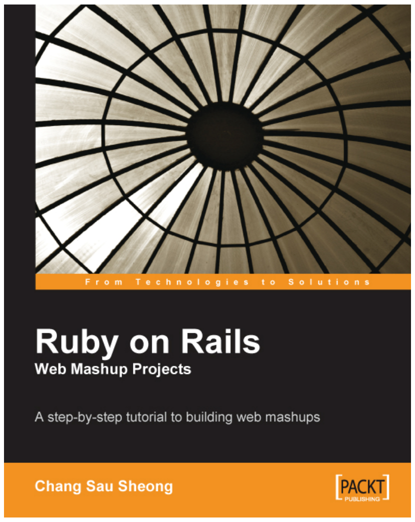
Ruby on Rails Web Mashup Projects

A step-by-step tutorial to building web mashups

Chang Sau Sheong


From Technologies to Solutions
Ruby on Rails
Web Mashup Projects
A step-by-step tutorial to building web mashups
Chang Sau Sheong
PACKT PUBLISHING

Chapter No. 2
"'Find closest' mashup plugin"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.2 "'Find closest' mashup plugin"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Chang Sau Sheong has more than 12 years experience in software application development and has spent much of his career in Web and Internet-based applications. He has a wide range of experience in banking payment-related as well as Internet-based e-commerce software. Currently he is the Director of Software Development of a 50+ strong software development team in Welcome Real-time, a multi-national payment/loyalty software company based in France and Singapore.

Sau Sheong frequently writes for technical magazines and journals including Java Report, Java World, and Dr. Dobb's Journal. He also contributes to open-source projects in various technologies including smart cards, Ruby, and Java. His interests revolve mainly around technology and software development. He has done programming in Java/Java EE, C, C++, PHP, Python, Perl, Smalltalk, Erlang, Ruby/ Ruby on Rails, various smart card platforms, and also worked on various databases. He has a wide range of experience in banking payment-related as well as Internet-based e-commerce software.

Sau Sheong hails from tropical Malaysia but has spent most of his adult and working life in sunny Singapore, where he shares his spare time enthusiastically writing software and equally playing Nintendo Wii with his wife and son. He has a Bachelor's degree in Computer Engineering, a Master's degree in Commercial Law, and is a certified international arbitrator.

Ruby on Rails Web Mashup Projects

A step-by-step tutorial to building web mashups

A web mashup is a new type of web application that uses data and services from one or more external sources to build entirely new and different web applications. Web mashups usually mash up data and services that are available on the Internet—freely, commercially, or through other partnership agreements. The external sources that a mashup uses are known as mashup APIs.

This book shows you how to write web mashups using Ruby on Rails—the new web application development framework. The book has seven real-world projects—the format of each project is similar, with a statement of the project, discussion of the main protocols involved, an overview of the API, and then complete code for building the project. You will be led methodically through concrete steps to build the mashup, with asides to explain the theory behind the code.

What This Book Covers

The first chapter introduces the concepts of web mashups to the reader and provides a general introduction to the benefits and pitfalls of using web mashups as standalone applications or as part of existing web applications.

The first project is a mashup plugin into an existing web application that allows users to find the location of the closest facility from a particular geographic location based on a specified search radius. The location is mapped and displayed on Google Maps.

The second project is another mashup plugin. This plugin allows users to send messages to their own list of recipients, people who are previously unknown to the website, on behalf of the website. The project uses Google Spreadsheets and EditGrid to aggregate the information, and Clickatell and Interfax to send SMS messages and faxes respectively.

The third project describes a mashup plugin that allows you to track the sales ranking and customer reviews of a particular product from Amazon.com. The main API used is the Amazon E-Commerce Service (ECS).

The fourth project shows you how to create a full-fl edged Facebook application that allows a user to perform some of the functions and features of a job board. This mashup uses Facebook, Google Maps, Daylife, Technorati and Indeed.com APIs.

The fifth project shows you how to create a full web mashup application that allows users to view information on a location. This is the chapter that uses the most mashup APIs, including Google Maps, FUTEF, WebserviceX, Yahoo! Geocoding services, WeatherBug, Kayak, GeoNames, Flickr, and Hostip.info.

The sixth project describes a mashup plugin that allows an online event ticketing application to receive payment through Paypal, send SMS receipts, and add event records in the customer's Google Calendar account. The APIs used are Google Calendar, PayPal, and Clickatell.

The final project shows a complex mashup plugin used for making corporate expense claims. It allows an employee to submit expense claims in Google Docs and Spreadsheets, attaching the claims form and the supporting receipts. His or her manager, also using Google Docs and Spreadsheets, then approves the expense claims and the approved claims are retrieved by the mashup and used to reimburse the employee through PayPal. It uses the PayPal APIs and various Google APIs.

2
'Find closest' mashup plugin

What does it do?

This mashup plugin allows your Rails website or application to have an additional feature that allows your users to find the location of the closest facility from a particular geographic location based on a specified search radius. This mashup plugin integrates with your existing website that has a database of locations of the facilities.

Building a kiosk locator feature for your site

Your company has just deployed 500 multi-purpose payment kiosks around the country, cash cows for the milking. Another 500 more are on the way, promising to bring in the big bucks for all the hardworking employees in the company. Naturally your boss wants as many people as possible to know about them and use them. The problem is that while the marketing machine churns away on the marvels and benefits of the kiosks, the customers need to know where they are located to use them. He commands you:

> "*Find a way to show our users where the nearest kiosks to him are, and directions to reach them!*"

What you have is a database of all the 500 locations where the kiosks are located, by their full address. What can you do?

Requirements overview

Quickly gathering your wits, you penned down the following quick requirements:

1. Each customer who comes to your site needs to be able to find the closest kiosk to his or her current location.
2. He or she might also want to know the closest kiosk to any location.
3. You want to let the users determine the radius of the search.
4. Finding the locations of the closest kiosks, you need to show him how to reach them.
5. You have 500 kiosks now, (and you need to show where they are) but another 500 will be coming, in 10s and 20s, so the location of the kiosks need to be specified during the entry of the kiosks. You want to put all of these on some kind of map.

Sounds difficult? Only if you didn't know about web mashups!

Design

The design for this first project is rather simple. We will build a simple database application using Rails and create a main Kiosk class in which to store the kiosk information including its address, longitude, and latitude information. After populating the database with the kiosk information and address, we will use a geolocation service to discover its longitude and latitude. We store the information in the same table. Next, we will take the kiosk information and mash it up with Google Maps and display the kiosks as pushpins on the online map and place its information inside an info box attached to each pushpin.

Mashup APIs on the menu

In this chapter we will be using the following services to create a 'find closest' mashup plugin:

- Google Maps APIs including geocoding services
- Yahoo geocoding services (part of Yahoo Maps APIs)
- Geocoder.us geocoding services
- Geocoder.ca geocoding services
- Hostip.info

Google Maps

Google Maps is a free web-based mapping service provided by Google. It provides a map that can be navigated by dragging the mouse across it and zoomed in and out using the mouse wheel or a zoom bar. It has three forms of views—map, satellite and a hybrid of map and satellite. Google Maps is coded almost entirely in JavaScript and XML and Google provides a free JavaScript API library that allows developers to integrate Google Maps into their own applications. Google Maps APIs also provide geocoding capabilities, that is, they able to convert addresses to longitude and latitude coordinates.

We will be using two parts of Google Maps:

- Firstly to geocode addresses as part of GeoKit's APIs
- Secondly to display the found kiosk on a customized Google Maps map

Yahoo Maps

Yahoo Maps is a free mapping service provided by Yahoo. Much like Google Maps it also provides a map that is navigable in a similar way and also provides an extensive set of APIs. Yahoo's mapping APIs range from simply including the map directly from the Yahoo Maps website, to Flash APIs and JavaScript APIs. Yahoo Maps also provides geocoding services. We will be using Yahoo Maps geocoding services as part of GeoKit's API to geocode addresses.

Geocoder.us

Geocoder.us is a website that provides free geocoding of addresses and intersections in the United States. It relies on `Geo::Coder::US`, a Perl module available for download from the CPAN and derives its data from the TIGER/Line data set, public-domain data from the US Census Bureau. Its reliability is higher in urban areas but lower in the other parts of the country. We will be using Geocoder.us as part of GeoKit's API to geocode addresses.

Geocoder.ca

Geocoder.ca is a website that provides free geocoding of addresses in the United States and Canada. Like Geocoder.us. it uses data from TIGER/Line but in addition, draws data from GeoBase, the Canadian government-related initiative that provides geospatial information on Canadian territories. We will be using Geocoder.ca as part of GeoKit's API to geocode addresses.

Hostip.info

Hostip.info is a website that provides free geocoding of IP addresses. Hostip.info offers an HTTP-based API as well as its entire database for integration at no cost. We will be using Hostip.info as part of GeoKit's API to geocode IP addresses.

GeoKit

GeoKit is a Rails plugin that enables you to build location-based applications. For this chapter we will be using GeoKit for its geocoding capabilities in two ways:

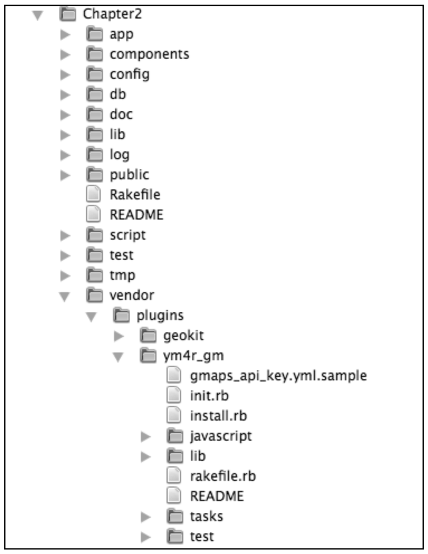
- To determine the longitude and latitude coordinates of the kiosk from its given address
- To determine the longitude and latitude coordinates of the user from his or her IP address

GeoKit is a plugin to your Rails application so installing it means more or less copying the source files from the GeoKit Subversion repository and running through an installation script that adds certain default parameters in your `environment.rb` file.

To install the GeoKit, go to your Rails application folder and execute this at the command line:

```
$./script/plugin install svn://rubyforge.org/var/svn/geokit/trunk
```

This will copy the necessary files to your `RAILS_ROOT/vendor/plugins` folder and run the `install.rb` script.



```
▼ Chapter2
  ▶ app
  ▶ components
  ▶ config
  ▶ db
  ▶ doc
  ▶ lib
  ▶ log
  ▶ public
    Rakefile
    README
  ▶ script
  ▶ test
  ▶ tmp
  ▼ vendor
    ▼ plugins
      ▶ geokit
      ▼ ym4r_gm
          gmaps_api_key.yml.sample
          init.rb
          install.rb
        ▶ javascript
        ▶ lib
          rakefile.rb
          README
        ▶ tasks
        ▶ test
```

Configuring GeoKit

After installing GeoKit you will need to configure it properly to allow it to work. GeoKit allows you to use a few sets of geocoding APIs, including Yahoo, Google, Geocoder.us, and Geocoder.ca.

These geocoding providers can be used directly or through a cascading failover sequence. Using Yahoo or Google requires you to register for an API key but they are free. Geocoder.us is also free under certain terms and conditions but both Geocoder.us and Geocoder.ca have commercial accounts. In this chapter I will briefly go through how to get an application ID from Yahoo and a Google Maps API key from Google.

Getting an application ID from Yahoo

Yahoo's application ID is needed for any Yahoo web service API calls. You can use the same application ID for all services in the same application or multiple applications or one application ID per service.

To get the Yahoo application ID, go to `https://developer.yahoo.com/wsregapp/index.php` and provide the necessary information. Note that for this application you don't need user authentication. Once you click on submit, you will be provided an application ID.

Getting a Google Maps API key from Google

To use Google Maps you will need to have a Google Maps API key. Go to `http://www.google.com/apis/maps/signup.html`. After reading the terms and conditions you will be asked to give a website URL that will use the Google Maps API.

For geocoding purposes, this is not important (anything will do) but to display Google Maps on a website, this is important because Google Maps will not display if the URL doesn't match. However all is not lost if you have provided the wrong URL at first; you can create any number of API keys from Google.

Configuring evironment.rb

Now that you have a Yahoo application ID and a Google Maps API key, go to `environment.rb` under the `RAILS_ROOT/config` folder. Installing GeoKit should have added the following to your `environment.rb` file:

```
# Include your application configuration below
# These defaults are
used in GeoKit::Mappable.distance_to and in acts_as_mappable
GeoKit::default_units = :miles
```

```
GeoKit::default_formula = :sphere
# This is the timeout value in seconds to be used for calls to the
geocoder web
# services. For no timeout at all, comment out the setting. The
timeout unit is in seconds.
# GeoKit::Geocoders::timeout = 3
# These settings are used if web service calls must be routed through
a proxy.
# These setting can be nil if not needed, otherwise, addr and port
must be filled in at a minimum. If the proxy requires authentication,
the username and password can be provided as well.
GeoKit::Geocoders::proxy_addr = nil
GeoKit::Geocoders::proxy_port = nil
GeoKit::Geocoders::proxy_user = nil
GeoKit::Geocoders::proxy_pass = nil
# This is your yahoo application key for the Yahoo Geocoder
# See http://developer.yahoo.com/faq/index.html#appid and
http://developer.yahoo.com/maps/rest/V1/geocode.html
GeoKit::Geocoders::yahoo = <YOUR YAHOO APP ID>
# This is your Google Maps geocoder key.
# See http://www.google.com/apis/maps/signup.html and
http://www.google.com/apis/maps/documentation/#Geocoding_Examples
GeoKit::Geocoders::google = <YOUR GOOGLE MAPS KEY>
# This is your username and password for geocoder.us
# To use the free service, the value can be set to nil or false. For
usage tied to an account, the value should be set to
username:password.
# See http://geocoder.us and
http://geocoder.us/user/signup
GeoKit::Geocoders::geocoder_us = false
# This is your authorization key for geocoder.ca.
# To use the free service, the value can be set to nil or false. For
usage tied to an account, set the value to the key obtained from
Geocoder.ca
# See http://geocoder.ca and
http://geocoder.ca/?register=1
GeoKit::Geocoders::geocoder_ca = false
# This is the order in which the geocoders are called in a failover
scenario
# If you only want to use a single geocoder, put a single symbol in
the array.
# Valid symbols are :google, :yahoo, :us, and :ca
# Be aware that there are Terms of Use restrictions on how you can
use the various geocoders. Make sure you read up on relevant Terms of
Use for each geocoder you are going to use.
GeoKit::Geocoders::provider_order = [:google,:yahoo]
```

Go to the lines where you are asked to put in the Yahoo and Google keys and change the values accordingly. Make sure the keys are within apostrophes.

Then go to the provider order and put in the order you want (the first will be tried; if that fails it will go to the next until all are exhausted):

```
GeoKit::Geocoders::provider_order = [:google,:yahoo]
```

This completes the configuration of GeoKit.

YM4R/GM

YM4R/GM is another Rails plugin, one that facilitates the use of Google Maps APIs. We will be using YM4R/GM to display the kiosk locations on a customized Google Map. This API essentially wraps around the Google Maps APIs but also provides additional features to make it easier to use from Ruby. To install it, go to your Rails application folder and execute this at the command line:

```
$./script/plugin install svn://rubyforge.org/var/svn/ym4r/Plugins/GM/
trunk/ym4r_gm
```

During the installation, the JavaScript files found in the `RAILS_ROOT/vendors/plugin/javascript` folder will be copied to the `RAILS_ROOT/public/javascripts` folder.

A `gmaps_api_key.yml` file is also created in the `RAILS_ROOT/config` folder. This file is a YAML representation of a hash, like the `database.yml` file in which you can set up a test, development, and production environment. This is where you will put in your Google Maps API key (in addition to the `environment.rb` you have changed earlier).

For your local testing you will not need to change the values but once you deploy this in production on an Internet site you will need to put in a real value according to your domain.

What we will be doing

As this project is a mashup plugin, normally you would already have an existing Rails application you want to add this to. However for the purpose of this chapter, I show how the mashup can be created on a fresh project. This is what we will be doing:

- Create a new Rails project
- Install the Rails plugins (GeoKit and YM4R/GM) that will use the various mashup APIs

- Configure the database access and create the database
- Create the standard scaffolding
- Populate the longitude and latitude of the kiosks
- Create the find feature
- Display the found kiosk locations on Google Maps

Creating a new Rails project

This is the easiest part:

```
$rails Chapter2
```

This will create a new blank Rails project.

Installing the Rails plugins that will use the various mashup APIs

In this mashup plugin we'll need to use GeoKit, a Ruby geocoding library created by Bill Eisenhauer and Andre Lewis, and YM4R/GM—a Ruby Google Maps mapping API created by Guilhem Vellut. Install them according to the instructions given in the section above.

Next, we need to create the database that we will be using.

Configuring database access and creating the database

Assuming that you already know how database migration works in Rails, generate a migration using the migration generator:

```
$./script/generate migration create_kiosks
```

This will create a file `001_create_kiosks.rb` file in the `RAILS_ROOT/db/migrate` folder. Ensure the file has the following information:

```
class CreateKiosks < ActiveRecord::Migration
  def self.up
    create_table :kiosks do |t|
      t.column :name, :string
      t.column :street, :string
      t.column :city, :string
      t.column :state, :string
```

```
        t.column :zipcode, :string
        t.column :lng, :float
        t.column :lat, :float
      end
  end
    def self.down
        drop_table :kiosks
  end
  end
```

GeoKit specifies that the two columns must be named `lat` and `lng`. These two columns are critical to calculating the closest kiosks to a specific location.

Now that you have the migration script, run it to create the Kiosk table in your `RAILS_ROOT` folder:

Now that you have the migration script, run migrate to create the Kiosk table in your `RAILS_ROOT` folder:

```
$rake db:migrate
```

This should create the database and populate the kiosks table with a set of data. If it doesn't work please check if you have created a database schema with your favorite relational database. The database schema should be named `chapter2_development`. If this name displeases you somehow, you can change it in the `RAILS_ROOT/config/database.yml` file.

Creating scaffolding for the project

You should have the tables and data set up by now so the next step is to create a simple scaffold for the project. Run the following in your `RAILS_ROOT` folder:

```
$./script/generate scaffold Kiosk
```

This will generate the Kiosk controller and views as well as the Kiosk model. This is the data model for Kiosk, in the `kiosk.rb` file. This is found in `RAILS_ROOT/app/models/`.

```
  class Kiosk < ActiveRecord::Base
    def address
      "#{self.street}, #{self.city}, #{self.state}, #{self.zipcode}"
    end
  end
```

Just add in the `address` convenience method to have quick access to the full address of the kiosk. This will be used later for the display in the info box.

Populating kiosk locations with longitude and latitude information

Before we begin geolocating the kiosks, we need to put physical addresses to them. We need to put in the street, city, state, and zipcode information for each of the kiosks. After this, we will need to geolocate them and add their longitude and latitude information. This information is the crux of the entire plugin as it allows you to find the closest kiosks.

In addition you will need to modify the kiosk creation screens to add in the longitude and latitude information when the database entry is created.

Populate the database with sample data

In the source code bundle you will find a migration file named `002_populate_kiosks.rb` that will populate some test data (admittedly less than 500 kiosks) into the system. We will use this data to test our plugin. Place the file in `RAILS_ROOT/db/migrate` and then run:

```
$rake db:migrate
```

Alternatively you can have some fun entering your own kiosk addresses into the database directly, or find a nice list of addresses you can use to populate the database by any other means.

Note that we need to create the static scaffold first before populating the database using the migration script above. This is because the migration script uses the `Kiosk` class to create the records in the database. You should realize by now that migration scripts are also Ruby scripts.

Bulk adding of longitude and latitude

One of the very useful tools in Ruby, also used frequently in Rails, is **rake**. Rake is a simple make utility with rake scripts that are entirely written in Ruby. Rails has a number of rake scripts distributed along with its installation, which you can find out using this command:

```
$rake --tasks
```

Rails rake tasks are very useful because you can access the Rails environment, including libraries and ActiveRecord objects directly in the rake script. You can create your own customized rake task by putting your rake script into the `RAILS_ROOT/lib/tasks` folder.

We will use rake to add longitude and latitude information to the kiosks records that are already created in the database.

Create an `add_kiosk_coordinates.rake` file with the following code:

```
namespace :Chapter2 do
  desc 'Update kiosks with longitude and latitude information'
  task :add_kiosk_coordinates => :environment do
    include GeoKit::Geocoders

    kiosks = Kiosk.find(:all)
    begin
      kiosks.each { |kiosk|
        loc = MultiGeocoder.geocode(kiosk.address)

        kiosk.lat = loc.lat
        kiosk.lng = loc.lng
        kiosk.update
        puts "updated kiosk #{kiosk.name} #{kiosk.address} =>
                                   [#{loc.lat}, #{loc.lng}]"
                 }
      rescue
      puts $!
    end
  end
end
```

In this rake script you first include the Geocoders module that is the main tool for discovering the coordinate information. Then for each kiosk, you find its longitude and latitude and update the kiosk record.

Run the script from the console in the `RAILS_ROOT` folder:

```
$rake Chapter2:add_kiosk_coordinates
```

Depending on your network connection (running this rake script will of course require you to be connected to the Internet) it might take some time. Run it over a long lunch break or overnight and check the next day to make sure all records have a longitude and latitude entry. This should provide your mashup with the longitude and latitude coordinates of each kiosk. However your mileage may differ depending on the location of the kiosk and the ability of the geocoding API to derive the coordinates from the addresses.

Adding longitude and latitude during kiosk creation entry

Assuming that you have a `kiosks_controller.rb` already in place (it would be generated automatically along with the rest of the scaffolding), you need to add in a few lines very similar to the ones above to allow the kiosk created to have longitude and latitude information.

First, include the geocoders by adding GeoKit after the controller definition, in `kiosks_controller.rb`.

```
class KiosksController < ApplicationController
include GeoKit::Geocoders
```

Next, add in the highlighted lines in the `create` method of the controller.

```
    def create
      @kiosk = Kiosk.new(params[:kiosk])
      loc = MultiGeocoder.geocode(@kiosk.address)
      @kiosk.lat = loc.lat
      @kiosk.lng = loc.lng

      if @kiosk.save
        flash[:notice] = 'Kiosk was successfully created.'
        redirect_to :action => 'list'
      else
        render :action => 'new'
      end
    end
```

Finally, modify the `update` method in the controller to update the correct longitude and latitude information if the kiosk location changes.

```
    def update
      @kiosk = Kiosk.find(params[:id])
      address = "#{params[:kiosk][:street]}, #{params[:kiosk][:city]},
  #{params[:kiosk][:state]}"
      loc = MultiGeocoder.geocode(address)
      params[:kiosk][:lat] = loc.lat
      params[:kiosk][:lng] = loc.lng
      if @kiosk.update_attributes(params[:kiosk])
        flash[:notice] = 'Kiosk was successfully updated.'
        redirect_to :action => 'show', :id => @kiosk
      else
        render :action => 'edit'
      end
    end
```

Creating the find closest feature

Now that you have the kiosk data ready, it's time to go down to the meat of the code. What you'll be creating is a search page. This page will have a text field for the user to enter the location from which a number of kiosks closest to it will be displayed. However, to be user-friendly, the initial location of the user is guessed and displayed on the text field.

Create a `search` action in your controller (called `search.rhtml`, and place it in `RAILS_ROOT/app/views/kiosks/`) to find your current location from the IP address retrieved from your user.

```
def search
  loc = IpGeocoder.geocode(request.remote_ip)
  @location = []
  @location << loc.street_address << loc.city << loc.country_code
end
```

The `remote_ip` method of the Rails-provided request object returns the originating IP address, which is used by GeoKit to guess the location from Hostip.info. The location is then used by `search.rhtml` to display the guessed location.

Note that if you're running this locally, i.e. if you are browsing the application from your PC to a locally running server (for example, off your PC as well), you will not get anything. To overcome this, you can use a dynamic DNS service to point an Internet domain name to the public IP address that is assigned to your PC by your ISP. You will usually need to install a small application on your PC that will automatically update the DNS entry whenever your ISP-assigned IP address changes. There are many freely available dynamic DNS services on the Internet.

When accessing this application, use the hostname given by the dynamic DNS service instead of using localhost. Remember that if you're running through an internal firewall you need to open up the port you're starting up your server with. If you have a router to your ISP you might need to allow port forwarding.

This is a technique you will use subsequently in Chapters 5 and 6.

Create a `search.rhtml` file and place it in the `RAILS_ROOT/app/view/kiosks` folder with the following code:

```
<h1>Enter source location</h1>
Enter a source location and a radius to search for the closest kiosk.
<% form_tag :action => 'find_closest' do %>
<%= text_field_tag 'location', @location.compact.join(',')  %>
<%= select_tag 'radius', options_for_select({'5 miles' => 5, '10
miles' => 10, '15 miles' => 15}, 5) %>
<%= submit_tag 'find' %>
<% end %>
```

Here you're asking for the kiosks closest to a specific location that are within a certain mile radius. We will be using this information later on to limit the search radius.

After that, mix-in the `ActsAsMappable` module into the Kiosk model in `kiosk.rb`.

```
class Kiosk < ActiveRecord::Base
  acts_as_mappable
end
```

This will add in a calculated column called (by default) `distance`, which you can use in your condition and order options. One thing to note here is that the `ActsAsMappable` module uses database-specific code for some of its functions, which are only available in MySQL and PostgresSQL.

Next, create the `find_closest` action to determine the location of nearest kiosks.

```
def find_closest
  @location = MultiGeocoder.geocode(params[:location])
  if @location.success
      @kiosks = Kiosk.find(:all,
               :origin => [@location.lat, @location.lng],
               :conditions => "distance < #{params[:radius]}",
               :order=>'distance')
  end
end
```

The `ActsAsMappable` module mixed in also overrides the find method to include an originating location, either based on a geocode-able string or a 2-element array containing the longitude/latitude information. The returned result is a collection of kiosks that are found with the given parameters.

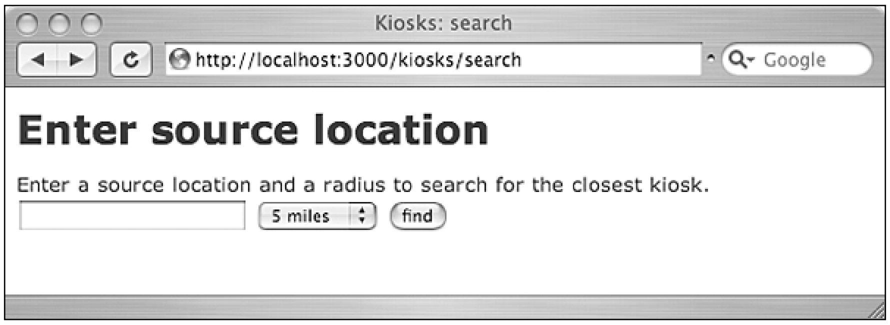
Finally create a simple `find_closest.rhtml` view template (and place it in the `RAILS_ROOT/app/view/kiosks/` folder) to display the kiosks that are retrieved. We'll add in the complex stuff later on.

```
<h1><%= h @kiosks.size %> kiosks found within your search radius</h1>
<ol>
<% @kiosks.each do |kiosk| %>
<li><%= kiosk.name%><br/></li>
<% end %>
</ol>
```

Do a quick trial run and see if it works.

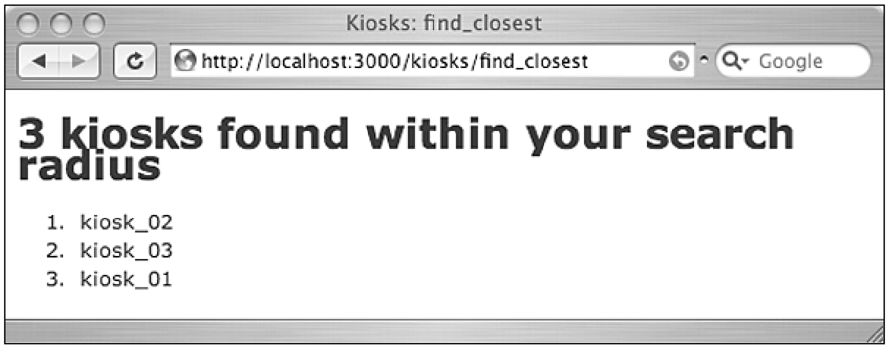
```
$./script/server
```

Then go to `http://localhost:3000/kiosks/search`. If you have some data, put in a nearby location (e.g. from our source data: San Francisco) and click on 'find'. You should be able to retrieve some nearby kiosks.



Enter source location

Enter a source location and a radius to search for the closest kiosk.



3 kiosks found within your search radius

1. kiosk_02
2. kiosk_03
3. kiosk_01

Displaying kiosks on Google Maps

Now that you know where the kiosks are located, it's time to show them on Google Maps. For this we'll be using the YM4R/GM plugin. If you haven't installed this plugin yet, it's time to go back and install it.

To add display to Google Maps, you will need to change the `find_closest` action as well as the `find_closest` view template. First, add the `find_closest` action in the `kiosks_controller.rb`:

```
def find_closest
 @location = MultiGeocoder.geocode(params[:location])
 if @location.success
   @kiosks = Kiosk.find(:all,
             :origin => [@location.lat, @location.lng],
             :conditions => ["distance < ?", params[:radius]],
             :order=>'distance')
   @map = GMap.new("map_div")
```

```
        @map.control_init(:large_map => true, :map_type => true)
        # create marker for the source location
        @map.icon_global_init( GIcon.new(:image =>
              "http://www.google.com/mapfiles/ms/icons/red-pushpin.png",
                                  :shadow => "http://www.google.com/
                                              mapfiles/shadow50.png",
                                  :icon_size => GSize.new(32,32),
                                  :shadow_size => GSize.new(37,32),
                                  :icon_anchor => GPoint.new(9,32),
                                  :info_window_anchor => GPoint.new(9,2),
                                  :info_shadow_anchor =>
                                          GPoint.new(18,25)),
               "icon_source")
        icon_source = Variable.new("icon_source")
        source = GMarker.new([@location.lat, @location.lng],
                  :title => 'Source',
                  :info_window => "You searched for kiosks
                       <br>#{params[:radius]} miles around this source",
                  :icon => icon_source)
        @map.overlay_init(source)
        # create markers one for each location found
        markers = []
        @kiosks.each { |kiosk|
          info = <<EOS
  <em>#{kiosk.name}</em><br/>
  #{kiosk.distance_from(@location).round} miles away<br/>
  <a href="http://maps.google.com/maps?saddr=#{u(@location.to_
  geocodeable_s)}&daddr=#{u(kiosk.address)}>directions here from
  source</a>
  EOS
          markers << GMarker.new([kiosk.lat, kiosk.lng], :title =>
  kiosk.name, :info_window => info)
        }
        @map.overlay_global_init(GMarkerGroup.new(true, markers),"kiosk_
  markers")
        # zoom to the source
        @map.center_zoom_init([@location.lat, @location.lng], 12)
      end
    end
```

Google Maps API is a JavaScript library and YM4R/GM code is a library that creates JavaScript scripts to interact and manipulate the Google Maps API. Almost all classes in the library correspond with an equivalent Google Maps API class, so it is important that you are also familiar with the Google Maps API. The online documentation comes in very useful here so you might want to open up the Google Maps reference documentation (`http://www.google.com/apis/maps/documentation/reference.html`) as you are coding.

Let's go over the code closely.

The first line creates a `GMap` object that is placed inside a `<div>` tag with the id `map_div` while the second line sets some control options.

```
@map = GMap.new("map_div")
@map.control_init(:large_map => true, :map_type => true)
```

The next few lines then create a `GMarker` object from the source location that the user entered that uses a specific icon to show it then overlays it on the map. There are several options you can play around with here involving setting the image to be shown as the marker. For this chapter I used a red-colored pushpin from Google Maps itself but you can use any image instead. You can also set the text information window that is displayed when you click on the marker. The text can be in HTML so you can add in other information including images, formatting, and so on.

```
# create marker for the source location
@map.icon_global_init( GIcon.new(:image =>
      "http://www.google.com/mapfiles/ms/icons/red-pushpin.png",
                        :shadow => "http://www.google.com/
                                   mapfiles/shadow50.png",
                        :icon_size => GSize.new(32,32),
                        :shadow_size => GSize.new(37,32),
                        :icon_anchor => GPoint.new(9,32),
                        :info_window_anchor => GPoint.new(9,2),
                        :info_shadow_anchor =>
                         GPoint.new(18,25)), "icon_source")
icon_source = Variable.new("icon_source")
source = GMarker.new([@location.lat, @location.lng],
          :title => 'Source',
          :info_window => "You searched for kiosks
              <br>#{params[:radius]} miles around this source",
          :icon => icon_source)
@map.overlay_init(source)
```

The lines of code after that go through each of the located kiosks and create a `GMarker` object then overlay it on the map too. For each kiosk location, we put in an info window that describes the distance away from the source location and a link that shows the directions to get from the source to this kiosk. This link goes back to Google and will provide the user with instructions to navigate from the source location to the marked location.

Note that you need to URL encode the location/address strings of the source and kiosks, so you need to include `ERB::Util` as well (along with `GeoKit::Geocoders`). This is the `u()` method. In `kiosks_controller.rb`, add:

```
include ERB::Util
```

then add the following (beneath the code entered above):

```
         # create markers one for each location found
        markers = []
        @kiosks.each
        { |kiosk|
          info = <<EOS
<em>#{kiosk.name}</em><br/>
#{kiosk.distance_from(@location).round} miles away<br/>
<a href="http://maps.google.com/maps?saddr=#{u(@location.
to_geocodeable_s)}&daddr=#{u(kiosk.address)}>directions here from
source</a>
EOS
          markers << GMarker.new([kiosk.lat, kiosk.lng],
                 :title => kiosk.name, :info_window => info)
        }
        @map.overlay_global_init(GMarkerGroup.new(true, markers),
                                                 "kiosk_markers")
```

Finally the last line zooms in and centers on the source location.

```
        # zoom to the source
        @map.center_zoom_init([@location.lat, @location.lng], 12)
```

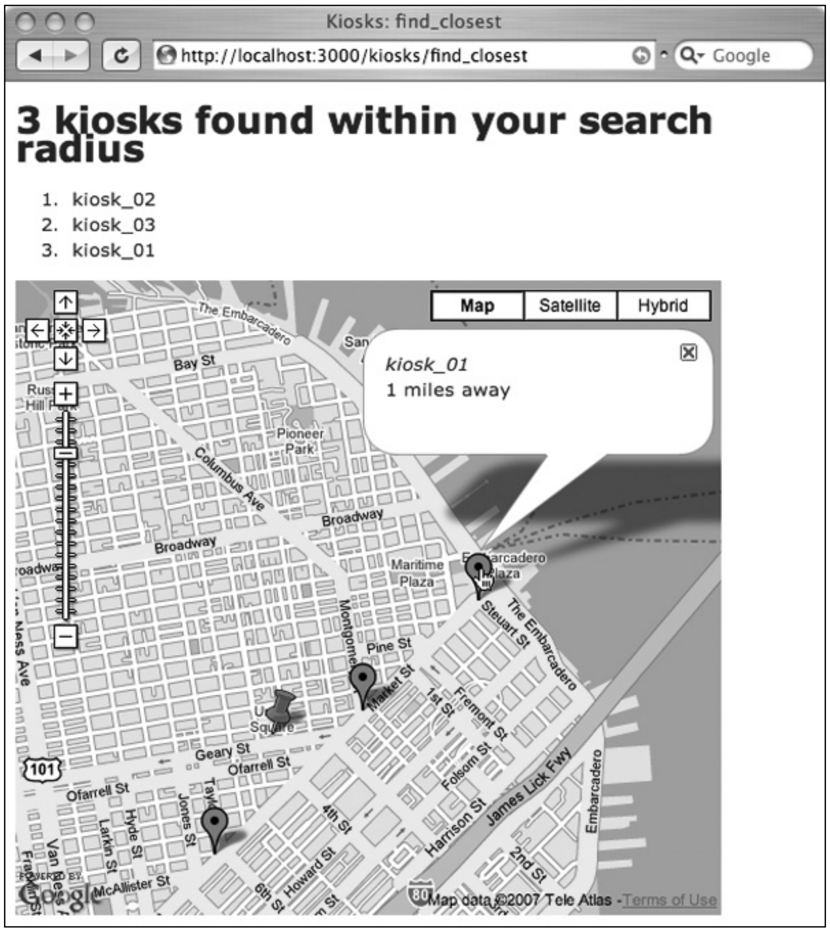
Now let's look at how the view template is modified to display Google Maps. The bulk of the work has already been done by YM4R/GM so you need only to include a few lines.

```
<h1><%= h @kiosks.size %> kiosks found within your search radius</h1>
<ol>
<% @kiosks.each do |kiosk| %>
<li><%= kiosk.name%><br/></li>
<% end %>
</ol>
```

```
<%= GMap.header %>
<%= javascript_include_tag("markerGroup") %>
<%= @map.to_html%>
<%= @map.div(:width => 500, :height => 450)%>
```

`Gmap.header` creates the header information for the map, including YM4R/GM and Google Maps API JavaScript files. We are also using `GMarkerGroups` so we need to include the GMarkerGroup JavaScript libraries. Next, we need to initialize the map by calling `map.to_html`. Finally we'll need to have a `div` tag that is the same as the one passed to the GMap constructor in the controller (`map_div`). This is done by calling the `div` method of the `GMap` object. To size the map correctly we will also need to pass on its dimensions (height and width here).

And you're ready to roll! Although the page doesn't display the best layout, you can spice things up by adding the necessary stylesheets to make the view more presentable.


Kiosks: find_closest
http://localhost:3000/kiosks/find_closest
Google
Map
Satellite
Hybrid
Map data ©2007 Tele Atlas - Terms of Use

3 kiosks found within your search radius

1. kiosk_02
2. kiosk_03
3. kiosk_01

kiosk_01
1 miles away

Summary

What we've learned in this chapter is to create a mashup with Ruby on Rails on a number of mapping and geocoding providers including Yahoo, Google, geocoder.us, geocoder.ca, and hostip.info. We learned to create a mashup that gives us a map of the closest kiosks to a particular location, given an existing database of kiosks that have location addresses. This is just an introduction to the synergistic value that mashups bring to the table, creating value that was not available in individual APIs. When they are all put together, you have a useful feature for your website.