Write to Google Sheets

View on GitHub

In this tutorial, we'll show how to use Modal to schedule a daily report in a spreadsheet on Google Sheets that combines data from a PostgreSQL database with data from an external API.

In particular, we'll extract the city of each user from the database, look up the current weather in that city, and then build a count/histogram of how many users are experiencing each type of weather.

Entering credentials

We begin by setting up some credentials that we'll need in order to access our database and output spreadsheet. To do that in a secure manner, we log in to our Modal account on the web and go to the Secrets section.

Database

First we will enter our database credentials. The easiest way to do this is to click **New secret** and select the **Postgres compatible** Secret preset and fill in the requested information. Then we press **Next** and name our Secret `example-postgres-secret` and click **Create**.

Google Sheets/GCP

We'll now add another Secret for Google Sheets access through Google Cloud Platform. Click **New secret** and select the Google Sheets preset.

In order to access the Google Sheets API, we'll need to create a *Service Account* in Google Cloud Platform. You can skip this step if you already have a Service Account json file.

1. Sign up to Google Cloud Platform or log in if you haven't (https://cloud.google.com/).
2. Go to https://console.cloud.google.com/.
3. In the navigation pane on the left, go to **IAM & Admin** > **Service Accounts**.
4. Click the **+ CREATE SERVICE ACCOUNT** button.
5. Give the service account a suitable name, like "sheet-access-bot". Click **Done**. You don't have to grant it any specific access privileges at this time.
6. Click your new service account in the list view that appears and navigate to the **Keys** section.
7. Click **Add key** and choose **Create new key**. Use the **JSON** key type and confirm by clicking **Create**.
8. A json key file should be downloaded to your computer at this point. Copy the contents of that file and use it as the value for the `SERVICE_ACCOUNT_JSON` field in your new secret.

We'll name this other Secret "my-gsheets-secret".

Now you can access the values of your Secrets from Modal Functions that you annotate with the corresponding `modal.Secret` s, e.g.:

```
import os

import modal

app = modal.App("example-db-to-sheet")


@app.function(secrets=[modal.Secret.from_name("example-postgres-secret")])
def show_host():
    # automatically filled from the specified secret
    print("Host is " + os.environ["PGHOST"])
```

In order to connect to the database, we'll use the `psycopg2` Python package. To make it available to your Modal Function you need to supply it with an `image` argument that tells Modal how to build the container image that contains that package. We'll base it off of the `Image.debian_slim` base image that's built into Modal, and make sure to install the required binary packages as well as the `psycopg2` package itself:

```
pg_image = (
    modal.Image.debian_slim(python_version="3.11")
    .apt_install("libpq-dev")
    .pip_install("psycopg2~=2.9.9")
)
```

Since the default keynames for a **Postgres compatible** secret correspond to the environment variables that `psycopg2` looks for, we can now easily connect to the database even without explicit credentials in your code. We'll create a simple function that queries the city for each user in the `users` table.

```
@app.function(
    image=pg_image, secrets=[modal.Secret.from_name("example-postgres-secret")]
)
def get_db_rows(verbose=True):
    import psycopg2

    conn = psycopg2.connect()  # no explicit credentials needed
    cur = conn.cursor()
    cur.execute("SELECT city FROM users")
    results = [row[0] for row in cur.fetchall()]
    if verbose:
        print(results)
    return results
```

Note that we import `psycopg2` inside our function instead of the global scope. This allows us to run this Modal Function even from an environment where `psycopg2` is not installed. We can test run this function using the `modal run` shell command: `modal run db_to_sheet.py::app.get_db_rows`.

To run this function, make sure there is a table called `users` in your database with a column called `city`. You can populate the table with some example data using the following SQL commands:

```
CREATE TABLE users (city TEXT);
INSERT INTO users VALUES ('Stockholm,,Sweden');
INSERT INTO users VALUES ('New York,NY,USA');
INSERT INTO users VALUES ('Tokyo,,Japan');
```

Applying Python logic

For each row in our source data we'll run an online lookup of the current weather using the http://openweathermap.org API. To do this, we'll add the API key to another Modal Secret. We'll use a custom secret called "weather-secret" with the key `OPENWEATHER_API_KEY` containing our API key for OpenWeatherMap.

```
requests_image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "requests~=2.31.0"
)
```

```
@app.function(
    image=requests_image, secrets=[modal.Secret.from_name("weather-secret")]
```

```
)
def city_weather(city):
    import requests

    url = "https://api.openweathermap.org/data/2.5/weather"
    params = {"q": city, "appid": os.environ["OPENWEATHER_API_KEY"]}
    response = requests.get(url, params=params)
    weather_label = response.json()["weather"][0]["main"]
    return weather_label
```

We'll make use of Modal's built-in `function.map` method to create our report. `function.map` makes it really easy to parallelize work by executing a Function on every element in a sequence of data. For this example we'll just do a simple count of rows per weather type — answering the question "how many of our users are experiencing each type of weather?".

```
from collections import Counter


@app.function()
def create_report(cities):
    # run city_weather for each city in parallel
    user_weather = city_weather.map(cities)
    count_users_by_weather = Counter(user_weather).items()
    return count_users_by_weather
```

Let's try to run this! To make it simple to trigger the function with some predefined input data, we create a "local entrypoint" that can be run from the command line with

```
modal run db_to_sheet.py
```

```
@app.local_entrypoint()
def main():
    cities = [
        "Stockholm,,Sweden",
        "New York,NY,USA",
        "Tokyo,,Japan",
    ]
    print(create_report.remote(cities))
```

Running the local entrypoint using `modal run db_to_sheet.py` should print something like: `dict_items([('Clouds', 3)])`. Note that since this file only has a single app, and the app has only one local entrypoint we only have to specify the file to run it - the function/entrypoint is inferred.

In this case the logic is quite simple, but in a real world context you could have applied a machine learning model or any other tool you could build into a container to transform the data.

Sending output to a Google Sheet

We'll set up a new Google Sheet to send our report to. Using the "Sharing" dialog in Google Sheets, share the document to the service account's email address (the value of the `client_email` field in the json file) and make the service account an editor of the document.

You may also need to enable the Google Sheets API for your project in the Google Cloud Platform console. If so, the URL will be printed inside the message of a 403 Forbidden error when you run the function. It begins with https://console.developers.google.com/apis/api/sheets.googleapis.com/overview.

Lastly, we need to point our code to the correct Google Sheet. We'll need the *key* of the document. You can find the key in the URL of the Google Sheet. It appears after the `/d/` in the URL, like: `https://docs.google.com/spreadsheets/d/1wOktal.....IJR77jD8Do` .

We'll make use of the `pygsheets` python package to authenticate with Google Sheets and then update the spreadsheet with information from the report we just created:

```
pygsheets_image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "pygsheets~=2.0.6"
)


@app.function(
    image=pygsheets_image,
    secrets=[modal.Secret.from_name("my-gsheets-secret")],
)
def update_sheet_report(rows):
    import pygsheets

    gc = pygsheets.authorize(service_account_env_var="SERVICE_ACCOUNT_JSON")
    document_key = "1JxhGsht4wltyPFFOd2hP0eIv6lxZ5pVxJN_ZwNT-l3c"
    sh = gc.open_by_key(document_key)
    worksheet = sh.sheet1
    worksheet.clear("A2")

    worksheet.update_values("A2", [list(row) for row in rows])
```

At this point, we have everything we need in order to run the full program. We can put it all together in another Modal function, and add a `schedule` argument so it runs every day automatically:

```
@app.function(schedule=modal.Period(days=1))
def db_to_sheet():
    rows = get_db_rows.remote()
    report = create_report.remote(rows)
    update_sheet_report.remote(report)
    print("Updated sheet with new weather distribution")
    for weather, count in report:
        print(f"{weather}: {count}")
```

This entire app can now be deployed using `modal deploy db_to_sheet.py`. The apps page shows our cron job's execution history and lets you navigate to each invocation's logs. To trigger a manual run from your local code during development, you can also trigger this function using the cli: `modal run db_to_sheet.py::db_to_sheet`

Note that all of the `@app.function()` annotated functions above run remotely in isolated containers that are specified per function, but they are called as seamlessly as if we were using regular Python functions. This is a simple showcase of how you can mix and match Modal Functions that use different environments and have them feed into each other or even call each other as if they were all functions in the same local program.