**Rails (a Ruby framework)**

1. **Theory**
   1. Rails works primarily through the URL to perform tasks, process overview:



* + 1. The router determines which controller and action to send requests off to
    2. The controller determines what to do with requests
    3. The view determines what to render to the screen
    4. The model determines what data is available

1. **Rails Basics**
   1. “rails” and “rake” commands are basically the same thing
   2. Terminal: rails s //starts your rails server
      1. Option: -p 5000 //starts your rails server on port 5000
   3. Log Files are available in src/log/development.log
   4. Developing in Rails
      1. Often times, it’s smart to use ‘error driven development’ to help point you in the direction of ‘what to do next’
      2. Example, if I want to create a new page to show all the people in my Person table:
         1. In my browser, type the url path of my desired query, perhaps something like: localhost:3000/people
         2. I should get an error related to a missing route, so make the route
         3. Refresh, now I should get an error for a missing controller, so make the controller
         4. Refresh, I should get an error for a missing view, so make the view
         5. And so on..
   5. Using the rails console:
      1. To get into the rails console, in Terminal: rails console
      2. ‘Hirb’ is a gem (add it to the Gemfile) that makes your rails console tables suck less
         1. Once bundled, use ‘hirb.enable’ in the rails console to turn it on
      3. Viewing
         1. Person.all - shows you all entries in the Person table
         2. Person.last - shows you the last entry in the Person table
      4. Creating
         1. Person.create(name: “Andy”, age:36, mood: “happy”)
            1. Adds an entry to the Person table with the 3 attributes passed in
      5. Destroying
         1. Person.destroy\_all - removes all entries in the Person table
      6. Troubleshooting
         1. ‘.persisted?’
         2. ‘.valid?’
         3. ‘.errors’
            1. ‘.errors.full\_messages’
         4. You can create classes to help troubleshoot your entries
            1. Ex: gabe = People.new(name:”Gabe”)
            2. Now you can troubleshoot ‘gabe’

‘gabe.valid?’

‘gabe.errors.full\_messages’

* + - * 1. And change gabe’s values:

‘gabe.age = 25’

‘gabe.mood= “excited”’

* + - * 1. And save gabe to the model once errors have been corrected

‘gabe.save’

1. **Database Basics**
   1. In the background, rails uses databases like SQL, postgreSQL, etc. to do the database functions
   2. Your primary interface with the database is Rails’ “Active Records”
   3. Active Records does the hard database-language stuff for you so you can interface with Ruby
   4. Lifecycle of a Database
      1. Terminal: rails db:drop //drops your database, start fresh
      2. Terminal: rails db:create //creates your database, only run if no db exists
      3. Terminal: rails db:migrate //turns rails migration files into models + schema
      4. Terminal: rails db:seed //populates your models with data from ‘seed.rb’
      5. You can chain these together: rails db:drop db:create db:migrate db:seed
2. **Models**
   1. Model Files - where your data is accessed
      1. Located in: src/app/models/MODEL\_NAME.rb
      2. Example of a model file:



* + 1. The model files should always open with: class ModelName < ApplicationRecord
    2. Models should be edited to control the behavior of your data set
       1. Associations - see Active Record documentation for details.
          1. Does the model contain a column whose data belongs to information on another table?
          2. Does it need to access data in other models?
       2. (User friendly) Errors
          1. Use keyword ‘validates’ with ‘, presence: true’
    3. Any model can be viewed as JSON in the browser (if there is a proper route and controller) by navigating to localhost:3000/path/model\_name.json
  1. Migration Files - what creates your models
     1. Located in: src/db/migrate/MIGRATION\_NAME.rb
     2. Example of a migration file:



* + 1. To create migration files from scratch, in Terminal do this:

SYNTAX: rails g model MODEL\_NAME ATTRIBUTE1:TYPE ATTRIBUTE2:TYPE

Ex: rails g model Person name:string age:integer email

* + - 1. Notes -
         1. The above example creates a model named ‘Person’ with 3 rows: name (string), age (integer), email (string)
         2. ‘g’ stands for generate, can use ‘generate’ in place of g
         3. ‘string’ is the default data type for any attribute, doesn’t need to be specified (ex: email above)
         4. The models name should always be capitol and singular
    1. Migrations should be edited to control the rules of your data set
       1. Rules:
          1. Null: Is this data able to be blank and still process?

add ‘, null: false’ to any line you want to be required

* + - * 1. Default: If no value is provided, should we auto-assign a value?

add ‘, default: “meh”’ to any line you want the string ‘meh’ to be the default value if no value is passed

* 1. Seed Files - your default data set
     1. Located in: src/db/seed.rb
     2. Example of a seed file:



* + 1. Seeds should be edited to populate your data set with default info
       1. This is ideal for dev/test environments
    2. It’s wise to start your seed file with “MODEL\_NAME.destroy\_all” to prevent you from having duplicate records
    3. Faker is a great gem to generate random things (like names, colors, numbers, etc.)
    4. It’s wise to end your seed file with a message that confirms how much data was generated by the seed
  1. Database file - located at src/db/development ???

1. **Views**
   1. Overview
      1. Your ‘web pages’ live in src/app/views
      2. File extension .html.erb allows you to inject ruby into html using:
         1. <% %> - for things not to be rendered
         2. <%= %> - for things to be rendered
      3. Your boiler plate lives in src/app/views/layouts
         1. <%= yield %> is where the other html.erb files get injected
         2. This is where you need to add dependencies and things that need to be on every page (like a nav bar)
   2. Views are always placed in a folder corresponding to the controllers name
   3. CSS Stylesheets - located at src/app/assets/stylesheets
   4. JavaScript files - located at src/app/assets/javascripts
2. **Controllers**
   1. Controllers are always named in plural
   2. For every controller, there should be a folder in views with the appropriate name
3. **Routes**
   1. Overview
      1. The Route file takes incoming HTML requests and directs them to the correct controller
      2. To view all routes in your app, start the rails server and navigate to: localhost:3000/rails/info/routes
   2. Resource or Resources?
      1. If there is only one in your app (like a profile, or calendar), use singular resource
      2. If there are many (like a library of shows or movies), use plural resources
4. **Starting a new project from Scratch**
   1. Initializing the project
      1. Terminal: rails new projectNameHere
         1. Option: --database=putDatabaseTypeHere
         2. Option: --skip-coffee
         3. Option: -T #Means no tests, I’ll bring my own in later
         4. If you have database version issues, modify the version in the Gemfile & re-bundle
            1. ‘~>1.3’ means ‘give me the highest minor version of 1.3 (drunk rocket)
   2. Decision point, to Scaffold or not to Scaffold?
      1. Scaffold is *“an entire resource, from model and migration to controller and views, along with a full test suite. The resource is ready to use as a starting point for your RESTful, resource-oriented application. (More documentation in your terminal window)”*
      2. Scaffold is basically a “set everything up for me” tool used to quickly deploy a front/backend; it is ideal for testing things quickly, but not ideal for production level code
      3. Terminal: rails g scaffold
         1. g stands for ‘generate’
         2. This is how to scaffold with default options
      4. Terminal: rails g scaffold task description:string --javascript-engine=js
         1. This generates a scaffold for the model named ‘task’
5. **React in Rails**
   1. React files live in src/app/javascripts (note, NOT src/app/assets/javascripts)
   2. Rails does not break a page when JavaScript code is broken, instead you need to check the JavaScript console in the browser (command option J)
   3. To add a new react file/page/component:
      1. Create a new folder in src/app/javascripts with the name of your file (example, /calendar)
      2. Create a ‘components’ folder in your new folder
         1. Your components to your react file go here
      3. Create a react file (ex: calendar.jsx) in your new folder
         1. This is where your main React.Component class will live
         2. Make sure you export it!
      4. Register your components in jacascript/packs
         1. ReactOnRails.register( {ReactFileName, ReactFileName2} )
         2. Make sure your pack imports ReactOnRails and ReactFileName
      5. Link your component into the html.erb file with <% react\_component ‘component name’ %>

**UNREFINED NOTES START HERE**

1. **Roster Project**
   * 1. Look at the error... It’s telling you there is no route
        1. ‘routes.rb’ is the file to fix
     2. Now look at the error.. It’s telling you the :controller key is missing... Look at your chart
        1. ‘routes.rb’: get ‘/people’ => ‘people#index’
           1. ‘index’ shows collections of things, opposed to ‘show’ which is a single thing
     3. Now look at your error... It’s telling you there is no PeopleController
        1. app-->controllers--> create a new file people\_controller.rb
        2. In the file, declare: class PeopleController < ApplicationController

end

* + 1. Now look at your error... You have no index action on your controller
       1. Add ‘def index end’ to your people\_controller.rb
    2. Now look at your error... missing a template for the request...
       1. Create an ‘index.html.erb’ file in views --> (new folder) people
       2. What you put in this file is what you get back on your /people page
          1. <% %> is the syntax to ‘start ruby’ in a .erb file
          2. <%= %> is the syntax to see ruby code on the page
    3. Print the table to the screen...
       1. In index.html.erb:

<table> <thead> <tr>

<th>Name</th>

<th>Age</th>

<th>Mood</th>

</tr> </thead>

<tbody>

<% people.each do |person| %>

<tr>

<td> <%= person.name %> </td>

<td> <%= person.age %> </td>

<td> <%= person.mood %> </td>

</tr> </tbody> </table>

* + - 1. In people.controller.rb, inside index...
         1. @people = Person.all
         2. This has the controller set the variables, so the view reads them
         3. Make sure @people in controller matches @people in index...
  1. Display a single entry:
     1. Add new route for /people/:id
     2. Go through the process in 3.f all over again...
     3. In show.html.erb...
        1. <%= params[“id”] %> will give you the value of the id
           1. Also works as <%= params[:id] %>
     4. Maybe we call (in people controller) in def show/end a single person @person = Person.find(params[:id])
     5. Now you can do shit like @person.name
  2. New Shit
     1. Rails? built in method: link\_to(what\_to\_say, where\_to\_go)
        1. Ex: <%= link\_to(person.name, “/people/#{person.id}”) %>
        2. Ex2: <%= link\_to “All People”, “/people” %>
     2. In routes.rb, “get ‘/people’=> ‘people#index’ and “get ‘/people’=>’people#show’”is a weak way of doing ‘resources :people, only: [:index, :show]
        1. If the route has a named path, you don’t have to type out the url path in link\_to
           1. Ex: <%= link\_to person.name, person %>
     3. Sorting Data (in people controller)
        1. instead of Person.all, do Person.order(name: :asc)
           1. ‘:desc’ is descending
     4. Different routes available to you:
        1. Create - create a new thing (form submission of new)
        2. New - return the HTML form to create a new thing
        3. Index - display a list of all things
        4. Show - display a single thing
        5. Edit - return an HTML for editing a thing
        6. Update - update a specific thing (form submission of edit)
        7. Destroy - delete a specific thing

1. **Forms**
   1. Theory
      1. Originally, forms are designed to make HTML requests, like get, put, post, etc.
      2. Unlike in JS, forms are not meant to be event.preventDefault()’d...
      3. Forms take two attributes, action and method.
         1. Action: what to submit to (i.e. outside API)
         2. Method: what type of submission (get, pull, etc.)
            1. Get request is the default if a method is not supplied
      4. Use buttons to submit forms
      5. Use inputs to change the parameters of the form submission
         1. ‘step’ attribute determines the size of changes (ex 1 means only integers)
         2. ‘min’ attribute sets a minimum value (ex 0 means no negative numbers)
         3. ‘value’ attribute sets the default value of the input
         4. ‘name’ attribute causes the form submission to change the url when a form is submitted
         5. ‘type=”hidden”’ attribute makes an input submit without prompt (useful when you want to pull information from the current URL for a new form submission)
   2. Get Request Implementation
      1. In the html.erb file:

<form>

<label for=”min\_age”>Min Age</label>

<input name=”min\_age” type=”number” step=”1” min=”0” value=”0” />

<label for=”max\_age”>Max Age</label>

<input name=”max\_age” type=”number” step=”1” min=”0” value=”0” />

<input name=”mood” type=”hidden” value=”<% params[:mood] %>” />

<button type=”submit”>Search</button>

</form>

* + 1. In the controller file (min\_age: bad way (vuln to SQL injection), max\_age: good way):

@people=Person.all

if params[:min\_age].present?

@people = @people.where(“age >= #{params[:min\_age]}”)

end

if params[:max\_age].present?

@people = @people.where(“age <= ?”, params[:max\_age])

end

* + 1. You never want to do string interpolation with anything that ends up in a SQL database, because SQL injection
       1. Instead, escape the string (see 4.b.ii for example of good/bad)
    2. To change the default parameters to equal the min/max age of entries in the models, change the html in 4.b.i input value to equal:

<%= params[:min\_age].to\_i %>

* 1. Post Request Implementation
     1. In the html.erb file (bad way):

<form action=”/people” method=”post”>

<button type=”submit”> Add Person </button>

</form>

* + - 1. Better way (plus more stuff):

<%= form\_tag “/people”, method: :post do %>

#You can do regular HTML in here, as the line above gives you the authenticity token

<div>

<%= label\_tag “Name” %>

<%= text\_field\_tag “name” %>

</div>

<div>

<%= label\_tag “Age” %>

<%= number\_field\_tag “age” %>

</div>

<div>

<%= label\_tag “Mood” %>

<%= text\_field\_tag “mood” %>

</div>

<%= submit\_tag %>

<% end %>

* + - 1. If you don’t do the better way, you will need to configure the authenticity token
    1. Add the post to routes.rb (bad way):

post ‘people’ => ‘people#create’

* + - 1. Better way: add :create to the resources line (from 3.h.2)
    1. In the controller.rb

def create

#read the data from the post request

#use it to add a person to the database

Person = person.new(

#params from forms come from the ‘name’ attribute in HTML inputs

name: params[:name]

age: params[:age]

mood: params[:mood]

)

#send user back to new index page

if person.save

redirect\_to people\_path, notice: “Just added a new person”

else

redirect\_to people\_path, notice: “Inavlid inputs”

#can also do people.errors.full\_messages.to\_sentance instead of the string “invalid inputs”

end

* + 1. In application.html.erb
       1. Add <%= notice %> and <%= alert %> to display notices/alerts in 4.c.3
  1. A form that will create or update based on rails using logic...
     1. <% form\_with(model: @person) do |form| %>
  2. View Partials are ‘snippets’ of HTML code that you can reuse in different places
     1. Files start with an underscore, ex: \_form.html.erb, and go in the same folder as your other html.erb files
     2. Call the partial in the views using <%= render “partial\_name”, person: @person %>

1. Controllers can render JSON pretty easy... ex:

def api

render json: Person.all

end

1. Models can dictate what JSON responses will look like (i.e. doesn’t have to be 1:1 based on the information in the tables), ex:

def as\_json(options={})

{

name: name,

age: age,

mood: mood

}

* 1. The above would strip some of the things ruby gives, like date created, date modified, etc.
  2. You can choose not to do this and just get a 1:1 from the database as a JSON response

1. Lessons Learned
   1. Keep your models fat, your controllers skinny, and your views anorexic...
      1. Did you use .where in your controller? Turn it into a scope in the model
      2. Did you do a computation in a view for a table output? Turn it into a method in the model
   2. Model Validation - keep bad data out of your database and/or app!
      1. Ideally, you want to set rules when you create/migrate your models
      2. It’s okay to remove database logic (less strict), but adding logic would likely cause errors (more strict) with currently stored data
      3. If it’s too late, you can add model validation statements into your models .rb file to prevent further bad data from being input
      4. It’s okay to duplicate controls in database logic & app-level model validation
   3. Adding to a model (with special syntax) from command line (not rails console): “rails g migration AddNAMETHETHINGToTABLENAME rowName:dataType
   4. Form with Model allows the form to act as both a creator and editor
      1. Ex: Does the entry exist? Behave like an update
      2. Ex: Does the entry not exist? Behave like a create
   5. When generating submodels, add something like ‘parentModel:belongs\_to’ to the ‘rails g model’ command along with the other attributes
      1. This creates a ‘foreign\_key’ relation in the child model (also migration)
      2. A foreign key is a reference to another tables primary key
      3. Instead of “belongs\_to” you can also do “has\_many”, but you probably want to add “dependent: :destroy”
      4. Can also do “has\_one”