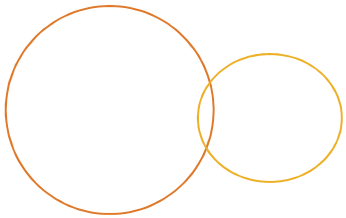
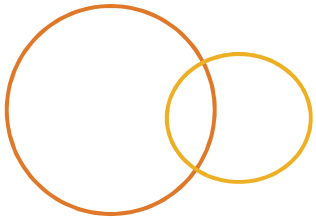
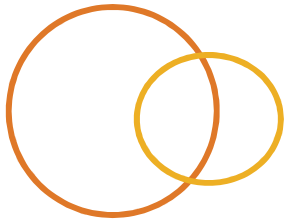
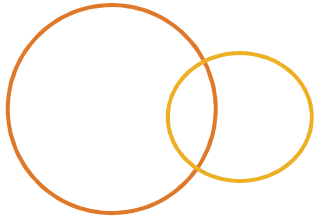
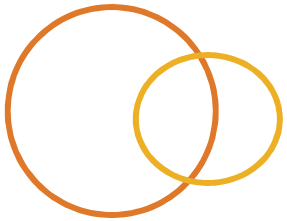
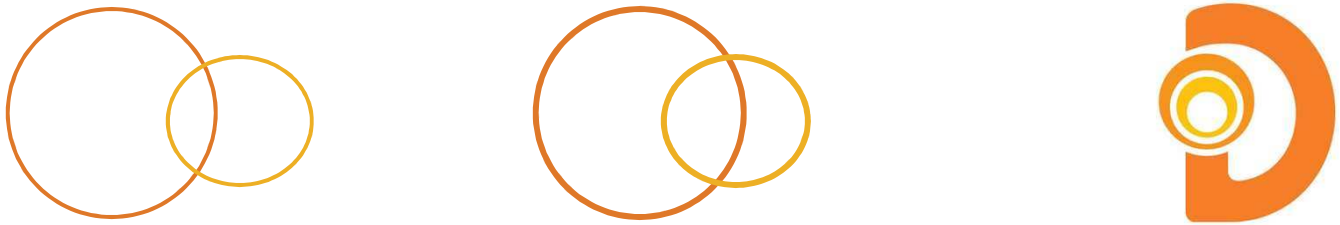




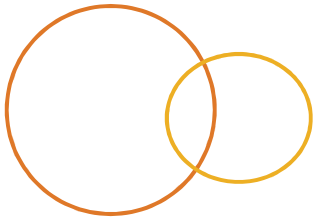

Introduction to REST With JAX-RS

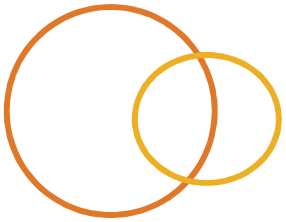




Before We Begin



Logistics





- Start and end times
  - Please be prompt, it's not fair to others to be late and then take up time asking questions to catch up
- Breaks
  - Approximately every hour, ten minutes (but please be prompt)
  - Important opportunity to refresh your brain (focus / willpower are more limited than you think)
  - Stand up and leave the room (don't just sit and read email!)

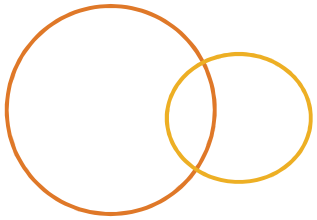
About Learning

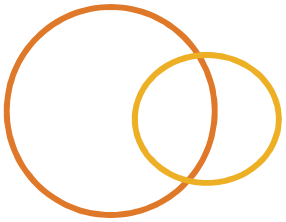
- Staying awake is crucial!
  - If you feel yourself dragging, feel free to stand up, move, at the back of the room, blood flow will help
  - If it's been too long since we took a break, speak up, we'll take it
- Please, ask questions, participate in discussions
  - ***Your*** questions, and those of your peers, make it more interesting to your brain, help you learn
  - Your examples, your challenges, are all excellent ways to help your brain recognize relevance
  - Sometimes, a discussion must be curtailed, or deferred, to keep overall course on track

About Learning

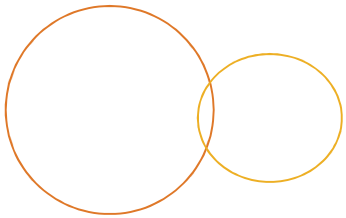
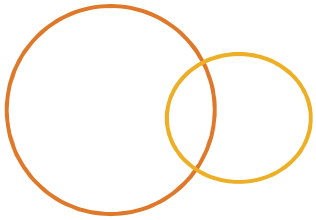
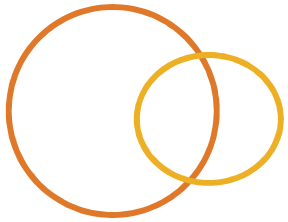
- Raise your topics
  - If practical in the time allotted, we'll talk about any related topics too; it's *your* course
- Lab exercises
  - "Doing" is the most important part. If you already know this topic, use the time to push your knowledge, try something new in the API
  - Your own investigation is preferred (your brain knows it's relevant) but suggested lab guide may be used if you have a "blank sheet" problem.

Say Hello!



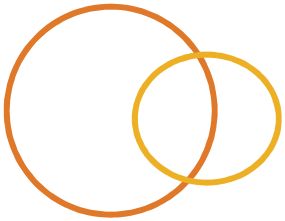
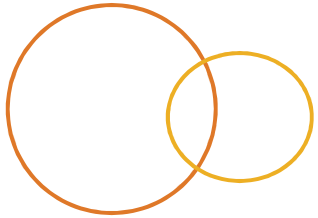
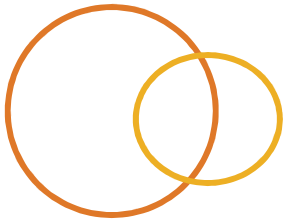


- Briefly:
  - Who you are
  - Approximate experience level with these topics
  - Number one thing you want to be able to do by the end of this
  - Optional: something of “human interest” about you (hobby, etc.)

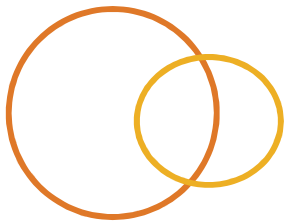





Course Overview

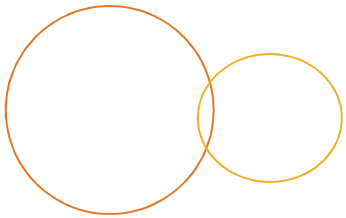
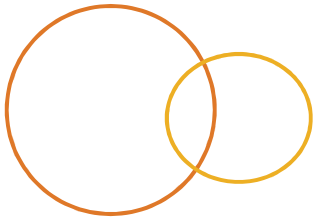
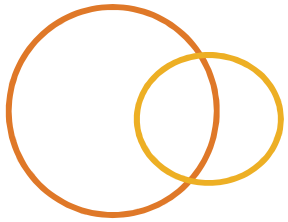


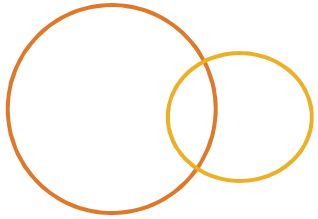
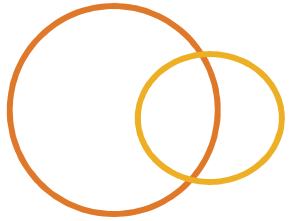

Course Overview



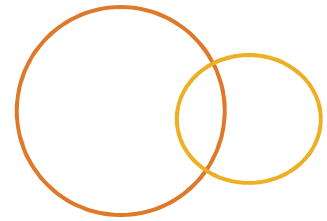
- Overview Of REST
- Getting Started With JAX-RS
- Core JAX-RS Features
- More Injection Features of JAX-RS
- Controlling the Response
- Handling HTTP entities
- Error Handling
- Sub-resource navigation and Design Considerations
- JAX-RS 2.x Client API

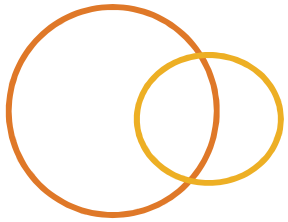





Overview Of REST

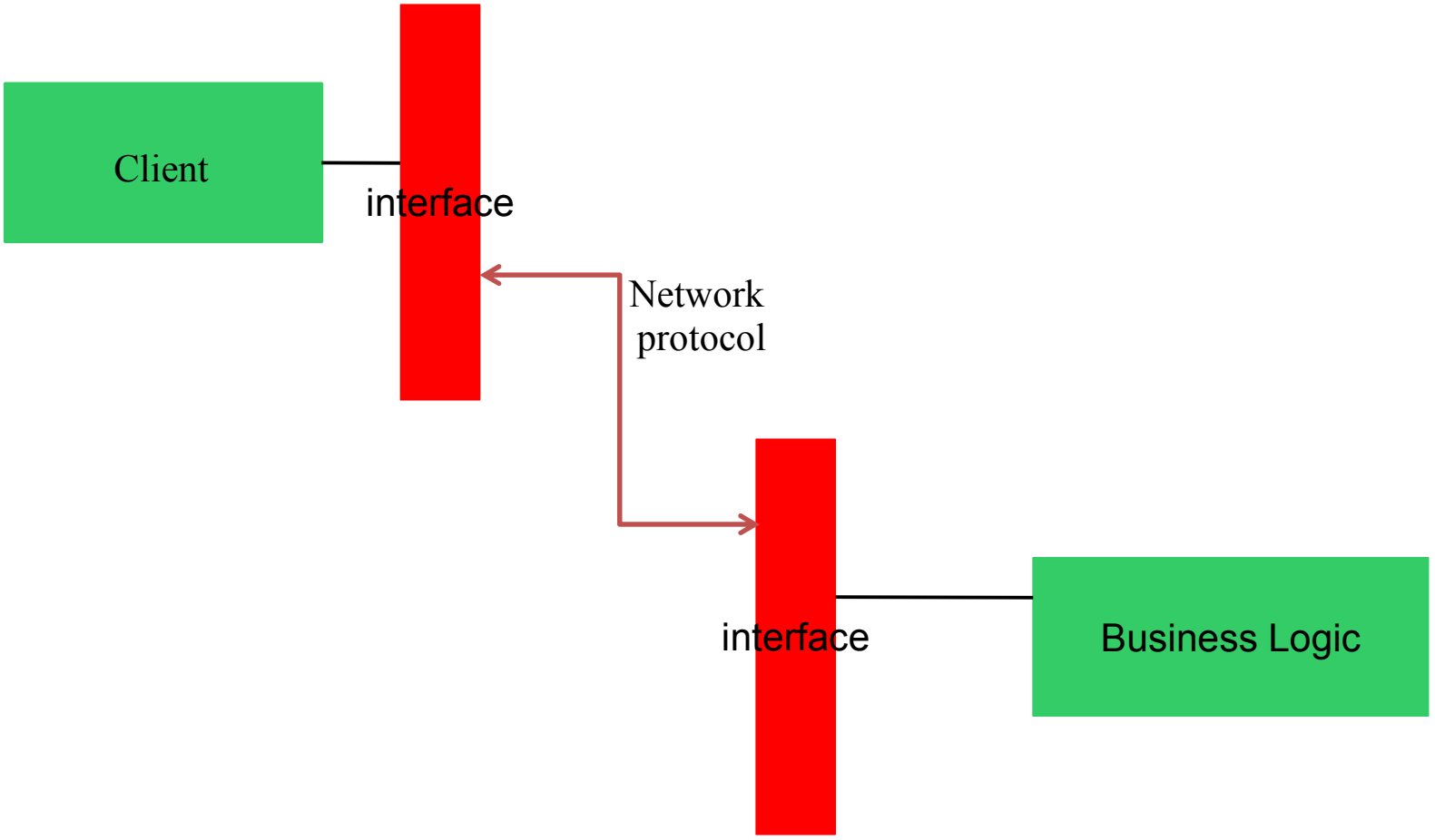

Objectives



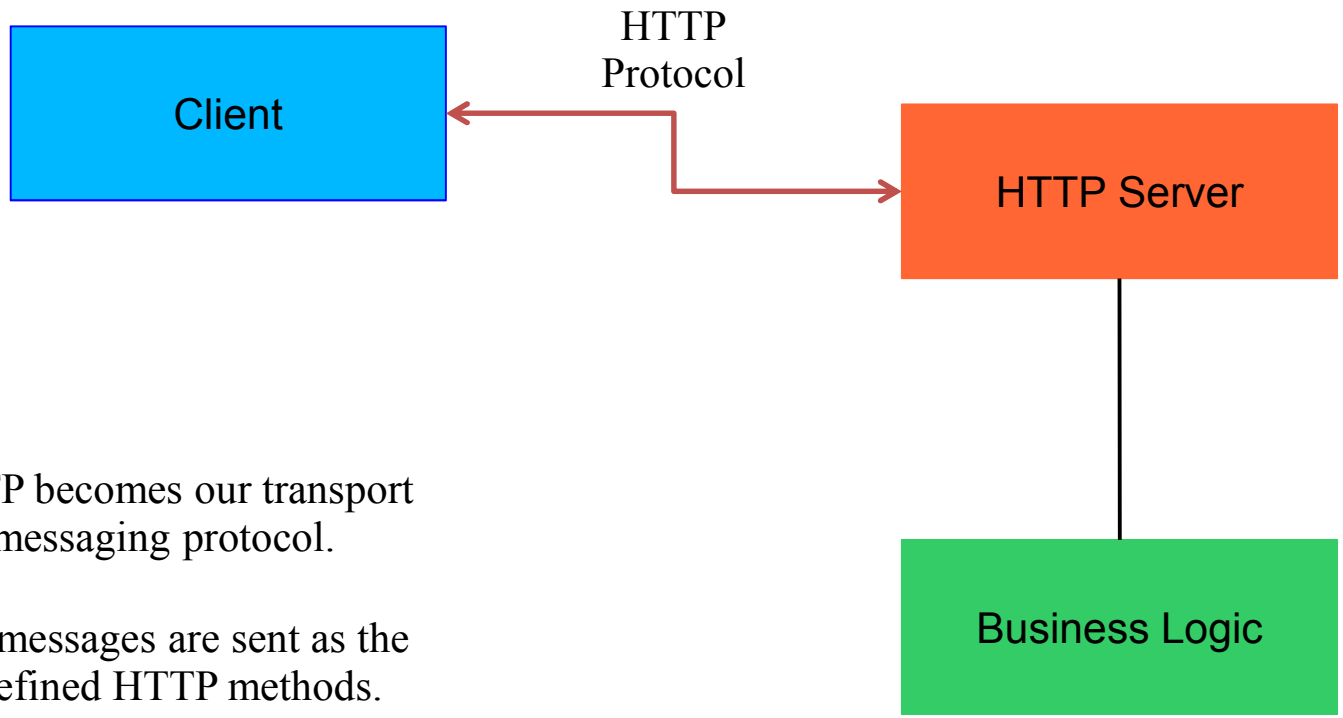


- Identify the benefit of REST
- Select appropriate HTTP methods for the implementation of particular behaviors in a REST interface
- Map simple entity relationship data models to REST URIs
- Select appropriate mechanisms for carrying data and request meta data between service and client

Traditional Network Communication



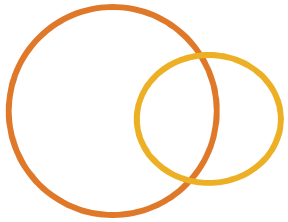
RESTful Network Communication



HTTP becomes our transport and messaging protocol.

The messages are sent as the predefined HTTP methods.

Why REST?

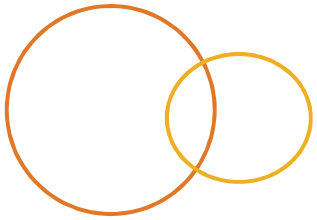


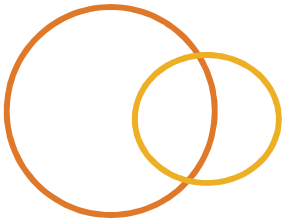
- Universal client
  - So called “low entry barrier”
  - All it takes is HTTP & JSON
- Other reasons are less clear in practice
  - Extensibility—but in practice, changes have a tendency to break clients
  - Uniform Interface—but this is hard to do, and developers often have to learn data structures and URIs
  - Scalability—from caching, but, data often cannot be properly cached due to concurrent server-side changes

REST Interactions with HTTP

- HTTP methods are used in a database-like mode
  - POST - Creates resource
  - GET - Reads resource
  - PUT - Updates resource
  - DELETE - Deletes resource
- And sometimes:
  - PATCH - Partial update of resource
- HTTP specification allows arbitrary “user-defined” methods, but this should be avoided

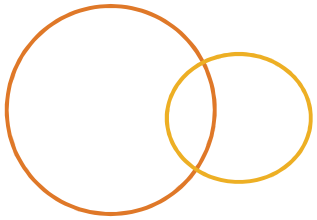
Example

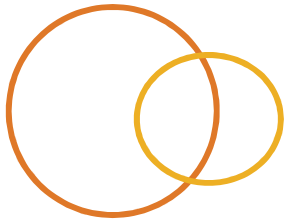




- System has Customer entities, with name and address. Address has street, city, state, zip
- Read all customers:
  - `GET /customers`
- Read customer with PK 1234
  - `GET /customers/1234`
- Get address of customer with PK 1234
  - `GET /customers/1234/address`
- Get zip of customer with PK 1234
  - `GET /customers/1234/address/zip`

Example



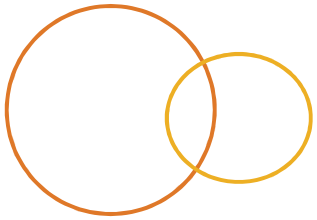


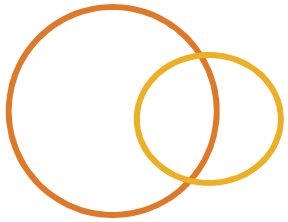
```
GET /customers/1234
Accept: application/json
```

- HTTP response entity:

```
{ "name" :  "FloorMart",
  "address" : {
    "street" : "123 Acacia Gdns",
    "city" : "Rainbowville",
    "state" : "OZ",
    "zip" : "00000"
  }
}
```

Example

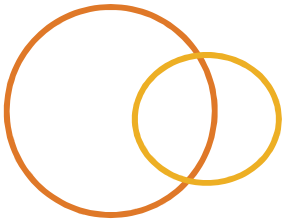




- Create a new customer

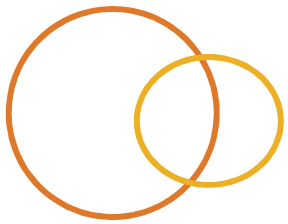
```
POST /customers
Content-type: application/json

{ “name” :  “GarageDepot”,
  “address” : {
    “street” : “1 Storage Lane”,
    “city” : “Exhausttown”,
    “state” : “OZ”,
    “zip” : “00000”
  }
}
```

Hierarchical Structure



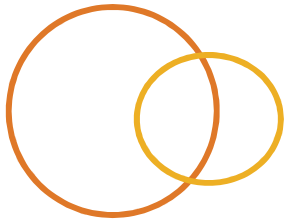
- URI structure, left to right qualifies scope
- A whole customer
  - /customers/1234
- Find a customer, then select the name
  - /customers/1234/name
- Find the customer, get the list of suppliers, get the third item in the list, then get the name
  - /customers/1234/suppliers/2/name ←

Handling Actions



- CRUD operations are easy to understand, but REST services aren't just for database access
  - How can we trigger behavior? E.g. pay this invoice, print this document

Action Jobs



`POST /print-requests`

[[Entity body is document to print]]

- Response: `Location: 1234`

- Allows, optionally, job control features:

`GET /print-requests/1234`

[[Might reports status of job]]

`DELETE /print-requests/1234`

[[Cancel the print job]]

Options For Data Transfer

- Client sends:
  - HTTP method
  - URI
  - Query parameters
  - Headers
  - Entity body
- Server returns:
  - Status code
  - Headers
  - Entity body

Data / Metadata From Client

- URI may contain “parameters”, such as primary key or index into a list
  - Use to qualify what is being addressed
- Query parameters are suitable for non-hierarchical “qualification”
  - E.g. query language expression

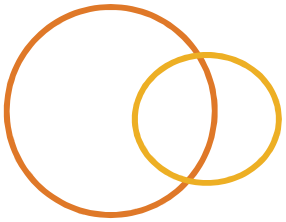
Data / Metadata From Client

- Headers should not change the essential nature of the request, but are suited for language selection, security credentials, and other metadata
  - Used extensively by HTTP standard for such things as specifying/requesting entity and character encoding
- Entity body carries representation, possibly partial, of the target resource
  - Note that GET requests do not carry entity bodies

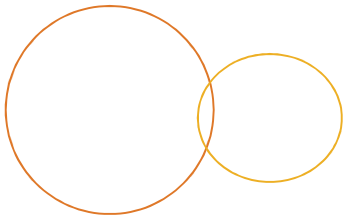
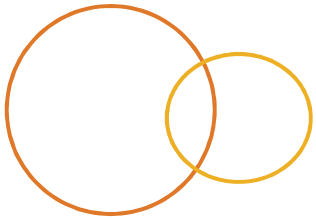
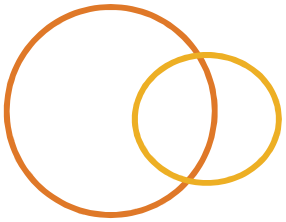
Data / Metadata From Server

- Status code indicates success, failure, deferment, etc. of request
  - Might carry some information about nature of failure
- Headers carry similar metadata as in client request: charset / entity body encoding, date/time, and other “metadata”
- Entity body carries representation, possibly partial, of the result

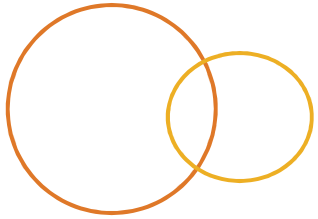
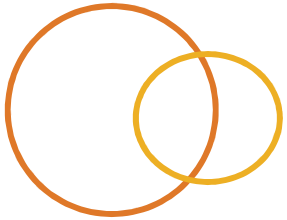
Lab Exercise



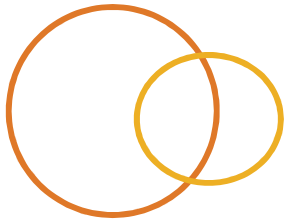
- Identify a service you are familiar with (theoretical or real)
- Identify a domain entity it represents
- Describe three levels (/x, /x/y, /x/y/z) of URI that the service could use
- Identify one “operation” the service might provide and alternative APIs for that operation
- Present questions, difficulties, and conclusions to the class


Getting Started With JAX-RS

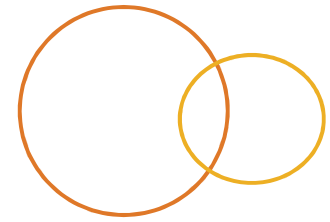



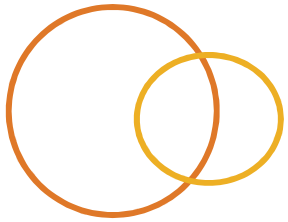

Objectives



- Build the JAX-RS Java EE maven archetype project
- Install and run the project in your preferred IDE
- Use the @Path annotation to define a “Root Resource” in a JAX-RS project
- Use @Path with a literal value, and @GET to cause JAX-RS to invoke a programmer-provided service method in response to an appropriate HTTP request

Objectives





- Describe the meaning of the term “Provider” in JAX-RS
- State two annotations that can be used to make a class a JAX-RS Provider
- Use the @ApplicationPath annotation to modify the base URI to which a JAX-RS application responds
- Give an overview of two alternative methods by which JAX-RS can be configured, including how to find specific details for these methods in both Jersey and RESTeasy implementations of JAX-RS

JAX-RS Root Resource

- JAX-RS service starts by matching a “Root Resource”
  - Class annotated with @Path(“/some-path”)

```
@Path(“/customers”)
public class CustomersRootResource {
```

  - Can respond to requests with URIs starting with /customers
  - This is the first glimpse of JAX-RS “routing”, which selects business logic code to service a request

Java EE Context Root

- Java web containers (e.g. Tomcat) use the first part of the URL to identify a particular web application.

```
http://myhost.myco.com:8080/my-service/customers
```

- First part leads request to the web container
- Second part leads to a particular web application
- Third part forms the “root” of the URI seen by our web application in JAX-RS

JAX-RS Service Method

- To handle a request (or “route” the request)
  - Path must be matched
  - HTTP method must be matched
  - (Other specified criteria might need to be matched)

```
@Path(“/customers”)
public class CustomersRootResource {
  @GET // matches HTTP GET method
  public String getCustomers() {
    // Matches GET /customers
    // Returns “all customers”
```

JAX-RS Service Method

- Routing can happen on the method
  - This matches /customers/1234 in two steps

```
@Path(“/customers”)
public class CustomersRootResource {
  @Path(“/1234”)
  @GET // matches HTTP GET method
  public String getCustomer1234() {
    // Matches GET /customers/1234
    // Returns customer 1234
```

JAX-RS Configuration Options

- JAX-RS is a standard, implementations are available from multiple sources
- Plan to read the product documentation!
  - Jersey
    - The reference implementation
    - http://jersey.java.net/
  - REST Easy
    - Jboss project
    - http://resteasy.jboss.org/

General JAX-RS Configuration

- JAX-RS implementations offer many configuration options
- Container or not?
  - Run your application in a Java EE web container (such as Tomcat)
  - Run your application stand alone using “Grizzly”

From Request to Service Code

- Recall that for Java EE web-apps, the base URL must reach our web-container
- Next, the URL must match the “context root”

```
http://myhost.myco.com:8080/my-service/customers
```

- First part leads request to the web container
- Second part leads to a particular web application
- Third part forms the “root” of the URI seen by our web application in JAX-RS (but not necessarily the root resources)

From Request to Service Code

- Java web-apps are built from servlets, and/or servlet filters
- Once inside our web-app, routing can follow instructions in web.xml, or use annotation-based configuration, to find the JAX-RS implementation, which is usually a servlet, but might be a filter
- Product documentation will tell you what you need to know; web.xml is easier to see what's going on

From Request to Service Code

- When servlets or filters are deployed, it's possible to define the sub-path that "triggers" them

```
…blah.com:8080/my-service/servlet-prefix/customers
```

- First part leads request to the web container
- Second part leads to a particular web application
- Third part is the mapping of the servlet / filter
  - This might be multiple-levels
- Fourth part leads to a root resource

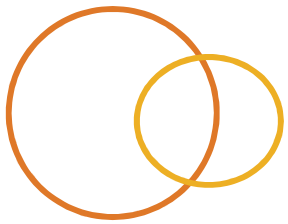
From Request to Service Code

- Once in the JAX-RS implementation, JAX-RS must be able to find our root resources and other elements we wrote
- Classes, provided by us, that must be found directly by JAX-RS, are called providers. E.g.:
  - Root resources, annotated with @Path on the class
  - Generalized error handlers.
  - Utilities for converting between Java objects and wire formats such as JSON or XML
  - Unless there's a specific annotation (such as @Path) these are annotated @Provder

From Request to Service Code

- Providers can be located by JAX-RS in three common ways
  - Explicit listing in the return of a specific method in a specific class
  - Explicit listing in an <init-param> in web.xml
  - Package scanning in packages located in the web-app, based on packages listed in an <init-param> in web.xml
- The first approach is standard, the other two are generally more convenient, but are implementation dependent
  - Further implementation specific approaches may exist

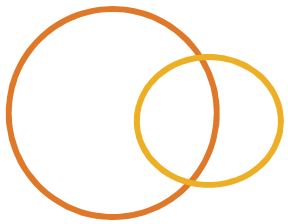
The Application Class



- The one standard method of locating classes is to provide a subclass of `javax.ws.rs.core.Application`
- The class is annotated @ApplicationPath()
- The class might be declared in an <init-param> for the JAX-RS implementation servlet

```
<init-param>
  <param-name>javax.ws.rs.Application</param-name>
  <param-value>org.foo.MyApplication</param-value>
</init-param>
```

The Application Class

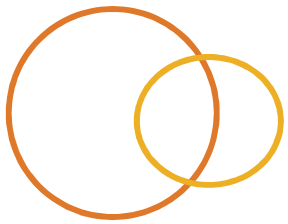


- When used, the Application class defines two methods

```
Set<Class<?>> getClasses()
Set<Object> getSingletons()
```

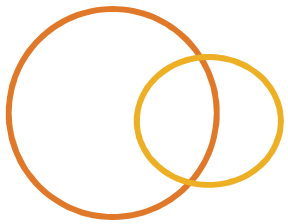
- The classes, and instances, returned by these will be used in the application

The ApplicationPath



- The Application classes, when used, is annotated @ApplicationPath
- This annotation takes an argument which might modify the “base” URI from which service is offered
  - Exact rules are quite complex, and depend on how the deployment is configured; refer to documentation, or just experiment

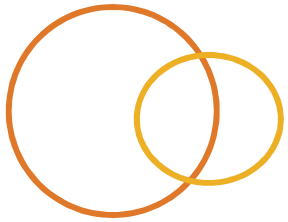
The Application Path



….com/my-svc/map/application-path/customers

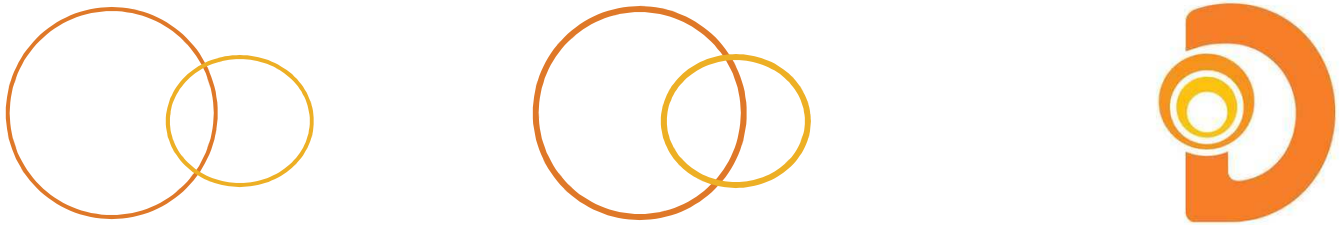
- First part leads request to the web container
- Second part leads to a particular web application
- Third part is the mapping of the servlet / filter
- Fourth part is the application path
- Fifth part leads to a root resource

Lab Exercise

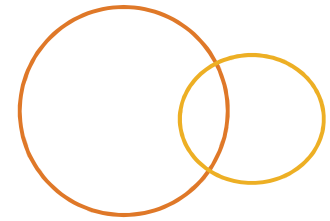


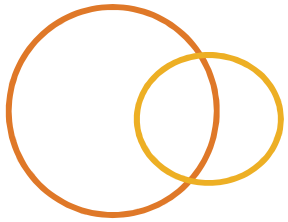
- Build, run, and exercise the initial project according to the instructions on the Jersey “Getting Started” page
- Create a new root resource in your application
- The sub-path to your resource (below all the initial parts) should be /fruits
- Respond to a GET request on this URI with the String “Bananas Apples and Oranges”

Core JAX-RS Features



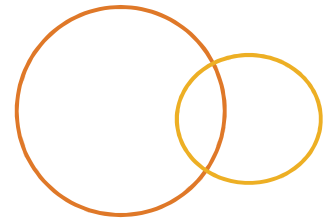
Objectives

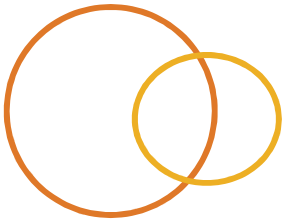




- Write a JAX-RS service method that uses an @Path annotation to define a JAX-RS variable and injects the String value of that variable into a method using @PathParam
- Write a JAX-RS service method that receives a query parameter using the annotation @QueryParam
- Write a JAX-RS service method that receives a header information using the annotation @HeaderParam
- Use the annotation @DefaultValue to inject non-null values for query parameters and/or headers that are absent from the request

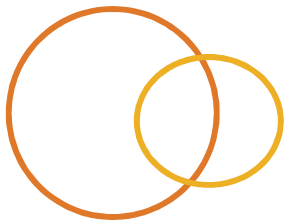
Objectives





- State the requirements for a Java data type to be convertible by JAX-RS prior to injection using @PathParam, @QueryParam, @HeaderParam
- Write a JAX-RS service method that responds to a DELETE request
- Use a Java Regular Expression in an @Path annotation to restrict the URIs that JAX-RS will route to a given service method
- Give an overview of the mechanism by which JAX-RS matches URI and HTTP method of an incoming request to annotations on service classes and methods to select the service method to invoke

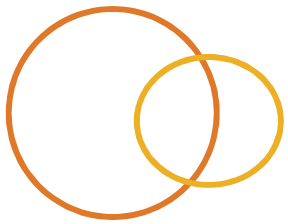
Path Parameters



- REST often selects an item by a primary key embedded in the URI
  - GET /customers/1234
- Part of the URI must be treated as a variable, even though that part still participates in routing
- JAX-RS defines and injects these like this:

```
@GET
@Path("/{id}")
public String getOneCustomer(
      @PathParam("id") String customerPK) {
      …
```

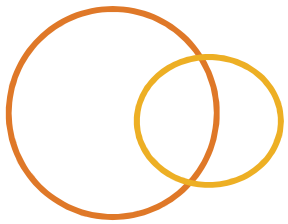
Query Parameters



- Query parameters can be injected into service methods too
- To pick up …`/stuff?name=Fred+Jones`

```
@Path("/stuff")
@GET
public String getStuff(
      @QueryParam("name") String name) {
      …
```

Header Injection



- Headers in the request can be injected too
- To read the value of the header `tid`

```
@Path(“/stuff”)
@GET
public String getStuff(
      @HeaderParam(“tid”)
      String transactionId) {
      …
```

More About Parameter Injection

- JAX-RS can inject more data too, including cookies and form data
- If a parameter is missing, null is injected
- If desired, missing parameters can be set to a specific String value by default

```
public String doStuff(
      @QueryParam("name")
      @DefaultValue("John Doe")
      String name) {
```

Injecting Non-String Types

- Injecting a String allows control of the conversion and recognition of missing data
- Alternatively some types can be converted and injected directly by JAX-RS

```
doX(@QueryParam("count") Integer count) {
```

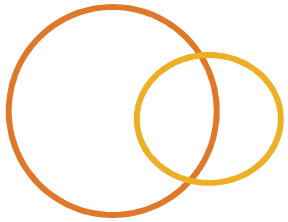
…

- Rules govern which types can be injected

Injection / Conversion Rules

- In addition to String, a type X can be converted and injected if:
  - It's a primitive
  - It's a wrapper of a primitive
  - If X contains:
    - A constructor taking a single String argument
    - static X fromString(String s)
    - static X valueOf(String s)
  - It's a List<Y>, Set<Y>, or SortedSet<Y> that would be acceptable individually

Other HTTP Methods



- JAX-RS provides several annotations that can be applied to a method, identifying it as a potential target for that HTTP method
- @GET
- @POST
- @PUT
- @DELETE
- @OPTIONS
- @HEAD
- A service method requires ***exactly one*** of these

Restricting @Path Variable Matching

- The annotation @Path("/{id}") matches any one segment
  - Any number of characters, but not including "/" itself
- JAX-RS allows restricting the match using standard Java regular expressions
- E.g. to match only digits:

@Path("/{id: \\d+")

- If the regular expression doesn't match, JAX-RS looks elsewhere for a potential service method

Overlapping @Path Specifications

- Regular expressions might “overlap” literal ones
- Two regular expressions might both match the same input string

```
@Path(“/banana”)
@Path(“/{f: [ban]+”)

@Path(“/{f: \\p{Lower}+}”)
@Path(“/{f: [ban]+”)
```

Overlapping @Path Specifications

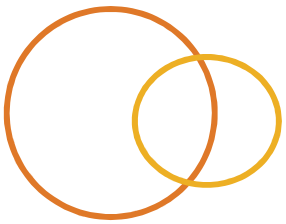
- JAX-RS 2.x always prefers the literal match
- Conflicting literals should fail to deploy
  - Or at least issue warnings
- Selecting between two overlapping regular expressions is unpredictable
  - They might overlap only for some input data
- All these conflicts probably reflect poor API design
  - They're potentially very confusing for client programmer

More Complex @Path Matches

- An @Path annotation can contain multiple variables, with regular expressions
- An @Path annotation can contain multiple segments
- Regular expressions can span segments

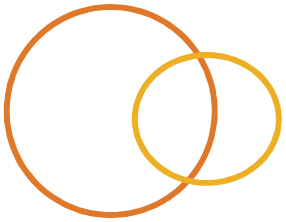
```
@Path("/customers/{id: \\d\\d-\\d\\d\\d}"
    + "/suppliers/{idx: \\d+}")

@Path("/documents/{path: [a-zA-Z0-9/]+}")
```

JAX-RS Routing



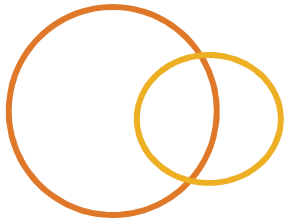
- Given an input request URI and HTTP method, JAX-RS selects which method to invoke by:
  - Looking for a class annotated @Path("/blah") where /blah (possibly multi-segment) matches the beginning of the URI
    - This might produce more than one class, but it's probably bad design if it does
  - Looking for methods that match the remaining path, and have the right HTTP method annotation (@GET, etc.)
  - If one is a literal match on @Path use that
  - If there's no literal match but multiple regex matches, use the one that has the most literal match points

JAX-RS Routing



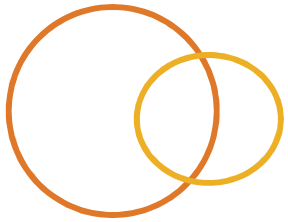
- Other factors will affect this later, including the type of data being sent, or requested
  - HTTP headers “Content-type” and “Accept”
- We’ll see these later
- Note that query parameters do ***not*** affect routing
  - Create a single method, and test if the query parameter is injected as null

Lab Exercise



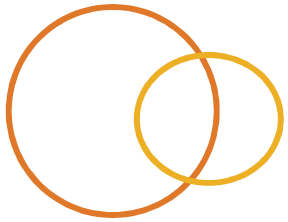
- Create a service endpoint that responds to a GET request on a URI of this form:
- …/fruits?color=red
- and returns the name fruit of the color specified
- Modify the service so that if the color is specified as “any”, one fruit is chosen at random.
- Arrange that if no color query parameter is specified the services responds as if “any” were specified

Lab Exercise



- Create a service endpoint that responds to a GET request on a URI of this form:
- …/fruits/3
- where the “3” can be any single digit. Return the name of a fruit taken from a small array (ten or fewer fruits)
- Arrange that this service checks the header “accept-language”. If this has the value “fr” report the fruit name in French, otherwise in English

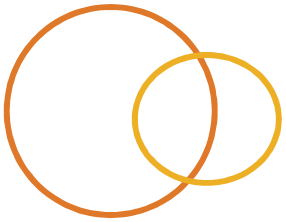
Lab Exercise



- Create a service endpoint that responds to a DELETE request on a URI of this form:
- …/fruits/3
- where the “3” can be a sequence of digits.

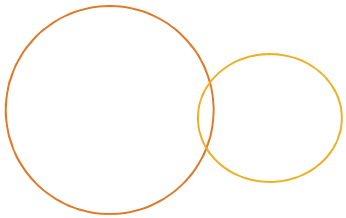
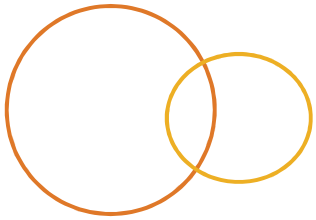
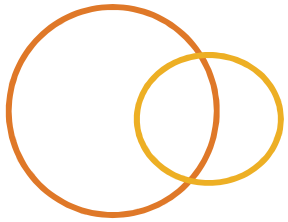
Return the message “Deleting fruit: xxx” where xxx is the name of the fruit with the index specified by the digits in the URI

Lab Exercise

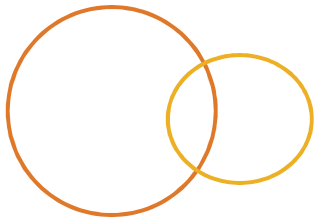
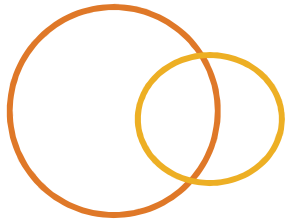


- You might find this helpful (define your own compatible Fruit class):

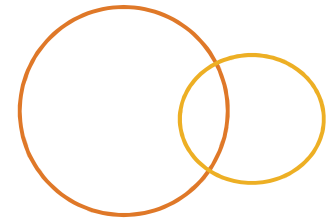
```
Fruit [] fruits = {
  new Fruit("raspberry", "framboise", "red"),
  new Fruit("orange", "orange", "orange"),
  new Fruit("lemon", "citron", "yellow"),
  new Fruit("lime", "citron vert", "green"),
  new Fruit("blueberry", "myrtille", "blue"),
  new Fruit("blackberry", "mûre", "black"),
};
```

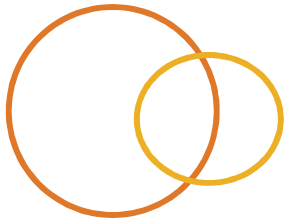





More Injection Features of JAX-RS

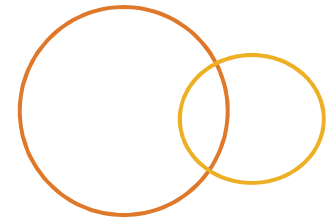

Objectives

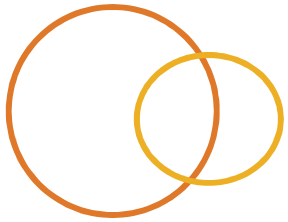




- State the per-invocation lifecycle model of JAX-RS service objects
- Use @Singleton to create singleton service objects
- State the restriction on parameter injection that applies to singleton service objects
- Inject parameters directly into instance variables for non-singleton service objects
- Inject parameters into object constructors for non-singleton service objects
- Use @Context to inject UriInfo and/or HttpHeaders objects

Objectives





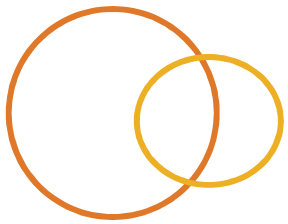
- Extract all the headers of a request from an HttpHeaders object into a MultivalueMap.
- Extract single values from a MultivalueMap
- Extract all query parameters from a UriInfo object

JAX-RS Default Object Lifecycle

- By default, every incoming request gets a newly created instance of the root resource class
- This allows the use of instance variables, rather than method parameters, as targets for injection either directly, or via a constructor

```
public class Resource {
  @HeaderParam("tid") private String tid;
  public Resource(@QueryParam("name") String name) {
```

Member Injection



- Injecting HTTP parameters into the service object, rather than the method parameters, can simplify handling large numbers of parameters, particularly if consistency rules or other code needs to be applied in all cases

Modifying The Lifecycle

- If the per-invocation object lifecycle is not needed, it can be disabled
  - Annotate the service class @Singleton (JAX-RS 2.x)
  - If using the Application class approach to configuration, return an instance of the class from the method

```
public Set<Object> getSingletons()
```

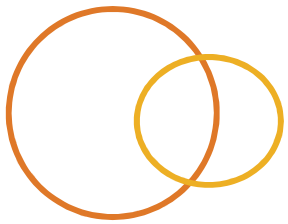
  - And do not return the class in the method

```
public Set<Class<?>> getClasses()
```

Injection In Singletons

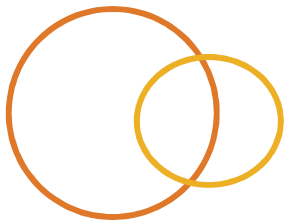
- If a root resource is configured as a singleton, it cannot have injection into either the constructor, or member variables
  - This would be nonsensical, since the object is shared between multiple, potentially concurrent, requests, but the injected parameter data must be per-request

Using @Context



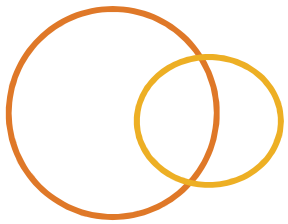
- The @Context annotation can inject (in any of the ways seen so far) some additional data, beyond the various XxxParam types seen so far
- Two types that can be injected with this annotation are generally interesting:
  - UriInfo—provides information about the request URI, such as absolute URI, path segments, and all query params (even ones you can't name in advance)
  - HttpHeaders—gives access to ***all*** the headers (even ones you can't name in advance)

Using HttpHeaders



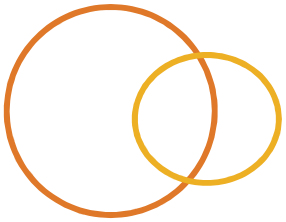
```
@Path("/headers") @GET
public String getInfo(
  @Context HttpHeaders headers) {
  StringBuilder sb = new StringBuilder();
  sb.append("x-my-header: ")
    .append(headers.getHeaderString("x-my-header"))
    .append('\n');
  return sb.toString();
}
```

Using HttpHeaders



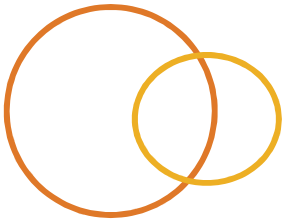
```
@Path("/headers") @GET
public String getInfo(
  @Context HttpHeaders headers) {
  StringBuilder sb = new StringBuilder();
  MultivaluedMap<String, String> hd =
    headers.getRequestHeaders();
  hd.forEach((k, lv)-> {
    sb.append("key: ").append(k).append('\n');
    lv.stream().forEach(v->{
      sb.append("  ").append(v).append('\n');
    });
  });
  return sb.toString();
}
```

Using UriInfo



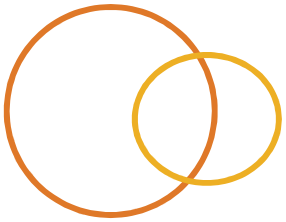
```
@Path("/uri") @GET
public String getUriInfo(
  @Context UriInfo uriInfo) {
  StringBuilder sb = new StringBuilder();
  sb.append("Full URI: ")
    .append(uriInfo.getAbsolutePath().toString())
    .append('\n');
  List<PathSegment> lps = uriInfo.getPathSegments();
  lps.forEach(ps->
    sb.append(ps.getPath()).append('\n')
  );
  return sb.toString();
}
```

Using UriInfo

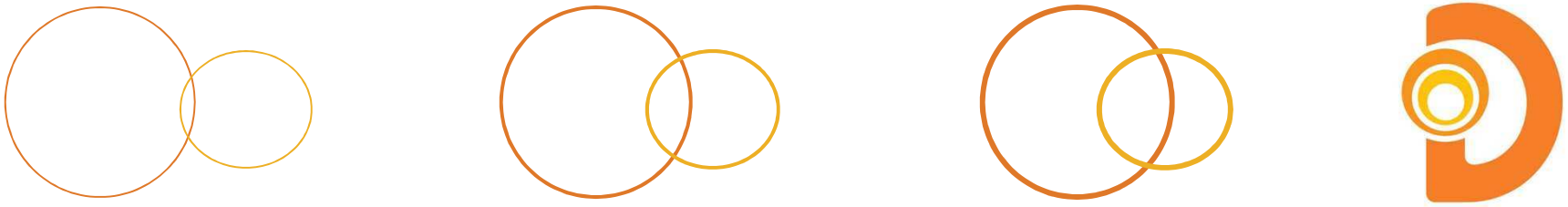


```
@Path("/uri") @GET
public String getUriInfo(
  @Context UriInfo uriInfo) {
  StringBuilder sb = new StringBuilder();
  MultivaluedMap<String, String> mmqp =
    uriInfo.getQueryParameters();
  mmqp.forEach((k, lv)-> {
    sb.append("key: ").append(k).append('\n');
    lv.stream().forEach(v->{
      sb.append("  ").append(v).append('\n');
    });
  });
  return sb.toString();
}
```

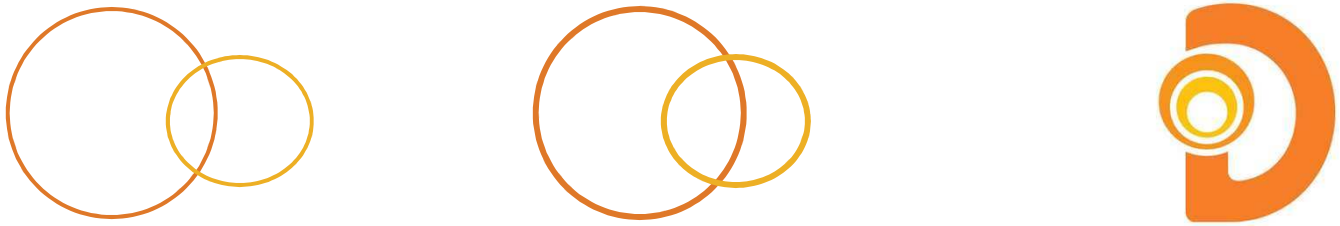
Lab Exercise



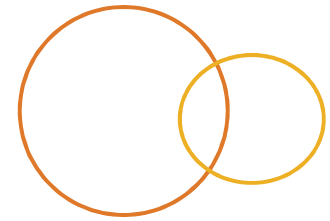
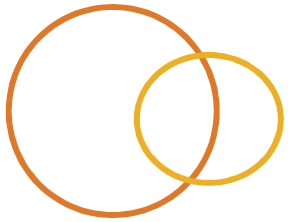
- Create a service endpoint that responds to any URI under /info (e.g. /info, /info/more/stuff etc.)
- The service should generate a text response that reports
  - All header keys and values
  - All query parameters, keys and values, sent with the request
  - The full URL on which the request was made



Controlling the Response



Objectives


- Obtain a ResponseBuilder initialized to an OK status
- Use a ResponseBuilder to configure/reconfigure headers, entity, status code, and content-type
- Prepare a Response object from a ResponseBuilder
- Return a Response object from a JAX-RS service method

What's In A Response?

- HTTP requests receive three things in response
  - Status code
  - Headers
  - Entity body
- A service method that simply returns a Java object has no control over the status code, nor headers
  - JAX-RS will fill in some headers, and return a success code if the request was routed successfully and the method did not throw an exception.

Controlling HTTP Response

- javax.ws.rs.core.Response object aggregates HTTP status code, headers, and entity
- Response has a static factory that creates a builder
- Response.ResponseBuilder allows manipulation of all three elements, then creation of final Response object
  - Initial creation of builder requires a guess at the expected status code, but this code can be changed later

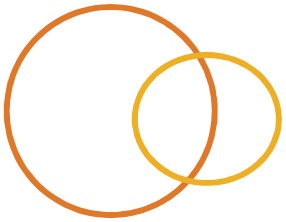
Using Response / ResponseBuilder

```
public Response doStuff() {
  ResponseBuilder rb = Response.ok();
  rb.header("x-header", "a header value");
  String entity = getEntity();
  if (entity != null) {
    rb.entity(entity);
  } else {
    rb.entity("Sorry, that didn't work");
    rb.status(Response.Status.NOT_FOUND);
  }
  return rb.build();
}
```

Response And Content-Type

- By default JAX-RS will attempt to set the content type of the response by negotiation with the caller
- The Response object can override this using the type() method of the ResponseBuilder
  - Do this with caution, JAX-RS usually makes the right choice
- JAX-RS also converts to the target type, though we have not looked at this yet
  - i.e. Java objects ⟷ JSON or XML

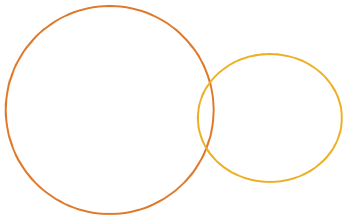
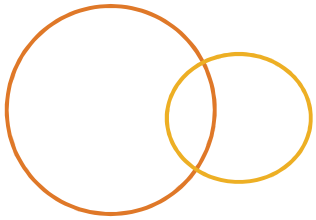
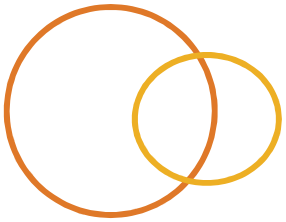
Lab Exercise



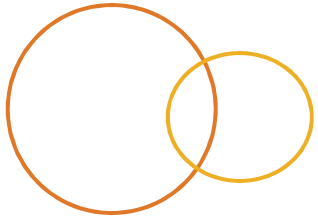
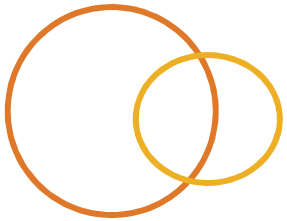
- Create a service endpoint on the URI /response/<id> where <id> is any number
  - If the value of <id> is between 0 and 100, return a text message “Item <id> found”, along with a status code of 200, and a header “x-range” set to the value of “ok”
  - If the value of <id> is outside the limits, return status 404 and the html message

```
<html><body><h1>ID Not Found</h1></body></html>
```

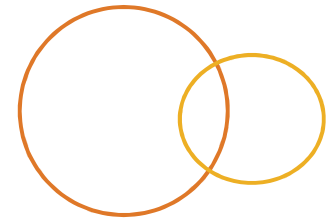
    - Be sure to set the content type.

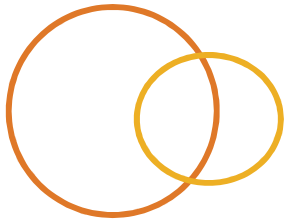





Handling HTTP entities

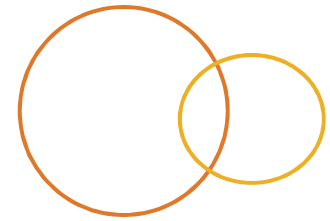

Objectives

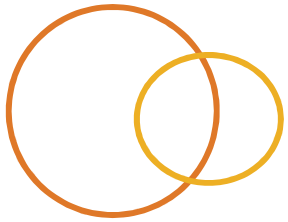




- Give an overview of the role of MessageBodyReader and MessageBodyWriter classes in conversion of HTTP entity bodies to and from Java objects.
- Give an overview of the installation of MessageBodyReader/Writer Provider objects into a JAX-RS system to enable conversion of a particular MIME data type to and from Java objects
- Send data structures to a client using JSON format
- Receive JSON data structures from a client
- Use the @Produces annotation to inform the JAX-RS dispatch/routing infrastructure that a particular service method is appropriate for returning a given MIME data type to the caller

Objectives





- Give an overview of how JAX-RS can return binary data to a client using either an InputStream, a byte [], or a StreamingOutput
- State the limitation of JAX-RS 2.x with respect to handling multipart/form-data
- State the significance of charset for web-based operations involving text and textual representations of structured data
- Locate documentation for the Jackson conversion libraries

REST And Structured Entities

- REST services often send and receive entities representing structured data in textual forms
- Often done using XML or JSON
  - JSON seems to be much more popular

JAX-RS And Structured Entities

- JAX-RS expects to convert Java objects to and from text-based wire formats
  - Generally, no manual intervention is needed
- It does this using special classes annotated @Provider implementing MessageBodyReader and MessageBodyWriter interfaces
  - These can be programmer defined (but not often)
  - Reader/Writer for XML are required as part of the JAX-RS specification
  - JSON converters are readily available, distributed with some implementations, but might need to be enabled

Supplying The Provider

- The Provider class(es) that implement MessageBodyReader/Writer must be known to the JAX-RS infrastructure
  - Check the documentation for your particular implementation
  - Package scanning
  - Explicit classes in web.xml
  - Pure annotations in Servlets 3.x containers
  - Set<Class<?>> getClasses() in Application
  - Set<Object> getSingletons() in Application

The “Jackson” Provider

- The “Jackson” JSON support provider is preconfigured in some implementations of JAX-RS
- More information on Jackson may be found at:

```
https://github.com/FasterXML/jackson

https://github.com/FasterXML/jackson-docs
```

- Older Jackson releases were at codehaus.org

Giving Permission For Conversions

- For a response entity to be converted to a target type (such as JSON):
  - The client should have indicated that it can handle the target
    - This is done by the client sending the Accept: header, though if absent, this implies Accept: */*
  - The service method should be annotated to indicate that the programmer permits this conversion
    - E.g. @Produces({MediaType.APPLICATION_JSON})
  - JAX-RS must have a suitable Provider that can convert from the Java class of the entity provided by the service method, and the desired target type

Giving Permission For Conversions

- For an entity from the client to be converted to a target type:
  - The client ***must*** have indicated the target type
    - This is done by the client sending the Content-type:
  - The service method should be annotated to indicate that the programmer permits this conversion
    - E.g. @Consumes("application/json")
  - JAX-RS must have a suitable Provider that can convert from the Java class of the entity provided by the service method, and the desired target type

Sending A Structured Response

```
@GET @Path("/one")
@Produces({MediaType.APPLICATION_XML,
           MediaType.APPLICATION_JSON})
public Response getOne() {
  DataTO dto = new DataTO(99, “banana”);
  return Response.ok(dto).build();
}

public class DataTO {
  public Integer value; public String fruit;
  public DataTO(int v, String f) {
    value = v; fruit = f;
  }
}
```

Receiving Structured Data

```
@POST @Path("/one")
@Consumes({ MediaType.APPLICATION_XML,
            MediaType.APPLICATION_JSON})
public Response getOne(DataTO theData) {
  long primaryKey = DataTO.store(theData);
  return Response.ok("Success")
    .status(Response.Status.CREATED)
    .header("Location", "" + primaryKey);
    .build();
}
```

Helping / Hinting Converters

- Different conversion providers might require hints for making some conversions
  - XML conversion is built into JAX-RS, as a specification requirement, using JAX-B
  - JAX-B uses annotations to provide help/hints on conversion
  - Handling lists / arrays is not always transparent
  - Jackson also offers some annotations e.g. @JsonIgnore, @JsonProperty (see Jackson docs)
  - Handling missing fields might vary between converters

Helping / Hinting JAX-B

- Minimal JAX-B provisions:
  - Annotate the class @XmlRootElement
  - Provide a zero argument constructor
  - Provide public fields ***or*** public get/set method pairs (***not*** both!)

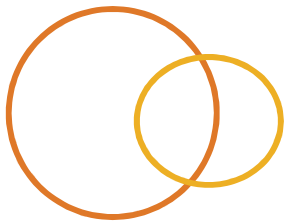
What's In A `null` Field?

- In a PUT (update) request, it's normal to treat a null in the input structure as "don't modify this element"
  - Sometimes, null means "set this to null"—this problem might be better addressed using PATCH
  - Omitting fields can reduce bandwidth usage
- Primitive fields in Java objects cannot be omitted, but wrappers (Integer, Double, etc.) can have null values

Handling Repeating Data

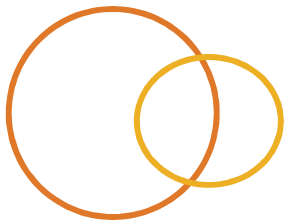
- Jackson handles T, List<T>, and T[] without help for JSON conversion for input and output
- JAX-B handles T[] for both input and output if it handles T
- JAX-B can also convert a sequence of T into List<T> for input
- JAX-B fails if given List<T> directly for output
  - Can you avoid List, and simply use an array?
  - Create a wrapper or “Transfer Object” that contains List<T> as a field (perhaps surprisingly, this works fine)
  - Could also use javax.ws.rs.core.GenericEntity

Transfer Objects



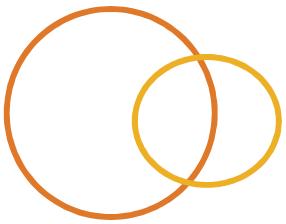
- Because of the differing needs of wire-transfer and business entity representation / validation, it's often a good idea to create specific transfer object or T.O.
  - Annotations about wire transfer are not placed on business entity objects
  - Validation is undesirable—this should be a responsibility of the domain entity
  - Fields can be public if desired
  - Provide utility methods in the T.O. class to create T.O. from domain entity, and vice-versa, and to modify a domain entity based on non-null T.O. fields

Other Data Types



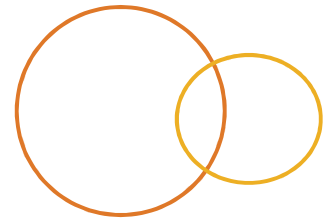
- JAX-RS supports additional entity body types including:
  - InputStream—the stream is read and fed to the client
  - byte[]—the contents of the byte array are sent directly to the client
  - StreamingOutput—return an implementation of this interface and JAX-RS will call the implemented method
  - void write(OutputStream output)
  - allowing you to use the OutputStream to write data

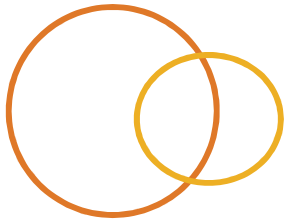
Multipart Form Data



- JAX-RS 2 (the standard) does not provide a mechanism for directly handling multipart form data input to service methods
  - Multipart can be useful for uploading arbitrary binary data, such as images, though it is mostly convenient when using an HTML based client
- Implementations, including Jersey, provide implementation-specific extensions for this
  - Of course, these make your code implementation specific

Charset



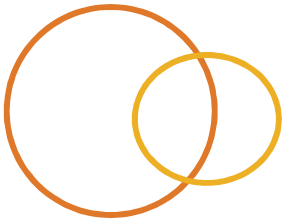


- To ensure that the client handles international language text consistently
  - Ensure you're working with UTF-8 in your server
  - Arrange that your service method declares

```
@Produces(MediaType.APPLICATION_JSON
            + ";charset=UTF-8")
```

  to inform the client what is being sent

Lab Exercise



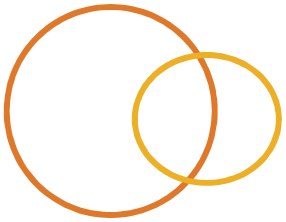
- Create a class to act as a Data Transfer Object
  - Give it fields:

    String name

    Integer count

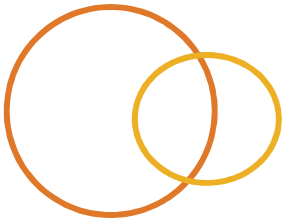
    String [] more
  - Give it a constructor to initialize the fields
  - Give it a toString method to allow readable display of the object

Lab Exercise

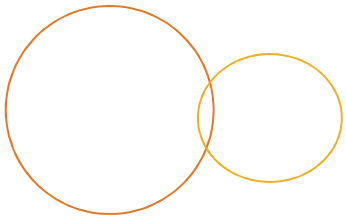
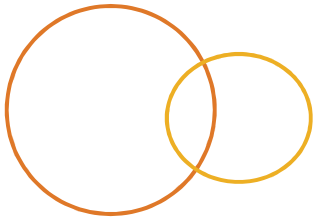
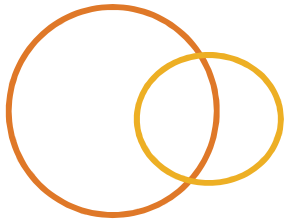


- Create a service endpoint on the URI /structure
  - Respond to a GET request by returning one of these structures, with a status of 200 and a header named “x-structure” with the value “yes”
  - Respond to a POST request that receives one of these structures:
    - Print out the contents received on the console output
    - Return a simple text form of the same object that was received

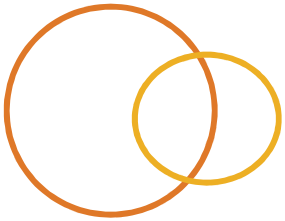
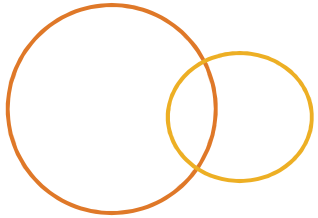
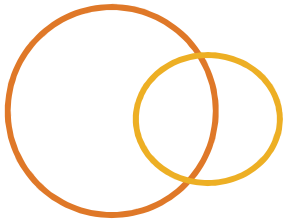
Lab Exercise



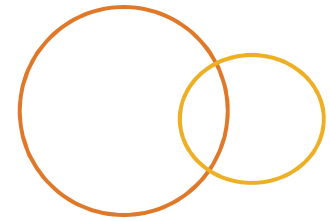
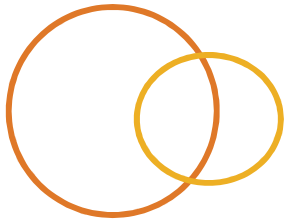
- Ensure that the service offers and accepts both JSON and XML formats for this structure
- Observe the effect of a null field when responding to a GET
- Observe the effect of a missing element from the JSON format


Error Handling


Objectives


- Apply standard Java language tools to identify/catch problems and use the Response object to send an appropriate entity body and HTTP status code to the client.
- Use WebApplicationException to control how a problem is converted to a response in a method that does not return a Response object
- Use an ExceptionMapper object to provide catch-all handling of a category of exceptions thrown by the application
- Use an ExceptionMapper object to take control of how problems arising in the JAX-RS navigation/service method selection are reported to the client

Three Approaches To Problems

- Report non successful status directly through Response object
  - Applicable to service methods, particularly for unique error situations
- Throw a WebApplicationException
  - Applicable to non-service methods for unique error situations
- Throw an application domain exception, handle with ExceptionMapper<DomainException>
  - Applicable to service and non-service methods for recurring error situations

Using Response To Report Status

- Use Java code to identify the error, use the ResponseBuilder status method to set the status representing the problem

```
public Response doStuff() {
  ResponseBuilder rb = Response.ok();
  // try something..
  if (unsuccessful) {
    rb.status(404).entity("Not Found!");
  }
  return rb.build();
```

Using Response to Report Status

```
@GET @Path("/vehicles/{id}")
public Response getVeh(@PathParam("id") int id) {
  ResponseBuilder rb = Response.ok();
  try {
    String vehicle = dbLookup(id);
    rb.entity(vehicle);
  } catch (SQLException sqle) {
    rb.status(Response.Status.NOT_FOUND);
    rb.entity("Broken!");
  }
  return rb.build();
}
```

Using Response To Report Status

- Easy to use
- Only workable when the method returns a Response
  - Can pass exceptions up call stack to the service method and then use this technique
- Tends to cause code duplication with errors that occur in many places
- Tends to clutter service code with unhappy path code
- Embeds “presentation” in service logic

Using WebApplicationException To Report Status

- javax.ws.rs.WebApplicationException is a RuntimeException
- If thrown from a service method, can specify aspects of the response to the caller

```
throw new WebApplicationException(
  Response.status(404)
    .entity("Not Found!")
    .build());
```

- The Response embedded in WebApplicationException is sent to caller

Using WebApplicationException To Report Status

- Easy to use
- Can be used from methods, e.g. “subroutines”, that don’t directly return Response

- Tends to cause code duplication with errors that occur in many places
- Tends to clutter service code with unhappy path code
- Embeds “presentation” in service logic

Using WebApplicationException To Report Status

```
String dbLookup(int id) {
  // perform lookup
  if (failed) {
    throw new WebApplicationException(
      Response
        .status(Response.Status.NOT_FOUND)
        .entity("WAE Not found!")
        .build()
    );
  }
}
```

- Stylistically, this example is bad because it clutters dbLookup with JAX-RS specifics

Using ExceptionMapper<T> To Report Status

- ExceptionMapper<E extends Throwable> may be embedded as a Provider class in JAX-RS system
- If a T is thrown from service code into the JAX-RS system, JAX-RS will look for an ExceptionMapper<T>
  - If found, JAX-RS passes the exception to the method
  - Response toResponse(T exception)
  - in that ExceptionMapper, and uses the Response to send to the caller

Using ExceptionMapper<T> To Report Status

- Mappers are type specific and generalized, just like catch blocks
- The most specific, applicable, mapper is used
- You can install many mappers, and based on the type they are declared to handle, they will be called, just like having many catch blocks

- ExceptionMappers will also be checked when the JAX-RS system has a problem
  - E.g. no method found to handle a given request
  - Allows standard error format, even for system problems

Using ExceptionMapper<T> To Report Status

```
public class WidgetException extends
  RuntimeException {
  // various standard constructors
}
```

```
private String findWidget() {
  throw new WidgetException("Widgets used up");
}

@GET @Path("/widgets")
public Response getWidget() {
  return Response
    .ok(findWidget())
    .build();
}
```

Using ExceptionMapper<T> To Report Status

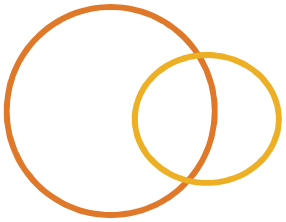
```
@Provider
public class WidgetExceptionMapper
  implements ExceptionMapper<WidgetException> {
  @Override
  public Response toResponse(WidgetException ex) {
    return Response
      .status(Response.Status.GONE)
      .entity("No widgets here! "
              + exception.getMessage())
      .build();
  }
}
```

Using ExceptionMapper<T> To Report Status

- Can be used from methods, e.g. “subroutines”, that don’t directly return Response
- Handles system errors
- Handles domain-specific errors
- Avoids duplication of error handling
- Separates “presentation” and service logic

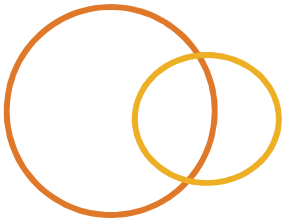
- More complex to configure

Lab Exercise



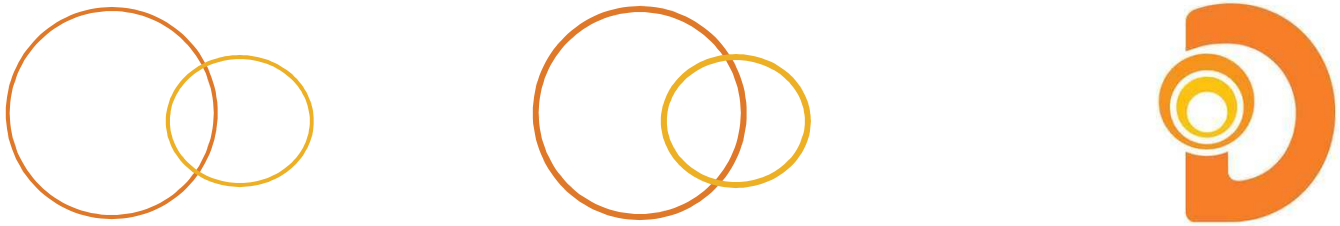
- Create a new service endpoint that responds to /problems/<id>
- The service method should be declared to return a Response object
- The service should respond to a GET request as follows:
  - If <id> is negative, use the Response to set the status to 404, with an textual entity body
  - If <id> is between 0 and 99, throw a WebApplicationException to report a status of 400, with an entity message of your choice

Lab Exercise

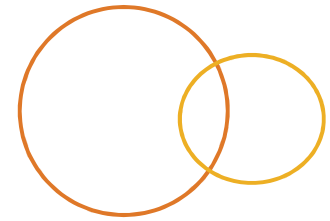


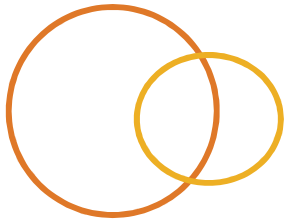
- Declare a new exception class MyException, as a subclass of RuntimeException
- Define. and install into your service, an ExceptionMapper<MyException> that returns a 500 error with an entity body of your choice
- Arrange that the service endpoint throws this exception if the value of <id> is 100 or more

Sub-resource navigation and Design Considerations



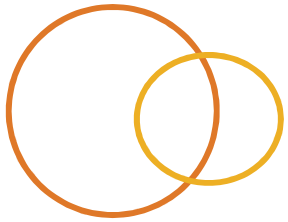
Objectives





- Create sub-resource classes that allow URIs to be processed in segments by JAX-RS, facilitating consistent and bi-directional mapping of object models to RESTful URIs.
- Create sub-resource classes that can be reused when navigating to a particular type of resource via many different URI paths.
- Use a design approach that separates the unrelated concerns of business entity modeling, wire transfer format of business entity representations, and mapping between business entity attributes and URIs used to access those features.

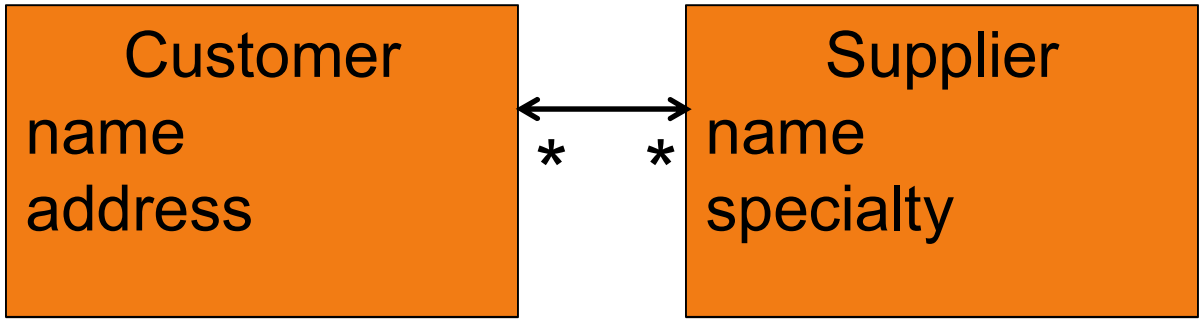
Objectives



- Give an overview of how this design approach supports flexibility in your code, minimizes consequences of change, and helps other programmers know where to look, and which source files to modify, when making changes and enhancements to the software.

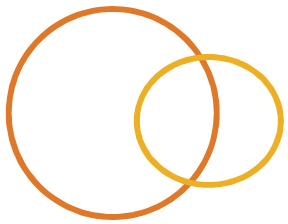
What's The Problem?

- Consider this entity relationship diagram



- These URIs are reasonable
  - `/customers/<pk>/suppliers/<ix>`
  - `/customers/<pk>/suppliers/<ix>/name`
  - `/customers/<pk>/suppliers/<ix>/specialty`
  - `/customers/<pk>/suppliers/<ix>/customers/<ix>`
  - `/customers/<pk>/suppliers/<ix>/customers/<ix>/name`

What's The Problem?



- Good separation of concerns requires that customer-parts of the URI and supplier-parts of the URI should be handled indepenently

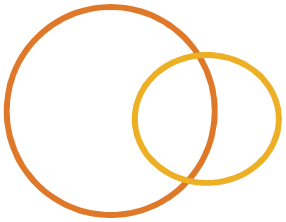
Addressing The Problem

- JAX-RS can handle a URI in multiple segments

`/customers/<pk>/suppliers/<ix>/customers/<ix>/name`

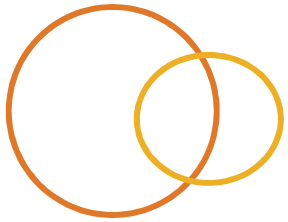
- Find a customer without a context (static-like)
- Find a specific supplier from the list of suppliers of the specific customer (customer-instance like)
- Find a specific customer from the list of customers of the specific supplier (supplier-instance-like)
- Find the name of the specific customer (customer-instance like)

Coding Multiple Steps



- JAX-RS deliberately allows this multi-step handling of a URI
- Search always starts at a root resource
  - Class annotated @Path
- Each step can continue to another class
  - The current method returns the object that will be used to handle the next step
  - There's no turning back! Once JAX-RS moves on to a new object, that object either completes the request, passes on to another, or fails. If it fails, the whole request fails (404 – NOT FOUND)

Coding Multiple Steps



- If a method carries an **HTTP method annotation**, it **must complete the URI**
- If a method carries @Path, but **not** an HTTP method annotation, it is used to step along the URI
  - The return from this method will be used to progress down the URI
- To succeed, this search **must** end with the path **exactly** matched, and at a method that declares the right HTTP method annotation, and compatible @Produces / @Consumes

Separation Of Concerns Part 1

- This approach raises several concerns that should be separated:
  - Representing URI elements that allow identifying a single customer from “all customers”
  - Representing URI elements that allow identifying / navigating aspects of a specific customer
  - Representing URI elements that allow identifying / navigating aspects of a specific supplier

Separation Of Concerns Part 1

- Note, if this separation is implemented, a new attribute added to a supplier would immediately be available from all URIs that navigate to any supplier, no matter what prefix URI lead to it
  - This provides for easier maintenance
  - Compare this approach with dotted ("fluent" / "train wreck") OO coding style:

```
Customers.findByPk(id)
   .getSupplierByIndex(sIdx)
   .getCustomerByIndex(cIdx)
   .getName()
```

Separation Of Concerns Part 2

- There are other concerns in the larger system that should (or might) be separated:
  - Representation of the business domain entities
    - Validation might be critical here
  - Representation of the business domain entities as they are transferred over the network (as JSON or XML)
    - Validation is rarely appropriate here
  - Possibly, persistence of the business domain entities
    - But, if use of the BDEs without persistence is impossible, then it's often sufficient to separate the storage ***mapping***, e.g. with JPA / Hibernate
  - And, of course, all the JAX-RS / REST navigation concerns already addressed

Implementing Separation Of Concerns

- These concerns suggest that for each entity we need:
  - Business domain entity
  - Data Access Object, or other persistence mechanism
  - Data Transfer Object for wire format
  - Root Resource
  - Instance REST Navigation Object
- Further, the business domain entity should not depend on any of these, possibly excepting persistence

Example: Customer Entity

```
public class Customer {
  private String name;
  private String address;

  public String getName() { return name; }
  public String getAddress() { return address; }

  public Customer(String name, String address) {
    this.name = name;
    this.address = address;
  }
}
```

Example: Customer Transfer Object

```
@XmlRootElement // for JAX-B
public class CustomerTO {
  public String name;
  public String address;
  public CustomerTO() { } // for JAX-B
  public CustomerTO(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public CustomerTO(Customer c) {
    this.name = c.getName();
    this.address = c.getAddress();
  }
}
```

Example: Customer DAO

```
public class CustomerDAO {
  private static Customer[] store = {
    new Customer("Fred", "Here"),
    new Customer("Jill", "Somewhere")
  };
  public static Customer getById(int id) {
    return store[id];
  }
}
```

This class is (obviously) fake. It solely illustrates the separation provided by a Data Access Object pattern

Example: Customer Root Resource

```
@Path("/customers")
public class CustomersRootResource {
   @Path("/{id}")
   public CustomerNav
      findOneCustomer(@PathParam("id") int id)
   {
      return new CustomerNav(CustomerDAO.getById(id));
   }
}
```

Note, ***no*** "@GET" or other HTTP method annotation

Example: Customer Navigation

```
public class CustomerNav {
  Customer self;
  public CustomerNav(Customer self) {
    this.self = self;
  }

  @Path("/name") @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String getName() { return self.getName(); }
```

Many accessor/mutator would exist in a real navigation class

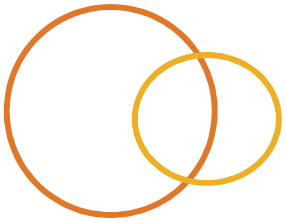
Note: There is ***no*** @Path annotation on the class. Instances of this are handed to JAX-RS by other classes, such as CustomersRootResource

Example: Customer Navigation

```
    @Path("/") @GET
    @Produces({MediaType.APPLICATION_JSON,
               MediaType.APPLICATION_XML})
    public Response getWholeCustomer() {
      return Response.ok(
        new CustomerTO(self)).build();
    }
}
```

An @Path(“/”) annotation means “this method adds nothing to the URI being traversed. Provided the method carries an HTTP method annotation, it will be invoked to complete processing where the URI has been completely traversed, but the method has not yet been matched

Lab Exercise

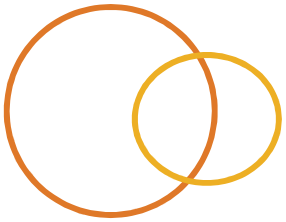


- Create a Transfer Object and Domain Entity for a Customer. The Customer should have a name, an account number, and an address. In the domain entity, these fields may not be null
- Create a simulated Customer Data Access Object, that carries an array of four customers
- The DAO should allow lookup of a single customer by primary key (the index into the array)

Lab Exercise

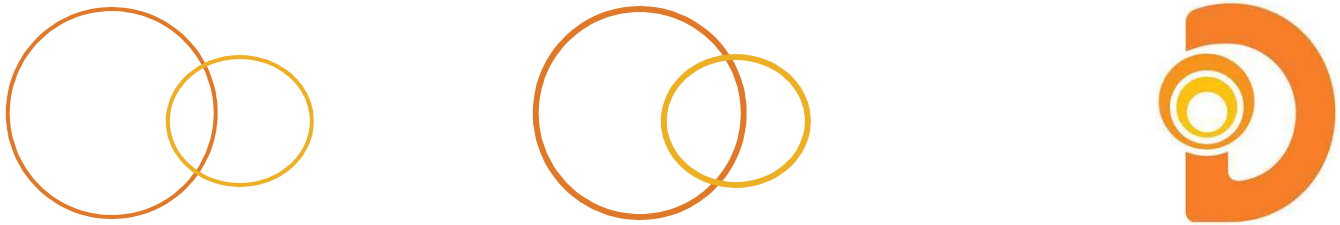
- Create a CustomersRootResource that responds to URIs of the form /customers/<pk> and returns a CustomerNav object to JAX-RS according to the design pattern described in this unit
- Arrange that the CustomerNav object supports operations as follows:
  - GET / — fetches a JSON or XML representation of the entire customer
  - GET /name — fetches the customer's name
  - GET /address — fetches the customer's address

Lab Exercise

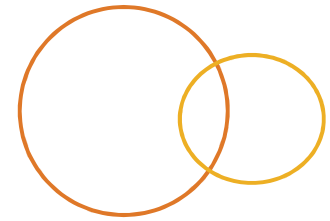


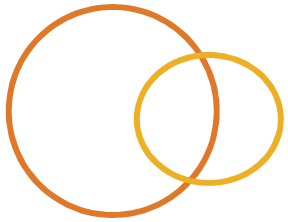
- Extra credit 1: Arrange that the CustomerNav supports the operation:
  - PUT / — modifies the name and / or address fields of the customer, based on non-null elements in the provided JSON structure
- Extra credit 2: Arrange that the CustomersRootResource support fetching the list of all customers in a JSON representation

JAX-RS 2.x Client API



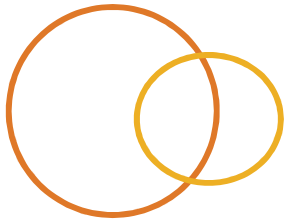
Objectives





- Send a GET, PUT, POST, PATCH, or DELETE request and get a Response object
- Control the “Accept:” MIME type of a request
- Send an entity body with a request and specify the content-type to which should be converted
- Read the response status from a response
- Read headers from a response
- Read the returned Entity from a response object, including converting structured entities to Java objects from JSON

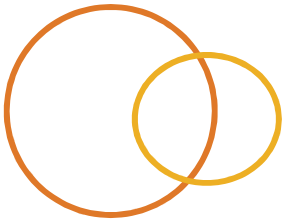
JAX-RS 2 Client



- JAX-RS 2 client is designed for fluent style programming
- "Object flow" has some variations, but the basic form is:
  - ClientBuilder
  - Client
  - WebTarget
  - InvocationBuilder
  - Invocation
  - Response

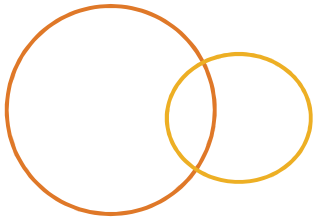
- Entity, provided to Invocation / Invocation.Builder

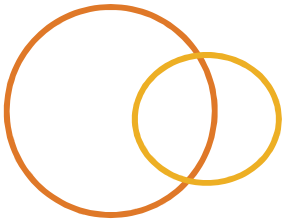
ClientBuilder



- The ClientBuilder is obtained through a static factory
  - ClientBuilder.newBuilder()
- ClientBuilder is generally used to configure SSL/TLS related features, such as key/trust store
- ClientBuilder.newClient provides the Client object

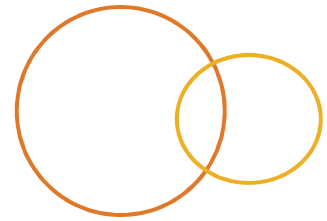
Client

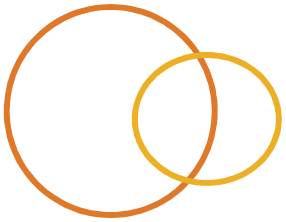




- The Client object is typically used to configure filters
- Client is then used to create one or more WebTarget objects

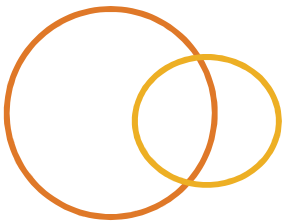
WebTarget





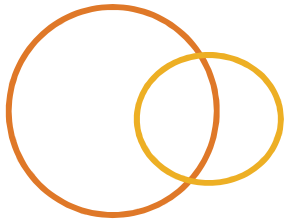
- WebTarget is used to describe a URL and can be used to create other WebTargets
  - In this sense, a URL can be used as a base URL for more specific requests
- WebTarget allows configuring of matrix and query parameters
  - Usually parameters will be configured on the “final” URLs, rather than on base URLs from which others will be derived
- WebTarget then creates an Invocation.Builder

Invocation.Builder



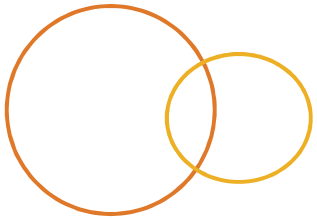
- Unsurprisingly, a builder for an Invocation
- The Builder may be used to manipulate headers that will be associated with the final invocation
  - Several methods are specific to particular, common, headers, such as accept
- Builder is then typically used to prepare an Invocation that's specific to a request type, e.g.:
  - ib.buildPost(Entity e)
  - ib.buildGet()

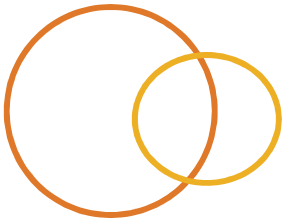
Invocation



- Invocation can be used to make the request immediately, resulting in a Response object, or allowing the response entity to be extracted directly
  - Extracting the entity directly prevents checking headers / status code
- Invocation can be used to launch the request asynchronously
  - Obtain a Future or a callback

Entity<T>





- Entity is generic allowing it to represent structured data that will be converted to JSON or similar
- Entity has static factory methods allowing several entity variations
  - form(MultivaluedMap<String,String> formData)
  - json(T entity)
  - text(T entity)
  - entity(T entity, MediaType mediaType)

Example GET Receiving Text

```
Client cl = ClientBuilder.newClient();
WebTarget base =
  cl.target("http://localhost:8080/”
            + "jeecontext/v1/customers/");
WebTarget oneCustomerName = base.path("/0/name");

Invocation.Builder ib = oneCustomerName.request();
ib.accept(MediaType.TEXT_PLAIN);

Response resp = ib.get();
String name = resp.readEntity(String.class);

System.out.println("Response is " + name);
```

Example GET Receiving JSON

```
Client cl = ... ← as before
WebTarget base = ... ← as before
Invocation.Builder ib = ... ← as before

WebTarget oneWholeCustomer = base.path("1");
Invocation.Builder ib = oneWholeCustomer.request();
ib.accept(MediaType.APPLICATION_JSON);

Response resp = ib.get();
System.out.println("As object response is “
  + resp.readEntity(CustomerTO.class));
```

Example POST Sending JSON

```
Client cl = ... ← as before
WebTarget base = ... ← as before
Invocation.Builder ib = ... ← as before

ib.accept(MediaType.APPLICATION_JSON);
Response resp = ib.buildPost(Entity.json(
    new CustomerTO("Tony", "Dunroamin")
  )).invoke();

CustomerTO returned =
  resp.readEntity(CustomerTO.class));
```

Other Response Elements

- Response is the same class as used in the server
- Status and headers can be read:
  - int status = resp.getStatus();
  - String aHeader = resp.getHeaderString("x-my-header");
  - MultivaluedMap<String, Object> hv = r.getHeaders();
- Other response data such as length, date, cookies, media type, and allowed methods, can be read from this object too

Sending PATCH Requests

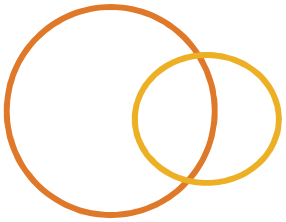
- In principle, a PATCH (or other non-standard request) may be sent like this:

```
...
Invocation.Builder ib = ...

Response r = ib.build("PATCH",
    Entity.json(new CustomerTO("Phoenix", null))
  ).invoke();
```

- In practice in Jersey this must be enabled first:

```
myClient.property(
    HttpUrlConnectorProvider.SET_METHOD_WORKAROUND,
    true);
```

Lab Exercise



- Create a stand-alone Java program that connects to your service and invokes an operation on one of your existing service endpoints, so that it fetches a JSON structure over the wire, and JAX-RS client converts it into a Java object in memory