02267: Software Development of Web Services

Week 3

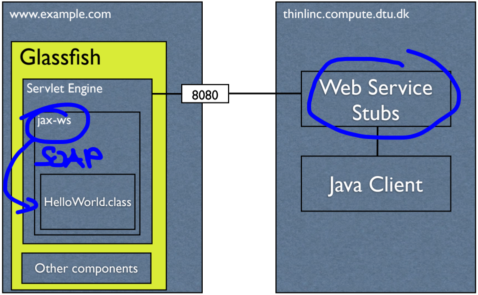
Hubert Baumeister

huba@dtu.dk

Department of Applied Mathematics and Computer Science
Technical University of Denmark

Fall 2015

Recap



- Monitoring Web Services
- XML + XML Namespaces
- Basic SOAP
- ~~HTTP + SOAP~~

Contents

HTTP & SOAP

SOAP Part II

WSDL

How to create Web services

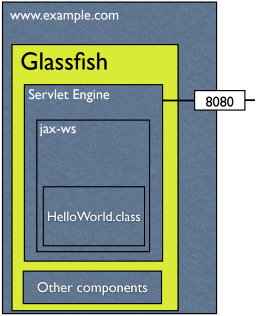
WS Architecture

- SOAP messages require a transport protocol for transmission (e.g. HTTP, SMTP, . . .)

→ Commonly used HTTP

  - SOAP messages are transported by the HTTP protocol via POST requests

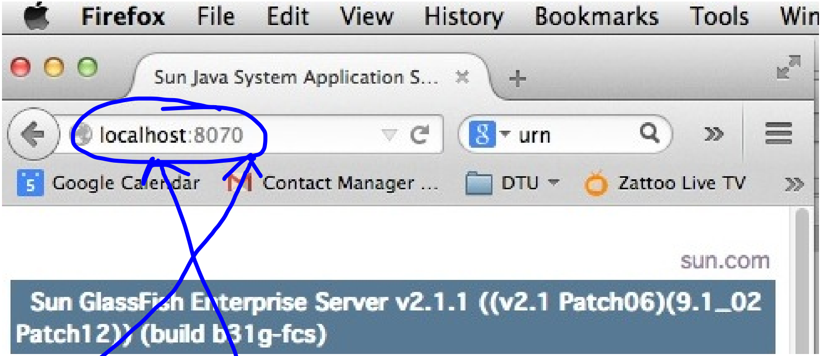
- Distinction between message layer (SOAP) and transport layer (HTTP, SMTP, . . .)



HTTP

- HTTP = Hypertext Transfer Protocol
- The basic protocol for the Web
- Is used to retrieve Web sites from Web servers
  - Static Web pages
  - On the fly generated Web pages: CGI bins / Servlets in Java / ...
- HTTP/1.1 is defined in RFC 2616 (`http://www.w3.org/Protocols/`)

HTTP Get Request



Sun GlassFish Enterprise Server v2.1.1 ((v2.1 Patch06)(9.1_02 Patch12)) (build b31g-fcs)

```
GET / HTTP/1.1
Host: 127.0.0.1:8070
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1.1) Gecko/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

HTTP Answer

Status

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun GlassFish Enterprise Server v2.1
ETag: W/"4827-1251914047000"
Last-Modified: Wed, 02 Sep 2009 17:54:07 GMT
Content-Type: text/html
Content-Length: 4827
Date: Fri, 04 Sep 2009 00:04:56 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
<html lang="en">
<head>
...
</body>
</html>
```

HTTP Access Methods

- Most common access methods:
  - GET: Get information, such as a Web page from the server
  - POST: Submit information (e.g. forms) to the Web server and wait for an answer
  - PUT: Replace the resource identified by the request URI with the enclosed information
  - DELETE: Delete the resource identified by the request URI
  - . . .
- POST: used by Web services to send a SOAP message
- GET, POST, PUT, DELETE

  → REST (Representational State Transfer) architectural principle
  - used with RESTful Web services

SOAP over HTTP

- HTTP requests must be POST request
- content-type must be text/xml
- HTTP header must have a SOAPAction field SOAPAction: ”URI”. This identifies the action/service and allows firewalls to detect and handle SOAP messages. Ex:
  - SOAPAction ”http://electrocommerce.org/abc#MyMessage”
  - SOAPAction ”axis/EchoString.jws”
  - SOAPAction ”” means that the URI of the HTTP request is the SOAPAction URI

SOAP over HTTP example: Web Service Request

HTTP

```
POST /Echo/EchoService HTTP/1.1
SOAPAction: ""
Accept: text/xml, multipart/related, text/html, image/jpg, image/jpeg,
Content-Type: text/xml; charset=utf-8
User-Agent: Java/1.6.0_07
Host: 127.0.0.1:8070
Connection: keep-alive
Content-Length: 182
```

SOAP

```
<?xml version="1.0" ?>
   <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
      <S:Body>
         <ns2:echo xmlns:ns2="http://week01/">
            <arg0>Hello</arg0>
         </ns2:echo>
      </S:Body>
   </S:Envelope>
```

SOAP over HTTP example: Answer

4xx

5xx

HTTP

SOAP

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun GlassFish Enterprise Server v2.1
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Fri, 04 Sep 2009 00:11:03 GMT
ca
<?xml version="1.0" ?>
   <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
      <S:Body>
         <ns2:echoResponse xmlns:ns2="http://week01/">
            <return>Hello</return>
         </ns2:echoResponse>
      </S:Body>
   </S:Envelope>
0
```

Contents

HTTP & SOAP

SOAP Part II

WSDL

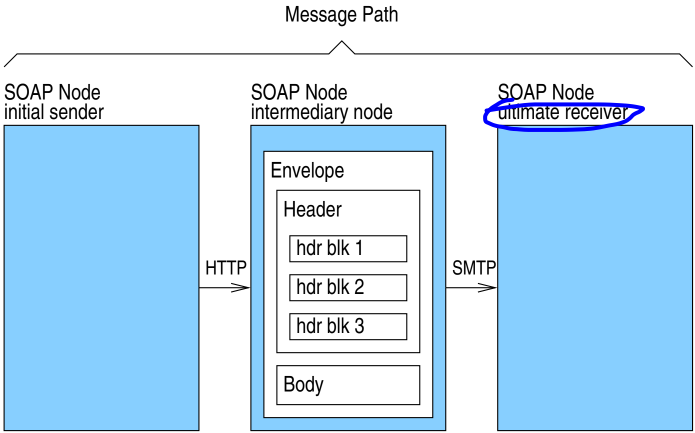
How to create Web services

Recap: Structure of a SOAP Document

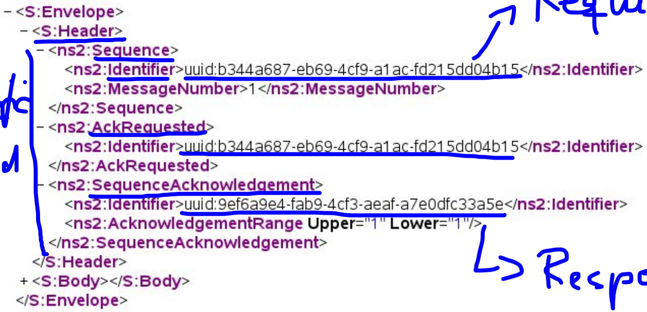
```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/
 <S:Header>
  .. Application specific XML with their own namespace ..
   (several elements possible)
 </S:Header>
 <S:Body>
  .. Application specific XML  with their own namespace
     or <S:Fault> element ..
   (one element only)
 </S:Body>
</S:Envelope>
```

SOAP Messaging Model

Logical view



Example: WS-Reliable Messaging



```
- <S:Envelope>
  - <S:Header>
    - <ns2:Sequence>
        <ns2:Identifier>uuid:b344a687-eb69-4cf9-a1ac-fd215dd04b15</ns2:Identifier>
        <ns2:MessageNumber>1</ns2:MessageNumber>
      </ns2:Sequence>
    - <ns2:AckRequested>
        <ns2:Identifier>uuid:b344a687-eb69-4cf9-a1ac-fd215dd04b15</ns2:Identifier>
      </ns2:AckRequested>
    - <ns2:SequenceAcknowledgement>
        <ns2:Identifier>uuid:9ef6a9e4-fab9-4cf3-aeaf-a7e0dfc33a5e</ns2:Identifier>
        <ns2:AcknowledgementRange Upper="1" Lower="1"/>
      </ns2:SequenceAcknowledgement>
    </S:Header>
  + <S:Body></S:Body>
  </S:Envelope>
```

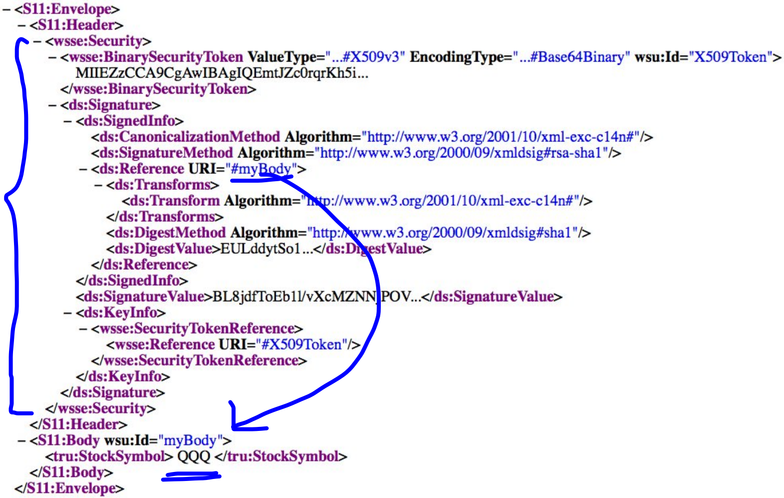
~~App specific~~

Standard

Requests

Responses

Example: WS-Security



```
– <S11:Envelope>
  – <S11:Header>
    – <wsse:Security>
      – <wsse:BinarySecurityToken ValueType="...#X509v3" EncodingType="...#Base64Binary" wsu:Id="X509Token">
          MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
        </wsse:BinarySecurityToken>
      – <ds:Signature>
        – <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          – <ds:Reference URI="#myBody">
            – <ds:Transforms>
                <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
              </ds:Transforms>
              <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
              <ds:DigestValue>EULddytSo1...</ds:DigestValue>
            </ds:Reference>
          </ds:SignedInfo>
          <ds:SignatureValue>BL8jdfToEb1l/vXcMZNNjPOV...</ds:SignatureValue>
        – <ds:KeyInfo>
          – <wsse:SecurityTokenReference>
              <wsse:Reference URI="#X509Token"/>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>
        </ds:Signature>
      </wsse:Security>
    </S11:Header>
  – <S11:Body wsu:Id="myBody">
      <tru:StockSymbol> QQQ </tru:StockSymbol>
    </S11:Body>
  </S11:Envelope>
```

SOAP Header

SOAP header can be used, among others, to

- include processing instructions for the service intermediaries
  - like signing / encrypting / decrypting a message
  - logging a message
- routing of messages
  - who should be dealing with that message?
- context- / meta data and transaction management
  - transaction identifier / start of transaction / end of transaction / common data
  - information necessary to establish reliable messaging

How faults are handled: Unchecked Exceptions

```
public int divide(int a, int b) { return a/b; }
```

- What happens if the Web services is called with *b* is 0?

How faults are handled: Unchecked Exceptions

```
public int divide(int a, int b) { return a/b; }
```

- What happens if the Web services is called with *b* is 0?
- On the client: exception is caught in the client stubs and converted to a SOAP Fault in the SOAP message
- On the server: Receiving a SOAP message containing a SOAP fault throws a SOAPFaultException

Fault example: Unchecked Exception

- Service called with $b = 0$

```
public int divide(int a, int b) { a / b; }
```

- HTTP status is 500 (Internal Server Error)
- SOAPFault in the return SOAP message:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="...">
  <S:Body>
    <S:Fault>
      <faultcode>S:Server</faultcode>
      <faultstring>/ by zero</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

- Client receives a SOAPFaultException

```
@Test(expected=SOAPFaultException.class)
public void testDivide() {
   divide(4,0);
}
```

How faults are handled: Checked Exceptions

```
public int divide2(int a, int b) throws MyOwnException {
   if (b == 0) {
      throw new MyOwnException(a,b);
   }
   return a/b;
}
```

MyOwnException

```
public class MyOwnException extends Exception {
   int a;
   int b;
   public MyOwnException(int a, int b) {
      super(String.format("Division by zero %d/%d",a,b));
      this.a = a;
      this.b = b;
   }
   ...
}
```

How faults are handled: Checked Exceptions

```
public int divide2(int a, int b) throws MyOwnException {
   if (b == 0) {
      throw new MyOwnException(a,b);
   }
   return a/b;
}
```

How faults are handled: Checked Exceptions

```
public int divide2(int a, int b) throws MyOwnException {
   if (b == 0) {
      throw new MyOwnException(a,b);
   }
   return a/b;
}
```

- Exception is converted to a custom exception

```
@Test
public void testDivide2() throws MyOwnException_Exception {
   try {
      divide2(4,0); fail();
   } catch (MyOwnException_Exception e) {
      MyOwnException fi = e.getFaultInfo();
      assertEquals(4,fi.getA());
      assertEquals(0,fi.getB());
   }
}
```

Fault example: Checked Exception

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>Division by zero 4/0</faultstring>
      <detail>
        <ns2:MyOwnException xmlns:ns2="http://dk.dtu.ws/">
          <a>4</a>
          <b>0</b>
          <message>Division by zero 4/0</message>
        </ns2:MyOwnException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

App. specific

faultInfo

<Fault> (SOAP 1.1)

- `<faultcode>`
  - Possible contents
    - `VersionMismatch`
    - `MustUnderstand`
    - `Client` (error lies with the sender of the message)
    - `Server` (error lies with the receiver of the message)
  - `Faultcode.xxx` (subclassification of the fault code)

    e.g. `Server.UserException`
- `<faultstring>` (human readable explanation of the fault)
- `<faultactor>` (URI identifying the faulty actor)
- `<detail>`
  - Application defined XML

Contents

HTTP & SOAP

SOAP Part II

WSDL

How to create Web services

Web Service Description Language (WSDL)

- To use Web services in a loosely coupled system, Web services need to be described, so that their description can be used to discover them
- History
  - WSDL 1.1, while it is a de-facto standard, it is not endorsed by W3C
  - WSDL 2.0 is a W3C recommendation
- Namespaces
  - `http://schemas.xmlsoap.org/wsdl/` (WSDL 1.1)
  - `http://schemas.xmlsoap.org/wsdl/soap/` (SOAP binding)
  - `http://xml.apache.org/xml-soap` (SOAP 1.1) (predefined datatypes)
  - `http://www.w3.org/2001/XMLSchema` (XML Schema) (user defined datatypes and predefined datatypes)

WSDL for a Calculator Service

```
@WebService()
public class CalculatorService {

    public int add(int arg1, int arg2) {
        return arg1 + arg2;
    }

    public int division(int arg1, int arg2) {
        return arg1/arg2;
    }

}
```

Structure of a WSDL Document

```
<definitions name="....">
 <types>..</types>
 <message>..</message>
 ..
 <message>..</message>
 <portType>..</portType>
 ..
 <portType>..</portType>
 <binding>..</binding>
 ..
 <binding>..</binding>
 <service>..</service>
 ..
 <service>..</service>
</definitions>
```

abstract
independent
of msg. prot
(SOAP)

Concrete
- msg prot
(SOAP)
- how to acces

Creating the WSDL for the Calculator service

- Definitions element

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

- The namespace `http://calculator.ws.dtu` is defined by using the targetNamespace attribute
- But `http://calculator.ws.dtu` is also used in the WSDL file
  → its namespace needs to be declared:
  `xmlns:tns="http://calculator.ws.dtu"`

TargetNamespace and tns

- Definition of a name space and its use are separated
  → Corresponds to that Java would need to import the classes it defines in a package.

```
package tree;
   // Corresponds to targetNamespace="http://tree"
import tree.*;  → not Java but needed in XML
   // Corresponds to xmln:tns = "http://tree"

class Tree {
   Tree left, right;
   ...
}
```

Port Types

The calculatorPorttype has two operations with each one input message and one output message

```
<portType name="calculatorPortType">
 <operation name="division">
  <input name="input1" message="tns:divisionRequest"/>
  <output name="output1" message="tns:divisionResponse"/>
 </operation>
 <operation name="add">
  <input name="input2" message="tns:addRequest"/>
  <output name="output2" message="tns:addResponse"/>
 </operation>
</portType>
```

we have defined

Message Exchange Patterns: One way

- One-way: Send out a message to the Web service, but don’t wait for an answer
  - Used for notifications or asynchronous message calls
- Operation has only one input message

```
<portType name="...">
 <operation name="..">
   <input name="input1" message="tns:..."/>
 </operation>
</portType>
```

Message Exchange Patterns: Request/Response

- Request / Response: Send out a message and wait for an answer
- Operation has an input message followed by an output message
- Most commonly used

```
<portType name="...">
 <operation name="..">
   <input name="input1" message="tns:..."/>
   <output name="output1" message="tns:..."/>
 </operation>
</portType>
```

Message Exchange Patterns: Solicit-response

- Solicit-response: Wait for a message from the Web service and then answer that message
- Operation has an output followed by an input message
- Does not fit well HTTP as a transport and is not really used

```
<portType name="...">
 <operation name="..">
   <output name="output1" message="tns:..."/>
   <input name="input1" message="tns:..."/>
 </operation>
</portType>
```

Message Exchange Patterns: Notification

- Notification: The Web service sends a notification to the client
- Operation has only an output message

```
<portType name="...">
 <operation name="..">
   <output name="output1" message="tns:..."/>
 </operation>
</portType>
```

Messages

- Abstract definition of the messsage that are being exchanged between client and server
- Only the binding actually determines the concrete XML/SOAP message
- Two messages: addRequest and addResponse
- addRequest has two parts and addResponse one part

```
<message name="addRequest">
 <part name="arg1" type="xsd:int"/>
 <part name="arg2" type="xsd:int"/>
</message>
<message name="addResponse">
 <part name="result" type="xsd:int"/>
</message>
```

The binding element

- The binding element describes how the abstract port type is mapped to an actual message exchange
  - Both message layer (e.g. SOAP) and transport layer (e.g. HTTP)
- One port type can have several bindings!

```
<binding name="calculatorPortTypeBinding"
         type="tns:calculatorPortType">
 <soap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="add">
  <soap:operation/>
  <input name="input2">
   <soap:body use="literal"
    namespace="http://calculator.ws.dtu"/>
  </input>
  <output name="output2">
   <soap:body use="literal"
    namespace="http://calculator.ws.dtu"/>
  </output>
 </operation>
 ...
</binding>
```

!

RPC binding style

- Calling operation $op(p_1, \cdots, p_n)$

→ use operation name receiveOrder as root tag in the SOAP body (e.g. receiveOrder for $receiveOrder(input : String)$

- Arguments are sublements (tag name is irelevant)
- Use of tag: opResponse for the response
- Can have more than 1 return parameter

```
<s:Envelope>
 ...
 <s:Body>
  <n:op
    xmlns:n="http://ws.imm.dtu.dk/ns">
   <n:p1>...</n:p1>
   <n:p2>...</n:p2>
   ...
  </n:op>
 </s:Body>
</s:Envelope>
```

↳ parts in the message

```
<s:Envelope>
  ...
 <s:Body>
  <n:opResponse
   xmlns:n="http://ws.imm.dtu.dk/ns
  <n:p1>...</n:p1>
  <n:p2>...</n:p2>
    ...
  </n:op>
 </s:Body>
</s:Envelope>
```

Resulting SOAP message for the Calculator

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/
 <S:Body>
  <ns2:add xmlns:ns2="http://calculator.ws.dtu">
   <arg1>3</arg1>
   <arg2>4</arg2>
  </ns2:add>
 </S:Body>
</S:Envelope>
```

def

↳ binding sect.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/
 <S:Body>
  <ns2:addResponse xmlns:ns2="http://calculator.ws.dtu">
   <result>7</result>
  </ns2:addResponse>
 </S:Body>
</S:Envelope>
```

Service section

- The service section defines how to reach a service by defining a location and a binding (how to communicate with that service)
  - It is possible that the operations of one port type are offered with several endpoints and different protocols
- Services consists of a set of ports
- A port refers to a
  - portType via a binding
  - and defines an endpoint to access the service
- The service section can contain several ports each describing a way to access the operations defined in the port types by
  - providing different endpoints and/or
  - different bindings

```
<service name="calculatorService">
 <port name="calculatorPort"
      binding="tns:calculatorPortTypeBinding">
  <soap:address
     location=
"http://localhost:${HttpDefaultPort}/calculatorService/calculatorPort
 </port>
</service>
```

Endpoint of service

Contents

HTTP & SOAP

SOAP Part II

WSDL

How to create Web services

Two ways to create Web services

- Bottom up
  - Use an existing program (existing or self written) and publish it as a Web service
  - Use this if you have existing applications that one wants to publish as Web services
  - The WSDL file is generated automatically

  → This was done in the last two exercises
- Top down
  - First create a WSDL file and then an implementation fitting the port types described in the WSDL file
  - Use this when implementing a standard interface given as a WSDL file

Top down development of Web services using NetBeans

1. Create a Web application project
2. Create a WSDL file (File>New File>XML>WSDL Document)
   - ▸ Note that the SOAP-address is replaced by GlassFish on deployment
   - → Don't use this file for creating the Web service client; instead use the WSDL generated by GlassFish
3. Create a Web service implementation realising the services in a WSDL file (File>New File>Web Services>Web Service from WSDL)
   - ▸ Note: Each service needs to have a separate implementation class
   - ▸ The program uses wsimport to generate the interface that the Web Service implementation class is implementing
   - ▸ The generated files can be viewed in the Files view by going to the `build/generated/wsimport/service/` directory