

# JAX-WS 2.2 and JAX-RS 1.1 support in WebSphere Application Server Version 8.0 Beta



# Agenda

- JAX-WS 2.2
  - WS-A and JAX-WS overview
  - New Features for JAX-WS 2.2
- JAX-RS 1.1
  - REST and JAX-RS overview
  - New Features for JAX-RS 1.1
- SOAP vs REST
- Summary and Resources

## Disclaimer

IBM's statements regarding its plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. Information regarding potential future products is intended to outline our general product direction and it should not be relied on in making a purchasing decision. The information mentioned regarding potential future products is not a commitment, promise, or legal obligation to deliver any material, code or functionality. Information about potential future products may not be incorporated into any contract. The development, release, and timing of any future features or functionality described for our products remains at our sole discretion.

# Agenda

- JAX-WS 2.2
  - WS-A and JAX-WS overview
  - New Features for JAX-WS 2.2
- JAX-RS 1.1
  - REST and JAX-RS overview
  - New Features for JAX-RS 1.1
- SOAP vs REST
- Summary and Resources

## What is WS-A 1.0?

- Web Services Addressing (WS-A) 1.0 is a set of 3 W3C specifications:
  - Core
    - Defines a transport-neutral set of abstract properties and an XML representation
    - Facilitates end-to-end addressing of endpoints in messages
  - SOAP Binding
    - Defines the binding of the abstract properties defined in the Core specification to SOAP Messages
  - Metadata
    - Defines how the abstract properties defined in the Core specification are described using WSDL
    - Defines how to include WSDL metadata in endpoint references (EPRs)
    - Defines how WS-Policy can be used to indicate the support of WS-A by a Web service
    - Defines which of the core message properties are mandatory for messages in the various message exchange patterns defined by WSDL
- Supersedes the earlier W3C submission specification from August, 2004

## What is JAX-WS?

- Java™ API for XML-Based Web Services (JAX-WS)
- Java Specification Request (JSR) 224, new to Java EE 5
- Successor to the previous standard, Java API for XML based RPC (JAX-RPC)
  - Still uses SOAP messages and WSDL documents to describe services
  - Annotation based programming model
  - Asynchronous invocations
    - Enable clients to make multiple requests to provider simultaneously
    - Polling or callback models
  - Java ↔ WSDL mapping
    - Command line tools
    - Rational Application Developer
  - All generated code is portable

## Supported Versions

<b>Name</b>	<b>Finalised</b>	<b>WebSphere Version</b>	<b>Comments</b>
WS-A Submission	Aug 2004	6.1 (internal use in 6.0 – not supported)	Split into the next 3 specifications for the final version
WS-A 1.0 – Core	May 2006	6.1	
WS-A 1.0 – SOAP Binding	May 2006	6.1	
WS-A 1.0 – Metadata	Sep 2007	7.0	Earlier WSDL Binding Candidate Recommendation supported in 6.1
JAX-WS 2.0	Apr 2006	6.1 Web Services Feature Pack	No WS-A support
JAX-WS 2.1	May 2007	7.0	Supports WS-A 1.0 – Core and SOAP Binding
JAX-WS 2.2	Dec 2009	8.0 Beta	Supports WS-A 1.0 – Metadata

## WS-A – Message Addressing Properties

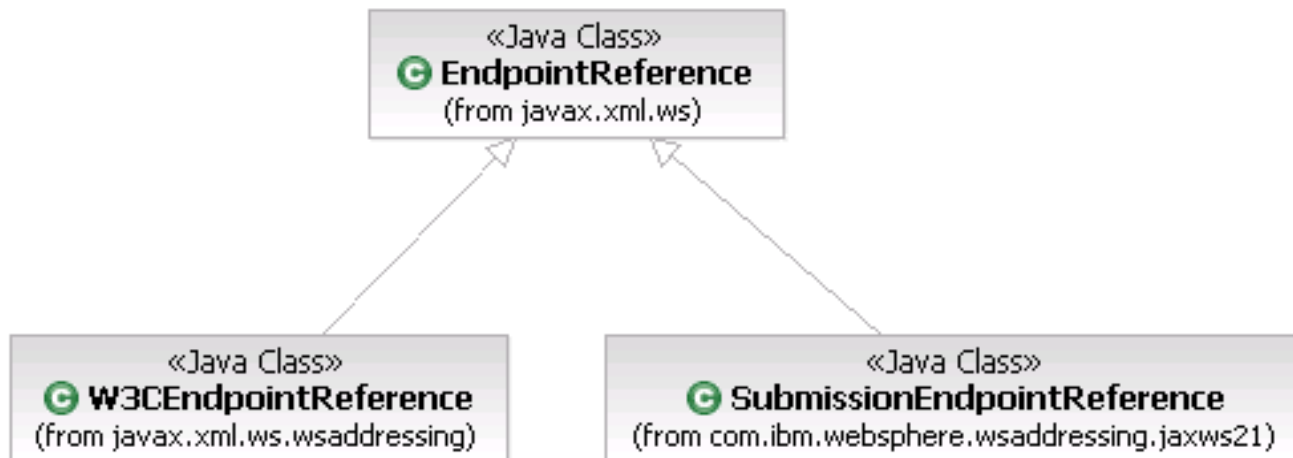
- Message addressing properties (MAPs) are a set of well defined WS-Addressing properties that can be represented as elements in SOAP headers and provide a standard way of conveying information
- E.g. The endpoint to which message replies should be directed (wsa:ReplyTo)

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsa:To>http://test.ibm.com:9080/Test/Service</wsa:To>
    <wsa:ReplyTo>
      <wsa:Address>http://test.com:9080/Test/Service1.Port</wsa:Address>
    </wsa:ReplyTo>
    <wsa:FaultTo>
      <wsa:Address>http://test.com:9080/Test/Service2.Port</wsa:Address>
    </wsa:FaultTo>
    <wsa:MessageID>urn:uuid:fe910bd5-b00c-47b5</wsa:MessageID>
    <wsa:Action>http://test.com:9080/Test/PortType/action</wsa:Action>
  </soapenv:Header>
  <soapenv:Body>
    ...
  </soapenv:Body>
</soapenv:Envelope>
```



## WS-A – Endpoint References

- Endpoint References (EPRs) provide a standard mechanism to encapsulate information about specific endpoints
- EPRs can be propagated to other parties and then used to target the web service endpoint that they represent
- The JAX-WS 2.1 API introduced an EndpointReference class
  - Subclass W3CEndpointReference represents EPRs that follow the WS-A 1.0 Core specification
  - IBM proprietary subclass SubmissionEndpointReference represents endpoint references that follow the Submission version of the WS-A specification
  - There are a number of ways to create and use them in your applications in the API



## JAX-WS 2.2 – Metadata in EPRs

```
<wsa:EndpointReference xmlns:wsa="http://www.w3.org/2005/08/addressing"
  xmlns:test="http://service.test.ibm.com">
  <wsa:Address>http://test.ibm.com:9080/Test/Service</wsa:Address>
  <wsa:ReferenceParameters>
    <test:ID>123456789</test:ID>
  </wsa:ReferenceParameters>
  <wsa:Metadata xmlns:wsdli="http://www.w3.org/2006/01/wsdli-instance"
    wsdli:wsdlLocation="http://service.test.ibm.com
    http://test.ibm.com:9080/Test/Service.wsdl">
    <wsam:ServiceName
      xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
      EndpointName="Port">test:Service</wsam:ServiceName>
    <wsam:InterfaceName>test:Interface</wsam:InterfaceName>
  </wsa:Metadata>
</wsa:EndpointReference>
```

- Metadata will be generated by any of the JAX-WS APIs that return an EPR e.g. `BindingProvider.getEndpointReference` and `W3CEndpointReferenceBuilder.build`
- Used by `EndpointReference.getPort`, which returns a proxy configured using it
- Metadata already generated in version 7.0 to support the WS-A 1.0 Metadata specification, but now it will be compliant with other vendor's JAX-WS runtimes.

## Enabling and Configuring WS-A

- There are 3 properties used to enable and configure WS-A:

Property	Values	Description
enabled	true (default) false	Whether WS-A headers are included on messages
required	true false (default)	Whether to reject messages without WS-A headers  Enforced on the client as well as the service (new for JAX-WS 2.2)
Responses <b>(new for JAX-WS 2.2)</b>	Responses.All (default) Responses.ANONYMOUS Responses.NON_ANONYMOUS	Whether the service requires a synchronous or an asynchronous message exchange pattern.  Synchronous messaging uses an anonymous wsa:ReplyTo  Asynchronous messaging uses a non-anonymous wsa:ReplyTo  Only enforced on the service side

## JAX-WS 2.2 – Configuring WS-A using WSDL WS-Policy

```
<wsp:Policy>
  <wsam:Addressing wsp:Optional="true">
    <wsp:Policy/>
  </wsam:Addressing>
</wsp:Policy>
```

WS-A supported

```
<wsp:Policy>
  <wsam:Addressing>
    <wsp:Policy>
      <wsam:AnonymousResponses/>
    </wsp:Policy>
  </wsam:Addressing>
</wsp:Policy>
```

WS-A required  
and sync only

```
<wsp:Policy>
  <wsam:Addressing wsp:Optional="true">
    <wsp:Policy>
      <wsam:NonAnonymousResponses/>
    </wsp:Policy>
  </wsam:Addressing>
</wsp:Policy>
```

WS-A optional  
and async only

- Enable and configure WS-A on a client or service by attaching WS-Policy assertions to the wsd:port or wsd:binding in the WSDL document
- A proxy created using a getPort call will be configured with the addressing requirements in the associated WSDL

```
<wsp:Policy>
  <wsam:Addressing>
    <wsp:Policy/>
  </wsam:Addressing>
</wsp:Policy>
```

WS-A  
required

## @Addressing Annotation and AddressingFeature

- JAX-WS 2.1 introduced the concept of features to configure clients during development
- JAX-WS 2.1 also introduced the @Addressing annotation as the server-side equivalent of the AddressingFeature
- The responses property was added to both in JAX-WS 2.2

```
new AddressingFeature(true, false, Responses.ALL)
```

WS-A enabled, optional, and sync or async (default)

WS-A enabled, required and async only

```
new AddressingFeature(true, true, Responses.NON_ANONYMOUS)
```

WS-A enabled, optional and sync only

```
@Addressing(enabled=true, required=false, responses=Responses.ANONYMOUS)
```

- Note: There is also IBM proprietary @SubmissionAddressing annotation and SubmissionAddressingFeature for working with the Submission WS-A specification, but they do not have the responses attribute because message exchange patterns were not covered in that specification

## JAX-WS 2.2 – Client Side @Addressing Annotations

- The `@WebServiceRef` annotation defines a reference to a web service invoked by the client
- The `@WebServiceRef` annotation can be used to inject instances of JAX-WS services and ports
- You can now specify the Addressing annotation in combination with the `WebServiceRef` annotation on the client side:

```
public class MyClientApplication {  
    @Addressing(enabled=true, responses=Responses.ANONYMOUS)  
    @WebServiceRef(MyService.class)  
    private MyPortType myPort;  
    ...  
}
```

**WS-A enabled, optional and sync only**

## JSR 109 1.3 – Addressing Deployment Descriptor

- JSR 109 is the Web Services for Java EE specification
- Support for version 1.3 was added to the version 8.0 Beta
- Includes configuring WS-A using deployment descriptors on the client or server application during the packaging phase
- In the service application, add the `<addressing>` element to the `<port-component>` element within the `<webservice-description>` element:
- In the client application, add the `<addressing>` element under the `<port-component-ref>` element within the `<service-ref>` element

```
<port-component>
  ...
  <addressing>
    <enabled>true</enabled>
    <required>true</required>
    <responses>ANONYMOUS</responses>
  </addressing>
  ...
</port-component>
```

```
<service-ref>
  ...
  <port-component-ref>
    ...
    <addressing>
      <enabled>true</enabled>
    </addressing>
  </port-component-ref>
</service-ref>
```

## Policy Sets

- Policy Sets are an IBM proprietary way for administrators to enable and configure qualities of service (WS-Security, WS-A, WS-ReliableMessaging, WS-Transaction, SSL, HTTP, JMS)
- Available since the Web Services Feature Pack for version 6.1
- Both client and provider side
- A Policy Set is
  - Designed for high reuse
  - Decoupled from applications
  - Identified by a unique Name per Cell
  - Designed for ease of management
    - Import and export
    - Copy and modify
    - Automated scripting
    - Defaults out of the box
- There is support in Rational Application Developer to attach Policy Sets to applications (they must be created in WebSphere Application Server first though)



# Creating a WS-A Policy Set – Application Policy Sets Pane

Services → Policy sets → Application policy sets

Click New to create a Policy Set

**Application policy sets**

Use this panel to manage or import application policy sets. Application policy sets define quality of service policies for business-related messages defined in the WSDL. Additional default application policy sets are also available. You can import these policy sets from the default repository with the Import button. Default policy sets are not editable, but you can copy the default policy sets and modify them to suit your needs.

+ Preferences

New... Delete Copy... Import Export...

Select	Name	Editable	Description
You can administer the following resources:			
<input type="checkbox"/>	<a href="#">Kerberos V5 HTTPS default</a>	Not editable	Policies: WS-Security, SSL transport, WS-Addressing <ul style="list-style-type: none"> <li>Message authentication: Using Kerberos V5 token</li> <li>Transport security: Using SSL for HTTP</li> <li>Follows the OASIS Kerberos Token Profile specification</li> </ul>
<input type="checkbox"/>	<a href="#">LTPA WSSecurity default</a>	Not editable	Policies: WS-Security, WS-Addressing <ul style="list-style-type: none"> <li>Message integrity: Digitally sign body, timestamp, addressing headers and LTPA token using RSA digital signing</li> <li>Message confidentiality: Encrypt body, signature, and LTPA token using RSA encryption</li> <li>Message authentication: Using LTPA token</li> </ul>
<input type="checkbox"/>	<a href="#">SSL WSTransaction</a>	Not editable	Policies: WS-Transaction, SSL transport <ul style="list-style-type: none"> <li>Transactional integrity: WS-AtomicTransaction and WS-BusinessActivity context propagation using SSL</li> </ul>
<input type="checkbox"/>	<a href="#">Username SecureConversation</a>	Not editable	Policies: WS-Security, WS-Addressing <ul style="list-style-type: none"> <li>Message integrity: Digitally sign body, timestamp, signature confirmation, addressing headers and Username token</li> </ul>

# Creating a WS-A Policy Set – Add the WS-A Policy Type

Application policy sets
?

**Application policy sets** > New...

Use this page to configure a policy set.

---

**General Properties**

\* Name

Description

The additional properties will not be available until the general properties for this item are applied or saved.

**Additional Properties**

- Attached applications
- Attached deployed assets

---

**Policies**

Use the collection below to choose which policy types to add to the policy set. For each, you can keep the default settings or specify your own.

		State	Description
<input type="button" value="Add"/> <input type="button" value="Delete"/> <input type="button" value="Enable"/> <input type="button" value="Disable"/>			
Custom properties			
HTTP transport			
JMS transport			
SSL transport			
WS-Addressing			
WS-ReliableMessaging			
WS-Security			
WS-Transaction			

# Creating a WS-A Policy Set – Define the Assertions

**Policies**

Use the collection below to choose which policy types to add to the policy set. For each, you can keep the default settings or specify your own.

Select	Policy	State	Description
<input type="checkbox"/>	<a href="#">WS-Addressing</a>	Enabled	Policies for addressing Web services using endpoint references and message addressing properties.

Total 1

Click the Policy Type to configure its settings

Select the settings you want (required async only in this case) and click OK

**Application policy sets > WS-A Required Async Only > WS-Addressing**

Define the appropriate WS-Addressing policy assertions for this policy set.

WS-Addressing is mandatory

**Messaging style**

- Synchronous and asynchronous
- Synchronous only
- Asynchronous only

# Attaching a WS-A Policy Set to a Service

Select the service and then click Attach Policy Set

Applications → Application Types → WebSphere enterprise applications → click on the service application → Service provider policy sets and bindings

Select	Application/Service/Endpoint/Operation	Attached Policy Set	Binding	Policy Sharing
You can administer the following resources:				
<input checked="" type="checkbox"/>	WSAEA-MixedNS	None	Not applicable	Not applicable
<input type="checkbox"/>	DestinationRFService	None	Not applicable	Not applicable

Select the Policy Set from the drop down list

Select	Application/Service/Endpoint/Operation	Attached Policy Set	Binding	Policy Sharing
You can administer the following resources:				
<input type="checkbox"/>	WSAEA-MixedNS	None	Not applicable	Not applicable
<input type="checkbox"/>	DestinationRFService	None	Not applicable	Not applicable

## Disabling WS-A

```
@Addressing(enabled=false)
```

```
new AddressingFeature(false)
```

- Disabling WS-A on clients prevents WS-A headers being sent on outbound messages
- Disabling WS-A on servers additionally prevents processing of incoming WS-A headers
- You do not have to disable WS-A support even if your application does not require it, it does not have a negative impact on the running of applications in most cases
- WS-A support is disabled by default on clients and enabled by default on services
- Disable WS-A support by setting `enabled=false` on either the `@Addressing` annotation, Addressing deployment descriptor, `AddressingFeature` (client only)

```
public class MyClientApplication {  
    @Addressing(enabled=false)  
    @WebServiceRef(MyService.class)  
    private MyPortType myPort;  
    ...  
}
```

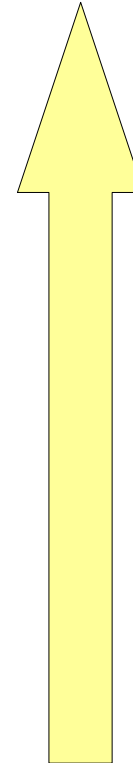
```
<port-component>  
    ...  
    <addressing>  
        <enabled>false</enabled>  
    </addressing>  
    ...  
</port-component>
```

```
<service-ref>  
    ...  
    <port-component-ref>  
        ...  
        <addressing>  
            <enabled>false</enabled>  
        </addressing>  
    </port-component-ref>  
</service-ref>
```

## JAX-WS 2.2 – Configuration Precedence Hierarchy

- Policy Set
- The `com.ibm.websphere.webservices.use.async.mep` property on the request context (client only)
- The IBM proprietary WS-A SPI message addressing properties on the request context (client only)
- Deployment Descriptor
- Annotation
- Feature (client only)
- WSDL WS-Policy
- UsingAddressing WSDL element

Highest Precedence

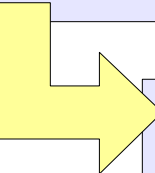


Lowest Precedence

## WSDL to Java Code Generation

- JAX-WS defines a mapping from WSDL 1.1 to Java. This mapping is used when generating web service interfaces for clients and endpoints from a WSDL 1.1 description.
- Generated Java code will contain `@Action` and `@FaultAction` annotations (JAX-WS 2.2)

```
...  
<operation name="test">  
  <input wsam:Action="input" message="tns:test"/>  
  <output wsam:Action="output" message="tns:testResponse"/>  
  <fault wsam:Action="fault1" message="tns:Exception1" name="Exception1"/>  
  <fault wsam:Action="fault2" message="tns:Exception2" name="Exception2"/>  
</operation>  
...
```

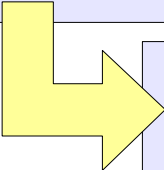


```
...  
@Action(input = "input", output = "output", fault = {  
    @FaultAction(className = Exception1.class, value = "fault1"),  
    @FaultAction(className = Exception2.class, value = "fault2")  
})  
public String test(String param) throws Exception1, Exception2;
```

## Java Code to WSDL Generation

- JAX-WS also defines a mapping from Java to WSDL 1.1. This mapping is used when generating web service endpoints from existing Java interfaces.
- This mapping is also used at runtime when publishing the WSDL for a service without a packaged WSDL
- As well as generating wsam:Action attributes on the operation child elements for @Action and @FaultAction annotations on methods, WS-Policy is generated for @Addressing annotations

```
@Addressing(enabled=true, required=false, responses=Responses.NON_ANONYMOUS)
```



```
<binding name="TestBinding" type="tns:Test">  
  <wsp:PolicyReference URI="#WSAM_Addresssing_Policy"/>  
  ...  
</binding>  
  ...  
<wsp:Policy wsu:Id="WSAM_Addresssing_Policy">  
  <wsam:Addresssing wsp:Optional="true">  
    <wsp:Policy>  
      <wsam:NonAnonymousResponses/>  
    </wsp:Policy>  
  </wsam:Addresssing>  
</wsp:Policy>
```



## JAX-WS 2.2 – Published WSDL WS-Policy

- There are 4 ways to access the WSDL document for a Web service:
  - 1) Services > Service providers > click on the service name > WSDL document
  - 2) Applications > Application Types > WebSphere enterprise applications > click on the application name > Publish WSDL files > Click the WSDL zip file to download
  - 3) HTTP GET request
  - 4) Web Services Metadata Exchange (WS-MetadataExchange) GetMetadata request
- We return a WSDL packaged in the application without modification
  - Developers should ensure any WS-A WS-Policy matches the runtime behaviour of the service to allow clients to be configured correctly
- If there is no WSDL packaged in the application, one will be generated by the JAX-WS runtime and it will contain WS-A WS-Policy matching any annotations in the Java code
  - This does not take account of deployment descriptors and policy sets which could also make the runtime behaviour differ from the published WS-Policy
- Administrators can ensure the published WS-Policy matches runtime behaviour using Policy Sets combined with WS-Policy Sharing
  - Any existing WS-Policy is stripped out of the WSDL and new WS-Policy is generated from the runtime configuration

# Policy Sharing – Service Provider Policy Sets and Bindings

- To configure WS-Policy Sharing:
  - Applications > Application Types > WebSphere enterprise applications > click on the service application > Service provider policy sets and bindings
  - Click the link in the Policy sharing column in the row with the Policy Set attached that you want to share:

<input type="button" value="Attach Policy Set"/> <input type="button" value="Detach Policy Set"/> <input type="button" value="Assign Binding"/>				
Select	Application/Service/Endpoint/Operation	Attached Policy Set	Binding	Policy Sharing
You can administer the following resources:				
<input type="checkbox"/>	WSAEA-MixedNS	<a href="#">WS-A Required Async Only</a>	Default	<a href="#">Disabled</a>
<input type="checkbox"/>	DestinationRFService	WS-A Required Async	Default	Disabled

## Policy Sharing – The Policy Sharing Pane

- For WSDLs obtained by using an HTTP GET request, select Exported WSDL
- To enable WS-MetadataExchange and for WSDLs obtained by using a WS-MetadataExchange GetMetada request, select WS-MetadataExchange request
  - Optional: If you want to use message-level security, select Attach a system policy set to the WS-MetadataExchange, then select a suitable policy set and binding

### Service Provider WS-Policy Control Properties

Allow clients to acquire policy from:

- Exported WSDL (HTTP messages secured with the application transport policy if defined)
- WS-MetadataExchange request
  - Attach a system policy set to the WS-MetadataExchange request

Policy set:

SystemWSSecurityDefault ▼

Binding:

Default ▼

Apply OK Reset Cancel

# Agenda

- JAX-WS 2.2
  - WS-A and JAX-WS overview
  - New Features for JAX-WS 2.2
- JAX-RS 1.1
  - REST and JAX-RS overview
  - New Features for JAX-RS 1.1
- SOAP vs REST
- Summary and Resources

## What is REST?

- REpresentational State Transfer (REST)
- Architectural style
  - Roy Fielding's doctoral dissertation
  - No formal specification (for architectural style)
  - Specs are for HTTP and related technologies (XML, JSON, HTML, CSS, JavaScript etc)
- Manipulate resource representations (nouns) defined at URIs with pre-defined methods (verbs)

## What are resources?

- Data
  - Person
  - Book
  - Collection of books,
  - Shopping cart,
  - Application metadata
  - Rational Team Concert defect
- Every resource is addressable by a URI (URL)
  - [http://localhost:9080/mywebapp/People/Mark\\_Twain](http://localhost:9080/mywebapp/People/Mark_Twain)
  - <http://localhost:9080/mywebapp/Books>
  - [http://localhost:9080/mywebapp/Books/The\\_Adventures\\_of\\_Tom\\_Sawyer](http://localhost:9080/mywebapp/Books/The_Adventures_of_Tom_Sawyer)
- Every resource can be represented by one **or more** data formats (media types)
  - XML
  - JSON
  - Plain Text
  - Images

## What are media types?

- Essentially data formats
- “application/xml”, “text/plain”, “application/octet-stream”, “application/json”, “text/html”
  - “type”/”subtype”
  - “application/\*”, “text/\*”, “custom/\*”
- “Content-Type” HTTP header
  - As request header, client sends a request with a message body; the server needs to understand what format the data is in.
  - As a response header, the server has decided what type the response is; informs the client what format the data is in
- “Accept” HTTP header
  - Different clients can understand different types
  - Content negotiation used to select the best representation for a given response when there are multiple representations available
- Media types can also have parameters (i.e. application/xml; charset=UTF-8)

## How do you manipulate resources?

- Use HTTP methods to manipulate resource
- To read a resource, you issue a HTTP GET client request and retrieve the resource
- To create a resource, you could issue a HTTP POST
- Ideas of “safe” methods (GET / HEAD; not POST, PUT, DELETE)
- Another idea is idempotent methods (GET, PUT, DELETE; not POST)
  - Browser warnings about re-submitting forms via POST

<b>Database Action</b>	<b>HTTP Method</b>
Create	POST (or PUT)
Read	GET
Update	PUT (or POST)
Delete	DELETE



## Simple HTTP client request / server response

- Client Request :

```
POST /mywebapp/People/Jane?q=123 HTTP/1.1  
Host: www.example.com  
Content-Type: application/xml
```

```
<?xml version="1.0"?><data>Client request entity. This could have been binary  
content.</data>
```

- Server Response:

```
HTTP/1.1 200 OK  
Date: Wed, 19 Jan 2011 17:08:12 GMT  
Server: WAS  
Content-Type: text/plain; charset=UTF-8
```

```
Hello World! Content of server response.
```

## What is JAX-RS?

- Java™ API for RESTful Web Services (JAX-RS)
- JSR-311, new to Java EE 6
- Annotation-based approach to developing RESTful services
- Nothing new for administrators, basically a servlet API that lives in the Web container
- Makes servlet / web application development easier
- Two major parts:
  - Routing (finding out which code to invoke)
  - Serializing/deserializing. Turning request message bodies (entities) into Java types and turning Java types into response message bodies.

## Simple JAX-RS class

```
@javax.ws.rs.Path("/people")
public class PersonCollection {

    private static String people = "Amy, Bob, Carol, David";

    @javax.ws.rs.GET
    @javax.ws.rs.Produces("text/plain")
    public String getPeople() {
        return people;
    }

    @javax.ws.rs.POST
    @javax.ws.rs.Produces("text/plain")
    @javax.ws.rs.Consumes("text/plain")
    public String getPeople(String requestEntity) {
        people += requestEntity;
        return people;
    }
}
```

**Root resource** – class annotated with `@Path`.  
This exposes the class at the `/people` path.

**Resource method** - method  
in a root resource that is  
bound to HTTP methods  
using annotations: `@GET`,  
`@POST`, `@PUT`, `@DELETE`  
and `@HEAD`.

HTTP GET requests sent to  
the `/people` path will be  
handled by the `getPeople()`  
resource method.

## How does JAX-RS route requests?

- Routing is based on URLs and classes with their `@Path` values
- `http://<hostname>:<port>/<Context Root of Web App>/<servlet mapping>/<@Path value>`
  - `http://localhost:9080/mywebapp/rest/people`
  - Context root defined in the EAR's `application.xml` deployment descriptor or set when installing the application.
  - Servlet mapping is defined in `web.xml`. Basically declare the JAX-RS servlet to map to a “`/<something>/*`” URL. For example “`/rest/*`” in the URL above.

## More on URL targeting

```
@Path("/people/{personID}")
/* URLs like /people/1234 or /people/abcd */
public class Person {

    @GET
    @Produces("text/plain")
    public String getPersonWithID(@PathParam("personID") String personID) {
        String somePerson = ""; /* lookup via database the information */
        return "the person as a string";
    }

}
```

- Can capture parts of the URL to identify resource (“personID”) by using the `@PathParam` annotation
- By default, captures “[^/]+?”
  - Basically any character up to the next /
- {personID} could also be {personID:<Java regular expression>}
  - {personID:a[^/]+?} to capture paths that start with the character “a”

## Sub-resource Methods

```
@Path("/people")
public class PersonCollection {
    private static String people = "Amy, Bob, Carol, David";

    @GET
    @Produces("text/plain")
    public String getPeople() {
        return people;
    }

    @GET
    @Path("/{personID}") // note removed /people from previous slide
    @Produces("text/plain")
    public String getPersonWithID(@PathParam("personID") String person) {
        String somePerson = ""; /* lookup via database the information */
        return "the person as a string";
    }
}
```

**Sub-resource method** – resource methods that are also annotated with an `@Path` annotation that further qualifies the selection of the method.

An HTTP GET request sent to the `/people/{personID}` path would be handled by the `getPersonWithID` subresource method.

## Even more advanced, dynamic sub-resource locators

**Sub-resource locator** – methods that further resolve the resource that should handle a given request. They have an `@Path` annotation like subresource methods, but they do not have an HTTP request method annotation.

```
@Path("/people")
public class PersonCollection {
    /* notice no @GET or any other HTTP method*/
    @Path("/{personID}")
    public Object getPersonWithID(@PathParam("personID") String personID) {
        if (personID.equals("1234")) {
            return new Manager();
        }
        return new Employee();
    }
}
```

**Subresource** – similar to root resources, except they are not annotated with the `@Path` annotation since their path is described on the subresource locator. Usually contain methods that are annotated with HTTP request method designators to serve the request.

```
public class Manager {
    @GET /* GET method for /people/1234 */
    public String get() { return "hello"; }
}
```

Any HTTP request to the `/people/{personID}` path would be handled by the `getPersonWithID` subresource locator.

```
public class Employee {
    @GET /* GET method for /people/5678 */
    public String getTotallyDifferentMethod() {
        return "world";
    }
}
```

An HTTP GET request to the `/people/5678` path would be handled by `getTotallyDifferentMethod()` in the `Employee` subresource.

```
@POST /* POST method for /people/5678 */
public String uniquePostMethod(String requestEntity) { return "hi"; }
```

## What about those media types in JAX-RS?

```
@GET
@Produces("text/plain")
public String getPeopleAsText() {
    return "the people as a string";
}
```

- How did it turn that String into the server response entity (the body of the response)?
- JAX-RS has the concept of a MessageBodyReader and MessageBodyWriter.
- Some are built-in for certain common types (byte arrays, JAXB types, Strings, etc.)
- Custom ones can be added to the application.

```
@javax.ws.rs.ext.Provider
@Produces("application/json")
/* you can limit the media types for which this provider will be used */
public class MyCustomTypeWriter implements javax.ws.rs.ext.MessageBodyWriter<CustomType>

@javax.ws.rs.ext.Provider
@Consumes("text/xml")
public class MyCustomTypeReader implements javax.ws.rs.ext.MessageBodyReader<CustomType>
```

- User defined ones take precedence over system provided ones



# String MessageBodyWriter

```
@Provider
public class StringProvider implements MessageBodyWriter<String> {

    public long getSize(String t,
                        Class<?> type,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mediaType) {
        Return -1; /* if you know the Content-Length, return it; else -1 */
    }

    public boolean isWriteable(Class<?> type,
                               Type genericType,
                               Annotation[] annotations,
                               MediaType mediaType) {
        return String.class == type;
    }

    public void writeTo(String t,
                        Class<?> type,
                        Type genericType,
                        Annotation[] annotations,
                        MediaType mediaType,
                        MultivaluedMap<String, Object> responseHttpHeaders,
                        OutputStream entityStream) throws IOException {
        /* put String onto entityStream */
    }
}
```

## What if I want to serve multiple media types?

```
@GET
@Produces("text/plain")
public String getPeopleAsText() {
    return "the people as a string";
}
```

```
@GET
@Produces("application/xml")
public String getPeopleAsXML() {
    return
```

```
"<people><person><name>Jane</name></person><person><name>Bob</name></person></people>";
}
```

```
@GET
@Produces({"application/xml", "application/json"})
public MyJAXBType getPerson() {
    return new MyJAXBType();
}
```

- For generic types (i.e. String, byte arrays) that just copy bytes out to OutputStream, need to fill in the correct data for the response type chosen. Either via separate JAX-RS methods or via code inside a single method.
- For specific types (i.e. JAXB types, custom types) which have different MessageBodyWriters (one for JSON and one for XML), then can re-use same method

## How do the client and server agree on a response?

- If client sends a request, how does the server know what type to send back?
- Sometimes in the URL:
  - `http://www.example.com/File.xml` vs. `http://www.example.com/File.json`
  - `http://www.example.com/File?format=xml` vs. `http://www.example.com/File?format=json`
- More advanced via HTTP Accept header. Client sends an Accept header in request:
  - `Accept: application/xml;q=0.5, application/json;q=1.0; text/plain;q=0.8 */*;q=0.2`
    - “q” parameter is quality parameter. From 0.0 to 1.0. Higher value means more preferred.
    - Wildcards (\*) allow any type
    - `application/json, text/plain, application/xml`, and then anything else
- JAX-RS will respect Accept header
- Other Accept\* headers in Accept-Language, Accept-Charset

## How does JAX-RS know which message body provider to use?

- In the end, we have:
  - 1) Determined a media type to write as (either JSON, plain text, image, etc.).
  - 2) We also have the type of the returned object from the JAX-RS annotated method.
- Get all the registered `MessageBodyWriters`.
- Remove any that aren't media type compatible (i.e. if we're writing as text/plain, any `MessageBodyWriter` that only writes in application/xml is discarded)
- Remove any that aren't Java type compatible. (i.e. you want to write a `CustomType` but the `MessageBodyWriter` only writes Strings)
- Re-order the user defined ones ahead of the system defined ones
- Call each remaining provider's `isWritable` . The first one that returns true, use it.

## What else can I put on the response?

- Use `javax.ws.rs.core.Response` object to return additional information (i.e. headers)
- Uses builder design pattern

```
@GET
public Response getSimple() {
    return Response.ok("This would be the message body entity").build();
}

@GET
public Response getMore() {
    return Response.status(299).header("X-My-Custom-Header", "value")
        .entity("The message body").type("text/plain").build();
}
```

## How do I read the incoming client request entity?

- Java method is allowed one non-JAX-RS annotated parameter.
- Use `MessageBodyReaders` to turn incoming message body bytes into Java types

```
@POST
@Consumes("text/plain")
public Response postSomething(String requestEntity) throws Exception {
    /* do something with the entity */
    return Response.status(201).location(new
URI("http://www.example.com/collection/1234")).build();
}

@PUT
public Response totallyUnrelatedMethod(MyJAXBType requestEntity) throws Exception {
    /* do something with the entity */
    return Response.ok("Got the update").build();
}
```

## Is the message body the only way to pass information?

- Other types of parameters can be passed in a HTTP request to a server by annotating method parameters (like `@PathParam` on slide 37)
- Cookies, query parameters, matrix parameters, header parameters, etc.

```
@GET
public Object getSomething(@QueryParam("format") String format) {
    return /* something not */ null;
}

@POST
@Consumes("text/plain")
public Response postSomething(String requestEntity,
                              @HeaderParam("X-Custom-Header") int customHeader)
    throws Exception {
    /* do something with the entity */
    return Response.status(201)
        .location(new URI("http://example.com/collections/1234")).build();
}
```

## What about other types of parameters?

JAX-RS Annotation	Example
@QueryParam("q")	http://www.google.com/search?q=test
@HeaderParam("Content-Type")	(In HTTP Headers)
@MatrixParam("m1")	http://www.example.com/some/path;m1=abcValue
@CookieParam("c1")	(HTTP Cookie)
@PathParam("personID")	http://www.example.com/{personID} http://www.example.com/abcd
@FormParam("textBox1")	For web forms, the contents of the form are basically put together into a query encoded string and put in the request message body (textBox1=Name%20of%20feature&textBox2=Description&checkbox1=yes). Form param allows you to get the data easier.



## What else can help me build RESTful services?

- UriBuilder – Use API to “build” URIs from parts
- @Context injected objects (fields/parameters)
  - HttpHeaders – get the HTTP request headers
  - UriInfo – get the current URI (i.e. base URI of web application), parses the path
  - SecurityContext – get the current user of the thread
  - Request – helps determine what the client wants
  - Providers – gets the MessageBodyReaders/Writers available in application

```
@Context
private HttpHeaders requestHttpHeaders;

@GET
public String echoHeader() {
    return requestHttpHeaders.getRequestHeader("Content-Length").toString();
}

@POST
public String getNameOfUser(@Context SecurityContext secContext) {
    return secContext.getUserPrincipal().getName();
}
```

## What happens if an exception is thrown by some underlying code?

- By default, exception propagates to container (i.e. leads to JSP error pages).
- HTTP has concept of status codes (404 Not Found, 500 Internal Server Error)
- Status codes are easier for programmatic clients to parse
- In case something goes wrong in RESTful service, use ExceptionMapper:

```
@Provider
public class MyExceptionHandler implements ExceptionMapper<NullPointerException> {

    public Response toResponse(NullPointerException exception) {
        return Response.status(500)
            .entity("Whoops I forgot to check for null")
            .type("text/plain").build();
    }
}
```

## Application configuration (1)

- So far we've created these classes out of nowhere. How does WebSphere Application Server know they exist?
- Need to create a `javax.ws.rs.core.Application` sub-class and list relevant JAX-RS classes

```
package com.example;

public class MyApp extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(MyResource.class);
        s.add(MyProvider.class);
        return s;
    }
}
```

- Also need to configure the WAR file `web.xml`
  - Add servlet definition for JAX-RS Servlet
  - Pass parameter to JAX-RS Servlet for Application

## Application configuration (2) – web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
  version="3.0">
  <servlet>
    <servlet-name>MyRESTApplication</servlet-name>
    <servlet-class>com.ibm.websphere.jaxrs.server.IBMRestServlet</servlet-class>
    <init-param>
      <param-name>javax.ws.rs.Application</param-name>
      <param-value>com.example.MyApp</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyRESTApplication</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The JAX-RS runtime system  
servlet

Param-name =  
javax.ws.rs.Application  
Param-value = the  
Application subclass name

List each possible URL  
pattern, or use this /\*  
wild card registration

## New Features For JAX-RS 1.1

- The WebSphere Application Server Feature Pack for Web 2.0 introduced JAX-RS 1.0 support
  - Simultaneously released for WebSphere Application Server versions 6.1, 7.0, CE 2.0 and CE 2.1
  
- The WebSphere Application Server version 8.0 Beta upgraded to JAX-RS 1.1 support:
  - Feature Pack integrated into the core runtime
  - Integration with Java EE 6 technologies
    - Servlet 3.0
    - EJB 3.1
    - JCDI
    - JSR 250

## Servlet 3.0 Integration

- Servlet 3.0 no longer requires a web.xml file.
- For application configuration, can create an Application subclass with `@ApplicationPath` annotation to set the servlet mapping:

```
package com.example;

@ApplicationPath("/rest/")
public class MyApp extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> s = new HashSet<Class<?>>();
        s.add(MyResource.class);
        s.add(MyProvider.class);
        return s;
    }
}
```

- Alternatively, the JAX-RS runtime will scan for and automatically add all of your JAX-RS providers and resources to the application so no Application subclass is needed
  - Just package classes in WEB-INF/classes then configure the web.xml
  - Adding new resources is greatly simplified; no change to existing code is required!

## Servlet 3.0 Integration – V8.0 Beta web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd"
version="3.0">
  <servlet>
    <servlet-name>MyRESTApplication</servlet-name>
    <servlet-class>com.example.MyApp</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyRESTApplication</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

The Application subclass name can be supplied as the value of the servlet-class element within the servlet definition so no parameter is required

Can still list each possible URL pattern, or use /\* wild card registration

## EJB 3.1 Integration

- If you have enterprise JavaBeans (EJB) applications, you can expose a RESTful interface to the enterprise bean using JAX-RS
  - You keep the EJB functionality including transaction support, injection of Java EE components and resources, and other EJB session bean capabilities.
  - With EJB 3.1 you now have the option of exposing a local view of an enterprise bean without an explicit EJB local interface. Instead, the enterprise bean has a no-interface client view that is based on the public methods of your bean class.

```
package com.example;
```

```
@Path("/hello/{name}")
```

```
@Singleton
```

```
public class HelloWorldResource {
```

```
    @EJB
```

```
    Nickname nickname;
```

```
    @GET
```

```
    @Produces("text/plain")
```

```
    public String getResourceRepresentation(@PathParam("name") String name) {
```

```
        return "Hello " + name + ", aka " + nickname;
```

```
    }
```

```
}
```

Note the EJB annotations (in bold)  
used with JAX-RS annotations



## JCDI Integration

- Java Contexts and Dependency Injection (JCDI) is a new Java EE 6 feature
- It can change the programming model to make applications easier to develop while increasing maintainability
- JAX-RS developers can now use JCDI features, such as `@javax.inject.Inject` support, in root resource and provider classes

```
package com.example;
```

```
@Path("/hello/{name}")
```

```
public class HelloWorldResource {
```

```
    @Inject
```

```
    private DatabaseAdapter dbAdapter;
```

```
    @GET
```

```
    @Produces("text/plain")
```

```
    public String getResourceRepresentation(@PathParam("name") String name) {
```

```
        return "Hello " + name + ", aka " + dbAdapter.getNicknameFor(name);
```

```
    }
```

```
}
```

Note the JCDI `@Inject` annotation used with JAX-RS annotations

## JSR 250 Integration

- Secure JAX-RS resources using annotations for security supported by JSR 250 (Common Annotations for the Java Platform)
- Use the following annotations to add authorization semantics to JAX-RS resources:
  - `@PermitAll` - specifies that all security roles are permitted to access your JAX-RS resources
  - `@DenyAll` - specifies that no security roles are permitted to access your JAX-RS resources
  - `@RolesAllowed` - specifies the security roles that are permitted to access your JAX-RS resources

```
package com.example;
```

```
@Path("/hello/{name}")
```

```
public class HelloWorldResource {
```

```
    @Inject
```

```
    private DatabaseAdapter dbAdapter;
```

```
    /* hello world can now delete! */
```

```
    @DELETE
```

```
    @RolesAllowed("admin")
```

```
    public String delUserByName(@PathParam("name") String name) {
```

```
        dbAdapter.delete(name);
```

```
    }
```

```
}
```

Note the JSR 250 `@RolesAllowed` annotation used with JAX-RS annotations

# Agenda

- JAX-WS 2.2
  - WS-A and JAX-WS overview
  - New Features for JAX-WS 2.2
- JAX-RS 1.1
  - REST and JAX-RS overview
  - New Features for JAX-RS 1.1
- SOAP vs REST
- Summary and Resources

## SOAP vs REST

- REST:
  - Only supports HTTP transport
  - Re-use web development knowledge and technologies
  - Easier for “last leg” kinds of clients (user interfaces, browsers, mobile devices, etc.)
  - Need to handle quality of service issues in your code
  - Use for human users so they can respond to errors
  
- SOAP:
  - Transport protocol neutral e.g. can use over JMS
  - More mature technology
  - Strong interfaces, provable correctness, and management and migration of service interfaces
  - WSDL makes it possible to dynamically generate and validate code – speeds up development using graphical tools such as Rational Application Developer
  - Use for application integration without human supervision
  
- Use the best tool for the job, a hybrid may be sensible

## SOAP vs REST – Qualities of Service

- Reliable Messaging
  - REST applications can achieve a level of reliability by ensuring that:
    - Operations always send a response and are idempotent
    - Coding the application to retry the operation if it fails to receive a response
  - Otherwise WS-ReliableMessaging is needed
- Transactions
  - REST applications can be coded to:
    - Store the resources in a backend transactional system such as a database
    - Expose these transactions as resources themselves that are available to clients
  - The key is that the backend consists of a single transactional system
  - WS-Transaction specifications are needed for global/distributed transactions in order to coordinate transactions across a number of backend systems
- Security
  - REST (and SOAP) applications can achieve point-to-point security using TLS/SSL
  - Can also modify applications to use XML-ENC / XML-SIG directly for XML media types
  - WS-Security is really need for
    - End-to-end security if the intermediary is not trusted
    - Alternative transport bindings, HTTPS not available over JMS
    - Security tokens to convey information about the user to the service

# Agenda

- JAX-WS 2.2
  - WS-A and JAX-WS overview
  - New Features for JAX-WS 2.2
- JAX-RS 1.1
  - REST and JAX-RS overview
  - New Features for JAX-RS 1.1
- SOAP vs REST
- Summary and Resources

## Summary

- WebSphere Application Server V8.0 Beta adds support for JAX-WS 2.2 and JAX-RS 1.1
- JAX-WS 2.2 finishes support for WS-A 1.0 by incorporating the Metadata specification
- New features for WS-A:
  - Web service metadata in EPRs
  - New responses property on AddressingFeature and @Addressing annotations
  - Client side @Addressing annotations
  - Required property on AddressingFeature and @Addressing annotations also enforced on the client side
  - Enable and configure WS-A using WS-Policy in the WSDL
  - Enable/disable and configure WS-A using deployment descriptors (JSR 109 1.3)
  - Generate @Addressing, @Action and @FaultAction annotations in Java code from WS-A WS-Policy and wsam:Action attributes on WSDL operations
  - Publish WS-Policy for the runtime configuration taking all configuration methods into account
- JAX-RS provides an annotation-based approach to developing RESTful services, now fully integrated into the core runtime
- JAX-RS 1.1 adds integration with other Java EE 6 technologies such as Servlet 3.0, EJB 3.1, JCDI and JSR 250 security annotations

## Resources

- Specifications;
  - WS-Addressing 1.0:
    - Core – <http://www.w3.org/TR/ws-addr-core/>
    - SOAP Binding – <http://www.w3.org/TR/ws-addr-soap/>
    - Metadata – <http://www.w3.org/TR/ws-addr-metadata/>
  - JAX-WS 2.2 – <http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index3.html>
  - HTTP – <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
  - JAX-RS 1.1 – <http://jcp.org/aboutJava/communityprocess/mrel/jsr311/index.html>
- Products:
  - IBM WebSphere Application Server V8.0 Beta Program – <https://www14.software.ibm.com/iwm/web/cc/earlyprograms/websphere/wsasoa/>
  - WebSphere Application Server Feature Pack for Web 2.0 – <http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/web20/>
- JAX-RS developerWorks article – <http://www.ibm.com/developerworks/web/library/wa-jaxrs/>
- My email – [katherine\\_sanders@uk.ibm.com](mailto:katherine_sanders@uk.ibm.com)



Any Questions?