Rails101s

sdlong

Published
with GitBook

Table of Contents

原作： xdite (原著)

改編： sdlong

原始碼

blog 位置

本書內容為 Rails 101 without Bootstrapper

不用 Bootstrapper, 一切從 0 開始建置出 Rails 101 裡的專案 讓學 Rails 的新手可以減少遇到地雷的機會

並且從中理解到 Bootstrapper 裡的預設設定 (例如裝了哪些 gem) 是有什麼意義的

重新排版與改編教程，延伸閱讀的註解 幫助各位用循序漸進，易讀的方式理解 Rails 101 裡面的 Code 的真正價值！

Rails Outreach 是個志於讓大家快速學習 Rails 的推廣活動

以在國外非常熱門流行的 RailsBridge 為教材 帶領完全沒接觸過網站開發 / Rails 相關資訊的新人，有個快速進入狀況，體驗 Rails 威力的學習方式

RailsBridge 中文版是由 鴨七 翻譯並 Open Sources 我們需要您的一臂之力，繼續翻譯其他尚未翻譯的章節。

Rails Outreach 從 2014 年 05 月 台北場開始，短短二個月內陸陸續續在新竹、台中、台南、高雄等地開辦除了您的參與學習，也鼓勵您成為下一場活動的種子教練，帶領更多人進入 Rails 的世界

Rails Outreach 活動時間表

Rails Taiwan

Rails Taiwan是源於2011年6月開始， 每週二固定在臺北舉辦的Rails Taipei Meetup 社群， 專注分享與交流網站開發的技術與心得。

Rails Taiwan 社群網站	網址
Rails Taiwan 討論版	http://forum.rails-taiwan.org/
Learn Rails Today	http://learn-rails.today/
Meetup Rails Taiwan	http://www.meetup.com/rails-taiwan/

有任何問題，包括文字 / 排版 / 程式 / 解釋錯誤，請聯絡本人修正，感謝！

本章將用 Rails 本身內建的 鷹架(Scaffold) 模式

快速建置出一個有留言板功能的網站

並且用 Bootstrap 做簡單的套版

快速體驗 Ruby on Rails

原文網址

建立一個全新專案

打開 Terminal or iTerm2

`rails new first-app` 建立一個新的 Rails 專案

`cd first-app` 進入 first-app 專案資料夾

`git init` 建立 git 版本控制

`git add .` `git commit -m "Commit Initial"` 第一次 Commit (存檔)

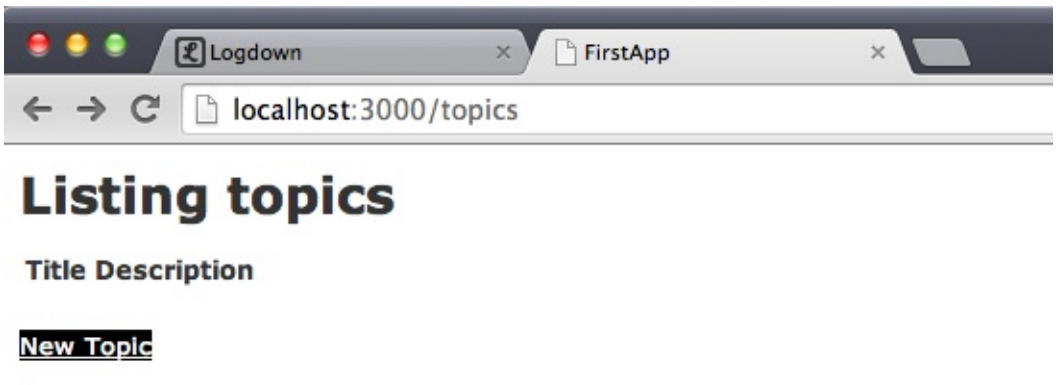
使用鷹架(Scaffold)功能

`rails g scaffold topic title:string description:text` 使用鷹架建置一個叫 topic 的簡易留言板

`rake db:migrate` 建立資料庫

`rails s` 開啓 server 模式

打開瀏覽器 網址: localhost:3000/topics



即可操作簡易的留言板功能

建立 welcome 頁面

插入一段 code

原本

app/controllers/topics_controller.rb

```
class TopicsController < ApplicationController
  before_action :set_topic, only: [:show, :edit, :update, :destroy]

  # GET /topics
  # GET /topics.json
  ...
  ...
```

插入

app/controllers/topics_controller.rb

```
class TopicsController < ApplicationController
  before_action :set_topic, only: [:show, :edit, :update, :destroy]

  def welcome
  end

  # GET /topics
  # GET /topics.json
  ...
  ...
```

設定 **routes**

原本

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics
  ...
  ...
```

插入

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics

  get 'welcome', to: 'topics#welcome'

  ...
  ...
```

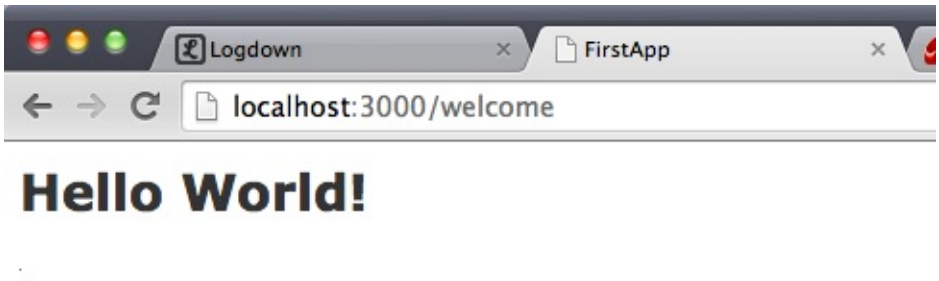
建立 **welcome.html.erb**

到 app/views/topics/ 資料夾裡建立一個檔案

app/views/topics/welcome.html.erb

```
<h1> Hello World! </h1>
```

打開瀏覽器，網址: http://localhost:3000/welcome



設定首頁為 http://localhost:3000/topics

原本

config/routes.rb

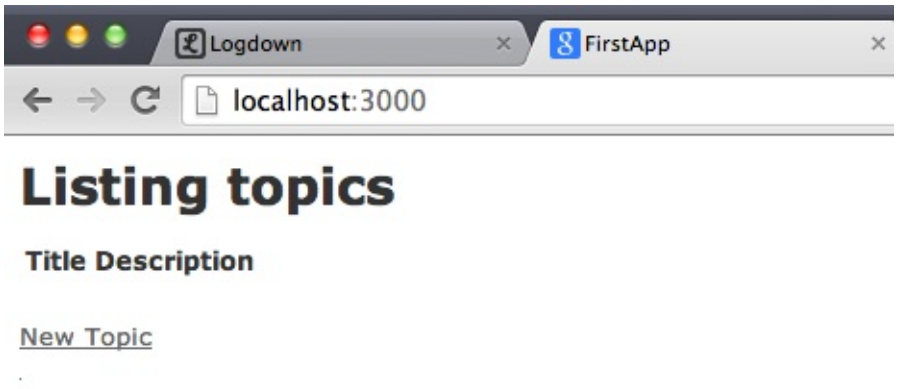
```
Rails.application.routes.draw do
  resources :topics

  get 'welcome', to: 'topics#welcome'

  ...
  ...
```

改成

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics

  get 'welcome', to: 'topics#welcome'
  root 'topics#index'
  ...
  ...
```

打開瀏覽器，網址: http://localhost:3000


Logdown
FirstApp
localhost:3000
Listing topics
Title Description
New Topic

使用 **Bootstrap** 做前端套版

原文網址

http://getbootstrap.com/

Bootstrap 是目前網站開發裡最好用的前端 CSS 套件 在 Rails 可以使用 ruby 函式庫 (gem) 來把 Bootstrap 裝進我們的專案裡面

安裝 **gem 'bootstrap-sass'**

原本

gemfile

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.1.1'
...
...
```

插入

gemfile

```
source 'https://rubygems.org'

gem 'bootstrap-sass'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.1.1'
...
...
```

`bundle install` 安裝 gem

將 **Bootstrap** 的 **CSS** 套件裝進專案裡面

原本

app/assets/stylesheets/application.css

```
/*
 * ...(一堆註解)
 *= require_tree .
 *= require_self
 */
```

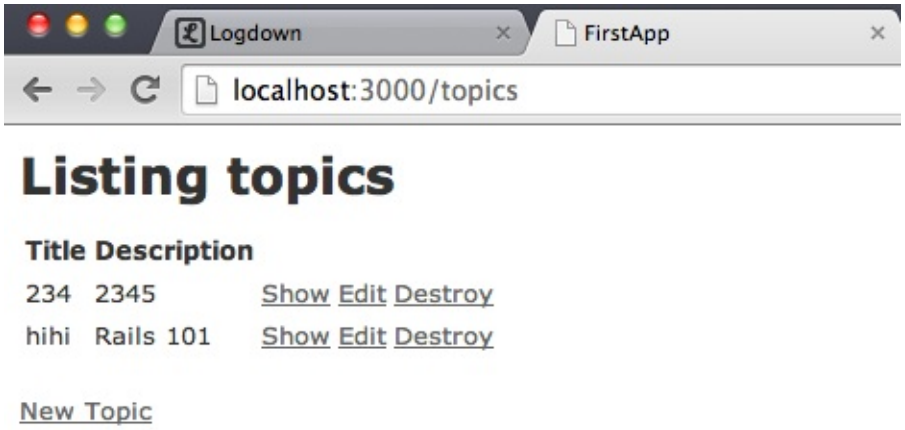
插入

app/assets/stylesheets/application.css

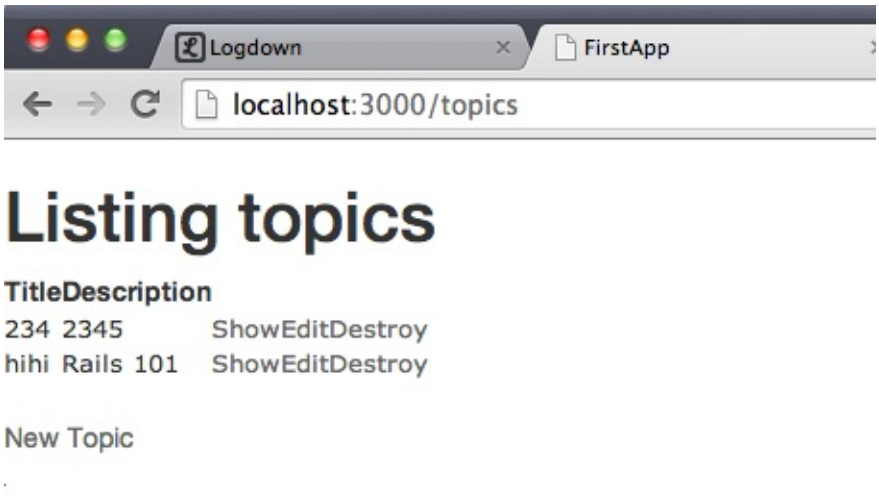
```
/*
 * ...(一堆註解)
 *= require_tree .
 *= require_self
 */

 @import "bootstrap";
```

記得重開 `rails s`

使用前



使用後



似乎改變不大？ 那是因為我們還沒套入 Bootstrap 到 html 裡面去

幫你的專案裝上 **Navbar** 跟 **Footbar**

到 app/views 增加一個資料夾 : common

增加二個檔案

app/views/common/_navbar.html.erb

```
<nav class="navbar navbar-default" role="navigation">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <a class="navbar-brand" href="/">Rails 101</a>
    </div>

    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav navbar-right">
        <li> <%= link_to("登入", '#') %> </li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
```

app/views/common/_footer.html.erb

```
<footer class="container" style="margin-top: 100px;">
  <p class="text-center">Copyright ©2014 Rails101s
    <br>Design by <a href="http://sdlong.logdown.com" target=_new>sdlong</a>
  </p>
</footer>
```

修改你的 **app/views/layout/application.html.erb**

原本

app/views/layouts/application.html.erb

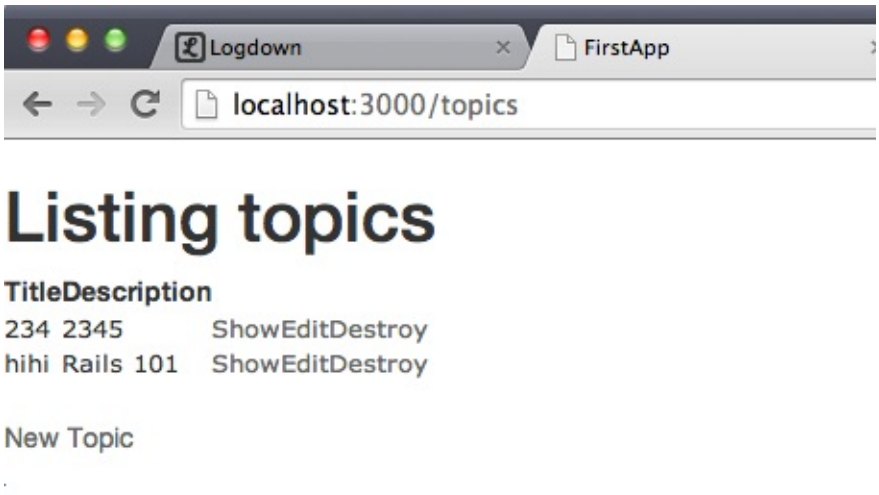
```
...
...
<body>

<%= yield %>

</body>
</html>
```

改成

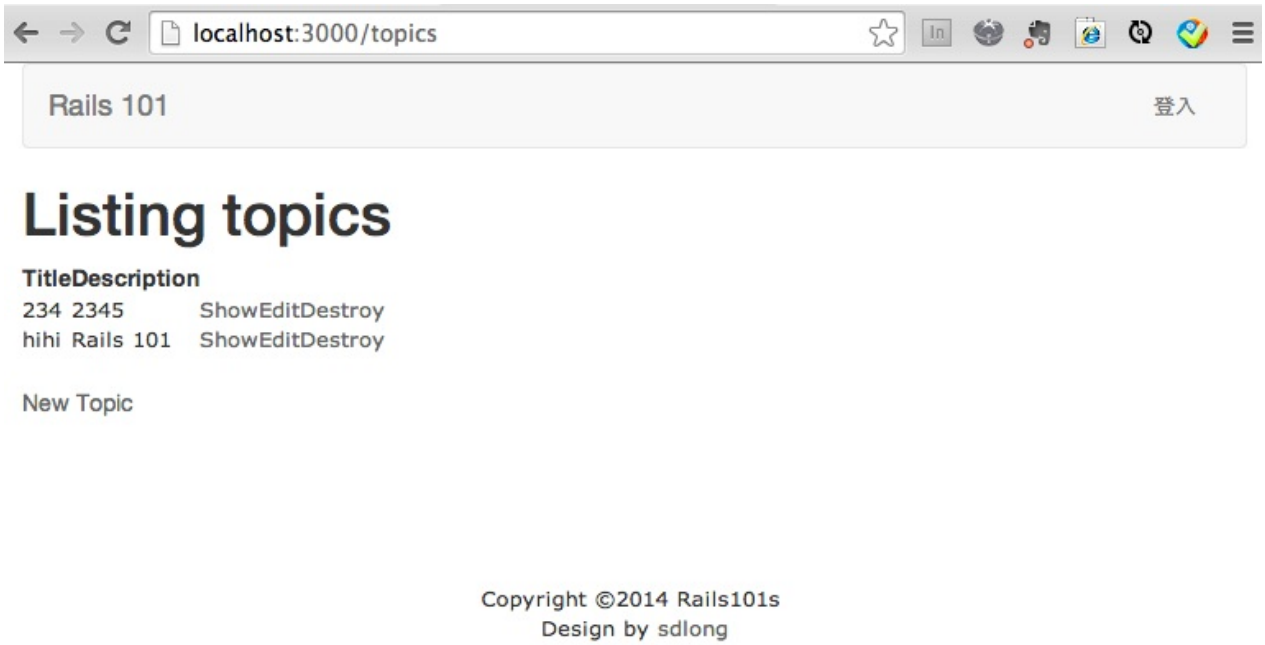
app/views/layouts/application.html.erb

```
...
...
<body>

<div class="container-fluid">
  <%= render "common/navbar" %>
  <%= yield %>
</div>
  <%= render "common/footer" %>

</body>
</html>
```

before



after



代入 **bootstrap** 的 **table** 套件

原本

app/views/topics/index.html.erb

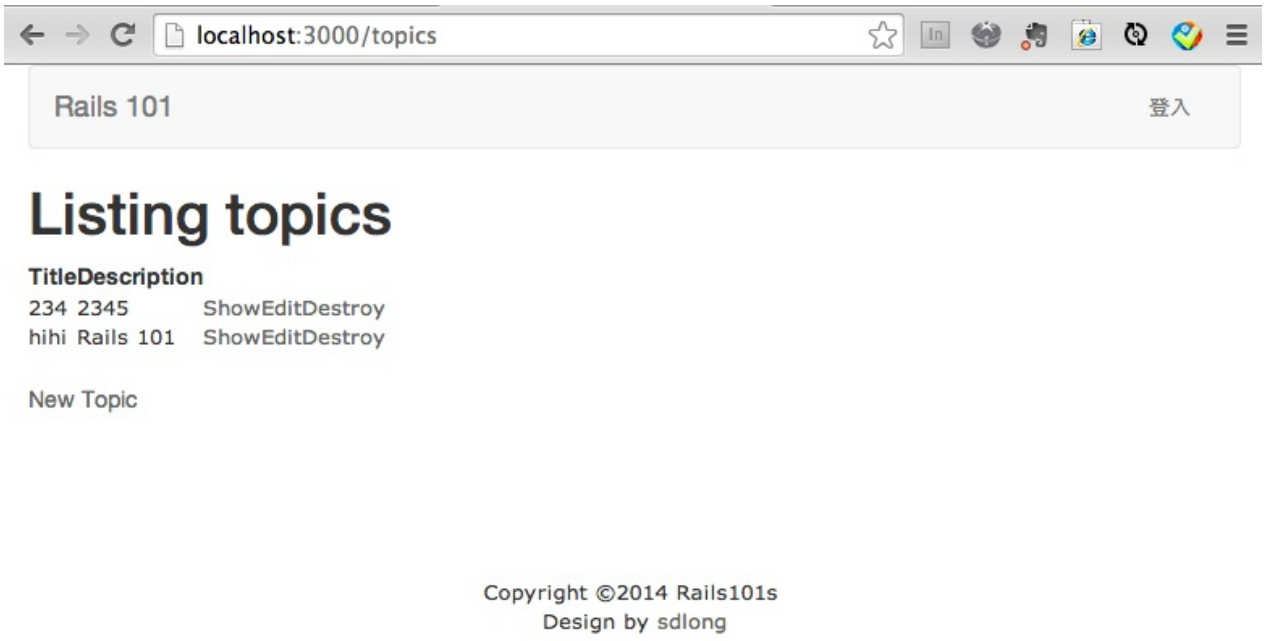
```
<h1 >Listing topics</h1>

<table>
  <thead>
...
...
```

改成

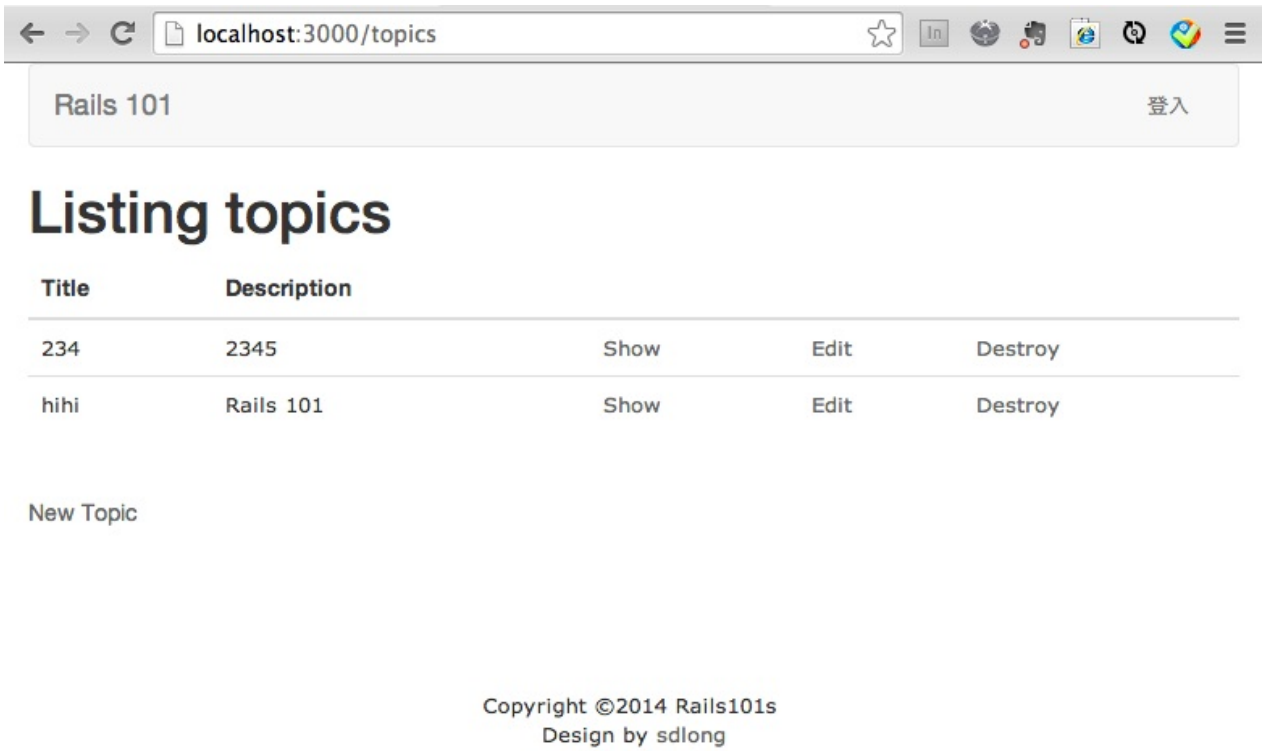
app/views/topics/index.html.erb

```
<h1 >Listing topics</h1>

<table class="table">
  <thead>
...
...
```

before



after



更多套件請參考 http://getbootstrap.com/

進入真正的主題前的須知

原文網址

Why should not use scaffold?

在前二篇，我們體驗了用 Rails 內建的鷹架模式快速地做了一個有 『 *CRUD* (註) 』 功能的留言板

Scaffold 是個很方便的功能，但如果您想成為一個稱職的 Rails developer

必須要學會如何 從零做出一個 **CRUD** 功能

有很多很酷的功能，都是由最基本的 CRUD 設定去做延伸，

如果太依賴 Scaffold ， 會讓您很難理解如何學習、創造出來

How to use this blog for lean Rails ?

強烈建議一定要『 親自動手做 』

並請把『 遇到bug 』當成是一個非常正常的事情

debug 的過程，才是您進步的原動力

在開始真正的教程之前，

建議您先把環境設定好

Ruby on Rails 最佳開發環境建置

(以下是預設您使用 Mac) Ruby on Rails 最佳裝機實務

Sublime Text 2 設定

下載並安裝 iTerm2, 打開 iTerm2 做 終端機設定

以上是基本中的基本，其他更多好用的設定、套件未來會另闢專區分享

我卡關了！要如何找到人求救？

歡迎您到各地 Rails Meetup 帶著你的筆電與問題找人詢問

如果您要在網路上找人詢問求救， xdite 有寫過一篇『 發問的藝術 』

供您參考，絕對會大幅提高找到人回應您問題的機會

> ### 為什麼上網發問沒有人人要理你?
>
> - 不知道你的作業系統與 Ruby 版本
> - 不知道你的錯誤訊息
> - 不知道你的錯誤畫面面

> - 沒有你的原始檔案
>
> ## 發問格式(範例)
>
> - 我目前在「實作檔案上傳」。我在「傳檔時」遇到錯誤。
> - 我用用的是 「Ubuntu 12.04」 , ruby 是用用 「rvm 上的的 2.0.0」
> - (貼 console 「全部訊息到 gist 」)
> - (貼瀏覽器的 「全部錯誤訊息到 gist 」)
> - (截瀏覽器整張圖張貼)
> - 把 Repo 網址公開(例如專案的 github),讓人人直接看
> - 少一樣都不行！

請不要直接把錯誤訊息貼在 Facebook 上， 因為 FB 的排版格式用在貼 Code 是場悲劇

除非經過當事人同意， 也請不要直接用 FB 的私訊問問題 因為大部份的人都是把 FB 用於私人時間上 一二次還好，太多次是會讓當事人想把你封鎖的

請保持應有的禮貌 沒有人有義務幫你解決問題

只要保持以上原則，相信您一定能得到別人善意的幫助！

[延伸閱讀] CRUD

原文網址

What is CRUD?

CRUD 是一個縮寫，由

Create Read Update Delete

四個單字組成，剛好也是一個網站運作最最最基本的功能

創建 讀取 更新 刪除

CRUD 功能的開發 in Rails

在 Rails 的架構中，有一套規範強制要求開發者 一定要照它的規則來開發 CRUD 功能

不同於其他程式語言做出來的網站，有很大的自由度隨你設定

這套規則就是 **MVC** 架構 與 **RESTful** 在初期要理解這套規則對新手來說無異是非常大的障礙

我們未來會開新篇幅解釋何謂 MVC架構 與 RESTful 對於身為新手的您只要知道一件事

先實做出來就對了

未來當您開發到一定的階段，就會理解到何為『 制約即解放 』

由於 CRUD 功能是一個網站最基本的規劃，太過自由的設定(例如命名、變數、流程) 會造成未來維護與再開發時一個非常大的困擾

寫 **code** 只需要一次， but 卻會花很多次來讀 **code**

如何把 code 寫的簡潔、乾淨、易讀、好維護是門很深的學問 本 blog 的教程也會儘量以上述的原則來寫出 Code

而 MVC 架構 跟 RESTful 則可以幫助我們在建置 CRUD 功能時， 做出一套簡潔、乾淨、易讀、好維護的程式碼

本章將開始 Rails 101 的實戰

架構出基礎 **CRUD** 功能 (基礎建置)

原文網址

本章給想理解如何從 0 寫出跟 Scaffold 一模一樣功能的人 如果您有心想把 Rails 學好，這章的內容務必要熟練， 甚至 Controller 的部分寫十遍把它背起來

本章的作業目標：

- 建立一個討論版(group)，但是不用 Scaffold 來做
- group 有 title 跟 description 二個欄位

建置一個全新的專案

`rails new group101` 建立一個全新的 Rails 專案，叫做 group101

`cd group101` 進入 group101 資料夾

`git init` 建立 git 版本控制

`git add .` 把現有檔案納入版本控制

`git commit -m "Initial Commit"` 做初次存檔

建立 **controller groups**

`rails g controller groups` 建立一個 Controller: groups (要加s)

`rails g model group title:string description:text` 建立一個 Model: group (不加s) 並順便建立資料表 group 的二個欄位: title (string 字串屬性) 跟 description (text 文字屬性)

`rake db:migrate` 將資料庫建立起來

設定 **routes**

原本

config/routes.rb

```
Rails.application.routes.draw do
...
...
```

改成

config/routes.rb

```
Rails.application.routes.draw do

resources :groups

root 'groups#index' #這行代表把 localhost:3000/groups 這個網址設成首頁
```

設定 **Controller groups**

原本

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController
end
```

插入

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController
  def index
  end
end
```

設定 **app/views/groups/index.html.erb**

到 app/views/groups 這個資料夾 建立一個叫做 index.html.erb 的檔案

原本

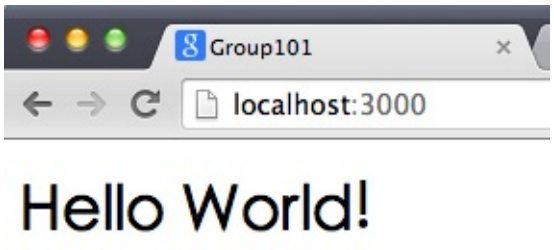
```
<!-- (空的) -->
```

插入

```
<h1>Hello World!</h1>
```

看看成果

`rails s` 打開 Server 模擬模式 打開瀏覽器，網址: localhost:3000



初步頁面完成，我們將一步步地把 CRUD 功能建置起來 結束前別忘了

`git add .`

`git commit -m "generate group"`

git 存檔

套入 **Bootstrap** 與版型設定

原文網址

本章的作業目標：

- 套入前端套件 Bootstrap
- 建立 navbar 與 footer
- 建立 notice_message

安裝 **gem 'bootstrap-sass'**

原本

gemfile

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.1.1'
...
...
```

插入

gemfile

```
source 'https://rubygems.org'

gem 'bootstrap-sass'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.1.1'
...
...
```

`bundle install` 安裝 gem

將 **Bootstrap** 的 **CSS** 套件裝進專案裡面

原本

app/assets/stylesheets/application.css

```
/*
 * ...(一堆註解)
 *= require_tree .
 *= require_self
 */
```

插入

app/assets/stylesheets/application.css

```
/*
 * ...(一堆註解)
 *= require_tree .
 *= require_self
 */

 @import "bootstrap";
```

記得重開 `rails s`

幫你的專案裝上 **Navbar** 跟 **Footbar**

到 app/views 增加一個資料夾 : common

增加二個檔案

app/views/common/_navbar.html.erb

```
<nav class="navbar navbar-default" role="navigation">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <a class="navbar-brand" href="/">Rails 101</a>
    </div>

    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
      <ul class="nav navbar-nav navbar-right">
        <li> <%= link_to("登入", '#') %> </li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
```

app/views/common/_footer.html.erb

```
<footer class="container" style="margin-top: 100px;">
  <p class="text-center">Copyright ©2014 Rails101s
    <br>Design by <a href="http://sdlong.logdown.com" target=_new>sdlong</a>
  </p>
</footer>
```

修改你的 **app/views/layout/application.html.erb**

原本

app/views/layouts/application.html.erb

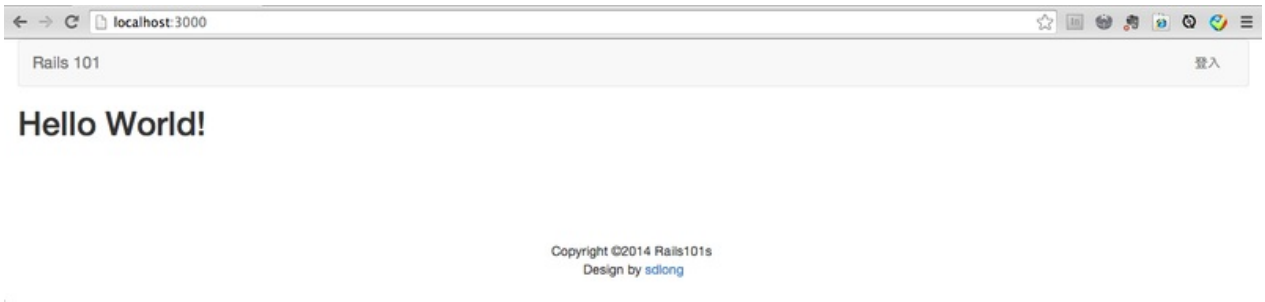
```
...
...
<body>

<%= yield %>

</body>
</html>
```

改成

app/views/layouts/application.html.erb

```
...
...
<body>

<div class="container-fluid">
  <%= render "common/navbar" %>
  <%= yield %>
</div>
  <%= render "common/footer" %>

</body>
</html>
```

打開瀏覽器看看



以上的步驟其實跟 1-1 一樣，接下來我們再多做二個功能

套入 **Bootstrap** 的 **javascript** 套件

原本

app/assets/javascripts/application.js

```
...
...
//= require turbolinks
//= require_tree .
```

改成

app/assets/javascripts/application.js

```
...
...
//= require turbolinks
//= require bootstrap/dropdown
//= require bootstrap/alert
//= require_tree .
```

由於套入太多 js 效果會影響網站的效能，所以先套入後面會用到的二個功能就好 對 Bootstrap 內建的 JS 效果有興趣的話可參考 http://getbootstrap.com/javascript/

安裝 notice_messsage

原本

app/helpers/application_helper.rb

```
module ApplicationHelper
end
```

改成

app/helpers/application_helper.rb

```
module ApplicationHelper

  def notice_message
    alert_types = { :notice => :success, :alert => :danger }

    close_button_options = { :class => "close", "data-dismiss" => "alert", "aria-hidden" => true }
    close_button = content_tag(:button, "×", close_button_options)

    alerts = flash.map do |type, message|
      alert_content = close_button + message

      alert_type = alert_types[type.to_sym] || type
      alert_class = "alert alert-#{alert_type} alert-dismissable"

      content_tag(:div, alert_content, :class => alert_class)
    end

    alerts.join("\n").html_safe
  end
end
```

在 layout 放入 notice_message

原本

app/views/layouts/application.html.erb

```
...
...
<div class="container-fluid">
  <%= render "common/navbar" %>
  <%= yield %>
</div>
...
...
```

插入

app/views/layouts/application.html.erb

```
...
...
<div class="container-fluid">
  <%= render "common/navbar" %>
  <%= notice_message  %>
  <%= yield %>
</div>
...
...
```

How to use notice_message

測試: :notice

原本

app/controllers/groups_controller.rb

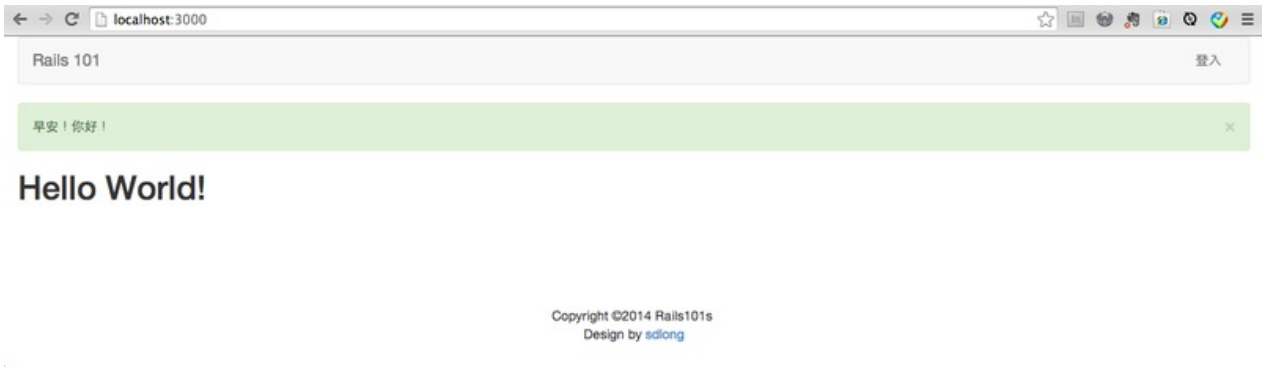
```
class GroupsController < ApplicationController

  def index
  end
end
```

改成

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  def index
    flash[:notice] = "早安！你好！"
  end
end
```

結果



測試 :alert

原本

app/controllers/groups_controller.rb

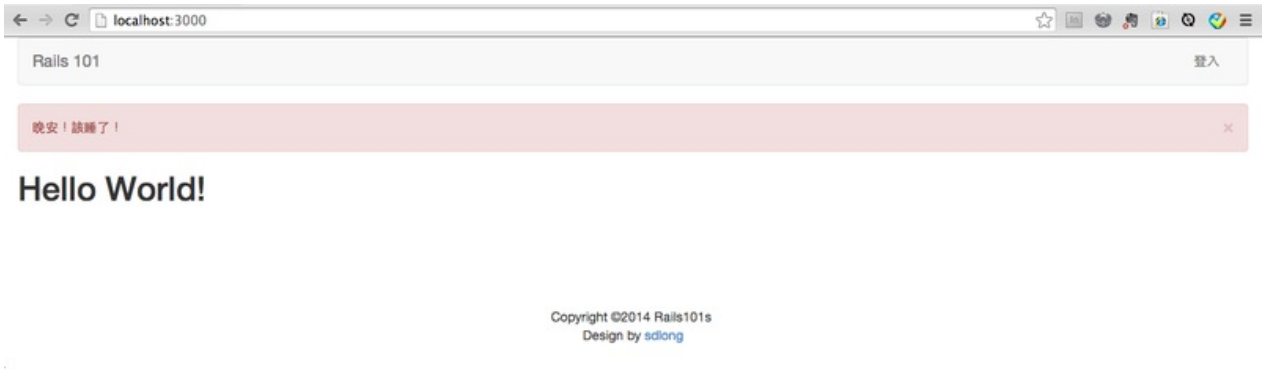
```
class GroupsController < ApplicationController

  def index
  end
end
```

改成

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  def index
    flash[:alert] = "晚安！該睡了！"
  end
end
```

結果



我們可以使用 這個叫做 notice_message 的"輔助函式" (在 Rails 叫做 『 helper 』) 來幫助我們做出訊息提醒/警告的功能 => 例如當討論版創建完成後顯示 創建成功

架構出基礎 **CRUD** 功能 (實做)

原文網址

本章的作業目標：

- 安裝 gem "simple_form"
- 不用 Scaffold 實作出有 CRUD 功能的 group
- 連前端網頁部分也做完
- 資料驗證 => title 欄位不能空白

安裝 **gem "simple_form"**

插入

gemfile

```
gem "simple_form"
```

執行 `bundle install`

前置設定: **controller**

打開 **groups_controller**

原本

```
class GroupsController < ApplicationController

  def index
  end
end
```

改成

```
class GroupsController < ApplicationController

  def index
  end

  def show
  end

  def new
  end

  def edit
  end

  def create
  end

  def update
  end

  def destroy
  end
end
```

前置設定: Views

到 app/views/groups/ 資料夾裡面

建立以下三個空白的檔案:

show.html.erb

new.html.erb

edit.html.erb

我們在上一章有做一個 index.html.erb

所以這個資料夾應該有四個檔案

建立 Index action

Controller

原本

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

def index
end
...
...
```

改成

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

def index
  @groups = Group.all
end
...
...
```

view index

原本

app/views/groups/index.html.erb

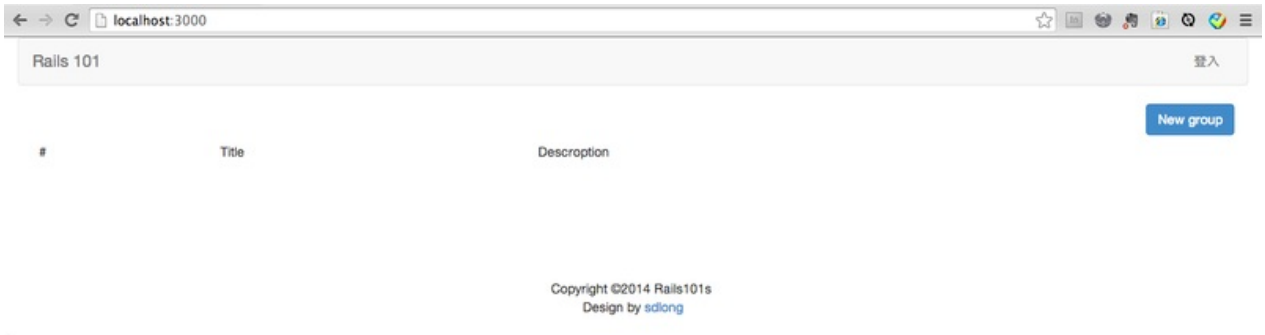
```
<h1>Hello World!</h1>
```

改成

app/views/groups/index.html.erb

```
<div class="col-md-12">
  <div class="group">
    <%= link_to("New group", new_group_path , :class => "btn btn-primary pull-right" ) %>
  </div>

  <table class="table table-hover">
    <thead>
      <tr>
        <td> # </td>
        <td> Title </td>
        <td> Description </td>
      </tr>
    </thead>

    <tbody>
          <% @groups.each do |group| %>
    <tr>
      <td> # </td>
      <td>  <%= link_to(group.title, group_path(group)) %> </td>
      <td>  <%= group.description %> </td>
      <td>  <%= link_to("Edit",
                  edit_group_path(group),
                  :class => "btn btn-sm btn-default")%>

          <%= link_to("Delete",
                  group_path(group),
                  :class => "btn btn-sm btn-default",
                  :method=>:delete,
                  data: { confirm: "Are you sure?" } )%></td>
    </tr>
          <% end %>
    </tbody>
  </table>
</div>
```

結果:



我們會發現按了 **"New group"** 只有跑出空白頁面，所以我們要繼續做下去

建立 New action

Controller

原本

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

...
...
def new
end
...
...
```

改成

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

...
...
def new
  @group = Group.new
end
...
...
```

view new

原本

app/views/groups/new.html.erb

```
(無)
```

改成

app/views/groups/new.html.erb

```
<div class="col-md-4 col-md-offset-4">
<h1>新增討論版</h1>

<hr>
<%= simple_form_for @group do |f| %>
  <%= f.input :title, :input_html => { :class => "form-control"} %>
  <%= f.input :description, :input_html => { :class => "form-control"} %>
<hr>
  <%= f.submit "Submit", :disable_with => 'Submitting...', :class => "btn btn-primary"%>
<% end %>
</div>
...
...
```

建立 create action

將填好的 New 表單裡的資料送到資料庫裡建立 (create)

controller

原本

app/controllers/groups_controller.rb

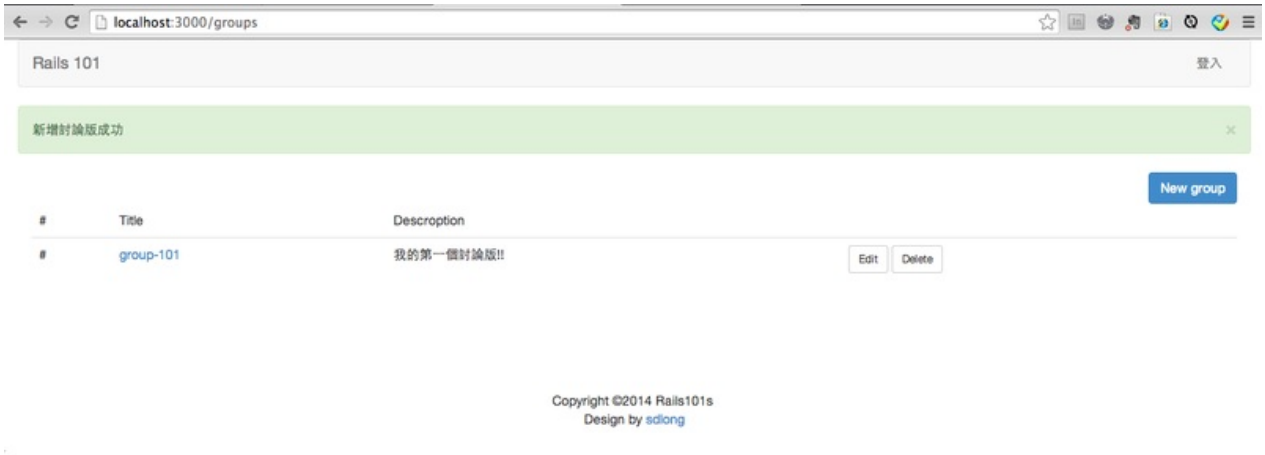
```
...
...
  def create
  end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def create
    @group = Group.new(group_params)

    if @group.save
      redirect_to groups_path, :notice => '新增討論版成功'
    else
      render :new
    end
  end
...
...
#(放在最下面)
  private

  def group_params
      params.require(:group).permit(:title, :description)
  end
```

接下來我們可以測試新增討論版的效果了



Show action

接下來要能觀看單一討論版頁面

controller

原本

app/controllers/groups_controller.rb

```
...
...
def show
end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
def show
  @group = Group.find(params[:id])
end
...
...
```

views show

原本

app/views/groups/show.html.erb

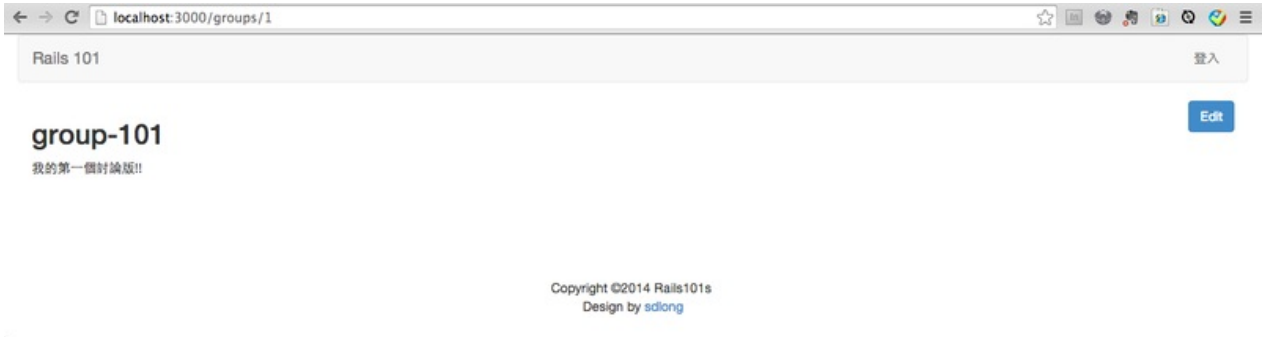
```
(空的)
```

改成

app/views/groups/show.html.erb

```
<div class="col-md-12">
  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right")%>
  </div>
  <h2> <%= @group.title %> </h2>
  <p> <%= @group.description %> </p>
</div>
```

點進第一個討論版連結後的結果



Edit Action

接下來要能夠修改討論版內容

controller

原本

app/controllers/groups_controller.rb

```
...
...
  def edit
  end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def edit
    @group = Group.find(params[:id])
  end
...
...
```

views edit

原本

app/views/groups/show.html.erb

```
(空的)
```

改為 (其實跟 new.html.erb 一模一樣，只有標題修改)

app/views/groups/show.html.erb

```
<div class="col-md-4 col-md-offset-4">
<h1>修改討論版</h1>

<hr>
<%= simple_form_for @group do |f| %>
  <%= f.input :title, :input_html => { :class => "form-control"} %>
  <%= f.input :description, :input_html => { :class => "form-control"} %>
<hr>
  <%= f.submit "Submit", :disable_with => 'Submitting...', :class => "btn btn-primary"%>
<% end %>
</div>
```

update action

要將填好的 edit 表單資料更新 (update) 到資料庫

原本

app/controllers/groups_controller.rb

```
...
...
  def update
  end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def update
    @group = Group.find(params[:id])

    if @group.update(group_params)
      redirect_to groups_path, :notice => '修改討論版成功'
    else
      render :edit
    end
  end
...
...
```

來看看實際運作畫面

destroy action

最後我們要能把討論版刪除掉

原本

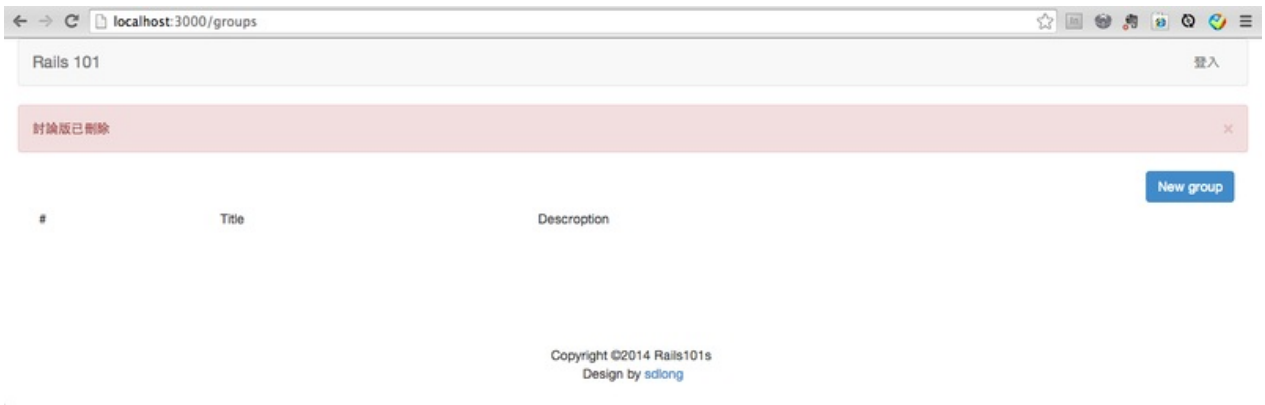
app/controllers/groups_controller.rb

```
...
...
  def destroy
  end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def destroy
    @group = Group.find(params[:id])

    @group.destroy
    redirect_to groups_path, :alert => '討論版已刪除'
  end
...
...
```



如果整個 CRUD 操作有問題，可能是你的 groups_controller.rb 的設定有誤

完整的:

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  def index
    @groups = Group.all
  end

  def new
    @group = Group.new
  end


  def show
    @group = Group.find(params[:id])
  end


  def edit
    @group = Group.find(params[:id])
  end

  def create
    @group = Group.new(group_params)

    if @group.save
      redirect_to groups_path, :notice => '新增討論版成功'
    else
      render :new
    end
  end

  def update
    @group = Group.find(params[:id])

    if @group.update(group_params)
      redirect_to groups_path, :notice => '修改討論版成功'
    else
      render :edit
    end
  end

  def destroy
    @group = Group.find(params[:id])

    @group.destroy
    redirect_to groups_path, :alert => '討論版已刪除'
  end

  private

  def group_params
      params.require(:group).permit(:title, :description)
  end
end
```

資料驗證: **title** 不能空白

當我們要創建討論版的時候，如果版名是空白的不是很怪嗎？

model 可以做出所有跟資料庫相關的設定

原本

app/models/group.rb

```
class Group < ActiveRecord::Base
end
```

插入

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true
end
```

這行就能設定 title 不能空白

最後別忘了 git 存檔

`git add .`

`git commit -m "完成 groups 的 CRUD 設定"`

[延伸閱讀] MVC 架構與 RESTful 概念

原文網址

本章將淺談：

HTTP Request 與 HTTP Verb

MVC架構 => Rails的基本運作原理

RESTful淺談

模擬要建立一個有 CRUD (Create Read Update Destroy)功能的討論版 在MVC架構下要做什麼東西出來

HTTP Request(請求) 與 HTTP Verb (HTTP動詞)

是瀏覽器端跟網站的 server 之前存取資料的動作，每當我們輸入網址，點擊某個連結，都是代表 我們從瀏覽器端發出請求(request)指令到網站 server 端，後者再以 request 的內容來回應資料給前者 也就是網頁資料的呈現

request只有四個動詞(Verb)種類：

GET	POST	PUT/PATCH	DELETE
讀取	新增	更新	刪除
Read	Create	Update	Destroy

絕大多數都是GET，也就是跟Server端請求讀取某個網址的資料

有要更動到 server 裡資料庫的資料，就會用到另外的三個動作 當我們要用 Rails 做一個有 CRUD功能(**Create** 、 **Read** 、 **Update** 、 **Destroy**)的討論版 正好就對應到 HTTP Verb 上的四個動詞

MVC架構

『網頁』跟『網站』製作最大的差別，在於前者只有 純讀取 資訊的功能 後者可以運作 CRUD 功能，讓系統功能強大 但從前者走向後者，整個系統的規劃與運作就會變得非常龐大與複雜

為了方便可以區分功能與團隊分工合作將一個網站專案建立起來 近代的網站開發走向了 MVC 架構：

	Models	Views	Controllers
功能	資料庫相關	前端網頁呈現	流程控制與運作

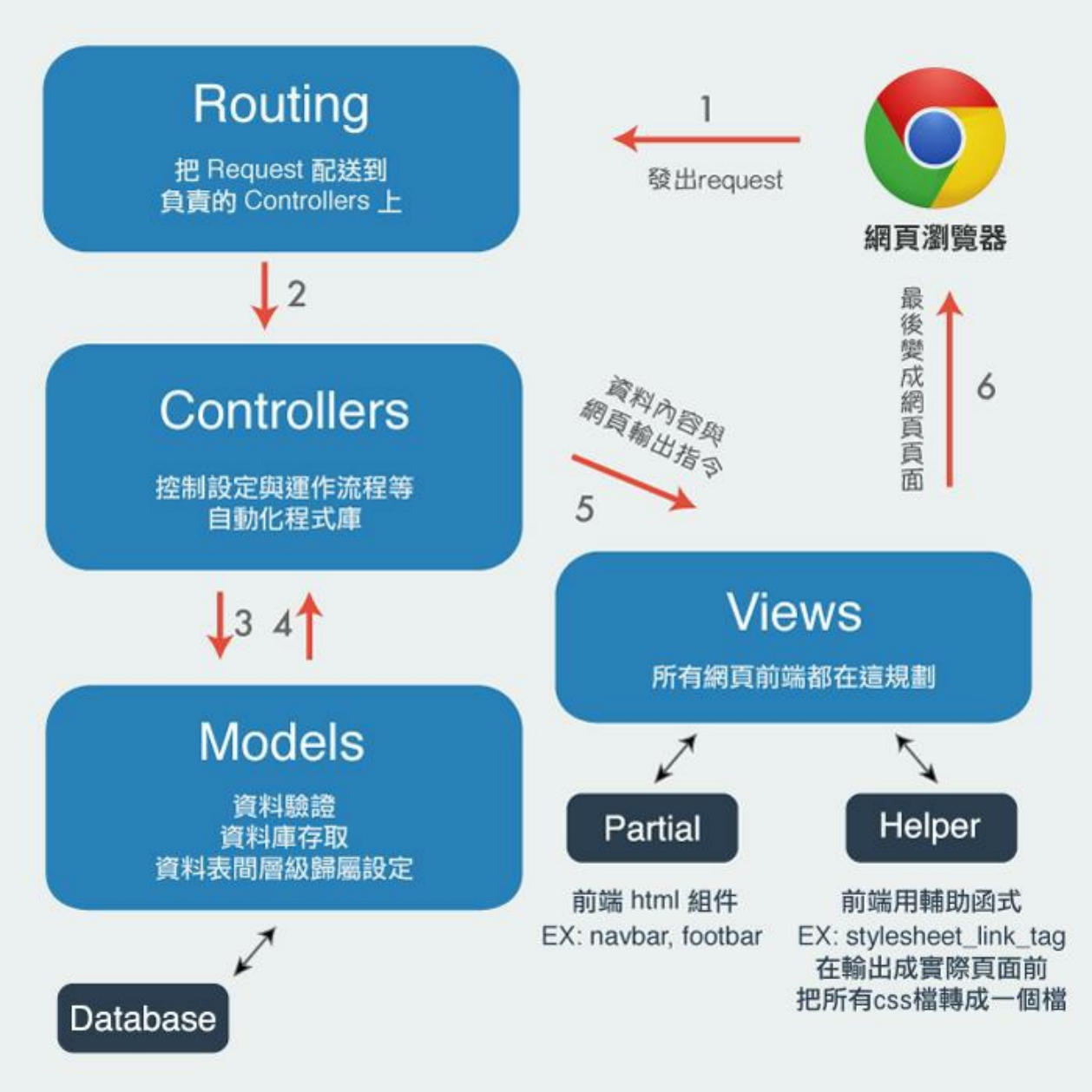
優點是什麼？

可以把整個專案(Project)拆分 負責做前端的人只做 Views 寫流程控制的專心寫 Controllers 所有資料相關集中在 Models

三者之間不互相打架，出問題與狀況也容易追出問題的位置

> Ruby on Rails 就是用 **Ruby** 這個程式語言寫出來的網站 MVC 框架(Framework)

Rails 的運作原理與流程



這張圖可以清楚看出Rails是怎麼跟使用者的瀏覽器互動來運作的

如果我們拿 1-0 所做的 Hello World! 來解釋：

當我們輸入 http://localhost:3000/welcome 這個指令，即是 瀏覽器對**server**發出讀取**(GET)**首頁資料的請求**(request)**

config/routes.rb 裡已經設定好這個Request的工作

config/routes.rb

```
get 'welcome', to: 'topics#welcome'
```

Routing(路由)會把這個Request送到

app/controllers/topics_controller.rb

並執行裡面的 welcome 這個動作 (action)

app/controllers/topics_controller.rb

```
def welcome
end
```

welcome action裡目前沒有存取資料庫(Models部分)的動作，所以會跳至Views

把 app/views/layouts/application.html.erb 裡網站主的樣板抓出來 再把
app/views/topics/welcome.html.erb 裡的資料抓出來，合併在前者的 <%= yield %> 區域裡面

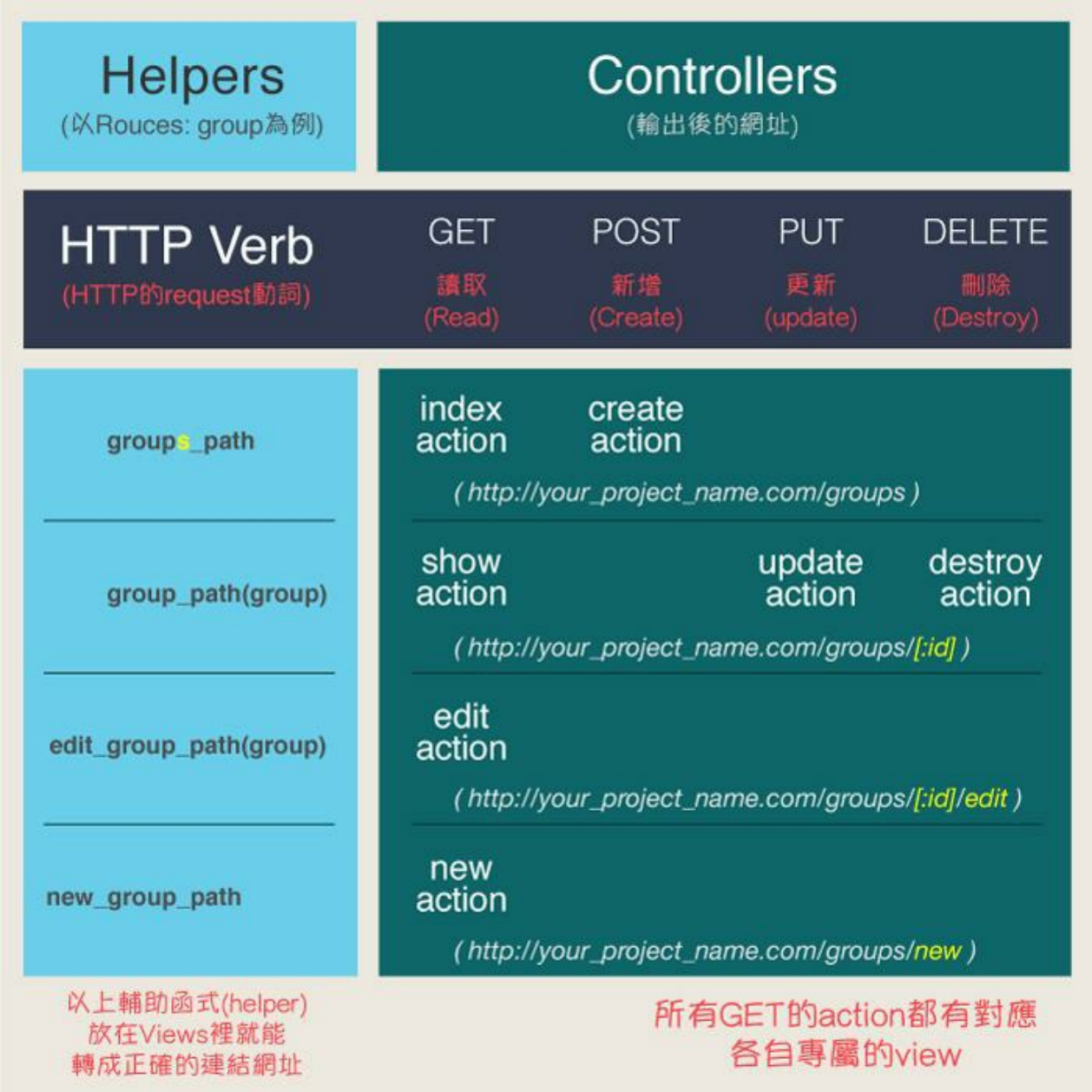
專案開發與資訊安全問題

我們可以任意設定 Action 的命名，只要 Request 正確，就有相對應的動作 但太過自由的設計，會導致專案開發的災難

1. 若無對 Request 動作分類做統一的規範，一樣的 index action，可能會有不一樣的內容 導致團隊夥伴在開發時必須花時間閱讀程式碼的內容，也許這個 action 僅需花 1 分鐘 可是 10 個夥伴，就會花費整個團隊 10 分鐘的時間在這件事上面
2. 若夥伴或你本人剛做好一個新的 action ，卻發現這個功能之前就做過一個類似的... 未來若又要沿用這個 action 做延伸功能，要嘛再重讀一次之前的程式碼，不然又要再重做一個新的 action ...
3. 資訊安全問題，在新增，更新，刪除某筆資料時，必須要有驗證使用者權限的機制 延續 2nd ， 要怎麼知道之前的夥伴是否有做？(花時間重讀) 還是乾脆做一個新的(重複動作) 又要怎麼讓資訊安全機制不會變成龐大工程，讓團隊精力專心在專案開發上？

最佳解決方案： **RESTful**

Rails的基礎 RESTful概念



由於RESTful是個很複雜的概念，號稱學 Ruby on Rails 初期最大障礙 本篇儘量用淺顯易懂的方式解釋，如果想了解更多的請到後面的延伸閱讀

以下例子以 2 - 2 , 做出 CRUD 功能的討論版來解釋：

討論版這個功能的名稱： groups 為了要控制它的運作，我們會創建一個新的 Controller檔案 叫做 groups_controller.rb

它是一個用Ruby寫的程式，將好幾個方法(Method, 或稱函式)組成的類別(Class) 以上概念不懂的請回頭看小弟以前寫的 Ruby 學習心得：1 , 2 , 3

每個 Method 剛好就是本篇文章一開始所說的動作 (action)

HTTP Verb 有四個，除了讀取 (GET) 以外，另外三個都有相對應的動作

1. 新增 (create)
2. 更新 (update)
3. 刪除 (destroy)

在這個功能裡面，我們會做出四個頁面出來：

1. 首頁(index)：用來列出所有的討論版，可以選擇各個單版
2. 各個討論版專屬頁面(show)：顯示討論版版名跟簡介
3. 修改頁面(Edit)：裡面會有表單呈現現有資料來，填完資料後可以送出
4. 新增頁面(New)：裡面會有表單，填完以後可以送出

這七個 action，是每個CRUD功能裡最基本的動作，缺一不可 只要我們在 controller.rb 裡定義 (Define) 好這七個方法 (Method) 當作動作 (action)

其他的部分 Rails 就會幫我們處理了 像是設定 Routing (路由) 只需要簡單一行:

config/routes.rb

```
resources :groups
```

這樣不管瀏覽器端跑來什麼樣的請求 (request)，Routing都會聰明地幫我們導向正確的 action 去運作後續流程

這樣也解決了資訊安全問題，因為只有 **Create Update destroy** 這三個action才會動到server端的資料異動

對於網址跟對應的Action與頁面來說

/groups/ 一定就是 **index**，首頁，對應Routes路徑(或稱為helper)： groups_path /groups/123 一定就是 **show**，名字為123的討論版個版頁面，對應Routes路徑(或稱為helper)： group_path(123) /groups/123/edit 一定就是 **edit**，修改討論版123的表單頁面，對應Routes路徑(或稱為helper)：edit_group_path(123) /groups/new 一定就是 **new**，新增討論版的表單頁面，對應Routes路徑(或稱為helper)：new_group_path

總結

所以製作一個RESTful風格的討論版，我們僅需在Routes設定一行程式即可

config/routes.rb

```
FirstApp::Application.routes.draw do
  resources :groups
end
```

MVC架構中:

Models:

app/models/group.rb

```
class Group < ActiveRecord::Base
end
```

Controllers

在 app/controllers 僅需定義 (Define) 七個方法 (Method) 當動作 (action)

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  def index
  end

  def show
  end

  def new
  end

  def edit
  end

  def create
  end

  def update
  end

  def destroy
  end
end
```

Views

在app/views/groups/ 資料夾裡做四個html檔案：

index.html.erb

show.html.erb

edit.html.erb

new.html.erb

有沒有發現，Views 裡面的前端 html 檔案剛好都對應到HTTP Verb是GET的action? 當我們運作 Create、Update、Destroy等動作時，Rails不會笨笨的指向開啟"create這個網頁"的動作

這樣我們就用MVC架構把 group 功能完整個拆分出來了

- Models來做資料庫設定
- Controllers做流程運作控制

- Views做前端設計

好處是藉此定下規範： Views 裡不會出現流程控制 資料存取出狀況找 Models 的部分就對了 Controller 裡面一定不會有html/css/js碼

延伸閱讀:

RESTful應用程式 by iHower

RESTful教學1 by XDite

RESTful教學2 by XDite

RESTful教學3 by XDite

[延伸閱讀] 深度解析 CRUD 功能裡的 Controller & Model

原文網址

本章淺談：

暫時不去看 view 的部分

好好觀察在 CRUD 的機制裡 Controller 是怎麼運作的

又是怎麼跟 Model 互動

Model 又是用什麼方式來驗證資料跟存取資料庫

可以另外再開一個分頁 打開 2 - 2. 架構出基礎 CRUD 功能(實做) 最下面 app/controllers/groups_controller.eb 的內容，用來對應下面要描述的東西

基本概念

以 2 - 2 的案例來示範，我們已經設定好 Group 這個class(在 app/models/group.rb) 並且設定好它的資料庫格式： Title欄位(屬性: 字串string) 跟 Description欄位(屬性：多行文字text)

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true
end
```

雖然整個檔案只有短短三行 code 但有多種功能：

1. `< ActiveRecord::Base` 代表這個 class 繼承了 Rails 裡內建的函式庫： ActiveRecord::Base 裡的功能
2. ActiveRecord 是 Rails 的 ORM(Object Relational Mapping) 元件，負責與資料庫溝通，讓我們能用 Ruby 語法操作資料庫
3. 以本例來說，我們就能在 Controller 或是 Rails Console (主控台)用 Group 來呼叫對應在資料庫裡的『Group』這個資料表，並操作 CRUD 功能
4. 更能在 app/models/group.rb 裡面做出許多跟資料庫相關的設定，例如第二行就是設定 title 這個欄位必填(不能空白)

延伸閱讀

建立一個物件，一個能寫進資料庫: group 裡變成一筆新資料的物件

```
@group = Group.new
```

代表我們將 @group 這個實例變數設定成一個新物件，格式與規則等同 app/models/group.rb 的設定 (可以看看這段程式碼出現在哪個 action)

這個物件的格式就是

```
@group = {  :id => "",
          :title => "",
          :description => ""}
```

有欄位設定，因為是新建立的，無內容

以下是資料庫裡 groups 的資料，我們會用這個資料表做範例：

groups

id	title	description
1	Ariel	Ariel的秘密花園
2	Bill	Bill之家
3	Carl	Carl的工作室
4	Della	Della的小窩

我們可以使用

```
@group = Group.find(3)
```

讀取(read)出 id 是 3 的這筆資料 (查看看這段函式是不是跟某幾個Action很像？)

這個物件格式內的資料就是

```
@group = {  :id => "3",
          :title => "Carl",
          :description => "Carl的工作室"}
```

{ } 裡的資料，就是所謂的雜湊(Hash)，一個陣列化的變數 以此為例，如果這個Hash的格式與資料能通過 app/model/group.rb 裡的驗證，格式與資料符合，就能允許將它儲存起來

這時候如果輸入

```
puts @group[:title]
=> "Carl"

puts @group[:description]
=> "Carl的工作室"
```

CRUD 運作解密 (Controller)

Create

```
@group = Group.new
@group.title = "Esse"
@group.description = "Esse的衣帽間"
@group.save
```

Read

```
@group = Group.find(2)
```

Update

```
@group = Group.find(2)
@group.description = "Bill的秘密基地"
@group.save
```

Delete

```
@group = Group.find(2)
@group.destroy
```

以上就是 CRUD 的運作邏輯 我們要把它運用進網站的運作，所以實際的運作會是這樣：

Create運作

(點擊建立新group按鈕) => 觸發new action(@group = Group.new) => 開啓new.html這個頁面

在new.html裡把資料寫好以後 例如 標題: Esse , 敘述:Esse的衣帽間 => 點擊送出(submit)按鈕 => 觸發 create action

create action的內容是

```
@group.new(:title => "Esse", :description => "Esse的衣帽間")
@group.save
```

Read運作

app/controllers/groups_controller.eb

```
@group = Group.find(params[:id])
```

裡面的 params[:id] 是參數設定，可以讓我們在輸入 HTTP request(例如網址: group/2)時 轉成 @group = Group.find(2)

除了.find外，我們還可以用其他指令做運用

指令	用法
Group.first	叫出第一筆資料
Group.last	叫出最後一筆資料
Group.all	叫出全部資料(請看看index action)
Group.count	叫出資料表內有多少資料筆數
Group.limit(10)	只叫出十筆資料

延伸閱讀：更多用法

在網站運作裡，會觸發 show action 轉成討論版個版的頁面

Update運作

點擊個版頁面裡的 Edit 按鈕 => 觸發 edit action => 進入edit.html

輸入修改後的資料 => 點送出按鈕 => 觸發 update action

Delete運作

bj4 ... XD

CRUD 運作解密 (Model)

bj4 ... XD

其他 Model 資料驗證的語法：

參數	意思
:presence => true	此欄位必填
:numericality => true	此欄位只能填數字
:uniqueness => true	此欄位資料不能重複
:confirmation => true	需要在表單輸入二次，例如密碼確認
:acceptance => true	此欄位為Checkbox方塊，例如使用者條款確認
:length => { :minimum => 0, :maximum => 2000} :	限定欄位字元長度(本例是最短0，最長2000)
:format => { :with => /.*/ }	設定欄位資料的格式(例如網址、email)
:inclusion => { :in => [18..99]}	只能允許輸入 18 ~ 99的值
:exclusion => { :in => [0..18]}	禁止1 ~ 17之間的數字輸入 (例如18禁)

延伸閱讀：更多用法

我們在上一章做了一個建立 討論版 的功能 本章將做出在 討論版裡發表文章 的功能

可以在討論版裡發表文章 (前置設定)

原文網址

本章的作業目標：

- 可以在 Group 裡面 post 文章 (models, routes 設定)
- 並且文章網址是使用 /group/1/post/2 這種網址表示。

建立 **posts controller**

`rails g controller posts` 建立一個 Controller: posts (要加s)

`rails g model post content:text group_id:integer` 建立一個 Model: post (不加s) 並順便建立資料表 post 的二個欄位: content(text 文字屬性) 跟 group_id (integer 整數屬性)

`rake db:migrate` 將資料庫建立起來

建立 **group** 與 **post** 二資料表間的關聯性 **(relationship)**

routes 設定

原本

config/routes.rb

```
Rails.application.routes.draw do

  resources :groups

  root 'groups#index'
...
...
```

改成

config/routes.rb

```
Rails.application.routes.draw do

  resources :groups do
    resources :posts
  end

  root 'groups#index'
...
...
```

Models 設定

models/group.rb

原本

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true
end
```

改成

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true

  has_many :posts
end
```

models/post.rb

原本

app/models/group.rb

```
class Post < ActiveRecord::Base
end
```

改成

app/models/group.rb

```
class Post < ActiveRecord::Base
  belongs_to :group
end
```

解說

routes 設定完以後，即可輸入 /groups/1/posts/1 這樣的網址來指向想前往的討論版與文章

Models 裡面 group 跟 post 設定好關聯性後

即可在 rails console 模式做資料表操作來模擬

rails console/logs

```
# 叫出 Group 第一筆的資料
2.0.0-p353 :029 > a = Group.find(1)
  Group Load (0.4ms)  SELECT  "groups".* FROM "groups"  WHERE "groups"."id" = ? LIMIT 1  [["id", 1]]
 => #<Group id: 1, title: "rails101", description: "my first app", created_at: "2014-07-18 22:03:35", upd

# 叫出該筆資料裡擁有 (has_many) 的全部 post
2.0.0-p353 :030 > b = a.posts
  Post Load (0.3ms)  SELECT "posts".* FROM "posts"  WHERE "posts"."group_id" = ?  [["group_id", 1]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Post id: 1, content: "hello world", group_id: 1, crea

# 叫出某單筆 post 資料
2.0.0-p353 :031 > c = a.posts.find(1)
  Post Load (0.2ms)  SELECT  "posts".* FROM "posts"  WHERE "posts"."group_id" = ? AND "posts"."id" = 1
 => #<Post id: 1, content: "hello world", group_id: 1, created_at: "2014-07-18 22:06:23", updated_at: "2

# 該筆 post 資料屬於 (belongs_to) 哪個 group
2.0.0-p353 :032 > c.group
 => #<Group id: 1, title: "rails101", description: "my first app", created_at: "2014-07-18 22:03:35", upd
```

我們將用這樣的原理來設定 Controller

可以在討論版裡發表文章 (CRUD實作)

原文網址

本章的作業目標：

- 可以在 Group 裡面 post 文章 (controllers 與 views 設定)

要能在 **group** 的 **show** 頁面顯示所擁有的 **post**

controller

原本

app/controllers/groups_controller.rb

```
...
...
  def show
    @group = Group.find(params[:id])
  end
...
...
```

插入

app/controllers/groups_controller.rb

```
...
...
  def show
    @group = Group.find(params[:id])
    @posts = @group.posts
  end
...
...
```

views

原本

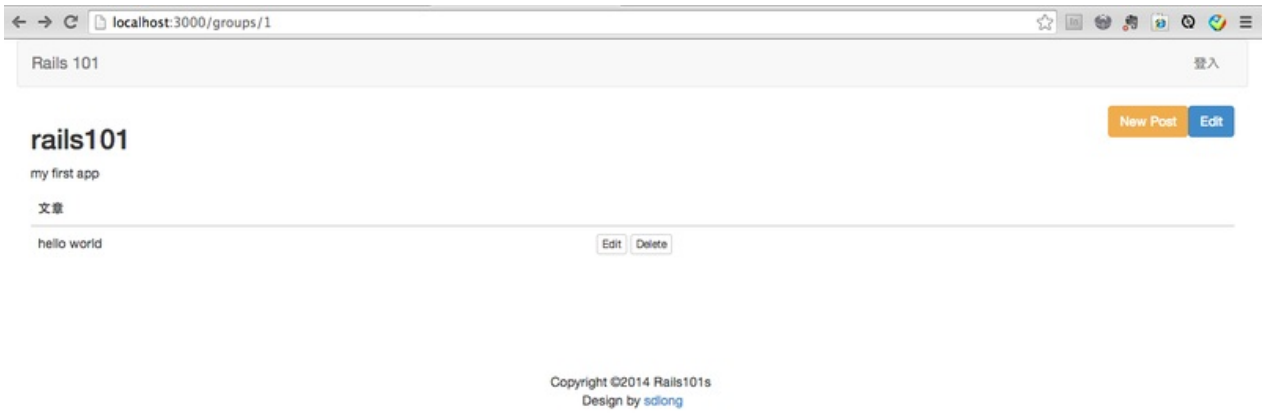
app/views/groups/show.html.erb

```
<div class="col-md-12">
  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right")%>
  </div>
  <h2> <%= @group.title %> </h2>
  <p> <%= @group.description %> </p>
</div>
```

改成

app/views/groups/show.html.erb

```
<div class="col-md-12">
  <div class="group">
    <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-warning pull-right")%>
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right")%>
  </div>
  <h2> <%= @group.title %> </h2>
  <p> <%= @group.description %> </p>

  <table class="table">
    <thead>
      <tr>
        <th>文章</th>
        <th colspan="2"></th>
      </tr>
    </thead>
    <tbody>
      <% @posts.each do |post| %>
        <tr>
          <td> <%= post.content %> </td>
          <td> <%= link_to("Edit",
                      edit_group_post_path(post.group, post),
                      :class => "btn btn-default btn-xs")%>

               <%= link_to("Delete",
                      group_post_path(post.group, post),
                      :class => "btn btn-default btn-xs ",
                      :method => :delete,
                      data: { confirm: "Are you sure?" } )%></td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
```



已上圖是『已經有一筆 post 資料』的模擬圖，我們需要做完後面的 controller 設定才能新增文章

---

post 的 CRUD 設定

前置

原本

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
end
```

改成

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  def new
  end

  def edit
  end

  def create
  end

  def update
  end

  def destroy
  end
end
```

app/views/posts/ 資料夾新增二個檔案:

app/views/posts/new.html.erb

```
(空的)
```

app/views/posts/edit.html.erb

```
(空的)
```

---

new action

controller

原本

app/controllers/posts_controller.rb

```
...
...
  def new
  end
...
...
```

改成

app/controllers/posts_controller.rb

```
...
...
  def new
    @group = Group.find(params[:group_id])
    @post = @group.posts.new
  end
 ...
 ...
```

view

原本

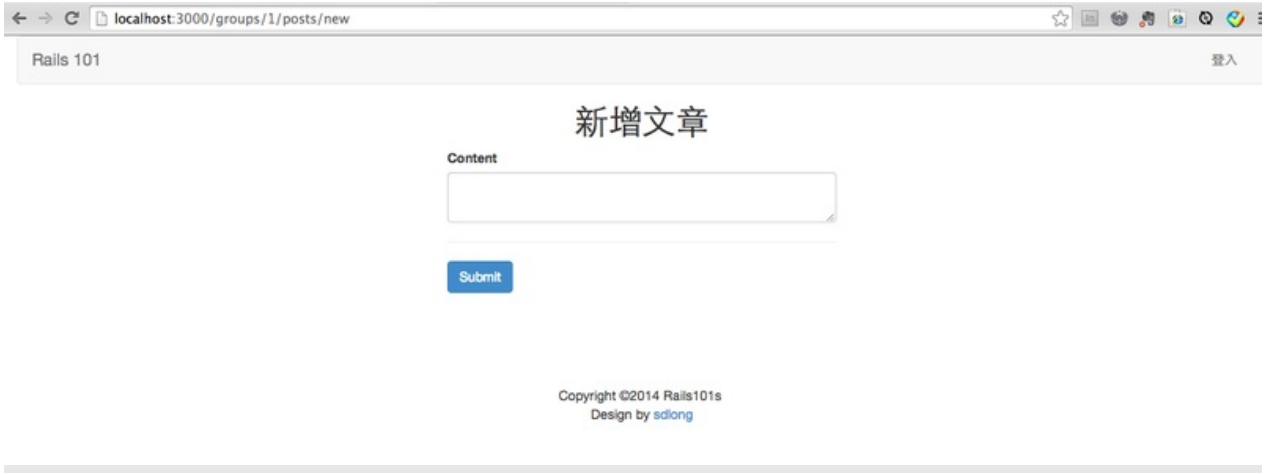
app/views/posts/new.html.erb

```
(空的)
```

改成

app/views/posts/new.html.erb

```
<h1 class="text-center">新增文章</h1>

<div class="col-md-4 col-md-offset-4">
  <%= simple_form_for [@group,@post] do |f| %>
    <%= f.input :content, :input_html => { :class => "form-control"} %>

    <hr>
    <div class="form-actions">
      <%= f.submit "Submit", :disable_with => 'Submiting...', :class => "btn btn-primary"%>
    </div>
  <% end %>
</div>
```



create action

原本

app/controllers/posts_controller.rb

```
...
...
  def create
  end
...
...
```

改成

app/controllers/posts_controller.rb

```
...
...
  def create
    @group = Group.find(params[:group_id])
    @post = @group.posts.new(post_params)

    if @post.save
      redirect_to group_path(@group), :notice => '新增文章成功！'
    else
      render :new
    end
  end
...
...
#(放到最下面)
  private

  def post_params
    params.require(:post).permit(:content)
  end
```

---

edit & update action

controller

原本

app/controllers/posts_controller.rb

```
...
  def edit
  end
...
...
  def update
  end
....
```

改成

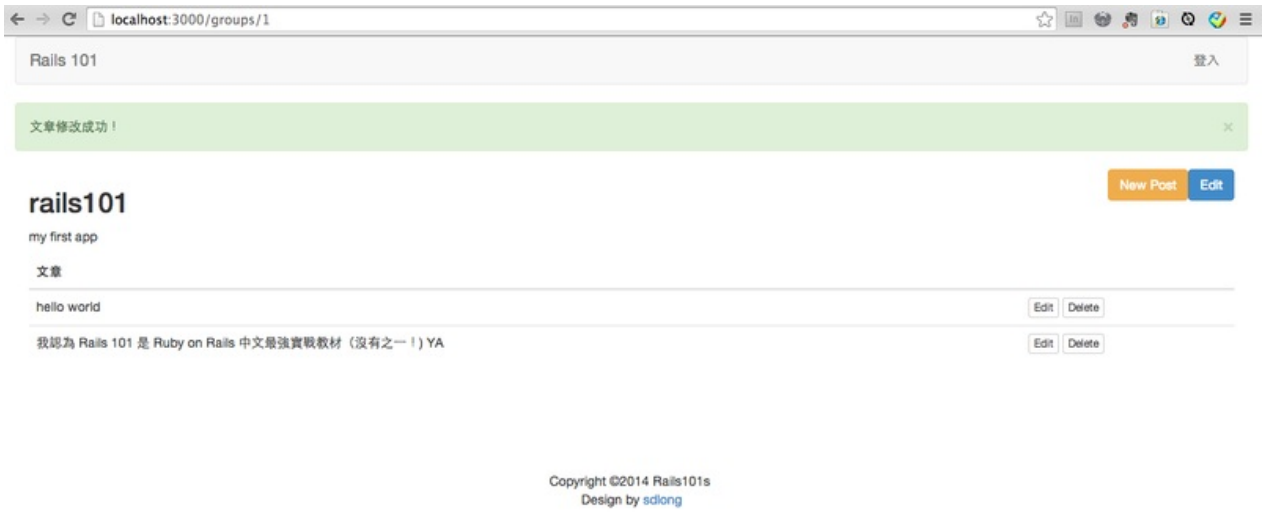
app/controllers/posts_controller.rb

```
...
...
  def edit
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])
  end
...
...
  def update
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    if @post.update(post_params)
      redirect_to group_path(@group), :notice => '文章修改成功！'
    else
      render :edit
    end
  end
...
...
```

view

app/views/posts/edit.html.erb

(跟 new.html.erb 一模一樣, 把『新增文章』改成『修改文章』即可)



destroy action

原本

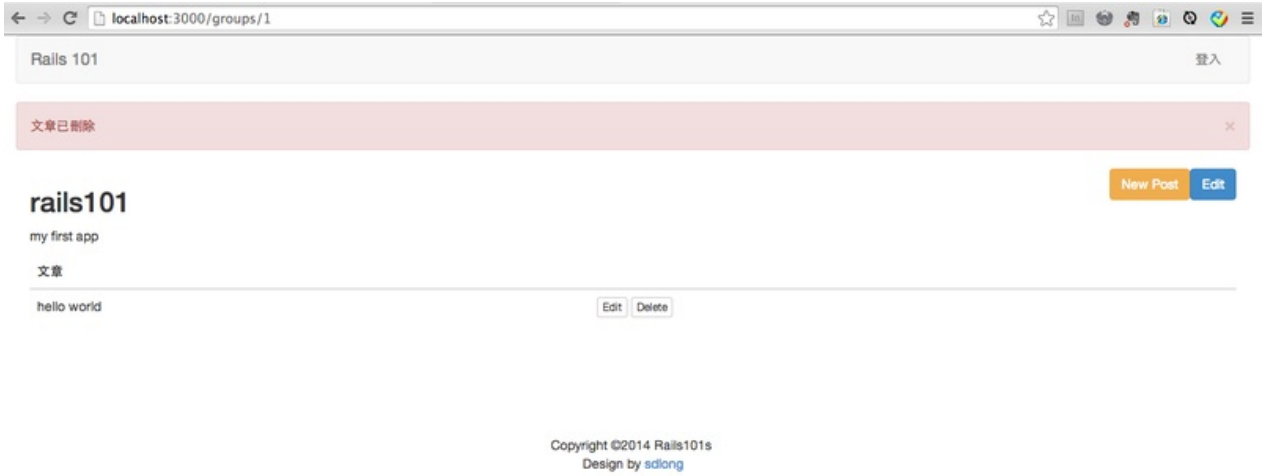
app/controllers/posts_controller.rb

```
...
...
  def destroy
  end
...
...
```

改成

app/controllers/posts_controller.rb

```
...
...
  def destroy
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    @post.destroy
    redirect_to group_path(@group), :alert => '文章已刪除'
  end
...
...
```



content 不能為空白

當然，我們也要設下資料驗證，文章內容不能空白

原本

app/models/post.rb

```
classPost<ActiveRecord::Base
  belongs_to :group
end
```

改成

app/models/post.rb

```
classPost<ActiveRecord::Base
  belongs_to :group
  validates :content, :presence => true
end
```

最後別忘了 **git commit**

`git add .`

`git commit -m "可以在討論版裡發表文章"`

附錄: 一個完整的 **app/controllers/posts_controller.rb** 的內容:

```
class PostsController < ApplicationController

  def new
    @group = Group.find(params[:group_id])
    @post = @group.posts.new
  end

  def edit
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])
  end

  def create
    @group = Group.find(params[:group_id])
    @post = @group.posts.new(post_params)

    if @post.save
      redirect_to group_path(@group), :notice => '新增文章成功！'
    else
      render :new
    end
  end

  def update
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    if @post.update(post_params)
      redirect_to group_path(@group), :notice => '文章修改成功！'
    else
      render :edit
    end
  end

  def destroy
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    @post.destroy
    redirect_to group_path(@group), :alert => '文章已刪除'
  end



  private

  def post_params
    params.require(:post).permit(:content)
  end

end
```

以 **before_action** 整理重複的程式碼

原文網址

在上一章我們做完了 app/controllers/posts_controller.rb

看到每個 Action 都有重複的程式碼，不會覺得很礙眼嗎 XD

我們可以使用一個小技巧來簡化 code

原始

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  def new
    @group = Group.find(params[:group_id])
    @post = @group.posts.new
  end

  def edit
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])
  end

  def create
    @group = Group.find(params[:group_id])
    @post = @group.posts.new(post_params)

    if @post.save
      redirect_to group_path(@group)
    else
      render :new
    end
  end

  def update
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    if @post.update(post_params)
      redirect_to group_path(@group), :notice => '文章修改成功！'
    else
      render :edit
    end
  end

  def destroy
    @group = Group.find(params[:group_id])
    @post = @group.posts.find(params[:id])

    @post.destroy

    redirect_to group_path(@group), :alert => '文章已刪除'
  end

  private

  def post_params
    params.require(:post).permit(:content)
  end

end
```

可以看到每個 action 前面都有一行 『 @group = Group.find(params[:group_id]) 』

所以能使用 before_action 來宣告此 Controller 每個 action 在執行之前都會先做 before_action 的設定

來把每個 action 的那行簡化掉

在最下面增加一段 code

變成

app/controllers/posts_controller.rb

```
...
...

  private

...
...

  def find_group
     @group = Group.find(params[:group_id])
  end
```

然後在最前面加一行 before_action :find_group

變成

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  before_action :find_group
...
...
```

最後再把每個 action 裡的 『 @group = Group.find(params[:group_id]) 』 刪掉

完整版的 code:

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  before_action :find_group

  def new
    @post = @group.posts.new
  end

  def edit
    @post = @group.posts.find(params[:id])
  end

  def create
    @post = @group.posts.new(post_params)

    if @post.save
      redirect_to group_path(@group)
    else
      render :new
    end
  end

  def update
    @post = @group.posts.find(params[:id])

    if @post.update(post_params)
      redirect_to group_path(@group), :notice => '文章修改成功！'
    else
      render :edit
    end
  end

  def destroy
    @post = @group.posts.find(params[:id])

    @post.destroy

    redirect_to group_path(@group), :alert => '文章已刪除'
  end


  private

  def post_params
    params.require(:post).permit(:content)
  end

  def find_group
      @group = Group.find(params[:group_id])
  end
end
```

結論

before_action 是一個常見的 controller 技巧,用來收納重複的程式碼。

before_action 可以用 only,指定某些 action 執行:

```
before_action: find_group, :only => [:edit, :update]
```

或者使用 except,排除某些 action 不執行:

```
before_action: find_group, :except => [:show, :index]
```

我們已經做好 基本版型 討論版 與發表文章功能

本章將著手建置使用者功能

並且做出作者機制 與 簡易後台功能

加入使用者功能

原文網址

本章的作業目標:

- 安裝 gem
- 設定 devise
- 在 navbar 安裝登入/登出按鈕
- 調整修改 註冊 / 登入頁面
- 利用 before_action :authenticate_user! 來做要求登入的設定
- new, create, edit, update, destroy 等 action 必須先登入才能操作

安裝 **gem**

gemfile 插入 『 gem 'devise' 』

gemfile

```
...
...
  gem 'devise'
...
...
```

`bundle install` 安裝新增的 gem

建置 **devise** 與設定

`rails g devise:install` devise 安裝

`rails g devise user` 建立 user 功能

`rake db:migrate` 由於需要一個資料庫來儲存使用者資料 devise 很聰明的幫我們把相關設定都做好了 所以只需要執行這行來建立 User 的資料庫

`rails g devise:views` 叫出(原本是隱藏的) devise views 未來可以客製化修改

別忘了重開 rails server

在 **navbar** 安裝登入**/**登出按鈕

接下來要能在 navbar 上面將 登入/登出 按鈕功能做出來

原本

app/views/common/_navbar.html.erb

```
...
...
      <li> <%= link_to("登入", "#") %> </li>
...
...
```

改成

app/views/common/_navbar.html.erb

```
...
...
      <% if !current_user %>
      <li> <%= link_to("註冊", new_user_registration_path) %> </li>
      <li> <%= link_to("登入", new_user_session_path) %> </li>
      <% else %>
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">
          Hi!, <%= current_user.email %>
          <b class="caret"></b>
        </a>
        <ul class="dropdown-menu">
          <li> <%= link_to("登出", destroy_user_session_path,
                           :method => :delete ) %></li>
        </ul>
      </li>
      <% end %>
...
...
```

調整修改 註冊 / 登入頁面

因為導入 bootstrap ， 所以跟原始的 CSS 設定有衝突導致版面炸掉 我們可以手動修改，讓頁面美觀

本 blog 不解釋前端部分，請直接 copy 程式碼直接覆蓋就好

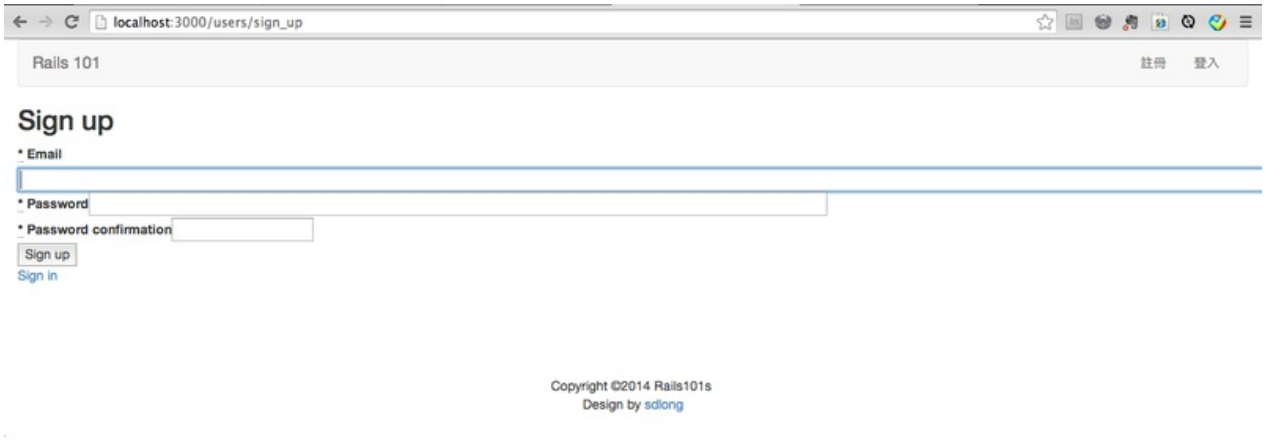
註冊頁面

原本

app/views/devise/registrations/new.html.erb

```
<h2>Sign up</h2>

<%= simple_form_for(resource, as: resource_name, url: registration_path(resource_name)) do |f| %>
  <%= f.error_notification %>

  <div class="form-inputs">
    <%= f.input :email, required: true, autofocus: true %>
    <%= f.input :password, required: true %>
    <%= f.input :password_confirmation, required: true %>
  </div>

  <div class="form-actions">
    <%= f.button :submit, "Sign up" %>
  </div>
<% end %>

<%= render "devise/shared/links" %>
```

before

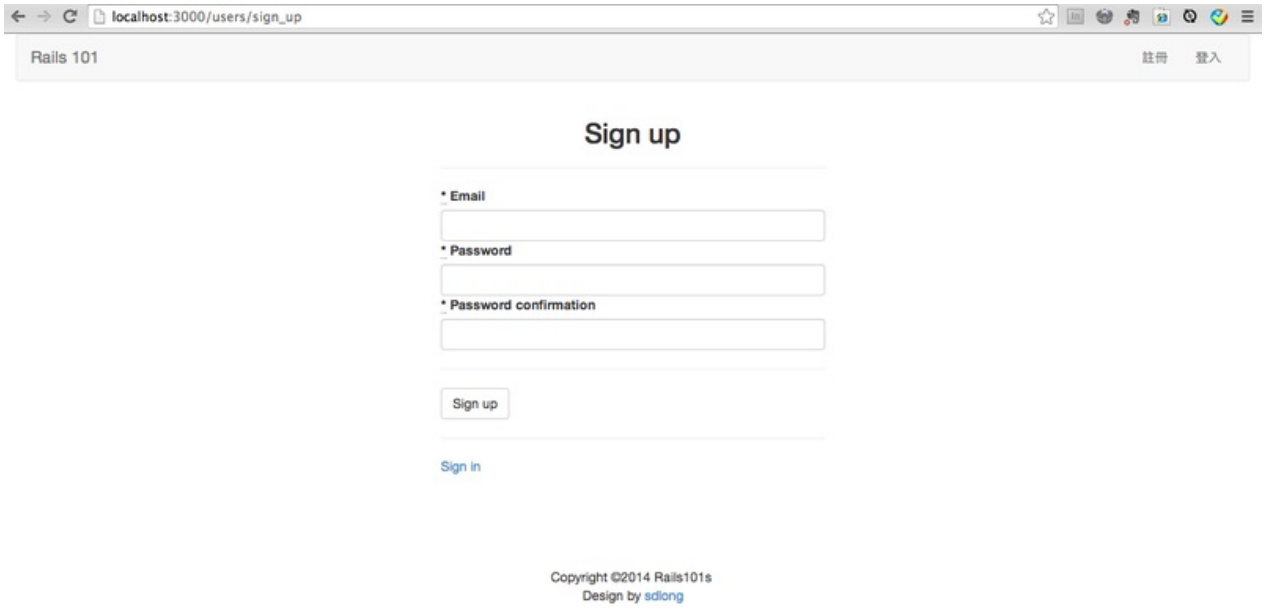


改成

app/views/devise/registrations/new.html.erb

```
<div class="col-md-4 col-md-offset-4">

  <h2 class="text-center">Sign up</h2>

  <hr>

  <%= simple_form_for(resource, as: resource_name, url: registration_path(resource_name)) do |f| %>
    <%= f.error_notification %>

    <div class="form-inputs">
      <%= f.input :email, :input_html => { :class => "form-control"}, required: true, autofocus: true %>
      <%= f.input :password, :input_html => { :class => "form-control"}, required: true %>
      <%= f.input :password_confirmation, :input_html => { :class => "form-control"}, required: true %>
    </div>

    <hr>

    <div class="form-actions">
      <p><%= f.button :submit, "Sign up", :class => "btn btn-default" %></p>
    </div>
  <% end %>

  <hr>
  <%= render "devise/shared/links" %>
</div>
```

after



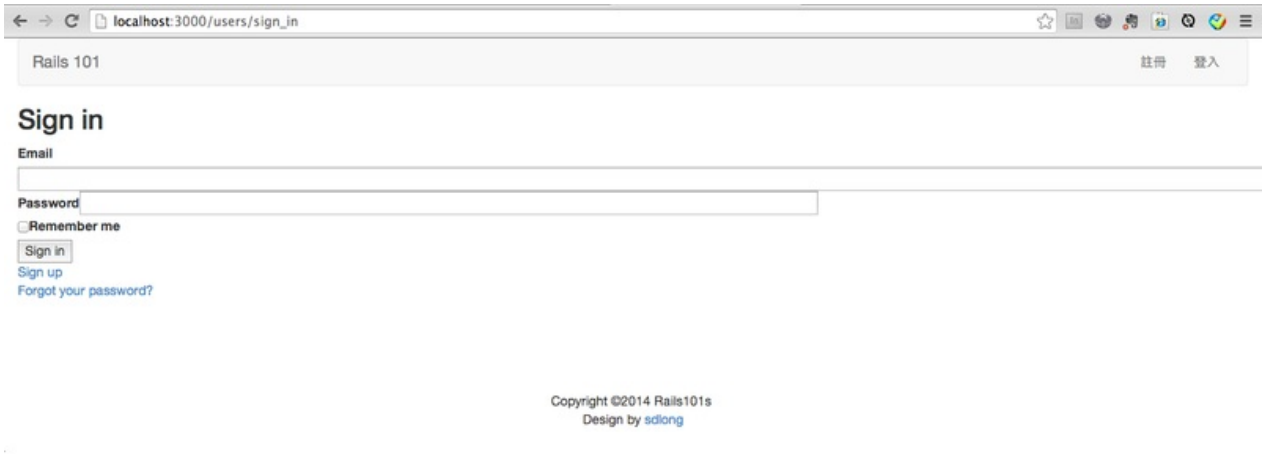
登入頁面

原本

app/views/devise/sessions/new.html.erb

```
<h2>Sign in</h2>

<%= simple_form_for(resource, as: resource_name, url: session_path(resource_name)) do |f| %>
  <div class="form-inputs">
    <%= f.input :email, required: false, autofocus: true %>
    <%= f.input :password, required: false %>
    <%= f.input :remember_me, as: :boolean if devise_mapping.rememberable? %>
  </div>

  <div class="form-actions">
    <%= f.button :submit, "Sign in" %>
  </div>
<% end %>

<%= render "devise/shared/links" %>
```

before

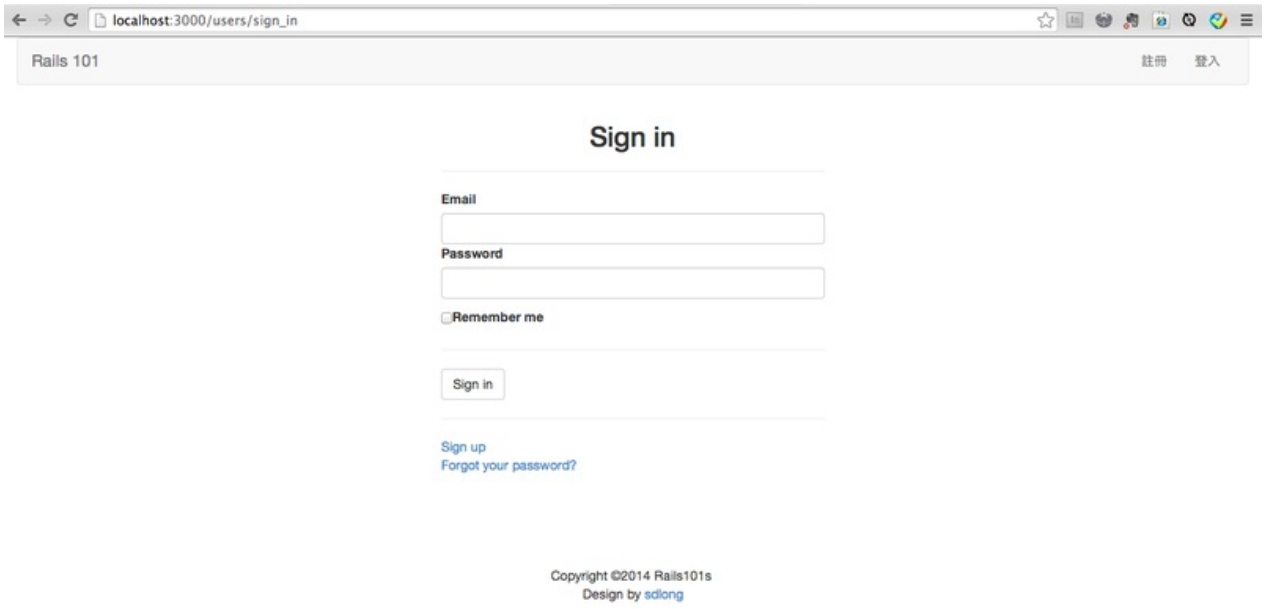


改成

app/views/devise/sessions/new.html.erb

```
<div class="col-md-4 col-md-offset-4">

  <h2 class="text-center">Sign in</h2>

  <hr>
  <%= simple_form_for(resource, as: resource_name, url: session_path(resource_name)) do |f| %>
    <div class="form-inputs">
      <%= f.input :email, :input_html => { :class => "form-control"}, required: false, autofocus: true %>
      <%= f.input :password, :input_html => { :class => "form-control"}, required: false %>
   <p><%= f.input :remember_me, as: :boolean if devise_mapping.rememberable? %></p>
    </div>

    <hr>
    <div class="form-actions">
      <%= f.button :submit, "Sign in", :class => "btn btn-default" %>
    </div>
  <% end %>

  <hr>
  <%= render "devise/shared/links" %>
</div>
```

after



利用 **before_action :authenticate_user!** 來做要求登入的設定

```
before_action :authenticate_user!
```

這是 devise 內建的功能，只要把它放進 controller 裡面， 就會自動驗證使用者是否入

if yes => 繼續下面的程序 if no => 轉到登入畫面

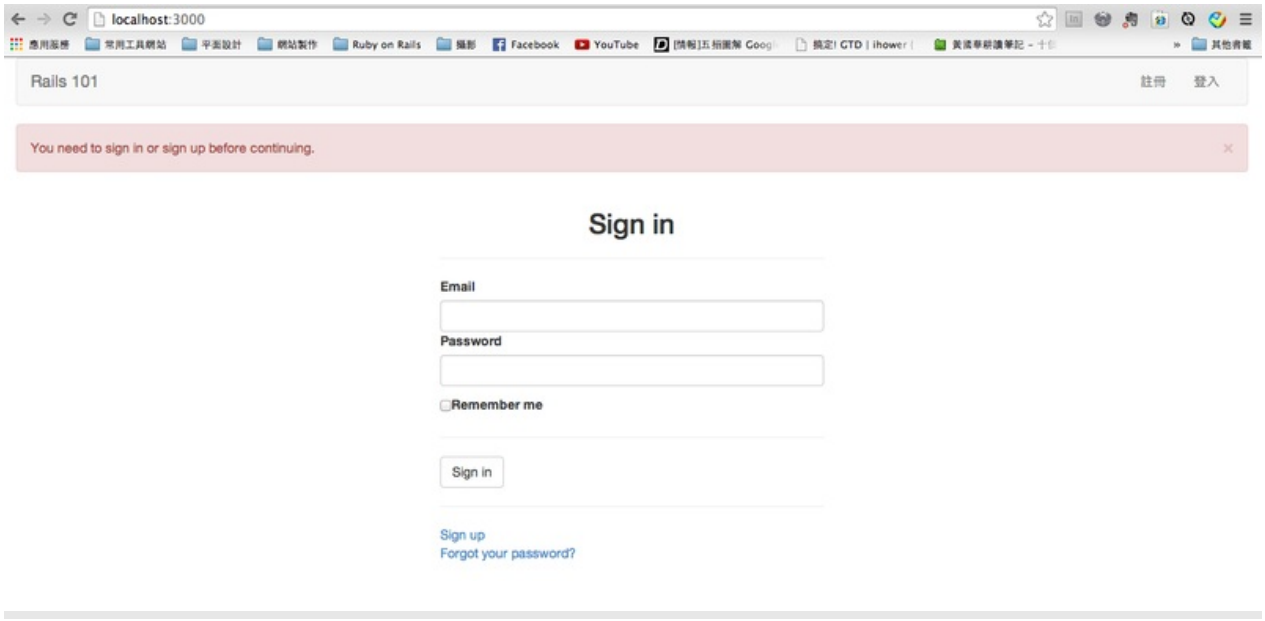
我們可以把這一行放進前面做的二個 controller : groups 跟 posts 裡面

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  before_action :authenticate_user!
...
...
```

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  before_action :authenticate_user!
...
...
```



new, create, edit, update, destroy 等 action 必須先登入才能操作

我們會發現，不是所有 action 都一定要登入才行 只有跟 新增 / 修改 / 刪除 有關的 action 才需要先登入

原本

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  before_action :authenticate_user!
...
...
```

改成

app/controllers/groups_controller.rb

```
class GroupsController < ApplicationController

  before_action :authenticate_user!, :only => [:new, :edit, :create, :update, :destroy]
...
...
```

posts_controller 由於只有以上五個 action, 就不需要再設定 :only

最後別忘了 git 存檔

`git add .`

`git commit -m "install devise"

使用者功能新增 **name** 欄位 與 帳號設定 功能

原文網址

本章的作業目標:

- 客製化 devise => 新增一個 "name" 的欄位
- 客製化 devise 的 views => 註冊頁面新增 name 欄位
- 新增 "帳號設定" 功能，並能修改自己帳號的 name (前端 + 後端)

客製化 **devise =>** 新增一個 **"name"** 的欄位

devise 內建的資料格式並沒有 name 這個欄位，所以我們要客製化一個出來

`rails g migration add_name_to_user` 新增一個資料庫異動設定, 名稱是 add_name_to_user (user 表單新增一個 name 欄位)

打開新增的檔案, 位於 db/migrate/(一堆數字)_add_name_to_user.rb

原本

db/migrate/(一堆數字)_add_name_to_user.rb

```
class AddNameToUser < ActiveRecord::Migration
  def change
  end
end
```

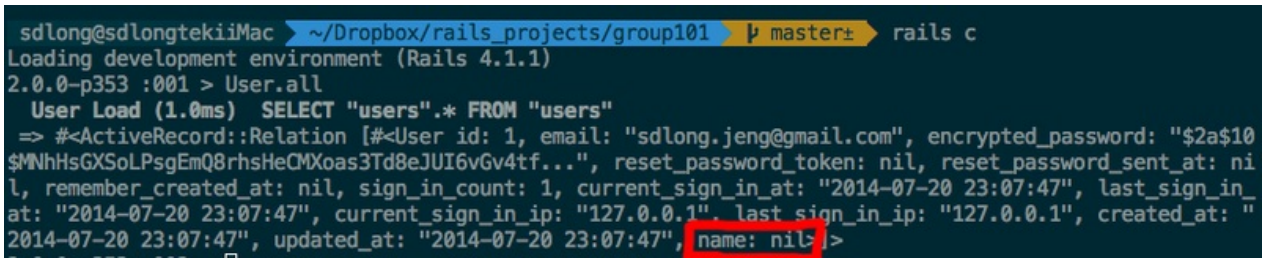
改成

db/migrate/(一堆數字)_add_name_to_user.rb

```
class AddNameToUser < ActiveRecord::Migration
  def change
    add_column :users, :name, :string
  end
end
```

接下來跑 `rake db:migrate`

這樣 user 資料庫就有 name 這個欄位了



--

客製化 **devise** 的 **views =>** 註冊頁面新增 **name** 欄位

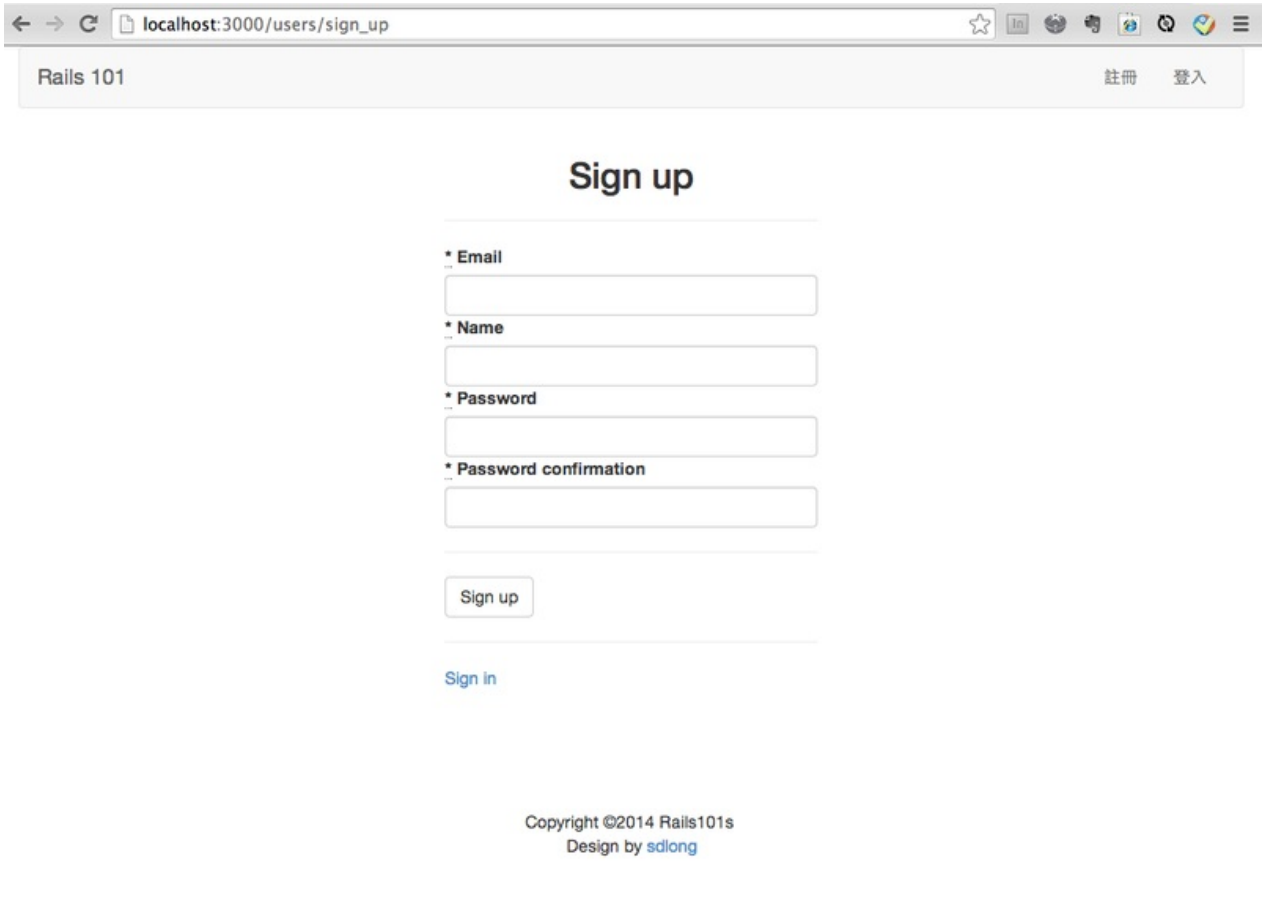
打開 **app/views/devise/registrations/new.html.erb**

原本

```
...
...
    <%= f.input :email, :input_html => { :class => "form-control"}, required: true, autofocus: true %>
    <%= f.input :password, :input_html => { :class => "form-control"}, required: true %>
    <%= f.input :password_confirmation, :input_html => { :class => "form-control"}, required: true %>
  </div>
...
...
```

改成

```
...
...
    <%= f.input :email, :input_html => { :class => "form-control"}, required: true, autofocus: true %>
    <%= f.input :name, :input_html => { :class => "form-control"}, required: true %>
    <%= f.input :password, :input_html => { :class => "form-control"}, required: true %>
    <%= f.input :password_confirmation, :input_html => { :class => "form-control"}, required: true %>
  </div>
...
...
```



現在還只是前端的部分完成，後端還要再加一個設定，才能讓 name 輸入的值真正存到資料庫裡面

加入 **strong_parameters** 與 **devise** 整合的 **hack**

打開 **app/controller/application_controller**

原本

```
class ApplicationController < ActionController::Base
...
...

end
```

改成

```
class ApplicationController < ActionController::Base
...
...

  before_filter :configure_permitted_parameters, if: :devise_controller?


    protected

    def configure_permitted_parameters
     devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:name, :email, :password, :password_confirmati
    end
end
```

這樣註冊新會員功能就完成了， 請新建立一個測試用帳號 來測試功能

後面將會開篇章節來解釋什麼是 strong_parameters

---

navbar 上的 **hi! email** 改成 **hi! name**

既然我們都能讓會員取名字了， navbar 上的 greeting 也該改成 name 吧？

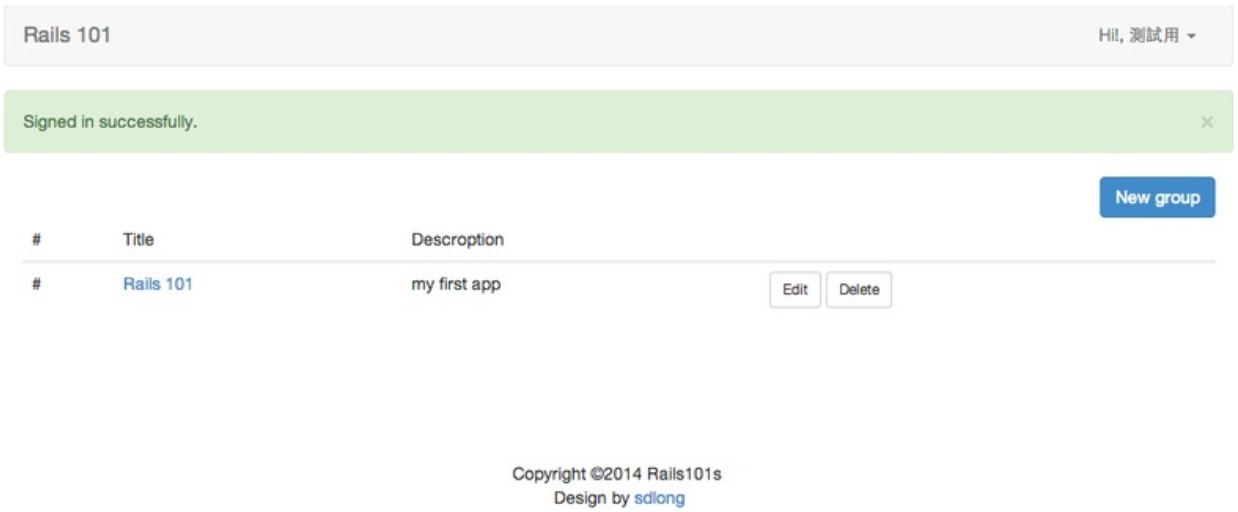
打開 **app/views/common/_navbar.html.erb**

原本

```
...
...
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">
          Hi!, <%= current_user.email %>
          <b class="caret"></b>
        </a>
...
...
```

改成

```
...
...
      <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">
          Hi!, <%= current_user.name %>
          <b class="caret"></b>
        </a>
...
...
```



新增 "帳號設定" 功能，並能修改自己帳號的 **name (前端 + 後端)**

我們在前面章節也有創造一個帳號，但是卻還沒命名

所以需要做一個 "使用者帳號設定" 頁面來把還沒命名的帳號命名

Devise 已經幫我們建好內建的功能

打開 **app/views/common/_navbar.html.erb**

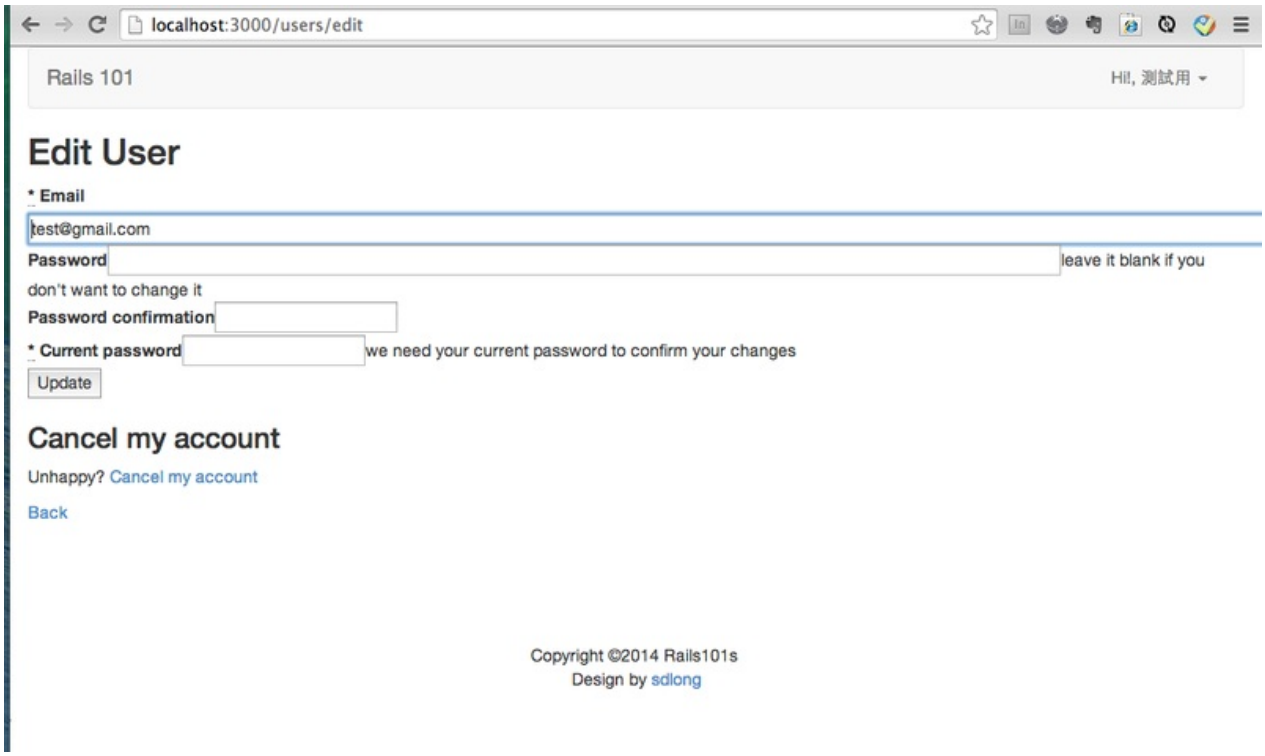
原本

```
...
...
        <ul class="dropdown-menu">
          <li> <%= link_to("登出", destroy_user_session_path,
                        :method => :delete ) %></li>
        </ul>
...
...
```

改成

```
...
...
        <ul class="dropdown-menu">
          <li><%= link_to("帳號設定", edit_user_registration_path )%></li>
          <li> <%= link_to("登出", destroy_user_session_path,
                        :method => :delete ) %></li>
        </ul>
...
...
```

一樣有版面炸掉的問題

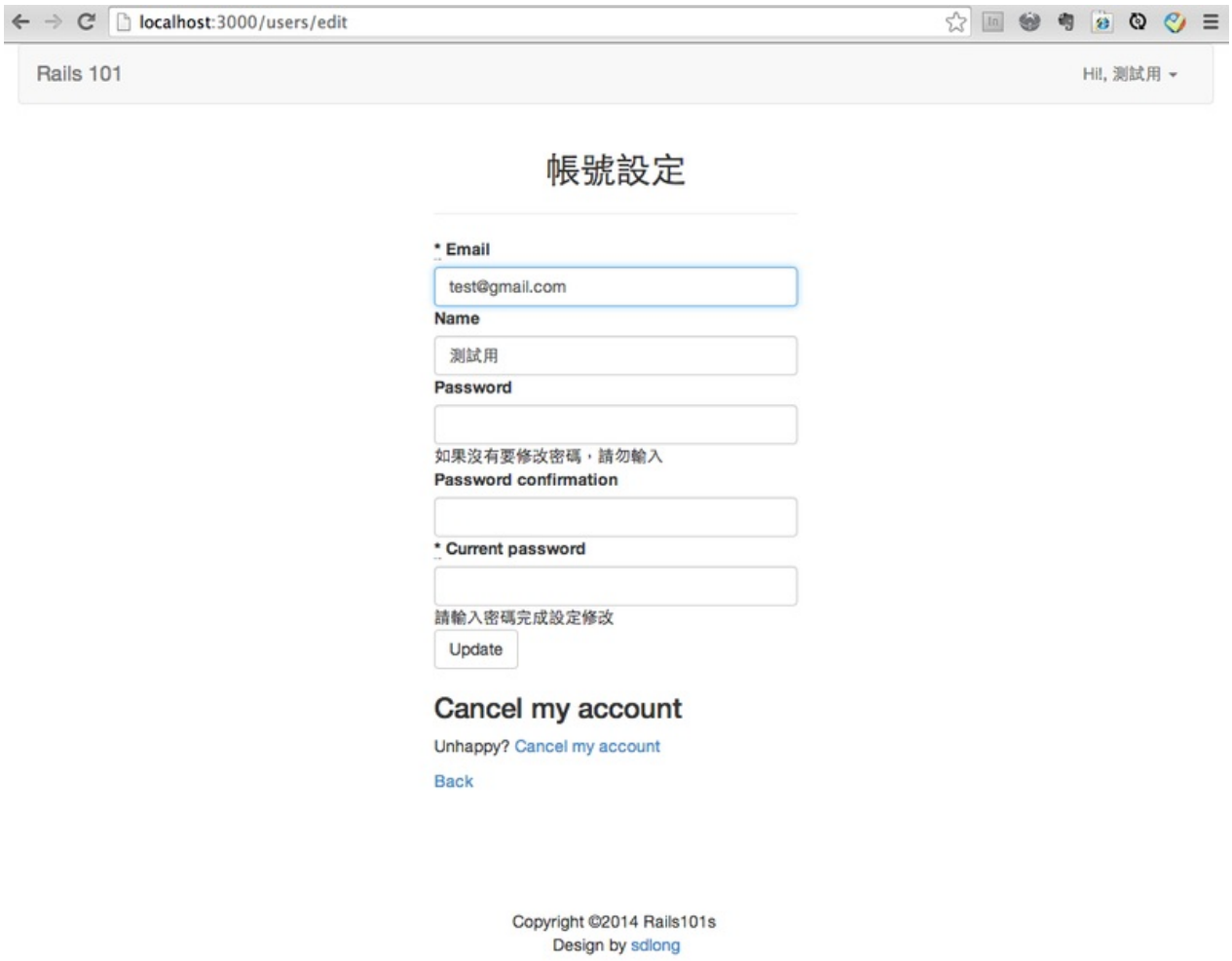


由於前端設計不在本 blog 的範疇，所以請直接複製貼上

打開 **app/views/devise/registrations/edit.html.erb**

```
<div class="col-md-4 col-md-offset-4">

  <h2 class="text-center">帳號設定</h2>

  <hr>

  <%= simple_form_for(resource,
                      as: resource_name,
                      url: registration_path(resource_name),
                      html: { method: :put }) do |f| %>

    <%= f.error_notification %>

    <div class="form-inputs">

      <%= f.input :email,
                :input_html => { :class => "form-control"},
                required: true, autofocus: true %>

      <% if devise_mapping.confirmable? && resource.pending_reconfirmation? %>
        <p>Currently waiting confirmation for: <%= resource.unconfirmed_email %></p>
      <% end %>

      <%= f.input :name,
                :input_html => { :class => "form-control"} %>

      <%= f.input :password,
                :input_html => { :class => "form-control"},
                autocomplete: "off",
                hint: "如果沒有要修改密碼，請勿輸入",
                required: false %>

      <%= f.input :password_confirmation,
                :input_html => { :class => "form-control"},
                required: false %>

      <%= f.input :current_password,
                :input_html => { :class => "form-control"},
                hint: "請輸入密碼完成設定修改",
                required: true %>
    </div>

    <div class="form-actions">
      <%= f.button :submit, "Update", :class => "btn btn-default" %>
    </div>
  <% end %>

  <h3>Cancel my account</h3>

  <p>Unhappy? <%= link_to "Cancel my account",
                      registration_path(resource_name),
                      data: { confirm: "Are you sure?" },
                      method: :delete %></p>

  <%= link_to "Back", :back %>

</div>
```

修改頁面的 view 完成



再修改 edit_account 的 strong_parameters

原本

app/controller/application_controller

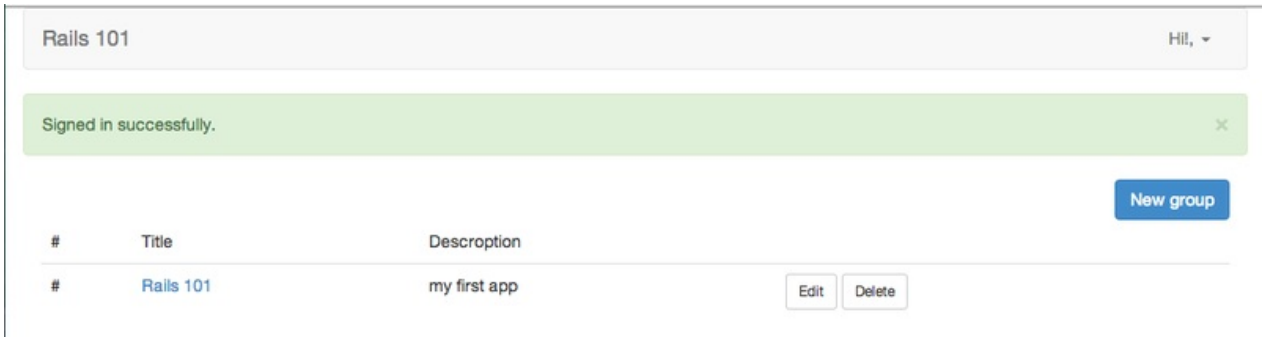
```
...
...
  def configure_permitted_parameters
   devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:name, :email, :password, :password_confirmati
  end
end
```

改成

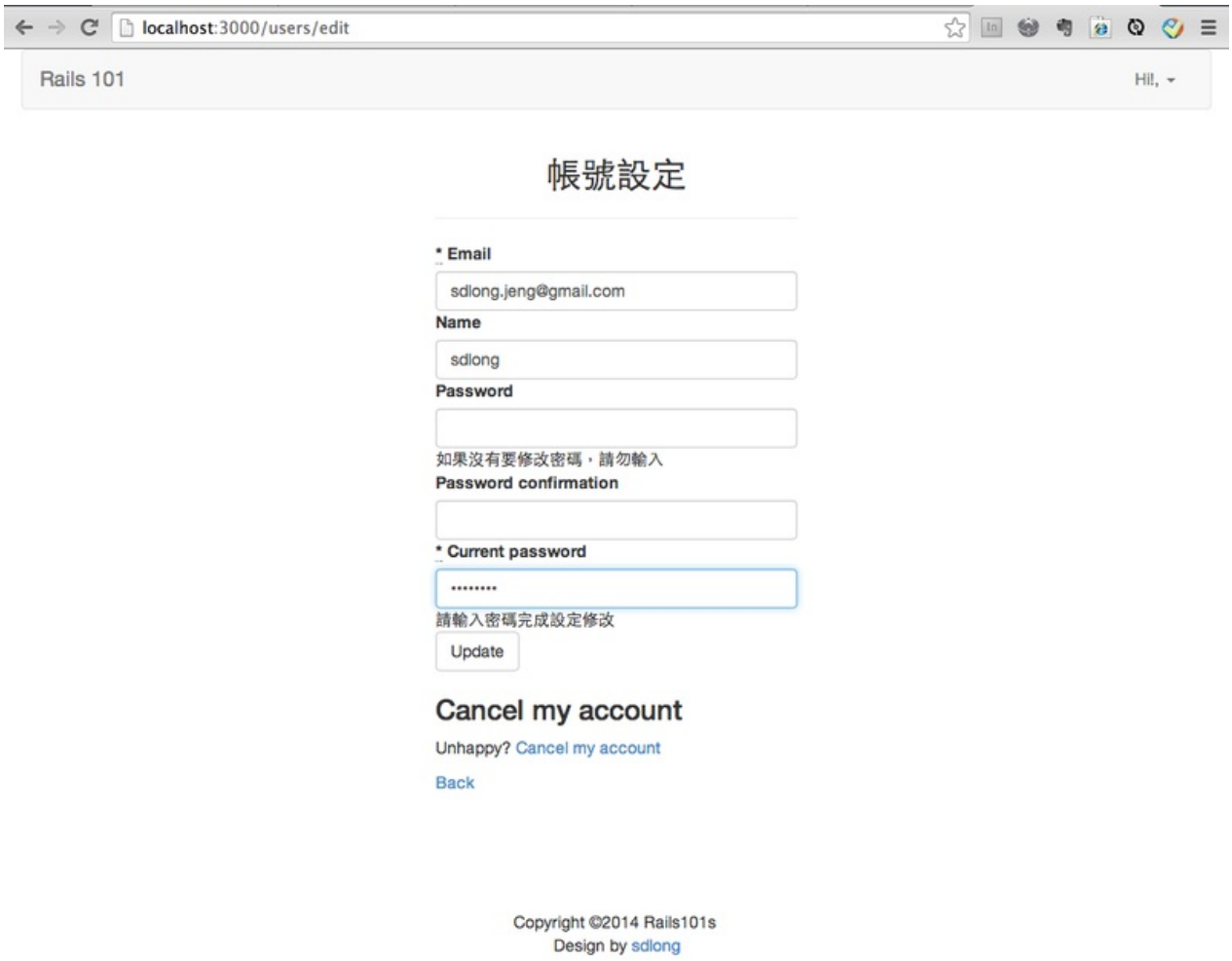
app/controller/application_controller

```
...
...
  def configure_permitted_parameters
   devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:name, :email, :password, :password_confirmati
   devise_parameter_sanitizer.for(:account_update) { |u| u.permit(:name, :email, :password, :password_c
  end
end
```

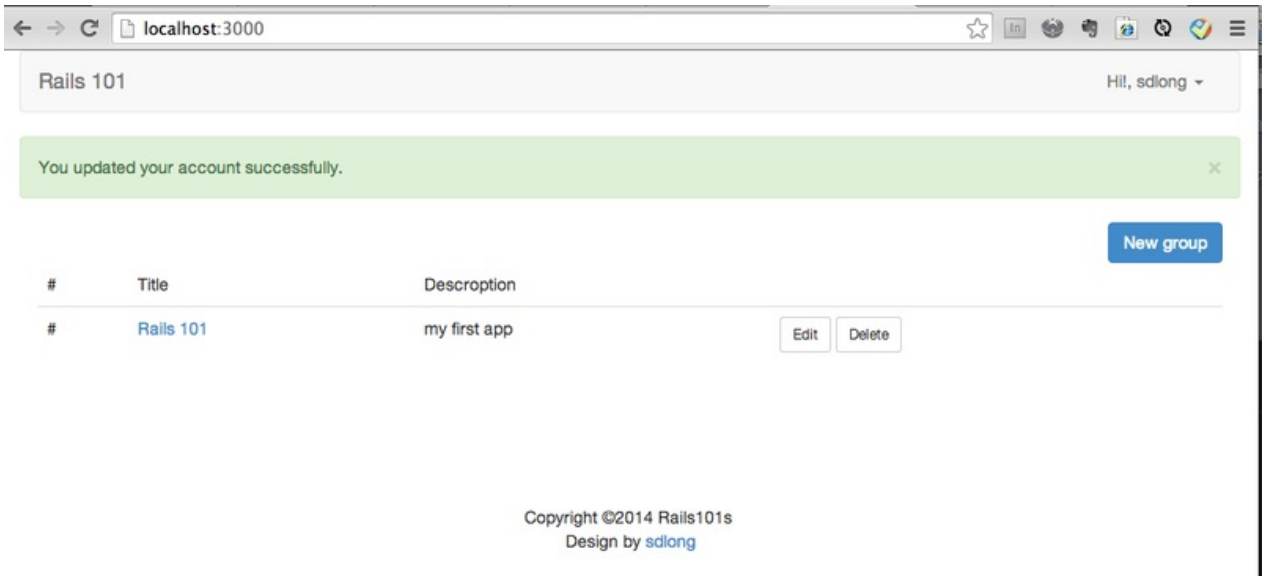
請切回上一章所建立的帳號



進入修改帳號頁面，填上你要取的 name 再輸入密碼 送出 update



右上角就顯示你的 name 了 !!



最後別忘了 git 存檔

`git add .`

`git commit -m "完成 user 新增 name 欄位與 帳號設定 功能"`

使用者功能帶進 **group** 與 **post** 裡 **(** 作者機制 **)**

原文網址

本章的作業目標:

- 使用者功能 整合進 group 裡面
- 只有作者才能有 group 的修改/刪除權限
- 使用者功能 整合進 post 裡面
- 只有作者才能有 post 的修改/刪除權限

使用者功能 整合進 **group** 裡面

group 資料表新增 **user_id** 欄位

`rails g migration add_user_id_to_group`

打開 db/migrate/(一堆數字)_add_user_id_to_group.rb

原本

db/migrate/(一堆數字)_add_user_id_to_group.rb

```
class AddUserIdToGroup < ActiveRecord::Migration
  def change
  end
end
```

改成

db/migrate/(一堆數字)_add_user_id_to_group.rb

```
class AddUserIdToGroup < ActiveRecord::Migration
  def change
    add_column :groups, :user_id, :integer
  end
end
```

`rake db:migrate` 這樣 group 資料庫裡面，就多了 user_id 欄位

對 migration 想了解更多的 可參考 Rails 資料庫遷移(中文) / Rails 資料庫遷移(英文)

設定 **user** 跟 **group** 之間的資料庫關聯 **(database relationship)**

打開 **app/models/user.rb**

原本

```
class User < ActiveRecord::Base
...
...
end
```

改成

```
class User < ActiveRecord::Base
...
...
  has_many :groups

end
```

打開 **app/models/group.rb**

原本

```
class Group < ActiveRecord::Base
  validates :title, :presence => true

  has_many :posts
end
```

改成

```
class Group < ActiveRecord::Base
  validates :title, :presence => true

  has_many :posts

  belongs_to :owner, :class_name => "User", :foreign_key => :user_id
end
```

延伸閱讀 : Rails Active Record 關聯(中文)

只有作者才能有 **group** 的修改/刪除權限

打開 app/controllers/group_controller.rb 只要找出 edit, create, update, destroy 這四個 action

原本 (只列出需要修改的部分)

```
class GroupsController < ApplicationController
…
…
  def edit
    @group = Group.find(params[:id])
  end

  def create
    @group = Group.new(group_params)
…
…
  end

  def update
    @group = Group.find(params[:id])
…
…
  end

  def destroy
    @group = Group.find(params[:id])
…
…
  end
…
…
```

改成

```
class GroupsController < ApplicationController
…
…
  def edit
    @group = current_user.groups.find(params[:id])
  end

  def create
    @group = current_user.groups.new(group_params)
…
…
  end

  def update
    @group = current_user.groups.find(params[:id])
…
…
  end

  def destroy
    @group = current_user.groups.find(params[:id])
…
…
  end
…
…
```

解說

這段 code 簡單來說就是 把 Group 換成 current_user (無誤！)

運作邏輯是這樣:

把 create, edit, update, destroy 這四個 action 的運作

從原本

> 只是單純呼叫 group 的資料庫來 找某筆資料(.find(params[:id])) or 建立一筆新資料 (.new(group_params))

改成

> 登入的使用者(current_user) 所擁有的 " group資料 (.groups) " 來做 找某筆資料(.find(params[:id])) or 建立一筆新資料 (.new(group_params))

當運作 create 創建新資料時 就會把 登入的使用者 (current_user) 的 id 寫進 group 資料庫裡的 user_id 的欄位裡面

所以這個 group 資料裡的 『 user_id 的欄位裡的值 』 就會等於 『 使用者(user)的 id 』 => 自動弄出作者這功能

至於 edit, update, destroy 這三個會讓資料異動的功能 則會自動驗證 group 裡的 user_id 跟 current_user 的 id 是否一致

如果是 true => 執行後續的程序 如果是 false => 直接跳 error (開發模式) or 404 (產品模式)

修改 **view =>** 只有作者才會出現 **edit / delete** 按鈕

打開 **app/models/group.rb**

原本

```
...
...
  belongs_to :owner, :class_name => "User", :foreign_key => :user_id
end
```

改成

```
...
...
  belongs_to :owner, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == owner
  end
end
```

打開 app/views/groups/index.html.erb

原本

```
...
...
    <td>  <%= link_to("Edit",
                edit_group_path(group),
                :class => "btn btn-sm btn-default")%>

          <%= link_to("Delete",
                group_path(group),
                :class => "btn btn-sm btn-default",
                :method=>:delete,
                data: { confirm: "Are you sure?" } )%></td>
...
...
```

改成

```
...
...
    <td>
        <% if current_user && group.editable_by?(current_user) %>
          <%= link_to("Edit",
                edit_group_path(group),
                :class => "btn btn-sm btn-default")%>

          <%= link_to("Delete",
                group_path(group),
                :class => "btn btn-sm btn-default",
                :method=>:delete,
                data: { confirm: "Are you sure?" } )%>
        <% end %>
    </td>
...
...
```

解說

這段 code 重點在於

```
# app/models/group.rb
  def editable_by?(user)
    (user) && (user == owner)
  end

# app/views/groups/index.html.erb
<% if current_user && group.editable_by?(current_user) %>
```

會去驗證 "登入的使用者" 跟 "作者的 id" 是否一致

if true => 顯示 edit 跟 delete 按鈕 if false => 隱藏

使用者功能 整合進 **post** 裡面

跟上述差不多，就直接貼 code , bj4 (不解釋) 了 ... XD

`rails g migration add_user_id_to_post user_id:integer`

`rake db:migrate`

打開 **app/models/user.rb**

原本

```
class User < ActiveRecord::Base
...
...
  has_many :groups
end
```

改成

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

打開 **app/models/post.rb**

原本

```
class Post < ActiveRecord::Base
..
..
end
```

改成

```
class Post < ActiveRecord::Base
...
...
  belongs_to :author, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == author
  end
end
```

打開 **app/controllers/posts_controller.rb**

原本

```
...
...

  def edit
    @post = @group.posts.find(params[:id])
  end

  def create
    @post = @group.posts.new(post_params)
...
...
  end

  def update
    @post = @group.posts.find(params[:id])
...
...
  end

  def destroy
    @post = @group.posts.find(params[:id])
...
...
  end
...
...
```

改成

```
...
...

  def edit
    @post = current_user.posts.find(params[:id])
  end

  def create
    @post = @group.posts.new(post_params)
    @post.author = current_user
...
...
  end

  def update
    @post = current_user.posts.find(params[:id])
...
...
  end

  def destroy
    @post = current_user.posts.find(params[:id])
...
...
  end
...
...
```

打開 **app/views/groups/show.html.erb**

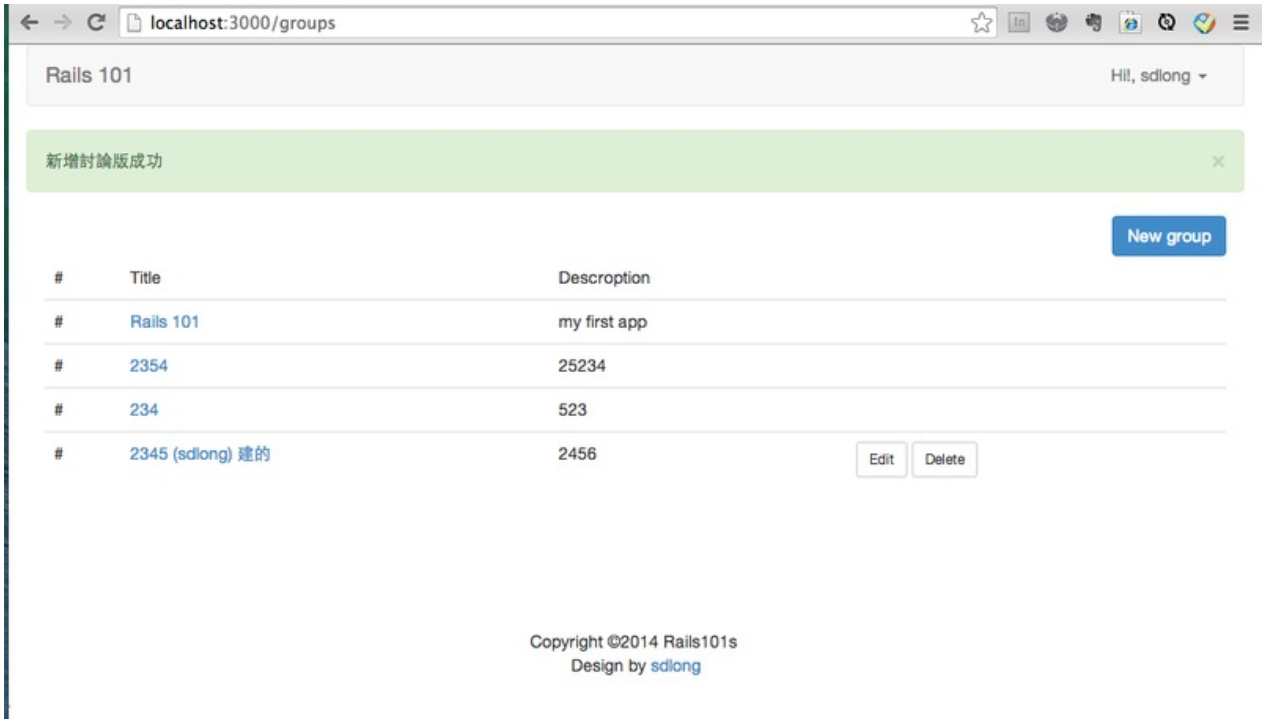
原本

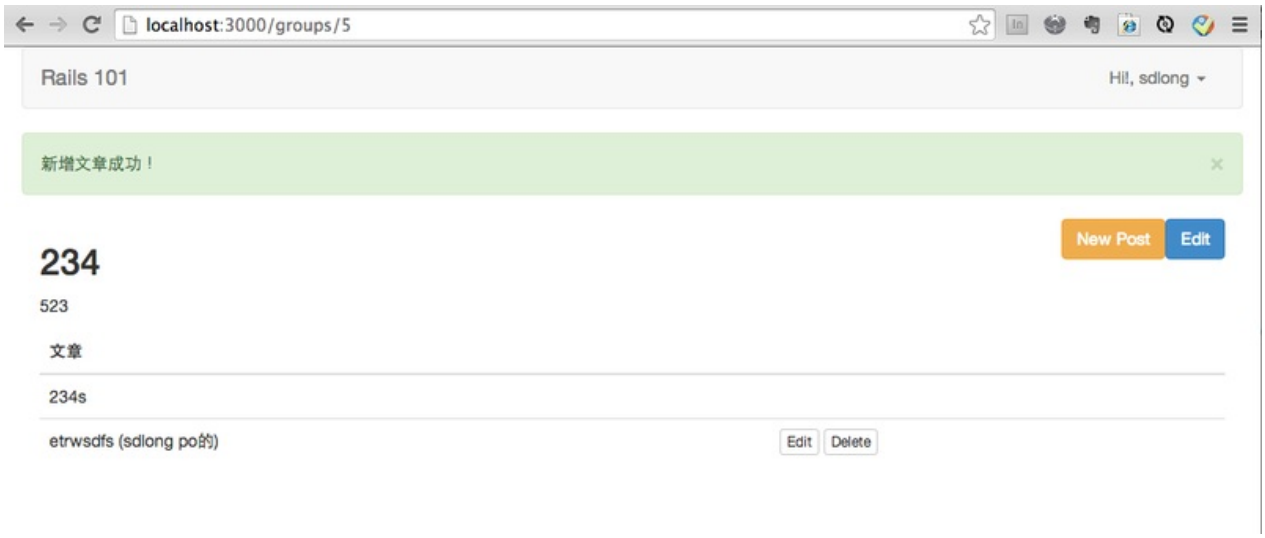
```
<td> <%= link_to("Edit",
            edit_group_post_path(post.group, post),
            :class => "btn btn-default btn-xs")%>

   <%= link_to("Delete",
            group_post_path(post.group, post),
            :class => "btn btn-default btn-xs ",
            :method => :delete,
            data: { confirm: "Are you sure?" } )%></td>
```

改成

```
<td>
   <% if current_user && post.editable_by?(current_user) %>
    <%= link_to("Edit",
              edit_group_post_path(post.group, post),
              :class => "btn btn-default btn-xs")%>

    <%= link_to("Delete",
              group_post_path(post.group, post),
              :class => "btn btn-default btn-xs ",
              :method => :delete,
              data: { confirm: "Are you sure?" } )%>
   <% end %>
</td>
```

完成圖





最後別忘記存檔

`git add .`

`git commit -m "使用者功能帶進 group 與 post 裡"`

user 可以加入、退出 group

原文網址

本章的作業目標:

- 建立 討論版成員 (group_user) 設定
- user 必須要是這個社團的成員才能發表文章
- user 可以在 group 頁面 加入 / 退出 此 group
- User 在建立 group 後自動成為 group 的一員
- 若不是 group 創辦人，就不會顯示 edit 按鈕

建立 討論版成員 (group_user) 設定

`rails g model group_user group_id:integer user_id:integer`

新增一個叫 **group_user** 資料表, 裡面的欄位是 group_id 跟 user_id

`rake db:migrate`

執行資料庫異動設定

打開 app/models/user.rb

原本

app/models/user.rb

```
...
...
  has_many :groups
  has_many :posts
end
```

改成

app/models/user.rb

```
...
...
  has_many :groups
  has_many :posts

  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group
end
```

打開 app/models/group.rb

原本

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true

  has_many :posts
...
...
```

改成

app/models/group.rb

```
class Group < ActiveRecord::Base
  validates :title, :presence => true

  has_many :posts
  has_many :group_users
  has_many :members, :through => :group_users, :source => :user
..
..
```

打開 **app/models/group_user.rb**

原本

app/models/group_user.rb

```
class GroupUser < ActiveRecord::Base
end
```

改成

app/models/group_user.rb

```
class GroupUser < ActiveRecord::Base
  belongs_to :user
  belongs_to :group
end
```

解說

[延伸閱讀] Active Record Association

---

user 必須要是這個社團的成員才能發表文章

打開 **ruby app/models/user.rb**

原本

app/models/user.rb

```
...
...
  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group
end
```

改成

app/models/user.rb

```
...
...
  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group

  def is_member_of?(group)
    participated_groups.include?(group)
  end
end
```

打開 **app/controllers/posts_controller.rb**

原本

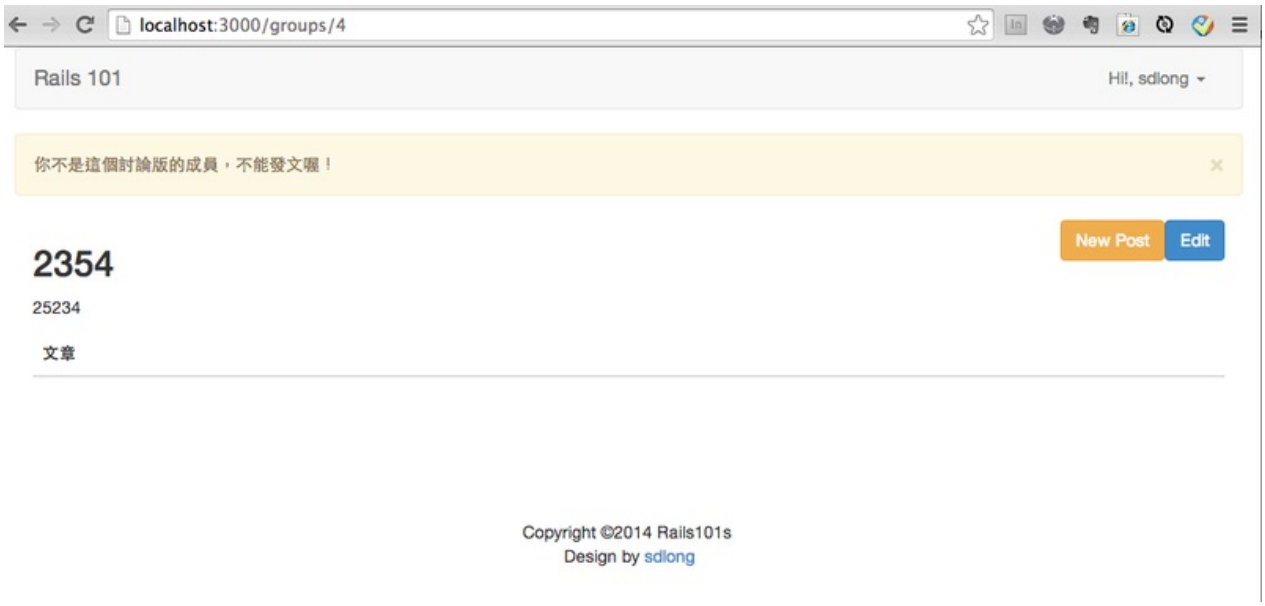
app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  before_action :authenticate_user!

  before_action :find_group
...
...
  private
..
..
end
```

改成

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController

  before_action :authenticate_user!

  before_action :find_group

  before_action :member_required, :only => [:new, :create ]
...
...
  private
..
..
  def member_required
    if !current_user.is_member_of?(@group)
      flash[:warning] = "你不是這個討論版的成員，不能發文喔！"
      redirect_to group_path(@group)
    end
  end
end
```



接下來要做出讓 user 可以加入退出 group 的功能

---

user 可以在 group 頁面 加入 / 退出 此 group

打開 app/models/user.rb

原本

app/models/user.rb

```
class User < ActiveRecord::Base
...
...
  def is_member_of?(group)
    participated_groups.include?(group)
  end
end
```

改成

app/models/user.rb

```
class User < ActiveRecord::Base
...
...
  def join!(group)
    participated_groups << group
  end

  def quit!(group)
    participated_groups.delete(group)
  end

  def is_member_of?(group)
    participated_groups.include?(group)
  end
end
```

打開 **app/controllers/groups_controller.rb**

原本

app/controllers/groups_controller.rb

```
...
...

  private
 ...
 ...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def join
    @group = Group.find(params[:id])

    if !current_user.is_member_of?(@group)
      current_user.join!(@group)
      flash[:notice] = "加入本討論版成功！"
    else
      flash[:warning] = "你已經是本討論版成員了！"
    end

    redirect_to group_path(@group)
  end

  def quit
    @group = Group.find(params[:id])

    if current_user.is_member_of?(@group)
      current_user.quit!(@group)
      flash[:alert] = "已退出本討論版！"
    else
      flash[:warning] = "你不是本討論版成員，怎麼退出 XD"
    end

    redirect_to group_path(@group)
  end


  private
 ...
 ...
```

打開 **config/routes**

原本

config/routes

```
...
...
  resources :groups do
    resources :posts
  end
...
...
```

改成

config/routes

```
...
...
  resources :groups do
    member do
      post :join
      post :quit
    end

    resources :posts
  end
...
...
```

打開 **app/views/groups/show.html.erb**

原本

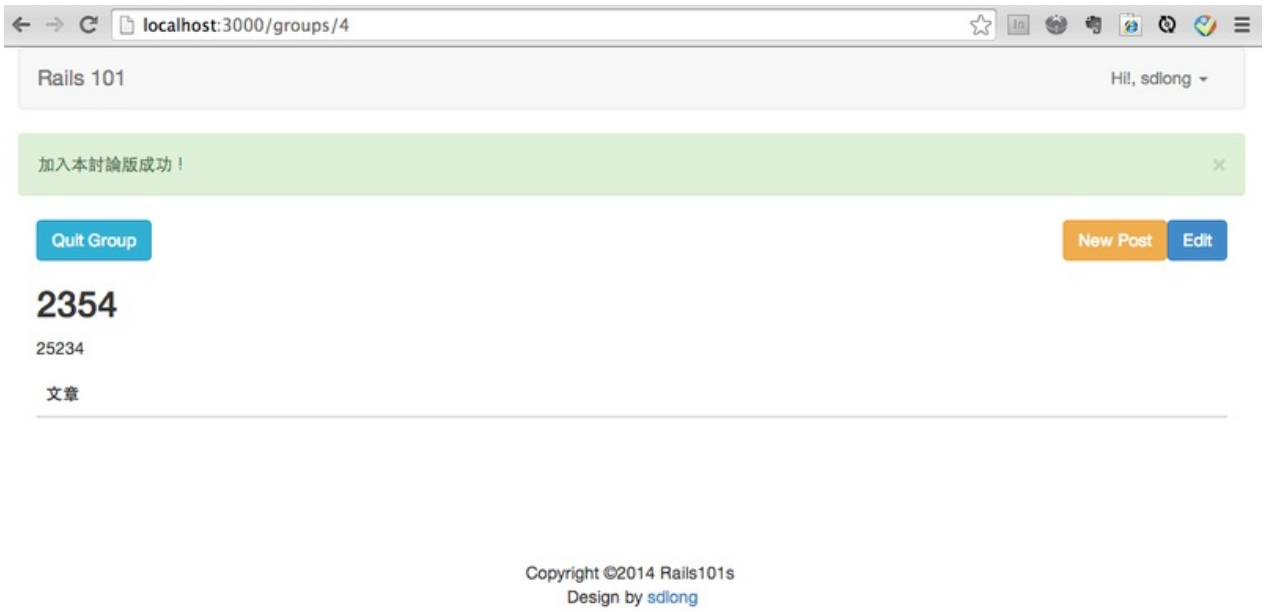
app/views/groups/show.html.erb

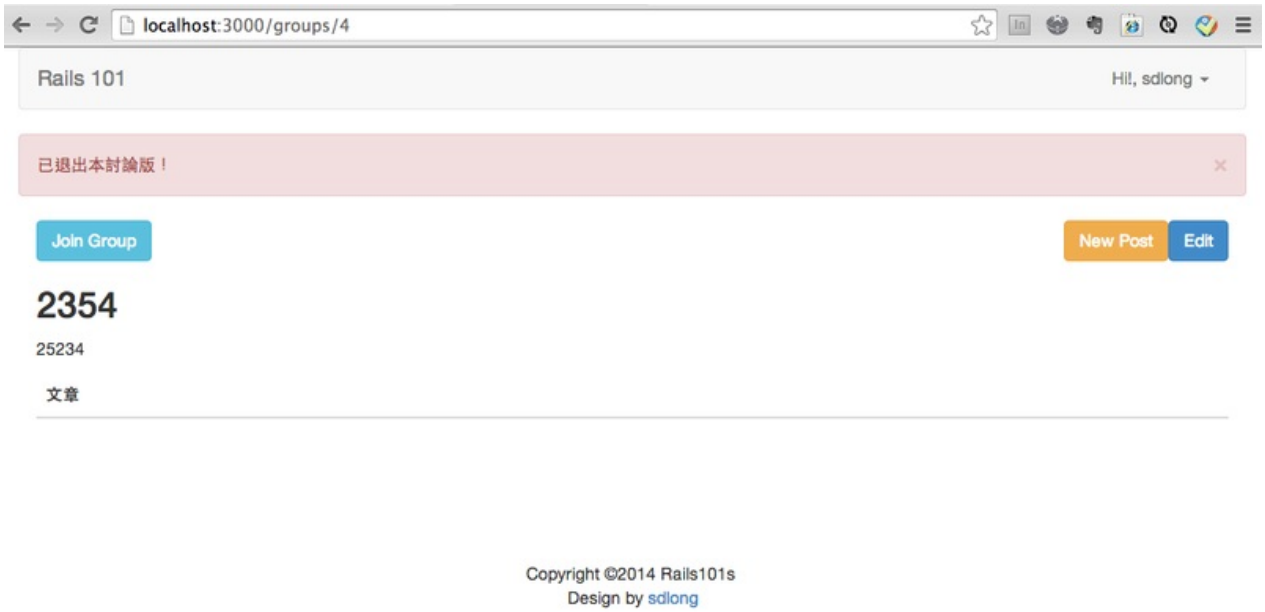
```
...
...
  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right")%>
    <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-warning pull-right")%>
  </div>
...
...
```

改成

app/views/groups/show.html.erb

```
...
...
  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right")%>
    <%= link_to("New Post", new_group_post_path(@group) , :class => "btn btn-warning pull-right")%>


    <% if !current_user.nil? %>
      <% if current_user.is_member_of?(@group) %>
        <%= link_to("Quit Group", quit_group_path(@group), :method => :post, :class => "btn btn-info") %
      <% else %>
        <%= link_to("Join Group", join_group_path(@group), :method => :post, :class => "btn btn-info") %
      <% end %>
    <% end %>

  </div>
...
...
```





User 在建立 group 後自動成為 group 的一員

當你創辦了 group， 卻還要手動再加入這個 group 不會很怪嗎？ XD

原本

app/controllers/groups_controller.rb

```
...
...
  def create
    @group = current_user.groups.new(group_params)

    if @group.save
      redirect_to groups_path, :notice => '新增討論版成功'
    else
      render :new
    end
  end
...
...
```

改成

app/controllers/groups_controller.rb

```
...
...
  def create
    @group = current_user.groups.new(group_params)

    if @group.save
      current_user.join!(@group)
      redirect_to groups_path, :notice => '新增討論版成功'
    else
      render :new
    end
  end
...
...
```

若不是 **group** 創辦人，就不會顯示 **edit** 按鈕

就算已經設定不是作者，按了 edit 也會跳 error 但不是作者還跑出 edit 來讓你按實在是很怪，對吧？

打開 **app/views/groups/show.html.erb**

原本

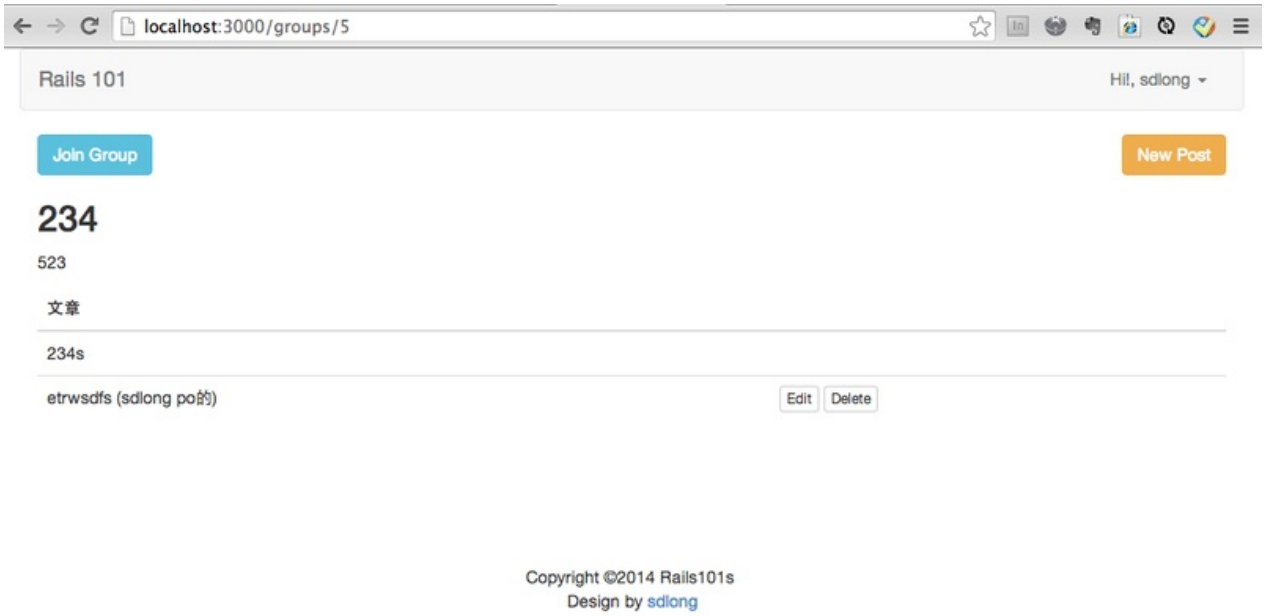
app/views/groups/show.html.erb

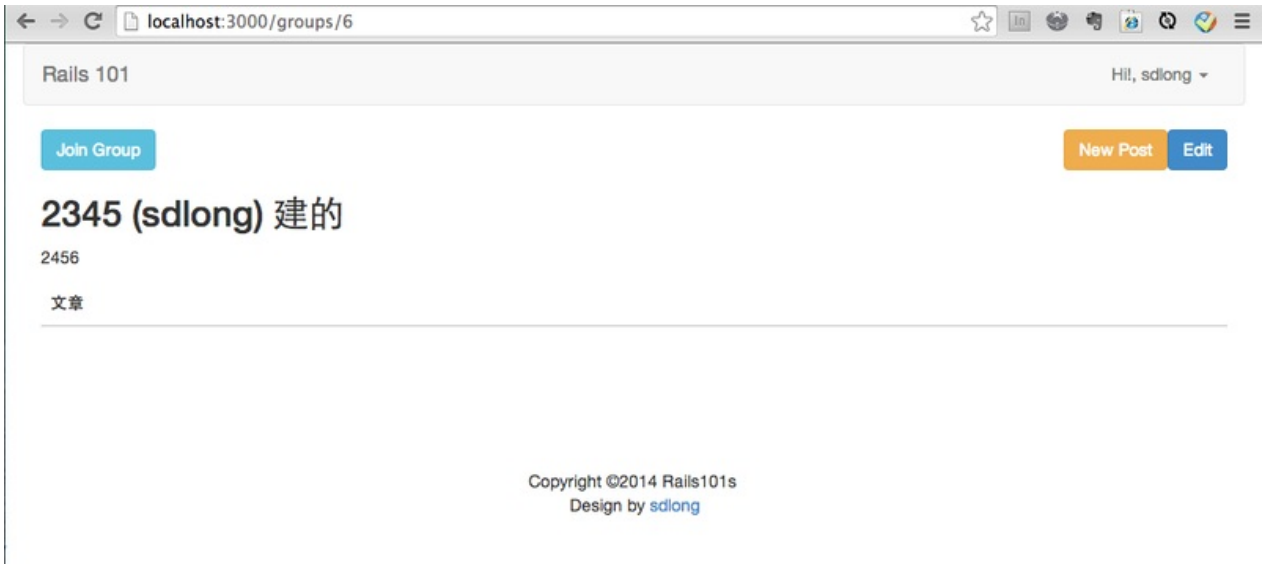
```
...
...
  <div class="group">
    <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right") %>
...
...
```

改成

app/views/groups/show.html.erb

```
...
...
  <div class="group">
    <% if @group.editable_by?(current_user) %>
      <%= link_to("Edit", edit_group_path(@group) , :class => "btn btn-primary pull-right") %>
    <% end %>
...
...
```





最後別忘了 git 存檔

`git add .`

`git commit -m "user 可以加入、退出 group"`

PS: 其實揪團 / 辦活動的網站裡面的『報名功能』，也是用類似這種功能做出來的喔！

實做簡單的後台機制

原文網址

本章的作業目標:

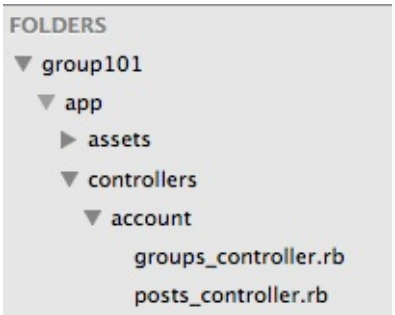
- 建立使用者後台 (前置作業)
- 使用者可以在後台看到自己參加的 group
- 使用可可以在後台看到自己發表的 post
- post 的排列要以時間 (最新 => 最舊) 順序排列
- group 的排列要以 文章數量 當熱門度排列
- bug 解決 => 當 group 刪除時，所屬的 posts 也要跟著刪除

建立使用者後台 (前置作業)

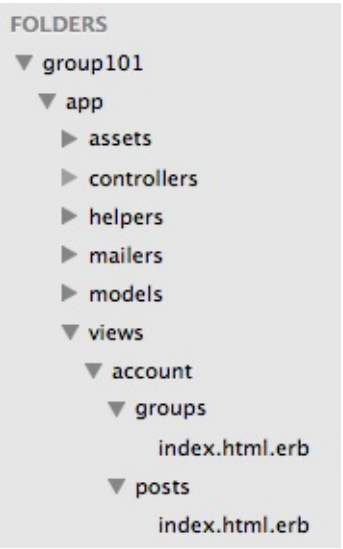
我們要做出二個後台網頁, 網址是 account/groups 跟 account/posts

`rails g controller account/groups`

`rails g controller account/posts`



在 app/views/account/groups 跟 app/views/account/posts 這二個資料夾 各建出一個 index.html.erb 檔案，內容是空的



打開 **config/routes**

原本

config/routes

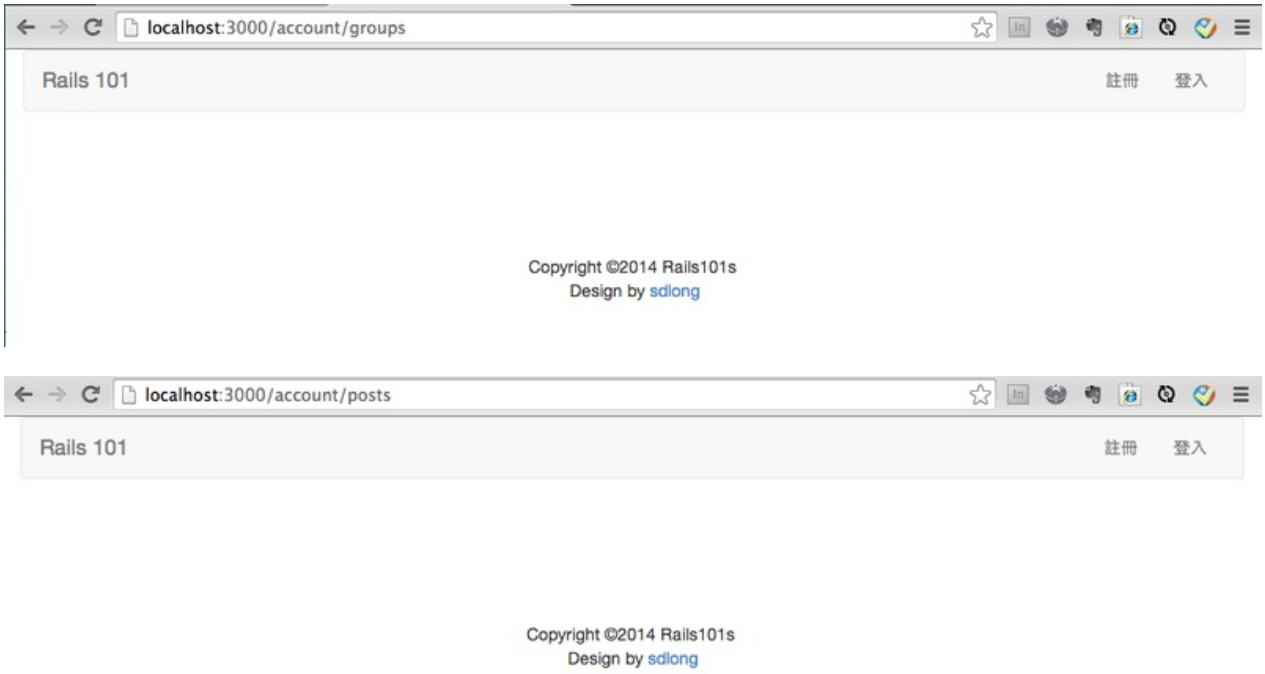
```
Rails.application.routes.draw do
...
...
```

改成

config/routes

```
Rails.application.routes.draw do

  namespace :account do
    resources :groups
    resources :posts
  end
...
...
```

這樣我們就能輸入後台網址，呈現空白網頁了！ localhost:3000/account/groups
localhost:3000/account/posts



navbar 裝上連到後台網址的按鈕

打開 app/views/common/_navbar.html.erb

原本

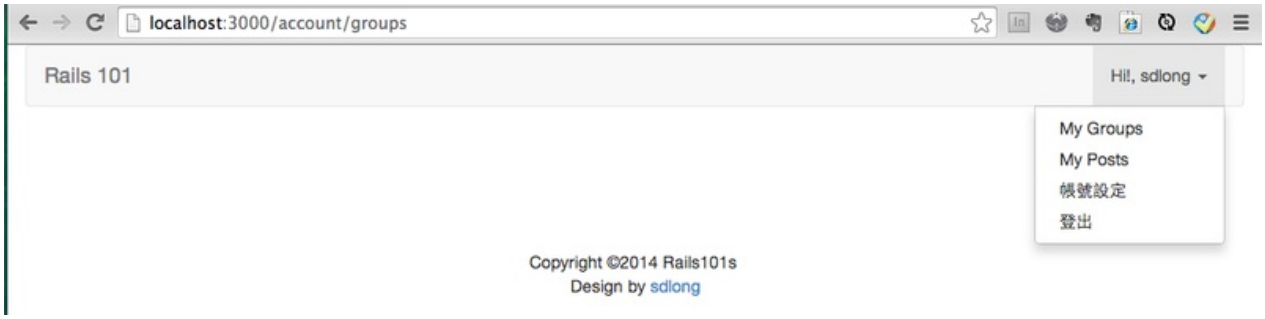
app/views/common/_navbar.html.erb

```
...
...
        <ul class="dropdown-menu">
          <li> <%= link_to("帳號設定", edit_user_registration_path )%></li>
          <li> <%= link_to("登出", destroy_user_session_path,
                      :method => :delete ) %></li>
        </ul>
...
...
```

改成

app/views/common/_navbar.html.erb

```
...
...
        <ul class="dropdown-menu">
          <li> <%= link_to("My Groups", account_groups_path) %></li>
          <li> <%= link_to("My Posts", account_posts_path) %></li>
          <li> <%= link_to("帳號設定", edit_user_registration_path )%></li>
          <li> <%= link_to("登出", destroy_user_session_path,
                      :method => :delete ) %></li>
        </ul>
...
...
```



接下來實做出後台的內容

使用者可以在後台看到自己參加的 **group**

打開 **app/controllers/account/groups_controller.rb**

原本

app/controllers/account/groups_controller.rb

```
class Account::GroupsController < ApplicationController
end
```

改成

app/controllers/account/groups_controller.rb

```
class Account::GroupsController < ApplicationController

  before_action :authenticate_user!

  def index
    @groups = current_user.participated_groups
  end
end
```

打開 **app/views/account/groups/index.html.erb**

原本

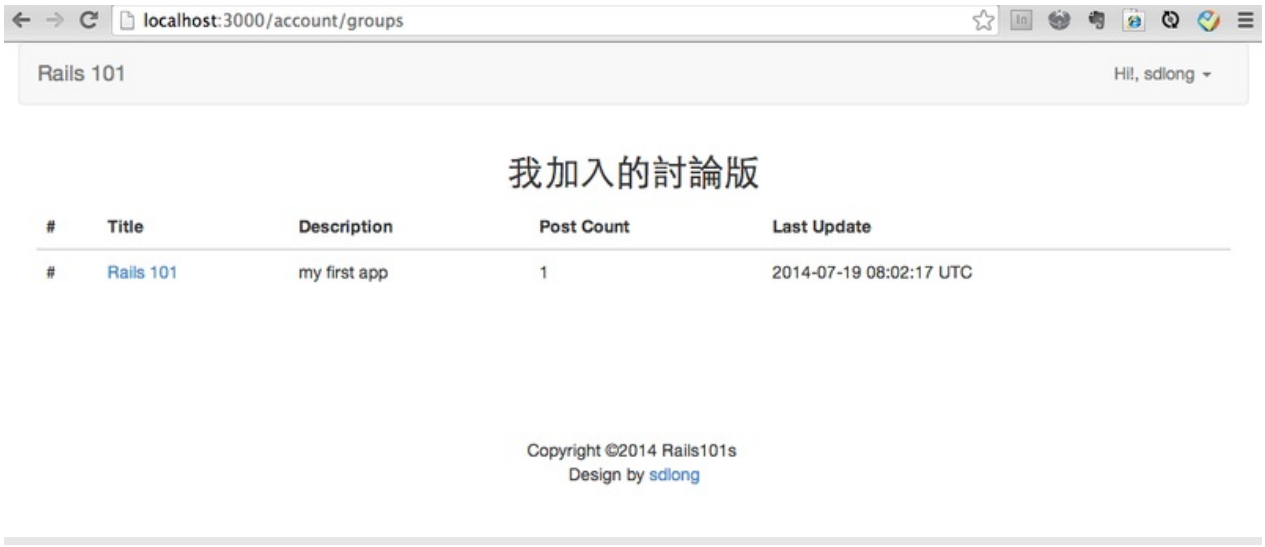
app/views/account/groups/index.html.erb

```
(空的)
```

改成

app/views/account/groups/index.html.erb

```
<div class="col-md-12">

  <h2 class="text-center"> 我加入的討論版 </h2>

  <table class="table">
    <thead>
      <tr>
        <th> # </th>
        <th> Title </th>
        <th> Description </th>
        <th> Post Count </th>
        <th> Last Update </th>
      </tr>
    </thead>

    <tbody>
      <% @groups.each do |group| %>
        <tr>
          <td> # </td>
          <td> <%= link_to(group.title, group_path(group)) %> </td>
          <td> <%= group.description %> </td>
          <td> <%= group.posts.count %> </td>
          <td> <%= group.updated_at %> </td>
        </tr>
      <% end %>
    </tbody>
  </table>
</div>
```



使用可可以在後台看到自己發表的 **post**

打開 **app/controllers/account/posts_controller.rb**

原本

app/controllers/account/posts_controller.rb

```
class Account::PostsController < ApplicationController
end
```

改成

app/controllers/account/posts_controller.rb

```
class Account::PostsController < ApplicationController

  before_action :authenticate_user!

  def index
    @posts = current_user.posts
  end
end
```

打開 **app/views/account/posts/index.html.erb**

原本

app/views/account/posts/index.html.erb

```
(空的)
```

改成

app/views/account/posts/index.html.erb

```
<div class="col-md-12">

 <h2 class="text-center"> 我發表過的文章 </h2>

 <table class="table">
  <thead>
   <tr>
    <th> Content </th>
    <th> Group Name </th>
    <th> Last Update </th>
    <th colspan="2"></th>
   </tr>
  </thead>

  <tbody>
   <% @posts.each do |post| %>
    <tr>
     <td> <%= post.content %> </td>
     <td> <%= post.group.title %> </td>
     <td> <%= post.updated_at %> </td>
     <td> <%= link_to('Edit',
                edit_group_post_path(post.group, post),
                :class => "btn btn-info btn-xs") %></td>
     <td> <%= link_to('Delete',
                group_post_path(post.group, post),
                :method => :delete,
                data: { confirm: "Are you sure?" },
                :class => "btn btn-danger btn-xs") %></td>
    </tr>
   <% end %>
  </tbody>
 </table>
</div>
```

post 的排列要以時間 (最新 => 最舊) 順序排列

我們會發現， account/posts 裡的文章是從最舊 => 最新 的順序排列的 我們應該要把最新的放在最上面才對

運作邏輯是 => 修改時間 (updated_at) 用 倒序排列 (DESC)

打開 **app/controllers/account/posts_controller.rb**

原本

app/controllers/account/posts_controller.rb

```
class Account::PostsController < ApplicationController
...
...
  def index
    @posts = current_user.posts
  end
end
```

改成

app/controllers/account/posts_controller.rb

```
class Account::PostsController < ApplicationController
...
...
  def index
    @posts = current_user.posts.order("updated_at DESC")
  end
end
```

before

after

group 的排列要以 文章數量 當熱門度排列

我們在 app/views/account/groups/index.html.erb 裡面有一行

```
<td> <%= group.posts.count %> </td>
```

用來呈現該 group 裡的 post 篇數 但這行有個很大的問題就是 : 在程式的迴圈裡面跑 SQL 搜尋 如果我加入了二個 group, 實際上運作時會執行二次

```
(0.2ms) SELECT COUNT(*) FROM `posts` WHERE `posts`.`group_id` = 1
(0.2ms) SELECT COUNT(*) FROM `posts` WHERE `posts`.`group_id` = 2
```

如果我加入了 20 甚至 100 個 group 的話 ... 會是個很噁心的程式 造成網站 server 的負擔

解決方案就是，在 group 裡面加入 post_count 欄位來記錄 post 的篇數

`rails g migration add_posts_count_to_group posts_count:integer`

打開 db/migrate/(一堆數字)_add_posts_count_to_group.rb

原本

db/migrate/(一堆數字)_add_posts_count_to_group.rb

```
class AddPostsCountToGroup < ActiveRecord::Migration
  def change
    add_column :groups, :posts_count, :integer
  end
end
```

改成

db/migrate/(一堆數字)_add_posts_count_to_group.rb

```
class AddPostsCountToGroup < ActiveRecord::Migration
  def change
    add_column :groups, :posts_count, :integer, :default => 0
  end
end
```

`rake db:migrate` 這樣 group 這個資料表就多了 posts_count 這個欄位

Rails 的 Model 裡面有一個內建的 counter_cache 功能幫助你記錄 只要設定好 counter_cache: :posts_count <== (你剛剛建立的欄位名稱) 只要 post 有 create 跟 destroy 的動作，就會自動在 posts_count 欄位 +1 跟 -1

打開 **app/models/post.rb**

原本

app/models/post.rb

```
...
...
  belongs_to :group
...
...
```

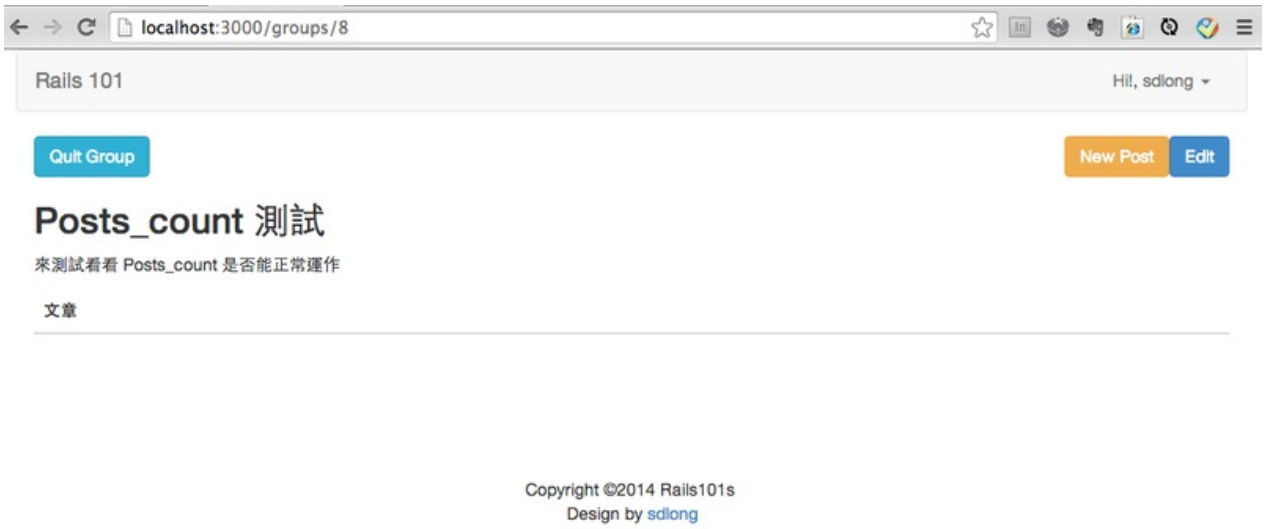
改成

app/models/post.rb

```
...
...
  belongs_to :group, counter_cache: :posts_count
...
...
```

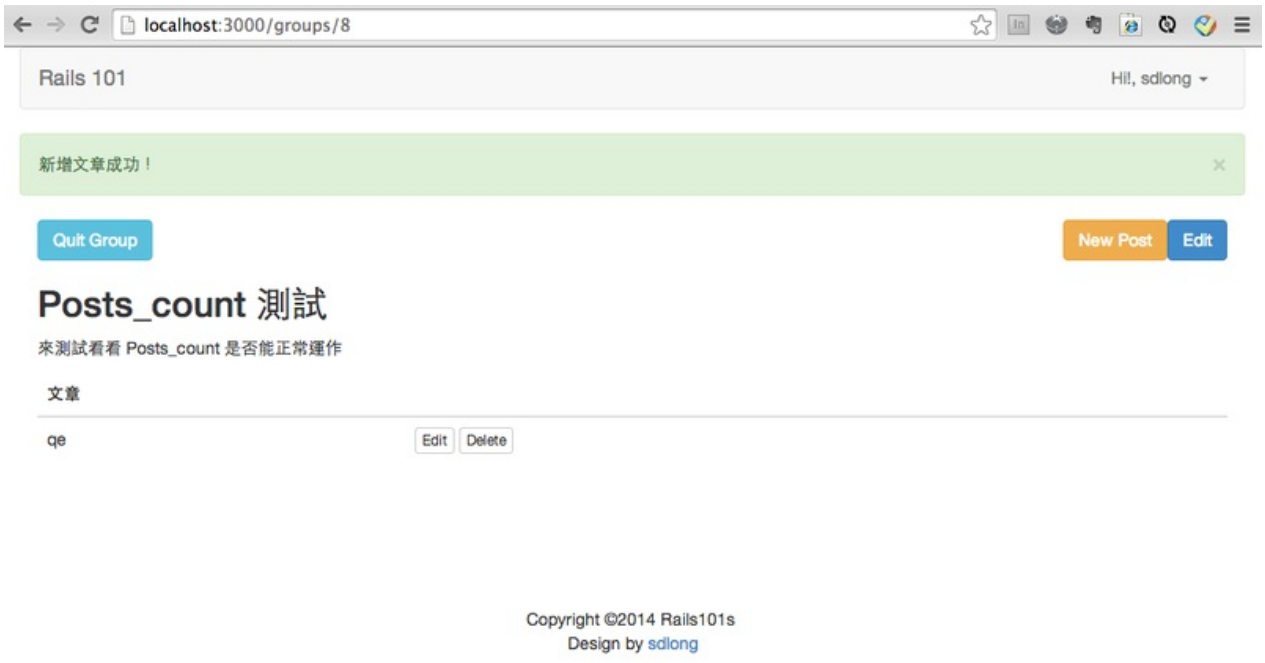
功能測試

原始: Rails Console 初始畫面



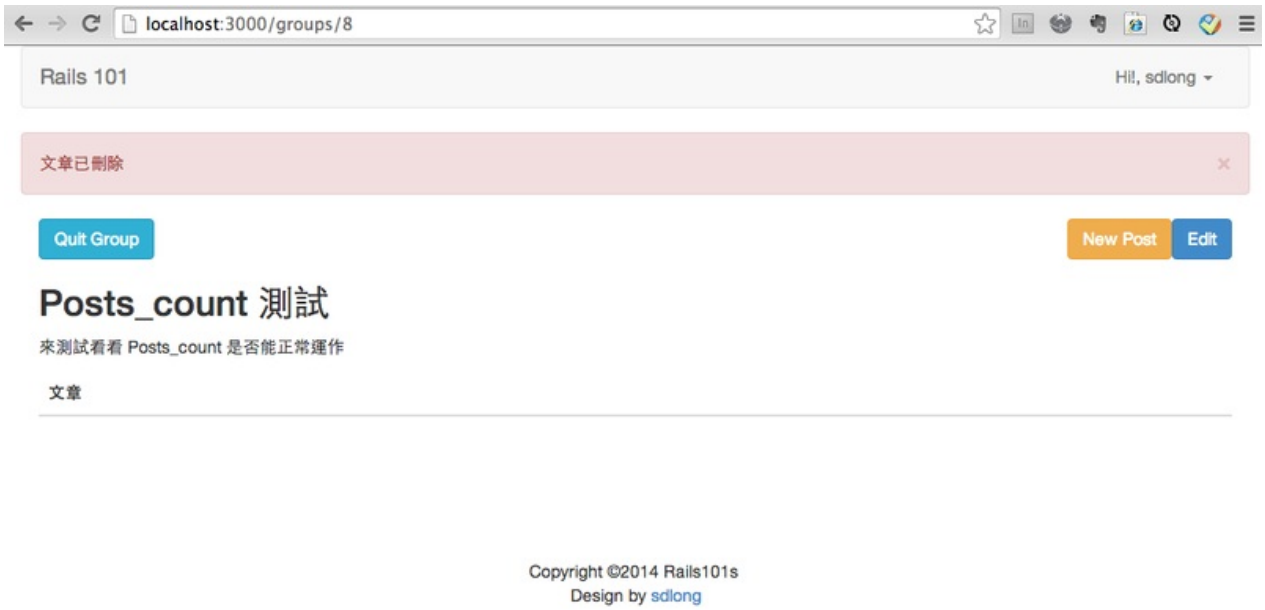

```
sdlong@sdlongtekiiMac ~/Dropbox/rails_projects/group101 μ master± rails c
Loading development environment (Rails 4.1.1)
2.0.0-p353 :001 > a = Group.find(8)
  Group Load (0.1ms)  SELECT  "groups".* FROM "groups"  WHERE "groups"."id" = ? LIMIT 1  [["id", 8]]
 => #<Group id: 8, title: "Posts_count 測試", description: "來測試看看 Posts_count 是否能正常運作", created_at: "2014-07-2
2 04:00:50", updated_at: "2014-07-22 04:00:50", user_id: 1, posts_count: 0>
```

測試: 新增文章




```
2.0.0-p353 :002 > reload!
Reloading...
 => true
2.0.0-p353 :003 > a = Group.find(8)
  Group Load (0.1ms)  SELECT  "groups".* FROM "groups"  WHERE "groups"."id" = ? LIMIT 1  [["id", 8]]
 => #<Group id: 8, title: "Posts_count 測試", description: "來測試看看 Posts_count 是否能正常運作", created_at: "2014-07-2
2 04:00:50", updated_at: "2014-07-22 04:00:50", user_id: 1, posts_count: 1>
```

測試: 刪除文章




```
2.0.0-p353 :004 > reload!
Reloading...
 => true
2.0.0-p353 :005 > a = Group.find(8)
  Group Load (0.1ms)  SELECT  "groups".* FROM "groups"  WHERE "groups"."id" = ? LIMIT 1  [["id", 8]]
 => #<Group id: 8, title: "Posts_count 測試", description: "來測試看看 Posts_count 是否能正常運作", created_at: "2014-07-2
2 04:00:50", updated_at: "2014-07-22 04:00:50", user_id: 1, posts_count: 0>
2.0.0-p353 :006 >
```

接下來把 account/groups 的 views 改一下，從用 SQL 一篇篇抓出數量改成直接抓 posts_count 的值

打開 app/views/account/groups/index.html.erb

原本

app/views/account/groups/index.html.erb

```
...
...
      <td> <%= group.posts.count %> </td>
...
...
```

改成

app/views/account/groups/index.html.erb

```
...
...
      <td> <%= group.posts.size %> </td>
...
...
```

---

bug 解決 => 當 group 刪除時，所屬的 posts 也要跟著刪除

當我們把 group 刪除掉的時候， 若所屬的 post 資料還在，就會變成所謂的孤兒資料

這會讓我們在打開 account/posts 頁面的時候跑出 error => 找不到該 post 所屬的 group (因為被刪了)

所以我們要設定當 group 刪除的時候，他所屬的 posts 也要跟著刪除

原本

app/model/group.rb

```
...
...
  has_many :posts
...
...
```

改成

app/model/group.rb

```
...
...
  has_many :posts, :dependent => :destroy
...
...
```

最後別忘了 git 存檔

`git add .`

`git commit -m "實做簡單的後台機制“`

我們已經完成了一個基本的專案了！

但是程式碼也越來越肥大

本章將簡單介紹 Rails 裡面很實用的 Refactor Code 的方法

Refactor Code 整理你的程式碼

原文網址

本章的作業目標:

- 使用系統 helper 整理 code
- 自己撰寫的 helper 包裝 html
- 使用 partial 整理 html
- 使用 scope 整理 query

使用系統 helper 整理 code

時間格式

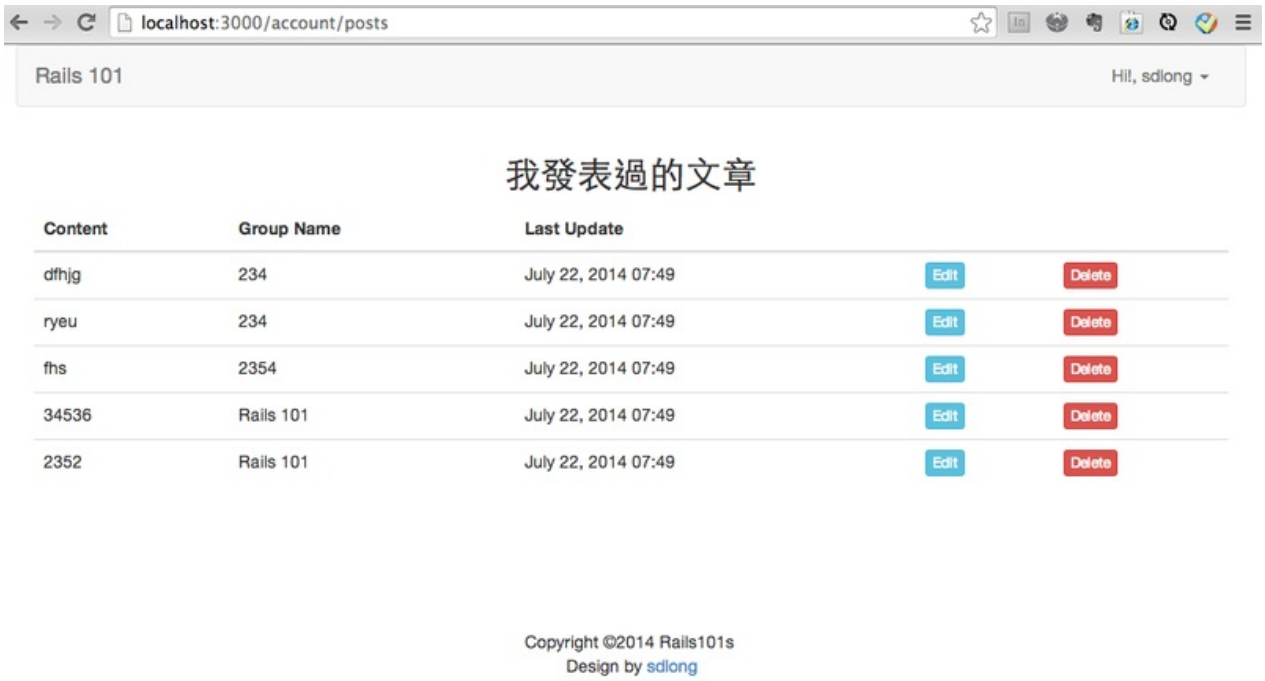
我們會發現在時間格式上是顯示

2014-07-19 08:02:17 UTC

我們可以改用這樣的寫法:

app/views/account/posts/index.html.erb

```
...
...
        <td> <%= post.updated_at.to_s(:long) %> </td>
...
...
```



or

app/views/account/posts/index.html.erb

```
...
...
        <td> <%= post.updated_at.to_s(:short) %> </td>
...
...
```

內文自動斷行

要讓 post 的 content 能自動斷行

原本

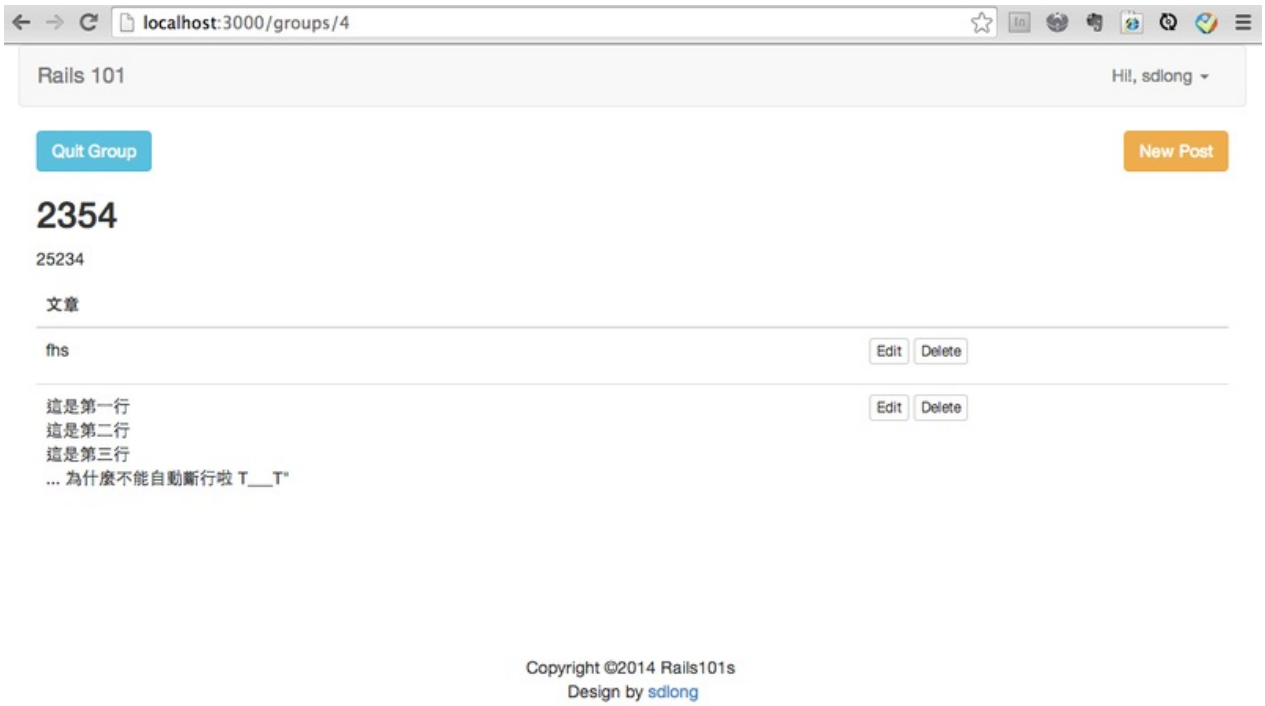
app/views/groups/show.html.erb

```
...
...
        <td> <%= post.content %> </td>
...
...
```

改成

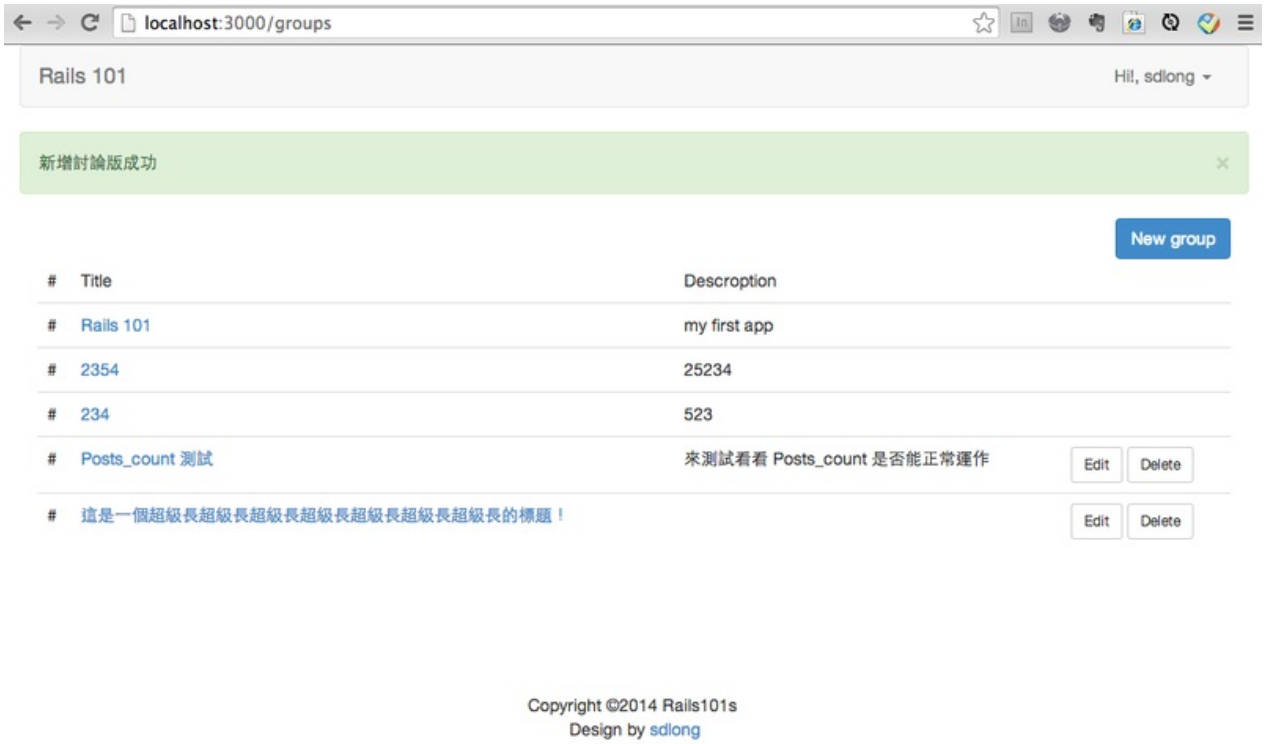
app/views/groups/show.html.erb

```
...
...
        <td> <%= simple_format(post.content) %> </td>
...
...
```



如果後台的 account/post 也要改的話， 一樣打開 app/views/account/posts/index.html.erb 把上述那行改掉

標題文字太長就...



原本

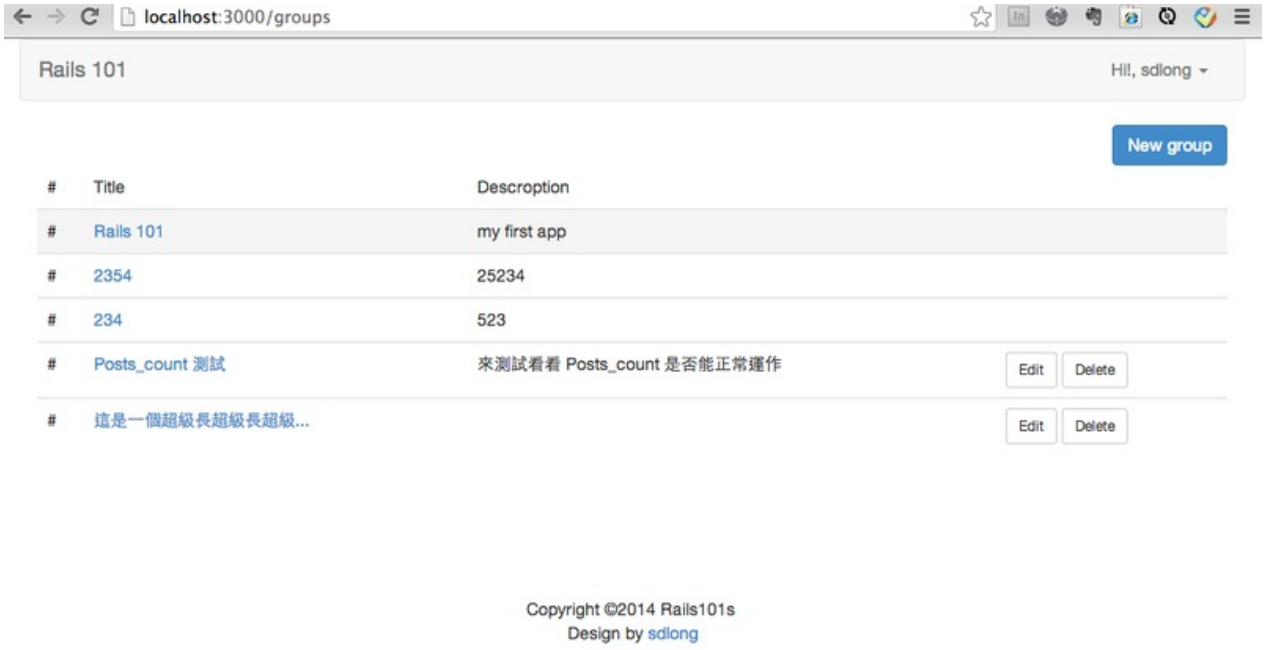
app/views/groups/index.html.erb

```
...
...
    <td>  <%= link_to(group.title, group_path(group)) %> </td>
...
...
```

改成

app/views/groups/index.html.erb

```
...
...
    <td>  <%= link_to(truncate(group.title, :length => 15 ), group_path(group)) %> </td>
...
...
```



自己撰寫的 **helper** 包裝 **html**

在剛剛的專案當中，顯示Post 的程式碼如下：

```
<%= @post.content %> %>
```

隨著專案變遷，這樣的程式碼，可能會依需求改成：（需要內容斷行）

```
 <%= simple_format(@post.content) %> %>
```

之後又改成（只顯示頭一百字）

```
<%= truncate(simple_format(@post.content), :lenth => 100) %>
```

而麻煩的是，這樣類似的內容，常常在專案出現。每當需求變更，開發者就需要去找出來， 有十個地方，就需要改十遍，很是麻煩。

Helper 就是用在這樣的地方

一開始就設計一個 Helper

```
<%= render_post_content(@post) %>  (這是放在 views 裡面的)
```

然後打開 app/posts_helper.rb

app/posts_helper.rb

```
def render_post_content(post)
  truncate(simple_format(post.content), :lenth => 100)
end
```

以後對 @post.content 有任何的異動，只要改 helper 裡的東西就好 不用再到各個有放 @post.content 的檔案一一修改

helper 是全域的變數，即使寫在 posts_helper.rb 裡面 在其他地方像是 groups / account/groups / account/posts 都能呼叫得出來

還有一個好處就是把 view 單純化，把所有運作邏輯全部轉到 helper 裡面 在未來需要維護的時候，會一整個非常乾淨好讀

使用 **partial** 整理 **html**

其實我們的 app/views/layout/application.html.erb 就用的 Partial 的用法

把 navbar 跟 footer 拆分

讓檔案看起來乾淨清爽

使用前

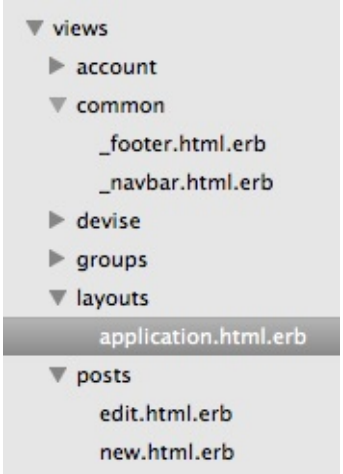
app/views/layout/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Group101</title>
  <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<div class="container-fluid">
  <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
      <!-- Brand and toggle get grouped for better mobile display -->
      <div class="navbar-header">
        <a class="navbar-brand" href="/">Rails 101</a>
      </div>

      <!-- Collect the nav links, forms, and other content for toggling -->
      <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav navbar-right">
          <% if !current_user %>
          <li> <%= link_to("註冊", new_user_registration_path) %> </li>
          <li> <%= link_to("登入", new_user_session_path) %> </li>
          <% else %>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Hi!, <%= current_user.name %>
              <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li> <%= link_to("My Groups", account_groups_path) %></li>
              <li> <%= link_to("My Posts", account_posts_path) %></li>
              <li> <%= link_to("帳號設定", edit_user_registration_path )%></li>
              <li> <%= link_to("登出", destroy_user_session_path,
                          :method => :delete ) %></li>
            </ul>
          </li>
          <% end %>
        </ul>
      </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
  </nav>

  <%= notice_message  %>

  <%= yield %>
</div>

<footer class="container" style="margin-top: 100px;">
  <p class="text-center">Copyright ©2014 Rails101s
    <br>Design by <a href="http://sdlong.logdown.com" target=_new>sdlong</a>
  </p>
</footer>

</body>
</html>
```

使用後

app/views/layout/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Group101</title>
  <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<div class="container-fluid">
  <%= render "common/navbar" %>
  <%= notice_message  %>
  <%= yield %>
</div>
  <%= render "common/footer" %>

</body>
</html>
```



使用 **scope** 整理 **query**

在 6 - 0. 我們在設計了一行程式碼， 讓文章永遠按照最新的時間降序排列

app/controllers/account/posts_controller.rb

```
...
...
  def index
    @posts = current_user.posts.order("updated_at DESC")
  end
...
...
```

```
.order("updated_at DESC")
```

其實是一段可能會很常用到的程式碼 既然能在 View 裡面用 helper 跟 partial 整理 在 controller 當然也能用 scope 整理

打開 app/models/post.rb

原本

app/models/post.rb

```
class Post < ActiveRecord::Base
...
...
```

改成

app/models/post.rb

```
class Post < ActiveRecord::Base

  scope :recent, -> { order("updated_at DESC") }
...
...
```

有需要用 最新 -> 最舊 排序的 controller 就直接改放 .recent 即可

app/controllers/account/posts_controller.rb

```
...
...
  def index
    @posts = current_user.posts.recent
  end
...
...
```

最後別忘了 git 存檔

`git add .`

`git commit -m "Refactor Code"`

撰寫 db:seed 與 自動化重整資料庫程式

原文網址

本章的作業目標:

- rake 的概念解說
- 撰寫 seed.rb 檔
- 撰寫 rake dev:build => 自動化執行清空+重跑 migration +種子載入資料

---

rake 的概念解說

rake 直譯上來說就是 Ruby Make， 用 ruby 語言開發的 build tool

也可以說是一套 task 管理工具，幫助我們自動化執行所需要的指令

`rake -T` 會列出所有目前能執行的 rake 指令 (包含 install gem 以後建置在內的)

像我們之前常用的 `rake db:migrate` 就是執行 資料庫異動設定 的 task

撰寫 seed.rb 檔

我們在開發專案的時候，常會需要弄些假資料來填充版面 或是說當資料庫炸掉的時候，會很想一股腦把 資料庫整個 reset 掉 並希望 reset 完還能把一些資料放進去，像是 user, groups, posts，不用一個個手動輸入

我們來做一個 seed 檔，需求:

1. 建立一個 user 帳號 => example@gmail.com , 密碼 12345678
2. 並用此帳號建立 20 個 groups
3. 每個 group 各有 30 個 posts

打開 db/seeds.rb

原本

db/seeds.rb

```
(空的)
```

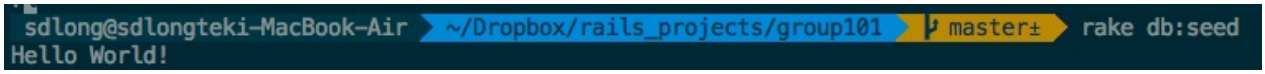
改成

db/seeds.rb

```
puts "Hello World!"
```

`rake db:seed`

執行 db:seed



原本

db/seeds.rb

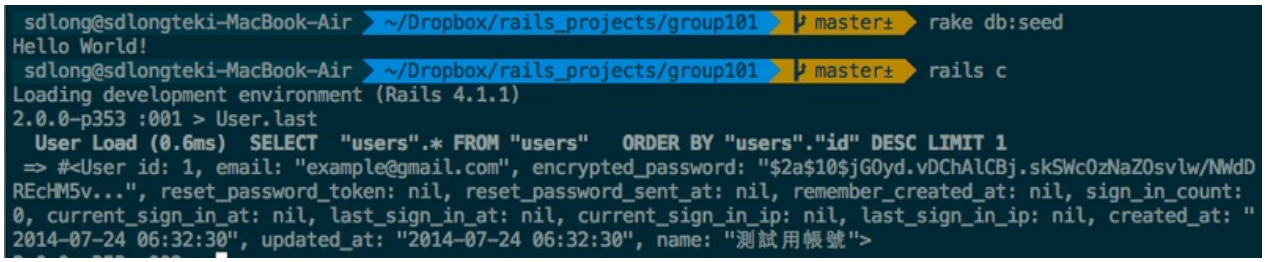
```
puts "Hello World!"
```

改成

db/seeds.rb

```
puts "Hello World!"

create_account = User.create([email: 'example@gmail.com', password: '12345678', password_confirmati
```

rake db:seed



原本

db/seeds.rb

```
puts "Hello World!"

create_account = User.create([email: 'example@gmail.com', password: '12345678', password_confirmati
```

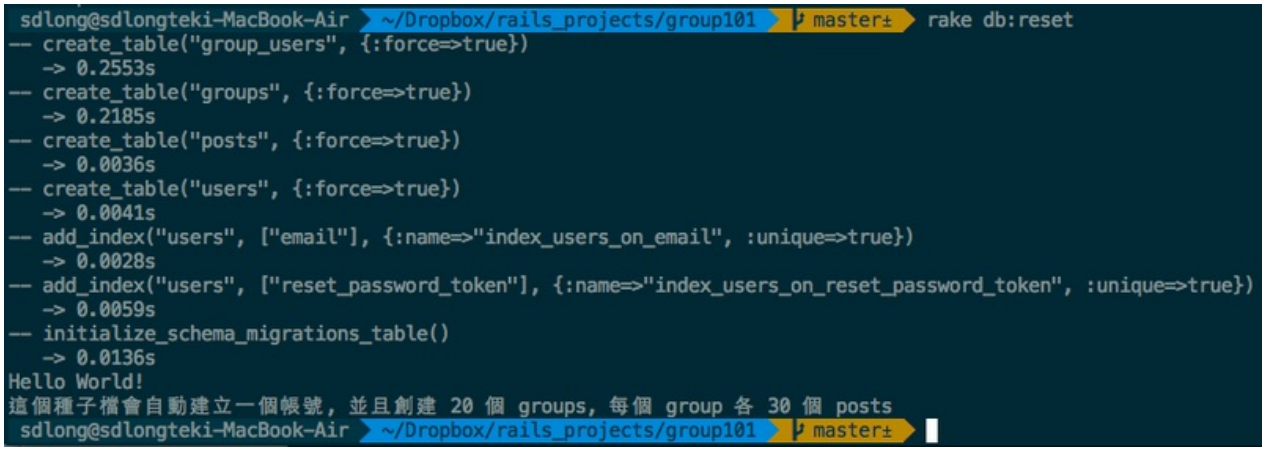
改成 db/seeds.rb

```
puts 'Hello World!'
puts "這個種子檔會自動建立一個帳號, 並且創建 20 個 groups, 每個 group 各 30 個 posts"

create_account = User.create([email: 'example@gmail.com', password: '12345678', password_confirmati

create_groups = for i in 1..20 do
              Group.create!([title: "Group no.#{i}", description: "這是用種子建立的第 #{i} 個討論版", user_id:
              for k in 1..30 do
                Post.create!([group_id: "#{i}",content: "這是用種子建立的第 #{k} 個留言", user_id: "1"])
              end
            end
```

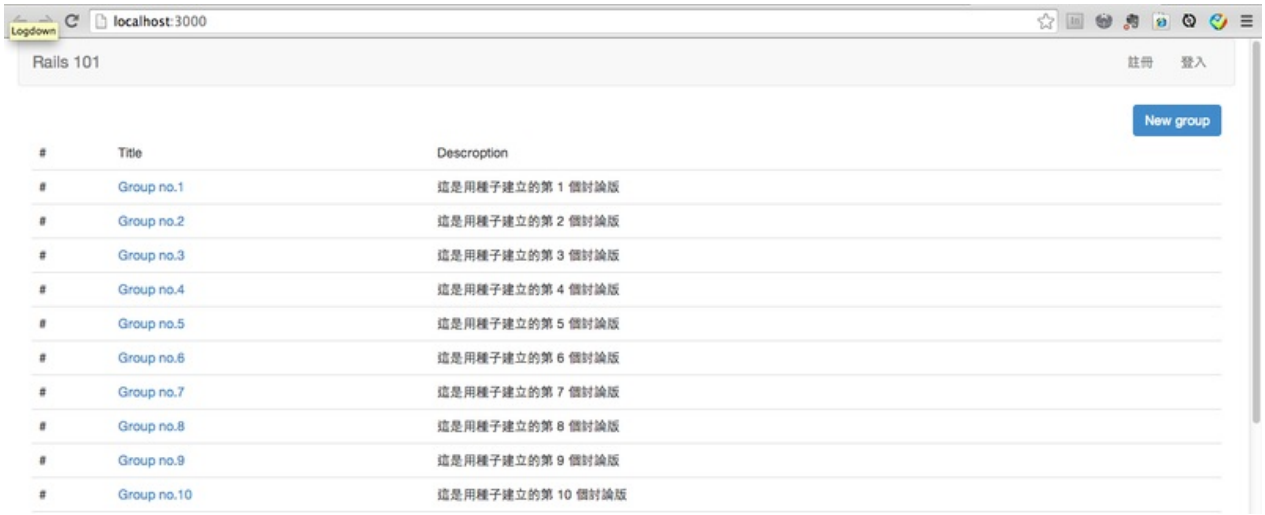
```
rake db:reset
```

rake db:reset 是內建的 task， 會執行一整套流程

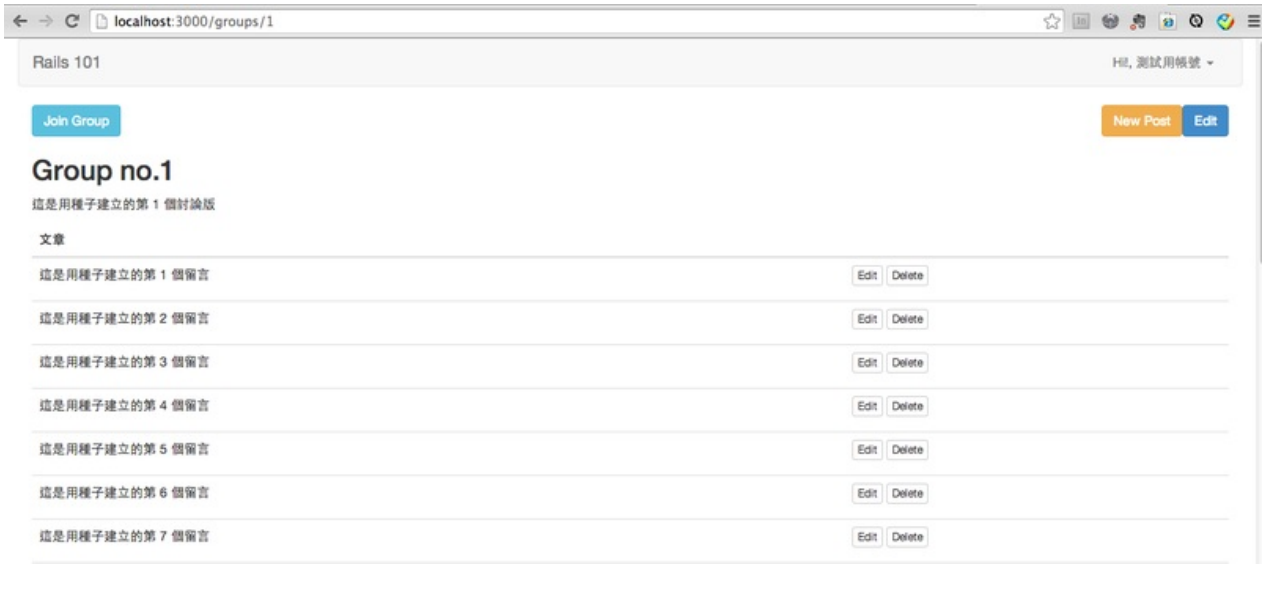
db:drop(資料庫移除) => db:create(資料庫建立) => db:schema:load (資料庫欄位建立) => db:seed (執行資料庫種子)



重新啟動 rails s



用帳號: example@gmail.com 密碼 12345678 登入，並進入隨便一個討論版



撰寫 **rake dev:build =>** 自動化執行清空+重跑 **migration +**種子載入資料

雖然 db:reset 已經能自動地幫我們把資料庫清空、重建、跑 seed 如果想把整個開發中專案的資料一切歸零，還需要跑 `rake log:clear` 跟 `rake tmp:clear` 如果懶得一個個打上述指令 + `rake db:drop` + `rake db:create` + `rake db:migrate` + `rake db:seed` 來把整個專案清空重建並建立 seed 檔 可以寫一個 rake 來自動化執行

在 lib/tasks/ 資料夾建立一個檔案 dev.rake

原本

lib/tasks/dev.rake

```
(空的)
```

改成

lib/tasks/dev.rake

```
namespace :dev do
  desc "Rebuild system"
  task :build => ["tmp:clear", "log:clear", "db:drop", "db:create", "db:migrate"]
  task :rebuild => [ "dev:build", "db:seed" ]
end
```

這樣只要打

`rake dev:build` 即可把專案資料一切清空歸零重建

`rake dev:rebuild` 即可重建完再跑 seed

[延伸閱讀] Strong Parameters

原文網址

2012 年 Github 會經發生轟動一時的 被 hack 事件 原本只是一個好心的 hacker 發現漏洞回報問題 因為沒人理他(!?) 所以乾脆直接 hack 給大家看，來證明此問題的嚴重性 XDDD

關於整事情的前因後果跟安全機制的演化過程可以參考 原著 Rails 101 的 p.52 這邊簡單地講解決方法

簡單來說呢，在 Rails 的架構下，表單的欄位內容是綁 Model 資料庫的欄位 例如當我們 create 一筆 group 資料，如果用 HTTP Request 的訊息來看，其實是這樣

```
# 準備上傳一組 Hash (雜湊) 內容如下:

Started POST "/groups" for 127.0.0.1 at 2014-07-24 16:57:19 +0800
# HTTP verb: post , 網址: "/groups"

Processing by GroupsController#create as HTML
# 執行 group_controller.rb 的 Create action

  Parameters: {"utf8"=>"✓",
              "authenticity_token"=>"un1ahUt88HhFVxLcopkQojkdD6PaoPfPUjAQ8YNFN+0=",
              "group"=>{"title"=>"new group",
                        "description"=>"bala bala ...."},
              "commit"=>"Submit"}
# 送到 server 端的參數: ..... (bj4)
```

然後 Rails 很聰明地幫我們跟資料庫溝通，執行以下動作

```
  User Load (0.4ms)  SELECT  "users".* FROM "users"  WHERE "users"."id" = 1  ORDER BY "users"."id" AS
# 讀取 Current_user (登入的使用者) 資料

   (0.2ms)  begin transaction

  SQL (0.8ms)  INSERT INTO "groups" ("created_at",
                                     "description",
                                     "title",
                                     "updated_at",
                                     "user_id")
              VALUES (?, ?, ?, ?, ?)  [["created_at", "2014-07-24 08:57:19.933822"],
                                       ["description", "bala bala ...."],
                                       ["title", "new group"],
                                       ["updated_at", "2014-07-24 08:57:19.933822"],
                                       ["user_id", 1]]
# 用 SQL 語法真正把資料存進資料庫裡面, 有 create_at, title, description, updated_at, user_id 等欄位被存入資料

(1.0ms)  commit transaction
   (0.2ms)  begin transaction

  SQL (64.5ms)  INSERT INTO "group_users" ("created_at",
                                           "group_id",
                                           "updated_at",
                                           "user_id")
                  VALUES (?, ?, ?, ?)  [["created_at", "2014-07-24 08:57:20.157364"],
                                        ["group_id", 21],
                                        ["updated_at", "2014-07-24 08:57:20.157364"],
                                        ["user_id", 1]]
# 開始執行 create action 裡面寫的 if @group.save => current_user.join!(@group) (自動加入成為該 group 成員

   (8.6ms)  commit transaction

Redirected to http://localhost:3000/groups
# 開始執行 create action 裡面寫的 if @group.save =>  redirect_to groups_path (回到 groups#index 頁面)

Completed 302 Found in 395ms (ActiveRecord: 76.3ms)
```

我們就會在瀏覽器出這個畫面

開始執行 *create action* 裡面寫的 *if @group.save => (,:notice => '新增討論版成功')*

這是一個理想的狀態， but

在早期沒對 Hash 做安全機制以前，很有可能會在 Hash 裡面動手腳 => 例如 user_id 明明是 2, 卻改成身為 admin 的 1 => 來做進一步的 hack 或是其他 SQL 指令 => 把你資料庫全部清空

現在的解決方案是這樣： 在 Controller 裡面設定 strong_params 也就是利用這個設定(以 groups_controller 舉例)

```
def group_params
   params.require(:group).permit(:title, :description)
end
```

整串 Hash 參數 (params) 只允許 (.permit) group 資料表 (:group) 的 :title 跟 :descrition 這二個欄位的資料通過， 進入下一個階段的 SQL 資料庫存取，其餘的參數一律忽略

用此方式來做到資訊安全機制

所以我們在 4 - 1. 使用者功能新增 name 資料 與 帳號設定 功能 裡面

當增加了 name 的欄位，就必須要在 devise 預設的 strong_params => configure_permitted_parameters

```
def configure_permitted_parameters
 devise_parameter_sanitizer.for(:sign_up) { |u| u.permit(:name, :email, :password, :password_confirmati
end
```

把 :name 加進去，這樣在表單建立資料的時候， name 欄位的資料才會被允許送進資料庫存取裡面

然後在修改 (update) 表單的頁面還要加入

```
devise_parameter_sanitizer.for(:account_update) { |u| u.permit(:name, :email, :password, :password_co
```

name 跟 current_password (使用者賬戶密碼確認)，才能通過，允許資料修改

[延伸閱讀] Active Record Association

原文網址

本章將淺談： 從本 blog 裡用到的 Database Relationship (資料庫關聯) 為例 解說 Rails 裡的 Active Record Association

- 基本概念
- 延伸運用
- 多對多關聯

基本概念

我們在 3 - 0. 可以在討論版裡發表文章 (前置設定) 裡有做到

app/models/group.rb

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

app/models/group.rb

```
class Post < ActiveRecord::Base
  belongs_to :group
end
```

來把 group 跟 post 這二個資料建立關聯

二者之間的關係圖如下：

在 Rails 的架構裡面，只要『 被屬於 』(本例是 post) 的資料表 有 『 擁有者_id 』(本例是 group_id) 的欄位 即可成為二資料表間關聯性的『 索引 』

舉例來說

假設 post 的資料表內容如下

資料表: Post		
id	content	group_id
1	a	1
2	b	2
3	c	2
4	a	1
5	b	1
6	c	3
7	a	3
8	b	1
9	c	3
10	a	2
11	b	2
12	c	1
13	a	1

當我們在 controller or rails console 模式輸入

`group.find(1).posts.all`

意思等於 => 列出 id 是 1 的 group 裡所擁有的全部 posts => 找出 group_id 是 1 的 posts

就像這樣

資料表: Post		
id	content	group_id
1	a	1
4	a	1
5	b	1
8	b	1
12	c	1
13	a	1

這時候再回頭看 groups_controller.rb 裡的 show action

app/controllers/groups_controller.rb

```
def show
  @group = Group.find(params[:id])
  @posts = @group.posts
end
```

以上就是 Active Record Association 的基本概念

延伸運用

我們在 4 - 2. 使用者功能帶進 group 與 post 裡 (作者機制) 有做:

app/models/user.rb

```
class User < ActiveRecord::Base

  has_many :groups
  has_many :posts

end
```

app/models/group.rb

```
class Group < ActiveRecord::Base

  belongs_to :owner, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == owner
  end
end
```

app/models/post.rb

```
class Post < ActiveRecord::Base
...
...
  belongs_to :author, :class_name => "User", :foreign_key => :user_id

  def editable_by?(user)
    user && user == author
  end
end
```

在本例裡面，其實 group 跟 post 都是 belongs_to :user 我們可以去客製化，讓

```
User.find(params[:id]).groups => User.find(params[:id]).owners
User.find(params[:id]).posts  => User.find(params[:id]).authors

Group.find(params[:id]).user  => Group.find(params[:id]).owner  # group 的擁有者 id
Post.find(params[:id]).user   => Post.find(params[:id]).author # post 的作者 id
```

只要先設定好自定的變數，然後定義好對應的資料表 (:class_name => "User"), 以及 『 索引 』 (:foreign_key => :user_id) 即可

多對多關聯

我們在 5 - 0. user 可以加入、退出 group 裡面做出 users 跟 groups 的多對多關聯

它的原理概念是這樣：

藉由一個中介資料表 (group_user) 來做到多對多關聯

app/models/user.rb

```
  has_many :group_users
  has_many :participated_groups, :through => :group_users, :source => :group
end
```

app/models/group_user.rb

```
class GroupUser < ActiveRecord::Base
  belongs_to :user
  belongs_to :group
end
```

app/models/group.rb

```
class Group < ActiveRecord::Base

  has_many :group_users
  has_many :members, :through => :group_users, :source => :user
```

其中

```
has_many :members,             :through => :group_users, :source => :user
has_many :participated_groups, :through => :group_users, :source => :group
```

代表此變數 (memebers, participated_groups) 是經由 (through) group_users 這個資料表 來取得 (user , group) 資料

範例

A	B	C
資料表: group_user		
id	user_id	group_id
1	1	4
2	2	3
3	1	1
4	2	2
5	1	1
6	1	3
7	3	2
8	4	2
9	3	4
10	3	1
11	2	3
12	1	2
13	1	3
14	2	4
15	3	2

如果

`User.find(1).participated_groups.all` <=== 我們的使用者後台列出 使用者所加入的 group 就是用這原理做成的

等同於

A	B	C
資料表: group_user		
id	user_id	group_id
1	1	4
3	1	1
5	1	1
6	1	3
12	1	2
13	1	3

如果

`Group.find(1).members.all` <=== 可以用此概念來列出該 group 所有成員

等同於

A	B	C
資料表: group_user		
id	user_id	group_id
3	1	1
5	1	1
10	3	1