Ruby On Rails: A Straightforward Primer

There are two modes for implementing a Rails app,

- Static Mode
- Dynamic Mode

In ***Static Mode*** there is no interaction with a database and the programme doesn't contain any models responsible for communicating with the databases.

In ***Dynamic Mode*** the app implements CRUD workflow by interacting with a database from within the models of the app.

Starting a new project

To create a new Rails project, simply execute the following in the terminal,

```
$ rails new ProjectName
```

Install all packages from within the Gemfile using bundle like so,

```
$ bundle install
```

Run the server and go to the localhost page to see if the app is successfully created,

```
$ rails server
```

OR

```
$ rails s –p 4001
```

In the second command, the port is defined to be equal to 4001.

Static Mode

While in static mode, the programme does not interact with any databases and thus, does not contain any models.

The 3 primary components are,

- Controller
- Route
- View

To create a `controller` called Pages, execute the following command,

```
$ rails generate controller Pages
```

That creates all necessary files for the newly created `PagesController` (don't forget to use plural when generating a new controller).

Let's say that we want to create a new route which takes the user to the homepage of the app. For that we need to execute the following steps,

1. Add a controller action (method) called home (could also be index) inside of the PagesController class.

```
class PagesController < ApplicationController

    def home
    end

end
```

By convention, we say that the Pages controller now has a home action.

2. We then create the route that relates the GET request with the controller by putting the following command in the routes file,

```
get 'welcome' => 'pages#home'
```

If we wanted to set our home page as the root route for our app ('/'), then we could do,

```
root 'pages#home'
```

The first part of the string `'pages#home'` corresponds to the controller we're referring to and the second one to the respective action.

3. Having created both the controller and the route for our pages homepage, the only thing missing is the view file. There is a convention (pattern) followed here as well. Inside of the views directory, we create another directory for the pages controller,

```
$ mkdir views/pages/
```

and then, a view file that has the action name in it, like so,

```
$ touch views/pages/home.html.erb
```

> What's really important to note is that a controller's view file is different from a layout view file. In short, a layout file is an html file containing all information about the `head`, the `header` and `footer` as well as any CSS that the app is using. If you want to create something that shows up on every page of the app then you should put it in the layout file.

> The controller's view files contain standalone sections of html with some erb (embedded ruby) in them that solely correspond to a certain action of a specific controller. In order to have a complete HTML document for the app, we need to combine the layout with the controllers' view files by using <%= yield %&> in the body of the layout file.

Think of the layout being the template of the app, containing the child templates (controller's views) in its body.

A useful piece of info would be how to import custom Google Fonts for the apps we're creating,

```
<link href="https://fonts.googleapis.com/css?family=Oxygen,300,400,300"
rel="stylesheet" type="text/css">
```

This example uses a specific font family but you can change it to whichever one you want to work with.

Working with data and a Database

In this mode, one extra element must be introduced, called the Model, whose role is to interact with the database and work with data. The workflow for creating a new model, is the following,

1. Create a new rails model

```
$ rails generate model Message
```

Make sure that the model name is in singular.

2. Inside of the db/migrate/ dir find the database table created and notice that it has a timestamp attribute to it by default. The timestamp corresponds to 2 columns in the table, namely, `created_at` and `updated_at`, which save the datetime at creation or update. A sample new column would be,

```
    t.text :context
```

where `text` defines the type of the column's data (could be t.string for example), and `:context` is the column's name (context).

3. After editting the table's contents, we need to update the database to save the changes, for that, execute one of the following,

```
$ bundle exec rake db:migrate
```

OR

```
$ rails db:migrate
```

4. After putting in all desired columns, we can seed the table with our default data that are found inside the db/seeds.rb file, using the following command

```
bundle exec rake db:seed
```

Our migrations file after these changes would look like this,

```
class CreateMessages < ActiveRecord::Migrations
    def change
        create_table :messages do |t|
            t.text :content
            t.timestamps
        end
    end
end
```

Example Messaging Functionality

Let's assume that we would like to add some functionality to our app so that we can create a message on the website using a form, and then post it there and having it recorded in the database.

First, we need to `GET` the page that contains the form. For that, we need to work with the router and controller,

Router

```
get '/messages/new' => 'messages#new'
```

That means that for creating a new message, we access the 'url/messages/new' route in the url and that communicates with the messages controller's new action (new method).

Controller

```
def new
    @message = Message.new
end
```

In the controller, we create a new action that creates a new instance of the Message class and returns it to the `new.html.erb` view file.

We want to post the message for the data to be recorder in the database, so, we need a `POST` route.

Router

```
post 'messages' => 'messages#create'
```

Controller

We need to have access to the message attributes from inside the message table, so that we can safely collect data from the form and update the database with them, and for that we need the following private action inside of the MessagesController,

```
private

def message_params
    params.require(:message).permit(:content)
end
```

After writing the new and message_params actions, we need to write the `create` action that creates the message,

```
def create
    @message = Message.new(message_params)
    if @message.save
        redirect_to '/messages'
    else
        render 'new'
    end
end
```

Essentially, if the new message is successfully saved in the database, the user gets redirected to the '/messages' route. If not, the user is taken to the 'new' route.

For creating a new message,

1. Create a view file

```
$ touch app/views/messages/new.html.erb
```

Let's create the form inside that view file,

```
<%= form_for(@message) do |f| %>
    <div class="field">
        <%= f.label :message %><br>
        <%= f.text_area :content %>
    </div>
    <div class="actions">
        <%= f.submit "Create" %>
    </div>
<% end %>
```

To access the form from the home page, let's enter a link on the home page,

```
<%= link_to 'New Message', 'messages/new'%>
```

According to the 7 actions of Rails,

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

instead of defining each route separately in the route file, we can assign all routes to a controller by typing,

```
resources :messages
```

For using all routes except for 2 specific ones, use,

```
resources :messages, only: [:index, :show]
```