Document Extractor Framework

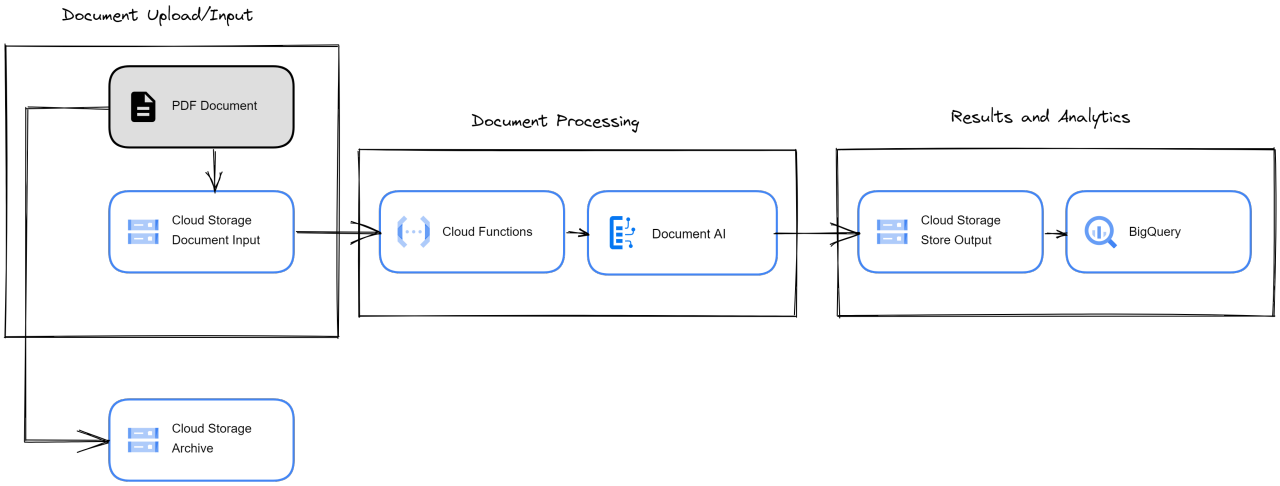
Accenture UKI Data Studio

January 2023

Introduction

The aim of the Document Extractor project is to create an end-to-end framework to automate information retrieval and storing. An end user should be able to upload documents of a standard template to a website and store all the relevant fields in an online database. Google Cloud Platform services form the foundation on which this framework is built. Several cloud services (Document AI, Cloud Functions, Compute Engine, Cloud Storage) are strung together in a pipeline which will streamline document processing. This is a fully cloud-native solution that aims to automate high volume repetitive data processing tasks for analysis and insight.

Architecture



Components

The following GCP components are used in this project:

- Google Cloud Storage
- Google Cloud Functions
- Google Document AI
- Google Big Query

Workflow

An user will upload a passport document in the form of a PDF file-type. This will be saved in a *Cloud Storage* bucket, which will then trigger a *Cloud Function* event. The document will be sent to a specialised passport parser processor in *Document AI* through an API call. The JSON results are first stored in an output bucket; following which the relevant information are written in a *Big Query* table.

Implemented:

- Google Cloud Storage input and output buckets
- Document AI processor custom trained for this work
- Cloud Functions Python Script
- Big Query output database
- Website Front End and APIs
- Google Cloud Storage Input and Output Bucket
- Google Cloud Storage Archive Bucket
- Testing batch processing with multiple file upload

Setup and Deployment

Clone the github repo:
git clone https://github.com/accenture-gcp-assets/gcp-assets-doc-extractor.git

Deploy Cloud Function:
gcloud functions deploy docxtractor-python −−region=YOUR_REGION −−runtime=python310 −−source=source_code −−entry-point=extract_doc −−trigger-bucket=YOUR_INPUT_BUCKET

Code files:

- **main.py** - This is the main Python script which has all the functions defined to create a processor, enable a process, send a request to the processor and populate Big Query with the results. The entry point to the Cloud Function is extract_doc function which takes the landing file's details from the function's events and executes the extraction process.
- **requirements.txt -** Contains Python dependencies required to run the code.
- **config.properties -** Configuration file which contains all parameters required to execute the code.

Code

This section briefly explains the code which is used to build the document extractor framework.

The following code snippet imports the necessary client libraries including the ones required for API access to GCP services:

```
import os
from google.cloud import bigquery
from google.cloud import storage
from google.api_core.client_options import ClientOptions
from google.api_core.exceptions import InternalServerError, RetryError
from google.cloud import documentai_v1 as documentai
from google.api_core.exceptions import FailedPrecondition
import pandas as pd
import configparser
import json
import re
import datetime
```

2. **Create Processor** - This function takes project ID, location, processor name and processor type as parameters and returns a processor object. The function begins by setting the API endpoint customised to the location (EU) where the processor will be based on. Following this, there will be a check if the mentioned processor is already running in the project and notify the user if it is (and returning the same); if not, it will provision/return a new one and print out some details specific to it. Currently a custom trained processor is used in the project.

```
def create_processor(project_id, location, processor_name, processor_type):
    """Creates and returns a processor object.
    Args:
        project_id (str): GCP project id.
        location (str): location where processor is to be created and run.
        processor_name (str): name of the processor to be created.
        processor_type (str): type of processor to be used for ex- US_PASSPORT_PROCESSOR.
    """
    # Setting the api endpoint
    opts = ClientOptions(api_endpoint=f"{location}-documentai.googleapis.com")
    client = documentai.DocumentProcessorServiceClient(client_options=opts)

    # Allocating processor to project ID and location
    parent = client.common_location_path(project_id, location)

    # Check if the processor already exists
    processor_list = client.list_processors(parent=parent)
    for processor in processor_list:
        if processor.display_name == processor_name:
            existing_processor = processor
            processor_found = True
            print('Processor already exists.')
            break
        else:
            processor_found = False
            print('Processor does not exist. Creating a new one...')

    if processor_found:
        processor = existing_processor
    else:
        processor = client.create_processor(parent=parent,
                                            processor=documentai.Processor(
                                                display_name=processor_name,
                                                type_=processor_type
                                            ))
    print(f"Processor Name: {processor.name}")
    print(f"Processor Display Name: {processor.display_name}")
    print(f"Processor Type: {processor.type_}")
    return(processor)
```

3. **Enable Processor:** Taking project id, location and the processor object (returned from Create Processor function), this function will enable the processor that was previously created so that the documents could be sent. The process starts by creating the API endpoint, locating the processor and enabling it. If in any case, the processor cannot be activated, the relevant error message will be returned for debugging assist.

```
def enable_processor(project_id, location, processor):
    """Enable the processor object created via create_processor function.
    Args:
        project_id (str): GCP project id.
        location (str): location where processor is to be created and run.
        processor (processor): object of Class processor created
        via create_processor function.
    """
    # setting the api end point to EU
    opts = ClientOptions(api_endpoint=f"{location}-documentai.googleapis.com")
    processor_id = str(processor.name.split('/')[-1])
    client = documentai.DocumentProcessorServiceClient(client_options=opts)
    processor_name = client.processor_path(project_id, location, processor_id)
    request = documentai.EnableProcessorRequest(name=processor_name)
    try:
        # make processor request
        operation = client.enable_processor(request=request)
        print(operation.operation.name)
        return(operation.result())
    except FailedPrecondition as e:
        # throws error if processor is already activated
        print(e.message)
```

4. **Send Processing Request:** This function locates the processor based on processor ID, project and location (sent via input parameters) and converts it to a raw document format. This is then sent to the custom Document AI processor through an API call. The text section of the result JSON output is returned from this function.

```
def send_processing_req(project_id, location, processor_id,
                                mime_type, bucket_name, file_name):
    """Send a request to Document AI API to process the landing
    file using processor created via create_processor function.
    Args:
        project_id (str): GCP project id.
        location (str): location where processor is to be created and run.
        processor_id (str): id for processor created via create_processor function.
        mime_type (str): media type for ex- application/pdf,image/jpeg.
        gcs_input_uri (str): uri for the file landing on the input bucket.
    """
    docai_client = documentai.DocumentProcessorServiceClient(
    client_options = ClientOptions(api_endpoint=f'{location}-documentai.googleapis.com')
    )
    resource_name = docai_client.processor_path(project_id, location, processor_id)
    client = storage.Client()
    bucket = client.get_bucket(bucket_name)
    blob = bucket.get_blob(file_name)
    blob_as_string = blob.download_as_string()
    raw_doc = documentai.RawDocument(content=blob_as_string, mime_type=mime_type)
    request = documentai.ProcessRequest(name=resource_name, raw_document=raw_doc)

    result = docai_client.process_document(request=request)

    document_object = result.document
    print('Document processing complete')
    return(document_object)
```

5. **Extract Document Entities -** The document output will have several fields (not only the relevant info, but also bounding boxes and their co-ordinates inside the page), all of which are not relevant in this work. Hence, a function has been written which outputs only the important fields (name, nationality, date of issue etc) in the form of a dictionary when the output document object is provided. This function is crucial in ensuring the output data is as clean and understandable as possible before it is written to Big Query.

```
def extract_document_entities(document: documentai.Document) -> dict:
    """Get all entities from a document and output as a dictionary
    Flattens nested entities/properties
    Format: entity.type_: entity.mention_text OR entity.normalized_value.text
    """

    document_entities: Dict[str, Any] = {}

    def extract_document_entity(entity: documentai.Document.Entity):
        """
        Extract Single Entity and Add to Entity Dictionary
        """
        entity_key = entity.type_.replace('/', '_')
        entity_key = entity.type_.replace(' ', '_')
        entity_key = entity.type_.replace('-', '_')
        normalized_value = getattr(entity, 'normalized_value', None)

        new_entity_value = (
            normalized_value.text if normalized_value else entity.mention_text
        )

        existing_entity = document_entities.get(entity_key)
        # for entities that can have multiple lines
        if existing_entity:
            # change entity type to a list
            if not isinstance(existing_entity, list):
                existing_entity = list([existing_entity])

            existing_entity.append(new_entity_value)
            document_entities[entity_key] = existing_entity
        else:
            document_entities.update({entity_key: new_entity_value})

    for entity in document.entities:
        extract_document_entity(entity)

        for prop in entity.properties:
            extract_document_entity(prop)

    return document_entities
```

6. **Process Document -** The process document function takes document_object, input_file_name, dataset_name, entities_table_name, output_bucket_name as input, grabs the important entities by the calling the extract document entities function (mentioned above) and populates with the relevant info. Finally, it writes the output to Big Query and Google Cloud Storage output bucket simultaneously by calling write_to_bq and write_to_gcs functions (mentioned below).

```
def process_document(document_object, input_bucket, input_file_name, dataset_name, entities_table
    """ Save the extracted entities to Bigquery & Cloud Storage
    Args:
        document_object (documentai.Document): output returned from Document AI API
        input_bucket (str): GCS bucket where the input file is located
        input_file_name (str): name of the file which is being processed
        dataset_name (str): dataset in BigQuery where output is to be written to
        entities_table_name (str): table in BigQuery where output is to be written to
        output_bucket_name (str): GCS bucket where output is to be written to
        archive_bucket_name: GCS bucket where the input file is to be moved to
    """

    # reading all entities into a dict to write to bq table
    entities = extract_document_entities(document_object)
    entities = format_keys(entities)
    entities['input_file_name'] = input_file_name
    entities['text'] = document_object.text
    entities['time_stamp'] = str(datetime.datetime.now())

    # print('Entities: ', entities)
    # print('Writing DocAI Entities to bq')
    print(f'Writing result for {input_file_name} to bq')

    write_to_bq(dataset_name, entities_table_name, entities)
    write_to_gcs(entities, output_bucket_name, input_file_name)
    cleanup_gcs(
            input_bucket,
            input_file_name,
            archive_bucket_name,
        )

    return
```

7. **Write to GCS -** The formatted information extracted from the passport is finally uploaded to a GCS bucket as a binary large object (blob) file format. If that fails for some reason, this function will return the error message for debugging. final_output_text and bucket_name (to locate an existing bucket) are the two parameters this function takes returning True if the upload operation is successful and False otherwise.

```
def write_to_gcs(entities_extracted_dict, bucket_name, input_file_name):
    """
    Args:
        entities_extracted_dict (dict): dictionary of entities extracted from Document AI output
        bucket_name (str): Cloud Storage bucket where output is to be written to
        input_file_name (str): name of the file being processed
    """
    # convert extracted dictionary of entities to json string
    json_string = json.dumps(entities_extracted_dict, indent=4)

    # create a Cloud Storage client and write to an object.
    storage_client = storage.Client()
    output_file = str(input_file_name.split('.')[0]) + '_processed.json'
    try:
        # Use the output bucket passed as parameter
        bucket = storage_client.get_bucket(bucket_name)
        # upload to bucket as a binary large object (blob)
        blob = bucket.blob(output_file)
        blob.upload_from_string(json_string)
        return True
    except Exception as e:
        # print error if upload fails
        print(e)
        return False
```

8. **Write to BQ -** The write_to_BQ function takes the extracted entities dictionary, input filename and the Biq Query table name and writes the extracted information from the Document AI processor. First, the function looks for existing tables with the name provided - if it is not found; it creates a new table with relevant schema, otherwise it uses an existing table. This is the last step in this framework.

```
def write_to_bq(dataset_name, table_name, entities_extracted_dict):
    """Use BigQuery client API (Python) to write output data to bigquery
    Args:
        dataset_name (str): dataset in BigQuery where output is to be written to
        table_name (str): table in BigQuery where output is to be written to
        entities_extracted_dict (dict): dictionary of entities
        extracted from Document AI output
    """
    bq_client = bigquery.Client()
    dataset_ref = bq_client.dataset(dataset_name)
    table_ref = dataset_ref.table(table_name)

    def table_exists(client, table_ref):
        try:
            client.get_table(table_ref)
            return True
        except:
            return False

    if not table_exists(bq_client, table_ref):
        print('table does not exist, creating table')
        schema = [bigquery.SchemaField(
            'input_file_name', 'STRING', mode='NULLABLE')]
        table = bigquery.Table(table_ref, schema=schema)
        table = bq_client.create_table(table)
        print(f'Created table {table.project}.{table.dataset_id}.{table.table_id}')

    row_to_insert = []
    row_to_insert.append(entities_extracted_dict)

    json_data = json.dumps(row_to_insert, sort_keys=False)
    json_object = json.loads(json_data)

    schema_update_options = [
        bigquery.SchemaUpdateOption.ALLOW_FIELD_ADDITION,
        bigquery.SchemaUpdateOption.ALLOW_FIELD_RELAXATION,
    ]
    source_format = bigquery.SourceFormat.NEWLINE_DELIMITED_JSON

    job_config = bigquery.LoadJobConfig(
        schema_update_options=schema_update_options,
        source_format=source_format
    )

    job = bq_client.load_table_from_json(
        json_object, table_ref, job_config=job_config)
    print(job.result())
```

9. **Cleanup GCS -** The cleanup_GCS function moves all the files from the input GCS bucket to the archive GCS bucket for long term storage. It also cleans up the input bucket by deleting all the processed files.

```
def cleanup_gcs(input_bucket, input_filename, archive_bucket):
    """
    Moving Input Files to Archive
    """
    storage_client = storage.Client()

    # Copy input file to archive bucket
    source_bucket = storage_client.bucket(input_bucket)
    source_blob = source_bucket.blob(input_filename)
    destination_bucket = storage_client.bucket(archive_bucket)

    source_bucket.copy_blob(source_blob, destination_bucket, input_filename)

    # delete from the input folder
    source_blob.delete()
    print(f"{source_blob.name} deleted")
```