Spring Data REST Reference Documentation

Jon Brisbin, Oliver Gierke

Table of Contents

2.2.2.RELEASE

Chapter 1. Introduction

REST web services have become the number one means for application integration on the web. In its core, REST defines that a system consists of resources that clients interact with. These resources are implemented in a hypermedia drive way. Spring MVC offers a solid foundation to build theses kinds of services but implementing very basic functionality of REST web service can be tedious and result in a lot of boilerplate code.

Spring Data REST builds on top of Spring Data repositories and automatically exports those as REST resources. It leverages hypermedia to allow clients to find functionality exposed by the repositories and allows to integrate the resources into related hypermedia based functionality as easy as possible.

Chapter 2. Getting started

2.1. Introduction

Spring Data REST is itself a Spring MVC application and is designed in such a way that it should integrate with your existing Spring MVC applications with very little effort. An existing (or future) layer of services can run alongside Spring Data REST with only minor considerations.

To install Spring Data REST alongside your application, simply add the required dependencies, include the stock `@Configuration` class `RepositoryRestMvcConfiguration` (or subclass it and perform any required manual configuration), and map some URLs to be managed by Spring Data REST.

2.2. Adding Spring Data REST to a Gradle project

To add Spring Data REST to a Gradle-based project, add the `spring-data-rest-webmvc` artifact to your compile-time dependencies:

```
dependencies {
     other project dependencies
  compile "org.springframework.data:spring-data-rest-webmvc:${spring-data-rest-version}"
}
```

2.3. Adding Spring Data REST to a Maven project

To add Spring Data REST to a Maven-based project, add the `spring-data-rest-webmvc` artifact to your compile-time dependencies:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-webmvc</artifactId>
  <version>${spring-data-rest-version}</version>
</dependency>
```

2.4. Configuring Spring Data REST

To install Spring Data REST alongside your existing Spring MVC application, you need to include the appropriate MVC configuration. Spring Data REST configuration is defined in a class called `RepositoryRestMvcConfiguration`. You can either import this class into your existing configuration using an `@Import` annotation or you can subclass it and override any of the `configureXXX` methods to add your own configuration to that of Spring Data REST.

In the following example, we'll subclass the standard `RepositoryRestMvcConfiguration` and add some `ResourceMapping` configurations for the `Person` domain object to alter how the JSON will look and how the links to related entities will be handled.

```
@Configuration
@Import(RepositoryRestMvcConfiguration.class)
public class MyWebConfiguration extends RepositoryRestMvcConfiguration {

  //    further configuration
}
```

Make sure you also configure Spring Data repositories for the store you use. For details on that, please consult the reference documentation for the corresponding Spring Data module.

2.5. Starting the application

As Spring Data REST is built on SpringMVC, you simply stick to the means you use to bootstrap Spring MVC. In a Servlet 3.0 environment this might look something like this:

```
public class RestExporterWebInitializer implements WebApplicationInitializer {

  @Override public void onStartup(ServletContext servletContext) throws ServletException
{

    // Bootstrap repositories in root application context
    AnnotationConfigWebApplicationContext rootCtx = new
AnnotationConfigWebApplicationContext();
    rootCtx.register(JpaRepositoryConfig.class); // Include JPA entities, Repositories
    servletContext.addListener(new ContextLoaderListener(rootCtx));

    // Enable Spring Data REST in the DispatcherServlet
    AnnotationConfigWebApplicationContext webCtx = new
AnnotationConfigWebApplicationContext();
    webCtx.register(MyWebConfiguration.class);

    DispatcherServlet dispatcherServlet = new DispatcherServlet(webCtx);
    ServletRegistration.Dynamic reg = servletContext.addServlet("rest-exporter",
dispatcherServlet);
    reg.setLoadOnStartup(1);
    reg.addMapping("/*");
  }
}
```

The equivalent of the above in a standard web.xml will also work identically to this configuration if

you are still in a servlet 2.5 environment. When you deploy this application to your servlet container, you should be able to see what repositories are exported by accessing the root of the application.

Chapter 3. Repository resources

3.1. Fundamentals

The core functionality of Spring Data REST is to export resources for Spring Data repositories. Thus, the core artifact to look at and potentially tweak to customize the way the exporting works is the repository interface. Assume the following repository interface:

```
public interface OrderRepository extends CrudRepository<Order, Long> { }
```

For this repository, Spring Data REST exposes a collection resource at `/orders`. The path is derived from the uncapitalized, pluralized, simple class name of the domain class being managed. It also exposes an item resource for each of the items managed by the repository under the URI template `/orders/{id}`.

By default the HTTP methods to interact with these resources map to the according methods of `CrudRepository`. Read more on that in the sections on collection resources and item resources.

3.1.1. Default status codes

For the resources exposed, we use a set of default status codes:

- `200 OK` - for plain `GET` requests.
- `201 Created` - for `POST` and `PUT` requests that create new resources.
- `204 No Content` - for `PUT`, `PATCH`, and `DELETE` requests if the configuration is set to not return response bodies for resource updates (`RepositoryRestConfiguration.returnBodyOnUpdate`). If the configuration value is set to include responses for `PUT`, `200 OK` will be returned for updates, `201 Created` will be returned for resource created through `PUT`.

3.1.2. Resource discoverability

A core principle of HATEOAS is that resources should be discoverable through the publication of links that point to the available resources. There are a few competing de-facto standards of how to represent links in JSON. By default, Spring Data REST uses HAL to render responses. HAL defines links to be contained in a property of the returned document.

Resource discovery starts at the top level of the application. By issuing a request to the root URL under which the Spring Data REST application is deployed, the client can extract a set of links from the returned JSON object that represent the next level of resources that are available to the client.

For example, to discover what resources are available at the root of the application, issue an HTTP `GET` to the root URL:

```
curl -v http://localhost:8080/

< HTTP/1.1 200 OK
< Content-Type: application/hal+json

{ "_links" : {
    "orders" : {
      "href" : "http://localhost:8080/orders"
    }
  }
}
```

The property of the result document is an object in itself consisting of keys representing the relation type with nested link objects as specified in HAL.

3.2. The collection resource

Spring Data REST exposes a collection resource named after the uncapitalized, pluralized version of the domain class the exported repository is handling. Both the name of the resource and the path can be customized using the `@RepositoryRestResource` on the repository interface.

3.2.1. Supported HTTP Methods

Collections resources support both `GET` and `POST`. All other HTTP methods will cause a `405 Method Not Allowed`.

GET

Returns all entities the repository servers through its `findAll( )` method. If the repository is a paging repository we include the pagination links if necessary and additional page metadata.

Parameters

If the repository has pagination capabilities the resource takes the following parameters:

- `page` - the page number to access (0 indexed, defaults to 0).
- `size` - the page size requested (defaults to 20).
- `sort` - a collection of sort directives in the format `($propertyname,)+[asc|desc]`?.

Custom status codes

- `405 Method Not Allowed` - if the `findAll( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

Supported media types

- application/hal+json
- application/json

Related resources

- `search` - a search resource if the backing repository exposes query methods.

HEAD

Returns whether the collection resource is available.

POST

Creates a new entity from the given request body.

Custom status codes

- `405 Method Not Allowed` - if the `save( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

Supported media types

- application/hal+json
- application/json

3.3. The item resource

Spring Data REST exposes a resource for individual collection items as sub-resources of the collection resource.

3.3.1. Supported HTTP methods

Item resources generally support `GET`, `PUT`, `PATCH` and `DELETE` unless explicit configuration prevents that (see below for details).

GET

Returns a single entity.

Custom status codes

- `405 Method Not Allowed` - if the `findOne( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

Supported media types

- application/hal+json
- application/json

Related resources

For every association of the domain type we expose links named after the association property. This can be customized by using `@RestResource` on the property. The related resources are of type association resource.

HEAD

Returns whether the item resource is available.

PUT

Replaces the state of the target resource with the supplied request body.

Custom status codes

- `405 Method Not Allowed` - if the `save( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

Supported media types

- application/hal+json
- application/json

PATCH

Similar to `PUT` but partially updating the resources state.

Custom status codes

- `405 Method Not Allowed` - if the `save( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

Supported media types

- application/hal+json
- application/json
- application/patch+json
- application/merge-patch+json

DELETE

Deletes the resource exposed.

Custom status codes

- `405 Method Not Allowed` - if the `delete( )` methods was not exported (through `@RestResource(exported = false)`) or is not present in the repository at all.

3.4. The association resource

Spring Data REST exposes sub-resources of every item resource for each of the associations the item resource has. The name and path of the of the resource defaults to the name of the association property and can be customized using `@RestResource` on the association property.

3.4.1. Supported HTTP methods

GET

Returns the state of the association resource

Supported media types

- application/hal+json
- application/json

PUT

Binds the resource pointed to by the given URI(s) to the resource. This

Custom status codes

- `400 Bad Request` - if multiple URIs were given for a to-one-association.

Supported media types

- text/uri-list - URIs pointing to the resource to bind to the association.

POST

Only supported for collection associations. Adds a new element to the collection.

Supported media types

- text/uri-list - URIs pointing to the resource to add to the association.

DELETE

Unbinds the association.

Custom status codes

- `405 Method Not Allowed` - if the association is non-optional.

3.5. The search resource

The search resource returns links for all query methods exposed by a repository. The path and name of the query method resources can be modified using `@RestResource` on the method declaration.

3.5.1. Supported HTTP methods

As the search resource is a read-only resource it supports `GET` only.

GET

Returns a list of links pointing to the individual query method resources

Supported media types

- application/hal+json
- application/json

Related resources

For every query method declared in the repository we expose a query method resource. If the resource supports pagination, the URI pointing to it will be a URI template containing the pagination parameters.

HEAD

Returns whether the search resource is available. A 404 return code indicates no query method resources available at all.

3.6. The query method resource

The query method resource executes the query exposed through an individual query method on the repository interface.

3.6.1. Supported HTTP methods

As the search resource is a read-only resource it supports `GET` only.

GET

Returns the result of the query execution.

Parameters

If the query method has pagination capabilities (indicated in the URI template pointing to the resource) the resource takes the following parameters:

- `page` - the page number to access (0 indexed, defaults to 0).
- `size` - the page size requested (defaults to 20).
- `sort` - a collection of sort directives in the format `($propertyname,)+[asc|desc]`?.

Supported media types

- application/hal+json
- application/json

HEAD

Returns whether a query method resource is available.

Chapter 4. Domain Object Representations

4.1. Object Mapping

Spring Data REST returns a representation of a domain object that corresponds to the requested `Accept` type specified in the HTTP request.

Currently, only JSON representations are supported. Other representation types can be supported in the future by adding an appropriate converter and updating the controller methods with the appropriate content-type.

Sometimes the behavior of the Spring Data REST's ObjectMapper, which has been specially configured to use intelligent serializers that can turn domain objects into links and back again, may not handle your domain model correctly. There are so many ways one can structure your data that you may find your own domain model isn't being translated to JSON correctly. It's also sometimes not practical in these cases to try and support a complex domain model in a generic way. Sometimes, depending on the complexity, it's not even possible to offer a generic solution.

4.1.1. Adding custom (de)serializers to Jackson's ObjectMapper

To accommodate the largest percentage of use cases, Spring Data REST tries very hard to render your object graph correctly. It will try and serialize unmanaged beans as normal POJOs and it will try and create links to managed beans where that's necessary. But if your domain model doesn't easily lend itself to reading or writing plain JSON, you may want to configure Jackson's ObjectMapper with your own custom type mappings and (de)serializers.

Abstract class registration

One key configuration point you might need to hook into is when you're using an abstract class (or an interface) in your domain model. Jackson won't know by default what implementation to create for an interface. Take the following example:

```
@Entity
public class MyEntity {
  @OneToMany
  private List<MyInterface> interfaces;
}
```

In a default configuration, Jackson has no idea what class to instantiate when POSTing new data to the exporter. This is something you'll need to tell Jackson either through an annotation, or, more cleanly, by registering a type mapping using a `Module`.

To add your own Jackson configuration to the `ObjectMapper` used by Spring Data REST, override the `configureJacksonObjectMapper` method. That method will be passed an `ObjectMapper` instance that has a

special module to handle serializing and deserializing `PersistentEntity`s. You can register your own modules as well, like in the following example.

```
@Override
protected void configureJacksonObjectMapper(ObjectMapper objectMapper) {
    objectMapper.registerModule(new SimpleModule("MyCustomModule") {
        @Override
        public void setupModule(SetupContext context) {
            context.addAbstractTypeResolver(
                new SimpleAbstractTypeResolver().addMapping(MyInterface.class,
                                                            MyInterfaceImpl.class)
            );
        }
    });
}
```

Once you have access to the `SetupContext` object in your `Module`, you can do all sorts of cool things to configure Jackson's JSON mapping. You can read more about how `Module`s work on Jackson's wiki: http://wiki.fasterxml.com/JacksonFeatureModules

Adding custom serializers for domain types

If you want to (de)serialize a domain type in a special way, you can register your own implementations with Jackson's `ObjectMapper` and the Spring Data REST exporter will transparently handle those domain objects correctly. To add serializers, from your `setupModule` method implementation, do something like the following:

```
@Override
public void setupModule(SetupContext context) {
  SimpleSerializers serializers = new SimpleSerializers();
  SimpleDeserializers deserializers = new SimpleDeserializers();

  serializers.addSerializer(MyEntity.class, new MyEntitySerializer());
  deserializers.addDeserializer(MyEntity.class, new MyEntityDeserializer());

  context.addSerializers(serializers);
  context.addDeserializers(deserializers);
}
```

Chapter 5. Validation

There are two ways to register a `Validator` instance in Spring Data REST: wire it by bean name or register the validator manually. For the majority of cases, the simple bean name prefix style will be sufficient.

In order to tell Spring Data REST you want a particular `Validator` assigned to a particular event, you simply prefix the bean name with the event you're interested in. For example, to validate instances of the `Person` class before new ones are saved into the repository, you would declare an instance of a `Validator<Person>` in your `ApplicationContext` with the bean name "beforeCreatePersonValidator". Since the prefix "beforeCreate" matches a known Spring Data REST event, that validator will be wired to the correct event.

5.1. Assigning Validators manually

If you would rather not use the bean name prefix approach, then you simply need to register an instance of your validator with the bean whose job it is to invoke validators after the correct event. In your configuration that subclasses Spring Data REST's `RepositoryRestMvcConfiguration`, override the `configureValidatingRepositoryEventListener` method and call the `addValidator` method on the `ValidatingRepositoryEventListener`, passing the event you want this validator to be triggered on, and an instance of the validator.

```
@Override
protected void configureValidatingRepositoryEventListener
(ValidatingRepositoryEventListener v) {
  v.addValidator("beforeSave", new BeforeSaveValidator());
}
```

Chapter 6. Events

There are eight different events that the REST exporter emits throughout the process of working with an entity. Those are:

- BeforeCreateEvent
- AfterCreateEvent
- BeforeSaveEvent
- AfterSaveEvent
- BeforeLinkSaveEvent
- AfterLinkSaveEvent
- BeforeDeleteEvent
- AfterDeleteEvent

6.1. Writing an ApplicationListener

There is an abstract class you can subclass which listens for these kinds of events and calls the appropriate method based on the event type. You just override the methods for the events you're interested in.

```
public class BeforeSaveEventListener extends AbstractRepositoryEventListener {

  @Override public void onBeforeSave(Object entity) {
    ... logic to handle inspecting the entity before the Repository saves it
  }

  @Override public void onAfterDelete(Object entity) {
    ... send a message that this entity has been deleted
  }
}
```

One thing to note with this approach, however, is that it makes no distinction based on the type of the entity. You'll have to inspect that yourself.

6.2. Writing an annotated handler

Another approach is to use an annotated handler, which does filter events based on domain type.

To declare a handler, create a POJO and put the `@RepositoryEventHandler` annotation on it. This tells the `BeanPostProcessor` that this class needs to be inspected for handler methods.

Once it finds a bean with this annotation, it iterates over the exposed methods and looks for annotations that correspond to the event you're interested in. For example, to handle BeforeSaveEvents in an annotated POJO for different kinds of domain types, you'd define your class like this:

```
@RepositoryEventHandler
public class PersonEventHandler {

  @HandleBeforeSave(Person.class) public void handlePersonSave(Person p) {
    ... you can now deal with Person in a type-safe way
  }

  @HandleBeforeSave(Profile.class) public void handleProfileSave(Profile p) {
    ... you can now deal with Profile in a type-safe way
  }
}
```

You can also declare the domain type at the class level:

```
@RepositoryEventHandler(Person.class)
public class PersonEventHandler {

  @HandleBeforeSave public void handleBeforeSave(Person p) {
    ...
  }

  @HandleAfterDelete public void handleAfterDelete(Person p) {
    ...
  }
}
```

Just declare an instance of your annotated bean in your `ApplicationContext` and the `BeanPostProcessor` that is by default created in `RepositoryRestMvcConfiguration` will inspect the bean for handlers and wire them to the correct events.

```
@Configuration
public class RepositoryConfiguration {

  @Bean PersonEventHandler personEventHandler() {
    return new PersonEventHandler();
  }
}
```