AngularJS + Rails

Jumpstart your front end web applications using Angular and Rails

Jonathan Birkholz &
Jesse Wolgamott

AngularJS + Rails

Jumpstart your front end web applications using Angular and Rails.

Jonathan Birkholz and Jesse Wolgamott

This book is for sale at http://leanpub.com/angularails

This version was published on 2014-03-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Jonathan Birkholz and Jesse Wolgamott by spreading the word about this book on Twitter!

The suggested hashtag for this book is #angularails.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#angularails

Contents

1 Introduction

Thanks for taking the time to read our guide!

This started out as just a hodge podge of notes and exercises from workshops we organized to help teach people AngularJS. We are Rails guys so our training naturally had instructions and tips on how to integrate AngularJS with Rails. Pro tip: It is super easy.

We loved our training material and kept referring back to it while we developed. That is when we had the idea to turn our notes into a comprehensive guide.

1.1 What this guide is

- A great starting point to learning AngularJS.
- An easy to follow step-by-step guide to using AngularJS in Rails.
- A place to learn easy ways in which AngularJS integrates with Rails.
- A great reference to refer back to when building your application.
- In short, a rapid way to get productive with AngularJS and Rails.

1.2 What this guide isn't

- An exhaustive guide to all things AngularJS. We will get the ball rolling and you building your application. You can then use other resources when you get to the difficult subjects.
- A step-by-step guide to learning Rails. We assume you have some experience with Rails. Although no experience is required to go through the guide, we don't spend a lot of time explaining the Rails side of things.
- The end of your developing journey. We are the staring point. We all have time crunches and this guide will get you up to speed and building awesomeness fast.

2 Status of AngularJS + Rails = AngulaRails

This book is in the editing phase. We're going through and working through the exercises, trying to find bugs, typos, and better ways to express the information.

If you find anything that is incomplete, not working, or cray-cray, email us at Jesse at jesse@comalproductions.com or JB at rookieone@gmail.com.

3 Why Angular?

Every time you check the Interwebs, there seems to be a new shiny JavaScript framework. The client-side framework space has seen a lot of activity the past few years. Every framework has their fanatical element whose honor is challenged if someone else picks a different framework for their application.

In reality, every framework has its own strengths and weaknesses. Not one of the frameworks is inherently better than another; instead, they just offer different solutions to similar problems. So in a world of infinite client-side javascript frameworks, why would you choose Angular?

3.1 Pros: The greener grass

Binding! Yay!

AngularJS uses binding to update content on the page. The binding syntax works by adding simple attributes to existing DOM elements. These attributes are known in AngularJS as directives. Directives offer far more than just binding constructs, but that is for a later chapter. With AngularJS, it is sufficient to know there is no need for a separate template engine. What this means in a Rails application is that you can easily use AngularJS with your existing view engines like ERB or HAML.

AngularJS is super awesome because you can bind to simple, native JSON. Pull data down from an API, and just slap it onto $scope. BOOM! Fast actin' binding action.

Binding to JSON is easy

```
1 $http({ method: "GET", url: "http://some_url/books"})
2   .success(function(response) {
3     $scope.books = response.books
4   });
```

Flexibility

AngularJS's greatest strength is its flexibility. AngularJS can grow and shrink to fit the dynamic evolution of your application. You can have single components or multiple AngularJS apps operating on the same page. Your application can be just a dynamic page or a complete single-page application with all the features you need baked in.

You can also easily integrate AngularJS with other libraries and tools. When you use a framework, the general rule is to do everything in the framework. i.e. no jQuery in your views, viewmodels,

controllers, whatever. But sometimes you need to get something to work and it doesn't have to be pretty. AngularJS is more fluid, allowing you to be naughty inside your different controllers or views. Color outside the lines and apply some duct tape. Refactor when the dust settles.

Scalability

Part of the flexibility is AngularJS's scalability. You can start upgrading one page in an existing application. Then you can upgrade another, and another. You then can turn multiple pages into a Single Page Application using routing. Whether your AngularJS features are in a single page application or just a simple upgrade, AngularJS is easy to drop in and scale.

Embraces JavaScript's strengths

One of JavaScript's strengths is its functional nature. AngularJS embraces that with no complicated object models. You define functions and bind to native JSON. AngularJS provides simple lightweight mechanisms to start building an application without having to learn a large complicated framework.

3.2 Cons: Here be dragons

Binding! Yuck!

Some people are completely averse to binding. Often binding-phobes have the understandable desire to know the DOM is going to be updated. So if you dislike binding voodoo, then AngularJS isn't the framework for you.

No model layer

For some, the model is the single source of truth. Having the model's state and actions in one class is a comfortable object-oriented approach to providing structure to an application's domain. Since AngularJS binds to native JSON, you don't need a fancy model layer. With AngularJS, you will find that the path of least resistance is to have domain related services that operate on your simple JSON. This is one of the ways AngularJS is more functional and embraces JavaScript's strengths.

This means that instead of saving a user like `user.save()`, you would have a `User` service you inject into the controller and call `User.save($scope.user)`. We view this as a big win because testing a function is super easy!

No framework enforced guidance on building large applications

If you are building a large application from the ground up, you will need to do some work with AngularJS to enforce best practices and code structure. The power of AngularJS is its flexibilty, but

with that power comes the burden of being responsible with your code. In the wrong hands, you can create an ugly beast of spaghetti code. While generally true with any framework, there are some frameworks that enforce some semblance of structure. When it comes to large applications, you will have to create your own guidance and just enforce the patterns through code reviews.

Conclusion

If you don't mind the binding, and the idea of simply binding to JSON is attractive to you, then you should take a serious look at AngularJS. AngularJS is flexible and terse allowing you to deliver extraordinary features with fewer lines of code. AngularJS is a fantastic tool to have in your toolbox and is a go-to framework for delivering awesome client-side UX.

4 Hello AngularJS

But enough with the talking points. Let's get started with the very basics of setting up AngularJS in Rails.

What is covered:

- Basic Rails application creation
- Creating an AngularJS application with `ng-app`
- Simple view binding using {{ }}
- Binding a value to a text input with `ng-model`
- Creating and using an AngularJS controller with `ng-controller`
- Setting a value using `$scope`

4.1 Create your Rails Application

If you are a Rails veteran and have your own flavor of Rails, then skip this section. But if you are relatively new to Rails, or are just curious as to the setup we are using for this guide, continue reading.

First we will create a new rails application and generate all the folders and files that Rails will use.

terminal

```
$ rails new angularails
```

The default Rails generators, generate a bunch of junk we don't need for this guide. Junk as in… tests. Who needs those! (PSA: We actually love tests and is one of the reasons Angular is so awesome!)

config/application.rb

```
config.generators do |g|
  g.stylesheets false
  g.helper false
  g.javascripts false
  g.test_framework false
end
```

We are going to add a few gems to our `Gemfile`.

First we are going to add `active_model_serializers`. There are many options for how you handle model serialization with Rails. Active Model Serializers is a great and popular option which we will be employing here.

We are also going to be adding `font-awesome-rails`. We are installing the font icons from FontAwesome using the gem just to simplify our setup.

We are also going to remove some default gems we won't be using. In the end, your `Gemfile` should look like:

Gemfile

```
source 'https://rubygems.org'

# Rails
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# Use jquery as the JavaScript library
gem 'jquery-rails'

# Model serialization
gem 'active_model_serializers'

# Font Awesome icons
gem 'font-awesome-rails'

# Paging
gem 'kaminari'
```

Add CSS and JavaScript

We will be using Twitter Bootstrap for our basic styling. We also will be using `angular.min.js` (duh!).

You can get the files at `http://www.angularails.com/files`

Add the `bootstrap.min.css` file to the `assets/stylesheets` folder. We won't change the default `application.css` file and let `require_tree` load the file for us.

Next we will add the both the `bootstrap.min.js` and `angular.min.js` JavaScript files to the `assets/javascripts` folder.

Now for the fun stuff! Time to bring in AngularJS. First we are going to remove the `require_tree` from `application.js`. There is just something unsettling about loading JavaScript in any order we didn't specify directly.

app/assets/javascripts/application.js

```
1  //= require jquery
2  //= require jquery_ujs
3  //= require bootstrap.min
4  //= require angular.min
```

Using Twitter Bootstrap Styling

Replace your `application.html.erb` with:

application.html.erb

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=Edge,chrome=1">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title><%= content_for?(:title) ? yield(:title) : "Angularails" %></title>
8      <%= csrf_meta_tags %>
9      <%= stylesheet_link_tag "application", :media => "all" %>
10     <%= javascript_include_tag "application" %>
11   </head>
12   <body>
13     <div class="navbar navbar-inverse">
14       <div class="container">
15         <a class="navbar-brand" href="#">Angularails</a>
```

```
16        </div>
17      </div>
18
19      <div class="container">
20        <%= yield %>
21      </div>
22    </body>
23  </html>
```

4.2 Hello AngularJS

Now we will use the Rails generator to create a simple controller and view.

terminal

```
1  $ rails g controller HelloAngular show
```

This will create a `HelloAngular` controller with a `show.html.erb` view.

Replace the generated markup in the `show.html.erb` view with:

views/hello_angular/show.html.erb

```
1  <div class="row" ng-app>
2    <div class="col-md-6">
3      <h1>Hello AngularJS</h1>
4      <p>Type in the input box and watch as AngularJS updates the property through \
5  the magic of binding.</p>
6      <input type="text" class="form-control" ng-model="myText" >
7      {{ myText }}
8    </div>
9  </div>
```

Line 1 `ng-app`
: By setting `ng-app` on a HTML element we are creating an Angular application whose scope is passed to its children.

Line 6 `ng-model`
: This will bind the value from the input to the `myText` property on our application scope.

Line 7 `{{ myText }}`
: The mustache curly braces will display the current value of `myText`.

Try it out!

Time to test if we did everything right. Start up your rails server and go to `http://localhost:3000/hello_angular/show` As we type, we should see our input duplicated besides the input box. Awesome, right? Super cool.

4.3 Using an AngularJS Application and Controller

While the binding is cool, we will need an AngularJS application and a controller to do anything more complicated.

We will first create a JavaScript file to load our application and all our future AngularJS files.

Create javascripts/angular-application.js

```
1 //= require_self
2 //= require_tree ./angular
3
4 AngulaRails = angular.module("AngulaRails", []);
```

Line 1 `require_self`
This will load the current file before it loads the other files in the `angular` folder we will create.

Line 2 `require_tree ./angular`
This will load all the files we will create in the `angular` folder. This will be controllers, routes, directives, and any other javascript file we create in that folder.

Line 4 Creating the application
This will create our Angular application that we are naming `AngulaRails`. You can name this whatever you want.

`angular-application.js` will use sprockets to load all our future AngularJS files (like controllers) allowing those files to reference our `AngulaRails` Angular application.

Now we need to add `angular-application` to our `application.js` so that the file gets loaded.

javascripts/application.js

```
1 //= require angular-application
```

Create an `angular` folder under the `assets/javascripts` folder.

Inside the `angular` folder, create a new file:

javascripts/angular/hello_world_controller.js.coffee

```
1 AngulaRails.controller "HelloWorldController", ($scope) ->
2   $scope.myText = "Hello World"
```

Line 1: `$scope`

`$scope` is an injected variable on which we attach properties and methods we want to reference in our views.

Line 2: `$scope.myText`

We don't have to initialize `$scope.myText` to use that property on our view, but for demonstration purposes we will initialize it with "Hello World".

Back in our view, we can now reference the `AngulaRails` application and the `HelloWorldController` we just created.

views/hello_angular/show.html.erb

```
1 <div class="row" ng-app="AngulaRails" ng-controller="HelloWorldController">
2   <div class="col-md-6">
3     <h1>Hello AngularJS</h1>
4     <p>Type in the input box and watch as AngularJS updates the property through \
5 the magic of binding.</p>
6     <input type="text" class="form-control" ng-model="myText" >
7     {{ myText }}
8   </div>
9 </div>
```

Line 1: `ng-app`

This sets the Angular application to our `AngulaRails` that we created in `angular-application.js`.

Line 1: `ng-controller`

This will set our controller to the `HelloWorldController` we created in `hello_world_controller.js.coffee`.

Try it out!

Reload the page at `http://localhost:3000/hello_angular/show`. Now the input should start with the "Hello World" value we set in our controller.

Conclusion

Nothing crazy or exciting right? Just basic wiring and some simple setup. Future exercises in this guide will assume this setup. We recommend saving a base copy of this Rails + AngularJS application so you can use it as a common starting point.

5 Forms with AngularJS

The goal of this chapter is to explore how to build a dynamic form which binds all the form elements interactively. Our order form handles Tacos. Mmmmmmmmm Tacos.

What is covered:

- Binding to text input with `ng-model`
- Binding form submission with `ng-submit`
- Binding to `select` elements using `ng-options`
  - using premade options
  - using options from binding
- Binding to `checkbox` elements
  - using regular checkboxes
  - using dynamic list of checkboxes
- Binding to `radio` elements
  - using regular radio elements
  - using dynamic list of radio buttons

Basic Setup

terminal

```
$ rails g controller TacoOrder edit
```

This will create a Rails `TacoOrder` controller with a `edit.html.erb` view.

Let's add some basic markup to layout our Taco Order page.

views/taco_order/edit.html.erb

```
<div class="row">
  <div class="col-md-6">
    <h3>Add Taco to Order</h3>

    <form>
      <div class="form-group">
        <label for="name">Name your creation</label>
        <input type="text" class="form-control" placeholder="Super Tasy"></input>
```

```
9         </div>
10        <button class="btn btn-primary">
11          Submit
12        </button>
13      </form>
14    </div>
15    <div class="col-md-6">
16      <h3>Your Order</h3>
17
18      <table class="table">
19        <tbody>
20        </tbody>
21      </table>
22    </div>
23  </div>
```

Now we will create our AngularJS controller. We define a holding place for all of our tacos ($scope.tacos), and an object for a taco that is being currently created and edited ($scope.taco).

javascripts/angular/taco_order_controller.js.coffee

```
1  AngulaRails.controller "TacoOrderController", ($scope) ->
2    $scope.tacos = []
3    $scope.taco = {}
4
5    $scope.clear = () ->
6      $scope.taco = {}
7
8    $scope.addTaco = () ->
9      $scope.tacos.push($scope.taco)
10
11     $scope.clear()
```

Line 8

When "Submit" is clicked, we will call the `addTaco` method. In this method, we will simply push the `$scope.taco` object straight onto our `$scope.tacos` array.

Time to wire up our controller and `addTaco` method to the view.

views/taco_order/edit.html.erb

```
1  <div class="row" ng-app="AngulaRails" ng-controller="TacoOrderController">
2    <div class="col-md-6">
3      <h3>Add Taco to Order</h3>
4
5      <form ng-submit="addTaco()">
6        <div class="form-group">
7          <label for="name">Name your creation</label>
8          <input type="text" class="form-control" placeholder="Super Tasy" ng-model\
9  ="taco.name"></input>
10       </div>
11       <button class="btn btn-primary">
12         Submit
13       </button>
14     </form>
15   </div>
16   <div class="col-md-6">
17     <h3>Your Order</h3>
18
19     <table class="table">
20       <tbody>
21         <tr ng-repeat="taco in tacos">
22           <td>{{ taco.name }}</td>
23         </tr>
24       </tbody>
25     </table>
26   </div>
27 </div>
```

Line 1 `ng-app` & `ng-controller`
Set the Angular app and the controller.

Line 5 `ng-submit`
On form submission, we will execute the `addTaco` method on our controller.

Line 9 `ng-model`
Bind the input to the `name` property on our `$scope.taco` object.

Line 19 `ng-repeat`
We can display the items in our array using `ng-repeat`.

Try it out!

Go to `http://localhost:3000/taco_order/edit` We should be able to name our taco and add it to our order table.

5.1 Using Selects

For our tacos, we will need to be able to select a super tasty filling. We are going to use a `select` and demonstrate several methods of connecting a `select` form element with AngularJS.

views/taco_order/edit.html.erb

```
1  <form ng-submit="addTaco()">
2    <div class="form-group">
3      <label for="name">Name your creation</label>
4      <input type="text" name="name" class="form-control" placeholder="Super Tasty"\
5   ng-model="taco.name"></input>
6    </div>
7    <div class="form-group">
8      <label for="filling">Select your filling</label>
9      <select ng-model="taco.filling" class="form-control">
10       <option>Beef</option>
11       <option>Chicken</option>
12       <option>Fish</option>
13     </select>
14   </div>
15   <button class="btn btn-primary">
16     Submit
17   </button>
18 </form>
```

Line 9 `ng-model`

Bind the selected option's value to the `filling` property on our `$scope.taco` object.

Display filling choice in our taco table.

views/taco_order/edit.html.erb

```
<table class="table">
  <tbody>
    <tr ng-repeat="taco in tacos">
      <td>{{ taco.name }}</td>
      <td>{{ taco.filling }}</td>
    </tr>
  </tbody>
</table>
```

Try it out!

Go to `http://localhost:3000/taco_order/edit` We should be able to select a filling for our taco and see the filling when we add it to our order table.

Use Binding for Select Options

What if we want the options for our fillings to be dynamically created?

javascripts/angular/taco_order_controller.js.coffee

```
AngulaRails.controller "TacoOrderController", ($scope) ->
  $scope.tacos = []
  $scope.taco = {}
  $scope.fillings = ["Beef", "Chicken", "Fish", "Carnitas"]

  $scope.clear = () ->
    $scope.taco = {}

  $scope.addTaco = () ->
    $scope.tacos.push($scope.taco)

    $scope.clear()
```

Bind to `$scope.fillings` to create options for select using `ng-options`.

views/taco_order/edit.html.erb

```
<div class="form-group">
  <label for="filling">Select your filling</label>
  <select ng-model="taco.filling" class="form-control" ng-options="f for f in fil\
lings">
  </select>
</div>
```

With `ng-options` I find it easier to read this from right to left.

For each filling `f` in `$scope.fillings` Set the option value and text to `f`.

If we have a collection of objects instead of just an array of strings, we could use the first part to declare which property we wanted to use for the text and value.

```
$scope.fillings = [ {id: 1, name: "Pollo"}, {id:2, name: "Carne Asada"}]
```

If we wanted to use the id for the value and the name as the label, we would setup the binding like:

```
ng-options="f.id as f.name for f in fillings"
```

Having a Default / Null Value

If we wanted a null value / default message in our select, we can just set the option directly in the markup.

views/taco_order/edit.html.erb

```
<div class="form-group">
  <label for="filling">Select your filling</label>
  <select ng-model="taco.filling" class="form-control" ng-options="f for f in fil\
lings">
    <option value="">Pick a filling</option>
  </select>
</div>
```

Try it out!

Go to `http://localhost:3000/taco_order/edit` We should now have a default value and our options are populated from binding.

5.2 Using Checkboxes

Next we are going to use checkboxes to allow a user to select different options for their Taco.

views/taco_order/edit.html.erb

```
<form ng-submit="addTaco()">
  <!-- Rest of form -->
  <div class="form-group">
    <label>Extras</label>
    <div class="checkbox">
      <label>
        <input type="checkbox" ng-model="taco.sourCream"></input>
        Sour cream
      </label>
    </div>
    <div class="checkbox">
      <label>
        <input type="checkbox" ng-model="taco.guac"></input>
        Guac
      </label>
    </div>
  </div>
  <button class="btn btn-primary">
    Submit
  </button>
</form>
```

And add a spot to display the extra choice in our taco table.

views/taco_order/edit.html.erb

```
<table class="table">
  <tbody>
    <tr ng-repeat="taco in tacos">
      <td>{{ taco.name }}</td>
      <td>{{ taco.filling }}</td>
      <td>
        <span ng-show="taco.sourCream">Sour Cream</span>
        <span ng-show="taco.guac">Guac</span>
      </td>
    </tr>
  </tbody>
</table>
```

Line 7 `ng-show`

Display `span` if `taco.sourCream` is `true`. Line 8 `ng-show`

Display `span` if `taco.guac` is `true`

Try it out!

Reload `http://localhost:3000/taco_order/edit` We should be able to select which extras we want on our taco.

Using binding to create check boxes

Instead of hard coding the extras, lets get the extras from an array on the controller.

javascripts/angular/taco_order_controller.js.coffee

```
1  AngulaRails.controller "TacoOrderController", ($scope, $filter) ->
2    $scope.tacos = []
3    $scope.taco = {}
4    $scope.fillings = ["Beef", "Chicken", "Fish", "Carnitas"]
5    $scope.extras = [ { name: "Sour Cream" }, { name: "Guac" }, { name: "Salsa" }]
6
7    $scope.clear = () ->
8      $scope.taco = {}
9      for extra in $scope.extras
10       extra.checked = false
11
12   $scope.getExtras = () ->
13     extras = []
14     for extra in $scope.extras
15       if extra.checked == true
16         extras.push(extra.name)
17     extras
18
19   $scope.addTaco = () ->
20     $scope.taco.extras = $scope.getExtras()
21     $scope.tacos.push($scope.taco)
22     $scope.clear()
```

Line 4

Set the `extras` array as an array of objects with a `name` property. Line 8

To clear our selections, we need to loop through our `extras` array and set all the `checked` properties to false. Line 11

Since we are not binding directly to `taco` we need to take return a new array with only the `checked` extras. Line 19

Set the extras on `taco` to the new array of only the selected extras returned by our `getExtras()` method.

Bind to `$scope.extras`:

views/taco_order/edit.html.erb

```
1  <div class="form-group">
2    <label>Extras</label>
3    <div class="checkbox" ng-repeat="extra in extras">
4      <label>
5        <input type="checkbox" ng-model="extra.checked"></input>
6        {{ extra.name }}
7      </label>
8    </div>
9  </div>
```

Display the new `extras` array on taco:

views/taco_order/edit.html.erb

```
1  <table class="table">
2    <tbody>
3      <tr ng-repeat="taco in tacos">
4        <td>{{ taco.name }}</td>
5        <td>{{ taco.filling }}</td>
6        <td>
7          <span ng-repeat="extra in taco.extras">
8            {{ extra }}
9          </span>
10       </td>
11     </tr>
12   </tbody>
13 </table>
```

Try it out!

Reload `http://localhost:3000/taco_order/edit` We should be able to select which extras we want on our taco. Just like before.

5.3 Using Radio Buttons

We will be using radio buttons to select our cheese options.

views/taco_order/edit.html.erb

```
<div class="form-group">
  <label>Cheese</label>
  <div class="radio">
    <label>
      <input type="radio" value="No Cheese" ng-model="taco.cheese"></input>
      No Cheese
    </label>
  </div>
  <div class="radio">
    <label>
      <input type="radio" value="Normal Cheese" ng-model="taco.cheese"></input>
      Normal Cheese
    </label>
  </div>
  <div class="radio">
    <label>
      <input type="radio" value="Mucho Queso" ng-model="taco.cheese"></input>
      Mucho Queso
    </label>
  </div>
</div>
```

And add a spot to display the cheese choice in our taco table

views/taco_order/edit.html.erb

```
<table class="table">
  <tbody>
    <tr ng-repeat="taco in tacos">
      <td>{{ taco.name }}</td>
      <td>{{ taco.filling }}</td>
      <td>
        <span ng-repeat="extra in taco.extras">
          {{ extra }}
        </span>
      </td>
      <td>{{ taco.cheese }}</td>
    </tr>
  </tbody>
</table>
```

Try it out!

Reload `http://localhost:3000/taco_order/edit` We should be able to select different cheese options for your taco.

Use binding to create Radio Buttons

javascripts/angular/taco_order_controller.js.coffee

```
AngulaRails.controller "TacoOrderController", ($scope) ->
  $scope.tacos = []
  $scope.taco = {}
  $scope.fillings = ["Beef", "Chicken", "Fish", "Carnitas"]
  $scope.extras = [ { name: "Sour Cream" }, { name: "Guac" }, { name: "Salsa" }]
  $scope.cheeses = ["No Cheese", "Normal Cheese", "Mucho Queso", "Monterrey Jack"]
```

Bind to `$scope.cheeses`.

views/taco_order/edit.html.erb

```
<div class="form-group">
  <label>Cheese</label>
  <div class="radio" ng-repeat="cheese in cheeses">
    <label>
      <input type="radio" value="{{ cheese }}" ng-model="taco.cheese"></input>
      {{ cheese }}
    </label>
  </div>
</div>
```

Try it out!

Reload the page at `http://localhost:3000/taco_order/edit`. You can add and edit delicious, delicious tacos.

Conclusion

How cool is that? You created an in-memory checkout process with AngularJS! In this chapter we demonstrated 80% of the scenarios you will run into while creating forms with AngularJS. Often times this seems an easy no brainer task, but you can run into snags when you start creating your own form. This content will be a handy reference for you when you run into any issues.

6 Models Need Dots

The golden rule of Angular binding is that `ng-model` should be bound to models with dots. The reason behind this is that you can easily burn yourself if you don't understand AngularJS binding and scoping. In this short chapter we will demonstrate a simple example of what happens when you do not follow the golden rule.

What is covered:

- Binding scope
- Why the "models need dots" golden rule is a good rule to follow

Setup

Create a `ModelsNeedDots` controller with a `show.html.erb` view.

terminal

```
1 $ rails g controller ModelsNeedDots show
```

Lets create a simple AngularJS controller with an array of stuff to bind to.

javascripts/angular/models_need_dots_controller.js.coffee

```
1 AngulaRails.controller "ModelsNeedDotsController", ($scope, $http) ->
2   $scope.stuff = [1,2,3]
```

6.1 Dangerous Binding

We are going to start with a text input that we bind to a `message` property.

We are going to use stuff to create 3 list items. Inside each `li` we are going to add another text input and bind to the same `message` property.

views/models_need_dots/show.html.erb

```
<div ng-app="AngulaRails" ng-controller="ModelsNeedDotsController">
  <div class="row">
    <div class="col-md-12">
      <h1>Oh noes!</h1>
      Message is "{{ message }}"
      <input type="text" class="form-control" ng-model="message" />
      <ul class="list-unstyled">
        <li ng-repeat="s in stuff" style="margin-top: 10px">
          Message is "{{ message }}"
          <input type="text" class="form-control" ng-model="message" />
        </li>
      </ul>
    </div>
  </div>
</div>
```

Try it out!

Go to `http://localhost:3000/models_need_dots/show`. When you type in the first text box, notice it updates all the other `messages`. Now try out the other inputs. Oh noes! The `message` is only updated in the nearby binding. #### OMG ITS A BUG! Nope. Not a bug.

6.2 Correct Binding

The previous binding doesn't work because each subview under the `ng-repeat` has their own scope. So the binding to `message` is bound to the message property of its own scope.

By making `message` a property on another object, we ensure that the child scopes look up to a parent scope for to find the object to bind to.

views/models_need_dots/show.html.erb

```
<div ng-app="AngulaRails" ng-controller="ModelsNeedDotsController">
  <div class="row">
    <div class="col-md-12">
      <h1>Oh noes!</h1>
      Message is "{{ someModel.message }}"
      <input type="text" class="form-control" ng-model="someModel.message" />
      <ul class="list-unstyled">
        <li ng-repeat="s in stuff" style="margin-top: 10px">
          Message is "{{ someModel.message }}"
          <input type="text" class="form-control" ng-model="someModel.message" />
        </li>
      </ul>
    </div>
  </div>
</div>
```

Now the parent and child scopes are all looking at the same object with the same `message` property.

Try it out!

Reload to `http://localhost:3000/models_need_dots/show` Now when we type in any of the inputs, message is updated everywhere. #### CRISIS AVERTED!

Conclusion

When you first run into this issue (and you will), you will be totally confused and think it is a bug. Once you understand the mechanics behind this pitfall, it makes perfect sense. Just to be safe, remember the golden rule of AngularJS binding. `ng-model` needs dots.

7 Extending Rails Forms with AngularJS Validation

One of the amazing things you can do with AngularJS is extend exisiting Rails forms. Crazy, right?

You can just add AngularJS directives to your existing Rails form view to provide additional functionality. Your Rails backend will remain unchanged. The form POST will still go to your controller. Even your tests will still pass!

What is covered:

- Adding AngularJS to an existing Rails form
- AngularJS HTML5 validation
- AngularJS validation
- Toggle classes with `ng-class`
- Toggle visibility of an HTML element using `ng-show`
- Disable a button using `ng-disable`

Setup

For our example, we are going to make Widgets. Widgets have a name and a price.

terminal

```
$ rails g scaffold Widget name:string price:integer
$ rake db:migrate
```

Now there is some important validation for making a Widget. * Name is required and must be at least 7 characters * Price is required and must be a number greater than 10.

Why? Because of reasons.

This validation is straight forward on the Rails side.

models/widget.rb

```
class Widget < ActiveRecord::Base
  validates :price, presence: true, numericality: { greater_than: 10 }
  validates :name, presence: true, length: { minimum: 7 }
end
```

We will need to change our scaffolded for a bit to look like this:

views/widgets/_form.html.erb

```
<%= form_for(@widget) do |f| %>
  <% if @widget.errors.any? %>
    <div class="alert alert-danger">
      <h4><%= pluralize(@widget.errors.count, "error") %></h4>

      <ul>
      <% @widget.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="form-group">
    <%= f.label :name %>
    <%= f.text_field :name, class: "form-control" %>
  </div>
  <div class="form-group">
    <%= f.label :price %>
    <%= f.number_field :price, class: "form-control" %>
  </div>
  <%= f.submit "Save", class: "btn btn-primary" %>
<% end %>
```

Try it out!

If we go to `http://localhost:3000/widgets`, we can then use the Rails scaffolding to add widgets. If we try to create an invalid widget, Rails will return the validation error messages.

7.1 HTML5 Validation with AngularJS

Let's add our AngularJS application to the form and turn off browser default validation. We also will need to give our form a name for validation in AngularJS to work.

views/widgets/_form.html.erb

```
<div ng-app="AngulaRails">
  <%= form_for @widget, html: { name: "widgetForm", "novalidate" => true } do |f|\
 %>
    <% if @widget.errors.any? %>
      <div class="alert alert-danger">
        <h4><%= pluralize(@widget.errors.count, "error") %></h4>

        <ul>
        <% @widget.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
        </ul>
      </div>
    <% end %>
    <div class="form-group">
      <%= f.label :name %>
      <%= f.text_field :name, class: "form-control" %>
    </div>
    <div class="form-group">
      <%= f.label :price %>
      <%= f.number_field :price, class: "form-control" %>
    </div>
    <%= f.submit "Save", class: "btn btn-primary" %>
  <% end %>
</div>
```

AngularJS can use the HTML5 validation attributes for its validation. Since both fields are required, lets add the `required` attribute to each input. We can also add the HTML5 validation attribute `min` to the price input.

Lets add some styling so the input fields will change color to show if the value is invalid.

And finally, we will disable the `Save` button if the form is invalid.

views/widgets/_form.html.erb

```
<div ng-app="AngulaRails">
  <%= form_for @widget, id: "widgetForm", html: { name: "widgetForm", "novalidate\
" => true } do |f| %>
    <% if @widget.errors.any? %>
      <div class="alert alert-danger">
        <h4><%= pluralize(@widget.errors.count, "error") %></h4>

        <ul>
        <% @widget.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
        </ul>
      </div>
    <% end %>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[name]'].$\
invalid }">
      <%= f.label :name %>
      <%= f.text_field :name, class: "form-control", "ng-model" => "name", requir\
ed: true %>
    </div>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[price]'].\
$invalid }">
      <%= f.label :price %>
      <%= f.number_field :price, class: "form-control", "ng-model" => "price", re\
quired: true, "min" => "10" %>
    </div>
    <%= f.submit "Save", class: "btn btn-primary", "ng-disabled" => "widgetForm.$\
invalid" %>
  <% end %>
</div>
```

Try it out!

Go to `http://localhost:3000/widgets/new` The inputs should have a red border signifying they are invalid. The submit button should also disabled. Once you type something in the input, the invalid border should go away.

Display the error message

While the red border is helpful, it would be nice to show an error message to the user so they know why the input is invalid.

views/widgets/_form.html.erb

```
<div ng-app="AngulaRails">
  <%= form_for @widget, id: "widgetForm", html: { name: "widgetForm", "novalidate\
" => true } do |f| %>
    <% if @widget.errors.any? %>
      <div class="alert alert-danger">
        <h4><%= pluralize(@widget.errors.count, "error") %></h4>

        <ul>
        <% @widget.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
        </ul>
      </div>
    <% end %>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[name]'].$\
invalid }">
      <%= f.label :name %>
      <%= f.text_field :name, class: "form-control", "ng-model" => "name", requir\
ed: true %>
      <span class="help-block" ng-show="widgetForm['widget[name]'].$error.require\
d">Required</span>
    </div>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[price]'].\
$invalid }">
      <%= f.label :price %>
      <%= f.number_field :price, class: "form-control", "ng-model" => "price", re\
quired: true, "min" => "10" %>
      <span class="help-block" ng-show="widgetForm['widget[price]'].$error.requir\
ed">A number is required</span>
      <span class="help-block" ng-show="widgetForm['widget[price]'].$error.min">M\
ost be greater than 10</span>
    </div>
    <%= f.submit "Save", class: "btn btn-primary", "ng-disabled" => "widgetForm.$\
invalid" %>
  <% end %>
</div>
```

Try it out!

Go to `http://localhost:3000/widgets/new` Now we have an appropriate error message to inform the user why the input is invalid.

7.2 Using AngularJS Validation

Not all our validations fall within HTML5 validation. AngularJS provides a supplement to those validation rules. The API documentation found at `http://angularjs.org/` lists all the validations AngularJS provides and naturally this will change over time. For our example, we are going to use the `ng-minlength` validation directive to enforce our name length validation rule.

views/widgets/_form.html.erb

```
<div ng-app="AngulaRails">
  <%= form_for @widget, id: "widgetForm", html: { name: "widgetForm", "novalidate\
" => true } do |f| %>
    <% if @widget.errors.any? %>
      <div class="alert alert-danger">
        <h4><%= pluralize(@widget.errors.count, "error") %></h4>

        <ul>
        <% @widget.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
        </ul>
      </div>
    <% end %>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[name]'].$\
invalid }">
      <%= f.label :name %>
      <%= f.text_field :name, class: "form-control", "ng-model" => "name", requir\
ed: true, "ng-minlength" => "7" %>
      <span class="help-block" ng-show="widgetForm['widget[name]'].$error.require\
d">Required</span>
      <span class="help-block" ng-show="widgetForm['widget[name]'].$error.minleng\
th">Minimum length of 7</span>
    </div>
    <div class="form-group" ng-class="{ 'has-error': widgetForm['widget[price]'].\
```

```
$invalid }">
      <%= f.label :price %>
      <%= f.number_field :price, class: "form-control", "ng-model" => "price", re\
quired: true, "min" => "10" %>
      <span class="help-block" ng-show="widgetForm['widget[price]'].$error.requir\
ed">Required</span>
      <span class="help-block" ng-show="widgetForm['widget[price]'].$error.min">M\
ost be greater than 10</span>
    </div>
    <%= f.submit "Save", class: "btn btn-primary", "ng-disabled" => "widgetForm.$\
invalid" %>
  <% end %>
</div>
```

Try it out!

Go to `http://localhost:3000/widgets/new` With our final addition, our validation for a Widget is enforced on the client and server side. The user is also presented with a better user experience.

Conclusion

The ability to extend an existing Rails form highlights AngularJS's great flexibility. You can provide a greater user experience with very little effort. In this example we simply added validation, but extending a Rails form with AngularJS can be used for other features.

8 HTTP with AngularJS

Life on the web would be boring without HTTP requests. Have no fear, in this chapter we will be making HTTP requests to GitHub to demonstrate the basics on binding to data that is pulled from GET requests.

What we will cover:

- Using the `$http` provider to make HTTP requests
- Another use of our good friends `ng-model` and `ng-submit`
- Using `ng-repeat` to loop over an array
- Using `ng-show` to show an HTML element
- Using jQuery to make a HTTP request

Setup

First we will create a `GitHttp` controller with a `show.html.erb` view.

terminal

```
> rails g controller GitHttp show
```

8.1 GET request with $http

We are going to start with our controller and make a GET request to GitHub using AngularJS's `$http` provider. Like the `$scope` provider, `$http` is a service that AngularJS provides which can be injected into your controller for use.

javascripts/angular/git_http_controller.js.coffee

```
AngulaRails.controller "GitHttpController", ($scope, $http) ->

  $scope.search = () ->
    url = "https://api.github.com/users/#{$scope.username}/repos"
    $http({ method: "GET", url: url })
      .success (data) ->
        $scope.repos = data
```

Line 1 `$http`

AngularJS comes packaged with a set of services. `$http` allows for async http requests using promises. Line 4 Getting the username

We use the current value of `$scope.username` to create the url to the GitHub api. Line 7 Binding to JSON

With Angular, we can just stick the JSON we get back from the http request right into our `$scope`. No need to wrap it in any other object.

Now let's create the search from on the `show` page.

views/git_http/show.html.erb

```
1  <div class="row" ng-app="AngulaRails" ng-controller="GitHttpController">
2    <div class="col-md-6">
3      <h1>Git $http</h1>
4      <form ng-submit="search()">
5        <div class="input-group">
6          <span class="input-group-addon">
7            <i class="icon-search"></i>
8          </span>
9          <input type="text" class="form-control" placeholder="Git Username" ng-mod\
10 el="username">
11       </div>
12     </form>
13   </div>
14 </div>
```

Line 1 `ng-app` & `ng-controller`

Set our application and our controller. Line 4 `ng-submit`

When we submit this form, execute the `search()` method on our controller. Notice in this case we don't have a submit button. Search is initiated by hitting return while inside the input Line 7 `ng-model`

Bind the value of the input to `username`. We then use this value when we execute our search.

Now lets take the array of git hub repos we pulled down with our $http GET request and display them in a list.

views/git_http/show.html.erb

```
1  <div class="row" ng-app="AngulaRails" ng-controller="GitHttpController">
2    <div class="col-md-6">
3      <h1>Git $http</h1>
4      <!-- THE FORM -->
5
6      <hr />
7
8      <ul class="list-group">
9        <li class="list-group-item" ng-repeat="repo in repos">
10         <h4 class="list-group-item-heading">{{ repo.full_name }}</h4>
11         <p class="list-group-item-text text-muted">{{ repo.url }}</p>
12       </li>
13     </ul>
14   </div>
15 </div>
```

Line 6 `ng-repeat`

This will iterate over all the objects in the array found in the `repos` property. Each object is set to `repo`. Line 8,9 Bind to `repo`

We can now use the `repo` variable set by `ng-repeat` to display properties on the object.

Try it out!

Go to `http://localhost:3000/git_http/show`. If you search for a valid username, you should then see the list of their public repos. If you don't have a GitHub account, then you can search on our usernames (`rookieone` or `jwo`).

8.2 Adding a spinner

Let's clear the repos and add a spinner to display as we make our request.

javascripts/angular/git_http_controller.js.coffee

```
1  AngulaRails.controller "GitHttpController", ($scope, $http) ->
2
3    $scope.search = () ->
4      $scope.repos = []
5      $scope.searching = true
6      url = "https://api.github.com/users/#{$scope.username}/repos"
7      $http({ method: "GET", url: url })
8        .success (data) ->
9          $scope.searching = false
10         $scope.repos = data
```

Line 4: Clear repos

Clear array in `$scope.repos`. Line 5

Set searching to true before we make our http request. Line 9

Set searching to false after we get our result.

views/git_http/show.html.erb

```
1  <i class="icon-refresh icon-spin" ng-show="searching"></i>
2
3  <ul class="list-group">
4    <li class="list-group-item" ng-repeat="repo in repos">
5      <h4 class="list-group-item-heading">{{ repo.full_name }}</h4>
6      <p class="list-group-item-text text-muted">{{ repo.url }}</p>
7    </li>
8  </ul>
```

Line 1: `ng-show`

Display the HTML element if `searching` evaluates as true. In this scenario we are looking the `searching` property, but this could be a javascript expression. ie. `repos.length == 0`

FYI ng-hide

There is an `ng-hide` as well. Instead of showing an element when the expression evaluates to true, it hides the element.

Try it out!

Reload `http://localhost:3000/git_http/show`. The search is pretty fast, but you should see a spinner briefly between searches. You can also add a timeout to slow the search down.

8.3 Handle Errors

We need to handle 404 errors for the times we search for a user that doesn't exist.

javascripts/angular/git_http_controller.js.coffee

```
1  $scope.search = () ->
2    $scope.repos = []
3    $scope.searching = true
4    $scope.errorMessage = ""
5    url = "https://api.github.com/users/#{$scope.username}/repos"
6    $http({ method: "GET", url: url })
7      .success (data) ->
8        $scope.searching = false
9        $scope.repos = data
10     .error (data, status) ->
11       $scope.searching = false
12       if status == 404
13         $scope.errorMessage = "User not found"
```

Line 4

Clear `$scope.errorMessage`. Line 10

Handle errors from our http request. We can look at the data from the response along with the status code. Line 13

Set `$scope.errorMessage` to a "User not found" message.

And now we need to display the `errorMessage` on our view.

views/git_http/show.html.erb

```
1  <div class="alert alert-danger" ng-show="errorMessage">
2    <strong><i class="icon-warning-sign"></i> Error!</strong>
3    {{ errorMessage }}
4  </div>
5
6  <i class="icon-refresh icon-spin" ng-show="searching"></i>
7
8  <ul class="list-group">
9    <li class="list-group-item" ng-repeat="repo in repos">
10     <h4 class="list-group-item-heading">{{ repo.full_name }}</h4>
11     <p class="list-group-item-text text-muted">{{ repo.url }}</p>
12   </li>
13 </ul>
```

Line 1

Display the `errorMessage` that we set in our `search` method.

Try it out!

Reload `http://localhost:3000/git_http/show`. This time search for some random user that can't possibly exist. How about `angularailsissuperawesome`?

8.4 Using jQuery

You should really strive to use AngularJS's `$http`, but sometimes you just have to fall back to what you know. Also, this is a great demonstration of the flexibility of AngularJS, and how awesome it is to integrate with third party libraries that operate outside of it. However, don't learn to use jQuery ajax for all your HTTP requests. You have been warned!

javascripts/angular/git_http_controller.js.coffee

```
1  $scope.search = () ->
2    $scope.repos = []
3    $scope.searching = true
4    $scope.errorMessage = ""
5    url = "https://api.github.com/users/#{$scope.username}/repos"
6    $.ajax
7      type: "GET"
8      url: url
9      success: (data) ->
10       $scope.searching = false
11       $scope.repos = data
12       $scope.$apply()
13     error: (error) ->
14       $scope.searching = false
15       if error.status == 404
16         $scope.errorMessage = "User not found"
17       $scope.$apply()
```

Line 6 `$.ajax`

Yes... jQuery in our Angular controller. Keep it secret! Line 12,17 `$scope.$apply()`

Since the jQuery ajax request occurs outside the eyes of Angular, we need to directly inform Angular to update all the bindings.

Try it out!

Reload `http://localhost:3000/git_http/show`. Everything should work the same as before... except underneath the covers we are being very naughty and using jquery.

Conclusion

The building block basics of using AngularJS are being revealed piece by piece. In this chapter we learned how to make HTTP requests using `$http`. We also saw how easily AngularJS can integrate with third party JavaScript libraries that operate outside AngularJS's normal scope.

9 Talking to Rails

Aren't we supposed to be talking about Rails somewhere in this book? Well in this chapter we are going to be using our AngularJS knowledge to start talking to Rails. We will be doing basic CRUD operations on a simple model.

What we will cover:

- Adding Rails's token to prevent CSRF
- Setting default header options for `$http`

Setup

This will create a `Book` model along with related CRUD controllers. Most importantly, it will include JSON api endpoints that we will use to perform CRUD operations.

terminal

```
1 $ rails g scaffold Book title:string author:string
```

9.1 HTTP Header Defaults for Rails

Before we dive into doing CRUD on Rails, we will configure the `$http` service to play nicely with Rails.

First we are going to pull the CSRF token from our page. The Cross Site Request Forgery protection token is used by Rails to prevent security threats from cross site requests. If we did not set the header, we would receive a `422` / "Can't verify CSRF token authenticity" error for every non-GET request.

Next we are going to change the default `Accept` header to ask for `application/json`. The default for AngularJS accepts multiple formats allowing Rails to decide to return HTML instead of JSON.

javascripts/angular/http.js.coffee

```
AngulaRails.config ["$httpProvider", ($httpProvider) ->
  $httpProvider.defaults.headers.common['X-CSRF-Token'] = $('meta[name=csrf-token\
]').attr('content')
  $httpProvider.defaults.headers.common.Accept = "application/json"
]
```

9.2 GET /books

We first need to update our Rails `BookController`.

controllers/books_controller.rb

```
# GET /books
def index
  @books = Book.all

  respond_to do |format|
    format.html { }
    format.json { render json: @books }
  end
end
```

Lets replace the Rails scaffolding for `index.html.erb` to use AngularJS.

views/books/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="BooksController" ng-init="getBooks()">
  <div class="row">
    <div class="col-md-6">
      <h1>Books</h1>
      <ul>
        <li ng-repeat="book in books">
          {{ book.title }} by {{ book.author }}
        </li>
      </ul>
    </div>
  </div>
</div>
```

Line 1,2 `ng-init`

With `ng-init` we can call our `getBooks()` method on our controller after the page is loaded.

Let's create our Angular controller and get a list of books from the Rails app.

javascripts/angular/books_controller.js.coffee

```
1 AngulaRails.controller "BooksController", ($scope, $http) ->
2   $scope.getBooks = () ->
3     $http({ method: "GET", url: "/books" })
4       .success (response) ->
5         $scope.books = response.books
```

Line 1 `$http`

We are going to use the `$http` service to talk to our Rails server.

Line 3 `/books`

We will call the books url that we know exists.

If we looked at the page right now, we would see nothing because we have no books! Open rails console and add some books.

terminal

```
1 $ rails console
2 irb> Book.create(title: "The Hobbit", author: "J.R.R. Tolkien")
3 irb> Book.create(title: "Ender's Game", author: "Orson Scott Card")
```

Try it out!

Go to `http://localhost/books` You should see the list of the books you just added in the console.

9.3 Pass in the url

Instead of using a string to build the url to call, let's have Rails pass in the url for us to use.

controllers/books_controller.rb

```ruby
# GET /books
def index
  @books = Book.all

  respond_to do |format|
    format.html {
      @urls = {
        books: books_path
      }
    }
    format.json { render json: @books }
  end
end
```

Now we can pass the `@urls` hash into our Angular controller as a JSON object.

views/books/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="BooksController" ng-init="urls=<%= @urls\
.to_json %>; getBooks()">
```

javascript/angular/books_controller.js.coffee

```
AngulaRails.controller "BooksController", ($scope, $http) ->
  $scope.getBooks = () ->
    $http({ method: "GET", url: $scope.urls.books })
      .success (response) ->
        $scope.books = response.books
```

Try it out!

Reload `http://localhost/books` You should see the same list of the books but now we pass in the url to use.

9.4 Pass data from Rails to AngularJS

We have the books when we build the page, so why do we need to make an ajax call? Let's pass in more data from Rails to AngularJS.

app/serializers/book_serializer.rb

```
class BookSerializer < ActiveModel::Serializer
  attributes :id, :title, :author, :url

  def url
    book_path(object)
  end
end
```

controllers/books_controller.rb

```
# GET /books
def index
  @books = Book.all

  respond_to do |format|
    format.html {
      @books_json = @books.map{ |b| BookSerializer.new(b).serializable_hash }
      @urls = {
        books: books_path
      }
    }
    format.json { render json: @books }
  end
end
```

Now we can pass in the `@books_json` array of hashes and not call the `getBooks()` method.

views/books/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="BooksController" ng-init="urls=<%= @urls\
.to_json %>; books=<%= @books_json.to_json %>">
```

Try it out!

Reload `http://localhost/books` You should see the same list of the books but now with no GET request.

9.5 POST /books

Now let's create a book. First we will add a form to the page.

views/books/index.html.erb

```
<form ng-submit="save()">
  <div class="form-group">
    <label>Title</label>
    <input type="text" class="form-control" ng-model="book.title" />
  </div>
  <div class="form-group">
    <label>Author</label>
    <input type="text" class="form-control" ng-model="book.author" />
  </div>
  <button class="btn btn-primary">
    Save
  </button>
</form>
```

javascripts/angular/books_controller.js.coffee

```
AngulaRails.controller "BooksController", ($scope, $http) ->

  $scope.getBooks = () ->
    $http({ method: "GET", url: $scope.urls.books })
      .success (response) ->
        $scope.books = response.books

  $scope.save = () ->
    $http({ method: "POST", url: $scope.urls.books, data: $scope.book })
      .success (response) ->
        $scope.book = {}
        $scope.getBooks()
```

For our Rails controller, we will update the create method to look like:

controllers/books_controller.rb

```ruby
# POST /books
def create
  @book = Book.create(book_params)

  respond_to do |format|
    if @book.save
      format.html { redirect_to @book, notice: 'Book was successfully created.' }
      format.json { render json: @book }
    else
      format.html { render action: 'new' }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
```

Try it out!

Reload to `http://localhost:3000/books` We are now able to add books using AngularJS posting to a Rails endpoint. If you recieve a `422` error, be sure to update your default headers from earlier in the chapter.

9.6 DELETE /books/:id

Deleting a book is straight forward. All we need to do is make a DELETE request to the Rails endpoint.

views/books/index.html.erb

```
<h1>Books</h1>
<ul>
  <li ng-repeat="book in books">
    <div class="btn btn-default" ng-click="delete(book)">
      <i class="icon-trash" />
    </div>
    {{ book.title }} by {{ book.author }}
  </li>
</ul>
```

javascripts/angular/books_controller.js.coffee

```
  $scope.delete = (book) ->
    $http({ method: "DELETE", url: book.url })
      .success (response) ->
        $scope.getBooks()
```

Notice that we send the DELETE request to the url returned by Rails when it serialized `Book`. By default the url is not added when we scaffold the serializer. Let's add the url to our serializer.

serializers/book_serializer.rb

```
class BookSerializer < ActiveModel::Serializer
  attributes :id, :title, :author, :url

  def url
    book_path(object)
  end
end
```

We need to also adjust our destroy method on our `BooksController`

controllers/books_controller.rb

```
# DELETE /books/1
def destroy
  @book.destroy
  respond_to do |format|
    format.html {
      redirect_to books_url, notice: 'Book was deleted'
    }
    format.json { render json: { message: "Book was deleted" } }
  end
end
```

Try it out!

Reload to `http://localhost:3000/books` You should be able to delete books.

9.7 PUT /books/:id

Edit is a bit more complicated, but nothing terrible. First we modify our book list to include an edit button.

views/books/index.html.erb

```
<h1>Books</h1>
<ul>
  <li ng-repeat="book in books">
    <div class="btn btn-default" ng-click="edit(book)">
      <i class="icon-pencil" />
    </div>
    <div class="btn btn-default" ng-click="delete(book)">
      <i class="icon-trash" />
    </div>
    {{ book.title }} by {{ book.author }}
  </li>
</ul>
```

Next we are going to add an edit method to our controller.

javascripts/angular/books_controller.js.coffee

```
  $scope.edit = (book) ->
    $scope.book =
      id: book.id
      title: book.title
      author: book.author
      url: book.url
```

We then need to update our save method to take into consideration whether the save is a create or an update.

javascripts/angular/books_controller.js.coffee

```
  $scope.save = () ->
    if $scope.book.id?
      $http({ method: "PUT", url: $scope.book.url, data: $scope.book })
        .success (response) ->
          $scope.book = {}
          $scope.getBooks()
    else
      $http({ method: "POST", url: $scope.urls.books, data: $scope.book })
        .success (response) ->
          $scope.book = {}
          $scope.getBooks()
```

We need to do some updates to our `BooksController` as well.

controllers/books_controller.rb

```
# PATCH/PUT /books/1
def update
  respond_to do |format|
    if @book.update(book_params)
      format.html { redirect_to @book, notice: 'Book was successfully updated.' }
      format.json { render json: @book }
    else
      format.html { render action: 'edit' }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
```

Try it out!

Reload to `http://localhost:3000/books` You should be able to edit books.

Conclusion

It was a bit of a journey but we guided you through the basics of CRUD operations between AngularJS and Rails. The final code looks like:

controllers/books_controller.rb

```
class BooksController < ApplicationController
  before_action :set_book, only: [:show, :edit, :update, :destroy]

  # GET /books
  def index
    @books = Book.all

    respond_to do |format|
      format.html {
        @books_json = @books.map{ |b| BookSerializer.new(b).serializable_hash }
        @urls = {
          books: books_path
        }
      }
      format.json { render json: @books }
    end
  end

  # GET /books/1
  def show
  end

  # GET /books/new
  def new
    @book = Book.new
  end

  # GET /books/1/edit
  def edit
  end

  # POST /books
  def create
    @book = Book.create(book_params)

    respond_to do |format|
```

```
37        if @book.save
38          format.html { redirect_to @book, notice: 'Book was successfully created.'\
39  }
40          format.json { render json: @book }
41        else
42          format.html { render action: 'new' }
43          format.json { render json: @book.errors, status: :unprocessable_entity }
44        end
45      end
46    end
47
48    # PATCH/PUT /books/1
49    def update
50      respond_to do |format|
51        if @book.update(book_params)
52          format.html { redirect_to @book, notice: 'Book was successfully updated.'\
53  }
54          format.json { render json: @book }
55        else
56          format.html { render action: 'edit' }
57          format.json { render json: @book.errors, status: :unprocessable_entity }
58        end
59      end
60    end
61
62    # DELETE /books/1
63    def destroy
64      @book.destroy
65      respond_to do |format|
66        format.html {
67          redirect_to books_url, notice: 'Book was deleted'
68        }
69        format.json { render json: { message: "Book was deleted" } }
70      end
71    end
72
73    private
74      # Use callbacks to share common setup or constraints between actions.
75      def set_book
76        @book = Book.find(params[:id])
77      end
78
```

```
79      # Only allow a trusted parameter "white list" through.
80      def book_params
81        params.require(:book).permit(:title, :author)
82      end
83  end
```

javascripts/angular/books_controller.js.coffee

```
1   AngulaRails.controller "BooksController", ($scope, $http) ->
2
3     $scope.getBooks = () ->
4       $http({ method: "GET", url: $scope.urls.books })
5         .success (response) ->
6           $scope.books = response.books
7
8     $scope.save = () ->
9       if $scope.book.id?
10        $http({ method: "PUT", url: $scope.book.url, data: $scope.book })
11          .success (response) ->
12            $scope.book = {}
13            $scope.getBooks()
14      else
15        $http({ method: "POST", url: $scope.urls.books, data: $scope.book })
16          .success (response) ->
17            $scope.book = {}
18            $scope.getBooks()
19
20    $scope.delete = (book) ->
21      $http({ method: "DELETE", url: book.url })
22        .success (response) ->
23          $scope.getBooks()
24
25    $scope.edit = (book) ->
26      $scope.book =
27        id: book.id
28        title: book.title
29        author: book.author
30        url: book.url
```

views/books/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="BooksController" ng-init="urls=<%= @urls\
.to_json %>; books=<%= @books_json.to_json %>">
  <div class="row">
    <div class="col-md-6">

      <form ng-submit="save()">
        <div class="form-group">
          <label>Title</label>
          <input type="text" class="form-control" ng-model="book.title" />
        </div>
        <div class="form-group">
          <label>Author</label>
          <input type="text" class="form-control" ng-model="book.author" />
        </div>
        <button class="btn btn-primary">
          Save
        </button>
      </form>

      <h1>Books</h1>
      <ul>
        <li ng-repeat="book in books">
          <div class="btn btn-default" ng-click="edit(book)">
            <i class="icon-pencil" />
          </div>
          <div class="btn btn-default" ng-click="delete(book)">
            <i class="icon-trash" />
          </div>
          {{ book.title }} by {{ book.author }}
        </li>
      </ul>
    </div>
  </div>
</div>
```

10 Services

In the previous chapter, we ended with basic CRUD in AngularJS with a Rails application. Lets not stop there. We can refactor our CRUD HTTP code into a serice that we can inject into our controller.

What we will cover: * Creating a custom service * Injecting the service into a controller * Using $q promises

Setup

Look to the previous chapter for setup. We are going to assume the ending point from the last chapter as the starting point for this chapter.

10.1 Create a custom service

We want to create a method to get all the books. To see that it works, we need to remove the passing of the `books` to the controller from Rails.

views/books/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="BooksController" ng-init="urls=<%= @urls\
.to_json %>; getBooks()">
```

Lets create our `Book` service for AngularJS.

javascripts/angular/book_service.js.coffee

```
AngulaRails.factory "Book", () ->
  self = {}
  self
```

Now we are going to add a method to our service to fetch a list of books and attach the books to the passed in `$scope`. Notice that we can pass in dependencies to be injected in the constructor just like we did with our controller. In this example, we need to use AngularJS's `$http` service again to make the GET request.

javascripts/angular/book_service.js.coffee

```
AngulaRails.factory "Book", ($http) ->
  self = {}

  self.getBooks = (scope) ->
    $http({ method: "GET", url: scope.urls.books })
      .success (response) ->
        scope.books = response.books

  self
```

And then we update out `BooksController` to use our new service. We first remove our old `getBooks` method. We then add an `init` method that we are calling in our review.

javascripts/angular/books_controler.js.coffee

```
AngulaRails.controller "BooksController", ($scope, Book, $http) ->

  $scope.getBooks = () ->
    Book.getBooks($scope)
```

Try it out

Go to `http://localhost:3000/books` We should still get our list of books we created in our previous example.

10.2 Using promises

While this works, we are passing `scope` into our service. This isn't bad per se, but might not be something we desire. Lets create an implementation that uses AngularJS promises.

AngularJS comes with a promise service called `$q`.

javascripts/angular/book_service.js.coffee

```
AngulaRails.factory "Book", ($http, $q) ->
  self = {}

  self.getBooksWithPromises = () ->
    deferred = $q.defer()
    $http({ method: "GET", url: "/books" })
      .success (response) ->
        deferred.resolve(response.books)

    deferred.promise

  self
```

We then need to change our `BookController` to use the promise method.

javascripts/angular/books_controler.js.coffee

```
$scope.getBooks = () ->
  Book.getBooksWithPromises().then (books) -> $scope.books = books
```

Try it out

Go to `http://localhost:3000/books` We should still get our list of books but now we are using a promise.

Conclusion

Now we know how to create a custom service that we can inject into controllers and other services. This is one tool we can use to encapsulate related code and reduce the amount of code in our controllers.

11 Using ng-resource with Rails

If you are dealing with a lot of CRUD operations and want a module to help with a lot of the boiler plate code, then AngularJS's `ng-resource` module could be just the solution.

In this chapter we will bring in the `ng-resource` module to perform some basic CRUD operations using a Rails api.

We are also going to be using `haml` as our view engine.

Why `haml`? We originally had the book in `haml` but were asked multiple times to show examples in `erb`. We consider this an optional chapter and we do a lot of our work in `haml` anyway. If the `haml` examples are difficult to follow, then you can also use a `haml` to `erb` converter. Like: http://haml2erb.herokuapp.com/

What we will cover:

- Adding Rails's token to prevent CSRF
- Setting default header options for `$http`
- Using AngularJS directives with haml
- using `ng-resource`

Setup

If you have the api endpoints from the previous chapters, then you can skip some of the setup and Rails code. We are going to include the code to be thorough.

terminal

```
$ rails g scaffold Book title author
$ rake db:migrate
```

This will create a `Book` model along with related CRUD controllers. The scaffolding will also add the JSON api endpoints to our Rails application. It will also create our `BookSerializer`.

app/serializers/book_serializer.rb

```
1 class BookSerializer < ActiveModel::Serializer
2   attributes :id, :title, :author
3 end
```

HTTP Header Defaults for Rails

For a more detailed description of the $http setup, please refer back to the Talking to Rails chapter.

We need to make sure we have our CSRF token and setup a basic header configuration for our $http service.

javascripts/angular/http.js.coffee

```
1 AngulaRails.config ["$httpProvider", ($httpProvider) ->
2   $httpProvider.defaults.headers.common['X-CSRF-Token'] = $('meta[name=csrf-token\
3 ]').attr('content')
4   $httpProvider.defaults.headers.common.Accept = "application/json"
5 ]
```

11.1 Adding ngResource

AngularJS used to have the ngResource module packaged into the main js file. Now the module is packaged separately. So first we need to download the angular-resource.js file and add it to our javascript folder. We then need to reference the file in our application.js.

javascripts/application.js

```
1 //= require angular
2 //= require angular-resource
```

Line 2

Add angular-resource javascript file to our application js manifest.

We then need to load the ngResource module in our AngularJS application.

javascripts/angular-application.js

```
1 //= require_self
2 //= require_tree ./angular
3
4 AngulaRails = angular.module("AngulaRails", ["ngResource"]);
```

Line 4

Load the ngResource module with our app.

11.2 GET /books

First we will setup our BooksController's index action.

controllers/books_controller.rb

```
1 # GET /books
2 def index
3   @books = Book.all
4
5   respond_to do |format|
6     format.html { }
7     format.json { render json: @books }
8   end
9 end
```

We delete the Rails scaffolding for index.html.erb and replace it with index.html.haml.

views/books/index.html.haml

```
1 .row{"ng-app" => "AngulaRails", "ng-controller" => "BooksController", "ng-init" =\
2 > "getBooks()"}
3   .col-md-12
4     %h3 Bookshelf
5     %ul
6       %li{"ng-repeat" => "book in books"}
7         {{ book.title }} by {{ book.author }}
```

Line 1,2 ng-init

With ng-init we can call our getBooks() method on our controller after the page is loaded.

We are going to use the same controller code from the Talking to Rails chapter.

javascripts/angular/books_controller.js.coffee

```
1 AngulaRails.controller "BooksController", ($scope, $http) ->
2   $scope.getBooks = () ->
3     $http({ method: "GET", url: "/books" })
4       .success (response) ->
5         $scope.books = response.books
```

Line 1 `$http`

We are going to use the `$http` service to talk to our Rails server.

Line 3 `/books`

We will call the books url that we know exists.

And let's add some books.

terminal

```
1 $ rails console
2 irb> Book.create(title: "The Hobbit", author: "J.R.R. Tolkien")
3 irb> Book.create(title: "Ender's Game", author: "Orson Scott Card")
```

Try it out!

Go to `http://localhost/books` You should see the list of the books you just added in the console.

11.3 Book Resource

We did nothing new in that last section. Now it is time to use the `ngResource` to replace our `GET` request. First we create our `Book` resource.

javascripts/angular/book.js.coffee

```
1 AngulaRails.factory "Book", ($resource) ->
2   $resource("/books/:id")
```

Line 1 `$resource`

By adding the `ngResource` module, we get access to the `$resource` service. Line 2 `/books/:id`

The first parameter is the url the resource will use to perform different CRUD operations. The following is the list of methods the `$resource` provides along with the HTTP verb that will be used for the requests:

```
{
  'get':    {method:'GET'},
  'save':   {method:'POST'},
  'query':  {method:'GET', isArray:true},
  'remove': {method:'DELETE'},
  'delete': {method:'DELETE'}
}
```

Now with our Book resource, we can replace our $http code with a simple query() call on our resource.

```
AngulaRails.controller "BooksController", ($scope, Book) ->

  $scope.getBooks = () ->
    $scope.books = Book.query()
```

Line 1 Book

We are going to list the Book resource we just specified as a dependency for our controller.
Line 4 Book.query()

This will make a GET request to /books and place the results in $scope.books.

If you try this out now, you will get an error. ngResource expects to be returned an array. We are returning json with a root object. We need to change our index action in our controller to not return a root object.

controllers/books_controller.rb

```
def index
  @books = Book.all

  respond_to do |format|
    format.html { }
    format.json { render json: @books, root: false, each_serializer: BookSerializ\
er }
  end
end
```

Try it out!

Go to http://localhost/books You should see the list of the books just like before.

Another way of configuring our resource and Rails

We could also have configured our resource to not expect an array. We would then have to adjust the response before assigning to books.

javascripts/angular/book.js.coffee

```
AngulaRails.factory "Book", ($resource) ->
  $resource("/books/:id", {}, {
    query: { method: "GET", isArray: false }
  })
```

```
AngulaRails.controller "BooksController", ($scope, Book) ->
  $scope.getBooks = () ->
    Book.query (response) ->
      $scope.books = response.books
```

11.4 Promises

But wait! Isn't that an ajax request? Where is my callback?

Angular resources uses promises. This means we can treat these async operations synchronously.

If we want to leverage the updated AngularJS promises more, we can use the newer promise interface.

```
AngulaRails.controller "BooksController", ($scope, Book) ->
  $scope.getBooks = () ->
    Book.query().$promise.then (books) ->
      $scope.books = books
```

11.5 Insert a Book

Let's first add the form to our view.

views/books/index.html.haml

```
1  .row{"ng-app" => "AngulaRails", "ng-controller" => "BooksController", "ng-init" =\
2  > "getBooks()"}
3    .col-md-12
4      %form{"ng-submit" => "save()"}
5        .form-group
6          %label Title
7          %input{type: "text", "ng-model" => "book.title", class: "form-control"}
8        .form-group
9          %label Author
10         %input{type: "text", "ng-model" => "book.author", class: "form-control"}
11       %button.btn.btn-primary
12         Save
13
14     %h3 Bookshelf
15     %ul
16       %li{"ng-repeat" => "book in books"}
17         {{ book.title }} by {{ book.author }}
```

Line 4 `ng-submit`

Submitting the form will call the `save()` method on our controller

Let's create our `save()` method.

javascripts/angular/books_controller.js.coffee

```
1  AngulaRails.controller "BooksController", ($scope, Book) ->
2
3    $scope.getBooks = () ->
4      $scope.books = Book.query()
5
6    $scope.save = () ->
7      Book.save($scope.book)
8      $scope.book = {}
9      $scope.getBooks()
```

Try it out!

Reload to `http://localhost:3000/books`. You should be able to add books using the form.

11.6 Edit a Book

So we can create books, but what about editing books? First, we will add an edit button to our book list.

views/books/index.html.haml

```
%h3 Bookshelf
%ul
  %li{"ng-repeat" => "book in books"}
    .btn.btn-default{"ng-click" => "edit(book)"}
      %i.icon-pencil
    {{ book.title }} by {{ book.author }}
```

Now we need to create our `edit()` action.

javascripts/angular/books_controller.js.coffee

```
AngulaRails.controller "BooksController", ($scope, Book) ->

  $scope.getBooks = () ->
    $scope.books = Book.query()

  $scope.edit = (book) ->
    $scope.book = Book.get({ id: book.id })

  $scope.save = () ->
    Book.save($scope.book)

    $scope.book = {}
    $scope.getBooks()
```

We need to update our `show` action on our controller to return the JSON in the format `ngResource` likes.

controllers/books_controller.rb

```
  def show
    respond_to do |format|
      format.html { }
      format.json { render json: @book, root: false }
    end
  end
```

Try it out!

Reload to `http://localhost:3000/books`. Notice the books don't get edited, but instead get added as a new book. WTH!

Well, by default AngularJS does a `POST` when updating. We need to change the resource to add an `update` action that will use a `PUT` request.

javascripts/angular/book.js.coffee

```
AngulaRails.factory "Book", ($resource) ->
  $resource("/books/:id", { id: "@id" }, {
      update: { method: 'PUT' }
    })
```

We then need to update our `save` method to use either the `save` or the `update` action on our resource.

javascripts/angular/books_controller.js.coffee

```
AngulaRails.controller "BooksController", ($scope, Book) ->

  $scope.getBooks = () ->
    $scope.books = Book.query()

  $scope.edit = (book) ->
    $scope.book = Book.get({ id: book.id })

  $scope.save = () ->
    if $scope.book.id?
      Book.update($scope.book)
    else
```

```
13        Book.save($scope.book)
14
15      $scope.book = {}
16      $scope.getBooks()
```

Try it out!

Reload to `http://localhost:3000/books`. You should be able to edit books now.

11.7 Delete a Book

We are going to modify our book list again, but this time we are going to add a delete button.

views/books/index.html.haml

```
1  %h3 Bookshelf
2  %ul
3    %li{"ng-repeat" => "book in books"}
4      .btn.btn-default{"ng-click" => "edit(book)"}
5        %i.icon-pencil
6      .btn.btn-default{"ng-click" => "delete(book)"}
7        %i.icon-trash
8      {{ book.title }} by {{ book.author }}
```

Next, we will add the `delete` method on our controller.

javascripts/angular/books_controller.js.coffee

```
1  AngulaRails.controller "BooksController", ($scope, Book) ->
2
3    $scope.getBooks = () ->
4      $scope.books = Book.query()
5
6    $scope.edit = (book) ->
7      $scope.book = Book.get({ id: book.id })
8
9    $scope.delete = (book) ->
10     book.$delete()
```

```
11      $scope.getBooks()
12
13    $scope.save = () ->
14      if $scope.book.id?
15        Book.update($scope.book)
16      else
17        Book.save($scope.book)
18
19      $scope.book = {}
20      $scope.getBooks()
```

Try it out!

You now have an AngularJS way to perform some fast CRUD!

Conclusion

We showed some simple CRUD examples of using the ngResource module with Rails.

12 Paging with AngularJS and Rails

We often need to return a lot of results but don't want to display them all on one page. On Rails we using paging gems like kaminari to provide easy paging through large result sets.

In this chapter we will combine our Rails paging with AngularJS.

What we will cover: * paging in Rails using kaminari * building paging ui using Twitter bootstrap * manually paging through results on AngularJS and Rails

Setup

Lets create our `Game` model and a bunch of Games for us to page through.

```
$ rails g scaffold Game name:string
$ rake db:migrate
$ rails runner -e "53.times { |i| Game.create(name: 'Game #{i}') }"
```

Our `Game` serializer should look like:

app/serializers/game_serializer.rb

```
class GameSerializer < ActiveModel::Serializer
  attributes :id, :name
end
```

Update the index action on the `GamesController`.

app/controllers/games_controller.rb

```
  def index
    @games = Game.all
    respond_to do |format|
      format.html { }
      format.json { render json: @games }
    end
  end
```

And update `index.html.erb`.

views/games/index.html.erb

```
<div ng-app="AngulaRails" ng-controller="GamesController" ng-init="init()">
  <h1>Games</h1>

  <%= link_to 'New Game', new_game_path, class: "btn btn-default" %>

  <table class="table">
    <thead>
      <tr>
        <th>Name</th>
      </tr>
    </thead>

    <tbody>
      <tr ng-repeat="game in games">
        <td>{{ game.name }}</td>
      </tr>
    </tbody>
  </table>
</div>
```

assets/javascript/angular/games_controller.js.coffee

```
AngulaRails.controller "GamesController", ($http, $scope) ->

  $scope.init = () ->
    $scope.getGames()

  $scope.getGames = () ->
    $http({ method: "GET", url: "/games" })
      .success (response) ->
        $scope.games = response.games
```

Try it out!

Go to `http://localhost:3000/games` We should be shown a large list of all the games we created.

12.1 Add Paging

First we need to add the `kaminari` gem to our `Gemfile`. Be sure to run `bundle`!

Gemfilelanguage=ruby

```ruby
  gem "kaminari"
```

Update games controller to page the results using Kaminari.

app/controllers/games_controller.rblanguage=ruby

```ruby
  def index
    @games = Game.page(params[:page]).per(10)
    respond_to do |format|
      format.html { }
      format.json { render json: @games }
    end
  end
```

Try it out!

Since we added paging, we should only see the first 10 games now.

12.2 Next and Previous Page

Now we'll add the prev / next paging functionality.

```
<div ng-app="AngulaRails" ng-controller="GamesController" ng-init="init()">
  <h1>Games</h1>

  <%= link_to 'New Game', new_game_path, class: "btn btn-default" %>

  <br />

  <ul class="pagination">
    <li><a ng-click="setPage(currentPage - 1)">&amp;laquo;</a></li>
```

```
10      <li><a ng-click="setPage(currentPage + 1)">&raquo;</a></li>
11    </ul>
12
13    <table class="table">
14      <thead>
15        <tr>
16          <th>Name</th>
17        </tr>
18      </thead>
19
20      <tbody>
21        <tr ng-repeat="game in games">
22          <td>{{ game.name }}</td>
23        </tr>
24      </tbody>
25    </table>
26  </div>
```

Update the AngularJS controller to send a param `page` for the current page with our GET request.

assets/javascript/angular/games_controller.js.coffee

```
1  AngulaRails.controller "GamesController", ($http, $scope) ->
2
3    $scope.init = () ->
4      $scope.currentPage = 1
5      $scope.getGames()
6
7    $scope.getGames = () ->
8      http =
9        method: "GET"
10       url: "/games"
11       params:
12         page: $scope.currentPage
13
14     $http(http)
15       .success (response) ->
16         $scope.games = response.games
17
18   $scope.setPage = (newPage) ->
19     newPage = 1 if newPage < 1
20
21     $scope.currentPage = newPage
22     $scope.getGames()
```

Try it out!

You should be able to page up and down through the results.

12.3 Navigate pages

Return paging data with games, using ActiveModelSerializer's cool `meta` ability, useful for just this purpose.

app/controllers/games_controller.rb

```
1  def index
2    @games = Game.page(params[:page]).per(10)
3    respond_to do |format|
4      format.html { }
5      format.json { render json: @games, meta: {
6        number_of_pages: @games.num_pages,
7        current_page: @games.current_page,
8        total_count: @games.total_count
9        }
10     }
11   end
12 end
```

Create pages from the metadata on responses.

assets/javascript/angular/games_controller.js.coffee

```
1  $scope.getGames = () ->
2    http =
3      method: "GET"
4      url: "/games"
5      params:
6        page: $scope.currentPage
7
8    $http(http)
9      .success (response) ->
```

```
10          $scope.games = response.games
11          $scope.paging = response.meta
12          $scope.createPages()
13
14    $scope.createPages = () ->
15      $scope.pages = [1..$scope.paging.number_of_pages]
```

Show we can show the pages in view. When you click on a page, it'll call `setPage` to change the page on the controller.

app/views/games/index.html.erb

```
1   <ul class="pagination">
2     <li><a ng-click="setPage(currentPage - 1)">&laquo;</a></li>
3     <li ng-repeat="page in pages" ng-class="{ active: currentPage == page }">
4       <a ng-click="setPage(page)">
5         {{ page }}
6       </a>
7     </li>
8     <li><a ng-click="setPage(currentPage + 1)">&raquo;</a></li>
9   </ul>
```

Try it out!

You should be able to click on the individual page numbers and get the results.

Conclusion

We covered using paging on Rails and AngularJS. In the next chapter, we take this to another level and showcase how we can reuse this code in an AngularJS directive.

13 Directives

When you first hear about AngularJS Directives, they appear magic in the 'omg what is going on here' way. As a newcomer to AngularJS, the idea of extending HTML with your own elements seems foreign, or a callback to IE6. Without mastering directives, you'll limit the potential of your AngularJS apps – not cool.

It doesn't have to be this way: you can create Directives to reuse code and make templates easier for you (and future-you) to understand. You'll think of Directives as creating a DSL in HTML, and become a Wizard of unimaginable power! Ok, maybe not, but you will write easier reading code, and code that's easier to reuse.

All you need is that one example that turns the lightbulb on, lets' create one below.

Setup

We are going to make a directive out of the paging code we worked on in the last chapter.

13.1 Making a directive

At first all we are going to do is replace our paging markup with a template that we bring in using a directive.

assets/javascripts/angular/paging.js.coffee

```
1  AngulaRails.directive "arPaging", () ->
2    templateUrl: "/assets/paging.html"
```

Now we just need to create our template.

assets/templates/paging.html

```
<ul class="pagination">
  <li><a ng-click="setPage(currentPage - 1)">&laquo;</a></li>
  <li ng-repeat="page in pages" ng-class="{ active: currentPage == page }">
    <a ng-click="setPage(page)">
      {{ page }}
    </a>
  </li>
  <li><a ng-click="setPage(currentPage + 1)">&raquo;</a></li>
</ul>
```

Now in our view we will replace our paging markup with this simple div with our directive.

app/views/games/index.html.erb

```
<div ar-paging></div>

<!-- <ul class="pagination">
  <li><a ng-click="setPage(currentPage - 1)">&laquo;</a></li>
  <li ng-repeat="page in pages" ng-class="{ active: currentPage == page }">
    <a ng-click="setPage(page)">
      {{ page }}
    </a>
  </li>
  <li><a ng-click="setPage(currentPage + 1)">&raquo;</a></li>
</ul> -->
```

Try it out!

Everything should work as before. We now have a simple directive to reuse the same html in different AngularJS views.

13.2 More than just templates

Time to upgrade our directive. Let's move some of the paging specific code into the directive to clean up the controller and maximize code reuse.

assets/javascripts/angular/paging.js.coffee

```
AngulaRails.directive "arPaging", () ->
  templateUrl: "/assets/paging.html"
  controller: ($scope) ->
    $scope.initPaging = (data) ->
      $scope.paging = data
      $scope.pages = [1..$scope.paging.number_of_pages]

    $scope.setPage = (newPage) ->
      newPage = 1 if newPage < 1

      $scope.currentPage = newPage
      $scope.getGames()
```

In our `GamesController` we can remove the paging code since it is now in our directive. Notice we can all the `initPaging` method on our directive inside our controller. This works because the scope of our directive is currently set to be the same as our controller. You can specify a directive to have its own independent scope but for this example we kept the scope the same.

assets/javascripts/angular/games_controller.js.coffee

```
AngulaRails.controller "GamesController", ($http, $scope) ->

  $scope.init = () ->
    $scope.currentPage = 1
    $scope.getGames()

  $scope.getGames = () ->
    http =
      method: "GET"
      url: "/games"
      params:
        page: $scope.currentPage

    $http(http)
      .success (response) ->
        $scope.games = response.games
        $scope.initPaging(response.meta)
```

Try it out!

Everything should work as before. This code keeps getting better and better!

13.3 Bind to our directive

Notice in our directive, we are still calling `getGames` on the controller. This will not be helpful when we want to reuse this directive on other pages. What would be nice is if we could tell the directive what method to call like how we use `ng-submit`. Since `ng-submit` is also a directive, we know we can achieve this.

In our directive we are now going to set values on our directive's scope from the attributes on our directive. We are going to read `pagingChange` from our directive binding on `ar-paging-change`.

assets/javascripts/angular/paging.js.coffee

```
1  AngulaRails.directive "arPaging", () ->
2    templateUrl: "/assets/paging.html"
3    scope:
4      currentPage: "=arCurrentPage"
5      paging: "=arPaging"
6      pagingChange: "=arPagingChange"
7    controller: ($scope) ->
8      $scope.$watch "paging", () ->
9        if $scope.paging?
10         $scope.pages = [1..$scope.paging.number_of_pages]
11
12     $scope.$watch "currentPage", () ->
13       $scope.pagingChange()
14
15     $scope.setPage = (newPage) ->
16       newPage = 1 if newPage < 1
17       $scope.currentPage = newPage
```

Now let's update our view to use the new attributes.

views/games/index.html.erb

```
<div ar-paging="paging" ar-current-page="currentPage" ar-paging-change="getGames"\
></div>
```

Try it out!

Victory! Everything should work as before. As a bonus, we have an awesome directive for paging we can reuse throughout our application.

Conclusion

We turned our paging code into a directive that we can now use throughout our application. Directives are a wonderful tool for packaging UI with common functionality. Although you don't have to reuse directives, the most common practice is to package common components into directives for reuse.

14 Image Avatar Directives

For our second example, let's create a `<multi-avatar>` HTML element that displays a user's image. This example will work with and without Rails. Here are some requirements:

This example will work with and without Rails.

1. If the user has a facebook-id, we want to use the Facebook image.
2. If the user has a github username, we want to use Github's shiny avatar service.
3. Otherwise, use Gravatar with the user's email.

So, our Interface in the HTML:

```
<multi-avatar data-facebook-id='' data-github-username='' data-email=''>
```

After AngularJS parses this Directive, the browser will see a simple:

```
<img src="https://example.com/555.img" alt='Avatar'/>
```

To create our Multi-Avatar:

1. Create a file named 'multi-avatar-directive.js'
2. Create a module to house our code. Since it's UI based, we use 'ui-multi-gravatar'.

```
;'use strict';
angular.module('ui-multi-gravatar', [])
```

3. Create the directive that returns a temporary `<h6>` holder

```
angular.module('ui-multi-gravatar', [])
  .directive('multiAvatar', [function () {
    return {
      restrict: 'E',
      link:function(scope, element, attrs) {
        var tag = '<h6>Temporary Holder</h6>';
        element.append(tag);
      }
    };
  }]);
```

4. To get the value of a data attribute, you use attrs. So to get `data-facebook-id`, you get attrs.facebookId (notice the camel case). Let's use that to use Facebook, then GitHub, then Gravatar:

```
angular.module('ui-multi-gravatar', [])
  .directive('multiAvatar', [function () {
    return {
      restrict: 'E',
      link:function(scope, element, attrs) {

        var facebookId = attrs.facebookId;
        var githubUsername = attrs.githubUsername;
        var email = attrs.email;

        var tag = '';
        if ((facebookId !== null) && (facebookId !== undefined) && (facebookId !=\
= '')) {
          tag = '<h6>Use Facebook</h6>' ;
        } else if ((githubUsername !== null) && (githubUsername !== undefined) &&\
 (githubUsername !== '')){
          tag = '<h6>Use github</h6>' ;
        } else {
          tag = '<h6>Use gravatar</h6>';
        }

        element.append(tag);
      }
    };
  }]);
```

1. Let's add some HTML to use:

```
    <div ng-app="YourApp">
       <div class="container" ng-controller='UsersController'>
         <div class="row">
           <div class="col-lg-4">
             <h2>With Facebook ID</h2>
             <multi-avatar data-facebook-id="717976707"/>
           </div>
           <div class="col-lg-4">
             <h2>With GitHub Username</h2>
                <multi-avatar data-github-username="jwo"/>
           </div>
           <div class="col-lg-4">
             <h2>With Email</h2>
             <multi-avatar data-email="jesse@rubyoffrails.com"/>
           </div>
           <div class="col-lg-4">
             <h2>Blank</h2>
             <multi-avatar data-facebook-id='' data-github-username='' data-email\
='' >
           </div>
         </div>
     </div>
```

Let's add the image creation logic:

```
        if ((facebookId !== null) && (facebookId !== undefined) && (facebookId !=\
= '')) {
          tag = '<img src="http://graph.facebook.com/' + facebookId + '/picture?t\
ype=large" class="img-responsive"/>'
        } else if ((githubUsername !== null) && (githubUsername !== undefined) &&\
 (githubUsername !== '')){
          tag = '<img src="https://identicons.github.com/' + githubUsername + '.p\
ng" class="img-responsive"/>';
        } else {
          tag = '<h6>Use gravatar</h6>';
        }
```

Fun! With the facebook and GitHub working, leaving only gravatar to implement:

We need to generate an MD5 hash of the email to send to Gravatar, so we'll use the well-worn MD5 library, ported to an AngularJS service by Jim Lavin[1].

[1] https://github.com/lavinjj/angularjs-gravatardirective

1. Add `<script src="../src/md5.js"></script>` to your HTML
2. Add the module to our angular app

```
angular.module('YourApp', ['ui-multi-gravatar', 'md5']);
```

1. Update our module to include the md5 dependency:

```
angular.module('ui-multi-gravatar', [])
  .directive('multiAvatar', ['md5', function (md5) {
    return {
      restrict: 'E',
      link:function(scope, element, attrs) {
```

1. Update our last else to use gravatars:

```
        } else {
          var hash = ""
          if ((email !== null) && (email !== undefined) && (email !== '')){
            var hash = md5.createHash(email.toLowerCase());
          }
          var src = 'https://secure.gravatar.com/avatar/' + hash + '?s=200&d=mm'
          tag = '<img src=' + src + ' class="img-responsive"/>'
        }
```

To test out the functionality that of Gravatar for a default image:

```
<h2>Blank</h2>
<multi-avatar/>
```

or

```
<h2>Blank</h2>
<multi-avatar data-facebook-id='' data-github-username='' data-email=''>
```

14.1 Full AngularJS Code

```
;'use strict';

angular.module('ui-multi-gravatar', [])
  .directive('multiAvatar', ['md5', function (md5) {
    return {
      restrict: 'E',
      link:function(scope, element, attrs) {

        var facebookId = attrs.facebookId;
        var githubUsername = attrs.githubUsername;
        var email = attrs.email;

        var tag = '';
        if ((facebookId !== null) && (facebookId !== undefined) && (facebookId !=\
= '')) {
          tag = '<img src="http://graph.facebook.com/' + facebookId + '/picture?w\
idth=200&height=200" class="img-responsive"/>'
        } else if ((githubUsername !== null) && (githubUsername !== undefined) &&\
 (githubUsername !== '')){
          tag = '<img src="https://identicons.github.com/' + githubUsername + '.p\
ng" style="width:200px; height:200px" class="img-responsive"/>';
        } else {
          var hash = ""
          if ((email !== null) && (email !== undefined) && (email !== '')){
            var hash = md5.createHash(email.toLowerCase());
          }
          var src = 'https://secure.gravatar.com/avatar/' + hash + '?s=200&d=mm'
          tag = '<img src=' + src + ' class="img-responsive"/>'
        }

        element.append(tag);
      }
    };
  }]);
```

Try it out

You can add avatars with GitHub, Facebook, or a Gravatar. Yay!

15 Routing

In this chapter we are going to explore using the `ngRoute` module with AngularJS. This used to be packaged with the main AngularJS file but is now a separate module you can download.

In the following chapter we will take a look at the increasingly popular ui-router module. We prefer the ui-router because of the awesomeness of state based routing, but since it is still beta we wanted to share both options.

What will be cover:

- Setup `ngRoute` module
- How to add a module to an AngularJS application
- `resolve`
- `$q` promises to resolve route dependencies

15.1 Setup ngRoute

You can download the `angular-route.min.js` files from `http://www.angularails.com/files`.

Add the file to the `javascripts` folder and include it in our `application.js`.

javascripts/application.js

```
1 //= require angular-route.min
```

We then need to include the module in our AngularJS application. The second parameter of the `module` method is an array.

Previously we were passing an empty array signifying that we were not bringing in any extra modules, but now we want to bring in the ngRoute module so we just add the string `"ngRoute"` as an element into the array

javascripts/angular-application.js

```
1 AngulaRails = angular.module("AngulaRails", ["ngRoute"]);
```

Setup

This will create a `Routing` controller with a `show.html.erb` view.

terminal

```
$ rails g controller Routing show
```

Replace the generated contents of `show.html.erb` with:

views/routing/show.html.erb

```
<div class="row" ng-app="AngulaRails">
  <div class="col-md-12">
    <h1>Routing Example</h1>
    <div ng-view></div>
  </div>
</div>
```

Line 3 `ng-view`

By setting the `ng-view` directive on an HTML element, we are specifying where we want our templates to be displayed. We will be using templates with our routes in order to change the views based on the route.

We are going to use a crew service to provide the data for this exercise. No need to complicate things with actual API calls.

javascripts/angular/crew.js.coffee

```
AngulaRails.factory "Crew", () ->
  self = {}
  self.crew = [
    id: 1
    name: "Kirk"
  ,
    id: 2
    name: "Spock"
  ,
    id: 3
    name: "McCoy"
  ]

  self.all = () ->
    self.crew

  self.find = (id) ->
```

```
18      for c in self.crew
19        if c.id == parseInt(id)
20          return c
21      null
22
23    self.getPosition = (id) ->
24      switch id
25        when "1"
26          "Captain"
27        when "2"
28          "Science Officer"
29        when "3"
30          "Doctor"
31
32    self
```

15.2 Root Route

We are going to create a root route that will display a list of crew members.

javascripts/angular/routes.js.coffee

```
1  AngulaRails.config ($routeProvider) ->
2    $routeProvider
3      .when "/",
4        templateUrl: "/assets/crew/index.html"
5        controller: "CrewMembersController"
```

Line 1 `$routeProvider`

Inside the `app` config function, we can resolve providers such as the `$routeProvider`. We use this provider to create the routes for our application.

Line 3 `when`

We call the `when` function on `$routeProvider` to create routes. The first parameter is the route. The second parameter sets up how to handle that route.

Line 4 `templateUrl`

This url is used to find the template to use for the controller. In our case, we are storing templates in the `assets/templates` folder.

Line 5 `controller`

The string provided is used to resolve the controller to use for this route.

For our AngularJS controller, we are just going to set `$scope.crew` to the array of crew we get back from our injected `Crew` service.

javascripts/angular/crew_members_controller.js.coffee

```
AngulaRails.controller "CrewMembersController" , ($scope, Crew) ->
  $scope.crew = Crew.all()
```

Create a `templates` folder under the `assets` folder. Then create `assets/templates/crew/index.html` with the following contents:

templates/crew/index.html

```
  <h3>Enterprise Crew</h3>
  <ul ng-repeat="c in crew">
    <li>{{ c.name }}</li>
  </ul>
```

Try it out!

Go to `http://localhost:3000/routing/show`. You should see the list of crew members.

If you get a 404 when trying to find your template. Stop and restart your Rails server. We added that templates folder and Rails will need a restart to know that the folder is there.

15.3 Default Routes

For any route not specified, we can redirect the user to our root route.

javascripts/angular/routes.js.coffee

```
AngulaRails.config ($routeProvider) ->
  $routeProvider
    .when "/",
      templateUrl: "/assets/crew/index.html"
      controller: "CrewMembersController"
    .otherwise redirectTo: "/"
```

Try it out!

Go to `http://localhost:3000/routing/show`. Try different routes like: http://localhost:3000/routing/show/#/foobar

15.4 Detail Page

Let's create a route and page to display a single crew member. To know what crew member we want to show, our route will need the `id` for the crew member.

javascripts/angularails/routes.js.coffee

```
1  AngulaRails.config ($routeProvider) ->
2    $routeProvider
3      .when "/",
4        templateUrl: "/assets/crew/index.html"
5        controller: "CrewMembersController"
6      .when "/crew/:id",
7        templateUrl: "/assets/crew/show.html"
8        controller: "CrewMemberController"
9      .otherwise redirectTo: "/"
```

Line 6 `:id`
: We can add parameters in our route using `:` then the name of the parameter.

Line 7
: We will use a different template for the crew show route.

Line 8:
: We will use a different controller for the crew show route.

Now time to create our `CrewMemberController`.

javascripts/angular/crew_member_controller.js.coffee

```
1  AngulaRails.controller "CrewMemberController" , ($scope, Crew, $routeParams) ->
2    $scope.crew = Crew.find($routeParams.id)
```

Line 1 `$routeParams`
: AngularJS will pass in the route parameters in a JSON object called `$routeParams`.

Line 2

We will use the `id` parameter from the routes and pass it into our `Crew` service to get the crew member data.

For our template, we will display the name of the crew member and a link back to the root.

assets/templates/crew/show.html

```
1 <h2>{{ crew.name }}</h2>
2
3 <a ng-href="#/">Back</a>
```

Line 3 `ng-href`

Using `ng-href`, we will navigate back to our root route when the link is clicked.

Finally we will update our `index.html` template to add links to the crew member's detail page.

assets/templates/crew/index.html

```
1 <h3>Enterprise Crew</h3>
2
3 <ul ng-repeat="c in crew">
4   <li><a ng-href="#/crew/{{ c.id }}">{{ c.name }}</a></li>
5 </ul>
```

Try it out!

Go to `http://localhost:3000/routing/show`. We can now navigate to a crew member's page and then back to our crew member list page.

15.5 Routing with Promises (Resolving Later)

Routing in AngularJS lets us resolve dependencies for any route. Often times this involves making http requests to retrieve data the controller needs. For this scenario, we are just going to use a timeout to simulate a http request.

javascripts/angular/routes.js.coffee

```
1  AngulaRails.config ($routeProvider) ->
2    $routeProvider
3      .when "/",
4        templateUrl: "/assets/crew/index.html"
5        controller: "CrewMembersController"
6      .when "/crew/:id",
7        templateUrl: "/assets/crew/show.html"
8        controller: "CrewMemberController"
9        resolve:
10         position: ($q, $route, $timeout, Crew) ->
11           id = $route.current.params.id
12           defer = $q.defer()
13           $timeout () ->
14             position = Crew.getPosition(id)
15             defer.resolve(position)
16           , 1000
17
18           defer.promise
19
20     .otherwise redirectTo: "/"
```

Line 9 `resolve`

With the resolve option, we can provide a collection of properties that will be made available to the controller.

Line 10 `position`

By using a function, we can pass in parameters that Angular will resolve for us. These are your normal collection of providers and services. Notice we can even pass in our own `Crew` service.

Line 11 `$route`

We can pull the current route parameters off of `$route.current.params`.

Line 12 `$q`

`$q` is a promise provider for AngularJS. In our scenario we will use this to simulate an http request.

Line 13: `$timeout`

`$timeout` is a replacement for `setTimeout`. If you used `setTimeout`, you would need to call `$scope.$apply()` to update the bindings.

Line 15: `resolve`

Inside our timeout callback, we will resolve the promise and return the result from our `Crew.getPosition` call.

Line 18: `promise`

We return the promise. We resolve the promise in our `$timeout` function.

Use resolved variable in controller

javascripts/angular/crew_member_controller.js.coffee

```
1 AngulaRails.controller "CrewMemberController" , ($scope, Crew, $route, $routePara\
2 ms) ->
3   $scope.crew = Crew.find($routeParams.id)
4   $scope.position = $route.current.locals.position
```

Line 1: `$route`

`$route` is the same `$routeProvider` we use in our app.config function. In this context we are using the provider to gain access to the properties we resolved.

Line 3: `$route.current.locals`

This is how we can access the resolved properties for the current route.

assets/templates/crew/show.html

```
1 <h2>{{ crew.name }}</h2>
2 <h4>{{ position }}</h4>
3
4 <a ng-href="#/">Back</a>
```

Try it out!

Go to `http://localhost:3000/secure/show` When we navigate to a crew member page, we should see a 1 sec delay as we resolve our dependencies.

Refactor Promise Logic to Controller

Currently our routes configuration look like:

javascripts/angular/routes.js.coffee

```
AngulaRails.config ($routeProvider) ->
  $routeProvider
    .when "/",
      templateUrl: "../../assets/crew/index.html"
      controller: "CrewMembersController"
    .when "/crew/:id",
      templateUrl: "../../assets/crew/show.html"
      controller: "CrewMemberController"
      resolve:
        position: ($q, $route, $timeout, Crew) ->
          id = $route.current.params.id
          defer = $q.defer()
          $timeout () ->
            position = Crew.getPosition(id)
            defer.resolve(position)
          , 1000

          defer.promise

    .otherwise redirectTo: "/"
```

As you can see, it takes up quite a bit of code real estate to setup even a simple route with properties to resolve. This will become problematic when we start adding more routes, plus our configuration has to know a lot about how to resolve this controller.

A common pattern to fix this issue is to add methods on the controller object. These methods will be in the same file as the controller and therefore the intimacy is not as jarring.

javascripts/angular/crew_member_controller.js.coffee

```
AngulaRails.CrewMemberController = ($scope, Crew, $route, $routeParams) ->
  $scope.crew = Crew.find($routeParams.id)
  $scope.position = $route.current.locals.position

AngulaRails.CrewMemberController.templateUrl = "/assets/crew/show.html"
AngulaRails.CrewMemberController.resolve =
  position: ($q, $route, $timeout, Crew) ->
    id = $route.current.params.id
    defer = $q.defer()
    $timeout () ->
      position = Crew.getPosition(id)
```

```
12        defer.resolve(position)
13      , 1000
14
15      defer.promise
16
17  AngulaRails.controller "CrewMemberController" , AngulaRails.CrewMemberController
```

Line 1
Instead of registering the controller constructor function directly, we will set the function to a property off our application object.

Line 5 `templateUrl`
Now we can specify the template in the same file as the controller.

Line 6 `resolve`
Now we can specify the properties to be resolved in the same file as the controller.

Line 17
We then register the controller with our application.

Now we can clean up our route configuration.

javascripts/angular/routes.js.coffee

```
1  AngulaRails.config ($routeProvider) ->
2    $routeProvider
3      .when "/",
4        templateUrl: "/assets/crew/index.html"
5        controller: "CrewMembersController"
6      .when "/crew/:id",
7        templateUrl: AngulaRails.CrewMemberController.templateUrl
8        controller: "CrewMemberController"
9        resolve: AngulaRails.CrewMemberController.resolve
10
11     .otherwise redirectTo: "/"
```

Line 7 `templateUrl`‘
Use the `templateUrl` property on our controller.

Line 9 `resolve`‘
Use the `resolve` property on our controller.

Try it out!

Go to `http://localhost:3000/secure/show` Everything should still be working.

Conclusion

In this chapter we showed how to use the previously default AngularJS routing provider. In the next chapter we will show how to use the increasingly popular `ui-router`.

16 UI Router

Using the new `ui router` is super awesome. It is a state based routing engine that makes having subviews an easy task.

In this chapter we are going to go follow our previous routing example but change it up to showcase the stateful nature of the ui-router.

Although we are using the same example as the last chapter, we are going to start from scratch (otherwise we would have a lot of remove these files steps).

What we will cover: * installing `ui-router` * creating routes/states with `ui-router` * creating a sub-state

16.1 Install ui-router

First download the ui-router. You can download the files at `https://github.com/angular-ui/ui-router` or from `http://www.angularails.com/files`.

Add the `angular-ui-router.js` file to the `javascripts` folder and then add it to the `application.js`.

javascripts/application.js

```
1  //= require angular-ui-router
```

We next need to add the `ui-router` to our AngularJS application just like we did with the `ngRoute` module.

javascripts/angular-application.js

```
1  AngulaRails = angular.module("AngulaRails", ["ui.router"]);
```

Setup

Create our Rails controller and view.

terminal

```
$ rails g controller Routing show
```

We are going to use the same Crew Service as last time.

javascripts/angular/crew.js.coffee

```
AngulaRails.factory "Crew", () ->
  self = {}
  self.crew = [
    id: 1
    name: "Kirk"
  ,
    id: 2
    name: "Spock"
  ,
    id: 3
    name: "McCoy"
  ]

  self.all = () ->
    self.crew

  self.find = (id) ->
    for c in self.crew
      if c.id == parseInt(id)
        return c
    null

  self.getPosition = (id) ->
    switch id
      when "1"
        "Captain"
      when "2"
        "Science Officer"
      when "3"
        "Doctor"

  self
```

16.2 Setup Root Route

To setup the root route we need to configure the $stateProvider.

javascripts/angular/routes.js.coffee

```
AngulaRails.config ($stateProvider, $urlRouterProvider) ->
  $urlRouterProvider.otherwise("/")

  $stateProvider
    .state('app',
      url: "/"
      templateUrl: "/assets/crew/index.html"
      controller: "CrewMembersController"
    )
```

This should look pretty familiar. There is a lot of similarities with how we setup the routes using ngRoute. One of the key differences is that we give a route / state a name. In this case we are naming our root route app.

To see this in action we will need to build our controller and the view.

javascripts/crew_members_controller.js.coffee

```
AngulaRails.controller "CrewMembersController" , ($scope, Crew) ->

  $scope.crew = Crew.all()
```

assets/templates/crew/index.html

```
<h3>Enterprise Crew</h3>

<div class="row">
  <div class="col-md-4">
    <div class="list-group">
      <a class="list-group-item"
        ng-repeat="c in crew">
        {{ c.name }}
      </a>
    </div>
  </div>
</div>
```

Try it out!

Go to `http://localhost:3000/routing/show` We should be presented with the list of crew members.

16.3 Crew Route

Now lets create a child route to display a crew member. Notice that our state name includes the parent state. So instead of just `crew` the state name is `app.crew`.

javascripts/angular/routes.js.coffee

```
AngulaRails.config ($stateProvider, $urlRouterProvider) ->
  $urlRouterProvider.otherwise("/")

  $stateProvider
    .state('app',
      url: "/"
      templateUrl: "/assets/crew/index.html"
      controller: "CrewMembersController"
    )
    .state('app.crew',
      url: "crew/:id"
      templateUrl: "/assets/crew/show.html"
      controller: "CrewMemberController"
    )
```

Let's create our controller and view.

javascripts/crew_member_controller.js.coffee

```
AngulaRails.controller "CrewMemberController" , ($scope, Crew, $stateParams) ->
  $scope.crew = Crew.find($stateParams.id)
```

assets/templates/crew/show.html

```
<h2>{{ crew.name }}</h2>
```

Now for an interesting twist. Instead of the crew page being a complete separate page, we are going to make it a subsection on the root crews index page.

By adding a `ui-view` directive, we are specifying where we want child views of this view to be displayed.

assets/templates/crew/index.html

```
<h3>Enterprise Crew</h3>

<div class="row">
  <div class="col-md-4">
    <div class="list-group">
      <a class="list-group-item"
        ng-repeat="c in crew"
        ui-sref-active="active"
        ui-sref="app.crew({ id: c.id })">{{ c.name }}
      </a>
    </div>
  </div>
  <div class="col-md-4">
    <div ui-view></div>
  </div>
</div>
```

Try it out!

Go to `http://localhost:3000/routing/show` We should now be able to clickon the crew names and the child crew view should be displayed to the right of the crew list.

16.4 Promises

Just like 'ngRoute', the 'ui-router' module allows for resolving dependencies on different states.

javascripts/angular/routes.js.coffee

```
AngulaRails.config ($stateProvider, $urlRouterProvider) ->
  $urlRouterProvider.otherwise("/")

  $stateProvider
    .state('app',
      url: "/"
      templateUrl: "/assets/crew/index.html"
      controller: "CrewMembersController"
    )
    .state('app.crew',
      url: "crew/:id"
      templateUrl: "/assets/crew/show.html"
      controller: "CrewMemberController"
      resolve:
        position: ($q, $stateParams, $timeout, Crew) ->
          defer = $q.defer()
          $timeout () ->
            position = Crew.getPosition($stateParams.id)
            defer.resolve(position)
          , 1000

          defer.promise
    )
```

In our `CrewMemberController` we can then access the resolved `position` dependency.

javascripts/angular/crew_member_controller.js.coffee

```
AngulaRails.controller "CrewMemberController" , ($scope, Crew, position, $statePa\
rams) ->
  $scope.crew = Crew.find($stateParams.id)
  $scope.position = position
```

We will now display the position in our view.

assets/templates/crew/show.html

```
1 <h2>{{ crew.name }}</h2>
2 <h4>{{ position }}</h4>
```

Try it out!

Go to `http://localhost:3000/routing/show` We should now be able to clickon the crew names and the child crew view should be displayed to the right of the crew list.

Conclusion

We did a quick tour of the `ui-router` module. The `ui-router` provides an easy to use state based routing system for AngularJS applications. Although still beta, this is a go-to module for AngularJS application developers.

17 Devise Authentication

In this chapter we will be using `devise` for Rails authentication. We will show how we can show or hide content based on whether a user has been authenticated or not.

Setup

Add `devise` to your Gemfile.

Gemfile

```
1  gem 'devise', '3.0.0'
```

terminal

```
1  $ bundle
```

Now we will setup our user.

terminal

```
1  > rails g devise:install
2  > rails g devise user
3  > rake db:migrate
```

Make sure you stop and start your Rails app since we've added Gems and added configurations to the `config/initializers` folder.

Create a user we can use to login. The following is all one line.

terminal

```
1  > rails runner "User.create! email:
2  'jwo@example.com', password: '12345678', password_confirmation: '12345678'"
```

Line 1

The runner will execute code as if you were in rails console, but quit right after. It's ideal for cases just like this.

Time to generate our Rails controller and view.

terminal

```
> rails g controller secure show
```

views/secure/show.html.erb

```
<div class="row">
  <div class="span12">
    <h1>My Super Secure Stuff</h1>
    <dl>
      <dt>My Luggage (and planetary shield's Password)</dt>
      <dd>1-2-3-4-5</dd>
    </dl>
  </div>
</div>
```

Why 12345?
Because SPACEBALLS!, obvs.

Try it out!

Go to `http://localhost:3000/secure/show` You should see the luggage password

17.1 Secure via AngularJS

views/secure/show.html.erb

```
<div class="row" ng-app="AngulaRails" ng-controller="SessionsController"></div>
<div class="secure" ng-show="authorized()">
  <dl>
    <dt>My Luggage (and planetary shield's Password</dt>
    <dd>1-2-3-4-5</dd>
  </dl>
</div>
```

Line 1: `ng-app`and `ng-controller`
Add the ng-app and ng-controller as we've done before

Line 2: `.secure{"ng-show"=>"authorized()"}`

Add a new html node that will only be shown if the Controller.authorized method returns true. Move the %dl over to be a child of .secure.

app/assets/javascripts/angular/sessions_controller.js.coffee

```
1  AngulaRails.controller "SessionsController", ($scope) ->
2    $scope.authorized = () ->
3      false
```

We can show some other text to let the user know they are not authorized.

views/secure/show.html.erb

```
1  <div class="login" ng-show="!authorized()">
2    <div class="alert warning">
3      Please Login to access my secrets
4    </div>
5  </div>
6  <div class="secure" ng-show="authorized()">
7    <dl>
8      <dt>My Luggage (and planetary shield's Password)</dt>
9      <dd>1-2-3-4-5</dd>
10   </dl>
11 </div>
```

Line 1: `ng-show` users `!authorized()`

Since this is an expression, we can use `!` to say not-authorized.

Try it out!

Go to `http://localhost:3000/secure/show` Now you shouldn't see the password (since you're not logged in). You should see "Please Login"

17.2 Login with Rails

Now we'll add normal Rails login (outside of AngularJS). This is sometimes the easiest way to integrate with a JavaScript framework, but we'll see that the experience can be improved in the next step.

app/views/secure/show.html.erb

```
<div class="login" ng-show="!authorized()">
  <div class="alert warning">
    Please Login to access my secrets
  </div>
  <%= form_for(User.new, :as => "user", :url => session_path("user")) do |f| %>
    <div class="control-group">
      <%= f.label :email %>
      <div class="controls">
        <%= f.email_field :email %>
      </div>
    </div>
    <div class="control-group">
      <%= f.label :password %>
      <div class="controls">
        <%= f.password_field :password %>
      </div>
    </div>
    <%= f.submit "Sign In", class: "btn btn-large btn-primary" %>
  <% end %>
</div>
```

Line 4-12: `form_for`

This is the standard Rails Devise login form, using standard Rails forms. You can generate devise views by running `rails g devise:views`, and then copy where you want to use them.

After someone is logged in, we need to tell Angular about the current_user.

app/views/layout/application.html.erb

```
<%= csrf_meta_tags %>
<meta content="<%= current_user.to_json %>" name="<%= :current_user %>"/>
```

Line 2: `meta` We serialize the `current_user` that Rails knows about to JSON, and store it in a meta tag for Angular to parse and use.

app/assets/javascript/angular/sessions_controller.js.coffee

```
1  AngulaRails.controller "SessionsController", ($scope) ->
2    $scope.current_user = JSON.parse($("meta[name='current_user']").attr('content'))
3    $scope.authorized = () ->
4      $scope.current_user? and $scope.current_user.id?
```

Line 2

We parse the meta tag with current_user to JSON, and then store it in the scope as 'current_user'.

Line 4

We replace the "false" and have it check that current_user is set, and that it has an id.

Try it out!

Go to `http://localhost:3000/secure/show` Now you shouldn't see the password (since you're not logged in). You should see "Please Login" Login with email:`jwo@example.com` & password:`12345678` Return to `http://localhost:3000/secure/show`. You should see that you're logged in (and see the secured content).

17.3 Logout with Rails

Let's add a way to log out.

app/views/secure/show.html.erb

```
1  <div class="secure" ng-show="authorized()">
2    <h1>My Super Secure Stuff</h1>
3    <div class="pull-right">
4      <%= link_to "Sign Out", destroy_user_session_path, class: "btn", method: :del\
5  ete %>
6    </div>
7  </div>
```

Line 3,4

This is the standard Rails Devise sign out link.

If you click “Sign Out” while you’re logged in, and you return to ‘/secure/show’, you’ll see the login form.

This is because Rails has logged you out.

Try it out!

Go to `http://localhost:3000/secure/show`

17.4 Logout via Angular

Let’s start swapping Rails with AngularJS by changing how we sign out.

app/views/secure/show.html.erb

```
1 <div class="secure" ng-show="authorized()">
2   <div class="pull-right">
3     <button class="btn" ng-click="signOut()">Sign Out</button>
4   </div>
5   <dl>
6     <dt>My Luggage (and planetary shield's Password)</dt>
7     <dd>1-2-3-4-5</dd>
8   </dl>
9 </div>
```

Line 3

We remove the Rails sign out link and replace with a button that wil call `signOut` in our Angular controller.

app/assets/javascript/angular/sessions_controller.js.coffee

```
1 AngulaRails.controller "SessionsController", ($scope, $http) ->
2   $scope.current_user = JSON.parse($("meta[name='current_user']").attr('content'))
3   $scope.signOut = () ->
4     $http(
5       method: "DELETE"
6       url: "/users/sign_out"
7       headers: {'X-CSRF-Token': $('meta[name=csrf-token]').attr('content')}
8     )
```

```
9       .success (data) ->
10        $scope.current_user = null
11      .error (data, status) ->
12        alert("Error: #{status}.\n#{data}")
```

Line 1: `$http`
Since we'll be making an http call to Rails, we need Angular to give us the $http module

Line 3-11:
We implement the signOut method. It posts to Rails's Devise routes, and sets the current_user in Angular scope to null if there are no errors

Line 7: `headers: {'X-CSRF-Token': $('meta[name=csrf-token]').attr('content')}`
For Rails to accept POST, PUT, and DELETE http methods, it needs a valid CSRF token (for security). We grab it from Rails's meta tags and append to our headers.

Try it out!

Go to `http://localhost:3000/secure/show` Login with email:`jwo@example.com` & password:`12345678`. Click Sign Out and you should see the page instantly show the login form without refreshing.

17.5 Login via Angular

Logging in via Rails is barely acceptable – it's pretty slow, and by default we ended up at the Rails root page. Instead, let's login via Angular

We'll now change the form from a Rails form to a Angular form

app/views/secure/show.html.erb

```
1  <dl>
2    <dt>My Luggage (and planetary shield's Password</dt>
3    <dd>1-2-3-4-5</dd>
4  </dl>
5  <form ng-submit="signIn()">
6    <div class="alert alert-error" id="errors" ng-show="authErrors">
7      {{ authErrors }}
8    </div>
```

```
9     <div class="control-group">
10      <%= label_tag :email %>
11      <div class="controls">
12        <%= email_field_tag :email, '', autofocus: true, "ng-model"=>"email" %>
13      </div>
14    </div>
15    <div class="control-group">
16      <%= label_tag :password %>
17      <div class="controls">
18        <%= password_field_tag :password, '', "ng-model"=>"password" %>
19      </div>
20    </div>
21    <%= submit_tag "Sign In", class: "btn btn-large btn-primary" %>
22  </form>
```

Line 4-17

We replace the Rails form with a plain old HTML form, and attach `ng-submit` to call the `signIn()` method on AngularJS controller.

Line 5-6

If there are any errors, we show them with `{{authErrors}}`. And, we only show the alert box if `authErrors` is not empty.

And add the `signIn()` and `authErrors` to our AngularJS controller.

app/assets/javascript/angular/sessions_controller.js.coffee

```
1   AngulaRails.controller "SessionsController", ($scope, $http) ->
2     $scope.current_user = JSON.parse($("meta[name='current_user']").attr('content'))
3     $scope.authErrors = []
4
5     $scope.authorized = () ->
6       $scope.current_user? and $scope.current_user.id?
7
8     $scope.signIn = () ->
9       $scope.authErrors = []
10      authData = {user: {email: $scope.email, password: $scope.password}}
11
12      $http(
13        method: "POST"
14        url: "/users/sign_in"
15        headers: {'X-CSRF-Token': $('meta[name=csrf-token]').attr('content')}
16        data: authData
```

```
17      )
18        .success (data) ->
19          $scope.current_user = data
20        .error (data, status) ->
21          $scope.authErrors = data['error']
22
23    $scope.signOut = () ->
24      $http(
25        method: "DELETE"
26        url: "/users/sign_out"
27        headers: {'X-CSRF-Token': $('meta[name=csrf-token]').attr('content')}
28      )
29        .success (data) ->
30          $scope.current_user = null
31        .error (data, status) ->
32          alert("Error: #{status}.\n#{data}")
```

Line 8-20
: Implement `signIn()`. We grab the email and password that were binding to the model, and append the CSRF using the same technique as the signOut()

Line 3
: We initialize `authErrors` to be an empty array, and set it to have the errors provided by Rails in line 20

Now, when Rails receives a successful authentication, it will send back a redirect (302) to the proper location. But, we want to receive back the user as JSON so we don't have to refresh the entire page.

Time to create a new sessions controller for devise to use.

app/controllers/sessions_controller.rb

```
1  class SessionsController < Devise::SessionsController
2    respond_to :json
3
4    def create
5      warden.authenticate!(:scope => resource_name, :recall => "#{controller_path}#\
6  failure")
7      render :json => current_user.to_json
8    end
9
10   def destroy
11     super
```

```
12    end
13
14    def failure
15      render :json => {:success => false, :errors => ["Login Failed"]}
16    end
17
18  end
```

Line 4-7

This is the Rails sign in method (it is creating a session). Line 5 is unchanged from the Devise defaults. In line 6, we return JSON rather than redirecting

Line 8

Since we are not changing the sign out behavior, we default to Devise default behavior with `super`

And, we need to tell Rails to use our new Sessions controller

config/routes.rb

```
1  AngulaRails::Application.routes.draw do
2    get "secure/show"
3    devise_for :users, :controllers => { :sessions => "sessions" }
```

Line 3

Replace the `devise_for :users` with this line that tells Devise to use our new SessionsController, instead of its default Devise::SessionsController.

Try it out!

Go to `http://localhost:3000/secure/show` Login with email:`jwo@example.com` & password:`12345678`. You can now sign in and out using AngularJS & Devise

Conclusion

We explored a lot of ways we can interact with Rails and Devise using AngularJS. Yet again, this is a demonstration on how flexible AngularJS is and how easily it integrates with Rails.

18 Deploy to Heroku

In this chapter we are going to share some beginner tips for deploying your Rails and AngularJS application to Heroku.

What we will cover:

- Turning off uglification
- Getting around uglification with AngularJS

18.1 Brief overview of AngularJS's dependency injection & Uglification

The dependency injection of AngularJS is one of its core awesome features. The default way it works is by turning the function you are calling into a string. Type the following JavaScript into your console to see `toString()` in action.

```
> var fx = function(foo, bar){};
undefined
> fx.toString()
"function (foo, bar){}"
```

After AngularJS has the function as a string, it can read the name of parameters of the function. With the names of the parameters, AngularJS can look through it's services and providers to inject the correct object.

Given that knowledge, it is easy to see how AngularJS's injection magic works.

```
AngulaRails.factory "Book", () ->
  # BOOK SERVICE CODE

AngulaRails.controller "BooksController", ($scope, Book, $http) ->
  # BOOK CONTROLLER CODE
```

When we are working on Rails in our development environment, the uglification of the asset pipeline is turned off. When we push to Heroku, the uglification negatively impacts AngularJS. All that string matching doesn't work when Uglifier "mangles" our variable names. Mangling changes variables from names like `Book` to `aa`. With the name changed, AngularJS can no longer do it's magic of finding the dependency. We get error messages like:

```
Error: Unknown provider: aaProvider
```

So what are we to do? Have no fear, we have a couple of solutions. An added bonus is that they are both super easy.

18.2 Turn mangle off

To turn off mangle, we just need to change the configuration for Uglifier.

/config/production.rb

```
  config.assets.js_compressor = Uglifier.new(mangle: false)
```

If you use this method, you need to make sure that your `uglifier` gem is not in the `:assets` group in your `Gemfile`.

18.3 Specify the dependency using AngularJS

Another option is another method of specifying dependencies with AngularJS. With our normal method of adding a controller, we passed in a string containing the name of the controller and then the function to construct the controller. What we can do is instead of passing in just the function, we pass in an array. The last element of the array will be the function and the other elements will be the dependecies.

```
AngulaRails.controller "BooksController", ["$scope", "Book", "$http", ($scope, Bo\
ok, $http) ->
  # BOOK CONTROLLER CODE
]
```

By using this method, AngularJS will use the names provided in the first elements in the array to match and inject dependencies.

Conclusion

Pushing to Heroku is often our first exposure to the conflict of uglification and AngularJS's dependency injection magic. Lucky for us we have two easy methods to get around the issue. We can either turn off mangle or use AngularJS's more specific method of specifying dependencies.

19 Reduce HTTP Calls

By default, AngularJS sites are slower than they need to be. Most AngularJS sites are bogged down with multiple HTTP calls on the initial page load. Multiple HTTP calls result in slower response time, and higher heroku bills. Sites that load slow can see users flee your site, especially on Mobile connections.

Imagine your site responding as fast as possible, with fast load times, and impressive interactivity. Speed is a feature, nay, THE Feature of your site, and you want to do everything you can to make a speedy first impression for your visitors.

The easiest way to speed up your site with the minimal investment of your time is to reduce the number of HTTP calls from your site to a server, each of which slow down your site (Yahoo 2006[1]). HTTP call reduction 'is the most important guideline for improving performance for first time visitors'.

AngularJS apps have two low hanging, easy wins for reducing HTTP calls: ng-init for preloading JSON, and preloading your templates using the asset pipeline.

19.1 Preload JSON with AngularJS

In most applications, the very first thing your app does is fetch data, using a pattern similar to the following:

```
YourApp.controller "SuperFun", ($scope, $http) ->
  fetchRecords = () ->
    $http.get('/records')
    .success (data) ->
      $scope.records = data
```

```
<div ng-controller='SuperFun' ng-init='fetchRecords()'>
```

This loads your page and kicks off an HTTP request. The AJAX request has to connect with your Rails instance, parse its response, add it to the scope, and then render the records. Slow!

Instead, you want to pass JSON into your ng-init from Rails (hopefully caching it there). That'll look like:

[1] http://developer.yahoo.com/performance/rules.html

```
<div ng-controller='SuperFun' ng-init='setRecords(<%= json_for(@records))'>
```

```
YourApp.controller "SuperFun", ($scope) ->
  setRecords = (records) ->
    $scope.records = data
```

Add to your Gemfile:

Gemfile

```
gem "active_model_serializers"
```

terminal

```
> bundle
```

terminal

```
>rails g serializer record
```

In your Rails `application_helper.rb` RailsCast Credit[2]

application_helper.rb

```
def json_for(target, options = {})
  options[:scope] ||= self
  options[:url_options] ||= url_options
  target.active_model_serializer.new(target, options).to_json
end
```

In your Rails controller

[2] http://railscasts.com/episodes/409-active-model-serializers?view=asciicast

app/controllers/angular_controller.rb

```
class AngularController < ApplicationController
  def show
    @records = Record.all
  end
end
```

Restart your Rails instance

BAM! no more AJAX data requests. Your pages will load noticeably faster.

19.2 Preload Templates with Rails Asset Pipeline

When your AngularJS app reaches a certain size, you'll likely move to using routes, which load HTML templates. Continuing our example, you might first show a list of records, and when clicked, transfer the user to a record detail page. You might add a routes file, and following most examples, it'll look something like this:

routes.js.coffee

```
YourApp.config ($routeProvider, $locationProvider) ->
  $routeProvider
    .when "/",
      templateUrl: "../../assets/records/list.html"
      controller: "DashboardController"
    .when "/records/:uid",
      templateUrl: "../../assets/records/detail.html"
      controller: "RecordsController"
    .otherwise redirectTo: "/"
  $locationProvider.html5Mode(true)
```

The template URL is a location for AngularJS to pull the HTML. So, before it can use the template to List, it needs to make an HTTP request, interpret it into JavaScript, and run the AngularJS commands you've added to it.

Instead, we can compile these templates into the Rails asset pipeline JST, using the ECO gem.

add `gem 'eco'` to your Gemfile and run `bundle` Move your templates into app/javascripts/angular/templates Rename the templates to `detail.jst.eco`, and use them the exact same way as HTML.

Finally, change your routes file to:

routes.js.coffee

```
YourApp.config ($routeProvider, $locationProvider) ->
  $routeProvider
    .when "/",
      template: JST['angular/templates/records/list']
      controller: "RecordsController"
    .when "/records/:uid",
      template: JST['angular/templates/records/detail']
      controller: "RecordsController"
    .otherwise redirectTo: "/"
  $locationProvider.html5Mode(true)
```

Try it out

The result? Your application.js has access to a compiled version of the templates in the global JST variable. AngularJS pulls from there instead of making an HTTP call.

20 Appendix 1 : JSON Models are good enough

Javascript isn't a classical object-oriented language, and yet many client-side frameworks try to bring in object-oriented concepts. Frameworks, like Ember and Backbone, have model objects that inherit from model base classes where you explicitly defined properties. These model classes can then be extended to hold domain behavior for your application.

Angular is different in that there are no model classes. Angular binding works directly with regular JSON. No need for wrappers. Change the value in the JSON and your bound UI gets updated. Behavior isn't tacked onto models, but instead lives on functional services that are injected into your controllers.

Reading and writing data

Whether Backbone or Ember, models end up with a lot of ceremony around declaring properties. There is a significant amount of code to set up well-defined objects for dynamic data. Sure, there are automapping techniques and libraries, but it is still a concern you have deal with. And the more dynamic and hierarchical your data, the more pain it is to model out. Ugh!

```
// Backbone
var Starship = Backbone.Model.extend({
  defaults: {
    name: "",
    captain: ""
  },
});
```

```
// Ember
App.Starship = DS.Model.extend({
  name: DS.attr(),
  captain: DS.attr()
});
```

With Angular, you don't need to specify what properties you are going to bind to. Angular binding works on normal JSON objects.

```
// Angular
$scope.starship = {
  name: "Enterprise",
  captain: "Kirk"
}
```

This might seem trivial at first, but when you start working with hierarchical data whose format can change trying to fit it into a static model can quickly become a nightmare. Spending less time making boilerplate models == WIN.

Updating data with Backbone and Ember models can also a challenge.

```
// Backbone
var enterprise = new Starship({
  name: "Enterprise",
  captain: "Kirk"
});
enterprise.set({ captain: "Picard" })
```

```
// Ember
var enterprise = App.Starship.create();
enterprise.set("captain", "Picard");
```

With Angular, you change the value of a property like you would any other object.

```
// Angular
$scope.starship.captain = "Picard";
```

Ultimately this means that your Angular application deals a lot more with updating JSON than packing data into objects in order to manipulate the data. The result is you spending less time configuring models and more time building your application.

ng-resource

This is not to say that you cannot have model-like constructs in an Angular application. For those who like RESTful route CRUD models, there is the ng-resource module. This allows the creation of model-like instances to be created to easily integrate with a RESTful API.

```
// Angular
AngulaRails.factory("Starship", function($resource) {
  return $resource("/starship/:id");
}

AngulaRails.controller("StarshipsController", function($scope, Starship) {

  $scope.getStarships = function() {
    $scope.starships = Starship.query();
  }

  $scope.save = function() {
    Starship.save($scope.starship);
  }

});
```

Where does the domain code go?

If you want to have a robust, object-oriented domain model on your client-side application then frameworks like Ember and Backbone could be more attractive. Your painstakingly built and maintained models can be home to your domain code.

```
// Backbone
var Starship = Backbone.Model.extend({
  defaults: {
    name: "",
    captain: ""
  },
  goToWarp: function() {
    // complicated code goes here
  }
});
```

```
// Ember
App.Starship = DS.Model.extend({
  name: DS.attr(),
  captain: DS.attr(),
  goToWarp: function() {
    // complicated code goes here
  }
});
```

On the other hand, you can embrace javascript's functional aspect and leverage Angular services. Use composition over inheritance to build your client-side domain logic.

```
// Angular
AngulaRails.factory("WarpEngine", function() {
   return {
    goToWarp: function(starship) {
        // complicated code goes here
    }
   }
}
```

Conclusion

Why be object-oriented in javascript? Instead, use a declarative, functional client side framework. Discover what thousands of other developers have discovered: That binding to JSON and using functional services makes development with Angular easy and fast.