Wild Web

Getting Started with Ruby on Rails

Author : Paul Krause

Wild Web

What are we trying to do?

This tutorial aims to provide an quick introduction to the world of web application development. It uses the programming language Ruby, and a popular framework for developing web applications called Ruby on Rails.

We hope that you will take a copy of this tutorial away with you and explore it a bit more when you get home. In order to avoid having to set up tools on your own computer, the tutorial uses a cloud-based development environment called Nitrous.io. You will, of course, find life easier with a development environment on your own machine, but this way gets you started (hopefully) in a relatively hassle free way.

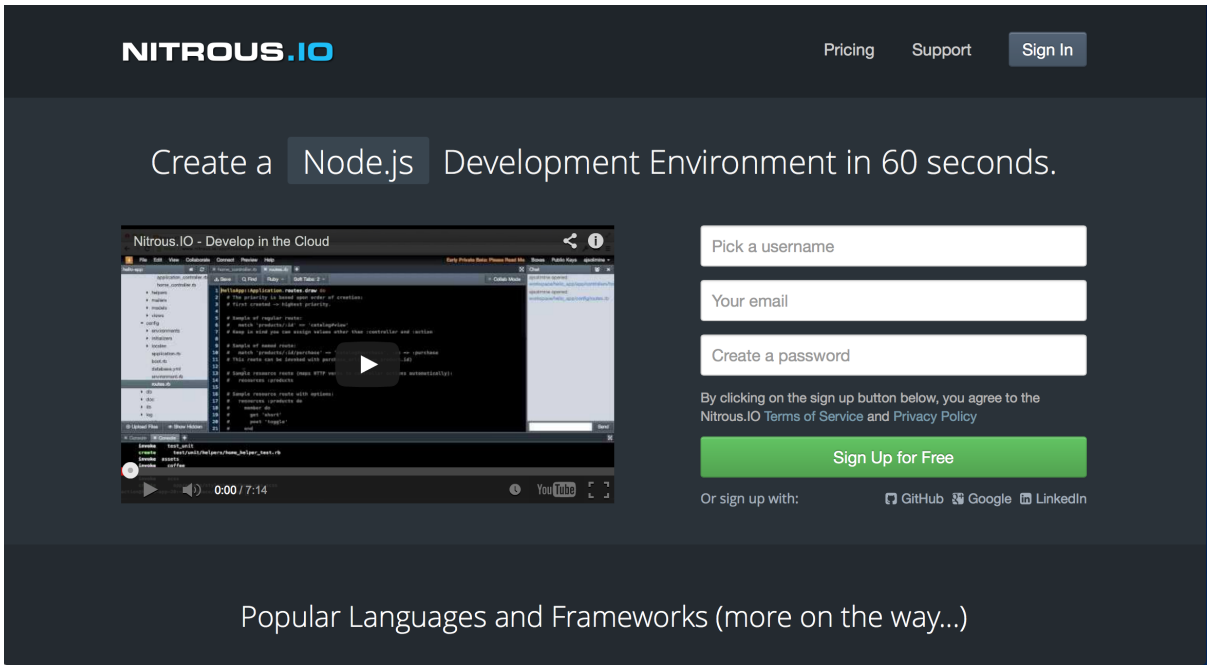
N.B. I have included chapters on the "Ruby Version Manager" and "Updating Rails". These show you how to maintain up to date versions of what is a steadily evolving development environment. However, you can skip these chapters when you go through this tutorial for the first time. In fact this material will not be needed for today's tutorial as Nitrous do keep the default versions of Ruby and Rails up to date.

You should also not worry about installing the Nitrous Desktop as this probably won't work on the computers you are using today.

Nitrous.io

Signing Up

Open up your browser and go to Nitrous.io.

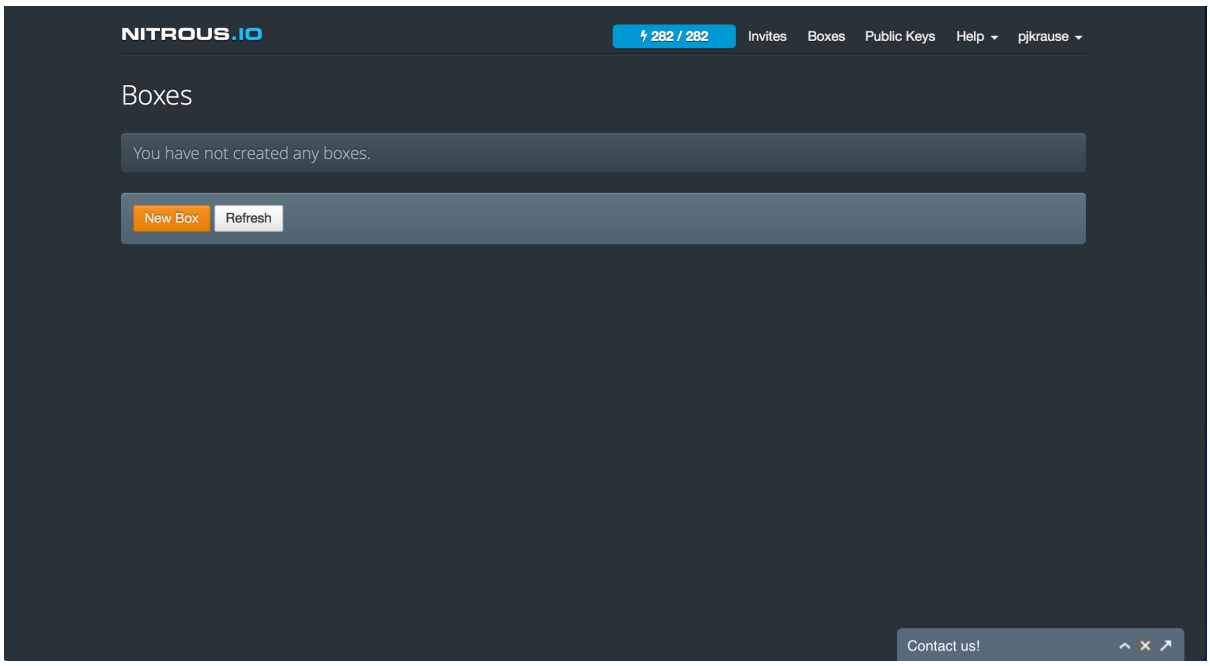


Click on "Sign Up for Free" and follow the instructions.

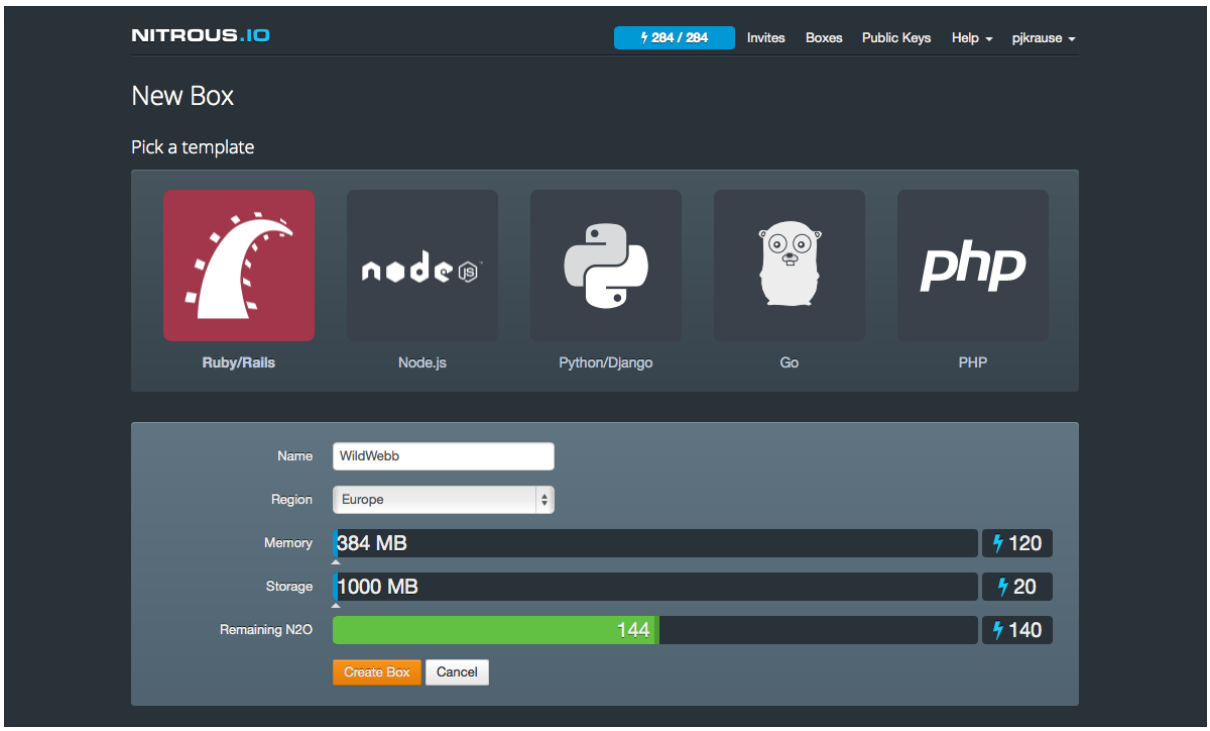
Creating a Box

Nitrous supports many different programming languages. We are going to work with Ruby and Rails. Even more exciting, we are going to learn how to build web applications using Rails.

Once you have signed up and logged in, you will see a screen like the one on the next page (you may need to scroll down to the bottom of the landing page and click "Start Coding" first):



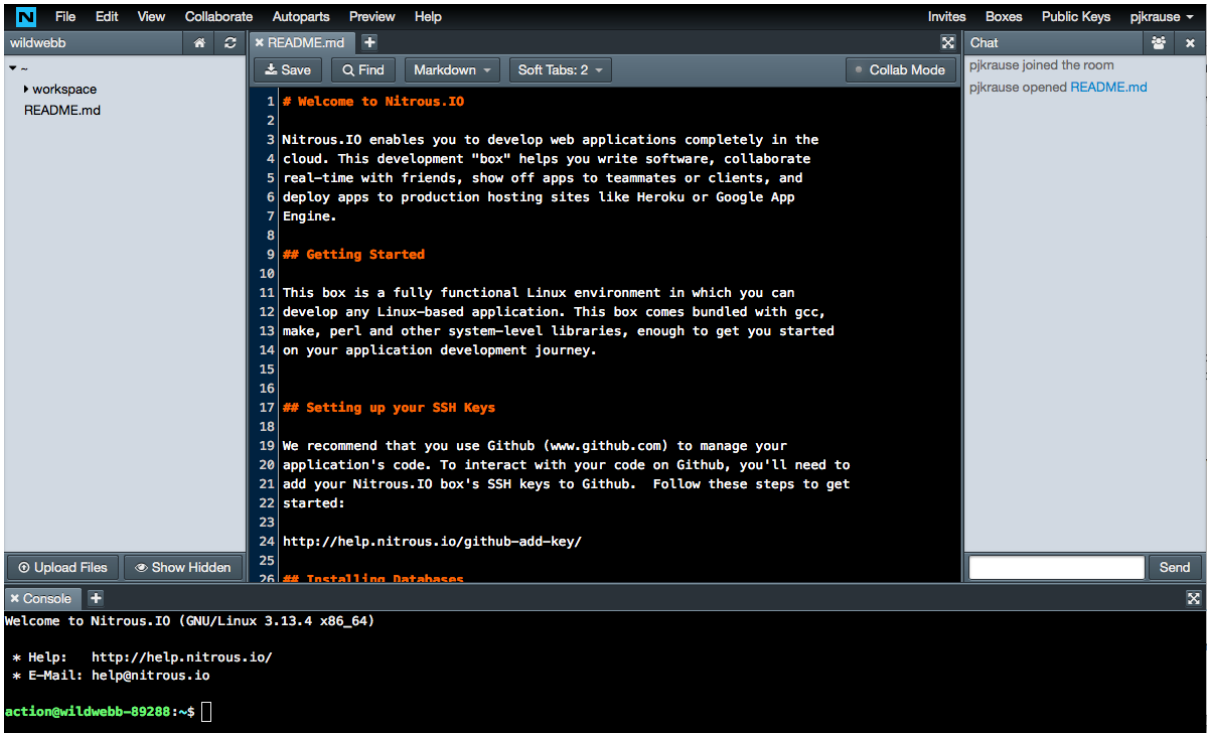
Your free account will give you enough Nitrous to create just one box, but this will be sufficient for this course. So go ahead and click on "New Box"!



You can give your box a name. You can also select the region you live in so Nitrous will be running on a server near to you. But most of

all, make sure you click on the BIG Ruby/Rails button so that Nitrous knows to set up all the things you need!

Once you have done that, hit “Create Box” and you will see this:



How cool is that! Nitrous has set you up with a complete desktop in your browser with a finder space on the left side where you can see all your files and folders (although we only have one folder called “workspace” just now), a big area for editing your code, a console along the bottom where you can run your code and do all sorts of exciting things. There is even a chat room where you can talk to your friends while you code!

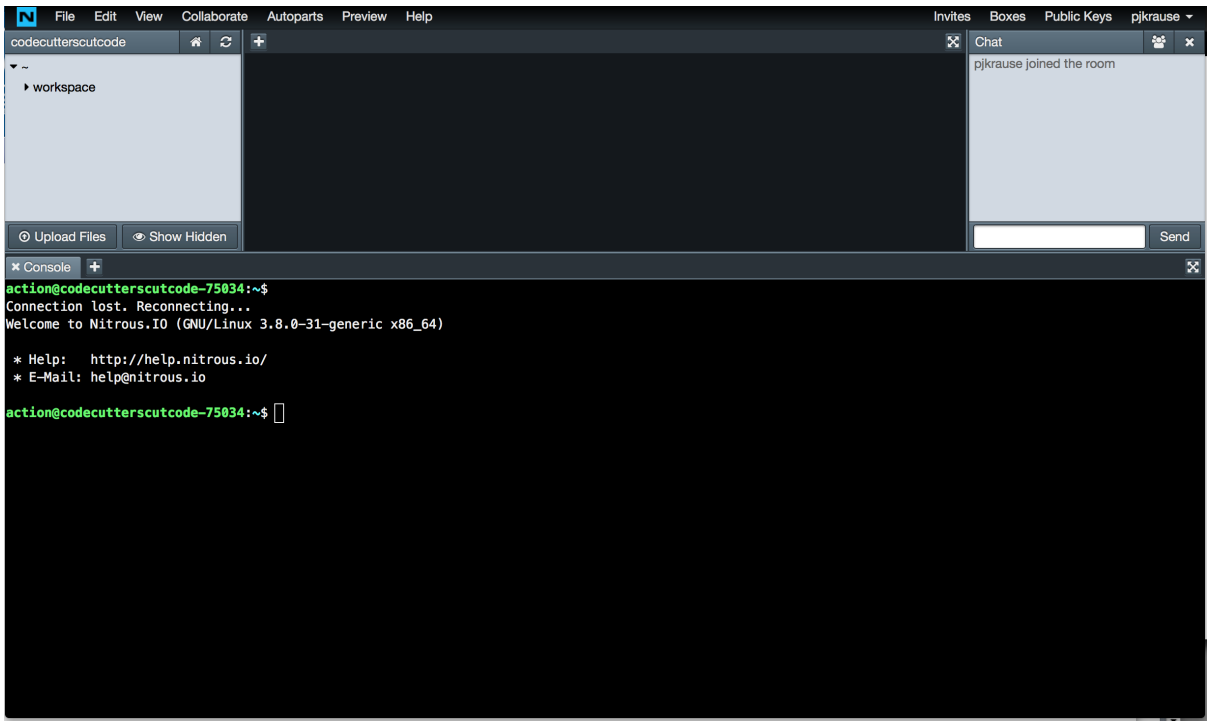
There’s loads of stuff in there and you don’t need to do a thing to keep it all up to date - Nitrous will do that for you. This is one of the BIG advantages of “working in the cloud”.

Working at the Console

Linux is there to help you

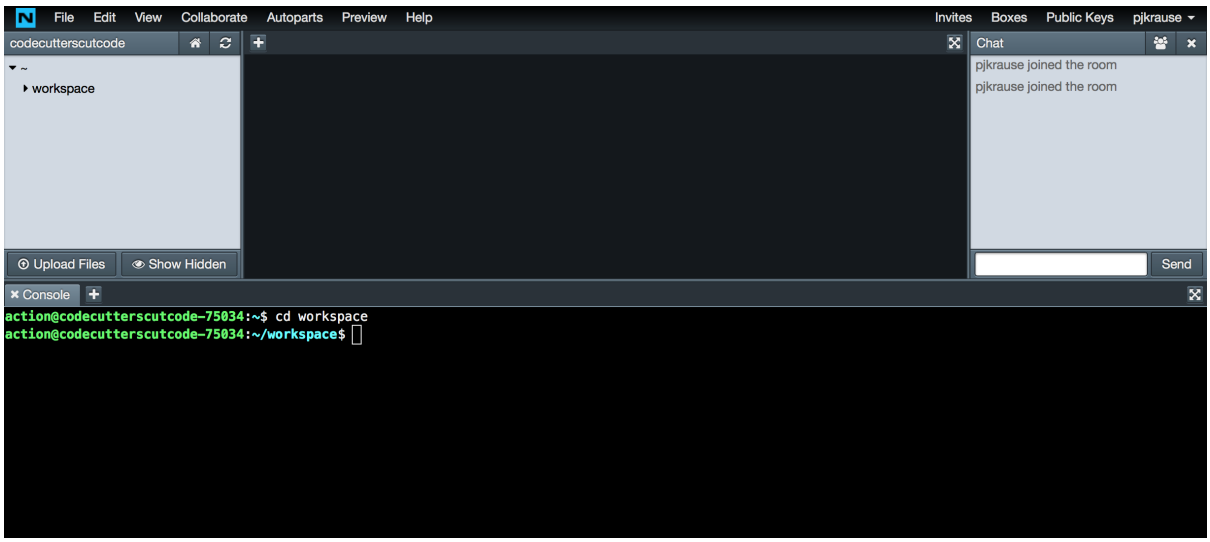
Any computer program needs to run inside an operating system. That operating system acts as an environment for the program to live in, so that it can communicate with files, printers, mouse and keyboard, for example. If we are aiming to build a web application, then we really need an operating system that is good at multi-tasking. There are lots of things that it needs to do at the same time - it might be saving some date to a file file for one user, while doing some sums for another, and waiting for a third one to fill in a form. Unix is a very stable multi-tasking operating system that was developed by some talented computer scientists at Bell Labs in the 1960s. Unfortunately, it is a bit expensive. So instead, we will use Linux, which is an open source version of Unix. The kernel of Linux was first released in 1991 by Linus Torvalds, and it has developed since then into an operating system that is very widely used by the web application development and deployment community.

The Console at the bottom of your Nitrous desktop is your window into the Linux operating system. These is where you type in commands to get Linux to do useful things for you. We can make the window a bit bigger, which is worth doing for this chapter and the next as we are not going to use the editor for a little while. Just move the cursor so that it is over the line that divides the editor window from the console and a double sided arrow will appear. Then click on the line and drag it upwards so that your Nitrous desktop looks something like this:



Computers try to hide low-level commands for everyday use. However, if you are developing programs then you really need to learn to talk directly to the operating system through the console. It's a little bit harder to get started as you need to learn some commands instead of just clicking on things, but you soon start to remember the commands you need.

Let's try one or two. Nitrous will have started the console session in the "root" folder, or "directory". As you can see from the finder window, you only have one other directory (I am going to stick with the term "directory" instead of folder from now onwards as this is the terminology of professionals!). This is called "workspace" and we will soon be putting some files into it. So, let's move the console to that folder. You use the command "cd" for changing directory (fairly obvious really!). So type this at the console and press the "return" key. You will see that the prompt in the console reminds you of the directory that you have just moved to (see the picture on the next page).



Is you add in new directories, you gradually build up a tree-like structure starting from the "root". Actually, it is a bit like an upside-down tree as you will find you think about moving down the tree from root to a sub-directory like "workspace". You can always move up a level by typing:

$ cd ..

(The dollar sign represents the prompt in the console so you don't actually type that - it is already there). Try it now, and you will end up back in the root directory.

I'll let you into another little secret - Linux can recognise some of the words you might use without you need to type them. This is known as "tab autocompletion". So, for example, in the root directory there is only one sub-directory and this starts with a "w" (for workspace). So if you now type at the prompt

$ cd w

and before pressing the return key, you press the tab key, Linux will complete this for you so you will see:

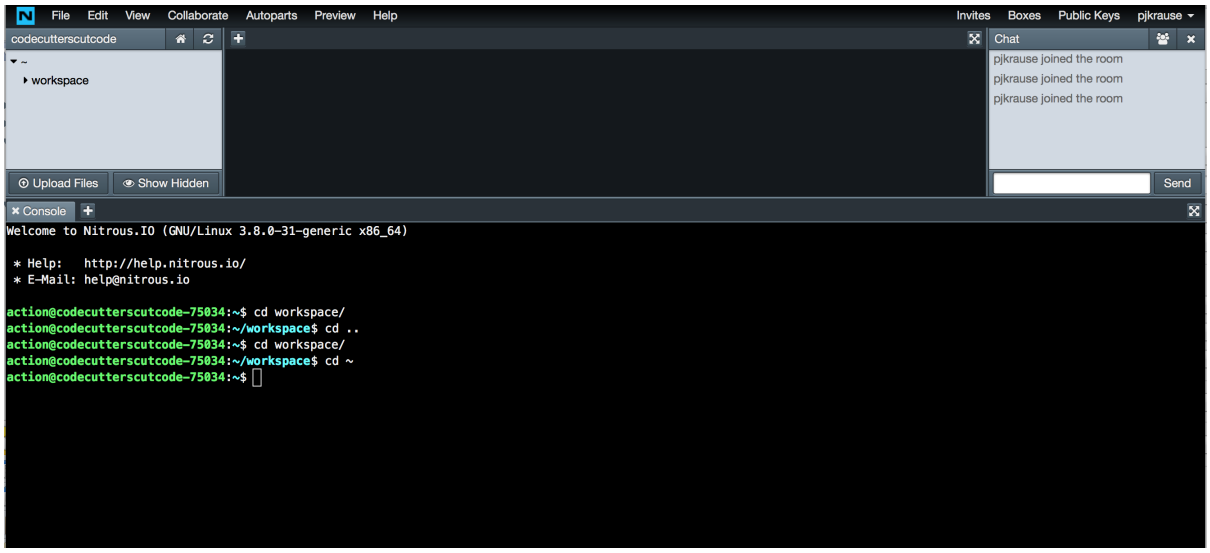
$ cd workspace

You can then press “return” and the directory will change.

You will find that as time goes by, your directory structure will build up to become quite complex. Fortunately, there is a quick way “home”. As you may have seen from the command prompt, Linux recognises “~” as a shorthand for the root directory. If you type

$ cd ~

from any position in your directory structure, Linux will take you back to the root directory. Try it now.



After all that, you should see a trail of commands rather like the above picture. That’s all for Linux for now. Let’s do a little programming.

RVM

Ruby Version Manager

The Ruby Version Manager (RVM) is a tool for managing, installing and working with multiple Ruby environments. It will work on any *-nix operating system. You can find out all about it at: https://rvm.io

If you are a Windows fan and at some point wanting to move to managing Ruby on your local machine then try Pik (https://www.ruby-toolbox.com/projects/pik)

For some reason, Nitrous do not preinstall RVM. The easiest way to do this is to enter the following into the console:

$ \curl -L https://get.rvm.io | bash -s stable --auto-dotfiles --ruby

(Don't type the "$" of course, that is there already!)

The "—ruby" flag will ensure that the latest version of Ruby is also installed (2.1.3 at time of writing).

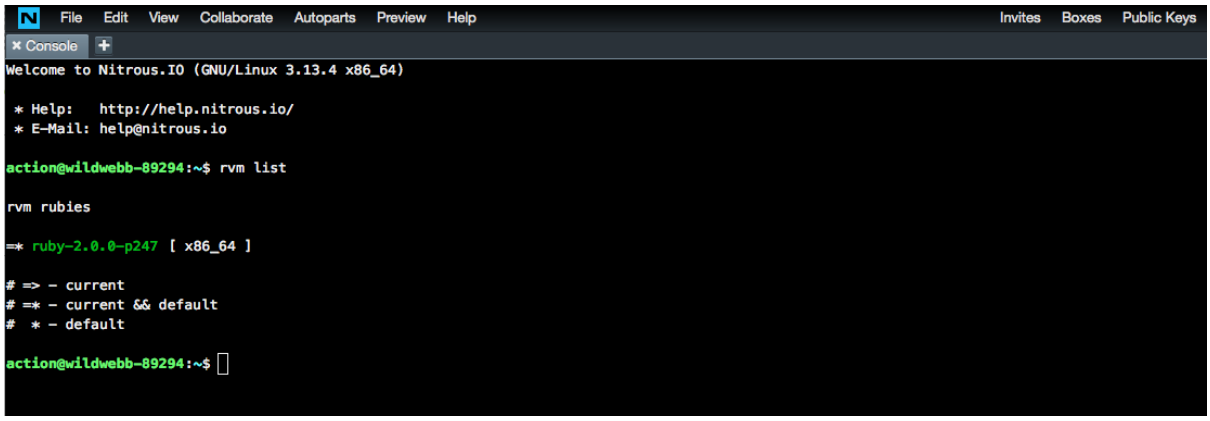
You will also need to activate RVM, so enter this once the installation is complete:

$ source /home/action/.rvm/scripts/rvm

Now you can check that RVM is installed:

$ rvm -v

Lets now start using it just to get a feel for managing the very basics of your Ruby environment. The first thing to do is to list the Rubies that are currently installed:



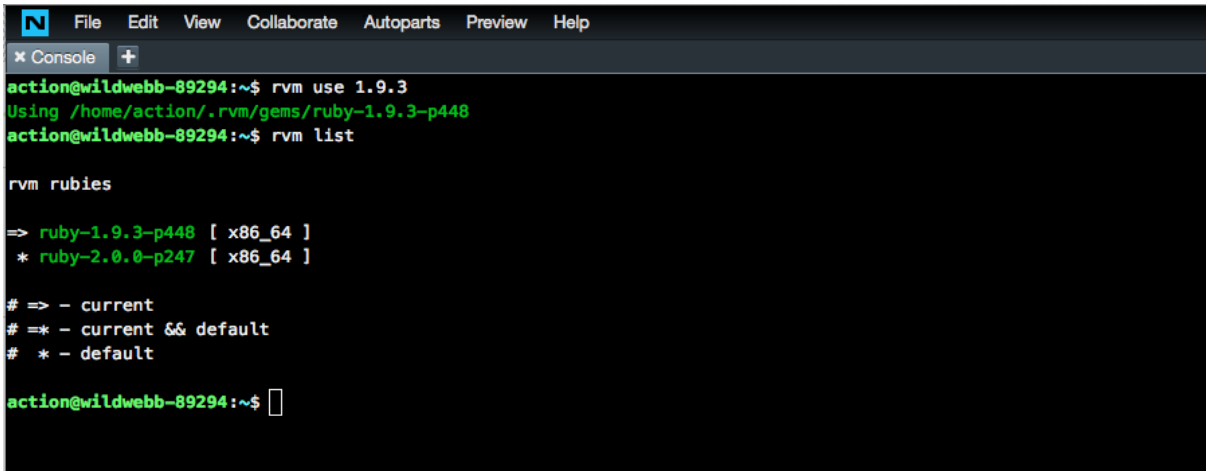
Note that the editor is not visible. I clicked the icon in the top right hand side of the console to toggle it into full screen as we are not going to use the editor for a while.

You see that when I typed the command “rvm list” I found out that version 2.0.0 was the only one installed and that it was the current and default Ruby (of course). This image was taken a while ago so you should see 2.1.3 listed instead.

Now suppose I want to work on a project that was started before Ruby 2 was released. I will need to use the earlier version of Ruby. Let’s first use RVM to install that earlier version (1.9.3 in this hypothetical scenario).

As you see, all I have to do is type “rvm install 1.9.3” at the console and the RVM gets on with the job (do this now). Once that succeeds, if we type “rvm list” we have some more info.

We are not there yet, though. As you can see, we need to send another command to change the current Ruby version.



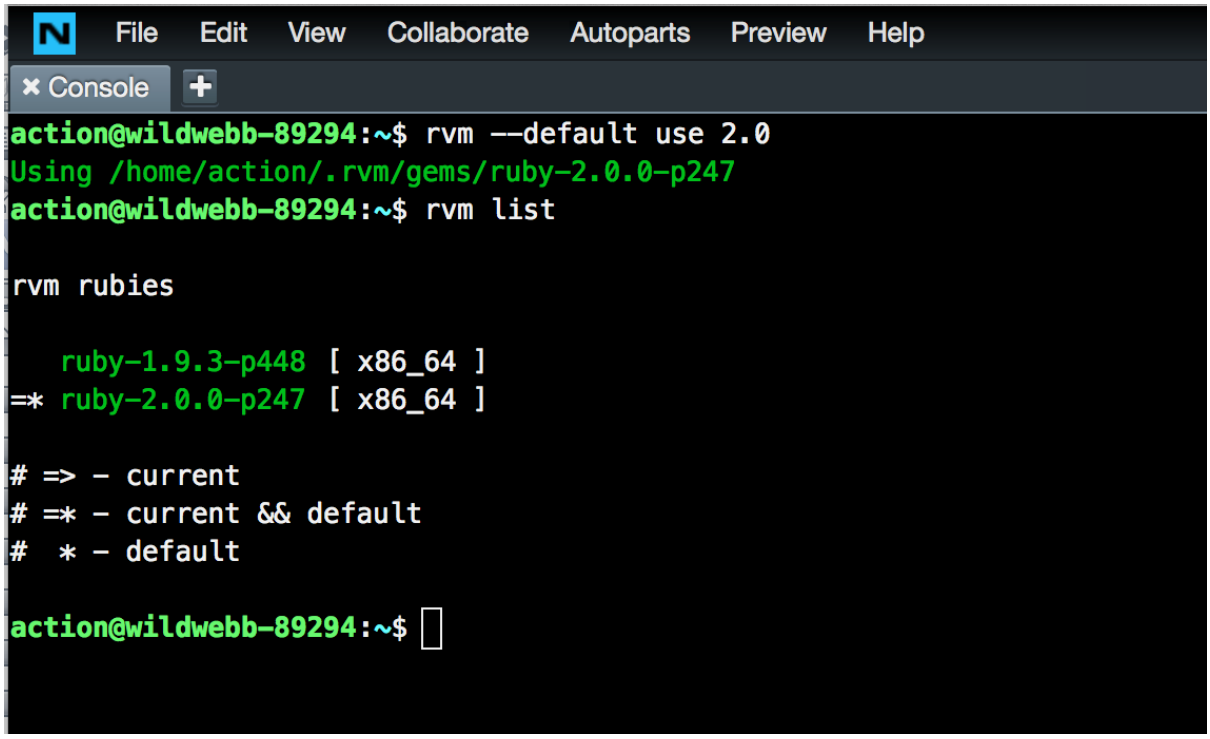
Now if we list the Ruby versions again, we will see that 1.9.3 is the current version (but not the default):

This means that Ruby 1.9.3 will only be used for the duration of the current console session. When I login again, we will be back to Ruby 2. That's a bit risky if I am going to be working on this old project for a while, so lets change 1.9.3 to the default Ruby:

What I did here was to set the "–default" flag for the "use" action (there are two minus signs before the word "default"), and then enter "rvm list" to show the status of my Rubies.

That is all for now, although there is much, much more that can be done with the RVM. We will come back to it later but for now let's

just reset everything back to Ruby 2 because that is the version we are going to stick with for the time being (Note again, you will have Ruby 2.1.3, not 2.0.0):



Updating Rails

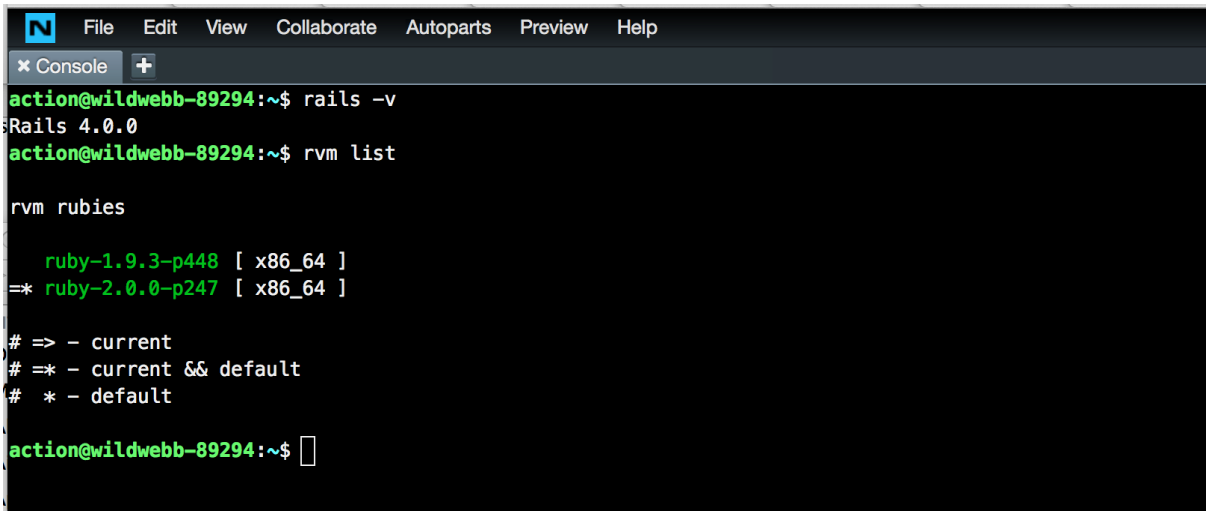
Getting Ready to Start

N.B. If you have followed the previous chapter, you may need to install the Rails gem, but that is all:

$ gem install rails

The remainder of this chapter is only of value if you are starting up a new project some time after the latest versions of Ruby and Rails were installed.

It is always worth making sure that you have the latest versions of Ruby and of Rails when you are starting a new project. This chapter provides an easy recipe for doing just that. Let use first take a look at the environment that Nitrous provides you (as of 7 March 2014) when you start up a new Rails box:



As you will recall from the last chapter, Ruby version 2.0.0 is set as the default. In addition, I have checked the Rails version ("rails -v") and you will see that version 4.0.0 is installed.

Now, the Rails team recommend usage of Ruby 2.1. It provides some performance advantages over 2.0 so it makes sense to use it with Rails. In addition, the current stable release of Rails is 4.1.6. This contains some important security fixes so it is important to use it.

As before, we will use the RVM to install this updated version of Ruby. However, RVM itself needs to be updated to make sure that it is a version that "knows all about" the latest version of Ruby. We do that with "rvm get stable":

```
action@wildwebb-89294:~$ rvm get stable
Downloading https://get.rvm.io
Downloading https://github.com/wayneeseguin/rvm/archive/stable.tar.gz

Upgrading the RVM installation in /home/action/.rvm/
    RVM PATH line found in /home/action/.bashrc /home/action/.zshrc.
    RVM sourcing line found in /home/action/.bash_profile /home/action/.zprofile.
Upgrade of RVM in /home/action/.rvm/ is complete.

# action,
#
#   Thank you for using RVM!
#   We sincerely hope that RVM helps to make your life easier and more enjoyable!!!
#
# ~Wayne, Michal & team.

In case of problems: http://rvm.io/help and https://twitter.com/rvm_io

Upgrade Notes:

  * No new notes to display.

RVM reloaded!
action@wildwebb-89294:~$
```

As it happens, I had already got an updated version of the RVM installed, but it you have just started with a new Nitrous box the chances are it will need to download and install an update. So you will see more messages than I have shown here while it does the necessary.

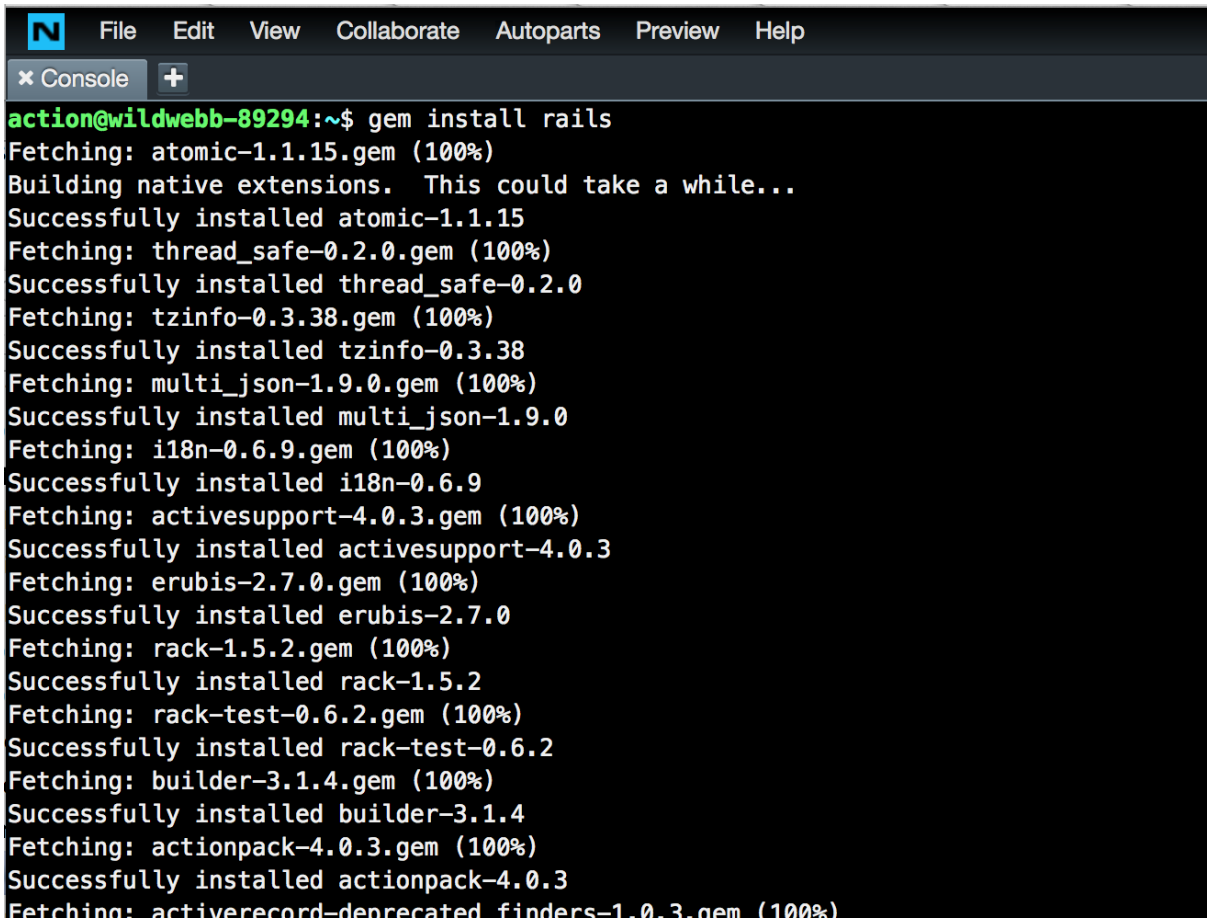
Now we can use the RVM to install the latest Ruby:

```
action@wildwebb-89294:~$ rvm install 2.1.1
Searching for binary rubies, this might take some time.
No binary rubies available for: ubuntu/12.04/x86_64/ruby-2.1.1.
Continuing with compilation. Please read 'rvm help mount' to get more information on binary rubies.
Checking requirements for ubuntu.
Requirements installation successful.
Installing Ruby from source to: /home/action/.rvm/rubies/ruby-2.1.1, this may take a while depending on your cpu(s)...
ruby-2.1.1 - #downloading ruby-2.1.1, this may take a while depending on your connection...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 11.4M  100 11.4M    0     0  23.3M      0 --:--:-- --:--:-- --:--:-- 26.3M
ruby-2.1.1 - #extracting ruby-2.1.1 to /home/action/.rvm/src/ruby-2.1.1.
ruby-2.1.1 - #configuring.....................................................
ruby-2.1.1 - #post-configuration.
ruby-2.1.1 - #compiling.......................................................................
ruby-2.1.1 - #installing.....................................
ruby-2.1.1 - #making binaries executable.
Rubygems 2.2.2 already installed, skipping installation, use --force to reinstall.
ruby-2.1.1 - #gemset created /home/action/.rvm/gems/ruby-2.1.1@global
ruby-2.1.1 - #importing gemset /home/action/.rvm/gemsets/global.gems.....
ruby-2.1.1 - #generating global wrappers.
ruby-2.1.1 - #gemset created /home/action/.rvm/gems/ruby-2.1.1
ruby-2.1.1 - #importing gemsetfile /home/action/.rvm/gemsets/default.gems evaluated to empty gem list
ruby-2.1.1 - #generating default wrappers.
ruby-2.1.1 - #adjusting #shebangs for (gem irb erb ri rdoc testrb rake).
Install of ruby-2.1.1 - #complete
Ruby was built without documentation, to build it run: rvm docs generate-ri
action@wildwebb-89294:~$
```

It takes a little while (but it does tell you what it is doing), so now is a good time to make a cup of coffee…

Now that is done, it is a good time to install the latest version of Rails. Rails is prepared as a Ruby "gem". Gems are Ruby components that contain pre-packaged chunks of functionality. This is what makes coding in Ruby and Rails so cool - there is massive of prepackaged stuff available. With Ruby installed we can just ask for the latest version of Rails to be installed as a gem. All the gems that Rails itself depends on will also be installed.

As you will see on the next page, all we need to do is enter the command "gem install rails". This will automatically download the most recent stable release.

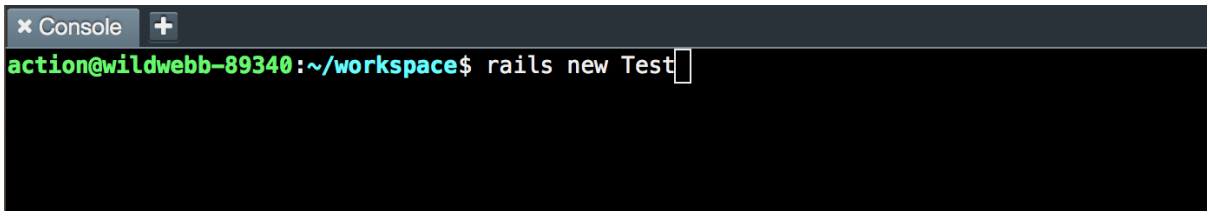


Many more dependent gems than you see here will be downloaded (including the Rails gem itself).

And now we are set to go. Remember that these settings only apply to the current Nitrous box. If you (terminate this one and) start a new one, you will have to do the same updates.

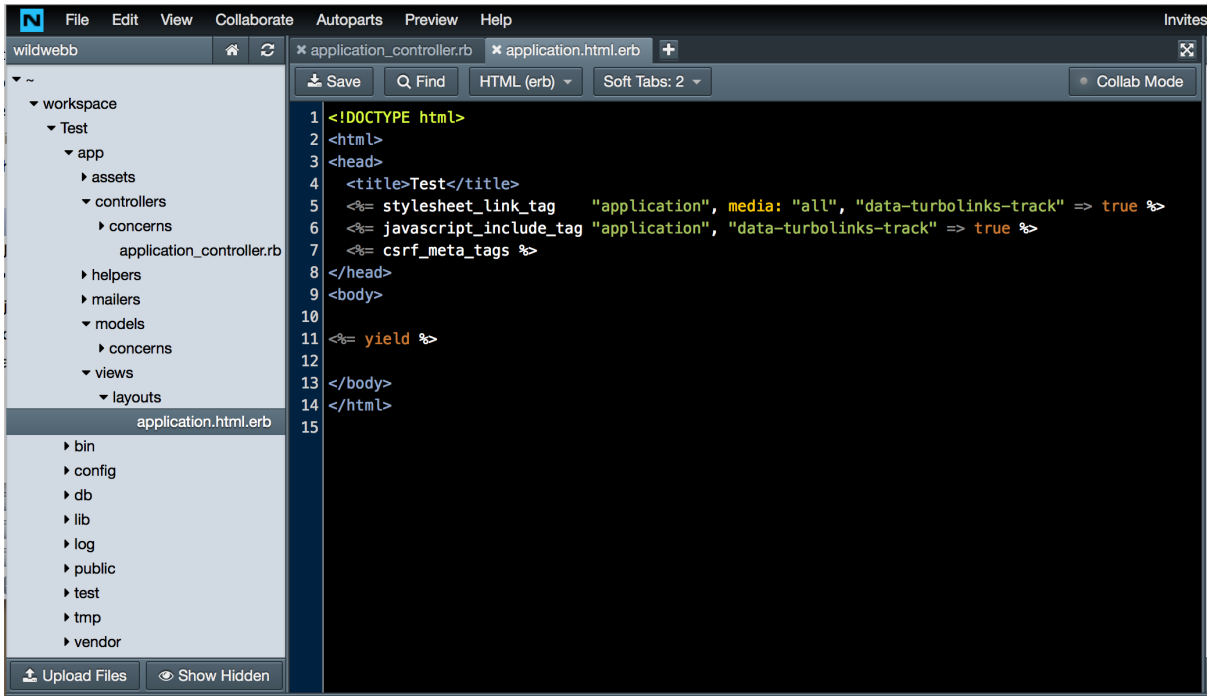
Working with Nitrous

Before we dive into Ruby and Rails, it is worth exploring Nitrous.io a little more. I don't want to say too much about Rails in this chapter. However, just so that we have something to look at in Nitrous, let's create a new Rails project. Focus on the console and change directory into the workspace. Then you can create a new project by entering the following command:



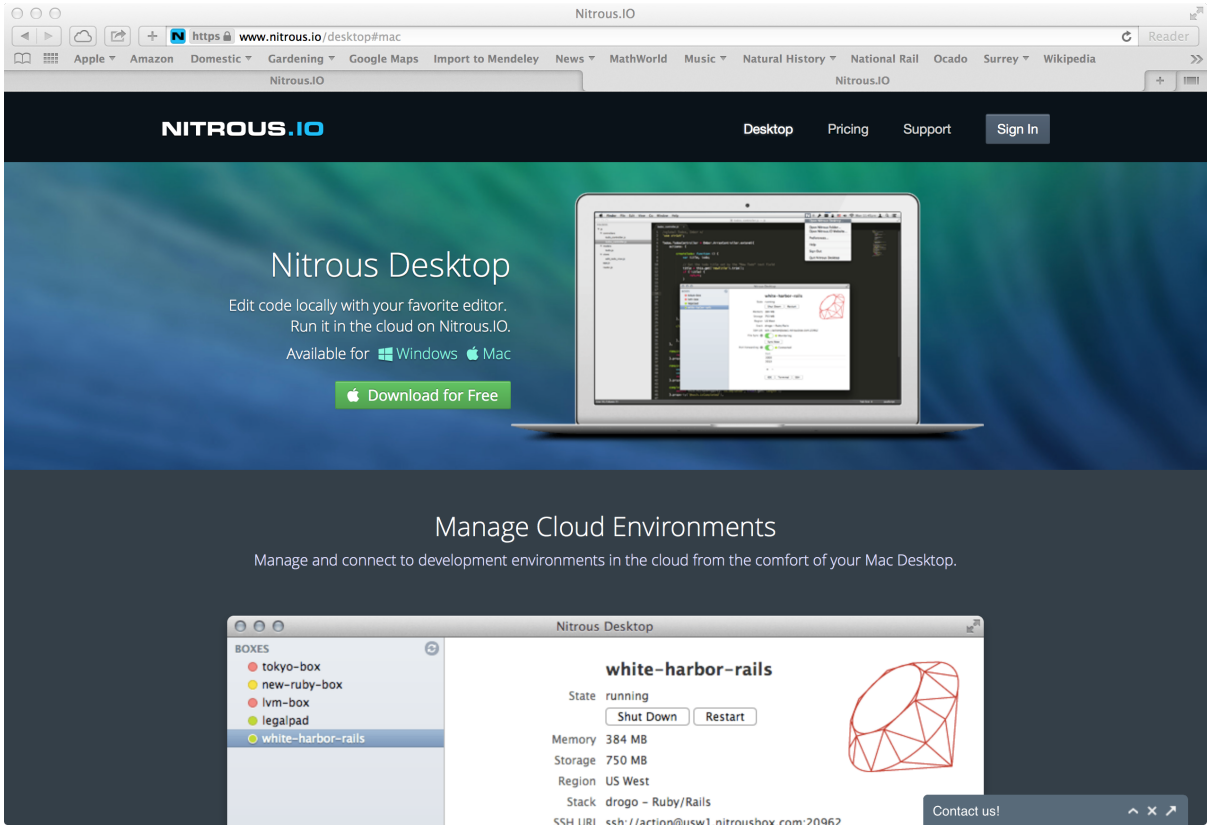
Rails will generate a "skeleton project" for you with lots of files and folders. You will see this in the finder window:

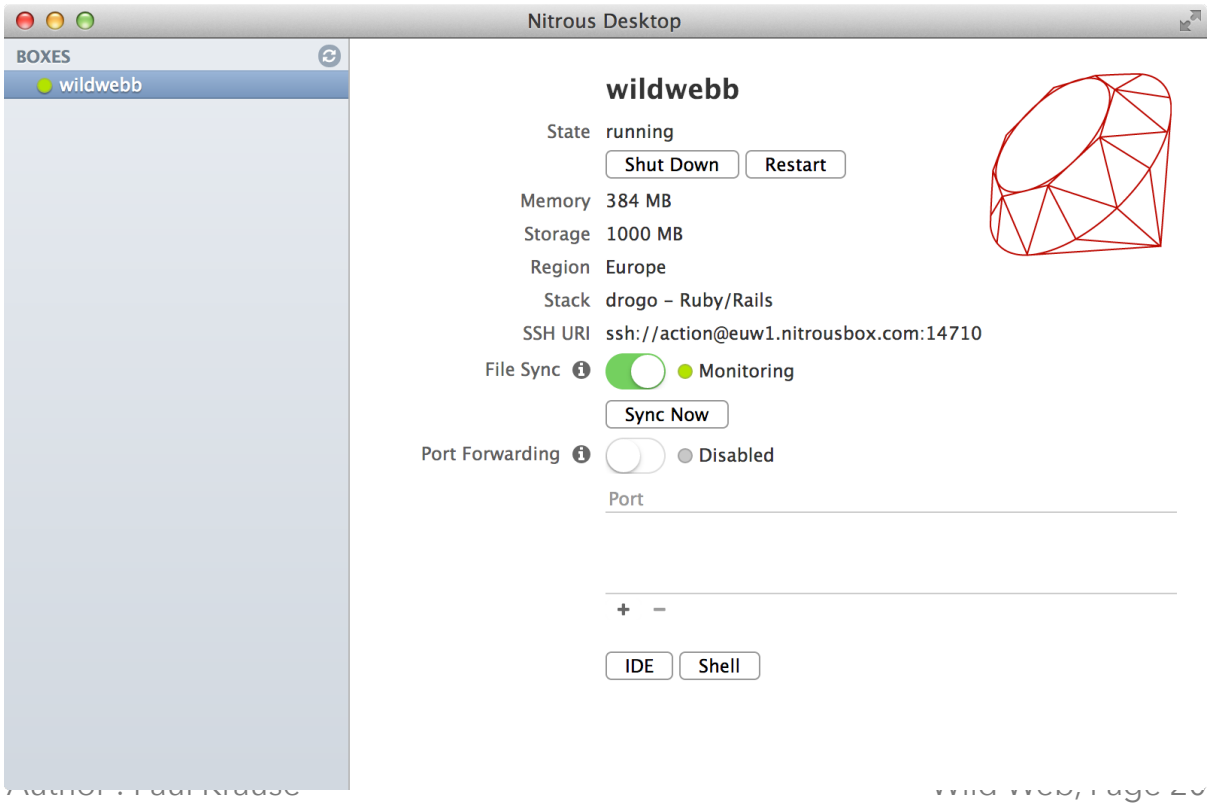
Just to see some code, you can click on the downward facing arrows in the finder window to open the “Test/app/views/layouts” folder and then double click the “application.html.erb” file to open it in the editor. You will see this:



Now there is something that might start to worry you. As you start to build out your project, all your precious code will be stored on Nitrous.io’s own servers. What if you want to keep a copy of it on your own machine? You should do this for safety. Indeed, we will see two ways of exporting your project from Nitrous. Both are important, but one thing at a time and let’s get a copy of the code onto your own machine first.

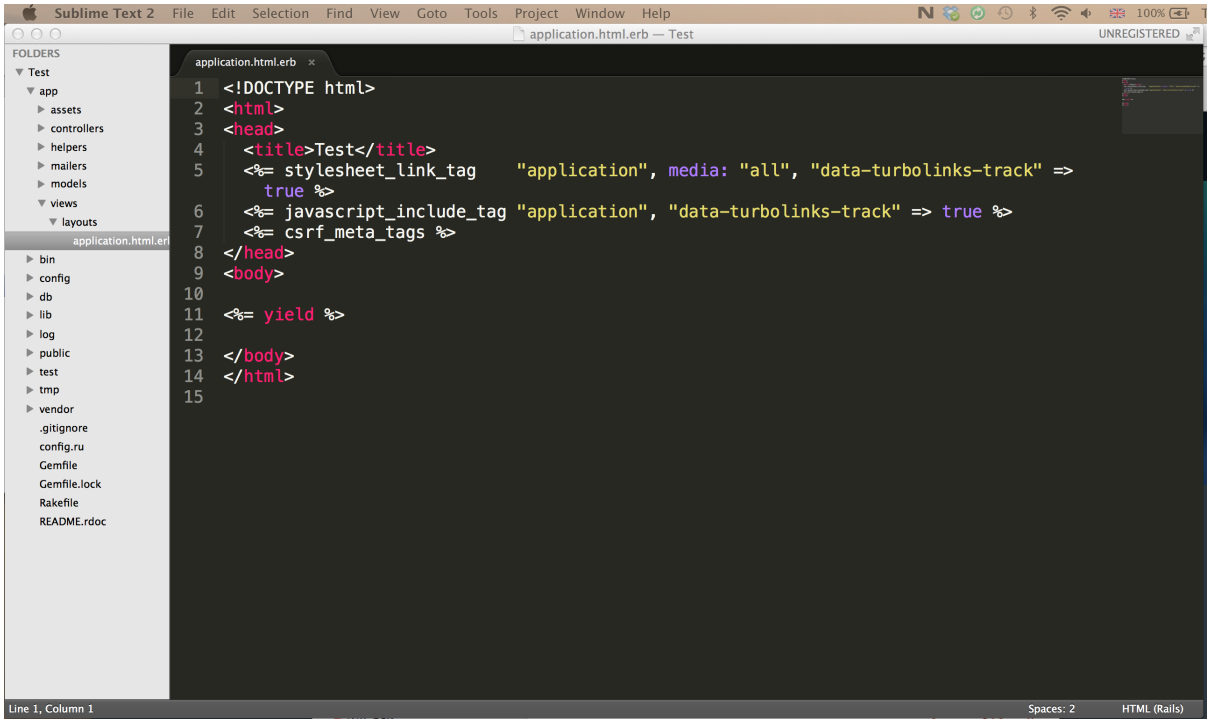
If you go to “Nitrous.io/desktop” you will see that you can download the Nitrous desktop for either Windows or Mac. Do that now, and install it. (N.B. the desktop is only useful if you are using your personal computer - don’t worry about this in the University labs.)


Nitrous.IO
NITROUS.IO
Desktop
Pricing
Support
Sign In
Nitrous Desktop
Edit code locally with your favorite editor.
Run it in the cloud on Nitrous.IO.
Available for Windows Mac
Download for Free
Manage Cloud Environments
Manage and connect to development environments in the cloud from the comfort of your Mac Desktop.
Nitrous Desktop
BOXES
tokyo-box
new-ruby-box
lvm-box
legalpad
white-harbor-rails
white-harbor-rails
State running
Shut Down
Restart
Memory 384 MB
Storage 750 MB
Region US West
Stack drogo - Ruby/Rails
Contact us!

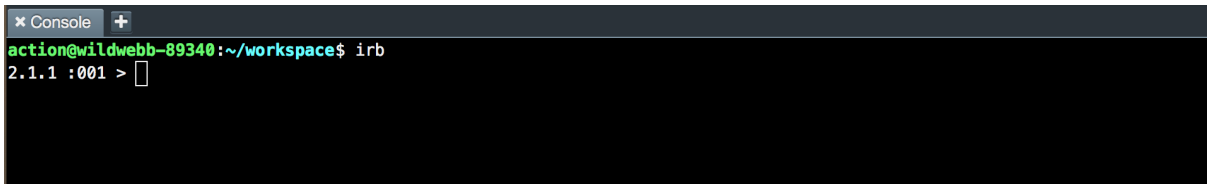

Nitrous Desktop
BOXES
wildwebb
wildwebb
State running
Shut Down
Restart
Memory 384 MB
Storage 1000 MB
Region Europe
Stack drogo - Ruby/Rails
SSH URI ssh://action@euw1.nitrousbox.com:14710
File Sync
Monitoring
Sync Now
Port Forwarding
Disabled
Port
IDE
Shell

If you open the desktop you will see a list of your current boxes on the left hand side (only one just now!), and details of the selected box in the larger pane on the right hand side:

Now all you need to do is to click on the “File Sync” button in the middle of the right hand pane to get the contents of your Nitrous box(es) synchronised with your own computer. These will be saved in a directory called “Nitrous” that will be created in your root directory. Now you have the choice of editing your project locally on your own



machine (I like to use Sublime Text 2), or editing it in the Nitrous editor. All changes will be synchronised between your own computer and Nitrous almost immediately so long as you have the Nitrous desktop

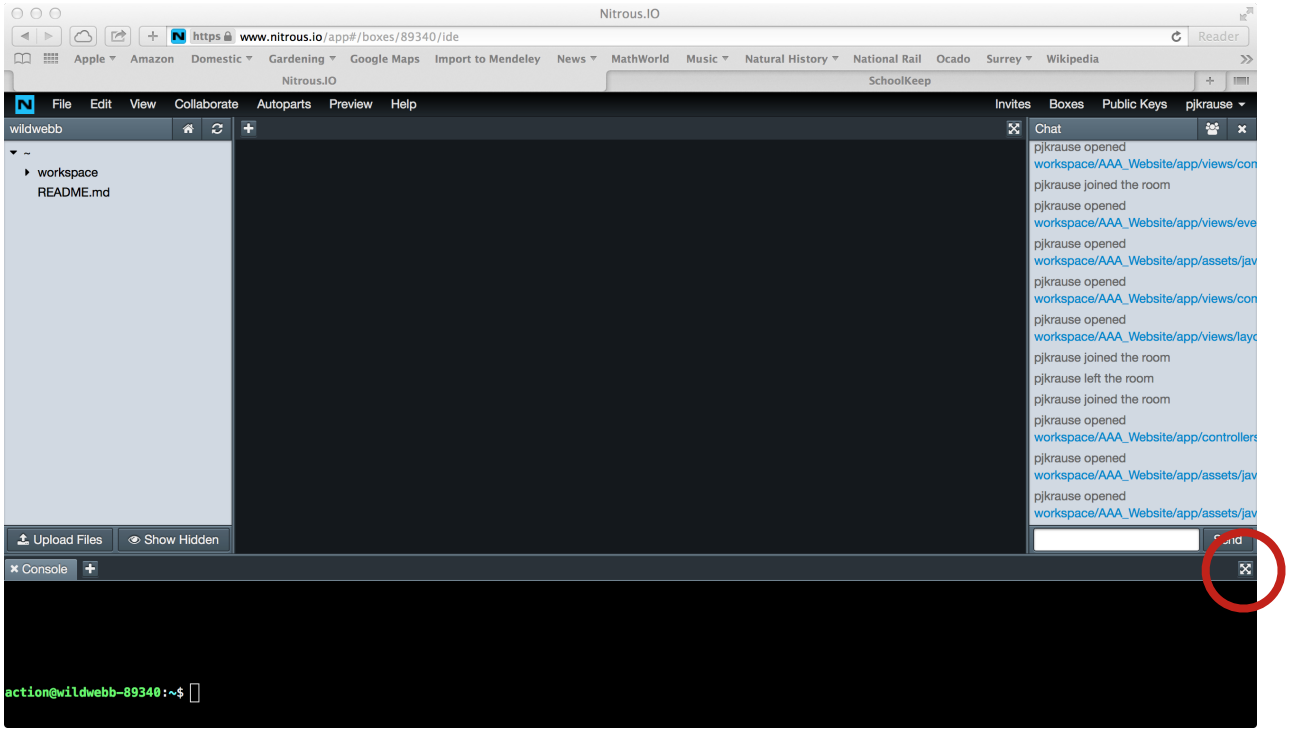


open and are connected to the internet.

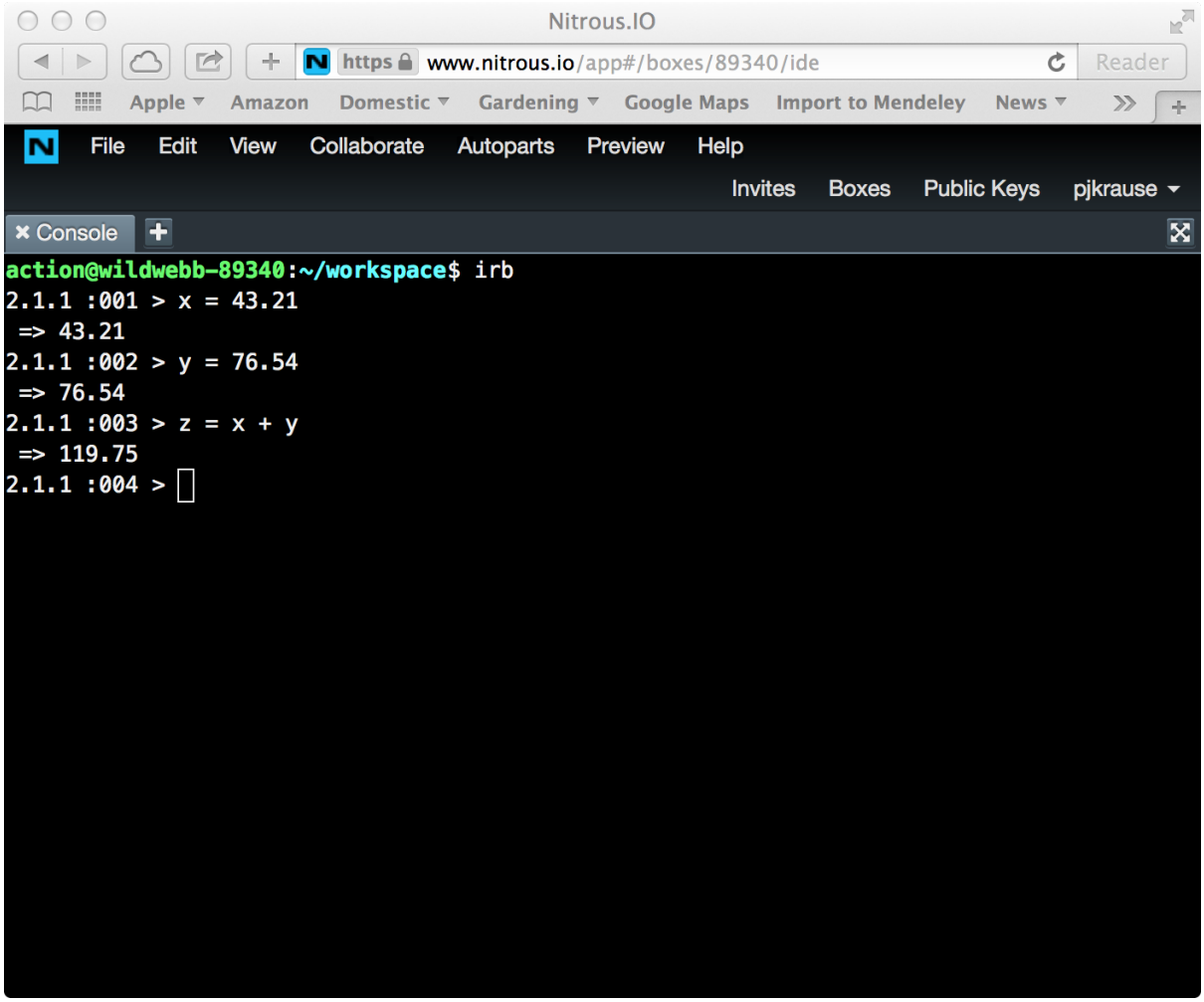
Now there is just one more little thing before we start looking at Rails. We will experiment with Ruby using the Interactive Ruby Console. You can open this in Nitrous by typing "irb" at the console:

Of course, if you do have Ruby installed on your own machine then you can open a terminal window and run the irb there. This is what I have done in many of the screenshots that follow in the next couple of chapters.

We are not going to use the editor for the moment so you might like to increase the size of the terminal window. You can do this by clicking the icon on the top right of the terminal window (circled in red below):

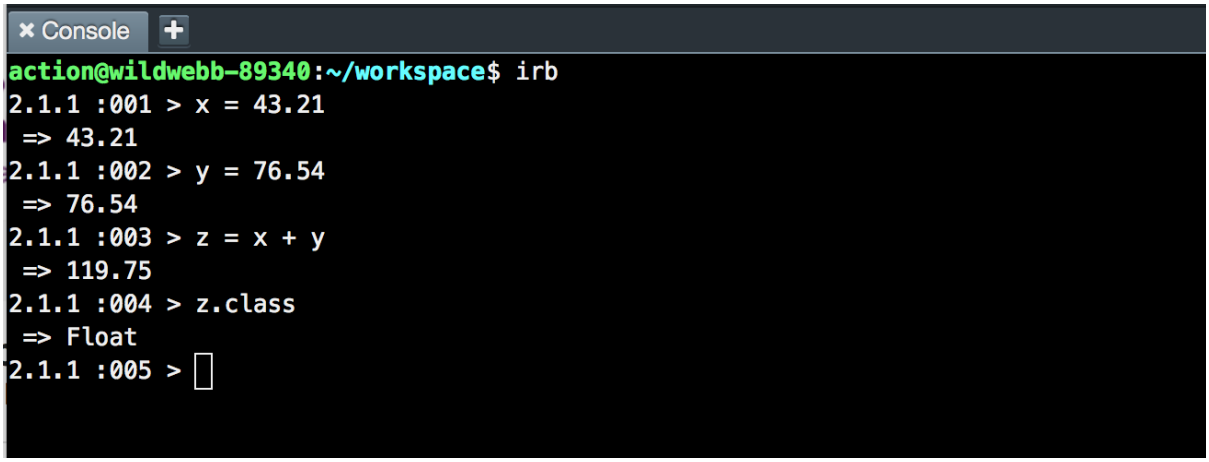


Ruby is an "interpreted language". This means you can run code straight away, so lets do something. Remember to have the "irb" running in the terminal as we had just now. Then we can easily add two numbers together by repeating what is in the illustration on the next page.



As well as being interpreted, Ruby is an "object-oriented language". This means that when we write code we can build "objects" from templates, or class descriptions. These objects have data values (the object "x" has a value of 43.21 in our example), and also have functions associated with them. These functions can be used to manipulate data within the object, or to ask the object something about itself. For example, we can ask any Ruby object what class it

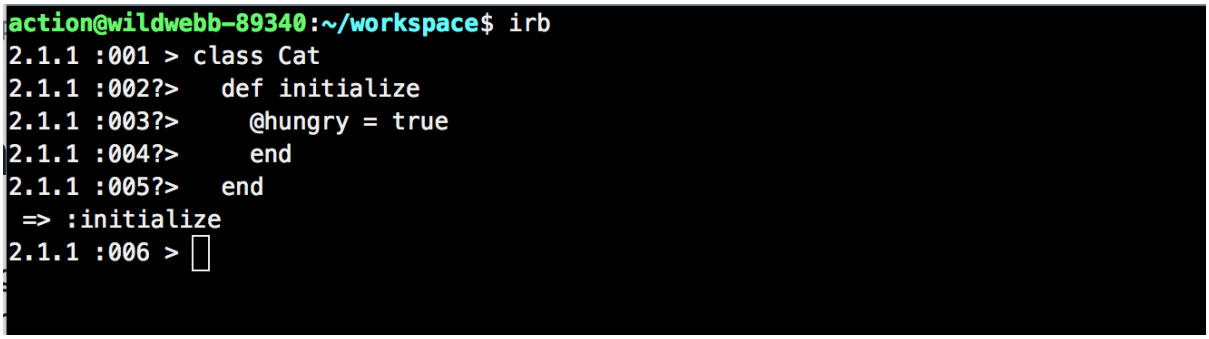
belongs to using the “class” function:



Notice the use of the “dot” notation where we invoke a function on an object: object.function.

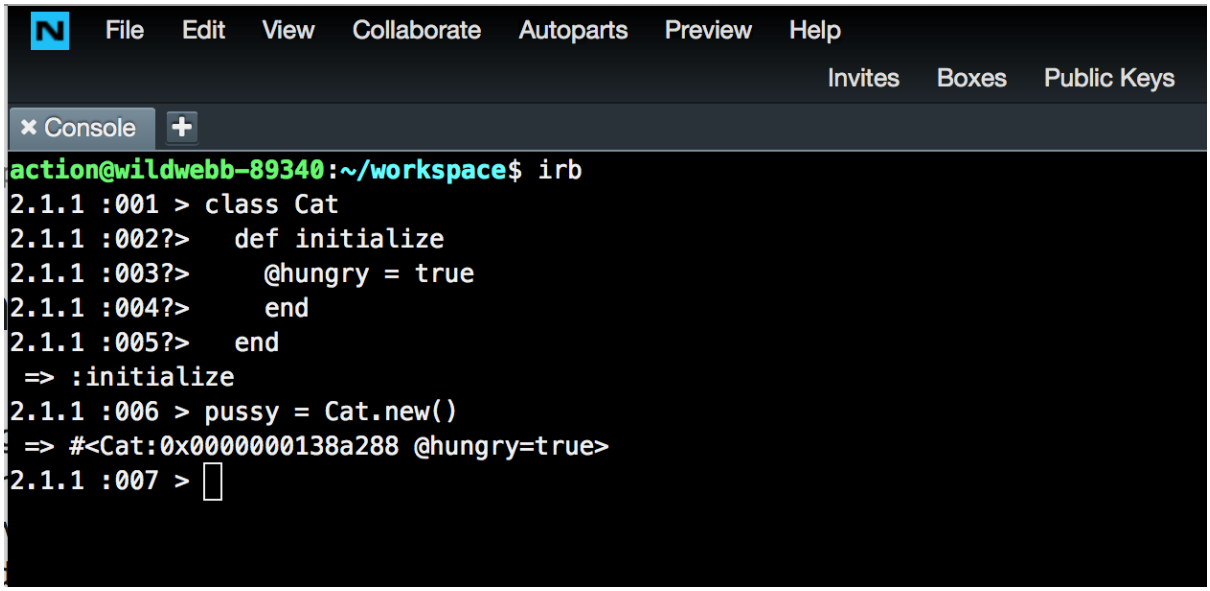
Some classes are built in, but we can also create our own classes. In this case, x, y and z are all members of the built-in class “Float”. This is shorthand for “floating point number”.

Let’s build a class of our own. We can do this within the irb, although the result won’t be saved anywhere. But this example is just a throw-away demonstration so there is no need to worry about that for now. In programs, classes are usually a representation of some real-world entity. We just focus on the properties of that entity that are interesting, and ignore the rest. In this example we are going to build a class called “Cat”, and all we are interested in is whether a cat is hungry or not. Just a reminder - you type everything that follows the “>” prompt, anything else is a response from the irb:



The “class” keyword tells Ruby that we are about to start the definition of a new class. Within the definition of a class, we always need to include a function “initialize” that tells Ruby the initial data values for any newly created object of that class. In this case, any new Cat will be born hungry!

Now we can go ahead and create a new Cat. This is an object, and we can refer to it by a name within our program. In this example, we have chosen to refer to it by the name (or “variable) “pussy”:



After we have created our new object (with “Cat.new()”), the irb replies to us with some data that says the object is of class “Cat”, has an internal identifier with lots of digits, and an “instance variable” @hungry that is set to true.

Now, we want to be able to feed our Cat. We do this by defining a new function called feed, that takes some data (the “food”) and does something to our Cat. If the food is “fish” the cat is not hungry any

more, otherwise it tells us that it doesn't recognise the food.

```
2.1.1 :079 > class Cat
2.1.1 :080?>   def feed(food)
2.1.1 :081?>     if food == "fish"
2.1.1 :082?>       puts "Yummy!"
2.1.1 :083?>       @hungry = false
2.1.1 :084?>       else
2.1.1 :085 >         puts "I am NOT eating that!!!"
2.1.1 :086?>       end
2.1.1 :087?>     end
2.1.1 :088?>   end
 => :feed
2.1.1 :089 >
```

By opening the class definition within the irb, the new function is added to the existing class definition.

Now we can feed the cat:

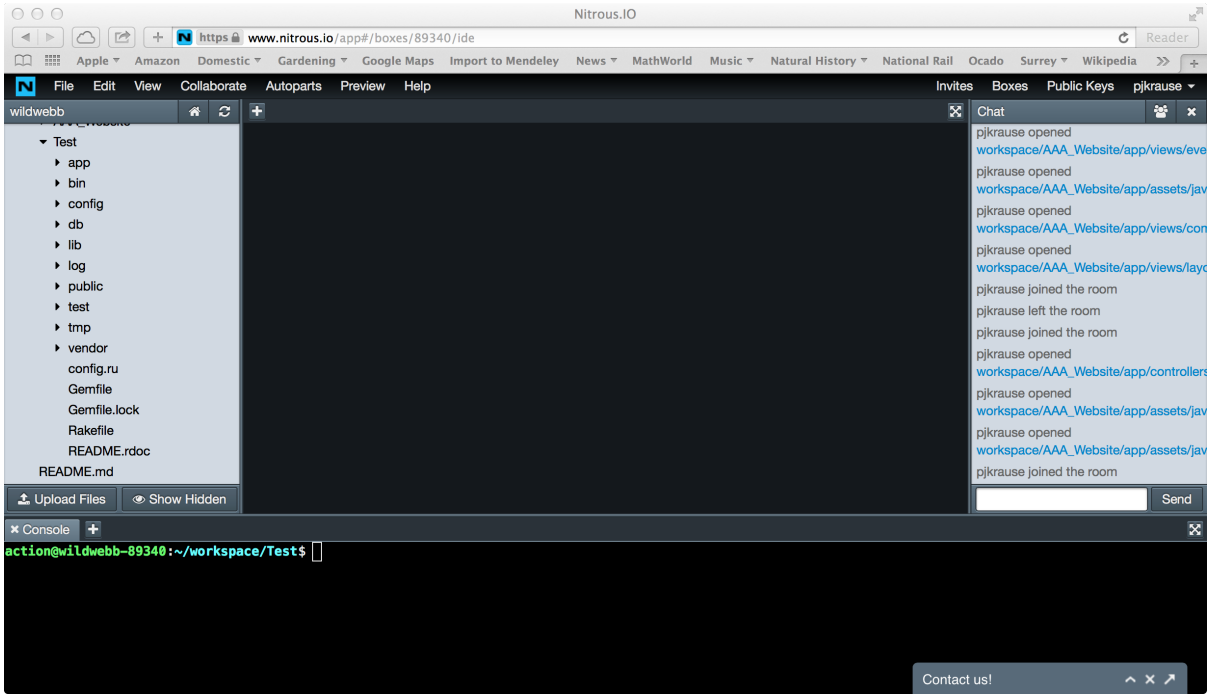
```
2.1.1 :089 > pussy.feed("fish")
Yummy!
 => false
2.1.1 :090 > pussy
 => #<Cat:0x0000000138a288 @hungry=false>
2.1.1 :091 > pussy.feed("cake")
I am NOT eating that!!!
 => nil
2.1.1 :092 >
```

You will notice that the “puts” function prints out the string that is next to it. Do also notice that there are two kinds of “equals”. The double “==” symbol means “check that the left hand side and the right hand side are the same”. For example, in the feed function, we want to check that the variable food is equal to the string “fish”. The single “=” symbol means “asign the value on the right hand side to the variable on the left hand side”. For example, in the same function if the preceding test is true, we want to assign the value of *false* to the instance variable *@hungry*.

That's it for a little bit of Ruby coding just now. Close the irb (use exit or quit), and let's get back to using the Rails framework.

My First Website

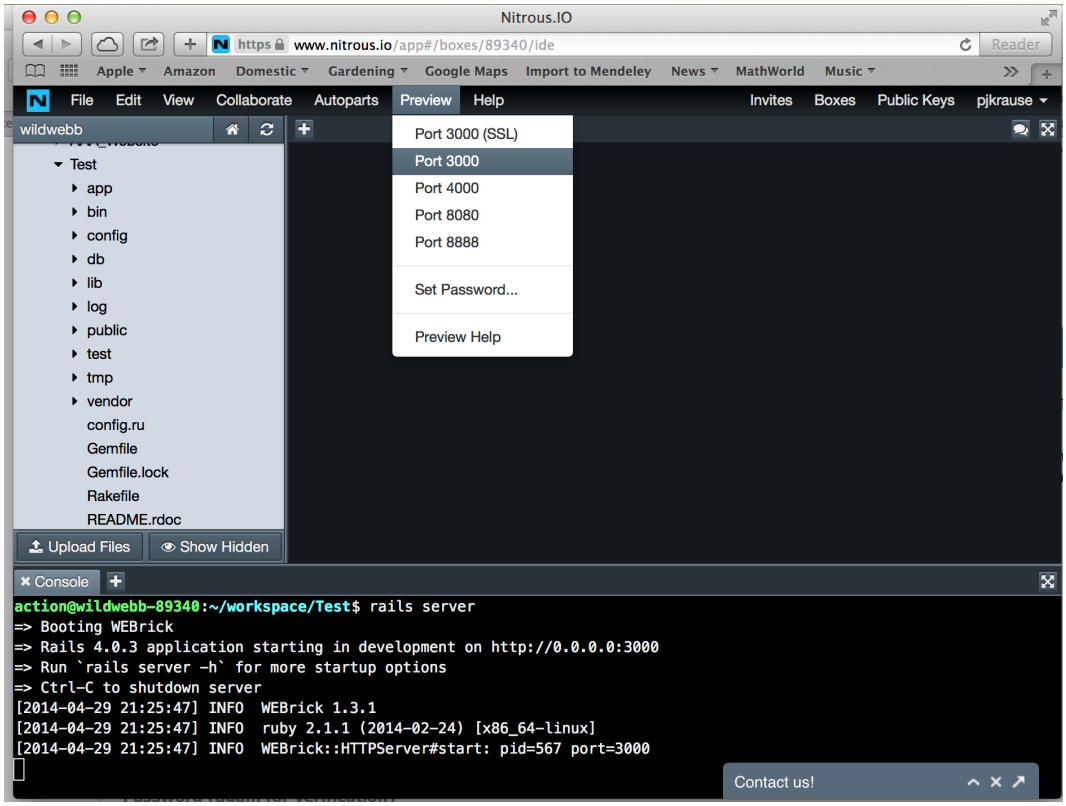
Actually, you have made one already! Remember that we generated some code with a single command. Click the icon on the top right hand side of the console again to reveal the finder window and editor once more. Open the Test folder in the finder and you will see all the code again:



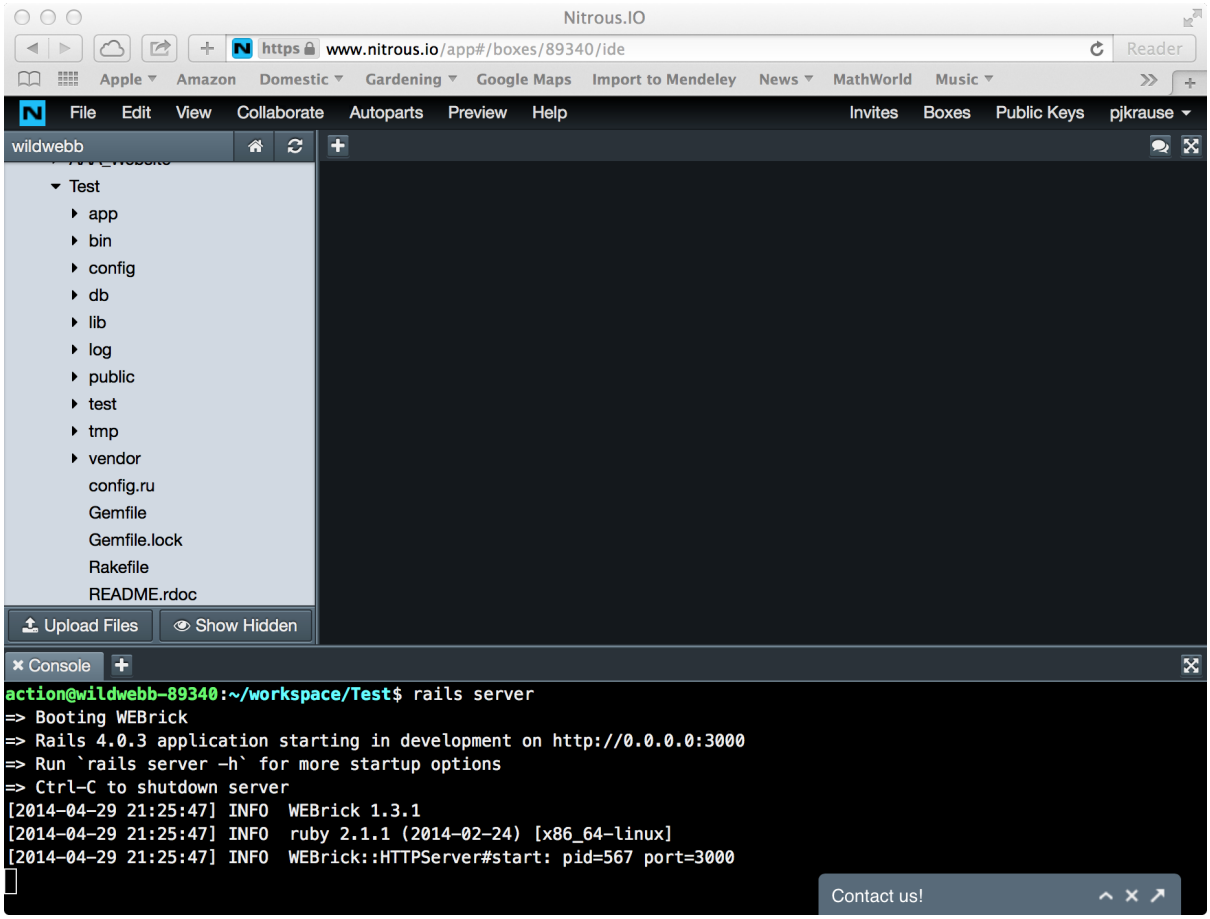
Make sure that the console is set to the root folder of the project (~/workspace/Test), as in the illustration above.

What we are going to do is to start up a "server" to publish our web page. A server is a program that receives requests from browsers that can be anywhere in the world and responds by sending back information that is displayed back in the browser window - usually in the form of an html file.

So let's do this!



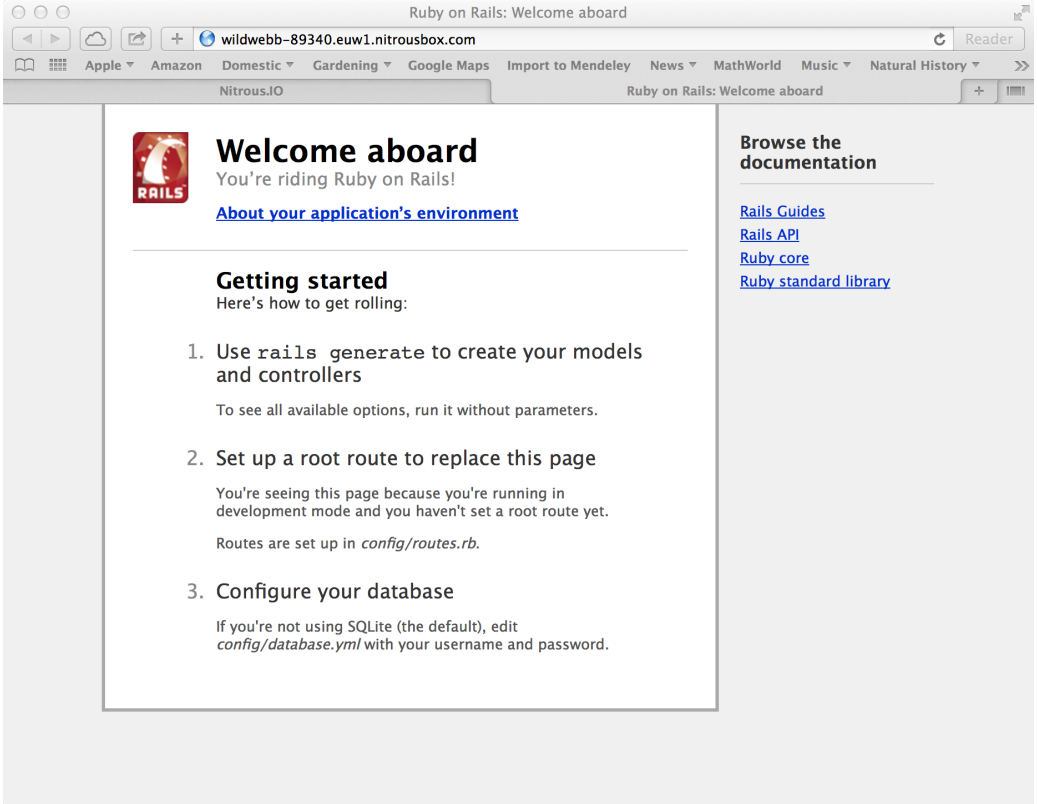
All you need to do is to type “rails server” in the console:



There are many different server programs. As you will see, the server we are using at the moment is “WEBrick”. If you see all the above messages, then it will be working fine.

Notice the reference to “port=3000”. This acts a bit like a socket for the world wide web to plug into, except that it is a software interface rather than a physical socket. Now the server is running, we can preview our website by “looking into” port 3000. You can do this by selecting “Preview/Port 3000” from the menu bar of Nitrous (see the next page).

Once you have done that, you will see a second tab appear with a welcome screen. This is the only page on your website but it's a start!



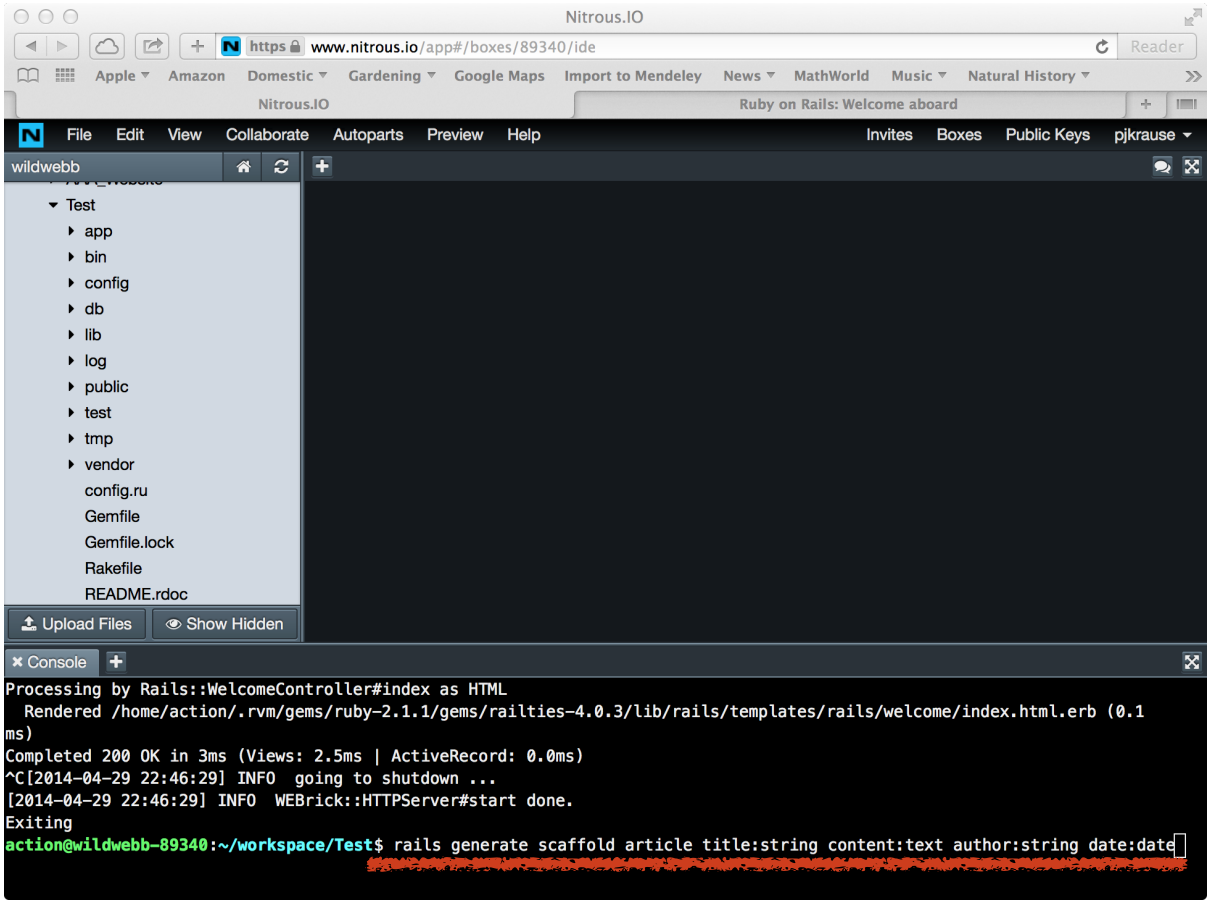
This really is live to the world. You can test this by selecting and copying the url in the address bar and e-mailing it to your friends. They will be able to see your website, so long as your Nitrous box is still running.

The trouble is, your website will look the same as everyone else's. So let's customise it.

My First Blog

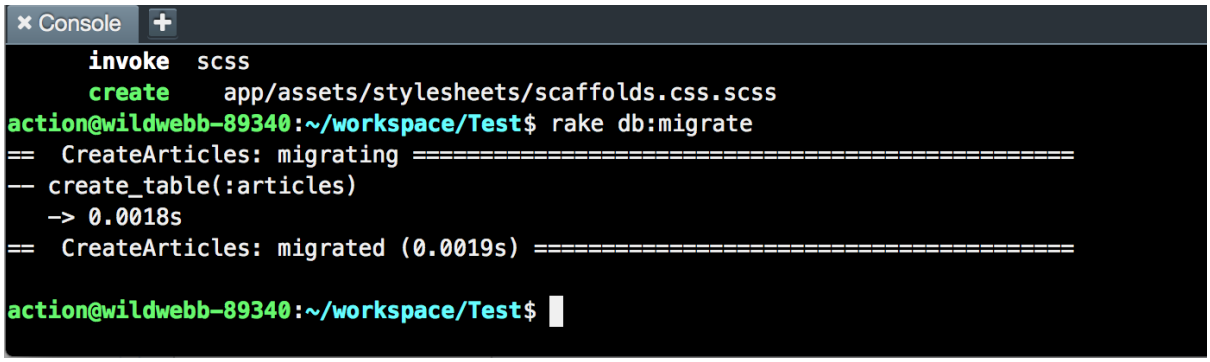
We are going to create a simple blog in a remarkably short time. The first thing you need to do is close down the server if it is still running. To do this, just type “Ctrl C” in the console (hold down the “ctrl” button and the “c” button at the same time).

Now, we are going to just use one command to create all we need to be able to publish blog articles, where the article has a title, content, an author and a date. Carefully enter the line in the image below into your console, and press “return”:

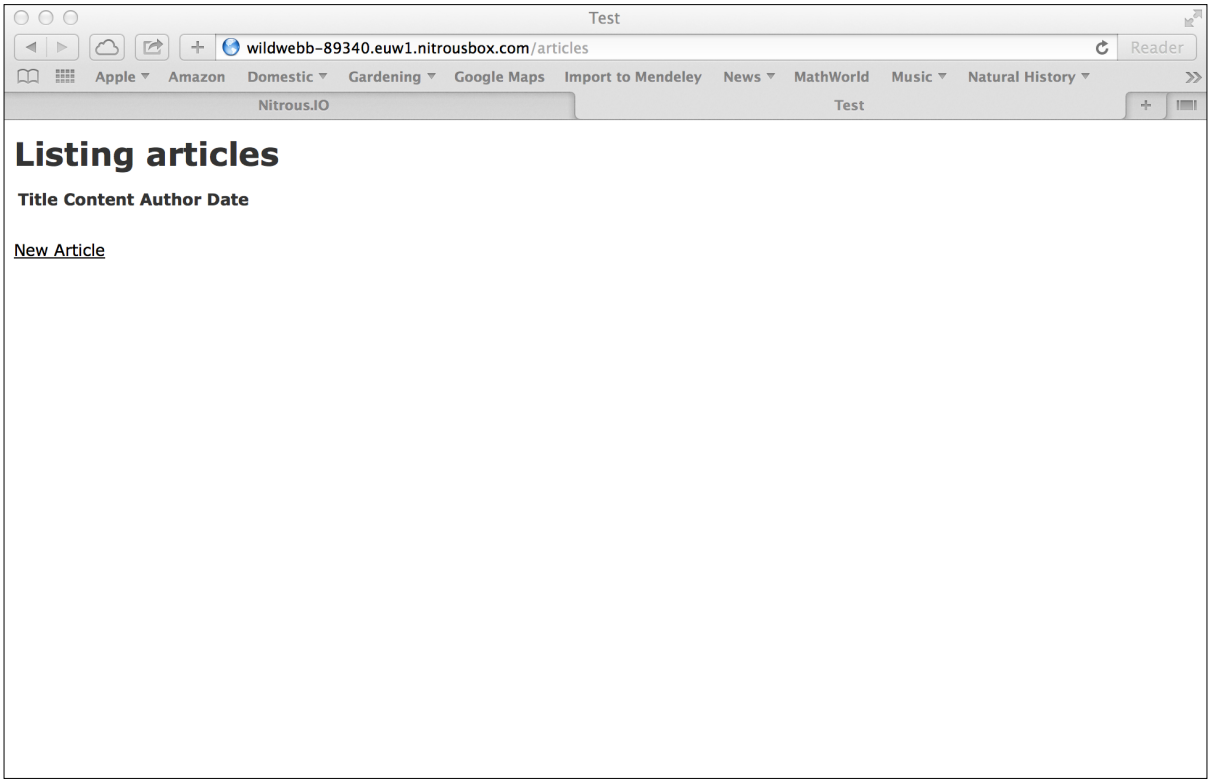


Once you have pressed “return”, a whole load more code is generated for you. It will also generate a database that will store all the blog articles. We have actually got a pretty complete website, but

before we run the server again, we must first make sure that a table is generated in the database that will record our blog articles. All we need to do to create this is run the "rake db:migrate" command as below:

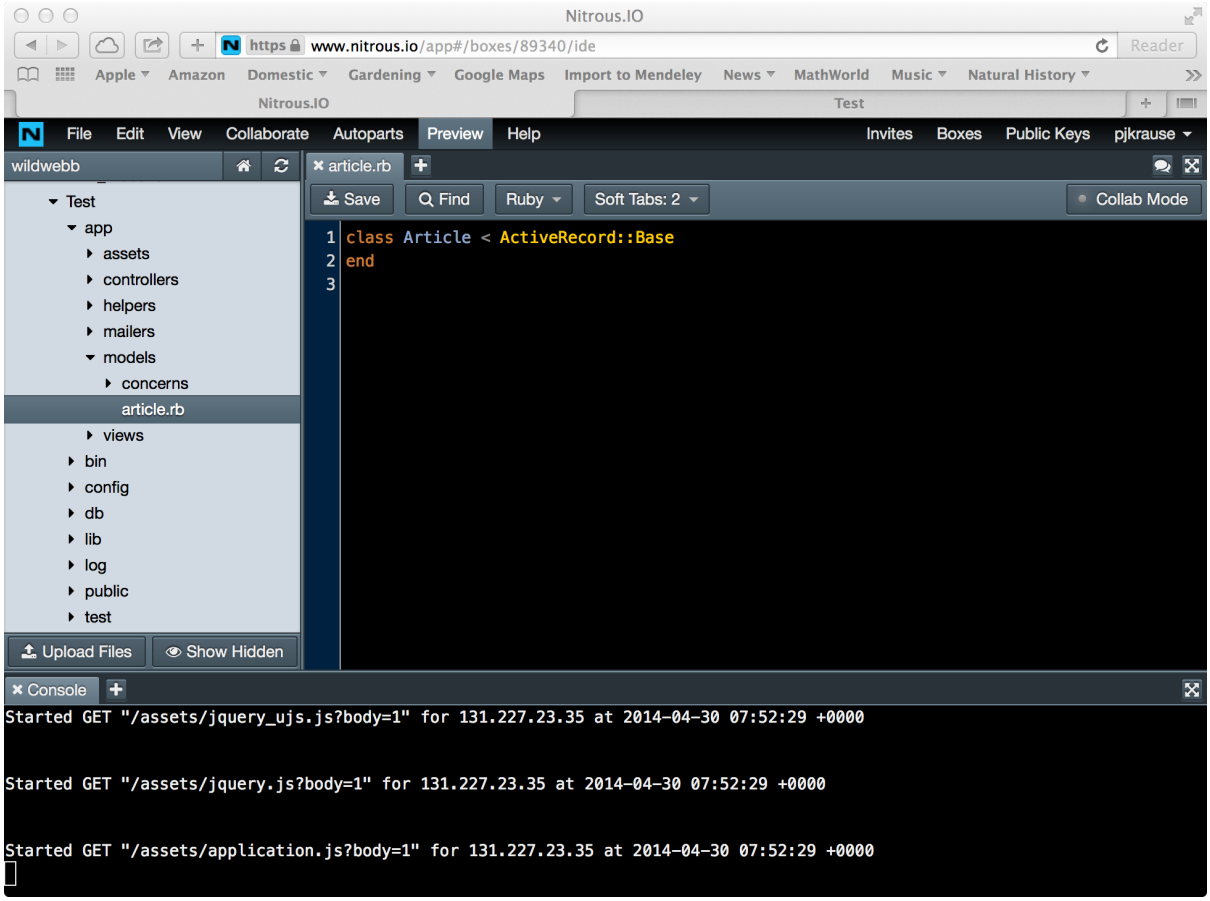


Now we can start up the rails server again. Do you remember how to do that? Once you have done that, go to the tab with the preview of your website and refresh the page. You won't see any changes to the welcome page but if you append "/articles" to the url in the address bar, then hit return, you will see something like this:



Go on, have an explore. If you click “New Article” you will be taken to another page where you can create a new blog article. You will also see that you can show, edit or delete existing articles - just follow the links to explore.

What Rails has done for you is to create a “model” class called *Article*. It has also created a series of web pages that enable you to Create, Read, Update and Destroy instances of your model (objects). And (provided you ran the “Rake db:migrate” command!) it has linked this to a back end database that ensures your articles are saved.



You can find the model class in a file called “article.rb” in the Test/app/models directory (see above). You will also see that as you (or

someone else) interacts with your blog, messages that log the activity will appear in the Console.

It is worth just convincing yourself that the articles are indeed being safely stored. Make sure that you have created at least one article (click on the “New Article” link to open a form that enables you to create an article). Then:

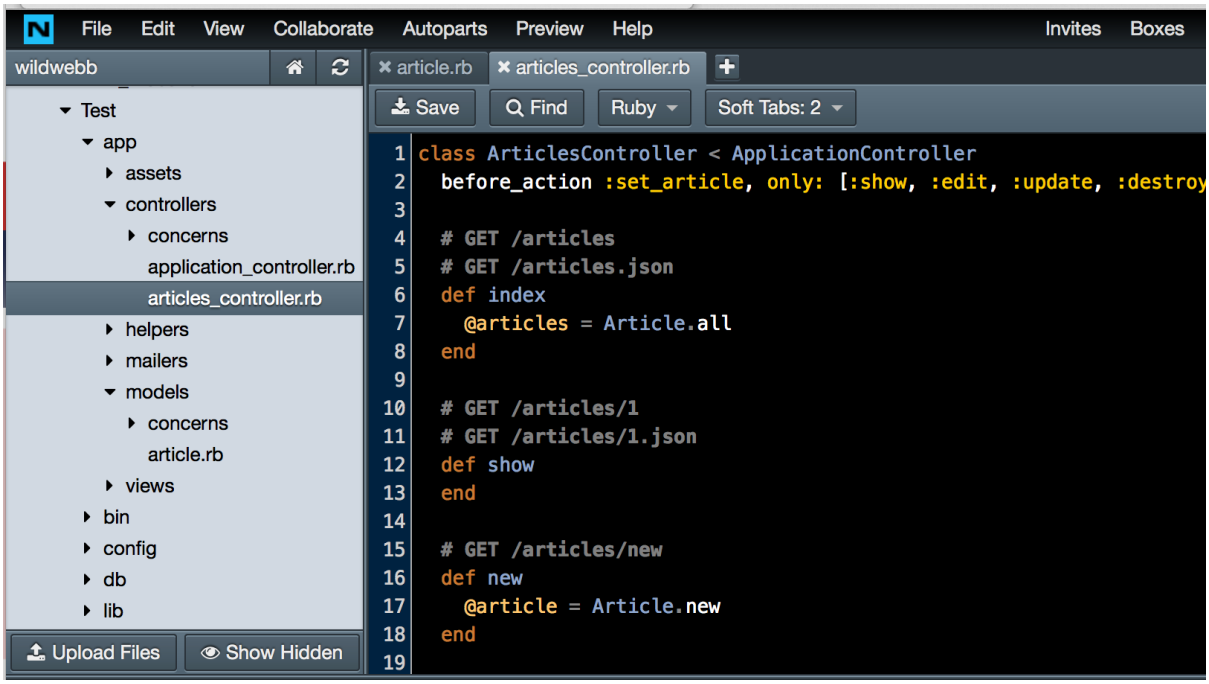
press “ctrl C” in the Console to close down the server;

start up the server again (“rails server” - “rails s” will also work);

go to the tab with your blog and refresh the “/articles” page.

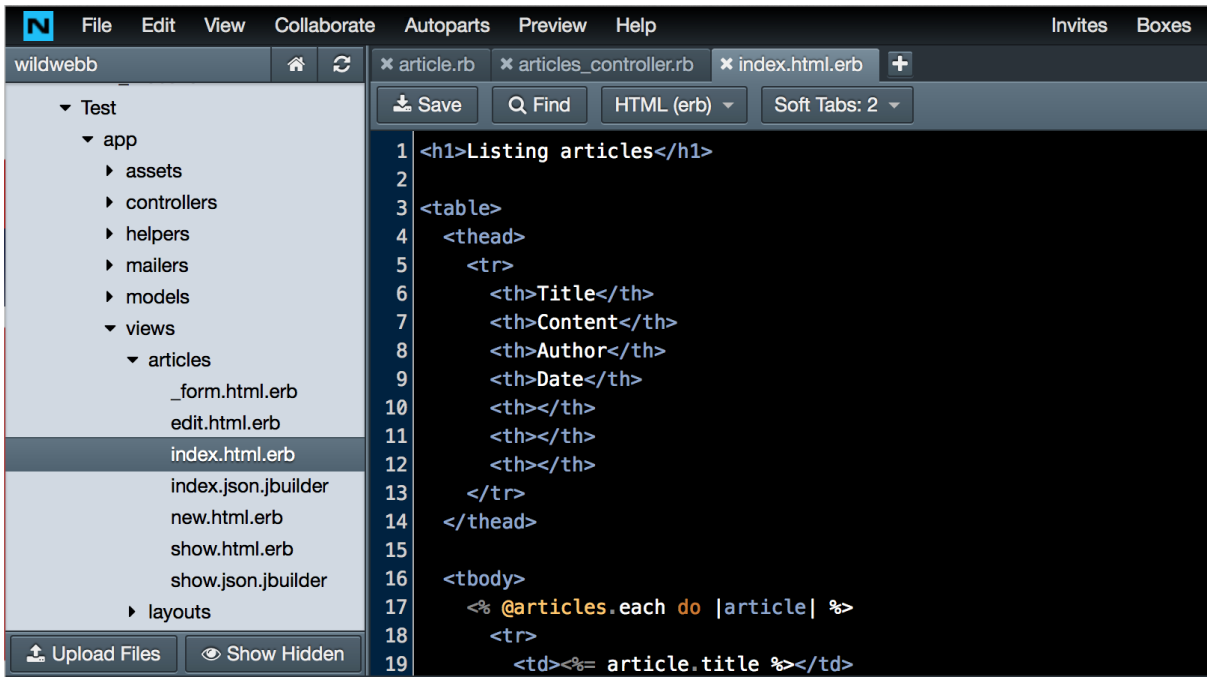
You will see that the article(s) you entered are still there.

Rails has generated many other files for you. One of the most important is “articles_controller.rb”:



This contains all the functions that are available to the outside world and can be used to manipulate articles. For example, the “index”

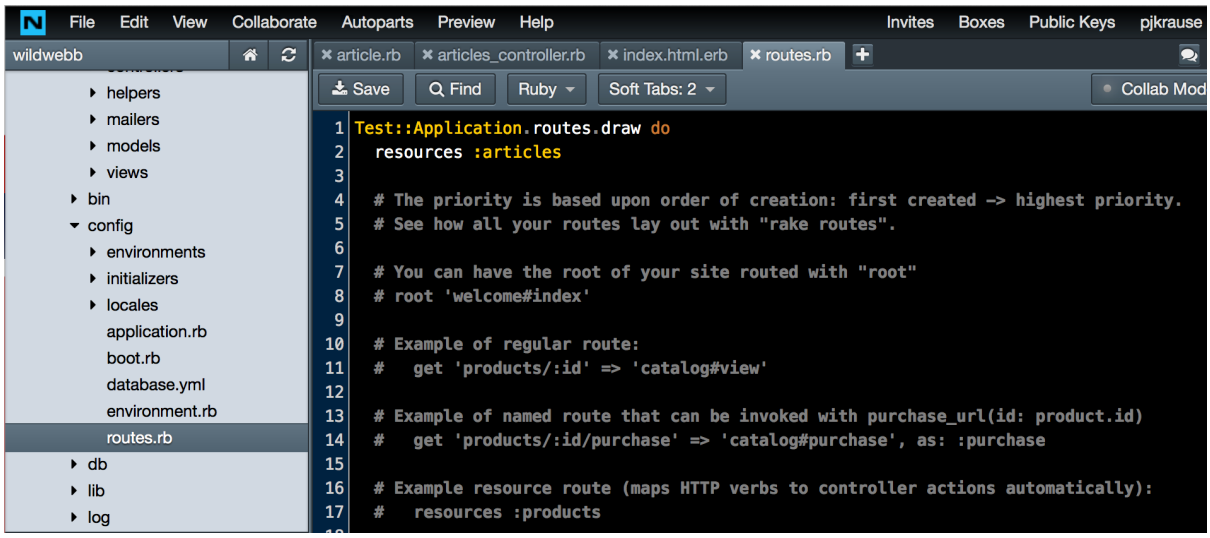
function collects up all the Articles that are currently available (it gets these from the database). This is just data, and we need to present it in a viewable format in response to our client's request. Rails does this by then passing control onto a "view file" that embeds the raw data into an html file:



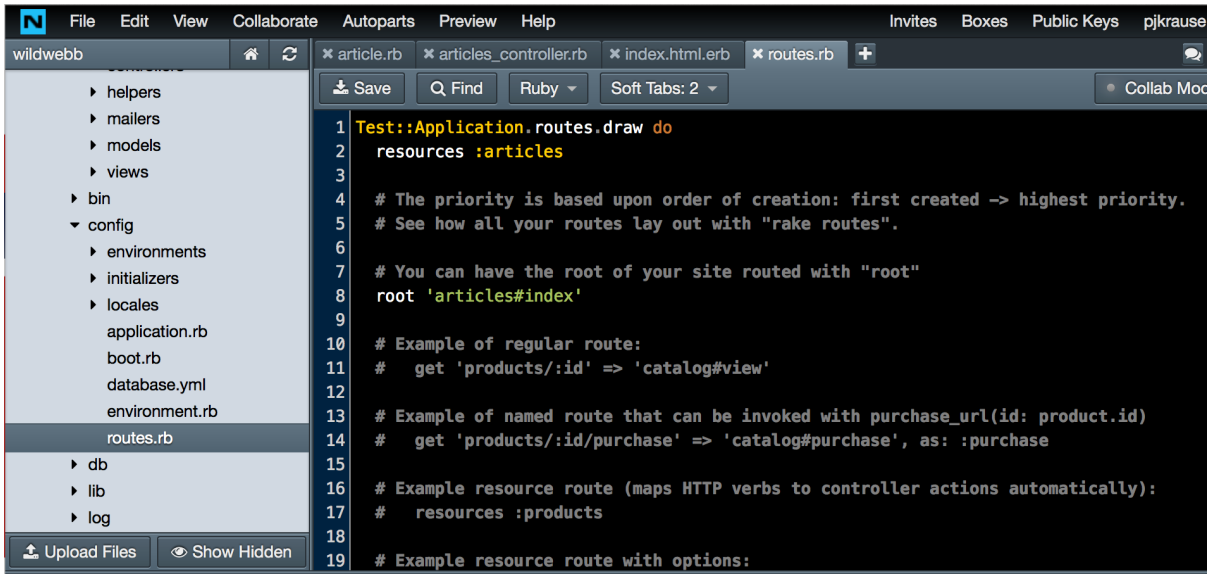
In this case, the articles are presented as rows in an html table. Don't worry about the details of the html for the moment - we will come back to that shortly. But do notice that there are separate view files for showing an index of articles, creating new articles and showing an existing article.

You could share your link again with your friends so that they can also see and enter articles in your blog. But there is something that is worth fixing first. You will have noticed that you need to add "/articles" to the url in order to see the index of articles. Actually, we would like to follow the normal web convention of just being able to enter the raw address of the host, and have it show this index page. You can fix that easily. Rails has a file that maps urls that are requested by a client on to

specific actions in one of the controllers. This can be found in the "config" directory:

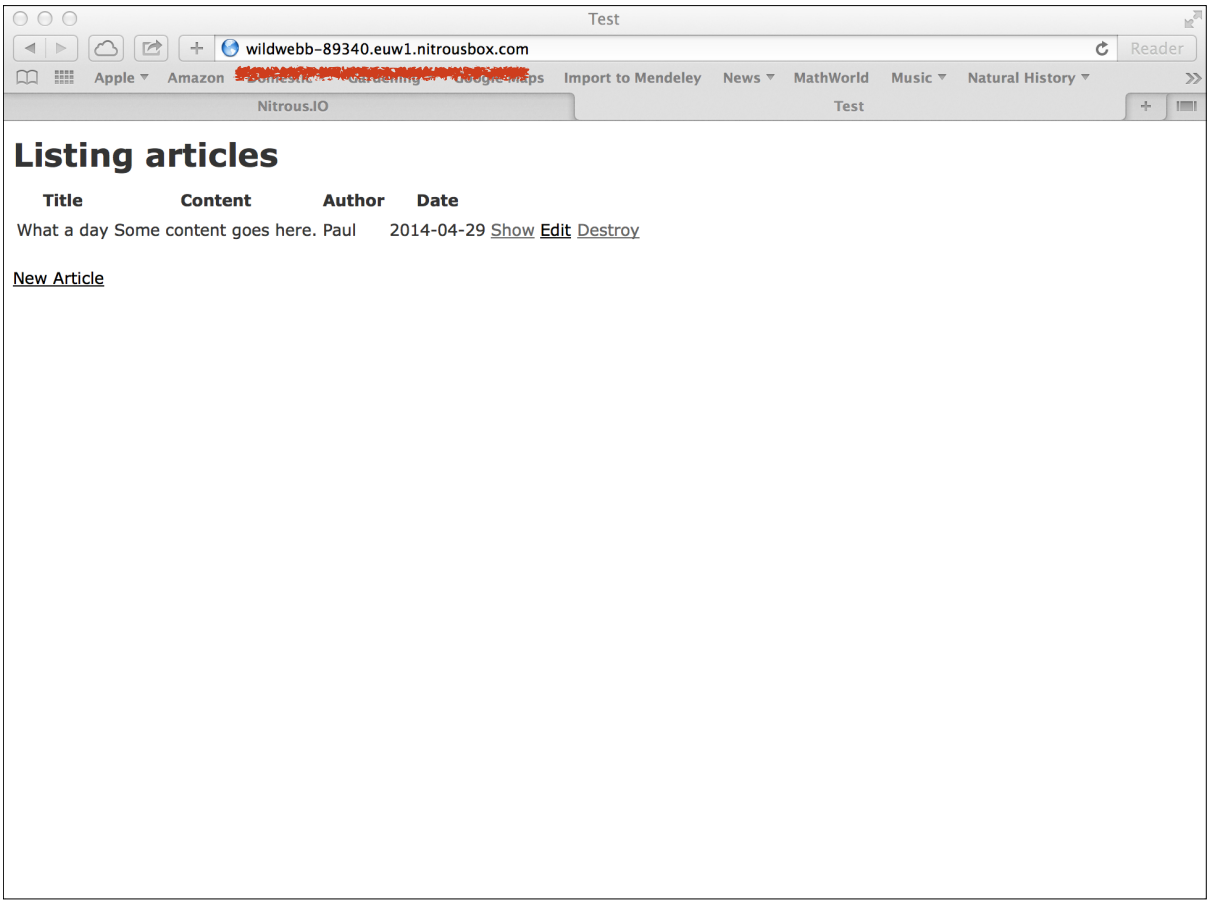


There is not much in there at the moment (most of the lines are commented out by starting them with the '#' symbol), as rails uses a standard pattern "resources" to set up all the routes to the controller class that we were looking at earlier. There is not much that we need to do here other than uncomment and slightly edit line 8 so that it looks like this:



You see that I have deleted the '#' and replaced "welcome" with "articles".

Now you can get to this page just by entering the host address of your blog in the address bar:

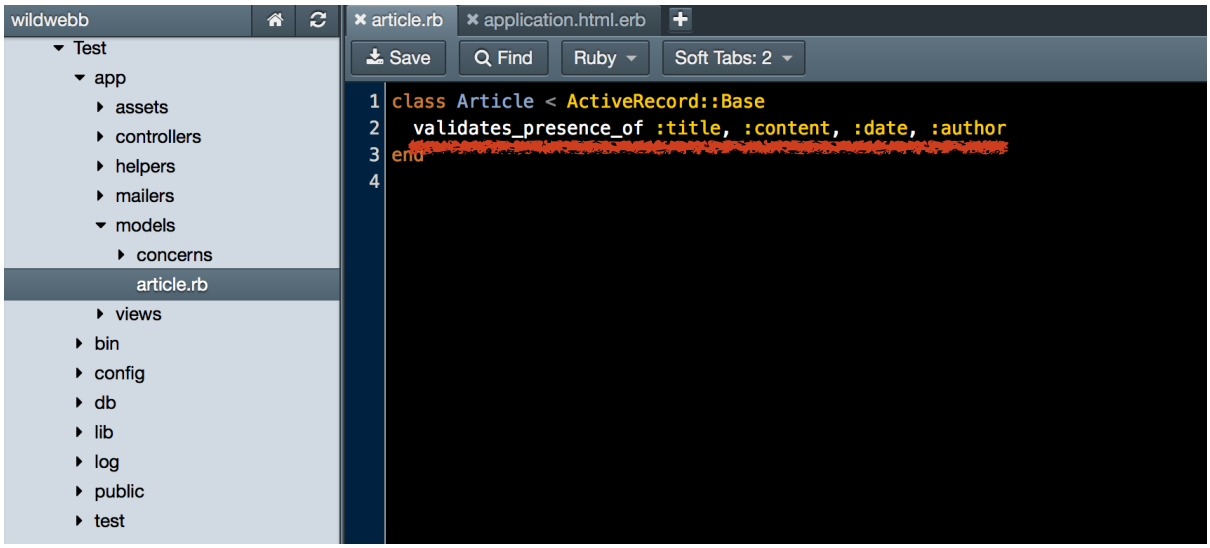


So now is a good time to send the address to your friends. However, it does not really look to be recognisably yours so let's customise it a bit.

Making the Blog your own

There is one issue that you should fix first. It is possible for you to create a new Blog article with absolutely no content at all! Try it - just leave the form fields empty and save it. That's not what we want. Instead we want to make sure that some test takes place to ensure we have at least some content before server space is allocated to a Blog article.

Rails makes this easy for you. All you need to do is add one line to your article model, and an inbuilt function is invoked to check content is provided. Insert line 2 below into your articles.rb file:



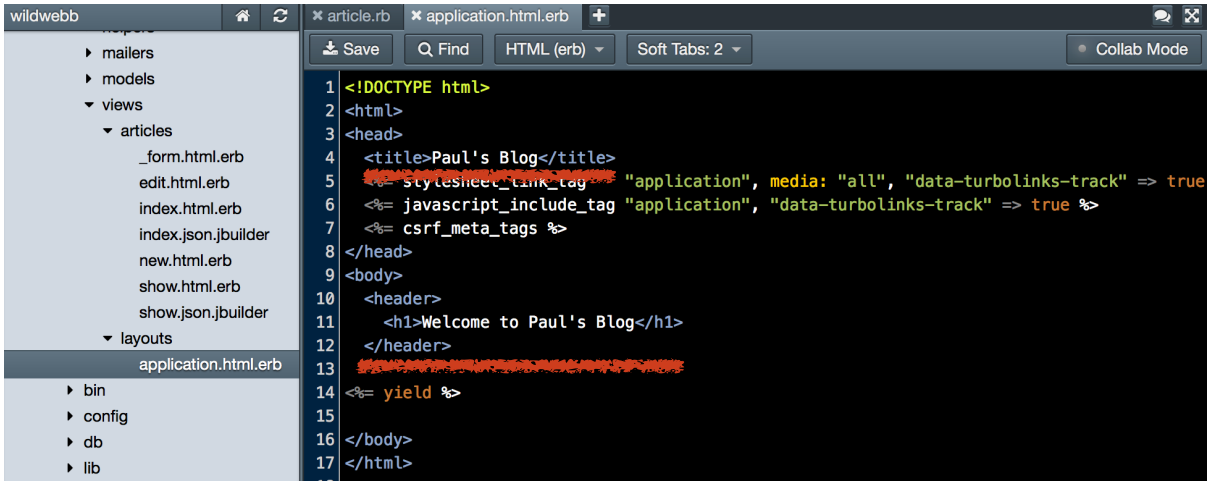
Notice that I have also requested that *date* be validated, even though it is not possible to submit the edited content of an article without a date being present. This is done because it is always possible for someone to try and "hack" the system by using some other method to submit data to our application so you always need to include

validations in your application on the assumption that sooner or later someone will find a way to submit invalid data. You want that to be rejected.

Once you have edited and saved your article.rb file, go back to your blog and try and submit a new article with no content. You should see these warnings are presented and submission is blocked:

You can clear the errors by either giving up and clicking “Back” or by entering some data and clicking “Create Article”.

But no one knows that this is your blog! We need to give it a title. Rails makes it easy to provide a layout that is common to all the subpages in your website. Take a look at the *application.html.erb* file that is in *views/layouts*. This file in effect "wraps around" all the view files for specific subpages. So if we edit this file, the content will be reflected across the whole site. Change it along the lines shown below:



You should, of course, use your own choice of title (unless your name also happens to be Paul :)

Now reload the page in the tab showing your blog and you will see the header is present throughout your website and the content of the title bar is more informative.

But it's not very colourful…

Essentially we use html to describe the content of a document with tags that indicate to a browser, or search engine, how the various elements of the document should be interpreted. For example, we just added a "header" to all of the documents which will provide some branding information - every up to date browser and search engine will understand this is the intention behind this new element. What we

don't do, though, is to include any information about how to style this content! This belongs in a different place.

Why? For three main reasons.

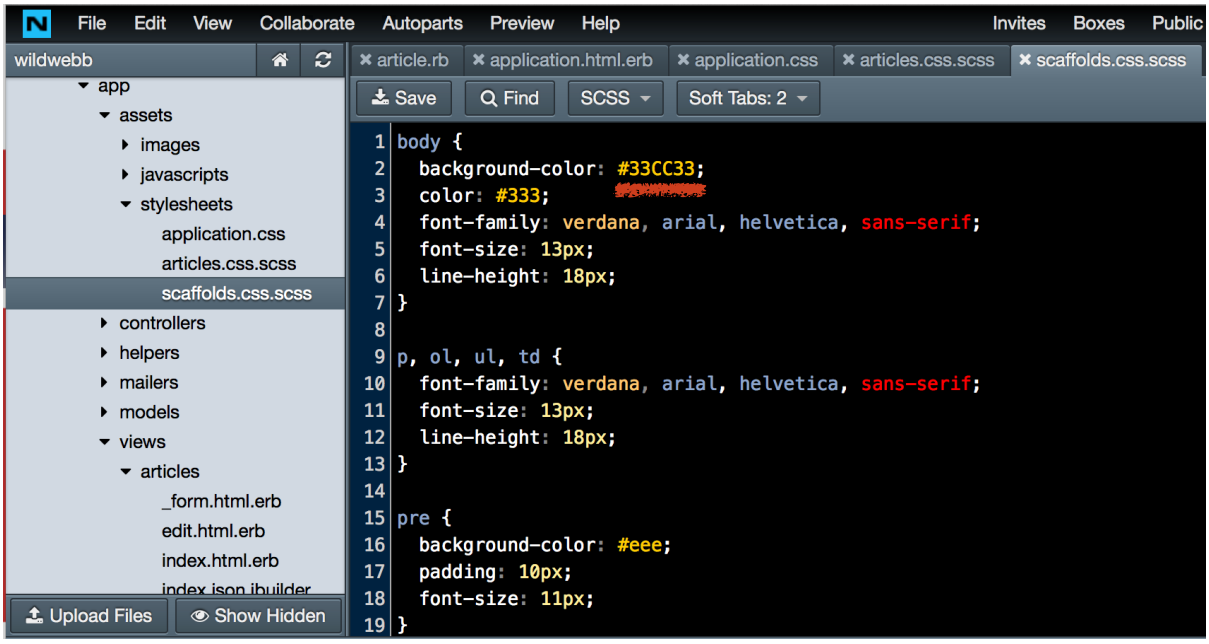
Firstly, we want a clearly defined responsibility for the people who are creating the html - they are domain experts who understand *what* should be presented in the website. They are not designers, so won't have anything to say about *how* that content should be presented.

Secondly, our content may be viewed through one of a range of different devices - laptop, tablet, smartphone, TV, printer… We may want the content to be styled differently depending on which device is being used to view the website.

Thirdly, for ease of maintenance we want to put all our styling information in a single place so that if we change the branding of our site we only need to change it in one place and the change will be reflected throughout the site.

So, what we do is to keep styling information out of the html and place it in separate stylesheets. These are then imported where appropriate into the html when it is loaded by the browser. You can see this happening in the image on the previous page - line 5 imports the application stylesheet. Let's change that a little so we can make our site more attractive.

Rails structures the styling a little as this can get quite complex. All the stylesheets are in a single directory: /app/assets/stylesheets. The main application.css file is used to pull in all the different stylesheets needed to style our application. We are going to just focus on the style sheet that was generated for our scaffold:



Colours are represented as hexadecimal codes for red, green and blue. Looking at the above illustration, you will see that I have changed the background colour for the body element to be shade of green (the "CC" indicates that green is the strongest component).

Make the same change to your file and then reload the blog. (You can of course experiment with different colours!).

Now lets make the header stand out a little more:

What I have done here is to make the header a darker shade of green, change the font colour to white and increase its height. I have

also arranged for the level 1 heading (h1) within the header element to be centred.

Note that I have used the slightly more compact "scss" notation within this file. It is compiled into the normal cascading style sheet (css) notation by the server.

Add lines 9 to 19 in the above figure to your own file, save it and then refresh the browser window that is showing your blog. It should look like this:

This is where we will end things for the present.

Where to now?

There are many, many resources available online. Ruby on Rails is not the only framework that is available, and Nitrous supports most of the good alternatives. If you would like to explore Rails some more, you can register for my online "Getting Started with Ruby on Rails" course if you use the coupon code "FriendsOfPaul" (clicking this link will get you straight there).