Spring MVC Tutorial – Contact List Application

Step 08: Implementing CRUD REST Controller Endpoints

SOFTWARE GUILD

The Learning House

427 S 4th Street #300

Louisville KY 40202

Step 08: Implementing CRUD REST Controller Endpoints

Overview

In this step of the tutorial, we will implement the controller endpoints for creating, retrieving, updating, and deleting Contacts in the system. We will build these endpoints using an architectural style called Representational State Transfer or REST, which is a lightweight approach to building web services over HTTP. Our endpoints will be called from our view components via Ajax calls. The endpoints will return data in JavaScript Object Notation (JSON), which will be parsed by the client and rendered in HTML.

What is REST?

As mentioned in the Overview, REST is an architectural approach to building lightweight web services. REST is not a standard per se; rather, it is an approach or style for building networked applications. A RESTful application consists of one or more endpoints (i.e. URLs) that map to functionality in the application, and these endpoints comprise the API of the application. Each of these endpoint URLs will also be mapped to one of the HTTP verbs — we will be using GET, POST, PUT, and DELETE. Our REST endpoints will return JSON data rather than HTML content. The JSON data will not be consumed directly by the browser; instead, it will be consumed by custom JavaScript code that we will write.

Designing Our URLs

Before we begin writing code for our controller, we must design the URLs that will make up the REST API for our Contact List application.

One thing to keep in mind about REST endpoints is that they refer to resources; in other words, they are noun-driven instead of verb-driven. The HTTP verbs tell us what action to take with respect to the resource (noun) specified in the URL. This is a different approach than we have been using until now — so far, the endpoints in our applications have been methods which are action- or verb-oriented.

Another difference between REST URLs and regular web application URLs is that REST URLs pass parameters as part of the URL path rather than as form data or request parameters (i.e. ?key=value&key1=value1). For example, a URL for retrieving the Contact with id=42 might look like this in a regular web application URL: www.mysite.com/displayContact?contactId=42

The RESTful URL might look like this: www.mysite.com/contact/42

Below are the descriptions of the URLs for retrieving, creating, deleting, updating, and listing all Contacts in our application:

Retrieving a Contact

Verb: GET

URL: /contact/{contactId}

RequestBody: None

ResponseBody: JSON object containing the requested Contact data

Notes: This basic URL (/contact) will be used for getting, adding, deleting, and updating Contacts in the system. The only difference between these actions will be the HTTP verb used for each. This request has no RequestBody because everything the server needs to service the request (i.e. the contactId) is in the URL.

Creating a Contact

Verb: POST

URL: /contact

RequestBody: JSON object containing the Contact data

ResponseBody: JSON object containing all of the original Contact data plus the contactId that the DAO assigned to the newly-created Contact.

Notes: There is no contactId in the path because a Contact does not have an id until it is persisted by the DAO.

Deleting a Contact

Verb: DELETE

URL: /contact/{contactId}

RequestBody: None

ResponseBody: None

Notes: This URL is similar to the URL for retrieving a contact. There is no RequestBody because the server only needs the contactId and no ResponseBody because we are not returning anything to the caller.

Updating a Contact

Verb: PUT

URL: /contact/{contactId}

RequestBody: JSON object containing the Contact data. Because we are updating an existing Contact, the JSON object will include the contactId.

ResponseBody: None

Notes: There is no need for a ResponseBody because the JSON object in the RequestBody contains all of the changes to the Contact — we are simply persisting those changes.

Retrieving All Contacts

Verb: GET

URL: /contacts

RequestBody: None

ResponseBody: An array of JSON Contact objects.

Notes: Notice that the URL for this endpoint is /contacts (plural).

Controller Implementation

Now, we'll move on to the implementation of the controller. In this section, we will use annotations and constructor injection to wire the DAO into our controller; we will also create the endpoints described in the previous section.

DAO Injection

Wiring the DAO dependency into our controller requires no additional XML because we are using annotations to configure our MVC components. We can use either the @Inject or @Autowired annotations (they are synonymous) to inject a bean defined in the Spring configuration file into our controller. Since we defined the bean for our DAO in a previous step, we are ready for the annotation. Add a private, class-level ContactListDao variable and a new constructor to your HomeController as in the code below:

```
// The controller uses the DAO to do all the heavy lifting of storing
// and retrieving Contacts
private ContactListDao dao;

// @Inject and @Autowired are synonyms
// This annotation tells the Spring Framework to hand an object of type
// ContactListDao to this constructor when it creates an instance of this
// class (which happens when the web application starts).  If there is no
// object of type ContactListDao defined in the Spring application context,
// Spring Framework will throw an exception.
@Inject
public HomeController(ContactListDao dao) {
    this.dao = dao;
}
```

Retrieving a Contact

Here, we will add the endpoint for retrieving a Contact. Add the following method to your HomeController:

```
    // This method will be invoked by Spring MVC when it sees a GET request for
    // ContactListMVC/contact/<some-contact-id>.  It retrieves the Contact
    // associated with the given contact id (or null if no such Contact
    // exists).
    //
    // @ResponseBody indicates that the object returned by this method should
    // be put in the body of the response going back to the caller.
    //
    // @PathVariable indicates that the portion of the URL path marked by curly
    // braces {...} should be stripped out, converted to an int and passed into
    // this method when it is invoked.
    @RequestMapping(value="/contact/{id}", method=RequestMethod.GET)
    @ResponseBody public Contact getContact(@PathVariable("id") int id) {
        // Retrieve the Contact associated with the given id and return it
        return dao.getContactById(id);
    }
```

Creating a Contact

Add the following method to your HomeController. This endpoint creates a new Contact in the application:

```
// This method will be invoked by Spring MVC when it sees a POST request for
// ContactListMVC/contact.  It persists the given Contact to the data layer.
//
// @ResponseStatus tells Spring MVC to return an HTTP CREATED status upon success
//
// @ResponseBody indicates that the object returned by this method should
// be put in the body of the response going back to the caller.
//
// @RequestBody indicates that we expect a Contact object
// in the body of the incoming request.
@RequestMapping(value="/contact", method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
@ResponseBody public Contact createContact(@Valid @RequestBody Contact contact) {
    // persist the incoming contact
    dao.addContact(contact);
    // The addContact call to the dao assigned a contactId to the incoming
    // Contact and set that value on the object.  Now we return the updated
    // object to the caller.
    return contact;
}
```

Deleting a Contact

Add the following method to your HomeController. This endpoint deletes the specified Contact:

```
// This method will be invoked by Spring MVC when it sees a DELETE request
// for ContactListMVC/contact/<some-contact-id>.  It deletes the Contact
// associated with the give id from the data layer (it does nothing if there
// is no such Contact).
//
// @ResponseStatus tells Spring MVC to return HTTP NO_CONTENT from this call
// because this method has no return value.
//
// @PathVariable indicates that the portion of the URL path marked by curly
// braces {...} should be stripped out, converted to an int and passed into
// this method when it is invoked.
@RequestMapping(value="/contact/{id}", method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteContact(@PathVariable("id") int id) {
    // remove the Contact associated with the given id from the data layer
    dao.removeContact(id);
}
```

Updating a Contact

Add the following code to your Home Controller. This endpoint updates a Contact in the application:

```
// This method will be invoked by Spring MVC when it sees a PUT request for
// ContactListMVC/contact/<some-contact-id>.  It updates the given Contact
// to the data layer.
//
// @ResponseStatus tells Spring MVC to return HTTP NO_CONTENT from this call
// because this method has no return value.
//
// @PathVariable indicates that the portion of the URL path marked by curly
// braces {...} should be stripped out, converted to an int and passed into
// this method when it is invoked.
//
// @RequestBody indicates that we expect a Contact object in the body of the
// incoming request.
@RequestMapping(value="/contact/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void putContact(@PathVariable("id") int id, @RequestBody Contact contact) {
    // set the value of the PathVariable id on the incoming Contact object
    // to ensure that a) the contact id is set on the object and b) that
    // the value of the PathVariable id and the Contact object id are the
    // same.
    contact.setContactId(id);
    // update the contact
    dao.updateContact(contact);
}
```

Retrieving All Contacts

Add the following method to your HomeController. This endpoint retrieves all of the Contacts stored in the application:

```
// This method will be invoked by Spring MVC when it sees a GET request for
// ContactListMVC/contacts.  It retrieves all of the Contacts from the
// data layer and returns them in a List.
//
// @ResponseBody indicates that the List returned by this method should
// be put in the body of the response going back to the caller.
@RequestMapping(value="/contacts", method=RequestMethod.GET)
@ResponseBody public List<Contact> getAllContacts() {
    // get all of the Contacts from the data layer
    return dao.getAllContacts();
}
```

Wrap-up

That's it for this part of the tutorial. In this step, we did the following:

1. Learned how to wire dependencies into a controller with annotations
2. Learned about REST
3. Designed REST endpoint URLs for our application
4. Implemented the REST endpoints in our controller

In the next step, we will see how to wire these endpoints into our view components.