Intro to Rails

INFO 2310:
Topics in Web Design and Programming

ISSA General Meeting
Mon 9/8/08 @ 4:30 in Upson 205

The Information Science Student Association will have its first General Meeting this Monday (9/8) in Upson 205 at 4:30

Meet other people in the IS major and learn about where/how they spent their summer

Get involved in planning social events

Get information about jobs/internships

Bring other IS students and friends who are interested in the major

We will have food!

infoscience STUDENT ASSOCIATION

Today

- We'll start our blog project
- We'll talk a lot about Rails design and philosophy
  - Rails is a philosophy-heavy framework

Without further ado…

Let's just do it, shall we? In the spirit of doing things first, understanding them later…

- Pull up a command shell
- If you're using a flash drive, change to flash drive (`F:` or `G:` or whatever…)
- Enter '`rails –d sqlite3 blog`'

```
F:\>rails -d sqlite3 new_blog
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      create  config/initializers
      create  db
      create  doc
      create  lib
      create  lib/tasks
      create  log
      create  public/images
      create  public/javascripts
      create  public/stylesheets
      create  script/performance
      create  script/process
      . . .
```

Rails creates lots of files and directories…

Now…

- `cd blog`
- `ruby script/generate scaffold post title:string body:text`
- `rake db:migrate`

Fire up the web server…

Rails has its own built-in webserver (like Apache, for example). We'll use it to look at our application.

- `ruby script/server`

```
=> Booting WEBrick...
=> Rails 2.1.0 application started on
   http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help
   for options
[2008-08-30 20:46:58] INFO  WEBrick 1.3.1
[2008-08-30 20:46:58] INFO  ruby 1.8.6 (2007-09-
   24) [i386-mswin32]
[2008-08-30 20:46:58] INFO
   WEBrick::HTTPServer#start: pid=4796 port=3000
```

Let's take a look!

The server is listening on port 3000. Start a web browser and navigate to http://localhost:3000.

But I wanna see it DO something…

Go to http://localhost:3000/posts.

Go ahead and play around for a while.

Look at the URLs as you enter/edit/delete posts.

If you want, look at the underlying DB. Go to blog/db, type '`sqlite3 development.sqlite3`'. Now you can '`SELECT * FROM posts;`'

What did we do?

At a high level:

- `rails –d sqlite3 blog`
  - Create the blog project, using the SQLite3 database
- `ruby script/generate scaffold post title:string body:text`
  - For the blog project, create one 'model' called 'post' having two attributes, title (type string) and body (type text), and create the associated files (including instructions on creating associated DB).

- `rake db:migrate`
  - Create the posts DB.

Object-Relational Model

Objects and DBs

Object-oriented programming is a useful thing to be able to do.

How can connect information in DBs with objects?

Object-Relational Mapping

Each DB table (e.g. ‘posts’) has a corresponding class (e.g. ‘Post’).

Each column of the table is an attribute of the class.

Each row of the table can be fetched/saved as an instance of the class.

See it

The Post class is defined in blog/app/models/post.rb.

Take a look…

post.rb

```
class Post < ActiveRecord::Base
end
```

Not much there, eh? Each table is a subclass of `ActiveRecord`, which has lots of methods we can call to access the table.

Try it

Hopefully you added some posts to your blog.

We can interact directly with our Rails application via irb. In your blog directory, type ‘`ruby script/console`’.

Create a new post directly from the console.

```
mypost = Post.new
mypost.title = “Another post!”
mypost.body = “I made it at the console!”
mypost.save
```

Check that it is there in your webbrowser…

Or pass a hash as the initial input…

```
mypost2 = Post.new(:title =>
  “Yet another post!”, :body =>
  “Isn’t this great?”)
mypost2.save
```

You can also do DB lookups using the Post class.

```
Post.find(idnum)
Post.find(:first)
allposts = Post.find(:all)
Post.find(:first, :conditions =>
  {:title => “Another post!”})
Post.find_by_title(“Another post!”)
```

In general, most of the easy stuff we want to do via SQL we can just do with method calls…

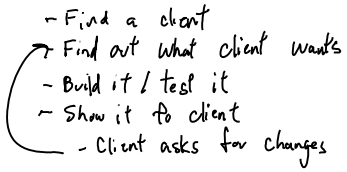
Rails philosophy

Lots of philosophical underpinnings to Rails. It will be easier to understand how it works if you know why it was made to work that way.

(You can keep poking around while we discuss this; maybe start looking through all the files that were created).

Agile software

How does one usually design a website for a client?



- Gather client requirements
- Document and design system
- Implement and test system
- Show system to client
- Repeat

Agile Manifesto

Value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Agility and Rails

It's easy to make small modifications and get them working.

Then you can more easily work by making incremental changes, getting client feedback, and responding to it.

Don't repeat yourself (DRY)

Define/do something only in one place. "If you're doing it twice, you're doing it wrong!"

Examples:

- Username/password/database name info to connect to a database – in one place, not multiple places in your code
  - Look this up in blog/config/database.yml.

database.yml

```
# SQLite version 3.x
#   gem install sqlite3-ruby (not necessary on OS X Leopard)
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  adapter: sqlite3
  database: db/test.sqlite3
  timeout: 5000

production:
  adapter: sqlite3
  database: db/production.sqlite3
  timeout: 5000
```

DRY

- Making DB changes, such as adding attributes to a table. Write one script (a "migration"). `Rake` then automatically generates the correct SQL to make the change, makes it, and alters the object definitions in your code.

Convention over configuration

Only define *unconventional* configurations. For everything else, just follow the defaults ('conventions') and things should just work.

E.g. we'll soon create another DB table called 'comments'. We'll be able to tell Rails that any Post

```
has_many :comments
```

Rails will automatically assume that the comments table has a column 'post_id' that is a foreign key to the posts table; i.e. you can then easily do joins.

You can override things as needed. But if you follow the conventions, you write less code.

Model-View-Controller Pattern

Model-View-Controller

Model-View-Controller (MVC) is a (now) standard architecture in software engineering. It separates the data and operations on the data from code that actually displays it.

Separation means that we can make changes to one part without affecting what happens to the other.

PHP

Things tend to get mixed up in the PHP applications we wrote in 230.

```
<form method="GET" action="thispage.php">
<table>
 <?php
 if (isset($_GET['sort'])) {
   $query = "SELECT * FROM Movies ORDER BY ".$_GET['sort'];
 } else $query = "SELECT * FROM Movies";
 $results = mysql_query($query,$link);
 while ($row = $mysql_fetch_array($results, MYSQL_ASSOC)) {
    print("<tr>");
    foreach ($row as $item) {
        print ("<td>$item</td>");
    }
    print("</tr>");
 } ?>
 <input type="submit" name="sort" value="title" />
```

Model

The objects used to store and manipulate the data of the applications.

We've already seen an example with the 'Post' class.

Controller

Take requests (generated by the user interface) and decide which of the views to *render*.

View

HTML (plus some Ruby) for displaying data from the models and getting user input from forms.

The cycle

- User interacts with the browser and triggers an event (e.g. submits a post)
- Controller receives input from the interface (e.g. the submitted post)
- Controller accesses the model, updating it in some way (e.g. creating a new post)
- The controller renders an updated view (e.g. shows the new post)
- The browser waits for new input from the user.

Controller code

We've seen the 'Post' code; where are the views and controllers?

Controller for 'posts' is in `blog/app/controllers/posts_controller.rb`.

Take a look…

```
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.xml
  def index
    @posts = Post.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @posts }
    end
  end

  # GET /posts/1
  # GET /posts/1.xml
  def show
    @post = Post.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @post }
    end
  end
```

Roughly – what to do for each possible action on a post:

- Create (new)
- Read (show)
- Update (update)
- Delete (destroy)

sometimes abbreviated as CRUD.

Routes the URLs we saw in the application: posts/1, posts/1/edit…

Decides which view to send to browser.

Views

HTML in two locations:

- blog/app/views/layout/posts.html.erb
- blog/app/views/posts/index.html.erb,
  …/show.html.erb
  …/edit.html.erb
  …/new.html.erb

posts.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
   transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
   lang="en">
<head>
  <meta http-equiv="content-type"
   content="text/html;charset=UTF-8" />
  <title>Posts: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= yield  %>

</body>
</html>
```

index.html.erb

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Body</th>
  </tr>

<% for post in @posts %>
  <tr>
    <td><%=h post.title %></td>
    <td><%=h post.body %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, :confirm => 'Are you sure?',
   :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New post', new_post_path %>
```

@posts = Post.find(:all)

PHP vs. Rails

PHP:

- Only current PHP file (and any 'required' files) is processed

Rails:

- Your entire application is loaded at once.

Learning More

Documentation

Can learn more about ActiveRecord at http://api.rubyonrails.org/files/vendor/rails/activerecord/README.html

Also ActionPack class has ActionView class and ActionController. ActionPack documentation at http://api.rubyonrails.org/files/vendor/rails/actionpack/README.html.

To do

Try to improve the very basic styling of the post pages.

CSS file is in public/stylesheets/scaffold.css.

ISSA General Meeting
Mon 9/8/08 @ 4:30 in Upson 205

The Information Science Student Association will have its first General Meeting this Monday (9/8) in Upson 205 at 4:30

Meet other people in the IS major and learn about where/how they spent their summer

Get involved in planning social events

Get information about jobs/internships

Bring other IS students and friends who are interested in the major

We will have food!