Fearless Refactoring

Rails Controllers

Andrzej Krzywda

Fearless Refactoring

Rails Controllers

Andrzej Krzywda

Contents

Explicitly render views with locals

Introduction

The default practice in Rails apps is not to care about calling views. They are called and rendered using conventions. Whenever an action is called, there's an implicit call to render, so you don't have to do that manually. Less code, more conventions.

Such conventions are very useful at the early stage of the project. They speed up the prototyping phase. Once the project becomes more complex, it might sometimes be useful to be more explicit with our code.

There are two things that are implicit.

- The call to render itself
- The way the data is passed to the view

In theory, this refactoring is simple. Go to the view, replace all @foo with foo. The same in the controller. Also, in the controller, at the end of the action, call

```
render :locals => {:foo => foo}
```

In practice, you need to be careful when the view renders a partial. You need to explicitly pass the local variable further down. Also, views are not tied to one action. There are typical reusable views like 'new', 'edit', '_form'. In those cases all the actions need now to pass locals in appropriate places.

It's also important to remember, that you're not just rendering a view. You're rendering the whole layout. The view is just rendered inside. What that means, is that the whole layout depends on the @ivars or locals. It's easy to forget to check what exactly the layout depends on.

The nice thing about render :locals, is that it doesn't mean all or nothing. It means that, if your view relies on many @ivars, for the safety, you can make the transition, gradually. One @ivar into local, at a time. It also means, that you don't have to be scared that the layout depends on some @ivar set in an ApplicationController before_filter (a common pattern).

After this transformation is done, you've got a little verbose "render" calls. They're now taking params explicitly. It's good to apply the "Extract render/redirect methods" refactoring afterwards.

Example

The example comes from the lobste.rs project (a HackerNews clone). The 'tree' action is responsible for retrieving all users in the system, grouping them by parent (a person who invited the user). The view then takes this data structure and displays as a tree-like representation, with nesting.

```
class UsersController < ApplicationController

  def tree
    @title = "Users"

    users = User.order("id DESC").to_a

    @user_count = users.length
    @users_by_parent = users.group_by(&:invited_by_user_id)
  end
end


<div class="box wide">
  <p><strong>Users (<%= @user_count %>)</strong></p>

  <ul class="root">

    <% subtree = @users_by_parent[nil] %>
    <% ancestors = [] %>

    <% while subtree %>
      <% if (user = subtree.pop) %>
        <li>
          <a href="/u/<%= user.username %>"
            <% if !user.is_active? %>
              class="inactive_user"
            <% elsif user.is_new? %>
              class="new_user"
            <% end %>
            ><%= user.username %></a> (<%= user.karma %>)
          <% if user.is_admin? %>
            (administrator)
          <% elsif user.is_moderator? %>
            (moderator)
          <% end %>
          <% if (children = @users_by_parent[user.id]) %>
            <% # drill down deeper in the tree %>
            <% ancestors << subtree %>
            <% subtree = children %>
            <ul class="user_tree">
          <% else %>
            </li>
```

```
          <% end %>
        <% else %>
          <% # climb back out %>
          <% subtree = ancestors.pop %>
          <% if subtree %>
            </ul></li>
          <% end %>
        <% end %>
      <% end %>
    </ul>
</div>
```

The action prepares 3 instance variables: @title, @user_count and @users_by_parent.

It's worth noting that @title is only used in the layout, not in the direct view. The "title concern" is orthogonal to what the action does. Let's leave it as it was. We don't gain much by moving it to locals, yet.

Before we go further, let's double check that the view doesn't call any other partial. It doesn't in this case. If it called a partial, we'd have to change the partial as well (and all places that call it).

The @user_count and @users_by_parent are quite simple to change. Let's first change the controller:

```
  def tree
    @title = "Users"

    users = User.order("id DESC").to_a

    user_count = users.length
    users_by_parent = users.group_by(&:invited_by_user_id)
    render 'tree', locals: {user_count: user_count, users_by_parent: users_by_par\
ent}
  end
```

We also change the view, by doing two find/replace operations:

@user_count -> user_count

@users_by_parant -> users_by_parent

This is a relatively safe transformation. If the view and controllers are so simple, you can even change them without test coverage.

Benefits

With more explicitness we gain a clear interface to the view layer. We know what needs to be passed. The view is no longer this place which magically accesses some global data, but it behaves more like a proper object.

This technique makes it easier, if one day, this view will be turned into a JavaScript widget (a popular trend recently). In that case, we can move the html part to the client-side (handlebars, mustache). This widget would make an ajax call to get the same data that we're now explicitly passing. Note that this requires more explicitness with helpers usage. See chapter "Turn helper calls into passing locals" for more details.

After this refactoring, we're now able to safely do the "Extract Service Object with SimpleDelegator" transformation.

Warnings

The locals will not be automatically available in partials, neither inside your action view nor inside the layout.

Make sure that you know all the actions that call this particular view, you're changing. You need to change all the calls.

Some views (or helpers) are accessing the instance variables in a different way. Here's a snippet from the Redmine project:

```
  def error_messages_for(*objects)
    html = ""
    objects = objects.map {|o| o.is_a?(String) ? instance_variable_get("@#{o}") :\
 o}.compact
    errors = objects.map {|o| o.errors.full_messages}.flatten
    ...
  end
```

As you see, here instance_variable_get is used, which tries to access the @ivar. Those places need to be changed before you change ivars into local variables. It's a common trap to fall into, if you just search for '@' in the file.

Some developers like to test whether the assigns are set with:

```
assert assigns(:time_entry).errors[:issue_id].present?
```

After this refactoring technique, you need to change this test. As long as you have no service object yet, it's better to test controller+view as a black box. This means, turn the test into one that test the html output (whether the error is displayed). See chapters "Testing" and "Refactor to test controller + view as a black box" for details, which tests make most sense, depending on the state of your code.

Resources

http://thepugautomatic.com/2013/05/locals/

http://therealadam.com/2014/02/09/a-tale-of-two-rails-views/

http://www.naildrivin5.com/blog/2014/02/09/a-defense-of-ivars-in-rails-controllers.html

http://www.slideshare.net/elia.schito/rails-oo-views

https://github.com/hudge/proffer - An Action Controller module to hide instance variables from views by default