Introduction to Ruby on Rails

León Jaramillo

softserve

Ruby on Rails

- **Ruby on Rails** (RoR or Rails) is a web framework written in Ruby.
- Rails was developed by the Danish developer **David Heinemeier Hansson**, and its current **version** is 7.0.
- It’s a **server-side** framework, which implements the Model-View-Controller (**MVC**) pattern.
- Rails provides a **default directory structure** to manage the database, web pages and other assets.
- It relies on known **patterns**, such as, Convention over Configuration, Don’t repeat yourself (DRY) and Active Record.
- Rails **influenced** many web frameworks, like Grails, Laravel, CakePHP AND Django.
- It is used in **sites** like GitHub, Twitch, Shopify and Airbnb.

Setting up the environment

- **Ruby** should be installed (check using `ruby -v` in a CLI).

- A DBMS. **SQLite3** is recommended for learning and testing purposes.
  1. Download your OS's version from here: https://www.sqlite.org/download.html
  2. Follow the instructions for your OS found here: https://www.tutorialspoint.com/sqlite/sqlite_installation.htm
  3. In Windows, it will consist in:
     - Downloading SQLite DLL and command line tools.
     - Unzipping all of it in a newly created folder.
     - Adding this folder to the PATH environment variable, as you'll find out here: https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/
  4. You can check using `SQLite3 --version` in a CLI.

- Install Rails using `gem install rails` in a CLI. Then, you can check using `rails --version`

softserve

Creating an app

- You can perform many tasks using the `rails` command and its several instructions.
- You can create a new app using:

```
rails new [app_name]
```

- The command above will create a directory structure detailed here: https://guides.rubyonrails.org/getting_started.html
- Once you are situated in the directory created, you can tun the app using the command:
  - `ruby bin/rails server` in Windows.
  - `bin/rails server` otherwise.
- You can stop the server using *Control+C*.

softserve

Basic Components in Ruby on Rails

- Basic **components** in Ruby on Rails are routes, controllers, actions and views.
- A **route** maps a request to a controller action. It's a rule written in a Ruby DSL.
- A **controller action** performs the necessary work to handle the request, it includes preparing data for a view. The controller is a Ruby class whose public methods are actions.
- A **view** prepares the resulting data in a desired format. Views are templates, written using HTML and embedded Ruby.

Creating a “Hello, World”

1. Add a route to the `config/routes.rb` file:

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

This means that `GET products` requests map to the `index` action within `ProductsController`.

2. Create the appropriate controller and view (but not the route) using the following command:

```
ruby bin/rails generate controller Products index --skip-routes
```

3. It will create, among other files, the controller (`product_controllers.rb`) and the view (`index.html.erb`).

4. You can invoke the view from the controller. If not, the framework will invoke a matching one. You can now edit both files.

softserve

Creating a Model

- As we have seen, Rails is based on the MVC pattern, where the Model manages the data of the application.

- Just as with controllers, we can create a model using:

```
ruby bin/rails generate model Product name:string description:text
```

- You can see on screen what files were created. However, we'll focus on the model (`product.rb`) and the migration (`_create_products.rb`) files. The table created will have its explicit fields, an id field and its timestamps.

- We should, also, run the migration using: `ruby bin/rails db:migrate`

- In Ruby, as in many similar frameworks, we use **migrations** to alter the structure of the application's database.

Interacting with the Model Using the Console

- We can run the console using: `ruby bin/rails console`
- Using the console, we can create a product, using: `a_product = Product.new(name: "Xbox", description: "A good gaming console!")`
- Then, we can save it to the database, using: `a_product.save`
- We can get all the products in the database, using: `Product.all`
- Also, we can get a product given its id, using: `Product.find(1)`
- This can be useful to automate database population.

softserve

Listing from a Model

- In Rails, we can **list**, easily, the items from a model. In this case, we'll list all the items from the products model.

- Firstly, we get all the items from the **model**, in the corresponding **action** of the desired **controller**. In our case, we add the following line in the **index** action of the **products** controller: `@products = Product.all`

- Then, we can iterate over the **instance variable** `products` in the **index** view, resulting in the products list:

```
<h1>Products</h1>
<ul>
  <% @products.each do |product| %>
    <li>
      <strong><%= product.name %>:</strong> <%= product.description %>
    </li>
  <% end %>
</ul>
```

- Now, we can review the whole **MVC** pattern as implemented by Ruby on Rails.

softserve

Using a Route Parameter (1/2)

- For a controller, it is common to receive requests with **parameters**. They usually include some info that is important within the request.
- For instance, we'll enable to show the information of just one product receiving its **id** as a parameter.
- Firstly, we'll add the following route in the corresponding file: `get "/products/:id", to: "products#show"`
- Then, we'll add the corresponding action to the products controller (below the `index` one).

```
def show
   @product = Product.find(params[:id])
 end
```

softserve

Using a Route Parameter (2/2)

- Now, let's create a view (show.html.erb) with the content that you want, but including the following lines:

```
<h1><%= @product.name %></h1>
<p><%= @product.description %></p>
```

- Finally, we'll modify the `index` view to include the link to the other view:

```
<h1>Products</h1>
<ul>
  <% @products.each do |product| %>
    <li>
      <strong><%= product.name %>:</strong>
      <a href="/products/<%= product.id %>">See more</a>
    </li>
  <% end %>
</ul>
```

softserve

Towards a CRUD Using Ruby on Rails

- **Many** use cases while developing a web application comprise **CRUDs**. Rails offers some tolos to ease their development.
- Since creating a CRUD implies creating several routes, Rails implements **resourceful routing**. It’s about defining a resource, which will automatically create the routes needed.
- To check this, firstly run this command in the CLI `rails routes`, and among others, we’ll see just our two routes.
- Now, replace the two existing routes for these lines of code, the index and a resource:

```
root "products#index"

resources :products
```

- Then, let’s inspect the **routes** again.

softserve

Some Helpers Enabled by Resources

- Resources define some **helpers** to get the right paths easily given a model.
- One of those helpers is `_path`, which will give us the **proper path** for a given product:

```
<h1>Products</h1>
<ul>
  <% @products.each do |product| %>
    <li>
      <strong><%= product.name %>:</strong>
      <a href="<%= product_path(product) %>">See more</a>
    </li>
  <% end %>
</ul>
```

- The other one is `link_to`, which will generate **a link** to the path of the model in its second parameter, with the text in its first one:

```
<h1>Products</h1>
<ul>
  <% @products.each do |product| %>
    <li>
      <strong><%= product.name %>:</strong>
      <%= link_to "See more", product %>
    </li>
  <% end %>
</ul>
```

Creating a New Product

- For creating a new record, a Rails app follows, typically, these steps:
  1. The client requests a form to fill out.
  2. The user at the client fills and submits it.
  3. If there is no error, the record is saved, followed by any sort of confirmation.
  4. If there is any error, the form indicates it.

- Adding the following code to our existing controller (bellow the `show` action) will enable these steps (using **strong parameters**):

```
def new
   @product = Product.new
 end

 def create
   @product = Product.new(product_params)

   if @product.save
     redirect_to @product
   else
     render :new, status: :unprocessable_entity
   end
 end

 private
   def product_params
     params.require(:product).permit(:name, :description)
   end
```

Building the Proper View

- In addition to writing the controller, we should write the **view**.
- We can build a form which fully complies with Rails conventions using a **Form Builder**.
- It uses embedded Ruby, specifically, **Action View Form Helpers** (https://guides.rubyonrails.org/form_helpers.html).
- In our case, we could use the following code in the (new) **file** `new.html.erb`, in the views' folder.
- This will render an HTML file, easing the **management** of the model within the form.

```
<h1>New Product</h1>
<%= form_with model: @product do |form| %>
  <div>
    <%= form.label :name %><br>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.label :description %><br>
    <%= form.text_area :description %>
  </div>

  <div>
    <%= form.submit %>
  </div>
<% end %>
```

softserve

Linking to the View

- Now, we already have **set up** the route, the controller, the model and the view.
- However, we should not forget to **link** our app to that new feature.
- As for many other tasks, we can use a **helper** considering strongly **convention over configuration**.
- Adding the following line to our `index` view, adds the proper link:

```
<%= link_to "New Product", new_product_path %>
```

softserve

Updating an Existing Product

- Just as creating a product, updating it is a multi-step process:
  1. The client requests a form to fill out.
  2. The user at the client fills and submits it.
  3. If there is no error, the record is saved, followed by any sort of confirmation.
  4. If there is any error, the form indicates it.

- Adding the following code to our existing controller (bellow the `new` and `create` actions and above `product_params`) will enable these steps:

```
def edit
    @product = Product.find(params[:id])
  end

  def update
    @product = Product.find(params[:id])

    if @product.update(product_params)
      redirect_to @product
    else
      render :edit, status: :unprocessable_entity
    end
  end
```

Building the Proper View (For Editing)

- In addition to writing the controller, we should write the **view**.
- We can build a form which fully complies with Rails conventions using a **Form Builder**.
- It uses embedded Ruby, specifically, **Action View Form Helpers** (https://guides.rubyonrails.org/form_helpers.html).
- In our case, we could use the following code in the (new) **file** `edit.html.erb`, in the views' folder.
- In this case the code is **almost the same** than when creating a new product.

```
<h1>Edit Product</h1>
<%= form_with model: @product do |form| %>
  <div>
    <%= form.label :name %><br>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.label :description %><br>
    <%= form.text_area :description %>
  </div>

  <div>
    <%= form.submit %>
  </div>
<% end %>
```

softserve

Linking to the View (For Editing)

- Now, we already have **set up** the route, the controller, the model and the view.
- However, we should not forget to **link** our app to that new feature.
- As for many other tasks, we can use a **helper** considering strongly **convention over configuration**.
- Adding the following line to our `show` view, adds the proper link:

```
<p><%= link_to "Edit", edit_product_path(@product) %></p>
```

Deleting an Existing Product

The Controller

- In our case, enabling products deletion just needs a controller, a route and is corresponding link.
- The route is already created (just check them), so we'll add this **action** to our existing controller (bellow the `update` one):

```
 def destroy
    @product = Product.find(params[:id])
    @product.destroy

    redirect_to root_path, status:
:see_other
  end
```

The Link to The Controller

- Finally, we'll add the "Delete" **link** in our `show` view:

```
<p>
    <%= link_to "Delete", product_path(@product),
data: {
        turbo_method: :delete,
        turbo_confirm: "Are you sure?"
        } %>
</p>
```

- In the above code we use **Turbo** (https://turbo.hotwired.dev/) which helps us with some JS-related tasks.
- In pages like these, it's nice to have a link to **get back** to the items list. So, add the following code bellow the "Delete" link:

```
<p><%= link_to "Back", root_path %></p>
```

softserve

Useful Resources

- MVC Pattern (English): https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/
- MVC Pattern (Spanish): https://codigofacilito.com/articulos/mvc-model-view-controller-explicado
- Convention over Configuration: https://senthilnayagan.medium.com/convention-over-configuration-d17930f712de
- Active Record Basics (Ruby on Rails): https://guides.rubyonrails.org/active_record_basics.html
- Getting Started with Rails: https://guides.rubyonrails.org/getting_started.html
- Ruby on Rails Controllers Overview: https://guides.rubyonrails.org/action_controller_overview.html
- Rails Routing from the Outside In: https://guides.rubyonrails.org/routing.html
- Action View Overview: https://guides.rubyonrails.org/action_view_overview.html
- Learn Ruby on Rails - Full Course: https://www.youtube.com/watch?v=fmyvWz5TUWg&ab_channel=freeCodeCamp.org
- Curso Ruby on Rails en Español: https://www.youtube.com/watch?v=0Qj3LUxx3Zg&list=PLP06kydD_xaUS6plnsdonHa5ySbPx1PrP&ab_channel=aprendev

softserve

COLOMBIA DC

Thanks! Any question?

softserve