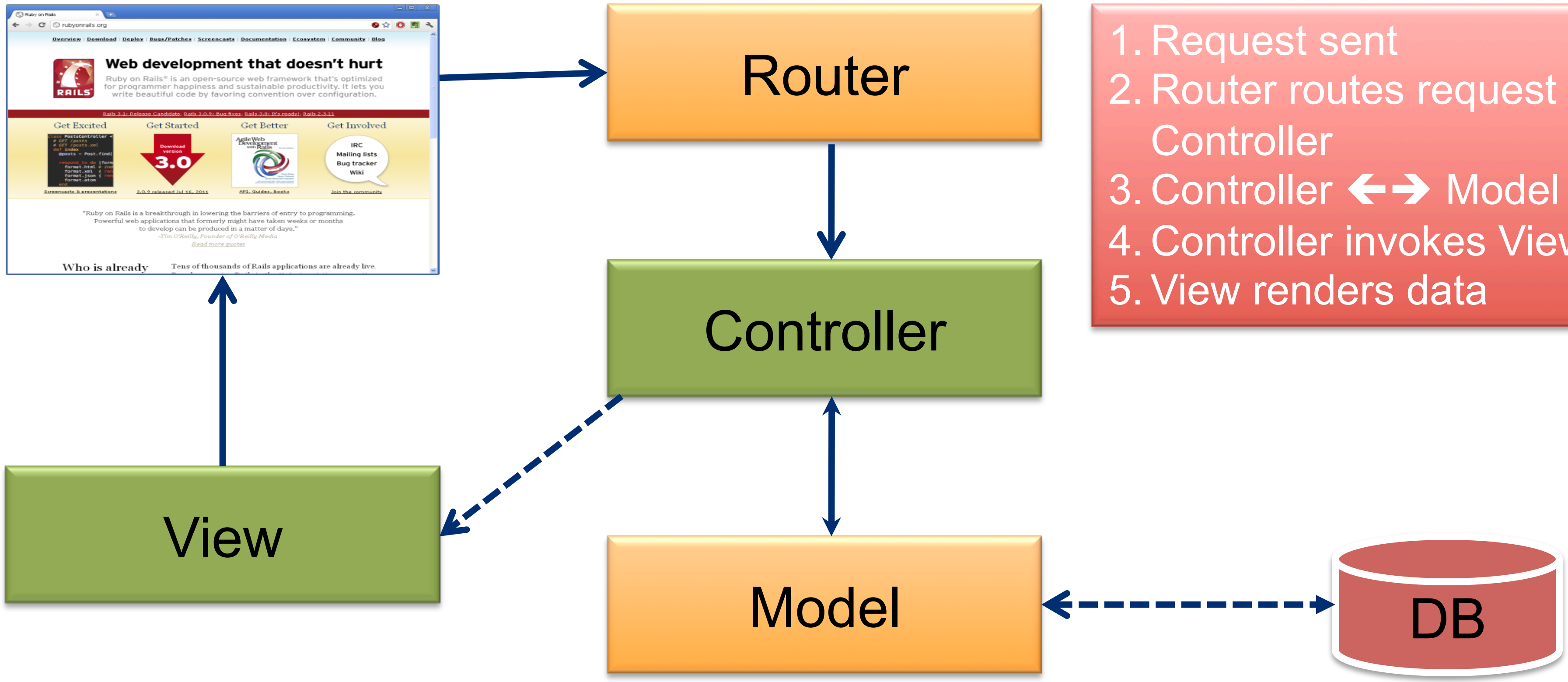
In this lecture, we will discuss…

- ✧ Introduction to Action Pack

ActionController + ActionView = AP



Ruby on Rails

rubyonrails.org

Overview | Download | Deploy | Bugs/Patches | Screencasts | Documentation | Ecosystem | Community | Blog

Web development that doesn't hurt

Ruby on Rails® is an open-source web framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration.

Rails 3.1: Release Candidate, Rails 3.0.9: Bug fixes, Rails 3.0: It's ready!, Rails 2.3.11

Get Excited | Get Started | Get Better | Get Involved

```
class PostsController <
  # GET /posts
  # GET /posts.xml
  def index
    @posts = Post.find(

    respond_to do |form
      format.html # ind
      format.xml  { ren
      format.json { ren
      format.atom
  end
```

Download version 3.0

Agile Web Development with Rails

IRC
Mailing lists
Bug tracker
Wiki

Screencasts & presentations | 3.0.9 released Jul 16, 2011 | API, Guides, Books | Join the community

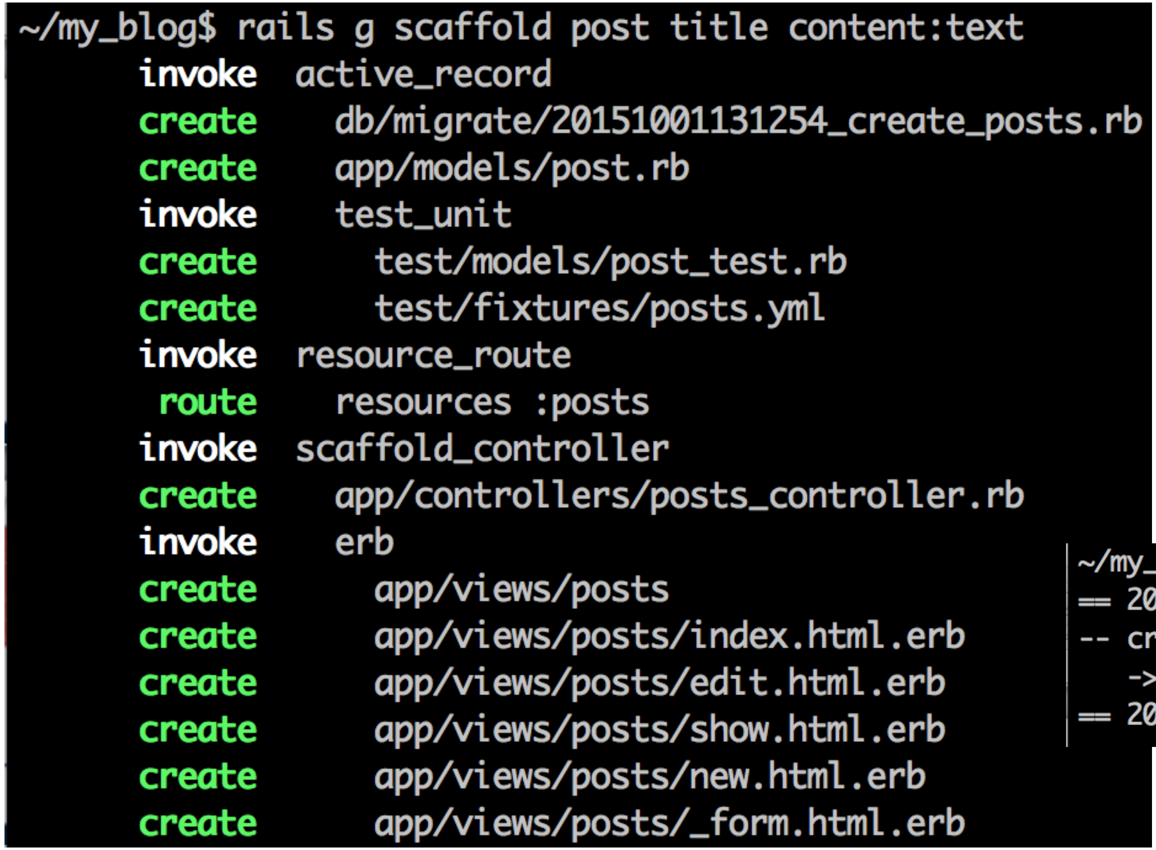
"Ruby on Rails is a breakthrough in lowering the barriers of entry to programming. Powerful web applications that formerly might have taken weeks or months to develop can be produced in a matter of days."
-Tim O'Reilly, Founder of O'Reilly Media
Read more quotes

Who is already

Tens of thousands of Rails applications are already live.

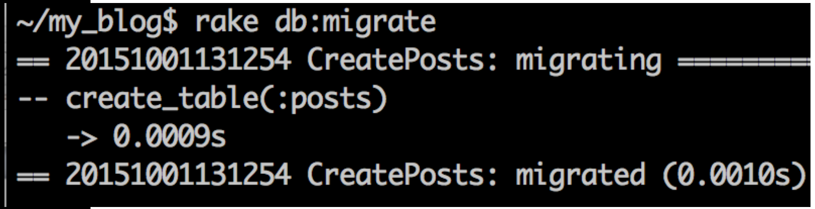
1. Request sent
2. Router routes request to Controller
3. Controller ←→ Model
4. Controller invokes View
5. View renders data

Blog Scaffolding



```
~/my_blog$ rails g scaffold post title content:text
      invoke  active_record
      create    db/migrate/20151001131254_create_posts.rb
      create    app/models/post.rb
      invoke    test_unit
      create      test/models/post_test.rb
      create      test/fixtures/posts.yml
      invoke  resource_route
       route    resources :posts
      invoke  scaffold_controller
      create    app/controllers/posts_controller.rb
      invoke    erb
      create      app/views/posts
      create      app/views/posts/index.html.erb
      create      app/views/posts/edit.html.erb
      create      app/views/posts/show.html.erb
      create      app/views/posts/new.html.erb
      create      app/views/posts/_form.html.erb
```

$rails new my_blog



```
~/my_blog$ rake db:migrate
== 20151001131254 CreatePosts: migrating =========
-- create_table(:posts)
   -> 0.0009s
== 20151001131254 CreatePosts: migrated (0.0010s)
```

Scaffolding: Explaining The Magic

- Scaffolding creates:
  1. Migration
  2. Model
  3. *Routes*
  4. *Restful Controller*
  5. *Views*
  6. More…

Discuss these now…

ActionView: ERB

- ✧ HTML file with an .erb extension
- ✧ ERb is a templating library (similar to JSP) that lets you embed Ruby into your html
- ✧ Two tag patterns to learn:
  - `<% …ruby code… %>` - evaluate Ruby code
  - `<%= …ruby code… %>` - output evaluated Ruby code

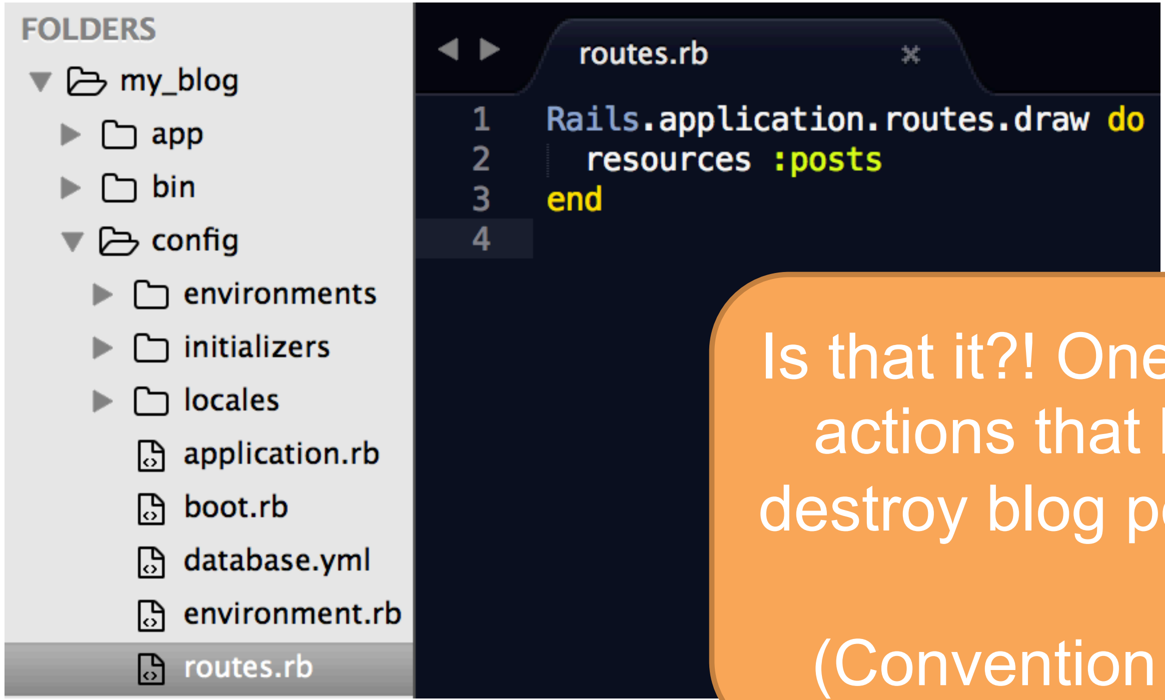
…And of course, there are many view helpers that assist in code creation, like link_to…

Action Controller

- Ruby class containing one or more actions
- Each action is responsible for responding to a request to perform some task
- Unless otherwise stated – when an action is finished firing (or even if the action is not physically present) it renders a view with the same name as the action
- The action always needs to be mapped in `routes.rb`

Speaking of Routes…

✧ Let’s see scaffolded routes



```ruby
Rails.application.routes.draw do
  resources :posts
end

```

Is that it?! One line lets you route to actions that list, create, edit and destroy blog postings? Impossible…

(Convention Over Configuration)

Summary

- ✧ Action Pack is Controller and View work together to let you interact with resources in the Model layer

What’s Next?

- ✧ REST and Rails

In this lecture, we will discuss…

- REST
- How Rails adapted RESTful principles

REST

- Representational State Transfer
- Roy T. Fielding’s Ph.D. dissertation

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.”

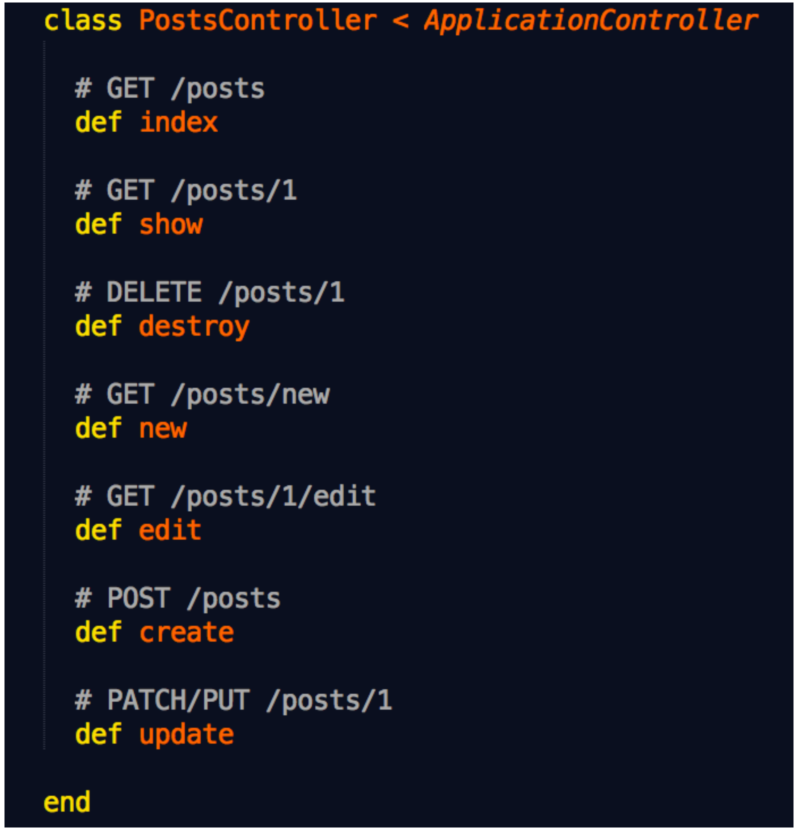
Check out the following great resource on REST
http://www.xfront.com/REST-Web-Services.html

REST = Resources

- ✧ REST is all about resources
- ✧ You should be to able to:

1. List available resources
2. Show a specific resource
3. Destroy an existing resource
4. Provide a way to create a new resource
5. Create a new resource
6. Provide a way to update an existing resource
7. Update an existing resource

REST: A Simple Rails Convention



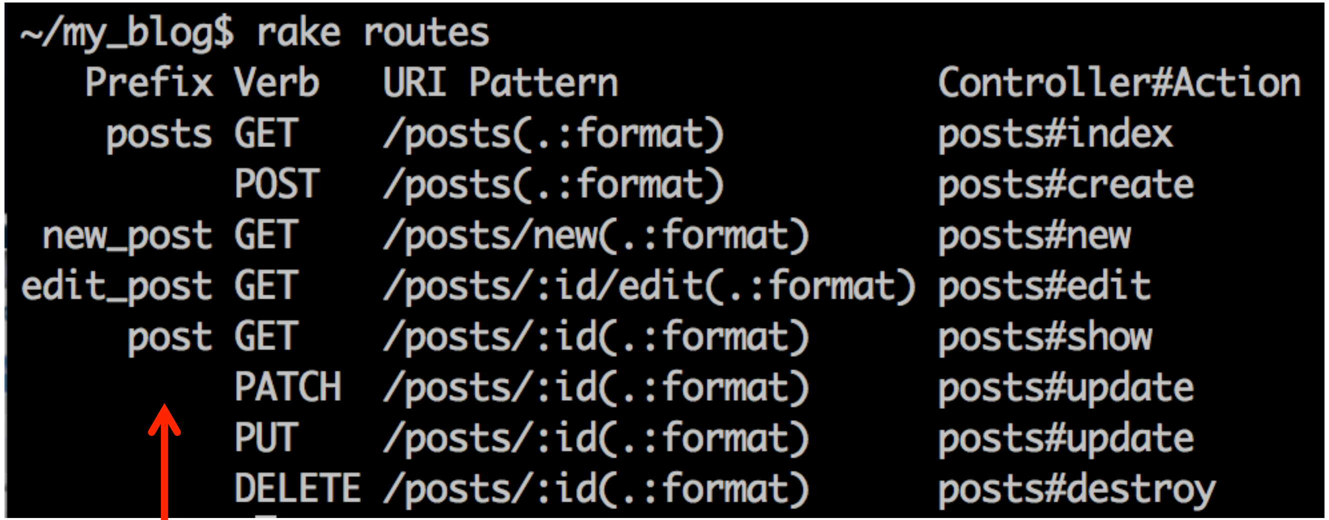
```
class PostsController < ApplicationController

  # GET /posts
  def index

  # GET /posts/1
  def show

  # DELETE /posts/1
  def destroy

  # GET /posts/new
  def new

  # GET /posts/1/edit
  def edit

  # POST /posts
  def create

  # PATCH/PUT /posts/1
  def update

end
```

Named Routes From ‘Resources :Posts’

HTTP Method	Named Routes	Parameters	Controller Action	Purpose
GET	posts_path		index	List all
GET	post_path	ID	show	Show one
GET	new_post_path		new	Provide form to input new post
POST	posts_path	Record hash	create	Create new record (in DB)
GET	edit_post_path	ID	edit	Provide form to edit post
PUT/PATCH	post_path	ID and Record hash	update	Update record (in DB)
DELETE	post_path	ID	destroy	Remove record

$rake routes

- If you forget the chart on the previous page, you can always just run `$rake routes`



```
~/my_blog$ rake routes
   Prefix Verb   URI Pattern                  Controller#Action
    posts GET    /posts(.:format)             posts#index
          POST   /posts(.:format)             posts#create
 new_post GET    /posts/new(.:format)         posts#new
edit_post GET    /posts/:id/edit(.:format)    posts#edit
     post GET    /posts/:id(.:format)         posts#show
          PATCH  /posts/:id(.:format)         posts#update
          PUT    /posts/:id(.:format)         posts#update
          DELETE /posts/:id(.:format)         posts#destroy
```

Named routes column

Summary

- Think of your application in terms of resources
- Think of the 7 RESTful actions that need to be done with those resources

What’s Next?

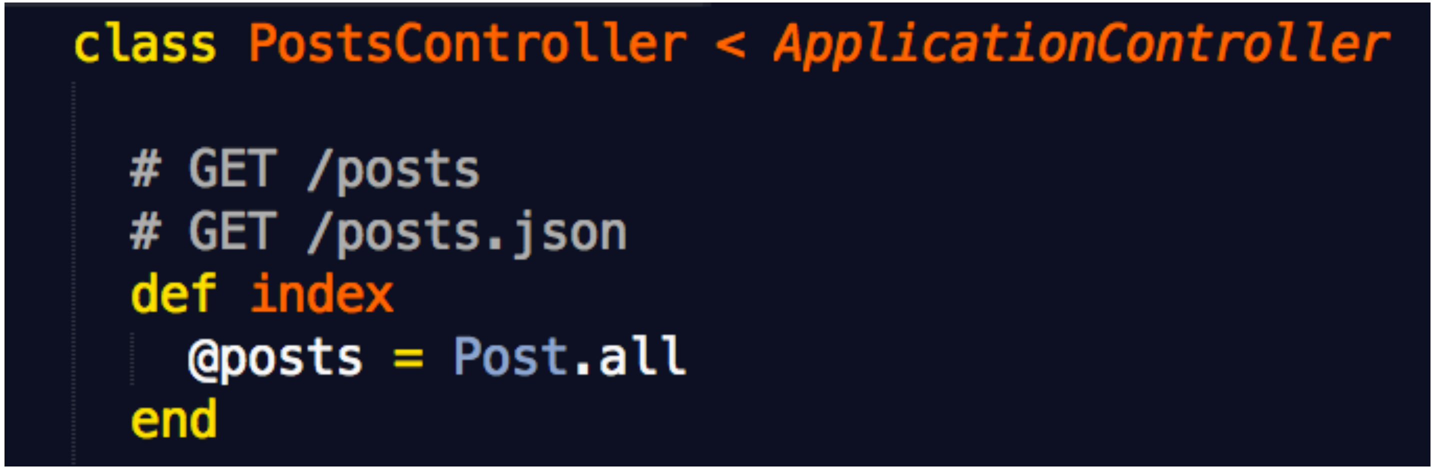
- Restful Actions: index

In this lecture, we will discuss…

- ✧ index RESTful action

Examining Seven Actions – Index

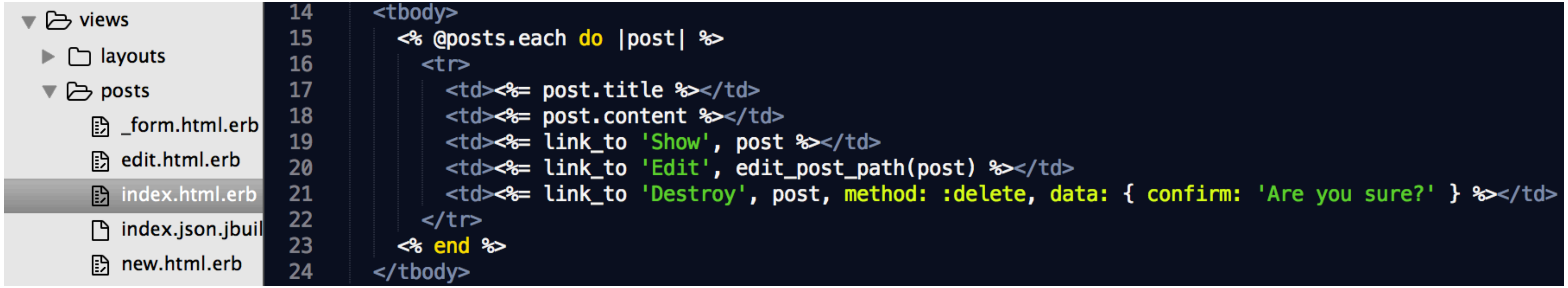
1. Retrieve all posts
2. (Implicit) Look for `index.html.erb` template to render response



```
class PostsController < ApplicationController

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
  end
```

index.html.erb

Browsers only support GET and POST methods - how do we tell Rails to treat request as DELETE?



```
<tbody>
  <% @posts.each do |post| %>
    <tr>
      <td><%= post.title %></td>
      <td><%= post.content %></td>
      <td><%= link_to 'Show', post %></td>
      <td><%= link_to 'Edit', edit_post_path(post) %></td>
      <td><%= link_to 'Destroy', post, method: :delete, data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
</tbody>
```

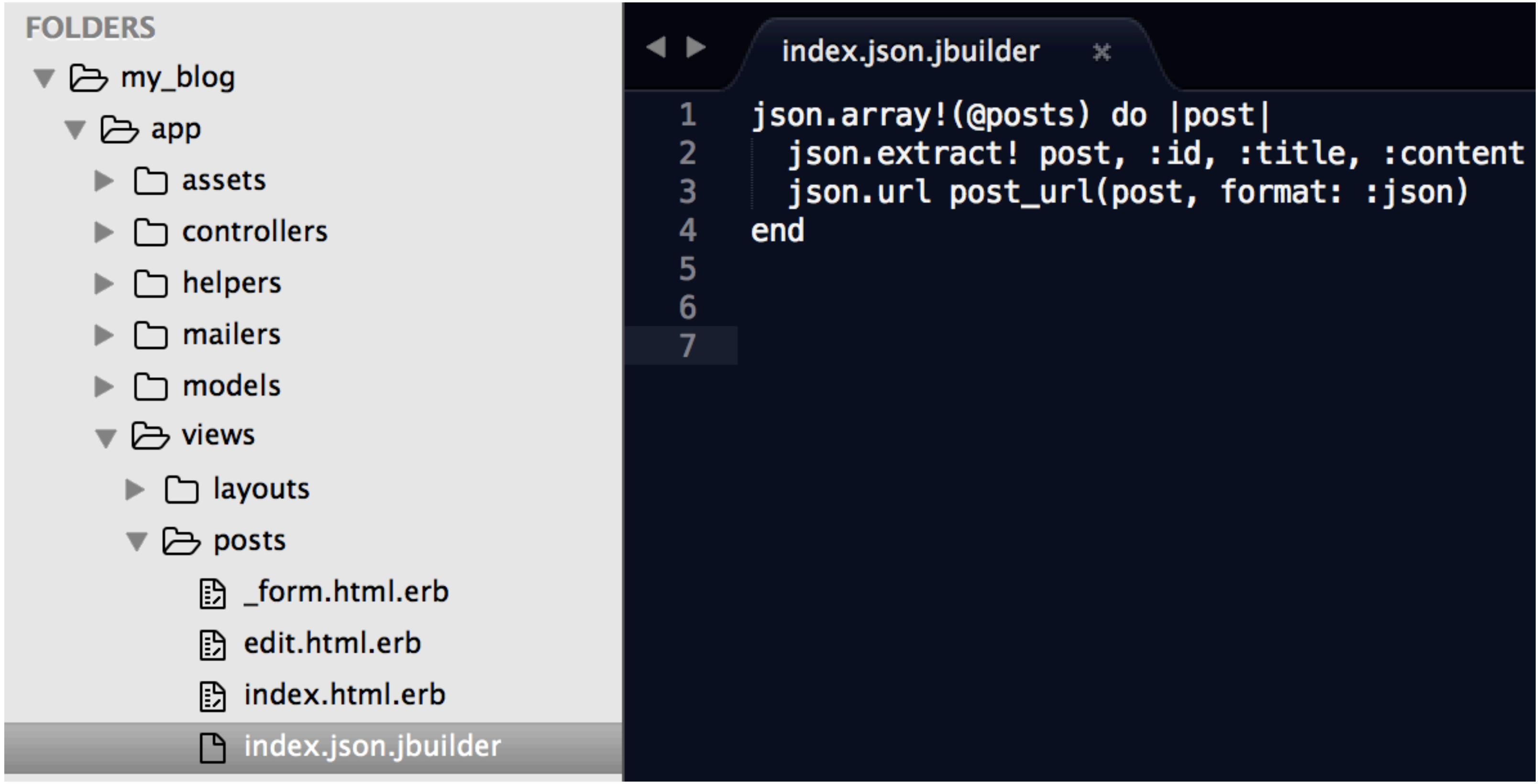
post = post_path(post)

index.html.erb

```
<tr>
  <td>Welcome</td>
  <td>Happy Action Packing!</td>
  <td><a href="/posts/1">Show</a></td>
  <td><a href="/posts/1/edit">Edit</a></td>
  <td><a data-confirm="Are you sure?" rel="nofollow" data-method="delete" href="/posts/1">Destroy</a></td>
</tr>
```

HTML5 data attributes…

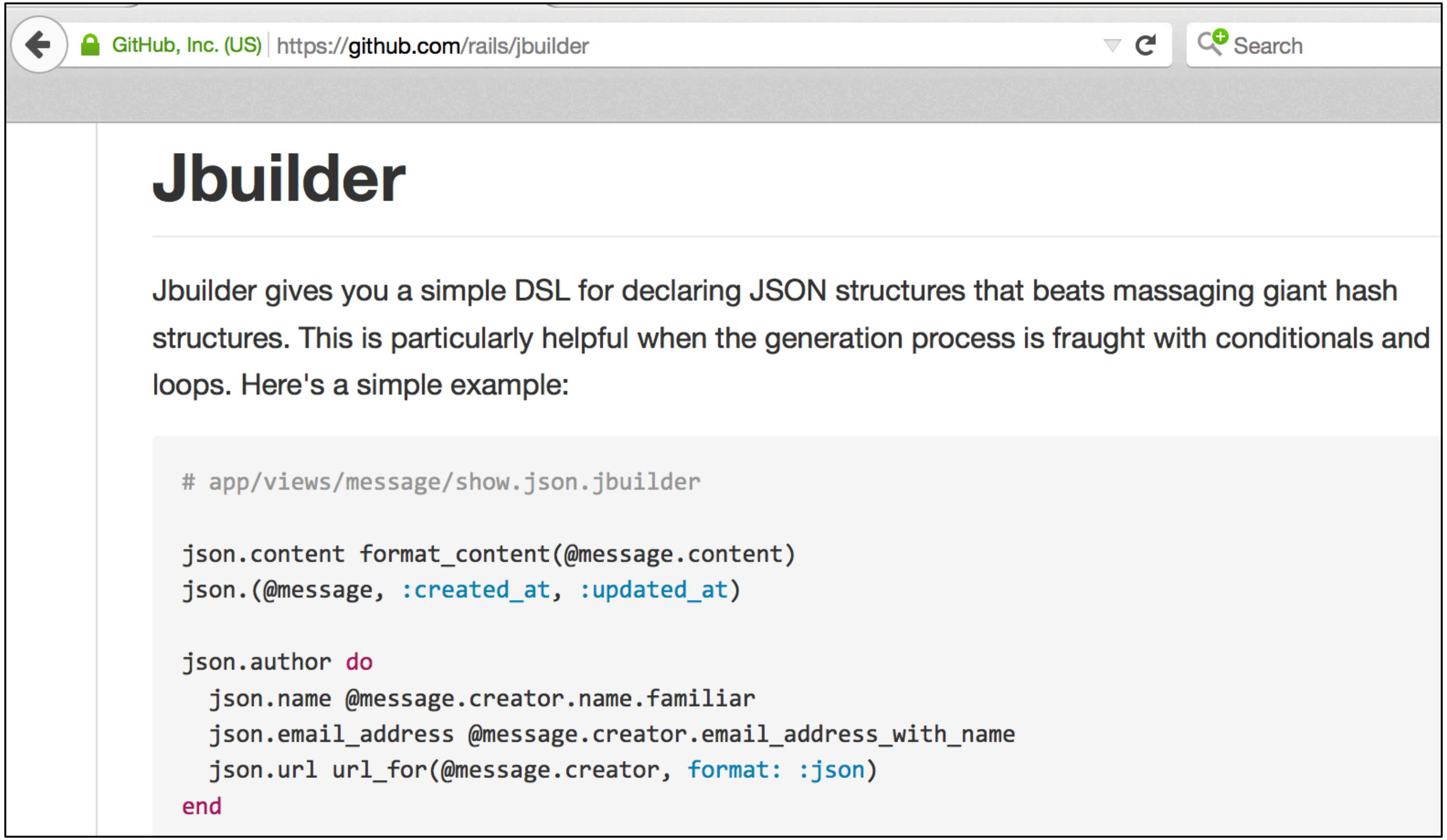
index.json.jbuilder



index.json.jbuilder

```ruby
json.array!(@posts) do |post|
  json.extract! post, :id, :title, :content
  json.url post_url(post, format: :json)
end
```

Jbuilder

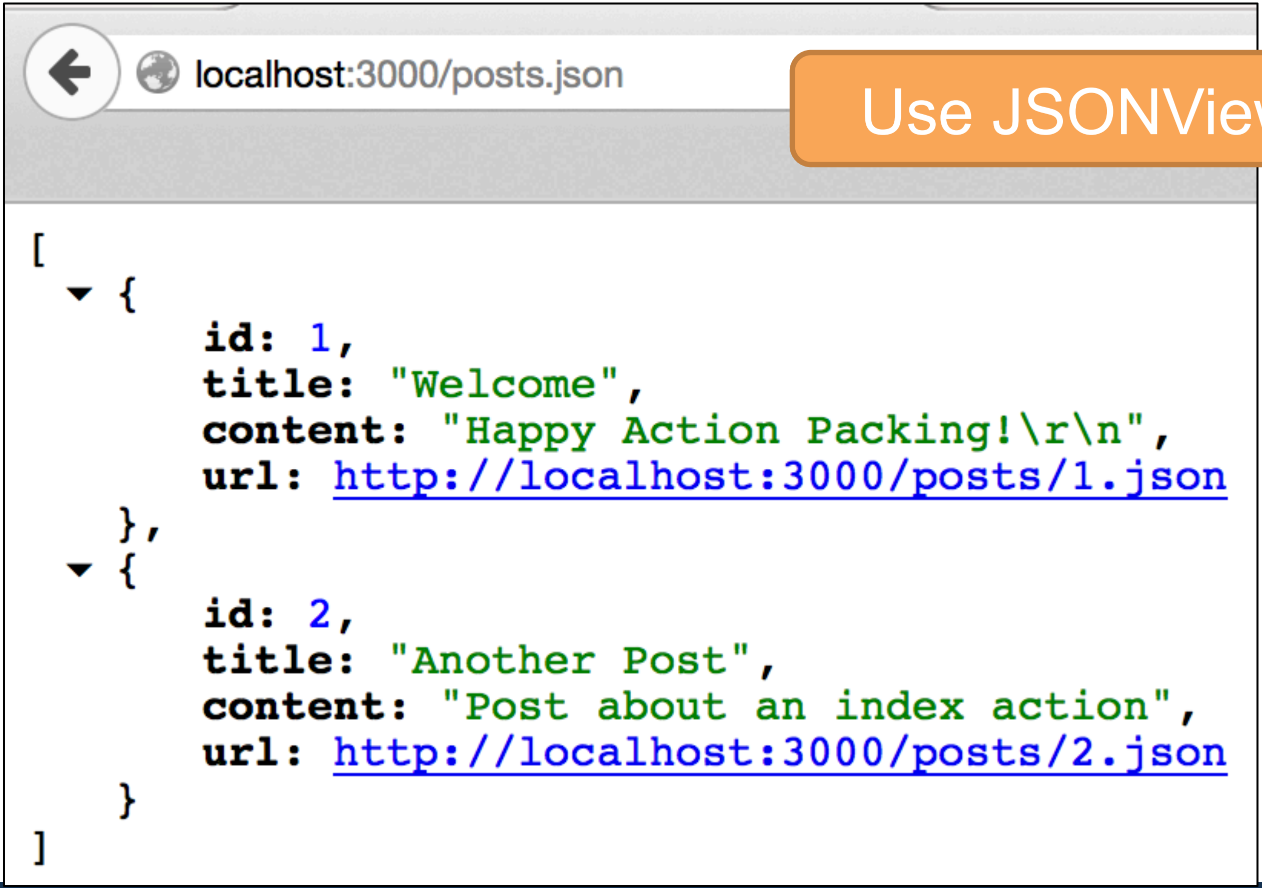


Jbuilder

Jbuilder gives you a simple DSL for declaring JSON structures that beats massaging giant hash structures. This is particularly helpful when the generation process is fraught with conditionals and loops. Here's a simple example:

```
# app/views/message/show.json.jbuilder

json.content format_content(@message.content)
json.(@message, :created_at, :updated_at)

json.author do
  json.name @message.creator.name.familiar
  json.email_address @message.creator.email_address_with_name
  json.url url_for(@message.creator, format: :json)
end
```

index.json.jbuilder



localhost:3000/posts.json

```
[
  ▾ {
        id: 1,
        title: "Welcome",
        content: "Happy Action Packing!\r\n",
        url: http://localhost:3000/posts/1.json
    },
  ▾ {
        id: 2,
        title: "Another Post",
        content: "Post about an index action",
        url: http://localhost:3000/posts/2.json
    }
]
```

Use JSONView browser plugin

Summary

- ✧ index action retrieves resources from Data layer
- ✧ Then, implicitly invokes either HTML or JSON templates

What's Next?

- ✧ show and destroy RESTful actions

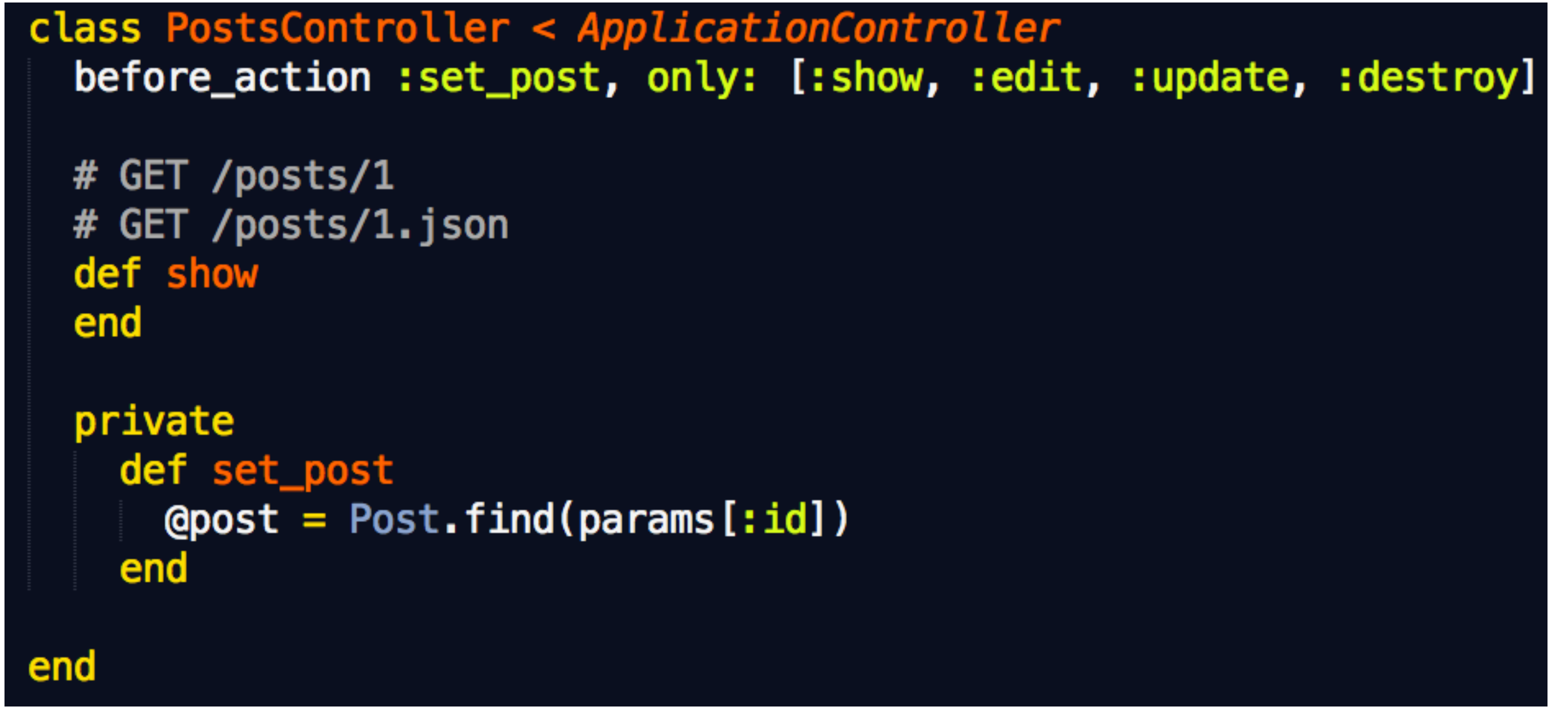
In this lecture, we will discuss…

- ✧ show RESTful action
- ✧ destroy RESTful action

Examining Seven Actions – show

1. Retrieve specific post based on `id` parameter passed in (as part of URL)
2. (Implicit) Look for `show.html.erb` template to render response

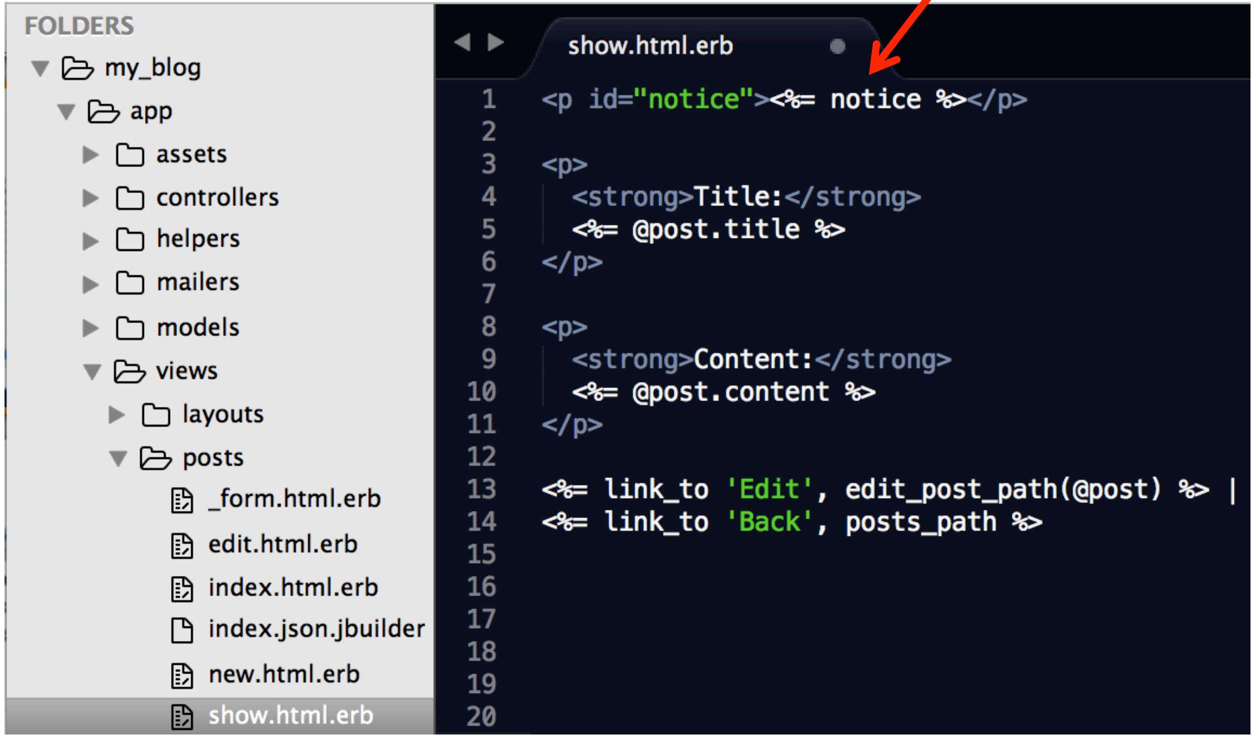
Examining Seven Actions – show



```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # GET /posts/1
  # GET /posts/1.json
  def show
  end

  private
    def set_post
      @post = Post.find(params[:id])
    end

end
```

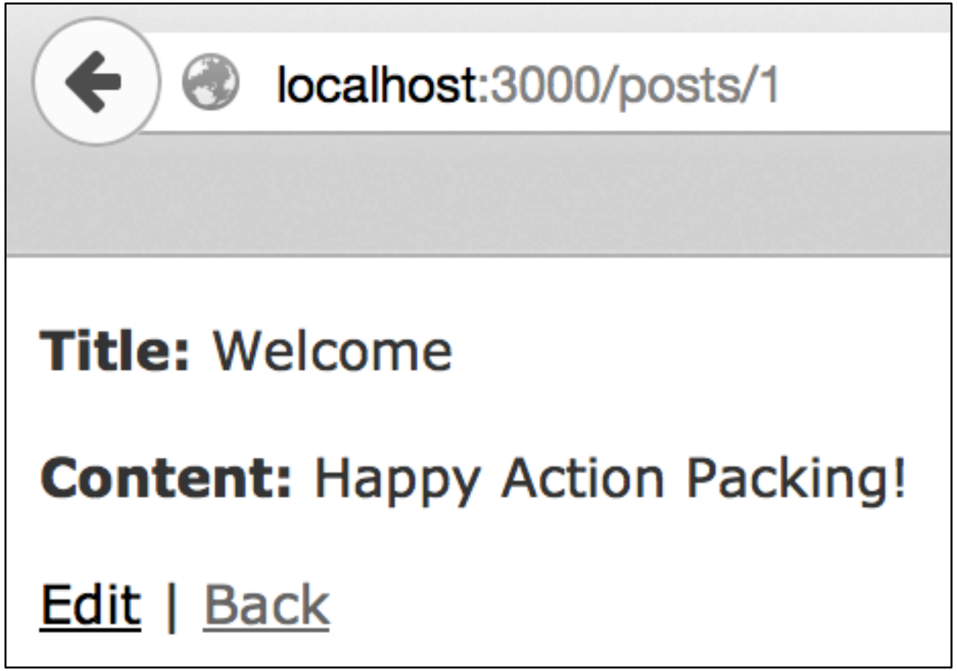
show.html.erb

Explained later



```
<p id="notice"><%= notice %></p>

<p>
  <strong>Title:</strong>
  <%= @post.title %>
</p>

<p>
  <strong>Content:</strong>
  <%= @post.content %>
</p>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

show.html.erb

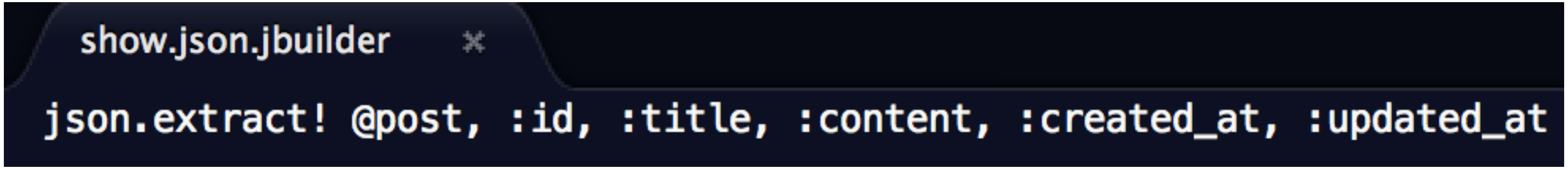

localhost:3000/posts/1

Title: Welcome

Content: Happy Action Packing!

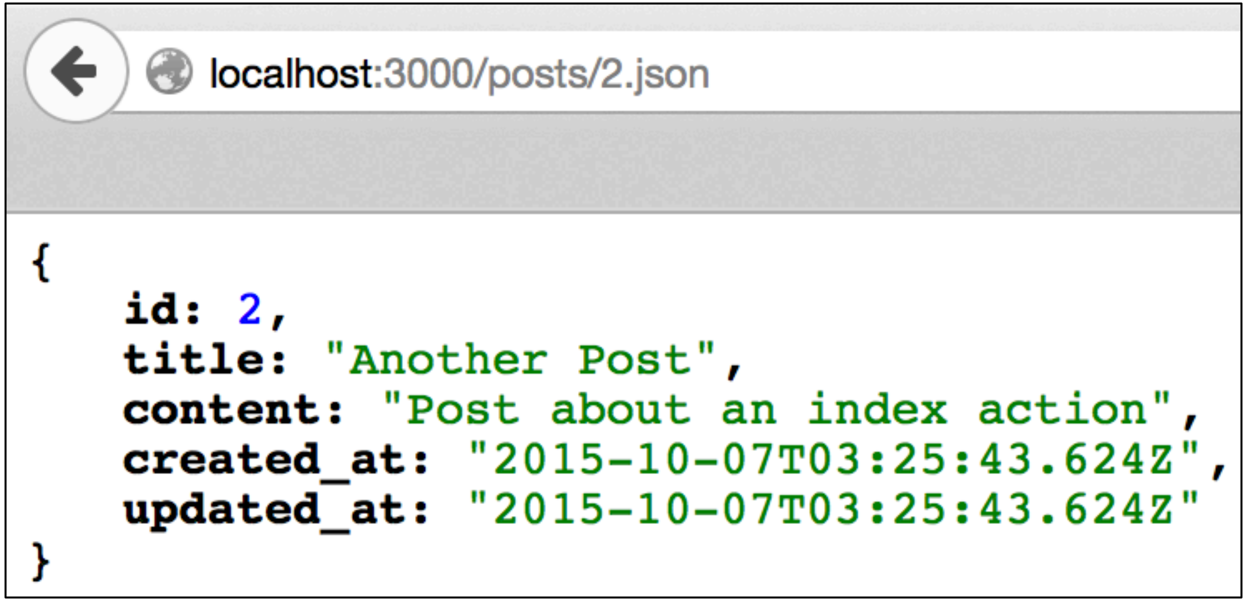
Edit | Back

show.json.jbuilder



show.json.jbuilder

```
json.extract! @post, :id, :title, :content, :created_at, :updated_at
```



localhost:3000/posts/2.json

```
{
    id: 2,
    title: "Another Post",
    content: "Post about an index action",
    created_at: "2015-10-07T03:25:43.624Z",
    updated_at: "2015-10-07T03:25:43.624Z"
}
```

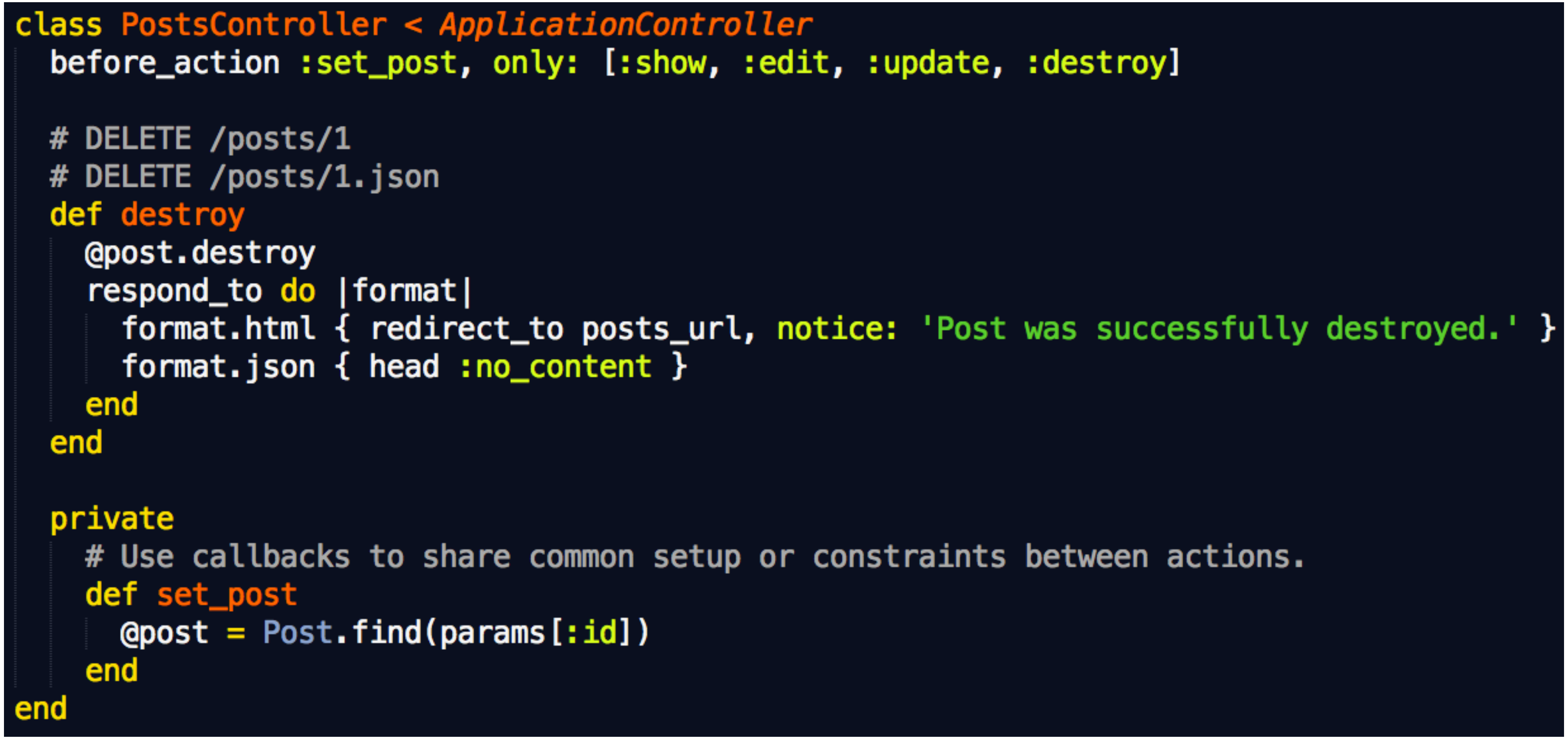
respond_to

- Rails helper that specifies how to respond to a request based on a request format
- Takes an optional block where the argument is the format (e.g. html, json, xml etc.)
- Block specifies how to handle each format:
  - `format.format_name` – matching template
  - `format.format_name { do_something_other_than_just_displaying_the_matching_template }`

redirect_to

- Instead of rendering a template – send a response to the browser: “Go here!”
- Usually takes a (full) URL as a parameter
- Could either be a regular URL (like http://google.com) or a named route
- If the parameter is an object – Rails will attempt to generate a URL for that object

Examining Seven Actions - destroy



```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy
    @post.destroy
    respond_to do |format|
      format.html { redirect_to posts_url, notice: 'Post was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_post
      @post = Post.find(params[:id])
    end
end
```

Why redirect?

- Even though redirect involves an extra step (roundtrip to the browser) – sometimes it just makes sense
- Obvious examples:
  - When you want the client to be able to bookmark a certain page or you don't have a specific template to show (destroy action) and instead want the client to go to a generic page (index)

Summary

- ✧ show action involves retrieving a resource and showing it inside an HTML or JSON template
- ✧ destroy action destroys a resource and then redirects the browser to another page

What's Next?

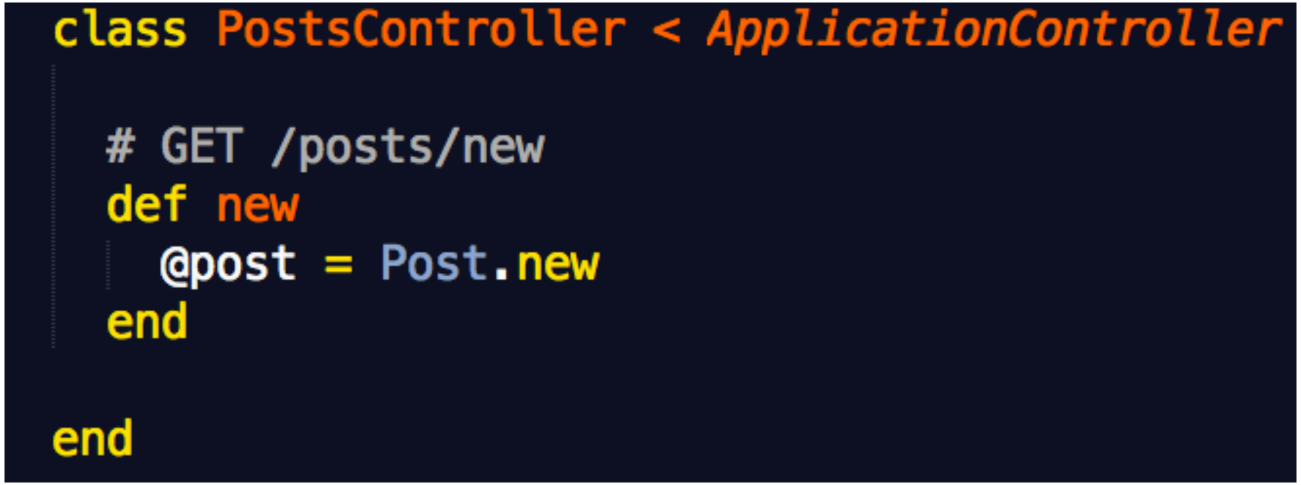
- ✧ new and create actions

In this lecture, we will discuss…

- new RESTful action
- create RESTful action

Examining Seven Actions – new

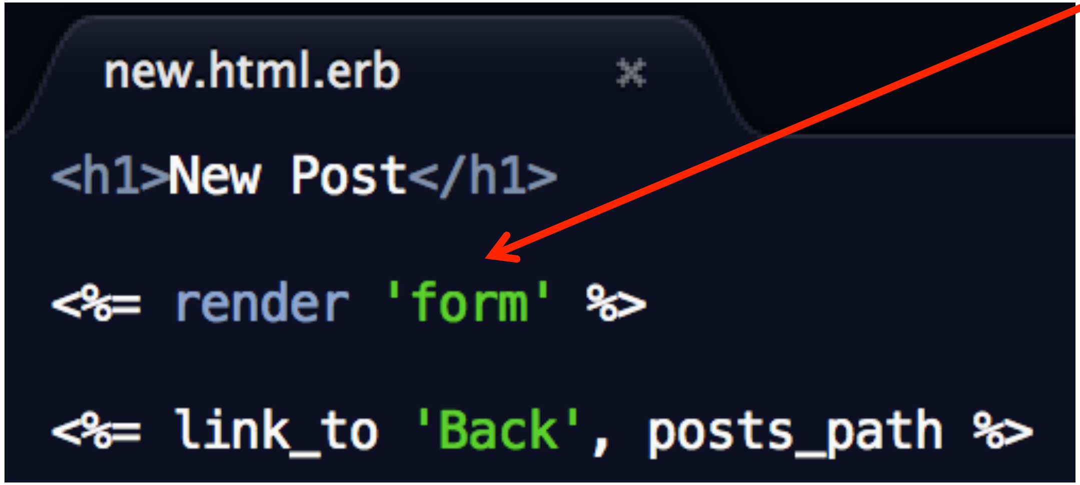
1. Create a new empty post object
2. (Implicit) Look for `new.html.erb`



```ruby
class PostsController < ApplicationController

  # GET /posts/new
  def new
    @post = Post.new
  end

end
```

new.html.erb

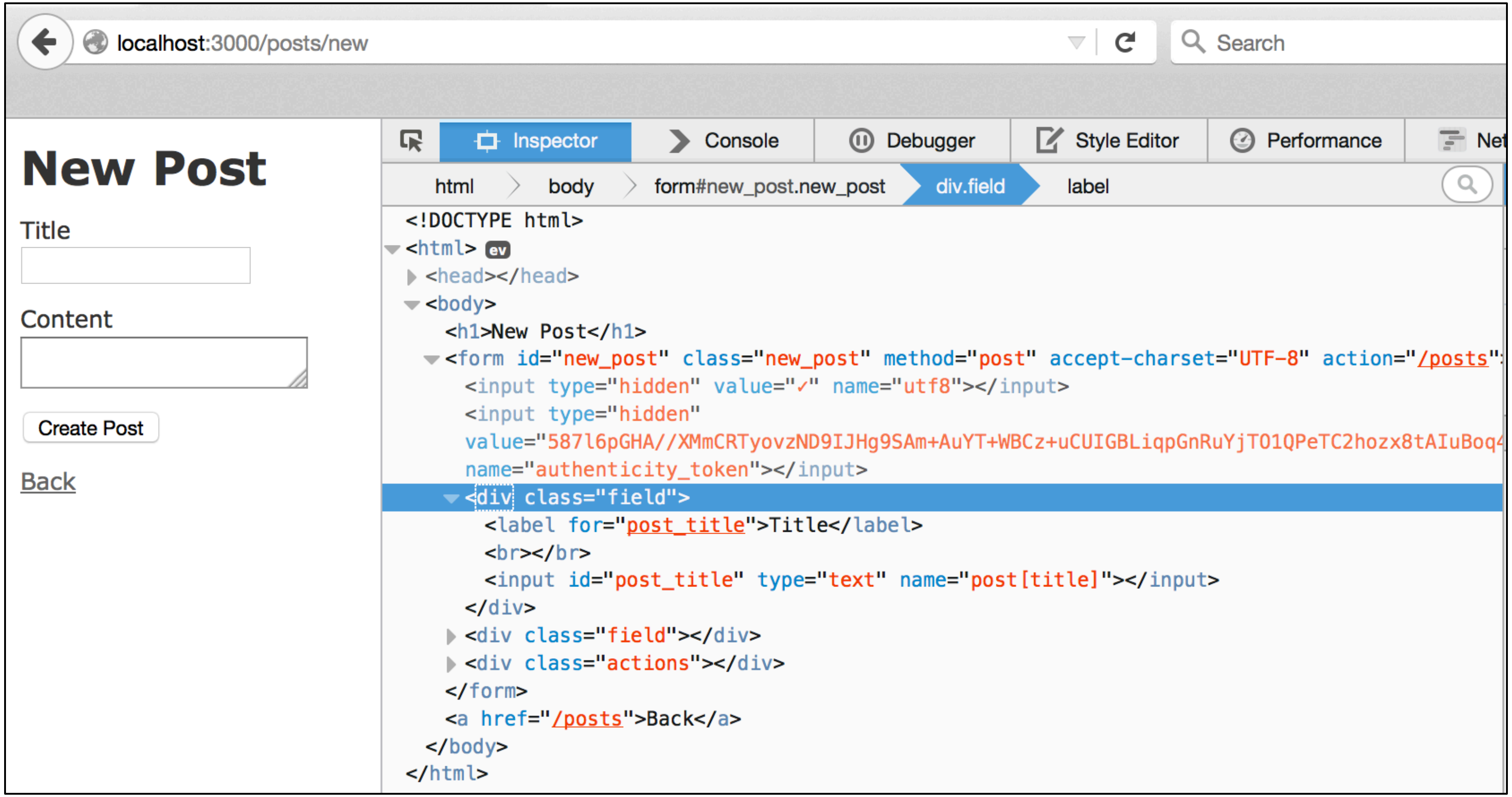
Partial – explained later



new.html.erb

```
<h1>New Post</h1>

<%= render 'form' %>

<%= link_to 'Back', posts_path %>
```

new.html.erb



New Post

Title

Content

Create Post

Back

```
<!DOCTYPE html>
<html> ev
  <head></head>
  <body>
    <h1>New Post</h1>
    <form id="new_post" class="new_post" method="post" accept-charset="UTF-8" action="/posts"
      <input type="hidden" value="✓" name="utf8"></input>
      <input type="hidden"
      value="587l6pGHA//XMmCRTyovzND9IJHg9SAm+AuYT+WBCz+uCUIGBLiqpGnRuYjT01QPeTC2hozx8tAIuBoq4
      name="authenticity_token"></input>
      <div class="field">
        <label for="post_title">Title</label>
        <br></br>
        <input id="post_title" type="text" name="post[title]"></input>
      </div>
      <div class="field"></div>
      <div class="actions"></div>
    </form>
    <a href="/posts">Back</a>
  </body>
</html>
```

Examining Seven Actions – create

1. Create a new post object with parameters that were passed from the `new` form
2. Try to save the object to the database
3. If successful, redirect to `show` template
4. If unsuccessful, render `new` action (template - again)
    - Why would it not be successful? Validations did not pass for example.

Summary

- new action provides a form to be filled out to create a new resource
- create action accepts parameters passed in from filling out the form in the new action

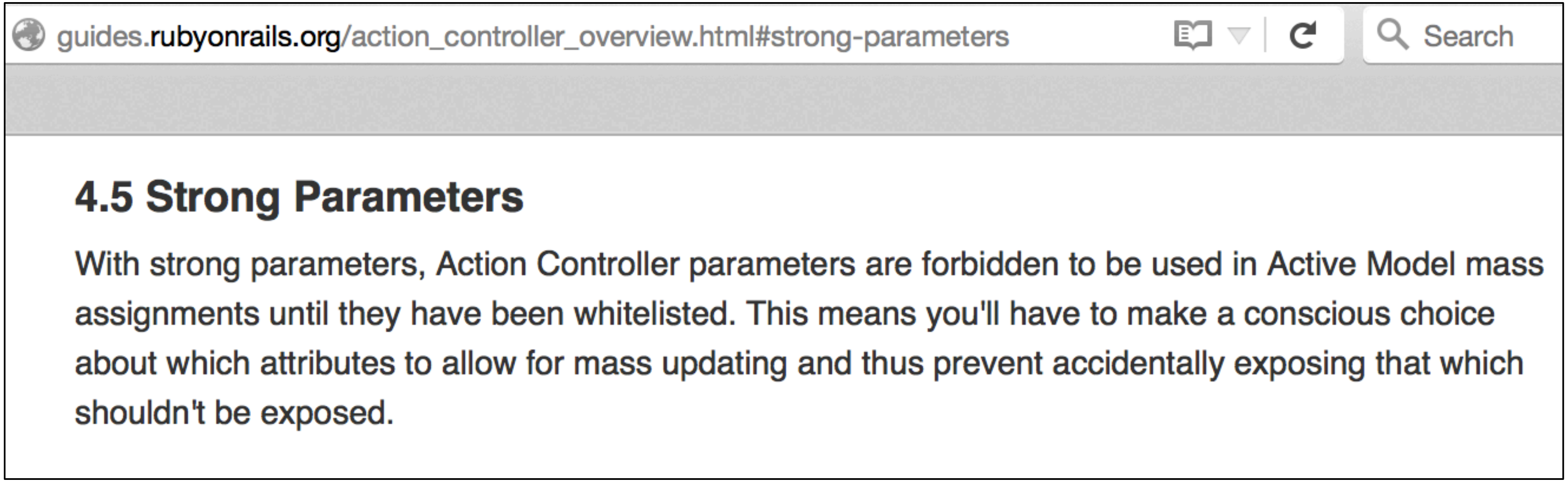
What's Next?

- Strong Parameters and Flash

In this lecture, we will discuss…

- Strong parameters
- Flash
- How create action works

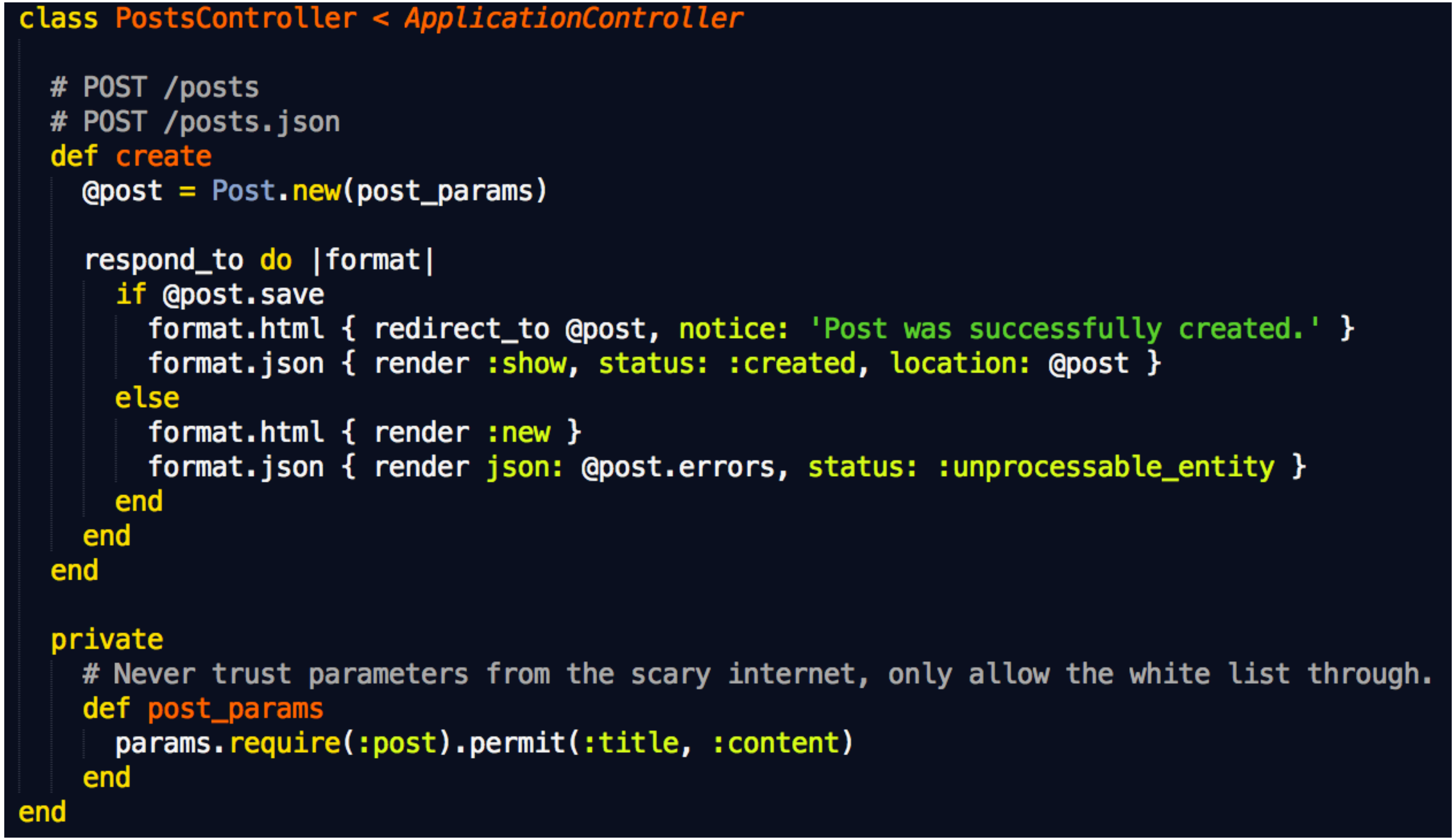
Strong Parameters


guides.rubyonrails.org/action_controller_overview.html#strong-parameters
Search

4.5 Strong Parameters

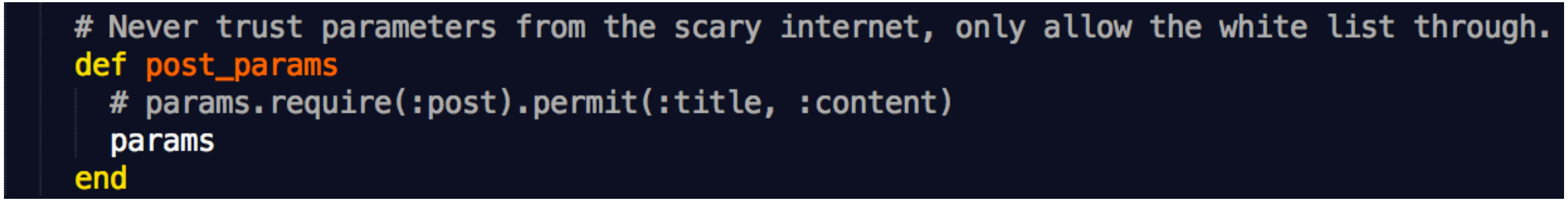
With strong parameters, Action Controller parameters are forbidden to be used in Active Model mass assignments until they have been whitelisted. This means you'll have to make a conscious choice about which attributes to allow for mass updating and thus prevent accidentally exposing that which shouldn't be exposed.

create action



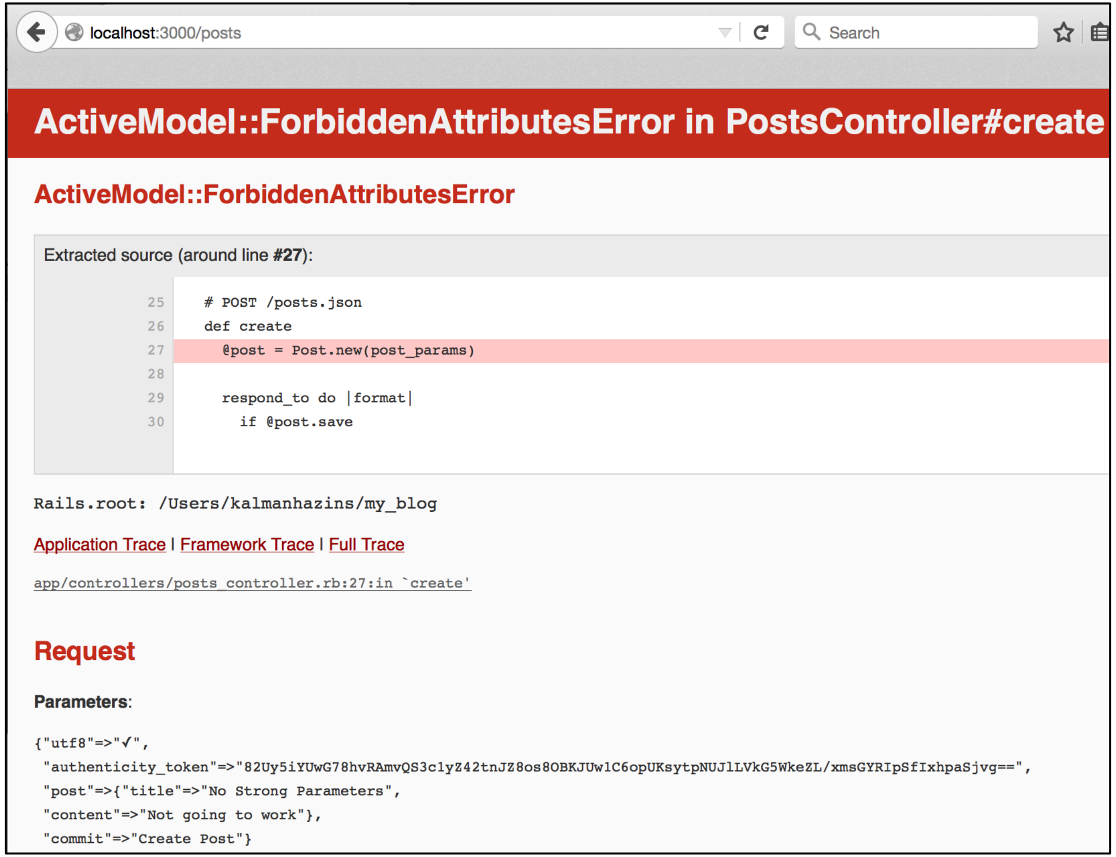
```
class PostsController < ApplicationController

  # POST /posts
  # POST /posts.json
  def create
    @post = Post.new(post_params)

    respond_to do |format|
      if @post.save
        format.html { redirect_to @post, notice: 'Post was successfully created.' }
        format.json { render :show, status: :created, location: @post }
      else
        format.html { render :new }
        format.json { render json: @post.errors, status: :unprocessable_entity }
      end
    end
  end

  private
    # Never trust parameters from the scary internet, only allow the white list through.
    def post_params
      params.require(:post).permit(:title, :content)
    end
end
```

Strong Parameters Not Implemented



```
# Never trust parameters from the scary internet, only allow the white list through.
def post_params
  # params.require(:post).permit(:title, :content)
  params
end
```

Strong Parameters Not Implemented



ActiveModel::ForbiddenAttributesError in PostsController#create

ActiveModel::ForbiddenAttributesError

Extracted source (around line **#27**):

```
# POST /posts.json
def create
  @post = Post.new(post_params)

  respond_to do |format|
    if @post.save
```

```
Rails.root: /Users/kalmanhazins/my_blog
```

Application Trace | Framework Trace | Full Trace

```
app/controllers/posts_controller.rb:27:in `create'
```

Request

Parameters:

```
{"utf8"=>"✓",
 "authenticity_token"=>"82Uy5iYUwG78hvRAmvQS3c1yZ42tnJZ8os8OBKJUw1C6opUKsytpNUJlLVkG5WkeZL/xmsGYRIpSfIxhpaSjvg==",
 "post"=>{"title"=>"No Strong Parameters",
 "content"=>"Not going to work"},
 "commit"=>"Create Post"}
```

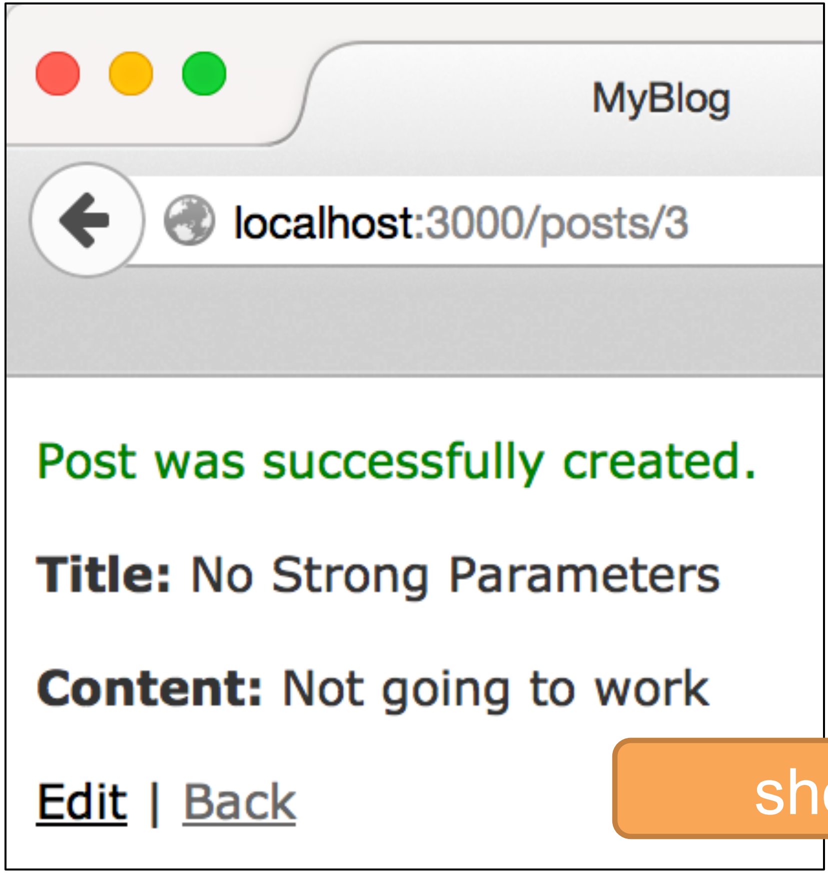
Flash

- **Problem:** We want to *redirect* a user to a different page on our site, but at the same time give him some sort of a message? For example, “Post created!”
- **Solution:** flash – a hash where the data you put in persists for exactly ONE request AFTER the current request.

Flash

- You can put your content into flash by doing `flash[:attribute] = value`
- Two very common attributes are `:notice` (good) and `:alert` (bad)
- These are so common in fact, that the `redirect_to` takes a `:notice` or `:alert` keys

create action



MyBlog

localhost:3000/posts/3

Post was successfully created.

Title: No Strong Parameters

Content: Not going to work

Edit | Back

show.html.erb (with a notice)

Summary

- ✧ Strong parameters requires you to whitelist the parameters that you intend to create/update
- ✧ Flash persists for exactly one request after the current request/response cycle

What's Next?

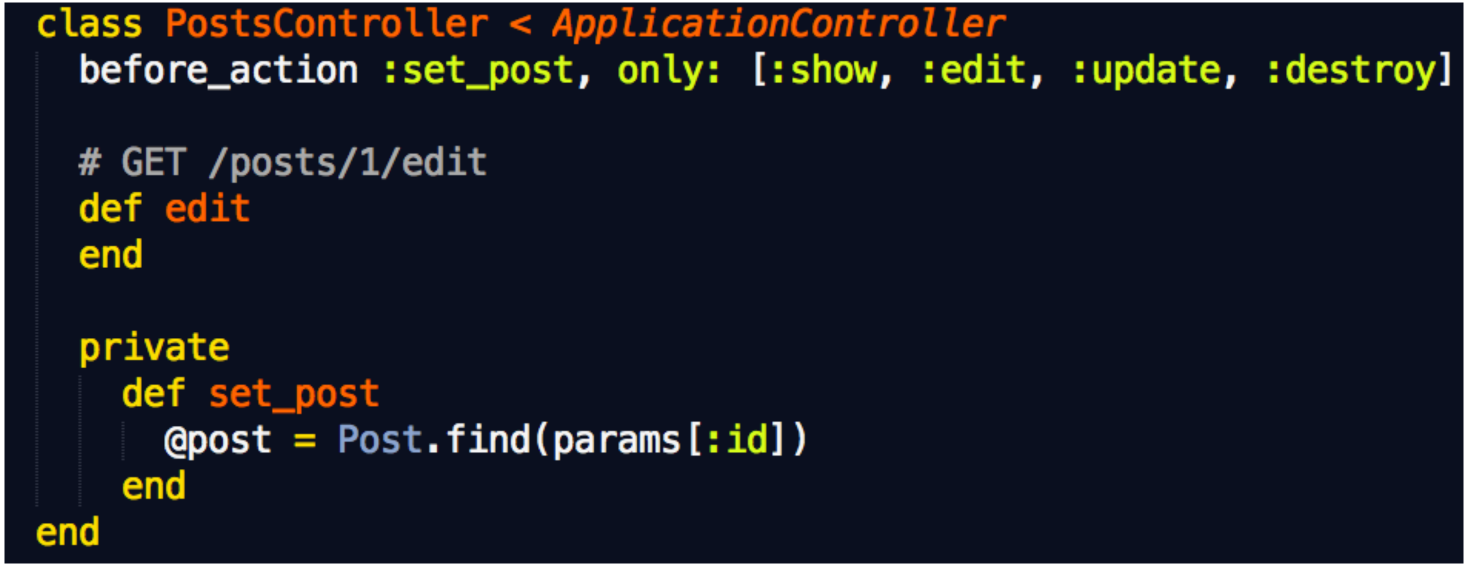
- ✧ edit and update actions

In this lecture, we will discuss…

- edit action
- update action

Examining Seven Actions – edit

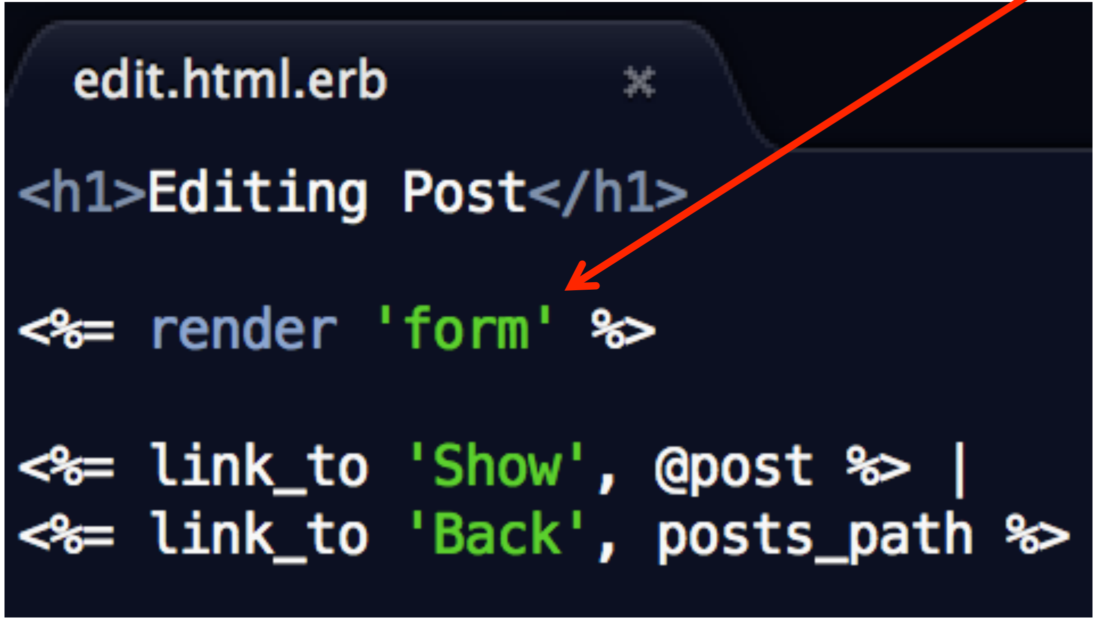
1. Retrieve a post object based on the `id` provided (as part of the URI)
2. (Implicit) Look for **`edit.html.erb`**



```ruby
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # GET /posts/1/edit
  def edit
  end

  private
    def set_post
      @post = Post.find(params[:id])
    end
end
```

edit.html.erb

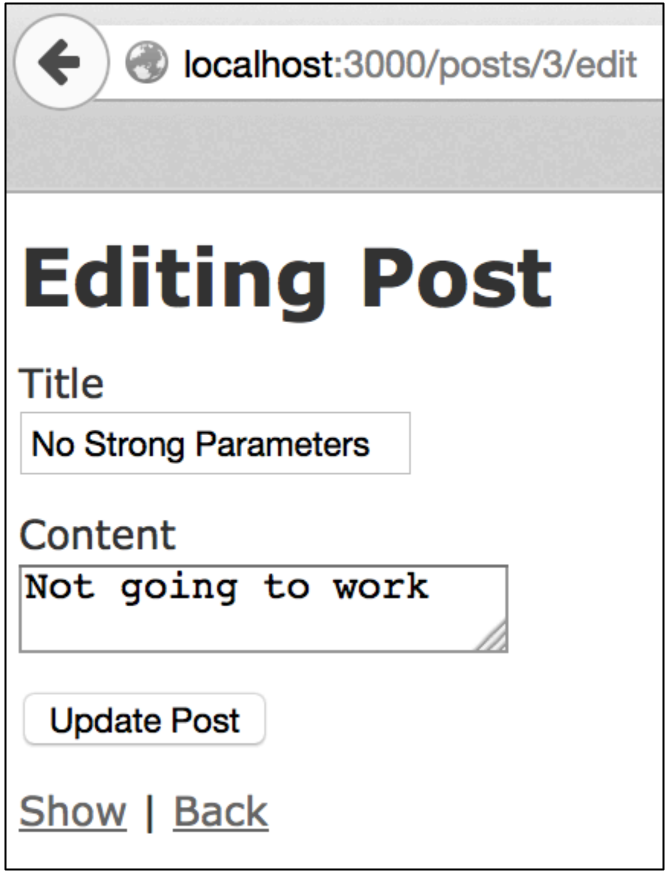
Partial – explained later



edit.html.erb

```
<h1>Editing Post</h1>

<%= render 'form' %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

edit.html.erb


localhost:3000/posts/3/edit

Editing Post

Title

No Strong Parameters

Content

Not going to work

Update Post

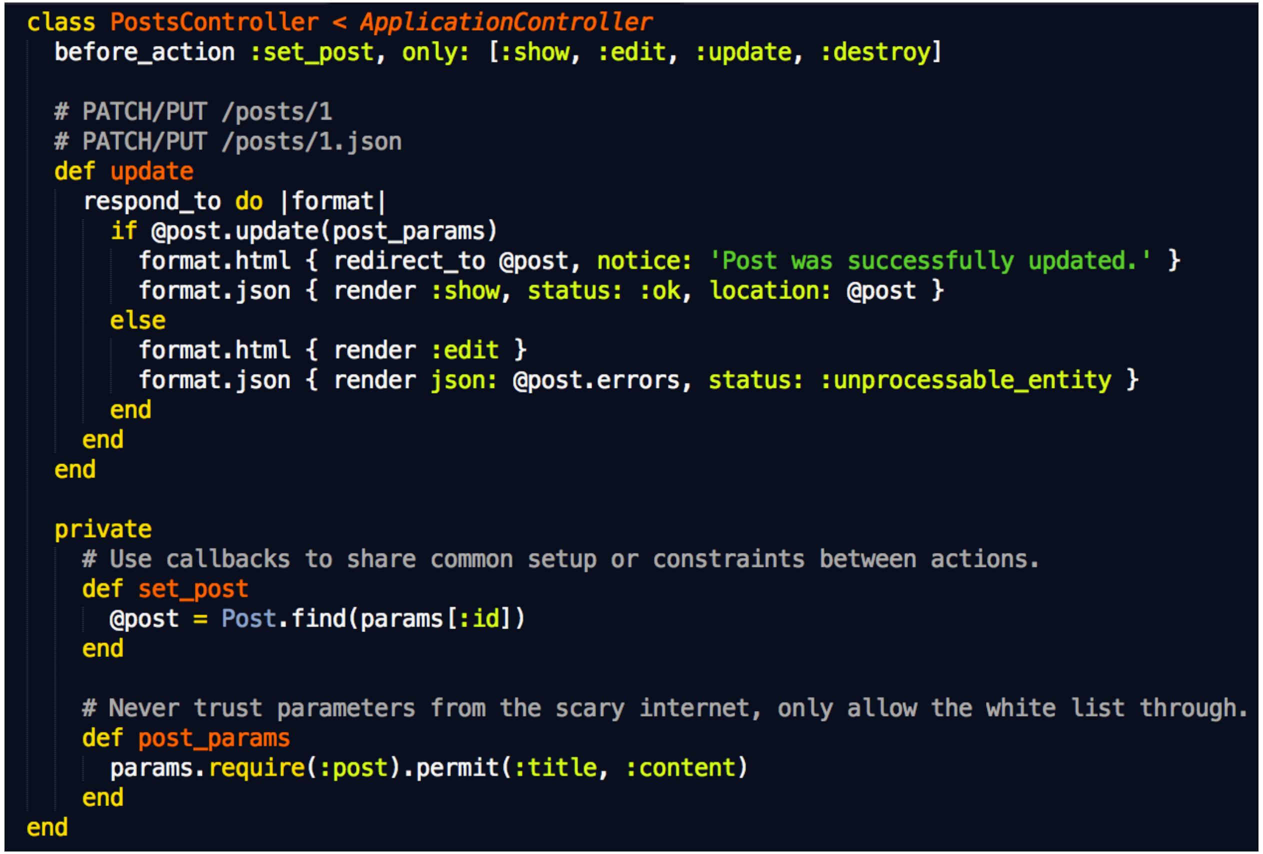
Show | Back

This looks remarkably similar to new…

Examining Seven Actions – update

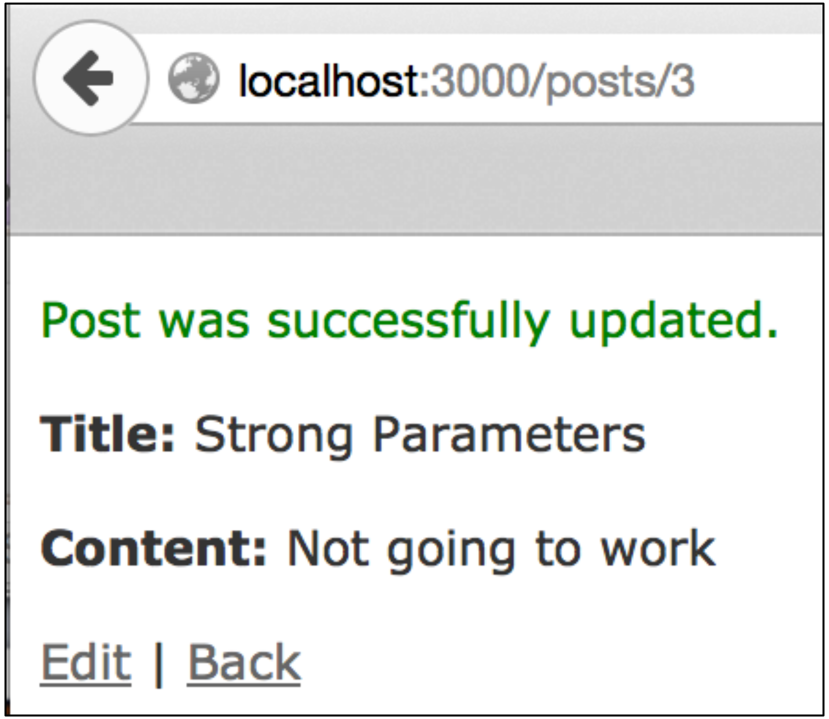
1. Retrieve an existing post using `id` parameter
2. Update post object with (**strong**) parameters that were passed from the `edit` form
3. Try to (re)save the object to the database
4. If successful, redirect to `show` template
5. If unsuccessful, render `edit` action (template) again

update action



```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :destroy]

  # PATCH/PUT /posts/1
  # PATCH/PUT /posts/1.json
  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to @post, notice: 'Post was successfully updated.' }
        format.json { render :show, status: :ok, location: @post }
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unprocessable_entity }
      end
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_post
      @post = Post.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list through.
    def post_params
      params.require(:post).permit(:title, :content)
    end
end
```

update action



localhost:3000/posts/3

Post was successfully updated.

Title: Strong Parameters

Content: Not going to work

Edit | Back

show.html.erb (with a notice)

Summary

- edit/update is very similar to new/create except there is an id of an existing resource that is being kept track of
- Strong parameters apply to updating a resource as well as creating one

What’s Next?

- Partials

In this lecture, we will discuss…

- ✧ Partials
- ✧ How the “form” partial works

Partials: DRY (Don’t Repeat Yourself)

- ✧ Rails encourages the DRY principle
- ✧ We already know about the `application.html.erb`, which enables you to maintain layout code for the entire application in one place (more on this later)
- ✧ It would also be nice to reuse snippets of view code in multiple templates
- ✧ For example, `edit` and `new` forms – are they really that much different?

Partials

- ✧ Partials are similar to regular templates, but they have a more refined set of capabilities
- ✧ Named with underscore (_) as the leading character
- ✧ Rendered with `render 'partialname'` (no underscore)
- ✧ `render` also accepts a second argument, a hash of **local** variables used in the partial

Object Partial

- Similar to passing local variables, you can also render a specific object
- `<%= render @post %>` will render a partial in `app/views/posts/_post.html.erb` and automatically assign a local variable `post`

Convention Over Configuration

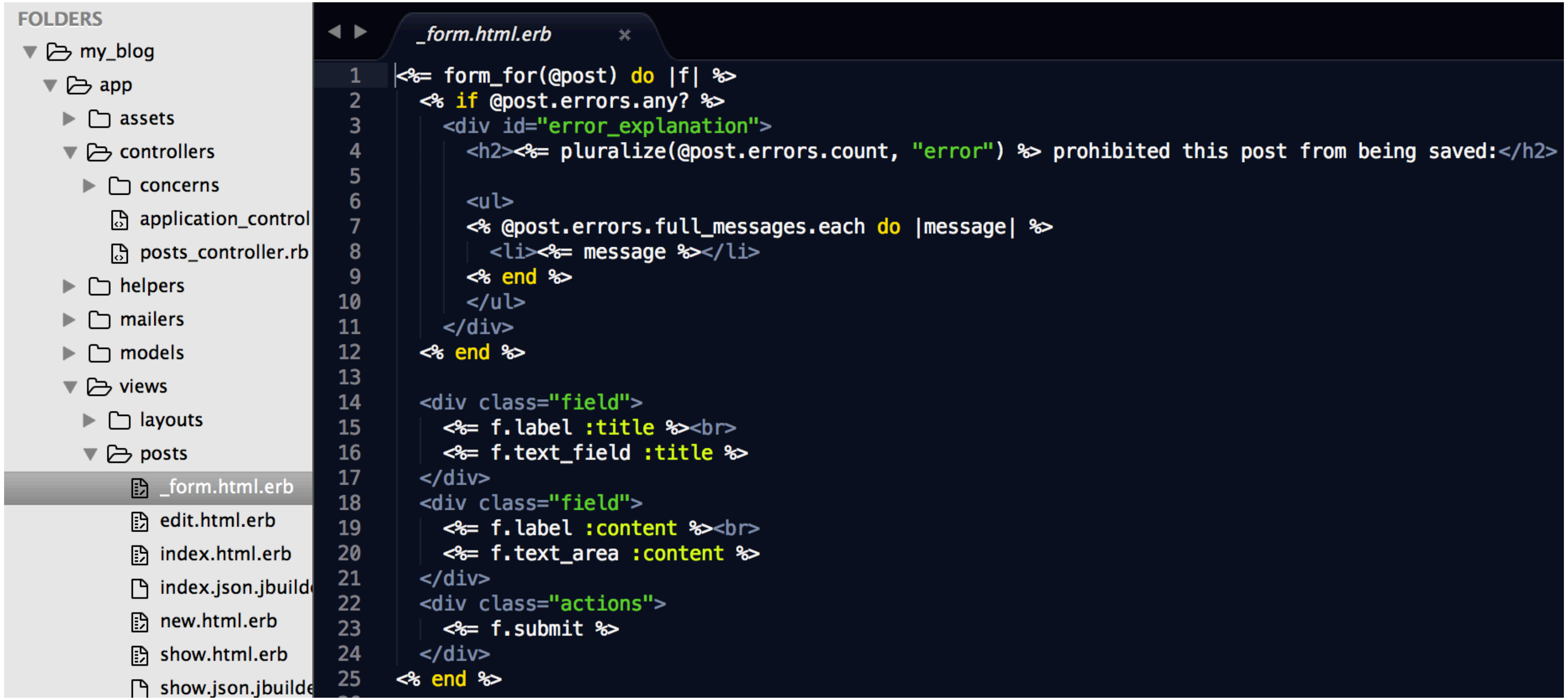
Rendering Collection of Partials

```
<%= render @posts %>
```

is equivalent to

```
<% @posts.each do |post| %>
  <%= render post %>
<% end %>
```

_form.html.erb – display errors

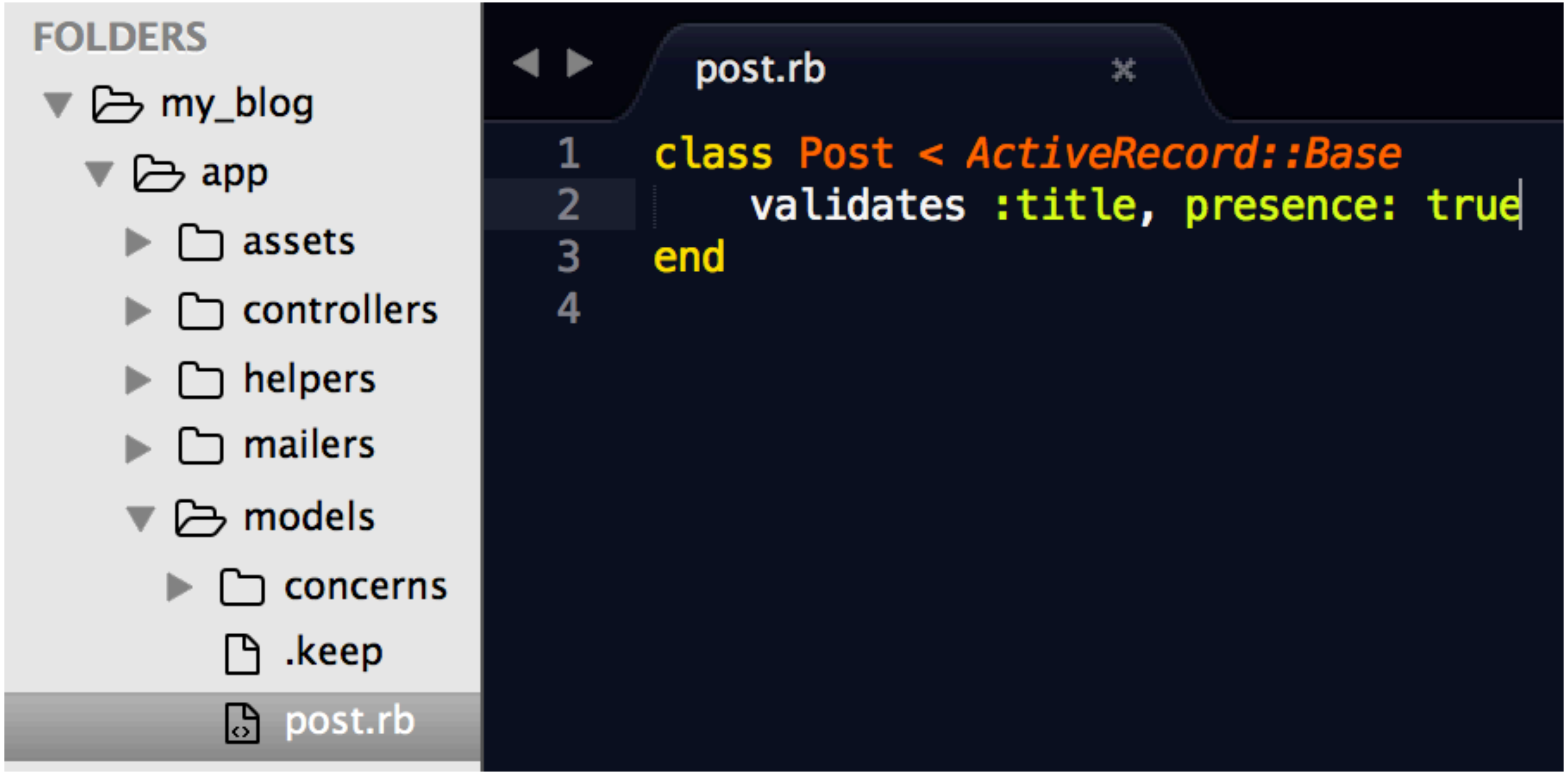


FOLDERS
- my_blog
  - app
    - assets
    - controllers
      - concerns
      - application_control
      - posts_controller.rb
    - helpers
    - mailers
    - models
    - views
      - layouts
      - posts
        - _form.html.erb
        - edit.html.erb
        - index.html.erb
        - index.json.jbuild
        - new.html.erb
        - show.html.erb
        - show.json.jbuilde

_form.html.erb

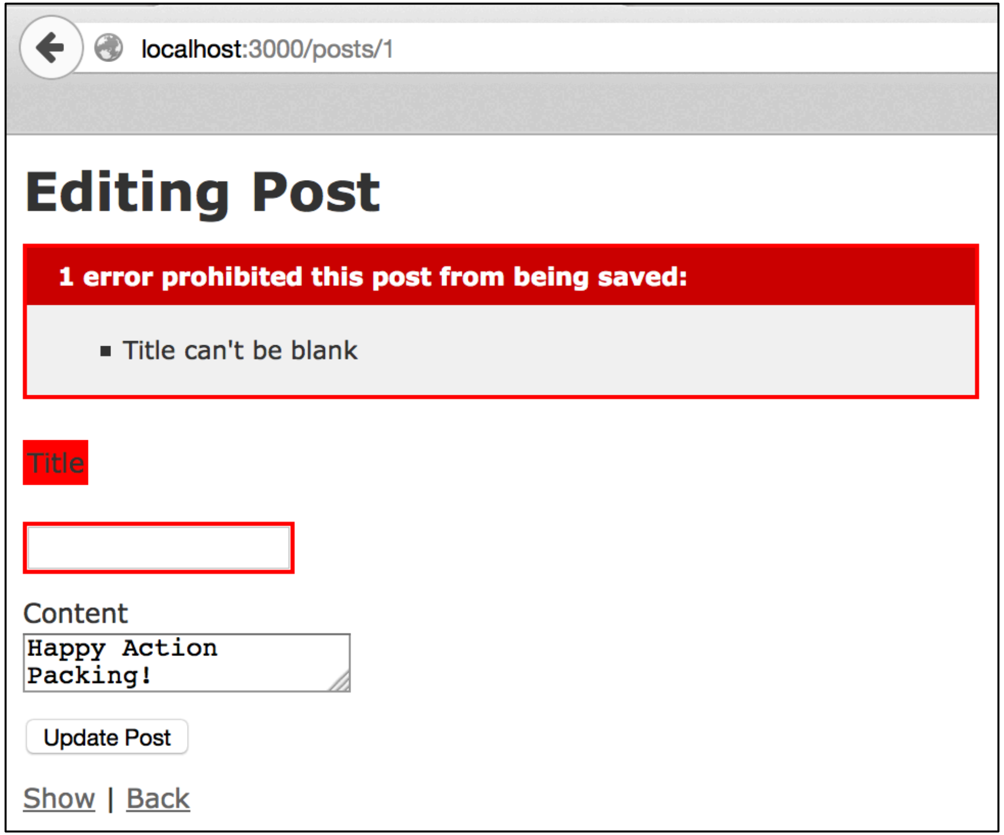
```erb
<%= form_for(@post) do |f| %>
  <% if @post.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@post.errors.count, "error") %> prohibited this post from being saved:</h2>

      <ul>
      <% @post.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br>
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Require Presence of Title



```ruby
class Post < ActiveRecord::Base
    validates :title, presence: true
end
```

_form.html.erb – display errors



localhost:3000/posts/1

Editing Post

1 error prohibited this post from being saved:

- Title can't be blank

Title

Content

```
Happy Action
Packing!
```

Summary

- Partial is a snippet of reusable template that has an underscore in its name and accepts parameters when rendered

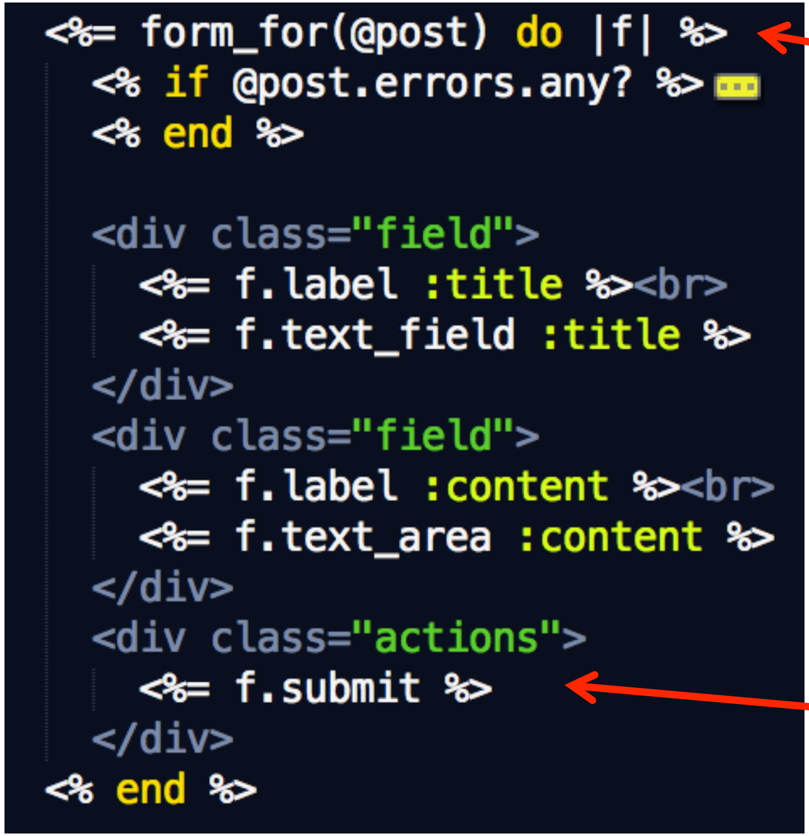
What's Next?

- Form helpers and Layouts

In this lecture, we will discuss…

- Form helpers
- Layouts

_form.html.erb



```
<%= form_for(@post) do |f| %>
  <% if @post.errors.any? %> ...
  <% end %>

  <div class="field">
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br>
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Form with parameters that match up with model's attributes

Submit button for submitting the form

Form Helpers

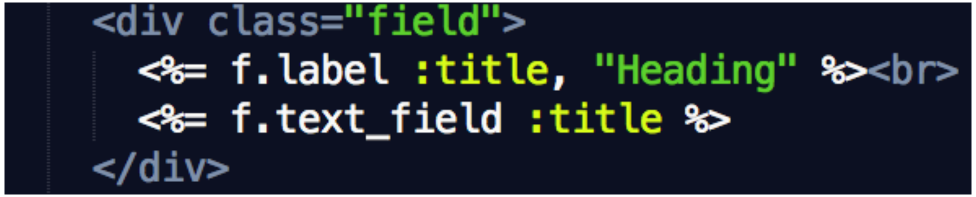
✧ `form_for`

- Generates a form tag for passed in object
- Unlike a regular HTML form, Rails uses POST by default
- This of course makes a lot of sense:
  1. Your password is not passed as part of your URL
  2. Anything that will end up modifying data on the server should definitely be a POST and not GET

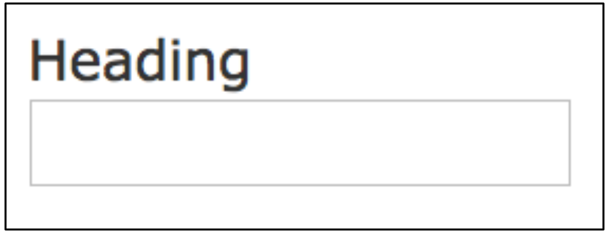
Form helpers – f.label

✧ `f.label`

- Outputs HTML label tag for the provided attribute
- To customize label description, pass in a string as a second parameter

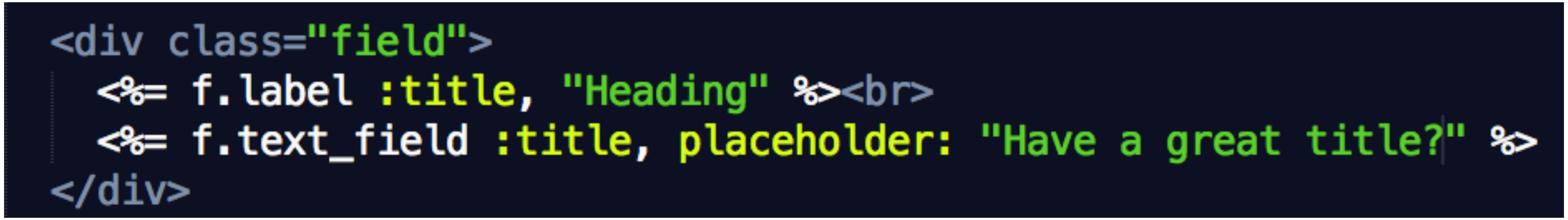


```
<div class="field">
  <%= f.label :title, "Heading" %><br>
  <%= f.text_field :title %>
</div>
```

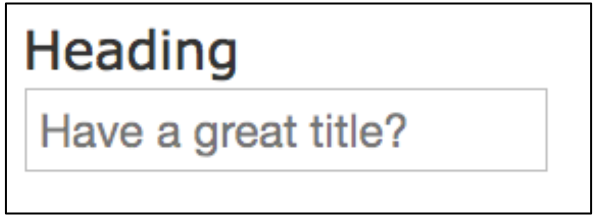


Form Helpers – f.text_field

- `f.text_field`
  - Generates input type="text" field
  - Use `:placeholder` hash entry to specify a placeholder (hint) to be displayed inside the field until the user provides a value



```
<div class="field">
  <%= f.label :title, "Heading" %><br>
  <%= f.text_field :title, placeholder: "Have a great title?" %>
</div>
```

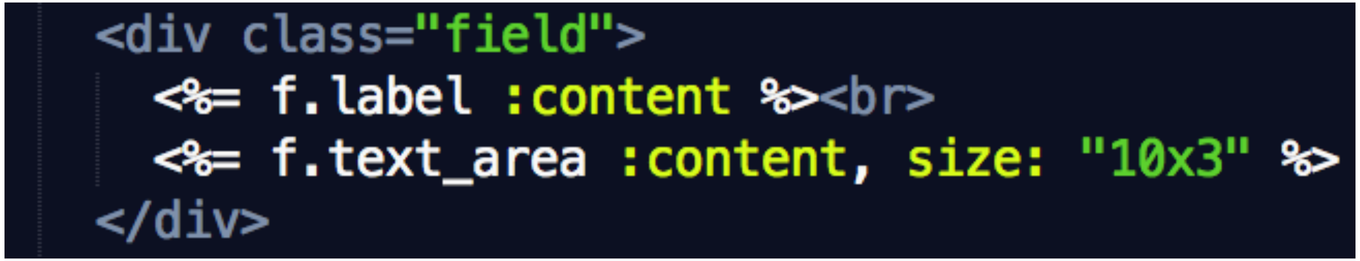


Heading

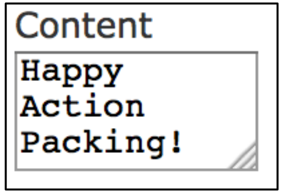
Have a great title?

Form Helpers – f.text_area

- `f.text_area`
  - Similar to `f.text_field`, but for a text area instead of a text field input (default: 40 cols x 20 rows)
  - Can specify a different size (colsXrows) with a :size attribute



```
<div class="field">
  <%= f.label :content %><br>
  <%= f.text_area :content, size: "10x3" %>
</div>
```



Content

```
Happy
Action
Packing!
```

Date Helpers

- `f.date_select`
  - Set of select tags (year, month, day) pre-selected for accessing an attribute in the DB. Many formatting options `f.time_select`
- `f.datetime_select`
- `distance_of_time_in_words_to_now`
- And many many more…
- See ActionView::Helpers::DateHelper docs
  - http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html

Form Helpers – Others

- `search_field`
- `telephone_field`
- `url_field`
- `email_field`
- `number_field`
- `range_field`

Some of these are browser-dependent – will take advantage of the browsers that are ready for prime time and will still look okay in others…

Form Helpers – f.submit

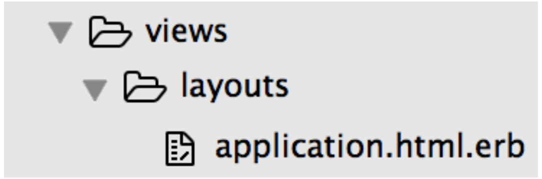
✧ `f.submit`

- Submit button
- Accepts the name of the submit button as its first argument
- If you don't provide a name – generates one based on the model and type of action, e.g. "Create Post" or "Update Post"

http://guides.rubyonrails.org/form_helpers.html

More on Layouts

1. Layout named `application.html.erb` is applied by default as a shell for any view template



2. Layout that matches the name of a controller is applied if present (overriding 1. above)

3. You can use `layout` method inside controller (outside any action) to set a layout for the entire controller

```
layout 'some_layout'
```

Layouts During Rendering

- ✧ You can include a layout for a specific action with an explicit call to `render` inside the action
  `render layout: ‘my_layout’`
- ✧ If you don’t want a layout (for some reason) – just pass false instead of layout name `render layout: false`

Summary

- ✧ Form helpers are a quick way to generate forms as well as form elements
- ✧ Layouts let you display a common “shell” around application template or around particular actions or resources