RUBY ON RAILS

NOTES

Model, View, Controller and Rails App Structure - Text references

So Rails follows a MVC pattern of web applications and mvc creates a separation of presentation layer (what the user of the application sees in the browser/mobile device) and the business-logic or backend (invisible layer).

MVC - Model, View, Controller

MODELS :- models are the resources which are in our web application like :- users , post , stock , article, etc we know we have a database or tables in database , like for users we have users data base , hence we have models associated with these databases such that users model have users table and these models helps to communicate with databases.

VIEWS:- view makes the front-end of our application that will be seen by our users like homepage , port-folio page , etc and we will use ruby codes in the templates so ruby has embedded code for these templates

Hence , instead of .html file , we have to save file with .html.erb

CONTROLLERS :- controllers are invisible to the users bcz they are in backend , like for users we have users\_controller to control the user activity in our application and so on.

General flow of Rails application:

-> Request made at browser

-> Request received at router of rails application

-> Request routed to appropriate action in a controller

-> Controller#action either renders a view template or communicates with model

-> Model communicates with database

-> Model sends back information to controller

-> Controller renders view

Root route, controller and more MVC - Text directions and references

To set a root route, navigate to config/routes.rb file and enter in the following code ->

root 'pages#home'

In order to preview the application, you will need to start the rails server. You can start the server by navigating to the terminal and typing in the following command from your application directory followed by enter/return:

rails server

You can also type rails s for short. Once the server starts you can open up a browser window and type in localhost:3000 to preview the app. If you are using AWS cloud9, please use the videos in section 14 where previewing the applications are covered (after creating new rails apps using rails 5 and 6) since the preview link differs (it won't be localhost:3000).

Back to the app, in order for the code to work there will need to be a pages controller and a home action within it.

Create a pages controller by navigating to the terminal/command line and typing in the following command from your application directory:

rails generate controller pages

There should now be a pages\_controller.rb file under app/controllers folder.

Open up this file and add the home action to it with a rendering of 'Hello World!' like below:

1. def home
2. render html: 'Hello World!'
3. end

Once you save this file and reload the browser preview, Hello World! should be displayed in the browser.

In order to do this the conventional way using views remove the code within the home action and leave it as an empty method/action like below:

1. def home
2. end

This will expect a home.html.erb template under the pages folder under views. This pages folder under the views folder was created when the pages controller was generated.

Within this app/views/pages folder, create a new file called home.html.erb and add in the following code inside the file:

Hello World!

Once you save this file and preview it from the browser, it should still say Hello World! as expected.

Congratulations, you have made your Rails application say 'Hello World!'

GENERATE A VIEW

1. define a route that points to a controller#action
2. have an approximately named controller , for example : if dealing with layouts or static pages of the application, a name could be pages\_controller
3. have an approximately named action, for example : if dealing with a homepage, the action/method could be named home
4. if done this way, under views, rails will expect a pages folder (named for the pages controller) and a [home.html.erb](http://home.html.erb) template (named for the home action)

Add About page and homework assignment - Text reference and code

To add an About page to the application, first add the following route to the config/routes.rb folder:

get 'about', to: 'pages#about'

You can then add the about action to the pages controller like below:

1. def about
2. end

Now you will need a view template, so add an about.html.erb file under the app/views/pages folder and fill in some text to display within it:

<h1>This is the About page</h1>

Once you save the file, you'll be able to preview this page by appending /about to your homepage. In your local machine it would be localhost:3000/about

Homework:

1. Create an alpha-blog application.

2. Setup tracking with Git and make a commit of your code with the message "Initial commit". Set up a GitHub repository for the app.

3. Create a home page view (as the root route) and an about page like the test app.

4. Make sure you are able to start the server and preview the application. Take snapshots of the two pages (home and about) and post them to the Q & A.

5. Make a commit of your code and push to your GitHub repository for the application.

CRUD and scaffold generators - Text directions, references and code

Rails uses a middle man named as ORM :- Object Relational Mapper to communicate between rails application code and database as we know in rails we use ruby codes so for communicate with database rails uses ORM bcz it resolves the problem of using SQL queries as we can use ruby and in rails this ORM is known as “Active Record”.

In models folder we see a file named as application\_record.rb which has the following code containing Active record follow with Application record which we use further

class ApplicationRecord < ActiveRecord::Base

  self.abstract\_class = true

end

Query language to communicate with database: SQL (Structured Query Language)

CRUD actions:

C - Create

R - Read

U - Update

D - Delete

Scaffold generator command to create an article model (with two attributes), articles controller, views for articles and migration file to create articles table:

rails generate scaffold Article title:string description:text

Command to see routes presented in a viewer-friendly way:

rails routes --expanded

The line resources :articles in the config/routes.rb file provides the following routes:

- index of articles (GET request)

- new article (GET)

- create article (POST)

- edit article (GET)

- update article (PUT and PATCH)

- show article (GET)

- delete article (DELETE)

From UI perspective ->

- index lists all the articles in the articles table in the database of the app

- new article deals with the form to enter in new article details

- create handles the submission of the items in the new article form

- edit article deals with the form to enter edited information for an existing article

- update article deals with the submission of the edit article form

- show article displays an individual article based on selection

- delete article deletes an article from the articles table

In preparation for the next section, learn and practice SQL here: <https://www.w3schools.com/sql/>

MY FIRST RAILS APPLICATION :- PRINCE\_VLOGS

Intro to tables, migrations, rails conventions - text references

#### Details

Model name: article

Class name: Article -> Capitalized A and singular, CamelCase

File name: article.rb -> singular and all lowercase, snake\_case

Table name: articles -> plural of model name and all lowercase

Additional example:

Model name: user

Class name: User -> Capitalized U and singular, CamelCase

File name: user.rb -> singular and all lowercase, snake\_case

Table name: users -> plural of model name

Generate a migration to create a table (in this example articles):

rails generate migration create\_articles in my “db folder”

To add attributes for the table in the migration file, add the following inside create\_table block:

t.string :title

To run the migration file, run the following command from the terminal:

rails db:migrate

The first time you run the migration file, it will create the database, the articles table and a schema.rb file.

To rollback or undo the changes made by the last migration file that was run, you may use the following command:

rails db:rollback

If you have run the rollback step, then you can update the previous migration file and add the following line to add a description attribute (column) to the articles table:

1. t.text :description

To run the newly edited migration file again, you can run rails db:migrate

Note: This above line will only work if you had rolled back the prior migration.

To generate a new migration file to add or make changes to your articles table you can generate a new file:

rails generate migration name\_of\_migration\_file

Then within the def change method in the migration file you can add the following lines:

1. add\_column :articles, :created\_at, :datetime
2. add\_column :articles, :updated\_at, :datetime

HERE, add\_column is word to create a column in the table and (:articles)is the name of the table and then second is the attribute name , its have to be created\_at otherwise rails will not get this magic field for us and then the last argument is data type (i.e datetime)

Now we will get the new migration file or we can say the latest migration file by typing rails db:migrate in prompt

You can run the newly created migrations file by running rails db:migrate from the command line and check out the schema.rb file to check that the changes were reflected properly.

Models and rails console - text references

To create an article model, create an article.rb model file under app/models folder and fill it in:

class Article < ApplicationRecord

end

Every model we create will inherit from this ApplicationRecord file. Here, Article is the name of the model and now with the help of this model we can communicate with the database

Note: Make sure ApplicationRecord is CamelCase.

Now, provided you have the articles table already, you can use the Rails console and work with the articles table using this article.rb model file.

To start the rails console, type in rails console  or rails c from the terminal.

Once in the console, you can exit it at any time by typing in exit followed by enter/return.

You can test out your connection to the articles table by typing the following command from within your rails console:

Article.all

If you get an empty collection/array-like structure as a response, you're good to go.

To create a new article, you can use any of the following methods:

1. Article.create(title: "first article", description: "Description of first article") # make sure Article is capitalized if using this method
2. article = Article.new
3. article.title = "second article"
4. article.description = "description of second article"
5. article.save
6. article = Article.new(title: "third article", description: "description of third article")
7. article.save

To check all the articles that exist in your articles table, you can use the following command:

Article.all

CRUD ops from rails console - text directions and code

To find an article by id you can use the find method like below:

Article.find(1) # replace 1 with id of article you want to find

You can save this to a variable and use it like below

1. article = Article.find(1)
2. article.title # to display (get) the title
3. article.description # to display (get) the description

You can use the methods below to view the first and last articles of the articles table:

1. Article.first # display the first article in the articles table
2. Article.last # display the last article in the articles table

You can update an article by finding it first and then using setters for the attributes that the model provides like below:

1. article = Article.find(id of article you want to edit)
2. article.title = "new title"
3. article.description = "new description"
4. article.save

You can delete an article by using the destroy method. A sample sequence could be like below:

1. article = Article.find(id of article you want to delete)
2. article.destroy

VALIDATIONS:-

Validations enforce constraints on your model so you can have greater control on what you are allowing as data to be saved in your database/tables.

We do this bcz we don’t want to save the articles without title or description. So to do this we will go to our file/model article.rb and validate in this file

class Article < ApplicationRecord

class Article < ApplicationRecord

    validates :title, presence: true, length: {minimum: 6, maximum: 100}

    validates :description, presence: true, length: {minimum: 10, maximum: 500} # this ensures the presence of title of any article to be saved

end

    validates :title, presence: true, length: {minimum: 6, maximum: 100}

    validates :description, presenclass Article < ApplicationRecord

validates :title, presence: true, length: {minimum: 6, maximum: 100}

validates :description, presence: true, length: {minimum: 10, maximum: 500} # this ensures the presence of title of any article to be saved

endce: true, length: {minimum: 10, maximum: 500} # this ensures the presence of title of any article to be saved

end

Here validates is used for validation and now presence: true will ensure that title must be present and then the length is used for make the min. and max. length for the article.

And for more validations we can go to google and type “active record validations”

SOME IMPORTANT TERMS:-

1. @Article = Article.find(params[:id]) , Here params

Is a hash data structure it means when we type id in url it will get contents of this id.

1. <% @articles.each do |article| %>

<% end %>

here @articles in an attribute in index method and .each is used for getting all the articles

1. Form\_with is a type of form helper and it sub,its using ajax by default.

<%= form\_with(model: @article, class: "shadow p-3 mb-3 bg-info rounded", local: true) do |f| %>

Here we have to provide the model name

<%= f.submit class: "btn btn-warning" %>

The best part about form\_with is that if you just pass it a model object like @article in the example above, Rails will check for you if the object has been saved yet. If it’s a new object, it will send the form to your #create action. If the object has been saved before, so we know that we’re editing an existing object, it will send the object to your #update action instead. This is done by automatically generating the correct URL when the form is created. Magic!

1. params.require(:article).permit(:title,:description)

This is used for creating an article permit is used for permitting the title and description

1. # redirect\_to article\_path(@article)

Redirect to show page for this id and now @article we know we used it during creation so to redirect to the required id we use this @article.

1. Layout links:- adding paths by clicking on links/buttons

<%= link\_to "View", article\_path(article), class: "btn btn-outline-success" %>

Link\_to is used for linking and article\_path is the prefix or path for this and “view” is for displaying a link or button name

Like for delete and edit , we have

But for delete I used a model so it is something different shown here, I didn’t display modal code

<%= link\_to "Edit", edit\_article\_path(article), class: "btn btn-outline-info" %>

<%= link\_to 'Delete', '#', "data-toggle"=>"modal", "data-target" => "#delete-#{article.id}",:class=>'btn btn-outline-success' %></td>

1. truncate:- we will use truncate method for this bcz length can be of any limit so we want to limit it
2. time\_ago\_in\_words:- we will use this handy method provided by rails time\_ago\_in\_words as it is an helper method , we can display time without this also but it will not give us a clear pattern or we can say it will not be a user friendly
3. simple\_format :- it is used to display the text/description as it is in which format it was uploaded/saved.

FLASHING MESSAGES AND VALIDATIONS :-

def create

@article = Article.new(article\_params)

if @article.save

flash[:notice] = "Article was created succesfully." # we can also use alert insted of notice

# redirect\_to article\_path(@article)

redirect\_to @article

else

render 'new'

end

end

now if article is not saved it will render to new page and we will get error there so we use this code to get errors

<% if @article.errors.any? %>

<h4 class="alert-heading">The following errors prevented the article from being saved</h4>

<ul>

<% @article.errors.full\_messages.each do |msg| %>

<li><%= msg %></li>

<% end %>

NOW for when article is saved , we will use a flash which is a rails provider helper

flash[:notice] = "Article was created succesfully." # we can also use alert insted of notice and we will do the display work in our main layout file that is application.html.erb file by using this code

<% flash.each do |name,msg| %>

<div class="alert alert-success alert-dismissible fade show" role="alert">

<%= msg %>

<button type="button" class="close" data-dismiss="alert" aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

</div>

</div>

<% end %>

It takes two arguments(i.e key value pair)name for the key and msg for msg displayed we can take any name.

INSTALL BOOTSTRAP ,ASSET PIPELINE,JAVASCRIPT,WEBPACK:-

Run the following command from your application directory (if you don't have yarn, install it):

***yarn add bootstrap@4.3.1 jquery popper.js***

Now for JavaScript

**Make the config/webpack/environment.js file look like below:**

Make the config/webpack/environment.js file look like below (you are adding the lines which are highlighted in green below):

*const { environment } = require('@rails/webpacker')*

*const webpack = require("webpack")*

*environment.plugins.append("Provide", new webpack.ProvidePlugin({*

*$: 'jquery',*

*jQuery: 'jquery',*

*Popper: ['popper.js', 'default']*

*}))*

*module.exports = environment*

This is so your application's javascript understands jquery and popper.js syntax.

Now you have to import bootstrap and it's goodies to your application.js, go to app/javascript/packs/application.js and add the following line to the bottom:

*import "bootstrap"*

That's it for javascript, now onto CSS

Now onto CSS

Go to app/assets/stylesheets/application.css and add the following line above the require\_tree and require\_self lines:

*\*= require bootstrap*

So it looks like below:

*\*= require bootstrap*

*\*= require\_tree .*

*\*= require\_self*

Now to add custom styles to the app styling, create a new file within this folder (app/assets/stylesheets folder) and call it custom.css.scss (app/assets/stylesheets/custom.css.scss).

Here import the bootstrap library first so you are able to modify styles for not just your new classes but existing bootstrap classes as well:

*@import 'bootstrap/dist/css/bootstrap';*

**CREATING USERS TABLE: -**

**To create a migration to create users table:**

**$ rails generate migration create\_users**

**Then within the migration file, add in the following code within the create\_table block to add the username, email and timestamps columns (created\_at and updated\_at):**

**t.string :username**

**t.string :email**

**t.timestamps**

**To run the migration file and create the users table:**

**$ rails db:migrate**

**Now create the User model file. Add a user.rb file under app/models folder and fill in the following:**

**class User < ApplicationRecord**

**end**

**Then you can test out the mode using the rails console. You can start the rails console and try out the following commands (or other commands as you choose):**

**$ rails console**

**User.all**

**User**

**user = User.create(username: "test", email: "test@example.com")**

**user = User.create(username: "test2", email: "test2@example.com")**

**# To grab first user and update their email address:**

**user = User.find(1)**

**# OR**

**user = User.first**

**# update the email**

**user.email = "test3@example.com"**

**user.save**

**# To destroy user with id of 2 (if that id exists in your users table, or select**

**# an appropriate id that does exist**

**user = User.find(2)**

**user.destroy**

**VALIDATIONS:-**

**validates :username, presence: true, uniqueness: { case\_sensitive: false}, length: {minimum:3, maximum:25}**

**VALID\_EMAIL\_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i#format of email for email verification**

**validates :email, presence: true, uniqueness: { case\_sensitive: false}, length: { maximum:105},format: { with: VALID\_EMAIL\_REGEX }**

**uniqueness: {case\_sensitive: false} means a user will have a unique id without any consideration of capital or small letters**

NOW CREATE A CONNECTION BETWEEN USERS MODEL AND ARTICLES MODEL BY ONE TO MANY ASSOCIATIONS

To generate a migration to add the user\_id column to articles table, you can use the command below:

$ rails generate migration add\_user\_id\_to\_articles

Then within the change method fill in the code specifying the change:

add\_column :articles, :user\_id, :int

Run the migration file to effect the change:

$ rails db:migrate

To form the association among the models, add the following line to article.rb model file:

belongs\_to :user

And add the following line to user.rb model file:

has\_many :articles

Ensure you have a couple of users in your users table created using the rails console. Then add in a line in the create action to temporarily grab and hardcode a user to each article that's created:

def create

@article = Article.new(article\_params)

@article.user = User.first # <--- Add this line this is for temporary basis so that every article will get first user

if @article.save

flash[:notice] = "Article was created successfully."

redirect\_to @article

else

render 'new'

end

end

IMP.NOTE:- during validations write this code also

before\_save {self.email = email.downcase} # this is used to save the entered email in downcase letters even if user enters any uppercase letter

NOW ADDING SECURE PASSWORD FOR A USER:-

3 step process to add auth system functionality from the back-end.

Step 1) Add bcrypt gem:

In the Gemfile uncomment the line that lists the gem:

gem 'bcrypt', '~> 3.1.7'

Then run $ bundle install to install the gem in your app.

Step 2) Add has\_secure\_password to your user model. Add the line below in your user.rb model file:

has\_secure\_password

Step 3) Create a migration file to add the password\_digest column to the users table.

$ rails generate migration add\_password\_digest\_to\_users

Then pull up the migration file and fill in the column details within the def change method:

add\_column :users, :password\_digest, :string

Save the file and run $ rails db:migrate to make the change to the table

You can test out the functionality from the console by adding passwords to a couple of your existing users and verifying them using the authenticate method. A sample progression of commands from the console could be like below:

user = User.last

user.password = "password123"

user.save

The commands above will add the hashed version (with salt) of the string "password123" to the user's record.

Then you can authenticate and test the password for the user like below:

user = User.last # (or User.find(enter id of user here))

user.authenticate("password123") # This, being the correct password will return the user object

user.authenticate("enterincorrectpassword") # This, being an incorrect password, will return false

NEW USER SIGNUP:-

To add the route for the new user signup page and the rest of the restful routes separately, add the following two lines to your routes.rb file:

get 'signup', to: 'users#new'

resources :users, except: [:new]

The changes made in the video in addition to the routes above are listed below (reference the commit for the specific code):

- Create users controller.

- Add new action along with initiating a new user instance variable.

- Create a users folder under app/views and add the new.html.erb template.

- Add a form partial, \_form.html.erb, in the same folder.

- Update app/views/shared/\_errors.html.erb and replace @article references with obj.

- Update app/views/articles/\_form.html.erb and add obj: @article when rendering the errors partial.

Test out the form by going to /signup page and ensure it shows up.

Show user and profile image - text directions and code

To have a profile image associated with an email account you control, add one using the site en.gravatar.com. This will be the globally recognized avatar (gravatar for short) associated with that email address (this step is not necessary to get the functionality working in the app).

Summary of changes (actual ode can be found in the commit):

- Add show action in users controller and a show.html.erb template under views/users.

- Add gravatar\_for as a helper method in app/helpers/application\_helper.rb file.

- Extract the articles listing code from views/articles/index.html.erb file and add it to an \_article.html.erb partial within the same folder.

- Render this partial from both the show and index templates. Be sure to initiate the appropriate instance variable from the users show action.

ADD PAGINATION TO VIEWS:-

## perform a paginated query:

@posts = Post.paginate(page: params[:page])

# or, use an explicit "per page" limit:

Post.paginate(page: params[:page], per\_page: 30)

## render page links in the view:

<%= will\_paginate @posts %>

# set per\_page globally

WillPaginate.per\_page = 10

Then use this code in users show page for showing articles pagination

@articles = @user.articles.paginate(page: params[:page], per\_page: 3)

Then use this code in users index page for showing users pagination

@users = User.paginate(page: params[:page], per\_page: 3)

For styling in paginations we can refer to this link

<http://mislav.github.io/will_paginate/>

NOW ADD LOGIN FORM FOR USERS:-

- Added routes for login - new, create and destroy (to represent login form, logging in a user and logging out a user).

- Added a sessions controller with the new, create and destroy actions.

- Added a sessions folder under app/views and created a new.html.erb template to display the login form.

CREATE AND DESTROY SESSIONS FOR USERS:

- Completed create and destroy actions for sessions controller to log users in and out.

- Added log in and log out links to navigation partial.

AUTHENTICATION HELPER METHODS:-

- Added current\_user and logged\_in? helper methods.

- Modified the navigation partial and utilized these methods to display links based on logged in state.

- Modified the users controller create action so users' are logged in once they sign-up.

- Added set\_user method and used it as before\_action for show, edit and update actions in users controller.

CONTROLLER METHODS AS HELPER METHODS:-

- Moved the current\_user and logged\_in? authentication methods from application helper to application controller and declared them as helper methods so they can be used in views as well.

- Updated the create action in articles controller so the logged in user is associated with the article being created instead of hardcoding the user.

- Updated pages controller home action (root route) to redirect to articles index page if users are logged in.

ADMIN USER INTRO:-

- Generated migration file to add admin field to users table, with a default value of false for all users.

Whether a user is admin or not can be tested by first getting a user object in the rails console and checking user.admin? from there.

To switch a user to an admin user you can use the following command in the rails console (assuming you have already selected a user object):

user.toggle!(:admin)

FACILITIES FOR ADMINS:-

- Navigation partial updated to display "(Admin)" for admin user accounts.

- Article actions (edit and delete) made available to admins for all articles in both for articles index (rendering article partial) and articles show views.

- Added ability for admin users to delete other user accounts through the users index page.

MANY TO MANY ASSOCIATIONS AND AUTOMATED TESTING –

INTEGRATION , FUNCTION AND MUCH MORE:-

- Create new migration file to add article\_categories table with category\_id and article\_id columns.

- Create article\_category.rb model file and interact with the article\_categories table.

- Add the many-to-many association code using the relevant has\_many through: association helpers in the article, category and article\_category models.

CODE IN ARTICLE\_CATEGORY MODEL:-

belongs\_to :article

belongs\_to :category

CODE IN ARTICLE MODEL:-

has\_many :article\_categories

has\_many :categories, through: :article\_categories

CODE IN CATEGORY MODEL:-

has\_many :article\_categories

has\_many :articles, through: :article\_categories