Module 2 Day 14 Lecture 12

Server Side APIs

Part 2

Module 2 Day 14 Lecture 11
Can you / Do you ?

1. Understand the high-level architectural elements of the REST architecture
    a. Explain how HTTP methods are used to interact with RESTful resources
    b. Conform to RESTful conventions when using HTTP status codes
2. Define CRUD and how it relates to HTTP methods and APIs
3. Handle errors from backend code and alert the frontend that something has gone wrong using HTTP response codes
4. Perform validation on client supplied request data
5. Implement a RESTful web service that provides full CRUD functionality

Server Side Validation

We can add special bits of annotation code to our classes to ensure that the data is consistent and free of errors. Some examples of these:

- In an Automobile class, an attribute measuring fuel tank capacity must be between 0 and 20 gallons.
- For a Hotel reservation class, a begin date or end date must be provided.
- For a Customer class, a value must be provided for the customer name.

The goal is to implement these validation rules right on the classes that contain the fields that require validation.

Validation Annotations in Spring

Here is a list of common Spring validation annotations:

- **@NotBlank(“message”)**: Will check if a field is blank or not.
- **@Email(“message”)**: Verify in an input conforms to an email format.
- **@Min(value=<<x>>, message = “message”)**: A form must have a minimum input value, where <<x>> is that number.
- **@Max(value=<<y>>, message = “message”)**: A form must have a maximum input value, where <<y>> is that number.
- **@Pattern(regexp= “<<z>>” , message=“message”)**: A form’s value must conform to a regular expression, where <<z>> is that expression enclosed in double quotes.

Server Side Validation Example

Consider the following code:

```
public class Reservation {

    private int id;
    @Min( value = 1, message = "The field 'hotelID' is required.")
    // a value must be provided for a hotel id:
    private int hotelID;
    @NotBlank( message = "The field 'fullName' is required.")
    // a value must be provided for the guest name
    private String fullName;
    @NotBlank( message = "The field 'checkinDate' is required.")
    private String checkinDate;
    @NotBlank( message = "The field 'checkoutDate' is required.")
    private String checkoutDate;
… }
```

Let’s implement some validation rules

PUT Requests

- A PUT request is used to update existing data.
- Like a POST, PUT requires a body containing the updated JSON Object:
- The validation techniques we learned with POST sill apply!
- Consider the following example:

```
    @RequestMapping(path = "/reservations/{id}", method = RequestMethod.PUT)
    public Reservation update(@Valid @RequestBody Reservation reservation, @PathVariable int id) throws ReservationNotFoundException
{
        return reservationDAO.update(reservation, id);
    }
```

DELETE Requests

- A DELETE request is used to remove data.

```
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.DELETE)
public void delete(@PathVariable int id) throws ReservationNotFoundException {
    reservationDAO.delete(id);
}
```

Let’s implement the PUT & DELETE requests

Summary of Request Types

We have now covered the four basic persistent data storage operations. This is commonly referred to as CRUD (**C**reate, **R**ead, **U**pdate, and **D**elete).

Modulating the Response Code

Finally, Spring gives us the ability to tweak the response code a user receives. To review, recall the status code ranges:

- **2XX** : Everything is fine.
- **4XX** : There is a client side problem, something is wrong with your request.
- **5XX**: There is a server side problem

Common examples of each:

- **200**: Success
- **401**: Your request contains bad credentials.
- **500**: Internal Server Error

Modulating the Response Code

We can provide slightly more descriptive codes by using the @ResponseStatus annotations:

```
@ResponseStatus(HttpStatus.CREATED)
@RequestMapping( path = "/hotels/{id}/reservations", method = RequestMethod.POST)
public Reservation addReservation(...) {
...
}
```

Provided the request completed without issue, the response back to the API user will now be 201 instead of 200.

Let’s add in @ResponseStatus

Consuming UPDATE and DELETE

Let’s bring things full circle and review how to consume an UPDATE or DELETE endpoint from another application (not Postman!)

Consuming an Update

Here is the Server API endpoint:

```
@RequestMapping(path = "/reservations/{id}", method = RequestMethod.PUT)
public Reservation update(@Valid @RequestBody Reservation reservation, @PathVariable int id) throws ReservationNotFoundException   {
   return reservationDAO.update(reservation, id);
}
```

… and here is how another Java application can consume the endpoint above:

```
HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_JSON);
HttpEntity<Reservation> entity = new HttpEntity<>(reservation, headers);
restTemplate.put(BASE_URL + "reservations/" + reservation.getId(), entity);
```

Consuming a Delete

Here is the Server API endpoint:

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

… and here is how another Java application can consume the endpoint above:

```
restTemplate.delete(BASE_URL + "reservations/" + id);
```

Let’s revisit the client side app