Concerns / Presenters and Decorators

Devpoint Labs - Jake Sorce / Dave Jungst

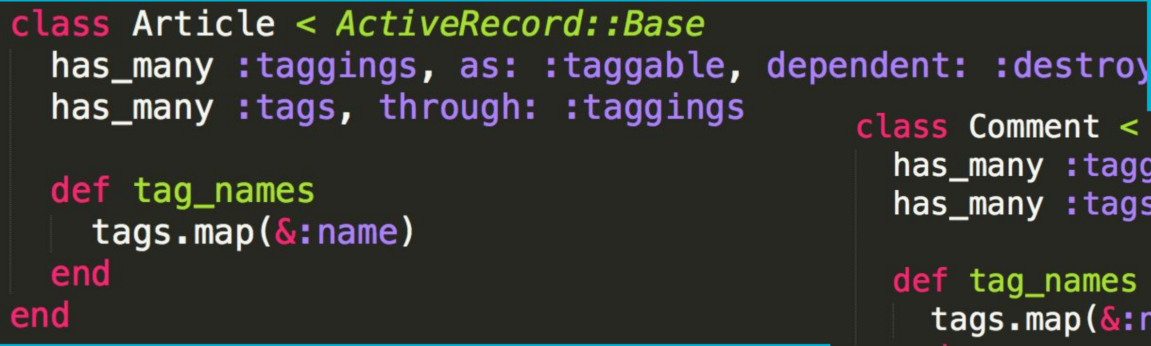
Concerns

When setting up a new Rails 4 application you will notice some auto generated folders:

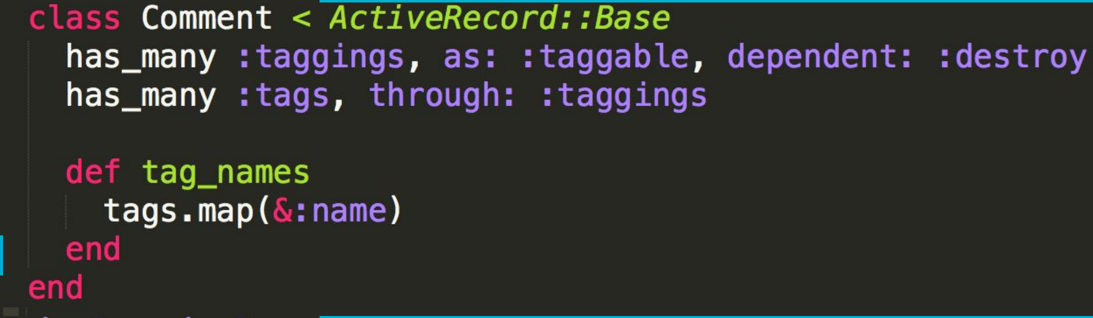
- app/models/concerns
- app/controllers/concerns

Concerns are modules that can be mixed into your models and controllers to share code between them.

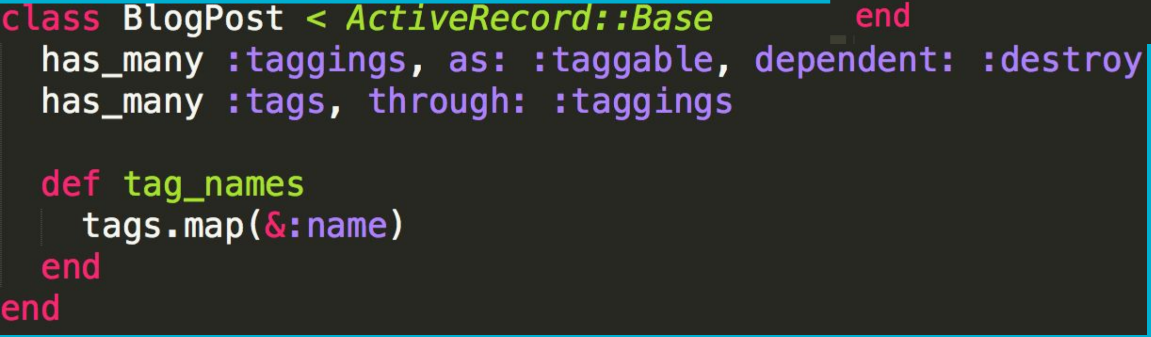
Models - Without Concerns



```ruby
class Article < ActiveRecord::Base
  has_many :taggings, as: :taggable, dependent: :destroy
  has_many :tags, through: :taggings

  def tag_names
    tags.map(&:name)
  end
end
```

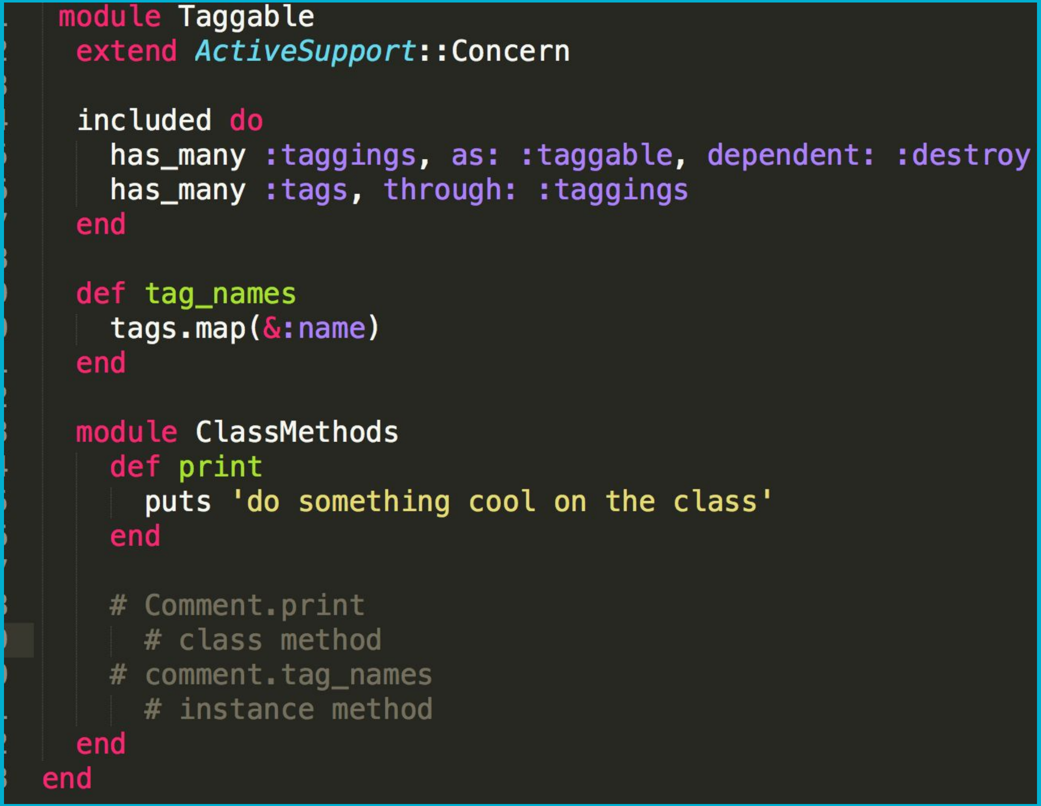


```ruby
class Comment < ActiveRecord::Base
  has_many :taggings, as: :taggable, dependent: :destroy
  has_many :tags, through: :taggings

  def tag_names
    tags.map(&:name)
  end
end
```



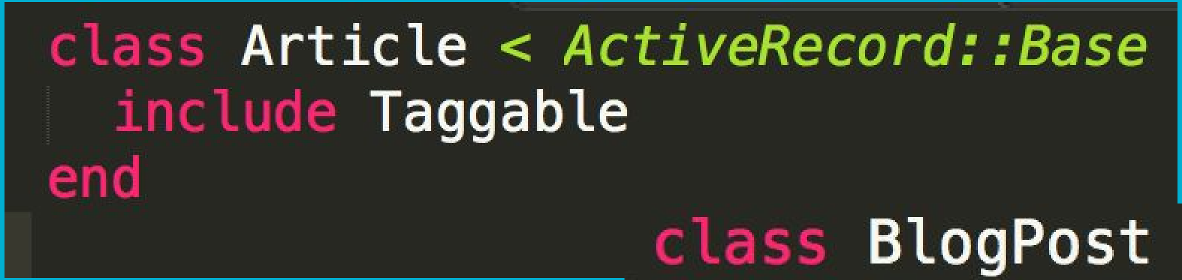
```ruby
class BlogPost < ActiveRecord::Base
  has_many :taggings, as: :taggable, dependent: :destroy
  has_many :tags, through: :taggings

  def tag_names
    tags.map(&:name)
  end
end
```

Models - Concern

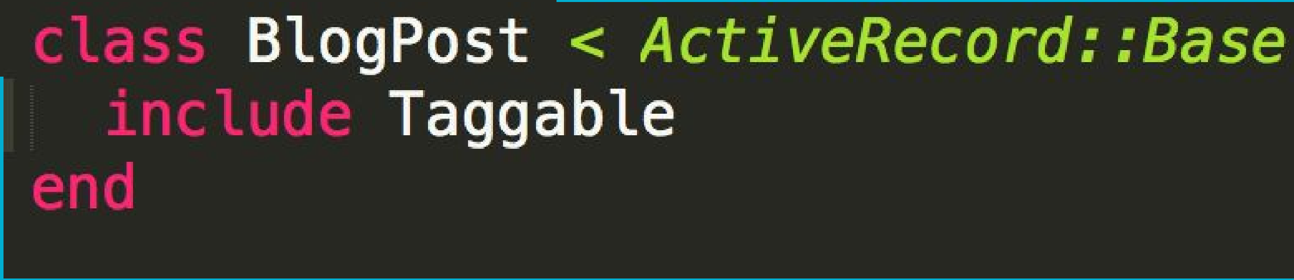


```ruby
module Taggable
  extend ActiveSupport::Concern

  included do
    has_many :taggings, as: :taggable, dependent: :destroy
    has_many :tags, through: :taggings
  end

  def tag_names
    tags.map(&:name)
  end

  module ClassMethods
    def print
      puts 'do something cool on the class'
    end

    # Comment.print
      # class method
    # comment.tag_names
      # instance method
  end
end
```

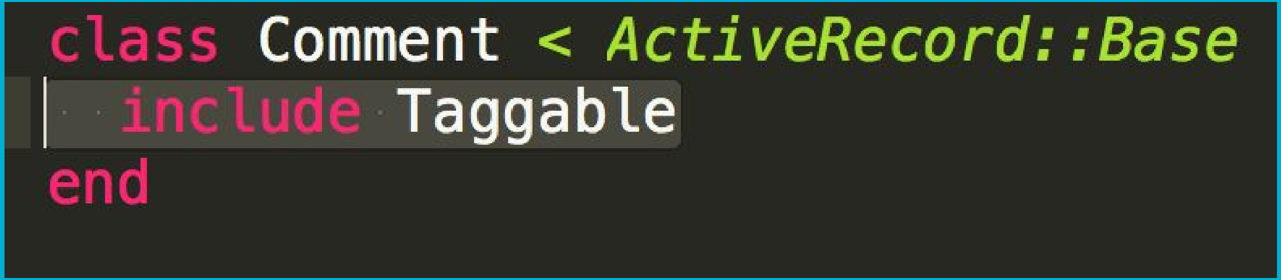
Models - With Concerns



```ruby
class Article < ActiveRecord::Base
  include Taggable
end
```

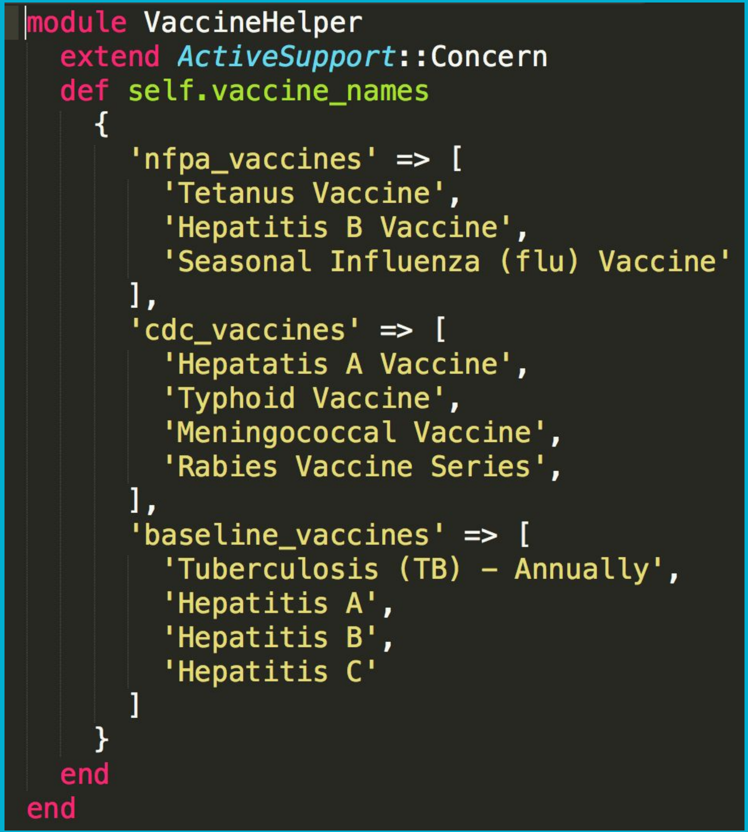


```ruby
class BlogPost < ActiveRecord::Base
  include Taggable
end
```



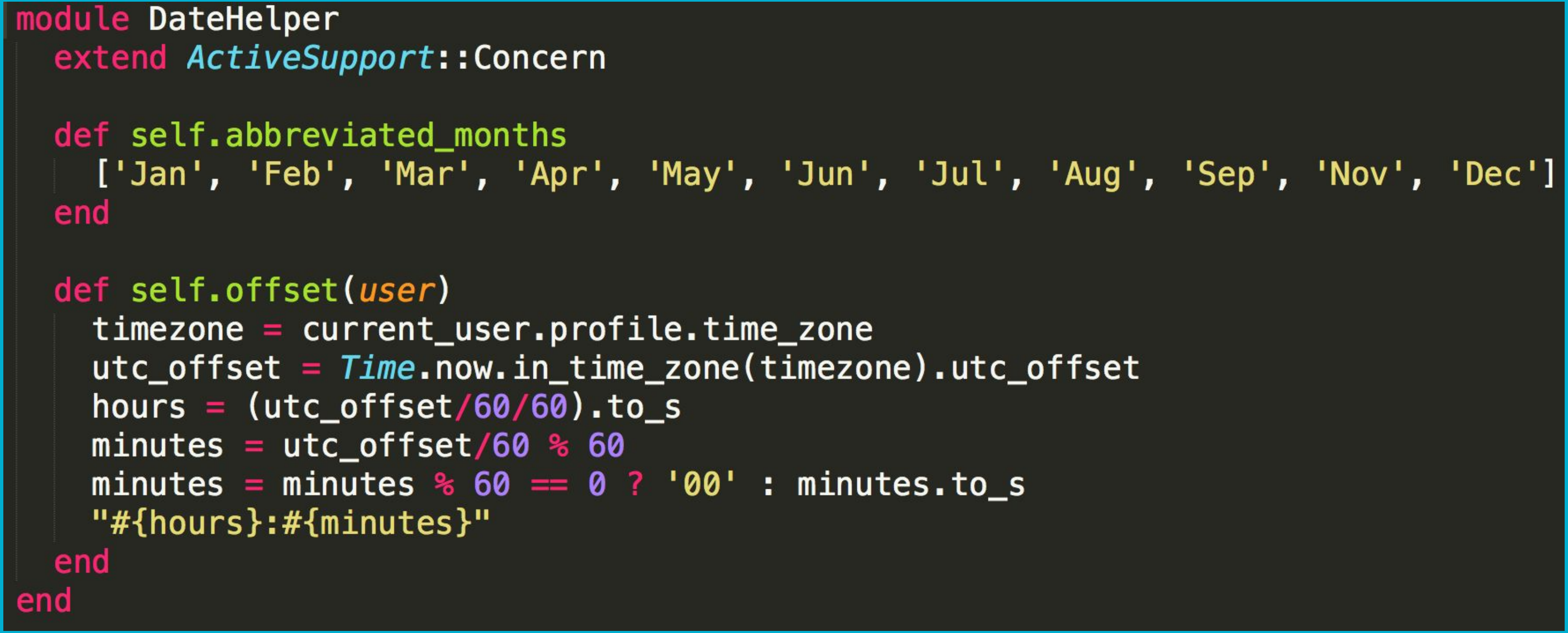
```ruby
class Comment < ActiveRecord::Base
  include Taggable
end
```

Controllers - With Concerns Cont...



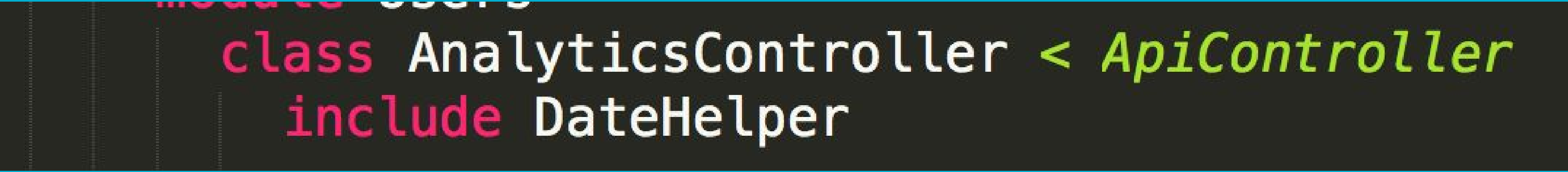
```ruby
module VaccineHelper
  extend ActiveSupport::Concern
  def self.vaccine_names
    {
      'nfpa_vaccines' => [
        'Tetanus Vaccine',
        'Hepatitis B Vaccine',
        'Seasonal Influenza (flu) Vaccine'
      ],
      'cdc_vaccines' => [
        'Hepatatis A Vaccine',
        'Typhoid Vaccine',
        'Meningococcal Vaccine',
        'Rabies Vaccine Series',
      ],
      'baseline_vaccines' => [
        'Tuberculosis (TB) - Annually',
        'Hepatitis A',
        'Hepatitis B',
        'Hepatitis C'
      ]
    }
  end
end
```

Controllers - With Concerns

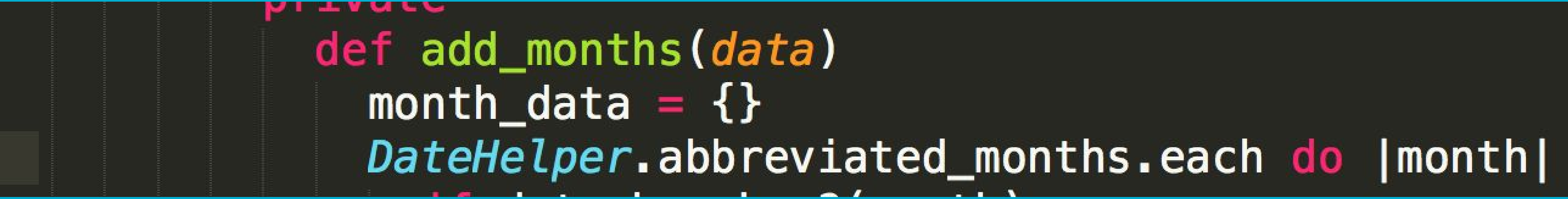


```ruby
module DateHelper
  extend ActiveSupport::Concern

  def self.abbreviated_months
    ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Nov', 'Dec']
  end

  def self.offset(user)
    timezone = current_user.profile.time_zone
    utc_offset = Time.now.in_time_zone(timezone).utc_offset
    hours = (utc_offset/60/60).to_s
    minutes = utc_offset/60 % 60
    minutes = minutes % 60 == 0 ? '00' : minutes.to_s
    "#{hours}:#{minutes}"
  end
end
```

Controllers - With Concerns Cont...



```
  class AnalyticsController < ApiController
    include DateHelper
```



```
    def add_months(data)
      month_data = {}
      DateHelper.abbreviated_months.each do |month|
```

Concern Wrap up

Benefits of Concerns:

- Code can quickly and easily be spread upon as many models as needed.

- Great way to keep your code organized and DRY.
  - Change code in 1 place not multiple

- Additional specific model functionality can be added if needed.

Presenters / Decorators

Presenters and Decorators help clean up the complexity of your views.

Presenters and Decorators are basically the same and we will take out some of the complexity and set up of presenters by using the draper gem to present / decorate our views

Decorators / Presenters

- Draper Gem
- What are decorators / presenters?
- Why do we care?
- When to use decorators / presenters
- Examples

Draper - The Decorator Gem

https://github.com/drapergem/draper

Draper adds an object-oriented layer of presentation logic to your Rails application.

Without Draper, this functionality might have been tangled up in helpers or adding bulk to your models. With Draper decorators, you can wrap your models with presentation-related logic to organise - and test - this layer of your app much more effectively.

Decorators / Presenters - What are they?

The decorator pattern is a design pattern that allows behavior to be added to an individual object without affecting the behavior of other objects from the same class.

Imagine your application has an Article model. With Draper, you'd create a corresponding ArticleDecorator. The decorator wraps the model, and deals only with presentational concerns. In the controller, you decorate the article before handing it off to the view:

Decorators / Presenters - Why do we care?

Your views should be stupid.

I like to use the analogy that views should read similarly to a shopping list. There shouldn't be any complexity or logic. It should read similarly to:

- Email
- Name
- Joined date
- Favorite color

Of course your views will be a little more complicated than this but this is a good way to think about it.

Decorators / Presenters - Why do we care? Cont...

Steve Klabnik stated:

“The whole idea of logic in templates leads to all kinds of problems. They're hard to test, they're hard to read, and it's not just a slippery slope, but a steep one. Things go downhill rapidly.”

Decorators / Presenters - When should we use them?

- Do NOT use them in early stage apps
    - prototyping is always faster and easier

- Your views and models should be growing rapidly, and bursting at the seams before you consider utilizing Draper

- The Decorator/Presenter pattern should be treating painful symptoms like complex views and unmaintainable models, rather than prematurely accounting for them.

Why shouldn’t we just use helper methods?

- Why can’t we just use helper methods to clean our views up?
  - Well we could but all helper methods are in a global namespace and there is nothing object orientated about them at all

- Example
  - Given in slack, presenters_decorators_example_bad
  - Given in slack, presenters_decorators_example_good

Using the Draper gem

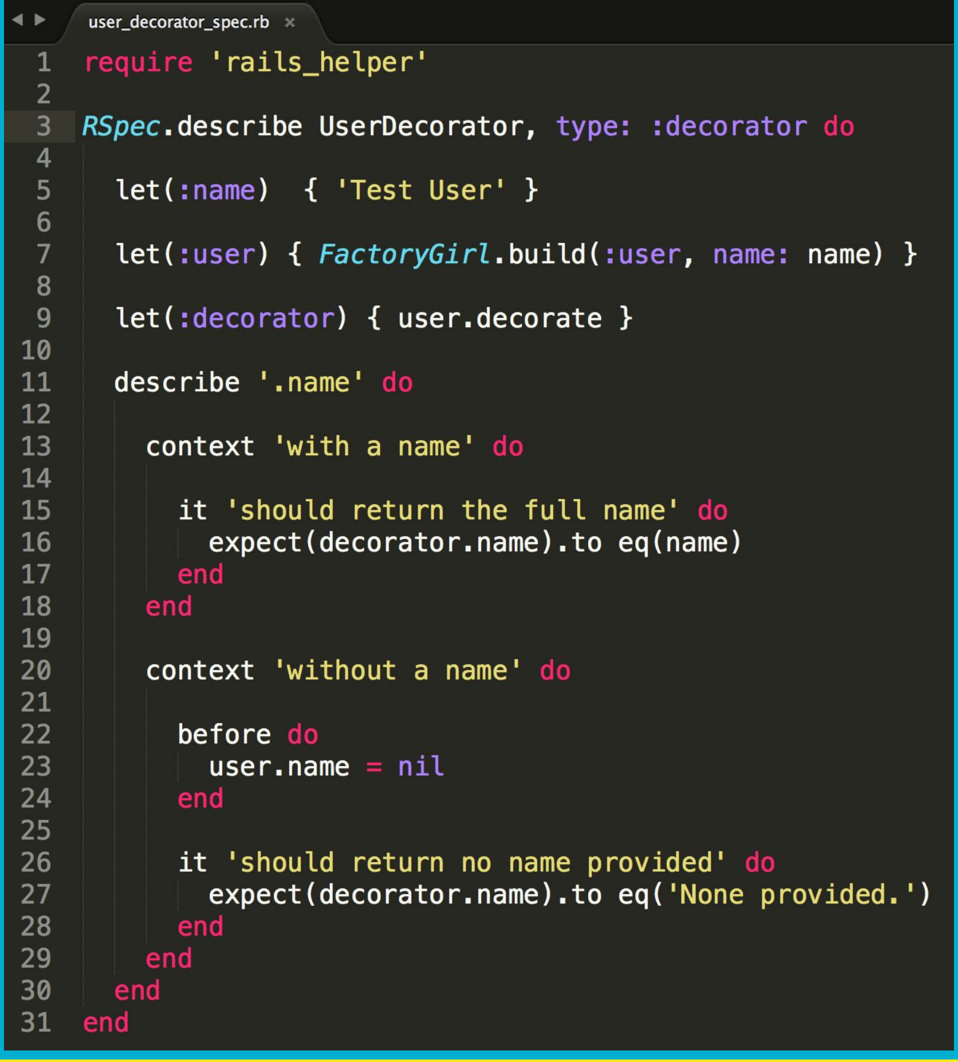
1. add gem ‘draper’ to your gem file
    a. make sure it is in the main part of the gemfile
2. bundle install
3. create a new decorator
    a. rails generate decorator <model>
        i. for the example purposes rails generate decorator user
            1. this will already be generated for you if you have the draper gem installed before creating the controller

Decorator TDD

- add gem 'factory_girl_rails'
  - to your test, development gemfile group
- add gem 'rspec-rails', '~> 3.0'
  - to your test, development gemfile group
- Add gem ‘database_cleaner’
- configure your rspec - http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md
- bundle install
- rails generate rspec:install

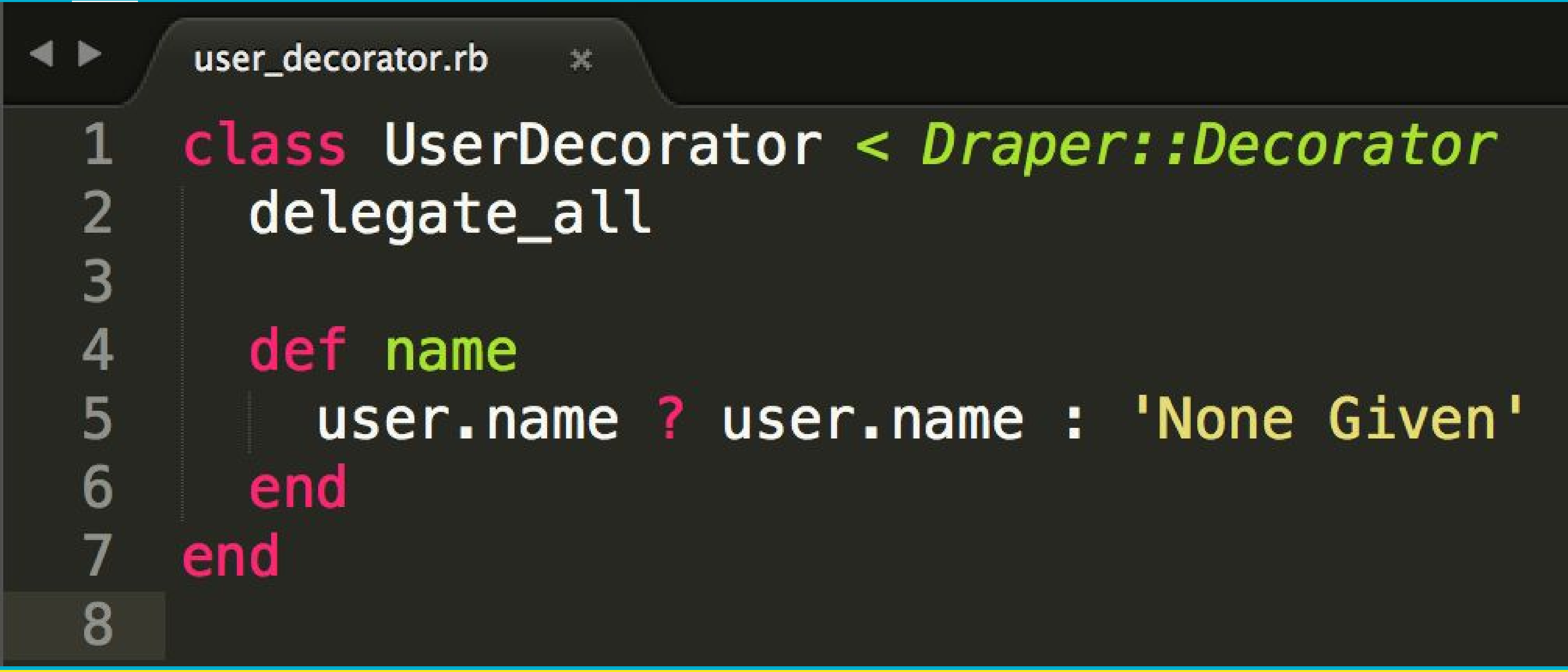
Decorator TDD Cont...

- add your user factory
- add a new folder called factories under your spec folder
- create a new file called user.rb inside of the factories folder
- generate the factory with the appropriate attributes
    - http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md
- create a folder in spec called decorators
- create a file inside the folder called user_decorator_spec.rb
- create the tests for the decorator
    - these will fail that is the point of TDD, once we have the tests setup we can start to make them pass by adding the code into the decorator
- run your decorator specs - bundle exec rspec spec/decorators/



```ruby
require 'rails_helper'

RSpec.describe UserDecorator, type: :decorator do

  let(:name)  { 'Test User' }

  let(:user) { FactoryGirl.build(:user, name: name) }

  let(:decorator) { user.decorate }

  describe '.name' do

    context 'with a name' do

      it 'should return the full name' do
        expect(decorator.name).to eq(name)
      end
    end

    context 'without a name' do

      before do
        user.name = nil
      end

      it 'should return no name provided' do
        expect(decorator.name).to eq('None provided.')
      end
    end
  end
end
```

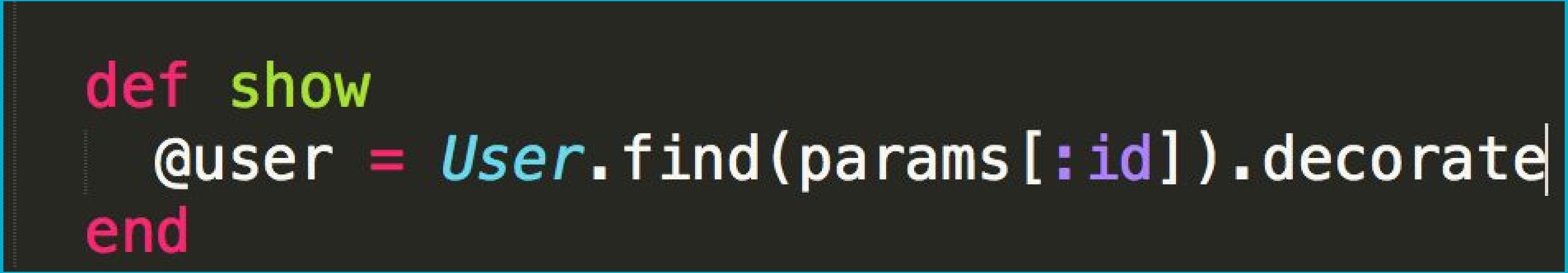
TDD - Make the tests pass



user_decorator.rb

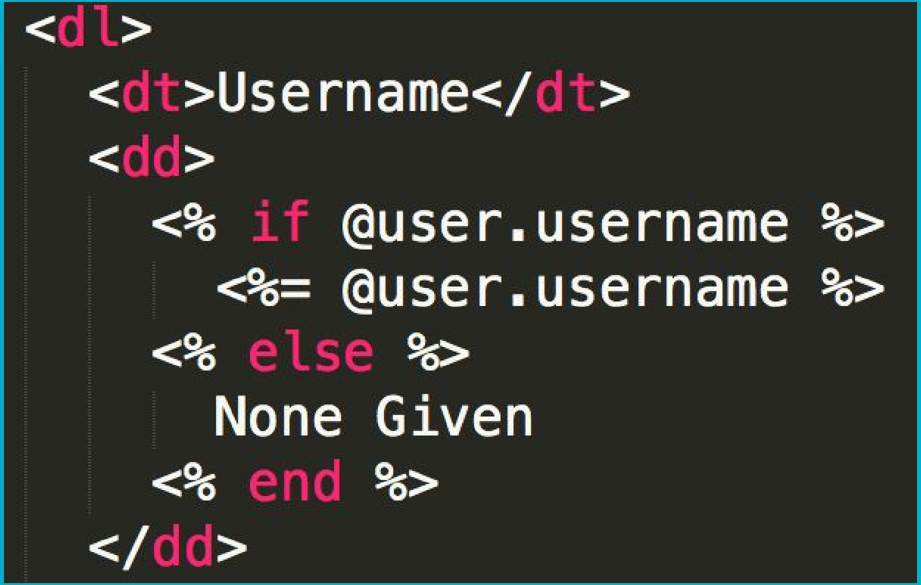
```ruby
class UserDecorator < Draper::Decorator
  delegate_all

  def name
    user.name ? user.name : 'None Given'
  end
end
```

Find the user and decorate it

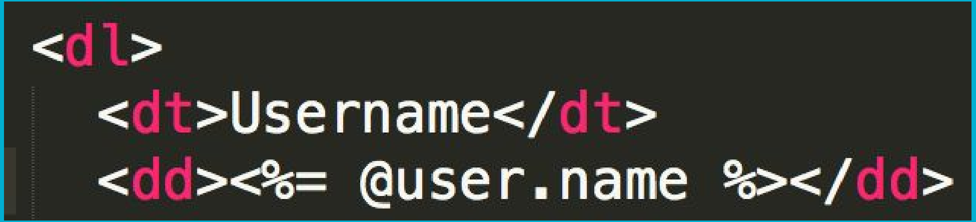


```ruby
def show
  @user = User.find(params[:id]).decorate
end
```

Refactor your views to use the decorator



```
<dl>
  <dt>Username</dt>
  <dd>
    <% if @user.username %>
      <%= @user.username %>
    <% else %>
      None Given
    <% end %>
  </dd>
```



```
<dl>
  <dt>Username</dt>
  <dd><%= @user.name %></dd>
```

Decorators / Presenters Wrap up

- The decorator pattern is a great way to clean up complex views
- The decorator pattern makes your views very easily tested
- Change code in 1 place and all the views using that decorated object change
  - DRY code is the best code!
- Don’t start with the decorator pattern, get your app to work and then clean up your complex views