Rails Project Structure

Ynon Perek

Why should all projects look the same?

Major Parts In Rails App

- Models, Views, Controllers
- Concerns
- Actions
- Helpers, Partials
- Channels
- Jobs
- Mailers
- Initialisers
- Migrations
- Rake tasks, scripts
- Router
- Middlewares
- Plugins
- Engines

Models

- The “brain” of your application
- Most of the code goes here
- Validations
- Business logic concerning one entity
- Examples: User, Photo, Article, Video, Comment

Model Concerns

- Code reuse between models

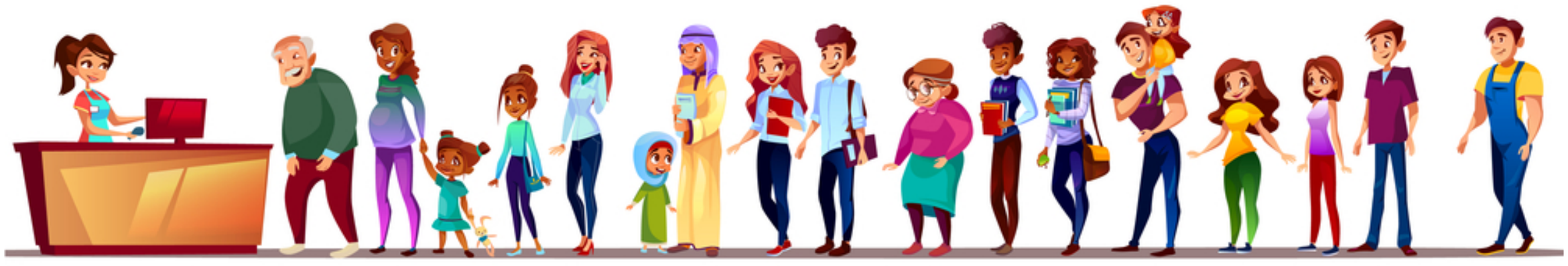
- **Best Practices:**
  - Work in isolation
  - Used by multiple models

```
module Emailable
  include ActiveSupport::Concern
  def deliver(email)
    # send email here...
  end
end
```

My Rails Models Are Bloated. Should I Use Concerns?

Controllers

- Receive incoming requests
- Create and send responses
- **Rails Way - Fat controllers, Thin models**



Tip: Use Standard Controller Actions

```ruby
class InboxesController < ApplicationController
  def index
  end

  def pendings
  end
end
```

```ruby
class InboxesController < ApplicationController
  def index
  end
end

class Inboxes::PendingsController < ApplicationController
  def index
  end
end
```

Controller Concerns

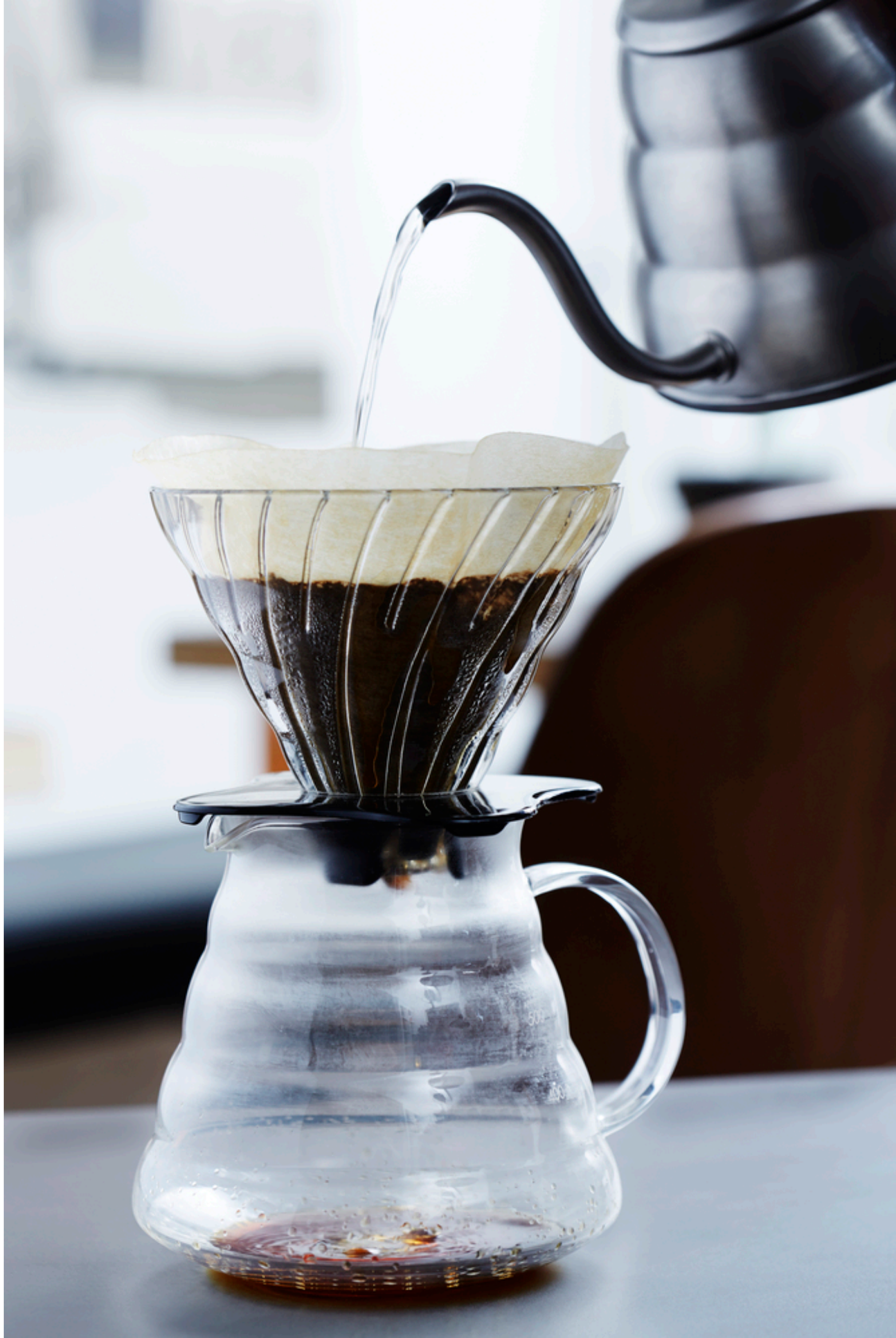
- Alternative to controller inheritance
- Same pitfalls as with model concerns

```ruby
module IncrementableVisits
  extend ActiveSupport::Concern

  included do
    before_action :increment_visits, only: [:index, :show, :new, :edit]
  end

  def increment_visits
    current_user.inc(visits: 1) # Mongoid ODM example
  end
end
```

My Rails Controllers Are Bloated. Should I Use Concerns?

Views

- Describe the structure of the response
- Extension matters:
  - .html.erb
  - .json.erb
  - .json.jbuilder
- Uses @variables from controllers



View Best Practices

- Focus on structure, no logic
- Replace instance variable with local variable
- Use only one dot
- Don’t hit database in views
- Avoid the variables assignments inside views

Implicit / Explicit N + 1 Prevention

- How many queries will this code perform?

```
<% @users.each do |user|%>
    <%= user.house.address %>
<% end %>
```

Implicit / Explicit N + 1 Prevention

- Well, it depends…

```ruby
@users = User.limit(50)
```

```ruby
@users = User.includes(:house).limit(50)
```

Implicit / Explicit N + 1 Prevention

- Better to use explicit joins and:

```
<% @users.each do |user|%>
    <%= user.house_address %>
<% end %>
```

Decorator Pattern

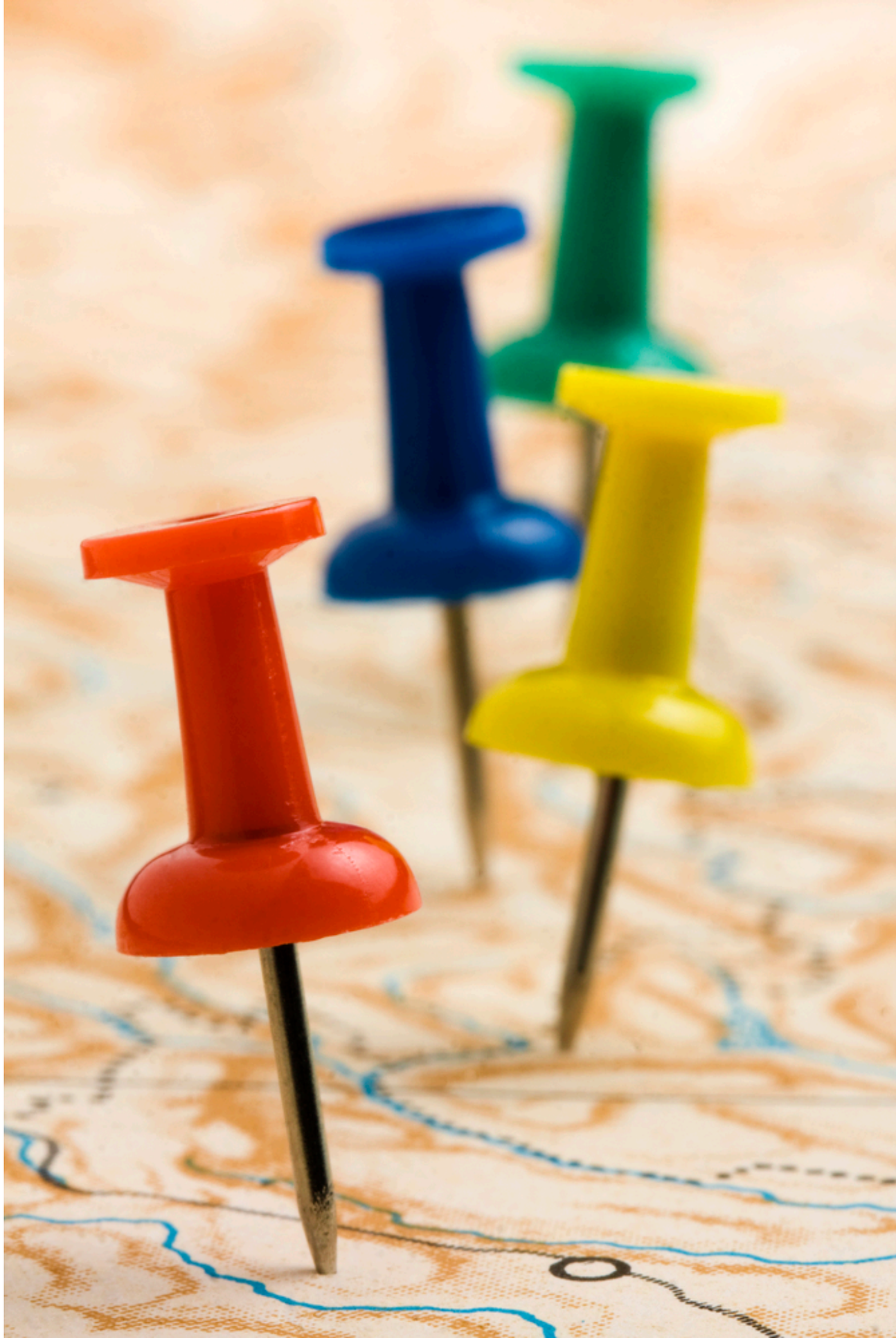
- Used to add methods to models just for the view
- Usually saved in files: app/decorators/repository_decorator.rb

```
class RepositoryDecorator < SimpleDelegator
  def display_name
    name.gsub("-", " ").titleize
  end
end
```

```
RepositoryDecorator.new(@repository).display_name
```

Actions (Service Object)

- Combines code that affects multiple models
- Created when the action:
  - is complex
  - uses APIs of external services without models
  - uses several models



Made up example

```ruby
class TweetController < ApplicationController
  def create
    send_tweet(params[:message])
  end

  private

  def send_tweet(tweet)
    client = Twitter::REST::Client.new do |config|
      config.consumer_key        = ENV['TWITTER_CONSUMER_KEY']
      config.consumer_secret     = ENV['TWITTER_CONSUMER_SECRET']
      config.access_token        = ENV['TWITTER_ACCESS_TOKEN']
      config.access_token_secret = ENV['TWITTER_ACCESS_SECRET']
    end
    client.update(tweet)
  end
end
```

Real World Example

```
class PurchaseFinaliser
  def payment_received(shopping_cart_id)
    cart = ShoppingCart.find(shopping_cart_id)
    user = cart.user

    create_and_send_invoice(user, cart)

    cart.products.each do |product|
      user.grant_access(product)
    end
  end
end
```

Actions vs. Jobs

- Some would argue to call Service Objects from your jobs
- Others would argue to put the logic inside the job

```
class PostJob < ActiveJob::Base
  def perform
    Post::SomeService.perform
  end
end
```

Service Objects vs. Models

- If your logic fits well in a specific model - it should be written there

- If your logic spans multiple models, consider a service object

- **We need a way to add a user to a group. Do we add User#join_group or Group#add_member?**

Service Objects Examples

- CreateUser
- CreateGroup
- AddUserToGroup
- BanUserFromGroup

Service Objects vs. Models (DHH)

- Hi @dhh Where do you store business logic in @basecamp. In models or in some kind of service objects?

- **99% Models.**

Helpers & Partials

View Helpers

```
<% if @user && @user.email.present? %>
  <%= @user.email %>
<% end %>
```

```
<%= user_email(@user) %>

module SiteHelper
  def user_email(user)
    user.email if user && user.email.present?
  end
end
```

Helpers Can Use Other Helpers

```
def menu_item(path, text, condition=true)
  return if !condition
  content_tag(
    :li,
    link_to( I18n.t(text), path ),
    class: current_class?(path)
  )
end
```

Helpers Can Use Concat

```ruby
def payment_method_radio_button(type:, name:)
  label_tag "radio_#{type}", nil, class: 'icon-n-text _vertical' do
    concat content_tag :span, '', class: "icon icon-#{type}"
    concat content_tag :div, name, class: 'text'
  end
end
```

View Partials - simplify your views

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

<%= render "shared/footer" %>
```

View Partials - generalise your views with variables

```
<h1>New zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>

<%= render partial: "customer", object: @new_customer %>
```

View partials - collections

- Partials are very useful in rendering collections. When you pass a collection to a partial via the :collection option, the partial will be inserted once for each member in the collection:

```
<h1>Products</h1>
<%= render partial: "product", collection: @products %>
```

Partials Vs. Helpers Vs. Presenters

- Use helper when focus is on the logic
- Use partials when focus is on the markup
- Use a presenter if there's interdependent calls and/or state between helper methods

Initializers

- Rails initialisers run once when the application starts
- Use for logic that doesn't change throughout the application's lifecycle
- Main use - configuration
- Also used to add middlewares (see omniauth)

Rails Router

Router Goal

- Get the request to the correct controller
- Create route helpers
- How:
  - Route definitions
  - Constraints
  - Namespace

Router Constraints

- You know the basics:

```
match 'photos', to: 'photos#show',
via: [:get, :post]
```

But it gets interesting quickly

```
get 'photos/:id', to: 'photos#show', constraints: {
id: /[A-Z]\d{5}/ }

get 'photos', to: 'photos#index', constraints:
{ subdomain: 'admin' }

get '*path', to: 'blacklist#index', constraints:
lambda { |request| Blacklist.retrieve_ips.include?
(request.remote_ip) }
```

But it gets interesting quickly

```
get     'sample/url' => 'my#new_user',         constraints: NewUserConstraint
get     'sample/url' => 'my#wants_photo',      constraints: WantsPhotoConstraint
get     'sample/url' => 'my#user_to_update', constraints: UserToUpdateConstraint
```

```
class NewUserConstraint
   def self.matches?(request)
     request.query_parameters['strategy'] == 'new_user'
   end
end
```

Router Concerns

```
resources :messages, concerns: :commentable
resources :articles, concerns: [:commentable, :image_attachable]

concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end
```

Middlewares

- Rails middlewares are saved in app/middleware
- Load and use middleware with an initialiser:

```
Rails.application.config.middleware.use Foobar
```

When To Use Middlewares

- Logging
- Warden (user management)
- Security
- Blacklist IPs

Plugins / Engines

- Extract large functionality into external part of the application

- Examples:
  - forum (threaded)
  - user management (devise)

Q & A