THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2019 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS

CLASS NEWS

CLASS NEWS

- Office hours
  - Thursday @11AM
  - Hannah Sunday

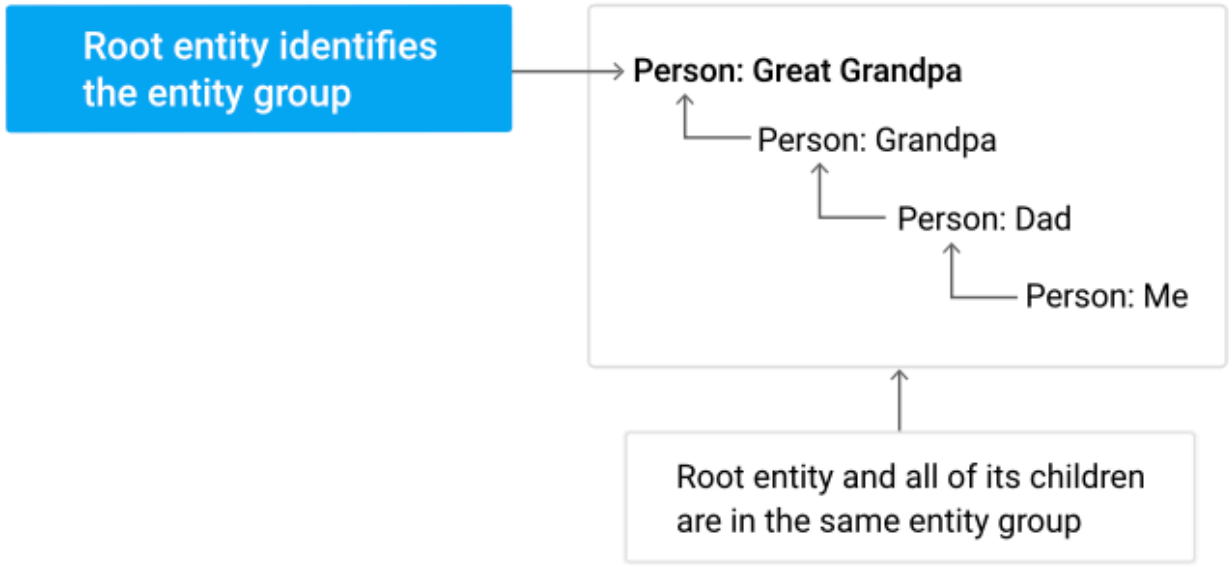
PARENTS JUST DON'T UNDERSTAND

PARENT JUST DON'T UNDERSTAND

- The Parent key in Datastore
- https://cloud.google.com/appengine/docs/standard/python/datastore/entities

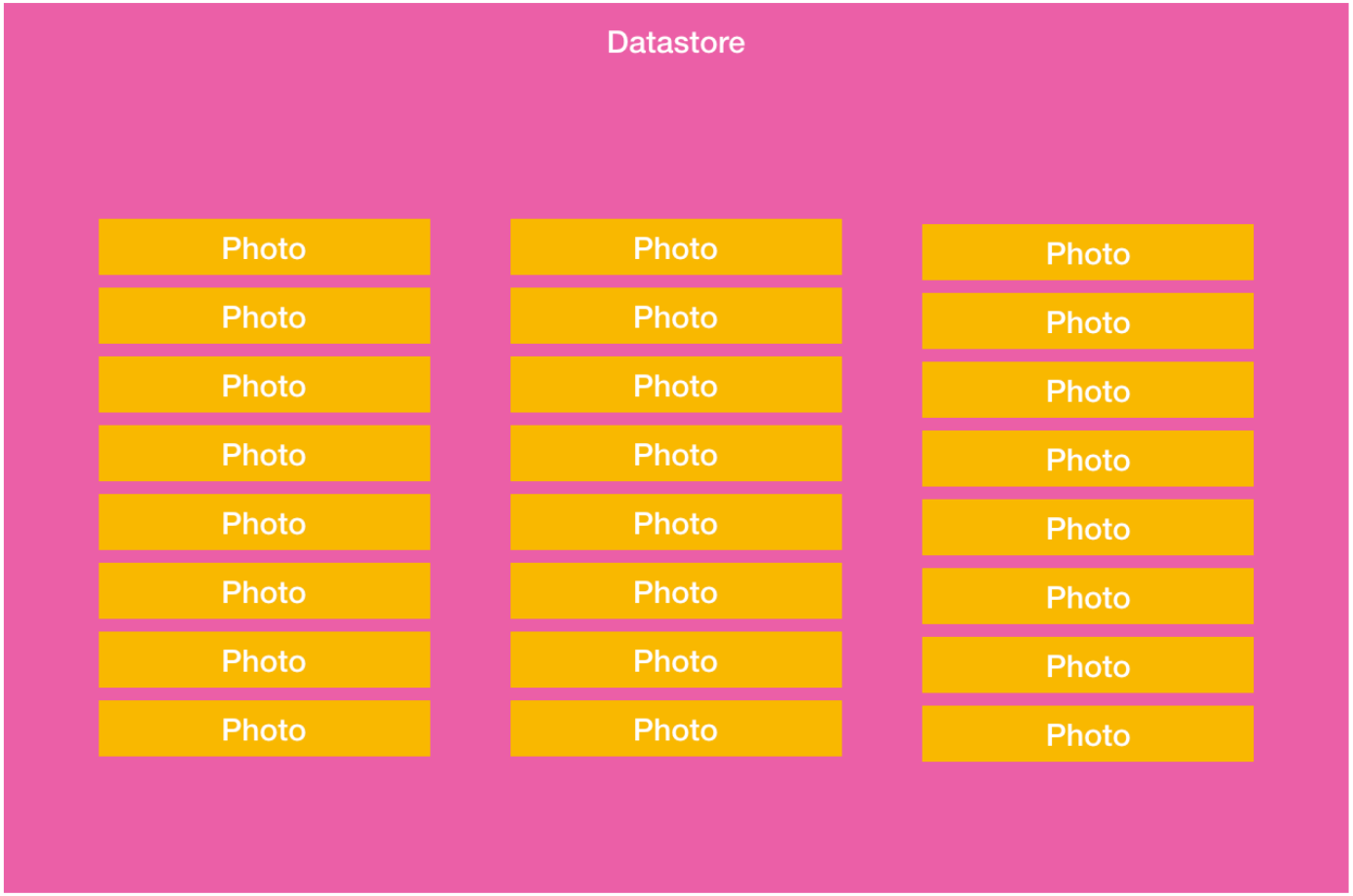
PARENT JUST DON'T UNDERSTAND

- Parent child relationship form a hierarchical structure



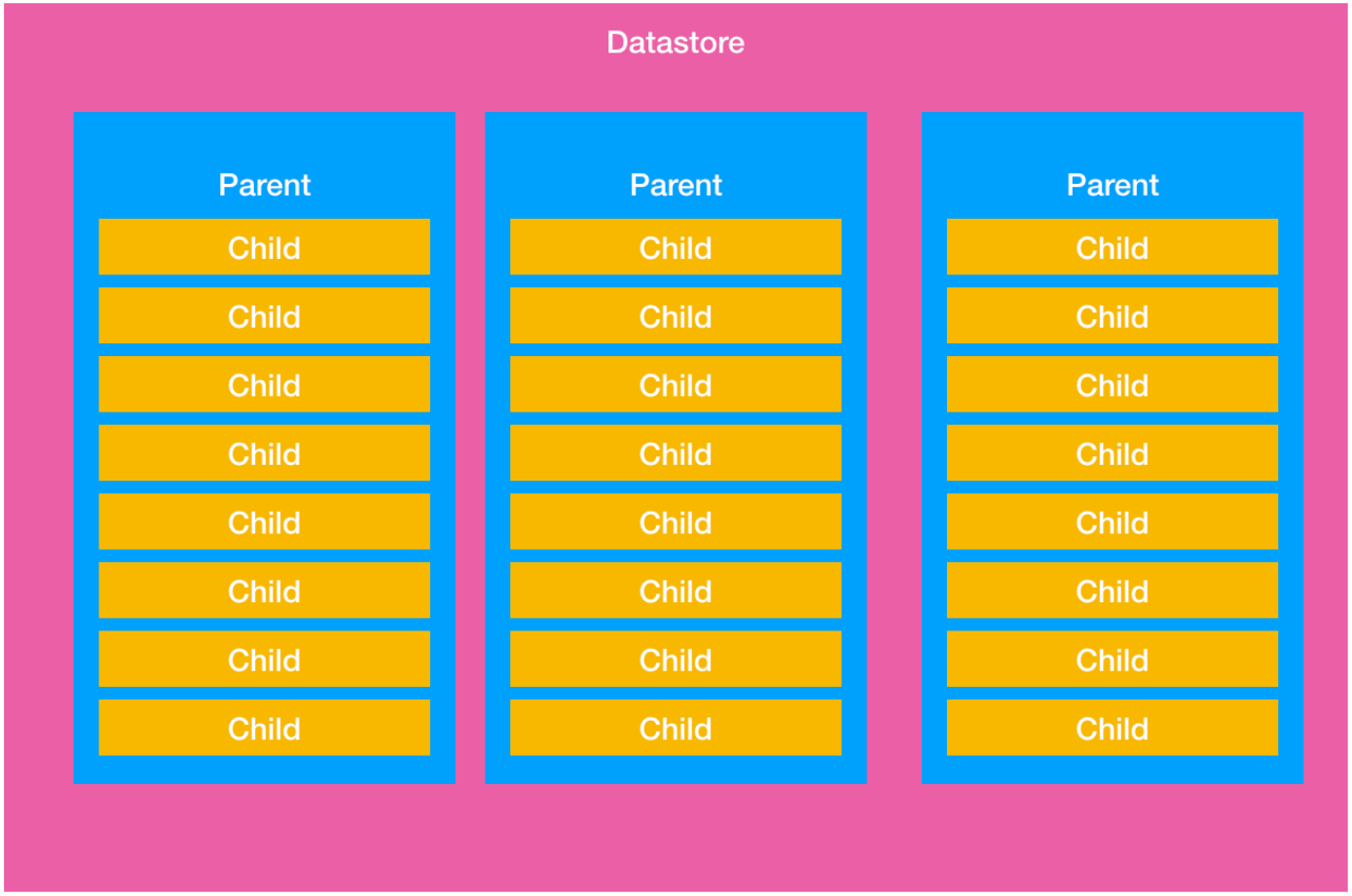
PARENT JUST DON'T UNDERSTAND

- Datastore is NOT a relational database
- To query all photos by a user would require to go through all entities



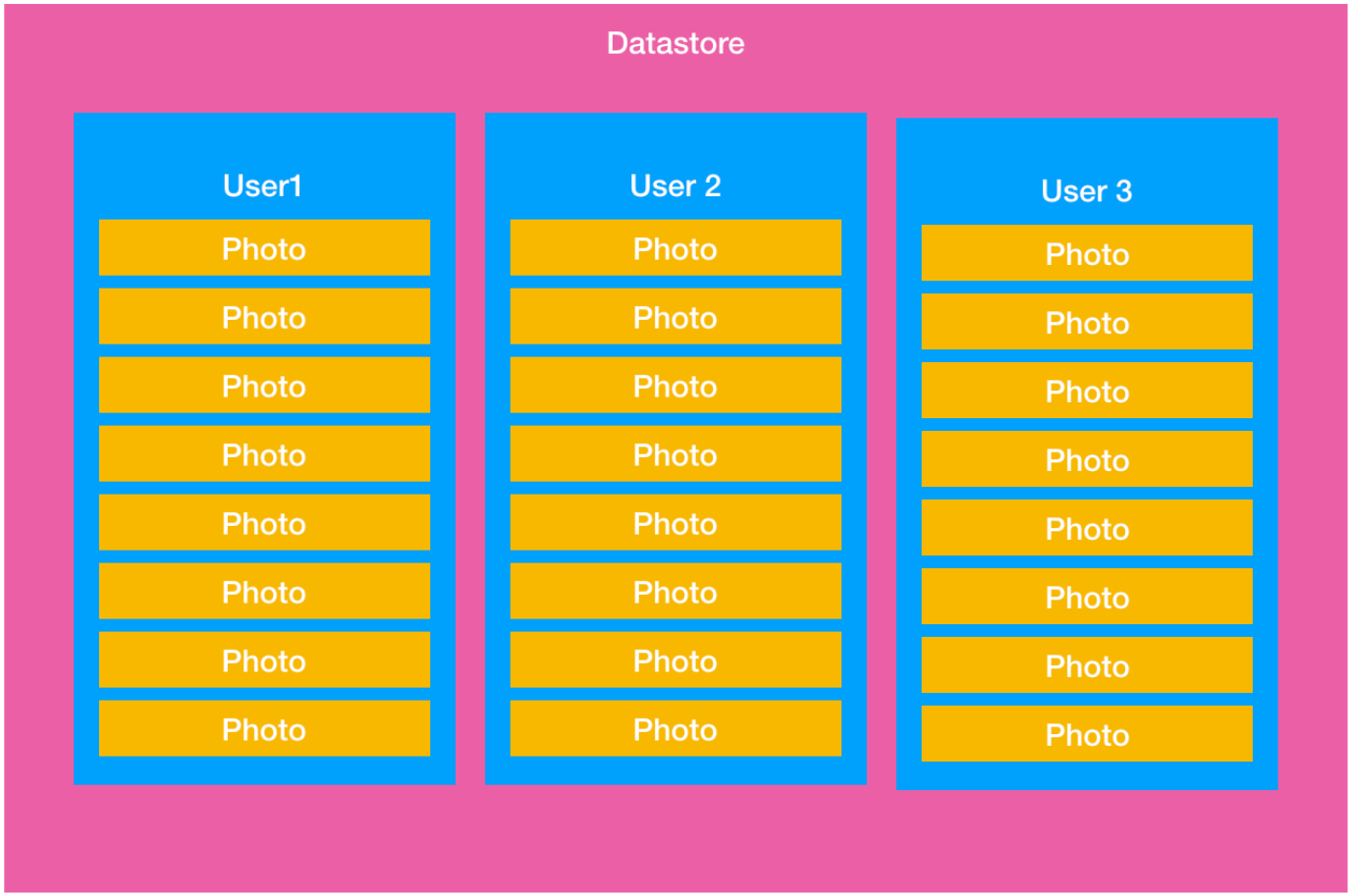
PARENT JUST DON'T UNDERSTAND

- Parent/child relationship can help organize data
- Child has a reference to parent



PARENT JUST DON'T UNDERSTAND

- Query for all a users photos is a subset of data
- The design of your entity models and their relationship enforces this



PARENT JUST DON'T UNDERSTAND

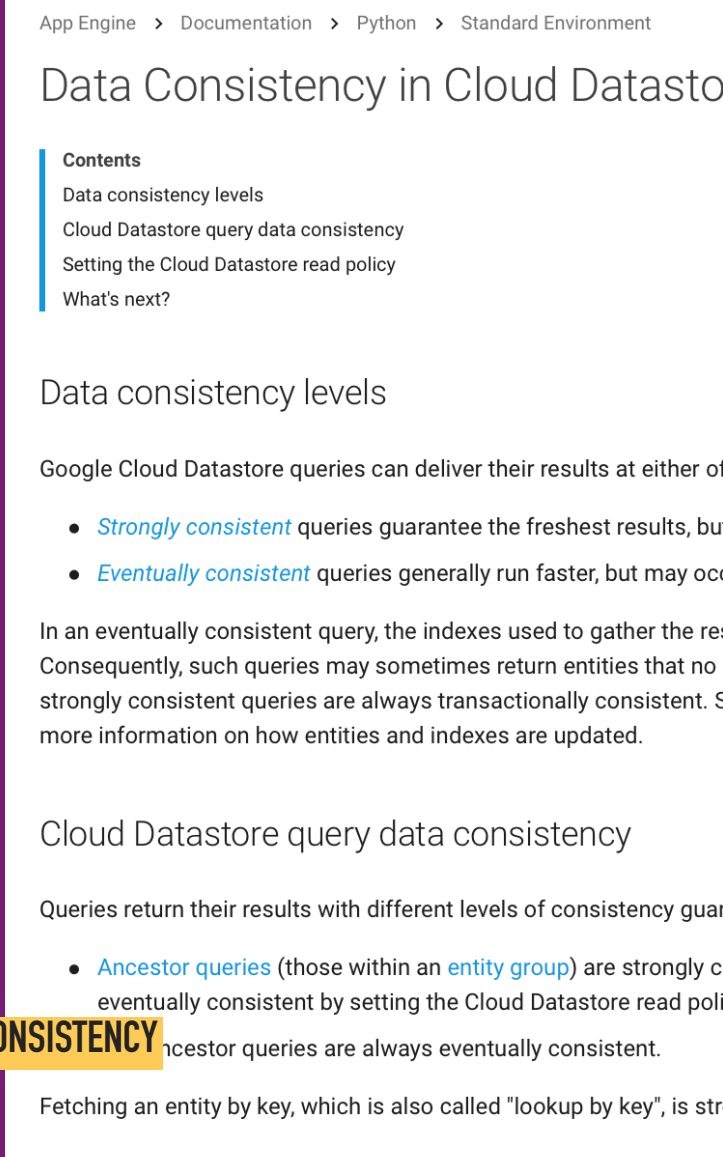
- Ancestor queries
- https://cloud.google.com/appengine/docs/standard/python/datastore/queries

CONSISTENCY

CONSISTENCY

- Data consistency
  - **Strongly consistent** queries guarantee the freshest results, but may take longer to complete
  - **Eventually consistent** queries generally run faster, but may occasionally return stale results

HTTPS://CLOUD.GOOGLE.COM/APPENGINE/DOCS/STANDARD/PYTHON/DATASTORE/DATA-CONSISTENCY



App Engine › Documentation › Python › Standard Environment

Data Consistency in Cloud Datasto

Contents

Data consistency levels
Cloud Datastore query data consistency
Setting the Cloud Datastore read policy
What's next?

Data consistency levels

Google Cloud Datastore queries can deliver their results at either of

- *Strongly consistent* queries guarantee the freshest results, bu
- *Eventually consistent* queries generally run faster, but may oc

In an eventually consistent query, the indexes used to gather the re
Consequently, such queries may sometimes return entities that no
strongly consistent queries are always transactionally consistent. S
more information on how entities and indexes are updated.

Cloud Datastore query data consistency

Queries return their results with different levels of consistency guar

- Ancestor queries (those within an entity group) are strongly c
  eventually consistent by setting the Cloud Datastore read poli

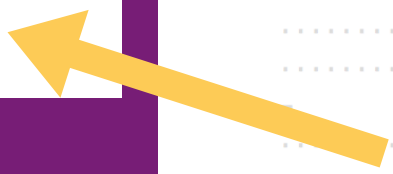
ncestor queries are always eventually consistent.

Fetching an entity by key, which is also called "lookup by key", is str

CONSISTENCY

- Add a new photo to datastore
- Immediately return the users stream via JSON

```python
def json_results(self,photos):
    """Return formatted json from the datastore query"""
    json_array = []
    for photo in photos:
        dict = {}
        dict['image_url'] = "image/%s/" % photo.key.urlsafe()
        dict['caption'] = photo.caption
        dict['user'] = photo.user
        dict['date'] = str(photo.date)
        json_array.append(dict)
    return json.dumps({'results' : json_array})
```



```python
class PostHandler(webapp2.RequestHandler):
    def post(self,user):

        # If we are submitting from the web form, we will be passing
        # the user from the textbox.  If the post is coming from the
        # API then the username will be embedded in the URL
        if self.request.get('user'):
            user = self.request.get('user')

        # Be nice to our quotas
        thumbnail = images.resize(self.request.get('image'), 30,30)

        # Create and add a new Photo entity
        #
        # We set a parent key on the 'Photos' to ensure that they are all
        # in the same entity group. Queries across the single entity group
        # will be consistent. However, the write rate should be limited to
        # ~1/second.
        photo = Photo(parent=ndb.Key("User", user),
                      user=user,
                      caption=self.request.get('caption'),
                      image=thumbnail)
        photo.put()

        # Clear the cache (the cached version is going to be outdated)
        key = user + "_photos"
        memcache.delete(key)

        # Redirect to print out JSON
        self.redirect('/user/%s/json/' % user)
```

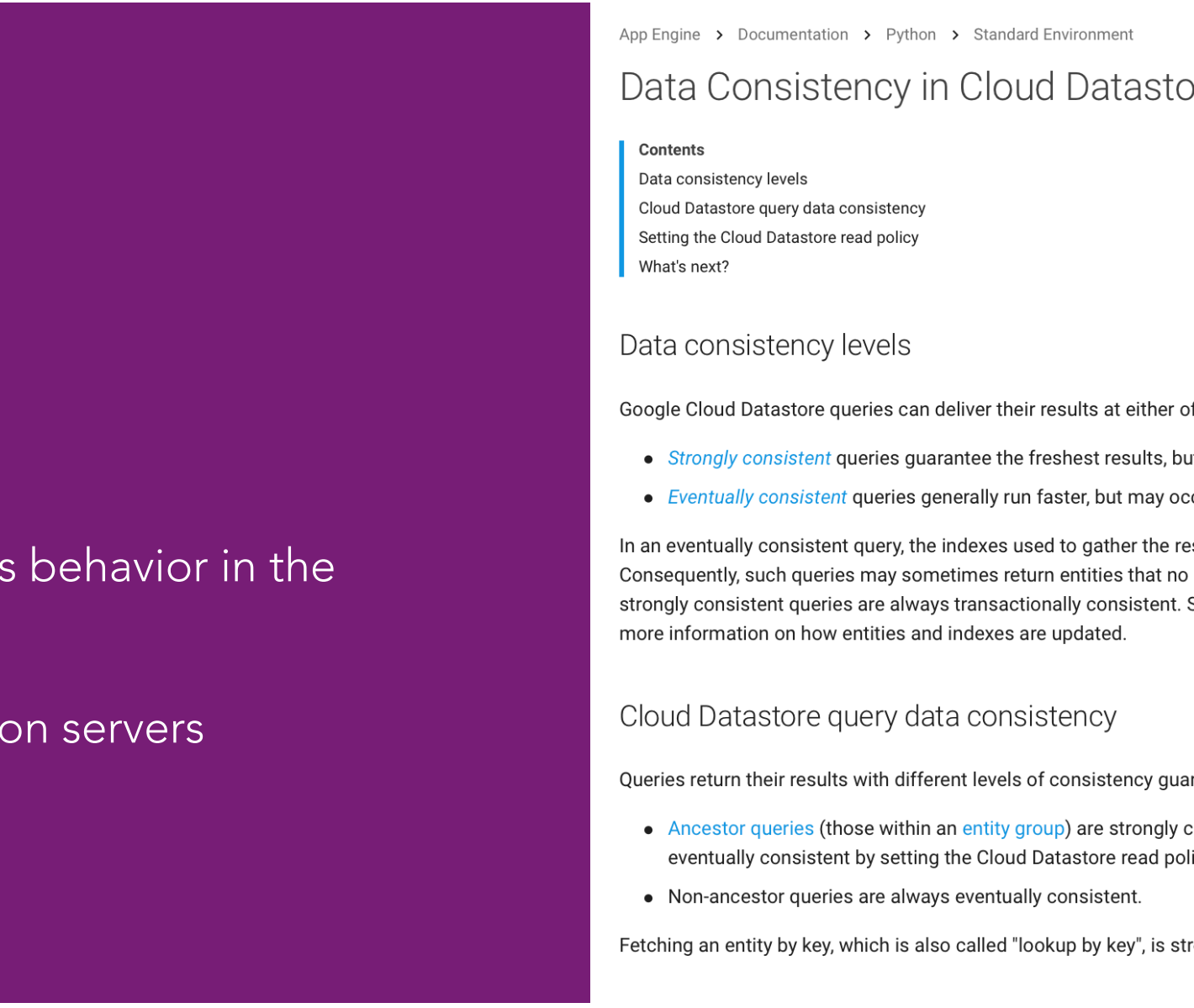
CONSISTENCY

- Without common ancestor
  - The new photo will NOT be listed in the JSON
  - Depends on locations (ie only reads the data on the machine the query was executed on)
- With common ancestor
  - The photo WILL be in the JSON

```
class PostHandler(webapp2.RequestHandler):
    def post(self,user):

        # If we are submitting from the web fo
        # the user from the textbox.  If the p
        # API then the username will be embedd
        if self.request.get('user'):
            user = self.request.get('user')

        # Be nice to our quotas
        thumbnail = images.resize(self.request

        # Create and add a new Photo entity
        #
        # We set a parent key on the 'Photos'
        # in the same entity group. Queries ac
        # will be consistent. However, the wri
        # ~1/second.
        photo = Photo(parent=ndb.Key("User", u
                      user=user,
                      caption=self.request.get('capt
                      image=thumbnail)
        photo.put()

        # Clear the cache (the cached version
        key = user + "_photos"
        memcache.delete(key)

        # Redirect to print out JSON
        self.redirect('/user/%s/json/' % user)
```

CONSISTENCY

- Warning
  - You will never see this behavior in the development server
  - Only on the production servers



App Engine › Documentation › Python › Standard Environment

Data Consistency in Cloud Datasto

Contents

Data consistency levels
Cloud Datastore query data consistency
Setting the Cloud Datastore read policy
What's next?

Data consistency levels

Google Cloud Datastore queries can deliver their results at either of

- *Strongly consistent* queries guarantee the freshest results, bu
- *Eventually consistent* queries generally run faster, but may oc

In an eventually consistent query, the indexes used to gather the re
Consequently, such queries may sometimes return entities that no
strongly consistent queries are always transactionally consistent. S
more information on how entities and indexes are updated.

Cloud Datastore query data consistency

Queries return their results with different levels of consistency guar

- Ancestor queries (those within an entity group) are strongly c
  eventually consistent by setting the Cloud Datastore read poli
- Non-ancestor queries are always eventually consistent.

Fetching an entity by key, which is also called "lookup by key", is str

CONSISTENCY

```
for i in `seq 1 100`; do

curl -X POST -H "Content-Type: multipart/form-data" -F
caption='curl' -F id_token=123 -F "image=@DSC_0665.jpg"
http://localhost:8080/post/poobyj/

done
```

CONSISTENCY

- Writing to a single entity group
  - Consistency
  - Limits changes to the guestbook to no more than 1 write per second (the supported limit for entity groups)

Structuring Data for Strong Consistency

> ★ **Note:** Developers building new applications are **strongly encouraged** to use the NDB Client Library, which
> benefits compared to this client library, such as automatic entity caching via the Memcache API. If you a
> using the older DB Client Library, read the DB to NDB Migration Guide

Google Cloud Datastore provides high availability, scalability and durability by distributing data over man
using masterless, synchronous replication over a wide geographic area. However, there is a tradeoff in th
is that the write throughput for any single *entity group* is limited to about one commit per second, and th
limitations on queries or transactions that span multiple entity groups. This page describes these limitat
detail and discusses best practices for structuring your data to support strong consistency while still me
application's write throughput requirements.

Strongly-consistent reads always return current data, and, if performed within a transaction, will appear t
single, consistent snapshot. However, queries must specify an ancestor filter in order to be strongly-cons
participate in a transaction, and transactions can involve at most 25 entity groups. Eventually-consistent
have those limitations, and are adequate in many cases. Using eventually-consistent reads can allow you
your data among a larger number of entity groups, enabling you to obtain greater write throughput by exe
in parallel on the different entity groups. But, you need to understand the characteristics of eventually-co
order to determine whether they are suitable for your application:

- The results from these reads might not reflect the latest transactions. This can occur because thes
  ensure that the replica they are running on is up-to-date. Instead, they use whatever data is availabl
  at the time of query execution. Replication latency is almost always less than a few seconds.
- A committed transaction that spanned multiple entities might appear to have been applied to some
  and not others. Note, though, that a transaction will never appear to have been partially applied wit
  entity.
- The query results can include entities that should not have been included according to the filter crit
  exclude entities that should have been included. This can occur because indexes might be read at
  version than the entity itself is read at.

To understand how to structure your data for strong consistency, compare two different approaches fro
Guestbook tutorial exercise. The first approach creates a new root entity for each greeting:

CONSISTENCY

- Writing to a single entity group
  - Consistency
  - Limits changes to the guestbook to no more than 1 write per second (the supported limit for entity groups)

Structuring Data for Strong Consistency

★ **Note:** Developers building new applications are **strongly encouraged** to use the NDB Client Library, which benefits compared to this client library, such as automatic entity caching via the Memcache API. If you a using the older DB Client Library, read the DB to NDB Migration Guide

Google Cloud Datastore provides high availability, scalability and durability by distributing data over man using masterless, synchronous replication over a wide geographic area. However, there is a tradeoff in th is that the write throughput for any single *entity group* is limited to about one commit per second, and th limitations on queries or transactions that span multiple entity groups. This page describes these limitat detail and discusses best practices for structuring your data to support strong consistency while still me application's write throughput requirements.

Strongly-consistent re erformed within a transaction, will appear t single, consistent sna n ancestor filter in order to be strongly-cons participate in a transa ost 25 entity groups. Eventually-consistent have those limitation g eventually-consistent reads can allow you your data among a la u to obtain greater write throughput by exe in parallel on the diffe erstand the characteristics of eventually-co order to determin ion:

USE MEMEACHE TO STORE PENDING WRITES

- The result transactions. This can occur because thes ensure that the Instead, they use whatever data is availabl at the time of q ost always less than a few seconds.
- A committed tr might appear to have been applied to som and not others. r appear to have been partially applied wit entity.
- The query result ave been included according to the filter cri exclude entities that should have been included. This can occur because indexes might be read at version than the entity itself is read at.

To understand how to structure your data for strong consistency, compare two different approaches fro Guestbook tutorial exercise. The first approach creates a new root entity for each greeting:

CLOUD TASKS

CLOUD TASKS

- Cloud Tasks API lets applications perform work asynchronously outside of a user request
  - Longer run time
  - Not user intiatied

Cloud Tasks > Documentation

SEND FEEDBACK

Quickstart for Cloud Tasks queues

Contents

Before you begin
Set up the sample
Create a Cloud Tasks queue
Add a task to the Cloud Tasks queue
Clean up
What's next

The following instructions allow you to try out basic operations using Cloud Tasks queues via Cloud Tasks API :

1. Before you begin (set up your environment)
2. Set up the sample code
3. Create a queue
4. Create a task and add it to the queue
5. Clean up your resources

Before you begin

To set up your Cloud environment, create a GCP project and add an App Engine application with billing enabled. You must have an App Engine application in your project to run your queue. For more information on GCP projects, App Engine

CLOUD TASKS

- Dispatch requests at a reliable, steady rate
- They guarantee reliable task execution
- You can control the workers' scaling behavior (and hence your costs)
- Tasks handled by automatic scaling services must finish in ten minutes
- Tasks handled by basic and manual scaling services can run for up to 24 hours

☆ ☆ ☆ ☆ ☆

SEND FEEDBACK

Quickstart for Cloud Tasks queues

Contents

Before you begin
Set up the sample
Create a Cloud Tasks queue
Add a task to the Cloud Tasks queue
Clean up
What's next

The following instructions allow you to try out basic operations using Cloud Tasks queues via Cloud Tasks API :

1. Before you begin (set up your environment)
2. Set up the sample code
3. Create a queue
4. Create a task and add it to the queue
5. Clean up your resources

Before you begin

To set up your Cloud environment, create a GCP project and add an App Engine application with billing enabled. You must have an App Engine application in your project to run your queue. For more information on GCP projects, App Engine applications, and billing in general, see here.

CLOUD TASKS

- Requests are delivered at a constant rate
- If a task fails, the service will retry the task, sending another request
- An HTTP response code between 200–299 indicates success
  - All other values indicate the task failed

Cloud Tasks > Documentation

SEND FEEDBACK

Quickstart for Cloud Tasks queues

Contents

Before you begin
Set up the sample
Create a Cloud Tasks queue
Add a task to the Cloud Tasks queue
Clean up
What's next

The following instructions allow you to try out basic operations using Cloud Tasks queues via Cloud Tasks API :

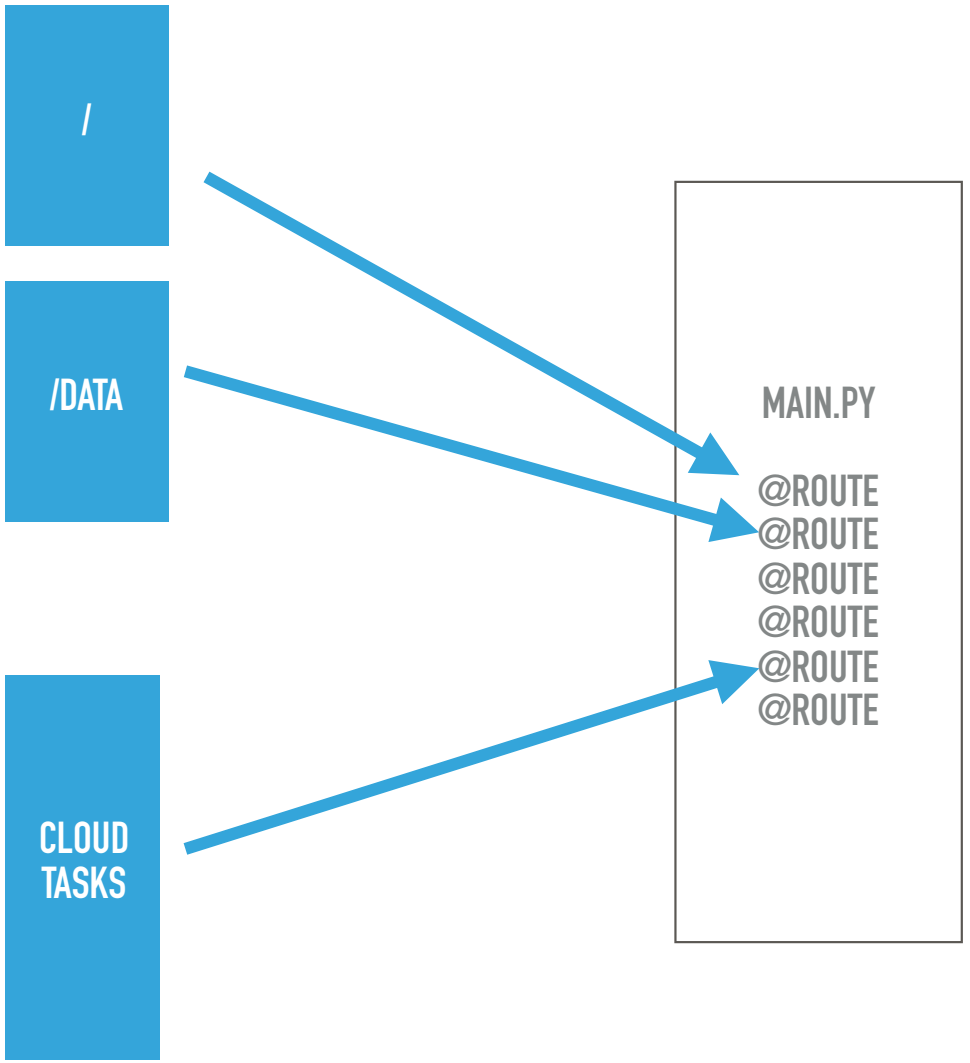
1. Before you begin (set up your environment)
2. Set up the sample code
3. Create a queue
4. Create a task and add it to the queue
5. Clean up your resources

Before you begin

To set up your Cloud environment, create a GCP project and add an App Engine application with billing enabled. You must have an App Engine application in your project to run your queue. For more information on GCP projects, App Engine

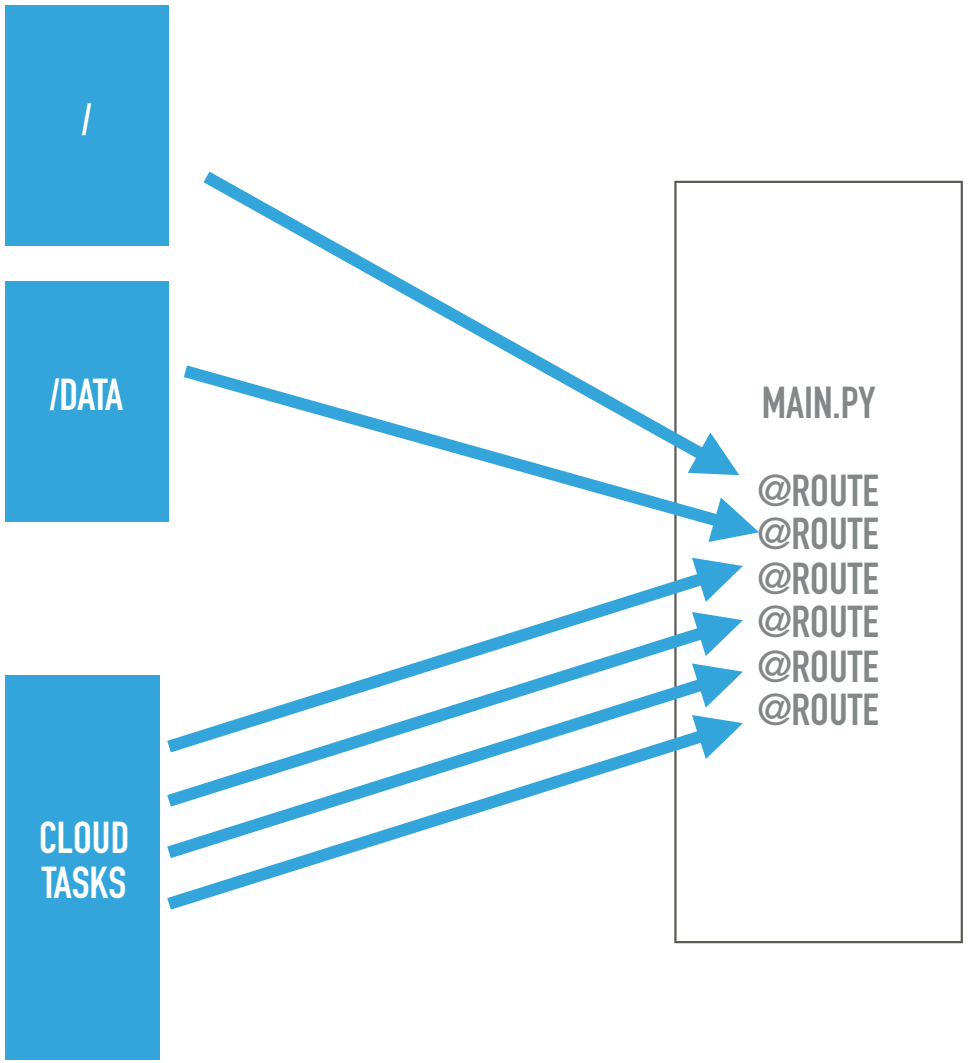
CLOUD TASKS

- Define a queue
- Write a handler to process a task request
- Create tasks and add them to the queue



CLOUD TASKS

- A single service can have multiple handlers for different kinds of tasks,
  - You you can use different services for different task types



SETTING UP A TASK

CLOUD TASKS

- Add API to project

```
Flask==1.0.2
google-cloud-datastore==1.7.3
google-cloud-storage==1.13.2
google-cloud-vision==0.35.2
google-cloud-tasks==1.2.1
```

PIP INSTALL -R REQUIREMENTS.TXT

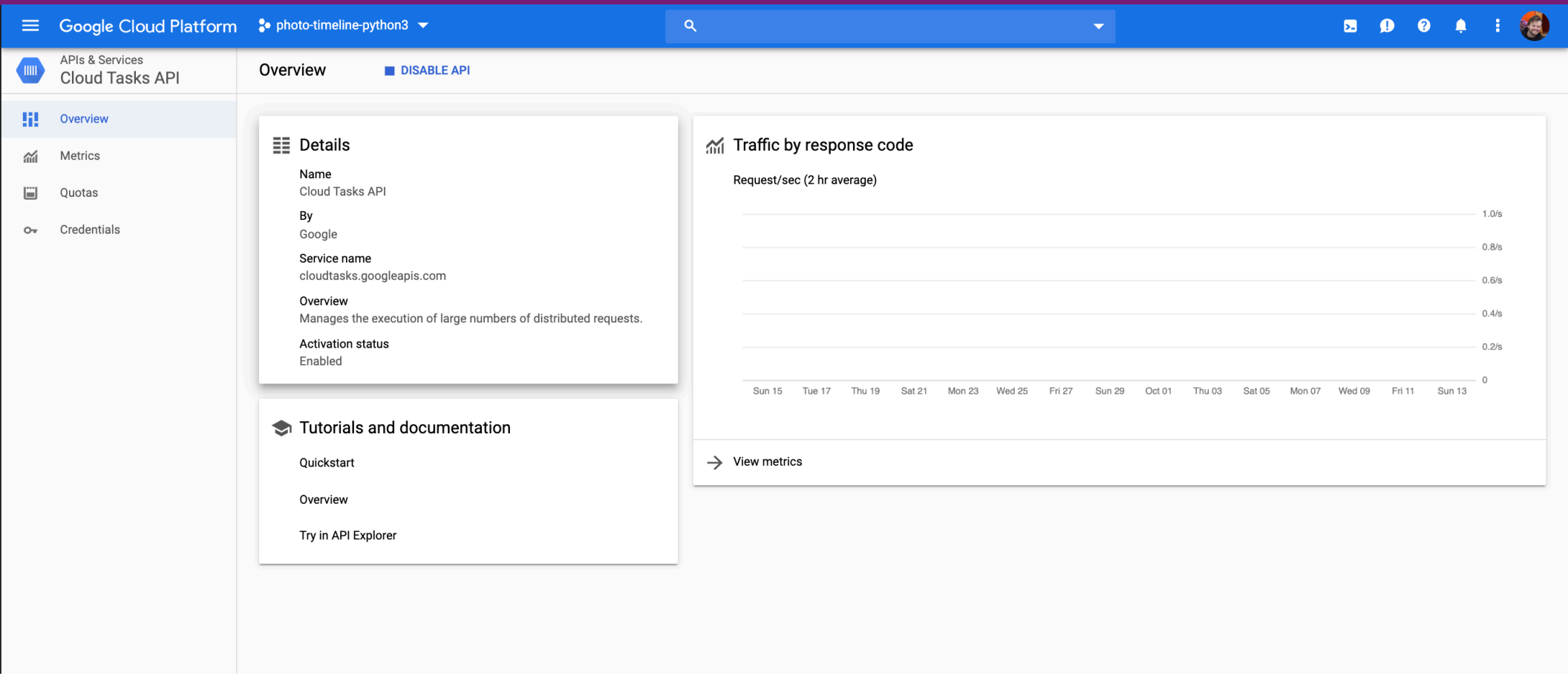
CLOUD TASKS

- Enable Cloud Tasks API

```
572 % gcloud services  enable cloudtasks.googleapis.com
```

GCLOUD SERVICES ENABLE CLOUDTASKS.GOOGLEAPIS.COM

CLOUD TASKS


Google Cloud Platform
photo-timeline-python3
APIs & Services
Cloud Tasks API
Overview
Metrics
Quotas
Credentials
Overview
DISABLE API
Details
Name
Cloud Tasks API
By
Google
Service name
cloudtasks.googleapis.com
Overview
Manages the execution of large numbers of distributed requests.
Activation status
Enabled
Tutorials and documentation
Quickstart
Overview
Try in API Explorer
Traffic by response code
Request/sec (2 hr average)
1.0/s
0.8/s
0.6/s
0.4/s
0.2/s
0
Sun 15
Tue 17
Thu 19
Sat 21
Mon 23
Wed 25
Fri 27
Sun 29
Oct 01
Thu 03
Sat 05
Mon 07
Wed 09
Fri 11
Sun 13
View metrics

CLOUD TASKS

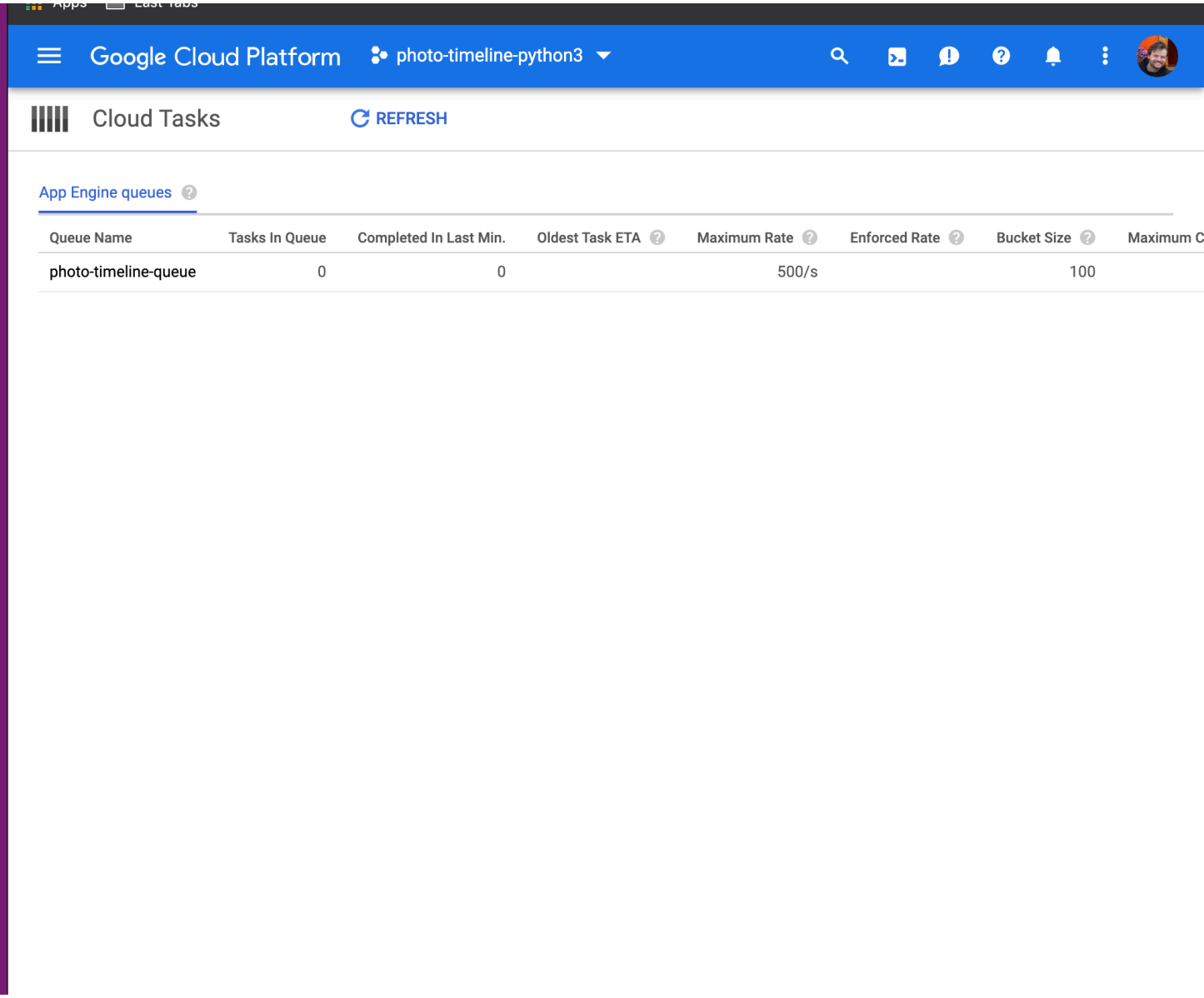
- Create a task queue

```
tabinkowski:mpcs51033-2019-autumn-code/photo-timelime-python3 (master)
500 % gcloud tasks queues create photo-timeline-queue
API [cloudtasks.googleapis.com] not enabled on project [742401181957].
 Would you like to enable and retry (this will take a few minutes)?
(y/N)?  y

Enabling service [cloudtasks.googleapis.com] on project [742401181957]...
Operation "operations/acf.feee48fe-97dc-4a6a-ba41-4a6fff18cf72" finished s
WARNING: You are managing queues with gcloud, do not use queue.yaml or que
More details at: https://cloud.google.com/tasks/docs/queue-yaml.
Created queue [photo-timeline-queue].
```

GCLOUD TASKS QUEUES CREATE [QUEUE_ID]

CLOUD TASKS

- Configuring your queue
- https://cloud.google.com/tasks/docs/creating-queues



CLOUD TASKS

- Describe queue

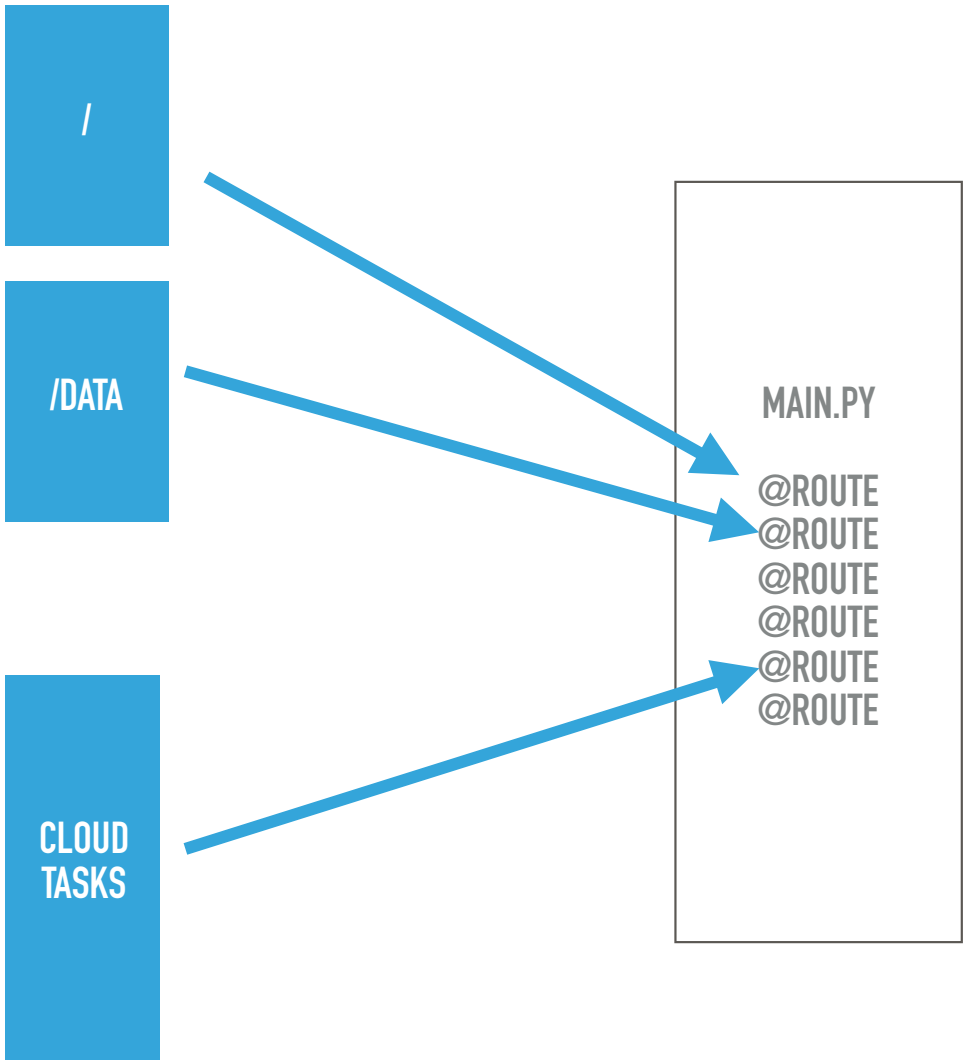
```
578 % gcloud tasks queues describe photo-timeline-queue
name: projects/photo-timeline-python3/locations/us-central1/queues/p
hoto-timeline-queue
rateLimits:
  maxBurstSize: 100
  maxConcurrentDispatches: 1000
  maxDispatchesPerSecond: 500.0
retryConfig:
  maxAttempts: 100
  maxBackoff: 3600s
  maxDoublings: 16
  minBackoff: 0.100s
state: RUNNING
```

GCLOUD TASKS QUEUES DESCRIBE PHOTO-TIMELINE-QUEUE

TASK HANDLER

CLOUD TASKS

- Task make a request to a route
- Pass an encoded payload



CLOUD TASKS

- main.py

```
# Begin Flask app
app = Flask(__name__)

@app.route('/')
def homepage():…

@app.route('/data')
def data():…

@app.route('/upload_photo', methods=['GET', 'POST'])
def upload_photo():…

# Cloud task handler
@app.route('/example_task_handler', methods=['POST'])
def example_task_handler():…

@app.route('/cron_task_handler')
def cron_task_handler():…
```

CLOUD TASKS

- main.py

```python
# Cloud task handler
@app.route('/example_task_handler', methods=['POST'])
def example_task_handler():
    """Log the request payload."""
    payload = request.get_data(as_text=True) or '(empty payload)'
    print('Received task with payload: {}'.format(payload))

    # Create a Task entry in datastore (just for illustrative purposes)
    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cloud task"
    datastore_client.put(entity)

    return 'Printed task payload: {}'.format(payload)
```

CLOUD TASKS

- main.py

```python
# Cloud task handler
@app.route('/example_task_handler', methods=['POST'])
def example_task_handler():
    """Log the request payload."""
    payload = request.get_data(as_text=True) or '(empty payload)'
    print('Received task with payload: {}'.format(payload))

    # Create a Task entry in datastore (just for illustrative purposes)
    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cloud task"
    datastore_client.put(entity)

    return 'Printed task payload: {}'.format(payload)
```

CLOUD TASKS

- main.py

```python
# Cloud task handler
@app.route('/example_task_handler', methods=['POST'])
def example_task_handler():
    """Log the request payload."""
    payload = request.get_data(as_text=True) or '(empty payload)'
    print('Received task with payload: {}'.format(payload))

    # Create a Task entry in datastore (just for illustrative purposes)
    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cloud task"
    datastore_client.put(entity)

    return 'Printed task payload: {}'.format(payload)
```

CREATE A TASK

CLOUD TASKS

- Create a task
  - Posts a request to your App Engine app that will run on a different service

```
import datetime

from google.cloud import tasks_v2
from google.protobuf import timestamp_pb2

# Create a client.
client = tasks_v2.CloudTasksClient()

project = 'photo-timeline-python3'
queue = 'photo-timeline-queue'
location = 'us-central1'

# Construct the fully qualified queue name.
parent = client.queue_path(project, location, queue)

# Construct the request body.
task = {
        'app_engine_http_request': {  # Specify the type of request.
            'http_method': 'POST',
            'relative_uri': '/example_task_handler'
        }
}

# The API expects a payload of type bytes.
payload = "This is a task."
converted_payload = payload.encode()

# Add the payload to the request.
task['app_engine_http_request']['body'] = converted_payload

# Use the client to build and send the task.
response = client.create_task(parent, task)

print('Created task {}'.format(response.name))
```

CLOUD TASKS

- Create a task
  - Posts a request to your App Engine app that will run on a different service

```
import datetime

from google.cloud import tasks_v2
from google.protobuf import timestamp_pb2

# Create a client.
client = tasks_v2.CloudTasksClient()

project = 'photo-timeline-python3'
queue = 'photo-timeline-queue'
location = 'us-central1'

# Construct the fully qualified queue name.
parent = client.queue_path(project, location, queue)

# Construct the request body.
task = {
        'app_engine_http_request': {  # Specify the type of request.
            'http_method': 'POST',
            'relative_uri': '/example_task_handler'
        }
}

# The API expects a payload of type bytes.
payload = "This is a task."
converted_payload = payload.encode()

# Add the payload to the request.
task['app_engine_http_request']['body'] = converted_payload

# Use the client to build and send the task.
response = client.create_task(parent, task)

print('Created task {}'.format(response.name))
```

CLOUD TASKS

- Create a task
  - Posts a request to your App Engine app that will run on a different service

```
import datetime

from google.cloud import tasks_v2
from google.protobuf import timestamp_pb2

# Create a client.
client = tasks_v2.CloudTasksClient()

project = 'photo-timeline-python3'
queue = 'photo-timeline-queue'
location = 'us-central1'

# Construct the fully qualified queue name.
parent = client.queue_path(project, location, queue)

# Construct the request body.
task = {
        'app_engine_http_request': {  # Specify the type of request.
            'http_method': 'POST',
            'relative_uri': '/example_task_handler'
        }
}

# The API expects a payload of type bytes.
payload = "This is a task."
converted_payload = payload.encode()

# Add the payload to the request.
task['app_engine_http_request']['body'] = converted_payload

# Use the client to build and send the task.
response = client.create_task(parent, task)

print('Created task {}'.format(response.name))
```

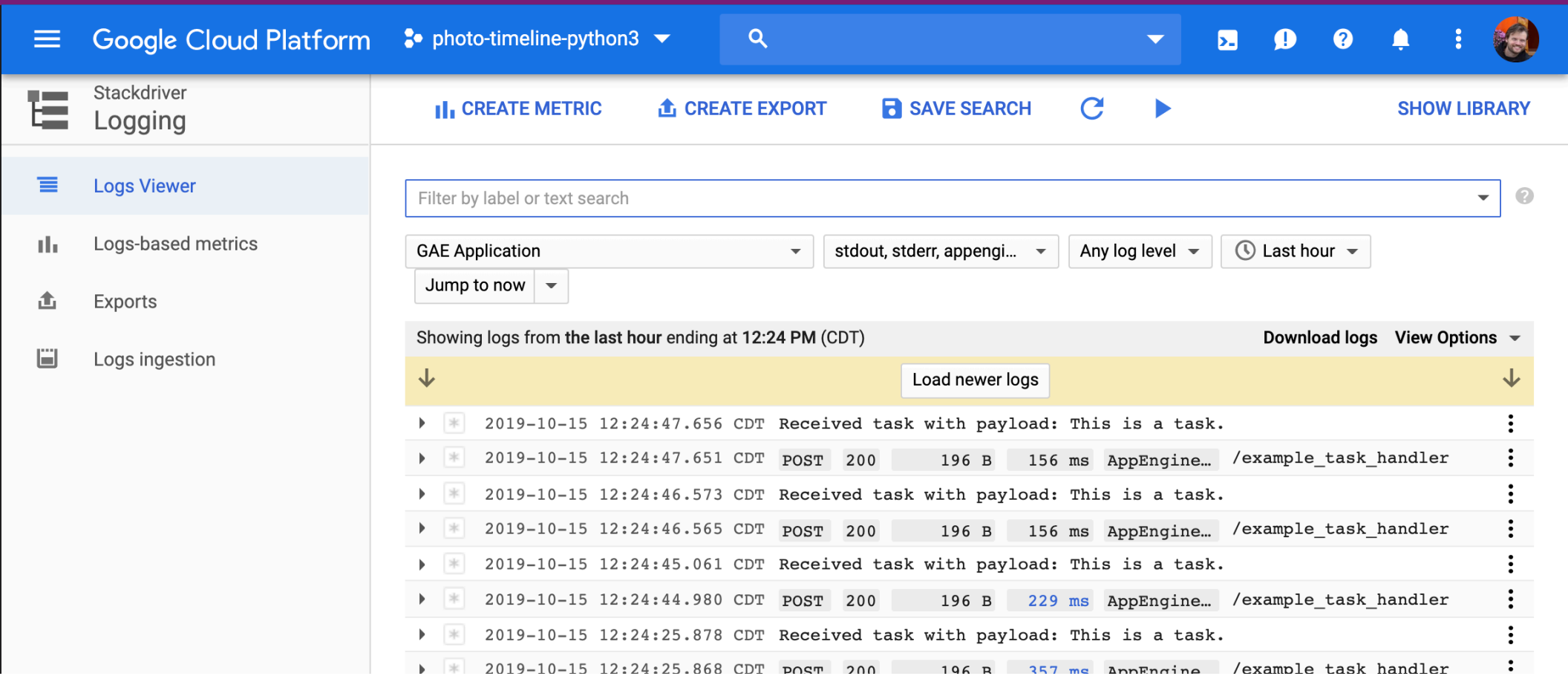
CLOUD TASKS

- Can schedule the task

```
# Construct the request body.
task = {
        'app_engine_http_request': {  # Specify the type of request.
            'http_method': 'POST',
            'relative_uri': '/example_task_handler'
        }
}

# The API expects a payload of type bytes.
payload = "This is a task."
converted_payload = payload.encode()

# Add the payload to the request.
task['app_engine_http_request']['body'] = converted_payload

# Convert "seconds from now" into an rfc3339 datetime string.
in_seconds = 10
d = datetime.datetime.utcnow() + datetime.timedelta(seconds=in_seconds)

# Create Timestamp protobuf.
timestamp = timestamp_pb2.Timestamp()
timestamp.FromDatetime(d)

# Add the timestamp to the tasks.
task['schedule_time'] = timestamp

# Use the client to build and send the task.
response = client.create_task(parent, task)
```

MONITOR A TASK

CLOUD TASKS


Google Cloud Platform
photo-timeline-python3
Stackdriver
Logging
CREATE METRIC
CREATE EXPORT
SAVE SEARCH
SHOW LIBRARY
Logs Viewer
Logs-based metrics
Exports
Logs ingestion
Filter by label or text search
GAE Application
stdout, stderr, appengi...
Any log level
Last hour
Jump to now
Showing logs from the last hour ending at 12:24 PM (CDT)
Download logs
View Options
Load newer logs
2019-10-15 12:24:47.656 CDT Received task with payload: This is a task.
2019-10-15 12:24:47.651 CDT POST 200 196 B 156 ms AppEngine… /example_task_handler
2019-10-15 12:24:46.573 CDT Received task with payload: This is a task.
2019-10-15 12:24:46.565 CDT POST 200 196 B 156 ms AppEngine… /example_task_handler
2019-10-15 12:24:45.061 CDT Received task with payload: This is a task.
2019-10-15 12:24:44.980 CDT POST 200 196 B 229 ms AppEngine… /example_task_handler
2019-10-15 12:24:25.878 CDT Received task with payload: This is a task.
2019-10-15 12:24:25.868 CDT POST 200 196 B 357 ms AppEngine… /example_task_handler

CLOUD TASKS

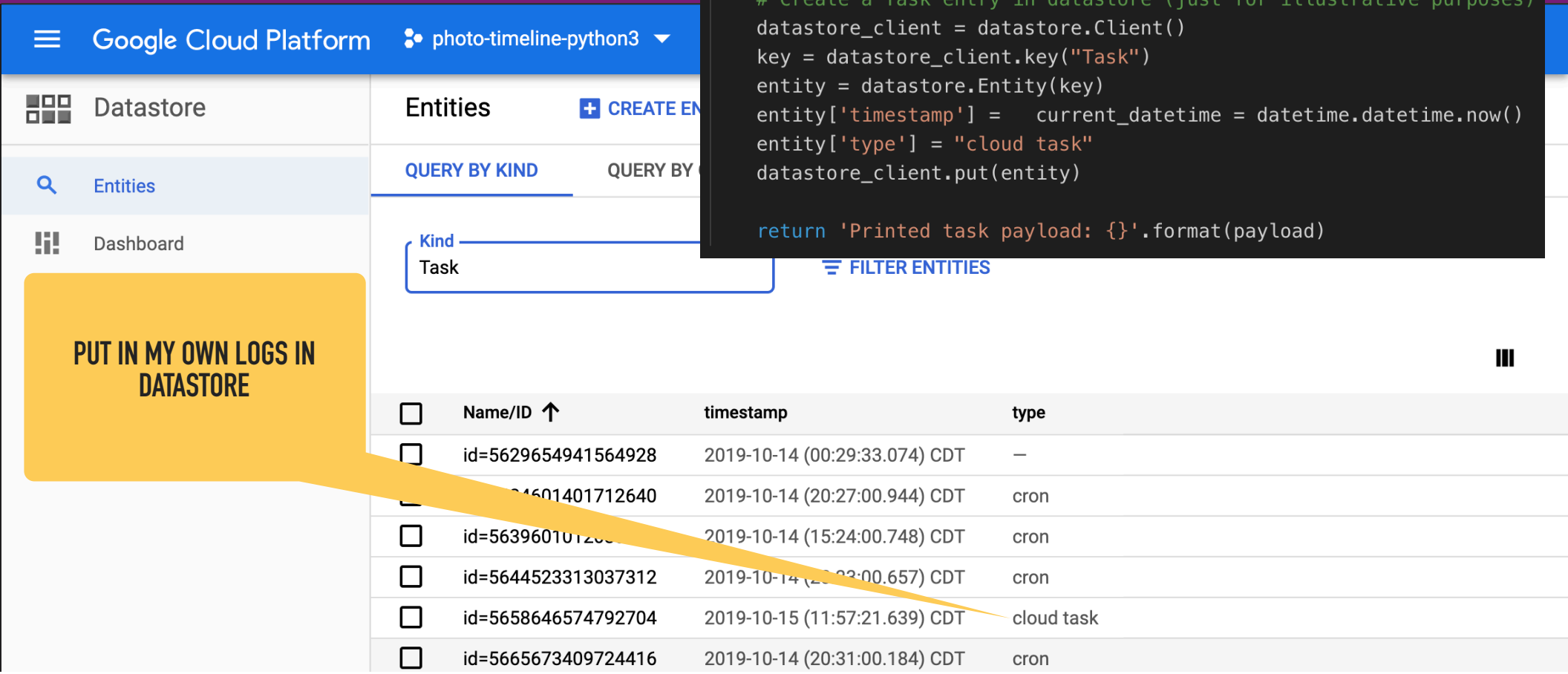
- Create a task
  - Posts a request to your App Engine app that will run on a different service

```
2019-10-15 01:25:02 default[20191014t200735]  SyntaxError: invalid syntax
2019-10-15 01:25:02 default[20191014t200735]  [2019-10-15 01:25:02 +0000] [23] [INFO] Worker exit
2019-10-15 01:25:02 default[20191014t200735]  [2019-10-15 01:25:02 +0000] [7] [INFO] Shutting dow
2019-10-15 01:25:02 default[20191014t200735]  [2019-10-15 01:25:02 +0000] [7] [INFO] Reason: Work
2019-10-15 01:26:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:26:02 default[20191014t202421]  [2019-10-15 01:26:02 +0000] [8] [INFO] Starting gun
2019-10-15 01:26:02 default[20191014t202421]  [2019-10-15 01:26:02 +0000] [8] [INFO] Listening at
2019-10-15 01:26:02 default[20191014t202421]  [2019-10-15 01:26:02 +0000] [8] [INFO] Using worker
2019-10-15 01:26:02 default[20191014t202421]  [2019-10-15 01:26:02 +0000] [20] [INFO] Booting wor
2019-10-15 01:26:02 default[20191014t202421]  [2019-10-15 01:26:02 +0000] [22] [INFO] Booting wor
2019-10-15 01:27:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:28:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:29:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:30:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:31:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:32:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:33:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:34:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:35:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 01:36:00 default[20191014t202421]  "GET /cron_task_handler HTTP/1.1" 200
2019-10-15 16:57:21 default[20191014t202421]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:06:01 default[20191014t202421]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:14:13 default[20191014t202421]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:18:45 default[20191015t121528]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:18:47 default[20191015t121528]  [2019-10-15 17:18:47 +0000] [7] [INFO] Starting gun
2019-10-15 17:18:47 default[20191015t121528]  [2019-10-15 17:18:47 +0000] [7] [INFO] Listening at
2019-10-15 17:18:47 default[20191015t121528]  [2019-10-15 17:18:47 +0000] [7] [INFO] Using worker
2019-10-15 17:18:47 default[20191015t121528]  [2019-10-15 17:18:47 +0000] [19] [INFO] Booting wor
2019-10-15 17:18:47 default[20191015t121528]  [2019-10-15 17:18:47 +0000] [22] [INFO] Booting wor
2019-10-15 17:18:51 default[20191015t121528]  Received task with payload: This is a task.
2019-10-15 17:23:12 default[20191015t121528]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:23:13 default[201                                          payload: This is a task.
2019-10-15 17:24:25 default[201                                          handler HTTP/1.1" 200
2019-10-15 17:24:25 default[201                                          payload: This is a task.
2019-10-15 17:24:44 default[20191015t121528]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:24:45 default[20191015t121528]  Received task with payload: This is a task.
2019-10-15 17:24:46 default[20191015t121528]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:24:46 default[20191015t121528]  Received task with payload: This is a task.
2019-10-15 17:24:47 default[20191015t121528]  "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:24:47 default[20191015t121528]  Received task with payload: This is a task.
```

GCLOUD APP LOGS READ

CLOUD TASKS

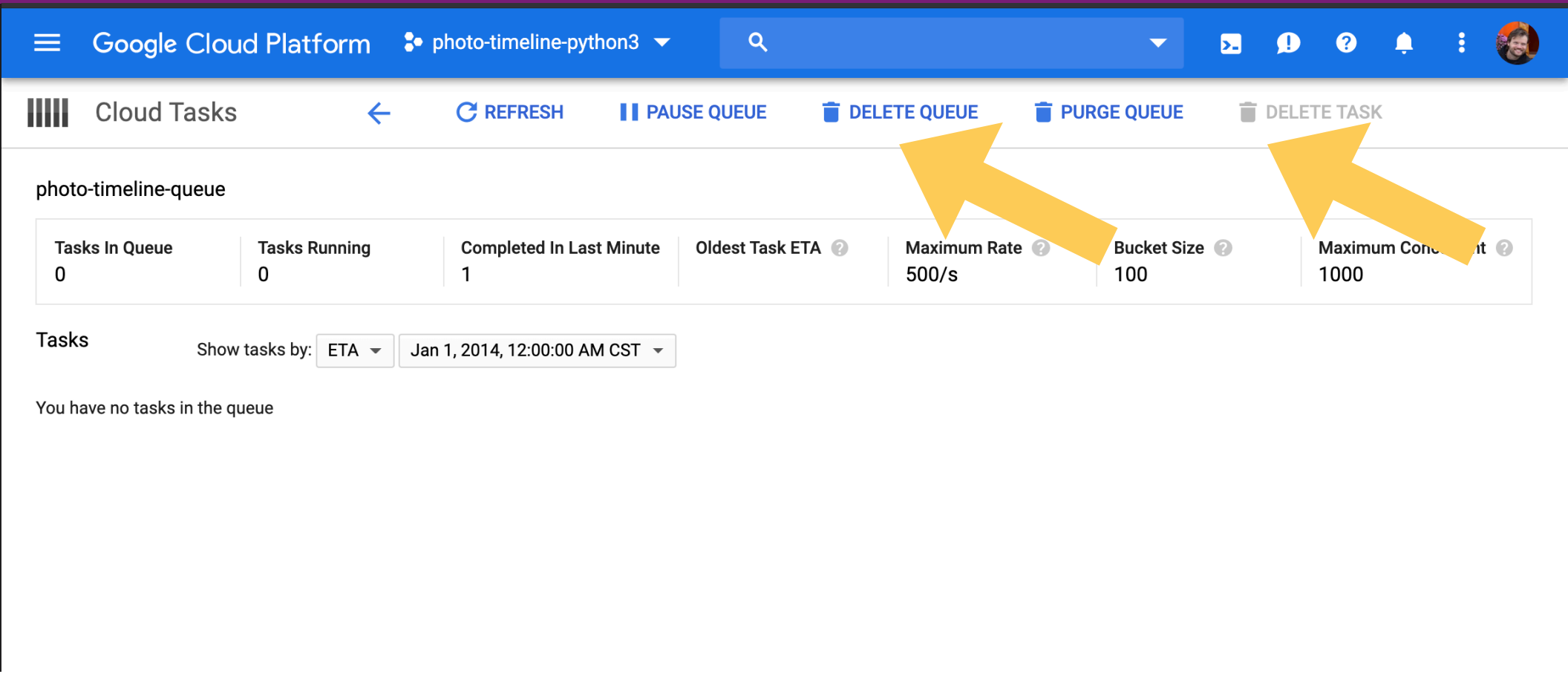
```python
@app.route('/example_task_handler', methods=['POST'])
def example_task_handler():
    """Log the request payload."""
    payload = request.get_data(as_text=True) or '(empty payload)'
    print('Received task with payload: {}'.format(payload))

    # Create a Task entry in datastore (just for illustrative purposes)
    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =    current_datetime = datetime.datetime.now()
    entity['type'] = "cloud task"
    datastore_client.put(entity)

    return 'Printed task payload: {}'.format(payload)
```



PUT IN MY OWN LOGS IN DATASTORE

DELETE A TASK

CLOUD TASKS


Google Cloud Platform
photo-timeline-python3
Cloud Tasks
REFRESH
PAUSE QUEUE
DELETE QUEUE
PURGE QUEUE
DELETE TASK
photo-timeline-queue
Tasks In Queue
0
Tasks Running
0
Completed In Last Minute
1
Oldest Task ETA
Maximum Rate
500/s
Bucket Size
100
1000
Tasks
Show tasks by:
ETA
Jan 1, 2014, 12:00:00 AM CST
You have no tasks in the queue

CLOUD TASKS

- API
  - Must retain the task ID

```
queue_path = client.queue_path(project, location, queue)
response = client.delete_queue(queue_path)
```

```
queues/photo-timeline-queue/tasks/7976648373617469170l
```

CLOUD TASKS

- Push queue
  - Long running operations
  - Scheduled tasks
- Pull queue
  - Tasks that are interdependent
  - Related tasks that can be batched for efficiency

CLOUD TASKS

- Task queues come in two flavors, push and pull
- The manner in which the Task Queue service dispatches task requests to worker services is different for the different queues

Task Queue Overview

Contents

Push queues and pull queues
Use cases
  Push queues
  Pull queues
What's next

Python | Java | P

This page describes what task queues are, and when and how to use them. The Tas
API lets applications perform work, called *tasks*, asynchronously outside of a user r
an app needs to execute work in the background, it adds tasks to *task queues*. The t
executed later, by scalable App Engine worker services in your application.

Push queues and pull queues

Task queues come in two flavors, *push* and *pull*. The manner in which the Task Queu
service dispatches task requests to worker services is different for the different que

Push queues dispatch requests at a reliable, steady rate. They guarantee reliable ta
execution. Because you can control the rate at which tasks are sent from the queue
control the workers' scaling behavior and hence your costs.

Because tasks are executed as App Engine requests targeted at services, they are s
stringent deadlines. Tasks handled by automatic scaling services must finish in ten
Tasks handled by basic and manual scaling services can run for up to 24 hours.

SCHEDULING TASKS WITH CRON

SCHEDULING TASKS WITH CRON

- Cron Service allows you to configure regularly scheduled tasks that operate at defined times or regular intervals
  - "cron" jobs in unix
- Jobs are automatically triggered by the App Engine Cron Service

Scheduling Tasks With Cron for Python

Contents

Creating a cron job
Testing cron jobs in the development server
Uploading cron jobs
Deleting all cron jobs

...

Pyth

The App Engine Cron Service allows you to configure regularly scheduled tasks that opera
regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are aut
the App Engine Cron Service. For instance, you might use a cron job to send out an email
or to update some cached data every 10 minutes, or refresh summary information once a

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job r
same limits as those for push task queues.

Creating a cron job

1. Create the `cron.yaml` file in the root directory of your application (alongside `app.y`
2. Add one or more `<cron>` entries to your file and define the necessary elements for
   required `<url>` and `<schedule>` elements.

   The following example creates a basic cron job that runs daily:

```
cron:
- description: daily summary job
```

SCHEDULING TASKS WITH CRON

- Use cases
  - Send out an email report on a daily basis
  - Update cached data every 10 minutes
  - Refresh summary information once an hour

Scheduling Tasks With Cron for Python

Contents

Creating a cron job
Testing cron jobs in the development server
Uploading cron jobs
Deleting all cron jobs
...

Pyth

The App Engine Cron Service allows you to configure regularly scheduled tasks that opera
regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are aut
the App Engine Cron Service. For instance, you might use a cron job to send out an email
or to update some cached data every 10 minutes, or refresh summary information once a

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job r
same limits as those for push task queues.

Creating a cron job

1. Create the `cron.yaml` file in the root directory of your application (alongside `app.y`
2. Add one or more `<cron>` entries to your file and define the necessary elements for
   required `<url>` and `<schedule>` elements.

   The following example creates a basic cron job that runs daily:

```
cron:
- description: daily summary job
```

SCHEDULING TASKS WITH CRON

- A cron job
  - invokes a URL
  - using an HTTP GET request
  - at a given time of day

Scheduling Tasks With Cron for Python

Contents

Creating a cron job
Testing cron jobs in the development server
Uploading cron jobs
Deleting all cron jobs

Pyth

The App Engine Cron Service allows you to configure regularly scheduled tasks that opera
regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are aut
the App Engine Cron Service. For instance, you might use a cron job to send out an email
or to update some cached data every 10 minutes, or refresh summary information once a

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job r
same limits as those for push task queues.

Creating a cron job

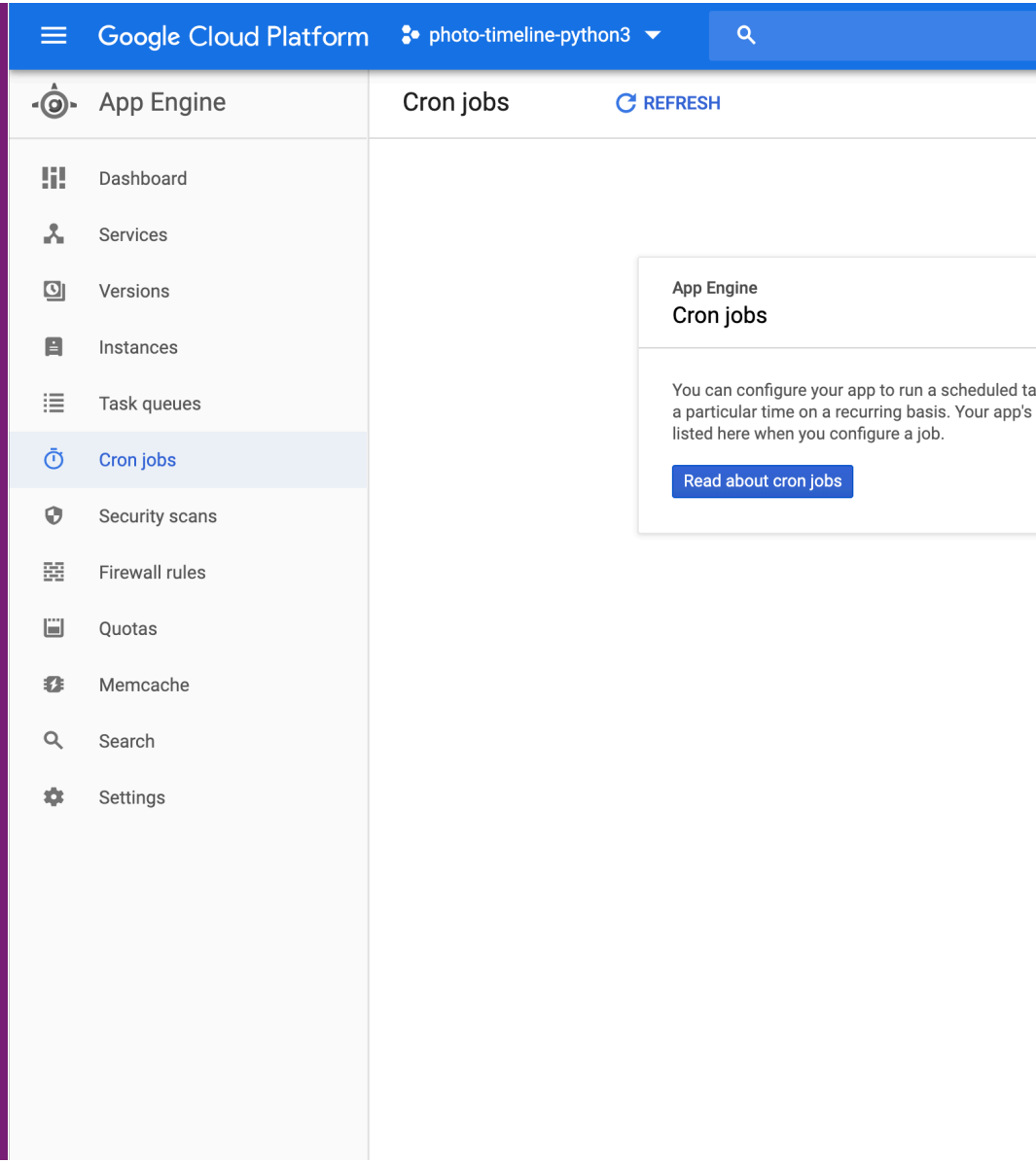
1. Create the `cron.yaml` file in the root directory of your application (alongside `app.y`
2. Add one or more `<cron>` entries to your file and define the necessary elements for
   required `<url>` and `<schedule>` elements.

   The following example creates a basic cron job that runs daily:

```
cron:
- description: daily summary job
```

SCHEDULING TASKS WITH CRON

- Free applications can have up to 20 scheduled tasks
- Paid applications can have up to 250 scheduled tasks.



SCHEDULING TASKS WITH CRON

- Create a cron job
  - Add a cron.yaml file
  - Add entries
  - Create a handler for cron jobs URL

```
cron:

- description: "test cron"
  url: /cron_task_handler
  schedule: every mins

  - description: "monday morni
  url: /mail/weekly
  schedule: every monday 09:00

  - description: "daily summar
  url: /tasks/summary
  schedule: every 24 hours
```

CRON.YAML

SCHEDULING TASKS WITH CRON

App Engine > Documentation > Python > Standard Environment for Python 3

Scheduling Jobs with cron.yaml

☆ ☆ ☆ ☆ ☆

SEND FEEDBACK

Contents

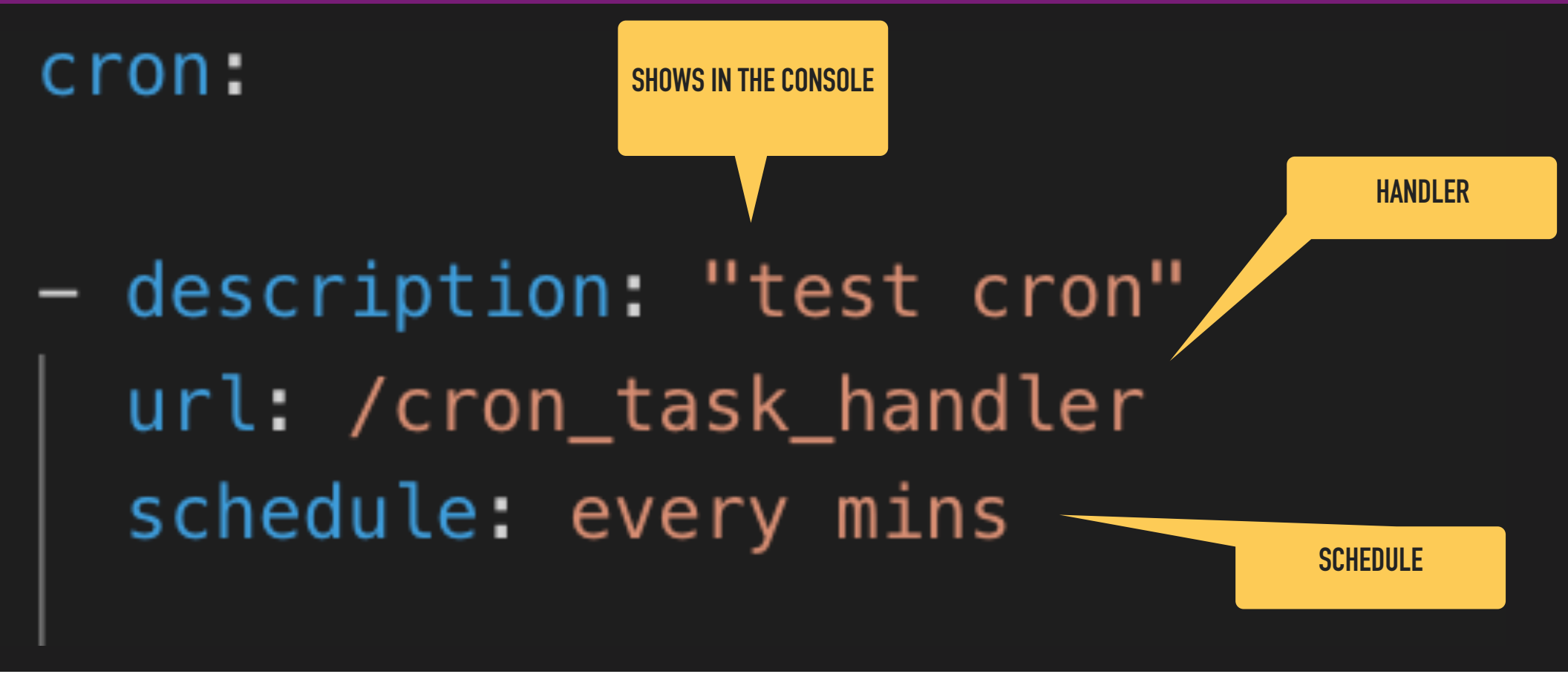
About the cron configuration file
Defining the cron job schedule
Formatting the schedule
Specifying retries
...

HTTPS://CLOUD.GOOGLE.COM/APPENGINE/DOCS/STANDARD/PYTHON3/SCHEDULING-JOBS-WITH-CRON-YAML

Python 2.7/**3.7** | Java 8/11 | PHP 5/7 | Ruby | Go 1.9/1.11/1.12 | Node.js

The App Engine Cron Service allows you to configure regularly scheduled tasks that operate at defined

SCHEDULING TASKS WITH CRON



SCHEDULING TASKS WITH CRON

```
cron:
- description: "test cron"
  url: /cron_task_handler
  schedule: every mins
- description: "monday morning mailout"
  url: /mail/weekly
  schedule: every monday 09:00
- description: "daily summary job"
  url: /tasks/summary
  schedule: every 24 hours
```

SCHEDULING TASKS WITH CRON

```
every 12 hours
every 5 minutes from 10:00 to 14:00
every day 00:00
every monday 09:00
2nd,third mon,wed,thu of march 17:00
1st monday of sep,oct,nov 17:00
1 of jan,april,july,oct 00:00
```

- Schedule format

SCHEDULING TASKS WITH CRON

```yaml
cron:

- description: "test cron"
  url: /cron_task_handler
  schedule: every mins
  target: beta
```

RUNS ON DEFAULT BY DEFAULT. OTHERWISE SPECIFY A TARGET OR VERSION

SCHEDULING TASKS WITH CRON

```
cron:

- description: "test cron"
  url: /cron_task_handler
  schedule: every mins
  target: beta
```

MAIN.PY

```
@app.route('/cron_task_handler')
def cron_task_handler():…
```

CRON HANDLER ROUTES

SCHEDULING TASKS WITH CRON

- The handler should execute any tasks that you want scheduled
- The handler should respond with an HTTP status code between 200 and 299 (inclusive) to indicate success
  - Other status codes can be returned and can be used to trigger retrying the job

```
cron:

- description: "test
  url: /cron_task_har
  schedule: every mir

  - description: "mor
  url: /mail/weekly
  schedule: every mor

  - description: "dai
  url: /tasks/summary
  schedule: every 24
```

SCHEDULING TASKS WITH CRON

```
cron:

- description: "test cron"
  url: /cron_task_handler
  schedule: every mins
  target: beta
```

MAIN.PY

```
@app.route('/cron_task_handler')
def cron_task_handler():…
```

SCHEDULING TASKS WITH CRON

```
@app.route('/cron_task_handler')
def cron_task_handler():
    """Log out the request"""
    logging.info('Running cron task')

    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cron"
    datastore_client.put(entity)
    return "Ran cron task"
```

SCHEDULING TASKS WITH CRON

```
@app.route('/cron_task_handler')
def cron_task_handler():
    """Log out the request"""
    logging.info('Running cron task')

    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cron"
    datastore_client.put(entity)
    return "Ran cron task"
```

TASK IS TO ADD A NEW TASK ENTITY IN DATASTORE

SCHEDULING TASKS WITH CRON

```
@app.route('/cron_task_handler')
def cron_task_handler():
    """Log out the request"""
    logging.info('Running cron task')

    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.datetime.now()
    entity['type'] = "cron"
    datastore_client.put(entity)
    return "Ran cron task"
```

YOU'LL ONLY SEE THE LOGGING STATEMENT TO KNOW HOW IT RAN

SCHEDULING TASKS WITH CRON

```
@app.route('/cron_task_handler')
def cron_task_handler():
    """Log out the request"""
    logging.info('Running cron task')

    datastore_client = datastore.Client()
    key = datastore_client.key("Task")
    entity = datastore.Entity(key)
    entity['timestamp'] =   current_datetime = datetime.                    )
    entity['type'] = "cron"
    datastore_client.put(entity)
    return "Ran cron task"
```

YOU'LL ONLY SEE THE LOGGING STATEMENT TO KNOW HOW IT RAN

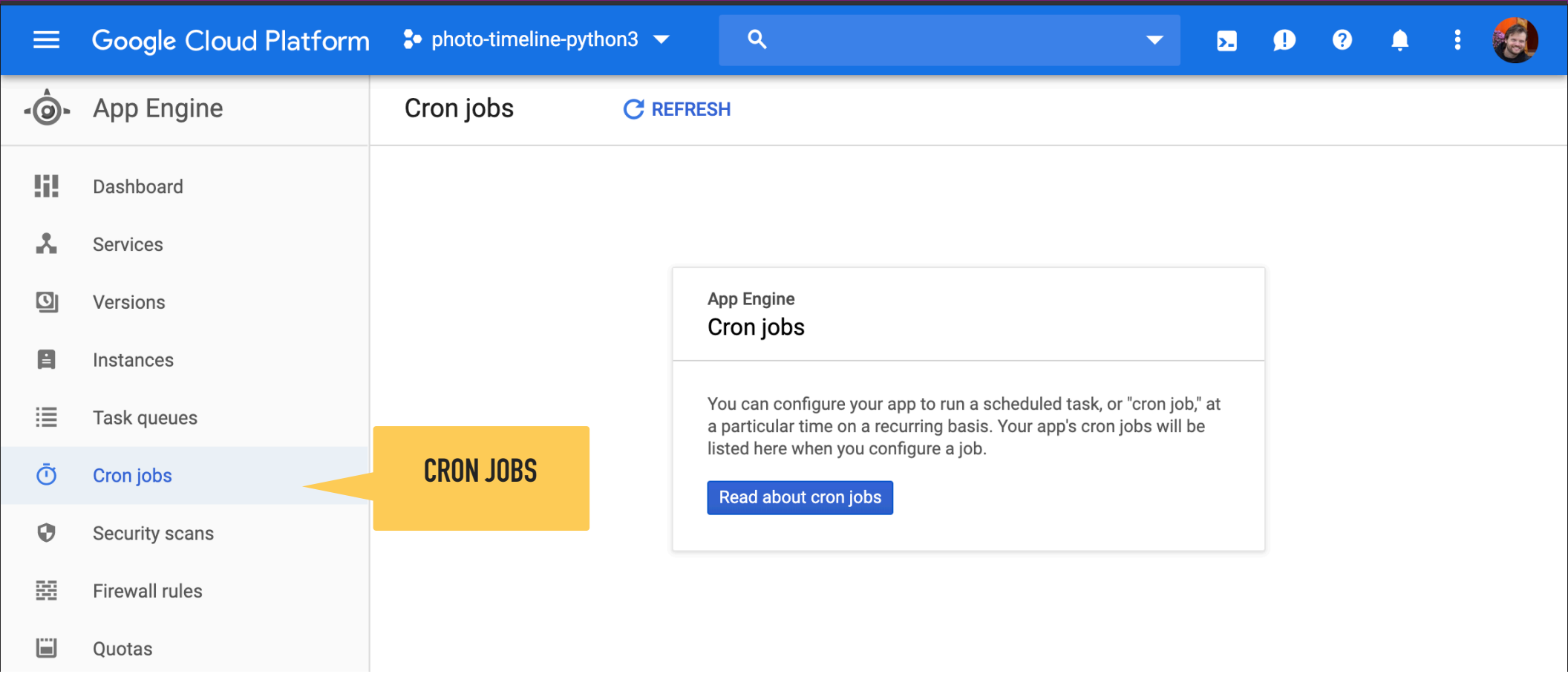
MUST RETURN A 200 CODE FOR SUCESS

CRON ROUTES

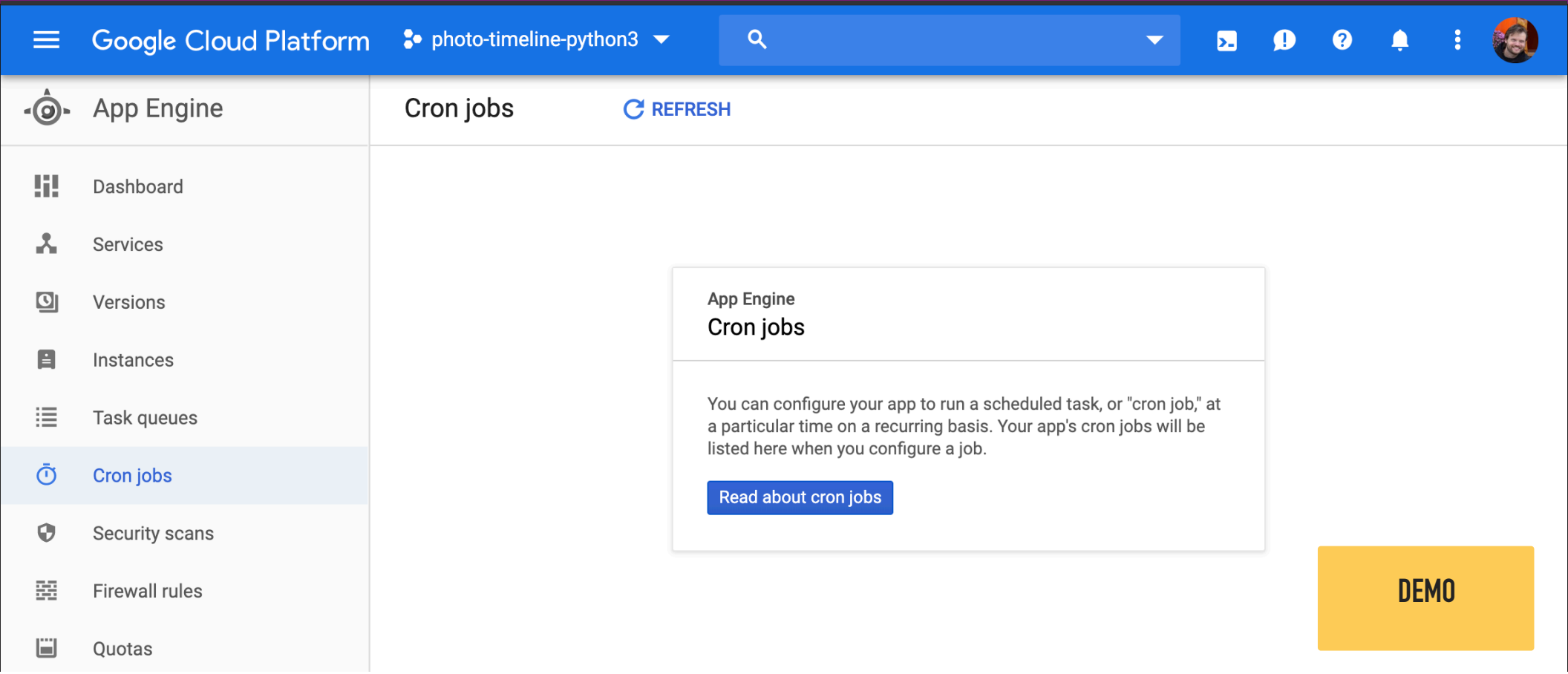
SCHEDULING TASKS WITH CRON

```
# To upload your cron jobs, you must specify the
cron.yaml as a parameter to the following gcloud
command

gcloud app deploy cron.yaml
```

PUSH QUEUES


Google Cloud Platform
photo-timeline-python3
App Engine
Dashboard
Services
Versions
Instances
Task queues
Cron jobs
Security scans
Firewall rules
Quotas
Cron jobs
REFRESH
App Engine
Cron jobs
You can configure your app to run a scheduled task, or "cron job," at a particular time on a recurring basis. Your app's cron jobs will be listed here when you configure a job.
Read about cron jobs
CRON JOBS

PUSH QUEUES


Google Cloud Platform
photo-timeline-python3
App Engine
Cron jobs
REFRESH
Dashboard
Services
Versions
Instances
Task queues
Cron jobs
Security scans
Firewall rules
Quotas
App Engine
Cron jobs
You can configure your app to run a scheduled task, or "cron job," at a particular time on a recurring basis. Your app's cron jobs will be listed here when you configure a job.
Read about cron jobs
DEMO

SCHEDULING TASKS WITH CRON

```
# To delete all cron jobs, change the cron.yaml file
to just contain:

cron:



# Deploy
gcloud app deploy cron.yaml
```

SCHEDULING TASKS WITH CRON

```
# You can display the parsed version of your cron
jobs, including the times the jobs will run

appcfg.py cron_info command
```

QUERY CURSORS

QUERY CURSORS

- Do not want to query/return all results for a mobile application

```python
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photos')
    image_entities = list(query.fetch(limit=2))

    # Return a Jinja2 HTML template and pass in image_entities as a paramet
    return render_template('homepage.html', image_entities=image_entities,
```

COULD LIMIT RESULTS RETURNED...BUT WHERE DID WE LEAVE OFF?

QUERY CURSORS

- Cursors datastore feature that simplifies retrieving continuous batched results
  - pagination

Cursors, limits, and offsets

You can specify a *limit* for your query to control the maximum number of results returned in one batch. The following example retrieves at most five Task entities:

C# GO JAVA NODE.JS PHP **PYTHON** RUBY GQL

To learn how to install and use the client library for Cloud Datastore, see the Cloud Datastore Client Libraries . For more information, see the Cloud Datastore Python API reference documentation.

VIEW ON GITHUB FEEDBACK

```
query = client.query()
tasks = list(query.fetch(limit=5))
```

Query cursors allow an application to retrieve a query's results in convenient batches without incurring the overhead of a query offset. After performing a retrieval operation, the application can obtain a cursor, which is an opaque byte string marking the index position of the last result retrieved. The application can save this string (for instance in your Datastore mode database, a cache, or embedded in a web page as a base-64 encoded HTTP `GET` or `POST` parameter), and can then use the cursor as the starting point for a subsequent retrieval operation to obtain the next batch of results from the point where the previous retrieval ended. A retrieval can also specify an end cursor, to limit the extent of the result set returned.

The following example demonstrates the use of cursors for pagination:

C# GO JAVA NODE.JS PHP **PYTHON** RUBY GQL

To learn how to install and use the client library for Cloud Datastore, see the Cloud Datastore Client Libraries . For more information, see the Cloud Datastore Python API reference documentation.

VIEW ON GITHUB FEEDBACK

```
def get_one_page_of_tasks(cursor=None):
    query = client.query(kind='Task')
    query_iter = query.fetch(start_cursor=cursor, limit=5)
    page = next(query_iter.pages)
```

QUERY CURSORS

```python
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photos')
    #image_entities = list(query.fetch(limit=2))

    # Get cursor (if available)
    cursor = request.args.get('cursor', default=None)

    # Create the query
    query_iter = query.fetch(start_cursor=cursor, limit=2)
    page = next(query_iter.pages)
    image_entities = list(page)
    next_cursor = query_iter.next_page_token
    #print("cursor",cursor)

    # Return a Jinja2 HTML template and pass in image_entities as a parameter.
    return render_template('homepage.html', image_entities=image_entities, cursor=next_cursor.decode('UTF-8'))
```

SET UP QUERY AS NORMAL

QUERY CURSORS

GET THE CURSOR FROM THE REQUEST (IF THERE IS ONE)

LOCALHOST:8080?CURSOR=XXXX

SET CURSOR TO NONE FOR THE FIRST BATCH

```python
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photos')
    #image_entities = list(query.fetch(limit

    # Get cursor (if available)
    cursor = request.args.get('cursor', default=None)

    # Create the query
    query_iter = query.fetch(start_cursor=cursor, limit=2)
    page = next(query_iter.pages)
    image_entities = list(page)
    next_cursor = query_iter.next_page_token
    #print("cursor",cursor)

    # Return a Jinja2 HTML template and pass in image_entities as a parameter.
    return render_template('homepage.html', image_entities=image_entities, cursor=next_cursor.decode('UTF-8'))
```

QUERY CURSORS

```
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Ph      )
    #image_entities = list(query.fetch(li    2))

    # Get cursor (if available)
    cursor = request.args.get('cu    or', default=None)

    # Create the query
    query_iter = query.fetch(start_cursor=cursor, limit=2)
    page = next(query_iter.pages)
    image_entities = list(page)
    next_cursor = query_iter.next_page_token
    #print("cursor",cursor)

    # Return a Jinja2 HTML template and pass in image_entities as a parameter.
    return render_template('homepage.html', image_entities=image_entities, cursor=next_cursor.decode('UTF-8'))
```

CREATE QUERY

HOW MANY

GET NEXT PAGE A
ND CURSOR

QUERY CURSORS

```python
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photos')
    #image_entities = list(query.fetch(limit=2))

    # Get cursor (if available)
    cursor = request.args.get('cursor', default=None)

    # Create the query
    query_iter = query.fetch(start_cursor=cursor, limit=2)
    page = next(query_iter.pages)
    image_entities = list(page)
    next_cursor = query_iter.next_page_token
    #print("cursor",cursor)

    # Return a Jinja2 HTML template and pass in image_entities as a parameter.
    return render_template('homepage.html', image_entities=image_entities, cursor=next_cursor.decode('UTF-8'))
```

PASS IT BACK TO THE WEBPAGE

QUERY CURSORS

```python
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photos')
    #image_entities = list(query.fetch(limit=2))

    # Get cursor (if available)
    cursor = request.args.get('cursor', default=None)

    # Create the query
    query_iter = query.fetch(start_cursor=cursor, limit=2)
    page = next(query_iter.pages)
    image_entities = list(page)
    next_cursor = query_iter.next_page_token
    #print("cursor",cursor)

    # Return a Jinja2 HTML template and pass in image_entities as a parameter.
    return render_template('homepage.html', image_entities=image_entities, cursor=next_cursor.decode('UTF-8'))
```

CURSOR IS 'NONE" WHEN THERE ARE NO MORE PAGES

COULD USE BETTER ERROR HANDLING

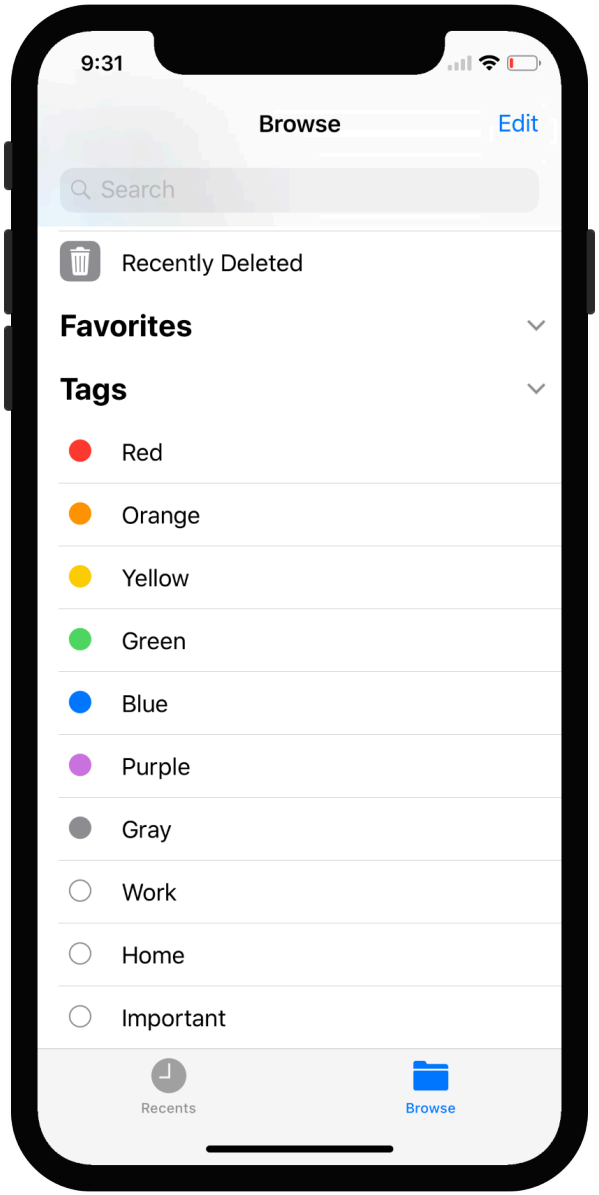
STRATEGIES FOR LOADING A TABLE

QUERY CURSORS

MOBILE CURSOR STRATEGY #1

- Monitor last table view cell
  - When loading the last cell; trigger new download
  - Append the results to the table view data source

```
if indexPath.row == privateList.count - 1 { // last cell
    if totalItems > privateList.count { // more items to fetch
        loadItem() // increment `fromIndex` by 20 before server call
    }
}
```

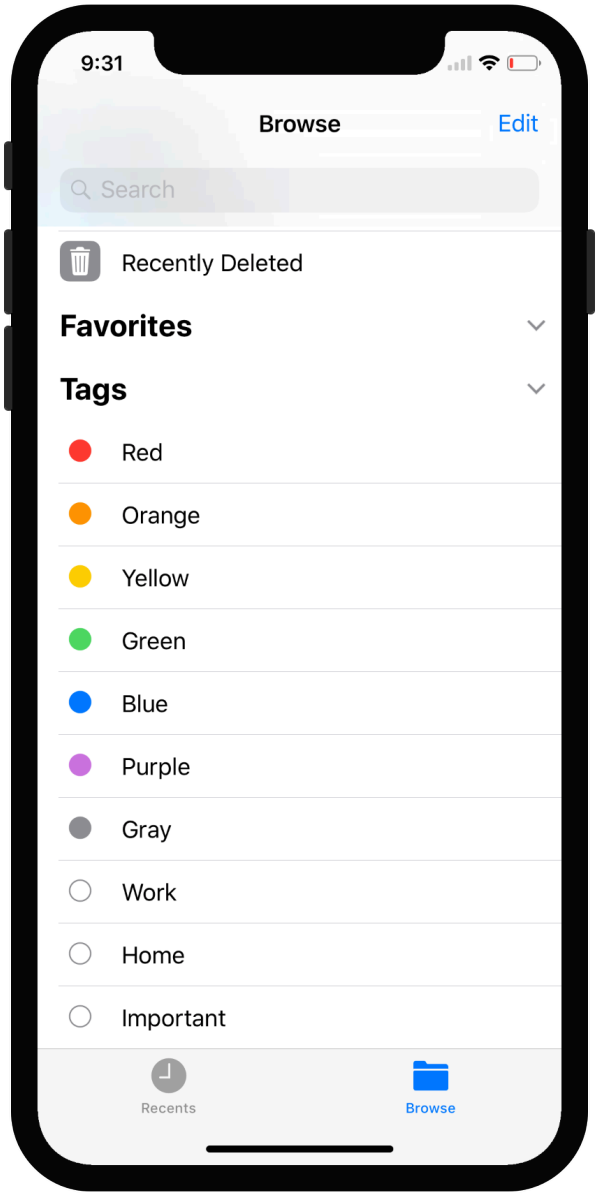


QUERY CURSORS

MOBILE CURSOR STRATEGY #1

- Add one more table view cell; monitor

```
internal func numberOfSectionsInTableView(tableView: UITableView) -> Int
{
    return 2;
}

internal func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int
    {
        if section == 0 {
            return privateList.count
        } else if section == 1 {    // this is going to be the last section with just 1 cell which will show the loading
            return 1
        }
    }

internal func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
{
    if section == 0 {
        let cell:myCell = tableView.dequeueReusableCellWithIdentifier("myCell") as! myCell

        cell.titleLabel.text = privateList[indexPath.row]


        return cell
    } else if section == 1 {
        //create the cell to show loading indicator
        ...

        //here we call loadItems so that there is an indication that something is loading and once loaded we relaod the t
        self.loadItems()
    }
}
```

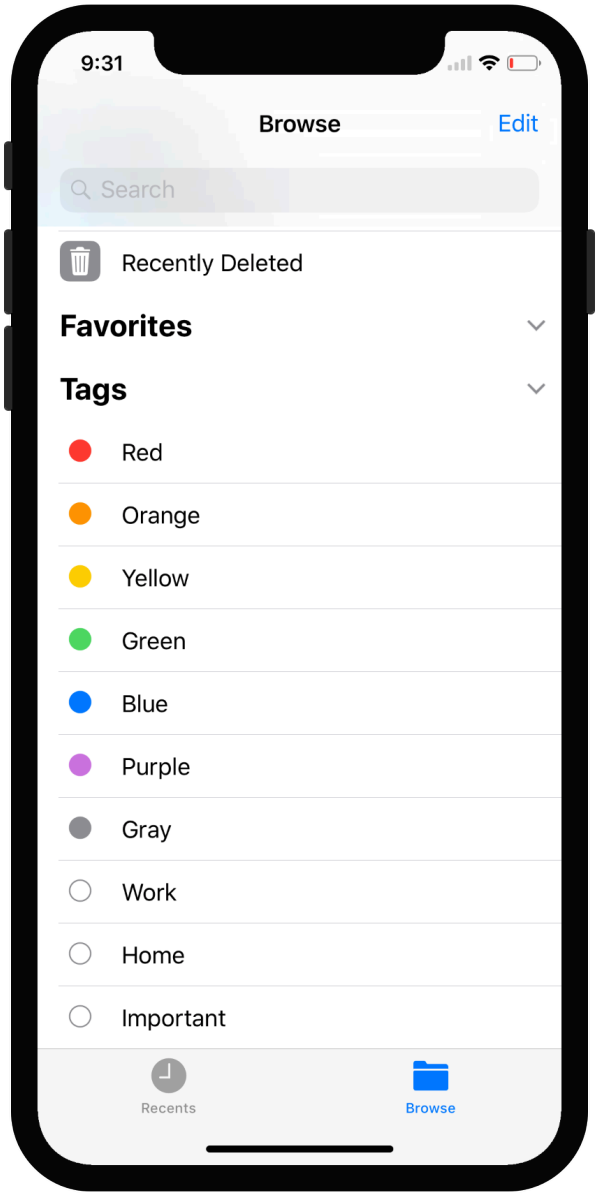


QUERY CURSORS

MOBILE CURSOR STRATEGY #3

- Monitor scroll view

```
    func scrollViewDidEndDecelerating(_ scrollView: UIScrollView) {
        print("scrollViewDidEndDecelerating")
    }
    //Pagination
    func scrollViewDidEndDragging(_ scrollView: UIScrollView, willDecelerate decelerate: Bool) {

            print("scrollViewDidEndDragging")
            if ((tableView.contentOffset.y + tableView.frame.size.height) >= tableView.contentSize.height)
            {
                if !isDataLoading{
                    isDataLoading = true
                    self.pageNo=self.pageNo+1
                    self.limit=self.limit+10
                    self.offset=self.limit * self.pageNo
                    loadCallLogData(offset: self.offset, limit: self.limit)

                }
            }


    }
```



QUERY CURSORS

- Limitations of cursors
  - A cursor can be used only by the same application that performed the original query, and only to continue the same query
  - Results aren't consistent with multiple inequality filters
  - Cursors have stale data

Limitations of cursors

Cursors are subject to the following limitations:

- A cursor can be used only by the same application that perf
  same query. To use the cursor in a subsequent retrieval ope
  including the same entity kind, ancestor filter, property filters
  using a cursor without setting up the same query from whic
- Because the `!=` and `IN` operators are implemented with m
  cursors.
- Cursors don't always work as expected with a query that use
  multiple values. The de-duplication logic for such multiple-va
  possibly causing the same result to be returned more than o
- New App Engine releases might change internal implementa
  If an application attempts to use a cursor that is no longer v
  exception.

Cursors and data updates

The cursor's position is defined as the location in the result list aft
position in the list (it's not an offset); it's a marker to which Cloud
results. If the results for a query change between uses of a cursor
after the cursor. If a new result appears before the cursor's positio
after the cursor are fetched. Similarly, if an entity is no longer a res
results that appear after the cursor do not change. If the last resu
still knows how to locate the next result.

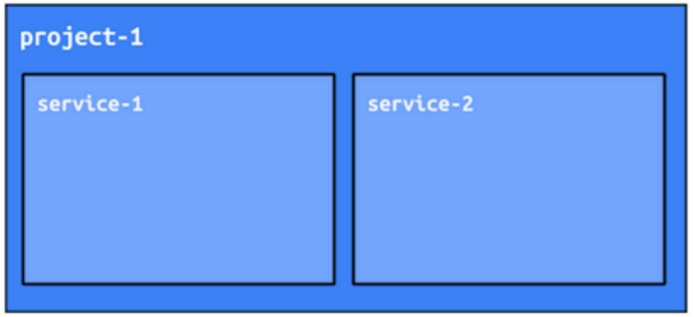
When retrieving query results, you can use both a start cursor and
from Cloud Datastore. When using a start and end cursor to retrie
the results will be the same as when you generated the cursors. E
between the time the cursors are generated and when they are us

BREAK TIME 

MICROSERVICES

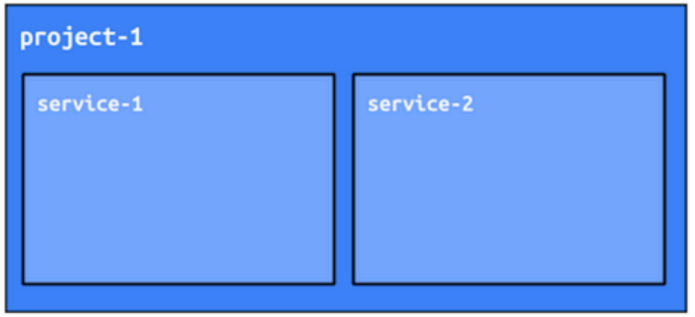
MICROSERVICES

- Microservices is an architectural style for developing applications/backends
- Allow a large application to be decomposed into independent parts
  - Each has its own realm of responsibility
- To serve a single user or API request, a microservices-based application can call many internal microservices to compose its response



MICROSERVICES

- Why (according to Google)
  - Define strong contracts between the various microservices
  - Allow for independent deployment cycles, including rollback
  - Facilitate concurrent, A/B release testing on subsystems
  - Minimize test automation and quality-assurance overhead
  - Improve clarity of logging and monitoring
  - Provide fine-grained cost accounting
  - Increase overall application scalability and reliability.



MICROSERVICES

- Functionality in our photo-timeline example
  - Post and retrieve images
  - Delete photos
  - Use account
  - Send emails
  - Run scheduled tasks to send summary

```
app = webapp2.WSGIApplication([
    ('/', HomeHandler),
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/', handler=PostHandler),
    webapp2.Route('/user/<user>/<type>/',handler=UserHandler)
    ],
    debug=True)
```

MICROSERVICES

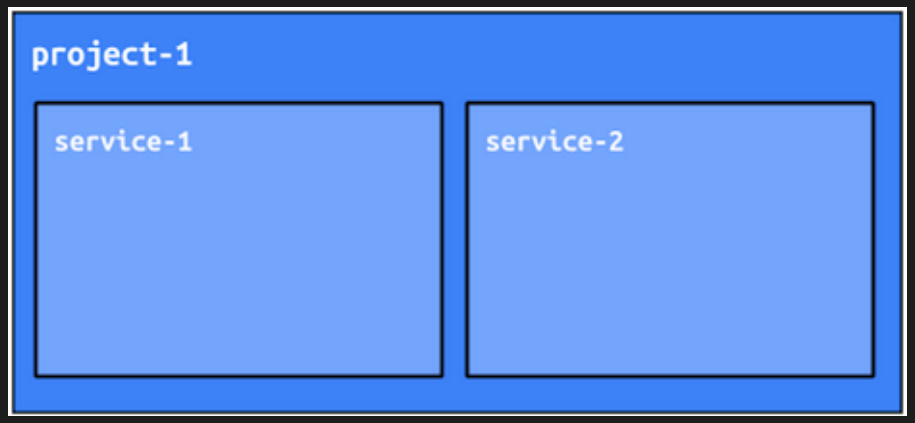
- Manageable now, but what happens when we add new features, tasks, schedule tasks?
- Compartmentalizing the function into logical segments may help
  - And minimize problems (potentially)

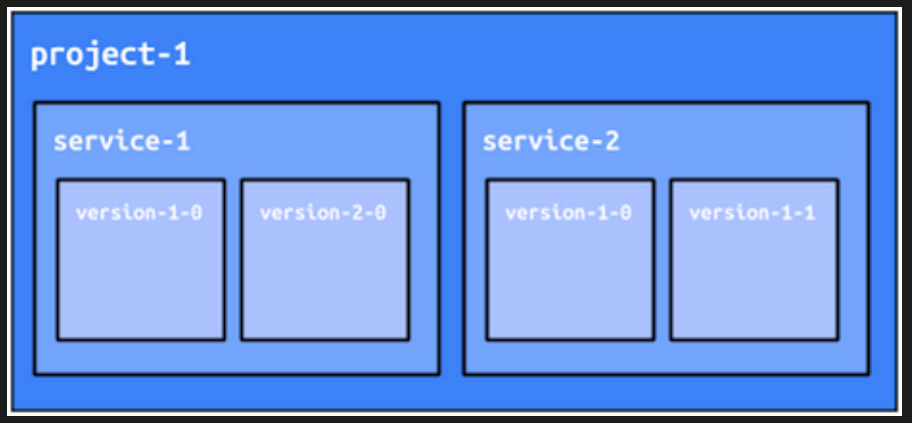
```
app = webapp2.WSGIApplication([
    ('/', HomeHandler),
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/', handler=PostHandler),
    webapp2.Route('/user/<user>/<type>/',handler=UserHandler)
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/', handler=PostHandler),
    webapp2.Route('/user/<user>/<type>/',handler=UserHandler)
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/', handler=PostHandler),
    webapp2.Route('/user/<user>/<type>/',handler=UserHandler)
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/', handler=PostHandler),
    webapp2.Route('/user/<user>/<type>/',handler=UserHandler)
    ],
    debug=True)
```

MICROSERVICES

- You can deploy multiple microservices as separate services (modules)
- These services have full isolation of code
  - Communication is done through HTTP; no direct calling another service
- Code can be deployed to services independently
  - Services can be written in different languages
- Autoscaling, load balancing, and machine instance types are all managed independently for services.

MICROSERVICES

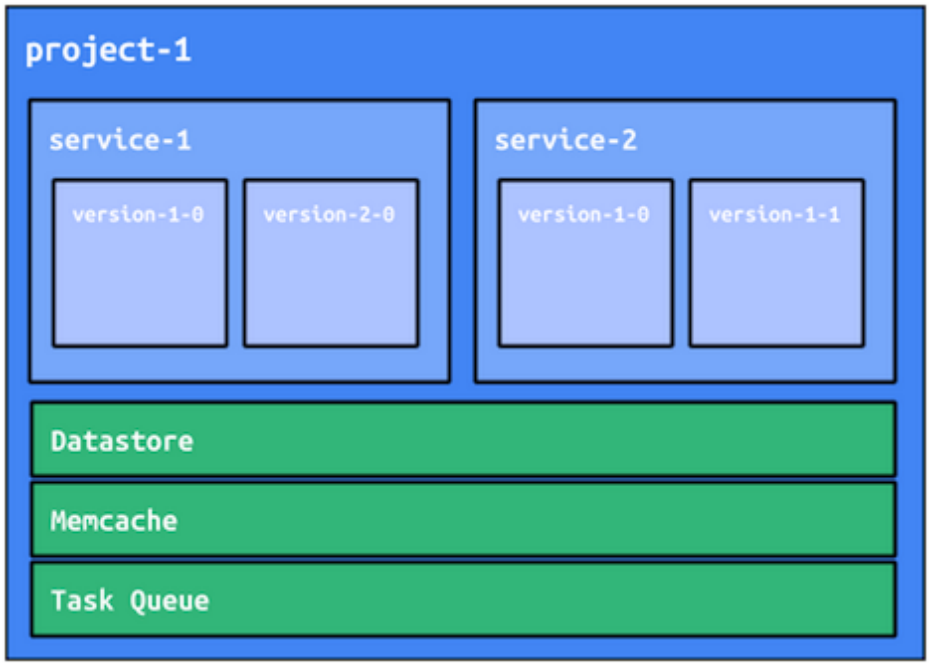




A PROJECT CAN HAVE SERVICES WITH VERSIONS AND SO ON....EASY UPDATE AND TESTING

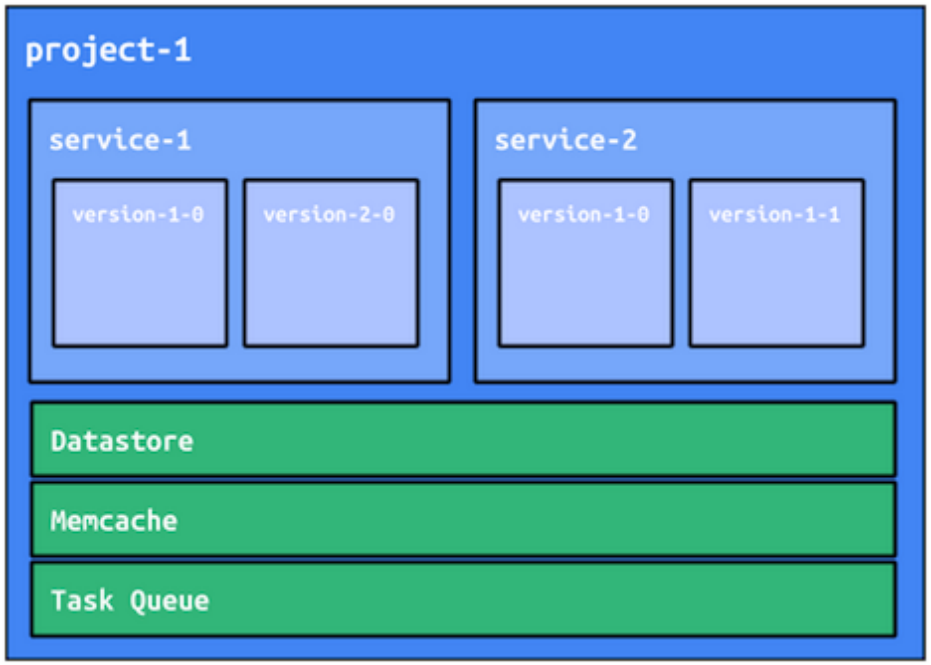
MICROSERVICES

- Some services share some App Engine resources
  - Cloud Datastore
  - Memcache
  - Task Queues



MICROSERVICES

- If this doesn't fit your application, having different projects is an option
  - Transparent to user
  - Domain routing



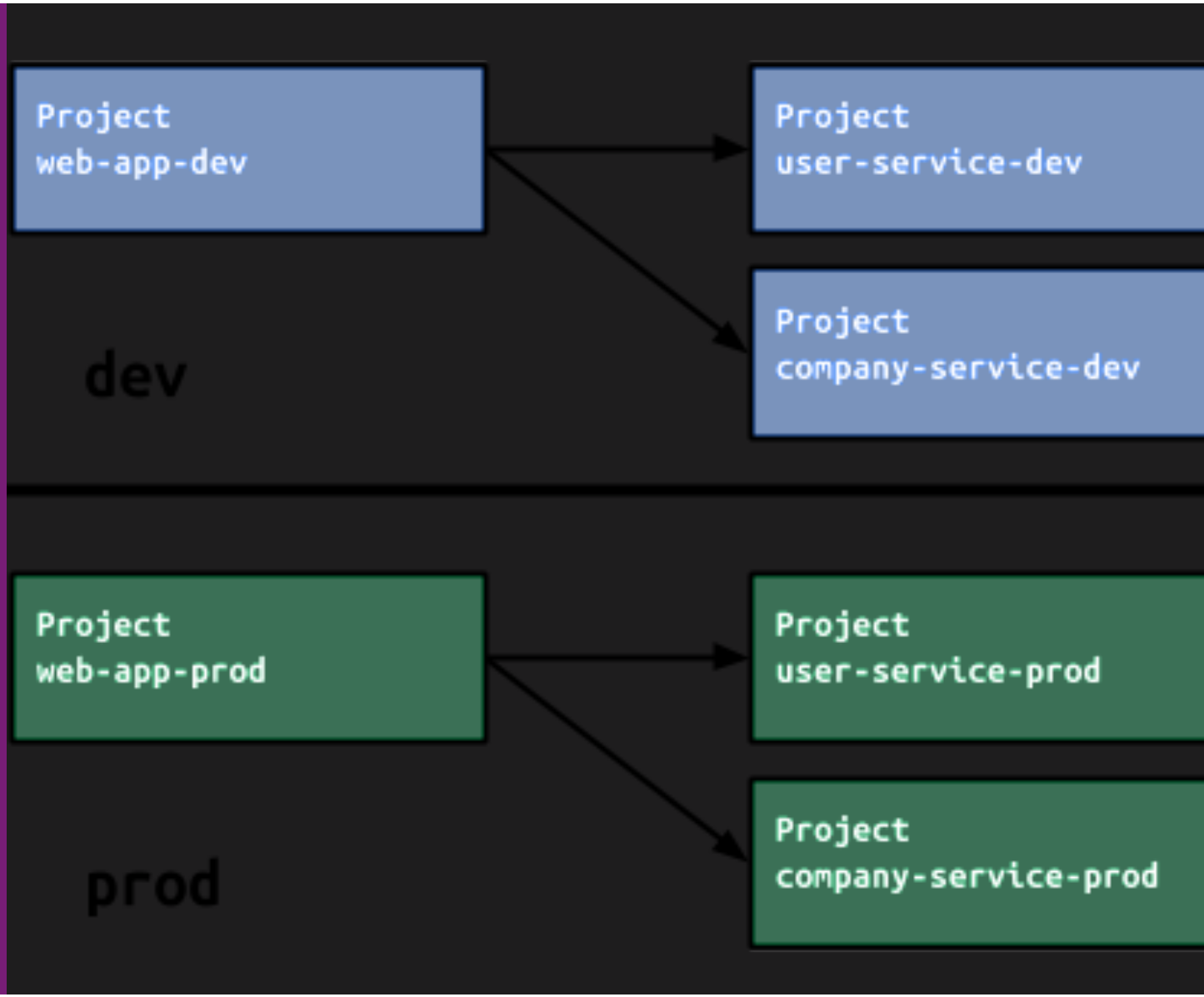
MICROSERVICES

HTTPS://CLOUD.GOOGLE.COM/APPENGINE/DOCS/STANDARD/PYTHON/MICROSERVICES-ON-APP-ENGINE

	Queue isolation, a developer convention of queue names can be employed, such as `user-service-queue-1`.	
Log isolation	Each service (and version) has independent logs, though they can be viewed together.	Each project (and service and version of each project) has independent logs, though all the logs for a given project can be viewed together. Logs across multiple projects cannot be viewed together.
Performance overhead	Services of the same project are deployed in the same datacenter, so the latency in calling one service from another by using HTTP is very low.	Projects might be deployed in different datacenters, so HTTP latencies could be higher, though still quite low because Google's network is world-class.
Cost accounting	Costs for instance-hours (the CPU and memory for running your code) are not separated for services; all the	Costs for different projects are split, making it very easy to see the cost of

MICROSERVICES

- Best practices
  - Environments for development vs. production



MIGRATING TO MICROSERVICES

MIGRATING TO MICROSERVICES

- Identify naturally segregated services
  - User or account information
  - Authorization and session management
  - Preferences or configuration settings
  - Notifications and communications services
  - Photos and media, especially metadata

MIGRATING TO MICROSERVICES

- Each service needs
  - a .yaml file
  - Be listed in the dispatch.yaml file
  - Have a webapp application

```yaml
dispatch:
  # Default service serves the typical web resources and
  - url: "*/favicon.ico"
    service: default

  # Default service serves simple hostname request.
  - url: "uchicago-cloud-photo-timeline.appspot.com/"
    service: default

  # Send all work to the one static backend.
  - url: "*/tasks/*"
    service: tasks-backend

  # Send all work to the one static backend.
  - url: "*/api/*"
    service: api-backend
```

MIGRATING TO MICROSERVICES

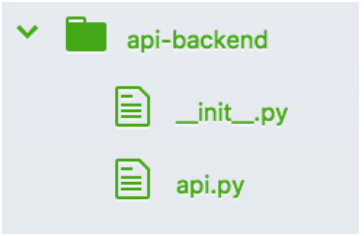
```
- url: "*/favicon.ico"
  service: default

# Default service serves simple hostname request.
- url: "uchicago-cloud-photo-timeline.appspot.com/"
  service: default

# Send all work to the one static backend.
- url: "*/tasks/*"
  service: tasks-backend

# Send all work to the one static backend.
- url: "*/api/*"
  service: api-backend
```

ANYTHING THAT MATCHES API WILL BE ROUTED TO THE API-BACKEND SERVICE

MIGRATING TO MICROSERVICES

- Yaml file describes this service
- Provides the handler to the web app for this service
  - Can be in module for project organization

```yaml
# api-backend.yaml

service: api-backend
runtime: python27
threadsafe: true

handlers:
- url: /.*
  script: api-backend.api.app
```



MIGRATING TO MICROSERVICES

- Webapp file looks the same
- Notice that you still need
  - /api/

```
# api.py

import cgi
import datetime
import urllib
import webapp2
import json
import logging

from google.appengine.api import taskqueue
from google.appengine.api import mail

from default.models import *

class MainPage(webapp2.RequestHandler):

    def get(self):
        self.response.headers['Content-Type'] = 'text/plain'
        self.response.write('API!')

app = webapp2.WSGIApplication([
    ('/api/', MainPage),
], debug=True)
```

MIGRATING TO MICROSERVICES

```
# Development server
dev_appserver.py app.yaml dispatch.yaml tasks-backend.yaml
api-backend.yaml

# Deploy
gcloud app deploy app.yaml tasks-backend.yaml api-
backend.yaml dispatch.yaml
```

MIGRATING TO MICROSERVICES

NEED TO TIGHTEN UP YOUR ROUTING RULES FOR "STRONGER CONTRACTS"

OR
SUBDOMAIN ROUTING

```
# Service available at

https://project-name.appspot.com/api/
```

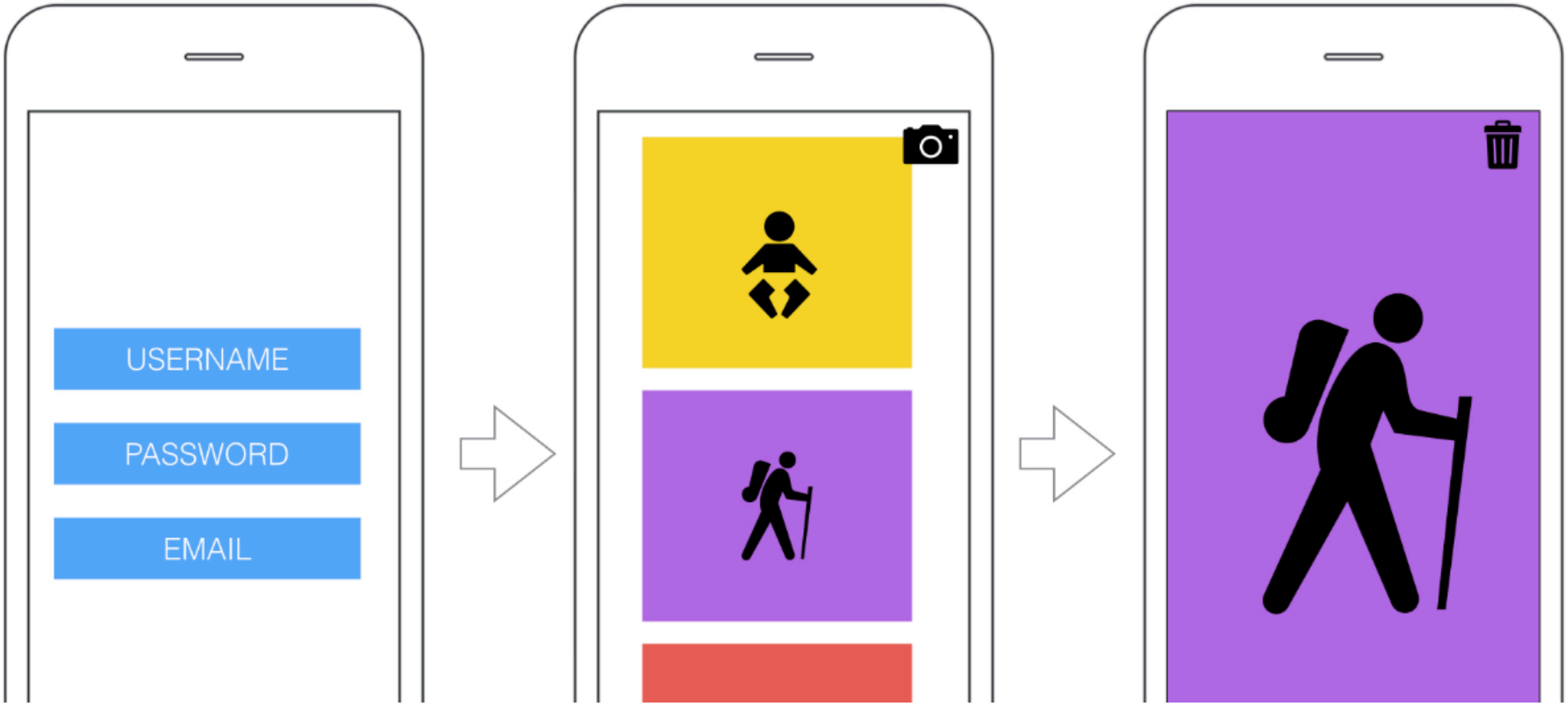
MIGRATING TO MICROSERVICES

- Don't forget that you need to update the targets for some services

ASSIGNMENT 3

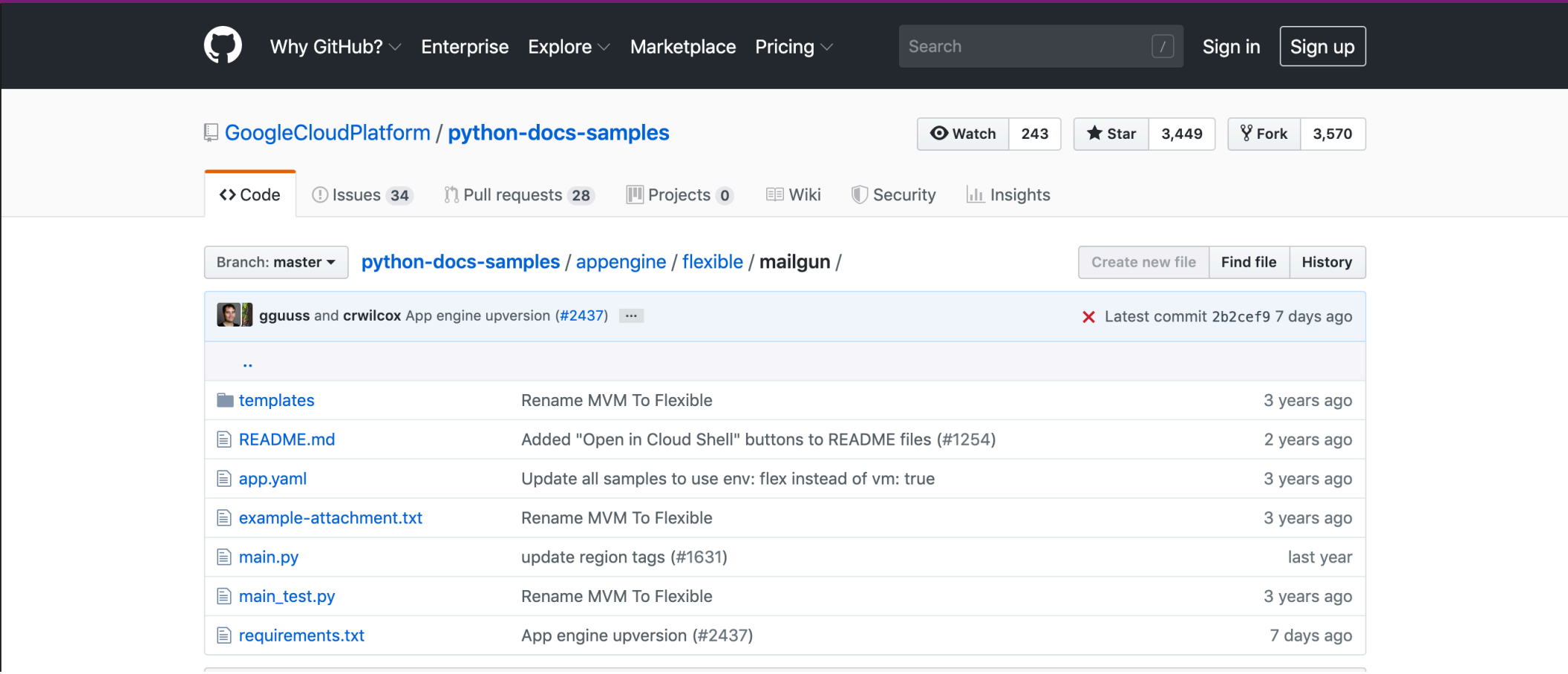
ASSIGNMENT 3

HTTP://UCHICAGO.CLOUD/SESSIONS/SESSION3/



ASSIGNMENT 2

HTTP://UCHICAGO.CLOUD/SESSIONS/SESSION3/



THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2019 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS