Middleware and the Internet

**Middleware today**
- Designed for special purposes (e.g. DCOM) or with „overloaded“ specification (e.g. CORBA)
- Specifying own protocols – integration in real world network?
- Non-performant runtime behaviour
- Security questions: how to deal with firewalls?
- ...

**Internet today**
- Web designed for applications to be used by human beings
- Enabling B2B e-commerce and non-automated B2B interactions
- Build upon Internet protocols (HTTP, together with HTML)
- No assumptions made about platforms

**Problem today**
- How to enable automatic application-to-application interaction in the Web?
- E-Marketplaces, business process integration, resource sharing, …
Example: Shopping Service

Solution:
What we need is a Web site that provides a programmatic interface.
What could be possible?

*CORBA Component Model:*
- Building modular, distributed applications
- Ad-hoc creation of complex applications
- Reuse and integration of existing applications/components

*CORBA services, esp. Trading Service*
- Strong mechanism for dynamic binding of components

Service-oriented architecture
- Applications/components are seen as services, characterising their functionality
- Application construction be composing services
- Easy realisation of business processes
- Three roles: service provider, service requestor, service registry
Service Oriented Architecture

- Manage service descriptions and provide search facilities

Service Requestor  ➔ Service Registry ➔ Service Provider

- Find binding information in service descriptions to locate a service
- Bind to that service to invoke it

Service Description

- Publish services by advertising service descriptions in the registry
Web Services

What are Web Services?

- Web services is an effort to build a distributed computing platform for the Web
- Web service applications are encapsulated, loosely coupled Web “components” that can bind dynamically to each other
- SUN: “a Web service is a modular piece of code on the Internet that provides one or more business functions, and that can be discovered and used on demand.”

Goals and requirements:

- Enable universal interoperability
- Enable widespread adoption
- Enable dynamic binding (service oriented architecture)
- Support Web environment efficiently
- Base on open, extensible standards
- Assume minimal amount of required infrastructure
- Focus on messages and documents, not on APIs
Web Service Definition

A software component that can be

- **Described** using a service-description language, which
  - is in formal XML notation,
  - covers all the details necessary to interact with the service (message formats for operations, transport protocols and location), and
  - hides the implementation details of the service
- **Published** to a registry of services
- **Discovered** through a standard mechanism
- **Invoked** through a declared API, usually through a network
- **Composed** with other services
  - enabling loosely coupled, component-oriented, cross-technology application implementations.
Web Services Example

Often-cited examples of a Web Service:

- stock quote service, in which the request asks for the current price of a specified stock, and the response gives the stock price. This is one of the simplest forms of a Web service in that the request is filled almost immediately, with the request and response being parts of the same method call

- instead of including credit card transaction processing functions in an online retail application, an application can access a Web service that provides those functions
Web Services Framework

Publish, Find, Use Services:

*Universal Description, Discovery and Integration (UDDI)*

Formal Service Descriptions:

*Web Services Description Language (WSDL)*

Service Interactions:

*Simple Object Access Protocol (SOAP)*

Universal Data Format: XML

Ubiquitous Communications: TCP/IP, HTTP

What allows us to find these descriptions: Discovery of services.

What describes what goes on the wire: Description languages.

What goes “on the wire”: Formats and protocols.
SOAP, WSDL and UDDI

Discovery

Description

Interaction

HTTP

TCP/IP

- Automatically generate WSDL from server code
- Generate proxy, service location stored in WSDL
- Publish service description
- Client can locate the service querying UDDI
- Stubs can be generated from WSDL automatically
- Client invokes service (almost) like a local method
Web Services and Distributed Objects

Similarities:

- Both have some sort of description language
  - Operations, signatures, return types, exceptions
  - Compilers generate client proxy and server skeleton
  - Run-time middleware mediates the client-server interaction
- Both have well-defined network interactions
- Both have a similar mechanism for registering and discovering available components

Differences:

- Distributed objects enable stateful computing; the server lifecycle depends on the client-server interaction. Web services - at their basic incarnation - are designed for stateless computing.
- A client object can hold a reference to a server and access the server state through the server’s lifetime. There is no web-service reference mechanism.
- Distributed objects were designed mainly for within an intranet, and were conceived as decentralisation technology. Web services are intended as a technology in support of integration on the web.
Simple Object Access Protocol

SOAP...

- ... is an application layer protocol, carried within HTTP messages (also can be used with other application layer protocols, e.g. FTP, SMTP)
- ... defines a messaging framework for exchanging structured and typed information (formatted XML data) across the Internet
- ... is neutral with respect to operating systems, programming languages, and computing platforms
- ... adopts RPC to transfer complete XML documents
- ... also allows one-way transfer in a document-oriented approach
- ... bridges heterogeneous implementations
SOAP defines
- An XML envelope for XML messaging
- A HTTP binding for SOAP messaging
- A convention for doing RPC
- An XML serialisation format for structured data

HTTP headers
- Pass through web server
- Specifies global web service handler

SOAP envelope
- Names XML schemas in use, encoding style

SOAP header
- Metadata about the message
- Attributes for runtime services

SOAP body
- Data, method call, response, faults
The SOAP Envelope

```xml
<SOAP-ENV:Envelope
  xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    ...
  </SOAP-ENV:Header>

  <SOAP-ENV:Body>
    ...
  </SOAP-ENV:Body>

  ...
</SOAP-ENV:Envelope>
```

- **Additional information**: instructing run-time environment, target application, coordinator, …
- **Real content**: Method calls, parameters, responses, error codes, …
SOAP Example Message

**HTTP headers**
- POST /ocs/contentService HTTP/1.1
- Host: www.wns.com
- 
- SOAPAction: "http://wns.com/#searchRequest"

**SOAP envelope**
- xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/
- xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
- xmlns:xsd="http://www.w3.org/1999/XMLSchema"
- SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

**SOAP header**
- <t:Transaction xmlns:t="http://wns.com/acl"
- SOAP:mustUnderstand="1">3</t:access>

**SOAP body**
- <fd:search xmlns:fd="http://wns.com/search"
- <article-id xsi:type="xsd:string">1245</article-id>
- </fd:search>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Parts of SOAP Message

POST /ocs/contentService HTTP/1.1
Host: www.wns.com

... SOAPAction: "http://wns.com/#searchRequest"
<SOAP-ENV:Envelope
  xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
/>
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="http://wns.com/acl"
      SOAP:mustUnderstand="1">3</t:access>
  </SOAP-ENV:Header>
</SOAP-ENV:Body>
  <fd:search xmlns:fd="http://wns.com/search">
    <article-id xsi:type="xsd:string">1245</article-id>
  </fd:search>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

- Specifying the SOAPAction, i.e. the target application
- Naming the XML schemas used for coding
- Defining the encoding style for data transfer
Parts of SOAP Message

Information “describing” the SOAP request:

- Request belongs to a transaction
- Transaction is assigned the TID 3
- mustUnderstand is set to 1— instructing the server to either process the request as a transaction or fail to process it

```xml
POST /ocs/contentService HTTP/1.1
Host: www.wns.com
SOAPAction: "http://wns.com/#searchRequest"

<SOAP-ENV:Envelope
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
><SOAP-ENV:Header>
  <t:Transaction xmlns:t="http://wns.com/acl"
    SOAP:mustUnderstand="1">3</t:Transaction>
</SOAP-ENV:Header>

<SOAP-ENV:Body>
  <fd:search xmlns:fd="http://wns.com/search"
    >1245</article-id>
</fd:search>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
Parts of SOAP Message

POST /ocs/contentService HTTP/1.1
Host: www.wns.com

Simply a method call search(article-id)

<SOAP-ENV:Envelope
  xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <t:Transaction xmlns:t="http://wns.com/acl"
      SOAP:mustUnderstand="1">3</t:access>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <fd:search xmlns:fd="http://wns.com/search">
      <article-id xsi:type="xsd:string">1245</article-id>
    </fd:search>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Typical RPC:
String strYear = objVehicle.getVehicleYear("739ADF984");

XML-RPC
Request:
<getVehicleYear>
  <VIN>739ADF984</VIN>
</getVehicleYear>

Response:
<getVehicleYearResponse>1996</getVehicleYearResponse>

XML-RPC:
- Encode and bind data structures into XML
- Encode an RPC call
SOAP defines a serialisation for formatting data (e.g. parameters in a request) to XML structures:

```java
class VehicleDescription {
    String brand = "Ford";
    int doors = 3;
}
```

```xml
<VehicleDescription>
    <brand type="xsd:string">Ford</brand>
    <doors type="xsd:int">3</doors>
</VehicleDescription>
```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <!-- Header elements go here, and are optional. -->
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <!-- Message or method call elements go here.-->
        <getVehicleYear>
            <VIN>739ADF984</VIN>
        </getVehicleYear>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
     <!-- Response from web service. -->
     <getVehicleYearResponse>
       1996
     </getVehicleYearResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Error Response

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
  <SOAP-ENV:Fault>
   <faultcode>SOAP-ENV:Server</faultcode>
   <faultstring>Server Error</faultstring>
   <detail>
    <e:myfaultdetails xmlns:e="http://www.ints.com/cars">
     <message>
      Server busy. Please try again later.
     </message>
     <errorcode>1001</errorcode>
    </e:myfaultdetails>
   </detail>
  </SOAP-ENV:Fault>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Response contains result formatted in XML, or an error description
There are three components that take part in a SOAP application:

- **Client application**: A program/Servlet/... that sends a SOAP request. Wants to use a service.
- **SOAP processor**: A program that can receive SOAP requests and act accordingly (e.g., call an method of the Application Server)
- **Application Server**: A program that supplies the Web service
What do we have to Program?

- We won't directly read or write SOAP messages
- Instead, use Java methods that create request and analyse result
- Use a SOAP processor that is actually a Servlet
- Code the client application and the application server
- Your application server does not need anything special
- In fact, your application server does not have to "know" that it is being used as a Web Service
Creating a SOAP Envelope

```java
import javax.xml.soap.*;
import javax.xml.messaging.*;

//Create Soap Message
MessageFactory msgFactory = MessageFactory.newInstance();
SOAPMessage soapMsg = msgFactory.createMessage();

//Create Soap Envelope elements
SOAPPart soapPart = soapMsg.getSOAPPart();
SOAPEnvelope soapEnv = soapPart.getEnvelope();
SOAPHeader soapHeader = soapEnv.getHeader();
SOAPBody soapBody = soapEnv.getBody();
soapHeader.detachNode(); //SOAP Header is optional
```
Building the SOAP Message

```java
//Create the request element Name
Name nameRequest = soapEnv.createName("getVehicleYear",
    "m",
    "http://www.ints.com/cars");

//Add a request element to the SoapBody
SOAPBodyElement soapRequestElement =
    soapBody.addBodyElement(nameRequest);

//Set the request element's value
soapRequestElement.addTextNode("739ADF984");
```
Sending a SOAP Message

//Create Soap Connection
SOAPConnectionFactory scFactory =
    SOAPConnectionFactory.newInstance();
SOAPConnection sc = scFactory.createConnection();

//Create URL Endpoint and send Soap Message
URLEndpoint endPoint = new
    URLEndpoint("http://localhost/MyApp/SoapServlet");
SOAPMessage response = sc.call(soapMsg, endPoint);
sc.close();
Web Services Description Language

WSDL describes, how and where to access a service, i.e. the service interface, similar to remote object approaches like CORBA:

• What can the service do?
  - What operations are provided?
  - Abstract access port for message operations (input, output)
• What data types are used?
  - Definition of abstract messages (Data types, formats, …)
• How to invoke the service?
  - Transport protocol used (usually SOAP over HTTP)
  - Network address of a service

• Used for the description: XML
• Stubs for RPC are generated from a WSDL document
WSDL Elements

A WSDL document consists of:

- **Documentation**: A human readable description (optional)
- **Types**: Data type definitions used in operations
- **Message**: Logical content of message; basing on the defined types, a definitions of the data being communicated
- **Operation**: An abstract definition of an action supported by the service
- **PortType**: Abstract set of operations, using messages as input/output
- **Binding**: Binding to concrete protocol and data format (SOAP, HTTP, …)
- **Service**: A single endpoint for communication (binding and network address)
- **Port**: A collection of related endpoints
WSDL Elements

**Service Implementation:**
- Equivalence in CORBA: IIOP and IOR
- “Contact details” about the service implementation

**Service Interface:**
- Like in CORBA IDL: defining an interface
- Generation of stubs from this definition

Different from IDL: an “object reference” is included in the specification
WSDL Example (Interface)

```xml
<?xml version="1.0"?>
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:xsd1="http://example.com/stockquote.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="TradePriceRequest">
            <complexType>
                <all>
                    <element name="tickerSymbol" type="string"/>
                </all>
            </complexType>
        </element>
        <element name="TradePrice">
            <complexType>
                <all>
                    <element name="price" type="float"/>
                </all>
            </complexType>
        </element>
    </schema>
</types>
```
WSDL Example (Interface)

```xml
<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
```
WSDL Example (Implementation)

```xml
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
</definitions>
```
Universal Description, Discovery and Integration

- UDDI = Registry for advertise and discover Web Services
- Contains categorised information about businesses and the services they offer
- Association of services with a technical specification (defined in WSDL)
- Defines operations for registering services (Business, technical specifications, service and service endpoints) and for service discovery
- UDDI itself is realised as a Web Service

Three types of UDDI registries:

1. Public directory, a collection of UDDI servers everyone can use without charge (see http://www.uddi.org)
2. Protected directories, shared by groups of companies with shared interests
3. Private directories, for company-internal usage or to share with cooperation partners
UDDI Usage

1. Create WSDL describing service
2. Submit description to UDDI registry
3. Query registry for WSDL of a service
4. Return matching service info
5. Use WSDL to generate correct messages
6. Interact with service
What is stored in the UDDI Registry?

Different categories of information:

1. White pages
2. Yellow pages
3. Green Pages

- Business name
- General business description
- Contact info (name, phone number, …)
- List of unique identifiers (e.g. domain name)
- Business categories (Industry, product, geographic location)
- Description of “how to do e-commerce” with a provider
- Model for business process, technical service specifications, binding information
Relation to WSDL Specifications

WSDL:

- `<service>`
  - `<port>`
  - `<port>`
  - `<types>`
    - `<message>`
    - `<portType>`
    - `<binding>`

Service Interface

UDDI:

- BusinessEntity
- BusinessService
- BindingTemplate
- BindingTemplate
- tModel

Service Implementation
UDDI Information

• Business Entity
  – A business entity contains information about a business including its name, a short description, and some basic contact information.
  – Each business can also be associated with unique business identifiers and with a list of categorizations that describe the business.
  – UDDI provides built-in support for a number of taxonomies.
    • Examples include SIC (Standard Industrial Classification codes), NAICS (North American Industry Classification System), UNSPSC (Universal Standard Products and Services Codes), and a geopolitical taxonomy.
  – Businesses and industry groups can create additional taxonomies to categorize their businesses and services.
• **Binding Templates**
  – Associated with each business service entry is a list of binding templates that provide information on where to find the service and how to use the service.
  – A binding template may contain the access point of the service implementation and a pointer to information on how to use the service.
  – The binding template also associates the business service with a service type.
UDDI Information

- **Service Types**
  - A service type, defined by a construct called a tModel, defines an abstract service.
  - Multiple businesses can offer the same type of service, all supporting the same service interface.

- **tModel**
  - A tModel specifies information such as the tModel name, a list of categories that describe the tModel, and pointers to technical specifications for the tModel.
  - A tModel may point to a WSDL document that describes the abstract service type.
UDDI Example

```
<businessDetail generic="2.0" operator="www.ibm.com/services/uddi" truncated="false">
  <businessEntity businessKey="413E39E0-0807-11D8-B704-000629DC0A53" operator="www.ibm.com/services/uddi">
    <discoveryURLs>
      <discoveryURL useType="businessEntity">
      </discoveryURL>
    </discoveryURLs>
    <name xml:lang="en">Stock Company</name>
    <businessServices>
      <businessService serviceKey="B7E326A0-0807-11D8-B704-000629DC0A53" businessKey="413E39E0-0807-11D8-B704-000629DC0A53">
        <name xml:lang="en">StockService</name>
        <description xml:lang="en">StockService IBM testing</description>
        <bindingTemplates>
          <bindingTemplate bindingKey="B7F28FF0-0807-11D8-B704-000629DC0A53" serviceKey="B7E326A0-0807-11D8-B704-000629DC0A53">
            <description xml:lang="en"/>
            <accessPoint URLType="http">
              http://www.example.com/StockService/StockService
            </accessPoint>
            <tModelInstanceDetails>
              <tModelInstanceInfo tModelKey="UUID:B55ACE10-0807-11D8-B704-000629DC0A53"/>
            </tModelInstanceDetails>
          </bindingTemplate>
        </bindingTemplates>
        <categoryBag>
          <keyedReference tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2" keyValue="52392" keyName="Portfolio Management">
          </keyedReference>
        </categoryBag>
      </businessService>
    </businessServices>
  </businessEntity>
</businessDetail>
```
UDDI Example

```
< userModelDetail generic="2.0" operator="www.ibm.com/services/uddi" truncated="false">
  < userModel id="UUID:B55ACE10-0807-11D8-B704-000629DC0A53"
    operator="www.ibm.com/services/uddi">
    < name>
      StockService Specification
    </ name>
    < description xml:lang="en">
      T-model for service interface definition
    </ description>
    < overviewDoc>
      < overviewURL>
        http://www.example.com/StockService/StockService.wsdl
      </ overviewURL>
    </ overviewDoc>
    < categoryBag>
      < keyedReference id="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
        keyName="uddi-org.types" keyValue="wsdlSpec"/>
    </ categoryBag>
  </ userModel>
</ userModelDetail>
```
Implementing Web Services

- Java has an API for XML-based remote procedure calls (JAX-RPC) which simplifies the process of building Web services that incorporate XML-based RPC.
- It defines mappings between Java types and XML types that attempt to hide the details of XML and provide a familiar method-call paradigm.
- JAX-RPC can be used to implement and call SOAP-based Web services described by WSDL using Apache's open source tools:
  - *Apache Tomcat* for deployment
  - *Apache Axis* for SOAP implementation
JAX-RPC Canonical Architecture
Apache Axis Architecture

Diagram showing the architecture of Apache Axis, including Java VM, J2EE server, Web container, Axis router, EJB container, SOAP messages, and client Java objects.
How does it work?

- On the client side (Application 1)
  - The Java support packages (e.g., org.apache.axis.client)
    - generate the appropriate SOAP request - according to the server that the client is accessing,
    - send it to the Axis engine as an HTTP request,
    - receive the resulting SOAP response,
    - process it, and
    - return the de-serialized return java object to the original calling client method.

- On the server side (Application 2)
  - The Axis engine provides all of the support to
    - process the SOAP request,
    - call the indicated method in the deployed service class,
    - receive its return java object,
    - package it in a SOAP response, and
    - return it via HTTP to the client.
First Step: Service Interface and Implementation

```java
package iis;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloIF extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

- Web Service Interface is derived from class `Remote`
- Methods required to throw `RemoteException`

```java
package iis;
public class HelloImpl implements HelloIF {
    public String message = "Hello";
    public String sayHello(String s) throws RemoteException{
        return message + s;
    }
}
```

Note: no Web Service specific code here!

Compile the classes:

```
javac HelloIF.java HelloImpl.java
```
 Configuration File

- All relevant information on Web Service

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service
        name="HelloWorldService"
        targetNamespace="http://lsirwww.epfl.ch/"
        typeNamespace="http://lsirwww.epfl.ch/"
        packageName="iis">
        <interface name="cis.HelloIF"/>
    </service>
</configuration>
```

- Automatically derive WSDL from Interface and Configuration File

```
wscompile -define -mapping build/mapping.xml
    -d build -nd build -classpath build config.xml
```
Structure of WSDL

<?xml version="1.0">
<definitions name="HelloWorldService" … Name Space Information …>
<types>
  <schema>
    definition of parameter data types in XML Schema (optional)
  </schema>
</types>
<message name="HelloIF_sayHello">
  definition of a message (request, reply)
</message>
<portType name="HelloIF">
  <operation name="sayHello">
    definition of an operation (request - reply pair)
  </operation>
</portType>
<binding name="HelloIfBinding" type="HelloIF">
  definition of a protocol binding (typically SOAP)
</binding>
<service name="HelloWorldService">
  <port name="StockQuotePort">
    definition of a port (an Internet address)
  </port>
</service>
</definitions>
WSDL Content

Provides message names and passing of parameters:

```xml
<message name="HelloIF_sayHello">
  <part name="String_1" type="xsd:string"/>
</message>
:message name="HelloIF_sayHelloResponse">
  <part name="result" type="xsd:string"/>
</message>
```

Define message sequences corresponding to a service invocation

```xml
<portType name="HelloIF">
  <operation name="sayHello" parameterOrder="String_1">
    <input message="tns:HelloIF_sayHello"/>
    <output message="tns:HelloIF_sayHelloResponse"/>
  </operation>
</portType>
```
Protocol Binding

Implement abstract messages according to SOAP protocol

```xml
(binding name="HelloIFBinding" type="tns:HelloIF">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  <operation name="sayHello">
    <soap:operation soapAction=""/>
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
        use="encoded" namespace="http://lsirwww.epfl.ch/"></input>
    <output>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
        use="encoded" namespace="http://lsirwww.epfl.ch/"></output>
  </operation>
</binding>
```
Service Access

Location not known before deployment

```xml
<service name="HelloWorldService">
    <port name="HelloIFPort"
        binding="HelloIFBinding">
        <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
</service>
```

After deployment: this can be published via UDDI

```xml
<service name="HelloWorldService">
    <port name="HelloIFPort" binding="tns:HelloIFBinding">
        <soap:address location="http://lsir-cis-pcx:8009/hello/helloService"/>
    </port>
</service>
```
Deploy Service

- Deploying: tell the SOAP processor about your application
  - Create a deployment descriptor
  - Call the java command that deploys the web application

```
<isd:service
  xmlns:isd="http://xml.apache.org/xml-soap/deployment"
  id="URI of the application">
  <isd:provider type="java"
    scope="application"
    methods="sayHello">
    <isd:java class="iis.HelloImpl"/>
  </isd:provider>
  <isd:faultListener>
    org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>
```

- Fill in your URI here
- Means that all SOAP requests will be sent to the same object
- List of available methods
- Responsible for error handling
- Name of Java class implementing the service in form packageName.className

**Deploying:**
```
java org.apache.soap.server.ServiceManagerClient
http://<host>:<port>/soap/servlet/rpcrouter deploy HelloDescriptor.xml
```
Register with UDDI

**UDDI Business Test Registry**
Universal Description, Discovery, and Integration

**Add a Business**

The only field required to enter a valid business within the UDDI Registry is the name of the business. Once you have entered a valid business name press the **Continue** button and you will then be able to enter additional details about your business. If you do not wish to add a business at this time press the **Cancel** button.

**Enter the name of your business**

- **Language**: English
- **Name**

**Buttons**: Cancel, Continue
Generate Stubs

- Client Configuration File

```xml
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <wsdl
        location="build/HelloWorld.wsdl"
        packageName="iis"/>
</configuration>
```

- Automatically created using WSDL and client configuration file

```
wscompile -gen:client -d build -classpath build config-client.xml
```
package iis;
import javax.xml.rpc.Stub;
public class HelloClient {
    private String endpointAddress;
    public static void main(String[] args) {
        try {
            Stub stub = createProxy();
            stub._setProperty
                (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
            HelloIF hello = (HelloIF)stub;
            System.out.println(hello.sayHello(args[1]));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
Creating Proxy (Stub)

```java
private static Stub createProxy()
{
    return (Stub)
    (new HelloWorldService_Impl().getHelloIFPort());
}
```

attaching _Impl to the service name is an implementation-specific naming convention
compiling, packaging and invoking

```
compile
javac -classpath system_jars:server_class_files:
    stub_class_files HelloClient.java

package
jar cvf hello-client.jar
    all_client_class_files:all_server_class_files
```

Invoke the client:

```
java -classpath hello-client.jar:jwsdp-jars hello.HelloClient
```
Using Web Services is not as Simple as it Looks

- It is not practical to automatically find web services for your needs
- There is no built-in mechanism for payment for use of a web service
- There is no built-in security control
- When a web service changes (e.g., adds a parameter to its method), the program using it breaks
- How to deal with quality demands to a Web Service?