Web Services POST, PUT, DELETE Tutorial (Java)

In this tutorial, you'll extend the meetup locations example from the first day by adding features to:

- Add a new location (`POST`)
- Modify an existing location (`PUT`)
- Remove a location (`DELETE`)
- Capture and handle HTTP exceptions

When complete, this produces a full **C**reate **R**ead **U**pdate **D**elete (CRUD) console Web client application.

Step One: Start the server

Before you start, make sure that the web API is up and running. First, change directories into the `./server/` folder.

Next, run the command `npm install` to install any dependencies. You won't need to do this on any subsequent run.

While still in the command line, run the command `npm start` to start the json-server application. If there aren't any errors, you'll see the following, which means that you've successfully set up your web API:

```
\{^_^}/ hi!

 Loading data-generation.js
 Done

 Resources
 http://localhost:3000/locations

 Home
 http://localhost:3000

 Type s + enter at any time to create a snapshot of the database
```

When json-server is running on port 3000, no other applications—including other copies of json-server—are able to use port 3000. To free up the port, be sure to stop json-server when you're finished with this tutorial. You do that by selecting the terminal where you typed `npm start` and pressing `Ctrl+C`. Or if you've already closed that terminal, open a new terminal and type:

```
taskkill -T -F -IM node.exe
```

In this tutorial, you'll modify data on the server. As you're working, you may come across a situation where you want to reset the data. To do this, first stop the server with `Ctrl+C`, then restart it with `npm start`.

Step Two: Review the starting code

Application structure

The `src/main/java/com/` folder:

- techelevator
  - locations
    - App.java `<-- main application driver class`
    - model
      - Location.java `<-- Location data model class`
    - services
      - ConsoleService.java `<-- Console input and output service`
      - LocationService.java `<-- REST client and Web API access performed here`
  - util
    - BasicLogger.java `<-- Error logging class`
    - BasicLoggerExeception.java `<-- Logger exception class`

Provided code versus your code

Everything but part of the `LocationService` class is provided for you. You'll complete the `add()`, `update()`, and `delete()` methods in that class. The `getAll()` and `getOne()` methods are based on the work you did in a previous tutorial. The `makeEntity()` method is a helper method to assist with making a Web API request.

The methods in the `App` class use the `ConsoleService` class to prompt and retrieve input from the user and use the `LocationService` class to request and retrieve the data from the API.

Step Three: Add a location with POST

Open the `LocationService.java` file and find the `add()` method.

```
public Location add(Location newLocation) {
    //Step Three: Add a location with POST
    return null;
}
```

The code that calls this function passes in a newly created `Location` object. This method sends the data for that location to the API.

First, you'll use the helper method named `makeEntity()`. The purpose of this method is to add a header to the POST request. This lets the server know the *Content Type* contained in the request. Here, that type is set to `MediaType.APPLICATION_JSON`. Then an `HttpEntity` is created, containing both the new header and the location object.

To use this helper method, you'll need to pass it the location, and assign the return to a new variable of type `HttpEntity`. Add this as the first line of the `add()` method:

```
HttpEntity<Location> entity = makeEntity(newLocation);
```

Now that you've constructed an `HttpEntity`, you're ready to use `RestTemplate` to `POST` it to the server. To do this, you'll use the `postForObject()` method of `RestTemplate`. This method returns the object that's returned from the server.

The `postForObject()` method requires three parameters: a URL, the Entity, and the class used to construct the return object. Use the `API_BASE_URL` as the URL and `Location.class` as the third parameter. Remember that the URL of the request changes based upon the goal. In this case, the POST request is made to "http://localhost:3000/locations." This is the same URL as a GET request. The difference is the HTTP method being used.

Create a new `Location` variable named `returnedLocation`, and assign the result of this call to it. Finally, return `returnedLocation`:

```
Location returnedLocation = null;
returnedLocation = restTemplate.postForObject(API_BASE_URL, entity,
Location.class);
return returnedLocation;
```

Note that the difference between `returnedLocation` and `newLocation` is that `returnedLocation` has the id that was assigned by the API when it added it to the datastore.

Next, remove the `return null` statement at the end of the method, if still present. You only needed it to satisfy the Java compiler when you first opened the tutorial project.

The `add()` method looks like this:

```
public Location add(Location newLocation) {
    HttpEntity<Location> entity = makeEntity(newLocation);

    Location returnedLocation = null;
    returnedLocation = restTemplate.postForObject(API_BASE_URL, entity,
Location.class);
    return returnedLocation;
}
```

Step Four: Modify a location with PUT

You'll modify the `update()` method next. This method is invoked similarly to the `add()` method. It's passed an existing location modified by the user rather than a brand new location, and it returns a `boolean` value indicating if it was successful or not. There's no need for it to return an object, since the returned object would be identical to the one that's passed in.

This code is similar to the code you added in the `add()` method. The first line is the same—make the `HttpEntity` using the `updatedLocation` variable that's passed in:

```
public boolean update(Location updatedLocation) {
    HttpEntity<Location> entity = makeEntity(updatedLocation);

    return false;
}
```

The difference is in the use of `RestTemplate`. To update a record, you'll use the HTTP `PUT` method and append the `id` of the location to update to the URL. Since you have a `Location` object, you'll use `updatedLocation.getId()` to retrieve the `id` to append to the URL.

The `RestTemplate.put()` method takes the URL with `id` and the Location object containing the updates. It doesn't return anything, so don't make an assignment statement here. Add this code:

```
restTemplate.put(API_BASE_URL + updatedLocation.getId(), entity);
```

Next, change the `return false` statement at the end of the method to `return true`. You'll do more with the return value in Step Seven.

The complete method looks like this:

```
public boolean update(Location updatedLocation) {
    HttpEntity<Location> entity = makeEntity(updatedLocation);

    restTemplate.put(API_BASE_URL + updatedLocation.getId(), entity);

    return true;
}
```

Step Five: Delete a location with DELETE

To delete a location, you only need to send the `id` of the location to delete. So, only the `id` of the location to delete is passed into the `delete()` method rather than a complete `Location` object.

Inside the `delete()` method, you'll make one call to `restTemplate.delete()`. This method call takes the URL with the `id` appended to it and returns nothing. Add this code to the `delete()` method:

```
restTemplate.delete(API_BASE_URL + id);
```

Next, change the `return false` statement at the end of the method to `return true`. You'll do more with the return value in Step Seven.

The complete method now looks like this:

```
```

```
public boolean delete(int id) {
    restTemplate.delete(API_BASE_URL + id);
    return true;
}
```

Step Six: Test your application

Run the application and execute each menu item. If you followed the instructions, the application works as expected. If you encounter any issues, go back and review the previous steps.

From the main menu, select option 5. When prompted to select a location, enter an invalid number like 999 and observe the result. The program stops because of an error that was returned from the server.

Step Seven: Add exception handling for HTTP errors

Next you'll capture and log the errors sent back from the server and prevent the application from crashing. To do this, you'll use `try/catch` blocks. `RestTemplate` throws a `RestClientResponseException` when an error response code is received or a `ResourceAccessException` when no response is received at all. You'll `catch` those exceptions and log them by calling `BasicLogger.log()`.

Inside the `catch` block, you'll use the exception methods `getRawStatusCode()`, `getStatusText()`, and `getMessage()` to get more detailed information about what happened, and include it in the string sent to the log. You'll also make sure the method returns a `null` or `false` value, to communicate the failure to the caller. The `try/catch` block pattern looks like this:

```
try {
    //Call to RestTemplate goes here
} catch (RestClientResponseException ex) {
    BasicLogger.log(ex.getRawStatusCode() + " : " + ex.getStatusText());
} catch (ResourceAccessException ex) {
    BasicLogger.log(ex.getMessage());
}
```

You can see an example of this exception handling in the `getAll()` method of `LocationService` :

```
public Location[] getAll() {
    Location[] locations = null;
    try {
        locations = restTemplate.getForObject(API_BASE_URL, Location[].class);
    } catch (RestClientResponseException ex) {
        BasicLogger.log(ex.getRawStatusCode() + " : " + ex.getStatusText());
    } catch (ResourceAccessException ex) {
        BasicLogger.log(ex.getMessage());
    }
    return locations;
}
```

Apply the same exception handling pattern to the `add()` method you wrote:

```
public Location add(Location newLocation) {
    HttpEntity<Location> entity = makeEntity(newLocation);

    Location returnedLocation = null;
    try {
        returnedLocation = restTemplate.postForObject(API_BASE_URL, entity,
Location.class);
    } catch (RestClientResponseException ex) {
        BasicLogger.log(ex.getRawStatusCode() + " : " + ex.getStatusText());
    } catch (ResourceAccessException ex) {
        BasicLogger.log(ex.getMessage());
    }
    return returnedLocation;
}
```

Next, apply it to the `update()` method:

```
public boolean update(Location updatedLocation) {
    HttpEntity<Location> entity = makeEntity(updatedLocation);

    boolean success = false;
    try {
        restTemplate.put(API_BASE_URL + updatedLocation.getId(), entity);
        success = true;
    } catch (RestClientResponseException ex) {
        BasicLogger.log(ex.getRawStatusCode() + " : " + ex.getStatusText());
    } catch (ResourceAccessException ex) {
        BasicLogger.log(ex.getMessage());
    }
    return success;
}
```

Note that in this case, a new `boolean` variable called `success` was also added. It's initially set to `false`, and only changed to `true` after the call to `restTemplate.put()`. If an exception is thrown, it never gets changed to `true`, so the method returns `false`, indicating failure.

Finally, apply the same exception handling pattern to the `delete()` method:

```
public boolean delete(int id) {
    boolean success = false;
    try {
        restTemplate.delete(API_BASE_URL + id);
        success = true;
    } catch (RestClientResponseException ex) {
        BasicLogger.log(ex.getRawStatusCode() + " : " + ex.getStatusText());
    } catch (ResourceAccessException ex) {
```

```
            BasicLogger.log(ex.getMessage());
        }
        return success;
    }
```

After making those changes, rerun the program, select menu option 5, and enter 999. You'll receive a brief error message, and see a file containing more details in the `logs` folder. The application continues to run.

Summary

In this tutorial, you learned how to:

- Use the HTTP `POST` Web API call to add a new Location
- Use the HTTP `PUT` Web API call to modify a new Location
- Use the HTTP `DELETE` Web API call to delete a new Location
- Use exception handling for HTTP errors

Don't forget to stop json-server

When you're done with the tutorial, remember to stop json-server. Directions are under Step One.