9
Using Google Maps

In this chapter we will cover:

- Creating a geographic chart with the Google Visualization API
- Obtaining a Google API key
- Building a Google map
- Adding markers and events
- Customizing controls and overlapping maps
- Redesigning maps using styles

Introduction

This chapter will be dedicated to exploring some of the features available on Google Maps to get us ready to work with mapping in general. Mapping on its own isn't data visualization, but after we establish our base by understanding how to work with maps, we will have a very stable background that will enable us to create many cutting-edge, cool projects by integrating data and data visualization.

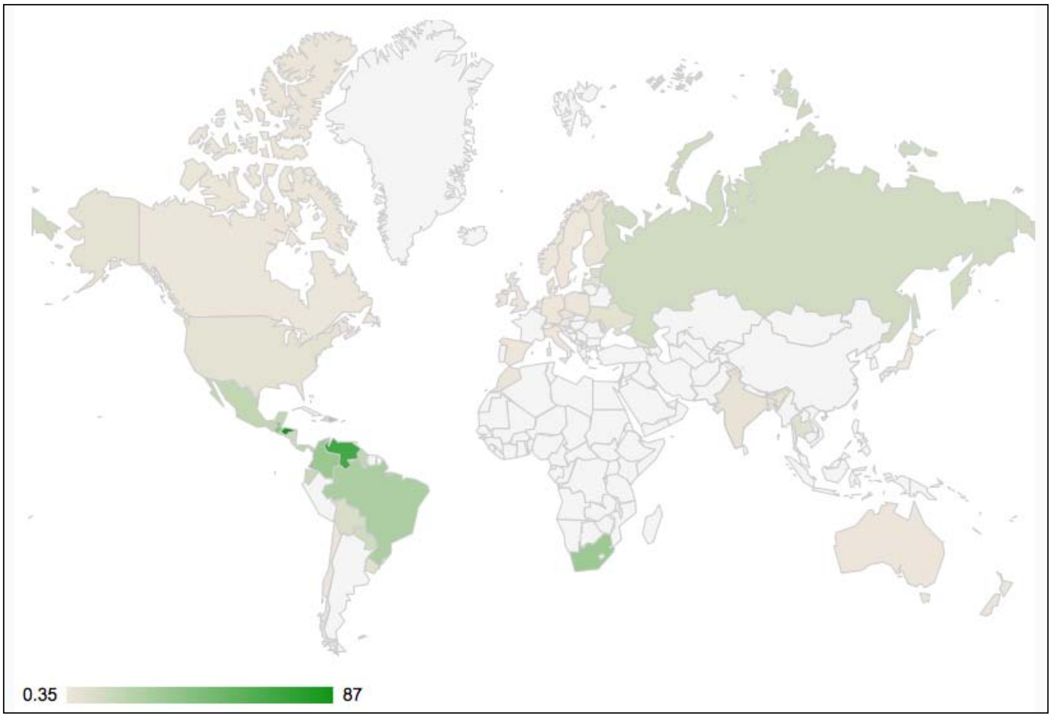
In this chapter, we will explore the main ways to create maps in the Google sphere.

Creating a geographic chart with Google Visualization API

In our first recipe for this chapter, we will start working with a vector-based map of the world. We will use it to highlight countries based on a data feed. In our case, we will use Wikipedia's list of countries as per the intentional homicide rate (latest numbers).

To view this raw data go to `http://en.wikipedia.org/wiki/List_of_countries_by_intentional_homicide_rate`.

Our goal will be to have a map of the world highlighted with a range of colors according to the number of intentional homicides per 100,000 people. As of the latest data in 2012 according to Wikipedia, it sounds like the most unsafe place to live in is Honduras—if you don't want to be intentionally killed—while you should feel really safe from intentional killing in Japan. How is your country doing? Mine is not that bad. I should probably avoid my local news stations that make me feel like I'm living in a war zone.



Getting ready

There isn't much that you need to do. We will be using the Google Visualization API for creating a geographic chart.

How to do it...

We will create a new HTML and a new JavaScript file and call them `08.01.geo-chart.html` and `08.01.geo-chart.js`. Follow these steps:

1. In the HTML file add the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Geo Charts</title>
    <meta charset="utf-8" />
    <script src="http://www.google.com/jsapi"></script>
    <script src="./08.01.geo-chart.js"></script>
  </head>
  <body style="background:#fafafa">
    <div id="chart"></div>
  </body>
</html>
```

2. Let's move to the `js` file. This time around we will want to request from the Google Visualization package the `geochart` functionality. To do that we will start our code, as follows:

```
google.load('visualization','1',{'packages': ['geochart']});
```

3. We'll then add a callback that will trigger the `init` function when the package is ready:

```
google.setOnLoadCallback(init);
function init(){
 //...
}
```

4. Now it's time to add the logic within the `init` function. In the first step, we will format the data from Wikipedia to another format that will work for the Google Visualization API:

```
var data = google.visualization.arrayToDataTable([
    ['Country','Intentional Homicide Rate per 100,000'],
    ['Honduras',87],['El Salvador',71],['Saint Kitts and
    Nevis',68],
    ['Venezuela',67],['Belize',39],['Guatemala',39],
    ['Jamaica',39],
    ['Bahamas',36],['Colombia',33],['South Africa', 32],
    ['Dominican Republic',31],['Trinidad and
    Tobago',28],['Brazil',26],
    ['Dominica', 22],['Saint Lucia',22],
    ['Saint Vincent and the Grenadines',22],
    ['Panama',20],['Guyana',18],['Mexico',18],
    ['Ecuador',16],
    ['Nicaragua',13],['Grenada',12],
    ['Paraguay',12],['Russia',12],
    ['Barbados',11],['Costa Rica',10 ],['Bolivia',8.9],
    ['Estonia',7.5],['Moldova',7.4],['Haiti',6.9],
    ['Antigua and
    Barbuda',6.8],['Uruguay',6.1],['Thailand',5.3],
    ['Ukraine',5.2],['United States',4.7 ],
    ['Georgia',4.1],['Latvia',4.1 ],
    ['India',3.2],['Taiwan',3.0 ],['Bangladesh',2.4 ],
    ['Lebanon',2.2],
    ['Finland',2.1 ],['Israel', 2.1],
    ['Macedonia',1.94 ],['Canada',1.7],
    ['Czech Republic',1.67],
    ['New Zealand',1.41],['Morocco',1.40 ],
    ['Chile',1.33],
    ['United Kingdom',1.23 ],['Australia',1.16],
    ['Poland',1.1 ],['Ireland',0.96 ],
    ['Italy',.87 ],['Netherlands',.86 ],
    ['Sweden',.86],
    ['Denmark',.85],['Germany',.81 ],['Spain',0.72],
    ['Norway',0.68],['Austria',0.56],['Japan',.35]
]);
```

5. Let's configure our chart options:

   ```
   var options = {width:800,height:600};
   ```

6. Last but definitely not the least, let's create our chart:

   ```
    var chart = new google.visualization.GeoChart(document.
   getElementById('chart'));
     chart.draw(data,options);
   }//end of init function
   ```

When you load the HTML file, you will find the countries of the world with highlighted colors that reflect the homicide rate. (We don't have a full list of all the countries of the world, and some countries are so small that it will be hard to find them.)

How it works...

The logic of this recipe is very simple, so let's quickly dash through it and add some extra features. As in all the other visualization charts, there are three separate steps:

- Defining the data source
- Setting up the chart
- Drawing the chart

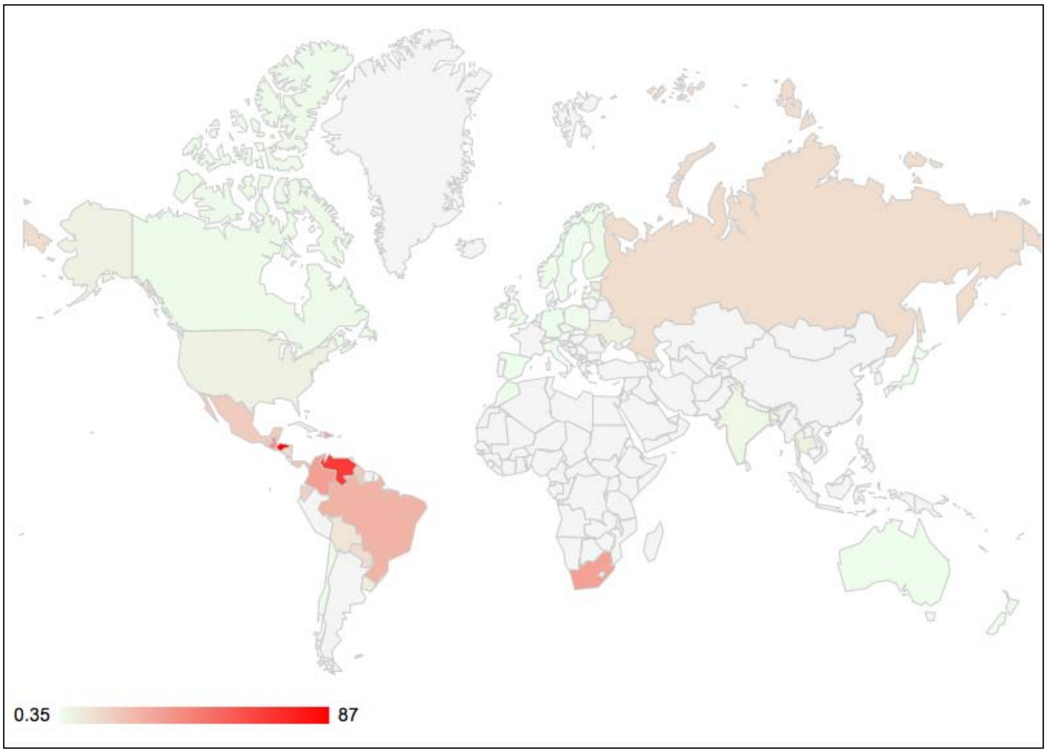
Not all countries are alike. If you are having issues while working with a country that is outlined, search through the latest Google documents on the supported countries. For the full list, you can check the sheet available at `http://gmaps-samples.googlecode.com/svn/trunk/mapcoverage_filtered.html`.

There's more...

Let's add some extra customization to our chart. As with all Google Visualization library elements, we can control many visuals through the `options` object.

The green color highlighted in our map just seems wrong. You would think that the less the killings the greener a country would be, so where the killings are more, a darker shade of red would be more appropriate. So let's change the colors by updating the `options` object:

```
var options = {width:800,height:600,
  colorAxis: {colors: ['#eeffee', 'red']}
    };
```

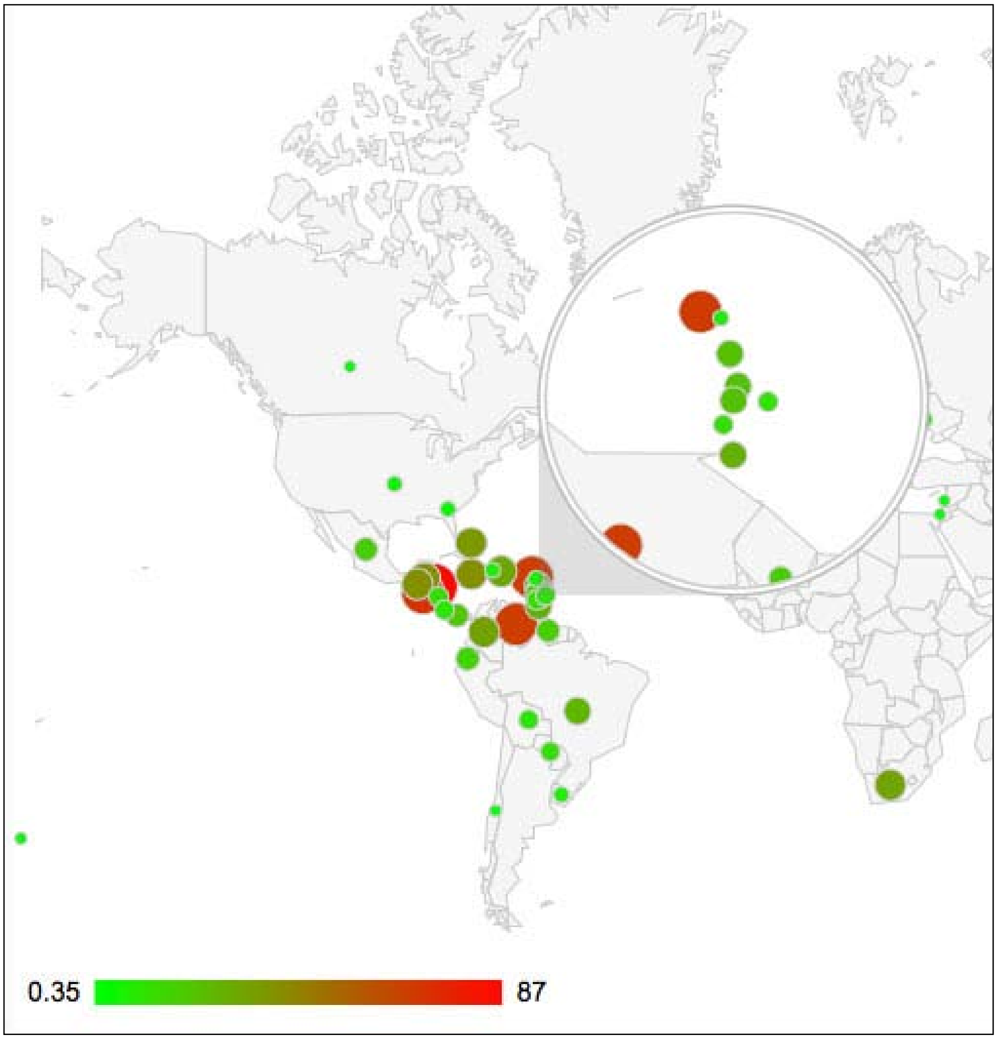


Making smaller areas more visible

To solve the problem of really small invisible countries, we can switch our rendering to be marker based. Instead of highlighting the land itself, we can switch to a marker-based rendering mode:

```
var options = {width:800,height:600,
    displayMode: 'markers',
        colorAxis: {colors: ['#22ff22', 'red']}
      };
```

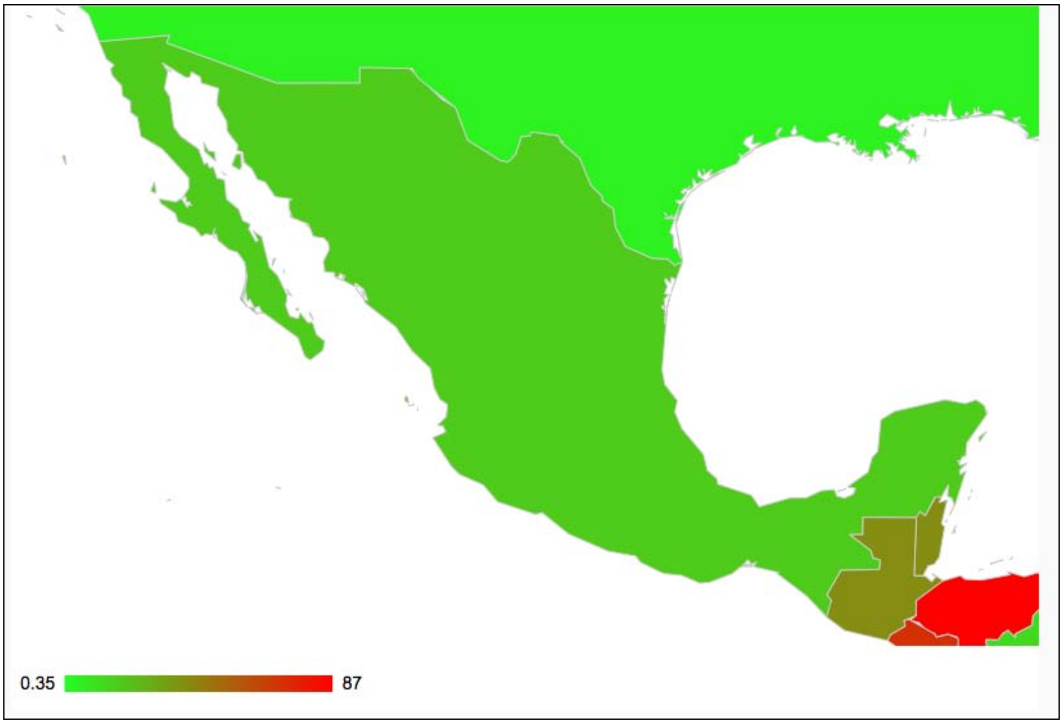
By default when rendering visualization maps with markers, when you roll over condensed areas, a highlight zoom view will help create a clearer view:



Another option instead would be to zoom into the area (we can do both or just zoom in). To zoom into an area we will use this code:

```
var options = {width:800,height:600,
  region:'MX',
      colorAxis: {colors: ['#22ff22', 'red']}
    };
```

To find out the list of possible values, refer back to the list of countries from earlier in this chapter. In this case we are zooming into the `MX` region:



That covers the basics of working with a geographical chart. For more information on working with the Google Visualization API, please refer back to *Chapter 8*, *Playing with Google Charts*.

Obtaining a Google API key

To work with most of the Google APIs, you must have a Google API key. As such we will go through the steps involved in getting a Google API key.

Google API has certain limitations and constraints. Although most of the APIs are free to use for small- to medium-sized sites, you are bound by some rules. Please refer to each library for its rules and regulations.

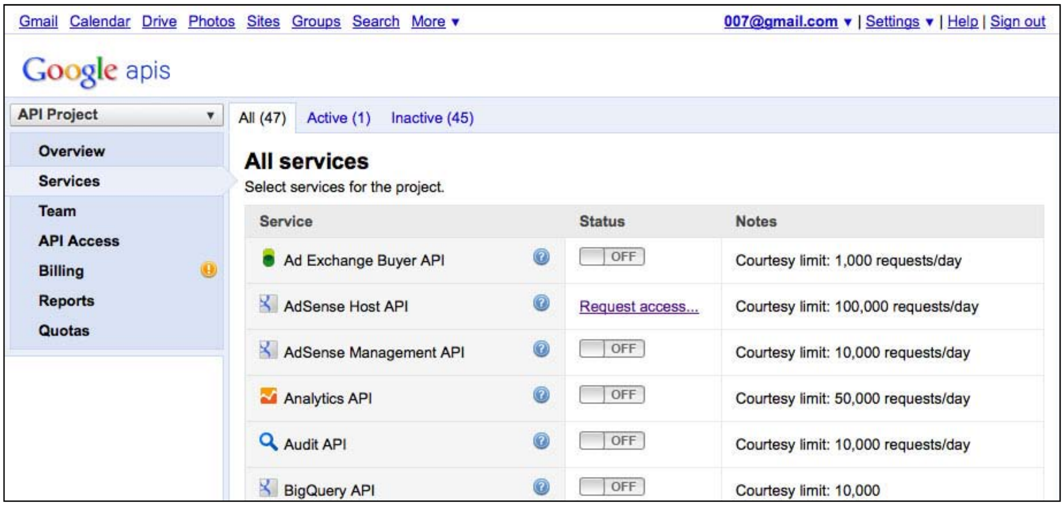
Getting ready

To get through this recipe you must have a Google ID; if you do not have one, you will need to create one.

How to do it...

Let's list the steps required to gain access to the Google API:

1. Log in to the API console at `https://code.google.com/apis/console`.
2. Select the **Services** option from the left-hand side menu:

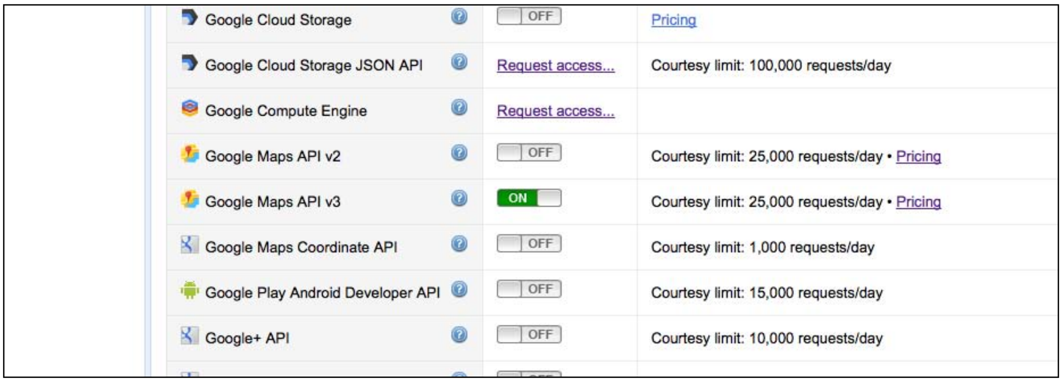


Gmail Calendar Drive Photos Sites Groups Search More ▾
007@gmail.com ▾ | Settings ▾ | Help | Sign out
Google apis
API Project ▾
Overview
Services
Team
API Access
Billing
Reports
Quotas
All (47) Active (1) Inactive (45)

All services
Select services for the project.

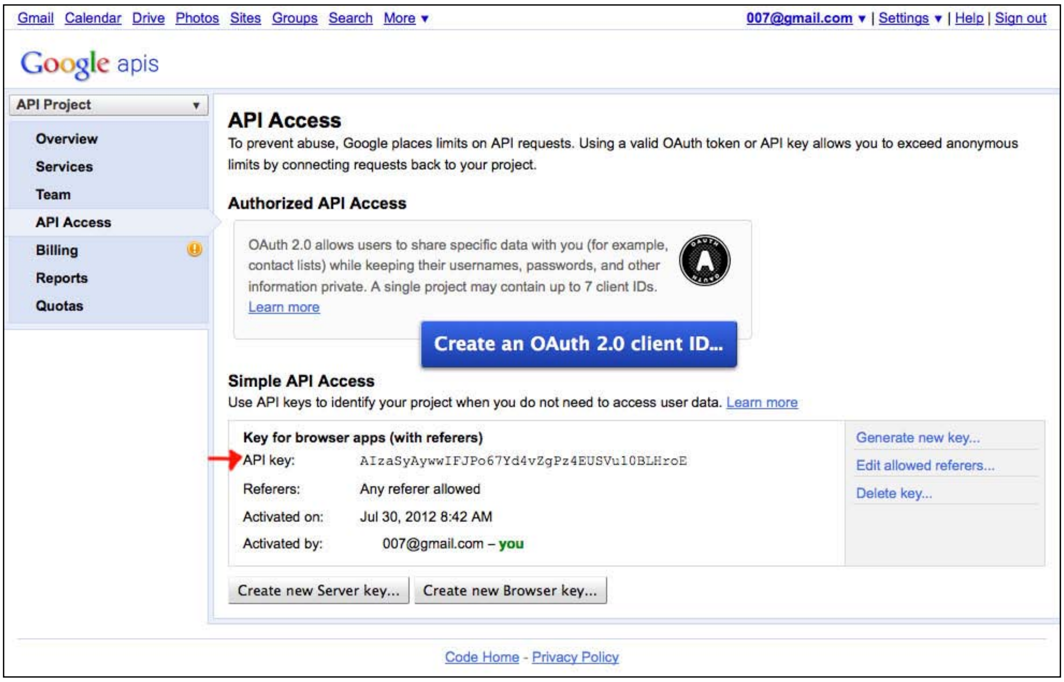
Service	Status	Notes
Ad Exchange Buyer API	OFF	Courtesy limit: 1,000 requests/day
AdSense Host API	Request access...	Courtesy limit: 100,000 requests/day
AdSense Management API	OFF	Courtesy limit: 10,000 requests/day
Analytics API	OFF	Courtesy limit: 50,000 requests/day
Audit API	OFF	Courtesy limit: 10,000 requests/day
BigQuery API	OFF	Courtesy limit: 10,000

3. Activate the API that you want to use (for example, in the next recipe *Building a Google map* we will use the Google Maps API v3 service):



Google Cloud Storage	OFF	Pricing
Google Cloud Storage JSON API	Request access...	Courtesy limit: 100,000 requests/day
Google Compute Engine	Request access...	
Google Maps API v2	OFF	Courtesy limit: 25,000 requests/day • Pricing
Google Maps API v3	ON	Courtesy limit: 25,000 requests/day • Pricing
Google Maps Coordinate API	OFF	Courtesy limit: 1,000 requests/day
Google Play Android Developer API	OFF	Courtesy limit: 15,000 requests/day
Google+ API	OFF	Courtesy limit: 10,000 requests/day

4. Again in the left-hand side menu select the **API Access** option. You will have to copy the **API key** and replace it in future Google API projects:


Gmail Calendar Drive Photos Sites Groups Search More ▾
007@gmail.com ▾ | Settings ▾ | Help | Sign out
Google apis
API Project ▾
Overview
Services
Team
API Access
Billing
Reports
Quotas
Create an OAuth 2.0 client ID...
Generate new key...
Edit allowed referers...
Delete key...
Create new Server key...
Create new Browser key...
Code Home - Privacy Policy

API Access

To prevent abuse, Google places limits on API requests. Using a valid OAuth token or API key allows you to exceed anonymous limits by connecting requests back to your project.

Authorized API Access

OAuth 2.0 allows users to share specific data with you (for example, contact lists) while keeping their usernames, passwords, and other information private. A single project may contain up to 7 client IDs. Learn more

Simple API Access

Use API keys to identify your project when you do not need to access user data. Learn more

Key for browser apps (with referers)

API key:	AIzaSyAywwIFJPo67Yd4vZgPz4EUSVul0BLHroE
Referers:	Any referer allowed
Activated on:	Jul 30, 2012 8:42 AM
Activated by:	007@gmail.com – you

This is the only time we will be talking about keys and permissions for working with the Google API platforms. Please validate that you have activated a key and also set the right libraries to be accessible to you.

How it works...

It's not hard to understand how this works. You just need to remember the steps as they will be the baseline of the future Google API interactions that we create.

As you must have noticed, there are many more API's in the Google library than we could even go through, but I do recommend that you scan through them and explore your options. In the next few recipes, we will be using the Google API to perform some mapping-related tasks.

Building a Google map

Data and geography have a very natural relation. Data has more meaning when it's on a map. Using live maps is a very good option as it would enable the users to interact with a UI that is integrated with your own data presentations within the geographic area. In this recipe, we will integrate our first, real live map.

Getting ready

To get through this recipe, you must have a Google ID. If you do not have one, you will need to create one. Beyond that you will need to activate the Google Maps API v3 service in the API Console. For more information on this, please review the recipe *Obtaining a Google API key* discussed earlier in this chapter.

Our goal will be to create a full-screen Google map that will be zoomed in and focused on France:

Map
Satellite
France
Nantes
Poitiers
Limoges
Clermont-Ferrand
Bordeaux
La Rochelle
Google
Map data ©2012 GeoBasis-DE/BKG (©2009), Google, Tele Atlas - Terms of Use

How to do it...

Let's list the steps to create this sample. To create this sample we will create two files—a `.html` file and a `.js` file:

1. Let's start with the HTML file. We'll create a basic HTML file baseline for our project:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Maps Hello world</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div id="jsmap"></div>
  </body>
</html>
```

2. We will add to the HTML viewport information. This is the direction for mobile devices on how to render the page (this step can be skipped if you don't care about accessing the maps in mobile devices):

   ```
   <head>
     <title>Google Maps Hello world</title>
     <meta charset="utf-8" />
     <meta name="viewport" content="initial-scale=1.0,
     user-scalable=no" />
   </head>
   ```

3. Add the style information into the header:

   ```
   <style>
     html { height: 100% }
     body { height: 100%; margin: 0; padding: 0 }
     #jsmap { height: 100%; width:100% }
   </style>
   ```

4. Load in Google Maps v3 API (replace the bolded text with your API key):

   ```
   <script src="http://maps.googleapis.com/maps/api/js?key=ADD_YOUR_
   KEY&sensor=true">
   ```

5. Add the script source for our `09.03.googleJSmaps.js` JavaScript file:

   ```
   <script src="./09.03.googleJSmaps.js"></script>
   ```

6. Add an `onload` trigger that will call the `init` function (this is to be created in the next step):

   ```
   <body onload="init();">
   ```

7. In the `09.03.googleJSmaps.js` JavaScript file, add the `init` function:

   ```
   function init() {
     var mapOptions = {
       center: new google.maps.LatLng(45.52, 0),
       zoom: 7,
       mapTypeId: google.maps.MapTypeId.ROADMAP
     };
     var map = new
     google.maps.Map(document.getElementById("jsmap"),
     mapOptions);
   }
   ```

8. Load the HTML file, and you should find a full-screen roadmap zoomed into France.

How it works...

The most important and first step is loading the `maps` API. For Google to honor your request, you must have a valid API key. As such, don't forget to replace the bolded text with your key:

```
<script src="http://maps.googleapis.com/maps/api/js?key=YOUR_
KEY&sensor=true">
```

Don't forget to use your own key. You might find yourself with a site that has a broken map. The `sensor` parameter in the URL is mandatory and must be set to `true` or `false`. If your map needs to know where the user location is, you must set it to `true`, and if not you may set it to `false`.

Another interesting thing to note in our application is that it's the first time we have used the viewport in our samples. As this topic is out of the scope of this book, I wanted to leave it. I know that many of you will end up using maps in mobile devices and will want to have the map default to a vertical/horizontal view. To learn more on how viewports work, check out the article available at: `https://developer.mozilla.org/en/Mobile/Viewport_meta_tag/`.

You must have noticed that we set many things in our CSS to be 100 percent, and as you probably guessed, it was for backward compatibility and validating that the map will fill the entire screen. If you just want to create a hard-set width/height, you may do so by replacing the CSS with the following code:

```
<style>
    #jsmap { height: 200px; width:300px; }
</style>
```

That covers the main things we need to do in the HTML file.

There's more...

We haven't yet covered the details of how the `init` function works. The basics of the `init` function are very simple. There are only two steps involved with creating a map. We need to know what `div` layer we want the map to be in and what options we want sent to our map:

```
var map = new google.maps.Map(div,options);
```

Contrary to the Google Visualization API that had three steps in the last recipe, we can see that the Google `maps` API has only one step and within it we send the two options directly to be rendered (there is no step between creating and rendering).

Let's take a deeper look into the options as they are what will change the majority of the visuals and functionality of the map.

Working with latitude and longitude

Latitude and longitude (**lat/long**) is a coordinate system that divides the Earth into a grid-like pattern, making it easy to locate points on the Earth. The lat represents the vertical space while the long represents the horizontal space. It's important to note that Google uses the World Geodetic System WGS84 standard. There are other standards out there, so if you don't use the same standard with your lat/long, you will find yourself at a different location from what you were originally looking for.

The easiest way to locate areas based on lat/long is through a helper tool on our map or through searching for the lat/long information of major cities.

`http://www.gorissen.info/Pierre/maps/googleMapLocation.php` will help you locate a dot by clicking on the Google map directly. Another option in this category is to turn on the labs feature in the main Google Map site (`http://maps.google.com/`). In the main Google Map site on the bottom-left side of the screen, you will find **Map Labs**. In there you will find a few lat/long helpers.

Or you can search the data per city by visiting `http://www.realestate3d.com/gps/latlong.htm`.

In our case when we are ready to make our pick, we will update the `options center` property to reflect where we want the map to be centered and adjust the zoom level until it feels right:

```
var mapOptions = {
    center: new google.maps.LatLng(45.52, 0),
    zoom: 7,
    mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

Map types

There are many map types, and you can even create your own custom ones, but for our needs we will focus on the basic ones that are used most often:

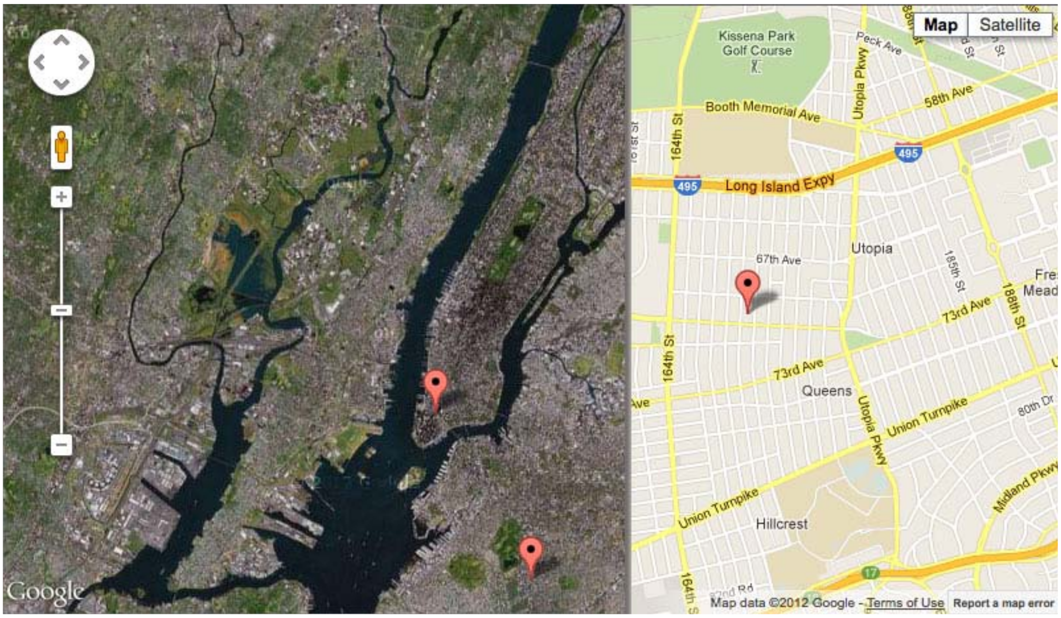
- `google.maps.MapTypeId.ROADMAP`: Displays the normal, default 2D tiles of Google maps
- `google.maps.MapTypeId.SATELLITE`: Displays the photographic tiles
- `google.maps.MapTypeId.HYBRID`: Displays a mix of photographic tiles and a tile layer for prominent features (roads, city names, and so on)
- `google.maps.MapTypeId.TERRAIN`: Displays physical relief tiles for displaying elevation and water features (mountains, rivers, and so on)

This covers the basics that you need to know to get rolling with integrating maps onto a site.

Adding markers and events

It's great that we have a map on our screen (assuming that you have followed the last recipe *Building a Google map*), but what about connecting data and integrating it into our map. I'm glad you asked about that, as this recipe will be our first step into adding data in the form of markers and events.

In this sample, our goal is to place four markers in New York City. When the markers are clicked, we will zoom into that area and switch the map view type.



Getting ready

At this stage, you should have created (at least once) a Google map by using the JS API; if you haven't, please revert back to the *Building a Google map* recipe.

How to do it...

We are not making any further changes in the HTML page created in the last recipe *Building a Google map*; as such we will focus our attention on the JavaScript file:

1. Create an `init` function:

```
function init(){
//all the rest of logic in here
}
```

2. Create map constants in the `base` state and then zoom in on the state:

```
function init() {
  var BASE_CENTER = new google.maps.LatLng(40.7142,-
  74.0064 );
  var BASE_ZOOM = 11;
  var BASE_MAP_TYPE = google.maps.MapTypeId.SATELLITE;
  var INNER_ZOOM = 14;
  var INNER_MAP_TYPE = google.maps.MapTypeId.ROADMAP;
```

3. Create the default map options:

```
//40.7142° N, -74.0064 E NYC
var mapOptions = {
  center: BASE_CENTER,
  zoom: BASE_ZOOM,
  mapTypeId: BASE_MAP_TYPE
};
var map = new google.maps.Map(document.getElementById("jsmap"),
mapOptions);
```

4. Create a data source for our points:

```
var aMarkers = [
  {label:'New York City',
  local: map.getCenter()},
  {label:'Brooklyn',
  local: new google.maps.LatLng(40.648, -73.957)},
  {label:'Queens',
  local: new google.maps.LatLng(40.732, -73.800)},
  {label:'Bronx',
  local: new google.maps.LatLng(40.851, -73.871)},

];
```

5. Loop through each array element and create a marker with an event that will zoom to the location, switch the view and pan to the correct location:

```
var marker;

for(var i=0; i<aMarkers.length; i++){
  marker = new google.maps.Marker({
    position: aMarkers[i].local,
    map: map,
    title: aMarkers[i].label
  });
  google.maps.event.addListener(marker, 'click',
  function(ev) {
    map.setZoom(INNER_ZOOM);
    map.panTo(ev.latLng);
    map.setMapTypeId(INNER_MAP_TYPE);
  });

}
```

6. Last but not the least, make the map clickable. So when the user clicks on the map, it should reset to its original state:

```
      google.maps.event.addListener(map, 'click', function() {
              map.setZoom(BASE_ZOOM);
          map.panTo(BASE_CENTER);
          map.setMapTypeId(BASE_MAP_TYPE);

      });
```

When you run the application, you will find four markers on the screen. When you click on them, you will jump into a deeper zoom view. When you click on an empty area, it will take you back to the original view.

How it works...

Working with events and Google Maps is very easy. The steps involved always start from calling the static method `google.maps.event.addListener`. This function takes in three parameters, namely the item to be listened to, the event type (as a string), and a function.

For example, in our `for` loop we create markers and then add events to them:

```
    google.maps.event.addListener(marker, 'click', function(ev) {
      map.setZoom(INNER_ZOOM);
      map.panTo(ev.latLng);
      map.setMapTypeId(INNER_MAP_TYPE);
     });
```

Instead we can create the event and then do not need to recreate a new anonymous function each time we loop through:

```
   for(var i=0; i<aMarkers.length; i++){
     marker = new google.maps.Marker({
       position: aMarkers[i].local,
       map: map,
       title: aMarkers[i].label
     });

     google.maps.event.addListener(marker, 'click', onMarkerClicked);
```

```
}

function onMarkerClicked(ev){
  map.setZoom(INNER_ZOOM);
  map.panTo(ev.latLng);
  map.setMapTypeId(INNER_MAP_TYPE);
}
```

The advantage is really big. Instead of creating a function for each loop, we are using the same function throughout (smarter and smaller memory footprint). In our code, we are not mentioning any hardcoded values. Instead we are using the event information to get the `latLng` property. We can re-use the same function without any issue. By the way, you might have noticed that this is the first time that we put a named function inside another named function (`init` function). This isn't a problem and it works exactly the same way as the variable scopes work. In other words, this function that we created will have visibility only within the `init` function scope.

The creation of a marker is very simple; all we need to do is create a new `google.maps.Marker` and assign a position and a map to it. All other options are optional. (For a full list, please review the Google API documentation available at `https://developers.google.com/maps/documentation/javascript/reference#MarkerOptions`.)

There's more...

You might have noticed that we use the method `map.panTo`, but no panning actually happens and everything snaps to place. If you run the map, you will discover that we don't actually see any panning; that is because we are switching the map type, zooming out, and panning at the same time. Only panning can actually animate without a few tricks and bypasses, but all these steps make our application a lot more complex and the actual control over animation is very limited. We will come up with a solution to that in the next recipe as we use two maps instead of one in *Customizing controls and overlapping maps*. If we wanted we could add in a delay and do each step separately and animate the pan, but if we want to create a smooth transition, I would think about the idea of having two separate maps, one on top of each other instead, and fading in and out the main world map.

Customizing controls and overlapping maps

The goal of this recipe is to practice working with Google Maps. We will integrate what we learned about working with Google Maps in this chapter and incorporate our control over the user behaviors, such as what controllers the user can use, into it. We will start digging into creating our own unsupported undocumented behaviors, such as locking the users' pan area.

Our main task in this recipe will be to take our work from the previous recipe, and instead of having the map zoom in and move around, create clean transitions between the zoomed in and zoomed out options; but as that isn't supported in a clean way through the interface, we will use external focuses. The idea is simple; we will stack up two maps on top of each other and fade in and out the top map, giving us total control over the fluidity of the transitions.

Getting ready

Even though we are starting from scratch, a lot of the work that we did in the last recipe is being re-used, so I strongly encourage you to go through the last recipe *Adding markers and events* before moving into this one.

In this recipe, we will be integrating jQuery into our work as well, to save us time on creating our own animator tool (or re-using the one that we created in the *Animating independent layers* recipe in *Chapter 6*, *Bringing Static Things to Life*), as it would take us away from our main topic.

How to do it...

In this recipe we will be creating two files. An HTML file and a JS file. Let's look into it, starting with the HTML file:

1. Create an HTML file and import the Google `maps` API and jQuery:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Google Maps Markers and Events</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="initial-scale=1.0,
    user-scalable=no" />
    <script src="http://ajax.googleapis.com/
    ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script
    src="http://maps.googleapis.com/maps/api/js?key=
    AIzaSyAywwIFJPo67Yd4vZgPz4EUSVu10BLHroE&sensor=true">
    </script>
    <script src="./09.05.controls.js"></script>
  </head>
  <body onload="init();">
    <div id="mapIn"></div>
  <div id="mapOut"></div>
  </body>
</html>
```

2. Use CSS to stack the map's layers on top of each other:

```
<style>
    html { height: 100% }
    body { height: 100%; margin: 0; padding: 0 }
    #mapIn, #mapOut { height: 100%; width:100%;
    position:absolute; top:0px; left:0px }
</style>
```

3. Create the `09.05.controls.js` JS file and create an `init` function in it (from this point onwards the rest of the code will be in the `init` function):

```
function init(){
  //rest of code in here
}
```

4. Create the two maps with their custom information:

```
var BASE_CENTER = new google.maps.LatLng(40.7142,-74.0064 );

//40.7142¬∞ N, -74.0064 E NYC
var mapOut = new google.maps.Map(document.
getElementById("mapOut"),{
  center: BASE_CENTER,
  zoom: 11,
  mapTypeId: google.maps.MapTypeId.SATELLITE,
  disableDefaultUI: true
});
var mapIn = new google.maps.Map(document.getElementById("mapIn"),{
  center: BASE_CENTER,
  zoom: 14,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  disableDefaultUI: true,
  panControl:true
});
```

5. Add the markers to the upper layer map:

```
var aMarkers = [
  {label:'New York City',
  local: mapOut.getCenter()},
  {label:'Brooklyn',
  local: new google.maps.LatLng(40.648, -73.957)},
  {label:'Queens',
  local: new google.maps.LatLng(40.732, -73.800)},
  {label:'Bronx',
  local: new google.maps.LatLng(40.851, -73.871)},

];
var marker;
```

```
    for(var i=0; i<aMarkers.length; i++){
      marker = new google.maps.Marker({
        position: aMarkers[i].local,
        map: mapOut,
        title: aMarkers[i].label
      });

    google.maps.event.addListener(marker, 'click',
    onMarkerClicked);

    }

    function onMarkerClicked(ev){
      mapIn.panTo(ev.latLng);
      $("#mapOut").fadeOut(1000);
    }
```

6. Add the `click` event to the internal map, and when you have clicked on it, you will be returned to the upper map:

   ```
   google.maps.event.addListener(mapIn, 'click', function() {
     mapIn.panTo(BASE_CENTER);
     $("#mapOut").fadeIn(1000);
     });
   ```

7. Force the user to disable `pan` in the upper map using the `center_changed` event:

   ```
   google.maps.event.addListener(mapOut, 'center_changed', function()
   {
           mapOut.panTo(BASE_CENTER);
   //always force users back to center point in external map
   });
   ```

When you load the HTML file, you will find a fullscreen map that cannot be dragged. When you click on a marker, it will fade into the selected area. You can now drag the cursor around the map. The next time you click in the internal map (regular click on any area), the map will fade back to the original upper layer.

How it works...

Our biggest step is the creation of two maps, one overlapping the other. We did that with some CSS magic by layering the elements and putting our top layer at the last position in the stack (we could probably use the z-index to validate it, but it worked so I didn't add that to the CSS). After that we created our two `div` layers and set their CSS code. In the JavaScript code, contrary to the way we did in the last recipe, we hardcoded the values that we wanted into both the maps.

In our options for both the maps, we set the default controllers not to take effect by setting the property `disableDefaultUI` to be `true`, while in `mapIn` we set `panControl` to be `true` to showcase that the map can be panned through:

```
var mapOut = new google.maps.Map(document.getElementById("mapOut"),{
  center: BASE_CENTER,
  zoom: 11,
  mapTypeId: google.maps.MapTypeId.SATELLITE,
  disableDefaultUI: true
});
var mapIn = new google.maps.Map(document.getElementById("mapIn"),{
  center: BASE_CENTER,
  zoom: 14,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  disableDefaultUI: true,
  panControl:true
});
```

We can manually set all the controllers by setting a Boolean value to any of the following options:

- `panControl`
- `zoomControl`
- `mapTypeControl`
- `streetViewControl`
- `overviewMapControl`

Our `event` logic works in the exact same way that it did in the last recipe. The only change is within the actual listeners where we switch between the maps using jQuery:

```
function onMarkerClicked(ev){
  mapIn.panTo(ev.latLng);
  $("#mapOut").fadeOut(1000);
}

google.maps.event.addListener(mapIn, 'click', function() {
  mapIn.panTo(BASE_CENTER);
  $("#mapOut").fadeIn(1000);
});
```

In both the event for the markers and the `click` event of the map, we are using the `fadeIn` and `fadeOut` methods of jQuery to animate our external maps visibility.

There's more...

When you try to drag around the higher-level map (the first visible map), you will notice that the map cannot move—it's not pannable. Google API v3 doesn't support the capability to disable the panning, but it does support the capability to get updated every time the map center point changes.

As such we listen in to the following change:

```
google.maps.event.addListener(mapOut, 'center_changed', function() {
        mapOut.panTo(BASE_CENTER);
});
```

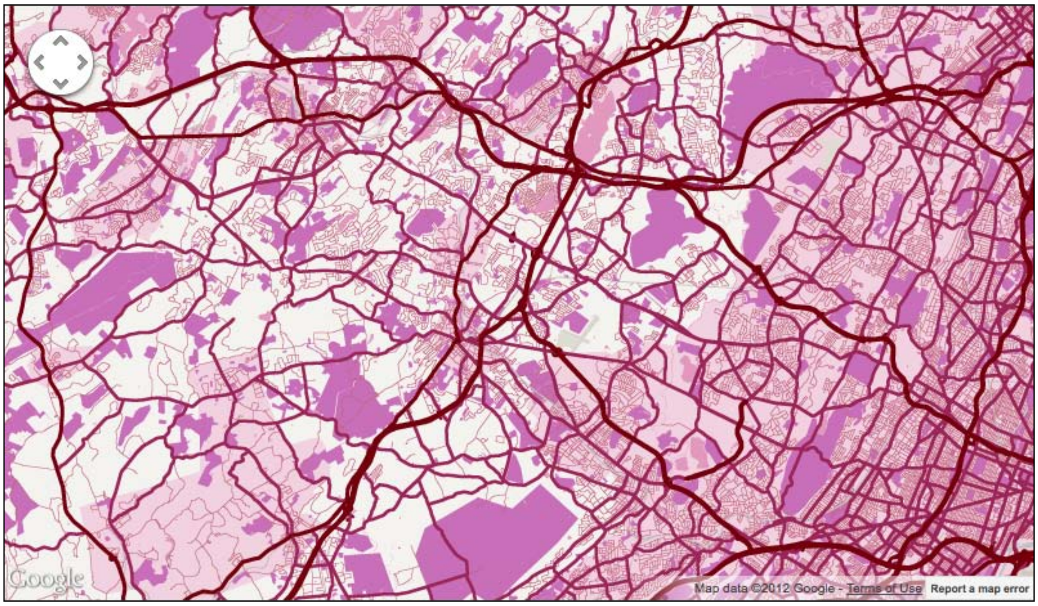
All we are doing is that each time the map position changes, we force it back to its original position, making it impossible to move our map around.

Redesigning maps using styles

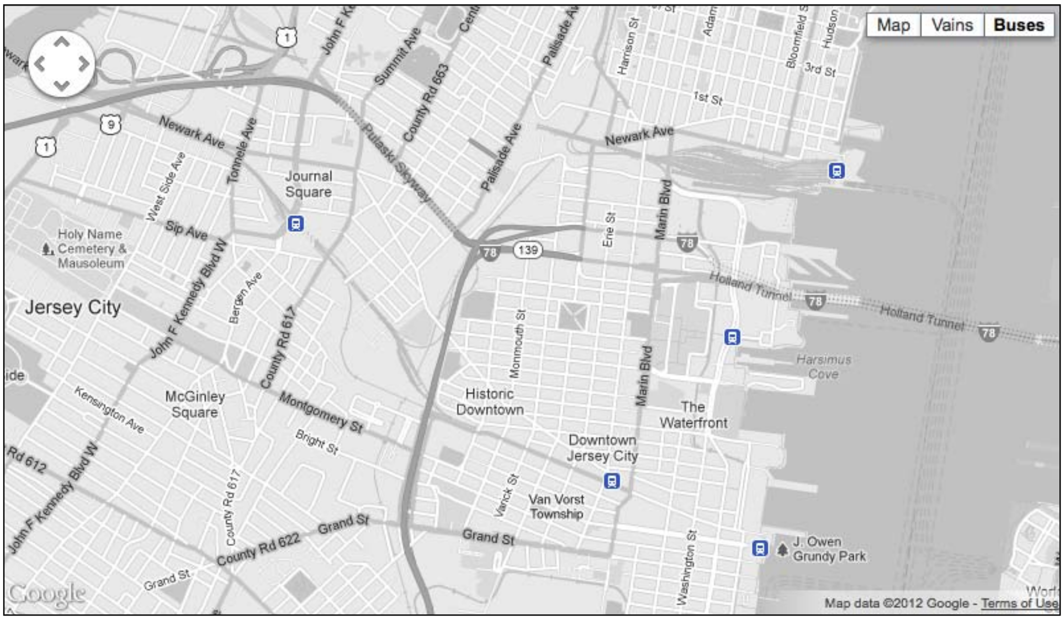
Many a time when creating more advanced applications by using Google Maps, you will want to create your own custom, skinned maps. This is really useful when you want to have a foreground content and don't want to have it compete with the background content.

In this recipe we will create a few styled maps. By the end of this recipe, you will know how to create global customization, individual styles, and last, but not least, add new map types.

Here is one style that we will create:



And here is a second style that we will create:



Getting ready

To complete this recipe, you will have to start from a copy of the previous recipe. We will only describe the new steps that differ from the last example in this recipe. To view and understand all the steps, please read the *Customizing controls and overlapping maps* recipe.

As such, we will skip the HTML code as it's exactly the same as in the previous recipe.

How to do it...

Open up your JavaScript file (`09.05.controls.js`) from the last recipe and follow these steps:

1. Within the `init` function create a `aVeinStyle` array. This array contains all the visual guides for skinning the map in the vein style:

```
var aVeinStyle =  [
  {
    featureType:'water',
    elementType: "geometry",
    stylers:[{color:'#E398BF'}]
  },
  {
    featureType:'road',
    elementType: "geometry",
    stylers:[{color:'#C26580'}]
  },
  {
    featureType:'road.arterial',
    elementType: "geometry",
    stylers:[{color:'#9B2559'}]
  },
  {
    featureType:'road.highway',
    elementType: "geometry",
    stylers:[{color:'#75000D'}]
  },
  {
    featureType:'landscape.man_made',
    elementType: "geometry",
    stylers:[{color:'#F2D2E0'}]
  },
  {
    featureType:'poi',
    elementType: "geometry",
    stylers:[{color:'#C96FB9'}]
  },
  {
    elementType: "labels",
    stylers:[{visibility:'off'}]
  }
];
```

2. Create a new `google.maps.StyledMapType` map with the name `Veins`:

```
var veinStyle = new google.maps.StyledMapType(aveinStyle,{name:
"Veins"});
```

3. Create a bus style:

```
var aBusStyle =  [
  {
    stylers: [{saturation: -100}]
  },
  {
    featureType:'transit.station.rail',
    stylers:[{ saturation:
    60},{hue:'#0044ff'},{visibility:'on'}]

  }
];

var busStyle = new google.maps.StyledMapType(aBusStyle,{name:
"Buses"});
```

4. For the internal map, make the map-type controller visible and include in it the IDs for our new map styles:

```
var mapIn = new google.maps.Map(document.getElementById("mapIn"),{
  center: BASE_CENTER,
  zoom: 14,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  disableDefaultUI: true,
  panControl:true,
  mapTypeControl:true,
  mapTypeControlOptions: {
    mapTypeIds: [google.maps.MapTypeId.ROADMAP,
    'veinStyle', 'busStyle']
  }

});
```

5. Add the map style information into the `mapIn` object:

```
mapIn.mapTypes.set('veinStyle', veinStyle);
mapIn.mapTypes.set('busStyle', busStyle);
```

6. Set a default map type:

```
mapIn.setMapTypeId('busStyle');
```

When you restart the HTML file in the internal map (after clicking on one of the markers), you will find a controller menu that enables you to switch between custom map types.

How it works...

Working with Google styles is fun, and they work in a way that is very similar to the way CSS works. The style we set has a few steps; the first step is to create the rules of the style, the next one is to define a Google-style object (`google.maps.StyledMapType`), and the last step is to define what map this style information is relevant to. Styles can only be applied to the maps of the `google.maps.MapTypeId.ROADMAP` type.

The first example is the creation of the bus style. The goal of this style is to make the map black and white and only highlight the public transportation stations:

```
var aBusStyle =  [
  {
    stylers: [{saturation: -100}]
  },
  {
    featureType:'transit.station.rail',
    stylers:[{ saturation:
    60},{hue:'#0044ff'},{visibility:'on'}]
```

```
      }
    ];

    var busStyle = new google.maps.StyledMapType(aBusStyle,{name:
    "Buses"});
```

The first variable is a regular array. We can add into it as many styles as we want; each time we want to define the rules (the search terms) that would apply before we actually list out the styles. Let's take a deeper look at one style rule:

```
{stylers: [{saturation: -100}]}
```

This example is the most basic. We have no rule, or in other words we want to apply this style to everything. As in this example we are setting the saturation to `-100`, we are making everything black and white (saturation values by default are `0` and can take values between `-100` and `100`).

The possible style properties are as follows:

- `visibility`: This is a string value (`no`, `off`, or `simplified`). This adds or removes elements from the map; for the most part, it would be used to remove text such as labels and details out of the elements depending on the information provided.
- `gamma`: This is a number value between `0.01` and `10` (`1.0` is the default). This option controls how much light is within the view. While lower values (lower than `1`) would sharpen the difference between lighter and darker colors, higher numbers (more than `1`) would have a more global effect, making everything glow more as the value goes up.
- `hue`: This is a hexadecimal color value wrapped into a string (such as `#222222`). For the best way to describe what the hue does, imagine putting on sunglasses that have tinted glass that matches the provided hexadecimal value. The way the tinted glass affects the colors around you and changes them is the same way that the hue colors of the map change.
- `lightness`: This is a value between `-100` and `100` (the default is `0`). This effect is really simple if you provide a value lower than `0`. It's the same effect as putting a black rectangle on top of your map and changing its opacity (that is, `-30` would match up to the opacity of 30 percent). You might have guessed the result of positive values—for positive values the idea is the same, but only with a white rectangle.
- `saturation`: This is a value between `-100` and `100` (the default is `0`). This effect focuses on a pixel-by-pixel value of `-100`. It would create grayscale image values that are nearer to `100`. It would remove all gray colors from the image, making everything more vivid.

This is all the style information that is available, and with it we can control every style element within the map. Each style property's information needs to be sent as a separate object within the `stylers` array; for example, if we wanted to add a `hue` to our snippet, it would look like this:

```
{stylers: [{saturation: -40},{hue:60}]}
```

Now that we know all the different ways in which we can change the visuals of the map, it's time to understand how we'll define what should be selected. In the last code snippet we've controlled the full map, but we can filter out what we want to control by adding filtering logic:

```
{elementType: "geometry",
  stylers:[{color:'#E398BF'}]
```

In this snippet, we are filtering out that we want to change the color of all the `geometry` elements, which means that whatever isn't a `geometry` element will not be affected. There are three types of element-type options:

- `all` (the default option)
- `geometry`
- `labels`

There is one more way to filter information, by using the `featureType` property. For example:

```
{
  featureType:'landscape.man_made',
  elementType: "geometry",
  stylers:[{color:'#F2D2E0'}]
}
```

In this case, we are listing out exactly what we want to focus on. We want to focus on both the type of feature and the element type. If we were to extract the `elementType` property, our color effect would affect both `geometry` and `labels`. Whereas if we were to extract `featureType`, it would affect all the `geometry` elements in the map.

For the full list of the `featureType` property options, please visit `http://goo.gl/H7HSO`.

There's more...

Now that we have under our belts how to create the styles we want to use, the next critical step is to actually connect our style to the map. The easiest way to do it (if we only have one style) is to connect it directly to the map:

```
inMap.setOptions({styles: styles});
```

This can be done by calling the `setOptions` function or by adding the `style` property when we create our map. Styles can only be added to road maps, and as such if you add this style to a map that isn't a road map, it would not be applied.

As we want to add more than one style option, we have to list out the map types. Before we can do that, we need to create a new map type object by using the following code:

```
var busStyle = new google.maps.StyledMapType(aBusStyle,{name:
"Buses"});
```

While creating the new map, we provided a name that will be used as our name in a controller—if we choose to create a controller (in our example we do). It's important to note that this name is not the ID of our element but just the label of the element, and we still need to create an ID for our element before sending it into the map. To do that we will first add the IDs into our controller and make our controller visible:

```
var mapIn = new google.maps.Map(document.getElementById("mapIn"),{
  center: BASE_CENTER,
  zoom: 14,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  disableDefaultUI: true,
  panControl:true,
  mapTypeControl:true,
  mapTypeControlOptions: {
    mapTypeIds: [google.maps.MapTypeId.ROADMAP, 'veinStyle',
    'busStyle']
  }

});
```

Following this, we will add the set instructions connecting our new map types to their style objects:

```
mapIn.mapTypes.set('veinStyle', veinStyle);
mapIn.mapTypes.set('busStyle', busStyle);
```

Last but not least, we can change our default map to be one of our styled maps:

```
mapIn.setMapTypeId('busStyle');
```

There you go. You now know everything that you need to know about how to work with styles in Google Maps.