Spring HATEOAS - Reference Documentation

Oliver Gierke, Greg Turnquist

Version 0.24.0.RELEASE, 2017-10-30

Table of Contents

This project provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC. The core problem it tries to address is link creation and representation assembly.

Chapter 1. Fundmentals

1.1. Jackson / JAXB integration

As representations for REST web services are usually rendered in either XML or JSON the natural choice of technology to achieve this is either Jackson, JAXB, or both in combination. To follow HATEOAS principles you need to incorporate links into those representation. Spring HATEOAS provides a set of useful types to ease working with those.

1.2. Links

The `Link` value object follows the Atom link definition and consists of a `rel` and an `href` attribute. It contains a few constants for well known rels such as `self`, `next` etc. The XML representation will render in the Atom namespace.

```
Link link = new Link("http://localhost:8080/something");
assertThat(link.getHref(), is("http://localhost:8080/something"));
assertThat(link.getRel(), is(Link.REL_SELF));

Link link = new Link("http://localhost:8080/something", "my-rel");
assertThat(link.getHref(), is("http://localhost:8080/something"));
assertThat(link.getRel(), is("my-rel"));
```

1.3. Resources

As pretty much every representation of a resource will contain some links (at least the `self` one) we provide a base class to actually inherit from when designing representation classes.

```
class PersonResource extends ResourceSupport {

  String firstname;
  String lastname;
}
```

Inheriting from `ResourceSupport` will allow adding links easily:

```
PersonResource resource = new PersonResource();
resource.firstname = "Dave";
resource.lastname = "Matthews";
resource.add(new Link("http://myhost/people"));
```

This would render as follows in JSON:

```
{
    "firstname" : "Dave",
    "lastname" : "Matthews",
    _"links" : [
        {
            "rel" : "self",
            "href" : "http://myhost/people"
        }
    ]
}
```

… or slightly more verbose in XML …

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <firstname>Dave</firstname>
  <lastname>Matthews</lastname>
  <links>
    <atom:link rel="self" href="http://myhost/people" />
  </links>
</person>
```

You can also easily access links contained in that resource:

```
Link selfLink = new Link("http://myhost/people");
assertThat(resource.getId(), is(selfLink));
assertThat(resource.getLink(Link.REL_SELF), is(selfLink));
```

1.4. Obtaining links

1.4.1. Link builder

Now we've got the domain vocabulary in place, but the main challenge remains: how to create the actual URIs to be wrapped into `Link`s in a less fragile way. Right now we'd have to duplicate URI strings all over the place which is brittle and unmaintainable.

Assume you have your Spring MVC controllers implemented as follows:

```
@Controller
@RequestMapping("/people")
class PersonController {

  @RequestMapping(method = RequestMethod.GET)
  public HttpEntity<PersonResource> showAll() { … }

  @RequestMapping(value = "/{person}", method = RequestMethod.GET)
  public HttpEntity<PersonResource> show(@PathVariable Long person) { … }
}
```

We see two conventions here. There's a collection resource exposed through the controller class' `@RequestMapping` annotation with individual elements of that collections exposed as direct sub resource. The collection resource might be exposed at a simple URI (as just shown) or more complex ones like `/people/{id}/addresses`. Let's say you would like to actually link to the collection resource of all people. Following the approach from up above would cause two problems:

1. To create an absolute URI you'd need to lookup the protocol, hostname, port, servlet base etc. This is cumbersome and requires ugly manual string concatenation code.
2. You probably don't want to concatenate the `/people` on top of your base URI because you'd have to maintain the information in multiple places then. Change the mapping, change all the clients pointing to it.

Spring Hateoas now provides a `ControllerLinkBuilder` that allows to create links by pointing to controller classes:

```
import static org.sfw.hateoas.mvc.ControllerLinkBuilder.*;

Link link = linkTo(PersonController.class).withRel("people");
assertThat(link.getRel(), is("people"));
assertThat(link.getHref(), endsWith("/people"));
```

The `ControllerLinkBuilder` uses Spring's `ServletUriComponentsBuilder` under the hood to obtain the basic URI information from the current request. Assuming your application runs at `http://localhost:8080/your-app` This will be exactly the URI you're constructing additional parts on top. The builder now inspects the given controller class for its root mapping and thus end up with `http://localhost:8080/your-app/people`. You can also easily build more nested links as well:

```
Person person = new Person(1L, "Dave", "Matthews");
//                 /person                /     1
Link link = linkTo(PersonController.class).slash(person.getId()).withSelfRel();
assertThat(link.getRel(), is(Link.REL_SELF));
assertThat(link.getHref(), endsWith("/people/1"));
```

If your domain class implements the `Identifiable` interface the `slash(…)` method will rather invoke `getId()` on the given object instead of `toString()`. Thus the just shown link creation can be

abbreviated to:

```
class Person implements Identifiable<Long> {
  public Long getId() { … }
}

Link link = linkTo(PersonController.class).slash(person).withSelfRel();
```

The builder also allows creating URI instances to build up e.g. response header values:

```
HttpHeaders headers = new HttpHeaders();
headers.setLocation(linkTo(PersonController.class).slash(person).toUri());
return new ResponseEntity<PersonResource>(headers, HttpStatus.CREATED);
```

1.4.2. Building links pointing to methods

As of version 0.4 you can even easily build links pointing to methods or creating dummy controller method invocations. The first approach is to hand a `Method` instance to the `ControllerLinkBuilder`:

```
Method method = PersonController.class.getMethod("show", Long.class);
Link link = linkTo(method, 2L).withSelfRel();

assertThat(link.getHref(), endsWith("/people/2")));
```

This is still a bit dissatisfying as we have to get a `Method` instance first, which throws an exception and is generally quite cumbersome. At least we don't repeat the mapping. An even better approach is to have a dummy method invocation of the target method on a controller proxy we can create easily using the `methodOn(…)` helper.

```
Link link = linkTo(methodOn(PersonController.class).show(2L)).withSelfRel();
assertThat(link.getHref(), endsWith("/people/2")));
```

`methodOn(…)` creates a proxy of the controller class that is recording the method invocation and exposes it in a proxy created for the return type of the method. This allows the fluent expression of the method we want to obtain the mapping for. However there are a few constraints on the methods that can be obtained using this technique:

1. The return type has to be capable of proxying as we need to expose the method invocation on it.
2. The parameters handed into the methods are generally neglected, except the ones referred to through `@PathVariable` as they make up the URI.

1.4.3. EntityLinks

So far we have created links by pointing to the web-framework implementations (i.e. Spring MVC controllers or JAX-RS resource classes) and inspected the mapping. In many cases these classes

essentially read and write representations backed by a model class.

The `EntityLinks` interface now exposes an API to lookup a `Link` or `LinkBuilder` based on the model types. The methods essentially return links to either point to the collection resource (e.g. `/people`) or a single resource (e.g. `/people/1`).

```
EntityLinks links = …;
LinkBuilder builder = links.linkFor(CustomerResource.class);
Link link = links.linkToSingleResource(CustomerResource.class, 1L);
```

`EntityLinks` is available for dependency injection by activating `@EnableEntityLinks` in your Spring MVC configuration. Activating this functionality will cause all your Spring MVC controllers and JAX-RS resource implementations available in the current `ApplicationContext` being inspected for the `@ExposesResourceFor(…)` annotation. The annotation exposes which model type the controller manages. Beyond that we assume you follow the URI mapping convention of a class level base mapping and assuming you have controller methods handling an appended `/{id}`. Here's an example implementation of an `EntityLinks` capable controller:

```
@Controller
@ExposesResourceFor(Order.class)
@RequestMapping("/orders")
class OrderController {

  @RequestMapping
  ResponseEntity orders(…) { … }

  @RequestMapping("/{id}")
  ResponseEntity order(@PathVariable("id") … ) { … }
}
```

The controller exposes that it manages `Order` instances and exposes handler methods that are mapped to our convention. Enabling `EntityLinks` through `@EnableEntityLinks` in your Spring MVC configuration you can now go ahead and create links to the just shown controller as follows.

```
@Controller
class PaymentController {

  @Autowired EntityLinks entityLinks;

  @RequestMapping(…, method = HttpMethod.PUT)
  ResponseEntity payment(@PathVariable Long orderId) {

    Link link = entityLinks.linkToSingleResource(Order.class, orderId);
    …
  }
}
```

As you can see you can refer to the `Order` instances without even referring to the `OrderController`.

1.5. Resource assembler

As the mapping from an entity to a resource type will have to be used in multiple places it makes sense to create a dedicated class responsible for doing so. The conversion will of course contain very custom steps but also a few boilerplate ones:

1. Instantiation of the resource class
2. Adding a link with rel `self` pointing to the resource that gets rendered.

Spring Hateoas now provides a `ResourceAssemblerSupport` base class that helps reducing the amount of code needed to be written:

```
class PersonResourceAssembler extends ResourceAssemblerSupport<Person, PersonResource>
{

  public PersonResourceAssembler() {
    super(PersonController.class, PersonResource.class);
  }

  @Override
  public PersonResource toResource(Person person) {

    PersonResource resource = createResource(person);
    // ⋯ do further mapping
    return resource;
  }
}
```

Setting the class up like this gives you the following benefits: there are a handful of `createResource(⋯)` methods that will allow you to create an instance of the resource and have a `Link` with a rel of `self` added to it. The href of that link is determined by the configured controller's request mapping plus the id of the `Identifiable` (e.g. `/people/1` in our case). The resource type gets instantiated by reflection and expects a no-arg constructor. Simply override `instantiateResource(⋯)` in case you'd like to use a dedicated constructor or avoid the reflection performance overhead.

The assembler can then be used to either assemble a single resource or an `Iterable` of them:

```
Person person = new Person(⋯);
Iterable<Person> people = Collections.singletonList(person);

PersonResourceAssembler assembler = new PersonResourceAssembler();
PersonResource resource = assembler.toResource(person);
List<PersonResource> resources = assembler.toResources(people);
```

Chapter 2. Configuration

2.1. @EnableHypermediaSupport

To enable the `ResourceSupport` subtypes be rendered according to the specification of various hypermedia representations types, the support for a particular hypermedia representation format can be activated through `@EnableHypermediaSupport`. The annotation takes a `HypermediaType` enumeration as argument. Currently we support HAL as well as a default rendering. Using the annotation triggers the following:

- registers necessary Jackson modules to render `Resource`/`Resources` in the hypermedia specific format.
- if JSONPath is on the classpath, it automatically registers a `LinkDiscoverer` instance to lookup links by their `rel` in plain JSON representations (see LinkDiscoverers).
- enables `@EnableEntityLinks` by default (see EntityLinks), will automatically pick up `EntityLinks` implementations and bundle them into a `DelegatingEntityLinks` instance available for autowiring.
- automatically picks up all `RelProvider` implementations in the `ApplicationContext` and bundles them into a `DelegatingRelProvider` available for autowiring. Registers providers to consider `@Relation` on domain types as well as Spring MVC controllers. If EVO inflector is on the classpath collection rels are derived using the pluralizing algorithm implemented in the library (see RelProvider API).

Chapter 3. SPIs

3.1. RelProvider API

When building links you usually need to determine the relation type to be used for the link. In most cases the relation type is directly associated with a (domain) type. We encapsulate the detailed algorithm to lookup the relation types behind a `RelProvider` API that allows to determine the relation types for single and collection resources. Here's the algorithm the relation type is looked up:

1. If the type is annotated with `@Relation` we use the values configured in the annotation.
2. if not, we default to the uncapitalized simple class name plus an appended `List` for the collection rel.
3. in case the EVO inflector JAR is in the classpath, we rather use the plural of the single resource rel provided by the pluralizing algorithm.
4. `@Controller` classes annotated with `@ExposesResourceFor` (see EntityLinks for details) will transparently lookup the relation types for the type configured in the annotation, so that you can use `relProvider.getSingleResourceRelFor(MyController.class)` and get the relation type of the domain type exposed.

A `RelProvider` is exposed as Spring bean when using `@EnableHypermediaSupport` automatically. You can plug in custom providers by simply implementing the interface and exposing them as Spring bean in turn.

3.2. CurieProvider API

The Web Linking RFC describes registered and extension link relation types. Registered rels are well-known strings registered with the IANA registry of link relation types. Extension rels can be used by applications that do not wish to register a relation type. They are a URI that uniquely identifies the relation type. The rel URI can be serialized as a compact URI or Curie. E.g. a curie `ex:persons` stands for the link relation type `http://example.com/rels/persons` if `ex` is defined as `http://example.com/rels/{rels}`. If curies are used, the base URI must be present in the response scope.

The rels created by the default `RelProvider` are extension relation types and as such must be URIs, which can cause a lot of overhead. The `CurieProvider` API takes care of that: it allows to define a base URI as URI template and a prefix which stands for that base URI. If a `CurieProvider` is present, the `RelProvider` prepends all rels with the curie prefix. Furthermore a `curies` link is automatically added to the HAL resource.

The configuration below defines a default curie provider.

```
@Configuration
@EnableWebMvc
@EnableHypermediaSupport(type= {HypermediaType.HAL})
public class Config {

  @Bean
  public CurieProvider curieProvider() {
    return new DefaultCurieProvider("ex", new UriTemplate(
"http://www.example.com/rels/{rel}"));
  }
}
```

Note that now the prefix `ex:` automatically appears before all rels which are not registered with IANA, as in `ex:orders`. Clients can use the `curies` link to resolve a curie to its full form:

```
{
  _"links" : {
    "self" : { href: "http://myhost/person/1" },
    "curies" : {
         "name" : "ex",
         "href" : "http://example.com/rels/{rel}",
         "templated" : true
    },
    "ex:orders" : { href : "http://myhost/person/1/orders" }
  },
  "firstname" : "Dave",
  "lastname" : "Matthews"
}
```

Since the purpose of the `CurieProvider` API is to allow for automatic curie creation, you can define only one `CurieProvider` bean per application scope.

Chapter 4. Client side support

4.1. Traverson

As of version 0.11 Spring HATEOAS provides an API for client side service traversal inspired by the Traverson JavaScript library.

```
Map<String, Object> parameters = new HashMap<>();
parameters.put("user", 27);

Traverson traverson = new Traverson(new URI("http://localhost:8080/api/"), MediaTypes
.HAL_JSON);
String name = traverson.follow("movies", "movie", "actor").
  withTemplateParameters(parameters).
  toObject("$.name");
```

You set up a `Traverson` instance by pointing it to a REST server and configure the media types you want to set as `Accept` header. You then go ahead and define the relation names you want to discover and follow. relation names can either be simple names or JSONPath expressions (starting with an `$`).

The sample then hands a parameter map into the execution. The parameters will be used to expand URIs found during the traversal that are templated. The traversal is concluded by accessing the representation of the final traversal. In the case of the sample we evaluate a JSONPath expression to access the actor's name.

The example listed above is the simplest version of traversal, where the rels are strings, and at each hop, the same template parameters are applied.

There are more options to customize template parameters at each level.

```
ParameterizedTypeReference<Resource<Item>> resourceParameterizedTypeReference = new
ParameterizedTypeReference<Resource<Item>>() {};

Resource<Item> itemResource = traverson.//
        follow(rel("items").withParameter("projection", "noImages")).//
        follow("$._embedded.items[0]._links.self.href").//
        toObject(resourceParameterizedTypeReference);
```

The static `rel(…)` function is a convenient way to define a single `Hop`. Using `.withParameter(key, value)` makes it simple to specify URI Template variables.

NOTE `.withParameter()` returns a new Hop object that is chainable. You can string together as many `.withParameter` as you like. The result is a single Hop definition.

```
ParameterizedTypeReference<Resource<Item>> resourceParameterizedTypeReference = new
ParameterizedTypeReference<Resource<Item>>() {};

Map<String, String> params = new HashMap<String, String>();
params.put("projection", "noImages");

Resource<Item> itemResource = traverson.//
        follow(rel("items").withParameters(params)).//
        follow("$._embedded.items[0]._links.self.href").//
        toObject(resourceParameterizedTypeReference);
```

It's also possible to load an entire `Map` of parameters via `.withParameters(Map)`.

NOTE `follow()` is chainable, meaning you can string together multiple hops as shown above. You can either put multiple, simple string-based rels (`follow("items", "item")`) or a single hop with specific parameters.

4.1.1. `Resource<T>` vs. `Resources<T>`

The examples shown so far demonstrate how to side step Java's type erasure and convert a single JSON-formatted resource into a `Resource<Item>` object. But what if you get a collection like an **_embedded** HAL collection?

One slight tweak and you're set.

```
ParameterizedTypeReference<Resources<Item>> resourceParameterizedTypeReference =
        new ParameterizedTypeReference<Resources<Item>>() {};

Resources<Item> itemResource = traverson.//
        follow(rel("items")).//
        toObject(resourceParameterizedTypeReference);
```

Instead of fetching a single resource, this one deserializes a collection into `Resources`.

4.2. LinkDiscoverers

When working with hypermedia enabled representations, a common task is to find a link with a particular relation type in them. Spring HATEOAS provides JSONPath based implementations of the `LinkDiscoverer` interface for either the default representation rendering or HAL out of the box. When using `@EnableHypermediaSupport` we automatically expose an instance supporting the configured hypermedia type as Spring bean.

Alternatively you can simply setup and use an instance like this:

```
String content = "{'_links' :  { 'foo' : { 'href' : '/foo/bar' }}}";
LinkDiscoverer discoverer = new HalLinkDiscoverer();
Link link = discoverer.findLinkWithRel("foo", content);

assertThat(link.getRel(), is("foo"));
assertThat(link.getHref(), is("/foo/bar"));
```