Google Ads API Web App Demo Guide

Authors: @amberxie, @kjchan, @samcarlos
Last Updated: 2020-08-07

This article assumes that you are familiar with the Google Ads API Beta and are looking to integrate API functionality with a web application.

Overview

This demo guide will provide more information about the design and code structure of the Google Ads API Web App. If you are familiar with full stack web development, we recommend you take a look at the Authentication and Resources sections in order to branch off and develop your own application.

Frameworks

The front end is built through React.js, with components sourced from the Material-UI library. The backend is built on Java8, with dependencies managed by Maven. The project is hosted on Google App Engine, and uses Google Cloud Datastore for storage.

Development & Deployment

Front end development:

- Within the `frontend` directory, run `npm run local`. The react application will run on a localhost in your browser, and saving code changes will automatically rerender the application. Running `npm install` first may be necessary to load new modules.

Back end development:

- Within the `server` directory, run `mvn package appengine:run`. The front end will be served as static files within the `webapp` directory.
- To run the back end with updated front end files, run `npm run build` within the `frontend` directory, and copy over the `build` folder into the `server/src/main` directory, making sure to rename the folder to `webapp`.

Deployment:

1. `pom.xml`: handles the deployment of the maven project to Google App Engine.

a. Dependencies: java servlets, ads api, gson, appengine, etc.
b. Project-id: find the projectId of the project you will deploy on Google Cloud Platform.

```
<configuration>
  <deploy.projectId><!-- TODO: set project ID. --></deploy.projectId>
  <deploy.version>1</deploy.version>
</configuration>
```

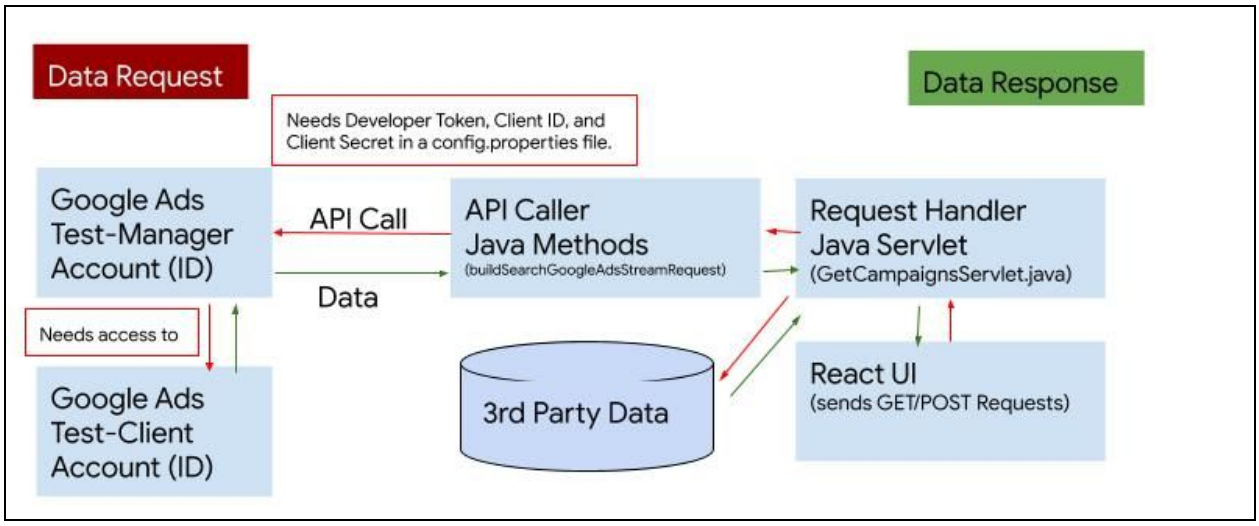
2. `config.properties`: stores your Developer Token, OAuth Client ID and Client Secret, which are necessary for making API calls. Keep this file local (you can add it to the .gitignore), and deploy these credentials to Datastore. Currently, we use the method `addCredentialsToDatastore()` in `OAuthServlet.java` to perform this task.
3. `appengine-web.xml`: The Google Ads API client library has a large JAR file that may be difficult to upload to Google App Engine. Should this error occur during deployment, set the enable-jar-splitting tag to true.

```
<staging>
  <enable-jar-splitting>true</enable-jar-splitting>
</staging>
```

4. `web.xml`: We use this file to manually map a 404-error code to the `RouterServlet.java`.

Design

Fork the repository here.



React Front End

File Structure and Important Files

/frontend/src

```
├──── Dashboard
│    ├──── Dashboard.js
│    ├──── AppBarWithDrawer
│    │    ├────(AppBarWithDrawer Files)
│    ├──── Pages (New pages get added in this folder)
│    │    ├──── PageRouter.js (This file conditionally renders pages based on the current URL)
│    │    ├──── LoginPage
│    │    │    ├──── (LoginPage Files)
│    │    ├──── QueryPage
│    │    │    ├──── (QueryPage Files)
│    │    ├──── ReportsPage
│    │    │    ├──── Reports(New reports get added in this folder)
│    │    │    │    ├──── (All Report Components)
│    │    │    ├──── (QueryPage Files)
│    └──── Utilities
│         ├──── Constants.js
│         ├──── Styles.js (Any custom styling should be defined here)
│         └──── Walkthrough.js (Tutorial text is defined here)
```

Page Structure

To see what makes up a page, let's look at an example: the Reports Page.

```
/frontend/src/Dashboard/Pages/ReportsPage
├──── ReportsPageLayout.js
├──── Reports
│    ├──── CampaignData.js
│    ├──── Chart.js
│    ├──── ClicksPerCampaignChart.js
│    ├──── Deposits.js
│    ├──── SentimentGraph.js
```

The main file here is `/ReportsPageLayout.js`. Each page should have a layout file with a similar structure. This file contains the grid layout defining how each of the reports should be displayed. More information on the grid api can be found here: Material UI | Grid API . Furthermore, in this file the Joyride tutorial is implemented.

The reports that `/ReportsPageLayout.js` uses are defined in the `/Reports/` folder. Let's look at a specific report, `/Reports/ClicksPerCampaignChart.js`. This chart, along with all the rest, is a functional component that is stateful using React Hooks. The two state variables in this file are **data**, and **state** and they are defined here:

```
const [data, setData] = useState([]); //data for chart and setter function
const [state, setState] = useState('loading');
```

The setters for these variables are used by the **useEffect** hook which asynchronously calls the backend through an axios call. Once the axios call is completed, the **data** and **state** variables will be updated according to what happened (loaded, or error). Before, during, and after the axios call, the function **pickContentToDisplay()** renders different components depending on the **state** variable. If all goes well (the axios call receives a successful response from the backend), the **state** variable is set to 'loaded', the **data** variable is set to the data from the backend, and **pickContentToDisplay()** displays the graph as expected.

Adding a New Report

To add a new report, you can follow the same structure as an existing report. Create a file in the `/Reports/` folder and have it export a React component. It can be either a function component or a class component. Then, modify `/ReportsPageLayout.js` to include your new report.

Adding a New Page

To add a new page, follow the same structure as the existing pages. Create a folder for your page in `/frontend/src/Dashboard/Pages/` and create a layout file within it. After you create the layout file, add that component along with its attributes to the **PagesWithAttributes** variable located in `/frontend/src/Dashboard/Utilities/Constants.js`. Your page should automatically show up in the side drawer and should load when you click on it.

Servlets

All servlets are in `server/src/main/java/com/google/sps/servlets/servlets/`, and we have some helper classes in `server/src/main/java/com/google/sps/servlets/utils/`.

GetCampaignsServlet

The main servlet used is `GetCampaignsServlet`. This servlet receives POST requests with a GAQL query, makes a call to the API with that query, and returns the response in JSON form. The main steps of this process are:

1. Retrieve the customer and login ID from Datastore, based on credentials (for more information, read the section on Datastore).
2. Run the example.
    a. Create a `GoogleAdsClient` and a `GoogleAdsServiceClient`.

   b. Formulate the query.
   c. Receive the response, and parse the protobuf response into JSON.
3. Process the messy JSON into a more intuitive response.
   a. Since the returned JSON includes many nested values, we take those values and format them more intuitively.
4. Return the JSON.
   a. Throughout the process, if any errors occur, a JSON will be returned with a status other than 200. Otherwise, the JSON meta will return with a status 200, which indicates everything has gone through smoothly.
5. (Extra) Create a spreadsheet, if the servlet is triggered by the **Export** button on the queries page.

We used `GetCampaigns` code from the Java client library as a starter template, but we modified the code in order to allow for more flexible GAQL queries, which was reflected in how we read and processed the protobuf response.

After Parsing from Protobuf

```
{"results": [{
 "campaign": {
  "resourceName": "customers/0/campaigns/0",
  "id": "0",
  "name": "Campaign 0"
 }
}, {
 "campaign": {
  "resourceName": "customers/0/campaigns/1",
  "id": "1",
  "name": "Campaign 1"
 }
}]}
```

After We Clean Up JSON

```
{"fieldmask": ["campaign.id","campaign.name"],
"response": [
 {
  "campaign.name":"Campaign 0",
  "campaign.id":"0"
 },
 {
  "campaign.name":"Campaign 1",
  "campaign.id":"1"
 }
],
"meta":
 {"status":"200"}
}
```

GetChart1Servlet

This servlet combines the Ads API with the Sentiment Analysis API as an example of how using the Ads API allows for greater flexibility in data analysis. Because we already have a functional servlet that takes in GAQL queries and returns responses, we extend that servlet and build off of that. The process is as such:

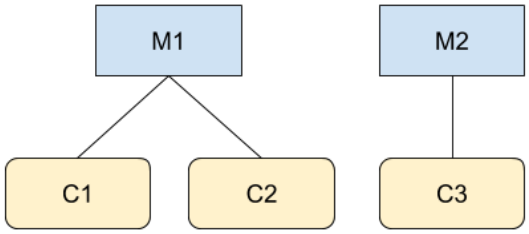
1. Follow the same procedure, calling the methods from super, until a valid return JSON has been generated.
2. Then, given this JSON, add Sentiment Analysis on the headlines.
3. Return the JSON.

The sentiment analysis API is just one example of how external APIs and Data can be used in conjunction with the Ads API, and we encourage this further exploration.

AccessibleCustomersServlet

This servlet handles the construction of account hierarchies after a user authenticates their Google Ads account. Most API calls will require a Login ID and Client Customer ID in order to query from the correct ad account. Following a front-end GET request to the servlet, the general process is:

1. Removing the previously set Login/Customer ID pair from Datastore.
2. Listing accessible customers (note that a Login ID is not required for this API call).
3. Iterate through top level accounts (i.e. M1, M2) and call `createCustomerClientToHierarchy()` on each account.
4. Recursively build out the account hierarchy to map top level accounts will all children beneath it.



Note on test accounts:

1. The API Client Code `ListAccessibleCustomers()` does not work on production-level ad accounts if the developer token is not approved
    a. You can still test some aspects of this method on a pending-approval developer token against test accounts.
2. `createCustomerClientToHierarchy()` contains API calls that do not work on test accounts at all
3. You can apply to get your developer token approved through your Google Ads manager account.

Adding New Servlets

Many operations can be made using our general `GetCampaignsServlet`, as it takes in GAQL If there are other calls you want to make, such as making mutative calls, we recommend you look at the Java Client Library, read through the code, and incorporate it into a servlet. The structure will likely be similar to our current servlets, and front-end calls to your new servlet will also likely follow the same format.

Authentication

Authentication is done through three main servlets: generating the authorization link, retrieving the refresh token, and logging out. When the user logs in, the `OAuthServlet`

generates an authorization link, which includes the redirect URI to the `CallbackServlet`. There, the user is prompted to accept, and afterwards, they are redirected to the `CallbackServlet`, which retrieves the access token, displays a status message to the user, and provides a link to go back to the web page.

If you are unfamiliar with the OAuth2.0 procedure, this provides a simplified overview of the flow.

OAuthServlet

Much of this servlet was built upon the Authentication code in the Java Client Library. You can run the code to initially obtain your credentials, but we adjusted the sample code to work with our backend. Because the sample opens up a local socket, we had to break the process into two separate servlets.

1. First, the servlet generates a state key, which is stored in Datastore along with the session ID. This is to prevent anti-forgery. This state key will be taken into account later in the `CallbackServlet`.
2. The servlet builds the base URI, which depends on the client ID and client Secret of the application, the SCOPES (which would be managing Ads campaigns and Google Sheets. This can be changed if other Google APIs are used), and the Callback URI, which points to our callback Servlet.
3. The servlet returns this link, and the middle-layer redirects the user.

CallbackServlet

Once the user logs in and accepts the scopes, they will be redirected to the callback servlet.

1. Once redirected, the Callback servlet will process the parameters passed into the GET request.
2. The servlet checks for any errors, some of which are listed here:
    a. If there is no code, then there is no way to generate a token. This returns an error.
    b. The servlet checks the state (which was generated in the `OAuthServlet` and now embedded as part of the GET request), and compares it with the states in Datastore. If there is no such state token corresponding to the user's current session, this may be a forgery attack, and the refresh token will not be generated. Here is an explanation of why the state token is necessary.
3. Otherwise, the `UserAuthorizer` uses the code (provided as a parameter) to generate the refresh token. The user is notified of the success and may renavigate back to the home page, where they are now authenticated.

LogoutServlet

Removes the OAuth state and Refresh token based on the current user's Session ID from Datastore.

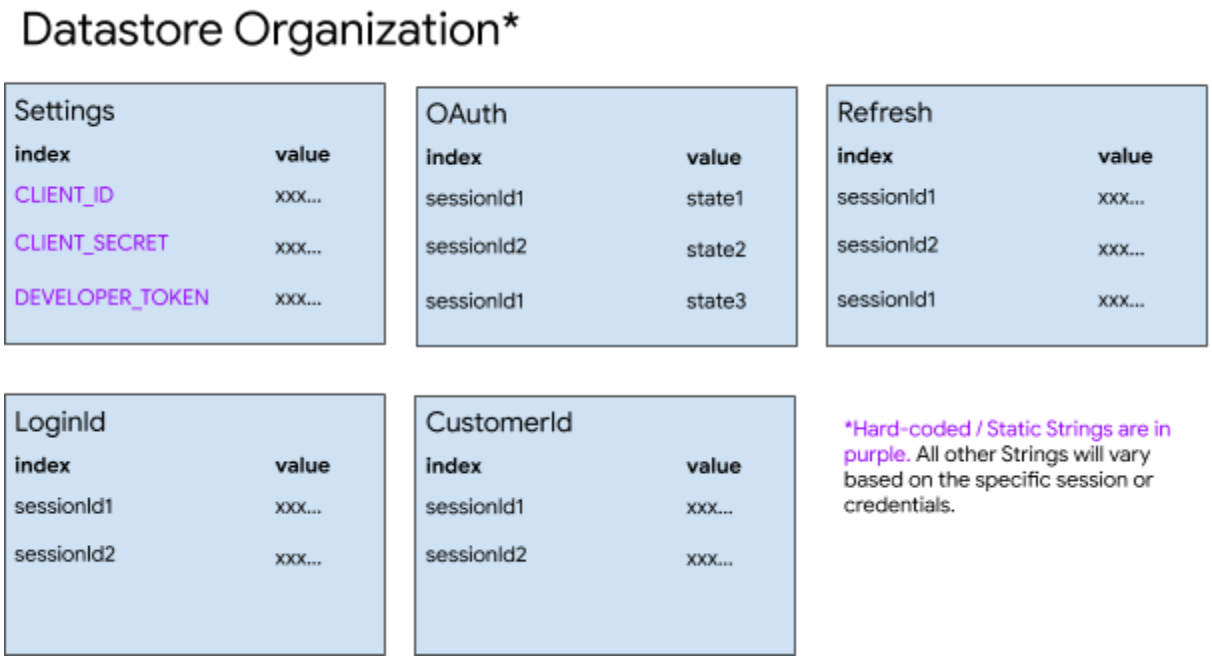
SetClientAccServlet

Either sets or removes the LoginId, CustomerId, and AccountName in Datastore depending on the front-end request.

Datastore

Because most of the data for each user can be generated by making the right calls to the Ads API, there is no need to store large amounts of user data for the application. However, there are a few important cases where we use Datastore:

1. Credentials
    a. GCP does not have an easy way to retrieve environment variables, so we store important credentials in Datastore. If you are running or deploying the app for the first time, you must initially add these important credentials to Datastore.
    b. These are stored with the label "Settings"
2. Authentication
    a. In between the OAuth and Callback servlets, the Datastore will hold the state token so the Callback server can be certain that the refresh token can be securely generated.
    b. State tokens are stored with the label "OAuth"
3. Session Management
    a. While a user is logged in, there are a variety of login and customer IDs they may choose from to query or show data. On the login page, the user must select which login / customer ID pair they want, and on the Dashboard and Query pages, the servlets will refer to values in Datastore when making calls to the Ads API.
    b. These are stored with labels "LoginId" and "CustomerId"



Since many different servlets will need to refer to Datastore for the current credentials and session information, we have created util classes with static methods to retrieve values or credentials from Datastore:

- DatastoreRetrieval.java
- CredentialRetrieval.java

Testing

Frontend Testing

The following commands related to testing can be run from within the client directory:

Command	Purpose
`npm run test`	Run all tests and see results
`npm run coverage`	Run all tests and see code coverage results
`Npm run test -- -u`	Update snapshots for all tests

Front end tests for each file are located in the same directory as each file that's being tested. The name of each test is the name of the file it is testing with the extension **.test.js** instead of

.js. For example, the test for `/ReportsPageLayout.js` is located at `/ReportsPageLayout.test.js`.

Each test file consists of 2 major parts. First, mounting the component to be tested, and second, checking to make sure the component behaves as expected. Sometimes, this involves mocking backend api calls. For this, we use Jest to mock the return values of the axios call being made. See this documentation for more details. Mounting the component is done through Enzyme using either shallow rendering or full rendering. The expect() function from Jest is used to make sure components behave expectedly.

Backend Testing

All backend tests and resources are in `server/src/test/java/com/google/sps/` and can be run by `mvn test`. The two test files test `GetCampaignsServlet` and `AccessibleCustomerServlet`, and the corresponding mocks are used to override and mock calls to the API. For the sake of reading in longer JSON text, there are a series of .txt files in `server/src/test/resources` which are referenced in tests.

To mock calls to the API, we use Mockito in most instances, and PowerMock to override final and static methods. In addition, we test the Mocked servlet instead of the original servlet in order to override methods that make calls to the API or builders. This allows us to control the responses of the API and work on testing our own methods.

Resources

Further Development

Using this webapp as a starter project, you can make more complicated API calls or incorporate more APIs. Here are some examples and ideas we have come up with, but because we have aimed to create a simple and flexible demo, we hope any new ideas will come naturally!

- Allow for sorting of columns on the Queries Page, similar to how the Campaign Data chart is sorted on the Dashboard page.
- Allow for mutative calls, i.e. change campaign status, change campaign name, etc.
    - This can be connected with the front-end. For instance, you can create checkboxes by the Campaign Data table, and remove or apply mutative operations to them.
    - Example code for Removing an Ad can be incorporated into a new servlet.
- Design a system to aggregate data from advertisement platforms outside of Google Ads and visualize comparisons on the dashboard.

    - Currently, we are able to export Query Data to Google Sheets, but reading in data would be just as useful!

Helpful Links

- Google Ads API Beta
- Google Ads Java Client Library
- Web App Demo
- Web App Demo Github

Resources We Used While Developing

- Alternative way to retrieve access token
- Google Group for Ads / Adwords API support
- Python GAQL Program-- used as reference and for experimentation
- Interactive GAQL Builder-- for finding different things to query
- Documentation of GAQL Query Structure
- Session Management in Java
- Java Quickstart for Google Sheets API