Spring Remoting

Practical Guide for Part 3 - REST

Table of Contents

Requirements

Please watch the Chapter 28 video first. I will take you through how to set up dates in the correct format (needed for this exercise) and I'll also run through the requirements.

The requirements are quite similar to the part 1 and 2 practicals: we are going to be exposing the "recordCall" business process and "getAllActionsForUser" to clients.

The difficulty is there are many ways of approaching this as REST is more open than SOAP. So you can go your own way (but the walkthrough might not help if you have a very different strategy).

Or, you can follow the steps in this guide. I'll keep this high-level so that you've still got plenty of thinking to do, but by following this structure you'll be doing similar work to the walkthrough in chapter 28.

> **Note**: we are not going to code a standalone client using the RESTTemplate, because the focus of the course is server side development, and I feel that the client will take a long time to code without adding value (you have to code representation classes for the client that will be nearly identical to the ones on the server). We're going to test the server code using the REST shell.

Step 1: Design the URI Scheme for Record Call

This is at approximately 41m30s in the walkthrough.

We're going to support:

- **POST /customer/{id}/calls**
  (this will map to "recordCall") and will take a representation in the format as in the data.json file we supplied
- **GET /customer/{id}/calls**
  (this will return all calls for that customer).

But we are NOT going to support: GET /customer/{id}/call/{id} or GET /call/{id}

The real reason for this is we don't have a "getCallById" feature on the server. Of course, you could add one but this will add extra work which isn't related to REST. So I'm going to claim this requirement has been decided on because we don't want to expose individual calls to clients, I don't want clients guessing IDs and randomly grabbing them; they MUST get the data through a customer representation.

Some people will claim this isn't restful because we aren't providing full access to clients. I claim that this is an API design issue and it's perfectly restful. Effectively, we are saying that call isn't a representation; it's more like a "field" of customer. Or put it another way, customers and calls are so tightly bound together that we're going to treat them as a single representation, always.

Step 2: Code the Controller Method

This is at approximately 42m30s in the walkthrough.

In the RESTController, code a method to handle the POST /customer/{id}/calls. To support this, you will need to write a representation of the Call/Action combination. On the walkthrough, mine looks something like this:

```
@XmlRootElement
public class CallActionRepresentation
{
     private Call call;
     private Collection<Action> actions;
     public CallActionRepresentation() {}
     // plus gets/sets
}
```

Note: I didn't give the XmlRootElement a name in this session, so this means that clients who want to send XML will need to build an xml document like this:

```
<callActionRepresentation>
   <call>
   </call>

   <actions>
   </actions>
</callActionRepresentation>
```

Of course, you can provide a name if you wish to make the XML "friendlier", but you can't use the name "call" as this will clash with the child element. You'd need namespaces to avoid this, but maybe "callReport" would have been a better outer tag name. I wish I had done this on the video!

Step 3: Test

This is at approximately 50m in the walkthrough.

Test with your REST Shell. Be very careful to check the output in the rest shell log file.

```
post customer/100029/calls --from ./data.json
```

Step 4: Getting all Calls

This is at approximately 50m in the walkthrough.

In the RESTController, code a method to handle the GET /customer/{id}/calls and test this in the browser using both XML and JSON. Check that your timestamps come back correctly.

(Note: the timestamps are sent in the representation; don't expect them to change every time you run the test.)

You will need a representation class to handle the whole collection.

Step 5: Returning all Actions for a User

This is at approximately 59mins in the walkthrough.

I decided to design a URI scheme that makes the user become a representation:

- **/user/{username}/actions**

This may surprise - on our server the action is a class, and there is no class for user. But my thinking is that a client is more likely to think in terms of users, and users having actions, than the other way around.

I could have perfectly well decided on

/actions/{username}
or
/actions?username={username}

But to me both of these feel slightly more awkward. It's a perfectly valid option however, and this is a great example of how you need to think things through in REST.

Code this up; again you will need a representation class for the Action collection.

Step 6: Adding Validation

This is at approximately 65mins in the walkthrough.

The CallActionRepresentation class has two properties, a Call object and a collection of Action objects. We need to apply validation to both of these properties, but the actual validation rules will be inside these child objects. How do we do that?

Easy - the annotation to add to the property is @Valid:

```
@Valid
private Call call;
```

This tells the validator it must dig into the call class and check the validation rules inside there. You can do the same for the action collection, and JSR303 is smart enough to work with a collection - of course it will check that every element in the collection is valid.

For validating Date and Calendar, you can use the **NotNull** constraint. If an invalid date or an empty string is sent from the client, the conversion to Date or Calendar will fail and this will leave a null in the field. So **NotNull** will catch any invalid or missing dates.

Step 7: (Optional): Sending Full or Truncated Customers

This is at approximately 1hr 12mins in the walkthrough.

There are different ways of doing this, have a think and if you're stuck, check the video for details of how I handled this.

Watch out for LazyInitialization exceptions here - the strategy for handling this will be as on the theory sessions, but if you need help, check the walkthrough at around 78mins.

Step 8: (Optional): HATEOAS

This is at approximately 1hr 19mins in the walkthrough.

We should sent back a link to the created resource, but for a call report, there are multiple resources created - a new call and several actions.

As individual calls and actions don't have URIs, we can't return back the URI of each action, but we can return back the collection URIs. So we need to return back

- **/customer/{id}/calls** - but this is not very useful to the client because it is actually the same URI that they have just posted to.

and

- /**user/{username}/actions** (this is VERY useful to the client because they now know what URI to send a GET to in order to get the created actions).

But we can't return multiple URIs in the Location: header. What can we do? Thankfully the HTTP spec is fairly clear about this. I'll go into more detail on the

video, but the spec in section 6.3.2 (http://tools.ietf.org/html/rfc7231#section-6.3.2) says:

> *"The 201 response payload typically describes and links to the resource(s) created."*

So, on the video I will show how to return HTTP 201, with a body containing the two URIs.

On the video, I will just return the actions for the user identified in the first action. However, a call can have multiple actions, with different users. So to implement this properly, you need to loop around each action and check if you've already built a link for this user. Easy Java so I won't slow the video down with that.

When testing, ensure that in the rest shell, your URIs do NOT start with a /, otherwise you will get mangled links:

Example REST Shell command line:

```
post customer/100029/calls --from ./data.json
```

Step 9: One Last Link!

This is at approximately 1hr 28mins in the walkthrough.

When a new customer is created, it would be good to tell the client the URI to the calls, so they know what to POST to when recording a call.

So, in "createNewCustomer", return a link to the calls, something like:

```
{"rel":"calls","href":"/customer/{id}/calls"}
```

This will mean that once a new customer has been POSTed, the client will know exactly the URI for the calls, so they can start issuing POSTs. This is the sprit of HATEOAS, although I won't go any further on the walkthrough!