Thinking about IT

Examples, solutions, recipes and thoughts about IT

23 June, 2013

HATEOAS using Spring Framework

HATEOAS is a constraint of the REST application architecture which is very often ignored by developers. In many cases this is caused by the lack of support of frameworks used to build and expose RESTful services. In this post I will show how quickly we can add HATEOAS to RESTful web services using Spring Framework. Source code of the demo web application presented below are available for download, run it using `mvn tomcat7:run` command. Also you can read a nice presentation about why HATEOAS is important.

The Spring HATEOAS Library (currently in Release 0.6) provides API to help creating REST representations that follow the HATEOAS principle. The core problem it tries to address is link creation and representation assembly. Let's see the main steps required for enabling the HATEOAS for a RESTful web service:

1. **Add support for hypermedia information to exposed resources.** This is done by inheriting resource's classes from `ResourceSupport`. As result you get the support for adding `Link`(s) to the resources. Below you can see the resource class representing an author:

```
import org.springframework.hateoas.ResourceSupport;

public class AuthorResource extends ResourceSupport {
    private int authorId;
    private String name;

    public AuthorResource() {
    }

    public AuthorResource(int authorId, String name) {
        this.authorId = authorId;
        this.name = name;
    }

    public int getAuthorId() {
        return authorId;
    }
    public void setAuthorId(int authorId) {
        this.authorId = authorId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

and this is an example of adding a link to it:

```
AuthorResource resource = new AuthorResource(123, "Joshua Bloch");
resource.add(new Link("http://localhost:8080/hateoas-demo/authors/123"));
```

The `Link` value object follows the Atom link definition and consists of a `rel` and an `href` attribute.

2. **Building links.** Spring Hateoas provides a `ControllerLinkBuilder` that allows to create links by pointing to controller classes:

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

resource.add(linkTo(AuthorController.class).slash(author.getAuthorId()).slash("books").withRel("books"));

// or by pointing directly to a controller method
resource.add(linkTo(methodOn(AuthorController.class).getAuthorBooks(author.getAuthorId())).withRel("books
```

The builder inspects the given controller class for its root mapping and it frees developer from ugly manual string concatenation code (protocol, hostname, port, servlet base, etc.).

3. **Encapsulate resource creation in a separate class.** Spring Hateoas provides a `ResourceAssemblerSupport` base class that helps reducing the amount of code needed to be written for mapping from an entity to a resource type and adding respective links. The assembler can then be used to either assemble a single resource or an `Iterable` of them. You can see below the resource assembler for author resource:

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import org.springframework.stereotype.Component;

@Component
```

```
public class AuthorResourceAssembler extends ResourceAssemblerSupport<Author, AuthorResource> {
    public AuthorResourceAssembler() {
        super(AuthorController.class, AuthorResource.class);
    }

    @Override
    public AuthorResource toResource(Author author) {
        // will add also a link with rel self pointing itself
        AuthorResource resource = createResourceWithId(author.getAuthorId(), author);
        // adding a link with rel books pointing to the author's books
        resource.add(linkTo(methodOn(AuthorController.class).getAuthorBooks(author.getAuthorId())).withR
        return resource;
    }

    @Override
    protected AuthorResource instantiateResource(Author author) {
        return new AuthorResource(author.getAuthorId(), author.getName());
    }
}
```

4. **Exposing resources.** This is achieved by writing the actual controller. Nothing special here. Just invoke the business logic services (in our case `BookRepository`) and map the business entities to their resource representations using resource assemblers:

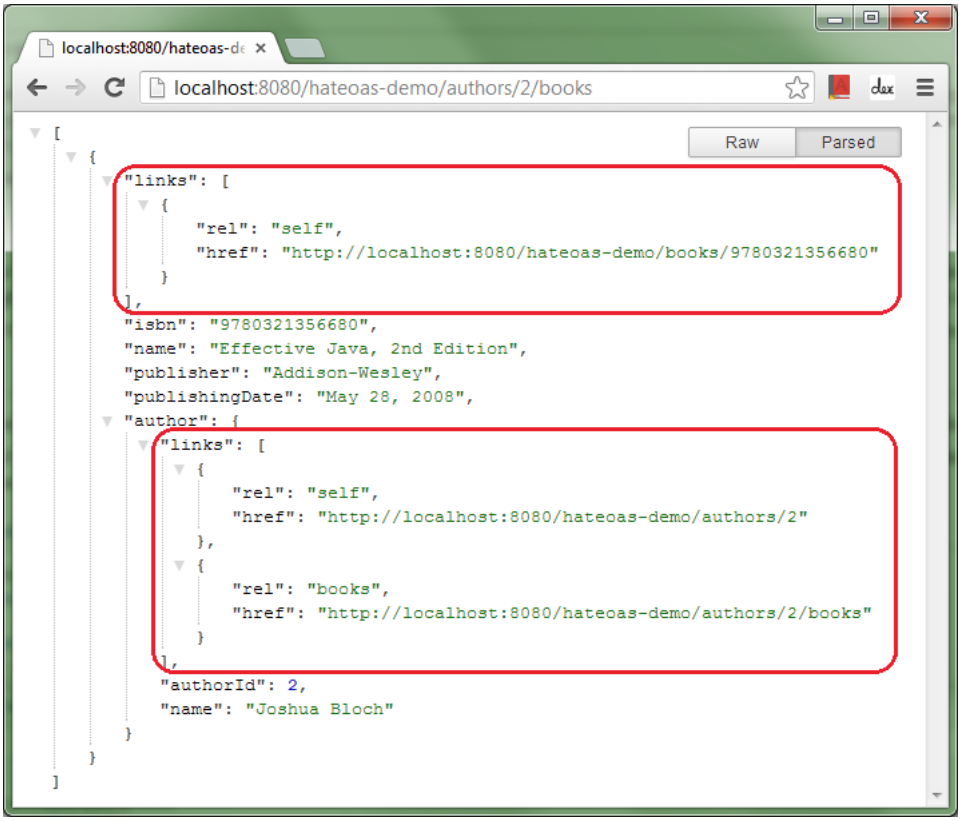
```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/authors")
public class AuthorController extends AbstractController {
    @Autowired
    private BookRepository bookRepository;
    @Autowired
    private AuthorResourceAssembler authorResourceAssembler;
    @Autowired
    private BookResourceAssembler bookResourceAssembler;

    @RequestMapping(method = RequestMethod.POST)
    @ResponseBody
    public ResponseEntity<Void> createNewAuthor(@RequestBody NewAuthor newAuthor) {
        Author author = bookRepository.createNewAuthor(newAuthor.getName());
        HttpHeaders headers = new HttpHeaders();
        headers.setLocation(linkTo(methodOn(getClass()).getAuthor(author.getAuthorId())).toUri());
        return new ResponseEntity<Void>(headers, HttpStatus.CREATED);
    }

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    public List<AuthorResource> getAuthors() {
        return authorResourceAssembler.toResources(bookRepository.findAuthors());
    }

    @RequestMapping(value = "/{authorId}", method = RequestMethod.GET)
    @ResponseBody
    public AuthorResource getAuthor(@PathVariable("authorId") int authorId) {
        return authorResourceAssembler.toResource(findAuthorAndValidate(authorId));
    }

    @RequestMapping(value = "/{authorId}/books", method = RequestMethod.GET)
    @ResponseBody
    public List<BookResource> getAuthorBooks(@PathVariable("authorId") int authorId) {
        return bookResourceAssembler.toResources(findAuthorAndValidate(authorId).getBooks());
    }

    private Author findAuthorAndValidate(int authorId) {
        Author author = bookRepository.findAuthor(authorId);
        if (author == null) {
            throw new ResourceNotFoundException("Unable to find author with id=" + authorId);
        }
        return author;
    }
}
```

5. **Testing.** Let's eat our own dog food:

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.util.List;

import org.junit.Ignore;
import org.junit.Test;
import org.springframework.hateoas.Link;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class AuthorControllerTest {
```

```
14      private static final String BASE_URI = "http://localhost:8080/hateoas-demo";
15
16      @Test
17      public void createNewAuthor() {
18          RestTemplate restTemplate = new RestTemplate();
19
20          // creating new author
21          NewAuthor newAuthor = new NewAuthor("Brian Goetz");
22          ResponseEntity<Void> response = restTemplate.postForEntity(BASE_URI + "/authors", newAuthor, Voi
23          assertEquals(HttpStatus.CREATED, response.getStatusCode());
24
25          // retrieving the newly created author details using the URI received in Location header
26          AuthorResource author = restTemplate.getForObject(response.getHeaders().getLocation(), AuthorRes
27          assertEquals(newAuthor.getName(), author.getName());
28          assertTrue(author.getAuthorId() > 0);
29
30          // getting the author's books using the link with rel books
31          Link authorBooksLink = author.getLink("books");
32          List<BookResource> authorBooks = restTemplate.getForObject(authorBooksLink.getHref(), List.class
33          assertTrue(authorBooks.isEmpty());
34      }
35  }
```

6. **How it looks like:** (output formatted using JSON Formatter Chrome extension)



7. **That's it, hyperlink it!**

Posted by Andrei Zagorneanu at 18:37

Labels: java, rest, spring framework

Reactions: util (1) interesting (3) funny (2) cool (1) 50/50 (2)

7 comments:

robpatrick 17 July, 2013 17:23

We're using MockMvc to test our services rather that using restTemplate:

```
@Autowired
private WebApplicationContext ctx

private MockMvc mockMvc

def setup( ) {
mockMvc = MockMvcBuilders.webAppContextSetup( ctx ).build()
}

...
```

```
String result = mockMvc.perform( get( "/people/1" ).accept( PersonServiceMediaType.PERSONV1JSON ) )
.andDo( print() )
.andExpect( status().isOk() )
.andExpect( content().contentType( PersonServiceMediaType.PERSONV1JSON ) )
.andReturn().response.contentAsString

def parsedResult = JSONValue.parse( result )

assert parsedResult.forename == 'Hugh'
assert parsedResult.dateOfBirth == '1977-03-29'
```

Reply

Replies

Andrei Zagorneanu 17 July, 2013 18:12

Yes, you can test them using MockMvc also. We are using MockMvc in our integration tests and RestTemplate in our acceptance tests.

Reply

Vijayanand Bharadwaj 29 August, 2013 01:41

Nice article. Very clear and concise. Look forward to trying your example.

Reply

Replies

Andrei Zagorneanu 01 September, 2013 21:58

Thanks.

Reply

Kass 21 September, 2013 14:31

nice job! simple and to the point.

Reply

Replies

Andrei Zagorneanu 21 September, 2013 19:26

Thanks!

Reply

Vishwanath Poodari 26 February, 2014 23:31

Hi,
I read your article.Its clean and precise. I have a question in connection to this.
Please see the below Response

```
{"enrollment": {"optionId": "AETNA"
, "actions": [{"method": "PUT", "uri": "/12345/54321/processes/111222/
, "dependentParticipation":[
{"dependentId": "1001", "enrolled":true ,"effectiveStartDate":"2010/04/04", "effectiveEndDate":"999/12/31"}
,{"dependentId": "1002", "enrolled":true ,"effectiveStartDate":"2010/04/04", "effectiveEndDate":"999/12/31"}
]}
}
```

I am able to get the above output without the actions. Can i get that actions inserted in the output without creating an actions object.

Reply

Enter your comment...

Comment as: Google Accoun

Publish Preview

Links to this post

Create a Link

Home Older Post

Subscribe to: Post Comments (Atom)