AngularJS HTML5 routing on PlayFramework

Posted by Paul Dijou on Feb 17, 2013 16 comments Improve this article

The problem

AngularJS provides a nice `$location` service which allows you to deal with routing and url inside your application. It supports both a fallback mode using hashbang and a real HTML5 routing using the new History API. Since we love bleeding edge technologies, that's obvious we want to use the History API if possible. Lucky us, it appears it's quite easy to do so with a Play Framework application. Let's see how to do that.

The full code of the demo is available on GitHub.

Simple configuration

Let's keep things simple. Creating a new Play Framework application with `play new`. We will use a Scala application in this article, but it should as easy with a Java one. Then add AngularJS using CDN (see main.scala.html#L19). Then we will create an Angular module for our app and enable the HTML5 routing mode which is disable by default.

```
var app = angular.module("app", ["ngResource"])
  .config(["$routeProvider", function($routeProvider) {
      return $routeProvider.when("/", {
        templateUrl: "/views/index",
        controller: "AppCtrl"
      }).otherwise({
        redirectTo: "/"
      });
    }
  ])
  .config(["$locationProvider", function($locationProvider) {
      return $locationProvider.html5Mode(true).hashPrefix("!");
    }
  ]);
```

Nice. Now we will make some modification to the default views generated by Play. We want a single-page application, which means our root url must map to a main page which will contains all resource dependencies and includes the `ng-view` of AngularJS. I've done that by editing the `main.scala.html` file as you can see here and add a new Play route:

```
GET     /                            controllers.Application.main(any = "none")
```

And a new `Action` in my Application controller:

```
def main(any: String) = Action {
  Ok(views.html.main())
}
```

Wait. Why is there a String param for the main method? And why do we pass a default `any = "none"` value? It seems a bit useless... Fair enough, I will explain it later on the article, don't mind it for now. So, we have our main page. Let's add some content for the landing page. As we have specified in the AngularJS routing, our root url, which is defined with `$routeProvider.when("/", ...)` is pointing to `/views/index`. So we need a Play route for that... but before, let's talk about all those routes.

Routes, routes, routes...

As we advance in our coding, we can see that the term "route" is used in different contexts. It's important to fully understand them all. Play routes, which are defined in the `conf/routes` file, are the core of your application. It's those routes which will answer to the HTTP requests, serving either HTML, JSON, or whatever. But you, as the developer, will be the only one to know about them. Users of your application will only use and see AngularJS routes, which are defined in your main module using the `$routeProvider`.

The idea is that when a user land on your site, it should use AngularJS routing which will then use Play routes to load the template (= the content) of the page. So why not using the exact same url for both the AngularJS route and the templateUrl route? Well, in the case of the root, it's obvious: the Play "/" route is already required for our main page. Ok. But in a more general case, could we do:

```
$routeProvider.when("/page", {
  templateUrl: "/page",
  controller: "PageCtrl"
})
```

Nope, you can't. Well, that's a small lie... you could do that, and it would work if your user **always** open your main page first. If he does that, then all AngularJS routing is loaded, and when the user will go to "/page" url, it will be catch by AngularJS which will ask for the "/page" on Play routes which will give him the content of the page. So what's the problem? If you user **directly** land on the "/page" url, because he arrived for the first time on your application after having clicked on a link somewhere which was pointing at "/page", then the Play route will be directly called since there is no AngularJS routing initialized at all... And that's bad, right? The user would have the raw content of the page without any JavaScript or CSS from the main page. It's a fail.

In order to prevent that, I have chosen to use some conventions. There is nothing official, you can use your own, the only thing to keep in mind is that AngularJS routes and Play routes should **never** be equals. My solution to do that is to prefix all Play routes serving HTML with a "/views/" prefix, all Play routes serving JSON with a "/api/" prefix and, as it is by default, all Play routes serving resources with a "/assets/" prefix. Meaning I can now use all routes I want inside AngularJS routing but never starts them with one of those prefix.

Of course, it will just modify the problem without solving it. Now, if a user land directly on "/page", he will have a huge orange Play error telling him that the route doesn't exist. As before, AngularJS routing isn't loaded, so it's Play routing which try to handle that and, since there is no longer any "/page" route in Play, it crash.

So what's the point of having different routes? The next trick is to redirect all unknow routes to our main page. By doing so, if the user land on "/page", Play will say "*I don't know that route, go to the main page*", so the user will indeed go to your main page, but then, all AngularJS routing will be loaded and super awesome AngularJS will say "*Hey, I do know that route, and I need the /views/page template*", and then Play will respond "*Oh yeah, I can serve you /views/page, here is the HTML to display*", and your user will see the correct "/page1" content, even if he lands directly on it.

Another question? Why didn't we use this trick before and then have the same routes for both AngularJS and Play? Because it's important that the trick apply only to unknow routes. We cannot redirect to the main page an existing Play route because it needs to serve its own content. So by using different routes, we can intercept AngularJS routes inside Play as "unknown routes" and redirect them to the main page so AngularJS can handle them. I hope I'm clear enough here, that might be the hardest part of the article. In order to achieve that, we just need to put a Play route at the end of the route file which intercept all routes and redirect them to the main page:

```
GET     /*any                       controllers.Application.main(any)
```

And now you should understand why we had a `any: String` for our main page at the beginning.

First pages

Enough with all the theory, let's do two concrete pages: creating "page1.scala.html" and "page2.scala.html" files with basic content (see source code, I'm using a template to stay DRY about the title and the menu but that's not important), add then as Play routes (with a "/views/" as stated before) and also add them inside our controller:

```
@()

@template() {
  <h2>Page1</h2>
}
```

```
GET     /views/page1                controllers.Application.page1
```

```
def page1 = Action {
  Ok(views.html.page1())
}
```

And then add the AngularJS routing:

```
.when("/page1", {
  templateUrl: "/views/page1"
})
```

That's it. We can now create a link, using a `a` tag, to point to the url "/page1" and AngularJS will do the rest, loading the "/views/page1" template using Play route which will return the HTML code generated from our Scala template. Nice isn't it? The url is correct: http://localhost:9000/page1, you can use the back button to return to the index page, no ugly hashbang, but all in Ajax with a single-page application.

Handling url parameters

So, we have our url working by using the History API thanks to the HTML5 mode of the AngularJS $routeProvider. But that's only for raw url. What if we want to have dynamic parameters inside them? Just like REST, having url as "/colors/1" and "/colors/2" if we deal with colors (yeah, why not?). Do you think that's easy? Well, you could, after all, both AngularJS and Play know how to handle parameters in their url. We could do something like:

```
.when("/colors/:id", {
  templateUrl: "/views/color/:id",
  controller: "ColorCtrl"
})

GET     /views/color/:id            controllers.Application.color(id: String)
```

Nope, we can't. Why? Because the "templateUrl" cannot handle url parameters. It's just a template. And of course, using `templateUrl: "/views/color"` will not pass any "id" parameter to the Play route. Damn...

That's not a problem in fact. Just keep to the AngularJS way of doing things: it's not Play who should handle the data any longuer, it's AngularJS who rules the world now, so you don't need any params in your Scala code, just trust AngularJS. Ok, but we need a way to load the correct color depending on the id in the url right? Sure thing. It's the role of AngularJS controller to that. That's why there is a "ColorCtrl" along with our template, and guess what, the AngularJS controller know about the url params. Here is the code of the controller:

```
app.controller("ColorCtrl", ["$scope", "$routeParams", function($scope, $routeParams) {
  // Thanks to scope inheritance, we can access the "db" from the AppCtrl scope
  $scope.color = $scope.db[$routeParams.id];
  if (!$scope.color) {
    $scope.msg = "There is no color for id "+$routeParams.id;
  } else {
    $scope.msg = undefined;
  }
}])
```

And just for fun, I've defined the fake database in the "AppCtrl":

```
app.controller("AppCtrl", ["$scope", function($scope) {
  $scope.db = {
    1: {
      name: "black",
      hex: "000000"
    },
    2: {
      name: "white",
      hex: "FFFFFF"
    }
  };
}]);
```

So as we can see, using the `$routeParams` service, we can retrieve the url params and then load the correct color in the `$scope`. At the end, it's just a matter of displaying that color in our view using AngularJS data-binding:

```
@()

@template() {
  <div data-ng-show="msg">
    <h2>{{msg}}</h2>
  </div>

  <div data-ng-hide="msg">
    <h2>Color {{color.name}}: # {{color.hex}}</h2>
  </div>
}
```

And that's it! you can now display some data depending of params inside your url (and show an error message in case the data doesn't exist). Cool isn't it?

Want more?

Oh God, you are still reading? You should already be able to do anything you need. But if you want, we can dive in a more complex example, using AngularJS `$resource` service and serve JSON from Play like a REST API would do. Sounds good for you? Ok, let's do that. First, we will create a "Users" controller and it will have two methods: "all()" and "find(id: String)" which will return an Array of Json and a Json object representing our list of users and one particular user based on its id. I will not use a real database, but something like MongoDB would fit really good in there. Here is the code:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.libs.json._

object Users extends Controller {
  val db = Json.arr(
    Json.obj( "id" -> "1", "name" -> "John" ),
    Json.obj( "id" -> "2", "name" -> "Suzanne" )
  )

  def all() = Action {
    Ok(db)
  }

  def find(id: String) = Action {
    Ok(db.value.filter(v => (v \ "id").as[JsString].value == id).headOption.getOrElse(new JsUndefined("")))
  }
}
```

If you don't fully understand it, you can check the PlayFramework documentation about handling Json (ScalaJsonRequests). I am using the new Json API from Play 2.1 (play.api.libs.json.package). Next we need our "UserCtrl" which will use the `$resource</code> service to retrieve the data:

```
app.controller("UserCtrl", ["$scope", "$routeParams", "$resource", "apiUrl", function($scope, $routeParams, $resource, apiUrl) {
  var Users = $resource(apiUrl + "/users/:id", {id:"@id"});

  if($routeParams.id) {
    $scope.user = Users.get({id: $routeParams.id}, function() {
      if (!$scope.user.id) {
        $scope.msg = "There is no user for id "+$routeParams.id;
      } else {
        $scope.msg = undefined;
      }
    });
  } else {
    $scope.users = Users.query();
  }
}])
```

To learn more about AngularJS $resource definition, see: ngResource.$resource. What is that "apiUrl" by the way? It's a constant I've defined in my AngularJS app:

```
var app = angular.module("app", ["ngResource"])
  .constant("apiUrl", "http://localhost:9000\:9000/api")
```

Pro tip Why is there the 9000 port twice? That's because if we had written `http://localhost:9000/api`, AngularJS syntax would have analyzed that as a url with a dynamic parameter named "9000" because it's placed right after a `:` character. So we need that strange syntax to tell AngularJS that this is not a parameter but a real value in our url.

Next, we are creating our resource by extending this apiUrl with our routing "/users/:id". This time, ":id" is a real parameter. We can now use "get" and "query" methods on the resource, passing or not a value to the id, in order to retrieve our Json code and assign it to the `$scope`. We will need two Scala views: one for the list and one for the detail of course.

```
@()

@template() {
  <h2>Users</h2>

  <ul>
    <li data-ng-repeat="u in users">
      <a data-ng-href="/users/{{u.id}}">User# {{u.id}}: {{u.name}}</a>
    </li>
  </ul>
}
```

```
@()

@template() {
  <div data-ng-show="msg">
    <h2>{{msg}}</h2>
  </div>

  <div data-ng-hide="msg">
    <h2>User# {{user.id}} {{user.name}}</h2>
  </div>
}
```

And that's it. We now have a list of users and links to each user detail, and all that data is fetch from a REST API using Json.

Pro tip By the way, since our Play routes doesn't clash with AngularJS routes, you can load them directly in your browser. It will works because the Play route handling the redirection to the main page is at the end of the route file, so any real Play route will be loaded before the redirection. If you have the demo running, check http://localhost:9000/api/users and http://localhost:9000/api/users/1 to see your REST API working nicely.

Conclusion

I hope this article will help you bootstrap your next awesome application using awesome tools like AngularJS and PlayFramework. Your next step (if that's not already the case) would be to learn more about the new PlayFramework Json API so you can have typesafe Json (if I can say so). You would also probably need to plug a database on it. One choice could be a MongDB database since it can store raw Json, and interact with it using a nice driver like ReactiveMongo if you want to go all the way down to asynchronous application.

It's up to you!

Permalink 16 comments Improve this article Improve this article
Tags : angular play html5
Previous
Next

16 Comments Paul Dijou Wellington Torrejais

Recommend Share Sort by Oldest

Join the discussion…

Etienne Vallette d'Osia • 3 years ago

Super article ! Je me posais des questions sur les limitations d'une single app page (je ne connais que très très mal AngularJS et ses amis), mais là tu viens de résoudre mes 2 problèmes : URL étrange (l'History API est impressionnante d'ailleurs) et chargement de tous les templates dans le JS.

J'aime beaucoup la séparation /views qui rend des templates et /api qui rend des données :-)

Bref, je sens que je vais réutiliser pas mal de choses de ton article dans pas longtemps…

• Reply • Share ›

Paul Mod Etienne Vallette d'Osia • 3 years ago

Fais toi plaisir Etienne, c'est là pour ça après tout. Toujours content de pouvoir aider. Si tu te lances dans le single-app avec Play, je suis preneur de tout retour à ce sujet.

A noter que tu auras de nouveaux des urls étranges si jamais le navigateur de ton utilisateur ne supporte pas l'History API, mais bon, pas le trop le choix dans ce cas là.

• Reply • Share ›

Janne • 3 years ago

Great article, thanks!

Instead of implementing one action for each view i wrote a "view-resolving" action that takes the view name and uses reflection to render the compiled view.

routes:

GET /views/*viewname controllers.Application.viewResolver(viewname)

Application.scala:

```
def viewResolver(name: String) = Action {
val cls = Class.forName("views.html.%s$".format(name.replace("/", ".")))
val template =
cls.getField("MODULE$").get(null).asInstanceOf[play.api.templates.Template0[play.api.te
Ok(template.render())
}
```

• Reply • Share ›

Paul Mod Janne • 3 years ago

Hey there,

Thanks for the feedback and sorry for the delay, a bit busy.

Personally, when I arrive on a Play project, my first action is to open the routes file and read it. It gives me a nice overview of the site. That's why I like writing all my routes most of the time. Another good point is that you can perform calculations for each routes on the server side before rendering the template. The most common one would be permission checking. But I understand your point.

So what if we don't want to write all that routes anymore? I didn't find a solution with my knowledge so far but I would try Scala Macros if I knew how to use them.

I pushed a half solution on the GitHub repo using a new Router controller with three methods ("public", "authenticated" and "admin") which will first check if the