Atlan-Backend-Challenge

The Atlan Backend Challenge is to design an **ecosystem for integrations** where each use case can be plugged-in without an overhaul on the backend.

My solution

I have designed the **Pub/Sub event-driven architecture** approach for the solution. The problem statement requires us to create a system that facilitates eventual **consistency**, **fail-safety**, high **availability**, and **scalability**.

The **Cloud Pub/Sub** by Google that I've used in my solution guarantees all of these characteristics.

- If one of the subscribers/services goes down, the **message is retained** in the Topic so that once the service is up and running again, the message is delivered to it. This ensures eventual **consistency** and **fail-safely**.
- **Multiple instances** of a service may be deployed and subscribed to a common topic, so if the service fails in any zone, the others can pick up the load automatically. This ensures high **availability** and **reliability**.

Technologies used

- Node.js, Express, Google Cloud Services
- Firebase Cloud Firestore for data storage
- Sheets API for implementing one of the services
- Twilio API for SMS service

Architechture (Plugin architecture)

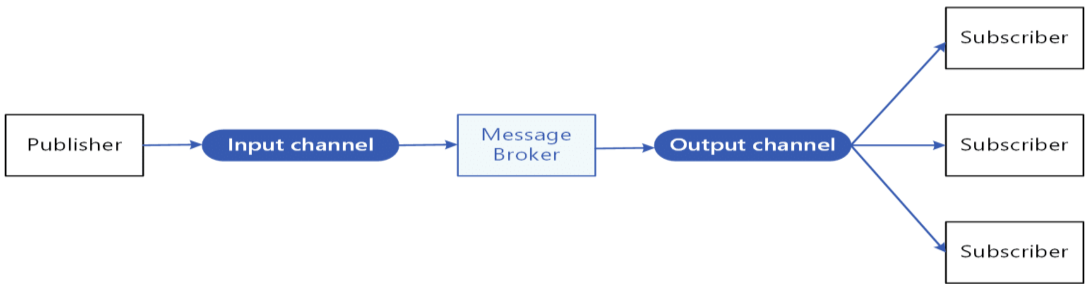


Fig : Data flow diagram

In our case the **publisher** is our **main server** which is doing all the queries related to database whose logic is written in **index.js,** our message broker is **Pub/Sub** and our **subscribers** are **Sheet-plugin , Log plugin** and **SMS plugin.**

How I Implemented Pub/Sub

To publish the response to a topic:

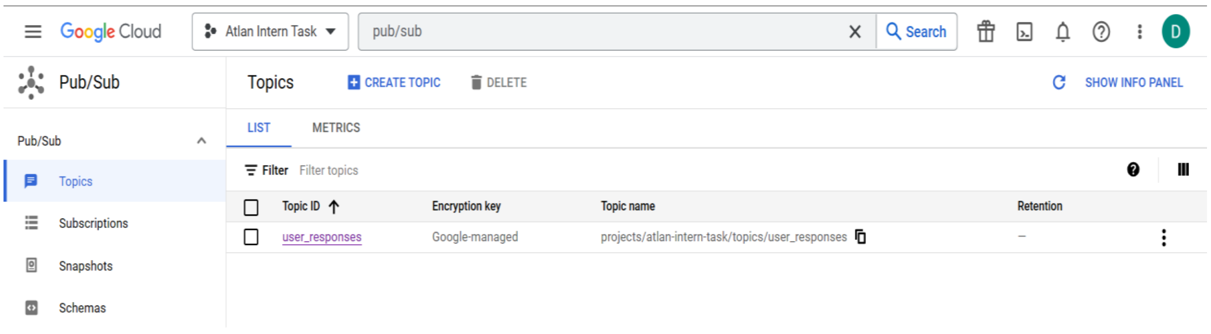
```
const publishNewMessage = async (client,topicName,payloadData)=>{
    const data = Buffer.from(JSON.stringify(payloadData));
    const message = await client.topic(topicName).publish(data);

    console.log('message published ' + message)
    return message
}

module.exports = {
    publishNewMessage
}
```

The above piece of code publishes a message to the topic. The above snippet of code can be found in **src/publisher/pub.js**

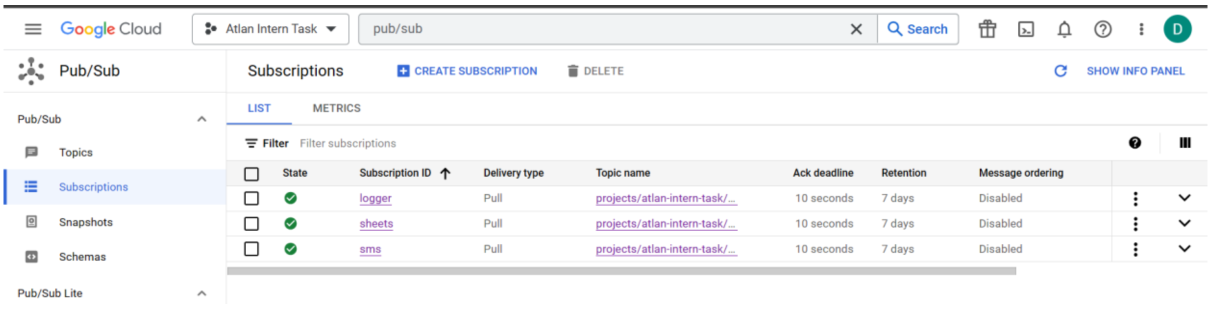
To create a topic in Pub/Sub:



Topic ID	Encryption key	Topic name	Retention
user_responses	Google-managed	projects/atlan-intern-task/topics/user_responses	–

A new topic can be created from create topic button. This is the one and only topic I have created.

To subscribe to a topic for the response:



State	Subscription ID	Delivery type	Topic name	Ack deadline	Retention	Message ordering
✓	logger	Pull	projects/atlan-intern-task/...	10 seconds	7 days	Disabled
✓	sheets	Pull	projects/atlan-intern-task/...	10 seconds	7 days	Disabled
✓	sms	Pull	projects/atlan-intern-task/...	10 seconds	7 days	Disabled

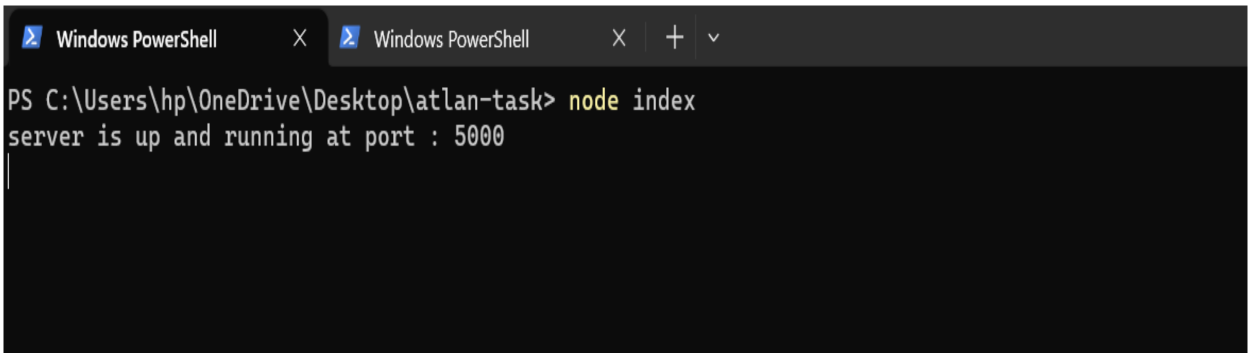
Above image is the snapshot of my subscriptions tab in Pub/Sub window at google cloud console. A new subscriber can be created by clicking the create subscription option and filling a basic form after it.

Managing fail safety

- Cloud Pub/Sub inherently provides fail-safety mechanisms. When a message is published, it is retained within the topic until all subscribed services have both received and confirmed receipt of the message.
- If a service like the Google Sheets service experiences an outage, the message remains preserved within a queue. Upon service restoration, the queued message is then successfully received. Moreover, the option to operate multiple service instances is available, distributing the workload, a configuration that can be further optimized by implementing load balancers.

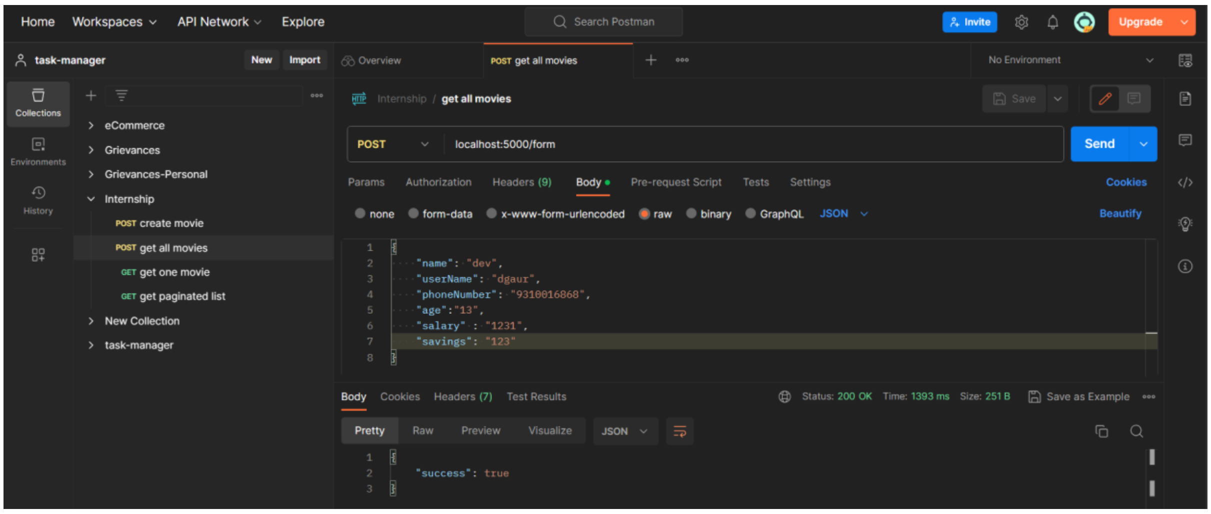
Relevant Screenshots

When the Google Sheets service has not started:



```
PS C:\Users\hp\OneDrive\Desktop\atlan-task> node index
server is up and running at port : 5000
```

Response sent via Postman :

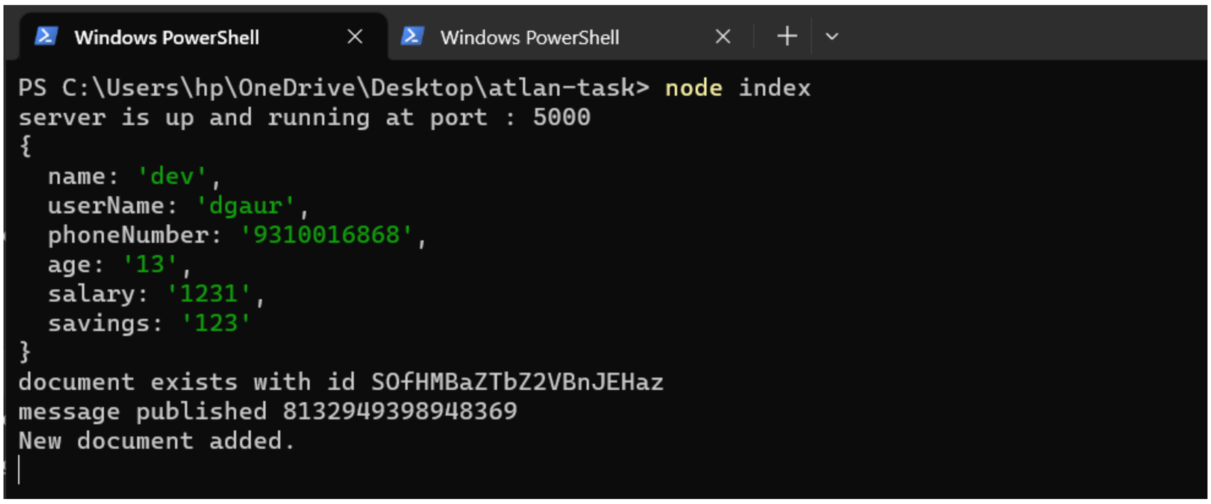

POST localhost:5000/form

```
{
    "name": "dev",
    "userName": "dgaur",
    "phoneNumber": "9310016868",
    "age":"13",
    "salary" : "1231",
    "savings": "123"
}
```

Status: 200 OK Time: 1393 ms Size: 251 B

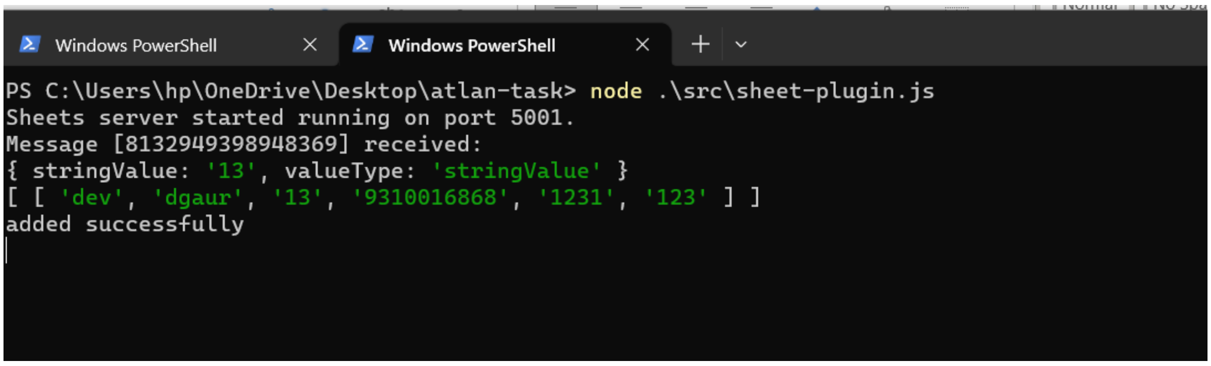
```
{
    "success": true
}
```

Message sent through topic:



```
PS C:\Users\hp\OneDrive\Desktop\atlan-task> node index
server is up and running at port : 5000
{
  name: 'dev',
  userName: 'dgaur',
  phoneNumber: '9310016868',
  age: '13',
  salary: '1231',
  savings: '123'
}
document exists with id SOfHMBaZTbZ2VBnJEHaz
message published 8132949398948369
New document added.
```

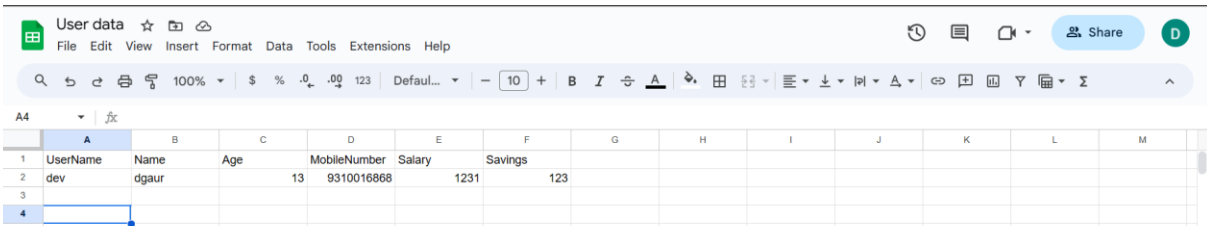
When the Google Sheets service has started:



```
PS C:\Users\hp\OneDrive\Desktop\atlan-task> node .\src\sheet-plugin.js
Sheets server started running on port 5001.
Message [8132949398948369] received:
{ stringValue: '13', valueType: 'stringValue' }
[ [ 'dev', 'dgaur', '13', '9310016868', '1231', '123' ] ]
added successfully
```

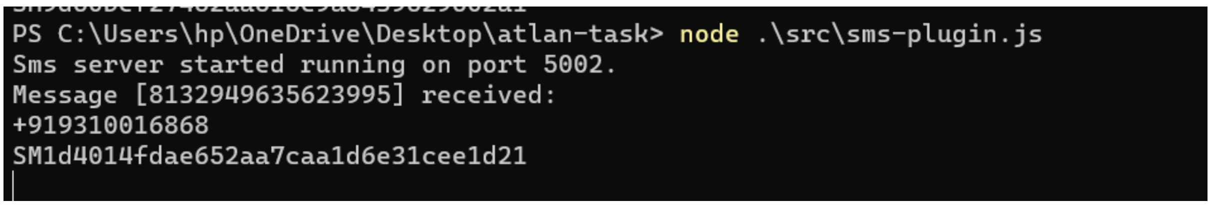
Here we can match the ids of messages and hence know that we are sending and receiving the same message through pub/sub.

Excel sheet response:



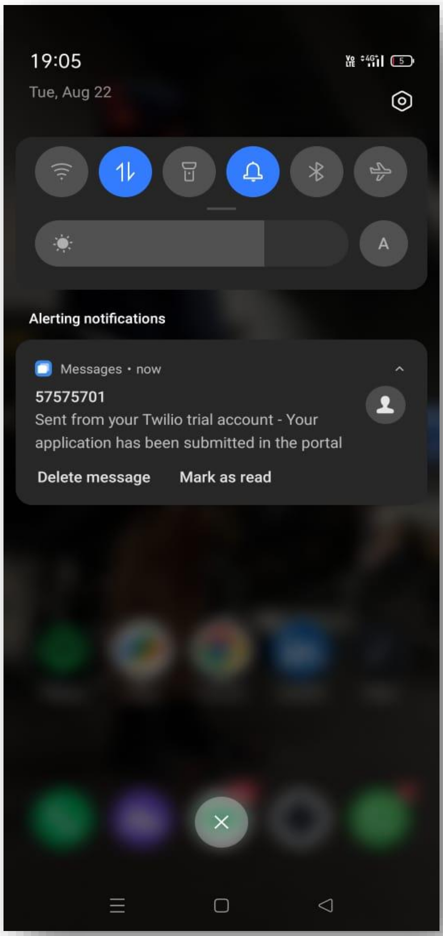
	A	B	C	D	E	F
1	UserName	Name	Age	MobileNumber	Salary	Savings
2	dev	dgaur	13	9310016868	1231	123
3						
4						

When sms server is started:



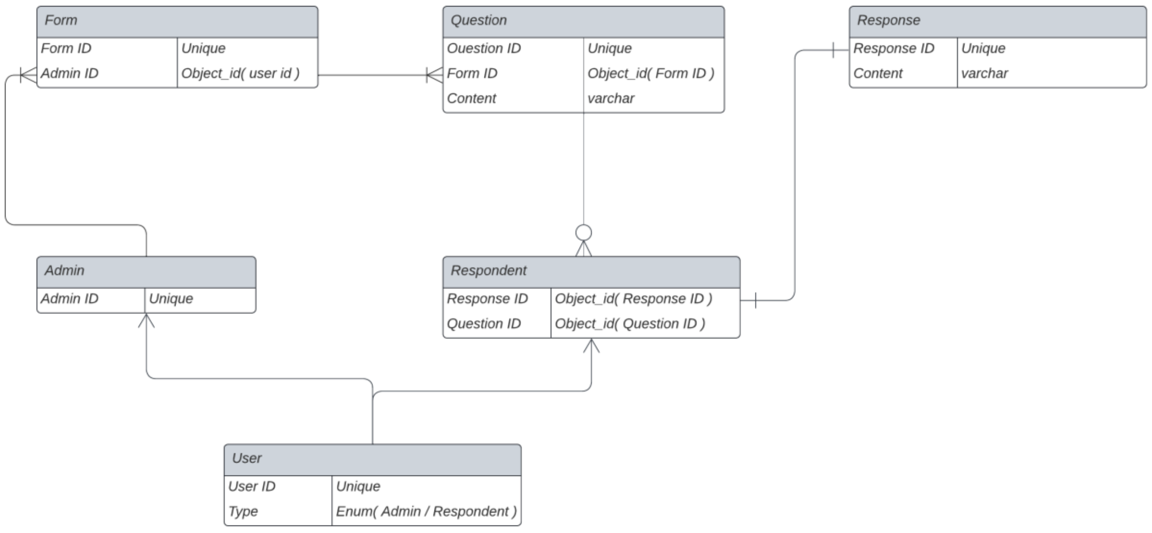
```
PS C:\Users\hp\OneDrive\Desktop\atlan-task> node .\src\sms-plugin.js
Sms server started running on port 5002.
Message [8132949635623995] received:
+919310016868
SM1d4014fdae652aa7caa1d6e31cee1d21
```

Message received at my number:



57575701
Sent from your Twilio trial account - Your application has been submitted in the portal

Database Schema (for forms, questions and responses)



Benchmarking

Pub/Sub Messaging is based on publishers publishing messages and subscribers subscribing to those messages. Thus, it makes sense to **benchmark the system based on publisher and subscriber throughput**.

1) **Thread parallelism** can be used to process multiple messages on different cores of the CPU. Since a higher number of generated events can be processed with increased CPU cores, the publisher throughput should increase. Therefore, employing parallelism is an effective strategy to benchmark publisher and subscriber throughput.

2) We can publish batches of messages (e.g., 1000 messages or 5 MB per batch). However, large batch settings may increase per-message latency.

Logging

The following is the workflow that I have followed to demonstrate my idea. *Please note that I have only subscribed the Logging Plugin to the events emitted by the central server in the current codebase. This should suffice the proof of concept.*

The implementation of event logging can be found in **src/logger/logger.js** and **src/log-plugin.js**

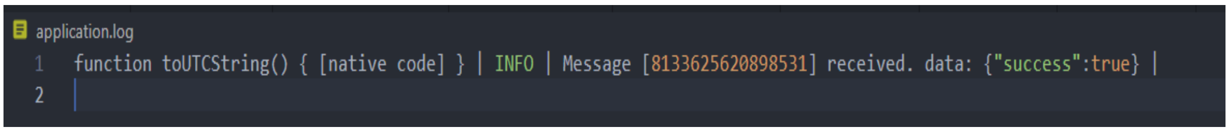
Log Format Used:

1. UTC Date and Time

2. Type of the log (Info, Debug, Error)

3. The event that occurred

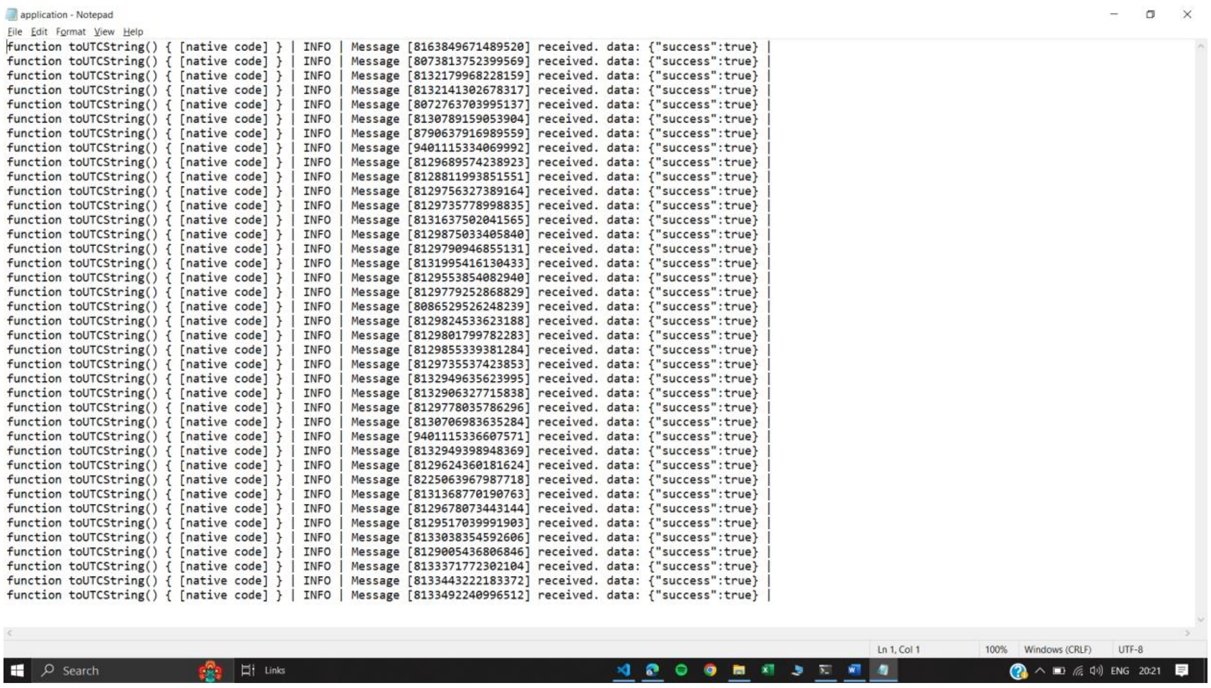
4. Additional data of the event



application.log

```
function toUTCString() { [native code] } | INFO | Message [8133625620898531] received. data: {"success":true} |
```

Snapshot: Logging done in terminal



application - Notepad

```
function toUTCString() { [native code] } | INFO | Message [8163849671489520] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8073813752399569] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8132179968228159] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8132141302678317] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8072763703995137] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8130789159053904] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8790637916989559] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [9401115334069992] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129689574238923] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8128811993851551] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129756327389164] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129735778998835] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8131637502041565] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129875033405840] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129790946855131] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8131995416130433] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129553854082940] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129779252868829] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8086529526248239] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129824533623188] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129801799782283] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129855339381284] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129735537423853] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8132949635623995] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8132906327715838] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129778035786296] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8130706983635284] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [9401115336607571] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8132949398948369] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129624360181624] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8225063967987718] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8131368770190763] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129678073443144] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129517039991903] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8133038354592606] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8129005436806846] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8133371772302104] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8133443222183372] received. data: {"success":true} |
function toUTCString() { [native code] } | INFO | Message [8133492240996512] received. data: {"success":true} |
```

Snapshot: Full log file

Health Monitoring

Based on the Pub/Sub model, I have come up with the following ideas to monitor the system's health.

1) **To keep the subscribers healthy:**

- **Monitor number of undelivered messages**: The unusually large value of this metric in the context of our system can generate insights into system health. For instance, if the number of undelivered messages grows with time, it can mean that the subscriber is not keeping up with message volume.

Adding multiple instances of the subscriber or looking up for bugs in the code that prevent it from successfully acknowledging messages can help solve the issue.

- **Monitor acknowledgement deadline expiration**: A robust Pub/Sub model should allow subscribers a limited amount of time to acknowledge a given message before re-delivering the message. It can be helpful to measure the rate at which subscribers miss the ack deadline. If this rate is high, this could potentially mean:

  - The subscribers are under-provisioned.

  - Each message takes longer to process than the message ack deadline.

  - Some messages consistently crash the client.

- **Monitor forwarded undelivered messages**: If the Pub/Sub service attempts to deliver a message, but the subscriber can't acknowledge it, Pub/Sub may forward the undeliverable message to another topic (for example, dead-messages topic). The delivery to the "dead-messages" topic would occur only when the limit of delivery attempts to the subscriber is exceeded. The system health can then be analysed by monitoring the undeliverable messages.

2) **<u>To keep the publishers healthy:</u>**

The channel where the message is published is at the core of the Pub/Sub model. Therefore, it is equally important to ascertain whether the publisher is working as expected. Its performance can be monitored by a kind of send **_request_count, grouped by response_code** metric. This metric will indicate whether the system is healthy and accepting requests by providing the rate of retriable errors.

Limitations of APIs Used

Limitations on the Google Sheets API

There are some limitations on the usage of Sheets API, which can hinder the performance of the Google Sheets plugin.

- Sheets API has a **limit of 500 requests per 100 seconds** per project and **100 requests per 100 seconds** per user.

- It can only be increased to **2500 requests per account** even with the paid quota.

- If the Sheets API is down, it means the Google Sheets Plugin cannot function. Even when the service is up, the number of spreadsheets that can be created depends on the membership plan of the Google account. Free Google Accounts give **15 GB of storage**.

Limitations on Twilio API

There are some limitations on the usage of Twilio API, which can hinder the performance of the SMS plugin.

- Twilio's platform supports long messages up to 1600 characters across all Programmable Messaging channels, including SMS. However, for SMS messaging, Twilio recommends sending messages that are no more than 320 characters to ensure the best deliverability and user experience.

- Only verified numbers can receive messages from Twilio API. So every mobile number has to be verified before sending the SMS. Message to anyone feature is only for paid quota.

Note:- For setting up the project refer to Readme.md

Important Links

User data - Google Sheets

Regards,

Dev Gaur

(dgaur20102002@gmail.com)

(+919311016868)