



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

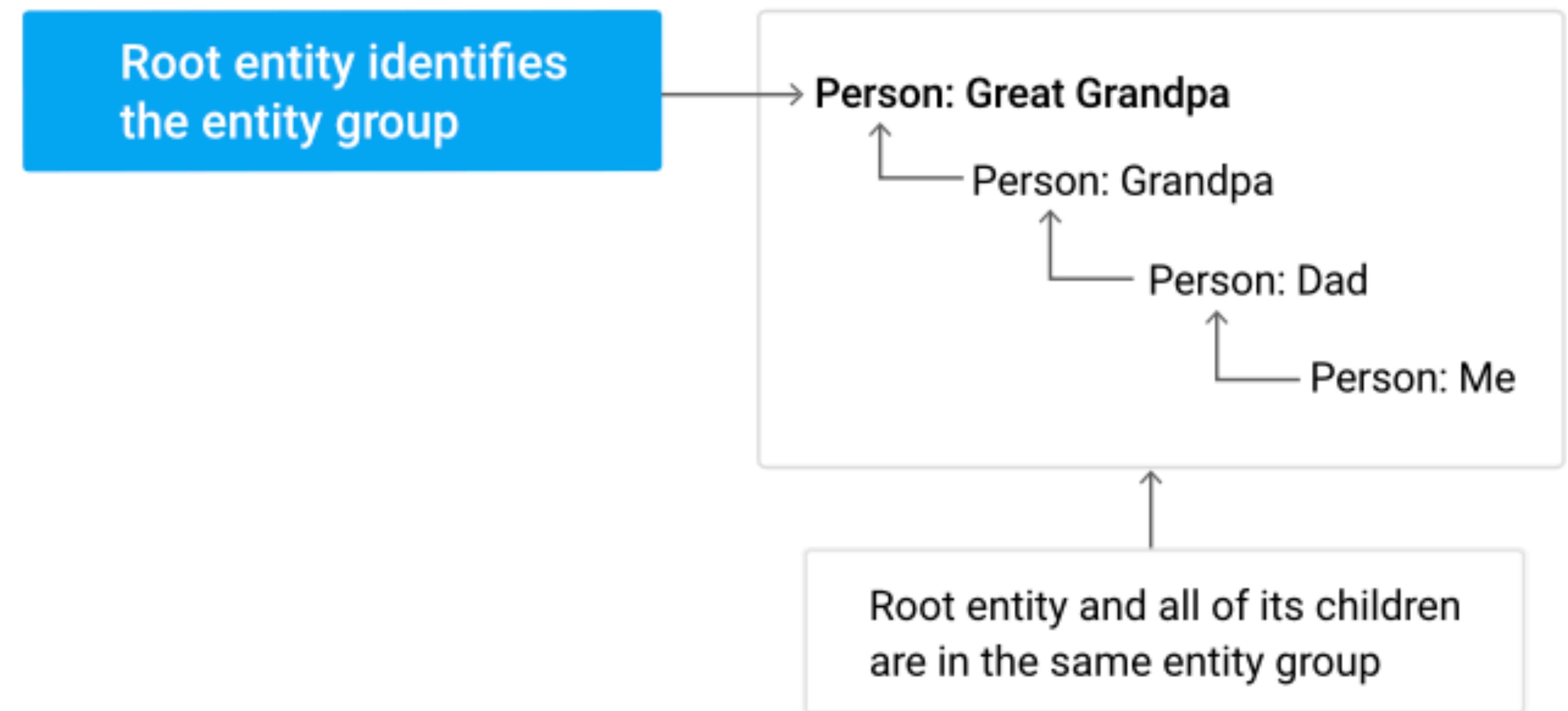
PARENT JUST DON'T
UNDERSTAND

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



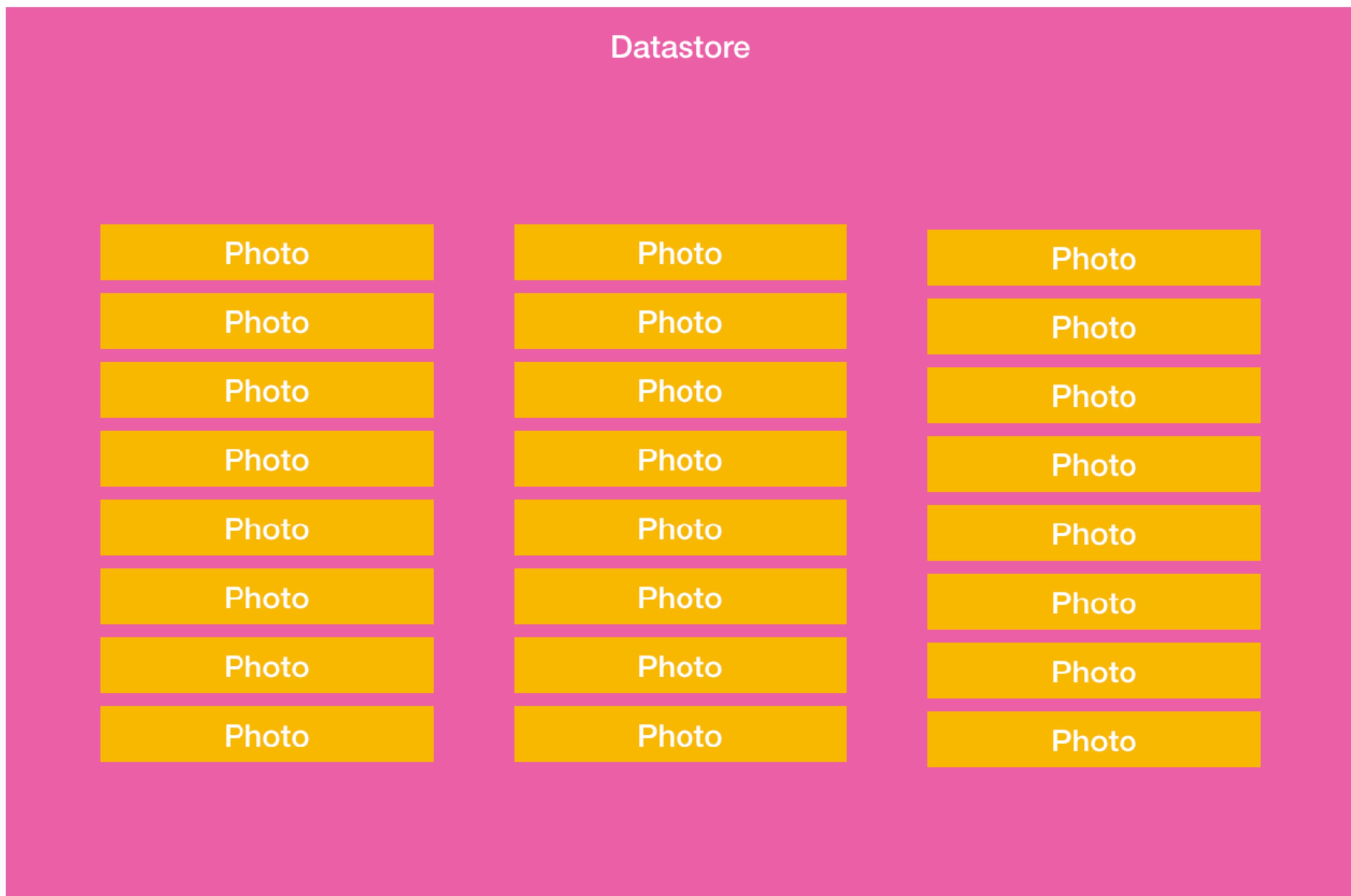
PARENT JUST DON'T UNDERSTAND

- Parent child relationship form a hierarchical structure



PARENT JUST DON'T UNDERSTAND

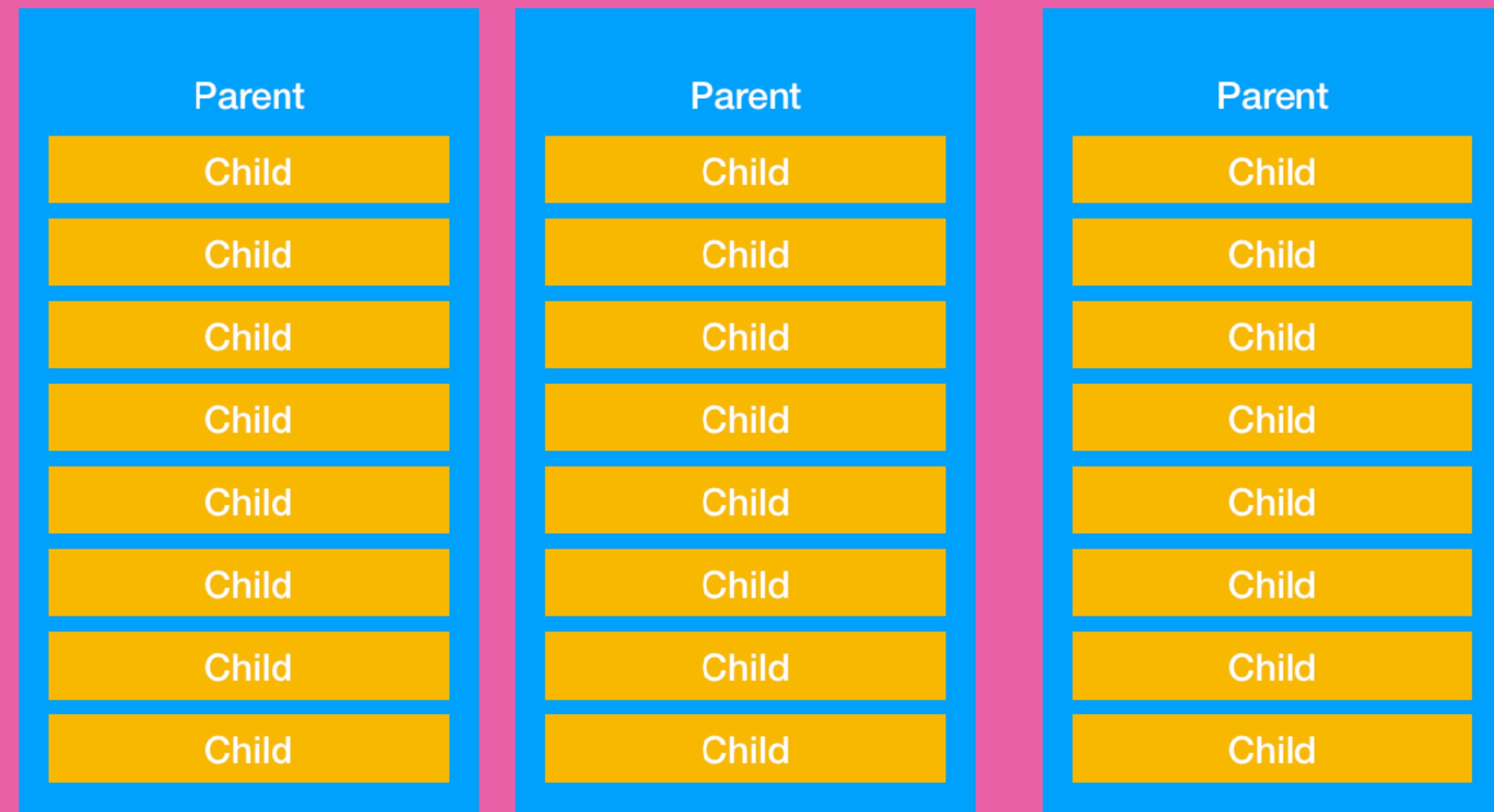
- Datastore is NOT a relational database
- To query all photos by a user would require to go through all entities



PARENT JUST DON'T UNDERSTAND

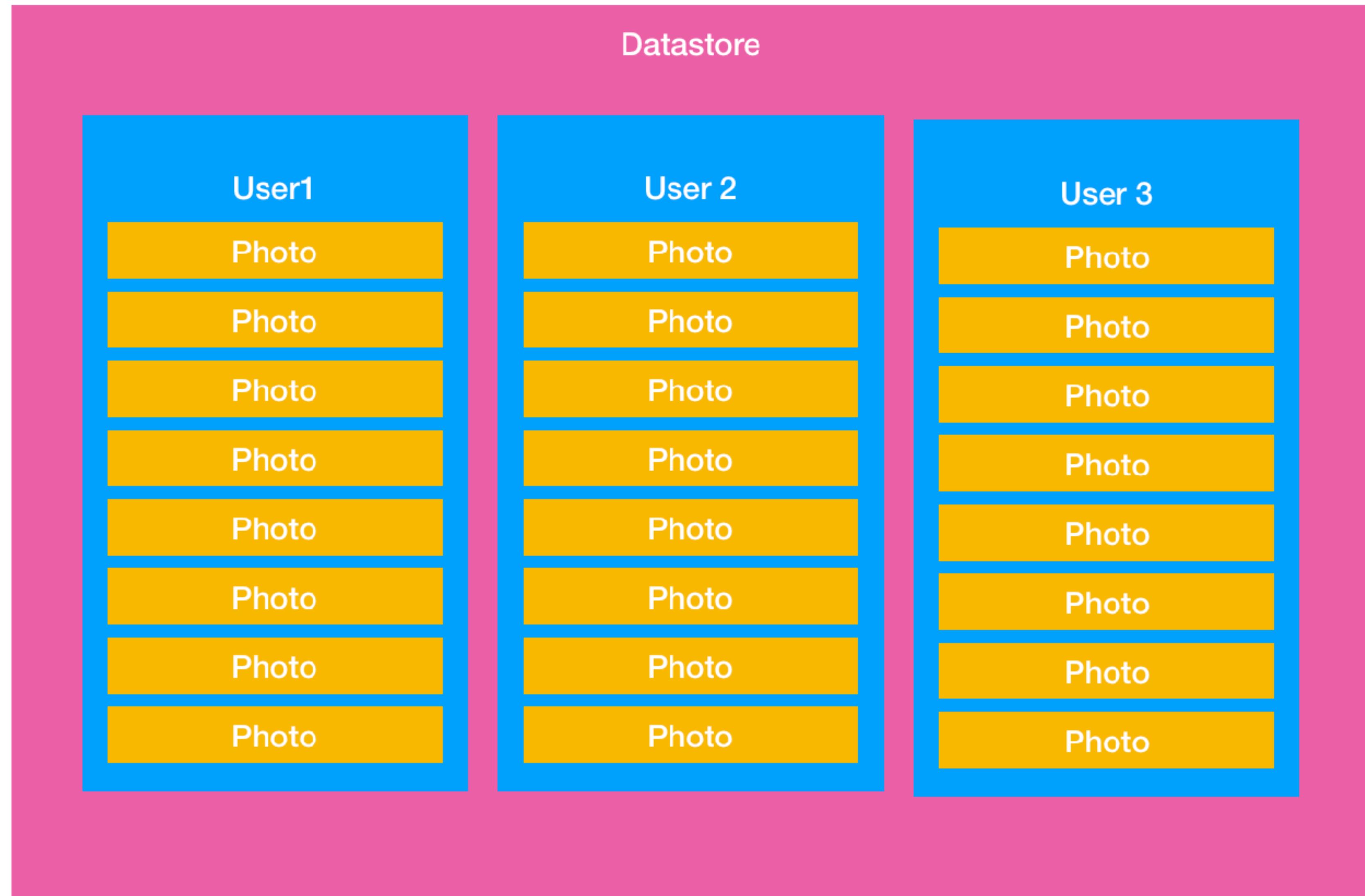
- Parent/child relationship can help organize data
- Child has a reference to parent

Datastore



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](https://cloud.google.com/appengine/docs/standard/python/datastore/data-consistency)

Data Consistency in Cloud Datastore

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 occ

In an eventually consistent query, the indexes used to gather the res
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

Ancestor 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

```
def json_results(self,photos):  
    """Return formatted json from the datastore query"""\n    json_array = []\n    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})
```



```
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 form  
        # the user from the textbox. If the post  
        # API then the username will be embedded  
        if self.request.get('user'):   
            user = self.request.get('user')  
  
        # Be nice to our quotas  
        thumbnail = images.resize(self.request.get('image'))  
  
        # Create and add a new Photo entity  
        #  
        # We set a parent key on the 'Photos'  
        # in the same entity group. Queries across  
        # will be consistent. However, the writes  
        # ~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  
        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

Data Consistency in Cloud Datastore

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 occ

In an eventually consistent query, the indexes used to gather the res
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 are 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 many nodes using masterless, synchronous replication over a wide geographic area. However, there is a tradeoff in that it is that the write throughput for any single *entity group* is limited to about one commit per second, and there are limitations on queries or transactions that span multiple entity groups. This page describes these limitations in detail and discusses best practices for structuring your data to support strong consistency while still meeting your application's write throughput requirements.

Strongly-consistent reads always return current data, and, if performed within a transaction, will appear to be a single, consistent snapshot. However, queries must specify an ancestor filter in order to be strongly-consistent. Transactions can participate in a transaction, and transactions can involve at most 25 entity groups. Eventually-consistent reads do not have those limitations, and are adequate in many cases. Using eventually-consistent reads can allow you to distribute your data among a larger number of entity groups, enabling you to obtain greater write throughput by executing writes in parallel on the different entity groups. But, you need to understand the characteristics of eventually-consistent reads in 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 these reads ensure that the replica they are running on is up-to-date. Instead, they use whatever data is available 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 entities and not others. Note, though, that a transaction will never appear to have been partially applied within a single entity.
- The query results can include entities that should not have been included according to the filter criteria, or exclude entities that should have been included. This can occur because indexes might be read at an earlier version than the entity itself is read at.

To understand how to structure your data for strong consistency, compare two different approaches from the [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 are 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 many nodes using masterless, synchronous replication over a wide geographic area. However, there is a tradeoff in that there is that the write throughput for any single *entity group* is limited to about one commit per second, and there are limitations on queries or transactions that span multiple entity groups. This page describes these limitations in detail and discusses best practices for structuring your data to support strong consistency while still meeting your application's write throughput requirements.

Strongly-consistent reads, performed within a transaction, will appear to be single, consistent snapshots. Ancestor queries, which can participate in a transaction, also have those limitations. Sharding your data among a large number of entities, in parallel on the different shards, in order to determine which shard to query,

USE MEMEACHE TO STORE PENDING WRITES

- The results of a query, ensure that the data at the time of query was committed.
- A committed transaction, and not others, can affect the entity.
- The query results might exclude entities that should have been included according to the filter criteria.

performed within a transaction, will appear to be single, consistent snapshots. Ancestor queries, which can participate in a transaction, also have those limitations. Sharding your data among a large number of entities, in parallel on the different shards, in order to determine which shard to query, will appear to be single, consistent snapshots. Ancestor queries, which can participate in a transaction, also have those limitations. Eventually-consistent reads can allow you to obtain greater write throughput by executing them outside of a transaction. To understand the characteristics of eventually-consistent reads, see the section:

transactions. This can occur because these writes are not atomic. Instead, they use whatever data is available at the time of the query, which is almost always less than a few seconds.

might appear to have been applied to some entities, and not others. Entities might appear to have been partially applied with different values.

have been included according to the filter criteria. This can occur because indexes might be read at a different version than the entity itself is read at.

To understand how to structure your data for strong consistency, compare two different approaches from the [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 initiated

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

- 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

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

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](#)

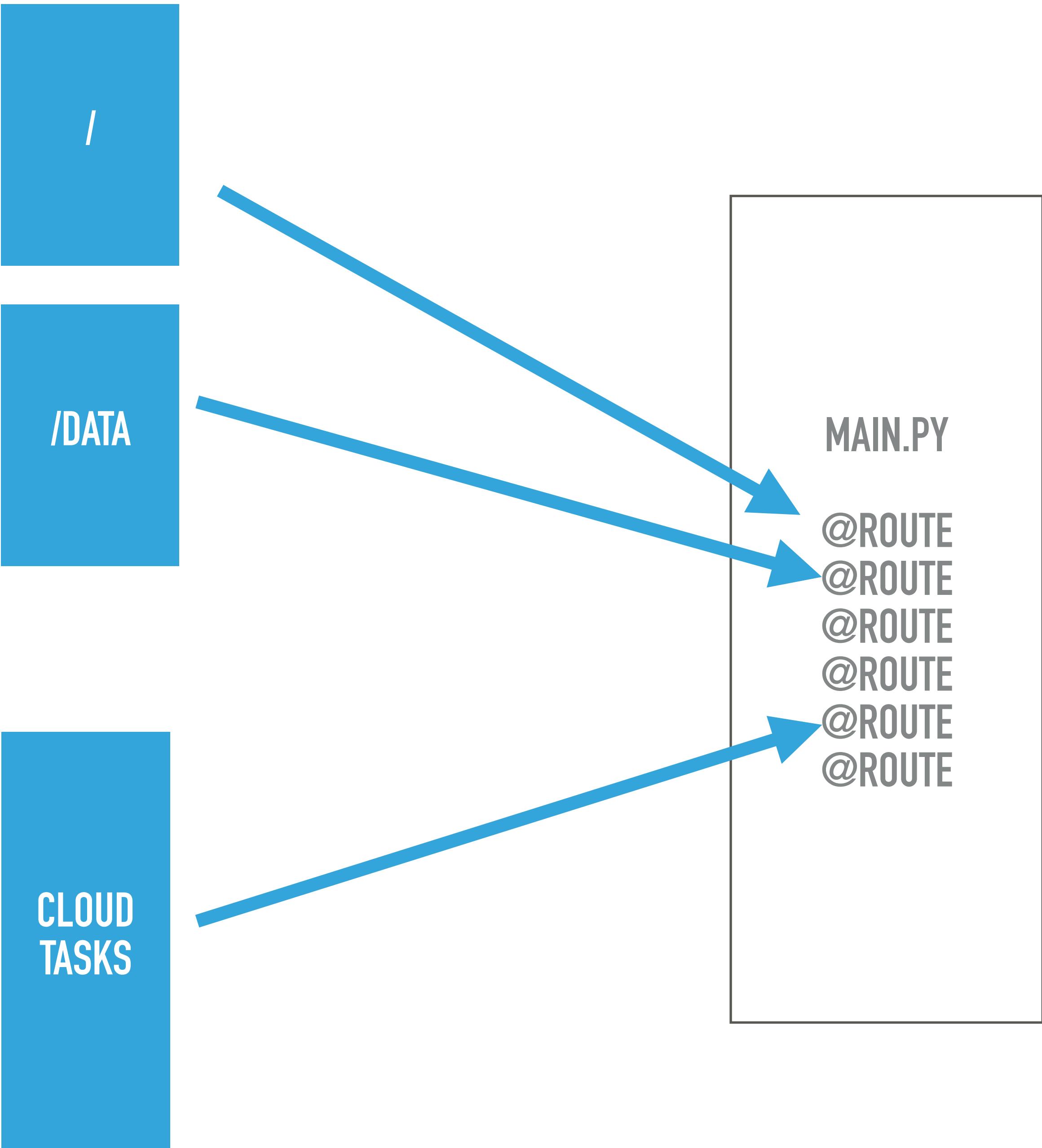
Before you begin

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](#).



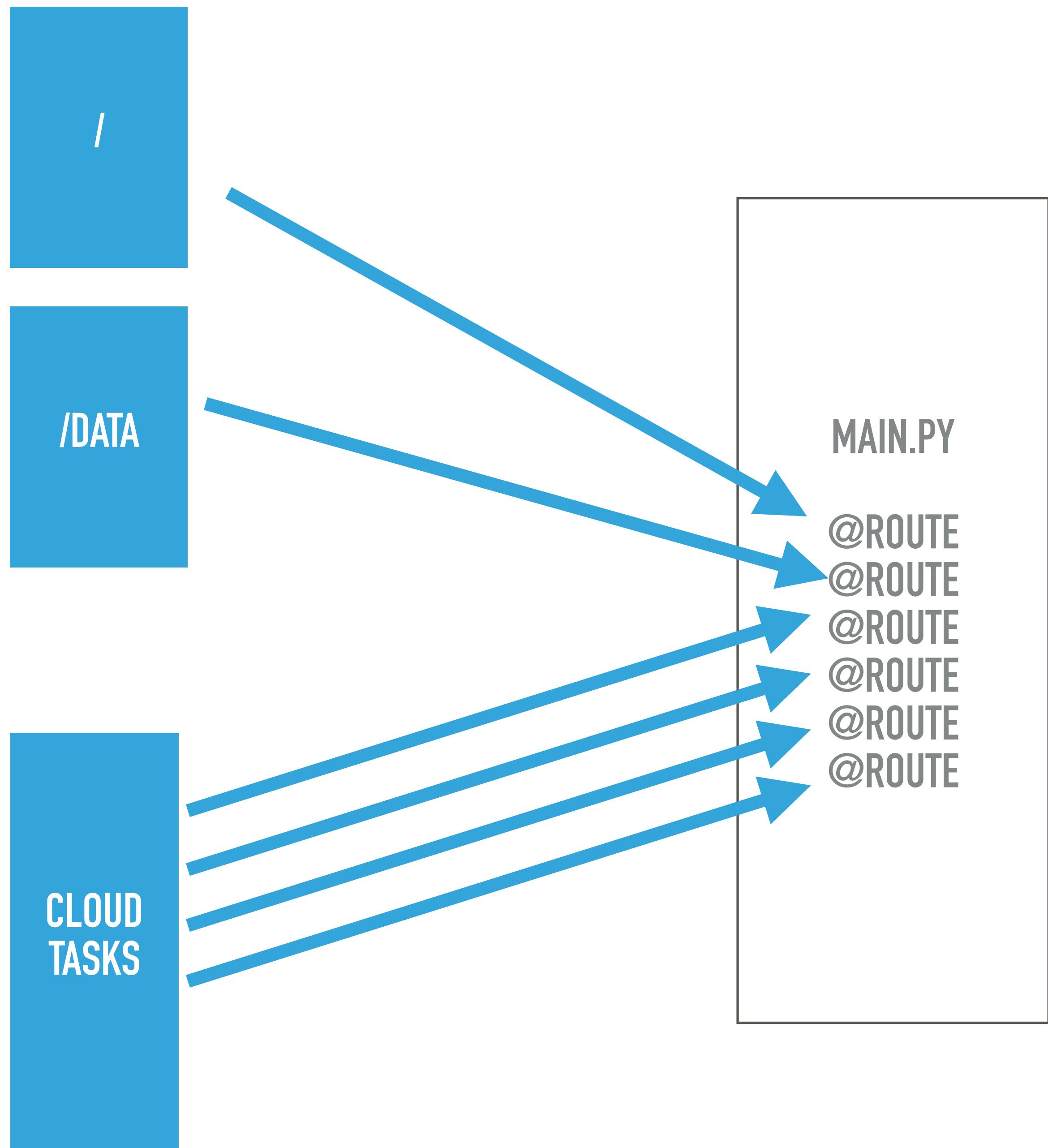
CLOUD TASKS

- 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 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 ■ DISABLE API

Overview Metrics Quotas Credentials

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

Traffic by response code

Request/sec (2 hr average)

Response Code	Request/sec (2 hr average)
200	1.0/s
202	0.8/s
204	0.6/s
404	0.4/s
500	0.2/s
503	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

Tutorials and documentation

Quickstart
Overview
Try in API Explorer

View metrics

This screenshot shows the Google Cloud Platform Cloud Tasks API Overview page. The left sidebar lists 'Cloud Tasks API' under 'APIs & Services'. The main content area has tabs for 'Overview' (selected), 'Metrics', 'Quotas', and 'Credentials'. The 'Overview' tab displays 'Details' (Name: Cloud Tasks API, By: Google, Service name: cloudtasks.googleapis.com) and an 'Overview' section stating it manages distributed requests. It also shows an 'Activation status' as 'Enabled'. To the right is a chart titled 'Traffic by response code' showing request rates over time. Below the chart is a 'Tutorials and documentation' section with links to 'Quickstart', 'Overview', and 'Try in API Explorer'.

CLOUD TASKS

- Create a task queue

```
tabinkowski:mpcs51033-2019-autumn-code/photo-timeline-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>

Apps Last tabs

☰ Google Cloud Platform photo-timeline-python3

Cloud Tasks REFRESH

App Engine queues

Queue Name	Tasks In Queue	Completed In Last Min.	Oldest Task ETA	Maximum Rate	Enforced Rate	Bucket Size	Maximum C
photo-timeline-queue	0	0		500/s		100	

CLOUD TASKS

- Describe queue

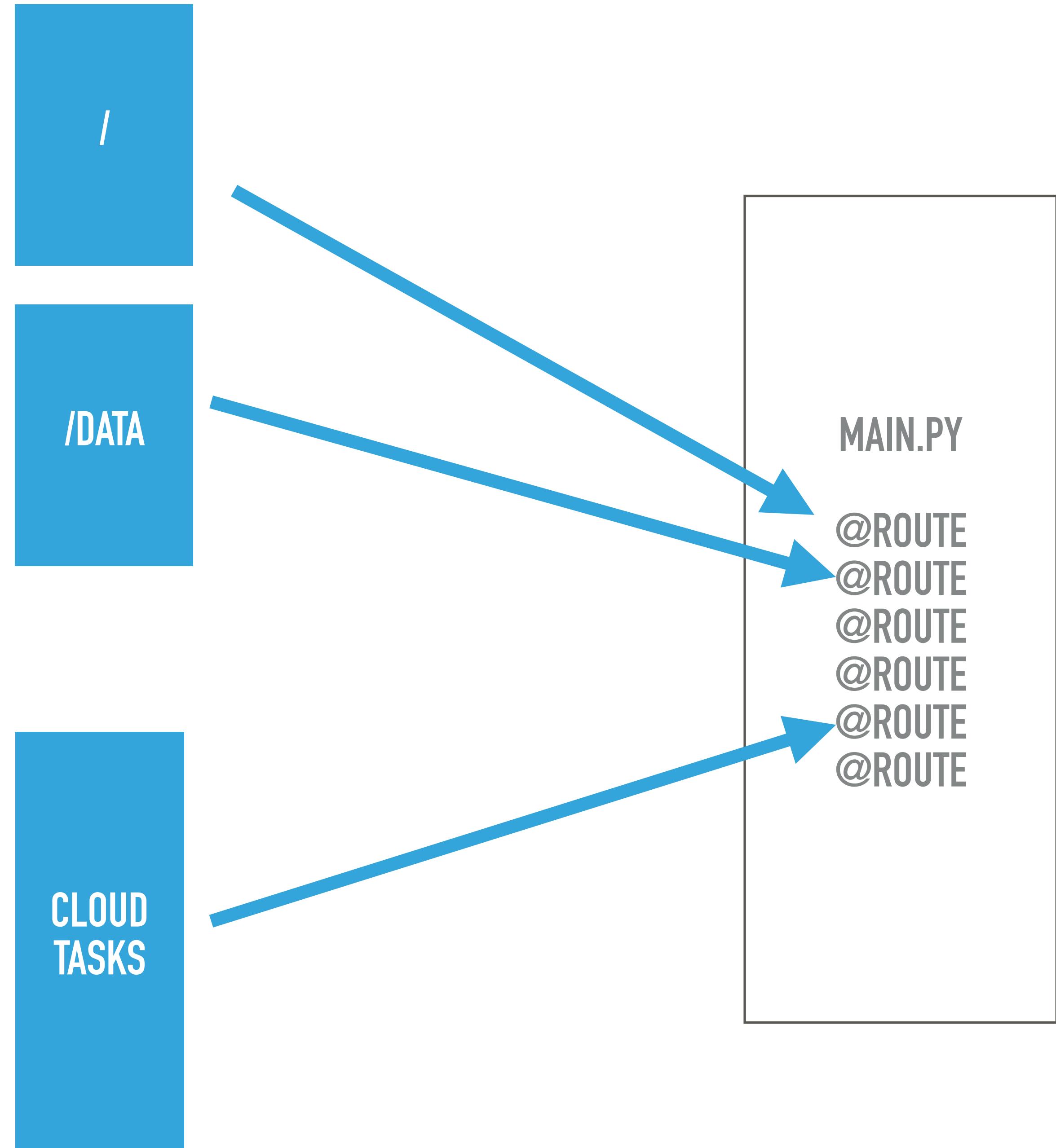
```
578 % gcloud tasks queues describe photo-timeline-queue
name: projects/photo-timeline-python3/locations/us-central1/queues/p
hoto-timeline-queue
rateLimits:
  maxBurstSize: 100
  maxConcurrentDispatches: 1000
  maxDispatchesPerSecond: 500.0
retryConfig:
  maxAttempts: 100
  maxBackoff: 3600s
  maxDoublings: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

```
# 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

```
# 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

```
# 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

Logs Viewer

Logs-based metrics

Exports

Logs ingestion

CREATE METRIC CREATE EXPORT SAVE SEARCH ⌂ ⌂ SHOW LIBRARY

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

Time	Message
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 down
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[20191015t121528] payload: This is a task.
2019-10-15 17:24:25 default[20191015t121528] "POST /example_task_handler HTTP/1.1" 200
2019-10-15 17:24:25 default[20191015t121528] Received task with 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

≡ Google Cloud Platform

✿ photo-timeline-python3 ▾

Datastore

Entities [+ CREATE ENTITY](#)

QUERY BY KIND

QUERY BY

Kind

Task

PUT IN MY OWN LOGS IN
DATASTORE

```
@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)
```

<input type="checkbox"/>	Name/ID 	timestamp	type
<input type="checkbox"/>	id=5629654941564928	2019-10-14 (00:29:33.074) CDT	-
<input type="checkbox"/>	id=5639601401712640	2019-10-14 (20:27:00.944) CDT	cron
<input type="checkbox"/>	id=5639601401712640	2019-10-14 (15:24:00.748) CDT	cron
<input type="checkbox"/>	id=5644523313037312	2019-10-14 (20:23:00.657) CDT	cron
<input type="checkbox"/>	id=5658646574792704	2019-10-15 (11:57:21.639) CDT	cloud task
<input type="checkbox"/>	id=5665673409724416	2019-10-14 (20:31:00.184) CDT	cron

DELETE A TASK

CLOUD TASKS

Google Cloud Platform photo-timeline-python3 ▾

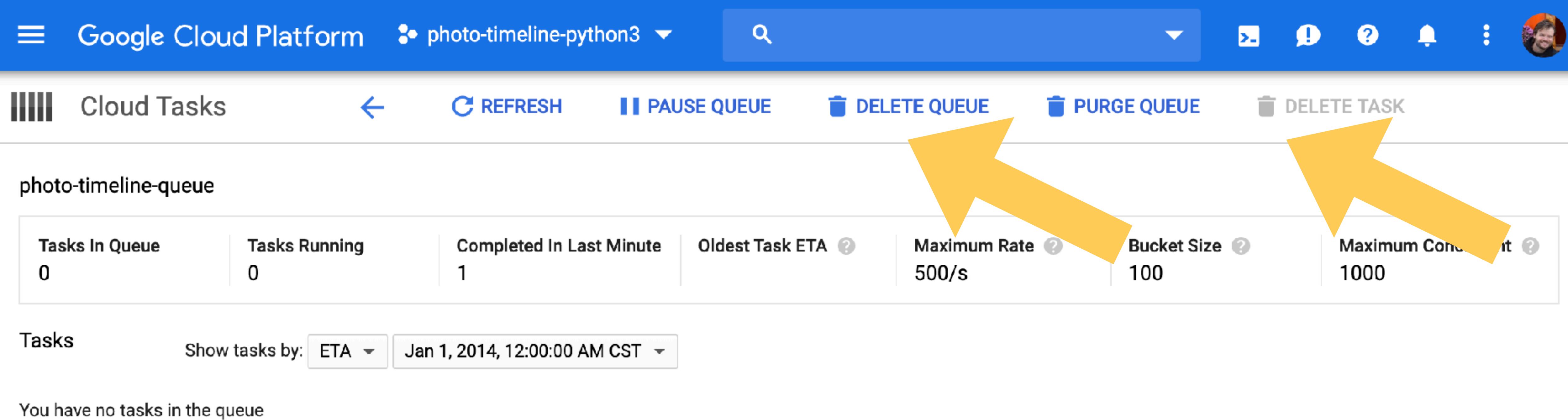
Cloud Tasks ← REFRESH PAUSE QUEUE DELETE QUEUE PURGE QUEUE DELETE TASK

photo-timeline-queue

Tasks In Queue	Tasks Running	Completed In Last Minute	Oldest Task ETA	Maximum Rate	Bucket Size	Maximum Concurrent
0	0	1		500/s	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)
```

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 Task Queue API lets applications perform work, called *tasks*, asynchronously outside of a user request. When an app needs to execute work in the background, it adds tasks to *task queues*. The tasks are then executed later, by scalable App Engine worker services in your application.

Push queues and pull queues

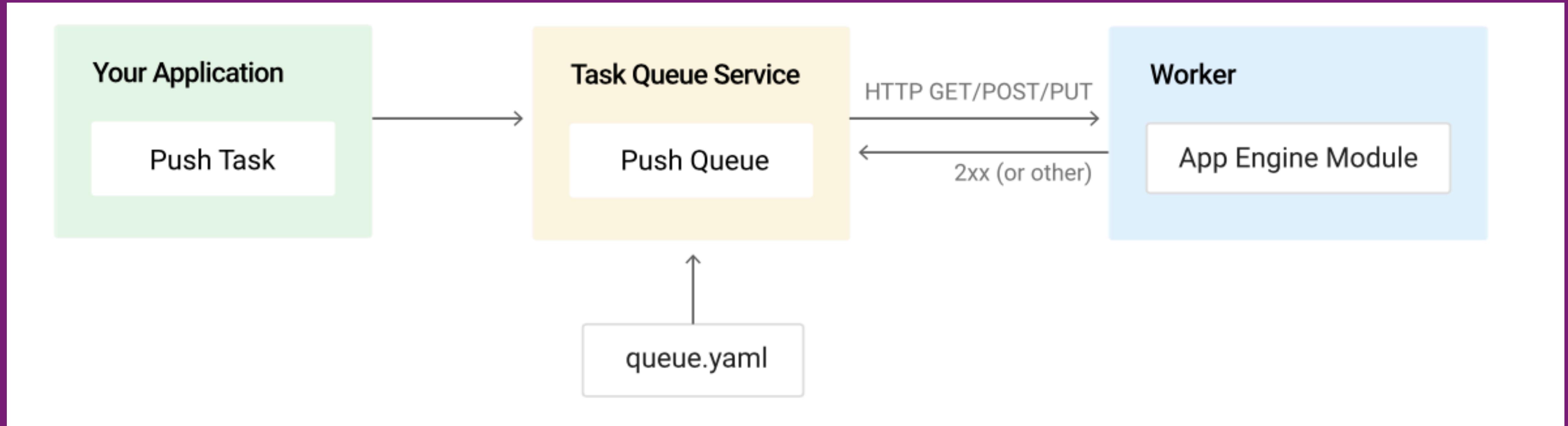
Task queues come in two flavors, *push* and *pull*. The manner in which the Task Queue service dispatches task requests to worker services is different for the different queue types.

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

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

PUSH QUEUE

PUSH QUEUES



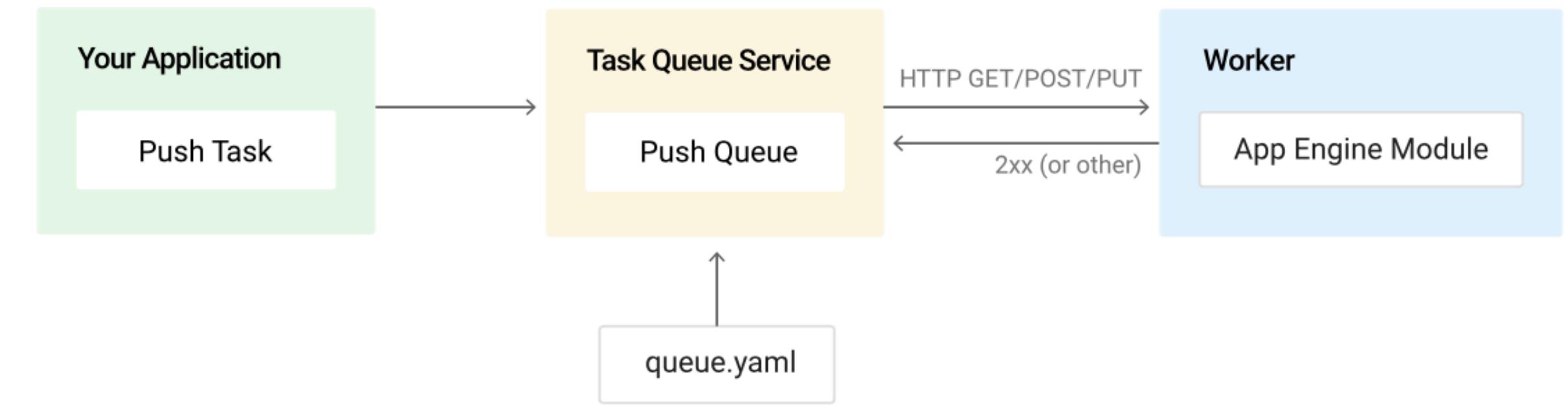
- Push queues run tasks by delivering HTTP requests to App Engine worker services

TASK QUEUE

- Push queue
 - 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

PUSH QUEUES

- 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



PUSH QUEUES

- You must write a handler for every kind of task you use
- A single service can have multiple handlers for different kinds of tasks,
 - You can use different services for different task types

PUSH QUEUES

- Requirements
 - Define a queue (or be content with default)
 - Write a handler to process a task request
 - Create tasks and add them to the queue

PUSH QUEUES

CREATE A QUEUE

- You can have a maximum of 10 queues (not including the default queue)
- Optional parameters help to control quotas

*e refresh rate of the default queue from 5
lt*

r-a

: 40

ent_requests: 10

r-b

PUSH QUEUES

CREATE A TASK

- Create and add a task to the "worker" queue

```
task = taskqueue.add(  
    url='/count_pictures',  
    target='worker',  
    params={'amount': amount})
```

PUSH QUEUES

CREATE A TASK

```
taskqueue.add(method=GET, url='/update-counter?key=blue', target='worker')
taskqueue.add(url='/update-counter', params={'key': 'blue'}, target='worker')
taskqueue.add(url='/update-counter', payload="{'key': 'blue'}", target='worker')
```

- Different ways of passing parameters to the task
- No difference in behavior

PUSH QUEUES

CREATE A TASK HANDLER

- Create a class to handle the task and a handler to route it

```
class EmailTaskHandler(webapp2.RequestHandler):
    """Handler for task queue emails"""

    def post(self):
        message = self.request.get('message', default_value='default')
        logging.info('This is an info message')
        mail.send_mail(sender="me@uchicago-mobi-photo-timeline.appspotmail.com",
                      to="abinkowski@uchicago.edu",
                      subject="New Photo!",
                      body="A new photo has been uploaded to your account.")

#####
#####
#####
#####
```

```
webapp2.WSGIApplication([
    '/', HomeHandler],
    [('/email_task/', EmailTaskHandler),
     app2.Route('/logging/', handler=LoggingHandler),
     app2.Route('/image/<key>/', handler=ImageHandler),
     app2.Route('/post/<user>/', handler=PostHandler),
     app2.Route('/user/<user>/<type>/', handler=UserHandler)

], debug=True)
```

PUSH QUEUES

CREATE A TASK HANDLER

- Putting it all together
- You can test by directly accessing the URL

```
task = taskqueue.add(  
    url='/email_task/',  
    params={'message': 'hi'}  
)  
logging.debug('Task {} enqueue, ETA {}'.format(task.name, task.eta))  
  
# Redirect to print out JSON  
self.redirect('/user/%s/json/' % user)  
  
  
class LoggingHandler(webapp2.RequestHandler):  
  
class EmailTaskHandler(webapp2.RequestHandler):  
    """Handler for task queue emails"""  
  
    def post(self):  
        message = self.request.get('message', default_value='default')  
        logging.info('This is an info message')  
        mail.send_mail(sender="me@uchicago-mobi-photo-timeline.appspotmail.com",  
                      to="abinkowski@uchicago.edu",  
                      subject="New Photo!",  
                      body="A new photo has been uploaded to your account.")  
  
#####  
#  
#####  
app = webapp2.WSGIApplication([  
    ('/', HomeHandler),  
    ('/email_task/', EmailTaskHandler),  
    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)
```

PUSH QUEUES

CREATE A TASK HANDLER

```
class LoggingHandler(webapp2.RequestHandler):  
  
    def get(self):  
        logging.info('This is an info message')  
  
class EmailTaskHandler(webapp2.RequestHandler):  
    """Handler for task queue emails"""\n  
    def post(self):  
        message = self.request.get('message', default_value='default message')  
        logging.info('This is an info message')  
        mail.send_mail(sender="me@uchicago-mobi-photo-timeline.appspot.com",  
                      to="abinkowski@uchicago.edu",  
                      subject="New Photo!",  
                      body="A new photo has been uploaded to your account.")  
  
#####  
#  
#####  
app = webapp2.WSGIApplication([
```

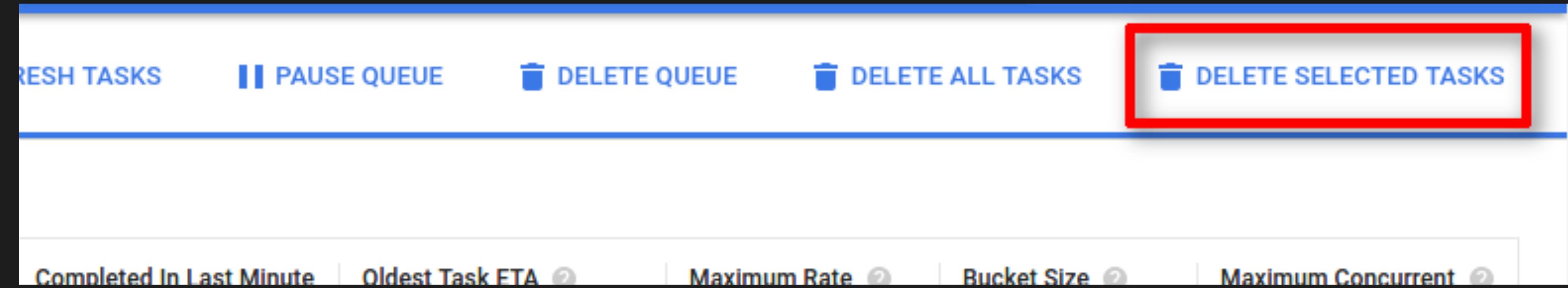
DEMO

PUSH QUEUES

DELETE TASK

```
from google.appengine.api import taskqueue  
  
# Delete an individual task...  
q = taskqueue.Queue('queue1')  
q.delete_tasks(taskqueue.Task(name='foo'))
```

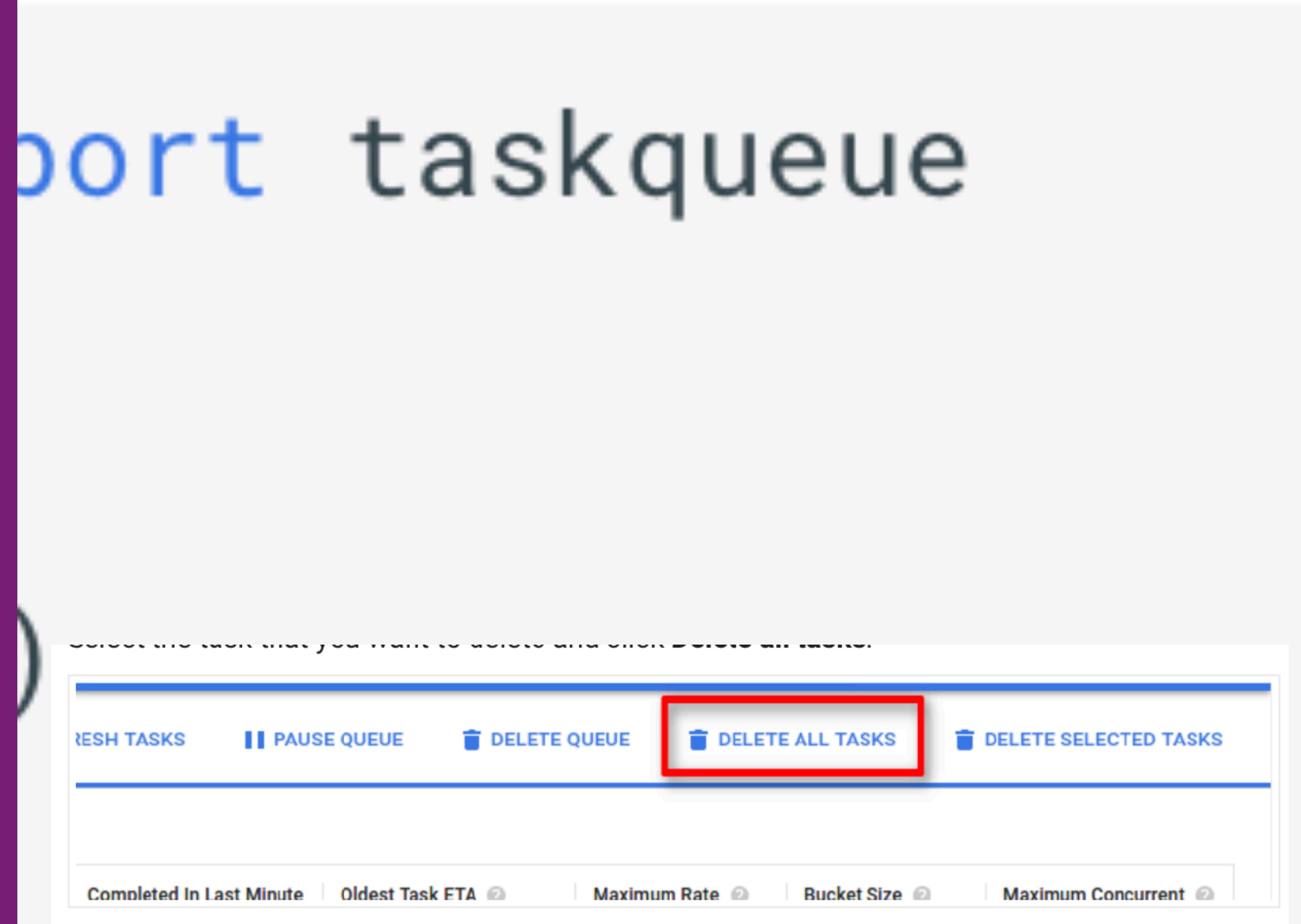
- Delete a task from a queue



PUSH QUEUES

DELETE TASK

- Delete all tasks from a queue



PUSH QUEUES

DEFERRED TASK

- Submit ad-hoc tasks with deferred
- No difference in how its run compared to push queue

Contents ▾

Introduction

Example: A datastore mapper

Using the mapper

Deferred tips and tricks

...

Nick Johnson

October 15, 2009

Introduction

Thanks to the [Task Queue API](#) released in SDK 1.2.3, it's easier than ever to do work 'offline', separate from user serving requests. In some cases, however, setting up a handler for each distinct task you want to run can be cumbersome, as can serializing and deserializing complex arguments for the task - particularly if you have many diverse but small tasks that you want to run on the queue.

Fortunately, a new library in release 1.2.5 of the SDK makes these ad-hoc tasks much easier to write and execute. This library is found in `google.appengine.ext.deferred`, and from here on in we'll refer to it as the 'deferred' library. The deferred library lets you bypass all the work of setting up dedicated task handlers and serializing and deserializing your parameters by



PUSH QUEUES

DEFERRED TASKS

```
from google.appengine.ext import deferred

def do_something_expensive(a, b, c=None):
    logging.info("Doing something expensive!")
    # Do your work here

# Somewhere else
deferred.defer(do_something_expensive, "Hello, world!", 42, c=True)
```

- Run in the default queue

PUSH QUEUES

DEFERRED TASK

```
12     from google.appengine.api import taskqueue  
13  
14     from models import *  
15  
16  
17     from google.appengine.ext import deferred  
18  
19     def do_something_expensive(a, b, c=None):  
20         logging.info("Doing something expensive!")  
21         # Do your work here  
22  
23 #####  
24 """The home page of the app"""  
25 class HomeHandler(webapp2.RequestHandler):  
26  
27     """Show the webform when the user is on the home page"""  
28     def get(self):  
29         self.response.out.write('<html><body>')  
30  
31         # Print out some stats on caching  
32         stats = memcache.get_stats()  
33         self.response.write('<b>Cache Hits:{}</b><br>'.format(stats['hits']))  
34         self.response.write('<b>Cache Misses:{}</b><br>'.format(stats['misses']))  
35  
36  
37         user = self.request.get('user')  
38         ancestor_key = ndb.Key("User", user or "*notitle*")  
39         # Query the datastore
```

DEMO

PULL QUEUE

PULL QUEUE

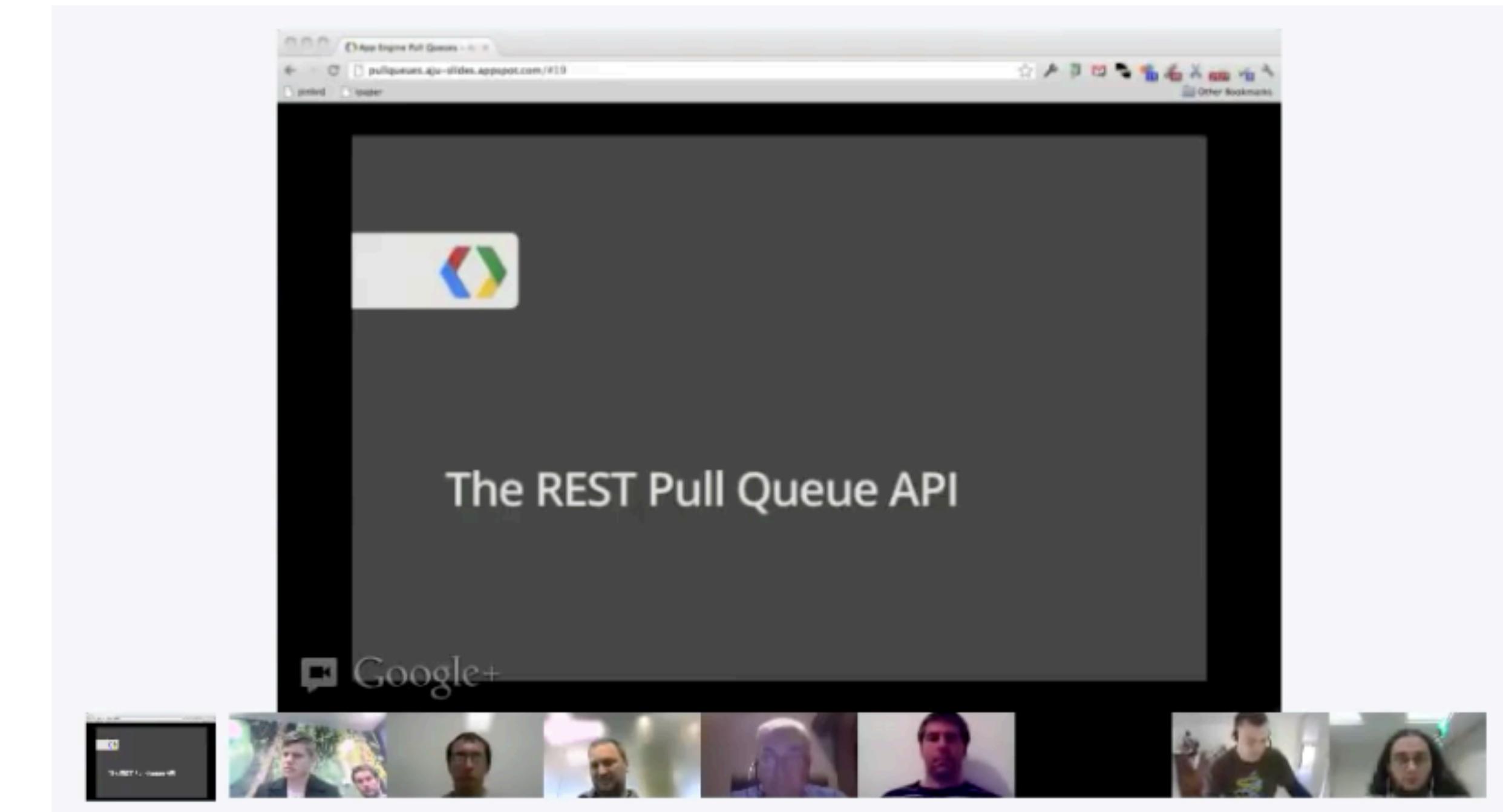
- Pull queues allow a task consumer to process tasks outside of App Engine's default task processing system
 - More customizable
- If the task consumer is a part of your App Engine app, you can manipulate tasks using simple API calls from the `google.appengine.api.taskqueue` module
- Task consumers outside of App Engine can pull tasks using the Task Queue REST API
 - More interoperability with outside services

PULL QUEUE

- Pull queue
 - Do not dispatch tasks at all
 - They depend on other worker services to "lease" tasks from the queue on their own initiative
 - Pull queues give you more power and flexibility over when and where tasks are processed, but they also require you to do more queue management
 - Tasks must complete in the specified deadline or it will be expire

TASK QUEUE

- Google Hangout video with developer relations
 - <https://cloud.google.com/appengine/docs/standard/python/taskqueue/overview-pull#>



TASK QUEUE

- In practice, start with push queues and consider pull queues when you need more control

[SEND FEEDBACK](#)

Using Pull Queues in Python

Contents ▾

- Pull queue overview
- Pulling tasks within App Engine
 - Defining pull queues
 - Adding tasks to a pull queue

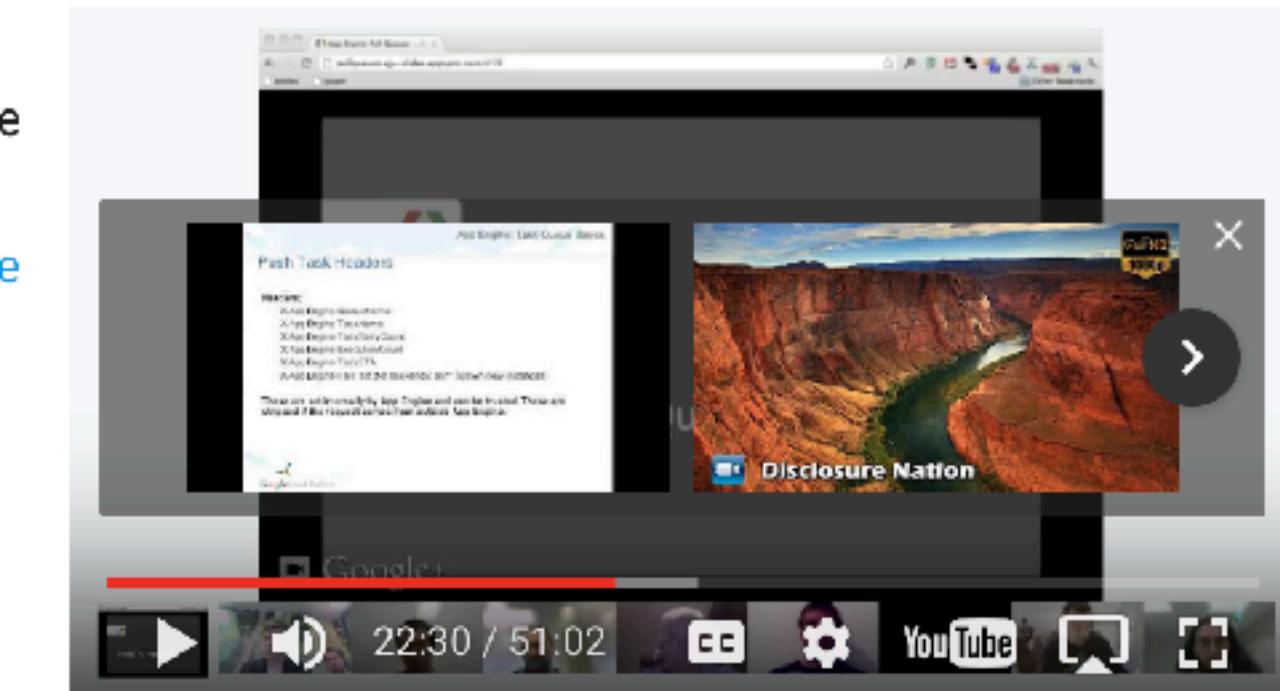
...

Pull queues allow you to design your own system to consume App Engine tasks. The task consumer can be part of your App Engine app, such as a [service](#), or a system outside of App Engine by using the [Task Queue REST API](#). The task consumer leases a specific number of tasks for a specific duration, then processes and deletes them before the lease ends.

Using pull queues requires your application to handle some functions that are automated in push queues:

- Your application needs to scale the number of workers based on processing volume. If your application does not handle scaling, you risk wasting computing resources if there are no tasks to process; you also risk latency if you have too many tasks to process.
- Your application also needs to explicitly delete tasks after processing. In push queues, App Engine deletes the tasks for you. If your application does not delete pull queue tasks after processing, another worker might re-process the task. This wastes computing resources and risks errors if tasks are not [idempotent](#).

Pull queues require a specific configuration in `queue.yaml`. For more information, see [Defining Pull Queues](#).



DEPLOY QUEUES

DEPLOY TASK QUEUES

- Cron.yaml file

```
# cron.yaml

cron:
  - description: daily summary job
    url: /tasks/summary_email/
    schedule: every 1 minutes

  - description: weekly summary job
    url: /logging/
    schedule: every 5 minutes
```

DEPLOY TASK QUEUES

```
session2-cloud-photo-timeline — bash — 103x17
...ion2-cloud-photo-timeline — python -m dev_appserver.py app.yaml ... ...-playground-private/session2-cloud-photo-timeline — bash
binkowski@Ts-MacBook-Pro ~/Google Drive/g-Teaching/uchicago.cloud/mpcs51033-2017-autumn/mpcs51033-2017-autumn-playground-private/session2-cloud-photo-timeline (email_tasks)
6 % gcloud app deploy queue.yaml --project mpcs51033-2017-autumn-photos
nfigurations to update:

scriptor:      [/Users/tabinkowski/Google Drive/g-Teaching/uchicago.cloud/mpcs51033-2017-autumn/mpcs51033-2017-autumn-playground-private/session2-cloud-photo-timeline/queue.yaml]
pe:           [task queues]
rget project: [mpcs51033-2017-autumn-photos]

you want to continue (Y/n)? Y
  dating config [queue]...done.

ask queues have been updated.
```

Deploy task queue

DEPLOY TASK QUEUES

≡ Google Cloud Platform mpcs51033-2017-autum... ▾

Task queues REFRESH

Push Queues Pull Queues Cron Jobs

A cron job is a scheduled task that runs at a specific time or at regular intervals.

Cron job ^	Description	Frequency	Last run	Status	Log
/logging/	weekly summary job	every 5 minutes (GMT)		Has not run yet	View Run now
/tasks/summary_email/	daily summary job	every 1 minutes (GMT)		Has not run yet	View Run now

Task queues

- Dashboard
- Services
- Versions
- Instances
- Task queues
- Security scans
- Firewall rules
- Quotas
- Blobstore
- Memcache

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 operate at regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are automatically triggered by the App Engine Cron Service. For instance, you might use a cron job to send out an email every day, or to update some cached data every 10 minutes, or refresh summary information once a week.

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job runs under the 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.yaml`).
2. Add one or more `<cron>` entries to your file and define the necessary elements for each entry, including 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 operate at regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are automatically triggered by 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 day.

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job runs under the 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.yaml`).
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:

```
crontab:  
- 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 operate at regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are automatically triggered by 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 day.

A cron job invokes a URL, using an HTTP `GET` request, at a given time of day. A cron job runs under the 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.yaml`).
2. Add one or more `<cron>` entries to your file and define the necessary elements for each entry, including 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.

Google Cloud Platform photo-timeline-python3 ▾

App Engine Cron jobs REFRESH

Dashboard
Services
Versions
Instances
Task queues
Cron jobs
Security scans
Firewall rules
Quotas
Memcache
Search
Settings

App Engine Cron jobs

You can configure your app to run a scheduled task at a particular time on a recurring basis. Your app's listed here when you configure a job.

[Read about cron jobs](#)

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 morning"  
    url: /mail/weekly  
    schedule: every monday 09:00  
  
  - description: "daily summary"  
    url: /tasks/summary  
    schedule: every 24 hours
```

CRON.YAML

SCHEDULING TASKS WITH CRON

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



[SEND FEEDBACK](#)

Scheduling Jobs with cron.yaml

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](https://cloud.google.com/appengine/docs/standard/python3/scheduling-jobs-with-cron-yaml)

SCHEDULING TASKS WITH CRON

cron:

SHOWS IN THE CONSOLE

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

HANDLER

SCHEDULE

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

```
  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_handler`
`schedule: every minute`
- `description: "monthly mail"`
`url: /mail/weekly`
`schedule: every month`
- `description: "daily tasks"`
`url: /tasks/summary`
`schedule: every 24 hours`

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

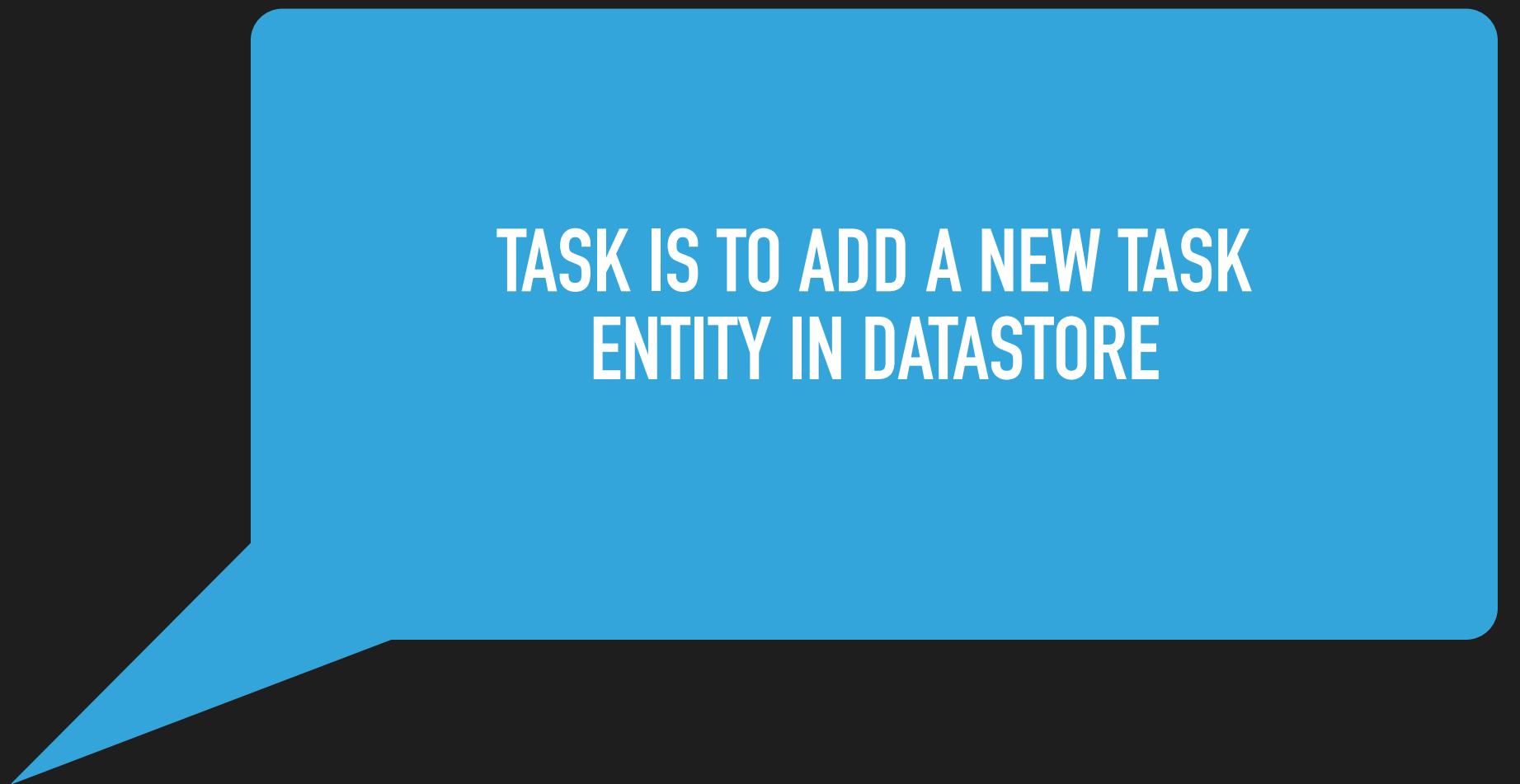
```
@app.route('/cron_task_handler')
def cron_task_handler():
    """Log out the request"""
    logging.info('Running cron task')

    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

The screenshot shows the Google Cloud Platform interface for the App Engine service. The top navigation bar includes the 'Google Cloud Platform' logo, the project name 'photo-timeline-python3', a search bar, and various navigation icons. On the left, a sidebar lists several options: Dashboard, Services, Versions, Instances, Task queues, Cron jobs (which is highlighted in blue), Security scans, Firewall rules, and Quotas. A large yellow arrow points from the 'Cron jobs' link in the sidebar to the 'CRON JOBS' section in the main content area. The main content area is titled 'Cron jobs' and features a 'REFRESH' button. It contains a box with the heading 'App Engine Cron jobs' and a descriptive text: '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.' Below this is a blue 'Read about cron jobs' button.

≡ Google Cloud Platform ⚙ photo-timeline-python3 ▾

Dashboard Services Versions Instances Task queues **Cron jobs** Security scans Firewall rules Quotas

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 ▾

REFRESH

App Engine

- Dashboard
- Services
- Versions
- Instances
- Task queues
- Cron jobs
- Security scans
- Firewall rules
- Quotas

Demo

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

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
```

BREAK TIME



QUERY CURSORS

QUERY CURSORS

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

```
@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 parameter
    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

```
@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

```
@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=10))

    # 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'))
```

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

LOCALHOST:8080?CURSOR=XXXX

SET CURSOR TO NONE FOR THE FIRST BATCH

QUERY CURSORS

```
@app.route('/')
def homepage():
    # Create a Cloud Datastore client.
    datastore_client = datastore.Client()
    query = datastore_client.query(kind='Photo')
    #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'))
```

CREATE QUERY

HOW MANY

GET NEXT PAGE AND CURSOR

QUERY CURSORS

```
@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

```
@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

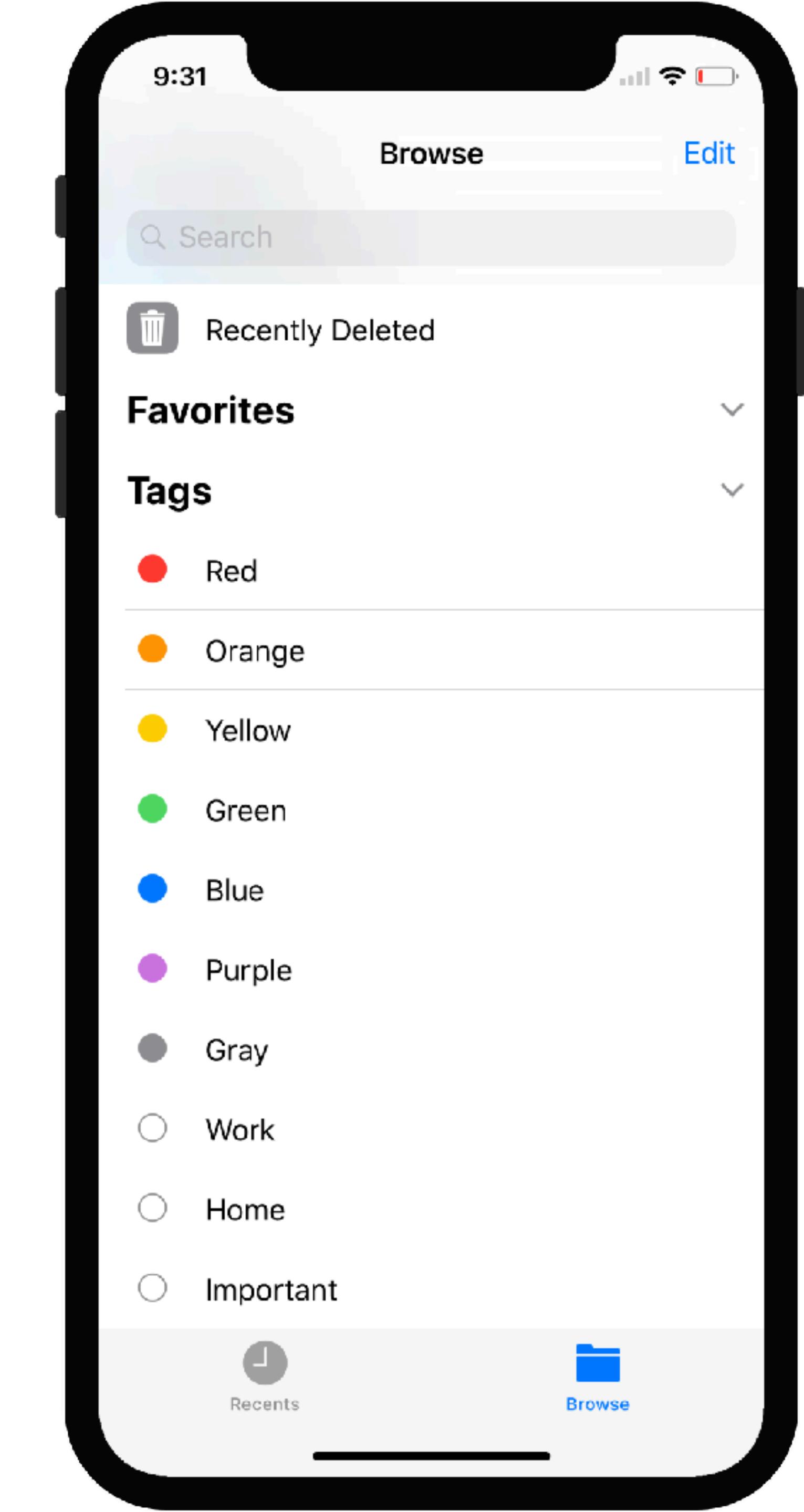
STRATEGIES FOR LOADING A TABLE

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

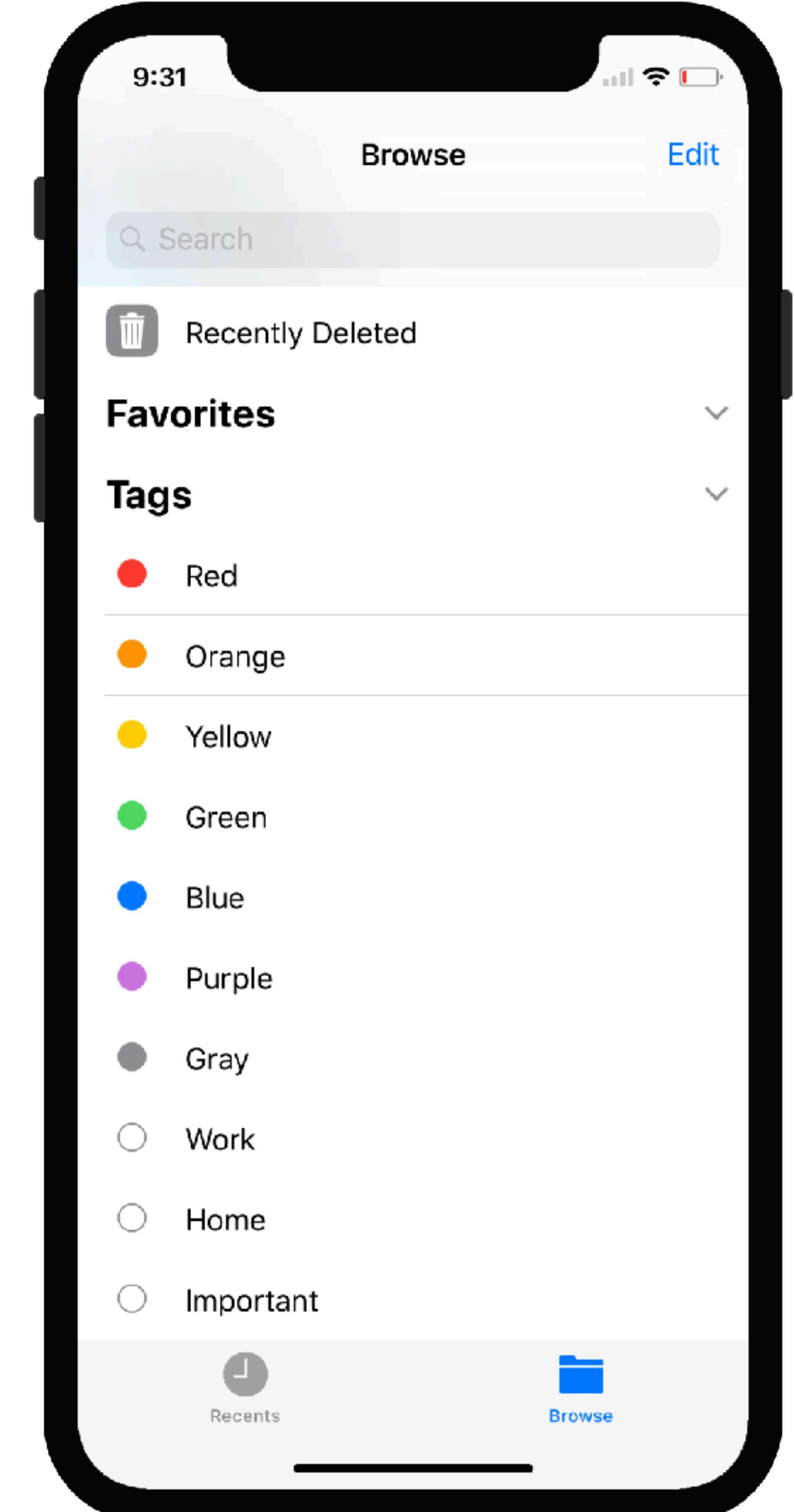
MOBILE CURSOR STRATEGY #1

- 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 indicator
        return 1
    }
}

internal func tableView(tableView: UITableView, cellForRowAt 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 reload the table
        self.loadItems()
    }
}
```

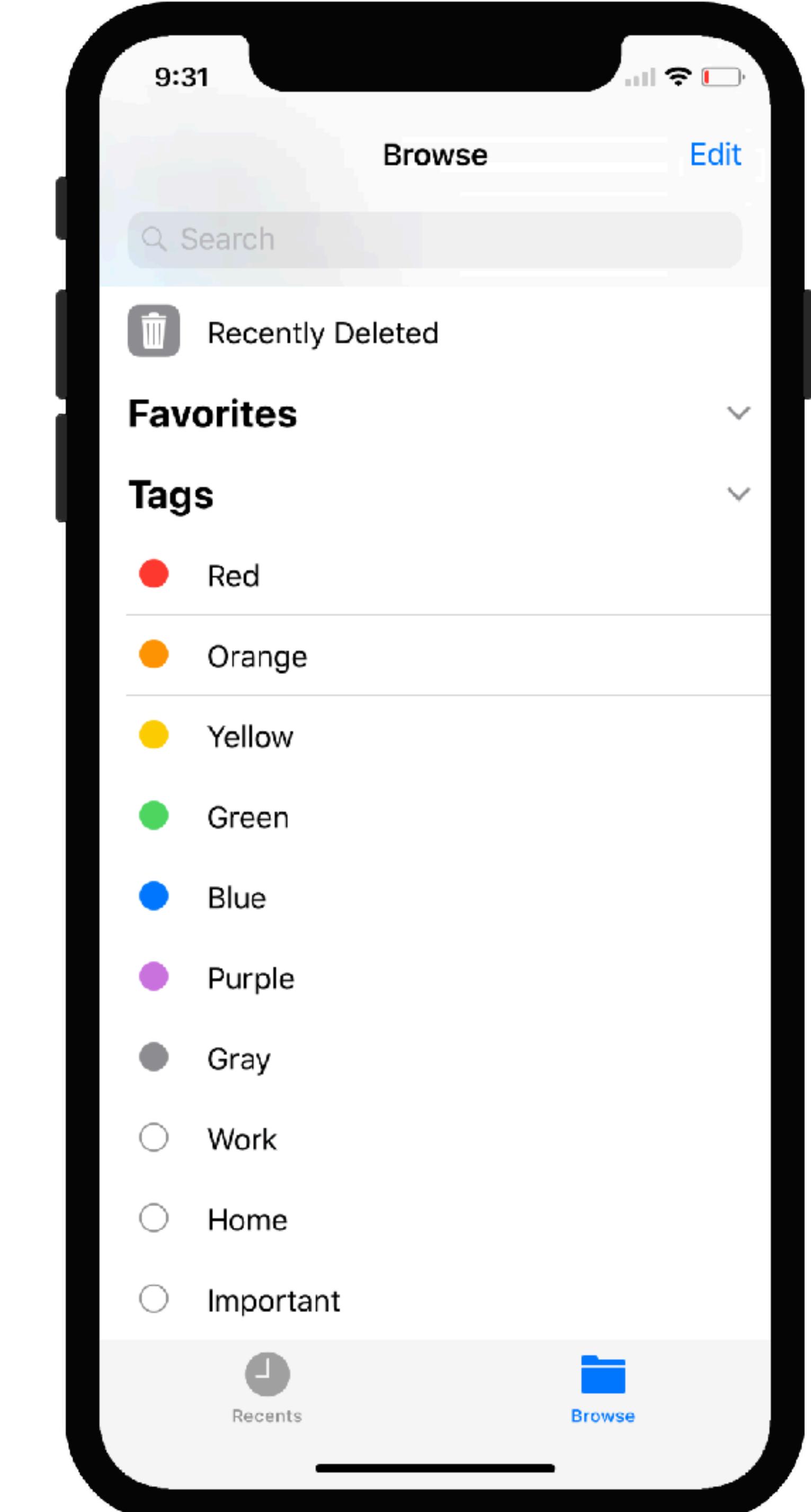


QUERY CURSORS

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 performed the original query. To use the cursor in a subsequent retrieval operation, including the same entity kind, ancestor filter, property filters, and ordering, you must use the same query from which the cursor was generated.
- Because the `!=` and `IN` operators are implemented with materialized cursors, they are not supported in cursored queries.
- Cursors don't always work as expected with a query that uses multiple values. The de-duplication logic for such multiple-value queries might possibly cause the same result to be returned more than once.
- New App Engine releases might change internal implementation details. If an application attempts to use a cursor that is no longer valid, it will receive a `CursorNotFoundException`.

Cursors and data updates

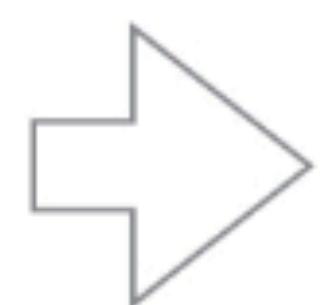
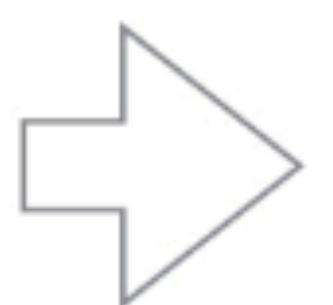
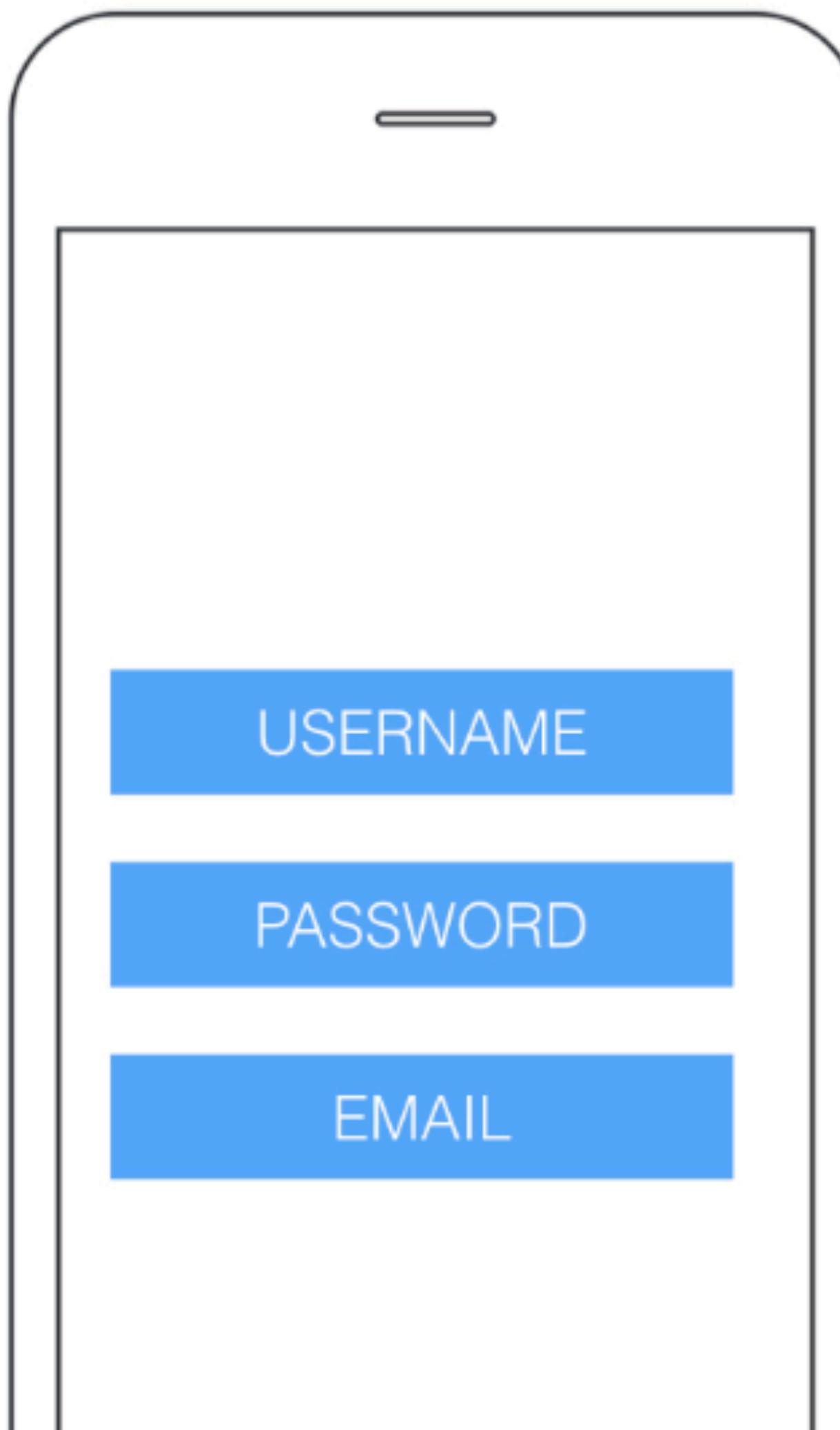
The cursor's position is defined as the location in the result list after the last result that was returned. It's a relative position in the list (it's not an offset); it's a marker to which Cloud Datastore returns results. If the results for a query change between uses of a cursor, the results returned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

When retrieving query results, you can use both a start cursor and an end cursor to retrieve results from Cloud Datastore. When using a start and end cursor to retrieve results, the results will be the same as when you generated the cursors. Entity deletions between the time the cursors are generated and when they are used will not affect the results.

ASSIGNMENT 3

ASSIGNMENT 3

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



ASSIGNMENT 2

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

Why GitHub? Enterprise Explore Marketplace Pricing

Search Sign in Sign up

GoogleCloudPlatform / python-docs-samples

Watch 243 Star 3,449 Fork 3,570

Code Issues 34 Pull requests 28 Projects 0 Wiki Security Insights

Branch: master ▾ python-docs-samples / appengine / flexible / mailgun / Create new file Find file History

gguuss and crwilcox App engine upversion (#2437) ... Latest commit 2b2cef9 7 days ago

..

File	Description	Time
templates	Rename MVM To Flexible	3 years ago
README.md	Added "Open in Cloud Shell" buttons to README files (#1254)	2 years ago
app.yaml	Update all samples to use env: flex instead of vm: true	3 years ago
example-attachment.txt	Rename MVM To Flexible	3 years ago
main.py	update region tags (#1631)	last year
main_test.py	Rename MVM To Flexible	3 years ago
requirements.txt	App engine upversion (#2437)	7 days ago



MPCS 51033 • AUTUMN 2019 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS