**Real-time web application with Ruby on Rails**

by

Jacob Tarr

This project is submitted to the Gannon University graduate faculty in

partial fulfillment for the degree Master of Science in Computer and Information Science.

Option: Web Development

Approved:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *Cannell, Jeremy C.* |  | *Xu, Frank* |
| Advising Professor in Charge of Research |  | Committee Member |
|  |  |  |
|  |  |  |
| *Yunkai, Liu* |  | *Mei-Huei, Tang* |
| Committee Member |  | ***Chair, Computer and Information Science Department*** |
|  |  |  |

Gannon University

Erie, Pennsylvania 16541

December 2014

**Acknowledgements**

*< The student thanks significant stakeholders and supporters of the research effort. Consider thanking mentors, colleagues, individuals or institutions supporting the research, family. If a explicit permission was given to reproduce or to use material in the research, those entities are named and the relevance of their permission identified and credited. >*

**Table of Contents**

**Abstract**

**1.** **Introduction**

1.1 Overview

1.2 Setup and Workflow

**2.** **Technology**

2.1 MVC Architecture

2.2 Active Record

**3.** **Application Structure**

3.1 Asset Pipeline

3.2 The Model

3.3 The Controller

3.4 The View

**4.** **Adding Real-Time to Lemur**

**5.** **Deployment to Heroku**

**Abstract**

*This project was intended to be a study of the web framework Ruby on Rails and its effectiveness in creating real-time web applications. For this reason, this project is also intended to be a “technology transfer” for future use at the Computer Science Department at Gannon University.*

# 

# 

1. **Introduction**
   1. **Overview**

Lemur, as the project is named, was built on Ruby on Rails, a web framework that utilizes MVC architecture and the “convention over configuration” principle for application building. The project is a fully-functional application that allows users to sign up and post events or mini stories. Users can also follow each other if they want to be connected with the people around them.

* 1. **Setup and Workflow**

In order to maintain a project this large, it is strongly recommended to use a robust text editor such as Sublime Text, VIM, or Atom in order to speed up development. Tools such as Emmet, syntax highlighting and other tools that these text editors come equipped with are convenient and can go a long way to make sure the development process goes as smoothly as possible.

Version control, more specifically Git, is also something that has become essential in a Rails application environment. It allows for easy collaboration between team members and can also come into importance when a developer makes a mistake and wants to go back and make revisions. The steps to getting a repository setup with Git only calls for a few simple steps:

|  |
| --- |
| git init  git add . // Adds all project files in the current directory  git commit -m "My first commit" |

The repository can then be pushed online to sites like Github and BitBucket. Lemur’s repository currently lives at https://github.com/jacobtarr/lemur.

1. **Technology**

2.1 **MVC Architecture of Rails**

Rails uses something called MVC, an abbreviation for “Model View Controller,” a modern software approach to building web applications. In the case of Rails, models store and validate data. They also talk directly to the database. The model can also make associations across entities in the databases. For instance, in the case of polymorphic associations, a user belongs to many posts, but only one post can belong to a user.

The controller talks to both the model(s) and view(s). It gives actions to the model and establishes rules on how the model can interact with the application. Additionally, the controller can also interact with the views to control the representational aspect of the application.

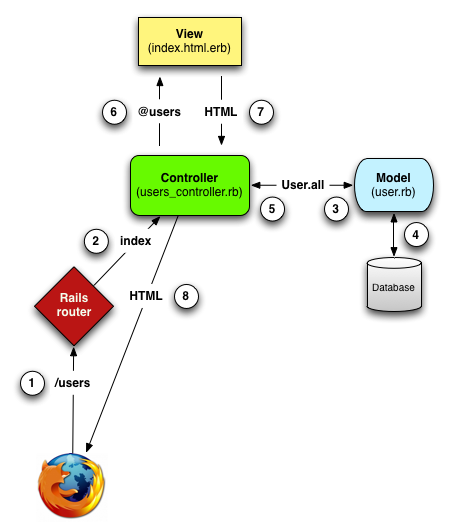
The views control what gets output to the browser and is what the client sees as a result of this architecture. It listens to the controller and changes state based on what rules the controller establishes.

A separate entity that isn’t part of the MVC name but is a key cog in its system is the Rails router. The rails router resides in the “db” folder under the name “config.rb.” In the context of Lemur, the config.rb looks like this:

|  |
| --- |
| Lemur::Application.routes.draw do  get "password\_resets/new"  get "password\_resets/edit"  get "sessions/new"  get "users/new"  root 'static\_pages#home'  get 'about' => 'static\_pages#about'  get 'help' => 'static\_pages#help'  get 'terms' => 'static\_pages#terms'  get 'signup' => 'users#new'  get 'login' => 'sessions#new'  post 'login' => 'sessions#create'  delete 'logout' => 'sessions#destroy'  resources :users do  member do  get :following, :followers  end  end  resources :users  resources :account\_activations, only: [:edit]  resources :password\_resets, only: [:new, :create, :edit, :update]  resources :microposts, only: [:create, :show, :destroy]  resources :relationships, only: [:create, :destroy]  end |

The Rails routing system controls what URLs get generated from the controller. This is what as known in Rails as RESTful (Representational State Transfer) design. In order to get a clearer picture of the routes generated by this file, one can simply run rake routes into the command line. This is what appears in the command line as a result in the context of Lemur:

|  |
| --- |
|  |

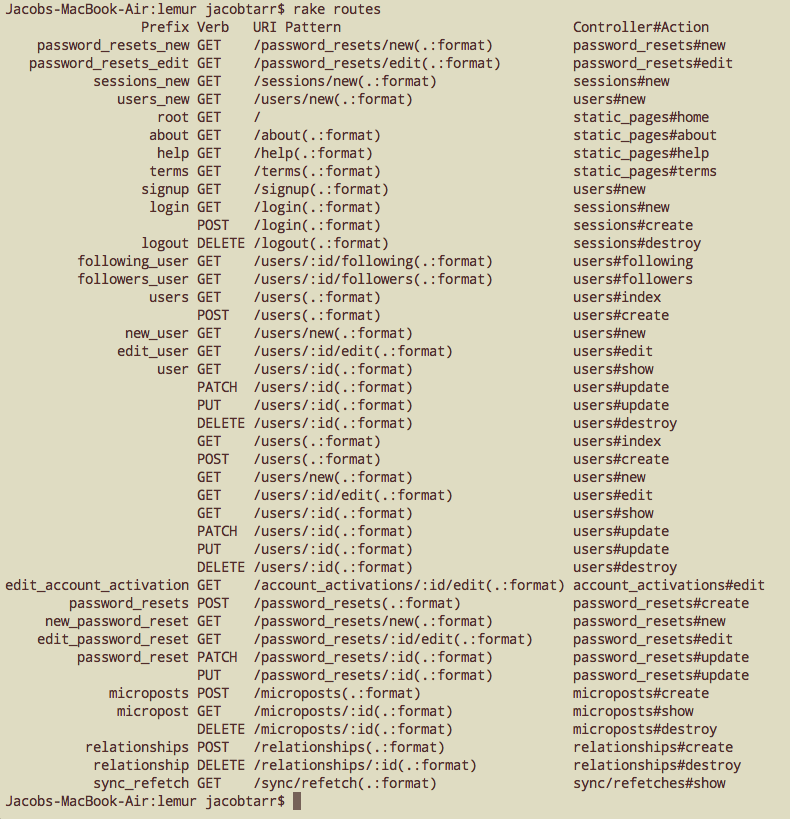


**Figure 1: A diagram showcasing the MVC principle in action. (Source: *https://github.com/sergey-alekseev/bsu-on-rails/wiki/Week-2-%C2%B7-MVC-architecture,-components-of-Rails*)**

2.2 **Active Record**

Active Record is the personal butler for a Rails developer. Through Active Record, Rails can interact with the database without the need for a developer to use SQL injection statements. Instead, AR uses Ruby code to interact with the database for things such as dropping and adding columns. Rails also allows the use of command line snippets to generate these “migrations.” For instance, entering in rails generate migration AddRoomNumberToHotels room\_number:string to the command line, will generate the following:

|  |
| --- |
| class AddRoomNumberToHotels < ActiveRecord::Migration  def change  add\_column :hotels, :room\_number, :string  end  end |



**Figure 2: Running “rake routes” in the command line generates a listing of all the URI patterns.**

These routes and what gets shown is managed by the controllers. For instance, certain RESTful patterns like DELETE and PATCH are exclusively associated with only certain controller actions like “#destroy” and “#update” respectfully.

However, in order to make sure these changes get migrated over to the database, Rails requires the developer to run bundle exec rake db:migrate in the command line. All changes show up in “schema.rb” in the “db” folder. The schema file shows all of the structural architecture of the database, showing things like the names of columns and datatypes.

3. **Application Structure**

Lemur’s application structure, like most Rails applications, is consisted of the following tree:

|  |
| --- |
| lemur/   * app/   + assets/     - images/     - javascripts/     - stylesheets/   + controllers/   + helpers/   + mailers/   + models/   + views/ * config/ * db/ * lib/ * log/ * public/ * test/ * tmp/ * vendor/ |

3.1 **Asset Pipeline**

Rails packages assets like stylesheets, javascripts and images through the “Asset Pipeline.” This is where all the aesthetic assets used to represent the application’s visual look are served. When Rails is used in production, all of the files in the stylesheets and javascripts is minified and concatenated to speed up page loads. In the old days, Rails developers had the unfortunate luxury of trying to maintain large CSS files in development. Although Rails made this easier to maintain with the introduction of the Asset pipeline, it made scaling a large application an arduous task. In 2014, web development has advanced to the point where CSS preprocessing is now possible. The most popular choice of these preprocessors is Sass. Rails automatically adds the sass-rails gem by default in every created application.

Lemur’s Sass setup uses a tweaked version of the SMACSS approach. SMACSS, or Scalable and Modular Architecture for CSS, isn’t a language. Instead, it’s a modern approach to building CSS by dividing states and sections of the CSS into various folders, as you can see below.

|  |
| --- |
| stylesheets/   * base/ * modules/ * sections/ * vendor/ * application.scss |

In application.scss, all of the files (called partials) are gathered to send to the asset pipeline. Lemur uses mixins, which are located in the base folder, to eliminate duplicated lines of code and to tidy up the stylesheet as well. Mixins are like functions in CSS which can be used across the application to embed pieces of CSS code.

For instance, \_header.scss uses a mixin that returns the max-width of a given element and centers it.

**app/assets/stylesheets/base/\_mixins.scss**

|  |
| --- |
| // Set common layout properties of content in a container  @mixin content-layout($max-width) {  max-width: $max-width;  margin: 0 auto;  } |

**app/assets/stylesheets/sections/\_header.scss**

|  |
| --- |
| .site-header-content {  @include content-layout($site-width\_\_main);  padding: 20px 0;  padding-left: 15px !important;  padding-right: 15px !important;  } |

3.2 **The Model**

As previously mentioned, the model is where the data is stored and validated. The model is the entity in the MVC that has the ability to communicate directly with the database.

For Lemur, there are three models: the User model, the Relationship model and the Micropost model.

**app/models/user.rb**

|  |
| --- |
| class User < ActiveRecord::Base  has\_many :microposts, dependent: :destroy  has\_many :active\_relationships, class\_name: "Relationship",  foreign\_key: "follower\_id",  dependent: :destroy  has\_many :passive\_relationships, class\_name: "Relationship",  foreign\_key: "followed\_id",  dependent: :destroy  has\_many :following, through: :active\_relationships, source: :followed  has\_many :followers, through: :passive\_relationships, source: :follower  attr\_accessor :remember\_token, :activation\_token, :reset\_token  before\_save :downcase\_email  before\_create :create\_activation\_digest  validates :name, presence: true, length: { maximum: 50 }  VALID\_USERNAME\_REGEX = /\A[a-z0-9\-\_]+\z/  validates :username, presence: true, length: { maximum: 20 },  format: { with: VALID\_USERNAME\_REGEX },  uniqueness: { case\_sensitive: false }  VALID\_EMAIL\_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i  validates :email, presence: true, length: { maximum: 255 },  format: { with: VALID\_EMAIL\_REGEX },  uniqueness: { case\_sensitive: false }  has\_secure\_password  validates :password, length: { minimum: 6 }, allow\_blank: true  validate :user\_location  validates :about\_me, length: { maximum: 160 }  validate :website  …  end |

The first few lines of the User model contain polymorphism. Polymorphism is when one class is said to belong to another class. It is like saying many children belong to one parent, but only one parent belongs to one children. By explicitly telling Rails these associations, a developer has access to parent classes and methods. VALID\_USERNAME\_REGEX is a variable, which is set to an obscure set of characters and numbers. This is known as a regular expression, a sequence of characters that tells Rails that certain data can only contain a specific set of characters. In this case, the username for a user can only contain letters and numbers.

The username for a user is further marked up through the use of a validation. In Rails, validating data is useful because it ensures that the data types don’t exceed or include any unnecessary characters. The username in this situation can’t exceed 20 characters. Otherwise, the system would allow a user to have a paragraph for a username.

3.3  **The Controller**

If the model is considered the brain of the MVC, the controller can be thought as more of the blood stream. It breathes life into all aspects of the MVC, including the Rails routing system.

Lemur has a total of 8 controllers: the application\_controller.rb, account\_activation\_controller.rb, microposts\_controller.rb, password\_resets\_controller.rb, relationships\_controller.rb, sessions\_controller.rb, static\_pages\_controller.rb and the users\_controller.rb.

Taking a look at some of the code included in the users\_controller.rb, one will notice that there are several methods defined.

**app/controllers/user\_controller.rb**

|  |
| --- |
| class UsersController < ApplicationController  before\_action :logged\_in\_user, only: [:index, :edit, :update, :destroy,  :following, :followers]  before\_action :logged\_in\_user, only: [:index, :edit, :update, :destroy]  before\_action :correct\_user, only: [:edit, :update]  before\_action :admin\_user, only: :destroy  def index  @user = User.find(current\_user)  @users = User.paginate(:page => params[:page], :per\_page => 20)  end  def show  @user = User.find(params[:id])  @microposts = @user.microposts.paginate(page: params[:page])  end  def new  @user = User.new  end  def create  @user = User.new(user\_params)  if @user.save  @user.send\_activation\_email  UserMailer.account\_activation(@user).deliver\_now  flash[:info] = "Please check your email to activate your account."  redirect\_to root\_url  else  render 'new'  end  end  def edit  @user = User.find(params[:id])  end  def update  @user = User.find(params[:id])  if @user.update\_attributes(user\_params)  flash[:success] = "Profile updated"  redirect\_to @user  else  render 'edit'  end  end  …  end |

The first few lines of the controller give the before\_action to certain methods which are declared in other parts of the application. Basically, before\_action commands tell Rails to filter only certain actions. For instance, an admin can only destroy posts. They cannot edit other users’ posts.

Each one of these actions is associated with their respective views. For instance, the “show” action represents show.html.erb in the users view. In that view, the instance variable @user finds a user by his or her parameters. This instance variable is used in the view to show various data of a user, such as the “About Me”, username, name and other data types that are declared in the model.

3.3  **The View**

The views are what actually gets outputted to the browser. Views are built using HTML with embedded ruby code controlling the content. Since views can often get very massive in size, it’s often good practice to use partials to render views. Partials are little components of an application which a developer can insert into another view in order to make code more readable.

Lemur includes views for account activations, layouts, microposts, password resets, relationships, sessions, shared, static pages, user mailer and users.

Taking a look at the show.html.erb in the users folder, one can see how these partials work:

**app/views/users/show.html.erb**

|  |
| --- |
| <% provide(:title, @user.name) %>  <% provide(:main\_class\_modifier, "--dashboard") %>  <div class="dashboard">  <aside class="dashboard\_\_sidebar">  <%= render 'shared/dashboard/user\_info' %>  <%= render 'follow\_form' if logged\_in? %>  <%= render 'shared/dashboard/user\_vitals' %>  </aside>  <section class="dashboard\_\_content">  <%= render 'user\_microposts' %>  </section>  </div> |

The first render uses the \_user\_info.html.erb partial inside the shared folder. For this particular partial, the user’s gravatar and follower stats are displayed in the sidebar:

**app/views/shared/dashboard/\_user\_info.html.erb**

|  |
| --- |
| <div class="dashboard\_\_sidebar--avatar-section">  <div class="avatar-top">  <%= link\_to gravatar\_for(@user, size: 120), @user %>  <h1><%= @user.name %></h1>  <span><%= at\_username @user %></span>  </div>  <div class="avatar-bottom">  <%= link\_to(@user, class: "avatar-bottom\_\_stat") do %>  <% if @user.microposts.any? %>  <h2><%= @user.microposts.count %></h2>  <% end %>  <span>posts</span>  <% end %>  <% @user ||= current\_user %>  <a href="<%= following\_user\_path(@user) %>" class="avatar-bottom\_\_stat">  <h2 id="following"><%= @user.following.count %></h2>  <span>following</span>  </a>  <a href="<%= followers\_user\_path(@user) %>" class="avatar-bottom\_\_stat">  <h2 id="followers"><%= @user.followers.count %></h2>  <span>followers</span>  </a>  </div>  </div> |

As one can see, the instance variable @user is used to display user-specific information by adding methods onto it.

The “dashboard\_\_content” section is used to display all of the user’s posts:

**app/views/user/\_user\_microposts.html.erb**

|  |
| --- |
| <div class="dashboard\_\_content--feed-section">  <div class="pagelet">  <div class="pagelet\_\_header">  <i class="fa fa-rss"></i><%= @user.username %>'s Posts  </div>  <div class="pagelet\_\_body">  <% if @user.microposts.any? %>  <ul class="posts-list">  <%= render @user.microposts %>  </ul>  <%= will\_paginate @microposts %>  <% end %>  </div>  </div>  </div> |

When the @microposts variable is declared, it uses the paginate method on top of users to split the lists of posts into different pages. This uses the will\_paginate gem, which makes pagination easy in Rails.

4 **Adding Real-Time to Lemur**

Unfortunately, as it stands now due to certain constraints, there is no real-time aspect to Lemur currently. The development stage of adding real-time was a lot of trial and error, but was not implemented in the end. The original plan was to use the sync gem. According to the Github page made by its creator, “sync lets you render partials for models that, with minimal code, update in realtime in the browser when changes occur on the server.”

The following steps need to be taken in order to make a RESTful Rails application real-time using the sync gem:

1. Add the following gems to the Gemfile (Sync uses Faye or Pusher to handle real-time events). Then run “bundle install” and “rails g sync:install” in the command line.

|  |
| --- |
| gem 'faye' // or gem 'pusher'  gem 'thin', require: false  gem 'sync' |

2. Write //= require sync in the Javascript manifest file application.js.

3. Under the include tag for the javascript manifest file in the application layout (application.html.erb), write <%= javascript\_include\_tag Sync.adapter\_javascript\_url %> so that the pubsub adapter’s javascript is included.

4. Alongside the desired Rails server choice, run “rackup sync.ru -E production” in the command line.

5. Without all of that configured, change any partial that needs rendering to say <%= sync partial: “post\_partial”, resource: @post %> instead of <%= render partial: “post\_partial”, locals: { post: @post } %>.

In theory, those are the necessary steps in order to make a chunk of the application real-time.

5 **Deployment to Heroku**

Lemur was deployed to Heroku, which can be accessed via https://getlemur.herokuapp.com/. Heroku is a cloud-based PaaS (platform as a service) provider. It’s completely free to host something like a side project or any type of project. Premium accounts exist for more robust and enterprise applications.

Heroku also makes it incredibly simple to create an application and deploy it. Below are the necessary steps to get an app up and running on Heroku.

1. Sign up for a Heroku account.

2. Add the current project to a Git repository. Heroku requires the use of Git and uses it to push a Git repository into production on the web.

3. In the Gemfile of the application, type in the following:

|  |
| --- |
| group :production do  gem 'pg'  gem 'rails\_12factor'  end |

Since Heroku uses the PostgreSQL database in production, it is important to add the “pg” gem to the Gemfile. Additionally, the “rails\_12factor” is also required because it is used by Heroku to serve static assets such as images. Run “bundle install --without production” to tell Bundler to only allow these gems in production.

4. Run heroku version in the command line. If nothing pops up, install the Heroku toolbelt from the Heroku website.

5. Run heroku login and login with the necessary account credentials. Next type in heroku keys:add to add the SSH key.

6. Lastly, run heroku create in the command line to create the Heroku application. Once done, navigate to the random address that Heroku assigns to the application. This can be changed on the Heroku website.

7. Done!