AgriAnalysis Overview and Documentation

Produced by Jherez Taylor

Products and Libraries Used

Google App Engine - https://developers.google.com/appengine/

Google App for Eclipse - https://developers.google.com/appengine/docs/java/tools/eclipse

jQuery - http://jquery.com/

Bootstrap - http://getbootstrap.com/

Moment.JS - http://momentjs.com/

Lo-Dash - http://lodash.com/

AdminLTE web template - https://github.com/almasaeed2010/AdminLTE/

HighCharts - http://www.highcharts.com/

Source code for AgriAnalysis can be found at:
https://github.com/steffanboodhoo/agriExpenseTT/tree/AppEng

Copyright Information

Project Overview

AgriAnalysis is a companion app for the AgriExpense Android App. Data collected form the Android App is sent to a database provided by Google, known hereafter as the App Engine. The purpose of AgriAnalysis is to provide non identifying visualizations of the data submitted by the farmers. As it relates to usable categories of information, there exists:

- Crop Cycle – as it says on the tin, this is the category that stores information about the crop cycle that the farmers generate. It stores the account name, the cost per unit of produce, a unique identifier for the crop, the crop name, the amount harvested, the unit of harvest (Lbs, Kg), land quantity, the unit of land (Acre, Hectre), the start date, and the total amount spent on that crop cycle.

- Purchases – this stores the purchases made by each farmer. It stores the account name, the cost, the name of the item purchased, a unique identifier for the item, the

amount purchased, the amount remaining, the unit of measurement (L, Kg, g, ml, oz, Lb, seed, seedling) and the type of the item (Chemical, Fertilizer, Soil Amendment, Planting Material).

This data is parsed and used to provide a macro overview of what is occurring in the local agriculture industry. It aims to visualize the production breakdown by cost and volume and to show the costs of production either nationally or by the selected county. A live version can be found at http://1-dot-thematic-ruler-633.appspot.com/ but it is important to note that the address is dependent on the account of the user that set it up. This will be explained in the App Engine overview.

App Engine Overview

"Google App Engine is a Platform as a Service (PaaS) offering that lets you build and run applications on Google's infrastructure. App Engine applications are easy to build, easy to maintain, and easy to scale as your traffic and data storage needs change. With App Engine, there are no servers for you to maintain. You simply upload your application and it's ready to go." https://developers.google.com/appengine/docs/whatisgoogleappengine

Google App Engine can be developed in either Java, Python, PHP and Go. AgriAnalysis and AgriExpense both have a Java backend. It is available as a plugin for the Eclipse IDE. At this point it is important to note that the source code for AgriAnalysis is and must be maintained within the AgriExpense Android App. Here's how it works, when creating an Android App, the App Engine section is created and tied to it. Conceptually, the App Engine is exactly that, an engine that powers the application, analogous to an automobile engine. It is advised that you peruse the App Engine documentation that was linked in order to get an in depth understanding of the product, but for the purposes of this documentation we will focus on the Cloud Endpoints.

In order to conduct further maintenance on the application, it is necessary to register for a Google Developer account, the instructions will be on the page linked at the beginning of this section, under Getting Started. Once complete, we need to change the application ID in Eclipse, this is done by clicking on the App Engine project and selecting Google->App Engine Settings. The next step would be to expand the folder for the Android App, AgriExpenseTT->src->helper->CloudEndpointUtils.java. In this file we must change:

protected static final String LOCAL_APP_ENGINE_SERVER_URL = "https://your-project-id-here.appspot.com/";

and

protected static final String LOCAL_APP_ENGINE_SERVER_URL_FOR_ANDROID = "https://your-project-id-here.appspot.com";

NB. The project ID must be the same one that is set in the version of the Android App that has been deployed to the users.

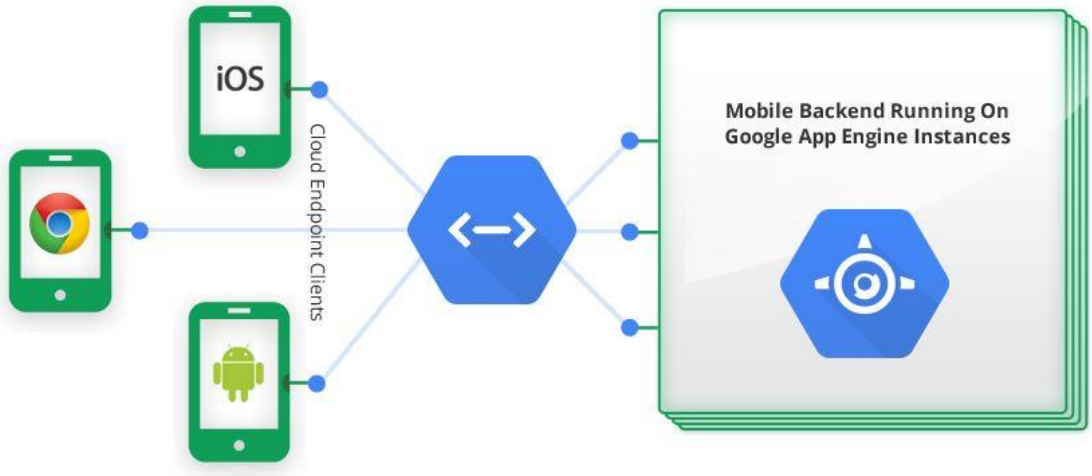
The following links will prove useful for a more thorough explanation

https://developers.google.com/appengine/docs/java/endpoints/getstarted/backend/code_walkthrough

https://developers.google.com/appengine/docs/java/endpoints/getstarted/backend/code_walkthrough

“Google Cloud Endpoints consists of tools, libraries and capabilities that allow you to generate APIs and client libraries from an App Engine application, referred to as an API (Application Program Interface) backend, to simplify client access to data from other applications. Endpoints makes it easier to create a web backend for web clients and mobile clients such as Android or Apple's iOS.”

What this means is that we are able to write Java (or whatever language you choose to base your App Engine on) code once and then generate APIs and code in order to access and manipulate the same app engine from other platforms. The visual below should provide a better understanding.



In our case, Java methods are written for the Android App (AgriExpense) and an API can be automatically generated by the Google App Engine Plugin to allow those methods to be accessed via the web through HTML and JavaScript. This is the primary reason why the project was based around the Google App Engine, because of the ability to allow you to write once and reuse across other platforms. Development goes as follows: First we write our code for the Android App, then we write the backend code for it under the App Engine folder. Once classes are written we then right click on a class and hit “Generate Cloud Endpoint class”. This generates a class file that allows us to write code that will used by both the Android Application (AgriExpense) and the web component (AgriAnalysis). Once we are finished writing our backend code, we right click on the project and click “Generate Cloud Endpoint Client Library”. This is where the explanation of the cloud endpoint reveals itself. Doing this generates a web API that allows us to call the methods in our backend code from a web page through HTML and JavaScript.

When building out future pages, it is necessary to include loadAPI.js each page along with the following script:

```
<!-- Load APP Engine -->
   <script src="https://apis.google.com/js/client.js?onload=loadGapi">
      {
         "client": {},
         "googleapis.config": {
            root: API_URL
         }
      }
   </script>
```

Methods are accessed by first loading it in javaScript

```
// This method loads the Endpoint libraries
function loadGapi() {
   'use strict';
   gapi.client.load('cycleendpoint', 'v1', function () {
      console.log("Cycle API Loaded");
   });
   gapi.client.load('rpurchaseendpoint', 'v1', function () {
      console.log("RPurchase API Loaded");
   });
}
```

They can then be accessed like

```
function fetchProductionData() {
    'use strict';
    var cropValue = $("#cropSelection").val(),
        selectedArea = $("#areaSelection").val(),
        queryData = gapi.client.cycleendpoint.getMatchingCycles({
            "cropName": cropValue,
            "selectedArea": selectedArea,
            "start_date": dateFilter_start.toString(),
            "end_date": dateFilter_end.toString()
        });
    queryData.execute(function (resp) {
        generateProductionCharts(resp);
    });
}
```

The response is a JSON object that we then parse and perform visualisation operations with.

Datastore Overview - https://developers.google.com/appengine/docs/java/datastore/

In the linked resources, pay attention to the sections on Indexes, Entities and Properties and Datastore queries.The datastore is where the farmer records are stored within the app engine. It is comparable to database but there are a few key differences. The first thing to keep in mind is that the datastore is not a traditional Relational Database Management System. Records are stored by what is referred to as a Kind, each entity is of a particular kind, which is used to categorize the entity for the purposes of queries. For instance, a Cycle, as outlined previously is a Kind. When performing queries on the datastore they must use a defined index. “An index is defined on a list of properties of a given entity kind, with a corresponding order (ascending or descending) for each property. For use with ancestor queries, the index may also optionally include an entity's ancestors”

The indexes are entered manually and can be found in AgriAnalysisTT->WEB-INF->datastore-indexes.xml. Let’s say we want to select all cycles that match the parameters passed in,

county, crop name, land quantity and the start date. The resulting index that we would need to define would be as follows:

```
<datastore-index kind="Cycle" ancestor="false">
    <property name="cropName" direction="asc" />
    <property name="county" direction="asc" />
    <property name="startDate" direction="asc" />
  </datastore-index>
```

And the query would look like this:

```
mgr = getEntityManager();
query = mgr.createQuery("SELECT FROM Cycle AS Cycle WHERE cropName = :p1 AND county = :p2 AND startDate >= :p3 AND startDate <= :p4 ORDER BY startDate ASC");
query.setParameter("p1", cropName.toUpperCase());
query.setParameter("p2", selectedArea.toUpperCase());
query.setParameter("p3", Long.parseLong(start_date));
query.setParameter("p4", Long.parseLong(end_date));
```

Keep in mind that queries ca only be performed on the defined indexes.

Another important concept is that of namespaces. This represented a significant pain point during the development of the prototype. Due to the data model of the Android App, each users data is stored in its own namespace. A namespace is generated by using the Google account that is tied to the device that the user is logged into. So let's say a user's Google account is JohnDoe@gmail.com. The generated namespace would be _johndoe and his data would be stored only within that namespace.

The challenge came about due to how the datastore operates. As mentioned, queries are carried out on indexes, but since each users data is stored within a namespace then we need to fetch the list of namespaces and iterate through that list by setting the namespace for that query, performing that query and adding it to a list that we return at the end.

Observe:

```
@SuppressWarnings({"unchecked", "unused"})
  @ApiMethod(name="getMatchingCycles")
  public CollectionResponse <Cycle> getMatchingCyles(
    @Named("cropName")String cropName,
      @Named("start")Double start,
      @Named("end") Double end,
      @Nullable @Named("cursor") String cursorString,
      @Nullable @Named("limit") Integer limit) {

    //  NamespaceManager.set("_spydakat");

      EntityManager mgr = null;
      Cursor cursor = null;
      List<Cycle> execute = null;

      /*For namespace list fetching */
      DatastoreService ds = DatastoreServiceFactory.getDatastoreService();
    com.google.appengine.api.datastore.Query q = new
com.google.appengine.api.datastore.Query(Entities.NAMESPACE_METADATA_KIND);

    List<String> results = new ArrayList<String>();
    for(Entity e : ds.prepare(q).asIterable()){
       if (e.getKey().getId() != 0) {
            System.out.println("<default>");
          } else {
           // System.out.println(e.getKey().getName());
            results.add(Entities.getNamespaceFromNamespaceKey(e.getKey()));
          }
    }

      // Set each namespace then return all results under that given
namespace
    for(Iterator<String> i = results.iterator(); i.hasNext(); ) {
        NamespaceManager.set(i.next());

        try{
          mgr = getEntityManager();
          Query query = mgr.createQuery("select from Cycle as Cycle where
cropName=:x and landQty>=:y and landQty<=:z");
          query.setParameter("x",cropName.toUpperCase());
          query.setParameter("y",start);
          query.setParameter("z",end);

          if (cursorString != null && cursorString != "") {
            cursor = Cursor.fromWebSafeString(cursorString);
            query.setHint(JPACursorHelper.CURSOR_HINT, cursor);
          }

          if (limit != null) {
            query.setFirstResult(0);
              query.setMaxResults(limit);
          }

          execute = (List<Cycle>) query.getResultList();
            cursor = JPACursorHelper.getCursor(execute);

            if (cursor != null) cursorString = cursor.toWebSafeString();
            // Tight loop for fetching all entities from datastore and
accomodate
```

```
            // for lazy fetch.
            for (Cycle obj : execute);
            } finally {
              mgr.close();
            }
        }

        return CollectionResponse.<Cycle>builder()
          .setItems(execute)
          .setNextPageToken(cursorString)
          .build();
    }
```

For this method, the first order of business is to cycle through the datastore and get a list of all the namespaces. Next we iterate through that list of namespaces and set each one, perform the query and add the result to a list of cycles. It is returned at the end. The take away here is that for any method that needs to query across the entire datastore we must first fetch the list of namespaces and do as explained.

Future Development

The web application currently provides analysis on the cost of production and the harvest amounts. What has not been completed is visualisations into how the items purchased relate the crop cycles. It may not be feasible to modify the data model, but the issues encountered with this section is that the county is not stored on the purchase entity. Refer to the top of this document to refresh yourself on what exactly is stored with each purchase entity. The plan that was devised was to first fetch a list of the accounts and the associated counties. This would be done by simply returning all cycles results and parsing it through JavaScript. What we would need to filter out is the unique account and the county in order to obtain a paired list of account and county. Once this is done, we can simply run a query for the purchases that match the parameters set and then use the previous list to associate an account with a given county.

Visualizations can be prepared to show where a chart is most frequently grown and which materials are used in which counties. Other items can be patterned from http://faostat.fao.org/

Final Meeting Notes

In presenting the prediction model for AgriViz, one of the companion application under AgriNeTT, it came out that while NamDevCo wanted to come up with ways to analyse the data that they currently have, estimated volume and price, it became clear that this data was not enough to generate price predictions. What was suggested was that the AgriExpense app, which is tracking production, can be used as a starting point to collect more detailed data which can lead to better price predictions. Another problem was that the data NamDevCo currently has is not standardized in some cases, in terms of unit measurements. The take away from this is that the separate projects can feed into each

other. AgriExpense can allow for more detailed data collection and to standardize unit measurements and crop names. AgriAnalysis can then be used to provide a top down view of what is occurring in the market place and AgriViz, the market watcher can then use the data generated by AgriExpense to generate more accurate price predictions models.