Metro (JAX-WS)

Doug Kohlert
Senior Staff Engineer
Sun Microsystems, Inc.

Agenda

- Metro Announcement
- JAX-WS Standards
- Metro
- GlassFish Intro

Project Metro Announcement

- Just announced today on java.net
  - > http://metro.dev.java.net
- New Java.net project representing the GlassFish web services stack.
- The core of Metro is the JAX-WS RI.
  - > Includes JAXB, SAAJ, WSIT, etc.
- Why Metro?
  - > Gives a common name to all of the components that make up the GlassFish WS stack

Agenda

- Metro Announcement
- JAX-WS Standards
- Metro
- GlassFish Intro

JAX-WS Standards

- New, easy to use web services API
  - Replaces JAX-RPC
- Embrace plain old Java object (POJO) concepts
- Descriptor-free programming
- Layered Architecture
- SOAP 1.2, WS-I BP 1.1, MTOM, REST
- 100% of XML Schema via JAXB data binding
- Application portability, smaller footprint
- Part of Java SE 6 and Java EE 5 platforms

Using JAX-RPC 1.1

```
import java.rmi.*;

public class CalculatorServiceImpl
        implements CalculatorServiceSEI {

        public int add(int a, int b) {
      return a+b;
   }
}

import java.rmi.*;

public interface CalculatorServiceSEI
                        extends java.rmi.Remote {
        public int add(int a, int b)
                throws java.rmi.RemoteException;
}
```

```
<?xml version='1.0' encoding='UTF-8' ?>
<webservices xmlns='http://java.sun.com/xml/ns/j2ee' version='1.1'>
  <webservice-description>
    <webservice-description-name>
     CalculatorService</webservice-description-name>
    <wsdl-file> WEB-INF/wsdl/CalculatorService.wsdl</wsdl-file>
    <jaxrpc-mapping-file> WEB-INF/CalculatorService-mapping.xml /jaxrpc-mapping-file>
    <port-component xmlns:wsdl-port_ns='urn:CalculatorService/wsdl'>
      <port-component-name>CaculatorService</port-component-name>
      <wsdl-port>wsdl-port_ns:CalculatorServiceSEIPort</wsdl-port>
      <service-endpoint-interface>endpoint.CalculatorServiceSEI</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>WSServlet_CalculatorService</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
<?xml version='1.0' encoding='UTF-8' ?>
<configuration    xmlns='http://java.sun.com/xml/ns/jax-rpc/ri/config'>
  <service name='CalculatorService'    targetNamespace='urn:CalculatorService/wsdl'
      typeNamespace='urn:CalculatorService/types'    packageName='endpoint'>
    <interface name='endpoint.CalclatorServiceSEI'
        servantName='endpoint.CalulatorServiceImpl'>
    </interface>
  </service>
</configuration>
```

Server-side Programming Model

1. Write a POJO implementing the service
2. Add @Webservice to it
3. Optionallly, inject a WebServiceContext
4. Deploy the application
5. Point your client at the WSDL
   - http://myserver/myapp/MyService?WSDL

Example: Servlet-Based Endpoint

```
@WebService
public class Calculator {
    public int add(int a, int b) {
        return a+b;
    }
}
```

- All public methods become web service operations
- Default values for service name, etc.
- WSDL/Schema generated automatically

Example: EJB 3.0-Based Endpoint

```
@WebService
@Stateless
public class Calculator {
        @Resource
        WebServiceContext context;

    public int add(int a, int b) {
        return a+b;
    }
}
```

- It's a regular EJB 3.0 component so it can use EJB features
  - Transactions, security, interceptors...

Infinite Customizability via Annotations

```
@WebService(name="CreditRatingService",
          targetNamespace="http://example.org")
  public class CreditRating {

    @WebMethod(operationName="getCreditScore")
    public Score getCredit(
        @WebParam(name="customer")
        Customer c) {
      // ... implementation code ...
    }
}
```

Data Binding

JAXB Integrated With JAX-WS

- Lower layer in JAX-WS
- JAX-WS 2.0 delegates all data binding functionality to JAXB 2.0
- One mapping, one set of annotations
- XML Schema 100% supported
- Attachment support (MTOM/XOP)
- Richer type mapping via Java API for XML Processing (JAXP)
  - e.g. javax.xml.datatype.XMLGregorianCalendar

Data Binding Tips

- Use regular Java classes as data types
- Follow JavaBeans™ based property pattern:

```
public String getName() { ... }
public void setName(String name) {...}
```

- Or use public fields:

```
public String name;
```

- Use enumerated types and collections:

```
public enum Color {RED, WHITE, BLUE};
public Color garmentColor;
public List<Person> contacts;
```

Java SE Client-Side Programming

- Point a tool at the WSDL for the service
  - `wsimport http://example.org/calculator.wsdl`
- Generate annotated classes and interfaces
- Call new on the service class
- Get a proxy using a getPort method
- Invoke any remote operations

Example: Java SE-Based Client

```
CalculatorService svc = new CalculatorService();
Calculator proxy = svc.getCalculatorPort();
int answer = proxy.add(35, 7);
```

- No factories yet the code is fully portable
  - CalculatorService is defined by the specification
  - Internally it uses a delegation model

Java EE Client-Side Programming

1. Point a tool at the WSDL for the service
   - `wsimport http://example.org/calculator.wsdl`
2. Generate annotated classes and interfaces
3. Inject a WebServiceReference of the appropriate type
4. Invoke any remote operations

Example: Java EE-Based Client

Still No Factories and No Java Naming and Directory Interface™ API Either!

```
@Stateless
public class MyBean {

   @WebServiceRef(CalculatorService.class)
   Calculator proxy;

   public int mymethod() {
      return proxy.add(35, 7);
}
```

Can I Rename The Generated Classes?

Using The Binding Customization Language

- You can rename pretty much everything
- When you run the tool to import a WSDL, specify some customizations
- Customizations are written in XML
- Two models:
  - Embedded in WSDL/Schema
  - As a separate customization file
- JAXB customizations work the same way

Example: Customization File

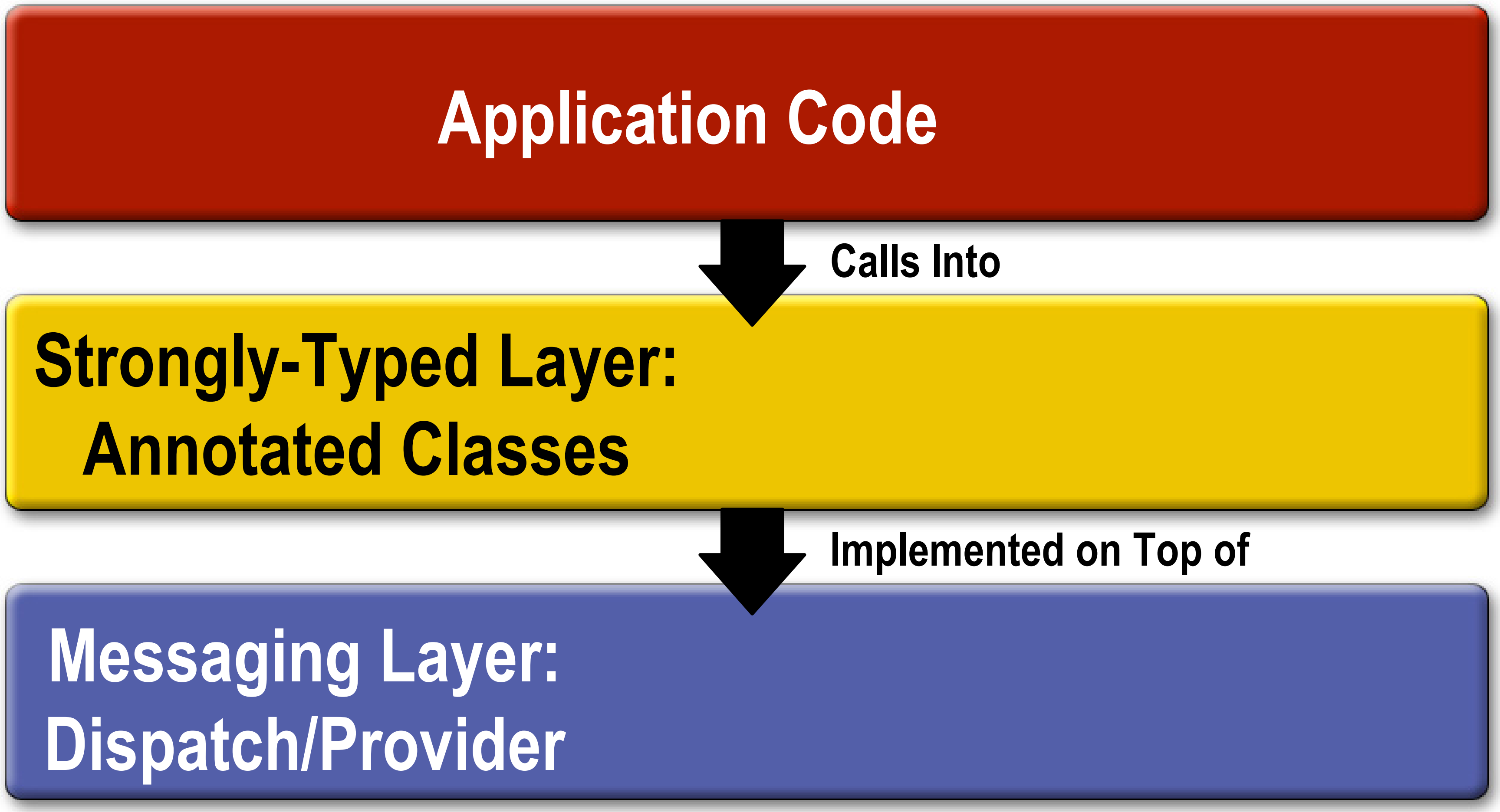
```
<jaxws:bindings
       wsdlLocation="http://example.org/calculator.wsdl">
  <jaxws:package name="org.example.calculator"/>
  <jaxws:bindings
     node="wsdl:portType[@name='Calculator']">
    <jaxws:bindings node="wsdl:operation[@name='add']">
     <jaxws:method name="performAddition"/>
    </jaxws:bindings>
  </jaxws:bindings>
    ...additional binding declarations....
</jaxws:bindings>
```

Protocol and Transport Independence

No SOAP In Sight

- Typical application code is protocol-agnostic
- Default binding in use is SOAP 1.1/HTTP
- Server can specify a different binding, e.g.
  @BindingType(SOAPBinding.SOAP12HTTP_BINDING)
- Client must use binding specified in WSDL
- Bindings are extensible, expect to see more of them
  - e.g. SOAP/Java Message Service(JMS) or XML/SMTP

Layered Architecture



What Does It Mean?

- Upper layer uses annotations extensively
  - > Easy to use
  - > Great toolability
  - > Fewer generated classes
- Lower layer is more traditional
  - > API-based
  - > For advanced scenarios
- Most application will use the upper layer only
- Either way, portability is guaranteed

Lower Level

- Message or Payload access
- Client XML API: Dispatch<T>
  - one-way and asynch calls available
- Server XML API: Provider<T> for T:
  - javax.xml.transform.Source (JAXP)
  - javax.activation.DataSource (ACTIVATION)
  - javax.xml.soap.SOAPMessage (SAAJ)
  - Object using JAXB (strongly-typed)
- May be used to create RESTful clients/services

Dispatch Using PAYLOAD

```
private void invokeAddNumbers(int a,int b) throws
   Exception {
   Dispatch<Source> sourceDispatch = service.createDispatch
     (portQName, Source.class, Service.Mode.PAYLOAD);
   String request = "<addNumbers><num1>" + a +
       "</num1><num2>" + b + "</num2></addNumbers>";

   Source result = sourceDispatch.invoke(new
      StreamSource(new StringReader(request)));

   String xmlResult = sourceToXMLString(result);
   System.out.println("Received xml response: " +
       xmlResult);
}
```

Dispatch Using MESSAGE

```
private void invokeAddNumbers(int a,int b) throws Exception {
    Dispatch<Source> sourceDispatch = service.createDispatch
       (portQName, Source.class, Service.Mode.MESSAGE);
    String request = "<addNumbers><num1>" + a +
        "</num1><num2>" + b + "</num2></addNumbers>";
    String message = "<soapenv:Envelope><soapenv:Body>" +
        request + "</soapenv:Body></soapenv:Envelope>";
    Source result = sourceDispatch.invoke(new
        StreamSource(new StringReader(message)));

    String xmlResult = sourceToXMLString(result);
    System.out.println("Received xml response: " +
        xmlResult);
}
```

Provider Using PAYLOAD

```
public Source invoke(Source source, JAXRPCContext context)
    throws RemoteException {
    try {
        DOMResult dom = new DOMResult();
        Transformer trans = ...;
        trans.transform(source, dom);
        Node addNumbers = dom.getNode().getFirstChild();
        Node num1Element = addNumbers();
        int num1 = Integer.decode(num1Element.getFirstChild()
            .getNodeValue());
        Node num2Element = num1Element.getNextSibling();
        int num2 = Integer.decode(num2Element.getFirstChild()
            .getNodeValue());
        return sendSource(num1, num2);
    } catch(Exception e) {...}
}
```

Provider Using PAYLOAD continued

```
private Source sendSource(int number1, int number2) {
   int sum = number1+number2;
   String body =
       "<ns:addNumbersResponse xmlns:ns=\"...\"><return>"
       + sum
       + "</return></ns:addNumbersResponse>";
   Source source = new StreamSource(
       new ByteArrayInputStream(body.getBytes()));
   return source;
}
```

Web Service Endpoints on the Java SE Platform

- New in Mustang
- Endpoint classes are annotated POJOs
- Application creates an instance and publishes it
- Easy and error-free
- Lots of defaults applied automatically
  - > WSDL, data binding, port number, threading...

Publishing a POJO

```
@WebService
public class Calculator {
    @Resource
    WebServiceContext context;

    public int add(int a, int b) {
        return a+b;
    }
}

// create and publish an endpoint
Calculator calculator = new Calculator();
Endpoint endpoint =
        Endpoint.publish(“http://localhost/calculator”,
                    calculator);
```

Endpoint.publish is All it Takes!

- Really!
- Simple HTTP server embedded in Mustang
- Reasonable defaults for threading, etc.
- WSDL created and published on the fly:

  `http://localhost/calculator?WSDL`

- Optionally, applications can control low-level functionality, e.g.
  - > Threading via an Executor object
  - > WSDL/XML Schema via metadata

Type Substitution

- JAX-WS 2.1 allows for abstract types to be used in SEI
- Use the JAXB @XmlSeeAlso annotation to specify additional classes
  - Can be placed in any JAXB bean or on the SEI
  - Schema types will be generated for all specified types
- When importing WSDLs that include schemas containing type substitutions
  - SEI generated will contain the appropriate @XmlSeeAlso annotation

Type Substitution Example

```
// abstract
public abstract class Car {...}

// concrete classes
public class Toyota extends Car { ... }
public class GMC extends Car { ... }

// SEI
@WebService
@XmlSeeAlso({Toyota.class, GMC.class})
public Dealership {
        public Car tradeIn(Car oldCar) {
                if (oldCar instanceof GMC) {...}
        }
   ...
}
```

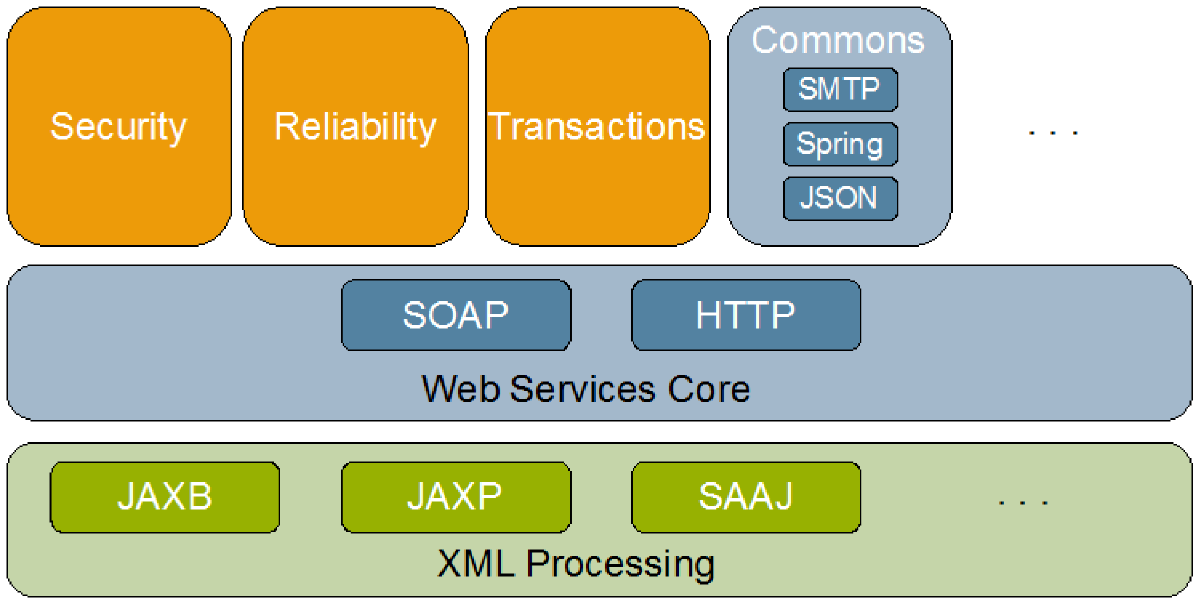
Agenda

- Metro Announcement
- JAX-WS Standards
- Metro
- GlassFish Intro

Metro

- High Performance
- Easy to use
- Extensible

Metro



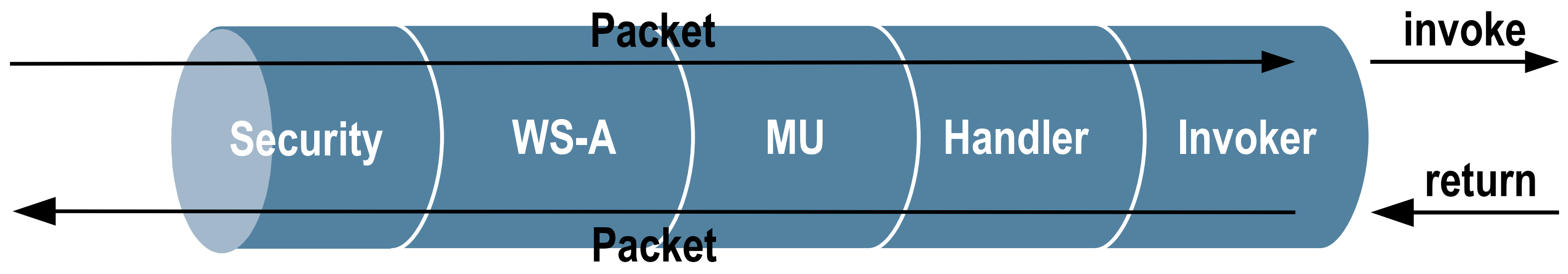
Metro – GlassFish Web Services Stack
metro.dev.java.net

Core Abstractions

- Packet, Message, Header, and Attachment
  - > SOAP message abstraction
- Tube
  - > Abstraction of processing SOAP messages
  - > Composed to form a tubeline
- WSDL Model and SEI Model
  - > Representation, construction, and extensibility

Tubeline

- Tube works like a filter. Acts on a Packet, and then it tells the JAX-WS that the packet should be passed into another Tube.
- Multiple Tube(s) are assembled to form a tubeline
- Typical server-side tubeline



Tube

- SOAP level processing unit
- Runs asynchronously
- Typically extended from base classes
  - AbstractFilterTubeImpl, AbstractTubeImpl

Sample Tube

```
class DumpTube implements AbstractFilterTubeImpl {

    DumpTube(Tube next) {
        super(next);
    }

    NextAction processRequest(Packet req) {
        dump("request", req);
        return super.processRequest(req);
    }

    NextAction processResponse(Packet res) {
        dump("response", res);
        return super.processResponse(res);
    }
}
```

Tubeline Assembly

- Done by TubelineAssembler extension
- Default JAX-WS implementation
  - Covers JAX-WS API functionality
  - For e.g Handlers, MU processing
- Plug-in custom tubeline assemblers

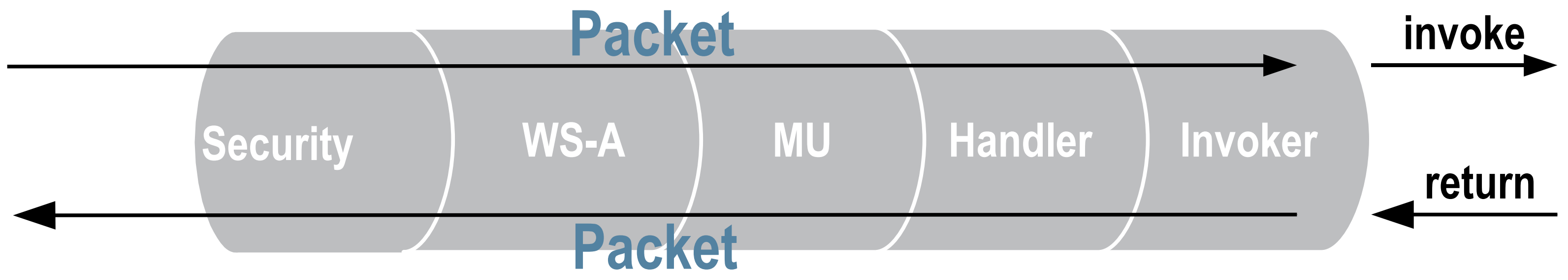
Custom Tubeline Assemblers

- Write your own Assembler
  - > To add more functionality
  - > To replace the default JAX-WS RI Tubes
- WSIT provides its own Assembler
  - > Adds more tubes for WS-* (e.g. RM, MEX, Transactions)

Tubeline Assembly Factory

- TubelineAssemblerFactory is used to find Assembler
  - Container could provide one for a specific endpoint
  - Otherwise, the first one found using META-INF/services mechanism

Packet



- Wrapper around Message
- Adds a few more properties
  - Map for storing "random stuff"
  - Target endpoint address
  - SOAP action

Message

- Single interface to access SOAP message
- Hide physical data representation
  - Implemented many times by different data store
- Randomly accessible headers + read-once body
- Consists of
  - "Header" for each header
  - "Attachment" for BLOBs

Message

```
class Message {
    HeaderList getHeaders()

    String getPayloadLocalPart()

    AttachmentSet getAttachments()

    Source readPayloadAsSource()
    SOAPMessage readAsSOAPMessage()
    Object readPayloadAsJAXB(Unmarshaller)

    writeTo(XMLStreamWriter)

    Message copy()
    ...
}
```

Message

- Backed by
  - InputStream
  - JAXB objects
  - Source for SOAP envelope
  - Source for SOAP payload
  - DOM node
  - SAAJ SOAPMessage
  - None (empty payload)

WSDLModel

- Abstraction of a WSDL document
  - wsdl:service, wsdl:binding, wsdl:portType, extensibilility elements etc.
- Used by the runtime and extensions
  - For example, WSIT TubelineAssembler builds the Tube line based on the policy assertions stored in the WSDLModel
  - Parameter bindings that are not captured in the SEI
    - Rpc/Literal, un-bound parameters, WSDL MIME binding

SEIModel

- Abstraction of the Java Service Endpoint Interface (SEI)
- Data binding for each operation in SEI
- Method dispatching

SEIModel - JavaMethod

```
public interface JavaMethod{
   SOAPBinding getBinding(); //SOAPBinding of this method
   String getOperationName();//wsdl:operation name
   MEP getMEP(); //tells the message exchange pattern
   Method getMethod(); //gives the java.lang.Method
   QName getRequestPayloadName(); //Tag of S:Body child
   SEIModel getOwner(); // SEIModel that owns it
}
```

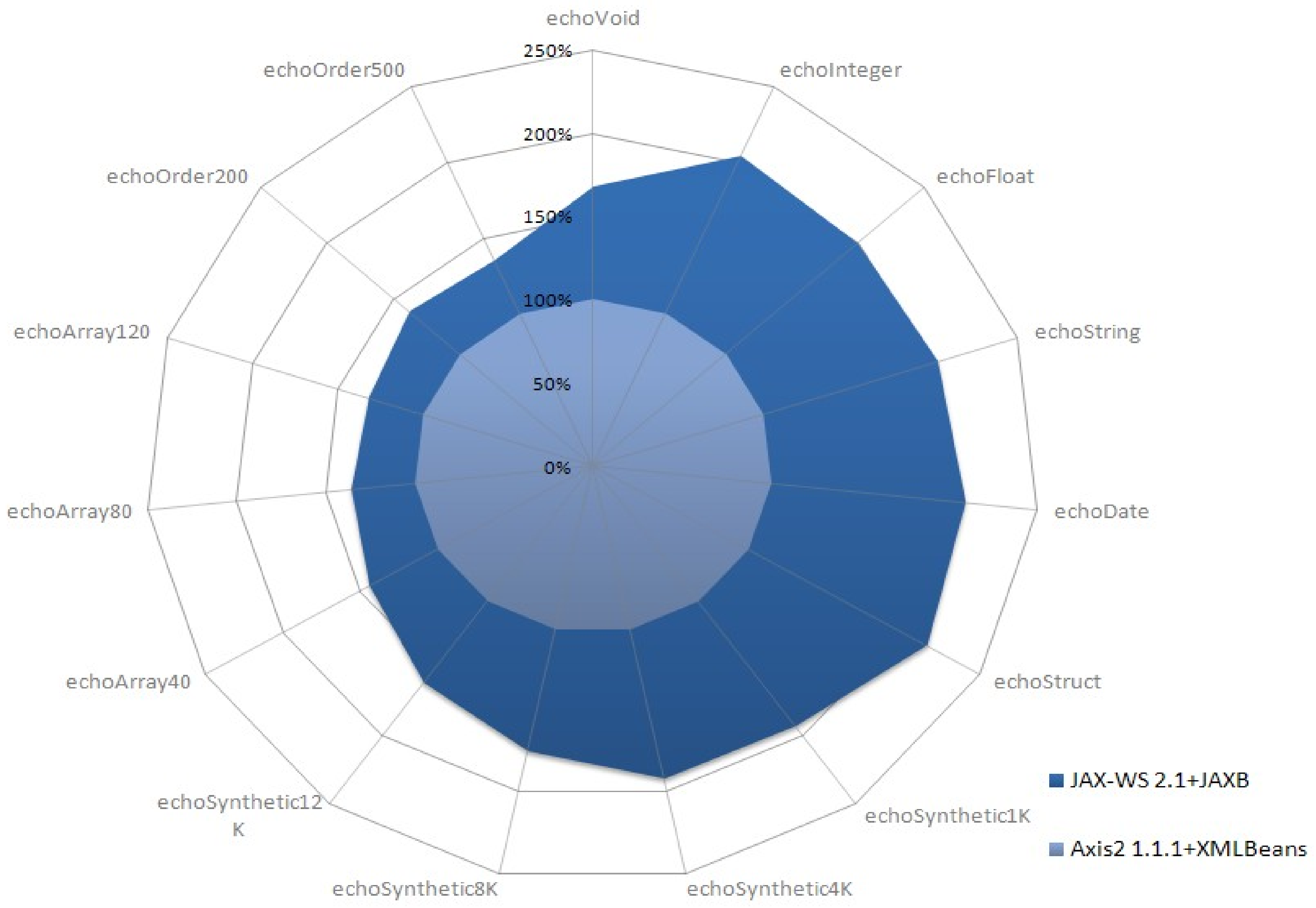
Metro Transports

- In-VM
- Servlet (standard)
- JMS
- SOAP over TCP
- For more information
  - > https://jax-ws.dev.java.net/transport.html

Other Extensibility Points

- WSDL Parsing
- WSDL Generation
- Provide your own transport
- WSIT is a great example
  - Built using the JAX-WS extension points
  - Adds WS-* capability to Metro

Performance vs Axis2



Agenda

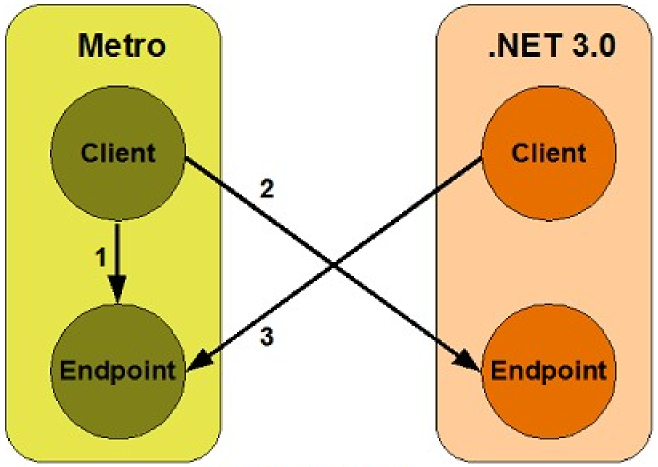
- Metro Announcement
- JAX-WS Standards
- Metro
- GlassFish Intro

For More Information

- http://metro.dev.java.net
- http://forums.java.net/jive/forum.jspa?forumID=46
- users@jax-ws.dev.java.net
- http://glassfish.dev.java.net
- http://wsit.dev.java.net

Metro (JAX-WS)

Doug Kohlert
doug.kohlert@sun.com



Project Metro
metro.dev.java.net