This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 37 Star 132 Fork 455

Code Issues 1 Pull requests 3 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / rails / readings / mailing-1.md Find file Copy path

mfeniseycopes Replace deprecated '#deliver' with '#deliver_now' 4661695 29 days ago

3 contributors

224 lines (167 sloc) 6.92 KB Raw Blame History

Action Mailer Basics

Action Mailer allows you to send emails from your application. There are two parts:

- A mailer class extending `ActionMailer::Base`. This works similarly to our `ApplicationController` extending `ActionController::Base`. Mailers live in `app/mailers`.
- Views, which live in `app/views/[mailer_name]`.

Your first mailer

Generate the Mailer

```
$ rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create    app/views/user_mailer
invoke  test_unit
create    test/functional/user_mailer_test.rb
```

So we have the mailer, the views, and the tests.

Edit the mailer

`app/mailers/user_mailer.rb` contains an empty mailer:

```ruby
class UserMailer < ActionMailer::Base
  default from: 'from@example.com'
end
```

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```ruby
# app/mailers/user_mailer.rb
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: user.email, subject: 'Welcome to My Awesome Site')
  end
  def reminder_email(user)
    # ...
  end

  # other emails...
end
```

Somewhat like a controller, this adds a mailer action that will send a `welcome_email` to the passed user. We set the `to` and the `subject` (as well as `from`; which is set from the default above). The content of the email lives in a view (more in a sec) which will be rendered.

You can also set `cc` and `bcc` attributes. To send to multiple emails, use an array of email strings.

You can put the name of the recipient in the email address like so: `"Ned Ruggeri <ned@appacademy.io>"`. This is a nice way of personalizing your email. Likewise you should personalize the sender: `"App Academy <contact@appacademy.io>"`.

The `mail` method returns an email object (of type `Mail::Message`); it doesn't mail it though. The caller of `UserMailer#welcome_email` will then call `#deliver_now` on the `Message` object:

```
# app/controllers/users_controller.rb

  def create
    u = User.create(user_params)

    msg = UserMailer.welcome_email(u)
    msg.deliver_now

    render :root
  end
```

Create a mailer view

Wait. What about the content? Create a file called `welcome_email.html.erb` in `app/views/user_mailer/`. This will be the template used for the email, formatted in HTML:

```
<!-- app/views/user_mailer/welcome_email.html.erb -->

<!DOCTYPE html>
<html>
  <head>
    <meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
    <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br/>
    </p>
    <p>
      To login to the site, just follow this link: <%= @url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

Just like controllers, any instance variables we define in the method become available for use in the views.

It is also a good idea to make a text version for this email. People who don't like HTML emails should have a text version to look at. Also, spam filters will ding you if you don't have a text version; a lot of email gets flagged as spam, so this is a problem.

To do this, create a file called `welcome_email.text.erb` in `app/views/user_mailer/`:

```
<%# app/views/user_mailer/welcome_email.text.erb %>

Welcome to example.com, <%= @user.name %>
===============================================

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!
```

When you call the `mail` method now, Action Mailer will detect the two templates (text and HTML) and automatically generate a `multipart/alternative` email; the user's email client will be able to choose between the two formats.

Mailing is slow

Sending out an email often takes up to 1sec. This is kind of slow. In particular, if your Rails app is sending a mail as part of a controller action, the user will have to wait an extra second for the HTTP response to be sent.

We'll eventually learn how to "batch up" and send emails offline, but for now just know that if you try to send 100 emails in a controller method, you're going to have trouble responding to requests promptly.

Mailing deets

Adding attachments

Adding attachments is pretty simple, just add new key/value pairs to the hash-like `attachments` variable. This is provided in each instance of `ActionMailer::Base`.

```ruby
# app/mailers/user_mailer.rb

  def welcome_email
    attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
    # ...
  end
```

Generating URLs in Action Mailer Views

You will probably want to embed URLs into your mailer views just like for your controller views. Somewhat oddly, you must set an option in `config/environments/production.rb` (and `config/environments/development.rb` for development) so that the mailer knows the base url of the website:

```ruby
# app/config/environments/development.rb
config.action_mailer.default_url_options = { host: 'localhost:3000' }

# app/config/environments/production.rb
config.action_mailer.default_url_options = { host: 'www.production-domain.com' }
```

You would think that the Rails app knows the hostname (e.g., it doesn't need you to set this for `*_url` methods in controller views). "Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the :host parameter yourself." (Rails Guides - Action Mailer Basics)

Make sure to (continue to) use the `*_url` form of the url helpers, since when the user opens their email, the email needs to contain the full hostname of the site to know what host to send the request to. This is because the email is being opened by a mail-client that isn't on the same domain as your site (e.g., email is opened on gmail.com, link needs to point to appacademy.io).

Letter Opener

Testing email sending in development can be a PITA. Use the wonderful letter_opener gem. When running the development environment (your local machine), instead of sending an email out to the real world, letter_opener will instead pop open the "sent" email in the browser.

Setup is two lines:

```
# Gemfile
gem "letter_opener", group: :development

# config/environments/development.rb
config.action_mailer.delivery_method = :letter_opener
```

Resources

- ActionMailer::Base

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 37 Star 132 Fork 455

Code Issues 1 Pull requests 3 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / rails / readings / helpers.md Find file Copy path

loschorts reverse directory structure to go topics -> types c1e6862 on May 16, 2016

1 contributor

176 lines (137 sloc) 5.21 KB Raw Blame History

View helpers

When we write our view templates, we try to keep them DRY like our code. We've seen that we can do this using partials. Rails gives us another tool: view helpers.

View helpers are Ruby methods that can be called from your view. The job of view helpers is to hold view specific code that DRYs up templates. The most popular view helper is `link_to`. Other common helpers are for building forms (`form_for`), including assets (`javascript_include_tag`), etc. We'll see plenty more Rails helpers in our time.

We can also write our own helpers: they are stored in `app/helpers/#{controller}_helper.rb`. Any view logic that will be repeated should be broken out to a helper method.

It's common to write helper methods for repeated chunks of HTML you might use all over the place. Some examples might be to highlight a string, format a quote correctly, or display a user picture.

```ruby
# app/helpers/application_helper.rb
module ApplicationHelper
  def highlight(text)
    "<strong class=\"highlight\">#{h(text)}</strong>".html_safe
  end

  def picture_for(user)
    html = "<div class=\"user-picture\">"
    html += "<img src=\"#{user.picture_url}\" alt=\"#{h(user.name)}\">"
    html += "</div>"

    html.html_safe
  end
end
```

You can then use these helpers in all your views:

```erb
<!-- app/views/cats/show.html.erb -->
<p>
  How can one not like <%= highlight "cats" %>? They are my favorite!
</p>

<%= picture_for @cat %>
```

You should really lean on helpers to keep your templates clean and readable. View partials are useful for breaking up repeated larger bits of code, but repeated snippets of a few lines are best stored in helpers.

To organize your helper methods, Rails lets you make as many helper module files as necessary. They go in the folder `app/helpers`. All the methods in these modules will be available in every view; even something in `app/helpers/user_helpers` will be available in a `app/views/posts` view. You may put non-specifc helper methods in the `ApplicationHelper` module Rails has generated for you.

Escaping HTML

Rails escapes all HTML when it prints out content in ERB tags.

```
<%= '<p>This paragraph tag will be escaped</p>' %>
<!--
Outputs: &lt;p&gt;This paragraph tag will be escaped&lt;/p&gt;
-->
```

This prevents user generated HTML from sneaking into the page. It's the same idea as how `?` in a SQL query is meant to protect against a SQL injection attack. This prevents HTML injection.

In HTML injection, a malicious user embeds links or JavaScript in their uploaded content. Anyone who views that content will be affected. A common case is embedding a link to the hacker's site to improve SEO. In the worst case though, it could be an official-looking link to a phishing site or JavaScript that scrapes private user data from the page and sends it to the hacker's servers.

When making helper methods that generate HTML for us this becomes problematic. Your helper builds up a string, which should be inserted into the HTML unescaped.

Fortunately, Rails added a `.html_safe` method to the String class, which lets us bypass the escaping. Any string marked `html_safe` will not be escaped when inserted into a view.

```
<%= '<p>This will NOT be escaped</p>'.html_safe %>
<!--
Outputs: <p>This will NOT be escaped</p>
-->
```

So, for instance:

```
# app/helpers/application_helper.rb
module ApplicationHelper
  def highlight(text)
    "<strong class=\"highlight\">#{text}</strong>".html_safe
  end
end
```

wouldn't work without `html_safe`.

However, we're now at risk of an injection attack. Do we trust `text`? Could that come from the user? What if they try something sneaky?

Just like there is a way to mark a string as HTML safe so that it won't be escaped when inserted into a template, there's a way to escape an HTML unsafe string: you use the `html_escape` method (an alias, `h` is conveniently short and thus more common):

```
# app/helpers/application_helper.rb
module ApplicationHelper
  def highlight(text)
    "<strong class=\"highlight\">#{h(text)}</strong>".html_safe
  end
end
```

You can read more here.

Capturing a block in ERB

Sometimes we want to pass a lot of content into a helper. Rails conveniently lets us render ERB blocks into text using the `capture` method.

```
def long_quote(author, &block)
  text = capture(&block)

  html = "<blockquote cite=\"#{h(author)}\">"
  html += text
  html += "</blockquote>"

  html.html_safe
end
```

Alright, let's do it!

```
<%= long_quote("_whytheluckystiff") do %>
  <p>
    All you need to know thus far is that Ruby is basically built from
    sentences. They aren’t exactly English sentences. They are short
    collections of words and punctuation which encompass a single
    thought.
  </p>

  <p>
    Look ma, a <%= link_to "link",
    "http://en.wikipedia.org/wiki/Why_the_lucky_stiff" %>
  </p>

  <p>
    These sentences can form books. They can form pages. They can form
    entire novels, when strung together. Novels that can be read by
    humans, but also by computers.
  </p>
<% end %>
```

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 37 Star 132 Fork 455

Code Issues 1 Pull requests 3 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / rails / readings / layouts.md Find file Copy path

loschorts reverse directory structure to go topics -> types c1e6862 on May 16, 2016

1 contributor

114 lines (90 sloc) 3.12 KB Raw Blame History

Layouts

When Rails renders a view as a response, it does so by inserting the rendered template into a layout to form the complete response. This lets us focus on writing the page-specific information in the template, and not duplicate surrounding data like the `<html>`, `<head>`, etc tags, any header, navbar, or footer structure. You might also load site-wide JavaScript in the layout.

Layouts live in `app/views/layouts`; here's the default `application.html.erb` layout:

```
<!DOCTYPE html>
<html>
<head>
  <title>Testapp1</title>
  <%= stylesheet_link_tag    "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

Understanding `yield`

Within the context of a layout, `yield` identifies a section where content from the view should be inserted. The simplest way to use this is to have a single `yield`, into which the entire contents of the view currently being rendered is inserted (see above).

You can also create a layout with multiple yielding regions:

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

The main body of the view will always render into the unnamed `yield`. To render content into a named `yield`, you use the `content_for` method.

The `content_for` method allows you to insert content into a named `yield` block in your layout. For example, this view would work with the layout that you just saw:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

The result of rendering this page into the supplied layout would be this HTML:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

The `content_for` method is very helpful when your layout contains distinct regions such as sidebars and footers that should get their own blocks of content inserted. It's also useful for inserting tags that load page-specific JavaScript or CSS files into the header of an otherwise generic layout.

Asset tags

Layouts are typically responsible for directing the browser to load site-wide JavaScript and CSS. JavaScript and CSS source are not (typically) embedded in the HTML response; instead, the HTML contains `<script>` and `<link>` tags in the header that direct the browser to make subsequent requests for the JS and CSS files to load.

For instance, to direct the client to load `common.js`, you would use the `javascript_include_tag` helper:

```
javascript_include_tag "common"
# => <script src="/assets/common.js"></script>
```

Likewise, to tell the browser to use the css file `application.css`, you would use `stylesheet_link_tag`:

```
stylesheet_link_tag "application"
# => <link href="/assets/application.css" media="screen" rel="stylesheet" />
```

As you can see, both of these tags direct the browser to fetch another resource from the server. When we learn about the asset pipeline, we'll learn where these files are stored (`app/assets`) and how to add your own JS/CSS files.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 37 Star 132 Fork 455

Code Issues 1 Pull requests 3 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / rails / readings / partials.md Find file Copy path

mfeniseycopes Clarify reasoning for using locals 15c24af on Mar 21

2 contributors

161 lines (119 sloc) 4.64 KB Raw Blame History

Partials

Views can get really long and complicated. Just like you break up long methods into smaller ones, we break up views into smaller bits, called partials. Like small methods, partials are often reusable in different views, so they also help us keep our code DRY.

Naming Partials

To render a partial as part of a view, you use the `render` method within the view:

```
<h1>Page title</h1>

<p>
  Here's some content for you
</p>

<!-- insert standard footer -->
<%= render "footer" %>
```

This will render a template named `_footer.html.erb` and insert the rendered content at this point within the view. Note the leading underscore character: partials are named with a leading underscore to distinguish them from regular views, even though they are referred to without the underscore. This holds true even when you're pulling in a partial from another folder:

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

That code will pull in the partial from `app/views/shared/_footer.html.erb`.

Using Partials to Simplify Views

One way to use partials is to treat them as the equivalent of subroutines: as a way to move details out of a view so that you can grasp what's going on more easily. For example, you might have a view that looked like this:

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

<h1>Products</h1>

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

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

Here, the `_ad_banner.html.erb` and `_footer.html.erb` partials could contain content that is shared among many pages in your application. You don't need to see the details of these sections when you're concentrating on a particular page.

Passing Local Variables

Like methods, you can also pass local variables into partials, making them even more powerful and flexible. For example, you can use this technique to reduce duplication between new and edit pages, while still keeping a bit of distinct content:

```
<!-- app/views/user/new.html.erb -->
<%= render "form", user: @user, action: :new %>

<!-- app/views/user/edit.html.erb -->
<%= render "form", user: @user, action: :edit %>

<!-- app/views/user/_form.html.erb -->
<!-- Is this a new user to create, or an existing one to edit? -->
<% action_url = (action == :new) ? users_url : user_url(user) %>

<form action="<%= action_url %>" method="post">
  <% if action == :edit %>
    <input type="hidden" name="_method" value="patch">
  <% end %>
  <!-- inputs go here... -->
</form>
```

Rendering objects

It is typical to have to render model objects often. Here's the long way:

```
<% @cats.each do |cat| %>
  <%= render "cat", cat: cat %>
<% end %>
```

There is a shorthand for this:

```
<% @cats.each do |cat| %>
  <%= render cat %>
<% end %>
```

Rails will look at the `cat` value, see it is a `Cat` model object, and will look for a `_cat.html.erb` partial template to use. It will then render the template, setting a variable named `cat` inside the partial, so that you can use a more semantically meaningful name within:

```
<!-- app/views/shared/_cat.html.erb -->
<ul>
  <li>Name: <%= cat.name %></li>
  <li>Age: <%= cat.age %></li>
</ul>
```

How do you think it knows to set the `cat` variable? The same way it decides the partial to render: it looks at the class name of the model object, and uses that.

Rendering Collections

As seen above, partials are very useful in rendering collections. Just like there is a shortcut to render a model object, you can also easily render an array of model objects:

```
<%= render @cats %>
```

Rails will render the `_cat.html.erb` partial once for each item, setting the `cat` local variable as before.

Hints

Pass locals, don't use instance variables

Partials, like regular views, have access to controller instance variables. You could do the following:

```
<!-- app/views/user/new.html.erb -->
<!-- controller has set `@user` -->
<%= render "form" %>

<!-- app/views/user/_form.html.erb -->
<!-- Is this a new user to create, or an existing one to edit? -->
<% action_url = @user.persisted? ? user_url(@user) : users_url %>

<form action="<%= action_url %>" method="post">
  <% if @user.persisted? %>
    <input type="hidden" name="_method" value="put">
  <% end %>
  <!-- inputs go here... -->
</form>
```

However convenient it may seem to just use instance variables when creating partials, remember that we want to make partials flexible and non-reliant on sources of data defined outside of their scope. This also simplifies our debugging process. Failing to provide a required instance variable can produce confusing errors, rather than the unambiguous `"no local variable defined"` error raised when a local variable is undefined.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

This repository Search Pull requests Issues Gist

appacademy / curriculum Private Watch 37 Star 132 Fork 455

Code Issues 1 Pull requests 3 Projects 0 Wiki Pulse Graphs

Branch: master curriculum / rails / readings / rails_nontech.md Find file Copy path

ragegage add rails nontech reading c63cb57 on Dec 22, 2016

1 contributor

230 lines (174 sloc) 10.8 KB Raw Blame History

Ruby on Rails

Ruby on Rails is a web application framework written in Ruby. It was developed by David Heinemeier Hansson and first released to the public in July 2004.

Not only is Rails popular with start-ups for its ease in deploying to production, but a number of larger companies like Twitter, Hulu, and Kickstarter use Rails in their stack as well.

Rails became popular for a number of reasons.

One major reason is that Ruby is a very readable, versatile, and elegant language. Ruby was designed to be a "joy to use" and that can be seen in its clean, concise code that reads like English.

Another reason why Rails is so popular is that it is an open-source framework and has a large, active community. Not only is the framework itself well maintained, but there are also a lot of external libraries available through RubyGems that can be easily added to a Rails application.

Also, Rails follows a number of design principles that contribute to its efficacy in getting an application pushed to production quickly.

Design Principles

MVC

Rails is an MVC framework. MVC, or model-view-controller, is a design pattern that divides an application into three basic parts, each responsible for handling a different task.

- The model is concerned with retrieval of information from storage, as well as any logic done to that information.
- The view is concerned with presenting information.
- The controller is concerned with constructing the server's HTTP response and sends commands to the model to fetch or update information, and to the view to display information. It acts as the interface between the model and the view.

Convention Over Configuration

Rails introduced the concept of convention over configuration (CoC). The idea behind CoC is that the default configuration is already programmed into the framework, and developers only have to be specific upon doing something unconventional.

One convention that Rails makes extensive use of is the naming of various parts of the application. For instance, if we have a `cats` database table, the default model name is `Cat`, and the default controller is `CatsController`. By naming in this way, Rails will automatically know how to load the application to tie all these pieces together.

CoC circumvents having to write a lot of boilerplate configuration. Part of why Rails is popular is that it allows developers to create an application very quickly due to all of this configuration being handled. It also means that any Rails developer can look at any project and know how to navigate it with less extensive documentation.

REST

REST, or Representational State Transfer, is an architectural style for web services. A RESTful web service allows for access to web resources (e.g., a document, image, web page -- practically anything accessible on the internet!) using standard operations, such as HTTP's `GET`, `POST`, `PATCH`, or `DELETE`. The idea behind REST is that each one of these operations, in combination with a particular resource identifier, will have a predictable, standardized result. For instance, a `GET` request to a resource with a specified `id` is expected to return data on that specific item. This is true no matter what the resource is.

RESTful web services are stateless; this means that each request/response cycle is independent of any other request/response cycle. The server will not retain any session information over multiple requests. However, data can be cached on the client side.

Rails defaults to a RESTful API using its `resources` method, which automatically generates routes by linking HTTP verbs and URIs to controller actions.

Components

ActiveRecord

All models in Rails inherit from ActiveRecord. ActiveRecord is an ORM, or Object Relational Mapping. It allows us to represent data from a database as Ruby objects, which can then easily be presented or manipulated. In addition, ActiveRecord contains a number of methods which implement basic CRUD (Create, Read, Update, Delete) functionality and which mirror SQL queries.

ActionController

ActionController handles the controller logic. Among other things, ActionController allows for access to various cookies, which can store small amounts of data such as information about the session or brief error messages that can persist across requests.

It can also provide protection from cross-site request forgery (CSRF) attacks by checking for the presence of authenticity tokens in requests that could modify data (e.g., `POST` or `PATCH` requests).

ActionView

ActionView is responsible for rendering the views. ActionView templates can take several forms. One of the supported file types is ERB, or Embedded Ruby. ERB files are a mixture of HTML and Ruby code. Writing Ruby code in the views is very helpful - to generate a list of items, just write a loop! Rails also has a number of built-in helper methods that generate HTML that can be added to a view with ERB.

Another supported file type is a builder file. Instead of rendering HTML, a view could instead render XML or JSON. This is incredibly useful for a route that only returns data that will then be parsed later by more code.

Rake

Rake is a utility tool packaged with Rails that handles task management. It can be thought of as a scripting tool for Rails, used to carry out administrative tasks such as migrating a database and displaying routes.

Custom rake tasks are easy to create. They're written in Ruby and can access any part of your Rails app, such as a model.

Rack

Rack is a piece of middleware that sits between the Rails router and the web server (e.g., WEBrick or Puma), receiving HTTP requests from the server and generating responses using information from the router. There are many different Ruby frameworks that expect a request to come in in a certain way and format their responses in a certain way, and many different web servers that expect the same. Rack ensures that every server can interface with every framework.

WEBrick

WEBrick is the HTTP server that comes standard with Rails 4, though Rails 5 ships with the Puma HTTP server. WEBrick is written entirely in Ruby and is reliable and easy to use, but it is single threaded - that is, it can only handle a single request at a time. This means it doesn't scale well and can experience a lot of delay. This makes it ideal for development but not for a production environment.

Puma, however, is very fast and multi-threaded, able to handle requests concurrently, making it a good choice for a production server.

Other Frameworks

Sinatra (Ruby)

While both frameworks are coded in Ruby, Sinatra is a lot smaller and simpler than Rails, and can be very useful for small apps. It's easy to make simple requests and present information, but code can get large and messy when trying to accomplish more complex tasks, such as interacting with a database or presenting complex views. To make Sinatra scaleable, we'd have to essentially implement code that comes packaged with Rails, such as ActiveRecord. For large applications, it is probably easier to use a more robust framework.

Node.js (JavaScript)

Node.js is part of the MEAN stack. The MEAN stack consists of four JavaScript technologies (MongoDB, Express.js, AngularJS, Node.js) that can be used together to create a full-stack app, thus allowing developers to only use one language in all parts of their applications.

Node isn't exactly a framework. It's a runtime environment, which means that it can run JavaScript code. Like Ruby's gems, Node has packages called modules -- but unlike Rails, which comes with certain gems and functionality packaged into it, a Node user has to install any modules they want and then write the code to integrate them by hand.

Node offers a lot of flexibility because a developer can choose exactly which modules they want to include, but developing in Rails can be significantly faster because Rails has such strong defaults.

Django (Python)

Django is also an MVC framework, but where Rails follows the principles of CoC and leverages the readability of Ruby, Django is written in Python, which follows a principle of "explicit is better than implicit." This means that Django code is easy to read and understand because every bit of functionality is explicitly written into every file - but it also means that a developer will have to write all that code.

Using Rails over Django comes down to preference. Some developers prefer having a lot of the boilerplate done for them, and some prefer seeing and writing exactly how their server will construct a response.

Spring MVC (Java)

Spring MVC is a Java framework. There are a lot of resources available for Spring, and more resources are continuing to be developed. However, Spring can be very slow. In order for code changes to take effect, Spring has to be completely restarted, which can take a lot of time - in contrast with Rails, which allows for most code changes to be seen upon simply refreshing the page. It has very few built-in tools. It also suffers from lack of scalability, because any kind of I/O holds up a thread and slows down the application. In today's world of single-page apps with constant user input, this is a serious drawback.

Play! (Scala)

Play! can be written in either Scala or Java, and was built to be a more dynamic framework that Java developers would be able to use. Unlike Spring, it supports hot reload, allowing for changes to the code to be seen upon refresh. It also has a lot more built-in functionality and is much more responsive than Spring.

One of the real strengths of Rails is its gem library. Play! suffers from a lack of plugins. However, Scala is a faster language than Ruby, and that means that Play! can be faster and therefore more scalable than Rails.

ASP.NET MVC(.NET)

ASP.NET is developed by Microsoft and meant to be an MVC framework for .NET languages. Many .NET languages, such as C#, are supported. As a Microsoft product, it's particularly suited for development on Windows machines, unlike Rails, whose gems expect Unix-like behavior. A lot of enterprise projects are already built with .NET code, and it's easy to build an ASP.NET framework over them.

Rails and ASP.NET are fairly similar, and often the decision to use one over the other comes down to language preference between Ruby and C# or other .NET languages.

Laravel (PHP)

PHP is the most popular server-side language in use, and Laravel, although fairly young, is its most popular framework. It was developed with modern architecture in mind, including support for user authentication, recurring billing services, and scheduling periodically executed tasks. PHP is an easier language to learn than Ruby, but is more verbose and was designed to be a scripting language. Laravel requires the code it runs to be more explicit; it lacks the power of Rails's implicit conventions. However, for this reason, it is more flexible because Rails is such a structured framework.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About