Tutorial

Preface

Concept: Rails Components

Before we create our next Rails application, lets dive a little deeper into the core components of Rails we've already used without knowing it.

Action Pack

Action Pack is a single gem that contains Action Controller, Action View and Action Dispatch. The “VC” part of “MVC”.

Action Controller

Action Controller is the component that manages the controllers in a Rails application. The Action Controller framework processes incoming requests to a Rails application, extracts parameters, and dispatches them to the intended action. Services provided by Action Controller include session management, template rendering, and redirect management.

Action View

Action View manages the views of your Rails application. It can create both HTML and XML output by default. Action View manages rendering templates, including nested and partial templates, and includes built-in AJAX support.

Action Dispatch

Action Dispatch handles routing of web requests and dispatches them as you want, either to your application or any other Rack application.

Active Model

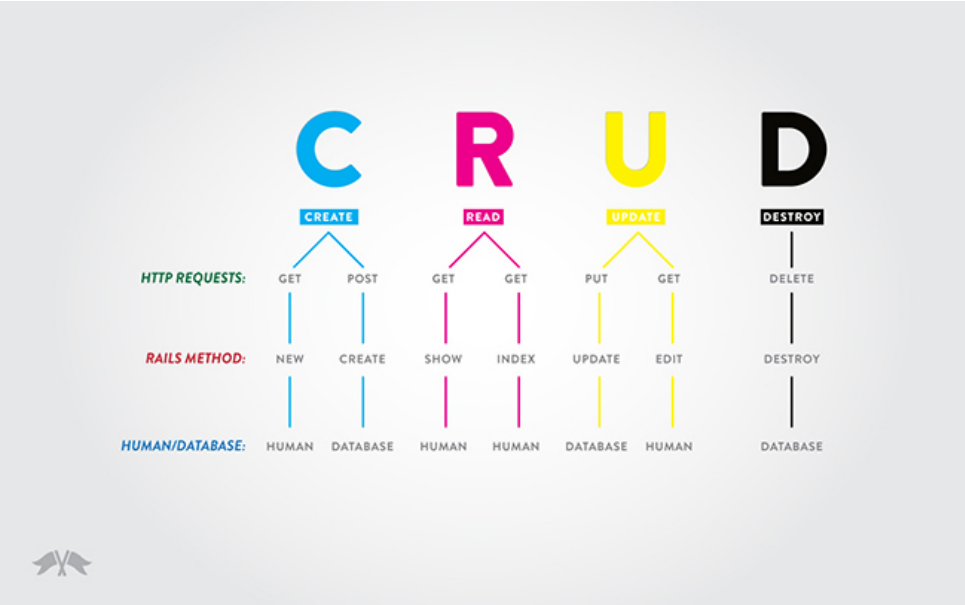
Active Model provides a defined interface between the Action Pack gem services and Object Relationship Mapping gems such as Active Record. Active Model allows Rails to utilize other ORM frameworks in place of Active Record if your application needs this.

Active Record

Active Record is the base for the models in a Rails application. It provides database independence, basic CRUD (create, read, update, delete) functionality, advanced finding capabilities, and the ability to relate models to one another, among other services.

Active Resource

Active Resource provides a framework for managing the connection between business objects and RESTful web services. It implements a way to map web-based resources to local objects with CRUD semantics.

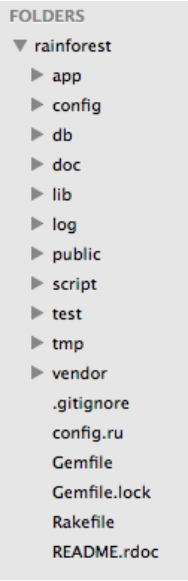


1. Initial Commit

Now that we have some context, lets create a new Rails application by running the following on your command line:

```
$ rails new rainforest
```

To reiterate, this creates a folder called `rainforest` and puts all of the Rails files inside it. If you open this folder in Sublime you should see all of the files and the files in folders on the left-hand side:



Let's look more closely the sub-directories of the important directories using our previous workflow pattern RCAV, now with Models involved. We'll call this expanded workflow MRCAV, which stands for Model, Routes, Controller, Actions, and Views:

`app/models`

A model represents the information (data) of the application and the rules to manipulate that data. In the case of Rails, models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, each table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models.

`config/routes.rb`

This is your application's routing file which holds entries in a special DSL (domain-specific language) that tells Rails how to connect incoming requests to controllers and actions.

`app/controllers`

Controllers provide the "glue" between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

`app/views`

Views represent the user interface of your application. In Rails, views are often HTML files with embedded Ruby code that perform tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application.

We've created a new Rails app. Don't forget to initialize Git, stage all of your files, and make your first commit:

```
$ git init
$ git add --all
$ git commit -m "Initial commit"
```

Once we've made this commit, we want to create a remote repo so that we can track our changes. Go to github.com and create a repository. Feel free to call it something descriptive, but it doesn't have to be `rainforest` if you don't want it to be.

```
$ git remote add origin https://github.com/[YOUR-USERNAME]/[YOUR-REPO-NAME].git
$ git push -u origin master
```

What we covered in this commit

- The `app` directory
- The `routes.rb` file

2. Generating the product model

As you go through the below processes, make sure you are checking everything you do with your browser. Start your server and browse to the appropriate urls to check that everything is working as planned.

Concept: Rails Command Line

- $ rails new app_name
- $ rails server (or rails s)
- $ rails generate (or rails g)
- $ rails console (or rails c)
- $ rails dbconsole
- $ rake

There are a number of rails commands you will frequently use throughout the development process. You have already used many of them throughout the development of Photogur and we'd like to bring the commonly used shortcuts (also known as aliases) to your attention. We will be using these shortcuts from now on. If you would like to read more about these, please check out Layouts and Rendering in Rails: 3.4 Using Partials.

Lets use our first shortcut now and start up our server.

```
$ rails s
```

Don't forget to delete the default `public/index.html` page.

Generate your product model with the attributes name, description, and price_in_cents with the string, text and integer datatypes respectively.

```
$ rails g model Product name:string description:text price_in_cents:integer --no-test-framework
```

Concept: Migration

A migration is a convenient way for you to alter your database in a structured and organized manner.

The above command created the product model `app/models/product.rb` and a migration that looks like the following:

`db/migrate/[TIMESTAMP]_create_products.rb`

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description
      t.integer :price_in_cents

      t.timestamps
    end
  end
end
```

This migration represents a table with the attributes name, description and price_in_cents with the database column types of string, text and integer respectively. You can read more about the supported database column types in Migrations: 1 Anatomy of a Migration

Concept: Rake

Try running:

```
$ rake --tasks
```

Rake is Ruby Make, a standalone Ruby utility that replaces the Unix utility 'make', and uses a 'Rakefile' and .rake files to build up a list of tasks. In Rails, Rake is used for common administration tasks, especially sophisticated ones that build off of each other.

After you generate any migration (this happens when you generate a model or a standalone migration), you will need to run `rake db:migrate` to run the up or change method for all the migrations that have not yet been run. In this case, only having one migration, Active Record will create a table in your database and update your db/schema.rb file to match the structure of your database. It will run these migrations in order based on the date of the migration. If there are no such migrations, it exits.

```
$ rake db:migrate
$ git status
$ git add --all
$ git commit -m "Generating the product model"
$ git push
```

What we covered in this commit

- Rails Command Line
- Migrations
- Rake

3. Generating the products controller and views

Remember to delete `public/index.html` and write the corresponding route to make your http://localhost:3000 view and http://localhost:3000/products view match.

```
$ rails g controller products index show new edit --no-test-framework
```

This command makes sure a bunch of directories are in our application, and created a controller file, a products folder, a few view files, a helper for the views, a javascript file, a stylesheet file and wrote a number of routes in `config/routes.rb`. For now, lets focus on the routes:

`config/routes.rb` *

```
Rainforest::Application.routes.draw do
  get "products/index"
  get "products/show"
  get "products/new"
  get "products/edit"
end
```

Try running:

```
$ rake routes
```

As expected, we get the four specified routes. If we are creating a typical CRUD (create-read-update-delete) application we should replace all of these routes with `resources :products` to have Rails generate the 7 RESTful routes. Lets do that now.

`config/routes.rb`

```
Rainforest::Application.routes.draw do
  resources :products
end
```

Try running:

```
$ rake routes
```

Notice the difference?

What you may be thinking is we now have 7 routes for 4 actions in our controller? Try writing these 7 RESTful routes (the golden 7) by hand. Writing each method out will make you focus on what exactly is going on in each action and most likely formulate a new set of questions about what is going on. This is a good thing.

Using Photogur as a reference, write the below actions by hand, and again, keep thinking about what each action is actually doing. Find out what all of the class methods (.all, .find, .new) and instance methods (.create, .save, .update_attributes, .destroy) are doing and exactly how they work.

Did I mention how beneficial it is to write these by hand?

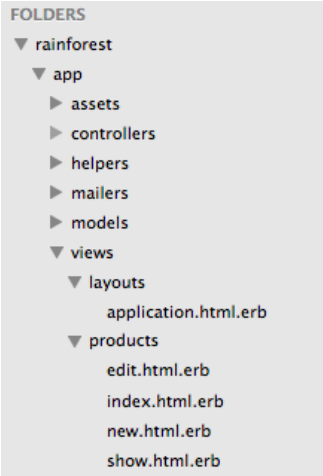
`app/controllers/products_controller.rb`

```
class ProductsController < ApplicationController
  def index
    @products = Product.all
  end

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

  def new
    @product = Product.new
  end

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

  def create
```

```
    @product = Product.new(params[:product])

    if @product.save
      redirect_to products_url
    else
      render :new
    end
  end

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

    if @product.update_attributes(params[:product])
      redirect_to product_path(@product)
    else
      render :edit
    end
  end

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

Now lets make the corresponding views functional. You will notice we have a ** `views/products/` ** directory containing the files `index.html.erb`, `show.html.erb`, `new.html.erb` and `edit.html.erb`. You can use Photogur as reference.

- Note: You will have to create a product to test if the show and edit pages work.



Write the appropriate html to make them function as expected. Don't forget the form partial you can use for both the new and edit actions.

Partials

Partial templates – usually just called “partials” – are another device for breaking apart the rendering process into more manageable chunks. With a partial, you can move the code for rendering a particular piece of a response to its own file. - Rails Guides

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

```
<h1>New product</h1>

<%= render 'form' %>

<%= link_to 'Back', products_path %>
```

`app/views/products/_form.html.erb`

```
<%= form_for(@product) do |f| %>
```

```
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :price_in_cents %><br />
    <%= f.number_field :price_in_cents %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Phew! We're finished creating the correct actions for the products controller and views.

```
$ git status
$ git add --all
$ git commit -m "Generating the products controller and views"
$ git push
```

What we covered in this commit

- Practice writing RESTful actions
- Practice writing views

4. Refactoring the products controller to respond to JSON

Concept: Creating a Basic API

Below is an example respond_to block for the index action.

`controllers/products_controller.rb`

```
 def index
    @products = Product.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render json: @products }
    end
  end
```

The respond_to block handles both HTML and JSON calls to this action. If you browse to http://localhost:3000/products.json, you'll see a JSON containing all of the products. The HTML format looks for a view in app/views/products/ with a name that corresponds to the action name. Rails makes all of the instance variables from the action available to the view. To refactor all of our actions to respond to JSON we would add respond_to blocks to each (as shown below).

`controllers/products_controller.rb`

```
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index
    @products = Product.all

    respond_to do |format|
```

```
    format.html # index.html.erb
    format.json { render json: @products }
  end
end

# GET /products/1
# GET /products/1.json
def show
  @product = Product.find(params[:id])

  respond_to do |format|
    format.html # show.html.erb
    format.json { render json: @product }
  end
end

# GET /products/new
# GET /products/new.json
def new
  @product = Product.new

  respond_to do |format|
    format.html # new.html.erb
    format.json { render json: @product }
  end
end

# GET /products/1/edit
def edit
  @product = Product.find(params[:id])
end

# POST /products
# POST /products.json
def create
  @product = Product.new(params[:product])

  respond_to do |format|
    if @product.save
      format.html { redirect_to @product, notice: 'Product was successfully created.' }
      format.json { render json: @product, status: :created, location: @product }
    else
      format.html { render action: "new" }
      format.json { render json: @product.errors, status: :unprocessable_entity }
    end
  end
end

# PUT /products/1
# PUT /products/1.json
def update
  @product = Product.find(params[:id])

  respond_to do |format|
    if @product.update_attributes(params[:product])
      format.html { redirect_to @product, notice: 'Product was successfully updated.' }
      format.json { head :no_content }
    else
      format.html { render action: "edit" }
      format.json { render json: @product.errors, status: :unprocessable_entity }
    end
  end
end

# DELETE /products/1
# DELETE /products/1.json
def destroy
  @product = Product.find(params[:id])
  @product.destroy
```

```
    respond_to do |format|
      format.html { redirect_to products_url }
      format.json { head :no_content }
    end
  end
end
```

Now that we've added the respond_to blocks to all of the products controller actions, we can can browse to any of our application's urls and add `.json` to the url and we'll be returned JSON. This clearly isn't the preferable presentation for a human, but for another application its pretty darn sexy. Let's commit our changes.

```
$ git status
$ git add --all
$ git commit -m "Refactoring the products controller to respond to JSON"
$ git push
```

What we covered in this commit

- Creating a Basic API

5. Writing product validations

Concept: Validations

Validations are used to ensure that only valid data is saved into your database. For example, it may be important to your application to ensure that every user provides a valid email address and mailing address.

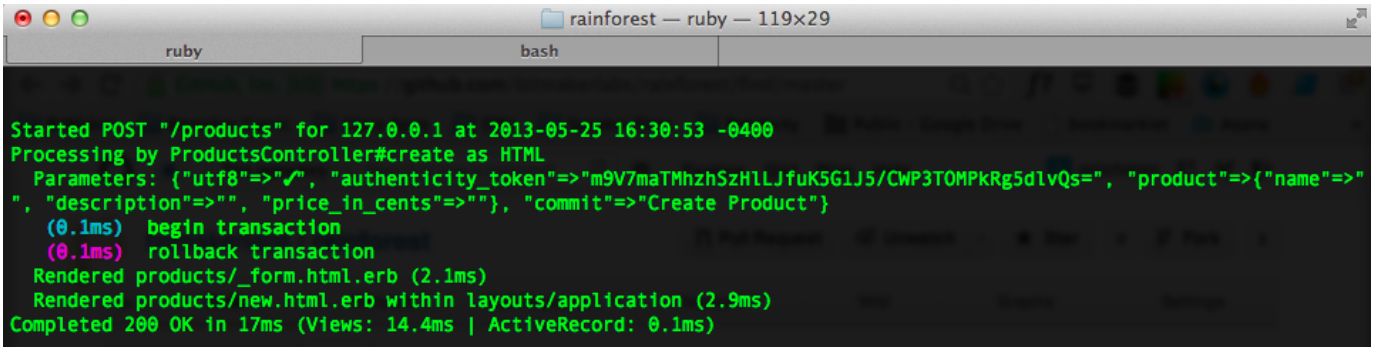
To understand when validations are run, it is important to know there are two kinds of Active Record objects: those that correspond to a row inside your database and those that do not. When you create a fresh object, for example using the new method, that object does not belong to the database yet. Once you call `save` upon that object it will be saved into the appropriate database table and your validation. - Rails guides

1. When you create a product, both the description and name have to be present
2. When you create a product, the price_in_cents must be an integer

`app/models/product.rb`

```
class Product < ActiveRecord::Base
  attr_accessible :description, :name, :price_in_cents

  validates :description, :name, :presence => true
  validates :price_in_cents, :numericality => {:only_integer => true}
end
```

This will prevent users from trying to save undesirable information to the database. Now that you have written proper validations, try creating a product with no information. What happens?



We can see that our validations are working, but it's a little tough to tell what is actually happening as a user. We'll fix that in our next commit, for now lets commit our validations.

```
$ git status
$ git add --all
$ git commit -m "Writing product validations"
$ git push
```

What we covered in this commit

- Validations

6. Displaying validation errors on the product form

Concept: Validation Errors

Rails provides built-in helpers to display the error messages of your models in your view templates. We'll need to refactor the `app/views/products/_form.html.erb` to display appropriate errors to the user. Feel to read about it further in Active Record Validations and Callbacks: 7 Working with Validation Errors

The default pattern to display errors in in a form can be found below. The best practice is to place it in the `form_for` block, however it can be placed anywhere on the page.

`app/views/products/_form.html.erb`

```
<%= form_for(@product) do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@product.errors.count, "error") %> prohibited this product from being saved:</h2>

      <ul>
      <% @product.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  .
  .
  .
```

After we've added the ability to display error messages, lets commit our changes..

```
$ git status
$ git add --all
$ git commit -m "Displaying validation errors on the product form"
$ git push
```

What we covered in this commit

- Validation Errors

7. Displaying price in dollars

Concept: Defining Class and Instance Methods

Remember all of those Ruby games where we defined various class and instance methods? Nothing has changed now that we're using Rails.

Lets quickly go through some of the things that you have seen so far.

1. user (lowercase u) is an object, and you can call instance methods like `user.save`.
2. User (capital U) is a class method - you don't need an object to call it like `User.find`. ActiveRecord adds both instance and class methods to your model.

Thanks to the blog post from Intermediate Rails: Understanding Models, Views, and Controllers

Class and Instance Methods

In your product model, write a method called formatted price that converts your price in cents to a string that is a price in dollars

- For assurance, you can use a method called sprintf to round your number to two decimals.

`app/models/product.rb`

```
class Product < ActiveRecord::Base
  .
  .
  .
  def formatted_price
    price_in_dollars = price_in_cents.to_f / 100
    sprintf("%.2f", price_in_dollars)
  end
end
```

In your `products/show.html.erb` ** and ** `products/index.html.erb` call "product.formatted_price" to display each products correct price. It's pretty basic at this point so I'll let you figure that one out.

It was only a brief change, but definitely commit worthy. Lets practice those commit skills and make another.

```
$ git status
$ git add --all
$ git commit -m "Displaying price in dollars"
$ git push
```

What we covered in this commit

- Writing methods in the model

8. Creating a user model, controller and view

Before we create a user model and start signing up million of users, lets walk through an issue almost every web application has to deal with - authentication.

Concept: Authentication

Be careful not to confuse *authentication* with *authorization*. Authentication is the process of verifying that "you are who you say you are", authorization is the process of verifying that "you are permitted to do what you are trying to do". Authorization thus presupposes authentication.

For example, a client showing proper identification credentials to a bank teller is asking to be authenticated that he really is the one whose identification he is showing. A client whose authentication request is approved becomes authorized to access the accounts of that account holder, but no others. - Wikipedia

To do this we will need to be familiar with the `bcrypt` gem and the `has_secure_password` method it provides us.

Bcrypt

bcrypt is a hashing algorithm designed by Niels PRovos and David Mazia. A hash algorithm takes a chunk of data (e.g. your user's password) and creates a "digital fingerprint", or hash, of it. This process is not reversible, so you can only go from the password to the hash and not vice versa. If you would like to read the documentation, check out Rubyforge - Bcrypt.

has_secure_password is a method to set and authenticate against a BCrypt password. To use has_secure_password we are require to have a password_digest attribute in our user model, but use the password and password_confirmation fields in our view.

Lets get BCrypt to work. First we uncomment the `bcrypt-ruby` gem in our `Gemfile`.

`Gemfile`

```
# To use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

Any time we change our Gemfile, we will need to run:

```
$ bundle install
```

Use a model generator to generate your user model with the attributes email and password_digest both with the string data column type.

```
$ rails g model user email:string password_digest:string --no-test-framework
```

The important thing here is that we name the field that will store passwords in our database "password_digest".

Next we'll need to add has_secure_password to the User model and as good practice, a validation requiring a password when a user is created.

- Note: The attr_accessible symbols are `:password` and `:password_confirmation`, not :password_digest. This is because the form will reflect these attributes, only when the data gets saved to the model does it become "password_digest".

`models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create
end
```

After a careful check of our migration, we can run:

```
$ rake db:migrate
```

Next we add the appropriate routes to let us create new users. The `only` keyword restricts the routes to only the actions specified in the array.

`config/routes.rb`

```
Rainforest::Application.routes.draw do
  resources :products
  resources :users, :only => [:new, :create]
end
```

Use a controller generator to generate your users controller with the actions new and create.

```
$ rails g controller users new create --no-test-framework
```

We now fill in the appropriate actions in the users controller.

`app/controllers/users_controllers.rb`

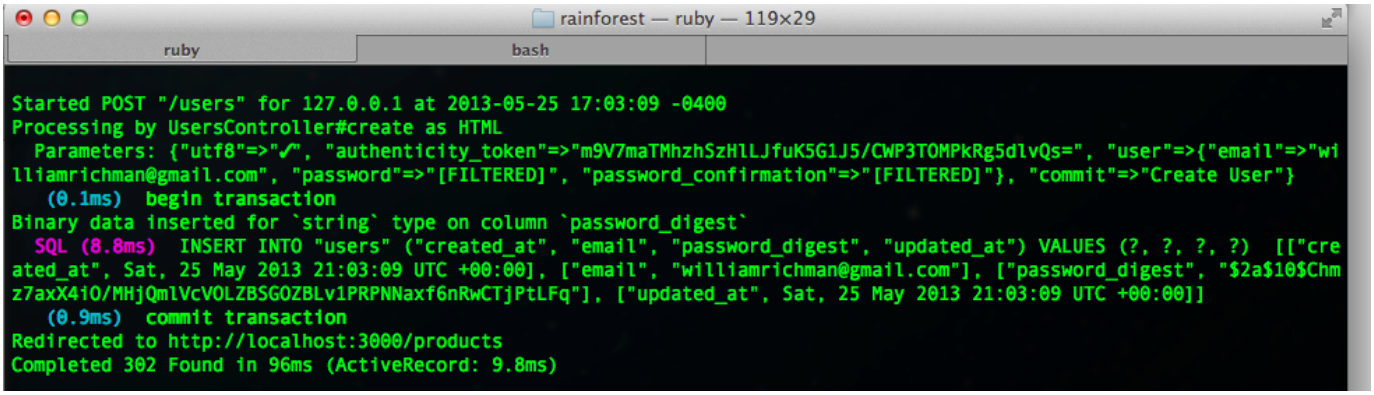
```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to products_url, :notice => "Signed up!"
    else
      render "new"
    end
  end
end
```

Now that we have the appropriate actions in the users controller, lets create the users form.

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

```
<h1>Sign Up</h1>

<%= form_for @user do |f| %>
  <% if @user.errors.any? %>
    <div class="error_messages">
      <h2>Form is invalid</h2>
      <ul>
        <% for message in @user.errors.full_messages %>
          <li><%= message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <div class="field">
    <%= f.label :email %><br/>
    <%= f.text_field :email %>
  </div>
  <div class="field">
    <%= f.label :password %><br/>
    <%= f.password_field :password %>
  </div>
  <div class="field">
    <%= f.label :password_confirmation %><br/>
    <%= f.password_field :password_confirmation %>
  </div>
  <div class="actions">
<% end %>
```

You'll want to test this by creating a user, at this point you will likely need to restart your server. Find the terminal session and press "CMD + C" to stop your server. Start it back up again to create a user. Check out the server log to see what is actually happening in the background.



You can check that you have created a user by going into your console and querying the database.

```
$ rails c
$ User.all or User.first
```

Don't forget to delete `app/views/users/create.html.erb`.

We can now add authenticated, validated users to our application! Let's commit our changes and stretch out bask in all our glory for... 30 seconds sound fair? Okay back to programming!

```
$ git status
$ git add --all
$ git commit -m "Creating a user model, controller and view"
$ git push
```

What we covered in this commit

- Authentication
- Bcrypt

9. Creating log in functionality

We want to create log in functionality for our app. This requires the use of "sessions", which will be explained later in this commit. Just like we did for the user routes, we can specify which routes we would like Rails to create by using `:only`

`config/routes.rb`

```
Rainforest::Application.routes.draw do
  .
  .
  .
  resources :sessions, :only => [:new, :create, :destroy]
end
```

Time to create out sessions controller and views. We don't need a model here because there is nothing saved to a database when logging in.

```
$ rails g controller sessions new create destroy --no-test-framework
```

Our sessions controller looks a little different than we have been used to so far. To create a log in, there is no need to instantiate a new 'session' by creating a new instance. When we hit the 'submit' button, our create action is triggered. There are a couple new things in our create action:

`app/controller/sessions_controller.rb`

```
class SessionsController < ApplicationController
  def new
  end

  def create
    user = User.find_by_email(params[:email])
    if user && user.authenticate(params[:password])
      session[:user_id] = user.id
      redirect_to products_url, :notice => "Logged in!"
    else
      render "new"
    end
  end

  def destroy
    session[:user_id] = nil
    redirect_to products_url, :notice => "Logged out!"
  end
end
```

1. We find a user by the email typed in the email input field and assign it to the variable 'user'
2. We check if that user exists and that it can be authenticated with the password typed in the password input field
3. If both of those evaluate to 'true', we create a key-value pair in the session hash. ':user_id' is the key and the user's id is the value
4. The session key has been assigned, the user is redirected to the products index page

Now to create the associated session form. Take note that we are using a `form_tag` and not a `form_for` here. This is because there is no model associated with a session.

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

```
<h1>Log in</h1>

<%= form_tag sessions_path do %>
  <div class="field">
    <%= label_tag :email %><br/>
    <%= text_field_tag :email, params[:email] %>
  </div>
  <div class="field">
    <%= label_tag :password %><br/>
    <%= password_field_tag :password %>
  </div>
  <div class="actions"><%= submit_tag "Log in" %></div>
<% end %>
```

We can check that our login works by going to "http://localhost:3000/sessions/new" and logging in. If we get directed to the products index page, it worked!

```
$ git status
$ git add --all
$ git commit -m "Creating log in functionality"
$ git push
```

What we covered in this commit

- Sessions

10. Adding flash alerts and notices

Concept: Flash Notices and Errors

The flash is a special part of the session which is cleared with each request. This means that values stored there will only be available in the next request, which is useful for storing error messages etc. It is accessed in much the same way as the session, like a hash. Feel free to read into more detail (The Flash Section 4.2)[http://guides.rubyonrails.org/action_controller_overview.html#the-flash]

`app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController
  .
  .
  .
  def create
    user = User.find_by_email(params[:email])
    if user && user.authenticate(params[:password])
      session[:user_id] = user.id
      redirect_to products_url, :notice => "Logged in!"
    else
      flash.now[:alert] = "Invalid email or password"
      render "new"
    end
```

```
  end
  .
  .
  .
end
```

We can write some code in the application view to display notices or alerts whenever available. The application view is appropriate because we only have to write this code once and it can be reused across all views.

`app/views/layouts/application.html.erb`

```
  .
  .
  .
  <body>
    <% if flash[:notice] %>
      <p id="notice"><%= flash[:notice] %></p>
    <% end %>
    <% if flash[:alert] %>
      <p id="alert"><%= flash[:alert] %></p>
    <% end %>

    <%= yield %>
  </body>
</html>
```

Don't forget to commit your changes.

```
$ git status
$ git add --all
$ git commit -m "Adding flash alerts and notices"
$ git push
```

What we covered in this commit

- Flash notices and errors

11. Adding logic to display 'Signed in as' or 'Login or Sign up' in the application view

Concept: Helper Methods

Helper methods are are use to share methods defined in the controller with the view. This is use for any method that you need to access from both controllers and helpers/views. Typical helper methods are not available in controllers.

Create a current_user helper method to show the log in status of the user. If the user is logged in we want to display "Signed in as [USERNAME]" or else display "Login or Sign up"

`app/controllers/application_controller.rb` *

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  private

  def current_user
    @current_user ||= User.find(session[:user_id]) if session[:user_id]
  end
```

```
  helper_method :current_user
end
```

`app/views/layouts/application.html.erb`

```
  <body>
  .
  .
  .
  <%= yield %>

    <p class="userinfo">
      <% if current_user %>
        Signed in as <%= current_user.email %>. <%= link_to "Log out", session_path("current"), :method => :delete %>
      <% else %>
        <%= link_to "Log in", new_session_path %> or <%= link_to "Sign up", new_user_path %>
      <% end %>
    </p>
  </body>
</html>
```

Lets commit these changes.

```
$ git status
$ git add --all
$ git commit -m "Adding logic to display 'Signed in as' or 'Login or Sign up' in the application view"
$ git push
```

What we covered in this commit

- Helper methods

12. Creating review model and associating the user, product and review models

Concept: Associations

In Rails, an association is a connection between two Active Record models. Associations are implemented using macro-style calls, so that you can declaratively add features to your models. For example, by declaring that one model belongs_to another, you instruct Rails to maintain Primary Key–Foreign Key information between instances of the two models, and you also get a number of utility methods added to your model. Rails supports six types of associations:

Associations come in 6 different forms:

- belongs_to
- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

Our app will use the "has_many :through" relationship, but we go through all of the smallers steps to get there. Lets start with figuring out how a product and review are related. Keep reading until you see the bash command to create the corresponding models before writing any code.

Does each product have one review? That doesn't make sense because that would limit the perspective shared on a product. Does each product have many reviews? Yes that makes sense, so lets write that now.

`app/models/product.rb`

```
class Product < ActiveRecord::Base
  has_many :reviews
end
```

Does each review have many products? That doesn't make sense, why would you create one review for many products? That wouldn't be very specific information. Does each review have one product? It does, but there is a subtle difference, it "belongs to" a product. The distinction does two things, it establishes the direction of the relationship and it specifies by convention that review must have a foreign key.

A foreign key is a field in a relational table that matches a candidate key of another table. The foreign key can be used to cross-reference tables. -Wikipedia

The convention in Rails is that the foreign key is the referenced model name followed by an "_id". For example, the foreign key field required to reference the product model from the review model would be "product_id". The review model would look as follows.

`app/models/review.rb`

```
class Review < ActiveRecord::Base
  attr_accessible :comment, :product_id
  belongs_to :product
end
```

Given the relationship is the same for the review model and user model. We would repeat the same process.

`app/models/product.rb`

```
class Product < ActiveRecord::Base
  .
  .
  .
  has_many :reviews
  has_many :users :through => :reviews
end

class Review < ActiveRecord::Base
  .
  .
  .
  belongs_to :user
  belongs_to :product
end

class Users < ActiveRecord::Base
  ...
  has_many :reviews
  has_many :products, :through => :reviews
end
```

TBD - Picture of the associated tables

Lets create the correct review model and attributes.

```
$ rails g model review comment:text product_id:integer user_id:integer --no-test-framework
```

Using the associations explanation above, modify your models so that all of the relationships are correct.

Now that our models have been created and correctly associated lets plan out how we would like to display our reviews. Should we display all of the reviews in a list? Should they be intermingled with all of the products? Should we be able to see each review individually?

It would probably be best if a user could view all of the reviews associated with a product on the product show page. A good way to do that, is to follow RESTful convention and use a concept called "Nested Resources" to create well structured routes.

What we covered in this commit

- Associations
  - belongs_to
  - has_many
  - has_many, :through

13. Writing the review routes, controllers and views

Concept: Nested Resources

Nested routes allow you to capture the below relationship in your routing. In this case, you could include this route declaration:

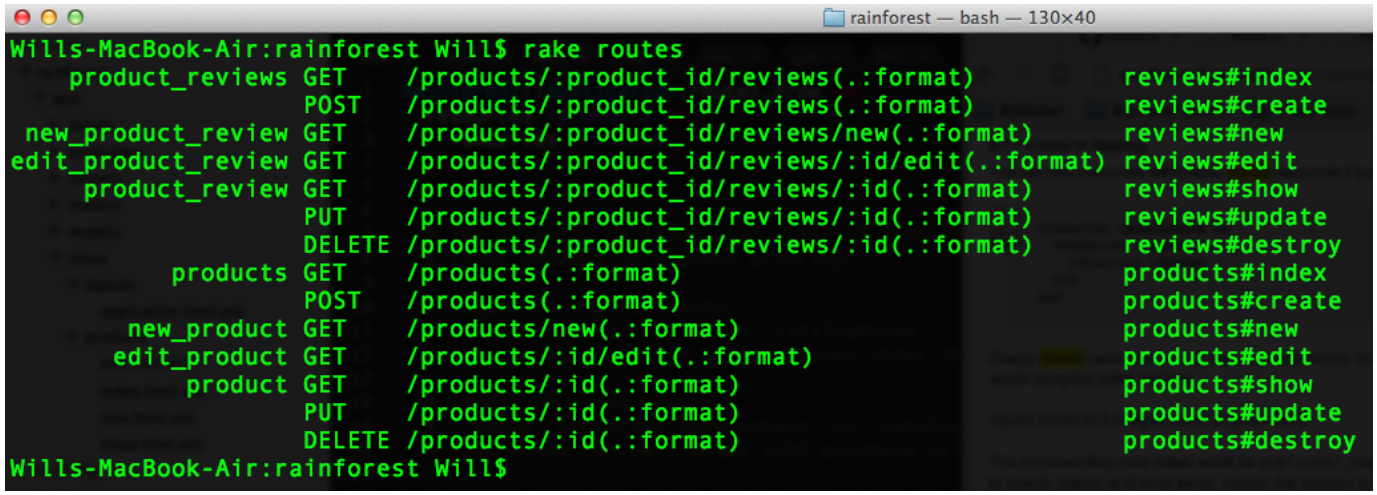
`config/routes.rb`

```
Rainforest::Application.routes.draw do
  .
  .
  .
  resources :products do
    resources :reviews, :except => [:index]
  end
end
```

Now try the following command:

```
$ rake routes
```

It should display the below in your terminal.



This routing convention lets us easily find a product by its id and either a particular review or all of the reviews associated with it. It may be a little fuzzy right now, but it will make sense by the end of this commit.

```
$ rails g controller reviews show new edit --no-test-framework
```

Concept: Filters

In order to associate a review with a product, we will need to find a product by its ID. We can use a method called a 'before_filter' in our reviews controller to run before every action. Lets call a load_product method that retrieves the appropriate product so a new review can be associated to it. If you would like to read more, try Action Controller: 7 Filters

`app/controllers/reviews_controller.rb`

```
class ReviewsController < ApplicationController
  before_filter :load_product

  def show
    @review = Review.find(params[:id])
  end

  def create
    @review = @product.reviews.build(params[:review])
    # Check out this article on [.build](http://stackoverflow.com/questions/783584/ruby-on-rails-how-do-i-use-the-active-record-bu
    # You could use a longer alternate syntax if it makes more sense to you
    #
    # @review = Review.new(
    #   :comment    => params[:review][:comment],
    #   :product_id => @product.id,
    #   :user_id    => current_user.id
    # )

    if @review.save
      redirect_to products_path, notice: 'Review created successfully'
    else
      render :action => :show
    end
  end

  def destroy
    @review = Review.find(params[:id])
    @review.destroy
  end

  private

  def load_product
    @product = Product.find(params[:product_id])
  end
end
```

`app/controllers/products_controller.rb`

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

  if current_user
    @review = @product.reviews.build
  end

  respond_to do |format|
    format.html # show.html.erb
    format.json { render json: @product }
  end
end
```

Now that we have created the appropriate actions in the reviews controller, lets set up the ability to view reviews on the product show page.

`app/views/products/show.html.erb`

```
.
.
.
<% if current_user %>

  <h3>Reviews</h3>

  <p>Reviews for <%= @product.name %></p>
```

```
  <% @product.reviews.each do |review| %>
    <p>
      <%= review.comment %>
      <br>
      <% if review.user != nil %>
        <em> by <%= review.user.email %></em>
        Added on: <%= review.created_at %>
      <% end %>
    </p>
  <% end %>

<% end %>
.
.
.
```

Since we are required to associate a review with a product when it one is created, we will need to use a nested form to build that association. Luckily Rails' `form_for` makes it incredibly easy for us to do this. We pass an array to form_for like so: `<%= form_for([@product, @review]) do |f| %>`.

`app/views/products/show.html.erb`

```
.
.
.
<% if current_user %>
.
.
.
  <h4>New Review</h4>

  <%= form_for([@product, @review]) do |f| %>
    <% if @review.errors.any? %>
      <div id="error_explanation">
        <h2><%= pluralize(@review.errors.count, "error") %> prohibited this review from being saved:</h2>

        <ul>
        <% @review.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
        </ul>
      </div>
    <% end %>

    <div class="field">
      <%= f.label :comment %><br />
      <%= f.text_area :comment %>
    </div>
    <div class="actions">
      <%= f.submit %>
    </div>
  <% end %>

<% else %>
  <p>Please <%= link_to "log in", new_session_path %> to add a review.</p>
<% end %>
.
.
.
```

Stretch Assignment: Refactor the show page

Create a separate partial to display to reviews and one to create new reviews on the products show page. You will be able to use the form partial in the `app/views/reviews/edit.html.erb`.

Now that everything is functioning lets commit those changes.

```
$ git status
$ git add --all
$ git commit -m "Writing the review routes, controllers and views"
$ git push
```

What we covered in this commit

- Nested Resources
- Filters
- Nested Forms
- Practice Refactoring views

14. Adding authorization to ensure users are logged in before they create reviews

Concept: Authorization

We would like to restrict what a visitor can do without logging in. We do this by creating a method to be called before controller actions that would require a user to be logged in. Lets call this method `ensure_logged_in`.

`app/controllers/application_controller.rb`

```
class ApplicationController < ActionController::Base
  .
  .
  .

  def ensure_logged_in
    unless current_user
      flash[:alert] = "Please log in"
      redirect_to new_session_path
    end
  end
end
```

`app/controllers/reviews_controller.rb`

```
class ReviewsController < ApplicationController
  before_filter :ensure_logged_in, :only => [:edit, :create, :show, :update, :destroy]
  .
  .
  .
end
```

`app/controllers/reviews_controller.rb`

```
class ProductsController < ApplicationController
  before_filter :ensure_logged_in, :only => [:show]
  .
  .
  .
end
```

```
$ git status
$ git add --all
$ git commit -m "Adding authorization to ensure users are logged in before they create reviews"
$ git push
```

What we covered in this commit

- Authorization

15. Adding user names

We'd like to be able to display our users names to , and in order to do that we'd like to know their names. To do that we're going to add a name column to our user model.

We can use the generator command to create a migration that once run will add exactly what we need.

```
$ rails g migration AddNameToUser name:string
```

This command creates a pre configured migration with an `add_column` method that takes a table name, column name and a column data type as its parameters.

`[TIMESTAMP]_add_name_to_user.rb`

```
class AddNameToUser < ActiveRecord::Migration
  def change
    add_column :users, :name, :string
  end
end
```

Run `rake db:migrate` to modify the table. Let's make sure that a visitor inputs a name when they create a user.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation, :name
  .
  .
  .

  validates_presence_of :name

  .
  .
  .
end
```

We'll also need to modify our form to accept a user's name as input.

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

```
.
.
.
<div class="field">
  <%= f.label :name %><br/>
  <%= f.text_field :name %>
</div>
.
.
.
```

Lastly we'll need to modify the `views/layout/application.html.erb` ** and ** `views/products/show.html.erb` pages to display the user's name instead of their email.

Great job! Lets commit our changes.

```
$ git status
$ git add --all
$ git commit -m "Adding user names"
$ git push
```

What we covered in this commit

- Adding a column

16. Stretch Assignments

- User profile page
- Sort reviews by newest first
- Create categories and tags
- Add categories to products
- Create a Homepage for the application
- Add CSS to your application

Last edited by Will, 3 days ago