Building spatial applications with Google Cloud SQL and Google Maps API

Last updated: 22 December 2014

Contents

Third-party products: This document describes how Google products work with third-party products and the configurations that Google recommends. Google does not provide technical support for configuring third-party products. GOOGLE ACCEPTS NO RESPONSIBILITY FOR THIRD-PARTY PRODUCTS. Please consult the product's Web site for the latest configuration and support information. You may also contact Google Partners for consulting services.

Introduction

This document shows you how to query and visualize geospatial data using Google Cloud SQL and the Google Maps version 3 JavaScript API. As of December 2014, Google Cloud SQL supports MySQL version 5.6, which supports a variety of geospatial searches to enable your web applications, such as:

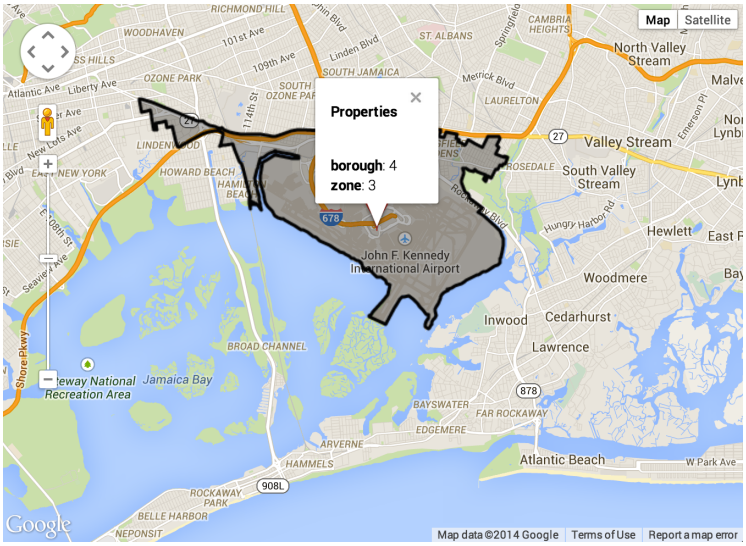
- **Distance:** Finds locations near a specific starting point
- **Contains:** Checks which polygon contains a specific location
- **Within:** Checks if a location is within a specific polygon

What this document covers

Using this document, you'll complete the following:

- Create a cloud project in the Google Developers Console.
- Create and configure a Cloud SQL instance.
- Create a Cloud SQL database table and import a geospatial data file.
- Develop a Google App Engine service to query Cloud SQL and return GeoJSON.
- Perform a geospatial query (such as point-in-polygon) with Cloud SQL.
- Create a Google Map web application to visualize the data.

After completing the steps in this document, you'll have a fully functional spatial application, hosted completely on Google Cloud Platform (GCP), that can perform an ST_Contains spatial operation to check if a point is within a polygon. You can view the sample application here: https://project-wander-1.appspot.com/.



Polygon displayed on a Google Map from the Cloud SQL application

Before you begin

Before following the steps in this document, set up the following:

- **Google Account:** Consider using a shared Google Account for your organization, rather than a personal Google Account.

- **IDE:** If you don't already have a development environment, consider installing an IDE such as Eclipse with the Google Plugin for Eclipse to help you deploy your application to App Engine.

- **Geospatial data management tool:** You make uploading data to Google Cloud SQL easier by using open-source tools such as GDAL or commercially available tools such as Safe Software FME. In this document, we'll use the GDAL toolset to upload vector data to Google Cloud SQL. You'll need a GDAL built with the MySQL driver. The Debian package `gdal-bin` comes with this driver, so an easy way to configure a machine for loading spatial data to Google Cloud SQL is to create a Debian Linux virtual machine with Google Compute Engine, and install the GDAL software package:

    1. Create a new Debian 7 virtual machine.
    2. Update the software: `sudo apt-get update`
    3. Install GDAL: `sudo apt-get install gdal-bin`

    Visit the Debian website for details about the `gdal-bin` package

- **Database management tool:** You'll need a tool such as MySQL Workbench to connect to your database hosted in Google Cloud SQL.

- **Dataset:** Download the New York City Hurricane Evacuation Zones vector dataset to follow along with the examples.

Part A: Storing geospatial data in Google Cloud SQL

We'll create an online database within Google Cloud SQL that stores the New York City Hurricane Evacuation data. The finished map will use these data to check which polygon a specified location is part of.

Create a Cloud Project in the Google Developers Console

The first step is to create a new project in the Cloud Developers Console. Specify a project name and project ID (the identifier for your App Engine instance, which can't be changed). For more information about Google App Engine, see the developer's documentation.

If you don't already have a development environment, consider installing an IDE such as Eclipse, with the Google Plugin for Eclipse to help you deploy your application to App Engine.

Create and configure a Google Cloud SQL instance

After you create a project, you need to create the Google Cloud SQL database and configure the database management tools:

1. Open the project you just created in the Google Cloud Console.

2. In the left pane, click **Storage**, and then click **Cloud SQL**.

3. Select **Create an instance**.

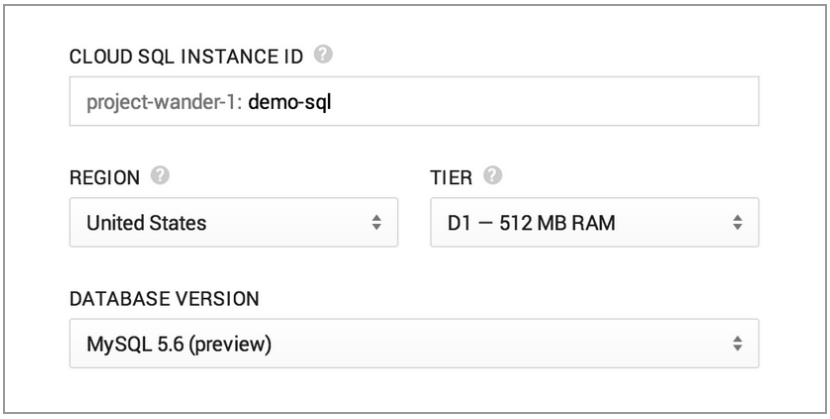
4. At the top, click **Show advanced options**.

5. Specify a name for the **Cloud SQL Instance ID** (example: `demo-sql`).

6. From the **Database Version** drop-down menu, select **MySQL 5.6 (preview)**.

7. For **Authorized Networks**, specify your own external IP (v4) address.

Note: You can also enter 0.0.0.0/0 to allow connections to the database instance from any IP address for demonstration purposes or to temporarily upload data. However, we strongly recommend that you restrict access when you're ready to use your Cloud SQL instance for production.



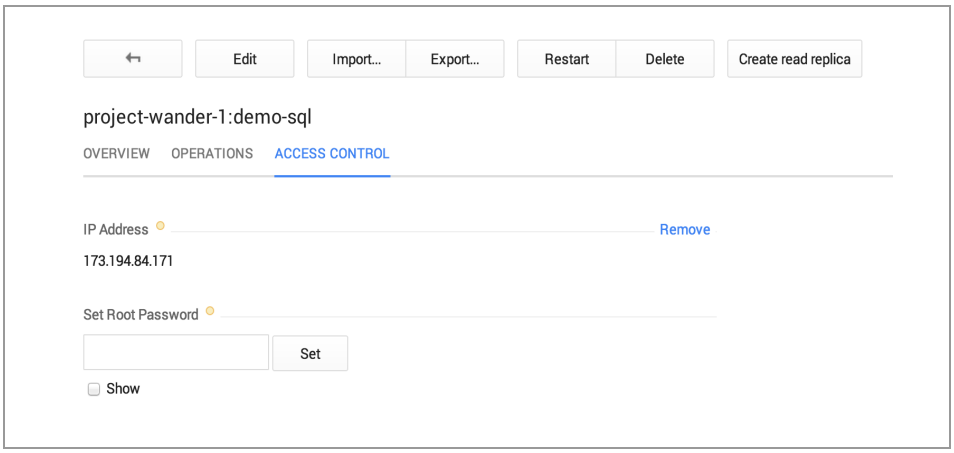
8. Click **Save** to create the Cloud SQL instance.

   Once you create your instance, you'll see it in the Developers Console:

9. Click the link for your instance under **Instance ID**. You'll now see a detailed console view of the Cloud SQL instance.

10. Click the **Access Control** tab.

11. Under **Set Root Password**, enter a password.



You can now connect to the instance using third-party tools, such as MySQL Workbench.

Note: In this example, the root user is used to connect to the Google Cloud SQL instance. Typically, you would use a non-root user with appropriate permissions to connect to the database instance. For information about creating MySQL users, see Adding Users in the MySQL documentation.

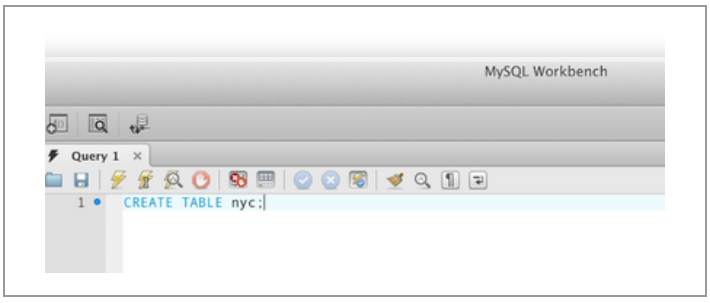
Connect to the instance and create a Google Cloud SQL database

You can use one of many open-source and licensed third-party tools to connect to your Google Cloud SQL instance, such as MySQL Workbench. You'll need one of these tools to complete the following steps.

1. Connect to your Cloud SQL instance.

2. Create a new database.

```
CREATE DATABASE nyc;
```

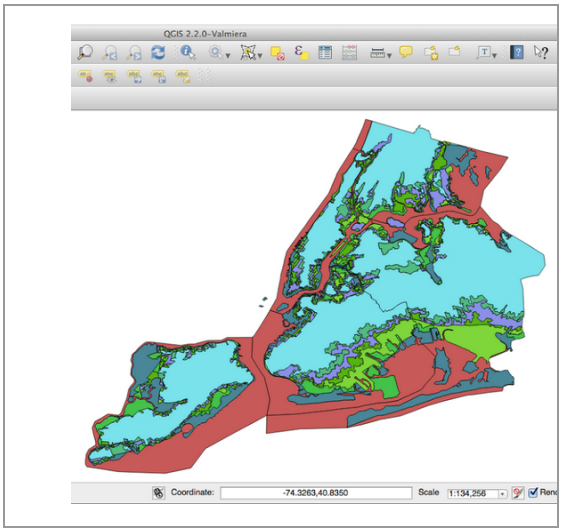


3. Click the lightning bolt button to execute the query.

Once your query runs, you'll have a database called `nyc`. You can can now close MySQL Workbench.

Load geospatial data into Google Cloud SQL

1. Now that you've created a database, upload the vector data of New York City Hurricane Evacuation Zones for your application.



2. Use the GDAL `ogr2ogr` command to load the Hurricane Evacuation Zones shapefile to the Cloud SQL instance database. Here's the example `ogr2ogr` command to load the data to your Cloud SQL instance:

```
ogr2ogr -F MySQL
MySQL:nyc,user=youruser,password=yourpassword,host=YOUR.IP
.ADD.RESS NYCZones.shp -nln nyczones -update -overwrite
-progress -lco GEOMETRY_NAME=geometry -lco engine=MYISAM
```

Important: You must include the '`-lco engine=MYISAM`' parameter with this command. This parameter is typically not included in other online examples, but it's required in this case.

Here's a description of the actions specified by to the `ogr2ogr` command:

1. Convert the `NYCZones.shp` shapefile to upload to our **`nyc`** Google Cloud SQL database.
2. Store the converted data in a new table named **`nyczones`**.
3. Create a geometry column named **`geometry`**.

The set of vector data you'll use with your application is now stored in Google Cloud SQL. You'll use these polygons later with your web page application, which returns the polygon in which the user clicks.

Part B: Performing spatial queries with Google Cloud SQL

MySQL version 5.6 supports spatial operations to interact with stored data. For you application, you'll use the ST_Contains() function to determine which polygon contains a specified point. This section also discusses some other frequently used spatial functions that you can use with your own applications.

Determining which area relates to a selected point

A typical spatial query is to determine if a given point is within a polygon, which is commonly called a "point-in-polygon" query in Geographic Information Systems (GIS) terms. The Cloud SQL ST_Contains() function handles this type of query.

For your example application, you want to determine which NYC Hurricane Evacuation Zone corresponds to the location that a user clicks on a map. Assume a user clicks the John F. Kennedy airport, which is located at 40.643363 degrees latitude, -73.782065 degrees longitude. Our corresponding query to determine the correct NYC Hurricane Evacuation Zone is:

```
SELECT zone, AsWKT(geometry) AS wktgeom FROM nyc.nyczones WHERE
ST_CONTAINS(geometry, GeomFromText('Point(-73.782065
40.643363)'));
```

This query's action is to return columns `zone` and `geometry` when one of the available geometries in the `nyc.nyczones` table contains the specified point location -73.782065 40.643363.

The result is the row within the `nyc.nyczones` table that contains the two requested columns, including the actual geometry shape of the corresponding Hurricane Evacuation Zone.

Additional spatial queries you can use

MySQL version 5.6 enables several types of spatial operations, in addition to the Contains() function, to support your applications. Here are two more commonly used spatial operations for mapping applications that can might enable your own applications:

ST_Distance()

> **Description:** Find all locations within a specified distance of a location.
> **Useful for:** Selecting all of your stores within 2 miles of an address, or all of your restaurants within 0.1 kilometers of a user's driving directions.
>
> See the “Optimizing Performance” section below for performance improvement considerations.

ST_Within()

> **Description:** Check if a specified location is within another geometry.
> **Useful for:** Checking if a vehicle has left a specified geofence.

For a full list of spatial functions supported by MySQL, see the MySQL documentation.

Using Google App Engine to bridge Cloud SQL and Maps API

Next, you need to create a connector to handle incoming location queries from our map application, as Google Cloud SQL does not have a native RESTful API. You’ll create the connector to Google App Engine to transact queries with your Google Cloud SQL instance from your Google Maps API application.

There are a number of methods for developing a Google App Engine application to connect to Google Cloud SQL. Many libraries, code samples, and documentation for a variety of languages, such as Java, Python, PHP and Go are available at the Google App Engine Developer Center.

This document shows you one approach to querying Google Cloud SQL from Google App Engine using Python.

To get started with Google App Engine, visit the Developer Center.

```
import geojson
from geomet import wkt

# Sends a query to our CloudSQL server
def QueryCloudSQL(query):
    socket_name = '/cloudsql/%s' % _CLOUD_SQL_INSTANCE
    db = MySQLdb.connect(unix_socket=socket_name,
                         db=_DATABASE, user='root')
    cursor = db.cursor()
    #Use the haversine formula in querying. See below.
    cursor.execute(query)
    cols = [i[0] for i in cursor.description]
    rows = cursor.fetchall()
    feature_id = 0
    for row in rows:
        wktgeom = row[-1]
        props = dict(zip(cols[:-1], row[:-1]))
        # wkt.loads returns a dict which corresponds to the geometry
        # We dump this as a string, and let geojson parse it
        geom = geojson.loads(json.dumps(wkt.loads(wktgeom)))
        # Turn the geojson geometry into a proper GeoJSON feature
        feature = geojson.Feature(geometry=geom, properties=props,
                                  id=feature_id)
        feature_id += 1
        # Add the feature to our list of features.
        features.append(feature)
    # Close the cursor, now that we are done with it.
    cursor.close()

# This function does a point in polygon query
def GetPolygonForPoint(lat, lng):
    query_parts = [
        'SELECT zone, AsWKT(geometry) AS wktgeom',
        'FROM nyc.nyczones',
        'WHERE ST_CONTAINS(geometry, GeomFromText(\'Point(%f %f)\'))' % (lng,
lat)]
    return QueryCloudSQL(' '.join(query_parts))
```

You now have a way to check your database and determine which polygon contains a user-specified location, using the MySQL ST_Contains() spatial function. You also have a connector that you can use with your Google Maps API web application to forward requests to the NYC Hurricane Evacuation Zone polygons stored in Google Cloud SQL.

In the next section, you'll bring these lookup pieces together with the map application for user interaction and displaying the results.

Part C: Visualizing data with Google Maps version 3 JavaScript API

In Parts A and B, you stored your vector table in Google Cloud SQL, created a query to check if a location is within a specific polygon, and a created a connector that can send user-specified locations to our Cloud SQL instance and return the GeoJSON result to display on your Google Maps API application. In this section, you'll build a web page that:

1. Loads the Google Maps version 3 JavaScript API and displays a map.
2. Sends user selected locations to your Google App Engine connector.
3. Draws the polygon returned from Google Cloud SQL onto the map.

Creating a Google Maps version 3 JavaScript API application

You'll need a web page to host your Google Maps version 3 JavaScript API application, along with a web service to host the web page.

Here's a copy of our example web application, below, which you can copy and paste onto your own web page. To use the example, add your own Google Maps API key. You can also create a new API key in the Google Cloud Console.

```
<!DOCTYPE html>
<html>
  <head>
    <style type="text/css">
      html, body, #map-canvas { height: 100%; margin: 0; padding: 0;}
    </style>
    <script type="text/javascript"
      src="https://maps.googleapis.com/maps/api/js?key=API_KEY">
    </script>
    <script type="text/javascript">
      function initialize() {
        var mapOptions = {
          center: { lat: 40.661407, lng: -74.036646},
          zoom: 8
        };
        var map = new
google.maps.Map(document.getElementById('map-canvas'),
            mapOptions);
      }
      google.maps.event.addDomListener(window, 'load', initialize);
    </script>
  </head>
  <body>
<div id="map-canvas"></div>
  </body>
```

```
</html>
```

Next, you'll create a JavaScript prototype function to pass the latitude/longitude values for where a user clicks on the map to the Google App Engine connector.

```
CloudSqlJsonApi.prototype.PiP = function(table, select, location,
callback) {
    var args = {
      lat: location.lat(),
      lng: location.lng(),
      select: select
    };
    var encoded_args = $.param(args);
    this.pipCallbacks.add(callback);
    var self = this;
    var url = "/pip/"+this.database+":"+table+"?"+encoded_args;
    $.getJSON(url, function(response){
        self.pipCallbacks.fire(response);
        self.pipCallbacks.remove(callback);
    })
  .fail(function(jqXHR, errorJson) {
    alert(errorJson.error);
  });
}
```

Now, you'll add your point-in-polygon function to your application, along with event listeners, to collect the latitude/longitude values for where users click on the map.

```
var pip = new CloudSqlJsonApi('nyc');
// Get the polygon that contains the clicked point.
pip.PiP("nyczones", "zone", latlng, function(geojson) {
  if(geojson.features.length == 0) {
    alert('Point is not in Polygon in CloudSQL Database');
    return;
  }
  map.data.forEach(function(feature) {
    map.data.remove(feature);
  });

  var features = map.data.addGeoJson(geojson);
  map.data.setStyle({
    clickable: true
  });
  map.data.addListener('click', HandlePolygonClick);

  activePolygons = [];
  for(i in features) {
```

```
    activePolygons.push(features[i].getId());
  }
  map.data.setStyle(function(feature) {
    var color = "black";
    var clickable = false;
    if($.inArray(feature.getId(), activePolygons) > -1) {
      color = "blue";
      clickable = true;
    }
    return {
      fillColor: color,
      strokeColor: color,
      fillOpacity: 0.2,
      clickable: clickable,
    }
  });
});
```

Your application will now send user-selected locations on the map to your Google App Engine connector, which then queries your Google Cloud SQL instance to find the correct polygon.

You can also use Google App Engine to host your completed web application. For instructions, see the Google App Engine documentation.

The complete application

You now have a complete example application that supports the Contains() spatial query, hosts vector data within Google Cloud SQL, and displays the result on Google Maps API.

You can see the completed example at https://project-wander-1.appspot.com/

Optimizing performance

Here are some tips to help you optimize performance with Google App Engine and Cloud SQL.

Enable spatial indexes on MySQL tables

You can improve query response performance by creating a spatial index on MySQL tables with geospatial datasets. Details about how the spatial index benefits search operations is available at the MySQL documentation library.

MySQL enables three methods to create a spatial index. For our example with the New York City Hurricane Evacuation Zones example, you can add a spatial index to table `nyc` with:

```
ALTER TABLE <table-name> ADD SPATIAL INDEX(<geometry-column-name>);
```

Connection pooling

If the time to create a new database connection is greater than checking and reusing existing connections, you might want to use connection pools. Connection pooling allows you to keep your database connections open for reuse. With App Engine, connection pooling is rarely necessary, but it might improve performance if your web application is hosted elsewhere. For more information about connection pooling, see the Google Cloud SQL FAQ.

Memcache

Leveraging Memcache to store frequently accessed queries can be a great way to improve application performance. Memcache allows you to store key/value pairs of data directly on an App Engine instance. If your application has a default map view or popular geographic searches (such as San Francisco, New York, or London), you may decide to store those values in Memcache to offset some of the load to your Cloud SQL instance. In our testing, we found that Memcache could return cached GeoJSON data over 10 times faster than requerying Cloud SQL for that same information. Items in Memcache are flushed periodically; therefore, check if the item exists in Memcache first and, if not, retrieve from the original data source (Cloud SQL), return, and store it in Memcache. You can also flush cached data based on an expiration time or by explicitly removing it via the Memcache API.

Cloud SQL database replication

Cloud SQL supports replication from a master Cloud SQL instance to multiple slave Cloud SQL instances. Replication provides for additional read capacity for applications dominated by reads. As of December 2014, Cloud SQL replication is in beta and only supports Cloud SQL 5.5. You can find more information in the Google Cloud SQL documentation.