# Spring Data REST Reference Documentation

Jon Brisbin, Oliver Gierke, Greg Turnquist

Version 2.3.0.RC1
2015-03-05

# Table of Contents

# Preface

# Chapter 1. Project metadata

- Version control - https://github.com/spring-projects/spring-data-rest
- Bugtracker - https://jira.spring.io/browse/DATAREST
- Project page - http://projects.spring.io/spring-data-rest
- Release repository - https://repo.spring.io/libs-release
- Milestone repository - https://repo.spring.io/libs-milestone
- Snapshot repository - https://repo.spring.io/libs-snapshot

# Reference Documentation

# Chapter 2. 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 3. Getting started

## 3.1. 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.

## 3.2. Adding Spring Data REST to a Spring Boot project

The simplest way to get to started is if you are building a Spring Boot application. That's because Spring Data REST has both a starter as well as auto-configuration.

*Spring Boot configuration with Gradle*

```
dependencies {
  ...
  compile("org.springframework.boot:spring-boot-starter-data-rest")
  ...
}
```

*Spring Boot configuration with Maven*

```
<dependencies>
  ...
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
  </dependency>
  ...
</dependencies>
```

**NOTE** You don't have to supply the version number if you are using the Spring Boot Gradle plugin or the Spring Boot Maven plugin.

## 3.3. 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:2.3.0.RC1")
}
```

## 3.4. 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>2.3.0.RC1</version>
</dependency>
```

## 3.5. 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.

| | |
|---|---|
| **IMPORTANT** | This step is unnecessary if you are using Spring Boot's auto-configuration. Spring Boot will automatically enable Spring Data REST when you include **spring-boot-starter-data-rest** and either in your list of dependencies, and you your app is flagged with either `@SpringBootApplication` or `@EnableAutoConfiguration`. |

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.

## 3.6. Starting the application

At this point, you must also configure your key data store.

Spring Data REST officially supports:

- Spring Data JPA
- Spring Data MongoDB
- Spring Data Neo4j
- Spring Data GemFire

These linked guides introduce how to add dependencies for the related data store, configure domain objects, and define repositories.

You can run your application as either a Spring Boot app (with links showns above) or configure it as a classic Spring MVC app.

From this point, you can are free to customize Spring Data REST with various options.

# Chapter 4. Repository resources

## 4.1. 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.

### 4.1.1. 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`.

### 4.1.2. 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.

# 4.2. 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.

## 4.2.1. 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

# 4.3. The item resource

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

## 4.3.1. 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.

## 4.4. 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.

### 4.4.1. 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.

# 4.5. 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.

## 4.5.1. 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.

# 4.6. The query method resource

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

## 4.6.1. 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 5. Paging and Sorting

*This documents Spring Data REST's usage of the Spring Data Repository paging and sorting abstractions. To familiarize yourself with those features, please see the Spring Data documentation for the Repository implementation you're using.*

## 5.1. Paging

Rather than return everything from a large result set, Spring Data REST recognizes some URL parameters that will influence the page size and starting page number.

If you extend `PagingAndSortingRepository<T, ID>` and access the list of all entities, you'll get links to the first 20 entities. To set the page size to any other number, add a `size` parameter:

```
http://localhost:8080/people/?size=5
```

This will set the page size to 5.

To use paging in your own query methods, you need to change the method signature to accept an additional `Pageable` parameter and return a `Page` rather than a `List`. For example, the following query method will be exported to `/people/search/nameStartsWith` and will support paging:

```
@RestResource(path = "nameStartsWith", rel = "nameStartsWith")
public Page findByNameStartsWith(@Param("name") String name, Pageable p);
```

The Spring Data REST exporter will recognize the returned `Page` and give you the results in the body of the response, just as it would with a non-paged response, but additional links will be added to the resource to represent the previous and next pages of data.

### 5.1.1. Previous and Next Links

Each paged response will return links to the previous and next pages of results based on the current page using the IANA defined link relations `prev` and `next`. If you are currently at the first page of results, however, no `prev` link will be rendered. The same is true for the last page of results: no `next` link will be rendered.

Look at the following example, where we set the page size to 5:

```
curl localhost:8080/people?size=5
```

```
{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons{&sort,page,size}", <1>
      "templated" : true
    },
    "next" : {
      "href" : "http://localhost:8080/persons?page=1&size=5{&sort}", <2>
      "templated" : true
    }
  },
  "_embedded" : {
    ... data ...
  },
  "page" : { <3>
    "size" : 5,
    "totalElements" : 50,
    "totalPages" : 10,
    "number" : 0
  }
}
```

At the top, we see `_links`:

① This `self` link serves up the whole collection with some options

② This `next` link points to the next page, assuming the same page size.

③ At the bottom is extra data about the page settings, including the size of a page, total elements, total pages, and the page number you are currently viewing.

| NOTE | When using tools like **curl** on the command line, if you have a "&" in your statement, wrap the whole URI with quotes. |
|---|---|

It's also important to notice that the `self` and `next` URIs are, in fact, URI templates. They accept not only `size`, but also `page`, `sort` as optional flags.

As mentioned, at the bottom of the HAL document, is a collection of details about the page. This extra information makes it very easy for you to configure UI tools like sliders or indicators to reflect overall position the user is in viewing the data. For example, the document above shows we are looking at the first page (with page numbers indexed to 0 being the first).

What happens if we follow the `next` link?

```
$ curl "http://localhost:8080/persons?page=1&size=5"
```

```
{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons{&sort,projection,page,size}",
      "templated" : true
    },
    "next" : {
      "href" : "http://localhost:8080/persons?page=2&size=5{&sort,projection}", <1>
      "templated" : true
    },
    "prev" : {
      "href" : "http://localhost:8080/persons?page=0&size=5{&sort,projection}", <2>
      "templated" : true
    }
  },
  "_embedded" : {
... data ...
  },
  "page" : {
    "size" : 5,
    "totalElements" : 50,
    "totalPages" : 10,
    "number" : 1 <3>
  }
}
```

This looks very similar, except for the following differences:

① The `next` link now points to yet another page, indicating it's relative perspective to the `self` link.

② A `prev` link now appears, giving us a path to the previous page.

③ The current number is now 1 (indicating the second page).

This feature makes it quite easy to map optional buttons on the screen to these hypermedia controls, hence allowing easy navigational features for the UI experience without having to hard code the URIs. In fact, the user can be empowered to pick from a list of page sizes, dynamically changing the content served, without having to rewrite the `next` and `prev controls at the top or bottom.

## 5.2. Sorting

Spring Data REST recognizes sorting parameters that will use the Repository sorting support.

To have your results sorted on a particular property, add a `sort` URL parameter with the name of the property you want to sort the results on. You can control the direction of the sort by appending a `,` to the the property name plus either `asc` or `desc`. The following would use the `findByNameStartsWith` query method defined on the `PersonRepository` for all `Person` entities with names starting with the letter "K"

and add sort data that orders the results on the `name` property in descending order:

```
curl -v "http://localhost:8080/people/search/nameStartsWith?name=K&sort=name,desc"
```

To sort the results by more than one property, keep adding as many `sort=PROPERTY` parameters as you need. They will be added to the `Pageable` in the order they appear in the query string.

# Chapter 6. Domain Object Representations

## 6.1. 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.

### 6.1.1. 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 7. Projections and Excerpts

Spring Data REST presents a default view of the domain model you are exporting. But sometimes, you may need to alter the view of that model for various reasons. In this section, you will learn how to define projections and excerpts to serve up simplified and reduced views of resources.

## 7.1. Projections

Look at the following domain model:

```
@Entity
public class Person {

  @Id @GeneratedValue
  private Long id;
  private String firstName, lastName;

  @OneToOne
  private Address address;

}
```

This `Person` has several attributes:

- `id` is the primary key
- `firstName` and `lastName` are data attributes
- `address` is a link to another domain object

Now assume we create a corresponding repository as follows:

```
interface PersonRepository extends CrudRepository<Person, Long> {}
```

By default, Spring Data REST will export this domain object including all of its attributes. `firstName` and `lastName` will be exported as the plain data objects that they are. There are two options regarding the `address` attribute. One option is to also define a repository for `Address` objects like this:

```
interface AddressRepository extends CrudRepository<Address, Long> {}
```

In this situation, a `Person` resource will render the `address` attribute as a URI to it's corresponding `Address` resource. If we were to look up "Frodo" in the system, we could expect to see a HAL document

like this:

```
{
  "firstName" : "Frodo",
  "lastName" : "Baggins",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons/1"
    },
    "address" : {
      "href" : "http://localhost:8080/persons/1/address"
    }
  }
}
```

There is another route. If the `Address` domain object does not have it's own repository definition, Spring Data REST will inline the data fields right inside the `Person` resource.

```
{
  "firstName" : "Frodo",
  "lastName" : "Baggins",
  "address" : {
    "street": "Bag End",
    "state": "The Shire",
    "country": "Middle Earth"
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons/1"
    }
  }
}
```

But what if you don't want `address` details at all? Again, by default, Spring Data REST will export all its attributes (except the `id`). You can offer the consumer of your REST service an alternative by defining one or more projections.

```
@Projection(name = "noAddresses", types = { Person.class }) <1>
interface NoAddresses { <2>

  String getFirstName(); <3>

  String getLastName(); <4>
}
```

This projection has the following details:

① The `@Projection` annotation flags this as a projection. The `name` attributes provides the name of the projection, which you'll see how to use shortly. The `types` attributes targets this projection to only apply to `Person` objects.

② It's a Java interface making it declarative.

③ It exports the `firstName`.

④ It exports the `lastName`.

The `NoAddresses` projection only has getters for `firstName` and `lastName` meaning that it won't serve up any address information. Assuming you have a separate repository for `Address` resources, the default view from Spring Data REST is slightly different as shown below:

```
{
  "firstName" : "Frodo",
  "lastName" : "Baggins",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons/1{?projection}", <1>
      "templated" : true <2>
    },
    "address" : {
      "href" : "http://localhost:8080/persons/1/address"
    }
  }
}
```

① There is a new option listed for this resource, `{?projection}`.

② The `self` URI is a URI Template.

To view apply the projection to the resource, look up `http://localhost:8080/persons/1/projection=noAddresses`.

**NOTE** The value supplied to the `projection` query parameter is the same as specified in `@Projection(name = "noAddress")`. It has nothing to do with the name of the projection's interface.

It's possible to have multiple projections.

### 7.1.1. Finding existing projections

Spring Data REST provides hypermedia metadata by exposing Application-Level Profile Semantics (ALPS) documents, a micro metadata format. To view the ALPS metadata, follow the `profile` link exposed by the root resource. If you navigate down to the ALPS document for `Person` resources (which would be `/alps/persons`), you can find many details about `Person` resources. Projections will be listed

along with the details about the `GET` REST transitions, something like this:

```
{
  "id" : "get-person", <1>
  "name" : "person",
  "type" : "SAFE",
  "rt" : "#person-representation",
  "descriptors" : [ {
    "name" : "projection", <2>
    "doc" : {
      "value" : "The projection that shall be applied when rendering the response.
Acceptable values available in nested descriptors.",
      "format" : "TEXT"
    },
    "type" : "SEMANTIC",
    "descriptors" : [ {
      "name" : "noAddresses", <3>
      "type" : "SEMANTIC",
      "descriptors" : [ {
        "name" : "firstName", <4>
        "type" : "SEMANTIC"
      }, {
        "name" : "lastName", <4>
        "type" : "SEMANTIC"
      } ]
    } ]
  } ]
},
```

① This part of the ALPS document shows details about `GET` and `Person` resources.

② Further down are the `projection` options.

③ Further down you can see projection `noAddresses` listed.

④ The actual attributes served up by this projection include `firstName` and `lastName`.

### 7.1.2. Bringing in hidden data

So far, you have seen how projections can be used to reduce the information that is presented to the user. Projections can also bring in normally unseen data. For example, Spring Data REST will ignore fields or getters that are marked up with `@JsonIgnore` annotations. Look at the following domain object:

```
@Entity
public class User {

@Id @GeneratedValue
private Long id;
private String name;

@JsonIgnore <1>
private String password;
private String[] roles;
```

① Jackson's `@JsonIgnore` is used to prevent the `password` field from getting serialized into JSON.

This `User` class can be used to store user information as well as integration with Spring Security. If you create a `UserRepository`, the `password` field would normally have been exported. Not good! In this example, we prevent that from happening by applying Jackson's `@JsonIgnore` on the `password` field.

| NOTE | Jackson will also not serialize the field into JSON if `@JsonIgnore` is on the field's corresponding getter function. |
|---|---|

However, projections introduce the ability to still serve this field. It's possible to create a projection like this:

```
@Projection(name = "passwords", types = { User.class }) <2>
interface PasswordProjection {

  String getPassword();
}
```

If such a projection is created and used, it will side step the `@JsonIgnore` directive placed on `User.password`.

| IMPORTANT | This example may seem a bit contrived, but it's possible with a richer domain model and many projections, to accidentally leak such details. Since Spring Data REST cannot discern the sensitivity of such data, it is up to the developers to avoid such situations. |
|---|---|

## 7.2. Excerpts

An excerpt is a projection that is applied to a repository automatically. For an example, you can alter the `PersonRepository` as follows:

```
@RepositoryRestResource(excerptProjection = NoAddresses.class)
interface PersonRepository extends CrudRepository<Person, Long> {}
```

This directs Spring Data REST to use the `NoAddresses` projection when embedding `Person` resources into collections or related resources.

**NOTE** Excerpt projections do NOT apply when rendering a single resource.

In addition to altering the default rendering, excerpts have additional rednering options as shown below.

## 7.3. Excerpting commonly accessed data

A common situation with REST services arises when you compose domain objects. For example, a `Person` is stored in one table and their related `Address` is stored in another. By default, Spring Data REST will serve up the person's `address` as a URI the client must navigate. But if it's common for consumers to always fetch this extra piece of data, an excerpt projection can go ahead and inline this extra piece of data, saving you an extra `GET`. To do so, let's define another excerpt projection:

```
@Projection(name = "inlineAddress", types = { Person.class }) <1>
interface InlineAddress {

  String getFirstName();

  String getLastName();

  Address getAddress(); <2>
}
```

① This projection has been named `inlineAddress`.

② This projection adds in `getAddress` which returns the `Address` field. When used inside a projection, it causes the information to be inlined.

We can plug it into the `PersonRepository` definition as follows:

```
@RepositoryRestResource(excerptProjection = InlineAddress.class)
interface PersonRepository extends CrudRepository<Person, Long> {}
```

This will cause the HAL document to appear as follows:

```
{
  "firstName" : "Frodo",
  "lastName" : "Baggins",
  "address" : { <1>
    "street": "Bag End",
    "state": "The Shire",
    "country": "Middle Earth"
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons/1"
    },
    "address" : { <2>
      "href" : "http://localhost:8080/persons/1/address"
    }
  }
}
```

This should appear as a mix of what you've seen so far.

① The `address` data is inlined directly, so you don't have to navigate to get it.

② The link to the `Address` resource is still provided, making it still possible to navigate to its own resource.

| | |
|---|---|
| **WARNING** | Configuring `@RepositoryRestResource(excerptProjection= )` for a repository alters the default behavior. This can potentially case breaking change to consumers of your service if you have already made a release. Use with caution. |

# Chapter 8. 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.

## 8.1. 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 9. 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`

## 9.1. 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.

## 9.2. 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 `BeforeSaveEvent`s in an annotated POJO for different kinds of domain types, you'd define your class like this:

```
@RepositoryEventHandler
public class PersonEventHandler {

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

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

The domain type whose events you're interested in is determined from the type of the first parameter of the annotated methods.

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();
  }
}
```

# Chapter 10. Metadata

This section details the various forms of metadata provided by a Spring Data REST-based application.

## 10.1. Application-Level Profile Semantics (ALPS)

> ALPS is a data format for defining simple descriptions of application-level semantics, similar in complexity to HTML microformats. An ALPS document can be used as a profile to explain the application semantics of a document with an application-agnostic media type (such as HTML, HAL, Collection+JSON, Siren, etc.). This increases the reusability of profile documents across media types.
>
> — M. Admundsen / L. Richardson / M. Foster, http://tools.ietf.org/html/draft-amundsen-richardson-foster-alps-00

Spring Data REST provides an ALPS document for every exported repository. It contains information about both the RESTful transitions as well as the attributes of each repository.

At the root of a Spring Data REST app is a **profile** link. Assuming you had an app with both **persons** and related **address**, the root document would look like this:

```
{
  "_links" : {
    "persons" : {
      "href" : "http://localhost:8080/persons"
    },
    "addresses" : {
      "href" : "http://localhost:8080/addresses"
    },
    "profile" : {
      "href" : "http://localhost:8080/alps"
    }
  }
}
```

A **profile** link, as defined in RFC 6906, is a place to include application level details. The ALPS draft spec is meant to define a particular profile format which we'll explore further down in this section.

If you navigate into the **profile** link at `localhost:8080/alps`, you would see something like this:

```
{
  "version" : "1.0",
  "descriptors" : [ {
    "href" : "http://localhost:8080/alps/persons",
    "name" : "persons"
  }, {
    "href" : "http://localhost:8080/alps/addresses",
    "name" : "addresses"
  } ]
}
```

**IMPORTANT** At the root level, **profile** is a single link and hence can't handle serving up more than one application profile. That is why you must navigate to `/alps` to find a link for each resource's ALPS metadata.

**NOTE** This JSON document has a media type of `application/alps+json`. This is different than the previous JSON document, which had a media type of `application/hal+json`. These formats are different and governed by different specs.

Let's navigate to `/alps/persons` and look at the profile data for a `Person` resource.

```
{
  "version" : "1.0",
  "descriptors" : [ {
    "id" : "person-representation", <1>
    "descriptors" : [ {
      "name" : "firstName",
      "type" : "SEMANTIC"
    }, {
      "name" : "lastName",
      "type" : "SEMANTIC"
    }, {
      "name" : "id",
      "type" : "SEMANTIC"
    }, {
      "name" : "address",
      "type" : "SAFE",
      "rt" : "http://localhost:8080/addresses#address"
    } ]
  }, {
    "id" : "create-persons", <2>
    "name" : "persons", <3>
    "type" : "UNSAFE", <4>
    "rt" : "#person-representation" <5>
  }, {
    "id" : "get-persons",
    "name" : "persons",
    "type" : "SAFE",
    "rt" : "#person-representation"
  }, {
    "id" : "delete-person",
    "name" : "person",
    "type" : "IDEMPOTENT",
    "rt" : "#person-representation"
  }, {
    "id" : "patch-person",
    "name" : "person",
    "type" : "UNSAFE",
    "rt" : "#person-representation"
  }, {
    "id" : "update-person",
    "name" : "person",
    "type" : "IDEMPOTENT",
    "rt" : "#person-representation"
  }, {
    "id" : "get-person",
    "name" : "person",
    "type" : "SAFE",
```

```
    "rt" : "#person-representation"
  } ]
}
```

① At the top is a detailed listing of the attributes of a `Person` resource, identified as `#person-representation`. It lists the names of the attributes.

② After the resource representation are all the supported operations. This one is how to create a new `Person`.

③ The name is **persons**, which indicates that a POST should be applied to the whole collection, not a single **person**.

④ The **type** is `UNSAFE` because this operation can alter the state of the system. <5>

### 10.1.1. Hypermedia control types

ALPS displays types for each hypermedia control. They include:

*Table 1. ALPS types*

| Type | Description |
|---|---|
| SEMANTIC | A state element (e.g. HTML.SPAN, HTML.INPUT, etc.). |
| SAFE | A hypermedia control that triggers a safe, idempotent state transition (e.g. **GET** or **HEAD**). |
| IDEMPOTENT | A hypermedia control that triggers an unsafe, idempotent state transition (e.g. **PUT** or **DELETE**). |
| UNSAFE | A hypermedia control that triggers an unsafe, non-idempotent state transition (e.g. **POST**). |

In the representation section up above, bits of data from the application are marked **SEMANTIC**. The **address** field is a link that involves a safe **GET** to retrive. Hence, it is marked **SAFE**. Hypermedia operations themselves map onto the types as shown the table.

### 10.1.2. ALPS with Projections

If you define any projections, they are also listed in the ALPS metadata. Assuming we also defined **inlineAddress** and **noAddresses**, they would appear inside the relevant operations, i.e. **GET** for the whole collection as well **GET** for a single resource. The following shows the alternate version of the **get-persons** subsection:

```
...
  {
    "id" : "get-persons",
    "name" : "persons",
    "type" : "SAFE",
    "rt" : "#person-representation",
    "descriptors" : [ { <1>
      "name" : "projection",
      "doc" : {
        "value" : "The projection that shall be applied when rendering the response.
Acceptable values available in nested descriptors.",
        "format" : "TEXT"
      },
      "type" : "SEMANTIC",
      "descriptors" : [ {
        "name" : "inlineAddress", <2>
        "type" : "SEMANTIC",
        "descriptors" : [ {
          "name" : "address",
          "type" : "SEMANTIC"
        }, {
          "name" : "firstName",
          "type" : "SEMANTIC"
        }, {
          "name" : "lastName",
          "type" : "SEMANTIC"
        } ]
      }, {
        "name" : "noAddresses", <3>
        "type" : "SEMANTIC",
        "descriptors" : [ {
          "name" : "firstName",
          "type" : "SEMANTIC"
        }, {
          "name" : "lastName",
          "type" : "SEMANTIC"
        } ]
      } ]
    } ]
  }
...
```

① A new attribute, **descriptors**, appears containing an array with one entry, **projection**.

② Inside the **projection.descriptors** we can see **inLineAddress** listed. It will render **address**, **firstName**, and **lastName**. Relationships rendered inside a projection result in inlining the data fields.

③ Also found is **noAddresses**, which serves up a subset containing **firstName** and **lastName**.

With all this information, a client should be able to deduce not only the RESTful transitions avaiable, but also, to some degree, the data elements needed to interact.

### 10.1.3. Adding custom details to your ALPS descriptions

It's possible to create custom messages that appear in your ALPS metadata. Just create `rest-messages.properties` like this:

```
rest.description.person=A collection of people
rest.description.person.id=primary key used internally to store a person (not for RESTful
usage)
rest.description.person.firstName=Person's first name
rest.description.person.lastName=Person's last name
rest.description.person.address=Person's address
```

As you can see, this defines details to display for a `Person` resource. They alter the ALPS format of the **person-representation** as follows:

```
...
  {
    "id" : "person-representation",
    "doc" : {
      "value" : "A collection of people", <1>
      "format" : "TEXT"
    },
    "descriptors" : [ {
      "name" : "firstName",
      "doc" : {
        "value" : "Person's first name", <2>
        "format" : "TEXT"
      },
      "type" : "SEMANTIC"
    }, {
      "name" : "lastName",
      "doc" : {
        "value" : "Person's last name", <3>
        "format" : "TEXT"
      },
      "type" : "SEMANTIC"
    }, {
      "name" : "id",
      "doc" : {
        "value" : "primary key used internally to store a person (not for RESTful usage)
", <4>
        "format" : "TEXT"
      },
      "type" : "SEMANTIC"
    }, {
      "name" : "address",
      "doc" : {
        "value" : "Person's address", <5>
        "format" : "TEXT"
      },
      "type" : "SAFE",
      "rt" : "http://localhost:8080/addresses#address"
    } ]
  }
...
```

By supplying these property settings, each field has an extra **doc** attribute.

① The value of `rest.description.person` maps into the whole representation.

② The value of `rest.description.person.firstName` maps to the **firstName** attribute.

③ The value of `rest.description.person.lastName` maps to the **lastName** attribute.

④ The value of `rest.description.person.id` maps to the **id** attribute, a field not normally displayed.

⑤ The value of `rest.description.person.address` maps to the **address** attribute.

**NOTE** Spring MVC (which is the essence of a Spring Data REST application) supports locales, meaning you can bundle up multiple properties files with different messages.

# Chapter 11. Customizing Spring Data REST

There are many options to tailor Spring Data REST. These subsections will show how.

## 11.1. Configuring the REST URL path

Configuring the segments of the URL path under which the resources of a JPA repository are exported is simple. You just add an annotation at the class level and/or at the query method level.

By default, the exporter will expose your `CrudRepository` using the name of the domain class. Spring Data REST also applies the Evo Inflector to pluralize this word. So a repository defined as follows:

```
interface PersonRepository extends CrudRepository<Person, Long> {}
```

Will, by default, be exposed under the URL `http://localhost:8080/persons/`

To change how the repository is exported, add a `@RestResource` annotation at the class level:

```
@RestResource(path = "people")
interface PersonRepository extends CrudRepository<Person, Long> {}
```

Now the repository will be accessible under the URL: `http://localhost:8080/people/`

If you have query methods defined, those also default to be exposed by their name:

```
interface PersonRepository extends CrudRepository<Person, Long> {

  List<Person> findByName(String name);
}
```

This would be exposed under the URL: `http://localhost:8080/persons/search/findByName`

> **NOTE** All query method resources are exposed under the resource `search`.

To change the segment of the URL under which this query method is exposed, use the `@RestResource` annotation again:

```
@RestResource(path = "people")
interface PersonRepository extends CrudRepository<Person, Long> {

  @RestResource(path = "names")
  List<Person> findByName(String name);
}
```

Now this query method will be exposed under the URL: `http://localhost:8080/people/search/names`

### 11.1.1. Handling rels

Since these resources are all discoverable, you can also affect how the "rel" attribute is displayed in the links sent out by the exporter.

For instance, in the default configuration, if you issue a request to `http://localhost:8080/persons/search` to find out what query methods are exposed, you'll get back a list of links:

```
{
  "_links" : {
    "findByName" : {
      "href" : "http://localhost:8080/persons/search/findByName"
    }
  }
}
```

To change the rel value, use the `rel` property on the `@RestResource` annotation:

```
@RestResource(path = "people")
interface PersonRepository extends CrudRepository<Person, Long> {

  @RestResource(path = "names", rel = "names")
  List<Person> findByName(String name);
}
```

This would result in a link value of:

```
{
  "_links" : {
    "names" : {
      "href" : "http://localhost:8080/persons/search/names"
    }
  }
}
```

**NOTE** These snippets of JSON assume you are using Spring Data REST's default format of HAL. It's possible to turn off HAL, which would cause the output to look different. But your ability to override rel names is totally independent of the rendering format.

```
@RestResource(path = "people", rel = "people")
interface PersonRepository extends CrudRepository<Person, Long> {

  @RestResource(path = "names", rel = "names")
  List<Person> findByName(String name);
}
```

Altering the rel of a Repository changes the top level name:

```
{
  "_links" : {
    "people" : {
      "href" : "http://localhost:8080/people"
    },

  }
}
```

In the top level fragment above:

- `path = "people"` changed the value in `href` from `/persons` to `/people`
- `rel = "people"` changed the name of that link from `persons` to `people`

When you navigate to the **search** resource of this repository, the finder-method's `@RestResource` annotation has altered the path as shown below:

```
{
  "_links" : {
    "names" : {
      "href" : "http://localhost:8080/people/search/names"
    }
  }
}
```

This collection of annotations in your Repository definition has caused the following changes:

- The Repository-level annotation's `path = "people"` is reflected in the base URI with `/people`
- Being a finder method provides you with `/people/search`
- `path = "names"` creates a URI of `/people/search/names`
- `rel = "names"` changes the name of that link from `findByNames` to `names`

### 11.1.2. Hiding certain repositories, query methods, or fields

You may not want a certain repository, a query method on a repository, or a field of your entity to be exported at all. Examples include hiding fields like `password` on a `User` object or similar sensitive data. To tell the exporter to not export these items, annotate them with `@RestResource` and set `exported = false`.

For example, to skip exporting a Repository:

```
@RestResource(exported = false)
interface PersonRepository extends CrudRepository<Person, Long> {}
```

To skip exporting a query method:

```
@RestResource(path = "people", rel = "people")
interface PersonRepository extends CrudRepository<Person, Long> {

  @RestResource(exported = false)
  List<Person> findByName(String name);
}
```

Or to skip exporting a field:

```
@Entity
public class Person {

  @Id @GeneratedValue private Long id;

  @OneToMany
  @RestResource(exported = false)
  private Map<String, Profile> profiles;
}
```

| WARNING | Projections provide the means to change what is exported and effectively side step these settings. If you create any projections against the same domain object, it's your responsiblity to NOT export the fields. |
|---|---|

### 11.1.3. Hiding repository CRUD methods

If you don't want to expose a save or delete method on your `CrudRepository`, you can use the `@RestResource(exported = false)` setting by overriding the method you want to turn off and placing the annotation on the overriden version. For example, to prevent HTTP users from invoking the delete methods of `CrudRepository`, override all of them and add the annotation to the overriden methods.

```
@RestResource(path = "people", rel = "people")
interface PersonRepository extends CrudRepository<Person, Long> {

  @Override
  @RestResource(exported = false)
  void delete(Long id);

  @Override
  @RestResource(exported = false)
  void delete(Person entity);
}
```

| WARNING | It is important that you override *both* delete methods as the exporter currently uses a somewhat naive algorithm for determing which CRUD method to use in the interest of faster runtime performance. It's not currently possible to turn off the version of delete which takes an ID but leave exported the version that takes an entity instance. For the time being, you can either export the delete methods or not. If you want turn them off, then just keep in mind you have to annotate both versions with `exported = false`. |
|---|---|

## 11.2. Adding Spring Data REST to an existing Spring MVC Application

If you have an existing Spring MVC application and you'd like to integrate Spring Data REST, it's actually very easy.

Somewhere in your Spring MVC configuration (most likely where you configure your MVC resources) add a bean reference to the JavaConfig class that is responsible for configuring the `RepositoryRestController`. The class name is `org.springframework.data.rest.webmvc.RepositoryRestMvcConfiguration`.

In Java, this would look like:

```
import org.springframework.context.annotation.Import;
import org.springframework.data.rest.webmvc.RepositoryRestMvcConfiguration;

@Configuration
@Import(RepositoryRestMvConfiguration.class)
public class MyApplicationConfiguration {


}
```

In XML this would look like:

```
<bean class="org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration
"/>
```

When your ApplicationContext comes across this bean definition it will bootstrap the necessary Spring MVC resources to fully-configure the controller for exporting the repositories it finds in that `ApplicationContext` and any parent contexts.

### 11.2.1. More on required configuration

There are a couple Spring MVC resources that Spring Data REST depends on that must be configured correctly for it to work inside an existing Spring MVC application. We've tried to isolate those resources from whatever similar resources already exist within your application, but it may be that you want to customize some of the behavior of Spring Data REST by modifying these MVC components.

The most important things that we configure especially for use by Spring Data REST include:

**RepositoryRestHandlerMapping**

We register a custom `HandlerMapping` instance that responds only to the `RepositoryRestController` and only if a path is meant to be handled by Spring Data REST. In order to keep paths that are meant to be handled by your application separate from those handled by Spring Data REST, this custom `HandlerMapping` inspects the URL path and checks to see if a Repository has been exported under that name. If it has, it allows the request to be handled by Spring Data REST. If there is no Repository exported under that name, it returns `null`, which just means "let other `HandlerMapping` instances try to service this request".

The Spring Data REST `HandlerMapping` is configured with `order=(Ordered.LOWEST_PRECEDENCE - 100)` which means it will usually be first in line when it comes time to map a URL path. Your existing application will never get a chance to service a request that is meant for a repository. For example, if you have a repository exported under the name "person", then all requests to your application that start with `/person` will be handled by Spring Data REST and your application will never see that request. If your repository is exported under a different name, however (like "people"), then requests to `/people` will go to Spring Data REST and requests to "/person" will be handled by your application.

# 11.3. Customizing the JSON output

Sometimes in your application you need to provide links to other resources from a particular entity. For example, a `Customer` response might be enriched with links to a current shopping cart, or links to manage resources related to that entity. Spring Data REST provides integration with Spring HATEOAS and provides an extension hook for users to alter the representation of resources going out to the client.

## 11.3.1. The ResourceProcessor interface

Spring HATEOAS defines a `ResourceProcessor<>` interface for processing entities. All beans of type `ResourceProcessor<Resource<T>>` will be automatically picked up by the Spring Data REST exporter and triggered when serializing an entity of type `T`.

For example, to define a processor for a `Person` entity, add a `@Bean` to your `ApplicationContext like the following (which is taken from the Spring Data REST tests):

```
@Bean
public ResourceProcessor<Resource<Person>> personProcessor() {

   return new ResourceProcessor<Resource<Person>>() {

     @Override
     public Resource<Person> process(Resource<Person> resource) {

      resource.add(new Link("http://localhost:8080/people", "added-link"));
      return resource;
     }
   };
}
```

| IMPORTANT | This example hard codes a link to `http://localhost:8080/people`. If you have a Spring MVC endpoint inside your app that you wish to link to, consider using Spring HATEOAS's `linkTo( )` method to avoid managing the URL. |
|---|---|

### 11.3.2. Adding Links

It's possible to add links to the default representation of an entity by simply calling `resource.add(Link)` like the example above. Any links you add to the `Resource` will be added to the final output.

### 11.3.3. Customizing the representation

The Spring Data REST exporter executes any discovered `ResourceProcessor`s before it creates the output representation. It does this by registering a `Converter<Entity, Resource>` instance with an internal `ConversionService`. This is the component responsible for creating the links to referenced entities (e.g. those objects under the **_links** property in the object's JSON representation). It takes an `@Entity` and iterates over its properties, creating links for those properties that are managed by a `Repository` and copying across any embedded or simple properties.

If your project needs to have output in a different format, however, it's possible to completely replace the default outgoing JSON representation with your own. If you register your own `ConversionService` in the ApplicationContext and register your own `Converter<Person, Resource>`, then you can return a `Resource` implementation of your choosing.

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

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.

So to accommodate the largest percentage of the 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.

### 11.4.1. 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.

Any `Module` bean declared within the scope of your `ApplicationContext` will be picked up by the exporter and registered with its `ObjectMapper`. To add this special abstract class type mapping, create a `Module` bean and in the `setupModule` method, add an appropriate `TypeResolver`:

```
public class MyCustomModule extends SimpleModule {

  private MyCustomModule() {
    super("MyCustomModule", new Version(1, 0, 0, "SNAPSHOT"));
  }

  @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 Jackon's JSON mapping. You can read more about how Module`s work on Jackson's wiki.

### 11.4.2. 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:

```
public class MyCustomModule extends SimpleModule {



  @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);
  }
}
```

Now Spring Data REST will correctly handle your domain objects in case they are too complex for the 80% generic use case that Spring Data REST tries to cover.

# Appendix

# Appendix A: Using curl to talk to Spring Data REST

This appendix contains a list of guides that demonstrate interacting with a Spring Data REST service via curl:

- Accessing JPA Data with REST
- Accessing Neo4j Data with REST
- Accessing MongoDB Data with REST
- Accessing GemFire Data with REST