Create your first Dataflow Flex Template and set up a CI/CD pipeline for it on Cloud Build

Miren Esnaola
Cloud Consultant @ Google Cloud
Github: apichick
bit.ly/3CcGeZo

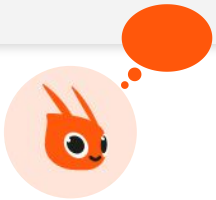
BEAM SUMMIT

What are Flex Templates?

Flex templates package Dataflow pipeline code, including dependencies, as Docker images and stage them in Google Container Registry (GCR). Template spec files referencing the image location and the parameters to pass are created and stored in Google Cloud Storage (GCS). Users can invoke a pipeline in the console, with the CLI or using the API by referring to the spec file.

Why are Flex Templates so cool?

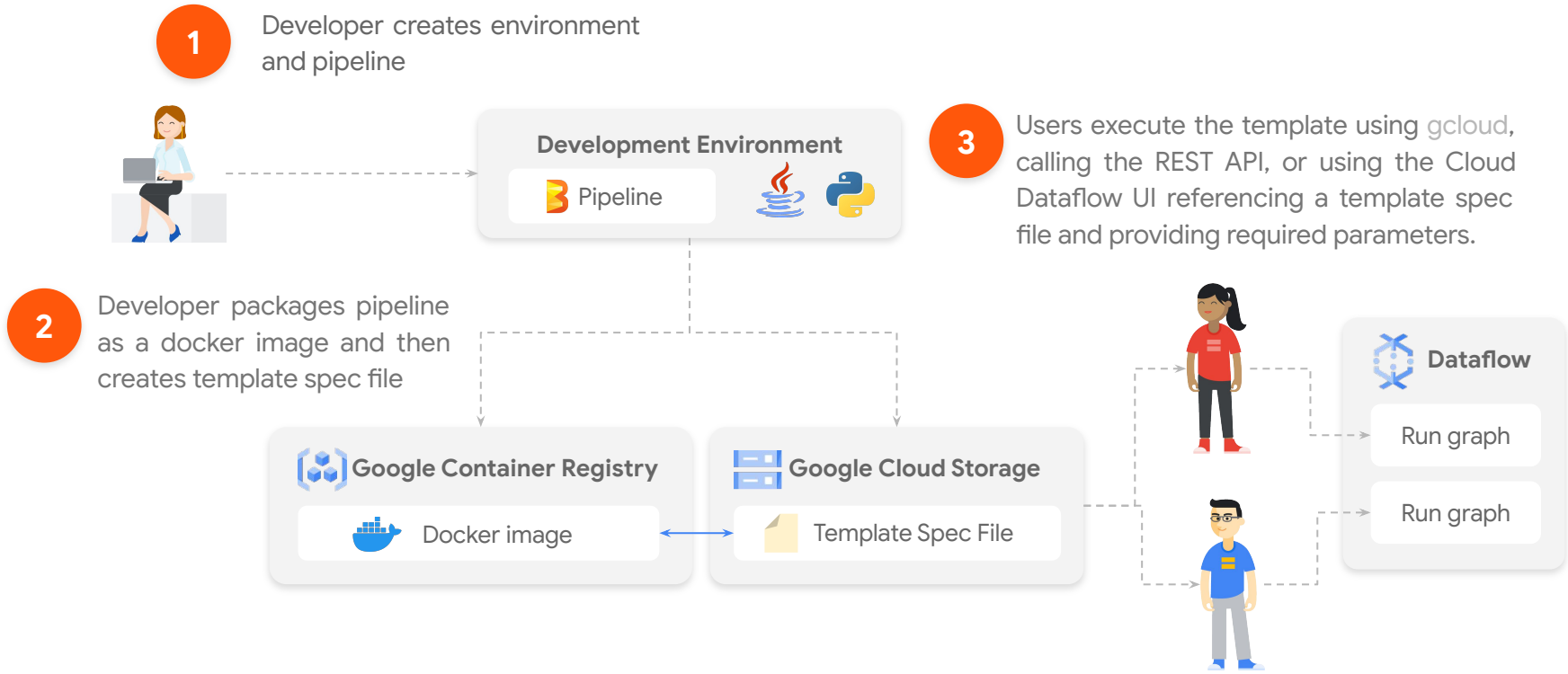
I ♥ Flex Templates



Flex templates separate pipeline authoring (development + packaging) from execution so:

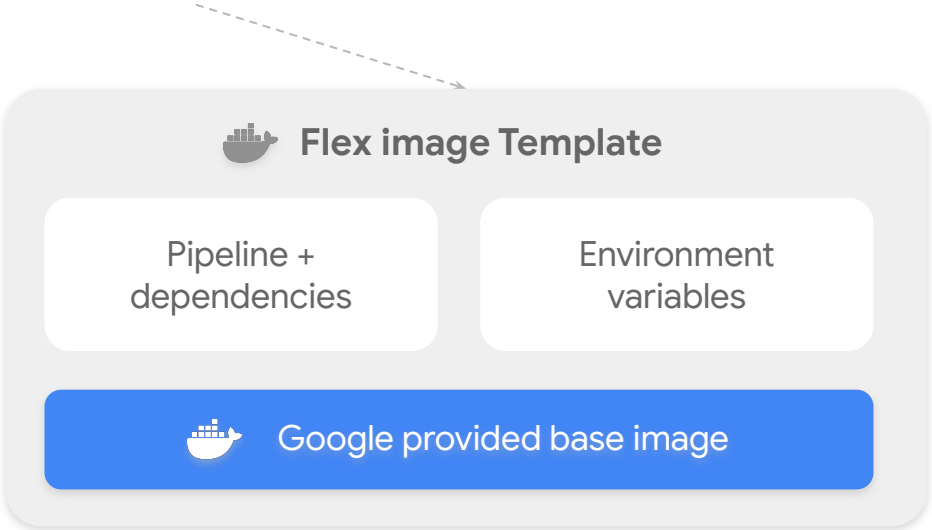
- A platform team can build and share reusable pipelines.
- Business users can launch pipelines without the need of environment dependencies.
- Pipeline execution can be scheduled using cloud-native services *(e.g. Cloud Scheduler)*.

Why are Flex Templates so cool?



Flex Template Image

Docker image does not contain the JSON serialized execution graph.



Flex Template Spec File

This spec file contains all of the necessary information to run the job:

- Container Registry image location,
- SDK language
- Metadata
  - Name
  - Description
  - Required or optional parameters*

** Regex can be used to validate input parameters provided by user.*

```
{
  "image": "gcr.io/project-id/image-name",
  "metadata": {
    "name": "Streaming data generator",
    "description": "Generates Synthetic data as
per user specified schema at a fixed QPS and
writes to Sink of user choice.",
    "parameters": [
      {
        "name": "schemaLocation",
        "label": "Location of Schema file.",
        "helpText": "GCS path of schema location",
        "is_optional": false,
        "regexes": [
          "^gs:\\/\\/[^\\n\\r]+$"
        ],
        "paramType": "GCS_READ_FILE"
      },
      ...
   ],
   "sdk_info": {
     "language": "JAVA"
   }
}
```

Flex Templates

Development

Setting up the environment

Python

1. (Optional) Enable Kaniko cache use by default.

```
$ gcloud config set builds/use_kaniko True
```

Kaniko cache is a Cloud Build feature that caches container build artifacts by storing and indexing intermediate layers within a container image registry, such as Google Container Registry.

Setting up the environment

Python

2 Create the Dockerfile.

```
FROM gcr.io/dataflow-templates-base/python3-template-launcher-base

ARG WORKDIR=/template
RUN mkdir -p ${WORKDIR}
WORKDIR ${WORKDIR}

# Due to a change in the Apache Beam base image in version 2.24, you must to install
# libffi-dev manually as a dependency. For more information:
# https://github.com/GoogleCloudPlatform/python-docs-samples/issues/4891
RUN apt-get update && apt-get install -y libffi-dev && rm -rf /var/lib/apt/lists/*

COPY requirements.txt .
COPY main.py .

ENV FLEX_TEMPLATE_PYTHON_REQUIREMENTS_FILE="${WORKDIR}/requirements.txt"
ENV FLEX_TEMPLATE_PYTHON_PY_FILE="${WORKDIR}/main.py"

RUN pip install -U -r ./requirements.txt
```

(*) Do not explicitly install job packages in the Dockerfile. Packages to be installed must be referenced in the 'requirements' file.

Setting up the environment

Python

3 Build the docker image.

```
$ export TEMPLATE_IMAGE="gcr.io/<PROJECT>/<IMAGE_NAME>:<TAG>"

$ gcloud builds submit --tag <TEMPLATE_IMAGE> .
```

Creating the template

Python

1 Create the metadata file in the local file system with the name and description of the template and the parameters it takes.

```
{
 "name": "<NAME>",
 "description": "<DESCRIPTION>",
 "parameters": [
   {
     "name": "<NAME>",
     "label": "<LABEL>",
     "helpText": "<HELP_TEXT>",
     "regexes": [
       "<REGEXP>",
       ...
     ]
   },
   ...
 ]
}
```

Creating the template

Python

2 Create the template spec file in a Google Cloud Storage location

```
$ export TEMPLATE_PATH="gs://<BUCKET_NAME>/<PATH>/<FILENAME>"

$ gcloud dataflow flex-template build <TEMPLATE_PATH> \
   --image "<TEMPLATE_IMAGE>" \
   --sdk-language "PYTHON" \
   --metadata-file "<METADATA_FILE_PATH>"
```

Flex Templates

Execution

How do you run a template?

Option A — Using the CLI

```
$ gcloud dataflow flex-template run "<JOB_NAME>" \
      --template-file-gcs-location "<TEMPLATE_PATH>" \
      --parameters <PARAMETER_1_NAME>="<PARAMETER_1_VALUE>" \
      --parameters <PARAMETER_2_NAME>="<PARAMETER_2_VALUE>"\
      …
      --parameters <PARAMETER_N_NAME>="<PARAMETER_N_VALUE>"\
      --region <REGION>
```

How do you run a template?

Option B — REST API

```
$ curl -X POST \
  "https://dataflow.googleapis.com/v1b3/projects/$PROJECT/locations/<REGION>/flexTemplates:launch" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $(gcloud auth print-access-token)" \
  -d '{
    "launch_parameter": {
      "jobName": "<JOB_NAME>",
      "parameters": {
        "<PARAM_NAME_1>": "<PARAM_VALUE_1>",
        "<PARAM_NAME_2>": "<PARAM_VALUE_2>",
        ...
        "<PARAM_NAME_N>": "<PARAM_VALUE_N>",
      },
      "containerSpecGcsPath": "<TEMPLATE_PATH>"
    }
  }'
```

Flex Templates

Access Control

Understanding Flex Template Permissions

Build

- Storage Admin (`roles/storage.admin`)

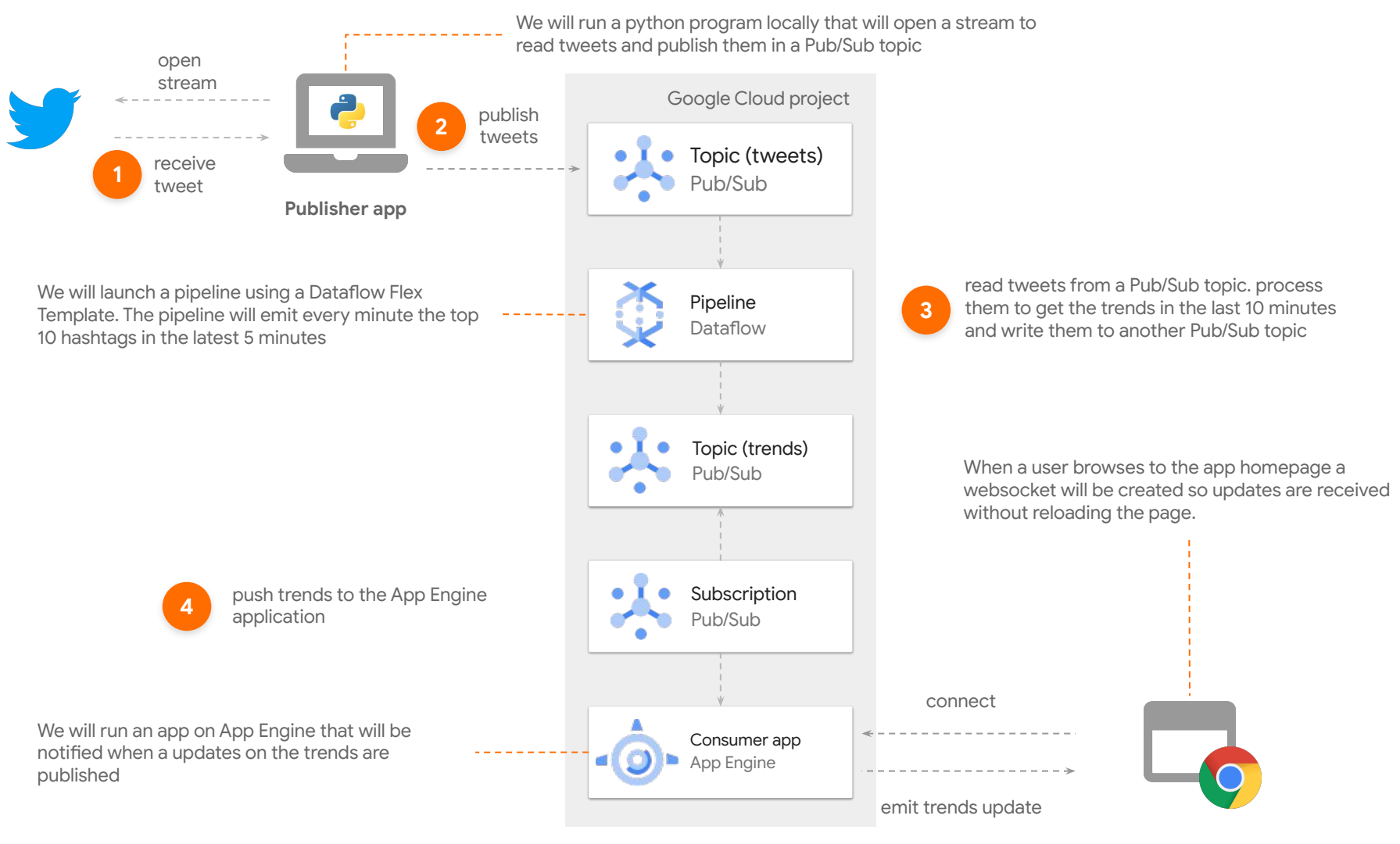
Run

- Storage Object Admin (`roles/storage.objectAdmin`)
- Viewer (`roles/viewer`)
- Dataflow Worker (`roles/dataflow.worker`)

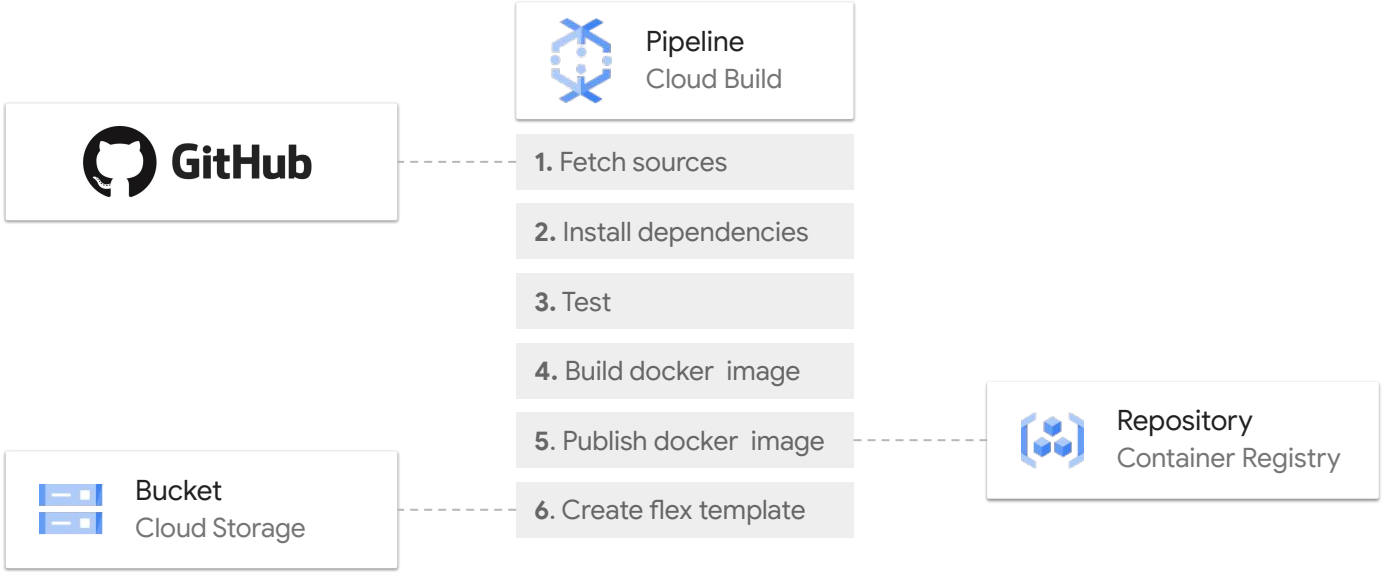
Flex Templates

Live-coding session

ARCHITECTURE



CI/CD


Pipeline
Cloud Build
GitHub
1. Fetch sources
2. Install dependencies
3. Test
4. Build docker image
5. Publish docker image
6. Create flex template
Repository
Container Registry
Bucket
Cloud Storage

Let’s get started

Log in to your Github account and create a new private repository

Log in to your Github account and create a new private repository

Set up the environment in Google Cloud

We are going to use **Terraform** to:

- Create a Google Cloud **project**.
- Enable the **services**:
  - `cloudbuild.googleapis.com`
  - `pubsub.googleapis.com`
  - `dataflow.googleapis.com`
- Create the **App Engine application** in the desired region.
- Create the **service account** and a **service account key** for the tweet publisher application that will run locally. We will export the service account key to a file in your local file system.
- Create the `tweets` **Pub/Sub topic** and assign the Pub/Sub publisher role (`roles/pubsub.publisher`) to the service account created for the tweet publisher app on it.
- Create the **service account** the Dataflow workers will act as and assign the *Pub/Sub Editor* (`roles/pubsub.editor`) role, the *Dataflow Worker* role (`roles/dataflow.worker`), *Viewer* role (`roles/viewer`), *Storage Object Admin* role (`roles/storage.objectAdmin`) to it on the project.
- Create the `trends` **Pub/Sub topic**, assign the *Pub/Sub Publisher* role (`roles/pubsub.publisher`) to the service account that the Dataflow workers will act as on it and create the subscription that will push the trends to the App Engine app.
- Create the **Cloud Storage bucket** where the Flex Template Spec will be stored.
- Create the **Cloud Build trigger** that will fetch the sources of the template from Github, test, build the docker image, publish it to GCR and create the Flex template spec file in Cloud Storage.

Create the Terraform Configuration

The steps to follow to create the required terraform configuration are:

1. Authenticate with Google Cloud

```
$ gcloud auth application-default login
```

2. Create a directory named **`terraform`** in your local file system and change to it

```
$ mkdir terraform
$ cd terraform
```

3. Create a file named **`providers.tf`** file inside the **`terraform`** directory and add the `google` and `google-beta` providers:

```
provider "provider-name" {

}
```

Hint

Create the Terraform Configuration

4. Create a file `variables.tf` and create the following variables:

NAME	DESCRIPTION	TYPE
`billing_account_id`	The identifier of the Google Cloud billing account	`string`
`parent`	The id of the Google Cloud organization (organizations/organization_id) or folder (folders/folder_id) where the project will be created	`string`
`project_id`	The id of the project that will be created	`string`
`app_engine_location`	The location off the App Engine application	`string`
`bucket_location`	The location of the bucket	`string`
`github_organization`	The name of the Github organization where the repository with the code of the flex template is	`string`
`github_repo`	The name of the Github repo where the code of the flex template is available.	`string`

Create the Terraform Configuration

```
variable "var-name" {
}
```

Hint

5. Create a file named `terraform.tfvars` inside the `terraform` directory and set the value of the variables:

```
billing_account_id=
parent=
project_id=
app_engine_location=
bucket_location=
github_organization=
github_repo=
```

HashiCorp Terraform

Create the Terraform Configuration

6. Clone this Github repository in a directory on your local file system

```
$ git clone git@github.com:terraform-google-modules/cloud-foundation-fabric.git
```

7. Copy the subdirectory `modules` inside `cloud-foundation-fabric` the to the **`terraform`** directory.

8. Create a file named **`main.tf`** inside the **`terraform`** directory. In this file we are going to specify the Google Cloud resources that will be created. To simplify this, we will be using the *Cloud Foundations Fabric* modules.

9. Add the following module to the **main.tf** file to create the project and enable the required services in it.

```
module "project" {
 source              = "./modules/project"
 billing_account     = var.billing_account_id
 name                = var.project_id
 parent              = var.parent
 auto_create_network = true
 services = [
   "cloudbuild.googleapis.com", "pubsub.googleapis.com", "dataflow.googleapis.com"
 ]
}
```

HashiCorp Terraform

Create the Terraform Configuration

10. Apply the changes.

```
$ terraform init
$ terraform apply
```

Verify in the console UI that the project has been created and the services have been enabled.

11. Add the following resource to the `main.tf` file to create the App Engine application

```
resource "google_app_engine_application" "app" {
 project     = module.project.project_id
 location_id = var.app_engine_location
}
```

12. Apply the changes.

```
$ terraform apply
```

Create the Terraform Configuration

13. Add the following module to the **main.tf** file to create the service account for the publisher app that we will be running locally.

```
module "tweet-publisher-sa" {
 source       = "./modules/iam-service-account"
 project_id   = module.project.project_id
 name         = "tweet-publisher"
 generate_key = true
}
```

Notice that the `generate_key` property is set to `true`. We will be running the tweet publisher app locally and not in Google Cloud so we will need to have the key.

14. Add the following resource to the **main.tf** file to export the service account key to a file in your file system:

```
resource "local_file" "tweet-publisher-sa-key-file" {
 content  = base64decode(module.tweet-publisher-sa.key.private_key)
 filename = "${path.module}/tweet-publisher-sa-key.json"
}
```

Create the Terraform Configuration

15. Apply the changes.

```
$ terraform init
$ terraform apply
```

Verify in the console UI that the service account has been created and that the key has been exported to a file in your local file system..

16. Add the following module to the **`main.tf`** file to create the `tweets` Pub/Sub topics and assign the `tweet-publisher` account a *Pub/Sub Publisher* role on it.

```
module "tweets-pubsub" {
 source     = "./modules/pubsub"
 project_id = module.project.project_id
 name       = "tweets"
 iam = {
   "roles/pubsub.publisher" = [
     module.tweet-publisher-sa.iam_email
   ]
 }
}
```

Create the Terraform Configuration

17. Apply the changes.

```
$ terraform init
$ terraform apply
```

18. Verify in the console UI that the topic has been created and that the `tweet-publisher account` has been assigned the right role on it.

19. Add the following module to the **`main.tf`** file to create the the service account the Dataflow workers with act as and assign the roles it requires on the project.

```
module "tweet-processor-sa" {
 source       = "./modules/iam-service-account"
 project_id   = module.project.project_id
 name         = "tweet-processor"
 generate_key = false
 iam_project_roles = {
   (module.project.project_id) = [
     "roles/dataflow.worker",
     "roles/storage.objectAdmin",
     "roles/viewer",
     "roles/pubsub.editor"
   ]
 }
}
```

Create the Terraform Configuration

20. Apply the changes.

```
$ terraform init
$ terraform apply
```

Verify in the console UI that the topic has been created and that the `tweet-processor account` has been assigned the right roles on the project.

Create the Terraform Configuration

21. Add the following module to the **`main.tf`** file to the create the `trends` Pub/Sub topic, assign the *Pub/Sub Publisher* role to the `tweet-processor` service account on it and create the subscription necessary to push the messages received in that topic to an App Engine application endpoint.

```
module "trends-pubsub" {
 source     = "./modules/pubsub"
 project_id = module.project.project_id
 name       = "trends"
 iam = {
   "roles/pubsub.publisher" = [
     module.tweet-processor-sa.iam_email
   ]
 }
 subscriptions = {
   trends-push = null
 }
 push_configs = {
   trends-push = {
     endpoint   = "https:/${google_app_engine_application.app.default_hostname}/notify"
     attributes = null
     oidc_token = null
   }
 }
```

Create the Terraform Configuration

22. Apply the changes.

```
$ terraform init
$ terraform apply
```

Verify in the console UI that the topic has been created. that the `tweet-processor` service account has the right role assigned on it and that the subscription has been created.

23. Add the following module to the **`main.tf`** file to create the bucket where the Flex Template Spec file will be created.

```
module "bucket" {
 source     = "./modules/gcs"
 project_id = module.project.project_id
 name       = module.project.project_id
 location = var.bucket_location
}
```

24. Apply the changes.

```
$ terraform init
$ terraform apply
```

Create the Terraform Configuration

Verify in the console UI that the topic has been created. that the `tweet-processor` service account has the right role assigned on it and that the subscription has been created.

25. Add the following resource to the **`main.tf`** file to the create Cloud Build trigger.

```
resource "google_cloudbuild_trigger" "trigger" {
 provider = google-beta
 project = module.project.project_id
 filename = "cloudbuild.yaml"

 github {
   owner = var.github_organization
   name = var.github_repo
   push {
     branch = ".*"
   }
 }

 substitutions = {
   _REPO_NAME = "dataflow"
   _IMAGE_NAME = "tweettrends"
   _TEMPLATE_GCS_LOCATION = "gs://${module.bucket.name}/dataflow/templates/tweettrends.json"
 }
}
```

Create the Terraform Configuration

26. Apply the changes.

```
$ terraform apply
```

Verify in the console UI that the Cloud Build trigger has been created.

Run the Publisher App

1. Create a Twitter developer account, if you don't have one already. You can apply for one here.
2. Create a Twitter developer app. You can create one here.
3. Save the following for later:
    - API key and secret.
    - Access token and secret.
4. Clone this Github repository to get the publisher app code.

```
$ git clone git@github.com:apichick/beamsummit-2021-publisher-app.git
```

5. Change to the `tweet-publisher` subdirectory inside the `beamsummit-2021-flex-template` directory:

```
$ cd beamsummit-2021-flex-template/tweet-publisher
```

Publisher App

Run the Publisher App

6. Create a file called **`config.txt`** inside that directory with the following contents:

```
[twitter]
api_key=<APP_API_KEY>
api_key_secret=<APP_API_KEY>
access_token=<ACCESS_TOKEN>
access_token_secret=<ACCESS_TOKEN)SECRET>
[pubsub]
topic=projects/<PROJECT_ID>/topics/tweets
key_file=<TWEET_PUBLISHER_SERVICE_ACCOUNT_KEY_FILE_PATH>
```

7. Install the application dependencies.

```
$ pip install -r requirements.txt
```

8. Run the application

```
$ python main.py --locations " -74,40,-73,41" --config-file config.txt
```

The option `--locations` specifies a set of bounding boxes to track (In the example above, the coordinates correspond to New York City). More details here.

Run the Consumer App

1. Clone this Github repository to get the consumer app code.

```
$ git clone git@github.com:apichick/beamsummit-2021-consumer-app.git
```

2. Change to the **beamsummit-2021-consumer-app** directory.

```
$ cd beamsummit-2021-consumer-app
```

3. Authenticate with Google Cloud

```
$ gcloud auth login
```

4. Deploy the application.

```
$ gcloud app deploy --project <PROJECT_ID>
```

5. Open the application in the browser

```
$ gcloud app browse --project <PROJECT_ID>
```

Consumer App

Develop the Pipeline

1. Create a directory named **`flex-template`** in your local file system and change to it.

```
$ mkdir flex-template
$ cd flex-template
```

2. Initialize the git repository

```
$ git init
$ cd flex-template
```

3. Add the git upstream repository

```
git remote add origin git@github.com:<GIT_ORGANIZATION>/<GIT_REPO>.git
```

Develop the Pipeline

3. Create a file called **`main.py`** in the **`flex-template`** directory with the following content.

```
import argparse

if __name__ == "__main__":
   parser = argparse.ArgumentParser()
   parser.add_argument(
       "--tweets_topic",
       help="Input Cloud Pub/Sub topic"
       "projects/PROJECT/topics/TOPIC",
   )
   parser.add_argument(
       "--trends_topic",
       help="Output Cloud Pub/Sub topic"
       "projects/PROJECT/topics/TOPIC",
   )
   args, beam_args = parser.parse_known_args()
```

Our pipeline will take two input parameters, the names of the Cloud Pub/Sub topics.

Develop the Pipeline

4. Create a package called **pipeline** and a module called **tweettrends** inside it.

```
$ mkdir pipeline
$ touch pipeline/__init__.py
$ touch pipeline/tweettrends.py
```

5. Add a the following code to that newly created module:

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

def run(tweets_topic, trends_topic, beam_args):

    opts = PipelineOptions(beam_args, save_main_session=True, streaming=True)

    with beam.Pipeline(options=options) as pipeline:
        pass
        # TODO
```

We will shortly explain how to implement the actual pipeline.

Develop the Pipeline

6. Edit the **main.tf** file to invoke the run function that we just created after the input arguments have been parsed.

```
import argparse

from pipeline import run

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--tweets_topic",
        help="Input Cloud Pub/Sub topic"
        "projects/PROJECT/topics/TOPIC",
    )
    parser.add_argument(
        "--trends_topic",
        help="Output Cloud Pub/Sub topic"
        "projects/PROJECT/topics/TOPIC",
    )
    args, beam_args = parser.parse_known_args()
    run(tweet_topic=args.tweets_topic,
        trends_topic=args.trends_topic,
        beam_args)
```

Pipeline DAG

Read from Pub/Sub

Parse JSON

Add timestamp

Apply sliding window

Extract hashtags

Transform to key-value pairs

Count per key

Get top 10 hashtags

Format

Write to Pub/Sub

Dataflow Pipeline

Pipeline transforms

Read from Pub/Sub	`p | beam.io.ReadFromPubSub(topic='projects/PROJECT_ID/topics/TOPIC')`
Parse JSON	`p | beam.Map(json.load)`
Add timestamp	`p | beam.Map(lambda item: beam.window.TimestampedValue(item, int(item['timestamp_ms']) / 1000)`
Apply sliding window	`p | beam.WindowInto(SlidingWindows(duration, period))`

Extract hashtags	`def get_tweet_hashtags(tweet):`
	`  for entity in tweet['entities']['hashtags']:`
	`  yield '#%s' % entity['text']`
	
	`p | beam.ParDo(get_tweet_hashtags)`

Dataflow Pipeline

Pipeline transforms

Transform to key-value pairs	`p | beam.io.Map(lambda hashtag: (hashtag, 1))`
Count per key	`p | beam.combiners.CountPerKey()`
Get top 10 hashtags	`p | beam.CombineGlobally(TopCombineFn(n=10, key=lambda item: item[1])).without_defaults()`

Format	`def format_message(items):`
	`  result = {}`
	`  result['metric'] = 'top10hashtags'`
	`  result['data'] = [ {'hashtag': item[0], 'ocurrences': item[1]} for item in items]`
	`  yield json.dumps(result).encode('utf-8')`
	
	`p | beam.ParDo(format_message)`

Write to Pub/Sub	`p | beam.io.WriteToPubSub(topic='projects/PROJECT_ID/topics/TOPIC')`

Dataflow Pipeline

Develop the Pipeline

7. Create a **`setup.py`** file in the **`flex-template`** directory so Dataflow can install it as a package

```
import setuptools

setuptools.setup(
   name="tweet-trends-flextemplate",
   version="0.0.1",
   description="A flex template that reads tweets from a Pub/Sub topic and writes tweet trends in
another one",
   packages=setuptools.find_packages(),
)
```

Test your Pipeline

1. Create a file named **`requirements_test.txt`** in the **`flex-template`** directory for the dependencies required to run tests of the pipeline.

```
apache-beam[gcp]==2.31.0
pytest==6.2.4
```

We will use `pytest` to run the tests.

2. Create a file named **`pytest.ini`** in the **`flex-template`** directory for the pytest configuration.

```
[pytest]
addopts = --disable-pytest-warnings
```

With those setting `pytest` will not write warnings to the output.

3. Create a file named **`test_pipeline.py`** in the **`flex-template`** directory. You will be writing your tests here.

4. Run the tests using the following command.

```
$ python -m pytest
```

In order to test an individual transform create a function called `test_*` in the **`test_pipeline.py`** file looking like this:

```
import apache_beam as beam
from apache_beam.testing.test_pipeline import TestPipeline
from apache_beam.testing.util import assert_that, equal_to

def test_transform():

    with TestPipeline() as p:

        INPUT_ITEMS = ...
        EXPECTED_OUTPUT_ITEMS = ...
        input = p | beam.Create(INPUT_ITEMS)
        output = input | <TRANSFORM>
        assert_that(output, equal_to(EXPECTED_OUTPUT_ITEMS), label='<ASSERTION LABEL>')
```

Turn your Pipeline into a Flex Template

1. Create a **`metadata.json`** file inside the **`flex-template`** directory with the name and the parameters that your template will be taking.

```
{
   "name": "tweettrends-flex-template",
   "description": "Tweet trends flex template",
   "parameters": [
     {
       "name": "tweets_topic",
       "label": "Input Cloud Pub/Sub topic.",
       "helpText": "Name of Cloud Pub/Sub topic where the tweets are published",
       "regexes": [
         "projects/[^/]+/topics/[a-zA-Z][-_.~+%a-zA-Z0-9]{2,}"
       ]
     },
     {
       "name": "trends_topic",
       "label": "Output Cloud Pub/Sub topic.",
       "helpText": "Name of the Cloud Pub/Sub topic where the twitter trends are published",
       "isOptional": true,
       "regexes": [
         "projects/[^/]+/topics/[a-zA-Z][-_.~+%a-zA-Z0-9]{2,}"
       ]
     }
   ]
 }
```

Dataflow Flex Template

Turn your Pipeline into a Flex Template

2. Create a **`Dockerfile`** file inside the **`flex-template`** with the following content.

```
FROM gcr.io/dataflow-templates-base/python3-template-launcher-base

ARG WORKDIR=/template
RUN mkdir -p ${WORKDIR}
WORKDIR ${WORKDIR}

ENV FLEX_TEMPLATE_PYTHON_REQUIREMENTS_FILE="${WORKDIR}/requirements.txt"
ENV FLEX_TEMPLATE_PYTHON_PY_FILE="/${WORKDIR}/main.py"

COPY pipeline ./pipeline
COPY main.py .
COPY setup.py .
COPY requirements.txt .

# We could get rid of installing libffi-dev and git, or we could leave them.
RUN apt-get update \
   && apt-get install -y libffi-dev git \
   && rm -rf /var/lib/apt/lists/* \
   # Upgrade pip and install the requirements.
   && pip install --no-cache-dir --upgrade pip \
   && pip install --no-cache-dir -r $FLEX_TEMPLATE_PYTHON_REQUIREMENTS_FILE \
   # Download the requirements to speed up launching the Dataflow job.
   && pip download --no-cache-dir --dest /tmp/dataflow-requirements-cache -r $FLEX_TEMPLATE_PYTHON_REQUIREMENTS_FILE

# Since we already downloaded all the dependencies, there's no need to rebuild everything.
ENV PIP_NO_DEPS=True
```

Create the CI/CD pipeline

1. Create a file called **`cloudbuild.yaml`** inside the **`flex-template`** directory with the following contents:

```
steps:
 # Install dependencies
 - name: python
   entrypoint: pip
   args: ["install", "-r", "requirements_test.txt", "--user"]
 # Run tests
 - name: python
   entrypoint: python
   args: ["-m", "pytest"]
 # Build docker image
 - name: 'gcr.io/cloud-builders/docker'
   args: ['build', '-t', 'gcr.io/${PROJECT_ID}/dataflow/templates/${_IMAGE_NAME}:latest', '.']
 # Push docker image to GCR
 - name: 'gcr.io/cloud-builders/docker'
   args: ['push',  'gcr.io/${PROJECT_ID}/dataflow/templates/${_IMAGE_NAME}:latest']
 # Build dataflow template
 - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
   entrypoint: 'gcloud'
   args: [ 'dataflow', 'flex-template', 'build', '${_TEMPLATE_GCS_LOCATION}', '--image',
'gcr.io/${PROJECT_ID}/dataflow/templates/${_IMAGE_NAME}:latest', '--sdk-language', 'PYTHON',
'--metadata-file', 'metadata.json']
```

Trigger the CI/CD pipeline

1. Add and commit all the changes that you have made to the flex-template directory.

```
$ git add.
$ git commit -m "Initial import"
```

2. Push the code to the remote.

```
$ git push origin main
```

Pushing the `main` branch to the remote will trigger the execution of the pipeline in Cloud Build.

Run the pipeline

1. Run the pipeline using the template:

```
$ PROJECT_ID=<PROJECT_ID> gcloud dataflow flex-template run "tweettrends-`date +%Y%m%d-%H%M%S`" \
    --template-file-gcs-location "gs://${PROJECT_ID}/dataflow/templates/tweet_trends.json" \
    --parameters tweets_topic="projects/${PROJECT_ID}/topics/tweets" \
    --parameters trends_topic="projects/${PROJECT_ID}/topics/trends" \
    --service-account-email="tweet-processor@${PROJECT_ID}.iam.gserviceaccount.com" \
    --region "${REGION}" \
    --project "${PROJECT_ID}
```

2. Verify in the console that the job is streaming.

3. Head to the browser to see how the top 10 hashtags change.

Dataflow Flex Template

Q&A

Thanks for joining!