QuickTime™ and a
decompressor
are needed to see this picture.

RESTful Java with JAX-RS

Bill Burke
Engineering Fellow
Red Hat

QuickTime™ and a
decompressor
are needed to see this picture.

Agenda

- Why REST?
- REST Principles
- Writing RESTFul Web Services in Java
  - JAX-RS

Speaker's Qualifications

QuickTime™ and a decompressor are needed to see this picture.

- 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

QuickTime™ and a
decompressor
are needed to see this picture.

What are the goals of SOA?

SOA Goals

QuickTime™ and a
decompressor
are needed to see this picture.

- Reusable
- Interoperable
- Evolvable
  - Versioning
- Scalable
- Manageable

QuickTime™ and a
decompressor
are needed to see this picture.

What system has these properties?

QuickTime™ and a
decompressor
are needed to see this picture.

The Web!

QuickTime™ and a
decompressor
are needed to see this picture.

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?

QuickTime™ and a
decompressor
are needed to see this picture.

- It can mean a simple, “lightweight”, distributed interface over HTTP
- REST is really a set of architectural principles
  - Principles that make the Web unique
- 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-*
- Rediscovery of HTTP

QuickTime™ and a
decompressor
are needed to see this picture.

Why REST?

- HTTP is everywhere
- Zero-footprint clients
  - A “Lightweight” stack
- “Lightweight” interoperability
- Evolvability
  - Link driven systems allow you to redirect easily
  - Content negotiation allows you to support old and new formats

REST Architectural Principles

QuickTime™ and a
decompressor
are needed to see this picture.

- Addressable Resources
- Representation Oriented
- Constrained interface
- Hypermedia and Link Driven
- Communicate statelessly

QuickTime™ and a
decompressor
are needed to see this picture.

Let’s build a RESTful interface!

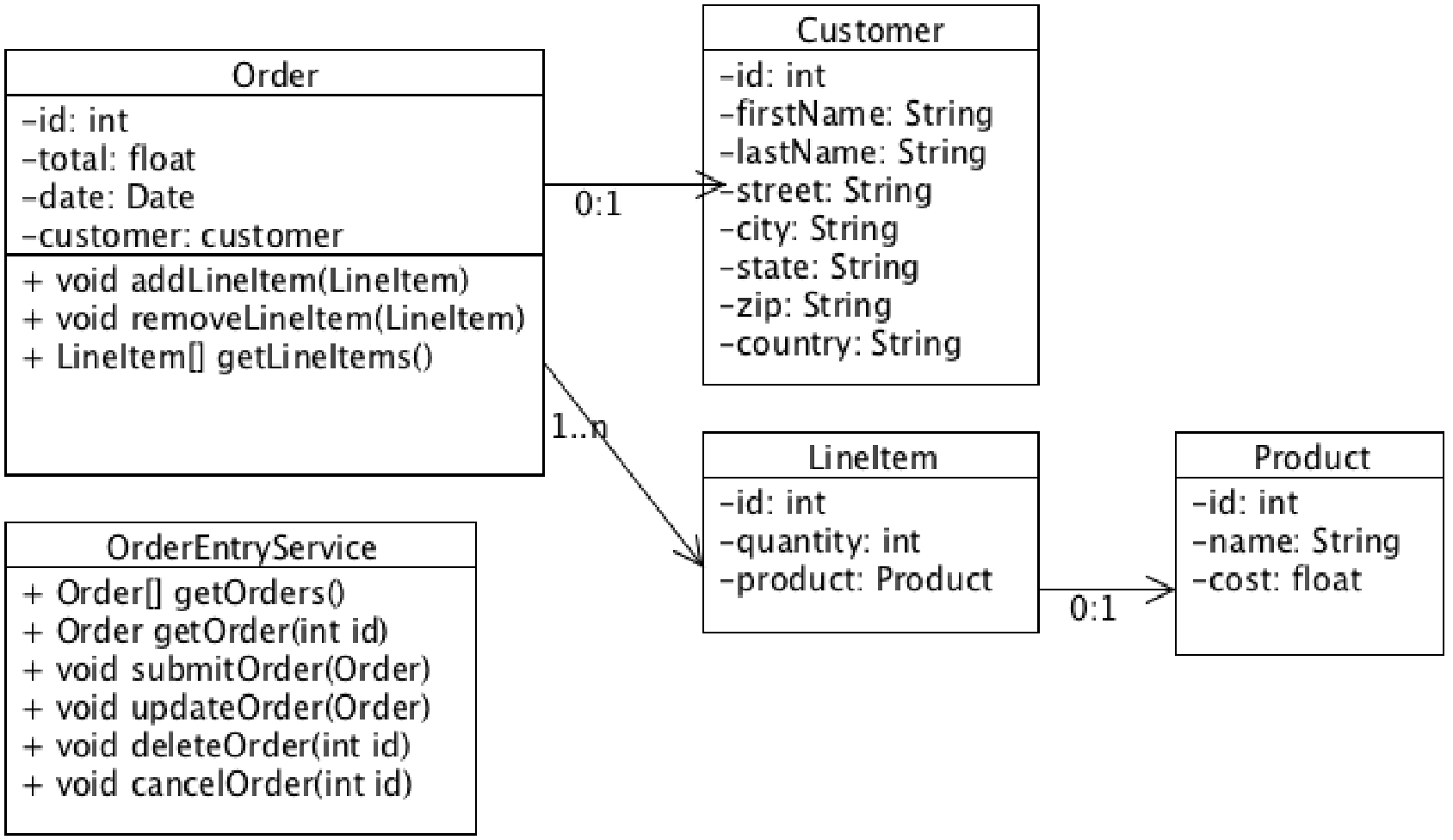
Building a RESTful Interface

QuickTime™ and a
decompressor
are needed to see this picture.

- We'll build a simple Order Entry System
- We'll apply each architectural principle as we design
- I'll describe the implications of each principle

Simple Order Entry System

QuickTime™ and a
decompressor
are needed to see this picture.



Addressable Resources

QuickTime™ and a
decompressor
are needed to see this picture.

- Resources are our endpoints in a RESTful interface
- The things in our object model become resources
  - Order
  - Customer
  - Product
- Each resource should have its own URI

QuickTime™ and a
decompressor
are needed to see this picture.

URI Scheme

- /orders
  - This URI represents all orders
  - We'll be able to query and create orders from this URI
- /orders/{id}
  - This URI represents one order
  - From this URI, we'll be able to read, update, and remove an order
  - {id} is a matching pattern. A wildcard.
- /orders/{id}/lineitems
  - We may or may not want to make lineitems addressable

QuickTime™ and a
decompressor
are needed to see this picture.

URI Scheme

- Similar URI Scheme for other objects
  - /customers
  - /customers/{id}
  - /products
  - /products/{id}

QuickTime™ and a
decompressor
are needed to see this picture.

Implications of Addressability

- Use HTTP's identification mechanism
  - WS-* usually has one URI you communicate through
  - WS-* requires tunnelling additional information about object identity through SOAP contexts.
- Allows for linking
- Enables the constrained interface (we'll see later)
- URI schemes should be an implementation detail
  - They should be opaque
  - Published via links (we'll see later)

Representation Oriented

QuickTime™ and a decompressor are needed to see this picture.

- Clients and servers exchange representations of a resource through the uniform interface (which we'll discuss later)
  - XML documents
  - JSON documents
- HTTP's Content-Type header identifies what we're exchanging
- 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!

Choosing a Representation

QuickTime™ and a
decompressor
are needed to see this picture.

- We'll choose XML
- Can add others as needed

QuickTime™ and a
decompressor
are needed to see this picture.

Customer XML

```
<customer id="771">
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
   <street>555 Beacon Str.</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
</customer>
```

Product XML

QuickTime™ and a
decompressor
are needed to see this picture.

```
<product id="543">
   <name>iPhone</name>
   <cost>$199.99</cost>
</customer>
```

QuickTime™ and a
decompressor
are needed to see this picture.

Order XML

```
<order id="133">
   <total>$199.99<total>
   <date>01/20/2010<date>
   <customer id="771">
      <first-name>Bill</first-name>
      <last-name>Burke</last-name>
      <street>555 Beacon Str.</street>
      <city>Boston</city>
      <state>MA</state>
      <zip>02115</zip>
   </customer>
   <line-items>
      <line-item>
         <product id="543">
            <name>iPhone</name>
            <cost>$199.99</cost>
         </product>
      </line-item>
   </line-items>
</customer>
```

Implications of Representations

QuickTime™ and a decompressor are needed to see this picture.

- Each URI can exchange multiple representations
- HTTP Content Negotiation allows clients and servers to choose what's best for them

QuickTime™ and a
decompressor
are needed to see this picture.

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

QuickTime™ and a
decompressor
are needed to see this picture.

HTTP Negotiation

- HTTP Headers manage this negotiation
  - 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
```

Implications of Representations

QuickTime™ and a
decompressor
are needed to see this picture.

- Evolvable integration-friendly services
  - Common consistent location (URI)
  - Common consistent set of operations (uniform interface)
  - Interactions defined, formats slapped on as needed
- Built-in service versioning
  - application/customers+xml;version=1
  - application/customers+xml;version=2

Constrained, Uniform Interface

QuickTime™ and a decompressor are needed to see this picture.

- The idea is to have a well-defined, fixed, finite set of operations
  - Clients 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

Constrained, Uniform Interface

QuickTime™ and a decompressor are needed to see this picture.

- GET - readonly operation
- PUT - used for insert or update of a resource
- DELETE - remove a resource
- POST - used for creation or as an “anything goes” operation
- GET, PUT, DELETE are idempotent
  - If you invoke same operation more than once, you should get the same result every time
- POST is not idempotent
  - Each POST can have a different effect on the resource

Read a Customer

QuickTime™ and a
decompressor
are needed to see this picture.

```
Request:
GET /customer/771 HTTP/1.1

Response:
HTTP/1.1 200 OK
Content-Type: application/xml

<customer id="771">
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
   <street>555 Beacon Str.</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
</customer>
```

QuickTime™ and a
decompressor
are needed to see this picture.

Update a Customer: Change address

```
Request:
PUT /customer/771 HTTP/1.1
Content-Type: application/xml

<customer id="771">
   <first-name>Bill</first-name>
   <last-name>Burke</last-name>
   <street>101 Dartmouth Str.</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
</customer>
```

Creation

QuickTime™ and a
decompressor
are needed to see this picture.

- There is a common pattern for creation
- POST to a top resource URI
- Get back the location (URI) of created resource
  - Response contains a Location header

QuickTime™ and a
decompressor
are needed to see this picture.

Create a Customer

```
Request:
POST /customers HTTP/1.1
Content-Type: application/xml

<customer>
   <first-name>Monica</first-name>
   <last-name>Burke</last-name>
   <street>101 Dartmouth Str.</street>
   <city>Boston</city>
   <state>MA</state>
   <zip>02115</zip>
</customer>

Response:
HTTP/1.1 201 Created
Location: http://example.com/customers/2322
```

When 4 methods don’t fit

QuickTime™ and a
decompressor
are needed to see this picture.

- What operations are required on Orders?
  - Create - POST on /orders
  - Read - GET on /orders/{id}
  - Update - PUT on /orders/{id}
  - Remove - DELETE on /orders/{id}
  - Cancel?

Operations modeled as state

QuickTime™ and a
decompressor
are needed to see this picture.

- Can Cancel be modeled as state?
  - Yes, cancelled is a state of the order
  - Let’s add a <cancelled> element to our representation
  - The act of cancelling becomes an update of the representation

Cancel an Order

QuickTime™ and a
decompressor
are needed to see this picture.

```
Request:
PUT /order/331 HTTP/1.1
Content-Type: application/xml

<order id="331">
    <total>$199.99</total>
    <date>01/20/2010</date>
    <cancelled>true</cancelled>
    …
</order>
```

QuickTime™ and a
decompressor
are needed to see this picture.

Operations not modeled as state

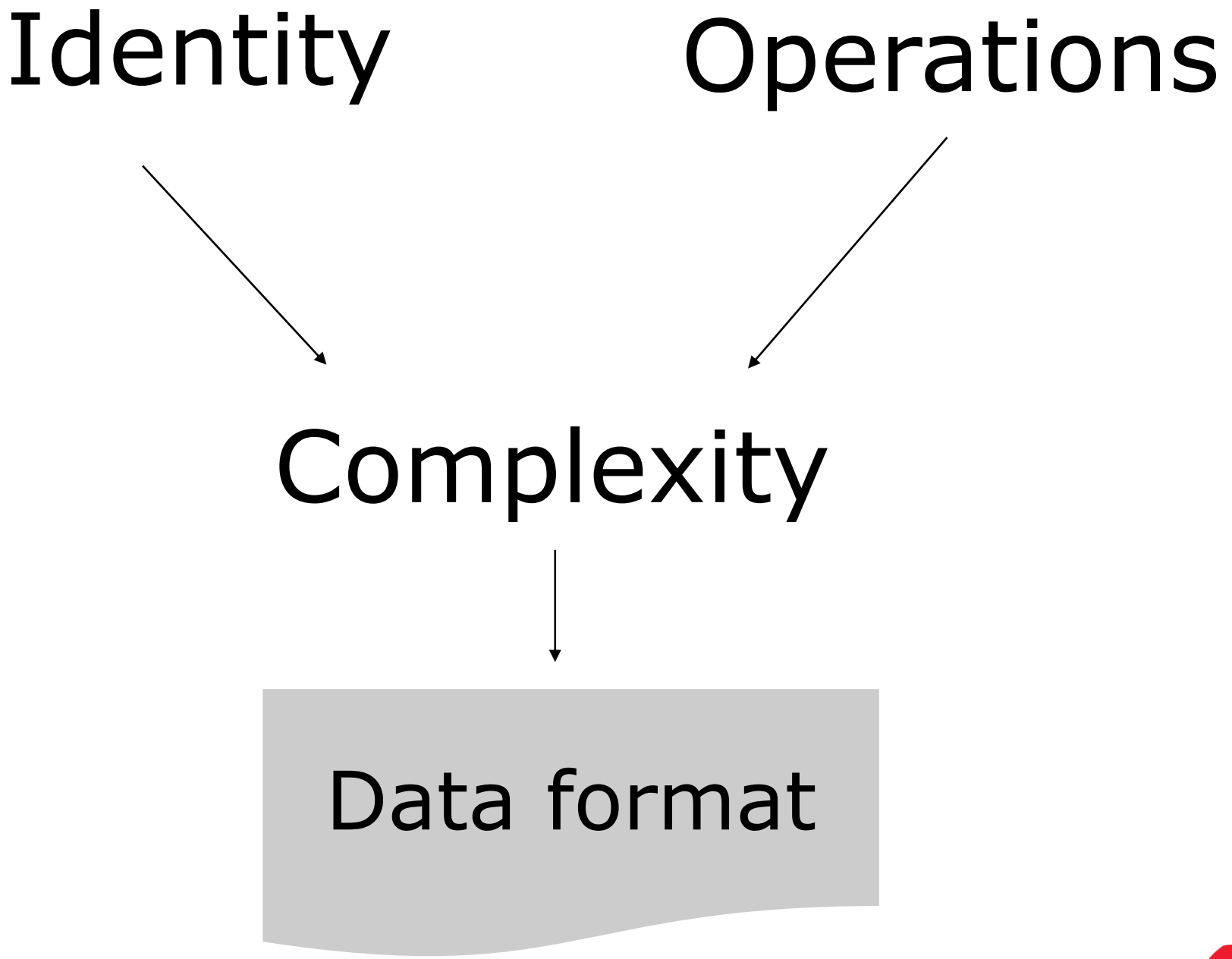
- What if an operation can’t be modeled as state?
- Example: order purging
  - Remove all cancelled orders.
- In this case, define a new resource:
  - /orders/purge
- POST or PUT to this resource

Implications of Uniform Interface

QuickTime™ and a
decompressor
are needed to see this picture.

- Simplified
  - No stubs you have to generate distribute
  - Nothing to install, maintain, upgrade
  - No vendor you have to pay big bucks to

QuickTime™ and a
decompressor
are needed to see this picture.



Implications of Uniform Interface

QuickTime™ and a decompressor are needed to see this picture.

- Interoperability
  - HTTP a stable protocol
  - WS-*, again, is a moving target
  - Ask CXF, Axis, and Metro how difficult Microsoft interoperability has been
  - Focus on interoperability between applications rather focusing on the interoperability between vendors.
- Familiarity
  - Operations and admins know how to secure, partition, route, and cache HTTP traffic
  - Leverage existing tools and infrastructure instead of creating new ones

Hypermedia, or rather Links

QuickTime™ and a
decompressor
are needed to see this picture.

- Links drive interactions
  - When a human uses a browser
  - No idea what the URI scheme is beforehand
  - Human just follows links
  - Google follows links to create search indexes

QuickTime™ and a
decompressor
are needed to see this picture.

Implications of Links

- Links allow you to compose data

```
<order id="133">
   <total>$199.99<total>
   <date>01/20/2010<date>
   <link rel="customer"
         href="http://example.com/customers/771"/>
   <line-items>
      <line-item>
         <link rel="product"
               href="http://example.com/products/543"/>
      </line-item>
   </line-items>
</customer>
```

QuickTime™ and a
decompressor
are needed to see this picture.

Implications of Links

- Links allow URIs to become opaque
- URIs become an implementation detail
- RESTful systems usually have very few published URIs
- URIs schemes can change without breaking clients

QuickTime™ and a
decompressor
are needed to see this picture.

Implications of Links

- One URL for Order Entry System
- Query base URI, then traverse links to interact

```
Request:
GET /order-entry-system HTTP/1.1

Response:
HTTP/1.1 200 OK
Content-Type: application/xml

<services>
   <link rel="orders" href="http://…"/>
   <link rel="customers" href="http://…"/>
   <link rel="products" href="http://…"/>
</services
```

QuickTime™ and a
decompressor
are needed to see this picture.

Statelessness

- A RESTFul web service does not maintain sessions/conversations on the server
- Doesn’t mean a web service can’t have state
- REST mandates
  - That state be converted to resource state
  - Conversational state be held on client and transferred with each request

QuickTime™ and a
decompressor
are needed to see this picture.

Statelessness

- Sessions are not linkable
  - You can’t link a reference to a service that requires a session
- A stateless application scales
  - Sessions require replication
  - Stateless services only require load balancing

REST in Conclusion

QuickTime™ and a decompressor are needed to see this picture.

- 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?
- Promises
  - Simplicity
  - Zero-footprint clients.
  - Interoperability
  - Platform independence
  - Change resistance

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS

RESTFul Web Services in Java

JAX-RS

QuickTime™ and a
decompressor
are needed to see this picture.

- JCP Specification
  - Required in Java EE 6
- Annotation Framework
- Allows you to map HTTP requests to Java method invocations

JAX-RS: GET /orders/3323

QuickTime™ and a
decompressor
are needed to see this picture.

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

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

JAX-RS Annotations

QuickTime™ and a decompressor are needed to see this picture.

- @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

QuickTime™ and a decompressor are needed to see this picture.

- @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

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

Base URI path to resource

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

Additional URI pattern that getOrder() method maps to

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

Defines a URI path segment pattern

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

HTTP method Java getOrder() maps to

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

What's the CONTENT-TYPE returned?

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

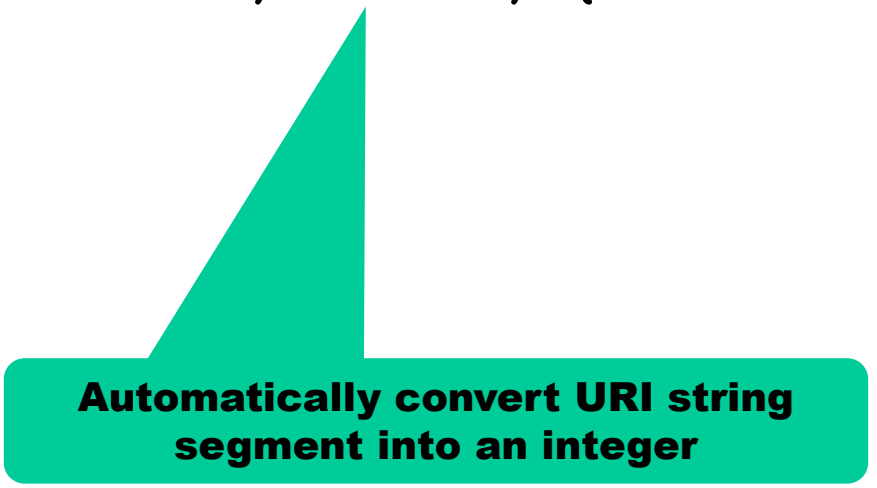
Inject value of URI segment into the *id* Java parameter

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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



QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: GET /orders/3323

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

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

Content handlers can convert from Java to Data Format

JAX-RS: POST /orders

QuickTime™ and a decompressor are needed to see this picture.

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

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

What CONTENT-TYPE is this method expecting from client?

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: POST /orders

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

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

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

QuickTime™ and a
decompressor
are needed to see this picture.

JAX-RS: POST /orders

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

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

Content handlers can convert from data format into Java object

More on Content Handlers

QuickTime™ and a decompressor are needed to see this picture.

- Media type, annotations, object type are all used to find a handler

```
@XmlElement
public class Order {
…
}

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

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

More on Content Handlers

QuickTime™ and a
decompressor
are needed to see this picture.

- JAXB and other simple types required by specification
- JSON? Jackson project is a great provider
- Atom, multipart, XOP and other formats available
- You can write your own custom ones

QuickTime™ and a
decompressor
are needed to see this picture.

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, order);
   builder.type("text/xml")
          .header("custom-header", "33333");
   return builder.build();
}
```

JAX-RS Content Negotiation

QuickTime™ and a
decompressor
are needed to see this picture.

- Matched up and chosen based on request ACCEPT header
  - Accept: application/json;q=1.0,application/xml;q=0.5

```
@GET
@Produces("application/xml")
String getXmlOrder() {…}

@GET
@Produces("application/json")
String getJsonOrder() {…}
```

ExceptionMappers

QuickTime™ and a
decompressor
are needed to see this picture.

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

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

QuickTime™ and a
decompressor
are needed to see this picture.

RESTFul Java Clients

QuickTime™ and a
decompressor
are needed to see this picture.

RESTFul Java Clients

- java.net.URL
  - Ugly, buggy, clumsy
- Apache HTTP Client
  - Full featured
  - Verbose
  - Not JAX-RS aware
- 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

QuickTime™ and a decompressor are needed to see this picture.

```
@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");
```

QuickTime™ and a
decompressor
are needed to see this picture.

RESTEasy

- Embeddable
- Spring, EJB, Guice, and Seam integration
- Client Framework
- Asynchronous HTTP (COMET)
- Client and Server Side Caching
- Interceptor model
- GZIP encoding support
- Data format support
  - Atom, JAXB, JSON, Multipart, XOP

JAX-RS Conclusions

QuickTime™ and a
decompressor
are needed to see this picture.

- Mapping HTTP requests using annotations
- A la carte HTTP information
- Nice content handlers
- Nice routing

References

QuickTime™ and a
decompressor
are needed to see this picture.

- Links
  - http://jsr311.dev.java.net/
  - http://jboss.org/resteasy
  - http://rest-star.org
- O'Reilly Books
  - "RESTFul Java with JAX-RS" by me
  - "RESTful Web Services"
  - "RESTful Web Services Cookbook"

QuickTime™ and a
decompressor
are needed to see this picture.