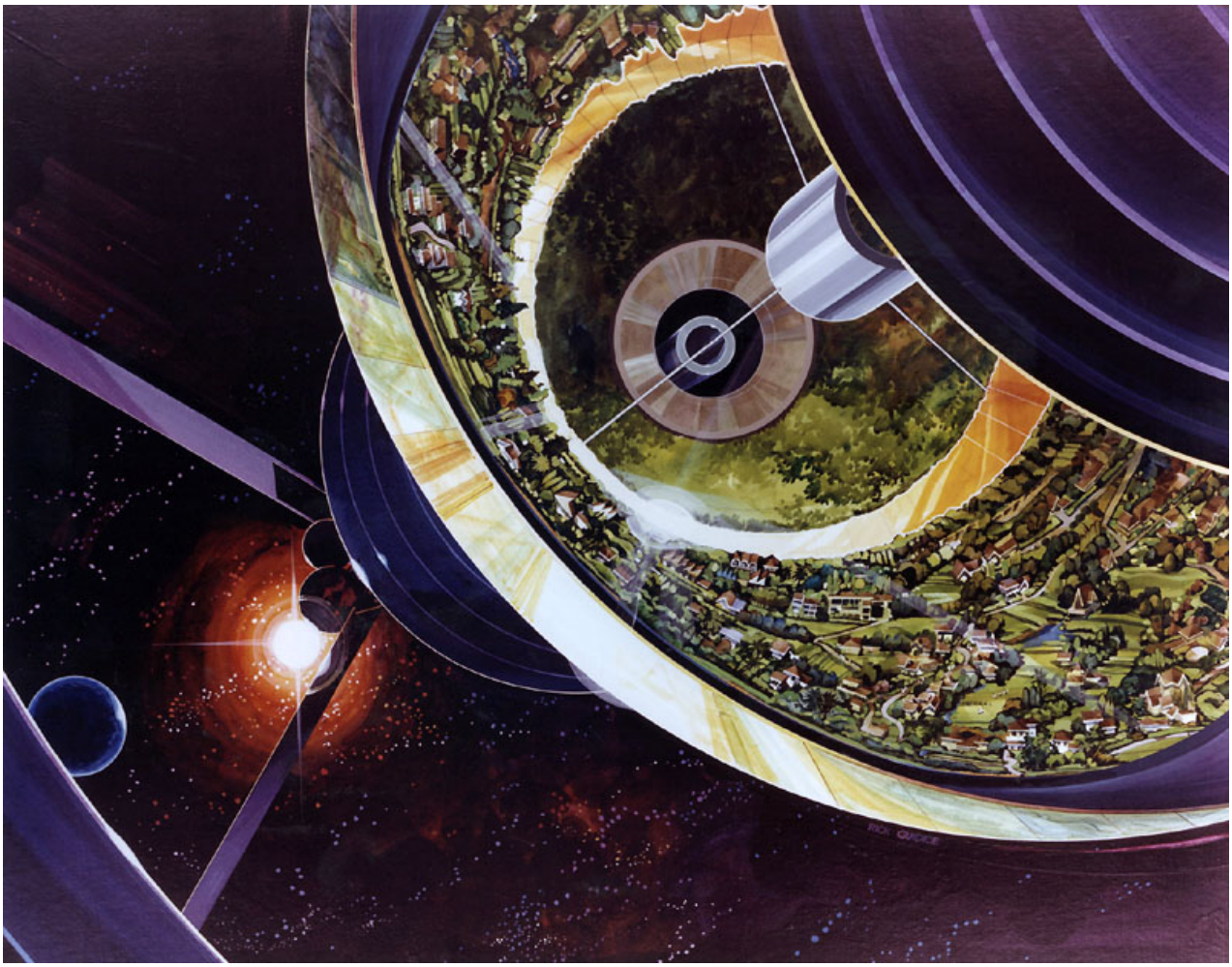
The Rails Book

Step by Logical Step

By Nicholas Johnson

version 0.9.0 - Beta



Why you you might quite like Rails

Rails is a rapid application development framework. It launched Twitter, and thousands of other startups all over the world.

It excels at producing minimum viable products. If you have an idea which you want to take to market very quickly, Rails is probably what you need.

Hello World

Hello World is the traditional way to start any new framework. Hello World in Rails is fairly simple, around six lines of code. In this section we'll create an MVC hello world using a simple model, a controller, and a view, in fact the full MVC stack.

Create your rails application

At a terminal type:

```
rails new HelloWorld
```

This initialises a new blank application called HelloWorld

Start your rails application server

At a terminal navigate to your HelloWorld directory and type:

```
ruby script/rails server
```

Wait for the application to start then in a web browser visit:

http://localhost:3000

Creating a Controller

Create a new file in the app/controllers directory called hello_controller.rb

```
class HelloController < ApplicationController
  def index
    @text = "Hello World!"
  end
end
```

This controller defines a single method called index. Index is the default controller action. If you're familiar with HTML this should make sense.

The index method creates an instance variable called @text.

Creating a View

In the app/views/hello directory (you'll need to create this directory) create a file called index.html.erb containing the following

```
<h1>Hello World</h1>
<%= @text %>
```

Views have access to the controller's instance variables. You don't need to do any work to pass them through, they're just there.

Create a route

Visit config/routes.rb. Take a moment to read the helpful instructions, then add the line:

```
1    get '/hello' => 'hello#index', :as => :hello_world
```

This line creates a named route (called hello_world). It matches the

Hey Presto

In a web browser visit:

http://localhost:3000/hello

And there we are. Of course we're only touching on the awesome power of Rails here, but this should give you an idea of how easy things are to get going.

Exercise - Goodbye World

Change your app into a goodbye world app. Here are the thing's you'll need to do:

- Add a new controller goodbye_controller.rb.
- Add a route.
- Add a view

Make it so that you can visit http://localhost:3000/goodbye to see the goodbye view.

Exercise - Tell the time

We're going to create a simple website that displays the current time and date.

1. Create a controller called something like time_controller.
2. Add a route
3. In your controller create a variable called @time which contains the value: Time.now. This is just the current time.
4. Write a view which outputs the value of the @time variable. Look up strftime for help with this.

Exercises - Root URL

Adjust your app so it responds to the root url. You should be able to visit:

http://localhost:3000/

and see hello world.

- First delete public/index.html
- Now view the instructions in config/routes.rb to create a default URL.

Debugging Rails

Rails comes with a couple of fully featured debugging systems.

The Console

The rails console lets you fire up an instance of your rails server on the command line. You have full access to Active Record and all of your models. It's terribly good.

Fire it up with:

```
rails console
```

or just

```
rails c
```

You can create models, define methods, and do anything you can do in a regular rails instance.

If you make changes to your code, you can load them in to your console by typing:

```
reload!
```

The Debugger

If you need to poke around inside a running Rails instance to see what's going on, you can use the debugger. To use it you must start your Rails instance in debug mode like so:

```
rails server --debugger
```

or just

```
rails s --debugger
```

Then insert breakpoints into your code by typing:

```
debugger
```

You can put breakpoints in your models, views or controllers. Within the debugger you have access to all of the variables defined at or before the breakpoint. You can change variables,

execute code, write ruby, etc, etc.

To exit the debugger, continue with

```
c
```

Tip: Pressing enter will repeat the last console command. If you have multiple debug statements, you can just hold down enter to skip through them.

Exercise - Debugging

In this exercise we'll have a go at using the debugging tools.

Let's use the debugger to debug our Hello World server. Fire up your hello world instance in debug mode. Now insert a breakpoint in the hello controller (or goodbye controller), right after you define the @text instance variable.

Hit the URL, and the app will freeze. Go to your terminal, and you'll find it's waiting for input. You can view the contents of the @text variable, and even change it's value.

Layouts and Partials

Layouts allow you to wrap your code in a standard header and footer.

The default layout is called app/views/layouts.application.html.erb

Layouts Exercise

Your default layout file lives in app/views/application.html.erb. Any changes you make to this file will be visible in every page on your site, unless you explicitly use a different layout.

Add a header and footer to your application.html.erb file. It should be visible on each of your pages.

Passing a title

Try to pass through a @title variable from the controller to the view. Use this as the page title at the top of your layout file.

Partials

Partials are partial views. They allow you to break your code up into smaller sensible chunks.

Partials exercise

Partial filenames start with an underscore.

Create a header partial called /app/views/layouts/_header.html.erb

Include it in your template using the render method, like so:

```
1    <%= render partial: "layouts/header" %>
```

You may wish to create partials for things like:

- Headers and footers
- Metadata
- JavaScript includes
- Facebook widgets
- Tables
- Forms

Create a footer partial now.

Params

The params hash contains all the url and post parameters that were passed to your application.

Exercise - Make a calculator

We are going to make a simple calculator, you will be able to give it two numbers, and have it give you the sum.

First create a controller and route. The routes should have space for two numbers. It should be possible to hit a URL like this:

http://localhost:3000/calculate/12/15

Retrieve the values and store in instance variables

Now in your controller, retrieve these from the params hash and store them in instance variables, @number_1 and @number_2

Sum them in the view

Finally create a view. In the view, sum together the two numbers, giving a nice output.

Optional Extension

Try and think of a way you can make this work using a form. You will need to change the route to accomplish this.

Models, Validation and Resources

We are going to create a Cat model to held information about a single cat. Use the model generator to create a simple model:

```
rails g model cat name:string description:text
```

You will see we have made a nice cat class in the models directory, and also a migration in the db/migrations directory.

Run the migration using:

```
rake db:migrate
```

This has updated your database schema.

Test using the console

Drop into the rails console by typing:

```
rails c
```

Now you can play with your Cat.

Try out the following commands in the console:

```
c = Cat.create name: "Markey"
Cat.all
c.name = "Dave"
c.save
Cat.first
Cat.first.name
Cat.find(1)
```

What do they all do? You can do all this in your controller.

Now create 10 more cats, and try out the following.

```
Cat.order(:name)
Cat.order(:name).limit(5)
Cat.where(name: "Dave")
Cat.where("created_at > ?", Time.now - 1.minute)
Cat.count
```

Making use of our cat

Now create a controller and route. The route should point to a method called show in the controller. It should be possible to visit:

http://localhost:3000/cats/1

In the controller retrieve the cat based on the id in the params hash.

Now make a view to display your cat. Output all the fields nicely.

Optional Extension, create an index page

Create a route of the form:

http://localhost:3000/cats

This should point to the index method in the cat_controller. This method should retrieve all of the cats with Cat.all.

Make a view which loops over the cats, displaying them all. To loop over a collection, you can do something like this:

```
1    <% @cats.each do |cat| %>
2      <%= cat.name %>
3    <% end %>
```

Scaffolding

The Scaffold Generator allows us to create a controller, model, views, partials, stylesheets and tests with a single call.

In this section we are going to look at using scaffolding to create a CRUD app in double quick time.

Scaffold the resource

It's traditional to make a blog, but feel free to make a kitten, or similar.

First create a BlogPost scaffold and give it some attributes. You can use the following generator as a jumping off point. You will need to add a content:text and probably also a date:datetime too.

```
1    rails g scaffold blog_post title:string
```

Look over the controller

Take a look at the controller that was made, you will find 7 standard actions allowing you to create, edit, show, index and destroy (edit and create get two methods each).

Spend a few minutes reading through the code and understanding it.

Look at the routes

You will find one line has been added to the routes file, resources.

This single line generates all of the crud routes for you. Check them out from a console by typing:

```
1    rake routes
```

Views

Look at the views. See the form partial? It's used by the new and edit templates. Have a read and try to understand what's going on.

Tests

The tests that have been built for you should work right out of the box. Run:

```
1    rake test
```

to run all of the tests.

Further Exercise: Validation

Use validates_presence_of :title to validate that the blog_post has a title. It is now not possible to save a blog_post without a title. Add validation for the content.

Try and create a blog post without a title, look at the error reporting. Do you see how it works?

Further Further Exercise: Homepage

Set blog_post#index as the homepage, so when you visit your Blog, you see a list of entries.

Harder Exercise: Friendly URLs

Add a slug attribute to the blog_post. Do a find_by_slug instead of a regular find in your show method like this:

```
1    BlogPost.find_by_slug params[:id]
```

Use a migration to add the field, generate the migration like this:

```
1    rails g migration add_slug_to_blog_post
```

Within the migration you will want to do something like this:

```
1    add_column :blog_posts, :slug, :string, index: true
```

Finally, ensure slug is a required field.

You can now hit a URL like this

http://localhost/blog_posts/having-fun-learning-rails

Associations Between Models

We're going to add Comments to our blog. Comments belong to BlogPosts and blog_posts have many comments.

Scaffolding

First of all, scaffold the Comment model. Refer to the last exercise if you can't remember how to do this.

Comments will need several fields, I'll leave this part up to you, but **crucially, comments will need a blog_post_id: integer** field. Notice how I highlighted that part.

Now set up the relationships

You'll need to extend your models something like the following:

```
class BlogPost
  has_many :comments
end

class Comment
  belongs_to :blog_post
end
```

Test the association

Drop into the console and check your association. You should be able to call something like:

```
post = BlogPost.first
post.comments
```

Validation

Add validation to your comment. A comment needs a bog_post_id to be valid, plus a couple of other fields. Enforce this.

Listing comments

On your blog_post show page, list all the comments for a particular blog post. Remember you can use @blog_post.comments to get an array of the comments.

Great. You can now create comments from the comments form, and see them when you view a blog but you will need to manually enter the blog_post_id when creating the comment. Let's fix that.

Extension: Nested Routes

Use a nested route to nest your comment inside a BlogPost. Modify your routes.rb file like so:

```
resources :blog_posts do
  resources :comments
end
```

Use rake routes to check the routes you have made.

```
rake routes
```

Now you can visit a URL like

http://localhost:3000/blog_post/1/comments/new to create a comment. Remove the blog_post_id field from the comment form view. Instead, in your controller set it using the params hash. You may need to drop into the debugger to check the params hash here.

Optional finishing up

Pick from the following:

- See if you can integrate your comment form right into your blog_post_show page. (tip, in the blog_post_show controller execute:

```
@comment = Comment.new.
```

This will let the comment form just work without modifications.)

Sessions

Rails allows you to store session data on the client in the form of an encrypted cookie. You can read from and write to the session cookie simply by accessing to the session hash:

session[:user_id] = 1234

It's up to you how much data you store on the client, though if you store too much you will probably encounter a cookie overrun.

The most common use for sessions is for login. We might log a user in by storing the id of a User model in the session.

We might then log out by session this value to nil.

Read more on sessions here:

http://guides.rubyonrails.org/action_controller_overview.html#session

Beware cookie attacks

Be aware that although the cookie is encrypted, all client side data is potentially open to attack. If your admin user has an id of 1, you might be opening a ready attack vector.

For this reason I generally store user keys in the session, where user keys are large random alphanumeric values generated in some non-standard way.

Exercise - Access the session hash

A simple one first.

1. Create a _header partial which outputs session[:username].
2. Now create a form which submits a username to a controller.
3. In the controller, store the username in the session.

Now as long as the session persists, the username is visible in the header, even if you navigate away from the page.

Exercise - Login

We are going to implement login.

1. Create a user model with an email and password. Use the console to create a few user objects.
2. Your user model should have a login method that accepts a password. If the password matches, this method returns true.
3. Create a session controller. Give it a new method. Have the view render a login form which accepts a username and password.
4. Give your session controller a create method. Post the data from the login form to it. This method should find a user by email, and if one is found, call the login method with the password.
5. If everything checks out, add the user id to the session
6. Now in ApplicationController add a before_filter. This should check the session for a user_id, pull the user from the database and save it in @current_user instance variable.
7. Finally write a partial which displays the session user email.

Bonus - RSpec

Write rspec

Assets and SCSS

Assets are files which belong to the web page, such as images, JavaScripts and CSS files. They live in the app/assets folder.

The asset pipeline is the conduit through which you import stylesheets, JavaScript files, images, etc. They can be served individually or else precached, minified, concatenated, etc.

In development, assets are served individually. This is set in config/environments/development.rb with this line:

```
1    config.assets.debug = true
```

Asset Digest

A Digest is a string which is added to the end of your asset filename. It's created by hashing the file. If any of your files change, the digest is updated, and so the filename is updated. This gets round caching issues.

Turn on digests like this:

```
1    config.assets.digest = true
```

Now view source. You'll see all your assets are compressed.

Asset Compilation

Let's have a look at asset compilation:

By default your development server will serve all your assets as separate files. Turn off asset debugging in config/environments/development.rb by modifying the line:

```
1    config.assets.debug = true
```

to:

```
1    config.assets.debug = false
```

Restart your server to see the results.

Namespacing CSS

Because all your stylesheets are imported with each request, you have to be clever not to apply page specific rules everywhere. I typically add the controller and model as a class to the body

element. This allows me to namespace my SCSS.

app/helpers/application_helper.rb is a module that's imported by all your views. You can define helpful methods in it. The following method will create a string based on your controller, action, and optionally a version parameter. Add it to the application helper, or modify it your purposes.

```
def body_class
  bc = []
  bc << params[:controller].gsub('/', ' ')
  bc << params[:action]
  if @version
    bc << "version_"
  end
  bc.join(' ')
end
```

Now in your app/views/layouts/application.html.erb file, add the new class:

```
<body class="<%= body_class %>" >
```

In your SCSS files you can now write things like:

```
.home.index {
  h1 {
    font-size:5em;
  }
}
```

Exercise - Compiling SCSS

In this section we will look at scss, Look in the app/assets/stylesheets folder, you'll find a set of scss files in there which you can modify. If you create additional files they are automatically included.

1. Create a stylesheet called header.css.scss.
2. Add code to style the header on the page. Perhaps add a pretty pink background or something.
3. Verify the header has been styled.
4. View the page source in your browser. Look for the link tags that include the stylesheets.

Optionally style the footer.

Exercise - Sprockets

1. Now edit config/environments/development.rb.
2. Set config.assets.debug = false
3. Restart the server (you need to restart if you change configuration settings)
4. View the page source. You should see all the assets rolled into a single file.

Digest

1. Edit config/environments/development.rb.
2. config.assets.digest = true
3. Restart and view the page source. See the filename has been rewritten to defeat caching.

RSpec Rails

We are going to use RSpec plus FactoryGirl to test our site.

First we need to add the gems to our Gemfile:

```
group :test, :development do
  gem 'debugger'
  gem 'rspec-rails'
  gem 'factory_girl_rails'
end
```

Now bundle and restart the server.

We now need to initialise the spec directory and helpers. We can do this using:

```
rails generate rspec:install
```

Next Configure Rails use rspec in the generators. Inside Application.rb, add a config block:

```
config.generators do |g|
  g.test_framework :rspec
  g.integration_tool :rspec
end
```

Now when we run a generator we also get a spec file generated for us as well too.

Generate scaffold Rspec

We are going to create a User scaffold for our blog. Posts will have Users.

Generate the scaffold. Now look in the spec directory. See the spec there?

rails g scaffold user name:String age:Integer

Run rake spec to run the specs. It will tell you if you have any errors and where they are.

Exercise - Use FactoryGirl

The spec we generated creates a user model for us. This is OK, but not very scalable. We might need users in other places. We are going to use Factory Girl to make the Users for us.

Check out the FactoryGirl documentation here:

https://github.com/thoughtbot/factory_girl

Now create a User factory. Modify your generated spec to use the factory instead.

Modify the .rspec config file

Edit the .rspec file so it looks like this:

```
--color
--format documentation
```

You will find the output much easier to read.

Testing a model

Say we have a Product model we want to test. We would first create a file spec/models/product_spec.rb

```
require 'spec_helper'

describe Product do
  it "can be initialised"
  it "has a title"
end
```

Running the test would show yellow, as the tests are not yet implemented.

Here I have filled in the initialiser test:

```
require 'spec_helper'

describe Product do
  before :all do
    @product = Product.new
  end
  it "can be created" do
    expect(@product).to be_a(Product)
  end
  it "has a title"
end
```

Now let's implement a new feature, a title:

```
require 'spec_helper'

describe Product do
  before :all do
    @product = Product.new :arkham
  end
  it "can be created" do
    expect(@product).to be_a(Product)
  end
  it "has a title" do
    expect(@product.title).to be("Arkham")
  end
end
```

Exercise - Write specs for your application

First have a looks through the Better Specs guide, it is good:

http://betterspecs.org/

Now, using the generated User specs as a guide, write specs for your scaffold application. You might like to write some specs for your model and Controller. Remember Use FactoryGirl and run your specs often.

Concerns

Concerns are little modules that encapsulate reusable functionality. Rather than adding them to our models and controllers directly, we put them in a module and include them.

An example of a concern might be "Taggable", or "Ratable" or "HasImage"

Exercise - HasImage

We are going to create a HasImage concern which will add standard paperclip functionality to our BlogPost. We will be able to use it like this:

```
class BlogPost < ActiveRecord::Base
  include HasImage
end
```

Define your concern in a module in the app/models/concerns directory. To be automatically imported the file must be have the same name as the module, with underscores.

Because Paperclip requires you to work with the class directly we are going to use the included callback to add class methods to our module, like so:

```
module HasImage
  def self.included(klass)
    klass.extend ClassMethods
  end

  module ClassMethods
  end
end
```

The Paperclip documentation can be found here:

https://github.com/thoughtbot/paperclip

Remember to write specs for your concern.

Further exercise

Create a Rateable concern to allow users to rate content. The model should gain thumbs_up and thumbs_down methods, and a rating method. Store the data either in a field, or in a separate Rating model which belongs to the User model (if you have one)

Alternatives to ERB - Haml and Markdown (also Slim)

We are going to try out Haml as an alternative templating engine. Haml is a beautiful, simple HTML preprocessor. Haml compiles directly to HTML, there's a one to one mapping.

Haml has:

1. Semantic indentation. Close tags are not necessary.
2. Ruby style hashes for attributes.

Exercise

1. We are going to convert one of our views into Haml. Pick one at random from your blog, you could also choose a partial.
2. Add gem 'haml' to your Gemfile. bundle and restart your server.
3. Now check out the Haml documentation here: http://haml.info/
4. Manually convert your view to basic Haml

Exercise - Filters

We are going to use a markdown filter to write text in Markdown. Markdown is a simple text preprocessing language.

1. Check out filters in the Haml docs: http://haml.info/docs/yardoc/file.REFERENCE.html#filters
2. Add a markdown filter to your template
3. Now add a block of text. 2 newlines makea paragraph. A # makes an h1. ## makes an h2.

Writing text blocks in markdown can really save you time and make your ideas flow more clearly.

Internationalisation

Rails Locales

Rails defines Yaml files for internationalisation of strings. These are held in config/locales.

The Yaml files contain a set of strings. To access a string from your controller, use: I18n.t 'hello'. In your view, you can use: <%= t('hello') %>

If a string is not defined in your locale, the default locale is used instead. This can be set in application.rb.

Sample locales.

Check out Sven Fuchs' rails i18n project for a pretty good basic locale set.

https://github.com/svenfuchs/rails-i18n/tree/master/rails/locale

Finding the locale from your URL.

Your locale will not be set for you. You can use your URL to set it, then retrieve it using a filter in application_controller.rb

```
before_action :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

Now if you hit a URL like this:

localhost:3000?locale=fr

The locale will be set to 'fr' (French).

There are various other options here. You can read about them in the documentation here:
http://guides.rubyonrails.org/i18n.html

Exercise: Localise your blog

Create a locale file for a language you know. Now take one of the strings, e.g. 'New Blog Post'. Create a locale file to localise this string.

Blog integration exercise

Create a new Rails App using:

```
rails.new
```

We're going to put everything together and make a simple blog.

Create the model

First create a BlogPost model and give it some attributes. You can use the following generator as a jumping off point. You will need to add a content:text and probably also a date:datetime too.

```
rails g model blog_post title:string
```

Create the controller

Next we'll need a BlogPost controller. Use a generator like this:

```
rails g controller blog_post
```

Create your routes

Use resources to create the standard CRUD routes. Like this:

```
resources :blog_post
```

Create the CRUD methods

Review your scaffold code and create the standard Rails CRUD actions in your controller (index, show, new, create, edit, update, destroy). Create views to go along with them.

Validation

Use validates_presence_of :title to validate that the blog_post has a title. It is now not possible to save a blog_post without a title. Add validation for the content.

Homepage

Set blog_post#index as the homepage

Friendly URLs

Add a slug attribute to the blog_post. Do a find_by_slug instead of a regular find in your show method.

You can use a migration to add the field, something like:

```
rails g migration add_slug_to_blog_post
```

Within the migration you will want to do:

```
add_column :blog_posts, :slug, :string, index: true
```

Comments

We're going to create a Comment model. Comments belong to BlogPosts and blog_posts have many comments. You'll need something like the following:

```
class BlogPost
  has_many :comments
end
class Comment
  belongs_to :blog_post
end
```

Comment will need a blog_post_id:integer attribute.

Use the generator to create the post.

Routes. We'll use a nested route to generate the resources. Modify your routes.rb file like so:

```
resources :blog_posts do
  resources :comments
end
```

rake routes to check the routes you have made.

(side project - read up on routes here: http://api.rubyonrails.org/classes/ActionDispatch/Routing.html and here http://guides.rubyonrails.org/routing.html)

Comment Controller

Make a comment controller with a new and create method. It should receive a comment and create it. Make a matching view and form.

Form

Integrate the comment form into the BlogPost#show view. Integrate the list of comments into this view.

Optional finishing up

- Style your blog nicely using assets.
- Deploy to Heroku. https://devcenter.heroku.com/articles/getting-started-with-rails4