Modular Page Assembly in Rails

How to use views, partials, and layouts in a modular approach to page assembly

Greg Willits • Rev 2a • Oct 27, 2007

http://www.gregwillits.ws/

In this article we look at the main elements and features available in Rails for managing page assembly, and present a structured approach to modular page design.

If you've come from a template-driven web application framework, the way Rails builds pages can feel a bit backwards. If you've tinkered with Rails at all, you've no doubt seen that when it comes to building the visible web page, Rails first processes what it considers the root view file, and then looks for a layout file to wrap around the view file. For me, I was accustomed to the exact opposite. The layout was the starting point, and its structure determined which view fragments the framework would include (based on dynamic file name conventions).

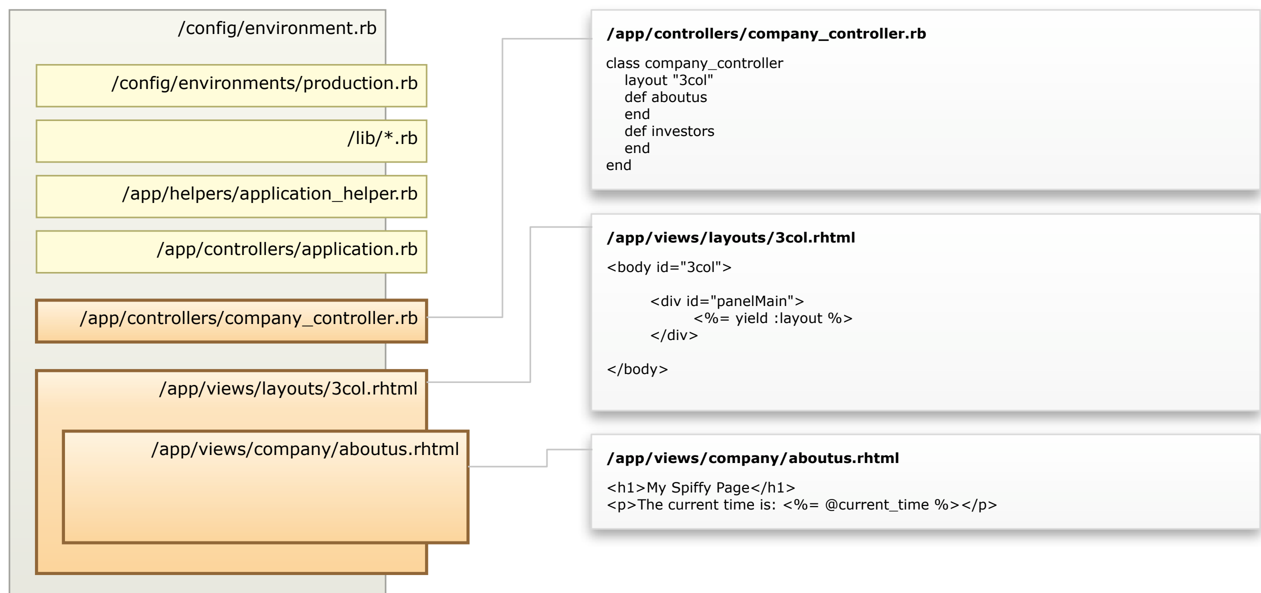
So, first I had to invert my thinking. Next, it seemed to me that Rails had no particular dialog for describing page layouts and populating arbitrary areas (I call them panels) with specific content. Before I could get too far into just going with the flow in Rails, I had to figure out how my pages could be structured with layouts that defined reusable panels for content like the header, footer, and even arbitrary panels like news headlines, contact information, and navigation menus. Only after I figured this out was I going to feel free to study the details of the logical side of Rails so I could visualize how I was going to organize the output of that code.

Rails Default Page Assembly

Just to set our starting point, let's have a quick review of the default page assembly process in Rails using the following URL:
http://www.example.com/company/aboutus

The figure at the bottom of the page shows the overall nesting of the main players. For now let's focus on the three orange (heavy outlined) boxes. By default, Rails will look for a folder with the same name as the controller, company, in the

Figure 1 • Typical Default Page Assembly Files for www.example.com/company/aboutus



views folder and a file with the same name as the action which would be aboutus.rhtml. If that file is not found, it will look for an index.rhtml file. If in the controller, we specify a layout, Rails will then look for that file to wrap around the aboutus view file. In the example below, a layout name of 3col was declared. So, Rails will look for the file /layouts/3col.rhtml. Using a minimum of options, this is the typical extent of building a default web page in Rails. In Rails terminology, the /company/aboutus.rhtml view gets wrapped in the /layouts/3col.rhtml layout.

With this method, we would likely author the page header and footer and other content not a part of the aboutus view into the layout. While the layout is free to use variables to make the content dynamic, we're going to be generating a lot of duplicated code if we use more than one layout.

So, my first question was how do I extract common material like a header or footer so it can be shared amount multiple layouts? The second question was how do I create a layout which has multiple dynamic panels?

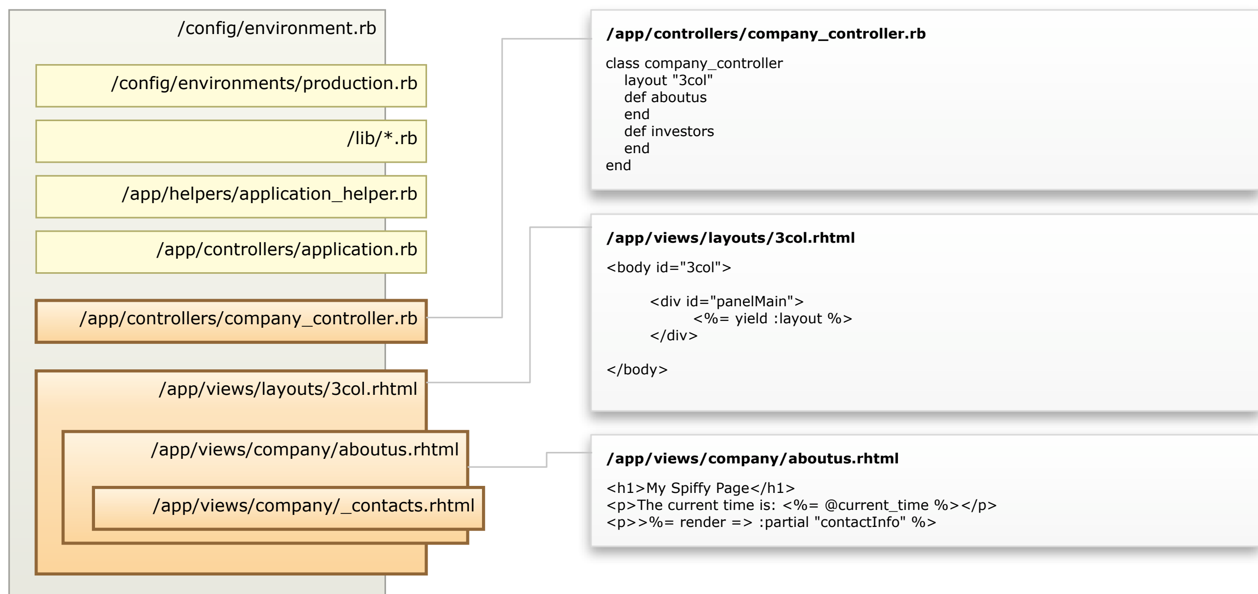
Using Rails Partials

To integrate the contents of multiple files into a layout, Rails uses a `render` command which integrates an arbitrary content fragment into a layout (or any template for that matter). You provide Rails the name and location of a file, and Rails loads it into the overall view content at the position of the `render` command. Rails calls this file a *partial*. A partial is any code fragment written just like the code in a view which can have HTML, ERb, Javascript, CSS, etc. The main thing that is different about a partial, is that the filename must begin with an underscore. Native view files (those that the Rails searches for based on the controller action) cannot start with underscores, but partials must.

So, let's consider a case where we need to display some company contact information, but it's something we want to share among several pages in the company views. To do that we'd create a separate file named _contactInfo.rhtml, and then we'd have that file rendered in our views like the aboutus view. In Figure 2, the code fragment for aboutus.rhtml has been extended to include a `render` command to do just that. There are additional options for the `render` command, but I'll leave it up to the Rails docs and many good books to explain the details of those. For this article, we only need to think about where to use `render`, not so much about its full capabilities.

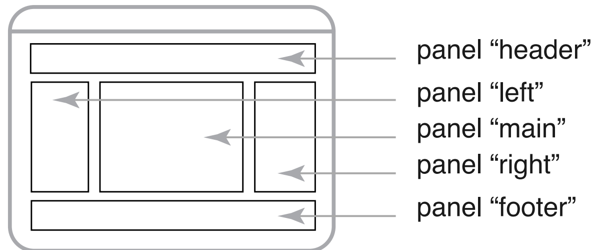
OK, so now we're armed with the basic methods of how Rails builds a page by using a view, a layout, and potentially many partials.

Figure 2 • Typical Page Assembly with a :partial in the View



A Classic Templated Approach

Let's dive right in, and look at a structure we could use to represent the typical 3-column web page layout which consists of the panels indicated in the illustration below:



Let's make the assumption that each of these panels is a separate file, each of which is brought together in the layout. Out of all these panels, Rails is really only interested in automating one of them for us, and that is the root view file. Which file should that be? If we assume that this view file is usually the most important content on the page, and that it fills the predominant panel of the page, we'll call that panel main.

For the purposes of the article, we'll label the left and right panels based on their positions, but it is likely that more meaningful names like navigation, headlines, adverts, or something that reflects the nature of the content is better. If you have a handful of sub panels with varying purposes, then you might resort to left and right for the div wrappers, and label each sub-panel. At any rate, label the panels as you see fit.

Using what we know so far, that Rails will seek out the view file, and the layout to wrap it, and that we can use the render command to insert partials, it seems logical to use the layout to manage the process of pulling in partials to build the page contents.

Figure 3 shows the end result of taking this approach. It's a logical approach, but I discovered that it has some flaws. Still, it's a good step to study, so we'll do that.

Figure 3 • Emulating a Typical Templated Approach (it's a logical approach, but has some weaknesses)

/config/environment.rb
/config/environments/production.rb
/lib/*.rb
/app/helpers/application_helper.rb
/app/controllers/application.rb
/app/controllers/company_controller.rb
/app/views/layouts/3col.rhtml
/app/views/_panels/_hdr_default.rhtml
/app/views/company/_aboutus_left.rhtml
/app/views/_panels/_contactInfo.rhtml
/app/views/company/aboutus.rhtml
/app/views/_panels/_whatever.rhtml
/app/views/company/_aboutus_right.rhtml
/app/views/_panels/_newsHeadlines.rhtml
/app/views/_panels/_ftr_default.rhtml

/app/controllers/company_controller.rb

```
class company_controller
   layout "3col"
   def aboutus
   end
   def investors
   end
end
```

/app/views/layouts/3col.rhtml

```
<body id="3col">
       <div id="pageHeader">
              <%= render(:partial => "_panels/hdr_default") rescue nil %>
       </div>

       <div id="panelLeft">
              <%= render(:partial => (controller.action_name + "_left")) rescue nil %>
       </div>

       <div id="panelMain">
                     <%= yield :layout %>
       </div>

       <div id="panelRight">
              <%= render(:partial => (controller.action_name + "_right")) rescue nil %>
       </div>

       <div id="pageFooter">
              <%= render(:partial => "_panels/ftr_default") rescue nil %>
       </div>
</body>
```

/app/views/company/_aboutus_left.rhtml

```
<p>Some constant text</p>
<%= render(:partial => "_panels/contactInfo") rescue nil %>
```

/app/views/company/aboutus.rhtml

```
<p>Some constant text</p>
<%= render(:partial => "_panels/whatever") rescue nil %>
```

/app/views/company/_aboutus_right.rhtml

```
<p>Some constant text</p>
<%= render(:partial => "_panels/newsHeadlines") rescue nil %>
```

The Layout

The 3col.rhtml layout establishes some naming conventions and takes care of some error handling scenarios for us.

First, you'll notice that all the partials for panels not specific to the page being viewed are stored in a folder at /views/_panels/ (see sidebar). Right now we're expressing that with a literal path name. If we want to move that path, we have a lot of retyping to do, so later we'll fix that by using a variable, but for now, it's easier to read.

Next, notice that the left and right panels have some expressions for file names in the :partial parameter.

```
render(:partial =>
   (controller.action_name + "_left"))
   rescue nil
```

The clause controller.action_name is a Rails object and attribute. controller is an object name that holds information about the current controller, and action_name is an attribute of that object that tells us what the current action of controller is. In our URL of /company/aboutus, "aboutus" is the action name. Therefore, the string aboutus will be inserted as the first part of the :partial file name. This is followed by hard coded declarations for each :partial as having a _left or _right suffix. These hard coded strings are our names of the panels, and completes the file names for Rails to find. Note that if we wanted the panel name to be headlines or blogBuddies, then we'd use strings like _blogBuddies for the filename in the render command instead of _left.

You'll also see the rescue nil clause of the render command. This is standard Ruby error trapping taken advantage of by Rails. If the file we've defined to be loaded doesn't exist, Ruby would normally complain and stop processing. Using rescue tells it to do something other than complain, and nil tells it to do nothing. Every page which uses the 3col layout may not need every panel filled out. So, for some pages there may not be a _right panel file to load. By using the rescue command, we don't have worry about that. If there's a right panel, Rails will find it. If there's no right panel, neither Rails nor Ruby will bother to complain about it, and that makes life easier on us.

Finally, you'll see I have not defined a _main file. That's because if I did, Rails wouldn't find it. Rails is designed to look for /views/company/aboutus.rhtml. While there's likely a way to override that, there' not a lot to be gained from doing so. We may as well leave that conventional part alone, and focus on how to build around it.

The Partials

In the code fragments for the layout and partials you'll notice that some use the _panels folder and some do not. Where you put files will be up to you, but the logic I follow is that if the panel is directly related to the content presented in the main view file, then I put those panel files in the controller-specific view folder (i.e. /company/). They're directly related to the view, so they should be in close proximity. If the panel is generic to multiple pages, then I'll put it in the _panels folder.

In my example layout, I'm declaring that for the purpose of the 3col layout, the left and right panels are for content directly related to the main view content. Therefore, the layout defines file names for partials based on that. For example, if we say that aboutus.rhtml has a bio on the company, then the left panel might have a list of company officers, and probably links to their bios. It's not something that is part of the main navigation. It's specific to the page. The right panel might be used

Not Sharing in the Idiom

I break a page into the elements which begins with the layout. This is the level of HTML with provides the overall structure of the page, It lays out the big chunks of content. I call those chunks panels. Panels are the regions of the page like the header, footer, sidebar, news headlines, contact info block, etc. Rails' use of the term partial is good in that it imposes no judgement of the role of the file. However, when discussing layout and design, I prefer a term with a more concrete definition of scope, so I came up with panels. As far a page layout goes, a panel might have a partial which declares structure, then include several other partials to fill the content.

Apparently it is common among Rails developers to place partials which would be shared by multiple controllers in a folder at /views/shared/. I suppose that makes sense, but it's a little too vague for me, so I store them in a folder at /views/_panels/. I use an underscore to emphasize that the folder is for structural organization and not a part of my application-specific content organization. It may be anti-idiomatic, but maybe I can start a new idiom :-)

mainly for photos to go along with the content. In a site with this layout, that type of relationship of content would exist wherever 3col is used. You may have a site where left and right are both for generic content. In which case, they would use the _panels subdirectory.

Now, let's say at the bottom of the left panel for every page, I want a small block of contact information. This is something that isn't unique to just this page, so it doesn't really belong in the controller-specific view folder. This is the type of partial, or sub-panel that would belong in the _panels folder.

I have the example code in Figure 3 written to include the _contactInfo file inside the _left file. If contactInfo is something of an independent block reused on many pages, shouldn't it have it's own panel defined in the layout file 3col? Well, maybe. This is where you get to play with HTML structure and layout decisions which have nothing to do with Rails. Define your layouts to suit your content and design structures, and let the files and folders needed fall naturally from that.

This article isn't trying to advocate specific HTML layouts or file systems structures, that's up to you and your project. All I'm really trying to accomplish is a more complete description about how to organize files and use the `render` command to build more complex page layouts than I've managed to find in Rails tutorials.

Getting Data into Panels

Your structured page is all well and good, but how do you get data into the panels? Taking this next step of the page layout challenge, I found some flaws in where we currently sit with the methodology of Figure 3.

Rails Partials are View Methods

The seemingly obvious answer to getting data into the panels is use the instance variables of the controller. If one of the goals of the controller was to fetch news into a `@newsHeadlines` instance variable, well, we would use that var inside the news partial. It's simple, it works. It's also flawed from a clean object-oriented code philosophy.

If we step back a bit, we know a basic practice in object-oriented programing ("OOP") is to pass to an object's method all data that it needs. The method should be unaware of its environment (in the majority of cases, so let's just stick with that). If all data that the method needs is handed to it each time, that method can continue to work correctly even if the environment changes.

By using the instance var name, we're giving the partial template insider knowledge of it's calling controller. In OOP, this isn't good. If it turns out that we can use the partial to output news in another page, but that page was written by someone else who decided to put data into an instance var called `@newsTidBits`, then the partial won't work. It expects to use `@newsHeadlines`.

Just like we do with methods in models and controllers, we want to define views with interfaces which abstract the details of what goes on inside. So, we want to pass data into views using parameters and avoid using instance vars.

Passing Data Through Render

We need to pass data into partials as if they were methods, but how can this be done when there's no `def` declaration like a regular Ruby method? Where's the interface that defines what the parameters are?

Partials are Methods?

The way Rails uses templates for views, it can be difficult to see them as counterparts to model and controller methods. In some systems, a view would be programmed like this:

```
class CompanyView
def aboutusMain
   content = String.new
   content += "<h1>About Us</h1>"
   content += "<p>We are...</p>"
   content += "<p>As a global...</p>"
   return content
end
end
```

What in Rails we're accustomed to writing in partials files would be programmed as a regular class with methods that build strings of HTML to return. This gets quite awkward to write, and prevents us from using HTML and CSS tools, so Rails and many web frameworks allow HTML in files like partials.

In the above code it is easy to recognize that if we wanted to use some data in that HTML, we'd have to pass it as parameters. We want to treat our partials in exactly this manner. Visualize that each partial is a method.

Rails has built this interface into the `render` command with a few mechanisms for passing data. One of these mechanisms is a parameter called `:locals` which looks something like this:

```
render( :partial => "filename",
      :locals => {:paramName = @anyVar})
```

where `@anyVar` is the the name of the object you're passing into the partial, and `:paramName` is the application-specific parameter name (like a method parameter name) you're going to call this data inside the partial. Let's look at a more explicit example.

Let's say we have a partial that shows news headlines. Maybe these headlines are the 5 most recent articles, or the 10 most popular. The display format is the same, so we want the partial to be reusable, but the data may have unique sources. So, inside the partial we'd have something like this:

```
<% headlines.each do |thisHeadline| %>
  <h2><%= h(thisHeadline['title']) %></h2>
  <p><%= h(thisHeadline['date']) %></p>
  <p><%= h(thisHeadline['location']) %></p>
  <p><%= h(thisHeadline['brief']) %></p>
<% end %>
```

As you can see, we're expecting a local variable named `headlines` to be available, in OOP-speak we're treating `newsHeadlines` as a parameter to this view method. To make that happen, we use a `render` command that looks like this:

```
render(
   :partial => "newsHeadlines",
   :locals => {
      :headlines = @topStories})
```

In another part of the application we might call it like this (notice the different instance var):

```
render(
   :partial => "newsHeadlines",
   :locals => {
      :headlines => @recentStories})
```

This lets us reuse the _newsHeadlines partial in multiple places on the site with different types of data fed in to it thanks to the interface that the `render` command creates for us.

The `:locals` option is only one way this can be done. Each option is discussed on pages 510-511 of the AWDWR[1] 2nd Edition, so I won't repeat the technical details of each. I'll focus on `:locals` for this article, and you can investigate the usefulness of the other options `:object` and `:collection`.

The render :object Shortcut

I said I wasn't going to address the details of other render options, but there's one area I'd like to throw my 2 cents in on.

If you have a partial named _newsHeadlines, and you want it to use a local variable internally called newsHeadlines, and you want to pass it some data from an instance var @newsHeadlines, then Rails says "gee, that's a lot of typing of 'newsHeadlines'" and goes ahead and makes the assumption that if you're using a partial of that name, and you have an instance var by the same name as that partial, Rails will automatically make that instance var available to a local variable in the partial. So, this code:

```
render(
   :partial => "newsHeadlines",
   :object => @newsHeadlines)
```

can be shortened to just this:

```
render(:partial => "newsHeadlines")
```

and the partial can use a local var named newsHeadlines.

My 2 cents on this is that it's a nifty trick to save some typing, but I find it lacking clarity. This is the type of implicit coding I find problematic about infamous PERL one liners. I simply prefer the interfacing of controllers, views, and partials to have explicit details so that its obvious what's going on, and obvious where changes will have impact. So, I'm not a fan of this shortcut, even at the expense of a little code monotony.

This system is a bit lopsided in that it's not obvious from within the partial code what's happening. It's a design compromise, but one that allows us both to write HTML/ERb fragments, and to integrate these fragments using abstracted parameters to pass data into them. We'll be on our own to identify inside our partial code what they expect in terms of parameters. Usually, the partial will be small enough that the parameters end up being self documenting. If not, a few comments in your partial file can clear things up.

Revisiting our Layout

The layout plan in Figure 3 has some problems when it comes to this idea of partials being abstracted view methods. Partials like _aboutus_left were integrated by using `render` commands from within the layout. However, we've just seen that `render` may need to have multiple versions to allow for passing data into the partials. Do we use two layouts? That would cause redundancy, and defeat the purpose of having a layout.

With the render command in the layout, there's little or no opportunity to use data passing parameters like `:object` or `:locals`. The layout is supposed to be reusable for multiple pages, but there's no way for it to know what data we might want passed into the partial. We could use a `:local` or an `:object` generically named for the panel like `leftData`, but that would be seriously lame. It's meaningless in the context of our application domain, and go against everything in the name of code readability.

Slicing up the layout into panels and partials makes good sense for design flexibility and organization, but since we can't effectively abstract the data passing, we have a problem.

The cure is to move the `render` command somewhere else in the system where it can be adaptable to the immediate data needs of the controller action, yet still find its way to filling in the panels of a layout.

It's all about the View

It turns out that the answer all along has been with the Rails view file. A combination of the Rails command `content_for` and a Ruby command called `yield` that Rails takes advantage of in the view/layout processing provides us the ability to pass data to partials in an OOP-approved manner, is flexible for each controller action, and generates content in a way that Rails knows how to add to our layout.

The view file is more than just a place to write HTML/ERb to generate your primarily display. It's actually the hub for determining how display for the entire page is constructed. While for a simple page the view may be nothing more than a Rails template to combine HTML and ERb, for more complex pages of a modular design that we're discussing, we can use the view to instruct Rails how to build every panel in the layout.

If we go back to looking at how a desktop style program would be written, I said that a partial is like a view method. Each partial plays a small role in creating some specific part of the display. If so, then what determines which partials get used, and what data get passed to them? In object orient programming, these kinds of processes are often called view logic. It's not logic which determines which data gets acted on or how, but rather it is logic which determines how the views will be assembled. If certain data from the controller is available, then a portion of the screen gets that data drawn to it, otherwise a default display or no display is rendered. These types of decisions are often placed in a view controller. It's a controller, but with the specific job of determining how views are built, not how data is collected.

In Rails, the natural place to put this code turns out to be the view. Just as with our partials-is-a-method comparison, it's not necessarily evident that the view is the place to do this because we're introduced to the view file as the place to put our HTML. It is, but the way Rails works, it also provides tools in the view to help us build content for other panels as well.

We've already recognized that Rails processes the view file before it processes the layout. Rails takes the view then wraps the layout around it. It doesn't take the layout then insert the view. This order of operations is very deliberate. It enables the view file as the focal point for determining what content will be created, and defining where it will go in the layout, which is exactly the type of control we identified above that we need.

OK, so enough background. I said Rails combines the `content_for` command, and the Ruby `yield` command, so let's have a closer look at how these are used.

If you read AWDWR closely, you came across a discussion about `content_for`. I admit, my first reading of `content_for` in AWDWR didn't strike me as particularly brilliant. In my layout oriented world, it made no sense at all that I would be defining the content for layout section Y from within a file that was intended for layout section X. However, when I recognized the larger role of the view file it become clearer why this is.

Using content_for

Within the view file, the `content_for` command can be used to label a fragment of HTML/ERb code which Rails will store to be output in another place. Let's look at a small example just get some bearings. Let's say the following is the contents of a view file.

```
<p>This code is in the default scope of the
view file. By default, this code will be
displayed as an output of the action.</p>

<% content_for(: right) do %>
   <p>This code is being captured and stored
with a symbol name of :left. It will not be
displayed by calling the action unless we
refer to it expressly in a layout.</p>
<% end %>
```

If we did no other work than create a controller, give it no method, then create a view file named index.rhtml with the above contents, then calling a URL of that controller will show us the string in that first <p>, but not what is in the second <p>. To show the contents wrapped by the Ruby block which is named `:right`, we use a layout.

You likely already know that when using a layout, you place the view file content into the layout by using `yield :layout` (seems like it should be `yield :view`, but for whatever reason it isn't). If we place a `yield :right` somewhere in a layout, for the above view file, then we'll see the contents of that second <p>.

Our layout might look like this then:

```
<div>
   <%= yield :layout %>
</div>
<div>
   <%= yield :right %>
</div>
```

OK, so now we have a method of generating content and labeling it to be inserted into a layout with using the render `command`. What all can we do with `content_for`? We can do anything! Can we use `render` to call a partial? Yes!

So, getting a little more creative we can do something like this in our view:

```
<p>This code is in the default scope of the
view file. By default, this code will be
displayed as an output of the action.</p>

<% content_for(:right) do %>
   <%= render(:partial => "aboutus_right" %>
<% end %>
```

That, of course, will render the about_right partial into the block labeled `:right` which can be inserted into the layout using `yield`. Naturally, we can now revisit the issue of passing data, and end up with this:

```
<% content_for(:right) do %>
<%= render(
      :partial => "newsHeadlines",
      :locals => {
         :headlines => @topStories})
%>
<% end %>
```

Now, remember our context. This code would be in a view file linked directly to the action of our controller. We are allowed (by Rails, and by the OOP police) to use all the instance variables of the controller in the view. With this technique we are achieving all our goals: a) we are making the newsHeadlines partial reusable by having it use only on local variables internally, b) we are abstracting the data being passed to the newsHeadlines partials through `:locals`, and c) we can designate content to go within a specific panel of the layout. Sweet!

Just to make sure it's clear, the `content_for` block can be comprised of any HTML and ERb code. So, we could have something like this:

```
<% content_for(:right) do %>
<h2>Top Stories</h2>
<%= render(
      :partial => "newsHeadlines",
      :locals => {
         :headlines => @topStories})
%>
<% end %>
```

We talked about using the newsHeadlines for both a top stories display and a recent stories display. Same display format but different content and contexts. By using `content_for`, we can push the common parts of that display into the partial, and leave the unique parts like the `h2` title in the view file which is where it belongs. Therefore, on another page, we might reuse the newHeadlines partials in a view with code like this:

```
<% content_for(:left) do %>
<h2>Most Recently Posted Stories</h2>
<%= render(
      :partial => "newsHeadlines",
      :locals => {
         :headlines => @recentStories})
%>
<% end %>
```

We're able to reuse the partial, define content unique to the panel, and for that matter even reuse the partial twice on the same page with with different content and have it inserted in different areas of the page.

Contingency Plans

A couple details before putting it all together in a final plan for page assembly.

Sometimes you may have a panel that has the same content 8 out 10 pages. If so, you don't want to be declaring the `content_for` on those 8 pages when it is the same stuff.

Sometimes you'll have pages where most of the time you want the `:right` panel, but there's a few pages where you don't.

It would be nice if the layout could be flexible enough to recognize when we have dynamic content for a panel and when we have static content, and even when there's no partial to include at all. We can do that with a tad of cleverness in `yield` statement in the layout. If we expand the `yield` statement to something like, we get three possible options (it's a little messy being squished into this column).

```
<%=
(yield :left) ||
(render (:partial =>
   (controller.action_name  + "_left"))
rescue nil)
%>
```

What this does is first attempt to insert a block named `:left` which we can define to in the view file. If there is no `:left` block available, Rails will look for an rhtml file named with the action and a trailing _left string. If that file cannot be found, then Rails will do nothing. With a single statement, we gain some flexibility which can be implemented uniquely for each page.

Putting it All Together

In Figure 4 below, our page assembly schematic has been finalized to reveal the use of `yield` in the template, and the use of `content_for` in the view. Each panel is show a little differently to highlight some possible options (not every panel has to be put together the same way). This schematic reflects some of the details of the example source code too.

Source Code Example

When you found this article, you should have also found a link to a source code download[2]. See footnotes for links if needed. There's a few more details about the source code that are probably worth mentioning.

In the code fragments throughout this article I refer to the /_panels/ folder. In my apps, I'll always have an object which is hash of paths within the application. I use that object to refer to all paths. This makes it much easier to move folders around and rename them when needed. You'll find these paths declared in /lib/site_paths.rb, and loaded by /app/models/application.rb as `@gSitePaths`.

Similarly, I'll keep details together about a page such as its title, the header and footer partial to use, and others depending on the app, in an object called `@gCurrentPage`. The base declarations are in the class file in /lib, but the details are redeclared for each page.

Using :main instead of :layout

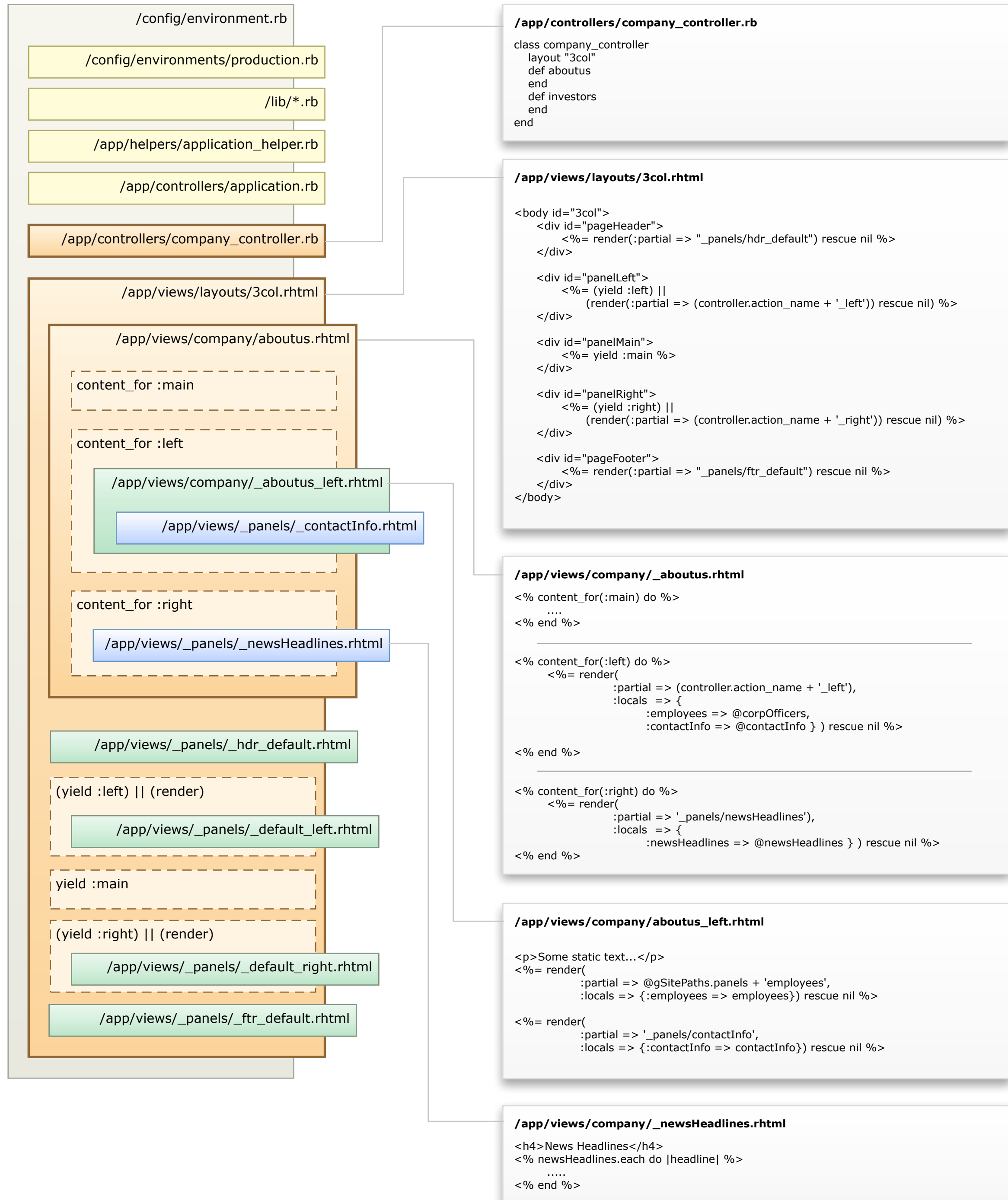
I know the convention is to let the primary output of the view file generate content without the use of a `content_for`, then use `yield :layout`, but I prefer to be explicit about labeling that content as `:main` in the layout. It propagates a consistency in how we assemble output for each layout panel.

If we use `main` in the layout CSS names, it's nice to see `:main` in the view file to know what goes where just like we would do for all the other panels. This way there is one convention instead of two by not having to mentally recognize Rails' method of filling `(yield :layout)` with the output

of the uncaptured portions of the view. I'm just clarifying what I do. You can do what you prefer.

Layouts, Partials, and content_for

In the source code, I spread things around to show how to include them in various ways. The choice of whether to put content within a content_for block, a first tier panel, or a partial that goes inside a panel is ultimately dictated by a) at what level it needs to be reusable, b) your preferred organization. In the source code, some things might make sense in other places, so please remember that I'm trying to show *how* to do some things not necessarily *where* these things best belong.



[1] Dave Thomas, David Heinemeier Hansson, Agile Web Development with Rails, Pragmatic Bookshelf, 2006

[2] http://www.gregwillits.ws/articles/modular-page-assembly-in-rails