Rails 4.0 with MongoDB and Mongoid

## Requirements

This tutorial assumes that you already have Ruby (with DevKit), Rails and MongoDB installed. If not, you can follow the instructions at their respective websites using the default installations.

I use a batch file to get MongoDB running.

cmd /k "C:\Program Files\MongoDB\Server\3.0\bin\mongod.exe"

The default install of MongoDB puts it in the “P{rogram Files” folder. In order to use a path with a space on the command line, it must be in quotes.

## Setup the Environment

---------------------------------------Do I need this?------------------------------------------------

Create the mongo db first. Create the folders for the database in the default location at “C:\data\db”

Run the batch file to start MongoDB then open a command prompt and start the Mongo interface.

C:\Program Files\MongoDB\Server\3.0\bin\mongo.exe

Create you database using the Mongo interface. Mongo won’t save and empty database, so create a temporary collection and document.

> use myDB

> db.users.save( {username:"tempUser"} )

Creating and populating the collection will be taken care of with Rails.

---------------------------------------------------------------------------------------------------------

Generate your Rails application with

$ rails new myapp --skip-active-record --skip-turbolinks --skip-spring --skip-test-unit

The --skip-active-record is important because it doesn't include ActiveRecord in the app that is generated. We don’t want to use TurboLinks, because it’s going to clash with the Java-Script we’ll be writing later on when we start to use AngularJS. We’re skipping Spring as well, mostly because it isn’t 100% reliable, and it could cause your experience with these examples to not mimic the one we’re describing in the book. Finally, we’re skipping Test::Unit as our testing framework, because we’re going to be using RSpec.

We need to modify the Gemfile to remove sqlite3 and add Mongoid. You'll want to add these lines to your Gemfile:

gem'mongoid', '~> 5.0.0'

gem'bson\_ext'

gem "nokogiri", ">= 1.6.7.rc"

Gemfile

---------------------------------------Do I need this?------------------------------------------------

I had an issue with nokogiri so I downloaded and installed it manually (https://github.com/paulgrant999/ruby-2.2.2-nokogiri-1.6.6.2-x86-x64-mingw32.gem/raw/master/nokogiri-1.6.6.2-x64-mingw32.gem).

$ gem install --local C:\Users\$user$\Downloads\nokogiri-1.6.6.2-x64-mingw32.gem

Now that these are added you can run bundle to install the gems for your Rails app.

----------------------------------------------------------------------------------------------------------

Now we need to generate the Mongoid configuration file that is very much like your config/database.yml that you may be used to with ActiveRecord. We'll create the mongoid file by running the following command:

$ rails g mongoid:config

This generates config/mongoid.yml which you can take a look at and make any configuration changes as necessary (ie. Connection parameters).

Modify config/mongoid.yml so the app can connect to the database:

development:

sessions:

default:

database: myDB

hosts:

- localhost:27017

options:

options:

test:

sessions:

default:

database: myDB

hosts:

- localhost:27017

# This is my production setup at Heroku.com. Yours will vary.

production:

sessions:

default:

uri: <%= ENV['MONGOHQ\_URL'] %>

options:

config/mongoid.yml

(If you want to be able to inspect your database, I recommend installing a separate application “RoboMongo”.)

## Create the DefaultPage

Add the route to config/routes.rb.

root 'home#index'

config/routes.rb

Next, create app/controllers/home\_controller.rb.

class HomeController < ApplicationController>

def index

end

end

app/controllers/dashboard\_controller.rb

Last, create app/views/home/index.html.erb with some basic content.

<header>

<h1>

Welcome to my Rails/Mongodb application!

</h1>

<h2>

We're using Rails <%= Rails.version %>

</h2>

</header>

<section>

<p>

Future content goes here!

</p>

</section>

app/views/home/index.html.erb

Test it out.

$ rails server

## Install Devise

First, do a manual install of the bcrypt gem to insure DevKit builds the dependencies.

$ gem install bcrypt --platform=ruby

If this gets screwed up, do an uninstall of bcrypt and insure bcrypt is in a folder called bcrypt-3.1.10, or what ever is appropriate to the version your are using. Not bcrypt-3.1.10-x64-mingw32.

Add Devise and bcrypt to our Gemfile.

gem 'devise'

gem 'bcrypt'

Gemfile

Next, we’ll install it using Bundler.

$ bundle install

Devise includes several generators we can use to simplify the setup and initial configuration. The devise:install generator is the first one we’ll need to include Devise in our app.

$ rails generate devise:install

Next, tell Devise what model and table we want to use for authentication.

$ rails generate devise user

Next, indicate which controller actions require authentication. Since we want all pages and actions to be restricted, put the devise controller filter in the ApplicationController.

Class ApplicationController < ActionController::Base

protect\_from\_forgery with: :exception

before\_action :authenticate\_user!

End

app/controllers/application\_controller.rb

Last, put the logged in user’s email in the default page as verification that the authentication worked.

<header>

<h1>

Welcome to my Rails/Mongodb application, <%= current\_user.email %>

</h1>

<h2>

We're using Rails <%= Rails.version %>

</h2>

</header>

<section>

<p>

Future content goes here!

</p>

</section>

app/views/home/index.html.erb

Now, after restarting your app, you will be asked to log in. All the authentication functionality and view have been provided by Devise.

## Using Bower to Install Bootstrap

Bower allows us to download Bootstrap and store it with our application’s source. Bower is written in JavaScript, and so it requires a JavaScript runtime in order to work. That means you’ll have to install Node.js.

To install Node, simply visit <http://nodejs.org> and follow the instructions for your operating system. Once you have Node installed, you’ll have access to the npm command-line application. This is a package manager for JavaScript, and we’ll use that to install Bower.

$ npm install -g bower

Add the bower-rails to the Gemfile.

Gem ‘bower-rails’

Gemfile

Install the gem with Bundler

$ bundle install

The gem allows us to specify dependencies in a simple file called **Bowerfile**, which will be easier to work with than the JSON format required by the bower command-line app. Bower-rails also provides rake tasks to run Bower for us.

Now that Bower is installed, we’ll add our first front-end dependency, which is for Bootstrap. Bower-rails will look for Bowerfile to find our list of dependencies. It looks very similar to a Gemfile and exposes the method asset for specifying a front-end dependency. Create Bowerfile in the app root. Add the following line to the file.

asset 'bootstrap-sass-official'

Bowerfile

Now use rake to tell bower to install the dependencies

$ rake bower:install

JQuery will also be installed along with Bootstrap.

## Configure the Asset Pipeline

The Rails asset pipeline manages the deployment of front-end assets to a user’s browser. Essentially, it allows us to organize our JavaScript and CSS however we’d like, but have all of it packaged up at runtime into only two files: one for CSS, one for JavaScript.

Rails’ default configuration of the asset pipeline grabs all files in app/assets/stylesheets and app/assets/javascripts and packages them as application.css and application.js, respectively. Since we’ve used Bower to bring in CSS (and, eventually, JavaScript) outside of app/assets, we’ll need to add a bit more configuration for the asset pipeline to know about them.

Much like how we piece together a Ruby application by setting paths and using require, the asset pipeline (which is powered by sprockets), is configured with asset paths which contain directives which describe all the files we want to serve as assets.

Bower-rails will automatically add vendor/assets/bower\_components to the asset path for us, so there’s no need to do any additional configuration. To use Bootstrap’s CSS files, we’ll use the require directive to tell it to bring them into our application. Because the default application layout references application.css we can add this directive in app/assets/stylesheets/application.css, which is where Rails places that file by default.

/\*

\*= require\_tree .

\*= require\_self

\*= require 'bootstrap-sass-official'

\*/

app/assets/stylesheets/application.css

## Styling the Views with Bootstrap

The login and registration forms are provided by Devise. To style these forms we must extract them into our application.

$ rails generate devise:views

Put a logout button on the home page so we can see the login and registration pages again. Eventually the logout button will reside in our nav menu.

<section>

<p>

Future content goes here!

</p>

<%= link\_to "Log Out", destroy\_user\_session\_path, method: :delete %>

</section>

app/views/home/index.html.erb

Before editing the login and registration files, add Bootstrap’s container-fluid class to the layout and a way to display notices and alerts.

<body>

<main>

<div class="container-fluid">

<% if notice.present? %>

<aside class="alert alert-info alert-dismissible" role="alert">

<button type="button" class="close" data-dismiss="alert" aria-label="Close"><span aria-hidden="true" class="glyphiconglyphicon-remove-circle"></span></button>

<%= notice %>

</aside>

<% end %>

<% if alert.present? %>

<aside class="alert alert-warning alert-dismissible" role="alert">

<button type="button" class="close" data-dismiss="alert" aria-label="Close"><span aria-hidden="true" class="glyphiconglyphicon-remove-circle"></span></button>

<%= alert %>

</aside>

<% end %>

<div class="clearfix"></div>

<%= yield %>

</div>

</main>

</body>

app/views/layouts/application.html.erb

Modify the login page.

<div class="row">

<div class="col-md-4 col-md-offset-4">

<h1>Log in</h1>

<%= form\_for(resource, as: resource\_name,

url: session\_path(resource\_name)) do |f| %>

<div class="form-group">

<%= f.label :email %>

<%= f.email\_field :email, autofocus: true, class: "form-control" %>

</div>

<div class="form-group">

<%= f.label :password %>

<%= f.password\_field :password, autocomplete: "off", class: "form-control" %>

</div>

<% if devise\_mapping.rememberable? -%>

<div class="checkbox">

<label>

<%= f.check\_box :remember\_me %> Remember Me

</label>

</div>

<% end -%>

<%= f.submit "Log in", class: "btnbtn-primary btn-lg" %>

<% end %>

<%= render "devise/shared/links" %>

</div>

</div>

app/views/devise/sessions.new.html.erb

Modify the registration page.

<div class="row">

<div class="col-md-4 col-md-offset-4">

<h2>Sign up</h2>

<%= form\_for(resource, as: resource\_name,

url: registration\_path(resource\_name)) do |f| %>

<%= devise\_error\_messages! %>

<div class="form-group">

<%= f.label :email %><br />

<%= f.email\_field :email, autofocus: true, class: "form-control" %>

</div>

<div class="form-group">

<%= f.label :password %>

<% if @validatable %>

<em>(<%= @minimum\_password\_length %> characters minimum)</em>

<% end %><br />

<%= f.password\_field :password, autocomplete: "off",

class: "form-control" %>

</div>

<div class="form-group">

<%= f.label :password\_confirmation %><br />

<%= f.password\_field :password\_confirmation, autocomplete: "off", class: "form-control" %>

</div>

<%= f.submit "Sign up", class: "btnbtn-primary btn-lg" %>

<% end %>

<%= render "devise/shared/links" %>

</div>

</div>

app/views/devise/registrations/new.html.erb

## Adding AngularJS

Add the angular asset to the Bowerfile

asset 'bootstrap-sass-official'

asset 'angular', '~> 1.4'

Bowerfile

Install angular using the rake task

$ rake bower:install

Add bootstrap and angular to the asset pipeline.

//= require jquery

//= require jquery\_ujs

//= require bootstrap

//= require angular

//= require\_tree .

app/assets/javascripts/application.js

Set up an angular test page. Add this line to our routes.

get "angular\_test", to: "angular\_test#index"

config/routes.rb

Create a test controller.

class AngularTestController < ApplicationController

def index

end

end

app/controllers/angular\_test\_controller.rb

Create a test page where some header text is bound to an input.

<article data-ng-app="angular\_test">

<header>

<h1 ng-if="name">Hello, {{name}}</h1>

</header>

<section>

<form class="form-inline">

<div class="form-group">

<label for="name">Name</label>

<input class="form-control"

name="name"

type="text"

placeholder="Enter your name"

autofocus

ng-model="name">

</div>

</form>

</section>

</article>

app/views/angular\_test/index.html.erb

Using “data-ng-app” instead of just “ng-app” ensures the HTML will validate.

Define the ng-app name.

angular.module('angular\_test',[ ]);

app/assets/javascripts/angular\_test.js

Set up a link to our test page on the Home page. If this works, then AngularJS has been set up correctly.

<%= link\_to "Log Out", destroy\_user\_session\_path, method: :delete %>

<br/>

<%= link\_to "Angular Test", 'angular\_test' %>

</section>

app/views/home/index.html.erb

[@@so far so good!]

## Setting UpRSpec

Add RSpec to Gemfile

group :development, :test do

gem "rspec-rails"

end

Gemfile

Then

$ bundle install

Modify the configuration file.

RSpec.configure do |config|

config.expect\_with :rspec do |expectations|

expectations.include\_chain\_clauses\_in\_custom\_matcher\_descriptions = true

expectations.syntax = [:expect]

end

config.mock\_with :rspec do |mocks|

mocks.verify\_partial\_doubles = true

mocks.verify\_doubled\_constant\_names = true

end

config.filter\_run :focus

config.run\_all\_when\_everything\_filtered = true

config.disable\_monkey\_patching!

config.expose\_dsl\_globally = true

if config.files\_to\_run.one?

config.default\_formatter = 'doc'

end

config.profile\_examples = 10

config.order= :random

Kernel.srandconfig.seed

end

spec/spec\_helper.rb

Create a dummy spec file to verify everything is working

require "rails\_helper.rb"

describe "testing that rspec is configured" do

it "should pass" do

expect(true).to eq(true)

end

it "can fail" do

expect(false).to eq(true)

end

end

spec/dummy\_spec.rb

Use rake to run the test

$ rake

You should see some output from the test. This was just a validation. Do some research to discover how to use RSpec.

## Conclusion

And that is our barebones app with Rails, Mongoid, Devise, Bootstrap, AngularJS and RSpec! While it's much more work than you might expect, I imagine this will become a smoother process in the future once the Rails 4 gem has been released. Be sure to check out [Mongoid's documentation](http://mongoid.org/) for more information on how to use it.