

```
<%= 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:

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 -->

    Name: <%= cat.name %>
    Age: <%= cat.age %>
```

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

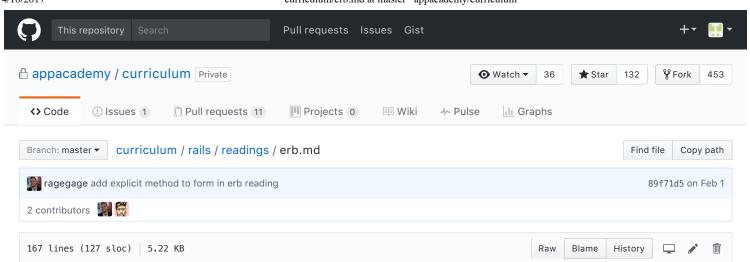
Pass locals, don't use instance variables

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

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

()



Templates

As we've discussed, controllers cause templates to be rendered by calling the render method. But how are templates structured? Answer: with HTML and ERB (and love!).

ERB (embedded Ruby)

Templates consist of HTML, but they are augmented with Ruby code. ERB templates are pretty simple:

- <% ruby_code_here %> executes Ruby code that does not return anything. For example, conditions, loops or blocks
- <= % is used when you want to embed the return value into the template. i.e. Something that will actually show up in the HTML.

For example:

The loop is setup in regular embedding tags <% >> and the name is written using the output embedding tag <%= %> .

Output functions like print or puts won't work with ERB templates. So this would be wrong:

```
<!-- WRONG -->
Hi, Mr. <% puts "Frodo" %>
```

It's important to note that the ERB is simply helping construct HTML server-side. When the view is finished rendering, it will be pure HTML and it is the pure HTML when it is sent out to the user. Your user will never know you are using ERB.

Commenting out ERB

Say you want to comment out some broken Ruby code in your ERB file that's throwing an error:

```
<!-- <%= my_broken_ruby_code %> -->
```

Even though you wrap the embedded Ruby in an HTML comment, the Ruby code will still be evaluated (and inserted as the body of the comment). If the code was erroring out previously, it will still be run and will still cause errors.

To stop the code from running, simply add a # . So:

```
<%#= my_broken_ruby_code %>
```

The %# means to not evaluate the embedded Ruby. The '=' is dangling.

Power User Atom Shortcuts

In a .html.erb file, type = followed by Tab to get <%= %> . Type - followed by Tab to get <% >> .

Instance variables

Controllers make data available to the view layer by setting instance variables. It may seem a bit silly that this is the mechanism by which data is shared since instance variables are all about keeping private data, but that's how Rails does it. When the view is rendered, it copies over the instance variables of the controller so that the view has access to the variables; the view cannot otherwise get access to the controller or its methods.

Let's give a full example:

```
# app/controllers/products_controller.rb
class ProductsController < ActionController::Base
  def index
    # get an array of all products, make it available to view
    @products = Product.all
    render :index
  end
end</pre>
```

It is good practice to make all your database queries inside the controller, setting the results to instance variables. Never make database queries in your views; it can make it harder to find hidden performance issues caused by unintended queries.

link_to and button_to

To help us generate HTML, we may use helper methods. We'll talk about writing our own helpers in a later chapter, but first we'll talk about the two most commonly used helper methods provided by Rails.

You may have seen link_to around before; it generates the HTML code for a link. Here's a few uses:

```
<%= link_to "Cat Pictures", "http://cashcats.biz" %>
<a href="http://cashcats.biz">Cat Pictures</a> <!-- output -->

<%= link_to "New Comment", new_comment_url %>
<a href="c%= new_comment_url %>">New Comment</a> <!-- equivalent to the above code -->
<a href="www.example.com/comments/new">New Comment</a> <!-- output -->
```

When a user clicks on an anchor tag, a GET request is issued. If you want to issue a POST, PATCH, or DELETE request, you can use a button and specify the method:

Technically you can specify the :method attribute for link_to, but you should reserve a tags for GET requests. We'll talk about why later. If you want to issue a POST / PATCH / DELETE, use a button.

Do not rely on these helper methods blindly. Always look at the HTML they generate. You should be able to write the same HTML yourself, if necessary. You'll have to when we start doing JS and React.

Often you want to send some parameters along with the request; for instance, you want to make a POST request to create a new Comment, passing in the comment's text, submitter name, etc. To do this, we want to create an HTML form; we'll learn how to do this in a later chapter.

When it suffices to send parameters in the query string, recall from the routing chapter that you can do this like so: xyz_url(key: :value) .

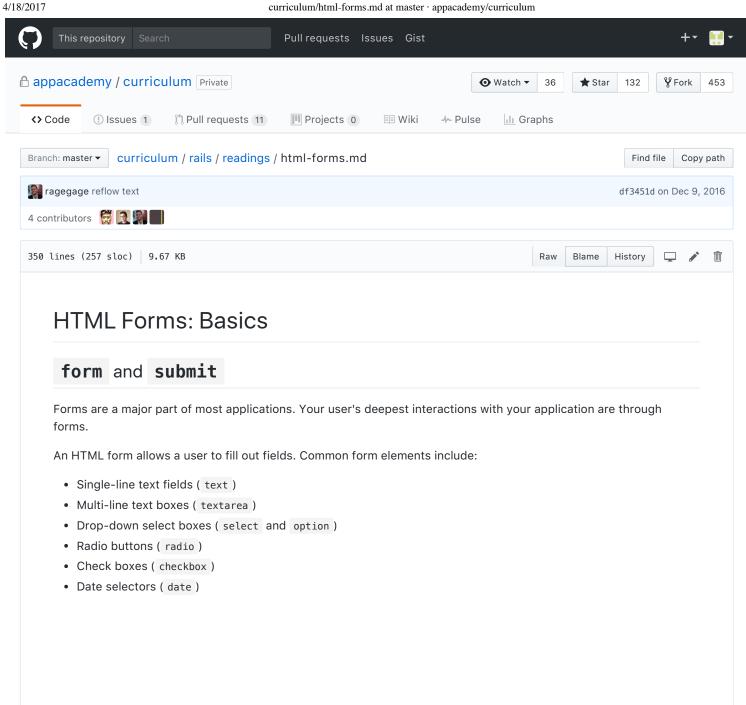
See the URLHelper docs for more info on link to and button to.

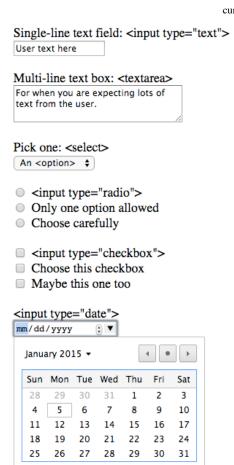
Resources

• URLHelper docs

© 2017 GitHub, Inc. Terms Privacy Security Status Help







A form's input tags are wrapped in a form tag. The form should present a button (an input tag with type=submit). When the user clicks the submit button, the browser will package up the input data, put it in the body of a request, and send it to the server:

```
<form action="http://99cats.com/cats" method="post">
    <!-- input elements -->
    <input type="submit" value="Create cat">
    </form>
```

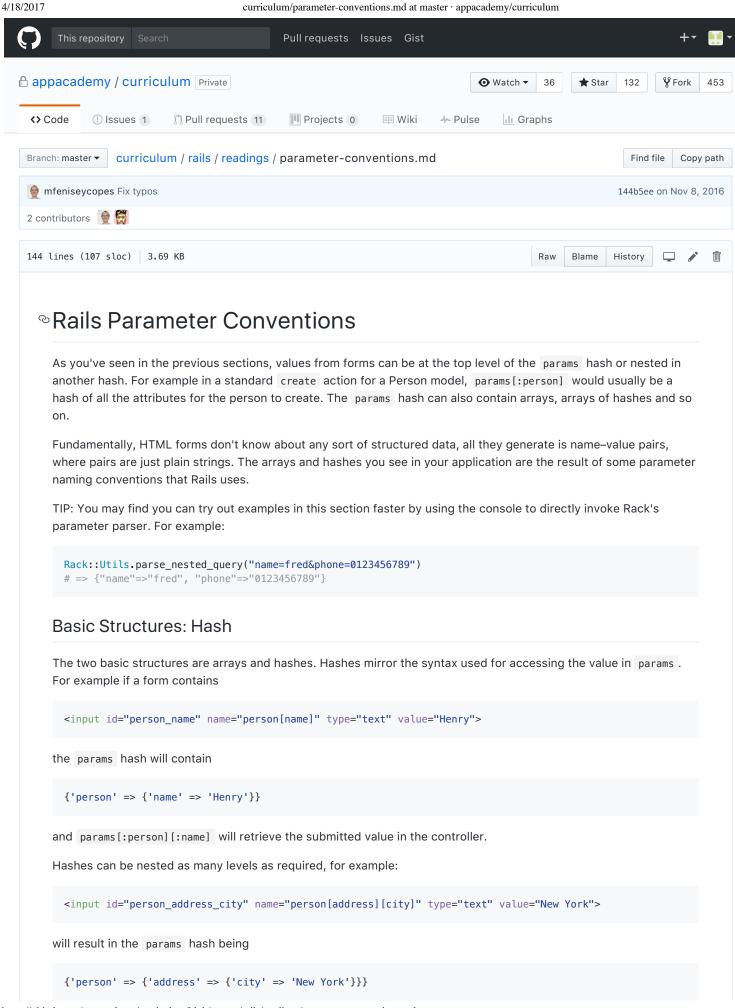
The form tag's action attribute contains the URL to which the form data should be sent, the method attribute describes the HTTP method that should be used. A typical form will POST the data; GET forms are typically only used for search forms, which will merely request data to be fetched.

text input type

The most common type of input is a single-line text field. A text input tag presents a single line of text input for the user to submit:

```
<form action="http://99cats.com/cats" method="post">
    <input type="text" name="cat[first_name]">
    <input type="text" name="cat[last_name]">
    <input type="submit" value="Create cat">
    </form>
```

HTML's form and Rails' params



Basic Structures: Arrays

Normally Rails ignores duplicate parameter names. For instance, if your form contains:

```
<input type="hidden" name="key" value="value1">
<input type="hidden" name="key" value="value2">
```

then on submission, the browser will send:

```
key=value1&key=value2
```

When Rails goes to parse this, it will return:

```
{ "key" => "value2" }
```

The last value wins. Rails overwrites any prior values. Note that Rails is the one which does this: the browser is happy to upload multiple values for the same name.

When you want Rails to build an array, you append an empty set of square brackets to the name:

```
<input name="person[phone_numbers][]" type="text" value="555-123-4567">
<input name="person[phone_numbers][]" type="text" value="555-765-4321">
<input name="person[phone_numbers][]" type="text" value="555-135-2468">
```

Rails will parse the uploaded params as:

```
{ "person" => {
    "phone_numbers" => [
        "555-123-4567",
        "555-765-4321",
        "555-135-2468"
    ]
}
```

Rule 2.5: No arrays of hashes

You can't create arrays of hashes:

```
<input name="persons[][phone_number]" type="text" value="555-123-4567">
<input name="persons[][phone_number]" type="text" value="555-765-4321">
<input name="persons[][phone_number]" type="text" value="555-135-2468">
```

You want:

But this won't work. For whatever reason, Rails won't let you do this. Instead, there's a hack:

```
<input name="persons[0][phone_number]" type="text" value="555-123-4567">
<input name="persons[1][phone_number]" type="text" value="555-765-4321">
```

```
<input name="persons[2][phone_number]" type="text" value="555-135-2468">
```

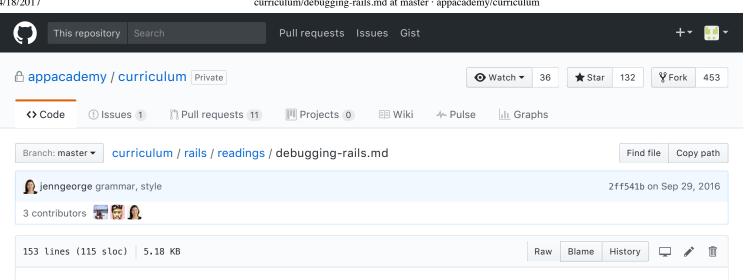
Which Rails translates as:

```
{ "persons" => {
    "0" => { "phone_number" => "555-123-4567" },
    "1" => { "phone_number" => "555-765-4321" },
    "2" => { "phone_number" => "555-135-2468" }
}
```

When we read about nested forms (forms which create more than one object), we will have occasion to use this trick.

© 2017 GitHub, Inc. Terms Privacy Security Status Help





[∞] Debugging Rails

Congratulations, you can write a Rails application! Now it's important to learn how to debug Rails applications well.

Before we start

Bugs happen. Don't be afraid of trying something because you are scared it might not work. Part of being a programmer is givng your crazy ideas a shot to see if they work. That is why you have Git at your disposal; remember to use Git as a safety net that WILL save you if you ever mess up really badly.

Google is your friend

When it comes to Rails, chances are that someone has asked the exact same question as you before. StackOverflow will generally have a question related to yours on it. Be sure to get into the habit of googling when something goes wrong or you don't know something. Your TA is always happy to help, but when you leave AppAcademy, your TA won't be there for you. Part of being a good developer is knowing how to find the answers yourself.

Byebug

Models are tucked away from you and aren't openly exposed through a UI like views and controllers. This may make it a little more difficult at times to work out if something is going wrong in your model. However, this is where error messages will become your friend. Look out for things that relate to your models such as:

- Unknown attribute XYZ
- Undefined method ABC for XYZ

Models behave fairly similarly to classes that you would have written in the first few weeks of App Academy. Why? Because they are classes themselves, just with extra Rails functionality built into them.

Therefore to debug them, we make use of our good friend: byebug.

Add gem 'byebug' to your Gemfile and bundle install.

If you need a refresher on byebug and debugging, be sure to have a look over the previous reading.

You can also type

(byebug) help

inside of byebug to get a list of commands that are available for use.

Better Errors

NB: The latest version of better_errors requires ruby 2+. If your ruby version is not up to snuff upgrade with these instructions.

So now you can debug your models like a pro, but that's only 1/3 of the battle. Being able to debug your controllers is a crucial skill and one that you should make sure that you are comfortable with.

It's important that you can realize when something is going wrong in your controller, for example maybe an object isn't being created properly from your params or you aren't being sent the right params in the first place.

In order to debug our controllers, we are going to start using these two gems:

```
# Gemfile
group :development do
    gem 'better_errors'
    gem 'binding_of_caller'
end

# It's important that these go in a development group. If you have
# these available in production mode, then when you launch your site,
# if an error occurs, users will have access to your code and be able
# to do things like User.destroy_all
```

Better Errors will make your error pages a lot nicer to read, you will be able to get stack traces as well as see useful information such as params. (Note that Better Errors causes issues with RSpec, and you should probably not use it during assessments.)

binding_of_caller gives you an interactive REPL inside the better_errors page. It's like a cross between the debugger and pry, except you can use it in the browser. This is very useful for inspecting values that you have assigned and testing your code as it runs.

Using better_errors and binding_of_caller is required. Always set this up. :-)

Using Better Errors

Better errors is going to open up whenever our code throws an exception of some sort. What happens if our code isn't throwing an error, but doesn't work the way that we want it to?

Well luckily our controllers have a method called fail. By typing fail in a controller, it basically pauses your code at that point and will open up better errors. EG:

```
# app/controllers/posts_controller.rb

class PostsController < ApplicationController
  def create
    @post = Post.new(params[:id])
    fail

  if @post.save
    redirect_to @post
  else
    flash[:errors] ||= []
    flash[:errors] << @post.errors.full_messages.to_sentence
    render :new
  end
end</pre>
```

With this fail, our code will stop just after <code>@post = Post.new(params[:id])</code>, we can now do things like <code>p @post and</code> try to save it manually to see what's going wrong with our code.

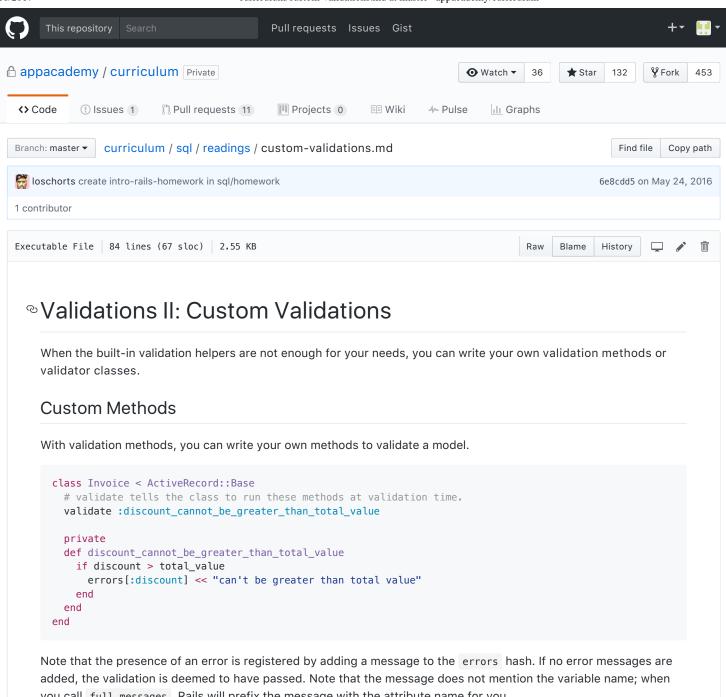
Using fail is very useful and is something you should do often. It will allow you to step into your controller as it runs and make sure things like params are coming in correctly.

Useful things to do inside of a controller when debugging

- · Check what the params are
- Try @object.save . If it returns false then call @object.valid? And check @object.errors.full_messages . This will allow you to see what validations are failing.
- Make sure things like current_user are working.
- Make sure instance variables are set correctly.
- Check that you have called permit on the params when building objects.
- Check that objects being built via associations are built correctly. EG: current_user.posts.new(post_params)

© 2017 GitHub, Inc. Terms Privacy Security Status Help





you call full_messages, Rails will prefix the message with the attribute name for you.

errors[:base]

Sometimes an error is not specific to any one attribute. In this case, you add the error to errors [:base] . Since errors[:base] is an array, you can simply add a string to it and it will be used as an error message.

```
class Person < ActiveRecord::Base</pre>
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid because ..."</pre>
  end
end
```

Custom Validators

Custom validators are classes that extend ActiveModel::EachValidator . Prefer writing a custom validator class when you want to reuse validation logic for multiple models or multiple columns. Otherwise, it's simpler to use a validator method.

Your custom validator class must implement a validate_each method which takes three arguments: the record, the attribute name and its value.

```
# app/validators/email_validator.rb
class EmailValidator < ActiveModel::EachValidator</pre>
  CRAZY\_EMAIL\_REGEX = /A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
  def validate_each(record, attribute_name, value)
    unless value =~ CRAZY EMAIL REGEX
      # we can use `EachValidator#options` to access custom options
      # passed to the validator.
      message = options[:message] || "is not an email"
      record.errors[attribute_name] << message</pre>
  end
end
# app/models/person.rb
class Person < ActiveRecord::Base</pre>
  # Rails knows `:email` means `EmailValidator`.
 validates :email, :presence => true, :email => true
 # not required, but must also be an email
  validates :backup_email, :email => {:message => "isn't even valid"}
```

As shown in the example, you can also combine standard validations with your own custom validators.

© 2017 GitHub, Inc. Terms Privacy Security Status Help

