03 GCP Connections -KirkYagami 🧑‍💻 🕵️

1. GCP GCS Connections

Overview

Google Cloud Storage (GCS) is a scalable, secure, and highly available object storage service on Google Cloud Platform (GCP). Establishing connections to GCS is essential for various operations like reading and writing data, managing buckets, and integrating with other GCP services. Understanding how to connect to GCS using different tools, SDKs, and APIs is crucial for managing your data effectively.

Real-World Scenario 1: Connecting to GCS from a Local Machine

Imagine you are working on a data analytics project where you need to process and analyze large datasets stored in GCS from your local development environment. You can use the GCP SDK or APIs to establish a connection to GCS and perform operations like uploading and downloading files.

Step 1: Setting Up the GCP SDK

First, install the Google Cloud SDK on your local machine:

```
# On Linux/MacOS
curl https://sdk.cloud.google.com | bash

# On Windows (using PowerShell)
Invoke-WebRequest -Uri https://sdk.cloud.google.com | Out-File -FilePath
install.ps1; powershell.exe -File install.ps1

OR

Download the exe by visting the link:
`https://cloud.google.com/sdk/docs/install#windows`
```

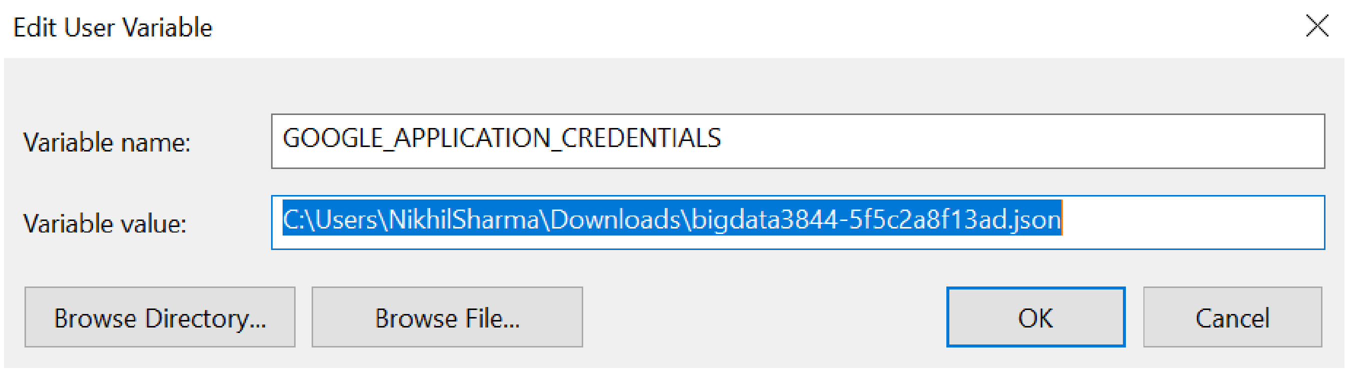
Step 2: Authenticate the SDK

Authenticate the SDK with your Google account:

```
gcloud auth login
```

You must have a service account with appropriate privileges and create a user variable as follows:

- Download the JSON key file


Edit User Variable

Variable name: GOOGLE_APPLICATION_CREDENTIALS

Variable value: C:\Users\NikhilSharma\Downloads\bigdata3844-5f5c2a8f13ad.json

Step 3: Connecting to GCS Using Python

You can use the `google-cloud-storage` Python library to interact with GCS. Install it using pip:

```
pip install google-cloud-storage
```

To list all the buckets in your GCS project:

```
from google.cloud import storage

def list_buckets():
    storage_client = storage.Client()
    buckets = list(storage_client.list_buckets())
    print("Buckets:")
    for bucket in buckets:
        print(bucket.name)

if __name__ == "__main__":
    list_buckets()
```

Step 4: Uploading a File to GCS

Here's how to upload a file to a GCS bucket:

```
def upload_blob(bucket_name, source_file_name, destination_blob_name):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(destination_blob_name)

    blob.upload_from_filename(source_file_name)

    print(f"{source_file_name} uploaded as {destination_blob_name} in bucket
{bucket_name}  ")
```

```
if __name__ == "__main__":
        bucket_name = "customer_data_cf"
        source_file_name = "disney_plus_shows.csv"
        destination_blob_name = "disney_plus_shows.csv"

        upload_blob(bucket_name, source_file_name, destination_blob_name)
```

Step 5: Downloading a File from GCS

To download a file from a GCS bucket:
Sample code:

```
def download_blob(bucket_name, source_blob_name, destination_file_name):
    """Downloads a blob from the bucket."""
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(source_blob_name)
    blob.download_to_filename(destination_file_name)

    print(f"Blob {source_blob_name} downloaded to {destination_file_name}.")

if __name__ == "__main__":
    download_blob("my-bucket", "source-path/file.txt",
"/local/path/file.txt")
```

If creating a user variable for JSON key is not working as expected:

```
from google.cloud import storage
from google.oauth2 import service_account

def upload_csv_to_gcs(bucket_name, source_file_name, destination_blob_name,
key_path):
    try:
        # Load the credentials directly from the JSON key file
        credentials =
service_account.Credentials.from_service_account_file(key_path)

        # Initialize the storage client with the specified credentials
        storage_client = storage.Client(credentials=credentials)

        # Get the bucket and upload the file
        bucket = storage_client.bucket(bucket_name)
        blob = bucket.blob(destination_blob_name)
        blob.upload_from_filename(source_file_name)
        print(f"File {source_file_name} uploaded to
{bucket_name}/{destination_blob_name}.")

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    # Directly specify the path to the JSON key file
    key_path = r"C:\Users\NikhilSharma\Downloads\bigdata3844-
5f5c2a8f13ad.json"
    upload_csv_to_gcs("customer_data_cf", "transaction_logs.json",
"transaction_logs.json", key_path)
```

Real-World Scenario 2: Connecting to GCS from a Compute Engine Instance

You have a machine learning model deployed on a GCP Compute Engine instance. The model needs to load data from GCS for inference. Setting up a connection from the instance to GCS is essential.

Step 1: Assigning Permissions

Ensure that the Compute Engine instance has the necessary IAM role to access GCS. This is usually the "Storage Object Viewer" role.

Step 2: Using the `google-cloud-storage` Library on Compute Engine

Similar to the local machine example, you can use the `google-cloud-storage` Python library. The following code downloads a file from GCS to the Compute Engine instance:

```
def download_blob(bucket_name, source_blob_name, destination_file_name):
    """Downloads a blob from the bucket."""
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(source_blob_name)
    blob.download_to_filename(destination_file_name)

    print(f"Blob {source_blob_name} downloaded to {destination_file_name}.")

if __name__ == "__main__":
    download_blob("my-bucket", "source-path/file.txt",
"/local/path/file.txt")
```

2. Manage Connections

Overview

Managing connections in GCP involves setting up, configuring, and securing connections to various services. It is essential to understand how to manage connections to GCS, databases, and other GCP services to ensure seamless integration and data flow.

Real-World Scenario 1: Managing Service Account Keys for GCS Access

In an enterprise environment, you may need to manage multiple service accounts for different applications accessing GCS. Ensuring that service accounts have appropriate permissions and managing their keys is crucial for security.

Step 1: Creating a Service Account

Create a service account using the GCP Console or `gcloud` command:

```
gcloud iam service-accounts create my-service-account \
    --description="Service account for accessing GCS" \
    --display-name="My Service Account"
```

Step 2: Assigning IAM Roles

Assign the necessary IAM roles to the service account to allow both uploading and downloading files:

```
gcloud projects add-iam-policy-binding bigdata3844 \
    --member="serviceAccount:my-service-
account@bigdata3844.iam.gserviceaccount.com" \
    --role="roles/storage.objectViewer" \
    --role="roles/storage.objectCreator"
```

Step 3: Creating and Managing Service Account Keys

Generate a key for the service account and store it securely:

```
gcloud iam service-accounts keys create ~/key.json \
    --iam-account=my-service-account@bigdata3844.iam.gserviceaccount.com
```

Step 4: Authenticating Using the Service Account Key in Code

You can authenticate your application using the service account key:

```
import os
from google.cloud import storage

# Explicitly specify the path to the JSON key file
key_path = "/path/to/key.json"
```

```
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = key_path

def list_buckets():
    storage_client = storage.Client()
    buckets = list(storage_client.list_buckets())
    for bucket in buckets:
        print(bucket.name)

if __name__ == "__main__":
    list_buckets()
```

Step 5: Using the Service Account in Different Environments

Instead of setting environment variables, you can load the service account key directly in your code:

```
from google.oauth2 import service_account
from google.cloud import storage

def upload_blob(bucket_name, source_file_name, destination_blob_name,
key_path):
    """Uploads a file to the bucket using service account key."""
    credentials =
service_account.Credentials.from_service_account_file(key_path)
    storage_client = storage.Client(credentials=credentials)

    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(destination_blob_name)
    blob.upload_from_filename(source_file_name)

    print(f"File {source_file_name} uploaded to {destination_blob_name}.")

if __name__ == "__main__":
    upload_blob("my-bucket", "local-file.txt", "destination-path/file.txt",
"/path/to/key.json")
```

Real-World Scenario 2: Managing Connections to BigQuery

Suppose you're developing an application that needs to query large datasets in BigQuery. Efficiently managing connections to BigQuery is crucial to optimize performance and minimize costs.

Step 1: Connecting to BigQuery Using Python

You can connect to BigQuery using the `google-cloud-bigquery` library. Install it using pip:

```
pip install google-cloud-bigquery
```

To create a service account using the Google Cloud CLI (`gcloud`), follow these steps:

Step 1: Create the Service Account

Run the following command to create a new service account:

```
gcloud iam service-accounts create bigquery-access-sa \
    --description="Service account for accessing BigQuery" \
    --display-name="BigQuery Access Service Account"
```

This command creates a service account named `bigquery-access-sa` with the specified description and display name.

Step 2: Assign IAM Roles

Next, assign the necessary IAM roles to the service account. For BigQuery access, you'll typically need the `BigQuery Data Viewer` role:

```
gcloud projects add-iam-policy-binding bigdata3844 \
    --member="serviceAccount:bigquery-access-
sa@bigdata3844.iam.gserviceaccount.com" \
    --role="roles/bigquery.dataViewer"
```

This command binds the `BigQuery Data Viewer` role to the `bigquery-access-sa` service account within your project (`bigdata3844`).

Step 3: Create and Download the Service Account Key

Now, create a JSON key for the service account and download it to your local machine:

```
gcloud iam service-accounts keys create ~/bigquery-key.json \
    --iam-account=bigquery-access-sa@bigdata3844.iam.gserviceaccount.com
```

This command generates a new private key for the service account and saves it as `bigquery-key.json` in your home directory.

Step 4: Use the Service Account in Your Python Script

Update your Python script to use this service account key:

```
from google.cloud import bigquery
from google.oauth2 import service_account
```

```
def query_bigquery(query, key_path):
    """Runs a query on BigQuery using the specified service account key."""
    credentials =
service_account.Credentials.from_service_account_file(key_path)
    client = bigquery.Client(credentials=credentials)

    query_job = client.query(query)  # Make an API request.
    results = query_job.result()  # Wait for the job to complete.

    for row in results:
        print(row)

if __name__ == "__main__":
    query = """
        SELECT name, COUNT(*) as num_times
        FROM `bigquery-public-data.usa_names.usa_1910_current`
        WHERE state = 'TX'
        GROUP BY name
        ORDER BY num_times DESC
        LIMIT 10
    """
    query_bigquery(query, "/path/to/bigquery-key.json")  # Update with the
correct path to your key file.
```

Step 5: Execute the Script

Ensure that the path to the service account key is correct in your script, then run the script to execute the BigQuery query.
By following these steps, you can securely create and use a service account to access BigQuery from your Python application.

Step 3: Managing Query Costs and Performance

When working with BigQuery, it's important to manage query costs by optimizing your SQL queries, reducing the amount of data processed, and using partitioned tables. Additionally, you can manage the connection by batching queries, handling rate limits, and using caching where appropriate.

Conclusion

Understanding how to establish and manage connections to GCP services like GCS and BigQuery is essential for building robust and scalable cloud-based applications. The scenarios and code examples provided demonstrate how to connect to GCS and BigQuery from various environments and how to manage connections effectively, ensuring security, efficiency, and scalability.

Managing Connections to Cloud SQL MySQL

Overview

Managing connections to a Cloud SQL MySQL instance is crucial for applications running on Google Cloud Platform (GCP). Properly configuring and managing these connections ensures efficient database operations, security, and optimal performance. This section covers the essential steps and best practices for establishing and managing connections to a Cloud SQL MySQL instance.

Real-World Scenario: Managing Connections in a Web Application

Suppose you are developing a web application hosted on Google App Engine or Compute Engine, and the application requires persistent data storage using Cloud SQL MySQL. Managing the connections between the application and the database is vital to ensure that the application performs well under varying loads.

Step 1: Setting Up a Cloud SQL MySQL Instance

Before managing connections, you need to have a Cloud SQL MySQL instance set up. You can create one using the GCP Console or the `gcloud` command:

```
gcloud sql instances create my-sql-instance \
    --database-version=MYSQL_8_0 \
    --cpu=2 --memory=4GB \
    --region=us-central1
```

Step 2: Configuring the MySQL Instance for Connectivity

Ensure that your MySQL instance is configured to accept connections. You can allow connections from specific IP addresses or VPC networks:

```
gcloud sql instances patch my-sql-instance \
    --authorized-networks="203.0.113.0/24"
```

Step 3: Managing Connections Using the Cloud SQL Auth Proxy

To securely manage connections to your Cloud SQL MySQL instance, you can use the Cloud SQL Auth Proxy. This proxy provides secure access to the database without exposing your credentials or IP addresses.

- **Install the Cloud SQL Auth Proxy:**

```
wget https://dl.google.com/cloudsql/cloud_sql_proxy.linux.amd64 -O
cloud_sql_proxy
chmod +x cloud_sql_proxy
```

- **Run the Auth Proxy:**

```
./cloud_sql_proxy -instances=my-project:us-central1:my-sql-
instance=tcp:3306
```

Step 4: Connecting to Cloud SQL MySQL in Python

You can use the `pymysql` library to manage connections in your Python application:

```
pip install pymysql
```

Here's an example code to connect to your Cloud SQL MySQL instance through the Cloud SQL Auth Proxy:

```python
import pymysql

def connect_to_mysql():
    connection = pymysql.connect(
        host='127.0.0.1',
        user='your-username',
        password='your-password',
        database='your-database',
        port=3306
    )

    try:
        with connection.cursor() as cursor:
            cursor.execute("SELECT DATABASE();")
            result = cursor.fetchone()
            print("Connected to:", result)
    finally:
        connection.close()

if __name__ == "__main__":
    connect_to_mysql()
```

Step 5: Managing Connection Pools

For web applications with high traffic, managing connection pools is essential to handle multiple database connections efficiently. You can use SQLAlchemy with the `pymysql` dialect to manage connection pools in Python:

```
pip install sqlalchemy pymysql
```

Here's how you can configure and manage a connection pool:

```
from sqlalchemy import create_engine

# Create an engine with a connection pool
engine = create_engine(
    "mysql+pymysql://user:password@127.0.0.1:3306/dbname",
    pool_size=10,
    max_overflow=20
)

# Example query using the connection pool
with engine.connect() as connection:
    result = connection.execute("SELECT * FROM your_table")
    for row in result:
        print(row)
```

Step 6: Configuring Connection Limits

In your Cloud SQL MySQL instance, you can configure connection limits to ensure that your instance handles the expected load:

```
gcloud sql instances patch my-sql-instance --database-
flags=max_connections=500
```

This command sets the maximum number of connections to 500.

Step 7: Monitoring and Managing Connections

Monitoring your Cloud SQL instance is crucial to ensure that your connections are well-managed and your instance is performing optimally. You can use Stackdriver Monitoring to monitor connection metrics like connection count and connection errors.

- **View current connections:**

  ```
  gcloud sql instances describe my-sql-instance --
  format="value(connectionName)"
  ```

- **Set up monitoring alerts:**
  You can set up alerts in Google Cloud Monitoring (formerly Stackdriver) to notify you if your connections exceed a certain threshold.

Conclusion

Managing connections to Cloud SQL MySQL involves several steps, from setting up the instance and configuring secure connections to managing connection pools and monitoring performance. By following these best practices, you can ensure that your application interacts with the database efficiently, securely, and reliably.