

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 is REST?

- REpresentational State Transfer
 - ✓ PhD by Roy Fielding
 - ✓ The Web is the most successful application on the Internet
 - ✓ What makes the Web so successful?
- 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?

- REST is a set of architectural principles
- REST isn't protocol specific
 - ✓ But, usually REST == REST + HTTP
- 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?

- Addressable Resources
 - ✓ Every “thing” should have a URI
 - ✓ Every “thing” should be referenceable
- Constrained interface
 - ✓ Use the standard methods of the protocol
 - ✓ HTTP: GET, POST, PUT, DELETE, etc.
- Resources with multiple representations
 - ✓ Different applications need different formats
 - ✓ Different platforms need different representations (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>
```

...

Describing a URI

- Human readable URIs: Desired but not required
- URI Parameters

```
http://sales.com/customers/323421  
/customers/{customer-id}
```

- Query parameters to find other resources

```
http://sales.com/customers?zip=02115
```

- Matrix parameters to define resource attributes

```
http://sales.com/cars/mercedes/amg/e55;color=black
```

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

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 OrderEntryService {

    void submitOrder(Order order);
    Order[] getOrders();
    void updateOrder(Order order);
    void cancelOrder(int orderId);
    Order[] getCustomerOrders(Customer customer);
    double calculateAverageSale();
}

public interface CustomerService {

    void createCustomer(Customer cust);
    void deleteCustomer(int custId);
    Customer[] getCustomers();
    Customer findCustomer(String first, String last);
}
```

Designing with Uniform Interface

- When in doubt, define a new resource
- /orders
 - ✓ GET - list all orders
 - ✓ POST - submit a new order
- /orders/{order-id}
 - ✓ GET - get an order representation
 - ✓ PUT - update an order
 - ✓ DELETE - cancel an order
- /orders/average-sale
 - ✓ GET - calculate average sale
- /customers
 - ✓ GET - list all customers
 - ✓ POST - create a new customer
- /customers/{cust-id}
 - ✓ GET - get a customer representation
 - ✓ DELETE - remove a customer
- /customers/{cust-id}/orders
 - ✓ GET - get the orders of a customer

Resources and Representations

- 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 may 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

RESTFul Web Services in Java

JAX-RS

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 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) {
        ...
    }
}
```

JAX-RS: GET /orders/3323

`@Path("/orders")`

Base URI path to resource

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

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 {

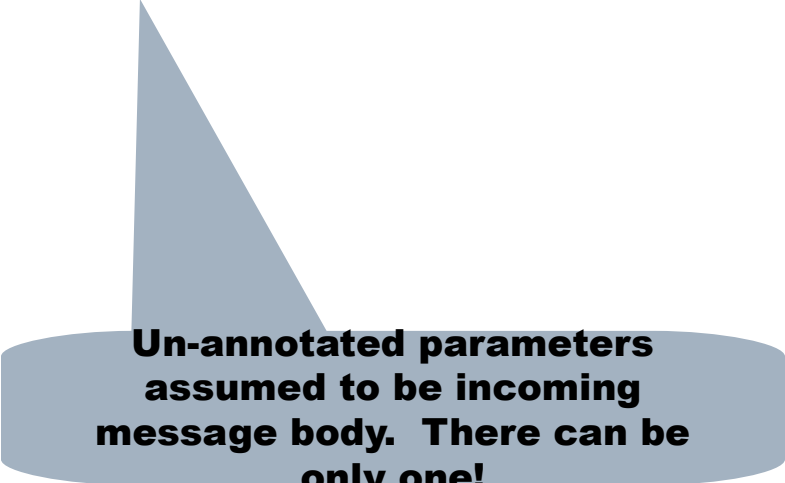
    @POST
    @Consumes("application/xml")
    void submitOrder(String orderXml) {
        ...
    }
}
```

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;
}
```

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

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
@Consumes({ "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);
    }
}
```

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 producible 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
- Media type extension mappings (used to be in spec)
 - ✓ Can map file extensions to media types
 - ✓ GET /customers/3333.xml
 - Inserts application/xml into ACCEPT header
- Language extension mappings (used to be in spec)
 - ✓ Can map file extensions to a specific desired language
 - ✓ GET /customers/3333.html.en_US
- Both extension mappings very useful for clients that cannot specify ACCEPT headers
 - ✓ a.k.a Browsers!

ExceptionMappers

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

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

Seeing it in action

JAX-RS Example

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
- Receiving a message from the queue

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

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

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 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://www.infoq.com/articles/rest-introduction>
 - ✓ <http://www.infoq.com/articles/tilkov-rest-doubts>
 - ✓ <http://rest.blueoxen.net/>
 - ✓ <http://bill.burkecentral.com/2007/09/18/distributed-compensation-w>
- Books:
 - ✓ Coming this summer "RESTful Java" by me
 - ✓ O'Reilly's "RESTful Web Services"
 - ✓ <http://oreilly.com/catalog/9780596529260/>

Questions

JBoss EJB 3.0

J2EE Evolves

Overall Agenda

- Overview
- Session Beans
- Interceptors
- Entity Beans
- JBoss Extensions

Goals of EJB 3.0

- EJB 2.1 is too “noisy”
 - ✓ Too many interfaces to implement
 - ✓ “XML Hell” too many complex deployment descriptors
 - ✓ API is too verbose
 - ✓ API too complicated in general
- Simplify the EJB programming model
- Focus on ease of use
- Facilitate Test Driven Development
- Make it simpler for average developer
- Increase developer base

Session Beans

Compare EJB 2.1 vs. EJB 3.0

EJB 2.1: Requirements

- Home interface
- Remote/Local interface
- Bean class must implement `javax.ejb.SessionBean`
- XML Deployment descriptor

EJB 2.1: Required Interfaces

- Homes for stateless beans unnecessary
- Remote interface must inherit from EJBObject
- Remote methods must throw RemoteException
 - ✓ Dependency on RMI

```
public interface CalculatorHome extends javax.ejb.EJBHome {  
    public Calculator create() throws CreateException;  
}
```

```
public interface Calculator extends EJBObject {  
    public int add(int x, int y) throws RemoteException;  
    public int subtract(int x, int y) throws RemoteException;  
}
```

EJB 2.1: Bean class

- Must extend verbose `javax.ejb.SessionBean`
- Unnecessary and verbose callback methods

```
public class CalculatorBean implements javax.ejb.Sessionbean {
    private SessionContext ctx;
    public void setSessionContext(SessionContext ctx) { this.ctx = ctx; }

    public void ejbCreate() { }
    public void ejbRemove() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }

    public int add(int x, int y) {
        return x + y;
    }

    public int subtract(int x, int y) {
        return x - y;
    }
}
```

EJB 2.1: XML Descriptor

```
<session>  
  <ejb-name>CalculatorBean</ejb-name>  
  <home>org.acme.CalculatorHome</home>  
  <bean>org.acme.CalculatorBean</bean>  
  <remote>org.acme.CalculatorRemote</remote>  
  <session-type>Stateless</session-type>  
  <transaction-type>Container</transaction-type>  
</session>
```

....

EJB 3.0: Goals

- Remove unnecessary interfaces
- Remove unnecessary callbacks
- Make deployment descriptors optional
- Make beans pojo-like
- Use default values where they make sense

Required Interfaces

- Homeless
- Methods don't throw RemoteException
- No verbose interface implementations

```
@Remote public interface Calculator {  
    public int add(int x, int y);  
    public int subtract(int x, int y);  
}  
  
@Stateless public class CalculatorBean implements Calculator {  
    public int add(int x, int y) {  
        return x + y;  
    }  
    public int subtract(int x, int y) {  
        Return x - y;  
    }  
}
```

EJB 3.0: XML Descriptor

Stateful Beans

- Still homeless
 - ✓ Created as they are looked up
- @Remove replaces EJBObject.remove
- Stateful bean is removed after method called

```
@Remote public interface ShoppingCart {  
    public void addItem(int prodId, int quantity);  
    public void checkout();  
}  
  
@Stateful public class ShoppingCartBean implements ShoppingCart {  
  
    @Remove  
    public void checkout() {  
        ...  
    }  
}
```

MDBs

- Just implements MessageListener
- XML turns to annotations

```
@MessageDriven(  
    activationConfig={ActivationConfigProperty(name="destination",  
value="queue/email")})  
public class EmailBean implements MessageListener {  
  
    void onMessage(Message msg) { }  
  
}
```

Transactions and Security

```
@Stateful public class ShoppingCartBean implements ShoppingCart {  
  
    @Remove  
    @TransactionAttribute(NEVER)  
    @MethodPermission({"valid_customer"})  
    public void checkout() {  
        ...  
    }  
}
```

EJB 3.0: Dependency Injection

- Bean class specifies dependencies instead of lookup
- Facilitates Test Driven Development
- Possible to test EJBs outside of container

```
@Stateful public class ShoppingCartBean implements ShoppingCart {  
  
    @Resource private SessionContext ctx;  
  
    @EJB  
    private CreditCardProcessor processor;  
  
    private DataSource jdbc;  
  
    @Resource(mappedName="java:/DefaultDS")  
    public void setDataSource(DataSource db) { this.jdbc = db; }  
  
}
```

EJB 3.0: Callbacks on demand

- Callback methods still available
- Annotate on an as-needed basis

```
@Stateless public class FacadeBean implements Facade {  
    @Timeout void licenseExpired(Timer t) {...}  
  
    @PostActivate public void initialize() {}  
}
```

EJB 3.0 Deployment Descriptor

- Believe it or not, people like XML deployment descriptors
- Externalize configuration
- Externalize system architecture

- Replace annotations with XML and you get pure POJOs

EJB 3.0 Deployment Descriptors

```
@Remote public interface ShoppingCart {  
    public void addItem(int prodId, int quantity);  
    public void checkout();  
}  
@Stateful public class ShoppingCartBean implements ShoppingCart {  
  
    @Remove  
    @TransactionAttribute(REQUIRED)  
    @MethodPermission({"valid_customer"})  
    public void checkout() {  
        ...  
    }  
}
```

EJB 3.0 Deployment Descriptors

```
public interface ShoppingCart {  
    public void addItem(int prodId, int quantity);  
    public void checkout();  
}  
  
public class ShoppingCartBean implements ShoppingCart {  
  
    public void checkout() {  
        ...  
    }  
}
```

XML Descriptors

- Partial deployment descriptors supported
 - ✓ Start off with annotations, configure as needed

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ShoppingCartBean</ejb-name>
      <ejb-local-ref>
        <ejb-ref-name>processor</ejb-ref-name>
        <local>com.titan.CreditCardProcessor</local>
      </ejb-local-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

EJB3 Interceptors

Standardized AOP

EJB 3.0 Interceptor Overview

- EJB 3.0 formalizes interceptors
 - ✓ already available in proprietary products
- Intercept incoming business method in container
 - ✓ Server side only!
- Intercept EJB lifecycle events

Purpose of EJB Interceptors

- Aspectizing your applications
- Pluggable annotations
- Ease of Extension
 - ✓ Not just ease of use
 - ✓ Framework for frameworks

Method Profiling

```
@Stateless
public class BankAccountBean implements BankAccount {

    @PersistenceContext EntityManager entityManager;

    public void withdraw(int acct, double amount) {
        long start = System.currentTimeMillis();
        try {
            Account account = entityManager.find(...);
            validateWithdrawal(account, amount);
            account.withdraw(amount);
            entityManager.flush();
        } finally {
            long time = start - System.currentTimeMillis();
            System.out.println("withdraw took " + time);
        }
    }
}
```

What's wrong with example?

- Bloated code
 - ✓ Profiling has nothing to do with business logic
- Difficult to enable/disable profiling
- Impossible to extend profiling behavior transparently

Interceptors to the rescue

- Provides structure where none exists in OOP
- Can encapsulate profiling logic in one class
 - ✓ Easier to extend
 - ✓ Easier to debug
- Facilities to transparently apply profiling logic
 - ✓ Easy to apply profiling logic

Interceptor implementation

```
public class ProfilingInterceptor {  
  
    @AroundInvoke  
    public Object profile(InvocationContext invocation)  
        throws Exception  
    {  
        long start = System.currentTimeMillis();  
        try {  
  
            return invocation.proceed();  
  
        } finally {  
            long time = start - System.currentTimeMillis();  
            Method method = invocation.getMethod();  
            System.out.println(method.toString() +  
                " took " + time + " (ms)");  
        }  
    }  
}
```

@AroundInvoke method

- Intercepts method being invoked
- Called in chain of other applied interceptors
- In same Java call stack as bean method
 - ✓ Wraps around
 - ✓ Thrown bean exceptions may be caught
- InvocationContext abstraction class for invoked method
 - ✓ Same idea as JBoss AOP Invocation interface

javax.interceptor.InvocationContext

```
public interface InvocationContext {  
    public Object getTarget();  
    public Method getMethod();  
    public Object getParameters();  
    public void setParameters(Object[] args);  
    public Map<String, Object> getContextData();  
    public Object proceed() throws Exception;  
}
```

Must always be called at the end of the interceptor implementation in order for the invocation to proceed.

Using the invocation

```
public class ProfilingInterceptor {  
  
    @AroundInvoke  
    public Object profile(InvocationContext invocation)  
        throws Exception  
    {  
        long start = System.currentTimeMillis();  
        try {  
  
            return invocation.proceed();  
  
        } finally {  
            long time = start - System.currentTimeMillis();  
            Method method = invocation.getMethod();  
            System.out.println(method.toString() +  
                " took " + time + " (ms)");  
        }  
    }  
}
```

Interceptor details

- Run in same tx and security context as method
- Interceptor has same lifecycle as EJB
 - ✓ Interceptor instance created per EJB instance
 - ✓ Pooled along with bean as well
 - ✓ Destroyed when its EJB instance is destroyed
 - ✓ Side effect? They can hold state
- Support XML and annotation driven injection
- Belong to the same ENC as EJB

Evolving the Profiler example

```
public class ProfilingInterceptor {  
    @Resource SessionContext ctx;  
    @PersistenceContext EntityManager manager;  
  
    @AroundInvoke  
    public Object profile(InvocationContext invocation)  
        throws Exception {  
        long start = System.currentTimeMillis();  
        try {  
  
            return invocation.proceed();  
  
        } finally {  
            long time = start - System.currentTimeMillis();  
            Profile prof = new Profile(  
                time, invocation.getMethod(),  
                ctx.getPrincipal()  
            );  
            manager.persist(prof);  
        }  
    }  
}
```

Inject and use an EntityManager to persist profiling information

Evolving the Profiler example

```
public class ProfilingInterceptor {  
  
    @EJB Profiler profiler;  
  
    @AroundInvoke  
    public Object profile(InvocationContext invocation)  
        throws Exception  
    {  
        long start = System.currentTimeMillis();  
        try {  
  
            return invocation.proceed();  
  
        } finally {  
            long time = start = System.currentTimeMillis();  
            profiler.log(invocation, time);  
        }  
    }  
}
```

Delegate profiling logic to
injected EJB

XML Equivalent

```
<ejb-jar>  
  <interceptors>  
    <interceptor>  
      <interceptor-class>  
        com.titan.ProfilingInterceptor  
      </interceptor-class>  
      <ejb-local-ref>  
        ...  
      </ejb-local-ref>  
    </interceptor>  
  </interceptors>  
</ejb-jar>
```

Applying Interceptors

- Through annotations
 - ✓ `@javax.interceptors.Interceptors`
- Through explicit XML
 - ✓ `ejb-jar.xml`
- Through default XML
 - ✓ Default interceptors

Applying Interceptors

Accepts an array of classes

```
@Stateless
@Interceptors(ProfilingInterceptor.class)
public class BankAccountBean implements BankAccount {

    @PersistenceContext EntityManager entityManager;

    public void withdraw(int acct, double amount) {
        Account account = entityManager.find(...);
        validateWithdrawal(account, amount);
        account.withdraw(amount);
        entityManager.flush();
    }
}
```

@Interceptors applied to class

- One or more can be applied
- Every method is intercepted
- Executed in order they are declared

XML Binding

```
<ejb-jar>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>BankAccountBean</ejb-name>
      <interceptor-class>
        com.titan.ProfilingInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

One or more <interceptor-class>
entries allowed

Per-method interceptors

- Interceptor executes for one given method
- Executed after any class level interceptors

Per-method interceptors

```
@Stateless
public class BankAccountBean implements BankAccount {

    @PersistenceContext EntityManager entityManager;

    @Interceptors(ProfilingInterceptor.class)
    public void withdraw(int acct, double amount) {
        ...
    }

    public void deposit(int acct, double amount) {
        ...
    }
}
```

Per-method XML

```
<ejb-jar>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>BankAccountBean</ejb-name>
      <interceptor-class>
        com.titan.ProfilingInterceptor
      </interceptor-class>
      <method>
        <method-name>withdraw</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Default interceptors

- You can apply a set of interceptors to every EJB
 - ✓ Per deployment only
 - ✓ Simple ejb-jar.xml description
- '*' wildcard in <ejb-name>

Default Interceptors

```
<ejb-jar>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        com.titan.ProfilingInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Exception Handling

- Allowed to abort invocation
- Allowed to catch and retry an invocation
- Allowed to throw a different exception

Aborting invocation: Validation

```
public class WithdrawValidation {  
  
    @Resource(name="maxWithdraw")  
    double maxWithdraw = 500.0;  
  
    @AroundInvoke  
    public Object validate(InvocationContext ctx)  
        throws Exception  
    {  
        double amount = (Double)ctx.getParameters()[0];  
        if (amount > maxWithdraw) {  
            throw new RuntimeException("Max Withdraw is "  
                + maxWithdraw);  
        }  
        return ctx.proceed();  
    }  
}
```

Aborting invocation: Validation

```
<ejb-jar>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>BankAccountBean</ejb-name>
      <interceptor-class>
        com.titan.WithdrawInvalidation
      </interceptor-class>
      <method>
        <method-name>withdraw</method-name>
      </method>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Custom Security

```
public class RuleBasedAuthorization {
    @EJB SecurityRulesEngine ejb;

    @AroundInvoke
    public Object authorize(InvocationContext ctx)
        throws Exception
    {
        if (!ejb.authorized(ctx.getMethod(),
            ctx.getParameters()) {
            throw new EJBAccessException(
                "Failed to Authorized"
            );
        }
        return ctx.proceed();
    }
}
```

Exception wrapping

- Map SQLException to an exception hierarchy
- Take vendor errno and convert it to:
 - ✓ DeadlockException
 - ✓ InvalidSqlException
 - ✓ Etc...
- Allows user to catch concrete exception
 - ✓ Vendor-specific error handling abstracted

SQLException Mapper

```
public class SQLExceptionMapper {  
  
    @AroundInvoke  
    public Object wrap(InvocationContext ctx)  
        throws Exception  
    {  
        try {  
            return ctx.proceed();  
        } catch (SQLException ex) {  
            switch (ex.getErrorCode()) {  
                case 3344:  
                    throw new DeadlockException(ex);  
                case 4223:  
                    throw new InvalidSqlException(ex);  
                ...  
            }  
        }  
    }  
}
```

SQLException Wrapper

```
@Stateless
@Interceptors(com.titan.SQLExceptionMapper)
public class MyDAOBean implements MyDAO {

    List queryStuff() throws SQLException {
        ...
    }
}
```

Intercepting Lifecycle Events

- Re-use callback annotations
- Same signature as @AroundInvoke methods
- In same Java call stack as any bean callbacks
 - ✓ If the bean has the callback

Custom injection annotation

- Java EE has no annotations to inject directly from JNDI
- Let's create a `@JndiInjected` annotation
 - ✓ Use callback interception to implement

Custom Injection Annotation

```
@Stateless
public class MyBean {

    @JndiInjected("jboss/employees/bill/address")
    Address address;

    ...

}
```

Step 1: Implement annotation

```
package com.titan;  
  
public @interface JndiInjected {  
    String value();  
}
```

Step 2: Write Interceptor

```
public class JndiInjector {  
  
    @PostConstruct  
    public void injector(InvocationContext inv) {  
        InitialContext ctx = new InitialContext();  
        Object target = ctx.getTarget();  
  
        for (Field f : target.getClass().getFields()) {  
            JndiInjected ji =  
                f.getAnnotation(JndiInjected.class);  
            if (ji != null) {  
                Object obj = ctx.lookup(ji.value());  
                f.set(target, obj);  
            }  
        }  
        ... // do same for setter methods  
  
        inv.proceed();  
    }  
}
```

Summary of Use Cases

- Framework components
 - ✓ assemble them transparently
- Pluggable annotations
 - ✓ annotations trigger interceptors
- Extending your EJB Container

Real World Use Cases

- JBoss/Spring integration
 - ✓ Deploy spring packages
 - ✓ Inject deployed spring beans into EJB fields
- JBoss SEAM
 - ✓ Integrates EJBs with the context of your invocation
 - ✓ Biject HTTP Session attributes into your EJB

JBoss/Spring/EJB3 integration

```
@Stateless
public class MyBean implements My {

    @Spring(bean="SomeBean")
    SomeBean bean;

    ...
}
```

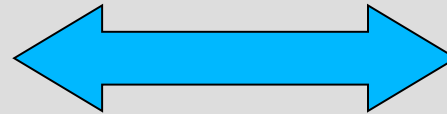
Deployed from
myspring-beans.jar

JBoss SEAM

```
@Stateless  
public class ControllerBean implements Controller {
```

```
    @In @Out Model model;
```

```
    public void action(String action) {  
        if (model.getData() == something) {  
            model.setSomeDate("hello world");  
        }  
    }  
}
```



Pulled from
HTTP Session

Integrating JBoss client interceptors

Client Interceptors and EJB 3.0

- JBoss AOP client interceptors can pass information to EJB3 Interceptor

```
public interface InvocationContext {  
    public Object getTarget();           // reference to the bean instance  
    public Method getMethod();          // method to be called  
    public Object[] getParameters();    // args to the method  
    public void setParameters(Object[]); // access to mutate the args  
    public Map getContextData();        // shared data between interceptors  
    public Object proceed() throws Exception;  
}
```

Client interceptors have API to populate context data map.

Client Interceptors and EJB 3.0

- The `org.jboss.ejb3.interceptor.ClientInterceptorUtil`
 - ✓ Helper methods to populate JBoss `Invocation.getMetadata()`

```
public class ClientInterceptorUtil {  
  
    public static void addMetadata(Invocation invocation, Object key,  
                                  Object value, PayloadKey payload);  
  
    public static void addMetadata(Invocation invocation, Object key, Object value);  
  
    public static Object getMetadata(Invocation invocation, Object key)  
  
}
```

Client + Server Interceptors

```
import org.jboss.ejb3.interceptor.ClientInterceptorUtil;

public class CustomClientInterceptor implements org.jboss.aop.advice.Interceptor,
        java.io.Serializable {

    public Object invoke(org.jboss.aop.joinpoint.Invocation invocation) throws Throwable
    {
        ClientInterceptorUtil.addMetaData(invocation, "Hello", "World");
        return invocation.invokeNext();
    }
}
```

```
public class CustomServerInterceptor {

    @AroundInvoke
    public Object invoke(InvocationContext invocation) throws Throwable
    {
        System.out.println("Context: " + invocation.getContextData().get("Hello"));
        return invocation.proceed();
    }
}
```

Summary

- Interceptors encapsulate cross-cutting concerns
- EJB 3.0 framework of frameworks
 - ✓ Ease of extension
 - ✓ Pluggable annotations
- Already being used in OSS products
- Specification has room to evolve

Entity Beans

POJO based persistence

Goals of Entity Beans

- Same goals as session beans
 - ✓ Fewer interfaces, optional XML DDs, etc.
- No required interfaces or subclassing
- Plain Java based
 - ✓ Allocated with new()
- Provide full Object/Relational mapping
- Supports Inheritance
- Expanded EJBQL
 - ✓ Fully featured
 - ✓ Parallel SQL
 - ✓ Polymorphic Queries

Defining Entity Beans

Full Object/Relational Database Mapping

EJB 3.0 Entity Beans

- O/R Mapping Metadata as annotations
 - ✓ Table mappings, @Table, @SecondaryTable
 - ✓ Column mappings, @Column, @JoinColumn
 - ✓ Relationships, @ManyToOne, @OneToOne, @OneToMany, @ManyToMany
 - ✓ Multi-Table mappings, @SecondaryTable
 - ✓ Embedded objects, @Embedded
 - ✓ Inheritance, @Inheritance, @DiscriminatorColumn
 - ✓ Identifier + Version properties, @Id, @Version

Entity Annotations

@Entity

```
public class Customer {  
    private long id;  
    private String firstName;  
    private String lastName;
```

@Id @GeneratedValue

```
public long getId() {  
    return id;  
}  
public void setId(long id) {  
    this.id = id;  
}
```

```
String getFirstName() { return firstName; }
```

...

Interacting With Entity Bean

Plain Java Objects

Entity Manager

- Entities created as any plain Java object
 - ✓ `Customer cust = new Customer();`
- Plain Java objects and homeless
- Can be detached and reattached to container
 - ✓ Can be serialized to remote client
 - ✓ Remote client can perform updates on local copy
 - ✓ Copy can be sent back to server and merged back in
- Persisted by the EntityManager service
 - ✓ All access through this service
 - ✓ Creation, retrieval, removal, and merging
 - ✓ Analogous to Hibernate Session

Create the objects

- Create the entities like you would any other object
- Allocate entire object graph like any other Java code

```
Customer cust = new Customer("Bill", "Burke");  
  
cust.setAddress(new Address("555 Boston Road"));
```

Entity Manager

- All entities persisted by the EntityManager service
 - ✓ All access through this service
 - ✓ Creation, retrieval, removal, and merging
 - ✓ Analogous to Hibernate Session
- Injected with dependency injection

EntityManager

```
@Stateless public class CustomerDAOImpl implements CustomerDAORemote {
```

```
    @PersistenceContext(unitName="Auction")
```

```
    private EntityManager em;
```

```
    public long create(Customer cust) {  
        em.persist(cust);  
        return cust.getId();  
    }
```

```
    public Customer findById(long id) {  
        return em.find(Customer.class, id);  
    }
```

```
    public void merge(Customer cust) {  
        em.merge(cust);  
    }
```



Inject the EntityManager service

EntityManager

```
@Stateless public class CustomerDAOImpl implements CustomerDAORemote {
```

```
    @PersistenceContext(unitName="Auction")
```

```
    private EntityManager em;
```

```
    public long create(Customer cust) {  
        em.persist(cust);  
        return cust.getId();  
    }
```

```
    public Customer findById(long id) {  
        return em.find(Customer.class, id);  
    }
```

```
    public void merge(Customer cust) {  
        em.merge(cust);  
    }
```

- **Customer allocated remotely**
- **If cascade PERSIST, entire object graph inserted into storage**

EntityManager

```
@Stateless public class CustomerDAOImpl implements CustomerDAORemote {
```

```
    @PersistenceContext(unitName="Auction")
```

```
    private EntityManager em;
```

```
    public long create(Customer cust) {  
        em.persist(cust);  
        return cust.getId();  
    }
```

```
    public Customer findById(long id) {  
        return em.find(cust.class, id);  
    }
```

```
    public void merge(Customer cust) {  
        em.merge(cust);  
    }
```

- **Customer found with primary key**
- **Detached from persistent storage at tx completion**
- **Can be serialized like any other object**

EntityManager

```
@Stateless public class CustomerDAOImpl implements CustomerDAORemote {
```

```
    @PersistenceContext(unitName="Auction")
```

```
    private EntityManager em;
```

```
    public long create(Customer cust) {  
        em.persist(cust);  
        return cust.getId();  
    }
```

```
    public Customer findById(long id) {  
        return em.find(Customer.class, id);  
    }
```

```
    public void merge(Customer cust) {  
        em.merge(cust);  
    }
```

- **Customer can be updated remotely and changes merged back to persistent storage**
- **merge() returns a managed copy of Customer**

Query API

- Queries may be expressed as EJBQL strings
 - ✓ Embedded in code
 - ✓ Externalized to metadata (named queries)
- Invoke via `Query` interface
 - ✓ Named parameter binding
 - ✓ Pagination control

```
@Stateless public class CustomerDAOImpl {  
    ...  
    public List findByDescription(String description, int page) {  
        return em.createQuery("from Customer i where i.description like :d")  
            .setParameter("d", description)  
            .setMaxResults(50)  
            .setFirstResult(page*50)  
            .getResultList();  
    }  
    ...  
}
```

EJB QL 3.0

- EJBQL 3.0 is very similar to HQL (Hibernate Query Language)
- Aggregation, projection
 - ✓ `select max(b.amount) from Bid b where b.item = :id`
 - ✓ `select new Name(c.first, c.last) from Customer c`
- Fetching
 - ✓ `from Item i left join fetch i.bids`
- Subselects
 - ✓ `from Item i join i.bids bid where bid.amount = (select max(b.amount) from i.bids b)`
- Group By, Having, Joins

EJBs + Java Persistence

Added value of EJBs on top of Java Persistence

Persistence Context

- Persistence Context == Persistence Session
 - ✓ Set of entity bean instances that are being managed/watched
 - ✓ Set of queued persistence operations
 - ✓ Analogous to a JDBC Connection
 - ✓ Equivalent to a Hibernate Session

Comparing JDBC

- JDBC Connection used by one user session
 - ✓ Dedicated to one user interaction
- Committed and closed by user when finished

```
Connection con = datasource.getConnection();
```

```
try {  
    Statement statement = ...;  
    con.commit();  
} finally {  
    con.close();  
}
```

Persistence Context

- EntityManagerFactory ~ DataSource
 - ✓ EntityManagerFactory creates persistence contexts
 - ✓ User uses the persistence context and closes when finished
 - ✓ Entity instances managed until EntityManager is closed

```
EntityManager em = entityManagerFactory.createEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();
    Customer cust = em.find(Customer.class, id);
    cust.setName("Billy");
    tx.commit();
} finally {
    em.close();
}
```

EJB Integration with Persistence

- JTA can transparently manage begin/commit/rollback
 - ✓ No need to use EntityTransaction API
- Session and MDBs have tight integration
 - ✓ No need to interact with EntityManagerFactory
 - ✓ Automatically manage persistence context
 - ✓ EJB controls lifecycle of persistence context
- Persistence context types
 - ✓ Transaction scoped
 - ✓ Extended

Transaction scoped

```
@Stateless public class CustomerDAOImpl implements CustomerDAORemote {  
  
    @PersistenceContext(unitName="Auction")  
    private EntityManager em;  
  
    public long create(Customer cust) {  
        em.persist(cust);  
        return cust.getId();  
    }  
  
    public Customer findById(long id) {  
        return em.find(cust.class, id);  
    }  
  
    public void merge(Customer cust) {  
        em.merge(cust);  
    }  
}
```

- **EntityManager instance created for duration of transaction**
- **When TX commits, persistence context is destroyed**

Extended Persistence Context

- Injectable only into stateful beans
- Persistence context is created and married to SFSB
- Entity instances remain managed after TX completes

Extended Persistence Context

```
@Stateful public class ShoppingCartBean implements ShoppingCart {
```

```
    @PersistenceContext(type=EXTENDED) EntityManager em;
```

```
    Customer cust;
```

```
    public void setCustomerId(int id) {  
        cust = em.find(Customer.class, id);  
    }
```

```
    public void updateName(String name) {  
        cust.setName(name);  
    }
```

- **Customer entity automatically synchronized at end of method (end of TX)**

Persistence Context Propagation

- Persistence contexts are “sticky” with TX
- If you interact with a PC within a TX
 - ✓ Same PC is propagated to any nested EJB calls

Extended Persistence Context

```
@Stateful public class ShoppingCartBean implements ShoppingCart {  
  
    @PersistenceContext(type=EXTENDED) EntityManager em;  
  
    @EJB CustomerDAOLocal dao;  
  
    Customer cust;  
  
    public void setCustomerId(int id) {  
        cust = em.find(Customer.class, id);  
    }  
  
    public void updateName(String name) {  
        dao.updateName(cust, name);  
    }  
}
```

Extended Persistence Context

```
@Stateless public class CustomerDAOBean implements ... {
```

```
    @PersistenceContext EntityManager em;
```

```
    public void updateName(Customer cust, String name) {
```

```
        Customer cust2 = em.find(Customer.class, cust.getId());
```

```
        assert(cust2 == cust);
```

```
        dao.updateName(cust);
```

```
    }
```

Same customer instances because it is the same persistence context

JBoss Extensions to EJB 3

From EJBs to Enterprise POJOs

Embeddable JBoss

Yes, you can use EJB 3.0 outside the container!

Embeddable JBoss

- Use EJB 3.0 outside of the application server
 - ✓ JUnit tests
 - ✓ Standalone Java SE apps
 - ✓ Tomcat standalone
- Fast boot time
- Real JBoss stack
 - ✓ JCA
 - ✓ JTA
 - ✓ JMS
 - ✓ EJB 3.0
 - ✓ Connection Pooling

Java Persistence & Security

Java Persistence & Security

- EJB 3.0 does not define Entity bean security
- JBoss 3.0 uses Java EE JACC service
- Secure CRUD operations of an entity bean
 - ✓ Usable only with EJBs (session or mdb)

Java Persistence & Security

- Configured in persistence.xml
- Specify role, entity class in property name
- Specify permissions allowed for that role

```
<persistence>
<persistence>
  <persistence-unit name="tempdb">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.jacc.MANAGER.org.jboss.tutorial.Customer"
        value="insert,update,delete,read"/>
      <property name="hibernate.jacc.enabled" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Java Persistence & Security

- Role allowed

```
<persistence>
<persistence>
  <persistence-unit name="tempdb">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.jacc.MANAGER.org.jboss.tutorial.Customer"
        value="insert,update,delete,read"/>
      <property name="hibernate.jacc.enabled" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Java Persistence & Security

- Entity class you want to secure

```
<persistence>
<persistence>
  <persistence-unit name="tempdb">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.jacc.MANAGER.org.jboss.tutorial.Customer"
        value="insert,update,delete,read"/>
      <property name="hibernate.jacc.enabled" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Java Persistence & Security

- Permissions allowed
 - ✓ insert, update, delete, or read

```
<persistence>
<persistence>
  <persistence-unit name="tempdb">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.jacc.MANAGER.org.jboss.tutorial.Customer"
        value="insert,update,delete,read"/>
      <property name="hibernate.jacc.enabled" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Java Persistence & Security

```
@Stateless
@SecurityDomain("CorporateSecurity")
@AspectDomain("JACC Stateless Bean")
public class CustomerDAOImpl implements CustomerDAORemote {

    @PersistenceContext(unitName="Auction")
    private EntityManager em;

    public long create(Customer cust) {
        em.persist(cust);
        return cust.getId();
    }

    public Customer findById(long id) {
        return em.find(Customer.class, id);
    }
}
```

The Service Bean

Service bean

- Singleton services
- Multi-threaded and stateful
- Lifecycle and dependency management
- Next iteration of the JBoss MBean

Service bean

- Full EJB 3 library of annotations available
 - ✓ @TransactionAttribute, @MethodPermissions, @Inject
- @Remote and @Local interface publishing
- New @Management interface
 - ✓ JMX is now an aspect

Service interfaces

- @Remote, @Local
- Just like EJB

@Remote

```
public interface CalculatorRemote
{
    BigDecimal add(float x, float y);
    BigDecimal divide(float x, float y);
}
```

Service interfaces

- @Management
- JMX interface, JMX aspect

@Management

```
public interface CalculatorManagement
{
    int getPrecision();
    void setPrecision();
}
```

Service bean class

- @Service deploys a singleton
- Multi-threaded and stateful

@Service

```
public class CalculatorService implements CalculatorRemote, CalculatorManagement
{
    private int precision;

    public int getPrecision() { return precision; }
    void setPrecision(int p) { precision = p; }

    BigDecimal divide(float x, float y) {...}
}
```

Message Driven POJOs

Typed MDBs

Message Driven POJOs

- MDBs with a typed business interface
- MDBs that publish a well known interface
- Simplify JMS
- Invoke methods, don't build messages
- Cleaner, clearer code

Message Driven POJOs

- Define a @Consumer bean class
- @Consumer implements 1-N @Producer interfaces
- @Producer a means to send JMS messages
- @Consumer a means to receive JMS messages

@Producer

@Producer

```
public interface EmailRemote {  
    public void send(String msg);  
    public void send(String to, String from, String msg, Map props);  
}
```

@Consumer

```
@Consumer(activation={
    ActivationConfigProperty(name="destinationType", value="javax.jms.queue"),
    ActivationConfigProperty(name="destination", value="queue/foo")
})
public class EmailerBean implements EmailRemote {

    public void send(String msg) {
        ...
    }

    public void send(String to, String from, String msg, Map props) {
        ...
    }
}
```

Using a Producer

- Producers implicitly extend ProducerObject interface

```
{  
    EmailRemote emailer = (EmailRemote)ctx.lookup(EmailRemote.class.getName());  
  
    ProducerConfig.connect(emailer);  
    emailer.send("spam");  
    emailer.send("bill@jboss.org", "tss@tss.com", "spam", new HashMap());  
    ProducerConfig.close(emailer);  
}
```

Configuring Message Properties

```
@Producer(transacted=true)
```

```
@MessageProperties(delivery=NON_PERSISTENT)
```

```
public interface EmailRemote {
```

```
    @MessageProperties(delivery=PERSISTENT, timeToLive=1000, priority=1)
```

```
    public void send(String msg);
```

```
    public void send(String to, String from, String msg, Map map);
```

```
}
```

Asynchronous EJBs

Lightweight asynchronous communication

Asynchronous EJBs

- Goals
 - ✓ Provide a lightweight asynchronous API for all EJB types
 - ✓ VM local and Remote capabilities
 - ✓ Zero changes to existing interfaces/code
 - ✓ Support obtaining method results asynchronously
 - Not just oneway.

Asynchronous EJBs

- Available for Stateless, Stateful, and Service
- Each @Remote and @Local interface implements EJBProxy

```
public class Asynch {  
    public static <T> T getAsynchronousProxy(Object obj) {...}  
}  
...  
CalculatorRemote calc = (CalculatorRemote)ctx.lookup("calculator");  
  
// turn a synchronous EJB proxy into an asynchronous proxy  
CalculatorRemote calcAsynch = Asynch.getAsynchronousProxy(calc);
```

Asynchronous EJBs

- Asynchronous Proxy same exact interface
- Invoking methods on proxy happen in background

```
CalculatorRemote calc = (CalculatorRemote)ctx.lookup("calculator");  
CalculatorRemote calcAsynch = Async.getAsynchronousProxy(calc);
```

```
calcAsynch.add(1, 1);    // invoked in background, in another thread
```

```
calcAsynch.divide(10, 2); // invoked in background
```

Asynchronous EJBs

- Obtain response asynchronously
 - ✓ Future interfaces

```
calcAsynch.add(1, 1);           // invoked in background, in another thread
Future result1 = Asynch.getFutureResult(calcAsynch);
calcAsynch.divide(10, 2);     // invoked in background
Future result2 = Asynch.getFutureResult(calcAsynch);

int val1 = result1.get();     // block until first method call returns
if (result2.isDone()) {     // poll to see if done
    int val2 = result2.get();
}
```

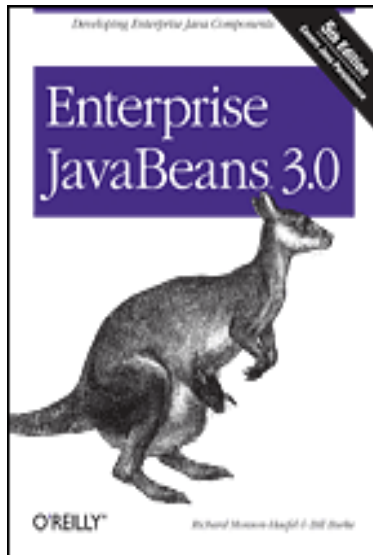
Summary

- Simpler programming model
 - ✓ Less artifacts, POJOs
 - ✓ Annotation alternative to XML
- Rich persistence model
 - ✓ EJB QL fully functional!
 - ✓ Full O/R mapping

- 1. Code
- 2. Compile
- 3. Jar
- 4. Run

JBoss Inc.

- JBoss EJB 3.0 Preview Available
 - ✓ Download at www.jboss.org
 - ✓ Tutorial with code examples



- ✓ O'Reilly EJB 3.0 5th Edition
 - Includes JBoss workbook