Creating Awesome Websites with Ruby on Rails

Tommy MacWilliam

Harvard University

November 13, 2010

Today

- the Ruby programming language
- MVC: what and why?
- Riding the Rails like a pro

Today

- are you ready?
- creating awesome web applications is not a spectator sport

Setup

- http://www.ruby-lang.org/en/downloads/
  - Ruby 1.9.2 is recommended (that's what I'll be using)
  - Windows users, make sure you check all the boxes on the installer
- then, run `gem install rails`
  - this will install Rails 3.0 (which is different than Rails 2!)
  - don't worry if nothing comes up on the terminal for a while, it's downloading

Setup

- if you run into an error message about sqlite, then grab a binary from http://www.sqlite.org/download.html
  - Windows: copy the DLL to the “bin” folder in your Ruby install path (on Windows, probably C:\Ruby192\bin)
  - UNIX: make sure you’ve installed `ruby`, `ruby-dev`, `sqlite3`, `libsqlite3-dev`, and `libsqlite3-ruby`
- if you get an error message about ruby or rails not being found, make sure the binaries are in your PATH
- Google is your friend, you’re not the first person to have trouble

Ruby

- Wikipedia says, “Ruby is a dynamic, reflective, general purpose object-oriented programming language”
  - sounds cool to me
- Ruby wants to help you get stuff done
  - clean, readable, intuitive syntax
  - no petty low-level stuff (aka pointers)
  - huge standard library, 100% documented

Ruby

- “Hello, World!” program in C (in case you forgot):

```
#include <stdio.h>
int main(int argc, char** argv) {
    printf(“Hello, World\n”);
    return 0;
}
```

Ruby

- “Hello, World!” program in Ruby

```
puts “Hello, World!”
```

Ruby

- owned.

Ruby

- minor syntactic differences
  - no more braces: `end` designates the end of a condition/loop
  - no more semicolons either
  - parentheses for function arguments are optional
  - `#` is a single-line comment
  - `=begin` starts a multi-line comment, `=end` ends a multi-line comment
  - `elsif` keyword is used instead of `else if`

Ruby

- Ruby is an interpreted language: no compiling, just write code and run it
- Ruby is dynamically typed: you don't need to specify types for variables and functions

```
def say_hello(name)
    puts "Hello, " + name
end
```

Ruby Arrays

- Ruby arrays do not have a fixed size and can contain multiple types
  - `numbers = [ 1, "2", 3 ]`
- access the 0th element of the array: `numbers[0]`
- add a new element to the array: `numbers.push(4)`
- get the value of the last element and remove it from the array: `numbers.pop()`
- get the value of the first element and remove it from the array: `numbers.shift()`
- concatenate two arrays: `numbers + [ 5, 6, 7 ]`

Ruby Hashes

- Ruby also has built-in support for hashtables
- `tf = { :name => “Tommy”, :rank => 1 }`
    - access the “name” field: `tf[:name]`
    - add a new field: `tf[:coolness] = “high”`

Ruby Blocks

- iterating over an array in Ruby is different (aka better) than what we've seen so far
- Ruby makes heavy use of "blocks": pieces of code that are passed as arguments to a function

```
numbers.each do |number|
    puts number
end
```

- every array has a function called `each`, which takes a single block as an argument
  - the block will be called on every element of the array individually
  - the argument to the block (called `number` and given inside pipes, not parentheses) is the current element of the array

Ruby Helpful Links

- a great (free) book if you want to learn even more Ruby: http://ruby-doc.org/docs/ProgrammingRuby/
- complete documentation (with examples) for every function in the standard library: http://ruby-doc.org/ruby-1.9/index.html

MVC: What?

- MVC stands for “model-view-controller”
- MVC is a design pattern: a solution to a common, general problem
  - in this case, “how do I structure my web application?”
- also used by frameworks like CakePHP and the iOS SDK

MVC: What?

- model: the database in your application
  - abstracts away SQL queries, access your database as if it were a Ruby object
- view: the user interface in your application
  - what the user actually sees, like HTML pages
- controller: the bridge between model and view
  - query the model for data and pass it to the view

MVC: Why?

- allows for clean separation of design and logic
- cleaner organization of code
- maximize code re-usability

Creating a new Rails Application

- creating a new application: `rails new <application name>`
  - creates a new folder corresponding to the application name we specified
- we're going to create a blog, so let's run `rails new blog`
- now, we just need to make sure everything is installed correctly
  - runing `bundle install` will take care of all that for you

Creating a new Rails Application

- example time!
  - creating a new application

Creating a new Rails Application

- WOAH. that's a lot of files Rails just made for us
- all of our code will go into the `app` directory
  - and there are already folders for models, views, and controllers!
  - you'll also notice folders for images, stylesheets, and javascripts in the `public` folder
  - sweet, that required like 0 effort

Rails Resources

- our blog needs to have posts, so we need to create a model/controller/views for creating, viewing, editing, and deleting posts
  - a post is called a “resource”: a single “thing” that can be manipulated and represented in a database
- before we create a resource, we need to know what fields the database table should have
  - for now, a post has a title and content

Rails Resources

- creating a resource: `rails generate scaffold <resource name> <column:type> ...`
  - “scaffold” tells rails to create the model, view, and controller all at once
- so we want to run `rails generate scaffold Post title:string content:text`

Rails Migrations

- now that we have our resource, we need to add a new table to our database to reflect that
- so we have to go into phpMyAdmin and create a new table and create the columns and give the columns types and stuff right?

Rails Migrations

- WRONG.
- just run `rake db:migrate`
  - in Rails, a modification to the database is called a “migration”
- Rails can take care of all that database stuff for you, since it created the resource for you

Rails Resources

- example time!
  - creating a new resource and running its migration

Rails Server

- okay, let's check out what Rails did for us
- running your application: `rails server`
  - now, navigate your browser to `http://localhost:3000`
  - we created a resource for blog posts, so let's head over to `http://localhost:3000/posts`

Rails Server

- example time!
    - viewing our application

Rails is Pretty Sweet

- http://www.youtube.com/watch?v=wacmF9_6WqU
  - that just happened.
- we just made a blog without writing a single line of code
  - nbd.

Rails Generated Code

- let's take a look at the code that was so nicely written for us
- the post model is located in `/app/models/post.rb`
- the post views are located in `/app/views/posts`
- the post controller is located in `/app/controllers/posts_controller.rb`

Rails Models

- the model is extremely simple: our posts aren't doing anything fancy with the database, so we just need an empty class
- notice the class inherits from `ActiveRecord::Base`
  - the parent class is going to take care of everything we need

Rails Controllers

- the controller is a bit more complicated
- each function in the controller corresponds to a single user action
    - creating a post is a single action
    - each action maps to a specific URL (which Rails has so nicely labeled in the comments)

Rails Controllers

- the `params` hash is analogous to PHP's `$_GET` and `$_POST` arrays
  - if the user makes a request to `/posts/edit/1`, then `params[:id] == 1`
- to pass a variable to the view, declare it with `@` before its name
- `Post.new` creates a new instance of the Post class (our model)
  - this object will represent a single post in the database
  - the `save` function saves the object to the database (INSERT-ing or UPDATE-ing as necessary, you don't have to worry about that either)

Rails Controllers

- `index`: get all posts and pass them on to the view
- `show`: get a single post with the ID specified in the URL and pass it to the view
- `new`: create a blank post object and pass it to a view that contains a form for a new post
- `edit`: send the data for an already-existing post to the view that contains a form to edit a post
- `create`: use the data passed from the view to create a new row in the Post table
- `update`: use the data passed from the view to update an existing fro in the Post table
- `destroy`: delete a post with the ID specified in the URL

Rails Views

- each user action also has its own view (`.erb` file)
- PHP uses `<? ?>` to embed PHP code in HTML, Ruby uses `<% %>`
- the `link_to` function generates URLs so we don't have to deal with ugly string concatenation
  - `link_to(<text>, <resource>)`: URL for the `show` function for a single instance of a resource
  - `link_to(<text>, edit_<resource>_path(<resource>)`: URL for the `edit` function for a single instance of a resource
  - `link_to(<text>, <resource>s_path)`: URL for the `index` function for all instances of a resource
- the `form_for` function generates the HTML for a form

Rails Views

- `index.html.erb`: use a Ruby block to iterate over all posts and display them in a table
- `edit.html.erb`: render and populate a form with values from an already-existing post
- `new.html.erb`: render a blank for to allow the user to create a new post
- `show.html.erb`: display the fields of a post

Adding Authors

- let’s add the ability to view all posts by a certain author
  - new field in our database for an author
  - new function in the controller
  - add author fields to existing views and create a new view to display author results

Adding Authors

- we need a database migration to add a new column to the posts table
  - add a column: `rails generate migration Add<column>To<table> <column:type>`
  - remove a column: `rails generate migration Remove<column>From<table> <column:type>`
- so we want to run: `rails generate migration AddAuthorToPosts author:string`
- now we just run `rake db:migrate` again and we're good to go

Adding Authors

- now we can create a new function called `author` in the Posts controller that will get all posts from a given author
- the `Post` class already has a built-in function called `where` that will query our database for us
  - `@posts = Post.where({ :author => params[:id] })`
  - this will get all the posts where the author is the author author specified in the URL and send them all to the view
- we can just re-use the index view, so copy `index.html.erb` and rename it to `author.html.erb`

Adding Authors

- notice there's an extra file called `_form.html.erb` in our views folder
- this is called a partial: a small chunk of re-usable view code
  - in this case, the form that will be displayed to the user when creating/editing a post
  - the create/edit views then simply use the `render` function, giving it the name of the partial, to display the same form
  - just like the PHP `require_once` function
- if we just edit this, then our changes will be reflected in both `/posts/new` and `/posts/edit`

Adding Authors

- notice how our URLs are magically mapped to functions in your controller
  - `/posts/new` knows to call the `new` function in `posts_controller`
- Ruby "routes" a URL to a controller/function based on the contents of `/config/routes.rb`
  - `resources` is a shortcut for mapping `index`, `new`, `show`, etc. individually
- to add our new `author` function, we can just follow the instructions given in the comments of `routes.rb`
  - `author` will be a member of the `posts` resource and accessed with a GET request
- you can view all routes in your application with `rake routes`

Adding Authors

- example time!
  - adding the author field

Adding Comments

- now let's give users the ability to comment on posts
- adding authors to posts required modifying the Post model, but we're going to need a new model for Comments
- just like before: `rails generate scaffold Comment author:string content:text post:references`
  - a comment must be tied to a specific post, so we need a special field containing which post it refers to
  - don't forget to `rake db:migrate`

Adding Comments

- when we fetch a Post from the database, we also want to get all of its associated comments
  - as you hopefully expected at this point, Rails will do this for you
- we need to update the models for Post and Comment to reflect this relationship
  - a Post `has_many` comments
  - a Comment `belongs_to` a post
- we also need to update our routes
  - Rails needs to know how to attach a comment to a specific post
  - a comment cannot exist without a post, so it must be a resource of posts

Adding Comments

- we need to update our views to allow users to comment on a post
  - the new comment form should go in `/app/views/posts/show.html.erb`
  - we can use `_form.html.erb` as a starting point
  - change the labels and text fields to reflect the columns in our comments model: author and content
- the post's comments must be added to the `form_for` method so the form submits to the comments controller
  - we need to call the `build` function on the `comments` field because the new comment will be linked to an existing post

Adding Comments

- this view should also display all of the given post's comments
  - we can use `index.html.erb` as a starting point
  - use the `each` iterator on the post's comments, then display the commenter's name and the comment

Adding Comments

- finally, we need to update our controller to save comments to the database as well
- remember, the form we just created will send data to the `create` function in the comments controller, so we need to modify that
  - first, we need to get the post being commented on from the database
    - because we set up our routes, `params` will contain a `post_id` field representing the ID of the post we're commenting on
  - to associate the submitted comment with this post, call the `create` function on the post object's `comments` field
  - finally, we can save the post to the database

Adding Comments

- example time!
  - adding comments

Rails Layouts

- you may have noticed that our views aren't full HTML documents
  - no `<html>` or `<body>` tags
- but, when we view the source of a page, it looks like we have valid HTML
- the secret lies in that `layouts` folder in `/app/views`

Rails Layouts

- `application.html.erb` is the layout for every page in your application
  - aha! the `<html>` tag!
- when Rails renders your page, it inserts the view's `.erb` file at the `<%= yield %>` statement in the layout
- adding a header or a navigation bar to every page in your application is as simple as editing the layout
- per-controller layouts are also supported
  - just create a file called `<resource>.html.erb` inside this folder (e.g. `posts.html.erb`)

Rails Layouts

- `stylesheet_link_tag` and `javascript_include_tag` will generate HTML to include CSS/JS
  - just put your CSS and JS files in `/public/stylesheets` and `/public/javascripts`
  - `javascript_include_tag("magic")` includes the file `magic.js`
- `image_tag` will generate the HTML to put images on your page
- you /must/ use these functions rather than writing `<img>` and `<script>` tags yourself
  - the same layout file is used for `/posts` and `/posts/1/edit`, but you need to specify the path to the image/CSS/JS file

Rails Helpful Links

- API documentation and helpful guides: http://rubyonrails.org
- free eBook about Rails: http://railstutorial.org/book

Thanks!

- go make something awesome!
- questions? comments? suggestions?
  `<tmacwilliam@cs50.net>`