



Building
Multi-Protocol
Open Source
Web Services
using JAX-WS and
Celtix Enterprise

adrian.trenaman@iona.com



Making Software Work Together™

Introduction

- This presentation shows how to build *multi-protocol, open-source, web services* using JAX-WS.
 - *Web Services*: service contracts are defined using WSDL
 - *Multi-protocol*: services support payloads like SOAP, XML, and JSON over transports like HTTP, JMS and AMQP
 - *Open Source*: solutions built using Celtix Enterprise, an open-source ESB that includes Apache CXF, Apache QPid and Eclipse STP.
 - *JAX-WS*: services implemented in Java using JAX-WS (Java API for XML Web Services).
- Discussion is motivated by a real-world use-case: transmission of UBL 2.0 Invoice documents.
 - Universal Business Language – OASIS standard for XML business documents: purchase orders, audit trail, invoicing, etc.



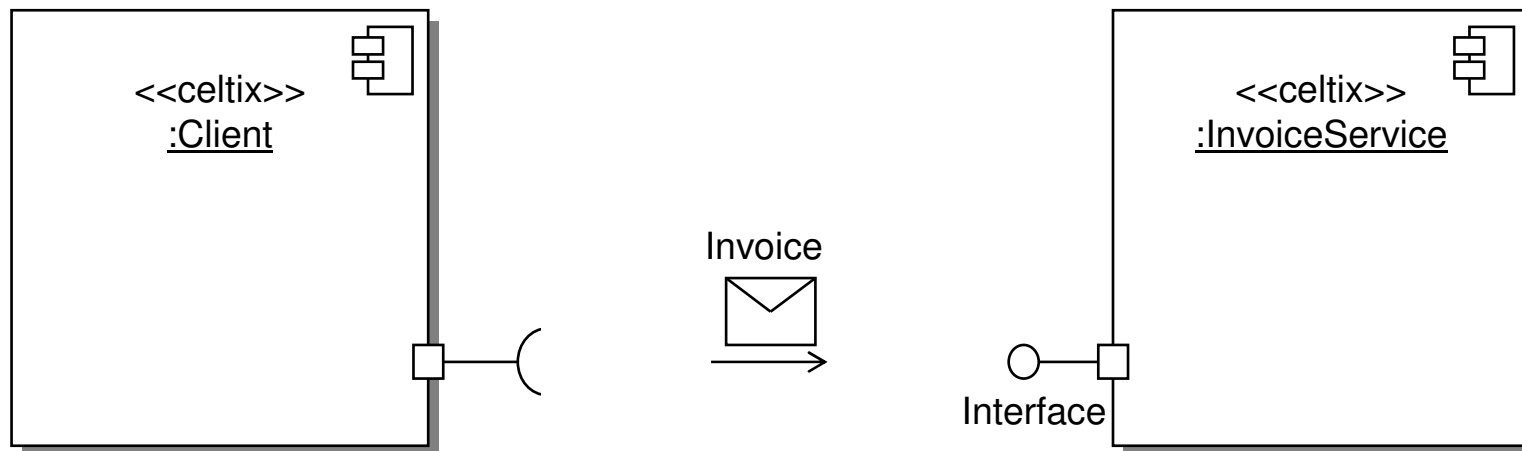
Motivation

- Most middleware allows you to separate interface from implementation.
 - However, in doing so it typically ties you to the middleware: CORBA, DCOM, .Net, JMS, etc..
- Using Celtix, you can separate interface from implementation *from middleware*.
 - The same programming model (JAX-WS) is used regardless of the underlying transport and protocol
 - Developers don't need specialist middleware skills, and can focus on business logic instead of integration logic.
 - New protocols can be plugged in with ease.



Motivating example: invoice processor

- Client sends a UBL Invoice in a one-way interaction.
 - The UBL Invoice type is non-trivial XML.
- We will use Celtix to show how to send the document using different payloads (SOAP, raw XML, JSON) and different transports (HTTP(S), JMS, AMQP).



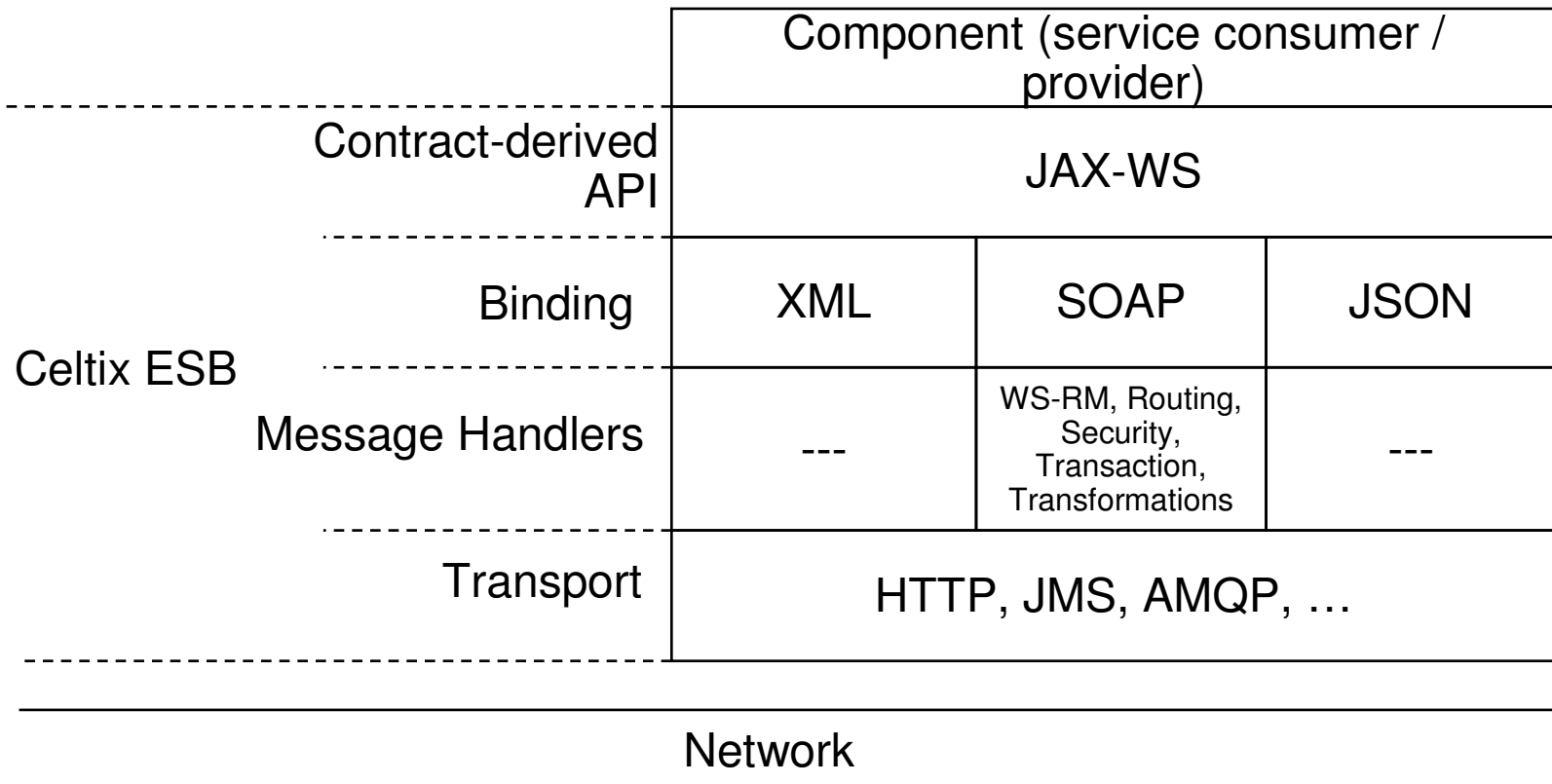
What does Multi-Protocol Mean?



Making Software Work Together™

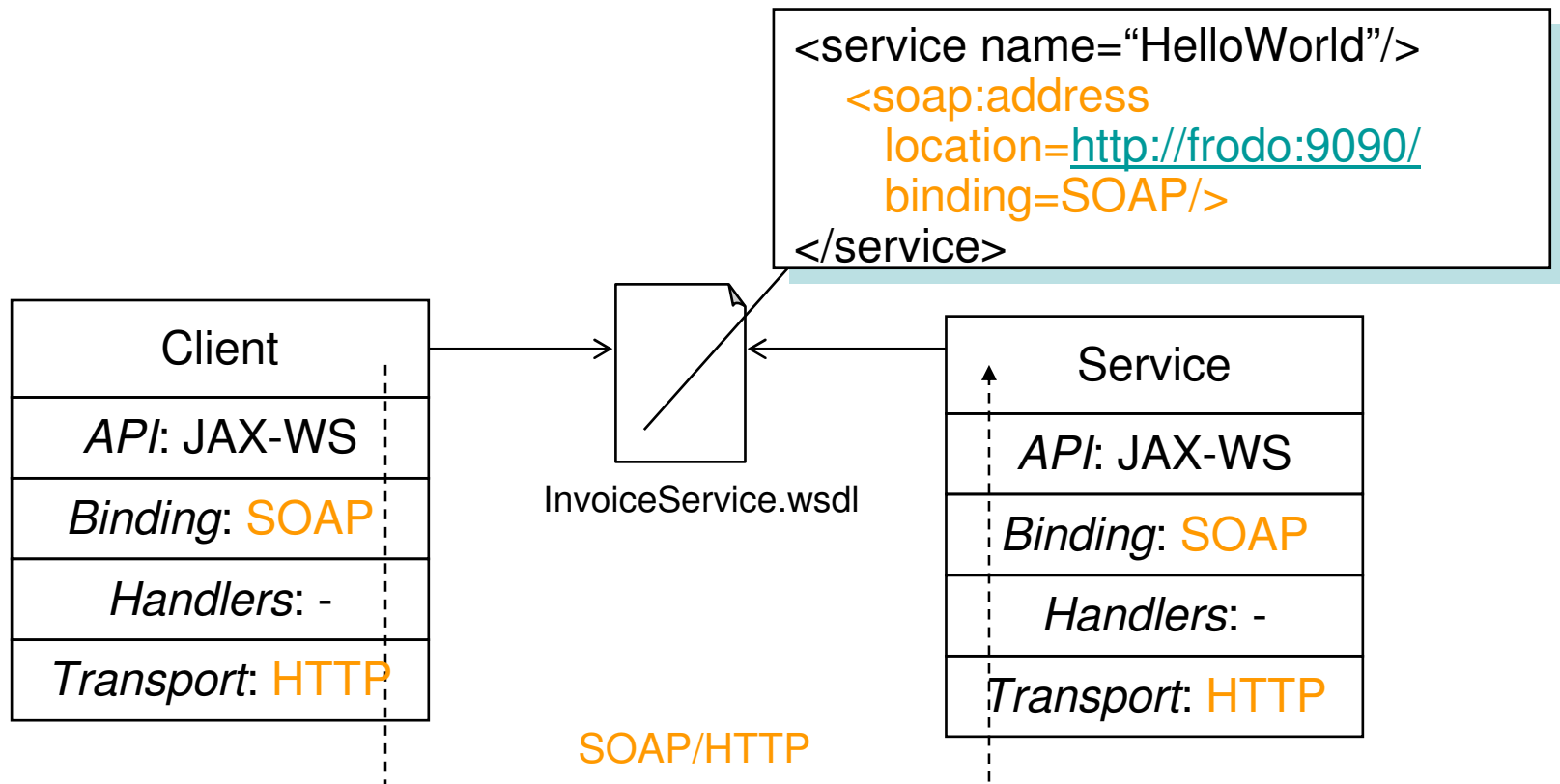
The Celtix communication stack

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



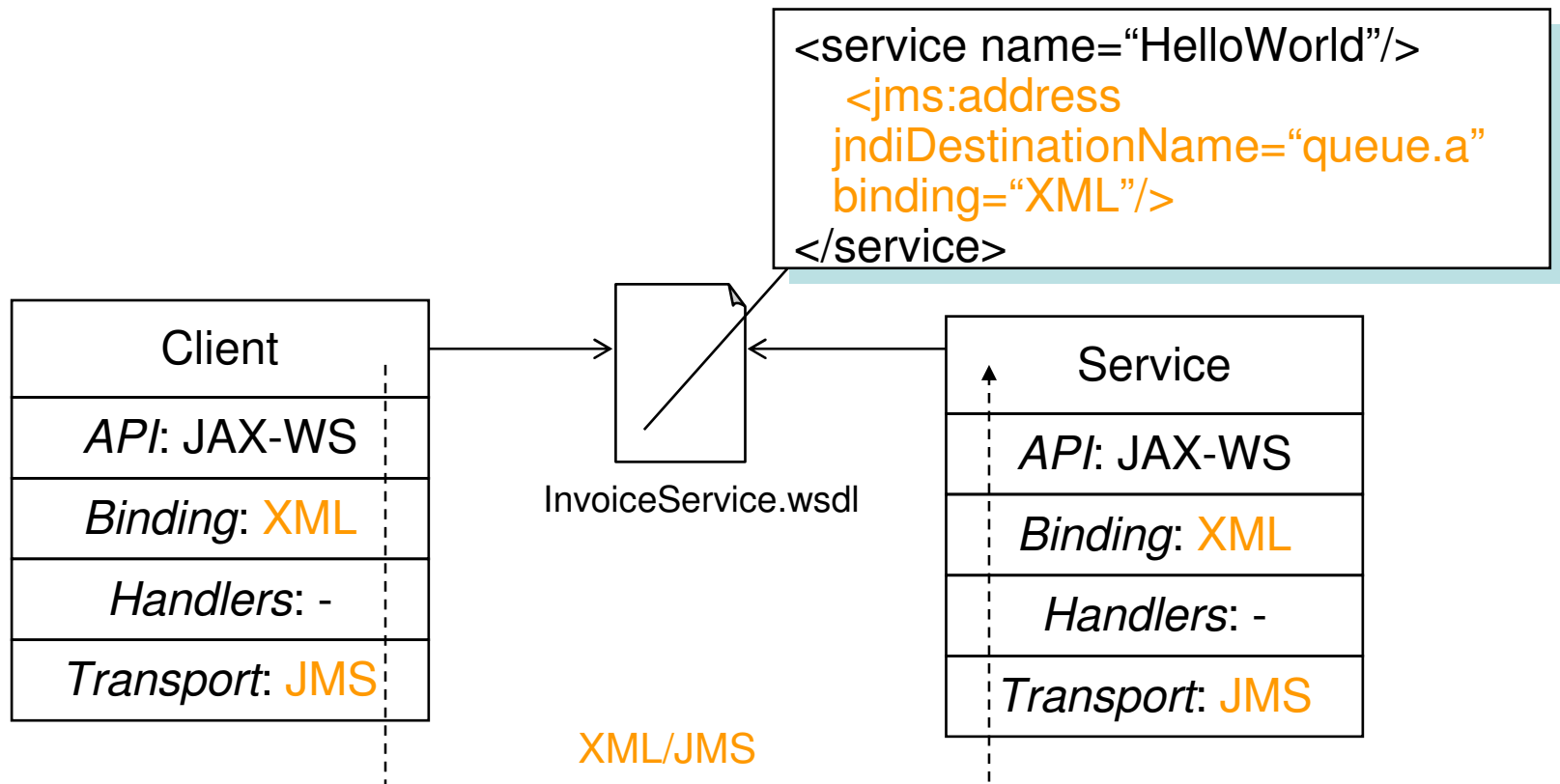
Celtix multi-protocol support

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



Celtix multi-protocol support

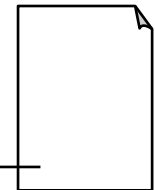
- Change the information in the contract and a different protocol is used.
 - No change to code.



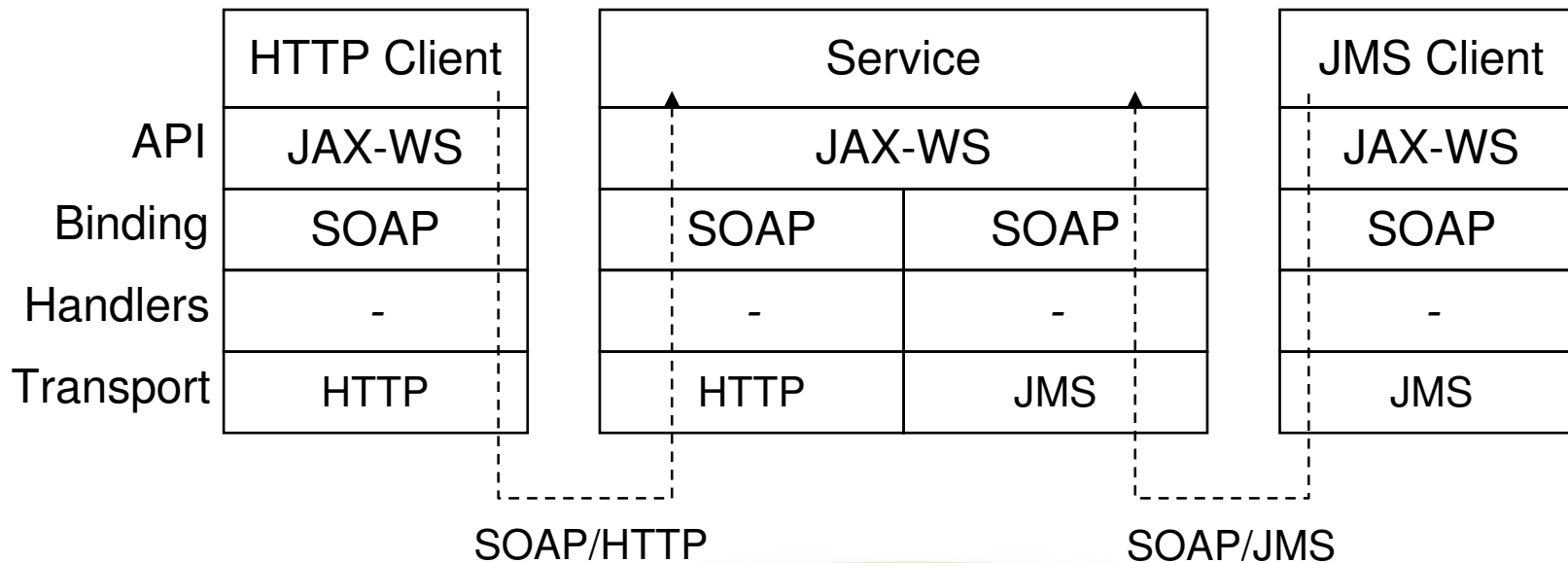
Celtix multi-protocol support (cont')

- › 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.wsdI



Payloads: SOAP

- SOAP allows you to wrap XML payload in a *SOAP envelope*
 - This allows you to add header information for security and transactions in the payload.

```
<soap:Envelope xmlns:soap=...>
  <soap:Header/>
  <soap:Body>
    <ns4:Invoice xmlns:...>
      <ns2:ID>007</ns2:ID>
      <ns2:CopyIndicator>>false</ns2:CopyIndicator>
      <ns2:IssueDate>1967-08-13</ns2:IssueDate>
    </ns4:Invoice>
  </soap:Body>
</soap:Envelope>
```



Payloads: XML

› Some prefer to just send raw-XML

- › Let the transport handle header information for security, transactions, etc.

```
<ns4:Invoice xmlns:...>  
  <ns2:ID>007</ns2:ID>  
  <ns2:CopyIndicator>>false</ns2:CopyIndicator>  
  <ns2:IssueDate>1967-08-13</ns2:IssueDate>  
</ns4:Invoice>
```



Payloads: JSON (Badgerfish)

- JSON (Java Script Object Notation) is an alternative way of marshalling XML on-the-wire.
 - May be preferred by JavaScript programmers invoking on RESTful services.
 - The example below shows the *Badgerfish* marshalling of an Invoice.

```
{ "ns4:Invoice": { "@xmlns": { "$": "http://www.w3.org/2005\n\n/08/addressing/wsdl", "ns2": "urn:oasis:names:draft:u\n\nubl:schema:xsd:CommonBasicComponents-\n\n2", "ns3": "urn:oasis:names:draft:ubl:schema:xsd:CommonA\n\nggregateComponents-\n\n2", "ns4": "urn:oasis:names:draft:ubl:schema:xsd:Invoice\n\n-\n\n2" }, "ns2:ID": { "$": "007" }, "ns2:CopyIndicator": { "$": "fal\n\nse" }, "ns2:IssueDate": { "$": "1967-08-13" }, ... } }
```



Payloads: JSON (Mapped)

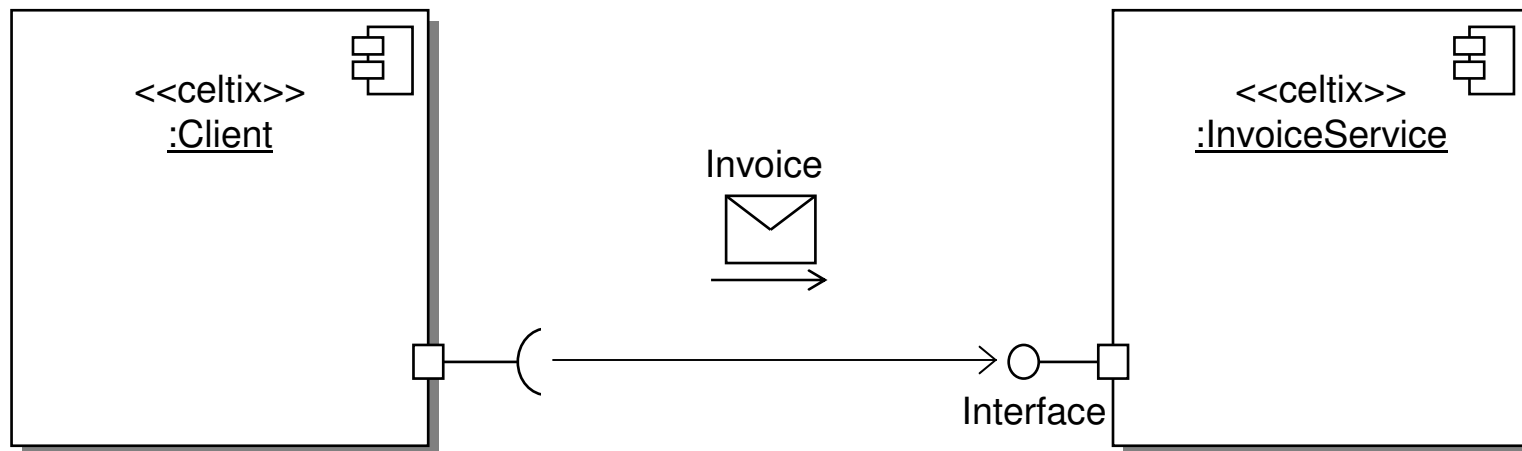
- The JSON *Badgerfish* format is cluttered with XML namespace information.
- A more compact *mapped* notation can be used.
 - Sender and receiver agree on a set of prefixes.

```
{ "inv.Invoice" : { "cbc.ID" : "007", "cbc.CopyIndicator" : "false", "cbc.IssueDate" : "1967-08-13", ... } }
```



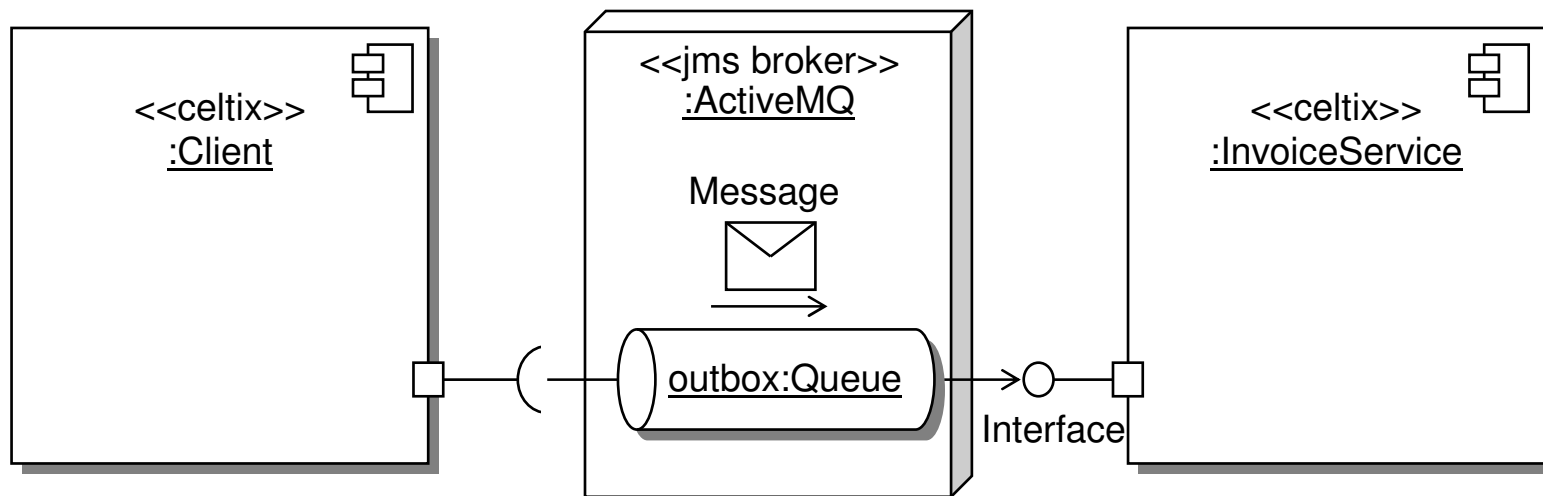
Transports: HTTP

- Popular protocol for web services traffic.
 - Connection-oriented, with no message persistence: suitable for RPC traffic.
 - Reliability can be enhanced if SOAP is used with WS-Reliable Messaging.
- Can send JSON, SOAP and XML over HTTP.



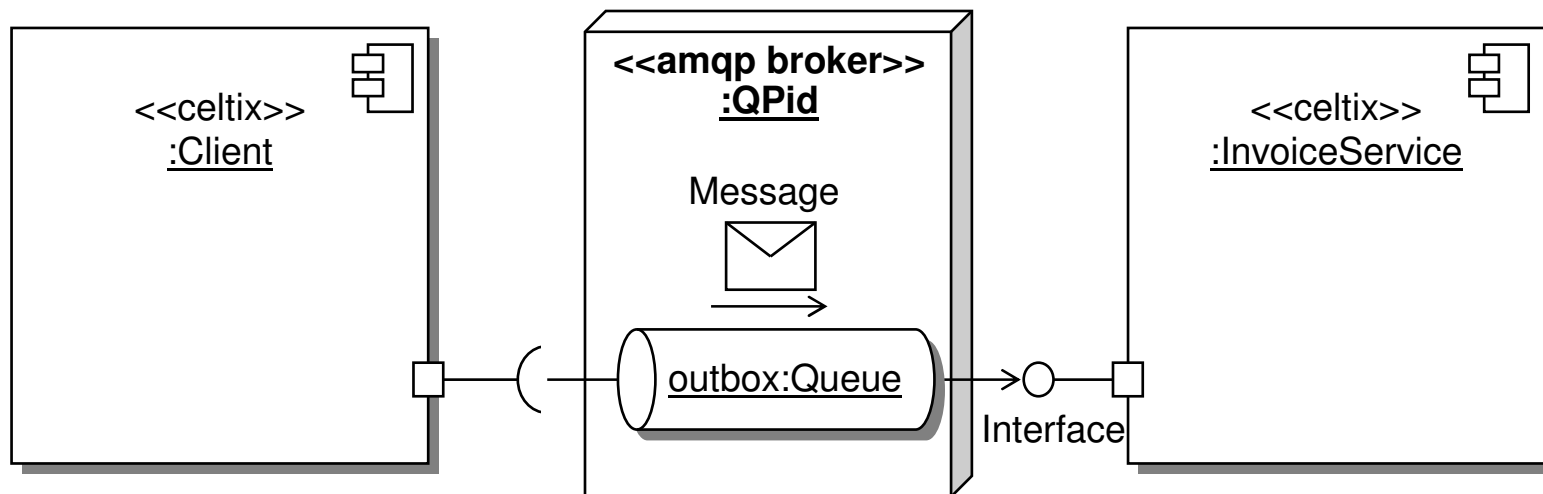
Transports: JMS

- › JMS is hugely popular within the enterprise.
 - › Facilitates one-way and request-response semantics.
- › Celtix integrates with ActiveMQ out-of-the-box.
 - › Other JMS implementations can be configured in.



Transports: AMQP

- AMQP provides an on-the-wire specification for messaging.
 - This addresses the viral nature of JMS implementations, whereby both sender and receiver must use the same JMS provider vendor.
- Celtix Enterprise includes Apache Qpid
 - The AMQP protocol is wrapped with the JMS API, offering a clean integration.



JAX-WS Development



Making Software Work Together™

Focus on contract-based service APIs

- Celtix uses WSDL to abstract away protocol and transport information.
 - The *logical* part of the WSDL contract is mapped to Java using JAX-WS
 - The *physical* part contains transport and protocol information, and is used at run-time.
- Using WSDL as an interface language offers a number of benefits:
 - Separates the service interface from underlying middleware
 - Services can support multiple communication protocols
 - Separates the service interface from underlying implementation
 - Services can be implemented in any language, OS or hardware.
 - Universally supported by the industry



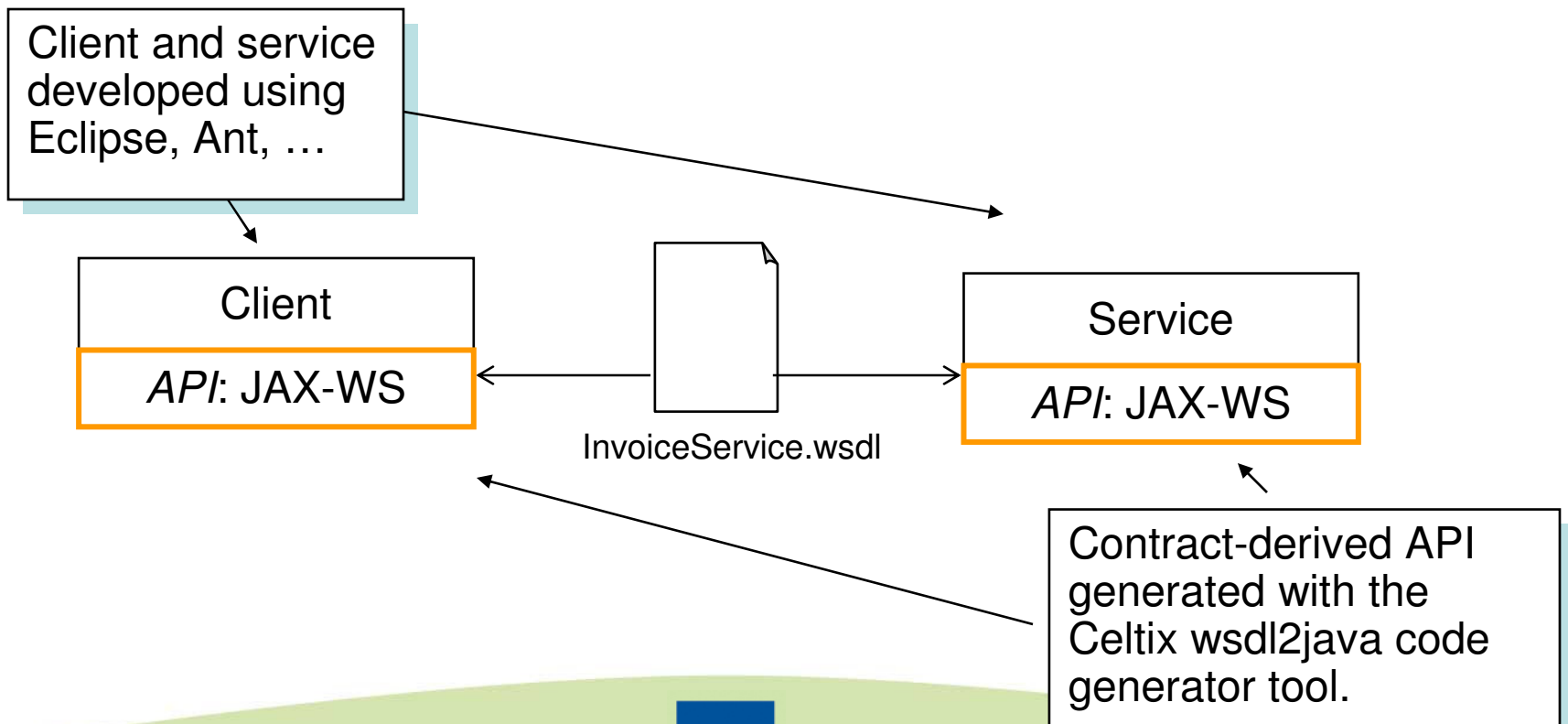
Aside: Java-first Services

- Celtix supports *both* Java-first and Contract-first service design.
 - Contract-first: generate code artifacts from WSDL/XSD contracts.
 - Java-first: use Java 5.0 annotations to mark-up Java byte code. WSDL and XSD artifacts will then be generated on the fly.
- Java-first services are considered easier to create, and are favoured by some for tactical integrations.
- Contract-first design is preferred for strategic SOA.
 - Java-first WSDL artifacts tend to bloat, containing duplicate type definitions across services.
 - Contract-first services tend to be cleaner modular, platform agnostic, and with better attention to versioning.
- This presentation focuses on the *WSDL-first* approach.



JAX-WS: protocol agnostic development

- Developers code to the JAX-WS standard using their favorite Java IDE.
 - Code is pure-Java: no need to worry about the underlying communication protocols.



Example: Hello, World!

- A WSDL contract for Hello-World is shown below
 - Tooling: Eclipse Web Tooling Project (WTP)



- Generate Java code using the Celtix `wsdl2java` tool...

```
C:\> wsdl2java HelloWorld.wsdl
```



Example: Hello, World! (cont')

- The service can be implemented in Java by extending the generated code.

```
public class HelloWorldImpl implements HelloWorld
{
    public String sayHello(String message)
    {
        return "Hi there!";
    }
}
```

- The service is deployed in a container or in a Java mainline:

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



Example: Hello, World! (cont')

- In JAX-WS, annotations are used to mark-up the code.
 - In the previous slide, annotations were inherited from the `HelloWorld` interface.
 - In practice, you should override the generated annotations.

```
@WebService(  
    name = "HelloWorld",  
    serviceName = "HelloWorldService",  
    targetNamespace =  
        "http://www.ionac.com/ps/courseware/HelloWorld",  
    wsdlLocation = "./wsdl/HelloWorld.wsdl",  
    portName = "SOAPOverHTTP"  
)  
public class HelloWorldImpl implements HelloWorld  
{  
    . . .  
}
```



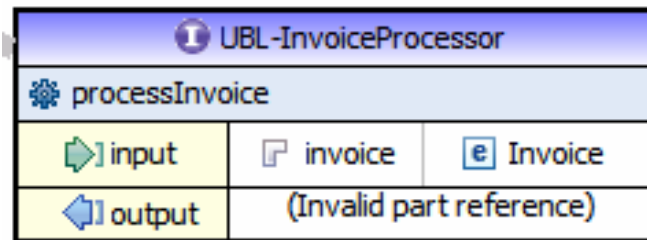
Creating a multi-protocol service for UBL documents



Making Software Work Together™

Designing a web service for UBL Invoices

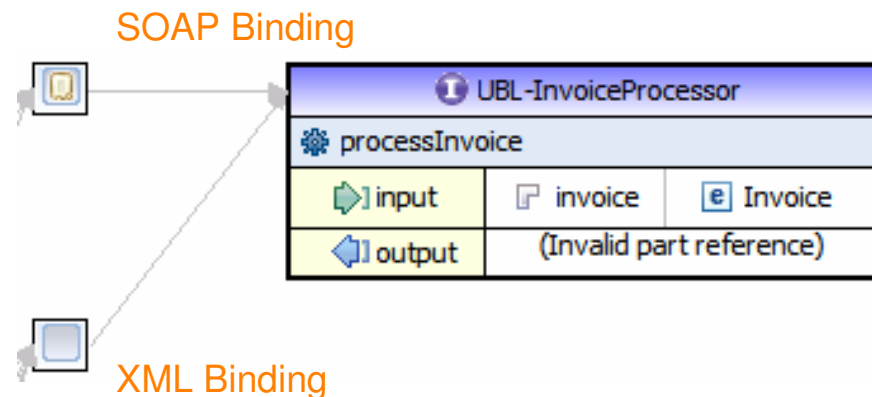
- A WSDL interface was designed to capture *InvoiceProcessor* service semantics.
 - A single operation `processInvoice`, with an input message containing the Invoice and an empty output message.



Designing a web service for UBL Invoices (cont')

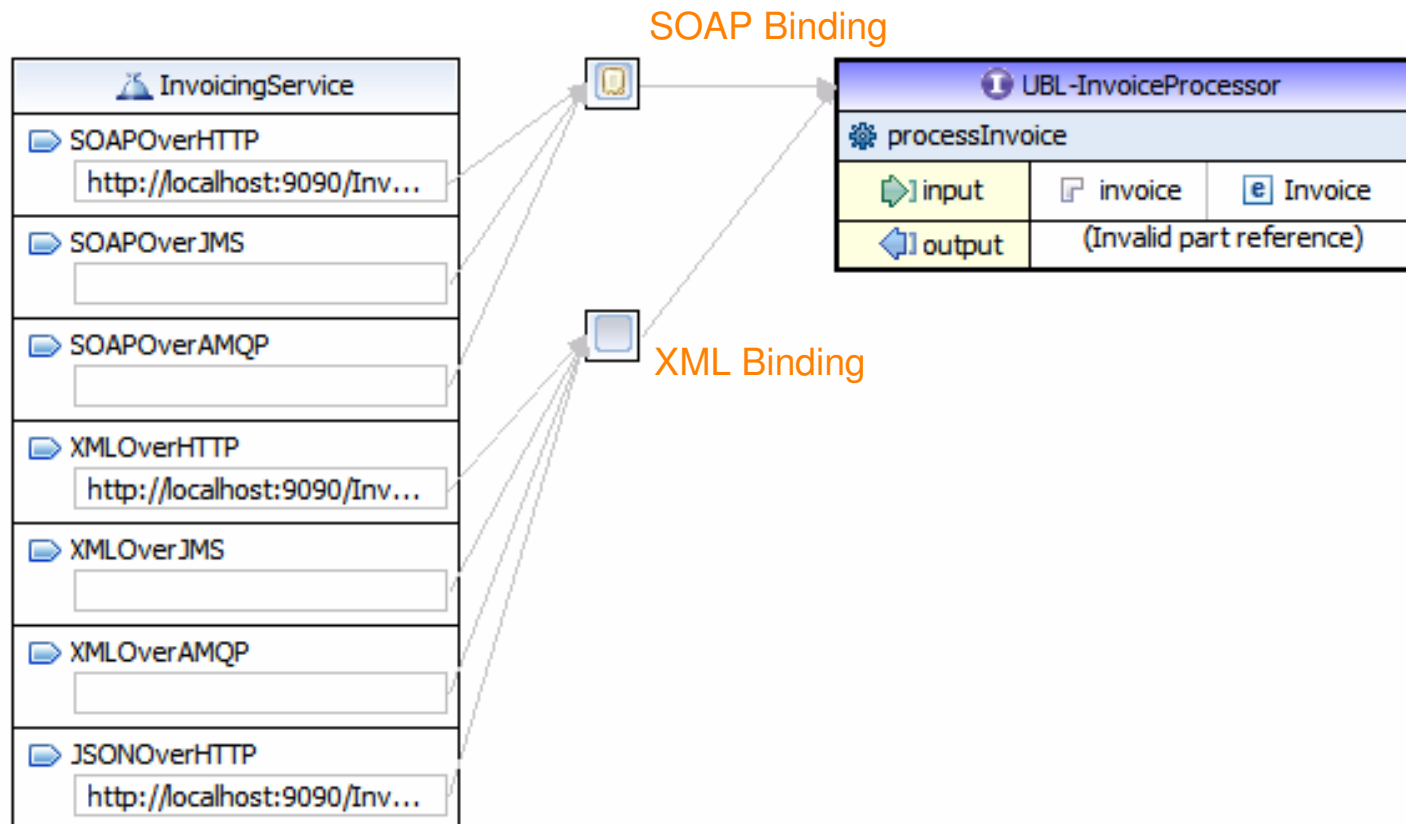
- Generate bindings for SOAP and XML payload.
 - SOAP binding is generated automatically for you by Eclipse WTP.
 - XML binding is generated using Celtix `wsdl2xml` tool.

```
C:\> wsdl2xml -i UBL-InvoiceProcessor InvoiceSvc.wsdl
```



Designing a web service for UBL Invoices (cont')

- Define a service, with a port for each payload/protocol combination.



HTTP Endpoints

- ▶ HTTP endpoints are straightforward.

```
<wsdl:port name="SOAPOverHTTP"
  binding="tns:UBL-InvoiceProcessorSOAP">
  <soap:address
    location="http://localhost:9090/Invoicing/SOAP"/>
</wsdl:port>
```

```
<wsdl:port name="XMLOverHTTP"
  binding="tns:UBL-InvoiceProcessor_XMLBinding">
  <http:address
    location="http://localhost:9090/Invoicing/XML" />
</wsdl:port>
```



JMS Endpoints

```
<wsdl:port name="XMLOverJMS"  
  binding="tns:UBL-InvoiceProcessor_XMLBinding">
```

```
<jms:address  
  destinationStyle="queue"  
  initialContextFactory="...ActiveMQInitialContextFactory"  
  jndiDestinationName="dynamicQueues/ubl/xml"  
  jndiConnectionFactoryName="ConnectionFactory"  
  messageType="text"  
  useMessageIDAsCorrelationID="true">
```

```
<jms:JMSNamingProperty name="java.naming.factory.initial"  
  value="...ActiveMQInitialContextFactory" />
```

```
<jms:JMSNamingProperty name="java.naming.provider.url"  
  value="tcp://localhost:61616" />
```

```
</jms:address>  
</wsdl:port>
```



Making Software Work Together™

AMQP Endpoints

- AMQP endpoints are similar to JMS, with different entries for JMSNamingProperty variables.

```
<wsdl:port name="XMLOverAMQP"  
  binding="tns:UBL-InvoiceProcessor_XMLBinding">
```

```
  <jms:address  
    destinationStyle="queue"  
    jndiDestinationName="demo-ubl/invoice/xml"  
    jndiConnectionFactoryName="local">
```

```
    . . .
```

```
</jms:address>
```

```
</wsdl:port>
```



AMQP Endpoints (cont')

```
<jms:JMSNamingProperty  
  name="java.naming.factory.initial"  
  value=  
  "...qpid.jndi.PropertiesFileInitialContextFactory" />  
  
<jms:JMSNamingProperty name="java.naming.provider.url"  
  value="tcp://localhost:5672" />  
  
<jms:JMSNamingProperty name="connectionfactory.local"  
  value=  
  "amqp://guest:guest@clientid/localhost?brokerlist='tcp://  
  localhost:5672'" />  
  
<jms:JMSNamingProperty name="queue.demo-ubl/invoice/xml"  
  value="queue://demo-ubl/invoice/xml" />
```



Implementing the service

- A JAX-WS service endpoint interface is generated from the WSDL contract using `wsdl2java`.

```
public interface UBLInvoiceProcessor {  
    public void processInvoice(  
        InvoiceType invoice  
    );  
}
```

- To provide an endpoint implementation:
 - Write a class that implements this interface; and,
 - Add appropriate annotations to specify the service and endpoint.



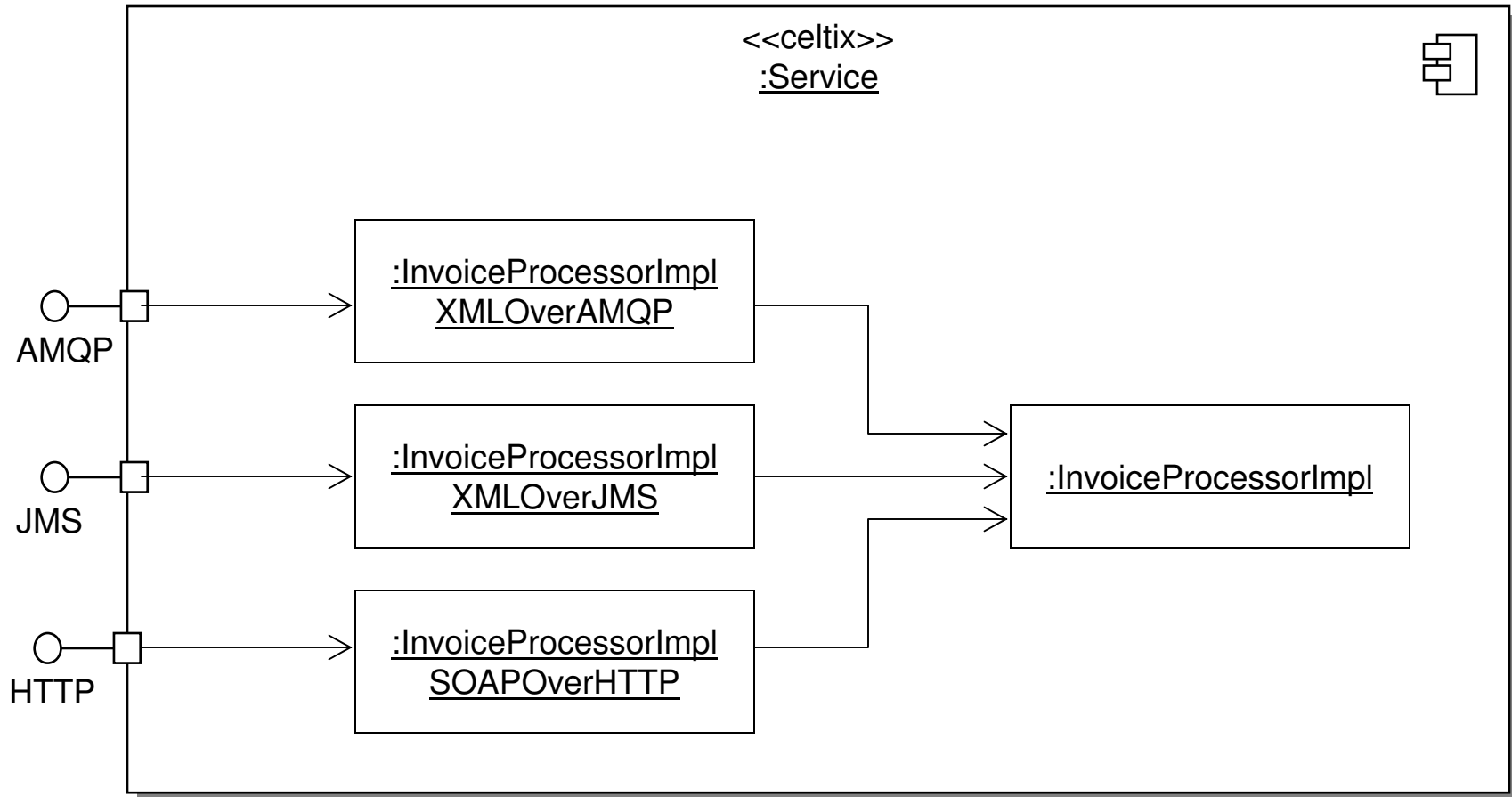
Implementing the service (cont')

- Most web services have just one endpoint: SOAPOverHTTP.
- Our service has more than one endpoint!
 - Need to create a service endpoint implementation for each endpoint.
 - Tip: write one implementation class, then a number of annotated classes that delegate to it.
- We end up with the following architecture (next slide)



Implementing the service (cont')

- End up with the following delegation architecture.



Implementing the service (cont')

➤ Implementation class:

```
public class UBLInvoiceProcessorImpl implements
    UBLInvoiceProcessor {

    public void processInvoice(InvoiceType invoice) {
        System.out.println("Got invoice with id "
            + invoice.getID());
    }

}
```



Implementing the service (cont')

```
@WebService(  
    wsdlLocation = "./wsdl/UBL-InvoiceProcessor.wsdl",  
    targetNamespace =  
        "http://www.iona.com/demos/UBL-InvoiceProcessor/",  
    name = "UBL-InvoiceProcessor",  
    serviceName= "InvoicingService",  
    portName = "SOAPOverAMQP"  
)  
public class UBLInvoiceProcessorSOAPOverAMQP implements  
    UBLInvoiceProcessor {  
  
    private UBLInvoiceProcessorImpl invoiceProcessorImpl;  
  
    public void processInvoice(InvoiceType invoice) {  
        invoiceProcessorImpl.processInvoice(invoice);  
    }  
}
```



Using JSON



Making Software Work Together™

How Celtix uses JSON

- Celtix employs a STAX (Streaming API for XML) parser.
 - High performance, pluggable implementation.
- To produce JSON payload, just use an XML binding, but replace the XML parser with a JSON parser at runtime.
 - In this case, we use *Jettison*.
 - This is performed in code; there is no JSON binding in the WSDL
- Aside: providing a JSON binding for WSDL would be a nicer approach
 - Makes it clear to both service consumer and provider that JSON is used on the wire.



Codesnap - server-side Badgerfish JSON enablement

```
BadgerFishXMLInputFactory xif = new  
    BadgerFishXMLInputFactory();
```

```
BadgerFishXMLOutputFactory xof = new  
    BadgerFishXMLOutputFactory();
```

```
properties.put(XMLOutputFactory.class.getName(), xof);  
properties.put(XMLInputFactory.class.getName(), xif);
```

```
Endpoint ep =  
    Endpoint.create(invoiceProcessorXMLOverHTTP);
```

```
ep.setProperties(properties);
```

```
ep.publish("http://localhost/9091/invoice");
```



Client-side code

- A similar approach is used to enable client-side JSON support in JAX-WS.
 - Less-likely to be used, as JSON is focussed on JavaScript AJAX developers.



Summary



Making Software Work Together™

Summary

- › Celtix allows you to build multi-protocol web services.
- › Developers focus on business logic rather than integration logic.
 - › Lower dependence on specialist middleware skills
 - › Reduced development time
- › Easy to plug in and out protocols as required.
 - › Change from HTTP to JMS? No problem!
 - › Change JMS provider? No problem!
 - › Move to AMQP? No problem!
 - › Support RESTful services with JSON/HTTP? No problem!



Resources

- › Celtix Enterprise: <http://www.iona.com/products/celtix/>
- › UBL:
<http://www.oasis-open.org>
<http://www.xml.com/pub/a/2001/11/07/bosakubl.html>
- › AMQP: <http://www.amqp.org/>
- › JSON: <http://www.json.org/>
- › Jettison: <http://jettison.codehaus.org/>

