Contents

[The Object Model 1](#_Toc134273062)

[Model And URIs 2](#_Toc134273063)

[Defining The Data Format 2](#_Toc134273064)

[Read And Update Format 3](#_Toc134273065)

[Common Link Element 3](#_Toc134273066)

NOTE: I skipped the introduction part.

## The Object Model

1. let create **object model** of our **Order Entry System**.
2. Each **order** in the system represents a single **transaction or purchase** and is associated with a particular **Customer**.
3. An **order** is made of one or more **Line Items**.  
   **Line items** represent the type and number of each product purchased.
4. Based on this description of our system, we can deduce that the objects in our model are
   1. **Order,**
   2. **Customer,**
   3. **LineItem, and**
   4. **Product.**
5. Each Data Object in our Model has a unique identifier, which is the **id** Property.
6. Figure - 2.1 shows the UML Diagram of our Object Model.
7. Diagram

   Description automatically generated
8. **The following operations we want to perform on our system**.
   1. We will want to **browse** all orders as well as each individual order in our system.
   2. We will also want to **submit** new orders and update existing ones.
   3. We will want to **cancel and delete** existing orders.
9. The **OrderEntryService** object represents the operations we want to perform on our Order, Customer, LineItem and Product objects.

## Model And URIs

1. The first thing that we are going to do to create our **Distributed interface** is define and name each of the **distributed endpoints** in our System.
2. In **RESTful system**, endpoints are usually **referred to as resources and identified using a URI**.  
   URIs satisfy the **address-ability requirements** of a **RESTful Service**.
3. In our Object Model, we will be interacting with Orders, Customers and Products.  
   These will be our top-level resources.  
   We want to be able to object the list of each of these top-level items and want to interact with an individual items.  
   NOTE: Line items will be aggregated within an Order so not top-level resource, and we will expose them as sub resources under one particular **Order**.  
   Given this, here is the list of URIs that will be exposed in our system:  
   Text

   Description automatically generated  
   Text

   Description automatically generated with low confidence
4. A screenshot of a computer

   Description automatically generated with medium confidence

## Defining The Data Format

1. One of the most important things we have to do when defining a RESTful interface is determine how our resources will be represented over the wire to a client.
2. The most popular formats over internet:
   1. **XML**: Let’s choose it.
   2. **JOSN**: More condensed and JS can interpret it directly. Best for AJAX App.
3. We will define XML Schema for each representation (of resource) which defines grammar of a Data Format which defines the rules about how a document can be put together.

### Read And Update Format

1. When we’re reading or updating a resource, at that time the format of resources.
2. The XML format of our representations (of resources) will look a tiny bit different when we **read and update resources** from the server as compared to when we create new resources on the server.  
   Let’s look at our **read and update format (XML)** first.

#### Common Link Element <link>

1. Each format for Order, Customer, Product will have a common XML element <link>
2. 
3. The link tells the client (that obtains the XML Doc) where on the network the can interact with that particular resource.
4. “**rel**” attribute tells the client what the link has with the resource the URI (**href** attribute’s value) points to.

#### The Details

1. Let’s first discuss about the **Customer Representation Format**:  
     
   Graphical user interface, text, application

   Description automatically generated  
     
   Pretty Straightforward. We just take the object model for Customer from Figure 2.1 and expand its attributes as XML Elements.  
     
   Following is data format for **Product**.  
   Logo

   Description automatically generated with low confidence  
     
   The following is **Order** Representation Format with line items aggregated:  
   Diagram

   Description automatically generated with medium confidence  
   The above snapshot (Order) is a good example of Composition with useful <link>

### Create Format

1. When creating a new resource (Customer, Product, Order), it doesn’t make any sense to include id attribute, <link> XML element.  
   As server will generate a new id for it will insert into DB.  
   We also don’t know the URI of a new object because the Server will generate this for new resource.
2. So the XML for creating a new Product would look something like this:  
   A picture containing shape

   Description automatically generated

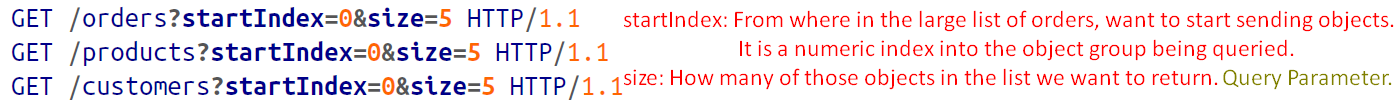
## Assigning HTTP Method

1. The final thing we have to do is decide.
   1. Which HTTP methods will be exposed for each of our resources and
   2. What these methods will do.
2. It is crucial that we don’t assign functionality to an HTTP method that supersedes the **specification-defined boundaries of that method**.
   1. **For example**: An HTTP GET on a particular resource should be read-only which means it should not change the state of the resource on which it is invoked.
3. Following snapshot is not clear.Text

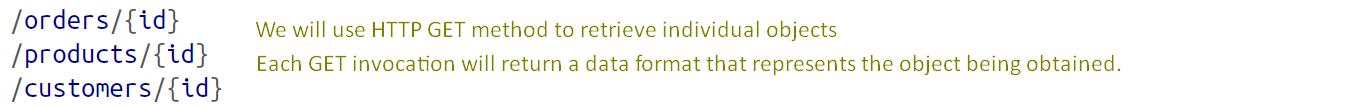
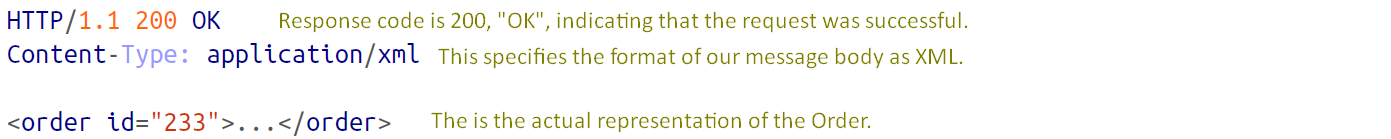
   Description automatically generated
4. Let’s walk through each method of our object model to determine which URIs and HTTP methods are used to represent them.

### Browsing All Orders, Customers, or Products

1. The objects Order, Customers and Products in our **Object Model** are very similar in how they are accessed or modified.
2. As our remote clients want to browse all the Orders, Customers, Products, so the following URIs represent objects as a group:  
     
   To get a list of Orders, Products, Customers, the remote client will call an HTTP method GET on the URI of the **Object Group**.  
     
   Our service will respond with a data format that represents all products.  
     
   Graphical user interface, text, application

   Description automatically generated
3. One problem with the bulk operation is that we may have thousands of Orders, Customers, Products in our system and we may overload our client and hurt our response time.  
   To mitigate this issue, we may allow our client to specify query parameters on the URI to limit the size of the dataset returned:  
     
   **NOTE**: Those query parameters will be optional. A client doesn’t have to specify in its URI when **crafting** its request to the server.

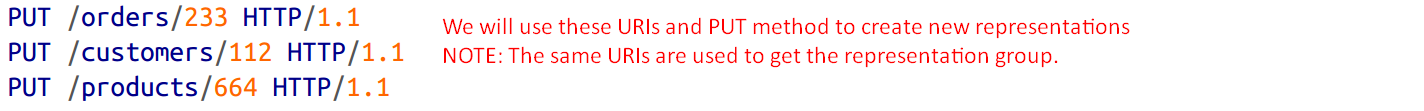
## Obtaining Individual Orders, Customers, or Products

1. As we mentioned already, we will use **URI pattern** to obtain individual Orders, Products, Customers.
2.   
     
     
     
   For this request, the client is interested in **getting a representation of the Order** with an order id of 233.  
   GET request for Product and Customer will work the same way.  
     
   **Response will look like**:  
   

## Creating an Order, Customer, or Product

1. There are two possible ways in which a client can create Orders, Customers, Products within our Order Entry System.
   1. By using HTTP PUT.
   2. By Using HTTP POST.

### Creating with PUT

1. The **HTTP** definition of **PUT** states that it can be **used to create or update a resource** on the server.
2. To create Order, Customer, Product with PUT, the client simply sends the representation of the new object to the URI Location that represents the object.  
   **For Example**:
   1. URI “/orders” represents the order group and the same URI, we will use to send representation of new order object to create on server.
3. In the following snapshot, note that while creating a new object on server using PUT, we’re also passing the ID of the new object in the URI itself.  
   We will study why passing ID in the URL just in a few moments.
4. **Specification:**
   1. **PUT** is required **by the specification** to send a response code of **201 “Created”** if a new resource is created **successfully** on the server as a result of the request.
   2. **Specification** also states that PUT is **idempotent** because no matter how many times we tell server to “Create” our Order with the same URI, the same bits are stored at the /orders/233 location.  
      Sometimes, a PUT request will fail and the client will not know if the request was delivered and processed at the server.  
      Idempotency guarantees that it is OK for the client to retransmit the PUT operation and not worry about any adverse side effects.
5. **Disadvantage of Using PUT**:
   1. The disadvantage of using PUT to create resources is that the client has to provide the Unique ID that represents the object it is creating.
   2. While it is usually possible for the client to generate this unique ID, most **Application Designers** prefer their servers (Usually through DB) to create this ID.
   3. In our hypothetical order entry system, we want our server to control the generation of **resource IDs**.
   4. **Solution**: We can switch to **HTTP POST**.

### Creating With Post

1. Creating resources with POST is a little more complex than using PUT.
2. A client sends a representation of the new object to the parent URI of its representation, leaving out the numeric target ID.  
   Graphical user interface

   Description automatically generated with medium confidence
3. The service will receive POST request and process the XML, and create a new order in the DB using a database-generated ID.
4. But we have put our client in the quandary.  
   What if the client wants to edit, update, or cancel the order that the client created with POST without passing resource id (as client doesn’t have resource id because now generated by server by DB)?  
   **Solution**: We will add the resource id generated by DB on server in HTTP response.  
   A picture containing graphical user interface

   Description automatically generated
5. HTTP requires to respond with 201, “Created” (just like PUT when using PUT as resource creation).