CLEANER, SCALABLE VIEWS WITH OBJECT ORIENTED COMPONENTS

Hi!

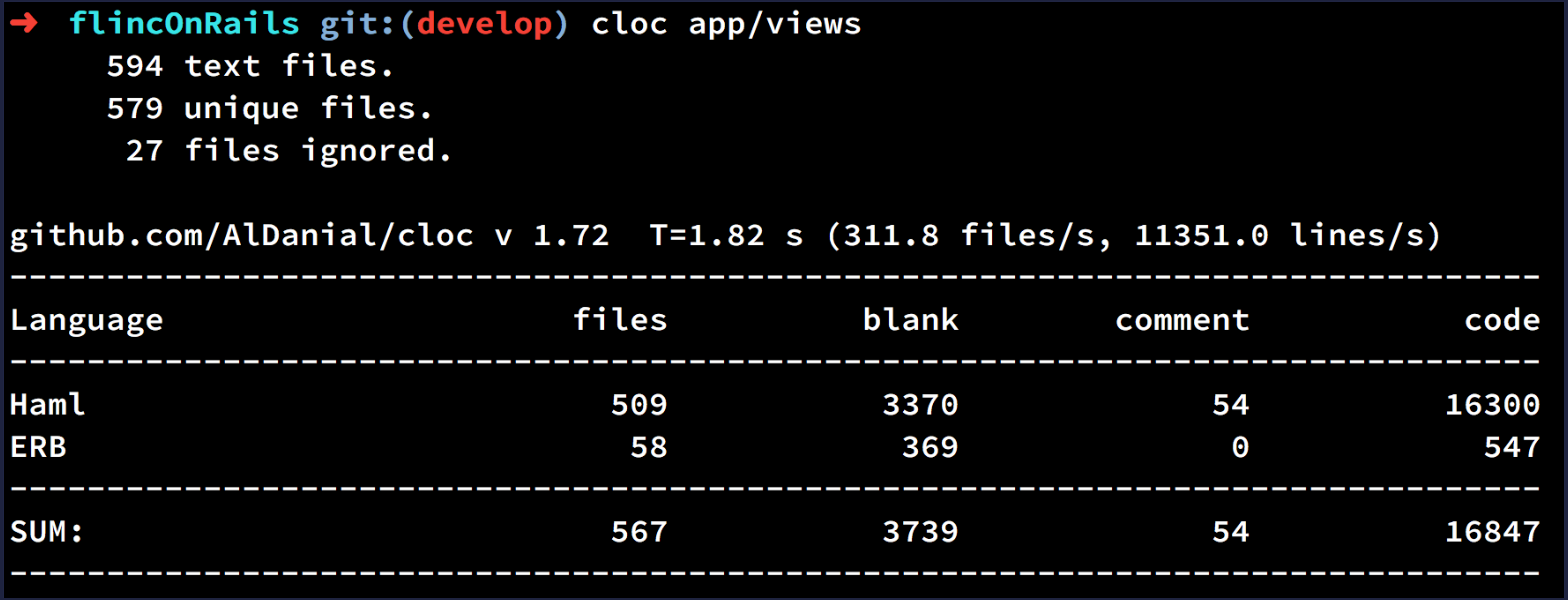
CHRISTIAN BÄUERLEIN

TWITTER.COM/FABRIK42 🐱 GITHUB.COM/FABRIK42 🐱 FABRIK42@GMAIL.COM

CLEANER, SCALABLE VIEWS WITH OBJECT ORIENTED COMPONENTS

Focus today: Scaling for maintainability

THE PROBLEM



```
➜  flincOnRails git:(develop) cloc app/views
     594 text files.
     579 unique files.
      27 files ignored.

github.com/AlDanial/cloc v 1.72  T=1.82 s (311.8 files/s, 11351.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Haml                           509           3370             54          16300
ERB                             58            369              0            547
-------------------------------------------------------------------------------
SUM:                           567           3739             54          16847
-------------------------------------------------------------------------------
```

The main platform has almost 600 views/partials.

VIEWS ARE HARD

Views are often the messiest part of a Rails application.

When using markup, the representation **is** the implementation.

When is it "just markup", when does it become "code duplication"?

Often, helpers and partials are fine,
but they fail with more complex configurations.

No explicit interface definition

Views often fail silently.

No `NoMethodError` for typos in CSS classes.

Yes, it's just HTML and CSS, but:
Components are often very complicated to configure.

How many possible variants does a button in your UI have?

OUR SOLUTION: ACTION WIDGET!

ActionWidget provides a lightweight and consistent way to define interface components for Ruby on Rails applications.

https://github.com/t6d/action_widget

A SIMPLE EXAMPLE

We want to create the following link button:

```
<a class="btn btn-small" href="/login">Login</a>
```

```ruby
class ButtonWidget < ActionWidget::Base
  property :caption,
    converts: :to_s,
    required: true

  property :target,
    converts: :to_s,
    accepts: lambda { |uri| URI.parse(uri) rescue false },
    required: true

  property :size,
    converts: :to_sym,
    accepts: [:small, :medium, :large],
    default: :medium

  def render
    content_tag(:a, caption, href: target, class: css_classes)
  end

protected

  def css_classes
    css_classes = ['btn']
    css_classes << "btn-#{size}" unless size == :medium
    css_classes
  end
end
```

```erb
# calling the class directly (self is an instance of ActionView)
<%= ButtonWidget.new(self, caption: 'Login', size: :small, target: '/login').render %>

# using the convenience helper
<%= button_widget caption: 'Login', size: :small, target: '/login' %>

# output
<a class="btn btn-small" href="/login">Login</a>
```

Passing blocks

```ruby
class PanelWidget < ActionWidget::Base
  property :title, required: true, converts: :to_s

  def render(&block)
    content_tag(:div, class: 'panel') do
      content_tag(:h2, title, class: 'title') +
        content_tag(:div, class: 'content', &block)
    end
  end
end
```

```
<%= panel_widget title: "Important Notice" do %>
  The system will be down for maintenance today.
<% end %>

# becomes
<div class="panel">
  <h2 class="title">Important Notice</h2>
  <div class="content">
    The system will be down for maintenance today.
  </div>
</div>
```

Nested Widgets

```erb
<%= menu_widget do |m| %>
  <%= m.item "Dashboard", "/" %>
  <%= m.submenu "Admin" do |m| %>
    <%= m.item "Manage Users", "/admin/users" %>
    <%= m.item "Manage Groups", "/admin/groups" %>
  <% end %>
<% end %>
```

Inheritance

```
class SidebarPanelWidget < PanelWidget
  def header
    content_tag(:h3, title)
  end
end
```

Input validation

```
class ButtonWidget < ActionWidget::Base
  property :caption,
    converts: :to_s,
    required: true

  property :target,
    converts: :to_s,
    accepts: lambda { |uri| URI.parse(uri) rescue false },
    required: true

  property :size,
    converts: :to_sym,
    accepts: [:small, :medium, :large],
    default: :medium

  def render
    content_tag(:a, caption, href: target, class: css_classes)
  end

protected

  def css_classes
    css_classes = ['btn']
    css_classes << "btn-#{size}" unless size == :medium
    css_classes
  end
end
```

Unit Testing

```ruby
describe 'WidgetHelper#button_widget', type: :helper do
  subject do
    helper.button_widget(target: '/', title: 'Home')
  end

  it 'renders a link with correct classes' do
    subject.should have_selector('a.btn')
  end

  it 'renders the caption "Home"' do
    subject.should have_content('Home')
  end
end
```

USE CASES

Where it paid off

Use Case

Migrating to *Twitter Bootstrap*

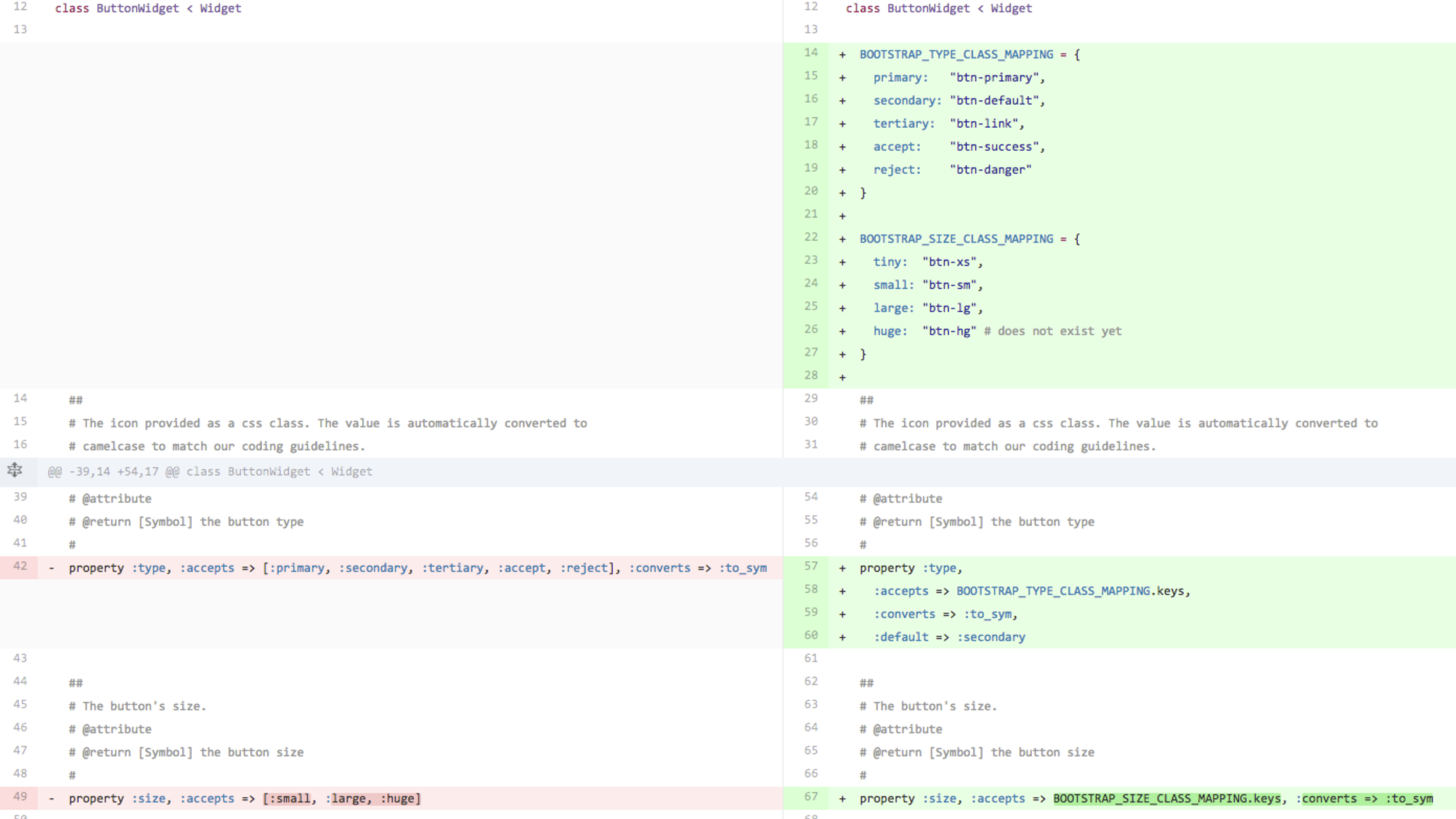
Fun Fact

We have 466 buttons in our view code

😰

```
# BUT: we did not have buttons as markup
<a class="btn btn-default primary large" href="/login">Login</a>

# all our buttons were already button widgets
<%= button_widget title: 'Example', type: :primary, size: :large %>
```

```ruby
class ButtonWidget < ActionWidget::Base

  # ...

  property :size,
    converts: :to_sym,
    accepts: [:small, :medium, :large],
    default: :medium

  def render
    content_tag(:a, caption, href: target, class: css_classes)
  end

protected

  def css_classes
    css_classes = ['btn']
    css_classes << "btn-#{size}" unless size == :medium
    css_classes
  end
end
```



```diff
  class ButtonWidget < Widget

+   BOOTSTRAP_TYPE_CLASS_MAPPING = {
+     primary:   "btn-primary",
+     secondary: "btn-default",
+     tertiary:  "btn-link",
+     accept:    "btn-success",
+     reject:    "btn-danger"
+   }
+
+   BOOTSTRAP_SIZE_CLASS_MAPPING = {
+     tiny:  "btn-xs",
+     small: "btn-sm",
+     large: "btn-lg",
+     huge:  "btn-hg" # does not exist yet
+   }
+
    ##
    # The icon provided as a css class. The value is automatically converted to
    # camelcase to match our coding guidelines.
@@ -39,14 +54,17 @@ class ButtonWidget < Widget
    # @attribute
    # @return [Symbol] the button type
    #
-   property :type, :accepts => [:primary, :secondary, :tertiary, :accept, :reject], :converts => :to_sym
+   property :type,
+     :accepts => BOOTSTRAP_TYPE_CLASS_MAPPING.keys,
+     :converts => :to_sym,
+     :default => :secondary

    ##
    # The button's size.
    # @attribute
    # @return [Symbol] the button size
    #
-   property :size, :accepts => [:small, :large, :huge]
+   property :size, :accepts => BOOTSTRAP_SIZE_CLASS_MAPPING.keys, :converts => :to_sym
```

Use Case

Encapsulate widget specific logic, hiding complexity

Rendering the avatar image of a user

```
avatar_widget size: :medium,
  image: user.avatar,
  title: user.screen_name,
  target: user_path(user)
```

But, this is more than an `<img/>` tag.

Logic needed, for e.g.
size specific stylings,
the default avatar or
a loading indicator
(if the avatar is being processed)

Logic is wrapped in `AvatarWidget` class, instead of shattered across views and helpers.

```
avatar_widget size: :medium,
  image: user.avatar,
  title: user.screen_name,
  target: user_path(user)
```

~~Use Case~~ Confession

We prerender clientside *Handlebars.js* templates in *HAML*, also using *ActionWidgets*, before we precompile it to *JavaScript* functions via *Node*.

This is fine.

Use Case

Rendering *Angular.js* views in a *Middleman* static page app

```
= form_widget prefix: 'user' do |f|
  = f.email_field placeholder: 'Email', field: 'email', required: true, size: :large
  = f.password_field placeholder: 'Password', field: 'password', required: true, size: :large
  = f.submit_button on_click: 'signIn(model)', label: "Submit"
  = f.base_errors
```

```html
<form class="form-widget" role="form" name="form">
  <div class="form-group" ng-class="{'has-error': !!form.email.$error.server}">
    <input class="form-control input-lg" id="user-email" name="email" ng-model="model.email" placeholder="Email"
required="" server-error="" type="email">
    <div class="errors" ng-show="form.email.$dirty &amp;&amp; form.email.$invalid">
      <ul class="list-unstyled">
        <li ng-show="form.email.$error.server">
          <i class="fa fa-exclamation-circle"></i>
          {{ formErrors.email }}
        </li>
        <li ng-show="form.email.$dirty &amp;&amp; form.email.$invalid">
          <i class="fa fa-info-circle"></i>
          Du musst eine richtige Email-Adresse angeben
        </li>
      </ul>
    </div>
  </div>
  <div class="form-group" ng-class="{'has-error': !!form.password.$error.server}">
    <input class="form-control input-lg" id="user-password" name="password" ng-model="model.password"
placeholder="Password" required="" server-error="" type="password">
    <div class="errors" ng-show="form.password.$dirty &amp;&amp; form.password.$invalid">
      <ul class="list-unstyled">
        <li ng-show="form.password.$error.server">
          <i class="fa fa-exclamation-circle"></i>
          {{ formErrors.password }}
        </li>
      </ul>
    </div>
  </div>
  <button class="btn btn-lg btn-success " form-submit-button="submit" ng-click="signIn(model)">Submit</button>
  <div class="errors alert alert-danger" ng-show="formErrors.base.length">
    <ul class="list-unstyled">
      <li ng-repeat="error in formErrors.base">
        {{ 'forms.error.' + error | translate }}
      </li>
    </ul>
  </div>
</form>
```

Use Case

Keeping your sanity: *Zurb Ink* in HTML emails

In "browser world", a button might look like this:

```html
<a class="btn btn-primary btn-lg" href="/login">Login</a>
```

But in "HTML email world", everything is at least two nested tables.

```
<table class="button">
  <tr>
    <td>
      <table>
        <tr>
          <td><a href="#">Button</a></td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

This widget provides an interface for the important details, hides the markup complexity from the user.

```
= email_button_widget href: '#' do

  Click here!
```

```
= email_row_widget do
  = email_col_widget width: 12, last: true do
    %h1 Schön, dass du dabei bist!

= email_row_widget do
  = email_col_widget width: 12, last: true do
    %p
      Hallo #{@user.first_name},
    %p
      wir freuen uns sehr, dass du bei flinc bist, deiner Mitfahr-App für jeden Tag.
= email_row_widget class: 'cta' do
  = email_col_widget width: 6, offset: 3, last: true do
    = email_button_widget href: root_url do
      Jetzt geht's los!
```

That's it!

TL;DR

Object oriented widgets solve real problems!

Good in addition to views, partials and helpers.

Do not try to replace all your markup with widgets!

Use widgets for the parts that

- are reused tens or hundreds of times
- are highly configurable
- need complex view specific logic
- have overly verbose markup for little presentation

Thank you!

CHRISTIAN BÄUERLEIN

TWITTER.COM/FABRIK42 🐱 GITHUB.COM/FABRIK42 🐱 FABRIK42@GMAIL.COM