SUN TECH DAYS 2008–2009
A Worldwide Developer Conference

REST

Carol McDonald, Java Architect

REpresentational State Transfer

Get
http://www.parts-depot.com/parts

REST Web Service

Response XML data =
REpresentational State

Transfer

Client State1

Client State2

- The URL is the resource
- GET the page from the server
- html page is transferred to the browser
  - REpresentational State transfer
- You click on a hyperlink in page (hypermedia), the process is repeated
- The page (application state) is stored on the client (browser)

REST Tenets

- Resources (nouns)
  - Identified by a URI, For example:
    - http://www.parts-depot.com/parts
- Methods (verbs) to manipulate the nouns
  - Small fixed set:
    - Create, Read, Update, Delete
- **Representation** is how you view the **State**
  - data and state transferred between client and server
  - XML, JSON...
- Use verbs to exchange application state and representation

HTTP Example

Request

```
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Method

Resource

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<recordings xmlns="…">
  <recording>…</recording>
  …
</recordings>
```

State transfer

Representation

REST in Five Steps*

- Give everything an ID
- Use standard methods
- Link things together
- Multiple representations
- Stateless communications

Inspired by Stefan Tilkov:
http://www.innoq.com/blog/st/presentations/2008/2008-03-13-REST-Intro--QCon-London.pdf

Give Everything an Id

Id is a URI, Every resource has a URI

http://company.com/customers/123456

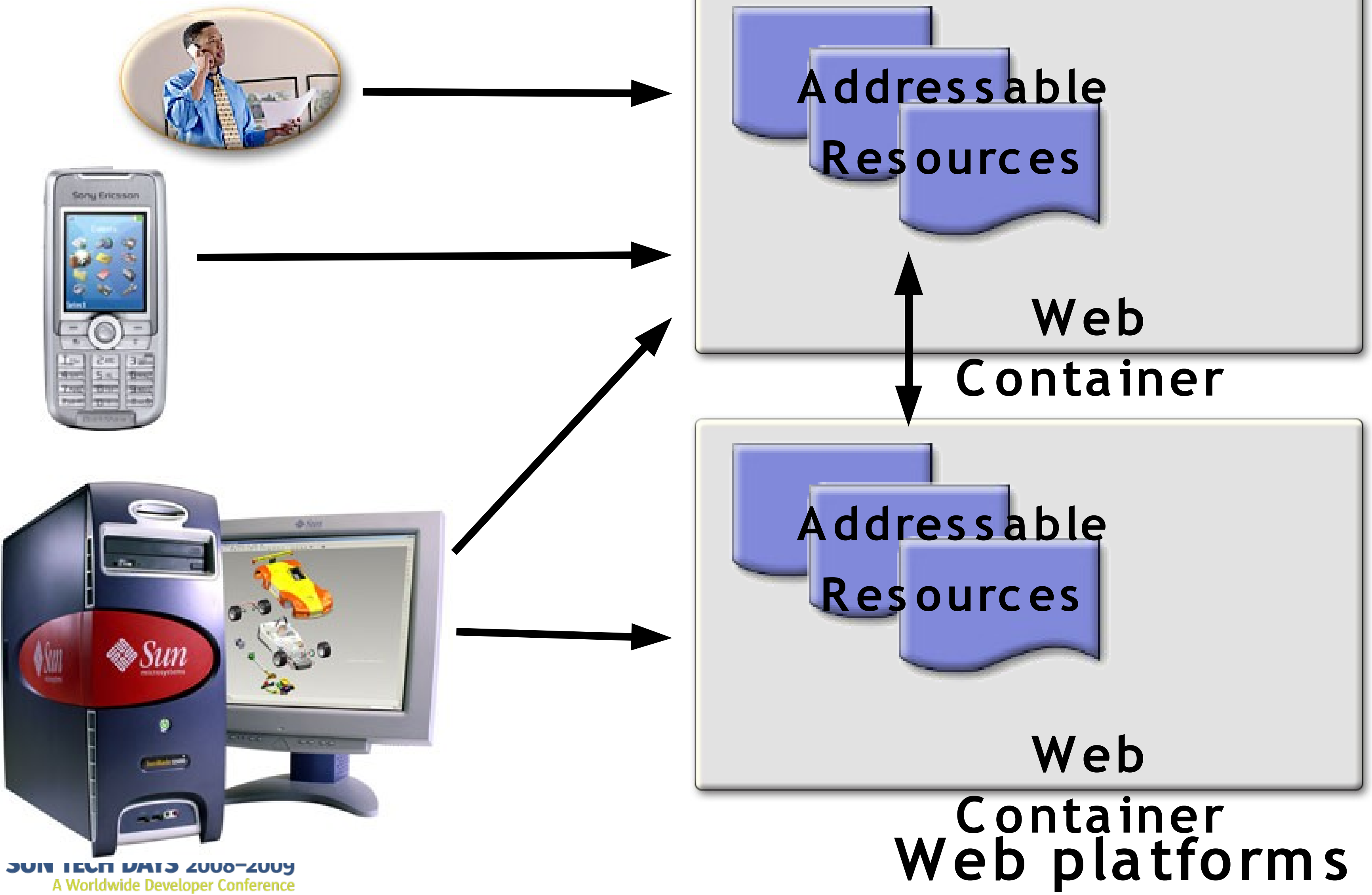
Resource Collection name

Primary key

- URIs identify all of the “high-level” resources that your application provides:
  - > items, collections of items, virtual and physical objects, or computation results.

http://company.com/customers/123456/orders/12

The RESTful Web



Use Standard Methods

CRUD

OO

```
class Resource {
    Resource(URI u);
    Response get();
    Response post(Request r);
    Response put(Request r);
    Response delete();
}
```

Method	Purpose
GET	Read
POST	create/append without ID
PUT	Update or create with ID
DELETE	Remove

Use Standard HTTP Methods: GET = Read

- GET to retrieve information
  - Retrieves a given URI
  - Should not modify anything
  - Idempotent:
    - can be repeated and the side-effects remain the same
  - Cacheable
  - Example
    - `GET /store/customers/123456`

"Give me the XML data for this customer"

Use Standard HTTP Methods: POST = Create

- POST to add new information
- add the entity , **append** to the container resource
  - > Example
    - `POST /store/customers`

Add the customer specified in the POSTDATA to the list of customers. return URI

Use Standard HTTP Methods: PUT = Update

- PUT to update information
- Full entity **create/replace** used when you know the “**id**”
  - Example
    - ```
      PUT /store/customers/123456
      Content-Type: application/xml
      ```

Replaces the representation of the customer with an updated version.
      ```

Use Standard HTTP Methods: DELETE

- Remove (logical) an entity
  - Example
    - `DELETE /store/customers/123456`

Deletes the customer '123456" from the system

Use Standard Methods:

Order Customer Mgmt Example

«interface»
Resource
- GET
- PUT
- POST
- DELETE

/orders
- GET - list all orders
- PUT - unused
- POST - add a new order
- DELETE - unused

/orders/{id}
- GET - get order details
- PUT - update order
- POST - add item
- DELETE - cancel order

/customers
- GET - list all customers
- PUT - unused
- POST - add new customer
- DELETE - unused

/customers/{id}
- GET - get customer details
- PUT - update customer
- POST - unused
- DELETE - delete customer

/customers/{id}/orders
- GET - get all orders for customer
- PUT - unused
- POST - add order
- DELETE - cancel all customer orders

http://www.infoq.com/articles/rest-introduction

Use Standard Methods:

Order Customer Mgmt Example

- /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

http://www.infoq.com/articles/rest-introduction

Use Standard HTTP Methods

- HTTP Get, Head
  - > Should not modify anything
- Idempotency:
  - > PUT, DELETE, GET, HEAD can be repeated and the results are the same
- Cache-able
  - > With Correct use of Last-Modified and ETag

REST in Five Steps*

- Give everything an ID
- Use standard methods
- Link things together
- Multiple representations
- Stateless communications

* Inspired by Stefan Tilkov:
http://www.innoq.com/blog/st/presentations/2008/2008-03-13-REST-Intro--QCon-London.pdf

Link Things Together

Hypermedia as the engine of application state

```
<prop self="http://example.com/orders/101230">
  <customer ref="http://example.com/customers/bar">
  <product ref="http://example.com/products/21034"/>
  <amount value="1"/>
</order>
```

- Service provides a set of links to the Client
  - Enables client to move the application from one state to the next by following a link
  - useful way to make an application dynamic

Link Things Together

- Representations contain links to other resources
- Put a service in different states by following links and filling in forms
  - “Hypermedia as the engine of application state”
  - HTML, XML, JSON, N3 can all be Hypermedia
- Client is in charge
  - Service offers potential states
  - Service does not impose state transition order

Multiple Representations

- Offer data in a variety of formats, for different needs
  - > XML
  - > JSON
  - > (X)HTML
- Maximize reach, Web UI can be Web API
- Support **content negotiation**
  - > Accept header

```
GET /foo
Accept: application/json
```

  - > URI-based

```
GET /foo.json
```

Stateless Communications

- HTTP protocol is stateless
- Everything required to process a request contained in the request
  - Service should not store session from previous requests
  - Eliminates many failure conditions
- Client responsible for application state
- Service responsible for resource state
  - States of Web service are resources
  - REST is the transfer of states
- Eg. Booking travel
  - Create itinerary resource, fill itinerary, post itinerary
  - All held on client as **not** on **server session**

Common Patterns: Container, Item

Server in control of URI path space

- List container contents: `GET /container`
- Add item to container: `POST /container`
  - with item in request
  - URI of item returned in HTTP response header
  - e.g. `Location: http://host/container/item`
- Read item: `GET /container/item`
- Update item: `PUT /container/item`
  - with updated item in request
- Remove item: `DELETE /container/item`
- Good example: Atom Publishing Protocol

Common Patterns: Map, Key, Value

Client in control of URI path space

- List key-value pairs: **`GET /map`**
- Put new value to map: **`PUT /map/{key}`**
  - with entry in request
  - e.g. **`PUT /map/dir/contents.xml`**
- Read value: **`GET /map/{key}`**
- Update value: **`PUT /map/{key}`**
  - with updated value in request
- Remove value: **`DELETE /map/{key}`**
- Good example: Amazon S3

Key Benefits

- Server side
  - Horizontal scaling
  - Straightforward failover
  - Cacheable
  - Reduced coupling
  - Works well with existing Web infrastructure
  - Uniform Interface
- Client side
  - Bookmarkable
  - Easy to experiment in browser
  - Broad programming language support
  - Choice of data formats

Agenda

- **REST Primer**
- RESTful Design and API Elements
  - > Give everything an ID
  - > Link things together
  - > Use standard methods
  - > Multiple representations
- Building a Simple Service
- Deployment Options
- Status
- Q & A

Give Everything an ID

- "Thing" == resource class
  - POJO, No required interfaces
- ID provided by **@Path** annotation
  - Relative to deployment context
  - Annotate class or "sub-resource locator" method

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

  @Path("customer")
  CustomerResource getCustomer(...) {...}
}
```

http://host/ctx/orders/12

http://host/ctx/orders/12/customer

EMPOWER: YOU

Mapping URIs to Classes

container for catalog it

```
@Path("/items")
public class Items {
    @Get
    public ItemsConverter get() {
        ... return new ItemsConverter(itemList);
    }
    @Path("{id}/")
    public ItemResource
getItem(@PathParam("id")int id) {
        ... return itemResource;
    }
}
```

one 'item'

Methods on root or sub resources

- Sub-resources allow navigation among related resources
- Two ways to create a sub-resource
  - > Add a HTTP method in the parent resource with a @Path annotation

```
public class ItemResource {

@Path("/items/{id}/")
@GET

public ItemConveter get(@PathParam("id") Long id) {
  .....

}
```

  - > Add a sub-resource locator method in the parent resource that returns an instance of a sub-resource class

```
@Path("/items/")
public class ItemsResource {
  @Path("{id}/")

   public ItemResource getItemResource(@PathParam("id") Long id) { ...
     return resource;

   }
```

Use Standard Methods

- Annotate resource class methods with standard method
  - @GET, @PUT, @POST, @DELETE, @HEAD
  - @HttpMethod meta-annotation allows extensions
- annotations on parameters specify mapping from request data
- Return value mapped to response

```
@Path("orders/{order_id}")
public class OrderResource {
  @GET
  Order getOrder(@PathParam("order_id") String id) {
    ...
  }
}
```

methods on root or sub resources

```
@Path(“/items”)
class Items {
   @GET <type> get() { ... }
   @POST <type> create(<type>) { ... }
}
```

```
class Item {
   @GET <type> get(...) { ... }
   @PUT void update(...) { ... }
   @DELETE void delete(...) { ... }
}
```

Java method name is not significant
The HTTP method is the method

Multiple Representations

Static and dynamic content negotiation

- Annotate methods or classes
  - **@ProduceMime**, **@ConsumeMime**

```
@GET
@ProduceMime({"application/xml","application/json"})
Order getOrder(@PathParam("order_id") String id) {
   ...
}
@GET
@ProduceMime("text/plain")
String getOrder(@PathParam("order_id") String id) {
   ...
}
```

Multiple Representations

HTTP Example

Request

```
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Accept HTTP header

Format

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<recordings xmlns="…">
  <recording>…</recording>
  …
</recordings>
```

Content-type HTTP header

Representation

Multiple Representations: JAX-RS consuming

- Annotated method parameters extract client request information
  - `@PathParam` extracts information from the request URI
    - **http://host/catalog/items/123**
  - `@QueryParam` extracts information from the request URI query parameters
    - **http://host/catalog/items/?start=0**

Multiple Representations: JAX-RS consuming

```
@Path("/items/")
@ConsumeMime("application/xml")
public class ItemsResource {

   @GET
   ItemsConverter get(@QueryParam("start")
     int start) {
     ...
   }

   @Path("{id}/")
   ItemResource getItemResource(@PathParam("id") Long id){
   ...
   }

}
```

http://host/catalog/items/?start=0

http://host/catalog/items/123

Multiple Representations:Supported Types

- JAX-RS can automatically (un)-marshall between HTTP message bodies and Java types ← **response**
  - Method return value marshalled into HTTP response body
  - Un-annotated method parameter unmarshalled from HTTP message content **request** →
- “Out-of-the-box” support for the following
  - **text/xml, application/xml, application/json** – JAXB class
  - ***/*** – byte[], InputStream, File, DataSource
  - **text/*** – String
  - **application/x-www-form-urlencoded** -MultivaluedMap<String, String>

EMPOWER: YOU

Multiple Representations: JAX-RS producing

- A HTTP method classifies the response type with `@ProduceMime`
- The method return type is the response
  - Java type that is the representation
  - Or `void` if no representation
- A response builder can produce arbitrary responses
  - e.g. created or redirected responses

Multiple Representations: producing a response

```
@Path("/items")
class Items {
   @GET @ProduceMime("application/xml")
   ItemsConverter get() { ... }

   @POST
   @ProduceMime("application/xml")
   Response create(Ent e) {
      // persist the new entry, create  URI
     return Response.created(
            uriInfo.getAbsolutePath().
            resolve(uri+"/")).build();
   }
}
```

JAXB class

Use Response class
to build "created"response

EMPOWER: YOU

Uniform interface: HTTP request and response

```
C: POST /items HTTP/1.1
C: Host: host.com
C: Content-Type: application/xml
C: Content-Length: 35
C:
C: <item><name>dog</name></item>

S: HTTP/1.1 201 Created
S: Location: http://host.com/employees/1234
S: Content-Length: 0
```

Multiple Representations

```
@Post
@ConsumeMime("application/x-www-form-urlencoded")
@ProduceMime("application/rss+xml")

public JAXBElement updateEmployee(
        MultivalueMap<String, String> form) {

  ...
```

Serialized to a XML stream

Converted to a map for accessing form's field

Returning Status Code

- Mainly for error conditions
- For example
  - 409 Conflict
  - 404 Not Found
- See http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
- Status can be returned either with WebApplicationException or Response

Return Code Examples

```
@GET
public Employee getEmployee(
        @PathParam("id") int id) {
    if (!doesEmployeeExist(id))
        throw new NotFoundException( id + " does not
exist");

@PUT
public Response putUser(
        @PathParam("id") int id) {
    if (!id.equals(userid) )
        return Response.status(409).entity("userids
differ").build();
```

- Use it when the method has a return type
- Simple error indicator

extends WebApplicationException

- Response is used to construct more sophisticated return result
- Can build responses for redirects, errors, ok, setting headers, etc

Link Things Together

- `UriInfo` provides information about deployment context, the request URI and the route to the resource
- `UriBuilder` provides facilities to easily construct URIs for resources

```
@Context UriInfo i;
OrderResource r = ...
UriBuilder b = i.getBaseUriBuilder();
URI u = b.path(OrderResource.class).build(r.id);

List<URI> ancestors = i.getAncestorResourceURIs();
URI parent = ancestors.get(ancestors.size()-1);
```

Statelessness: JAX-RS

- The default component lifecycle is per-request
  - A new instance created for every request
  - Constructor/fields used like plain old Java objects
  - Reduces concurrency issues
- HTTP session life-cycle is not supported
- Developer must model state
  - As resources; and
  - As application state in the representations

EMPOWER: YOU

Agenda

- REST Primer
- RESTful Design and API Elements
- **Building a Simple Service**
- Deployment Options
- Status

Example RESTful Catalog

http://weblogs.java.net/blog/caroljmcdonald/archive/2008/08/a_restful_pet_c.html

Carol McDonald's Blog: a RESTful Pet Catalog - Mo...

File Edit View History Bookmarks Tools Help

http://we restful ca

My pages Projects Communities java.net

Get Involved
- java-net Project
- Request a Project
- Project Help Wanted Ads
- Publicize your Project
- Submit Content

Get Informed
- About java.net
- Articles
- Blogs
- Events
- Also in Java Today
- java.net Online Books
- java.net Archives

Get Connected
- java.net Forums
- Wiki and Javapedia
- People, Partners, and Jobs
- Java User Groups
- RSS Feeds

Carol McDonald's Blog

«a Dyna

a RESTful Pet Catalog

Posted by caroljmcdonald on August 05, 2008 at 09:03 AM | Comn

a RESTful Pet Catalog

This Sample Pet Store Catalog application shows how to ex
applications, and it shows how to code a Dojo client which
Dojo grid). I re-implemented this Sample Catalog applicatio
side which is also available in the Java One Metro hands on

Download the RESTful Pet Catalog Code

Dojo is an open source DHTML toolkit written in JavaScript.

JAX-RS provides a standardized API for building RESTful we
resources identified by universal resource identifiers (URIs).
Java Objects (POJOs) to expose web resources identified b

Example RESTful Catalog

Example dojo and RESTful Web Service - Mozilla Firefox

File Edit View History Bookmarks Tools Help

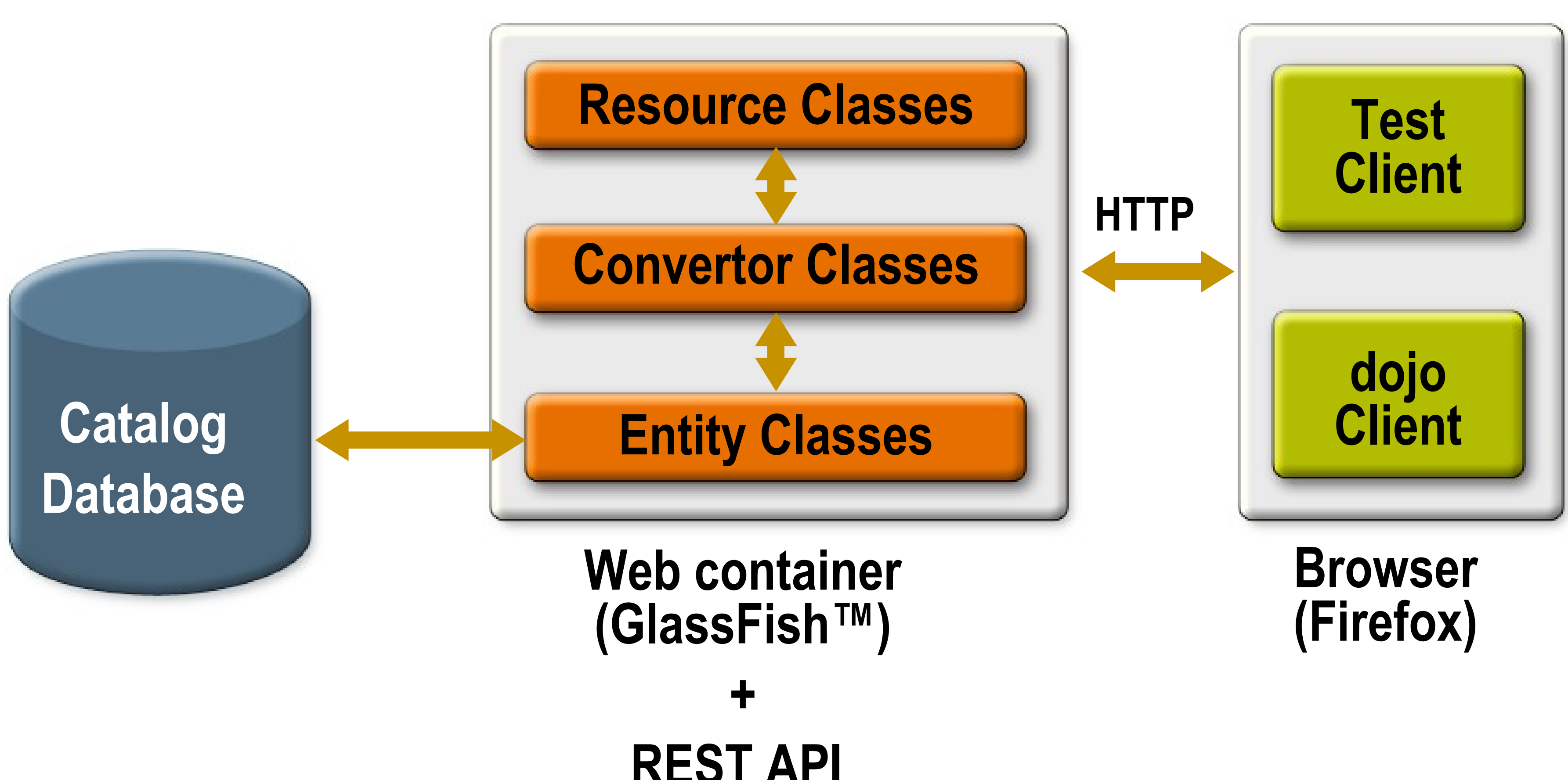
http://localhost:8 restful pet

Pet Catalog

<< >>

Name	Description	Photo	Price
Friendly Cat	This black and white colored cat is super friendly. Anyone passing by your front yard will find him puring at their feet and trying to make a new friend. His name is Anthony, but I call him Ant as a nickname since he loves to eat ants and other insects.		307.10
Fluffy Cat	A great pet for a hair stylist! Have fun combing Bailey's silver mane. Maybe trim his whiskers? He is very patient and loves to be pampered.		307.00
Sneaky Cat	My cat is so sneaky. He is so curious that he just has to poke his nose into everything going on in the house. Everytime I turn around, BAM, he is in the room peaking at what I am doing. Nothing escapes his keen eye. He should be a spy in the CIA!		307.20
	A great pet to lounge on the sofa		

Example RESTful Catalog Service



RESTful Catalog

- Dojo client, JAX-RS, JAXB, JPA

List Item - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Listing Items
Back to index
Item 31..40 of 102 Previous 10 Next 10

Name	photo	Price
Beach Dog		250.00
Scrapper Dog		257.00
Intense Worker Dog		500.00

Done

JAX-RS class
ItemsResource
Entity Class
Item
ItemsConverter
JAXB class
DB
Dojo client

Entity Classes

- Use JPA to map/retrieve data from the database as entities
  - > Can use NetBeans to generate JPA entity classes from a database
- To expose domain objects as RESTful resources
  - > Think of each entity as a resource that can be identified by one or more URLs

http://host/catalog/items/

Item			
ID	NAME	DESC	URL

```
@Entity
public class Item {
    @Id

     int id;
     String name;
     String descr;
     String url;

}
```

Converter Classes

- To convert the domain objects (JPA entities) into XML and/or JSON.
- Use JAXB :
  - **NetBeans** IDE provides tools for automating this process
    - creates **container** converter, **item** converter and URI converter for each entity class
  - supports marshaling and unmarshaling between **JSON** strings and **JAXB** instances

ItemsConverter JAXB annotated

```
@XmlRootElement(name = "items")
public class ItemsConverter {
    private Collection<ItemConverter> items;
    private URI uri;

    @XmlAttribute
    public URI getUri() {
        return uri;
    }
    @XmlElement
    public Collection<ItemConverter> getItem() {
        ...
        return items;
    }
}
```

ItemConverter JAXB annotated

```
@XmlRootElement(name = "item")
public class ItemConverter {
    private Item entity;
    private URI uri;

    @XmlAttribute
    public URI getUri() {
        return uri;
    }
    @XmlElement
    public Long getId() {
        return (expandLevel > 0) ? entity.getId() : null;
    }
    . ..
}
```

RESTful Resource Layer

- RESTful resources from the JPA entity classes
  - NetBeans IDE provides tools for automating this process
- Use the container-item pattern
  - Container resource retrieves a collection of entities
  - Item resource retrieves an individual entity
- Mapping resources to URIs
  - `/items` – URI for a list of catalog items
  - `/item/1` – URI for item 1
- Support for caching
  - JSR-311 provides API for evaluating preconditions based on lastModified timestamps and/or entity tags

ItemsResource

```
@Path("/items/")
public class ItemsResource {
    @Context
    protected UriInfo uriInfo;
    @GET
    @ProduceMime({"application/xml", "application/json"})
    public ItemsConverter get(@QueryParam("start")
            @DefaultValue("0") int start){
        return new ItemsConverter(
         getEntities(start), uriInfo.getAbsolutePath());

    }
```

JAXB class

Performs JPA Query, returns list of entities

ItemsResource

```
@Path("/items/")
public class ItemsResource {
    @POST
    @ConsumeMime({"application/xml", "application/json"})
    public Response post(ItemConverter data) {
            createEntity(data );
            return Response.created(
               uriInfo.getAbsolutePath().
               resolve(id) + "/")).build();

    }
```

Performs JPA persist, inserts entity in DB

Dojo Client

- Use dojo.xhrGet , dojo.xhrPut, dojo.xhrPost, dojo.xhrDelete to make HTTP method calls
- Example JSON data:


```
{"items":
  {"@uri":"http://host/catalog/resources/items/",
   "item":[
     {"@uri":"http://host/catalog/resources/items/1/",
       "name":"Friendly Cat",
      "description":"This black and white colored cat is super friendly.",
       "id":"1",
       "imageurl":"http://localhost:8080/CatalogService/images/anthony.jpg"},
     {"@uri":"http://host/catalog/resources/items/2/",
       "name":"Fluffy Cat",
       "description":"A great pet for a hair stylist!
       "id":"2",
       "imageurl":"http://localhost:8080/CatalogService/images/bailey.jpg"}
      ]
  }
}
```

Dojo client.js

```
// make request to the items web service
function loadTable(page){
    start = page * batchSize;
    var targetURL = "resources/items/?start="+
               encodeURIComponent(start);
    dojo.xhrGet({
        url: targetURL,
        handleAs: "json",
        load: handleResponse,
        error: handleError
    });
}
// Process the response from the items web service
function handleResponse(responseObject, ioArgs){
    // set the model object with the returned items list
    model.setData(responseObject.items.item);

}
function next() {
    page =page + 1;
    loadTable(page);
}
```

Performs HTTP GET on url catalog/items

Callback puts response data in dojo model for grid table

Dojo client index.html

```
<button dojoType="dijit.form.Button" id="next">
  >>
  <script type="dojo/method" event="onClick">
     next();
  </script>
</button>


<div id="grid" dojoType="dojox.Grid" model="model"
         structure="layout" autoWidth="true" >
```

http://weblogs.java.net/blog/caroljmcdonald/archive/2008/08/a_restful_pet_c.html

EMPOWER: YOU

Client & WADL

- NetBeans IDE can generate JavaScript client stubs from a WADL or projects containing RESTful resources
- NetBeans IDE exposes popular SaaS services using WADLs and can generate Java code to access them
-
  - Learn more at https://wadl.dev.java.net/
  - Example:

```
<application>
  <resources base="http://localhost:11109/CustomerDB/resources/">
    <resource path="/items/">
      <method name="GET">
        <request>
          <param default="0" type="xs:int" style="query"  name="start"/>
          <param default="10" type="xs:int" style="query" name="max"/>
        </request>
        <response>
          <representation mediaType="application/xml"/>
        </response>
      </method>
    </resource>
  </resources>
  .....
</application>
```

DEMO

Building the RESTful Catalog

EMPOWER: YOU

Agenda

- REST Primer
- RESTful Design and API Elements
- Building a Simple Service
- Deployment Options
  - > Java SE
  - > Java Servlet API
  - > Java EE
  - > Other
- Status
- Q & A

Java SE

- **`RuntimeDelegate`** used to create instances of a desired endpoint class
- Application supplies configuration information
  > List of resource classes and providers as subclass of **`ApplicationConfig`**
- Implementations can support any Java type
  > Jersey supports Grizzly (see below), LW HTTP server and JAX-WS Provider

```
ApplicationConfig config = ...
RuntimeDelegate rd = RuntimeDelegate.getInstance();
Adapter a = rd.createEndpoint(config, Adapter.class);
SelectorThread st = GrizzlyServerFactory.create(
    "http://127.0.0.1:8084/", a);
```

Servlet

- JAX-RS application packaged in **`WAR`** like a servlet
- For JAX-RS aware containers
  - **`web.xml`** can point to **`ApplicationConfig`** subclass
- For non-JAX-RS aware containers
  - **`web.xml`** points to implementation-specific **`Servlet`**; and
  - an **`init-param`** identifies the **`ApplicationConfig`** subclass
- Resource classes and providers can access **`Servlet`** request, context, config and response via injection

Java EE 6 Plans

- Applications deployed in an Java EE 6 Web container will have access to additional resources and capabilities:
  - Resources (**`@Resource`**, **`@Resources`**)
  - Web Services (**`@WebServiceRef`**, **`@WebServiceRefs`**)
  - EJB (**`@EJB`**, **`@EJBs`**)
  - Persistence (**`@PersistenceContext`**, **`@PersistenceUnit`**, ...)
  - Lifecycle management (**`@PostConstruct`**, **`@PreDestroy`**)
  - Security (**`@RolesAllowed`**, **`@RunAs`**, **`@PermitAll`**, ...)
- Hoping to be able to make use of Web Beans (JSR 299) for much of this

Other

- Easy to implement for other containers
- Implementation work underway for:
  - Restlet
  - JBoss RESTEasy
  - Apache CXF
  - Jersey

Agenda

- REST Primer
- RESTful Design and API Elements
- Building a Simple Service
- Deployment Options
- Status

Status

- Early Draft Review completed November 2007
- July 2008: Final Draft
- September 2008: Final Release

Summary

- REST architecture is gaining popularity
  - Simple, scalable and the infrastructure is already in place
- JAX-RS (JSR-311) provides a high level declarative programming model
  - http://jersey.dev.java.net
- NetBeans

EMPOWER: YOU

For More Information

- Official JSR Page
  - http://jcp.org/en/jsr/detail?id=311
- JSR Project
  - http://jsr311.dev.java.net/
- Reference Implementation
  - http://jersey.dev.java.net/
- Marc's Blog
  - http://weblogs.java.net/blog/mhadley/
- Paul's Blog
  - http://blogs.sun.com/sandoz/
- Jakub's Blog
  - http://blogs.sun.com/japod/
- Carol's Blog
  - http://weblogs.java.net/blog/caroljmcdonald/

Carol McDonald
Java Architect
http://weblogs.java.net/blog/caroljmcdonald/