# Creating a REST Web Service With Java and Spring (Part 3)

**by Justin Albano MVB · Sep. 23, 17 · Java Zone**



In the previous article, we laid the foundation of our web service by creating the data source and domain layers. In this article, we will continue with implementing our web service by constructing the most visible aspect of our web service: The presentation layer.

## Implementing the Presentation Layer

Without the aid of a web application framework, creating a presentation layer would be a daunting task, but after many years, the patterns and conventional designs of RESTful web services have been captured in the Spring Model-View-Controller (MVC) framework. This framework allows us to create RESTful endpoints with much the same ease as we saw during the development of our data source layer, using annotations and helper classes to do most of the heavy lifting for us.

Starting with the class that is the most depended on and requires the least dependencies, we will create the `OrderResource` first:

```
public class OrderResource extends ResourceSupport {

    private final long id;
    private final String description;
    private final long costInCents;
    private final boolean isComplete;

    public OrderResource(Order order) {
        id = order.getId();
        description = order.getDescription();
        costInCents = order.getCostInCents();
        isComplete = order.isComplete();
    }

    @JsonProperty("id")
    public Long getResourceId() {
        return id;
    }

    public String getDescription() {
        return description;
```

```
}
```

The `OrderResource` class is strikingly similar to our `Order` class, but with a few main differences. First, we inherit from the `ResourceSupport` class provided by the Spring HATEOAS packages, which allows us to attach links to our resource (we will revisit this topic shortly). Second, we have made all of the fields `final`. Although this is not a requirement, it is a good practice because we wish to restrict the values of the fields in the resource from changing after they have been set, ensuring that they reflect the values of the `Order` class for which it is acting as a resource.

In this simple case, the `OrderResource` class has a one-to-one field relationship with the `Order` class, which begs the question: Why not just use the `Order` class? The primary reason to create a separate resource class is that the resource class allows us to implement a level of indirection between the `Order` class itself and how that class is presented. In this case, although the fields are the same, we are also attaching links to the fields in the `Order` class. Without a dedicated resource class, we would have to intermix the domain logic with the presentation logic, which would cause serious dependency issues in a large-scale system.

A happy medium between the duplication between the `OrderResource` and `Order` classes is the use of Jackson annotations in order to use the fields of the `Order` class to act as the fields of the `OrderResource` class when the `OrderResource` class is serialized to JSON. In the Spring MVC framework, our resource class will be converted to JSON before being sent over HTTP to the consumers of our web service. The default serialization process takes each of the fields of our class and uses the field names as the keys and the field values as the values. For example, a serialized `Order` class may resemble the following:

```
{
    "id": 1,
    "description": "Some test description",
    "costInCents": 200,
    "complete": true
}
```

If we tried to directly embed the `Order` object inside our `OrderResource` object (implemented our `OrderResource` class to have a single field that holds an `Order` object in order to reuse the fields of the `Order` object), we would end up with the following:

```
{
    "order": {
        "id": 1,
        "description": "Some test description",
        "costInCents": 200,
        "complete": true
    }
```

```
    private final Order order;

    public OrderResource(Order order) {
        this.order = order;
    }
}
```

Serializing this class would result in our desired JSON:

```
{
    "id": 1,
    "description": "Some test description",
    "costInCents": 200,
    "complete": true
}
```

While unwrapping the nested `Order` object significantly reduces the size of the `OrderResource` class, it has one drawback: When the internal fields of the `Order` changes, so do the resulting serialized JSON produced from the `OrderResource` object. In essence, we have coupled the `OrderResource` class and the internal structure of the `Order` class, breaking encapsulation. We walk a fine line between the duplication seen in the first approach (replicating the `Order` fields within `OrderResource`) and the coupling seen in the JSON unwrapping approach. Both have advantages and drawbacks, and judgment and experience will dictate the best times to use each.

One final note on our `OrderResource` class: We cannot use the `getId()` method as our getter for our ID since the `ResourceSupport` class has a default `getId()` method that returns a link. Therefore, we use the `getResourceId()` method as our getter for our `id` field; thus, we have to annotate our `getResourceId()` method since, by default, our resource would serialize the ID field to `resourceId` due to the name of the getter method. To force this property to be serialized to `id`, we use the `@JsonProperty("id")` annotation.

With our resource class in place, we need to implement an assembler that will create an `OrderResource` from an `Order` domain object. To do this, we will focus on two methods: (1) `toResource`, which consumes a single `Order` object and produces an `OrderResource` object, and (2) `toResourceCollection`, which consumes a collection of `Order` objects and produces a collection of `OrderResource` objects. Since we can implement the latter in terms of the former, we will abstract this relationship into an ABC:

```
public abstract class ResourceAssembler<DomainType, ResourceType> {


    public abstract ResourceType toResource(DomainType domainObject);
```

calling the `toResource` method on each of the `Order` objects in the consumed list. We then create an `OrderResourceAssembler` class that provides an implementation for the `toResource` method:

```
@Component
public class OrderResourceAssembler extends ResourceAssembler<Order, C


    @Autowired
    protected EntityLinks entityLinks;

    private static final String UPDATE_REL = "update";
    private static final String DELETE_REL = "delete";

    @Override
    public OrderResource toResource(Order order) {

        OrderResource resource = new OrderResource(order);

        final Link selfLink = entityLinks.linkToSingleResource(order);


        resource.add(selfLink.withSelfRel());
        resource.add(selfLink.withRel(UPDATE_REL));
        resource.add(selfLink.withRel(DELETE_REL));

        return resource;
    }
}
```

In this concrete class, we simply extend the `ResourceAssembler` ABC, declaring the domain object type and the resource object type, respectively, as the generic arguments. We are already familiar with the `@Component` annotation, which will allow us to inject this assembler into other classes as needed.
The autowiring of the `EntityLinks` class requires some further explanation.

As we have already seen, creating links for a resource can be a difficult task. In order to remedy this difficulty, the Spring HATEOAS framework includes an `EntityLinks` class that provides helper methods that provide for the construction of links using just the domain object type. This is accomplished by having a REST endpoint class (which we will define shortly) use the `@ExposesResourceFor(Class domainClass)` annotation, which tells the HATEOAS framework that links built for the supplied domain class should point to that REST endpoint.

For example, suppose we create a REST endpoint that allows a client to create, retrieve, update, and delete `Order` objects. In order to allow for Spring HATEOAS to help in the creation of links to delete and update `Order` objects,

resource object itself is straightforward, but the creation of the links requires some explanation. Using the `EntityLinks` class, we can create a link to our own resource by specifying (using the `linkToSingleResource` method) that we wish to create a link to an `Order`, which uses the Spring HATEOAS `Identifiable` interface to obtain the ID of the object. We then reuse this link to create three separate links: (1) a self link, (2) an update link, and (3) a delete link. We set the relative value (rel) of the link using the `withRel` method. We then return the fully constructed resource object. Given the three links we have created, our resulting `OrderResource`, when serialized to JSON, looks as follows:

```
{
    "id": 1,
    "description": "Some sample order",
    "costInCents": 250,
    "complete": false
    "_links": {
        "self": {
            "href": "http://localhost:8080/order/1"
        },
        "udpate": {
            "href": "http://localhost:8080/order/1"
        },
        "delete": {
            "href": "http://localhost:8080/order/1"
        }
    }
}
```

The `self` link tells the consumer that if a link to this resource is needed, the provided HREF can be used. The `update` and `delete` links tell the consumer that if this resource should be updated or deleted, respectively, the provided HREF should be used.

With the `OrderResource` class and its assembler completed, we can move onto the last, and arguably most essential step: creating the REST endpoints. In the Spring MVC framework, a REST endpoint is created by implementing a controller class (a class annotated with `@Controller` or `@RestController`) and adding methods that correspond to the desired REST endpoints. We will list our controller class first and then explain the meaning of each section of code:

```
@RestController
@ExposesResourceFor(Order.class)
@RequestMapping(value = "/order", produces = "application/json")
public class OrderController {

    @Autowired
    private OrderRepository repository;
```

```
16      }
17
18      @RequestMapping(method = RequestMethod.POST, consumes = "applicati
19      public ResponseEntity<OrderResource> createOrder(@RequestBody Orde
20          Order createdOrder = repository.create(order);
21          return new ResponseEntity<>(assembler.toResource(createdOrder)
22      }
23
24      @RequestMapping(value = "/{id}", method = RequestMethod.GET)
25      public ResponseEntity<OrderResource> findOrderById(@PathVariable L
26          Optional<Order> order = repository.findById(id);
27
28          if (order.isPresent()) {
29              return new ResponseEntity<>(assembler.toResource(order.get
30          }
31          else {
32              return new ResponseEntity<>(HttpStatus.NOT_FOUND);
33          }
34      }
35
36      @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
37      public ResponseEntity<Void> deleteOrder(@PathVariable Long id) {
38          boolean wasDeleted = repository.delete(id);
39          HttpStatus responseStatus = wasDeleted ? HttpStatus.NO_CONTENT
40          return new ResponseEntity<>(responseStatus);
41      }
42
43      @RequestMapping(value = "/{id}", method = RequestMethod.PUT, consu
44      public ResponseEntity<OrderResource> updateOrder(@PathVariable Lon
45          boolean wasUpdated = repository.update(id, updatedOrder);
46
47          if (wasUpdated) {
48              return findOrderById(id);
49          }
50          else {
51              return new ResponseEntity<>(HttpStatus.NOT_FOUND);
52          }
53      }
54  }
```

The `@RestController` annotation, as stated above, tells Spring that this class

on is `http://localhost:8080`, the path to reach this controller will be `http://localhost:8080/order`. We also include the type of the data produced by the controller, or `application/json`, in the request mapping to instruct Spring that this controller class produces JSON output (Spring will, in turn, include `Content-Type: application/json` in the header of any HTTP responses sent).

Within the controller class, we inject the `OrderRepository` and `OrderResourceAssembler` components which will allow us to access the stored `Order` objects and create `OrderResource` objects from these domain objects, respectively. Although we have a dependency to the data store layer within our controller class, we lean on Spring to provide us with an instance of the `OrderRepository`, ensuring that we are only dependent on the external interface of the repository, rather than on the creation process.

The last portion of the controller class is the most crucial: the methods that will perform the REST operations. In order to declare a new REST endpoint, we use the `@RequestMapping` to annotate a method and supply the HTTP verb that we wish to use. For example, if we look at the `findAllOrders` method,

```
@RequestMapping(method = RequestMethod.GET)
public ResponseEntity<Collection<OrderResource>> findAllOrders() {
    List<Order> orders = repository.findAll();
    return new ResponseEntity<>(assembler.toResourceCollection(orders)
}
```

we use the `@RequsetMapping` annotation to inform the Spring MVC framework that `findAllOrders` is intended to be called when an HTTP GET is received. This process is called mapping, and as we will see later, Spring will establish this mapping during deployment. It is important to note that the path of the mapping is relative to the path declared at the controller level. For example, since our `OrderController` is annotated with `@RequestMapping("/order")` and no path is explicitly declared for our `findAllOrders` method, the path used for this method is `/orders`.

The return type of our `findAllOrders` method is particularly important. The `ResponseEntity` class is provided by the Spring MVC framework and represents an HTTP response to an HTTP request. The generic parameter of this class represents the class of the object that will be contained in the response body of the call; this response body object will be serialized to JSON and then returned to the requesting client as a JSON string.

In this case, we will return a collection of `OrderResource` objects (the list of all existing orders) after obtaining them from the `OrderRepository`. This list is then assembled into a list of `OrderResource` objects and packed into a `ResponseEntity` object in the following line:

notable exceptions. In the case of our `findOrderById`, `deleteOrder`, and `updateOrder` methods, we adjust the path of the REST endpoint to include `/{id}`. As previously stated, this path is relative to the controller path, and thus the resolved path for each of these methods is `/order/{id}`. The use of curly braces (`{` and `}`) in a path denotes a variable whose value will be resolved to a parameter in the method it annotates. For example, if we look at the `findOrderById` method

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
public ResponseEntity<OrderResource> findOrderById(@PathVariable Long
    // …Body hidden for brevity…
}
```

we see that the name of the parameter (`id`) matches the variable in the path and is decorated with the `@PathVariable` annotation. The combination of these two adornments tells Spring that we wish to have the value of the `id` variable in the path passed as the runtime value of the `id` parameter in our `findOrderById` method. For example, if a GET request is made to `/order/1`, the call to our `findOrderById` method will be `findOrderById(1)`.

Another difference that must be addressed is the return value of the `deleteOrder` method: The return value of `ResponseEntity<Void>` tells Spring MVC that we are returning a `ResponseEntity` with an associated HTTP status code but we are not including a response body (the response body is void). This results in an empty response body for the response sent to the requester.

The last difference deals with the parameters of `updateOrder`. In the case of updating an order, we must use the contents of the request body, but doing so as a string would be tedious. In that case, we would have to parse the string and extract the desired data, ensuring that we do not make an easy error during the parsing process. Instead, Spring MVC will deserialize the request body into an object of our choice. If we look at the `updateOrder` method

```
@RequestMapping(value = "/{id}", method = RequestMethod.PUT, consumes
public ResponseEntity<OrderResource> updateOrder(@PathVariable Long id
    // …Body hidden for brevity…
}
```

we see that the `updatedOrder` parameter is decorated with the `@RequestBody` annotation. This instructs Spring MVC to deserialize the HTTP request body into the `updateOrder` parameter, which takes the JSON request body (denoted by the `consumes = “application/json”` field in the `@RequestMapping` annotation) and deserializes it into an `Order` object (the type of `updateOrder`). We are then able to use the `updatedOrder` parameter in

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

The `@EnableEntityLinks` annotation configures Spring to include support for the `EntityLinks` class in our system (allowing us to inject the `EntityLinks` object).
Likewise, the `@EnableHypermediaSupport` annotation instructs Spring to include support for HATEOAS, using the Hypermedia Application Language (HAL) when producing links. The final annotation, `@SpringBootApplication` , marks our application a Spring Boot application, which configures the boilerplate code needed to start Spring and also instructs Spring to component scan our packages to find injectable classes (such as those annotated with `@Component` or `@Repository` ).

The remainder of the main method simply runs the Spring Boot application, passing the current class and the command line arguments to the `run` method. Using Spring Boot, starting our web application is nearly trivial, which leaves us with only one thing left to do: Deploy and consume our RESTful web service. We will cover this deployment and testing process in the next and final entry in this series.



Topics: WEB DEV , SPRING , JAVA 8 , REST WEB SERVICE , WEB APPLICATION DEVELOPMENT