02267: Software Development of Web Services

Week 4

Hubert Baumeister
huba@dtu.dk

Department of Applied Mathematics and Computer Science
Technical University of Denmark

Fall 2015

Recap

- SOAP part II: SOAP header, SOAP node, SOAP fault
- WSDL: Web Service Description Language
- Ways of developing Web services
  - Bottom up: Web services from existing programs
    - → first two lectures
- Top down or contract based: Web services from WSDL descriptions
  - Web Services are programming language independent
  - cleaner definition of services, operations, and data (i.e. using XML schema)

Contents

Student Registration RPC Binding: Example

- Operation registerStudent:
  - Provide student data (name, cpr, address)
  - Return the student number after registration
- Request

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:registerStudent
   xmlns:ns2="http://studentregistration.ws">
   <part1>
    <name>Henry</name>
    <cpr>123456</cpr>
    <address>
     <street>H.C. Andersen Boulevard</street>
     <city>Copenhagen</city>
    </address>
   </part1>
  </ns2:registerStudent>
 </S:Body>
</S:Envelope>
```

Complex datatype

Response

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:registerStudentResponse
   xmlns:ns2="http://studentregistration.ws">
     <studentNumber>5678</studentNumber>
  </ns2:registerStudentResponse>
 </S:Body>
</S:Envelope>
```

WSDL

- PortType

```
<portType name="studentregistrationPortType">
 <operation name="registerStudent">
  <input name="input1" message="tns:registerStudentRequest"/>
  <output name="output1" message="tns:registerStudentResponse"
  <fault name="fault1" message="tns:registerStudentFault"/>
 </operation>
</portType>
```

- Messages

```
<message name="registerStudentRequest">
 <part name="part1" type="tns:studentType"/>
</message>
<message name="registerStudentResponse">
 <part name="studentNumber" type="xsd:string"/>
</message>
```

- Question: how to define the studentType?
- Answer: Use of XML schema

XML Schema

- Describes (valid) XML Documents using XML
  - SOAP, WSDL, ...
  - Data used with Web services (e.g. in the types section of WSDL documents)
  - ...
- Compare with Document Type Definitions (DTD)
  - which has the same purpose
  - but
    - XML Schema use again XML (XML Schema definition for XML Schema :-) $\rightarrow$ no special parser needed!
    - and XML Schema is more flexible than DTD

References

- XML Schema 1.0; XML Schema 1.1 is being developed
  - http://www.w3.org/TR/xmlschema-0/ (Part 0: Primer)
  - http://www.w3.org/TR/xmlschema-1/ (Part 1: Structure)
  - http://www.w3.org/TR/xmlschema-2/ (Part 2: Datatypes)
  - http://www.w3.org/2001/XMLSchema.xsd (XML Schema Definition for XML Schema)
- XML Schema Best Practices: http://www.xfront.com/BestPracticesHomepage.html
- Here we only provide the basics of XML Schema sufficient for most Web services

XML schema

- XML has
  - Elements ✓
  - Attributes ✓
  - Entities ✓ ü → ü
- XML Schema defines
  - Elements ✓
  - Attributes ✓
  - Types for attributes and elements (Simple Types and Complex Types)
    → Attributes + elements
    → only for elements
  - Entities
  - ...
- Focussing on: element-, attribute-, and complex type definitions; XML Primer and best practice document gives more information

XML Schema: schema element

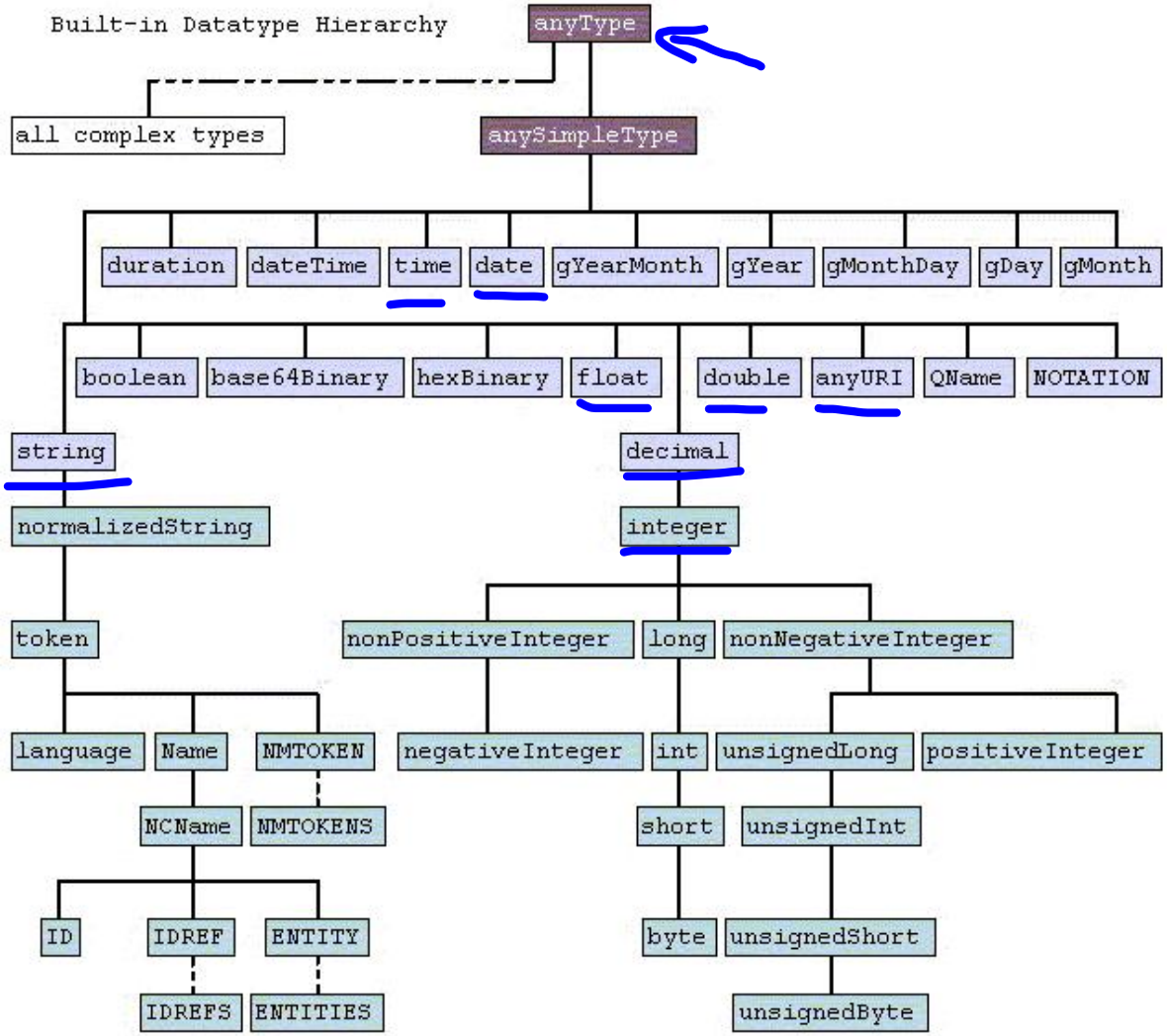
Example for the XML Schema for SOAP messages:
http://schemas.xmlsoap.org/soap/envelope/

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
 ...
</xs:schema>
```

Type Definitions

- Kind of types
  - Built-in datatypes
    - are used with attributes and elements
    - `http://www.w3.org/TR/xmlschema-2/#built-in-datatypes`
  - Simple types (<simpleType>)
    - are used with attributes and elements
    - cannot declare attributes or subelements
  - Complex types (<complexType>)
    - can be used with elements only
    - can declare attributes and subelements

Built-in types



Element definition with complex type

Element in an XML document

```
<person>
   <name>Kirsten Ahlborg</name>
   <address>
      ...
   </address>
</person>
```

Definition of that element in an XML Schema document

```
<element name="person" type="tns:PersonType"/>

<complexType name="PersonType">
   <sequence>
      <element name="name" type="xsd:string"/>
      <element name="address" type="tns:AddressType"/>
   </sequence>
</complexType>

<complexType name="AddressType">
   ...
</complexType>
```

Complex Types

- <complexType name="type name">
  - <sequence>
    - Subelement have to appear in the same order
  - <choice>
    - Only one subelement out of the mentioned elements may appear
  - <all>
    - Subelements may appear in arbitrary order
  - ...

minOccur, maxOccur

- Attributes on element definitions minOccurs and maxOccurs
  - → how often an element may occur in a type
- minOccurs
  - default = 1
  - 0, 1, . . .
- maxOccurs
  - default = 1
  - 0, 1, . . ., `unbound`

An array of persons

PersonArray

```
<complexType name="PersonArray">
  <sequence>
    <element name="person"
      maxOccurs="unbounded"
      minOccurs="0"
      type="PersonType"/>
  </sequence>
</complexType>
```

Example Instance

XML Schema Element declaration

```
<element name="persons" type="PersonArray"/>
```

XML

```
<persons>
   <person>...</person>
   <person>...</person>
   ..
</persons>
```

Attribute Definition

- Attribute definitions have a name and a type
- Attributes are defined
  - in (complex) type declarations for elements
  - can also be gloabal attributes applying to all elements in a schema definition

Attribute example

Example of an element with an attribute in XML

```
<person age="23">
   <name>Kirsten Ahlborg</name>
   <address>
      ...
   </address>
</person>
```

Element definition in XML Schema

inline def.

xsd:

xsd:

```
<element name="person">
  <complexType>
    <attribute name="age" type="integer"/>
    <sequence>
      <element name="name" type="string"/>
      <element name="address">
        <complexType>
           ...
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

Ways of defining elements

Using the **type** attribute

tns:

```
<element name="person" type="PersonType"/>
<complexType name="PersonType">
  ...
</complexType>
```

Anonymous type definitions (or **inlined** type definition)

```
<element name="person">
  <complexType>
    ...
  </complexType>
</element>
```

Using other Schema definitions: Import and Include

Use cases

1. **import** definitions from another XML Schema **keeping** the namespace of the definitions

```
<xsd:schema targetNamespace="urn:test">
 <xsd:import schemaLocation="schema_domain.xsd"
  namespace="urn.test.domain"/>
</xsd:schema>
```

   - Can be used to reuse other namespaces in the definitions

2. **include** definitions from another XML Schema **into** the namespace of the including namespace

```
<xsd:schema targetNamespace="urn:test">
 <xsd:include schemaLocation="schema.xsd"/>
</xsd:schema>
```

   - Can be used to distribute the definitions of one namespace in several files

Student Registration RPC: Example

- Operation registerStudent:
  - Provide student data (name, cpr, address)
  - Return the student number after registration
- Request

```
<S:Envelope
  xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:registerStudent
    xmlns:ns2="urn:ws.imm.dtu:studentregistrationrpc">
   <part1>
    <name>Henry</name>
    <cpr>123456</cpr>
    <address>
     <street>H.C. Andersen Boulevard</street>
     <city>Copenhagen</city>
    </address>
   </part1>
  </ns2:registerStudent>
 </S:Body>
</S:Envelope>
```

WSDL:

- PortType

```
<portType name="studentregistrationPortType">
 <operation name="registerStudent">
  <input name="input1" message="tns:registerStudentRequest
  <output name="output1" message="tns:registerStudentRespo
  <fault name="fault1" message="tns:registerStudentFault"/
 </operation>
</portType>
```

- Messages

```
<message name="registerStudentRequest">
 <part name="part1" type="tns:studentType"/>
</message>
<message name="registerStudentResponse">
 <part name="studentNumber" type="xsd:string"/>
</message>
```

- Question: how to define the studentType?
- Answer: Use of XML schema

Using the types section in the WSDL file

```
<types>
 <xsd:schema
   targetNamespace="urn:ws.imm.dtu:studentregistrationrpc"
   xmlns:tns="urn:ws.imm.dtu:studentregistrationrpc">
  <xsd:complexType name="studentType">
   <xsd:sequence>
    <xsd:element name="name" type="xsd:string"></xsd:element>
    <xsd:element name="cpr" type="xsd:string"></xsd:element>
    <xsd:element name="address" type="tns:addressType"></xsd:e
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="addressType">
   <xsd:sequence>
    <xsd:element name="street" type="xsd:string"></xsd:element
    <xsd:element name="city" type="xsd:string"></xsd:element>
   </xsd:sequence>
  </xsd:complexType>
    ...
 </xsd:schema>
</types>
```

Tips

- Use a structured editor to create the XML (WSDL and XML Schema)
- Use inline schema definitions in WSDL files for types/elements that are not reused (e.g. types/elements describing messages)
  - Use the WSDL schema editor
- Use external schema definitions if the datatypes are reused (e.g. domain datatypes)

Contents

Mapping Java Classes to XML Schema

- JAXB is a Java framework allowing the mapping
  - Java classes to XML schema definitions
  - Instances of Java classes to concrete XML
- JAXB has a standard way of mapping
  - This mapping can be customised using annotations so that it is possible to map classes to desired XML schemata
  - Bases of the mapping is the Java Beans convention
- Java Beans are Java classes with a standard interface which allows for easy introspection
- Any Java class can be a Java Bean class if it follows certain conventions
  - Default constructor
  - Properties should be accessed using `get...`, `set...`, and `is...` methods

Example Java Bean

no element generated
<company>
@XmlRootElement
generates
it

```
public class Company {

    protected String name;
    protected Address address;
    protected List<Employee> employees;

    public String getName() { return name; }

    public void setName(String value) { this.name = value; }

    public Address getAddress() { return address; }

    public void setAddress(Address value) { this.address = value; }

    public List<Employee> getEmployees() {
        if (employees == null) {
            employees = new ArrayList<Employee>();
        }
        return this.employees;
    }
}
```

XML Schema for the Company

```
<xs:schema xmlns:tns="http://week05.ws/" xmlns:xs="http://www.w3.org/2
 <xs:complexType name="company">
  <xs:sequence>
   <xs:element name="name" type="xsi:string" minOccurs="0"/>
   <xs:element name="address" type="tns:address" minOccurs="0"/>
   <xs:element name="employees" type="tns:employee" nillable="true"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="address">
  ...
 </xs:complexType>

 <xs:complexType name="employee">
   ...
</xs:schema>
```

- Classes are mapped to XML schema types
- Fields/properties (either public or defined by public getter and setter methods) are mapped to element definitions
  - It is also possible to map fields of simple datatypes to attributes with the help of annotations from `javax.xml.bind.annotation`
    - i.e. `javax.xml.bind.annotation.XmlAttribute`

Contents

SOAP Faults and WSDL

- Student Registration example: Send a SOAP fault when the student is already registered
  - → similar to checked exceptions in Java
  - user defined fault info:
    a) fault string
    b) XML elements for the fault: e.g. student and student number under which he/she is registered

Failure Response

```
<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>Student already registered</faultstring>
   <detail>
    <ns2:fault xmlns:ns2="http://studentregistration.ws">
     <student>
      <name>Henry</name>
      <cpr>123456</cpr>
      <address> ... </address>
     </student>
     <studentNumber>5678</studentNumber>
    </ns2:fault>
    <ns3:exception ...>
     <message>Student already registered</message>
     <ns3:stackTrace> ... </ns2:stackTrace>
    </ns3:exception>
   </detail>
  </S:Fault>
 </S:Body>
</S:Envelope>
```

stack trace

WSDL and Faults

- PortType

```
<portType name="studentregistrationPortType">
 <operation name="registerStudent">
  <input name="input1" message="tns:registerStudentRequest"/>
  <output name="output1" message="tns:registerStudentResponse"/>
  <fault name="fault1" message="tns:registerStudentFault"/>
 </operation>
</portType>
```

- Fault Message

```
<message name="registerStudentFault">
 <part name="part1" element="tns:fault"/>
</message>
```

only one part → has to be element not type

- Fault Type

```
<types>
 <xsd:schema>
    ...
  <xsd:element name="fault" type="tns:faultType"/>
  <xsd:complexType name="faultType">
   <xsd:sequence>
    <xsd:element name="student" type="tns:studentType"/>
    <xsd:element name="studentNumber" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
 <xsd:schema>
<types>
```

Service implementation

```
@WebService(...)
public String registerStudent(StudentType part1)
 throws RegisterStudentFault {
 ...
 FaultType faultInfo = new FaultType();
 faultInfo.setStudent(part1);
 faultInfo.setStudentNumber("1234");
 RegisterStudentFault fault =
  new RegisterStudentFault("Student already registered",
                           faultInfo);
 throw fault;
}
```

Contents

RPC vs Document style bindings

How to map WSDL messages to SOAP bodies?

- Port type

```
<portType name="servicePortType">
 <operation name="op">
  <input name="input1" message="tns:inMessage"/>
  <input name="output1" message="tns:outMessage"/>
 </operation>
</portType>
```

- Message

```
<message name="inMessage">
  <part name="p1" type/element ="...">
  <part name="p2" type/element ="...">
</message>
```

- SOAP body/header (application specific part)

```
<S:Body>
   ????
</S:Body>
```

RPC style bindings

- Port type

```
<portType name="servicePortType">
 <operation name="op">
  <input name="input1" message="tns:inMessage"/>
  <input name="output1" message="tns:outMessage"/>
 </operation>
</portType>
```

- Message

depends on binding

```
<message name="inMessage">
  <part name="p1" type ="xsd:string">
  <part name="p2" type ="tns:studentType">
</message>
<message name="outMessage">
  <part name="result" type ="xsd:string">
</message>
```

- SOAP body/header (application specific part)

```
<S:Body>
  <op xmlns="http://ws.dtu">
     <p1>...</p1>
     <p2>..</p2>
  </op>
</S:Body>
```

```
<S:Body>
   <opResponse xmlns="http://ws.dtu">
     <result>...</result>
   </op>
</S:Body>
```

RPC style binding

RPC style binding defines

- Namespace to be used in the SOAP body
- Whether to use the body or the header for the message
- Transport to be used, e.g., HTTP, and SOAPAction (optional)

WSDL

SOAP Binding

```
<binding name="servicePortTypeBinding"
         type="tns:servicePortType">
 <soap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="op">
  <soap:operation  soapAction="http://ws.dtu/op"/>
  <input name="input2">
   <soap:body use="literal"
    namespace="http://ws.dtu"/>
  </input>
  <output name="output2">
   <soap:body use="literal"
    namespace="http://ws.dtu"/>
  </output>
 </operation>
 ...
</binding>
```

RPC style binding

```
<S:Body>
   <op xmlns="http://ws.dtu/">
      <p1>...</p1>
      <p2>..</p2>
   </op>
</S:Body>
```

- RPC convention with SOAP messages predates WSDL
- Client needs to know WSDL conventions

→ He cannot just take a standard XML-Schema aware parser to parse the body

→ Alternative: Document style binding

Document style binding

- Port type

```
<portType name="servicePortType">
 <operation name="op">
  <input name="input1" message="tns:inMessage"/>
  <input name="output1" message="tns:outMessage"/>
 </operation>
</portType>
```

- Message

only one part

```
<message name="inMessage">
  <part name="p1" element ="tns:requestType">
</message>
```

- Type definition

/ element name="request" type="requestType"/>

```
<xsd:complexType name="requestType">
   <xsd:sequence>
      <xsd:element name="p1" type="xsd:string"/>
      <xsd:element name="p2" type="tns:studentType"/>
   </xsd:sequence>
</xsd:complexType>
```

- SOAP body/header (application specific part)

```
<S:Body>
  <request xmlns="http://ws.dtu/">
     <p1>...</p1>
     <p2>..</p2>
  </request>
</S:Body>
```

Document style: binding

Document style binding defines

- Whether to transport the message in the body or header
- Transport to be used, e.g., HTTP
- Defines SOAPAction HTTP header (optional)

```
<binding name="..." type="tns:servicePortType">
 <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http">
 </soap:binding>
 <operation name="registerStudent">
  <soap:operation soapAction="http://ws.dtu/op"/>
  <input name="i">
   <soap:body use="literal"></soap:body>
  </input>
   ...
</binding>
```

Document style binding

- Full control over the SOAP body using XML schema in WSDL
- No special of conventions are needed
  - → Only standard XML tools are need to parse the body
- A document style binding and message definition simulating RPC style SOAP bodies

→ called "Document Wrapped" style

Document style vs RPC style

- RPC style
  - Call operation $op(p_1, \ldots, p_n)$
  - Request and response have a predefined form
- Document style: shift in perspective
  - from calling operations to sending documents
  - defines the elements used for the request and the response

StudentRegistration Example: Document Style

- Operation registerStudent:
  - Request element: action (e.g. register), student data (simplified)
  - Response: student data where the student number field is set
- Request

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:request xmlns:ns2="urn:ws.imm.dtu:studentregistrationdocume
   <action>register</action>
   <student>
    <name>Henry</name>
   </student>
  </ns2:request>
 </S:Body>
</S:Envelope>
```

- Response

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:student xmlns:ns2="urn:ws.imm.dtu:studentregistrationdocume
   <name>Henry</name>
   <studyNumber>1234</studyNumber>
  </ns2:student>
 </S:Body>
</S:Envelope>0
```

WSDL

- PortType

```
<portType name="studentRegistrationDocumentPortType">
 <operation name="registerStudent">
  <input name="i" message="tns:registerStudentRequest">
  </input>
  <output name="o" message="tns:registerStudentResponse">
  </output>
 </operation>
</portType>
```

- Messages

```
<message name="registerStudentRequest">
 <part name="request" element="tns:request"></part>
</message>
<message name="registerStudentResponse">
 <part name="result" element="tns:student"></part>
</message>
```

Types section

```
<types>
 <xsd:schema
    targetNamespace="urn:ws.imm.dtu:studentregistrationdocument"
    xmlns:tns="urn:ws.imm.dtu:studentregistrationdocument">
  <xsd:element name="student" type="tns:studentType"></xsd:element>
  <xsd:element name="request" type="tns:requestType"></xsd:element>
  <xsd:complexType name="studentType">
   <xsd:sequence>
    <xsd:element name="name" type="xsd:string"></xsd:element>
    <xsd:element name="studyNumber" type="xsd:string"></xsd:element>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="requestType">
   <xsd:sequence>
    <xsd:element name="action" type="xsd:string"></xsd:element>
    <xsd:element name="student" type="tns:studentType"></xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:schema>
</types>
```

Binding section

```
<binding name="studentRegistrationBinding" type="tns:studentRe
 <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http">
 </soap:binding>
 <operation name="registerStudent">
  <soap:operation/>
  <input name="i">
   <soap:body use="literal"></soap:body>
  </input>
  <output name="o">
   <soap:body use="literal"></soap:body>
  </output>
 </operation>
</binding>
```

Technical differences between Document and RPC style

- RPC style binding
  - Each argument gets a part
  - Parts use the type attribute
- Document style binding
  - Only one part for the body and one for the header (optional)
  - Parts use the element attribute