Creating Todo List with Ruby On Rails
written using Ruby 1.8.4 with Rails 1.0

by thehua | beginner's guide | revision 1.0 - 3/2/2006

Ruby on Rails(**RoR**) is a rapid application development framework. Ruby is the language, Rails is the framework. The videos on rubyonrails.org are awe-inspiring and can convince any web developer to switch - or at the very least think about jumping ship. I, for one, became extremely interested in this and quickly hopped on board. Everything seemed so perfect - server setup was quick and simple, ruby on rails integration smooth, and even the initial *Hello World* was easy - then came the hard part, coding an application.

RoR is great except for one thing: lack of great tutorials. Sure onlamp.com has a few, but those are 1 sql table, scaffolding, and template editing tutorials. For the real data mining and business logic executing capabilities of RoR, one would really need to buy a book. (I highly recommend Agile Web Development with Rails - buy pdf online so you can get free lifetime updates of the book.) But for those of us who either can't afford or wish to merely play around with the language first - what should we do?

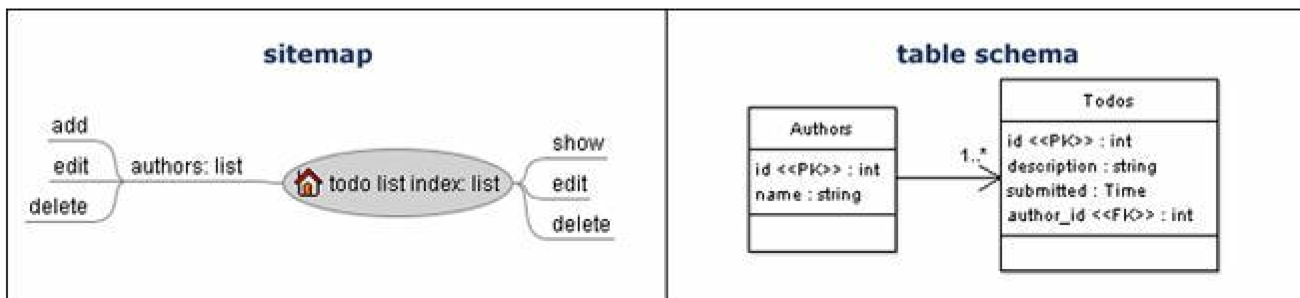
That's where I hope to help. To go beneath the surface of RoR, I've developed an up-to-date tutorial on creating a **Todo List** ruby on rails application. Sure there are other to-do list tutorials online, but I've found most are quite out of date and do not apply to rails 1.0 or beyond -- and they do not include relational database manipulations.

This tutorial is targeted towards Ruby on Rails beginners and I hope to answer most trivial questions encountered when first working with this framework. Please feel free to leave errata and comments.

Before we being, make sure you have a working development environment.
Jim Rutherford has created a wonderful step-by-step tutorial on installing Fedora Core 4 + Lighttpd + Fastcgi + RoR + MySQL . This is the environment I used and I highly recommend - either for development and/or production. (Keep in mind, production servers require hardened security permissions which I will not go into here.)

Overview

To demonstrate Ruby on Rails, we will create a simple todo list. We will use a good design priciple of site organization: display todo lists as the primary layout and keep author management as secondary. In addition, to best create todo lists we will use a database schema that will contain two tables: one holding todo items, the other holding the authors' information. Please see Fg 1.0 for details. We will refer to this structure throughout this tutorial.



Fg 1.0
(table schema represented using UML 2.0)

Creating the Foundation

First we need to initialize a RoR environment. Go to your working directory and type:

```
[ shell ]$ rails todolist
```

This creates a ruby environment - todolist- and generates skeleton files necessary for basic application deployment . As we step through the tutorial, I will explain in more detail what some of these sub-directories mean.

Create a test database in your MySQL and grant some test user. For simplicity sake, the schema below is what I used.

Now that the skeleton is created we need to create an appropriate database schema. Edit the appropriate database connection in file *config/database.yml*, and set appropriate connection details.

```
# config/database.yml

development:
  adapter: mysql
  database: ruby_test
  username: ruby_test
  password: ruby_test_password
  socket: /var/lib/mysql/mysql.sock
  host: localhost
```

You'll notice that *database.yml* contains numerous connection definitions. RoR allows different environments such as development, production, etc. This tutorial assumes you'll remain in the development environment and will not talk about deploying to a production state.

Once you've defined the database connection, we need to establish an appropriate database schema for handling our todo list data. Luckily RoR provides the `rake migrate` script which will allow for easy deployment of schema. We need to create individual schema files for every table required.

```
[ shell ]$ script/generate migration authors
```

The migration script generates a skeleton file for the authors table. Edit this file in *db/migrate/001_authors.rb*

```
# db/migrate/001_authors.rb

class Authors < ActiveRecord::Migration
  def self.up
    create_table :authors do |table|
      table.column :id, :integer
      table.column :name, :string
    end
  end

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

Similarly, create a skeleton file for todos table.

```
[ shell ]$ script/generate migration todos
```

Edit the script in *db/migrate/002_todos.rb*

```
# db/migrate/002_todos.rb

class Todos < ActiveRecord::Migration
  def self.up
    create_table :todos do |table|
      table.column :id, :integer
      table.column :description, :text
      table.column :submitted, :timestamp
      table.column :author_id, :integer
    end
  end

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

Now that the table schemas are created, we will need to tell RoR to invoke the `rake migrate` script. This script takes the created schema, appends a version system internally, and executes the sql scripts.

```
[ shell ]$ rake migrate
```

What's great about this script is that it allows for version rollbacks. The `self.down` action drops the associated table when you execute command `rake migrate VERSION=0`.

Connecting RoR with the Database

Now that the skeleton files are successfully created, we should move on and actually have our application do something. In other traditional languages, here comes the hard part of connecting code to the databases, fetch objects, then display it. Thankfully, with RoR, since it's a **MVC** (model, view, controller) rapid application development framework, it's extremely easy for it to connect with the backend database. We simply invoke the `scaffold` script. Scaffolding builds our required MVC components based on the underlying schema.

```
[ shell ]$ script/generate scaffold todo todos
```

The script above is identified as: `script/generate scaffold model_name controller_name`. In the MVC design principle: model is based off of modeling the underlying database structure, controller is used to manipulate the model, view is used to display the interactive user element. For more information on MVC please refer to wikipedia.

> *Caution! Ruby on Rails has a very particular naming convention. SQL Tables are be **plural**. Models are **singular**. Controllers are **plural**. Often when you try to run a script and forget about plurality, RoR will attempt to automatically pluralize or singularize for you - this can be annoying.*

Our First Test

Just like that our code is ready for first run. Yes - you heard right, there's nothing else you need to do - well, except start the server.

I recommend trying RoR's built in WEBrick server (in a new console window) first - unless you are comfortable with Lighttpd to run dispatch.fcgi script (again refer to Jim Rutherford's guide).

```
[ shell ]$ script/server
```

This starts the internal WEBrick's server, serving data out on the default port 3000. Now point your browser to *http://server_ip:3000/todos*, i.e. *http://127.0.0.1:3000/todos* for localhost, or in your private lan perhaps *http://192.168.1.2:3000/todos*.

> *(Make sure you have allowed port 3000 to be open on your default installation of Fedora Core 4. I won't go into much detail here since it's not related, but edit /etc/sysconfig/iptables, allow port 3000, then restart /etc/init.d/iptables.)*

You should see a fully functional website giving you list, show, edit, and delete operations on your todos table. (See Fg 2.0) Feel free to play around by adding and deleting items. Scaffold script has made all the simple MVC components so you can easily manipulate this data. Do keep in mind, scaffold only created a skeleton which we can freely edit and modify anytime.

Fg 2.0

Putting Relations in RoR

As cool as our current application is, it doesn't yet perform all the functionalities we planned. In order to do so, we need to add the key ingredient, relations. Based on our overview, we have determined that multiple authors can each have multiple todo lists. However, each todo item can only have 1 author. To reflect those relationships in our application, we need to edit our models' files.

We already have our todo model generated when we scaffolded todos controller. However, we still need to generate the author model.

```
[ shell ]$ script/generate model author
```

Open and edit *app/models/author.rb*

```
# app/models/author.rb

class Author < ActiveRecord::Base
  validates_presence_of :name
  has_many :todos, :order => "submitted"
end
```

`has_many :todos, :order => "submitted"` tells RoR that author has many-to-one relationship with todos table. `validates_presence_of :name` is a special form validation command that enforces the author's name field cannot be blank when inserting.

Next we need to establish the todo item to author relationship. Open and edit *app/models/todo.rb*

```
# app/models/todo.rb

class Todo < ActiveRecord::Base
  validates_presence_of :description
  belongs_to :author
end
```

`belongs_to :author` says there's one-to-many relationship to author model.
`validates_presence_of :description` again is a validation command so the field of description in todos table is not blank.

(Form validation methods can vary drastically and go beyond the purpose of this tutorial.)

Continuing on, since our relations are now formed, we need to find a way to display those results. As apparent from testing just the todos controller, the basic templates provided by the scripts are very basic. RoR organizes all user interfaces in the app/views directory. We'll begin by modifying the *app/views/todos/new.rhtml* view. This rhtml file is what RoR uses to generate the look. The basics of *.rhtml files are made up of html (and any other web related code) with RoR code inserted using <% %> and <%= %>. This enclosure method is siblings with php and asp counter parts.

```
# app/views/todos/new.rhtml

<h1>New todo</h1>
<%= start_form_tag :action => 'create' %>
  <%= render :partial => 'form' %>

  <b>Author:</b><br/>
  <select name="todo[author_id]">
  <% @authors.each do |author| %>
  <option value="<%= author.id %>">
  <%= author.name %>
  </option>
  <% end %>
  </select>

  <%= submit_tag "Create" %>
  <%= end_form_tag %>

<%= link_to 'Back', :action => 'list' %>
```

The inserted code appends into the default generated new view. This code extracts a list of authors from the authors table using a combination of html <select> and ruby code to iterate through table data. However, in order for this views page to work we need to change the controller so information is passed onto new.rhtml.

You might have wondered how exactly will pages magically obtain content. RoR handles variables labeled using '@'. `@authors.each do |authors`, for example, takes the @authors variable and uses it based the author model we generated earlier. We can control @authors when passed into the view by editing *app/controllers/todos_controller.rb*

```
# app/controllers/todos_controller.rb

...
def new
  @todo = Todo.new
  @authors = Author.find_all
end
...
```

While we have the todos_controller.rb open, we might as well add to the edit section as well.

```
# app/controllers/todos_controller.rb

...
def edit
  @todo = Todo.find(params[:id])
  @authors = Author.find_all
end
...
```

Having completed those modifications, we should update edit.rhtml in similar fashion as new.rhtml. Please modify *app/views/todos/edit.rhtml*

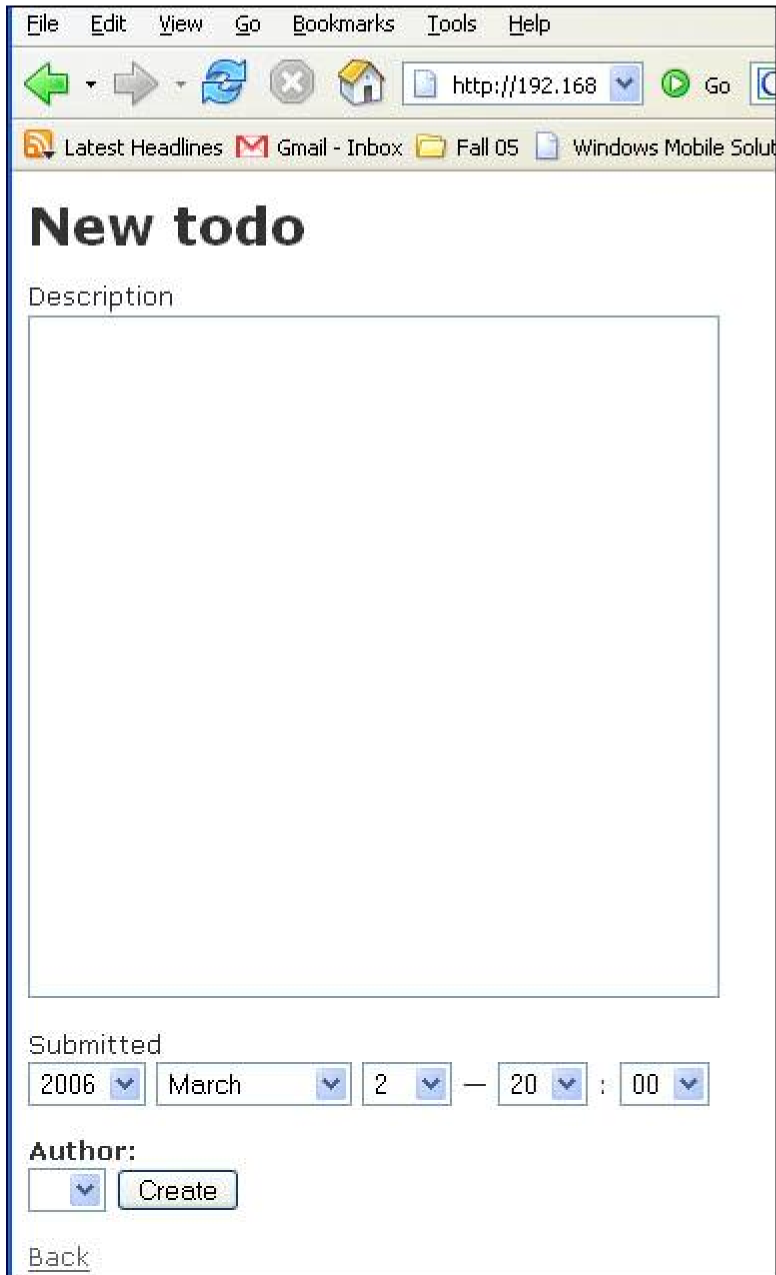
```
# app/views/todos/edit.rhtml

<h1>Editing todo</h1>
<%= start_form_tag :action => 'update', :id => @todo %>
  <%= render :partial => 'form' %>

  <b>Author:</b><br/>
  <select name="todo[author_id]">
  <% @authors.each do |author| %>
  <option value="<%= author.id %>">
  <%= author.name %>
  </option>
  <% end %>
  </select>

  <%= submit_tag 'Edit' %>
<%= end_form_tag %>
<%= link_to 'Show', :action => 'show', :id => @todo %> |
<%= link_to 'Back', :action => 'list' %>
```

We've modified edit.rhtml and new.rhtml with a drop down box populated with the authors in our authors table.

Let's test it.

Navigate to *http://server_url:3000/todos/new*
You should see the default new template with an additional drop down selection box - which is currently empty. (See Fg 3.0) Before you think it's not working - take a step back and think about why the drop down menu is blank. The page is pulling data directly from the database. Now our database is still only a skeleton - that explains it! Please open up your preferred method of inserting SQL queries and add a few authors to the Authors table.

> *(Unfortunately this tutorial relies on an outside SQL query method for the time being. I might update tutorial to use script/console in the future.)*



Fg 3.0

Once you add authors, the new page magically populates that data into the selection menu.

What's the point of all this if - sure we can add and edit todo list items - but we can't even see who they belong to? Well...we certainly can see which item belongs to whom by simply editing - you guessed it - *app/views/todos/list.rhtml*

```
# app/views/todos/list.rhtml

<h1>Todo List</h1>
<table>
  <tr>
  <th>Author</th>
  <% for column in Todo.content_columns %>
  <th><%= column.human_name %></th>
  <% end %>
  </tr>

  <% for todo in @todos %>
  <tr>
```

```
  <td><%= todo.author.name %></td>

  <% for column in Todo.content_columns %>
  <td><%=h todo.send(column.name) %></td>
  <% end %>

  <td><%= link_to 'Show', :action => 'show', :id => todo %></td>
  <td><%= link_to 'Edit', :action => 'edit', :id => todo %></td>
  <td><%= link_to 'Destroy', { :action => 'destroy', :id => todo }, :confirm => 'Are you sure?'
%></td>
  </tr>
  <% end %>
</table>
<%= link_to 'Previous page', { :page => @todo_pages.current.previous } if
@todo_pages.current.previous %>
<%= link_to 'Next page', { :page => @todo_pages.current.next } if @todo_pages.current.next %>
<br />
<%= link_to 'New todo', :action => 'new' %>
```

It's just that simple. (See Fg 3.1) Since we've created the relations in our models, by appending code `<%= todo.author.name %>`, on application run RoR automatically fetches the todo item's author. In the same respect, let's edit *app/views/todos/show.rhtml*

```
# app/views/todos/show.rhtml

<% for column in Todo.content_columns %>
  <p>
  <b><%= column.human_name %>:</b> <%=h @todo.send(column.name) %>
  </p>
  <% end %>
  <p>
  <b>Author:</b> <%= @todo.author.name %>
  </p>
<%= link_to 'Edit', :action => 'edit', :id => @todo %> |
<%= link_to 'Back', :action => 'list' %>
```

Try the test site now and you should see authors' names right next to each todo entry item. (See Fg 3.2)

Fg 3.1

Fg 3.2

The Final Relation

It's great that we can finally modify all the components of todo list items but I'd really like to be able to modify the authors as well. Thankfully, with the scaffold tool, similar to how to todos controller was created, we can create a controller for the authors too.

```
[ shell ]$ script/generate scaffold author authors
```

We include `scaffold author authors` because we've already generated an author model - remember this script says please scaffold a controller authors using model author. Now scaffold recognizes our existing model and will generate MVC accordingly.

If we navigate to *http://server_url:3000/authors* we see just what we expected, a controller to allow modifications of Authors table. (See Fg 4.0)

Fg 4.0

We could quit now and have a perfectly working system. However, we're forgetting about one business logic. Our todo list items rely on having a relationship to an author. We can't just delete an author if he or she already has a todo item, can we? Unfortunately, RoR does not, and cannot, automatically check this relationship, so yes right now you can accidentally delete the author and cause the application to crash.

To prevent this relation from breaking, we need to insert code into app/controllers/authors_controller.rb

```
# app/controllers/authors_controller.rb

...
def destroy
  if(Todo.find(:first, :conditions => [ "author_id = ?", params[:id]]) != nil)
    flash[:notice] = 'Author currently has todo lists.  Cannot delete.'
    list
    render :action => 'list'
  else
    Author.find(params[:id]).destroy
    redirect_to :action => 'list'
  end
end
...
```

The code above works by checking the todo item passed in to the destroy action and runs an SQL query to check to see if any todo items belong to this author. If there is at least one todo item, the author cannot be deleted and the action returns to list. However if the author has no todo items, it is safe to delete. (See Fg 4.1)

Fg 4.1

Conculsion

Now you have successfully created your first? relational Ruby on Rails application, I hope you've enjoyed the ease and quickness of creating such a diverse application. (If this were written in php, we've had tons more code and unnecessary complexity).

I didn't go into very much in depth about customizing the look of these pages - by all means, feel free to play around with both *app/views/todos/*.rhtml* and the newly added *app/views/authors/*.rhtml* also the css styles in *public/* . I personally linked todo list directly to author management controls by adding a simple hyperlink to */authors*.

Ruby on Rails is a very powerful and simple rapid application development tool. It takes sometime adjusting to this type of environment especially if you come from a php or asp background like myself. We are all learning more and more everyday and I hope to learn from your wonderful RoR experiences in the future.

Cheers.

Some Helpful References:
http://api.rubyonrails.org/
http://developer.apple.com/tools/rubyonrails.html
http://www.digitalmediaminute.com/howto/fc4rails/
http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

Creative Commons Attribution-NonCommercial 2.5 License