TRY PROGRAMMING

FOR PUBLISHERS

Learn • Get inspired • Develop • Publish

A GENERAL PRODUCTS LTD PUBLICATION

Try Programming for Publishers

A book to accompany the course commissioned by the Oxford Publishing Group

From the makers of Bibliocloud

Contents

Preface

This course is written and run by General Products Ltd, the makers of Biblio-cloud.com. It leans liberally on the Rails Guides which would be the best place to start after this course.

Other good resources:

The Rails Tutorial by Michael Hartl

The Odin Project

Railsgirls

Rails for Zombies

Try ruby

Chapter 1

Getting started

1.1 A recent history of code development.

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.

Here is a list of notable languages.

In this course well be using a language called Ruby, which has been around for about 20 years. Ruby became very popular as a language for developing web-based applications about ten years ago, after the release of an application development framework written in Ruby, called Ruby on Rails. The aim of Ruby on Rails was to make it very easy to develop web based applications by hiding nearly all of the complexity from the developer.

Over the past ten years, many more people have contributed further to the ease of developing Ruby on Rails applications by contributing more open source code to help with specific common tasks. These contributions, known as gems, help with tasks such as providing password-based access to systems, validating ISBNs, or allowing users of the system to upload files such as images or documents through the application.

So the trend over the past decade has been to make it easier and easier to solve common tasks by just using code that other people have contributed,

making it more simple to write sophisticated applications.

When the application has been written, it used to be the case that you then had to buy or rent a server that is connected to the internet, and then install and maintain all of the required software and security configuration. However another major change has been a recent profusion in companies, such as Amazon and Google, offering rental of their own computers for running code and managing applications. Other companies, such as Netflix, run their businesses entirely on rented servers. Furthermore, yet other 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 year or two, other companies have developed hosted systems that allow you to develop your code on their servers, so you dont even need to have a sophisticated and complex development environment on your own desktop or laptop computers. Today we will be using a service called Nitrous to develop our code.

1.2 Set up a Nitrous.io account

- Go to https://www.nitrous.io/
- Enter a username.
- Now enter your email address.
- Enter a password.
- Hit “Sign up for free”. If any of your details aren’t up to scratch, have another go.
- You’ll have to log in to your email to confirm you want to sign up. Do that and click on the link to confirm you want an account.

- Then you can sign in. Enter your email or username, and password.

1.3 Create a development box

- Click on the big green button that says **Open dashboard**.
- You'll see a panel that says **Create your first box**. Ruby on Rails will be highlighted. Leave everything as it is, and click the orange **Create Box** button.
- On the next screen, wait until the line at the bottom goes green and it says [your box name] is running. Click **Next**. On the next page, scroll down and click **Okay, Take me to my box**.

1.4 Logging back in

If your login expires, you can log back in with your user name and password.

Click on **Start** and then **IDE** (which stands for Interactive Development Environment).

1.5 A look around Nitrous

At the top of the page, the screen is divided into two. On the left is a file system navigator – just like youd see in Finder on the Mac or Windows Explorer on a PC.

On the right is the contents of whichever file is open on Nitrous.

At the bottom of the page you have a console. Its the same as what youd find if youve 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.

1.6 Create a Ruby on Rails application

Oh my! We're ready to create your first Ruby on Rails application. But what does that mean?

Box 1.1. What is a Rails application?

When you go to the Scribd website, or the Yellow Pages website, or Airbnb, Groupon or Bloomberg, you're looking at Ruby on Rails applications.

When you run the code of a Rails application, it generates HTML pages. Unlike a static HTML website, though, a Rails app can display different data, drawn from a database, depending on what the website user clicks on and types.

So: a Rails application generates HTML pages on the fly. And what are a bunch of HTML pages usually called? A website.

Lets set up our new Rails app in the correct folder on Nitrous. This next command is the same as going to Finder and clicking on a folder. But youre a programmer now, so you can use the command line. cd stands for change directory. In the console, click near the $ dollar sign, which is known as the command prompt, and type:

```
cd workspace
```

The console command prompt changes to show that youve changed into the directory folder called workspace.

You know when you install an app or a new software program that you've download onto your phone or computer? You need to do the same for the database program that we'll be using. We'll be using PostgreSQL for our database. To do that, run this command:

```
parts install postgresql
```

Words will appear in your console, whilst the installation is going on. Once PostgreSQL is installed, you'll be able to see the $-sign console prompt again. Now, you can start the PostgreSQL program. Run the following command:

```
parts start postgresql
```

We'll call our Rails app "Book Organiser". What we're aiming for is to build a website that will appear somewhere like **www.bookorganiser.com**. To create your new Rails app, run the following command:

```
rails new book_organiser -d postgresql
```

That's our database program installed and running, and our Rails app created, and we've only typed three commands! Next, we need to provide the correct username and password, to connect your new Rails app to a PostgreSQL database. Switch to the directory containing your new app:

```
cd book_organiser
```

Now, find the file at **workspace/book_organiser/config/database.yml**. Click on it, and its contents will appear in the right hand pane. Delete everything in this file. Now, copy all the following text and paste it into the file, to replace everything that was in there.

```
development:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_dev
  pool: 5
  host: localhost
  username: action
  password:
```

```
test:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_test
  pool: 5
  host: localhost
  username: action
  password:

production:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_prod
  pool: 5
  host: localhost
  username: action
  password:
```

Once you've saved the file, go back in the console, and run the following command:

```
rake db:create:all db:migrate
```

OK! We've installed PostgreSQL, created a new Rails app, created a database and connected our app to our database. Were ready to start up your new Rails app. Run the following command:

```
rails server
```

Later, you can stop the server with Control-c to return to the command prompt.

Then on Nitrouss navigation menu at the top of the page, click on **Preview > Port 3000**. You'll see a website declaring "Welcome aboard. Youre riding Ruby on Rails!"

Youve done it! Your new Rails application is running.

Chapter 2

Structuring our app

2.1 About the web application

Today we are going to build a little web application, but first we have to think about exactly what that means - what do we need to do in order to make that happen?

There are a few basic elements.

Firstly, we need to have a page in an internet browser that will allow users to type information into forms, just as if they are writing an email in GMail or Yahoo! Mail. The pages need to be sent to the browser as HTML. Users need to be able to click on a button that will make the web browser send the data they have typed off to the application, which is on a remote server somewhere in the world waiting to be told what to do.

Then the application needs to work out what to do with the information. Maybe you are adding new data, maybe you are deleting data that you no longer need, and maybe you are changing data.

The data also needs to be saved somewhere, by something that will not only remember it, but will also allow it to be found again.

Lastly, we need to be able to see web pages that show the information that we have previously stored.

There are quite a few different things that have to be done in there, and the Ruby on Rails framework provides different components that specialise in doing

different tasks.

- Views, which create the HTML pages that are sent to the browser.
- Controllers, which work out what to do with information sent from the browser, and which views then need to be used to send pages back again.
- A database, for storing and retrieving information.
- Models, which tell the Ruby on Rails system how to store and modify data in the database.

2.2 About the design

We are going to build a little system for managing publisher metadata. So what do we need to store information about?

The world is full of stuff. Books, cats, football matches, emails, office buildings, submarines, string quartets, rock concerts. Some of them are actual physical things, some of them are abstract, but all of them can be thought of as some kind of object.

Each of the above examples of stuff has a bunch of information associated with it.

- What time will the performance begin? 8:30
- How deep can the submarine go? 1,000 ft
- Whats the name of your cat? Rosemary
- Who did you send the email to? The prime minister
- Whats the ISBN of that book? 978-0205309023

These are all attributes of the object.

Each of the above stuffs has things that you can do to them.

- The performance can be cancelled.
- The submarine can be painted
- The cats name can be changed to Ambrose
- The email can be sent
- The book can be bought.

If were going to build a publishing management system, what sort of objects do we need?

Publishers make books, so we'll need to describe them. We'll try to use words that are as precise as possible, so if any other developers come to work on our code, it'll be easy for them to figure out what our app does.

The first model we'll create will be called "Work", and it will have a title, a description, a subject and a cover image.

The second model will be called "Product". We could call it "book", but that wouldn't be very helpful if we want to store information about apps, or maps, or marketing materials, or other non-book products, in our system. The word "book" can also mean "individual copy" which would be misleading. Our products will have a format, a price, an ISBN, a publication date and a page count.

Let's say that we have two products. The first is a paperback, published in June 2015. The second is a hardback, published in March 2015.

Both these products will belong to a work, whose title is "War and Peace Revisited". The work's subject is "fiction". The title and the subject from the work will be inherited by the products. We don't need to repeat them.

Box 2.1. Principle: Normalisation

We don't store the same data more than once in a database.

2.3 Create our first model

We'll keep the server running so we can see our changes as they happen. So let's open a new console window. At the bottom of the Nitrous page, click the + sign next to Console. Type the following to switch to your app's folder:

```
cd workspace/book_organiser
```

Now we can run commands which our Rails app will understand. Type the following:

```
rails generate scaffold work name:string description:text cover:string subject:string
```

This has created a new file with some code in called a migration. When we run this migration code, it'll tell the database to set up a new table, with a column called "name", a column called "description", a column called "cover" and a final column called "subject".

The database will expect most of the columns to contain data that is a 'string': short pieces of text up to 256 characters. The description column, however, will contain pieces of text with no length limit. Telling the database what sort of data it should be expecting makes for good quality data. For example, if the database was expecting a date, but a user tried to put in a string – say, "Monday", or "Next week" – the database wouldn't let the user save the data and would insist on an actual date.

Type the following to run the migration code:

```
rake db:migrate
```

Rake is a tool you can use with Ruby projects to run tasks in the console. In this case, we're using it to run all the database migrations that haven't yet been run.

Now you can have a look at your work so far. Go to the browser and type **/works** after the address in the URL.

2.4 Make this our home page

At the moment, “Welcome aboard” is occupying our important home page spot. Let’s change things so that our works page shows up when we go to our website’s root.

Box 2.2. Model View Controller

Rails likes to keep everything in the right place. It helps developers enormously to know that if you open up someone’s Rails app you’re going to know where to go to find the html.erb view files, where to go to find the database migrations, where the business logic is, where the stylesheets and javascript files will be, and so on. The structural conventions that Rails uses saves developers time – and therefore money – every day.

The structure that Rails apps use follow the MVC pattern (Wikipedia page).

Routing decides which controller receives which requests. A **controller’s** purpose is to receive specific requests for the application. A controller contains methods called actions, and each action’s purpose is to collect information to provide it to a **view**.

If you use the scaffold, like we did, then Rails controllers come with the following actions. Open up **app/controllers/works/works_controller.rb** as well to follow along.

- index
- show
- new
- edit
- create
- update
- destroy

For the first four of these actions, there is a corresponding view file. Have a look in **app/views/works** and see. The final three are triggered when a user sends an instruction from the view – for instance, to delete a record. In the case of a 'delete' instruction, the **destroy** controller action deletes the record then sends a message back to the relevant view, saying the record was deleted OK. There's no separate view for such an action.

A **view's** purpose is to display the information from a controller in a human readable format. An important distinction to make is that it is the **controller**, not the **view,** where information is collected. The **view** should only display that information.

(adapted from the Rails Guides).

We used the Rails scaffold to create all the model, view, controller and routing files at the same time as the database table. You can create them one by one, as well.

Open the file **app/config/routes.rb** in Nitrous.

This is your application's routing file which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions. This file contains many sample routes on commented lines. Add the following in, on the second line of the file:

```
root 'works#index'
```

root 'works#index' tells Rails to map requests to the root of the application to the works controller's index action. **resources :works** tells Rails to map requests to http://localhost:3000/works/index to the works controller's index action. This was created earlier when you ran the generator (**rails generate scaffold work**).

Navigate to your home page in your browser. You'll see the works index page there, indicating that this new route is indeed going to the work controller's index action and is rendering the view correctly.

Read more about routing in the Rails Guides.

Chapter 3

Making it look good

3.1 Adding some styling

Your new **works** page is a bit ugly, to be honest, so let's add some styling. There is a free, widely-used design resource called Twitter Bootstrap. Well use that to add some styling to our app by amending our application's template.

Open **app/views/layouts/application.html.erb** in your text editor.

Box 3.1. Why is that file named like that?

ERB stands for Embedded Ruby. You can take a normal HTML file – which would be called application.html, in this case – and if you rename it to application.html.erb, you can embed snippets of Ruby code into it. The code snippets get run when the page is loaded in the browser.

This comes in handy the more data you have. Say you have fifty books in your system. Instead of creating 50 html files, you can create one html.erb file and use Ruby to get the right data for each book, on the fly, when the browser asks for it. It's like using a template in Word, or any program: ERB saves time and duplication.

The **app/views/layouts/application.html.erb** file lays out the basic template for every page in our application. If you have seen any HTML before, most of

this page will look familiar. HTML, or HyperText Markup Language, is the standard markup language used to create Web pages. HTML is written in the form of HTML elements consisting of tags enclosed in angle brackets (like <html>). Read more in the Wikipedia article on HTML. There are lots of other online resources about HTML, but we can see the gist of how it works by looking at **application.html.erb**.

If you look at that file, then go to your browser and click on **View > Developer > View source** (in Chrome. It's a similar path for other browsers), you'll see that the code is practically the same. Both versions start with

```
<!DOCTYPE html>
<html>
<head>
  <title>BookOrganiser</title>
```

But see that **<%= stylesheet_link_tag %>** in the next line? That is a bit of Ruby code. We know it's Ruby because of the **<%=** at the beginning and the **%>** at the end. Anything between those two markers is Ruby, and will get evaluated. The **<%= stylesheet_link_tag %>** is the bit of Ruby code that will get evaluated. So the browser won't just show the phrase 'stylesheet_link_tag', or think that it's plain old HTML: some Ruby code will run instead.

This **stylesheet_link_tag** phrase is the name of a method – a little program that Rails understands. In this case, the code that runs makes sure that our Rails app references the right Cascading Stylesheets, or CSS. (The CSS makes the page look the way it does in the browser.) So when you View Source in your browser you'll see HTML code that isn't in the **application.html.erb** file. That code has been inserted dynamically.

Let's get on with enhancing how our app looks. Above the line

```
  <%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
```

add

```
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap-ther
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></script>
  <link href='http://fonts.googleapis.com/css?family=Oxygen:400,700,300' rel='stylesheet' type='t
```

Next, look for

```
  <%= yield %>
```

and replace it with

```
<div class="container">
  <%= yield %>
</div>
```

Box 3.2. Principle: Don't Repeat Yourself (DRY)

yield identifies a section where content from the view should be inserted. This means we don't have to put all the header code and the nav bar code on every page. The things that will appear on every page can be coded once, and we can just insert the unique code for the different pages. This means our code is DRY.

DRY is a principle of software development which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." By not writing the same information over and over again, our code is more maintainable, more extensible, and less buggy.

Lets also add a navigation bar to the layout. In the same file, under **<body>**, add

```
<nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
```

```
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-co
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">The Book Organiser app</a>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Now lets also change the styling of the Works table. Open **app/assets/stylesheets/applicatio** and at the bottom add

```
body { padding-top: 100px;   font-family: Oxygen, sans serif}
footer { margin-top: 100px; }
table, td, th { vertical-align: middle; border: none; }
th { border-bottom: 1px solid #DDD; }
```

Next, **open app/views/works/index.html.erb** and replace line 3:

```
<table>
```

with

```
<table class ='table'>
```

This adds a 'class' attribute called **table** to the HTML tag also called table. The **table** class is recognised by Twitter Bootstrap, and it makes the table take on Bootstrap's styling. It will give it nice padding and spacing, and some pleasing lines. (The people who wrote Bootstrap must have thought that giving the same name to the class as the tag was a good idea.)

Now make sure you saved your files and refresh the browser to see what has changed.

Let's add a bit of data. Click on **New Work** and use your shiny new website's form to add a new work to your system. Use Amazon to grab a cover image and a description. Type in the subject, but leave the cover field blank for now. Click on 'Create Work' when you're done.

Great! You've not only created a web application, but you've added some real data to it. Let's see what else we can make it do.

Chapter 4

Adding more features

4.1 Adding uploads

You can judge a book by it's cover – and we'll need to be able to manage the covers of our books. We're going to use a bundle of pre-written code called a "gem" to let us upload files to our Rails app.

Open the file called **Gemfile** in the project directory and at the bottom add

```
gem 'carrierwave'
gem 'isbn'
gem 'haml'
```

and save the file.

Box 4.1. Why is the carrierwave and the isbn code free?

Rails is an open source library of code, known as a "gem". There are many different gems, created by developers around the world, made freely available as part of the open source software movement. There's a website called rubygems.org which acts as a reference for all the gems published. You can usually see the gem's source code on code repository service github.com.

The Ruby community encourages its members to open-source and contribute to as much high-quality gems as they can, so that everyone can benefit from code that has had many different developers working on it.

We've included three gems here. The first is for uploading files. The second is an ISBN validations testing library, and the third is an alternative templating language to ERB. We'll look at them all in turn.

In the console run:

```
bundle
```

Now we can generate the code for handling uploads. In the terminal run:

```
rails generate uploader Cover
```

At this point you need to restart the Rails server process in the console. Hit CTRL-C in the terminal to quit the server. Once it has stopped, you can press the up arrow to get to the last command entered, then hit enter to start the server again. This is so the app can load the new code.

Open **app/models/work.rb** and under the line

```
class Work < ActiveRecord::Base
```

add

```
mount_uploader :cover, CoverUploader
```

Open **app/views/works/_form.html.erb** and change

```
<%= f.text_field :cover %>
```

to

```
<%= f.file_field :cover %>
```

Sometimes, you might get an TypeError: cant cast ActionDispatch::Http::UploadedFile to string.

If this happens, in **file app/views/works/_form.html.erb** change the line

```
<%= form_for(@work) do |f| %>
```

to

```
<%= form_for @work, :html => {:multipart => true} do |f| %>
```

In your browser, add a new work with a cover. When you upload a cover it doesnt look nice because it only shows a path to the file, so lets fix that.

Open **app/views/works/show.html.erb** and change

```
<%= @work.cover %>
```

to

```
<%= image_tag(@work.cover_url, :width => 300) if @work.cover.present? %>
```

Now refresh your browser to see what has changed.

4.2 Haml

We included the Haml gem earlier. Haml (HTML Abstraction Markup Language) is a layer on top of HTML that's designed to express the structure of documents in a non-repetitive, elegant, and easy way by using indentation rather than closing tags and allowing Ruby to be embedded with ease (see Rubygems for more and the Haml homepage). Haml is a good example of a time when a developer thought hmm, things could be easier, and wrote a library of code, then made it available to everyone else for free at which point Haml took on a life of its own, and is used widely. (The chap who wrote Haml was only about 20 at the time.)

Let's see how Haml helps. Go to http://htmltohaml.com and paste the whole contents of **app/views/works/_form.html.erb** in. The Haml translation will appear on the right. It's much shorter and more readable.

Copy the Haml version on the right, then go to **app/views/works/_form.html.erb** and paste it all in. Then rename the file to be **_form.html.haml**.

Restart the server by going to the console where the server is running and hitting CTRL-C, then up-arrow to get the last command, which was **rails server**, then hitting enter. You'll see that nothing has changed on the browser page, but our code is easier to read and more elegant.

A lot of programming is about making life easier.

Chapter 5

Making it useful

Click on the Show link from the works page for one of your books. We've got some data in but we haven't got it looking very compelling yet. Let's amend this show page so that it makes our data look good.

But where did this show page come from?

Box 5.1. Principle: Convention Over Configuration

Rails has opinions about the best way to do many things in a web application, and defaults to this set of conventions, rather than require that you specify every last thing. So when we used the Rails scaffold to create our works table, Rails also set up a bunch of files that most web developers would create, most of the time.

Go to the **app/views/works** folder. In there you'll see

- an Index ERB page
- a Show ERB page
- an Edit ERB page
- a New ERB page
- an ERB form.

Did you wonder why you were able to create a work without writing any code to define the form? Rails took an educated guess about what might be useful, and built it for you. Most of the time we like to see a summary listing of our data: that's taken care of by the Index page. We tend to want to create new data – that's the New page. When the data changes, that's the Edit page. If we just want to read the data, we go to the Show page. Rails has also set things up so we can delete data too, which is another common action – we'll see that in a bit.

We refer to these common actions as CRUD: create, read, update and delete.

Did you see that the 'form' file starts with an underscore? Files that start with an underscore are called 'partials'. They contain snippets of code that you can easily reference from other files. It's that DRY principle at work again – don't repeat yourself. The same form is used on the New page and the Edit page so we can reuse the code from one partial, to avoid having to have two versions of the same code to maintain.

Open **app/views/works/new.html.erb** to see what I mean. The file contains just three lines of code:

```
<h1>New work</h1>
<%= render 'form' %>
<%= link_to 'Back', works_path %>
```

Type **/works/new** into your browser address bar to see the page that this code produces.

Where did the navigation bar come from? It's not in our three lines of code. Well, do you remember looking at **application.html.erb**? The 'new' page uses the code produced by that template. Yep: the nav bar comes from the **application.html.erb** template. And where it said **yield** in that file, Rails is inserting the contents of the **new.html.erb** file. And even better – where it says **render** on line 2, here, Rails knows to use the code from the partial called 'form'. **render**, similar to **yield**, means 'go and get some code from another file and pop it in here'.

Re-using code like this keeps everything neat and tidy and less cluttered.

5.1 The show page

Let's look at the Show page. Click on **app/views/works/show.html.erb**. The first bit of Ruby that catches my eye is this line:

```
<%= @work.name %>
```

We know it's Ruby: we recognise the **<%= %>** opening and closing ERB tags. What's the **@work.name**, though?

Look in the browser to see if we can figure it out. Go to your /works index page and click on the Show link. At the top it says "Name:" and then the name of your work. So this embedded Ruby must be getting the name of the work, somehow.

Working backwards, we can probably figure out that putting **.name** after **@work** returns some data. In this case, **.name** is a method. It's one of the columns we set up in the database, and Rails knows that when we use one of the column names, we mean 'get the data from that so-called column'. So if you look a bit further down the page, and see **@work.description**, you can be pretty confident that Rails knows to get whatever's in the column called 'description'.

But for which work? There might be 50 in the database. So Rails uses the unique ID that's included in the URL to look up the right one. Look up in the browser URL address bar. It says something like **http://nitrousname.euw1-2.nitrousbox.com/works/1**. Focus on the /works/1 bit for the moment. We can use another Rails convention: when the URL says [table name]/[number], in this case works/1, Rails knows to look in the works table for the record with ID 1. If the URL said /works/231, Rails would look up the record in the works table with ID 231.

It's not the view that figures all this out, though. The view gets given the @work instance variable from somewhere else: the controller. Here's the code that the controller uses to get the data from the database:

```
@work = Work.find(params[:id])
```

This is a Ruby query. It says:

@work = create an instance variable called @work, and make it represent a particular works record. To find out which work record it is...

Work ...go to the works table...

.find ...and find...

(params[:id]) ...the work whose ID is the one just sent from the webpage.

So if the params are /works/314, then we know the value of **params[:id]** is 314.

We can see the contents of params if we look on the console. Open the console window that has the server running in it. Scroll to the bottom, and then in your browser click on 'Show' again. In the console it will say something like:

```
Started GET "/works/1" for 146.200.145.192 at 2015-01-17 18:51:23 +0000
Processing by WorksController#show as HTML
Parameters: {"id"=>"1"}
Work Load (0.3ms)  SELECT  "works".* FROM "works"  WHERE "works"."id" = $1 LIMIT 1  [["id", 1]]
Rendered works/show.html.erb within layouts/application (1.8ms)
Completed 200 OK in 53ms (Views: 51.1ms | ActiveRecord: 0.3ms)
```

This is all very useful information:

```
Started GET "/works/1" for 146.200.145.192 at 2015-01-17 18:51:23 +0000
```

Tells us that the browser sent a GET request for a work of ID 1.

```
Processing by WorksController#show as HTML
```

tells us that this request got routed to the correct controller.

```
Parameters: {"id"=>"1"}
```

tells us that the params passed along with this request are **{"id"=>"1"}**

```
Work Load (0.3ms)  SELECT  "works".* FROM "works"  WHERE "works"."id" = $1 LIMIT 1  [["id", 1]]
```

is the actual SQL query that runs to get the right data from the database.

```
Rendered works/show.html.erb within layouts/application (1.8ms)
```

tells us that after the request was completed, the controller routed us back to the show page.

```
Completed 200 OK in 53ms (Views: 51.1ms | ActiveRecord: 0.3ms)
```

tells us that all of this took less than half a second.

The params (which is short for 'parameters') are passed from the view to the controller as a "hash". Ruby hashes are one of the things that programmers love the most about the language because they're useful and flexible. They contain data in key-value pairs, in any order, like this:

```
person = {:name => "Emma", :age => "40", :hobby => "piano"}
```

You can find out much more about hashes on the internet. For now, you've seen that you can get the value of data in a hash if you say the name of the hash and then put the key of the data you want, in square brackets.

So in this example, doing **person[:name]** would return "Emma". **person[:hobby]** would return – yep, you guessed it – "piano". How would you get my age? **person[:age]**. (Let's look at Ruby arithmetic, later on, to knock some years off.)

In our real example, doing **params[:id]** would return: 1.

5.2 Creating an AI

Now we know how to get data from the database, and channel it through the right routes to get to the webpage view, let's try to make some of that data look pretty.

In **app/views/works/show.html.erb**, paste in this code, replacing everything that's in there already:

```
<p id="notice"><%= notice %></p>

<div class="row">
  <div class="col-sm-4">
    <%= image_tag(@work.cover_url, :width => 300) if @work.cover.present? %>
  </div>

  <div class="col-sm-8">
    <h1>
      <%= @work.name %>
    </h1>
      <strong>
      <%= @work.subject %>
      &bull;
    </strong>
    <%= simple_format @work.description %>
  </div>
</div>



<%= link_to 'Edit', edit_work_path(@work) %> |
<%= link_to 'Back', works_path %>
```

That new code uses **@work.cover_url**, **@work.name**, **@work.subject**, and **@work.description**. It's also arranged the page into two columns, using the Twitter Bootstrap styling we included. Bootstrap uses a grid layout of 12 columns, so if we set up one div of 4 columns and one of 8 columns, we know it'll fit nicely on the page.

The page is looking pretty nice so far. However, we've only got the work level data. We're really going to need to see the actual products' information if anyone's going to be able to use this to place orders. Let's add a products table.

5.3 Adding products to our works

Open a new console window and run the following:

```
rails generate scaffold Product isbn:string pub_date:date format:string price:decimal pages:int
```

then

```
rake db:migrate
```

Now we have to relate the products to their parent work, and vice versa. A work has many products, and a product belongs to a work. First we'll go to the work model and add in the association. Go to **app/models/work.rb**:

```
class Work < ActiveRecord::Base
  mount_uploader :cover, CoverUploader
end
```

Add the following in before the **end**.

```
has_many :products, :inverse_of => :work, :dependent => :destroy
```

And we add the reciprocal line in to the Product model:
Go to **app/models/product.rb**.

```
class Product < ActiveRecord::Base
end
```

Add the following in before the **end**.

```
belongs_to :work, :inverse_of => :products
```

Now let's add our new products listing to the navigation menu: find the following:

```
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
      </ul>
    </div>
```

and add in this line:

```
  <li><a href="/products">Products</a></li>
```

so it reads:

```
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
        <li><a href="/products">Products</a></li>
      </ul>
    </div>
```

Add **class = 'table'** to the products table. Go to **app/views/products/index.html.erb** and change line 3 to

```
<table class = 'table'>
```

Click New product. You can see the correct fields, but we can make it easier to fill them it accurately.

Replace

```
    <%= f.number_field :work_id %>
```

With

```
    <%= f.collection_select :work_id, Work.all, :id, :name %>
```

collection_select is a Rails helper method. This method takes four arguments. It's going to let us pass the value of work_id into the params when we hit 'save'. It'll get the ID from the dropdown list, which will be populated by Work.all (all the works). The next argument says that it's the id that'll be passed. And finally, the last argument tells Rails that it should display the name of the work to the user of the website.

Go to the browser and see that there's a drop down list now. Create a couple of products using the form.

You'll see that we can put any old thing into the ISBN field. Let's use the gem we added earlier to check whether the ISBN is properly formed.

In **app/views/products/edit.html.erb** add this code in just under the ISBN field div:

```
<% if ISBN.valid?(@product.isbn) %>
  <p class = 'bg-success'>Great -- the ISBN is valid!</p>
<% elsif @product.isbn.blank? %>
  <p class = 'bg-info'>Don't forget to add an ISBN!</p>
<% else %>
  <p class = 'bg-danger'> Oh no -- the ISBN is not valid!</p>
<% end %>
```

The **ISBN** method comes as part of the **isbn** gem that we included earlier. It's quite readable, which is one of the principles of Ruby. Method names should be easy to read and understand. You can practically say out loud in English what's going on:

"If the ISBN is valid – the product's ISBN – then say great, and put some green on the screen to signify that all's well. Otherwise, if the product's ISBN is blank, put up an information message to remind the user to add an ISBN. If the ISBN is neither valid nor missing, then it must be invalid, so report the problem to the user." In fact the Ruby is much easier to read than that sentence in English.

Now, let's make the products appear on the work show page. Go to **application/views/works/show.html.erb**. After the description field, paste this in:

```
<% @work.products.each do |product| %>
  <p>
    <%= product.format %>
    &bull;
    <%= product.isbn %>
    &bull;
    <%= product.pages %> pages
    &bull;
    č<%= product.price %>
    &bull;
    Published <%= product.pub_date.strftime('%d %B %y') %>
  </p>
<% end %>
```

That block of code is going to run for each of the child products of the parent work. We don't have to have a piece of code for each product – the code gets reused.

5.4 Other forms of the same data: APIs

Replace the index action in the works controller with this:

```
def index
  @works = Work.all
  respond_to do |format|
    format.html
    format.xml { render :xml => @works.to_xml }
    format.json { render :json => @works.to_json }
  end
end
```

Now go to the works listing page, and in the address bar of the browser, add **.xml** after the number. You'll see something like this:

```
<works type="array">
  <work>
    <id type="integer">1</id>
    <name>War and Peace</name>
    <description>
      War and Peace (Pre-reform Russian:    , Voyna i mir) is a novel by the Russian author Leo To
```

```
    </description>
    <cover>
      <url>/uploads/work/cover/1/9781909679436.jpg</url>
    </cover>
    <subject>Fiction</subject>
    <created-at type="dateTime">2015-01-17T14:14:45Z</created-at>
    <updated-at type="dateTime">2015-01-17T14:46:07Z</updated-at>
  </work>
</works>
```

Do the same but now appending **.json**. You'll see something like this:

```
[{"id":1,"name":"War and Peace","description":"War and Peace (Pre-reform Russian:   , Voyna i mi
```

This is a JSON API. If this was live, other websites would be able to ping it and use its data. APIs are hugely powerful because they release data in a lightweight format for apps to search, ingest and discover. Here's a good article on them, and another.

Go to **https://www.hurl.it/** and paste the full URL of your works.json page in, e.g.

```
http://supersonic-ghost-95-183846.euw1-2.nitrousbox.com/works.json
```

The page will show that your API has parsed successfully.

Chapter 6

Further Exercises

Here are some ideas for more ways you could improve your app. Can you:

- make a thumbnail image appear on the Works listing page, instead of the URL?
- change the links on the Works page so that they say “Al”, “Edit”, and “Remove”, instead of the boilerplate “Show”, “Edit” and “Destroy”?
- add a link on the works listing page to switch to the XML view? Here’s the Rails syntax for a link: **<%= link_to "XML", works_path(:format => :xml) %>**
- add a link to the work from the product page? Here’s the Rails syntax for the link: **<%= link_to "Work", work_path(@product.work) %>**
- Look at the Bootstrap documents. Are there any ways you could alter the styling of the HTML pages, using classes that Bootstrap responds to?
- create an Author model and associate it with the work?
- generate a JSON API response for a single work?

Chapter 7

Code, instructions and coach notes

This chapter contains only the instructions and the code that we'll be using on the course, and some brief coaching notes. The content is the same as in the previous chapters, summarised here for convenience during the course.

7.1 Chapter 1: Getting started

Box 7.1. Coach notes

Talk about the recent history of programming languages, and how nowadays there are tools to get started in programming without too much fuss.

7.1.1 Set up a Nitrous.io account

- Go to https://www.nitrous.io/
- Enter a username.
- Now enter your email address.

- Enter a password.
- Hit "Sign up for free". If any of your details aren't up to scratch, have another go.
- You'll have to log in to your email to confirm you want to sign up. Do that and click on the link to confirm you want an account.
- Then you can sign in. Enter your email or username, and password.

7.1.2 Create a development box

- Click on the big green button that says **Open dashboard**.
- You'll see a panel that says **Create your first box**. Ruby on Rails will be highlighted. Leave everything as it is, and click the orange **Create Box** button.
- On the next screen, wait until the line at the bottom goes green and it says [your box name] is running. Click **Next**. On the next page, scroll down and click **Okay, Take me to my box**.

7.1.3 Set up PostgreSQL

```
cd workspace
```

```
parts install postgresql
```

```
parts start postgresql
```

7.1.4 Create our app

Box 7.2. Coach notes

Talk about what a Rails app is: a way to produce HTML files on the fly.

```
rails new book_organiser -d postgresql
```

```
cd book_organiser
```

7.1.5 Link the app to the database

Replace the file at **workspace/book_organiser/config/database.yml** with:

```
development:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_dev
  pool: 5
  host: localhost
  username: action
  password:

test:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_test
  pool: 5
  host: localhost
  username: action
  password:

production:
  adapter: postgresql
  encoding: unicode
  database: book_organiser_prod
  pool: 5
  host: localhost
  username: action
  password:
```

```
rake db:create:all db:migrate
```

7.1.6 Look at our new app running in the browser

```
  rails server
```

7.2 Chapter 2: Structuring our app

Box 7.3. Coach notes

Talk about MVC and object oriented programming.

```
cd workspace/book_organiser
```

Box 7.4. Coach notes

Talk about Rails scaffolding, and data types.

```
rails generate scaffold work name:string description:text cover:string subject:string
```

```
rake db:migrate
```

Box 7.5. Coach notes

Talk some more about MVC, and particularly routing.

Open the file **app/config/routes.rb** in Nitrous. Add the following in, on the second line of the file:

```
root 'works#index'
```

7.3 Chapter 3: Making it look good

Box 7.6. Coach notes

Talk about ERB.

7.3.1 Enhance how our app looks with Bootstrap

Let's get on with enhancing how our app looks. Above the line

```
<%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
```

add

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap-ther
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></script>
<link href='http://fonts.googleapis.com/css?family=Oxygen:400,700,300' rel='stylesheet' type='t
```

7.3.2 Keep things DRY

Box 7.7. Coach notes

Talk about the DRY principle.

Next, look for

```
  <%= yield %>
```

and replace it with

```
<div class="container">
  <%= yield %>
</div>
```

Lets also add a navigation bar to the layout. In the same file, under **<body>**, add

```
<nav class="navbar navbar-default navbar-fixed-top navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-co
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">The Book Organiser app</a>
    </div>
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Now lets also change the styling of the Works table. Open **app/assets/stylesheets/applicatio** and at the bottom add

```
body { padding-top: 100px;   font-family: Oxygen, sans serif}
footer { margin-top: 100px; }
table, td, th { vertical-align: middle; border: none; }
th { border-bottom: 1px solid #DDD; }
```

Next, **open app/views/works/index.html.erb** and replace line 3:

```
<table>
```

with

```
<table class ='table'>
```

Add a work

Let's add a bit of data. Click on **New Work** and use your shiny new website's form to add a new work to your system. Use Amazon to grab a cover image and a description. Type in the subject, but leave the cover field blank for now. Click on 'Create Work' when you're done.

7.4 Chapter 4: Adding more features

7.4.1 Adding gems

Box 7.8. Coach notes

Talk about the open source movement and gems.

Open the file called **Gemfile** in the project directory and at the bottom add

```
  gem 'carrierwave'
  gem 'isbn'
  gem 'haml'
```

and save the file.

In the console run:

```
bundle
```

7.4.2 Adding uploads

Now we can generate the code for handling uploads. In the terminal run:

```
rails generate uploader Cover
```

At this point you need to restart the Rails server process in the console. Hit CTRL-C in the terminal to quit the server. Once it has stopped, you can press the up arrow to get to the last command entered, then hit enter to start the server again. This is so the app can load the new code.

Open **app/models/work.rb** and under the line

```
class Work < ActiveRecord::Base
```

add

```
mount_uploader :cover, CoverUploader
```

Open **app/views/works/_form.html.erb** and change

```
<%= f.text_field :cover %>
```

to

```
<%= f.file_field :cover %>
```

In your browser, add a new work with a cover. When you upload a cover it doesnt look nice because it only shows a path to the file, so lets fix that.

Open **app/views/works/show.html.erb** and change

```
<%= @work.cover %>
```

to

```
<%= image_tag(@work.cover_url, :width => 300) if @work.cover.present? %>
```

Now refresh your browser to see what has changed.

7.4.3 Improving the code

Box 7.9. Coach notes

Talk about Haml.

Go to http://htmltohaml.com and paste the whole contents of **app/views/works/_form.html.erb** in.

Copy the Haml version on the right, then go to **app/views/works/_form.html.erb** and paste it all in. Then rename the file to be **_form.html.haml**.

Restart the server by going to the console where the server is running and hitting CTRL-C, then up-arrow to get the last command, which was **rails server**, then hitting enter. You'll see that nothing has changed on the browser page, but our code is easier to read and more elegant.

7.5 Chapter 5: Making it useful

7.5.1 Convention over configuration

Box 7.10. Coach notes

Talk about convention over configuration.

In **app/views/works/show.html.erb**, paste in this code, replacing everything that's in there already:

```
<p id="notice"><%= notice %></p>

<div class="row">
  <div class="col-sm-4">
    <%= image_tag(@work.cover_url, :width => 300) if @work.cover.present? %>
  </div>

  <div class="col-sm-8">
    <h1>
      <%= @work.name %>
    </h1>
      <strong>
      <%= @work.subject %>
      &bull;
    </strong>
    <%= simple_format @work.description %>
  </div>
</div>
<%= link_to 'Edit', edit_work_path(@work) %> |
<%= link_to 'Back', works_path %>
```

7.5.2 Queries

Box 7.11. Coach notes

Talk about how methods and ActiveRecord queries work

7.5.3 Associations

Let's add in products. Open a new console window and run the following:

```
rails generate scaffold Product isbn:string pub_date:date format:string price:decimal pages:int
```

then

```
rake db:migrate
```

Box 7.12. Coach notes

Talk about associations.

Now we have to relate the products to their parent work, and vice versa. A work has many products, and a product belongs to a work. First we'll go to the work model and add in the association. Go to **app/models/work.rb**:

```
class Work < ActiveRecord::Base
  mount_uploader :cover, CoverUploader
end
```

Add the following in before the **end**.

```
has_many :products, :inverse_of => :work, :dependent => :destroy
```

And we add the reciprocal line in to the Product model:
Go to **app/models/product.rb**.

```
class Product < ActiveRecord::Base
end
```

Add the following in before the **end**.

```
  belongs_to :work, :inverse_of => :products
```

Now let's add our new products listing to the navigation menu: find the following:

```
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
      </ul>
    </div>
```

and add in this line:

```
  <li><a href="/products">Products</a></li>
```

so it reads:

```
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li><a href="/works">Works</a></li>
        <li><a href="/products">Products</a></li>
      </ul>
    </div>
```

Add **class = 'table'** to the products table. Go to **app/views/products/index.html.erb** and change line 3 to

```
<table class = 'table'>
```

Click New product. You can see the correct fields, but we can make it easier to fill them it accurately.

7.5.4 Using form helpers

Box 7.13. Coach notes

Talk about form helpers and the other sorts of Rails methods that save time.

Replace

```
<%= f.number_field :work_id %>
```

With

```
<%= f.collection_select :work_id, Work.all, :id, :name %>
```

Go to the browser and see that there's a drop down list now. Create a couple of products using the form.

7.5.5 Conditional statements

Box 7.14. Coach notes

Talk about using the ISBN gem, conditional statements, the notion of validations and how Ruby is human-readable.

In **app/views/products/edit.html.erb** add this code in just under the ISBN field div:

```
<% if ISBN.valid?(@product.isbn) %>
  <p class = 'bg-success'>Great -- the ISBN is valid!</p>
<% elsif @product.isbn.blank? %>
  <p class = 'bg-info'>Don't forget to add an ISBN!</p>
<% else %>
  <p class = 'bg-danger'> Oh no -- the ISBN is not valid!</p>
<% end %>
```

7.5.6 Code blocks

Box 7.15. Coach notes

Talk about blocks.

Now, let's make the products appear on the work show page. Go to **app/views/works/show.html**
After the description field, paste this in:

```
  <% @work.products.each do |product| %>
    <p>
      <%= product.format %>
      &bull;
      <%= product.isbn %>
      &bull;
      <%= product.pages %> pages
      &bull;
      č<%= product.price %>
      &bull;
      Published <%= product.pub_date.strftime('%d %B %y') %>
    </p>
  <% end %>
```

7.5.7 APIs

Box 7.16. Coach notes

> Talk about APIs.

Replace the index action in the works controller with this:

```
def index
  @works = Work.all
  respond_to do |format|
    format.html
    format.xml { render :xml => @works.to_xml }
    format.json { render :json => @works.to_json }
  end
end
```