



# **Introduction**

Michelle Davis

Sr. Solutions Architect

# Overview

---

Through this Tutorial, one will learn:

- Introduction to web services
  - Core web services technologies (XML, XSD, WSDL and SOAP) overview
  - How to use these core technologies .
- Overview of Celtix, the open-source SOA infrastructure
  - Celtix installation and environment
  - How to program web services clients and servers using JAX-WS and Celtix
  - Celtix Flexibility (multiple payloads, transport and communications model support
  - Deployment with Celtix
- IONA Jumpstart building blocks

# Audience

---

- The material is directed at those responsible for:
  - Analyzing IT problems; and,
  - Implementing web services solutions.
  
- As such, the course is suitable for:
  - Enterprise Architects;
  - Solution Architects; and
  - Software Developers



# Core Web Services Technologies

# Introduction

---

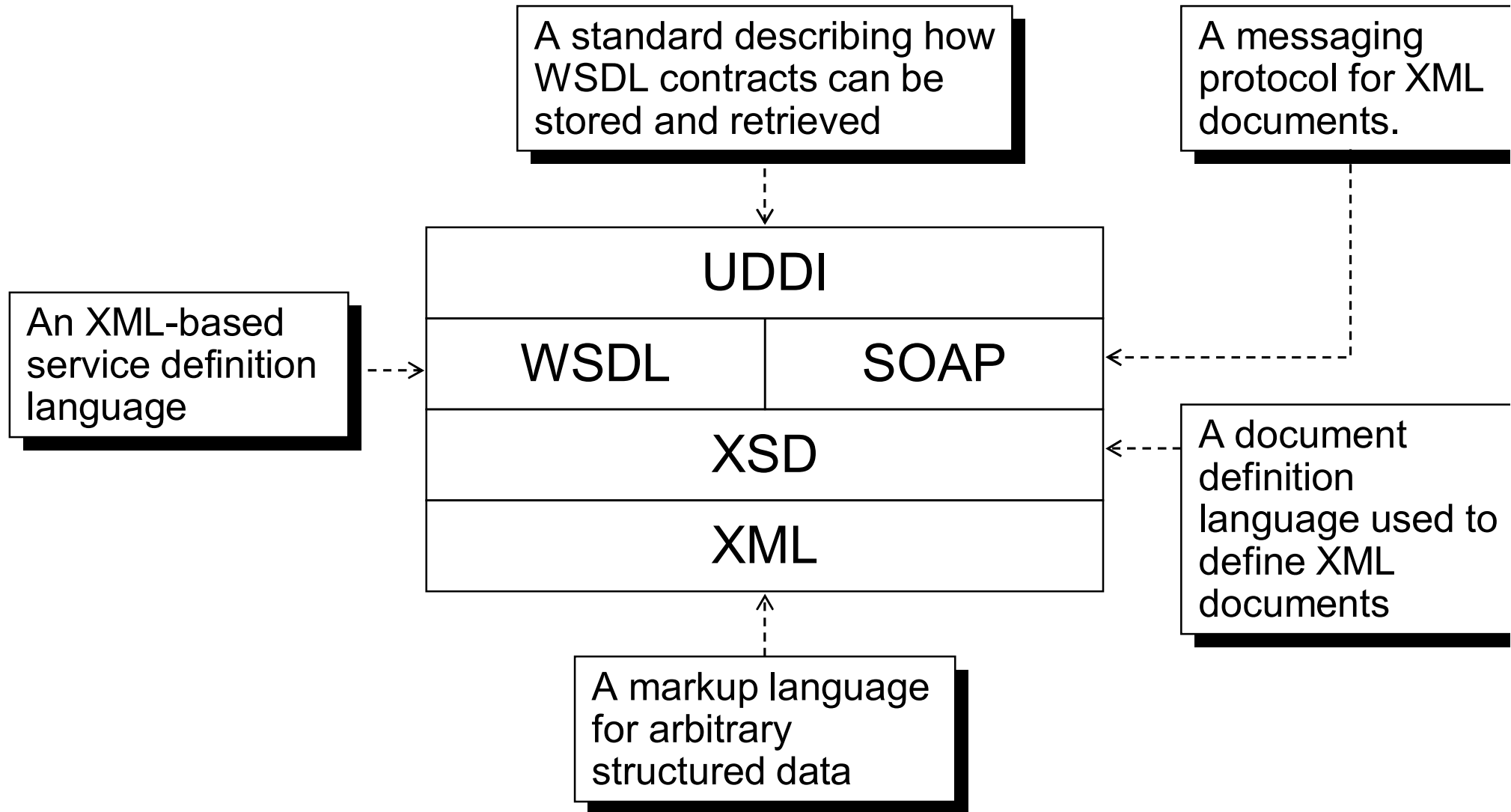
- XML, XSD, WSDL, SOAP and UDDI are the core technologies behind web services.
- These technologies are used for integration and interfacing:
  - They are *not* application development technologies
  - Developers still develop in existing or new software development environments
- These technologies make it easier to tie together existing or planned software components (services).
  - This is due to the language-, platform-, OS-, and hardware-neutral characteristics of the standards
- Web services technologies provide one way to implement the interfaces and messages for a service-oriented architecture

# Introduction (cont')

---

- This session shows where and how XML, XSD, WSDL, SOAP and UDDI fit in the web services technology stack.
  - At the end of this session, you will have the “big picture” of web services.
- A high-level overview will be given for each technology in turn.

# Web services technology stack



# Terminology

---

- XML (eXtensible Mark-up Language)
  - A plain-text notation for describing complex data.
- XSD (XML Schema Definition)
  - Defines the format of XML documents.
- WSDL (Web Services Description Language)
  - Defines the logical and physical interface of the service, that is, what the service does and how you can access it.
- SOAP (Simple Object Access Protocol)
  - A protocol for sending and receiving XML messages in both synchronous and asynchronous fashion
  - Defines how to format XML documents for transmission
- UDDI (Universal Description, Discovery and Integration)
  - One way to advertise and discover services; not universally adopted



# XML (eXtensible Markup Language)

---

- XML is a derivative of SGML in the family of markup languages; it is a full and evolving W3C standard.
- It is similar to HTML in appearance, but has a very different goal:
  - HTML focuses on the presentation and “style” of data.
  - XML focuses on the structure and content of the data.
- It is a parsable, extensible and self-describing text format for storing and exchanging information
  - XML is platform-, hardware, and programming-language neutral.
  - XML is highly portable across heterogeneous networks.
- XML is the technology on which WSDL and SOAP are based.
- Reference: <http://www.w3.org/XML>

# An example XML document

---

- An example XML document is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Address>
  <Street>123 Main Street</Street>
  <City>AnyCity</City>
  <State>AnyState</State>
  <ZIP>12345</ZIP>
</Address>
```

- Key features to note are:

- XML structures data as *elements*, using tags, like `<Address>` and `<Street>` to delimit the element's data.
- Elements can be nested as required to describe complex data.
- The information is stored and transmitted in plain-text in a readable form.

# XSD (XML Schema Definition)

---

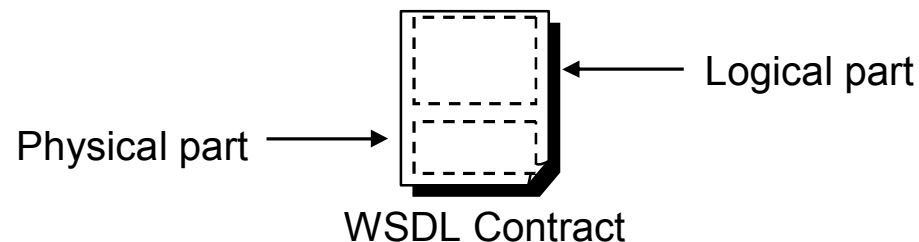
- XSD is used to define the contents of an XML document.
- For example, the XML document on the previous slide is defined by the following schema.

```
<schema...>
  <complexType name="Address">
    <sequence>
      <element name="Street" type="string"/>
      <element name="City" type="string"/>
      <element name="State" type="string"/>
      <element name="ZIP" type="string"/>
    </sequence>
  </complexType>
</schema>
```

# WSDL

---

- WSDL is simply XML that describes a service: it gives the *logical* description of the interface and its *physical* location.
  - *Logical*: what a service does; its data-types, messages, operations and interfaces.
  - *Physical*: how a service is accessed; its protocol-bindings and service-endpoints.
- WSDL acts as the “service contract” between a service provider and consumer
- WSDL contracts are typically quite verbose and difficult to read.
  - The next slides shows a *pseudo-code* version of a WSDL contract.



# logical vs. physical contracts

```
interface StockQuoteService
{
    float getStockQuote(String symbol)
        throws StockQuoteServiceFault;
    String getStockDescription(String symbol)
        throws StockQuoteServiceFault;
    String[] getStockSymbols()
        throws StockQuoteServiceFault;
}

fault StockQuoteServiceFault {
    string message;
    string errorCode;
}
```

Protocol binding: **SOAP**

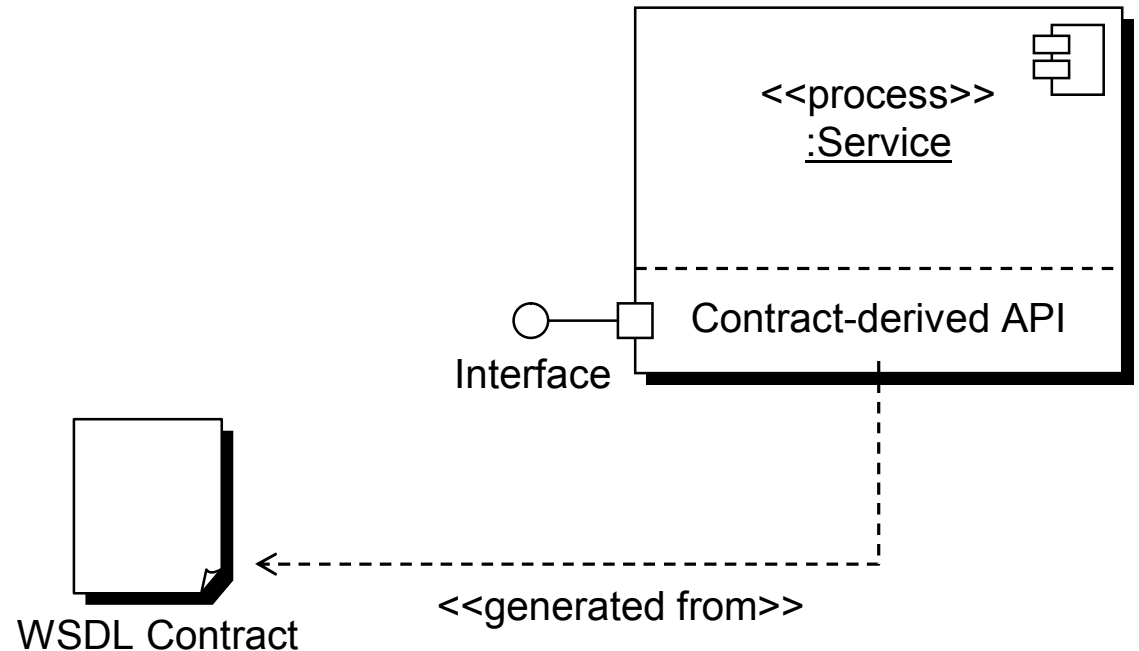
Endpoint: **http://frodo:8080/StockQuotes**

Logical: data-types, messages, operations and interfaces.

Physical: protocol bindings and service endpoints.

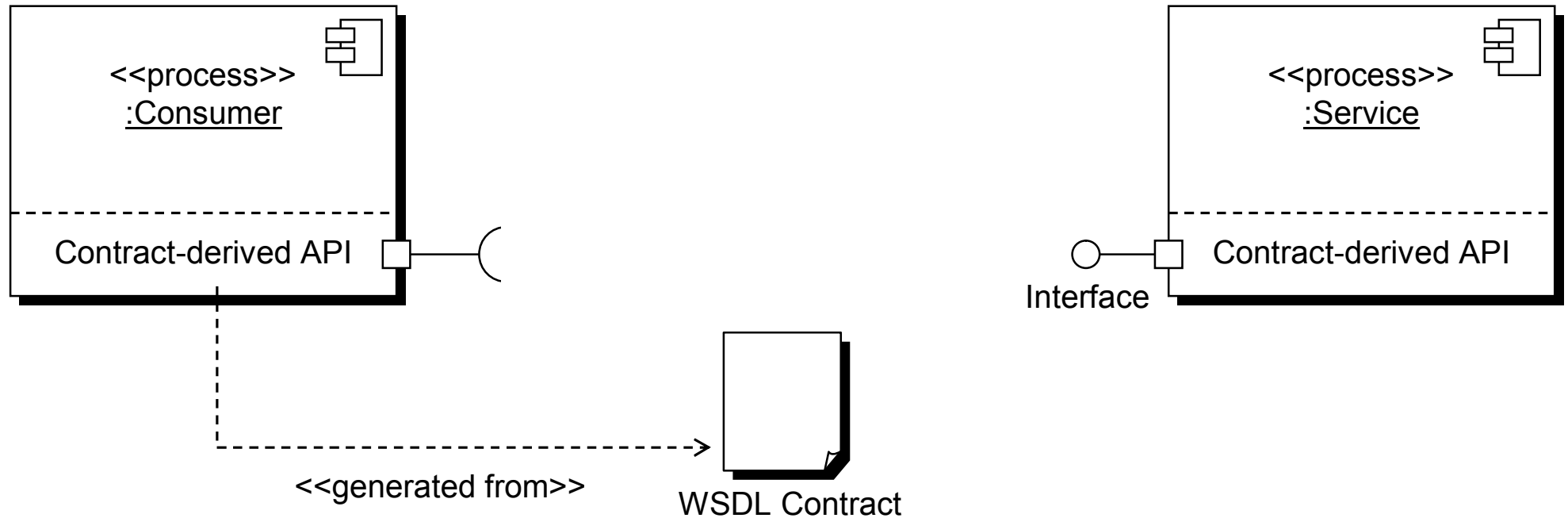
# WSDL-derived APIs

- A language specific application programmers interface (API) is *derived* from the logical contract.
  - The API is derived automatically using a *code generation tool*.
  - WSDL code generators exist for many languages:
    - Java, .Net (C#, J#, Visual Basic), C++, ...
- Using this API, a developer can implement a service.



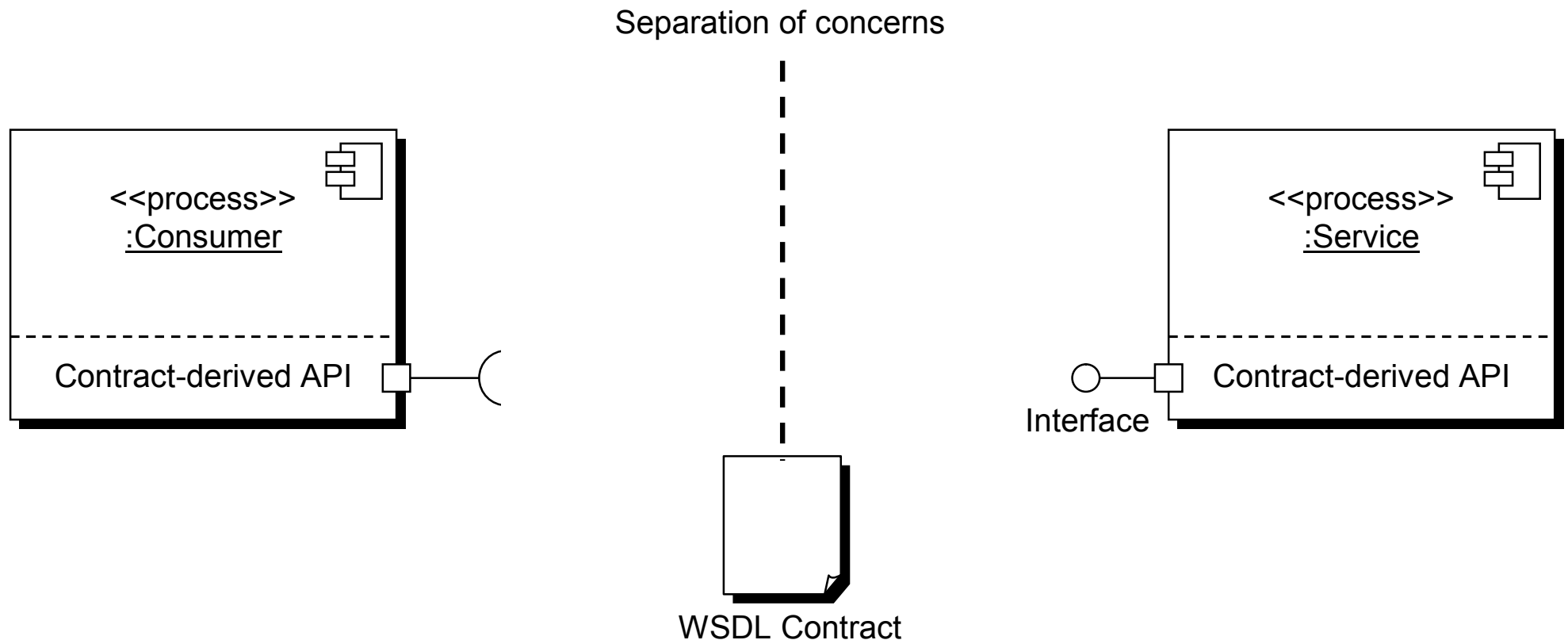
# WSDL-derived APIs (cont')

- In a similar fashion, a client-side API is derived from the contract.
- Using this API, a developer can implement a client.



# WSDL benefits: decoupling

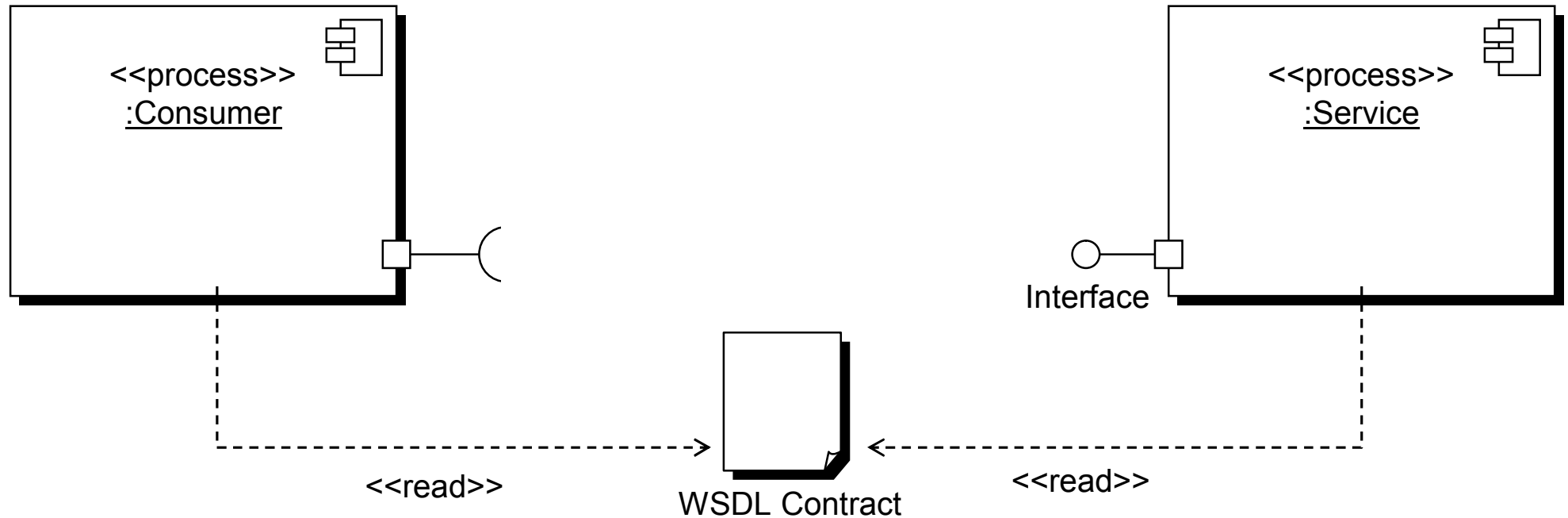
- Note that the consumer does not need to know *anything* about how the service is implemented.
  - All of the information required to use the service is contained in the contract.
  - The separation of interface from implementation is at the heart of SOA.





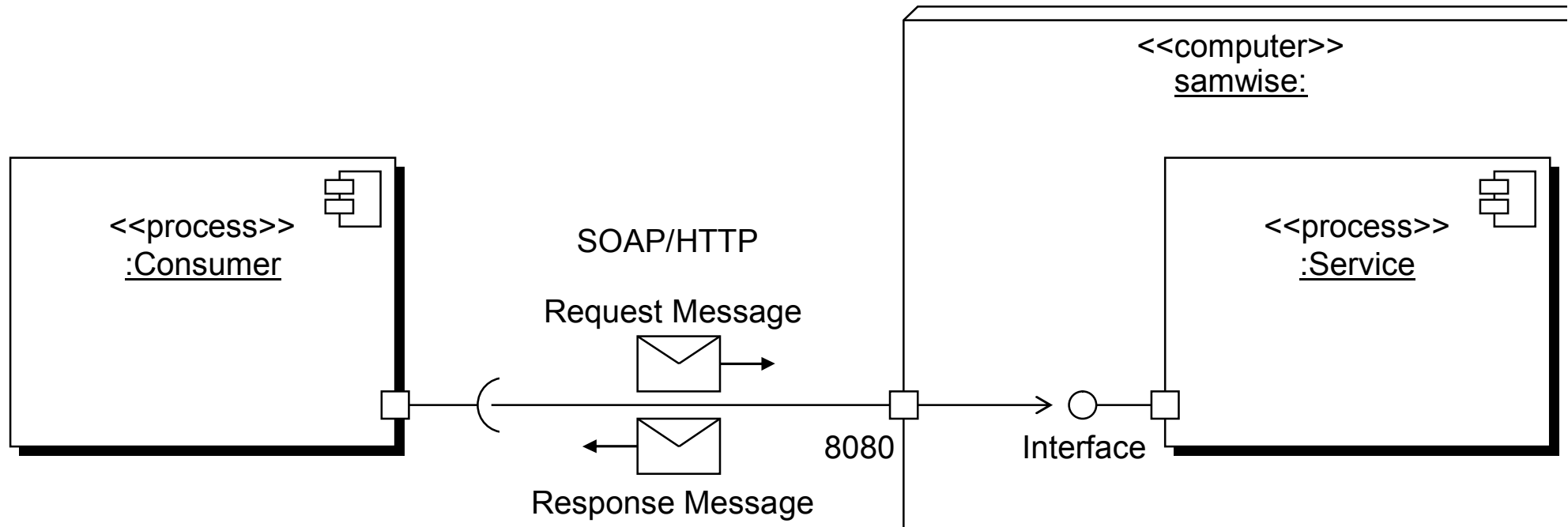
# WSDL: deployment

- At runtime, the service reads endpoint information from the physical part of the contract.
  - For example, “listen for SOAP/HTTP requests on port 8080”.
- The client also reads endpoint information from the physical part of the contract.
  - For example: “send requests to port 8080 using SOAP/HTTP”.



# WSDL: deployment (cont')

- The client and server can now communicate.
  - In the diagram below, the service is deployed on a computer called “samwise”
  - The service listens on port 8080 for SOAP/HTTP traffic.



# SOAP (Simple Object Access Protocol)

---

- SOAP is an XML-based communication protocol.
- SOAP was originally intended to provide a simple protocol to access distributed objects.
  - SOAP is no-longer simple or object-oriented.
  - “Complex Service Access Protocol” might be a more appropriate name.
- When a SOAP client or server sends a message, it *wraps* the XML content in an outer XML document.
  - A SOAP message is simply an XML document that provides a messaging *wrapper* around the XML payload.
- The message has an `Envelope` element that identifies the message boundary and includes:
  - an optional `Header` element (containing meta-data, system-level data or other auxiliary information); and
  - a `Body` element, containing the XML payload.

# Structure of a SOAP message

---

- A sample SOAP wrapper for the XML document we showed earlier is given below.

```
<?xml version="1.0"?>
<soap:Envelope>
  <soap:Header>
    . . . . .
  </soap:Header>
  <soap:Body>
    <Address>
      <Street>123 Main Street</Street>
      <City>AnyCity</City>
      <State>AnyState</State>
      <ZIP>12345</ZIP>
    </Address>
  </soap:Body>
</soap:Envelope>
```

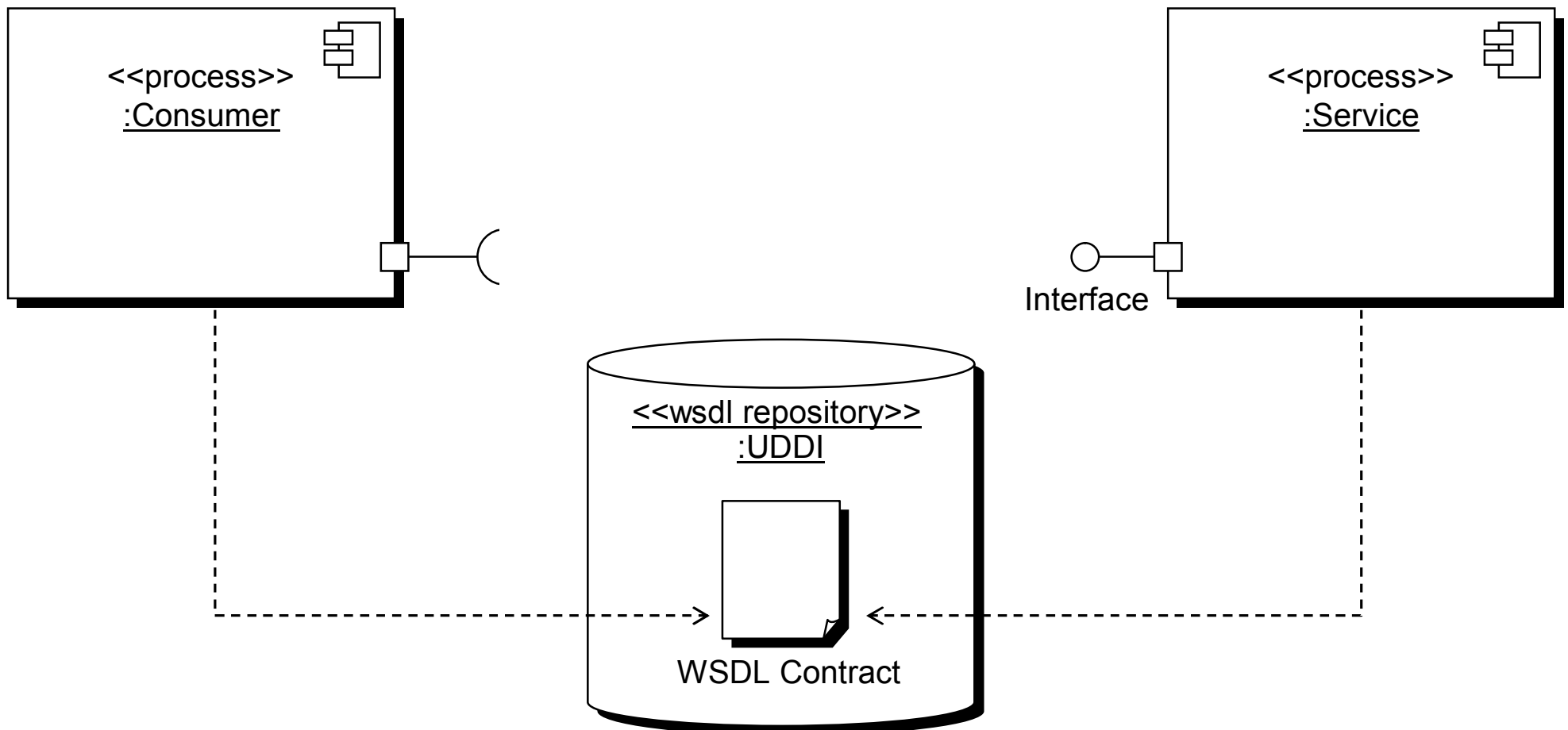
# UDDI

---

- UDDI (Universal Description, Discovery and Integration) is a standard for publishing and searching for WSDL contracts.
- UDDI is like a “yellow pages” for web services, supporting the ability to register and find services on the internet.
  - Service providers publish their services’ WSDL contracts, along with searchable attributes.
  - Potential clients search UDDI registries to retrieve WSDL suiting their service needs.
- Note: UDDI is the *least* accepted of the SOAP, WSDL and UDDI trio of technologies.
  - The ebXML standard from OASIS may replace UDDI.
- Resource: <http://www.uddi.org>

# UDDI

- UDDI repositories provide a storage and lookup facility for WSDL contracts.



# Summary

---

- Web Services are underpinned by a set of XML technologies: WSDL, SOAP and UDDI.
- WSDL is used to define the messages a server will receive and transmit, and provide the network address of the server
- SOAP is used to provide a standardised wrapper around XML message information.
- UDDI is used to distribute and share WSDL contracts.



# Web Services Introductory Demo



# Introductory demo

---

- In this session, the instructor will guide you through a live web-services demo.
  - The demo web service runs on the instructor's computer;
  - If time permits (and a network is available) then the instructor may access live web-services available on the internet.
  
- In this demo, you will see:
  - The web service's "contract", in both pseudo- and raw-WSDL form;
  - How a client (written in Java) can access the web service;
  - How the client and server communicates through the use of plain-text, human-readable XML messages;
  - How a dynamic browser-based client (SOAPUI) can access the web service.

# Starting the web service

---

- The stock-quote web service has been implemented using the Java programming language and Celtix, an open-source ESB.
  - The web service part of the Celtix Exercise System.
- A number of pre-requisites are required:
  - Java (JDK 1.5.07 or higher)
  - Celtix (1.0 or higher)
  - Ant (1.6.2 or higher)
  - SOAPUI (1.6 or higher)
- From a suitably configured shell, the instructor will start the web service using `stock-quote-service`
- The server starts, and listens for HTTP traffic. You should see a message :

```
Listening for requests...
```

# Viewing the web service contract

---

- The instructor will import the WSDL contract into the SOAPUI viewer.
  - Demo contracts can be found in the `wSDL` directory of the main stock-quote demo directory.
- The viewer allows you to view the interface in an abbreviated “pseudo” view, listing the operations available.
  - Using SOAP UI, you can create sample request messages, send them to the service, and view the response.
  - You can also perform load-testing and analysis of the service.

# A Java client

---

- The client is run with the script `stock-quote-client`
- To demonstrate the web service's programmatic interface, the instructor will modify the Java class file `src/stockquote/Client.java`.
  - The variable `stockQuoter` is a "stub" that connects to the server.
  - The methods `getStockDescription()`, `getStockQuote()` and `getStockSymbols()` can be used.
- When the client is run, you will see that the server writes to the console on every method invocation.

# Summary

---

- Web services technologies are XML-oriented:
  - Services are defined using WSDL; and
  - Messages are typically sent using the SOAP protocol over HTTP.
- WSDL allows you to separate interface from implementation.
  - Web services (and clients) can be implemented and deployed using popular languages such as Java, C++ and C#.
  - Web services can be implemented on many different operating systems and hardware architectures.



# Celtix Project Overview

# Overview

---

## ■ This session introduces Celtix

- Celtix Enterprise: an open source Java Enterprise Service Bus (ESB)
  - Built around Apache CXF, supported by IONA Technologies
    - CXF Project website: <http://cwiki.apache.org/CXF/>
    - IONA: <http://www.iona.com/celtix>
- The Celtix communication stack
  - The benefits of supporting multiple payloads and transports
- Celtix Use Cases
- Celtix Communication Models
  - Celtix supports a variety of communication models, including one-way, request-response, asynchronous messaging and publish-subscribe.
- Celtix Deployment
  - Celtix can be deployed as standalone client/server, within a J2EE application server, servlet engine, spring container or Java Business Integration (JBI) container.

# What is Celtix?



# What is Celtix?

---

- Celtix is an open-source Java Enterprise Service Bus (ESB)
- Celtix combines a number of open-source SOA-infrastructure projects into one integrated, tested and certified package.
  - CXF – Core ESB functionality
  - ActiveMQ – JMS Broker
  - Mule – Routing
  - Spring – Service container
  - Tomcat – Servlet container
  - ServiceMix – JBI container
  - Qpid – AMQP Broker
- Use Celtix for:
  - Designing, developing and deploying Web Services in Java and scripting languages such as E4XML.
  - Applications involving transmission, routing and transformation of XML messages.

# History

---

- Celtix was originally hosted by ObjectWeb, an open-source foundation that focuses on middleware technology.
- Celtix moved to Apache in 2006, was merged with the XFire project, and was renamed CeltiXFire
  - The name was subsequently changed to CXF
  - CXF project website: <http://cwiki.apache.org/CXF/>
  - IONA Technologies is a major contributor to both CXF and the original Celtix project.
- IONA uses the name “Celtix Advanced Service Engine” for its certified distribution of CXF.
  - See: <http://www.ionas.com/products/celtix/>
  - Celtix Enterprise includes the advanced service engine, advanced messaging (AMQP).

# Building service-oriented applications with Celtix

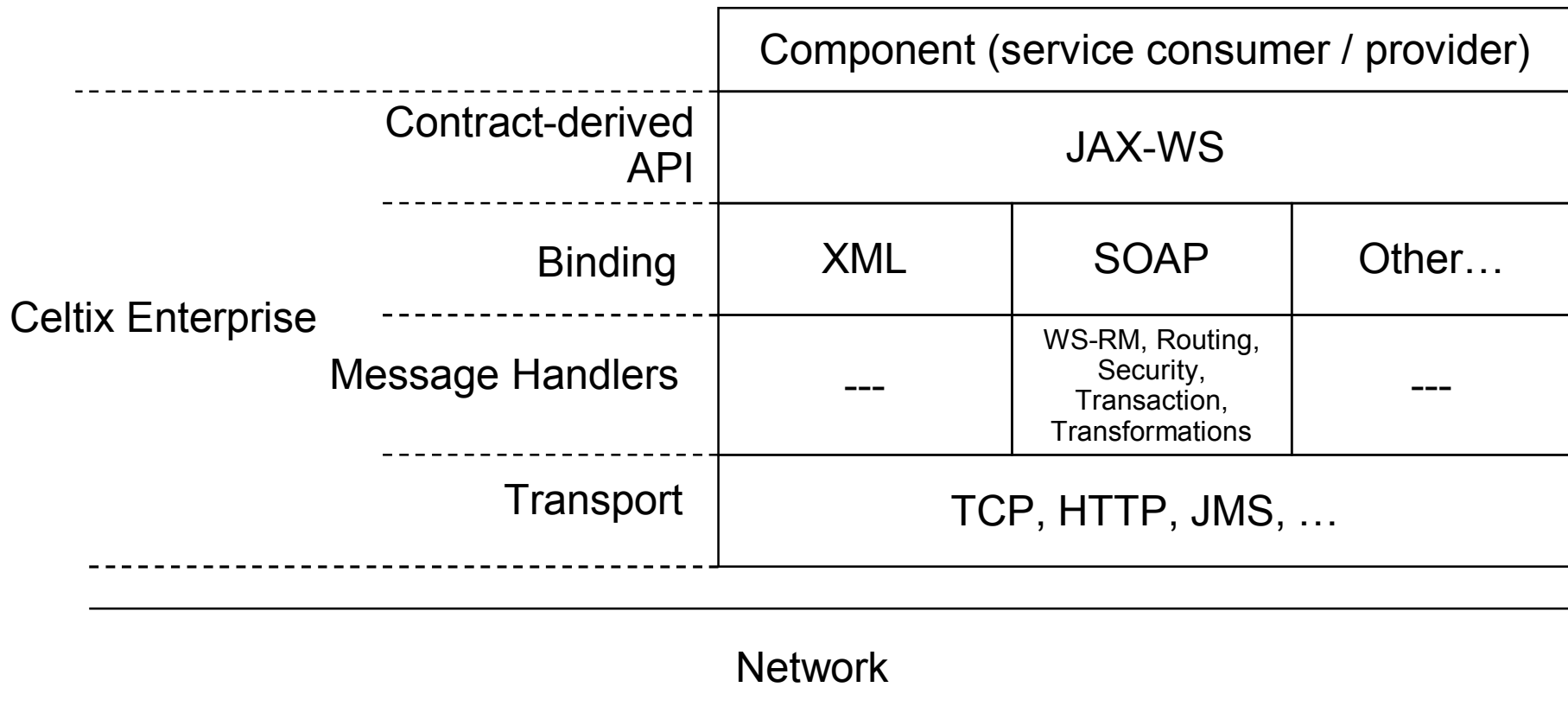
---

- Celtix abstracts away the details of inter-process communication between services and consumers.
- The service's *contract* is defined using WSDL (Web Services Description Language)
  - This contract is mapped to a semantically equivalent Java API in a standardized way
- Using WSDL as an interface language offers a number of benefits:
  - Programming language neutral: uses XML schema for type definition
  - Separates the logical interface (what a server does) from the physical interface (how to communicate with the server)
  - Services can support multiple communication protocols
  - Separates the service implementation from the service description
  - Services can be implemented in any language, OS or hardware.
  - Widely supported by the industry

# The Celtix Communication Stack

# The Celtix communication stack

- Celtix provides a layered communication stack to services and consumers.



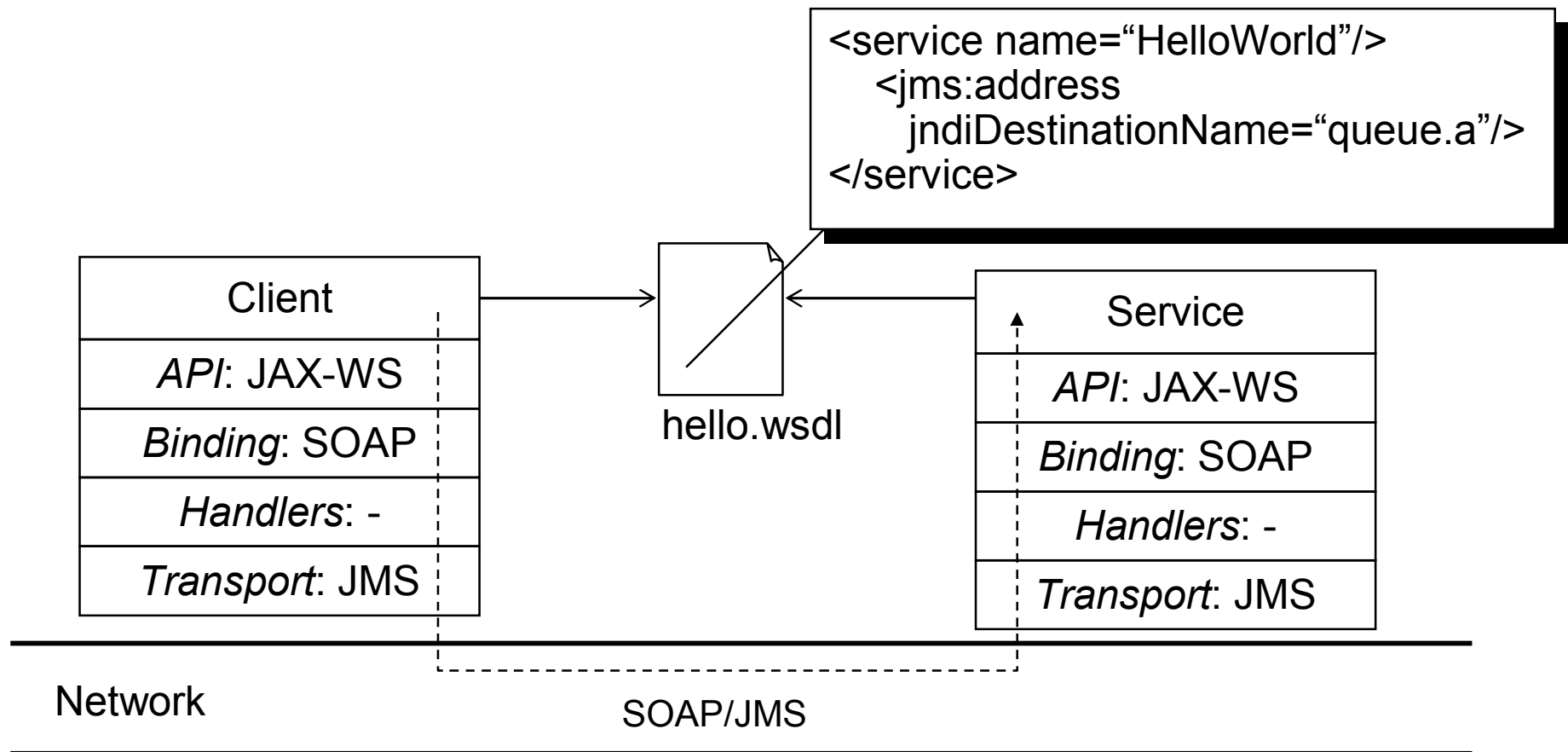
# The Celtix communication stack (cont')

---

- When a component sends a message to a service, it uses a Java API derived from the WSDL contract
  - The JAX-WS mapping is used.
  - The Java API is *protocol neutral*.
- A Celtix *binding* creates the message *payload*
  - Pure XML, SOAP, JSON, or some other message format.
- Quality-of-service extensions can be added using *message handlers*
  - Celtix supports a number of Web Services QoS standards.
- Celtix then transmits the message using a *transport*.
  - Celtix includes HTTP, JMS, AMQP, out-of-the-box.
  - You can write custom transports for Celtix; for example: TCP, SMTP, FTP.

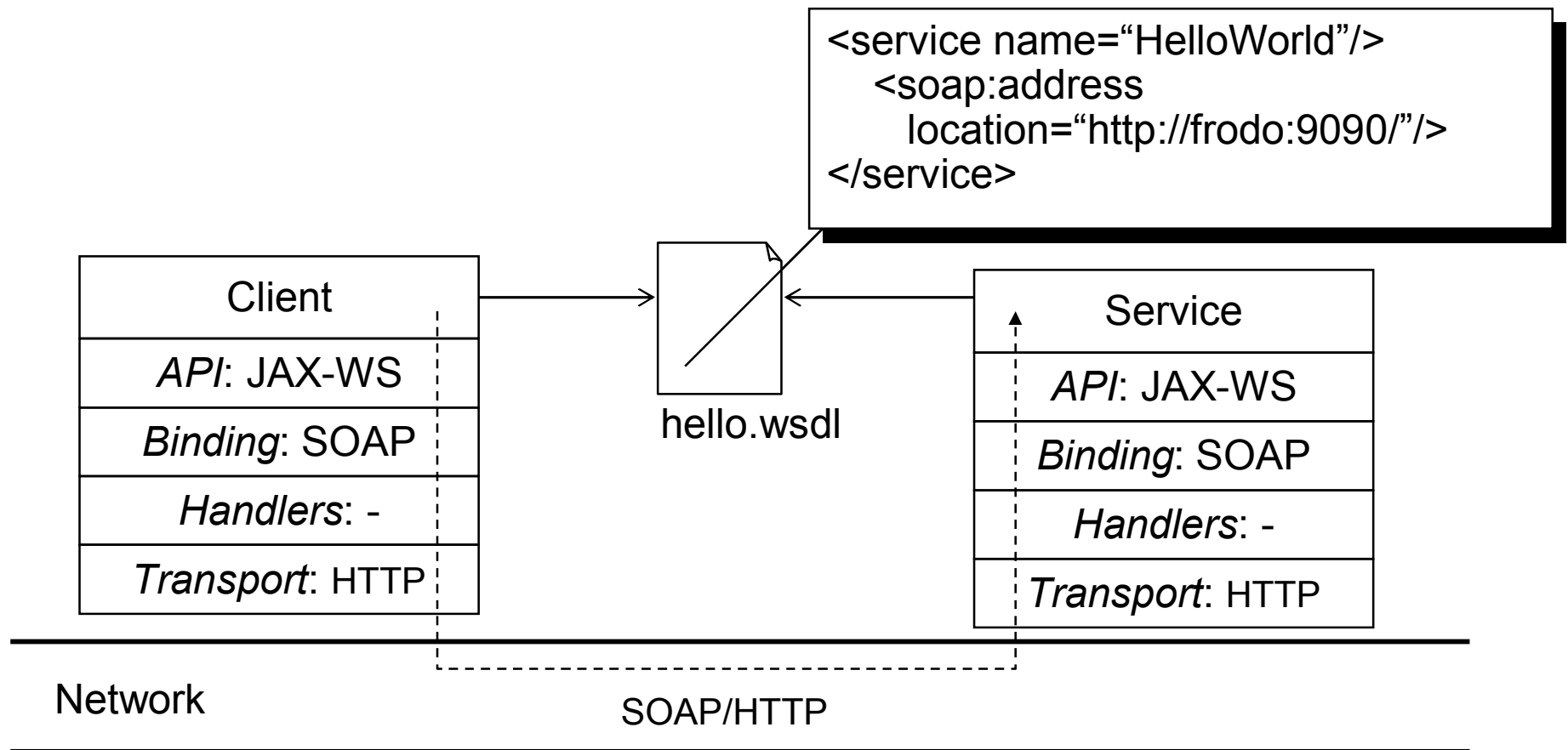
# The Celtix communication stack (cont')

- The binding and transport used by Celtix is determined by information in the WSDL service contract.



# The Celtix communication stack (cont')

- Change the service information, and Celtix will use a different set of plugins.
  - Client and server code is unaffected, as it is payload and protocol neutral.

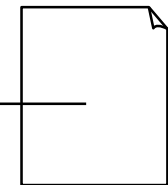




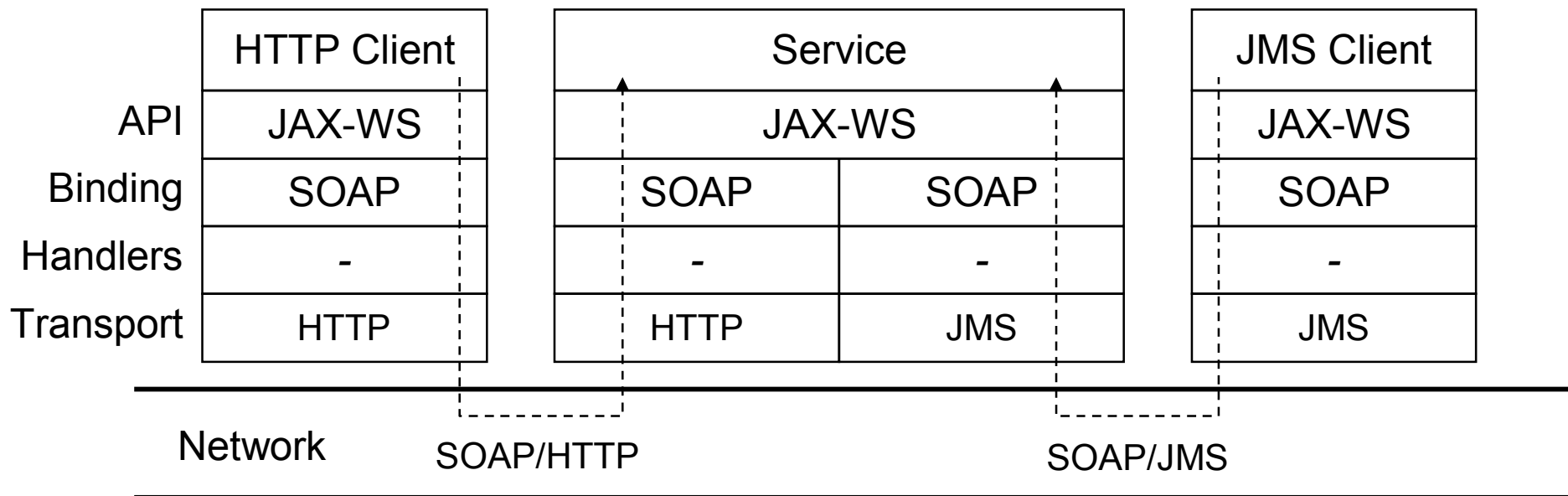
# The Celtix communication stack (cont')

- The same service can support multiple transports at the same time.

```
<service name="HelloWorld"/>  
  <soap:address location="http://frodo:9090"/>  
  <jms:address jndiDestinationName="queue.a"/>  
</service>
```



hello.wSDL



# Communication Models

---

- Celtix can be used for a number of different message-exchange-paradigms (MEPs):
  - *One-way*: raise event, submit document
  - *Request-response*: request information, request service
  - *Messaging*: send/receive document (with store-and-forward capability)
  - *Publishing*: publish-subscribe
- The ability to support all these styles, using a choice of standardized payload formats and transports, is a key strength of Celtix.

# Flexibility of Celtix

# Celtix is extendible

---

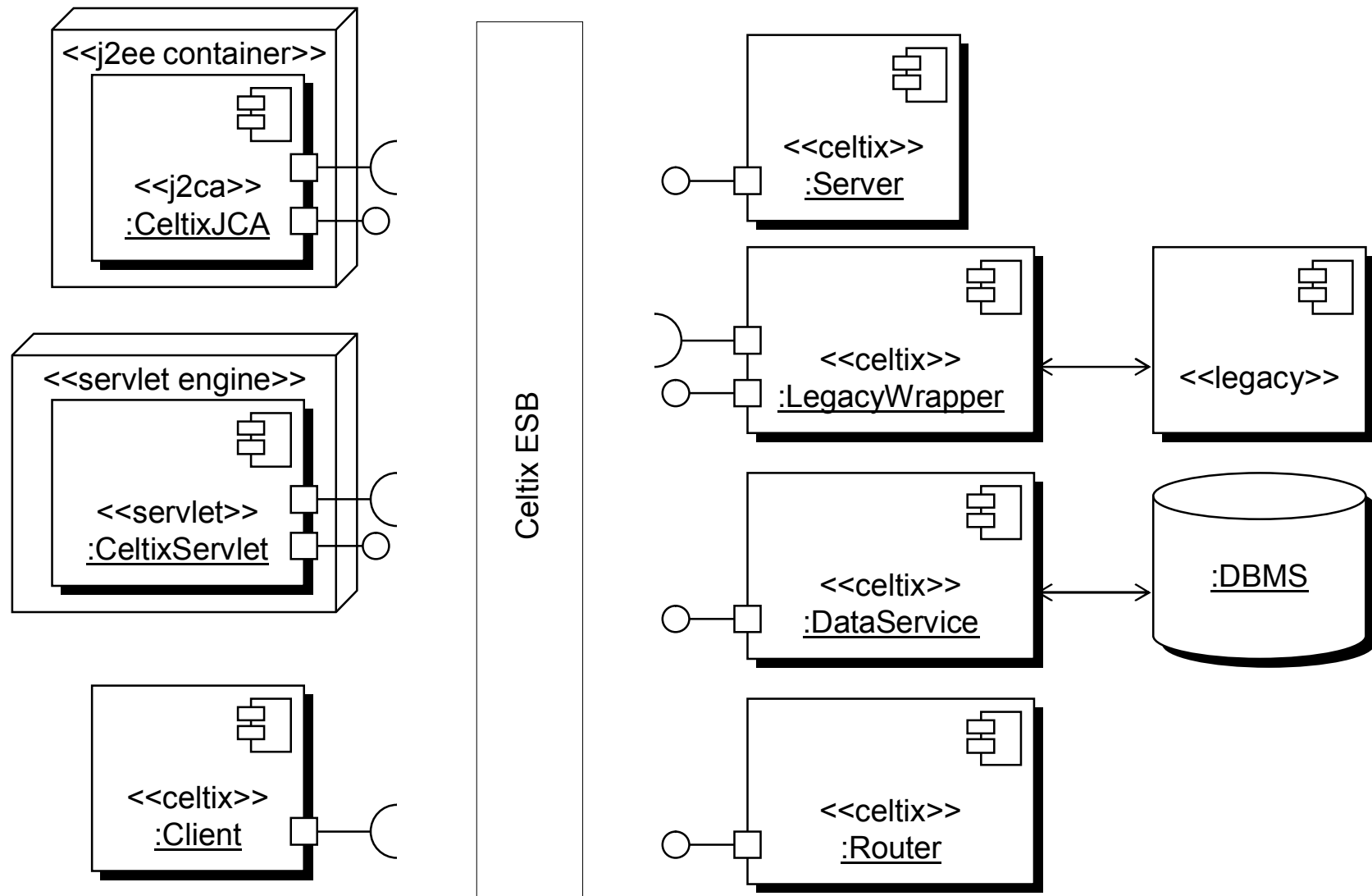
- Celtix is flexible: you can extend it to write your own bindings, handlers and transports.
- Bindings:
  - Add your own payload format.
- Handlers:
  - Perform logging or snooping; add message headers
- Transports:
  - Extend Celtix to access new transports, for example MQ, CORBA, ...
- Remember: Celtix is build from open source components
  - You can contribute your extensions to the Celtix code-base.

# Celtix use cases

---

- Celtix allows you to provide a multi-protocol, service-oriented interface for your IT systems
  - Build new services using Celtix
  - Wrap existing legacy systems in a service-oriented fashion using Celtix
  - Provide protocol-agnostic data services
  - Use Celtix to integrate J2EE applications with SOA services as a JCA connector
  - Develop clients (service consumers) using Celtix

# Celtix use cases

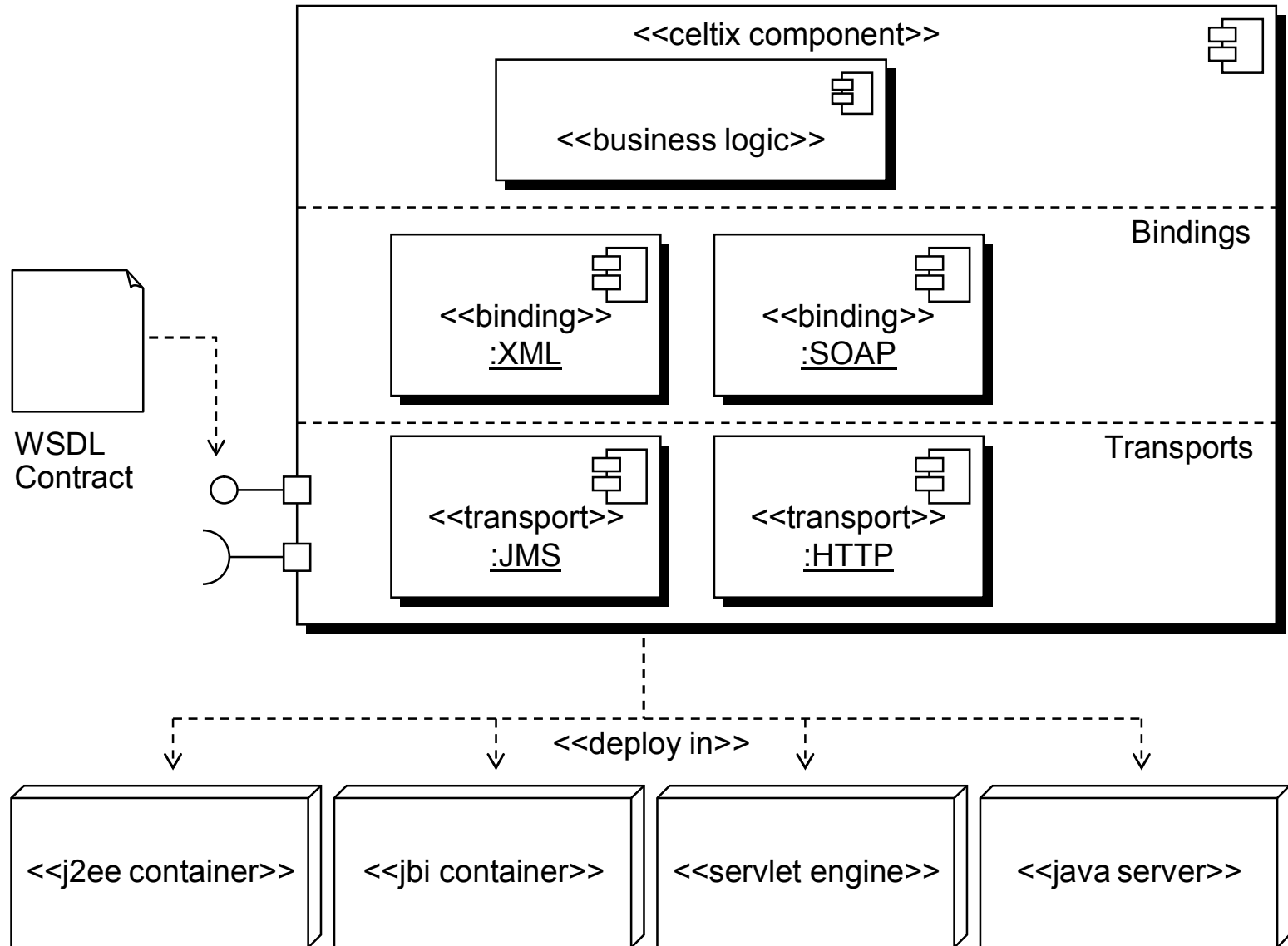


# Celtix deployment models

---

- Celtix can be used to write stand-alone Java clients and servers.
- Celtix can also be deployed in a number of different container technologies:
  - In a servlet engine (e.g. Tomcat)
  - In a J2EE container (using bi-directional JCA) (e.g. JBOSS)
  - In a JBI (Java Business Integration) container (e.g. ServiceMix)
  - In a lightweight Spring container
- This flexible deployment approach allows you to deploy your Celtix service-oriented components anywhere.

# Celtix deployment (cont')





# Celtix licensing

---

- Celtix is available under the *Celtix Enterprise Version 1.1 License*
  - This is based on the Apache License Version 2.0.
  - See <http://www.iona.com/forms/celtix/1.0/license.htm>
- For a good discussion on open source licenses, see
  - <http://www.gnu.org/philosophy/license-list.html>

# Summary

---

- This session has introduced Celtix Enterprise
  - Celtix Enterprise: an open source ESB
    - A suite of open-source products, built around the CXF core.
  - The Celtix communication stack
    - Supports multiple payloads (SOAP/XML/JSON) and transports (HTTP, JMS, AMQP).
  - Celtix Use Cases
  - Celtix Flexibility
    - Celtix supports a variety of communication models, including one-way, request-response, messaging and publishing.
  - Celtix Deployment
    - Celtix can be deployed as standalone client/server, within a J2EE application server, servlet engine, spring container or JBI container



# Celtix Installation and Environment

# Overview

---

- This chapter explains how to:
  - Obtain and install Celtix
  - Set up development & run-time environments
- Celtix consists of:
  - Runtime JAR files providing core web services functionality
  - Development tools used to generate Java code from WSDL
- To develop or run Celtix-based applications, you must set:
  - Java class path (`CLASSPATH` environment variable)
  - System executable path (`PATH` environment variable)
  - Assorted other environment variables
- Celtix can be used in any Java IDE; Eclipse is recommended.
  - Can also be used *without* an IDE.

# Downloading and Installing Java 1.5

# Downloading and Installing JDK 1.5

---

- Celtix uses Java features new to J2SE 5.0
  - Celtix will *not* work with versions of the JDK less than 1.5
  - This restriction is due to the JAX-WS 2.0 mapping, which mandates the use of features from J2SE 5.0.
- Download and install JDK 1.5.0\_09 or higher
  - Available from <http://java.sun.com>
- After installation, ensure that you are picking up the correct JDK:
  - Ensure `PATH` contains JDK `bin` directory *before* older JDK entries
  - Set `JAVA_HOME` to point to the installation directory of the JDK
  - Check you have the correct version of the JDK by running `java -version` in a command window or UNIX shell

# Downloading and Installing Celtix

# Downloading Celtix

---

- This course was developed for Celtix Enterprise 1.0
  - The material is not appropriate for earlier or later releases of Celtix.
- You can download Celtix from  
<http://www.ionapro.com/celtix/>
- You can download Celtix in *source* or *binary* form:
  - Source: if you wish to browse the code and learn how Celtix works.
  - Binary: if you want to get started building your own applications on Celtix
- Source and binary distributions are available Java Archive format (`.jar`), suitable for use on all platforms.
- You can download Celtix documentation from:  
<http://www.ionapro.com/celtix/>



# Installing a binary distribution of Celtix

---

- Celtix is packaged as a zip file.
  - Extract the zip file to a temporary directory
- Execute the installer using `install.bat` (or `install.sh` on Unix/Linux)
  - A GUI will guide you through the installation process.
  - Note: Celtix does *not* install system DLLs, system services, or registry settings on Windows-based systems.
- The GUI allows you to select an installation type.
  - For this course, choose “Celtix Enterprise Full Install”

# Installing a binary distribution of Celtix (cont')

---

- It's a good idea to embed the Celtix version number in the directory name:
  - Makes it easy for several versions of Celtix to co-exist on the same machine
  - Enables you to try a new version before committing to it
- A good naming scheme is  
`celtix-enterprise-<version-number>`
  - For example: `celtix-enterprise-1.0.1`
- Avoid installation directories with names that include white-space.
  - Avoid `c:\Program Files\celtix-1.0`
  - Prefer `c:\dev\celtix-1.0`

# Celtix Environment

# Celtix Environment

---

- Celtix should be used from an appropriately configured shell.
- Always ensure that:
  - The correct JDK (1.5.0 or higher) has been placed on the `PATH`; and
  - The `JAVA_HOME` environment variable has been set to the installation directory of the JDK.
  - The `CELTIX_HOME` environment variable has been set to the installation directory of Celtix.
- To set the environment for Celtix, run the `celtix_env.bat` script.
  - This is located in the `bin` directory of the Celtix installation.
  - It will set a number of environment variables for installed components.
    - For example, if Tomcat is included in the installation, then `CATALINA_HOME` will be set.

# Manifest CLASSPATH in `cxf-incubator.jar`

---

- Any JAR files that Celtix depends on are shipped with the Celtix distribution.
- The manifest file in `cxf-incubator.jar` provides links to these JARs
  - Therefore you only have to have `cxf-incubator.jar` on your CLASSPATH
  - Before Java 1.5 the Java compiler `javac` was unable to use classpath information present in the manifest – this has been recognized as a bug and fixed for JDK 1.5.0.

# Building Celtix from Source

# Using a Celtix source distribution

---

- Some users will prefer to download a source distribution of the Celtix core, CXF.
  - Suitable for those who wish to browse, debug, maintain or improve the Celtix source code.
- There are two ways to obtain the source.
  - Download an extract a source distribution from the CXF web site.
    - Again, use `java -jar <celtix-distribution-jar-file>`
  - Create a local subversion snapshot of the source.
    - Subversion is a popular source-code control system used by the CXF project.
- Using subversion allows you to easily update your snapshot on a regular basis.
  - Allowing you to pick up bug-fixes, enhancements, etc.
  - Allows you to contribute to the source (if you have contributor status).

# Using Subversion to obtain CXF

---

- To create a subversion snapshot of the Celtix source, you need to have a subversion client installed on your machine.
  - Download and install subversion from: <http://subversion.tigris.org>
  - Windows users can use *TortoiseSVN*, which provides a graphical interface to subversion.
- Use the following URI to create a local snapshot of the Celtix source:
  - `svn://svn.forge.objectweb.org/svnroot/celtix`
- You can now easily update your snapshot from the ObjectWeb repository as necessary.
  - Using `svn update` or the TortoisSVN graphical interface.
- See the Subversion documentation for more details.



# Building Celtix from a source distribution

---

- Celtix uses the Apache *Maven* build system.
  - Maven is included in the Celtix distribution.
  - For more information on Maven, see <http://maven.apache.org/>
  - For more information on how Celtix uses Maven, see the Celtix wiki pages.
- At compile-time, Maven contacts a Maven repository to download appropriate JAR files for third-party products used by Celtix.
  - A number of Maven repositories exist on the internet.
    - By default, Celtix uses the *ibiblio* repository:  
<http://www.ibiblio.net>
  - You can configure Celtix to use a different repository if you wish.
    - For example, IONA hosts it's own internal mirror of the Maven repository.
  - See the Maven site for a full list of repositories.

# Building Celtix from a source distribution (cont')

---

- Configure your environment to use Maven:
  - Add `<celtix-src-dir>/maven/bin` to the path.
- Build the source:
  - Move to the source directory: `<celtix-src-dir>`
  - Run: `mvn install`
- The celtix project provides some Maven options that will speed up the compilation process.
  - Run: `mvn -Dfastinstall install` to omit unit-testing and Java style-checking.
  - For more build options, see the Celtix project page.
- The source is now compiled; the instructions on the next slide show how to build a binary distribution.

## Building Celtix from a source distribution (cont')

---

- Move to the `celtix-distribution` directory.
  - Run: `mvn install`
- This will create JAR files for the source and binary distributions in the directory `celtix-distribution/target`.
- Select an appropriate binary distribution from the `target` directory and install.
  - Use the same instructions as given earlier in this chapter.

# Celtix Development Environment: Ant and Eclipse

# Ant

---

- Ant is an open-source build system for Java programs
  - Serves a similar purpose to `make`, but better
    - Does not have the “invisible tab” problem
    - Build “commands” are Java classes rather than OS-specific commands
  - Ant is available freely from <http://ant.apache.org>
  - The online Ant documentation contains a good introduction to Ant.
- Rules for the compilation and packaging are stored in a *build* file
  - Serves a similar purpose to a *Makefile* for `make`
  - Typically called `build.xml`
- An Ant build file can specify how to:
  - Generate code from WSDL files
  - Compile Java files
  - Package `.class` and support files into a `.jar` file

# Ant (cont')

---

- Some IDEs can be integrated easily with Ant:
  - Use IDE for editing and debugging
  - Click on button in IDE to run an Ant build file
- Alternatively, you can:
  - Use a text editor to edit Java files, and...
  - Use Ant to compile files
  - This is useful:
    - For automated, overnight builds
    - When doing development over a slow network connection
    - When doing development on another machine without your favorite IDE
- Advice: use Ant when developing Celtix-based applications
  - Hint: use the build file in the exercise system of this course as a basis for your own projects

# Using the exercise system build files

---

- The Celtix samples contain a `common_build.xml` build file that you can import into your own build files.
- The common build file sets appropriate classpaths for use with Celtix, using the `CELTIX_HOME` environment variable.
- It also contains some useful Ant macros
  - `wsdl2java` – to run the Celtix `wsdl2java` utility.
  - `celtixrun` – to run a Java class with appropriate `CLASSPATH` and JVM properties.
- The build file also contains useful targets
  - `build` – compile all Java source files
  - `clean` – remove all Java object files
  - `generate-code-if-necessary` – will execute a `generate.code` target if any files in the `wsdl` directory have been modified.

# Using the exercise system build files (cont')

```
<project name="..." default="build">  
  <property environment="env"/>
```

import default targets  
and properties for Celtix  
development

```
<import
```

```
  file="{env.CELTIX_HOME}/samples/build-common.xml"/>
```

This rule will get called by the default rules imported  
above.

```
<target name="generate.code" unless="codegen.notrequired">
```

```
  <wsdl2java destdir="{classes.dir}"
```

```
    srcdestdir="{src.dir}"
```

Use of wsdl2java macro

```
    file="{wsdl.dir}/HelloWorld.wsdl"/>
```

```
</target>
```

```
<target name="helloworld.Client" depends="build">
```

```
  <celtixrun classname="helloworld.Client"/>
```

```
</target>
```

Use of celtixrun macro

```
</project>
```



## Using the exercise system build files (cont')

---

- The common build file does a lot of the work for you – feel free to modify, reuse and improve it in your development environment.
- Aside: you may have spotted that Ant has access to the environment variable `CELTIX_HOME`
  - Java applications in general *do not* have access to environment variables for security reasons.
  - While Ant *is* implemented in Java, it uses a default rule to `exec` a system command to retrieve the environment data.
  - To access environment variables in ant, declare an environment property:  
`<property environment="env"/>`
  - Then, use the syntax to reference an environment variable.  
`${env.VARIABLE_NAME}`

# Eclipse

---

- Eclipse is an open-source IDE
  - Very popular among Java developers
  - Provides many features that enhance developer productivity:
    - Syntax highlighting and code completion
    - Refactoring
  - Can be downloaded from <http://www.eclipse.org>
  - For use with Celtix Enterprise, use Eclipse 3.2.1 or higher
- Start eclipse from a shell set for the Celtix environment
  - Enables you to run and debug Celtix applications from within Eclipse
- Tight integration makes Eclipse + Ant a compelling development environment.

## Note: Setting the Celtix classpath in Eclipse

---

- To set the Celtix class path in Eclipse, use the “user library” feature.
- In Eclipse 3.1.1, this is found under the menu path:
  - Window → Preferences → Java → Build Path → User Libraries
- Create a new user library, called “Celtix <version\_number>”.
- Add *all* the jars under the Celtix `lib` directory to the user library.
  - Unfortunately, Eclipse does not yet support the use of the manifest classpath present in `celtix.jar`.
- Also, ensure that Eclipse is configured to work with J2SE 5.0
  - Window → Preferences → Java → Compiler → JDK Compliance
- In your project, add the Celtix user library to the build path.

# Using Eclipse WTP and STP

---

- As well as providing a programming environment (IDE), Eclipse provides tooling plugins for XML Schema and WSDL contracts.
  - WTP (Web Tooling Project)
  - STP (SOA Tooling Project)
- These plugins are open-source, and available for use under the Eclipse license.
- Instructions for installing these plugins can be found in the `CELTIX_HOME/tools/stp` directory.

# Summary

---

- This chapter has told you how to:
  - Download and install Java 1.5
  - Download and install Celtix
  - Configure Ant and Eclipse to work with Celtix
- Celtix is installed by:
  - Untaring or unzipping distribution file
  - Setting a few environment variables
- Eclipse and Ant make a great development environment for Celtix



# Introduction to WSDL – RPC Style

# Overview

---

- Recall: WSDL is used to specify the interface to a service.
  - A WSDL contract has a logical part, used to capture the semantics of a service in a middleware neutral way.
  - A WSDL contract also has a physical part, used to provide payload format, protocol and contact details.
- WSDL can be used to describe:
  - Synchronous request-response interfaces (*RPC* style); and,
  - Asynchronous, messaging-style interfaces (*document* style).
- This chapter teaches how to use WSDL for RPC style interfaces.
  - Document style interfaces are treated in a later chapter.
  - The use of *wrapped-doc-literal* style will also be treated in a later chapter.

# Aside: the complexity of WSDL

---

- WSDL can be difficult for the newcomer.
  - The verbose XML syntax, with heavy use of namespaces, can seem cluttered and unergonomic.
  - With practise, and the use of appropriate GUI design tools, this complexity is greatly reduced.



# Motivating example - HelloWorld

---

- For the purpose of this chapter, we will use the following “HelloWorld” interface as a motivating example.

```
interface HelloWorld
{
    String sayHi();
    String greetMe(String me)
}
```

WSDL concepts: portType, messages and types.

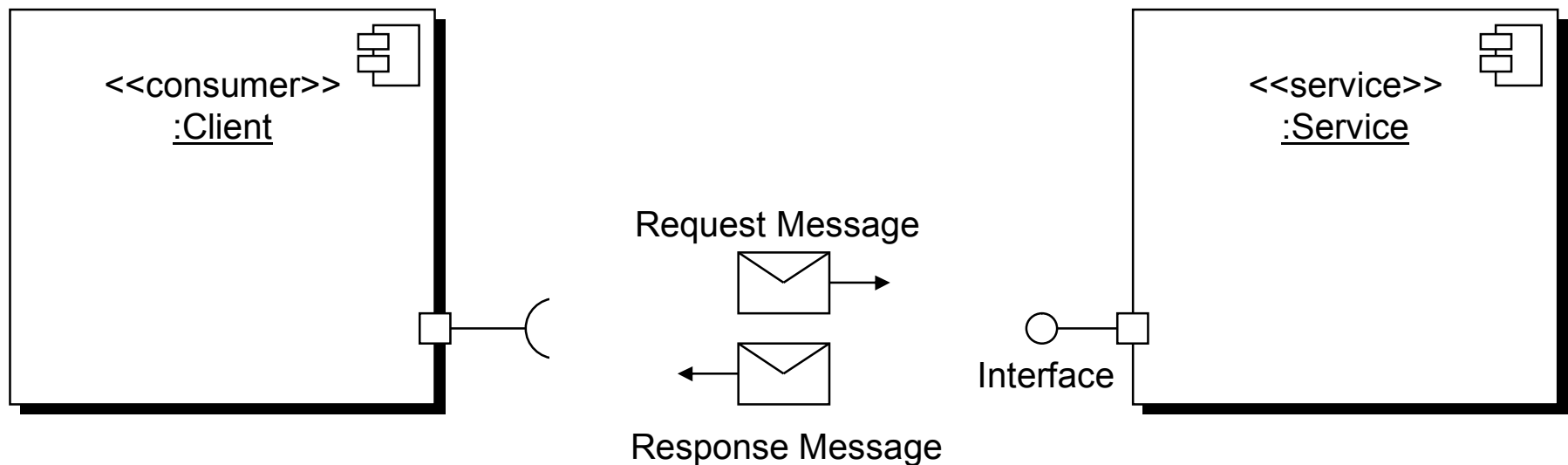
# WSDL concepts: portType and operation

---

- In WSDL terms, a service's interface exposes a number of *operations*.
  - This is similar in concept to a Java interface or C++ header file.
  - WSDL originally used the term *portType* instead of *interface*.
    - This is unfortunate, and can be confusing for some people.
    - This issue has been rectified in WSDL 2.0, where the term *interface* is used.
- In our example, we will define a `portType` (interface) named `HelloWorld`.
  - There are two operations, `sayHi()` and `greetMe()`.

# WSDL concepts: message

- WSDL messages are used to define the data that gets sent to and from a service.
- Each operation has:
  - An input message;
  - An optional output message; and
  - An optional number of *fault* messages.



# RPC-style messages

---

- When designing messages for RPC, each operation will typically have at least two *messages*.
  - The request message contains the input parameters.
  - The response message contains the return value and any additional output parameters.
- In this context, you can consider the message to be a “parameter list”.
  - Each WSDL message contains a number of *parts*, and each part has an associated *type*.
- By convention, the input message has the same name as the operation; the output message ends with `Response`.
- Our HelloWorld interface has *four* messages.
  - *sayHello*, *sayHelloResponse*, *greetMe*, *greetMeResponse*.

HelloWorld.wsdl **logical contract**

# Logical part of HelloWorld.wsdl

---

- We will walk through the logical contract for a simple HelloWorld service.
- We will proceed in a bottom up fashion:
  - Types
  - Messages
  - Operations
  - Interface (portType)
- The WSDL is verbose: you are not expected to absorb it all at once, just get an idea of the structure.

# HelloWorld.wsdl – namespaces and <types>

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld.wsdl"
  targetNamespace=
    "http://www.iona.com/ps/courseware/HelloWorld"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns=
    "http://www.iona.com/ps/courseware/HelloWorld"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types/>
```

No types are required for this simple contract. In this case, we could omit the <types> element altogether.



# HelloWorld.wsdl - <message>

---

```
String sayHi();
```

```
<message name="sayHi" />  
<message name="sayHiResponse">  
  <part name="return" type="xsd:string" />  
</message>
```

```
String greetMe(String me);
```

```
<message name="greetMe">  
  <part name="me" type="xsd:string" />  
</message>  
<message name="greetMeResponse">  
  <part name="return" type="xsd:string" />  
</message>
```

# HelloWorld.wsdl - <portType>

---

```
<wsdl:portType name="HelloWorld">
```

```
String sayHi();
```

```
  <wsdl:operation name="sayHi">
```

```
    <wsdl:input message="tns:sayHi"
      name="sayHi"/>
```

```
    <wsdl:output message="tns:sayHiResponse"
      name="sayHiResponse"/>
```

```
  </wsdl:operation>
```

```
String greetMe(String me);
```

```
  <wsdl:operation name="greetMe">
```

```
    <wsdl:input message="tns:greetMe"
      name="greetMe"/>
```

```
    <wsdl:output message="tns:greetMeResponse"
      name="greetMeResponse"/>
```

```
  </wsdl:operation>
```

```
</wsdl:portType>
```

# WSDL Operation Styles

# Request-response operations

---

- A request-response message has an `input`, `output` and optional `fault` messages.

```
<wsdl:operation name="...">
  <wsdl:input name="..." message="..." />
  <wsdl:output name="..." message="..." />
  <wsdl:fault name="..." message="..." />
  <wsdl:fault name="..." message="..." />
</wsdl:operation>
```

- This operation type is very common in synchronous web services.

# One-way operations

---

- A one-way operation just has a request-message and no fault message.

```
<wsdl:operation name="...">  
  <wsdl:input name="..." message="..." />  
</wsdl:operation>
```

- There is no explicit “one-way” qualifier; the operation is deduced to be one-way because no `wsdl:output` message is present.
- If your operation returns a void, and you don’t want it to be one-way, return a message with no `parts`.
- One-way operations can be used for asynchronous messaging.

# Solicit-response operations

---

- Solicit-response differs from the request-response form only in that the `output` comes before the `input`.
  - This operation style attempts to specify a “callback”; the Service will initiate an invocation on the client (with an output message), and the client will respond (with an input message).
  - However, there is no indication of how the client should register for this callback functionality.
  - Solicit-response is not used by anyone; there is currently no binding that supports it.

```
<wsdl:operation name="...">
  <wsdl:output name="..." message="..." />
  <wsdl:input name="..." message="..." />
  <wsdl:fault name="..." message="..." />
</wsdl:operation>
```

# Notification operations

---

- A “notification” operation has an `output` message only.
  - Like solicit-response, it attempts to capture callback semantics, but fails.
  - There is currently no binding that supports this.

```
<wsdl:operation name="...">  
  <wsdl:output name="..." message="..." />  
</wsdl:operation>
```

- *Philosophical note*: most standards are good; however, most have some flaws or warts.
  - Operations styles like “solicit-response” and “notification” are unfortunate flaws in the WSDL specification.

# Physical part of HelloWorld.wsdl



# WSDL concept: bindings

---

- The physical part of a contract contains *bindings*, *services* and *ports*.
- The binding describes how the messages will be formatted when sent on the wire.
  - Historically, WSDL provides two different bindings for RPC style documents; *RPC-encoded* and *RPC-literal*.
  - Of these, the RPC-encoded binding is now considered deprecated.
    - The term “encoded” refers to the “SOAP encoding”
    - Some tools continue to support it for backward compatibility.
- Bindings are structurally similar to `portType` elements.
  - They contain operations with inputs and outputs.
  - They decorate this structure with information on how to format the data on the wire.
  - For completeness, we show an RPC-literal binding on the next slide.

# HelloWorld.wsdl - <binding>

---

```
<wsdl:binding name="HelloWorldSOAPBinding"
  type="tns:HelloWorld">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="sayHi">
    <soap:operation soapAction="" style="rpc"/>
    <wsdl:input name="sayHi">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="sayHiResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>

  <!-- continued on next slide -->
```

# HelloWorld.wsdl - <binding> (cont')

---

```
<!-- continued from previous slide -->
```

```
<wsdl:operation name="greetMe">  
  <soap:operation soapAction="" style="rpc"/>  
  <wsdl:input name="greetMe">  
    <soap:body use="literal"/>  
  </wsdl:input>  
  <wsdl:output name="greetMeResponse">  
    <soap:body use="literal"/>  
  </wsdl:output>  
</wsdl:operation>
```

```
</wsdl:binding>
```

# Critique: <binding> element

---

- The `binding` element is verbose, and difficult to write and maintain by hand.
- The `binding` element is structurally quite similar to the `portType` element – it seems like unnecessary repetition.
- Advice: use a tool to automatically generate bindings for you:
  - Artix provides an Eclipse plug-in that can do this.
  - The Eclipse Web Tooling Project (WTP) and SOA Tooling Project (STP) provides similar functionality that can be used with Celtix.

# Critique: SOAP vs. literal binding for RPC

---

- To appreciate the difference between the SOAP and literal encodings, look at the messages get produced.
- For the `greetMe()` message, the SOAP encoding produces (omitting SOAP envelope and body):

```
<greetMe>  
  <me xsi:type="xs:string">Joe Bloggs</me>  
</greetMe>
```

- The equivalent message in *literal* encoding is:

```
<greetMe>  
  <me>Joe Bloggs</me>  
</greetMe>
```

# Critique: SOAP vs. literal binding for RPC (cont')

---

- The *SOAP-encoded* binding adds type information to each element in the message.
  - This exchange of type information on every message makes the message bulky.
- The *literal* binding is more compact.
  - Also, the literal encoding has better-defined rules for encoding arrays; the SOAP encoding was poor in this regard which lead to interoperability issues.

# Critique: SOAP vs. literal binding for RPC (cont')

---

- Messages produced from SOAP-encoded and literal bindings for RPC both suffer from the fact that they cannot be validated by an XML parser.

- Consider the RPC-literal message:

```
<greetMe>  
  <me>Joe Bloggs</me>  
</greetMe>
```

- The elements `greetMe` and `me` do not have an associated XML schema.
  - As such, they can be parsed *but not validated* by an XML parser.
  - This is a drawback; however, many SOAP toolkits turn off validation by default anyway, for performance reasons.

# WSDL concepts: port

---

- A WSDL port takes adds physical address information to a binding.
  - Below, the SOAP binding for the `HelloWorld` interface is bound to the HTTP transport.

```
<wsdl:port binding="tns:HelloWorld_SOAPBinding"
           name="SOAPOverHTTP">
  <soap:address location="http://localhost:9000"/>
</wsdl:port>
```

- A port must be placed in the context of a service element.
  - In WSDL terminology, a *service* is a collection of ports.



# HelloWorld.wsdl - <service> and <port>

---

- A service is a collection of ports.
  - This service below contains just one `port`.

```
<wsdl:service name="HelloWorldService">
  <wsdl:port binding="tns:HelloWorld_SOAPBinding"
    name="SOAPOverHTTP">
    <soap:address location="http://localhost:9000"/>
  </wsdl:port>
</wsdl:service>
```

- The semantics of the service element are flexible; it could mean;
  - “This service is accessible via a number of middleware technologies”;
  - or,
  - “There are a number of instances of this service”.

# Summary

---

- This chapter has used a simple interface to illustrate how a WSDL file is put together.
- WSDL is overly verbose
  - A four-line psuedo-code interface took pages of WSDL.
  - There is unfortunate repetition (e.g., structure of portType and binding)
  - XML syntax itself is verbose
  - ... but you get used to it.
- Most vendors provide tooling to assist in WSDL design
  - GUI tools for WSDL first; and
  - Code generation tools to generate WSDL from other interface languages such as COBOL, PL1, JAVA,
- The slide overleaf provides a useful reference overview of the syntax of a WSDL contract.

# Reference: outline of a WSDL contract

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld" xmlns:wsdl="...">

  <wsdl:types> ... </wsdl:types>
  <wsdl:message name="...">
    <wsdl:part name="..." element="..." />
  </wsdl:message>
  <wsdl:portType name="...">
    <wsdl:operation name="...">
      <wsdl:input name="..." message="...">
      </wsdl:operation>
    </wsdl:portType>
  </wsdl:portType>
  <wsdl:binding name="..." type="...">... </wsdl:binding>
  <wsdl:service name="...">
    <wsdl:port binding="..." name="..."> ... </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Logical part

Physical part



# WSDL-to-Java Mapping with JAXWS

# Overview

---

- The JAX-WS WSDL to Java mapping is largely intuitive.
  - WSDL `portType` → Java interface
  - WSDL `operation` → Java method
  - WSDL `message part` → to Java method parameters
  - WSDL `fault` → Java `Exception`
  - XSD types and elements → Java classes as per JAXB
  
- Most developers will learn what they need to know by browsing the generated Java code.
  - For a detailed description of the mapping, see the JAX-WS specification.
  
- This chapter discusses some not-so intuitive parts of the mapping.
  - Support for wrapped-doc-literal WSDL
  - Exception wrapping

# Support for Wrapped-doc-literal

# Support for wrapped-doc-literal

---

- Recall: in the *wrapped-doc-literal* style, an RPC call is transferred as a single XML element.
  - The name of the root element is the same as the operation name.
  - The sub-elements are the parameters.
- The JAX-WS specification *supports* both *doc-literal* and *wrapped doc-literal*.
  - The term *supports* means that a JAX-WS wsdl-to-java compiler should generate appropriate Java code for the intent of the WSDL designer.
  - If *wrapped-doc-literal* is detected then the operation parameters will “unfolded” and appear as a list of method arguments in the Java code.
  - If *doc-literal* is detected, then the Java method will contain just a single parameter, corresponding to the incoming document.
- The rules for detecting the wrapped-doc-literal style are specified in the JAX-WS specification.

# Mapping wrapped operations

---

- If an operation is detected as being wrapped, then its parameters are defined by the wrapper children
  - The term “wrapper children” is used for the elements in the wrapper sequence.
- Each wrapper child in the input message is an *in* parameter.
  - The wrapper child maps directly to the equivalent JAXB class.
- Each wrapper child in the output message is an *out* parameter
  - The wrapper child maps directly to a `Holder` for the equivalent JAXB class.
- Any wrapper child in the input *and* output message is treated as an *in-out* parameter.
  - The wrapper child maps to a `Holder` for the equivalent JAXB class.
  - Two children are “the same” if they have the same name and same type.



## Return type for *wrapped* operations

---

- If there is a single *out* wrapper child then it becomes the return-type.
- Alternatively, if there is more than one *out* wrapper child, then the child named `return` becomes the return-type.
  - If there is no child named `return` then the return-type is `void`.

# Example: wrapped-doc-literal operation

---

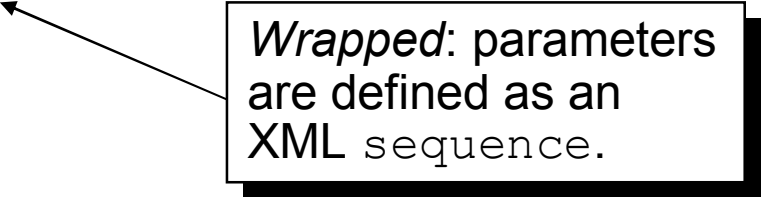
```
<types>
  <schema targetNamespace="http://www.iona.com/artix/mapping"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="doSomething">
      <complexType>
        <sequence>
          <element name="x" type="int"/>
          <element name="y" type="long"/>
        </sequence>
      </complexType>
    </element>
```

*Wrapped:* parameters are defined as an XML sequence.

# Example: wrapped-doc-literal operation (cont')

---

```
<element name="doSomethingResponse">
  <complexType>
    <sequence>
      <element name="y" type="long"/>
      <element name="z" type="float"/>
      <element name="return" type="boolean"/>
    </sequence>
  </complexType>
</element>
</schema>
</types>
```



*Wrapped:* parameters are defined as an XML sequence.

## Example: wrapped-doc-literal operation (cont')

*Wrapped:* messages have one part, referring to a complex sequence element.

```
<message name="doSomething">
  <part element="tns:doSomething" name="parameters" />
</message>
<message name="doSomethingResponse">
  <part element="tns:doSomethingResponse" name="parameters" />
</message>
<portType name="Foo">
  <operation name="doSomething">
    <input message="tns:doSomething" name="doSomething" />
    <output message="tns:doSomethingResponse"
      name="doSomethingResponse" />
  </operation>
</portType>
```

# Generated Code

---

- Using the rules for unwrapped operations, the JAX-WS specification produces the following service endpoint interface (SEI):

```
public interface Foo {  
    public boolean doSomething(  
        int x,  
        Holder<Long> y,  
        Holder<Float> z  
    );  
}
```

# JAX-WS fault-mapping

# JAX-WS fault-mapping

---

- A fault is usually defined as an XML sequence.
- For example, this fault contains a `message` and `errorCode`.

```
<element name="FooFault">
  <complexType>
    <sequence>
      <element name="message" type="string"/>
      <element name="errorCode" type="int"/>
    </sequence>
  </complexType>
</element>
```

# Defining a fault message

---

- To declare an exception in WSDL, you must create a fault message.

```
<message name="FooFault">
  <part element="tns:FooFault" name="parameters"/>
</message>
<portType name="Foo">
  <operation name="doSomething">
    <input message="tns:doSomething" name="doSomething"/>
    <output message="tns:doSomethingResponse"
      name="doSomethingResponse"/>
    <fault message="tns:FooFault" name="FooFault"/>
  </operation>
</portType>
```



# Mapping for faults

---

- The fault, `FooFault`, is mapped using JAXB to a Java class.
  - JAXB does not apply any special “fault” treatment to this `sequence`.
  - In particular, the generated class will *not* extend `java.lang.Exception` or implement `java.lang.Throwable`.
  - This class is known as the “fault bean”.
- The fault bean has get- and set- methods for each of the exception fields.
- Aside: the mapping for faults is not natural for Java developers, so this violates JAX-B’s goals.

# Mapping for fault message.

---

- The fault message, `FooFault`, is mapped to an *exception wrapper* class with the same name as the message.
  - If the message has the same name as the fault bean (as in this example) then the name is derived by concatenating the name of the fault bean with “\_Exception”.
- The exception wrapper class has the following properties:
  - It extends `java.lang.Exception`.
  - It has appropriate constructors:

```
FooFault_Exception(String message, FooFault faultInfo);  
FooFault_Exception(String message, FooFault faultInfo,  
                   Throwable cause);
```

- It has a `getFaultInfo()` method that returns the fault bean.

# Fault method signature

---

- The method signature now throws the appropriate exception wrapper class.

```
public interface Foo {  
    public boolean doSomething(  
        int x,  
        Holder<Long> y,  
        Holder<Float> z  
    ) throws FooFault_Exception;  
}
```

# Throwing an exception

---

```
public boolean doSomething(  
    int x,  
    Holder<Long> y,  
    Holder<Float> z  
) throws FooFault_Exception  
{  
    // Create the fault bean.  
    FooFault fault = new FooFault();  
    fault.setMessage("Something went wrong.");  
    fault.setErrorCode(001);  
  
    // Wrap the fault bean and throw the exception wrapper  
    throw new FooFault_Exception(fault);  
}
```

# Summary

---

- The JAX-WS WSDL to Java mapping is largely intuitive.
  - It relies on JAXB to map XML types and elements to Java classes.
- The best way to learn the mapping is to learn as you go.
  - Design some WSDL, generate code and review.
- We have looked in detail at two non-trivial aspects of the mapping.
  - How parameters and return-types are determined for wrapped WSDL operations.
  - How faults are mapped to fault beans and exception wrappers.



# Simple Object Access Protocol (SOAP)

# Overview

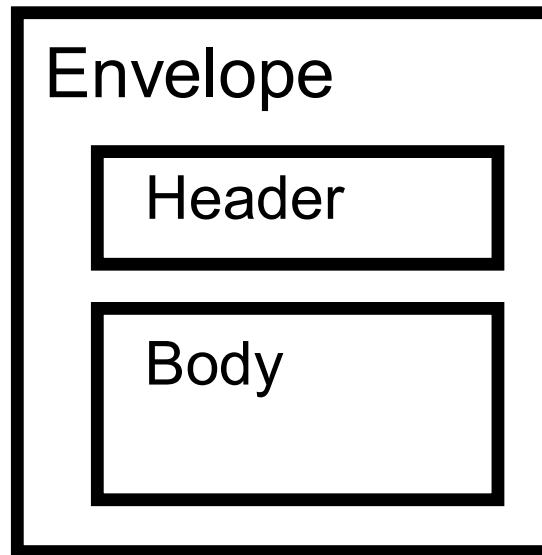
---

- SOAP stands for Simple Object Access Protocol.
  - Designed for accessing objects across a network using XML messaging.
  - The protocol was seen as an alternative to binary protocols such as COM, IIOP or RMI.
  - SOAP is “firewall-friendly” and suitable for use on the internet.
  - Despite it’s name, SOAP is concerned with *services* rather than objects.
- SOAP is no longer “simple”: it has matured to include alternative encodings, security, transactions, and encryption.
- The latest version of the SOAP standard is SOAP 1.2
  - Resource: <http://www.w3.org/TR/SOAP/>

# Structure of a SOAP message

---

- A SOAP message is an XML document that provides a “wrapper” around the XML *payload* (data).
- The message has an `Envelope` element that identifies the message boundary and includes:
  - An optional `Header` (containing meta-data, system-level data or other auxiliary information)
  - A `Body` (containing the XML payload).





# Skeleton SOAP Message

---

- The following fragment shows a skeleton SOAP message
- Like any XML document, it has one root element (the `Envelope`) and it makes extensive use of namespaces.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=" ... ">
  <soap:Header>
    ... ..
  </soap:Header>
  <soap:Body>
    ... ..
    <!-- message payload goes here -->

  </soap:Body>
</soap:Envelope>
```

# Sample SOAP request

---

- The following SOAP message is taken from a sample “hello world” server:

```
<?xml version='1.0' encoding='utf-8'?>
```

```
<ENV:Envelope
```

```
  xmlns:ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
  xmlns:m1="http://IONA.com/HelloWorld">
```

```
<ENV:Body>
```

```
  <m1:greetMe>
```

```
    <stringParam0 xsi:type="xsd:string">
```

```
      Hello From WS Client
```

```
    </stringParam0>
```

```
  </m1:greetMe>
```

```
</ENV:Body>
```

```
</ENV:Envelope>
```

# Sample SOAP response

---

- This message is the response to the previous message.

```
<?xml version='1.0' encoding='utf-8'?>
<ENV:Envelope
  xmlns:ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:m1="http://IONA.com/HelloWorld">
  <ENV:Body>
    <m1:greetMeResponse>
      <return xsi:type="xsd:string">
        Echo: Hello From WS Client
      </return>
    </m1:greetMeResponse>
  </ENV:Body>
</ENV:Envelope>
```

# Notes on the sample messages

---

- Neither message used a `Header` element.
  - Recall that headers are optional.
- Both messages differ only in the contents of the `Body` element.

# SOAP Fault

---

- A service can return a SOAP Fault in the event of an error.
  - A *fault* is semantically equivalent to an exception in C++ or Java.
- SOAP Faults are placed within the Body element of the SOAP message.
- SOAP Faults can contain the following elements:
  - `faultCode`: indicates the type of fault (see next slide)
  - `faultstring`: a human-readable message describing the error.
  - `detail`: XML data providing further information about the fault.
- The next slide gives an example of a SOAP Fault.
  - The fault captures an application-level server-side exception: a request arrived for a customer named “scott” who does not exist.

# SOAP Fault: example

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="...">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>NoSuchCustomer</faultstring>
      <detail>
        <ns5:NoSuchCustomer
          xmlns:ns5="...">
          <firstName>scott</firstName>
          <lastName></lastName>
        </ns5:NoSuchCustomer>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Indicates a server-side problem.

Descriptive human-readable text.

Application-level fault information.

# SOAP faultCode

---

- The faultCode element can contain four values
  - `Server`: the fault was raised by the server
  - `Client`: the fault was raised on the client
  - `MustUnderstand`: the original message contained a header that the service did not understand.
  - `VersionMismatch`: an incompatible SOAP version was encountered.

# RPC-style communication with SOAP

---

- SOAP can be used for RPC-style communication.
  - RPC tries to make remote network calls look and feel like local procedure calls in a programming language
- RPC communication is typically synchronous in nature.
  - The client blocks until the request has completed and the response (if any) has been returned.
- RPC communication is familiar to people who have used CORBA, RMI or DCOM
- When SOAP is used for RPC, the SOAP request and response contain the parameters (and optional return values) for the remote procedure call.



# Document-style communication with SOAP

---

- SOAP can also be used for document-style communication
  - Document-style systems focus on sending messages (documents) from one party to another.
- Document-style communication is typically *asynchronous* in nature.
  - The client sends a document but does not wait for a response.
  - The client may optionally listen for an asynchronous response using either polling or a callback.
- Document-style communication is familiar to people who have used JMS, CORBA Notification Service, or message queue systems.
- When SOAP is used for document-style communication, the SOAP request (and response) messages contain an XML document.

# SOAP with Attachments

---

- SOAP provides an “XML envelope” for XML documents.
- An “all-XML” approach may be inappropriate for some applications.
  - How would you transmit an X-ray image along with a patient record?
- Non-XML data can be transmitted in a SOAP message using MIME attachments.
  - MIME stands for Multi-purpose Internet Mail Extensions.
  - Originally intended for use with email (SMTP) protocol, it is now also used to implement “SOAP with Attachments”.
- A MIME message has:
  - Some MIME *headers*
  - A number of *parts*.

# SOAP with Attachments (cont')

```
Content-type: multipart/mixed; boundary="MIME_boundary"
```

```
MIME-version: 1.0
```

```
--MIME_boundary
```

```
Content-type: text/plain
```

```
<?xml version='1.0' encoding='utf-8'?>
```

```
<ENV:Envelope>
```

```
  <ENV:header>
```

```
    ...
```

```
  </ENV:header>
```

```
  <ENV:Body>
```

```
    ...
```

```
  </ENV:Body>
```

```
</ENV:Envelope>
```

```
--MIME_boundary
```

```
Content-type: application/octet-stream
```

```
Content-transfer-encoding: base64
```

```
gajwO4+....
```

```
--MIME_boundary--
```

MIME headers

SOAP Message is wrapped in a MIME part, transmitted in plain-text.

Attachment is transmitted as a base64 encoded octet stream.

# SOAP Processing

---

- Software responsible for the generation, transmission, reception and analysis of these messages is typically known as a SOAP Processor
- A SOAP processor can be a standalone listener on a TCP port
  - Accepting incoming SOAP messages and passing them up in the stack
- Alternatively, the SOAP processor may be part of a web server such as Apache.
  - If your web server doesn't include a SOAP processor out-of-the-box, then it is most likely that you can add a "SOAP plug-in".

# SOAP 1.2

---

- SOAP 1.1 is perhaps the most used version of SOAP specification
- The latest release is SOAP 1.2
  - Incorporates SOAP with attachments (controversial, as it implies that the client and server will understand the attachments)
  - Recommended by WS-I in its Basic Profile 1.0
    - Allows for additional information payload as an attached file – that may contain binary as well as text data
    - Now being incorporated into existing SOAP implementations
    - Other web service standards and interoperability efforts also starting to build on top of SOAP to allow for support of attachments
- Reference: <http://www.w3.org/TR/soap>

# Summary

---

- SOAP provides an XML-based “wrapper” around messages.
- While SOAP is typically transmitted over HTTP, it does not rely on HTTP in any way.
  - You can send SOAP messages over JMS, IIOP, email, ...
- SOAP supports both synchronous RPC-style and asynchronous document-style messaging



# Universal Discovery, Description and Integration (UDDI)

# UDDI

---

- UDDI (Universal Description, Discovery and Integration) is an OASIS sponsored standard.
- UDDI is like a “yellow” pages for web services:
  - Service providers register/publish services including WSDL file along with searchable attributes
  - Potential clients search UDDI registries to retrieve WSDL suiting their service needs
- Note: UDDI is the *least* accepted of the core web services technologies (SOAP, WSDL and UDDI).
  - It is complex, and burdened by a lot of business level aspects (e.g., descriptions of a business).
- Resource: <http://www.uddi.org>



# UDDI business registries

---

- Public UDDI business registry (UBR) nodes are hosted by IBM, Microsoft and SAP.
  - <http://uddi.ibm.com>
  - <http://uddi.microsoft.com>
  - <http://uddi.sap.com>
- UBRs are themselves web services
  - They implement the UDDI WSDL interface, which provides APIs to register and discover web services.

# UDDI implementations

---

- A number of implementations of UDDI exist that allow you to run your own UDDI registry.
- jUDDI – open source UDDI registry from Apache
  - <http://ws.apache.org/juddi/>
- SOAP UDDI – open source UDDI reference implementation
  - <http://soapuddi.sourceforge.net/>
- A number of commercially available web service toolkits also provide some support for UDDI clients and servers.

# Practical alternatives to UDDI

---

- For many, UDDI is unwieldy and over-engineered.
- It may be more appropriate to advertise web services through:
  - Web server: mount your WSDL files in a well known location so service-users can access them easily.
  - Shared file-system: place your WSDL files in a well-known place in the file-system.
  - File copy: give each client distribution it's own copy of the WSDL contract, for example in an "etc" directory.
    - This may lead to problems if the interface changes (but then, if the interface changes then it's likely that you will have a new version of the client software anyway)
    - Will cause problems if the service location information changes. However, location information is best kept configurable at application level.

# Summary

---

- UDDI provides a specification for web service repositories, known as UDDI Business Registries (UBRs)
- A number of open-source and commercial implementations of UDDI exist.
- UDDI is not an essential component of web services.
  - Instead, you can use shared file-systems, shared files or a web server to make your WSDL contracts available.



# Overview of Celtix Development

# Introduction

---

- This chapter uses a “Hello, World” client-server example.
- Structure of this chapter:
  - Recommended directory structure for a Celtix project
  - How to generate Java APIs from a WSDL contract;
  - Server-side code:
    - Finish the “servant” class, which implements the operations provided by the service and defined in WSDL;
    - Instantiate the servant;
    - Register the servant as an endpoint.
  - Client-side code:
    - Connect to the service and invoke an operation.
- We use common-sense conventions to organize code.

# Structure of a Celtix Project

# Directory structure of a Celtix project

---

- Some conventions for directory structure in Celtix project.
  - You do not have to follow these conventions, but it is beneficial to do so.
  
- A project typically contains the following directories:
  - `build/classes`      Contains compiled Java classes.
  - `build/src`            Contains generated Java source code.
  - `src`                    Contains handwritten Java source code.
  - `wsdl`                    Contains WSDL files.
  
- Some other directories are also common:
  - `cfg (or conf)`        Contains configuration information
  - `etc`                    Contains miscellaneous files.
  - `lib`                    Contains JAR files needed for compilation
  - `log`                    Contains log files generated at run-time.



# Directory structure of a Celtix project (cont')

---

- The top level project directory contains:
  - The Ant build file (`build.xml`).
  - Eclipse `.classpath` and `.project` files.
  - Any other project-related files.

# Overview of the HelloWorld Project

# Overview of “Hello, World”

---

## ■ Historical note:

- “Hello, World” dates back to 1978
- First program in *The C Programming Language* by Kernighan and Ritchie
- Original (non-distributed) example just printed “Hello, World” to the screen

## ■ A server provides `String sayHello(String msg)`

- Server prints the message it receives and returns “Hello to you too!”

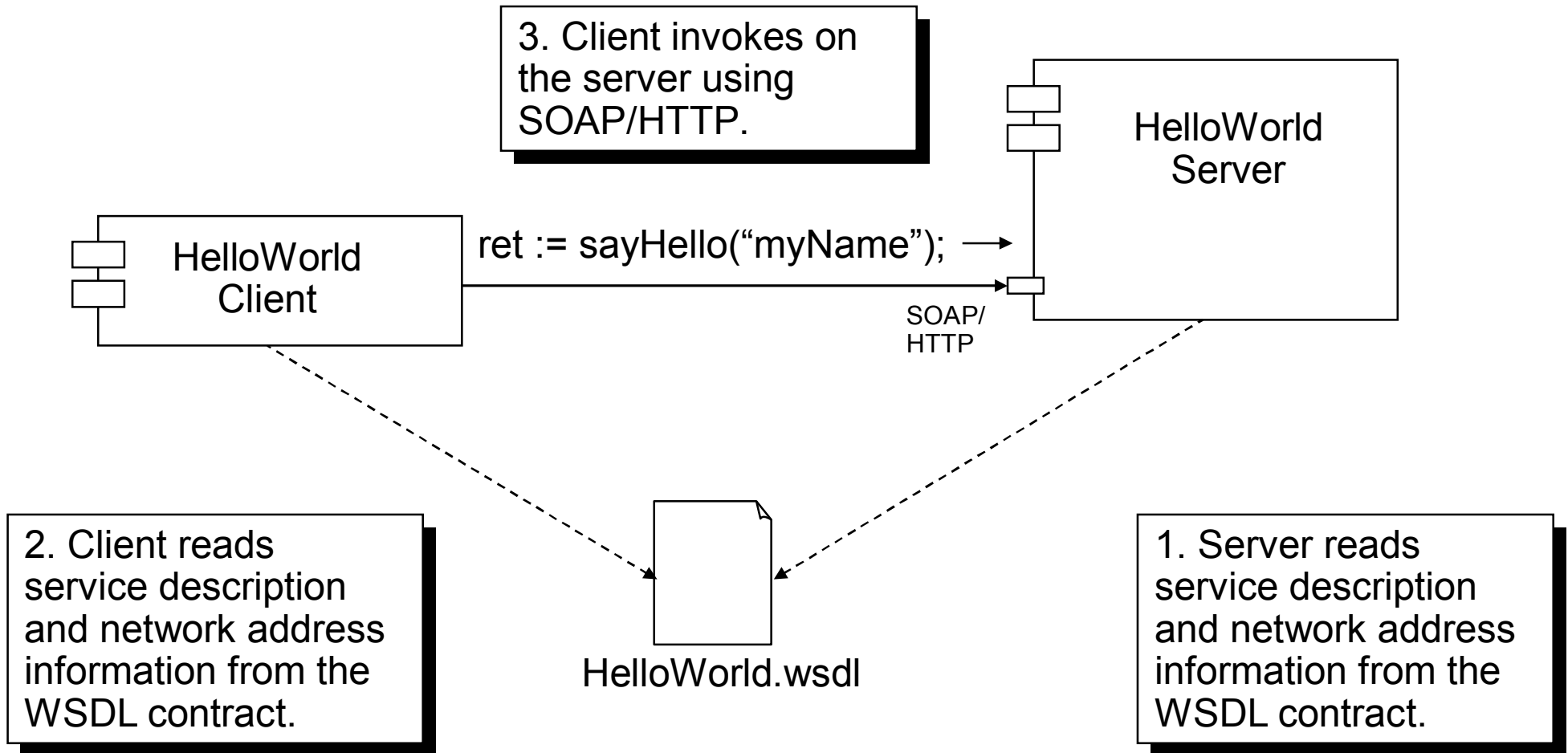
## ■ The server’s interface has already been defined using WSDL

- You can find this file in  
`celtix_exercises/HelloWorld/student/wsdl/HelloWorld.wsdl`

## ■ This example demonstrates key development tasks:

- How to generate Java support code from WSDL
- How to implement a web service
- How to implement a web service client

# Deployment view



# Deployment view (cont')

---

Notes on client and server interaction...

## ■ Server:

- Uses WSDL file to create a *service endpoint* for HelloWorld service.
- Goes into event loop to listen for incoming requests

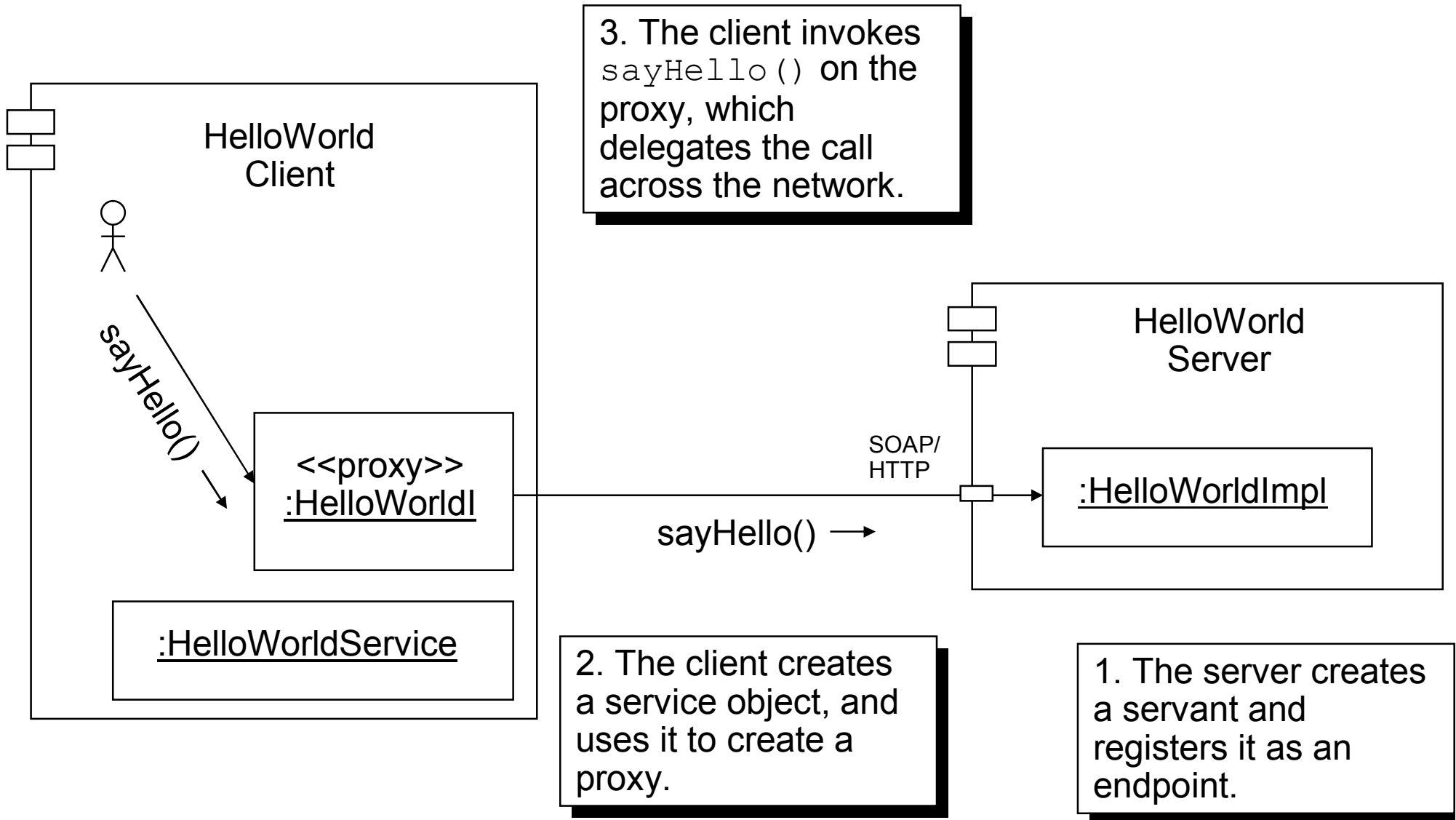
## ■ Client:

- Reads service endpoint information from the WSDL file
- Invokes `sayHello()`, sending a string message as a parameter.

## ■ Server (on receiving a request):

- Prints string parameter to screen
- Returns a string response.

# Object view



# Classes

---

- The classes described in the next slides are representative of *every* client and server:
- *Service*:
  - A client-side object that describes the remote service, and provides methods to create proxies to the remote service.
  - Generated from the WSDL by `wsdl2java` (a Celtix/CXF tool)
- *Proxy*:
  - A client-side object used to provide a local interface to a remote service.
  - Generated from the WSDL by `wsdl2java`
- *Servant*:
  - A server-side object that implements the operations defined in WSDL file
  - Handwritten by the developer.

# Generated code: the service class

```
@WebServiceClient(  
    name = "HelloWorldService",  
    targetNamespace =  
        "http://www.iona.com/ps/courseware/HelloWorld",  
    wsdlLocation = "./wsdl/HelloWorld.wsdl"  
)
```

Service information,  
declared using annotation.

```
public class HelloWorldService extends Service
```

```
{  
    public HelloWorldService(URL wsdlLocation,  
        QName serviceName) {...}
```

Constructor takes  
the WSDL  
location and  
service name

```
@WebEndpoint(name = "SOAPOverHTTPEndpoint")  
public HelloWorldI getSOAPOverHTTPEndpoint()
```

Method to create  
a HelloWorld  
proxy.

```
}
```



# Generated code: the proxy interface

```
@WebService (  
    name = "HelloWorld",  
    targetNamespace =  
        "http://www.iona.com/ps/courseware/HelloWorld",  
    wsdlLocation = "./wsdl/HelloWorld.wsdl"  
)  
public interface HelloWorld  
{  
    // some code & annotations omitted for clarity  
    //  
    public String sayHello(String message);  
}
```

Service information,  
declared using annotation.

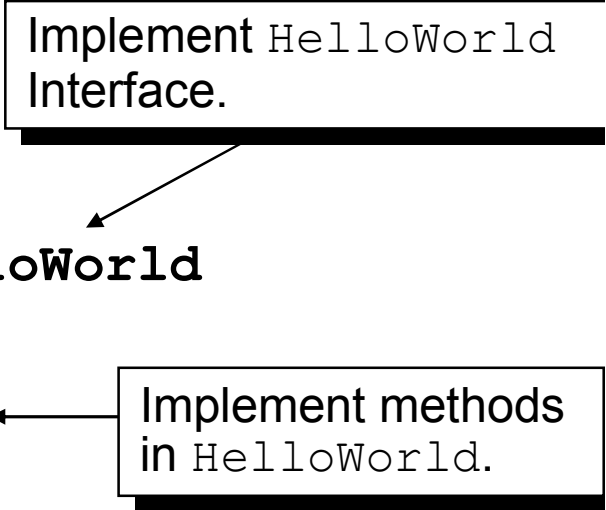
Interface generated  
from WSDL  
HelloWorld  
interface

Method generated  
from WSDL  
sayHello  
operation.

# The servant

---

```
@WebService(  
    name = "HelloWorld",  
    serviceName = "HelloWorldService",  
    targetNamespace  
        = "http://www.iona.com/ps/courseware/HelloWorld",  
    wsdlLocation = "./wsdl/HelloWorld.wsdl",  
    portName = "SOAPOverHTTPEndpoint"  
)  
public class HelloWorldImpl implements HelloWorld  
{  
    public String sayHello(String message) ← Implement methods  
    in HelloWorld.  
    {  
        // Your implementation code goes here  
    }  
}
```



# Directory Structure organization of HelloWorld

---

- The HelloWorld project uses the directory structure discussed in the *Structure of a Celtix Project* section earlier in this chapter
- The `src` directory contains one package:
  - Package `helloworld` (handwritten files)
    - `Client.java`
    - `Server.java`
    - `HelloWorldImpl.java`
- The `wsdl` directory contains `HelloWorld.wsdl`

Note: it is a good idea to keep handwritten code in a separate package/directory from generated code.

# Generating Code from WSDL Files

# Generating code with `wsdl2java`

---

- Every web services product provide a tool to generate programming APIs from WSDL:
  - The Celtix tool is called `wsdl2java`
  - It generates Java classes *and* compiles them

- Typical usage:

```
wsdl2java -s src -d classes -keep ./wsdl/HelloWorld.wsdl
```

- The command-line options are as follows:

- s `src`: put generated Java code into the source directory `src`.
- d `classes`: put compiled code into the destination directory `classes`.
- keep: keep the generated Java source code (it gets deleted by default).

- The `wsdl2java` tool generates *many* Java classes – you only need to know those that correspond to your service interface and types

# Default package name

---

- The generated classes are placed in a Java package.
  - The default package name is derived from the target namespace of the WSDL document.

- Example: the namespace in `HelloWorld.wsdl` is:

`http://www.ionas.com/ps/courseware/HelloWorld`

- This is converted to a package name as follows:
  - The leading `http://www` is stripped
  - The order of the domain name is reversed, giving `com.ionas`
  - The remaining components in the URL path are converted to lower case and appended using a “.” separator, giving:  
`com.ionas.ps.courseware.helloworld`

# Implementing the HelloWorld web service

# Implementing the service

---

## ■ Convention:

- Name of implementation class = `<name-of-interface>Impl`
- This is just a convention, and you're free to ignore it.

## ■ You may wish to explicitly implement the generated interface that corresponds to the WSDL `portType`.

- This is optional.

## ■ You should provide appropriate values for the `@WebService` annotation.

- You should explicitly identify the service and port that this class implements.
- See next slide.



## Implementing the service (cont')

---

```
@WebService(  
    name = "HelloWorld",  
    serviceName = "HelloWorldService",  
    targetNamespace  
        = "http://www.iona.com/ps/courseware/HelloWorld",  
    wsdlLocation = "../wsdl/HelloWorld.wsdl",  
    portName = "SOAPOverHTTPEndpoint"  
)  
public class HelloWorldImpl implements HelloWorld  
{  
    public String sayHello(String message)  
    {  
        return "Hello right back at ya!";  
    }  
}
```

# Implementing the server mainline

---

- A server mainline typically does the following:
  - Create the servant (service implementation object)
  - Register the servant as an “endpoint”
- The next slide show how to perform these tasks:
  - A later chapter provides more detail on the API calls.
- You will need to import *at least* the following classes:

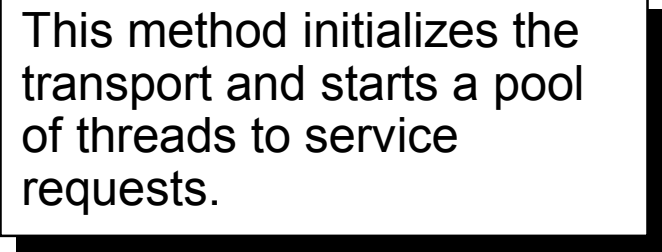
```
import javax.xml.ws.Endpoint;
```

# Pseudo-code of server mainline

---

```
void main(String[] args)
{
    Object helloWorldImpl = null;
    String address = "http://localhost:9090/helloworld";

    helloWorldImpl = new HelloWorldImpl();
    Endpoint.publish(address, helloWorldImpl);
}
```



This method initializes the transport and starts a pool of threads to service requests.

## Aside: `Endpoint.publish()`

---

- The call to `Endpoint.publish()` is non-blocking – it returns immediately.
- As a result the main-thread of the program will complete
- However, the program will not exit.
  - It remains running because of the underlying pool of threads created by the call to `Endpoint.publish()`.
- This behavior may seem strange to non-Java programmers.

# Implementing the HelloWorld Client

# Implementing the client

---

- A client mainline typically performs the following tasks:
  - Declare the service QName and the location of the WSDL file.
    - The term *QName* refers to a “qualified name”, containing a namespace and, in this case, the service name.
  - Create the service
  - Use the service to create a proxy to the *remote* service implementation
  - Invoke on the service.
  
- The next slides show how to perform these tasks:
  - A later chapter provides more detail on the API calls.

# Implementing the client (contd.)

---

- You will probably need to import *at least* the following classes:

```
import java.io.File;
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
```

# Pseudo-code of client mainline

```
public static void main(String[] args)
{
```

Identifies service in WSDL file.

```
    QName serviceName = new QName (
        "http://www.iona.com/ps/courseware/HelloWorld",
        "HelloWorldService");
```

Specifies location of WSDL file as a valid `javax.net.URL`.

```
    URL wsdlURL = null;
```

```
    String wsdlFileLocation = "./wsdl/HelloWorld.wsdl";
```

```
    try {
```

```
        wsdlURL = new File(wsdlFileLocation).toURL();
```

```
    }
```

```
    catch (MalformedURLException e) {
```

```
        System.out.println("Error creating a URL from file '" +
            wsdlFileLocation + "'; details: " + e);
```

```
    }
```



# Pseudo-code of client mainline (cont')

---

Creates a service object using the `wSDLURL` and the `serviceName`.

```
HelloWorldService helloWorldService =  
    new HelloWorldService(wSDLURL, serviceName);
```

Creates a port for the SOAP-over-HTTP endpoint.

```
HelloWorld helloWorld =  
    helloWorldService.getPort(  
        new QName(  
            "http://www.iona.com/ps/courseware/HelloWorld",  
            "SOAPOverHTTPEndpoint"),  
        HelloWorld.class);
```

Invokes the `sayHello()` operation on the server.

```
String response = helloWorld.sayHello("Hello!");
```

# Running the client and server

# Running the client and server using Ant

---

- Ant can be used to launch Java processes.
  - This has the advantage that the Java process will pick up the same environment as that used to build the project.
  - The “Hello, World” demo provides tasks to launch the client and server.
  - While this is useful in development, it is not suitable for production due to the unnecessary overhead of starting Ant and parsing the build.xml file.
  
- Open two command windows and go into the root directory of the HelloWorld project
  
- In one window, run:

```
ant helloworld.Server
```
  
- In the other, run:

```
ant helloworld.Client
```

# Running the client and server from a script

---

- Celtix clients and servers can also be run from the shell.
  - You have to make sure that the CLASSPATH and other environment variables are set correctly – this is best done in a script.
  - The “Hello, World” demo provides scripts to run the client and server.
  - Feel free to model your own startup scripts on those provided.
- Open two command windows and go into the root directory of the HelloWorld project
- In one window, run:  
`helloworldserver`
- In the other, run:  
`helloworldclient`

# Summary

---

- This chapter has shown how to develop a client and server using Celtix, starting from a WSDL contract.
- A code generation tool, `wsdl2java`, is used to generate a proxy interface and service class for the service.
- The proxy interface is extended to provide a service implementation object.
- Server code creates the service implementation object, registers it with an endpoint, and listens for requests.
- Client code creates a local service object, creates a proxy object, and uses it to invoke on the server.
- An appropriate code organisation strategy was presented.



# Getting Started with Annotations

Java 5.0

# Jax-WS Annotations

---

- JAX-WS relies on the annotation feature of Java 5.
- The JAX-WS annotations are used to specify the metadata used to map the SEI to a fully specified service definition. Among the information provided in the annotations are the following:
  1. The target namespace for the service.
  2. The name of the class used to hold the request message.
  3. The name of the class used to hold the response message.
  4. If an operation is a one way operation.
  5. The binding style the service uses.
  6. The name of the class used for any custom exceptions.
  7. The namespaces under which the types used by the service are defined.

# JAX-WS Annotations

---

- Most of the annotations have sensible defaults and do not need to be specified.
- However, the more information you provide in the annotations, the better defined your service definition.
- A solid service definition increases the likely hood that all parts of a distributed application will work together.



# Required Annotations

---

## The `@WebService` annotation

- You must add the `@WebService()` annotation on both the SEI and the implementation class
- The `@WebService` annotation is defined by the `javax.jws.WebService` interface and it is placed on an interface or a class that is intended to be used as a service.

# @Webservice Annotation Properties

---

- The following table describes the properties of the `@Webservice` annotation.

| Property                 | Description  |
|--------------------------|--|
| <b>name</b>              | Specifies the name of the service interface. This property is mapped to the name attribute of the <code>wsdl:portType</code> element that defines the service's interface in a WSDL contract. The default is to append <code>PortType</code> to the name of the implementation class.                                |
| <b>targetNamespace</b>   | Specifies the target namespace under which the service is defined. If this property is not specified, the target namespace is derived from the package name.   |
| <b>serviceName</b>       | Specifies the name of the published service. This property is mapped to the name attribute of the <code>wsdl:service</code> element that defines the published service. The default is to use the name of the service's implementation class.  |
| <b>wsdlLocation</b>      | Specifies the URI at which the service's WSDL contract is stored. The default is the URI at which the service is deployed.   |
| <b>endpointInterface</b> | Specifies the full name of the SEI that the implementation class implements. This property is only used when the attribute is used on a service implementation class.  |
| <b>portName</b>          | Specifies the name of the endpoint at which the service is published. This property is mapped to the name attribute of the <code>wsdl:port</code> element that specifies the endpoint details for a published service. The default is to append <code>Port</code> to the name of the service's implementation class. |

# Optional Annotations

---

## The `@SOAPBinding` annotation

- The `@SOAPBinding` annotation is defined by the `javax.jws.soap.SOAPBinding` interface. It provides details about the SOAP binding used by the service when it is deployed.
- If the `@SOAPBinding` annotation is not specified, a service is published using a wrapped doc/literal SOAP binding.
- You can put the `@SOAPBinding` annotation on the SEI and any of the SEI's methods.
  - When it is used on a method, setting of the method's `@SOAPBinding` annotation take precedent.

# The @SOAPBinding annotation

---

- The following table describes the properties of the @SOAPBinding annotation.

| Property       | Values   | Description   |
|----------------|--|---|
| style          | <code>Style.DOCUMENT</code><br><b>(default)</b><br><code>Style.RPC</code>                | Specifies the style of the SOAP message. If <code>RPC</code> style is specified, each message part within the SOAP body is a parameter or return value and will appear inside a wrapper element within the <code>soap:body</code> element. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. If <code>DOCUMENT</code> style is specified, the contents of the SOAP body must be a valid XML document, but its form is not as tightly constrained.   |
| use            | <code>Use.LITERAL</code> <b>(default)</b><br><code>Use.ENCODED</code>                    | Specifies how the data of the SOAP message is streamed.   |
| parameterStyle | <code>ParameterStyle.BARE</code><br><code>ParameterStyle.WRAPPED</code> <b>(default)</b> | Specifies how the method parameters, which correspond to message parts in a WSDL contract, are placed into the SOAP message body. A parameter style of <code>BARE</code> means that each parameter is placed into the message body as a child element of the message root. A parameter style of <code>WRAPPED</code> means that all of the input parameters are wrapped into a single element on a request message and that all of the output parameters are wrapped into a single element in the response message. If you set the style to <code>RPC</code> you must use the <code>WRAPPED</code> parameter style. |

## The `@WebMethod` annotation

---

- The `@WebMethod` annotation is defined by the `javax.jws.WebMethod` interface.
- It is placed on the methods in the SEI.
- The `@WebMethod` annotation provides the information that is normally represented in the `wsdl:operation` element describing the operation to which the method is associated.

# The `@Webmethod` annotation

---

- The following table describes the properties of the `@Webmethod` annotation.

| Property                   | Description   |
|----------------------------|---|
| <code>operationName</code> | Specifies the value of the associated <code>wsdl:operation</code> element's name. The default value is the name of the method.  |
| <code>action</code>        | Specifies the value of the <code>soapAction</code> attribute of the <code>soap:operation</code> element generated for the method. The default value is an empty string. |
| <code>exclude</code>       | Specifies if the method should be excluded from the service interface. The default is <code>false</code> .  |

# The `@RequestWrapper` annotation

---

- The `@RequestWrapper` annotation is defined by the `javax.xml.ws.RequestWrapper` interface.
- It is placed on the methods in the SEI.
- As the name implies, `@RequestWrapper` specifies the Java class that implements the wrapper bean for the method parameters that are included in the request message sent in a remote invocation.
- It is also used to specify the element names, and namespaces, used by the runtime when marshalling and unmarshalling the request messages

# The `@RequestWrapper` annotation

---

- The following table describes the properties of the `@RequestWrapper` annotation.

| Property                     | Description   |
|------------------------------|---|
| <code>localName</code>       | Specifies the local name of the wrapper element in the XML representation of the request message. The default value is the name of the method or the value of the <code>@WebMethod</code> annotation's <code>operationName</code> property. |
| <code>targetNamespace</code> | Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.   |
| <code>className</code>       | Specifies the full name of the Java class that implements the wrapper element.  |



# The `@ResponseWrapper` annotation

---

- The `@ResponseWrapper` annotation is defined by the `javax.xml.ws.ResponseWrapper` interface.
- It is placed on the methods in the SEI. As the name implies, `@ResponseWrapper` specifies the Java class that implements the wrapper bean for the method parameters that are included in the response message sent in a remote invocation.
- It is also used to specify the element names, and namespaces, used by the runtime when marshalling and unmarshalling the response messages.

# The @ResponseWrapper annotation

---

- The following table describes the properties of the `@ResponseWrapper` annotation.

| Property                     | Description  |
|------------------------------|--|
| <code>localName</code>       | Specifies the local name of the wrapper element in the XML representation of the response message. The default value is the name of the method with <code>Response</code> appended or the value of the <code>@WebMethod</code> annotation's <code>operationName</code> property with <code>Response</code> appended. |
| <code>targetNamespace</code> | Specifies the namespace under which the XML wrapper element is defined. The default value is the target namespace of the SEI.  |
| <code>className</code>       | Specifies the full name of the Java class that implements the wrapper element.   |

## The `@WebFault` annotation

---

- The `@WebFault` annotation is defined by the `javax.xml.ws.WebFault` interface.
- It is placed on methods in the SEI that throw exceptions.
- The `@WebFault` annotation is used to map the Java exception to a `wsdl:fault` element. This information is used to marshall the exceptions into a representation that can be processed by both the service and its consumers.

# The `@WebFault` annotation

---

- The following table describes the properties of the `@WebFault` annotation.

| Property               | Description   |
|------------------------|---|
| <b>name</b>            | Specifies the local name of the fault element.  |
| <b>targetNamespace</b> | Specifies the namespace under which the fault element is defined. The default value is the target namespace of the SEI. |
| <b>faultName</b>       | Specifies the full name of the Java class that implements the exception.  |

## The `@OneWay` annotation

---

- The `@OneWay` annotation is defined by the `javax.jws.OneWay` interface.
- It is placed on the methods in the SEI that will not require a response from the service.
- The `@OneWay` annotation tells the run time that it can optimize the execution of the method by not waiting for a response and not reserving any resources to process a response.

# Defining Parameters with Annotations

---

The method parameters in the SEI correspond to the `wsdl:message` elements and their `wsdl:part` elements.

JAX-WS provides annotations that allow you to describe the `wsdl:part` elements that are generated for the method parameters.

- **The `@WebParam` annotation**
- **The `@WebResult` annotation**

## The @WebParam annotation

---

- The @WebParam annotation is defined by the `javax.jws.WebParam` interface.
- It is placed on the parameters on the methods defined in the SEI.
- The @WebParam annotation allows you to specify the direction of the parameter, if the parameter will be placed in the SOAP header, and other properties of the generated `wsdl:part`.

# The @WebParam annotation

---

- The following table describes the properties of the @WebParam annotation

| Property        | Values   | Description   |
|-----------------|--|---|
| name            |  | Specifies the name of the parameter as it appears in the WSDL. For RPC bindings, this is name of the <code>wsdl:part</code> representing the parameter. For document bindings, this is the local name of the XML element representing the parameter. The default is to use the name of the parameter as it appears in the method argument list. |
| targetNamespace |  | Specifies the namespace for the parameter. It is only used with document bindings where the parameter maps to an XML element. The defaults is to use the service's namespace.   |
| mode            | <code>Mode.IN</code><br><b>(default)</b><br><code>Mode.OUT</code><br><code>Mode.INOUT</code> | Specifies the direction of the parameter.   |
| header          | <code>false</code> <b>(default)</b><br><code>true</code>                                     | Specifies if the parameter is passed as part of the SOAP header.  |
| partName        |  | Specifies the value of the name attribute of the <code>wsdl:part</code> element for the parameter when the binding is document.   |



## The `@WebResult` annotation

---

- The `@WebResult` annotation is defined by the `javax.jws.WebResult` interface.
- It is placed on the methods defined in the SEI.
- The `@WebResult` annotation allows you to specify the properties of the generated `wsdl:part` that is generated for the method's return value

# The @WebResult annotation

---

- The following table describes the properties of the @WebResult annotation

| Property        | Description  |
|-----------------|--|
| name            | Specifies the name of the return value as it appears in the WSDL. For RPC bindings, this is name of the <code>wsdl:part</code> representing the return value. For document bindings, this is the local name of the XML element representing the return value. The default value is <code>return</code> . |
| targetNamespace | Specifies the namespace for the return value. It is only used with document bindings where the return value maps to an XML element. The default is to use the service's namespace.   |
| header          | Specifies if the return value is passed as part of the SOAP header.  |
| partName        | Specifies the value of the <code>name</code> attribute of the <code>wsdl:part</code> element for the return value when the binding is document.  |

# Fully Annotated SEI

---

```
package org.apache.cxf;
import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;
@WebService(name="quoteReporter")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)public interface quoteReporter
{ @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.iona.com/types",
    className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.iona.com/types",
    className="org.eric.demo.Quote")
  @WebResult(targetNamespace="http://demo.iona.com/types",
    name="updatedQuote") public Quote getQuote(
    @WebParam(targetNamespace="http://demo.iona.com/types",
      name="stockTicker",
      mode=Mode.IN)
    String ticker );
}
```

# Summary

---

- This chapter has shown that JAX-WS annotations are used to specify the metadata used to map the SEI to a fully specified service definition
- Annotations play a critical role in JAX-WS 2.0.
- First, annotations are used in mapping Java to WSDL and schema.
- Second, annotations are used a runtime to control how the JAX-WS runtime processes and responds to web service invocations
- This makes it possible to use “Java to WSDL” approach as the primary way for developing Web services (for the server side) superceding JAX-RPC.
- Currently the annotations utilized by JAXR-WS 2.0 are defined in separate JSRs: 1) JSR 181: Web Services Metadata for the Java™ Platform, 2) JSR 222: Java™ Architecture for XML Binding (JAXB) 2.0, 3) JSR 224: Java™ API for XML Web Services (JAX-WS) 2.0, 4) JSR 250: Common Annotations for the Java™ Platform.



# Getting Started with Celtix

# Developing a Service using JAX-WS

---

You can develop a service using one of two approaches:

1. Start with a WSDL contract and generate Java objects to implement the service.
2. Start with a Java object and service enable it using annotations. For new development the preferred path is to design your services in WSDL and then generate the code to implement them. This approach enforces the concept that a service is an abstract entity that is implementation neutral. It also means you can spend more time working out the exact interface your service requires before you start coding.

## Existing Application Integration

---

**Caveat: To service enable an existing application**

1. You will need to add annotations to the source
2. You will need to migrate your code to Java 5.0.

# WSDL First Development

---

- Start with a WSDL document
  - can be obtained from another developer, a system architect, a UDDI registry, or you could write it yourself.
  - must contain at least a fully specified logical interface
- Three Step process follows:
  1. Generate starting point code.
  2. Implement the service's operations.
  3. Publish the implemented service



# JAX-WS Implementation Mapping

---

- JAX-WS specifies a detailed mapping from a service defined in WSDL to the Java classes that will implement that service.
  - The logical interface, defined by the `wsdl:portType` element, is mapped to a service endpoint interface (SEI).
  - Any complex types defined in the WSDL are mapped into Java classes following the mapping defined by the Java Architecture for XML Binding (JAXB) specification.
  - The endpoint defined by the `wsdl:service` element is also generated into a Java class that is used by consumers to access endpoints implementing the service.

# Generating the Starting Point Code

---

The **wsdl2java** command automates the generation of this code.

- Provides options for:
  - generating starting point code for your implementation
  - creating an ant based makefile to build the application
  - controlling the generated code.

**wsdl2java -ant -impl -server -d *outputDir* myService.wsdl**

- The command does the following:
  - The `-ant` argument generates a Ant makefile, called `build.xml`, for your application.
  - The `-impl` argument generates a shell implementation class for each portType element in the WSDL document.
  - The `-server` argument generates a simple `main()` to launch your service as a stand alone application
  - The `-d outputDir` argument tells **wsdl2java** to write the generated code to a directory called `outputDir`.
  - `myService.wsdl` is the WSDL document from which code is generated.

# Generated Files

---

| File                                   | Description   |
|--|---|
| <code>portTypeName.java</code>         | <b>The SEI.</b> This file contains the interface your service implements. You should not edit this file.      |
| <code>serviceName.java</code>          | <b>The endpoint.</b> This file contains the Java class your clients will use to make requests on the service. |
| <code>_portTypeName_Impl.java</code>   | <b>The skeleton implementation class.</b> You will modify this file to implement your service.                |
| <code>_portTypeName_Server.java</code> | <b>A basic server <code>main()</code></b> that allows you to deploy your service as a stand alone process.    |

# Implementing the service

---

## ■ Convention:

- Name of implementation class = `<name-of-interface>Impl`
- This is just a convention, and you're free to ignore it.

## ■ You may wish to explicitly implement the generated interface that corresponds to the WSDL `portType`.

- This is optional.

## ■ You should provide appropriate values for the `@WebService` annotation.

- You should explicitly identify the service and port that this class implements.
- See next slide.

## Implementing the service (cont')

---

```
@javax.jws.WebService(portName = "SoapPort",
    serviceName = "SOAPService", targetNamespace =
    "http://apache.org/hello_world_soap_http",
    endpointInterface =
    "org.apache.hello_world_soap_http.Greeter")
```

```
public class HelloWorldImpl implements HelloWorld
{
    public String sayHello(String message)
    {
        return "Hello right back at ya!";
    }
}
```

# Java First Development

---

To create a service starting from Java you need to do the following:

1. Create a Service Endpoint Interface (SEI) that defines the methods you wish to expose as a service.
  - You can work directly from a Java class, but working from an interface is the ***recommended*** approach. Interfaces are better for sharing with the developers who will be responsible for developing the applications consuming your service. The interface is smaller and does not provide any of the service's implementation details.
2. Add the required annotations to your code.
3. Generate the WSDL contract for your service.
  - **Tip**  
If you intend to use the SEI as the service's contract, it is not necessary to generate a WSDL contract
4. Publish the service.

# Creating an SEI

---

- The service endpoint interface (SEI) is the piece of Java code that is shared between a service and the consumers that make requests on it.
  - When starting with a WSDL contract, the SEI is generated by the code generators. However, when starting from Java, it is the up to a developer to create the SEI.
  - If you have an existing set of functionality that is implemented as a Java class and you want to service enable it, you will need to do two things:
    1. Create an SEI that contains **only** the operations that are going to be exposed as part of the service.
    2. Modify the existing Java class so that it implements the SEI.
- **Note**
- You can add the JAX-WS annotations to a Java class, but that is not recommended.

# Simple Standard Interface (SEI)

---

```
package org.apache.cxf;  
public interface quoteReporter  
{ public Quote getQuote(String ticker);  
}
```

In order to create a service from Java code you are only **REQUIRED** to add one annotation to your code. You must add the `@WebService()` annotation on both the SEI and the implementation class.



# Annotated SEI

---

```
package com.iona.demo;

import javax.jws.*;

@WebService(name="quoteUpdater",
    targetNamespace="http://cxf.apache.org",
    serviceName="updateQuoteService",
    wsdlLocation="http://cxf.apache.org/quoteExampleService?wsdl",
    portName="updateQuotePort")

public interface quoteReporter
{
    public Quote getQuote(String ticker);
}
```

## Annotating the service implementation

---

- In addition to annotating the SEI with the `@WebService` annotation, you also have to annotate the service implementation class with the `@WebService` annotation.
- When adding the annotation to the service implementation class you only need to specify the `endpointInterface` property. set to the full name of the SEI.

```
package org.apache.cxf;  
  
import javax.jws.*;  
  
@WebService(endpointInterface="org.apache.cxf.quote  
Reporter")  
  
public class stockQuoteReporter implements  
quoteReporter  
{public Quote getQuote(String ticker) { ... }}
```

# Generate WSDL file

---

- Once you have annotated your code, you can generate a WSDL contract for your service using the **java2wsdl** command.
- Example generated WSDL file

```
.....  
<xs:complexType name="quote">  
  <xs:sequence>  
    <xs:element name="ID" type="xs:string" minOccurs="0"/>  
    <xs:element name="time" type="xs:string" minOccurs="0"/>  
    <xs:element name="val" type="xs:float"/>  
  </xs:sequence>  
</xs:complexType>  
</xsd:schema>  
</wsdl:types>  
  <wsdl:message name="getStockQuote">  <wsdl:part name="stockTicker"  
    type="xsd:string">  </wsdl:part> </wsdl:message>  
<wsdl:message name="getStockQuoteResponse">  <wsdl:part name="updatedQuote"  
    type="tns:quote">  </wsdl:part> </wsdl:message>
```

# Summary

---

- This chapter has shown how to develop a client and server using Celtix, starting from a WSDL contract.
- A code generation tool, `wsdl2java`, is used to generate a proxy interface and service class for the service.
- The proxy interface is extended to provide a service implementation object.
- Server code creates the service implementation object, registers it with an endpoint, and listens for requests.
- Client code creates a local service object, creates a proxy object, and uses it to invoke on the server.
- An appropriate code organization strategy was presented.



# Web Services Security

# Introduction

---

- A full description of web services security is beyond the scope of this material.
  - Recommended reading:
    - OASIS Web Services Security (WSS) Technical Committee – <http://www.oasis-open.org>
    - WS-I Basic Security Profile – <http://www.ws-i.org>
- No overall web services security architecture has yet been defined.
  - Such an architecture may include firewalls, proxies, security servers, and identity management.
- WS-Security (from OASIS) is the accepted framework
- This section shows how web services can be made secure using a combination of SSL/TLS (HTTPS) and WS-Security.
- Other specifications are discussed.

# What is security?

---

- Here are some general concepts of “secure” communications, and what they mean to users...
- *Confidentiality:* We want to exclude uninvited listeners from our private communications
  - e.g. sending credit card numbers or social identity
- *Authenticity:* We want to verify the identity of each person we talk to, so that we do not end up talking to an impostor
- *Integrity:* We want to protect the content of our conversation, so that nobody can change it mid-stream, thereby causing confusion or misunderstandings

# What is security? (cont')

---

- Other optional concepts...
- *Access Control*: Some people have more privileges than others, allowing them to do things others cannot
  - e.g. a bank manager can audit any teller's transactions, but a teller cannot even view the manager's work
- *Circle of Trust*: Members of a group can share tasks, by passing around the security credentials required to perform these tasks
- *Revocation*: When people change jobs, or lose our trust somehow, we must be able to exclude them from conversations where they were welcome before



# Web service attacks

---

- Web services are prone to a number of different kinds of attack; for example:
  - *Snooping* - as a non-binary plain-text protocol, SOAP messages are easy to intercept and view.
  - *Man-in-the-middle attacks* - messages can be easily intercepted, modified and resent by malicious parties.
  - *Brute-force attacks* - a malicious party can attempt to crack username/passwords with a brute force approach.
    - This is tedious for browser-based web-sites, but can be trivially automated for web services.
  - *Denial of service (DOS) attacks* - a malicious party can “hammer” your server with large or incorrectly formed XML messages, in an attempt to bring down your server or at least impair performance
  - *Data attacks* - a malicious party might guess web service operations parameters that affect your program logic and exploit this (see next slide).

# Web service attacks (cont')

---

- SQL injection is an clever (and dangerous) form of data attack.
  - The malicious party sees a method that requires a username, for example, "scott".
  - The malicious party guesses that you will run an SQL statement of the form:

```
select * from users where username='scott';
```
  - They enter the following as a username:

```
xyz'; drop table users;
```
  - This results in the following SQL statements

```
select * from users where username='xyz'; drop table users;
```
  - As a result, the select statement gives no result, and the user table is dropped (removed!) from the database.
- This kind of attack is a motivating example: clever hackers will exploit any weakness in your application.

# Defend in depth; monitor for trouble

---

- In this session, we will show how to use SSL and WS-Security to guard against snooping and man-in-the-middle attacks.
  - These techniques provide confidentiality, message integrity, authentication and authorisation.
- These techniques may form only *part* of your overall security solution.
  - They cannot protect against brute-force attacks
  - They cannot protect against denial-of-service attacks
- You will need to use alternative techniques to protect against DOS and brute-force attacks.
  - Discuss with your IT department or Internet Service Provider on tools to detect and protect from these attacks.
- Adopt the philosophy: “Defend in depth; monitor for trouble”.

# Cryptography and Public Key Infrastructure (PKI)

# Cryptography and PKI

---

- Cryptology can be termed the mathematics of secrecy; cryptography is the algorithmic application of those mathematics
  - Algorithms rely on keys (large constant numbers) to establish privacy
  - Keys can be exchanged by parties who desire a private conversation
  - Symmetric Algorithms – parties share keys
    - Sharing is easy, but managing the shared keys is problematic
  - Asymmetric Algorithms – parties use public and private keys as a pair in a specific secret conversation
    - Slower, but key management becomes easier
  
- Public Key Infrastructure (PKI) supports...
  - Key Exchange – asymmetric public key algorithms, used to authenticate one or both parties and derive a symmetric key
  - Encryption – symmetric private key algorithms, used to encrypt/decrypt messages and thus ensure confidentiality
  - Hashing – algorithms for generating digital signatures, used to ensure message integrity

# Cryptography and PKI (cont')

---

## ■ Key exchange algorithms

- Most popular option is Rivest Shamir Adleman (RSA) public key encryption using X.509v3 certificates

## ■ Symmetric key encryption algorithms

- Data Encryption Standard (DES) with 56-bit key

## ■ Secure hash algorithms

- SHA or Message Digest 5 (MD5)

## ■ Cipher suites are collections of PKI algorithms

- For example, `RSA_WITH_RC4_128_MD5` denotes a cipher suite with...
  - RSA public key exchange
  - RC4 symmetric key encryption with 128-bit key
  - MD5 hash digest signatures
- Different combinations offer trade-offs between performance and strength

# Cryptography and PKI (cont')

---

## ■ Security certificates (X.509)

- Used by clients and servers to encapsulate their keys
- X.509v3 format is the standard

## ■ Certificate Authority (CA)

- Essentially a trusted 3<sup>rd</sup> party who vouches the identity of a user (a X.500 DN) and the binding of that user to a public key

## ■ Certificate Revocation List (CRL)

- Serial numbers for certificates that should be held or rejected

## ■ PEM versus PKCS#12

- Portable file formats for sending security credentials
- PEM is a base64-encoded text file, usually containing certificates or a CRL
- PKCS#12 is a binary file, usually containing certificates and a private key

# Web Services and SSL



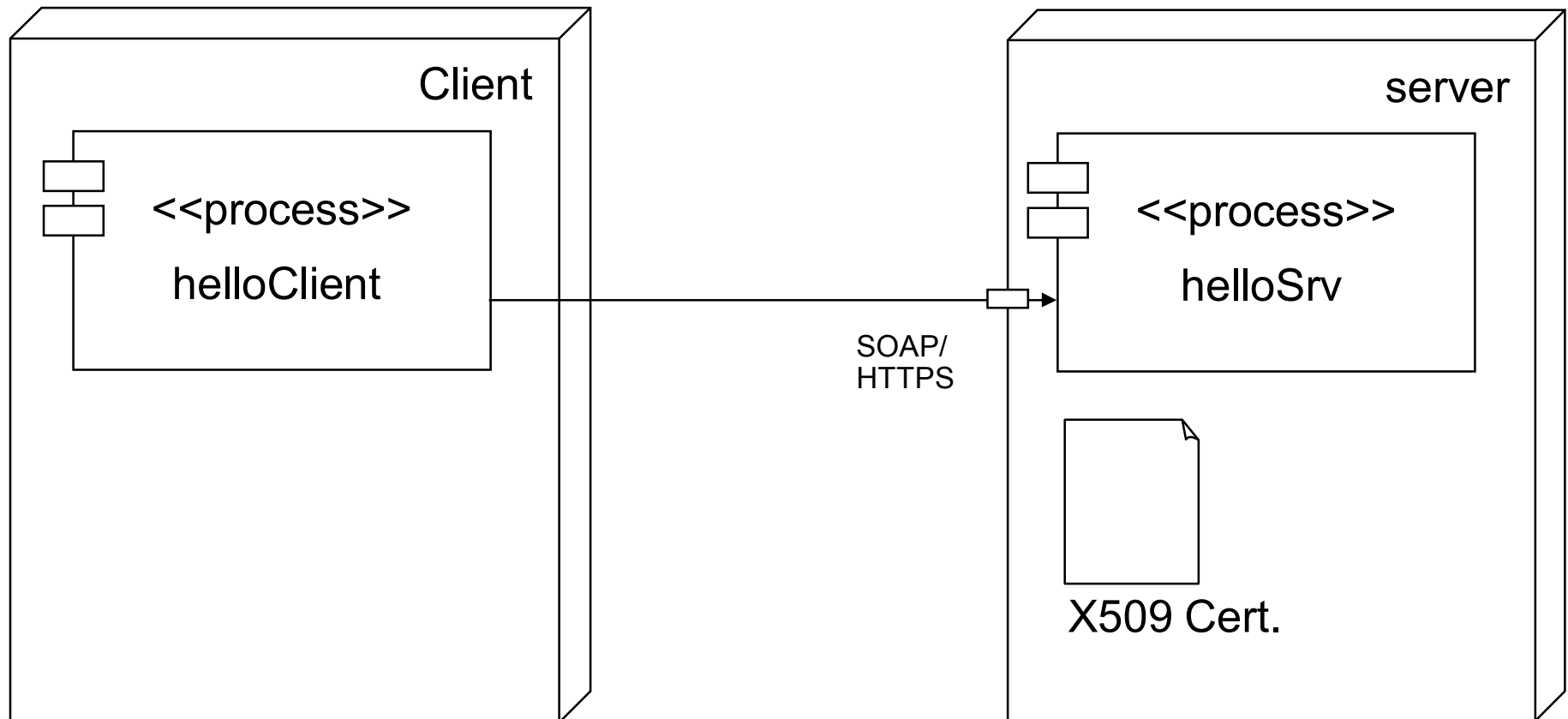
# Web services and SSL

---

- For most users, web services security will begin with encryption of the transport layer: that is, through the use of HTTPS rather than HTTP.
- HTTPS makes use of SSL/TLS (Secure Sockets Layer / Transport Layer Security), commonly abbreviated to just SSL.
  - TLS is the successor to SSL v3.0
  - SSL/TLS provides:
    - Client- and server-side authentication with X.509 certificates.
    - Confidentiality, Integrity, Protection against message-replay
- SSL/TLS often provides a key line-of-defence in web services security.
  - Due to the use of strong encryption, hackers cannot intercept, snoop, or replay messages.
  - However, SSL/TLS provides only limited support for authorisation.

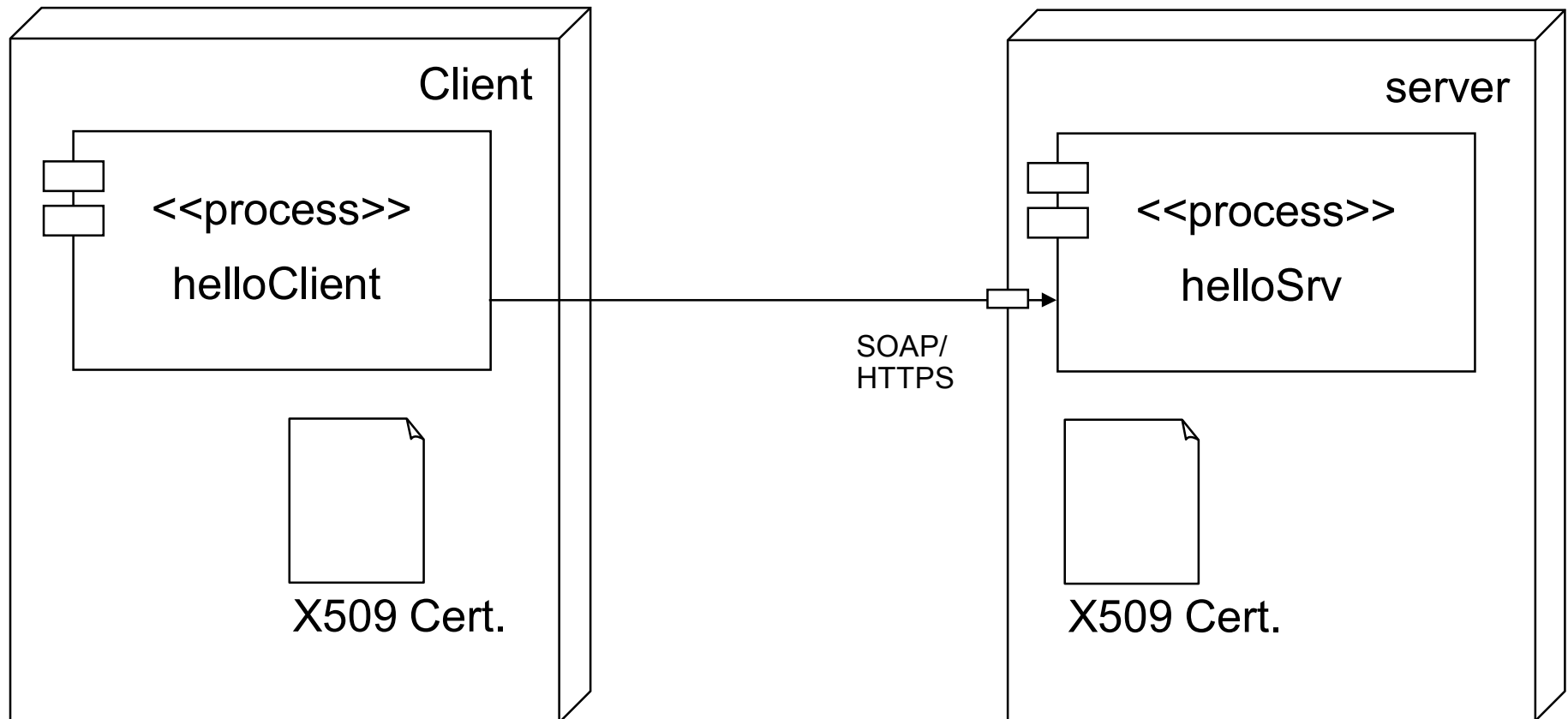
# Web services and SSL (cont')

- Clients communicate using HTTPS: they will only communicate with a server that provides a trusted certificate.
  - This is called “server-side” authentication.



# Web services and SSL (cont')

- Servers can authenticate clients too: a server can be configured to accept only trusted certificates.
  - Further, the server can be configured to trust only certificates with particular attributes; e.g. "department=finance".



# Some myths about SSL

---

## ■ Myth #1 – SSL is *always* slower

- SSL handshake makes the first request slower
- Overhead for encryption/decryption of data thereafter is negligible
- Overheads depend on which cipher suite is selected

## ■ Myth #2 – SSL can be cracked

- True for original 40-bit keys
  - Can be cracked using brute-force, parallel computing attacks
- Not true for new 128-bit keys
  - Non-exportable from US
  - Modulus factorization for  $2^{128}$  (or  $3.402823669209385e+38$ )
  - Joe Hacker can give up starting right now
  - Time for alien technology or major advances in quantum computing
    - Quantum Cryptography group in Los Alamos, NM have performed key en/decryption using photons!!!!

# Authentication and Authorisation

# Authentication and authorisation

---

- SSL/TLS provides limited support for “authorisation”
  - Authorisation is that part of your security infrastructure that decides who is allowed do what.
- Authorisation typically involves sending some kind of “credential” token with each message
  - A username/password pair
  - An X.509 certificate
  - A Kerberos token.
- Credentials can be transmitted to the server in a number of ways.
  - HTTP-Basic Authentication can transmit username/password
  - WSSE Header (i.e. an element in a SOAP header) can transmit username/password, client’s X.509 certificate or other token.
  - Client’s X.509 certificate can be read from the secure transport.

# A note on header information

---

- Most transports (HTTP included) provide their own mechanisms for transmission of header-information.
  - Security information can be sent using the HTTP standard headers.
- SOAP messages themselves also contain a SOAP-header.
  - Security information can *also* be sent using the HTTP standard headers.

HTTP Header

*Can put authentication information here.*

```
<?xml version="1.0"?>
<soap:Envelope>
  <soap:Header>... or here ...</soap:Header>
  <soap:Body></soap:Body>
</soap:Envelope>
```

# A note on header information (cont')

---

- When authentication information (username/password, or security token) is sent in the transport header, this is called “message-level” authentication.
  - Message-level authentication provides an advantage in that the SOAP message does not have to be parsed in order to extract security information.
- When authentication information is sent in the SOAP header this is called “request-level” authentication.
  - Request-level authentication ensures that authentication information pertinent to this message is stored with the message, regardless of any transport switching or persistence.
- The choice of message- or request-level authentication is up to you.
  - It is wise to support both kinds of access.



# HTTP-Basic message-level authentication

---

- The HTTP transport supports “basic authentication” where the username/password is transmitted as a base64 encoded string in the request header.

```
GET /path/file.html HTTP/1.0
```

```
Authorization: Basic <username:password>
```

- The use of the base-64 encoding obfuscates the username/password information.
  - *Obfuscate*: to conceal, confuse, muddle
- Obfuscation is not the same as encryption!
  - Anyone with a knowledge of the base-64 algorithm can decode the username and password

# HTTP Basic Authentication (cont')

---

- HTTP-Basic authentication should *only* be used over HTTPS
  - Security credentials will then be properly encrypted
- All major SOAP toolkits provide a way to programmatically set the username and password

# WSSE request-level authentication

---

- The WS-Security specification from OASIS defines a standardised way to transmit credentials as a SOAP header.

```
<SOAP-ENV:Header>
  <wsse:Security
    xmlns:wsse="http://docs.oasis-open.org/wss/...">
    <wsse:UsernameToken>
      <wsse:Username>scott</wsse:Username>
      <wsse:Password>tiger</wsse:Password>
    </wsse:UsernameToken>
  </wsse:Security>
</SOAP-ENV:Header>
```

- WSSE (Web Services Security Extensions) headers are supported by all major vendors.
- As the information is transmitted in plain-text, this should *only* be used with SSL/TLS (i.e. HTTPS).

# WS-Security

---

- **WS-Security has 3 major components**
  - Security tokens – username/password, X.509, Kerberos
  - XML Encryption – reference list for encrypted SOAP body parts
  - XML Signatures – reference list for digital signing of SOAP body parts
- **Other profiles for SOAP Attachments (SwA), Rights Expression Language (REL) tokens, and more...**
- **WS-Security is just part of the overall security framework**
  - A foundation for WS-Policy, WS-Trust, WS-Privacy, and other higher-level WS specifications

# Using credentials for role-based access

---

- When the messages arrives at the server, the credentials can be accessed explicitly.
  - Application code can then determine the roles assigned to the caller, and thus whether the caller is allowed to invoke on the service or operation.
  
- Some SOAP toolkits can provide support for user- or role-based access out-of-the-box.
  - Authorisation and authentication is turned on via configuration, and requires little-or-no coding.
    - Access Control Lists (ACLs) are used to configure access.
  - Toolkits will generally provide adapters to popular security systems, or may provide an adapter framework that allows you to write your own custom adapters.

# Relevant Security Technologies and Standards

# Relevant technologies and standards

---

- There is a long shopping-list of technologies and standards that are relevant to web services security.
  - These standards are complimentary and are not competing.
- SSL/TLS encryption - IETF (Internet Engineering Task Force).
  - Often used on Web (HTTPS)
  - Utilises public-key cryptographic systems (PKCS) for authentication and encryption.
- XML Signature - W3C
  - XML Signature allows an XML document (or its parts) to be signed to enforce non-repudiation.
  - Web services messages are XML documents; as such XML Signature
- XML Key Management Specification (XKMS) - W3C
  - Allows a client to obtain key information (values, certificates, management or trust data) from a web service.

# Relevant technologies and standards (cont')

---

## ■ XML Encryption - W3C

- XML Signature allows an XML document (or its parts) to be encrypted.

## ■ WS-Security - OASIS

- As we have seen, WS-Security is a framework for including security information in SOAP headers.
- It supports authentication tokens such as X.509 and Kerberos, and simple username/password tokens.
- Requesters and providers need to agree on token format – otherwise no interoperability

## ■ WS-Identity - W3C

- Addresses the issue of tracking an individual's identity (username and password) in a global framework.
- Liberty Alliance Project is a major player ([www.libertyproject.org](http://www.libertyproject.org))
- Microsoft Passport also competed in this space, but has been abandoned.



# Relevant technologies and standards (cont')

---

- **Access Control Markup Language (XACML) - OASIS**
  - Used to represent and evaluate access control policies for role-based authorization
- **Security Assertion Markup Language (SAML) - OASIS**
  - Defining and maintaining a standard, XML-based framework for creating and exchanging security information between online partners

# Summary

---

- Security is an important consideration in any web services project.
- Secure transports like HTTPS provide on-the-wire encryption and authentication.
- WS-Security Extensions (WSSE) provides standardised SOAP headers to carry authentication information.
  - For example, username/password, kerberos ticket, or security token.
- Together, secure transports and WSSE can support message integrity, privacy, authentication and authorisation.
  - However, they cannot provide automatic protection against weaknesses in your design or implementation.
- Adopt the policy: *defend in depth; monitor for trouble.*



# **Celtix Client-side Programming**

# Overview

---

- Much of Celtix/CXF client-side programming will use the API's generated by `wSDL2java`.
- The general approach is:
  - Create a *service object*
    - Note: a service object is a client-side representation of a remote service.
  - Use the service object to get a proxy
  - Invoke on the proxy
- We have seen this approach in the “Celtix Development” chapter.
- This chapter deals with other client-side issues.
  - Different ways to get a proxy from the service object.
  - How to override a service endpoint address using a request context.
  - How to access the Celtix `BUS` object.

Obtaining a proxy from the service object.

# Static and dynamic port creation

---

- There are two ways to obtain a proxy object from the Service object.
  - Static: use the generated port creation methods
  - Dynamic: use the generic `getPort()` method from the class `javax.xml.ws.Service`.
- You can use whichever you prefer.
  - The static approach is less code, and more intuitive to beginners.
  - The dynamic approach is more flexible, as the endpoint details are not hard-coded into your code-base.
    - Using the dynamic approach you can defer the choice of endpoint to runtime configuration.
  - The dynamic approach is advocated in this course (and is the approach shown earlier in the “Hello, World” example).

# Static port creation

---

- The JAX-WS mapping generates a static port creation method for each port in your service.
  - If a service contains ports called "SOAPOverHTTPEndpoint" and "SOAPOverJMSEndpoint" then the methods `getSOAPOverHTTPEndpoint()` and `getSOAPOverJMSEndpoint()` will be generated on the client-side service object.
  - You can use these statically generated stubs to get service proxy objects.

## ■ Example:

```
HelloWorldService helloWorldService =  
    new HelloWorldService(wsdlURL, serviceName);  
  
HelloWorldI helloWorld =  
    helloWorldService.getSOAPOverHTTPEndpoint();
```

# Dynamic port creation

---

- The dynamic approach uses the `getPort()` method to create a client-side proxy.
  - `getPort()` requires two parameters:
    - A `QName` that identifies the port.
    - The endpoint interface class.

## ■ Example:

```
HelloWorldService helloWorldService =
    new HelloWorldService(wsdlURL, serviceName);
HelloWorldI helloWorld =
    helloWorldService.getPort(
        new QName("http://www.iona.com/ps/courseware/HelloWorld",
            "SOAPOverHTTPEndpoint"),
        HelloWorldI.class);
```



# Overriding a service endpoint address

# Overriding a service endpoint address

---

- A client reads a service's endpoint address from the WSDL file.

```
<service name="HelloWorldService">
  <port binding="tns:HelloWorld_DocLiteral_SOAPBinding"
    name="SOAPOverHTTPEndpoint">
    <soap:address location="http://localhost:9090/helloworld"/>
  </port>
</service>
```

- This is often undesirable.
  - What if the service has been deployed somewhere else?
  - How do we cater for unit-test, system-test, production- and live versions of the service?
- It is common practice to override the endpoint address in client code, using the `BindingProvider` API.

# Interface `javax.xml.ws.BindingProvider`

---

- The JAX-WS specification defines a `BindingProvider` interface.
  - Every client-side service proxy implements this interface.
- `BindingProvider` provides methods that operate on the underlying implementation of the service proxy, for example:
  - `getRequestContext()`
  - `getResponseContext()`
  - `getBinding()`
- The “request context” is a `Map<String, Object>`.
  - When you invoke on a proxy, all properties in the request context are copied into the message’s `MessageContext`.
- The request context determines how the message is handled.
  - One use is to override the default endpoint address.

# Obtaining the request context

---

## ■ To obtain the request context:

- Cast the service proxy to `BindingProvider`
- Invoke the `getRequestContext()` method
- Store the results in a `Map<String, Object>`.

## ■ For example:

```
HelloWorldService helloWorldService
    = new HelloWorldService(wsdlURL, serviceName);

HelloWorld helloWorld
    = helloWorldService.getSOAPOverHTTPPort();

Map<String, Object> requestContext =
    ((BindingProvider) helloWorld).getRequestContext();
```

# Using the request context to specify the endpoint

---

- The JAX-WS specification has reserved a key in the request context for specifying the endpoint address.
  - `"javax.xml.ws.service.endpoint.address"`
- This key is stored as a public static final String in the class `BindingProvider` under the name `ENDPOINT_ADDRESS_PROPERTY`.
- Use the `Map.put()` method to specify an endpoint address:

```
requestContext.put(  
    BindingProvider.ENDPOINT_ADDRESS_PROPERTY,  
    "http://localhost:4321/helloworld"  
);
```

# The CXF Bus

# The CXF Bus

---

- At runtime, CXF creates a default *Bus* object.
  - The `Bus` provides the underlying communication infrastructure used by proxy objects.
  - The `Bus` class is CXF-specific: is *not* part of the JAX-WS specification.
- Most of the time you will *not* need to access the `Bus`.
  - The `Bus` is used *transparently* by service proxies.
- You can use the `Bus` to access to CXF implementation details:
  - Configuration, interceptors, etc.
  - These aspects of CXF are discussed elsewhere.
  - The next slides show how to access the bus in the first instance.

# Getting an `org.apache.cxf.Bus`

---

- To get access to the `Bus`, use `BusFactory.getDefaultBus()`.
  - It returns the default `Bus` created by CXF.
- `BusFactory` is an interface.
  - It is implemented by concrete implementations such as:
    - `org.apache.cxf.bus.cxf.CXFBusFactory`; and,
    - `org.apache.cxf.bus.spring.CXFBusFactory`.
  - Use the `BusFactoryHelper` to get a concrete instance of the `Bus`.

## ■ Sample usage:

```
import org.apache.cxf.Bus;
```

```
Bus bus = BusFactoryHelper.newInstance().getDefaultBus();
```



# Creating an `org.apache.cxf.Bus`

---

- You can create a `Bus` using `BusFactory.createBus()`
  - You can then override the default bus using the `setDefaultBus()` method.

## Aside: using the Bus

---

- Most applications will *not* need to create their own `Bus`.
- This material has been presented for completeness.
  - To give an insight into the underlying CXF implementation.
  - So that more advanced users will know how to access the `Bus`.
- Keep in mind that the `Bus` class is specific to CXF – it is *not* part of the JAX-WS specification.
  - Any code you write that uses the `Bus` will not be portable to other JAX-WS implementations.

# Summary

---

- Client-side programming in Celtix/CXF is largely a matter of using the API's generated from the WSDL contract.
- This chapter has shown:
  - how to create service proxies using the static and dynamic approach;
  - how to override the default endpoint address (as per the WSDL contract) in a JAX-WS compliant way; and
  - how to access the CXF `Bus`, which can be used to access CXF implementation details.



# **Celtix/CXF Server-side Programming**

# Overview

---

- Implementing Celtix/CXF servers typically involves two classes:
  - An *implementation* class, providing service functionality. The implementation class implements the service endpoint interface.
  - A server mainline class, providing a `main()` that creates and publishes endpoints.
- We have seen this approach in the chapter on “Celtix Development”
- In this session, we show how to:
  - Create and publish endpoints using `javax.xml.ws.Endpoint`.
  - Specify single- or multi-threaded servant access.
  - Annotate a service implementation class.
  - Access a `MessageContext` in the implementation class.

# Creating and publishing endpoints

# Creating and publishing an Endpoint

---

- The server mainline creates implementation objects, and then “publishes” them as `Endpoint` objects.
  - To create an endpoint, use `Endpoint.create(Object impl)`
  - An `Endpoint` only begins to listen when it is *published*.
- For example:

```
Endpoint hwep
    = Endpoint.create(new HelloWorldImpl());
hwep.publish("http://localhost:9090/helloworld");
```

# Creating and publishing an Endpoint (cont')

---

- These two steps can be condensed by using the static version of `Endpoint.publish()`
- For example:

```
Endpoint hwep = Endpoint.publish(  
    "http://localhost:9090/helloworld",  
    new HelloWorldImpl()  
);
```



# Endpoint methods

---

- The `Endpoint` class provides a number of useful methods.
- `Endpoint.stop()`
  - Stop listening for events; once stopped, an endpoint *cannot* be restarted.
  - To “restart”, you can create and publish a new `Endpoint` for the implementation object.
- `Endpoint.isPublished()`
  - Returns true if the `Endpoint` has been published.
- `Endpoint.getBinding()`
  - Returns the `Binding` for the `Endpoint`; this is useful if you wish to add handlers to the `Endpoint`.
- `Endpoint.getProperties()`
  - Returns a `Map<String, Object>` containing `Endpoint` properties.

# Server-side threading issues

# Server-side threading

---

- The JAX-WS specification mandates that service endpoints should be designed for concurrent access.
  - By default, endpoints are multi-threaded.
- If you need to synchronize access to critical sections of code then there are two possibilities:
  - Use the Java `synchronize` keyword to serialize access to critical code.
  - Change the endpoint's `Executor` to be single-threaded.
- An endpoint's `Executor` can be set when the service is published, as follows:

```
Endpoint ep = Endpoint.publish(address, implementor);  
ep.setExecutor(Executors.createSingleThreadedExecutor());
```

## Server-side threading (cont')

---

- You can write your own `Executor` class to implement more complex threading algorithms.
- Advice: remember that your servant may be deployed in a servlet engine or application server.
  - Ensure that if single-threaded access is required then the appropriate single-threaded `Executor` is loaded.
    - How this is specified may differ from container to container.
  - For maximum portability, mark all critical sections of code as synchronized using the `synchronize` keyword.

# Annotating a Service Implementation

# Inheritance of annotations

---

- A service implementation class typically implements the generated service endpoint interface.

```
public class HelloWorldImpl implements HelloWorld
{
}
```

- As a result, the implementation class (`HelloWorldImpl`) inherits the annotations from the service proxy interface (`HelloWorld`).
- The inherited annotations can be inappropriate.
  - In particular, the `@WebService.wsdlLocation` annotation may be a hard-coded *absolute* filename.
  - This makes your code inflexible!

## Note on `@WebService.wsdlLocation`

---

- Annotations are set at compile-time; they cannot be changed programmatically at run-time.
- The `wsdlLocation` used by your implementation object *must* provide a valid WSDL contract at server startup.
- A reasonable approach is to specify `wsdlLocation` values relative to the project root directory – see next slide.

# Overriding the @WebService annotation

---

- Below we explicitly declare the @WebService annotation:

```
@WebService (  
    name = "HelloWorld",  
    targetNamespace =  
        "http://www.celtix.org/courseware/HelloWorld",  
    wsdlLocation = "./wsdl/HelloWorld.wsdl"  
)  
public class HelloWorldImpl implements HelloWorld  
{  
}
```

- @WebService.wsdlLocation is now relative to the project root directory, where we start up our server.



## Note: resolving @WebService.wsdlLocation

---

- A `wsdlLocation` like `“./wsdl/HelloWorld.wsdl”` is more flexible than an absolute path.
- However, it is not ideal.
  - What if the server is started from a directory other than the project root?
- We could instead use a URL.
  - However, then the endpoint can only be created if a network connection is present and the URL can be resolved
- In the future, Celtix will provide an XML “catalog facility”
  - It will map remote URL locations to local files, allowing the use of URLs in `wsdlLocation` without the need for a network connection.

# Using the `MessageContext`

# javax.xml.ws.handler.MessageContext

---

- The `MessageContext` is a server-side object.
- `MessageContext` contains useful information about the currently executing operation.
  - For example, HTTP request headers, WSDL operation name, etc.
  - Customized message handlers can also use the `MessageContext` to pass information to the service implementation.
- The current `MessageContext` can be obtained from the class `javax.xml.ws.WebServiceContext`.
- The `WebServiceContext` is declared in your implementation class using the Java 1.5 *resource injection* approach.
  - The next slides give a brief overview of *resource injection*.

## Aside: resource injection

---

- It is quite common for classes to make use of references to well-known resources.
  - Typically, your code must declare an instance of the object, and then initialize the object.
- As an illustrative example:

```
public class CustomerInformation
{
    private BackEndDataObject bedo;
    CustomerInformation() {
        bedo = new BackEndDataObject.instance();
    }
}
```

## Aside: resource injection (cont')

---

- JSR 250, implemented in Java 1.5, allows you to *inject* resources.
  - Resource injection allows you to delegate the responsibility for creating certain dependent objects to someone else, typically an underlying container.

```
public class CustomerInformation
{
    @Resource
    private BackEndDataObject bedo;

    // No need to instantiate bedo, it will be done for
    // us automatically by the container!
}
```

- Injected resources are used in the same way as any other member variable.

# Declaring WebServiceContext as a resource

---

- Recall: we need to create a `WebServiceContext`; it will then be used to obtain the `MessageContext`.
- The `WebServiceContext` can be declared using resource injection as follows:

```
public class HelloWorldImpl implements HelloWorld
{
    @Resource
    private WebServiceContext context;
}
```

# Obtaining the MessageContext

---

- Use the `WebServiceContext.getMessageContext()` method as follows:

```
public String sayHello(String message)
{
    MessageContext messageContext =
        context.getMessageContext() ;
}
```

- The `MessageContext` **extends** `java.util.Map`.
  - Use the `get()` method to extract values from the `MessageContext`.

# Using the MessageContext

---

- The `String` constants in the `MessageContext` class can be used as keys when calling `MessageContext.get()`.
- For example, the following code obtains the HTTP request headers for the current message.

```
public String sayHello(String message)
{
    MessageContext messageContext =
        context.getMessageContext();

    Map httpRequestHeaders = (Map)
        messageContext.get(MessageContext.HTTP_REQUEST_HEADERS);
}
```



# Summary

---

- Server-side programming with Celtix/CXF can be characterized as:
  - Implement the service endpoint interface
  - Write a server mainline to create and publish endpoints
- This session has shown how to:
  - Use the `Endpoint` class to `create()`, `publish()` and `stop()` endpoints.
  - Provide suitable `@WebService` annotations for your implementation class.
  - Access the current `MessageContext` from within your implementation class.