

Local Emulation of BigQuery, Firestore, DataStore, BigTable and Pub/Sub

Table of Contents:

1. Introduction
2. Background
3. Advantages of local emulation
4. Tools and Technologies Used
5. Setting up Local Emulation
 - 5.1. Emulating BigQuery
 - 5.2. Emulating Pub/Sub
 - 5.3. Emulating Firestore
6. Conclusion

1. Introduction:

This documentation provides a comprehensive guide on emulating the behavior of Google Cloud Services, specifically BigQuery, Firestore, and Pub/Sub, locally for testing and development purposes. By mimicking the functionality of these services in a local environment, developers can streamline the development process, reduce costs associated with cloud usage during testing, and enhance overall development efficiency.

2. Background

- **BigQuery:**

- BigQuery is a fully managed, highly scalable serverless data warehouse that serves real-time analytics with streaming data ingestion offered by GCP(Google Cloud Platform)
- It's designed to handle large-scale data analytics workloads and enable users to run SQL-like queries against multi-terabyte datasets quickly

- **Pub/Sub:**

- Pub/Sub is a fully managed messaging service that Google Cloud Platform (GCP) provides.
- It enables asynchronous, reliable messaging between independent applications or microservices.
- Pub/Sub follows a publish-subscribe model where publishers send messages to topics, and subscribers receive messages from these topics.

- **Firestore:**

- Firestore is a flexible, scalable database service provided by Google Cloud Platform (GCP).
- It is a NoSQL document database designed to store and sync data for client- and server-side development.

- **DataStore:**

- Datastore is a highly scalable NoSQL database for your applications.
- Datastore automatically handles sharding and replication, providing you with a highly available and durable database that scales automatically to handle your applications' load.

- **Bigtable:**

- Bigtable is a NoSQL database service, specifically a key-value store that allows for very wide tables with tens of thousands of columns,

3. Advantages of Local Emulation:

- Cost Savings
- Offline Development
- Rapid Iteration
- Isolation and Predictability
- Customization and Configuration
- Security and Compliance
- Performance Testing
- Reduce Dependency on Cloud Resources

4. Tools and Technologies

- Prerequisites: Docker is the only prerequisite.
- Technologies: Python

5. Setting up local emulation

- **Emulating BigQuery**

Although Google provides emulators for many of its cloud services, it surprisingly doesn't for BigQuery. Instead, they offer a free tier option.

<https://cloud.google.com/bigquery/pricing#free-tier>

However, for developers wanting to test and develop offline, an open-source solution exists: the BigQuery emulator found at

<https://github.com/goccy/bigquery-emulator>.

The emulator can be installed in two different ways:

- locally as an application
- Docker container

A. Installation & run

Get the Docker container

```
docker pull ghcr.io/goccy/bigquery-emulator:latest
```

B. Start the container

```
docker run --platform=linux/x86_64 -it -p 9050:9050
ghcr.io/goccy/bigquery-emulator:latest --project=test-project
```

C. Using the emulator

We can use the big query emulator through Python Client or REST APIs

a. Through Python Client

1. Initializing the bigQuery Client

```
1  from google.api_core.client_options import ClientOptions
2  from google.auth.credentials import AnonymousCredentials
3  from google.cloud import bigquery
4
5  client_options = ClientOptions(api_endpoint="http://localhost:9050")
6  client = bigquery.Client(
7      project="test-project",
8      client_options=client_options,
9      credentials=AnonymousCredentials(),
10 )
```

client_options: Defining the client options with the specified API endpoint pointing to the local BigQuery emulator (<http://localhost:9050>).

client: Creating a BigQuery client instance with the specified project ID, custom client options pointing to the emulator, and anonymous credentials for local testing.

a. Querying the table

```

# Define SQL query
sql_query = """
WITH cte AS (
    SELECT d.dept_name AS Department,
           e.name AS Employee,
           e.salary,
           DENSE_RANK() OVER (PARTITION BY e.departmentId ORDER BY e.salary DESC) AS x_row
    FROM company.employee AS e
    JOIN company.department AS d ON e.departmentId = d.id
)
SELECT Department, Employee, Salary
FROM cte
WHERE x_row <= 3
"""

# Execute query
query_job = client.query(sql_query)

# Fetch results
results = query_job.result()

# Display results
for row in results:
    print(row)

```

Results :

```

PS C:\Users\palak> C:/Python311/python.exe c:/Users/palak/OneDrive/Desktop/GCP/BigQuery/rank.py
Row(('IT', 'Max', 90000), {'Department': 0, 'Employee': 1, 'Salary': 2})
Row(('IT', 'Joe', 85000), {'Department': 0, 'Employee': 1, 'Salary': 2})
Row(('IT', 'Randy', 85000), {'Department': 0, 'Employee': 1, 'Salary': 2})
Row(('IT', 'Will', 70000), {'Department': 0, 'Employee': 1, 'Salary': 2})
Row(('Sales', 'Henry', 80000), {'Department': 0, 'Employee': 1, 'Salary': 2})
Row(('Sales', 'Sam', 60000), {'Department': 0, 'Employee': 1, 'Salary': 2})
PS C:\Users\palak> |

```

E. Through REST APIs

We can also use BigQuery APIs here:

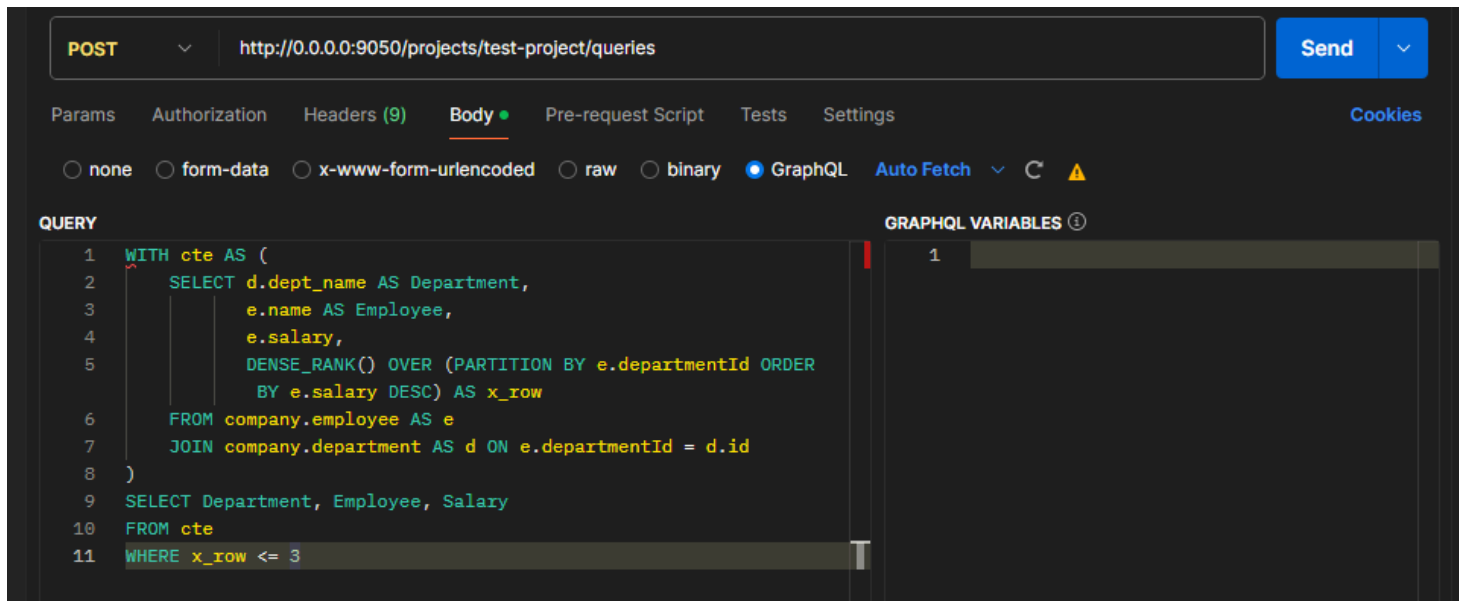
Here the service endpoint is the *api_endpoint(localhost:9050)*

Take reference: <https://cloud.google.com/bigquery/docs/reference/rest>

Eg: to execute SQL queries

`POST /bigquery/v2/projects/{projectId}/queries`

Runs a BigQuery SQL query synchronously and returns results if the query completes within a specified timeout.



Results

```
{  
  "jobReference": {  
    "jobId": "P9V36yVsKl4NhcgHKE4dxkTbk6B",  
    "projectId": "test-project",  
    "schema": {  
      "fields": [  
        {  
          "name": "Department",  
          "type": "STRING",  
          "mode": "REQUIRED",  
          "description": ""  
        },  
        {  
          "name": "Employee",  
          "type": "STRING",  
          "mode": "REQUIRED",  
          "description": ""  
        },  
        {  
          "name": "Salary",  
          "type": "INTEGER",  
          "mode": "REQUIRED",  
          "description": ""  
        }  
      ],  
      "rows": [  
        {  
          "f": {  
            "v": "IT",  
            "v": "Max",  
            "v": "90000"  
          },  
          "f": {  
            "v": "IT",  
            "v": "Joe",  
            "v": "85000"  
          },  
          "f": {  
            "v": "IT",  
            "v": "Randy",  
            "v": "85000"  
          },  
          "f": {  
            "v": "IT",  
            "v": "Will",  
            "v": "70000"  
          },  
          "f": {  
            "v": "Sales",  
            "v": "Henry",  
            "v": "80000"  
          },  
          "f": {  
            "v": "Sales",  
            "v": "Sam",  
            "v": "60000"  
          }  
        ],  
        "totalRows": "6",  
        "jobComplete": true  
      }  
    }  
  }  
}
```

- **Emulating Pub/Sub and Firestore:**

Firestore is used to develop rich applications using a fully managed, scalable, and serverless document database that effortlessly scales up or down to meet any demand, with no partitioning, maintenance windows, or downtime.

Steps to set up a firebase emulators, this will help us to use all three(Function, Firestore, Pub/Sub) emulators but with different ports:

A. Repo Structure:

- Add functions/index.js
- Add Dockerfile
- Add firebase.json
- Add .firebaserc
- Add docker-compose.yml

B. functions/index.js: Here is the simplest function possible that listens to HTTP calls and writes to firestore data received from the query parameter.

- a. *handleRequest*: this function will receive the data and then store that data in firestore which will appear on UI.

```

1  const { onRequest } = require("firebase-functions/v2/https");
2  const { initializeApp } = require("firebase-admin/app");
3  const { firestore } = require("firebase-admin");
4
5  initializeApp();
6
7  exports.handleRequest = onRequest(async (req, res) => {
8    console.log('[handleRequest] running inside the http method');
9
10   const collectionName = 'users';
11
12   const documentData = {
13     query: req.query.text
14   };
15
16   try {
17     const collectionRef = firestore().collection(collectionName);
18     await collectionRef.add(documentData);
19
20     res.json({ result: `Document added to Firestore` });
21   } catch (error) {
22     console.error('Error adding document: ', error);
23     res.status(500).json({ error: 'Internal Server Error' });
24   }
25 });

```

c. Add Dockerfile

```

Firebase > Dockerfile > RUN
1  FROM node:20-bullseye-slim
2
3  RUN apt update -y && apt install -y openjdk-11-jdk bash
4
5  RUN npm install -g firebase-tools
6
7  # Expose any necessary ports
8  EXPOSE 9005
9
10 COPY . .
11
12 RUN npm --prefix ./functions install
13
14 RUN echo '#!/bin/sh \n firebase emulators:start' > ./entrypoint.sh && \
15   chmod +x ./entrypoint.sh
16
17 ENTRYPOINT ["./entrypoint.sh"]

```

We are starting with a node image

`apt update -y && apt install -y openjdk-11-jdk bash` - installs jdk and bash (bash is for convenience to be able to run commands inside the container if you need).

`npm install -g firebase-tools` - installs firebase cli with which we can run all emulators.

`COPY . .` - copies the whole current directory content to the docker container, except `node_modules`, `dockerfile`, and `readme` which we excluded in `.dockerignore`.

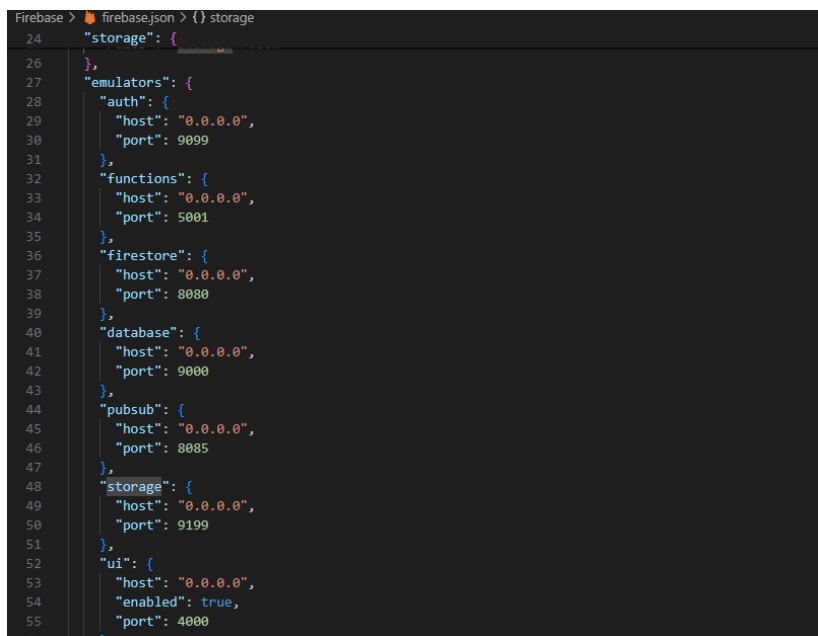
`RUN npm --prefix ./functions install` - builds functions folder with all dependencies that are needed for functions. For now only JS is supported in this dockerfile.

`RUN echo '#!/bin/sh \n firebase emulators:start' > ./entrypoint.sh && \ chmod +x ./entrypoint.sh`

The line above creates file “entrypoint.sh” with the following shell commands:

```
#!/bin/sh firebase emulators:start
```

D. Add firebase.json



```

24  "storage": {
25    "host": "0.0.0.0",
26    "port": 9000
27  },
28  "emulators": {
29    "auth": {
30      "host": "0.0.0.0",
31      "port": 9099
32    },
33    "functions": {
34      "host": "0.0.0.0",
35      "port": 5001
36    },
37    "firestore": {
38      "host": "0.0.0.0",
39      "port": 8080
40    },
41    "database": {
42      "host": "0.0.0.0",
43      "port": 9000
44    },
45    "pubsub": {
46      "host": "0.0.0.0",
47      "port": 8085
48    },
49    "storage": {
50      "host": "0.0.0.0",
51      "port": 9199
52    },
53    "ui": {
54      "host": "0.0.0.0",
55      "enabled": true,
56      "port": 4000
57    }
58  }
59 }
```

firebase.json is a firebase configuration file that will tell firebase-tools what to build. Here we have all running emulators specified to run on 0.0.0.0 (listen on all available network interfaces), and their ports. Besides that the important part of this config is functions. It will build functions code and make them available in the emulator. Without it, the functions emulator will show you that you don't have actions and will not start.

E. Add .firebaserc

```
{
  "projects": {
    "default": "react-my-burger-de634"
  }
}
```

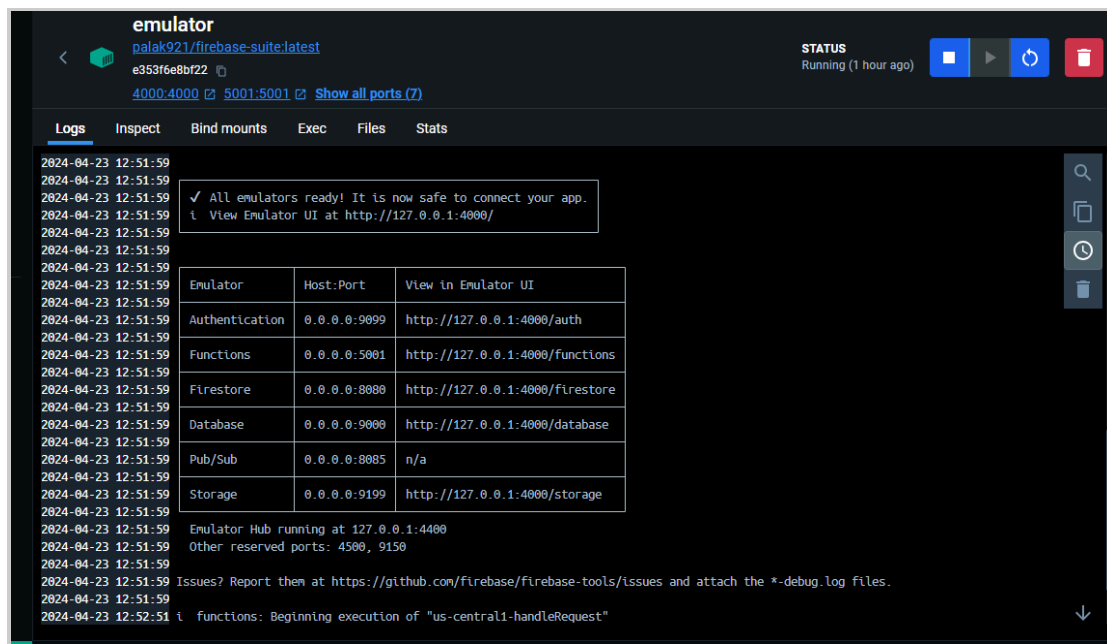
This is a Firebase configuration file, here we just specify our project ID.

F. Add docker-compose.yml

```
1  version: '4'
2
3  services:
4    firebase:
5      container_name: firebase-emulator
6      build:
7        context: .
8      ports:
9        - 8080:8080 # **FIRESTORE_PORT**
10       - 5005:5005 # **FIRESTORE_WS_PORT**
11       - 4000:4000 # **UI_PORT**
12       - 8085:8085 # **PUBSUB_PORT**
13       - 5001:5001 # **FUNCTIONS_PORT**
14       - 9099:9099 # **AUTH_PORT**
15       - 9000:9000 # **DATABASE_PORT**
16       - 9199:9199 # **STORAGE_PORT**
```

G. DEMO: To run the demo we need to run the command

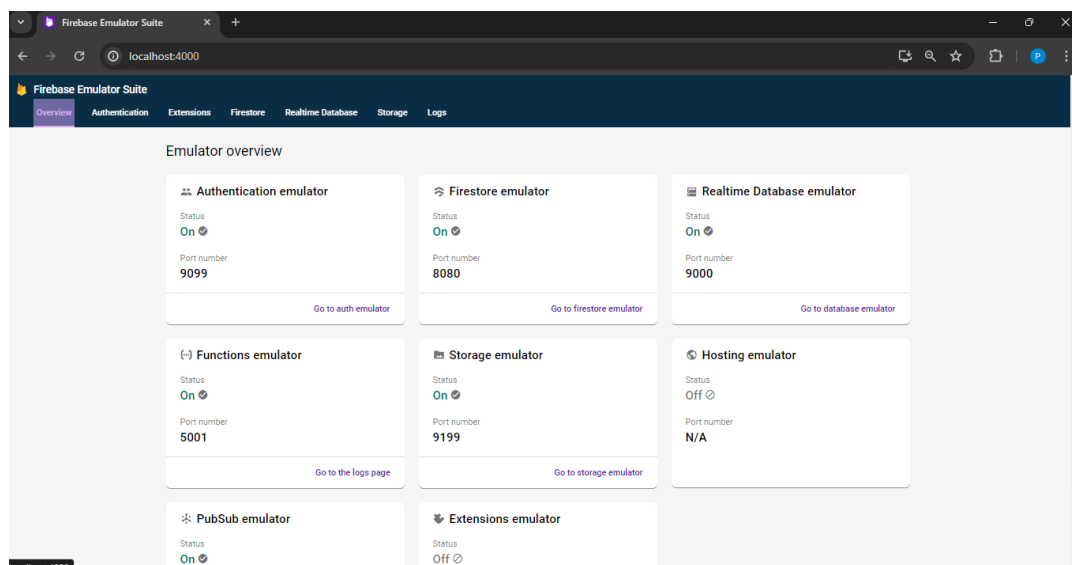
```
docker compose up -d
```



Now in the docker, we can see all the different emulators running on different port numbers.

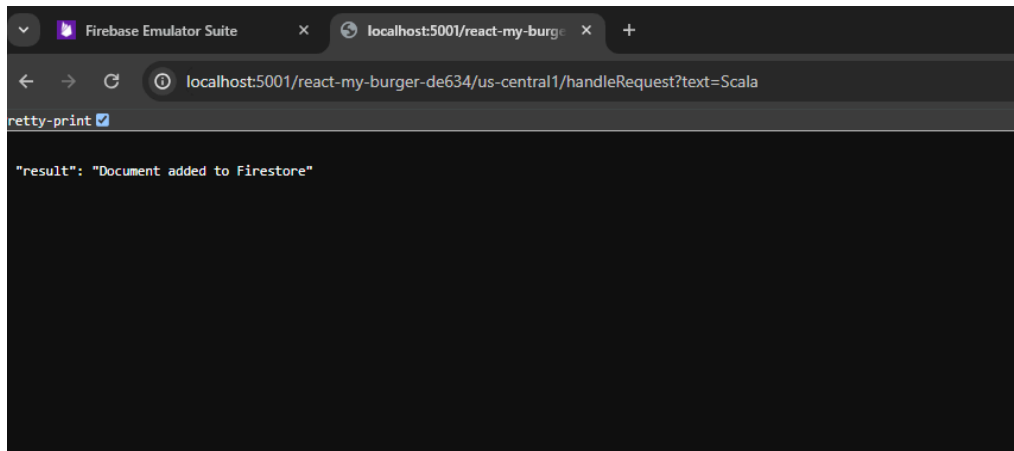
By navigating to <http://localhost:4000/>

We can see that this UI

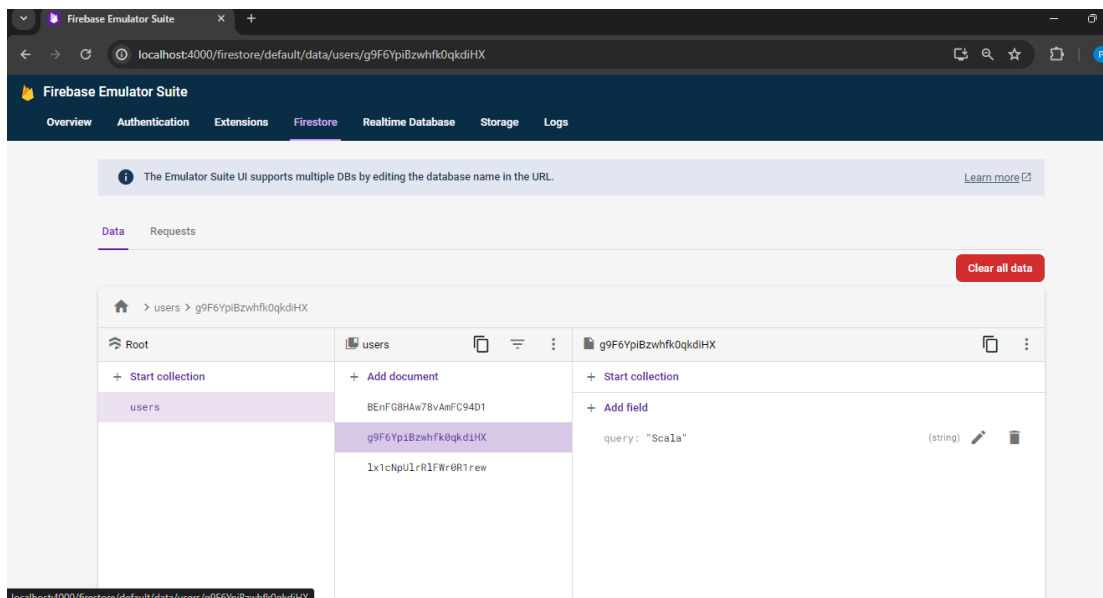


To test our function navigate to this link

<http://localhost:5001/react-my-burger-de634/us-central1/handleRequest?text=Scala>



In the firestore UI we can see the Document is added successfully.



- **Emulating Pub/Sub:**

In the UI at <http://localhost:4000> we can see the pub/sub emulator is running on port **8085**.

Testing Pub/Sub:

A. Creating topic:

```
PubSub > create-topic.py > ...
1 import os
2 from google.cloud import pubsub_v1
3
4 # Define the emulator host and port
5 emulator_host = "localhost:8085"
6 project_id = 'dummy-project' # project-id
7 topic_id = "hello"
8
9 # Set the environment variable for the Pub/Sub emulator host
10 os.environ["PUBSUB_EMULATOR_HOST"] = emulator_host
11 publisher = pubsub_v1.PublisherClient()
12 topic_path = publisher.topic_path(project_id, topic_id)
13
14 topic = publisher.create_topic(request={"name": topic_path})
15
16 print(f"Created topic: {topic.name}")
```

This code creates a new Pub/Sub topic with the specified *project ID* and *topic ID*. Here we have created a topic named “*hello*”.

To enable access to a port in this container from another container, utilize the system's IP address instead of localhost for connectivity.

Please consult the official documentation provided for guidance on publishing and subscribing messages.

```
PS C:\Users\palak> python C:\Users\palak\OneDrive\Desktop\GCP\PubSub\pub
.py
91
92
93
94
95
96
97
98
99
Published messages to projects/dummy-project/topics/hello.
PS C:\Users\palak> |

PS C:\Users\palak> & C:/Python311/python.exe c:/Users/palak/OneDrive/Des
ktop/GCP/PubSub/sub.py
Listening for messages for topic projects/dummy-project/topics/hello...
Received message: Message {
  data: b'{"message_number": 3, "content": "Message number 3..."
  ordering_key: ''
  attributes: {}
}
Received message: Message {
  data: b'{"message_number": 6, "content": "Message number 6..."
  ordering_key: ''
  attributes: {}
}
Received message: Message {
  data: b'{"message_number": 7, "content": "Message number 7..."
  ordering_key: ''
  attributes: {}
}

Windows PowerShell
data: b'{"message_number": 6, "content": "Message number 6..."
ordering_key: ''
attributes: {}
}
Received message: Message {
  data: b'{"message_number": 8, "content": "Message number 8..."
  ordering_key: ''
  attributes: {}
}
Received message: Message {
  data: b'{"message_number": 9, "content": "Message number 9..."
  ordering_key: ''
  attributes: {}
}
```

This image shows that the publisher is publishing msgs and two subscribers “hello-sub” and “hello-subscription” both receive msgs.

● Emulating DataStore

To use the Cloud Datastore Emulator locally using Docker, you need to follow these steps:

1. **Docker must be installed**
2. **Pull the Cloud SDK Docker image:** Google provides an official Cloud SDK Docker image, which includes the Datastore emulator.

```
docker pull google/cloud-sdk:latest
```

3. **Run the Cloud Datastore Emulator:** Start a Docker container with the Cloud Datastore Emulator using the following command:

```
docker run --rm -p 8081:8081 google/cloud-sdk:latest gcloud beta  
emulators datastore start --project=my-project-id --host-port=0.0.0.0:8081  
Now, you can use the DataStore Emulator:
```

```
4 # Set environment variable for the Datastore emulator host
5 os.environ["DATASTORE_EMULATOR_HOST"] = "localhost:8081"
6
7 # Connect to the Datastore emulator
8 datastore_client = datastore.Client(project="my-project-id", namespace="my-namespace")
9 # Create an entity
10 def create_entity():
11     # Define entity
12     entity_key = datastore_client.key("Person", "person1")
13     entity = datastore.Entity(key=entity_key)
14     entity.update({
15         "name": "John Doe",
16         "age": 30
17     })
18
19     # Save entity
20     datastore_client.put(entity)
21     print("Entity created:", entity.key)
22
23 # Query entities
24 def query_entities():
25     query = datastore_client.query(kind="Person")
26     entities = list(query.fetch())
27     print("Query result:")
28     for entity in entities:
29         print(entity.key.name, entity["name"], entity["age"])
30
31 # Retrieve an entity by key
32 def get_entity():
33     entity_key = datastore_client.key("Person", "person1")
34     entity = datastore_client.get(entity_key)
35     if entity:
36         print("Retrieved entity:", entity.key.name, entity["name"], entity["age"])
37     else:
```

The output will, be

```
Entity created: ('Person', 'person1')
Query result:
person1 John Doe 30
Retrieved entity: person1 John Doe 30
```

- **Emulating BigTable:**

- To use the Cloud Datastore Emulator locally using Docker, you need to follow these steps:

4. **Docker must be installed**

5. **Pull the Cloud SDK Docker image:** Google provides an official Cloud SDK Docker image, which includes the Datastore emulator.

```
docker pull google/cloud-sdk:latest
```

6. **Run the Cloud Datastore Emulator:** Start a Docker container with the Cloud Datastore Emulator using the following command:

```
docker run --rm -p 8081:8081 google/cloud-sdk:latest gcloud beta  
emulators datastore start --project=my-project-id --host-port=0.0.0.0:8081
```

```

ExBigtable.py > ...
1  import os
2  from google.cloud import bigtable
3
4  # Set environment variable for the Bigtable emulator host
5  os.environ["BIGTABLE_EMULATOR_HOST"] = "localhost:8086"
6
7  # Connect to the Bigtable emulator
8  client = bigtable.Client(project="my-project", admin=True)
9
10 # Create a table
11 def create_table():
12     instance = client.instance("my-instance")
13     table = instance.table("my-table")
14     column_family_id = "cf1"
15
16     if not table.exists():
17         print("Creating table...")
18         table.create()
19         print("Table created.")
20
21     # Create a column family
22     print("Creating column family...")
23     column_family_obj = table.column_family(column_family_id)
24     column_family_obj.create()
25     print("Column family created.")

```

- **Conclusion**

Local emulation of BigQuery, Firestore, Pub/Sub, and DataStore enhances development efficiency by enabling offline iteration and rapid testing, eliminating internet connectivity, and reducing costs associated with cloud resources. This approach accelerates the development lifecycle and fosters a controlled environment for debugging and troubleshooting, ultimately improving developer productivity. Additionally, emulation facilitates seamless collaboration among team members, as emulated environments can be easily shared and replicated across different development machines. Overall, local emulation empowers developers to work offline, iterate quickly, and deliver high-quality applications efficiently and cost-effectively.