How To Create a Blog in Ruby In Steel (On Rails)

By Huw Collingbourne – July 2006

These notes accompany the movie in which we show you how to create a Ruby In Rails Blog Application using **Ruby In Steel:**

http://www.sapphiresteel.com/static/movies/steel-blog/ruby_in_steel_blog.html

If you follow the notes carefully you will be able to recreate this application on your own PC. This demo is based on the well-known Rails Weblog demo by David Heinemeier Hansson. You can view his demo on the Ruby On Rails web site:
http://www.rubyonrails.com/screencasts

Things To Note..

In our movie, you will see that Ruby In Steel provides a number of special features which ensure that every part of the development process is done inside the Visual Studio environment. This means that you don't have to navigate directories, open command prompts, run scripts from a prompt, run the server from a prompt and so on. If you use SQL Server, you don't even have to leave Visual Studio in order to create a database and modify the tables. In fact, in our demo the only time we leave Ruby In Steel is when we want to preview the results in a web browser. You will also see that we provide full support for Visual Studio editing (colouring, collapsing, indenting and so on) and simplify project management by automatically placing all the generated files and directories into the Solution Explorer.

Requirements

You need to have the following software installed (note the site addressees for downloads):

- **Visual Studio 2005** (standard edition or higher)

- **Ruby In Steel** (*beta 0.7 or later*)

Download Page:
http://www.sapphiresteel.com/Steel-Download-and-Change-Log
Installation Guide:
http://www.sapphiresteel.com/Installing-Ruby-In-Steel

- **Ruby**

http://www.ruby-lang.org/ *(or...)*
http://rubyinstaller.rubyforge.org/wiki/wiki.pl

- **Rails**

http://www.rubyonrails.com/

- **Either MySQL or SQLServer**

MySQL
http://dev.mysql.com/
SQL Server Management Studio Express
http://msdn.microsoft.com/vstudio/express/sql/

Notes on MySQL

If you are unfamiliar with the ins and outs of MySQL, you may find that a graphical user interface will be helpful when creating and manipulating databases. We have found the free edition of SQLYog to be very useful for this purpose. SQLYog can be downloaded from:

http://www.webyog.com/

Notes on SQL Server...

For SQL Server, you will also need the a copy of the file **ADO.rb**. We provide this file in the Ruby In Steel download Zip archive. You should copy ADO.rb into the **\lib\ruby\site_ruby\1.8\DBD\ADO** directory beneath your Ruby installation. For example, if Ruby is installed in **C:\ruby** you would need to copy ADO.rb into:

ruby\lib\ruby\site_ruby\1.8\DBD\ADO

You may need to create the **\ADO** directory. **NOTE**: If you do *not* use SQL Server, the file, ADO.rb, is not required.

We recommend that you launch the SQL Server Management application before you begin creating a new Rails project. This should help to avoid the possibility of a timeout when a database connection is initially attempted.

Check Your Installation Paths

Ensure that Ruby In Steel is aware of the locations of your Ruby interpreter and database server (MySQL or SQL Server). You can set these paths by selecting the Tools menu in Visual Studio, then '*Configure Steel'*. Browse to set the paths and click OK to save them.

Where To Go For More Help...

On Programming Ruby
http://www.sapphiresteel.com/The-Little-Book-Of-Ruby

On Installing MySQL

http://www.bitwisemag.com/copy/features/internet/webdev/mysql/mysql_install.html

On Using Ruby In Steel
http://www.sapphiresteel.com/-Software-

NOTE:

The demo movie was recorded with beta version 0.7 of Ruby In Steel. Both the user interface and the options available will change in later releases.

Step By Step Guide To Creating A Blog Using Ruby In Steel

These are instructions to create the same Blog application which we created in our demo movie. We give the instructions in numbered ***Do This*** steps followed by a section explains What ***Happens*** in each step and a ***Commentary*** on any special features of Ruby In Steel or of the underlying Ruby or rails development. To recreate this project we recommend that you follow the steps one by one using the instructions in this document and also **refer to the movie** (making judicious use of the the pause button in the movie playback!) to ensure that you are following every step of the way.... ***Note****: you can save yourself some time by cutting and pasting the programming code from this document into Ruby In Steel as required. To do this, click the* ***Select*** *icon (top of the screen in Acrobat) then mark and copy text from the document).*

View the movie at: **http://www.sapphiresteel.com/static/movies/steel-blog/ruby_in_steel_blog.html**

1. Create A Rails Project

Do This

Select the File menu, New, Project, Steel, Rails Project
Enter project name - e.g. "MyBlog" in the Name Field
Make sure 'Create Directory For Solution' is checked
Click OK

A dialog box pops up to prompt you to select a database sever, name your database and host.

a) If Using MySQL

Make sure *MySQL* is selected. Check off Development and Test.

(**NOTE**: The actual details will vary according to your local settings. You should, for example, enter the host, user name and password (if any) which you have previously specified when setting up your database server. The following are *sample* details only...)

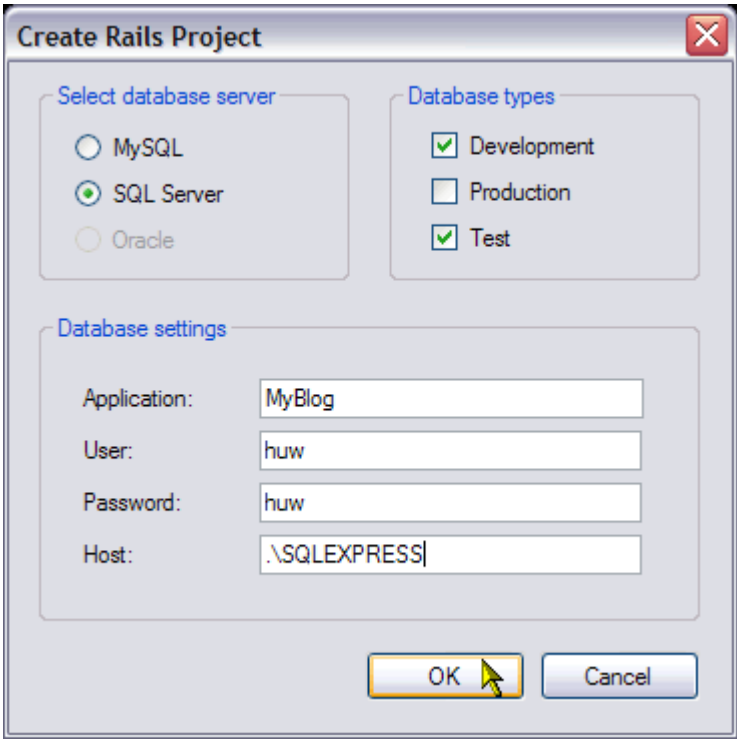
Enter:
Application = "MyBlog"
User = root
password = (leave blank)
Host = localhost

Click OK

b) If Using SQL Server

Make sure SQL is selected. Check off Development and Test.

(**NOTE**: The actual details will vary according to your local settings. You should, for example, enter the host, user name and password (if any) which you have previously specified when setting up your database server. The following are the details which we entered in the demo movie...)



These are the settings used in the demo

Enter:
Application = "MyBlog"
User = huw
password = huw
Host = .\SQLEXPRESS

Click OK

What Happens

Rails creates the files and directories which form the 'skeleton' of your application. It also creates the databases which you specified and places the necessary entries into the **config.yml** file which tells Rails which databases to use.

Commentary

Note that a new application is created using a New Project template – you don't have to open up a command window, navigate to a directory and run the Rails command from a prompt. The directories and files which you just create are automatically added to the Solution Explorer – there is no need to hunt around for them using the Windows Explorer

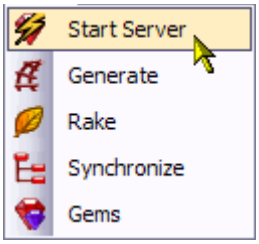
The creation of the database is done automatically. This avoids a number of separate steps which would otherwise be required. For example, you don't have to load up a separate utility in order to create each individual database, nor do you have to edit the Rails configuration file by hand. Ruby In Steel does it for you.

Note that we currently support SQL Server and MySQL and will be adding support for other databases later.

2. Run server:

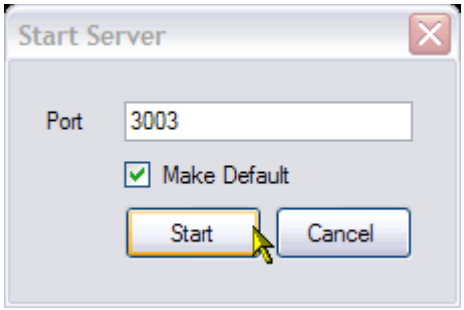
Do This

Select the Rails menu, Start Server.



The Rails menu gives quick access to a number of useful tools

A dialog pops up showing the default port (3000). This is normally OK. However, in some cases (if, for example, you already have a different Server on your PC), you may need to enter some other number here (e.g. 3003) to avoid a conflict. Finally press Start.



The Start Server dialog

*[**Note**: If you want to save a port number to be used automatically in future, be sure to click 'Make Default' in the server dialog.]*

Verify that the server is running and you are on Rails by entering the server name and port into a web browser.
In the demo movie, we enter:

http://localhost:3003

What Happens

We start the default Rails WEBrick server. This is required in order to let our web browser communicate with the Rails program. For example, you might enter into your browser a url which refers to your server (generally 'localhost') and its port (e.g. 3000) like this:

http://localhost:3000

The server communicates between the web page and Rails – possibly sending it some data from a web page to Rails. The Rails system can then process this data and construct new web pages which are passed back, via the server, to the web browser which then displays the new page.

Commentary

Usually in Rails when you want to start the server you have to open a command prompt and enter a command such as this:

ruby ./script/server –p3003

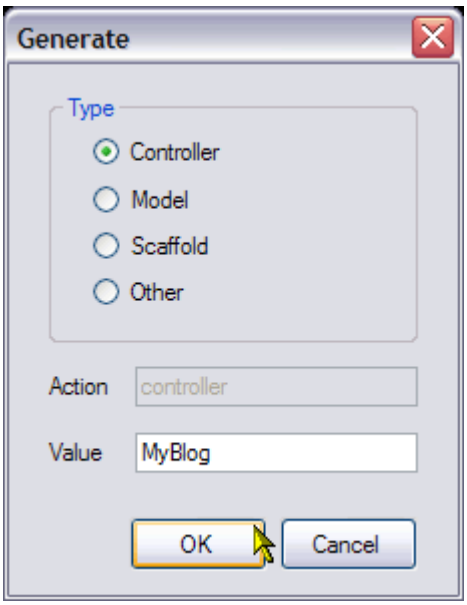
Here the optional final parameter is the port on which the server runs. In Ruby In Steel you can run the server using the Rails menu.

Rails applications can also use other servers, such as Apache. However, it may take some work to configure Apache to work with Rails. WEBrick is more or less 'ready-to-run' so is generally simpler to use at the outset. Currently we use WEBrick as standard but we shall add more server options in later versions of Steel.

3. Create a controller

Do This

Click Rails menu, then Generate (or click the *Generate* button on the Rails toolbar).
In the Generate dialog, make sure Controller is selected.
Enter controller name: (e.g. *MyBlog*)
Click OK.



Use the Generate Dialog to create controllers. Models and scaffolds

Expand Solution Explorer branch: **app\controllers**

What Happens

This step creates the controller, which is the Ruby file that will 'runs' the Blog application.

Commentary

The controller is created from inside Visual Studio in the correct directory. This avoids the need to open a command prompt, navigate around the disk to find the application directory and run a script such as:

ruby ./script/generate controller Blog

Notice that Solution Explorer is updated automatically as the new files and directories which you just created are added to the Solution hierarchy. For example, look under *app/controller* and you'll see **blog_controller.rb**; and under *app/helper* there is **helper.rb.**

Controller files contain Ruby code which may perform all kinds of different actions. Here we have created the 'main' controller for our MyBlog application – I've named the controller *Blog* – to which the suffix *_controller.rb* is appended automatically when the controller generate script executes.

4. Edit The Blog Controller

Do This

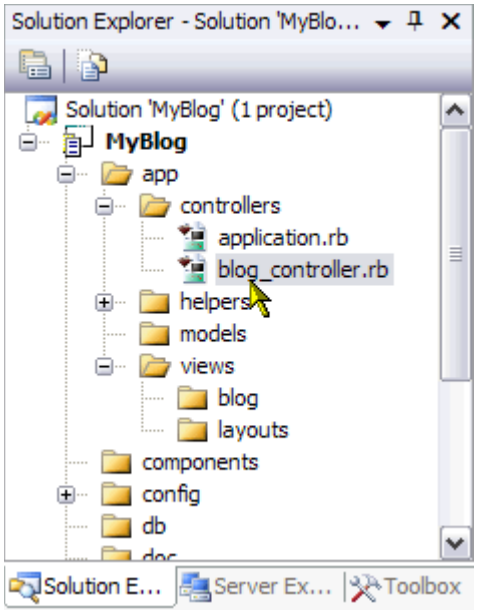
In Solution Explorer, under *app/controllers*…
Double-click **blog_controller.rb** to load it into the editor

Edit the code to match the following:

```
class BlogController < ApplicationController
      def index
            render :text => "Hello World!"
      end
end
```

Save.

The Solution Explorer organises all your files and directories



Test in Browser...
In your web browser, enter the host and port which you previously selected as the URL followed by **/Blog**. In the demo movie this is:

http://localhost:3003/Blog

This displays *"Hello world"*

Finally, in *blog_controller.rb,* Delete the entire **index** method, i.e. This code..

```
def index
      render :text => "Hello World!"
end
```

Save.

What Happens

The *index* method executes when someone enters the default URL of the Blog application into a web browser. Our method simply displays "Hello World!"

Commentary

Blog controllers may contain specific 'actions' – that is, Ruby methods which can be executed by specifying the method names in the URL. The index method of the Blog application could, for example, be executed by entering this URL (here I am using the port number, 3003, which I used when I ran the web server earlier; you may need to specify a different 'port' such as, for example, *localhost:3000*):

http://localhost:3003/Blog/index

However, the *index* method is special. It is treated as the default method and so its name may be omitted like this:

http://localhost:3003/Blog

Either way, the code in the index method is executed:

```
def index
      render :text => "Hello World!"
end
```

Here the *BlogController* class is a descendent of *ApplicationController* and it uses the *render* method of the *ApplicationController* class. Ruby programmers will recognise that the symbol **:text** forms the *key* part of a Hash item in which "Hello world!" is the value. The above code could be rewritten thus:

```
def index
      render( {:text => "Hello World!"} )
end
```

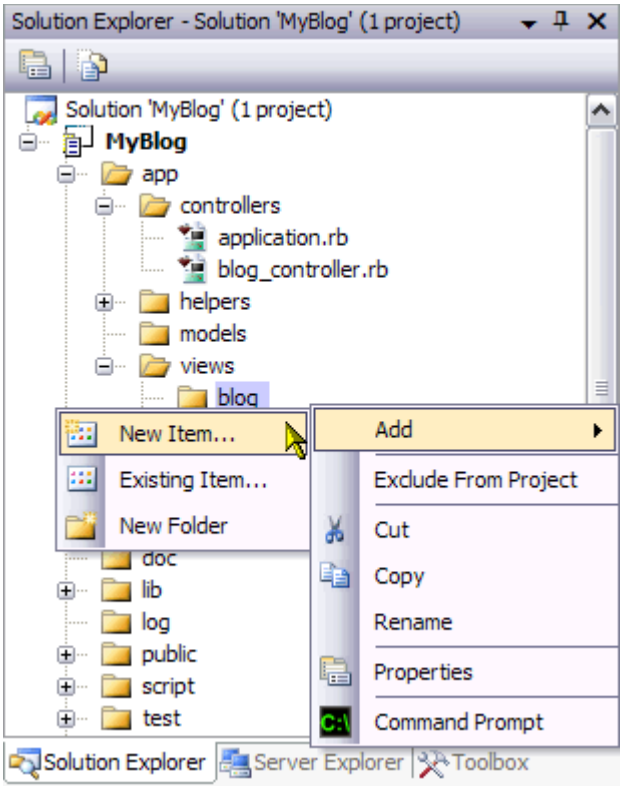
5. Create A template

Do This

Right-click *app/views/blog* directory in the Solution Explorer.
From the popup menu, select *Add* then *New Item*.
Pick the *empty_rhtml* Item

In the Name field of the dialog, name the file *index.rhtml*

Click Add



Right-click the Solution Explorer to add a new item

In this file (*index.rhtml*) enter this plain text:

hello from the template

Save the file.
Go to Web Browser and refresh.
This now shows "*hello from the template*"

What Happens

The *index.rhtml* file defines the default 'view template'. When no index method is defined in the Blog controller, Rails goes to the *index.rhtml* template to look for anything to be included in the HTML page which will be displayed in the browser.

Commentary

Here we've entered plain text into the template. We could just as well have entered any valid HTML. For example, if we wanted "Hello" to be shown in bold text and "Sapphire In Steel" to be displayed as a hyperlink leading to www.sapphiresteel.com, this is what we would enter:

```
<b>Hello</b> from
<a href="http://www.sapphiresteel.com">Sapphire In Steel</a>
```

Rails incorporates this fragment of HTML code into a fully formed HTML page which is then sent back to the browser.

Once again, *index*, forms an optional part of the URL which must be entered in order to display this page. Either of the following URLs will work:

http://localhost:3003/Blog/index
http://localhost:3003/Blog

If, on the other hand, you had called your rhtml file *myview.rhtml*, you would need to enter this URL:

http://localhost:3003/Blog/myview

6 & 7 (*There is No Step 6 or 7*)

In the original Rails Blog demo you would now have to enter database configuration details into the **config.yml** file. You would then have to create a MySQL database as a separate operation. Ruby In Steel did all that for you when you first created your project so you can sit back and have a cup of tea at this stage ;-)

8. Create tables

Do This

a) In MySQL

If you are already familiar with MySQL, use your usual method of creating the column definitions in this step. If not, we suggest you use a simple front-end such as SQLYog (http://www.webyog.com/).

Using SQLYog...
Create columns as follows:

id int 11 Not Null=yes, auto-increment
title varchar 255

Right-click: *create table*
Click *Create table button*
Enter **posts**
Save

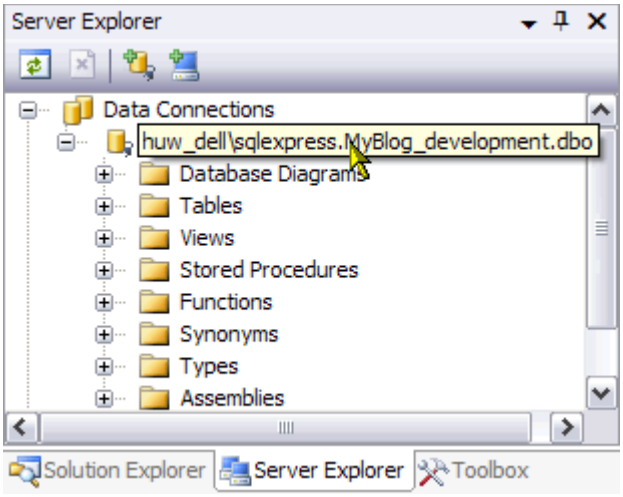
b) In SQL Server

If the **Server Explorer** is not visible in Ruby In Steel, press CTRL+W, L to show it. Right-click *Data Connections*. Select *Add Connection*. If Microsoft SQLServer (SqlClient) is not shown as Data Source, click the *Change* button and select Microsoft SQL Server. Then click OK.

In the *Add Connection* dialog, enter the server name (which you should have specified when installing SQLServer and starting this Rails application): e.g.
.\SQLEXPRESS
Select *Use SQL Server Authentication.*
Enter your user name and password for SQL Server. Once again, you need to have set these up earlier. In the Demo movie, I enter …

(name) huw
(password) huw

Select the database name from a drop down list
(e.g. **MyBlog_development**).
Click *Test Connection*.
All being well, click OK. (If you have errors, you will need to check the installation, name, password details etc. of SQLServer).



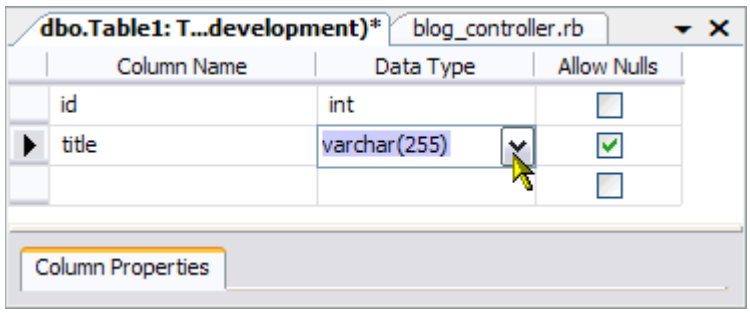
Use the Server Explorer to work with a SQLServer database

Click the database in Server Explorer to open its branches.
Right-click the Tables branch.
Select *Add New Table*

You will see a database grid labelled:
Column Type | *Data Type* | *Allow Nulls*

Enter two rows into these columns as follows:

Column Type	*Data Type*	*Allow Nulls*
id	**int**	(*NOT Selected*)
title	**varchar(255)**	



Use integrated table editing with SQL Server

Right-click the **id** row and select *Set As Primary key*

In Table Designer Properties panel set *Identity Column* as **id** (this causes auto increment).

Press CTRL+S to save table. Enter the name 'Posts'
Click OK

What Happens

This step sets up a database called *MyBlog*, creates a table called *posts* and defines two columns – or data 'fields' – a unique id field and a text field for each Blog entry.

Commentary

You will either need MySQL or SQL Server installed. MySQL is free. Microsoft has a free edition of SQL Server called SQL Server Management Studio Express To use Rails with SQLServer you will also need to install the file ADO.rb (*refer to the requirements and notes given at the start of this document*).

Currently, SQL Server integrates more fully into Visual Studio, so that you can create your data table definitions in the VS 2005 environment. A more integrated version of MySQL is under development however.

9. Generate Model

Do This

Click Rails menu. Select *Generate*.
In the Generate dialog, select *Model*
In value field enter: *Post*
Click OK

View Solution Explorer
The file *post.rb* is added to *app/models branch of* the Solution Explorer

What Happens

We have run a Rails script to create a 'model' for posts. A model is used to interface with data in the database. Here the post model will interface with the posts table.

Commentary

By now it should be no surprise that the Post model is generated using a dialog box rather than running a script and that the Solution Explorer is synchronized automatically to show the newly generated file.

10. Use Scaffold

Do This

Double-click **Blog_controller.rb**
Edit it to this by adding (just beneath **class BlogController < ApplicationController**), the following**:**

scaffold :post

Save
Go back to Web browser and refresh.

What Happens

Here **scaffold** is a method of the Rails *ActionController* class (from which the parent class of our BlogController (that is, *ApplicationController*) descends. The **scaffold** method adds various database access routines to the controller; it uses the specified symbol (**:post**) to create an instance variable with a similar name, (here **@post**) and also identify the model class (**Post**) to use.

Commentary

Scaffolding is a simple way of creating the basic skeleton of an application using a model. It provides access to routines to show and edit data in a web browser. In Rails, this is the simplest way of displaying and interacting with a database via a browser.

11. Create, show and edit a new post

Do This

In the browser..
click the *New Post* link

Enter some text: e.g. *"Hello East Grinstead!"*
Click *Create*

Click Show

Click Edit

Alter text: e.g. *"Goodbye East Grinstead!"*

Click Update

What Happens

We have now entered and edited our first record into the Posts table of the Blog database.

Commentary

The editing and updating features have been provided by the Rails scaffold which links the Post model to the Posts table. Each Every time we create a new post it is entered as a row in the Posts table. The scaffold provides the HTML elements required to display and edit posts in a web browser.

12. Add more database columns

Do This

(**NOTE**: From now on, these instructions assume you are using SQL Server. *If you are using **MySQL*** you will need to manipulate the database tables and records using whichever tools you used in **Step 8**….)

In Visual Studio…

In the *Posts* table editor tab
add new column:

column name	*data type*
body	**text**

Save (press *Ctrl+S*)

In the Web Browser...
Click *Edit* (to show the body)

In the text field, enter some text such as...
Hello, hello, hello...

Click *Update*

In Visual Studio….
In the *Posts* table editing tab….

Insert a new field ABOVE *body* field (i.e. right click *body* and select 'Insert Column'). Add...

Column name	*Data Type*
created_at	**datetime**

Save

In Web Browser…
Click *Edit* to edit a post.

Change the date by picking a new data from the combo list.

Click Update

What Happens

You have modified the database by adding new columns and the changes have propagated through to the view in the browser.

Commentary

Currently table editing from within Visual Studio itself is only supported for MS SQL Server. MySQL has announced plans to integrated with Visual Studio, however, and when this is done it should be possible to implement a similar feature for MySQL databases. We shall be adding a broader range of database support to subsequent versions of Steel but in many cases it may be necessary to use a standalone tool to create or manipulate the database structure.

13. Add Validation

Do This

In Visual Studio…
Using the Solution Explorer…
Open **models/post.rb**

add (just after : **class Post < ActiveRecord::Base**)

validates_presence_of :title

Save

In Web Browser...
Click Back (if necessary) to get back to List view
Click New Post.
Click Create (i.e. try to add new post with blank title)
The Browser shows error message.

Add some text to the Title and Body fields: e.g.

Title: My Lovely Title
Body: Ah, that's better…

Click Create

What Happens

You have added validation to the title field to prevent the user entering a blank title. If a blank is entered an error message will be shown in the browser.

Commentary

Here, **validates_presence_of** is a method of the Rails *ActiveRecord* class. It checks if a named attribute of an object (which corresponds to a database record) is blank. The name is specified using a Ruby symbol such as **:title**.

14. Generate Scaffold

Do This

In Visual Studio…
Click the *Generate* icon (or select *Generate* from the *Rails* menu).

In the Generate dialog select *Scaffold*
Enter value: **Post Blog**

Click OK

If *blog_controller.rb* is still open you will be shown a message that it has been changed outside of Visual Studio and asked if you want to reload it – click '*Yes To All*')

View **blog_controller.rb** to show all the actions that have been added.

What Happens

The *Scaffold* script generates a whole load of code 'actions' and template pages to create, show and view posts in simple layout inside a web browser.

Commentary

Once again, this is all done using a Ruby In Steel dialog box instead of having to use a command prompt. Notice that the template (RHTML) files are now available beneath the *\views\blog* directory in the Solution Explorer.

15. Edit template

Do This

Open **\view\blog\list.rhtml** in Solution Explorer

Change:

```
<h1>Listing posts</h1>
```

To:

```
<h1>My Wonderful Weblog</h1>
```

Delete all the rest of the file.

Cut and paste the following:

```
<% for post in @posts %>
  <div>
      <h2><%= link_to post.title, :action => 'show', :id =>
post %></h2>
      <p><%= post.body %></p>
      <p><small>
            <%= post.created_at.to_s %>
      </small></p>
  </div>
<% end %>


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

<br />

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

Save

Go back to Web Browser and refresh

In Visual Studio…

In **list.rhtml** edit first line to the following: (i.e. add **.reverse**)

```
<% for post in @posts.reverse %>
```

Save
Refresh browser

What Happens

Here we have added some Ruby code embedded in a Rails RHTML template. The Ruby code displays the data of each post which it finds in the database we created earlier. It also adds links to help us navigate in the web browser view and to add a new post.

Commentary

The **created_at** method which we used above (**post.created_at**) was generated by Rails to match the **created_at** database column which we added earlier. The variable **@posts** is an array of **Post** objects and the **reverse** method is a standard Ruby method to reverse the array.

Here we are creating templates for various bits of a complete HTML page. The HTML contains Ruby programming code inside ares delimited by **<%** or **<%=** and **%>**. The page itself will be constructed by Rails and sent to the browser to be displayed when a user navigates to that view (for example, as a result of click a button or hyperlink). By the time the page appears in the browser the Ruby code will have been stripped out and replaced with valid HTML such as text or data extracted from the database.

Notice that Steel automatically colour codes RHTML files and implements code collapsing to let you hide blocks of HTML code such as DIVs and paragraphs.

16. Use Textile Text Formatting

Do This

In Web Browser…
Click a message header (e.g. *'My Lovely Title'*)
Edit message to show Textile markup…
Click the Edit link
In the body enter:
***Hello* _world_**
Click *Edit*
Click *Back* (i.e. make sure the *blog/list* view is shown in the browser)

In Visual Studio…
In **list.rhtml**...

edit first **<p>** line (probably this will be on line 6 or thereabouts) to:

```
<p><%= textilize(post.body) %></p>
```

Save

In *Web Browser* refresh to show bold and italic.

What Happens

We have used some Textile formatting commands * and * for bold and _ and _ for italic then applied these in the list view using the **textilize** method in **list.rhtml**

Commentary

This requires the installation of an add-on package called *RedCloth*. If you haven't already got this installed you can install this using the Ruby In Steel Gem dialog. This is how that is done... From the *Rails* menu select *Gems* then, when the dialog appears, enter:

redcloth --version "3.0.4"

You need to be online to retrieve a Gem like this. All being well, the gem will be downloaded and installed. The equivalent way of installing this Gem from the command prompt would be:

gem install redcloth --version "3.0.4"

17. Customise View

Do This

In Visual Studio…
*In **blog_controller.rb**…*

Delete code of the *list* method and edit it to this:

```
def list
      @posts = Post.find(:all)
end
```

Save

In **list.rhtml**
Delete this line (probably on line 3)...
`<% for post in @posts.reverse %>`
...and the matching `<% end %>` (probably on line 10 or 11).

Then ***cut*** this...

```
<div>
<h2><%= link_to post.title, :action => 'show', :id => post
%></h2>
<p><%= textilize(post.body) %></p>
<p><small>
      <%= post.created_at.to_s %>
      <%= post.class %>
</small></p>
</div>
```

Save **list.rhtml**

Right-click *\views\Blog* and select *Add, New Item*
Pick *empty_rhtml*
Name to: **_post.rhtml**
Click *Add*

Now ***paste*** the code (that we just cut from *list.rhtml*) *into* into **_post.rhtml**

Save

In list.rhtml…
Delete all but first and last line. Add this:

```
<%= render :partial => "post", :collection => @posts.reverse
%>
```

(**Check this**: The code in **list.rhtml** should now be as follows:

```
<h1>My Wonderful Weblog</h1>

<%= render :partial => "post", :collection => @posts.reverse
%>

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

Save

Load **show.rhtml**

Delete this...

```
<% for column in Post.content_columns %>
<p>
  <b><%= column.human_name %>:</b> <%=h
@post.send(column.name) %>
</p>
<% end %>
```

Add this:

```
<%= render :partial => "post", :object => @post %>
```

Save

In Web Browser…
(Make sure you are in the *list* view showing more than one post)…

Click an item header (e.g. *'My Lovely Title'*)

Note, the *Show* view (in which just one post is shown) now uses the *_post.rhtml* template

What Happens

A *view* is the web page representation of your application and the user's interface to your data. Here the *list* and *show* templates are changed; the view

of an individual post has been cut out of the *list* view definition and put into its own file. This means that it is available to be used in another view – on other web pages – if we so wish. But the *_post.rhtml* view is incomplete – it doesn't form a workable html page. So the *blog_controller.rb* Ruby file has to be told that the "post" view is only part of a complete view.

Commentary

The magic that goes on here can be summarized in this line:

```
<%= render :partial => "post", :object => @post %>
```

Here **render** is the name of a Rails method. When passed the symbol **:partial**, it knows that it only needs to update a specified part of a page. Here the "post" view is specified.

In this example, the partial template for a post is used in both the *list* and the *show* page templates.

18. Add Comments

Do This

In Visual Studio…
To Generate the Comment model…

Click the Generate Icon
Select *Model*

Enter as Value:
Comment

Click OK

In VS Solution Explorer…
Open **models/comment.rb**

Edit method to…

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

Save

Open *models/post.rb*

Edit to:

```
class Post < ActiveRecord::Base
      validates_presence_of :title
      has_many :comments
end
```

Save

What Happens

We generate a Comment model and edit its ruby file, **comment.rb**, to associate Comment with post.

Commentary

The Rails *ActiveRecord* **belongs_to** method is passed a symbol identifying the type of object to which comments belong. The **has_many** method followed by the **:comments** symbol, tells the Post class that each Post may have many associated Comment objects.

19. Create Comments Table and Display Comments

Do This

In Visual Studio...
Switch to Server Explorer
Find the **Tables** branch of database
Right-click.
Select *Add New Table*

Add these columns...

Column Name	*data Type*	*Allow Nulls*
id	**int**	*Allow Nulls = NOT Selected*
body	**text**	
post_id	**int**	*Allow Nulls = NOT Selected*

Right-click **id** and select *Set As Primary key*

In Table Designer Properties panel set *Identity Column* as **id** (this causes auto increment).

Press CTRL+S to save table. Enter the name '*Comments*'

Click OK

To enter some data….
Right-click **Comments** table in Server Explorer
Select: *Show table Data*
In Table Data display add a comment…

(NULL) / **That's very interesting!** / **1**

Press Enter to go to next line (which saves the new data)

Go to Solution Explorer…
Load *\views\show.rhtml*

At very bottom of this file, enter:

```
<h2>Comments</h2>

<% for comment in @post.comments %>
      <%= comment.body %>
      <hr />
<% end %>
```

Save

In Web Browser…
Click *Back* to get to List view.
Click the last entry (*"Goodbye East Grinstead"* which happens to have ***id*** *= 1*)
The new comment should be appended in this view (if not, you may have used the wrong **post_id** set in the Comments Table or clicked the wrong post in the browser. In which case, try clicking the other post!)

What Happens

We create a database table for Comment records.

Commentary

The names of the database columns need to match the names specified in the Ruby code. As ever, if you are not using SQL Server, you will need to create an alter the table using a standalone tool.

20. Create An 'Add Comment' Form

Do This

In Visual Studio…

Go back to **show.rhtml** and add this code at the very bottom:

```
<%= form_tag :action => "comment" %>
      <%= text_area "comment", "body" %><br />
      <%= submit_tag "Comment" %>
</form>
```

Save

Go back to *Web Browser* with a single entry in view. Refresh to show the comment entry field at the bottom.

In Visual Studio…

Load *blog/show.rhtml*

Edit **form_tag** at the bottom as follows:

```
<%= form_tag :action => "comment", :id => @post %>
```

Save

(Now, let's create a new action called 'Comment' to add the comment to the post.)

In **blog_controller.rb** add (under the *index* method), add:

```
def comment
      Post.find(params[:id]).comments.create(params[:comment])
      flash[:notice] = "Added your comment"
      redirect_to :action => "show", :id => params[:id]
end
```

Save

Go back to Web browser. Refresh. Type a comment such as:

Yes, I agree!

Click the *Comment* button to add your comment.

Note that the 'flash' text has appeared at top to tell you that the comment has been added.

What Happens

We have to alter the **show.rhtml** template to provide a user interface which lets people enter new comments via a web browser.

Commentary

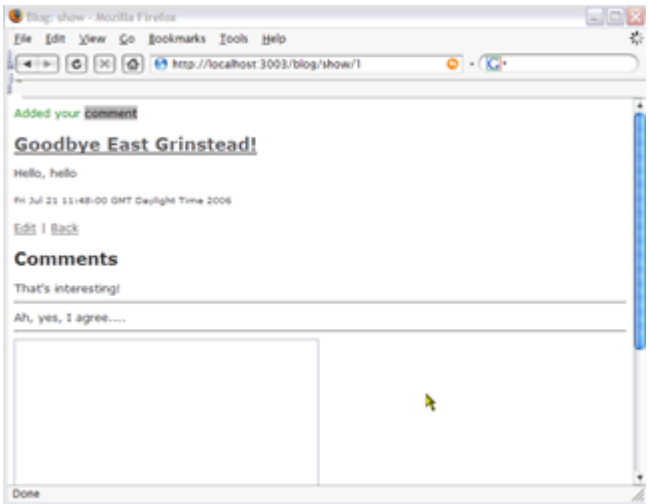
Here we add a **for** loop in the form of Ruby code embedded into HTML. The loop iterates over the comments associated with a given post (**@post.comments**) and displays the comment text (**comment.body**). Comments are added using the **form_tag** method; given the symbol, **:action** and the string *"comment"*, this will have the effect of posting any data entered and creating a new comment record in the database. The Comment action which actually submits the data is created in *comment.rb*. This takes the data entered and submits it as a comment on the current post - as specified by the post **id** in **show.rhtml**…

```
<%= form_tag :action => "comment", :id => @post %>
```

The Ruby **comment** method finds the Post specified by **:id**, gets its existing array of comments and adds the new one comment to it…

```
Post.find(params[:id]).comments.create(params[:comment])
```

...and that concludes our quick guide to creating a Ruby On Rails weblog using Ruby In Steel.



A view of the blog in a web browser, complete with added comments!