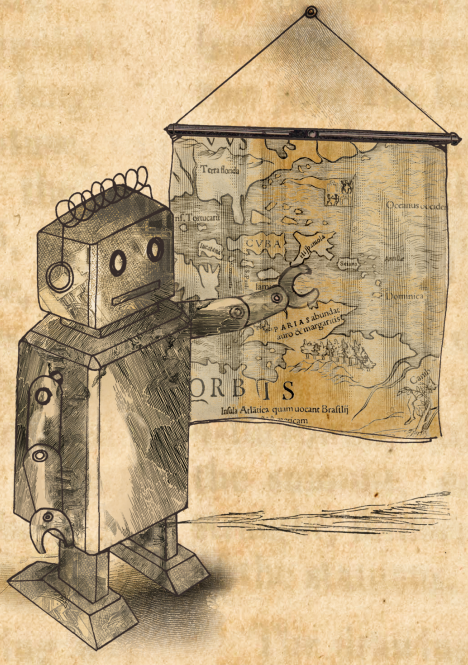
GEOCODING

ON

RAILS



Contents

Introduction

> “The world is a beautiful book, but it’s not much use if you don’t know how to read.” – Carlo Goldoni, *Pamela*

The human desire to acquire knowledge of the natural world has produced countless systems for aggregating, manipulating, and representing geospatial information. In recent years–now that we’re all generally agreed on the shape of the Earth–it has also produced a lot of software.

Working with geocoded data within the context of a web application can be tricky. While the availability of free and open-source libraries has greatly simplified the challenges of accurately geocoding data and performing meaningful analysis, there are still a plethora of decisions to be made before the first line of code is written: *What tools should we use? Where should the business logic live? What’s the best way to write a test for this?* And after the feature has been built, *Is it fast enough?*

Geocoding on Rails is a resource for developers seeking an object-oriented, test-driven approach to working with geocoded data within Rails applications. It is divided into four sections:

1. **Strategies** for selecting an external geocoding service
2. **Application Development** approaches for organizing your code as you’re developing features
3. **Improving Application Performance** with caching and other techniques
4. **Testing** techniques for your server- and client-side code

The code samples in this book come from commits in the bundled example application. The example application is a Rails app which lets users search for Starbucks locations nearby. Take a look at the README for instructions on setting it up.

This sample contains a few hand-picked sections covering various aspects of application development and performance improvements: plotting points on a map, geocoding browser requests from the server and caching results from external requests. Because these sections are pulled directly from the book, you're able to get a sense for the content, style, and delivery of the product.

If you enjoy the sample, you can get access to the entire book and sample application at:

`https://learn.thoughtbot.com/geocodingonrails`

As a purchaser of the book, you also get access to:

- Multiple formats, including HTML, PDF, EPUB, and Kindle.
- A complete example application containing code samples referenced from the book.
- The GitHub repository to receive updates as soon as they're pushed.
- GitHub issues, where you can provide feedback and tell us what you'd like to see.

Contact us

If you have any questions, or just want to get in touch, drop us a line at learn@thoughtbot.com.

Part I

Application Development

Plot Points on a Map

At this point, all the data has been geocoded; the next step is to display this information by rendering each location on a map. This allows users to understand spatial relationships between the data points.

We'll be using the Google Maps JavaScript API to display a map and plot a marker for each location. Because requests to the Google Maps API are made on the client side, there are no changes to the controller or model.

Changes to the View

Before jumping into querying the maps API, we make some decisions about how we're going to organize our client-side scripts.

First, we create a namespace for our application:

```
# app/assets/javascripts/base.coffee
@ExampleApp = {}
```

Then we create a `javascript.html.erb` partial which will be rendered in the application layout body:

```
# app/views/application/_javascript.html.erb
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>
<%= yield :javascript %>
```

```
# app/views/layouts/application.html.erb
<body>
  <div class="container">
    <%= yield %>
  </div>

  <%= render 'javascript' %>
</body>
```

With our initial setup complete, we now create some CoffeeScript classes to handle interactions with the maps API. We begin by creating a `Mapper` to display the map on the page with markers placed at the correct coordinates:

```
# app/assets/javascripts/mapper.coffee
class @ExampleApp.Mapper
  constructor: (cssSelector) ->
    @cssSelector = cssSelector
    @map = null
    @bounds = new ExampleApp.MapBounds

  addCoordinates: (latitude, longitude) ->
    if !_.isEmpty(latitude) and !_.isEmpty(longitude)
      @bounds.add(latitude, longitude)

  render: =>
    @map = new ExampleApp.Map(@cssSelector, @bounds)
    @map.build()
```

Next we create a `MapBounds` class which provides a simple interface for interacting with Google's representation of coordinates and bounds:

```
# app/assets/javascripts/map_bounds.coffee
class @ExampleApp.MapBounds
  constructor: ->
    @googleLatLngBounds = new google.maps.LatLngBounds()
    @latLngs = []

  add: (latitude, longitude) ->
    latLng = new google.maps.LatLng(latitude, longitude)
    @googleLatLngBounds.extend(latLng)
    @latLngs.push(latLng)

  getCenter: ->
    @googleLatLngBounds.getCenter()
```

We also create a `Map` class to provide a simple interface to the Google Maps JavaScript API for rendering a responsive map:

```
# app/assets/javascripts/map.coffee
class @ExampleApp.Map
  constructor: (cssSelector, bounds) ->
    @googleMap = new google.maps.Map($(cssSelector)[0], @_mapOptions())
    @bounds = bounds

    $(window).on 'resize', =>
      google.maps.event.trigger(@googleMap, 'resize')
      @_updateCenter()

  build: ->
    @_updateCenter()
    @_plotCoordinates()

  _updateCenter: ->
    @googleMap.fitBounds @bounds.googleLatLngBounds
    @googleMap.setCenter @bounds.getCenter()

  _plotCoordinates: ->
    for latLng in @bounds.latLngs
      new google.maps.Marker(position: latLng, map: @googleMap)

  _mapOptions: ->
    zoom: 13
    mapTypeId: google.maps.MapTypeId.SATELLITE
```

Finally, we add a function in the view which instantiates a `Mapper` and calls `addCoordinates()` and `render()` to display a map and plot each location on page load. We also add a `map` element to the DOM:

```
# app/views/locations/index.html.erb
<div id="map" style="height: 400px;"></div>

<ul class="locations">
  <% @locations.each do |location| %>
    <%= render location %>
  <% end %>
```

```
</ul>

<% content_for :javascript do %>
  <script type="text/javascript"
    src="//maps.googleapis.com/maps/api/js?sensor=false"></script>

  <%= javascript_tag do %>
    $(function() {
      var mapper = new ExampleApp.Mapper('#map');

      $('[data-latitude]').each(function(index, element) {
        mapper.addCoordinates(
          $(element).attr('data-latitude'),
          $(element).attr('data-longitude')
        );
      });

      mapper.render();
    });
  <% end %>
<% end %>

# app/views/locations/_location.html.erb
<%= content_tag_for :li, location,
  data: { latitude: location.latitude, longitude: location.longitude } do %>
  <header>
    <h1 data-role="name"><%= location.name %></h1>
    <% if location.respond_to?(:distance) %>
      <h2 data-role="distance"><%= location.distance.round(1) %> mi</h2>
    <% end %>
  </header>
  <section>
    <section class="location-info">
      <p data-role="address"><%= location.address %></p><br>
      <p data-role="phone-number">
        <%= link_to location.phone_number, "tel:#{location.phone_number}" %>
      </p>
    </section>
```

```
  </section>
<% end %>
```

Geocode Browser Requests on the Server Side

Once data in the application is searchable based on values other than postal codes, there are a number of usability improvements that can be made; one is pre-populating the search field with a guess at the city and state of the user.

Changes to the Controller and View

The geocoder gem extends the `request` object within Rails controllers with a new method, `#location`, which exposes information about both city and state. By creating a new class, `RequestGeocodingGatherer`, to handle calculating city and state, we're able to keep this logic out of the controller and have small classes, each with their own responsibility:

```
# app/models/request_geocoding_gatherer.rb
class RequestGeocodingGatherer
  def initialize(request)
    @request = request
  end

  def current_location
    if city && state
      [city, state].join ', '
    else
      ''
    end
  end

  private

  delegate :city, :state, to: :location
  delegate :location, to: :@request
end
```

Within the controller, we specify a `class_attribute :request_geocoding_gatherer` and assign it to our new class to retrieve the current location string from the `request`:

```
# app/controllers/locations_controller.rb
class LocationsController < ApplicationController
  class_attribute :request_geocoding_gatherer
  self.request_geocoding_gatherer = RequestGeocodingGatherer

  def index
    @current_location_by_ip = geocoded_request_information.current_location
    @locations = if search_value.present?
                   Location.near(search_value)
                 else
                   Location.all
                 end
  end

  private

  def search_value
    params[:search] && params[:search][:value]
  end

  def geocoded_request_information
    request_geocoding_gatherer.new(request)
  end
end
```

In the view, we set the search field's placeholder to `@current_location_by_ip`:

```
# app/views/locations/index.html.erb
<%= form_for :search, method: :get, url: root_path do |form| %>
  <%= form.label :value, 'Search by Location' %>
  <%= form.text_field :value, placeholder: @current_location_by_ip %>
  <%= form.submit 'Search' %>
<% end %>
```

Testing

- Geocoding with an External Service

Part II

Improving Application Performance

Cache Results from External Requests

The geocoder gem provides support for caching responses from external geocoding services by URL. When your application attempts to geocode a location that has already been geocoded, the gem will return the cached response for the request URL.

Start by defining a class to encapsulate the cache. In this example, we're using Memcache (because it works immediately with `Rails.cache`), but Redis is also supported. Geocoder requires that the cache object implement the following four methods:

1. `CacheClassName#[](key)`
2. `CacheClassName#[]=(key, value)`
3. `CacheClassName#del(key)`
4. `CacheClassName#keys`

The first three methods are invoked when setting, retrieving and deleting key-value pairs. The last method, `CacheClassName#keys`, is invoked only when clearing out the cache; in this implementation, it returns an empty array:

```
# app/models/geocoder_cache.rb
class GeocoderCache
  CACHE_KEY = 'geocoder_cache'

  def initialize(store = Rails.cache)
    @store = store
  end

  def []=(key, value)
    if value.nil?
      del key
    else
      write_to_store key, value
    end
  end
```

```
  def [](key)
    read_from_store key
  end

  def keys
    []
  end

  def del(key)
    store.delete full_key(key)
  end

  private

  def full_key(key)
    [CACHE_KEY, key].join(' ').parameterize
  end

  def read_from_store(key)
    store.read full_key(key)
  end

  def store
    @store
  end

  def write_to_store(key, value)
    store.write full_key(key), value
  end
end
```

Next, configure the cache store in an initializer:

```
# config/initializers/geocoder.rb
Geocoder.configure(cache: GeocoderCache.new)
```

Finally, ensure that you configure the `cache_store` setting within your Rails application correctly in your test environment:

```
# config/environments/test.rb
config.cache_store = :null_store
```

This disallows cached values within your test environment, which means you can be confident that any tests you write don't rely inadvertently on state from other tests.

Testing

- Testing `GeocoderCache`

Closing

Thanks for checking out the sample of *Geocoding on Rails*. If you'd like to get access to the full content, the example application, ongoing updates, and the ability to get your questions about Ruby on Rails answered by us, you can pick it up on our website:

https://learn.thoughtbot.com/geocodingonrails