Rails Reference Guide

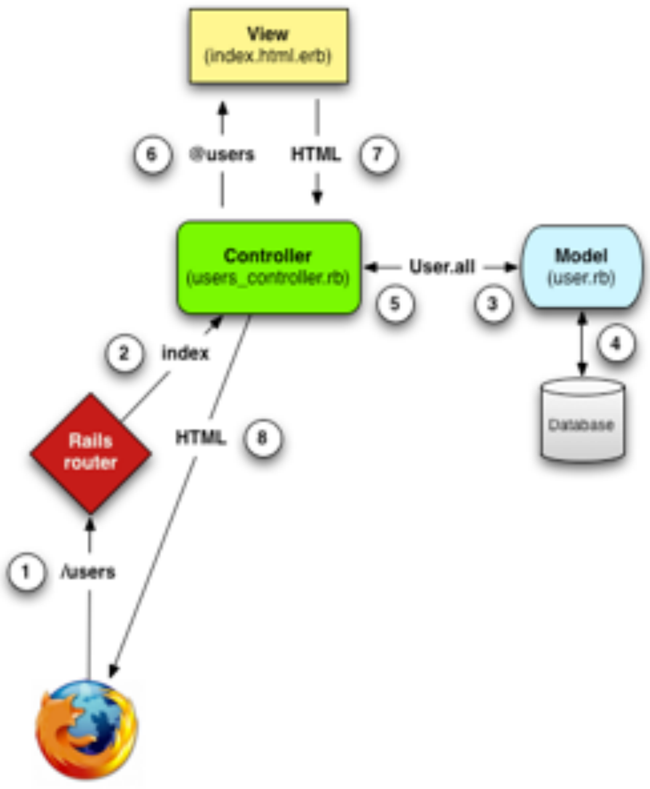
Rails Basics:

Model/View/Controller (MVC)

MODEL: This is the object that stores data for a particular instance in the database. Ruby is an Object Oriented Language. The Model is our object that we can the create instances of. The object contains fields (our database columns) much the same way you would declare instance variables in a class for java or C++.

VIEW: The View is exactly what it sounds like. It is what the user “views” when they are in the web browser. The View contains all the HTML required to define the appearance of the web page. We use rails syntax (see next page) in order to insert information from our controller into the View, or to use some kind of ruby functionality such as loops or conditionals.

CONTROLLER: The Controller is the layer that sits between the View and Model. It directs the traffic from the database to the view so that we can extract info from the database and then serve it up to the View to be rendered on our web page. There is a controller action defined for every view. This is where we perform all database queries and other ruby/rails work in order to create the instance variables we will be displaying in the View.



Common Rails Syntax

I. `<% some code here %>`
   A. Use this for when trying to insert ruby code in a view like a .each loop or a conditional statement
II. `<%= @some_instance_variable %>`
   A. Use this when trying to display some ruby code in a view from the controller, usually an instance variable as denoted by the @ symbol
III. The `@` symbol and instance variables
   A. The @ symbol defines and instance variable. These are variables created in a controller and accessible to the view. Not ever variable should be an instance variable as they could then be accessible in places they shouldn't be. If a variable is a local variable, meaning only being used for computations within a view or controller to create instance variables and not needed to be displayed or used elsewhere, it should not be an instance variable and can be declared without the @ symbol.

Creating Links

* On Champio we use links for two different purposes. Either as the classic sense of a link a user can click on to be brought to a new web page, or as a way of causing some action to happen in the database. This section will discuss both uses separately.

Links to a Different Webpage

1) Create the view you are trying to link to (If it does not already exist)
2) Create the controller action for the view you are trying to link to (If it does not already exist)
3) Add the route to config/routes.rb
   - `get`$_1$ `'users/view_name'`$_2$ `=> 'controller_name#action_name'`$_3$
     1. HTTP action you want the route to perform. This should be a GET request for linking to other page as we are "getting" the info we want to display on that page from the database
     2. The name of the path you want for the page. This should be the path of the view without the file extension
     3. The controller action you would like to serve up this view. This should be the controller name, i.e. 'users' followed by the action name you created for it separated by the '#' symbol
4) Add the link to the view (2 options)
   1) Text link
      - `<%= link_to 'Text to Display'`$_1$`, action_name_path`$_2$`(:param_name => param_value)`$_3$ `%>`
        1. Whatever text you would like to display as the link
        2. The name of the route path- do `rake routes` in the terminal to get a list of all route names
        3. OPTIONAL: params to send to the controller. Accessible from the controller as `params[:param_name]`
   2) Image or html/css link
      - `<%= link_to action_name_path(:param_name => param_value) do %>`

//some image or html/CSS here

`<% end %>`
* same explanation as text link

Links to Cause Database Action

1) Create the controller action for whatever you would like to update or create in the database
2) Add the route to config/routes.rb
   - `post`4 `'users/view_name'`5 `=> 'controller_name#action_name'`6
   4. HTTP action you want the route to perform. This should be a POST request for updating the DB as we are “posting” the info we want to update or create in the database
   5. The name of the path you want for the action
   6. The controller action doing the DB update. This should be the controller name, i.e. ‘users’ followed by the action name you created for it separated by the ‘#’ symbol
3) Add the link to the view (2 options)
   1) Text link
      - `<%= link_to 'Text to Display'`1`, action_name_path`2`(:param_name => param_value)`3`, :method => post`4`, :remote => true`5`, :format => 'js'`6 `%>`
      1. Whatever text you would like to display as the link
      2. The name of the route path- do `rake routes` in the terminal to get a list of all route names
      3. OPTIONAL: params to send to the controller. Accessible from the controller as `params[:param_name]`
      4. We set method to post to use the post route we put in routes.rb. Default method for link_to is get
      5. OPTIONAL: add :remote => true if trying to AJAX the link. This tells the page we want to process this action remotely rather than render a new page. See next section for more on adding AJAX.
      6. OPTIONAL: Required when using :remote => true for AJAX. This states that we want to process the link with JavaScript, meaning AJAX
   2) Image or html/css link
      - `<%= link_to action_name_path(:param_name => param_value), :method => post, :remote => true, :format => 'js' do %>`
        //some image or html/CSS here

        `<% end %>`
        * same explanations and options as text link

Adding AJAX

Updating the Controller to respond to AJAX

1) Add the respond to block to the controller action

```
def controller_action
     //whatever your controller does

     respond_to do |format|
          format.js
     end
end
```

The `respond_to` block tells the controller what format of requests we would like it to be able to process. In this case we are telling it we would like it to be able to process all requests coming in in the form of JavaScript. This is the format of AJAX requests.
*note we can also specify `format.html` if we want this to also respond to standard HTML requests

---

Creating AJAX Response File

1) Create the js.erb file.
   - This should be controller_action_name.js.erb where “controller_action_name” is replaced by the name of the controller action you just added the respond_to block to
   - This file should be placed in the same directory as the view it will be rendered in, AKA the view that the AJAX controller action is being called from
2) We generally use this file to either create an alert, or update some HTML element. Here’s how:
   1) Create an alert
      `newAlert("", “The message", false);`
   2) Update some HTML element
      `$(‘#html_element_id’).html("<%= j (render some_partial) %>”);`

Adding/Changing Fields in a Model

* Must enter command `rake db:migrate` in terminal after saving changes to migration for the changes to be reflected in the database

---

Generating a Rails Migration

1) Open a terminal and enter the command rails generate migration SomeNameOfMigration
   - Generally the migration name is either AddNameOfFieldToNameOfTable
   - i.e. AddUserIDToIdeas
2) Open the newly generated migration file

Adding a Field

Inside the change method in your newly created migration:

```
def change
     add_column :table_name₁, :field_name₂, :field_type₃, default₄:
some_value
end
```

1. Name of the table to add the field to. i.e. :users, :ideas, :successes, etc.
2. Whatever you want the name of the field to be
3. Types include: :integer, :boolean, :string, :text. There are more types you can lookup online
4. OPTIONAL: only use this if there needs to be a default, such as not being able to have the field be nil, or boolean types needing to be defaulted to true or false upon creation

---

Changing a Field

```
def change
      change_column :table_name[1], :old_field_name[2], :new_field_name[3],
      :new_field_type[4], default[5]: some_value
end
```

1. Name of the table to add the field to. i.e. :users, :ideas, :successes, etc.
2. Whatever the name of the field you would like to change is
3. The new name you want for the field. It can be the same name
4. The new type of the field. Types include: :integer, :boolean, :string, :text. There are more types you can lookup online. It can be the same old type
5. OPTIONAL: only use this if there needs to be a default, such as not being able to have the field be nil, or boolean types needing to be defaulted to true or false upon creation

Common Errors

- method missing error
  - Check for typos wherever you called the method
  - Check if the method actually exists. It must be defined in the model file for the type of model you are trying to call or be an existing ruby or rails method
- format error
  - This means the controller action does not respond to the format you are making a request to it with
  - Check that you added the `respond_to` block to the controller and indicated `format.js` for AJAX
- Missing template errror
  - There is no corresponding view for the controller you are linking to. Either make the view or if this is an AJAX link you likely forgot to specify `:remote => true` and `:format => ‘js’` in the `link_to` definition

Local Testing Setup Tips

- If you are testing file or image uploads make sure you are running a Redis Server instance
  - This can be done by opening a terminal and entering the command `redis-server`
  - Note you must do this before starting your rails server
- Be sure to migrate whenever you pull