Webapps

Rack

Rack is a library (gem) which provides a simplified interface between a ruby application and a server. Include rack in a script using:

*require ‘rack’*  & *require ‘rack/server’*

Then start a server using:

Rack::Server.start(app: <app-name>)

Rack apps can then be programmed using an array with a response to a call:

class HelloWorldApp

def self.call(env)

[<statuscode>, {<key-values>}, [<response-body-html>]]

end

end

Rack apps can be run using the commandline with ‘rackup config.ru’. The config file needs to require the app.rb file and run the webapp class.

In Rack apps, the params are put into the ‘env’ hash variable as key value pairs, these can be printed out or used in the body of the response message by calling env[<key>]. The params can be encapsulated into a request object using ‘request = Rack::Request.new(env)’, the request object then has many methods which can be performed on the params. See methods through documentation or request.methods.sort.

Responses can be iterativley made by a Rack::Response object. Create a new reponse object with a body using ‘response = Rack::Response.new(response\_string)’ then use the reponse object methods to add details to the response and finally use .finish to complete the response.

<https://ieftimov.com/post/writing-rack-middleware/>

<https://www.rubyguides.com/2018/09/rack-middleware/>

The real power of rack comes with its ability to chain applications together in a stack using middleware. Using ‘use App’ in the config.ru before running the main webapp, will allow for the middleware apps to be run before the main app logging, modifying, caching etc the env data. Middleware apps are stacked in the order which they are used, so for example:

use Logger

use Cacher

run HelloWorldApp

Would be stacked like:

[logger[cacher[helloworldapp]]]

With each middleware app being added into eachother via the initialize env argument, the request and response are then filtered through each program allowing it to do what it is designed for, ie authentithcation or logging.

Rack::Test

Rack apps can be tested in RSpec using by requiring the ‘rack/test’ gem and including ‘include Rack::Test::Methods’ in the body of the test.

The relevant app must then be loaded using:

def app

HelloWorldApp

end

Tests are then made similarly by getting a webpage then expecting responses:

it 'says hello' do

get '/'

expect(last\_response.body).to eq('Hello World')

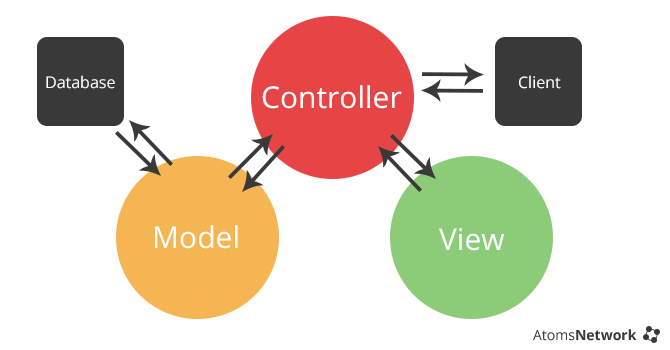
expect(last\_response.status).to eq(200)

end

MVC

When designing a web app, the different responsibilities for each thing the app does get split into three sections to allow for better maintainability:

* **Model**: The part of the software that handles application logic, such as what data is saved, and how that data should react to certain commands, etc. The model contact databases and formats the data in a way the view can handle without extensive logic.
* **View**: The part of the software that handles what is displayed to the user. This part of the program takes info from the Model part (and sometimes the Controller part) to display the information. In Sinatra Ruby uses the erb engine to format data passed in from the controller/model into html.
* **Controller**: The part of the software that handles user input. This part of the program waits for user input, and routes information to the correct models and view, then back to the controller for under input again. Depending on the type of input the controller may have to convert input data and then informs the other two parts to make appropriate changes to the state of the program.



Note that there aren't necessarily only one of each. For instance, if you have multiple files open, the spreadsheet program might have multiple models, or just one model that covers all of them, depending on the programming style.

This is called the seperation of concerns since the concerns for each part: model, view, and controller are seperated and do not interact. ie, the controller and view should not be doing any logic processing. As a rule of thumb, logic should be 'pushed down the stack' wherever possible. 'Views' are for lightweight logical lifting, and controllers are for middleweight stuff. Views should not be clutter with lots of complicated logic, generally limit it to only

* light conditionals - if / else
* light enumerables - each

Sinatra

Sinatra is libaray based off Rack which is used to create an MVC webapp with Ruby in a short period of time. Applications follow the bellow structure, where the sinatra environment can be set up to define the root and view folders:

set :root, File.join(File.dirname(\_\_FILE\_\_), '..')

set :views, Proc.new { File.join(root, "views") }

set :public\_folder, Proc.new { File.join(root, "../public") }

A screenshot of a cell phone

Description automatically generated

To use sinatra include ‘sinatra’ in the gem files and create a webapp controller subclass of Sinatra::Base. Then add to the config.ru file ‘run <classname>’. Routes are set up using the ‘get '/' do’ syntax which refer to an erb file containing html view template.

<http://sinatrarb.com/intro.html>

.erb files are stored in ./app/views and allow for simple html to be written and called in the ruby script. To call use ‘erb :<filename>’.

<https://www.stuartellis.name/articles/erb/>

<https://ruby-doc.org/stdlib-2.2.3/libdoc/erb/rdoc/ERB.html>

ruby code and instance variables can be called from an erb by encasing them in erb tags => <%= <code> %>

Query string parameters can be called into the controller code much like a hash, called simply by ‘params[:key]’.

Since HTML forms cannot submit DELETE and PUT requests (god knows why) these RESTful routes can be desgined into Sinatra providing ‘use RACK::MethodOverride’ is used in the webapp class. Forms can then be submitted as a POST request including the hidden value line:

‘<input type='hidden' name='\_method' value='<put><delete>'>’

Then to keep RESTful the action must route to the ID of the item being modified/deleted so the item id must be interpolated into the action route and to allow Sinatra to receive any id and pass it through as a param[:id], create the POST/DELETE route using the following syntax:

*delete '/bookmarks/:id' do*

*#some stuff*

*end*

Shotgun

Shotgun is an app used to run Sinatra apps to remove the need of stopping and starting the app each time the code is modified. To run an app use the following on the command line:

shotgun config.ru

As Shotgun restarts the server each code change, to keep sessions consistent during resets add the following code to the app:

configure(:development) { set :session\_secret, "something" }

The default port for shotgun is 9393, however the port can be selected using the following argument ‘-v <port\_num>’.

Rackup

Rackup is similar to shotgun, running apps on port 9292. More reliable, than shotgun so use this.

rackup config.ru

Capybara

The Capybara Gem can be used with RSpec to simulate user input and test ruby web applications features.

First Capybara must be set up in the RSpec helper file using the following:

*require 'capybara'* 🡨 require gems

*require 'capybara/rspec’*

*ENV['RACK\_ENV'] = 'test'*  🡨 set rack env to test

For Capybara to work it must be assigned an app. This can be done in either the RSpec helper file or each test file, the standard ‘require ‘app’’ or using the full app filepath if the file is not in the lib directory:, or in each test file.

*require File.join(File.dirname(\_\_FILE\_\_), '../lib/app.rb')* 🡨 get app file using helper path

Then assign Capybara the class to test the features of in either the helper file or each test:

*Capybara.app = Battle*  🡨 set capybara to work run Battle app

When writing tests put the files in the project ‘spec/features’ folder. For a capybara test you must add a suffix to the describe section or use the alias ‘feature’:

*describe "signin process", type: :feature do* || *feature “signin process” do*

Other capybara alias’ are:

* background -> before
* scenario -> it
* given -> let

Development Environments

Web projects normally have three environments:

* Development - development mocks used, i.e. development cards which still connect to centre. Full debugging
* Test - RSpec i.e. mock credit cards
* Production - Real life program, i.e. credit card charged for real. Minimal debugging

Databases must be considered by environment, since the same database shouldn’t be used for development and production else real data might get changed. Test databases should be empty and only test for items explicitly added. Therefore, before connecting to a database the environment env varible should be checked, and since systems should be fail safe it should always default to the development environment database.

Screen Readers

3.5% of webtraffic uses a screen reader (reads out text on a page), so any webapp created should be made compatibile to ensure it gains its full potential of traffic and users don’t get enraged.

To make HTML compatbile use semantic HTML markup ,which requires:

* The correct tags for each section, so screenreaders can work out what to read in what order.
* Requries separation of content and presentation using CSS

Using a semantic HTML cheatsheet will help with this:

<https://learn-the-web.algonquindesign.ca/topics/html-semantics-cheat-sheet/>