# JAX-RS Overview

Sang Shin
Michèle Garoche
www.javapassion.com
“Learn with Passion!”

# Topics

- JAX-RS quick overview
- Creating resources
  - *@Path, @PathParam, @QueryParam*
  - Sub-resource locators
- HTTP method annotations (Uniform interface)
  - *@GET, @POST, @PUT, @DELETE*
- Representations
  - *@Produces, @Consumes*

# Topics (Continued)

- Common patterns
  - Singleton, Container-Item
- Supported types
- Creating responses
  - *Response* class
- Building URIs
  - *UriBuilder, UriInfo* classes
- Exceptions
  - *WebApplicationException, ExceptionMapper* classes

# Topics (Continued)

- Deployment options
  - Web app or Java SE app
- Security
  - *@Context, SecurityContext class*
- Tools
- Samples

# JAX-RS Quick Overview

# Problem in Using Servlet API For Exposing a Resource (Too much coding)

```
public class Artist extends HttpServlet {

    public enum SupportedOutputFormat {XML, JSON};

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String accept = request.getHeader("accept").toLowerCase();
        String acceptableTypes[] = accept.split(",");
        SupportedOutputFormat outputType = null;
        for (String acceptableType: acceptableTypes) {
            if (acceptableType.contains("*/*") || acceptableType.contains("application/*") ||
                acceptableType.contains("application/xml")) {
                outputType=SupportedOutputFormat.XML;
                break;
            } else if (acceptableType.contains("application/json")) {
                outputType=SupportedOutputFormat.JSON;
                break;
            }
        }
        if (outputType==null)
            response.sendError(415);
        String path = request.getPathInfo();
        String pathSegments[] = path.split("/");
        String artist = pathSegments[1];
        if (pathSegments.length < 2 && pathSegments.length > 3)
            response.sendError(404);
        else if (pathSegments.length == 3 && pathSegments[2].equals("recordings")) {
            if (outputType == SupportedOutputFormat.XML)
                writeRecordingsForArtistAsXml(response, artist);
            else
                writeRecordingsForArtistAsJson(response, artist);
        } else {
            if (outputType == SupportedOutputFormat.XML)
                writeArtistAsXml(response, artist);
            else
                writeArtistAsJson(response, artist);
        }
    }
    private void writeRecordingsForArtistAsXml(HttpServletResponse response, String artist) { ... }
    private void writeRecordingsForArtistAsJson(HttpServletResponse response, String artist) { ... }
    private void writeArtistAsXml(HttpServletResponse response, String artist) { ... }
    private void writeArtistAsJson(HttpServletResponse response, String artist) { ... }
}
```

# Design Goals of JAX-RS: Java API for RESTful Web Services

- Clear mapping to REST concepts
  - Everything is a resource
  - Every resource is address'able via URI
  - HTTP methods provides uniform interface
  - Representations
- High level and Declarative
  - Use @ annotation in POJOs
- Generate or hide the boilerplate code
  - No need to write boilerplate code for every app

# JAX-RS (JSR 311) & Jersey

- Jersey – reference implementation of JAX-RS
  - Download it from http://jersey.dev.java.net
  - Comes with Glassfish, Java EE 6
  - Tools support in NetBeans

- Four other open source implementations:
  - Apache CXF
  - JBoss RESTEasy
  - Restlet
  - Triaxrs

# Creating a Resource with URI using *@Path* Annotation

# How to Create Root Resource Class?

- Create a POJO (Plain Old Java Object) and annotate it with *@Path* annotation with relative URI path as value
  - The base URI is the application context
- Implement resource methods inside the POJO with HTTP method annotations
  - *@GET, @PUT, @POST, @DELETE*

# Example: Root Resource Class

```
// Assume the application context is http://example.com/catalogue, then
//
// GET http://example.com/catalogue/widgets  - handled by the getList ()
// method
//
// GET http://example.com/catalogue/widgets/nnn - handled by the
// getWidget() method.

@Path("widgets")
public class WidgetsResource {
  @GET
  String getList() {...}

  @GET @Path("{id}")
  String getWidget(@PathParam("id") String id) {...}
}
```

# URI Path Template

# What is URI Path Template?

- URI path templates are URIs with variables embedded within the URI syntax.
- To obtain the value of the *username* variable, the *@PathParam* may be used on method parameter of a request method

```
// Will respond to http://example.com/users/SangShin
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

# @PathParam, @QueryParam

- 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

# Example: @PathParam, @QueryParam

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

   // Example request: http://host/catalog/items/123
   @Path("{id}/")
   ItemResource getItemResource(@PathParam("id")Long id){
   ...
   }

   // Example request: http://host/catalog/items/?start=0
   @GET
   ItemsConverter get(@QueryParam("start")int start) {
        ...
   }

}
```

# URI Path template variable with Reg. Expression pattern

- @Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
  - *username* variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character.
  - If a user name does not match that a 404 (Not Found) response will occur.

# Demo:
## Building & Testing "Hello World" RESTful Web Service

# Sub-Resource Locators

# What is Sub-resource locator?

- Sub-resource locator is a method
  - Annotated with *@Path*
  - Not annotated with *@GET* or *@POST*
  - Returns sub-resource, which contains methods with @GET or @POST annotations
- Sub-resource locators support polymorphism
  - A sub-resource locator may return different sub-types depending on the request (for example a sub-resource locator could return different sub-types dependent on the role of the principle that is authenticated).

# Example: Sub-resource Locator

```
@Path("/item")
public class ItemResource {

   // Sub-resource locator
   @Path("content")
   public ItemContentResource getItemContentResource() {
      return new ItemContentResource();
   }
   @GET
   @Produces("application/xml")
   public Item get() { ... }
}

public class ItemContentResource {
   @GET public Response get() { ... }
   @PUT @Path("{version}")
   public void put(
         @PathParam("version") int version,
         @Context HttpHeaders headers,
         byte[] in) { ... }
}
```

# HTTP Methods Annotations: @GET, @POST, @PUT, @DELETE

# Clear mapping to REST concepts: HTTP Methods

- Annotate resource class methods with standard HTTP method
  - `@GET`, `@PUT`, `@POST`, `@DELETE`

# Uniform interface: methods on resources

```
@Path("/employees")
class Employees {
   @GET <type> get() { ... }
   @POST <type> create(<type>) { ... }
}
```

---

```
@Path("/employees/{eid}")
class Employee {
   @GET <type> get(...) { ... }
   @PUT void update(...) { ... }
   @DELETE void delete(...) { ... }
}
```

Java method name is not significant.

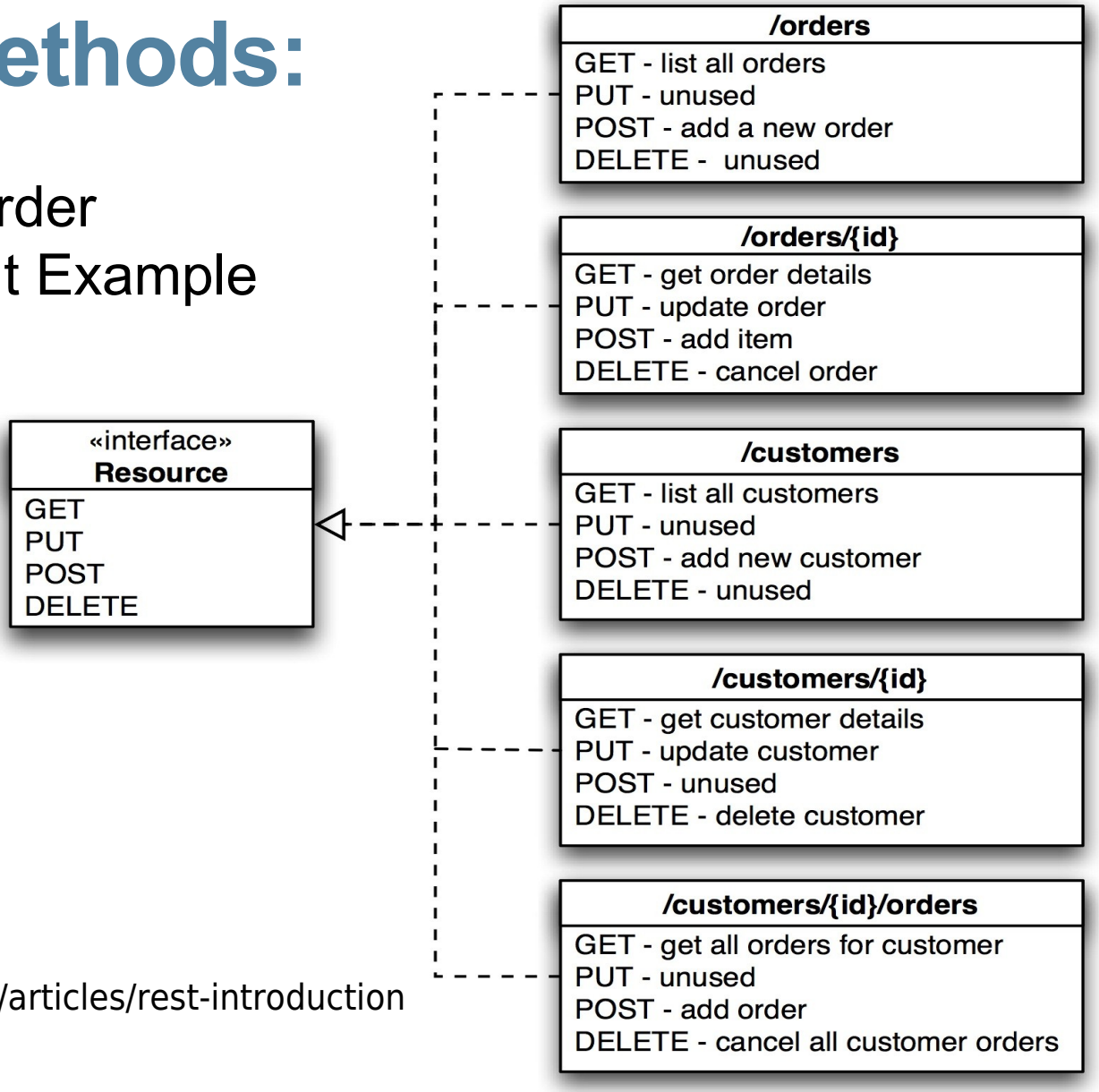
# CRUD Operations are Performed through “HTTP method” + “Resource”

## CRUD Operations

| | HTTP method | Resource |
|---|---|---|
| Create (Single) | POST | Collection URI |
| Read (Multiple) | GET | Collection URI |
| Read (Single) | GET | Entry URI |
| Update (Single) | PUT | Entry URI |
| Delete (Single) | DELETE | Entry URI |

# HTTP Methods:

Customer Order Management Example




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

# HTTP Methods:

- /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 all orders of a customer

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

# GET /customers

```
// Handles http://localhost:8080/CustomerDB/resources/customers/
@Path("/customers/")
public class CustomersResource {
    /**
     * Get method for retrieving a collection of Customer instance in XML format.
     * @return an instance of CustomersConverter
     */
    @GET
    @Produces({"application/xml", "application/json"})
    public CustomersConverter get(...) {
        try {
            return new CustomersConverter(getEntities(start, max, query),
                                          uriInfo.getAbsolutePath(), expandLevel);
        } finally {
            PersistenceService.getInstance().close();
        }
    }
```

# GET /customer/{id}

```
// Handles http://localhost:8080/CustomerDB/resources/customers/1
@Path("/customers/")
public class CustomersResource {
    ...
    /**
     * Returns a dynamic instance of CustomerResource used for entity navigation.
     * @return an instance of CustomerResource
     */
    @Path("{customerId}/")
    public CustomerResource
    getCustomerResource(@PathParam("customerId") Integer id) {
        CustomerResource resource =
                        resourceContext.getResource(CustomerResource.class);
        resource.setId(id);
        return resource;
    }
```

# Representations: @Produces & @Consumes

# Formats in HTTP

## Request

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

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>
```

State transfer

Representation

# Multiple Representations

- Resources can have multiple representations
  - Specified through 'Content-type' HTTP response header
  - Acceptable format through 'Accept' HTTP request header
- Type of representations
  - text/html – regular web page
  - application/xhtml+xml – in XML
  - application/rss+xml – as a RSS feed
  - application/octet-stream – an octet stream
  - application/rdf+xml – RDF format

# @Produces

- Used to specify the MIME media types of representations a resource can produce and send back to the client
- Can be applied at both the class and method levels
  - Method level overrides class level

# @Produces

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
   // defaults to the MIME type of the @Produces
   // annotation at the class level
    @GET
    public String doGetAsPlainText() {
        ...
    }
    // overrides the class-level @Produces setting
    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

# Choice of Mime Type Based on Client Preference

- If a resource class is capable of producing more that one MIME media type, then the resource method chosen will produce the most acceptable media type as declared by the client.
  - Accept header of the HTTP request
- For example,
  - *Accept: text/plain* - *doGetAsPlainText* method will be invoked
  - *Accept: text/plain;q=0.9, text/html* - *doGetAsHtml* method will be invoked

# Multiple types can be declared

```
@GET
// More than one media type may be declared in the same
// @Produces annotation.
//
// The doGetAsXmlOrJson method will get invoked if either of the
// media types "application/xml" and "application/json" are acceptable.
// If both are equally acceptable then the former will be chosen
// because it occurs first.
@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
...
}
```

# @Consumes

- Used to specify the MIME media types of representations a resource can consume
- Can be applied at both the class and method levels
  - Method level override a class level
- A container is responsible for ensuring that the method invoked is capable of consuming the media type of the HTTP request entity body.
  - If no such method is available the container must respond with a HTTP "415 Unsupported Media Type"

# @Consumes

```
@POST
// Consume representations identified by the MIME media type "text/plain".
// Notice that the resource method returns void. This means no representation
// is returned and response with a status code of 204 (No Content) will be
// returned.
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

# Demo:
## Building RESTful Web Services from Database Tables

# XML Binding via JAXB

# Producing via @Produces

```
@Post
@Consumes("application/x-www-form-urlencoded")
@Produces("application/xml")

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

    ...
```

converted to XML

Converted to a map for accessing form's fields

# Produces XML from Complex Object

```
// Method that returns XML of complex object graph
@GET
@Produces({"application/xml", "application/json"})
public CustomerConverter get(@QueryParam("expandLevel")
                             @DefaultValue("1") int expandLevel) {
    try {
        // See the next slide for the CustomerConverter class
        return new CustomerConverter(getEntity(),
                                     uriInfo.getAbsolutePath(), expandLevel);
    } finally {
        PersistenceService.getInstance().close();
    }
}
```

# Produces XML from Complex Object

```
@XmlRootElement(name = "customer")
public class CustomerConverter {
    private Customer entity;
    private URI uri;
    private int expandLevel;

    ...

    @XmlElement
    public Integer getCustomerId() {
        return (expandLevel > 0) ? entity.getCustomerId() : null;
    }

    @XmlElement
    public String getZip() {
        return (expandLevel > 0) ? entity.getZip() : null;
    }
    ...
```

# Produces XML from Complex Object

```
<?xml version="1.0" encoding="UTF-8"?>
  <customer
uri="http://localhost:8080/CustomerDB/resources/customers/2/">
      <addressline1>9754 Main Street</addressline1>
      <addressline2>P.O. Box 567</addressline2>
      <city>Miami</city>
      <creditLimit>50000</creditLimit>
      <customerId>2</customerId>
      <discountCode
uri="http://localhost:8080/CustomerDB/resources/customers/2/discou
ntCode/"/>
      <email>www.tsoftt.com</email>
      <fax>305-456-8889</fax>
      <name>Livermore Enterprises</name>
      <phone>305-456-8888</phone>
      <state>FL</state>
      <zip>33055</zip>
  </customer>
```

# Common Patterns

# Common Patterns

- Singleton
- Container-Item
- Client-Controlled Container-Item

# Common Patterns: Container-Item Server in control of URI

- Container – a collection of items
- List catalog items:
  - *GET /catalog/items*
- Add item to container:
  - *POST /catalog/items* with a new item in request
  - URI of item returned in HTTP response header
    - e.g. http://host/catalog/items/1
- Update item
  - *PUT /catalog/items/1* with updated item in request
- Good example: Atom Publishing Protocol

# Common Patterns: Container-Item Client in control of URI

- List key-value pairs: GET /map
- Put new value to map: PUT /map/{key}
  - with entry in request
- Read value: GET /map/{key}
- Update value: PUT /map/{key}
  - with updated value in request
- Remove value: DELETE /map/{key}
- Good example: Amazon S3

# Demo:
## Building RESTful Web Services using Various Patterns

# Creating Response with Response Class

# Building Responses

- Sometimes it is necessary to return additional information in response to a HTTP request. Such information may be built and returned using *Response* and *Response.ResponseBuilder*
- Response building provides other functionality such as setting the entity tag and last modified date of the representation.

# HTTP Response Codes

- JAX-RS returns default response codes
  - GET returns 200 OK
  - PUT returns 201 CREATED
- HTTP response codes
  - http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

200 OK
201 Created
202 Accepted
203 Non-Authoritative Information
204 No Content
205 Reset Content
206 Partial Content
207 Multi-Status
226 IM Used

# HTTP response for HTTP Post for Creating an item

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

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

# Creating a Response using Response class

```
@POST
@Consumes("application/xml")
// A common RESTful pattern for the creation of a new resource is to support a
// POST response that returns a 201 (Created) status code and a Location
// header whose value is the URI to the newly created resource
public Response post(String content) {
    URI createdUri = ...
    create(content);
    return Response.created(createdUri).build();
}
```

# Buidling URIs with UriBuilder & UriInfo Classes (Linking Things Together)

# UriBuilder Class

- A very important aspect of REST is hyperlinks, URIs, in representations that clients can use to transition the Web service to new application states
  - "hypermedia as the engine of application state"
- Building URIs and building them safely is not easy with *java.net.URI*, which is why JAX-RS has the *UriBuilder* class that makes it simple and easy to build URIs safely

# UriInfo Class

- Provides base URI information
  - The URIs that will be returned are typically built from the base URI the Web service is deployed at or from the request URI

# UriBuilder & UriInfo

```
@Path("/users/")
public class UsersResource {

    @Context UriInfo uriInfo;
    ...

    @GET
    @Produces("application/json")
    // The getUsersAsJsonArray method constructs a JSONArrray where
    // each element is a URI identifying a specific user resource
    public JSONArray getUsersAsJsonArray() {
        JSONArray uriArray = new JSONArray();
        for (UserEntity userEntity : getUsers()) {
            UriBuilder ub = uriInfo.getAbsolutePathBuilder();
            URI userUri = ub.
                    path(userEntity.getUserid()).
                    build();
            uriArray.put(userUri.toASCIIString());
        }
        return uriArray;
    }
}
```

# UriBuilder for extracting query parameters

```
// UriBuilder can be used to build/replace query parameters.
// URI templates can also be declared, for example the following will
// build the URI "http://localhost/xyz?name=value":
UriBuilder.fromUri("http://localhost/").
                                path("{a}").
                                queryParam("name", "{value}").
                                build("segment", "xyz");
```

# Exception

# NotFoundException

```
@Path("items/{itemid}/")
 public Item getItem(@PathParam("itemid") String itemid) {
        Item i = getItems().get(itemid);

        // Shows the throwing of a NotFoundException.
        // The NotFoundException exception is a Jersey specific
        // exception that extends WebApplicationException
        //and builds a HTTP response with
        // the 404 status code and an optional message
        //as the body of the response:
        if (i == null)
            throw new NotFoundException("Item,
                                                " + itemid + ", is not found");

        return i;
 }
```

# WebApplicationException

```
public class NotFoundException extends WebApplicationException {

    /**
     * Create a HTTP 404 (Not Found) exception.
     */
    public NotFoundException() {
        super(Responses.notFound().build());
    }

    /**
     * Create a HTTP 404 (Not Found) exception.
     * @param message the String that is the entity of the 404 response.
     */
    public NotFoundException(String message) {
        super(Response.status(Responses.NOT_FOUND).
                entity(message).type("text/plain").build());
    }

}
```

# ExceptionMapper

```
// In other cases it may not be appropriate to throw instances of
// WebApplicationException, or classes that extend
// WebApplicationException, and instead it may be preferable
// to map an existing exception to a response.
// For such cases it is possible to use the ExceptionMapper interface.
// For example, the following maps the EntityNotFoundException to a
// 404 (Not Found) response.
@Provider
public class EntityNotFoundMapper implements
      ExceptionMapper<javax.persistence.EntityNotFoundException> {

  public Response
  toResponse(javax.persistence.EntityNotFoundException ex) {
      return Response.status(404).
         entity(ex.getMessage()).
         type("text/plain").
         build();
   }
}
```

# Security

# Getting SecurityContext

- Security information is available by obtaining the *SecurityContext* using *@Context*, which is essentially the equivalent functionality available on the *HttpServletRequest*
- SecurityContext can be used in conjunction with sub-resource locators to return different resource depending on a user's role
  - For example, a sub-resource locator could return a different resource if a user is a preferred customer:

# SecurityContext

```
@Path("basket")
// Sub-resource locator could return a different resource if a user
// is a preferred customer:
public ShoppingBasketResource get(@Context SecurityContext sc) {
    if (sc.isUserInRole("PreferredCustomer") {
       return new PreferredCustomerShoppingBaskestResource();
    } else {
       return new ShoppingBasketResource();
    }
}
```

# Deployment Options

# Servlet

- JAX-RS applications are packaged in WAR like a servlet
- For JAX-RS aware containers (Java EE 6)
  - web.xml can point to Application subclass
- For non-JAX-RS aware containers (Java EE 5)
  - web.xml points to the servlet implementation of JAX-RS runtime
- Application declares resource classes
  - Can create your own by subclassing
  - Reuse PackagesResourceConfig

# JavaSE 6

- Use RuntimeDelegate to create instance of end point
- Provides configuration information
  - Subclass of Application
- Jersey supports Grizzly, LW HTTP server and JAX-WS provider

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

# Tools & Samples

# Development Tools

- IDE – for general purpose RESTful Web service development
  - NetBeans, Eclipse
- Client tools – for sending HTTP requests
  - “Poster” plug-in to Firefox
  - Several command line tools
    - curl  http://curl.haxx.se/
  - soapUI
- Browser

# Demo:
## Using Poster Plug-in of Firefox

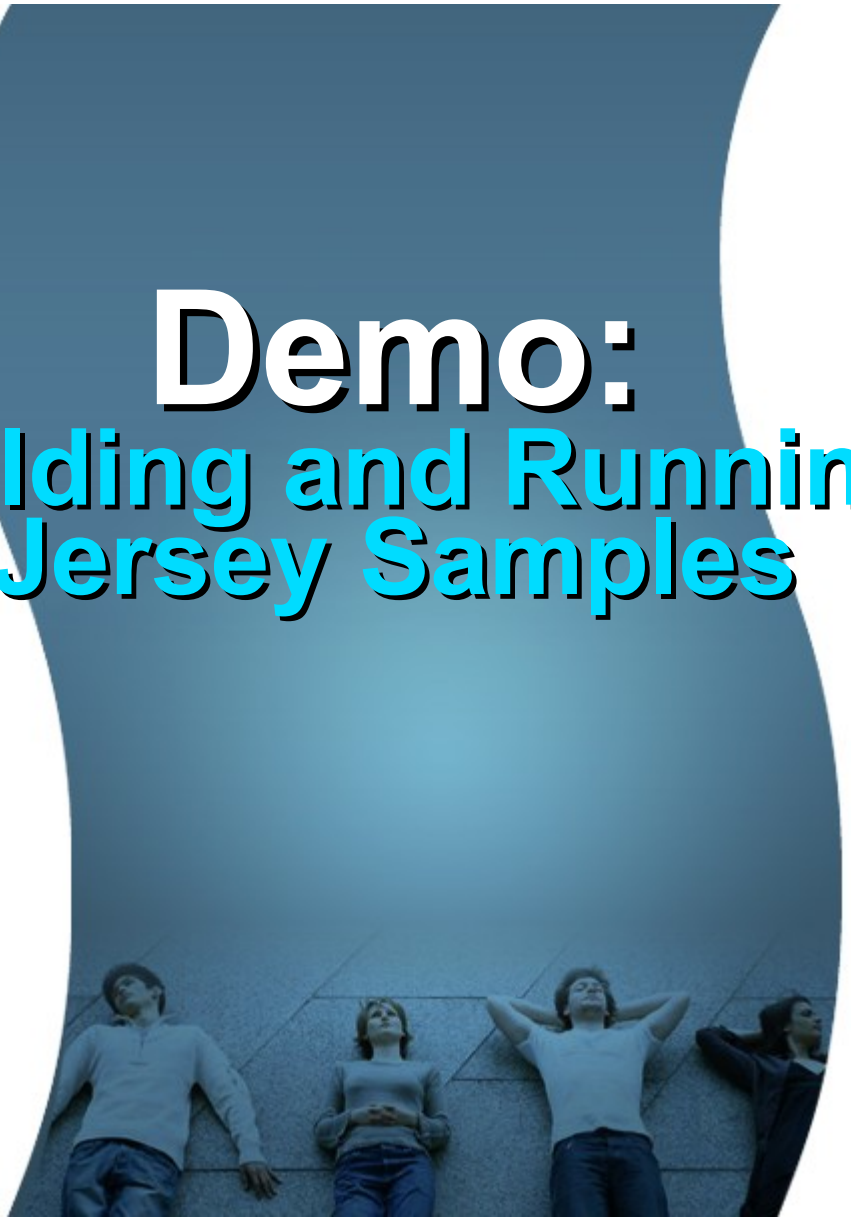
# Samples

- http://download.java.net/maven/2/com/sun/jersey/samples/jersey-samples/1.0.3/jersey-samples-1.0.3-project.zip (They are Maven projects)
- HelloWorld: This is how everybody starts using Grizzly as in the process HTTP server.
- HelloWorld Web app: This is how everybody starts using a Web application.
- Bookmark Web app: Demonstrates how to use JPA in the backend.
- Bookstore Web app: Demonstrates how to use polymorphism with resources and views that are JSP pages.
- EntityProvider: Demonstrates pluggable entity providers.
- Extended WADL Web app:Demonstrates how to customize generation of WADL.
- Generate WADL: Demonstrates how to customize generation of WADL.
- Jaxb: Demonstrates the use of JAXB-based resources.
- JMaki-backend Web app: Provides JSON to be consumed by jMaki widgets.
- JsonFromJaxb: Demonstrates how to use JSON representation of JAXB-based resources.

# Samples

- Mandel: A Mandelbrot service written in Scala using Scala's actors to scale-up the calculation.
- OptimisticConcurrency: Demonstrates the application of optimistic concurrency to a web resource.
- SimpleAtomServer:Simple Atom server that partially conforms to the Atom Publishing Format and Protocol.
- SimpleConsole: Demonstrates a simple service using Grizzly.
- SimpleServlet: Demonstrates how to use a Servlet container.
- Sparklines: A sparklines application inspired by Joe Gregorio's python application.
- Spring annotations: An example leveraging Jersey's Spring-based annotation support.
- StorageService: Demonstrates a basic in-memory web storage service.

# Demo:
## Building and Running Jersey Samples



# Thank you!

**Check JavaPassion.com Codecamps!**
**http://www.javapassion.com/codecamps**
**"Learn with Passion!"**