Phase 4 Notes

\*\*\*===### type 3 or more of any of these symbols and press Enter to create a horizontal line.

Possible project set up https://github.com/varteaga5/phase-4-authenticating-users-lab

Let's set up a controller:

rails g controller Birds --no-test-framework

Entering rails g model Test name:text in your command line will generate the following:

(1) A model file test.rb in your models directory:

class Test < ActiveRecord::Base

end

(2) A migration file timestamp\_create\_tests.rb in your db/migrate directory:

class CreateTests < ActiveRecord::Migration

def change

create\_table :tests do |t|

t.text :name

t.timestamps

end

end

end

###### **Generating Resources**

Entering rails g resource Test name:text in your command line will generate the following:

(1) A model file test.rb in your models directory:

class Test < ActiveRecord::Base

end

(2) A migration file timestamp\_create\_tests.rb  in your db/migrate directory:

class CreateTests < ActiveRecord::Migration

def change

create\_table :tests do |t|

t.text :name

t.timestamps

end

end

end

(3) a tests\_controller.rb file in your controllers directory.  This controller will be an empty shell:

class TestsController < ApplicationController

end

(4) resources :tests routes in your routes.rb file.

###### **Generating Scaffolding**

Entering rails g scaffold Test name:text in your command line will generate the following:

(1) A model file test.rb in your models directory:

class Test < ActiveRecord::Base

end

(2) A migration file timestamp\_create\_tests.rb in your db/migrate directory:

class CreateTests < ActiveRecord::Migration

def change

create\_table :tests do |t|

t.text :name

t.timestamps

end

end

end

(3) A tests\_controller.rb file in your controllers directory.  When a scaffold is generated, seven public methods and two private methods will be added to your controller:

class TestsController < ApplicationController

before\_action :set\_test, only: [:show, :edit, :update, :destroy]

# GET /tests

# GET /tests.json

def index

@tests = Test.all

end

# GET /tests/1

# GET /tests/1.json

def show

end

# GET /tests/new

def new

@test = Test.new

end

# GET /tests/1/edit

def edit

end

# POST /tests

# POST /tests.json

def create

@test = Test.new(test\_params)

respond\_to do |format|

if @test.save

format.html { redirect\_to @test, notice: 'Test was successfully created.' }

format.json { render action: 'show', status: :created, location: @test }

else

format.html { render action: 'new' }

format.json { render json: @test.errors, status: :unprocessable\_entity }

end

end

end

# PATCH/PUT /tests/1

# PATCH/PUT /tests/1.json

def update

respond\_to do |format|

if @test.update(test\_params)

format.html { redirect\_to @test, notice: 'Test was successfully updated.' }

format.json { head :no\_content }

else

format.html { render action: 'edit' }

format.json { render json: @test.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /tests/1

# DELETE /tests/1.json

def destroy

@test.destroy

respond\_to do |format|

format.html { redirect\_to tests\_url }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_test

@test = Test.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def test\_params

params.require(:test).permit(:name)

end

end

(4) resources :tests routes in your routes.rb file.

(5) Seven corresponding view files in your views directory: (a) \_form.html.erb, (b) edit.html.erb, (c) index.html.erb, (d) index.json.jbuilder, (e) new.html.erb, (f) show.html.erb and (g) show.json.jbuilder. Each view will contain html and embedded ruby.

9.16.21

**REST** (**RE**presentational **S**tate **T**ransfer)

**Definition of HTTP Verbs**

* **GET**: The GET method retrieves whatever information is identified by the Request URI. This means if you go to /posts, you will get all of the posts that the application has.
* **POST**: The POST method is used to send data enclosed in the request to the server. The server is expected to use this data to create some new resource.
* **PATCH/PUT**: The PATCH and PUT methods are both used to update existing resources. Sending either a PATCH or PUT request to /posts/1 will update the post with an id of 1. PUT is used when we want to replace an entire resource. PATCH is used when we want to update a specific part of a resource. Check out this explanation of the [difference between the two (Links to an external site.)](https://blog.fullstacktraining.com/restful-api-design-post-vs-put-vs-patch/).
* **DELETE**: The DELETE method requests that the server delete the resource identified by the Request URI. This means… that it deletes the record. It's nice and explicit.

This Command will create a model named Bird:

rails g model Bird name species --no-test-framework

Standard routes.rb content

Rails.application.routes.draw do

get '/birds' , to: 'birds#index'

get '/birds/:id' , to: 'birds#show'

end

this above code can be changed to:

Rails.application.routes.draw do

resources :birds

end

Then, run rails routes again:

Prefix Verb URI Pattern Controller#Action  
 birds GET /birds(.:format) birds#index  
 POST /birds(.:format) birds#create  
 bird GET /birds/:id(.:format) birds#show  
 PATCH /birds/:id(.:format) birds#update  
 PUT /birds/:id(.:format) birds#update  
 DELETE /birds/:id(.:format) birds#destroy

With just one line of code — resources :birds — Rails created all the RESTful routes we need and mapped them to the appropriate controller action!

With great power comes great responsibility. Even though we'll eventually add all of these RESTful routes to our API, for the time being, we only need two: the index and show routes. We can still use resources and customize which routes are created like so:

Rails.application.routes.draw do

resources :birds, only: [:index, :show]

end

Running rails routes now will give us the same output as when we wrote out the routes by hand:

Prefix Verb URI Pattern Controller#Action  
 birds GET /birds(.:format) birds#index  
 GET /birds/:id(.:format) birds#show

As a rule, **you should only generate routes that your API is actually using**. If you create a route using resources, but don't implement the controller action for that route, your API's consumers (any clients using your API) will get an unexpected response if they try to use a route that doesn't exist.

To complete our first couple RESTful actions, let's set up a controller:

rails g controller Birds --no-test-framework

9.17.21

# Testing APIs with Postman

<https://learning.flatironschool.com/courses/2648/pages/testing-apis-with-postman?module_item_id=265131>

# **Rails Resource Routing: Create**

Following REST conventions, we'll want our clients to make a POST request to /birds to create a new bird. Using the resources method, we can create this route by adding in create to the list of actions we want handled:

Rails.application.routes.draw do  
 resources :birds, only: [:index, :show, :create]  
end

After updating our routes, run rails routes to check what routes are now available:

Prefix Verb URI Pattern Controller#Action  
 birds GET /birds(.:format) birds#index  
 POST /birds(.:format) birds#create  
 bird GET /birds/:id(.:format) birds#show

We've seen params once before, as a way to access the dynamic part of the URL:

# GET /birds/:id  
def show  
 # params[:id] refers to the dynamic part of our route, defined by :id  
 # a request to /birds/2 would give params[:id] a value of 2  
 bird = Bird.find\_by(id: params[:id])  
 render json: bird  
end

Here, we can also see all the data from the body of our request added to this params hash as well! Let's use that to create our bird. Exit the byebug session by typing continue or c and hit enter. Then, update your controller action like so:

def create  
 bird = Bird.create(name: params[:name], species: params[:species])  
 render json: bird, status: :created  
end

9.20.21

Pod meeting starting a project to complete during each pod meeting left on leaving the last commit of: generate resource user define show method

https://github.com/varteaga5/ground-bnb-backend

9.20.21

# **Strong Params and Mass Assignment**

Since Active Record models also use this feature of mass assignment to take a hash of any key-value pairs and assign them to attributes on our models, passing in the entire params hash when creating a new record in our database opens us up to this [mass assignment vulnerability (Links to an external site.)](https://en.wikipedia.org/wiki/Mass_assignment_vulnerability).

So how do we fix it?

## Strong Params

Thanks to Rails' built-in security protection against the [mass assignment vulnerability (Links to an external site.)](https://en.wikipedia.org/wiki/Mass_assignment_vulnerability). We can't just pass in the entire params hash, since that would mean a malicious user could potentially update attributes of our model that we don't want to give them access to.

What we can do instead is use [Strong Parameters (Links to an external site.)](https://guides.rubyonrails.org/action_controller_overview.html#strong-parameters) to **whitelist** only the parameters that we want to use:

# POST /birds

def create

bird = Bird.create(params.permit(:name, :species))

render json: bird, status: :created

end

after a refactor it will look like this🡺

def create

bird = Bird.create(bird\_params)

render json: bird, status: :created

end

private

# all methods below here are private

def bird\_params

params.permit(:name, :species)

end

You may also have noticed that even though the request body only has this data:

{  
 "name": "Blue Jay",  
 "species": "Cyanocitta cristata"  
}

Our params hash looks like this:

{  
 "name"=>"Blue Jay",  
 "species"=>"Cyanocitta cristata",  
 "bird"=>{  
 "name"=>"Blue Jay",  
 "species"=>"Cyanocitta cristata"  
 }  
}

The reason for this is that Rails by default will [wrap JSON parameters (Links to an external site.)](https://edgeguides.rubyonrails.org/action_controller_overview.html#json-parameters) as a nested hash under a key based on the name of the controller (in our case, bird since we're in a BirdsController). This is the reason that in the Rails server log, even with our strong params in place, you'll still see Unpermitted parameters: :bird for our requests.

You can disable the wrap parameters feature in an individual controller:

class BirdsController < ApplicationController  
 wrap\_parameters format: []  
end

You can also disable it for all controllers if you like, by going into the config/initializers/wrap\_parameters.rb file and updating it like so:

ActiveSupport.on\_load(:action\_controller) do  
 wrap\_parameters format: []  
end

There are two ways we can work with params in Rails to keep our data safe.

* Explicitly specify the attributes we are trying to assign:

Bird.create(name: params[:name], species: params[:species])

* Or use strong params to whitelist specific attributes:

Bird.create(params.permit(:name, :species))

# **Rails Create, Index, and Show Lab**

The React application is in the client directory. To set it up, from the root directory, run:

npm install --prefix client

Using --prefix client will run the npm command within the client directory.

# **9.21.21**

# **Rails Resource Routing: Update**

## Changing Our Model With Migrations

Let's start by creating a new migration to update our Bird model and the associated birds table:

rails g migration AddLikesToBird likes:integer --no-test-framework

For a refresher on migrations, check out the [Active Record docs (Links to an external site.)](https://guides.rubyonrails.org/active_record_migrations.html)!

We can use resources to add this route by adding the :update action in our routes.rb file:

resources :birds, only: [:index, :show, :create, :update]

breaks RESTful conventions and create a custom route:

We could take some of that burden off of the frontend by providing a **custom route** that will do the work of calculating the number of likes and incrementing it, so that all the frontend has to do is send a request to our new custom route, without worrying about sending any data in the body of the request.

Update the routes.rb file like so:

Rails.application.routes.draw do  
 resources :birds, only: [:index, :show, :create, :update]  
 patch "/birds/:id/like", to: "birds#increment\_likes"  
end

Then create the increment\_likes controller action:

def increment\_likes  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 bird.update(likes: bird.likes + 1)  
 render json: bird  
 else  
 render json: { error: "Bird not found" }, status: :not\_found  
 end  
end

Notice that in this action, the only information we need from params is the id; we're able to use the bird's current number of likes to calculate the next number of likes! Our client app no longer needs to concern itself with sending that data or performing that calculation.

A note on breaking convention: by creating this custom route, we are breaking the REST conventions we had been following up to this point. One alternate way to structure this kind of feature and keep our routes and controllers RESTful would be to create a new controller, such as Birds::LikesController, and add a create action in this controller. The creator of Rails, DHH, advocates for [this approach for managing sub-resources (Links to an external site.)](http://jeromedalbert.com/how-dhh-organizes-his-rails-controllers/).

# **Rails Resource Routing: Destroy**

ndle a DELETE /birds/:id request. We can do so by adding :destroy to our resources:

resources :birds, only: [:index, :show, :create, :update, :destroy]

And since we're now using all five RESTful routes, we can omit the only option:

resources :birds

Running rails routes will show us all the RESTful routes in our application, plus our custom route:

Prefix Verb URI Pattern Controller#Action  
 birds GET /birds(.:format) birds#index  
 POST /birds(.:format) birds#create  
 bird GET /birds/:id(.:format) birds#show  
 PATCH /birds/:id(.:format) birds#update  
 PUT /birds/:id(.:format) birds#update  
 DELETE /birds/:id(.:format) birds#destroy  
 PATCH /birds/:id/like(.:format) birds#increment\_likes

We'll also need to add a destroy action to our controller where we'll be deleting the bird from the database:

def destroy  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 bird.destroy  
 head :no\_content  
 else  
 render json: { error: "Bird not found" }, status: :not\_found  
 end  
end

In this controller action, our goal is to:

* Find a bird using the ID from the route params
* Remove it from the database with bird.destroy

You'll also notice that instead of rendering a JSON response, we're returning head :no\_content if our bird was successfully deleted. :no\_content will give a 204 status code, indicating that the server has successfully fulfilled the request and that there is no content to send in the response. We're also not sending any payload of data in the body of the request.

# **Controller Exception Handling**

## DRYing Up Controller Code

# GET /birds/:id  
def show  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 render json: bird  
 else  
 render json: { error: "Bird not found" }, status: :not\_found  
 end  
end  
  
# PATCH /birds/:id  
def update  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 bird.update(bird\_params)  
 render json: bird  
 else  
 render json: { error: "Bird not found" }, status: :not\_found  
 end  
end

Between these two methods, there's a good amount of repeated code:

* Finding a bird based on the ID
* Performing control flow (if/else) based on whether or not the bird exists
* Returning an error message with a status of :not\_found if the bird doesn't exist

Let's start by making a private method for generating the :not\_found response:

private  
  
def render\_not\_found\_response  
 render json: { error: "Bird not found" }, status: :not\_found  
end

We can then update our actions to use this method instead of implementing the rendering logic directly:

# GET /birds/:id  
def show  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 render json: bird  
 else  
 render\_not\_found\_response  
 end  
end  
  
# PATCH /birds/:id  
def update  
 bird = Bird.find\_by(id: params[:id])  
 if bird  
 bird.update(bird\_params)  
 render json: bird  
 else  
 render\_not\_found\_response  
 end  
end

We can also make a helper method to find a bird based on the ID in the params hash:

private  
  
def find\_bird  
 Bird.find\_by(id: params[:id])  
end

Now, our controller actions don't need to worry about how the find\_bird method is implemented, as long as it returns a bird from the database. This frees us up to change how the bird finding logic is implemented in the future (for example, using something other than the ID to look up a bird in the database, like a URL slug or [UUID (Links to an external site.)](https://en.wikipedia.org/wiki/Universally_unique_identifier)).

Here's how our controller actions can use this method:

# GET /birds/:id  
def show  
 bird = find\_bird  
 if bird  
 render json: bird  
 else  
 render\_not\_found\_response  
 end  
end  
  
# PATCH /birds/:id  
def update  
 bird = find\_bird  
 if bird  
 bird.update(bird\_params)  
 render json: bird  
 else  
 render\_not\_found\_response  
 end  
end

Here's the fully refactored version of the controller:

class BirdsController < ApplicationController  
 rescue\_from ActiveRecord::RecordNotFound, with: :render\_not\_found\_response  
  
 # GET /birds  
 def index  
 birds = Bird.all  
 render json: birds  
 end  
  
 # POST /birds  
 def create  
 bird = Bird.create(bird\_params)  
 render json: bird, status: :created  
 end  
  
 # GET /birds/:id  
 def show  
 bird = find\_bird  
 render json: bird  
 end  
  
 # PATCH /birds/:id  
 def update  
 bird = find\_bird  
 bird.update(bird\_params)  
 render json: bird  
 end  
  
 # PATCH /birds/:id/like  
 def increment\_likes  
 bird = find\_bird  
 bird.update(likes: bird.likes + 1)  
 render json: bird  
 end  
  
 # DELETE /birds/:id  
 def destroy  
 bird = find\_bird  
 bird.destroy  
 head :no\_content  
 end  
  
 private  
  
 def find\_bird  
 Bird.find(params[:id])  
 end  
  
 def bird\_params  
 params.permit(:name, :species, :likes)  
 end  
  
 def render\_not\_found\_response  
 render json: { error: "Bird not found" }, status: :not\_found  
 end  
  
end

## Conclusion

Using exception handling techniques like rescue and rescue\_from opens up a lot of possibilities in terms of how you structure your code. For our controller actions in particular, it allows us to isolate the "happy path" of our code (performing CRUD actions and rendering a response to the users) from the exception handling logic. It also lets us handle exceptions in a consistent way, so that users of our API get the same response for common errors, like not being able to find a particular resource.

9.21.21

**Active Record Validations**

For more examples of basic validation usage, see the Rails Guide for [Active Record Validations (Links to an external site.)](http://guides.rubyonrails.org/active_record_validations.html). Take a few minutes to browse the helpers listed in Section 2.

**Database activity triggers validation**. An Active Record model instantiated with #new will not be validated, because no attempt to write to the database was made. Validations won't run unless you call a method that actually hits the DB, like #save.

To return an **array** of pre-formatted error messages, use the .errors.full\_messages method as shown

We can also get an array of nicely-formatted errors with the .full\_messages method:

def create  
 person = Person.create(person\_params)  
 if person.valid?  
 render json: person, status: :created  
 else  
 render json: { errors: person.errors.full\_messages }, status: :unprocessable\_entity  
 end  
end

Rails has a host of built-in helpers.

### Length

length is one of the most versatile:

class Person < Active Record::Base  
 validates :name, length: { minimum: 2 }  
 validates :bio, length: { maximum: 500 }  
 validates :password, length: { in: 6..20 }  
 validates :registration\_number, length: { is: 6 }  
end

The in argument makes use of a [Range (Links to an external site.)](http://ruby-doc.org/core/Range.html).

### Uniqueness

Another common built-in validator is uniqueness:

class Account < Active Record::Base  
 validates :email, uniqueness: true  
end

### Custom Validations

There are two ways to implement custom validations, with examples in [Section 6 (Links to an external site.)](http://guides.rubyonrails.org/active_record_validations.html#performing-custom-validations) of the Rails Guide.

Of the two, adding custom methods is the simplest. Here's an example of creating a custom validation method to check the validity of an email address:

class Person  
 validate :must\_have\_flatiron\_email  
  
 def must\_have\_flatiron\_email  
 unless email.match?(/flatironschool.com/)  
 errors.add(:email, "We're only allowed to have people who work for the company in the database!")  
 end  
 end  
end

Note that here, we are calling the #validate method, rather than #validates, and passing it a method we write ourselves to perform our custom validation.

Post category is either Fiction or Non-Fiction. This step requires an inclusion validator, which was not outlined in the lesson. You'll need to refer to the [Rails guide (Links to an external site.)](https://guides.rubyonrails.org/active_record_validations.html) to look up how to use it.

This helper validates that the attributes' values are included in a given set. In fact, this set can be any enumerable object.

class **Coffee** **<** ApplicationRecord

validates :size, inclusion: { in: %w(small medium large),

message: "%{value} is not a valid size" }

end

9.22.21

In the rescue block, the invalid variable is an instance of the exception itself. From that invalid variable, we can access the actual Active Record instance with the record method, where we can retrieve its errors.

We can take a similar approach to validation in our update method, since validations will also run when a model is updated:

def update  
 bird = find\_bird  
 bird.update!(bird\_params)  
 render json: bird  
rescue ActiveRecord::RecordInvalid => invalid  
 render json: { errors: invalid.record.errors }, status: :unprocessable\_entity  
end

We could also handle **all** ActiveRecord::RecordInvalid exceptions in the controller with the rescue\_from method:

class BirdsController < ApplicationController  
 rescue\_from ActiveRecord::RecordNotFound, with: :render\_not\_found\_response  
 # added rescue\_from  
 rescue\_from ActiveRecord::RecordInvalid, with: :render\_unprocessable\_entity\_response  
  
 # rest of controller actions...  
  
 private  
  
 def render\_unprocessable\_entity\_response(invalid)  
 render json: { errors: invalid.record.errors }, status: :unprocessable\_entity  
 end  
  
 # rest of private methods...  
end

Now, our create and update actions can focus on the happy path:

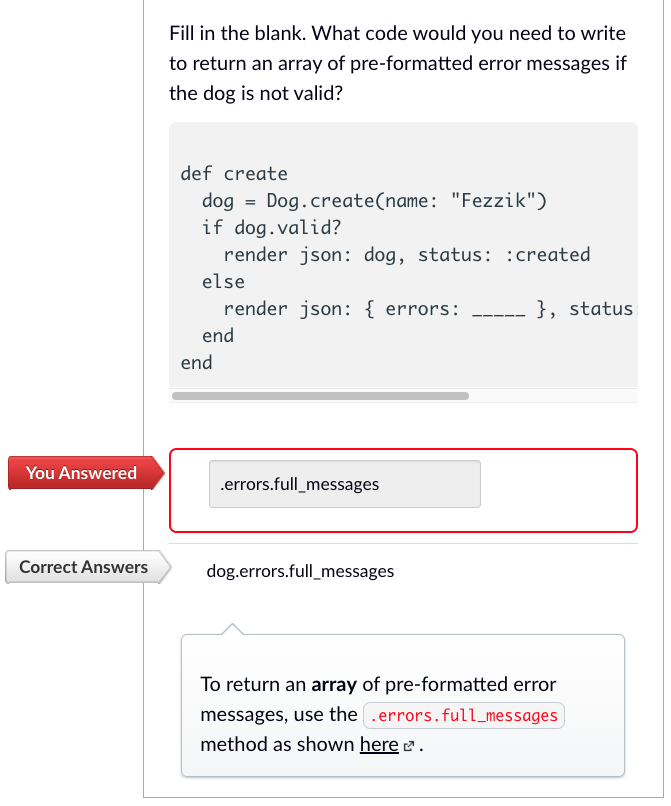
def create  
 # create! exceptions will be handled by the rescue\_from ActiveRecord::RecordInvalid code  
 bird = Bird.create!(bird\_params)  
 render json: bird, status: :created  
end  
  
def update  
 bird = find\_bird  
 # update! exceptions will be handled by the rescue\_from ActiveRecord::RecordInvalid code  
 bird.update!(bird\_params)  
 render json: bird  
end

9.23.21

# Validations Quiz

The following methods trigger validations, and will save the object to the database only if the object is valid:

* create
* create!
* save
* save!
* update
* update!



# **Creating a Rails API from Scratch**

## Generating a Rails API

Just like we saw at the beginning of the phase, we can use rails new to generate a new Rails application. We'll run that same command with a few additional options to optimize our Rails app. Let's generate the backend code for our dvd-shop. Use cd .. to navigate out of the lab directory, and run:

rails new dvd-shop --api --minimal

* --api: this flag will create our new application with some additional API-specific configuration, and will skip the code for generating .erb files with ActionView.
* --minimal: this flag skips a lot of additional Rails features that we won't use in our API, such as code for sending emails and processing images. Read more about the [--minimal flag (Links to an external site.)](https://bigbinary.com/blog/rails-6-1-adds-minimal-option-support).

The reason we ask you to cd out of the lab directory is because when you generate a new Rails project, it will automatically create a new Git repository for your Rails project. Since the lab directory is already a Git repository, it's better to create this new project in its own directory, so you don't end up with nested Git repositories.

**Thursday class meeting 9.23.21**

**Question for Thursday class meeting: when using rescue\_from in the controller, we have to use the ! on our create or update?**

**Go over lab with custom validations**

**Is there a way to do rails g scaffold without the front end stuff, including a flag**

Notes for Thursday class:

Validates :name, :age, :location, presence: true

Move the rescue\_from command to the application\_controller and all the other controllers will have this feature

# **Adding React to Rails**

## Generating a React Application

To get started, let's spin up our React application using create-react-app:

npx create-react-app client --use-npm

This will create a new React application in a client folder, and will use npm instead of yarn to manage our dependencies.

When we're running React and Rails in development, we'll need two separate servers running on different ports — we'll run React on port 4000, and Rails on port 3000. Whenever we want to make a request to our Rails API from React, we'll need to make sure that our requests are going to port 3000.

We can simplify this process of making requests to the correct port by using create-react-app in development to [proxy the requests to our API (Links to an external site.)](https://create-react-app.dev/docs/proxying-api-requests-in-development/). This will let us write our network requests like this:

fetch("/movies");  
// instead of fetch("http://localhost:3000/movies")

To set up this proxy feature, open the package.json file in the client directory and add this line at the top level of the JSON object:

"proxy": "http://localhost:3000"

Let's also update the "start" script in the the package.json file to specify a different port to run our React app in development:

"scripts": {  
 "start": "PORT=4000 react-scripts start"  
}

With that set up, let's try running our servers! In your terminal, run Rails:

rails s

Then, open a new terminal, and run React:

npm start --prefix client

This will run npm start in the client directory.

We can also see how to make a request using fetch. In the React application, update your App.js file with the following code:

import { useEffect } from "react";  
  
function App() {  
 useEffect(() => {  
 fetch("/movies")  
 .then((r) => r.json())  
 .then((movies) => console.log(movies));  
 }, []);  
  
 return <h1>Hello from React!</h1>;  
}  
  
export default App;

This will use the useEffect hook to fetch data from our Rails API, which you can then view in the console.

9.24.21

# **Debugging Network Requests**

But our Console tab tells a different story:

Uncaught (in promise) SyntaxError: Unexpected end of JSON input  
 at MovieForm.js:19

**Pay careful attention to this error!** This is a very common error message to see as you're developing API-driven applications. What do you think this error message indicates?

Well, it's a [SyntaxError (Links to an external site.)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError" \t "_blank), which means it's an error that occurs when JavaScript is parsing our code.

It's coming from our fetch request as part of the promise chain:

fetch("/movies", {  
 method: "POST",  
 headers: {  
 "Content-Type": "application/json",  
 },  
 body: JSON.stringify(formData),  
})  
 .then((r) => r.json())  
 .then((data) => {  
 console.log(data);  
 });

The reason for this specific error is that **we expect the server to return a string of JSON-formatted data**, but the server is not returning any content.

Remember, when we call r.json(), we are instructing JavaScript to read the response (r) and parse it as a JSON-formatted string. If the response does not contain JSON-formatted data, **this code will error out**. We are not sending **any** data in the response:

def create  
end

Keep this in mind when you see this error message! The solution to this problem is to:

* **return JSON data in the response** from your controller actions, or
* **don't use the .json() method on the response** if your controller does not respond with JSON data

**Debugging Params**

Let's continue working on this feature, and see how we can access the data from the fetch request in our controller action. Start by adding a byebug:

def create  
 byebug  
end

Then, enter some data in the form, and submit the form again to make another request. In your server tab, you'll now enter a debugging session:

8: def create  
 9: byebug  
=> 10: end  
 11:  
 12: end  
(byebug)

Use this as an opportunity to inspect the request object, in particular looking at the params hash:

(byebug) params  
#<ActionController::Parameters {"title"=>"", "year"=>2021, "length"=>"0",  
"director"=>"", ...>

You'll see that all the key-value pairs from the formData object — representing the input fields from the form that were sent in the body of the request — are present in the params hash. Type c and hit enter to exit the byebug.

We can use that to create a new Movie, and send the movie object back as a response:

def create  
 movie = Movie.create(movie\_params)  
 render json: movie, status: :created  
end  
  
private  
  
def movie\_params  
 params.permit(:title, :year, :length, :director, :description, :poster\_url, :category, :discount, :female\_director)  
end

With that in place, we can submit the request one more time to finally create a Movie! A good place to look after a successful request is in the Rails logs to see all the details of the request:

## Conclusion

When you see a 404 Not Found error:

* Check the Network Tab on the frontend
* Check the Rails Server logs in the backend
* Add a new route to handle the HTTP Verb + Path for this request

When you see an Unexpected end of JSON input error:

* Check request your fetch
* Check the controller action, and make sure to render json:

When you need to see data from the request in your controller:

* Add a byebug to your controller method
* Make a request from the browser
* Inspect params in the byebug session

When you see a 500 Internal Server Error error:

* Open your Rails server log
* Look for the last request that came through
* Find the error message and stack trace to identify where the error originated

# **Validation Messages and Error Handling**

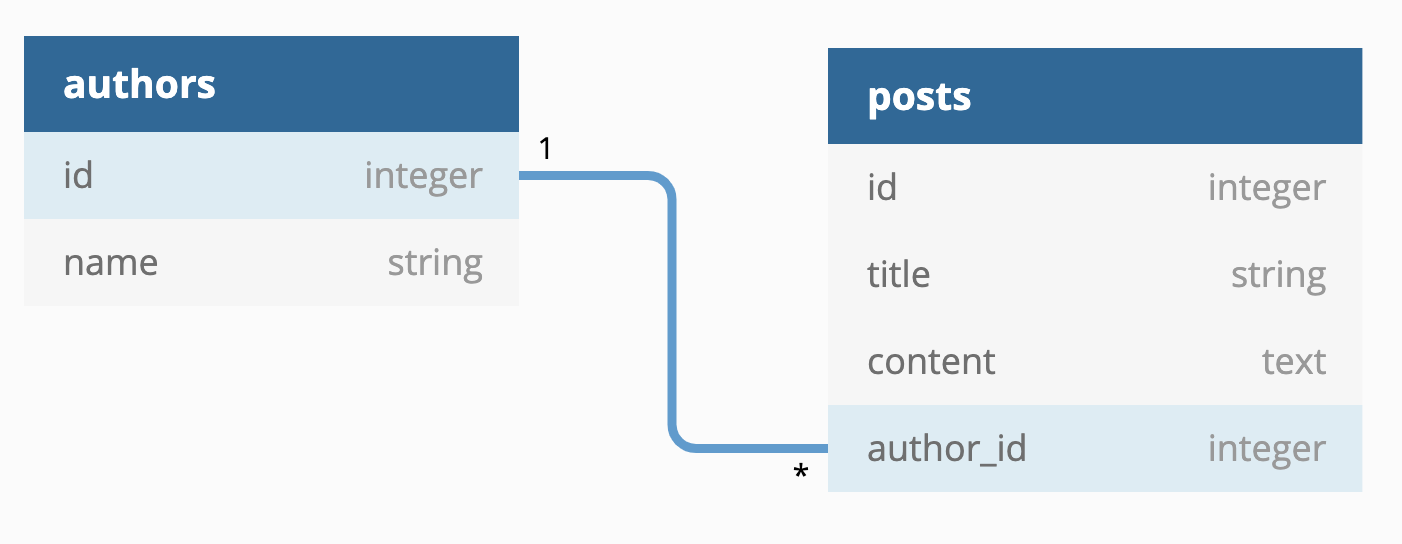
In the MovieForm component's handleSubmit function, modify the fetch request as follows:

We can use the .ok property of the response object to see whether the response has a good status code (200-300 range) or a bad status code (400-500 range) and handle the response accordingly. If the response is not ok, we'll want to display some error messages to the user; that means we'll need to keep track of those error messages in state, and re-render the component when those error messages are updated:

function handleSubmit(e) {  
 e.preventDefault();  
 fetch("/movies", {  
 method: "POST",  
 headers: {  
 "Content-Type": "application/json",  
 },  
 body: JSON.stringify(formData),  
 }).then((response) => {  
 if (response.ok) {  
 response.json().then((newMovie) => console.log(newMovie));  
 } else {  
 response.json().then((errorData) => setErrors(errorData.errors));  
 }  
 });  
}

9.26.21

# **Active Record Associations Review**



The schema for this ERD would be:

create\_table "authors", force: :cascade do |t|  
 t.string "name"  
end  
  
create\_table "posts", force: :cascade do |t|  
 t.string "title"  
 t.text "content"  
 t.integer "author\_id", null: false  
end

Like any other column, foreign keys are accessible through instance methods of the same name. This means you could find a given post's author with the following Active Record query:

Author.find(post.author\_id)

## One-To-Many Relationships

By using Active Record's macro-style association class methods, we can add some convenient instance methods to our models.

The most common relationship is **one-to-many**. Active Record gives us the has\_many and belongs\_to macros for creating instance methods to access data across models in a **one-to-many** relationship.

### belongs\_to

Each Post is associated with **one** Author.

class Post < ApplicationRecord  
 belongs\_to :author  
end

We now have access to some new instance methods, like author. This will return the actual Author object that is attached to that post.

post = Post.first  
post.author #=> #<Author @id=1>

### has\_many

In the opposite direction, each Author might be associated with zero, one, or many Post objects. We haven't changed the schema of the authors table at all; Active Record is just going to use posts.author\_id to do all of the lookups.

class Author < ApplicationRecord  
 has\_many :posts  
end

Now we can look up an author's posts just as easily:

author = Author.last  
author.posts #=> [#<Post @id=3>, #<Post @id=4>]

## Convenience Builders

### Building a new item in a collection

If you want to add a new post for an author, you might start this way:

new\_post = Post.create(author\_id: author.id, title: "Web Development for Cats")

But the association macros save the day again, allowing this instead:

author = Author.first  
new\_post = author.posts.create(title: "Web Development for Cats")  
  
author.posts  
#=> [#<Post @id=1>, #<Post @id=5>]

This will create a new Post object with the author\_id already set for you! We use this one as much as possible because it's just easier.

author.posts.create will create a new instance and persist it to the database. You can also use author.posts.build to generate a new instance without persisting.

### Setting a singular association

The setup process is a little bit less intuitive for singular associations. Remember, a given post belongs\_to an author. The verbose way of creating this association would be like so:

post.author = Author.new(name: "Lasandra Gulgowski")

In the previous section, author.posts always exists, even if it's an empty array. Here, post.author is nil until the author is defined, so using post.author.create would throw an error. Instead, Active Record allows us to prepend the attribute with build\_ or create\_. The create\_ option will persist to the database for you.

post = Post.new(title: "Web Development for Dogs")  
new\_author = post.create\_author(name: "Lasandra Gulgowski")  
post.save  
  
post.author  
#=> #<Author @name="Lasandra Gulgowski">  
new\_author.post  
#=> [#<Post @title="Web Development for Dogs">]

Remember, if you use the build\_ option, you'll need to persist your new author with #save.

### Collection Convenience

If you add an existing object to a collection association, Active Record will conveniently take care of setting the foreign key for you:

author = Author.find\_by(name: "Lasandra Gulgowski")  
author.posts  
#=> [#<Post @title="Web Development for Dogs">]  
  
post = Post.new(title: "Web Development for Cats")  
post.author  
#=> nil

# if you shovel ‘post’ into ‘author.posts’ the post will be assigned the appropriate author, in this case it is Lasandra  
author.posts << post  
post.author  
#=> #<Author @name="Lasandra Gulgowski">

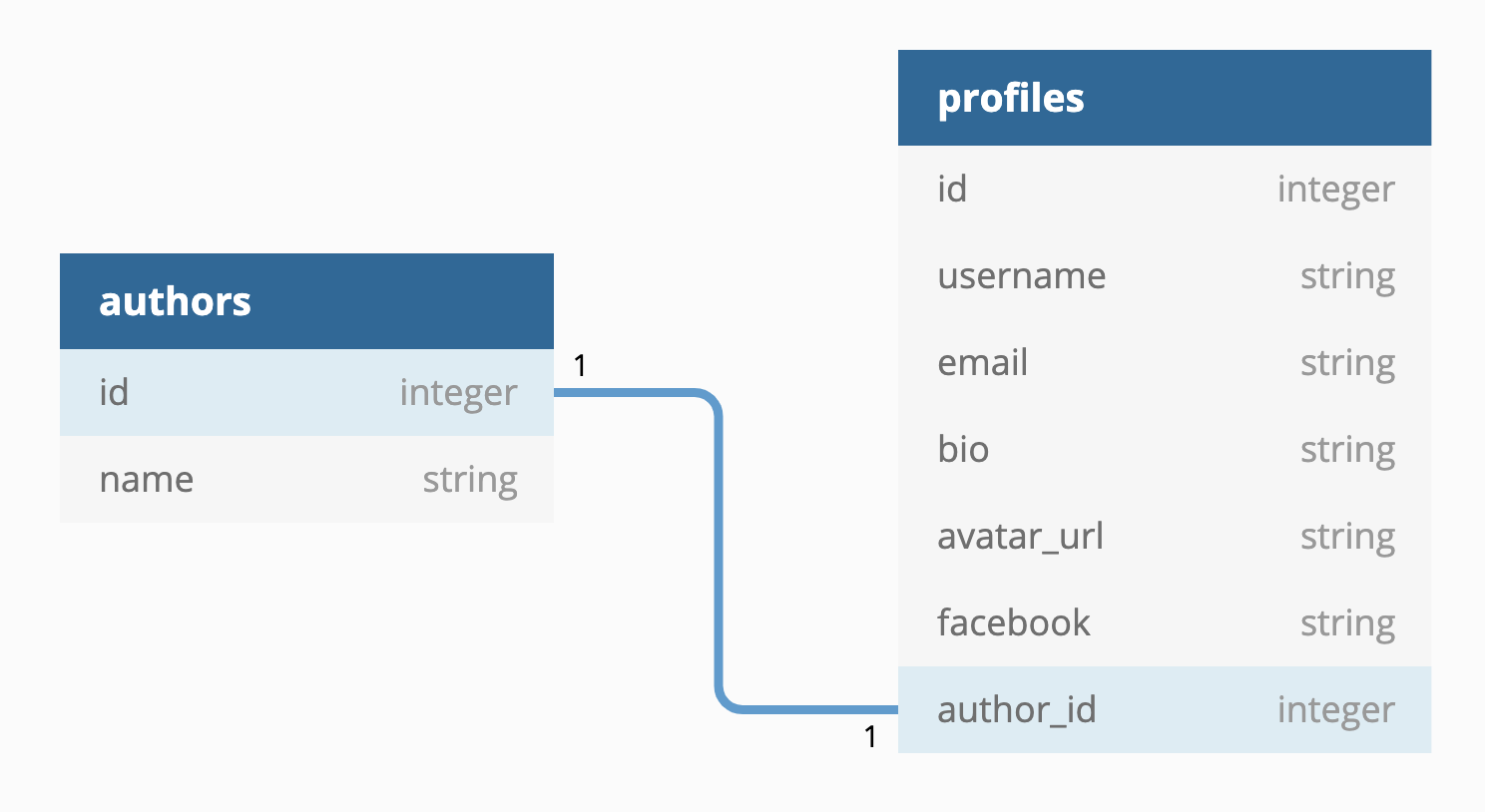
## One-to-One Relationships

A **one-to-one** relationship is probably the least common type of relationship you'll find.

One case where you might reach for a **one-to-one** relationship is for creating a separate Profile model with data related to an Author. Profiles can get pretty complex, so in large applications it can be a good idea to give them their own model. In this case:

* Every author would have one, and only one, profile.
* Every profile would have one, and only one, author.

Here's an example of what that ERD would look like:



belongs\_to makes another appearance in this relationship, but instead of has\_many the other model is declared with has\_one:

class Author < ApplicationRecord  
 has\_one :profile  
end  
  
class Profile < ApplicationRecord  
 belongs\_to :author  
end

If you're not sure which model should be declared with which macro, it's usually a safe bet to put belongs\_to on whichever model has the foreign key column in its database table.

With this in place, we can now do the following:

author = Author.first  
profile = Profile.first  
  
author.profile  
#=> #<Profile @username="ljenk">  
  
profile.author  
#=> #<Author @name="Leeroy Jenkins">

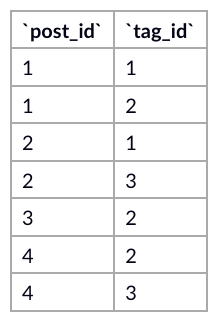
## Many-to-Many Relationships and Join Tables

Each author has many posts, each post has one author.

The universe is in balance. We're programmers, so this really disturbs us. Let's shake things up and think about tags.

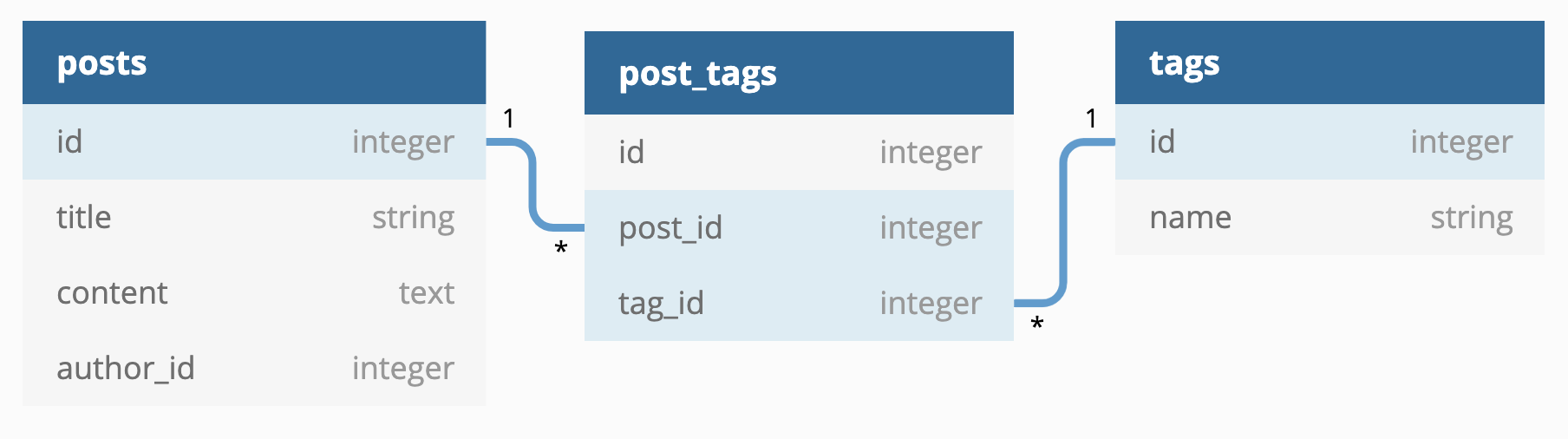
* **One-to-one** doesn't work because a post can have multiple tags.
* **One-to-many** doesn't work because a tag can appear on multiple posts.

Because there is no "owner" model in this relationship, there's also no right place to put the foreign key column.



This join table depicts the relationship between posts and tags in the seed data. Post 1 has tags 1 and 2, Post 2 has tags 1 and 3, etc.

We need a new table that sits between posts and tags:



### has\_many :through

To work with the join table, both our Post and Tag models will have a has\_many association with the post\_tags table. We also still need to associate Post and Tag themselves. Ideally, we'd like to be able to call a @my\_post.tags method, right? That's where has\_many :through comes in.

To do this requires a bit of focus. But you can do it! First of all, let's add the has\_many :post\_tags line to our Post and Tag models, and add the belongs\_to relationships to our PostTag model:

class Post < ApplicationRecord  
 belongs\_to :author  
 has\_many :post\_tags  
end  
  
class PostTag < ApplicationRecord  
 belongs\_to :post  
 belongs\_to :tag  
end  
  
class Tag < ApplicationRecord  
 has\_many :post\_tags  
end

So now we can run code like post.post\_tags to get all the join entries. This is kinda sorta what we want. What we really want is to be able to call post.tags, so we need one more has\_many relationship to complete the link between tags and posts: has\_many :through. Essentially, our Post model has many tags through the post\_tags table, and vice versa. Let's write that out:

class Post < ApplicationRecord  
 belongs\_to :author  
 has\_many :post\_tags  
 has\_many :tags, through: :post\_tags  
end  
  
class PostsTag < ApplicationRecord  
 belongs\_to :post  
 belongs\_to :tag  
end  
  
class Tag < ApplicationRecord  
 has\_many :post\_tags  
 has\_many :posts, through: :post\_tags  
end

Now we've unlocked our @post.tags and @tag.posts methods:

post = Post.first  
post.tags  
#=> [#<Tag @id=1>, #<Tag @id=2>]  
  
tag = Tag.last  
tag.posts  
#=> [#<Post @id=2>, #<Post @id=4>]

Consult the documentation to learn more about the [has many through (Links to an external site.)](http://guides.rubyonrails.org/association_basics.html#the-has-many-through-association) association.

**One-to-one** and **many-to-one** relationships only require a single foreign key, which is stored in the 'subordinate' or 'owned' model. The other model can access data in the associated table via a has\_one or has\_many method, respectively.

**Many-to-many** relationships require a join table containing a foreign key for both models. The models need to use the has\_many :through method to access data from the related table via the join table.

9.27.21

# **Active Record Associations Review Lab**

Flatiron School has started its own taxi service, FlatironX, and we need to start developing the application's domain model!

Three basic models have already been created, Passenger, Ride, and Taxi.

Your job is set up a has\_many :through relationship between these three models so that we can keep track of who's driving who around. It will work like this:

Taxi -< Rides >- Passengers

This way, a **taxi** can have many **passengers**, and a **passenger** can have many **taxis**!

The Ride model will ‘belong to’ the Passenger model and the Taxi model, the Ride model is the connecting link between the two. The Passenger model will ‘have many’ taxis through rides, and the Taxi model will ‘have many’ passengers through rides.

# **Displaying Associated Data**

Check out the controller action for this request:

# app/controllers/dog\_houses\_controller.rb  
def show  
 dog\_house = DogHouse.find(params[:id])  
 render json: dog\_house  
end

Currently, this route returns only the data about the dog house, not its reviews:

{  
 "id": 1,  
 "image": "https://assets.petco.com/petco/image/upload/f\_auto,q\_auto/1563564-right-1",  
 "name": "Cozy Studio in Historic District",  
 "city": "Denver",  
 "price": 90,  
 "favorite": false,  
 "latitude": "39.7433",  
 "longitude": "-104.98322"  
}

Based on our models, we know each dog house has many reviews associated with it, and we can use Active Record to access that data:

class DogHouse < ApplicationRecord  
 has\_many :reviews  
end

So we need some way to **include** this review data in the response from our controller!

Thankfully for us, Rails gives us some additional [serialization (Links to an external site.)](https://api.rubyonrails.org/classes/ActiveModel/Serializers/JSON.html#method-i-as_json) options when converting Active Record objects to JSON data. In this case, the include option will let us nest associated data in our response. Let's update the code in our controller:

render json: dog\_house, include: :reviews

Using include: :reviews will call the .reviews method that is provided with the has\_many :reviews macro, and will serialize the reviews as a nested array of JSON data. Try making that same GET /dog\_houses/1 request again, and you should now see the reviews listed along with the dog house they belong to. Again, your data will be different, but it should be structured as follows:

{  
 "id": 1,  
 "image": "https://assets.petco.com/petco/image/upload/f\_auto,q\_auto/1563564-right-1",  
 "name": "Cozy Studio in Historic District",  
 "city": "Denver",  
 "price": 90,  
 "favorite": false,  
 "latitude": "39.7433",  
 "longitude": "-104.98322",  
 "reviews": [  
 {  
 "id": 1,  
 "username": "emory\_rolfson",  
 "comment": "Listicle diy messenger bag food truck yuccie pug thundercats.",  
 "rating": 4,  
 "dog\_house\_id": 1  
 },  
 {  
 "id": 2,  
 "username": "willena",  
 "comment": "Brunch aesthetic williamsburg taxidermy.",  
 "rating": 1,  
 "dog\_house\_id": 1  
 }  
 ]  
}

## Nesting belongs\_to Data

One of the other pages our frontend will need is a page to list out all of the top reviews, along with their associated dog house

Again, we'd like to make just one request to get all of the data to populate this view. Currently, a GET to /reviews returns an array of all review data, but it's missing the data we need for the associated dog house:

[  
 {  
 "id": 3,  
 "username": "alton",  
 "comment": "Meh polaroid letterpress occupy freegan.",  
 "rating": 5,  
 "dog\_house\_id": 1  
 }

]

We can see that each review has an associated dog house based on the dog\_house\_id attribute, but it'd be quite the ordeal to make individual requests for each dog house to get the associated data! Again, we can leverage the power of our Active Record associations, and serialize the dog house along with each review:

Using include is to be used from parent to child only

class ReviewsController < ApplicationController  
  
 def index  
 reviews = Review.all.order(rating: :desc)  
 render json: reviews, include: :dog\_house  
 end  
  
end

With this include: :dog\_house option in place, we now get a nested object under each review representing the dog house that the review belongs to:

[  
 {  
 "id": 3,  
 "username": "alton",  
 "comment": "Meh polaroid letterpress occupy freegan.",  
 "rating": 5,  
 "dog\_house\_id": 1,  
 "dog\_house": {  
 "id": 1,  
 "image": "https://assets.petco.com/petco/image/upload/f\_auto,q\_auto/1563564-right-1",  
 "name": "Cozy Studio in Historic District",  
 "city": "Denver",  
 "price": 90,  
 "favorite": false,  
 "latitude": "39.7433",  
 "longitude": "-104.98322"  
 }  
 }

]

9.29.21

# **Deleting Associated Data**

Imagine we're creating a feature to give users the ability to delete a dog house from the site. When that dog house is deleted, what should happen to the reviews? Both from our users' perspective and from the database's perspective, it doesn't make much sense to keep a review around if there's no dog house for it to be associated with.

In fact, if you try removing a record from the database now, you'll see an error! Try this out in a Rails console session with rails c:

DogHouse.first.destroy  
# => ActiveRecord::InvalidForeignKey (SQLite3::ConstraintException: FOREIGN KEY constraint failed)

The issue is that this dog house has reviews associated with it:

DogHouse.first.reviews  
# => #ActiveRecord::Associations::CollectionProxy [#Review id: 1, ...

Those reviews must have a valid dog\_house\_id for their foreign key because of a database constraint that was established when we created the reviews table:

class CreateReviews < ActiveRecord::Migration[6.1]  
 def change  
 create\_table :reviews do |t|  
 t.string :username  
 t.string :comment  
 t.integer :rating  
  
 # foreign\_key: true establishes a relationship between a review and a dog house  
 t.belongs\_to :dog\_house, null: false, foreign\_key: true  
  
 t.timestamps  
 end  
 end  
end

So before removing the dog house, we must first remove the reviews.

## Using dependent: :destroy

As part of the class definition for our DogHouse model, we included the has\_many association reference:

# app/models/dog\_house.rb  
class DogHouse < ApplicationRecord  
 has\_many :reviews  
end

This is what lets us easily find all the reviews associated with a dog house instance by simply calling .reviews on any instance of the DogHouse class.

The has\_many association reference also lets you provide [additional options (Links to an external site.)](https://guides.rubyonrails.org/association_basics.html#options-for-has-many) to customize its behavior. In our case (and in many cases involving a one-to-many relationship), we can use the [dependent: :destroy (Links to an external site.)](https://guides.rubyonrails.org/association_basics.html#dependent) option. This will tell Active Record to delete all the associated records when the parent record is deleted.

add this code to the DogHouse class:

# app/models/dog\_house.rb  
class DogHouse < ApplicationRecord  
 has\_many :reviews, dependent: :destroy  
end

It's always a good idea to clean up any unused data in the database when deleting records, and to make sure there aren't any records that lose a necessary association when their parent record is deleted. With Active Record, we can use the dependent: :destroy option to automatically remove associated records when the parent record is deleted.

# **Nested Resource Routing**

If we look at our models, we see that a dog house has\_many :reviews and a review belongs\_to :dog\_house. Since a review can logically be considered a *child* object of a dog house, it can also be considered a *nested resource* of a dog house for routing purposes.

Nested resources give us a way to document that parent/child relationship in our routes and, ultimately, our URLs.

Let's get back into routes.rb, delete the two routes we just added, and recreate them as nested resources. We should end up with something like this:

# config/routes.rb  
  
Rails.application.routes.draw do  
  
 resources :dog\_houses, only: [:show] do  
 # nested resource for reviews  
 resources :reviews, only: [:show, :index]  
 end  
  
 resources :reviews, only: [:show, :index, :create]  
end

Now we have the resourced :dog\_houses route, but by adding the do...end we can pass it a block of its nested routes.

We can still do things to the nested resources that we do to a non-nested resource, like limit them to only certain actions. In this case, we only want to nest :show and :index under :dog\_houses.

Below that, we still have our regular resourced :reviews routes because we still want to let people see all reviews or a single review, create reviews, etc., outside of the context of a dog house.

You can see the routes available by running rails routes:

Prefix Verb URI Pattern Controller#Action  
dog\_house\_reviews GET /dog\_houses/:dog\_house\_id/reviews(.:format) reviews#index  
 dog\_house\_review GET /dog\_houses/:dog\_house\_id/reviews/:id(.:format) reviews#show  
 dog\_house GET /dog\_houses/:id(.:format) dog\_houses#show  
 reviews GET /reviews(.:format) reviews#index  
 POST /reviews(.:format) reviews#create

Now we need to update our reviews\_controller to handle the nested resource we just set up. Notice, in the 'Controller#Action' column, how now we are dealing with the reviews\_controller rather than the dog\_houses\_controller for our nested routes. Ultimately, the resource we're requesting is related to reviews, so Separation of Concerns tells us to put that code in the reviews\_controller. And, since we already have actions to handle :show and :index, we won't be repeating ourselves like we did in the dog\_houses\_controller.

Let's update index to account for the new routes:

# app/controllers/reviews\_controller.rb  
  
 def index  
 if params[:dog\_house\_id]  
 dog\_house = DogHouse.find(params[:dog\_house\_id])  
 reviews = dog\_house.reviews  
 else  
 reviews = Review.all  
 end  
 render json: reviews, include: :dog\_house  
 end

We added a condition to the reviews#index action to account for whether the user is trying to access the index of all reviews (Review.all) or just the index of all reviews for a certain dog house (dog\_house.reviews).

The condition hinges on whether there's a :dog\_house\_id key in the params hash — in other words, whether the user navigated to /dog\_houses/:dog\_house\_id/reviews or simply /reviews. We didn't have to create any new methods or make explicit calls to render new data. We just added a simple check for params[:dog\_house\_id], and we're good to go.

Where is params[:dog\_house\_id] coming from? Rails provides it for us through the nested route, so we don't have to worry about a collision with the :id parameter that reviews#show is looking for. Rails takes the parent resource's name and appends \_id to it for a nice, predictable way to find the parent resource's ID. Since some of our review routes are nested like this:

resources :dog\_houses, only: [:show] do  
 resources :reviews, only: [:show, :index]  
end

Nesting resources is a powerful tool that helps you keep your routes neat and tidy and is better than dynamic route segments for representing parent/child relationships in your system. However, as a general rule, you should only nest resources one level deep and ensure that you are considering Separation of Concerns in your routing.

10.1.21

# **Active Model Serializer**

ActiveModel::Serializer (or AMS) provides an easy way to customize how the JSON rendered by our controllers is structured. It is a very "Rails-y" tool, in that it uses a "convention over configuration" approach, and is consistent with separation of concerns. Let's take a look at how we can use it to render the JSON for our movie app.

First we need to add the gem:

# Gemfile  
#...  
gem 'active\_model\_serializers'

Run bundle install to activate the gem. Now we need to generate an ActiveModel::Serializer for our Movie model. Thankfully, the gem provides a generator for that. Drop into your console and run:

rails g serializer movie

Take a look at the generated movie\_serializer.rb in the app/serializers directory. It should look something like this:

# app/serializers/movie\_serializer.rb  
class MovieSerializer < ActiveModel::Serializer  
 attributes :id  
end

To customize our JSON, we simply provide the list of attributes that we want to be included:

class MovieSerializer < ActiveModel::Serializer  
 attributes :id, :title, :year, :length, :director, :description, :poster\_url, :category, :discount, :female\_director  
end

AMS provides a convention-based approach to serializing our resources, which means that if we have a Movie model, we can also have a MovieSerializer serializer, and by default, Rails will use our serializer if we simply call render json: movie in our controller.

### Custom Methods

So far, we've used AMS to return the values of the attributes for our Movie instances. But AMS also allows us to customize the information returned using an instance method on the MovieSerializer class. For example, say we wanted to create a movie summary that consisted of the movie's title and the first 50 characters of its description.

Let's start by adding summary to the list of attributes. Next, we'll define our method. For now, Let's put a byebug in the method's body:

class MovieSerializer < ActiveModel::Serializer  
 attributes :id, :title, :year, :length, :director, :description, :poster\_url, :category, :discount, :female\_director, :summary  
  
 def summary  
 byebug  
 end  
end

Refresh the page in the browser so you drop into byebug and enter self at the byebug prompt. The MovieSerializer instance that's returned includes an object attribute which, in turn, contains the first movie instance. This means you can enter self.object in byebug to access the movie instance, and self.object.<attribute\_name> to access a specific attribute.

With this information, let's enter q to break out of the byebug, and create our summary method:

def summary  
 "#{self.object.title} - #{self.object.description[0..49]}..."  
end

10.2.21

# **Serializing Associations**

Take a look at the new index and show routes for Director in the browser. You'll see that the JSON for the directors includes two attributes that we don't want: created\_at and updated\_at. Luckily we know how to fix this — we simply need to create a serializer for director as we did for movies:

rails g serializer director

We can then add the desired attributes to the directors\_serializer file:

# app/serializers/director\_serializer.rb  
class DirectorSerializer < ActiveModel::Serializer  
 attributes :id, :name, :birthplace, :female\_director  
end

Now if you navigate to /directors or /directors/:id you will see that we're only displaying the desired attributes.

Next, let's take a look at our new Movie index route. Now that we've removed the director and female\_director attributes, the JSON for movies no longer includes any information about director. We need to figure out how to add the information about each movie's associated director to the JSON being returned by the movies serializer. AMS allows us to do this using the same macros in the serializers that we use to set up associations in our model files. In this case, we want our serializer to reflect the fact that Movie belongs to Director, so we'll update the serializer as follows:

# serializers/movie\_serializer.rb  
class MovieSerializer < ActiveModel::Serializer  
 attributes :id, :title, :year, :length, :description, :poster\_url, :category, :discount  
  
 belongs\_to :director  
end

Now if you navigate to localhost:3000/movies/1, you should see the following:

{  
 "id": 1,  
 "title": "The Color Purple",  
 "year": 1985,  
 "length": 154,  
 "description": "Whoopi Goldberg brings Alice Walker's Pulitzer Prize-winning feminist novel to life as Celie, a Southern woman who suffered abuse over decades. A project brought to a hesitant Steven Spielberg by producer Quincy Jones, the film marks Spielberg's first female lead.",  
 "poster\_url": "https://pisces.bbystatic.com/image2/BestBuy\_US/images/products/3071/3071213\_so.jpg",  
 "category": "Drama",  
 "discount": false,  
 "director": {  
 "id": 1,  
 "name": "Steven Spielberg",  
 "birthplace": "Cincinnati, OH",  
 "female\_director": false  
 }  
}

We once again can see the director information for our movie!

We can also set up the relationship in the other direction, by adding the corresponding macro in our DirectorSerializer:

# serializers/director\_serializer.rb  
class DirectorSerializer < ActiveModel::Serializer  
 attributes :id, :name, :birthplace, :female\_director  
  
 has\_many :movies  
end

Because we have included the has\_many macro in the Director serializer, when we navigate to localhost:3000/directors/:id, we can see the list of movies that belong to that particular director:

{  
 "id": 1,  
 "name": "Steven Spielberg",  
 "birthplace": "Cincinnati, OH",  
 "female\_director": false,  
 "movies": [  
 {  
 "id": 1,  
 "title": "The Color Purple",  
 "year": 1985,  
 "length": 154,  
 "description": "Whoopi Goldberg brings Alice Walker's Pulitzer Prize-winning feminist novel to life as Celie, a Southern woman who suffered abuse over decades. A project brought to a hesitant Steven Spielberg by producer Quincy Jones, the film marks Spielberg's first female lead.",  
 "poster\_url": "https://pisces.bbystatic.com/image2/BestBuy\_US/images/products/3071/3071213\_so.jpg",  
 "category": "Drama",  
 "discount": false  
 }  
 ]  
}

**IMPORTANT**: You should only add macros to your serializers if you're sure you need the data! The level of complexity ramps up quickly as you add more macros, so keeping them to a minimum will save you headaches in the long run. It's also good to consider how much data is being sent with each request, since adding more data means running more SQL code to access that info from different tables in the database, which will make our responses slower.

## Deeply Nested Models

Now that we've got the JSON set up the way we want for our associated Movie and Director models, we can turn our attention to the Review model. Let's take another look at our model relationships:

Director -< Movies -< Reviews

We've already set up the association in the Movie and Review model files:

# app/models/movie.rb  
class Movie < ApplicationRecord  
 belongs\_to :director  
 has\_many :reviews  
end  
  
# app/models/review.rb  
class Review < ApplicationRecord  
 belongs\_to :movie  
end

Next, we'll create our review serializer:

rails g serializer review

We can also specify the attributes we want to include:

# app/serializers/review\_serializer.rb  
class ReviewSerializer < ActiveModel::Serializer  
 attributes :id, :author, :date, :url  
end

We can now go to localhost:3000/reviews and see our reviews listed. However, viewing a list of reviews separately from the information about the movies they're associated with is not particularly helpful.

What we really want to do is render the information about a movie's reviews along with the rest of the information about that movie. In fact, we don't really need to render information about reviews at all except as part of the data rendered for a particular movie!

Before we figure out how to get that in place, let's follow good programming practice and delete the code we no longer need: we'll remove the resource for reviews from the routes.rb file and the index action from the ReviewsController.

Once that's done, to get reviews included in the JSON that's returned for a given movie, we'll simply add the appropriate macro to the MovieSerializer:

class MovieSerializer < ActiveModel::Serializer  
 attributes :id, :title, :year, :length, :description, :poster\_url, :category, :discount  
  
 belongs\_to :director  
 has\_many :reviews  
end

**AMS only nests associations one level deep**.

This behavior is intended to protect against overly complex JSON that's nested many layers deep. Luckily, we can override the behavior by using the [include option (Links to an external site.)](https://github.com/rails-api/active_model_serializers/blob/v0.10.6/docs/general/adapters.md#include-option) in the top-level controller — in this case, the DirectorsController:

# app/controllers/directors\_controller.rb  
class DirectorsController < ApplicationController  
  
 def index  
 directors = Director.all  
 render json: directors, include: ['movies', 'movies.reviews']  
 end  
  
 def show  
 director = Director.find(params[:id])  
 render json: director, include: ['movies', 'movies.reviews']  
 end  
  
end

Let's take a look at the render statement in our show action:

render json: director, include: ['movies', 'movies.reviews']

This code tells AMS that we want to render information for the director, and to also include information for the movies associated with that director, and for the reviews associated with those movies.

Finally, because we're using our custom DirectorMovieSerializer to render the movies in our Director routes, we also need to add the has\_many :reviews macro to that serializer:

class DirectorMovieSerializer < ActiveModel::Serializer  
 attributes :title, :year  
  
 has\_many :reviews  
end

## Conclusion

* To customize the JSON returned for a resource, create a **serializer** for that resource and list the desired attributes.
* The serializer is used **implicitly** by Rails based on naming conventions; to override this, custom serializers can be **explicitly** passed in the controller.
* AMS enables the use of the belongs\_to and has\_many macros in serializers to render associated data; these macros should be used sparingly.
* By default, AMS will only nest associations one level deep in the serialized JSON. To override this, the include option can be used in the controller.

10.6.21

# Cookies and Sessions

Cookies are small pieces of information that are sent from the server to the client. They are then stored on the client (in the browser) and sent back to the server with each subsequent request.

## Configuring Cookies in Rails APIs

# config/application.rb  
module MyApp  
 class Application < Rails::Application  
 config.load\_defaults 6.1  
 # This is set in apps generated with the --api flag, and removes session/cookie middleware  
 config.api\_only = true  
  
 # Must add these lines!  
 # Adding back cookies and session middleware  
 config.middleware.use ActionDispatch::Cookies  
 config.middleware.use ActionDispatch::Session::CookieStore  
  
 # Use SameSite=Strict for all cookies to help protect against CSRF  
 config.action\_dispatch.cookies\_same\_site\_protection = :strict  
 end  
end

To access the cookies hash in our controllers, we also need to include the ActionController::Cookies module in our ApplicationController:

# app/controllers/application\_controller.rb  
class ApplicationController < ActionController::API  
 include ActionController::Cookies  
end

Then, in the browser, make a request to http://localhost:3000/sessions. This will run the code in our SessionsController#index method:

def index  
 session[:session\_hello] ||= "World"  
 cookies[:cookies\_hello] ||= "World"  
 render json: { session: session, cookies: cookies.to\_hash }  
end

In this method, we're setting values on the session hash and the cookies hash, and serializing them in the response so we can view their values in the browser.

If you haven't encountered [||= (Links to an external site.)](http://www.rubyinside.com/what-rubys-double-pipe-or-equals-really-does-5488.html) syntax in Ruby, it's a shorthand way to assign a value if the current value is nil or false. So if session[:session\_hello] has not already been assigned a value, it will be assigned a value of "World". Otherwise, it won't get assigned a new value.

10.8.21

**Authenticating Users**

Let's write a SessionsController to handle our login route. This controller has one action, create, which we'll map in routes.rb for POST requests to /login:

post "/login", to: "sessions#create"

Typically, your create method would look up a user in the database, verify their login credentials, and then store the authenticated user's id in the session:

class SessionsController < ApplicationController  
 def create  
 user = User.find\_by(username: params[:username])  
 session[:user\_id] = user.id  
 render json: user  
 end  
end

There's no way for the server to log you out right now. To log yourself out, you'll have to delete the cookie from your browser.

we need a way of getting the user data from the backend into state when the page first loads.

Here's how we might accomplish that. First, we need a route to retrieve the user's data from the database using the session hash:

get "/me", to: "users#show"

class UsersController < ApplicationController  
 def show  
 user = User.find\_by(id: session[:user\_id])  
 if user  
 render json: user  
 else  
 render json: { error: "Not authorized" }, status: :unauthorized  
 end  
 end  
end

## Logging Out

The log out flow is even simpler. We can add a new route for logging out:

delete "/logout", to: "sessions#destroy"

Then add a SessionsController#destroy method, which will clear the username out of the session:

def destroy  
 session.delete :user\_id  
 head :no\_content  
end

# Authorizing Requests

From a technical perspective, what does it actually mean for a user to *log in*? When a user logs in, all we are doing is using cookies to add their :user\_id to the session hash.

The first thing you might do is to add a **guard clause** as the first line of DocumentsController#show:

def show  
 return render json: { error: "Not authorized" }, status: :unauthorized unless session.include? :user\_id  
 document = Document.find(params[:id])  
 render json: document  
end

Unless the session includes :user\_id, we return an error. status: :unauthorized will return the specified HTTP status code. In this case, if a user isn't logged in, we return 401 Unauthorized.

## Refactor

Fortunately, Rails gives us a solution: [before\_action (Links to an external site.)](http://guides.rubyonrails.org/action_controller_overview.html" \l "filters" \t "_blank). We can refactor our code like so:

class DocumentsController < ApplicationController  
 before\_action :authorize  
  
 def show  
 document = Document.find(params[:id])  
 render json: document  
 end  
  
 def index  
 documents = Document.all  
 render json: documents  
 end  
  
 def create  
 document = Document.create(author\_id: session[:user\_id])  
 render json: document, status: :created  
 end  
  
 private  
  
 def authorize  
 return render json: { error: "Not authorized" }, status: :unauthorized unless session.include? :user\_id  
 end  
end

This is a call to the ActionController class method before\_action. before\_action registers a filter. A filter is a method which runs **before**, **after**, or **around** a controller's action. In this case, the filter runs before all DocumentsController's actions, and kicks requests out with 401 Unauthorized unless they're logged in.

## Skipping Filters for Certain Actions

What if we wanted to let anyone see a list of documents, but keep the before\_action filter for other DocumentsController methods? We could do this:

class DocumentsController < ApplicationController  
 before\_action :authorize  
 skip\_before\_action :authorize, only: [:index]  
  
 # ...  
end

This class method tells Rails to skip the authorize filter only on the index action:

skip\_before\_action :authorize, only: [:index]

# **Authorizing Requests Lab**

class MembersOnlyArticlesController < ApplicationController

rescue\_from ActiveRecord::RecordNotFound, with: :record\_not\_found

before\_action :authorize

def index

articles = Article.where(is\_member\_only: true).includes(:user).order(created\_at: :desc)

render json: articles, each\_serializer: ArticleListSerializer

end

def show

article = Article.find(params[:id])

render json: article

end

private

def record\_not\_found

render json: { error: "Article not found" }, status: :not\_found

end

# If the user is signed in, the #index and #show actions should return the JSON data for the members-only articles.

# If a user is not signed in, the #index and #show actions should return a status code of 401 unauthorized, along with an error message

def authorize

return render json: {error: "Not authorized" }, status: :unauthorized unless session.include? :user\_id

end

end

10.11.21

# Password Protection

## Hashing Passwords

Instead of storing their password in plain text, we store a hashed version of their password. A *hash* is a *fixed-length* output computed by feeding a string to a *hash function*. Hash functions have the property that they will always produce the same output given the same input.

A helpful analogy for a hash function is making a smoothie. If I put the exact same ingredients into the blender, I'll get the exact same smoothie every time. But there's no way to reverse the operation, and get back the original ingredients from the smoothie.

We could imagine using this function to avoid storing passwords in the database. Our User model and SessionsController might look like this:

# app/models/user.rb  
class User < ApplicationRecord  
  
 # takes a plaintext password and stores a hashed version as a password\_digest  
 def password=(new\_password)  
 self.password\_digest = dumb\_hash(new\_password)  
 end  
  
 # checks if the hashed plaintext password matches the password\_digest  
 def authenticate(password)  
 return nil unless dumb\_hash(password) == password\_digest  
 self  
 end  
  
 private  
  
 # the hashing method  
 def dumb\_hash(input)  
 input.bytes.reduce(:+)  
 end  
end  
  
# app/controllers/sessions\_controller.rb  
class SessionsController < ApplicationController  
 def create  
 user = User.find\_by(username: params[:username])  
 if user&.authenticate(params[:password])  
 session[:user\_id] = user.id  
 render json: user, status: :created  
 else  
 render json: { error: "Invalid username or password" }, status: :unauthorized  
 end  
 end  
end

**Note:** [&. (Links to an external site.)](https://mitrev.net/ruby/2015/11/13/the-operator-in-ruby/) is known in Ruby as the "safe navigation operator". If user is nil, it will return nil; if not, it will call the .authenticate method on user. It would be similar to writing user && user.authenticate(params[:password]).

Instead, Rails uses a library called BCrypt. BCrypt is designed with these properties in mind:

1. BCrypt hashes similar strings to very different values.
2. It is a cryptographic hash. That means that, if you have an output in mind, finding a string which produces that output is designed to be "very difficult." "Very difficult" means "even if Google put all their computers on it, they couldn't do it."
3. BCrypt is designed to be slow. It is intentionally computationally expensive.

The last two features make BCrypt a particularly good choice for passwords. (2) means that, even if an attacker gets your database of hashed passwords, it is not easy for them to turn a hash back into its original string. (3) means that, even if an attacker has a dictionary of common passwords to check against, it will still take them a considerable amount of time to check for your password against that list.

The [BCrypt gem (Links to an external site.)](https://github.com/bcrypt-ruby/bcrypt-ruby" \t "_blank) is open source, and their documentation has some excellent examples that demonstrate this functionality. If you're interested in exploring more, their docs and source code are a great resource.

# Gemfile:  
gem 'bcrypt'  
  
# app/models/user.rb  
class User < ActiveRecord::Base  
  
 # generate a salted + hashed password and save it to password\_digest  
 def password=(new\_password)  
 salt = BCrypt::Engine::generate\_salt  
 # => $2a$12$UW5etUc/o1YL4sSdeTBPku  
 self.password\_digest = BCrypt::Engine::hash\_secret(new\_password, salt)  
 # => $2a$12$UW5etUc/o1YL4sSdeTBPkueUWwNIPNdQNAwzuSGkS3L5coBKMMZHm"  
 end  
  
 # check the plaintext password against the salted + hashed password  
 def authenticate(password)  
 # Salts generated by generate\_salt are always 29 chars long.  
 salt = password\_digest[0..28]  
 # compare the saved password\_digest against the plaintext password by running the plaintext password through the same hashing function  
 return nil unless BCrypt::Engine::hash\_secret(password, salt) == self.password\_digest  
 self  
 end  
end

## Rails Makes It Easier

You don't have to deal with all this yourself. Rails provides a method called has\_secure\_password that you can use on your Active Record models to handle all of this. It looks like this:

class User < ApplicationRecord  
 has\_secure\_password  
end

To use the has\_secure\_password macro, you'll need to add gem 'bcrypt' to your Gemfile if it isn't there already.

When using [has\_secure\_password (Links to an external site.)](http://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html" \t "_blank), Rails will use the bcrypt gem to hash and salt all passwords on the User model.

The [has\_secure\_password (Links to an external site.)](http://api.rubyonrails.org/classes/ActiveModel/SecurePassword/ClassMethods.html" \t "_blank) method also provides two new instance methods on your User model: password and password\_confirmation. These methods don't correspond to database columns! Instead, to make these methods work, your users table **must** have a password\_digest column:

create\_table :users do |t|  
 t.string :username  
 t.string :password\_digest  
  
 t.timestamps  
end

These two instance methods enable you to easily include password and password confirmation fields in a signup (or password reset) form. has\_secure\_password handles these fields by adding a before\_save hook to your model that compares password and password\_confirmation. If they match (or if password\_confirmation is nil), the user is saved and the hashed version of the password is stored in the password\_digest column of the database, pretty much exactly like our example code before did.

Under the hood, has\_secure\_password calls upon an Active Record helper method, [validates\_confirmation\_of][validates*confirmation*of]. As such, as with other Active Record validator methods, when the fields don't match and the validation fails, an ActiveRecord::RecordInvalid exception will be raised. You can handle this exception by using rescue or rescue\_from.

10.17.21

# Intro to Deploying

As a web developer, it's essential to familiarize yourself with the development process for a few reasons:

* It's increasingly expected that developers participate in the deployment process. While many companies employ specialized DevOps (Developer Operations) engineers, it is not uncommon for engineers on other teams to have a hand in deploying and troubleshooting issues in production.

A note on databases: for the labs in this phase, we've been using SQLite as a lightweight database. It's great for quick development, but it doesn't scale well to larger applications. We'll be using Postgresql for our database moving forward. More on that later!

## Deploying Options

When it comes to deciding which platform to use to host our deployed applications, there are a number of options, each of which comes with some tradeoffs. For Rails in particular, a few popular choices are:

* [Heroku (Links to an external site.)](https://www.heroku.com/about)
* [Amazon Web Services (AWS) (Links to an external site.)](https://aws.amazon.com/)
* [Digital Ocean (Links to an external site.)](https://www.digitalocean.com/)

One thing these services all have in common: they all have the ability to easily configure an environment to run our Rails applications in. These services own the hardware (the physical servers and networking infrastructure) that your code will run on, but they also have configurable [**containers** (Links to an external site.)](https://en.wikipedia.org/wiki/OS-level_virtualization) with resources dedicated to run your code.

There are a lot of considerations when it comes to choosing a platform for deploying your application, such as:

* **Cost**: How much will this platform charge me? How much do I get for free? Flat rate per month? Calculated based on usage?
* **Ease of use**: How much work does it take to deploy my application the first time? How easy is it to deploy new versions?
* **Scalability**: As my application gets more users, how easy is it to handle increased traffic?
* **Performance**: Does my application need to handle a lot of traffic? Be optimized for video processing? Store a lot of files? Handle large database tables?

For now, since we're deploying our very first Rails projects, we're going to prioritize **cost** and **ease of use** to decide on a platform, and use Heroku for deploying in the coming lessons.

[Heroku (Links to an external site.)](https://www.heroku.com/about) is first and foremost a Platform as a Service (PaaS), which means they manage the hardware your code runs on as well as the software environment, with an aim of making it as simple as possible to take the code from your machine and run it on theirs. Heroku also has a free tier for developers to try out the service at no cost.

10.18.21

# Deploying a Rails API to Heroku

## Creating a Rails App to Deploy

We'll be following the steps in the [Heroku Rails Deploying Guide (Links to an external site.)](https://devcenter.heroku.com/articles/getting-started-with-rails6), so if you get stuck and are looking for more assistance, check that guide first.

The first thing we'll need to do is create our new Rails application. Make sure you're in a non-lab directory, then run:

rails new bird-app --api --minimal --database=postgresql

This will set up our app to run in API mode, with the minimum dependencies needed, and with Postgresql as the database.

cd into the app, and run this command:

bundle lock --add-platform x86\_64-linux --add-platform ruby

This will add additional platforms to your Gemfile.lock file that will allow the necessary dependencies to be installed after you deploy your app.

Then run this command to generate the database and run the migrations and seed file:

rails db:create db:migrate db:seed

rails db:create creates a new Postgresql database to be associated with your application based on the configuration in the config/database.yml file. Unlike with SQLite, the actual database file isn't created in the db folder; it lives elsewhere in your file system, depending on your Postgresql configuration. If you have problems with this step, see the **Troubleshooting** section below.

anybody on a macbook air M1 chip run into this error when trying to run rails db:create i already downloaded and initialized postgresql, now running into this:  
VCNTX@MacBook-Air bird-app % rails db:create  
rails aborted!  
LoadError: dlopen(/Users/VCNTX/.rvm/gems/ruby-2.7.3/gems/pg-1.2.3/lib/pg\_ext.bundle, 9): Library not loaded: /usr/local/opt/postgresql/lib/libpq.5.dylib  
  Referenced from: /Users/VCNTX/.rvm/gems/ruby-2.7.3/gems/pg-1.2.3/lib/pg\_ext.bundle  
  Reason: image not found - /Users/VCNTX/.rvm/gems/ruby-2.7.3/gems/pg-1.2.3/lib/pg\_ext.bundle  
/Users/VCNTX/.rvm/gems/ruby-2.7.3/gems/activesupport-

<https://restitution.bar/host-https-stackoverflow.com/questions/9182645/library-not-loaded-libpq-5-dylib>

thanks for the help all this is what fixed me up

## Deploying

Now that we've got some working code, it's time to get that code to run on a Heroku server! The process of uploading our code to Heroku is managed by Git. This makes it easy to deploy new versions using a tool most developers, including yourself, are already familiar with.

Make a commit to save your changes:

git add .  
git commit -m 'Initial commit'

Next, you'll need to create an application on Heroku:

heroku create

Now, deploying your code is as simple as using git push to upload the changes from your repository to Heroku:

git push heroku main

Note: depending on your Git configuration, your default branch might be named master or main. You can verify which by running git branch --show-current. If it's master, you'll need to run git push heroku master instead.

You've successfully pushed up your code!

To migrate and seed the database on the server, run:

heroku run rails db:migrate db:seed

When you prefix any command with heroku run, it will run that command on the server where your application was deployed. This command is very useful for troubleshooting: you can even run heroku run rails c to open a Rails console on the server!

You can now visit the site in the browser by running heroku open. Note that, because there is no root path ('/') defined in our routes, you will see a Page Not Found error when the app opens. Navigate to the /birds endpoint and verify that you are able to see an array of JSON data for all the birds in the database. If you aren't able to, check out the troubleshooting section below, or the [troubleshooting guide on Heroku (Links to an external site.)](https://devcenter.heroku.com/articles/getting-started-with-rails6#troubleshooting).

# Deploying a Rails-React App to Heroku

After downloading the code, set up the repository locally:

bundle install  
rails db:create db:migrate db:seed  
npm install --prefix client

This application has a Rails API with session-based authentication; a React frontend using React Router for client-side routing; and Postgresql for the database.

**React Production Build**

One of the great features that Create React App provides to developers is the ability to build different versions of a React application for different environments.

When working in the **development** environment, a typical workflow for adding new features to a React application is something like this:

* Run npm start to run a development server
* Make changes to the app by editing the files
* View those changes in the browser
* To enable this excellent developer experience, Create React App uses [webpack (Links to an external site.)](https://webpack.js.org/) under the hood to create a development server with hot module reloading, so any changes to the files in our application will be instantly visible to us in the browser. It also has a lot of other nice features in development mode, like showing us good error and warning messages via the console.

For our app to run in production, we have a different set of needs:

* **Build** the files needed to run our app in the browser, keeping them as small as possible
* **Serve** the application's files from a server hosted online, rather than a local webpack development server
* Don't show any error messages/warnings that are meant for developers rather than our website's users

To demonstrate this process of **building** the production version of our React app and **serving** it from the Rails app, do the following:

* Run npm run build --prefix client.
  + This will generate a bundled and minified version of our React app in the client/build folder. Check out the files in that directory, and in particular the JavaScript files. You'll notice they have very little resemblance to the files in your src directory! This is because of that **bundling** and **minification** process: taking the source code you wrote, along with any external JavaScript libraries your code depends on, and squishing it as small as possible.
* Copy all of the files and folders from within the client/build directory, and move them to the public directory. Make sure you move them to public, not client/public.
  + The public directory is used by Rails to serve assets, so when we run the Rails server, it will be able to display the files from our production version of the React application. When a user visits http://localhost:3000, Rails will return the index.html file from this directory.
* Run rails s and visit [http://localhost:3000 (Links to an external site.)](http://localhost:3000/) in the browser. You should see the production version of the React application!

## Configuring Rails for Client-Side Routing

In our React application, we're using React Router to handle client-side routing. Client-side routing means that a user should be able to navigate to the React application, load all the HTML/CSS/JavaScript code just **once**, and then click through links in our site to navigate to different pages without making another request to the server for a new HTML document.

When we run the app using npm start and webpack is handling the React server, it can handle these client-side routing requests just fine! **However**, when we're running React within the Rails application, we also have routes defined for our Rails API, and Rails will be responsible for all the routing logic in our application. So let's think about what will happen from the point of view of **Rails** when a user makes a request to these routes.

* GET /: Rails will respond with the public/index.html file
* GET /new: Rails will look for a GET /new route in the config/routes.rb file. If we don't have this route defined, it will return a 404 error.

Any other client-side routes we define in React will have the same issue as /new: since Rails is handling the routing logic, it will look for routes defined in the config/routes.rb file to determine how to handle all requests.

We can solve this problem by setting up a **custom route** in our Rails application, and handle any requests that come through that **aren't** requests for our API routes by returning the public/index.html file instead.

Here's how it works:

# config/routes.rb  
Rails.application.routes.draw do  
 namespace :api do  
 resources :recipes, only: [:index, :create]  
 post "/signup", to: "users#create"  
 get "/me", to: "users#show"  
 post "/login", to: "sessions#create"  
 delete "/logout", to: "sessions#destroy"  
 end  
  
 get "\*path", to: "fallback#index", constraints: ->(req) { !req.xhr? && req.format.html? }  
end

All the routes for our API are defined **first** in the routes.rb file. We use the [namespacing (Links to an external site.)](https://guides.rubyonrails.org/routing.html" \l "controller-namespaces-and-routing" \t "_blank) to differentiate the API requests from other requests.

The last method in the routes.rb file handles all other GET requests by sending them to a special FallbackController with an index action:

# app/controllers/fallback\_controller.rb  
class FallbackController < ActionController::Base  
 def index  
 render file: 'public/index.html'  
 end  
end

This action has just one job: to render the HTML file for our React application!

**Heroku Build Process**

Think about the steps to build our React application locally. What did we have to do to build the React application in such a way that it could be served by our Rails application? Well, we had to:

* Run npm install --prefix client to install any dependencies
* Use npm run build --prefix client to create the production app
* Move the code from the client/build folder to the public folder
* Run rails s

Ideally, we'd like to be able to **automate** those steps when we deploy this app to Heroku, so we can just push up new versions of our code to Heroku and deploy them like we were able to do in the previous lesson.

Thankfully, Heroku lets us do just that! Let's get started with the deploying process and talk through how this automation works.

First, in the demo project directory, create a new app on Heroku:

heroku create

Next, we'll need to tell Heroku that this project is not **just** a Rails project; we'll need to run some **NodeJS** code as well in order to execute the build scripts for our React application. We can do this via Heroku [buildpacks (Links to an external site.)](https://devcenter.heroku.com/articles/using-multiple-buildpacks-for-an-app" \t "_blank):

heroku buildpacks:add heroku/nodejs --index 1  
heroku buildpacks:add heroku/ruby --index 2

This will tell Heroku to first run a build script for our React app using NodeJS before running the build script for our Rails app (running bundle install and rails s).

To deploy the app, just like before, run:

git push heroku master

This will kick off the build process on Heroku for the React app, then the Rails app next. You should be able to visit the deployed site now and see the full project live on the internet!

<https://blog.heroku.com/a-rock-solid-modern-web-stack>

**Phase 4 Project Guidance**

**Planning**

**User Stories**

Start by deciding on a domain for your app (such as "AirBNB for dogs"). Then, decide what **user stories** your app will need. It is helpful to break up your user stories between what is required for the [**Minimum Viable Product** (MVP) (Links to an external site.)](https://blog.crisp.se/2016/01/25/henrikkniberg/making-sense-of-mvp) version of your app, and what you'd like to save for stretch features after you've met your MVP goals.

For example:

* MVP: As a user, I can:
  + Log into the site
  + View a list of all available dog houses in my area and their reviews
  + Create a review for one specific dog house
  + Modify or delete a review that I left
  + Create a new dog house listing
* Stretch: As a user, I can:
  + View dog houses on a map
  + Search dog houses based on their distance from my location
  + Filter dog houses based on their average rating

**Models and Relationships**

After deciding on your app's user stories, you can design the **models** that your application will need in order to represent these user stories.

Look at the list of your user stories, and pick out the different nouns/objects that you need to represent these user stories. These objects inform what models you need. For example, from the list above, we have:

* User
* Dog House
* Review

You can also get a sense of the relationships between the models and use that as the basis of your **Entity Relationship Diagram** (ERD). For example, we can tell based on the user stories above that a **review** belongs to a specific **user** — since a user is able to create a review — and a **review** belongs to a specific **dog house**.

You can use a website like [dbdiagram.io (Links to an external site.)](https://dbdiagram.io/) to help make an ERD and represent these relationships, or draw out something simple:

User -< Review >- DogHouse  
  
DogHouse >- User

This is also a good time to think about what attributes your models will need. What foreign keys are needed to establish relationships? What other attributes might you need to display data in your frontend, or make other aspects of your user stories work?

**Wireframes**

For your frontend, it's a good idea to follow the ideas from [Thinking in React (Links to an external site.)](https://reactjs.org/docs/thinking-in-react.html) as you're designing your React application. That means starting with a visual representation of what your application should look like, in the form of a wireframe. The wireframe should give you a basic visual representation of what each page of your application should look like, and it should capture all of your user stories.

Here are some tools for wireframing (pen and paper is also a fine choice!):

* [Excalidraw - basic hand-drawn wireframes (Links to an external site.)](https://excalidraw.com/)
* [Figma - professional design tool (Links to an external site.)](https://www.figma.com/)
* [Balsamiq - professional wireframe tool (Links to an external site.)](https://balsamiq.com/)

Use your wireframe to plan out what components you'll need and design your component hierarchy, following the ideas from [Thinking in React (Links to an external site.)](https://reactjs.org/docs/thinking-in-react.html).

**Execution**

Once you have your plan in place, and have a sense of your:

* User stories
* Models (including relationships and attributes)
* Wireframes

It's time to start building! As you're building, work on each feature in [vertical slices (Links to an external site.)](https://agileforall.com/vertical-slices-and-scale/) rather than horizontal. For example, rather than building out **all** the models, routes and controller actions in the backend, then working on the components in the frontend and finally styling everything, work on one **feature** at a time, such as working on login, then displaying a list of dog houses, then leaving a review.

You can visualize all the parts of an app you need to build as a grid, with the desired features in columns and the different layers of the stack in rows:

|  | **Sign in flow** | **View dog houses** | **Leave a review** |
| --- | --- | --- | --- |
| Migrations |  |  |  |
| Models |  |  |  |
| Seed Data |  |  |  |
| Controller actions |  |  |  |
| View Logic |  |  |  |
| Data Fetching |  |  |  |
| Styling |  |  |  |

You may be tempted to order your project timeline row-by-row. Do not do this! If you try to build all your migrations, then all your models, then all your controllers, then all your fetch calls, then all your view logic you will have a bad time. Inevitably, your view logic ends up requiring changes to the underlying layers, and you end up building models that you never use. If you instead build **each feature** (each **vertical slice**) in its entirety before moving on to the next feature, you'll minimize rewriting, and end up with working features without waste.

* Add feature by feature, not model by model or layer by layer.
* Test each feature, add styles, and create seed data as you go (not all at once at the end)

Also, remember to prioritize your MVP features. It can be tempting to try and build everything at once, but that is a sure-fire way to end up with many broken features instead of a solid core of working features.

**Deploying**

The template project has all the starter code needed to help you deploy your application to Heroku. It's recommended to deploy your project early and push up changes often to ensure that your code works equally well in production and development environments.

Follow the instructions in the template to deploy your app!

**Resources**

* [Project Template: React/Rails API (Links to an external site.)](https://github.com/learn-co-curriculum/project-template-react-rails-api)
* [ERD Visualizations: dbdiagram.io (Links to an external site.)](https://dbdiagram.io/)
* [Excalidraw - basic hand-drawn wireframes (Links to an external site.)](https://excalidraw.com/)
* [Figma - professional design tool (Links to an external site.)](https://www.figma.com/)
* [Balsamiq - professional wireframe tool](https://balsamiq.com/)

using the command ‘git clone’ and then pasting in the url for the repo and then adding a space and the desired file name and you will create the file with the desired name!

use Arteaga real estate readme to create a project with a template.

If running into this error:

rails opensslErrorStack: [ 'error:03000086:digital envelope routines::initialization error' ], library: 'digital envelope routines', reason: 'unsupported', code: 'ERR\_OSSL\_EVP\_UNSUPPORTED'

use the command: nvm use 14.18.1

use the command rafce

to generate a react arrow function with export

11.1.21

# **React State**

## What's state?

Let's quickly talk about what **state** is in React. State is data that is **dynamic** in your component: it changes over time as users interact with your application.

To do this, we must first import a function from React called useState. This special function is a **React Hook** that will let us "hook into" React's internal state inside of our function component.

### Initializing State

Once we've imported the useState hook, we can call it inside of our component, like so:

import React, { useState } from "react";

function Counter() {

const [count, setCount] = useState(0);

return <button>{count}</button>;

}

When we call useState(0) inside the function component, we're telling React to create some new internal state for our component with an **initial value** of 0 (or whatever value we pass to useState when we call it).

useState will return an **array** that has two variables inside of it:

* count: a reference to the current value of that state in React's internals
* setCount: a setter function so we can update that state
* We *could* create the variables and set their values by accessing the elements from the array individually, like this:
* const countState = useState(0);
* // => [0, setStateFunction]
* const count = countState[0];
* const setCount = countState[1];
* But to clean up the code, React recommends using [array destructuring (Links to an external site.)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) to achieve the same result in one line of code instead of three:
* const [count, setCount] = useState(0);
* Because setState is asynchronous — i.e., because the value of count isn't updated immediately — when setCount is called the second time, count is still equal to 0! As a result, we are effectively calling setCount(1) in *both* cases.
* React actually provides a built in solution for this problem. Rather than passing a new value into setCount, we can instead pass a callback function. That function, when called inside setCount, will be passed the state variable containing the current state of count. This is typically referred to as the *previous state* since it's the state before the current call to setCount is executed. With this knowledge, we can rewrite the increment function to:
* function increment() {
* setCount((currentCount) => currentCount + 1);
* setCount((currentCount) => currentCount + 1);
* }
* When using the callback version of setCount, React will pass in the current value of count before updating it. Now our code works as intended and updates count by 2 when the button is clicked.
* As a rule of thumb, **any time you need to set state based on the current value of state, you should use the callback syntax**.

### Rules of Hooks

Since the useState hook is the first of several React Hooks we'll be learning about, now's a good time to review some general [rules for working with hooks (Links to an external site.)](https://reactjs.org/docs/hooks-rules.html):

#### Only Call Hooks at the Top Level

Remember: **state is only for values that are expected to change during the component's life**.

First, let's decide: do we actually need state for this feature? We need to determine if the data for this feature is *static* (doesn't change) or *dynamic* (does change). If it's dynamic, we'll definitely need state! We should also ask if this feature could be made using **props** instead of **state**.

Our full checklist looks like this:

* 🚫 Is it passed as a prop?
* 🚫 Can you compute it based on any other state or props in your component?
* 🚫 Does it remain unchanged over time?

### How do I pass a parameter to an event handler or callback?

You can use an arrow function to wrap around an event handler and pass parameters:

<button onClick={() => this.handleClick(id)} />

This is equivalent to calling .bind:

<button onClick={this.handleClick.bind(this, id)} />

11.2.21 Scrimba play list thinking in React

Step 1 break up the UI into a component hierarchy

- Is it passed in from a parent via props? If so, it probably isn’t state.

- Does it remain unchanged over time? If so, it probably isn’t state.

- Can you compute it based on any other state or props in your component? If so, it isn’t state.

filteredProduct = products.filter(product => product.name === search)

Step 4: Identify Where Your State Should Live

- Identify every component that renders something based on that state.

- (SearchBar and ProductTable)

- Find a common owner component (a single component above all the components that need the state in the hierarchy).

- (FilterableProductTable)

- Either the common owner or another component higher up in the hierarchy should own the state.

- If you can’t find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.

# **React Information Flow**

In order to propagate information in the opposite direction, we can send a **callback function as props** from the parent component to its child.

This allows the callback to be *owned* by a different component than the one invoking it. Once invoked, the callback can send data to or change state in the parent component that *owns* it, instead of the child component that *invoked* it.

* **Sibling components cannot pass data to each other directly**
* **Data can only flow up and down between parent and child**
* So if we update the color of one Child component, we have no way to pass that data to the *other* Child component.
* The solution is to store the color of the Child in the state of the Parent component.

For information to propagate **down** the component tree, parents pass props to their children.

For information to propagate **up** the component tree, we must invoke **callbacks** that were passed from parents to children as props.

Components of the same level (sibling components) cannot communicate directly! We can only communicate up and down the component tree. So if multiple components need to share the same information, that state should live in the parent component (or a more general ancestor).

## Resources

* [Lifting State Up](https://reactjs.org/docs/lifting-state-up.html)

git reset --hard HEAD

Error: Invalid hook call. Hooks can only be called inside of the body of a function component. This could happen for one of the following reasons: 1. You might have mismatching versions of React and the renderer (such as React DOM) 2. You might be breaking the Rules of Hooks 3. You might have more than one copy of React in the same app See https://reactjs.org/link/invalid-hook-call for tips about how to debug and fix this problem.

DO NOT run any “npm” installs outside of client folder or with out –prefix client

When getting this error I installed: gem install shutup

and ran:

shutup

A server is already running. Check /home/..name/rprojects/railsapp/tmp/pids/server.pid.

Exiting

Error: rails request body showing up twice in params

Solution: try params.require(:vender).permit(the same stuff)