Spring HATEOAS - Reference Documentation

Oliver Gierke, Greg Turnquist, Jay Bryant

Version 1.0.0.BUILD-SNAPSHOT, 2019-05-13

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. Preface

1.1. Migrating to Spring HATEOAS 1.0

For 1.0 we took the chance to re-evaluate some of the design and package structure choices we had made for the 0.x branch. There had been an incredible amount of feedback on it and the major version bump seemed to be the most natural place to refactor those.

1.1.1. The changes

The biggest changes in package structure were driven by the introduction of a hypermedia type registration API to support additional media types in Spring HATEOAS. This lead to the clear separation of client and server APIs (packages named respectively) as well as media type implementations in the package `mediatype`.

The easiest way to get your code base upgraded to the new API is by using the migration script. Before we jump to that, here are the changes at a quick glance.

Representation models

The `ResourceSupport`/`Resource`/`Resources`/`PagedResources` group of classes never really felt appropriately named. After all, these types do not actually manifest resources but rather representation models that can be enriched with hypermedia information and affordances. Here's how new names map to the old ones:

- `ResourceSupport` is now `RepresentationModel`
- `Resource` is now `EntityModel`
- `Resources` is now `CollectionModel`
- `PagedResources` is now `PagedModel`

Consequently, `ResourceAssembler` has been renamed to `RepresentationModelAssembler` and its methods `toResource(…)` and `toResources(…)` have been renamed to `toModel(…)` and `toCollectionModel(…)` respectively. Also the name changes have been reflected in the classes contained in `TypeReferences`.

- `RepresentationModel.getLinks()` now exposes a `Links` instance (over a `List<Link>`) as that exposes additional API to concatenate and merge different `Links` instances using various strategies. Also it has been turned into a self-bound generic type to allow the methods that add links to the instance return the instance itself.
- The `LinkDiscoverer` API has been moved to the `client` package.
- The `LinkBuilder` and `EntityLinks` APIs have been moved to the `server` package.
- `ControllerLinkBuilder` has been moved into `server.mvc` and deprecated to be replaced by `WebMvcLinkBuilder`.
- `VndError` has been moved to the `mediatype.vnderror` package.

1.1.2. The migration script

You can find a script to run from your application root that will update all import statements and static method references to Spring HATEOAS types that moved in our source code repository. Simply download that, run it from your project root. By default it will inspect all Java source files and replace the legacy Spring HATEOAS type references with the new ones.

Example 1. Sample application of the migration script

```
$ ./migrate-to-1.0.sh

Migrating Spring HATEOAS references to 1.0 for files : *.java

Adapting ./src/main/java/…
…

Done!
```

Note that the script will not necessarily be able to entirely fix all changes, but it should cover the most important refactorings.

Now verify the changes made to the files in your favorite Git client and commit as appropriate. In case you find method or type references unmigrated, please open a ticket in out issue tracker.

Chapter 2. Fundamentals

This section covers the basics of Spring HATEOAS and its fundamental domain abstractions.

2.1. Links

The fundamental idea of hypermedia is to enrich the representation of a resource with hypermedia elements. The simplest form of that are links. They indicate a client that it can navigate to a certain resource. The semantics of a related resource are defined in a so called link relation. You might have seen this in the header of an HTML file already:

Example 2. A link in an HTML document

```
<link href="theme.css" rel="stylesheet" type="text/css" />
```

As you can see the link points to a resource `theme.css` and indicates that it is a style sheet. Links often carry additional information, like the media type that the resource pointed to will return. However, the fundamental building blocks of a link are its reference and relation.

Spring HATEOAS let's you work with links through its immutable `Link` value type. Its constructor take both an hypertext reference and a link relation, the latter being defaulted to the IANA link relation `self`. Read more on the latter in Link relations.

Example 3. Using links

```
Link link = new Link("/something");
assertThat(link.getHref()).isEqualTo("/something");
assertThat(link.getRel()).isEqualTo(IanaLinkRelations.SELF);

link = new Link("/something", "my-rel");
assertThat(link.getHref()).isEqualTo("/something");
assertThat(link.getRel()).isEqualTo(LinkRelation.of("my-rel"));
```

`Link` exposes other attributes as defined in RFC-5988. You can set them by calling the corresponding wither method on a `Link` instance.

Find more information on how to create links pointing to Spring MVC and Spring WebFlux controllers in [server.link-builder].

2.2. URI templates

For a Spring HATEOAS `Link`, the hypertext reference can not only be a URI, but also a URI template according to RFC-6570. A URI template contains so called template variables and allows expansion of these parameters. This allows clients to turn parameterized templates into URIs without having

to know about the structure of the final URI, it only needs to know about the names or the variables.

Example 4. Using links with templated URIs

```
Link link = new Link("/{segment}/something{?parameter}");
assertThat(link.isTemplated()).isTrue(); ①
assertThat(link.getVariableNames()).contains("segment", "parameter"); ②

Map<String, Object> values = new HashMap<>();
values.put("segment", "path");
values.put("parameter", 42);

assertThat(link.expand(values).getHref()) ③
    .isEqualTo("/path/something?parameter=42");
```

① The `Link` instance indicates that is templated, i.e. it contains a URI template.

② It exposes the parameters contained in the template.

③ It allows expansion of the parameters.

URI templates can be constructed manually and template variables added later on.

Example 5. Working with URI templates

```
UriTemplate template = new UriTemplate("/{segment}/something")
  .with(new TemplateVariable("parameter", VariableType.REQUEST_PARAM);

assertThat(template.toString()).isEqualTo("/{segment}/something{?parameter}");
```

2.3. Link relations

To indicate the relationship of target resource to the current one so called link relations are used. Spring HATEOAS provides a `LinkRelation` type to easily create `String`-based instances of it.

2.3.1. IANA link relations

The Internet Assigned Numbers Authority contains a set of predefined link relations. They can be referred to via `IanaLinkRelations`.

Example 6. Using IANA link relations

```
Link link = new Link("/some-resource"), IanaLinkRelations.NEXT);

assertThat(link.getRel()).isEqualTo(LinkRelation.of("next"));
assertThat(IanaLinkRelation.isIanaRel(link.getRel())).isTrue();
```

2.4. Representation models

To easily create hypermedia enriched representations, Spring HATEOAS provides a set of classes with `RepresentationModel` at their root. It's basically a container for a collection of `Link`s and has convenient methods to add those to the model. The models can later be rendered into various media type formats that will define how the hypermedia elements look in the representation. For more information on this, have a look at Media types

Example 7. The `RepresentationModel` class hierarchy

```
class RepresentationModel
class EntityModel
class CollectionModel
class PagedModel

EntityModel -|> RepresentationModel
CollectionModel -|> RepresentationModel
PagedModel -|> CollectionModel
```

The default way to work with a `RepresentationModel` is to create a subclass of it to contain all the properties the representation is supposed to contain, create instances of that class, populate the properties and enrich it with links.

Example 8. A sample representation model type

```
class PersonModel extends RepresentationModel<PersonModel> {

  String firstname, lastname;
}
```

The generic self-typing is necessary to let `RepresentationModel.add(…)` return instances of itself. The model type can now be used like this:

Example 9. Using the person representation model

```
PersonModel model = new PersonModel();
model.firstname = "Dave";
model.lastname = "Matthews";
model.add(new Link("https://myhost/people/42"));
```

If you returned such an instance from a Spring MVC or WebFlux controller and the client sent an `Accept` header set to `application/hal+json`, the response would look as follows:

Example 10. The HAL representation generated for the person representation model

```
{
  "_links" : {
    "self" : {
      "href" : "https://myhost/people/42"
    }
  },
  "firstname" : "Dave",
  "lastname" : "Matthews"
}
```

2.4.1. Item resource representation model

For a resource that's backed by a singular object or concept, a convenience `EntityModel` type exists. Instead of creating a custom model type for each concept, you can just reuse an already existing type and wrap instances of it into the `EntityModel`.

Example 11. Using `EntityModel` to wrap existing objects

```
Person person = new Person("Dave", "Matthews");
EntityModel<Person> model = new EntityModel<>(person);
```

2.4.2. Collection resource representation model

For resources that a conceptually collections, a `CollectionModel` is available. Its elements can either be simple objects or `RepresentationModel` instances in turn.

Example 12. Using `EntityModel` to wrap existing objects

```
Collection<Person> people = Collections.singleton(new Person("Dave", "Matthews"));
CollectionModel<Person> model = new CollectionModel<>(people);
```

2.5. Affordances

> The affordances of the environment are what it offers ... what it provides or furnishes, either for good or ill. The verb 'to afford' is found in the dictionary, but the noun 'affordance' is not. I have made it up.
>
> — James J. Gibson, The Ecological Approach to Visual Perception (page 126)

REST-based resources provide not just data but controls. The last ingredient to form a flexible service are detailed **affordances** on how to use the various controls.

Because affordances are associated with links, Spring HATEOAS provides an API to attach as many related methods as needed to a link. The following code shows how to take a **self** link and associate two more affordances:

Example 13. Connecting affordances to `GET /employees/{id}`

```
@GetMapping("/employees/{id}")
public EntityModel<Employee> findOne(@PathVariable Integer id) {

  Class<EmployeeController> controllerClass = EmployeeController.class;

  // Start the affordance with the "self" link, i.e. this method.
  Link findOneLink = linkTo(methodOn(controllerClass).findOne(id)).withSelfRel();
①

  // Return the affordance + a link back to the entire collection resource.
  return new EntityModel<>(EMPLOYEES.get(id), //
      findOneLink //
          .andAffordance(afford(methodOn(controllerClass).updateEmployee(null, id
))) ②
          .andAffordance(afford(methodOn(controllerClass).partiallyUpdateEmployee
(null, id)))); ③
}
```

① Create the **self** link.

② Associate the `updateEmployee` method with the `self` link.

③ Associate the `partiallyUpdateEmployee` method with the `self` link.

Using `.andAffordance(afford(⋯))`, you can use the controller's methods to connect a `PUT` and a `PATCH` operation to a `GET` operation.

Imagine that the related methods **afforded** above looking like this:

Example 14. `updateEmpoyee` method that responds to `PUT /employees/{id}`

```
@PutMapping("/employees/{id}")
public ResponseEntity<?> updateEmployee( //
    @RequestBody EntityModel<Employee> employee, @PathVariable Integer id)
```

Example 15. `partiallyUpdateEmployee` method that responds to `PATCH /employees/{id}`

```
@PatchMapping("/employees/{id}")
public ResponseEntity<?> partiallyUpdateEmployee( //
    @RequestBody EntityModel<Employee> employee, @PathVariable Integer id)
```

There are many media types that support rendering affordances. Unfortunately, HAL isn't one of them.

A HAL document for `GET /employees/{id}` would look like this:

Example 16. HAL document with no affordances

```
{
  "firstname" : "Frodo",
  "lastname" : "Baggins",
  "role" : "ring bearer",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/employees/1"
    }
  }
}
```

HAL supports providing links, but nothing else. While powerful, it doesn't let you show clients what inputs are required by its various operations. Nor does it show *what* HTTP methods are supported.

However, HAL-FORMS (`application/prs.hal-forms+json`), is a backwards compatible extension of HAL s that adds `_templates`. This affordance-aware media type can fill in what's missing.

The same resource above will render the following HAL-FORMS document:

Example 17. HAL-FORMS document with affordances

```
Unresolved directive in fundamentals.adoc - include::../../../src/docs/resources/
org/springframework/hateoas/docs/mediatype/hal/forms/hal-forms-sample-with-notes.
json[]
```

① The `_templates` attribute provided by HAL-FORMS with affordance-based information.

② The `updateEmployee` method's `@PutMapping` annotation is translated to `put`.

③ The method's `@RequestBody` input type is used to find domain `properties`.

④ For `POST` and `PUT`, all attributes are `required`.

⑤ The second affordance is named after the `partiallyUpdateEmployee` method.

⑥ `@PatchMapping` is translated into `patch`.

⑦ For `PATCH`, attributes are *not* `required`.

This rich document, consumable by any HAL-FORMS aware client includes enough extra details for full interaction with the resource.

In fact, this type of document makes it easy to write custom client-side code to generate an HTML form:

```
<form method="put" action="http://localhost:8080/employees/1">
  <input type="text" id="firstName" name="firstName"/>
  <input type="text" id="lastName" name="lastName" />
  <input type="text" id="role" name="role" />
  <input type="submit" value="Submit" />
</form>
```

Letting hypermedia drive web forms for users reduces the need for the client to know about the domain.

By trading in domain knowledge and instead adding protocol support for HAL-FORMS, clients can become flexible and receptive to server-side changes. No need to update your client every time a domain change is made on the server.

IMPORTANT	HAL-FORMS only supports affordances against the `self` link, but other affordance-aware media types may not have the same restriction. In general, don't define affordances based on one particular media type.

Chapter 3. Server-side support

3.1. Building links in Spring MVC

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

Assume you have your Spring MVC controllers implemented as follows:

```
@Controller
class PersonController {

  @GetMapping("/people")
  HttpEntity<PersonModel> showAll() { … }

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

We see two conventions here. The first is a collection resource that is exposed through `@GetMapping` annotation of the controller method, with individual elements of that collection exposed as direct sub resources. The collection resource might be exposed at a simple URI (as just shown) or more complex ones (such as `/people/{id}/addresses`). Suppose you would like to link to the collection resource of all people. Following the approach from up above would cause two problems:

- To create an absolute URI, you would need to look up the protocol, hostname, port, servlet base, and other values. This is cumbersome and requires ugly manual string concatenation code.
- You probably do not want to concatenate the `/people` on top of your base URI, because you would then have to maintain the information in multiple places. If you change the mapping, you then have to change all the clients pointing to it.

Spring HATEOAS now provides a `WebMvcLinkBuilder` that lets you create links by pointing to controller classes. The following example shows how to do so:

```
import static org.sfw.hateoas.server.mvc.WebMvcLinkBuilder.*;

Link link = linkTo(PersonController.class).withRel("people");

assertThat(link.getRel()).isEqualTo(LinkRelation.of("people"));
assertThat(link.getHref()).endsWith("/people");
```

The `WebMvcLinkBuilder` uses Spring's `ServletUriComponentsBuilder` under the hood to obtain the basic

URI information from the current request. Assuming your application runs at `localhost:8080/your-app`, this is exactly the URI on top of which you are constructing additional parts. The builder now inspects the given controller class for its root mapping and, thus, ends up with `localhost:8080/your-app/people`. You can also build more nested links as well. The following example shows how to do so:

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

The builder also allows creating URI instances to build up (for example, response header values):

```
HttpHeaders headers = new HttpHeaders();
headers.setLocation(linkTo(PersonController.class).slash(person).toUri());

return new ResponseEntity<PersonModel>(headers, HttpStatus.CREATED);
```

Building Links that Point to Methods

You can even build links that point to methods or create dummy controller method invocations. The first approach is to hand a `Method` instance to the `WebMvcLinkBuilder`. The following example shows how to do so:

```
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 first get a `Method` instance, which throws an exception and is generally quite cumbersome. At least we do not repeat the mapping. An even better approach is to have a dummy method invocation of the target method on a controller proxy, which we can create byi using the `methodOn(…)` helper. The following example shows how to do so:

```
Link link = linkTo(methodOn(PersonController.class).show(2L)).withSelfRel();

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

`methodOn(…)` creates a proxy of the controller class that records 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 for which we want to obtain the mapping. However, there are a few constraints on the methods that can be obtained byusing this technique:

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

3.2. Building links in Spring WebFlux

TODO

3.3. Forwarded header handling

RFC-7239 forwarding headers are most commonly used when your application is behind a proxy, behind a load balancer, or in the cloud. The node that actually receives the web request is part of the infrastructure, and *forwards* the request to your application.

Your application may be running on `localhost:8080`, but to the outside world, you're expected to be at `reallycoolsite.com` (and on web's standart port 80). By having the proxy include extra headers (which many already do), Spring HATEOAS can generate links properly as it uses Spring Framework functionality to obtain the base URI of the original request.

IMPORTANT	Anything that can change the root URI based on external inputs must be properly guarded. That's why, by default, forwarded header handling is **disabled**. You MUST enable it to be operational. If you are deploying to the cloud or into a configuration where you control the proxies and load balancers, then you'll certainly want to use this feature.

To enable forwarded header handling you need to register Spring's `ForwardedHeaderFilter` for Spring MVC (details here) or `ForwardedHeaderTransformer` for Spring WebFlux (details here) in your application. In a Spring Boot application those components can be simply declared as Spring beans as described here.

Example 18. Registering a `ForwardedHeaderFilter`

```
@Bean
ForwardedHeaderFilter forwardedHeaderFilter() {
    return new ForwardedHeaderFilter();
}
```

This will create a servlet filter that processes all the `X-Forwarded-…` headers. And it will register it properly with the servlet handlers.

For a Spring WebFlux application, the reactive counterpart is `ForwardedHeaderTransformer`:

Example 19. Registering a `ForwardedHeaderTransformer`

```
@Bean
ForwardedHeaderTransformer forwardedHeaderTransformer() {
    return new ForwardedHeaderTransformer();
}
```

This will create a function that transforms reactive web requests, processing `X-Forwarded-⋯` headers. And it will register it properly with WebFlux.

With configuration as shown above in place, a request passing `X-Forwarded-⋯` headers will see those reflected in the links generated:

Example 20. A request using `X-Forwarded-⋯` headers

```
curl -v localhost:8080/employees \
    -H 'X-Forwarded-Proto: https' \
    -H 'X-Forwarded-Host: example.com' \
    -H 'X-Forwarded-Port: 9001'
```

Example 21. The corresponding response with the links generated to consider those headers

```
{
  "_embedded": {
    "employees": [
      {
        "id": 1,
        "name": "Bilbo Baggins",
        "role": "burglar",
        "_links": {
          "self": {
            "href": "https://example.com:9001/employees/1"
          },
          "employees": {
            "href": "https://example.com:9001/employees"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "https://example.com:9001/employees"
    },
    "root": {
      "href": "https://example.com:9001"
    }
  }
}
```

3.4. Using the EntityLinks interface

So far, we have created links by pointing to the web-framework implementations (that is, the Spring MVC controllers) 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 look up a `Link` or `LinkBuilder` based on the model types. The methods essentially return links that point either to the collection resource (such as `/people`) or to an item resource (such as `/people/1`). The following example shows how to use `EntityLinks`:

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

`EntityLinks` is available via dependency injection by activating either `@EnableHypermediaSupprt` or `@EnableEntityLinks` in your Spring MVC configuration. This will cause a variety of default implementations of `EntityLinks` being registered. The most fundamental one is `ControllerEntityLinks` that inspects SpringMVC and Spring WebFlux controller classes. If you want to register your own implementation of `EntityLinks`, check out this section.

3.4.1. EntityLinks based on Spring MVC and WebFlux controllers

Activating entity links functionality causes all the Spring MVC and WebFlux controllers available in the current `ApplicationContext` to be inspected for the `@ExposesResourceFor(…)` annotation. The annotation exposes which model type the controller manages. Beyond that, we assume that you adhere to following the URI mapping setup and conventions:

- A type level `@ExposesResourceFor(…)` declaring which entity type the controller exposes collection and item resources for.
- A class level base mapping that represents the collection resource.
- An additional method level mapping that extends the mapping to append an identifier as additional path segment.

The following example shows an implementation of an `EntityLinks`-capable controller:

```
@Controller
@ExposesResourceFor(Order.class) ①
@RequestMapping("/orders") ②
class OrderController {

  @GetMapping ③
  ResponseEntity orders(…) { … }

  @GetMapping("{id}") ④
  ResponseEntity order(@PathVariable("id") … ) { … }
}
```

① The controller indicates it's exposing collection and item resources for the entity `Order`.

② Its collection resource is exposed under `/orders`

③ That collection resource can handle `GET` requests. Add more methods for other HTTP methods at your convenience.

④ An additional controller method to handle a subordinate resource taking a path variable to expose an item resource, i.e. a single `Order`.

With this in place, when youy enable `EntityLinks` through `@EnableEntityLinks` or `@EnableHypermediaSupport` in your Spring MVC configuration, you can create links to the controller, as follows:

```
@Controller
class PaymentController {

  private final EntityLinks entityLinks;

  PaymentController(EntityLinks entityLinks) { ①
    this.entityLinks = entityLinks;
  }

  @PutMapping(…)
  ResponseEntity payment(@PathVariable Long orderId) {

    Link link = entityLinks.linkToItemResource(Order.class, orderId); ②
    …
  }
}
```

① Inject `EntityLinks` made available by `@EnableEntityLinks` or `@EnableHypermediaSupport` in you configuration.

② Use the APIs to build links by using the entity types instead of controller classes.

As you can see, you can refer to resources managing `Order` instances without referring to `OrderController` explicitly.

3.4.2. EntityLinks API in detail

Fundamentally, `EntityLinks` allows to build `LinkBuilder`s and `Link` instances to collection and item resources of a entity type. Methods starting with `linkFor…` will produce `LinkBuilder` instances for you to extend and augment with additional path segments, parameters, etc. Methods starting with `linkTo` produce fully prepared `Link` instances.

While for collection resources providing an entity type is sufficient, links to item resources will need an identifier provided. This usually looks like this

Example 22. Obtaining a link to an item resource

```
entityLinks.linkToItemResource(order, order.getId());
```

If you find yourself repeating those method calls the identifier extraction step can be pulled out into a reusable `Function` to be reused throughout different invocations:

```
Function<Order, Object> idExtractor = Order::getId; ①

entityLinks.linkToItemResource(order, idExtractor); ②
```

① The identifier extraction is externalized so that it can be held in a field or constant.

② The link lookup using the extractor.

TypedEntityLinks

As controller implementations are often grouped around entity types, you'll very often find yourself using the same extractor function (see EntityLinks API in detail for details) all over the controller class. We can centralize the identifier extraction logic even more by obtaining a `TypedEntityLinks` instance providing the extractor once, so that the actually lookups don't have to deal with the extraction anymore at all.

Example 23. Using TypedEntityLinks

```
class OrderController {

  private final TypedEntityLinks<Order> links;

  OrderController(EntityLinks entityLinks) { ①
    this.links = entityLinks.forType(Order::getId); ②
  }

  @GetMapping
  ResponseEntity<Order> someMethod(…) {

    Order order = … // lookup order

    Link link = links.linkToItemResource(order); ③
  }
}
```

① Inject an `EntityLinks` instance.

② Indicate you're going to look up `Order` instances with a certain identifier extractor function.

③ Lookup item resource links based on a sole `Order` instance.

3.4.3. EntityLinks as SPI

The `EntityLinks` instance created by `@EnableEntityLinks` / `@EnableHypermediaSupport` is of type `DelegatingEntityLinks` which will in turn pick up all other `EntityLinks` implementations available as beans in the `ApplicationContext`. It's registered as primary bean so that it's always the sole injection candidate when you inject `EntityLinks` in general. `ControllerEntityLinks` is the default implementation that will be included in the setup, but users are free to implement and register

their own implementations. Making those available to the `EntityLinks` instance available for injection is a matter of registering your implementation as Spring bean.

Example 24. Declaring a custom EntityLinks implementation

```
@Configuration
class CustomEntityLinksConfiguration {

  @Bean
  MyEntityLinks myEntityLinks(…) {
    return new MyEntityLinks(…);
  }
}
```

An example for the extensibility of this mechanism is Spring Data REST's `RepositoryEntityLinks`, which uses the repository mapping information to create links pointing to resources backed by Spring Data repositories. At the same time, it even exposes additional lookup methods for other types of resources. If you want to make use of these, simply inject `RepositoryEntityLinks` explicitly.

3.5. Representation model assembler

As the mapping from an entity to a representation model must be used in multiple places, it makes sense to create a dedicated class responsible for doing so. The conversion contains very custom steps but also a few boilerplate steps:

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

Spring HATEOAS now provides a `RepresentationModelAssemblerSupport` base class that helps reduce the amount of code you need to write. The following example shows how to use it:

```
class PersonModelAssembler extends RepresentationModelAssemblerSupport<Person,
PersonModel> {

  public PersonModelAssembler() {
    super(PersonController.class, PersonModel.class);
  }

  @Override
  public PersonModel toModel(Person person) {

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

Setting the class up as we did in the preceding example gives you the following benefits:

- There are a handful of `createModelWithId(⋯)` methods that let you 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 entity (for example, `/people/1`).
- The resource type gets instantiated by reflection and expects a no-arg constructor. If you want to use a dedicated constructor or avoid the reflection performance overhead, you can override `instantiateModel(⋯)`.

You can then use the assembler to either assemble a `RepresentationModel` or a `CollectionModel`. The following example creates a `CollectionModel` of `PersonModel` instances:

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

PersonModelAssembler assembler = new PersonModelAssembler();
PersonModel model = assembler.toModel(person);
CollectionModel<PersonModel> model = assembler.toCollectionModel(people);
```

3.6. Representation Model Processors

Sometimes you need to tweak and adjust hypermedia representations after they have been assembled.

A perfect example is when you have a controller that deals with order fulfillment, but you need to add links related to making payments.

Imagine having your ordering system producing this type of hypermedia:

```
Unresolved directive in server.adoc - include::../../../src/docs/resources/org/sp
ringframework/hateoas/docs/order-plain.json[]
```

You wish to add a link so the client can make payment, but don't want to mix details about your `PaymentController` into the `OrderController`.

Instead of polluting the details of your ordering system, you can write a `RepresentationModelProcessor` like this:

```
public class PaymentProcessor implements RepresentationModelProcessor<EntityModel
<Order>> { ①

  @Override
  public EntityModel<Order> process(EntityModel<Order> model) {

    model.add( ②
        new Link("/payments/{orderId}").withRel(LinkRelation.of("payments")) //
            .expand(model.getContent().getOrderId()));

    return model; ③
  }
}
```

① This processor will only be applied to `EntityModel<Order>` objects.

② Manipulate the existing `EntityModel` object by adding an unconditional link.

③ Return the `EntityModel` so it can be serialized into the requested media type.

Register the processor with your application:

```
@Configuration
public class PaymentProcessingApp {

  @Bean
  PaymentProcessor paymentProcessor() {
    return new PaymentProcessor();
  }
}
```

Now when you issue a hypermedia respresentation of an `Order`, the client receives this:

```
Unresolved directive in server.adoc - include::../../../src/docs/resources/org
/springframework/hateoas/docs/order-with-payment-link.json[]
```

① You see the `LinkRelation.of("payments")` plugged in as this link's relation.

② The URI was provided by the processor.

This example is quite simple, but you can easily:

- Use `WebMvcLinkBuilder` or `WebFluxLinkBuilder` to construct a dynamic link to your `PaymentController`.
- Inject any services needed to conditionally add other links (e.g. `cancel`, `amend`) that are driven by state.
- Leverage cross cutting services like Spring Security to add, remove, or revise links based upon the current user's context.

Also, in this example, the `PaymentProcessor` alters the provided `EntityModel<Order>`. You also have the power to *replace* it with another object. Just be advised the API requires the return type to equal the input type.

3.7. Using the `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 look up the relation types behind a `RelProvider` API that lets you determine the relation types for single and collection resources. The algorithm for looking up the relation type follows:

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. If the EVO inflector JAR is in the classpath, we use the plural of the single resource `rel` provided by the pluralizing algorithm.
4. `@Controller` classes annotated with `@ExposesResourceFor` (see [fundamentals.obtaining-links.entity-links] for details) transparently look up the relation types for the type configured in the annotation, so that you can use `relProvider.getItemResourceRelFor(MyController.class)` and get the relation type of the domain type exposed.

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

Chapter 4. Media types

4.1. HAL – Hypertext Application Language

JSON Hypertext Application Language or HAL is one of the simplest and most widely adopted hypermedia media types adopted when not discussing specific web stacks.

It was the first spec-based media type adopted by Spring HATEAOS.

4.1.1. Configuring link rendering

In HAL, the `_links` entry is a JSON object. The property names are link relations and each value is either a link object or an array of link objects.

For a given link relation that has two or more links, the spec is clear on representation:

Example 25. HAL document with two links associated with one relation

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/org/springframework/hateoas/docs/mediatype/hal/hal-multiple-entry-link-relation.json[]
```

But if there is only one link for a given relation, the spec is ambiguous. You could render that as either a single object or as 1-item array.

By default, Spring HATEOAS uses the most terse approach and renders a single-link relation like this:

Example 26. HAL document with single link rendered as an object

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/org/springframework/hateoas/docs/mediatype/hal/hal-single-entry-link-relation-object.json[]
```

Some users prefer to not switch between arrays and objects when consuming HAL. They would prefer this type of rendering:

Example 27. HAL with single link rendered as an array

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/org/springframework/hateoas/docs/mediatype/hal/hal-single-entry-link-relation-array.json[]
```

If you wish to customize this policy, all you have to do is inject a `HalConfiguration` bean into your application configuration. There are multiple choices.

Example 28. Global HAL single-link rendering policy

```
@Bean
public HalConfiguration globalPolicy() {
  return new HalConfiguration() //
       .withRenderSingleLinks(RenderSingleLinks.AS_ARRAY); ①
}
```

① Override Spring HATEOAS's default by rendering ALL single-link relations as arrays.

If you prefer to only override some particular link relations, you can create a `HalConfiguration` bean like this:

Example 29. Link relation-based HAL single-link rendering policy

```
@Bean
public HalConfiguration linkRelationBasedPolicy() {
  return new HalConfiguration() //
       .withRenderSingleLinksFor( //
           IanaLinkRelations.ITEM, RenderSingleLinks.AS_ARRAY) ①
       .withRenderSingleLinksFor( //
           LinkRelation.of("prev"), RenderSingleLinks.AS_SINGLE); ②
}
```

① Always render `item` link relations as an array.

② Render `prev` link relations as an object when there is only one link.

If neither of these match your needs, you can use an Ant-style path patterns:

Example 30. Pattern-based HAL single-link rendering policy

```
@Bean
public HalConfiguration patternBasedPolicy() {
  return new HalConfiguration() //
       .withRenderSingleLinksFor( //
           "http*", RenderSingleLinks.AS_ARRAY); ①
}
```

① Render all link relations that start with `http` as an array.

NOTE	The pattern-based approach uses Spring's `AntPathMatcher`.

All of these `HalConfiguration` withers can be combined to form one comprehensive policy. Be sure to test your API extensively to avoid surprises.

4.1.2. Using the `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 `rel` URIs can be used by applications that do not wish to register a relation type. Each one is a URI that uniquely identifies the relation type. The `rel` URI can be serialized as a compact URI or Curie. For example, a curie of `ex:persons` stands for the link relation type `example.com/rels/persons` if `ex` is defined as `example.com/rels/{rel}`. If curies are used, the base URI must be present in the response scope.

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

The following configuration defines a default curie provider:

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

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

Note that now the `ex:` prefix automatically appears before all rel values that are not registered with IANA, as in `ex:orders`. Clients can use the `curies` link to resolve a curie to its full form. The following example shows how to do so:

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/or
g/springframework/hateoas/docs/mediatype/hal/hal-with-curies.json[]
```

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

4.2. HAL-FORMS

HAL-FORMS is designed to add runtime FORM support to the HAL media type.

> HAL-FORMS "looks like HAL." However, it is important to keep in mind that HAL-FORMS is not the same as HAL — the two should not be thought of as interchangeable in any way.
>
> — Mike Amundsen, HAL-FORMS spec

To enable this media type, put the following configuration in your code:

Example 31. HAL-FORMS enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.HAL_FORMS)
public class HalFormsApplication {

}
```

Anytime a client supplies an `Accept` header with `application/prs.hal-forms+json`, you can expect something like this:

Example 32. HAL-FORMS sample document

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/or
g/springframework/hateoas/docs/mediatype/hal/forms/hal-forms-sample.json[]
```

Checkout the HAL-FORMS spec to understand the details of the **_templates** attribute. Read about the Affordances API to augment your controllers with this extra metadata.

As for single-item (`EntityModel`) and aggregate root collections (`CollectionModel`), Spring HATEOAS renders them identically to HAL documents.

4.3. Collection+JSON

Collection+JSON is a JSON spec registered with IANA-approved media type `application/vnd.collection+json`.

> Collection+JSON is a JSON-based read/write hypermedia-type designed to support management and querying of simple collections.
>
> — Mike Amundsen, Collection+JSON spec

Collection+JSON provides a uniform way to represent both single item resources as well as

collections.

To enable this media type, put the following configuration in your code:

Example 33. Collection+JSON enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.COLLECTION_JSON)
public class CollectionJsonApplication {

}
```

This configuration will make your application respond to requests that have an `Accept` header of `application/vnd.collection+json` as shown below.

The following example from the spec shows a single item:

Example 34. Collection+JSON single item example

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/or
g/springframework/hateoas/docs/mediatype/collectionjson/spec-part3.json[]
```

① The `self` link is stored in the document's `href` attribute.

② The document's top `links` section contains collection-level links (minus the `self` link).

③ The `items` section contains a collection of data. Since this is a single-item document, it only has one entry.

④ The `data` section contains actual content. It's made up of properties.

⑤ The item's individual `links`.

IMPORTANT

The previous fragment was lifted from the spec. When Spring HATEOAS renders an `EntityModel`, it will:

- Put the `self` link into both the document's `href` attribute and the item-level `href` attribute.
- Put the rest of the model's links into both the top-level `links` as well as the item-level `links`.
- Extract the properties from the `EntityModel` and turn them into

When rendering a collection of resources, the document is almost the same, except there will be multiple entries inside the `items` JSON array, one for each entry.

Spring HATEOAS more specifically will:

- Put the entire collection's `self` link into the top-level `href` attribute.

- The `CollectionModel` links (minus `self`) will be put into the top-level `links`.
- Each item-level `href` will contain the corresponding `self` link for each entry from the `CollectionModel.content` collection.
- Each item-level `links` will contain all other links for each entry from `CollectionModel.content`.

4.4. UBER - Uniform Basis for Exchanging Representations

UBER is an experimental JSON spec

> The UBER document format is a minimal read/write hypermedia type designed to support simple state transfers and ad-hoc hypermedia-based transitions.
>
> — Mike Amundsen, UBER spec

UBER provides a uniform way to represent both single item resources as well as collections.

To enable this media type, put the following configuration in your code:

Example 35. UBER+JSON enabled application

```
@Configuration
@EnableHypermediaSupport(type = HypermediaType.UBER)
public class UberApplication {

}
```

This configuration will make your application respond to requests using the `Accept` header `application/vnd.amundsen-uber+json` as show below:

Example 36. UBER sample document

```
Unresolved directive in mediatypes.adoc - include::../../../src/docs/resources/or
g/springframework/hateoas/docs/mediatype/uber/uber-sample.json[]
```

This media type is still under development as is the spec itself. Feel free to open a ticket if you run into issues using it.

NOTE	**UBER media type** is not associated in any way with **Uber Technologies Inc.**, the ride sharing company.

4.5. Registering a custom media type

Spring HATEOAS allows to integrate support for custom media types through a set of SPIs, that third parties can implement. The building blocks of an such an implementations are:

1. Some form of Jackson ObjectMapper customization. In its most simple case that's a Jackson `Module` implementation.
2. A `LinkDiscoverer` implementation so that the client side support is able to detect links in representations generated.
3. Some configuration infrastructure that will allow Spring HATEOAS to find the custom implementation and pick up its configuration.

4.5.1. Custom media type configuration

Custom media type implementations are picked up through Spring's `SpringFactories` mechanism, similar to the Java `ServiceLoader` API. Each media type implementation needs to ship with a `spring.factories` in `META-INF` containing an implementation class entry for the `org.springframework.hateoas.config.MediaTypeConfigurationProvider` key:

Example 37. An example `MediaTypeConfigurationProvider` declaration

```
org.springframework.hateoas.config.MediaTypeConfigurationProvider=\
  com.acme.mymediatype.MyMediaTypeConfigurationProvider
```

That implementation class could then look as follows:

Example 38. An example `MediaTypeConfigurationProvider` implementation in `META-INF/spring.factories`

```
class MyMediaTypeConfigurationProvider
  implements MediaTypeConfigurationProvider {

  @Override
  public Class<? extends HypermediaMappingInformation> getConfiguration() {
    return MyMediaTypeConfiguration.class; ①
  }

  @Override
  public boolean supportsAny(Collection<MediaType> mediaTypes) {
    return mediaTypes.contains(MediaTypes.HAL_JSON); ②
  }
}
```

The configuration class needs to have a default constructor and expose two methods:

① A method returning a Spring configuration class that will be included in the application bootstrap when Spring HATEOAS is activated (either implicitly via Spring Boot auto-configuration or via `@EnableHypermediaSupport`).

② A callback method that will get the application selected media types to activate passed. This allows the media type implementation to control, when it it will be activated.

The configuration class has to implement `HypermediaMappingInformation`. It could look as simple as this:

```
@Configuration
class MyMediaTypeConfiguration implements HypermediaMappingInformation {

  @Override
  public List<MediaType> getMediaTypes() {
    return MediaType.parse("application/vnd-acme-media-type") ①
  }

  @Override
  public Module getJacksonModule() {
    return new Jackson2MediaTypeModule(); ②
  }

  @Bean
  MyLinkDiscoverer myLinkDiscoverer() {
    return new MyLinkDiscoverer(); ③
  }
}
```

① The configuration class returns the media type it wants to get Spring MVC / Spring WebFlux support set up.

② It overrides `getJacksonModule()` to provide custom serializers to create the media type specific representations.

③ It also declares a custom `LinkDiscoverer` implementation for client side support.

The Jackson module usually declares `Serializer` and `Deserializer` implementations for the representation model types `RepresentationModel`, `EntityModel`, `CollectionModel` and `PagedModel`. In case you need further customization of the Jackson `ObjectMapper` (like a custom `HandlerInstantiator`), you can alternatively override `configureObjectMapper(…)`.

4.5.2. Recommendations

The preferred way to implement media type representations is by providing a type hierarchy that matches the expected format and can be serialized by Jackson as is. In the `Serializer` and `Deserializer` implementations registered for `RepresentationModel`, convert the instances into the media type specific model types and then lookup the Jackson serializer for those.

The media types supported by default use the same configuration mechanism as third party implementations would. So it's worth studying the implementations in the `mediatype` package.

Chapter 5. Configuration

This section describes how to configure Spring HATEOAS.

5.1. Using `@EnableHypermediaSupport`

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

- It registers necessary Jackson modules to render `EntityModel` and `CollectionModel` in the hypermedia specific format.
- If JSONPath is on the classpath, it automatically registers a `LinkDiscoverer` instance to look up links by their `rel` in plain JSON representations (see Using `LinkDiscoverer` Instances).
- By default, it enables `@EnableEntityLinks` (see [fundamentals.obtaining-links.entity-links]) and automatically picks up `EntityLinks` implementations and bundles them into a `DelegatingEntityLinks` instance that you can autowire.
- It automatically picks up all `RelProvider` implementations in the `ApplicationContext` and bundles them into a `DelegatingRelProvider` that you can autowire. It registers providers to consider `@Relation` on domain types as well as Spring MVC controllers. If the EVO inflector is on the classpath, collection `rel` values are derived by using the pluralizing algorithm implemented in the library (see [spis.rel-provider]).

5.1.1. Explicitly enabling support for dedicated web stacks

By default, `@EnableHypermediaSupport` will reflectively detect the web application stack you're using and hook into the Spring components registered for those to enable support for hypermedia representations. However, there are situations in which you'd only explicitly want to activate support for a particular stack. E.g. if your Spring WebMVC based application uses WebFlux' `WebClient` to make outgoing requests and that one is not supposed to work with hypermedia elements, you can restrict the functionality to be enabled by explicitly declaring WebMvc in the configuration:

Example 39. Explicitly activating hypermedia support for a particular web stack

```
@EnableHypermediaSupport(…, stacks = WebStack.WEBMVC)
class MyHypermediaConfiguration { … }
```

Chapter 6. Client-side Support

This section describes Spring HATEOAS's support for clients.

6.1. Traverson

Spring HATEOAS provides an API for client-side service traversal. It is inspired by the Traverson JavaScript library. The following example shows how to use it:

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

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

You can set up a `Traverson` instance by pointing it to a REST server and configuring the media types you want to set as `Accept` headers. You can then 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 are used to expand URIs (which are templated) found during the traversal. The traversal is concluded by accessing the representation of the final traversal. In the preceding example, we evaluate a JSONPath expression to access the actor's name.

The preceding example is the simplest version of traversal, where the `rel` values are strings and, at each hop, the same template parameters are applied.

There are more options to customize template parameters at each level. The following example shows these options.

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

EntityModel<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. The following example shows one way to do so:

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

Map<String, Object> params = Collections.singletonMap("projection", "noImages");

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

You can also load an entire `Map` of parameters by using `.withParameters(Map)`.

NOTE

`follow()` is chainable, meaning you can string together multiple hops, as shown in the preceding examples. You can either put multiple string-based `rel` values (`follow("items", "item")`) or a single hop with specific parameters.

6.1.1. `EntityModel<T>` vs. `CollectionModel<T>`

The examples shown so far demonstrate how to sidestep Java's type erasure and convert a single JSON-formatted resource into a `EntityModel<Item>` object. However, what if you get a collection like an `\_embedded` HAL collection? You can do so with only one slight tweak, as the following example shows:

```
CollectionModelType<Item> collectionModelType =
        TypeReferences.CollectionModelType<Item>() {};

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

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

6.2. Using `LinkDiscoverer` Instances

When working with hypermedia enabled representations, a common task is to find a link with a particular relation type in it. 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 a Spring bean.

Alternatively, you can setup and use an instance as follows:

```
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"));
```