Action Controller Basics

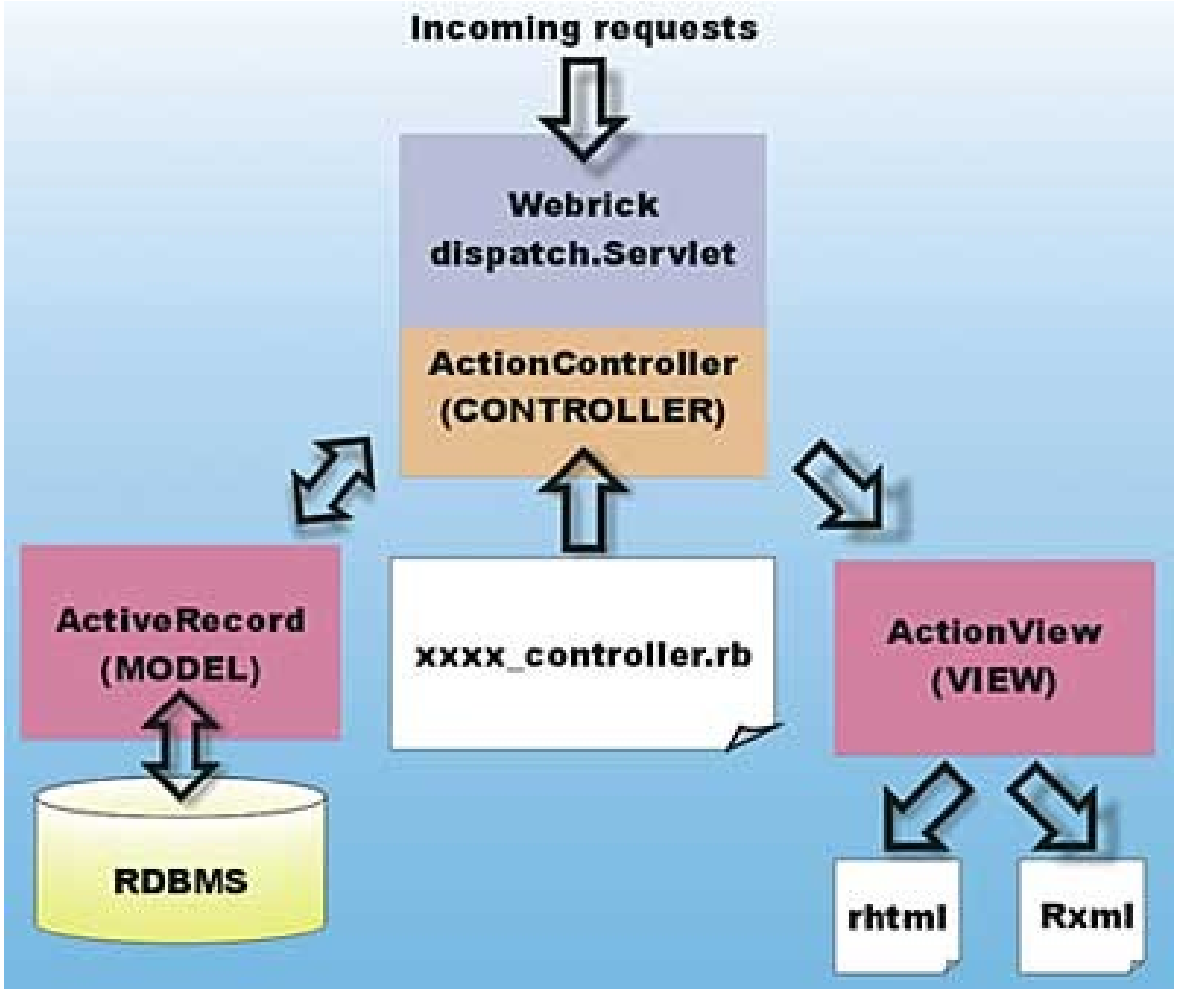
Sang Shin
www.javapassion.com
“Learn with Passion!”

Topics

- What is ActionController?
- ActionController classes
- Request handling
- Parameters
- Session & Flash
- Rendering
- Redirection
- Classic problem of user refresh & redirection
- Routing (request mapping)

What is ActionController?

“Ruby on Rails” MVC



source: http://www.ilug-cal.org

What is ActionController?

- ActionControllers handle incoming Web requests (HTTP requests)
- An ActionController is made up of one or more actions
- Actions are executed to handle the incoming requests and then either render a template or redirect to another action.
- An action is defined as a public method of a controller
- Mapping between a request's URL and an action is specified in the Rails routing map (*configuration/routes.rb*)

ActionController Classes

Example: ActionController Class

- ActionController class, *HelloController*, in the example below, is a subclass of *ApplicationController* class
- The *HelloController* has two action methods, *index* and *say_hello*

```
class HelloController < ApplicationController
  def index
    say_hello  # Call say_hello action method
    render :action=>'say_hello'
  end
  def say_hello
    @hello = Message.new(:greeting => "Hello World!")
  end
end
```

ApplicationController Class

- *ApplicationController* class is a subclass of *ActionController::Base* class
- *ApplicationController* class is automatically provided as *application.rb* file when you create a Rails project

```
class ApplicationController < ActionController::Base
  helper :all # include all helpers, all the time

  # See ActionController::RequestForgeryProtection for details
  # Uncomment the :secret if you're not using the cookie session sto
  protect_from_forgery # :secret =>
    'ccd81132ccd133b3d252a16e4f3627f8'
end
```

Request Handling

Request Handling

- Requests are processed by the Action Controller framework by extracting the value of the "action" key in the request parameters hash
- Once the action has been identified, the remaining request parameters, the session (if one is available), and the full request with all the http headers are made available to the action through instance variables.
- Then the action is performed.

Request Object

- The full request object is available with the *request* accessor and is primarily used to query for http headers.
- These queries are made by accessing the environment hash, like this:

```
def server_ip
    location = request.env["SERVER_ADDR"]
    render :text => "This server hosted at
   #{location}"
end
```

Parameters

Parameters

- All request parameters, whether they come from a GET or POST request, or from the URL, are available through the *params*, which returns a hash of parameters.
- For example, an action that was performed through /weblog/list?category=All&limit=5 will have

  { "category" => "All", "limit" => 5 } as a params hash.

Multi-Dimensional Parameters

- It‘s also possible to construct multi-dimensional parameter hashes by specifying keys using brackets, such as:

```
<input type="text" name="post[name]"
 value="david">
<input type="text" name="post[address]"
 value="somewhere">
```

- A request stemming from a form holding these inputs will include

```
{ "post" => { "name" => "david", "address" =>
  "somewhere" } }.
```

Multi-Dimensional Parameters (Cont)

- If the address input had been named "post[address][street]", the params would have included { "post" => { "address" => { "street" => "hyacintvej" } } }.
- There is no limit to the depth of the nesting.

Session

How Is Session Used for Web Applications?

- Sessions allows you to store objects in between requests.
  - The state of those objects are maintained between requests
- This is useful for objects that are not yet ready to be persisted, such as a *Signup* object constructed in a multi-paged process, or objects that don't change much and are needed during a session, such as a *User* object for a system that requires login.

How to Place Objects Into Session

- You can place objects in a session, which is a hash object

```
session[:person] =
   Person.authenticate(user_name,
                       password)
```

- And retrieve them through the same hash

```
Hello #{session[:person]}
```

- For removing objects from the session, you can either assign a single key to nil, like *session[:person] = nil*, or you can remove the entire session with reset_session.

Where Session Data Stored? (Default)

- Session data is stored in a local file system as a default
  - For a simple app server configuration, this is OK.
  - But for multi-appserver configuration with load-balancer, the session data might not be available when it is needed.

Alternative Place Where Session Data Can Be Stored

- *ActiveRecordStore: Sessions are stored in your database*
  - Works better with multiple app servers
- To use ActiveRecordStore, set the following in *environment.rb*
  - *config.action_controller.session_store = :active_record_store*
- And run *rake db:sessions:create* to create a database and a table for storing sessions
- You need restart the server

Flash

What is Flash?

- The flash provides a way to pass temporary objects between two consecutive actions.
  - Anything you place in the flash will be exposed to the very next action and then cleared out.
- This is a great way of doing notices and alerts, such as a create action that sets *flash[:notice] = "Successfully created"* before redirecting to a display action that can then expose the flash to its template.
- You can put any object in there
- You can put as many as you like at a time too

Flash Example

```
class WeblogController < ActionController::Base
  // 1st action
  def create
    # save post
    flash[:notice] = "Successfully created post"
    redirect_to :action => "display", :params => { :id => post.id }
  end
  // 2nd action
  def display
    # doesn't need to assign the flash notice to the template,
    # that's done automatically
  end
end

// Template for the 2nd action, display - display.erb
<% if flash[:notice] %><div class="notice"><%= flash[:notice]
  %></div><% end %> # if there is a flash notice, display it
```

Rendering

Automatic Rendering

- Rails framework, after an action is performed, searches for a template, whose name is *<name-of-the-action>.erb* or *<name-of-the-action>.rhtml* in the *app/views/<name-of-controller>* directory and renders it as a default
  - Example: After *say_hello* action is performed, Rails will look for say_hello.erb (or say_hello.html.erb) or say_hello.rhtml under app/views/hello directory.

Sharing Instance Variables

- The instance variables created in the controller are automatically available to templates
  - Controller code

```
def show
  @post = Post.find(params[:id])
end
```

  - Template

```
Title: <%= @post.title %>
```

Automatic Rendering Example

- For example, by default, the *index* action of the *GuestBookController* below would render the template *app/views/guestbook/index.erb* or *app/views/guestbook/index.rhtml* after populating the @entries instance variable.

```
class GuestBookController < ActionController::Base
    def index
      @entries = Entry.find(:all)
    end

    def sign
      Entry.create(params[:entry])
      redirect_to :action => "index"
    end
end
```

Manual Rendering

- You don‘t have to rely on the default rendering. Especially actions that could result in the rendering of different templates will use the manual rendering methods:

```
def search
  @results = Search.find(params[:query])
  case @results
    when 0 then render :action => "no_results"
    when 1 then render :action => "show"
    when 2..10 then render :action => "show_many"
  end
end
```

More Rendering Examples

```
# Renders the template for the action "goal" within the
# current controller
render :action => "goal"

# Renders the template for the action "short_goal" within
# the current controller, but without the current active
# layout
render :action => "short_goal", :layout => false

# Renders the template for the action "long_goal" within
# the current controller, but with a custom layout
render :action => "long_goal", :layout => "spectacular"
```

Even More Rendering Examples

```
render :action=>:index
render :action=>'index'
render :text => “Hello World!”
render :template=>'myweb/something'

# We will study render :partial in detail in
# ActionView
render :partial=>'fruits', :collection=>
  %w{apple,pear,banana}
```

Redirection

Redirects

- Redirects are used to move from one action to another
- Example scenario
  - For example, after a create action, which stores a blog entry to a database, we might like to show the user the new entry.
  - Because we‘re following good DRY principles (Don‘t Repeat Yourself), we‘re going to reuse (and redirect to) a show action that we‘ll assume has already been created.

Redirect Example

- The code might look like this:

```
def create
    @entry = Entry.new(params[:entry])
    if @entry.save
      # The entry was saved correctly, redirect to
   show
      redirect_to :action => 'show', :id => @entry.id
    else
      # things didn't go so well, do something else
    end
end
```

More Redirect Examples

```
#Redirect to show action with a param
redirect_to :action => "show", :id => 5

#Redirect to another website
redirect_to "http://www.rubyonrails.org"

#The current protocol and host is prepended to the string
redirect_to "/images/screenshot.jpg"

# Back to the page that issued the request.
redirect_to :back
```

Even More Redirect Examples

```
# The redirection happens as a "302 Moved"
# header unless otherwise specified.
redirect_to post_url(@post), :status=>:found
redirect_to :action=>'atom',
 :status=>:moved_permanently
redirect_to post_url(@post), :status=>301
redirect_to :action=>'atom', :status=>302
```

Classic Problem of User Refresh - Use “redirect” (rather than “render”)

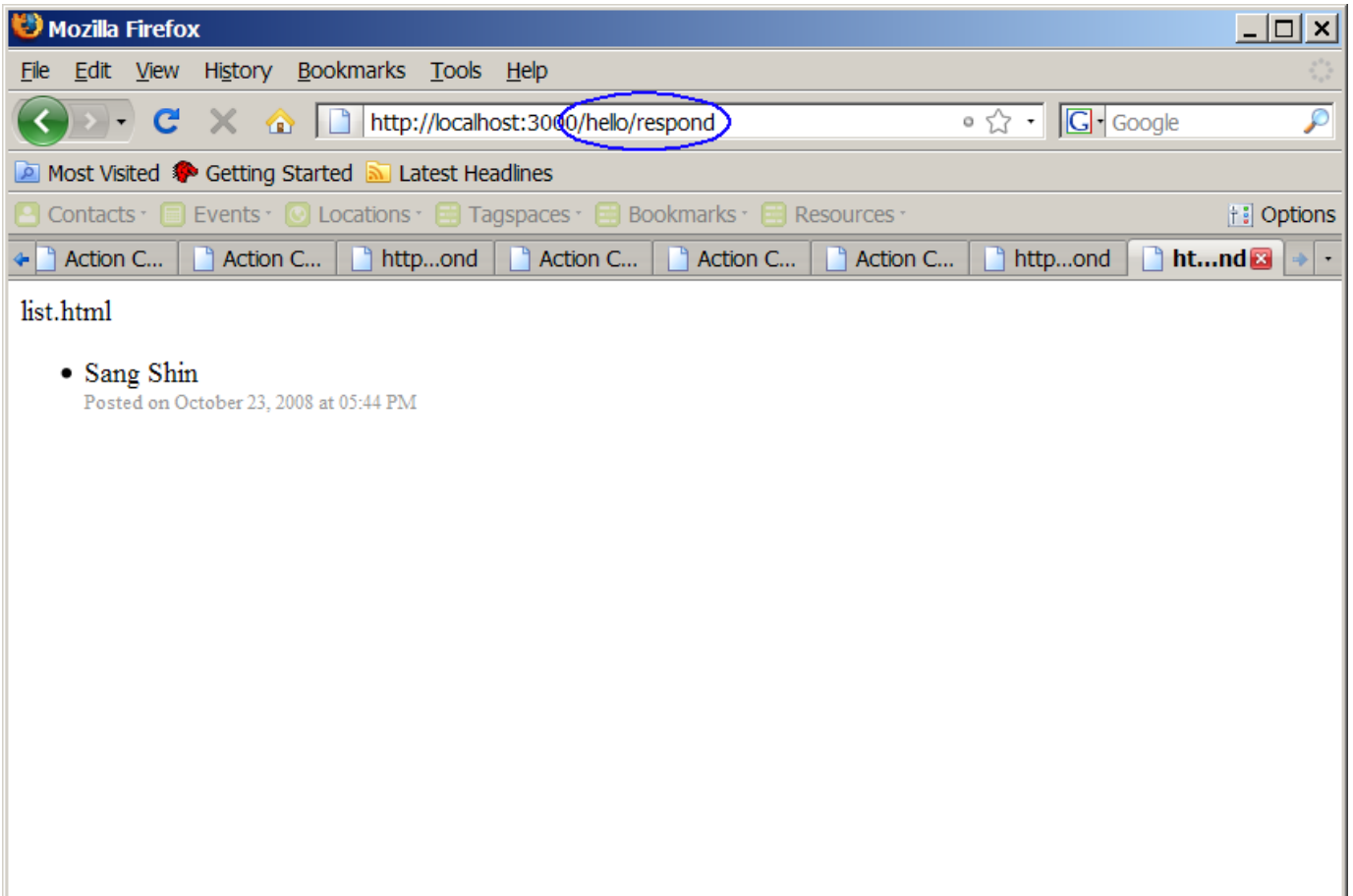
“User Pressing Refresh” with render

- Problem code - If you use “*render :action=>:list*” at the end of “state changing” action, such as *respond* action below - it is creating a new *user* record -, and a user presses refresh button of the browser, *respond* action is executed again, which creates a duplicate *user* record

```
def respond
    @user = User.new(params[:user])
    @user.save
    ###### DON"T DO THIS!  YOU SHOULD USE REDIRECT!
    ######
    render :action=>:list
  end
```

“User Pressing Refresh” with render

- Problem code - Note that the URL is set to */hello/respond*, so refreshing the page calls */hello/respond* again.



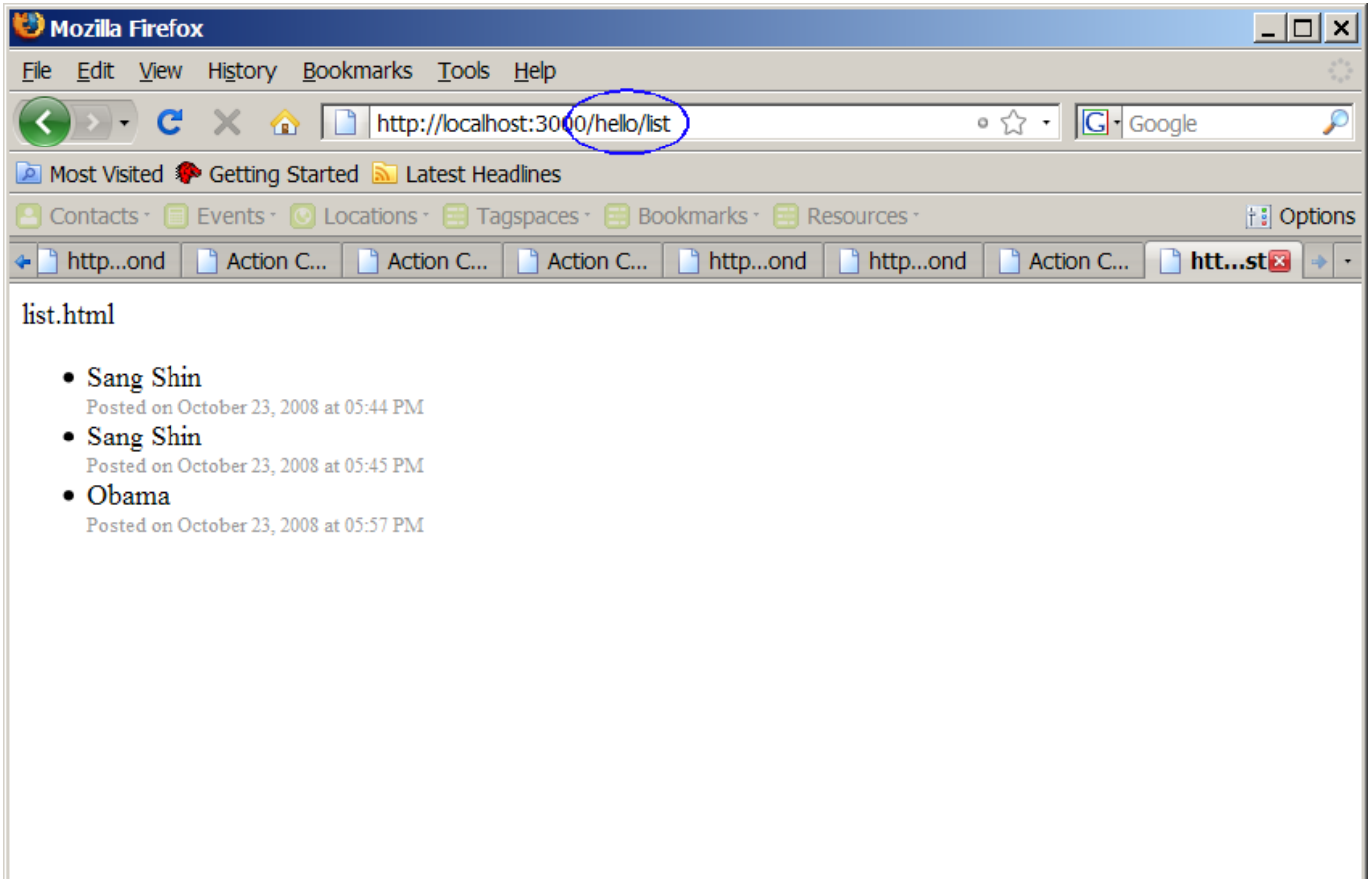
“User Pressing Refresh” with Redirect

- Corrected code - when *redirect_to :action=>:list* is used, the browser redirects itself to the *list.html.erb* page.

```
def respond
  @user = User.new(params[:user])
  @user.save
  ###### USING REDIRECT! ######
  redirect_to :action=>:list
end
```

“User Pressing Refresh” with Redirect

- Corrected code - Note that the URL is set to */hello/list*, so refreshing the page calls */hello/list* again.



Routing (Request Mapping)

Routing

- Maps incoming request to controllers and actions
- Routes are defined in *routes.rb* file
- *map.connect ':controller/:action/:id'*
  - Requests consist of a *:controller* followed by an *:action* that in turn is fed some *:id*
  - Suppose you get an incoming request for */blog/edit/22,* you'll end up with:

```
params = { :controller => 'blog',
           :action     => 'edit',
           :id         => '22'
         }
```

Setting Routes

- Setting a route is straightforward in Rails - you provide a pattern and a hash of parameters
- Example - The example below sets up *blog* as the default controller if no other is specified. This means visiting '/' would invoke the *blog* controller.

```
ActionController::Routing:Routes.draw do |map|
  map.connect   ':controller/:action/:id', :controller => 'blog'
end
```

Route Priority

- Routes have priority defined by the order of appearance of the routes in the *routes.rb* file.
- The priority goes from top to bottom.
- The last route in that file is at the lowest priority and will be applied last.
  - Your default route should be placed at the end
- If no route matches, 404 is returned.

Named Routes

- Routes can be named with the syntax *map.name_of_route* options, allowing for easy reference within your source as *name_of_route_url* for the full URL and *name_of_route_path* for the URI path.

Named Routes Example

- Routes can be named with the syntax *map.name_of_route* options, allowing for easy reference within your source as *name_of_route_url* for the full URL and *name_of_route_path* for the URI path.
- routes.rb

```
map.myindex '/hello/index', :controller=>'hello',
  :action=>'index'

map.myrespond '/hello/respond',
  :controller=>'hello', :action=>'respond'
```

- respond.html.erb

```
<%= link_to "Go to Index", myindex_url %>
```

routes.rb Example

```
ActionController::Routing::Routes.draw do |map|
  # The priority is based upon order of creation: first created -> highest
  # priority.

  # Sample of regular route:
  map.connect 'products/:id', :controller => 'catalog', :action => 'view'
  # Keep in mind you can assign values other than :controller and :action

  # Sample of named route:
  map.purchase 'products/:id/purchase', :controller => 'catalog', :action
    => 'purchase'
  # This route can be invoked with purchase_url(:id => product.id)

  # You can have the root of your site routed by hooking up ''
  # -- just remember to delete public/index.html.
  map.connect '', :controller => "welcome"
end
```

Thank you!

We do Instructor-led Codecamps!
http://www.javapassion.com/codecamps