Episode 273

Geocoder

If you have to work with Geographic data in your Rails applications the Geocoder gem[1] makes this much simpler. It can convert place names to coordinates and vice-versa and even convert IP addresses to street addresses. It can also find nearby locations with their distance and bearing and has many other useful features.

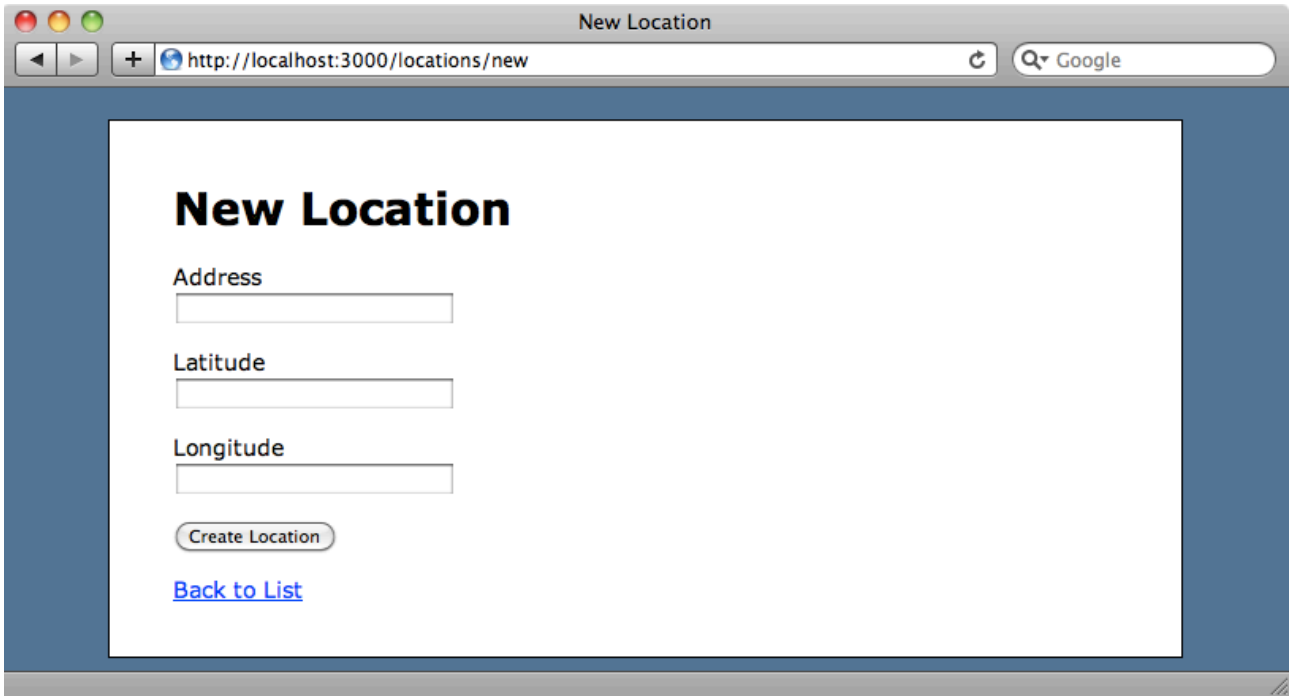
In this episode we'll create an application called Siteseer which will allow people to recommend tourist locations. Once we've created the application the next thing to do is to create a `Location` scaffold with `address`, `latitude` and `longitude` fields.

```
$ rails g nifty:scaffold location address:string latitude:float ↵
  longitude:float
```

The naming of the `latitude` and `longitude` fields is important as Geocoder will use these to store the location coordinates, although the default field names can be overridden. Both fields need to be `floats`. We'll need to migrate the database to create the new `locations` table.

```
$ rake db:migrate
```

Now we have the scaffolding set up we can create new locations but we have to enter the latitude and longitude manually. We'll update our application now so that these fields are filled in automatically by Geocoder.



[1] http://rubygeocoder.com/

Installing and Using Geocoder

Geocoder is installed in the usual way. First we add a reference to the gem in the `Gemfile` and then run `bundle` to install it.

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.9'
gem 'sqlite3'
gem 'nifty-generators'

gem 'geocoder'
```

Next we'll modify our `Location` model and add a call to `geocoded_by` to specify the attribute that we want Geocode to convert, in this case the `address`.

/app/models/location.rb

```
class Location < ActiveRecord::Base
  attr_accessible :address, :latitude, :longitude
  geocoded_by :address
end
```

If our address is stored over several fields we can specify a method here instead and write a method that returns the complete address.

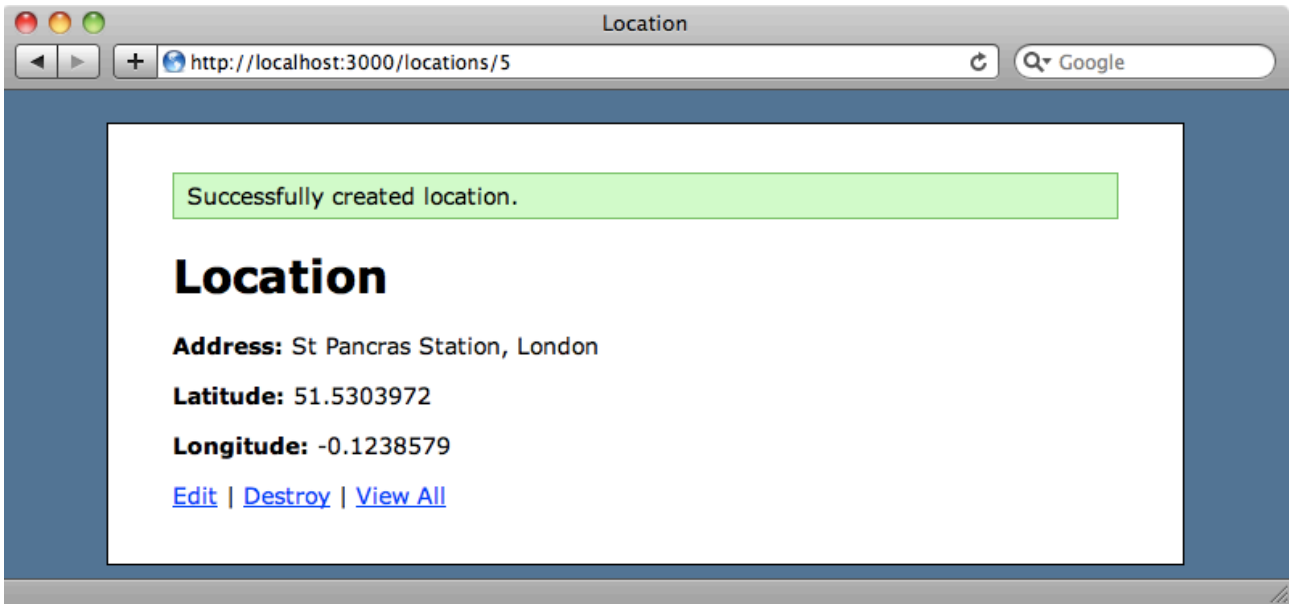
When a `location` is created or updated we need to call the geocode method that performs the geocoding. It's usual to use the `after_validation` callback to do this.

/app/models/location.rb

```
class Location < ActiveRecord::Base
  attr_accessible :address, :latitude, :longitude
  geocoded_by :address
  after_validation :geocode
end
```

The geocode method sends a request to an external API, by default the Google Maps API[2]. As it calls an external service it would be better to move this call into a background process. We won't do that here but to find out how it's down take a look at the recent episode on Resque[watch[3], read[4]].

We can start up our application now and try Geocoder out. When we enter a new location, say "St Pancras Station, London", without a latitude or longitude, Geocoder will fetch that data from the external API and add it to the location.



The external API will be called every time we update a location but this should only happen when the `address` field changes. We can make a simple change to the `after_validation` callback to implement this.

/app/models/location.rb

```
class Location < ActiveRecord::Base
  attr_accessible :address, :latitude, :longitude
  geocoded_by :address
  after_validation :geocode, :if => :address_changed?
end
```

---

[2] http://code.google.com/apis/maps/index.html

[3] http://railscasts.com/episodes/271-resque

[4] http://asciicasts.com/episodes/271-resque

This will use dirty tracking to determine if the address has changed and only call the API if it has.

We can also use a `reverse_geocoded_by` method to convert a latitude and longitude into an address. This works in a similar way to the `geocoded_by` method and we won't be using it in our application.

Getting nearby locations

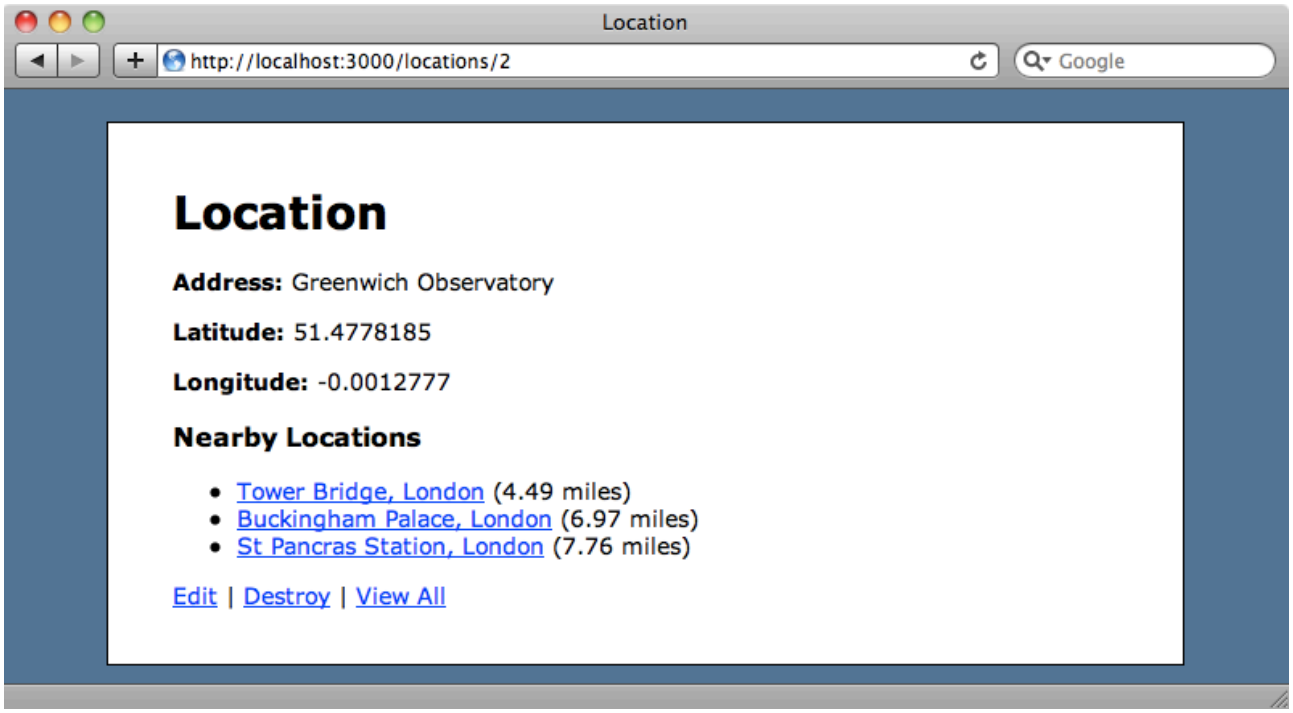
It would be useful when viewing a location to be able to see a list of nearby locations. We have a number of locations in our database now so we'll add this feature to our location page.

We want to list each location within a given distance and Geocoder makes getting nearby locations easy with its `nearbys` method. By default this will return locations within a twenty-mile radius, but we'll restrict it to ten. To list the nearby locations we just loop though each location in `@location.nearbys` and display a link to it along with its distance in miles.

/app/views/locations/show.html.erb

```
<ul>
<% for location in @location.nearbys(10) %>
  <li><%= link_to location.address, location %> ↵
    (<%= location.distance.round(2) %> miles)</li>
<% end %>
</ul><% title "Location" %>

<p>
  <strong>Address:</strong>
  <%= @location.address %>
</p>
<p>
  <strong>Latitude:</strong>
  <%= @location.latitude %>
</p>
<p>
  <strong>Longitude:</strong>
  <%= @location.longitude %>
</p>

<h3>Nearby Locations</h3>
<ul>
<% for location in @location.nearbys(10) %>
  <li><%= link_to location.address, location %> ↵
    (<%= location.distance.round(2) %> miles)</li>
<% end %>
</ul>

<p>
  <%= link_to "Edit", edit_location_path(@location) %> |
  <%= link_to "Destroy", @location, :confirm => ↵
    'Are you sure?', :method => :delete %> |
  <%= link_to "View All", locations_path %>
</p>
```

When we view a location now we'll see the nearby locations listed but the location we have in Liverpool, some 200 miles away, isn't shown in the list.



Next we'll add a search box on locations page so that we can search for locations in a given city. We'll add this in a new form at the top on the `index` view.

/app/views/locations/index.html.erb

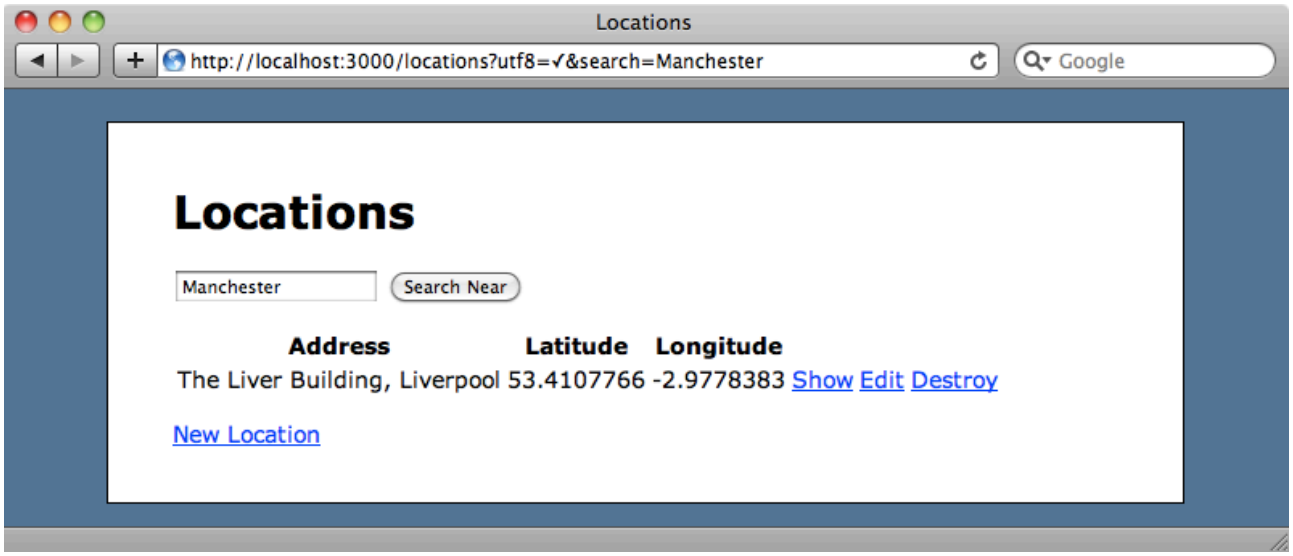
```
<% title "Locations" %>

<%= form_tag locations_path, :method => :get do %>
  <p>
    <%= text_field_tag :search, params[:search] %>
    <%= submit_tag "Search Near", :name => nil %>
<% end %>
<!-- rest of page -->
```

The form uses GET and will call the the `index` page with a `search` parameter when submitted. In the `LocationsController`'s `index` action we'll check for that parameter and if it exists we'll search for nearby locations.

/app/controllers/locations_controller.rb

```
def index
  if params[:search].present?
    @locations = Location.near(params[:search], 50, ↵
      :order => :distance)
  else
    @locations = Location.all
  end
end
```

If we reload the page and search for “Manchester” we’ll see the one matching location 30 miles away in Liverpool.



This is all we need to do to find locations that are near a given city.

Adding Maps

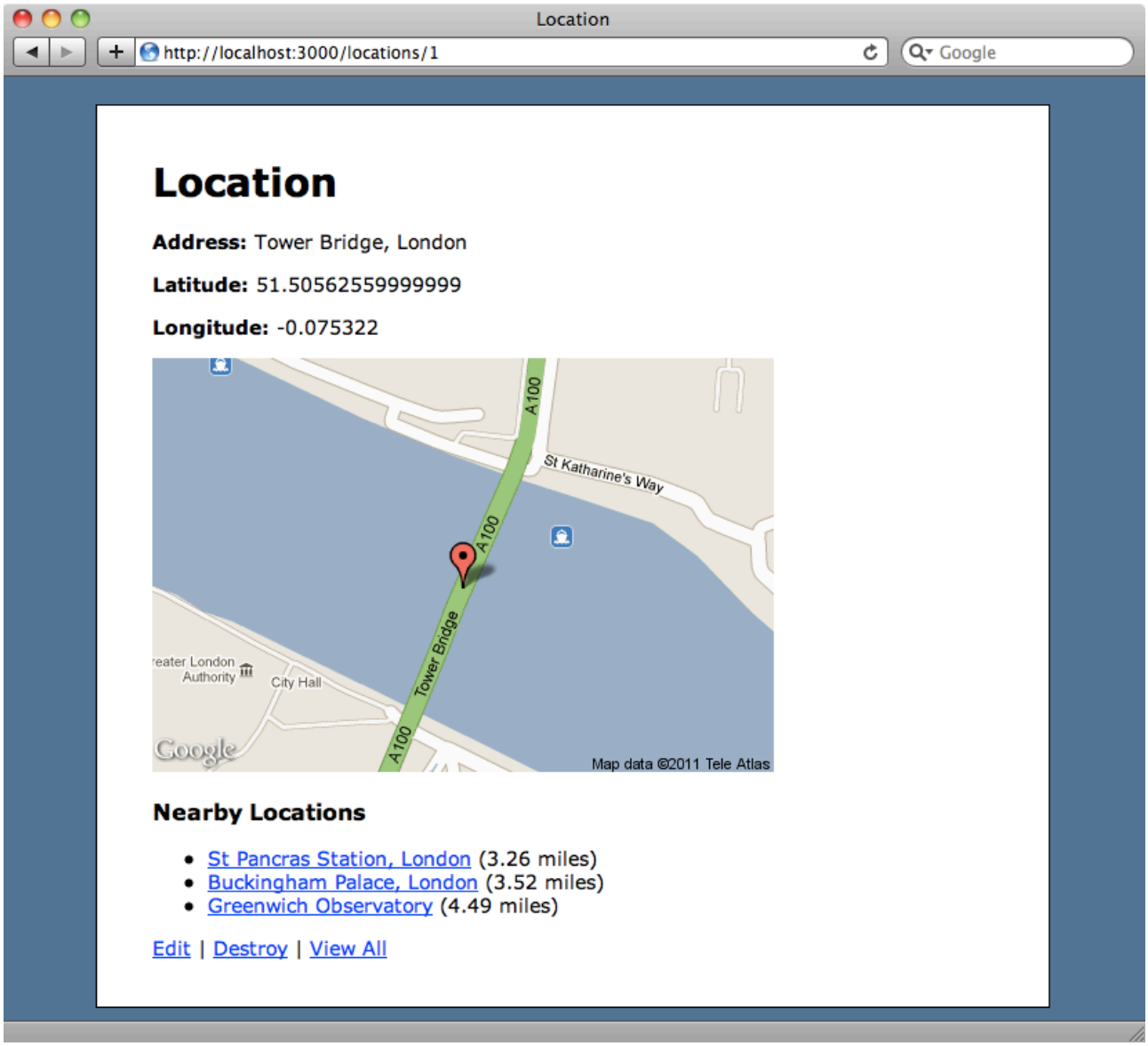
As we’re working with geographic locations it would be useful to have a map to display for each location. We’ll add one on the location page to show exactly where the location is. The Google Maps API provides a variety of ways to add maps to a web page but for simplicity we’ll use the Static Maps API[5]. Maps are added with a simple image tag whose URL has parameters to specify various aspects of the map, including the latitude and longitude, size, zoom and so on. We’ll add the map directly above the list of nearby locations.

[5] http://code.google.com/apis/maps/documentation/staticmaps/

/app/views/locations/show.html.erb

```
<%= image_tag "http://maps.google.com/maps/api/staticmap?
size=450x300&sensor=false&zoom=16&markers=#{@location.latitude}
%2C#{@location.longitude}" %>
```

All of the parameters we pass are static with the exception of the latitude and longitude which come from our `@location` object. When we view the page for a location now we'll see the map for that location.



If you want something a little fancier than a static image it's worth taking a look at the Google Maps For Rails gem[6] which makes interacting with the JavaScript API easy. This gives you a simple way to add interactive maps to your Rails applications.

[6] https://github.com/apneadiving/Google-Maps-for-Rails