



g00glen00b (<http://g00glen00b.be/>)

Tutorials by IT enthusiasts, for enthusiasts.

(<http://g00glen00b.be/>)



(<http://twitter.com/g00glen00b>)



(<http://linkedin.com/in/g00glen00b>)



(<http://github.com/g00glen00b>)



(<http://srv.carbonads.net/adsegment=placement:g00glen00b>)

We connect highly qualified audiences with highly relevant services, products, and brands.

(<http://srv.carbonads.net/adsegment=placement:g00glen00b>)

ads via Carbon
(<http://carbonads.net/>)

JavaScript

([/category/t/javascript](http://category/t/javascript))

Building modern webapps using Spring Data REST and AngularJS

30/11/2014 / BY G00GLEN00B ([HTTP://G00GLEN00B.BE/AUTHOR/G00GLEN00B/](http://G00GLEN00B.BE/AUTHOR/G00GLEN00B/)) / 13 COMMENTS
([HTTP://G00GLEN00B.BE/SPRING-DATA-ANGULAR/#DISQUS_THREAD](http://G00GLEN00B.BE/SPRING-DATA-ANGULAR/#DISQUS_THREAD))

Recently I wrote several “exotic” applications using WebSockets. If you’re not into WebSockets, but you’re interested in using Spring and AngularJS, this article may suite you. In this example I will setup a web project using the Spring framework and an in memory embedded HSQL database. The client-side of the application will be written using AngularJS. Let’s start!

Project setup

I’m gonna start of with a simple Maven webproject, in my **pom.xml** I will have to add the following dependencies:

AngularJS
([/tag/angularjs](http://tag/angularjs))

Dojo Toolkit
([/tag/dojo](http://tag/dojo))

Ember.js
([/tag/ember-js](http://tag/ember-js))

Meteor
([/tag/meteor-js](http://tag/meteor-js))

Node.js ([/tag/node-js](http://tag/node-js))

Java

(/category/t/jav

Spring MVC

(/tag/spring-mvc)

Spring Security

(/tag/spring-security)

Testing (/tag/testing)

Web services

(/tag/web-services)

Coding

(/category/t/co

CSS (/tag/css)

Web semantics

(/tag/web-
semantics)

Other

(/category/t/otl

PHP (/tag/php)

Design principles

(/tag/coding-rules)

```
1. <dependency>
2.   <groupId>org.springframework</groupId>
3.   <artifactId>spring-webmvc</artifactId>
4.   <version>4.1.1.RELEASE</version>
5. </dependency>
6. <dependency>
7.   <groupId>org.springframework.data</groupId>
8.   <artifactId>spring-data-rest-webmvc</artifactId>
9.   <version>2.2.0.RELEASE</version>
10. </dependency>
11. <dependency>
12.   <groupId>org.springframework.data</groupId>
13.   <artifactId>spring-data-jpa</artifactId>
14.   <version>1.7.0.RELEASE</version>
15. </dependency>
16. <dependency>
17.   <groupId>org.hibernate</groupId>
18.   <artifactId>hibernate-entitymanager</artifactId>
```

I will also need several front-end dependencies, I will use Bower to manage them:

```
1. {
2.   "name": "ng-spring-data",
3.   "dependencies": {
4.     "angular": "~1.3.0",
5.     "angular-spring-data-rest": "~0.3.0",
6.     "bootstrap-css-only": "~3.2.0",
7.     "lodash": "~2.4.1"
8.   }
9. }
```

To change the location of where Bower installs the libraries, I'm also going to add a **.bowerrc** folder to the root of my project, containing the following:

```
1.  {
2.      "directory": "src/main/webapp/libs",
3.      "json": "bower.json"
4.  }
```

If you're asking yourself why we need all these libraries, well, here's a description of why:

- **spring-webmvc:** The Web MVC framework will be used for having the MVC pattern for loading our web application.
- **spring-data-rest:** The Spring Data REST framework allows us to create RESTful webservices of our model quite easily using Spring HATEOAS.
- **spring-data-jpa:** Spring Data JPA is used to create JPA enabled repositories from our models, allowing you to use the repository pattern to work with your data.
- **hibernate-entitymanager:** JPA provides a cool API, but to actually use it, you will need an implementation like Hibernate or OpenJPA. In this case I'm going to use Hibernate, so I need this library as well.
- **hsqldb:** For storing our data we will be using an in memory HSQLDB. We also need to be able to connect to it, so we also use this library for having our JDBC driver.
- **javax.servlet-api:** To use JavaConfig we will need the servlet-api 3, and to use it in our code, we will have to add it as well. However, the web container will usually have this one as well, so we've set the scope to provided for this one.

Then we're also using several front-end libraries:

- **angular:** AngularJS allows us for using the MVC pattern for the client-side part of the application.
- **angular-spring-data-rest:** AngularJS has great integration with REST API's, however, the integration with HATEOAS based RESTful webservices is less, so we will need to use an external library for that.
- **bootstrap-css-only:** We don't want to waste a lot of time setting up the user interface of the application, so using a UI library like Bootstrap, Foundation or Semantic UI really helps.
- **lodash:** JavaScript is great, but there are several tools which you will use a lot that aren't available out of the box, so it's a good idea to have a utility belt library with you like Lo-Dash or Underscore.js.

WebAppInitializer

Modern webapps no longer contain files like web descriptors or Spring bean configuration files. With JavaConfig you can properly replace your web descriptor with a class. The contents will look similar to those of a web descriptor though:

```

1. public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
2.
3.     @Override
4.     protected void customizeRegistration(ServletRegistration.Dynamic registration) {
5.         registration.setInitParameter("dispatchOptionsRequest", "true");
6.         registration.setAsyncSupported(true);
7.     }
8.
9.     @Override
10.    protected Class<?>[] getRootConfigClasses() {
11.        return new Class<?>[] { AppConfig.class };
12.    }
13.
14.    @Override
15.    protected Class<?>[] getServletConfigClasses() {
16.        return new Class<?>[] { WebConfig.class };
17.    }
18.

```

We still have servlets, mappings and filters. In this case I'm gonna map a servlet to the root path and have the class `WebConfig` handle the web context (similar to what `springmvc-servlet.xml` did in the past). The application context is also replaced and now we have a class called `AppConfig` which replaces the `applicationContext.xml` file.

Finally I also added a filter that makes sure that everything is encoded as **UTF-8**. If you don't do that, you may get some strange responses when having special characters.

Setting up the application context

We have to implement two other configuration files called `AppConfig` and `WebConfig`. The first one, `AppConfig`, contains mostly the configuration for setting up the datasource and the entity manager:

```

1.  @Configuration
2.  @EnableJpaRepositories(basePackages = { "be.g00glen00b.repository" })
3.  @ComponentScan(basePackages = "be.g00glen00b", excludeFilters = {
4.      @ComponentScan.Filter(value = Controller.class, type = FilterType.ANNOTATION),
5.      @ComponentScan.Filter(value = Configuration.class, type = FilterType.ANNOTATION)
6.  })
7.  public class AppConfig extends RepositoryRestMvcConfiguration {
8.
9.      @Override
10.     protected void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {
11.         super.configureRepositoryRestConfiguration(config);
12.         try {
13.             config.setBaseUri(new URI("/api"));
14.         } catch (URISyntaxException e) {
15.             e.printStackTrace();
16.         }
17.     }
18. }

```

Let's see what this class contains. Starting from the top we notice several annotations to enable features in our application. First of all we have `@EnableJpaRepositories` to indicate we're going to use Spring Data JPA and that we have our repositories inside the package `be.g00glen00b.repository`.

Also, we have to scan all packages for beans, which we do by using the `@ComponentScan` annotation. However, we don't have to scan controllers, since they're part of the web context and neither do we have to scan configuration files, because they have to be loaded in a different way. That's why we add them to the exclusion filter (`excludeFilters`).

Then, when looking at the class, you can see we're inheriting from `RepositoryRestMvcConfiguration`. This allows us to create RESTful resources from our repositories.

This also means we have to override the `configureRepositoryRestConfiguration()` method, in which we can define the root path of these RESTful resources.

The other methods like `dataSource()`, `jpaVendorAdapter()`, `entityManagerFactory()`, `transactionManager()` and `jpaProperties()` are used for setting up our datasource and JPA.

Setting up the web context

We're now able to create RESTful webservice, however, our application also has a user interface, so we will have to configure our web context as well using `WebConfig`:

```
1.  @Configuration
2.  @EnableWebMvc
3.  @ComponentScan(basePackages = "be.g00glen00b.controller")
4.  public class WebConfig extends WebMvcConfigurerAdapter {
5.
6.      @Bean
7.      public InternalResourceViewResolver getInternalResourceViewResolver() {
8.          InternalResourceViewResolver resolver = new InternalResourceViewResolver();
9.          resolver.setPrefix("/WEB-INF/views/");
10.         resolver.setSuffix(".jsp");
11.         return resolver;
12.     }
13.
14.     @Override
15.     public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
16.         configurer.enable();
17.     }
18. }
```

Starting from the top again, you can see that we use the `@EnableWebMvc` annotation to indicate that we're using Spring Web MVC.

Our class extends from `WebMvcConfigurerAdapter` which allows us to add interceptors (`addInterceptors()`) and resource handlers (`addResourceHandlers()`).

We're adding the `webContentInterceptor()` interceptor that allows use to set cache controlling. The resource handlers are used to indicate which paths should be used to serve static resources. In this case it will be the **libs**, **app** and **assets** folder.

Then finally, we also have to create a bean called the `getInternalResourceViewResolver()` bean. This bean is used when defining controllers so that it's able to resolve the location of the view used inside the controller.

Adding a controller

Speaking of a controller... let's implement it now we're busy. Make sure that you're adding the controller to the `be.g00glen00b.controller` package, because that's the location used inside the component scan of the `WebConfig`.

The controller itself is quite easy, it has only one method, used to load the main application page:

```
1. @Controller
2. @RequestMapping("/")
3. public class AppController {
4.
5.     @RequestMapping(method = RequestMethod.GET)
6.     public String viewApplication() {
7.         return "index";
8.     }
9. }
```

As you can see, it's a quite simple method, that only returns the string `"index"`. Returning in a string means that we're using a view called **index** which matches the file `/WEB-INF/views/index.jsp` thanks to the `getInternalResourceViewResolver()` bean in our configuration.

Defining the model

The application I'm going to build will be a simple todo application, but if you're interested in building something else, you simply define another model.

A todo item can have a description and it can be checked or unchecked. We also need to add an ID, which will be used to know which todo item we're talking about.

The model will eventually look like this:

```

1.  @Entity
2.  public class Item {
3.
4.      @Id
5.      @GeneratedValue(strategy=GenerationType.IDENTITY)
6.      private int id;
7.
8.      @Column
9.      private boolean checked;
10.
11.      @Column
12.      private String description;
13.
14.      public int getId() {
15.          return id;
16.      }
17.
18.      public void setId(int id) {

```

Make sure you add this class to the `be.g00g1en00b.model` package, because this is the only package we said to scan inside the `entityManagerFactory`.

Creating the heart of our application, the repository

All we have to do now is to create a repository based upon the model. Creating a repository with the Spring Data JPA framework is quite easy though, we only have to define a simple interface like this:

```

1.  @RepositoryRestResource(collectionResourceRel = "items", path = "items")
2.  public interface ItemRepository extends PagingAndSortingRepository<Item, Integer> {
3.
4.  }

```

This adds our JPA repository and enables it for REST as well. If you deploy your application now and open your web browser, you can already access the REST API by going to:

`http://localhost:8080/ng-spring-data/api/items`

In this case **/ng-spring-data** is my content root and I'm running my web container on **http://localhost:8080**.



(<http://i2.wp.com/g00glen00b.be/wp-content/uploads/2014/11/rest-response.png>)

Working at the front-end

Before we start with writing JavaScript code, we have to create the HTML page first. As I mentioned before, we mapped the controller so that when the application is opened, the **/WEB-INF/views/index.jsp** file is used as the view. So, let's write that file. The content is quite simple:

```

1. <!DOCTYPE html>
2. <html lang="en">
3.   <head>
4.     <link rel="stylesheet" href="/libs/bootstrap-css-only/css/bootstrap.min.css" />
5.   </head>
6.   <body ng-app="myApp">
7.     <div class="container" ng-controller="AppController">
8.       <div class="page-header">
9.         <h1>A checklist</h1>
10.      </div>
11.      <div class="alert alert-info" role="alert" ng-hide="items && items.length > 0">
12.        There are no items yet.
13.      </div>
14.      <form class="form-horizontal" role="form" ng-submit="addItem(newItem)">
15.        <div class="form-group" ng-repeat="item in items">
16.          <div class="checkbox col-xs-9">
17.            <label>
18.              <input type="checkbox" ng-model="item.checked" ng-change="updateItem(item)" /> <input type="checkbox"

```

As you can see, most of it is simple HTML with some Twitter Bootstrap classes. However, you will also notice some other attributes like `ng-app` and `ng-controller`. These are AngularJS directives which contain the logic for translating this HTML template into the actual HTML visible in the application.

Bootstrapping AngularJS

First of all, we have the `ng-app` attribute, indicating that we have to bootstrap AngularJS in our application and allows us to pass it a name so that we can configure which application modules that should be used. A few lines below we find the `ng-controller="AppController"` attribute. This means that that part of the application is controller by a controller named `AppController`.

The `ng-controller` attribute does refer to an AngularJS controller, not a Spring MVC controller!

Hiding components

The next AngularJS directive has a bit more logic and shows the real power of these directives. The `ng-hide` directive allows you to hide specific elements if the expression passed to it validates to `true`. In this case we're using it to show a message in case there is no todo item yet, so I wrote the following:

`ng-hide="items && items.length > 0"`. This means that the `items` model should exist and should not be empty.

We can use the model in all our directives. For example, to loop over all items inside the `items` model we use the `ng-repeat` directive, for example `ng-repeat="item in items"`, which means that all the elements will be repeated for each item.

Model binding

The most beautiful part of AngularJS is the possibility to have two way binding. For example, if you have a textfield, you often need the value of the textbox inside a variable in JavaScript. In AngularJS you can bind a textbox to a model (= JavaScript property in the controller), so that each time you enter a character in the textbox, the model is updated.

The **two** in two way binding makes the opposite possible as well. If you change the model directly, the value in the textbox will also update.

To bind the model to a form element we use the `ng-model` directive, for example `ng-model="item.checked"`.

Event handling

We can do most things now, except that we have to handle events. If we're adding new items to the todo list, we want to capture the submit event on our form. We can do that by using the `ng-submit` directive.

If you want to track changes, you can also use the onChange event by using the `ng-change` directive, like you can see on the checkbox.

You can also use the `ng-click` directive to handle clicks, like we did on the delete button.

The functions we pass to these directives are functions defined in our controller.

Defining the controller

As you may have noticed now, the controller is the heart of our applications that controls all the small blocks in our application (= the directives).

As you may have noticed from the JSP page, we're including three scripts we didn't define until now: `app.js`, `controllers.js` and `services.js`.

In `controllers.js` we will define our controller, the `AppController`:

```

1. (function(angular) {
2.     var AppController = function($scope, Item) {
3.         Item.query(function(response) {
4.             $scope.items = response ? response : [];
5.         });
6.
7.         $scope.addItem = function(description) {
8.             new Item({
9.                 description: description,
10.                 checked: false
11.             }).save(function(item) {
12.                 $scope.items.push(item);
13.             });
14.             $scope.newItem = "";
15.         };
16.
17.         $scope.updateItem = function(item) {
18.             item.save();

```

As you can see here, we can find all the methods we used in our JSP page. For example, we have the functions `updateItem()`, `deleteItem()` and `addItem()` which we used in the event handling.

We also find the `items` model, called `$scope.items` and the `newItem` model called `$scope.newItem`.

Important to notice here is that we have no HTML code here. If we're deleting a todo item, all we have to do is to delete it from the list of items, (which is why we use the `splice()` function in `deleteItem()`).

We can also update form elements by updating the model, if you look at `addItem()`, you can see that we use `$scope.newItem = ""`, which will erase the contents of the textbox using `ng-model="newItem"`.

As you can see, most of the things we do, have to do with the `Item` object. This object is injected into the controller. AngularJS allows you to inject modules into other modules, which we're using here to inject the `Item` factory.

Using HATEOAS in the Item factory

What you may have noticed when viewing the REST API is that you get a lot of additional data, for example:

```
1.  {
2.    "_links" : {
3.      "self" : {
4.        "href" : "http://localhost:8080/ng-spring-data/api/items{?page,size,sort}",
5.        "templated" : true
6.      }
7.    },
8.    "page" : {
9.      "size" : 20,
10.     "totalElements" : 0,
11.     "totalPages" : 0,
12.     "number" : 0
13.   }
14. }
```

This is called HATEOAS (Hypermedia as the Engine of Application State). Like I said before, AngularJS does not have support for this out of the box (nor in their **angular-resource** project).

So, to allow us to use it without polluting the controller, we abstracted it away into a separate factory, so let's open `services.js`.

The code of this factory is a bit more complex, so let's split it into smaller parts first.

First of all we have to create the module itself, for example:

```

1.  (function(angular) {
2.      var HATEOAS_URL = './api/items';
3.      var ItemFactory = function($http, SpringDataRestAdapter) {
4.          function Item(item) {
5.              return item;
6.          }
7.
8.          return Item;
9.      };
10.
11.      ItemFactory.$inject = ['$http', 'SpringDataRestAdapter'];
12.      angular.module("myApp.services").factory("Item", ItemFactory);
13.  })(angular));

```

This looks quite similar to the setup of the controller, except the fact that we're using a function prototype here called `Item`.

Querying the RESTful webservice

Before we start implementing the prototype, I'm going to add a "static" function called `query()` which will access the API and return a list of items:

```

1.  Item.query = function(callback) {
2.      var deferred = $http.get(HATEOAS_URL);
3.      return SpringDataRestAdapter.processWithPromise(deferred).then(function(data) {
4.          Item.resources = data._resources("self");
5.          callback && callback(_.map(data._embeddedItems, function(item) {
6.              return new Item(item);
7.          }));
8.      });
9.  };
10.
11.  Item.resources = null;

```

We're using the `$http` module to execute an AJAX request to the REST API and then convert the response using the `angular-spring-data-rest` module. Then we're using the `Item` prototype to create new instances of it.

We're also saving the resources into `Item.resources`.

Prototyping Item

The next step is to create the prototype of `Item`. If we look back at the controller, we can see that it should have two functions called `save()` and `remove()`.

The `save()` function has two purposes in this case. When we're creating a new item in `addItem()`, we're using it to add the new item.

However, in `updateItem()` we're updating an already existing item.

To make a difference between these two, we verify if the item has resources, by using `item._resources`. If the factory created the Item, it will have these resources.

However, if the controller made the Item, it will not have them, allowing us to provide other implementation details for the `save()` functionality.

So if we look at the `Item` prototype it will have this structure:

```
1.  function Item(item) {
2.    if (item._resources) {
3.      item.save = function(callback) { };
4.
5.      item.remove = function(callback) { };
6.    } else {
7.      item.save = function(callback) { };
8.    }
9.
10.   return item;
11. }
```

Saving new items

First of all, let's write the `save()` function in case there are no resources. If the item itself has no resources, it has to add the todo item. However, Spring HATEOAS does not provide a response when adding the item, but it does provide the `Location` header, containing a link to the HATEOAS response of the new item.

So, we came up with writing something like this:

```
1. item.save = function(callback) {
2.   Item.resources.save(item, function(item, headers) {
3.     var deferred = $http.get(headers().location);
4.     return SpringDataRestAdapter.processWithPromise(deferred).then(function(newItem) {
5.       callback && callback(new Item(newItem));
6.     });
7.   });
8. };
```

It uses the `Item.resources`, because these resources tell us what the URL is for saving new items. After saving, we use the headers to retrieve the location header, by using `headers().location` and we use `$http` to call it using AJAX.

The response has to be promised using **angular-spring-data-rest**, similar to how we did inside the `query()` function, only this time it's for handling only one item.

Updating and deleting existing items

Each item individually has separate `_resources`, which will tell us what URL to use to update and delete the songs. So, updating and deleting is quite simple:


```

1.  item.resources = item._resources("self", {}, {
2.      update: {
3.          method: 'PUT'
4.      }
5.  });
6.  item.save = function(callback) {
7.      item.resources.update(item, function() {
8.          callback && callback(item);
9.      });
10. };
11.
12. item.remove = function(callback) {
13.     item.resources.remove(function() {
14.         callback && callback(item);
15.     });
16. };

```

Putting everything together in the factory we get:

```

1.  (function(angular) {
2.      var HATEOAS_URL = './api/items';
3.      var ItemFactory = function($http, SpringDataRestAdapter) {
4.          function Item(item) {
5.
6.              if (item._resources) {
7.                  item.resources = item._resources("self", {}, {
8.                      update: {
9.                          method: 'PUT'
10.                     }
11.                 });
12.                 item.save = function(callback) {
13.                     item.resources.update(item, function() {
14.                         callback && callback(item);
15.                     });
16.                 };
17.
18.                 item.remove = function(callback) {

```

Defining the application packages

The last file that's left is the `app.js` file, defining the packages `myApp`, `myApp.controllers` and `myApp.services`:

```
1. (function(angular) {  
2.     angular.module("myApp.controllers", []);  
3.     angular.module("myApp.services", []);  
4.     angular.module("myApp", ["ngResource", "spring-data-rest", "myApp.controllers", "myApp.services"]);  
5. })(angular));
```

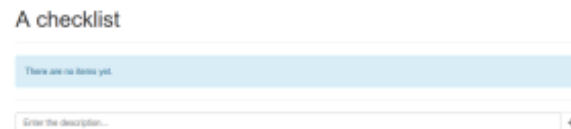
The name of the `myApp` module is the same name we have to enter when bootstrapping the application (`ng-app="myApp"`).

The array we're using as the second parameters is the list of module it depends on. In this case the controllers and services are not depending on any other modules, however, the application itself needs the `ngResource`, `spring-data-rest`, the controllers and the services modules to work.

Demo

That's it for the development of the application. As you can see the amount of Java code we had to write was pretty small, even configuring the application took us longer than actually writing the code. The JavaScript side was a bit more complex, but I hope to see more support for HATEOAS in the future in AngularJS. I tried to make it run similar to how `angular-resource` works, but that came with a cost for implementing such a factory.

But let's try out the application. If you deployed it in your web container, you should be able to see the following application in your web browser.



(<http://i2.wp.com/g00glen00b.be/wp-content/uploads/2014/11/checklist-no-items.png>)


If you add a new item, you will see that two network requests are sent, one for adding the item and another one for retrieving the contents of that item.


That's it for this tutorial. You're now able to write web applications using Spring MVC, Spring Data REST and AngularJS.




Achievement: Built modern webapps using Spring Data REST and AngularJS

Seeing this means you managed to read through the entire tutorial about developing modern webapps using Spring Data REST and AngularJS. If you're interested in the full code example, you can find it on [GitHub \(https://github.com/g00glen00b/ng-spring-data\)](https://github.com/g00glen00b/ng-spring-data). If you want to try out the code yourself, you can download an archive from [Github \(https://github.com/g00glen00b/ng-spring-data/archive/master.zip\)](https://github.com/g00glen00b/ng-spring-data/archive/master.zip).

 Facebook 3 (<http://g00glen00b.be/spring-data-angular/?share=facebook&nb=1>)

 Twitter 56 (<http://g00glen00b.be/spring-data-angular/?share=twitter&nb=1>)

 LinkedIn (<http://g00glen00b.be/spring-data-angular/?share=linkedin&nb=1>)

 Google (<http://g00glen00b.be/spring-data-angular/?share=google-plus-1&nb=1>)

 Reddit (<http://g00glen00b.be/spring-data-angular/?share=reddit&nb=1>)

DZone (<http://g00glen00b.be/spring-data-angular/?share=custom-1424415421&nb=1>)

Related

Writing real time applications using Spring and AngularJS (Part 1: Project setup) (<http://g00glen00b.be/spring-websockets-config/>)
09/03/2014
In "Java"

Using WebSockets with Spring, AngularJS and SockJS (<http://g00glen00b.be/spring-angular-sockjs/>)
05/10/2014
In "Java"

Rapid prototyping with Spring Boot and AngularJS (<http://g00glen00b.be/prototipi...spring-boot-angularjs/>)
20/12/2014
In "Java"

This entry was posted in Java (<http://g00glen00b.be/category/t/java/>), JavaScript (<http://g00glen00b.be/category/t/javascript/>), Tutorials (<http://g00glen00b.be/category/t/>).

13 Comments

g00glen00b

1 Login ▾

♥ Recommend

🔗 Share

Sort by Best ▾



Join the discussion...



Greg Turnquist • 5 months ago

Looks great! FYI @RepositoryRestResource(collectionResourceRel = "items", path = "items") is unnecessary. Spring Data REST takes the name of your domain class, and using the EvolInflator library, creates a pluralized version of it, which in this case, would be "items". Only if you want to change that choice do you need this annotation (like Person => "people").

BTW, have you looked at Spring Boot? With that, you can really reduce your Spring footprint and setup costs. it lets you move to the Angular part really fast. If you put spring-boot-starter-data-rest, spring-boot-data-jpa, and spring-boot-starter-web in your pom.xml, then you can drop your JS libs into src/main/resources/static, and be off the Angular!

Anyway, cheers on the blog entry!

2 ^ | ▾ • Reply • Share ›



g00glen00b Mod ➔ Greg Turnquist • 5 months ago

I've heard a lot of great things about Spring Boot lately, so I might try it out soon! Thanks for the comment :D

^ | ▾ • Reply • Share ›



g00glen00b Mod ➔ g00glen00b • 4 months ago

And so I did: <http://g00glen00b.be/prototypi...> It was really easy with Spring Boot. I'm now looking at writing integration tests (with the Spring Boot annotations).

^ | ▾ • Reply • Share ›



Aycan Adal • 2 months ago

Thanks for great article. I am new to angularjs and spring data rest. Few questions about the Item.query:

Why do you use \$http but not \$resource?

Why do you use SpringDataRestAdapter but not the interceptor?

How would you query a single item?

How can you set the Item.resources without loading all items?

1 ^ | v • Reply • Share ›



g00glen00b Mod → Aycan Adal • 2 months ago

I'm using `$http` here because `$resource` (imho) is more designed for non HATEOAS-resources. The URLs in HATEOAS resources should come from the response, and not a hardcoded URL in a resource, and that's why I'm using a specific library to do that.

The interceptor is certainly a possibility, but I tried to make a separate factory that resembles `$resource` a lot, but uses HATEOAS as its main resource for knowing which URLs to access. It could be that the SpringDataRestAdapter already has some feature that already does this, but I found it very hard to get through their docs.

Querying one item is certainly possible, but I didn't implement it in this example because I didn't need it. The HATEOAS response should contain a link used for updating/removing/querying a specific item, so you could use a similar approach as the remove functionality.

I'm not a HATEOAS expert, but I'm pretty sure you can't. After writing this article, my opinion is that I don't really like it, because it increases the workload on the front-end a lot. Because it follows the Representational State Transfer (REST) guidelines thoroughly, it means that you only know how to update/delete/find a specific item is by querying the REST resource first so the response tells you how to do it.

If you look at my more recent tutorial about Spring Boot (<http://g00glen00b.be/prototypi...>) you will see that I'm no longer using Spring Data REST, but I'm using Spring Data JPA with a simple Spring Web MVC controller that returns an array of items. It requires some additional code on the back-end, but makes the front-end a lot easier because I'm no longer using HATEOAS.

^ | v • Reply • Share ›



Aycan Adal → g00glen00b • 2 months ago

I've been messing around with consuming spring data rest api from angular for three days now. I started with your article because it's the only example out there nicely structured and coded. But it turns out that what you do here (crud) is totally doable and much cleaner with plain `$resource`. You don't even need SpringDataRestAdapter for this. And yes

SpringDataRestAdapter does provide means for doing crud out of the box as well but the real strength is how it helps you traverse links not only as in crud actions but as in related resources like posts of a user and complex actions that fall out of crud's scope. And it's fine tuned for spring data rest repositories. Out of angular-hateoas, restangular and other alternatives, SpringDataRestAdapter is definitely the best choice against a spring backend. So what you show here is pretty good for bigger scale applications but is an overkill for a project like this and for beginners like me. Check this: <http://blog.mgechev.com/2014/0...>

The equivalent of your item factory looks like:

```
module.factory('User', function ($resource) {  
  return $resource('/users/:id');  
});
```

Three lines of code :) And you can create(instantiate), get, query, update, delete and do much more just like you do in your controller using the prototype returned by \$resource. You can also post a new resource without having to load all the resources first.

1 ^ | v • Reply • Share ›



g00glen00b Mod ➔ Aycan Adal • 2 months ago

True, I know how to use \$resource. I'm using it for most of my blog articles as well. But for Spring-Data you have to do some modifications. For example, the `query()` method will not work with Spring Data (by default), because Angular-resource expects an array, while (if I recall correctly), Spring HATEOAS returns an object with an embedded array.

1 ^ | v • Reply • Share ›



Aegidius • a month ago

Is it possible to retrieve an embedded items ID?

^ | v • Reply • Share ›



g00glen00b Mod ➔ Aegidius • a month ago

They're not included in the response, but you could retrieve the ID from the link it includes with each embedded item. I'm not sure if you can override this behavior to include the ID.

^ | v • Reply • Share ›



Kaspar • 2 months ago



There is no main method? How can you even run this without no main method?

^ | v • Reply • Share ›



g00glen00b Mod ➔ Kaspar • 2 months ago

This example is a web application packaged as a WAR file. You don't need main methods there, because you use a **web descriptor**. Usually it would be web.xml, but with more recent versions of Spring we can use a Java class, in this case `WebAppInitializer`.

1 ^ | v • Reply • Share ›



sd • 2 months ago

Hi - I downloaded and attempted to run the app (using NetBeans v8.0.2, Oracle WebLogic 12c). My IE11 Browser URL resolves to `http://localhost:7001` (default) and page displays a "Error 403--Forbidden" message... - I tried manually typing in `http://localhost:7001/index.jsp`, but, same thing. Can you venture a couple of possibilities as to what I might be doing wrong? (thanks for your example in any case)

^ | v • Reply • Share ›



g00glen00b Mod ➔ sd • 2 months ago

Are you sure the context root of your application is `/`? In Eclipse it's usually the name of the application. If you're using the code from the Github repository, it should be `http://localhost:7001/ng-spring-data/`

^ | v • Reply • Share ›

ALSO ON G00GLEN00B

WHAT'S THIS?

Writing real time applications using Spring, AngularJS and WebSockets

11 comments • a year ago



g00glen00b — Yes and no. I'm using Bower to install my dependencies, so I didn't commit the libs folder on GitHub.

Testing your Spring Data JPA repository

2 comments • 2 months ago



rordonez — Nice Post!! If you use Spring 4, you can also make your class `ItemRepositoryIT` to extend ...

Working with the MEAN stack: MVC

6 comments • a year ago



Tyler Wendlandt — yeah - I hadn't made the switch to the api yet. Thanks for the heads up

Meteor Twitter streaming

9 comments • a year ago



g00glen00b — I also included the hidden folder though ;) So you should be able to run it anyways. The app simply streams `#JavaScript` ...

