Berner
Fachhochschule

Java Web Services

Stephan Fischli

Winter 2015/16

Objectives

- Understand the concept of web services
- Know the relevant standards of web services
- Know the different Java APIs used to realize web services

Contents

1 Introduction

Definition

Web Services are self-contained, modular applications that can be described, published, located and invoked over a network, generally the World Wide Web.

(IBM, September 2000)

Motivation

Bottom-up view:

- Web services allow remote procedure calls through firewalls

Top-down view

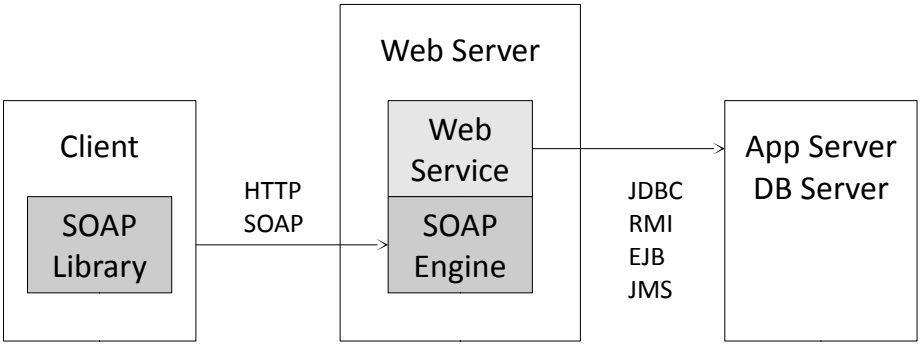
- Web services provide the means for application integration, especially between business partners

Historical view:

- Internet: network of computers
- World Wide Web: network of documents
- Web services: network of applications

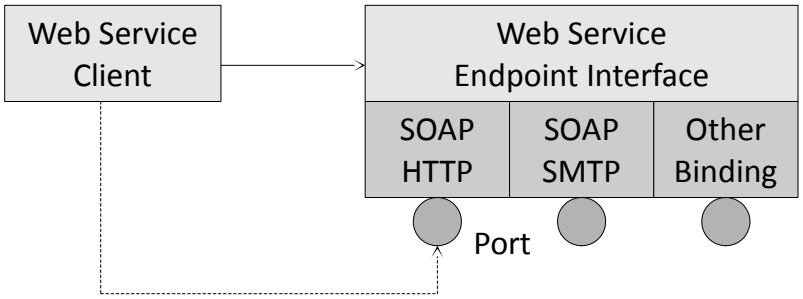
Architecture

- Web services use standardized protocols and data formats (e.g. HTTP, XML, SOAP)
- The access of a web service is independent of its implementation and deployment platform
- The service description provides all the details necessary to invoke a web service



Web Service Model

- The endpoint interface defines the operations of a web service
- A binding maps the endpoint interface to a protocol stack
- A port defines the endpoint address that a client can use to access the web service



Standards

W3C (www.w3.org):

- XML and XML Schema
- Simple Object Access Protocol (SOAP)
- Web Services Definition Language (WSDL)

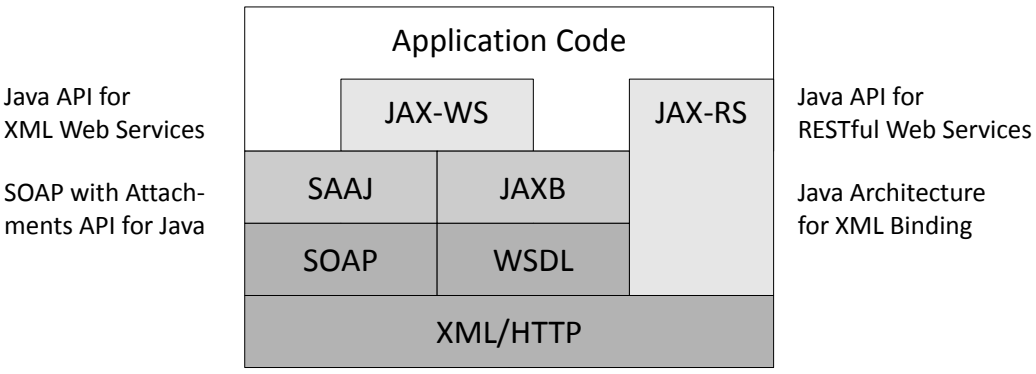
OASIS (www.oasis-open.org)

- Universal Description, Discovery and Integration (UDDI)
- Electronic Business using XML (ebXML)
- Web Services Security (WSS)
- Business Process Execution Language (BPEL)

WS-I (www.ws-i.org)

- Web Services Interoperability Profile

Java APIs



References

- Martin Kalin
  Java Web Services: Up and Running, O'Reilly, 2013
- Bill Burke
  RESTful Java with JAX-RS, O'Reilly, 2013

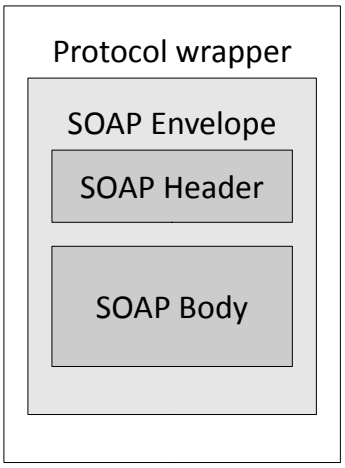
2 SOAP Protocol

Definition

SOAP is an XML-based communication protocol that defines

- the structure of messages (envelope)
- bindings to transport protocols (e.g. HTTP)
- encoding rules for data types
- conventions for remote procedure calls
- a mechanism for error handling

Structure of a SOAP Message



The envelope is the message container

The optional header may contain processing or routing information

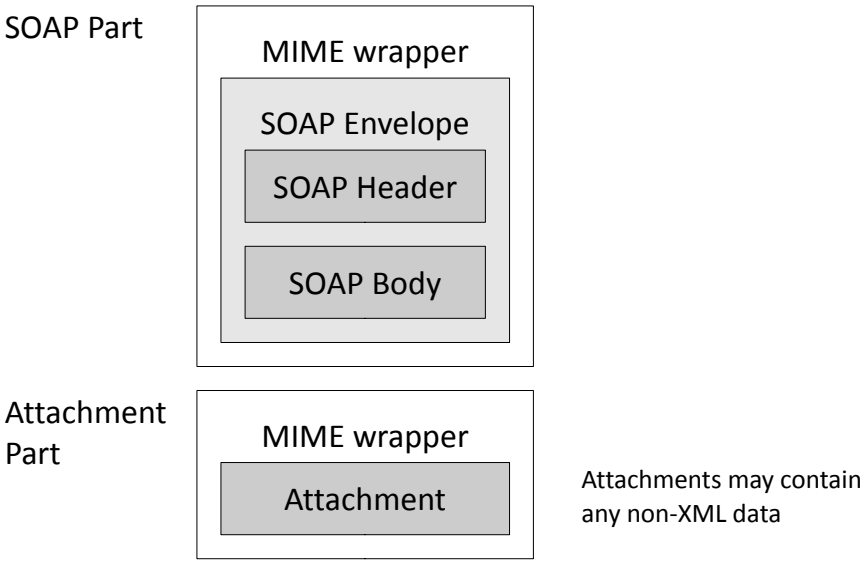
The body contains arbitrary XML data or a SOAP fault

Example of a SOAP Message

```
POST /contactbook/soap HTTP/1.1
...
Content-Type: text/xml; charset=utf-8
Content-Length: 210
Host: distsys.ch:8080
SOAPAction: ""

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header/>
   <soap:Body>
      <tns:findContact xmlns:tns="http://example.org/contactbook">
         <id>5</id>
      </tns:findContact>
   </soap:Body>
</soap:Envelope>
```

SOAP Messages with Attachments



SOAP Headers

SOAP Headers are not standardized and may contain additional information such as

- authentication and authorization
- transaction management
- tracing and auditing
- payment information

SOAP Intermediaries

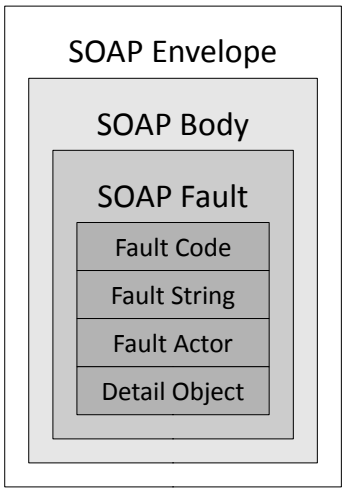
- SOAP intermediaries allow to route a message to multiple recipients
- Header elements may be associated to a specific recipient by the *actor* attribute
- The *mustUnderstand* attribute says whether the recipient is required to process the corresponding header element

Example of a SOAP Message with Header

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <tns:basicAuthentication xmlns:tns="http://example.org/contactbook"
          soap:actor="urn:security" soap:mustUnderstand="1">
      <username>admin</username>
      <password>*****</password>
    </tns:basicAuthentication>
  </soap:Header>
  <soap:Body>
    <tns:createContact xmlns:tns="http://example.org/contactbook">
      <name>John Doe</name>
      <phone>+1 123-456-7890</phone>
      <email>jd@example.org</email>
    </tns:createContact>
  </soap:Body>
</soap:Envelope>
```

SOAP Faults

Processing errors of a request are reported through SOAP faults in the body of the response message



Default codes are VersionMismatch, MustUnderstand, Client and Server

Description of the error

Causer of the error (optional)

Error details (optional)

Example of a SOAP Message with Fault

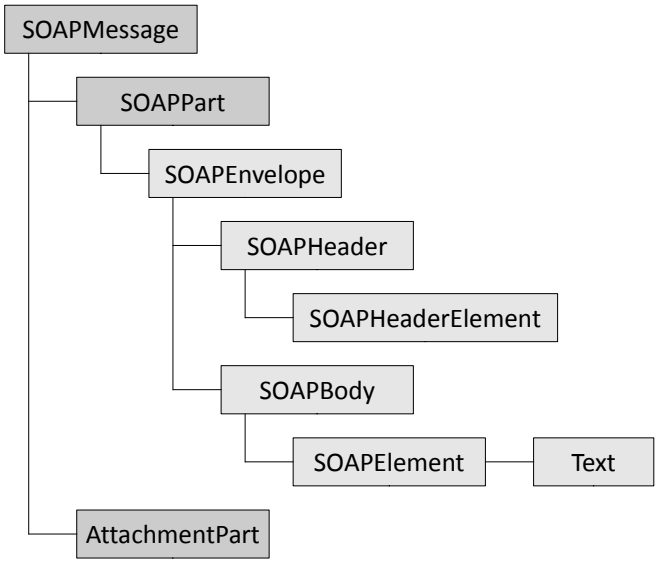
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Contact not found</faultstring>
      <detail>
        <tns:NotFoundFault xmlns:tns="http://example.org/contactbook">
          Contact not found
        </tns:NotFoundFault>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

SOAP with Attachment API for Java (SAAJ)

- SAAJ is a simple API for manipulating and sending SOAP messages
- SAAJ is especially suitable for accessing document oriented web services
- SAAJ can be used to access a web service even if its operations are unknown at compile time

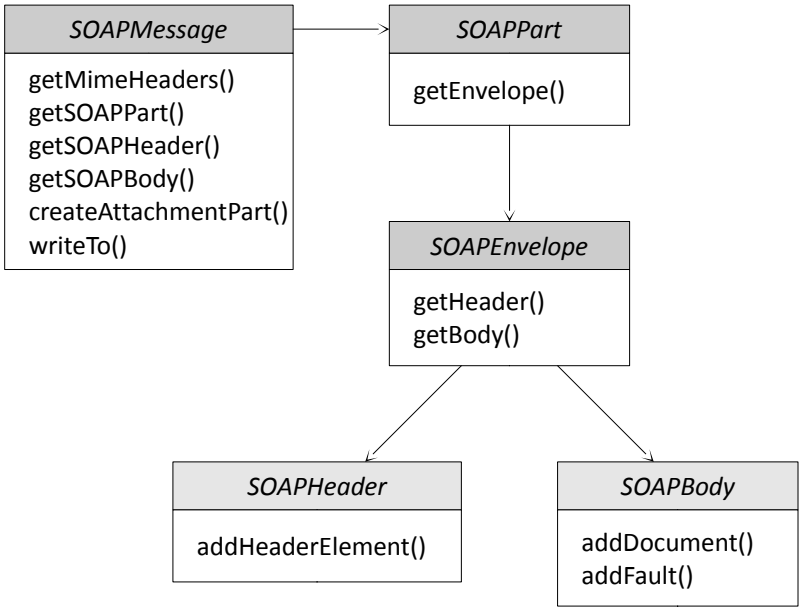
Anatomy of a SAAJ Message

A SAAJ message consists of a hierarchy of objects representing the different parts of the SOAP message and the XML elements



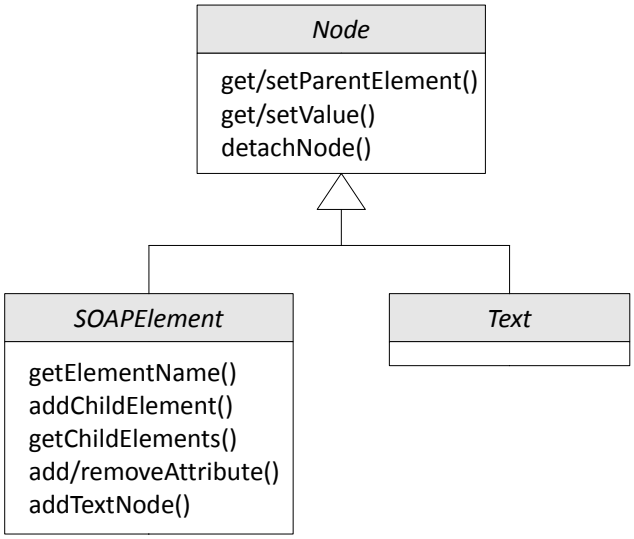
Interfaces and Classes

The following interfaces and classes are used to built the structure of a SAAJ message



Interfaces and Classes (cont.)

The following interfaces are used to construct the XML content of a SAAJ message



Creating a Message

A newly created message contains a SOAPEnvelope with an empty SOAPHeader and an empty SOAPBody element

```
import javax.xml.soap.*;
...
MessageFactory messageFactory = MessageFactory.newInstance();
SOAPMessage request = messageFactory.createMessage();
request.getSOAPHeader().detachNode();
SOAPBody body = request.getSOAPBody();
```


Adding Content to a Message

Content can be added to a message by creating a SOAPBodyElement and adding child elements to it

```
String namespace = "http://example.org/contactbook";
SOAPElement bodyElement =
   body.addChildElement("findContact", "tns", namespace);
SOAPElement idElement = bodyElement.addChildElement("id");
idElement.addTextNode("5");
```

Adding a Document to a Message

Instead of setting the content of a message, it is possible to add an entire DOM document to the body of a message

```
import javax.xml.parsers.*;
import org.w3c.dom.Document;
...
DocumentBuilderFactory builderFactory =
   DocumentBuilderFactory.newInstance();
builderFactory.setNamespaceAware(true);
DocumentBuilder builder = builderFactory.newDocumentBuilder();
Document document = builder.parse("findRequest.xml");
body.addDocument(document);
```


Sending a Message

A message can be sent by creating a SOAPConnection and providing the endpoint address of the web service

```
SOAPConnectionFactory connectionFactory =
   SOAPConnectionFactory.newInstance();
SOAPConnection connection = connectionFactory.createConnection();
String endpoint = "http://distsys.ch:8080/contactbook/soap";
SOAPMessage response = connection.call(request, endpoint);
connection.close();
```

Reading the Content of a Message

The content of a message can be read by retrieving the SOAPBody-Element and its child elements

```
SOAPBody body = response.getSOAPBody();
SOAPElement bodyElement = (SOAPElement)body.getFirstChild();
SOAPElement contactElement = (SOAPElement)bodyElement.getFirstChild();
Iterator<SOAPElement> iter = contactElement.getChildElements();
System.out.println("Name: " + iter.next().getValue());
System.out.println("Phone: " + iter.next().getValue());
System.out.println("Email: " + iter.next().getValue());
```


Reading Fault Information

If a message contains a fault, its information can be read by retrieving the SOAPFault element from the body

```
if (body.hasFault()) {
    SOAPFault fault = body.getFault();
    String faultCode = fault.getFaultCode();
    String faultString = fault.getFaultString();
    String faultActor = fault.getFaultActor();
    Detail detail = fault.getDetail();
    ...
}
```

3 Web Services Description Language (WSDL)

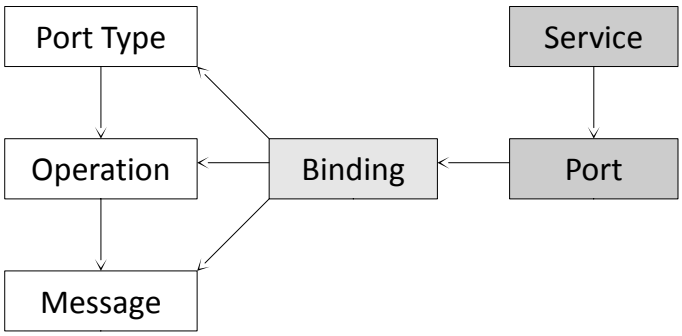
Introduction

- WSDL is an XML language that is used to describe the interface of a web service
- WSDL is platform, protocol and programming language independent
- The WSDL document of a web service is published at a well-known URL or in a registry
- Provider tools can parse WSDL and generate the code necessary to implement or to access a web service

Overview of WSDL Document

A WSDL document describes a web service in terms of

- the operations that the web service provides
- the data types that each operation requires by its input and output messages
- the binding that maps the input and output messages onto a protocol
- the address at which the service can be accessed



Structure of a WSDL Document

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" ...>
   <import ...>
   <types>
      <xsd:schema>...</xsd:schema>
   </types>
   <message name="findContact">...</message>
   <message name="findContactResponse">...</message>
   ...
   <portType name="ContactBook">
      <operation name="findContact">...</operation>
      ...
   </portType>
   <binding>
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" ... />
      ...
   </binding>
   <service>
      <port><soap:address location="..."/></port>
   </service>
</definitions>
```

Definitions

- The definitions element is the root element of a WSDL document
- The name given to the web service is for documentation purpose only
- The target namespace determines to which namespace the elements belong used to describe the web service

```
<definitions name="ContactBookService"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      targetNamespace="http://example.org/contactbook"
      xmlns:tns="http://example.org/contactbook"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
   ...
</definitions>
```


Imports

- An import element allows to import data type definitions from a separate XML schema document (not recommended)
- An import element allows to include another WSDL document and thus to separate the generic definition of a web service from its bindings and its location

```
<import namespace="http://example.org/contactbook"
   location="ContactBookTypes.xsd"/>
<import namespace="http://example.org/contactbook"
   location="ContactBookGeneric.wsdl"/>
```

Data Types

- The types element defines the data types that are used in the messages of the web service
- The data types should be defined using the XML schema language (either inline or by importing an external schema)

```
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://example.org/contactbook">
    <element name="findContact" type="tns:findContact"/>
    <complexType name="findContact">
      <sequence><element name="id" type="long"/></sequence>
    </complexType>
    ...
  </schema>
</types>
```


Messages

- The message elements define the messages used by the operations of the web service
- Each message may contain any number of part elements that represent an item of data
- The data type of a message part is declared by an element or a type attribute referencing a standard type or a user-defined type

```
<message name="findContact">
  <part name="parameters" element="tns:findContact"/>
</message>
<message name="findContactResponse">
  <part name="parameters" element="tns:findContactResponse"/>
</message>
<message name="NotFoundFault">
  <part name="fault" element="tns:NotFoundFault"/>
</message>
...
```

The Port Type

- The portType element defines the operations of a web service
- Each operation can have an input, an output and any number of fault messages

```
<portType name="ContactBook">
   <operation name="findContact">
      <input message="tns:findContact"/>
      <output message="tns:findContactResponse"/>
      <fault message="tns:NotFoundFault" name="NotFoundFault"/>
   </operation>
   ...
</portType>
```


Operation Types

The appearance and order of the input and output messages determine the type of an operation

Type	Messages
Request-Response	Input, Output, Fault
One-Way	Input
Solicit-Response*	Output, Input, Fault
Notification*	Output

*Not supported by WS-I and JAX-WS

Protocol Bindings

- The binding elements define for each port type how the input and output messages of its operations are mapped to concrete protocol messages
- The additional elements necessary for a concrete binding are defined by individual protocol specifications

```
<binding name="ContactBookBinding" type="tns:ContactBook">
   ...
   <operation name="findContact">
      ...
      <input>...</input>
      <output>...</output>
      <fault>...</fault>
   </operation>
   ...
</binding>
```


The SOAP Binding

The SOAP binding defines

- the transport protocol used to carry the SOAP messages
- the style of the operations (document or RPC)
- the value of the SOAPAction header used to access the service
- the appearance (body or header) and encoding (literal or encoded) of each message

```
<binding ...>
   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="findContact">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
      <fault><soap:fault use="literal" name="NotFoundFault"/></fault>
   </operation>
</binding>
```

Operation Styles

The style of an operation determines how the SOAP messages are constructed from the WSDL description

Document:

- the body contains one or more child elements that correspond to the parts of the corresponding message
- the part elements in the WSDL document point to schema elements, therefore the message body can easily be validated

RPC:

- the body consists of a single (wrapper) element named for the operation being invoked
- the wrapper element contains a child element for each parameter defined by the parts of the corresponding message

The recommended style is document wrapped

Message Encodings

The encoding rules of a message determines the serialization format of the message parts

Literal:

- data is serialized according to a schema definition usually expressed in the XML schema language
- the types of the elements in the SOAP message implicitly rely on the schema

Encoded:

- data is serialized according to some encoding rules usually the SOAP section 5 rules
- the elements in the SOAP message carry explicit type qualification

The Endpoint Address

- A port element maps a binding of a port type to a URI which can be used to access it
- A service element groups together a set of related ports

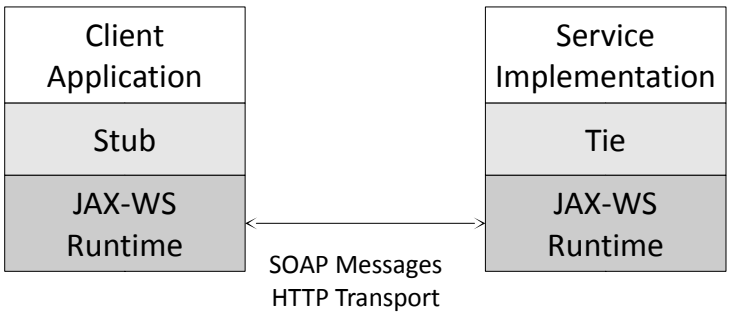
```
<service name="ContactBookService">
   <port name="ContactBookPort" binding="tns:ContactBookBinding">
      <soap:address location="http://distsys.ch:8080/contactbook/soap"/>
   </port>
</service>
```

4 Java API for XML Web Services (JAX-WS)

Introduction

- JAX-WS provides a high-level API for the implementation of web services and web service clients
- JAX-WS maps a web service endpoint interface given in WSDL to a Java interface and vice versa
- JAX-WS delegates the mapping of Java data types to and from XML definitions to JAXB
- JAX-WS is based on standard technologies and supports the WS-I Basic Profile

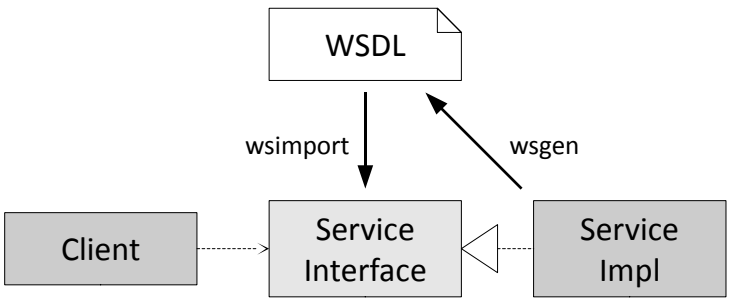
JAX-WS Architecture



- The client invokes a stub method which is delegated to the JAX-WS runtime in order to send an appropriate SOAP message to the server
- On the server side, the tie converts the received message back into a method call on the actual service implementation
- The stub and tie classes are dynamically generated by the provider platform

Using WSDL with JAX-WS

- JAX-WS can generate a WSDL document from the Java implementation of a web service (code-first)
- JAX-WS can import a WSDL document to generate the Java code required to implement a web service or a web service client (contract-first)



Java to WSDL Mapping

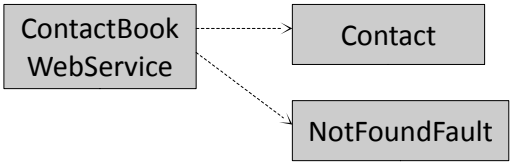
- Java packages are mapped to namespaces
- Java classes are mapped to portType elements
- Java methods are mapped to operation elements with corresponding input and output messages
- Java exceptions are mapped to fault elements

```
public class ContactBook {
   public Contact findContact(long id) throws NotFoundFault { ... }
}

<wsdl:portType name="ContactBook">
   <wsdl:operation name="findContact">
      <wsdl:input message="tns:findContact"/>
      <wsdl:output message="tns:findContactResponse"/>
      <wsdl:fault message="tns:NotFoundFault"/>
   </wsdl:operation>
</wsdl:portType>
```


Developing a Code-first Web Service

1. Implement and compile the web service class
2. Optionally generate the WSDL document using the wsgen utility
3. Package the service code into a web archive
4. Deploy the web archive into a running web container



The Web Service Class

- The web service class must be annotated with the @WebService annotation which defines the names and the target namespace of the web service
- The @SOAPBinding annotation can be used to specify the operation style and message encoding

```
@WebService(name="ContactBook",
   portName="ContactBookPort", serviceName="ContactBookService",
   targetNamespace="http://example.org/contactbook")
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL,
   parameterStyle=ParameterStyle.WRAPPED)
public class ContactBookWebService {
   ...
}
```


The Web Service Lifecycle

- The web service class must have a public default constructor
- The @PostConstruct and @PreDestroy annotations can be used for lifecycle callback events
- Resources can be obtained by JNDI lookup or by injection

```
@WebService(...)
public class ContactBookWebService {

   public ContactBookWebService() { ... }
   @PostConstruct private void init() { ... }
   @PreDestroy private void destroy() { ... }
   ...
}
```

The Web Service Methods

- The methods of a web service must have JAXB compatible parameters and return types
- The @WebMethod, @WebParam and @WebResult annotations can be used to customize the mapping of the method names, parameters and return values

```
@WebService(...)
public class ContactBookWebService {
   @WebMethod(operationName="findContact")
   @WebResult(name="contact")
   public Contact findContact(@WebParam(name="id") long id)
      throws NotFoundFault {
      ...
   }
}
```


Exceptions

- JAX-WS exceptions always contain a fault information in addition to the error message
- The @WebFault annotation can be used to define the name of the fault information

```
@WebFault(name="NotFoundFault")
public class NotFoundFault extends Exception {
   public NotFoundFault(String message) {
      super(message);
   }
   public String getFaultInfo() { return getMessage(); }
}
```

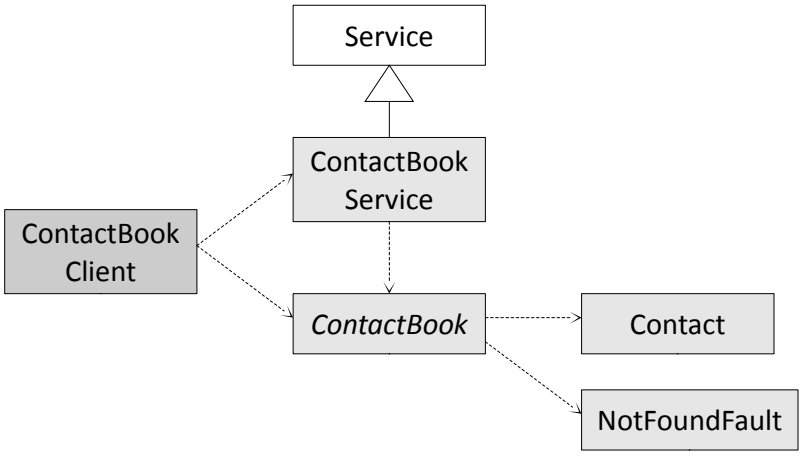
Oneway Operations

- If an operation has no return value and throws no exceptions, it can be annotated with the @OneWay annotation
- When a client invokes a one-way operation, it immediately returns and does not have to wait for the processing of the operation

```
@WebService(...)
public class ContactBookWebService {
    ...
    @Oneway @WebMethod(...)
    public void deleteContact(@WebParam(name="id") long id) { ... }
}
```


Developing a Web Service Client

1. Generate the artifacts needed to connect to the web service using the wsimport utility
2. Implement and compile the client application
3. Run the client



Generation of Client Artifacts

- The wsimport tool reads the WSDL document of a web service and generates the portable artifacts used to implement a client
- The generated files include a service factory, the endpoint interface and wrapper classes

```
wsimport -d build -s generated -p org.contacts.soap -b bindings.xml
   http://distsys.ch:8080/contactbook/soap?wsdl
```


Generation of Client Artifacts (cont.)

- Binding declarations can be used to customize the WSDL-to-Java mapping, e.g. to generate asynchronous methods or to disable the generation of wrapper style methods

```
<bindings xmlns="http://java.sun.com/xml/ns/jaxws">
   <enableAsyncMapping>true</enableAsyncMapping>
   <enableWrapperStyle>false</enableWrapperStyle>
   ...
</bindings>
```

- The generated service factory contains the URL of the WSDL document which is parsed again at runtime, but the wsdlLocation option can be used to configure a local path

The Client Implementation

- The client creates a service factory object, obtains a stub from it and invokes its methods
- Optionally, the default endpoint address of the stub can be overridden using the BindingProvider interface

```
public class ContactBookClient {
   public static void main(String[] args) throws Exception {
      ContactBookService service = new ContactBookService();
      ContactBook contactBook = service.getContactBookPort();
      ((BindingProvider)contactBook).getRequestContext().put(
         BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
         "http://distsys.ch:8080/contactbook/soap");
      Contact contact = contactBook.findContact(5);
      ...
   }
}
```


Developing a Contract-first Web Service

1. Provide the WSDL document of the web service
2. Generate the Java service interface and portable artifacts using the wsimport utility
3. Implement the generated web service interface and compile the web service class
4. Package the service code and optionally the WSDL document into a web archive
5. Deploy the web archive into a running web container

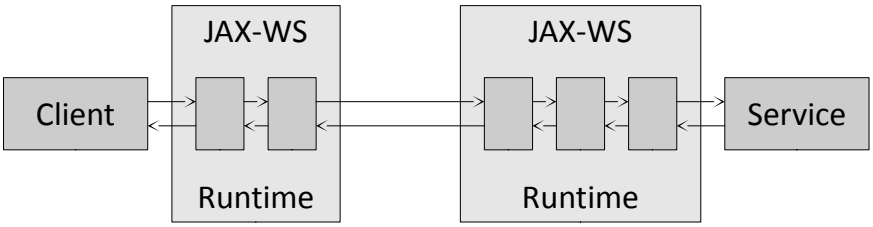
```
@WebService(endpointInterface="org.contacts.soap.ContactBook",
   wsdlLocation="ContactBook.wsdl", ...)
public class ContactBookWebService implements ContactBook {
   ...
}
```

Advanced Features

- Message Handlers are interceptors which can be used for additional processing of messages (e.g. logging, filtering, security)
- By default there is no schema validation of SOAP messages but the proprietary @SchemaValidation annotation of the JAX-WS reference implementation can be applied to web service classes
- Clients can invoke web service operations asynchronously, either by polling or by using a callback
- The Provider and Dispatch interfaces allow web service endpoints and web service clients to work at the message level
- The Message Transmission and Optimization Mechanism (MTOM) specifies how XML binary data can be send as attachment

Message Handlers

- Protocol handlers are specific to a protocol and can access or change any part of the messages in particular message headers
- Logical handlers are protocol-agnostic and only act on the payload of the messages
- Handlers are grouped into handler chains which are inserted in the processing path on the client, the server or both
- For an outbound message the logical handlers are executed before the protocol handlers, for an inbound message vice versa



Writing a Message Handler

- A message handler must implement the SOAPHandler or the LogicalHandler interface
- The message context passed to the handle methods contains a SOAPMessage or a LogicalMessage which can be manipulated using SAAJ or JAXB, respectively
- The return values indicate whether to continue processing of the handler chain or not

```
public class LoggingHandler
    implements SOAPHandler<SOAPMessageContext> {
    public boolean handleMessage(SOAPMessageContext context) {
        SOAPMessage message = context.getMessage();
        message.writeTo(System.out);
        return true;
    }
    ...
}
```


Configuring a Message Handler

- A message handler is part of a handler chain which is defined in an XML deployment descriptor

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-name>LoggingHandler</handler-name>
            <handler-class>org.contacts.soap.LoggingHandler</handler-class>
        </handler>
    </handler-chain>
</handler-chains>
```

- A handler chain can be associated with a web service through the @HandlerChain annotation which references the descriptor

```
@WebService(...)
@HandlerChain(file="handler-chain.xml")
public class ContactBookWebService { ... }
```

Asynchronous Clients

- When generating the client artifacts, a binding declaration can be used to generate asynchronous methods, one for polling and one using callbacks

```
@WebService(...)
public interface ContactBook {
    public Contact findContacts(long id);
    public Response<FindContactResponse> findContactAsync(long id);
    public Future<?> findContactAsync(
        String name, AsyncHandler<Contact> asyncHandler);
    ...
}
```


Asynchronous Polling

- The client invokes the polling method on the stub and repeatedly checks for a response on the returned Response object

```
public class ContactBookClient {
    public static void main(String[] args) {
        ContactBookService service = new ContactBookService();
        ContactBook contactBook = service.getContactBookPort();
        Response<FindContactResponse> response =
            contactBook.findContactAsync(long id);
        while (!response.isDone()) {
            ...
        }
        Contact contact = response.get().getContact();
        ...
    }
}
```

Asynchronous Callback

- The client provides an AsyncHandler object which will be called as soon as the response is available

```
public class ContactBookClient {
    public static void main(String[] args) {
        ContactBookService service = new ContactBookService();
        ContactBook contactBook = service.getContactBookPort();
        contactBook.findContactAsync(name, new ContactHandler());
        ...
    }
}
public class ContactHandler
        implements AsyncHandler<FindContactResponse> {
    public void handleResponse(
        Response<FindContactResponse> response) {
        Contact contact = response.get().getContact();
        ...
    }
}
```


Web Services and EJB

- Stateless session beans can be exposed as web services by annotating the bean with the @WebService annotation or by defining a separate web service interface which is implemented by the bean
- In any enterprise bean, the @WebServiceRef annotation can be used to inject the service factory of a web service or directly the web service interface type

```
@Stateless
public class AdminServiceBean implements AdminService {
    @WebServiceRef(ContactBookService.class)
    private ContactBook contactBook;
    ...
}
```

Web Service Endpoint Address

- For servlet-based web services the default endpoint address is

  http://<host>:<port>/<context-root>/<servlet-url-mapping>

- If there is no servlet to URL mapping in the web deployment descriptor, the serviceName attribute of the @WebService annotation is used instead
- For EJB-based web services the default endpoint address is

  http://<host>:<port>/<service-name>/<port-type-name>

  where the service and port type names are taken from the serviceName and name attributes of the @WebService annotation