Ruby on Rails
Controllers, Views, and Functional Tests

CPS353 Internet Programming

Simon Miner

Gordon College

Last Modified: 10/16/2013

Agenda

- Scripture (Philippians 2) and Prayer
- Check-in
- Ruby on Rails
  - Migrations, Models, and Unit Tess
- More Ruby on Rails
  - Controllers, Views, and Functional Tests

Check-in

- Updates
  - Syllabus
  - Project schema changes
    - Peers are now friendships, have a modified_at field
    - “modified_at” should be “updated_at” in all models/tables
    - Restaurant delivery flag is not required (because an unset flag means that the restaurant does not deliver)
  - “--skip-bundle” switch for “rails new” command
- Homework 4
- Milestone 4
- Milestone 5 – How’s it going? Questions?

Ruby on Rails
Migrations, Models, and Unit Tests

Continued from last week
(starting on slide 23)

The Action Pack Modules

- Action Dispatch – routes requests to controllers
  - Covered later
- Action Controller – converts requests into responses
  - Classes reside in app/controllers
- Action View – formats responses
  - Used by Action Controller
  - Classes reside in app/views

Controller Default Actions

- index – listing page for records in a model
- show – view details of a single model record
- new – display a form to create a new model record
- create – submit data to create a new model record
- edit – load data on an existing model record into a form for editing
- update – submit data to update an existing model record
- destroy – delete a model record

Controller Environment Variables

- Controller object instance variables are available in the view
- Per-request controller variables
  - action_name – name of the current controller action
  - cookies – associative array of cookies sent with the response
  - headers – hash of headers to be sent with the response
  - params – associative array of parameters passed into the current request
  - session – associative array of session data that persists between requests
  - flash – associative array to make data (typically errors) available to the next request
  - logger – object to log errors and messages to a file on the server
  - request – incoming request object with methods like url(), remote_ip(), user_agent(), etc.
  - response – response object (usually handled by Rails)

Responding to the User

- Controller responds to the user by
  - Rendering a template (HTML, JSON, XML, etc.)
    - Uses render() method – many varieties
  - Send binary data to the user – send_data()
  - Send the contents of a file – send_file()
  - Redirect the user to a different view
    - redirect_to() method
    - Often done after a successful create/update/save operation to return the user to the default or listing page
    - Post-Redirect-Get (PRG) pattern

Callbacks

- Allow you to execute code before, after, or around a controller action
  - Before – i.e. to set up necessary data, authenticate a user, etc.
  - After – i.e. logging events, compressing the response if the client browser supports it
  - Around – wrap controller call in logic (which can even not invoke the controller at all)
    - yield call invokes controller
- By default, callbacks execute for all controller actions
  - :only – only fire callback for listed actions
  - :except – fire callback for all actions except those listed

Callback in Controller Scaffolding

- setup_<model> method defined to load the entity being worked with
  - Invoked for most default controller actions

```
class ProductsController < ApplicationController
  before_action :set_product, only: [:show, :edit, :update, :destroy]
...
 private
    # Use callbacks to share common setup or constraints between actions.
    def set_product
      @product = Product.find(params[:id])
    end
...
end
```

Callback Example

```
module CurrentCart extend ActiveSupport::Concern
  private
  def set_cart
    @cart = Cart.find(session[:cart_id])
  rescue ActiveRecord::RecordNotFound
    @cart = Cart.create
    session[:cart_id] = @cart.id
  end
end
...
class LineItemsController < ApplicationController
  include CurrentCart
  before_action :set_cart, only: [:create]
  before_action :set_line_item, only: [:show, :edit, :update, :destroy]
  # GET /line_items
  #...
end
```

Functional Tests

- Tests to exercise controller action behavior
  - Located in test/controllers/*name*_controller.rb
- Writing functional tests
  - Scaffolding produces default functional tests for generated controller actions
  - Need to modify these
    - To test invalid and acceptable data based on model validations
    - To test new controllers
- Can make use of fixtures
  - Sometimes the same fixtures employed by unit tests

Functional Test Example

```
require 'test_helper'
class StoreControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_select '#columns #side a', minimum: 4
    assert_select '#main .entry', 3
    assert_select 'h3', 'Programming Ruby 1.9’
    assert_select '.price', /\$[,\d]+\.\d\d/
  end
End
```

- assert_select checks for CSS selectors and/or HTML tags present in the content
  - Can look for CSS ids (#), classes (.), and HTML tag names
  - Selectors can be nested

More Assertions (Rails-specific)

- assert_difference
- assert_no_difference
- assert_recognizes
- assert_generates
- assert_response
- assert_redirected_to
- assert_template

Most of these are used for functional (controller/ view) tests.

View Templates and Controls

- Default templates
  - index.html.erb – listing view for existing model objects
  - show.html.erb – display details for a given model object
  - new.html.erb – render form to create a new model object
  - edit.html.erb – render form to update an existing model object
  - _form.html.erb – (partial) form to capture model object dat
- UI Control Helper Methods
  - link_to – Render a link to a resource (using GET by default)
  - button_to – Render a button to submit a form (using POST by default)
  - redirect_to – Reroute the user to a different resource
    - Often used within controller actions that modify a model object (i.e. create and edit

Actions, View Templates, HTTP Methods, and UI Controls

Controller Action	URI (for Product object)	View Template(s) Rendered	HTTP Method	UI Control
Index	/products	Index.html.erb	GET	N/A
new	/products/new	new.html.erb	GET	link_to
create	/products/create	show.html.erb (success) new.html.erb (error)	POST	button_to
show	/products/1	show.html.erb	GET	link_to redirect_to
edit	/products/1/edit	edit.html.erb	GET	link_to
update	/products/1	show.html.erb (success) edit.html.erb (error)	PATCH/PUT	button_to
destroy	/products/1	index.html.erb	DELETE	link_to

Routing Basics

- Actions and methods listed on previous slide are common to all *resources*
  - Resource – something (i.e. model) that clients interact with via URLs
  - Terminology comes from Representational State Transfer (REST)
- By default, Rails sets up these 7 resource routes for each model
  - Check out any generated controller file for an example
    - Define helper methods: products_path, new_product_path, edit_product_path(:id), product_path(:id)
  - Can specify different actions/routes via “rails generate” command

```
rails generate controller CreditCards open debit credit close
```

  - config/routes.rb allows for custom routing

Routing Examples

- Resource routes – define default actions/routes previously described

```
resources :products
```

- Map the root of an application

```
root ‘welcome#index’
```

- Map a get request for products/123 to the Catalog controller’s view action with { id: 123 } in params

```
get 'products/:id' => 'catalog#view'
```

- Routes file prioritizes routes according to the order in which they are defined
- View defined routes with “rake routes”

Form Helper Methods

- label_tag(:q, “Search for”)
- text_field_tag(:q)
- submit_tag(“Search”)
- form_tag(
  controller: “people”,
  action: “search”,
  method: “get”,
  class: “nifty_form”)
- text_area(:description,
  “Something catchy”,
  size: “24x6)
- checkbox_tag(:delivers)
- radio_tag(:order_type)
- password_field_tag(
  :password)
- hidden_field_tag(:id)
- select_tag(:city,
  options_for_select(
    [‘Wenham’, 1],
    [‘Beverly’, 2],
    [‘Ipswich’, 3]),
  2)

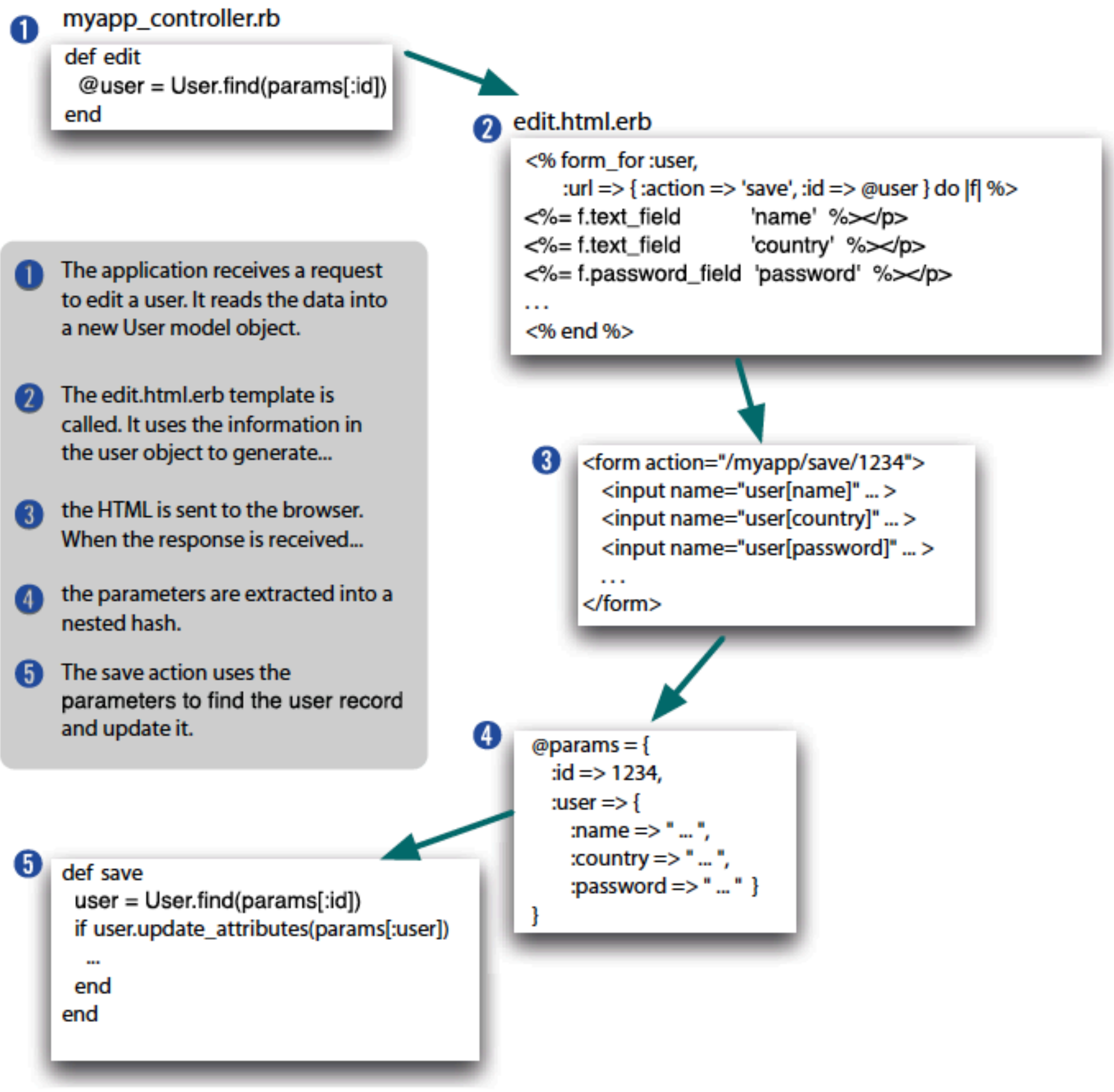
Form Helper Example

```
<%= form_for(:model) do |form| %>
<p>
<%= form.label :input %>
<%= form.text_field :input, :placeholder => 'Enter text here...' %>
</p>
<p>
<%= form.label :address, :style => 'float: left' %>
<%= form.text_area :address, :rows => 3, :cols => 40 %>
</p>
<p>
<%= form.label :color %>:
<%= form.radio_button :color, 'red' %><%= form.label :red %>
<%= form.radio_button :color, 'yellow' %><%= form.label :yellow %>
<%= form.radio_button :color, 'green' %><%= form.label :green %>
</p>
...
<% end %>
```

HTML5 Form Helper Methods

- email_tag
- search_field
- telephone_field
- url_field
- color_field
- date_field
- datetime_field
- datetime_local_field
- month_field
- week_field
- time_field

Controller/View Flow



1 myapp_controller.rb

```
def edit
  @user = User.find(params[:id])
end
```

2 edit.html.erb

```
<% form_for :user,
     :url => { :action => 'save', :id => @user } do |f| %>
<%= f.text_field        'name'    %></p>
<%= f.text_field        'country'  %></p>
<%= f.password_field  'password'  %></p>
. . .
<% end %>
```

3

```
<form action="/myapp/save/1234">
   <input name="user[name]" ... >
   <input name="user[country]" ... >
   <input name="user[password]" ... >
   . . .
</form>
```

4

```
@params = {
   :id => 1234,
   :user => {
      :name => " ... ",
      :country => " ... ",
      :password => " ... " }
}
```

5

```
def save
  user = User.find(params[:id])
  if user.update_attributes(params[:user])
    ...
  end
end
```

1 The application receives a request to edit a user. It reads the data into a new User model object.

2 The edit.html.erb template is called. It uses the information in the user object to generate...

3 the HTML is sent to the browser. When the response is received...

4 the parameters are extracted into a nested hash.

5 The save action uses the parameters to find the user record and update it.

View Helper Methods

- concat
- cycle
- pluralize
- truncate
- debug
- distance_of_time_in_words
- time_ago_in_words
- sanitize
- strip_links
- strip_tags
- number_to_currency
- number_to_phone
- number_to_human_size
- …and many more

Error Handling

- Why?
  - Avoid bad user experience
  - Track down bugs
  - Guard against security threats
- Steps
  - Catch an error condition
  - Take appropriate action
  - Alert the end user

Catching the Error Condition

- Controller actions are where errors occur (or propagate to), so controllers handle these errors
- rescue_from method
  - Takes a series of one or more exceptions
  - Along with a :with option and a method name or block

```
class CartsController < ApplicationController
  before_action :set_cart, only: [:show, :edit, :update, :destroy]
  rescue_from ActiveRecord::RecordNotFound,  with: :invalid_cart
...
```

- Common exception classes
  - ActiveRecord::RecordNotFound
  - ActiveRecord::RecordInvalid

Take Appropriate Action

```
def invalid_cart
  logger.error "Attempt to access invalid cart #{params[:id]}”
  redirect_to store_url, notice: 'Invalid cart’
end
```

- Log the error with the ActiveSupport::Logger class
  - logger.error
  - Several logging levels connoting different error severities
    - debug, info, warn, error, fatal, unknown
    - Each has its own method
  - Logs to the log for your application’s environment
    - log/development.log
    - log/test.log
- Also may want to attempt to recover from the error

Alert the User

```
def invalid_cart
  logger.error "Attempt to access invalid cart #{params[:id]}”
  redirect_to store_url, notice: 'Invalid cart’
End
```

- notice: adds a message to the flash
    - Persists to the following request, initiated by the redirect_to call

```
<% if notice %>
<p id=“notice”><%= notice %></p>
<% end %>
```

Only Allowing Valid Parameters

```
# Never trust parameters from the scary internet, only allow
# the white list through.
def line_item_params
  params.require(:line_item).permit(:product_id)
end
```

- Set up by default in scaffolded controller
  - require – ensures specified parameter(s) is present
  - permit – only allows specified parameters

Assets

- “Static” files located in app/assets
- CSS (app/assets/stylesheets)
  - Application-wide in application.css
  - Controller-specific in *controller*.css.scss
    - Supports CSS and SCSS/SASS
- Javascript (app/assets/javascript)
  - Application-wide in application.js
  - Controller-specific in *controller*.js.coffee
    - Supports Javascript and CoffeScript
- Images (app/assets/images)

Layouts

- Many pages share the same headers, footers, and side bars
  - Often with the same functionality – i.e. search box, login form
- Content from an action view can be placed within a layout
  - Rails actually renders two templates on each request – the one specified and an appropriate layout template (if it finds one)
    - In apps/views/layouts/*controller*.html.erb
    - In a layout declaration in the controller (to override the above default)
      - Potentially qualified with the actions to which the layout is to be applied
      - Or a method call which dynamically computes and returns a layout
    - In apps/views/layouts/application.html.erb (application default layout)

Layout Example

```
<html>
  <head>
    <title>Form: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <%= yield :layout %>
  </body>
</html>
```

- Layouts may contain dynamic ERB code
  - Have access to all data available to view templates
- Call to yield renders the view template
  - :layout contains the rendered view template for the action

Partials

- Partial-page templates – templates that can be reused to add "snippets" to a view
  - Can be thought of like functions
  - File names begin with "_" (i.e. _form.html.erb)
  - Typically reside in corresponding controller's directory
    - Can be shared across controllers by putting them in app/views/shared directory
- Invoked via the render() helper method
  - Can take a model object and other local variables
  - Can iterate over a collection of objects, generating content for each one
  - Can be rendered with a layout
  - Controllers can sometimes call partials (i.e. for Ajax updates to part of a page)

Partial Example

```
_article.html.erb
<div class="article">
  <div class="articleheader">
    <h3><%= article.title %> (for <%= suthorized_by %>)</h3>
  </div>
  <div class="articlebody">
    <%= article.body %>
  </div>
</div>
```

- Value passed to object key gets the same name as the partial file (for _article.html.erb, object gets named “article”).

```
render(partial: 'article',
  object: @an_article,
  locals: { authorized_by: session[:user_name],
            from_ip: request.remote_ip })
```