## Associations

INFO 2310:
Topics in Web Design and Programming

~

## Most of today

How do we add a related model to already existing models?

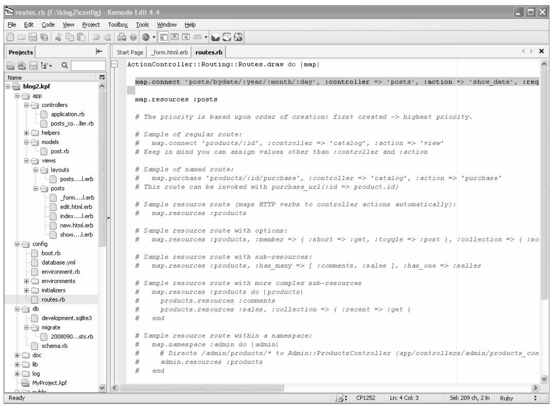
But first…

## Using an IDE

For your convenience, we've had Komodo Edit 4.4 installed on all the lab machines.

It makes navigating all your files a bit easier.

For Mac users, everyone I know seems to swear by TextMate.


```
ActionController::Routing::Routes.draw do |map|

  map.connect 'posts/bydate/:year/:month/:day', :controller => 'posts', :action => 'show_date', :req

  map.resources :posts

  # The priority is based upon order of creation: first created -> highest priority.

  # Sample of regular route:
  #   map.connect 'products/:id', :controller => 'catalog', :action => 'view'
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  #   map.purchase 'products/:id/purchase', :controller => 'catalog', :action => 'purchase'
  # This route can be invoked with purchase_url(:id => product.id)

  # Sample resource route (maps HTTP verbs to controller actions automatically):
  #   map.resources :products

  # Sample resource route with options:
  #   map.resources :products, :member => { :short => :get, :toggle => :post }, :collection => { :so

  # Sample resource route with sub-resources:
  #   map.resources :products, :has_many => [ :comments, :sales ], :has_one => :seller

  # Sample resource route with more complex sub-resources
  #   map.resources :products do |products|
  #     products.resources :comments
  #     products.resources :sales, :collection => { :recent => :get }
  #   end

  # Sample resource route within a namespace:
  #   map.namespace :admin do |admin|
  #     # Directs /admin/products/* to Admin::ProductsController (app/controllers/admin/products_con
  #     admin.resources :products
  #   end
```

To make a 'project' from your blog, do "File/New Project From Template".

Select the Ruby on Rails template (from "Common").

Name your project something ("blog", maybe) and select as its directory the current directory for your blog.

Komodo Edit is smart enough to realize that all the files in that directory belong to your project.

## Some nice things

You can run some Rails commands from inside the editor.

With your project open, look inside the folder 'Rails Tools'.

Try 'Run/run server'.

## Partials

## Partials

With 'partials', we can create 'partial' views that can be rendered inside other views.  A bit like a PHP 'include'.

The file name of a partial is prefixed with an '_'.

## Let's try one…

Notice that app/views/posts/new.html.erb and app/views/posts/edit.html.erb are almost identical.

Let's capture the common part in a partial.

## _form.html.erb

Create a new file 'app/views/posts/_form.html.erb'.

Copy the following from 'app/views/posts/edit.html.erb' into '_form':

```
<% form_for(@post) do |f| %>
<%= f.error_messages %>
<p>
  <%= f.label :title %><br />
  <%= f.text_field :title %>
</p>
 <p>
  <%= f.label :body %><br />
   <%= f.text_area :body %>
</p>
<p>
<%= f.submit "Create" %>
</p>
<% end %>
```

Now edit blog/app/views/posts/edit.html.erb by replacing the removed code with:

```
<%= render :partial => 'form' %>
```

and the same for blog/app/views/posts/new.html.erb.

Now try the blog…

ruby Script/server
http://localhost.3000/

## Problem…

The submit button says ‘Create’ for both entering a new entry and editing an old one.

We can solve this by passing in local variables to each…

## Edits

_form.html.erb; change f.submit line to
```
<%= f.submit action %>
```

new.html.erb; change render line to
```
<%= render :partial => 'form', :locals =>
  {:action => 'Create'} %>
```

edit.html.erb; change render line to
```
<%= render :partial => 'form', :locals =>
  {:action => 'Update'} %>
```

## Adding a model

Now we’ll get down to the business of adding another model to our site. We need to:

- Create the model and any associations with other models.
- Create the associated controller.
- Create the associated views.
- Update the database.
- Update the routes.

## Scaffolding

We can get a lot of this done via scaffolding; this will set up the files to create the model/controllers/views, just like we did last time for the posts model.

Go to your blog directory, and enter “`ruby script/generate scaffold comment body:text post_id:integer`”.

(Through Komodo: Ruby Tools/Generate/scaffold).

## Adding a table

Rails lets us modify our DB through *migrations*: Ruby code that explains the changes to make to our DB and then how to undo them.

## The migration

This creates a file in db/migrate called 2008xxxxxxx_create_comments.rb. You'll see another file in that directory, 2008xxxxx_create_posts.rb, which created the posts DB.

## create_comments

```
class CreateComments < ActiveRecord::Migration
  def self.up
    create_table :comments do |t|
      t.integer :post_id
      t.text :body

      t.timestamps
  end

  def self.down
    drop_table :comments
  end
end
```

## Migrating

To actually get this to run and create the table, we need to run '`rake db:migrate`'.

(Through Komodo: Rails Tools/Migrate/db:migrate).

We can use migrations to move back and forth between various versions of the DB if needed.

## Users

OK, let's scaffold out another model (and views and controllers) for users. Users will have a name, password, and email, all strings.

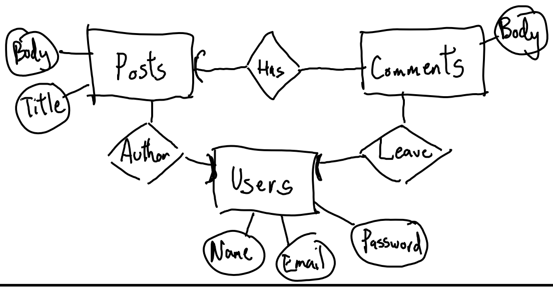
You do it, this time… user

Be sure to run the migration to create the table.

## Associations

We now want to be able to tell the models about the various connections that they have with each other.

## ER Diagram



We can have these relationships reflected in the models by adding information to the model files.

Open up the post model in app/models/post.rb. We can tell the post about associated comments.

```
class Post < ActiveRecord::Base
  has_many :comments
  validates_presence_of :title, :body
  validates_format_of :title, :with =>
  /^[\w\d]+$/
end
```

Because each comment has a post_id, Rails can automatically associate each comment with a particular post.

We can also tell the model how to associate itself with another model (if we don't follow the defaults).

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :author, :class_name =>
     "User", :foreign_key => "user_id"
  validates_presence_of :title, :body
  validates_format_of :title, :with =>
     /^[\w\d]+$/
end
```

Let's now enter the associations for other models as well.

```
class Comment < ActiveRecord::Base
  belongs_to :post
end

class User < ActiveRecord::Base
  has_many :posts
end
```

The association of the Post model with the User model isn't going to work quite yet; why not?

```
class Post < ActiveRecord::Base
  has_many :comments
  belongs_to :author, :class_name =>
     "User", :foreign_key => "user_id"
  validates_presence_of :title, :body
  validates_format_of :title, :with =>
     /^[\w\d]+$/
end
```

We need to add a column to the posts DB that tells us the user_id of the author of the post.

We can do this through a migration. Type ‘`ruby script/generate migration add_user_id`’ (or Rails Tools/Generators/migration in Komodo).

In db/migrations/2008xxxx_add_user_id, create a migration to add a column to the posts DB as follows:

```
class AddUserId < ActiveRecord::Migration
  def self.up
    add_column :posts, :user_id, :integer
  end

  def self.down
    remove_column :posts, :user_id
  end
end
```

Go ahead and run the migration (you do remember how to do that, right?)

Now you can go ahead and add comments and users by starting up the web server (`ruby script/server`) and navigating to http://localhost:3000/users and http://localhost:3000/comments.

This doesn’t let us enter/edit the user_ids associated with the posts because the associated views with posts are still the same. We can fix this by updating the views.

In app/views/posts/_form.html.erb, add two lines to allow the user id to be input/edited.

```
<% form_for(@post) do |f| %>
  <%= f.error_messages %>

  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.label :user_id %><br />
    <%= f.text_field :user_id %>
  <p>
    <%= f.submit action %>
  </p>
<% end %>
```

Go ahead and enter some users, some comments and posts associated with those users.

Now we can see how the associations work by starting up the console (`ruby script/console`).

```
@post = Post.find(:first)
@user = User.find(:first)
@post.comments
@user.posts.size
@post.author
@user.posts.create(:title => “New post
  through association”, :body => “It
  knows the user who made it!”)
```

## Partial collections

This is a little unsatisfying so far. Every post has an associated set of comments, but we’re entering them and editing them as independent entities linked by a post_id.

We like it to work like a real blog.

One idea: we want the URL posts/3 to list all the comments associated with post #3.

We can do this by editing the associated view and having it display the associated comments (@post.comments).

We could loop through these with a for loop, but instead let’s use partials again.

Create a file app/views/comments/_comment.html.erb.

Put in it something like this:

```
<div class="comment">
  <h5><%= comment.created_at.to_formatted_s(:long) %></h5>
  <p><%=h comment.body %></p>
</div>
```

Now we can render all the comments by adding this line to app/views/posts/show.html.erb.

```
<%= render :partial =>
  “/comments/comment", :collection =>
  @post.comments %>
```