Streaming Weather API with GCP

Github Repository: https://github.com/urjasvit/WeatherStreamingDataPipeliningGCP

07.16.2021

—

Urjasvit Sinha
MS Data Analytics

Overview

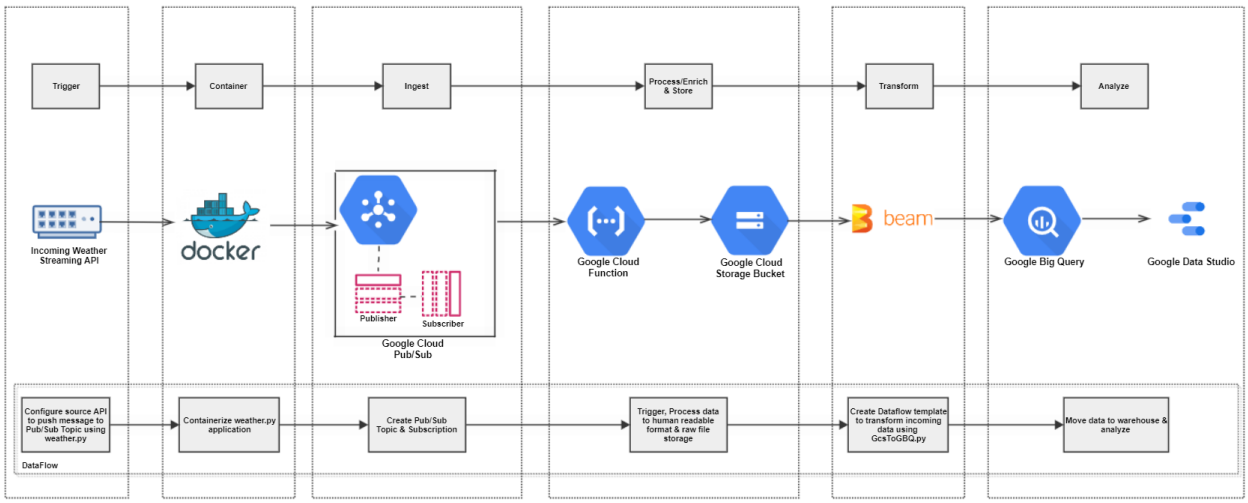
This project puts forth an idea of engineering and processing the real-time incoming weather updates from all over the country and creating reports for further analyses. It proposes to use the various services provided by Google on their Cloud Platform to develop a streaming analytics solution that is secure, reliable and is easily scalable from the instant it's generated.

Goals

1. Develop an end-to-end project to analyse real-time streaming API data using the services provided by Google Cloud Platform.
2. Ingest, process, and analyze real-time events to make data more organized, useful and accessible.
3. Showcasing high-level knowledge of the data engineering concepts to create a modular, reliable and scalable project.

Architecture Diagram and Implementation

Below is the architecture and data flow diagram that provides the overview of the implementation process to create a data engineering flow from fetching the data to analyzing it.



Trigger, Containerize and Ingest:

The data from the streaming weather API is fetched using python application(weather.py) which then publishes the real-time weather update to Google Cloud Pub/Sub Topic. This python application is containerized via Docker. It allows us to manage the infrastructure in the same ways that an application is managed.

Firstly, a weather class is created and initialization of the project details is done inside the constructor.

```
class weather():
    def __init__ (self):
        credentials = 'D:/NEU/EGEN/Project/weatherpubsubkey.json'
        os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = credentials
        self.project_id = "gcptraining-319415"
        self.topic_id = "weathercalls"
        self.publisher = pubsub_v1.PublisherClient()
        self.topic_path = self.publisher.topic_path(self.project_id, self.topic_id)
        self.publish_futures = []
```

Inside the weather class a function is defined that requests and fetches the data from the source url in json format.

```
def get_weather_api(self):
        try:
          response = requests.get("https://api.openweathermap.org/data/2.5/weather?id=4930956&units=metric&&appid=
          data = json.loads(response.text)
          requests_cache.install_cache()
          print(data)
          return (data)

        except (ConnectionError, Timeout, TooManyRedirects) as e:
          print(e)
```

After fetching the data, the class object then calls publish_messages() function which publishes the fetched message from the url to Google Cloud Pub/Sub Topic using the credentials provided during the class object initialization.

```python
def publish_messages(self, data):
    """Publishes multiple messages to a Pub/Sub topic with an error handler.."""

    def get_callback(publish_future, data):
        def callback(publish_future):
            try:
                # Wait 60 seconds for the publish call to succeed.
                print(publish_future.result(timeout=60))
            except futures.TimeoutError:
                print(f"Publishing {data} timed out.")

        return callback

    publish_future = self.publisher.publish(self.topic_path, json.dumps(data).encode("utf-8"))
    # Non-blocking. Publish failures are handled in the callback function.

    publish_future.add_done_callback(get_callback(publish_future, data))
    self.publish_futures.append(publish_future)

    print(f"Published messages to {self.topic_path}.{data}")
```

The requirements of the python application is freezed and then using Docker the application gets containerized.

Process, Enrich and Store:

As the message is received by the Pub/Sub Topic, the Cloud Function which acts as a trigger takes the incoming weather data and transforms it into human readable format. Another use of Cloud Function is to send this transformed data(.csv) to the data lake, which in this case is Google Cloud Storage bucket. This storage bucket acts as a raw file storage system providing the data back-up and security in case of any disaster or deletion.

The entry point for the Cloud Function is hello_pubsub() which contains the event and context as parameters. This basically gets triggered whenever a message is deployed to Pub/Sub Topic. Event contains the payload/data and context has the metadata for the event.

```
def hello_pubsub(event, context):
    """Triggered from a message on a Cloud Pub/Sub topic.
    Args:
         event (dict): Event payload.
         context (google.cloud.functions.Context): Metadata for the event.
    """
    logging.basicConfig(level=logging.INFO)
    service=LoadToStorage(event,context)
    message=service.getMsgData()
    df=service.payloadToDf(message)
    timestamp=str(int(time.time()))
    service.uploadToBucket(df,"weather-calls"+timestamp)
```

The object for the class LoadToStorage initializes the constructor that contains the output bucket address. When the object invoked getMsgData() the data from the Pub/Sub Topic is retrieved and returned as string.

```
class LoadToStorage:
    def __init__(self,event,context):
        self.event=event
        self.context=context
        self.bucket_name='weathercalls-bucket'

    def getMsgData(self) -> str:
        logging.info("Function triggered, retrieving data")
        message_chunk=base64.b64decode(self.event['data']).decode('utf-8')
        logging.info("Datapoint validated")
        return message_chunk
```

To convert the string data to human readable format(dataframe in this case), PayloadtoDf() is invoked. This function processes the dictionary object and converts the data into dataframe and returns it to the hello_pubsub.

Lastly, the timestamp of the data is generated so as to append the csv to the data lake storage bucket.

```
def uploadToBucket(self,df,filename):
    storage_client=Client()
    bucket=storage_client.bucket(self.bucket_name)
    blob=bucket.blob(f"{filename}.csv")
    blob.upload_from_string(data=df.to_csv(index=False),content_type='text/csv')
    logging.info("File uploaded to bucket")
```

Transform and Analyze:

The Apache Beam then applies the transformation steps on the raw data present in the bucket so that it can be stored in the Google Big Query warehouse. The python script(GcsToGBQ.py) uses the Apache Beam SDK and has the data flow steps used for transformation and to create/append the data into BigQuery. Lastly, the SQL queries are run on BigQuery and output data is exported to Google Data Studio is used for analysing the weather trends and generating reports.

As for the code implementation of GcsToGBQ.py, first the class LoadtoBQ constructor is initialized by providing the GCP project credentials along with the schema for the Big Query table destination.

```
class LoadtoBQ:

    def __init__(self):
        self.table_spec = bigquery.TableReference(
            projectId='gcptraining-319415',
            datasetId='weathercallsbq',
            tableId='gbq'
        )

        self.schema = 'city_name:string, \
                       temperature:float, \
                       minimum_temperature:float, \
                       maximum_temperature:float, \
                       humidity:float, \
                       visibility:integer, \
                       wind_speed:float, \
                       date_time:timestamp,\
                       conditions:string'
        self.log = {
            'city_name':' ',
            'temperature':' ',
            'minimum_temperature':' ',
            'maximum_temperature':' ',
            'humidity':' ',
            'visibility':' ',
            'wind_speed':' ',
            'date_time':' ',
            'conditions':' ',
        }
```

The parse() parses the incoming data from the gcs bucket and parses it line by line and creates data in the form that Bigquery table requires.

```
def parse(self, line):

    data = line.split(',')
    self.log['city_name']=data[7]
    self.log['temperature']=data[0]
    self.log['minimum_temperature']=data[1]
    self.log['maximum_temperature']=data[2]
    self.log['humidity']= data[3]
    self.log['visibility']=data[4]
    self.log['wind_speed']=data[5]
    self.log['date_time']=(datetime.strptime(str(datetime.utcfromtimestamp(int(data[6])).strftime('%m-%d-%Y %H:%M'))
    if(data[8]==data[9].capitalize()):
        self.log['conditions']=data[8]
    else:
        self.log['conditions']=data[8]+", "+data[9].capitalize()
    return self.log
```

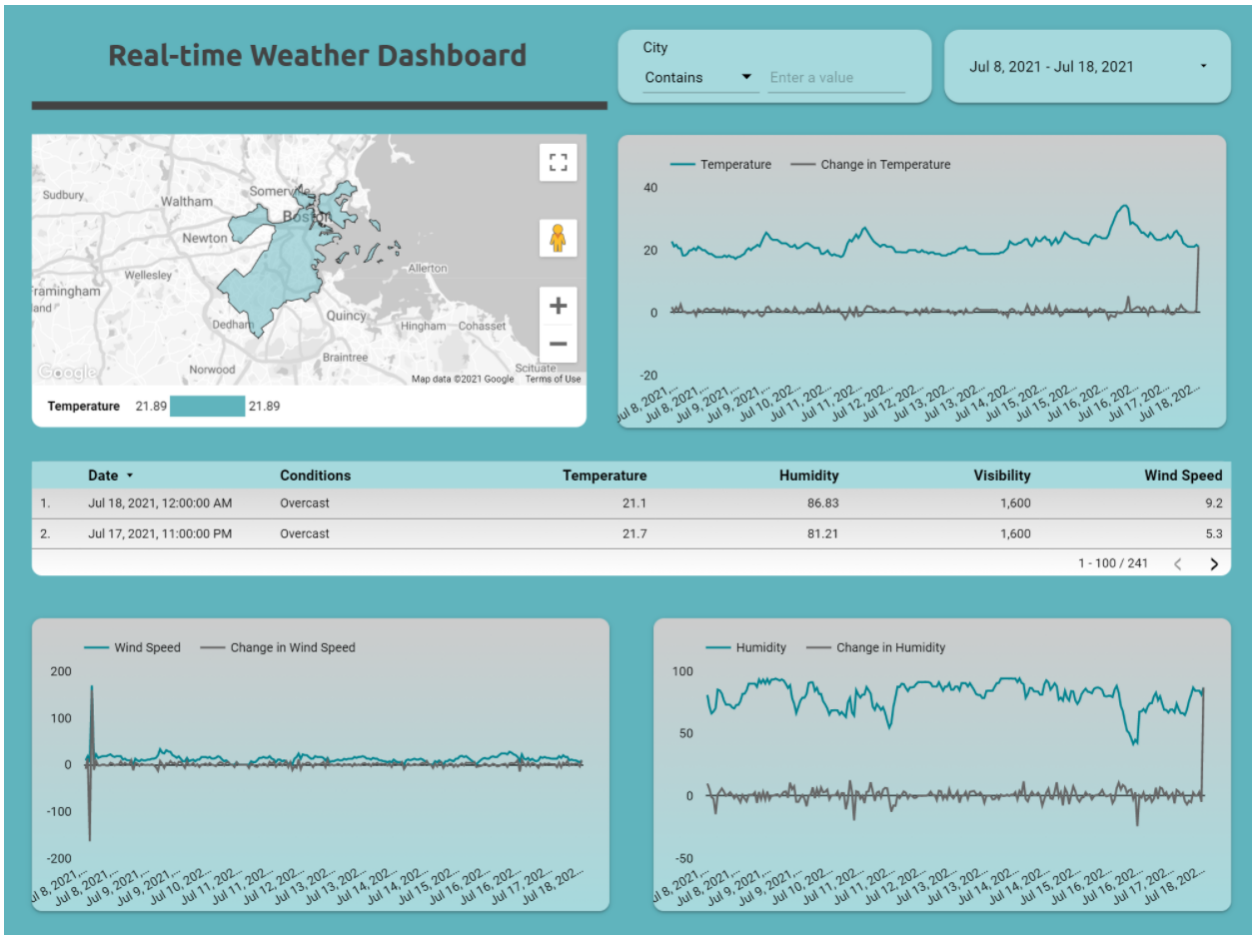
The run() is the main entry point that gets prompted when the python application is run. It has the argument parser using which the gcs bucket url input is specified and the data

from that is parsed. This parsed data goes through the pipeline steps which has 3 main steps:

1. Reading from the bucket.
2. Mapping the input from the bucket in form that can be used for storing in Bigquery.
3. Defining the storage schema of the Bigquery along with the dispositions and custom Gcs location to store the temporary data between fetching from GCS to loading into Bigquery. This temporary location can be used while checking for errors in the pipeline.

```
def run(argv=None, save_main_session=True):
    parser = argparse.ArgumentParser()

    parser.add_argument(
        '--input',
        dest='input',
        default='gs://weathercalls-bucket/weather-calls*',
        help='Input file to process.')


    known_args, pipeline_args = parser.parse_known_args(argv)

    pipeline_options = PipelineOptions(pipeline_args)
    pipeline_options.view_as(SetupOptions).save_main_session = save_main_session

    with beam.Pipeline(options=pipeline_options) as p:

        obj = LoadtoBQ()
        quotes = (
                p
                | 'Read' >> ReadFromText(known_args.input ,skip_header_lines=1)
                | 'Parse Log' >> beam.Map(lambda line: obj.parse(line))
        )

        #quotes | 'Write' >> WriteToText('output')
        quotes | beam.io.gcp.bigquery.WriteToBigQuery(
            obj.table_spec,
            schema=obj.schema,
            write_disposition=beam.io.BigQueryDisposition.WRITE_APPEND,
            create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
            custom_gcs_temp_location = 'gs://weathercalls-bucket/tmp',
#             method = 'STREAMING_INSERTS'
        )
```

Finally when the streaming data enters into the Bigquery warehouse, using the Bigquery connector the data is analyzed on Data Studio. Below is the real-time analysis of the incoming streaming data.



Resources

Google Cloud Platform:

- https://cloud.google.com/docs

Apache Beam:

- https://medium.com/swlh/apache-beam-google-cloud-dataflow-and-creating-custom-templates-using-python-c666c151b4bc
- https://beam.apache.org/documentation
- https://github.com/ankitakundra/GoogleCloudDataflow/blob/master/dataingestion.py