Rails Routing + MVC

Objectives
Student should be able to properly route a request for a resource
Student should be able to setup CRUD routing for a resource
Student should be able to explain the high level connection between the MVC pattern and RESTful routing for resources

Recap of Rails Philosophy

Separating Concerns

In writing a large application it is important to establish something known as **Separation of Concerns**, *writing modular code that focuses on one aspect within the application.* The benefit of this is similar to idea of **compartementalization** with respect to a production line, which allows for *more rapid development* by being able to **divide and conquer** the construction of a product. Comparments can focus on one task and optmize functional concerns far outside the scope of other compartments, but still work together to acheive the same product. Ultimately it reduces the headache of debugging and controlling a large application that can ultimately grow to a level of complexity that no one person could ever fully comprehend (nor want or need to).

Organizational Principles

In order to manage the development of emerging aspects within a project it is important to construct a guideline that will shape how things are separated, a **design pattern**, which everyone can use to maintain **consistent** organization of different aspects. This is a *conventional* choice that helps to understandably scale a project. Part of the role of a developer is to become familiar with using design patterns, but this takes time (and trust), as different patterns emphaize an array of qualities: scalability, modularity, security, performance, et cetera.

Conventions To Focus On

In Rails we see one of the most popular patterns of Web Application Design that has evolved over the years, **Model-View-Controller**. The **MVC** patterns seeks to separate components into **Data Concerns**, **Presentation Concerns**, and **Request and Response (or Action) Concerns** respectiveley:

MVC

Component	Type of Concern
Model	Data Concerns
View	Presentation Concerns
Controller	Request and Response (or Action) Concerns

Lesson Code Along Road Map

We'll be building an application to handle management of our favorite planes in Rails to demonstrate routing, controllers, views, and models (a little).

- Talk and Use CRUD and RESTful routing conventions
- Create a new **app folder** for our routing app

- Setup `index`
- write an **index route** for planes
- make a **controller** for planes
    - make an **index method**
- make an **index view**
- generate a **plane model**
- Setup `new`
- make a **new route** that presents a form for new planes
- make a **new method** in the **PlanesController**
- make a **new view**
- Setup `create`
- make a **create route** for submitting new planes
- make a **create method** for saving new planes and redirecting

CRUD and REST

Lesson Vocab:

Term	**Definition**
CRUD	**C**reate, **R**ead, **U**pdate, **D**elete
REST	**RE**presentational **S**tate **T**ransfer

CRUD

So what's CRUD? Think of CRUD as a set of the minimum and most common actions needed for interacting with data in an application. You shouldn't need to be convinced of this--you all saw it during project #1.

You also see CRUD in action all over the web. For example, on Facebook, each user must be able to: *Create* their facebook profile, *Read* it, Update* it, and if they're fed up with the constant interface changes, *Delete* it.

Additionally, we usually associate **CRUD** with the following **HTTP** verbs (AKA methods):

CRUD Operation	**HTTP Method**	**Example**
CREATE	POST	`POST "/puppies?name=spot"` (create a puppy named spot)
READ	GET	`GET "/puppies"` (Shows all puppies)
UPDATE	PUT	`PUT "/puppies/1?name=Lassie"` (change puppy number 1 to have name Lassie)
DELETE	DELETE	`DELETE "/puppies/1"` (destroy the first puppy, yikes!!!!)

REST - Representational State Transfer

Just like in Node.js, we'll be using RESTful routing practices in Rails. To be RESTful, something simply has to be named or written in a **semantic** (*relating to meaning in language or logic*) way. So RESTful routing is just a way of writing our `routes` so that the *purpose* of each route is clear to another developer (or even a user) just by looking at the route's structure.

Hook

The router is the doorman of your application. When an HTTP request arrives from the user's browser, it needs to know which controller action (method) should be run. Should we display the "new user" webpage? Should we edit an existing user with whatever data got sent along?

The **Router** is basically just a matching service. It looks at the HTTP verb (GET, POST, PUT, DELETE) and the URL that it being requested and matches it with the appropriate controller action to run. It's a pretty simple function but an essential one. If it can't find a route that matches the request, your application will throw an error.

We've talked before about the 7 basic CRUD routes. Using the example of a "photo" model, here's what those 7 CRUD routes end up being when paired with RESTful routing practices:

HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Taken from "Rails Routing from the Outside In"

Great Routing Overview - The Odin Project

Distinguish between a route and a path

What are the seven RESTful routes for a resource?

Which RESTful routes share the same URL but use different verbs?

Rake routes for an app with resource routing and walk through it and diagram on the board.

- a **path** is a sequence of segments (conceptually similar to directories, though not necessarily representing them) separated by a forward slash ("/").
- a **route** is a combination of an HTTP request and a path

Code Along - Part 1: On the Runway - INDEX

Setup With Rails New

Let's **familiarize ourselves with the initial setup** of a new application so we can start building our *planes* application.

```
$ rails new route_app
$ cd route_app
$ rails s
```

Now our app is up and running, localhost:3000. At our `root` route ('/') you should see a "Welcome aboard message". That's because we have yet to create a **controller** and **views** that we can set as our **root**.

A Look at Routes

In *Rails*, you write routes into a special file: `config/routes.rb`. The code you put in this file essentially defines how to connect **requests** to **controllers**.

Go to `config/routes.rb` and inside the routes block erase all the commented text. It should now look exactly as follows:

```
RouteApp::Application.routes.draw do

end
```

Now we can define all our routes.

> NOTE: A **Controller** is just a class that takes care of rendering views and managing data resources. It does this using methods **you'll** define.

Essentially, `routes.rb` just tells your app how to connect *HTTP* requests to a specific **Controller**. So, let's build our first route.

The nature of any route goes as follows:

```
`request_type '/for/some/path/goes', to: "controller#method"`
```

e.g. if we had a `PuppiesController` that had a `index` method we could say

```
`get "/puppies", to: "puppies#index"`
```

Rails already assumes that `"puppies"` in the `puppies#index` part of the route refers to a class called `PuppiesController`. It also assume it can find that class in the `app/controllers` folder. As a result, there's no need to write `Controller` to designate the class `PuppiesController` or to tell Rails where to look for this file in your application.

> Rails Vocab: Convention over Configuration (COC). This idea—that we can accomplish more by following a set of agreed upon conventions than by explicitly configuring our applications—is part of the core design philosophy behind Rails, REST, and much of the modern internet. As we learn Rails, try to keep a close eye on the conventions we point out, as they'll help you be much more efficient when you build your applications.

- Using the above routing pattern, let's create our first route:

`/config/routes.rb`

```
RouteApp::Application.routes.draw do
  get '/', to: "planes#index"
end
```

Let's go back to our web browser and reload the page. What happens?

ERROR!

Don't be afraid! Try to remember that error messages are our friends. And you should already see that Rails makes error messages a bit less intimidating.

So why did we get an error?

Turns out that once we define a route, we also have to define a `planes` controller with an `index` method to match.

A Bit More Syntactic Sugar: `root to`

Because just about every application has a `root` route, rails has a special `root to` shortcut available for our use. Let's use it in `routes.rb`

`/config/routes.rb` `ruby RouteApp::Application.routes.draw do root to: 'planes#index' end`

Making a Planes Controller

We want to create a *planes_controller*. **NOTE**: Controller names are always plural and files should always be `snake_case`.

$ subl app/controllers/planes_controller.rb

Let's begin with the following

```
  class PlanesController < ApplicationController
    def index
      render text: "Hello, pilots."
    end
  end
```

We have defined the `PlanesController` *class*, given it the method `#index` , and told the `#index` to render a *text* response `'Hello, pilots.'`

> Note: We've also indicated that `PlanesController` inherits from `ApplicationController` , which looks like the following:
>
> class ApplicationController < ActionController::Base # Prevent CSRF attacks by raising an exception. # For APIs, you may want to use :null*session instead. protect*from_forgery with: :exception end
>
> Indeed, `ApplicationController` also inherits from `ActionController::Base` , which is just the main *action* handling class. Actions it might handle are requests, responses, rendering views, etc. The `ApplicationController` helps define the setup/configuration of all other controllers and has methods defined accross the entire application.

If we go to localhost:3000/ we get the greeting.

A View For Planes

Let's seperate our rendered greeting into a view called `index.html.erb` , which by default `ActionController` will look for in a `app/views/planes/` folder. Create the file below

`app/views/planes/index.html.erb`

Hello, pilots!

and make the following changes to `PlanesController` .

`app/controllers/planes_controller.rb`

class PlanesController < ApplicationController

```
def index
  # Note it used to say
  # render text: 'Hello, pilots'
  render :index
end
```

end

A Model Plane

A model is just a representation of a SQL table in our database, and the communication between the two is handled by rails via `ActiveRecord` , which has a list of prestored SQL commands to facilitate communication.

In terminal, we create our plane model using a rails generator as follows,

```
  $ rails g model Plane name kind description
```

which just creates the instructions in our app to tell SQL to create our model.

NOTE- Our model names should be singular and the Rails Docs may have them capitalized. Also, if we don't specifiy a data type it will default to String.

To actually create this table data in our SQL database we do a migration. To migrate our database we use `rake` as follows:

```
  $ rake db:migrate
```

Now our application will have access to a model called `Plane` that will be persitent. Don't worry too much about the `rake` command that was just used as previous students have had the same frustration with it.

Rake

Rake is a software task management tool. It stands for "Ruby Make" and you can see them with `rake --tasks`.

Making your first Model

NOTE: DON'T SKIP THIS STEP

We go straight into terminal to enter *rails console*.

```
$ rails console
```

The command above enters the rails console to play with your application.

To create our first plane model in our database we use our reference the `Plane` class and call the `Plane#create` method to write our plane to our database.

> Plane.create({name: "x-wing", kind: "unknown", description: "top secret"}) => #

This will avoid issues later with `index` trying to render planes that aren't there.

Back to Routes - NEW/CREATE

Motive
We've seen a little of each part of the MVC framework, and now we cycle back through over and over as we develop an understanding of the work flow.

A new route for planes

We have one plane in our database that we created from `rails c`. Let's create a form so that we can create new ones with our app.

To be able to make planes we must create a route for them. The *RESTful* convention would be to make a form available at `/planes/new`.

Let's add this route.

`/config/routes.rb`

```
RouteApp::Application.routes.draw do
  root to: 'planes#index'

  # just to be RESTful
  get '/planes', to: 'planes#index'

  # it's a `get` because
  # someone is requesting
  #   a page with a form
  get '/planes/new', to: 'planes#new'

end
```

A new method for planes

The request for `/planes/new` will search for a `planes#new`, so we must create a method to handle this request. This will render the `new.html.erb` in the `app/views/planes` folder.

`app/controllers/planes_controller.rb`

```
PlanesController < ApplicationController

  ...

  def new
    render :new
  end

end
```

A new view for planes

Let's create the `app/views/planes/new.html.erb` with a form that the user can use to sumbit new planes to the application. Note: the action is `/planes` because it's the collection we are submiting to, and the method is `post` because we want to create.

`app/views/planes/new.html.erb`

```
<form action="/planes" method="post">
  <input type="text" name="plane[name]">
  <input type="text" name="plane[kind]">
  <textarea name="plane[description]"></textarea>

  <button> Save Plane </button>
</form>
```

Note: The form and method we just created motivates our next `route`, which will be

`post "/planes", to: "planes#create"`

Creating another route

Our submission of the `plane` form in `new.html.erb` isn't being routed at the moment let's change that

`/config/routes.rb`

```
  RouteApp::Application.routes.draw do
    root to: 'planes#index'

    get '/planes', to: 'planes#index'

    get '/planes/new', to: 'planes#new'

    # handle the submitted form
    post '/planes', to: 'planes#create'

  end
```

Creating another method

This leads to the most complicated method yet to be talked about. For now we will just make it redirect to the `"/planes"` route.

`app/controllers/planes_controller.rb`

PlanesController < ApplicationController

```
...

def create
  redirect_to "/planes"
end
```

end

> Someone should now be able to submit a form to our site, right???

Our first form

`app/views/planes/new.html.erb`

```
<%= form_for @planes do |f| %>
  <p>
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </p>

  <p>
    <%= f.label :kind %><br>
    <%= f.text_field :kind %>
  </p>

  <p>
    <%= f.label :description %><br>
    <%= f.text_field :description %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Our form should now submit properly. However, we will see that rails makes handling all the things required in a form easier using something called *form helpers* later.

An operational create method

We just need to save the data being sent in the request. We might be tempted to do the following.

`app/controllers/planes_controller.rb`

```
  PlanesController < ApplicationController

    ...

    def create
      plane = params[:plane]
      Plane.create(plane)
      redirect_to "/planes"
    end

  end
```

However, while this might be fine in rails `3.2` it won't fly in rails `4.0`, which has something called **strong parameters**. To follow this strong parameters convention we must change the way we accept params to something like one of the following.

Params Hash

In HTTP/HTML, the params are a series of key-value pairs. Params are a hash. Rails has special syntax for making params a hash w/ additional hashes inside:

We can grab the `:plane` hash out of the `params` hash, and the tell it to permit the keys we want: `:name`, `:kind`, and `:description`.

`app/controllers/planes_controller.rb`

PlanesController < ApplicationController

```
...

def create
  plane = params[:plane].permit(:name, :kind, :description)
  Plane.create(plane)
  redirect_to "/planes"
end
```

end

or, (preferably) just say `.require(:plane)`

`app/controllers/planes_controller.rb`

```
  PlanesController < ApplicationController

    ...

    def create
      plane = params.require(:plane).permit(:name, :kind, :description)
      Plane.create(plane)
      redirect_to "/planes"
    end

  end
```

In reality **strong params** is just a nice way of making sure someone isn't setting param values that you don't want them to be setting.

Refactoring our Index

We first need to setup our `#index` method in `planes`

`app/controllers/planes_controller.rb`

```
PlanesController < ApplicationController

  def index
    @planes = Plane.all
    render :index
  end

...

end
```

Let's finally put some `erb` in our `index` view.

`app/views/index.html.erb`

```
  <% @planes.each do |plane| %>

    <div>
      Name: <%= plane[:name] %> <br>
      Kind: <%= plane[:kind] %> <br>
      Description: <%= plane[:description] %>
    </div>

  <% end %>
```

Halfway there: Take off - SHOW

We've successfully made an `index` , `new` , and `create` . Next we will talk about adding a `show` , `edit` , and `update`

In routes.rb

```
get "/planes/:id", to: "planes#show"
```

In our controller

```
def show
 id = params[:id]
 @plane = Plane.find(id)
end

or

@plane = Plane.find(params[:id])
```

And then we create a show view (show.html.erb) and input the information using:

```
<h2><%= @plane.name %></h2>
<p><%= @plane.kind %></p>
```

In our index page we can also include a link to the individual plane page using `ruby <a href = "/planes/<%= plane.id %>">Show</a>`

We can dry this up later like so:

- `route.rb` - add `as: :plane` to the end of the show route.
- `index.html.erb` - `<%= link_to "Show me the individual plane!", plane_path(plane) %>`

Adding Edit + Update Actions

Edit + Update are related like New + Create. Remember that update does not need it's own view.

routes.rb

```
get 'planes/:id/edit', to: 'planes#edit', as: :edit_plane
patch 'planes/:id', to: 'planes#update'
```

edit.html.erb

Copy the form from `new.html.erb`

Planes Controller

```
  def edit
    @plane = Plane.find(params[:id])
  end

  def update
    @plane = Plane.find(params[:id])
    @plane.update_attributes(params.require(:plane).permit(:name, :kind, :description))
    redirect_to planes_path
  end
```

Adding Delete Action

routes.rb

`delete 'planes/:id', to: 'planes#destroy'`

Planes Controller

```
  def destroy
    @plane = Plane.find(params[:id])
    @plane.destroy
    redirect_to planes_path
  end
```

show.html.erb

`<%= link_to "Delete this Plane!", @plane, method: :delete, data: { confirm: 'Are you sure you want to delete it?' } %>`