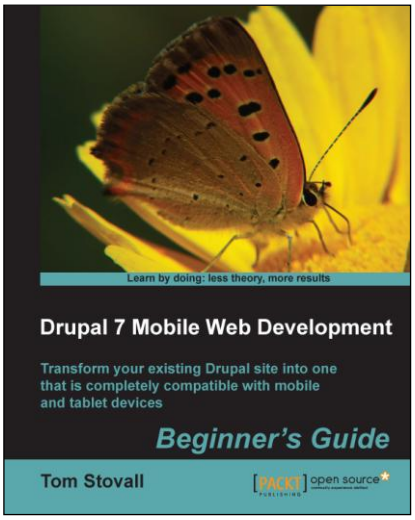
# Drupal 7 Mobile Web Development Beginner's Guide

**Tom Stovall**




## Chapter No. 7
## "Location, Location, Location"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Location, Location, Location"

A synopsis of the book's content

Information on where to buy this book

# About the Author

**Tom Stovall** got a Timex Sinclair 1000 in 1982 from his mom for his birthday and the first night he slept with it under his pillow. Both school teachers, his mom and dad always made sure he had access to computers and today's programming chops owe their origins to those lazy summers spent in front of whatever hardware he could beg, borrow, or use when no one was looking.

Tom started doing websites in 1995, then with PERL, later with PHP. He was the principal front-end developer on Performance.gov, a cost-tracking, Drupal-based website for REI Systems, Inc and the President's Office of Management and Budget during it's year-long development cycle. He now works for Apigee, Inc in Palo Alto, CA developing Drupal sites in support of their enterprise API product and is the maintainer on several Drupal contrib modules.

# Drupal 7 Mobile Web Development Beginner's Guide

It's not an overstatement to say that handhelds have changed the world. What was, just 10 years ago, simply a phone is now the center of your online life and, for many users, their primary Internet device. The power of the smart phone is shaking up the world from Main Street and Wall Street to Pennsylvania Avenue and Downing Street.

Drupal is the perfect platform on which to build a mobile strategy. The power of millions of developers world-wide ensures that there's no problem you face that has not already been overcome by multiple developers and solved with any one of the hundreds of thousands of Drupal contributed projects.

## What This Book Covers

*Chapter 1, When is a Phone Not a Phone?,* explains what we mean when we say "mobile." In this chapter, we'll take a look at the mobile platforms in use today and how they behave and render today's HTML standards.

*Chapter 2, Setting up a Local Development Environment,* teaches you to work in a team environment with version control and to create a local version of our site on Windows or Mac OS with Drush, Drush Make and a make file, and our standard open source PHP *AMP stack. It outlines a team workflow of building the code locally and pushing it to the live site.

*Chapter 3, Selecting the Right Domain for your Mobile Site,* guides you through setting up the Domain Access and Drupal Behaviors modules that redirect mobile and desktop browsers to the version of the website most appropriate for their client. In this chapter, we will learn to share content across sites without resorting to a multi site install.

*Chapter 4, Introduction to a Theme,* introduces the idea of progressive enhancement with CSS. In this chapter, we'll create a very simple HTML5 theme that will serve mobile clients with CSS Media Queries until a highly customized one can be devised.

*Chapter 5, A Home with a View,* demonstates the use of Context and Image Styles to create a customized view for the home page. In this chapter, we'll create a mobile-friendly menu and bundle it up into a feature that can push the new content to your live site in one fell swoop.

*Chapter 6, The Elephant in the Room: Audio, Video,* and Flash Media, teaches you to create a compelling audio and video experience without using Flash. It teaches you to create data visualization using data we've pulled from a View and the HighCharts JavaScript library.

*Chapter 7, Location, Location, Location* helps you to set up location services and cover some common use cases, as well as some uncommon ones using GMap, Location, Open Layers and Map Box.

*Chapter 8, Services with a Smile,* explores the Services module which serves up pieces of node content from a REST and/or SOAP API. In this chapter, we will leverage this module to add some interesting interactivity to our example site.

*Chapter 9, Putting it Together,* guides you in adding some advanced theming to your site and making the site more responsive to the various devices that will be accessing it.

*Chapter 10, Tabula Rasa: Nurturing your site for tablets,* explores the emerging tablet market and covers special design considerations and conventions for designing for tablet use.

*Chapter 11, A Home in the Clouds,* explores team deployment solutions such as Hudson/ Jenkins, Features integration hooks and breaks down the go-live process to something that's repeatable and, with any luck at all, scriptable.

*Appendix, Pop Quiz Answers,* contains the answers to all the pop quiz questions for all the chapters.

# 7

# Location, Location, Location

*Big Jimmy has been talking to some guys—franchise guys. Jimmy always had visions of opening one or more locations, but never really had that "push" to get the job done. The franchise guys are pushing him to open up a few more locations in the Tampa Bay area to prove the validity of the "pizza kitchen" concept. Then they want to turn the kitchen into a franchised restaurant and start taking on investors and new store owners.*

*With this information in mind, Little Jimmy and I had a discussion about some of the new information the website would need to accommodate in order to keep pace with his father's dreams for the restaurant. We'll need to be able to store location information about the different stores. We'll also want a list of stores giving the customer the ability to find the store closest to them. Luckily, Drupal can accommodate all of that and do it very well.*

In this chapter, we'll:

- Learn about the **Location** and **GMap** modules and how to use them as building blocks to create a rich mobile (and desktop) user experience
- Learn about the `navigator.geolocation` object
- Add location information to node objects
- Geocode a node's location data

**A word about browsers**

You will need to use Safari, Chrome, Firefox or Internet Explorer 9 or later in order to work on the tasks in this chapter. The Geolocation API wasn't included in IE until version 9. Safari, Chrome, and Firefox have had the Geolocation API for a year or so now, so any version that's less than a year old should work.

# Geolocation

Your position on the earth can be derived in one of the two ways. The first being the GPS satellite system. It's the system used in most cars and, lately, most smart phones. The GPS satellites constantly bounces signals off the earth and there's a special receiver in the GPS hardware that decodes that signal and uses it to determine longitude and latitude. The GPS hardware then returns the coordinates to any software needing them.

The second way is a little bit more complex. All over the world, people have implemented Wi-Fi networks and cell phone tower networks. Most of them are stationary and installed in houses, businesses, cities, and roadsides across the country. In the same way that Google Maps has cars traveling the country taking photos of street views, there are geolocation companies that travel the country and take a census of the available Wi-Fi networks and their relative signal strength and the geolocation of the Wi-Fi measuring device. All of the cell phone towers in the US are already geocoded and their longitude and latitude locations are known. Software can then, with an amazing degree of accuracy determine your location from the Wi-Fi and cell networks and relative strengths available to your desktop, laptop computer, or cell phone without any assistance from GPS-specific hardware.

For a year or so now, the desktop versions of Firefox and WebKit (Safari and Chrome) have implemented the HTML5 geolocation interface for JavaScript using the second method. The browsers will always warn the users about beginning a geolocation survey and allow the user to opt out for privacy reasons.

That's why, as a web developer, it's frustrating to me that more web developers don't take advantage of the geolocation JavaScript object when developing their websites, especially when it comes to locating the physical location of their client or their client's stores.

# The navigator.geolocation object

The JavaScript object that powers the desktop and handheld browser's location abilities is the `navigator.geolocation` object. Because it can take time to make a call to the geolocation hardware or website that will derive the location, using the object does not stop the JavaScript execution. You hand the `geolocation` object a function to execute once the location is derived and a function to execute on failure, and the JavaScript execution continues while the `geolocation` object goes to work. We'll look more into these calls in the *Time for action – downloding and enabling the close2u module* section.

First, let's create some nodes with the **Location** module and get them geocoded with longitude and latitude coordinates.

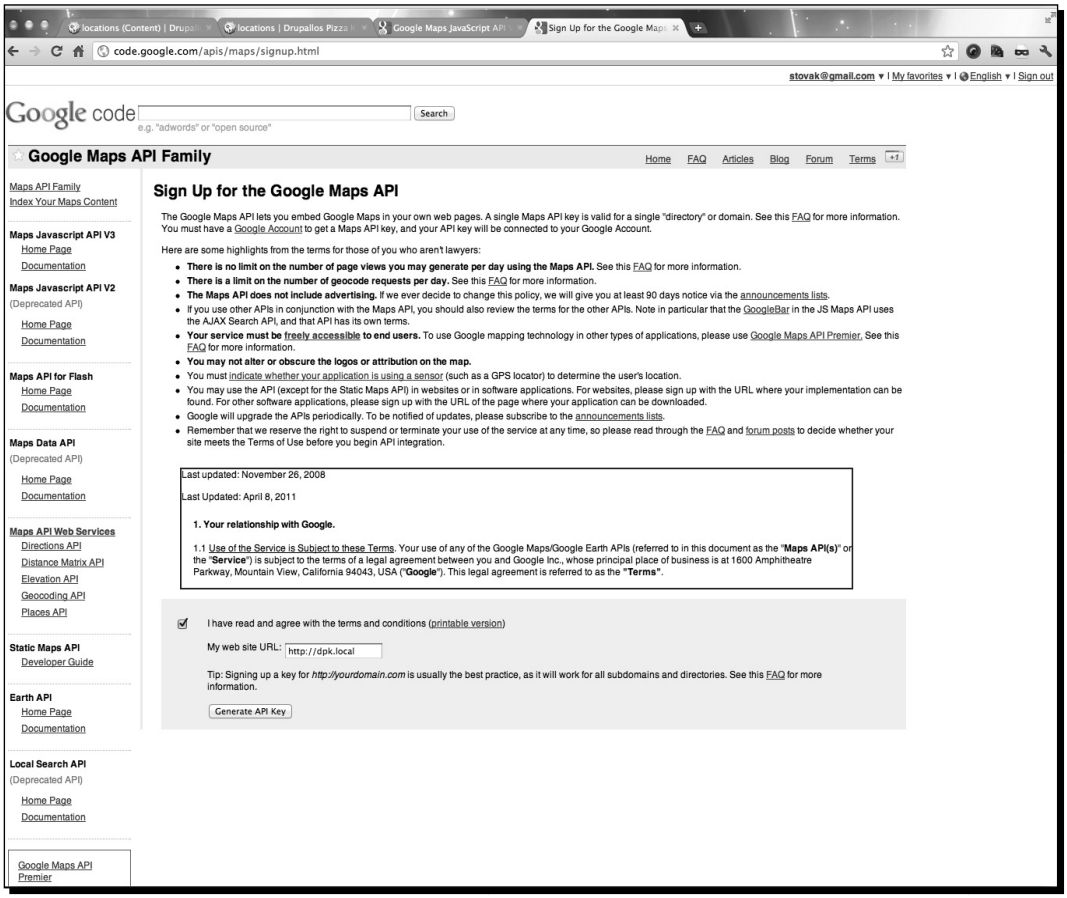
# Time for action – adding location data to nodes

If you take a look at the Drupal Database, you'll notice that the location has its own database table. Locations are separate database objects. They are then related to other Drupal data entities such as nodes and users. We're going to add some location data from Google's Map service to nodes in this example, but the location data could just as easily be linked to almost any data entity that Drupal recognizes:

1. Open a terminal window and enter the following commands:

   ```
   cd ~/Sites/dpk
   drush dl gmap location
   ```

2. Navigate to `http://code.google.com/apis/maps/signup.html`:



**3.** As shown in the preceding screenshot, enter the URL of your test site (`dpk.local`) and check the **I have read and agree with the terms and conditions** checkbox. You're just using this site for development. You will need to obtain another API key for any live site. Be sure that you actually agree to the terms and conditions if you use the key on a live site. Click on the **Generate API Key** button and on the next page, you will see a long API key, as shown in the following screenshot. Copy and paste it somewhere, where you won't lose it:

Your key is:

ABQIAAAA4JmBWcOQE1zkJ7MIbtelUhQ7_I80uvavCXkMI0RqNn7yoBv1qRRReuALK6uFiZ7mwLbuE4JSCuCogA

> Your API key will almost certainly be different from this one but will be a long string of letters, numbers, and symbols like this.

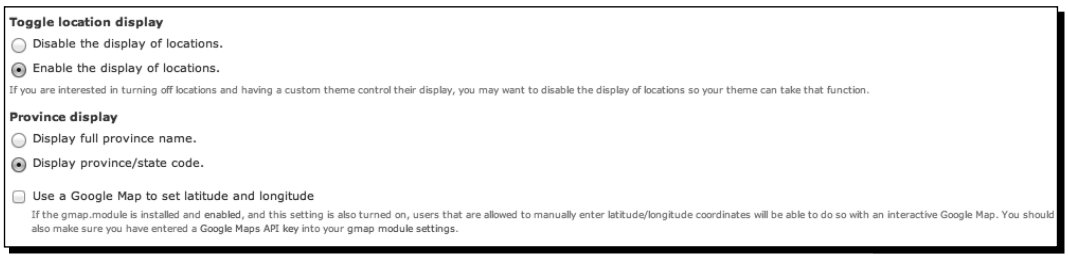
**4.** Navigate to **Admin** | **Modules** and ensure that all of the modules in the **Location** and **GMap** groups are enabled:

Location

| Enabled | Name | Version | Description | Operations |
|---|---|---|---|---|
| ☑ | **GMap** | 7.x-1.x-dev | Filter to allow insertion of a google map into a node<br>Required by: close2u (enabled), GMap Location (enabled), GMap Macro Builder (enabled), GMap Taxonomy Markers (enabled), Location Geocode Update (enabled) | Configure |
| ☑ | **GMap Location** | 7.x-1.x-dev | Display location.module information on Google Maps<br>Requires: GMap (enabled), Location (enabled) | Permissions Configure |
| ☑ | **GMap Macro Builder** | 7.x-1.x-dev | UI for building GMap macros.<br>Requires: GMap (enabled) | Permissions |
| ☑ | **GMap Taxonomy Markers** | 7.x-1.x-dev | Taxonomy based markers<br>Requires: Taxonomy (enabled), Options (enabled), Field (enabled), Field SQL storage (enabled), GMap (enabled) | |
| ☑ | **Location** | 7.x-3.x-dev | The location module allows you to associate a geographic location with content and users. Users can do proximity searches by postal code. This is useful for organizing communities that have a geographic presence.<br>Required by: close2u (enabled), GMap Location (enabled), Node Locations (enabled), Location Add Another (enabled), Location CCK (disabled), Location Fax (enabled), Location Generate (disabled), Location Geocode Update (enabled), Location Phone (enabled), Location Search (enabled), Location Taxonomy (enabled), User Locations (enabled) | Help Permissions Configure |
| ☑ | **Location Add Another** | 7.x-3.x-dev | Allows you to quickly add locations directly from a node without having to click 'edit' first.<br>Requires: Location (enabled), Node Locations (enabled) | |
| ☑ | **Location Fax** | 7.x-3.x-dev | Allows you to add a fax number to a location.<br>Requires: Location (enabled) | |
| ☑ | **Location Geocode Update** | | Adds action to update node location<br>Requires: Location (enabled), GMap (enabled) | |
| ☑ | **Location Phone** | 7.x-3.x-dev | Allows you to add a phone number to a location.<br>Requires: Location (enabled) | |
| ☑ | **Location Search** | 7.x-3.x-dev | Advanced search page for locations.<br>Requires: Search (enabled), Location (enabled) | |
| ☑ | **Location Taxonomy** | 7.x-3.x-dev | Associate locations with taxonomy terms.<br>Requires: Location (enabled), Taxonomy (enabled), Options (enabled), Field (enabled), Field SQL storage (enabled) | |
| ☑ | **Node Locations** | 7.x-3.x-dev | Associate locations with nodes.<br>Requires: Location (enabled)<br>Required by: Location Add Another (enabled) | |
| ☑ | **User Locations** | 7.x-3.x-dev | Associate locations with users.<br>Requires: Location (enabled) | Permissions Configure |

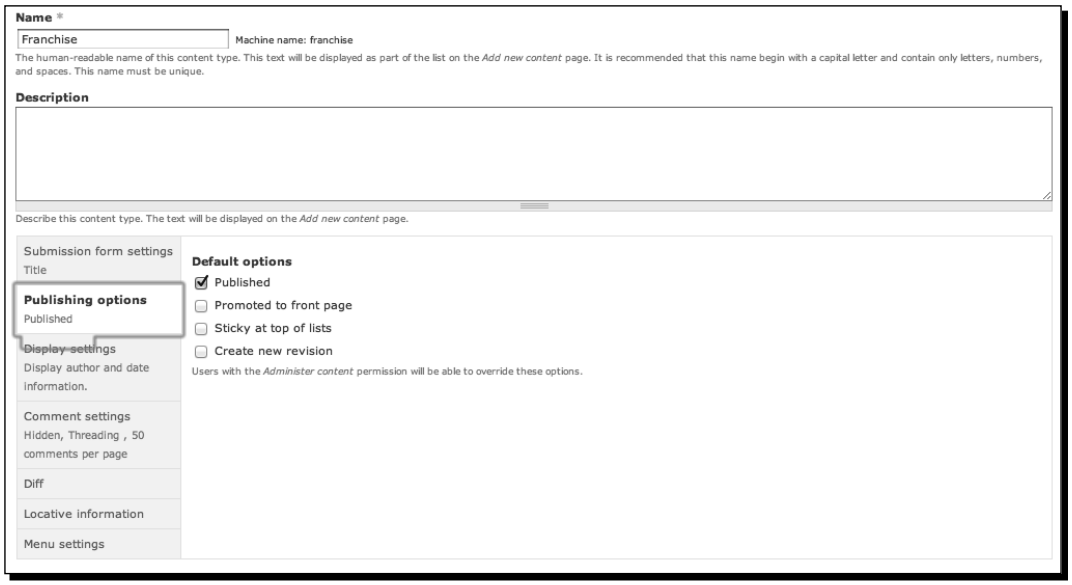
***5.*** Navigate to **Configuration** | **Services** | **GMap** and paste the API key in the **Google Maps API Key** field:


***6.*** Navigate to **Admin** | **Content** | **Location**. Make sure you select the **Enable the display of locations** options:

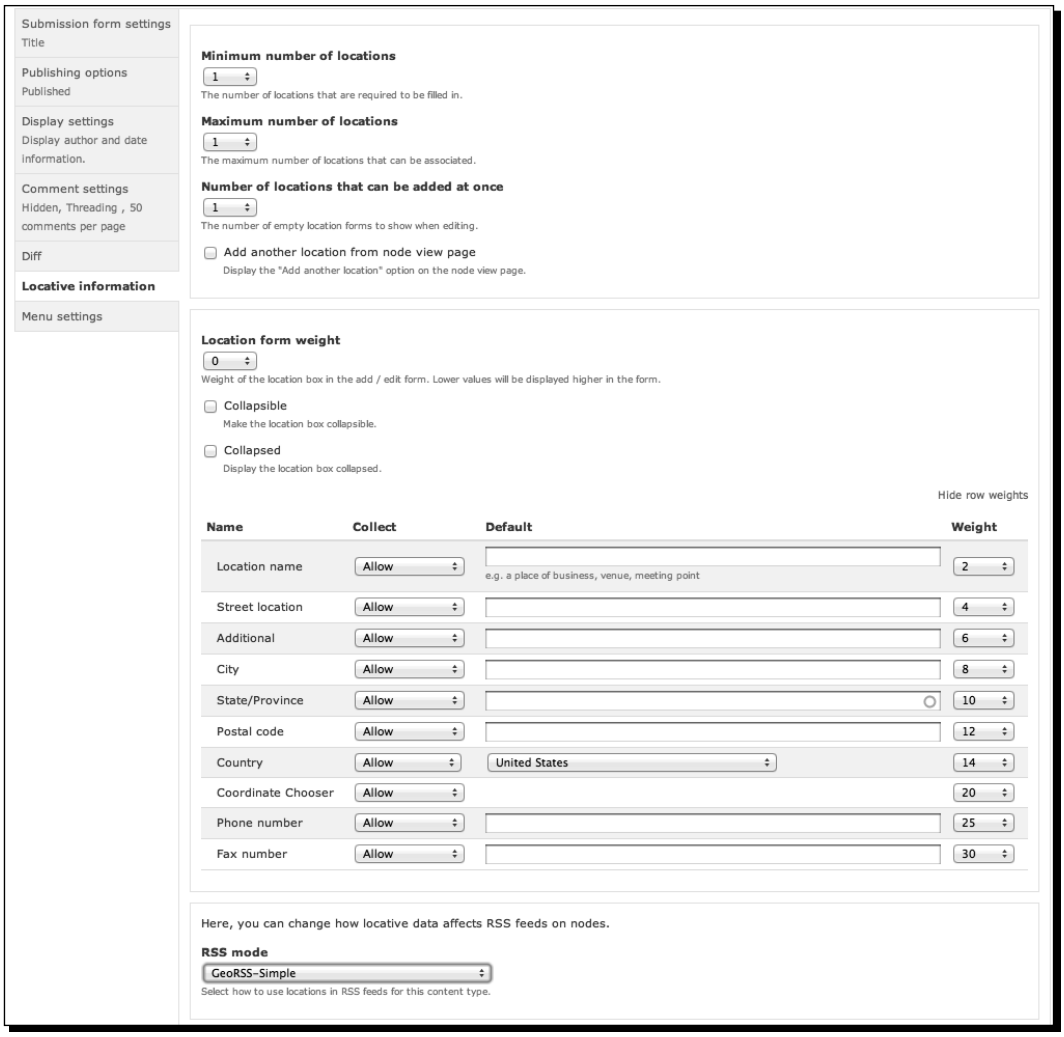



***7.*** Navigate to **Structure** | **Content types** | **Add content type**.

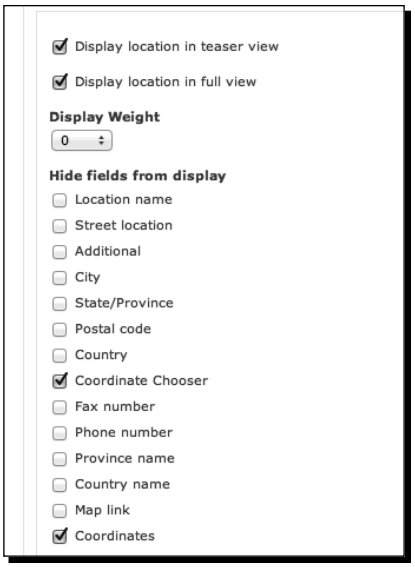
***8.*** We're going to call this content type **Franchise** (see the following screenshot). Under **Publishing options**, select the **Published** option, and the **Promoted to front page** option should be unchecked. Under **Comment settings**, select **Hidden**. Under **Locative information**, change **Minimum Number of Locations** to **1** (with **Maximum number of locations** set to **1**). Save the new content type:




***9.*** Now, navigate back to **Admin** | **Structure** | **Content types**. Under **Locative information**, there are options to gather **Postal code**, **City**, **State**, **Phone number**, **Fax number** and so on. Change them all to **Allow**:



***10.*** At the bottom of the location information, we will have to hide the **Coordinate Chooser** and check the **Display location in teaser view** and **Display location in full view** checkboxes:




***11.*** Once the content type is saved, navigate back to **Structure** | **Content types** | **Franchise** | **Manage fields**.

***12.*** Create a new field called **Hours** with the machine name as **hours**. Under **Widget Type**, choose **Text Field** and under **Hours fields settings** change **Number of Values** to **Unlimited**.

*Franchise* settings

These settings apply only to the *Hours* field when used in the *Franchise* type.

**Label** *

Hours

Required field

**Help text**

Instructions to present to the user below this field on the editing form.
Allowed HTML tags: <a> <b> <big> <code> <del>  <i> <ins> <pre> <q> <small>   <sub> <sup> <tt> <ol> <ul> <li> <p> <br> <img>

**Size of textfield** *

60

**Text processing**

Plain text  Filtered text (user selects text format)

Default value

The default value for this field, used when creating new content.

**Hours**

Sun-Thu 11am-10pm

*Hours* field settings

These settings apply to the *Hours* field everywhere it is used. Because the field already has data, some settings can no longer be changed.

**Number of values**

Unlimited

Maximum number of values users can enter for this field.
'Unlimited' will provide an 'Add more' button so the users can add as many values as they like.

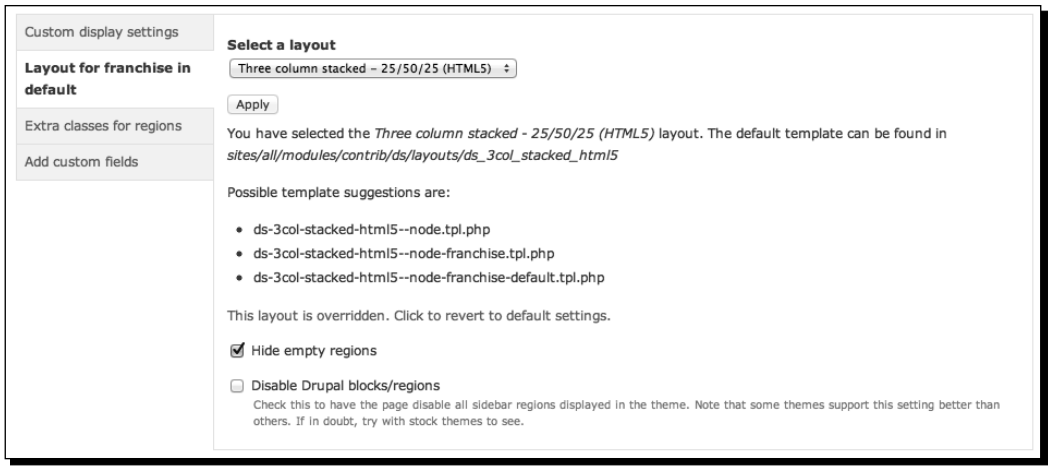
**Maximum length** *

255

The maximum length of the field in characters.

Save settings

**13.** Navigate to **Structure** | **Content types** | **Franchise** | **Manage display**. In the **Layout for franchise in default** tab, select **Three column stacked - 25/50/25 (HTML5)** under **Select a layout** and then save the settings:



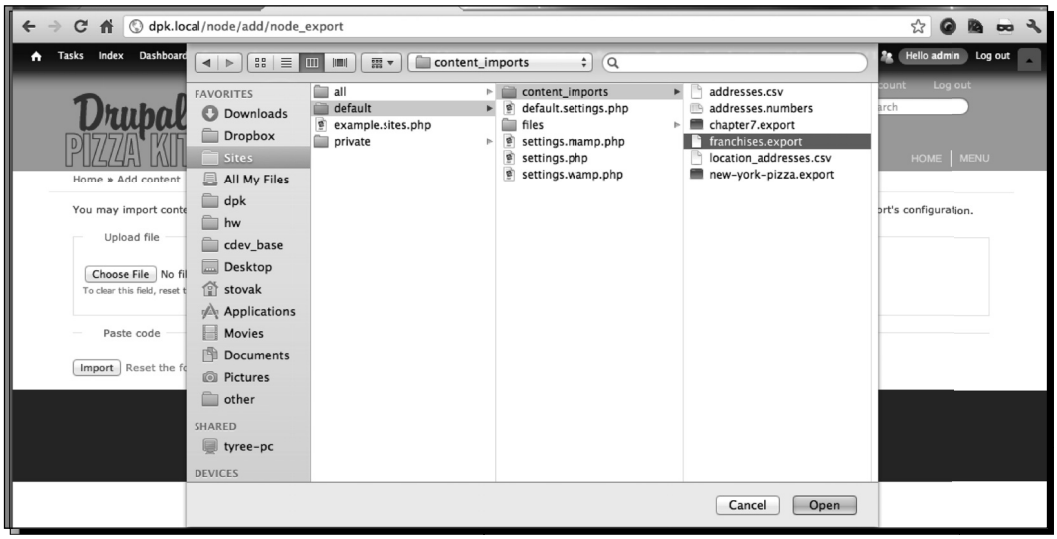

**14.** Click on the **Add custom fields** tab and then click on **Add a code field**. Call it **Location (Address)**. Check the **Node** entity. Add the following as the code value, then save the field:

```
<?php
  $settings = variable_get('location_settings_node_' . $entity-
>type, array());
if (isset($entity->locations)) {
print drupal_render(location_display($settings, $entity-
>locations));
  }
?>
```

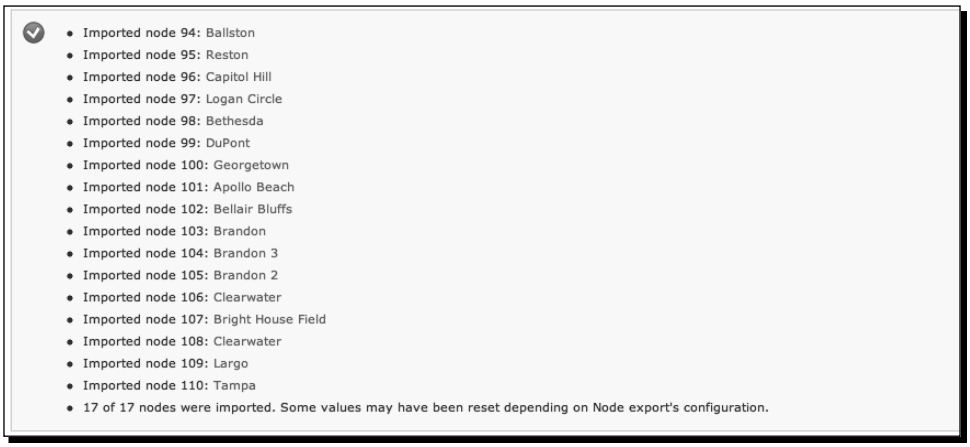
**15.** Move the **Title** and **Hours** fields to the **Left** region. Move the newly created **Location (Address)** field to the **Middle** region. Save the node display:

| Field | Weight | Parent | Region | Label | Format | |
|---|---|---|---|---|---|---|
| **Header** | | | | | | |
| *No fields are displayed in this region* | | | | | | |
| **Left** | | | | | | |
| Title | 0 | – None – | Left | <Hidden> | Default | Wrapper: h2 |
| Hours | 1 | – None – | Left | Above | Default # | |
| **Middle** | | | | | | |
| Location (Address) | 2 | – None – | Middle | <Hidden> | Default | |
| **Right** | | | | | | |
| *No fields are displayed in this region* | | | | | | |
| **Footer** | | | | | | |
| *No fields are displayed in this region* | | | | | | |
| **Disabled** | | | | | | |
| Post date | 0 | – None – | Disabled | <Hidden> | Long | |
| User picture | 0 | – None – | Disabled | <Hidden> | Thumbnail | |
| Author | 0 | – None – | Disabled | <Hidden> | Author | |
| Read more | 0 | – None – | Disabled | <Hidden> | Default | Link text: Read more<br>Link: Yes |
| Links | 0 | – None – | Disabled | <Hidden> | Default | |
| Comments | 0 | – None – | Disabled | <Hidden> | Default | |
| Domain access | 1 | – None – | Disabled | | Visible | |

***16.*** Choose **Content** | **Add content** | **Node Export: Import**. Click on **Upload file**. In the `sites/default/content_exports` folder, there should be a file called `franchiese.export`. Choose that file and click on **Upload**. Then click on **Import**:

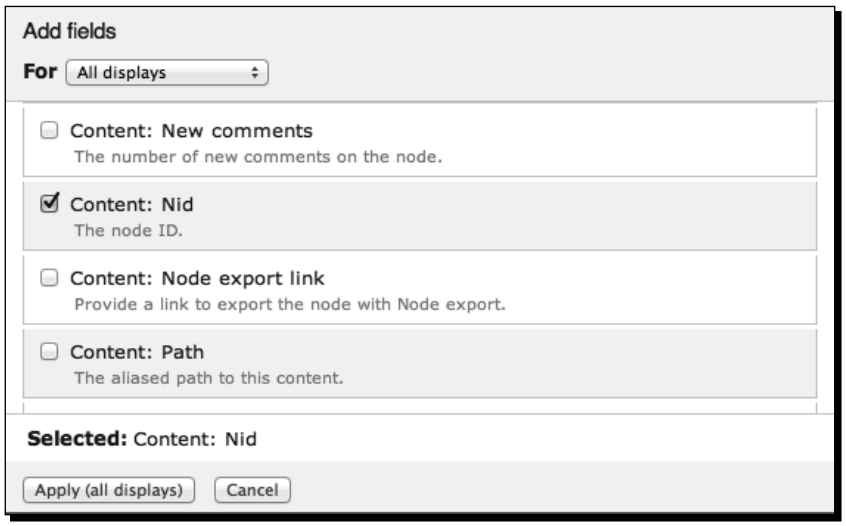


***17.*** As shown in the following screenshot, Drupal will list the nodes that have been imported:




***18.*** Choose **Structure** | **Views**. Choose **Add new view**. Create a new view named **Locations** showing **Content of type Franchise**. The display format should be **Extended GMap of teasers without links** and **without comments**. Click on **Continue & edit**:

***19.*** Under **Add fields**, select the **Content: Nid** option to add the Node ID field:

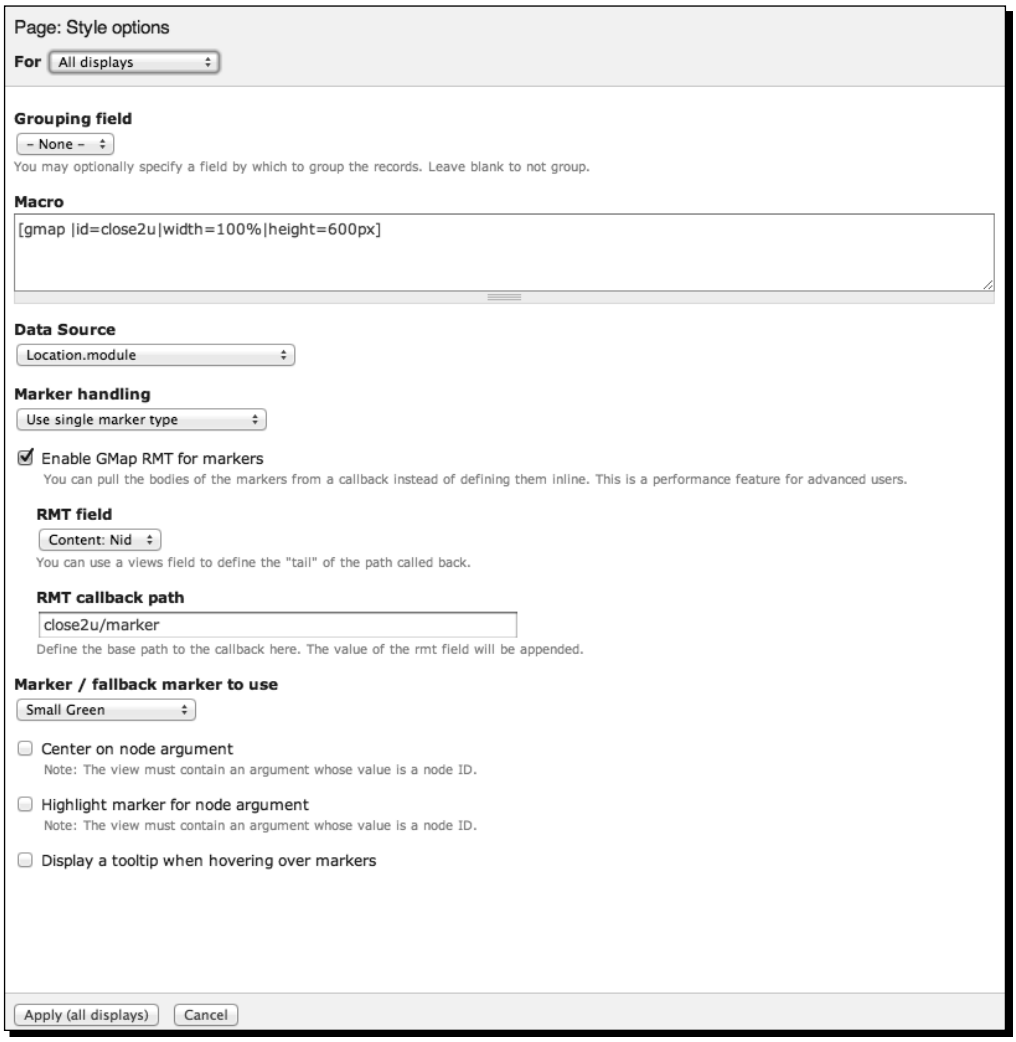



***20.*** Beside the **Format** line is the current format **GMap** and a **settings** link. Click on the **settings** link. In the **Macro** box, add the macro as follows:

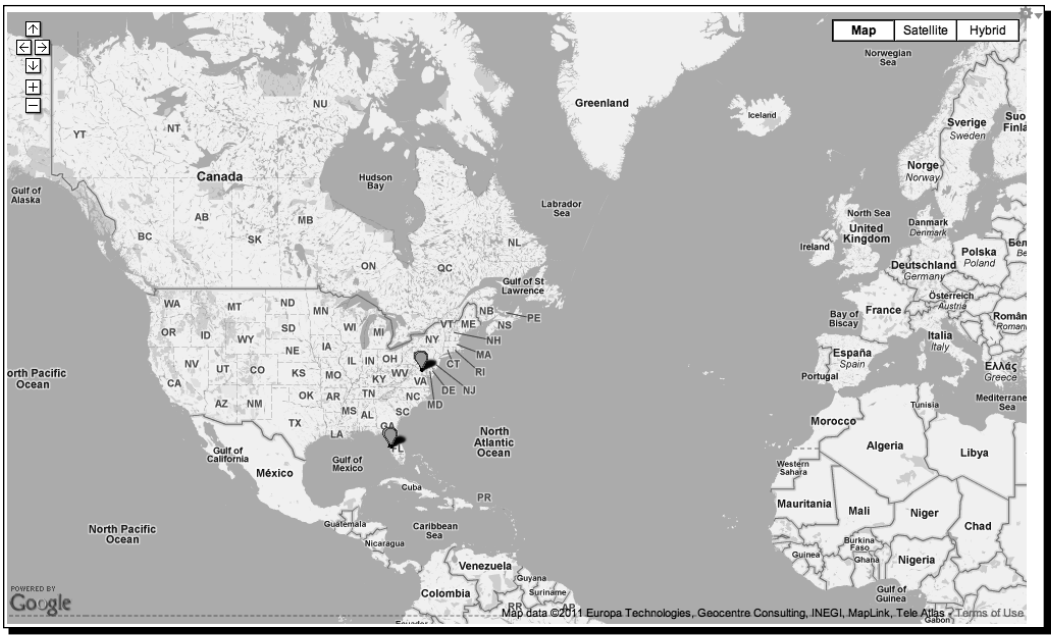
```
[gmap |id=close2u|width=100%|height=600px]
```

***21.*** Under **Data Source** select **Location.module**, and under **Marker Handling** select **Use single marker type**.

**22.** Under **RMT field** select **Content: NID**. Add a **RMT callback path** of **close2u/marker**. Click on **Apply (all displays)** and save the view:

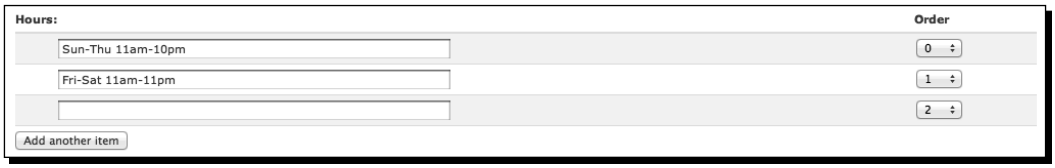



**23.** If all goes well, you should now be able to navigate to `http://dpk.local/locations` and have a proper Google map with markers for the listed locations. We'll address some of the map issues in the *Time for action – geocoding a node's location data* and *Time for action – downloading and enabling the close2u module* sections, but for now, your nodes are mapped:




## What just happened?

The first thing to notice is the way we've done the **Hours** field. Different locations have different hours. Some may need 3 lines to describe their regular hours, some may choose to put all 7 days of the week on a separate line. We can accommodate that by adding a text field and allowing unlimited values in that text field. When entering data for the **Hours** field, you can click on **Add another item** to allow multiple entries:

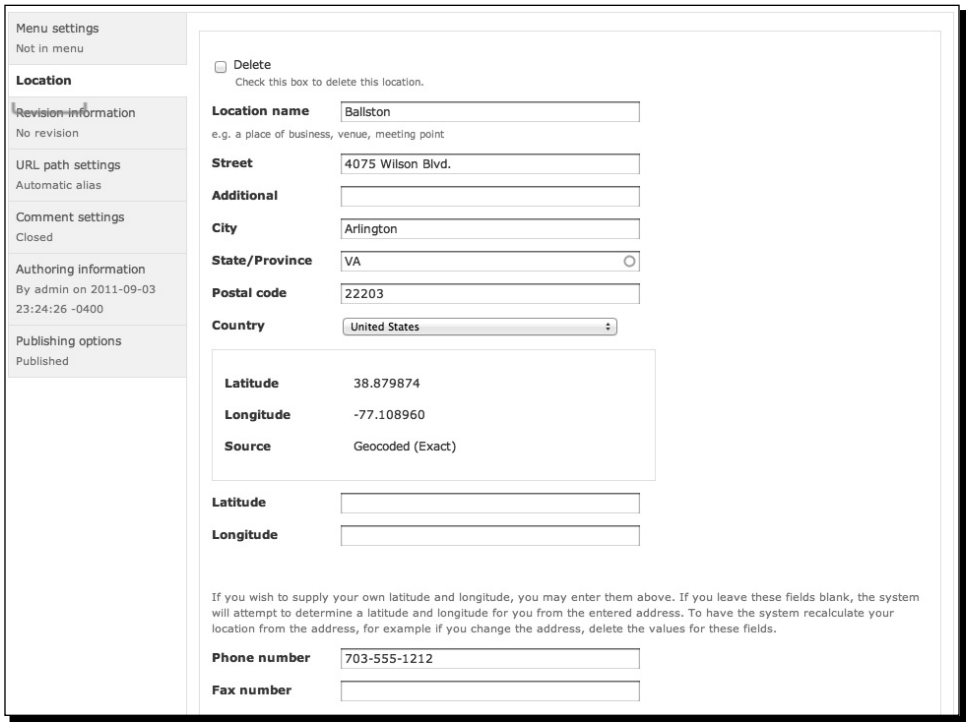



The **Location** module saves the address and geolocation data, and then associates that data with either a node or a user. You can enable the module to work with either. In this example, we've given the **Franchise** content type the ability to attach one or more locations to the nodes. We've then used a standard view and the **GMap** module to mash up those nodes into a GMap.

We've illustrated one of the simultaneously interesting and powerful things about Display Suite's code fields. Display Suite allows you to create customized fields that will execute the PHP code when a node is displayed. We created a Display Suite custom field that shows the address with a Google Maps link. We added the location title and the hours. It is also easy to white-screen your installation with a custom code that fails. Be very careful using this feature.

After we added the code field, we imported some sample data and then created a view that places the newly created nodes on a GMap. We set some values in the macro field of the GMap. These macros define the width and height of the map that is produced. There are other options you can add to determine the look and feel of the maps that the Google API produces. Those options are available in the Google API reference under the **google.maps. MapOptions object** section (`http://code.google.com/apis/maps/documentation/javascript/reference.html#MapOptions`). You should be able to set any map value from that command line using the syntax shown. One we specifically did not set was the MapCenter and the Zoom level. We'll set that a little later in this chapter, based on the user's location data.

If you take a look at one of the nodes we've imported and scrolled down to the node's location data, you'll see the address and a longitude and latitude value that allows Google to plot those coordinates on a map. We'll use those coordinates in the *Time for action – downloading and enabling the close2u module* section to gauge the distance from the user's location:



# From address to longitude and latitude

The addresses we've imported had longitude and latitude information encoded in their associated node locations. But unless you regularly list longitude and latitude coordinates with your addresses, you probably don't have this information at hand. You'll need to run the addresses through a process called **geocoding** which sends the address to a service such as Google Maps (or Yahoo! or Bing) and the map system returns a longitude and latitude coordinate for the given address.

There are multiple services that provide geocoding and nothing says you have to use Google or the module in this example. But for Drupal 7, at the time of this writing I found my options limited, so I wrote a companion project for the Drupal 7 **Location** module that I call **Location Geocode Update**. I hope by the time of this book's publication that this module will be a full-fledged Drupal project, but for now it's a sandbox project. Sandbox projects are projects developers create to try new things and use experimental code. The code in this module works for this example and I look forward to sharing it with the Drupal community at large as a full-fledged Drupal project.

**Playing in the Sandbox**

Drupal projects have several different phases. The first phase should be the sandbox mode. For the `geolocation` module and the `close2u` module I've created them as sandbox modules on `drupal.org`. You can find these at `http://drupal.org/sandbox/stovak/1262930` and `http://drupal.org/sandbox/stovak/1265648`. Use sandbox modules at your own risk. To install a project from the sandbox, you'll need to clone it from the GIT repository. Most sandbox modules have the GIT repository URL. Open the terminal window and change the directory to your project's `sites/all/modules/custom` directory. If your project doesn't have one, create it. Enter `git clone SANDBOX_URL`, where `SANDBOX_URL` is the GIT URL you obtained from the sandbox project listing. GIT will create a local copy of the sandbox. If the Drush commands to find the modules in this chapter fail, you can obtain the latest version by cloning the `drupal.org` sandbox. To do this, search for the module on `drupal.org` using the project name I've given you and clone the project locally. Let's get started!

# Time for action – geocoding a node's location data

Geocoding address data is a simple call to Google's API. I've encapsulated the API calls into a module that triggers on node save:

> Fake addresses will return incorrect geocoding data and should not be used for this example. We're testing the node's ability to add the longitude and latitude data without looking them up. Also, note that in order for Google geocoding to work correctly, you must provide a valid postal code for the given address.

1. Change the directory to your site's root directory and use Drush to install and enable the module with the following commands:

```
cd ~/Sites/dpk
drush dl location_geocode_update
drush pm-enable location_geocode_update
```

2. Navigate to **Modules** | **Admin** and enable the **Trigger** module from core if it's not already enabled.

3. Navigate to **Admin** | **Content** | **Location**. Check the **Enable JIT geocoding** checkbox.


4. Navigate to **Structure** | **Triggers** | **Node**. You will see several triggers listed with an action list of possible actions. Under the first trigger, **When either saving new content or updating existing content**, select the **Update Latitude/Longitude** action and click on **Assign.** The new action should appear in the list.

5. Navigate to **Content** | **Add content** | **Franchise**.

6. Name the franchise as **My Location**.

7. Add the hours as **9AM – 5PM Mon-Fri**.

8. Click on **Location** and add your address or an address you know is valid without looking it up. Save the node:


9. Drupal should take you to a full view of the newly added node. At the top, there will be a selected **View** tab as well as **Edit** and maybe **Export**, too. Edit the node and notice the address should now have longitude and latitude information:


### What just happened?

The **Triggers** module allows actions to be triggered (get it?) on predefined events for pieces of content. The Location Geocode Update module created a new action that we added to the node create/update trigger. This trigger will allow any node that allows the location data to be geocoded when saved or created.

## The close2u module

The Google map we created in the *Time for action – adding location data to nodes* section has a few problems. The first of which is the zoom magnification. It's way too far out. The second is that the center of the map is traditionally the user's location. The third is that the map itself doesn't have any user location data.

I mentioned earlier that the JavaScript `geolocation` object has been available in browsers for some time now. During the writing of this chapter, I couldn't actually find any Drupal module that used client geolocation data from the browser; so, I wrote this simple module that integrates with **Views**, **GMap** and the Drupal 7 **Location** modules to sort of re-imagine the "store locator" page that so many websites have. It's a good example of how to create a customized Drupal 7 module to solve many of the issues that exist with GMap and location integration in Drupal 7. As of the writing of this chapter, the module is a sandbox module. I hope to take it to full project status before the book is published. We'll step through the module's code after the exercise to show you how the magic happens.

Let's get started!

## Time for action – downloding and enabling the close2u module

The math to determine the distance between two objects on the earth is not simple. It's a complex formula. Luckily, that's all been worked out in advance for us and the code is in the `location` directory in a file called `earth.inc`. If you care to look that over, open up the file and take a look. If not, just trust that it's there. This module uses `earth.inc` to produce SQL that will sort objects by location:

1. Open a terminal window. Change the directory to your site root and use the `drush` command to download and enable the `close2u` module. Note the words at the beginning of the chapter with regards to sandbox modules.

   ```
   cd ~/dpk/
   drush dl close2u
   drush pm-enable close2u
   ```

2. First, navigate to **Structure** | **Views** and edit the **Locations** view we created in the *Time for action – geocoding a node's location data* section. Verify that the RMT values match the following screenshot. The **RMT field** value needs to be set to **Content: Nid** and the **RMT callback path** needs to be set to **close2u/marker**. Also of note, we're going to be working with the created GMap directly, so under **Macro**, make sure that the GMap ID value is `close2u`. Otherwise, our JavaScript won't be able to find the map:

Page: Style options

**For** All displays

**Grouping field**

– None –

You may optionally specify a field by which to group the records. Leave blank to not group.

**Macro**

```
[gmap |id=close2u|width=100%|height=600px]
```

**Data Source**

Location.module

**Marker handling**

Use single marker type

☑ Enable GMap RMT for markers

You can pull the bodies of the markers from a callback instead of defining them inline. This is a performance feature for advanced users.

**RMT field**

Content: Nid

You can use a views field to define the "tail" of the path called back.

**RMT callback path**

close2u/marker

Define the base path to the callback here. The value of the rmt field will be appended.

**Marker / fallback marker to use**

Small Green

☐ Center on node argument

Note: The view must contain an argument whose value is a node ID.

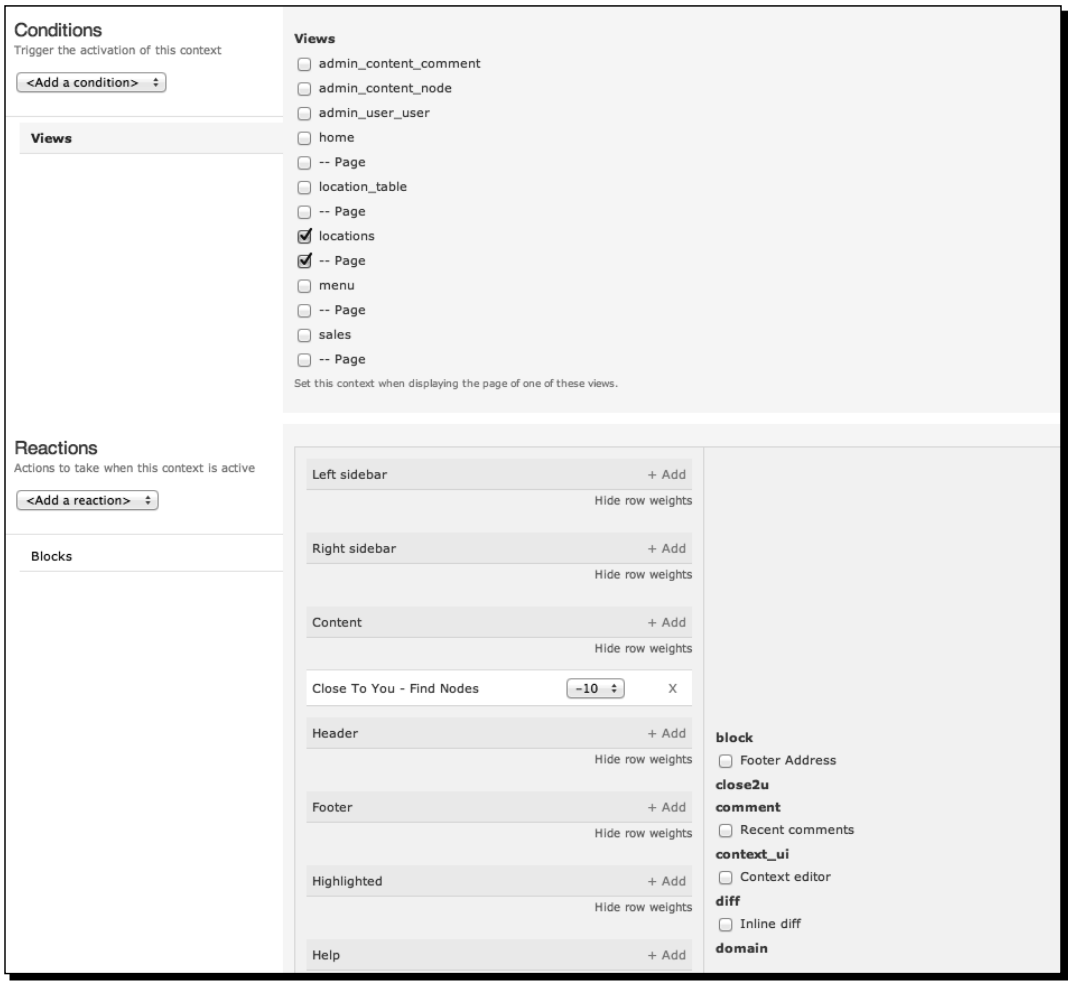
☐ Highlight marker for node argument

Note: The view must contain an argument whose value is a node ID.

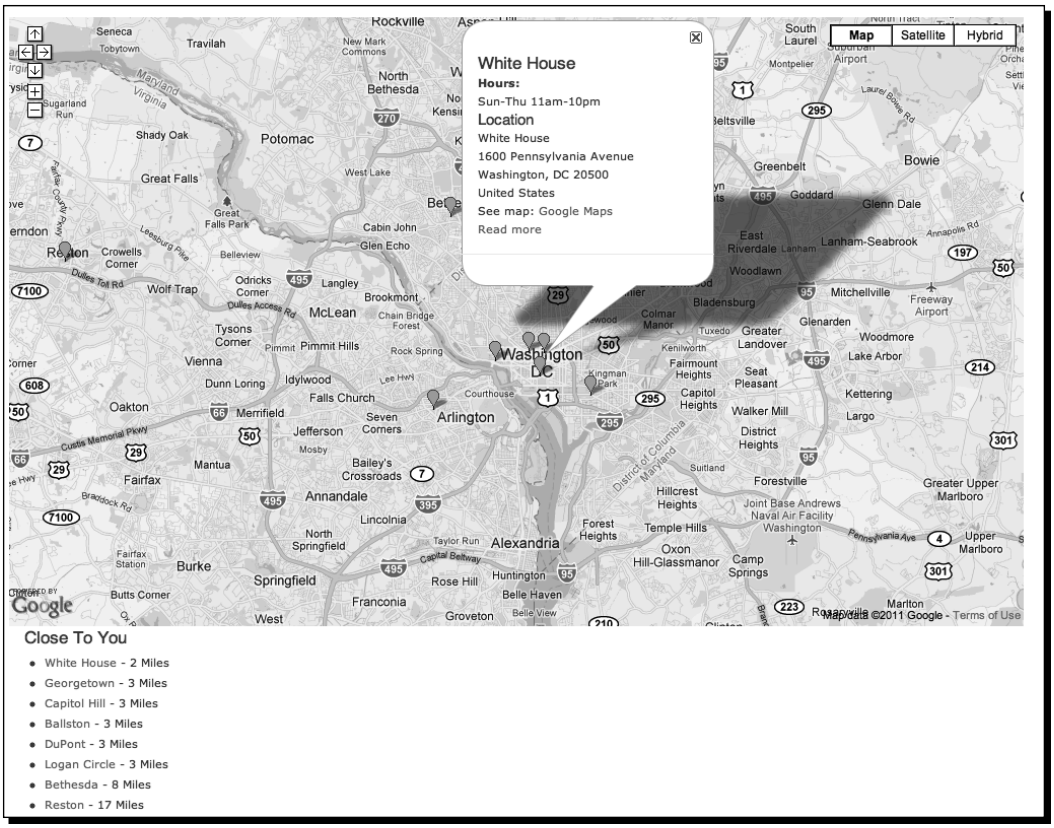
☐ Display a tooltip when hovering over markers

Apply (all displays) Cancel

3. Navigate to **Structure** | **Context** | **Add**. Title this context as **Locations**. Under **Conditions**, click on **Views** and then select the **locations** view we created earlier along with it's page display (**--Page**). Under **Reaction**, click on **Blocks**. Check the checkbox beside the **Close To You – Find Nodes** block, and besides **Content** click on **Add**. Save the context.




4. Navigate to the **Locations** view and see if it doesn't look a little friendlier:




## What just happened?

Let's take a look at what this `close2u` module does and how it interacts with our view.

First, the view creates a block that can be placed on any page. If you're used to Drupal 6's `hook_block` which provides all the information for a block in a single function, you should understand Drupal 7's block system. It's been split up into the actions required to define a block by adding the action to the hook name. For this block, we've created a `hook_block_info` that defines the blocks for the module and `hook_block_view`, which does the heavy lifting of putting the block together:

```
function close2u_block_info($delta = 0) {
 $blocks = array();
 $blocks['find_node'] = array(
   'info' => t('Close To You - Find Nodes'),
     'status' => 1,
   'cache' => DRUPAL_NO_CACHE,
 );
```

```
return $blocks;
}

function close2u_block_view($delta = 0) {
  $block = array();
switch ($delta) {
default:
      $block['subject'] = "Close To You";
      $block['content'] = close2u_page($delta);
  }
return $block;
 }
```

`close2u_block_view` calls a function `close2u_page` to do the action that initiates our block.

```
function close2u_page($delta) {
module_load_include('inc', 'uuid', 'uuid');
  $list_id = "close2u-" . uuid_generate();
drupal_add_js(array("close2u" => array("instances" => array($list_
id))), "setting");
drupal_add_js(drupal_get_path("module", "close2u") . '/close2u.js');
return theme("close2u_container", array("list_id" => $list_id, "delta"
=> $delta));
}
```

This function references the `uuid` module to create a unique ID for our block listing. Calling `drupal_add_js` with an array and a second argument of setting will add this `uuid` to our `Drupal.settings` object in JavaScript at the frontend. We'll use that in the file referenced in the next line, `close2u.js`. We then theme the container that will hold the list of nodes close to us. We've defined that the container in the hook theme and the file to generate the HTML in the templates folder.

```
function close2u_theme() {
  $path = drupal_get_path("module", "close2u") . "/templates";
  $items = array();

  $items['close2u_container'] = array(
    "template" => "close2u_container",
    "arguments" => array("uuid" => NULL, "delta" => NULL),
    "path" => $path,
  );
  $items['close2u_list_item'] = array(
    "template" => "close2u_list_item",
    "arguments" => array("result" => NULL),
```

```
    "path" => $path,
  );
return $items;
}
```

We've also defined a second theming function that we will use to theme the individual items. Our theme file in the `templates` folder is called `close2u_container.tpl.php`.

```
<div class="close2u-enter-location-container" style="display:none;">
  <form action="/close2u/address" method="get" accept-charset="utf-8">
    <label for="close2u-enter-location-text">
Enter an Address or Postal Code</label>
    <input id="close2u-enter-location-text" name="close2u-enter-
location-text" placeholder="Enter an Address or Postal Code">
    <p><input type="submit" value="find &rarr;"></p>
  </form>
</div>
<div class='close2u-container' id='<?php echo $list_id;?>' rel='<?php
echo $delta; ?>'></div>
```

It creates a container for our JSON call and contains a form to enter an address if our client-side geolocation fails.

The JavaScript file `close2u.js` defines the `Drupal.behavior` object that pulls all of this together. Let's take a look at it function-by-function. The basis for all Drupal 7 `Drupal.behavior` files are the `attach` and `detach` methods. We've talked about them in a previous chapter so I won't go over their function. Sufficient to say, the `attach` method gets everything started.

```
Drupal.behaviors.close2u = {
  attach: function(context) {
    if (Drupal.settings.close2u.origin == undefined) {
      if (navigator.geolocation) {
        Drupal.settings.close2u.origin = {
longitude: null,
latitude: null
};
        jQuery(Drupal.behaviors.close2u)
.bind("locationChange",
Drupal.behaviors.close2u.locationChangeHandler);
        navigator.geolocation.getCurrentPosition(
Drupal.behaviors.close2u.saveOrigin,
Drupal.behaviors.close2u.locationFail);
        Drupal.settings.close2u.watchId =
          navigator.geolocation.watchPosition(
Drupal.behaviors.close2u.saveOrigin);
```

```
        } else {
          Drupal.behaviors.close2u.locationFail();
        }
      }
    },
```

In order to determine distance, you need an origin and a destination. The destinations are, obviously, our node locations. The origin is the user. We'll be storing the user's origin in the `Drupal.settings` object. If that origin has not been defined, the function attempts to create it. If the browser supports geolocation, we define the latitude and longitude as null and bind a `location change` event responder to the `Drupal.behaviors.close2u` object. If this event is triggered, the `locationChangeHandler` function will attempt to handle the event. `navigator.geolocation.getCurrentPosition` is the client-side call that attempts to discern a location in the client's browser. The first function references execute if the attempt is successful, the second, if it is unsuccessful. We then put a `watchPosition` on the client's browser. If the client is using a handheld, it's helpful that we respond to their movements as they get closer or farther away. If the client's browser does not support geolocation, execute the same function as a failed geolocation call.

```
saveOrigin: function(position) {
    if (position) {
      Drupal.settings.close2u.origin = position.coords;
      Drupal.settings.close2u.origin.timestamp = position.timestamp;
      jQuery(Drupal.behaviors.close2u).trigger("locationChange");
    }
  },
```

The `saveOrigin` function responds to the `Navigator.geolocation` request. It receives a position object that has the longitude and latitude coordinates as well as some other information.

```
{
accuracy: 38
altitude: null
altitudeAccuracy: null
heading: null
latitude: 39.8624353
longitude: -76.0532586
speed: null
timestamp: 1315139386335
}
```

For some handhelds, you'll also get altitude, speed and other information about where you are. We store this information in the `Drupal.settings.close2u` object and then trigger a `locationChange` event on our behavior object. The `locationChange` event executes the `locationChangeHandler` function.

```
locationChangeHandler: function(evt) {
    if (Drupal.settings.close2u.origin.longitude != null && Drupal.
settings.close2u.origin.latitude != null) {
```

First, we make sure the longitude and latitude have values.

```
      jQuery.each(Drupal.settings.close2u.instances, function(idx,
value) {
```

We loop over the block instances for `close2u`. We store them in the `instances` object.

```
        url = jQuery("#"+value).attr("rel").replace(/_/g, "/");
        jQuery("#"+value).load("close2u/"+url, Drupal.settings.
close2u.origin, Drupal.behaviors.close2u.locationListHandler).
addClass("close2u-processed");
      });
```

For every instance, we execute a `jQuery.loadajax` request and give it the longitude and latitude from our origin. We'll see the response to this request in a minute. We dispatch this request and move on for it to load in the background.

```
      // gmap module integration
      if (Drupal.settings.gmap.close2u != undefined) {
        this.gmapObject = Drupal.gmap.getMap("close2u");
        //center and zoom
         this.gmapObject.map.setCenter(new GLatLng(Drupal.settings.
close2u.origin.latitude, Drupal.settings.close2u.origin.longitude));
        this.gmapObject.map.setZoom(11);
      }
```

We want to re-center the map so that the center is the user's location and then set the `setZoom` value to around `10` or `11`, which will give us a view of the metro area surrounding the user's location.

```
if (Drupal.settings.gmap.close2u != undefined &&
   (Drupal.behaviors.close2u.markers == undefined ||
   Drupal.behaviors.close2u.markers.length == 0)) {
  Drupal.behaviors.close2u.markers = {};
  for(i in Drupal.behaviors.close2u.gmapObject.vars.markers) {
  Drupal.behaviors.close2u.markers[Drupal.behaviors.close2u.
gmapObject.vars.markers[i].rmt] = Drupal.behaviors.close2u.gmapObject.
vars.markers[i];
        }
      }
    } else {
      jQuery(".close2u-enter-location").show();
    }
  },
```

When we created the view, we tell it to use the node ID as the RMT value for the pointer click. This node ID value is stored in every marker object on the page, but we can't just do a search for them. So we need an object that references the node ID so when the node IDs are clicked on the page, we can highlight the marker.

In the `close2u.module` file, we've set up a `hook_menu` that will handle the `.load` request for nodes close to the client.

```
function close2u_menu() {
  $items = array();
  $items['close2u'] = array(
    "page callback" => "close2u_page",
    "page arguments" => array(2),
    'access callback' => TRUE,
    "type" => MENU_CALLBACK,
  );
  $items['close2u/find/node'] = array(
    "page callback" => "close2u_find",
    "page arguments" => array(2),
    'access callback' => TRUE,
    "type" => MENU_CALLBACK,
  );
  $items['close2u/marker/%node'] = array(
    "page callback" => "close2u_marker_retrieve",
    "page arguments" => array(2),
    'access callback' => 'node_access',
    'access arguments' => array('view', 2),
    "type" => MENU_CALLBACK,
  );
return $items;
}
```

The `close2u/find/node` menu item will respond to our `jQuery.load` request by triggering the `close2u_find` function. Let's take this line by line.

```
//default search is nodes, but you can also search for users by type =
'uid'
function close2u_find($type = "node", $origin = NULL) {
  $toReturn = "<ul class='close2u-list'>";
if ($origin == NULL) {
if (isset($_REQUEST['longitude']) &&isset($_REQUEST["latitude"])) {
      $origin = $_REQUEST;
    }
elseif (property_exists($GLOBALS, "origin")) {
global $origin;
```

```
    }
else {
drupal_set_message("in order to find something close to you, I must
have an origin. Set \$GLOBALS\[\'origin\'\] or use as second argument
to close2u_find.", "error");
return FALSE;
    }
  }
```

We begin the list to be returned, and we set the origin, if it's not already set.

```
  $query = db_select("location", "l")->fields("l", array("lid",
"longitude", "latitude"));
```

Using Drupal 7's new database API, we create a database query object that will get details of the `location` field:

```
module_load_include("inc", "location", "earth");
  $query->addExpression(earth_distance_sql($origin['longitude'],
$origin['latitude']), "distance");
```

Calculating the distance is not as simple as subtracting one value from the other. The earth is a sphere and we calculate distance along the outside of that globe in a unit called **radians**. There's some complex math at work so we add a calculated field called distance, with the math contained in the earth functions of the location module. The `earth.inc` include has all of that worked out in advance for us.

```
  $query->join("location_instance", "li", "li.lid = l.lid");
  $query->fields("li");
  $max_distance = (array_key_exists($origin['max_distance']) ?
    $origin['max_distance'] : variable_get("close2u_default_max_
distance", NULL)
  );
if ($max_distance != NULL) {
    $query->having("distance < :max_distance ", array(
        ":max_distance" => $max_distance,
      ));
  }
```

We can add a `max_distance` to the request if we want to exclude nodes that are over a certain distance from the origin. For this experiment, I have the default `max_distance` set to `null`. Eventually, we should write a `hook_block_config` function that sets the system variable, `close2u_default_max_distance` and by the time of publication of this book, the module may have that. But for development purposes, we always want to show something in the results no matter how far it is from the user.

```
  $query->orderBy("distance");
```

Sort the returned objects by the calculated distance.

```
$query->range(0, 20);
```

Limit the results to `20` nodes.

```
$foreign_alias = substr($type, 0, 1);
$foreign_key   = $foreign_alias . "id";
$join_clause   = "li." . $foreign_key . " = " . $foreign_alias . "."
. $foreign_key";
$query->join($type, $foreign_alias, $join_clause);
```

So, this is a little complicated. We want to use a query that will work for either users or nodes so we create a join based on the word "node" and do a SQL join on the nodes table to get the node ID (`nid`).

```
$results = $query->execute();
```

The preceding command executes the query.

```
while ($result = $results->fetchObject()) {
    $result->node = node_load($result->nid);
    $toReturn .= theme("close2u_list_item", array("result" =>
$result));
  }
```

Retrieve and theme the results.

```
$toReturn .= "</ul>";
echo $toReturn;
exit();
}
```

Close the list, print the list, and echo the results. We then want to stop execution because we don't need the entire page's HTML, just the list itself.

In our JavaScript file, we process the incoming list when it loads:

```
locationListHandler: function(evt) {
    jQuery(".close2u-list-item")
      .not(".close2u-list-item-processed")
      .find("a.close2u-click-marker")
      .click(Drupal.behaviors.close2u.locationListItemClickHandler)
      .attr("href", "javscript:;")
      .addClass("close2u-list-item-processed");
    //first in list should be closes, click it.
    jQuery(".close2u-list-item:first-child a").click();
},
```

We select the list items that have not yet been processed. Select the link inside the list item and on click, give them a function to handle the click. We remove the direct link to the location's node and trigger the click action of the first one, or rather the one that's closest to the user's location.

```
locationListItemClickHandler: function(evt) {
    if(evt) evt.preventDefault();
    google.maps.Event.trigger(Drupal.behaviors.close2u.
markers[jQuery(this)
      .parent().attr("rel")].marker, "click");
    return false;
  },
```

In any event handler, the function's arguments are an event object. For this event, we want to prevent the default action and substitute our own triggers. When a location is clicked, we trigger the event of its complimentary marker being clicked. The Google Map fires another AJAX request to the `close2u` module that we've described in the **RMTcallback path** setting (**close2u/marker**) and append the node ID to that request. The request will be `close2u/marker/32` for node ID as 32. The receiver of that function is very simple:

```
function close2u_marker_retrieve($node) {
  echo drupal_render(node_view($node, "marker"));
  exit();
}
```

Grab the Node ID from the URL and view it in the `marker` build mode. Wait! There is no `marker` build mode! Well that's defined by the last two functions in the module:

```
function close2u_ctools_plugin_api() {
list($module, $api) = func_get_args();
if ($module == "ds" && $api == "ds") {
return array("version" => "1");
  }
}
function close2u_ds_view_modes_info() {
  $export = array();

  $ds_view_mode = new stdClass;
  $ds_view_mode->api_version = 1;
  $ds_view_mode->view_mode = 'marker';
  $ds_view_mode->label = 'Marker';
  $ds_view_mode->entities = array(
    'node' => 'node',
  );
  $export['mobile'] = $ds_view_mode;

return $export;
}
```

These two functions implement a new Display Suite build mode called `Marker`. The first function tells CTools which version of the Display Suite API to use. The second creates the build mode and returns it to Display Suite for using on every node. Every node will now be able to have its own "marker" display in Display Suite. If there's no specific marker display, Display Suite will show the default build.

# Finishing the page

We still have a couple of lingering issues with the page. The first of which is that we haven't really coded any alternative to geolocation. What about browsers that don't support the `navigator.geolocation` object or people in witness protection programs that ask you to click **Don't Allow** when you give them a location dialog? We have to plan for not being able to use the geolocation in the browser. Let's create that code together now.

## Time for action – finding the closest franchise the hard way

There are multiple reasons to code for situations where the location object is not available. Older versions of Internet Explorer and some marginal browsers do not support it. But there will also be times when the location is found incorrectly or you may want to find franchise locations that are not near the the computer or the handheld's current location. In these cases, the dependance on this feature is an inconvenience to the user. We need to write the code for that use case:

1. Create a form in the `templates/close2u_container.tpl.php` file to handle user input of location data. Add the following lines to the top of the template:

```
<div class="close2u-enter-location-container"
style="display:none;">
  <form action="/close2u/address" method="get" accept-
charset="utf-8" id="close2u-enter-location" name="close2u-enter-
location">
    <label for="close2u-enter-location-text">Enter an Address or
Postal Code</label>
    <input id="close2u-enter-location-text" name="close2u-enter-
location-text" placeholder="Enter an Address or Postal Code">
    <input type="hidden" name="list_id" value="<?php echo $list_
id;?>" id="list_id" class="close2u-enter-location-list-id">
    <p><input type="submit" value="find &rarr;"></p>
  </form>
</div>
```

2. Edit the `close2u.js` file as follows. Delete the `alert("location fail!");` line and add the following lines to the `locationFail` function to show the location form. Bind a `submit` event to the form so it can be submitted via AJAX:

```
    jQuery(".close2u-enter-location-container")
      .show()
      .find("form")
      .submit(Drupal.behaviors.close2u.userEnterLocationHandler);
  },
```

3. Edit the `close2u.js` file as follows. Create a `userEnterLocationHandlerfunction` to handle user-submitted location data and send it to the `close2u` module:

```
  userEnterLocationHandler: function(evt) {
    if (evt) evt.preventDefault();
    jQuery.getJSON(jQuery(this).attr("action"), jQuery(this).
serialize(), Drupal.behaviors.close2u.saveOrigin);
    return false;
  }
```

4. Create an entry in the `hook_menu` for the function that will mimic the call to the `geolocation` object:

```
function close2u_menu() {
  $items = array();
  $items['close2u'] = array(
    "page callback" => "close2u_page",
    "page arguments" => array(2),
    'access callback' => TRUE,
    "type" => MENU_CALLBACK,
  );
  $items['close2u/find/node'] = array(
    "page callback" => "close2u_find",
    "page arguments" => array(2),
    'access callback' => TRUE,
    "type" => MENU_CALLBACK,
  );
  $items['close2u/marker/%node'] = array(
    "page callback" => "close2u_marker_retrieve",
    "page arguments" => array(2),
    'access callback' => 'node_access',
    'access arguments' => array('view', 2),
    "type" => MENU_CALLBACK,
  );
  $items['close2u/address'] = array(
```

```
        "page callback" => "close2u_address_entry",
        "page arguments" => array(2),
        'access callback' => TRUE,
        "type" => MENU_CALLBACK,
  );
return $items;
}
```

**5.** Edit the `close2u` module. Create the function to mimic the call to the `geolocation` object:

```
function close2u_address_entry() {
  module_load_include("module", "gmap", "gmap");
module_load_include("inc", "location", "geocoding/google");
  $response = google_geocode_location(array("street" => $_
REQUEST['close2u-enter-location-text']));
  if (is_array($response)) {
    jsonjsonechojson_encode(array("coords" => array("longitude"=>
(float)$response['lon'], "latitude" => (float)$response['lat']),
"timestamp" => time(), "list_id" => $_REQUEST['list_id']));
  } else {
    jsonjsonechojson_encode(array("error" => "<h1>Google is unable
to find the location you entered.</h1>"));
  }
  exit();
}
```

**6.** Alter the `saveOrigin` function in `close2u.js` to handle error messages:

```
  saveOrigin: function(position) {
    if (position) {
      if (position.error != undefined) {
        $("#"+position.list_id).html(position.error);
      } else {
        Drupal.settings.close2u.origin = position.coords;
        Drupal.settings.close2u.origin.timestamp = position.
timestamp;
        jQuery(Drupal.behaviors.close2u).
trigger("locationChange");
      }
    }
  },
```

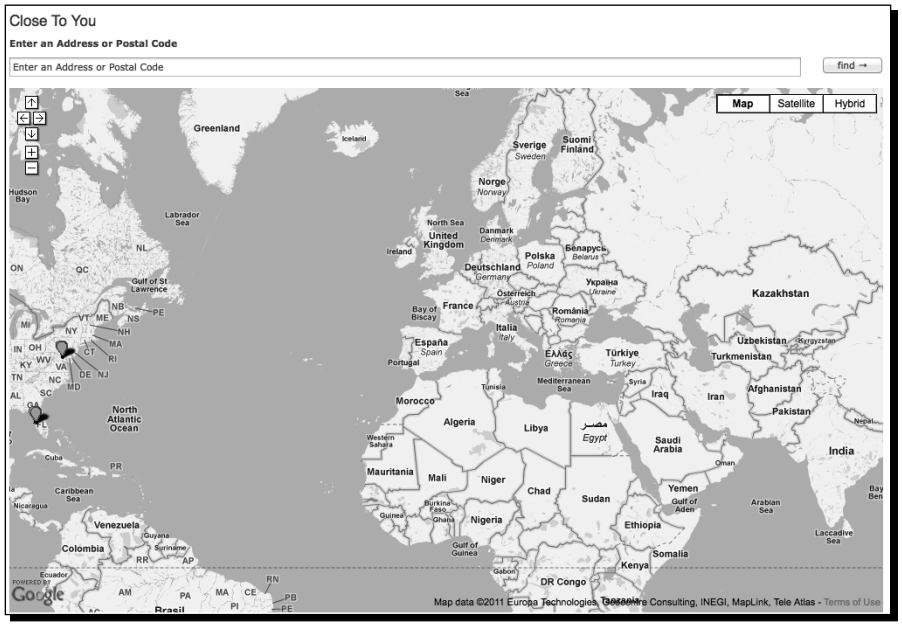
***7.*** Edit the `sites/all/themes/dpk/css/styles.css` file and add the following lines:

```
.close2u-list {
    -moz-column-count: 4;
    -moz-column-gap: 1em;
    -webkit-column-count: 4;
    -webkit-column-gap: 1em;
    column-count: 4;
    column-gap: 1em;
  }
```

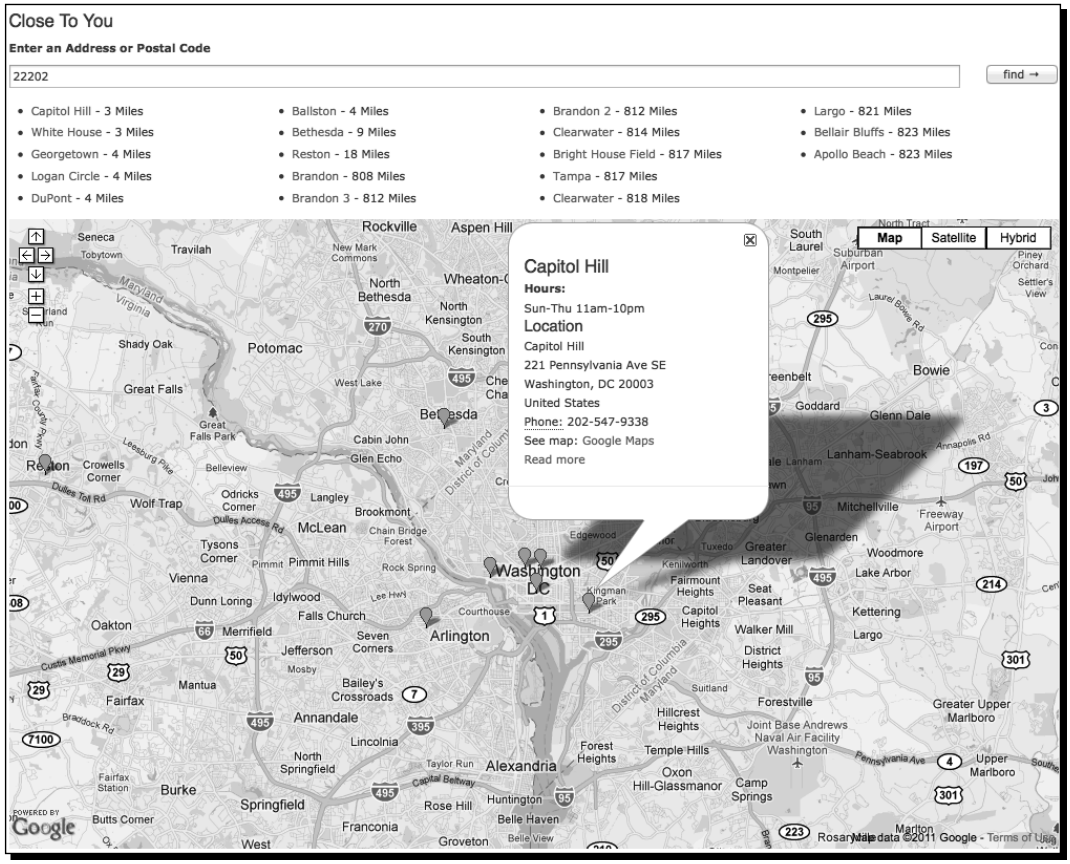
***8.*** Edit the `sites/all/themes/dpk_mobile/css/global.css` and add the following lines:

```
.close2u-list {
    -moz-column-count: 2;
    -moz-column-gap: 1em;
    -webkit-column-count: 2;
    -webkit-column-gap: 1em;
    column-count: 2;
    column-gap: 1em;
  }
```

***9.*** Navigate to `http://dpk.local/locations`. If the browser asks if you want to allow it to use geolocation, click on **Don't Allow**. You should see something like what is seen in the following screenshot:



***10.*** Enter **22202** into the **Enter an Address or Postal Code** text field and click on **find**:




## *What just happened?*

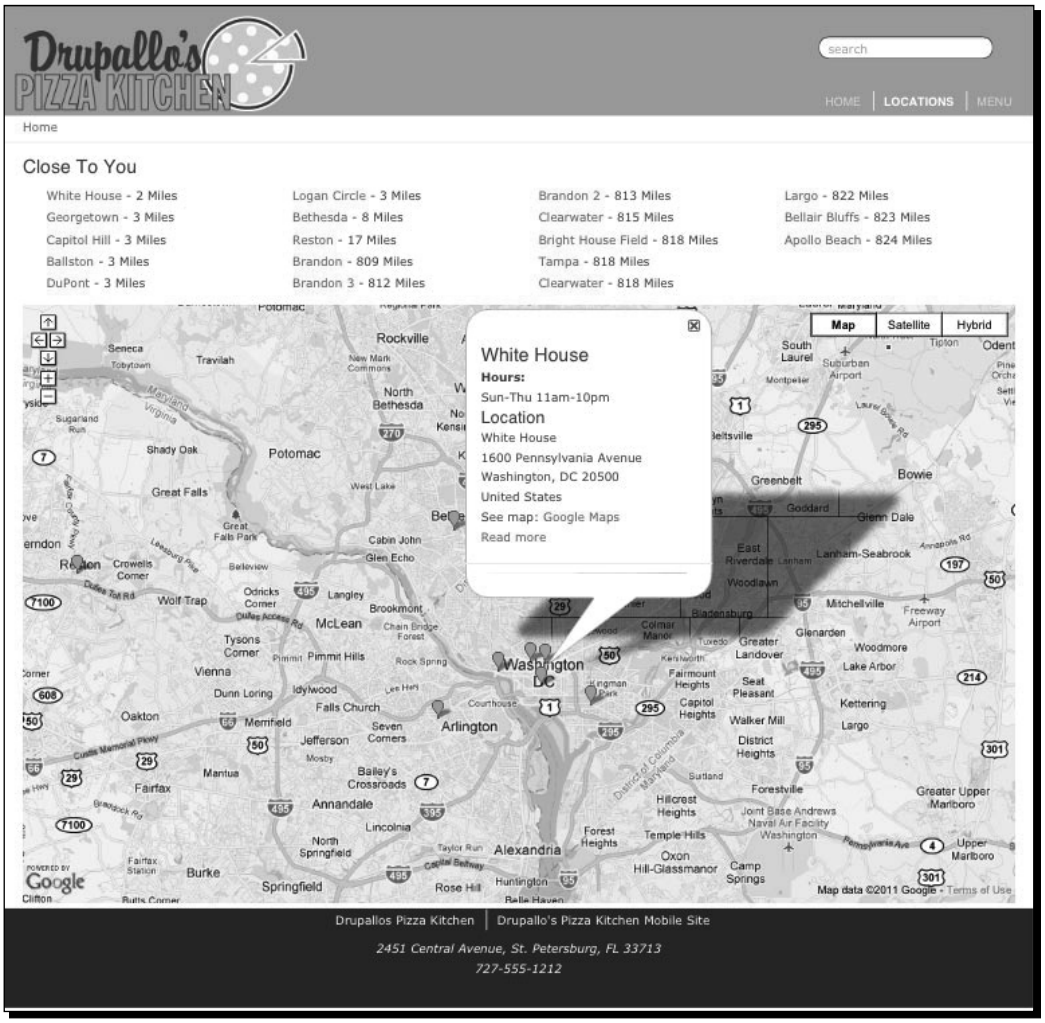
In the first step, we created a standard HTML form to send address data to the module for use in geocoding. In the second step, we intercepted the form's `submit` event and submitted the form via AJAX in exchange for a JSON object.

Step 4 creates a `hook_menu` that will call a function, `close2u_address_entry`. In the function, in step 5, we mimic the activity of the `geolocation` object by returning an object with its longitude and latitude values. If there's a problem with geocoding the address, we get an error.

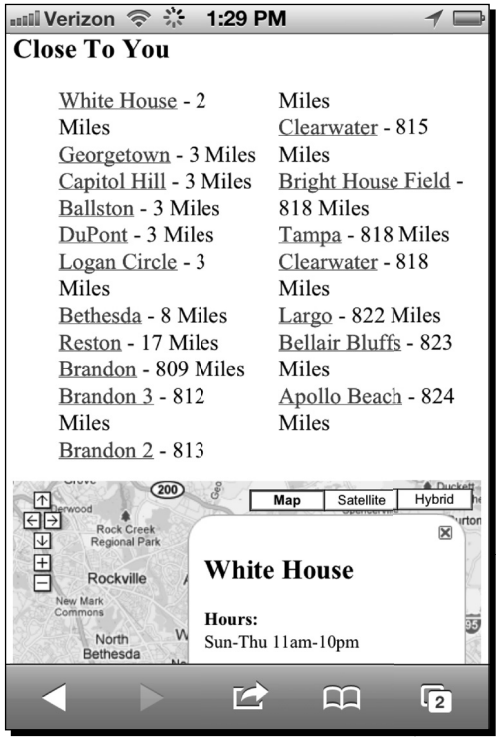
In step 6, we alter the `saveOrigin` JavaScript function to handle error objects as well as geolocation responses.

Finally, we added some CSS code that adds columns to the list of responses. It cleans up the look a little bit. It adds four columns for the desktop version and two columns for the handheld.

Taking a look at this page on your mobile device you can see how well everything we've done works with mobiles. If you click the Google Map links, the links should open in the mobile device's default maps application. The following screenshot is of the desktop version:



The following screenshot is of the mobile version:




## Pop quiz

1. The JavaScript object that allows you to read a browser's position is:
   a. `location.storage`
   b. `navigator.geolocation`
   c. `window.location.href`
   d. `document.location.href`
2. The module that allows you to map locations returned in a view is:
   a. `views_location`
   b. `gmap`
   c. `google_maps`
   d. `locale`

3. Address-based locations must first have longitude and latitude added to their data to place the items on a map:
    a. True
    b. False
4. The process of adding longitude and latitude to an address is called:
    a. Mapping
    b. Geocoding
    c. Location sharing
    d. Spelunking
5. You can use PHP to display a field in Display Suite by adding a:
    a. PHP file to the module
    b. New template
    c. Code field
    d. None of the above
6. To calculate the distance between two objects:
    a. You simply subtract the longitude and latitude values of one from the other
    b. Distance cannot be calculated by any method
    c. Use a complex set of radian math to figure the distance given the curve of the earth
    d. None of the above

# Summary

In this chapter, we've learned a lot about the **Location** and **GMap** modules and how to use them as building blocks to create a rich mobile (and desktop) user experience. We added location information to node objects and then produced a view of those objects. We've geocoded addresses and learned how to get by, when automatic geolocation isn't available to the client.

In the next chapter, we'll take a look at the **Services** module and other ways to retrieve data from Drupal with API calls.

## Where to buy this book

You can buy Drupal 7 Mobile Web Development Beginner's Guide from the Packt Publishing website: `http://www.packtpub.com/drupal-7-with-mobile-tablet-devices-web-development-beginners-guide/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.