Coding workshop

September 2016, London

Brought to you by the http://bibliocloud.com team:

@andypearson
@databasesponge
@emilylabram
@tikdalton
@has_many_books

#publisherswhocode

Contents

1: Introduction

The tools of our trade

There are lots of different programming languages, and more are being created all the time. They are often very specialised. COBOL was designed to make it easy to solve business problems. FORTRAN was, and still is, used to write high performance scientific programs.

In this course you'll be using four programming languages:

- HTML (express information)
- CSS (style information)
- JavaScript (give information behaviour)
- Ruby (dynamically find information)

One of the most interesting and heartwarming features of the modern programming world is the open source movement. We build on the shoulders of giants, reusing code that the community have built, tested and released for use at no charge. These are not hobbyist resources: open source has the benefit of being more thoroughly and widely tested than closed, proprietary software.

We will be using three open source resources today:

- Sinatra, a web development framework
- jQuery, a JavaScript library
- Google Books API, a data source of Google's book data.

HTML

HTML is the language of the web. Every single website you've ever been on is made from HTML, and ePubs are made from it, too. It is a markup language for describing web pages. HTML stands for **Hyper Text Markup Language**.

CSS

CSS is a style sheet language used for describing the presentation of a document written in a markup language, such as HTML. CSS stands for **Cascading Style Sheets**.

Ruby

Ruby has been around for about 20 years. Ruby became very popular as a language for developing web applications, which are websites that users can interact with, about ten years ago, after the release of a web application framework written in Ruby, called Ruby on Rails.

The aim of Ruby on Rails was to make it very easy to develop web applications by hiding nearly all of the complexity from the developer. Today we'll be using a similar framework called Sinatra[1], which is an even more simple framework than Rails, to quickly create a web application in Ruby with minimal effort. Our publishing management system, Bibliocloud, is written in Ruby on Rails.

JavaScript

JavaScript is a language that runs in your browser, rather than on a server. It is the "behaviour layer" of the web. Form validations, pop-up dialogue boxes and search boxes which update as you type are all uses of JavaScript that you've used on websites.

---

1 http://www.sinatrarb.com/

jQuery

jQuery is a JavaScript library: a collection of code written in JavaScript that gives you extra pieces of functionality, such as a way to show and hide content, so you don't have to write it yourself.

Nitrous

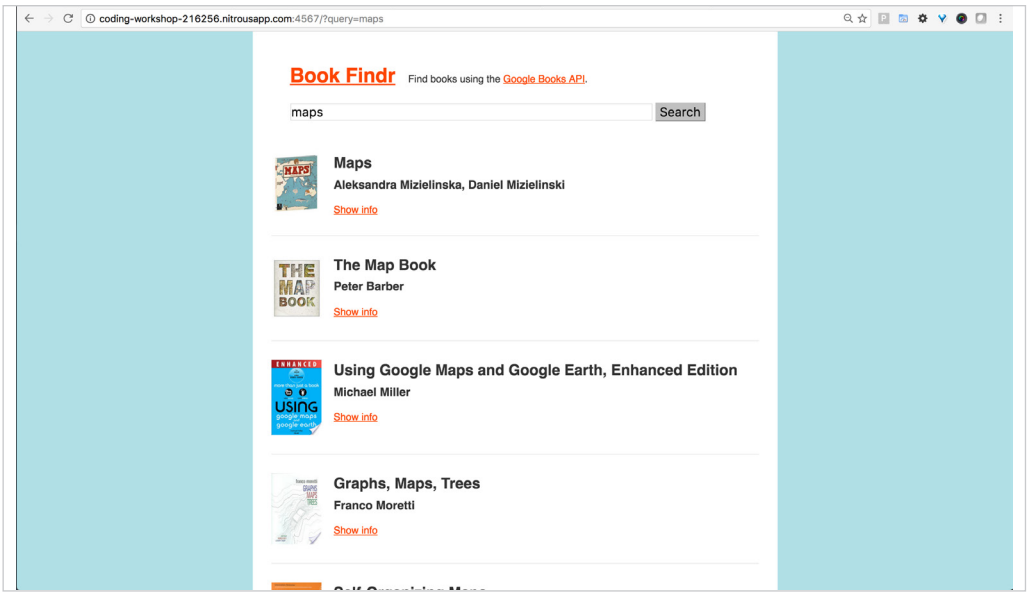
When a web application has been written, it used to be the case that you then had to buy or rent a server connected to the internet, and then install and maintain all of the required software and security configurations. However, another major recent change has been a profusion in companies, such as Amazon and Google, offering rental of their own computers for running code and managing applications. Some companies, such as Netflix, run their businesses entirely on rented servers, accessible over the internet in the cloud. And still more companies provide management layers on top of these services that hide their complexity from you, making it even easier to run your own application, and because they have a huge customer base of very similar systems, they can do so extremely cheaply.

Until very recently this still left you with the problem of setting up your own computer to develop your application, and this has often been a rather trying business. In the past few years, hosted systems have come to market that allow you to develop your code on remote servers, so you don't even need to have a sophisticated and complex development environment on your own desktop or laptop computers. Today you will be using an IDE called Nitrous to develop our code. IDE stands for **Interactive Development Environment**. The nice thing about that is that you will be able to access the work you do today from any computer via a browser -- so you can keep on learning and coding. You won't learn to program in one day. Today could be the start of the most rewarding journey of your life -- but it'll be a long

journey. Using an IDE such as Nitrous will set you up for easy collaboration and the ability to program anywhere.

Book Findr

Today you are going to write a web application which queries the Google Books API[2] and displays the results on a web page. You'll be using all the technologies described previously. So let's get started!



What you've learned

- You tell computers what to do by using a programming language.
- You can use more than one programming language at a time
- There are lots of different sorts of programming languages, each written with a particular speciality in mind.
- Learning to program takes time. Using a good development environment can help you get set up quickly so you can focus on programming, not installing software.

2 We'll look in detail at what an API is later on. For now, consider it a source of data

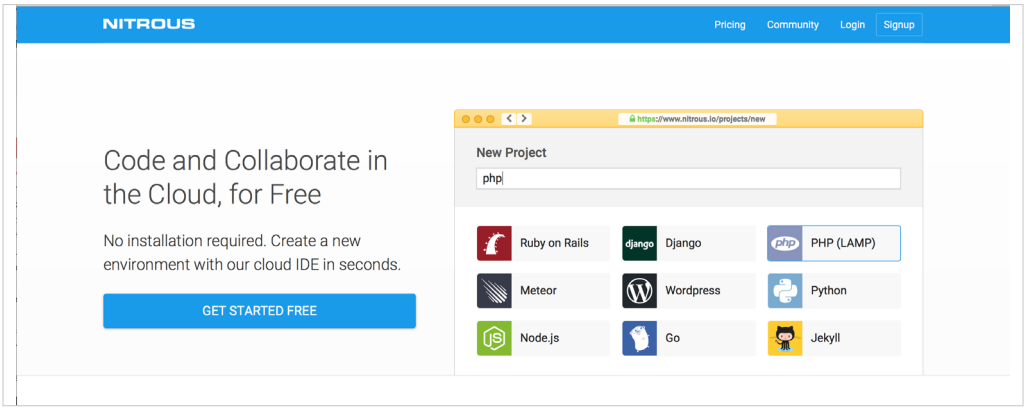
2: Set up

You'll need somewhere to type your code and run a server, and that's Nitrous, the IDE you were introduced to earlier. You'll need a Nitrous account to use their services. It's free, but requires an email address.

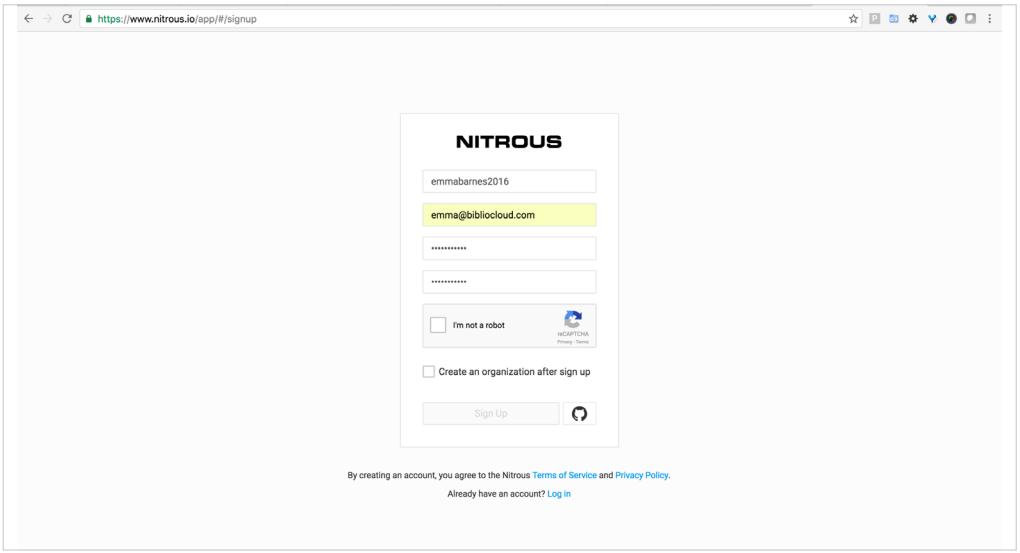
We recommend Chrome as a browser.

Set up a Nitrous account

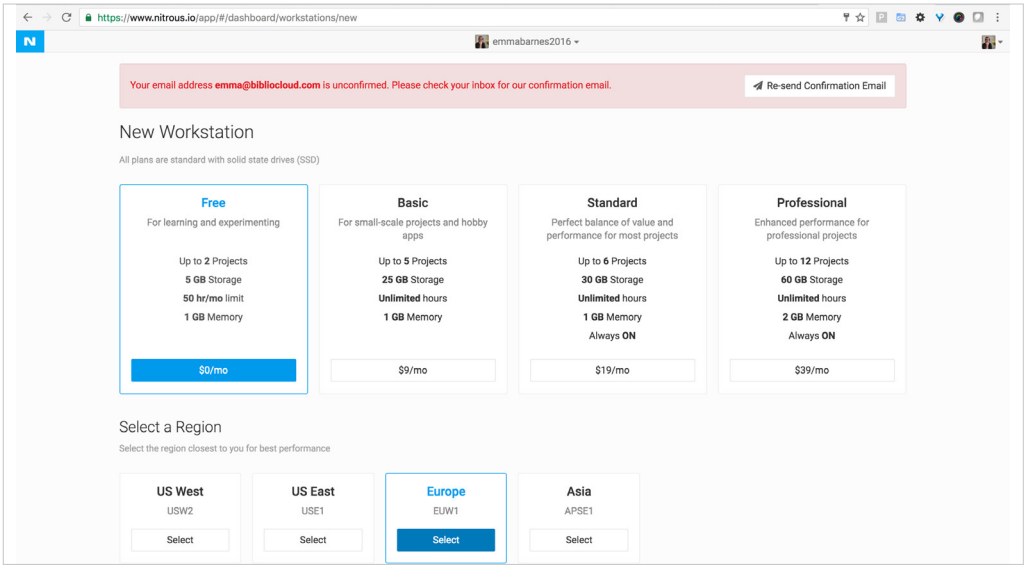
- In Chrome, go to **https://www.nitrous.io**
- Click 'Sign up'.



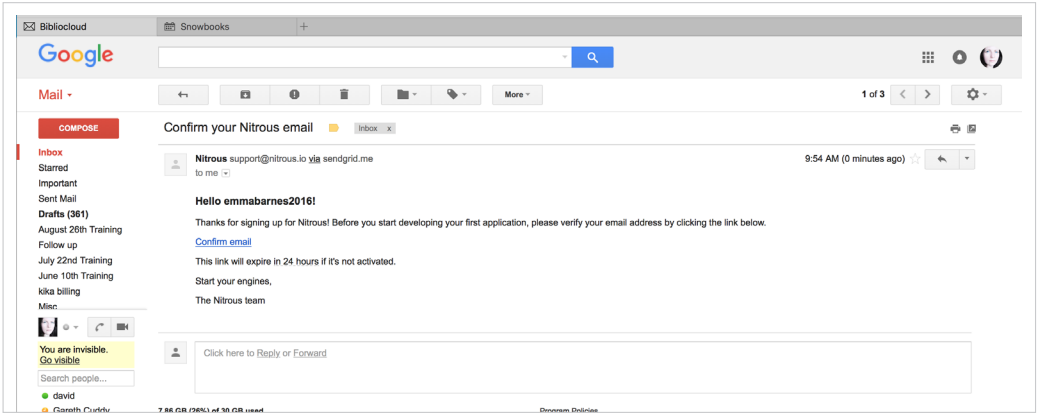
Fill out the form and click ‘Sign up’.



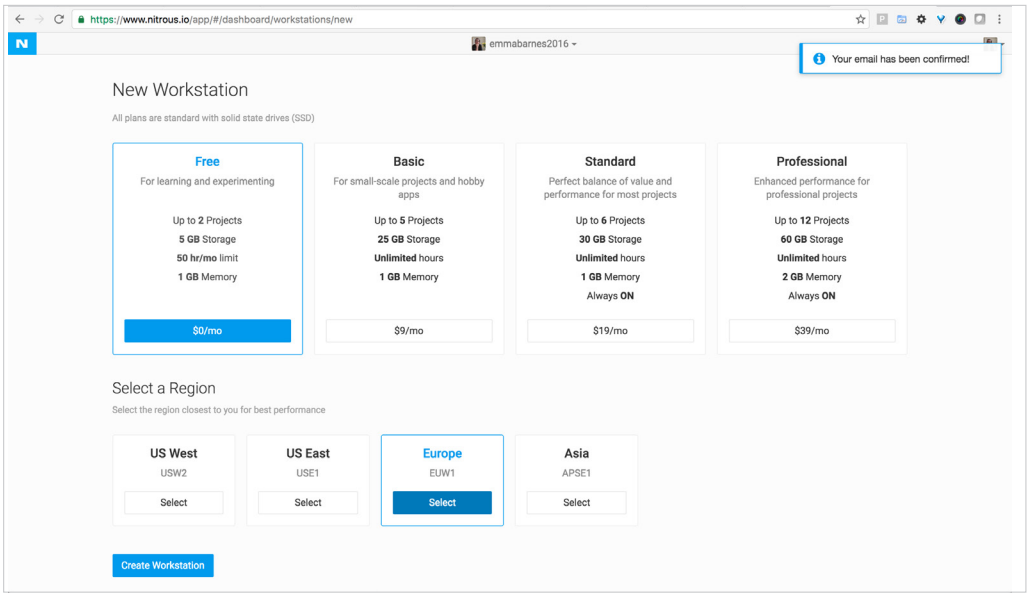
You’ll be taken to a page where you can create a new workspace. First, however, you’ll need to confirm your email address.



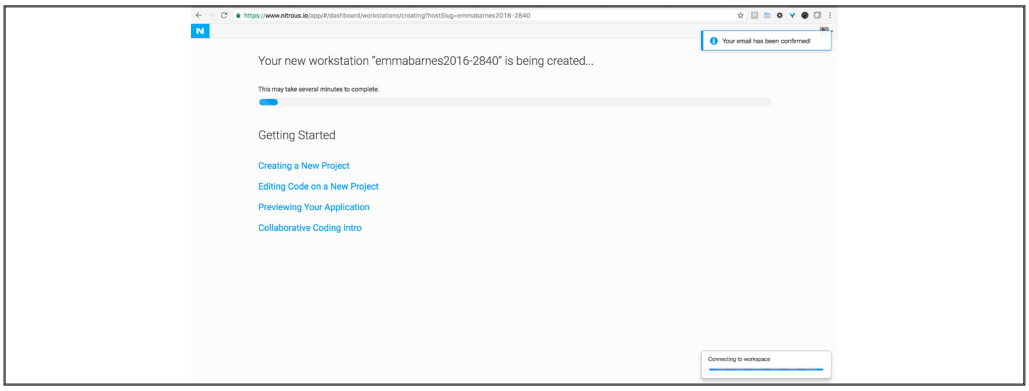
Click through from your email.



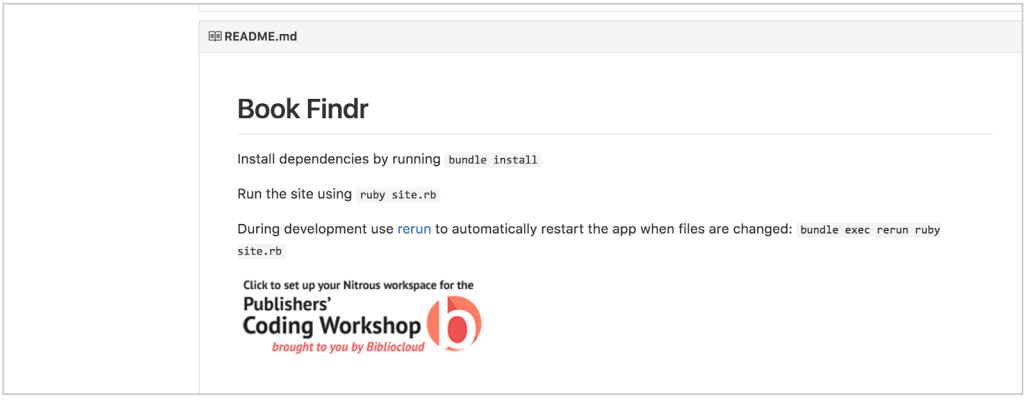
Now, select ‘Free’ and ‘Europe’ and ‘Create Workstation’.



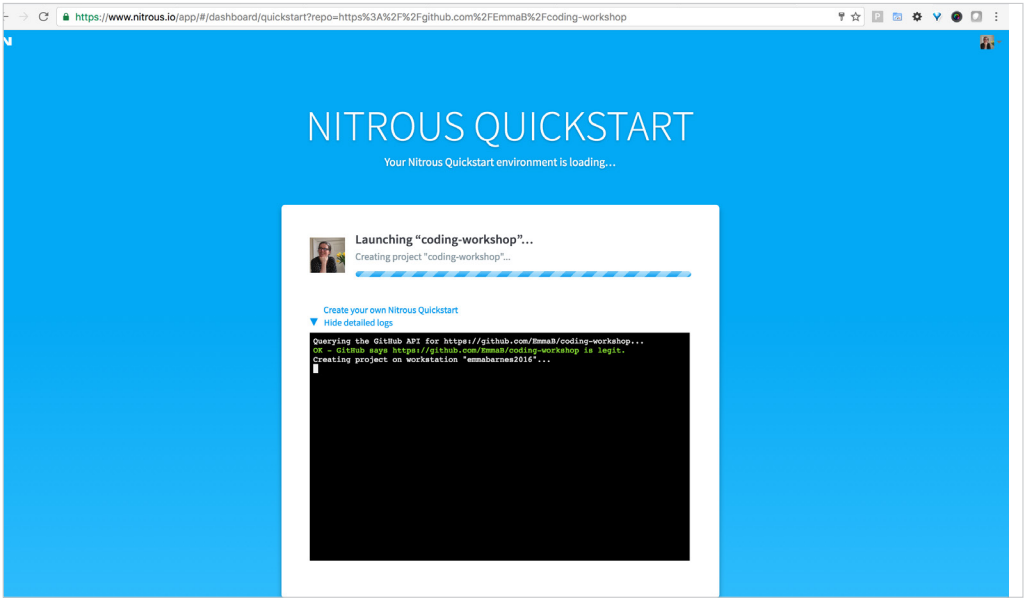
Leave this tab open for a few minutes. You’ll see that your workstation has started loading. We’ll load up some project files in the meantime.



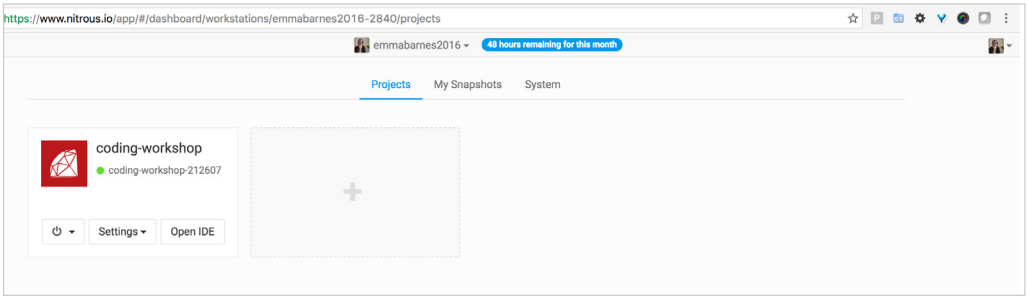
In a new tab on your browser, go to **http://bibliocloud.com/code**. This takes you to a Github code repository. Scroll to the bottom of the page and click on the Publishers’ Coding Workshop button:



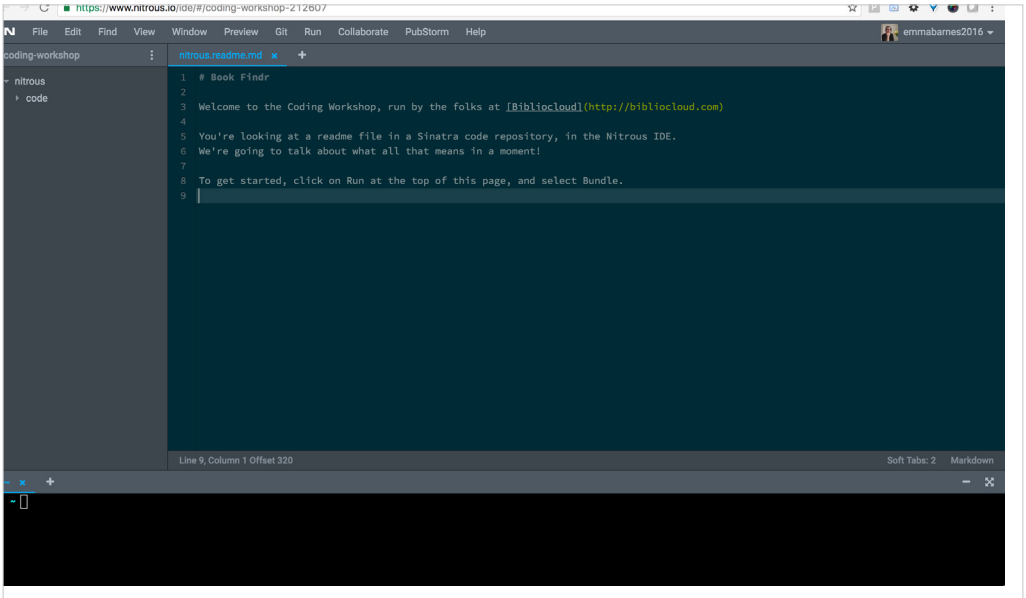
Nitrous will start to get set up. Click on `Detailed Logs` if you want to see what's happening.



Once it's done, in about three minutes, the page will refresh. Otherwise, click 'Open IDE' on your new 'coding-workshop' project.



You're taken to a page that looks like this:



This is your workspace. At the top of the workspace page, the screen is divided into two. On the left is a file system navigator – just like you'd see in Finder on the Mac or Windows Explorer on a PC. On the right is the contents of whichever file is open in your workspace. Nitrous will display a welcome message.

File structure

Expand the `code` folder on the left hand pane, and then expand the `coding-workshop` and `views` folders. These are the files that make up your app. At the moment there's not much code in them: you're going to be guided through the process today to turn this skeleton of files into a dynamic web app.

The most important files for today are in the `views` folder (or directory) and the one called `site.rb`. The `.rb` is a file suffix which means it is a Ruby file: the computer will expect to find Ruby code in it. The files in `views` are where you will write HTML and later embed some Ruby code to make your app dynamic.

The command line

At the bottom of the page you have the command line. It's the same as what you'd find if you've ever opened the Terminal on your Mac, or the Command Prompt on Windows. You can enter commands here which the computer will obey. Using the command line is often a lot quicker and more reliable than using the graphical user interface of your machine.

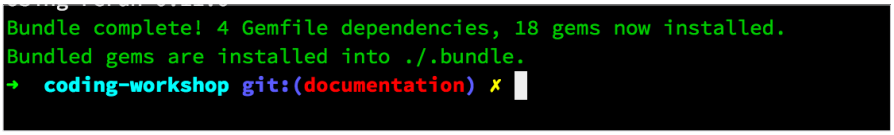
You're going to set up your project so it's got everything it needs to run its code.

On the top menu, click on `Run` and select

```
Bundle
```

This will download code from the internet that's needed to run your application. Watch the bottom command line pane -- this is called a stack trace, and acts as a record of what commands are being run. It will take about 3 minutes.

When that has finished, you will have a command prompt. Your command line pane will look like this:



Click on **Run** again and select:

```
Start the server
```

If you're quick, you'll see that the following appears on the command line:

```
cd ~/code/coding-workshop && ruby site.rb
```

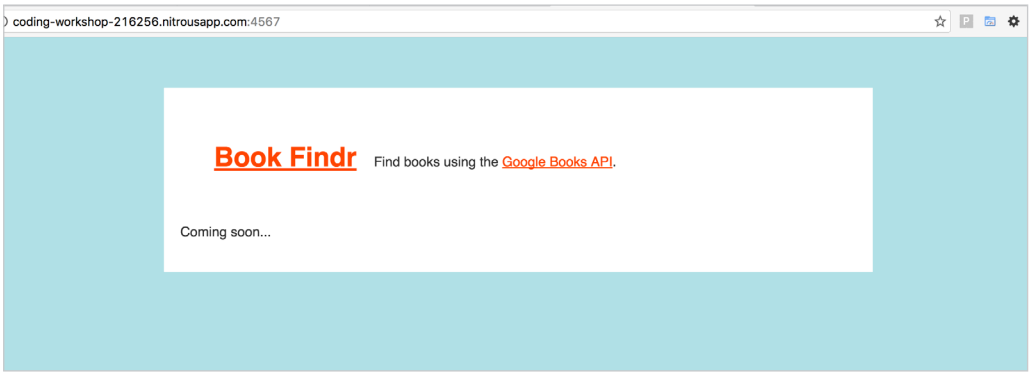
We have configured your **Run** menu to save you typing this in. If you had typed it in to the command line, though, it would have done the same thing.

The trace tells you something like this:

```
[2016-09-10 15:51:38] INFO  WEBrick 1.3.1
[2016-09-10 15:51:38] INFO  ruby 2.3.1 (2016-04-26)
[x86_64-linux]
== Sinatra (v1.4.7) has taken the stage on 4567 for
development with backup from WEBrick
[2016-09-10 15:51:38] INFO
WEBrick::HTTPServer#start: pid=16676 port=4567
```

There's a fair amount of computerese in there, but you can pick out some bits of sense. We're using version 2.3.1 of Ruby, and we're using something called Sinatra.

Click on **Preview** at the top of the page and select 'Port 4567'. A new browser window will open that looks something like this:



Congratulations -- you're running the Book Findr web app!

What you've learned

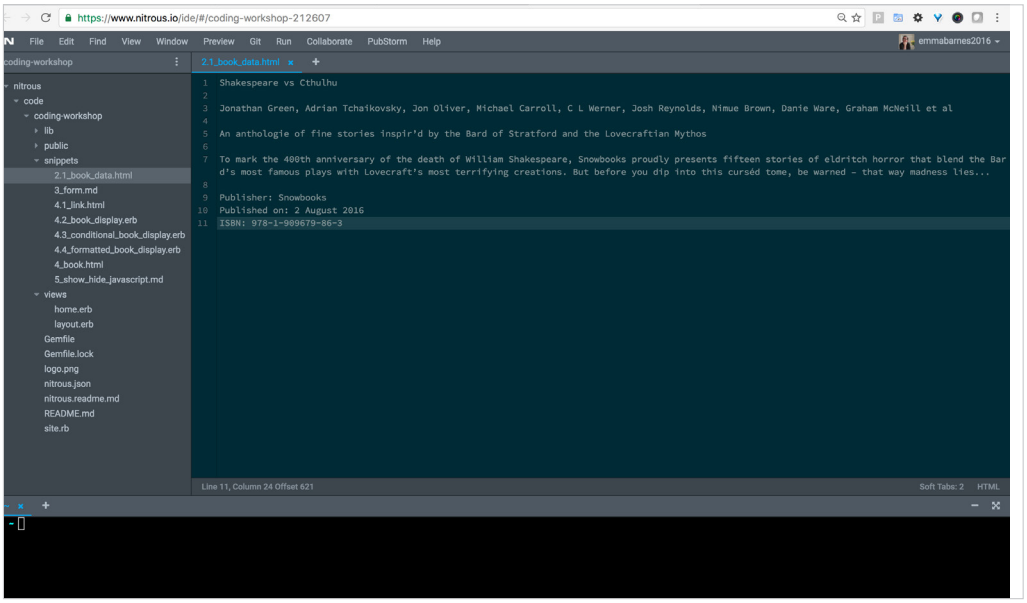
- You can use interactive development environments such as Nitrous to save you having to install lots of programs directly on your own computer.
- An IDE such as Nitrous contains a file directory, a pane to edit your code in, and a command line that you can write commands in, and run a server from.

3: HTML

Your app is looking pretty bare at the moment. You need to write some code! In this section you're going to develop your app so it displays one book's worth of data. Later on, you'll make this content dynamic, but for the moment the data will be hard-coded in HTML.

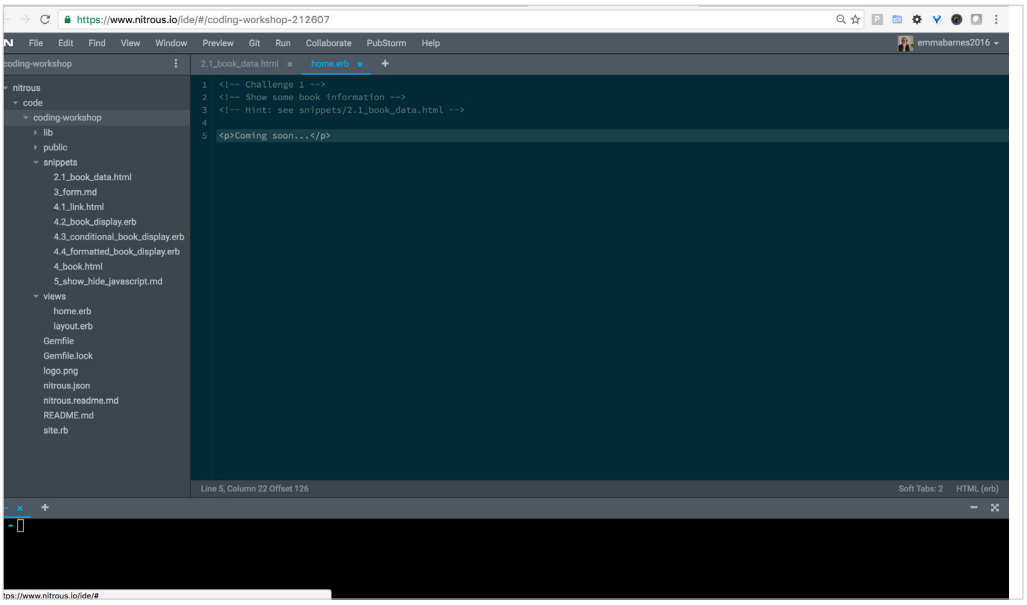
Challenge 3.1: Add some content

In your Nitrous workspace, navigate to the `snippets/3.1_book_data.html` file and click on it. It will open in the main pane, on the right:



That's some raw book data that we've put there for you to get started. Copy it by selecting it all and clicking **cmd+c** (Mac) or **ctrl+c** (Windows).

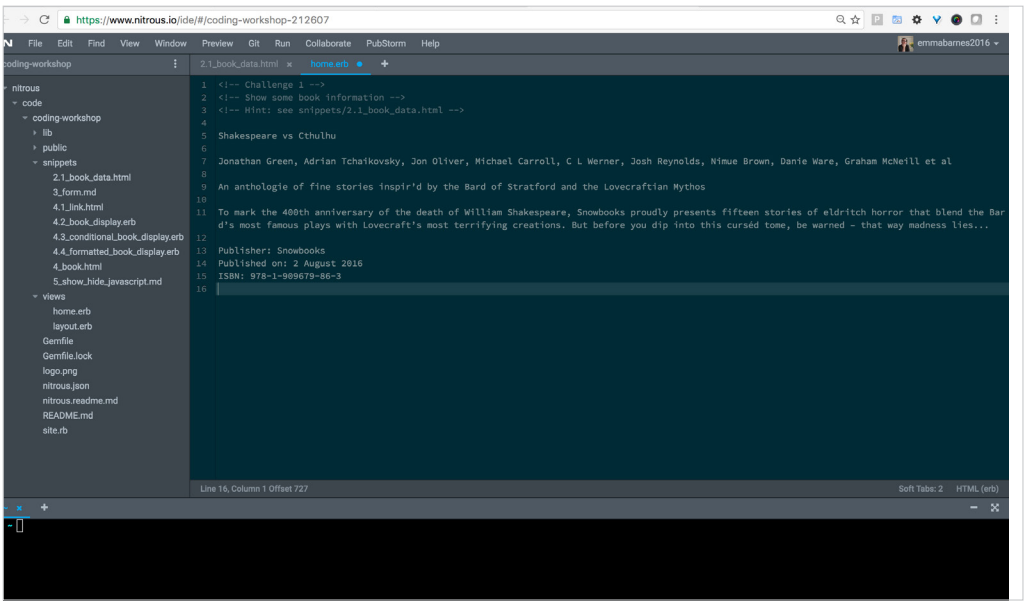
Next, navigate to the `views/home.erb` file.



Paste your book data into the page, completely replacing the line that reads

```
<p>Coming soon...</p>
```

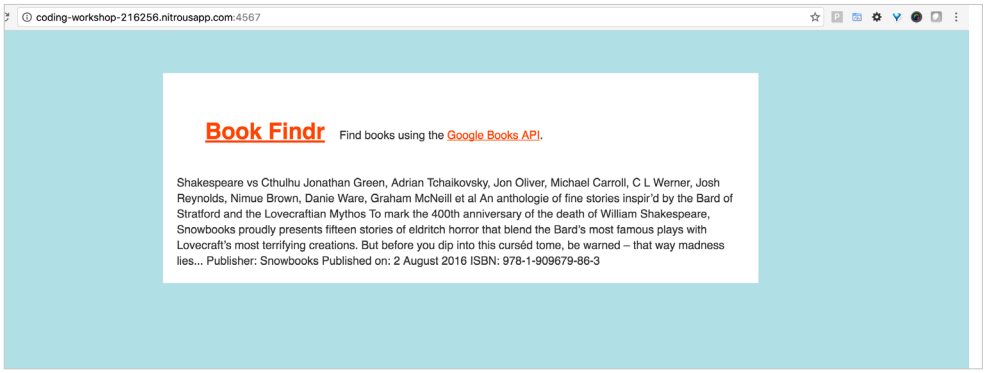
Click File and Save (or you can type cmd+s on Mac or ctrl+s on Windows). Your page should look like this:



> You will have to press ‘save’ every time you change one of the files. You don’t have to save command line commands: they run when you press ‘enter’.

You’re going to go to your Book Findr app browser window, and refresh the page. Before you do that, though, take a moment to consider what you expect to see.

Were your expectations correct? Here’s what your browser looks like:



Uh oh! All your line breaks are gone. That’s because so far your file contains content, but not HTML mark up.

HTML vs CSS

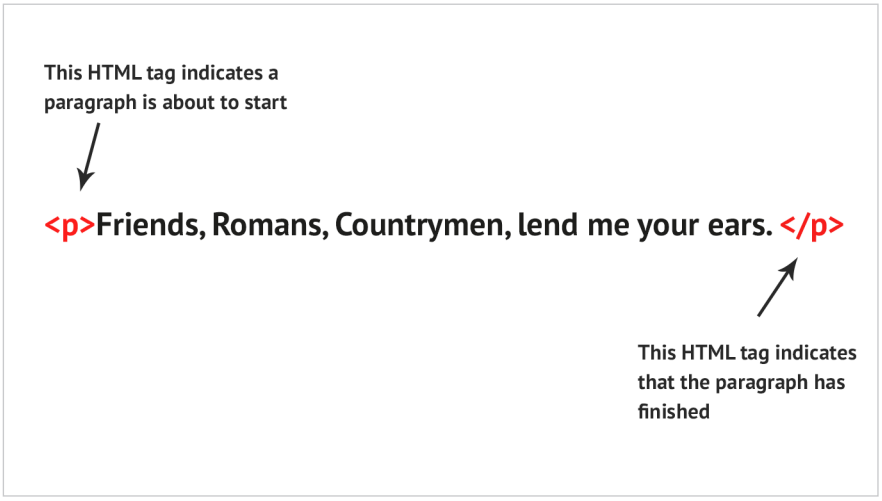
You are going to use HTML to define the structure of your web page. HTML provides the structure; later on, we’ll use CSS to provide the visual layout. This is an important point. A piece of text may be classed as a paragraph using HTML. That is a different to saying that the paragraph style should be Lucida Sans, 14pt, dark grey, and indented 5px on each side. You will define such stylistic instructions in CSS.

HTML tags

HTML is composed of HTML tags that go at the start and the finish of the content that you want to mark up. They come in two parts: an opening HTML tag such as `<p>` where you have a code such as `p` within two angle brackets, and a

closing HTML tag such as `</p>` with the `p` code within the same angle brackets, but with a `/` before the `p`.

It's a bit like the old advice for giving a speech. People say "Tell 'em what you're going to tell 'em; tell 'em; tell 'em what you've told 'em":



Here's a list of HTML tags. Your challenge is to use them to tell the browser how it should structure your page.

Paragraph

```
<p>...</p>
```

e.g.

```
<p>An anthologie of fine stories inspir'd by the
Bard of Stratford and the Lovecraftian Mythos</p>
```

Strong

```
<strong> ...</strong>
```

e.g.

```
<p>An anthologie of fine stories inspir'd by
the <strong>Bard of Stratford</strong> and the
Lovecraftian Mythos</p>
```

Headings

```
<h1> ...</h1>
```

```
<h2> ...</h2>
```

```
<h3> ...</h3>
```

e.g.

```
<h2>Shakespeare vs Cthulhu</h2>
```

List

```
<ul>
     <li>....</li>
</ul>
```

Usually interpreted as a bulleted list e.g.

```
<ul>
     <li>Publisher: Snowbooks</li>
     <li>Published on: 2 August 2016</li>
     <li>ISBN: 978-1-909679-86-3</li>
</ul>
```

Division

```
<div>
     ...
</div>
```

A division or section of the page e.g.

```
<div>
     <p>An anthologie of fine stories inspir'd by
the <strong>Bard of Stratford</strong> and the
Lovecraftian Mythos</p>
</div>
```

Image

```
<img src=''  alt=''>
```

Code which points to a file location of an image to display. `src` means source -- the file location. `alt` means alternative text -- vital for accessibility.

```
  <img src='images/jacket.jpg' alt='Shakespeare vs
Cthulhu cover image'>
```

Challenge 3.2: Apply HTML tags to your content

Use HTML tag syntax to mark up text in `home.erb`.

Have a go at typing in some HTML tags from the list above into the text you copied into home.erb. Then look in the snippets folder for `3.2_marked_up_book_data.html` where you'll find the answers for reference. Make your version match the snippet. Try not to copy and paste.

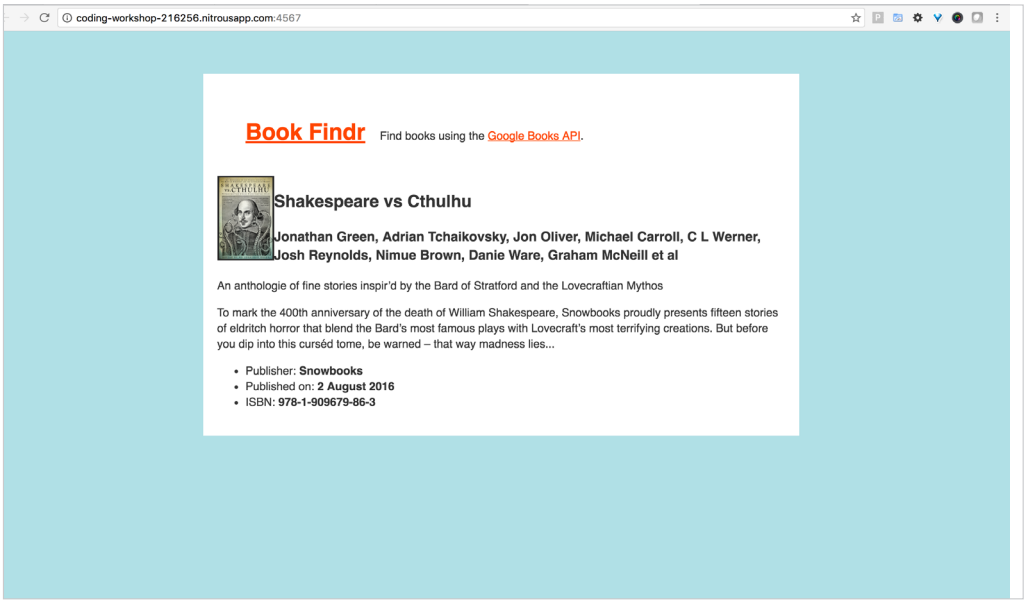
What you've learned

- HTML structures web pages; CSS styles them.
- There are opening and closing HTML tags.

4: CSS

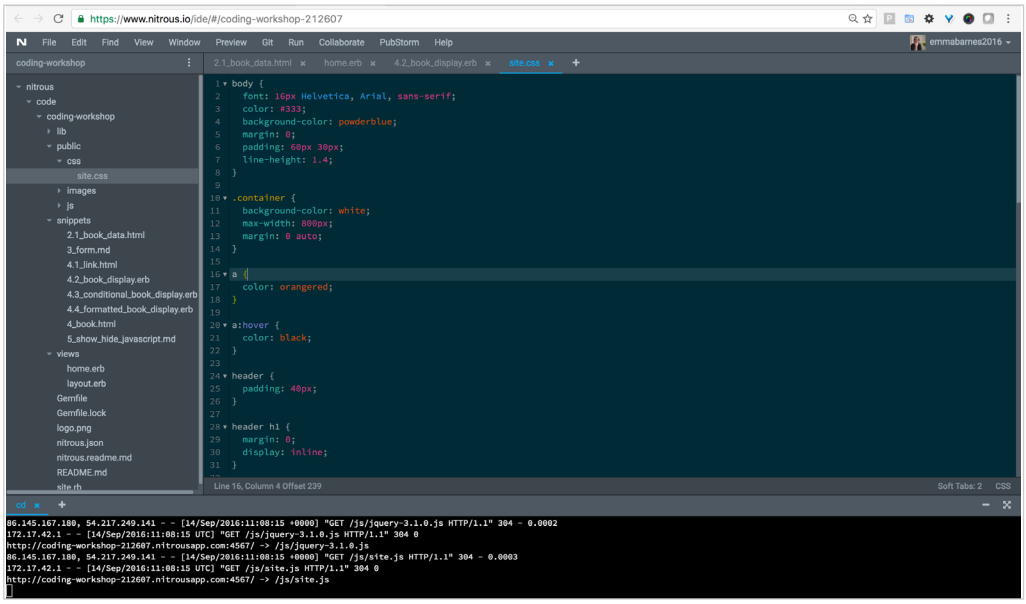
CSS

How does your app look now? Refresh your browser to see. You'll see something like this:



That looks a lot better -- but the text is bunched up against the image and you could sort out the spacing. You can make these style changes with CSS.

Navigate to `public/css/site.css`.



This file is full of rules that the browser will use to display your HTML. The first method in the file is:

```
body {
  font: 16px Helvetica, Arial, sans-serif;
  color: #333;
  background-color: powderblue;
  margin: 0;
  padding: 60px 30px;
  line-height: 1.4;
}
```

CSS has got beautifully readable syntax. Try reading that aloud and imagine what the HTML that uses that style definition will look like. The text it's applied to will be 16px, which is pretty much the same as 16pt in InDesign or Word or similar. It'll be in Helvetica, if the system can find that font, or Arial or another sans serif font as a fallback. The background colour will be a lovely shade of blue. The area

of the page will have some padding around it so that there's enough line space. And so on.

How do you get the HTML to use the CSS rules? Another word to describe these rules is "classes". If you want to have one of your `div`s use the `book-details` CSS rule, you change it from this:

```
<div>
```

to this:

```
<div class='book-details'>
```

Challenge 4.1: Apply CSS classes to your HTML

Apply CSS classes to the two `divs` in `views/home.erb`.

Look in the snippets folder for a file called `4.1_css_classes.html`. But try not to copy and paste.

Changing the styles

You have applied some CSS to the HTML. Now, you are going to write your own CSS.

Challenge 4.2: Change colours and fonts

Look in the snippets folder for a file called `4.2_css_changes.css`, and edit the CSS in `public/css/site.css` as the comments indicate. Try not to copy and paste.

What you've learned

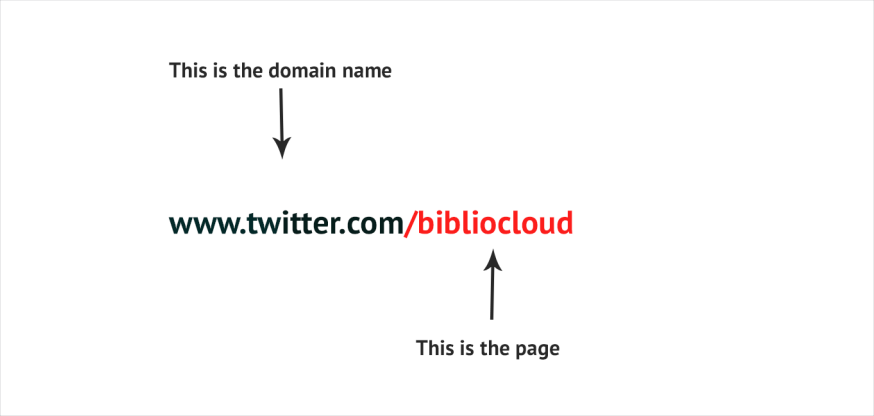
- You link HTML and CSS styles by declaring classes in your HTML.
- If you want to change the style of your web page, you edit the CSS.

5: A static website

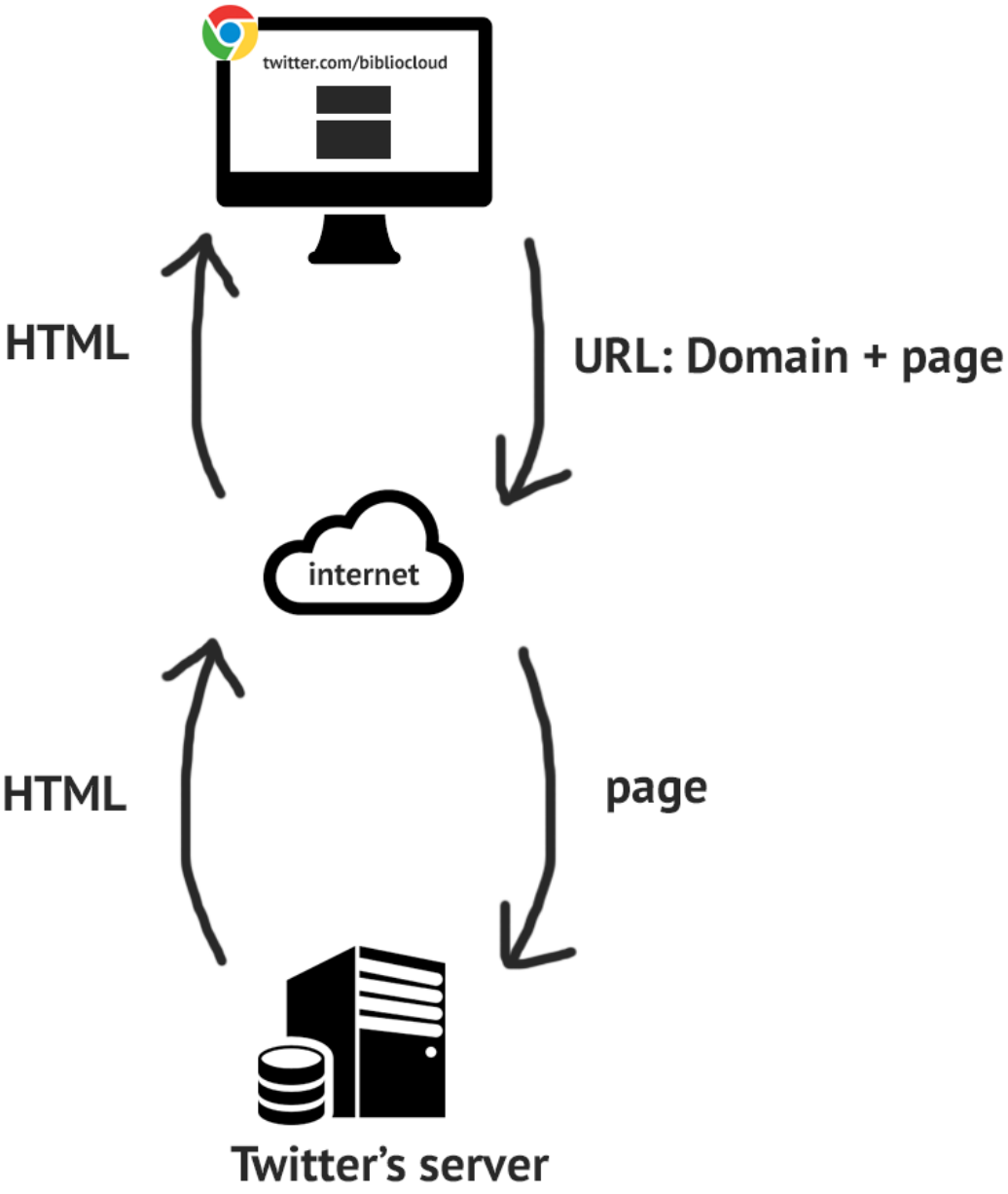
How the web works

It's probably worth going over what happens when you visit a website, since you're creating one.

You start the conversation when you tell your web browser the URL you want to visit. Let's imagine you want to visit Bibliocloud's Twitter page. You'd type 'www.twitter.com/bibliocloud' into the address bar of your browser (or maybe you'd look it up on Google and click on a link).



The URL gets broken into two halves. The first part – what you might think of as the 'domain name' – specifies which webserver you want to talk to. In this case you're connecting to the servers at twitter.com. The second part of the URL says which page from that website you want to look at. In this case that's the '/bibliocloud' part of the address. If you leave that part out, you'll typically get taken to the home page of the website.



The round trip from browser, via the internet, to a server and back.

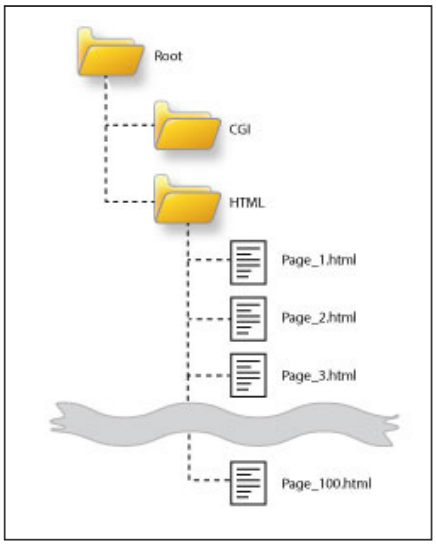
So what does a webserver do when it receives a request for a page? 99 times out of 100 it replies to that request by sending back an HTML file. The HTML might specify a heading which says ‘Bibliocloud’. Then the relevant piece of CSS will specify that it should appear in 15 point Lucida Sans in white.

To recap: the web browser on your computer requests a web page from some remote server. That server sends back an HTML file, plus some CSS. Then your browser uses the CSS to layout that HTML so that you can see the web page properly displayed on your screen.

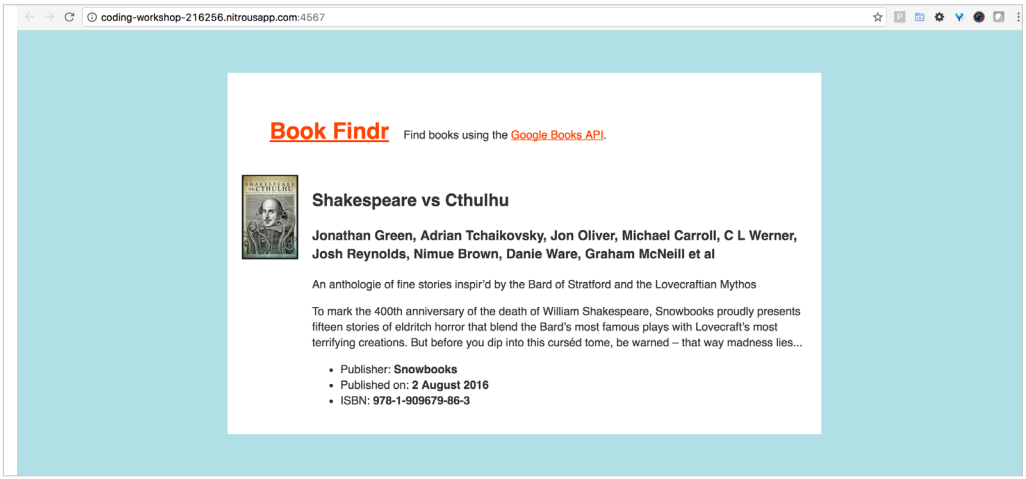
Static websites

Where does a webserver get the HTML files that it sends out? The simplest option – but not the one you’re going to choose – is to write them all out in advance. So if a website has 100 pages, then there will be 100 HTML files sitting in a folder on that server waiting for someone to request them.

That’s called a ‘static website’ because every time a user requests a particular web page they get the same HTML file sent to them. The HTML doesn’t change from day to day unless a designer goes in and manually edits it, which is why it’s called ‘static’.



The web app that you've set up so far is a static website. Look at the browser that's running your code:

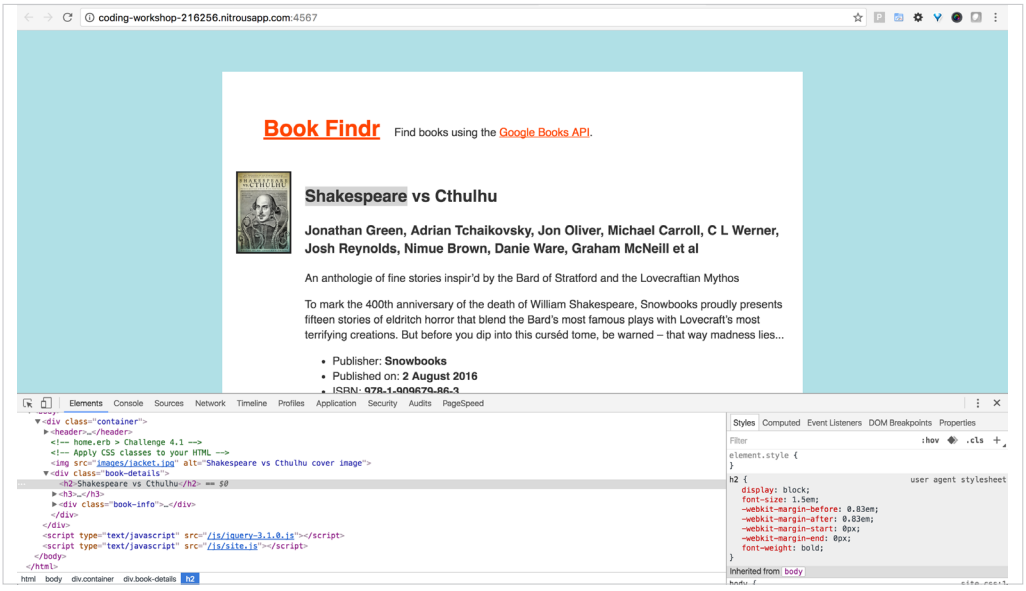


The browser is displaying a web page using the HTML you wrote. **All web pages are made from HTML.**

Let's prove it. Open the `views/home.erb` file and in the code, find the line that reads

```
<h2>Shakespeare vs Cthulhu</h2>
```

Now see if you can find those words on the web page: on the browser, hover your mouse over the words "Shakespeare vs Cthulhu". Right click your mouse, and choose 'Inspect'. A pane will open in your browser, showing the HTML that's generating the page. It's the same as in your `home.erb` file.



Challenge 5.1: Change the web!

Go to any web page -- theguardian.com, bbc.co.uk, twitter.com, anything you like. Hover over any text, then right mouse and Inspect. Edit the HTML in the Inspect pane by clicking and typing over some words. See what happens in the browser.

Result:

Editing the HTML changes what's displayed on the web page. **Web pages are made from HTML.**

What you've learned

- Static websites are one sort of website. Another is dynamic.
- The web pages that we see when we're browsing the internet are made from HTML.
- They are styled using CSS.

6: A dynamic website

Making it dynamic

So at the moment you have a static website. However, a second, and more sophisticated, type of webserver can understand bits of code embedded in an HTML file. Have you ever run a mail merge from Microsoft Word? You add in mail merge tags such as <firstname> and <lastname>. You can do the same thing in HTML. There are still lots of pre-prepared HTML files, but included in amongst the HTML will be Ruby tags. These Ruby tags are code commands; they instruct the webserver to perform a task. In most cases the Ruby commands will instruct the webserver to fetch some information: for instance, today's date. So you place the Ruby tags in amongst the HTML in the exact spot you want the date to go and the last thing the webserver does before sending out the file is to replace it with the actual current date.

(The example above uses made-up tags. In reality, the format and syntax will depend on which of the many types of webserver you're using.)

Pieces of information that stay the same from day to day (e.g. the words in an old blog post) can be written directly into an HTML file. But if you want users to see information that keeps changing (e.g. the total number of visitors to your site) you can embed code like this instead. A visitor to the website never gets to see that code, though, because it's always been replaced before the file is sent to your browser.

You can think of these Ruby tags as being a question. Before an HTML files gets sent out, all those questions are replaced with their HTML answers.

Databases

The ability to swap in data dynamically becomes even more useful if your website stores its content in a database rather than in lots of separate HTML files. If you want to offer 100 web pages, one for each book you've published, you don't need to create 100 HTML files. You can design one web page as a 'template' and use it for all 100 books, drawing their data from where it's stored in the database. For instance, the heading in your template could be made up of one Ruby tag for 'author name' and another Ruby tag for 'book title'. Those Ruby tags will be replaced with the relevant info, depending on which book's web page the user is requesting. You can think of a database as a bit like a spreadsheet, where the sheets can talk to each other, matching their data through identical keys.

APIs

Instead of a database, you can use remote sources of information. An API is a list of things you can ask a service to give you, or to do. Companies often expose an API to their systems, to be used and interacted with by its customers. You're going to be using the Google Books API, made available by Google over the internet, accessed by the search terms that you give it.

Why Ruby?

We have chosen to show you Ruby during this course because it is an elegant language. William Strunk, Jr. would be delighted with Ruby:

Omit needless words.
Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.
Strunk and White, The Elements of Style, 1918

As computing power has increased over the last few decades, we can have programming languages optimised for humans, not computers. Computers are powerful enough nowadays to allow Ruby methods to be written in clear English, which take longer to process than some terse machine code. Compare these two ways to get all the books. First, the Ruby approach:

```
Book.all
```

Ruby is quite readable. `Book.all` means “get all the books”. Now, the PHP way (PHP is another programming language often used in web development):

```
$this->Book->all();
```

Which language would you rather use?

Why Sinatra?

For your app, you’re going to use Sinatra, which is the next step up in terms of sophistication. Sinatra is a web app development framework, giving you the basic structure of a web app out of the box, so as a programmer you can jump to the fun stuff more quickly.

Sinatra is a more simple framework than Ruby on Rails. It is still written in Ruby, though. Other languages have other frameworks: PHP has Symphony, for example.

Ruby tags

You're going to turn your static website into a dynamic one, using Ruby tags and more sophisticated HTML.

Your first challenge is to replace the static HTML in the `home.erb` file with Ruby tags that will allow any book's data to be displayed.

Syntax

Syntax refers to the symbols and words that make up a programming language. Use the correct syntax and the computer will know what you want it to do.

The syntax for a Ruby tag is this:

```
<%= book.title %>
```

So that's a `<%=` at the start of the Ruby tag, and a `%>` at the end. Anything between those characters will be interpreted as Ruby code. The Ruby code will run, and the equals sign at the beginning will mean the result is printed to the screen. You'll get a piece of data, such as the book title "Shakespeare vs. Cthulhu", appearing on the page.

Attributes

Programs often get structured using the same names as you'd find in the real world[3]. In a book metadata app, it's likely you'd have a Book object.

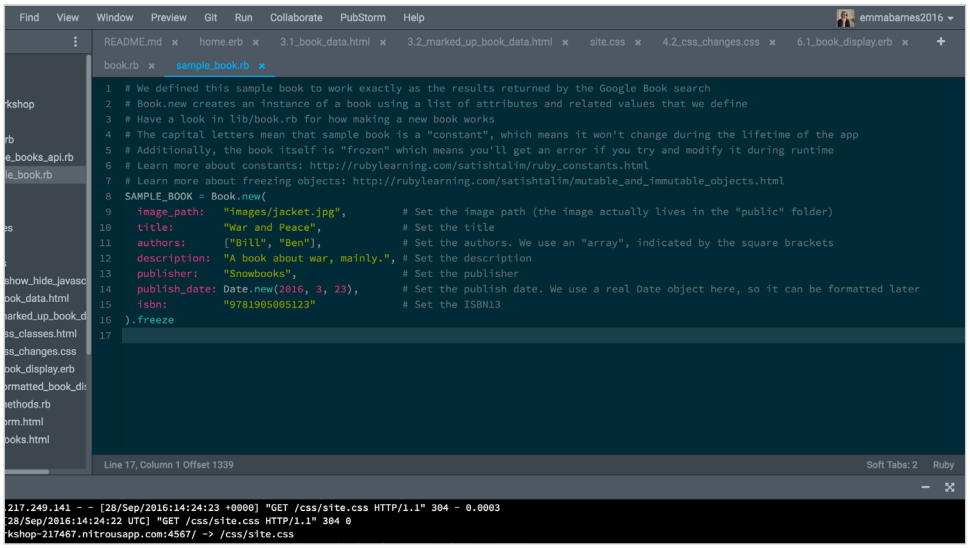
Following this pattern, attributes are the things that an object has -- both in the real world and in our program. A book object is likely to have an ISBN attribute, and a title

3 See Appendix: Object Oriented Programming

attribute. Here is a list of attributes you can use in the Book Findr app.

Book attribute	Example
book.image_path	/public/images/jacket.jpg
book.title	Shakespeare vs Cthulhu
book.authors	[Jonathan Green, Adrian Tchaikovsky, Jon Oliver, Michael Carroll, C L Werner, Josh Reynolds, Nimue Brown, Danie Ware, Graham McNeill]
book.description	An anthologie of fine stories inspir'd by the Bard of Stratford and the Lovecraftian Mythos. To mark the 400th anniversary of the death of William Shakespeare, Snowbooks proudly presents fifteen stories of eldritch horror that blend the Bard's most famous plays with Lovecraft's most terrifying creations. But before you dip into this curséd tome, be warned – that way madness lies...
book.publisher	Snowbooks
book.publish_date	2016-08-23
book.isbn	978-1-909679-86-3

In a while, you are going to query the Google Books API as a data source, but for now we have prepared a bit of sample data for you to use. Open `lib/sample_book.rb`. You'll see some sample data -- the title is *War and Peace*, the publisher is Snowbooks, and so on.



Challenge 6.1: Replace static text with Ruby tags

Use the Ruby tag syntax to replace static text in **`home.erb`** with the dynamic attributes in the table above.

Look in the snippets folder for a file called **`6.1_book_display.erb`**. But try not to copy and paste. Ruby's tag syntax is `<%= ... %>`

Optionally, in the **`lib/sample_book.rb`** file, try amending some of the data and seeing what happens when you refresh the browser.

Once you've completed Challenge 6.1, refresh your browser. You'll see dynamic data has replaced your hard coded HTML.

What you've learned

- Databases and APIs can be data sources.
- Ruby is an elegant object oriented language.
- Objects can have attributes.

7: Ruby methods

Methods describe the behaviours of an object in Ruby. You've been considering a book object: you're going to define methods that you'll use to get your book to behave how you want.

A method to format a date

In our interface, the date could be formatted in a more appealing way -- rather than '2016-05-12' it could be '12th May, 2016'.

Ruby has a method for formatting dates, called `strftime.`

Challenge 7.1: Format the publication date

Look in the snippets folder for `7.1_formatted_book_display.erb`. and type the method you see into the relevant place in `home.erb`.

If you have time, open a new tab and Google how Ruby handles dates. Much of a programmer's time is spent Googling for answers.

Methods

You'll have noticed by now the dot:

```
book.publish_date
```

The Ruby dot is how you ask an object for some of its information. This is also know as "calling a method" in Ruby. Here you'd say you're 'calling publish_date on book'.

`publish_date` is something that a `book` has. The syntax:

```
book.publish_date
```

is the same as saying "Book, please tell me your publish_date". You can do the same for title, isbn -- in fact any attribute that `book` knows about.

Ruby comes with a lot of methods already written such as the `strftime` method. You can increase the number of things that a book knows about by writing methods. Here's a method that figures out the stock delivery date of a book that's been printed:

```
def delivery_date
  publish_date - 1.month
end
```

We can get that date by calling

```
book.delivery_date
```

Here's a method that puts the title into capitals:

```
def title_in_caps
  title.upcase
end
```

This will result in the title being displayed in all capitals: SHAKESPEARE VS. CTHULHU, for example.

Challenge 7.2: Methods

Look in the snippets folder for a file called `7.2_methods.rb`. You'll see a method defined, called **uppercase_title**.

Open the `lib/book.rb` file and on line 40 type in this method to transform the title into uppercase. Try not to copy and paste.

Then use the method in your view code by using the Ruby dot to call your method on book. Your view code is in the file called `views/home.erb`. You should replace the code `book.title` with `book.uppercase_title`

If you want a further challenge, try writing more methods to transform the title. For instance, can you define a method called "rollercoaster" where every other character in the title is uppercase?

What you've learned

- Many modern programming languages are object-oriented.
- In Ruby, a dot means 'call a method on an object'.
- You can use methods to increase the number of things an object can know about or do.

8: An API data source

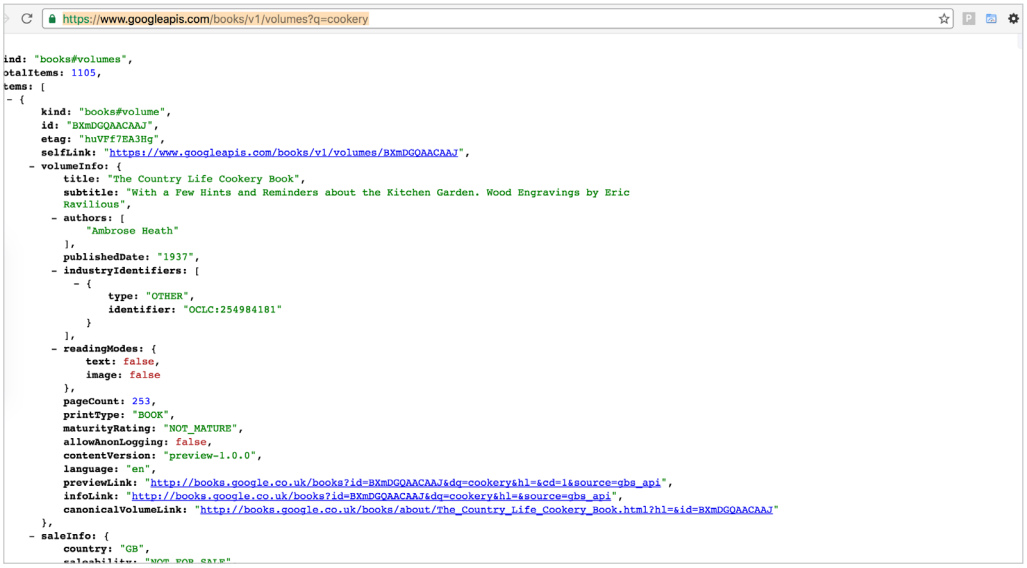
Your app displays one book so far, and its data comes from a static source in `lib/sample_book.rb`. Next, you're going to use a different data source: the Google Books API. API stands for Application Program Interface. An API gives you the rules around how you can interact with a program: what you can ask for, and what you can do.

Google Books API

What does the response to a call to the Google Books API look like? In a new browser window, put the following into the address bar:

```
https://www.googleapis.com/books/v1/volumes?q=cookery
```

You'll see something like this:



What you've done there is query the API, asking it to give you everything tagged as "cookery".

This is data in the JSON format. It's quite human readable, but it's particularly readable for computers, and is the default data format of modern APIs. You can also get the same data in the XML format, which you might have heard of in relation to the ONIX metadata standard.

You're going to add a form to your app which will do this sort of query, but through a form. Your app is going to interpret the results and display them nicely.

Challenge 8.1: Add a form to query the API

Add a form into `layout.erb` (**not** `home.erb` this time)

To do that, look in the snippets folder for a file called `8.1_form.html`

Copy the contents into line 15 of `layout.erb`.

Refresh your browser and see if you have a form input field. Type a book title or subject in and click Search. Then go to your command line: it will show the 'stack trace' which is a log of all the code that runs on the server. You'll see something like this:

```
172.17.42.1 - - [10/Sep/2016:18:20:36 UTC] "GET
/?query=you+are+the+hero HTTP/1.1" 200 1514
```

For your purposes, you want to **get** some results from Google's API, and display them. So you'll be using the GET HTTP verb – and sure enough, that's the word that you see in the stack trace:

```
172.17.42.1 - - [10/Sep/2016:18:20:36 UTC] "GET
/?query=you+are+the+hero HTTP/1.1" 200 1514
```

Find out more about HTTP in the Appendix.

The lifecycle of a request

How does this work? Just like in the diagram you saw earlier.

- You type a query into the form e.g. “You Are The Hero”.
- When you submit the form, the browser sends the string “You Are The Hero” to the code in your app in `site.rb`
- You see it being sent in the stack trace as a GET request:

```
172.17.42.1 - - [10/Sep/2016:18:20:36 UTC] “GET
/?query=you+are+the+hero HTTP/1.1” 200 1514
```

- `site.rb` uses the query in the method it runs every time the home page of the app loads:

```
books = GoogleBooksApi.new.search_
volumes(params[“query”])
```

or, in other words:

```
books = GoogleBooksApi.new.search_volumes(“You Are
The Hero”)
```

- The code found in `lib/google_books_api.rb` runs to set the `books` variable to contain the query results.
- You loop, or iterate, through `books` in `views/home.erb` to display your query results. (You’ll be writing the bit of code that iterates through `books` next.)

What you’ve learned

- An API is the way you are allowed to interact with a program. If you query the Google Books API, it responds with a set of data, in the JSON format
- API calls tend to be made over the internet
- The internet works by web servers passing around requests.

9: Displaying data

You have written the code which runs a query, but you've not seen any change in what your browser is displaying -- it's still displaying hard-coded sample data. You need to write code which displays the results of your queries.

In the next challenge you will wrap the code in `home.erb` in the code from `snippet/9.1_books.html.` Navigate to that file now.

Variables

Variables are used to label and store information. You can define a variable using an equals sign, like this:

```
books = Book.all
```

Look at the line

```
  <% books.each do |book| %>
```

`books` in that line is a variable which holds the results of your API query. The `books` variable is set in `site.rb` when this line of code runs:

```
books = GoogleBooksApi.new.search_
volumes(params["query"])
```

The `books` variable

Wouldn't it be nice to be able to see what **`books`** is so you know how to handle it? You can, if you use the command line to run an IRB session. (IRB stands for Interactive Ruby Shell.)

Add a new command line window by clicking on the plus sign:

Then from the **Run** menu, choose 'Interactive Ruby'. Once the command line shows this prompt:

```
irb(main):001:0>
```

then type in

```
books = GoogleBooksApi.new.search_volumes("bob")
```

and press return. The query runs and returns something like this:

```
=> [#<Book:0x0055708ce00108 title="Reading
Magic", authors=["Mem Fox"], publisher="Macmillan
Publishers Aus.", publish_date=#<Date: 2012-11-01
((2456233j,0s,0n),+0s,2299161j)>, description="If
parents understood the huge educational benefits
and intense happiness brought about by reading
aloud to their children, and if every parent
— and every adult caring for a child — read
aloud a minimum of three stories a day to the
children in their lives, we could probably wipe
out illiteracy within one generation. Mem Fox
lives and breathes by these words. As well as
being a bestselling author of children's books,
she has been a passionate supporter of reading
aloud to children for decades. The latest edition
```

```
of this classic bestseller has a brand new
chapter — a chapter that follows Mem's delightful
discoveries, and her joy, when she begins to read
aloud to the next generation of her family: her
grandchild. With her usual humour and insights,
Mem demonstrates the power of literacy by both
her actions and her words.", image_path="http://
books.google.ie/books/content?id=OuQLHDh_
LBIC&printsec=frontcover&img=1&zoom=1&source=gbs_
api", isbn="978-1-74328-525-1">]
```

If you format that differently, it's easier to see what's going on.

```
=> [#<Book:0x0055708ce00108

title="Reading Magic",
authors=["Mem Fox"],
publisher="Macmillan Publishers Aus.",
publish_date=#<Date: 2012-11-01 ((2456233j,0s,0n),+
0s,2299161j)>,
description="If parents understood the huge
educational benefits and intense happiness brought
about by reading aloud to their children, and if
every parent — and every adult caring for a child
— read aloud a minimum of three stories a day to
the children in their lives, we could probably
wipe out illiteracy within one generation. Mem Fox
lives and breathes by these words. As well as being
a bestselling author of children's books, she has
been a passionate supporter of reading aloud to
children for decades. The latest edition of this
classic bestseller has a brand new chapter — a
chapter that follows Mem's delightful discoveries,
and her joy, when she begins to read aloud to the
next generation of her family: her grandchild. With
her usual humour and insights, Mem demonstrates
the power of literacy by both her actions and her
words.",
image_path="http://books.google.
ie/books/content?id=OuQLHDh_
LBIC&printsec=frontcover&img=1&zoom=1&source=gbs_
api",
isbn="978-1-74328-525-1"
>]
```

This Book object is like the array you saw earlier, except instead of being a list of things, it's a list of named, labelled things. You don't have to guess what "Reading Magic" is – you are told that's the title.

Iteration

`books` doesn't only contain one book's worth of data – if you look on your command line you'll see there are many. In your app, you need to display each of those books in turn.

Doing that is called **iteration**. You loop through the collection of books, printing each one out, with each formatted the same, cookie-cutter style. The Ruby code which allows you to do that loop is

```
<% books.each do |book| %>
```

Ruby is nicely readable: read that line aloud. It says "books each do book". More verbosely, it means "For each book in turn, replace the contents of the `book` variable with information about the next book in the list".

Conditional logic

You're well on your way to successfully creating a dynamic web app. But now that you're working with dynamic data, things can get tricky. What happens, for instance, if your query doesn't return any results? What will happen with our code?

If the query is missing, then the app will throw an error. We can avoid that happening through writing some conditional logic.

We'll get to the syntax for it in a moment, but for now it's enough to write some pseudo code. This is what we want:

```
If the query exists
  then show the results of the query
otherwise don't show anything.
```

Now let's write that in Ruby code:

```
<% if query %>
   <!-- all the results of the query -->
<% end %>
```

If you have a beady eye (and what publisher doesn't?) then you'll note that the Ruby syntax is a little different here. You have `<% ... %>` Ruby tags surrounding the commands. There is no `=`. That's because you don't want anything to print out on the web page. This syntax means "evaluate the Ruby code but don't print the outcome in the HTML".

You'll also notice that you have an `<% end %>` Ruby tag. You need to add this Ruby tag in to mark the end of your piece of code.

Display the results

So go ahead and do the next challenge so you can use the `books` variable.

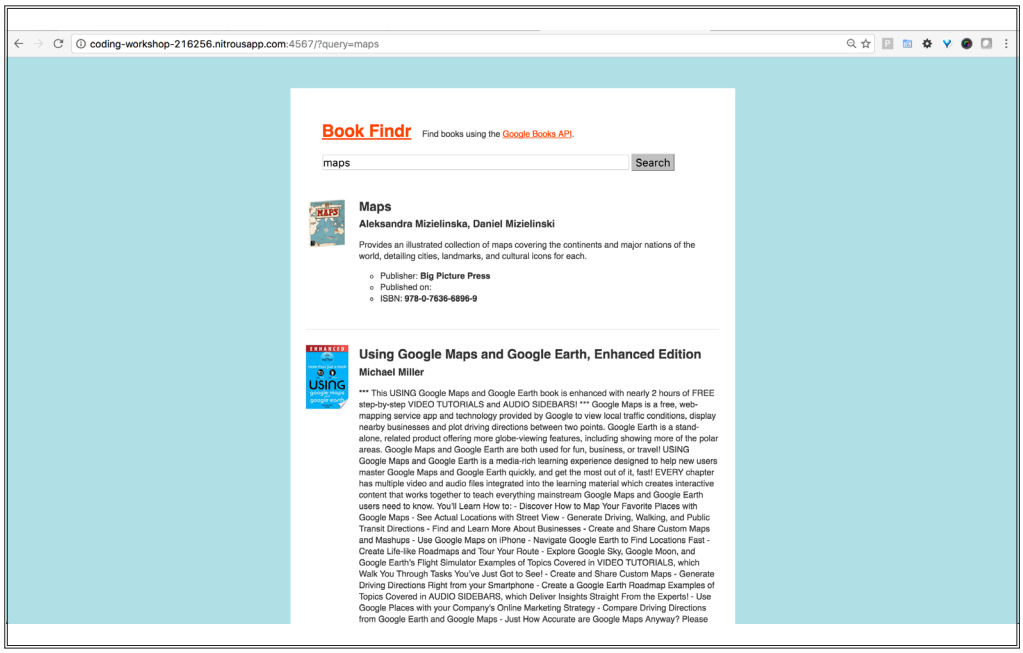
Challenge 9.1: Display the results

Add code to `home.erb` to iterate through the `books` variable and display each book.

To do this, look in the snippets folder for a file called `9.1_books.html`.

The comments explain which bits of the code in that snippet should go where.

When you're done, refresh the browser. You'll get something like this:



What you've learned

- Ruby tags can represent complex data, and more than one instance of an object.
- You can use IRB to run bits of code outside of your app.
- The `each` method lets you iterate through each instance of a book in the `books` variable.
- Conditional logic can be used to optionally display information.

10: JavaScript

The page looks good, but there are a lot of books to scroll through, and they look inconsistently spaced. The next programming you're going to do will tuck all the metadata into a show/hide area, to keep the page clean when it loads. To do that you're going to use some JavaScript.

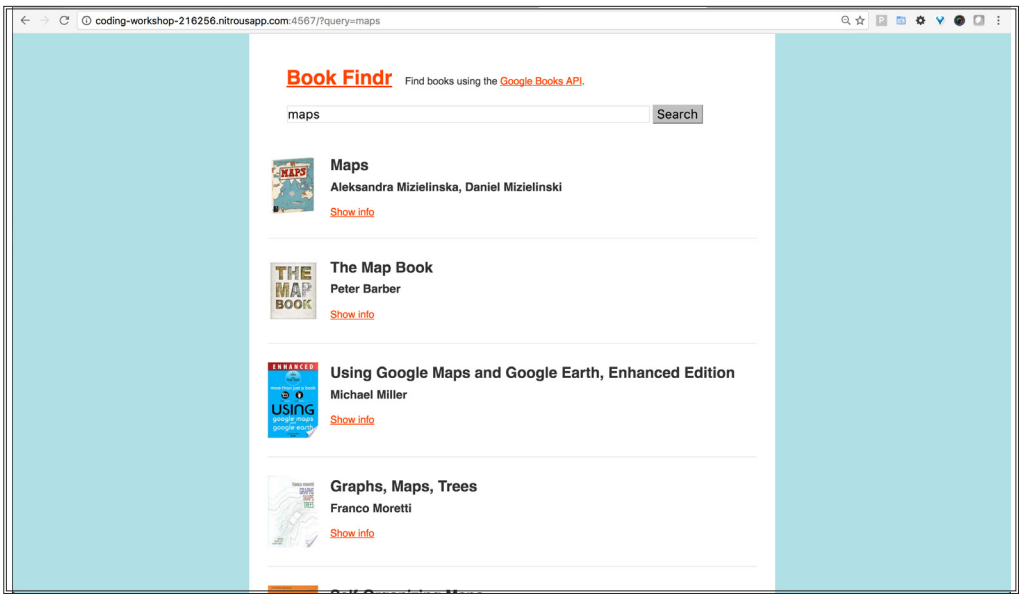
Challenge 10.1: Add a show/hide feature

Add code into `public/js/site.js` to show and hide the data.

To do this, look in the snippets folder for a file called `10.1_show_hide_javascript.js`. Copy and paste that code into `public/js/site.js`

NB: Nitrous sorts the file names as strings, not numbers, so 10.1 comes at the top of the list!

When you're done, refresh your browser. You'll see that you magically have a show link. Click on it, and your data scrolls into view. Pretty!



Client side vs server side

Browsers have built-in interpreters that read any JavaScript code in the page, and run it. By including your code in the JavaScript folder, it's included in the page. Look in `views/layout.erb`: at the bottom of the file you'll see the line:

```
   <script type="text/javascript" src="/js/site.
js"></script>
```

That's how the code in `site.js` gets included in the HTML page on your browser. src means "source".

The interesting thing is that you don't have to click a button or send any data to the server to make the JavaScript run -- the show/hide functionality works as soon as the page is loaded. That is different to what happens when you click the search button -- that sets off the round trip to the server that we looked at earlier.

What you've learned

- JavaScript works in the browser without you having to reload the page.

11: Next steps

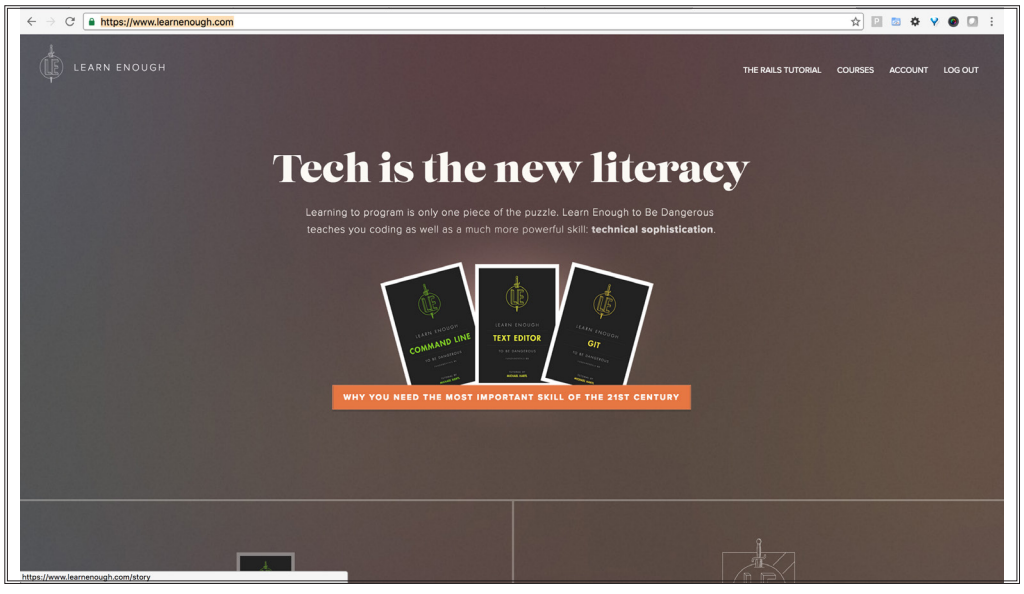
You've successfully developed your Book Findr app. Jolly well done!

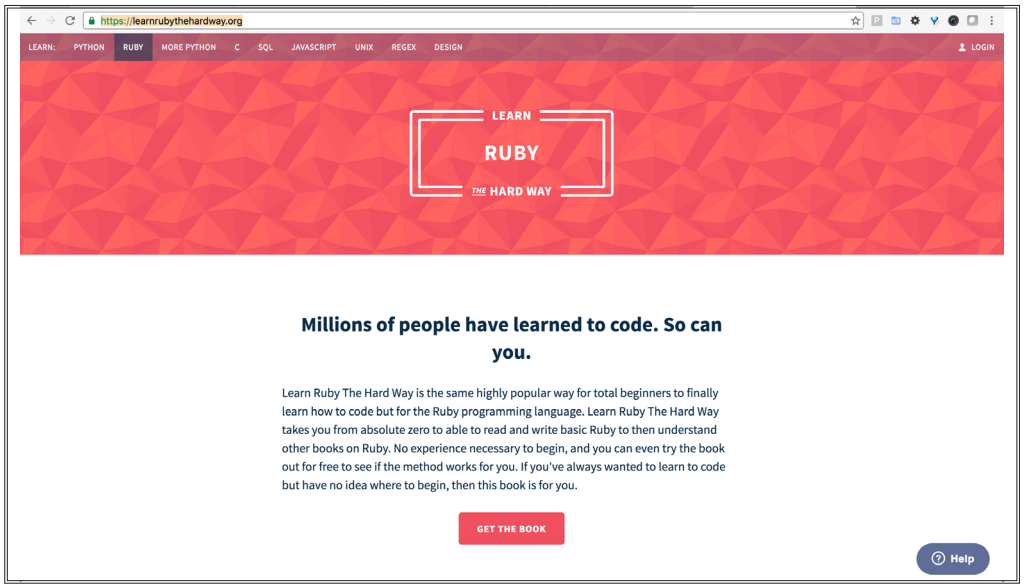
Has the bug bitten you? It's normal at this stage to feel overwhelmed, tired, brain dead and a bit weepy. You've had a tantalising, whistle-stop tour of a brand new set of concepts, and you might have a feeling of both elation -- you did it! -- and also an awareness that there is a lot to learn in programming.

If you feel that you would like to learn more, here are some good next steps to take.

Introverts' route

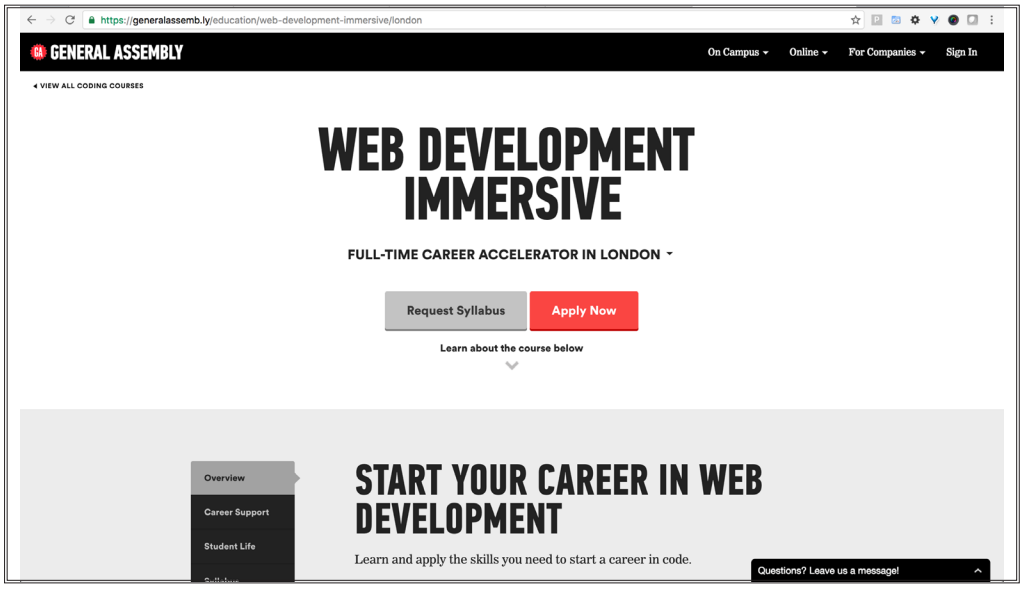
If you are an introvert, you're most likely to enjoy learning over the internet, from books and from videos. We recommend `https://www.learnenough.com/` as a great place to get started, followed by `https://learnrubythehardway.org/` and `http://railstutorial.org`





Extroverts' route

If you prefer learning alongside other people, then Codebar is a great and welcoming event that happens frequently in London, with a new chapter opening in Oxford on 11th October. Railsgirls London is also another good taster day and the Bibliocloud team volunteer there. General Assembly and Makers' Academy run crash courses, but they're not cheap.



Coding in publishing

How can you start to apply what you've learned today?

The JSON data that you received from the Google Books API today is a good example of structured book data. The same sort of structured approach can be found in ONIX, the XML standard that the book trade uses to share metadata. You could explore ONIX XML as well as JSON. You could extend the functionality of the app that you've written today to present more of the information accessible from the Google Books API and turn it into a selling or reader discovery tool. You could find that you want to learn more about Ruby, and web development frameworks, and write your own dynamic websites.

Whichever way your interests take you, the more knowledge you accrue, the more power you have. Any step towards demystifying code gives you power in third party relationships, on any digital project. You might find that the bug really bites you and that you want to delve deeper into programming -- but any technical knowledge improves your position. You don't need to be a professional programmer, but programming knowledge will make you a wiser project manager, product manager, marketer, designer and reader advocate. You will know what's possible.

Stay in touch!

Do stay in touch with us! We blog at www.coderead.co, and tweet at @bibliocloud: our tweeters are Emma @has_many_books, Emily @emilylabram, Andy @andypearson, Tik @tikdalton, and Dave @databasesponge

Please do give us feedback on this course at:
https://www.surveymonkey.co.uk/r/639YNPX

And if you need a modern, cloud-based publishing management system, you know who to ask!

Visit http://bibliocloud.com to find out what we do.

12: Appendix

Object Oriented Programming

Ruby is an object oriented programming language. One of the most useful things about that is that Ruby programmers often design their programs to mirror the real world. In publishing, for instance, we have a lot of books. And so in a Ruby program to do with publishing, it's likely that you'd create a Book object. Just like in the real world, a Ruby Book object has attributes such as a title, a publication date, a page count and an author.

Nowadays, because computers are really fast, programming tends to favour programmer happiness – naming objects and attributes in a way that makes most sense to the programmer, rather than working in a way most efficient for the computer. So in your program you'll be choosing words to describe your objects and attributes which are meaningful and easy for another programmer to understand.

It's for this reason that you've got a huge advantage over a programmer who might be really good at her craft, but who doesn't know the business domain of publishing. If you have a decent understanding of how publishing works, you're in a great position to design a decent publishing program.

HTTP requests

Sinatra, the web framework you're using, uses a combination of HTTP requests and route descriptions to show, create, destroy and update objects.

HTTP stands for “hyper text transmission protocol”. In other words, it’s the basis for how the whole internet works. It’s amazing how basic the whole set up is, considering how complex the internet appears to be.

For example, every page that loads on every web page is “stateless” – that is to say, every time a webpage loads, it starts from the point of view of knowing nothing about what came before. So as programmers we have to pass around a lot of data, constantly reminding the server as to what we know the current state of things to be. Each request assumes nothing.

And regardless of the complexity of a website, most actions that happen when you refresh the browser -- through clicking on a save button, or running a search, or submitting a form -- rely on just four main actions.

The four HTTP verbs that pretty much every website uses are:

POST	This is a request to create an object, usually in a database
PUT	This is a request to replace an object
GET	This is a request to display some records fetched from a data store
DELETE	This is a request to delete a record

Code comments

The code in the Book Findr app is commented. Use these comments to discover more about how the application works. There are also useful links for you to use the comments as a springboard to explore further.