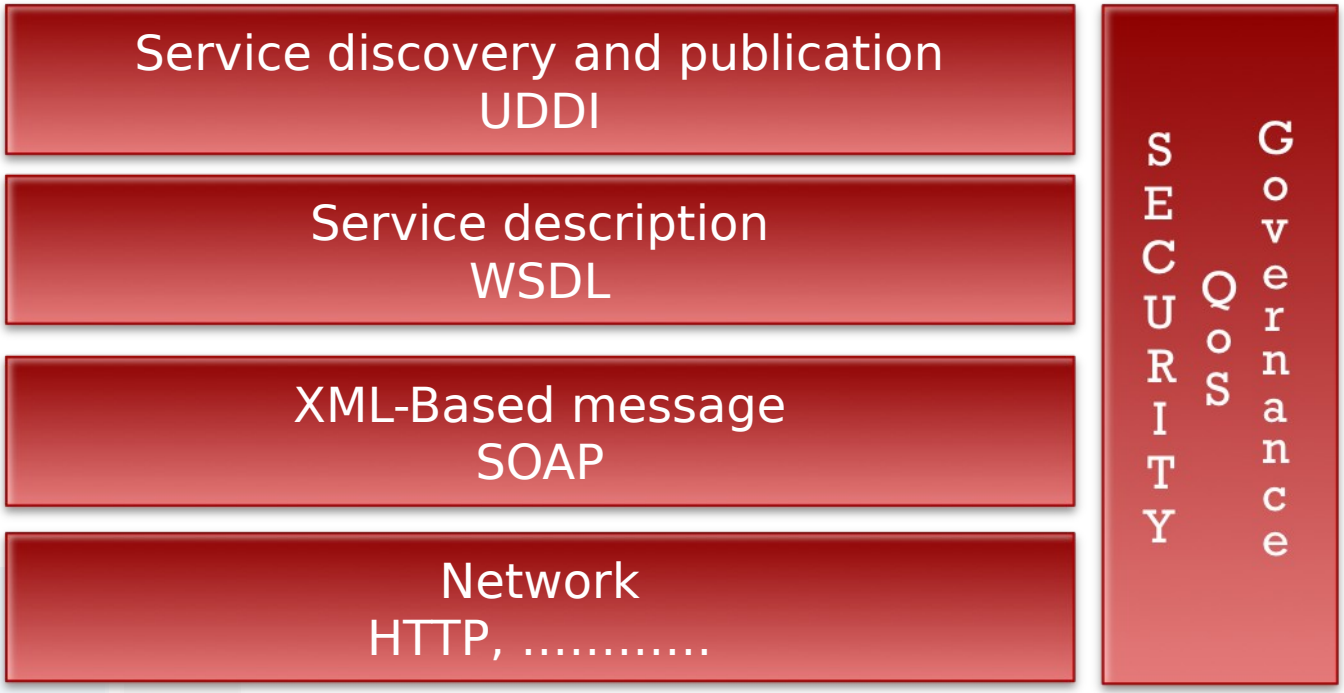
UDDI (Universal Description, Discovery and Integration)

- UDDI is a **platform-independent, Extensible Markup Language (XML)-based registry** by which businesses worldwide can list themselves, plus a mechanism to register and locate web service applications.
- It is a standard supported by the Organization for the Advancement of Structured Information Standards (OASIS)
- In the original plans for the discoverability of web services, a central role should have been played by UDDI

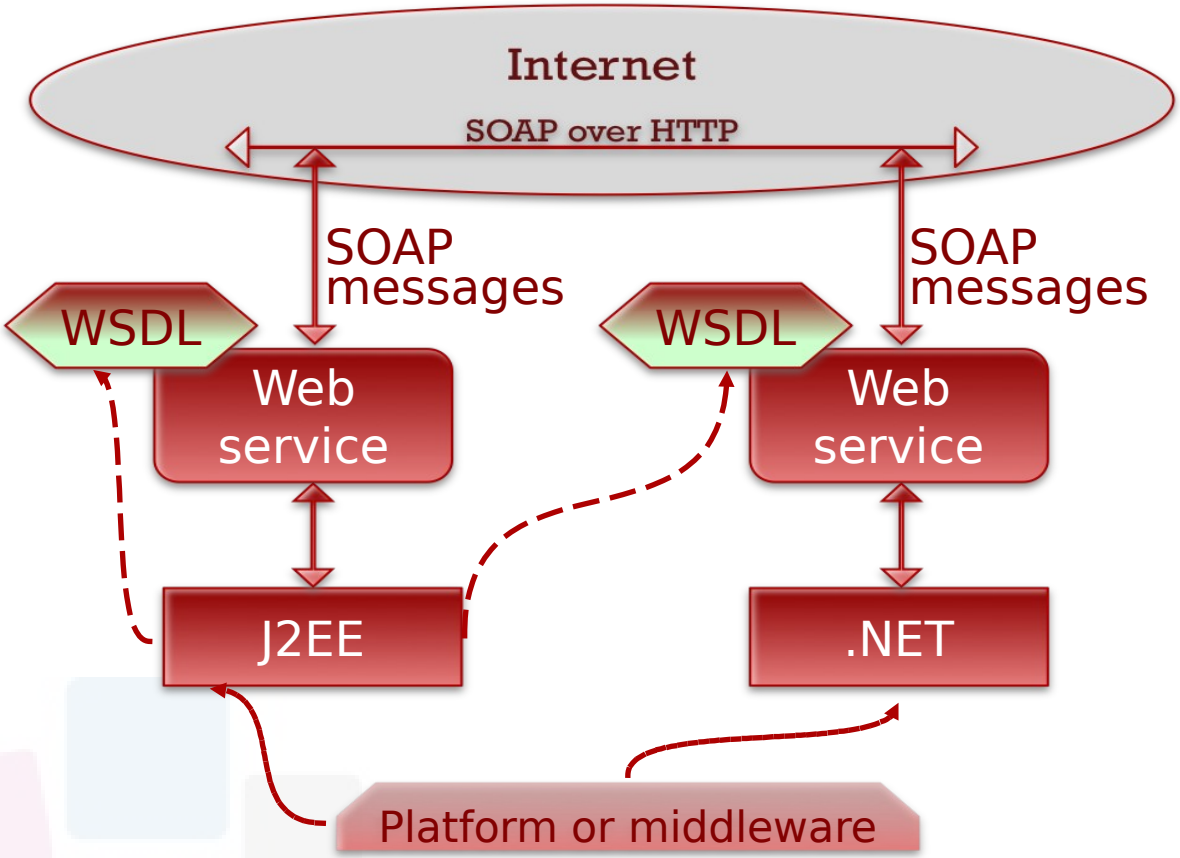
Public Registries *(well, it used to be...)*

- IBM Registration: https://uddi.ibm.com/ubr/registry.html
  - inquiryURL= https://uddi.ibm.com/ubr/inquiryapi
  - publishURL = https://uddi.ibm.com/ubr/publishapi
- UDDI has not been as successful as its creators had expected. IBM, Microsoft, and SAP **closed** their public UDDI nodes in 2006.
- The OASIS UDDI Specification Technical Committee has been **dismantled** as well.
- Microsoft **removed** UDDI services from the Windows Server operating system.
- UDDI systems **are most commonly found inside companies**, where they are used to dynamically bind client systems to implementations. However, much of the more advanced functionalities are not used.

Enabling technologies



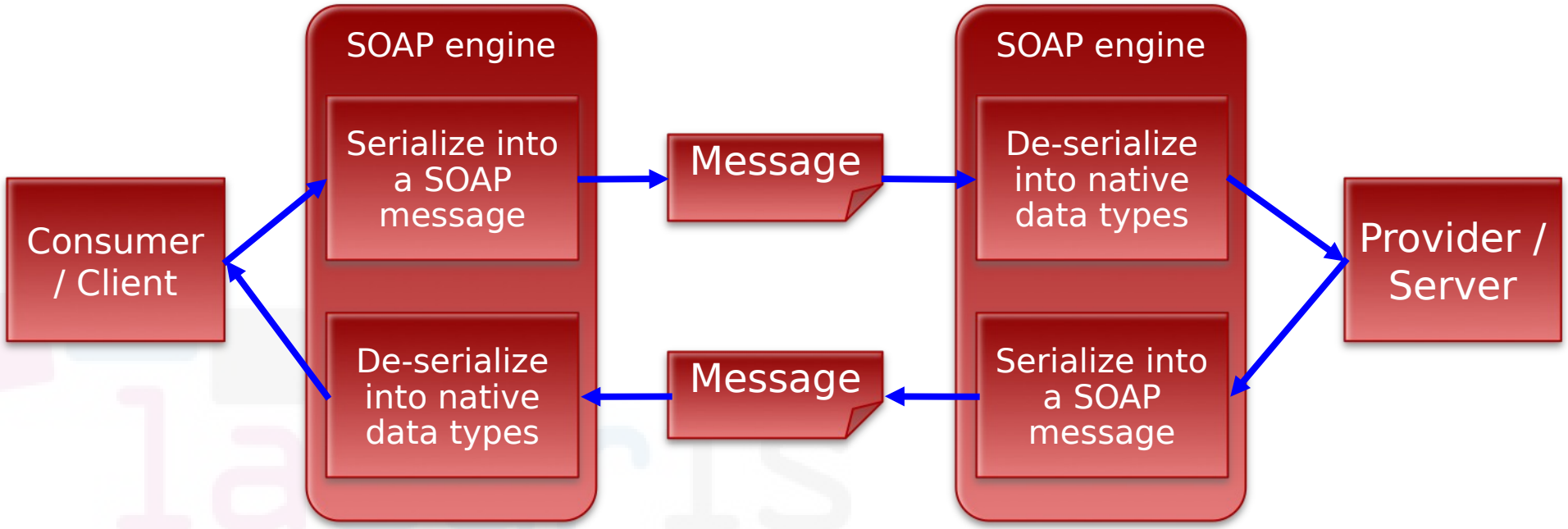
WS*

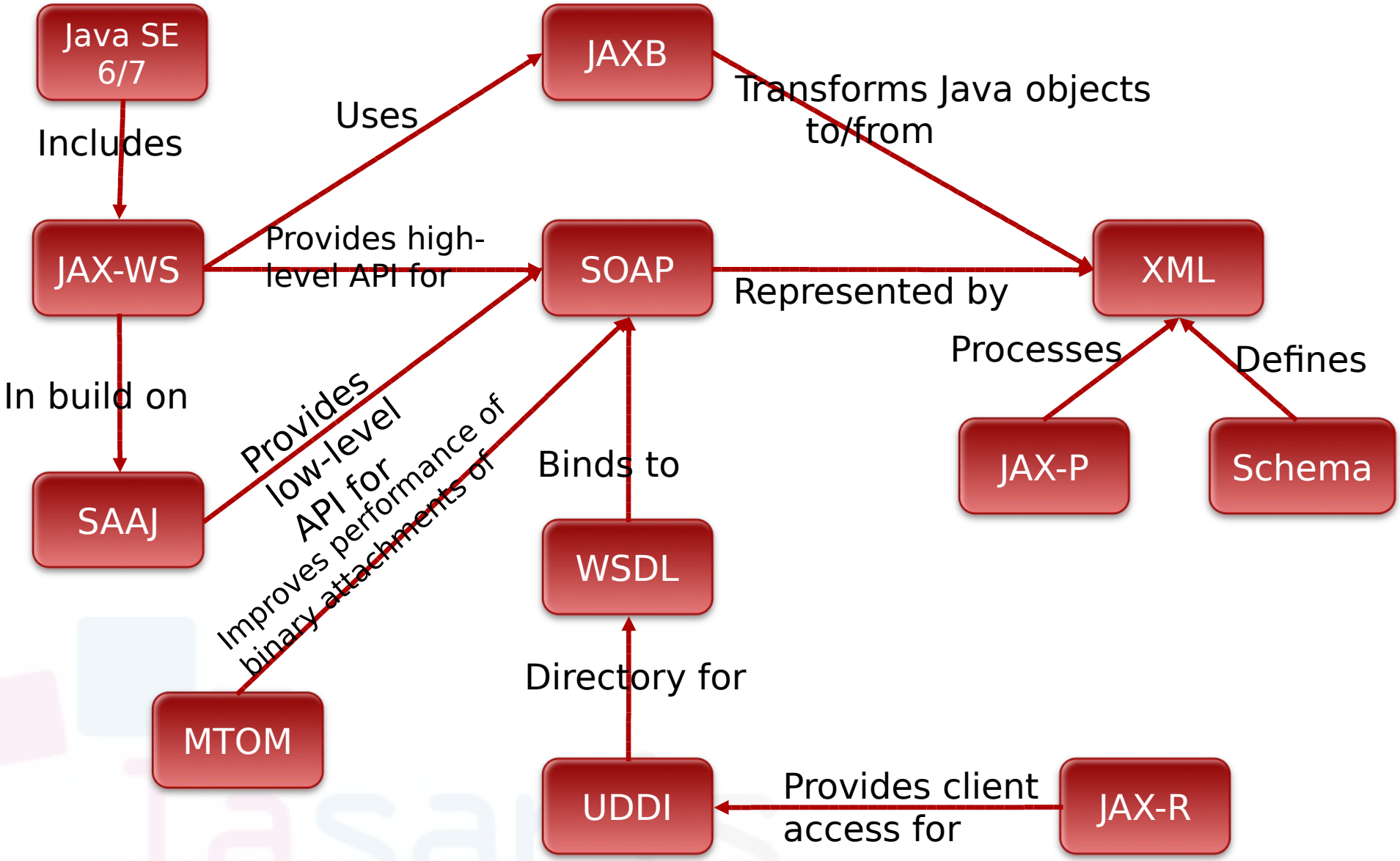


- clear specifications of the service interface and the data types in use
- communication protocol independent (platform, programming language)
- interoperability.

SOAP engines

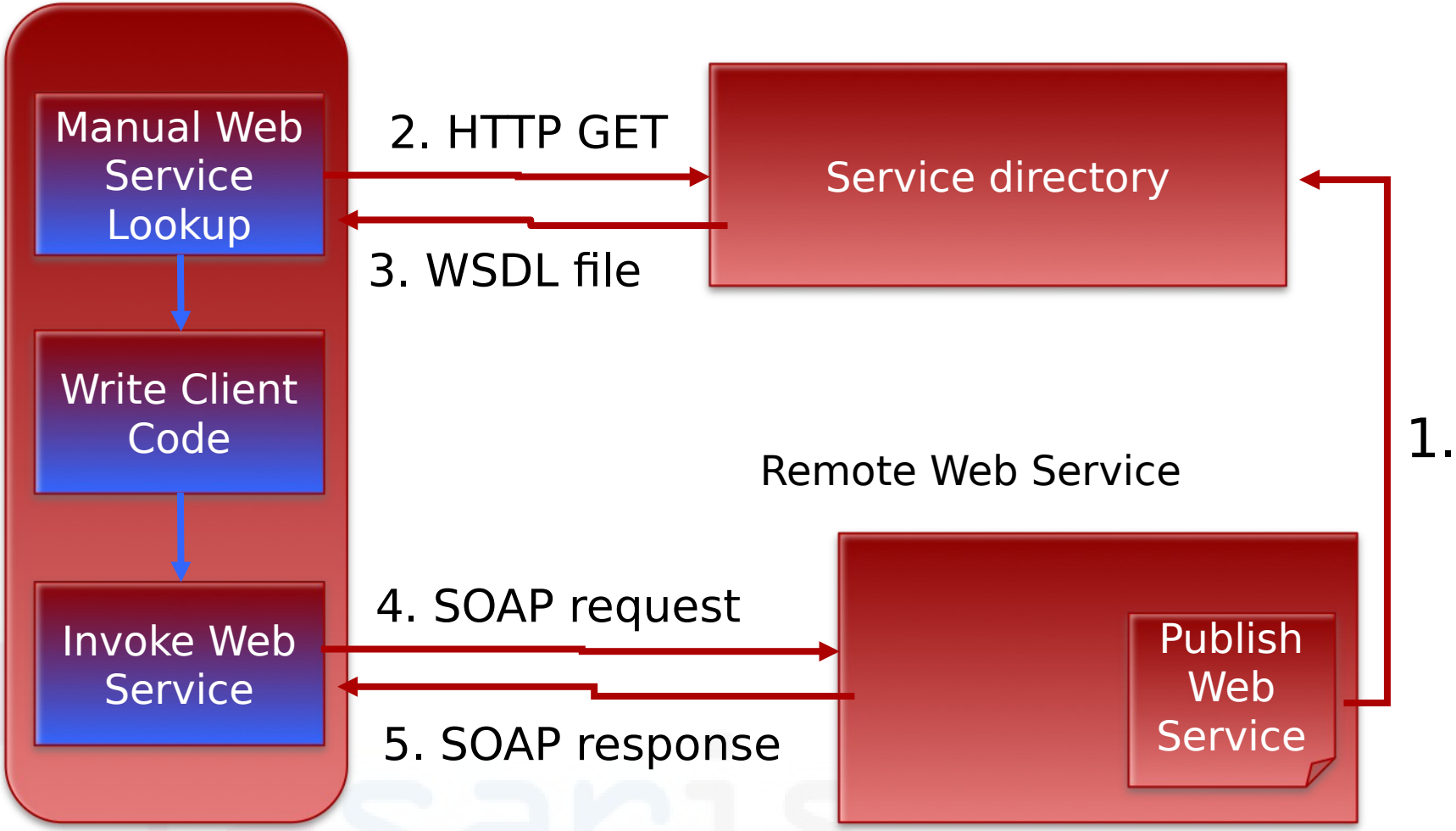
A **SOAP engine** allows to:
1.Serialize objects (from any supported language) into SOAP messages
2.Deserialize SOAP messages back into objects, i.e. create the appropriate data types and populate these with the message content.




Java SE 6/7
Includes
JAX-WS
Uses
JAXB
Transforms Java objects to/from
Provides high-level API for
SOAP
Represented by
XML
In build on
SAAJ
Provides low-level API for
Improves performance of binary attachments of
Binds to
WSDL
Directory for
MTOM
UDDI
Provides client access for
JAX-R
Processes
JAX-P
Defines
Schema

Developing Webservices

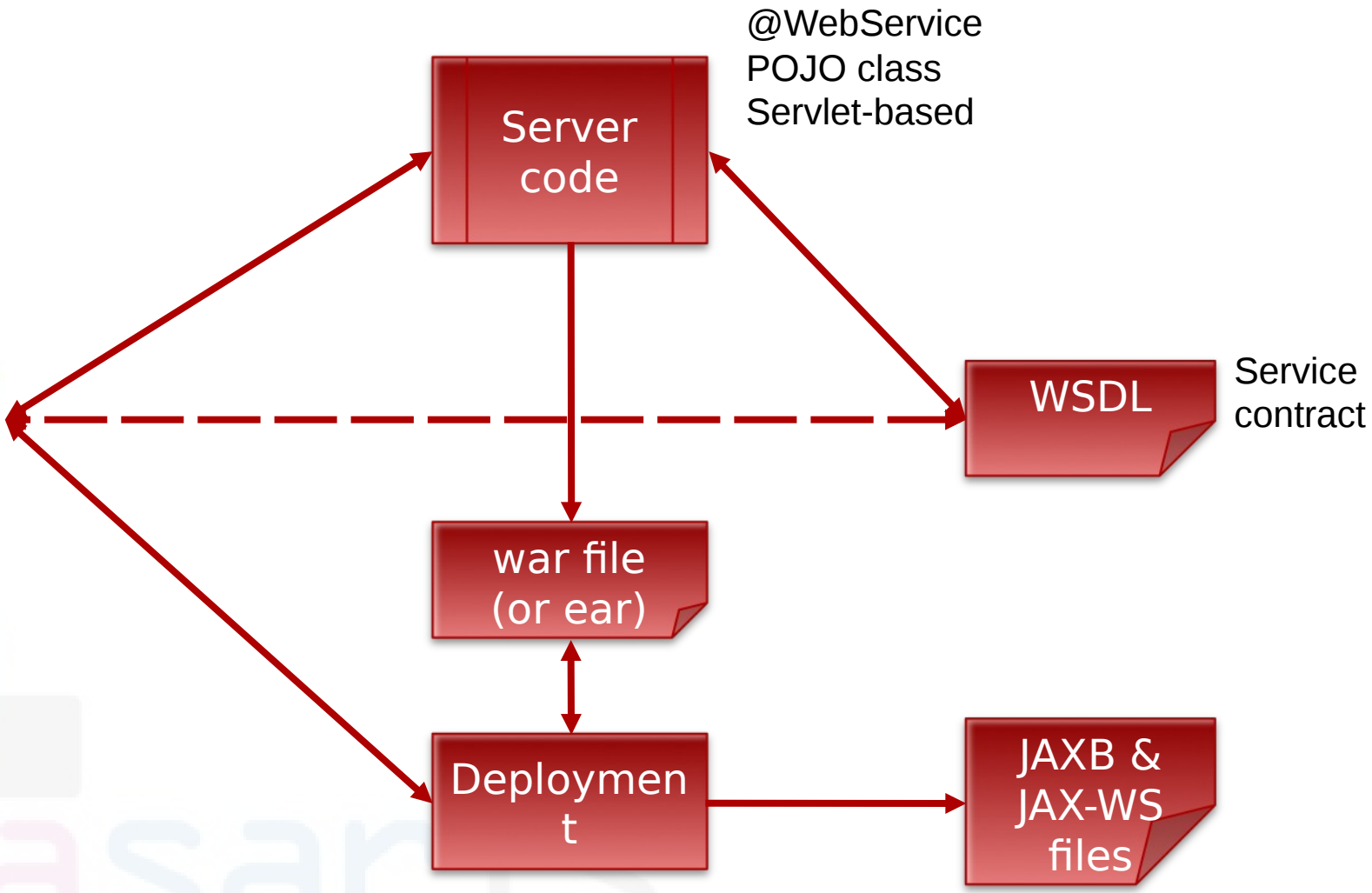
Simple Web Service Invocation



Client-side programming

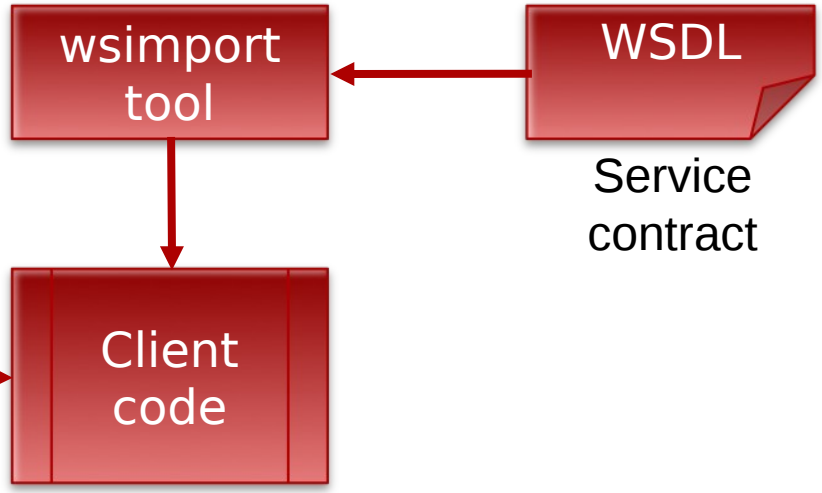
- Usually two ways:
- **Contract last**: first you create the code for your web service, then the contract (WSDL) is generated based on the code
- **Contract first**: you start with the **creation of the contract** for the web service and then source code templates are generated based on the contract

Developing a Web Service



Client-side programming

You develop Client which uses proxy to call Web Service



Generating XSD/Java representations

```java
package cz.fi.muni.pa165.dto;

import java.util.*;

public class ProductDTO
{
    private Long id;
    private byte[] image;
    private String imageMimeType;
    private String name;
    private String description;
    private Color color;
    private Date addedDate;
    private Set<CategoryDTO> categories = new
                                HashSet<>();
    private List<PriceDTO> priceHistory = new
                                ArrayList<>();
    private PriceDTO currentPrice;
    //...
}
```

schemagen *.java →

← xjc *.xsd

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="productDTO">
    <xs:sequence>
      <xs:element name="addedDate" type="xs:dateTime" minOccurs="0"/>
      <xs:element name="categories" type="categoryDTO" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="color">
        <xs:complexType>
          <xs:sequence>
            ...s="0" maxOccurs="unbounded">
            ...ccurs="0" type="xs:anyType"/>
            ...nOccurs="0" type="xs:anyType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="currentPrice" type="priceDTO" minOccurs="0"/>
      <xs:element name="description" type="xs:string" minOccurs="0"/>
      <xs:element name="id" type="xs:long" minOccurs="0"/>
      <xs:element name="image" type="xs:base64Binary" minOccurs="0"/>
      <xs:element name="imageMimeType" type="xs:string" minOccurs="0"/>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
      <xs:element name="priceHistory" type="priceDTO" nillable="true" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="categoryDTO">
    <xs:sequence>
      <xs:element name="id" type="xs:long" minOccurs="0"/>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

→ Why did Spring-WS opt for “Contract-first”? Which are the advantages?

An example (1/7)
Implementing a simple web service with Java

1. Create the “service endpoint interface”
   - `Interface for web service`
2. Create the “service implementation”
   - `Class that implements the service`
3. Create the “service publisher”

- Java supports web services in core Java
  - `JAX-WS (Java API for XML-Web Services)`
- In full production mode, one would use a Java application server such as Tomcat, Glassfish, etc. like we will see later with Spring-WS

An example (2/7)
Service Endpoint Interface

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;

@WebService      // This is a Service Endpoint Interface
@SOAPBinding(style = Style.RPC) // Needed for the WSDL
public interface EchoServer {
    @WebMethod    // This method is a service operation
        String EchoMessage(String strMsg); }
```

See https://docs.oracle.com/javaee/5/tutorial/doc/bnayn.html

An example (3/7)

Service Implementation

```
import javax.jws.WebService;
/**
   * The @WebService property endpointInterface links this class
   * to EchoServer class
*/
@WebService(endpointInterface = "EchoServer")
public class EchoServerImpl implements EchoServer {
      public String EchoMessage(String Msg) {
            String capitalizedMsg;
            System.out.println("Server: EchoMessage() invoked...");
            System.out.println("Server: Message > " + Msg);
            capitalizedMsg = Msg.toUpperCase();
            return(capitalizedMsg);
      }
}
```

An example (4/7)
Service Publisher

```
import javax.xml.ws.Endpoint;

public class EchoServerPublisher {
      public static void main(String[ ] args) {
      Endpoint.publish("http://localhost:8080/ws", new
      EchoServerImpl());
      }
}
```

An example (5/7)
Deploying and testing

1. Compile the Java code
2. Run the publisher
   - `java example.echo.EchoServerPublisher`
3. Testing the web service with a browser
   - `URL: http://localhost:8080/ws?wsdl`

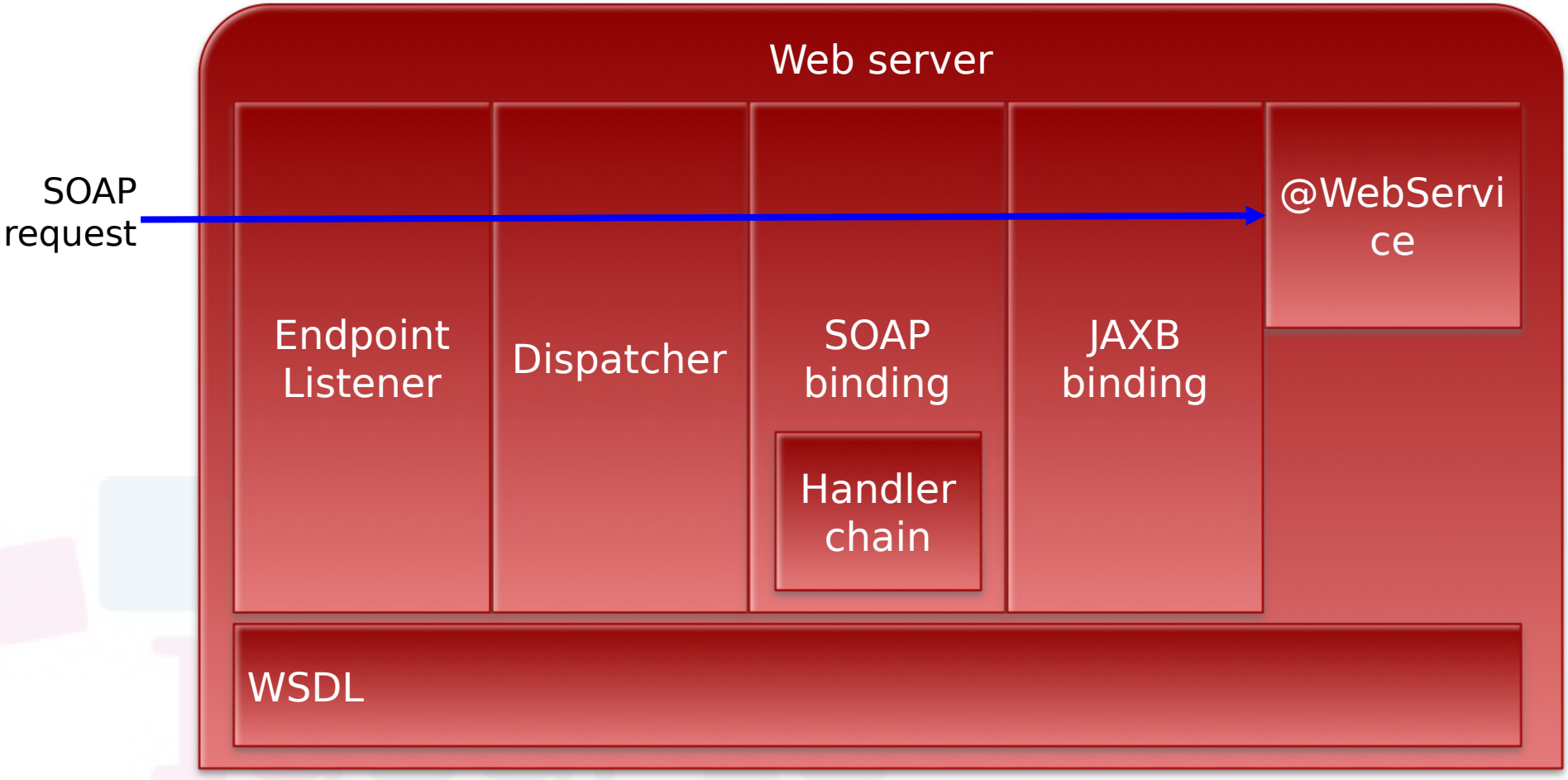
An Example (6/7)
WSDL for echo service

```
<definitions targetNamespace=”http://localhost/" name="EchoServerImplService">
<types/>
<message name="EchoMessage”> <part name="arg0" type="xsd:string"/> </message>
<message name="EchoMessageResponse”><part name="return"
type="xsd:string"/></message>

<portType name="EchoServer">
    <operation name="EchoMessage">
        <input message="tns:EchoMessage"/>
        <output message="tns:EchoMessageResponse"/>
    </operation>
</portType>

<binding name="EchoServerImplPortBinding" type="tns:EchoServer">
    <soap:binding transport=”http://schemas.xmlsoap.org/soap/http" style="rpc”/>
    <operation name="EchoMessage">
    <soap:operation soapAction=""/>
    <input> <soap:body use="literal" namespace="http://my.ws/"/> </input>
    <output> <soap:body use="literal" namespace="http://my.ws/"/> </output>
    inding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
    </operation>
</binding>

<service name="EchoServerImplService">
<port name="EchoServerImplPort" binding="tns:EchoServerImplPortBinding”>
    <soap:address location=”http://localhost:8080/ws"/>
</port>
</service>
</definitions>
```

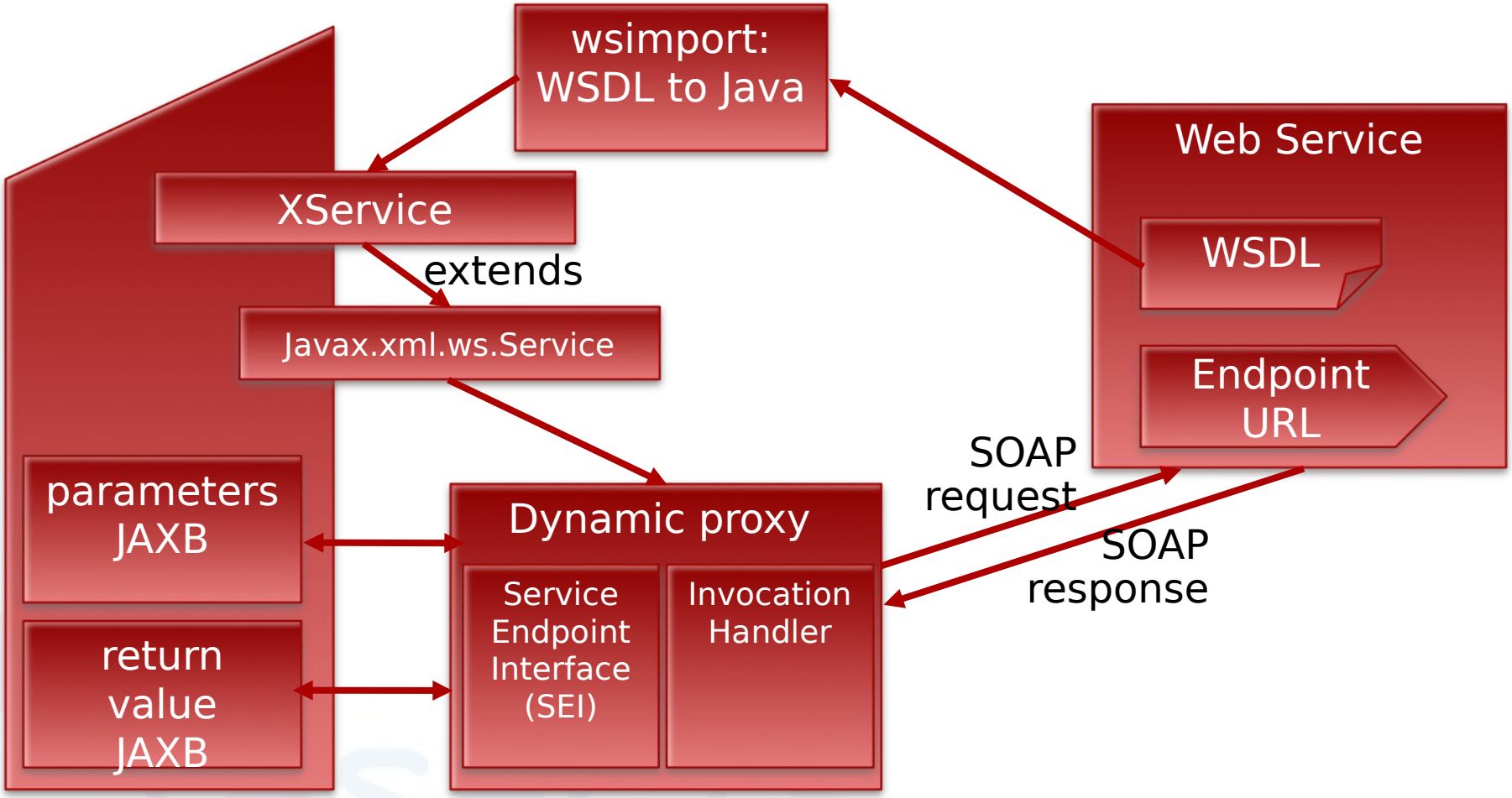
An Example (7/7)
EchoClient

```
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;

class EchoClient {
      public static void main(String argv[ ]) throws Exception {
            if (argv.length < 1) {
            System.out.println("Usage: java EchoClient \"MESSAGE\"");System.exit(1);}

            String strMsg = argv[0];
                  URL url = new URL(”http://localhost:8080/ws?wsdl");
            QName qname = new
            QName(“http://localhost/”,“EchoServerImplService");
                  Service service = Service.create(url, qname);
            EchoServer eif = service.getPort(EchoServer.class);
            System.out.println(eif.EchoMessage(strMsg));
      }
}
```

Server side


Web server
SOAP request
Endpoint Listener
Dispatcher
SOAP binding
Handler chain
JAXB binding
@WebServi ce
WSDL

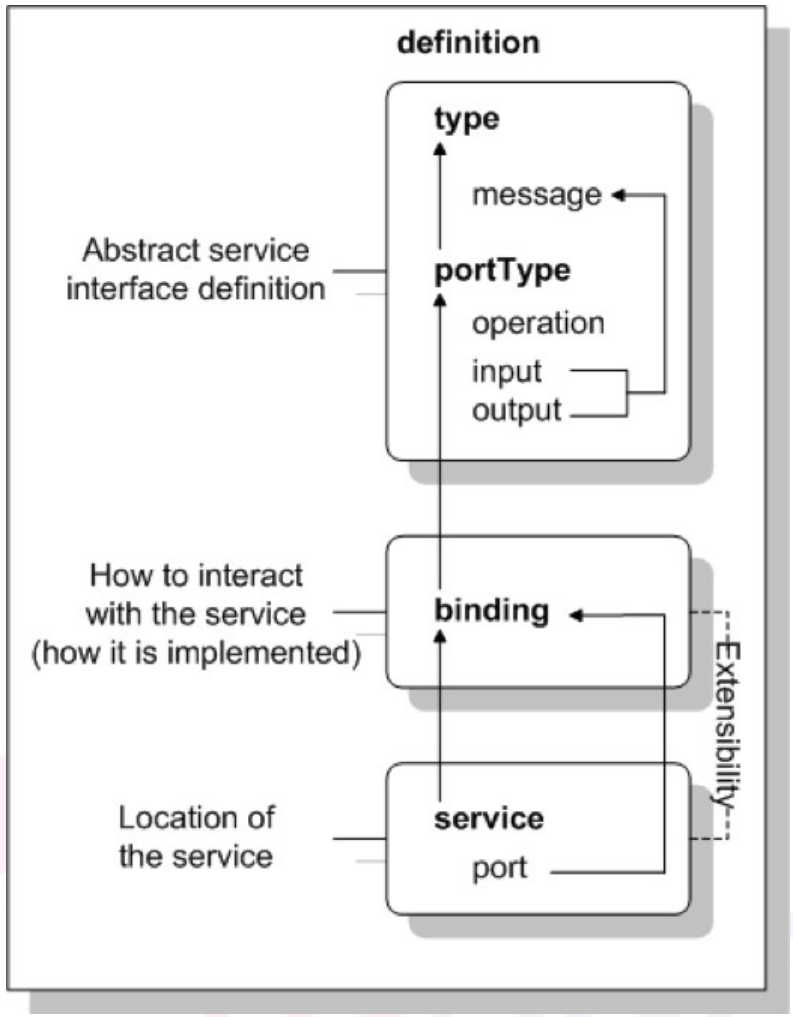
Client side


wsimport:
WSDL to Java
XService
extends
Javax.xml.ws.Service
parameters
JAXB
return
value
JAXB
Dynamic proxy
Service
Endpoint
Interface
(SEI)
Invocation
Handler
Web Service
WSDL
Endpoint
URL
SOAP
request
SOAP
response

WSDL

- A WSDL describes the point of contact for a service provider, also known as the service endpoint or just endpoint.
- Provides a formal definition of the endpoint interface
- requestors wishing to communicate with the service provider know exactly how to structure request messages
- Establishes the physical location (address) of the service.

WSDL elements



- **<types>**, the data types of input and output data, used by the web service
- **<message>**, messages to be exchanged, used by the web service
- **<portType>**, the operations input and output exposed by the web service. Note: parameters are represented as messages
- **<binding>**, the coupling and protocols used by the web service. This is were for example SOAP can be used as protocol
- **<port>** service location and binding

Web Service Example

A Web service WSDL example:

```
−<wsdl:definitions targetNamespace="http://muni.fi.cz/pa165/ws/entities/products">
  −<wsdl:types>
    −<xs:schema elementFormDefault="qualified" targetNamespace="http://muni.fi.cz/pa165/ws/entities/products">
       <xs:import namespace="http://muni.fi.cz/pa165/ws/entities/prices" schemaLocation="prices.xsd"/>
      −<!--
          Various requests and responses needed for operations on products
       -->
      −<xs:element name="getProductRequestByName">
        −<xs:complexType>
          −<xs:sequence>
             <xs:element name="name" type="xs:string"/>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
      −<xs:element name="getProductsRequest">
         <xs:complexType> </xs:complexType>
       </xs:element>
      −<xs:element name="getProductResponse">
        −<xs:complexType>
          −<xs:sequence>
             <xs:element maxOccurs="unbounded" minOccurs="0" name="product" type="tns:product"/>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
       <!-- Definition of type product -->
      −<xs:complexType name="product">
        −<xs:sequence>
           <xs:element name="id" type="xs:long"/>
           <xs:element name="name" type="xs:string"/>
           <xs:element name="description" type="xs:string"/>
           <xs:element name="addedDate" type="xs:date"/>
           <xs:element maxOccurs="unbounded" minOccurs="0" name="category" type="tns:category"/>
```

Generating the service code skeleton from the WSDL file

```
java2wsdl -cp . -tn . -stn calculator -cn Webservice
```

-cp = classpath; -tn target namespace; -stn schema target namespace; -cn class name

← Generate WSDL from Java

```
wsdl2java -ss -sd -uri Products.wsdl
```

-ss = server side; -sd = service descriptor

← Generate scheleton java webservice from WSDL

- A src directory is created with the source code for our server side files
- A resources directory is created with the WSDL file for the service and a service descriptor (services.xml) file
- A build.xml file is created in the current directory, which will be used to create the ws deployment file

Summary

- WS* standards and REST usually complement each others
- Different ways to develop “Contract first” vs “Contract last”
- Need to use frameworks for support (we see Spring-WS next)

Spring Web Services (Spring-WS)

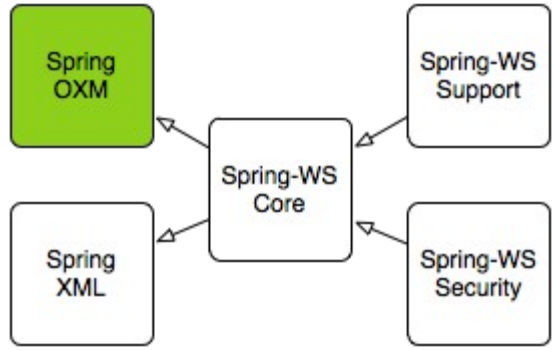
Spring-WS

- A Spring “sub-project” that allows to simplify WS-* development
- You can **reuse** as such your Spring application context and configuration in your application in your SOA application
- Plus, you get access to various **WS-* standards**
- Note that Spring-WS only supports **“contract first”** development

Spring-WS - Configuration

```
<dependencies>
    <dependency>
  <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-core</artifactId>
        <version>2.2.0.RELEASE</version>
    </dependency>
</dependencies>
```

Maven dependency




```
<beans xmlns="http://www.springframework.org/schema/beans">
    <bean id="webServiceClient" class="WebServiceClient">
        <property name="defaultUri"
value="http://localhost:8080/WebService"/>
    </bean>
</beans>
```

Webservice client bean

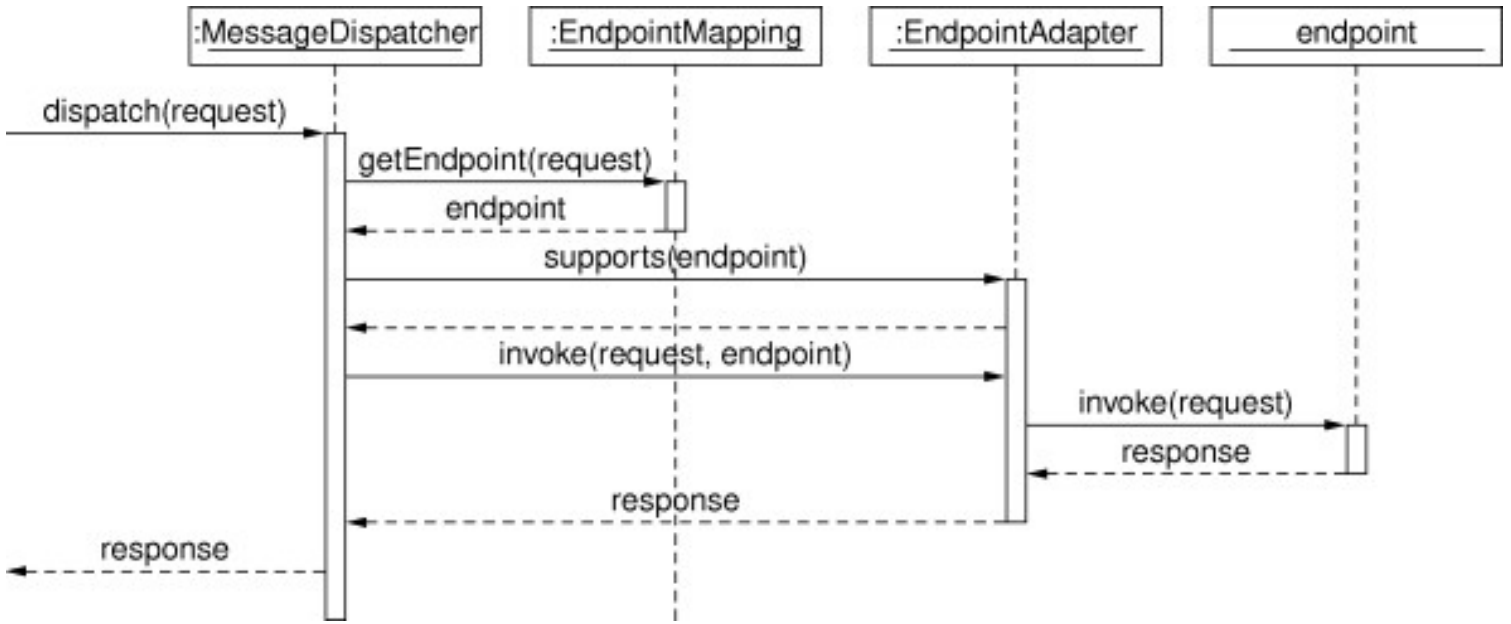
Spring-WS-Core depends
On Spring's Object/XML Mapping support (OXM) module and on Spring XML module

- See http://projects.spring.io/spring-ws/

Spring-WS

- Let's look at some of Spring-WS characteristics:
- MessageDispatcher & MessageDispatcherServlet
- Automatic WSDL exposure
- Endpoints & Endpoint Mapping
- Interceptors
- Exceptions
- Testing in Spring-WS

Spring-WS – Flow of invocations



① MessageDispatcher ≠ DispatcherServlet from SpringMVC
② MessageDispatcherServlet: a servlet that wraps MessageDispatcher
③ Exceptions thrown are taken care by any exception resolvers defined
④ Invocation chain for an endpoint: includes pre- and post-processors

Can you use a MessageDispatcher in Spring MVC DispatcherServlet?

Spring-WS – Automatic WSDL exposure

- By defining beans using **DefaultWsdl11Definition**, you can expose WSDL files to clients

```
@Bean(name = "products")
public DefaultWsdl11Definition productsWsdl11Definition(XsdSchema productsSchema) {
    DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
    wsdl11Definition.setPortTypeName("productsPort");
    wsdl11Definition.setLocationUri("/");
    wsdl11Definition.setTargetNamespace("http://muni.fi.cz/pa165/ws/entities/products");
    wsdl11Definition.setSchema(productsSchema);
    return wsdl11Definition;
}
```

Is it good idea to expose dynamically generated WSDL resources? What are the pros and cons?

Spring-WS – Automatic WSDL exposure

- By setting isTransformWsdlLocations() you can get automatic translation of the WSDL location based on requests

```
public class ServletInitializer extends AbstractAnnotationConfigMessageDispatcherServletInitializer
{
    @Override
    public boolean isTransformWsdlLocations() {
        return true;
    }
    //...
}
```

Spring-WS – Endpoints

```
@Endpoint
public class BookEndpoint {
     private static final String NAMESPACE_URI = "http://muni.cz/pa165/soa";
     private final BookRepository bookRepository;

     @Autowired
     public BookEndpoint(BookRepository bookRepository) {
          this.bookRepository = bookRepository;
     }

     @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getBookRequest")
     @ResponsePayload
     public GetBookResponse getBook(@RequestPayload GetBookRequest request) {

          GetBookResponse response = new GetBookResponse();

          response.setBook(bookRepository.getBookByTitle(request.getTitle()));
          return response;
     }

}
```

Spring-WS – Endpoint Mapping

- Maps the incoming messages to the correct endpoints
- **EndpointMapping** returns a **EndpointInvocationChain**, → **endpoint** that matches the incoming request and list of endpoint **interceptors** for request and response
- By default, **PayloadRootAnnotationMethodEndpointMapping** (using @PayloadRoot) and **SoapActionAnnotationMethodEndpointMapping** (using @SoapAction) are enabled by default
- If you want to use **WS-Addressing** as discussed before in the slides, you need to use AnnotationActionEndpointMapping and @Action and @Address in the endpoint

Spring-WS – SoapMessage (1/3)

- If you remember our SOAP with Attachments API for Java (SAAJ) example, it required quite some code
- In Spring you usually mostly care about the SOAP body that you can get in an endpoint by using @RequestPayload annotation that gives you access to the request
- Only in cases in which you want to modify/get header information or deal with attachments, you need to care about headers

Spring-WS – SoapMessage (2/3)

Name	Supported parameter types
TrAX	`javax.xml.transform.Source` and sub-interfaces (`DOMSource`, `SAXSource`, `StreamSource`, and `StAXSource`)
W3C DOM	`org.w3c.dom.Element`
dom4j	`org.dom4j.Element`
JDOM	`org.jdom.Element`
XOM	`nu.xom.Element`
StAX	`javax.xml.stream.XMLStreamReader` and `javax.xml.stream.XMLEventReader`
XPath	Any boolean, double, `String`, `org.w3c.Node`, `org.w3c.dom.NodeList`, or type that can be converted from a `String` by a Spring 3 conversion service, and that is annotated with `@XPathParam`.
Message context	`org.springframework.ws.context.MessageContext`
SOAP	`org.springframework.ws.soap.SoapMessage`, `org.springframework.ws.soap.SoapBody`, `org.springframework.ws.soap.SoapEnvelope`, `org.springframework.ws.soap.SoapHeader`, and `org.springframework.ws.soap.SoapHeaderElement`s when used in combination with the `@SoapHeader` annotation.
JAXB2	Any type that is annotated with `javax.xml.bind.annotation.XmlRootElement`, and `javax.xml.bind.JAXBElement`.
OXM	Any type supported by a Spring OXM Unmarshaller.

Spring-WS – SoapMessage (3/3)

- Example, we can get the list of all *"mustUnderstand"* elements from the header (see **mustUnderstand**)

```
@PayloadRoot(namespace = NAMESPACE_URI, localPart = "getProductRequestByName")
@ResponsePayload
public GetProductResponse getProduct(@RequestPayload GetProductRequestByName request,
                                                    SoapHeader header) {

        // …….

        Iterator<SoapHeaderElement> itMustUnderstand =
                            header.examineMustUnderstandHeaderElements(URI);

            while (itMustUnderstand.hasNext() ){
                SoapHeaderElement element = itMustUnderstand.next();
                // do something with the element in case it is not understood, return a
                   SoapFault
            }

        //.......
}
```

Spring-WS – Interceptors (1/4)

- Although you can process SOAP message headers in Endpoints, better is to use interceptors that will be applied to all requests/responses or to a filtered set
- HandleRequest(..) provides the possibility to handle the request **before** an endpoint is invoked. If **false** is returned, the execution chain is interrupted
- HandleResponse(..) and HandleFault(..) deal with the response **after** the endpoint is invoked for both the **normal** and **faulty case** – if returning **false**, the response will not be returned back

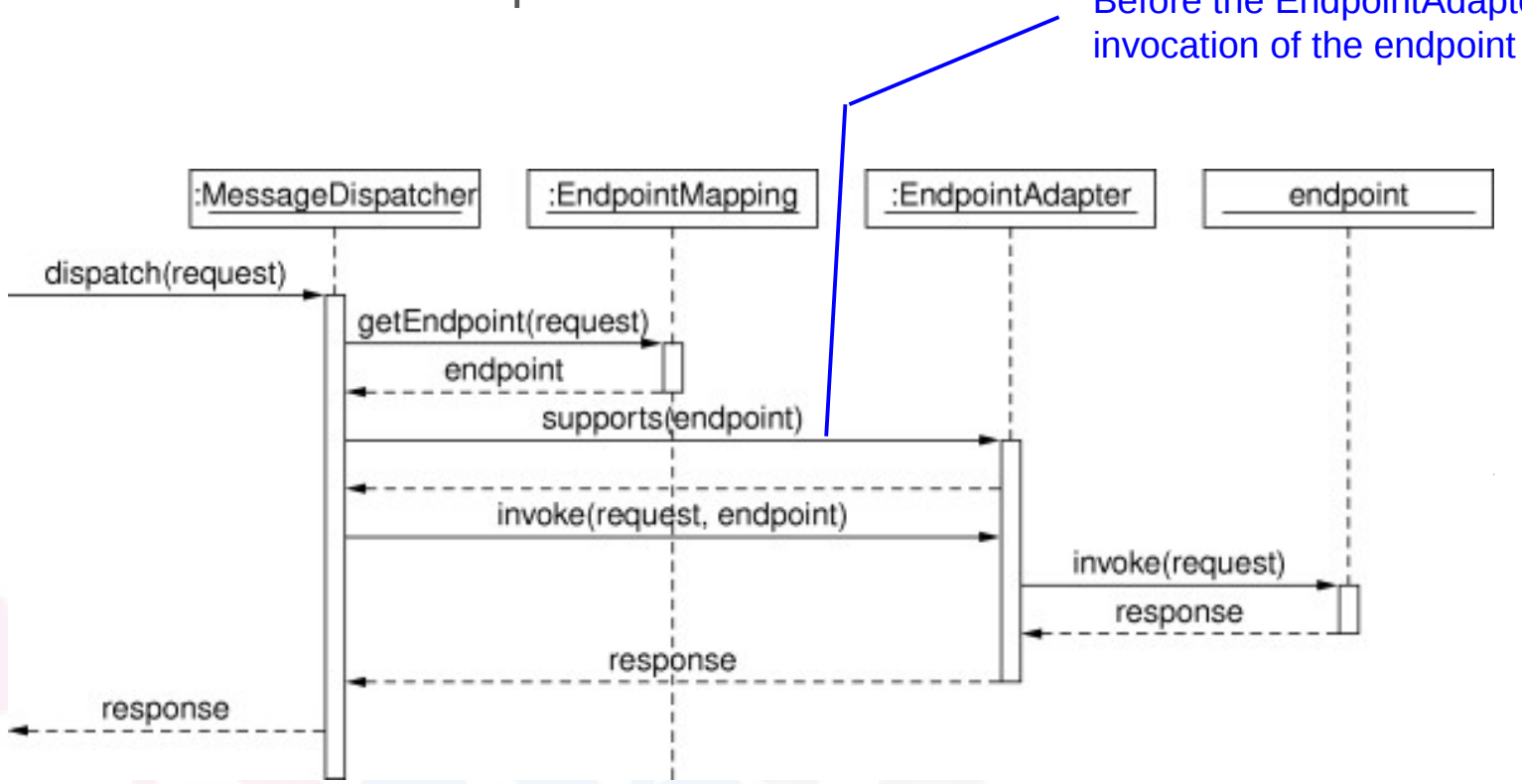
Spring-WS – Interceptors (2/4)

- Example

```
public class AnInterceptor implements EndpointInterceptor{

    @Override
    public boolean handleRequest(MessageContext mc, Object o) throws Exception {
        WebServiceMessage  wsm = mc.getRequest();     // from here you can access the
                                                         payload as in an enpoint
        SoapMessage sm         = (SoapMessage) wsm;   // cast the webservicemessage to
                                                         soapmessage
        SoapHeader  sh         = sm.getSoapHeader();  // you can now use the soapheader as
                                                         before

        // ....

        return true;
    }

     // other overridden necessary methods
}
```

Spring-WS – Interceptors (3/4)

- Where the interceptor is invoked



Spring-WS – Interceptors (4/4)

- There are some predefined interceptors that can be useful:
- **PayloadLoggingInterceptor** and **SoapEnvelopeLoggingInterceptor** allow to log payload or whole soap envelopes for responses and/or requests
- **PayloadValidatingInterceptor** to validate requests/response

```
@Bean
public PayloadValidatingInterceptor myPayLoadInterceptor() {
    final PayloadValidatingInterceptor interceptor = new
                                            PayloadValidatingInterceptor();
    interceptor.setXsdSchema(this.productsSchema());
    interceptor.setValidateRequest(true);
    interceptor.setValidateResponse(true);
    return interceptor;
}
```

Which one would be more meaningful to validate? Spring docs refer to Postel's law or robustness principle: *"Be conservative in what you send, be liberal in what you accept"*

Spring-WS – Exceptions (1/2)

- Easiest way to deal with exceptions is to annotate custom exceptions with **@SoapFault** that will be dealt with a pre-configured **SoapFaultAnnotationExceptionResolver**

```
@SoapFault(faultCode = FaultCode.SERVER, faultStringOrReason = "Product not found." )
public class ProductNotFoundException extends RuntimeException {
        public ProductNotFoundException(String productName) {
          super("could not find product " + productName );
     }
}
```

Spring-WS – Exceptions (2/2)

- If you need a more programmatic way, you can implement an **EndpointExceptionResolver** overriding method *resolveException(MessageContext, Endpoint, Exception)*
- Or SimpleSoapExceptionResolver to have access at the SOAP Fault

```
public class EndpointExceptionResolver extends SoapFaultMappingExceptionResolver {

   //...

   @Override
   protected void customizeFault(Object endpoint, Exception ex, SoapFault fault) {
      // ...
   }
}
```

You can filter based on the endpoint from which the exception comes from

You can filter based on the exception type

Add more information to the SOAP fault

Spring-WS – Testing (1/3)

- We use **MockWebServiceClient** to mock a webservice client with some request messages for the enpoints under test that are configured in the **ApplicationContext**
- The endpoints will handle the messages and return a response

```
@ContextConfiguration(classes = {WebServiceConfig.class})
public class ProductEndpointTest extends AbstractTestNGSpringContextTests {

    @Autowired
    private ApplicationContext applicationContext;

    @BeforeClass
    public void createClient() {
        mockClient = MockWebServiceClient.createClient(applicationContext);
    }
    //...
}
```

Spring-WS – Testing (2/3)

- We use then **MockWebServiceClient** to test against expected behaviour

```
Source requestPayload = new StringSource( "<getProductRequestByName
                        xmlns='http://muni.fi.cz/pa165/ws/entities/products'>"
              + "<name>No product</name>"
              + "</getProductRequestByName>");

mockClient.sendRequest(withPayload(requestPayload)).
                                    andExpect(serverOrReceiverFault("Product not
                                                                    found."));
```

Would you consider a test written using MockWebServiceClient a unit or an integration test?
Would you mock service endpoints?

Spring-WS – Testing (3/3)

- You might also implement your own Matcher by implementing ResponseMatcher interface
- There are however many that you can use:

`ResponseMatchers` method	Description
`payload()`	Expects a given response payload.
`validPayload()`	Expects the response payload to validate against given XSD schema(s).
`xpath()`	Expects a given XPath expression to exist, not exist, or evaluate to a given value.
`soapHeader()`	Expects a given SOAP header to exist in the response message.
`noFault()`	Expects that the response message does not contain a SOAP Fault.
`mustUnderstandFault()`, `clientOrSenderFault()`, `serverOrReceiverFault()`, and `versionMismatchFault()`	Expects the response message to contain a specific SOAP Fault.

References

- SOAP 1.2 Specifications
  http://www.w3.org/TR/2007/REC-soap12-part0-20070427/
  http://www.w3.org/TR/2007/REC-soap12-part1-20070427/
  http://www.w3.org/TR/2007/REC-soap12-part2-20070427/
- Spring-WS Reference
  http://docs.spring.io/spring-ws/docs/2.2.3.BUILD-SNAPSHOT/reference/htmlsingle/
- Webservices Standards Overview
  https://www.innoq.com/soa/ws-standards/poster/innoQ%20WS-Standards%20Poster%202007-02.pdf