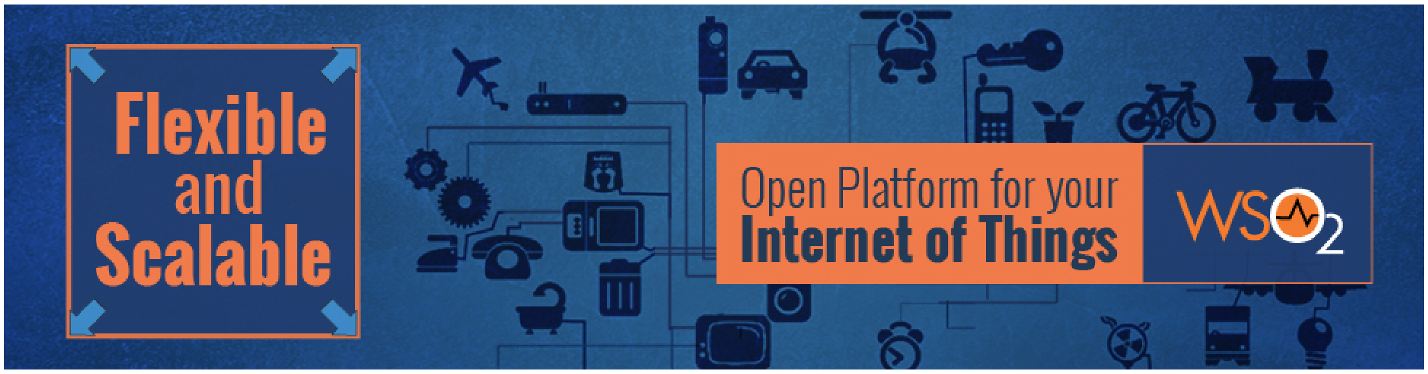

MVB GERAINT JONES Bio Website @city81limited

Spring MVC and the HATEOAS constraint

05.10.2013 | 10504 VIEWS | Like 1 Tweet 4 +1 2 SHARE 1

The Enterprise Integration Zone is brought to you in partnership with WSO2. Learn more about WSO2's API Management.

HATEOAS is a REST architecture principle where hypermedia is used to change application state. To change state, the returned resource representation contains links thereby 'constraining' the client on what steps to take next.

The Spring-HATEOAS project aims to assist those writing Spring MVC code in the creation of such links and the assembly of resources returned to the clients.

The example below will cover a simple scenario showing how links are created and returned for the resource, *Bet*. Each operation on the resource is described below:

- *createBet* - this *POST* operation will create a Bet.
- *updateBet* - this *PUT* operation will update the Bet.
- *getBet* - this *GET* operation will retrieve a Bet.
- *cancelBet* - this *DELETE* operation will cancel the Bet.

```
@Controller
@RequestMapping("/bets")
public class BetController {

 private BetService betService;
 private BetResourceAssembler betResourceAssembler;

 public BetController(BetService betService,
  BetResourceAssembler betResourceAssembler) {
  this.betService = betService;
  this.betResourceAssembler = betResourceAssembler;
 }

 @RequestMapping(method = RequestMethod.POST)
```

```
ResponseEntity<BetResource> createBet(@RequestBody Bet body) {
 Bet bet = betService.createBet(body.getMarketId(),
  body.getSelectionId(), body.getPrice(), body.getStake(),
  body.getType());
 BetResource resource = betResourceAssembler.toResource(bet);
 return new ResponseEntity<BetResource>(resource,
    HttpStatus.CREATED);
}

@RequestMapping(method = RequestMethod.PUT, value = "/{betId}")
ResponseEntity<BetResource> updateBet(@PathVariable Long betId,
 @RequestBody Bet body) throws BetNotFoundException,
    BetNotUnmatchedException {
 Bet bet = betService.updateBet(betId, body);
 BetResource resource = betResourceAssembler.toResource(bet);
 return new ResponseEntity<BetResource>(resource, HttpStatus.OK);
}

@RequestMapping(method = RequestMethod.GET, value = "/{betId}")
ResponseEntity<BetResource> getBet(@PathVariable Long betId) throws
   BetNotFoundException {
 Bet bet = betService.getBet(betId);
 BetResource resource = betResourceAssembler.toResource(bet);
 if (bet.getStatus() == BetStatus.UNMATCHED) {
  resource.add(linkTo(BetController.class).slash(bet.getId()).withRel
 }
 return new ResponseEntity<BetResource>(resource, HttpStatus.OK);
}

@RequestMapping(method = RequestMethod.GET)
ResponseEntity<List<BetResource>> getBets() {
 List<Bet> betList = betService.getAllBets();
 List<BetResource> resourceList =
    betResourceAssembler.toResources(betList);
 return new ResponseEntity<List<BetResource>>(resourceList,
    HttpStatus.OK);
}

@RequestMapping(method = RequestMethod.DELETE, value = "/{betId}")
ResponseEntity<BetResource> cancelBet(@PathVariable Long betId) {
 Bet bet = betService.cancelBet(betId);
 BetResource resource = betResourceAssembler.toResource(bet);
 return new ResponseEntity<BetResource>(resource, HttpStatus.OK);
}

@ExceptionHandler
ResponseEntity handleExceptions(Exception ex) {
 ResponseEntity responseEntity = null;
 if (ex instanceof BetNotFoundException) {
  responseEntity = new ResponseEntity(HttpStatus.NOT_FOUND);
 } else if (ex instanceof BetNotUnmatchedException) {
  responseEntity = new ResponseEntity(HttpStatus.CONFLICT);
 } else {
  responseEntity = new
    ResponseEntity(HttpStatus.INTERNAL_SERVER_ERROR);
 }
 return responseEntity;
}

}
```

All the operations will create a *BetResource* for returning to the client. This is done by calling*toResource* on the *BetResourceAssembler class*:

Spotlight Features

An Exhaustive List of Developer Topic Newsletters

Oracle Confirms New Java 9 Features

DevNexus Conference: Gearing up for DEVolution

My Most Frequent Code Review Comment

```
public class BetResourceAssembler extends
  ResourceAssemblerSupport<Bet, BetResource> {

 public BetResourceAssembler() {
 super(BetController.class, BetResource.class);
 }

 public BetResource toResource(Bet bet) {
 BetResource resource = instantiateResource(bet);
 resource.bet = bet;
   resource.add(linkTo(BetController.class).slash(bet.getId()).withS
 return resource;
 }

}
```

This class extends *ResourceAssemblerSupport* which requires the implementation of a *toResource*method as it implements the *ResourceAssembler* interface. This is where the mapping between *Bet*and *BetResource* is done. In this case, *BetResource* is just a wrapper for *Bet* so it is simply a case of setting the *bet* attribute. The *instantiateResource* method will return a *BetResource* without any links so links can be added at this point if required. In this example a link to self is added. An alternative approach would be to use *createResourceWithId* which will return a *BetResource* with the self link.

```
public class BetResource extends ResourceSupport {

 public Bet bet;

}
```

Also in this example, links are added to the *BetResource* within the *BetController* class to ensure the application of the HATEOAS constraint. If the REST service receives a GET request then a check is made on the status of the *Bet*. If the *Bet* is *UNMATCHED*, then a link to cancel the *Bet*can be added to the *BetResource*. This is done in similar fashion to the self link but with the relationship attribute name of cancel.An alternative approach to this is to build a link to a method as opposed to constructing a URI.

```
resource.add(linkTo(methodOn(BetController.class).cancel
.withRel("cancel"));
```

The *methodOn* would create a proxy of the *BetController* class and as a result the return type of the*cancelBet* method would have to be capable of proxying. Therefore in this example the return type of *cancelBet* method would be *HttpEntity<Bet>* and not *ResponseEntity<Bet>*. If the latter, then the likely exception from the server would be: [org.springframework.http.ResponseEntitycom.city81.hateoas.rest.BetResource> com.city81.hateoas.controller.BetController.getBet(java.lang.Long) throws com.city81.hateoas.BetNotFoundException]:org.springframework.aop.framework.AopConfigException: Could not generate CGLIB subclass of class [class org.springframework.http.ResponseEntity]: common causes of this problem include using a final class or a non-visible class; nested exception is java.lang.IllegalArgumentException: Superclass has no null constructors but no arguments were given Back to the GET request, and the returned JSON for requesting a *Bet* resource which has a status of *UNMATC*HED is shown below:

```
{
 "links":[
  {"rel":"self","href":http://localhost:8080/hateoas-1-SNAPSHOT/bets/0},
  {"rel":"cancel","href":http://localhost:8080/hateoas-1-SNAPSHOT/bets/0} ],
 "bet":{"id":0,"marketId":1,"selectionId":22,"price":4.0,"stake":2.0,"type":"BACK","status":"UNMATCHED"}
}
```

The client can therefore use the self link for retrieving and updating the *Bet*, and also the cancel link to effectively delete it.

This post describes just some of the functionality of the Spring-HATEOAS project which is evolving all the time. For an up to date and more detailed explanation, visit the GitHub pages.

Tags: Java REST Frameworks

The Enterprise Integration Zone is brought to you in partnership with WSO2. Learn more about WSO2's API Management.