Extracted from:

The Rails View

Creating a Beautiful and Maintainable User Experience

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Rails View

Create a Beautiful and Maintainable User Experience

John Athayde
and Bruce Williams

Edited by Brian P. Hogan

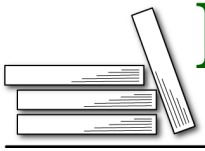
The Facets of Ruby Series

The Rails View

Creating a Beautiful and Maintainable User Experience

John Athayde
Bruce Williams

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian Hogan (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-687-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

6.1 Presenting a Record

Let's put together the presenter class to more easily expose the status data related to our Designer model. We call it DesignerStatus, since that's what it is, and to initialize it, we just pass in the Designer instance. We'll put it in lib/designer_status.rb:

```
class DesignerStatus
  def initialize(designer)
    @designer = designer
  end
end
```

Our DesignerStatus inherits directly from Ruby's default Object class. While it's easy to become accustomed to using the classes that Rails provides, we're not limited to them. Just like Rails itself, we can build our own classes any time we like.

The data we need to pull together for the view is pulled from some associations on the designer. We add a few methods to our class:

```
def active_projects_count
  active_projects.count
end

def pending_approvals_count
  active_creations.pending_approval.count
end

def approved_count
  active_creations.approved.count
end

def active_hours
  active_projects.total_hours
end

def hours_per_project
  active_projects.inject({}) do |memo, project|
    memo[project] = project.total_hours
    memo
  end
end

private

def active_projects
  @designer.projects.active
end
```

```
def active_creations
  @designer.creations.active
end
```

Our presenter only displays information on the active projects and creations for the designer, so we've created a couple of private methods, active_projects() and active_creations(), that handle getting that information for us. This way we won't need to have the same method chaining repeated in the methods we'll be calling from our template.

Now we need to instantiate our DesignerStatus presenter for use in our template. Sometimes it makes sense for the controller to set up the presenter, especially in cases where the presenter needs to be configured with session or request parameters. In this case, however, we prefer to instantiate our presenter in a helper because it's purely a view concern: it's only used from a template and it doesn't need any additional information about the request. The controller doesn't necessarily need to retrieve or instantiate every single object a template might need. Here it's the view's job. We'll use a helper method we'll put in app/helpers/designers_helper.rb to create the presenter instance.

```
module DesignersHelper
  def designer_status_for(designer = @designer)
    presenter = DesignerStatus.new(designer)
    if block_given?
      yield presenter
    else
      presenter
    end
  end
end
```

This helper takes an optional designer and defaults to the current @designer if it's not provided. When we're in an action template focused on a single designer (like the show() action of DesignersController), using this keeps our template brief and it doesn't lock us out of cases where we'd want to display status information for multiple designers on a single template, since we can just pass in the specific Designer record whenever we need it.

Once we instantiate our presenter, we yield it to the block if we can, which would let us invoke methods repeatedly on the presenter without having to assign it in the template (we *don't* do that, as we covered in Chapter 2, *Improving Readability*, on page ?).

Now that we have our presenter instance, let's put together the Designer Dashboard view that uses it, which is rendered by the DesignersController show() action from app/views/designers/show.html.erb:

```
<% designer_status_for do |status| %>
  <section class='designer-status'>
    <title>Status</title>
    <dl>
      <dt>Active Projects</dt>
      <dd><%= status.active_projects_count %></dd>
      <dt>Pending Approval</dt>
      <dd><%= status.pending_approvals_count %></dd>
      <dt>Approved</dt>
      <dd><%= status.approved_count %></dd>
    </dl>
    <h3>Active Project Hours</h3>
    <table>
      <tr>
        <th>Project</th>
        <th>Hours</th>
      </tr>
      <% status.hours_per_project.each do |project, hours| %>
        <tr>
          <td><%= link_to project.name, project %></td>
          <td><%= hours %></td>
        </tr>
      <% end %>
      <tr>
        <th>Total</th>
        <td><%= status.active_hours %></td>
      </tr>
    </table>
  </section>
<% end %>
```

This is great! We have all of these helpers bundled together into one unit without muddying our model or losing them in the crush of methods in our helper modules.

How can we support this more generically and make using the presenter elsewhere in the application as easy as possible? As we said before, we'd like to display this information in other places, too, but we'd like the information to be more condensed. Let's extract the designer status markup out of our show.html.erb into a partial, _status.html.erb, and add a condition to determine if we want the "expanded" view that includes our hourly breakdown by project. We'll also remove the designer_status_for() call since we won't need it; we'll be passing in our DesignerStatus instance when we render the partial, instead of creating it there.

artflow/presenters/app/views/designers/_status.html.erb

```
<section class='designer-status'>
  <title>Status</title>
  <dl>
```

Joe asks:
When Should I Use a Presenter?

Here are a few signs a part of your view could be better built or refactored as a presenter:

- It displays specialized, complex data for a record or an aggregation of records, especially if it requires grouping, sorting, calculations, or transformation to new data structures for view-specific iteration.
- It uses several interrelated helpers, especially if they call each other, pass around some type of shared state, are grouped together by a common prefix, or have been considered cohesive enough to be extracted into a separately named helper module.
- It's displayed by an action whose authentication or other environmental constraints would make testing the view difficult or slow.

```
    <dt>Active Projects</dt>
    <dd class='active-projects'><%= status.active_projects_count %></dd>
    <dt>Pending Approval</dt>
    <dd class='pending-creations'><%= status.pending_approvals_count %></dd>
    <dt>Approved</dt>
    <dd class='approved-creations'><%= status.approved_count %></dd>
  </dl>
➤ <% if status.expanded? %>
    <h3>Active Project Hours</h3>
    <table>
      <tr>
        <th>Project</th>
        <th>Hours</th>
      </tr>
      <% status.hours_per_project.each do |project, hours| %>
      <tr>
        <td><%= link_to project.name, project %></td>
        <td><%= hours %></td>
      </tr>
      <% end %>
      <tr>
        <th>Total</th>
        <td><%= status.active_hours %></td>
      </tr>
    </table>
➤ <% end %>
</section>
```

We need to add an `expanded?()` method to our `DesignerStatus` and support an options hash passed to our initializer:

artflow/presenters/lib/designer_status.v3.rb

```
def initialize(designer, options = {})
  @designer = designer
  @options = options
end

def expanded?
  @options[:expanded]
end
```

In expanded?() we just look for a non-nil or non–false :expanded option. Let's update our helper to accept additional options and pass them along. We'll make the default non-expanded, since usually we'll want the short status displayed:

```
def designer_status_for(designer = @designer, options = {})
  presenter = DesignerStatus.new(designer, options)
  if block_given?
    yield presenter
  else
    presenter
  end
end
```

Now we can change how our show.html.erb template renders the presenter now using the partial. We pass along the presenter instance using the :object option, which will make sure it's assigned to a variable with the same name as the partial (in this case, status):

artflow/presenters/app/views/designers/show.v2.html.erb

```
<%= render partial: 'status',
           object:   designer_status_for(@designer, expanded: true) %>
```

We can go even farther than this, tossing out the need for a render() in our template at all. We can make the DesignerStatus render *itself*! To do this, our class needs access to the template instance. This isn't a problem, since our helpers are executed in the context of the view; self is what we need to give our presenter. We edit our designer_status_for() helper and pass it along:

artflow/presenters/app/helpers/designers_helper.rb

```
  def designer_status_for(designer = @designer, options = {})
    presenter = DesignerStatus.new(designer, self, options)
    if block_given?
      yield presenter
    else
      presenter
    end
  end

end
```

Now the DesignerStatus initialize() method needs to be modified to accept the template argument:

artflow/presenters/lib/designer_status.rb
```
def initialize(designer, template, options = {})
  @designer = designer
  @template = template
  @options = options
end
```

Now that our presenter has the template instance, what can we do with it? Well, let's look at how we *want* to add the markup for the designer status from our template:

artflow/presenters/app/views/designers/show.html.erb
```
<%= designer_status_for(@designer, expanded: true) %>
```

Wow, that's short! What's going on here?

When we insert content with ERB, it automatically calls to_s() (read: "to string") on the content first. Let's define that method on our DesignerStatus presenter so that inserting our presenter will work out of the box:

artflow/presenters/lib/designer_status.rb
```
def to_s
  @template.render partial: 'designers/status', object: self
end
```

It's just as easy to generate the condensed version of our designer status elsewhere, as we do on the page for a project, showing the status for the designers styled as a badge:

artflow/presenters/app/views/projects/show.html.erb
```
<ul>
  <% @project.designers.each do |designer| %>
    <li><%= designer_status_for(designer) %></li>
  <% end %>
</ul>
```

Keep in mind we don't *need* to use the rendering shortcut or even need to use the partial at all. We can use designer_status_for() at any point, in any template, and extract any of the bits of data we need directly by calling methods on the DesignerStatus instance. We could support more options in our presenter, hide and show additional information, or even render an entirely different partial based on some criteria. Presenters can be amazingly flexible pieces of machinery.

Testing Template Presenters

There are a few aspects of these presenters that make sense to test. We should test the presenter instances themselves to make sure they're accurately extracting the data from the related records. We should also make sure users are seeing what we expect; that the helper creating the presenter instance behaves correctly, and that the template for the presenter displays the information as we'd like it to.

Let's focus on the DesignerStatus presenter, helper, and template that we put together in Section 6.1, *Presenting a Record*, on page 5, and look at how we might build our tests. We'll move from the core behavior of the presenter out to what the user sees.

Since presenters are just plain old Ruby objects, we can test them with a plain unit test. We'll use an ActiveSupport::TestCase, since it gives us some niceties (like String test names):

artflow/presenters/test/unit/designer_status_test.rb

```
require 'test_helper'
class DesignerTest < ActiveSupport::TestCase
  def setup
    setup_designer
    @status = DesignerStatus.new(@designer, nil)
  end

  test 'DesignerStatus instance calculates active projects' do
    assert_equal 3, @status.active_projects_count
  end

  test 'DesignerStatus instance calculates hours' do
    assert_equal [2, 2, 2], @status.hours_per_project.values
    assert_equal 6, @status.active_hours
  end
end
```

Here we instantiated our DesignerStatus just as our helper would, except we pass in nil instead of a template or a fancy mock. We're not testing the to_s() method: the template won't be tested.

In the test's setup() we create a Designer and related Project and Creation records by calling a method, setup_designer(), that we defined in our test helper:

artflow/presenters/test/test_helper.rb

```
require 'factories'
class ActiveSupport::TestCase

  # We don't use fixtures, so we comment this out:
  # fixtures :all
```

```
  def setup_designer
    @designer = Factory(:designer)
    3.times do
      creation = Factory(:creation, hours: 2, designer: @designer)
      @designer.projects << creation.project
    end
    @designer.save
  end

end
```

This setup_designer() utility method builds our objects using factory_girl, a test fixture library we prefer to Rails's built-in, static YAML-based fixtures.[1] Static fixtures are fine, but it's nice to be able to dynamically generate fixture data at will, trying out different combinations of data and using shortcuts like the Faker gem to give it a little variety.[2] Here are the definitions we're using and loading from factories.rb:

artflow/presenters/test/factories.rb

```
Factory.define :designer do |x|
  x.sequence(:email) { |n| "designer#{n}@artflowme.com" }
  x.password 'testtest'
end

Factory.define :project do |x|
  x.sequence(:name) { |n| "Project #{n}" }
  x.association :campaign
  x.active true
end

Factory.define :campaign do |x|
  x.sequence(:name) { |n| "Campaign #{n}" }
end

Factory.define :creation do |x|
  x.sequence(:name) { |n| "Creation #{n}" }
  x.association :project
  x.association :designer
  x.stage 'initial'
  x.revision 1
  x.description "This is a description"
end
```

With our prepopulated designer, we can test our active_projects_count(), hours_per_project(), and active_hours() methods to make sure they extract the data we expect from our record. We can build on this as the data we need to display

1. https://github.com/thoughtbot/factory_girl
2. http://rubygems.org/gems/faker

grows, and having these tests around will help prevent regression in the future; it seems likely time tracking and reporting will become more and more complex as our project grows.

Now let's make sure our helper behaves correctly. We'll do this with a ActionView::TestCase unit test:

artflow/presenters/test/unit/helpers/designers_helper_test.rb
```
require 'test_helper'

class DesignersHelperTest < ActionView::TestCase

  def setup
    setup_designer
    @status = DesignerStatus.new(@designer, nil)
  end

  test 'designer_status_for helper returns a DesignerStatus instance' do
    assert_kind_of DesignerStatus, designer_status_for(@designer)
  end

  test 'designer_status_for helper yields a DesignerStatus instance' do
    yielded = nil
    designer_status_for(@designer) { |obj| yielded = obj }
    assert_kind_of DesignerStatus, yielded
  end

end
```

So far we're just concerned with making sure the helper returns or yields the DesignerStatus, but we can test the presenter to_s() method, too; since ActionView::TestCase sets up a template for us, we can use render()! We'll check that it's working as expected by checking a bit of the resulting content:

artflow/presenters/test/unit/helpers/designers_helper_test.rb
```
test 'calling to_s returns status markup' do
  status = designer_status_for(@designer)
  assert status.to_s.include?('<title>Status</title>')
end

test 'non-expanded status markup does not include active hours' do
  status = designer_status_for(@designer)
  assert !status.to_s.include?('Active Project Hours')
end

test 'expanded status markup includes active hours' do
  status = designer_status_for(@designer, expanded: true)
  assert status.to_s.include?('Active Project Hours')
end
```

We're careful not to test too much of the markup. We want to avoid writing brittle tests that will break unnecessarily the next time someone tweaks the look and feel of the designer status widgets. Instead of ensuring that the structure of the returned markup meets today's expectations, we focus on verifying important pieces of information that are more likely to stand the test of time.

Let's add a quick test for our DesignersController show() action, where we display the "expanded" designer status. We'll limit our assertions to verifying that the presenter is rendered and just check the number of active projects displayed for our designer.

artflow/presenters/test/functional/designers_controller_test.rb

```
require 'test_helper'

class DesignersControllerTest < ActionController::TestCase

  def setup
    setup_designer
  end

  test "should render designer status presenter" do
    get :show, id: @designer.id
    assert_response :success
    assert_select 'section.designer-status .active-projects', text: '3'
  end

end
```

Once again we don't want to exhaustively test the structure of the markup, and since our unit tests will check the accuracy of the data our presenter extracts from the record, there's no need to double-check it here. Verifying the presenter is displayed for the designer is enough and is the best "bang for our buck."

Now that we've used a presenter to show information from one record, let's look at how we can use it to help us deal with aggregations of records.