JAX-RS Basics I

Sang Shin
www.JPassion.com
"Learn with Passion!"

Topics

- JAX-RS quick overview
- Creating resources
  - @Path
- URI Path template
  - *@PathParam, @QueryParam, @FormParam, @HeaderParam*
- HTTP method annotations (Uniform interface)
  - *@GET, @POST, @PUT, @DELETE*
- Creating responses
  - Response class
- Representations (Formats)
  - *@Produces, @Consumes*

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

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

Implementations of JAX-RS (JSR 311)

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

- Other open source implementations of JAX-RS
  - Apache CXF
  - JBoss RESTEasy
  - Restlet

Development Tools

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

Lab:

Exercise 0: Installation of REST tools
4360_ws_jaxrs_basics1.zip

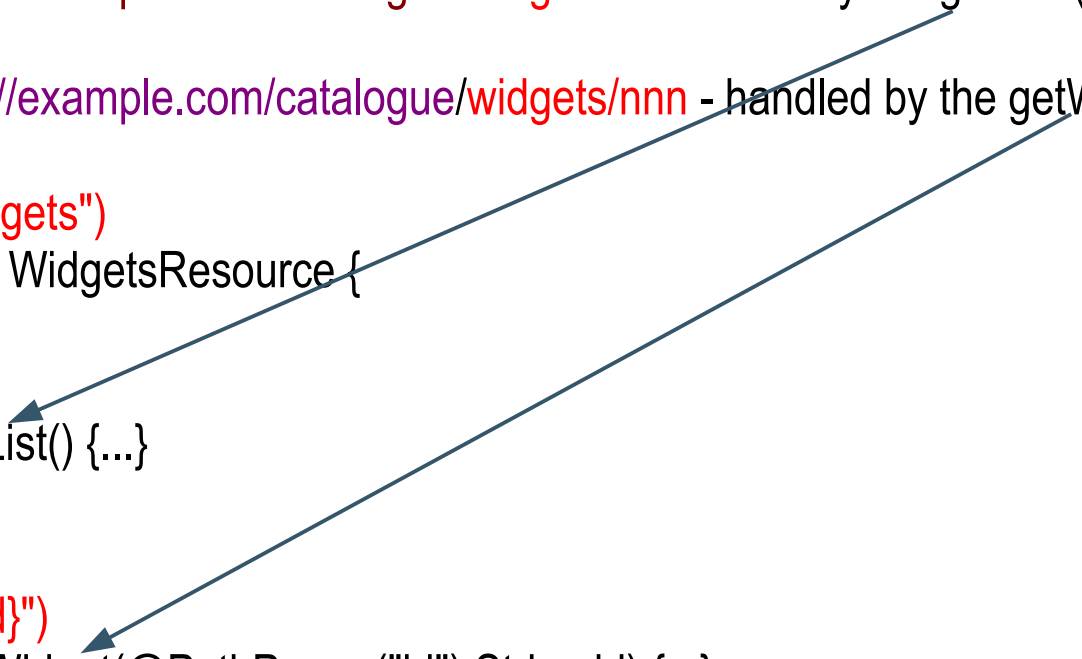
Creating a Resource with URI using *@Path* Annotation

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*

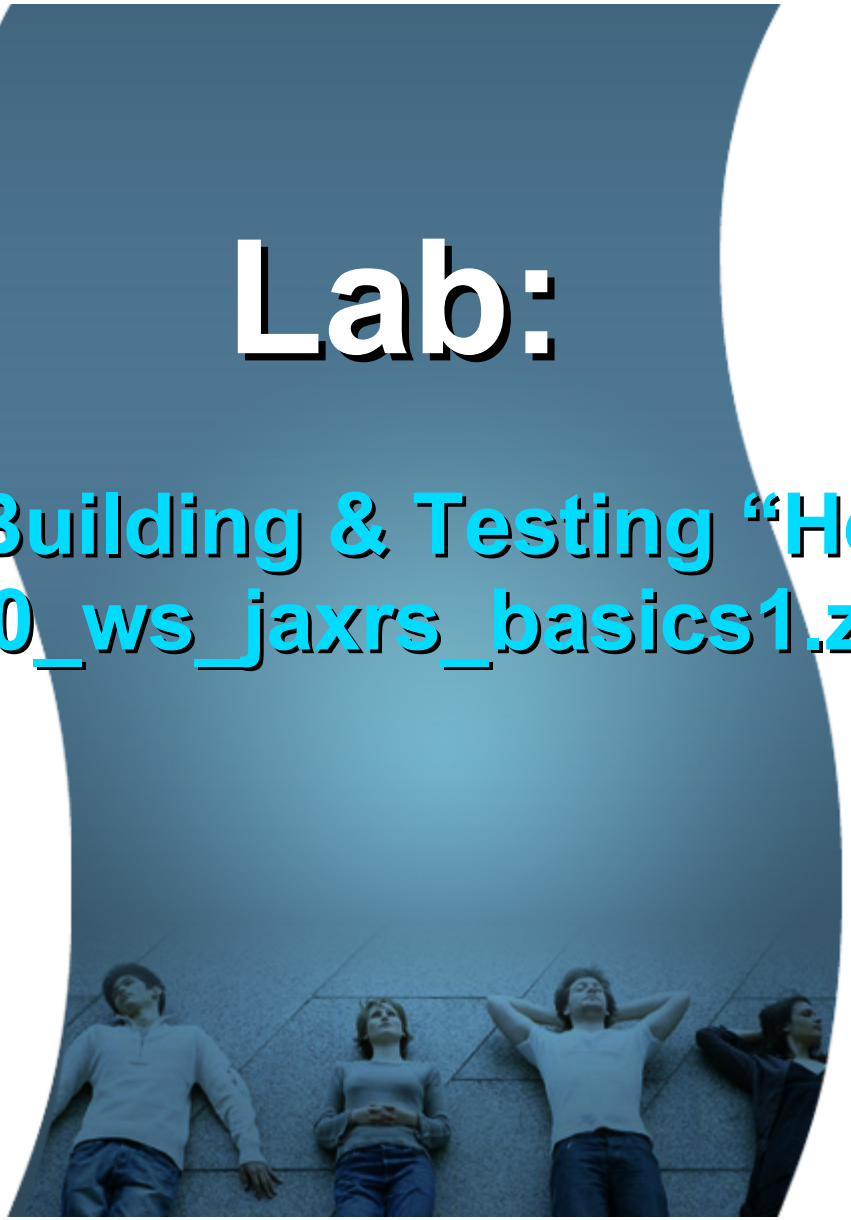
Example: Root Resource Class

```
// Assume the application context is http://example.com/catalogue
//
// 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) {...}
}
```



Lab:

Exercise 1: Building & Testing “Hello World”
4360_ws_jaxrs_basics1.zip



URI Path Template (@PathParam, @QueryParam, @FormParam, @HeaderParam)

What is URI Path Template?

- URI path templates are “URIs with variables embedded”
- The variable captures client request value

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

    @GET
    @Produces("text/xml")
    //  In order to obtain the value of the username variable, @PathParam
    //  is used on a method parameter
    public String getUser(@PathParam("username") String userName) {
        // “userName” variable has value of “SangShin”
    }
}
```

@PathParam, @QueryParam

- Annotated method parameters extract client request data
  - `@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, then a 404 (Not Found) response will be sent

@FormParam

- Binds the value(s) of a form parameter contained within a request entity body to a resource method parameter

```
@POST
 public Response addUser(
        @FormParam("name") String name,
        @FormParam("age") int age) {
    String output = "A User is added: " + name + ", age : " + age;
    return Response.status(200).entity(output).build();
 }
```

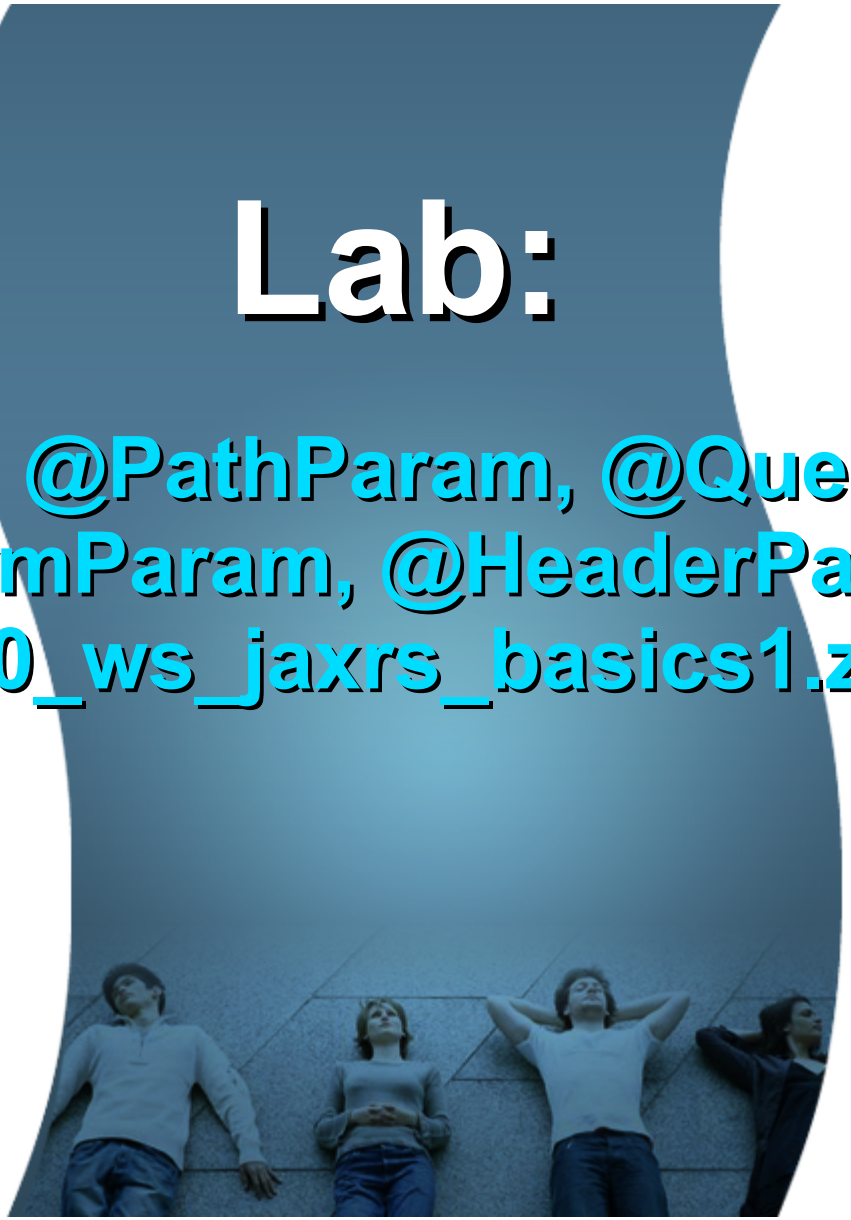
@HeaderParam

- Binds the value(s) of a HTTP header to a resource method parameter, resource class field, or resource class bean property

```
@GET
@Path("/get")
public Response addUser(@HeaderParam("accept") String accept) {
    String output = "addUser is called, accept header: " + accept;
    return Response.status(200).entity(output).build();

}
```

Lab:

Exercise 2: @PathParam, @QueryParam, @FormParam, @HeaderParam
4360_ws_jaxrs_basics1.zip



HTTP Method 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.

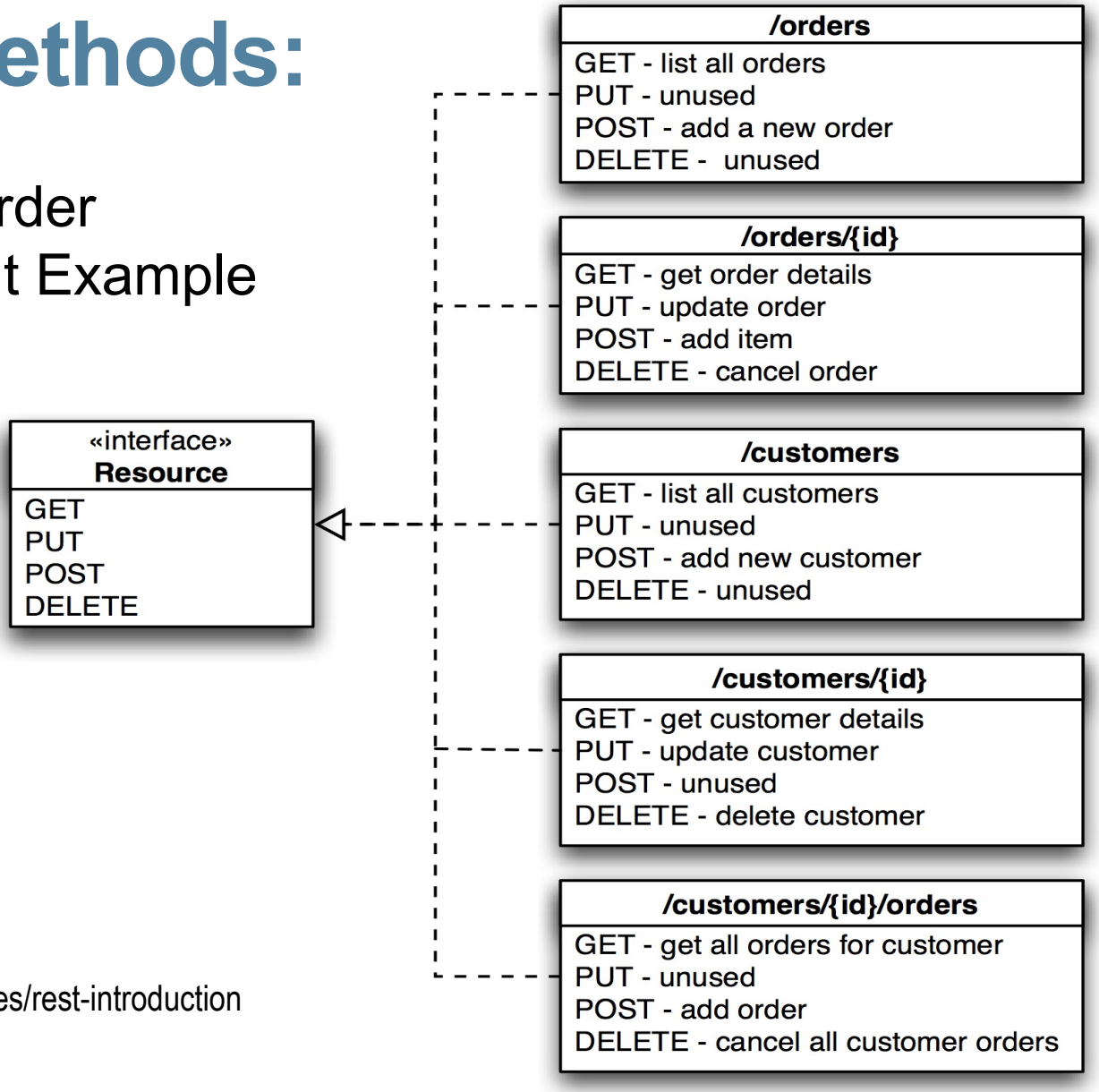
CRUD Operations are Performed through “HTTP method” + “Resource”

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

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
  - Acceptable format through 'Accept' HTTP request header
  - Specified through 'Content-type' HTTP response 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 - “test/plain”
    @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, using the example code in previous slide
  - *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

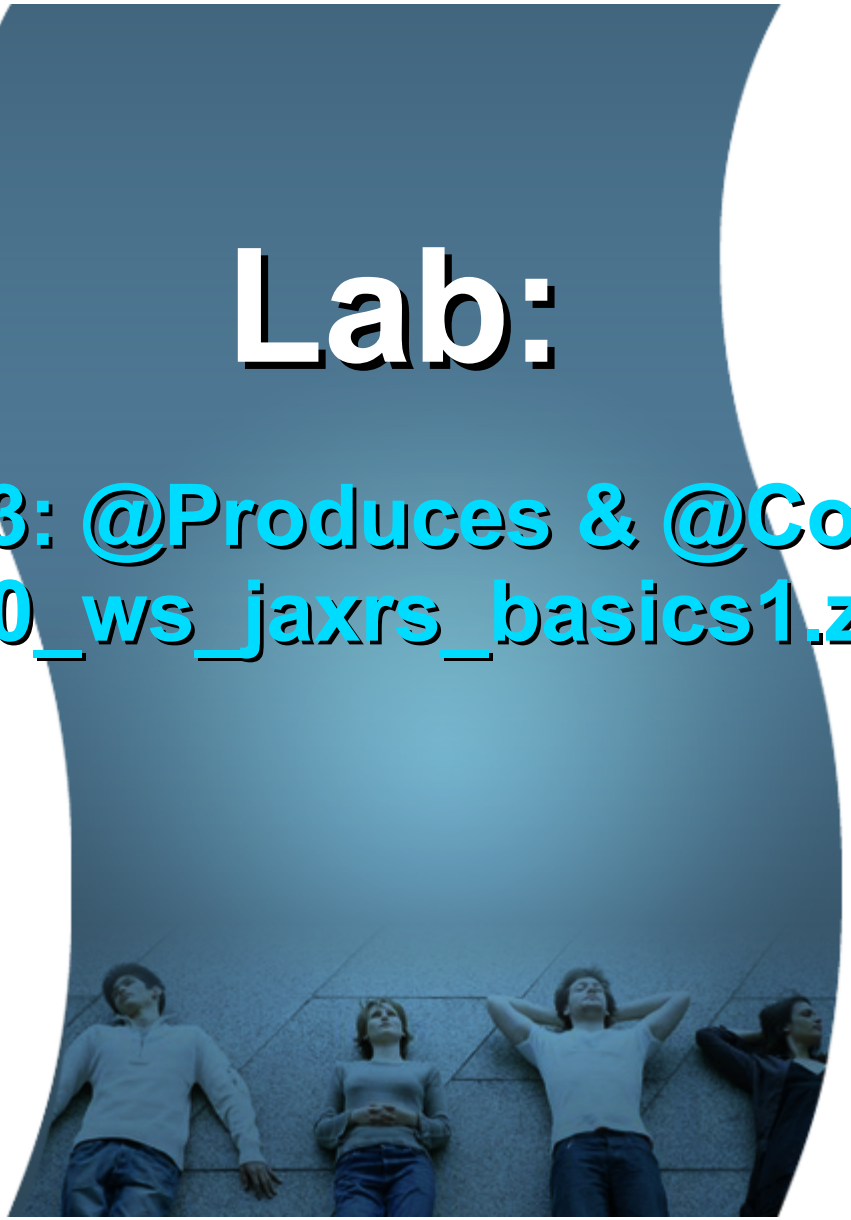
- Used to specify the MIME media types of representations a resource can consume
- Can be applied at both the class and method levels
  - Method level override a class level
- A container is responsible for ensuring that the method invoked is capable of consuming the media type of the HTTP request entity 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 in this example. 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
}
```

Lab:

Exercise 3: @Produces & @Consumes
4360_ws_jaxrs_basics1.zip



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/discountCode/"/>
      <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>
```

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*
  - *Response.ResponseBuilder*
- Response building provides other functionality such as
  - setting the entity tag
  - 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 a new resource

- 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

Example: HTTP response for HTTP Post for Creating a new resource

```
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();
}
```

Creating a Response using Response class

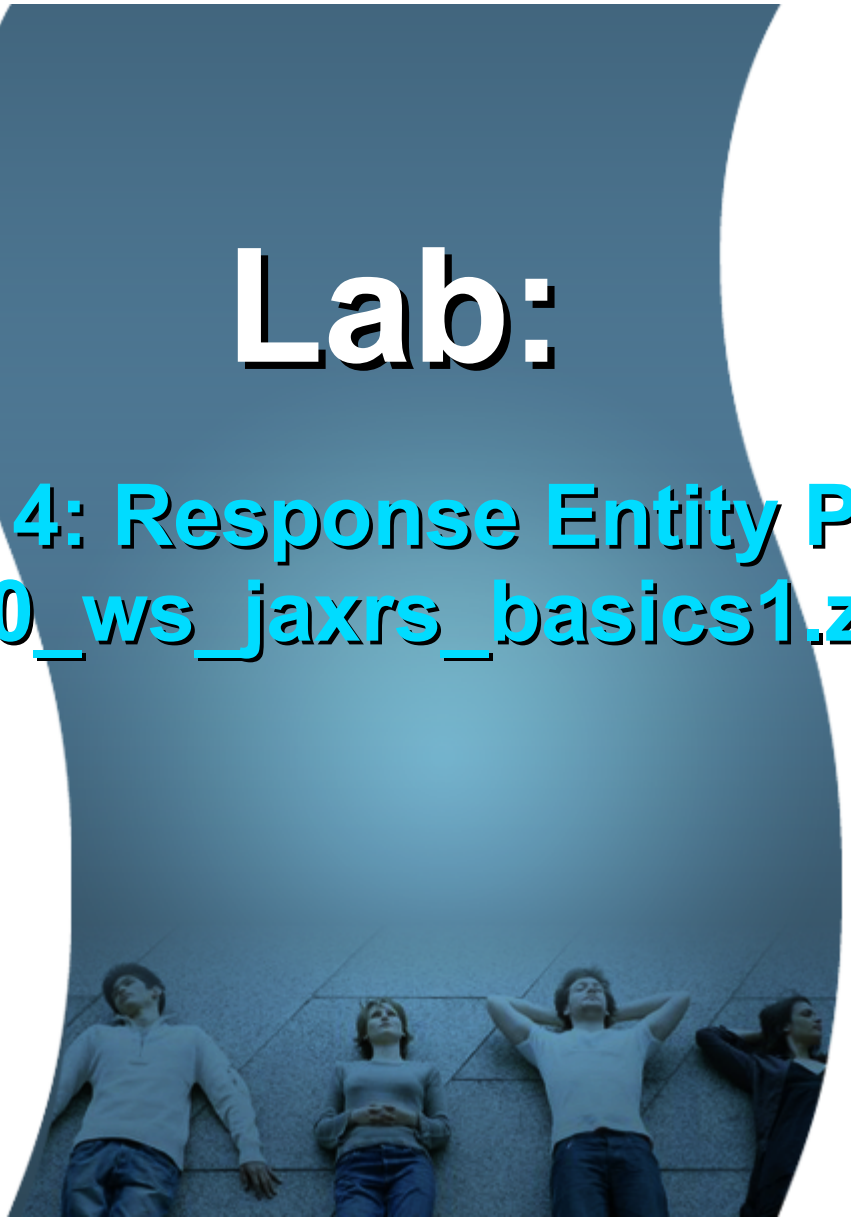
```
@GET
@Path("/getUserInXMLviaResponse")
@Produces(MediaType.APPLICATION_XML)
public Response getUserInXMLviaResponse() {

    User user = new User();
    user.setName("Sang Shin");
    user.setAge(100);
    final int DUMMY_INDEX = 2;

    UriBuilder uriBuilder = UriBuilder.fromUri(uriInfo.getRequestUri());

    return Response.created(uriBuilder.build(DUMMY_INDEX)).
            entity(user).
            type(MediaType.APPLICATION_XML).
            build();

}
```

Creating a Response using Entity Provider

```
@GET
@Path("/getXMLviaEntityProvider")
@Produces(MediaType.APPLICATION_XML)
public User getXMLviaEntityProvider() {

    User user = new User();
    user.setName("Sang Shin");
    user.setAge(100);

    return user;

}
```

Lab:

Exercise 4: Response Entity Provider
4360_ws_jaxrs_basics1.zip



Learn with Passion!
JPassion.com