geo-js$_{\text{rails}}$

Steven Fernandez

October 15, 2023

Contents

---

1 Requirements

1. Allow IP Addresses to be provided to an endpoint.
2. A second endpoint should allow users to query all previously obtained ip addresses. This implies persistence is needed. This is the `index` action.
3. Allow user to filter ip addresses by country and city.
4. Everything returns JSON.

2 Implementation

- I'm gonna use `dotenv-rails` to store some of the password information that I'm gonna use in development. I'm gonna use rails bytes for this too:

```
rails app:template LOCATION='https://railsbytes.com/script/zOvsQ0'
```

- Touch a file `.env.development.local` and `.env.test.local`. Then create the template files to commit to version control with:

```
dotenv -t .env.development.local
dotenv -t .env.test.local
```

- Make sure that you pull the `.env.development.local.template` and `.env.test.local.template` and modify the values to whatever string you want.
- Of course we need to modify the `.gitignore` file to not commit secrets into production.
- Create the databases and the users based on what you have in the `.env.development.local` and `.env.test.local` files:
- Launch a sudo shell with `postgres` user on `psql`

```
sudo -u postgres psql
```

```
create user sa_geo_js_rails_test with password 'super_secret_password';
create user sa_geo_js_rails_dev with password 'super_secret_password';
create database geo_js_rails_development owner sa_geo_js_rails_dev;
create database geo_js_rails_test owner sa_geo_js_rails_test;
alter user sa_geo_js_rails_test CREATEDB;
alter user sa_geo_js_rails_dev CREATEDB;
```

```
rails new geo-js_rails --api
```

- You need to ensure that `rails` can connect to the database. You can do that quickly with: `bundle exec rails s`. If you get an error for `ActiveRecord Connection` or something like that then you did something wrong. Otherwise you would get the homepage.
- Now that all of that is done we can finally scaffold the databases Just gonna scaffold IP Address based on the props that the api returns. I don't know what features we would need in the future but might as well just capture all the data since it seems relevant.

```
be rails g scaffold IPAddress ip:inet \
   area_code:string \
   country:string \
   country_code:string \
   country_code3:string \
   continent_code:string \
```

```
  city:string region:string \
  latitude:string \
  longitude:string \
  accuracy:integer \
  timezone:string \
  organization_name:string \
  organization:string \
  asn:integer
```

- I decided to add an index to the `ip` because ip addresses are unique. This leads to the following migration:

```
class CreateIpAddresses < ActiveRecord::Migration[7.1]

  # => Create a unique index on the ip
  def change
    create_table :ip_addresses do |t|
      t.inet :ip
      t.string :area_code
      t.string :country
      t.string :country_code
      t.string :country_code3
      t.string :continent_code
      t.string :city
      t.string :region
      t.string :latitude
      t.string :longitude
      t.integer :accuracy
      t.string :timezone
      t.string :organization_name
      t.integer :asn
      t.string :organization

      t.timestamps
    end
  end

  # ip addresses are unique, right?
  add_index :ip_addresses, :ip, unique: true
end
```

- Then we could just `bundle exec rails db:migrate`
- We need to add `rspec` im going to use rails bytes for that: Add Rspec script.

```
rails app:template LOCATION='https://railsbytes.com/script/z0gsLX'
```

- Didn't think I would need it but actually would be nice to use `Faker` to generate random IP Addresses.

```
bundle add faker
```

- I'm gonna use the `inet` type for the ip address. In postgresql this supports: IPv4 and IPv6 hosts and networks. According to postgresql that type also offers some nice input error checking and specialized operators and functions. I don't know a ton about ip addresses but it looks the database supports a lot of formatting and display rules: IP Address Functions.
- Tests should be fairly easy to do. I'm gonna use `rspec` to test two things only:
  1. When a user hits the endpoint with an ipaddress the response has a value in the country & city fields.
  2. An entry is added to the database.
- I don't think that I need to test that IP addresses are going to be listed when the `index` is hit that should work as long as I'm testing what is above.
- I think that is pretty much it for the high-level stuff. We can implement things now. We're gonna use the `--api` switch to ensure that we don't get random stuff we don't need.
- Now we get to the tests. I think I'm just going to create a request spec. I will use ChatGPT for the initial code generation. Prompt is gonna be exactly what I described above.
- I really like concerns because of the composability. I don't like service objects the idea of a `.call` method is not something that I think really makes a lot of sense.
- I'm gonna use the `http.rb` gem for `HTTP Requests`

```
ba http


# https://get.geojs.io/v1/ip/geo.json
# https://get.geojs.io/v1/ip/geo/{ip address}.json

module GeographicIp
  extend ActiveSupport::Concern

  included do
    def get_ip_address_info(ip_address:)
      return nil unless ip_address.present?
      begin
        geo_json_uri = "https://get.geojs.io/v1/ip/geo.json"
        geo_json_uri_ip_address = "https://get.geojs.io/v1/ip/geo/#{ip_address}.json"
        response = HTTP.timeout(5).get(geo_json_uri_ip_address)
      rescue Exception => e
        return nil # No information
      else
        JSON.parse(response.body)
      end
    end
  end
end
```

- I once read somewhere that setting a timeout is good for HTTP requests. The *magic number* is anywhere between 3-5 seconds.
- I debated whether or not to add `factory_bot_rails` but I figured it would just take a couple of minutes so lets do it. I'm gonna use rails bytes again for this:

```
rails app:template LOCATION='https://railsbytes.com/script/XnJsbX'
```

- I modified the factory using `ChatGPT` to prompt it to switch to test data generation with `Faker`. We get the following factory schema from that:

```
FactoryBot.define do
  factory :ip_address do
    sequence(:id) { |n| n }
```

```
    ip { [Faker::Internet.ip_v4_address, Faker::Internet.ip_v6_address].sample }
    area_code { Faker::Address.zip_code }
    country { Faker::Address.country }
    country_code { Faker::Address.country_code }
    country_code3 { Faker::Address.country_code_long }
    continent_code { ["AF", "AN", "AS", "EU", "NA", "OC", "SA"].sample }
    city { Faker::Address.city }
    region { Faker::Address.state }
    latitude { Faker::Address.latitude }
    longitude { Faker::Address.longitude }
    sequence(:accuracy) { |n| n }
    timezone { Faker::Address.time_zone }
    organization_name { Faker::Company.name }
    sequence(:asn) { |n| n }
    organization { Faker::Company.name }
    created_at { Faker::Time.between(from: DateTime.now - 1, to: DateTime.now) }
    updated_at { Faker::Time.between(from: DateTime.now - 1, to: DateTime.now) }
  end
end
```

- The next thing we can do is seed the database with some randomly generated ip addresses. We do this by modifying the `db/seeds.rb` file.

```
100.times do
  FactoryBot.create :ip_address
end
```

- I realized that we really only need `index, show, create` actions from our `resources` macro. It's important to only create the actions that you support, otherwise, rails will use memory for those routes which aren't used and I have found it creates confusion for later maintenance, so lets be good plumbers. Open `routes.rb` and modify it to:

```
resources :ip_addresses, only: %i[create show index]
```

- At this point I want to implement the filtering operation. Since we're already persisting the information for a country and city to the database we can start by just allowing country and city params in the request to our api and select using those in the database. I realize that input for a country and city string *could* be a toss up between: `"united states of america"` vs `"usa"` or even `"us"`. I know that the `countries` gem already supports some of this interfacing so lets add it:

```
bundle add countries
```

- Then I'm just gonna support a query that follows the `ISO3166::Country.all` decodes on `alpha2` which is the country code.

```
countries = ISO3166::Country.all
country_codes = all_countries.map(&:alpha2)
```

- I'm gonna add the filtering operations in our `IpAddress` model as a *scope*, one for `country_code` and another for `city` fields:

```
scope :filter_ip_addresses_by_country_code, -> (country_code) { where(country_code: cou
scope :filter_ip_addresses_by_city, -> (city) { where(city: city) }
```

- Keep in mind that we can chain scopes. Naturally, I want to start by `country_code` and then filter on that result set by `city`:

```
filtered_ips = IpAddress.filter_ip_addresses_by_country_code("US").filter_ip_addresses_
```

- We also could have done a composite scope like:

```
class IpAddress < ApplicationRecord
  scope :filter_by_country_and_city, -> (country_code, city) do
    where(country_code: country_code, city: city)
  end
end
```

- Either way we're gonna implement that as an action in our controller. Lets say the action is `filter`. We start in the `routes` file.

```
resources :ip_addresses, only: %i[create show index] do
  collection do
    get :filter
  end
end
```

- `collection` is a way to specify a route that will act on a collection of objects. This gives us `ip_addresses#filter` in our controller to implement like so:

```
def filter
  query_by = filter_params
  @filtered_ip_addresses = IpAddress.filter_ip_address_by_country_code(query_by[:countr
  render json: @filtered_ip_addresses
end

# Also add filter params
def filter_params
  params.require(:filter_params).permit(:country_code, :city)
end
```

- Next we need to create our spec for this new `action` :

```
describe "GET /filter" do
  it "filters ip addresses as expected" do
    # Simulate the values you want to filter by
    country_code_value = 'US'
    city_value = 'New York'

    # Make the API request to filter ip addresses
    get filter_ip_addresses_path, params: { country_code: country_code_value, city: cit

    # Parse JSON response
    json_response = JSON.parse(response.body)

    # Get the count from the database directly using the scopes or ActiveRecord query
    expected_count = IpAddress.filter_ip_address_by_country_code(country_code_value).fi

    # Compare the two counts
    expect(json_response.size).to eq(expected_count)
  end
end
```

- Keep in mind that to test the filter action you need some stuff in the database. Just run:

```
RAILS_ENV=test be rake db:seed
```

- You can then launch a db console and change the `country_code_value` and `city_value` to match some records that were seeded. It works on my end.

- I've noticed that with `bootsnap` you need to clear the cache sometimes if you're getting stale files. I added a rake task for that it looks like:

```
namespace :bootsnap do
  desc "Clear the Bootsnap cache"
  task :clear_cache => :environment do
    require 'fileutils'

    # => `**` globs to any directory and `*` globs any file...
    bootsnap_cached_files = Dir.glob(File.join(Rails.root, 'tmp', 'cache', 'bootsnap',

    puts "Clearing bootsnap cache..."
    bootsnap_cached_files.each do |file|
      FileUtils.rm_rf file
    end

    puts "Bootsnap cache cleared."
  end
end
```

- Run `bundle exec rake bootsnap:clear_cache` to clear the cache and run `bundle exec rspec` again to get the fresh `spec` files.

Finally run `bundle exec rspec` in the root of your project and everything should be good.

2.0.1 Post Implementation thoughts

- I think some of the stuff in the controller could be refactored. `@ip_address` is a little confusing.
- Possibility for duplicates in `ip_addresses` table. It could be worth to hit the database before going over the network for an ip address that was queried for before