

Four Days on Rails



compiled by John McCreesh

Table of Contents

| | |
|---|-----------|
| Introduction..... | 1 |
| Day 1 on Rails..... | 3 |
| The ‘To Do List’ application..... | 3 |
| Running the Rails script..... | 3 |
| Adding the Application to the Web Server..... | 3 |
| Defining the Application in the hosts file..... | 3 |
| Defining the Application in the Apache Configuration file..... | 3 |
| Switching to fastcgi..... | 3 |
| Checking that Rails is working..... | 4 |
| Versions of Rails..... | 4 |
| Setting up the Database..... | 4 |
| Creating the Categories Table..... | 4 |
| MySQL definition..... | 4 |
| Data Model..... | 5 |
| Scaffold..... | 5 |
| Enhancing the Model..... | 6 |
| Creating Data Validation Rules..... | 6 |
| Day 2 on Rails..... | 9 |
| The Generated Scaffold Code..... | 9 |
| The Controller..... | 9 |
| The View..... | 10 |
| Layout..... | 11 |
| Template..... | 11 |
| Partial..... | 12 |
| The Rendered View for the “New” action..... | 13 |
| Analysing the View for the ‘List’ action..... | 13 |
| Tailoring the Generated Scaffold Code..... | 15 |
| The Controller..... | 15 |
| The View..... | 15 |
| Displaying Flash Messages..... | 15 |
| Sharing Variables between the Template and Layout..... | 16 |
| Tidying up the Edit and New Screens..... | 17 |
| Day 3 on Rails..... | 19 |
| The ‘Items’ Table..... | 19 |
| MySQL table defintion..... | 19 |
| The Model..... | 19 |
| Validating Links between Tables..... | 20 |
| Validating User Input..... | 20 |
| The ‘Notes’ table..... | 20 |
| MySQL table defintion..... | 20 |
| The Model..... | 20 |
| Using a Model to maintain Referential Integrity..... | 21 |
| More Scaffolding..... | 21 |
| More on Views..... | 21 |
| Creating a Layout for the Application..... | 21 |
| The ‘To Do List’ screen..... | 22 |
| Purging completed ‘To Dos’ by clicking on an icon..... | 23 |
| Changing the Sort Order by clicking on the Column Headings..... | 24 |
| Adding a Helper..... | 24 |
| Using Javascript Navigation Buttons..... | 25 |
| Formatting a Table with a Partial..... | 25 |
| Formatting based on Data Values..... | 26 |
| Handling Missing Values in a Lookup..... | 26 |
| The ‘New To Do’ Screen..... | 26 |

| | |
|---|-----------|
| Creating a Drop-down List for a Date Field..... | 27 |
| Trapping Exceptions in Ruby..... | 27 |
| Creating a Drop-down List from a Lookup Table..... | 28 |
| Creating a Drop-down List from a List of Constants..... | 28 |
| Creating a Checkbox..... | 28 |
| Finishing Touches..... | 28 |
| Tailoring the Stylesheet..... | 28 |
| The ‘Edit To Do’ Screen..... | 29 |
| Day 4 on Rails..... | 31 |
| The ‘Notes’ screens..... | 31 |
| Linking ‘Notes’ to the ‘Edit To Do’..... | 31 |
| The ‘Edit Notes’ Screen..... | 32 |
| The ‘New Note’ Screen..... | 32 |
| Saving and retrieving Data using Session Variables..... | 33 |
| Changing the ‘Categories’ Screens..... | 33 |
| Navigation through the system..... | 34 |
| Setting the Home Page for the Application..... | 35 |
| Downloading a Copy of this Application..... | 35 |
| and finally..... | 35 |
| Appendix – afterthoughts..... | 37 |
| Multiple Updates..... | 37 |
| View..... | 37 |
| Controller..... | 38 |
| User Interface considerations..... | 39 |
| Still to be done..... | 39 |

Introduction

There have been many extravagant claims made about Rails. For example, an article in OnLAMP.com¹ 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 went 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> (this documentation is also installed on your PC as part of your gems installation in a location like C:\Program Files\ruby\lib\ruby\gems\n.n\doc\actionpack-n.n.n\rdoc\index.html)
- *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 channel² and the mailing list³. The on-line archives record their invaluable assistance as I clawed my way up the Rails and Ruby learning curves.

Version: 2.3 using version 0.12.1 of Rails – see <http://rails.homelinux.org> for latest version and to download a copy of the ToDo code. Document written and pdf file generated with [OpenOffice.org](http://www.openoffice.org) 'Writer'.

Copyright: this work is copyright ©2005 John McCreesh jpmcc@users.sourceforge.net and is licensed under the *Creative Commons Attribution-NonCommercial-ShareAlike License*. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

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 'To Do List' application

This document follows the building of a simple 'To Do 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 5: The 'To Do List' Screen* on page 23).

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 a new directory `ToDo\` and populates it with a series of files and subdirectories, the most important of which are as follows:

```
app
  contains the core of the application, split between model, view, controller, and
  'helper' subdirectories
config
  contains the database.yml file which provides details of the database to used with
  the application
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 images, javascripts, and
  stylesheets subdirectories
```

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

```
# For better performance replace the dispatcher with the fastcgi one
RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
```

Checking that Rails is working

The site should now be visible in your browser as `http://todo/` (you should see the Congratulations, you've put Ruby on Rails! page in your browser).

Versions of Rails

By the time you read this document, Rails will probably have moved on several versions. If you intend to work through this document, check the versions installed on your PC:

```
W:\ToDo>gem list --local
```

If they are different from the versions listed below, then I would strongly advise you to download the versions used in 'Four Days', e.g.:

```
W:\ToDo>gem install rails --version 0.12.1
```

This won't break anything; Ruby's gems library is designed to handle multiple versions. You can then force Rails to use the 'Four Days' versions with the 'To Do List' application by specifying:

config\environment.rb (excerpt)

```
# Require Rails libraries.
require 'rubygems'
require_gem 'activesupport', '= 1.0.4'
require_gem 'activerecord', '= 1.10.1'
require_gem 'actionpack', '= 1.8.1'
require_gem 'actionmailer', '= 0.9.1'
require_gem 'actionwebservice', '= 0.7.1'
require_gem 'rails', '= 0.12.1'
```

The reason using the same versions is quite simple. 'Four Days' uses a lot of code generated automatically by Rails. As Rails develops, so does this code – unfortunately, this document doesn't (until I get round to producing a new version!). So, make life easy for yourself, and keep to the same versions as used in 'Four Days'. Once you've finished working through 'Four Days', by all means go onto the latest and greatest Rails versions and see what improvements the Rails developers have come up with.

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

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 'To Do' 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
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/category.rb
create test/unit/category_test.rb
create test/fixtures/categories.yml

W:\ToDo>

```

which creates an empty `category.rb`, and two test files `category_controller_test.rb` and `categories.yml`. We’ll make some entries in the data model in a minute – leave it empty just now.

Scaffold

The controller is at the heart of a Rails application.

Running the generate controller script

```

W:\ToDo>ruby script/generate controller category
exists app/controllers/
exists app/helpers/
create app/views/category
exists test/functional/
create app/controllers/category_controller.rb
create test/functional/category_controller_test.rb
create app/helpers/category_helper.rb

W:\ToDo>

```

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 one line of code:

app\controllers\category_controller.rb

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

Documentation: ActionController::Scaffolding

Point your browser at <http://todo/category> and marvel at how clever it is :-)

Listing categories

| Category | Created on | Updated on | |
|------------------------------|---|---|---|
| Home & Family | Mon Jun 06 15:56:44 GMT Daylight Time 2005 | Wed Jun 15 17:09:59 GMT Daylight Time 2005 | Show Edit Destroy |
| Business | Mon Jun 06 15:57:00 GMT Daylight Time 2005 | Wed Jun 15 17:10:15 GMT Daylight Time 2005 | Show Edit Destroy |
| Rails documentation | Tue Jun 14 09:34:02 GMT Daylight Time 2005 | Tue Jun 14 09:34:02 GMT Daylight Time 2005 | Show Edit Destroy |
| Community Council | Tue Jun 14 09:34:34 GMT Daylight Time 2005 | Tue Jun 14 09:34:34 GMT Daylight Time 2005 | Show Edit Destroy |
| New category | | | |

Illustration 1: Scaffold 'List' screen

To find out how clever it is not, try adding the same new category twice. Rails will collapse with a messy error message 'ActiveRecord::StatementInvalid in Category#create'. You can fix this by adding validation into the Model.

Enhancing the Model

The Model is where all the data-related rules are stored, including data validation and relational integrity. This means you can define a rule 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

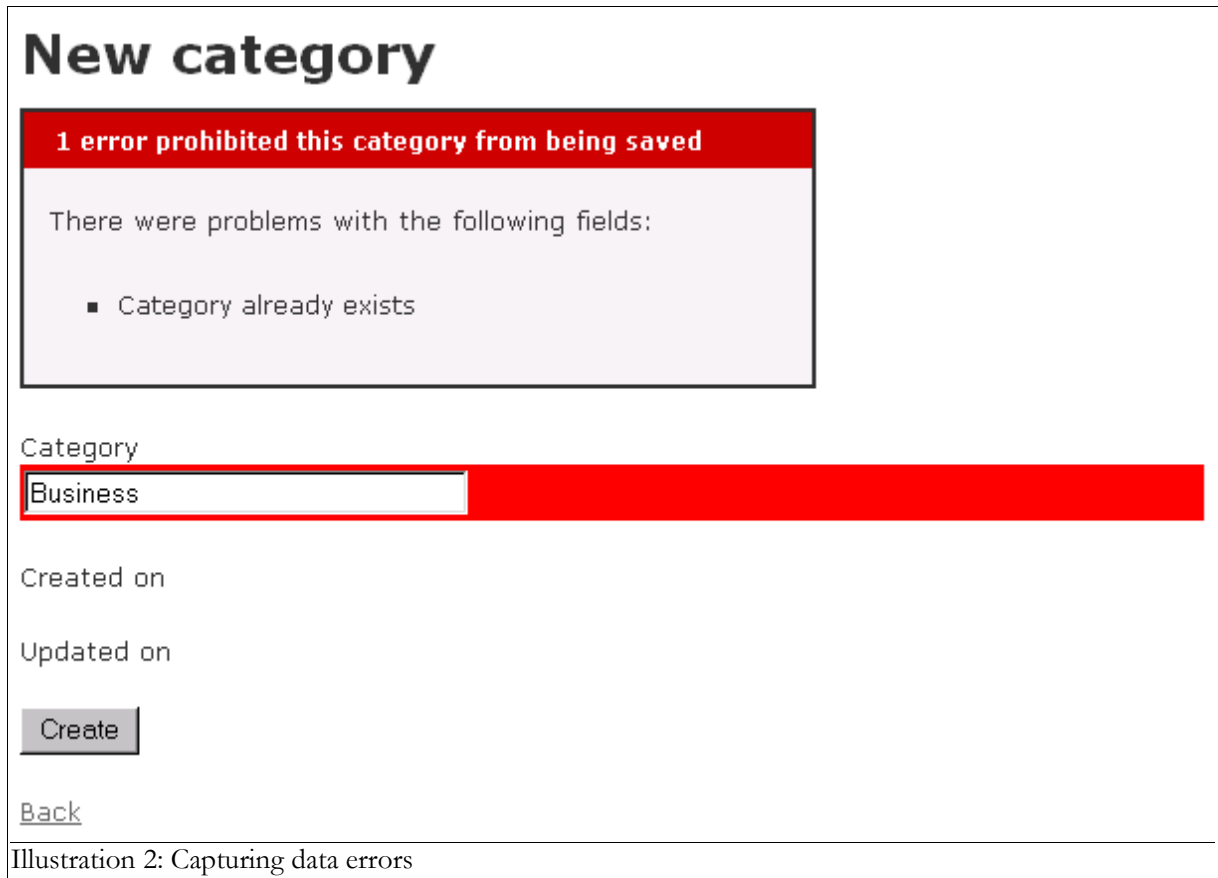
```
class Category < ActiveRecord::Base
  validates_length_of :category, :within => 1..20
  validates_uniqueness_of :category, :message => "already exists"
end
```

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. I don't like the default Rails error message - 'xxx has already been taken' - so I provide my own. This is a general feature of Rails - try the defaults first; if you don't like anything, overwrite it.

Documentation: ActiveRecord::Validations::ClassMethods

To try this out, now try to insert a duplicate record again. This time, Rails handles the error rather than crashing - see 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?



New category

1 error prohibited this category from being saved

There were problems with the following fields:

- Category already exists

Category

Created on

Updated on

[Back](#)

Illustration 2: Capturing data errors

Day 2 on Rails

To progress beyond this point, we need to see what's happening behind the scenes. During day 2, we will work systematically through the scaffold code generated by Rails, deciphering what it all means. 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
dependency  model
  exists    app/models/
  exists    test/unit/
  exists    test/fixtures/
  skip      app/models/category.rb
  skip      test/unit/category_test.rb
  skip      test/fixtures/categories.yml
exists      app/controllers/
exists      app/helpers/
create      app/views/categories
exists      test/functional/
create      app/controllers/categories_controller.rb
create      test/functional/categories_controller_test.rb
create      app/helpers/categories_helper.rb
create      app/views/layouts/categories.rhtml
create      public/stylesheets/scaffold.css
create      app/views/categories/list.rhtml
create      app/views/categories/show.rhtml
create      app/views/categories/new.rhtml
create      app/views/categories/edit.rhtml
create      app/views/categories/_form.rhtml

W:\ToDo>
```

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

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>. In fact, to avoid confusion, it's best to delete `app\controllers\category_controller.rb` etc in case you run it accidentally.

The Generated Scaffold Code

The Controller

Let's look at the code behind 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 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
    @category_pages, @categories = paginate :category, :per_page => 10
  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[:id])
    if @category.update_attributes(@params[:category])
      flash['notice'] = 'Category was successfully updated.'
      redirect_to :action => 'show', :id => @category
    else
      render_action 'edit'
    end
  end

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

```

When the user of a Rails application selects an action – e.g. ‘Show’ - the controller will execute any code in the appropriate section – ‘def show’ - and then by default will render a template of the same name - ‘show.rhtml’. This default behaviour can be overwritten:

- `render_template` allows you to render a different template – e.g. the `index` action will run the code for ‘list’ - ‘def 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. Note that you do not have to write any SQL statements, but if you want to see what SQL Rails is using, it’s all written to the `development.log` file.

Documentation: ActiveRecord::Base

Notice how one logical activity from the user’s perspective may require two passes through the controller: for example, updating a record in the table. 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.

The View

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

| Layout | Template | Partial |
|--|---|--|
| in app\views\layouts\ default: application.rhtml or <controller>.rhtml | in app\views\<controller>\ default: <action>.rhtml | in app\views\<controller>\ default _<partial>.rhtml |

- A Layout provides common code used by all actions, typically the start and end of the HTML sent to the browser.
- A Template provides code specific to an action, e.g. 'List' code, 'Edit' code, etc.
- A Partial provides common code - 'subroutines' - which can be used in used in multiple actions – e.g. code used to lay out tables for a form.

Layout

Rails Naming conventions: 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>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

  <%= @content_for_layout %>

</body>
</html>
```

This is mostly HTML, plus a few bits of Ruby code embedded within <% %> tags. This layout will be called by the rendering process regardless of the action being run. It contains the standard HTML tags – the <html><head>...</head><body>...</body></html> that will appear on every page.

The Ruby bits in bold are translated into HTML during the Rails rendering process as follows:

- `action_name` is an ActionController method which returns the name of the action the controller is processing (e.g. 'List') - this puts an appropriate title on the page, depending on the action being run.

Documentation: ActionController::Base

- `stylesheet_link_tag` is a Rails helper - a lazy way of generating code. There are a lot of these 'helpers' within Rails. This one simply generates the following HTML: <link href="/stylesheets/scaffold.css" media="screen" rel="Stylesheet" type="text/css" />

Documentation: ActionView::Helpers::AssetTagHelper

- `content_for_layout` is the key to what happens next. It 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 with the same name – see below.

Documentation: ActionController::Layout::ClassMethods.

Template

Rails naming convention: templates are held in app\views\categories\'action'.rhtml.

The new.rhtml created by the scaffold script is given below:

```
app\views\categories\new.rhtml
<h1>New category</h1>

<%= start_form_tag :action => 'create' %>
  <%= render_partial "form" %>
  <%= submit_tag "Create" %>
<%= end_form_tag %>

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

- start_form_tag is a Rails helper to start an HTML form – here it generates <form action="/categories/create" method="post">
- submit_tag by itself would generate <input name="submit" type="submit" value="Save changes" />, but the “Create” parameter overwrites the default “Save changes” with “Create”
- end_form_tag just outputs </form>, which is not the most useful Rails helper ever written :-) but it provides a satisfying end to the block of code

Documentation: ActionController::Helpers::FormTagHelper

- render_partial will invoke a Partial_form.rhtml - see the next section.

Documentation: ActionController::Partials

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

Documentation: ActionController::Helpers::UrlHelper

Partial

Rails naming convention: a partial ‘foo’ will go in a file app\views\‘action’_foo.rhtml (note the initial underscore).

The scaffold uses the same code to process both the ‘edit’ and ‘new’ actions, so it puts the code into a partial, invoked by the render_partial method.

```
app\views\categories\_form.rhtml
<%= error_messages_for 'category' %>

<!--[form:category]-->
<p><label for="category">Category</label><br/>
<%= text_field 'category', 'category' %></p>

<p><label for="category_created_on">Created on</label><br/>
</p>

<p><label for="category_updated_on">Updated on</label><br/>
</p>
<!--[eoform:category]-->
```

- 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 xxx 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>
```

We saw this in action on Day 1 - *Illustration 2: Capturing data errors* on page 7. Note: the css tags match

corresponding statements in the stylesheet created by the generate scaffold script.

Documentation: ActionView::Helpers::ActiveRecordHelper

- `text_field` is a Rails Helper which generate this HTML: `<input id="category_category" name="category[category]" size="30" type="text" value="" />`. The first parameter is the table name; the second is the field name.

Documentation: ActionView::Helpers::FormHelper

Note a little bug in Rails – it knows not to create input fields for the reserved field names `created_on` and `updated_on`, but it still generates labels for them.

The Rendered View for the “New” action

We’re now in a position to look at the code that’s returned to the browser in response to the “New” action, and see where it’s all come from. The Layout supplies the **bold** text; the Template the `Regular` text; and the Partial the *Italic* text:

app\views\categories\new.rhtml

```
<html>
<head>
  <title>Categories: new</title>
  <link href="/stylesheets/scaffold.css" media="screen" rel="Stylesheet"
type="text/css" />
</head>
<body>

<h1>New category</h1>

<form action="/categories/create" method="post">

<!--[form:category]-->
<p><label for="category_category">Category</label><br/>
<input id="category_category" name="category[category]" size="30" type="text" value=""
/></p>

<p><label for="category_created_on">Created on</label><br/>
</p>

<p><label for="category_updated_on">Updated on</label><br/>
</p>
<!--[eoform:category]-->

  <input name="submit" type="submit" value="Create" />
</form>

<a href="/categories/list">Back</a>

</body>
</html>
```

Analysing the View for the ‘List’ action

The ‘Edit’ and ‘Show’ views are similar to the ‘New’ view. ‘List’ contains a few new tricks. Remember how the controller ran the following piece of code before going off to render the ‘List’ template:

```
@category_pages, @categories = paginate :category, :per_page => 10
```

`paginate` populates the `@categories` instance variable with sorted records from the `Categories` table, `:per_page` records at a time, and contains all the logic for next page / previous page etc. navigation. `@category_pages` is a `Paginator` instance. How these are used in the template is explained at the end of the following section.

Documentation: ActionController::Pagination

The template is as follows:

```
app\views\categories\list.rhtml
<h1>Listing categories</h1>

<table>
  <tr>
    <% for column in Category.content_columns %>
      <th><%= column.human_name %></th>
    <% end %>
  </tr>

  <% for category in @categories %>
    <tr>
      <% for column in Category.content_columns %>
        <td><%=h category.send(column.name) %></td>
      <% end %>
      <td><%= link_to 'Show', :action => 'show', :id => category %></td>
      <td><%= link_to 'Edit', :action => 'edit', :id => category %></td>
      <td><%= link_to 'Destroy', {:action => 'destroy', :id => category}, :confirm =>
"Are you sure?" %></td>
    </tr>
  <% end %>
</table>

<%= link_to "Previous page", { :page => @category_pages.current.previous } if
@category_pages.current.previous %>
<%= link_to "Next page", { :page => @category_pages.current.next } if
@category_pages.current.next %>

<br />

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

- `content_columns` returns an array of column objects excluding any ‘special’ columns (the primary id, all columns ending in ‘_id’ or ‘_count’, and columns used for single table inheritance)

Documentation: ActionController::Base

- `human_name` is a synonym for `human_attribute_name`, which transforms attribute key names into a more human format, such as ‘First name’ instead of ‘first_name’

Documentation: ActiveRecord::Base

- `h` automatically ‘escapes’ HTML code. 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 displayed⁴. 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 `</table>`; which is harmless. Rails makes this really simple – just add an ‘h’ as shown
- `confirm` is a useful optional parameter for the `link_to` helper – it generates a Javascript pop-up box which forces the user to confirm the `Destroy` before actioning the link:



Illustration 3: Javascript pop-up

⁴ For example, think what would happen if a user typed in “`</table>`” as a `Category`.

Documentation: *ActionView::Helpers::UrlHelper*

The paging logic takes a bit of unravelling.. Ruby can use `if` as a modifier: `expression if boolean-expression` evaluates `expression` only if `boolean-expression` is true. `@category_pages.current` returns a Page object representing the paginator's current page

ActionController::Pagination::Paginator

and `@category_pages.current.previous` returns a new Page object representing the page just before this page, or nil if this is the first page

ActionController::Pagination::Paginator::Page

So, if there is a previous page to navigate to, then this construct will display a link; if there isn't, the link is suppressed.

The rendered code for page *n* will look like:

```
<a href="/categories/list?page=[n-1]">Previous page</a>
<a href="/categories/list?page=[n+1]">Next page</a>
```

Tailoring the Generated Scaffold Code

The code generated by the Scaffold script is perfectly usable 'out of the box', and is robust once you have added enough validation into your data model. However, if that's all there was to developing Rails applications, then programmers would be out of a job, which would clearly not be a good thing :-). So let's do some tailoring:

The Controller

In a 'List' view, I would expect the records to be displayed in alphabetical order. This requires a minor change to the controller:

```
app\controllers\categories_controller.rb (excerpt)
def list
  @category_pages, @categories = paginate :category,
    :per_page => 10, :order_by => 'category'
end
```

Documentation: *ActionController::Pagination*

In this application, the show screen is unnecessary – all the fields fit comfortably on a single row on the screen. So, `def show` can disappear, and let's go straight back to the `list` screen after an 'Edit':

```
app\controllers\categories_controller.rb (excerpt)
def update
  @category = Category.find(@params[: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. By default, the scaffold script doesn't display flash messages - we'll change this in a minute – see below.

The View

Displaying Flash Messages

Rails provides a technique for passing 'flash' messages back to the user – e.g. an 'Update Successful' message which displays on the next screen and then disappears. These can be picked up easily with a small change to the Layout (adding it to the Layout means it will appear on any screen):

```

app\views\layouts\categories.rhtml<html>
<head>
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<h1><%= @heading %></h1>
<% if @flash["notice"] %>
<span class="notice">
  <%=h @flash["notice"] %>
</span>
<% end %>
<%= @content_for_layout %>
</body>
</html>

```

Documentation: ActionController::Flash

A simple addition to the stylesheet makes the flash message more conspicuous:

```

public\stylesheets\scaffold.css (excerpt)
.notice {
  color: red;
}

```

Sharing Variables between the Template and Layout

Note that I've moved the `<h1>...</h1>` heading text out of the Template into the Layout so that it appears above the flash message. As each template will have a different heading, I need to set the value of the variable `@heading` in the Template. Rails is quite ok with this – Template variables are available to Layouts at rendering time.

I've made this change and some formatting changes to come up with my finished template:

```

app\views\categories\list.rhtml
<% @heading = "Categories" %>
<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 'Edit', :action => 'edit', :id => category %></td>
      <td><%= link_to 'Delete', {:action => 'destroy', :id => category},
        :confirm => "Are you sure you want to delete this category?" %></td>
    </tr>
  <% end %>
</table>
<br />
<%= link_to 'New category', :action => 'new' %>
<% if @category_pages.page_count>1 %>
<hr />
Page: <%=pagination_links @category_pages %>
<hr />
<% end %>

```

- I don't like the default date format, so I use a Ruby method `strftime()` to format the date and time fields the way I want them.

Ruby Documentation: class Time

- `pagination_links` creates a basic HTML link bar for the given paginator

ActionView::Helpers::PaginationHelper

Tidying up the Edit and New Screens

A few changes to the Partial used by 'New' and 'Edit': use a table to improve the layout; get rid of the unwanted `created_on/updated_on` labels; and prevent the user typing too much into the `Category` field:

app\views\categories_form.rhtml

```
<%= error_messages_for 'category' %>
<table>
<tr>
  <td><b><label for="category_category">Category:</label></b></td>
  <td><%= text_field "category", "category", "size"=>20, "maxlength"=>20 %></td>
</tr>
</table>
```

and a few minor changes to the two templates (note in particular the use of `@heading`):

app\views\categories\Edit.rhtml

```
<% @heading = "Edit Category" %>
<%= start_form_tag :action => 'update', :id => @category %>
  <%= render_partial "form" %>
  <hr />
  <%= submit_tag "Save" %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

app\views\categories\New.rhtml

```
<% @heading = "New Category" %>
<%= start_form_tag :action => 'create' %>
  <%= render_partial "form" %>
  <hr />
  <%= submit_tag "Save" %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

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 'To Dos'. 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.

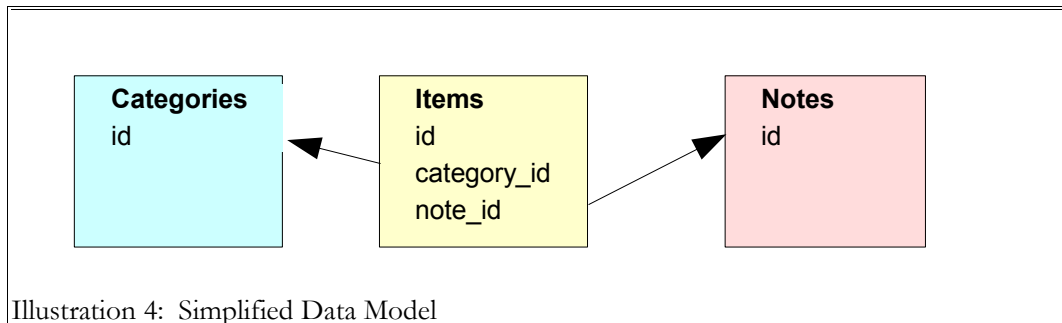


Illustration 4: Simplified Data Model

The 'Items' Table

MySQL table definition

The fields in the Items table are as follows:

- done - 1 means the To Do item has been completed⁵
- 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 To Do item 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

As before, Rails can generate an empty model file:

```
W:\ToDo>ruby script/generate model item  
exists app/models/  
exists test/unit/  
exists test/fixtures/  
create app/models/item.rb  
create test/unit/item_test.rb  
create test/fixtures/items.yml  
  
W:\ToDo>
```

⁵ MySQL doesn't have a 'boolean' type, so we have to use 0/1

which we can populate:

app\models\item.rb

```
class Item < ActiveRecord::Base
  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' input⁶.
- `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 stored⁷.

Documentation: ActiveRecord::Validations::ClassMethods

The 'Notes' table

This table contains a single free text field to hold further information for a particular To Do 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 :-)

MySQL table definition

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

Generate the empty model file, but it contains nothing new:

app\models\note.rb

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

⁶ 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

⁷ You could combine the two rules for the Description field into one: `validates_length_of :description, :within => 1..40`

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, you’ll see later that we can delete Item records by clicking on a Dustbin icon on the ‘To Do’ 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 :-)

Similarly, if a record is deleted from the Notes table, then any reference to it in the Items table needs to be erased:

app\models\note.rb (excerpt)

```
def before_destroy
  Item.find_by_note_id(id).update_attribute('note_id',NIL)
end
```

Documentation: ActiveRecord::Callbacks

More Scaffolding

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
[snip]
W:\ToDo>ruby script/generate scaffold Note
[snip]
W:\ToDo>
```

Note: as we tailored the stylesheet yesterday, reply “n” to the “overwrite public/stylesheets/scaffold.css? [Ynaq]” prompt.

More on Views

Creating a Layout for the Application

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 an application-wide layout. Delete all the app\views\layouts*.rhtml files,

and replace with a common application.rhtml.

app\views\layouts\application.rhtml

```
<html>
<head>
  <title><%= @heading %></title>
  <%= stylesheet_link_tag 'todo' %>
<script language="JavaScript">
<!-- Begin
function setFocus() {
  if (document.forms.length > 0) {
    var field = document.forms[0];
    for (i = 0; i < field.length; i++) {
      if ((field.elements[i].type == "text") || (field.elements[i].type == "textarea")
|| (field.elements[i].type.toString().charAt(0) == "s")) {
        document.forms[0].elements[i].focus();
        break;
      }
    }
  }
}
// End -->
</script>
</head>
<body OnLoad="setFocus()" >
<h1><%=@heading %></h1>
<% if @flash["notice"] %>
<span class="notice">
  <%=h @flash["notice"] %>
</span>
<% end %>
<%= @content_for_layout %>
</body>
</html>
```

The @heading set in the Template is now used for the <title> as well as <h1>. I've renamed the public/stylesheets/scaffold.css to todo.css for tidiness, and also generally played with colours, table borders, to give a prettier layout. I've also added in a little Javascript to automatically position the cursor in the first input field in the browser ready for the user to start typing.

The 'To Do List' screen

What I'm trying to achieve is a look based on a PalmPilot or similar PDA desktop. The end product is shown in Illustration 5: The 'To Do List' Screen⁸.

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 and shown in bold
- 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 To Do...' button at the bottom of the screen
- there's a button link to the 'Categories' stuff from day 2

⁸ 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...

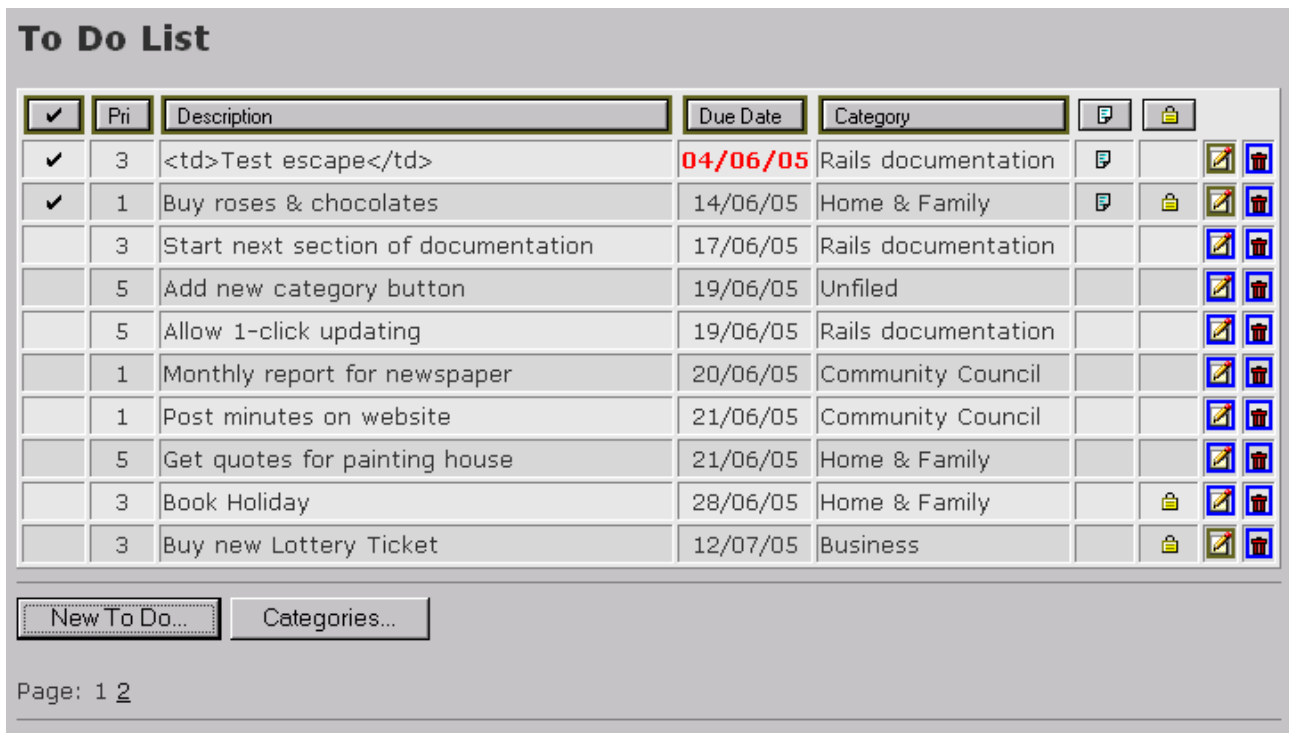


Illustration 5: The 'To Do List' Screen

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

```

app\views\items\list.rhtml
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'new' %>
<table>
  <tr>
    <th><%= link_to_image "done", {:action => "purge_completed"}, :confirm => "Are you
sure you want to permanently delete all completed To Dos?" %></th>
    <th><%= link_to_image "priority",{:action => "list_by_priority"}, "alt" => "Sort
by Priority" %></th>
    <th><%= link_to_image "description",{:action => "list_by_description"}, "alt" =>
"Sort by Description" %></th>
    <th><%= link_to_image "due_date", {:action => "list"}, "alt" => "Sort by Due Date"
%></th>
    <th><%= link_to_image "category", {: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 />
<%= submit_tag "New To Do..." %>
<%= submit_tag "Categories...", {:type => 'button', :onClick=>"parent.location='" +
url_for( :controller => 'categories', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action => @params["action"]
|| "index" }) + "<hr />" if @item_pages.page_count>1 %>

```

Purging completed 'To Dos' 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.

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
  @item_pages, @items = paginate :item,
    :per_page => 10, :order_by => 'due_date,priority'
end

def list_by_priority
  @item_pages, @items = paginate :item,
    :per_page => 10, :order_by => '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` method⁹. 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\application_helper.rb

```
module ApplicationHelper
  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\application.rb

```
class ApplicationController < ActionController::Base
  helper :Application
end
```

⁹ `list_by_description` and `list_by_category` are similar and are left as an easy exercise for the reader. However, if you get stuck with `list_by_category`, see *Still to be done* on page 39

it is available for all the templates in the application.

Documentation: ActionView::Helpers

Using Javascript Navigation Buttons

onClick is a standard Javascript 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 controller and an action, url_for will return the URL.

Documentation: ActionController::Base

Formatting a Table with a Partial

I wanted to create a nice stripey effect for the list of items. *Partials* provide the solution; 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
```

Documentation: ActionView::Partials

Rails also passes a sequential number list_stripes_counter to the Partial. 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.

The completed Partial is as follows:

app\views\items_list_stripes.rhtml

```
<tr class="<%= list_stripes_counter.modulo(2).nonzero? ? "dk_gray" : "lt_gray" %>">
  <td style="text-align: center"><%= list_stripes["done"] == 1 ?
show_image("done_ico.gif") : "&nbsp;" %></td>
  <td style="text-align: center"><%= 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" style="text-align: center">' : '<td style="text-align: center">' %><%=
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 `.past_due` 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 To Do items. In this case, the Category should be displayed as ‘Unfiled’:

```
list_stripes.category ? list_stripes.category["category"] : 'Unfiled'
```

OK. if you’ve followed this so far, you should have a ‘To Do List’ screen looking something like Illustration 5 *The ‘To Do List’ Screen* on page 23.

The ‘New To Do’ Screen

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

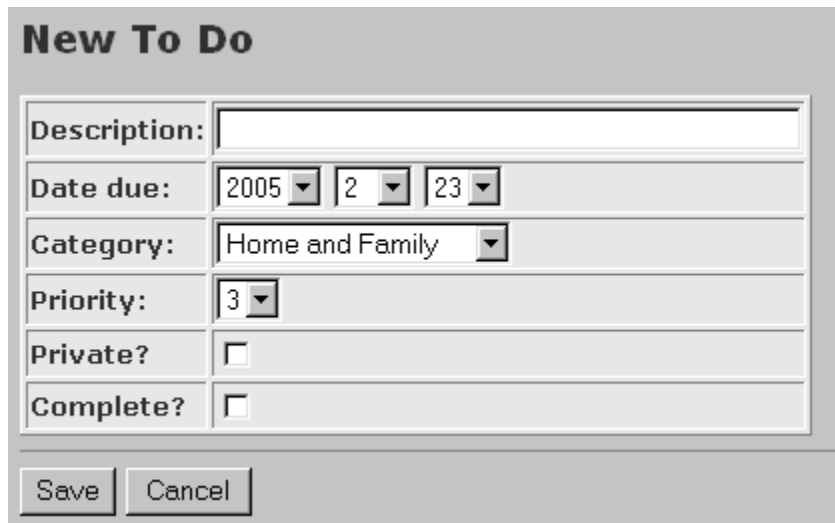


Illustration 6 New ‘To Do’ screen

The template is minimal:

```
app\views\items\new.rhtml
<% @heading = "New To Do" %>
<%= error_messages_for 'item' %>
<%= start_form_tag :action => 'create' %>
```

```

<table>
<%= render_partial "form" %>
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" + url_for(
:action =>'list' ) + "'" } %>
<%= end_form_tag %>

```

and the real work is done in the partial, where it can be shared with the 'Edit' action:

```

app\views\items\_form.rhtml
<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>
<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>

```

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

Trapping Exceptions in Ruby

Unfortunately, date_select quite happily accepts dates like 31st February. Rails then dies when it tries to save this 'date' to the database. One workaround 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
  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

Note that data driven drop down boxes have to get their data from somewhere – which means an addition to the controller:

app\controllers\items_controller.rb (excerpt)

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

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

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

Finishing Touches

Tailoring the Stylesheet

At this point, the 'To Do List' screen should work, and so should the 'New To Do' 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; }

.notice {
  color: red;
  background-color: white;
}
```



```
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 To Do’ Screen

The rest of Day 3 is taken up building the ‘Edit To Do’ screen, which is very similar to the ‘New To Do’. 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 you¹⁰.

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.

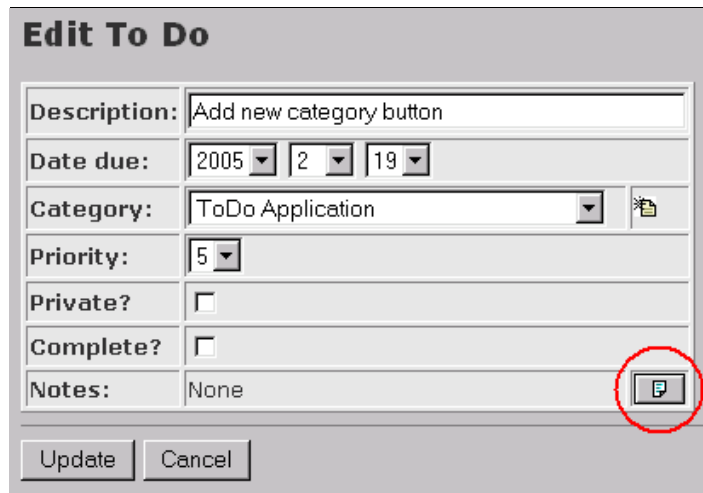
¹⁰ But unlike my college text book authors, I do reveal the answers on Day 4 :-) - see `app\views\items\edit.rhtml` on page 31

Day 4 on Rails

The 'Notes' screens

Linking 'Notes' to the 'Edit To Do'

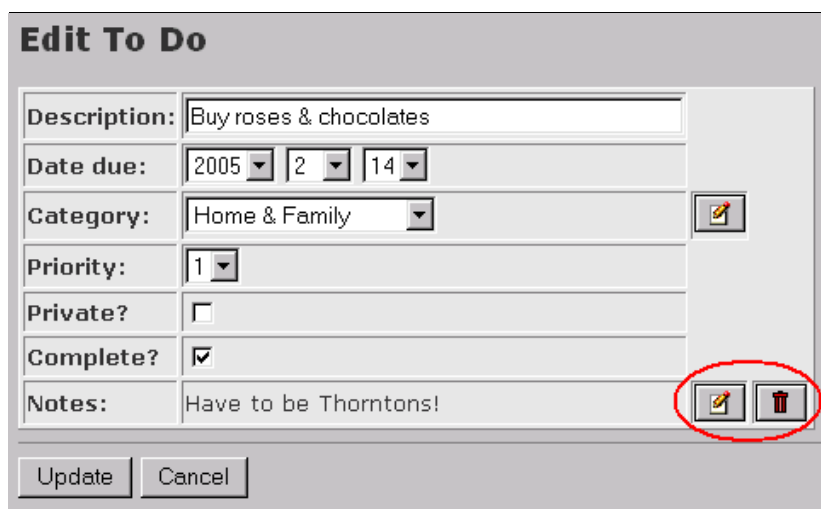
Although the Notes scaffold code gives the full CRUD facilities, we don't want the user to invoke any of this directly. Instead, if an Item has no associated Note, we want to be able to create one by clicking on a Notes icon on the Edit To Do screen:



The screenshot shows the 'Edit To Do' form with the following fields: Description: 'Add new category button', Date due: '2005 2 19', Category: 'ToDo Application', Priority: '5', Private?: unchecked, Complete?: unchecked, and Notes: 'None'. A red circle highlights a small icon with a plus sign and a document symbol in the bottom right corner of the Notes field.

Illustration 7: Creating a New Note from the 'Edit To Do' screen

If a Note already exists, we want to edit or delete it by clicking on the appropriate icon on the Edit To Do screen:



The screenshot shows the 'Edit To Do' form with the following fields: Description: 'Buy roses & chocolates', Date due: '2005 2 14', Category: 'Home & Family', Priority: '1', Private?: unchecked, Complete?: checked, and Notes: 'Have to be Thorntons!'. A red circle highlights two icons in the bottom right corner of the Notes field: a pencil icon for editing and a trash can icon for deleting.

Illustration 8: Editing or Deleting an existing Note

First of all, let's look at the code for the 'Edit To Do' 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' %>
<%= start_form_tag :action => 'update', :id => @item %>
  <table>
<%= render_partial "form" %>
```

```

      <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 />
  <%= submit_tag "Save" %>
  <%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" + url_for(
:action => 'list' ) + "'" } %>
  <%= end_form_tag %>

```

The 'Edit Notes' Screen

Editing an existing Note is pretty straightforward. This is the Template:

```

app\views\notes\edit.rhtml
<% @heading = "Edit Note" %>
<%= start_form_tag :action => 'update', :id => @note %>
  <%= render_partial "form" %>
  <%= submit_tag "Save" %>
  <%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" +
url_for( :controller => 'items', :action => 'list' ) + "'" } %>
<%= end_form_tag %>

```

and its matching Partial:

```

app\views\notes\_form.rhtml
<table>
  <tr>
    <td><label for="note_more_notes">More notes</label></td>
    <td><%= text_area 'note', 'more_notes' %></td>
  </tr>
</table>

```

Once the update or destroy of the Notes table is complete, we want to return to the 'To Do List' screen:

```

app\controllers\notes_controller.rb (excerpt)
def update
  @note = Note.find(@params[: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
  Note.find(@params[:id]).destroy
  redirect_to :controller => 'items', :action => 'list'
end

```

Remember that the referential integrity rules we have already created will ensure that when a Note is deleted, any references to it in Items will be removed too (see *Using a Model to maintain Referential Integrity* on page 21).

The 'New Note' Screen

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

Session variables provide a useful way of persisting data between screens – we can use them here to store the Id of the record in the Notes table.

Documentation: ActionController::Base

Saving and retrieving Data using Session Variables

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:

app\views\notes\new.rhtml

```
<% @heading = "New Note" %>
<%= start_form_tag :action => 'create' %>
<%= render_partial "form" %>
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='" + url_for(
:controller => 'items', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

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
```

Changing the ‘Categories’ Screens

There isn’t a great deal left to do on the system now, other than tidy up the templates created in earlier days so they have the same style of 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' %>
<%= start_form_tag :action => 'create' %>
    <%= render_partial "form" %>
    <hr />
    <input type="submit" value="Save" />
    <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'>
<%= end_form_tag %>

```

app\views\categories\edit.rhtml

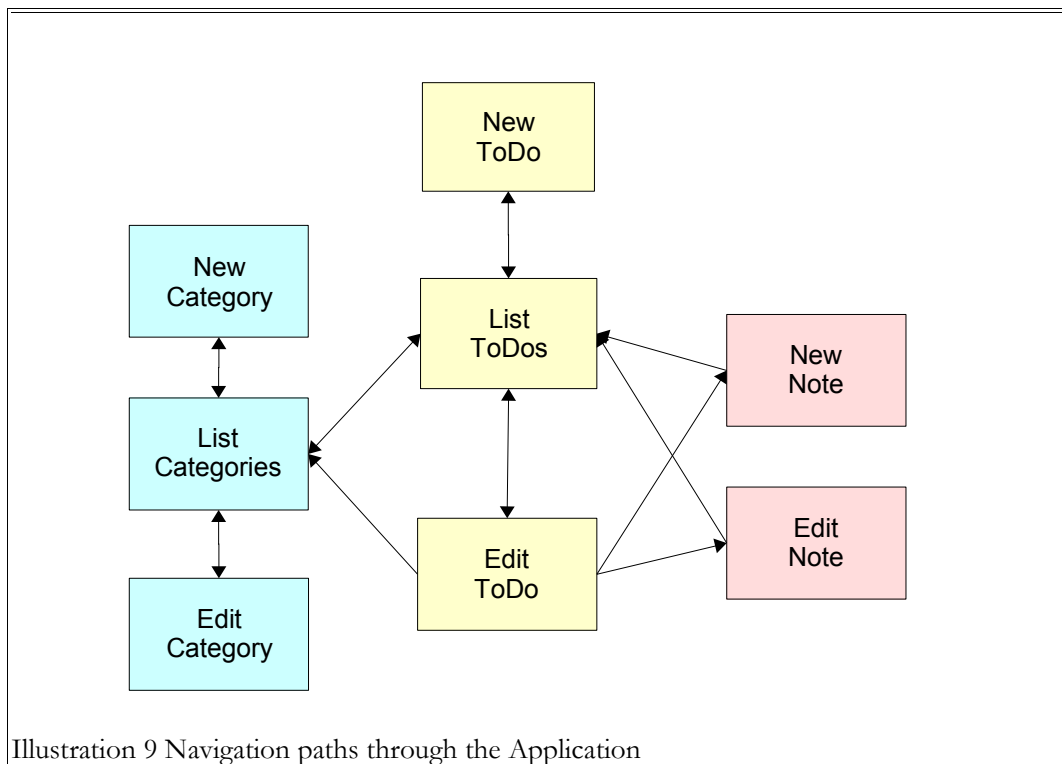
```

<% @heading = "Rename Category" %>
<%= error_messages_for 'category' %>
<%= start_form_tag :action => 'update', :id => @category %>
    <%= render_partial "form" %>
    <hr />
    <input type="submit" value="Update" />
    <input type="button" value="Cancel" onClick="parent.location='<%= url_for( :action
=> 'list' ) %>'>
<%= end_form_tag %>

```

Navigation through the system

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.



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 two steps:

- Add the home page definition to the Routes file:

config\routes.rb (excerpt)

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

- rename `public\index.html` `public\index.html.orig`

Downloading a Copy of this Application

If you'd like a copy of the 'To Do' 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!

Appendix – afterthoughts

After writing ‘Four Days’, I got a huge amount of feedback which greatly helped improve the quality of the document. One question did crop up repeatedly - “how do you update more than one record from the same screen” - so here’s an appendix covering this most Frequently Asked Question. It isn’t the easiest Rails concept to grasp, and it’s an area I would expect to see more “Helpers” appearing in the future.

Multiple Updates

In the screenshot below, the user can tick/untick multiple “To Dos” using the checkboxes in the extreme left hand column, and then press “Save” to store the results in the database.

| <input checked="" type="checkbox"/> | Pri | Description | Due Date | Category | | | | |
|-------------------------------------|-----|-------------------------------------|----------|---------------------|--|--|--|--|
| <input checked="" type="checkbox"/> | 3 | <td>Test escape</td> | 04/06/05 | Rails documentation | | | | |
| <input checked="" type="checkbox"/> | 1 | Buy roses & chocolates | 14/06/05 | Home & Family | | | | |
| <input type="checkbox"/> | 3 | Start next section of documentation | 17/06/05 | Rails documentation | | | | |
| <input type="checkbox"/> | 5 | Add new category button | 19/06/05 | Unfiled | | | | |
| <input type="checkbox"/> | 5 | Allow 1-click updating | 19/06/05 | Rails documentation | | | | |
| <input type="checkbox"/> | 1 | Monthly report for newspaper | 20/06/05 | Community Council | | | | |
| <input type="checkbox"/> | 1 | Post minutes on website | 21/06/05 | Community Council | | | | |
| <input type="checkbox"/> | 5 | Get quotes for painting house | 21/06/05 | Home & Family | | | | |
| <input type="checkbox"/> | 3 | Book Holiday | 28/06/05 | Home & Family | | | | |
| <input type="checkbox"/> | 3 | Buy new Lottery Ticket | 12/07/05 | Business | | | | |

Save New To Do... Categories...

Page: 1 2

Illustration 10: Multiple Updates

View

Rails supports multiple updates with another naming convention, which is to append the id of the record you are editing to the name within square brackets []. This enables you to pick out a particular record from multiple records on the screen.

Let’s work backwards from the HTML we are trying to generate. This is what it looks like for a record with id = 6:

```
<td style="text-align: center">
  <input type="checkbox" id="item_done" name="item[6][done]" value="1" checked />
  <input name="item[6][done]" type="hidden" value="0" />
</td>
```

(“checked” is omitted if the checkbox is not checked)

One way to generate this code is:

```
app\view\items\_list_stripes.rhtml (excerpt)
<td style="text-align: center">
  <%=check_box_tag("item["+list_stripes.id.to_s+"] [done]", "1", list_stripes["done"]==1)
  %>
```

```
<%=hidden_field_tag("item["+list_stripes.id.to_s+"][done]","0") %>
</td>
```

The parameters for `checkbox_tag` are `name`, `value = "1"`, `checked = false`, `options = {}`;
for `hidden_field_tag` `name`, `value = nil`, `options = {}`

Documentation: [ActionView::Helpers::FormTagHelper](#)

Plus of course we now need a Save button:

```
app\views\items\list.rhtml (excerpt)
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'updater' %>
<table>
...
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "New To Do...", {:type => 'button', :onClick=>"parent.location='" +
url_for( :controller => 'items', :action => 'new' ) + "'" } %>
<%= submit_tag "Categories...", {:type => 'button', :onClick=>"parent.location='" +
url_for( :controller => 'categories', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action => @params["action"]
|| "index" }) + "<hr />" if @item_pages.page_count>1 %>
```

Controller

What gets returned to the controller when you press the ‘Save’ button is the following hash:

```
params: {
  :controller=>"items",
  :item=> {
    "6"=>{"done"=>"0"},
    ... etc...
    "5"=>{"done"=>"1"}
  },
  :action=>"updater"
}
```

We’re interested in the `:item` bit. For example, the bold line means “the record with `id = 6` has the value of the `done` field set to 0”. From here, it’s a fairly easy job to update the `Items` table:

```
app\controller\items_controller (excerpt)
def updater
  @params[:item].each { |item_id, attr|
    item = Item.find(item_id)
    item.update_attribute(:done,attr[:done])
  }
  redirect_to :action => 'list'
end
```

each puts “6” into the variable `item_id`, and “done” => “0” into `attr`.

Ruby Documentation: [class Array](#)

This code works, but if you watch what is happening in `development.log`, you’ll see that Rails is retrieving and updating every record, whether it’s changed or not. Not only is this creating unnecessary database updates, but it also means that `updated_on` also gets changed, which isn’t really what we want. Much better to only update if ‘done’ has changed, but this means some coding :-)

```
app\controller\items_controller (excerpt)
def updater
  @params[:item].each { |item_id, contents|
    item = Item.find(item_id)
    if item.done != contents[:done].to_i
```

```
        item.update_attribute(:done, contents[:done])
      end
    }
    redirect_to :action => 'list'
  end
```

Note that we need to convert the string `done` to an integer using `to_i` so we can compare like with like. This is the kind of gotcha you can easily miss – it’s worth checking `development.log` from time to time to make sure Rails is doing what you expect.

User Interface considerations

This code works, and could be applied to make any field on the screen editable (another easy exercise for the reader :-). It does raise some interesting questions about what the user would expect. What if the user changes some check boxes, and then presses “New To Do..”, or re-sorts the display, without pressing “Save”? Should the system always “Save” before doing any other action? More easy exercises for the reader..

Still to be done

On page 24 I left `list_by_category` as an easy exercise for the reader. It proved to be less easy than it looked – in fact, I’m still looking for an elegant ‘Rails’ way to sort by a field in a lookup table. I ended up with this rather horrible code:

app\controller\items_controller (excerpt)

```
def list_by_category
  @item_pages = Paginator.new self, Item.count, 10, @params['page']
  @items = Item.find_by_sql 'SELECT i.*, c.category FROM categories c, items i ' +
    'WHERE ( c.id = i.category_id ) '+
    'ORDER BY c.category ' +
    'LIMIT 10 ' +
    "OFFSET #{@item_pages.current.to_sql[1]}"
  render_action 'list'
end
```

If anyone has a better solution, please let me know. I leave this code as a reassuring example that if all else fails, Rails will not leave you stuck but will allow you to resort to ‘old-fashioned’ coding!

Enjoy coding with Rails!

Index of Rails and Ruby Terms used in this Document

| | | |
|---|-----------|--|
| A | | |
| action_name..... | 11 | |
| B | | |
| before_type_cast..... | 20 | |
| belongs_to..... | 20 | |
| C | | |
| check_box..... | 28 | |
| check_box_tag..... | 38 | |
| confirm..... | 14, 23 | |
| content_columns..... | 14 | |
| content_for_layout..... | 11 | |
| created_at..... | 5 | |
| created_on..... | 5, 13 | |
| current..... | 15 | |
| D | | |
| date_select..... | 27 | |
| destroy..... | 10 | |
| destroy_all..... | 24 | |
| development.log..... | 3, 10, 39 | |
| E | | |
| end_form_tag..... | 12 | |
| error_messages_for..... | 12 | |
| F | | |
| find..... | 10 | |
| find_all..... | 10 | |
| Flash..... | 15 | |
| H | | |
| h..... | 14 | |
| helper..... | 11 | |
| hidden_field_tag..... | 38 | |
| HTML escape..... | 14 | |
| human_attribute_name..... | 14 | |
| human_name..... | 14 | |
| I | | |
| id..... | 5 | |
| L | | |
| Layout..... | 11 | |
| link_to..... | 12 | |
| link_to_image..... | 23 | |
| lock_version..... | 5 | |
| N | | |
| new..... | 10 | |
| O | | |
| options_from_collection_for_select..... | 28 | |
| P | | |
| paginate..... | 13 | |
| pagination_links..... | 17 | |
| Partial..... | 11 | |
| previous..... | 15 | |
| R | | |
| redirect_to..... | 10 | |
| Referential Integrity..... | 21 | |
| render_collection_of_partials..... | 25 | |
| render_partial..... | 12, 25 | |
| render_template..... | 10 | |
| rescue..... | 27 | |
| S | | |
| save..... | 10 | |
| select..... | 28 | |
| session variable..... | 33 | |
| start_form_tag..... | 12 | |
| strftime..... | 16 | |
| stylesheet_link_tag..... | 11 | |
| submit_tag..... | 12 | |
| T | | |
| Template..... | 11 | |
| text_field..... | 13 | |
| U | | |
| update_attribute..... | 38 | |
| update_attributes..... | 10 | |
| updated_at..... | 5 | |
| updated_on..... | 5, 13 | |
| url_for..... | 25 | |
| V | | |
| validates_associated..... | 20 | |
| validates_format_of..... | 20 | |
| validates_inclusion_of..... | 20 | |
| validates_length_of..... | 6, 20 | |
| validates_presence_of..... | 20 | |
| validates_uniqueness_of..... | 6 | |
| . | | |
| each..... | 38 | |