Building a store locator application with the Google Maps API and Google Cloud SQL

Last updated: 22 December 2014

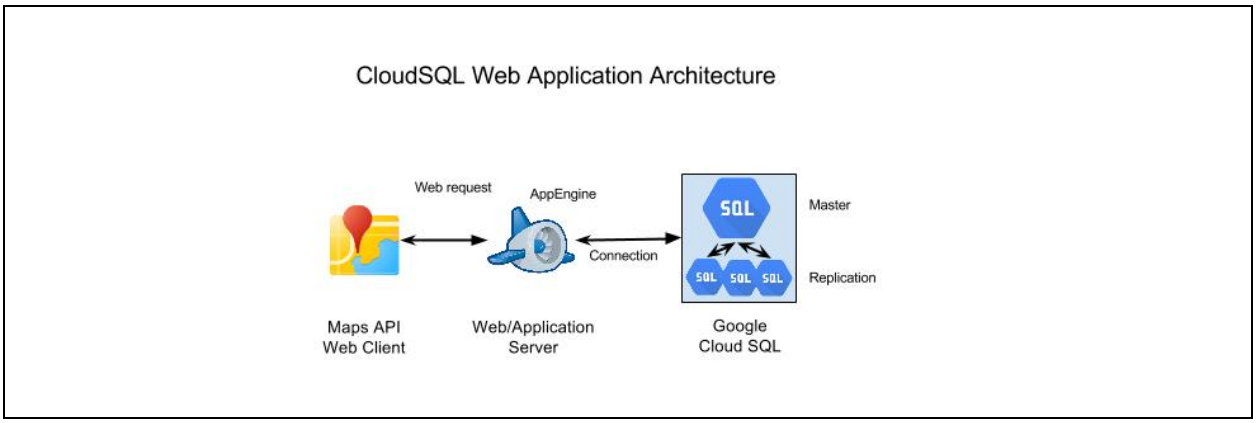
Contents

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 build a simple store locator using the Google Maps API, App Engine, and Google Cloud SQL. By following the steps in this document, you'll have a fully functional web application hosted completely on Google Cloud Platform (GCP).

The following diagram shows a basic Google Cloud SQL application architecture.



Because Cloud SQL does not have a RESTful endpoint that you can call client-side, you must create a server-side connection to Google Cloud SQL. One approach is to expose an endpoint on your application server (App Engine) to proxy these calls from your client.

Using this document, you'll complete the following steps:

1. Create a cloud project in the Google Developers Console.
2. Create and configure a Cloud SQL instance.
3. Create a Cloud SQL database table and populate with data.
4. Develop an App Engine service to query Cloud SQL and return GeoJSON.
5. Perform a geospatial query (distance search) with Cloud SQL.
6. Create a Google Map to visualize the data.

1. 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.

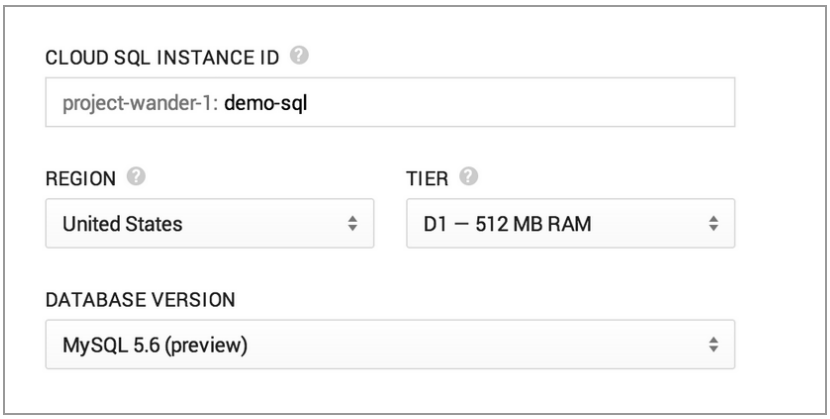
If you don't already have a development environment, consider installing an IDE such as Eclipse with the Google Plugin for Eclipse to help you deploy your application to App Engine.

2. Create and configure a 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.



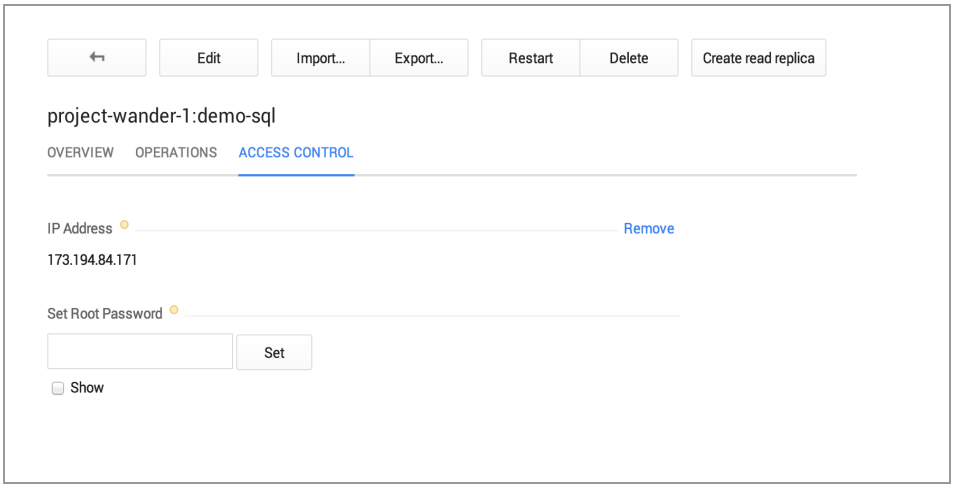
8. Click **Save** to create the Cloud SQL instance.

   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.

Other settings to consider:

- **Tier:** Select the tier appropriate for the level of usage you anticipate for your store locator.
- **Activation Policy:** To ensure the best performance, select **Always On** to prevent your instance from going to sleep. Note, however, that this option will result in higher uptime charges.
- **IPv4 address:** Assign an address for your instance, to allow third-party tools to access it.

- **Authorized Networks:** To connect to the database from your local machine, authorize the IP address of your machine.
- **Access Control:** Set a **root password** for your instance.

For more detailed information on setting up a Cloud SQL instance, see the developer's documentation, or watch this YouTube tutorial video.

3. Create a Cloud SQL database table and populate with data

The next step is to create a database table and load it with data. There are multiple approaches for loading data into Cloud SQL, which you'll find in the developer's documentation.

Using third-party tools

You can use third-party tools to make it easier to load data into Cloud SQL. Please remember, though, that Google does not endorse or support third-party tools.

Safe Software for synchronizing data

Safe Software provides the FME suite of tools, which can synchronize data between a number of different sources and destinations. As of December 2014, the Windows and Linux versions of FME support both GME and Cloud SQL and can be used to transfer data between the two environments. For more information about FME, see the Safe Software site. Also see our step-by-step guide on using FME to translate data from GME to Cloud SQL.

Tools for importing data from a CSV file

You can use a third-party tool, such as the MySQL command-line tool, MySQL Workbench, or ogr2ogr to make it easier to load data into Cloud SQL. For more details about this approach, see our step-by-step guide on building spatial applications with Google Cloud SQL and Google Maps API.

Connect to your database using your database administration tool. First create a new database, then create a table with fields for your stores, such as `id`, `name`, `address`, `lat`, and `lng`.

Here is a sample SQL statement to create this table:

```
CREATE  TABLE `store_locator`.`stores` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `name` VARCHAR( 60 ) NOT NULL ,
  `address` VARCHAR( 80 ) NOT NULL ,
  `lat` FLOAT( 10, 6 ) NOT NULL ,
  `lng` FLOAT( 10, 6 ) NOT NULL
) ENGINE = MYISAM ;
```

Alternatively, in MySQL 5.6, you can use the geometry type for for lat/lng data (rather than `float`) to store the position of this marker. This will enable the use of spatial indices that can improve performance. Creating a table with the geometry type looks like this:

```
CREATE  TABLE `store_locator`.`stores` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `name` VARCHAR( 60 ) NOT NULL ,
  `address` VARCHAR( 80 ) NOT NULL ,
   geometry GEOMETRY NOT NULL, SPATIAL INDEX(geometry)
) ENGINE = MYISAM ;
```

It's now time to import your list of stores. You can do this using a tool such as MySQL Workbench or the INSERT INTO command. To import data using MySQL Workbench, open your table using **Edit Table Data**. Then select the option to **Import Records from an External File**, and select a CSV file with your stores information.

To test data import, you can use this sample data from a Google Maps API tutorial.

```
Frankie Johnnie & Luigo Too,"939 W El Camino Real, Mountain View, CA",37.386339,-122.085823
Amici's East Coast Pizzeria,"790 Castro St, Mountain View, CA",37.38714,-122.083235
Kapp's Pizza Bar & Grill,"191 Castro St, Mountain View, CA",37.393885,-122.078916
Round Table Pizza: Mountain View,"570 N Shoreline Blvd, Mountain View,
CA",37.402653,-122.079354
Tony & Alba's Pizza & Pasta,"619 Escuela Ave, Mountain View, CA",37.394011,-122.095528
Oregano's Wood-Fired Pizza,"4546 El Camino Real, Los Altos, CA",37.401724,-122.114646
```

If you're using the geometry type for your lat/lng data, the `geometry` field should contain WKT— for example, "`POINT(lng lat)`". Format the row entry as follows:

```
Frankie Johnnie & Luigo Too,"939 W El Camino Real, Mountain View, CA",
"POINT(-122.085823 37.386339)"
```

4. Develop an App Engine service to query Cloud SQL and return GeoJSON

There are a number of ways for your App Engine application to connect to Google Cloud SQL. Libraries, code samples, and documentation for a variety of languages, such as Java, Python, PHP and Go are available on the developer's site.

Here's a sample servlet that demonstrates how to query Cloud SQL from App Engine using Python:

```
import geojson
from geomet import wkt

socket_name = '/cloudsql/%s' % _CLOUD_SQL_INSTANCE
db = MySQLdb.connect(unix_socket=socket_name,
                     db=_DATABASE, user='root')
cursor = db.cursor()
cursor.execute(query) #Use the haversine formula in querying. See below.
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()
```

The example above connects to the Google Cloud SQL instance as the root user, but you can connect to the instance as a specific database user with the following parameters:

```
db = MySQLdb.connect(unix_socket='/cloudsql/' + _CLOUD_SQL_INSTANCE, db=_DATABASE,
user='user', passwd='password')
```

For information about creating MySQL users, see Adding Users in the MySQL documentation.

You can expose the App Engine service to your web application in the form of a RESTful web service so it can be called from JavaScript or a mobile device. Additionally, you can use Google Cloud Endpoints to expose the function as an API call for your clients.

We recommend returning GeoJSON for easy integration into the Google Maps API using the DataLayer class. DataLayer provides an easy way to render styled geospatial data (points, polygons, polylines, etc.) on your Google Maps API implementation with minimal coding.

5. Perform a geospatial query (Distance Search) with Cloud SQL

To find locations in your stores table that are within a certain radius distance of a given latitude/longitude, you can use either of the following:

- A SELECT statement based on the haversine formula (recommended)
- The ST_DISTANCE function in Cloud SQL

Using the haversine formula

The haversine formula is used generally for computing great-circle distances between two pairs of coordinates on a sphere. An in-depth mathematical explanation is given by Wikipedia, and a good discussion of the formula as it relates to programming is on Movable Type's site.

The following SQL statement will find the closest 50 locations to the (36, -78) coordinate. It calculates the distance based on the latitude/longitude of that row and the target latitude/longitude, and then asks for only rows where the distance value is less than 25, orders the whole query by distance, and limits it to 20 results. To search by kilometers instead of miles, replace 3959 with 6371.

```
SELECT *, ( 3959 * acos( cos( radians(36) ) * cos( radians( lat ) ) * cos(
radians( lon ) - radians(-78) ) + sin( radians(36) ) * sin( radians( lat ))))
AS distance
FROM store_locator.store
ORDER BY distance
LIMIT 50;
```

You can include the haversine formula as a stored procedure or function in Cloud SQL to reduce query size and improve performance.

You can also improve performance by using a spatial index and a bounding box (with st_contains) to restrict the search space to a smaller area. For example:

```
SELECT *, astext(geometry), ( 3959 * acos( cos( radians(40.741) ) * cos(
radians( Y(geometry) ) ) * cos( radians( X(geometry) ) - radians(-74.0001) ) +
sin( radians(40.741) ) * sin( radians( Y(geometry) ) ) ) )
AS distance
FROM store_locator.store
where st_contains(GeomFromText('POLYGON((-74.09363384329599
```

```
36.205953339157645,-74.09363384329599 45.276046660842354,-73.90656615670402
45.276046660842354,-73.90656615670402 36.205953339157645,-74.09363384329599
36.205953339157645))'),geometry)
ORDER BY distance
LIMIT 50;
```

Using ST_DISTANCE

The following is an alternative approach using the ST_DISTANCE function in Cloud SQL. Note that this technique may be inaccurate over larger distances, as it's using a planar coordinate system, as opposed to spherical. Also note that ST_DISTANCE does not use spatial indexing, so it might be slower than haversine.

```
SELECT *, st_distance(POINT(lon,lat),POINT(-78, 36)) as distance FROM
store_locator.store ORDER BY distance LIMIT 50;
```

6. Create a Google map to visualize the data

The final step is to visualize the results using the Google Maps API. Take a look at the developer documentation on how to get started and putting markers on a map.

See the entire demo in action, and download both the sample connector source code and the store locator UI source code.

Optimizing performance

Here are some tips to help optimize the performance of your store locator.

Connection pooling

In some instances, if the time to create a new database connection is greater than checking and reusing existing connections, it might make sense to use connection pools. Connection pooling allows you to keep open your database connections for reuse. With App Engine, this is rarely necessary, but it may improve performance if your web application is hosted elsewhere. For more information about connection pooling, see the 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 supports only Cloud SQL 5.5. For more information, see the article on configuring replication in the Cloud SQL documentation.

Alternative solutions

While this document describes one solution for developing a store locator using Cloud SQL, there are other approaches you can take. The right technical solution is driven by considerations such as data size (number of locations) and performance requirements. Other solutions may include:

- Hosting a JSON file on your server and searching client-side, as shown in this example
- Using Google App Engine Search API
- Using Google Cloud Datastore
- Standing up your own database cluster on Google Compute Engine
- Using a locally-hosted database on your web server