Four Days on Rails

compiled by John McCreesh

Table of Contents

Introduction

There have been many extravagant claims made about Rails. For example, an article in OnLAMP.com[1] claimed that "you could develop a web application at least ten times faster with Rails than you could with a typical Java framework..." The article then goes on to show how to install Rails and Ruby on a PC and build a working 'scaffold' application with virtually no coding.

While this is impressive, 'real' web developers know that this is smoke and mirrors. 'Real' applications aren't as simple as that. What's actually going on beneath the surface? How hard is it to go on and build 'real' applications?

This is where life gets a little tricky. Rails is well documented on-line – in fact, possibly too well documented for beginners, with over 30,000 words of on-line documentation in the format of a reference manual. What's missing is a roadmap (railmap?) pointing to the key pages that you need to know to get up and running in Rails development.

This document sets out to fill that gap. It assumes you've got Ruby and Rails up on a PC (if you haven't got this far, go back and follow Curt's article). This takes you to the end of 'Day 1 on Rails'.

'Day 2 on Rails' starts getting behind the smoke and mirrors. It takes you through the 'scaffold' code. New features are highlighted in bold**,** explained in the text, and followed by a reference to either Rails or Ruby documentation where you can learn more.

'Day 3 on Rails' takes the scaffold and starts to build something recognisable as a 'real' application. All the time, you are building up your tool box of Rails goodies. Most important of all, you should also be feeling comfortable with the on-line documentation so you can continue your explorations by yourself.

'Day 4 on Rails' adds in another table and deals with some of the complexities of maintaining relational integrity. At the end, you'll have a working application, enough tools to get you started, and the knowledge of where to look for more help.

Ten times faster? after four days on Rails, judge for yourself!

Documentation: this document contains highlighted references, either to:

- *Documentation* – the Rails documentation at http://api.rubyonrails.com
- *Ruby Documentation* – "Programming Ruby - The Pragmatic Programmer's Guide" available online and for download at http://www.ruby-doc.org/docs/ruby-doc-bundle/ProgrammingRuby/index.html

Acknowledgements: many thanks to the helpful people on the the irc channel[2] and the mailing list[3]. The on-live archives record their invaluable assistance as I clawed my way up the Rails and Ruby leaning curves.

Version: 1.7 using version 0.10 of Rails – see http://rails.homelinux.org for latest version and to download a copy of the ToDo code.

1 *Rolling with Ruby on Rails,* Curt Hibbs 20-Jan2005 http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html
2 irc://irc.freenode.org/rubyonrails
3 http://lists.rubyonrails.org/mailman/listinfo/rails

Day 1 on Rails

The ToDo List application

This document follows the building of a simple 'ToDo List' application – the sort of thing you have on your PDA, with a list of items, grouped into categories, with optional notes (for a sneak preview of what it will look like, see Illustration 4 Main 'To Do' screen on page 17).

Running the Rails script

This example is on my MS-Windows PC. My web stuff is at `c:\www\webroot`, which I label as drive w: to cut down on typing:

```
C:\> subst w: c:\www\webroot
C:\> w:
W:\> rails ToDo
W:\> cd ToDo
W:\ToDo>
```

Running `rails ToDo` creates the following directory structure below `ToDo\`:

```
app
   Holds all the code that's specific to this particular application.
app\controllers
   Holds controllers which drive the program logic
app\models
   Holds models which describe the data structures, validation and integrity rules,
   etc.
app\views
   Holds the template files which form the basis of the rendered html pages. This
   directory can also be used to keep stylesheets, images, and so on that can be
   symlinked to public.
app\helpers
   Holds view helpers (common pieces of code)
config
   Configuration files for Apache, database, and other dependencies.
lib
   Application specific libraries. Basically, any kind of custom code that doesn't
   belong in controllers, models, or helpers. This directory is in the load path.
log
   Application specific logs. Note: development.log keeps a trace of every action Rails
   performs – very useful for error tracking, but does need regular purging!
public
   The directory available for Apache, which includes iamges, javascripts, and
   stylesheets subdirectories
script
   Helper scripts for automation and generation.
test
   Unit and functional tests along with fixtures.
vendor
   External libraries that the application depend on. This directory is in the load
   path.
```

Adding the Application to the Web Server

As I'm running everything (Apache2, MySQL, etc) on a single development PC, the next two steps give a friendly name for the application in my browser.

Defining the Application in the hosts file

```
C:\winnt\system32\drivers\etc\hosts (excerpt)

127.0.0.1 todo
```

Defining the Application in the Apache Configuration file

```
Apache2\conf\httpd.conf

<VirtualHost *>
  ServerName todo
  DocumentRoot /www/webroot/ToDo/public
  <Directory /www/webroot/ToDo/public/>
    Options ExecCGI FollowSymLinks
    AllowOverride all
    Allow from all
    Order allow,deny
  </Directory>
</VirtualHost>
```

Switching to fastcgi

Unless you are patient (or have a powerful PC) you should enable fastcgi for this application

```
public\.htaccess

# Change extension from .cgi to .fcgi to switch to FCGI and to .rb to switch to
mod_ruby
RewriteBase /dispatch.fcgi
```

Checking that Rails is working

The site should now be visible in your browser as `http://todo/`

Setting up the Database

I've set up a new database called '`todos`' in MySQL. Connection to the database is specified in the `config\database.yml` file

```
app\config\database.yml (excerpt)

development:
  adapter: mysql
  database: todos
  host: localhost
  username: foo
  password: bar
```

Creating the Categories Table

The `categories` table is used in the examples that follow. It's simply a list of categories that will be used to group items in our ToDo list.

MySQL definition

```
Categories table

CREATE TABLE `categories` (
  `id` smallint(5) unsigned NOT NULL auto_increment,
  `category` varchar(20) NOT NULL default '',
  `created_on` timestamp(14) NOT NULL,
  `updated_on` timestamp(14) NOT NULL,
  PRIMARY KEY  (`id`),
  UNIQUE KEY `category_key` (`category`)
) TYPE=MyISAM COMMENT='List of categories';
```

Some hints and gotchas for table and field naming:

- underscores in field names will be changed to spaces by Rails for 'human friendly' names
- beware mixed case in field names – some parts of the Rails code have case sensitivities
- every table should have a primary key called 'id' - in MySQL it's easiest to have this as numeric auto_increment
- links to other tables should follow the same '_id' naming convention

- Rails will automatically maintain fields called `created_at/created_on` or `updated_at/updated_on`, so it's a good idea to add them in

Documentation: ActiveRecord::Timestamp

- Useful tip: if you are building a multi-user system (not relevant here),Rails will also do optimistic locking if you add a field called `lock_version (integer default 0)`. All you need to remember is to include `lock_version` as a hidden field on your update forms.

Documentation: ActiveRecord::Locking

Data Model

Generate an empty file:

```
W:\ToDo>ruby script/generate model Category
```

which simply creates `app\modules\category.rb`

Scaffold

The controller is at the heart of a Rails application.

Running the generate controller script

```
W:\ToDo>ruby script/generate controller category
```

which creates two files and two empty directories:

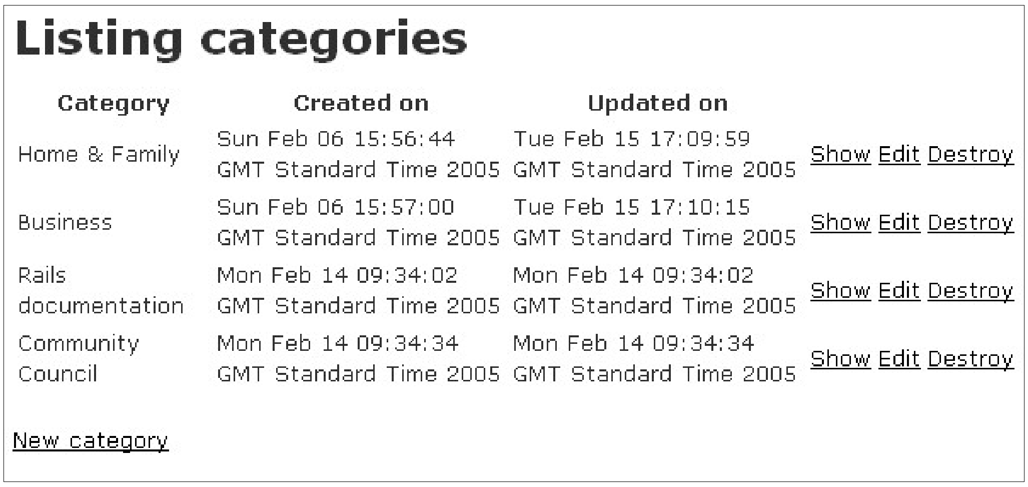
```
app\controllers\category_controller.rb
app\helpers\category_helper.rb
app\views\categories
app\views\layouts
```

If you haven't already seen the model / scaffold trick in operation in a beginner's tutorial like *Rolling with Ruby on Rails,* try it now and amazed yourself how a whole web app can be written in two lines of code:

app\controllers\category_controller.rb

```
class CategoryController < ApplicationController
  model :category
  scaffold :category
end
```

Point your browser at `http://todo/category` and be amazed at how clever it is :-)



Listing categories

Category	Created on	Updated on	
Home & Family	Sun Feb 06 15:56:44 GMT Standard Time 2005	Tue Feb 15 17:09:59 GMT Standard Time 2005	Show Edit Destroy
Business	Sun Feb 06 15:57:00 GMT Standard Time 2005	Tue Feb 15 17:10:15 GMT Standard Time 2005	Show Edit Destroy
Rails documentation	Mon Feb 14 09:34:02 GMT Standard Time 2005	Mon Feb 14 09:34:02 GMT Standard Time 2005	Show Edit Destroy
Community Council	Mon Feb 14 09:34:34 GMT Standard Time 2005	Mon Feb 14 09:34:34 GMT Standard Time 2005	Show Edit Destroy

New category

Illustration 1: Scaffold screen

Day 2 on Rails

To progress beyond this point, we need to see what's happening behind the scenes. With the scaffold action, Rails generates all the code it needs dynamically. By running scaffold as a script, we can get all the code written to disk where we can investigate it and then start tailoring it to our requirements.

Running the generate scaffold script

```
W:\ToDo>ruby script/generate scaffold Category
```

This script generates a range of files needed to create a complete application, including a controller, views, layouts, and even a style sheet:

```
app\controllers\categories_controller.rb
app\helpers\categories_helper.rb
app\views\categories\edit.rhtml
app\views\categories\list.rhtml
app\views\categories\new.rhtml
app\views\categories\show.rhtml
app\views\layouts\categories.rhtml
public\stylesheets\scaffold.css
```

Note the slightly bizarre naming convention – we've moved from the singular to the plural, so to use the new code you need to point your browser at `http://todo/categories`.

The Model

The Model is where all the data-related rules are stored, including data validation and relational integrity. This means you can define them once, and Rails will automatically apply them wherever the data is accessed.

Creating Data Validation Rules

Rails gives you a lot of error handling for free (almost). To demonstrate this, add some validation rules to the empty Category model:

app\models\category.rb

```
class Category < ActiveRecord::Base
  validates_length_of :category, :within => 1..20
  validates_uniqueness_of :category, :message => "already exists"
end
```

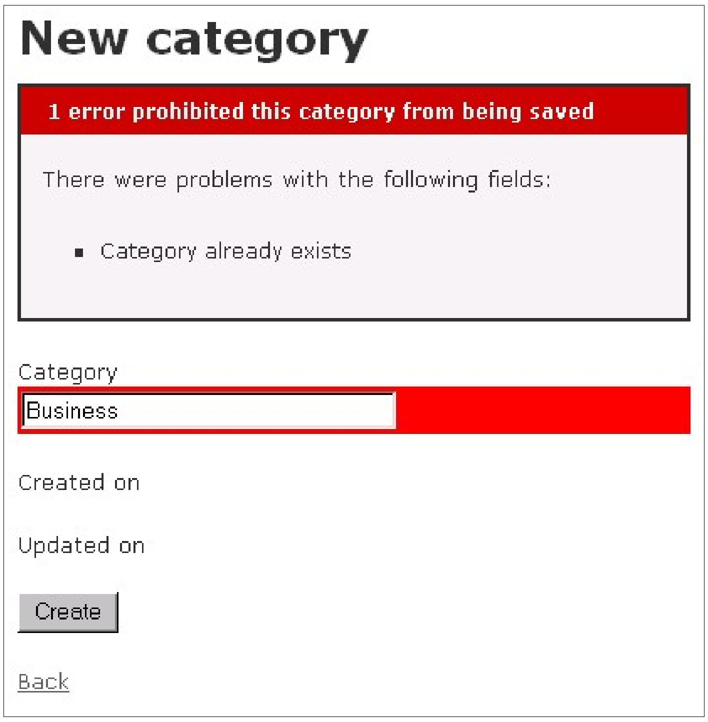
These entries will give automatic checking that:

- `validates_length_of`: the field is not blank and not too long
- `validates_uniqueness_of`: duplicate values are trapped

Documentation: ActiveRecord::Validations::ClassMethods

To try this out, try and insert a duplicate record (see *Illustration 2: Data Validation* below). The style is a bit in your face – it's not the most subtle of user interfaces. However, what do you expect for free?

Note: try the same test with the previous version `http://todo/category` version. The auto-rendered scaffold code can't cope with data validation. To prevent confusion, it's probably safer to delete the `app\controllers\category_controller.rb` and `app\helpers\category_helper.rb` files.


New category

1 error prohibited this category from being saved

There were problems with the following fields:

- Category already exists

Category

Business

Created on

Updated on

Create

Back

Illustration 2: Data Validation

The Controller

Now it's time to look at the controller. The controller is where the programming logic for the application lies. It interacts with the user using views, and with the database through models. You should be able to read the controller and see how the application hangs together.

The default Controller

The controller produced by the generate scaffold script is listed below

\app\controllers\categories_controller.rb

```
class CategoriesController < ApplicationController
  def index
    list
    render_action 'list'
  end

  def list
    @categories = Category.find_all
  end

  def show
    @category = Category.find(@params['id'])
  end

  def new
    @category = Category.new
  end

  def create
    @category = Category.new(@params['category'])
    if @category.save
      flash['notice'] = 'Category was successfully created.'
      redirect_to :action => 'list'
    else
      render_action 'new'
    end
  end
  def edit
    @category = Category.find(@params['id'])
  end

  def update
    @category = Category.find(@params['category']['id'])
    if @category.update_attributes(@params['category'])
      flash['notice'] = 'Category was successfully updated.'
      redirect_to :action => 'show', :id => @category.id
    else
      render_action 'edit'
    end
  end

  def destroy
    Category.find(@params['id']).destroy
    redirect_to :action => 'list'
  end
end
```

- The default action for the controller is to render a template matching the name of the action – e.g. the `list` action will populate the `@categories` instance variable and then the controller will render "`list.rhtml`".
- `render_template` allows you to render a different template – e.g. the `index` action will run the code for `list`, and will then render `list.rhtml` rather than `index.rhtml` (which doesn't exist)
- `redirect_to` goes one stage further, and uses an external "302 moved" HTTP response to loop back into the controller – e.g. the `destroy` action doesn't need to render a template. After performing its main purpose (destroying a category), it simply takes the user to the `list` action.

Documentation: ActionController::Base

The controller uses ActiveRecord methods such as `find, find_all, new, save, update_attributes`, and `destroy` to move data to and from the database tables.

Documentation: ActiveRecord::Base

Notice how several of the actions are split into two. For example, when the user selects `edit`, the controller extracts the record they want to edit from the model, and then renders the edit.view. When the user has finished editing, the edit view invokes the `update` action, which updates the model and then invokes the `show` action.

Tailoring the default Controller

Personally, I don't like the way Rails displays the `show` screen next – I prefer to go straight back to the `list` screen; the `show` screen isn't necessary in this application. However, it would be nice to display a message to say the edit has worked:

```
app\controllers\categories_controller.rb (excerpt)

  def update
    @category = Category.find(@params['category']['id'])
    if @category.update_attributes(@params['category'])
      flash['notice'] = 'Category was successfully updated.'
      redirect_to :action => 'list'
    else
      render_action 'edit'
    end
  end
```

The `flash` message will be picked up and displayed on the next screen to be displayed – in this case, the `list` screen (see *Tailoring the default 'List' View* on page 12).

Documentation: ActionController::Flash

Curiously, although the flash message has its own css tag, the stylesheet produced by the `generate scaffold` script doesn't do anything special with it. This is solved by a simple addition:

```
public\stylesheets\scaffold.css  (excerpt)

.notice {
    color: red;
}
```

The View

Views are where the user interface are defined. Rails can render the final HTML page presented to the user from three hierarchical components:

Layout	**Template**	**Partials**
in `app\views\layouts\` default: `application.rhtml` or `<controller>.rhtml`	in `app\views\<controller>\` default: `<action>.rhtml`	in `app\views\<controller>\` default `_<partial>.rhtml`

Layout

Rails Naming convention: if there is a template in `app\views\layouts\` with the same name as the current controller then it will be automatically set as that controller's layout unless explicitly told otherwise.

A layout with the name `application.rhtml` or `application.rxml` will be set as the default controller if there is no layout with the same name as the current controller and there is no layout explicitly assigned.

The layout generated by the scaffold script looks like this:

```
app\views\layouts\categories.rhtml

<html>
<head>
  <title>Scaffolding: <%= controller.controller_name %>#<%= controller.action_name
%></title>
  <link href="/stylesheets/scaffold.css" rel="stylesheet" type="text/css" />
</head>
<body>

<%= @content_for_layout %>

</body>
</html>
```

This is mostly HTML, but there isn't very much of it :-) The sections in bold are the key to the Rails rendering process:

- `controller_name` and `action_name` are ActionController methods which return parts of the URL which are displayed in the browser address bar.

Documentation: ActionController::Base

- `@content_for_layout` allows a single standard layout to have dynamic content inserted at rendering time based on the action being performed (e.g. 'edit', 'new', 'list'). This dynamic content comes from a template.

Documentation: ActionController::Layout::ClassMethods.

Templates

Rails naming convention: templates are held in `app\views\categories\'action'.rhtml`. For example, the `edit.rhtml` created by the scaffold script is given below:

```
app\views\categories\edit.rhtml

<h1>Editing category</h1>
```

```
<%= error_messages_for 'category' %>
<%= form 'category', :action => 'update' %>

<%= link_to 'Show', :action => 'show', :id => @category.id %> |
<%= link_to 'Back', :action => 'list' %>
```

This code for the 'edit' action is all that is required for Rails to render the complete HTML for an edit page. The magic is all in the bold type:

Displaying Errors trapped by the Data Model

`error_messages_for` returns a string with marked-up text for any error messages produced by a previous attempt to submit the form. If one or more errors is detected, the HTML looks like this:

```
<div class="errorExplanation" id="errorExplanation">
  <h2>n errors prohibited this user from being saved</h2>
  <p>There were problems with the following fields:</p>
  <ul>
    <li>field_1 error_message_1</li>
    <li>... ...</li>
    <li>field_n error_message_n</li>
  </ul>
</div>
```

Note: the css tags match corresponding statements in the stylesheet created by the generate scaffold script.

Documentation: ActionView::Helpers::ActiveRecordHelper

Creating a Form with minimal coding

`form` is Rails at its most economical. Given an Active Record Object, it renders an entire form. The following code:

```
<%= form 'category', :action => 'update' %>
```

will create all this HTML:

```
<form action="/categories/update" method="post">
<input id="category_id" name="category[id]" type="hidden" value="n" />
<p>
  <label for="category_category">Category</label><br />
  <input id="category_category" name="category[category]" size="30"
    type="text" value="y" />
</p>
<input type="submit" value="Update" /></form>
```

It's not the prettiest user interface ever created, but it works, and you can't get much quicker.

Documentation: ActionView::Helpers::ActiveRecordHelper

Creating Links

`link_to` simply creates a link – the most fundamental part of HTML...

Documentation: ActionView::Helpers::UrlHelper

Tailoring the default 'Edit' View

The default HTML produced by the `form` helper is functional, but if we want more control over the layout, we need to take more control over the HTML by editing the `rhtml` file. In this example, we want to use a table to line up the prompts for user input in front of the input boxes. This doesn't mean abandoning the Rails toolbox altogether:

```
app\views\categories\edit.rhtml

<h1>Rename Category</h1>

<%= error_messages_for 'category' %>
```

```
<form action="/categories/update" method="post">
  <%= hidden_field "category", "id" %>
  <table>
    <tr>
      <td><b>Category:</b></td>
      <td><%= text_field "category", "category", "size" => 20, "maxlength" => 20
%></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Update" />
</form>

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

`hidden_field, text_field` are quick ways to generate the corresponding HTML constructs.

Documentation: ActionView::Helpers::FormHelper

Tailoring the default 'List' View

app\views\categories\list.rhtml

```
<h1>Categories</h1>
<% if @flash["notice"] %>
<span class="notice">
  <%= @flash["notice"] %>
</span>
<% end %>
<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>
<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to 'Rename', :action => 'edit', :id => category.id %></td>
    <td><%= link_to 'Delete', { :action => 'destroy', :id => category.id }, :confirm
=> 'Are you sure you want to delete this category?' %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'Add new category', :action => 'new' %>
```

Escaping HTML Characters

One of the problems with allowing users to input data which is then displayed on the screen is that they could accidentally (or maliciously) type in code which could break the system when it was displayed. For example, think what would happen if a user types in '</table>' as a category.

To guard against this, it is good practice to html_escape any data which has been provided by users. This means that e.g. </table> is rendered as &lt;/table&gt; which is harmless.

Rails makes this really simple – just add an 'h':
`<%=h category["category"] %>`

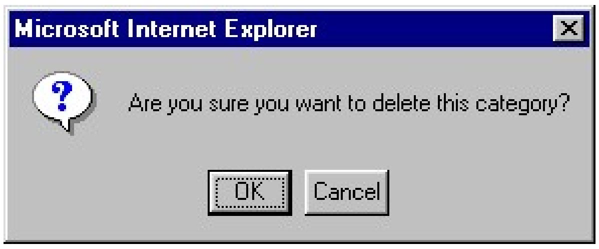
Using Ruby to format Date and Time

I've had to hard code this page manually so I can use a Ruby method `strftime()` to format the date and time fields the way I want them.

Ruby Documentation: class Time

Creating a Javascript confirmation Dialogue

Note also the use of a Javascript pop-up box in `link_to` to `:confirm` the delete before processing:



Are you sure you want to delete this category?

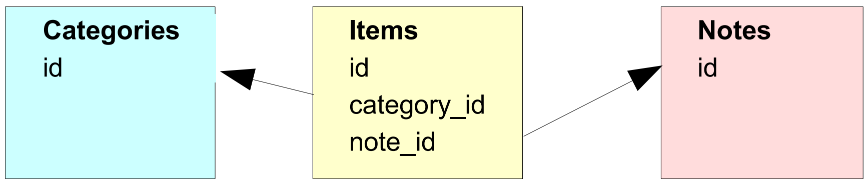
Illustration 3 Javascript Confirmation Dialogue

Documentation: ActionView::Helpers::UrlHelper

That takes us to the end of Day 2. We have a working system for maintaining our Categories table, and have started to take control of the scaffold code which Rails has generated.

Day 3 on Rails

Now it's time to start on the heart of the application. The Items table contains the list of 'todos'. Every Item may belong to one of the Categories we created on Day 2. An Item optionally may have one Note, held in a separate table, which we will look at tomorrow. Each table has a primary key 'id', which is also used to record links between the tables.



Let's generate some more scaffold code. We'll do this for both the Items table and the Notes table. We aren't ready to work on Notes as yet, but having the scaffold in place means we can refer to Notes in today's coding without generating lots of errors. Just like building a house – scaffolding allows you to build one wall at a time without everything crashing around your ears.

```
W:\ToDo>ruby script/generate scaffold Item
W:\ToDo>ruby script/generate scaffold Note
```

Note: this will empty any previous item or note files without warning

The 'Items' Table

MySQL table defintion

The fields in the Items table are as follows:

- done - 1 means the ToDo item has been completed[4]
- priority – 1 (high priority) to 5 (low priority)
- description – free text stating what is to be done
- due_date – stating when it is to be done by
- category_id – a link to the Category this item comes under ('id' in the Categories table)
- note_id – a link to an optional Note explaining this item ('id' in the Notes table)
- private – 1 means the ToDo items is classed as 'Private'

```
Items table

CREATE TABLE items (
  id smallint(5) unsigned NOT NULL auto_increment,
  done tinyint(1) unsigned NOT NULL default '0',
  priority tinyint(1) unsigned NOT NULL default '3',
  description varchar(40) NOT NULL default '',
  due_date date default NULL,
  category_id smallint(5) unsigned NOT NULL default '0',
  note_id smallint(5) unsigned default NULL,
  private tinyint(3) unsigned NOT NULL default '0',
  created_on timestamp(14) NOT NULL,
  updated_on timestamp(14) NOT NULL,
  PRIMARY KEY  (id)
) TYPE=MyISAM COMMENT='List of items to be done';
```

The Model

```
app\models\item.rb

class Item < ActiveRecord::Base
```

4 MySQL doesn't have a 'boolean' type, so we have to use 0/1

```
  belongs_to :category
  validates_associated :category
  validates_format_of :done_before_type_cast, :with => /[01]/, :message=>"must be 0 or
1"
  validates_inclusion_of :priority, :in=>1..5, :message=>"must be between 1 (high) and
5 (low)"
  validates_presence_of :description
  validates_length_of :description, :maximum=>40
  validates_format_of :private_before_type_cast, :with => /[01]/, :message=>"must be 0
or 1"
end
```

Validating Links between Tables

- the use of `belongs_to` and `validates_associated` links the Items table with the item_id field in the Category table.

Documentation: ActiveRecord::Associations::ClassMethods

Validating User Input

- `validates_presence_of` protects 'NOT NULL fields against missing user input
- `validates_format_of` uses regular expressions to check the format of user input
- when a user types input for a numeric field, Rails will always convert it to a number – if all else fails, a zero. If you want to check that the user has actually typed in a number, then you need to validate the input `_before_type_cast`, which lets you access the 'raw' input[5].
- `validates_inclusion_of` checks user input against a range of permitted values
- `validates_length_of` prevents the user entering data which would be truncated when stored[6].

Documentation: ActiveRecord::Base

More on Views

Sharing Variables between the Templates and the Layout

By now, it is becoming obvious that all my templates will have the same first few lines of code, so it makes sense to move this common code into the layout. Delete all the `app\views\layouts\*.rhtml` files, and replace with a common `application.rhtml`.

```
app\views\layouts\application.rhtml

<html>
<head>
  <title><%=h @heading %></title>
  <link href="/stylesheets/ToDo.css" rel="stylesheet" type="text/css" />
</head>
<body>
<h1><%=h @heading %></h1>
<% if @flash["notice"] %>
<span class="notice">
  <%= @flash["notice"] %>
</span>
<% end %>
<%= @content_for_layout %>
</body>
</html>
```

I've renamed the `public/stylesheets/acaffold.css` to `ToDo.css` for tidiness, and also generally played with colours, table borders, to give a prettier layout. However, returning to Rails, note how the `heading` variable is shared between the two files, which means that you can have content in the layout dynamically defined by a template:

5 What might seem a more obvious alternative: `validates_inclusion_of :done_before_type_cast, :in=>"0".."1", :message=>"must be between 0 and 1"` – fails if the input field is left blank
6 You could however combine the two rules for the Description field into one: `validates_length_of :description, :within => 1..20`

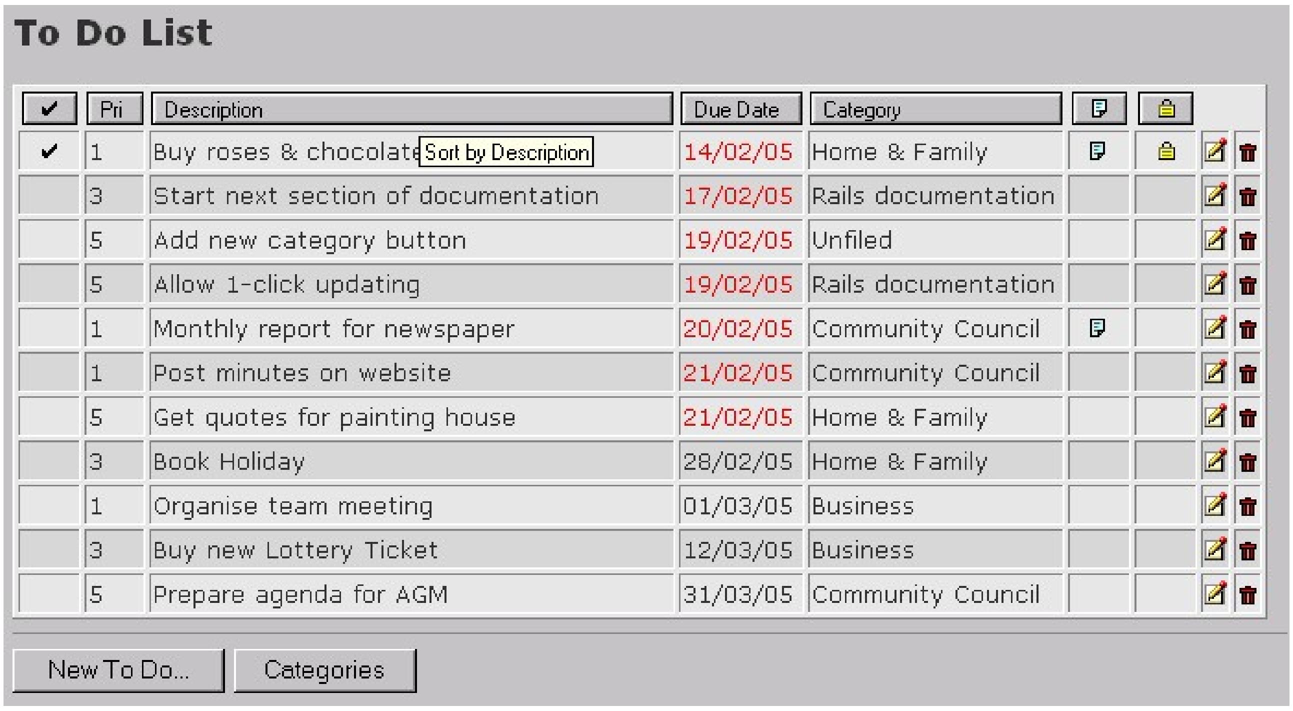
```
app\views\items\edit.rhtml (excerpt)

<% @heading = "Edit To Do" %>
<%= error_messages_for 'item' %>
<%= form 'item', :action => 'update' %>
```

The ToDo List screen

What I'm trying to do is a look based on a PalmPilot or similar PDA desktop. The end product is shown in Illustration 4 Main 'To Do' screen[7].

Some points:

- clicking on the 'tick' (✓) column heading will purge all the completed items (those marked with a tick)
- the display can be sorted by clicking on the 'Pri', 'Description', 'Due Date', and 'Category' column headings
- the 0/1 values for 'Done' are converted into a little 'tick' icon
- items past their due date are coloured red
- the presence of an associated note is shown by 'note' icon
- the 0/1 values for 'Private' are converted into a padlock symbol
- individual items can be edited or deleted by clicking on the icons on the right of the screen
- the display has a nice 'stripey' effect
- new items can be added by clicking on the 'New...' button at the bottom of the screen.



To Do List

✓	Pri	Description	Due Date	Category	🗒	🔒		
✓	1	Buy roses & chocolat	14/02/05	Home & Family	🗒	🔒	✎	🗑
	3	Start next section of documentation	17/02/05	Rails documentation			✎	🗑
	5	Add new category button	19/02/05	Unfiled			✎	🗑
	5	Allow 1-click updating	19/02/05	Rails documentation			✎	🗑
	1	Monthly report for newspaper	20/02/05	Community Council	🗒		✎	🗑
	1	Post minutes on website	21/02/05	Community Council			✎	🗑
	5	Get quotes for painting house	21/02/05	Home & Family			✎	🗑
	3	Book Holiday	28/02/05	Home & Family			✎	🗑
	1	Organise team meeting	01/03/05	Business			✎	🗑
	3	Buy new Lottery Ticket	12/03/05	Business			✎	🗑
	5	Prepare agenda for AGM	31/03/05	Community Council			✎	🗑

Illustration 4 Main 'To Do' screen

The template used to achieve this is built up as follows:

```
app\views\items\list.rhtml

<% @heading = "To Do List" %>
<form action="/items/new" method="post">
<table>
  <tr>
    <th><%= link_to_image "done", {:controller => 'items', :action =>
"purge_completed"}, :confirm => "Are you sure you want to permanently delete all
completed To Dos?" %></th>
    <th><%= link_to_image "priority",{:controller => 'items', :action =>
```

7 It's amazing what a few lines in a stylesheet can do to change the appearance of a screen, plus of course a collection of icons...

```
"list_by_priority"}, "alt" => "Sort by Priority" %></th>
    <th><%= link_to_image "description",{:controller => 'items', :action =>
"list_by_description"}, "alt" => "Sort by Description" %></th>
    <th><%= link_to_image "due_date", {:controller => 'items', :action => "list"},
"alt" => "Sort by Due Date" %></th>
    <th><%= link_to_image "category", {:controller => 'items', :action =>
"list_by_category"}, "alt" => "Sort by Category" %></th>
    <th><%= show_image "note" %></th>
    <th><%= show_image "private" %></th>
    <th>&nbsp;</th>
    <th>&nbsp;</th>
  </tr>
<%= render_collection_of_partials "list_stripes", @items %>
</table>
  <hr />
  <input type="submit" value="New To Do..." />
  <input type="button" value="Categories" onClick="parent.location='<%= url_for(
:controller => 'categories', :action => 'list' ) %>'">
</form>
```

Purging completed 'ToDos' by clicking on an icon

Clickable images are created by `link_to_image`, which by default expects to find an image in `pub/images` with a .png suffix; clicking on the image will run the specified method[8].

Adding in the `:confirm` parameter generates a javascript pop-up dialogue box as before.

Documentation: ActionView::Helpers::UrlHelper

Clicking 'OK' will invokes the `purge_completed` method. This new `purge_completed` method needs to be defined in the controller:

```
app\controllers\items_controller.rb (excerpt)

  def purge_completed
    Item.destroy_all "done = 1"
    redirect_to :action => 'list'
  end
```

`Item.destroy_all` deletes all the records in the `Items` table where the value of the field `done` is 1, and then reruns the `list` action.

Documentation: ActiveRecord::Base

Changing the Sort Order by clicking on the Column Headings

Clicking on the Pri icon invokes a `list_by_priority` method. This new `list_by_priority` method needs to be defined in the controller:

```
app\controllers\items_controller.rb (excerpt)

  def list
    @items = Item.find_all (nil,'due_date,priority')
  end

  def list_by_priority
    @items = Item.find_all (nil,'priority,due_date')
    render_action 'list'
  end
```

We've specified a sort order for the default `list` method, and created a new `list_by_priority` method[9]. Note: the first parameter in `find_all` is for specifying conditions (the 'WHERE' clause in SQL) – we want all the records returned here, so this parameter is 'nil'.

8 Note how I've explicitly specified the controller here: link_to_image "done", {:controller => 'items', :action => "purge_completed"},.... The simpler form: link_to_image "done", "purge_completed",... would also work for now ... I'll return to why I've done this on *Links on the Home Page* on page 31)

9 `list_by_description` and `list_by_category` are similar and are left as an easy exercise for the reader.

Documentation: ActiveRecord::Base

Note also that we need to explicitly `render_action 'list'`, as by default Rails would try to render a template called `list_by_priority` (which doesn't exist :-)

Adding a Helper

The headings for the Note and Private columns are images, but are not clickable. I decided to write a little method `show_image(name)` to just show the image:

app\helpers\items_helper.rb

```
module ItemsHelper
    def self.append_features(controller)
      controller.ancestors.include?(ActionController::Base) ?
        controller.add_template_helper(self) : super
    end

    def show_image(src)
       img_options = { "src" => src.include?("/") ? src : "/images/#{src}" }
       img_options["src"] = img_options["src"] + ".png" unless
img_options["src"].include?(".")
       img_options["border"] = "0"
       tag("img", img_options)
    end
end
```

Once this helper has been linked in by the controller:

app\controllers\items_controller.rb (excerpt)

```
class ItemsController < ApplicationController
  helper :Items
  def index
    list
    render_action 'list'
  end
```

it is available for the template.

Documentation: ActionView::Helpers

Using Javascript Navigation Buttons

`onClick` is a standard Javascipt technique for handling button actions such as navigating to a new web page. However, Rails goes to great lengths to rewrite pretty URLS, so we need to ask Rails for the correct URL to use. Given a `module` and an `action` `url_for` will return the URL...

Documentation: ActionController::Base

Partials – sub-templates

I wanted to create a nice stripey effect for the list of items. *Partials* enable a section of formatting to be delegated to a sub-template. They can either be invoked by the `render_partial` method:

```
<% for item in @items %>
  <%= render_partial "list_stripes", item %>
<% end %>
```

or by the more economical `render_collection_of_partials`:
```
render_collection_of_partials "list_stripes", @items
```

Either code (another Rails naming convention here) will invoke a sub-template `_list_stripes.rhtml` and pass to it the variable `item`.

Documentation: ActionView::Partials

Rails also passes a sequential number `list_stripes_counter` to the sub-template. This is the key to formatting alternate rows in the table with either a light grey background or a dark grey background. One way is simply to test whether the counter is odd or even: if odd, use light gray; if even, use dark gray..

A sub-template looks very similar to a template:

```
app\views\items\_list_stripes.rhtml

<tr class="<%= list_stripes_counter.modulo(2).nonzero?  ? "dk_gray" : "lt_gray" %>">
    <td><%= list_stripes["done"] == 1 ? show_image("done_ico.gif") : "&nbsp;" %></td>
    <td><%= list_stripes["priority"] %></td>
    <td><%=h list_stripes["description"] %></td>
<% if list_stripes["due_date"].nil? %>
   <td>&nbsp;</td>
<% else %>
    <%= list_stripes["due_date"] < Date.today ? '<td class="past_due">' : "<td>" %><%=
list_stripes["due_date"].strftime("%d/%m/%y") %></td>
<% end %>
    <td><%=h list_stripes.category ? list_stripes.category["category"] : "Unfiled"
%></td>
    <td><%= list_stripes["note_id"].nil? ? "&nbsp;" : show_image("note_ico.gif")
%></td>
    <td><%= list_stripes["private"]  == 1 ? show_image("private_ico.gif") : "&nbsp;"
%></td>
    <td><%= link_to_image("edit", { :controller => 'items', :action => "edit", :id =>
list_stripes.id }) %></td>
    <td><%= link_to_image("delete", { :controller => 'items', :action => "destroy",
:id => list_stripes.id }, :confirm => "Are you sure you want to delete this item?")
%></td>
</tr>
```

A little bit of Ruby is used to test if the counter is odd or even and render either `class="dk_gray"` or `class="lt_gray"`:
`list_stripes_counter.modulo(2).nonzero?  ? "dk_gray" : "lt_gray"`
the code as far as the first question mark asks: *is the remainder when you divide list_stripes_counter by 2 nonzero?*

Ruby Documentation: class Numeric

The remainder of the line is actually a cryptic *if then else* expression which sacrifices readability for brevity: *if the expression before the question mark is true, return the value before the colon; else return the value after the colon.*

Ruby Documentation: Expressions

The two tags `dk_gray` and `lt_gray` are then defined in the stylesheet:

```
public\stylesheets\ToDo.css (excerpt)

.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
```

Note: the same *if then else* construct is used to display the 'tick' icon if `list_stripes["done"]`equals 1, otherwise display an HTML blank space character:

```
list_stripes["done"] == 1 ? show_image("done_ico") : "&nbsp;"
```

Formatting based on Data Values

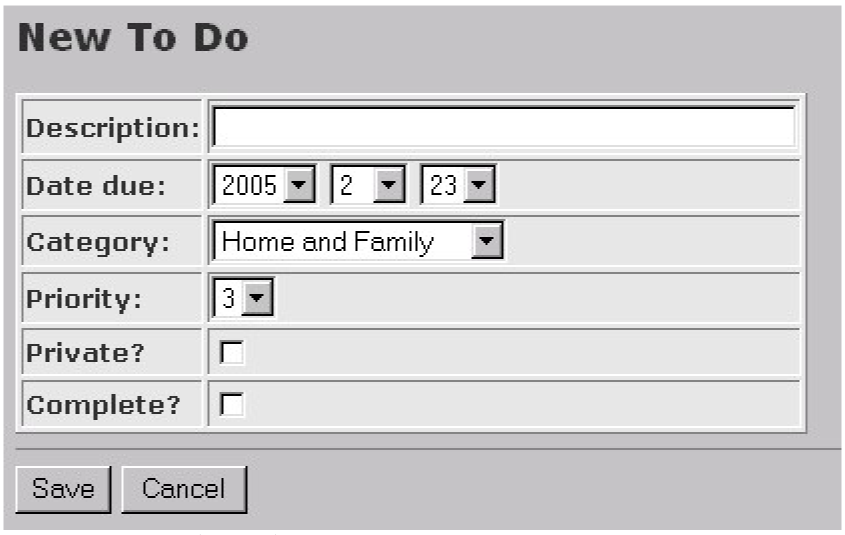
It's also easy to highlight specific data items – for example, dates in the past.
`list_stripes["due_date"] < Date.today ? '<td class="past_due">' : '<td>'`
Again, this needs a matching stylesheet entry.

Handling missing Values in a Lookup

We want the system to be able to cope with the situation where the user deletes a Category which is in use by ToDo items. In this case, the Category should be displayed as 'Unfiled':
`list_stripes.category ? list_stripes.category["category"] : 'Unfiled'`

The New ToDo Screen

Turning next to what happens when the 'New To Do...' button is pressed. Again, there are few new tricks lurking in the code.


New To Do

Description:	
Date due:	2005 / 2 / 23
Category:	Home and Family
Priority:	3
Private?	☐
Complete?	☐

Illustration 5 New 'To Do' screen

```
app\views\items\new.rhtml

<% @heading = "New To Do" %>
<%= error_messages_for 'item' %>
<form action="/items/create" method="post">
  <table>
    <tr>
      <td><b>Description: </b></td>
      <td><%= text_field "item", "description", "size" => 40, "maxlength" => 40
%></td>
    </tr>
    <tr>
      <td><b>Date due: </b></td>
      <td><%= date_select "item", "due_date", :use_month_numbers => true %></td>
    </tr>
    <tr>
      <td><b>Category: </b></td>
      <td><select id="item_category_id" name="item[category_id]">
        <%= options_from_collection_for_select @categories, "id", "category",
@item.category_id %>
          </select>
      </td>
    </tr>
    <tr>
      <td><b>Priority: </b></td>
      <% @item.priority = 3 %>
      <td><%= select "item","priority",[1,2,3,4,5]  %></td>
    </tr>
    <tr>
      <td><b>Private? </b></td>
      <td><%= check_box "item","private" %></td>
    </tr>
    <tr>
      <td><b>Complete? </b></td>
      <td><%= check_box "item", "done" %></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Save" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'">
```

```
</form>
```

Creating a Drop-down List for a Date Field

date_select generates a rudimentary drop-down menu for date input:

```
date_select "item", "due_date", :use_month_numbers => true
```

Documentation: ActionView::Helpers::DateHelper

Unfortunately it quite happily accepts dates like 31st February. Rails then dies when it tries to save this 'date' to the database. One workround is to trap this failed save using `rescue`, a Ruby exception handling method

```
app\controllers\items_controller.rb (excerpt)

  def create
     begin
      @item = Item.new(@params['item'])
      if @item.save
        flash['notice'] = 'Item was successfully created.'
        redirect_to :action => 'list_by_priority'
      else
        @categories = Category.find_all
        render_action 'new'
      end
     rescue
        flash['notice'] = 'Item could not be saved.'
        redirect_to :action => 'new'
     end
  end
```

Ruby Documentation: Exceptions, Catch, and Throw

Creating a Drop-down List from a Lookup Table

This is another example of Rails solving an everyday coding problem in an extremely economical way. In this example:

`options_from_collection_for_select @categories, "id", "category", @item.category_id`

`options_from_collection_for_select` reads all the records in categories and renders them as `<option value="`*`[value of id]`*`">`*`[value of category]`*`</option>`. The record that matches `@item_category_id` will be tagged as 'selected'. As is this wasn't enough, the code even html_escapes the data for you. Neat.

Documentation: ActionView::Helpers::FormOptionsHelper

Creating a Drop-down List from a List of Constants

This is a simpler version of the previous scenario. Hard-coding lists of values into selection boxes isn't always a good idea – it's easier to change data in tables than edit values in code. However, there are cases where it's a perfectly valid approach, so in Rails you do:

`select "item","priority",[1,2,3,4,5]`

Note also how to set a default value in the previous line of code.

Documentation: ActionView::Helpers::FormOptionsHelper

Creating a Checkbox

Another regular requirement; another helper in Rails:

`check_box "item","private"`

Documentation: ActionView::Helpers::FormHelper

Controller

Data driven drop down boxes have to get their data from somewhere – which has to be the controller

```
app\controllers\items_controller.rb (excerpt)

  def new
    @categories = Category.find_all
    @item = Item.new
  end
```

Finishing Touches

Tailoring the Stylesheet

At this point, the ToDo List screen should work, and so should the New ToDo button. To produce the screens shown here, I also made the following changes to the stylesheet:

```
public\stylesheets\ToDo.css

body { background-color: #c6c3c6; color: #333; }

h1 {
  font-family: verdana, arial, helvetica, sans-serif;
  font-size:   14pt;
  font-weight: bold;
}

table {
  background-color:#e7e7e7;
  border: outset 1px;
                border-collapse: separate;
                border-spacing: 1px;
}

td { border: inset 1px; }
.notice {
  color: red;
  background-color: white;
}
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
.hightlight_gray { background-color: #4a9284; }
.past_due { color: red }
```

The Edit ToDo Screen

The rest of Day 3 is taken up building the Edit ToDo screen, which is very similar to the New ToDo. I used to get really annoyed with college text books which stated: *this is left as an easy exercise for the reader,* so now it's great to be able to do the same to you[10].

Which takes us to the end of Day 3 – and the application now looks nothing like a Rails scaffold, but under the surface, we're still using a whole range of Rails tools to make development easy.

10 But unlike my college text book authors, I do reveal the answers on Day 4 :-) - see *app\views\items\edit.rhtml* on page 26

Day 4 on Rails

The 'Notes' table

The Model

This table contains a single free text field to hold further information for a particular ToDo Item. This data could of course have been held in a field on the Items table; however, if you do it this way you'll learn a lot more about Rails :-)

```
Notes table

CREATE TABLE notes (
  id smallint(6) NOT NULL auto_increment,
  more_notes text NOT NULL,
  created_on timestamp(14) NOT NULL,
  updated_on timestamp(14) NOT NULL,
  PRIMARY KEY  (id)
) TYPE=MyISAM COMMENT='Additional optional information for to-dos';
```

The model contains nothing new.

```
app\models\note.rb

class Note < ActiveRecord::Base
  validates_presence_of :more_notes
end
```

but we need to remember to add this link into the `Items` model:

```
app\models\item.rb (excerpt)

class Item < ActiveRecord::Base
  belongs_to :note
```

Using a Model to maintain Referential Integrity

The code we are about to develop will allow a user to add one Note to any Item. But what happens when a user deletes an Item which has an associated Note? clearly, we need to find a way of deleting the Note record too, otherwise we get left with 'orphaned' Notes records.

In the Model / View / Controller way of doing things, this code belongs in the Model. Why? well, we can delete Item records by clicking on the Dustbin icon on the ToDo' screen, but we can also delete them by clicking on Purge completed items. By putting the code into the Model, it will be run regardless of where the delete action comes from.

```
app\models\item.rb (excerpt)

  def before_destroy
    unless note_id.nil?
      Note.find(note_id).destroy
    end
  end
```

This reads: before you delete an Item record, find the record in Notes whose id equals the value of Note_id in the Item record you are about to delete, and delete it first. Unless there isn't one :-)[11]

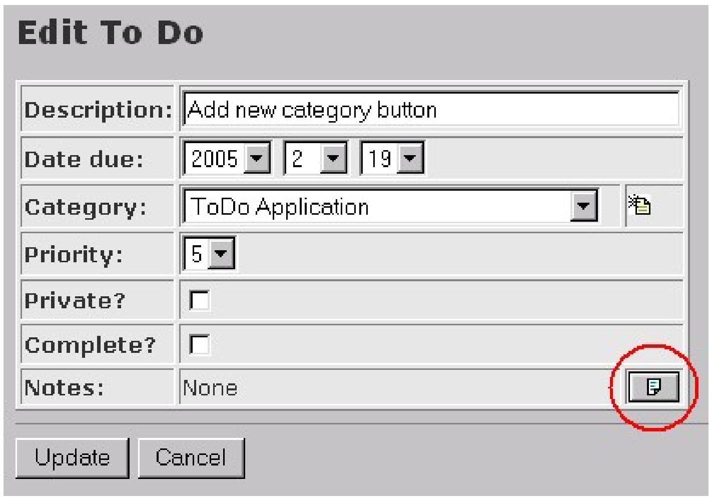
Documentation: ActiveRecord::Callbacks

The Views

Transferring the User between Controllers

Although the Notes scaffold code gives the full CRUD facilities, we don't want the user to invoke any of this directly. Instead, Notes can be created by clicking on the Notes button on the Edit ToDo screen:

11

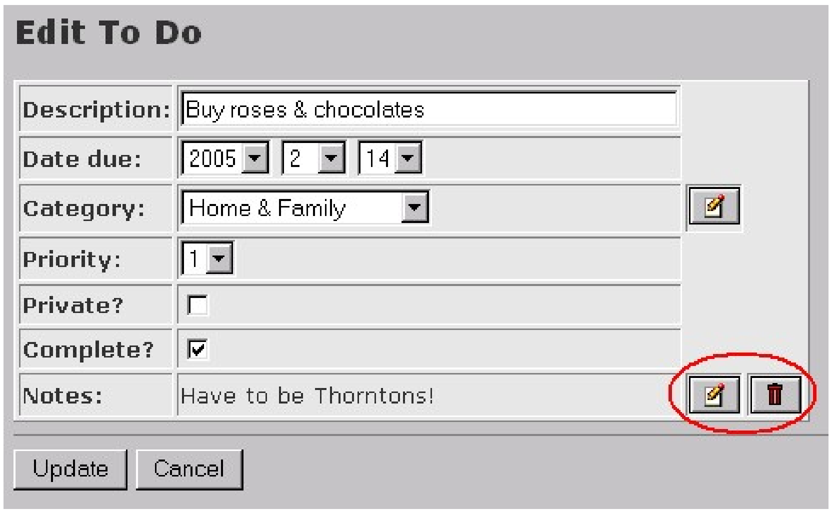


Edit To Do

Description:	Add new category button
Date due:	2005 2 19
Category:	ToDo Application
Priority:	5
Private?	☐
Complete?	☐
Notes:	None

Illustration 6: Creating a New Note from the Edit ToDo screen

and once a Note has been created, it can be edited or removed by clicking on the appropriate button:



Edit To Do

Description:	Buy roses & chocolates
Date due:	2005 2 14
Category:	Home & Family
Priority:	1
Private?	☐
Complete?	☑
Notes:	Have to be Thorntons!

Illustration 7: Editing or Deleting an existing Note

First of all, let's look at the code for the Edit ToDo screen. Note how the Notes buttons change according to whether a Note already exists, and how control is transferred to the Notes controller:

app\views\items\edit.rhtml

```
<% @heading = "Edit To Do" %>
<%= error_messages_for 'item' %>
<form action="/items/update" method="post">
  <%= hidden_field "item", "id" %>
  <table>
    <tr>
      <td><b>Description: </b></td>
      <td><%= text_field "item", "description", "size" => 40, "maxlength" => 40
%></td>
    </tr>
    <tr>
      <td><b>Date due: </b></td>
      <td><%= date_select "item", "due_date", :use_month_numbers => true %></td>
    </tr>
    <tr>
      <td><b>Category: </b></td>
      <td>
      <select id="item_category_id" name="item[category_id]">
        <%= options_from_collection_for_select @categories, "id", "category",
@item.category_id %>
      </select>
      </td>
      <td>
        <%= link_to_image 'edit_button', :controller => 'categories', :action =>
'list' %>
      </td>
    </tr>
    <tr>
      <td><b>Priority: </b></td>
      <td><%= select "item","priority",[1,2,3,4,5]  %></td>
    </tr>
    <tr>
      <td><b>Private? </b></td>
      <td><%= check_box "item","private" %></td>
    </tr>
    <tr>
      <td><b>Complete? </b></td>
      <td><%= check_box "item", "done" %></td>
    </tr>
    <tr>
      <td><b>Notes: </b></td>
<% if @item.note_id.nil? %>
      <td>None</td>
      <td><%= link_to_image "note", :controller => "notes", :action => "new", :id =>
@item.id %></td>
<% else %>
      <td><%=h @item.note.more_notes %></td>
      <td><%= link_to_image "edit_button", :controller => "notes", :action => "edit",
:id => @item.note_id %></td>
      <td><%= link_to_image "delete_button", {:controller => "notes", :action =>
"destroy", :id => @item.note_id }, :confirm => "Are you sure you want to delete this
note?" %></td>
<% end %>
</tr>
  </table>
  <hr />
  <input type="submit" value="Update" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'">
</form>
```

But before moving to the Notes screen, remember we need to make sure the variable for the dropdown list is populated any time we invoke the Edit screen:

```
app\controllers\items_controller.rb (excerpt)

  def edit
     @categories = Category.find_all
     @item = Item.find(@params['id'])
  end

  def update
    @item = Item.find(@params['item']['id'])
    if @item.update_attributes(@params['item'])
      flash['notice'] = 'Item was successfully updated.'
    redirect_to :action => 'list'
    else
      flash['notice'] = 'Item NOT updated.'
      redirect_to :action => 'list'
    end
  end
```

Editing an existing Note is easy:

```
app\views\notes\edit.rhtml

<% @heading = "Edit Note" %>
<%= error_messages_for 'note' %>
<form action="/notes/update" method="post">
  <%= hidden_field "note", "id" %>
  <table>
    <tr>
      <td><b>Note:</b></td>
    </tr>
    <tr>
      <td><%= text_area "note", "more_notes", "cols" => 60, "rows" => 20 %></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Update" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for(
:controller => 'items', :action => 'list' ) %>'">
</form>
```

and once the `update` or `destroy` of the Notes table is complete, we want to return to the ToDo List screen:

```
app\controllers\notes_controller.rb (excerpt)

  def update
    @note = Note.find(@params['note']['id'])
    if @note.update_attributes(@params['note'])
      flash['notice'] = 'Note was successfully updated.'
      redirect_to :controller => 'items', :action => 'list'
    else
      render_action 'edit'
    end
  end

  def destroy
    Item.find_by_note_id(@params['id']).update_attribute('note_id',NIL)
    Note.find(@params['id']).destroy
    redirect_to :controller => 'items', :action => 'list'
  end
```

Saving and retrieving Data using Session Variables

However, the create is a bit more tricky. What we want to do is:

- store the new note in the Notes table
- find the id of the newly created record in the Notes table
- record this id back in the notes_id field of the associated record in the Items table

First of all, when we go off to create the new Notes record, we pass the id of the Item we are editing:

```
app\views\items\edit.rhtml (excerpt)

      <td><%= link_to_image "note", :controller => "notes", :action => "new", :id =>
@item.id %></td>
```

The `new` method in the Notes controller stores this away in a session variable:

```
app\controllers\notes_controller.rb (excerpt)

  def new
    @session['item_id'] = @params['id']
    @note = Note.new
  end
```

The New Notes template has no surprises:

```
<% @heading = "New Note" %>

<%= error_messages_for 'note' %>
<form action="/notes/create" method="post">
  <%= hidden_field "note", "id" %>
  <table>
    <tr>
      <td><b>Note:</b></td>
    </tr>
    <tr>
      <td><%= text_area "note", "more_notes", "cols" => 60, "rows" => 15 %></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Save" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for(
:controller => 'items', :action => 'list' ) %>'">
</form>
```

The `create` method retrieves the session variable again and uses it to find the record in the Items table. It then updates the note_id in the Item table with the id of the record it has just created in the Note table, and returns to the Items controller again:

app\controllers\notes_controller.rb (excerpt)

```
  def create
    @note = Note.new(@params['note'])
    if @note.save
      flash['notice'] = 'Note was successfully created.'
      @item = Item.find(@session['item_id'])
      @item.update_attribute('note_id', @note.id)
      redirect_to :controller => 'items', :action => 'list'
    else
      render_action 'new'
    end
  end
```

Documentation: ActionController::Base

Tidying up Navigation

There isn't a great deal left to do on the system now, other than tidy up the templates created in earlier days and adding in navigation buttons:

app\views\categories\list.rhtml

```
<% @heading = "Categories" %>
<form action="/categories/new" method="post">
<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>
<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to_image 'edit', { :action => 'edit', :id => category.id } %></td>
    <td><%= link_to_image 'delete', { :action => 'destroy', :id => category.id },
:confirm => 'Are you sure you want to delete this category?' %></td>
  </tr>
<% end %>
</table>
<hr />
  <input type="submit" value="New Category..." />
  <input type="button" value="To Dos" onClick="parent.location='<%= url_for(
:controller => 'items', :action => 'list' ) %>'">
</form>
```

app\views\categories\new.rhtml

```
<% @heading = "Add new Category" %>
<%= error_messages_for 'category' %>
<form action="/categories/create" method="post">
  <table>
    <tr>
      <td><b>Category:</b></td>
      <td><%= text_field "category", "category", "size" => 20, "maxlength" => 20
%></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Save" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'">
</form>
```

app\views\categories\edit.rhtml

```
<% @heading = "Rename Category" %>
<%= error_messages_for 'category' %>
<form action="/categories/update" method="post">
  <%= hidden_field "category", "id" %>
  <table>
    <tr>
      <td><b>Category:</b></td>
      <td><%= text_field "category", "category", "size" => 20, "maxsize" => 20 %></td>
    </tr>
  </table>
  <hr />
  <input type="submit" value="Update" />
  <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'">
</form>
```

The final navigation paths through the application are shown below. Any redundant scaffold code – e.g. the `show.rhtml` files – can be simply deleted. That's the beauty of scaffold code – it didn't cost you any effort to code it in the first place, and once it's served its purpose, just get rid of it.

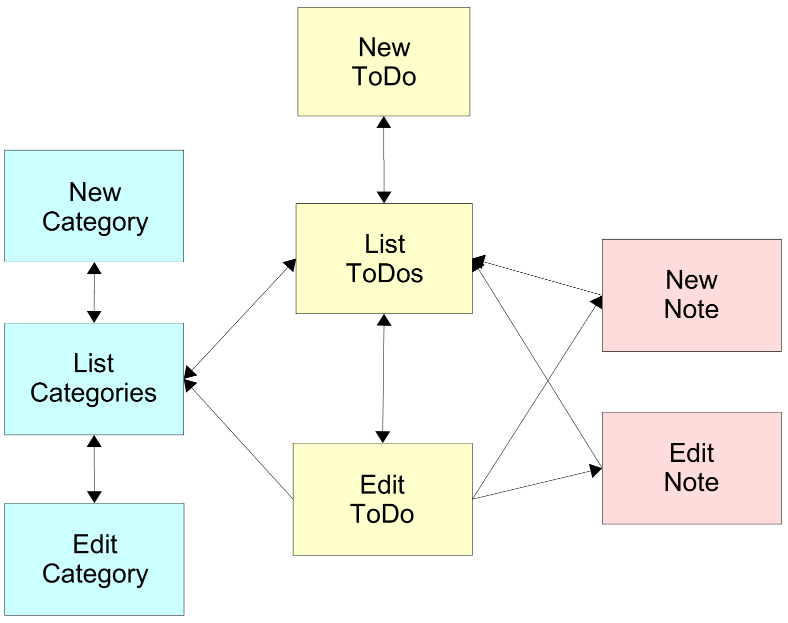


Illustration 8 Navigation paths through the Application

Setting the Home Page for the Application

As a final step, we need to kill the default 'Welcome to Rails' screen if the user points their browser to `http://todo`. There are three steps:

- change the Apache rewrite rule:

public\.htaccess (excerpt)

```
# Enable this rewrite rule to point to the controller/action that should serve root.
# RewriteRule ^$ /controller/action [R]
RewriteRule ^$ /items/list
```

- rename `public\index.html public\index.html.orig`
- point your browser to `http://todo`

Note: Rails 0.10 introduced *Routes* – a native Rails technique for working with custom URLs. If you are using 0.10 or above, you need to add the home page definition in the new Routes file:

config\routes.rb (excerpt)

```
  map.connect '', :controller => 'items', :action => 'list'
```

Links on the Home Page

There's a little Rails *gotcha* to watch out for here. When you access the `items/list` screen using this shortcut URL, Rails loses the plot a bit. If you use the simple form of `link_to` – e.g. `link_to_image "done", "purge_completed",...` – then you'll find the links don't work any more. However, if you explicitly specify the controller – e.g. `link_to_image "done", {:controller => 'items', :action => 'purge_completed'},...` – then everything works. You only need to worry about this on the your home page (both template and partials) – once you navigate away from the home page, then the system gets "back on the rails" again :-)

Downloading a Copy of this Application

If you'd like a copy of the ToDo application to play with, there's a link on http://rails.homelinux.org. You'll need to

- use Rails to set up the directory structure (see *Running the Rails script* on page 3)
- download the `todo_app.zip` file into the newly created `ToDo` directory
- unzip the files `unzip -o todo_app.zip`
- rename `public\index.html public\index.html.orig`
- if you want to use the sample database, `mysql -uroot -p < db/ToDo.sql`

and finally

I hope you found this document useful – I'm always happy to receive feedback, good or bad, to jpmcc@users.sourceforge.net.

Happy coding with Rails!

Index of Rails and Ruby Terms used in this Document