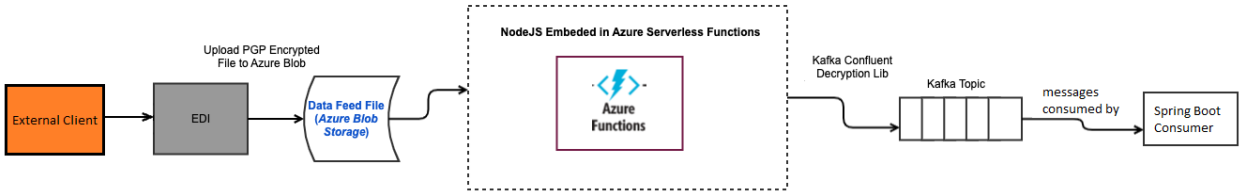
SERVERLESS SOLUTION FOR PUBLISHING FILE TO KAFKA

Project Description

This project is a Node.js application which can download encrypted files from azure blob storage, decrypt them and then publish the data to Kafka. The application can be deployed to Azure Functions and can be configured to work with serverless triggers.

Architecture



Prerequisites

- Node.js (v16 or later)
- A code editor such as VSCode
- Terminal
- An Azure account with an active subscription
- A Confluent cloud account
- Azure Functions Core Tools (v4.x)
- Azure CLI (v2.4 or later)

Setting up the Project

This project uses 3 external npm packages:

- **@azure/storage-blob**
- **openpgp**
- **kafkajs**

It also uses the **fs** and **stream** modules that come pre-packaged with node.

The required packages can be installed by running the following terminal command:

```
npm install @azure/storage-blob openpgp kafkajs
```

The required packages and functions can be imported using the following statements:

```
const BlobServiceClient =
require('@azure/storage-blob').BlobServiceClient;
const openpgp = require('openpgp');
const fs = require('fs');
const Transform = require('stream').Transform;
const Writable = require('stream').Writable;
const Kafka = require('kafkajs').Kafka;
```

These statements import the BlobServiceClient class from @azure/storage-blob, the Transform and Writable interfaces from stream, the Kafka class from kafkajs, and the complete fs and openpgp packages.

Async/await usage

Several methods used in this implementation are asynchronous, and often sequential execution of these methods is required for correct functioning of the project. Hence, all the code following the set-up is wrapped in an **async function** to permit the use of *await* when required.

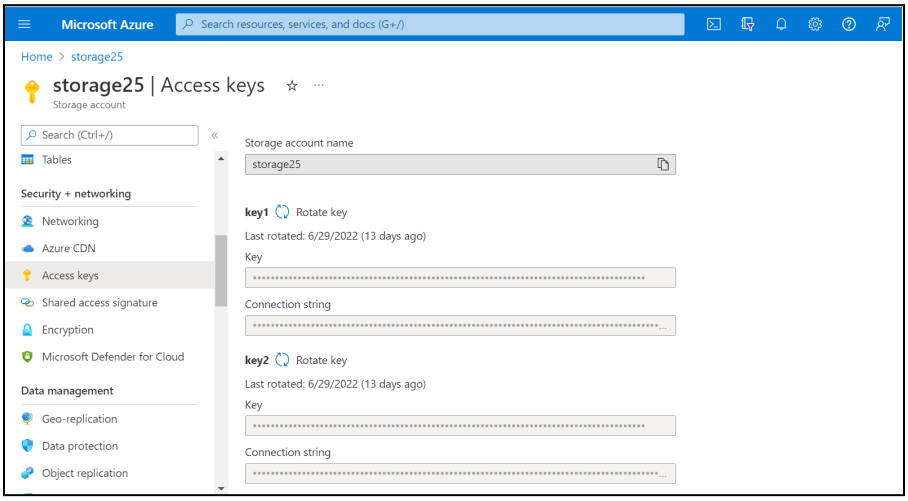
```
 async() => {
//code
//code
};
```

Primary Goals

1. Downloading file from Blob Storage
2. Decryption of file data
3. Publishing data to Kafka
4. Deploying to Azure Functions

TASK 1: Downloading file from Blob Storage

To connect to a Microsoft Azure Blob Storage account, the account's **connection string** is used. This can be obtained by going to
Microsoft Azure -> <your_storage_account> -> Access keys



The connection string is then used to set up clients in our application that connect to the storage account, using methods and classes from the @azure/storage-blob package.

```
const AZURE_STORAGE_CONNECTION_STRING = '<connection_string>';

const blobServiceClient =
BlobServiceClient.fromConnectionString(AZURE_STORAGE_CONNECTION_STRING);

const containerName = '<container_name>';
const containerClient =
blobServiceClient.getContainerClient(containerName);

const blobName = '<name_of_file>';
const blobClient = containerClient.getBlockBlobClient(blobName);
```

The *fromConnectionString* method of BlobServiceClient sets up a connection to the storage account. The *getContainerClient* method connects to the specific container the file lies in. The *getBlockBlobClient* method sets a connection to the specific file to be downloaded.

```
const encryptedDataStream = (await
blobClient.download(0)).readableStreamBody;

encryptedDataStream.setEncoding('utf-8');
```

The *download* method of the blobClient is used to download data from the blob. Specifically, a readable stream of data is created by adding the readableStreamBody attribute. The required encoding can be set at this stage (here, ‘utf-8’ is set).

TASK 2: Decryption of file data

The data received from Blob Storage here is PGP-encrypted. As such, the openpgp module is used to decrypt it. The decryption process requires a private key.
The key must also be retrieved from Blob Storage. If the key is stored in a different storage account, the aforementioned steps must be repeated to establish a connection to that account using its connection string. Here, the key is stored in the same container as the file. Hence, a new client is set up to connect to the key blob.

```
const keyBlobName = '<private_key>';
const keyBlobClient = containerClient.getBlockBlobClient(keyBlobName);
```

The key here is armored, so its passphrase must be used to decrypt it before it can be used. The key is also downloaded as a stream, but unlike the file, the key is completely downloaded and stored in a variable before proceeding.

```
const passphrase = '<passphrase>';
const keyStream = (await keyBlobClient.download(0)).readableStreamBody;
keyStream.setEncoding('utf-8');

let privateKeyArmored = '';

for await (const chunk of keyStream) {
    privateKeyArmored += Buffer.from(chunk).toString();
}
```

Now, the passphrase is used to decrypt the armored key and obtain the usable Private Key, using the *decryptKey* method of openpgp.

```
const privateKey = await openpgp.decryptKey({
     privateKey: await openpgp.readPrivateKey({armoredKey:
privateKeyArmored}),
     passphrase
});
```

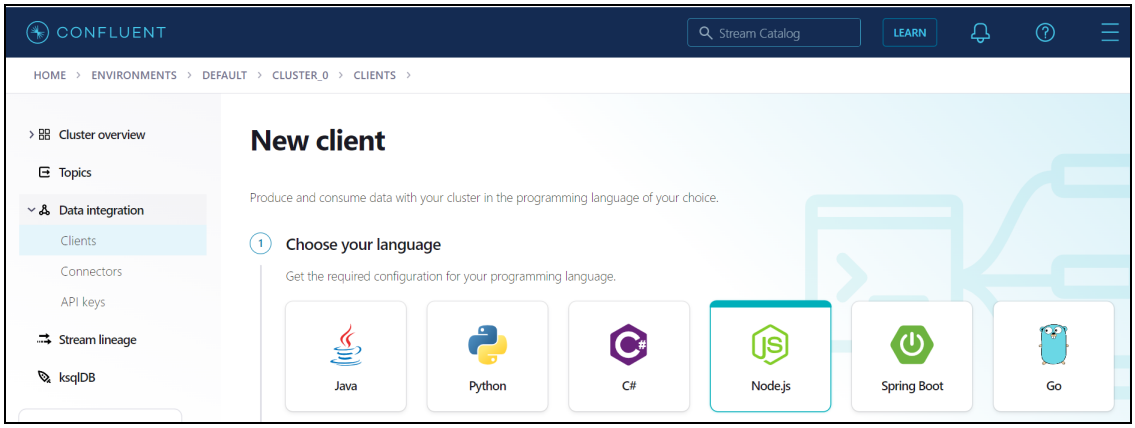
The private key is then used to decrypt the data read from the file, using the *readMessage* method of openpgp.

```
const decryptedData = await openpgp.decrypt({
      message: await openpgp.readMessage({armoredMessage:
encryptedDataStream}),
      decryptionKeys: privateKey,
      config: {allowUnauthenticatedStream: true}
 });
```

As the input is a data stream, and it is desired that the application should process data in chunks instead of downloading all data at once, the *allowUnauthenticatedStream* configuration is set to true in the *readMessage* method. This ensures that each chunk of data is made available to the rest of the application as soon as it is decrypted, without waiting for all the data to be decrypted.

TASK 3: Publishing data to Kafka

At this stage, a pipeline that extracts data from Blob Storage and decrypts it has been implemented. To publish the data to Kafka, the application must be connected to a Kafka cluster.
To establish this connection, the cluster credentials are required, which can be obtained by inspecting the cluster properties. In this case, a Confluent Kafka cluster is being used. The credentials can be obtained by going to:
Confluent Cloud Home -> Environments -> Default -> <Cluster_Name>
-> Data Integration -> Producers -> Set up a new Client -> Node.js



As SASL authentication is being used here, the API Key, Secret and Bootstrap server address are downloaded and used.

In the application code, the *Kafka* class of kafkajs is used to create a new Kafka client.

```
const kafka = new Kafka({
        ssl: true,
        brokers: ['<bootstrap_server_address>'],
        clientId: '<client_id>',
        sasl: {
            mechanism: 'plain',
            username: '<username>',
            password: '<password>'
          },
          connectionTimeout: 5000
      });
```

Using this client, a producer is set up, and a connection is established to the cluster.

```
const producer = kafka.producer();
await producer.connect();
```

Before publishing to Kafka, some additional processing needs to be carried out. The data pipeline at this stage provides chunks of data at a time, and it is required to publish each line of the data as an individual record. However, since chunks are created according to size, incomplete lines may be present in them. Further, there is a possibility of lines being split between chunks.

```
const t = new Transform({objectMode: true,
    transform(chunk, encoding, done) {
        let data = chunk.toString();
        if (this._lastLineData)
        data = this._lastLineData + data;

        let lines = data.split('\n');
        this._lastLineData = lines.splice(lines.length-1, 1)[0];

        lines.forEach(this.push.bind(this));
        done();
    },
    flush(done){
        if (this._lastLineData)
        this.push(this._lastLineData)
        this._lastLineData = null;
        done();
 }});
```

A transform stream is set up to handle this processing. For each chunk, its transform method does the following:

- Append the last line of the previous chunk to the beginning of the current chunk. This is done to ensure that incomplete lines that were split across chunks are rejoined. The last line for each chunk is stored in an attribute, *_lastLineData*.
- Separate the last line of the current chunk, and then push each line of the remaining data forward (using the newline character for demarcation).
- Store the last line in the *-lastLineData* attribute.

A flush method is also configured to run when all the data from the input stream has been consumed. It simply pushes forward the contents of *_lastLineData*.

Finally, to publish the data to Kafka, a writable stream is set up.

```
const w = new Writable({
        write(chunk, encoding, done){
              //additional processing if needed
              await producer.send({
                    topic: '<topic_name>',
                    messages: [{value: JSON.stringify(chunk.toString())}]
               });
              done();     } });
```

The writable stream can be used to carry out extra processing on each line of data if required (such as dropping fields). It then uses the *send* method of the kafka producer to publish data to the specified topic.

Once the streams have been created, the data pipeline can be completed. A finish event is added to the writable stream to terminate the application once there is no more data to publish. Lastly, the decrypted data stream is piped through the transform stream and into the writable stream. The *pipe* method ensures that Node handles backpressure automatically, thereby preventing data flow errors.

```
w.on('finish', () => {process.exit(1)});
decryptedData.data.pipe(t).pipe(w);
```

At this stage, the application is fully functional. If the async function is invoked, the application will read data from the specified Blob Storage file, decrypt it and publish it line by line to the specified Kafka topic.

TASK 4: Deploying to Azure Functions

Before deploying our application to Azure Functions, a local function project must be created, using the following terminal command:

```
func init <Project_Name> --javascript
```

This creates a folder containing various files for the project, including configuration files named local.settings.json and host.json.
After navigating to the project folder, the following terminal command is used to create a new function. Here, a Blob trigger template will be used to initialize the function.

```
func new --name <function_name> --template "Azure Blob Storage trigger"
```

This creates a subfolder matching the function name that contains a code file appropriate to the project's chosen language and a configuration file named *function.json*.

The index.js file in the subfolder is initialized with the following contents:

```
module.exports = async function (context, myBlob) {
    context.log("JavaScript blob trigger function processed blob\nBlob:",
context.bindingData.blobTrigger, "\n Blob Size:", myBlob.length, "Bytes");
};
```

The contents of this async function are replaced with the async function containing our application code. The import statements are also added to this file.
After navigating to the subfolder, the **npm install** commands are run again to install the modules in the function project.

```
npm install @azure/storage-blob openpgp kafkajs
```

In the local.settings.json configuration file in the parent directory, the AzureWebJobsStorage is set to the connection string of the storage account that is the source of the file.

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "AzureWebJobsStorage": "<connection_string>"
  }
}
```

Back in the subfolder, in the function.json configuration file, path is set to the name of the container that will contain the file, and connection is set to AzureWebJobsStorage.

```
{
  "bindings": [
    {
      "name": "myBlob",
      "type": "blobTrigger",
      "direction": "in",
      "path": "<container_name>",
      "connection": "AzureWebJobsStorage"
    }
  ]
}
```

The function project is now ready for deployment. Before deploying it to an Azure Functions App, supporting resources must first be created.

Initially, we sign into Azure using the following terminal command.

```
az login
```

Then, a resource group is created.

```
az group create --name <RESOURCE_GROUP_NAME> --location <REGION>
```

A storage account is created next.

```
az storage account create --name <STORAGE_NAME> --location <REGION>
--resource-group <RESOURCE_GROUP_NAME> --sku Standard_LRS
```

Finally, a function app is created.

```
az functionapp create --resource-group <RESOURCE_GROUP_NAME>
--consumption-plan-location <REGION> --runtime node --runtime-version 16
--functions-version 4 --name <APP_NAME> --storage-account <STORAGE_NAME>
```

In the functions app directory, the following terminal command is used to deploy our function project to the Azure Functions app.

```
func azure functionapp publish <APP_NAME>
```

At this stage, our solution is successfully deployed to Azure Functions. To invoke it, we simply need to upload the specified file to the specified Azure Blob Storage container, and the application gets triggered and starts publishing data to Kafka.
