Putting Java to REST

Bill Burke
Fellow
JBoss, a division of Red Hat
bburke@redhat.com

Agenda

- **What is REST?**
- **Why REST?**
- **Writing RESTFul Web Services in Java**
  - **JAX-RS**

Speaker's Qualifications

- **RESTEasy project lead**
  - **Fully certified JAX-RS implementation**
- **JAX-RS JSR member**
  - **Also served on EE 5 and EJB 3.0 committees**
- **JBoss contributor since 2001**
  - **Clustering, EJB, AOP**
- **Published author**
  - **Books, articles**

What are the goals of SOA?

SOA Goals

- **Reusable**
- **Interoperable**
- **Evolvable**
  - **Versioning**
- **Governable**
  - **Standards**
  - **Architectural Guidelines and Constraints**
  - **Predictable**
- **Scalable**
- **Managable**

What system has these properties?

The Web!

What is REST?

- **REpresentational State Transfer**
  - **PhD by Roy Fielding**
- **REST answers the questions of**
  - **Why is the Web so prevalent and ubiquitous?**
  - **What makes the Web scale?**
  - **How can I apply the architecture of the web to my applications?**

What is REST?

- **REST is a set of architectural principles**
- **REST isn't protocol specific**
  - **But, usually REST == REST + HTTP**
- **A different way to look at writing Web Services**
  - **Many say it's the anti-WS-***
  - **In my experience, hard for CORBA or WS-* to accept/digest**

What is REST?

- **Addressable Resources**
  - **Every “thing” should have a URI**
- **Constrained interface**
  - **Use the standard methods of the protocol**
  - **HTTP: GET, POST, PUT, DELETE, etc.**
- **Representation Oriented**
  - **Different applications need different formats (AJAX + JSON**
- **Communicate statelessly**
  - **Stateless application scale**

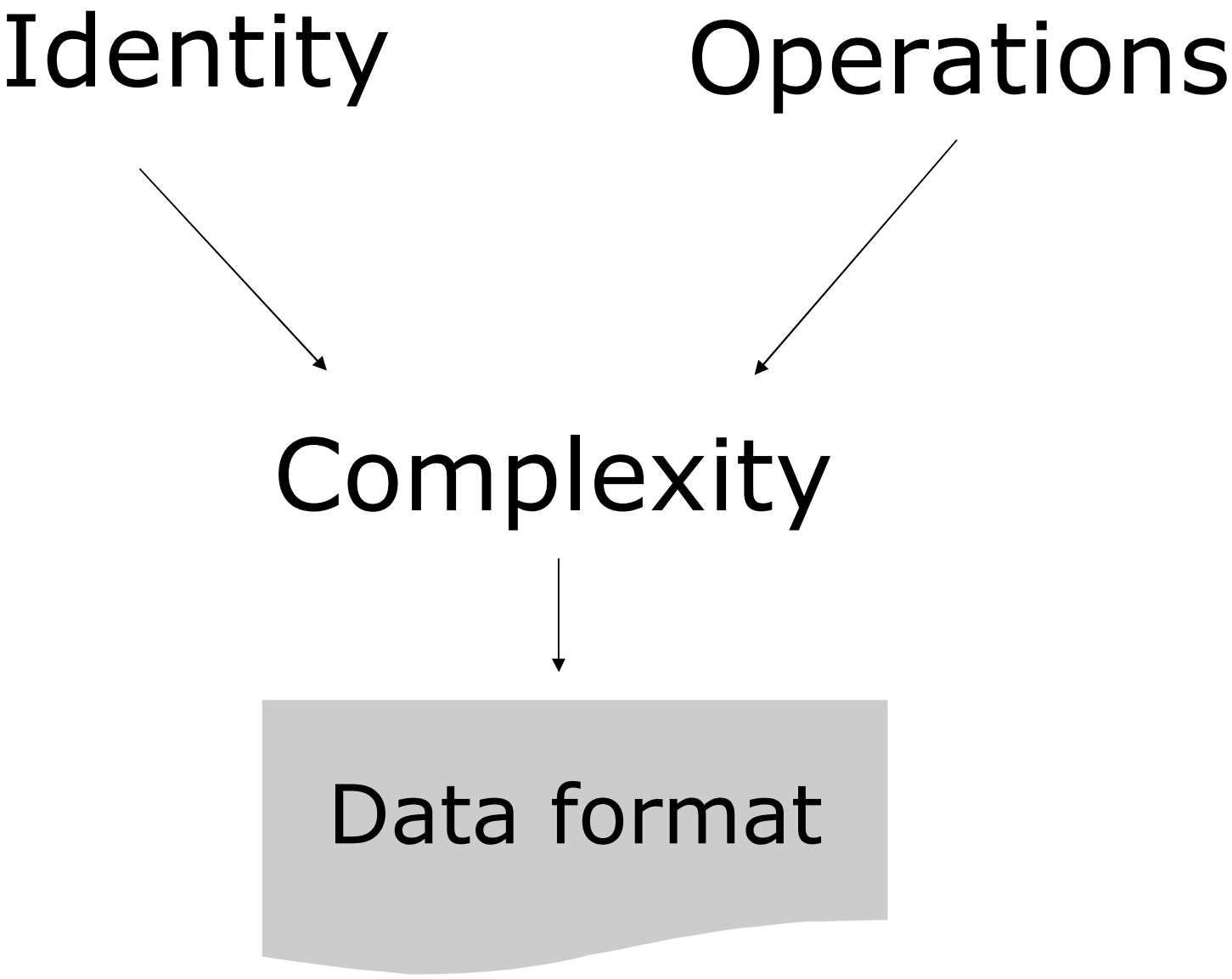
Addressability

- **Use URIs**
  - **Anybody that has used a browser understands URIs**
  - **Java EE has no standard addressability for components. Isn't that a portability headache?**
- **Linkability**
  - **Support finds a problem? Have them email you a URI that reproduces the problem**
  - **Resource representations have a standardized way of referencing other resource representations**
  - **Representations have a standardized way to compose themselves:**

```
<order id="111">
  <customer>http://sales.com/customers/32133</customer>
  <order-entries>
     <order-entry>
        <quantity>5</quantity>
        <product>http://sales.com/products/111</product>
…
```

Constrained, Uniform Interface

- **Hardest thing for those with CORBA and/or WS-* baggage to digest**
- **The idea is to have a well-defined, fixed, finite set of operations**
  - **Resources can only use these operations**
  - **Each operation has well-defined, explicit behavior**
  - **In HTTP land, these methods are GET, POST, PUT, DELETE**
- **How can we build applications with only 4+ methods?**
  - **SQL only has 4 operations: INSERT, UPDATE, SELECT, DELETE**
  - **JMS has a well-defined, fixed set of operations**
  - **Both are pretty powerful and useful APIs with constrained interfaces**


Identity
Operations
Complexity
Data format

Implications of Uniform Interface

- **Intuitive**
  - **You know what operations the resource will support**
- **Predictable behavior**
  - **GET - readonly and idempotent. Never changes the state of the resource**
  - **PUT - an idempotent insert or update of a resource. Idempotent because it is repeatable without side effects.**
  - **DELETE - resource removal and idempotent.**
  - **POST - non-idempotent, "anything goes" operation**
- **Clients, developers, admins, operations know what to expect**
  - **Much easier for admins to assign security roles**
  - **For idempotent messages, clients don't have to worry about duplicate messages.**

Implications of Uniform Interface

- **Simplified**
  - Nothing to install, maintain, upgrade
  - No stubs you have to generate distribute
  - No vendor you have to pay big bucks to
- **Platform portability**
  - HTTP is ubiquitous. Most (all?) popular languages have an HTTP client library
  - CORBA, WS-*, not as ubiquitous
  - (We'll talk later about multiple representations and HTTP content negotiation which also really helps with portability)
- **Interoperability**
  - HTTP a stable protocol
  - WS-*, again, is a moving target
  - Ask Xfire, Axis, and Metro how difficult Microsoft interoperability has been
  - Focus on interoperability between applications rather focusing on the interoperability between vendors.

Implications of Uniform Interface

- **Familiarity**
  - **Operations and admins know how to secure, partition, route, and cache HTTP traffic**
  - **Leverage existing tools and infrastructure instead of creating new ones**
- **Easily debugged**
  - **How cool is it to be able to use your browser as a debugging tool!**

Designing with Uniform Interface

```
public interface BankAccountService {
   Account getAccount(int id);
   void deleteAccount(int id);
   void updateAddress(int acct, Address address);
   void debit(double amount);
   void credit(double amount);

}
```

Designing with Uniform Interface

- **/accounts/{acct-id}**
  - **GET - retrieve representation of account**
  - **DELETE - remove an account**
- **Actions become things**
- **Update Address**
  - **/accounts/{acct-id}/address**
  - **PUT new XML representation of address**
- **Debit/Credit**
  - **Define a "Account Transaction" XML document**
  - **/accounts/{acct-id}/transactions**
  - **POST new XML representation of a credit or debit**

Representation Oriented

- URIs point to resources on the network
- Clients and servers exchange representations of a resource through the uniform interface
  - XML documents
  - JSON messages
- This is a familiar data exchange pattern for Java developers
  - Swing->RMI->Hibernate
  - Hibernate objects exchanged to and from client and server
  - Client modifies state, uses entities as DTOs, server merges changes
    - No different than how REST operates
  - No reason a RESTFul webservice and client can't exchange Java objects!

HTTP Negotiation

- **HTTP allows the client to specify the type of data it is sending and the type of data it would like to receive**
- **Depending on the environment, the client negotiates on the data exchanged**
  - **An AJAX application may want JSON**
  - **A Ruby application my want the XML representation of a resource**
  - **A server may want to serve up a CSV, MS Excel, or PDF representation of a resource**

HTTP Negotiation

- **HTTP Headers manage this negotiation**
  - **CONTENT-TYPE: specifies MIME type of message body**
  - **ACCEPT: comma delimited list of one or more MIME types the client would like to receive as a response**
  - **In the following example, the client is requesting a customer representation in either xml or json format**

```
GET /customers/33323
Accept: application/xml,application/json
```

  - **Preferences are supported and defined by HTTP specification**

```
GET /customers/33323
Accept: text/html;q=1.0,
        application/json;q=0.7;application/xml;q=0.5
```

HTTP Negotiation

- **Internationalization can be negotiated to**
  - **CONTENT-LANGUAGE: what language is the request body**
  - **ACCEPT-LANGUAGE: what language is desired by client**

```
GET /customers/33323
ACCEPT: application/xml
ACCEPT-LANGUAGE: en_US
```

Implications of Representations

- **Evolvable integration-friendly services**
  - **Common consistent location (URI)**
  - **Common consistent set of operations (uniform interface)**
  - **Slap on an exchange formats as needed**
- **Built-in service versioning**
  - **Add newer exchange format as an additional MIME type supported**
  - **application/vnd.myformat+xml**
  - **application/vnd.myformat-2+xml**
- **Internationalization becomes easy for clients**
  - **Most browsers can configure default ACCEPT-LANGUAGE**

Statelessness

- A RESTFul application does not maintain sessions/conversations on the server
- Doesn't mean an application can't have state
- REST mandates
  - That state be converted to resource state
  - Conversational state be held on client and transferred with each request
- Sessions are not linkable
  - You can't link a reference to a service that requires a session
- A stateless application scales
  - Sessions require replication
  - A simplified architecture is easier to debug
- Isolates client from changes on the server
  - Server topology could change during client interaction
  - DNS tables could be updated
  - Request could be rerouted to different machines

REST in Conclusion

- **REST answers questions of**
  - **Why does the Web scale?**
  - **Why is the Web so ubiquitous?**
  - **How can I apply the architecture of the Web to my applications?**
- **REST is tough to swallow**
  - **Make you rethink how you do things**
  - **Those with CORBA/WS-* baggage will resist (sometimes violently)**
- **Promises**
  - **Simplicity**
  - **Interoperability**
  - **Platform independence**
  - **Change resistance**

JAX-RS

RESTFul Web Services in Java

JAX-RS

- **JCP Specification**
  - **Lead by Sun, Marc Hadley**
  - **Finished in September 2008**
- **Annotation Framework**
- **Dispatch URI's to specific classes and methods that can handle requests**
- **Allows you to map HTTP requests to method invocations**
- **IMO, a beautiful example of the power of parameter annotations**
- **Nice URI manipulation functionality**

JAX-RS Annotations

- **@Path**
  - **Defines URI mappings and templates**
- **@Produces, @Consumes**
  - **What MIME types does the resource produce and consume**
- **@GET, @POST, @DELETE, @PUT, @HEAD**
  - **Identifies which HTTP method the Java method is interested in**

JAX-RS Parameter Annotations

- **@PathParam**
  - Allows you to extract URI parameters/named URI template segments
- **@QueryParam**
  - Access to specific parameter URI query string
- **@HeaderParam**
  - Access to a specific HTTP Header
- **@CookieParam**
  - Access to a specific cookie value
- **@MatrixParam**
  - Access to a specific matrix parameter
- **Above annotations can automatically map HTTP request values to**
  - String and primitive types
  - Class types that have a constructor that takes a String parameter
  - Class types that have a static valueOf(String val) method
  - List or Arrays of above types when there are multiple values
- **@Context**
  - Access to contextual information like the incoming URI

JAX-RS: GET /orders/3323

```
@Path(“/orders”)
public class OrderService {

    @Path(“/{order-id}”)
    @GET
    @Produces(“application/xml”)
    String getOrder(@PathParam(“order-id”) int id) {
     …
    }
}
```

JAX-RS Resource Classes

- **JAX-RS annotations are used on POJO classes**
- **The default component lifecycle is per-request**
  - **Same idea as @Stateless EJBs**
  - **Singletons supported too**
  - **EJB integration defined in EE 6**
  - **Most implementations have Spring integration**
- **Root resources identified via @Path annotation on class**

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
     …
    }
}
```

Base URI path to resource

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @ProduceMime("application/xml")
    String getOrder(@PathParam("order-id") int id) {
      …
    }
}
```

Additional URI pattern that getOrder() method maps to

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
      …
    }
}
```

Defines a URI path segment pattern

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
      …
    }
}
```

HTTP method Java getOrder() maps to

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
     …
    }
}
```

What's the CONTENT-TYPE returned?

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
      …
    }
}
```

Inject value of URI segment into the *id* Java parameter

JAX-RS: GET /orders/3323

```
@Path("/orders")
public class OrderService {

    @Path("/{order-id}")
    @GET
    @Produces("application/xml")
    String getOrder(@PathParam("order-id") int id) {
      …
    }
}
```

Automatically convert URI string segment into an integer

JAX-RS: POST /orders

```
@Path("/orders")
public class OrderService {

    @POST
    @Consumes("application/xml")
    void submitOrder(String orderXml) {
     …
    }
}
```

What CONTENT-TYPE is this method expecting from client?

JAX-RS: POST /orders

```
@Path("/orders")
public class OrderService {

    @POST
    @Consumes("application/xml")
    void submitOrder(String orderXml) {
      …
    }
}
```

Un-annotated parameters assumed to be incoming message body. There can be only one!

MessageBodyReader/Writers

- **JAX-RS can automatically (un)-marshall between HTTP message bodies and Java types**
  - **Method return value marshalled into HTTP response body**
  - **Un-annotated method parameter unmarshalled from HTTP message content**
- **JAX-RS has built-in MessageBodyReader/Writers**
  - **application/xml <-> JAXB annotated classes**
  - **text/* <-> String**
  - ***/* <-> byte[], java.io.InputStream, File, Reader**
  - **application/x-www-form-urlencoded <-> MultivaluedMap<String, String>**
  - ***/* <-> StreamingOutput, a JAX-RS specific streaming output interface**
- **Application can plug in custom MessageBodyReader/Writers**

MessageBodyReader

```
public interface MessageBodyReader<T>
{
 boolean isReadable(Class<?> type,
                    Type genericType,
                    Annotation annotations[]);

 T readFrom(Class<T> type, Type genericType,
            Annotation annotations[],
            MediaType mediaType,
            MultivaluedMap<String, String> httpHeaders,
            InputStream entityStream)
               throws IOException,
                      WebApplicationException;

}
```

MessageBodyWriter

```
public interface MessageBodyWriter<T>
{
  boolean isWriteable(Class<?> type,
                      Type genericType,
                      Annotation annotations[]);

  long getSize(T t);

  void writeTo(T t, Class<?> type, Type genericType,
               Annotation annotations[],
               MediaType mediaType,
               MultivaluedMap<String, Object> httpHeaders,
               OutputStream entityStream)
     throws IOException, WebApplicationException;
}
```

Writing MessageBodyReader/Writer

- **Must be annotated with @Provider**
- **MessageBodyReader must be annotated with @Consumes**
  - **To specify which MIME types it can convert to Java objects**
- **MessageBodyWriter must be annotated with @Produces**
  - **To specify which MIME types it can marshal Java objects to**
- **MessageBodyWriter.getSize()**
  - **Returning -1 will force chunk encoding**

Example MessageBodyReader

```
@Provider
@ConsumeMime({"application/xml", "text/xml"})
public class JAXBProviderReader implements
                                MessageBodyReader
{
  boolean isReadable(Class<?> type,
                     Type genericType,
                     Annotation annotations[])
  {
      return type.isAnnotationPresent(
                     XmlRootElement.class);
  }

  …
}
```

Example MessageBodyReader

```
Object readFrom(Class<Object> type, Type genericType,
          Annotation annotations[], MediaType mediaType,
          MultivaluedMap<String, String> httpHeaders,
          InputStream entityStream)
      throws IOException, WebApplicationException
  {
  try {
      JAXBContext jaxb = JAXBContext.newInstance(aClass);
      Object obj =
        jaxb.createUnmarshaller().unmarshal(inputStream);

        if (obj instanceof JAXBElement)
           obj = ((JAXBElement) obj).getValue();

        return obj;
     } catch (JAXBException e){
        throw new RuntimeException(e);
     }
  }
}
```

Default Response Codes

- **HTTP 1.1 specification defines response codes**
- **GET, DELETE and POST**
  - **200 (OK) if content sent back with response**
  - **204 (NO CONTENT) if no content sent back**

Response Object

- **JAX-RS has a Response and ResponseBuilder class**
  - **Customize response code**
  - **Specify specific response headers**
  - **Specify redirect URLs**
  - **Work with variants**

```
@GET
Response getOrder() {
   ResponseBuilder builder = Response.status(200);
   builder.type("text/xml")
          .header("custom-header", "33333");
   return builder.build();
}
```

JAX-RS Content Negotiation

- **@Produces can take array of producable MIME types**
  - **Matched up and chosen based on request ACCEPT header**
  - **Most JAX-RS implementations support weighted ACCEPT headers**
    - **I.e. Accept: text/html;q=1.0,application/xml;q=0.5**

ExceptionMappers

- **Map application thrown exceptions to a Response object**
  - **Implementations annotated by @Provider**

```
public interface ExceptionMapper<E>
{
    Response toResponse(E exception);
}
```

RESTFul Java Clients

RESTFul Java Clients

- **java.net.URL**
  - **Ugly, buggy, clumsy**
- **Apache HTTP Client**
  - **Full featured**
  - **Verbose**
  - **Not JAX-RS aware (MessageBodyReaders/Writers)**
- **Jersey and RESTEasy APIs**
  - **Similar in idea to Apache HTTP Client except JAX-RS aware**
- **RESTEasy Client Proxy Framework**
  - **Define an interface, re-use JAX-RS annotations for sending requests**

RESTEasy Client Proxy Framework

```
@Path("/customers")
public interface CustomerService {

   @GET
   @Path("{id})
   @Produces("application/xml")
   public Customer getCustomer(
         @PathParam("id") String id);
}


CustomerService service =
        ProxyFactory(CustomerService.class,
                        "http://example.com");

Customer cust = service.getCustomer("3322");
```

JAX-RS Example

Seeing it in action

RESTful JMS Facade

- **Let's define a simple RESTFul façade over a JMS queue**
  - **Store and forward asynch HTTP messages**
- **Work through REST resource design decisions**
  - **Introduce some new RESTful concepts**
- **Work through JAX-RS class design decisions**
  - **Introduce some other JAX-RS features**

RESTFul Interface

- **Sending a message to a queue**

```
POST /queues/{queue-name}?persistent=true
```

- **Receiving a message from the queue**

```
GET /queues/{queue-name}
```

JAX-RS Implementation

```
@Path("/queues/{name}")
public interface QueueService {

    @POST
    public void send(
            @PathParam("name") destination,
            @QueryParam("persistent")
                @DefaultValue("true") boolean persistent
            @Context HttpHeaders headers,
            InputStream body);



    @GET
    public Response receive(
            @PathParam("name") destination);

}
```

JAX-RS Implementation

```
@Path("/queues/{name}")
public interface QueueService {

    @POST
    public void send(
            @PathParam("name") destination,
            @QueryParam("persistent")
                @DefaultValue("true") boolean persistent
            @Context HttpHeaders headers,
            InputStream body);



    @GET
    public Response receive(
            @PathParam("name") destination);

}
```

Default value for an optional URI query parameter

JAX-RS Implementation

```
@Path("/queues/{name}")
public interface QueueService {

    @POST
    public void send(
            @PathParam("name") destination,
            @QueryParam("persistent")
                @DefaultValue("true") boolean persistent
            @Context HttpHeaders headers,
            InputStream body);

    @GET
    public Response receive(
            @PathParam("name") destination);

}
```

Access to all headers so we can forward them to receiver

Improvements to Send: Return created resource

- **When creating with a POST common pattern is to redirect to the created resource**
- **Status code 201 (Created)**
- **Redirect to a resource representing the message**
  - **Location: /queues/myQueue/messages/3334422**
  - **Subresources of this URI could be used to find out status of message**

Improvements to Send: Return created resource

```
@POST
public Response send(
            @PathParam("name") destination,
            @QueryParam("persistent")
                @DefaultValue("true") boolean persistent
            @Context HttpHeaders headers,
            @Context UriInfo uriInfo,
            InputStream body) {

        … create and post JMS message …

        URI messageUri = uriInfo.getAbsolutePathBuilder()
                .path(jmsMessage.getMessageID()).build();

        return Response.created(messageUri);
}
```

Improvements to Send: PUT instead of POST

- **What happens if there is a network failure during a client send of a message?**
  - **Client doesn't know if message successfully posted or not**
  - **It may up sending a duplicate message**
  - **POST is not idempotent**
- **Lets use PUT**
  - **Client generates unique message id**
  - **PUT /queues/{name}/messages/{message-id}**
  - **If a failure during PUT, resend**
  - **If message of that ID already there, no worries**

GET not Appropriate

- **HTTP 1.1 specification says GET is idempotent**
  - **Receiving messages with GET is not idempotent**
  - **It is changing the state of the resource**
  - **It is reading a message, but also consuming the queue**
- **Use POST for receiving**

GET not Appropriate

- **Problem, we are already are using POST for this resource**
- **Overload it?**
  - **POST /queues/{name}?action=[send|receive]**
  - **Ugly, it's a mini RPC**
  - **Doesn't map well to JAX-RS anyways**
- **When in doubt, create a resource**
  - **POST /queues/{name}/receiver**

Receiver gets message URI

- **Same idea as when sender get message URI**
- **Response code 200 (OK)**
- **Response header CONTENT-LOCATION**
  - **Means request processed ok, but here's a URI you can use**
  - **Content-Location: /queues/myQueue/messages/3334422**
  - **Can use URI to log bad messages**
  - **Can use URI to report bad messages**

One JAX-RS class not good design

- **Finding JMS ConnectionFactory and Destination not portable**
- **Separate finding the Destination from sending/receiving**
- **JAX-RS allows this through Subresources and Subresource Locators**
  - **One object processes part of the request**
  - **Another object finishes the request**

JAX-RS Implementation

```
@Path("/queues")
public class JBossDestinationLocator {

   @Path("/{name}")
   public QueueService findDestination(
               @PathParam("name") String name) {
       Destination destination = … find it …;
       return new QueueService(destination);
   }
}
public class QueueService {
   public QueueService(Destination dest) {…}

   @POST
   public void send(…) {}

   @Post
   @Path("/receiving")
   public Response receive(…) {…}
}
```

Why is this cool?

- **Platform independence**
  - **Can a Python client post messages?**
  - **Can a Ruby client receive messages?**
  - **Can a Java client post messages to a C++ receiver?**
- **Lightweight**
  - **Clients only need an HTTP library to use the queue**

JAX-RS Implementations

- **JBoss RESTEasy**
  - http://jboss.org/resteasy
  - Embeddable
  - Spring and EJB integration
  - Client Framework
  - Asynchronous HTTP abstractions
- **Jersey**
  - Sun reference implementation
  - WADL support
- **Apache CXF**
- **RESTLet**

References

- **Links**
  - http://jsr311.dev.java.net/
  - http://jboss.org/resteasy
  - http://rest.blueoxen.net/
  - http://java.dzone.com/articles/intro-rest
  - http://architects.dzone.com/articles/putting-java-rest **Books:**
- **Books**
  - **Coming this summer "RESTFul Java" by me**
  - **O'Reilly's "RESTful Web Services"**
  - http://oreilly.com/catalog/9780596529260/

Questions