

# ECE 495/595 – Web Architectures/Cloud Computing

## Module 2, Lecture 4: A Fully Functional Ruby on Rails Application

Professor G.L. Heileman

University of New Mexico



# Rails Applications

- In this lecture we'll take a closer look at how you develop Rails applications.
- This will include:
  - Looking at how different databases can be configured for different environments.
  - Consideration of more complicated ORM settings.
  - Authentication before granting access to certain resources.
  - Javascript integration.
- We'll do this by building the famous "Weblog in 15 Minutes" application:  
[http://media.rubyonrails.org/video/rails\\_blog\\_2.mov](http://media.rubyonrails.org/video/rails_blog_2.mov)  
But we'll take a little more time, and use Rails 3.

# Blog Application

## Usage specifications:

- Blogging application should allow the administer of the blog to enter new postings.
- Users should be able to comment on the postings.
- Users should not be able to modify the postings or other users' comments.

## Technical Specifications:

- The production environment uses SQLite3 and an the WEBrick Web Server.
- The application should handle browsers that do and do not have Javascript enabled.

# Blog Application

- First create the rails application using:

```
$ rails new blog
```

This will create a rails application called blog, and by default add in the SQLite3 adapter.

- Rails automatically installs SQLite database tools, later will look at how to make it support other databases.
- Look at ./config/database.yml, you should see something like:

```
development:
  adapter:  sqlite3
  database: db/development.sqlite3
  pool:    5
  timeout: 5000
```

This database is used by default when you execute:

```
$ rails server
```

- Notice that there are also specifications for test and deployment databases.

# The Models

- Create the posts model:

```
$ rails generate scaffold Post title:string body:text
```

- Then, create the comments model

```
$ rails generate scaffold Comment post_id:integer body:text
```

- Next, run

```
$ rake db:migrate
```

- If you type

```
$ rake routes
```

you'll how URLs are being mapped to controller actions.

- Now, go to the browser, and create a few Posts, then create a few Comments and link them to these Posts (using the post\_id field).
- Start up the rails console, and add a few Posts and Comments from the command line as well. E.g.,

```
> Post.new(:title => "My Post", :body => "...").save
```

# The Models – Associations

- We need to link up our two models, i.e., we need to associate comments with posts. Here's how:

```
# ./apps/models/post.rb
class Post < ActiveRecord::Base
  has_many :comments
end
```

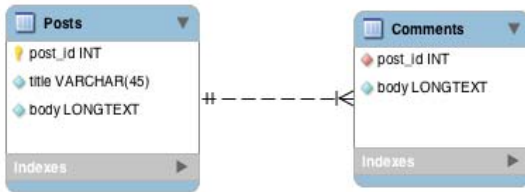
```
# ./apps/models/comment.rb
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

- The `has_many` declaration adds numerous methods to the `Post` class, allowing you to perform CRUD operations on Comments through Posts. E.g,

```
> post = Post.all
> post[0].comments
```

# Object-Relational Mapping

Here's what happening under the hood:



- Notice that the Comments table has a foreign key.
- The way to think of this is that this makes Comments “belongs to” Posts.

# Model Validations

Let's add some validations to our models:

```
# ./apps/models/post.rb
class Post < ActiveRecord::Base
  has_many :comments
  validates :title, :presence => true
  validates :body, :presence => true
end
```

```
# ./apps/models/comment.rb
class Comment < ActiveRecord::Base
  belongs_to :post
  validates :post_id, :presence => true
  validates :body, :presence => true
end
```

Now, try to add a Post or Comment, leave out any field, and see what happens.



# The View

We need to fix the view so that Comments appear with the Posts they refer to. Add the following to `./app/views/posts/show.html.erb`:

```
<h2>Comments</h2>
<div id="comments">
  <% @post.comments.each do |comment| %>
    <%= div_for comment do %>
      <p>
        <strong>
          Posted <%= time_ago_in_words(comment.created_at) %>
        </strong><br />
        <%= h(comment.body) %>
      </p>
    <% end %>
  <% end %>
</div>
```

# The View

We also need to be able to add new comments. Add the following to `./app/views/posts/show.html.erb`:

```
<%= form_for [@post, Comment.new] do |f| %>
  <p>
    <%= f.label :body, "New Comment" %><br />
    <%= f.text_area :body %>
  </p>
  <p><%= f.submit "Add Comment" %></p>
<% end %>
```

- This will only work if you modify the routes, by making comments a resource nested within posts. To do this, modify `./config/routes.rb` as follows:

```
resources :comments
resources :posts do
  resources :comments
end
```

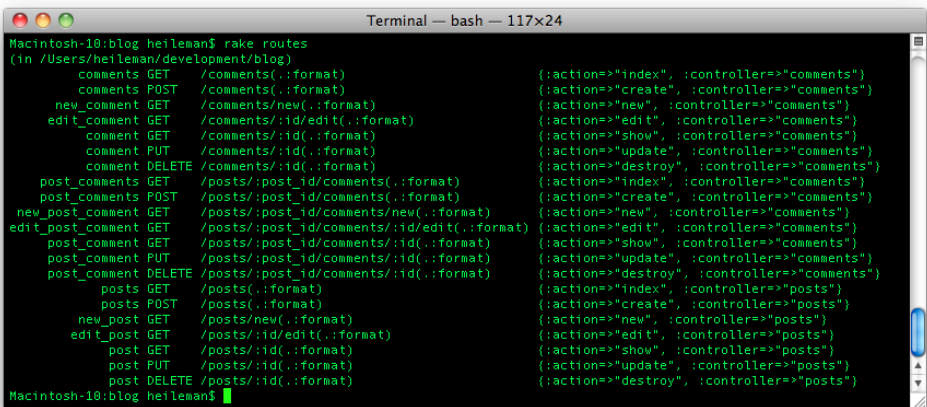
- Now you can submit URLs that look like:  
`http://localhost:3000/posts/1/comments/2`

# The View

Type:

```
$ rake routes
```

and you'll see the new routes that were added:



```
Macintosh-10:blog heileman$ rake routes
(in /Users/heileman/development/blog)
   comments GET    /comments(.:format)          {:action=>"index", :controller=>"comments"}
   comments POST   /comments(.:format)          {:action=>"create", :controller=>"comments"}
  new_comment GET    /comments/new(.:format)      {:action=>"new", :controller=>"comments"}
edit_comment GET    /comments/:id/edit(.:format) {:action=>"edit", :controller=>"comments"}
   comment GET    /comments/:id(.:format)      {:action=>"show", :controller=>"comments"}
   comment PUT     /comments/:id(.:format)      {:action=>"update", :controller=>"comments"}
   comment DELETE  /comments/:id(.:format)      {:action=>"destroy", :controller=>"comments"}
post_comments GET    /posts/:post_id/comments(.:format) {:action=>"index", :controller=>"comments"}
post_comments POST   /posts/:post_id/comments(.:format) {:action=>"create", :controller=>"comments"}
new_post_comment GET    /posts/:post_id/comments/new(.:format) {:action=>"new", :controller=>"comments"}
edit_post_comment GET    /posts/:post_id/comments/:id/edit(.:format) {:action=>"edit", :controller=>"comments"}
   post_comment GET    /posts/:post_id/comments/:id(.:format) {:action=>"show", :controller=>"comments"}
   post_comment PUT     /posts/:post_id/comments/:id(.:format) {:action=>"update", :controller=>"comments"}
   post_comment DELETE /posts/:post_id/comments/:id(.:format) {:action=>"destroy", :controller=>"comments"}
      posts GET    /posts(.:format)             {:action=>"index", :controller=>"posts"}
      posts POST   /posts(.:format)             {:action=>"create", :controller=>"posts"}
    new_post GET    /posts/new(.:format)         {:action=>"new", :controller=>"posts"}
   edit_post GET    /posts/:id/edit(.:format)    {:action=>"edit", :controller=>"posts"}
      post GET    /posts/:id(.:format)         {:action=>"show", :controller=>"posts"}
      post PUT     /posts/:id(.:format)         {:action=>"update", :controller=>"posts"}
      post DELETE  /posts/:id(.:format)         {:action=>"destroy", :controller=>"posts"}
Macintosh-10:blog heileman$
```

# The View

- Finally, let's make the postings look better, and also get the site ready for some JavaScript. We'll do this by creating some partial templates (or more simply “partials”).
- If multiple pages contain the same snippets of HTML code, you can make a partial out of this snippet, and share it with all these pages — this certainly supports the DRY principle.
- In addition, this modularization of the markup makes it easier to replace pieces of a page using JavaScript.
- Let's start by creating a partial for posts. In `./app/views/posts`, create a file called `_post.html.erb`, and put the following in it:

```
<%= div_for @post do %>
  <h2><%= link_to_unless_current h(@post.title) %></h2>
  <%= simple_format h(@post.body) %>
<% end %>
```

# The View

- The `div_for` method will create a HTML div tag using the post and it's id for the id attribute.
- The `h` method is a helper method that is used to format HTML. Use this to help prevent XSS attacks.
- Now, in `./app/views/posts/show.html.erb`, delete everything above the Comments h2 heading tag, and replace it with:

```
<%= render :partial => @post %>
```

- If you now browse to a posting, you should see what you saw before, but now we're using a partial.
- Let's create a partial for comments too, put the following in `./app/views/comments/_comment.html.erb`:

```
<%= div_for comment do %>
  <p><strong>
    Posted <%= time_ago_in_words(comment.created_at) %>
  </strong><br />
  <%= h(comment.body) %>
</p>
<% end %>
```

- Now, edit the `./app/views/posts/show.html.erb` file so that it uses this partial (leave the form as is):

```
<h2>Comments</h2>
  <div id="comments">
    <%= render :partial => @post.comments.reverse %>
  </div>
```

- Notice that the comments array is rendered in reverse order (the most recent comment will show up first), and that we're creating one div for the entire set of comments, and that that partial is creating a div for each comment.
- When you render the partial, you're associating an array of comments with it, but Rails is smart enough to invoke the `_comment.html.erb` partial on each comment in the array.
- Again, browse to the posts to make sure everything is still working.

# The Comments Controller

- You'll find that when you try to create a comment for a post, the validations catch the fact that the comment you're trying to create does not have a `post_id`.
- To fix this, we need to add the `post_id` for the current post when we create the comment. To do this, edit the `create` method in the `comments_controller`, deleting the line:

```
@comment = Comment.new(params[:comment])
```

and replacing it with:

```
@post = Post.find(params[:post_id])
```

```
@comment = @post.comments.create!(params[:comment])
```

- Do you understand why the `params` hash has both a `:post_id` and a `:comment`, and how they were set?
- **Note:** `create!` calls `save!` (rather than `save` in the case of `create`) and `save!` will raise an exception if the record you're trying to create is invalid (`save` will not raise an exception).

# Protecting the Posts

- Right now anyone can edit your posts, by simply pointing their browser to: `http://localhost:3000/posts/1/edit`.
- To fix this, let's edit `./app/controllers/posts_controller.rb`. At the top of the `PostsController` class add:

```
before_filter :authenticate, :except => [:index, :show]
```

Before filters may halt the request cycle. The code in the before filter will now be run before any action in the `posts_controller` is executed. If the `before_filter` returns false, the requested action will not be executed. At the bottom of the `PostsController` class add:

```
private
def authenticate
  authenticate_or_request_with_http_basic do |name, password|
    name == "admin" && password == "secret"
  end
end
```



# Protecting the Comments

- We could also add some authentication to the comments, but notice that we don't really need to give users access to any of the actions in the `comments_controller`, except for `new` and `create`, and that this is actually done through the `posts_controller`.
- Thus, we can delete all of the other actions in the `comments_controller` (everything except `new` and `create`), along with all of the files in the view folder associated with comments.
- Finally, if we edit the `./config/routes.rb` file, removing the line that deals with comments:

```
resources :comments
```

the user won't be able to navigate via their browser to the new comment action.

# Adding some JavaScript

- Rather than reloading the page every time someone adds a new comment, we'll use JavaScript to update only the comments div on that page.
- Starting with Rails 3.1, JQuery is the default JavaScript library that is installed when you create a new Rails application.
- JQuery supports the notion of “unobtrusive JavaScript”. I.e., that the behavior of an HTML object should not be specified in the HTML file itself, but rather in a separate file. This concept of separating behavior from form is a very important design concept. We'll also see the notion of separation of presentation from form when we discuss CSS.
- Together these imply that an HTML file should only consider the form of a document, not its styling (which should be specified in .css files) or the behavior of its elements (which should be specified in .js files).

# Adding some JavaScript

- Let's add some JavaScript to our code using the JQuery library — this can make web applications more interactive.
- Specifically, let's make the form for posts use JavaScript whenever a new comment is added. Change the form submission in show.html.erb so that it reads:

```
<% form_for [@post, Comment.new], :remote => true do |f| %>
```

All we did was add:

```
:remote => true
```

This tells Rails to use Ajax when the form is submitted (more about Ajax in a future lecture).

- What this means is that when a new comment is created, the view can provide some Javascript code that can be executed, rather than reloading the entire page.
- But we need to set things up properly.

# Adding some JavaScript

- First, create a file called `./app/views/comments/create.js.erb`, and put the following JQuery code in it:  

```
$('#comments').prepend("<%= escape_javascript(render(:partial => @comment))%>");  
  
$('#new_comment')[0].reset();
```
- The first line (which should all appear on one line) renders the comment partial we previously created, and appends it to the div that has the id “comments”, and the second line resets the form.
- In order to make it all work, we just need to let the create action in the comments controller know that it should render JavaScript. To do this, modify the create action in `./app/controllers/comments_controller.rb` as follows:

```
if @comment.save  
  format.html { <don't change this line> }  
  format.json { <don't change this line> }  
  format.js # create.js.erb  
else
```

All we did was add the `format.js` line.

# Adding some JavaScript

- Restart the web server and test your application, and add a comment. It's hard to tell whether or not JavaScript is working, but if it is, the entire page was NOT reloaded, only one comment div was dynamically added.
- You can use FireBug to check that the POST request used when creating a comment is using JavaScript.
- You could also add some JavaScript effects to make this more apparent to the end user. Replace the `./app/views/comments/create.js.erb` file with the following, and see what happens:

```
var new_comment = $("<%= escape_javascript(render  
                                (:partial => @comment))%>").hide();  
$('#comments').prepend(new_comment);  
$('#comment_<%= @comment.id %>').fadeIn('slow');  
$('#new_comment')[0].reset();
```

This will no doubt change your life, so enjoy!

# Degrading Gracefully

- If someone does not have Javascript enabled in their browser, your application should degrade gracefully to HTML.
- Recall that we modified the create action in the `comments_controller` so that it would handle JavaScript:

```
if @comment.save
  format.html { ... }
  format.json { ... }
  format.js # create.js.erb
else
```

- If JavaScript is not enabled, the `respond_to` block will instead use the HTML option, and redirect to show action in the `post_controller`.
- But we don't need to stop here, we can support other formats as well. Let's add support for XML, JSON, and even RSS feeds.

# Supporting Other Formats

- Modify the `respond_to` block of the `index` action in your `posts_controller` so that it looks like:

```
respond_to do |format|  
  format.html # index.html.erb  
  format.json { render json: @posts }  
  format.xml { render xml: @posts }  
  format.atom #index.atom.builder  
end
```

Now, when you list posts, you'll have the option of receiving XML or JSON data back (we'll look at the Atom protocol on the next slide).

- To verify this, browse to:

`http://localhost:3000/posts.xml`

and

`http://localhost:3000/posts.json`

and see what is returned.

# Supporting Other Formats

- Let's syndicate this blog! That's purpose of the `format.atom` line on the previous slide.
- This will output data using the Atom Syndication Format (an XML format used for web feeds). Rails will try to render this data using the file: `./app/views/posts/index.atom.builder`, so let's create one (please see the next slide).

- Finally, in the `./app/views/layouts/application.html.erb` file, add the following to the head section:

```
<%= auto_discovery_link_tag :atom, posts_path(:format => :atom) %>
```

This will place an icon for the feed in the browser window when a user views the posts.

- Now, if you browse to:

`http://localhost:3000/posts`

(or any other page associated with this application) you should see an icon for the syndication feed. Click on it to see the feed.



# Supporting Other Formats

- Put the following in the `./app/views/posts/index.atom.builder` file:

```
atom_feed do |feed|
  feed.title("Greg's World")
  feed.updated(@posts.first.created_at)

  @posts.each do |post|
    feed.entry(post) do |entry|
      entry.title(post.title)
      entry.counter(post.body, :type => 'html')
      entry.author { |author| author.name("Greg")}
    end
  end
end
```

- Note:** The Atom Publishing Protocol (AtomPub) is an application-level protocol that uses HTTP. To learn more, see: <http://bitworking.org/projects/atom/rfc5023.html>.