WHAT IS PUB/SUB? PUBLISH/SUBSCRIBE MESSAGING EXPLAINED

Known as pub/sub, Publish/Subscribe messaging is an asynchronous service-to-service communication method used in serverless and microservices architectures. Basically, the Pub/Sub model involves:

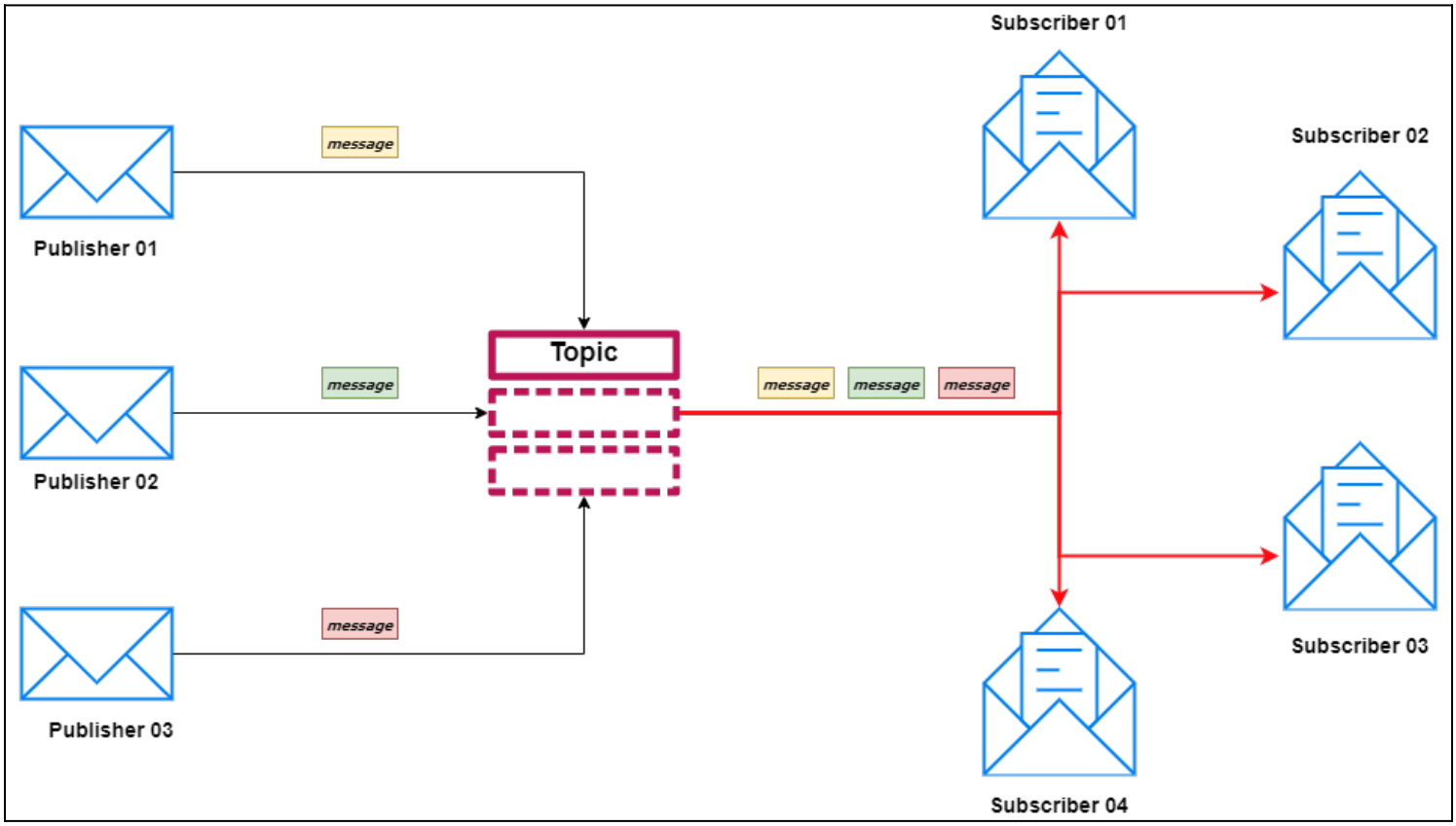
- A publisher who sends a message
- A subscriber who receives the message via a message broker

In this article, we'll see how pub/sub works, look at pros, cons, and use cases, and share a tutorial for setting up simple pub/sub messaging.

Basics of pub/sub messaging

With the popularity of decoupled and microservices-based applications, proper communication between components and services is crucial for overall application functionality. Pub/Sub messaging helps with this in two crucial ways:

- Allowing developers to create decoupled applications easily with a reliable communication method
- Enabling users to create event-driven architectures easily



A pub/sub model allows messages to be broadcasted asynchronously across multiple sections of the applications.

The core component that facilitates this functionality is something called a Topic. The publisher will push messages to a Topic, and the Topic will instantly push the message to all the subscribers. This is what differentiates the Pub/Sub model from traditional message brokers, where a message queue will batch individual messages until a user or service requests these messages and retrieves them.

Whatever the message is in the Pub/Sub model, it will be automatically pushed to all the subscribers. The only exception is user-created policies for subscribers that will filter out messages.

This approach makes it possible to create event-driven services without constantly querying a message queue for messages. It also enables developers to create different isolated functions using the same message (data) that can be executed parallelly with the ability to serve multiple subscribers.

The Pub/Sub pattern isolates publishers from subscribers so that publishers do not need to know where the message is being used while the subscriber does not need to know about the publisher. This helps to improve the overall security of the application organically.

Advantages of publish/subscribe pattern

A distributed microservices-based application developed using the pub/sub pattern benefits a whole organization, from software architects to QA engineers.

Here are the advantages of pub/sub:

Decoupled/loosely coupled components

Pub/Sub allows you to separate the communication and application logic easily, thereby creating isolated components. This results in:

- Creating more modularized, robust, and secure software components or modules
- Improving code quality and maintainability

Greater system-wide visibility

The simplicity of the pub/sub pattern means that users can understand the flow of the application easily.

The pattern also allows creating decoupled components that help us get a bird's eye view of the information flow. We can know exactly where information is coming from and where it is delivered without explicitly defining origins or destinations within the source code.

Real-time communication

Pub/sub delivers messages to subscribers instantaneously with push-based delivery, making it the ideal choice for near real-time communication requirements. This eliminates the need for any polling to check for messages in queues and reduces the delivery latency of the application.

Ease of development

Since pub/sub is not dependent on programming language, protocol, or a specific technology, any supported message broker can be easily integrated into it using any programming language. Additionally, Pub/Sub can be used as a bridge to enable communications between components built using different languages by managing inter-component communications.

This leads to easy integrations with external systems without having to create functionality to facilitate communications or worry about security implications. We can simply publish a message to a topic and let the external application subscribe to the topic, eliminating the need for direct interaction with the underlying application.

Increased scalability & reliability

This messaging pattern is considered elastic—we do not have to pre-define a set number of publishers or subscribers. They can be added to a required topic depending on the usage.

The separation between communication and logic also leads to easier troubleshooting as developers can focus on the specific component without worrying about it affecting the rest of the application.

Pub/sub also improves the scalability of an application by allowing to change message brokers architecture, filters, and users without affecting the underlying components. With pub/sub, a new messaging implementation is simply a matter of changing the topic if the message formats are compatible even with complex architectural changes.

Testability improvements

With the modularity of the overall application, tests can be targeted towards each module, creating a more streamlined testing pipeline. This drastically reduces the test case complexity by targeting tests for each component of the application.

The pub/sub pattern also helps to easily understand the origin and destination of the data and the information flow. It is particularly helpful in testing issues related to:

- Data corruption
- Formatting
- Security

Disadvantages of pub/sub pattern

Pub/Sub is a robust messaging service, yet it is not the best option for all requirements. Next, let's look briefly at some shortcomings of this pattern.

Unnecessary complexity in smaller systems

Pub/sub needs to be properly configured and maintained. Where scalability and a decoupled nature are not vital factors to your app, implementing Pub/Sub will be a waste of resources and lead to unnecessary complexity for smaller systems

Media streaming

Pub/sub is not suitable when dealing with media such as audio or video as they require smooth synchronous streaming between the host and the receiver. Because it does not support synchronous end-to-end communications, pub/sub messaging is ill-suited for:

- Video conferencing
- VOIP
- General media streaming applications

Use cases for publish/subscribe messaging

So, when is the right time to use pub/sub?

The Pub/Sub pattern can be used across different industries to facilitate real-time and distributed communications. For instance, automation is a key area that benefits from this pattern.

The following sections describe common use cases of Pub/Sub.

IoT (Internet of Things)

With smart devices, we need a reliable and efficient way to gather and distribute information. A control node or server can publish updates that will be automatically delivered to all the subscribed IoT devices.

End-user IoT devices can also act as publishers and publish notifications, sensor information, etc., to the cloud, which will then be notified to the user.

System monitoring & event notifications

Pub/sub allows users to create topics to gather system information and push them to visualization and notification frontends.

This is highly useful when dealing with large-scale deployments:

1. Messages can be categorized into different topics.
2. All servers or services can publish the data to these common topics without the need for separate notification pipelines.

We can extend this functionality further by subscribing maintenance or management functions to a topic. For example, if a server reports an error, it will trigger a function to automatically replace that server.

Database backup & replication

It's essential to make backups with multiple databases spread across different technologies and vendors. We can configure periodic backups or snapshots using cron jobs.

However, suppose that we need to move these backups to different regions or cloud storage. In that case, we can use Pub/Sub messaging to create a pipeline that will push a message informing of completed backup. Then, a subscribed function will use that message as the trigger to start the migration or copy process.

Log management

Pub/Sub can act as the go-between to aggregate and distribute logs. We can collect logs from multiple locations and push them to subscribed services like elastic search or simply store them across different designations.

Logs can be filtered by issues, audit trails, notification, background tasks, etc., and direct to different subscribers, enabling proper log management.

Pub/sub messaging services

There are multitudes of Pub/Sub messaging services, from dedicated message brokers to cloud offerings. Following is a list of some common Pub/Sub services.

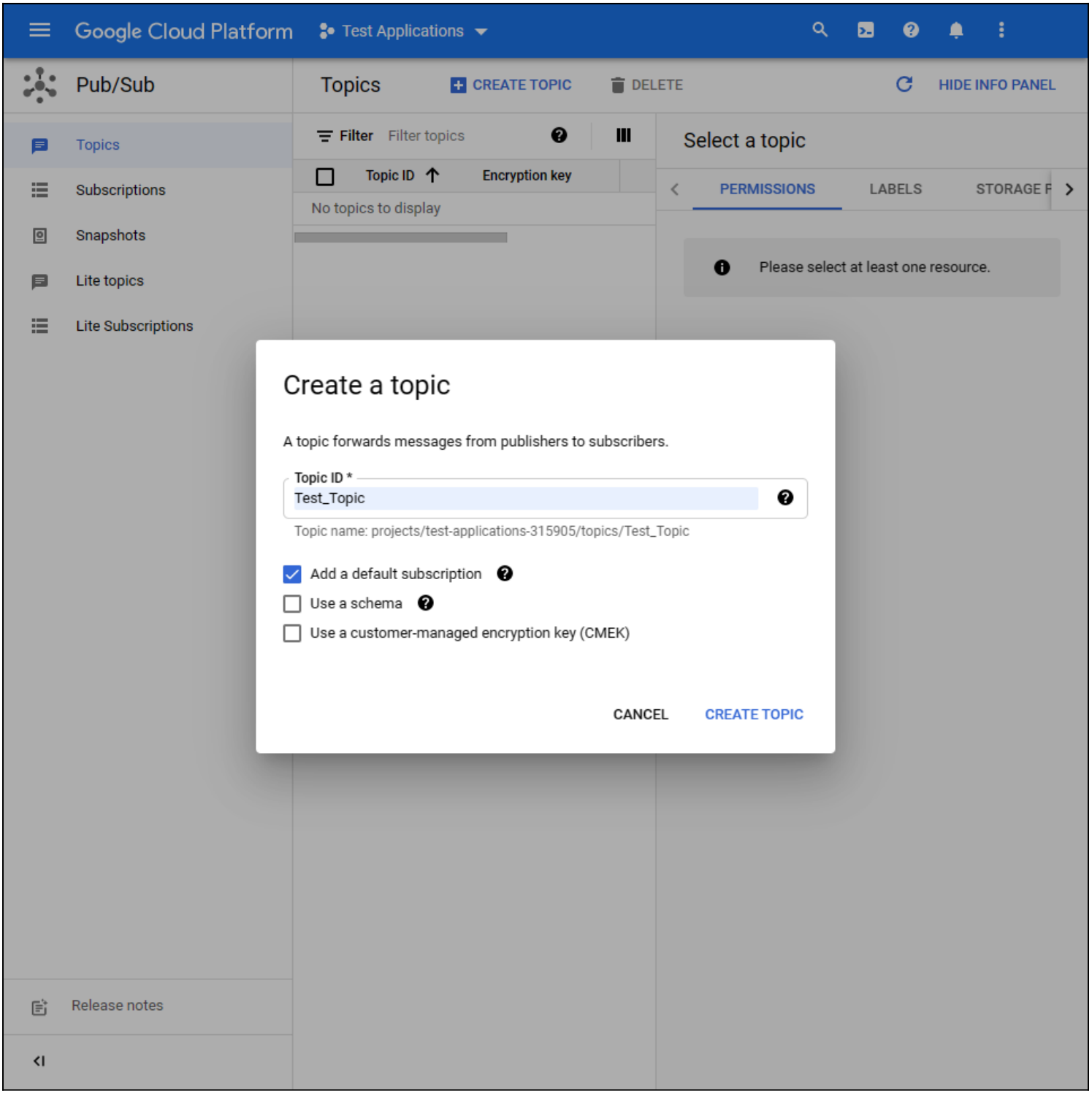
- **Apache Kafka**. Developed by Apache, Kafka has robust Pub/Sub messaging features with message logs.
- **Faye.** Simple Pub/Sub service designed to power web applications with servers designed for NodeJS and Ruby.
- **Redis.** This is one of the most popular message brokers with support for both traditional message queues as well as pub/sub pattern implementations.
- **Amazon SNS.** The Amazon Simple Notification Service is a fully managed service that offers Pub/Sub messages.
- **Google Pub/Sub.** GCP offering for pub/sub messaging service implementation.
- **Azure Service Bus.** A robust messaging service (MaaS) solution that offers Pub/Sub pattern.

Simple example: Publish/subscribe messaging

Since we now understand the Pub/Sub concepts, let's look at a simple workflow using Google Pub/Sub. It will publish a message to a topic and trigger a subscribed Google function to print the pushed message.

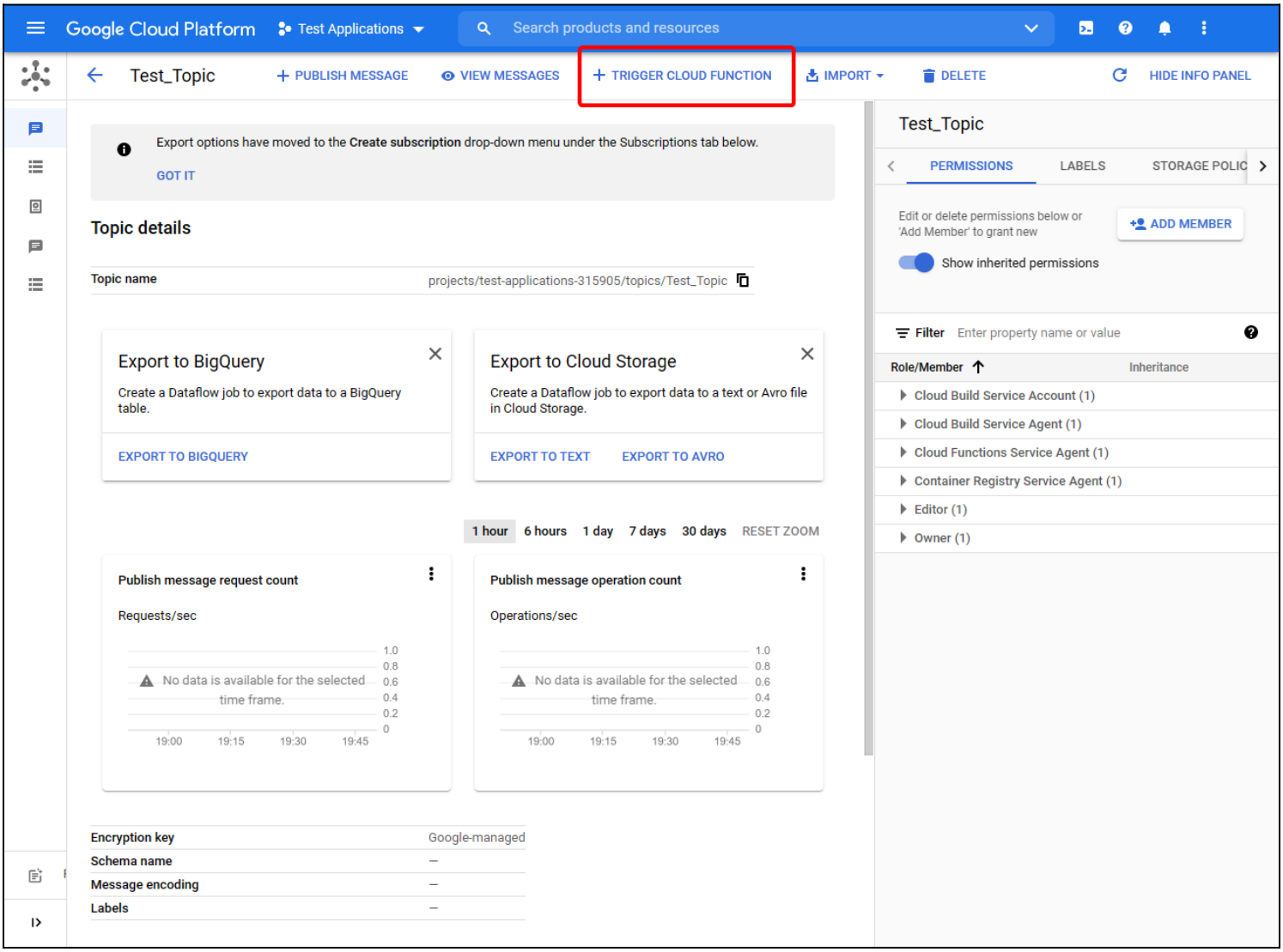
Step 1. Creating the topic

The first step is to create a Topic in Google Pub/Sub so that we can publish messages to that topic.



Step 2. Set up the trigger

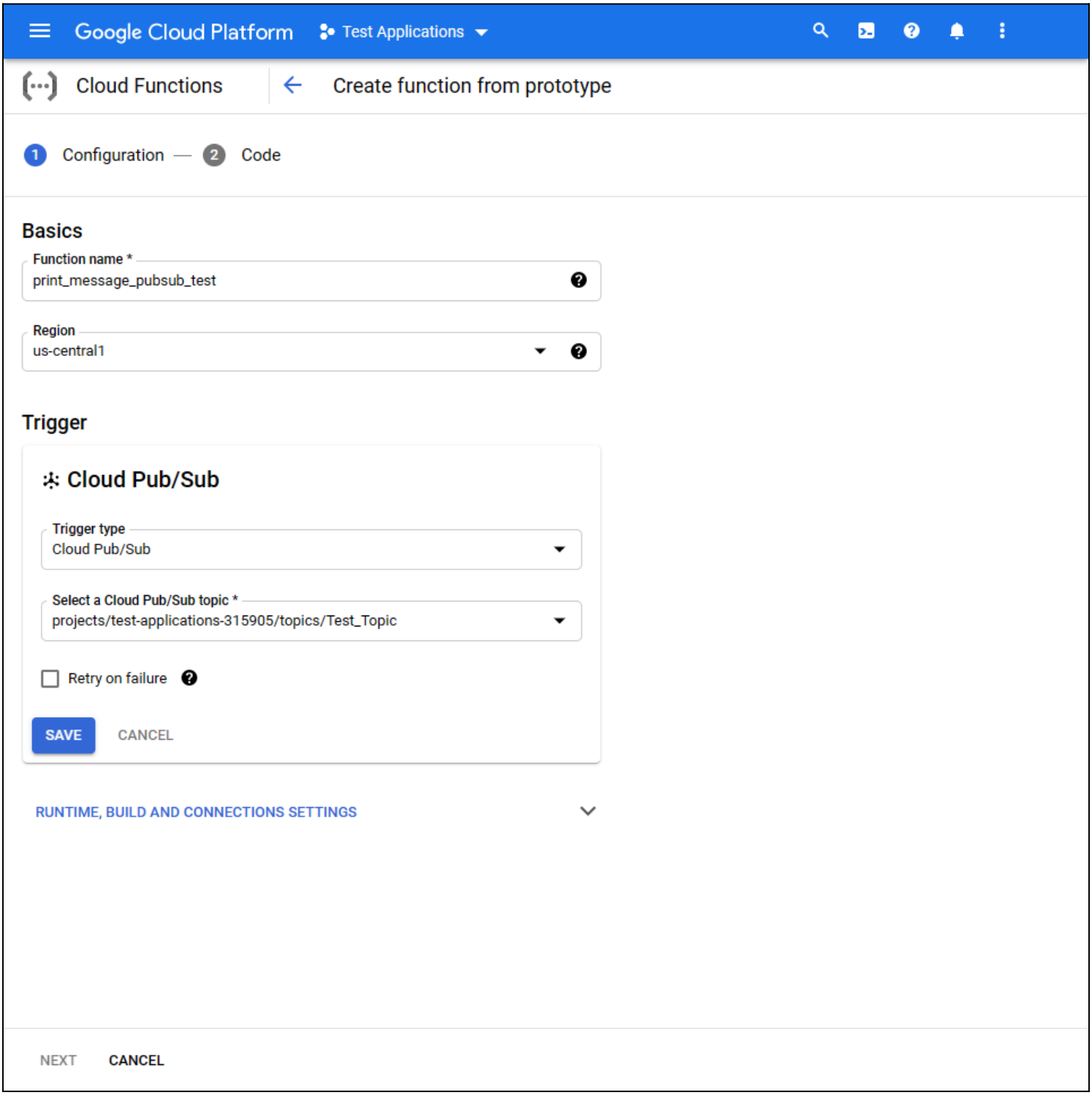
Navigate inside the created topic (Test_Topic) and click on the “Trigger Google Function” option. It will let you create a Google Function with the created topic as the trigger.



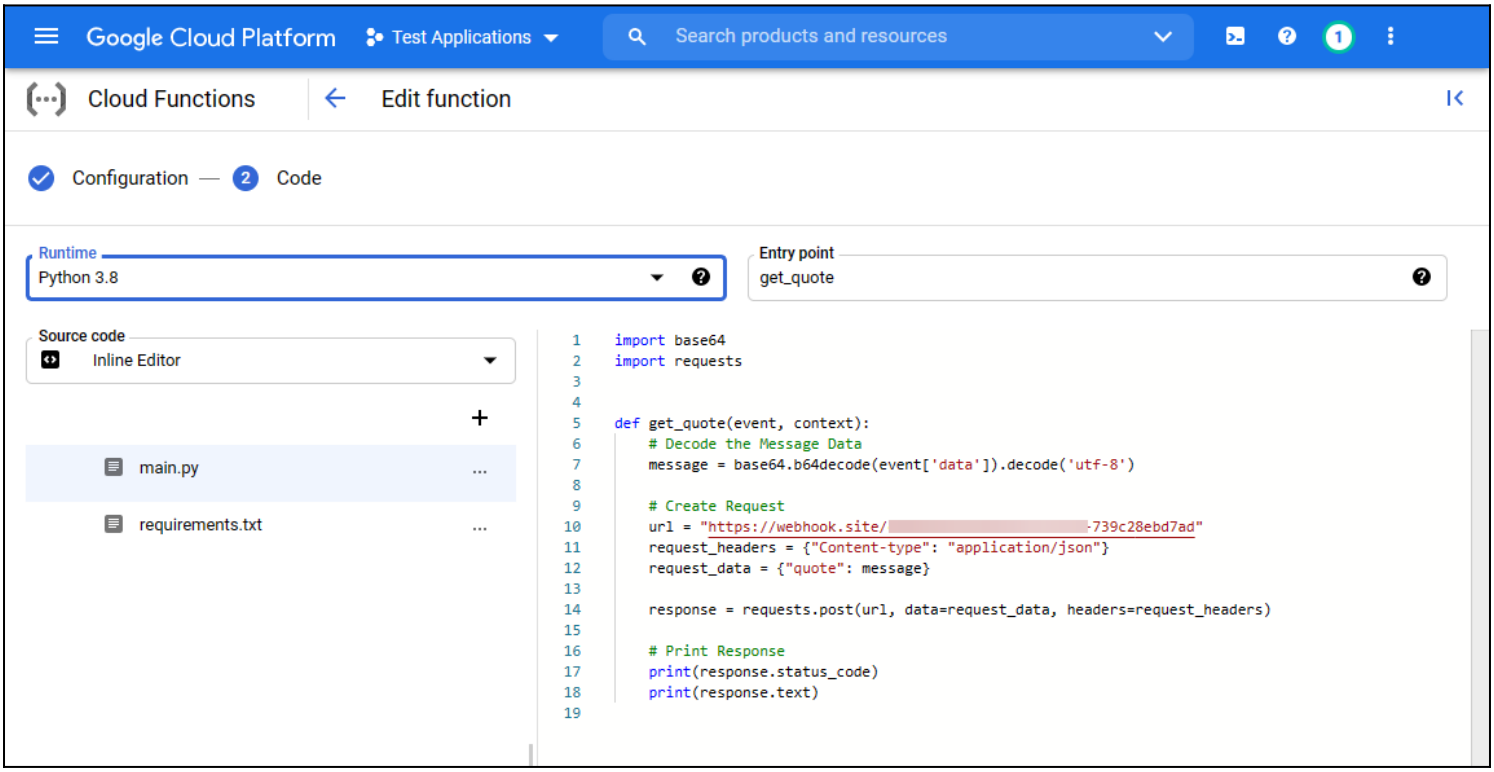
Step 3. Create the Google function (print_message_pubsub_test)

The first screen lets you name the Google function and set up the topic as the trigger. We will be using Python to create the function that simply captures the pushed data and send them to Webhook.site.

Also, we’ll be utilizing the requests library to create a POST request to send the data.

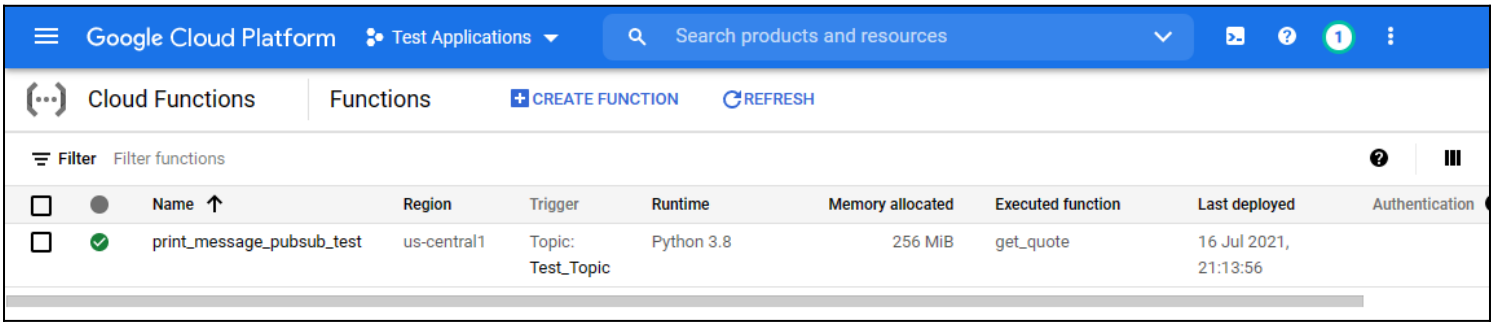


Cloud function code block:




```
import base64
import requests


def get_quote(event, context):
   # Decode the Message Data
   message = base64.b64decode(event).decode('utf-8')

   # Create Request
   url = "https://webhook.site/xxxxxxx-xxxx-xxxx-xxxx-739c28ebd7ad"
   request_headers = {"Content-type": "application/json"}
   request_data = {"quote": message}

   response = requests.post(url, data=request_data, headers=request_headers)

   # Print Response
   print(response.status_code)
   print(response.text)
```

Once the function is deployed successfully, you will notice that it indicates the Test_Topic as the trigger for the function.



Step 4. Set up the publisher

In this step, let's create a simple Python program to act as the publisher.

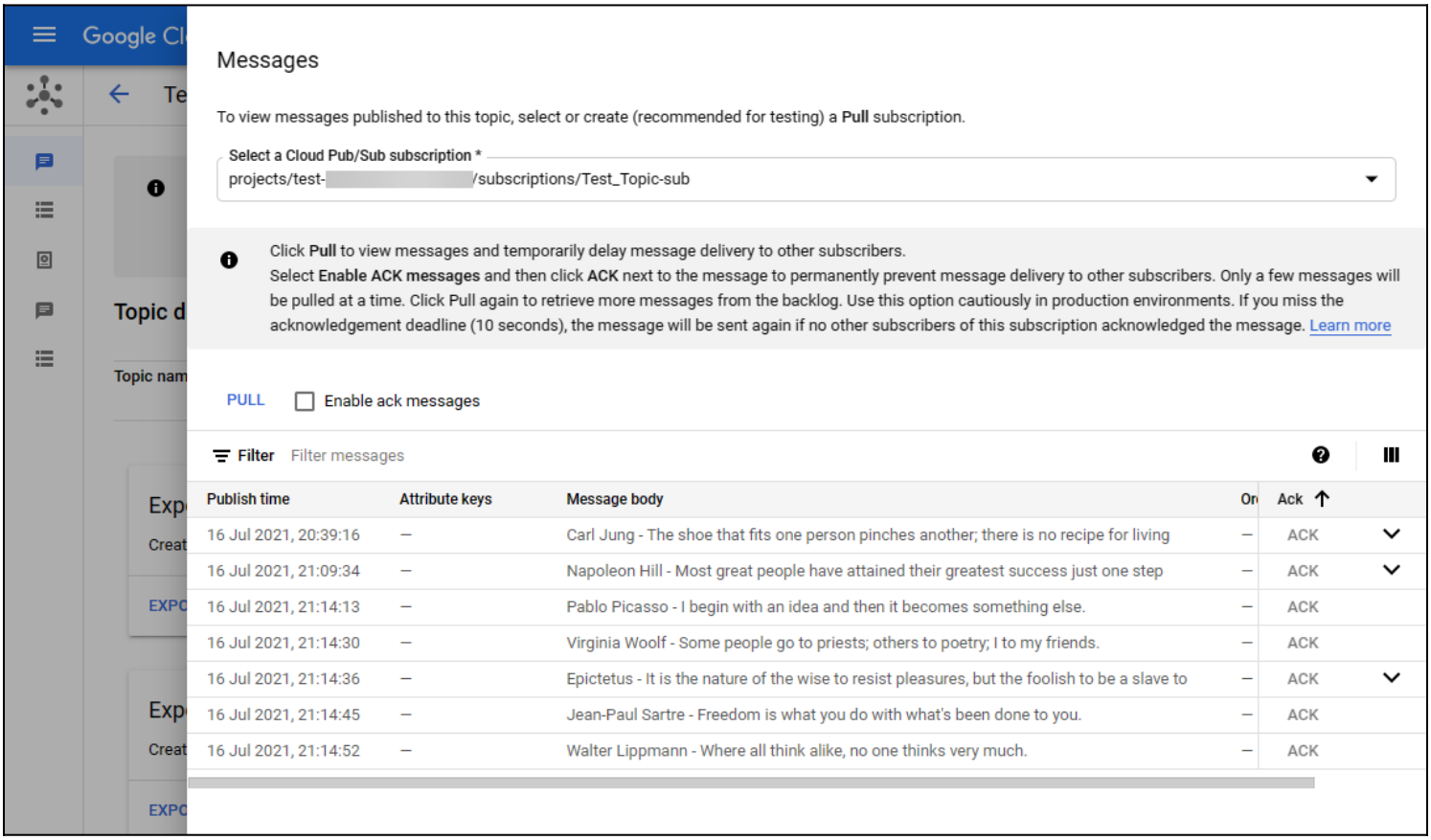
We will utilize the google cloud pubsub_v1 library to create a Publisher client and get a random inspirational quote from quotable.io. Then we will publish a concatenated string of the author and quote to the topic (Test_Topic)

message_publish.py

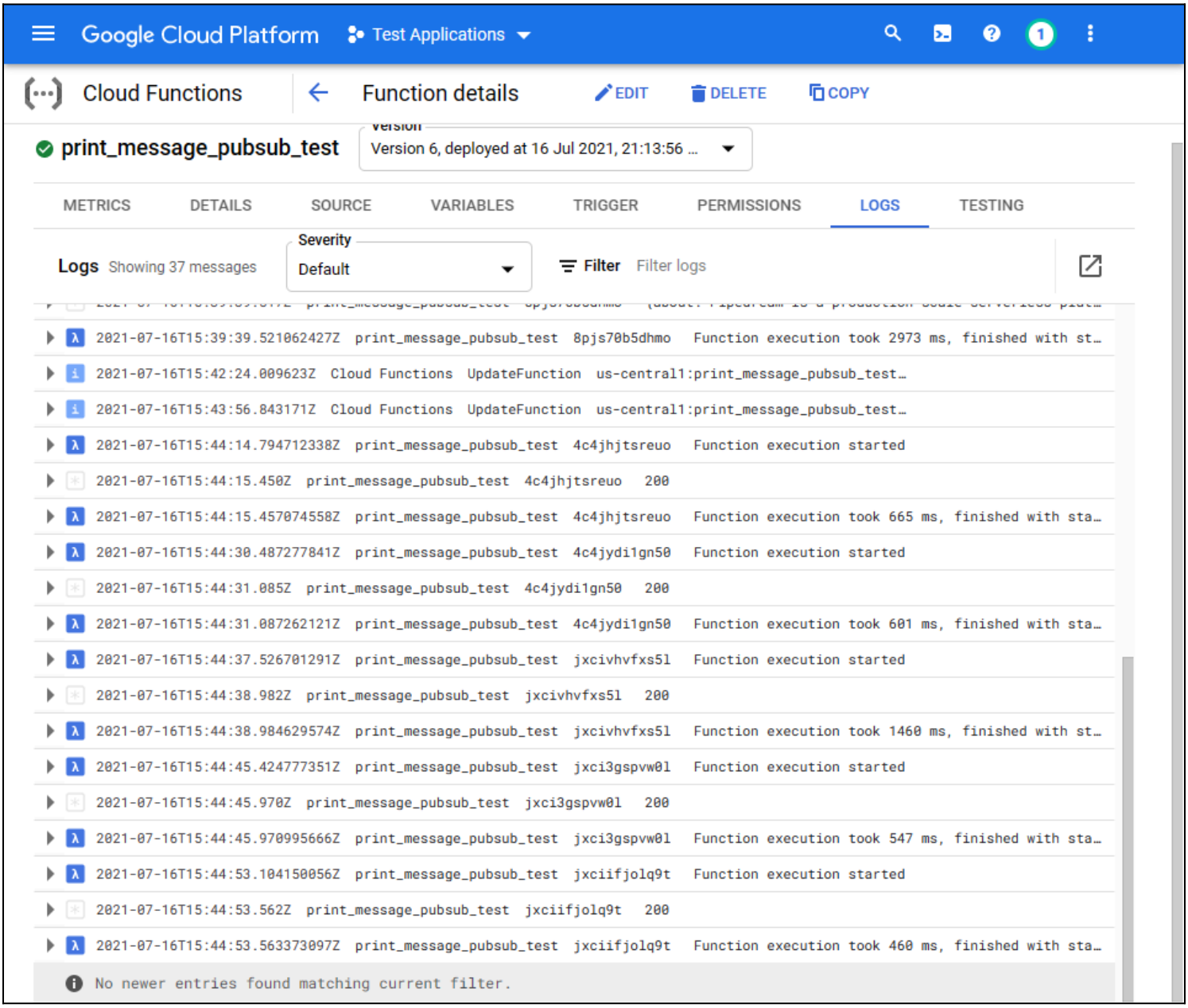
```
from google.oauth2 import service_account
from google.cloud import pubsub_v1
import requests

# Create Authentication Credentials
project_id = "test-applications-xxxxx"
topic_id = "Test_Topic"
gcp_credentials =
service_account.Credentials.from_service_account_file('test-applications-
xxxx-xxxxxxxxxx.json')

# Create Publisher Client
publisher = pubsub_v1.PublisherClient(credentials=gcp_credentials)
topic_path = publisher.topic_path(project_id, topic_id)

# Get a Random Quote
response = requests.get("https://api.quotable.io/random")
json_response = response.json()
message = f"{json_response} - {json_response}"

# Publish the Message
data = message.encode("utf-8")
future = publisher.publish(topic_path, data)

# Print Result
print(f"Published messages to {topic_path} - {future.result()}.")
```

That's it! We've successfully configured the messaging pipeline. When we run the "message_publish" script, it will publish the data to the Test_Topic and trigger the Google Cloud Function (print_message_pubsub_test), which will send the data to the Webhook site.

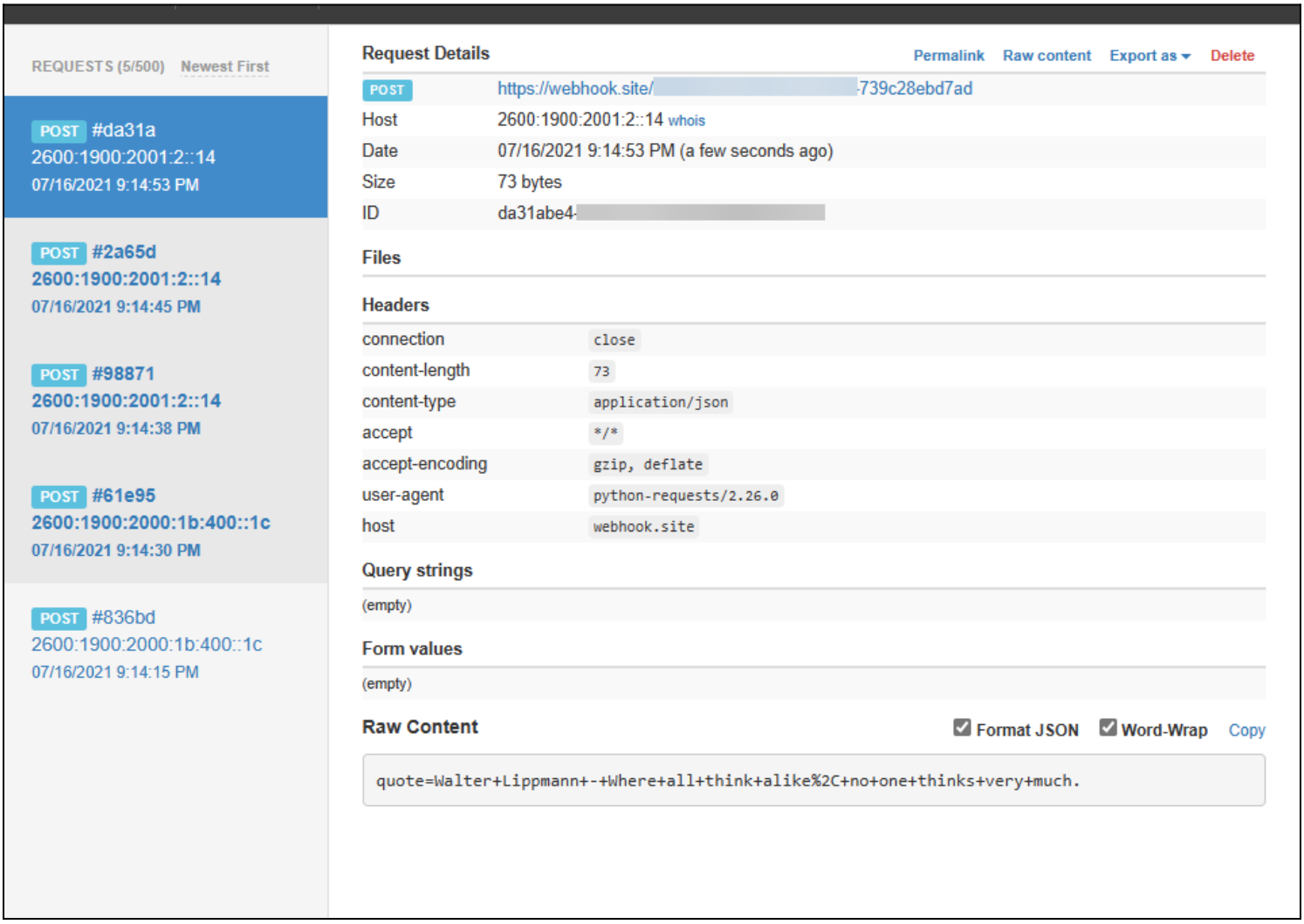
We can see the messages published to the topic within the Pub/Sub topic.



The logs of the Google cloud function will indicate that the function was triggered.



Finally, we can see all the messages that were received by the Webhook.site as shown below.



Above is the basic structure of any Pub/Sub workflow. We can use it as a simple template and extend it to facilitate any functionality.

Simple, powerful communication

The Pub/Sub messaging pattern is a powerful yet simple communication method. It acts as the cornerstone of powering real-time distributed microservices-based applications by handling all the communication between internal and external components.

Pub/Sub can be used to create asynchronous scalable message flows with minimal delivery delays due to all the benefits it offers over traditional message brokers.

Related reading

- BMC DevOps Blog
- 15 Best Practices for Building a Microservices Architecture
- Service-Oriented Architecture vs Microservices Architecture: Comparing SOA to MSA
- 3 Kubernetes Patterns for Cloud Native Applications
- Anti-Patterns vs Patterns: What is an Anti-Pattern?