Sinatra Cookbook

Recipes for the Ruby framework

Tim Millwood

Contents

Thanks

All the Kickstarter backers for Sinatra Cookbook: Josh Kaufman, Heroku, Paul Thomas, Darren Jones, Peter MS, Andrew Broman, clive mansfield, Kim Pedersen, Stephan Kämper, Tom O'Connor, James Gregory, Simon Starr, Zach Feldman, Tim Hilliard, Robert Ristroph, Victor Kane, Chris Pelsor, Daniel Nordh, Scott Windsor, Konstantin Haase, Christian, Rasmus Bang Grouleff, Gabriele Lana, Marcus Schappi, Scott MacFarlane, Alex Koloskov, Abraão Coelho, Sean Dunn, Ludovic Kuty, jerefrer, Ivan Johns, Charlie Gaines, Miguel Romero, Charlie Ahn, Peter Hellberg, Michael Allan, Katarzyna Jarmokowicz, Nick Charlton, moshe weitzman, Florian, Alexey Smirnov, Timothy Barnes, Andrew Selder, Avdi Grimm, Marc Stein, Eva Barabas, Joshua Antonishen, Brian Broom, Pedro Cambra, Takayuki Matsubara, Byron Norris, Paul Morganthall, Peter Cooper, robin paul, Coby Randquist, Michael Cook, Steve P. Sharpe, Justin Coy Martin, Lars Klaeboe, Tim Duckett, Rik Schneider, Ilya Vorontsov, Andrew Nesbitt, rajan venkataguru, Alex Brem, Jakub Suchy, Kerry Buckley, Pedro Arnal Puente, Eduardo Turiño, Garrow, Timo Borreck, Harriet Holman, Sean O'Toole, Steve Lloyd, Cary Gordon, Christian Aust, Jolyon Russ, Uur Özylmazel, George Hazlewood, Michael Booth, iain2k, grantmichaels, Guillaume Troppée, Miguel Veríssimo, Paul Trunz, Tom Bamford, Shaban Karumba, Wouter Willaert, Dave Porter, Paul Klonowski, Guillaume Laville, Jack Hutton, Nick McFarlane, Brian Martin, Cheng Zhang, Joel Hughes Peter Jahn, Dany Chavez, Wali, David Morris, raymond wang, Stu Robson, Luke Burford, David Hernández Ruiz, Grigory Petrov, Mike, Fuad Saud, Anne Cahalan, Federico Builes, blackwinter, Bryce Lambert, Peter Rootham-Smith, Andy Croll, Matthew Lucasiewicz, Aaron Moodie, Seiichi Yonezawa, Bèr Kessels, Bala Nair, Alpha Chen, Jason Sallis, Aurelien Navarre, Nathan Young-

man, James Pearson, Mike Vormwald, Markus Heurung, Raj Sahae, Anton maminov, H Asari, Alan Burgoyne, Leopoldo, Olivier Vigneresse, Heather James, Thomas Blakey, Damien McKenna, Mike Breen, Paulo Fierro, Brian Moelk, Paul Visscher, Nate Swart, Björn Skarner Bengtsson, Tom Lane, Robert Greenshields, Alvar 'Allu22' Hansen, Craig Lockwood, Gary Aston, Carl Furrow, Pierre-Adrien Buisson, Matias Keveri, Vadim Golub, Otto Colomina Pardo, richardj, manoj reddy, Alexander Gräfe, John Kelly Ferguson, Sam Mason, Richard Woeber, AG, Tieg Zaharia, Saul Maddox, Temba Mazingi, John Bintz, Airat Shigapov, Chris Charlton, Emily Reese, Emile Silvis

Preface

The aim of Sinatra Cookbook is to give context to code. When learning a new programming language or framework, its hard to make sense of without understanding how it relates to a real world use case. This books 12 chapters will stand as a guide through 12 different types of application and how they were built using Sinatra and other third-party gems.

0.1 About the author

Tim Millwood is a web developer based in Abergavenny, Wales, UK. By day working for enterprise Drupal company, Acquia (http://acquia.com), helping to launch some of the worlds biggest Drupal sites. By night hes behind Millwood Online (http://millwoodonline.co.uk), working on various Ruby and PHP based development projects.

0.2 Why Ruby?

When starting to write this book, one question that came up was why Ruby? Why not use Java, Node.js or Go? Although other languages may offer better performance or other features, Ruby is one of the best languages for developer experience. It is easy to learn, easy to use and fun to write.

0.3 Why Sinatra?

Ruby shot to fame with the Ruby on Rails framework written by David Heinemeier Hansson for Basecamp. This sparked a number of different Ruby based frameworks including Sinatra, which was released in 2007. Sinatra, like Ruby on Rails, is based on the Rack web server interface and focuses on being able to quickly create web-applications with minimal effort. Sinatra is small and flexible, this makes it easy to work with and performs fast. Many argue just use Rails when building a web application on Ruby, but Sinatra can offer great performance and in many situations, faster development times.

0.4 Getting started

To follow the tutorials in this book you will need to make sure you have Ruby (http://ruby-lang.org) and RubyGems (http://rubygems.org) installed. If youre running on an up to date Mac you will find these are already installed. Even so, on all operating systems RVM (http://rvm.io/) is the recommended way to install Ruby, allowing easy maintenance of single or multiple Ruby versions. You might also like to setup an account on Heroku and install the Heroku Toolbelt (https://toolbelt.heroku.com/). Many chapters will discuss deployment to Herokus hosting environment, and all example code will come Heroku ready.

0.5 Hello World!

Before we start looking at the promised real world applications heres a quick introduction to Sinatra using an obligatory Hello World! application. Sinatra can take two styles, Classic or Modular. The Classic style allows for quick and easy development of a single Sinatra application, and has a number of features and configuration enabled by default. This is the style we will be using for the Hello World! application. Modular style applications are written by subclassing Sinatra::Base and allow for multiple Sinatra applications to be combined and run as a single Ruby process. Modular style also have some settings disabled

by default. All 12 of the applications in this book will follow the Modular format because its the most scalable approach.

For the **`Hello World!`** application, after creating a folder to hold it, create a file **`app.rb`**. Within this add the following code:

```
require 'sinatra'

get '/' do
  'Hello World!'
end
```

Firstly Sinatra is required for the application. Then based on the HTTP methods we specify a get block, this is paired with / to denote the root route. Within the block Hello World! is added as the text to be returned when this route is requested by the browser.

Before we run this application, installation of the Sinatra gem is needed. This can either be done by running the command gem install sinatra or by setting up a Gemfile. A Gemfile will allow us to easily manage all gem dependencies for the application using bundler. Run the command gem install bundler, then create a file called Gemfile containing the following:

```
source 'https://rubygems.org'
ruby "2.1.0"

gem 'sinatra'
```

This specifies the source for the gems as rubygems.org, then the ruby version as 2.1.0. RVM uses the Gemfile to select the version of Ruby, this is also used by Heroku when deploying an application. Below this all the gems are listed, in this case the one gem required is Sinatra. The command bundle install can now be run to make sure Sinatra is installed, this will also create a Gemfile.lock file to store the gem versions. Deploying the application along with the Gemfile.lock will make sure that the gem versions are the same across all environment.

Running the command ruby app.rb will start the application, which will be accessible via http://localhost:4567/

Sinatra is based on Rack and therefore can be run using rackup via a config.ru file. Create this file containing the following code:

```
require './app'
run Sinatra::Application
```

This first requires our Sinatra application, then runs it. To execute config.ru, run the command rackup, the site can then be accessed at http://localhost:9292/.

The following commands can be executed to commit the applications files to a local git repository and deploy to Heroku.

```
git init
git add .
git commit -m "Initial commit"
heroku create
git push heroku master
```

Once complete, the command **`heroku open`** will open the application in your browser. The application will be re-deployed with each commit.

Chapter 1

Static app

Sinatra has native support for many different template languages, this makes it great for making static sites. In this chapter were going to create a site that had a single route, which will load a markdown file where the url parameter matches the filename. If a markdown file doesnt exist for the given parameter a custom 404 page will be shown. The main layout will use ERB as the template language, then markdown for each page. The site will be for a local gym and will feature an index page, about page and contact page.

1.1 Gemfile

To start, a Gemfile is needed, and populated with the gems needed for the application.

```
source 'https://rubygems.org'
ruby "2.1.0"

gem 'sinatra'
gem 'rdiscount'

group :test do
  gem 'minitest'
  gem 'rack-test', :require => 'rack/test'
  gem 'rake'
end
```

We are using rubgems.org as the source for the gems in the application along with Ruby version 2.1.0. For the main application well make use of sinatra, and rdiscount for parsing the markdown files. For testing the application well need a few gems, a group block is added to the Gemfile so bundler can exclude the test gems on environments that dont need then, such as in production. Minitest is the test suite well be using for this application, coupled with rack-test to allow for the testing of Sinatra, which is based on the rack webserver interface. Finally rake is used to run the tests.

Now we have the Gemfile you can run the command bundle install to make sure these are all installed. This will also create the file Gemfile.lock to store the gem versions being used.

1.2 Spec

Well be using minitest spec to test the application. Create a folder spec and in here a file spec_helper.rb containing the following code:

```
ENV['RACK_ENV'] = 'test'

require 'minitest/autorun'
require 'rack/test'
require_relative '../app'

include Rack::Test::Methods

def app
  StaticApp
end
```

This helper can be added to each spec to pull in the requirements. First the RACK_ENV environment variable to test so that Sinatra and the application knows were working the test version of the application. We then require minitest/autorun, the easy and explicit way to run all your tests, and rack/test. Using require_relative, the main Sinatra application is pulled in (even though its not yet written). The `Rack::Test::Methods` module is included to add a number of helper methods to allow the testing of rack applications. The final

Sinatra application will be contained in the StaticApp class, therefore we define this as the application.

```
require_relative 'spec_helper'

describe 'Static App' do

  it 'should have index page' do
    get '/'
    last_response.must_be :ok?
    last_response.body.must_include "Clive Mansfield's Gym"
  end

  it 'should have about page' do
    get '/about'
    last_response.must_be :ok?
    last_response.body.must_include "About us"
  end

  it 'should have contact page' do
    get '/contact'
    last_response.must_be :ok?
    last_response.body.must_include "Contact us"
  end

  it 'should display pretty 404 errors' do
    get '/404'
    last_response.status.must_equal 404
    last_response.body.must_include "Page not found"
  end

end
```

The above code is the to be added to a file app_spec.rb within the spec directory. In the case of this application we only have one spec describing the whole application, but as you will see later is the book its often good practice to breakdown the application into multiple specs.

We first start by requiring the spec_helper, then opening a describe block to describe Static App, within this we have a number of tests or behaviours. The first three of these test the planned pages exist, the index page, the about page and the contact page. To check they exist it looks for an ok response and look to see the planned title is included in the page. The final behaviour checks a 404 error is returned if a route (or markdown file) doesnt exist in the application.

To do this is checks the status is 404 and the page includes the text page not found, because we will add a custom 404 page with this text.

Before these test can be run a rakefile needs to be added, this goes into the applications root directory.

```
require 'rake/testtask'
Rake::TestTask.new do |t|
  t.pattern = 'spec/*_spec.rb'
end
```

Firstly rake/testtask is required, then a new instance of Rake::TestTask is instantiated, within this a pattern for the spec files is set. Now we have a rakefile the tests can be run with the command rake test. At this point you will see 4 runs and 4 errors. Each error will be NameError: uninitialized constant StaticApp. This is because at this point we have not even written the class StaticApp. So, lets create app.rb.

1.3 App

```
require 'sinatra/base'
require 'rdiscount'

class StaticApp < Sinatra::Base
  get '/:file?' do
    cache_control :public, :must_revalidate, :max_age => 60
    begin
      file = params[:file] || 'index'
      markdown file.to_sym, :layout_engine => :erb
    rescue Errno::ENOENT
      not_found
    end
  end

  not_found do
    markdown :not_found, :layout_engine => :erb
  end
end
```

This application is using Sinatras Modular style so we first require sinatra/base, then rdiscount, which well be using for markdown processing. We

create the class StaticApp as a subclass of Sinatra::Base and within this, two blocks. The first using the get method with the route /:file?. :file is a parameter from the URL that we will use to call a markdown file, the question mark makes this parameter optional. First, using the cache_control helper, were going to set caching headers. In this case the max-age is set to 60 seconds, allowing the browser or external service to cache the page for 60 seconds. We use exception handling for the error Errno::ENOENT and call not_found to return a 404 error. The file variable is set with the file parameter from the URL or index if no parameter exists. Markdown is loaded with that file variable, which weve converted to a symbol by using the to_sym method. The layout engine for this is set to erb affording us a little more control over the output. Unlike markdown, erb allows the use of Ruby within the view, for which we could pass variables or generate a dynamic output.

The second block within the StaticApp class is not_found, this allows us to return a view for 404 errors. Again we use markdown as a the main template engine and erb as a layout engine.

For the markdown and erb files we create a views directory. The first file in this directory is layout.erb.

```
<!DOCTYPE html>
<!--[if IE 9]><html class="lt-ie10" lang="en" > <![endif]-->
<html class="no-js" lang="en" >
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Clive's Gym</title>
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/foundation.css">
    <script src="js/vendor/custom.modernizr.js"></script>
    <style>
      img{ float: right; }
    </style>
  </head>
  <body>
    <nav class="top-bar" data-topbar>
      <ul class="title-area">
        <li class="name">
          <h1><a href="/">Clive's Gym</a></h1>
        </li>
        <li class="toggle-topbar menu-icon"><a href="#"><span></span></a></li>
      </ul>
```

```
      <section class="top-bar-section">
        <!-- Right Nav Section -->
        <ul class="right">
          <li><a href="/">Home</a></li>
          <li><a href="/about">About</a></li>
          <li><a href="/contact">Contact</a></li>
        </ul>
      </section>
    </nav>
    <div class="row">
      <div class="large-12 columns">
        <%= yield %>
      </div>
    </div>

    <script src="js/vendor/jquery.js"></script>
    <script src="js/foundation.min.js"></script>
    <script src="js/foundation/foundation.topbar.js"></script>
    <script>
      $(document).foundation();
    </script>
  </body>
</html>
```

Youll see from this erb file that its just standard HTML markup, in this case were using the foundation framework (http://foundation.zurb.com/). This file features all of the static items that dont change from page to page, such as the **<head>**, any javascript and css, and the navigation. One notable item is <%= yield %> this is where the markdown files will be rendered.

Our first markdown file, again in the views directory, is the default index.markdown

```
#Clive Mansfield's Gym
![bikes](img/bikes.jpg)
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut quis massa non elit viverra adipisc
```

Weve added a **<h1>** title by the hash symbol, an image with the alt text in square brackets and file path in curved brackets, then body text below which will be formatted with **<p>** tags. When the root path or /index is requested this markdown file will be rendered along within the yield tag of the layout.erb.

Three more markdown files are needed to complete the application and pass all tests, about.markdown, contact.markdown and not_found.markdown.

A public folder will need to be created for all static assets, such as the css and javascript for the Foundation framework, and the images added in the markdown files. When a URL is requested Sinatra will first check the public folder before looking for a route within the application. For example in layout.erb were using to load the CSS for the foundation framework. The file will need to be location at public/css/foundation.css within the application directory.

Before the application can be run we need a config.ru file.

```
require './app.rb'
run StaticApp
```

This rackup file is used by Rack to start the application. Here we simply require the Sinatra application, then run the StaticApp class. We can make the application Heroku ready by adding a procfile containing web: bundle exec rackup -p $PORT then use foreman to start the application or simply just run the command bundle exec rackup.

Running rack test should now show all tests passing, and all three markdown files will be accessible at their relevant urls.

1.4 Summary

Within this chapter we have looked at a number of elements within Sinatra. Weve created a get request returning multiple pages based on a URL parameter. This parameter loads a markdown file from the views directory, which is returned as part of a layout. We also used the not_found method to return a markdown file upon 404 error. A cache control header is set to allow for client side caching. All static assets are stored in the public directory and served from there by Sinatra. All expected URL paths are fully tested by Minitest, therefore checking the markdown files are loaded based on the URL parameter.

Chapter 2

API

The stripped down basic nature of Sinatra makes it a good choice for building APIs. Some of the worlds biggest companies use it just for this. This chapter will use Sinatra to build an API that would be ideal as the back-end of a native mobile application. HTTP authentication will be used to authenticate users against a SQLite database, which we will also use to store notes. Each note will have a title, body, created date and updated date, then be tied to the user who created it.

2.1 Tests

In the last chapter we started with tests, the tests for the API will follow the same pattern, although this time we need to test the HTTP authentication and test the full CRUD (create, read, update, destroy) process of the data. The only change to spec_helper.rb is the addition of include APIAPP, so that we can make use of the Datamapper modules contained in the app, and changing the app definition to App, as this will be the class name of the Sinatra app. Well have two spec files, one for actions carried out by an anonymous user and one for actions carried out by an authorized user, anonymous_spec.rb includes:

```
require_relative 'spec_helper'

describe 'API App' do
```

```
  it 'should return ok for root' do
    get '/'
    last_response.must_be :ok?
    last_response.body.must_include '{"status":"ok"}'
  end

  it 'should allow users to register' do
    post '/register', {:username => "andrewbroman", :password => "S1n4tra"}
    last_response.must_be :ok?
    last_response.body.must_include 'andrewbroman'
    User.first(:username => 'andrewbroman').destroy
  end
end
```

Using the contents of the last_response object, we first test the root route, checking it returns ok with the json response {status:ok}. Finally we check anonymous users can register. This is done as a post request, passing the username and password data. This should return an ok response, with ok and andrewbroman (the test username) in the json response. The user is then deleted by querying the User model for the user, based on username, then calling the destroy method. This is so we will be using the same details in further tests.

The **authorized_spec.rb** file has all the tests for authorized user actions:

```
require_relative 'spec_helper'

describe 'API App' do
  before do
    post '/register', {:username => "andrewbroman", :password => "S1n4tra"}
  end

  after do
    Note.all.destroy
    User.all.destroy
  end

  def create_note(title = "Sinatra", body = "Yay!")
    post '/notes', {:title => title, :body => body}
    last_response.must_be :ok?
    last_response.body.must_include %{"title":"#{title}"}
    last_response.body.must_include %{"body":"#{body}"}
  end

  def authorize_user
```

```
    authorize 'andrewbroman', 'S1n4tra'
  end

  def user
    User.first(:username => 'andrewbroman')
  end

  it 'should allow users to post notes' do
    authorize_user
    create_note("Ruby", "What awesomness!")
    create_note("Sinatra", "More awesomness!")
    user.notes.count.must_equal 2
  end

  it 'should return all notes' do
    authorize_user
    create_note("Ruby", "What awesomness!")
    create_note("Sinatra", "More awesomness!")
    get '/notes'
    last_response.must_be :ok?
    last_response.body.must_include '"title":"Ruby","body":"What awesomness!"'
    last_response.body.must_include '"title":"Sinatra","body":"More awesomness!"'
  end

  it 'should return a single note' do
    authorize_user
    create_note("Ruby", "What awesomness!")
    id = user.notes.last.id
    get "/notes/#{id}"
    last_response.must_be :ok?
    last_response.body.must_include '"title":"Ruby","body":"What awesomness!"'
  end

  it 'should be able to edit a note' do
    authorize_user
    create_note("Ruby", "What awesomness!")
    id = user.notes.last.id
    put "/notes/#{id}", {:body => 'Edited note'}
    last_response.must_be :ok?
    last_response.body.must_include '"body":"Edited note"'
  end

  it 'should be able to delete a note' do
    authorize_user
    create_note("Ruby", "What awesomness!")
    id = user.notes.last.id
    delete "/notes/#{id}"
    last_response.must_be :ok?
    last_response.body.must_include 'deleted'
  end
end
```

Before any of the tests there are a number of helper methods. Firstly, a before action to register a test user, this will run before all tests to make sure we have a user to authenticate. We then have an after action to remove all users and all notes after each tests, so the next test can start with a clean slate. We then have a create_note method to send a post request and generate a note, this is checked to make sure it returns ok and the title and body entered, it will help save efforts each time a note needs to be created as part of a test. The authorize_user method wraps the rack test authorize method to login the test user via http authentication. Finally, we have the user method, which fetches and returns our test user object via datamapper.

There are then five tests, which relate to the five routes we will have for authenticated users in the application. The first of which checks a note can be created by calling the create_note method twice, then checking there are two notes in the database for the test user by using the user model with the notes method to follow the one to many relationship users and notes have. The next test is to make sure all notes are returned, first by creating two notes, then calling the get method for the route /notes. Two checks are done to make sure that both notes created are returned as part of the json string via last_response.body.

To make sure that a single note can be returned we first create a note then fetch the id of the note last created. This id is passed via a get method to the /notes/:id route. A similar check is done that the expected json string is returned. Editing notes follows the same pattern, create a note and get its id, however we now call the put method with the id and the parameters to edit. A check is done that the edited text is returned, rather than the original note text. Finally well delete a note, again creating a note and getting its id. Then passing it to the delete method and checking deleted is returned as part of the json string.

2.2 App

There are a number of new gems needed for the API application. Create a Gemfile with the following gems.

```
source "https://rubygems.org"
ruby "2.1.0"

gem "sinatra"
gem "dm-sqlite-adapter"
gem "data_mapper"
gem "json"

group :test do
  gem 'minitest'
  gem 'rack-test', :require => 'rack/test'
  gem 'rake'
end
```

Youll see weve added in datamapper and its sqlite adapter. If you plan to deploy to Heroku youll need to use the gem “dm-postgres-adapter” instead of “dm-sqlite-adapter” for datamappers postgresql adapter. We are also making use of the json gem to encode all output in json.

Now on to the application, app.rb, where we first require the gems and create the APIAPP module.

```
require 'sinatra/base'
require 'data_mapper'
require 'digest/md5'
require 'json'

module APIAPP
  DataMapper.setup(:default, "sqlite:///#{File.expand_path(File.dirname(__FILE__))}/apiapp.db")
  class User
    include DataMapper::Resource

    property :id,          Serial
    property :username,    String, :required => true, :unique_index => true
    property :password,    String, :required => true

    has n, :notes
  end

  class Note
    include DataMapper::Resource

    property :id,         Serial
    property :title,      String, :required => true
    property :body,       Text, :required => true
    property :created_at, DateTime
    property :updated_at, DateTime
```

```
  belongs_to :user
end
DataMapper.finalize
DataMapper.auto_upgrade!

class App < Sinatra::Base
  helpers do
    def protected!
      return if authorized?
      headers['WWW-Authenticate'] = 'Basic realm="Restricted Area"'
      halt 401, "Not authorized\n"
    end

    def authorized?
      @auth ||=  Rack::Auth::Basic::Request.new(request.env)
      if @auth.provided? and @auth.basic? and @auth.credentials
        @user = User.first(:username => @auth.credentials[0])
        @user && @user.password == Digest::MD5.hexdigest(@auth.credentials[1])
      end
    end
  end

  get '/' do
    content_type :json
    {:status => "ok"}.to_json
  end

  post '/register' do
    content_type :json
    begin
      user = User.create(:username => params[:username], :password => Digest::MD5.hexdigest(p
      {:status => "ok", :data => user.username}.to_json
    rescue Exception => e
      {:status => "error", :data => e}.to_json
    end
  end

  post '/notes' do
    protected!
    content_type :json
    begin
      notes = @user.notes.create(params)
      {:status => "ok", :data => notes}.to_json
    rescue Exception => e
      {:status => "error", :data => e}.to_json
    end
  end

  get '/notes' do
    protected!
    content_type :json
```

```
      notes = @user.notes.all
      if !notes.empty?
        {:status => "ok", :data => notes}.to_json
      else
        {:status => "error", :data => "No notes"}.to_json
      end
    end

    get '/notes/:id' do
      protected!
      content_type :json
      if note = @user.notes.first(:id => params[:id])
        {:status => "ok", :data => note}.to_json
      else
        {:status => "error", :data => "No note"}.to_json
      end
    end

    put '/notes/:id' do
      protected!
      content_type :json
      begin
        note = @user.notes.first(:id => params[:id])
        note.update(:title => params[:title]) if params[:title]
        note.update(:body => params[:body]) if params[:body]
        note.save
        {:status => "ok", :data => note}.to_json
      rescue Exception => e
        {:status => "error", :data => e}.to_json
      end
    end

    delete '/notes/:id' do
      protected!
      content_type :json
      begin
        @user.notes.first(:id => params[:id]).destroy
        {:status => "ok", :data => "deleted"}.to_json
      rescue Exception => e
        {:status => "error", :data => e}.to_json
      end
    end
  end
end
```

Within the APIAPP module we setup datamapper with the path the sqlite database file. The file will not exist initially, but will get created by datamapper when the application is first run. When using postgresql this sqlite file path can be replaced with the postgresql url.

We can then create our two models, User and Note. A user can have many notes, but a note can only have one user. Therefore we need to setup a one to many relationship. We do this by adding has n, :notes to user and belongs_to :user to note. After the models we finalize and auto upgrade datamapper, this ensures that the database is created, and any changes are reflected on an existing database.

Once datamapper is setup the app class is created for Sinatra. Rather than set the HTTP authentication up for each route were going to add some helper functions to aid with it. The first function, protected!, checks to see the user is authenticated via the second function, authorized?. If the user is not authenticated a 401 error is sent. The authorized? function sets up a new Rack::Auth::Basic::Request instance before querying the User model for the username given. Finally TRUE is returned if the user exists and the password is a match.

We can now start with our routes. The first is the root route, this is little more than an API check, it just returns a json string of {status:ok}. The default content type returned by Sinatra normally depends on template language, but here were explicitly setting it to json. To allow users to register we set up a /register route using the post method. Again were setting the content type to json, then using exception handling to handle any errors when creating the user. The User.create method is called with the username, and the password parameters to generate a user. The plaintext password is passed through Digest::MD5.hexdigest to encode it as MD5 before saving to the database.

Now that a user has been created they will be able to save notes via the API. Using another post method, this time with the route /notes. We first call the protected! helper function to make sure the user is authenticated and again set the content type to json. The @user object is loaded in as part of the authorized? helper, via the relationship between the user and note models we can create a new note. Like the register route we use exception handling to return ok when successfully saving the note or the error if saving fails.

Once some notes have been created they can be returned. The same route of /notes will be used to return all notes, this time with the get method. Again with the @user object from the authorized? helper we fetch all notes for that user. If its not empty its returned as part of a json string, otherwise a json string

containing no notes is returned.

Next we will create a route to return individual notes. The path /notes/:id will allow us to get the second argument in the URL as params[:id]. With this we can fetch the first note of that id for the authorized user. Again if it exists we return it as a json string, otherwise returning no note.

Now that we can read each note well be looking to update and delete them. Using the put method we can update the notes. Just like the last route well use the id parameter to fetch note, we can then update the title if the parameter is set for it and the body if the parameter is set. Finally the save method is called on the note to save it. Well return the new note if saved, or the error if there was one during the update / save process.

To delete notes we use the delete method, again with the same /notes/:id route so we can query datamapper for the node of that id and simply appending .destroy to delete it. A successful deletion will return deleted in the json string, otherwise an error will be returned.

Using a similar config.ru and procfile as before this complete application can be deployed to Heroku, you can then query the API via ajax calls in javascript, native mobile applications or simply curl.

2.3 Summary

In this chapter we followed the full CRUD process of creating, reading, updating and deleting notes. These notes are all attached to a user, using a one to many relationship in datamapper. By setting the content type to json, and using the json gem, all data and errors are returned as a json string to be consumed by a third party, such as a native mobile app.

Chapter 3

Fist

In this chapter we will create an app to store and share code snippets in a similar way to Githubs Gist. However authentication will come via Twitter with tweeting of publicly accessible posts.

3.1 Twitter authentication

This application will authenticate via Twitter, therefore one of the first things you need to do is generate your key and secret. Head over to https://dev.twitter.com/apps and click Create a new application. Give your application a name, description and website URL, if youre running or even just testing locally the URL would just be http://127.0.0.1:5000/. A callback URL also needs to be set. Well use the omniauth gem with Sinatra to assist with authentication, the route /auth/twitter/callback, is suggested. When running locally the callback URL would be http://127.0.0.1:5000/auth/twitter/callback. Once created go into the Twitter application settings, set the access to Read and Write and check Allow this application to be used to Sign in with Twitter.

We now need to get the Consumer key and Consumer secret, youll find these on the details page of the Twitter application. These need to be added into a file called .env in the following format.

```
TWITTER_KEY=UiRVn9gn3FLRQd3H8fw1g
TWITTER_SECRET=sgoCf6oSjJKNaMmz1AfC81of9COP7ju2xR5cssP9Lzo
```

If we use foreman to load the application locally, all items in the .env file will be loaded as environment variables. Adding these type of details as environment variables rather than hardcoded in the application adds a layer of security and also allows different details to be used to development, testing and production. Run gem install foreman to install foreman ready. Hosting environments such as heroku allow the setting of environment variables for production use.

3.2 App

```
source "https://rubygems.org"
ruby "2.1.0"

gem "sinatra"
gem "dm-sqlite-adapter"
gem "data_mapper"
gem "sinatra-flash"
gem "omniauth"
gem "omniauth-twitter"
gem "twitter"
```

Again well use a Gemfile to manage the gems via bundler. The Sinatra gem will be needed again, along with datamapper and its sqlite adapter. The sinatra-flash gem allows us to store status messages, which are shown upon an action such as login, save and update. To authenticate with Twitter well use omniauth plus its Twitter strategy. Finally the Twitter gem will be used to send tweets using the token and secret obtained via omniauth.

app.rb require ‘sinatra/base’ require ‘sinatra/flash’ require ‘data_mapper’ require ‘omniauth’ require ‘omniauth-twitter’ require ‘twitter’

module FIST DataMapper.setup(:default, “sqlite:///#{File.expand_path(File.dirname(**FI**
class User include DataMapper::Resource

```
property :id,       Serial
```

```
property :uid,      String
property :username, String, :required => true, :unique_index
property :name,     String
property :token,    String
property :secret,   String

has n, :snippets   end
```

class Snippet include DataMapper::Resource

```
property :id,         Serial
property :title,      String, :required => true
property :body,       Text, :required => true
property :syntax,     String
property :created_at, DateTime
property :updated_at, DateTime

belongs_to :user   end   DataMapper.finalize   DataMapper.aut
```

end

The above code is added to app.rb, youll see a similar format to the previous chapter. Firstly requiring the gems then defining the FIST module. Within this we setup datamapper defining the sqlite database file. There are two models, User and Snippet, these have a one to many relationship. After the datamapper finalize and auto_upgrade! we can add the Sinatra class.

class App < Sinatra::Base enable :sessions, 'SinatraCookbook' enable :method_override register Sinatra::Flash

```
use OmniAuth::Builder do
  provider :twitter, ENV['TWITTER_KEY'], ENV['TWITTER_SECRET'
end

helpers do
  def current_user
```

```
    @current_user ||= User.get(session[:id]) if session[:id]
  end

  def protected!
    redirect '/' if !current_user
  end
end
```

end

Within the Sinatra App class we need to enable a couple of items. Sessions, so that we can store the users session, with this we also set a secret. Sinatra automatically generates a secret, but creating a custom one assures that its the same across instances. Method override also needs enabling to allow us to use the non-standard put method for editing of snippets. We also need to register the Sinatra flash class.

The OmniAuth::Builder block allows us to define Twitter application details, weve already got these stored in the .env file for local usage, when using Heroku these will need to be added as config variables. Within the helpers block well define two helper functions. current_user loads the user object based on the id from the users session, as long as the session id is set. protected! redirects the user to the root route if current_user returns false.

To the end of the App class we can add the first route.

```
get '/' do
  halt erb :anon if !current_user
  @snippets = current_user.snippets.all
  redirect '/new' if @snippets.empty?
  erb :index
end
```

The root route has three main actions. Firstly if the current_user helper returns false the Sinatra application will be halted and the anon.erb view will be returned, this view features a login link, linking to /auth/twitter, which is a route setup by omniauth. If the user is authenticated all snippets for that user

will be fetched from the database. If there are no snippets a redirect to the /new route will be done. If there are snippets then index.erb view will be returned.

index.erb, shown above, first we return the name, obtained from Twitter, of the user. It then features an each block to loop through the snippets and return the title as a link to the snippet and an edit link to the edit page, all as part of an unordered list.

Back in the App class of app.rb the next route is the Twitter authentication callback URL defined when we setup the Twitter application.

```
get '/auth/:name/callback' do
  auth = request.env['omniauth.auth']
  user = User.first_or_create({:uid => auth.uid},
        {:username => auth.info.nickname,
         :name => auth.info.name,
         :token => auth.credentials.token,
         :secret => auth.credentials.secret})
  session[:id] = user.id
  flash[:info] = "Welcome!"
  redirect '/'
end
```

First well grab the omniauth request data into the auth variable. Using this we can fetch or create the user. If a user exists with the Twitter user id this will be returned, otherwise a user will be created with username, token and secret from Twitter. The user id is stored in a session to keep the user authenticated. Finally we set the flash message Welcome! and a redirect to the root route is made. To return the flash messages <%= styled_flash %> is added to layout.erb. This will wrap flash messages in divs with ids and classes that can be used to style the messages.

```
get '/new' do
  protected!
  @snippet = current_user.snippets.new
  erb :new
```

```
end

post '/new' do
  protected!
  snippet = current_user.snippets.create(params[:snippet])
  flash[:info] = "Snippet saved"
  redirect "/#{snippet.id}"
end
```

The next set of routes, above, allow us to create a code snippet. The get method will display the form. First the protected! helper is loaded to make sure the user is authenticated, we then load an empty snippet into the @snippet variable before calling the new.erb view.

The form, when submitted, will send a post request to the /new route, after checking the user is authenticated we create the snippet then a redirect to the newly saved snippet page based on the incremental id.

The form in new.erb, shown above, will be used for both creating and updating snippets. We can tell if the form is being used for editing because the @snippet.id would be set, therefore in this instance well add a hidden field to allow the use of a put method. The form element names all follow a specific format, such as snippet[name], this allows for the data to be returned nested under the params hash. On the select list we need to specify which option is selected when editing a snippet, to do this we add a check for the item returned on each option.

```
get '/tweet/:id' do
  protected!
  snippet = current_user.snippets.get(params[:id])
  @message = "#{snippet.title} - #{request.env['rack.url_sche
  erb :tweet
end

post '/tweet/:id' do
  protected!
```

```
  snippet = current_user.snippets.get(params[:id])

  Twitter.configure do |config|
    config.consumer_key = ENV['TWITTER_KEY']
    config.consumer_secret = ENV['TWITTER_SECRET']
    config.oauth_token = current_user.token
    config.oauth_token_secret = current_user.secret
  end
  message = "#{snippet.title} - #{request.env['rack.url_schem
  client = Twitter::Client.new
  client.update(message)
  flash[:info] = "Tweeted"

  redirect '/'
end
```

The next two routes within the app relate to tweeting about a snippet. The get method checks the user is authenticated then queries for the snippet in the database with the id from the URL. A message consisting of title, URL and hashtag is generated, this will be returned as part of the view with a simple form and submit button to confirm the tweet. Submitting this form will send a post request handled by the post method. After checking the user is authenticated the snippet is loaded again. Twitter is configured using the key and secret from the environment variables, as used by omniauth, along with the users token and secret, which we stored when authenticating. The message is generated again, the twitter client is initialised and updated with the message. Finally the user is redirected back to the root route.

```
get '/:id/edit' do
  protected!
  @snippet = current_user.snippets.get(params[:id])
  erb :new
end
```

```
put '/:id/edit' do
 snippet = current_user.snippets.get(params[:id])
 snippet.update(params[:snippet])
 flash[:info] = "Snippet updated"
 redirect "/#{snippet.id}"
end
```

Well now look at editing snippets. The get method loads the snippet based on the id from the URL, this is passed on to the new.erb view, which we looked at earlier. The form will now submit to the put method. Here we simply load the snippet, then update it with the new parameters before redirecting to the newly edited snippet.

```
get '/:id' do
  @snippet = Snippet.get(params[:id])
  erb :show
end
```

The final route is displaying the snippets. This route needs to come last because its a single parameter. Routes in Sinatra are processed in order, if this route was to be at the start of the application URLs such as /new wouldve got caught by this route, thus throwing an error.

Within the route were not calling the protected! helper because we want complete snippets to be publically accessible to those clicking the tweeted link. The snippet is loaded into the @snippet variable, which is then returned within show.erb.

config.ru To load the application a config.ru file is needed, it simply requires the application file, then runs the app class.

require ‘./app’ run FIST::App

Procfile As mentioned earlier well use Forman to load the application locally along with the environment variables stored in .env, for this we will need a Procfile.

web: bundle exec rackup -p $PORT

This simply specifies the command used to run the application. Hosting providers such as Heroku also allow the use of procfiles to customise how the application is loaded, along with any workers.

Summary This chapter has followed many of the same principles as the API chapter, although this time a different authentication method and HTML views rather than JSON being returned. This type of application could easily be expanded to create a blog or simple CMS.

Chapter 4

SaaS

For this chapter we will create the authentication and subscription elements of a SaaS (software as a service) style application. Three plans will be defined within Stripe, which is the payment service well be using. Unlike other chapters, this time ActiveRecord will be used as the query interface instead of Datamapper. Also rather than using HTTP authentication or OmniAuth, as in previous chapters, this chapter will use custom authentication with bcrypt to hash passwords.

4.1 Stripe

First head over to https://stripe.com/ and get yourself a Stripe account. In the Plans section you can create three plans for the application. The id will be used as part of the URL when registering, the name will be shown on the sign up and account pages. You can choose a price and interval to suite your three plans. We will fetch these plans via the Stripe API to display on the sign up page. Then on registration well subscribe the user to one of these plans.

By default Stripe will start in a Test account, this is great for learning Sinatra with this application. If you want to build a real application and launch it, you will first need to activate your account. Activating the account requires website, company and bank account details.

Under account settings then API keys you will find both test and live API

keys. For now well just use the test ones. The test secret key will need to be added to your .env file and the test publishable key will need to be added to layout.erb. Well cover this later in the chapter when needed.

4.2 Application

4.2.1 Gemfile

As always the first step is to create a Gemfile containing all the gems needed for the application and install them using the command bundle install.

```
source "https://rubygems.org"
ruby "2.1.0"

gem "sinatra"
gem "sqlite3"
gem "activerecord"
gem "sinatra-activerecord"
gem "bcrypt"
gem "stripe"
```

We set the source to rubygems.org, specify the use of Ruby 2.1.0 and add the Sinatra gem. Again well use sqlite as the database, this time via the sqlite3 gem, rather than dm-sqlite-adapter, because were going to use activerecord as the query interface. Activerecord is the query interface for Ruby on Rails, although, its an independent library that can be used with Sinatra via the sinatra-activerecord gem. The bcrypt gem will be used to encrypt the users password along with a salt, an additional input used when hashing the password. Finally well use the stripe gem to allow easy access to the Stripe API.

4.2.2 rakefile.rb

To run the ActiveRecord rake commands a rakefile is needed. Create rakefile.rb with the following code.

```
require "./app"
require "sinatra/activerecord/rake"
```

The rake commands will allow us to generate database migrations and run them. Database migrations allow for the generation of versioned database changes, such as adding tables and columns.

4.2.3 app.rb

```
require 'sinatra/base'
require 'sinatra/activerecord'
require 'bcrypt'
require 'stripe'

module SAAS
  ActiveRecord::Base.establish_connection(
    :adapter => 'sqlite3',
    :database =>  'saas.db'
  )

  class User < ActiveRecord::Base
  end
end
```

After requiring all gems and defining the SAAS module, we can set up ActiveRecord. Using the establish_connection method we can define both the adapter as sqlite3, and the path to the database file. We then have the User class, which is used as a model. Within this model we could define any relationships, scopes or helper methods, however these are not needed for this application.

If you wished to deploy to Heroku and use postgresql you can define the database in the following way.

```
db = URI.parse('postgres://user:pass@localhost/dbname')

ActiveRecord::Base.establish_connection(
  :adapter  => db.scheme == 'postgres' ? 'postgresql' : db.scheme,
  :host     => db.host,
  :username => db.user,
  :password => db.password,
```

```
  :database => db.path[1..-1],
  :encoding => 'utf8'
)
```

For now well continue with sqlite.

4.2.4 Database migration

Using the rakefile created earlier, run the command rake db:create_migration NAME=create_users, this will generate a db folder containing the create_users.rb migration file. The filename is appended with the date and time the migration was created. Add the following code to this migration file.

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table(:users) do |t|
      t.string :email
      t.string :password
      t.string :salt
      t.string :stripe_card_token
      t.string :stripe_customer_id

      t.timestamps
    end
  end
end
```

The change method allows for reversible actions such as creating tables, the reverse would therefore be dropping the table. We create the users table, within this add a number of columns needed for the application, in this case each are of the type string. Finally timestamps are added, which creates two columns, created_at and updated_at. These are both managed by ActiveRecord and are populated when table entries are created or updated.

The command rake db:migrate will generate the sqlite database file and will run the database migration files, in this case creating the users table and all columns.

4.2.5 class App - app.rb

Back in app.rb we can now add the Sinatra application class after the User model class.

```
class App < Sinatra::Base
  enable :sessions

  helpers do
    def current_user
      @current_user ||= User.find(session[:user]) if session[:user]
    end

    def protected!
      redirect '/' if !current_user
    end
  end

  before do
    Stripe.api_key = ENV['STRIPE_API']
  end
end
```

The application will store the authenticated users id in a session, so well first enable sessions. We need a couple of helpers, first `current_user`, here well return the user object based on the user id in the session variable. The protected! helper will redirect to the root route if `current_user` returns false. Outside the helpers method we have a before filter, before filters are run before every route. Within the before filter well set the Stripe API key from an environment variable. To run locally you can set the Stripe key environment variable in a .env file in the format `STRIPE_API=sk_test_OQ8AxC5XWmdsT8tjyOrHjeyO`.

After the before filter we can add the first route.

```
get '/' do
  @plans = Stripe::Plan.all.data
  erb :index
end
```

The first route is the root path, we use the Stripe API to fetch all plans and set the returned data to an instance variable, which we can use in the view

called with erb :index. Within the view well use an each loop to go through each plan. We can then use the plan name and plan id to create a link to the registration page.

Three routes make up the registration functionality.

```
get '/register' do
  redirect '/'
end

get '/register/:plan' do
  redirect '/account' if current_user
  @plan = Stripe::Plan.retrieve(params[:plan])
  erb :register
end

post '/register/:plan' do
  customer = Stripe::Customer.create(email: params[:user][:email], plan: params[:plan], card: p
  password_salt = BCrypt::Engine.generate_salt
  password_hash = BCrypt::Engine.hash_secret(params[:user][:password], password_salt)

  user = User.create(:email => params[:user][:email],
         :password => password_hash,
         :salt => password_salt,
         :stripe_card_token => params[:user][:stripe_card_token],
         :stripe_customer_id => customer.id)
  session[:user] = user.id
  redirect '/account'
end
```

The first path, **`/register`**, is simply a redirect back to the root if no plan is specified. The next route then handles the selected plan passed as an id in the URL. If the **`current_user`** helper returns true, denoting the user being logged in, well redirect to the account page. Finally we call the register view containing the registration form. The registration form has a hidden **`user[stripe_card_token]`** field along with **`user[email]`** and **`user[password]`**. Card number, CVV code and expiry date fields are also added, but without a name attribute, because we dont want these details sent through to Sinatra, they will be processed via a Javascript call directly to Stripe. In the layout.erb view well need to add jQuery, either download it and add it to public/js, or point to one of the many CDN mirrors. Well also need to add the Stripe javascript **`<script type="text/javascript" src="https://js.stripe.com/v2/"></sc`** and your Stripe public key in the format **`<meta name="stripe-key" content="pk_te`**

Finally in layout.erb well need some custom javascript to post the card details to Stripe and fetch the card token.

```
<script type="text/javascript">
var subscription;

jQuery(function() {
                Stripe.setPublishableKey($('meta[name="stripe-key"]').attr('content'));
return subscription.setupForm();
});

subscription = {
  setupForm: function() {
    return $('#new_subscription').submit(function() {
      $('button[type=submit]').attr('disabled', true);
      if ($('#card_number').length) {
         subscription.processCard();
         return false;
       } else {
         return true;
       }
     });
   },
   processCard: function() {
     var card;
     card = {
       number: $('#card_number').val(),
       cvc: $('#card_code').val(),
       expMonth: $('#card_month').val(),
       expYear: $('#card_year').val()
     };
     return Stripe.createToken(card, subscription.handleStripeResponse);
   },
   handleStripeResponse: function(status, response) {
     if (status === 200) {
       $('#stripe_card_token').val(response.id);
       return $('#new_subscription')[0].submit();
     } else {
       $('#stripe_error').text(response.error.message);
       return $('input[type=submit]').attr('disabled', false);
     }
   }
 };
 </script>
```

First the Stripe public key from the meta tag set earlier is passed to the setPublishableKey method. The setupForm function is then loaded. Within this function the submit button is disabled and as long as the card number field

has been filled in the processCard function is called. Here all the card details are submitted to Stripe.createToken and subscription.handleStripeResponse is set to handle the response. If the status returns 200 confirming a successful response, the hidden card token field will be populated with the returned card token and the form will be submitted. Otherwise an error message returned from Stripe will be displayed and the submit button re-enabled.

Back in app.rb the final register route post '/register/:plan' handles the submission of the form. First the users email address, the plan and card token are submitted to the Stripe API via the Stripe::Customer.create method. By submitting a plan id, which matches a plan we already setup on the Stripe website a subscription so that plan will be created for this user. The email address is another identifier of the customer created, this will help you match your application user and Stripe customers. The card token will match all card details previously sent so that they can be charged each month.

Bcrypt is used to generate a salt, and then along with the password submitted on the registration form a hash is created. This hash and salt with the users email is saved via User model, as well as the Stripe card token and customer id. The newly created users id is stored in a session before redirecting to the account page.

The next route in app.rb handle the user login.

```
get '/login' do
  erb :login
end

post '/login' do
  user = User.find_by(:email => params[:user][:email])
  if user.password = BCrypt::Engine.hash_secret(params[:user][:password], user.salt)
    session[:user] = user.id
    redirect '/account'
  else
    redirect '/'
  end
end
```

The get method just loads the login.erb view, which is simply a form with email and password fields. This form will send a post request through to the /login route. Here we first find the user by email address. If the password in

that user object matches the hashed version of the submitted password, the user id is set to a session and the request it redirected to the account page. Failing a password match, a redirect to the root route takes place.

The final route of app.rb is for the account page.

```
get '/account' do
  protected!
  @customer = Stripe::Customer.retrieve(current_user.stripe_customer_id)
  erb :account
end
```

The account route starts with the protected! helper function, to make sure the user is authenticated. Then using the Stripe API well retrieve the customer details. The current_user helper function allows us to load the user object of the currently authenticated user, from here we can get the stripe customer id to pass to Stripes retrieve method. This is assigned to an instance variable so we can return the data in the account.erb view.

4.2.6 account.erb

```
Email: <%= @customer.email %><br/>
Plan: <%= @customer.subscriptions.data[0].plan.name %><br/>
Current period end: <%= Time.at(@customer.subscriptions.data[0].current_period_end).strftime("%
Card type: <%= @customer.cards.data[0].type %>
```

The account page is an example of what kind of data we can depend on Stripe for. By using the @customer instance variable, set in app.rb, we can return and make use of a lot of customer information. Such as the email address, plan subscribed to, when the subscription ends (this would most often be the next billing date) and the card type used. Youll notice that @customer.subscriptions.data and @customer.cards.data are arrays. This is because in Stripe a customer can have more than one subscription and more than one card. In our SaaS application wed expect users to only have one subscription and one card, so its fairly safe to hardcode the first item from the data array.

4.3 Summary

In this chapter we covered using ActiveRecord to connect to the database, using models and migrations. Users subscribing to the application was managed by Stripe using the Stripe API to send card details and manage recurring payments. We also used Bcrypt to create a hash of the users password and build a custom authentication method.

Chapter 5

Gallery

As part of this chapter well be creating a Sinatra based image gallery application. Images will be stored on Amazon S3, this will be done via the CarrierWave and Fog gems. ActiveRecord will again be used to connect to the database, and ImageMagick will be used to generate image thumbnails.

5.1 Gemfile

The Gemfile should start to be quite familiar now. Here we have three gems not previously used. Carrierwave is a gem that assists with the upload of files. Fog is a cloud services library, which will help us connect to Amazon S3. Finally mini_magick is a ruby based wrapper for ImageMagik.

```
source "https://rubygems.org"
ruby "2.1.0"

gem "sinatra"
gem "sqlite3"
gem "activerecord"
gem "sinatra-activerecord"
gem "carrierwave"
gem "fog"
gem "mini_magick"
```

5.2 ImageMagik

If youre planning to host on Heroku youll already have ImageMagik there ready and waiting. However this application is initially written to run locally. If installing on Mac OS then Brew or MacPorts is the easiest way to install ImageMagik. Your package manager on Linux will allow you to install it. Last but not least, for Windows theres executables available from imagemagick.org.

5.3 Rakefile

The same Rakefile as the previous chapter will be needed to create and run database migrations for ActiveRecord.

```
require "./app"
require "sinatra/activerecord/rake"
```

Before being able to run any rake commands well need to first create at least the required app.rb.

5.4 App

Firstly all the gems are required, youll notice carrierwave being added in two parts. The first is the main carrierwave library, the second is the ActiveRecord integration. Carrierwave also supports a number of other ORMs, and can work in a similar way if using Datamapper.

```
require 'sinatra/base'
require 'sinatra/activerecord'
require 'carrierwave'
require 'carrierwave/orm/activerecord'
require 'mini_magick'

module GALLERY
  ActiveRecord::Base.establish_connection(
    :adapter => 'sqlite3',
```

```
    :database =>  'gallery.db'
  )

end
```

Once all of the gems have been added we can declare the gallery module. Within this the ActiveRecord connection is established, again using sqlite as the adapter. This can always be switched to Postgres if deploying on Heroku.

Now that we have an application the rake command can be run to create a migration.

```
rake db:create_migration NAME=create_image
```

The above command will create the create_image.rb migration file within the db folder. Add the following code to this file.

```
class CreateImage < ActiveRecord::Migration
  def change
    create_table(:images) do |t|
      t.string :file

      t.timestamps
    end
  end
end
```

Here we define a change method in which **create_table** is called. This will generate the images table when **rake db:migrate** is run and drop the images table when rake **db:rollback** is run. Within the table a file column is added with the type string, along with the timestamps **created_at** and **updated_at**.

Back in app.rb the following carrierwave and fog configuration needs to be added to the gallery module.

```
CarrierWave.configure do |config|
  config.fog_credentials = {
    :provider               => 'AWS',
```

```
    :aws_access_key_id      => ENV['AWS_KEY'],
    :aws_secret_access_key  => ENV['AWS_SECRET'],
  }
  config.fog_directory  = 'sinatracookbook'
  config.fog_attributes = {'Cache-Control'=>'max-age=315576000'}
end
```

Within the CarrierWave.configure block we set a number of settings. The credentials for fog are set as a hash. Were using Amazon Web Services, so AWS is set as the provider along with the API key and secret. You will find these keys at https://portal.aws.amazon.com/gp/aws/securityCredentials after youve signed up for an AWS account. Now head to https://console.aws.amazon.com/s3 and create an S3 bucket. The S3 bucket name is set as the **fog_directory**. Finally **fog_attributes** is set with a cache-control header for the images, here the max age is set to cache the images in the browser for 10 years.

Next we need an uploader class for CarrierWave.

```
class Uploader < CarrierWave::Uploader::Base
  include CarrierWave::MiniMagick

  storage :fog
  def store_dir
    'images'
  end

  def filename
    "#{secure_token}.#{file.extension}" if original_filename.present?
  end

  version :thumb do
    process :resize_to_fill => [200, 200]
  end

  protected
  def secure_token
    var = :"@#{mounted_as}_secure_token"
    model.instance_variable_get(var) or model.instance_variable_set(var, SecureRandom.uuid)
  end
end
```

The Uploader class inherits from CarrierWave::Uploader::Base. First MiniMagick is included so we can make use of the ImageMagick functionality.

The storage is set to use fog, and therefore uploaded to S3 as specified in the configuration. This could be changed to storage :file to store files locally, however this would not work on Heroku due to its ephemeral filesystem. To separate out files on S3, folders can be generated, by setting a storage directory, or **store_dir**. In this case we set it to **images**. To account for files being uploaded with the same filename and overwriting each other, a filename can be set. If the **original_filename** is present the filename is set to the result of the **secure_token** method followed by the initial file extension. The **secure_token** method is added as a protected method at the end of the uploader. Here we get the instance variable if set or set the instance variable to a UUID (universally unique identifier) using Rubys SecureRandom class. Lastly in the uploader we generate a thumbnail of each image uploaded using the version method. We name the version **thumb**, this will also be prefixed to the filename of the resized image. Within the version the process is set to **resize_to_fill** resizing and cropping the image to specific dimensions, in this case 200 x 200 pixels. Along with the thumb version the original will also be uploaded.

```
class Image < ActiveRecord::Base
  extend CarrierWave::Mount
  mount_uploader :file, Uploader
end
```

The next class added in app.rb is the an ActiveRecord model, Image, for the images table. Initially **CarrierWave::Mount** is added, we then call **mount_uploader**. The file field name and uploader class name passed in to define which uploader should process the file.

Finally the sinatra App class.

```
class App < Sinatra::Base
  get "/" do
    @images = Image.all
    erb :index
  end

  get "/upload" do
    erb :upload
```

```
  end

  post "/upload" do
    Image.create(params[:image])
    redirect '/'
  end
end
```

The first route uses the get method with the root path. All images are fetched and set to the

The next route in the App class is a get method for /upload, this simply loads the upload.erb view. This view contains the most basic of file upload form:

```
<form action='/upload' method='post' accept-charset="utf-8" enctype="multipart/form-data">
  <label for='image_file'>Image</label>
  <input type="file" name="image[file]" id="image_file" />
  <button type='submit'>Save</button>
</form>
```

Here it submits a post request to /upload, which is the final Sinatra route in app.rb. For this route we pass params[:image], which is all the parameters from the upload form, to Image.create, the standard ActiveRecord method for creating an instance of a model. CarrierWave will handle the whole process of uploading the file to S3, via fog, and storing the relevant data in the database. Finally a redirect to the root route is done to display the newly uploaded image.

5.5 Summary

To complete the application, and allow it to run on Heroku well need config.ru and Procfile.

5.5.1 config.ru

```
require './app'
run GALLERY::App
```

5.5.2 Procfile

```
web: bundle exec rackup -p $PORT
```

Albeit simply, this application allows for a public image gallery to be created. Image uploads are handled via CarrierWave, and via Fog uploaded to Amazon S3. CarrierWave integrates seamlessly with ActiveRecord, but can also work with Datamapper. ImageMagik is used along with the mini-magik gem to automatically generate image thumbnails.

Chapter 6

Invoices

Using PDFs can often be a great way to share structured data such as reports, proposals or invoices. This application will allow users to input fixed data such as personal and company information, and then generate PDF invoices with a number of line items.

6.1 Prawn

Prawn is a Ruby library for generating PDFs. As an initial example install the gem by running gem install prawn, then create pdf.rb with the following code.

```
require "prawn"

Prawn::Document.generate("hello.pdf") do
  text "Hello World!"
end
```

Firstly Prawn is required, a document is then generated with the name hello.pdf. Within this the text Hello World! is added. Run this with the command ruby pdf.rb. Once complete you will find the file hello.pdf, which will be a full page containing the text Hello World! at the top.

6.2 Gemfile

As always we start with the Gemfile.

```
source "https://rubygems.org"
ruby "2.1.1"

gem "sinatra"
gem "sqlite3"
gem "activerecord"
gem "sinatra-activerecord"
gem "userbin"
gem "prawn"
```

Youll see again were using ActiveRecord for the database connection. Were also adding Userbin, which is a third party user authentication and management server, and also Prawn, which well be using to generate the PDFs. Install these gems by running bundle install.

6.3 App

Well now start writing the application in app.rb.

```
require 'sinatra/base'
require 'sinatra/activerecord'
require 'userbin'
require 'prawn'

module INVOICES
  ActiveRecord::Base.establish_connection(
    :adapter => 'sqlite3',
    :database =>  'invoices.db'
  )

end
```

First requiring all of the gems, then creating the invoices module where the ActiveRecord database connection can be established. For this were using Sqlite but again Postgres can be used if deploying to Heroku.

6.4 Database

Now we have an initial shell of an application the database tables can be setup, again well use ActiveRecord migrations. Adding the same Rakefile as previous chapters will expose the commands needed. The following three commands will create the migration files for the three tables:

```
rake db:create_migration NAME=create_user
rake db:create_migration NAME=create_invoices
rake db:create_migration NAME=create_lineitems
```

Within these migration files add the following code.

6.4.1 create_users.rb

Here were creating the user table, this will act as a cache of the data stored on Userbin, which were using for authentication, and also details needed for the PDF such as address.

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table(:users) do |t|
      t.string :userbin_id
      t.string :email
      t.string :name
      t.text :address

      t.timestamps
    end
  end
end
```

6.4.2 create_invoices.rb

The invoices will directly relate to a user, therefore the user_id needs to be stored as an integer along with a name for the invoice as a string.

```
class CreateInvoices < ActiveRecord::Migration
  def change
    create_table(:invoices) do |t|
      t.integer :user_id
      t.string :name

      t.timestamps
    end
  end
end
```

6.4.3 create_lineitems.rb

Each invoice will have many line items associated with it, stored in the lineitems table. These will reference the invoice with the invoice_id. The line items will also store the name of the item and the value.

```
class CreateLineitems < ActiveRecord::Migration
  def change
    create_table(:lineitems) do |t|
      t.integer :invoice_id
      t.string :name
      t.decimal :value

      t.timestamps
    end
  end
end
```

All of these migrations can be run using the rake db:migrate command.

6.5 App - part 2

Now that the database tables have been created we can look at the next part of the application, configuring Userbin. Within the invoices module add the following code.

```
Userbin.configure do |config|
  config.app_id = ENV['USERBIN_ID']
  config.api_secret = ENV['USERBIN_SECRET']

  config.root_path = "/"
  config.protected_path = "/account"

  config.find_user = -> (userbin_id) do
    User.find_by userbin_id: userbin_id
  end

  config.create_user = -> (profile) do
    User.create! do |user|
      user.userbin_id = profile.id
      user.email      = profile.email
    end
  end
end
```

Here we set a number of config variables within the Userbin configure block. Firstly the Userbin id and secret are set, these are loaded from environment variables. Locally these can be added to a .env file, and on Heroku added via the config:set command. To get an id and secret, sign up on https://userbin.com.

The **root_path** is then set to denote where the user should be redirected on logout. The protected_path is both a path that requires authentication and where the user is redirected after login.

The **find_user** config method allows an action to be setup upon login. Here were finding a user from the user table based on the **userbin_id**. Similarly the **create_user** method sets an action to be run when a user is created. Here we add the user to the user table with the id from Userbin and email address they entered on registration.

```
class User < ActiveRecord::Base
  has_many :invoices
end

class Invoice < ActiveRecord::Base
  belongs_to :user
  has_many :lineitems
  accepts_nested_attributes_for :lineitems, allow_destroy: true, reject_if: lambda {|attributes
end
```

```
class Lineitem < ActiveRecord::Base
  belongs_to :invoice
end
```

Below the Userbin configuration, setup the three models that relate to the three tables set earlier. The first is **User**, within this its specified that a user can have many invoices using the **has_many** method. The **Invoices** model is then defined first adding the association with the User model using **belongs_to**, this creates a one to many relationship. The association with **lineitems** is then added using **has_many**, before finally adding **accepts_nested_attributes_for**. This allows us to add **lineitems** using one **Invoice.create** method. We also add **allow_destroy** to allow the deletion of **lineitems** via invoices, and **reject_if** to prevent a **lineitem** being added if the name is blank. The last model is **Lineitem**, here we simply reciprocate the association back to **Invoice** with **belongs_to**.

We can now create the Sinatra application class.

```
class App < Sinatra::Base
  use Userbin::Authentication
  enable :method_override

  helpers do
    def authorize!
      unless Userbin.user_logged_in?
        redirect '/login'
      end
    end
  end

end
```

Firstly we tell Sinatra to use Userbin for authentication. We then enable method_override, allowing us to use Sinatras put method for editing invoices. The final section in the code above is Sinatra helpers. In this app we only have one helper, **authorize!**. If the user is not logged in via Userbin they will be redirected to the **/login** route.

Below are an initial four routes to add below the helpers in the App class.

```
get '/' do
  if Userbin.user_logged_in?
    redirect '/account'
  end
  erb :index
end

get '/login' do
  erb '<body><a class="ub-login-form"></a></body>', :layout => false
end

get '/account' do
  erb :account
end

post '/account' do
  Userbin.current_user.update(params[:account])
  redirect '/account'
end
```

The first route is for the root path. If the user is logged in they are instantly redirected to the account page, otherwise the index.erb view is loaded. This view simply features a sign up link, but could have information about the application, what it does, its features and why you should sign up.

The next route is the one we redirected to in the authorize helper. Here were not returning a view, the html is set inline, and layout is set to false to stop layout.erb from loading. The empty link with the class ub-login-form causes Userbin to load a login form via javascript which is injected into the page.

We then have two routes for /account, the first using the get method the second using the post method. The get method simply loads the account.erb view which features a form to update user information such as name and address. These will be returned on each invoice generated. The post method then uses the ActiveRecord update method to update all account details for the current Userbin user. The request is then redirected back to the account page.

```
get '/invoice' do
  authorize!
  @invoices = Userbin.current_user.invoices.all
  erb :invoices
end
```

```
get '/invoice/new' do
  authorize!
  @invoice = Userbin.current_user.invoices.new
  3.times do
    @invoice.lineitems.new
  end
  erb :invoice
end

post '/invoice/new' do
  authorize!
  invoice = Userbin.current_user.invoices.create(params[:invoice])
  redirect "/invoice/#{invoice.id}"
end
```

The next three routes are for listing and creating invoices. The first route initially calls the **`authorize!`** helper just to make sure the user is logged in before fetching the invoices. The currently logged in user can be fetched using **`Userbin.current_user`**, the invoices for this user can then be returned by using the **`invoices.all`** methods. These invoices are stored in an instance variable which can be looped through using an each method and returned in invoices.erb.

The **`/invoices/new`** route returns the invoice form in invoice.erb, first confirming the user is logged in with the **`authorize!`** helper. A new invoice is created for the current user and set to an instance variable. For this invoice three **`lineitems`** are then created using the association set previously in the ActiveRecord model.

The form for creating invoices in invoice.erb has been written in a way to handle the current creation of invoices, but also the later editing of invoices. In one form it handles the creation of **`invoices`** and **`lineitems`**, adding the association between the two.

```
<form method='post' accept-charset="utf-8">
  <% if @invoice.id %>
    <input name="_method" value="put" type="hidden">
  <% end %>
  <label for='name'>Name</label>
  <input type="text" name="invoice[name]" id="name" value="<%= @invoice.name if @invoice.name %
  <% @invoice.lineitems.each_with_index do |lineitem,id| %>
    <input type="hidden" name="invoice[lineitems_attributes][][id]" value="<%= lineitem.id if l
    <label for='lineitem_<%= id %>_name'>Item name</label>
```

```
    <input type="text" name="invoice[lineitems_attributes][][name]" id="lineitem_<%= id %>_name
    <label for='lineitem_<%= id %>_value'>Value</label>
    <input type="text" name="invoice[lineitems_attributes][][value]" id="lineitem_<%= id %>_val
  <% end %>
  <button type='submit'>Save</button>
  <% if @invoice.id %>
    <a href="/pdf/<%= @invoice.id %>">Create PDF</a>
  <% end %>
</form>
```

Youll see if the invoice id is set a hidden _method input is added to the form with the value put. This only gets added if the form is being used to edit an existing invoice, and thus having an id. It will submit the form using a put method instead of a post method.

Further down in the form you will see invoice **lineitems** are looped through with an **each_with_index** method. Even though for a new invoice the **lineitems** are empty, we already have three that were created in the main app. The elements for the **lineitem** are returned within this loop and the name is set with **[lineitems_attributes][]**, this adds each **lineitem** to an array once passed back to the app and because we set **accepts_nested_attributes_for** back in the model, ActiveRecord will know exactly what to do with it and store the **lineitems** with an association to their invoice.

At the bottom you will see a link to create the PDF, this again has an if statement around it to make sure its only returned for existing invoices or ones with an id.

```
get '/invoice/:id' do
  authorize!
  @invoice = Userbin.current_user.invoices.find(params[:id])
  3.times do
    @invoice.lineitems.new
  end
  erb :invoice
end

put '/invoice/:id' do
  authorize!
  invoice = Userbin.current_user.invoices.find(params[:id]).update(params[:invoice])
  redirect "/invoice/#{params[:id]}"
end
```

The further two routes handle the editing of an existing invoice. First a get method passing the id in the URL and finding the invoice for this id by the current user. Three **lineitems** are again created to give some blank fields if more **linitems** need to be entered, before loading the invoice.erb form shown earlier.

The put method will be called on form submission because of the hidden field set. It first finds the invoice based on the id in the URL, and then updating it with the parameters sent from the form. On completion the user will be redirected back to the form for the invoice they just edited.

```
get '/pdf' do
  authorize!
  redirect '/invoices'
end

get '/pdf/:id' do
  authorize!
  content_type 'application/pdf'
  invoice = Userbin.current_user.invoices.find(params[:id])
  Pdf.new(invoice).render
end
```

The final two routes handle PDF generation. First, the route /pdf is just there to prevent a 404 error if the id is removed from a PDF URL. It checks the user is logged in using the **authorize!** helper, then redirects back to the invoices page. The next route uses the same /pdf path but with the invoice id appended. After authorizing the user, the content type is set to PDF so the browser knows its loading a PDF file. The invoice is loaded based on the id passed in the URL, this again is taken using the association with the current user to make sure another users invoice isnt loaded. Finally a new instants of the Pdf class is rendered with the fetched invoice. So next, within the invoices module, but outside the Sinatra app class we need to add the Pdf class.

```
class Pdf < Prawn::Document
  def initialize(invoice)
    super()
    move_down 50
    text invoice.name, size: 15, style: :bold
    move_down 100
```

```
    text invoice.user.name, style: :bold, align: :right
    text invoice.user.address, align: :right
    text invoice.user.email, align: :right
    move_down 50
    table lineitems do
      row(0).font_style = :bold
      self.header = true
      self.row_colors = ['DDDDDD', 'FFFFFF']
      self.column_widths = [440, 100]
    end
  end

  def lineitems
    [['Name', 'Price']] +
      @invoice.lineitems.map do |lineitem|
      [lineitem.name, lineitem.value]
    end
  end
end
```

This Pdf class inherits from the Prawn document class allowing us to make use of Prawns PDF generation features. First an **initialize** method is defined with the invoice passed into it. Within this **super()** is used to pull in all items from the original Prawn document **initialize** method. The content for the PDF can then be added, **move_down** moved the cursor in the document down by the specified number of lines before we add the invoice name, user name, user address and user email. These are each given various settings to set the size, weight and alignment of the text. A table of line items is added, the first row is set to bold, a header setting is enabled, alternate colours for the rows are defined and widths for the two columns are added. The data for the table is stored in an array which is generated in the separate **lineitems** method. Name and price are added as the header titles for the two columns and the map method is used for the invoices line items to generate an array with the **lineitem** name and value. This PDF will then be rendered back in the route with the render method displayed or downloaded depending on the browsers settings.

As always to run the application we add a config.ru file and Procfile to the application directory.

6.6 Summary

In this chapter we covered using Userbin for user authentication and saving the information from there in a User table via ActiveRecord. Using associations we were able to tie invoices to users and `lineitems` to invoices. The `accepts_nested_attributes_for` option in the invoices model allowed us to create and update invoices in one simple method just by setting the attributes in the form field names. Finally the PDF was generated using Prawn rendering a number of lines of text and a two column table.

Chapter 7

Invite

Many startups build applications with an invite system where users can register for an invite. Then days, weeks or months later, they receive their invite to register for the application. In this chapter well create an application allowing users to register for their invite by entering their email address. A random 16 digit code will be generated along with the invite, which will be used to access it later. Emails will be generated and sent to the user to come back and register.

7.1 Gemfile

```
source "https://rubygems.org"
ruby "2.1.1"

gem "sinatra"
gem "sqlite3"
gem "activerecord"
gem "sinatra-activerecord"
gem "bcrypt"
gem "rack-flash3"
gem "pony"
```

The Gemfile will state the gems were using for this application, as a fairly simple application many of the gems are ones weve used before. Bcrypt will be used again to encrypt passwords for a custom registration system. Rack-flash3

will be used to show flash messages denoting errors or notices. Pony will be used to send the invite emails.

7.2 App

In app.rb well start the application by requiring all needed items. A module INVITE is created to house the whole application and this begins with establishing the ActiveRecord connection. Here well use sqlite again, and name the database file invite.db.

```
require 'sinatra/base'
require 'sinatra/activerecord'
require 'rack-flash'
require 'bcrypt'

module INVITE
  ActiveRecord::Base.establish_connection(
    :adapter => 'sqlite3',
    :database =>  'invite.db'
  )

  class Invite < ActiveRecord::Base
  end

  class User < ActiveRecord::Base
  end

  class App < Sinatra::Base
  end
end
```

After the ActiveRecord connection is configured we can define two methods. Invite, which will be used to store people who have requested an invite. User, which will be used to store users after accepting their invite. The final class defined is the Sinatra app class.

7.3 Database

To create the database migrations for ActiveRecord we can use the same Rakefile as previous chapters and run the following commands:

```
rake db:create_migration NAME=create_invite
rake db:create_migration NAME=create_user
```

Now the migration files have been created we can define the tables to be created

7.3.1 create_invite

Here well add a change method to the CreateInvites class to create the invites table. Theres an email column to store the address of the user who signed up, a boolean field to mark if the invite has been sent or not, then a random code well generate to be used on registration. Finally timestamps are added so we know the date and time the invite was created or updated.

```
class CreateInvites < ActiveRecord::Migration
  def change
    create_table(:invites) do |t|
      t.string :email
      t.boolean :sent
      t.string :code

      t.timestamps
    end
  end
end
```

7.3.2 create_user

The users table is created in the same way. The email will be stored again after registration along with the hashed password and the salt used in hashing. Just like the invites table timestamps are also added.

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table(:users) do |t|
      t.string :email
      t.string :password
```

```
      t.string :salt

      t.timestamps
    end
  end
end
```

7.4 App - routes

Back in the Sinatra app class we enable sessions, as these will be needed for rack-flash to work. We then setup rack-flash with the use method. Setting sweep to true will make sure old messages automatically expire if not viewed.

```
enable :sessions
use Rack::Flash, :sweep => true
```

Three helper functions are then defined.

```
helpers do
  def current_user
    @current_user ||= User.find(session[:user]) if session[:user]
  end

  def protected!
    if !current_user
      flash[:error] = "You must be logged in."
      redirect '/'
    end
  end

  def admin!
    if current_user.id != 1
      flash[:error] = "Admin only!"
      redirect '/'
    end
  end
end
```

The first, **current_user**, gets user object for the logged in user if the user id is set in a session. **Protected!** redirects to the root route and adds

a rack-flash error message if **current_user** returns false. **Admin!** similarly redirects to the root route and adds an error message, although if the current user is not id 1. Well use user id 1 as being the admin user, the user who can send invites.

```
get '/' do
  erb :index
end

post '/' do
  if !params[:email].blank?
    Invite.create(
      email: params[:email],
      code: SecureRandom.urlsafe_base64
      )
    flash[:notice] = "Thanks for registering."
  else
    flash[:error] = "You need to enter and email address."
  end
  redirect '/'
end
```

The root route simply just returns the index.erb view. In this view we have a simple form with an email field and submit button, which sends a post request back to the root path. In the post method we first check the email field was filled, if it was an invite is created. The email address from the form is passed in and a code is created using Rubys **SecureRandom** module. As the code will later be used as part of a URL the **SecureRandom.urlsafe_base64** method is used to generate the random code in a format safe to use in URLs. A rack-flash message is then created to thank the user for registering for an invite. If the email was not entered an error message will be set. Finally, if the email was set or not the user is redirected back to the root get method where the rack-flash messages will be displayed back with the form.

To display the rack-flash messages the following is added to layout.erb.

```
<% if flash.has?(:notice) %>
  <p class="notice"><%= flash[:notice] %></p>
<% end %>
<% if flash.has?(:error) %>
  <p class="error"><%= flash[:error] %></p>
<% end %>
```

Here we check if the flash hash contains a notice or an error. These are then returned as a <p> with the relevant class which can be styled appropriately.

The next routes in the Sinatra app class handle the registration.

```
get '/register/:code' do
  if invite = Invite.where(code: params[:code], sent: true).first
    @email = invite.email
    erb :register
  else
    flash[:error] = "Invalid code"
    redirect '/'
  end
end

post '/register/:code' do
  if invite = Invite.where(code: params[:code], sent: true).first
    password_salt = BCrypt::Engine.generate_salt
    password_hash = BCrypt::Engine.hash_secret(params[:user][:password], password_salt)

    user = User.create( :email => params[:user][:email],
                        :password => password_hash,
                        :salt => password_salt)
    session[:user] = user.id
    flash[:notice] = "Thanks for registering."
    redirect '/dashboard'
  else
    flash[:error] = "Invalid code"
    redirect '/'
  end
end
```

The get method is where well load the registration form via the register.erb view. However, first we check to see if the code in the registration url exists in the invites table and the invite has been sent. This is done by fetching the first invite matching the code and where sent is true. If this returned a record an instance variable will be set with the email address from the invite to try and enforce this email address being used again for the registration. If the invite query returns nothing the user is redirected back to the root route with an error message via rack-flash.

The post method is where the registration form is submitted. Again, even when the form is posted well check if the code in the URL is valid in the same way as in the get method. Returning the same error and redirect if no invite is returned. If all is well with the invite code a salt is generated using **Bcrypt**

and the password is hashed. We then create the user record using the submitted email address along with the hashed password and generated salt. Finally the user id of the newly created user is set to a session variable and the user is redirected to their dashboard.

```
get '/dashboard' do
  protected!
  erb :dashboard
end
```

In this case the dashboard simply returns a basic view, but here were making use of the **protected!** helper method to make sure only registered users can access the dashboard.

The admin section of the application features three routes. The first is a basic menu for all admin items. The admin! helper is used to confirm the user has id 1 and is therefore the admin user, the admin.erb view is then loaded with a link to the /admin/invites route.

```
get '/admin' do
  admin!
  erb :admin
end

get '/admin/invites' do
  admin!
  @invites = Invite.where(sent: [false, nil]).all
  if @invites.empty?
    return erb "<h2>No unsent invites</h2>"
  end
  erb :invites
end

post '/admin/invites' do
  admin!
  require 'pony'
  params[:invite].each do |invite|
    user_invite = Invite.find(invite).first
    options = {
      :to => user_invite.email,
      :from => "tim@millwoodonline.co.uk",
      :subject => "your invite",
      :body => "Come try our app http://#{request.host}/register/#{user_invite.code}",
      :via => :smtp,
```

```
      :via_options => {
        :port =>            '587',
        :address =>         'smtp.mandrillapp.com',
        :user_name =>       ENV['MANDRILL_USERNAME'],
        :password =>        ENV['MANDRILL_APIKEY'],
        :domain => request.host,
        :authentication => :plain,
        :enable_starttls_auto => true
        }
      }

    Pony.mail(options)
    user_invite.update(sent: true)
  end
  flash[:notice] = "Invites sent."
  redirect '/admin/invites'
end
```

The /admin/invites route features a form to allow invites to be sent to registered users. First we check the user is an admin with the admin! helper. Next we get all of the invites, however we dont want the invites that have already been sent. If we were to use the syntax **`where(send: false)`** or **`where(send: !true)`** this would not take into account most of the invites because when we create the invite were not actually setting the sent flag to false, therefore this is normally nil. So to overcome this we use **`where(send: [false, nil])`** to look for all the invites that have the sent flag set to false, or not set at all. If this query returns an empty arrat well just display a message noting there are no unsent invites. If the invites query returns some invites the invites.html.erb view is returned where we can loop through the invites returning email address and code, with a checkbox for the invite to be sent. This form submits to the /admin/invites post method.

In the post method we again check the user is an admin, then require the pony gem. It was not needed until now, so no need to require it any earlier. Each of the invite ids submitted on the form are looped through and a query is done to fetch the first invite matching the id. We can now build an options hash with all the elements needed for pony to send the email. The to address is set as the one from the invite, and the body of the email will feature the URL including code where the user can register. Were using Mandrill as the service to send the email so adding in their SMTP details including a username and api key as

environment variables. When working locally these can be stored in .env file and loaded with foreman as we have in previous chapters. Calling Pony.mail with these options will then send the email inviting the user to register. After this we set the invite as send and redirect back to the admin invites form.

7.5 Procfile and config.ru

Similar to other chapters the Profile and config.ru are both used to load the application. The only change for this application would be to update config.ru with the INVITE module name

7.6 Summary

In this chapter we build a complete invite and registration system for an application. It allows users to add their email address requesting an invite. The invite is generated there and then with a random url safe code being stored against their email address. Although the users will only see this when an admin sends the invite from the admin section of the application. Emails are sent using the Pony gem via the Mandrill service from Mailchimp. However any SMTP service, such as Sendgrid or even Gmail, would work just fine with Pony as long as it allows sending as the from address set in the options. Once the user has their invite they can register, creating an account with the email address they requested the invite for and a hashed version of the password they set.

Chapter 8

Traditional Ecommerce

Many of the worlds web sites are based around the sale of physical products, as more and more people shop online this will only increase over time. In this chapter will create a simple ecommerce site, well use Stripe to handle the credit card payments and ElasticSearch to allow customers to search for products.

8.1 Gemfile

```
source 'https://rubygems.org'
ruby '2.1.0'

gem 'sinatra'
gem 'sqlite3'
gem 'activerecord'
gem 'sinatra-activerecord'
gem 'elasticsearch-model'
gem 'stripe'
gem 'bcrypt'
gem 'json'
```

All of these gems weve used previously in other chapters apart from elasticsearch-model. This is actually a gem that makes up part of elasticsearch-rails, which is now the prefered method of connecting to an elasticsearch instance.

8.2 App

Well setup the app again in app.rb, first requiring all of the gems from the gemfile, then creating a module for the application. Much like other chapters where weve used ActiveRecord and the connection is established to the SQLite database as the first item in the module. Again we could use Postgresql if deploying to somewhere such as Heroku but SQLite makes things easy when working locally. The Elasticsearch client can then be configured. Here all were adding is logging, Elasticsearch will default to looking to localhost as the host. If youre using an external Elasticsearch provider such as Bonsai, the host will need to be set here too.

```
require 'sinatra/base'
require 'sinatra/activerecord'
require 'bcrypt'
require 'json'
require 'stripe'
require 'elasticsearch'
require 'elasticsearch/model'

module ECOMMERCE
  ActiveRecord::Base.establish_connection(
    :adapter => 'sqlite3',
    :database =>  'ecommerce.db'
  )
  Elasticsearch::Model.client = Elasticsearch::Client.new(
    log: true
  )

end
```

Now we have the barebones of the application we can create the migrations for ActiveRecord to generate the database tables. The same rakefile as preview chapters can be used to give the rake commands needed.

```
rake db:create_migration NAME=create_users
rake db:create_migration NAME=create_products
rake db:create_migration NAME=create_orders
rake db:create_migration NAME=create_ordersproducts
```

Run the above commands from within your application directory. This will generate the four migrations to create the four database tables needed in this application.

8.2.1 create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table(:users) do |t|
      t.string  :email
      t.string  :password
      t.string  :salt
      t.boolean :admin

      t.timestamps
    end
  end
end
```

8.2.2 create_products.rb

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table(:products) do |t|
      t.string  :name
      t.text  :description
      t.integer  :price

      t.timestamps
    end
  end
end
```

8.2.3 create_orders.rb

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table(:orders) do |t|
      t.integer :user_id
```

```
      t.string  :email
      t.string  :address
      t.string  :postcode
      t.string  :country

      t.timestamps
    end
  end
end
```

8.2.4 create_ordersproducts.rb

```
class CreateOrdersproducts < ActiveRecord::Migration
  def change
    create_table(:orders_products) do |t|
      t.integer  :order_id
      t.integer  :product_id
    end
  end
end
```

The first three of the tables generated in these migrations relate to the three models well create in the application. The final migration generates the orders_products table to handle the many-to-many relationship between orders and products. The models can be added to the end of the ECOMMERCE module:

```
class User < ActiveRecord::Base
  has_many :orders
  validates :email, uniqueness: true
end

class Order < ActiveRecord::Base
  belongs_to :user
  has_and_belongs_to_many :products

  attr_accessor :stripe_card_token
  attr_accessor :email
  attr_accessor :total

  def save_with_payment
    customer = Stripe::Customer.create(
```

```
      :email => email,
      :card  => stripe_card_token
    )

    charge = Stripe::Charge.create(
      :customer    => customer.id,
      :amount      => total,
      :description => 'order',
      :currency    => 'gbp'
    )
    save!
  end
end

class Product < ActiveRecord::Base
  include Elasticsearch::Model
  include Elasticsearch::Model::Callbacks

  has_and_belongs_to_many :orders
end
```

The first is the User model, this will be used to track both admin users and customers. The application will allow stand alone registration and also registration at the checkout. The **has_many** association is used to link users and orders. We also validate the email address to make sure its unique.

The order model is the most complex of the three. First the **belongs_to** association is added to link it back to the user model. **has_and_belongs_to_many** is then used to link the order model to the product model, this will work via the **orders_products** table. We then have three items defined with **attr_accessor**, these are items we want passed to the model, but dont have columns relating to them in the database. Finally we have a **save_with_payment** method to extract the stripe payment handling out of the Sinatra routes. The email address and stripe card token (well come to how to generate this later) are passed into the Stripe create method to create a customer within Stripe. This customer id can then be used to charge the customers card for the total amount of the order. Finally save! is called to save the order within our application database.

The last model is product, to allow the items in the products table to be searchable within ElasticSearch we include two classes. This will automatically index any product created or updated. It was also create some methods to

allow interaction with the search index. Well then use **has_and_belongs_to_many** to complete the association with the order model.

```
class App < Sinatra::Base
  enable :sessions

  before do
    Stripe.api_key = ENV['STRIPE_API']
  end

  helpers do
    def current_user
      @current_user ||= User.find(session[:user]) if session[:user]
    end

    def admin!
      halt 403 if !current_user || (current_user && !current_user.admin)
    end

    def register(user)
      password_salt = BCrypt::Engine.generate_salt
      password_hash = BCrypt::Engine.hash_secret(user[:password], password_salt)

      User.create( :email => user[:email],
                   :password => password_hash,
                   :salt => password_salt)
    end
  end
end
```

After the models in the ECOMMERCE module the Sinatra App class can be added. The first item here is to enable sessions for Sinatra as well be making use of sessions to store the current user and also the shopping basket items. We then use a before filter to add the Stripe API key, this is stored in an environment variable. This application has three helper methods. The first, current_user, queries the users table in the database for the user id stored in a session variable, if the session variable is set. The second, admin!, enforces only admin users being able to access admin pages. The request is halted and a 403 (access denied) status is returned if the current_user is not set, or if the user is not admin. The final helper method, register, creates a user in the users table. Users can be created either on a registration form or via the checkout, therefore to prevent duplicating code in the two routes we can use this helper. First a password salt is created using bcrypt, the submitted password is then hashed

using that salt. The users email address can then be saved along with the hashed password and salt.

Now that the helpers are defined we can start adding routes to the Sinatra app class.

```
get '/' do
  @products = Product.all
  erb :index
end

get '/register' do
  redirect '/' if current_user
  erb :register
end

post '/register' do
  user = register(params[:user])
  session[:user] = user.id
  redirect '/'
end
```

As always, the first route we define is for the root path. Here we want to list all of the products, the ActiveRecord all method will get all of the products from the database, the index.erb view can then be rendered. In the view the products can simply be looped through with the each method as shown below. The sprintf method allows the product price to be formatted with two decimal places. An add to basket link is then added with the product id that will be used later to submit an ajax request to add the product to a session variable.

```
<ul>
  <% @products.each do |product| %>
    <h2><a href="/products/<%= product.id %>"><%= product.name%></a></h2>
    <p><%= product.description %></p>
    <p><%= sprintf("%.2f", product.price) %></p>
    <a href="#" id="<%= product.id %>" class="add_basket">Add to basket</a>
  <% end %>
</ul>
```

The second route is the registration form, if the user is logged in they will be redirected to the root path, otherwise the register.erb view will be rendered. The form will submit to the third third route handling the registration post request.

The registration form parameters are passed to the register helper, which returns the created user object. We can then set the user id to a session variable before redirecting to the root path.

```
get '/signin' do
  erb :signin
end

post '/signin' do
  user = User.find_by(:email => params[:user][:email])
  if user.password = BCrypt::Engine.hash_secret(params[:user][:password], user.salt)
    session[:user] = user.id
  end
  redirect '/'
end
```

The next two routes handle user authentication on the site. The get request simply renders the login form. For the post request the user is fetched from the database based on the email address submitted. Using this users password salt and the submitted password the hash is checked. If its a match the user id is set to a session variable. No matter the result the request is then redirected to the root path.

```
get '/products' do
  redirect '/'
end

get '/products/new' do
  admin!
  @product = Product.new
  erb :product_new
end

post '/products' do
  admin!
  @product = Product.new(params[:product])
  if @product.save
    redirect "/products/#{@product.id}"
  else
    erb :product_new
  end
end

get '/products/:id' do
```

```
  @product = Product.find(params[:id])
  erb :product
end
```

There are then four roots in the application to handle products. First for the /products path, where we dont want to show anything because the products are listed on the root path. Therefore well just redirect all requests to that root path. We then want to allow the creation of new products within the application via the /products/new path. This is an admin only function so the **admin!** helper is called to enforce this. A new products object is then created before rendering the product_new.erb view containing the form. The post request for the product form also has the **admin!** helper to enforce only admin users can save a product. The product object is created again, this time passing in the params hash submitted with the form. The saving of the project object has been added into an if statement so that if the save is successful the request is redirected to the product page, otherwise the product_new.erb view is rendered again. The final route in this section is the product page. The product id is passed through as a parameter in the path, this can then be used with the find method on the Product model to get a specific product from the database. The product.erb view is then called where the single product is rendered along with the add to basket link as in the index.erb view for the root path.

In the layout.erb a simple form has been added for search doing a post request to /search. The route for this is the next to be added to app.rb.

```
post '/search' do
  @products = Product.search(params[:search_term])
  erb :search
end
```

For the Product model we included a couple of Elasticsearch modules, these add a search method to the model, which we can use to query the Elasticsearch index using the search term sent from the form in layout.erb. The search.erb view is then rendered displaying the results.

The next two routes in app.rb relate to the basket functionality. To make the add to basket links work a small amount of JavaScript has been added to layout.erb.

```
$("a.add_basket").click(function(){
  id = $(this).attr("id")
  $.ajax({
    url: "/basket",
    type: "post",
    data: {product_id: id}
  }).done(function(data){
    $("#basket").html("Basket: " + data.length);
  });
  event.preventDefault();
});
```

Here the click action on the add basket link is hijacked to send an ajax call to the /basket path. The id is taken from the link, which relates to the id of the product. This id is sent as part of the ajax call to the Sinatra route where the id is added to an array. Once the ajax call is complete the items in the returned array are counted and displayed at the top of the page. The `event.preventDefault();` line in the javascript prevents the standard link functionality from working.

The Sinatra route below is the one that handles the ajax call. First the content type is set to JSON as thats the format well be returning. A session variable for the basket is set as an empty array, unless already set using the `||=` operator. The `<<` operator is then used to add the product id, sent in the ajax call, to the basket array. This is finally converted to JSON and returned.

```
post '/basket' do
  content_type :json
  session[:basket] ||= []
  session[:basket] << params[:product_id]
  session[:basket].to_json
end

get '/basket' do
  begin
 @products = Product.find(session[:basket])
rescue
 return erb "There are no items in your basket"
end
  erb :basket
end
```

We then have a basket page, which will display all items in the basket as

a table. The find method for Activerecord allows an array of ids, therefore we can just pass straight in the session variable for the basket. Begin and rescue are used to handle the exception that might be caused if viewing the basket route if the session variable is empty. The basket.erb view can then be rendered looping through each product and listing as a row in a table. If a product has been added to the basket more than once the find method will only return it once, but we can count the amount of times the id for each product was in the session variable using the count method and display this as quantity column in the table. The subtotal for that product can be calculated by multiplying the quantity by the product price.

The final two routes of the application handle the checkout process.

```
get '/checkout' do
  begin
    @products = Product.find(session[:basket])
  rescue
    return erb "There are no items in your basket"
  end
  @order = Order.new
  erb :checkout
end

post '/checkout' do
  if !current_user
    current_user = register(params[:user])
  end

  total = Product.find(session[:basket]).map{ |product| session[:basket].count(product.id.to_s)

  order = params[:order]
  order[:product_ids] = session[:basket]
  order[:email] = current_user.email
  order[:total] = total
  session[:basket] = nil

  @order = current_user.orders.new(order)

  if @order.save_with_payment
    redirect '/'
  else
    erb :checkout
  end
end
```

The first route, the get method, simply renders the checkout form. Much

like the basket it finds all products in the basket using the begin and rescue to handle and exceptions. A new order object is created before rendering the checkout.erb view. Within the checkout checkout form we have a hidden field, order[stripe_card_token], which is used to store the stripe card token generated from the users credit card details. Stripe has a javascript API to generate the token, we will make use of it just as we did in the SaaS chapter. First we need to add the Stripe publishable key to a meta tag in layout.erb, then pull in jquery and the Stripe javascript. A check is done to see if the checkout form is on the page using the length method, before setting the Stripe key from the meta tag to the Stripe class. We can then go ahead and call the setupForm method. This disabled the submit button and as long as the card number is set the processCard method is called. Here the card details are all collected up and passed to Stripes createToken method along with the name of our response handler, handleStripeResponse. In this as long as the response is ok the Stripe token is set to our hidden field and the form submission is completed.

```
<meta name="stripe-key" content="<%= ENV['STRIPE_KEY'] %>">
<script src="//code.jquery.com/jquery-2.1.1.min.js"></script>
<script type="text/javascript" src="https://js.stripe.com/v2/"></script>
<script>
    $(function() {
      if($('form#checkout').length) {
          Stripe.setPublishableKey($('meta[name="stripe-key"]').attr('content'));
        return order.setupForm();
      }
    });

    order = {
      setupForm: function() {
        return $('form#checkout').submit(function() {
          $('button[type=submit]').attr('disabled', true);
          if ($('#card_number').length) {
            order.processCard();
            return false;
          } else {
            return true;
          }
        });
      },
      processCard: function() {
        var card;
        card = {
          number: $('#card_number').val(),
```

```
          cvc: $('#card_code').val(),
          expMonth: $('#card_month').val(),
          expYear: $('#card_year').val()
        };
        return Stripe.createToken(card, order.handleStripeResponse);
      },
      handleStripeResponse: function(status, response) {
        if (status === 200) {
          $('#stripe_card_token').val(response.id);
          return $('form#checkout')[0].submit();
        } else {
          $('#stripe_error').text(response.error.message);
          return $('input[type=submit]').attr('disabled', false);
        }
      }
    };
</script>
```

The post request for the checkout is pretty complex as it needs to calculate the total to charge, charge this via Stripe, add a Order object to the database, and attach this order with the products. First if there is no **current_user** defined the register helper is called with user details from the checkout form to register a new user. The total price for the order is fetched by finding all products in the basket session variable. The map method is used to multiply the price of each product with the quantity it appears in the session variable. The inject method allows us to add each price in the array together, before the total is multiplied by 100 to give pence rather than pounds (or cents rather than dollars) as this is the format Stripe requires. The order data from the form is stored in a variable, order, to this we can then add some manual items. **product_ids**, is the array of ids in the basket session variable, this generates the association between the order and the products. email is the email address of the current user whether they were already logged in or just registered via the checkout form. total is the price in pence (or cents) we generated earlier from all products in the basket. The basket session variable is then set to nil to empty the basket. Now we have all the data needed as part of the order variable we can create a new order object, using the orders method for **current_user** the order is associated with the user who is logged or just registered. The **save_with_payment** we defined in the order method can now be called, if successful the user is redirected back to the root path, otherwise the order.erb

view is reloaded.

Now that we have a complete application you can run it, for this you will need a config.ru file:

```
require './app'
run ECOMMERCE::App
```

A user can then be registered and to make the admin user well use the **irb** command to ender the ruby console, run this from the application route then enter the following two commands.

```
require './app.rb'
ECOMMERCE::User.find(1).update(admin: true)
```

Firstly the application is added, then we find the first user and update them to set admin as true.

8.3 Summary

In this chapter we have covered quite a lot. The Stripe integration should be quite familiar from the SaaS chapter however this time we did a single charge rather than a subscription. Elasticsearch was used to index products when theyre created and search for them via a simple Activerecord method. Finally we used an ajax request to add items to the basket then calculated the order and the total price from the products stored in the basket.