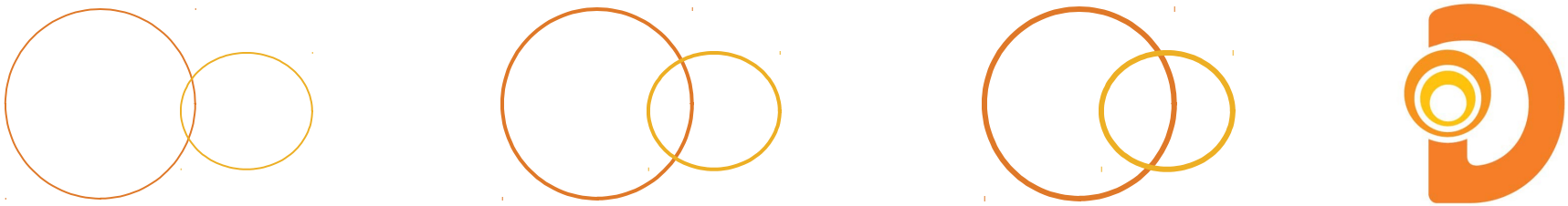
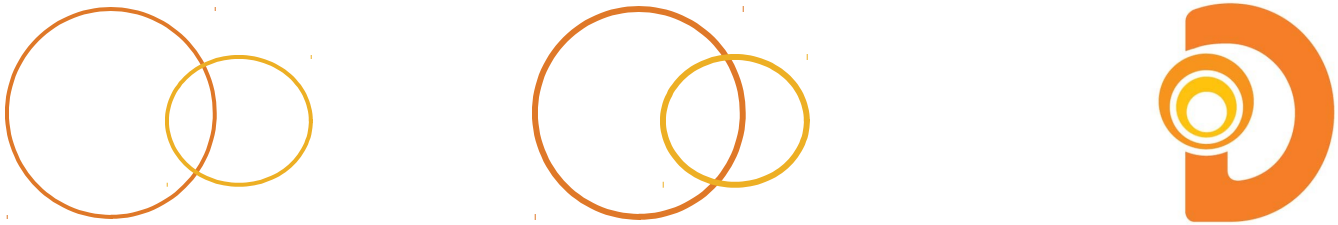


Sub-resource navigation and Design Considerations



Objectives

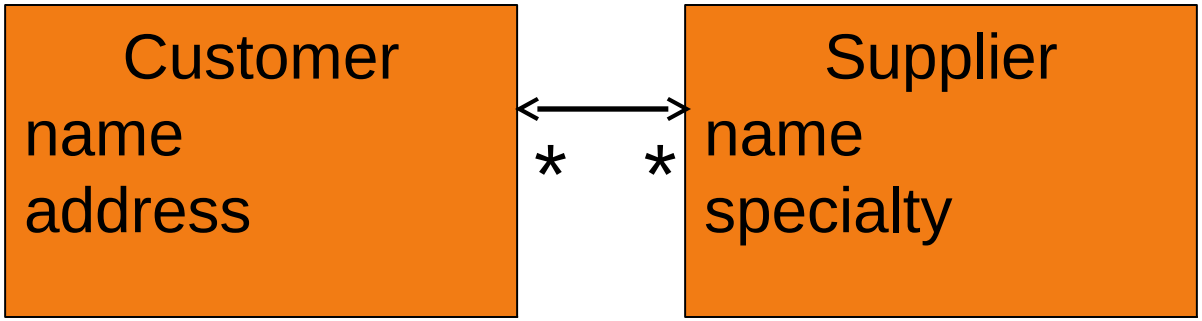
- Create sub-resource classes that allow URIs to be processed in segments by JAX-RS, facilitating consistent and bi-directional mapping of object models to RESTful URIs.
- Create sub-resource classes that can be reused when navigating to a particular type of resource via many different URI paths.
- Use 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.

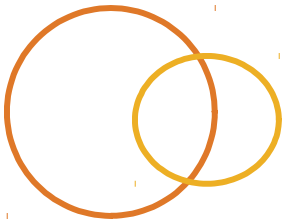
What's The Problem?

- 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

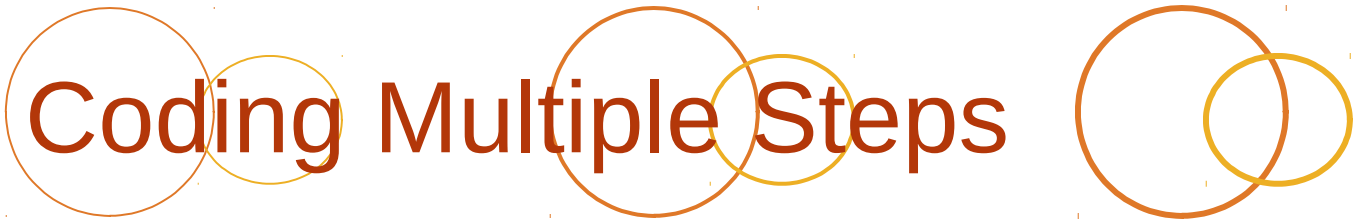
Addressing The Problem

- JAX-RS can handle a URI in multiple segments

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

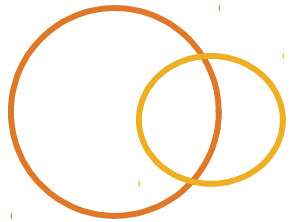
- Find a customer without a context (static-like)
- Find a specific supplier from the list of suppliers of the specific customer (customer-instance like)
- Find a specific customer from the list of customers of the specific supplier (supplier-instance-like)
- Find the name of the specific customer (customer-instance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