

|  |  |
| --- | --- |
| Employee Name: | MANOJ KUMAR YEKOLLU |
| Training Supervised by: | **KRISHNA BODDULURI** |
| Training Task #: | **ASSIGNMENT10162019** |
| Training Resource Materials: | **Udemy REST API Course** |
| Training Task Date: | **10/16/2019** |
| Task Due Date: | **10/30/2019** |
| Task Submitted Date: | **10/30/2019** |
| Github link: |  |
| Technologies used for Training | **REST, Java, Spring Boot** |

**Task Description/Requirement:**

**Learn How to Design REST API and Document the Findings**

**High Level Synopsis:**

**Designing a REST API:**

* **Resources, Associations, URI**
* **CRUD, Error handling, Caching**
* **Change Management, Versioning**
* **Partial Responses and Pagination**
* **How the Popular APIS managing these concepts.**

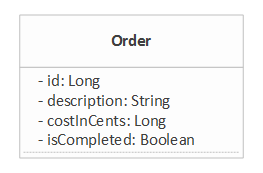
**I acknowledge that this document can be supplied to USCIS in compliance with CPT/OPT/STEM OPT audit:**

**Output:**

**Designing Domain Model:**

As part of developing the REST API we need to create a domain model.

For our API, we have the Order Object with few attributes.



Based on this Object, we can define the Response bodies as well.

{

“id” : 1,

“description” : “Some sample order”,

“CostInCents”: 250

“Complete”: false

}

There is an important addition that is made to our architecture: Domain objects are not sent directly to the user. Instead, they are wrapped in resources and the resources are provided to the user. This provides a level of indirection between the domain object and how we present the domain object to the user. For example, if we wish to present the user with a different name for a field in our domain model (say orderName instead of simply name), we can do so using a resource. Although this level of indirection is very useful in decoupling our presentation from the domain model, it does allow duplication to sneak in. In most cases, the resource will resemble the interface of the domain object, with a few minor additions. This issue is addressed later when we implement our presentation layer.

The resource object also provides an apt place for us to introduce our hypermedia links. According to the [Richardson model for REST web services](https://martinfowler.com/articles/richardsonMaturityModel.html), hypermedia-driven services are the highest capability level of a REST application and provide important information associated with the resource data. For example, we can provide links for deleting or updating the resource, which removes the need for the client consuming our REST web service to know the REST endpoints for these actions. In practice, the returned resource (deserialized to JSON) may resemble the following:

{

"id": 1,

"description": "Some sample order",

"costInCents": 250,

"complete": false

"\_links": {

"self": {

"href": "http://localhost:8080/order/1"

},

"update": {

"href": "http://localhost:8080/order/1"

},

"delete": {

"href": "http://localhost:8080/order/1"

}

}

}

Given these links, the consumer is no longer required to build the URLs for the update, delete, or self-reference REST endpoints. Instead, it can simply use the links provided in our hypermedia-driven response. Not only does this reduces the logic necessary for interacting with our REST web service (no longer do the URLs need to be built), but it also encapsulates the logic for the construction of the URLs. For example, suppose we change our REST service to require a query parameter, such as sorting: if we provide the links to the consumer, we can adjust for that change, without making any changes to the consumer:

{

"id": 1,

"description": "Some sample order",

"costInCents": 250,

"complete": false

"\_links": {

"self": {

"href": "http://localhost:8080/order/1?sorting=default"

},

"update": {

"href": "http://localhost:8080/order/1?sorting=default"

},

"delete": {

"href": "http://localhost:8080/order/1?sorting=default"

}

}

}

Although generating these links could be tedious and subject to a large number of bugs (i.e. what if the IP address of the machine hosting the web service changes?), the Spring Hypermedia as the Engine of Application State (HATEOAS, commonly pronounced *hay-tee-os*) framework provides numerous classes and builders that allow us to create these links with ease. This topic will be explored further when we delve into the implementation of our presentation layer.

Before moving to the implementation of our web service, we must pull our design together and devise a plan of action to create it. At the moment, we have a single domain object, Order, instances of whom will be persisted in an in-memory database and served up (within a resource) to clients using our REST endpoints. This design leaves us with four main steps:

1. Implement the domain model
   * Create the Order domain class
2. Implement the data source layer
   * Create an in-memory database
   * Implement the CRUD operations for the Order domain class
3. Implement the presentation layer
   * Create the REST endpoints
   * Create the Order resource
   * Create assembler to construct an Order resource with proper HATEOAS links
4. Pull the application together
   * Create the main method that will run the application

