Berner
Fachhochschule

Java Web Services

Stephan Fischli

Winter 2022/23

Contents

1 Introduction
2 SOAP Web Services
3 Java API for XML Web Services (JAX-WS)
4 REST Principles
5 Java API for RESTful Web Services (JAX-RS)

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

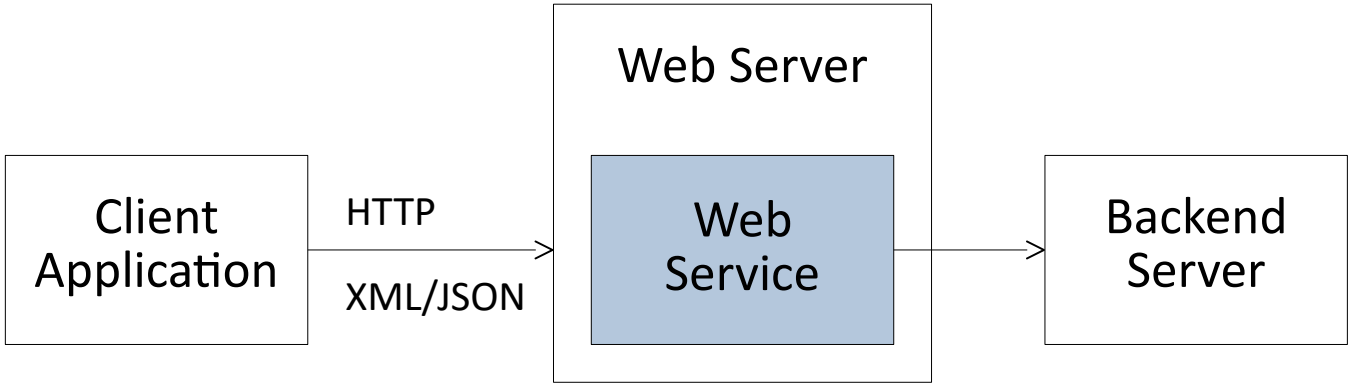
- Web services provide the means for application integration, especially between business partners

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/JSON, 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



Standards

W3C (www.w3.org):

- Hypertext Transfer Protocol (HTTP)
- Extensible Markup Language (XML)
- Simple Object Access Protocol (SOAP)
- Web Services Definition Language (WSDL)

OASIS (www.oasis-open.org)

- Web Services Security (WSS)
- Business Process Execution Language (BPEL)

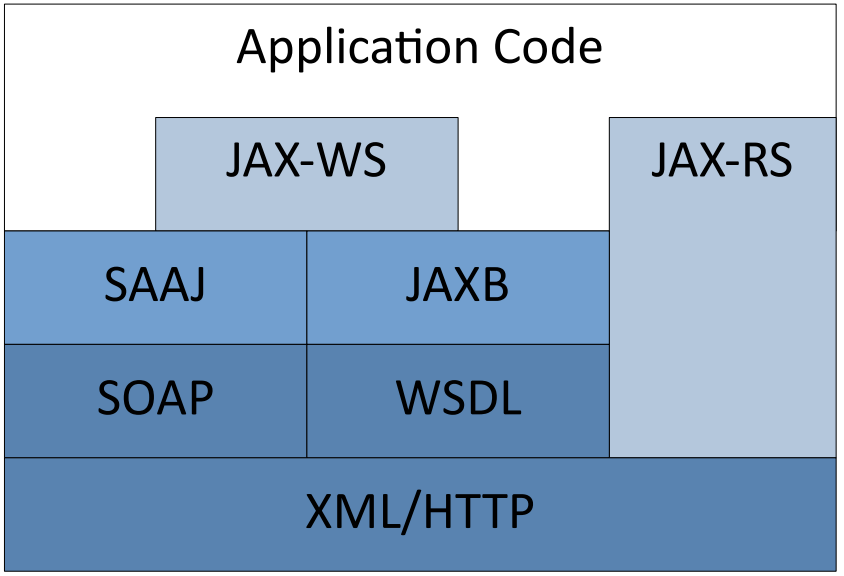
WS-I (www.ws-i.org)

- Web Services Interoperability Profile

Java APIs

Java API for
XML Web Services

SOAP with Attach-
ments API for Java



Java API for
RESTful Web Services

Java Architecture
for XML Binding

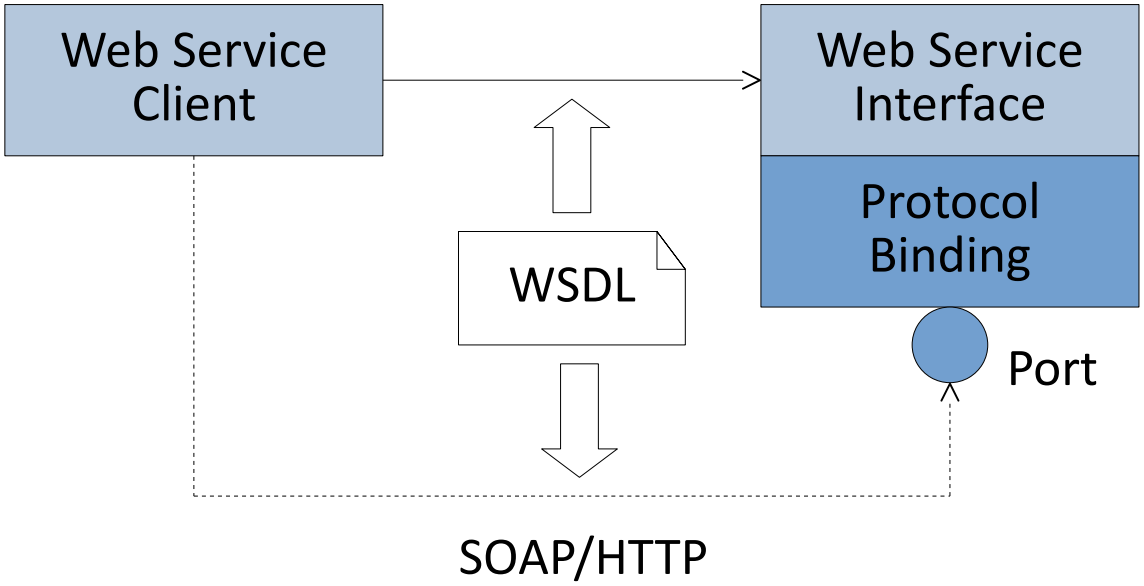
References

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

2 SOAP Web Services

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

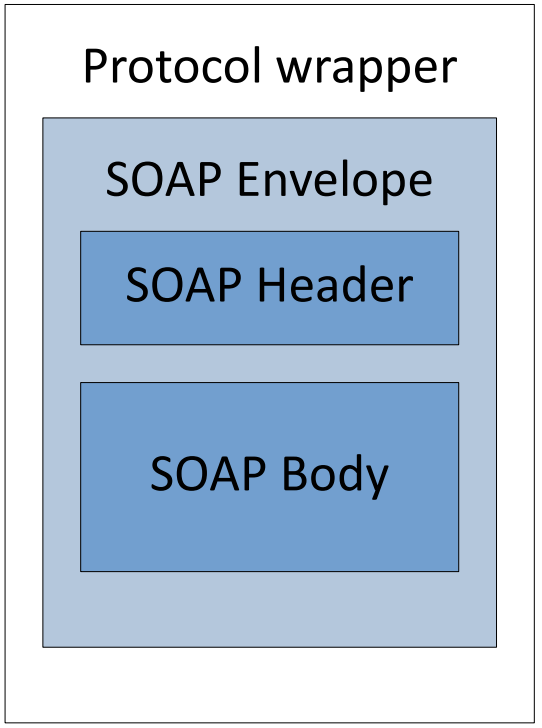


SOAP Protocol

SOAP is an XML-based communication protocol that defines

- the structure of messages
- bindings to transport protocols (HTTP POST requests)
- encoding rules for data types (e.g. arrays)
- conventions for remote procedure calls (e.g. parameters)
- 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 /soap/todolist 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:FindTodo xmlns:tns="http://example.org/todo">
         <id>5</id>
      </tns:FindTodo>
   </soap:Body>
</soap:Envelope>
```

SOAP Headers

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

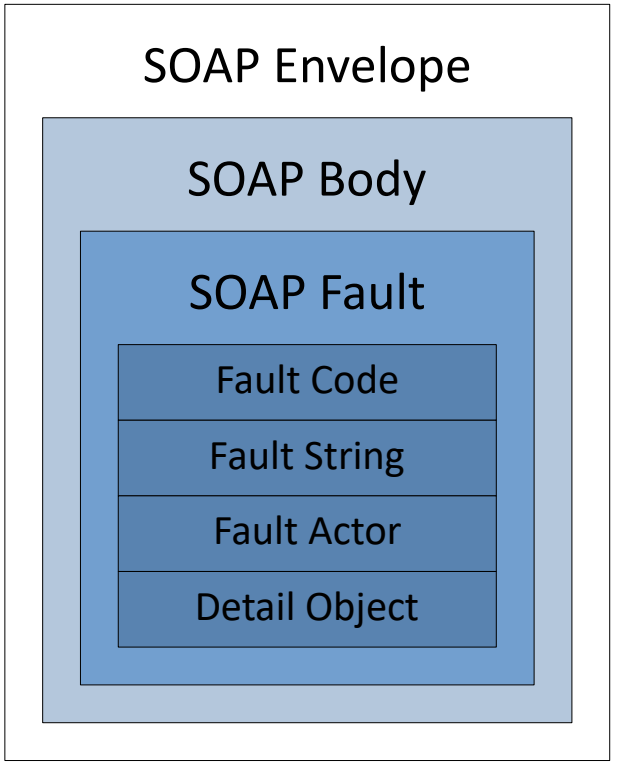
- authentication and authorization
- transaction management
- tracing and auditing
- payment information

Example of a SOAP Message with Header

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header>
      <authToken xmlns="http://example.org/todo">*****</authToken>
   </soap:Header>
   <soap:Body>
      <tns:AddTodo xmlns:tns="http://example.org/todo">
         <title>...</title>
         <dueDate>...</dueDate>
      </tns:AddTodo>
   </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>Todo with id 5 not found</faultstring>
         <detail>
            <tns:TodoNotFound xmlns:tns="http://example.org/todo">
               ...
            </tns:TodoNotFound>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

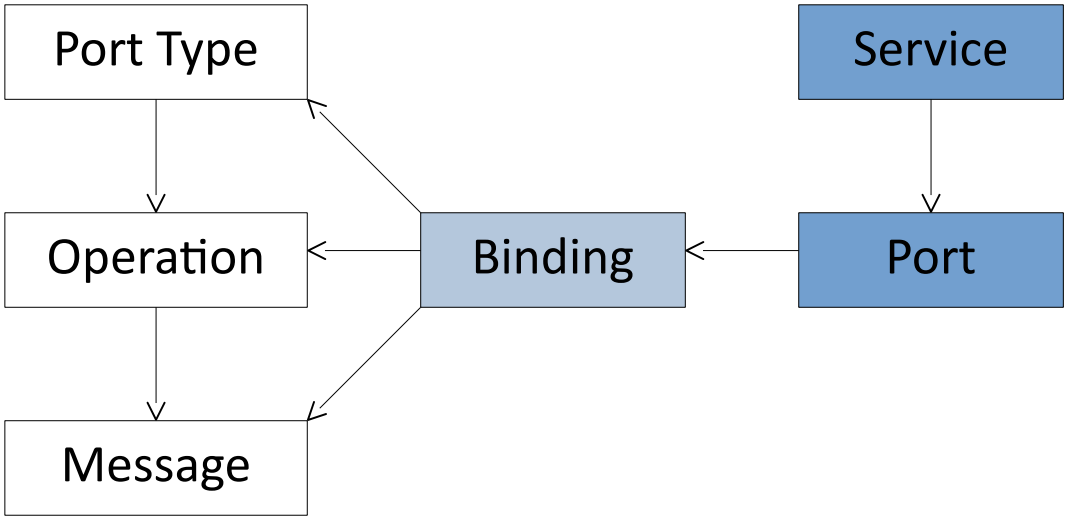
Web Services Description Language (WSDL)

- 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="FindTodo">...</message>
   <message name="FindTodoResponse">...</message>
   ...
   <portType name="TodoList">
      <operation name="FindTodo">...</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="TodoListService"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://example.org/todo"
        xmlns:tns="http://example.org/todo"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    ...
</definitions>
```

Imports

- 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/todo"
        location="TodoListGeneric.wsdl"/>
```

Data Types

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

```
<types>
   <xsd:schema targetNamespace="http://example.org/todo">
      <xsd:complexType name="FindTodo">
         <xsd:sequence><element name="id" type="long"/></xsd:sequence>
      </xsd:complexType>
      ...
   </xsd:schema>
</types>

<types>
   <xsd:schema>
      <xsd:import namespace="http://example.org/todo"
                  schemaLocation="TodoListTypes.xsd"/>
   </xsd: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="FindTodo">
   <part name="parameters" element="tns:FindTodo"/>
</message>
<message name="FindTodoResponse">
   <part name="parameters" element="tns:FindTodoResponse"/>
</message>
<message name="TodoNotFoundException">
   <part name="TodoNotFoundException" element="tns:TodoNotFound"/>
</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="TodoList">
   <operation name="FindTodo">
      <input name="FindTodo" message="tns:FindTodo"/>
      <output name="FindTodoResponse" message="tns:FindTodoResponse"/>
      <fault name="TodoNotFoundException"
             message="tns:TodoNotFoundException"/>
   </operation>
   ...
</portType>
```

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="TodoListBinding" type="tns:TodoList">
   ...
   <operation name="FindTodo">
      ...
      <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="FindTodo">
      <soap:operation soapAction=""/>
      <input><soap:body use="literal"/></input>
      <output><soap:body use="literal"/></output>
      <fault><soap:fault name="TodoNotFoundException" use="literal"/></fault>
   </operation>
</binding>
```

The Endpoint Address

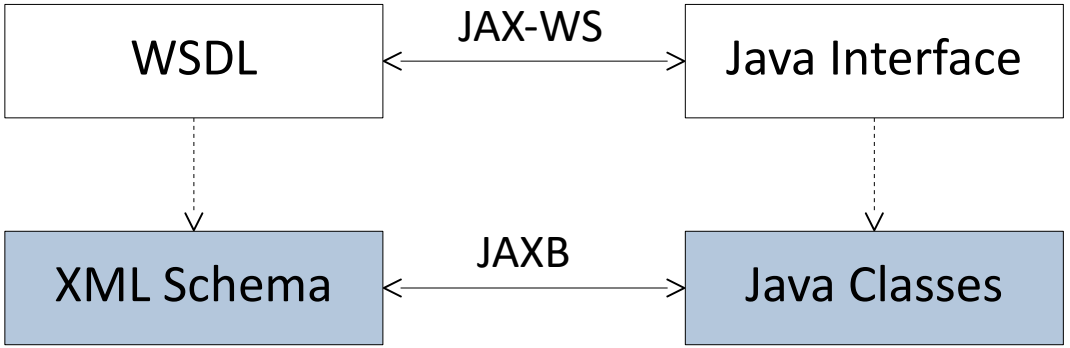
- A port element maps a binding of a port type to a URI which can be used to access it
- A service element groups together a set of related ports

```
<service name="TodoListService">
   <port name="TodoListPort" binding="tns:TodoListBinding">
      <soap:address location="http://distsys.ch:8080/soap/todolist"/>
   </port>
</service>
```

3 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



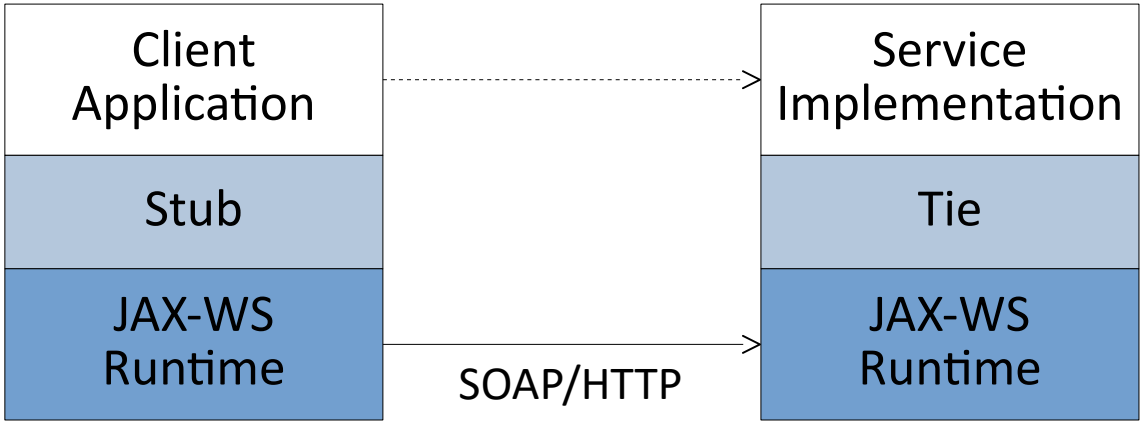
Java to WSDL Mapping

- 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 TodoList {
    public Todo findTodo(long id) throws TodoNotFoundException { ... }
}
```

```
<wsdl:portType name="TodoList">
    <wsdl:operation name="findTodo">
        <wsdl:input name="..." message="tns:findTodo"/>
        <wsdl:output name="..." message="tns:findTodoResponse"/>
        <wsdl:fault name="..." message="tns:TodoNotFoundException"/>
    </wsdl:operation>
</wsdl:portType>
```

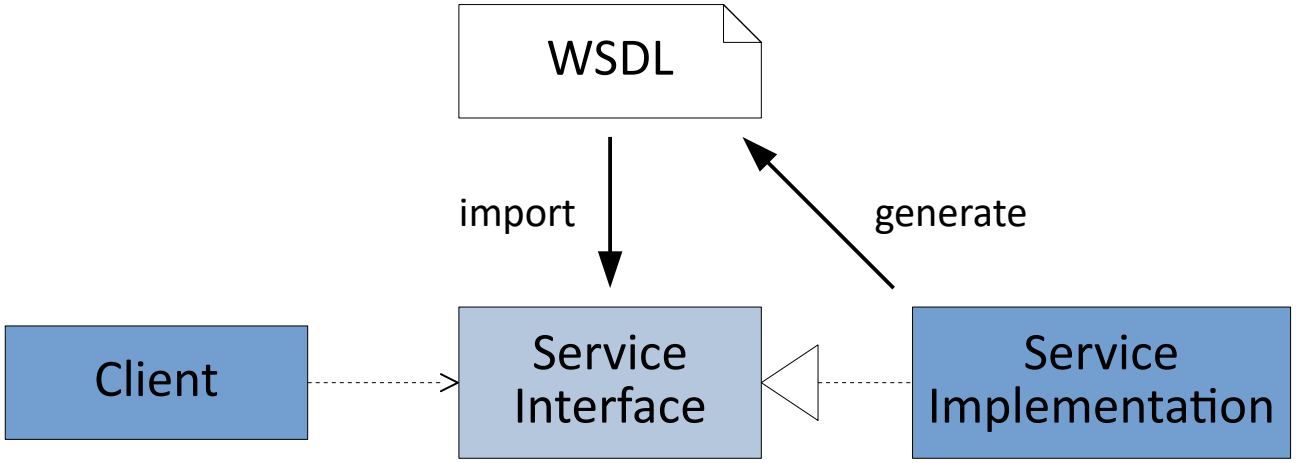
JAX-WS Architecture



- 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

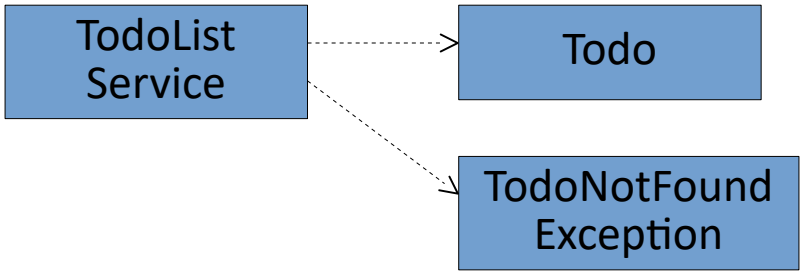
Code-first vs Contract-first

- 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)



Developing a Code-first Web Service

1. Implement and compile the web service class
2. Package the service code into a web archive
3. Deploy the web archive into a running web container whereby the WSDL is generated and published



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="TodoList", serviceName="TodoListService",
    targetNamespace="http://example.org/todo")
@SOAPBinding(...)
public class TodoListService {
    ...
}
```

The Web Service Lifecycle

- The web service class must have a public no-argument constructor
- The @PostConstruct and @PreDestroy annotations can be used for lifecycle callback events

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

   @PostConstruct
   protected void init() { ... }
   @PreDestroy
   protected 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 TodoListService {
   @WebMethod(operationName="FindTodo")
   @WebResult(name="todo")
   public Todo findTodo(@WebParam(name="id") long id)
      throws TodoNotFoundException {
      ...
   }
}
```

Exceptions

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

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

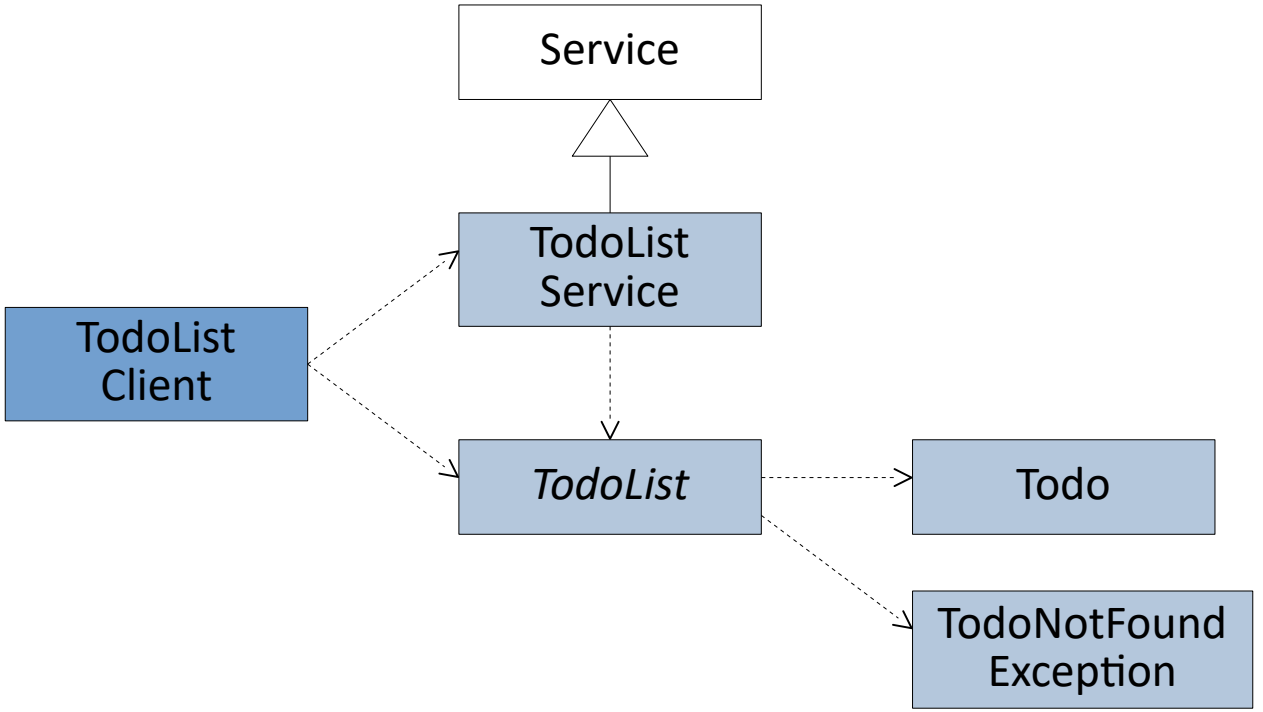
Oneway Operations

- 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 TodoListService {
    ...
    @Oneway @WebMethod(...)
    public void removeTodo(@WebParam(name="id") long id) { ... }
}
```

Developing a Web Service Client

1. Generate the artifacts needed to connect to the web service using an import command line utility (or a Maven plugin)
2. Implement and compile the client application
3. Run the client



Generation of Client Artifacts

- The import utility reads the WSDL document of a web service and generates the portable artifacts used to implement a client
- The code generation can be customized by command line options and by providing a bindings file
- The generated files include a service factory, the endpoint interface and wrapper classes

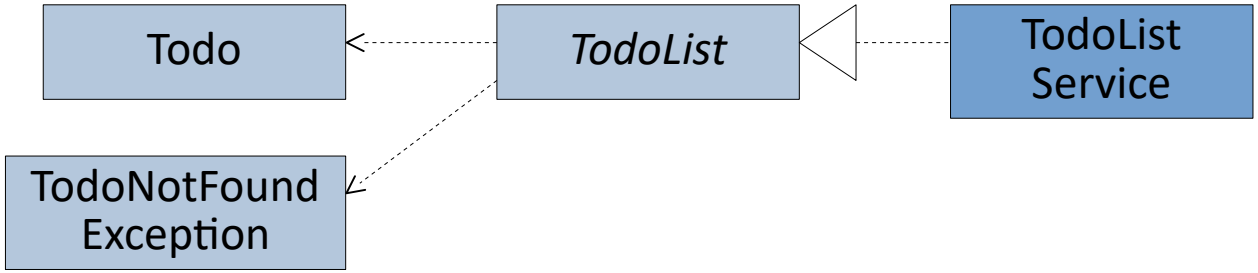
The Client Implementation

- 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 TodoListClient {
    public static void main(String[] args) throws Exception {
        TodoListService service = new TodoListService();
        TodoList todoList = service.getTodoListPort();
        if (args.length > 0)
            ((BindingProvider)todoList).getRequestContext()
                .put(ENDPOINT_ADDRESS_PROPERTY, args[0]);
        Todo todo = todoList.findTodo(id);
        ...
    }
}
```

Developing a Contract-first Web Service

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



The Web Service Implementation

- The implementation class has to implement the generated service interface
- In the @WebService annotation the endpoint interface is referenced and the location of the WSDL document can be defined

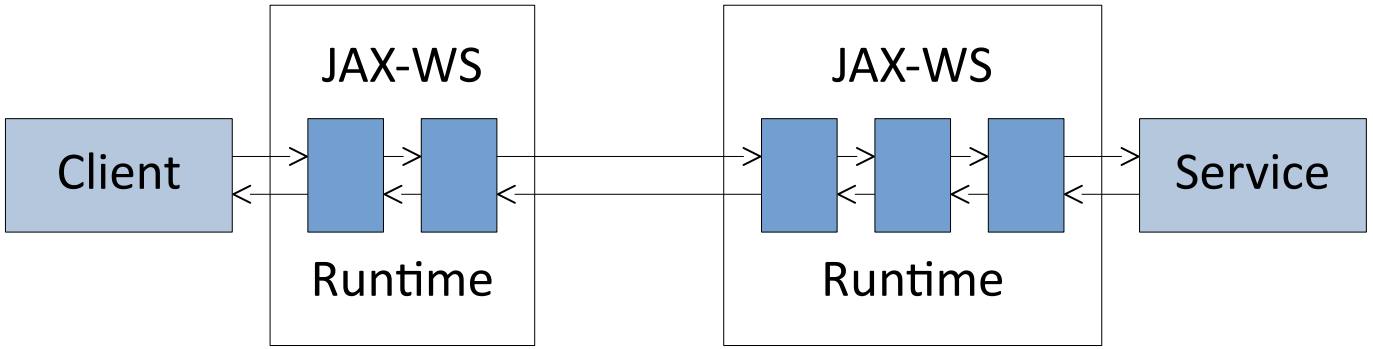
```
@WebService(endpointInterface="org.example.todo.TodoList",
    wsdlLocation="TodoListService.wsdl")
public class TodoListService implements TodoList {
    ...
}
```

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 can be applied to web service classes
- It is possible to generate client artifacts that 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 boolean return values indicate whether to continue processing of the handler chain or not

```
public class LogHandler implements SOAPHandler<SOAPMessageContext> {
    public boolean handleMessage(SOAPMessageContext context) {
        SOAPMessage message = context.getMessage();
        boolean outbound =
            (Boolean) context.get(MESSAGE_OUTBOUND_PROPERTY);
        if (!outbound) 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>LogHandler</handler-name>
         <handler-class>org.example.todo.LogHandler</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 TodoListService { ... }
```

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 TodoList {
    public Todo findTodo(long id);
    public Response<FindTodoResponse> findTodoAsync(long id);
    public Future<?> findTodoAsync(
        long id, AsyncHandler<FindTodoResponse> asyncHandler);
    ...
}
```

Polling Client

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

```
public class TodoListClient {
    public static void main(String[] args) {
        TodoListService service = new TodoListService();
        TodoList todoList = service.getTodoListPort();
        Response<FindTodoResponse> response =
            todoList.findTodoAsync(long id);
        while (!response.isDone()) {
            ...
        }
        Todo todo = response.get().getTodo();
        ...
    }
}
```

Client with Callback

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

```
public class TodoListClient {
    public static void main(String[] args) {
        TodoListService service = new TodoListService();
        TodoList todoList = service.getTodoListPort();
        todoList.findTodoAsync(id, new FindTodoHandler());
        ...
    }
}
public class FindTodoHandler implements AsyncHandler<FindTodoResponse>
{
    public void handleResponse(Response<FindTodoResponse> response) {
        Todo todo = response.get().getTodo();
        ...
    }
}
```

4 REST Principles

Introduction

- REST (representational state transfer) is an architectural style in which data and functionality are considered as resources
- The fundamental principles of REST resources are
  - identifiers
  - uniform interface (operations)
  - representations
  - hyper linking
  - stateless communication
- REST web services are web services that adhere to the REST principles using the HTTP protocol

Identifiers

- Each resource has a unique identifier by being addressable via a uniform resource identifier (URI), e.g.

  http://example.org/customers/12
  http://example.org/products/4554
  http://example.org/orders?year=2020

- A resource may contain subresources which is usually reflected in the corresponding URIs
- If a resource is addressed by a URI that does not exist, the error status *Not Found* (404) will be returned

Operations

- REST resources support a uniform interface, as they can be accessed by the standard HTTP methods

Method	Purpose	Safe	Idempotent
POST	Create a resource	no	no
GET	Read a resource	yes	yes
PUT	Update or create a resource	no	yes
PATCH	Update a resource	no	yes
DELETE	Delete a resource	no	yes
HEAD	Get information on a resource	yes	yes
OPTIONS	Get communication options	yes	yes

- Safe operations do not modify resources, idempotent operations can be invoked multiple times with the same outcome
- If a resource does not support a specific method, the error status *Method Not Allowed* (405) is returned

Operations (cont.)

- Both operations, POST and PUT, can create a resource
  - POST is used when the identifier of the new resource is determined by the web service
  - PUT is used when the identifier of the new resource is known by the client in advance

- Both operations, PUT and PATCH, can update a resource
  - PUT is used to update a resource completely
  - PATCH is used to update a resource partially

Data and Parameters

- Data can be provided in the HTTP message body
- Additionally, request parameters can be provided as
  - path parameters
    http://example.org/customers/12
  - query parameters
    http://example.org/orders?year=2020
  - header parameters
    Content-Type: application/json
  - cookie parameters
    Cookie: transactionId=555
  - form parameters (in the request body)
    productId=4554
    customerId=12

Representations

- A client can ask for a particular format of a resource using the Accept header

```
GET /customers/12 HTTP/1.1
Accept: application/json
```

- If the service is able to produce the requested format, it delivers the data specifying its format using the Content-Type header

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 50

{ "id": 12, "name": "John Doe", "email": "jd@example.org" }
```

- If the service does not support the requested format, it sends the error status *Not Acceptable* (406)

Representations (cont.)

- When the client sends data to the server, it specifies the format using the Content-Type header

```
POST /customers HTTP/1.1
Content-Type: application/json
Content-Length: 50

{ "name": "John Doe", "email": "jd@example.org" }
```

- If the server is not able to consume the provided format, it responds with the error status *Unsupported Media Type* (415)

Hyper Linking

- When delivering the data of a resource that is related to other resources, hyper links should be included to refer to those resources

```
<order id ="776654">
   <amount>23</amount>
   <product ref="http://example.org/products/4554"/>
   <customer ref="http://example.org/customers/12"/>
</order>
```

- With this, a client can dynamically follow the links to retrieve more information or even to change the application state
- This principle is also referred to as HATEOAS (hypermedia as the engine of applicaton state)

Communication

- With REST web services, the communication between the client and server is stateless, i.e. the server does not retain any communication state of the clients
- Thus, the application state is either turned into resource state or kept on the client

5 Java API for RESTful Web Services (JAX-RS)

Introduction

- JAX-RS is a Java API designed to develop RESTful web services
- Java classes are annotated to define resources and the actions to be performed on those resources
- HTTP requests are dispatched to Java methods depending on the method, path and mime type of the requests
- Resource classes and artifacts are packaged into a web archive and deployed to a Java EE server

JAX-RS Annotations

Annotation	Description
@Path	relative URI of the resource
@POST, @GET @PUT, @DELETE @HEAD, @OPTIONS	method designators indicating the HTTP request type to be processed
@Path-, @Query- @Form-, @Header- @CookieParam	parameters to be extracted from the HTTP request for use in the resource class
@Consumes @Produces	MIME media types of representations a resource can consume or produce
@Provider	information for the JAX-RS runtime such as message body reader and writer
@Context	information related to a request

The Path Annotation

- The @Path annotation identifies the partial URI path of a resource and can be specified at the class and method level
- The path of a method is appended to the path of the service class
- The paths are interpreted relative to the URL of the server, the context root and the application path
- A path may contain variables whose values are mapped to method parameters using the @PathParam annotation

```
@Path("todos")
public class TodoResource {
    @Path("{id}")
    public Todo findTodo(@PathParam("id") long id) { ... }
    ...
}
```

Request Method Annotations

- The request method annotations (@POST, @GET, @PUT, @DELETE) are used to map HTTP requests to corresponding methods of the resource class
- The resource methods must return void, a Java type or a Response object

```
@Path("todos")
public class TodoResource {
    @GET @Path("{id}")
    public Todo findTodo(@PathParam("id") long id) { ... }
    @POST
    public long addTodo(Todo todo) { ... }
    @PUT @Path("{id}")
    public void updateTodo(Todo todo) { ... }
    @DELETE @Path("{id}")
    public void removeTodo(@PathParam("id") long id) { ... }
}
```

Media Type Annotations

- The @Consumes and @Produces annotations specify the MIME media types a resource can consume or produce
- A client specifies the media type of request or response data using the HTTP headers Content-Type and Accept
- If a resource is not able to consume or to produce the requested media type, an HTTP error status (unsupported media type, not acceptable) is returned

```
@Path("todos")
public class TodoResource {
    @POST
    @Consumes(APPLICATION_JSON)
    @Produces(TEXT_PLAIN)
    public long addTodo(Todo todo) { ... }
    ...
}
```

Parameter-based Annotations

- Parameters of a resource method may be annotated to extract information from the HTTP request

Annotation	Parameter Source
@PathParam	Request URI path
@QueryParam	Request URI query
@FormParam	Request URI path or body
@HeaderParam	HTTP headers
@CookieParam	Cookie-related headers

- Parameters which are not annotated, are mapped from the HTTP request body (entity parameters)

Exception Handling

- If an error occurs in a resource method, a WebApplicationException can be thrown which produces an HTTP error response

```
@GET @Path("id")
public Todo findTodo(@PathParam("id") long id) {
    try { return todoList.find(id); }
    catch (TodoNotFoundException ex) {
        throw new WebApplicationException(ex.getMessage(),
            Response.status(NOT_FOUND).entity(ex.getMessage()).build());
    }
}
```

Entity Providers

- Message body readers and writers supply mappings between representations of resources and their corresponding Java types
- JAX-RS supports some standard types such as byte[], String and JAXB classes which are mapped automatically

```
@Provider
@Produces("text/csv")
public class CSVMessageWriter implements MessageBodyWriter<List<Todo>>
{
    public void writeTo(List<Todo> todos, ..., OutputStream stream) {
        PrintWriter writer = new PrintWriter(stream);
        for (Todo todo : todos)
            writer.println(todo.getId() + "," + todo.getTitle() + "," +
                           todo.getDueDate() + "," + todo.getCompleted());
        writer.flush();
    }
    ...
}
```

Context Information

- The @Context annotation can be used to obtain information about the HTTP request

```
@Path("todos")
public class TodoResource {
    @Context Request request;
    @Context UriInfo uriInfo;
    @Context HttpHeaders headers;
    ...
    private void log() {
        System.out.println(request.getMethod());
        System.out.println(uriInfo.getRequestUri());
        System.out.println(headers.getRequestHeaders());
    }
}
```

Application Configuration

- The resources and providers of a JAX-RS application must be configured via a subclass of Application
- The @ApplicationPath annotation is used to specify the application path which will be part of the resources' URIs

```
@ApplicationPath("rest")
public class ApplicationConfig extends Application {
    public Set<Class<?>> getClasses() {
        Set<Class<?>> classes = new HashSet<>();
        classes.add(TodoResource.class);
        classes.add(CSVMessageWriter.class);
        return classes;
    }
}
```

RESTful Web Service Client API

- JAX-RS provides a standardized client API to easily consume RESTful web services
- A client application first creates an instance of the Client class which configures and manages the HTTP connections
- The Client object is also a factory for WebTarget objects which encapsulate resources identified by a URI

```
String resourceURI = "http://distsys.ch:8080/rest/todos"
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target(resourceURI);
```

Sending a Request

- The WebTarget object can be used to specify the relative path of the resource and query parameters
- When building the request, the accepted response media types and other headers can be added
- Finally, the HTTP method is invoked, possibly passing an Entity object which will be included in the request body

```
webTarget.path(id).request(APPLICATION_JSON).get();
webTarget.queryParam("dueDate", date).request(APPLICATION_JSON).get();
webTarget.request(TEXT_PLAIN).post(Entity.json(todo));
webTarget.path(id).request().put(Entity.json(todo));
webTarget.path(id).request().delete();
```

Receiving a Response

- The invocation of the HTTP method returns a Response object from which the response status and headers can be obtained
- In order to get a response body, the readEntity method is invoked passing the object type of the body

```
Response response = webTarget.path(id).request(APPLICATION_JSON).get();
if (response.getStatus() == Status.OK.getStatusCode()) {
    Todo todo = response.readEntity(Todo.class);
    ...
}
```

Advanced Features

- Exception mappers can be used to map exceptions to HTTP responses
- Server-side asynchronous HTTP allows to implement long-polling interfaces or server-side push
- With asynchronous client API, client applications obtain the HTTP response using polling or a callback
- Filters allow to modify or process incoming and outgoing HTTP messages
- Entity Interceptors are concerned with the marshaling and unmarshaling of message bodies