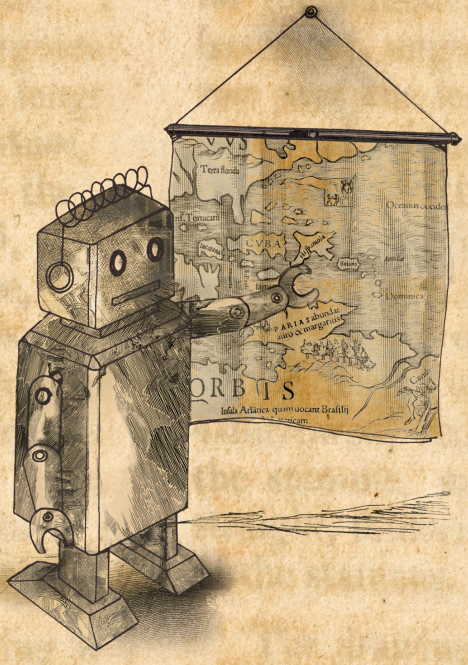
GEOCODING

ON

RAILS



Geocoding on Rails

thoughtbot Laila Winner Josh Clayton

October 2, 2013

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.

Part I

Strategies

Geocoding in Rails Applications

Geocoding is the process of obtaining coordinates (latitude and longitude) with full or partial address information. This information commonly takes the form of a postal code, full street address or the name of a point of interest, such as a library or airport. An application with geocoded data typically manages two aspects of data interaction:

1. Create data specific to a coordinate
2. Query against data near a coordinate

Within a Rails application, objects may have address information which needs to be plotted on a map or compared to other records in the database; in cases like this, the address must be geocoded to a geographic coordinate which can be used for displaying information or querying against.

There are plenty of gems that automate the process of geocoding data as it's added or updated. The geocoded object often exposes a method to be used during the geocoding process; the gem then updates the object's latitude and longitude after calculating its position. The gem also handles interactions with any external geocoding services.

Geocoding can also be performed on the client side: The W3C geolocation API is supported by most browsers, and organizations like Google and Yandex maintain JavaScript APIs for their mapping services.

Local Data

The most basic approach to integrating geocoding functionality is to maintain a local resource that maps address information to geographic coordinates.

Calculating Coordinates

The area gem relies on public domain records and does not make external requests to geocode addresses. This gem provides a simple interface for converting ZIP codes to coordinates by adding the method `#to_latlon` to `String`:

```
'02101'.to_latlng # "42.370567, -71.026964"
```

Although it's possible to use area to convert city and state to a ZIP code, such conversions are unreliable and error-prone because area operates on static data:

```
'Washington DC'.to_zip # []
'Washington, DC'.to_zip # ["20001", "20002", ...]
'Washington, D.C.'.to_zip # []
```

While the flaws in the data may be a deterrent to using gems which don't interact with an external service, geocoding with area is very fast and is sufficient if you only need to geocode US ZIP codes.

External Services

Choosing a Service

Selecting a geocoding service is best done with an estimate in mind of the daily volume of requests your application is likely to make. The free geocoding services offered by Google and Yandex are appropriate for most cases, but if their rate limits are too low for your needs, you may want to consider subscribing to a paid service.

Google and Yandex offer free services with rate limits of 2,500 and 25,000 requests per day, respectively. Client-side requests to the Google Geocoding API do not count toward the rate limit. Google Maps for Business is a paid service with a rate limit of 100,000 requests per day. Other good options for paid services are Yahoo BOSS and Geocoder.ca (US and Canada only).

While the free Google and Yandex services are robust and well-documented, open-source services are also worth considering. For example, using Nominatim or Data Science Toolkit allows your application to be independent of Google's or Yandex's terms of service.

Calculating Coordinates

Gems like geocoder provide a simple interface for querying an external service to convert any string to a coordinate. External services vary in support for points of interest (such as hotels and airports, rather than specific addresses), but will provide results for most types of queries:

```
Geocoder.coordinates('Logan Airport')
# [42.36954300000001, -71.020061]

Geocoder.coordinates('washington dc')
# [38.8951118, -77.0363658]
```

The request results will have varying levels of accuracy depending on the external service. This is significantly better than relying on local data because of the external service's ability to infer information from the string. However, geocoding with an external service is slower than geocoding locally–a single request will often take as long as 50ms to 150ms.

Web Requests

Calculating a User's Location by IP Address

Many applications ask users to provide their current location in order to perform a search for addresses nearby. However, it's often possible to retrieve this information from the browser request itself.

The geocoder gem provides a `#location` method on the `request` object which returns a location result with latitude and longitude:

```
> request.location
=> #<Geocoder::Result::Freegeoip:0x007f8ecca5d608
     @cache_hit=true,
     @data= {
       "ip"=>"199.21.87.210",
       "country_code"=>"US",
```

```
  "country_name"=>"United States",
  "region_code"=>"CA",
  "region_name"=>"California",
  "city"=>"Berkeley",
  "zipcode"=>"",
  "latitude"=>37.8716,
  "longitude"=>-122.2728,
  "metro_code"=>"807",
  "areacode"=>"510"
}>
```

Determining a user's location via IP address interacts with a different set of services than attempting to convert a physical address to a coordinate; in the example above, the geocoder gem is using the freegeoip.net service.

Calculating a User's Location from a Web Browser

Within a web browser, the W3C Geolocation API provides location data accessible with JavaScript:

```
navigator.geolocation.getCurrentPosition(successCallback, failureCallback);
```

The W3C Geolocation API is agnostic in how it calculates location. Depending on the user's device, location may be determined by GPS, inferred based on network IP addresses or triangulated based on distance from cellular towers.

Part II

Application Development

Introduction

The chapters in this section outline recommended approaches to building features that rely on the presence of geocoded data. As a starting point, let's assume we have a few thousand records in our database, each with accurate address information.

Without the ability to query our data by location, we can start by displaying addresses:

```
# app/views/locations/index.html.erb
<ul class="locations">
  <% @locations.each do |location| %>
    <li data-id="<%= location.id %>">
      <p><%= location.name %></p>
      <p>
        <%= location.street_line_1 %><br>
        <%= location.street_line_2 %><br>
        <%= location.city %>, <%= location.state %> <%= location.postal_code %>
      </p>
    </li>
  <% end %>
</ul>

# app/controllers/locations_controller.rb
class LocationsController < ApplicationController
  def index
    @locations = Location.all
  end
end

# app/models/location.rb
class Location < ActiveRecord::Base
  validates :store_number, uniqueness: true
end
```

While displaying this information is informative, it does not provide much value to users seeking to filter or visualize results by location. To facilitate meaning-

ful interactions with our data we need to add features like searching and user geolocation.

Search Data Locally

To start geocoding `Location`, we will add two previously mentioned gems: area and geocoder. Area will be used to geocode `Location` based on postal code, while geocoder will be used to do a radial search for locations within a distance.

Changes to the Controller and View

Instead of just assigning `Location.all` to `@locations`, we take into account possible search values; when a value is present, we call the method `Location.near` (provided by the geocoder gem), which adds some trigonometry to the SQL query to search within a certain radius. Without a search term, however, we continue to use `Location.all`:

```
# app/controllers/locations_controller.rb
def index
  @locations = if near_query.present?
                 Location.near(near_query)
               else
                 Location.all
               end
end
```

The method `.near` provided by geocoder is flexible; at this point, we provide coordinates calculated by the area gem (and rolled up into the `PostalCode` class). By providing coordinates, this ensures the geocoder gem does not hit any external services to calculate the center of the search, which makes for a more efficient process and supports faster page load:

```
# app/controllers/locations_controller.rb
def near_query
  PostalCode.new(search_value).coordinates
```

```
end

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


# app/models/postal_code.rb
class PostalCode
  def initialize(value)
    @value = value
  end

  def for_geocoding
    if @value.present?
      ('%05d' % @value.to_s.gsub(/\A(\d+)(-\d+)?\z/, '\1').to_i).first(5)
    end
  end

  def coordinates
    if for_geocoding && latlon = for_geocoding.to_latlon
      latlon.split(/\,/).map(&:strip).map(&:to_f)
    else
      []
    end
  end
end
```

The view changes very little, adding only a form to allow for searching:

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

Changes to the Model

The model now needs to do two things: know how to update its coordinates when the model is updated and recognize itself as a geocodable model.

To update coordinates, adding an `after_validation` callback to geocode the model is most straightforward:

```
# app/models/location.rb
class Location < ActiveRecord::Base
  validates :store_number, uniqueness: true
  geocoded_by :country_code

  after_validation :geocode_by_postal_code, if: :postal_code?

  private

  def geocode_by_postal_code
    self.latitude, self.longitude = PostalCode.new(postal_code).coordinates
  end
end
```

This callback relies on `PostalCode`, taking advantage of the area gem to convert `#postal_code` to useable coordinates.

To add the `.near` class method for searching based on location, the model needs to declare the attribute or method (passed as a symbol to `geocoded_by`) by which it can be geocoded. Because geocoding is being handled by the `PostalCode` class and not the geocoder gem, the attribute `:country_code` is perfectly acceptable for the current use case. When hitting an external service like Google, however, we'll need to change this attribute to something more specific, such as street address.

Testing

- Unit Tests

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 %>
```

Search Data Externally

While avoiding requests to third-party services results in faster geocoding and fewer dependencies, it is often inaccurate. Some postal codes cover hundreds of square miles; when plotting points on a map or performing a search, data accuracy is important. By relying on an external service to geocode data, coordinates become more accurate, searches become more helpful and maps provide greater value.

Changes to the Controller and View

Instead of converting the search string (a postal code) to coordinates within the controller, we instead pass the search string directly to `Location.near`, which will handle the geocoding. Because the geocoder gem is hitting an external service, the search string doesn't need to adhere to a specific format–the service will calculate coordinates as best it can. This removes the restriction of only searching by postal code, allowing users to search locations in a much more intuitive fashion:

```
# app/controllers/locations_controller.rb
class LocationsController < ApplicationController
  def index
    @locations = if search_value.present?
                   Location.near(search_value)
                 else
                   Location.all
                 end
  end

  private

  def search_value
```

```
    params[:search] && params[:search][:value]
  end
end
```

Changes to the Model

The model changes in two areas: `after_validation` calls the `#geocode` method (provided by the geocoder gem) and the model now considers itself geocoded by `#address` instead of `#country_code`. Every time we validate the model, geocoder makes a request to an external service, updating the model's coordinates. While naive, this approach works well and is often more accurate than geocoding by postal code.

```
# app/models/location.rb
class Location < ActiveRecord::Base
  validates :store_number, uniqueness: true
  geocoded_by :address

  after_validation :geocode

  private

  def address
    [street_line_1, street_line_2, city, state,
      postal_code, country_code].compact.join ', '
  end
end
```

Testing

- Geocoding with an External Service

Geocode Existing Data

Once a model is able to geocode itself, the next task at hand is updating existing records with coordinates. The geocoder gem provides a scope (`.not_geocoded`)

which returns all records missing latitude and longitude.

```
# lib/tasks/geocode_locations.rake
desc "Fill in coordinates for locations which haven't been geocoded"
task geocode_locations: :environment do
  Location.not_geocoded.each do |location|
    location.geocode
    location.save!
  end
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

Geocode Browser Requests on the Client Side

In the previous section we referred to the Rails `request` object to reverse geocode the user's location and pre-populate the search field with the user's city and state. As an alternative to the server-side approach, we can reverse geocode the user's location using the W3C Geolocation and Google Geocoding APIs.

Changes to the View

In the view, we add a function which calls `getLocation()` on an instance of `ExampleApp.CurrentLocation` if the search field's placeholder attribute is blank. We pass the function a callback which sets the placeholder attribute to the current city and state:

```
# app/views/locations/index.html.erb
$(function() {
  if (_.isEmpty($('#search_value').attr('placeholder'))) {
    var currentLocation = new ExampleApp.CurrentLocation();
    currentLocation.getLocation(function(location) {
      $('#search_value').attr('placeholder', location);
    })
```

```
  }
});
```

Next, we build out our `CurrentLocation` class. When a new `CurrentLocation` is instantiated, a call is made to the W3C Geolocation API. The API function `navigator.geolocation.getCurrentPosition()` requires a success callback and a failure callback; in this implementation, the success callback is a function that reverse geocodes the geographic coordinates returned. If either of the two external requests is unsuccessful, the `getLocation()` callback is executed using `CurrentLocation.DEFAULT_LOCATION`:

```
# app/assets/javascripts/current_location.coffee
class @ExampleApp.CurrentLocation
  @DEFAULT_LOCATION = 'Boston, MA'

  constructor: (deferredResolution) ->
    @deferredResolution = deferredResolution || (defer) =>
      navigator.geolocation.getCurrentPosition(
        @_reverseGeocodeLocation(defer), defer.reject
      )

  getLocation: (callback) =>
    successCallback = (value) -> callback(value)
    failureCallback = (value) -> callback(ExampleApp.CurrentLocation.DEFAULT_LOCATION)

    $.Deferred(@deferredResolution).then(successCallback, failureCallback)

  _reverseGeocodeLocation: (deferred) =>
    (geoposition) =>
      reverseGeocoder = new ExampleApp.ReverseGeocoder(
        geoposition.coords.latitude,
        geoposition.coords.longitude
      )
      reverseGeocoder.location(deferred)
```

The last step is to create the `ReverseGeocoder` to handle interactions with the external geocoding service:

```
# app/assets/javascripts/reverse_geocoder.coffee
class @ExampleApp.ReverseGeocoder
  constructor: (latitude, longitude) ->
    @latLng = new google.maps.LatLng(latitude, longitude)
    @geocoder = new google.maps.Geocoder

  location: (deferred) ->
    @geocoder.geocode { latLng: @latLng }, (response, status) =>
      if status is 'OK'
        deferred.resolve(@_locationFromResponse(response[0]))
      else
        deferred.reject()

  _locationFromResponse: (result) ->
    city = result.address_components[2].long_name
    state = result.address_components[4].short_name
    "#{city}, #{state}"
```

Testing

- CoffeeScript Unit Tests

Part III

Improving Application Performance

Introduction

As your application increases in complexity, you may see a decrease in performance. Broadly speaking, the performance of the geocoding aspects of your application can be improved in two ways: limiting the number of external requests and making database queries faster.

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`

Geocode Only When Necessary

Currently we're geocoding `Location` objects in an `after_validation` callback. This approach is less than ideal because it makes our application more likely to hit the daily rate limit of the external geocoding service. In addition, we're slowing down our application with unnecessary external requests: Geocoding with Google takes an average of 75ms. An easy way to improve improve application performance is to geocode only when the address changes.

Changes to the Model

To ensure we only geocode when the address changes, we build out `#geocoding_necessary?` and define the appropriate behavior where `set_coordinates` only runs when `#geocoding_necessary?` returns `true`:

```
# app/models/location.rb
class Location < ActiveRecord::Base
  ADDRESS_FIELDS = %w(street_line_1 street_line_2
    city state postal_code country_code).freeze

  class_attribute :geocoding_service
  self.geocoding_service = Geocoder

  validates :store_number, uniqueness: true
  geocoded_by :address

  after_validation :set_coordinates, if: :geocoding_necessary?

  def self.search_near(term)
    coordinates = geocoding_service.coordinates(term)
    near(coordinates)
  end

  def address
    address_field_values.compact.join ', '
  end

  private

  def address_field_values
    ADDRESS_FIELDS.map { |field| send field }
  end

  def address_changed?
    (changed & ADDRESS_FIELDS).any?
  end

  def geocoding_necessary?
    if new_record?
      missing_coordinates?
    else
      address_changed?
    end
```

```
  end

  def missing_coordinates?
    latitude.blank? || longitude.blank?
  end

  def set_coordinates
    self.latitude, self.longitude = geocoding_service.coordinates(address)
  end
end
```

Testing

- Testing Objects are Geocoded Only When Necessary

Speed Up Proximity Queries with PostGIS

What is PostGIS?

PostGIS is a powerful spatial database extender for PostgreSQL. Like PostgreSQL, it is free and open-source. Adding PostGIS to your database enables persistence of geographic data and makes it possible to retrieve the data with spatial queries using PostGIS functions. While an exhaustive discussion of PostGIS is outside the scope of this book, its utility as a tool for speeding up database queries makes it relevant to include some notes here on its use.

Why is PostGIS Faster?

PostGIS allows geocoded data to be persisted as points on a plane. Proximity queries using PostGIS are less expensive than non-spatial queries because locations, represented as geographic points, can be compared using the Pythagorean theorem. The geocoder gem's `.near` method, by contrast, compares geographic coordinates on the fly, using the haversine formula.

How Do I Use PostGIS?

While there are currently few comprehensive and up-to-date resources for using PostGIS with Rails applications in multiple environments, the challenges of installing and configuring PostGIS are worth tackling if significant improvements in application performance may be gained.

Using PostGIS with Rails requires installing the ActiveRecord PostGIS Adapter and RGeo gems. To learn more about PostGIS, consider purchasing a copy of PostGIS in Action. For an example of how to configure your Rails application for use with PostGIS, see Using PostGIS in Your Rails Application.

Part IV

Testing

Testing a Rails Application with Geocoded Data

While testing a normal Rails application can be tough, introducing geocoding and determining how and when to test various aspects of it may be downright daunting. Let's break down the various aspects of testing this Rails application as we build out functionality to shed a bit more light on this subject.

Acceptance Tests

The acceptance tests of our app focus on validating core features such as displaying location results and a functioning geospatial search. The entirety of browser interaction is handled through a page object, `LocationsOnPage`, which exposes methods for interacting with the application and asserting against information rendered:

```
# spec/support/features/locations_on_page.rb
class LocationsOnPage
  ELEMENT_ID_REGEX = /_(\d+)/

  include Capybara::DSL
  include Rails.application.routes.url_helpers

  def initialize
    visit root_path
  end

  def search(value)
    fill_in 'Search by Location', with: value
    click_on 'Search'
  end

  def suggested_search_value
    field_labeled('Search by Location')['placeholder']
  end

  def include?(location)
```

```
    locations.include? location
  end

  private

  def locations
    Location.where(id: location_ids)
  end

  def locations_element
    find('.locations')
  end

  def location_elements
    locations_element.all('li')
  end

  def location_ids
    location_elements.map { |node| node['id'][ELEMENT_ID_REGEX, 1] }
  end
end
```

Let's look at the very first acceptance test, which simply verifies that locations are rendered correctly when no search is applied:

```
# spec/features/guest_views_all_locations_spec.rb
require 'spec_helper'

feature 'Guest views all locations' do
  scenario 'each location is displayed with the correct information' do
    stub_geocoding_request '12 Winter St., Boston, MA, 02111, US',
      42.35548199999999, -71.0608386
    stub_geocoding_request '36 2nd St., San Francisco, CA, 94105, US',
      37.788587, -122.400958

    boston_location = create(:location, :in_boston)
    san_francisco_location = create(:location, :in_san_francisco)
```

```
    locations = LocationsOnPage.new

    expect(locations).to include(boston_location)
    expect(locations).to include(san_francisco_location)
  end
end
```

In this spec, we create two records and ensure both are displayed. This test doesn't actually verify whether any geocoding is taking place and likely falls into the category of a smoke test.

Next, let's look at the test which ensures that searching works correctly:

```
# spec/features/guest_searches_by_postal_code_spec.rb
require 'spec_helper'

feature 'Guest searches by postal code' do
  scenario 'only displays locations within the search radius' do
    stub_geocoding_request '12 Winter St., Boston, MA, 02111, US', '02111',
      42.35548199999999, -71.0608386
    stub_geocoding_request '36 2nd St., San Francisco, CA, 94105, US', '94105',
      37.788587, -122.400958

    boston_location = create(:location, :in_boston)
    san_francisco_location = create(:location, :in_san_francisco)

    locations = LocationsOnPage.new
    locations.search boston_location.postal_code

    expect(locations).to include(boston_location)
    expect(locations).not_to include(san_francisco_location)

    locations.search san_francisco_location.postal_code

    expect(locations).not_to include(boston_location)
    expect(locations).to include(san_francisco_location)
  end
end
```

This spec creates two locations and asserts searching by the postal code of each only returns the closest location. This ensures we create records, geocoding them correctly, and filter locations given a search term.

Finally, we verify we're suggesting the correct location (with the placeholder attribute on our `<input type="text">`) based on IP retrieval. Within `LocationsController`, we already exposed a `class_attribute :request_geocoding_gatherer`, allowing us to swap out the `RequestGeocodingGatherer` with a fake object returning a known value (`'New York, NY'`):

```
# spec/features/guest_receives_suggestion_for_search_value_spec.rb
require 'spec_helper'

feature 'Guest receives suggestion for search value' do
  scenario 'only displays locations within the search radius' do
    FakeRequestGeocodingGatherer = Struct.new(:request) do
      def current_location; 'New York, NY'; end
    end
    LocationsController.request_geocoding_gatherer = FakeRequestGeocodingGatherer
    locations = LocationsOnPage.new

    expect(locations.suggested_search_value).to eq 'New York, NY'
  end
end
```

As with any class attribute, we must reset its value after each test:

```
# spec/support/request_geocoding_gatherer.rb
RSpec.configure do |config|
  config.around do |example|
    cached_request_geocoding_gatherer = LocationsController.request_geocoding_gatherer
    LocationsController.request_geocoding_gatherer = NullRequestGeocodingGatherer

    example.run

    LocationsController.request_geocoding_gatherer = cached_request_geocoding_gatherer
  end
end
```

Unit Tests

In this section, we'll review the techniques we employ in our unit tests throughout the stages of application development.

Writing unit tests for our models is initially straightforward. Let's start with `PostalCode`, the object responsible for calculating coordinates given a postal code:

```
# spec/models/postal_code_spec.rb
describe PostalCode, '#for_geocoding' do
  it 'returns a five-digit code' do
    expect(postal_code_for_geocoding('123456')).to eq '12345'
  end

  it 'pads results' do
    expect(postal_code_for_geocoding('1234')).to eq '01234'
  end

  it 'handles integer values' do
    expect(postal_code_for_geocoding(1234)).to eq '01234'
  end

  it 'handles ZIP+4 codes' do
    expect(postal_code_for_geocoding('12345-6789')).to eq '12345'
  end

  it 'returns nil with a nil value' do
    expect(postal_code_for_geocoding(nil)).to be_nil
  end

  it 'returns nil with a blank value' do
    expect(postal_code_for_geocoding('')).to be_nil
  end

  def postal_code_for_geocoding(value)
    PostalCode.new(value).for_geocoding
  end
```

```
end
```

These tests cover the base cases for various types of input: `nil`, `''`, integers, strings without padding and strings longer than five characters:

```
# app/models/postal_code.rb
def for_geocoding
  if @value.present?
    ('%05d' % @value.to_s.gsub(/\A(\d+)(-\d+)?\z/, '\1').to_i).first(5)
  end
end
```

Next up is ensuring that `PostalCode#coordinates` works as expected:

```
# spec/models/postal_code_spec.rb
describe PostalCode, '#coordinates' do
  it 'uses the geocoding value to calculate' do
    expect(PostalCode.new('02115').coordinates).to eq [
      '02115'.to_lat.to_f,
      '02115'.to_lon.to_f
    ]
  end

  it 'handles postal codes which cannot be converted to coordinates' do
    expect(PostalCode.new('12000').coordinates).to eq []
  end

  it 'handles nil' do
    expect(PostalCode.new(nil).coordinates).to eq []
  end
end

# app/models/postal_code.rb
def coordinates
  if for_geocoding && latlon = for_geocoding.to_latlon
    latlon.split(/\,/).map(&:strip).map(&:to_f)
  else
```

```
    []
  end
end
```

Geocoding with an External Service

The next step is introducing geocoding with an external service, which we do with the geocoder gem. Geocoder provides support for stubbing geocoding requests with its `:test` lookup.

First, we write our test to determine what to stub:

```
# spec/models/location_spec.rb
describe Location, '#valid?' do
  it 'geocodes with Geocoder' do
    location = Location.new(street_line_1: 'Undefined address')
    location.valid?

    expect(location.latitude).to eq geocoder_stub('nonexistent').latitude
    expect(location.longitude).to eq geocoder_stub('nonexistent').longitude
  end
end
```

Second, we define `GeocoderStub` to make the geocoder test stubs easier to interact with:

```
# spec/support/geocoder_stub.rb
module GeocoderStub
  def geocoder_stub(key)
    result_hash = Geocoder::Lookup::Test.read_stub(key).first
    OpenStruct.new(result_hash)
  end
end
```

Third, we add a stub to a `geocoder.rb` support file:

```
# spec/support/geocoder.rb
Geocoder.configure(:lookup => :test)

Geocoder::Lookup::Test.set_default_stub [{
  latitude: 12,
  longitude: 34,
}]
```

Finally, we include `GeocoderStub` in our spec helper:

```
# spec/spec_helper.rb
RSpec.configure do |config|
  config.use_transactional_fixtures = false
  config.infer_base_class_for_anonymous_controllers = false
  config.order = "random"
  config.include FactoryGirl::Syntax::Methods
  config.include GeocoderStub
end
```

When we write more complex tests, we'll need to add a stub that's specific to a location. For example:

```
# spec/support/geocoder.rb
Geocoder::Lookup::Test.add_stub '12 Winter St., Boston, MA, 02111, US', [{
  'latitude' => 42.35548199999999,
  'longitude' => -71.0608386,
}]
```

Similarly, it's possible to stub geocoding based on IP addresses:

```
# spec/support/geocoder.rb
Geocoder.configure(:lookup => :test, ip_lookup: :test)

# spec/support/geocoder.rb
Geocoder::Lookup::Test.add_stub '555.555.1.1', [{
  'ip' => '555.555.1.1',
  'city' => 'New York',
  'state' => 'NY',
}]
```

Testing `GeocoderCache`

Testing `GeocoderCache` requires that we stub `Rails.cache` to return a cache object:

```
# spec/models/geocoder_cache_spec.rb
describe GeocoderCache do
  before do
    Rails.stub(:cache).and_return ActiveSupport::Cache.lookup_store(:memory_store)
  end
```

With `Rails.cache` stubbed, we can test assigning, retrieving and deleting cache keys:

```
# spec/models/geocoder_cache_spec.rb
it 'allows for cache assignment and retrieval' do
  subject['Boston, MA'] = [22.0, 22.0]
  expect(subject['Boston, MA']).to eq [22.0, 22.0]

  subject['New York, NY'] = [-10.0, -5.0]
  expect(subject['New York, NY']).to eq [-10.0, -5.0]
end

it 'allows keys to be deleted' do
  subject['Boston, MA'] = [22.0, 22.0]
  subject.del('Boston, MA')
  expect(subject['Boston, MA']).to be_nil
end
```

Testing to Ensure Objects are Geocoded Only When Necessary

Writing tests to ensure our objects are only geocoded when the address is updated indicates that we need to do some refactoring. In our unit test for `Location`, we want to be able to spy on the object receiving the method handling geocoding. Currently, the object which receives `geocode`–the `Location` instance–is the system under test:

```
# app/models/location.rb
class Location < ActiveRecord::Base
  validates :store_number, uniqueness: true
  geocoded_by :address

  after_validation :geocode
```

We'll start by writing a test to help drive our approach to refactoring. In this test, we rely on an assignable class attribute, `geocoding_service`, which will handle the entirety of the geocoding. Assigning an object to this attribute allows us to inject the dependency in various situations; in this case, we'll inject a `double` within the spec to grant us more control over the resulting `coordinates`:

```
# spec/models/location_spec.rb
it 'does not geocode when address does not change' do
  location = create(:location, :in_boston)
  Location.geocoding_service = double('geocoding service', coordinates: nil)

  location.valid?

  expect(Location.geocoding_service).not_to have_received(:coordinates)
end
```

To make the test pass, we define a `geocoding_service` class attribute on `Location`:

```
# app/models/location.rb
class Location < ActiveRecord::Base
  ADDRESS_FIELDS = %w(street_line_1 street_line_2
    city state postal_code country_code).freeze

  class_attribute :geocoding_service
  self.geocoding_service = Geocoder
```

We also change our `after_validation` to `:set_coordinates` so we can call `coordinates` (which `Geocoder` already defines) on `geocoding_service`:

```
# app/models/location.rb
after_validation :set_coordinates, if: :geocoding_necessary?
```

```
# app/models/location.rb
def set_coordinates
  self.latitude, self.longitude = geocoding_service.coordinates(address)
end
```

Finally, we reset the `geocoding_service` class attribute after each test, just as we did for `LocationsController.request_geocoding_gatherer`:

```
# spec/support/geocoding_service.rb
RSpec.configure do |config|
  config.around do |example|
    cached_geocoding_service = Location.geocoding_service
    example.run
    Location.geocoding_service = cached_geocoding_service
  end
end
```

Decoupling Our Application From the Geocoding Service Entirely

With `Location` allowing any object be assigned to the `geocoding_service` class attribute, we're able to do a significantly larger refactor, wherein we inject a `FakeGeocoder` for every test performing geocoding. There are many benefits to this approach:

1. It allows us to remove `spec/support/geocoder.rb`: All Geocoder `add_stub`s effectively introduce mystery guests.
2. It allows us to be explicit about how each geocoding request works per test: We can now choose exactly how the geocoder used by our code responds.
3. It provides a clear seam because we never refer to `Geocoder` explicitly: We can swap out `Geocoder` entirely or introduce an adapter to another geocoding library with very little work.

We start by removing `geocoder.rb` and rewriting our test to use a helper we define, `stub_geocoding_request`:

```
# spec/models/location_spec.rb
context 'when updating location address' do
  it 'geocodes location' do
    stub_geocoding_request '45 Winter St., Boston, MA, 02111, US', 42, -75
    stub_geocoding_request '12 Winter St., Boston, MA, 02111, US', 45, -70

    location = create(:location, :in_boston, street_line_1: '45 Winter St.')
    location.street_line_1 = '12 Winter St.'
    location.valid?

    expect(location.latitude).to eq 45
    expect(location.longitude).to eq -70
  end
end
```

Next, we define `GeocodingRequestStub` (the module which contains the new `stub_geocoding_request` method) and include it in our spec helper:

```
# spec/support/geocoding_request_stub.rb
module GeocodingRequestStub
  def stub_geocoding_request(*strings, latitude, longitude)
    strings.each do |string|
      FakeGeocoder[string] = [latitude, longitude]
    end
    Location.geocoding_service = FakeGeocoder
  end
end

# spec/spec_helper.rb
RSpec.configure do |config|
  config.include GeocodingRequestStub
end
```

`stub_geocoding_request` allows for mapping any number of strings (values to be geocoded) to a specific coordinate. Iterating over the list of `strings`, we use each string as a key within our new `FakeGeocoder`. `FakeGeocoder` acts as a dictionary object, mapping keys (strings to geocode) to values (a specific coordinate).

We test-drive development of `FakeGeocoder`, ensuring it allows for assignment (`FakeGeocoder.[]=(key, value)`) and retrieval exactly as we're using it throughout the existing system (`FakeGeocoder.coordinates(key)`). To safeguard against typos on our end, any attempt to geocode an undefined value raises an exception:

```
# spec/lib/fake_geocoder_spec.rb
require 'spec_helper'

describe FakeGeocoder do
  it 'allows for setting and retrieving geocoded values' do
    FakeGeocoder['search string'] = [12, 34]
    expect(FakeGeocoder.coordinates('search string')).to eq [12, 34]
  end

  it 'raises when trying to retrieve a nonexistent value' do
    expect do
      FakeGeocoder.coordinates('search string')
    end.to raise_error /search string/
  end
end
```

The implementation of `FakeGeocoder` is straightforward; the only method we don't test directly is `FakeGeocoder.clear`, which needs to be run before each test because the data is stored at a class level:

```
# lib/fake_geocoder.rb
class FakeGeocoder
  def self.[]=(key, value)
    @values[key] = value
  end

  def self.coordinates(key)
    @values.fetch(key)
  end

  def self.clear
    @values = {}
```

```
  end
end

# spec/support/geocoding_request_stub.rb
RSpec.configure do |config|
  config.before do
    FakeGeocoder.clear
  end
end
```

We make one additional change to `Location`: We define our own method, `search_near`, which uses the `geocoding_service` to calculate coordinates and pass them to the `near` method defined by the geocoder gem:

```
# app/models/location.rb
def self.search_near(term)
  coordinates = geocoding_service.coordinates(term)
  near(coordinates)
end
```

When provided a coordinate, the `near` scope provided by the geocoder gem does not geocode the value because the work has been done already; this ensures all geocoding logic is managed by `Location.geocoding_service`.

Lastly, `LocationsController` needs to take advantage of our new scope:

```
# app/controllers/locations_controller.rb
def index
  @current_location_by_ip = geocoded_request_information.current_location
  @locations = if search_value.present?
                 Location.search_near(search_value)
               else
                 Location.all
               end
end
```

This decoupling from the geocoder gem is significant; instead of relying on the geocoder gem throughout the application and its stubs in the test suite, we

instead rely on a simple interface, `coordinates(value)` and `[]=(key, value)`, to handle the entirety of our geocoding needs.

Ensuring No External Requests are Made during Geocoding

With geocoding completely handled by `FakeGeocoder`, we can add the WebMock gem to verify the application is making no external requests.

First, add the gem to the `Gemfile`:

```
# Gemfile
group :test do
  gem 'capybara', '~> 2.1.0'
  gem 'database_cleaner', '~> 1.0.1'
  gem 'factory_girl_rails', '~> 4.1.0'
  gem 'poltergeist', '~> 1.1'
  gem 'shoulda-matchers', '~> 2.2.0'
  gem 'webmock', require: false
end
```

Next, disable all network interaction with `WebMock.disable_net_connect!`:

```
# spec/spec_helper.rb
# This file is copied to spec/ when you run 'rails generate rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'rspec/autorun'

require 'webmock/rspec'
WebMock.disable_net_connect!

Dir[Rails.root.join("spec/support/**/*.rb")].each { |f| require f }
```

A green test suite verifies no external geocoding requests are made.

CoffeeScript Unit Tests

We'll use Konacha to write unit tests for our CoffeeScript. Konacha relies on the Mocha test framework and Chai assertion library for executing the tests and Poltergeist to run the tests in-memory within a rake task.

First, we add Konacha and Poltergeist to the Gemfile:

```
# Gemfile
group :development, :test do
  gem 'konacha'
  gem 'rspec-rails', '~> 2.14.0'
end

group :test do
  gem 'capybara', '~> 2.1.0'
  gem 'database_cleaner', '~> 1.0.1'
  gem 'factory_girl_rails', '~> 4.1.0'
  gem 'poltergeist', '~> 1.1'
  gem 'shoulda-matchers', '~> 2.2.0'
end
```

We make sure the `rake` command runs our JavaScript tests as well as our RSpec tests by adding the `konacha:run` task to the Rakefile:

```
# Rakefile
task default: ['konacha:run']
```

We configure Konacha to use Poltergeist in an initializer:

```
# config/initializers/konacha.rb
if defined?(Konacha)
  Konacha.configure do |config|
    require 'capybara/poltergeist'
    config.driver = :poltergeist
  end
end
```

We create a spec helper and include `application.js`:

```
# spec/javascripts/spec_helper.js.coffee
#= require application
```

Testing `ExampleApp.CurrentLocation`

Our first test ensures that `CurrentLocation#getLocation` returns a location upon successful resolution:

```
# spec/javascripts/current_location_spec.coffee
#= require spec_helper

describe 'CurrentLocation#getLocation', ->
  describe 'when the deferred object resolves', ->
    it 'returns the location', ->
      resolution = (defer) -> defer.resolve('Boston')

      currentLocation = new ExampleApp.CurrentLocation(resolution)
      currentLocation.getLocation (result) ->
        expect(result).to.equal('Boston')
```

Next, we assert a default location is returned if the resolution is rejected:

```
# spec/javascripts/current_location_spec.coffee
describe 'when the deferred object is rejected', ->
  it 'returns a default location', ->
    resolution = (defer) -> defer.reject()

    currentLocation = new ExampleApp.CurrentLocation(resolution)
    currentLocation.getLocation (result) ->
      expect(result).to.equal(ExampleApp.CurrentLocation.DEFAULT_LOCATION)
```

Finally, we ensure that `CurrentLocation::DEFAULT_LOCATION` returns the expected value:

```
# spec/javascripts/current_location_spec.coffee
describe 'CurrentLocation::DEFAULT_LOCATION', ->
  it 'returns "Boston, MA"', ->
    expect(ExampleApp.CurrentLocation.DEFAULT_LOCATION).to.equal 'Boston, MA'
```

Testing `ExampleApp.ReverseGeocoder`

To test `ReverseGeocoder#location`, we'll need to stub requests to the external geocoding service. First, we confirm that the success callback is executed if reverse geocoding is successful:

```
# spec/javascripts/reverse_geocoder_spec.js.coffee
#= require spec_helper

describe 'ReverseGeocoder#location', ->
  context 'when reverse geocoding is successful', ->
    it 'reverse geocodes coordinates', (done) ->
      ExampleApp.TestSupport.stubSuccessfulGoogleResponse 'San Francisco', 'CA'

      buildGeocoderWithCallback success: (result) ->
        expect(result).to.equal 'San Francisco, CA'
        done()
```

We define `ExampleApp.TestSupport.stubSuccessfulGoogleResponse` in our spec helper:

```
# spec/javascripts/spec_helper.js.coffee
geocodeResult = (city, state) ->
  [
    address_components: [
      null,
      null,
      long_name: city,
      null,
      short_name: state
    ]
```

```
  ]

ExampleApp.TestSupport =
  stubSuccessfulGoogleResponse: (city, state) ->
    window.google =
      maps:
        LatLng: (latitude, longitude) ->
          'latlng'

        Geocoder: ->
          geocode: (latLng, callback) ->
            callback(geocodeResult(city, state), 'OK')
```

We define the `buildGeocoderWithCallback` helper function which returns a jQuery Deferred object with the provided callbacks configured correctly:

```
# spec/javascripts/reverse_geocoder_spec.js.coffee
buildGeocoderWithCallback = (options) ->
  nullCallback = (result) ->
  successCallback = options.success || nullCallback
  failureCallback = options.failure || nullCallback

  reverseGeocoder = new ExampleApp.ReverseGeocoder(12, 34)

  $.Deferred(
    (defer) -> reverseGeocoder.location(defer)
  ).then(successCallback, failureCallback)
```

With the assertions complete when testing a successful resolution, we can now verify that `ReverseGeocoder#location` executes the failure callback when reverse geocoding is unsuccessful:

```
# spec/javascripts/reverse_geocoder_spec.js.coffee
context 'when reverse geocoding is unsuccessful', ->
  it 'does not return a value', (done) ->
    ExampleApp.TestSupport.stubUnsuccessfulGoogleResponse()
```

```
    buildGeocoderWithCallback failure: (result) ->
      expect(result).to.be.undefined
      done()

# spec/javascripts/spec_helper.js.coffee
stubUnsuccessfulGoogleResponse: ->
  window.google =
    maps:
      LatLng: (latitude, longitude) ->
        'latlng'

      Geocoder: ->
        geocode: (latLng, callback) ->
          callback(null, 'BAD')
```

Validating Reverse Geocoding in the Browser

In addition to writing unit tests for `CurrentLocation` and `ReverseGeocoder`, it's a good idea to confirm our application is behaving as expected by viewing it in the browser. We use ngrok to expose our local server to the Internet, allowing our application to access the W3C geolocation and Google Geocoding APIs in the development environment.

Testing the Google Map

We avoid testing the Google Map generated by our `ExampleApp.Map` class primarily because there's no good way to make assertions against the Google Maps JavaScript API without the tests being very brittle. Instead, we verify behavior by viewing the application in a web browser and interacting with the map.

Part V

Appendices

Gems

The RubyGem ecosystem is a great place to find existing solutions to geocoding in Rails applications. Here are a few gems which make geocoding easier within an ORM context.

Geocoder

Geocoder is a gem that touts itself as the "Complete Ruby geocoding solution." It supports geocoding, reverse geocoding and distance queries. It works well with most Ruby ORMs and is under active development.

GeoKit

GeoKit provides a similar feature set to geocoder; however, it is not currently in active development.

Graticule

Graticule allows geocoding with Google, Yahoo and most other geocoding services, and can also be used as a command line tool. It is often used in conjunction with the acts as geocodable gem, which provides hooks into `ActiveRecord` and allows for distance queries.

Area

Area uses public domain data to convert cities to ZIP codes to coordinates; this allows independence from reliance on an external service.

GeoIP

GeoIP is a gem which searches the MaxMind GeoIP database for a host or IP address and returns location information (including coordinates). MaxMind provides free copies of its data as well as a subscription service.

Using PostGIS with Rails and Heroku

Using PostGIS in Your Rails Application

First, make sure you have PostgreSQL 9.1 or higher installed. Then install PostGIS 2.0:

OS X

The PostGIS website recommends using Postgres.app to install PostGIS. Alternatively, you can use homebrew:

```
$ brew install postgis
```

Arch Linux

```
$ sudo pacman -S postgis
```

You can find more resources for installing PostGIS on the PostGIS website.

After installing PostGIS on your laptop, follow the steps below to configure your Rails application.

Create PostGIS extension

If you haven't created your local database yet, you can simply install the ActiveRecord PostGIS Adapter gem and add `postgis_extension: true` to your `database.yml` per the instructions below. ActiveRecord PostGIS Adapter will create the extension when `rake db:create` is run.

If you've already created your database, run the following commands to install the PostGIS extension:

```
$ cd my_application
$ rails dbconsole
=# CREATE EXTENSION postgis;
...
```

Confirm that extension was created

```
=# SELECT POSTGIS_FULL_VERSION();
   NOTICE:  Function postgis_topology_scripts_installed() not found. Is topology
    support enabled and topology.sql installed?
    postgis_full_version
   -----------------------------------------------------------------------------
   POSTGIS="2.0.3 r11128" GEOS="3.3.8-CAPI-1.7.8" PROJ="Rel. 4.8.0, 6 March 2012"
   GDAL="GDAL 1.9.2, released 2012/10/08" LIBXML="2.7.8" LIBJSON="UNKNOWN" RASTER
    (raster lib from "2.0.2 r10789" need upgrade)
    (1 row)

=#
=# \quit
```

Add ActiveRecord PostGIS Adapter to Gemfile

```
# Gemfile
gem 'activerecord-postgis-adapter'
```

Configure your local databases

Configure your test and development databases for use with the ActiveRecord PostGIS adapter. Be sure to set the `adapter` to `postgis` and `postgis_extension` to `true`. Setting `postgis_extension` to `true` will ensure that the PostGIS extension is created when the database is created.

Also note that the test database `schema_search_path` should be set to `public`. This ensures that the PostGIS table `spatial_ref_sys` will be loaded when you prepare your test database. If `schema_search_path` is set to `public, postgis`, PostGIS tables will not be made available:

```
# config/database.yml
development:
  <<: *common
  adapter: postgis
  encoding: unicode
```

```
  postgis_extension: true
  schema_search_path: public, postgis
  pool: 5
  database: <database_name>

test: &test
  <<: *common
  adapter: postgis
  postgis_extension: true
  schema_search_path: public
  encoding: unicode
  database: <database_name>
```

Update your DatabaseCleaner strategy

Ensure that your DatabaseCleaner strategy does not remove the PostGIS `spatial_ref_sys` table before or between tests:

```
# spec/support/database_cleaner.rb
RSpec.configure do |config|
  config.before(:suite) do
    DatabaseCleaner.clean_with :truncation, { except: %w[spatial_ref_sys] }
  end

  config.before(:each, js: true) do
    DatabaseCleaner.strategy = :truncation, { except: %w[spatial_ref_sys] }
  end
end
```

Setting Up Continuous Integration with PostGIS

After installing PostGIS locally and adding the PostGIS extension to your local databases, you should make sure your continuous integration service is configured for PostGIS.

Configure database loading on Tddium

Create a worker hook in `tddium.yml` to ensure that the Tddium database is PostGIS-enabled:

```
# config/tddium.yml
:tddium:
  :postgresql:
    :adapter: postgis
    :postgis_extension: true
    :schema_search_path: public
  :hooks:
    :worker:
      createdb $TDDIUM_DB_NAME;
      psql $TDDIUM_DB_NAME -c 'CREATE EXTENSION postgis;';
      bundle exec rake db:migrate
```

Setting Up PostGIS for Heroku

Heroku support for PostGIS is currently available in beta for production-tier databases.

Configuring your Heroku database

Check to make sure your primary database is a production-tier database

Heroku's least expensive production-tier option is the Crane database, priced at $50/month:

```
$ heroku pg:info --app <your-app>

=== HEROKU_POSTGRESQL_CHARTREUSE_URL (DATABASE_URL)
Plan:        Crane
Status:      available
Data Size:   1.00 GB
Tables:      2
```

```
PG Version:  9.2.4
Connections: 5
Fork/Follow: Available
Created:     2013-07-01 09:54 UTC
Maintenance: not required
```

If your plan is a starter-tier plan (Dev or Basic), be sure to upgrade before setting up the PostGIS add-on.

Create extension

```
$ heroku pg:psql --app <your-app>
=# CREATE EXTENSION postgis;
...
```

Confirm that extension was created

```
=# SELECT POSTGIS_FULL_VERSION();
   NOTICE:  Function postgis_topology_scripts_installed() not found. Is topology
    support enabled and topology.sql installed?
    postgis_full_version
   -----------------------------------------------------------------------------
   POSTGIS="2.0.3 r11128" GEOS="3.3.8-CAPI-1.7.8" PROJ="Rel. 4.8.0, 6 March 2012"
   GDAL="GDAL 1.9.2, released 2012/10/08" LIBXML="2.7.8" LIBJSON="UNKNOWN" RASTER
    (raster lib from "2.0.2 r10789" need upgrade)
    (1 row)

=#
...
=# \quit
```

Create an initializer to set the database adapter on Heroku

Set the Heroku database adapter and schema search path using an initializer:

```
# config/initializers/database.rb
Rails.application.config.after_initialize do
  ActiveRecord::Base.connection_pool.disconnect!

  ActiveSupport.on_load(:active_record) do
    config = Rails.application.config.database_configuration[Rails.env]
    config['adapter'] = 'postgis'
    config['schema_search_path'] = 'public, postgis'
    ActiveRecord::Base.establish_connection(config)
  end
end
```

Push to Heroku

```
$ git push staging master
```