Server Side APIs: Part 2 Tutorial

In this tutorial, you'll continue working on an application that uses Tech Elevator Locations as the data model. You'll add the ability to update and delete a location as well as perform data validation to inform the client of any problems.

Step One: Import into Eclipse and explore starting code

Before you begin, import the locations starter code into Eclipse using the "Import Existing Maven Projects" feature. Then, review the starting code. It should look familiar to you as it picks up where you left off in the previous tutorial.

DAO

In the previous tutorial, all of the locations were created in the `LocationController`'s constructor. For this tutorial, all of the data initialization and CRUD operations have been moved to the `MemoryLocationDAO.java` class. A DAO instance is provided to the controller's constructor at runtime by the Spring framework using Dependency Injection:

```
public LocationController(LocationDAO dao) {
    this.dao = dao;
}
```

If you open the `LocationDAO.java` file you will notice it is an interface. The `MemoryLocationDAO` class implements this interface and is annotated with `@Component` so that it can be injected into the `LocationController`.

```
@Component
public class MemoryLocationDAO implements LocationDAO {
    ...
}
```

Controller Update

In the previous version of this application, you defined each of the `@RequestMapping()` paths at the method level. If you have a base path like `/locations`, and you find yourself reusing it several times, you can move it to the class level:

```
@RestController
@RequestMapping("/locations")
public class LocationController {

}
```

If the path for listing locations is `/locations`, you enter a blank string. The base path is inherited from the class level `@RequestMapping("/locations")`:

```
@RequestMapping( path = "", method = RequestMethod.GET )
public List<Location> list() {
    return dao.list();
}
```

Location Not Found

In the reading material, you learned about creating a custom exception that could return a `404(Not Found)` response to the client when a resource couldn't be located. The class `LocationNotFoundException.java` has been created for you in the `com.techelevator.locations.exception` package:

```
@ResponseStatus( code = HttpStatus.NOT_FOUND, reason = "Location not found.")
public class LocationNotFoundException extends Exception {
    private static final long serialVersionUID = 1L;

    public LocationNotFoundException() {
        super("Location not found");
    }
}
```

You'll learn where and how to use this later in the tutorial.

Step Two: Run your project

Now that you've set up your project in Eclipse and reviewed the starting code, run it to make sure everything works. It's best to make sure the application runs before adding anything new to it.

Step Three: Add Location data validation

In the previous tutorial, there was some basic validation in the `add()` method of the controller. If the location wasn't null, you'd add it to the list of locations:

```
@RequestMapping( value = "/locations", method = RequestMethod.POST)
public Location add(@RequestBody Location location) {
    if( location != null ) {
      locations.add(location);
    }
}
```

What happens if the location isn't null but is missing information? For locations, you want all these fields to be required, not blank:

- name

- address
- city
- state
- zip

If you open `Location.java`, you'll see the list of fields for this class. Before looking at the answer, add the `@NotBlank()` annotation for each required field:

```
public class Location {

    private int id;
    private String name;
    private String address;
    private String city;
    private String state;
    private String zip;

}
```

> The ID is not required because it's generated by the `MemoryLocationDAO` class.

Next, add a custom message by setting the message parameter of the `@NotBlank` annotation. Remember that adding custom messages helps the client understand what field failed data validation:

```
public class Location {

    private int id;
    @NotBlank(message = "The field name is required.")
    private String name;
    @NotBlank(message = "The field address is required.")
    private String address;
    @NotBlank(message = "The field city is required.")
    private String city;
    @NotBlank(message = "The field state is required.")
    private String state;
    @NotBlank(message = "The field zip is required.")
    private String zip;

}
```

Feel free to test the `create()` method in Postman. If you send an object with a blank name, do you get the proper error message back?

Step Four: Create a new location

Now that you have some basic validation on your `Location` class, you'll need to make sure the controller receives a **valid** argument. To check them in your Controller, you need to add the `@Valid` annotation to the model:

```
@RequestMapping( value = "", method = RequestMethod.POST)
public Location add(@Valid @RequestBody Location location) {
    return dao.create(location);
}
```

This is also a good time to fix another issue from the previous tutorial. If you open Postman and create a new `Location`, what status code is returned?

Right now, you get back a `200(OK)` status code, but as a best practice, you should return a `201(Created)` status code so the client knows that the new resource was created. You can use the `@ResponseStatus` annotation on the controller method and specify what status code must be returned:

```
@ResponseStatus(HttpStatus.CREATED)
@RequestMapping( value = "", method = RequestMethod.POST)
public Location add(@Valid @RequestBody Location location) {
    return dao.create(location);
}
```

Step Five: Update an existing location

To add the ability to update an existing location, you need to create a new controller method that responds to the `PUT` request method. A client would specify the URL to a specific location, like `locations/1`, and send the JSON data that included the updated location. Using the ID in the URL, you can look up the existing location and update its data.

In `LocationController`, create a new method called `update()`:

```
@RequestMapping(path = "/{id}", method = RequestMethod.PUT)
public Location update(@Valid @RequestBody Location location, @PathVariable int
id) throws LocationNotFoundException {

}
```

You can see that `@Valid` is called, so the validation step is already being handled for you. Also, this method needs to throw the `LocationNotFoundException` that could come from the `dao.update()` method. This is the class you saw earlier and returns a `404(NotFound)` if the client tries to update a location that doesn't exist.

The last step is to call the `dao.update()` method which returns the updated location so you can return that back to the caller:

```
@RequestMapping(path = "/{id}", method = RequestMethod.PUT)
public Location update(@Valid @RequestBody Location location, @PathVariable int
id) throws LocationNotFoundException {
```

```
    return dao.update(location, id);
}
```

You can test this by running a `PUT` request in Postman. If you wanted to update the Tech Elevator Cleveland location, the path would be `http://localhost:8080/locations/1`. Make sure to change the request method to `PUT` and the `Content-Type` to `application/json`. You can send the following object to change the title:

```
{
    "id": 1,
    "name": "THIS IS A NEW TITLE",
    "address": "7100 Euclid Ave #14",
    "city": "Cleveland",
    "state": "OH",
    "zip": "44103"
}
```

If you try updating a location that doesn't exist, like `/locations/99`, what happens?

> Don't worry: the changes are only stored in memory, so you won't change this permanently. If you restart your application, you'll see the data has been reset.

Step Six: Delete a location

Add a new `delete()` method to your controller. Like `PUT`, `DELETE` targets the URL for a specific location. Unlike the `update()` method, you won't return anything, so you'll want to make sure you return the status code `204(No Content)`.

You also want to make sure you return a `404 Not Found` if the client tries to delete a location that doesn't exist by throwing a `LocationNotFoundException`:

```
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(path = "/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable int id) throws LocationNotFoundException {
    dao.delete(id);
}
```

To test in Postman, run a `DELETE` on any of the existing locations. After running it and getting a successful `204(No Content)` status back, run the `GET` to list all of the locations. You'll see that your location has been deleted. Like the `update()` method, try sending a `DELETE` request to a location that doesn't exist and see what happens.

Summary

In this tutorial, you learned:

- How to add validation to your data model.

- How to validate incoming requests with the `@Valid` annotation.
- How to return a specific status code using the `@ResponseStatus` annotation.
- How to create a method that updates a resource in your controller.
- How to create a method that deletes a resource in your controller.