# ActionView Basics

Sang Shin
Michèle Garoche
www.javapassion.com
"Learning is fun!"

# Topics

- Types of templates
- ERb template
- Builder template
- JavaScriptGenerator template
- Using sub-templates
- Layouts
- Partials

# Types of Templates

# Types of ActionView Templates

- Action View templates can be written in three ways.
  1. ERb: If the template file has a *.erb (or .rhtml)* extension then it uses a mixture of ERb (included in Ruby) and HTML.
  2. Builder: If the template file has a *.builder (or .rxml)* extension then Jim Weirich's *Builder::XmlMarkup* library is used.
  3. JavaScriptGenerator: If the template file has a *.rjs* extension then it will use ActionView::Helpers::PrototypeHelper::JavaScriptGenerator.

# ERb Template

# How ERb is Triggered

- You trigger ERb by using embeddings such as <% %>, <% -%>, and <%= %>.
  - The <%= %> tag set is used when you want output
- Example

```
<b>Names of all the people</b>
<% for person in @people %>
  Name: <%= person.name %><br/>
<% end %>
```

  - The loop is set up in regular embedding tags <% %> and the name is written using the output embedding tag <%= %>

# Builder Template

# What is Builder Template?

- Builder templates are a more programmatic alternative to ERb templates.
- They are especially useful for generating XML content.
- An *XmlMarkup* object named *xml* is automatically made available to templates with a *.builder* extension.
- Example

```
xml.em("emphasized")          # => <em>emphasized</em>
xml.em { xml.b("emph & bold") }    # => <em><b>emph
  &amp;amp; bold</b></em>
xml.a("A Link", "href"=>"http://onestepback.org") # => <a
```

# Nested Markup using a Block

- Any method with a block will be treated as an XML markup tag with nested markup in the block.
- Example

```
xml.div {
 xml.h1(@person.name)
 xml.p(@person.bio)
}
```

*would produce something like:*

```
<div>
 <h1>David Heinemeier Hansson</h1>
 <p>A product of Danish Design during the Winter of
'79...</p>
```

# JavaScriptGenerator Template (Details of this will be covered in Ajax Session.)

# JavaScriptGenerator Template

- JavaScriptGenerator templates end in *.rjs* extension.
- Unlike conventional templates which are used to render the results of an action, these templates generate instructions on how to modify an already rendered page.
  - This makes it easy to modify multiple elements on your page in one declarative Ajax response.
  - Actions with these templates are called in the background with Ajax and make updates to the page where the request originated from.

# Using Sub Templates

# Using Sub Templates

- Using sub templates allows you to sidestep tedious replication and extract common display structures in shared templates.
  - The classic example is the use of a header and footer (even though using layouts is a recommended scheme)

```
<%= render "shared/header" %>
Something really specific and terrific
<%= render "shared/footer" %>
```

  - The render call itself will just return a string holding the result of the rendering. The output embedding writes it to the current template.

# Sharing Variables Among Templates

- Templates can share variables amongst themselves by using instance variables defined using the regular embedding tags.
- Example
  - Define a instance variable called *@page_title* in the calling template

```
<% @page_title = "A Wonderful Hello" %>
<%= render "shared/header" %>
```

  - Now the *shared/header* template can pick up on the *@page_title* variable and use it for outputting a title tag:

```
<title><%= “Page title is #{@page_title}”
```

# Template Caching

- By default, Rails will compile each template to a method in order to render it.
- When you alter a template, Rails will check the file’s modification time and recompile it.

# Layouts

# Layout

- Common layout can be used for multiple pages
  - Header and footer
  - Stylesheet
  - JavaScript libraries

# Layout Example: application.rhtml

```
<html>
  <head>
    <title>Events</title>
    <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8" />
    <%= stylesheet_link_tag 'application' %>
    <%= javascript_include_tag :defaults %>
  </head>
  <body>
      ...
      <div id="page">
        <div id="content">
          <%= content_tag(:div, flash[:notice], :class => 'flash') if
    flash[:notice] %>
          <%= yield %>
        </div>
        <hr />
      </div>
```

# Layout Usage Conventions

- A layout named *views/layouts/application.rhtml* is applied automatically unless a more specific one exists or is explicitly specified in the controller
- A layout that matches the name of the controller is used if present for the pages related to the controller
- You can use *layout* directive at the class level in any controller
- You can include layout for a specific action

# Example: Layout

- Use a given layout for all of a controller's actions

```
class IndexController < AppplicationController
   layout 'mylayout'

   def index
   end

   def list
   end
end
```

# Example: Layout

- Use a given layout for a particular action

```
class IndexController < AppplicationController
  def index
    render :layout => 'mylayout'
  end

  def list
  end
end
```

# Rendering Partials

# What is Partials?

- Enables reusability of partial template (fragment)
- Follows naming convention of being prefixed with an underscore — as to separate them from regular templates
- In a template for Advertiser#account:

```
<%= render :partial => "account" %>
```

- This would render "*advertiser/_account.rhtml*" and pass the instance variable *@account* in as a local variable account to the template for display

# Example: Partials

- In another template for Advertiser#buy, we could have:

```
<%= render :partial => "account", :locals =>
 { :account => @buyer } %>

<% for ad in @advertisements %>
 <%= render :partial => "ad", :locals => { :ad
 => ad } %>
<% end %>
```

- This would first render "*advertiser/_account.rhtml*" with *@buyer* passed in as the local variable account, then render "*advertiser/_ad.rhtml*" and pass the local

# Rendering Partials

- Partial rendering in a controller is most commonly used together with Ajax calls that only update one or a few elements on a page without reloading.
- Rendering of partials from the controller makes it possible to use the same partial template in both the full-page rendering (by calling it from within the template) and when sub-page updates happen (from the controller action responding to Ajax calls).
- By default, the current layout is not used.

# Rendering Partials Examples

```
# Renders the same partial with a local variable.
render :partial => "person", :locals => { :name => "david" }

# Renders the partial, making @new_person available
  through
# the local variable 'person'
render :partial => "person", :object => @new_person

# Renders a collection of the same partial by making each
  element
# of @winners available through the local variable "person"
  as it
# builds the complete response.
render :partial => "person", :collection => @winners
```

# Rendering Partials Examples

```
# Renders the same collection of partials, but also renders
 the
# person_divider partial between each person partial.
render :partial => "person", :collection => @winners,
 :spacer_template => "person_divider"

# Renders a collection of partials located in a view subfolder
# outside of our current controller.  In this example we will be
# rendering app/views/shared/_note.r(html|xml)  Inside the
 partial
# each element of @new_notes is available as the local var
 "note".
render :partial => "shared/note", :collection => @new_notes

# Renders the partial with a status code of 500 (internal
```

# Thank you!

Sang Shin
Michèle Garoche
http://www.javapassion.com
“Learning is fun!”