Hello Rails

CITS3403 Web & Internet Technologies
Installing Rails

• Available for MacOSX, Linux, Windows
• Easy to install
  – uses *RubyGems* packaging system
• See *Ruby et al.*, Chapter 3
Creating a New Application

- Rails does the groundwork for you
  - sets up default directory structure
  - everything where it expects to see it (“convention over configuration”)
  - templates or skeletons for standard files
    - *nothing magic* - just Ruby code (you could do the same from scratch... given lots of time...)

rubys> cd work
work> rails new demo
create
create app/controllers
create app/helpers create app/models . . .
create log/test.log work>
Creating a New Application

• File structure...
  work> cd demo
demo> ls -p
  README    config/  lib/  script/
  vendor/  Rakefile  db/
  log/      test/
  app/      doc/     public/  tmp/

• Start with script and app

• script contains, you guessed it, scripts!
- Rails packaged with two built-in web servers
  - mongrel
  - WEBrick
- Started by the server script

12345678@cits3403:/home/cits3403/cits3403y/demo/script> rails server -p3050
=> Booting WEBrick
=> Rails 3.0.10 application starting in development on http://0.0.0.0:3050
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2011-10-04 11:19:22] INFO  WEBrick 1.3.1
[2011-10-04 11:19:22] INFO  WEBrick::HTTPServer#start: pid=11062 port=3050
Welcome Aboard

- Listens on port 3050...
Run from native server

- Link the **public** folder of your project into the `cits3403?/public_html` folder via the command
  
  ```bash
  ln -s ../demo/public demo
  ```

  (assuming the demo project directory is at the same level as `public_html` and you are inside the `public_html` directory)

- Edit the `.htaccess` file within **public_html** to include the line:
  
  ```apache
  RailsBaseURI /project_name
  ```

Then you can access your project **demo** using:

Hello, Rails! Our First Rails App

• Rails manifestation...
  – *requests-response cycle*
  – *model-view-controller* architecture
    • A model is more than just data; it acts as both a gatekeeper and a data store
    • The *view* is responsible for generating a user interface, normally based on data in the model.
    • *Controllers* orchestrate the application. Controllers receive events from the outside world (normally user input), interact with the model, and display an appropriate view to the user.
MVC and Request-Response Cycle

Figure 4 — The Model-View-Controller architecture

Figure 5 — Rails and MVC
Hello, Rails! Our First Rails App

- Hello, World! app has no data → no model
  - only need controller and view
- Simple case
  - request from browser
  - Rails decodes request to find appropriate controller
  - action method in controller, invokes a view
  - passed back to browser to display to user
  ➡ Rails takes care of “internal plumbing”
Controller

- `generate` script creates controllers
  - to generate a controller called “Say”...

  demo> rails generate controller say hello
  exists app/controllers/
  exists app/helpers/
  create app/views/say
  exists test/functional/
  create app/controllers/say_controller.rb
  create test/functional/say_controller_test.rb
  create app/helpers/say_helper.rb
Controller

class SayController < ApplicationController
end

http://pragprog.com/say/hello

1. First part of URL addresses the application
2. Then the controller [say]
3. And the action [hello]

Figure 4.2: URLs Are Mapped to Controllers and Actions
Controller

http://pragprog.com/say/hello

Create an instance of SayController

and invoke the action method hello

class SayController
  def hello
    # code for hello action
  end
end

class UserController

class SayController

class ProductController

class LoginController
  def login
    # code ...
  end
end
end

Figure 4.3: Rails Routes to Controllers and Actions
Controller

- Controller’s job to set things up so view knows what information (data, calculations, etc) to display
  - in this case, nothing to set up

```ruby
class SayController < ApplicationController
  def hello
    end
  end
end
```
View

- Recall `generate` also created a directory `app/views/say`
  - will contain views for controller `Say`
  - default: same name as controller, with extension `.html.erb`

- `app/views/say/hello.html.erb`

```html
<html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
  </body>
</html>
```
Structure So Far

Figure 4.4: Standard Locations for Controllers and Views
Dynamic Content

- Of course the power of server-side programming comes from being able to add *dynamically* generated content.

- Two ways in RoR
  - *builder templates*
  - *embedded Ruby code*
  - we will focus on latter

- `.erb` files preprocessed for embedded Ruby

- **ERb (embedded Ruby) filter**
  - content between `<%= .... %>` interpreted as Ruby code and executed
    - result converted to string and substituted
  - content between `<% .... %>` interpreted but not substituted
Embedded Ruby

• Eg.

```<ul>
  <li>Addition: <%= 1+2 %> </li>
  <li>Concatenation: <%= "cow" + "boy" %> </li>
  <li>Time in one hour: <%= 1.hour.from_now %> </li>
</ul>```

• Result

- Addition: 3
- Concatenation: cowboy
- Time in one hour: Tue Sep 29 11:30:32 -0400 2009
Embedded Ruby

• Can be intermixed with non-Ruby code

    <% 3.times do %> Ho!<br /> <% end %> Merry Christmas!

    Ho!<br />
Ho!<br />
Ho!<br />

Merry Christmas!

– Note: there are newline characters in the loop that can be removed by using `<% ... -%>`
Embedded Ruby

- Substitution
  ```ruby
  <% 3.downto(1) do |count| -%> <%= count %>...<br />
  <% end -%>
  Lift off!
  3...<br /> 2...<br /> 1...<br /> Lift off!
  ```

- XHTML character substitution

  Email: <%= h("Ann & Bill <frazers@isp.email>") %>  

  Email: Ann & Bill &lt;frazers@isp.email&gt;
Embedding Results from Controller

- **say_controller.rb**
  ```ruby
  class SayController < ApplicationController
    def hello
      @time = Time.now
    end
  end
  ```

- **hello.html.erb**
  ```html
  <html>
  <head>
    <title>Hello, Rails!</title>
  </head>
  <body>
    <h1>Hello from Rails!</h1>
    <p>It is now <%= @time %></p>
  </body>
  </html>
  ```
Linking Pages Together

• Assume we had a second page, goodbye.html.erb

```html
<html>
<head>
  <title>See You Later!</title>
</head>
<body>
  <h1>Goodbye!</h1>
  <p>It was nice having you here.</p>
</body>
</html>
```

and our controller...

class SayController < ApplicationController
  def hello
    @time = Time.now
  end

  def goodbye
  end
end
Linking Pages Together

- We could link the pages using relative addresses:

```html
<p>
  Say <a href="/say/goodbye">Goodbye</a>!
</p>
```

- but its brittle
  - move application to different location in deployment on web server
  - assumptions about Rails URLs which might change
    - “hard wired”
  ➜ should always link to the *action that generates the output*, not the output
Linking Pages Together

• *helper method: link_to*

  – creates hyperlink via the *action* that generates the page

```html
<body>
  <h1>Hello from Rails!</h1>
  <p>It is now <%= @time %></p>
  <p>
    Time to say
    <%= link_to "Goodbye!", :action => "goodbye" %>
  </p>
</body>
```
Review

• We have seen:
  – how to create a Rails application
    • skeleton structure, controller, views
  – how Rails maps incoming requests to methods in code
  – how to create dynamic content in controller and display it via view template
  – how to link pages together