Process Documents with Python Using the Document AI API

Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell** >_ at the top of the Google Cloud console.

2. Click through the following windows:

   - Continue through the Cloud Shell information window.
   - Authorize Cloud Shell to use your credentials to make Google Cloud API calls.

When you are connected, you are already authenticated, and the project is set to your **Project_ID**, `qwiklabs-gcp-03-1d398914d298`. The output contains a line that declares the **Project_ID** for this session:

```
Your Cloud Platform project in this session is set to qwiklabs-gcp-03-1d398914d298
```

`gcloud` is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

3. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

4. Click **Authorize**.

Output:

```
ACTIVE: *
ACCOUNT: "ACCOUNT"
```

```
To set the active account, run:
    $ gcloud config set account `ACCOUNT`
```

5. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Output:

```
[core]
project = qwiklabs-gcp-03-1d398914d298
```

Note: For full documentation of `gcloud`, in Google Cloud, refer to the gcloud CLI overview guide.

Task 1. Create and test a general form processor

In this task you will enable the Document AI API and create and test a general form processor. The general form processor will process any type of document and extract all the text content it can identify in the document. It is not limited to printed text, it can handle handwritten text and text in any orientation, supports a number of languages, and understands how form data elements are related to each other so that you can extract key:value pairs for form fields that have text labels.

Enable the Cloud Document AI API

Before you can begin using Document AI, you must enable the API.

1. In Cloud Console, from the **Navigation menu** (☰), click **APIs & services > Library**.

2. Search for **Cloud Document AI API**, then click the **Enable** button to use the API in your Google Cloud project.

If the Cloud Document AI API is already enabled you will see the **Manage** button and you can continue with the rest of the lab.

Create a general form processor

Create a Document AI processor using the Document AI form parser.

1. In the console, on the **Navigation menu** (☰), click **Document AI > Overview**.

2. Click **Explore processors** and click **Create Processor** for **Form Parser**, which is a type of general processor.

3. Specify the processor name as **form-parser** and select the region **US (United States)** from the list.

4. Click **Create** to create the general form-parser processor.

This will create the processor and return to the processor details page that will display the processor ID, status, and the prediction endpoint.

5. Make a note of the Processor ID as you will need to update variables in JupyterLab notebooks with the Processor ID in later tasks.

Task 2. Configure your Vertex AI Workbench instance to perform Document AI API calls

Next, connect to JupyterLab running on the Vertex AI Workbench instance that was created for you when the lab was started, then configure that environment for the remaining lab tasks.

1. In the Google Cloud console, on the **Navigation menu** (☰), click **Vertex AI > Workbench**.

2. Find the `jupyterlab` instance and click on the **Open JupyterLab** button.

The JupyterLab interface for your Workbench instance opens in a new browser tab.

3. Click **Terminal** to open a terminal shell inside the Vertex AI Workbench instance.

4. Enter the following command in the terminal shell to import the lab files into your Vertex AI Workbench instance:

```
gsutil cp gs://qwiklabs-gcp-03-1d398914d298-labconfig-
bucket/notebooks/*.ipynb .
```

Copied!

content_copy

5. Enter the following command in the terminal shell to install the Python client libraries required for Document AI and other required libraries:

```
python -m pip install --upgrade google-cloud-core google-cloud-
documentai google-cloud-storage prettytable
```

Copied!

content_copy

You should see output indicating that the libraries have been installed successfully.

Note: In case of permission related errors, re-run the command to ensure successful installation of the libraries. It can take a few minutes for the permissions to be applied.

6. Enter the following command in the terminal shell to import the sample health intake form:

```
gsutil cp gs://qwiklabs-gcp-03-1d398914d298-labconfig-bucket/health-
intake-form.pdf form.pdf
```

Copied!

content_copy

7. In the notebook interface open the JupyterLab notebook called `documentai-sync-v1.0.0.ipynb`.

8. In the **Select Kernel** dialog, choose **Python 3** from the list of available kernels.

Task 3. Make a synchronous process document request

Make a process document call using a synchronous Document AI API call. For processing large amounts of documents at a time you can also use the asynchronous API which you will use in a later task.

Review the Python code for synchronous Document AI API calls

Take a minute to review the Python code in the `documentai-sync-v1.0.0.ipynb` notebook.

The first code block imports the required libraries and initializes some variables.

```
from google.cloud import documentai_v1beta3 as documentai
from google.cloud import storage
from prettytable import PrettyTable

project_id = %system gcloud config get-value core/project
project_id = project_id[0]
location = 'us'
file_path = 'form.pdf'
```

The **Set your Processor ID** code cell sets the Processor ID that you have to manually set before you can process documents with the notebook.

```
processor_id = 'PROCESSOR_ID' # TODO: Replace with a valid Processor ID
```

You will need the Document AI processor ID of the processor you created in Task 1 for this step.

Tip: If you did not save it, then in the Cloud Console tab open the **Navigation menu** (☰), click **Document AI > My processors**, then click the name of your processor to open the details page. From here you can copy the processor ID.

The **Process Document Function** code cell defines the `process_document` function that is used to make a synchronous call to a Document AI processor. The function creates a Document AI API client object.

The processor name required by the API call is created using the `project_id`,`locations`, and `processor_id` parameters and the sample PDF document is read in and stored in a `mime_type` structure.

The function creates a request object that contains the full processor name of the document and uses that object as the parameter for a synchronous call to the Document AI API client. If the request is successful the document object that is returned will include properties that contain the entities detected in the form.

```
def process_document(
        project_id=project_id, location=location,
        processor_id=processor_id,  file_path=file_path
):
    # Instantiates a client
    client = documentai.DocumentProcessorServiceClient()
    # The full resource name of the processor, e.g.:
    # projects/project-id/locations/location/processor/processor-id
    # You must create new processors in the Cloud Console first
    name =
f"projects/{project_id}/locations/{location}/processors/{processor_id}"
    with open(file_path, "rb") as image:
        image_content = image.read()
    # Read the file into memory
    document = {"content": image_content, "mime_type":
"application/pdf"}
    # Configure the process request
    request = {"name": name, "document": document}
    # Use the Document AI client to process the sample form
    result = client.process_document(request=request)
    return result.document
```

The **Process Document** code cell calls the `process_document` function, saves the response in the `document` variable, and prints the raw text that has been detected. All of the processors will report some data for the `document.text` property.

```
document=process_document()
# print all detected text.
# All document processors will display the text content
print("Document processing complete.")
print("Text: {}".format(document.text))
```

The **Get Text Function** code cell defines the `get_text()` function that retrieves the text for a named element using the `text_anchor` `start_index` and `end_index` properties of the named element's `text_segments`. This function is used to retrieve the form name and form value for form data if that data is returned by the processor.

```
def get_text(doc_element: dict, document: dict):
    """
    Document AI identifies form fields by their offsets
    in document text. This function converts offsets
    to text snippets.
    """
    response = ""
    # If a text segment spans several lines, it will
    # be stored in different text segments.
    for segment in doc_element.text_anchor.text_segments:
        start_index = (
            int(segment.start_index)
            if segment in doc_element.text_anchor.text_segments
            else 0
        )
        end_index = int(segment.end_index)
        response += document.text[start_index:end_index]
    return response
```

The **Display Form Data** cell iterates over all pages that have been detected and for each `form_field` detected it uses the `get_text()` function to retrieve the field name and field value. Those values are then printed out, along with their corresponding confidence scores. Form data will be returned by processors that use the general form parser or the specialized parsers but will not be returned by processors that were created with the Document OCR parser.

```
document_pages = document.pages
print("Form data detected:\n")
```

```
# For each page fetch each form field and display fieldname, value and confidence scores
for page in document_pages:
    print("Page Number:{}".format(page.page_number))
    for form_field in page.form_fields:
        fieldName=get_text(form_field.field_name,document)
        nameConfidence = round(form_field.field_name.confidence,4)
        fieldValue = get_text(form_field.field_value,document)
        valueConfidence = round(form_field.field_value.confidence,4)
        print(fieldName+fieldValue +"  (Confidence Scores: (Name) "+str(nameConfidence)+", (Value) "+str(valueConfidence)+")\n")
```

The **Display Entity Data** cell extracts entity data from the document object and displays the entity type, value, and confidence properties for each entity detected. Entity data is *only* returned by processors that use specialized Document AI parsers such as the Procurement Expense parser. The general form parser and the Document OCR parser will not return entity data.

```
if 'entities' in dir(document):
    entities = document.entities
    # Grab each key/value pair and their confidence scores.
    table = PrettyTable(['Type', 'Value', 'Confidence'])
    for entity in entities:
    entity_type = entity.type_
    value = entity.mention_text
    confience = round(entity.confidence,4)
    table.add_row([entity_type, value, confience])
    print(table)
else:
    print("Document does not contain entity data.")
```

Task 4. Run the synchronous Document AI Python code

Execute the code to make synchronous calls to the Document AI API in the JupyterLab notebook.

1. In the second **Set your Processor ID** code cell replace the `PROCESSOR_ID` placeholder text with the Processor ID for the **form-parser** processor you created in an earlier step.

2. Select the first cell, click the **Run** menu and then click **Run Selected Cell and All Below** to run all the code in the notebook.

If you have used the sample health intake form, you will data similar to the following for the output cell for the form data:

```
Form data detected:

Page Number:1 Phone #: (906) 917-3486 (Confidence Scores: (Name) 1.0,
(Value) 1.0) ... Date: 9/14/19 (Confidence Scores: (Name) 0.9999,
(Value) 0.9999) ... Name: Sally Walker (Confidence Scores: (Name)
0.9973, (Value) 0.9973) ...
```

If you are able to create a specialised processor the final cell will display entity data, otherwise it will show an empty table.

3. In the JupyterLab menu click **File** and then click **Save Notebook** to save your progress.

Task 5. Create a Document AI Document OCR processor

In this task you will create a Document AI processor using the general Document OCR parser.

1. From the **Navigation menu**, click **Document AI > Overview**.

2. Click **Explore Processors** and then click **Create Processor** for **Document OCR**. This is a type of general processor.

3. Specify the processor name as **ocr-processor** and select the region **US (United States)** from the list.

4. Click **Create** to create your processor.

5. Make a note of the **processor ID**. You will need to specify this in a later task.

Task 6. Prepare your environment for asynchronous Document AI API calls

In this task you upload the sample JupyterLab notebook to test asynchronous Document AI API calls and copy some sample forms for the lab to Cloud Storage for asynchronous processing.

1. Click the **Terminal** tab to re-open the terminal shell inside the Vertex AI Workbench instance.

2. Create a Cloud Storage bucket for the input documents and copy the sample W2 forms into the bucket:

```
export PROJECT_ID="$(gcloud config get-value core/project)"
export BUCKET="${PROJECT_ID}"_doc_ai_async
gsutil mb gs://${BUCKET}
gsutil -m cp gs://qwiklabs-gcp-03-1d398914d298-labconfig-bucket/async/*.* gs://${BUCKET}/input
```

Copied!

content_copy

3. In the notebook interface open the JupyterLab notebook called `documentai-async-v1.0.0.ipynb`.

4. In the **Select Kernel** dialog, choose **Python 3** from the list of available kernels.

Task 7. Make an asynchronous process document request

Review the Python code for asynchronous Document AI API calls

Take a minute to review the Python code in the `documentai-async-v1.0.0.ipynb` notebook.

The first code cell imports the required libraries.

```
from google.cloud import documentai_v1beta3 as documentai
from google.cloud import storage

import re
import os
import pandas as pd
import simplejson as json
```

The **Set your Processor ID** code cell sets the Processor ID that you have to manually set before you can process documents with the notebook.

```
processor_id = "PROCESSOR_ID"  # TODO: Replace with a valid Processor ID
```

The **Set your variables** code cell defines the parameters that will be used to make the asynchronous call, including the location of the input and output Cloud Storage buckets that will be used for the source data and output files. You will update the placeholder values in this cell for the `PROJECT_ID` and the `PROCESSOR_ID` in the next section of the lab before you run the code. The other variables contain defaults for the processor location, input Cloud Storage Bucket, and output Cloud Storage bucket that you do not need to change.

```
project_id = %system gcloud config get-value core/project
project_id = project_id[0]
location = 'us'           # Replace with 'eu' if processor does not use 'us' location
gcs_input_bucket  = project_id+"_doc_ai_async"   # Bucket name only, no gs:// prefix
gcs_input_prefix  = "input/"                     # Input bucket folder e.g. input/
gcs_output_bucket = project_id+"_doc_ai_async"   # Bucket name only, no gs:// prefix
gcs_output_prefix = "output/"                    # Input bucket folder e.g. output/
timeout = 300
```

The **Define Google Cloud client objects** code cell initializes the Document AI and Cloud Storage clients.

```
client_options = {"api_endpoint": "{}-documentai.googleapis.com".format(location)}
client = documentai.DocumentProcessorServiceClient(client_options=client_options)
storage_client = storage.Client()
```

The **Create input configuration** code cell creates the input configuration array parameter for the source data that will be passed to the asynchronous Document AI request as an input configuration. This array stores the Cloud Storage source location, and the mime type, for each of the files that are found in the input Cloud Storage location.

```
blobs = storage_client.list_blobs(gcs_input_bucket, prefix=gcs_input_prefix)
input_configs = []
print("Input Files:")
for blob in blobs:
    if ".pdf" in blob.name:
        source = "gs://{bucket}/{name}".format(bucket = gcs_input_bucket, name = blob.name)
        print(source)
        input_config = documentai.types.document_processor_service.BatchProcessRequest.BatchInputConfig(
            gcs_source=source, mime_type="application/pdf"
        )
        input_configs.append(input_config)
```

The **Create output configuration** code cell creates the output parameter for the asynchronous request containing the output Cloud Storage bucket location and stores that as a Document AI batch output configuration.

```
destination_uri = f"gs://{gcs_output_bucket}/{gcs_output_prefix}"
output_config = documentai.types.document_processor_service.BatchProcessRequest.BatchOutputConfig(
    gcs_destination=destination_uri
)
```

The **Create the Document AI API request** code cell builds the asynchronous Document AI batch process request object using the input and output configuration objects.

```
name = f"projects/{project_id}/locations/{location}/processors/{processor_id}"
request = documentai.types.document_processor_service.BatchProcessRequest(
    name=name,
    input_configs=input_configs,
    output_config=output_config,
)
```

The **Start the batch (asynchronous) API operation** code cell makes an asynchronous document process request by passing the request object to the `batch_process_documents()` method. This is an asynchronous call so you use the `result()` method to force the notebook to wait until the background asynchronous job has completed.

```
operation = client.batch_process_documents(request)
```

```
# Wait for the operation to finish
operation.result(timeout=timeout)
print ("Batch process  completed.")
```

The **Fetch list of output files** cell enumerates the objects in the output bucket location as defined in the `destination_uri` variable.

The **Display detected text from asynchronous output JSON files** cell loads each output JSON file that is found as a Document AI document object and the text data detected by the Document OCR processor is printed out.

The **Display entity data** cell will display any entity data that is found, however, entity data is only available for processors that were created using a specialized parser. Entity data will not be displayed with the general Document AI OCR parser used in this task.

Run the asynchronous Document AI Python code

Use the sample code provided for you in the Jupyterlab notebook to process documents asynchronously using a Document AI batch processing request.

1. In the second code cell replace the `PROCESSOR_ID` placeholder text with the Processor ID for the **form-parser** processor you created in an earlier step.

2. Select the first cell, click the **Run** menu and then click **Run Selected Cell and All Below** to run all the code in the notebook.

3. As the code cells execute, you can step through the notebook reviewing the code and the comments that explain how the asynchronous request object is created and used.

The notebook will take a minute or two to wait for the asynchronous batch process operation to complete at the **Start the batch (asynchronous) API operation** code cell. While the batch process API call itself is asynchronous the notebook uses the `result` method to force the notebook to wait until the asynchronous call has completed before enumerating and displaying the output data.

If the asynchronous job takes longer than expected and times out you may have to run the remaining cells again to display the output. These are the cells after the **Start the batch (asynchronous) API operation** cell.

Your output will contain text listing the Document AI data detected in each file. The Document OCR parser does not detect form or entity data so there will be no form or entity data produced. If you can create a specialised processor then you will also see entity data printed out by the final cell.

4. In the JupyterLab menu click **File** and then click **Save Notebook** to save your progress.

```
Document processing complete.
Text: FakeDoc M.D.
HEALTH INTAKE FORM
Please fill out the questionnaire carefully. The information you
provide will be used to complete
your health profile and will be kept confidential.
Date:
Sally
Walker
Name:
9/14/19
...
```