Spring Data REST Reference Documentation

2.0.0.M1

Jon Brisbin , Oliver Gierke

Table of Contents

1. Introduction

Spring Data REST makes exporting domain objects to RESTful clients using HATEOAS principles very easy. It exposes the CRUD methods of the Spring Data `CrudRepository` interface to HTTP. Spring Data REST also reads the body of HTTP requests and interprets them as domain objects. It recognizes relationships between entities and represents that relationship in the form of a `Link`.

1.1 HTTP Verb to CRUD Method Mapping

Spring Data REST translates HTTP calls to method calls by mapping the HTTP verbs to CRUD methods. The following table illustrates the way in which an HTTP verb is mapped to a `CrudRepository` method.

Table 1.1. HTTP verb to CRUD method mapping

Verb	Method
`GET`	`CrudRepository<ID,T>.findOne(ID id)`
`POST`	`CrudRepository<ID,T>.save(T entity)`
`PUT`	`CrudRepository<ID,T>.save(T entity)`
`DELETE`	`CrudRepository<ID,T>.delete(ID id)`

By default, all of these methods are exported to clients. By placing an annotation on your `CrudRepository` subinterface, however, you can turn access to a method off. This is discussed in more detail in the section on configuration.

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 number of ways to accomplish this but no real standard way. Spring Data REST uses a link method that is consistent across Spring Data REST: it provides links in a property called `links`. That property is an array of objects that take the form of something resembling an atom:link element in the Atom XML namespace.

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/json
<
{
  "links" : [ {
    "rel" : "customer",
    "href" : "http://localhost:8080/customer"
  }, {
    "rel" : "profile",
    "href" : "http://localhost:8080/profile"
  }, {
    "rel" : "order",
    "href" : "http://localhost:8080/order"
  }, {
    "rel" : "people",
    "href" : "http://localhost:8080/people"
  }, {
    "rel" : "product",
    "href" : "http://localhost:8080/product"
  } ],
  "content" : [ ]
}
```

The links property of the result document contains an array of objects that have rel and href properties on them.

Compact vs. Verbose

When issuing a request to a resource, the default behavior is to provide as much information as possible in the body of the response. In the case of accessing a `Repository` resource, Spring Data REST will inline entities into the body of the response. This could lead to poor network performance in the case of a very large number of entities. To reduce the amount of data sent back in the response, a user agent can request the special content type `application/x-spring-data-compact+json` by placing this in the request `Accept` header. Rather than inlining the entities, this content-type provides a link to each entity in the links property.

```
curl -v -H "Accept: application/x-spring-data-compact+json" http://localhost:8080/customer

< HTTP/1.1 200 OK
< Content-Type: application/x-spring-data-compact+json
<
{
  "links" : [ {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "content" : [ ]
}
```

2. Installing Spring Data REST

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 (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:1.1.0.M1"
}
```

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>1.1.0.M1</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
public class MyWebConfiguration extends RepositoryRestMvcConfiguration {

  @Override protected void
 configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {
    config.addResourceMappingForDomainType(Person.class)
          .addResourceMappingFor("lastName")
          .setPath("surname"); // Change 'lastName' to 'surname' in the JSON
    config.addResourceMappingForDomainType(Person.class)
          .addResourceMappingFor("siblings")
          .setRel("siblings")
          .setPath("siblings"); // Pointless in this example,
                                // but shows how to change 'rel' and 'path' values.
  }

}
```

There are numerous methods on the `RepositoryRestConfiguration` object to allow you to configure various aspects of Spring Data REST. Please read the javadoc for that class to get detailed descriptions of the various settings you can control.

Adding custom converters

It may be necessary to add a custom converter to Spring Data REST. You might need to turn a query parameter into a complex object, for instance. In that case, you'll want to override the `configureConversionService` method and add your own converters. To convert a query parameter to a complex object, for instance, you would want to register a converter for `String[]` to `MyPojo`.

```
  @Bean public MyPojoConverter myPojoConverter() {
    return new MyPojoConverter();
  }

  @Override protected void configureConversionService(ConfigurableConversionService
conversionService) {
    conversionService.addConverter(String[].class, myPojoConverter());
  }
```

2.5 Adding Spring Data REST to Spring MVC

Since Spring Data REST is simply a Spring MVC application, you only need to include the REST configuration into the configuration for the `DispatcherServlet`. If using a Servlet 3.0 `WebApplicationInitializer` (the preferred configuration for Spring Data REST applications), you would add your subclassed configuration from above into the configuration for the `DispatcherServlet`. The following configuration class is from the example project and includes datasource configuration for three different datastores and domain models, which will all be exported by Spring Data REST.

```
public class RestExporterWebInitializer implements WebApplicationInitializer {

  @Override public void onStartup(ServletContext servletContext) throws ServletException {
    AnnotationConfigWebApplicationContext rootCtx = new
 AnnotationConfigWebApplicationContext();
    rootCtx.register(
        JpaRepositoryConfig.class, // Include JPA entities, Repositories
        MongoDbRepositoryConfig.class, // Include MongoDB document entities, Repositories
        GemfireRepositoryConfig.class // Inlucde Gemfire entities, Repositories
    );
    servletContext.addListener(new ContextLoaderListener(rootCtx));

    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. You can use curl or, more easily, the `rest-shell`:

```
$ rest-shell


 ___ ___  __ _____  __  _  _     _ _  __
| _ \ __/' _/_   _/' _/| || |   / / | \ \
| v / _|`._`. | | `._`.| >< |  / / /   > >
|_|_\___|___/ |_| |___/|_||_| |_/_/   /_/
1.2.1.RELEASE

Welcome to the REST shell. For assistance hit TAB or type "help".
http://localhost:8080:> list
rel         href
============================================
people      http://localhost:8080/people
profile     http://localhost:8080/profile
customer    http://localhost:8080/customer
order       http://localhost:8080/order
product     http://localhost:8080/product
```

3. Domain Object Representations

3.1 Links as First-Class Objects

Links are an essential part of RESTful resources and allow for easy discoverability of related resources. In Spring Data REST, a link is represented in JSON as an object with a `rel` and `href` property. These objects will appear in an array under an object's `links` property. These objects are meant to provide a user agent with the URLs necessary to retrieve resources related to the current resource being accessed.

When accessing the root of a Spring Data REST application, for example, links are provided to each repository that is exported. The user agent can then pick the link it is interested in and follow that`href`. Issue a `get` in the `rest-shell` to see an example of links.

```
http://localhost:8080:> get
> GET http://localhost:8080/

< 200 OK
< Content-Type: application/json
<
{
  "links" : [ {
    "rel" : "people",
    "href" : "http://localhost:8080/people"
  }, {
    "rel" : "profile",
    "href" : "http://localhost:8080/profile"
  }, {
    "rel" : "customer",
    "href" : "http://localhost:8080/customer"
  }, {
    "rel" : "order",
    "href" : "http://localhost:8080/order"
  }, {
    "rel" : "product",
    "href" : "http://localhost:8080/product"
  } ],
  "content" : [ ]
}
```

Entity Relationships

If two entities are related to one another through a database-defined relationship, then that relationship will appear in the JSON as a link. In JPA, one would place a `@ManyToOne`, `@OneToOne`, or other relationship annotation. If using Spring Data MongoDB, one would place a `@DBRef` annotation on a property to denote its special status as a reference to other entities. In the example project, the `Person` class has a related set of `Person` entities in the `siblings` property. If you `get` the resource of a `Person` you will see, in the `siblings` property, the link to follow to get the related `Persons`.

```
http://localhost:8080:> get people/1
> GET http://localhost:8080/people/1

< 200 OK
< Content-Type: application/json
<
{
  "firstName" : "Billy Bob",
  "surname" : "Thornton",
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/people/1"
  }, {
    "rel" : "people.person.father",
    "href" : "http://localhost:8080/people/1/father"
  }, {
    "rel" : "people.person.siblings",
    "href" : "http://localhost:8080/people/1/siblings"
  } ]
}
```

3.2 Object Mapping

Spring Data REST returns a representation of a domain object that corresponds to the requested `Accept` type specified in the HTTP request. [1]

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.

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;
}
```

[1] 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.

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 `PersistentEntitys`. 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 Jacskon's JSON mapping. You can read more about how `Modules` 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);
}
```

4. Paging and Sorting

4.1 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. To add paging support to your Repositories, you need to extend the `PagingAndSortingRepository<T,ID>` interface rather than the basic `CrudRepository<T,ID>` interface. This adds methods that accept a `Pageable` to control the number and page of results returned.

```
public Page findAll(Pageable pageable);
```

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 limit parameter:

```
http://localhost:8080/people/?limit=50
```

To get paging in your query methods, you must change the signature of your query methods to accept a `Pageable` as a parameter and return a `Page<T>` rather than a `List<T>`. Otherwise, you won't get any paging information in the JSON and specifying the query parameters that control paging will have no effect.

By default, the URL query parameters recognized are `page`, to specify page number `limit`, to specify how many results to return on a page, and `sort` to specify the query method parameter on which to sort. To change the names of the query parameters, simply call the appropriate method on `RepositoryRestConfiguration` and give it the text you would like to use for the query parameter. The following, for example, would set the paging parameter to `p`, the limit parameter to `l`, and the sort parameter to `q`:

```
@Override protected void configureRepositoryRestConfiguration(RepositoryRestConfiguration
 config) {
  config.setPageParamName("p")
        .setLimitParamName("l")
        .setSortParamName("q");
}
```

The URL to use these parameters would then be changed to:

```
http://localhost:8080/people/?p=2&l=50
```

Previous and Next Links

Each paged response will return links to the previous and next pages of results based on the current page. If you are currently at the first page of results, however, no "previous" link will be rendered. The same is true for the last page of results: no "next" link will be rendered if you are on the last page of results. The "rel" value of the link will end with ".next" for next links and ".prev" for previous links.

```
{
  "rel" : "people.next",
  "href" : "http://localhost:8080/people?page=2&limit=20"
}
```

4.2 Sorting

Spring Data REST also 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 specifying a URL parameter composed of the property name plus `.dir` and setting that value to 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&name.dir=desc
```

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 who's 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());
}
```

6. Events

There are six 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();
  }

}
```

7. Using the rest-shell

The **rest-shell** is a command-line shell that aims to make writing REST-based applications easier. It is based on spring-shell and integrated with Spring HATEOAS in such a way that REST resources that output JSON compliant with Spring HATEOAS can be discovered by the shell and interactions with the REST resources become much easier than by manipulating the URLs in bash using a tool like **curl**.

The rest-shell provides a number of useful commands for discovering and interacting with REST resources. For example discover will discover what resources are available and print out an easily-readable table of rels and URIs that relate to those resources. Once these resources have been discovered, the rel of those URIs can be used in place of the URI itself in most operations, thus cutting down on the amount of typing needed to issue HTTP requests to your REST resources.

7.1 Installing the rest-shell

If you're using Mac OS X and Homebrew, then installation is super easy:

```
brew install rest-shell
```

Other platforms are simple as well: just download the archive from the GitHub page and unzip it to a location on your local hard drive.

7.2 Discovering resources

The rest-shell is aimed at making it easier to interact with REST resources by managing the session baseUri much like a directory in a filesystem. Whenever resources are discovered, you can then follow to a new baseUri, which means you can then use relative URIs. Here's an example of discovering resources, then following a link by referencing its rel value, and then using a relative URI to access resources under that new baseUri:

```
http://localhost:8080:> discover
rel                href
==========================================================
address            http://localhost:8080/address
family             http://localhost:8080/family
people             http://localhost:8080/person
profile            http://localhost:8080/profile

http://localhost:8080:> follow people
http://localhost:8080/person:> list
rel             href
=================================================
people.Person    http://localhost:8080/person/1
people.Person    http://localhost:8080/person/2
people.search    http://localhost:8080/person/search

http://localhost:8080/person:> get 1
> GET http://localhost:8080/person/1

< 200 OK
< ETag: "2"
< Content-Type: application/json
<
{
    "links" : [ {
        "rel" : "self",
        "href" : "http://localhost:8080/person/1"
    }, {
        "rel" : "peeps.Person.profiles",
        "href" : "http://localhost:8080/person/1/profiles"
    }, {
        "rel" : "peeps.Person.addresses",
        "href" : "http://localhost:8080/person/1/addresses"
    } ],
    "name" : "John Doe"
}
```

NOTE: If you want tab completion of discovered rels, just use the --rel flag.

7.3 Creating new resources

The rest-shell can do basic parsing of JSON data within the shell (though there are some limitations due to the nature of the command line parsing being sensitive to whitespace). This makes it easy to create new resources by including JSON data directly in the shell:

```
http://localhost:8080/person:> post --data "{name: 'John Doe'}"
> POST http://localhost:8080/person/

< 201 CREATED
< Location: http://localhost:8080/person/8
< Content-Length: 0
<

http://localhost:8080/person:> get 8
> GET http://localhost:8080/person/8

< 200 OK
< ETag: "0"
< Content-Type: application/json
<
{
    "links" : [ {
        "rel" : "self",
        "href" : "http://localhost:8080/person/8"
    }, {
        "rel" : "people.Person.addresses",
        "href" : "http://localhost:8080/person/8/addresses"
    }, {
        "rel" : "people.Person.profiles",
        "href" : "http://localhost:8080/person/8/profiles"
    } ],
    "name" : "John Doe"
}
```

If your needs of representing JSON get more complicated than what the spring-shell interface can handle, you can create a directory somewhere with .json files in it, one file per entitiy, and use the --from option to the post command. This will walk the directory and make a POST request for each .json file found.

```
http://localhost:8080/person:> post --from work/people_to_load
128 items uploaded to the server using POST.
http://localhost:8080/person:>
```

You can also reference a specific file rather than an entire directory.

```
http://localhost:8080/person:> post --from work/people_to_load/someone.json
1 items uploaded to the server using POST.
http://localhost:8080/person:>
```

7.4 Passing query parameters

If you're calling URLs that require query parameters, you'll need to pass those as a JSON-like fragment in the --params parameter to the get and list commands. Here's an example of calling a URL that expects parameter input:

```
http://localhost:8080/person:> get search/byName --params "{name: 'John Doe'}"
```

7.5 Outputing results to a file

It's not always desirable to output the results of an HTTP request to the screen. It's handy for debugging but sometimes you want to save the results of a request because they're not easily reproducible or any

number of other equally valid reasons. All the HTTP commands take an --output parameter that writes the results of an HTTP operation to the given file. For example, to output the above search to a file:

```
http://localhost:8080/person:> get search/byName --params "{name: 'John Doe'}" --output
 by_name.txt >> by_name.txt
http://localhost:8080/person:>
```

7.6 Sending complex JSON

Because the rest-shell uses the spring-shell underneath, there are limitations on the format of the JSON data you can enter directly into the command line. If your JSON is too complex for the simplistic limitations of the shell --data parameter, you can simply load the JSON from a file or from all the files in a directory.

When doing a post or put, you can optionally pass the --from parameter. The value of this parameter should either be a file or a directory. If the value is a directory, the shell will read each file that ends with .json and make a POST or PUT with the contents of that file. If the parameter is a file, then the rest-shell will simpy load that file and POST/PUT that data in that individual file.

7.7 Shelling out to bash

One of the nice things about spring-shell is that you can directly shell out commands to the underlying terminal shell. This is useful for doing things like load a JSON file in an editor. For instance, assume I have the Sublime Text 2 command subl in my path. I can then load a JSON file for editing from the rest-shell like this:

```
http://localhost:8080/person:> ! subl test.json
        http://localhost:8080/person:>
```

I then edit the file as I wish. When I'm ready to POST that data to the server, I can do so using the --from parameter:

```
http://localhost:8080/person:> post --from test.json
1 items uploaded to the server using POST.
http://localhost:8080/person:>
```

7.8 Setting context variables

Starting with rest-shell version 1.1, you can also work with context variables during your shell session. This is useful for saving settings you might reference often. The rest-shell now integrates Spring Expression Language support, so these context variables are usable in expressions within the shell.

```
http://localhost:8080/person:> var set --name specialUri --value http://
longdomainname.com/api
http://localhost:8080/person:> var get --name specialUri
http://longdomainname.com/api
http://localhost:8080/person:> var list
{
  "responseHeaders" : {
    ... HTTP headers from last request
  },
  "responseBody" : {
    ... Body from the last request
  },
  "specialUri" : "http://longdomainname.com/api",
  "requestUrl" : ... URL from the last request,
  "env" : {
    ... System properties and environment variables
  }
}
```

The variables are accessible from SpEL expressions which are valid in a number of different contexts, most importantly in the path argument to the HTTP and discover commands, and in the data argument to the put and post commands.

Since the rest-shell is aware of environment variables and system properties, you can incorporate external parameters into your interaction with the shell. For example, to externally define a baseUri, you could set a system property before invoking the shell. The shell will incorporate anything defined in the JAVA_OPTS environment variable, so you could parameterize your interaction with a REST service.

```
JAVA_OPTS="-DbaseUri=http://mylongdomain.com/api" rest-shell

http://localhost:8080:> discover #{env.baseUri}
rel                href
================================================================
... resources for this URL
http://mylongdomain.com/api:>
```

7.9 Per-user shell initialization

The rest-shell supports a "dotrc" type of initialization by reading in all files found in the $HOME/.rest-shell/ directory and assuming they have shell commands in them. The rest-shell will execute these commands on startup. This makes it easy to set variables for commonly-used URIs or possibly set a baseUri.

```
echo "var set --name svcuri --value http://api.myservice.com/v1" > ~/.rest-shell/00-vars
echo "discover #{svcuri}" > ~/.rest-shell/01-baseUri

> rest-shell

INFO: No resources found...
INFO: Base URI set to 'http://api.myservice.com/v1'


 ___ ___  __ _____  __  _  _     _ _  __
| _ \ __/' _/_   _/' _/| || |   / / | \ \
| v / _|`._`. | | `._`.| >< |  / / /   > >
|_|_\___|___/ |_| |___/|_||_| |_/_/   /_/
1.2.1.RELEASE

Welcome to the REST shell. For assistance hit TAB or type "help".
http://api.myservice.com/v1:>
```

7.10 SSL Certificate Validation

If you generate a self-signed certificate for your server, by default the rest-shell will complain and refuse to connect. This is the default behavior of RestTemplate. To turn off certificate and hostname checking, use the ssl validate --enabled false command.

7.11 HTTP Basic authentication

There is also a convenience command for setting an HTTP Basic authentication header. Use auth basic --username user --pasword passwd to set a username and password to base64 encode and place into the Authorization header that will be part of the current session's headers.

You can clear the authentication by using the auth clear command or by removing the Authorization header using the headers clear command.

7.12 Commands

The rest-shell provides the following commands:

Table 7.1. rest-shell commands

Command	Description
`baseUri` { *uri* }	Set the base URI used for this point forward in the session. Relative URIs will be calculated relative to this setting.
`discover` [[--rel *rel*] | [*path*]]	Find out what resources are available at the given URI. If no URI is given, use the baseUri.
`follow` [[--rel *rel*] | [*path*]]	Set the baseUri to the URI assigned to this given rel or path but do not discover resources.
`list` [[--rel *rel*] | [*path*]] [--params *JSON*]	Find out what resources are available at the given URI.
`headers set` {--name *name* } {--value *value* }	Set an HTTP header for use from this point forward in the session.
`headers clear`	Clear all HTTP headers set during this session.
`headers list`	Print out the currently-set HTTP headers for this session.
`history list`	List the URIs previously set as baseUris during this session.
`history go` [*num*]	Jump to a URI by pulling one from the history.
`var clear`	Clear this shell's variable context.
`var get` [--name *name*] [--value *expression*]	Get a variable from this shell's context by name or evaluate a shell expression.
`var list`	List variables currently set in this shell's context.

Command	Description
`var set`	Set a variable in this shell's context.
`up`	Traverse one level up in the URL hierarchy.
`get` [[--rel *`rel`*] | [*`path`*]] [--follow true | false] [--params *`JSON`*] [--output *`filename`*]	HTTP GET from the given path. If `--follow true` is set, then follow any redirects automatically. If `--output filename` is set, output the the response into the given file.
`post`	HTTP POST to the given path, passing JSON given in the --data parameter.
`put`	HTTP PUT to the given path, passing JSON given in the --data parameter.
`delete`	HTTP DELETE to the given path.
`auth basic`	Set an HTTP Basic authentication token for use in this session.
`auth clear`	Clear the Authorization header currently in use.
`ssl validate`	Disable certificate checking to work with self-signed certificates.