THE UNIVERSITY OF CHICAGO

MPCS 51033 • SPRING 2017 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS

CLASS NEWS

MOBILE APP BACKEND SERVICES WITH GCP

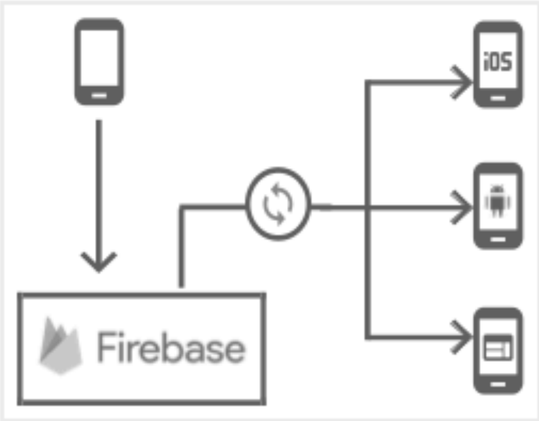
BACKEND SERVICES WITH GCP

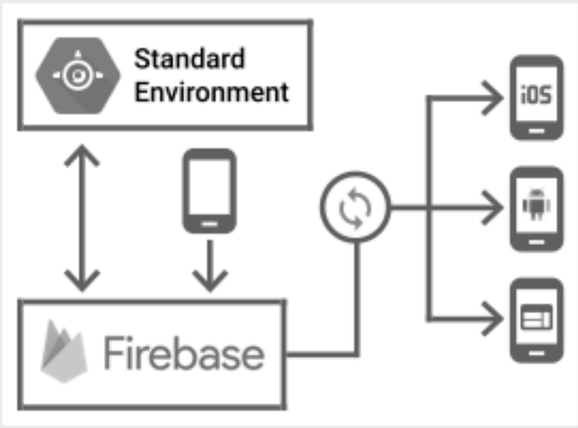
- Most mobile apps and games need a backend service for things that can’t be done solely on-device, such as sharing and processing data from multiple users, or storing large files.

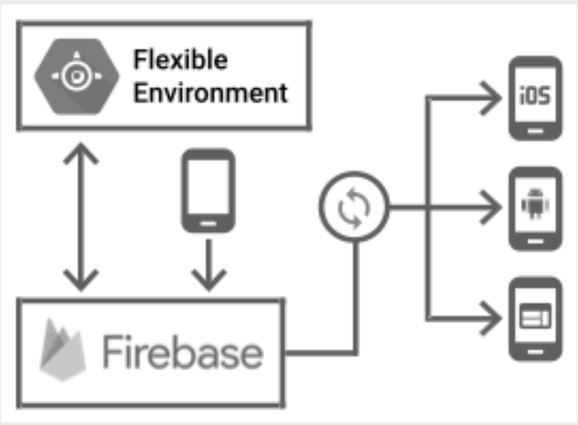
BACKEND SERVICES WITH GCP

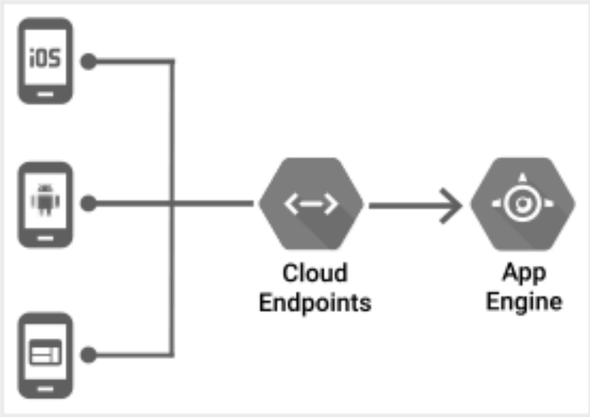
- Building a backend service for a mobile app is similar to building a web-based service, with some additional requirements:
  - Limit on-device data storage
  - Synchronize data across multiple devices
  - Handle the offline case gracefully
  - Send notifications and messages
  - Minimize battery drain

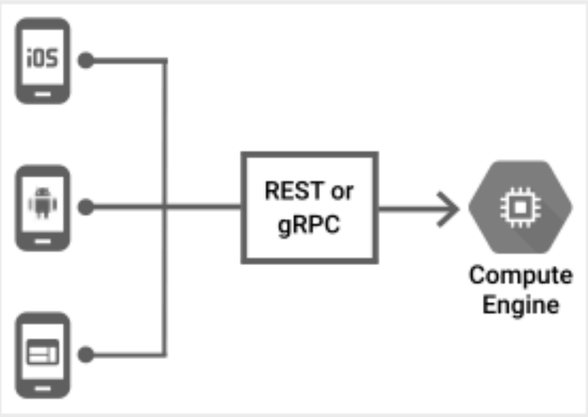
BACKEND SERVICES WITH GCP







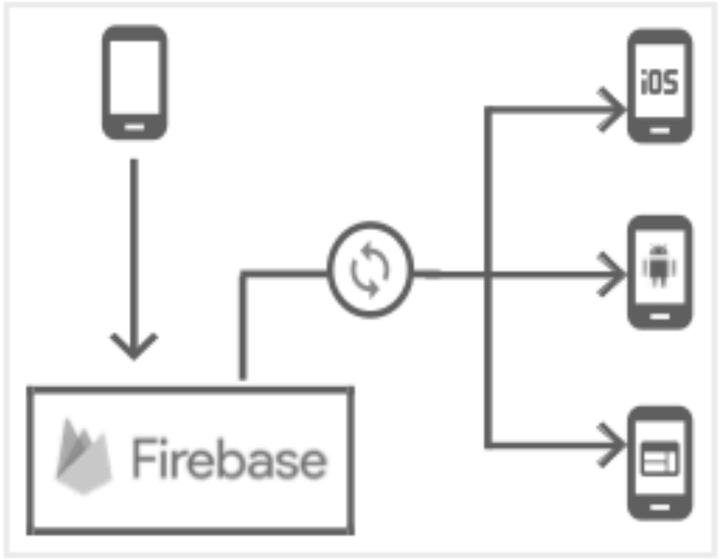


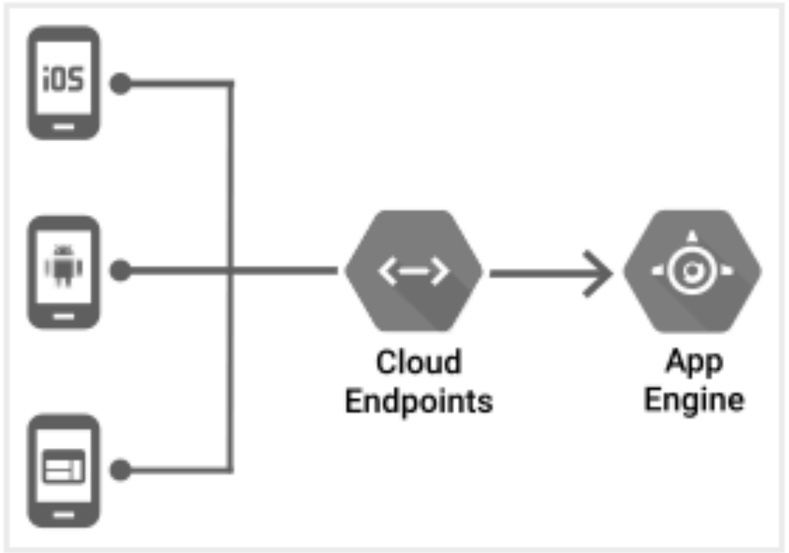


DESIGN PATTERNS USING GCP FOR BACKENDS SERVICES

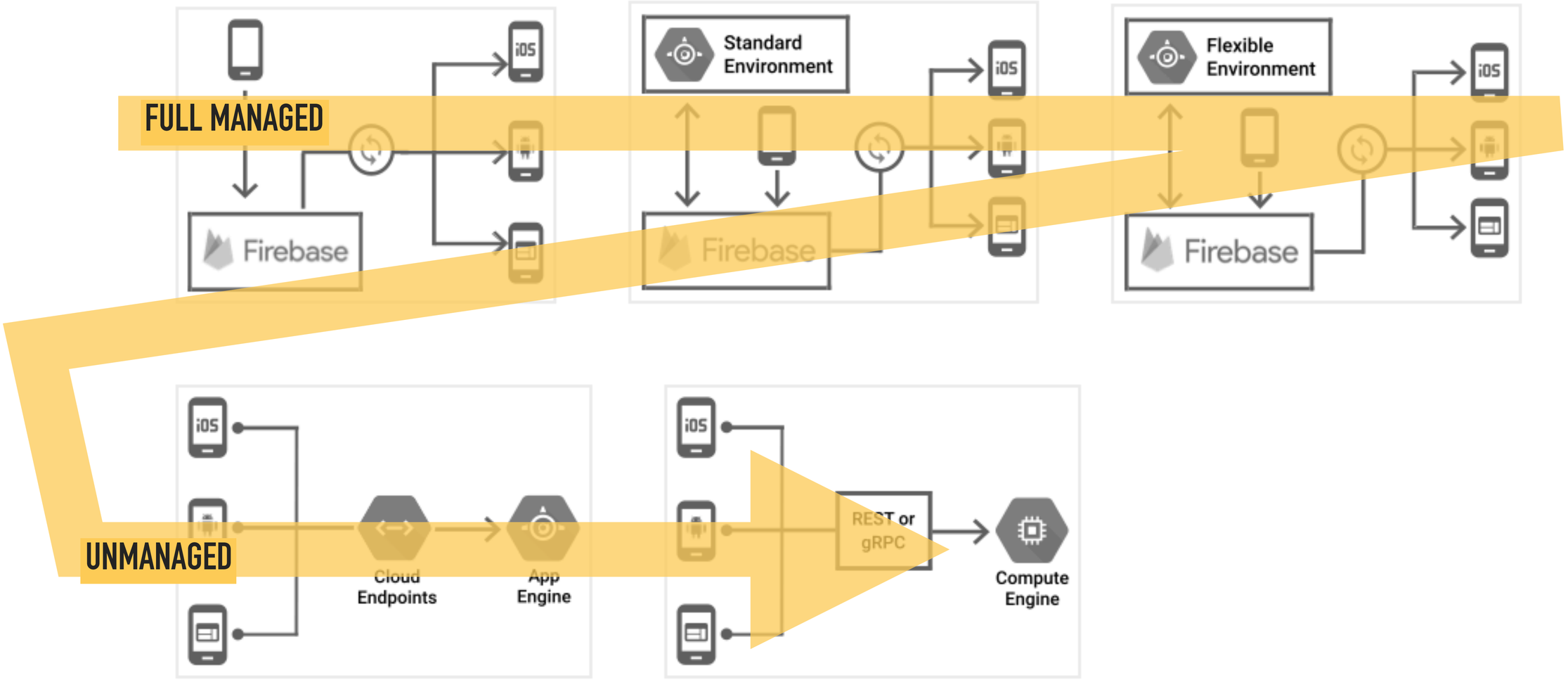
BACKEND SERVICES WITH GCP

- Two-tier architecture with Firebase
  - Mobile apps and Firebase both manipulate the data directly
  - Security and data validation handled in console
- Three-tier architecture with App Engine
  - Communication layer between the mobile app and the backend service
  - Responsible for authentication and data-validation code



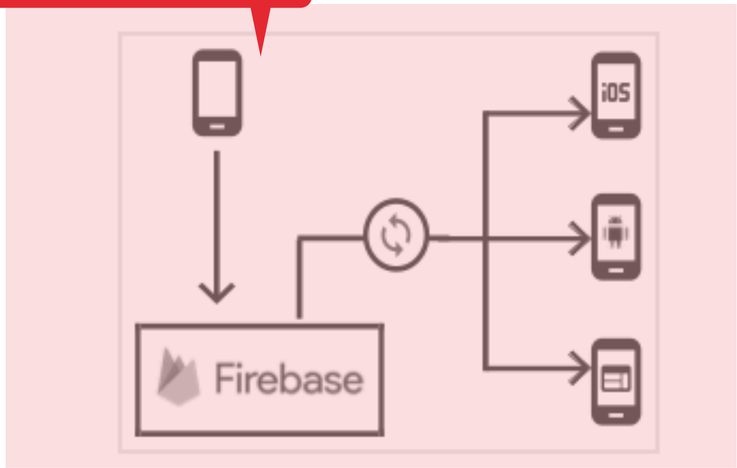


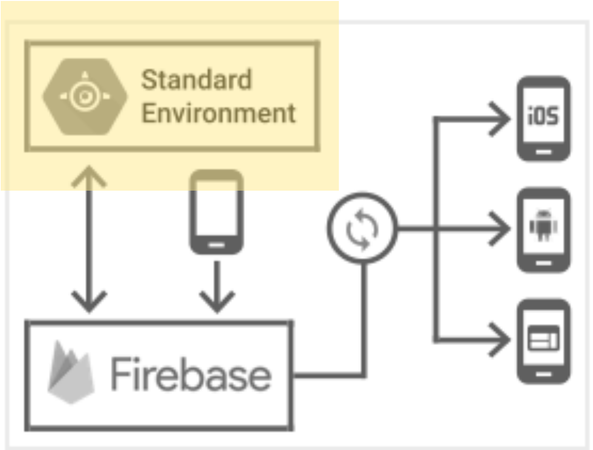
BACKEND SERVICES WITH GCP

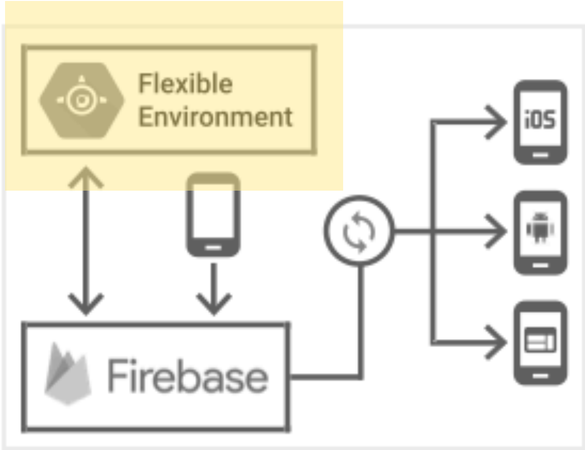


BACKEND SERVICES WITH GCP

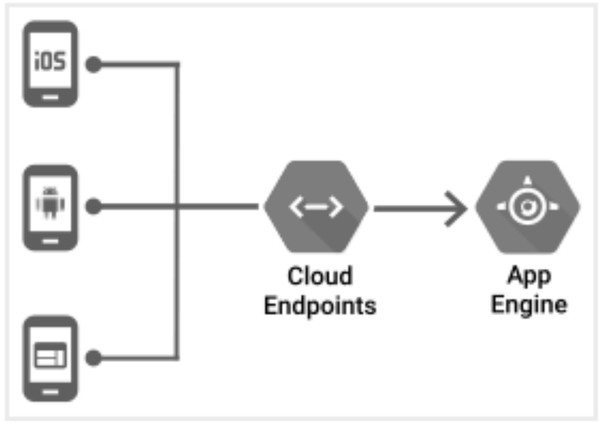
LATER IN THE COURSE

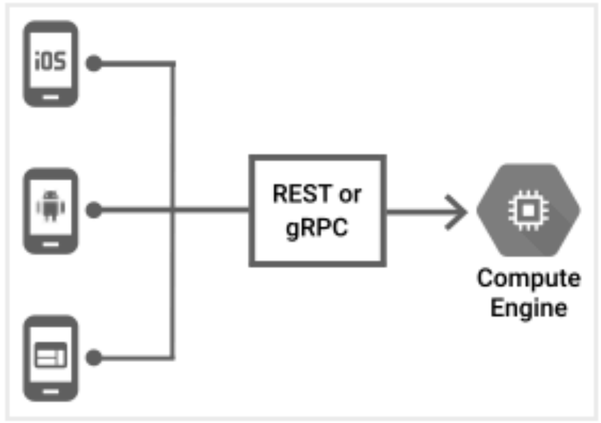






HYBRID PROVIDES MIX OF EASE OF USE AND FLEXIBILITY

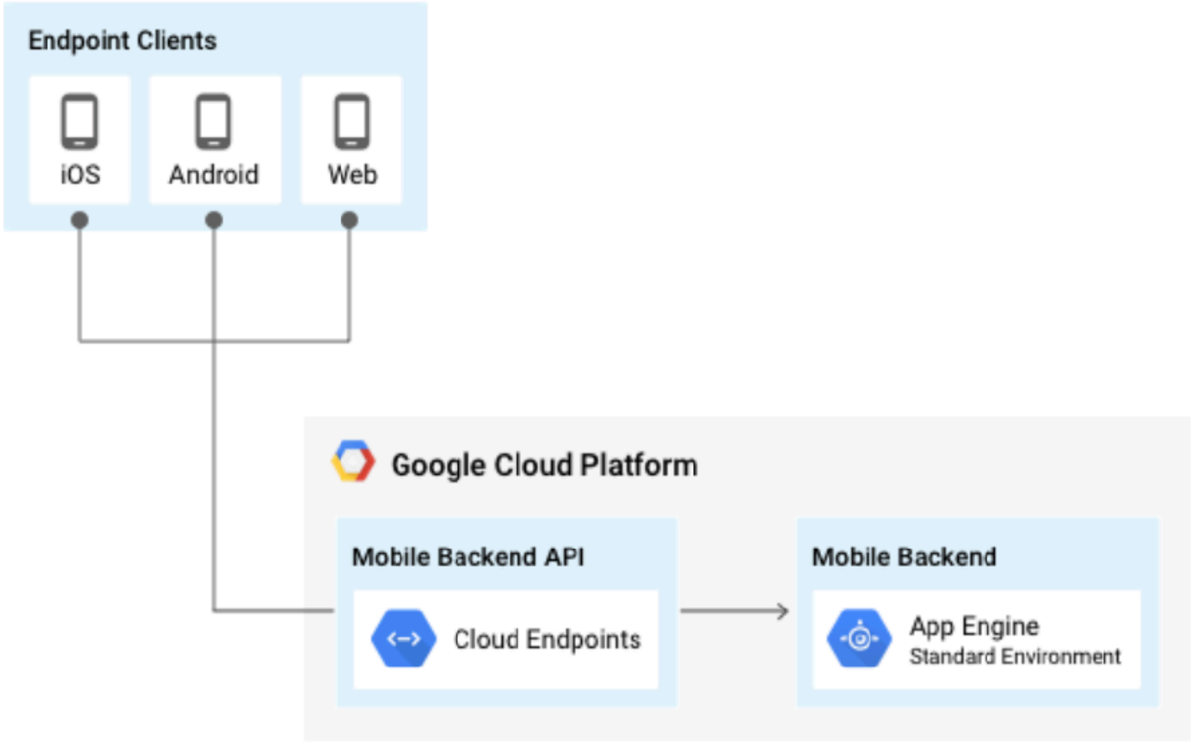




APP ENGINE

CLOUD ENDPOINTS

- Google Cloud Endpoints generates APIs, client libraries, and discovery documentation for an App Engine application
- You don’t write wrappers to handle communication with App Engine
- Cient libraries are generated by Cloud Endpoints to make direct API calls from your mobile app



APP ENGINE

CLOUD ENDPOINTS

- Google Cloud Endpoints generates APIs, client libraries, and discovery documentation for an App Engine application
- You don’t write wrappers to handle communication with App Engine
- Cient libraries are generated by Cloud Endpoints to make direct API calls from your mobile app

Cloud Endpoints

About Cloud Endpoints Frameworks

Contents

Google Cloud Endpoints Frameworks for App Engine consists of tools, libraries and capabilities that allow you to generate APIs and client libraries from an App Engine application. Referred to as an *API*, they simplify client access to data from other applications, such as web clients and Android mobile clients.

★ **Note:** Cloud Endpoints Frameworks supports only the App Engine standard environment; the App Engine flexible environment is not supported.

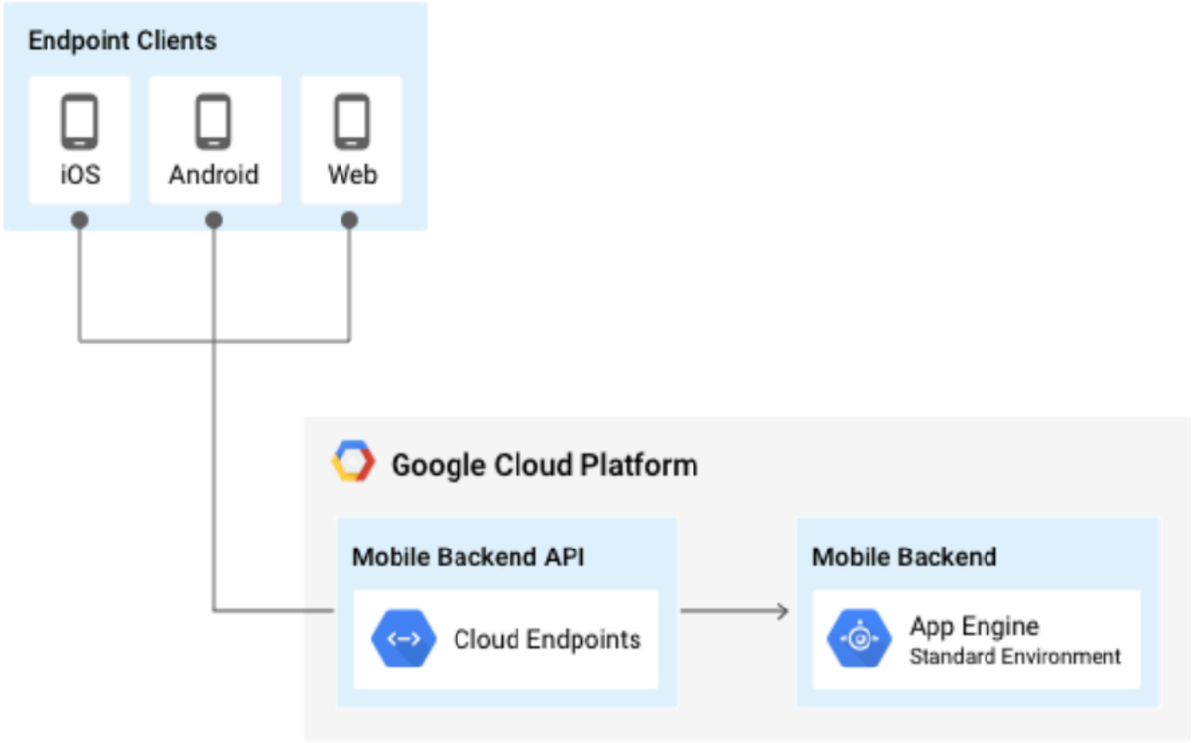
Cloud Endpoints Frameworks supports the following API management features provided by Cloud Endpoints:

- API key management
- API sharing
- user authentication
- View API metrics
- View API logs

APP ENGINE

CLOUD ENDPOINTS

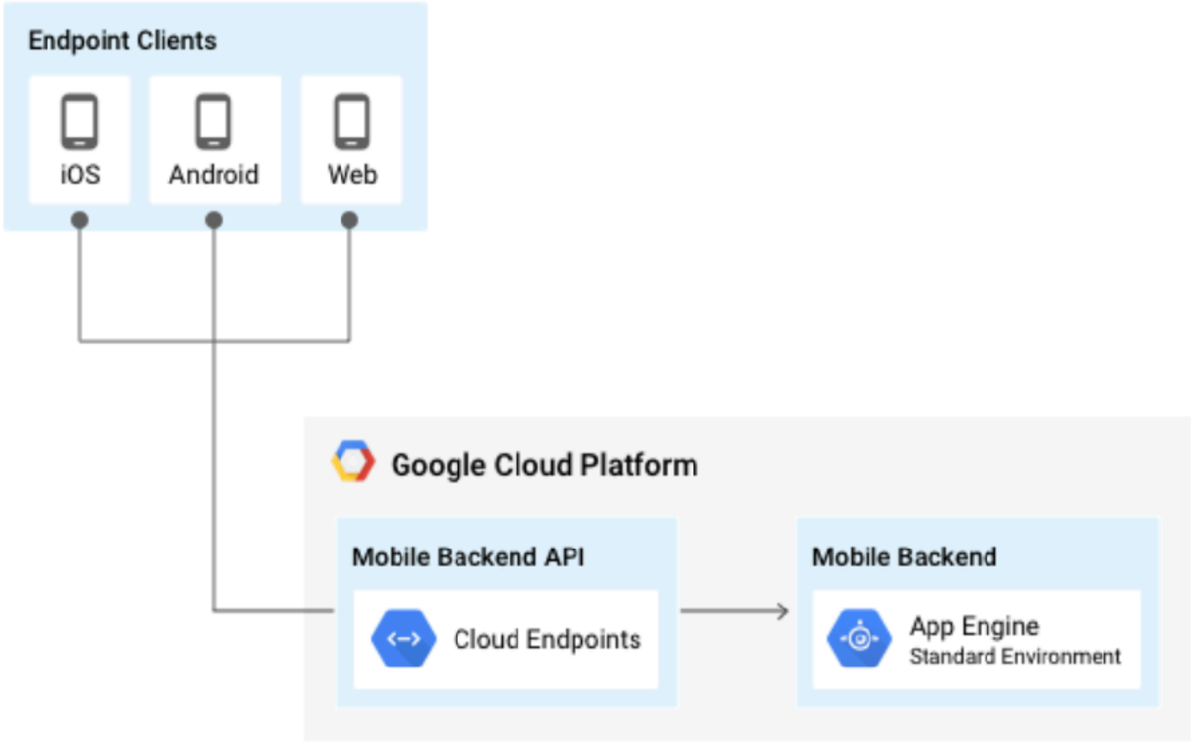
- Recommended use
  - Automated generation of client libraries that apps can use to call the backend service directly (ie ease of use)
  - Reducing on-device storage by moving files to Cloud Storage
  - Sending notifications by calling Cloud Messaging



APP ENGINE

CLOUD ENDPOINTS

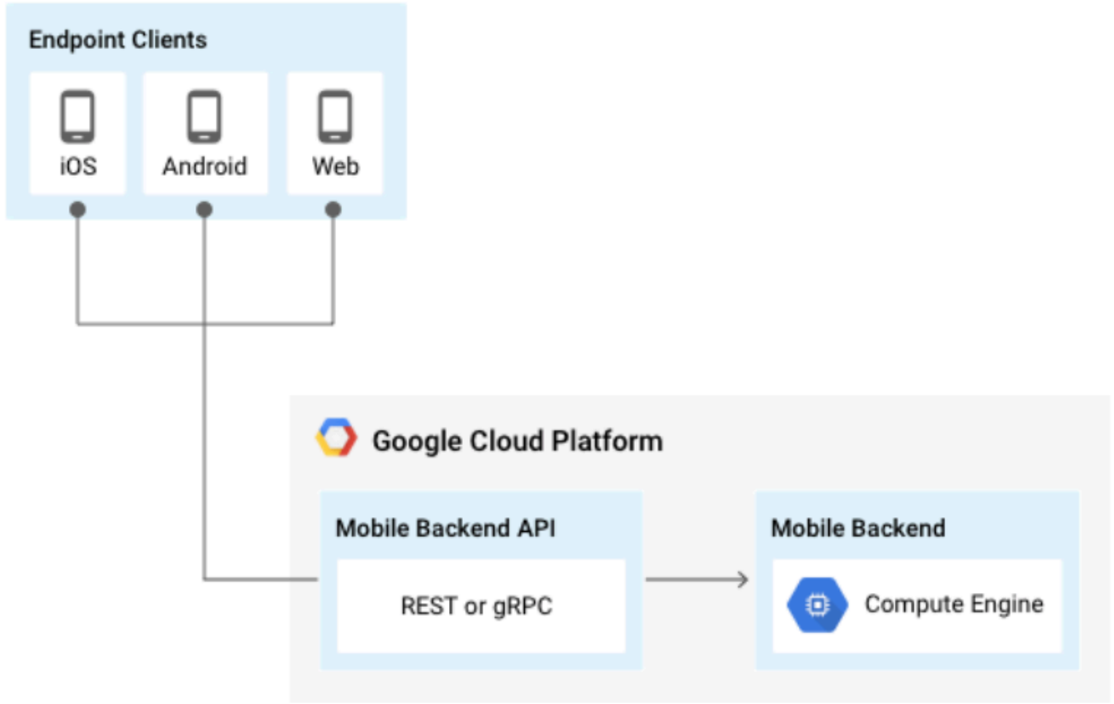
- Not recommended for
  - Apps that require automatic real-time data synchronization across devices
  - Backend services that require custom server or third party libraries.
  - Systems that do not support SSL; SSL is required by Cloud Endpoints.



APP ENGINE

COMPUTE ENGINE

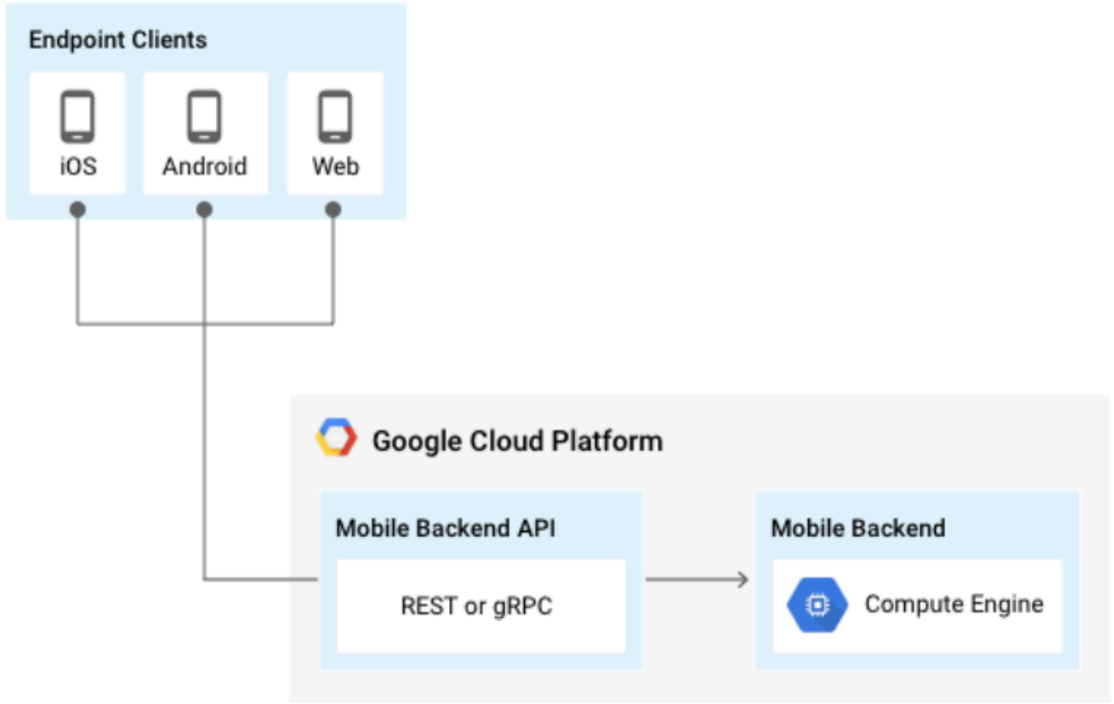
- Computer Engine lets you create and run virtual machines on Google infrastructure
- You have administrator rights to the server and full control over its configuration
    - You are responsible for updates and maintenance :(



APP ENGINE

COMPUTE ENGINE

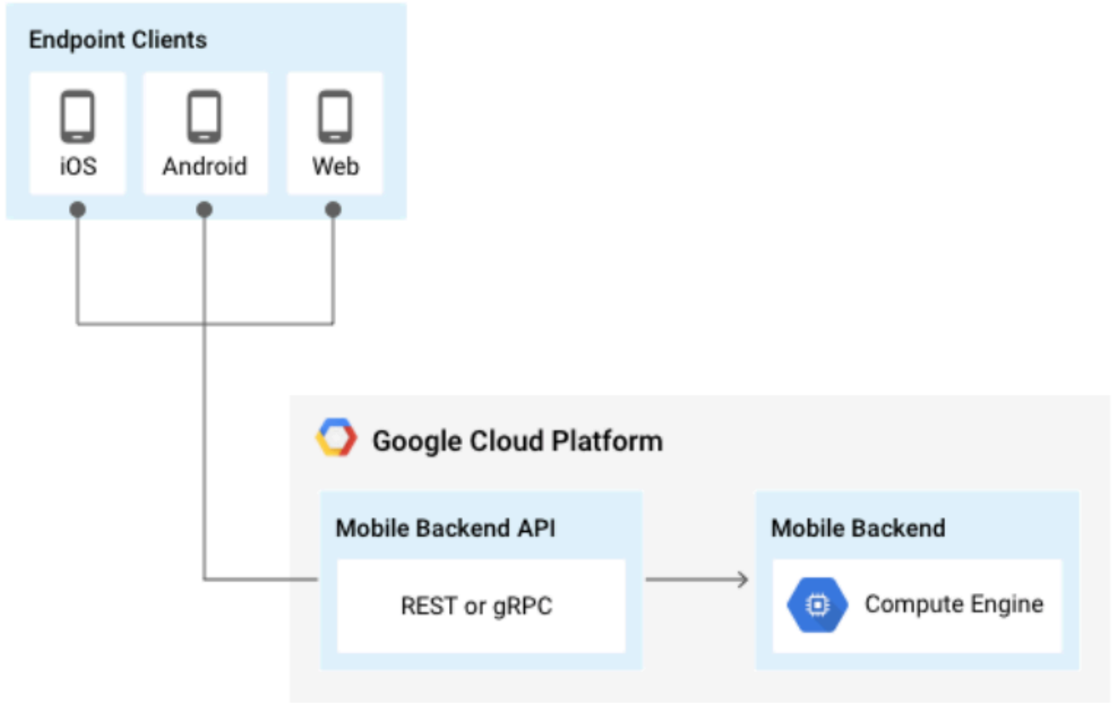
- Recommended for:
  - Porting an existing backend service running on an on-premise server or a virtual machine.
  - Backend services that require a custom server or third-party libraries.



APP ENGINE

COMPUTE ENGINE

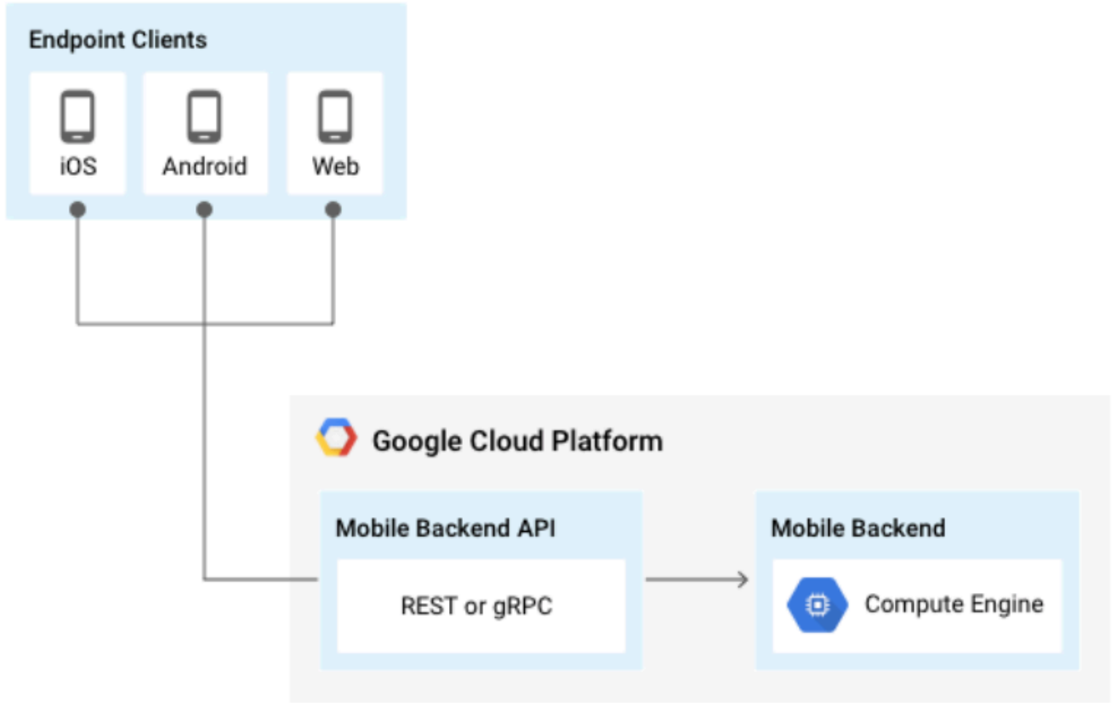
- Not recommended for:
  - Apps that require automatic real-time data synchronization across devices.
  - Automatic maintenance; you must maintain and upgrade the server yourself.
  - Automatic scaling; you must manually configure and manage an autoscaler.



APP ENGINE

COMPUTE ENGINE

- Protocols used to connect to compute engine
  - REST - architecture based on HTTP
  - gRPC - framework using http/2



CLOUD ENDPOINTS

CLOUD ENDPOINTS

- Interest??

SENDING EMAIL WITH APP ENGINE

SENDING EMAIL

- Two APIs for sending an email message in App Engine environment
  - mail.send_mail()
  - EmailMessage class

SENDING EMAIL

- Who can send email?
  - The Gmail or Google Apps Account of the user who is currently signed in
  - Any email address of the form anything@[APP_NAME].appspotmail.com or anything@[APP_ALIAS].appspotmail.com
  - Any email address listed in the Cloud Platform Console under Email API Authorized Senders

SENDING EMAIL

- Mail will bound back to the senders email
  - App Engine doesn't know
- There are special rules if you are sending out bulk emails

SENDING EMAIL

```
from google.appengine.api import mail


mail.send_mail(sender="me@uchicago-mobi-photo-
                       timeline.appspotmail.com",
                      to="abinkowski@uchicago.edu",
                      subject="New Photo!",
                      body="Hi!")
```

SENDING EMAIL

```
mail.send_mail(sender=sender_address,
                    to="Albert Johnson
<Albert.Johnson@example.com>",
                    subject="Your account has been approved",
                    body="""Dear Albert:

Your example.com account has been approved.  You can now visit
http://www.example.com/ and sign in using your Google Account to
access new features.

Please let us know if you have any questions.

The example.com Team
""")
```

SENDING EMAIL

```
message = mail.EmailMessage(
          sender=sender_address,
          subject="Your account has been approved")
message.to = "Albert Johnson <Albert.Johnson@example.com>"
message.body = "Hi"
message.send()
```

SENDING EMAIL

Limit	Amount
Maximum size of outgoing mail messages, including attachments	31.5 MB
Maximum size of incoming mail messages, including attachments	31.5 MB
Maximum size of message when an administrator is a recipient	16 KB
Maximum number of authorized senders	50

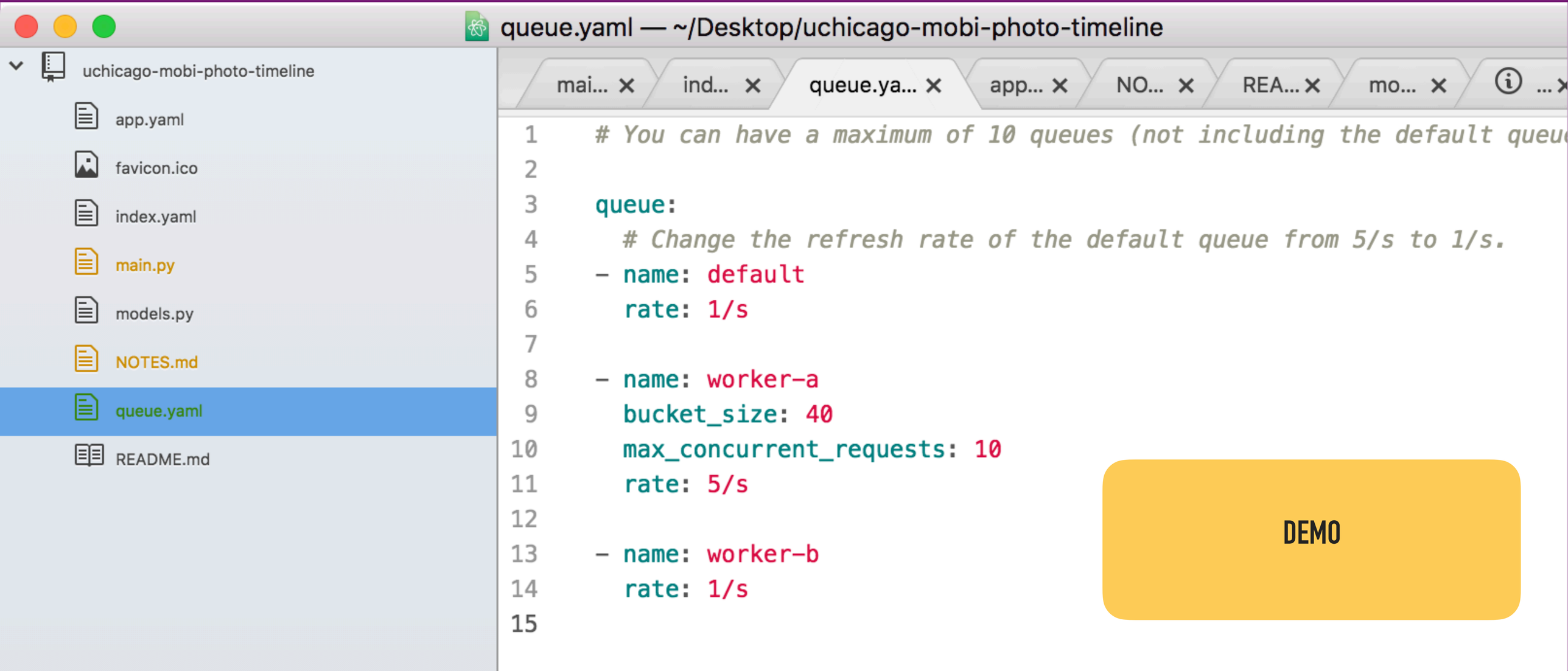
SENDING EMAIL

```
try:
  mail.SendMessage(to='test@example.com',
                   from='admin@example.com',
                   subject='Test Email',
                   body='Testing')
except apiproxy_errors.OverQuotaError, message:
  # Log the error.
  logging.error(message)
  # Display an informative message to the user.
  self.response.out.write('The email could not be sent. '
                          'Please try again later.')
```

SENDING EMAIL

```
try:
  mail.SendMessage(to='test@example.com',
                   from='admin@example.com',
                   subject='Test Email',
                   body='Testing')
except apiproxy_errors.OverQuotaError, message:
  # Log the error.
  logging.error(message)
  # Display an informative message to the user.
  self.response.out.write('The email could not be sent. '
                          'Please try again later.')
```

SENDING EMAIL



TASK QUEUE

TASK QUEUE

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

Task Queue Overview

Contents

Python | Java | PH

This page describes what task queues are, and when and how to use them. The Task API lets applications perform work, called *tasks*, asynchronously outside of a user re an app needs to execute work in the background, it adds tasks to *task queues*. The ta 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 queu

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, y control the workers' scaling behavior and hence your costs.

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

TASK QUEUE

- 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

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.

TASK QUEUE

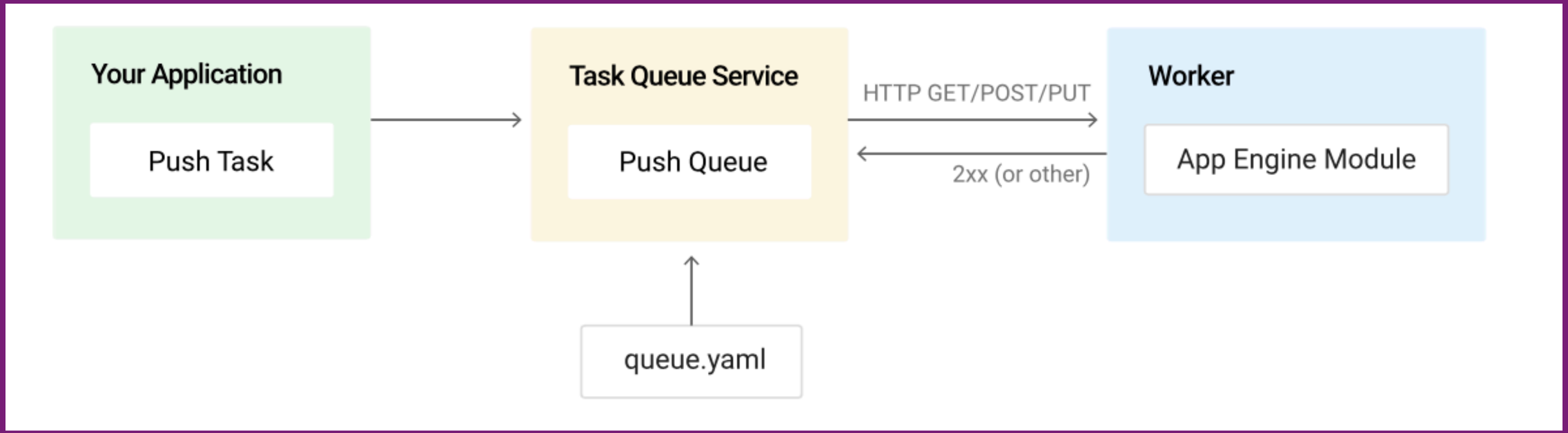
USE CASES

- Push queue
  - Long running operations
  - Scheduled tasks

- Pull queue
  - Tasks that are interdependent
  - Related tasks that can be batched for efficiency

PUSH QUEUE

PUSH QUEUES



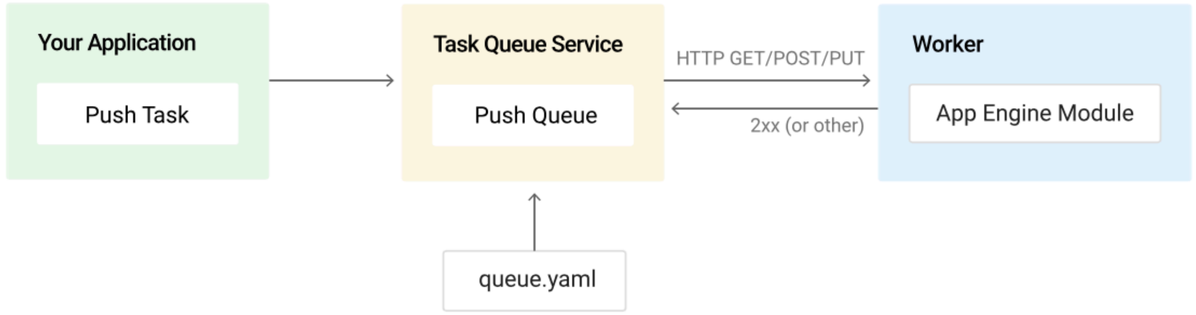
- Push queues run tasks by delivering HTTP requests to App Engine worker services

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 you can use different services for different task types
- Automatic scaling services must finish before 10 minutes have elapsed
- Manual and basic scaling services can run up to 24 hours

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

```
queue:
  # Change the refresh rate of the default queue from 5/s to 1/s.
- name: default
  rate: 1/s

- name: worker-a
  bucket_size: 40
  max_concurrent_requests: 10
  rate: 5/s

- name: worker-b
  rate: 1/s
```

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 tas

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.")


##################################################################################
#
##################################################################################
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

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

```
        task = taskqueue.add(
            url='/email_task/',
            params={'message': 'hi'}
        )
        logging.debug('Task {} enqueued, 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

```
        task = taskqueue.add(
            url='/email_task/',
            params={'message': 'hi'}
        )
        logging.debug('Task {} enqueued, 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')
```

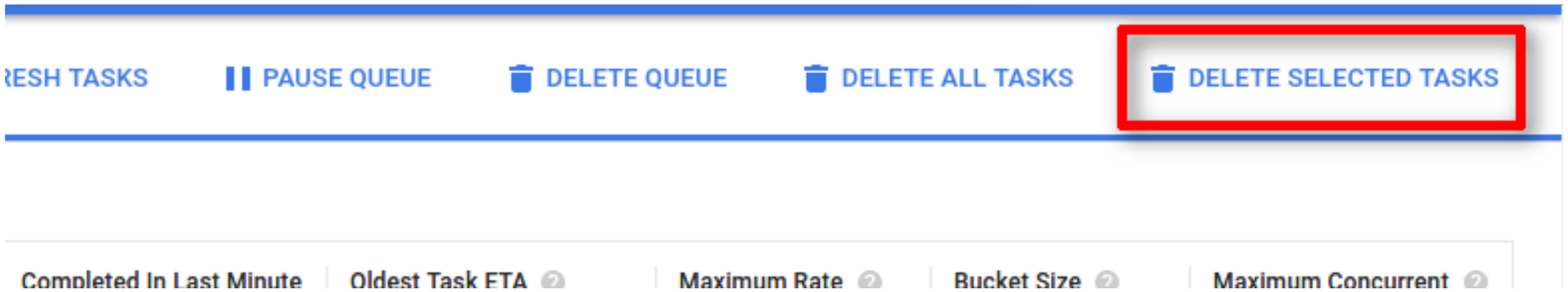
DEMO

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

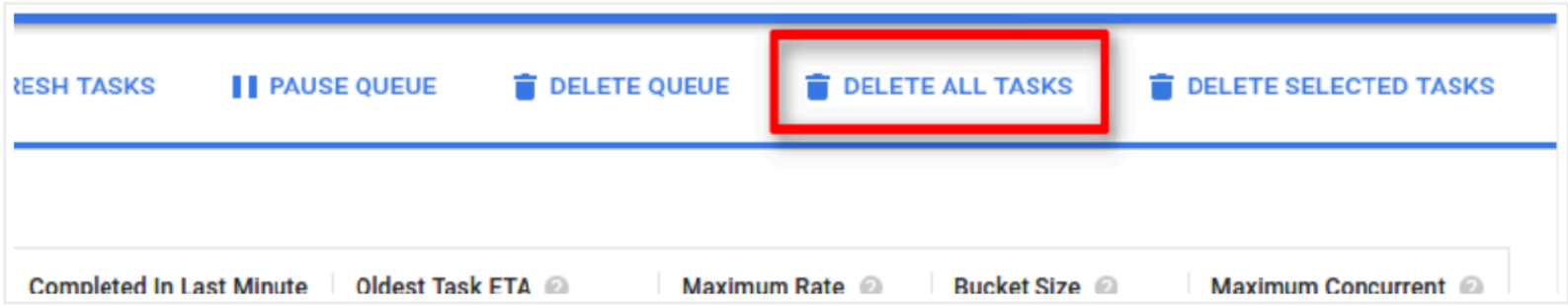


PUSH QUEUES

DELETE TASK

```
from google.appengine.api import taskqueue

# Purge entire queue...
q = taskqueue.Queue('queue1')
q.purge()
```

- Delete all tasks from a queue



PUSH QUEUES

DEFFERRED TASK

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

Background work with the deferred library

Contents

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

PUSH QUEUES

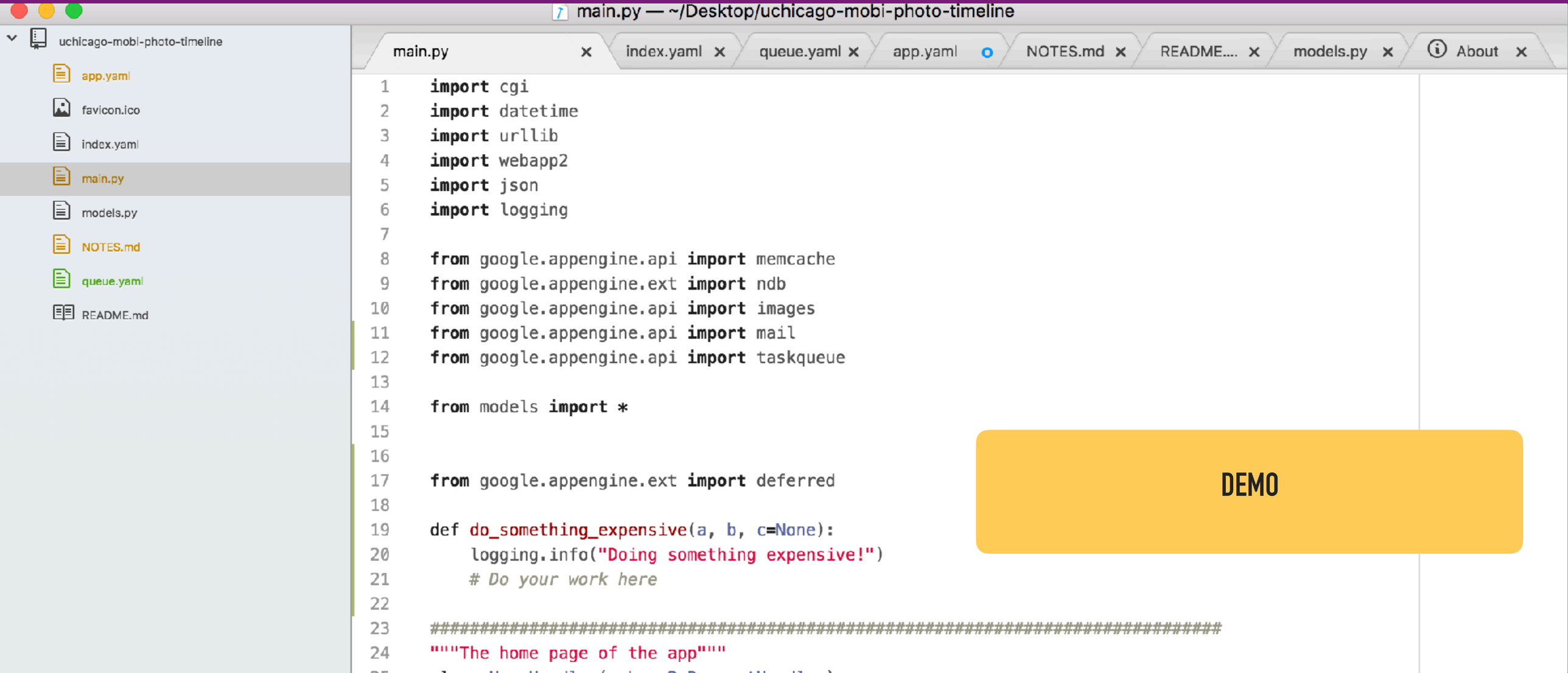
DEFERRED TASKS

```
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



PULL QUEUE

PULL QUEUE

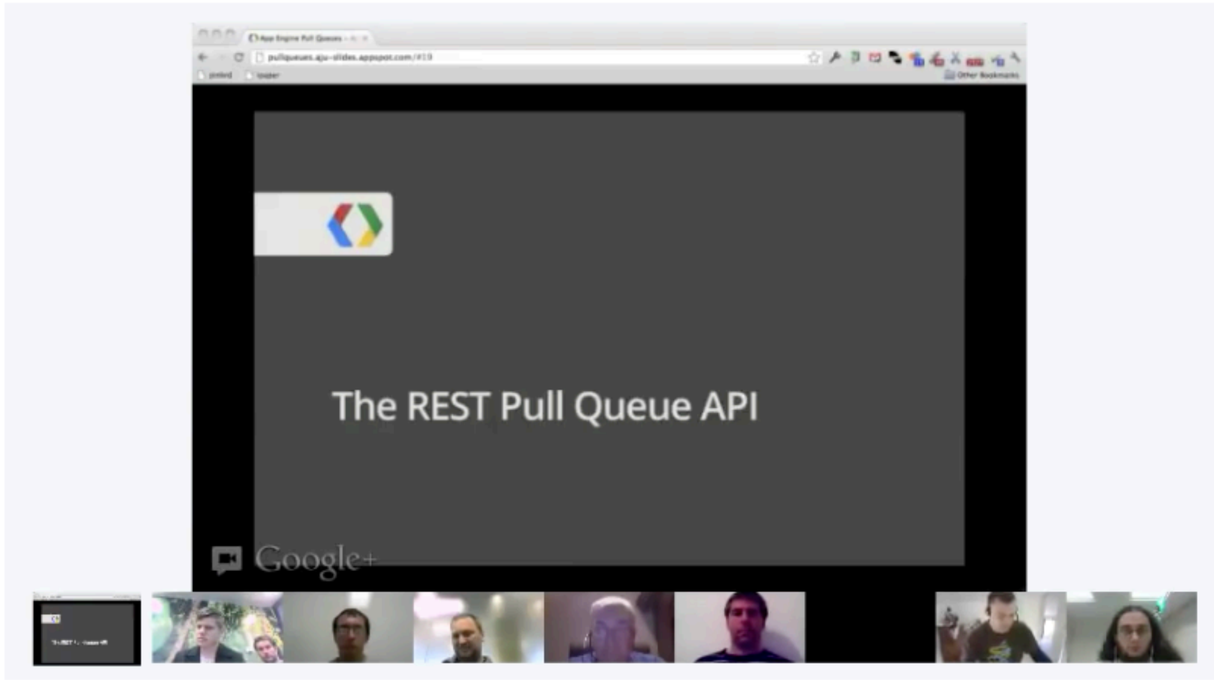
- Pull queues allow a task consumer to process tasks outside of App Engine's default task processing system
  - More cutstomizeable
- If the task consumer is a part of your App Engine app, you can manipulate tasks using simple API calls from the google.appengine.api.taskqueue module
- Task consumers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



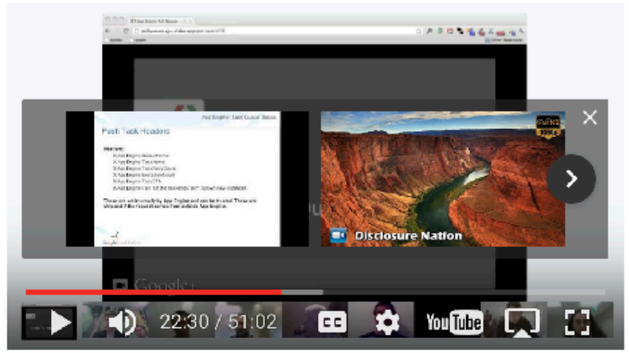
TASK QUEUE

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

SEND FEEDBACK

Using Pull Queues in Python

Contents



Pull 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.

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

•••

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

...

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
- Job request is subject to the same limits as those for push task queues

Scheduling Tasks With Cron for Python

Contents

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

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

```
cron:
- description: daily summary job
  url: /tasks/summary
  target: beta
  schedule: every 24 hours
```

SCHEDULING TASKS WITH CRON

```
cron:
- description: daily summary job
  url: /tasks/summary
  #target: beta
  schedule: every 1 minutes
```

SHOWS IN THE CONSOLE

HANDLER

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

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: "daily summary job"
  url: /tasks/summary
  schedule: every 24 hours
- description: "monday morning mailout"
  url: /mail/weekly
  schedule: every monday 09:00
  timezone: Australia/NSW
- description: "new daily summary job"
  url: /tasks/summary
  schedule: every 24 hours
  target: beta
```

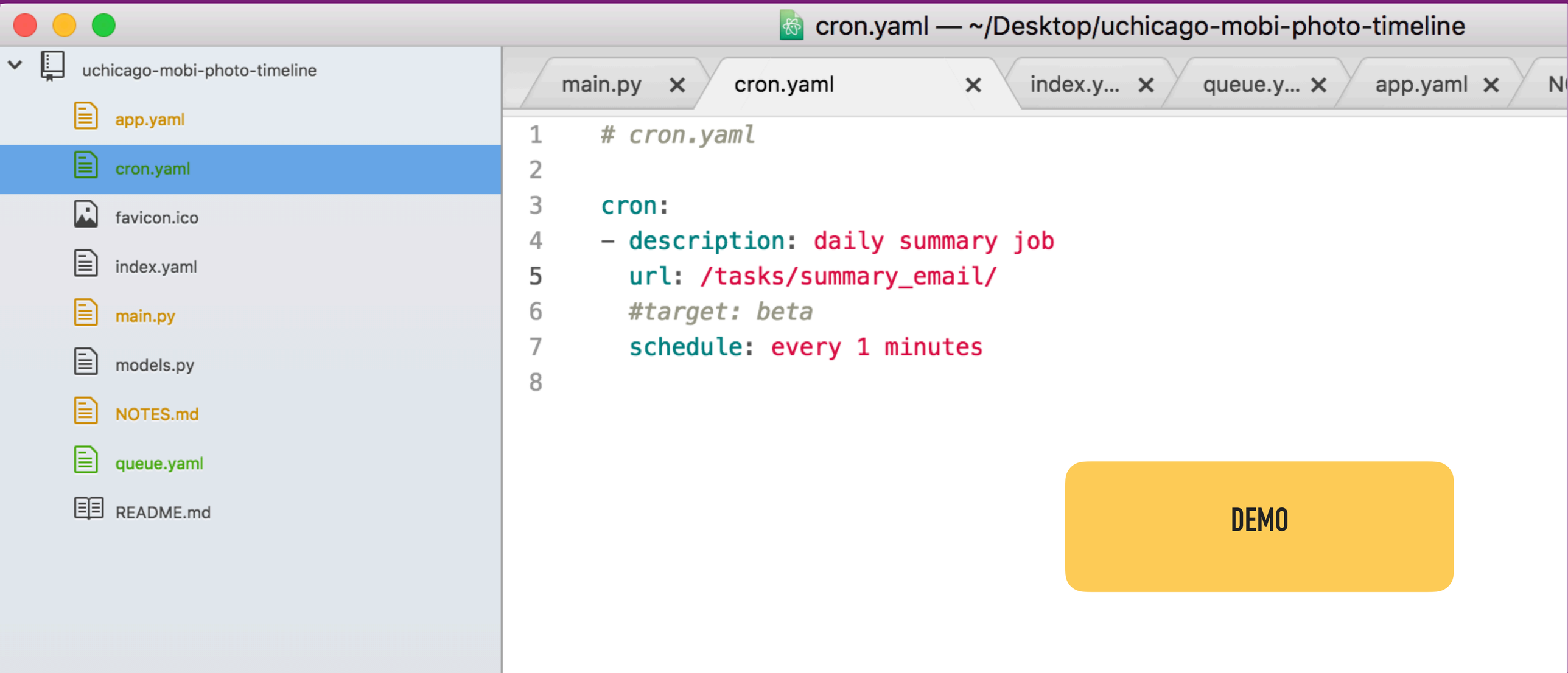
RUNS ON DEFAULT BY DEFAULT. OTHERWISE SPECIFY A TARGET OR VERSION

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: daily summary job
  url: /tasks/summary
  target: beta
  schedule: every 24 hours
```

SCHEDULING TASKS WITH CRON



BREAK TIME 😴

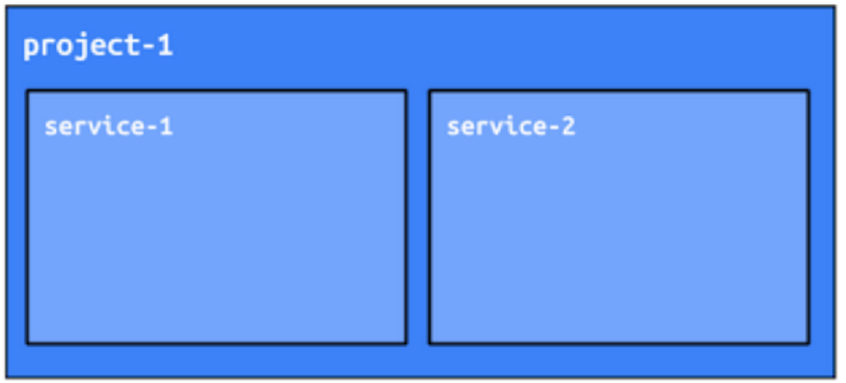
MICROSERVICES

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

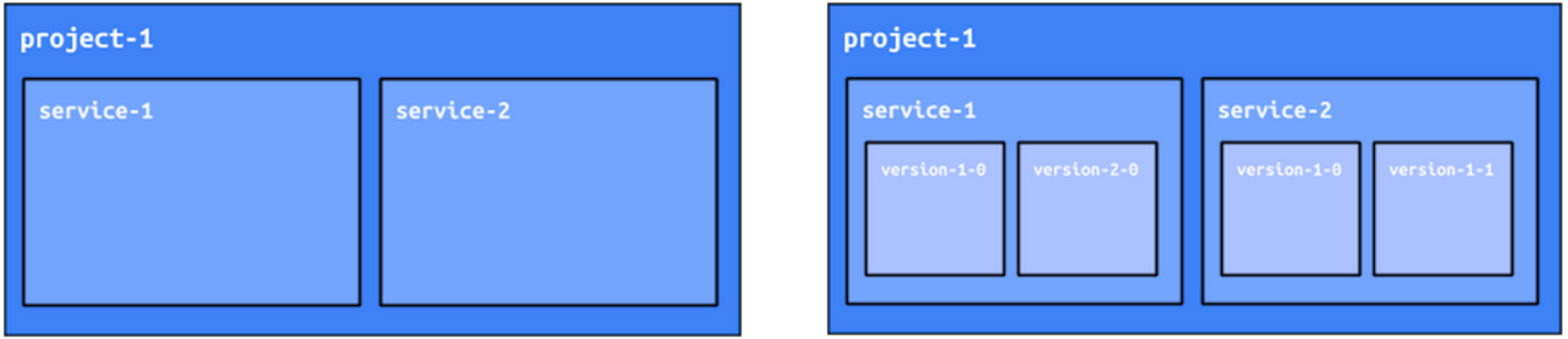
- Manageable now, but what happens when we add new features, tasks, schedule tasks?
- Compartmentalizing the function into logical segments may help
  - And minimize problems (potentially)

```
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.

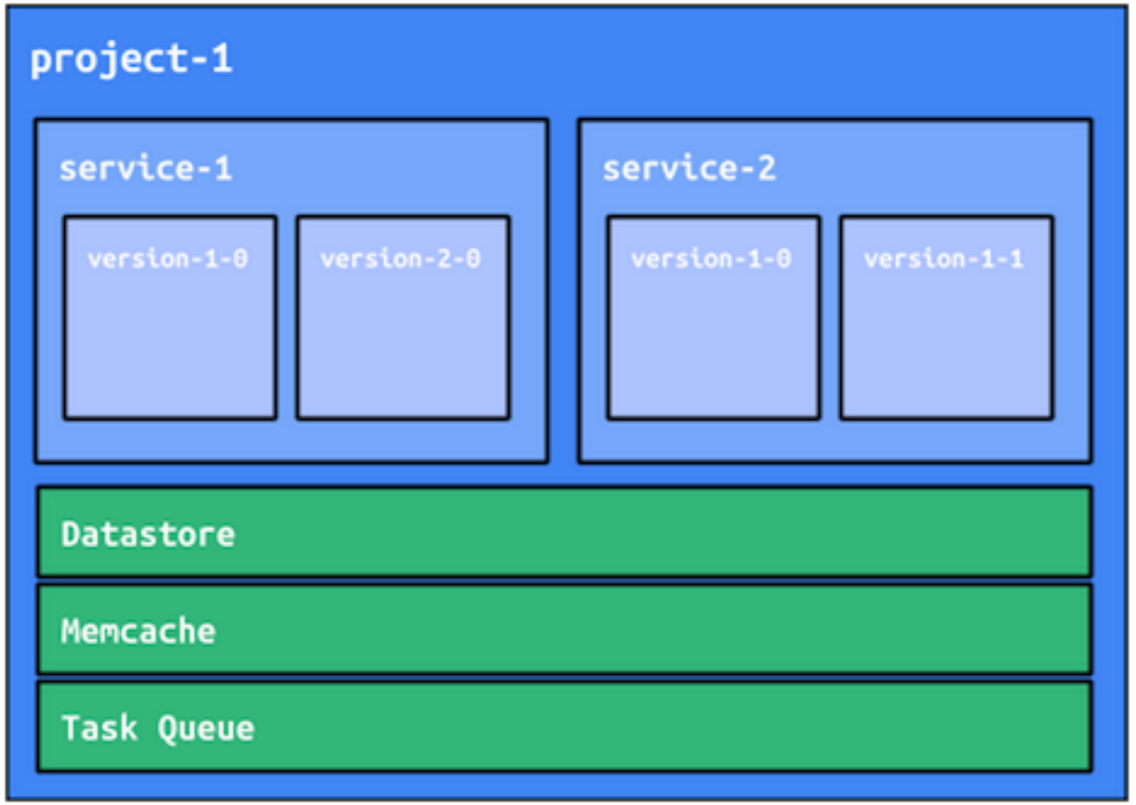
MICROSERVICES



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

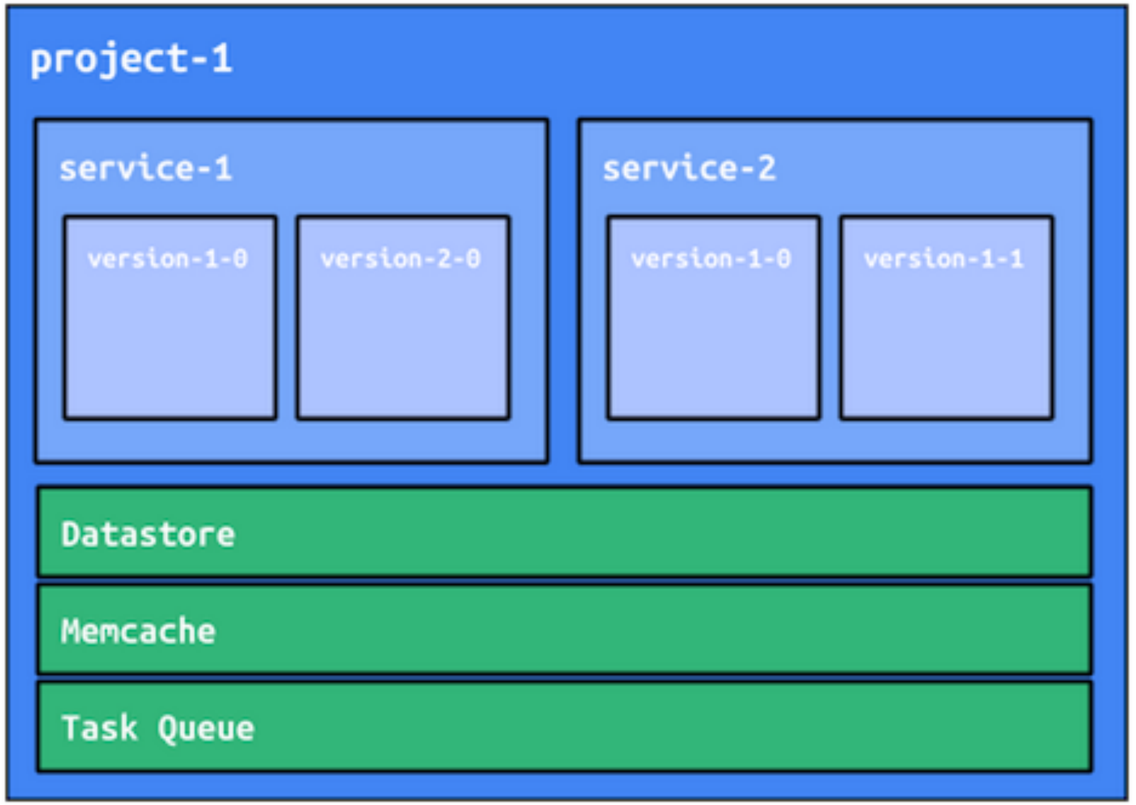
MICROSERVICES

- 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



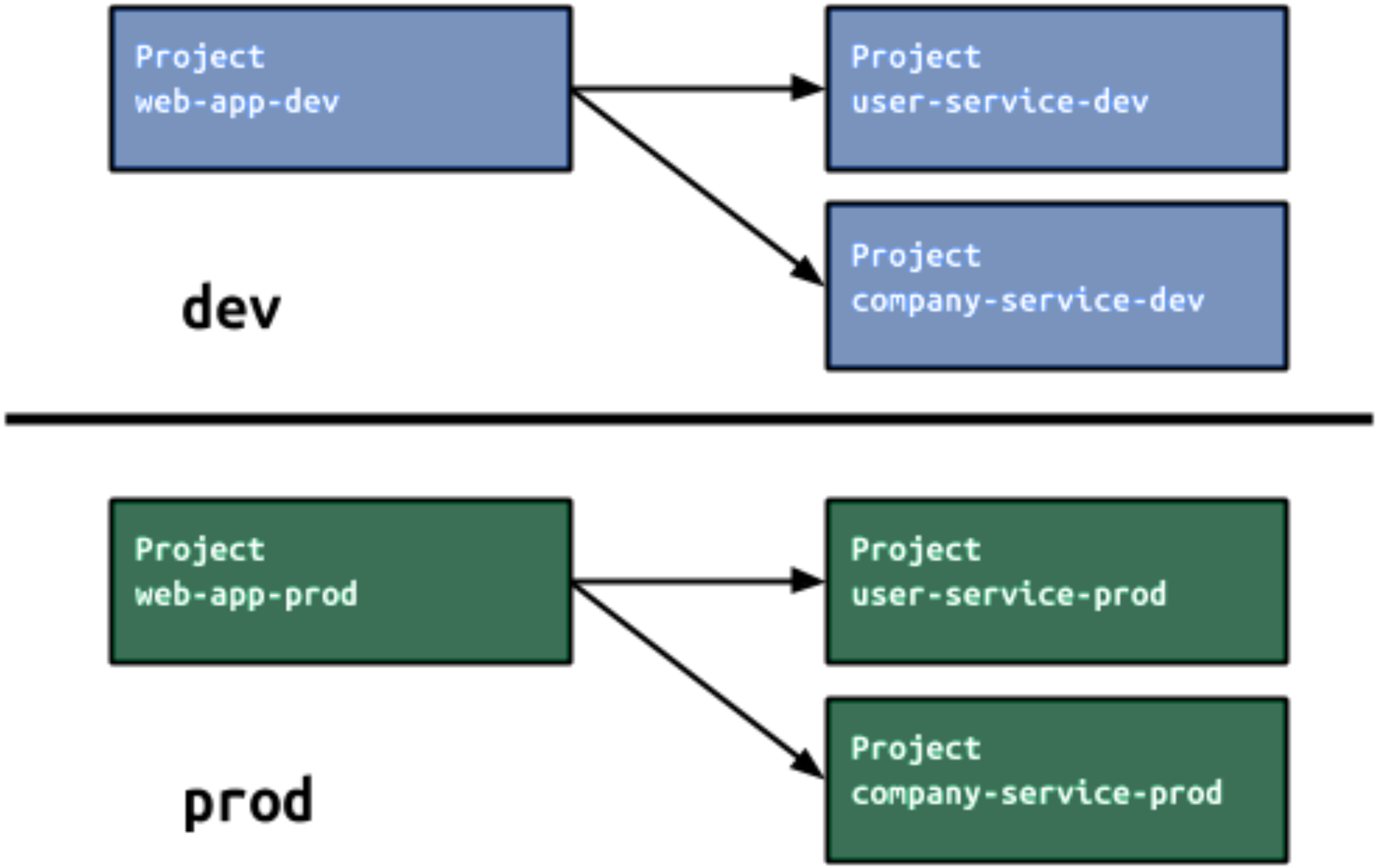
MICROSERVICES

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

	Multiple services	Multiple projects
Code isolation	Deployed code is completely independent between services and versions.	Deployed code is completely independent between projects, and between services and versions of each project.
Data isolation	Cloud Datastore and Memcache are shared between services and versions, however *namespaces* can be used as a developer pattern to isolate the data. For Task Queue isolation, a developer convention of queue names can be employed, such as **`user-service-queue-1`**.	Cloud Datastore, Memcache, and Task Queues are completely independent between projects.
	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

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 all static
  - 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

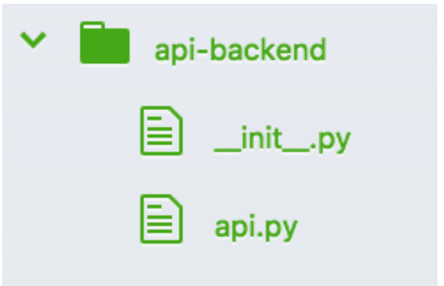
```
  service: default

# 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

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

SHARDING

SHARING

- You need to pay attention to how often an entity is updated
- Datastore can only update a single entity or entity group about five times a second
- App Engine can handle parallel requests much better

Sharding counters

Contents

Source
Conclusion
More Info
Related links

Joe Gregorio
October 2008, updated *August 2014*

When developing an efficient application on Google App Engine, you need to pay attention to ho
entity is updated. While App Engine's datastore scales to support a huge number of entities, it is
note that you can only expect to update any single entity or entity group about five times a seco
estimate and the actual update rate for an entity is dependent on several attributes of the entity,
how many properties it has, how large it is, and how many indexes need updating. While a single
entity group has a limit on how quickly it can be updated, App Engine excels at handling many p
requests distributed across distinct entities, and we can take advantage of this by using shardin

The question is, what if you had an entity that you wanted to update faster than five times a sec
example, you might count the number of votes in a poll, the number of comments, or even the n
visitors to your site. Take this simple example:

PYTHON JAVA GO

```
class Counter(ndb.Model):
  count = ndb.IntegerProperty()
```

If you had a single entity that was the counter and the update rate was too fast, then you would
contention as the serialized writes would stack up and start to timeout. The way to solve this pr
little counter-intuitive if you are coming from a relational database; the solution relies on the fac
from the App Engine datastore are extremely fast and cheap. The way to reduce the contention

SHARING

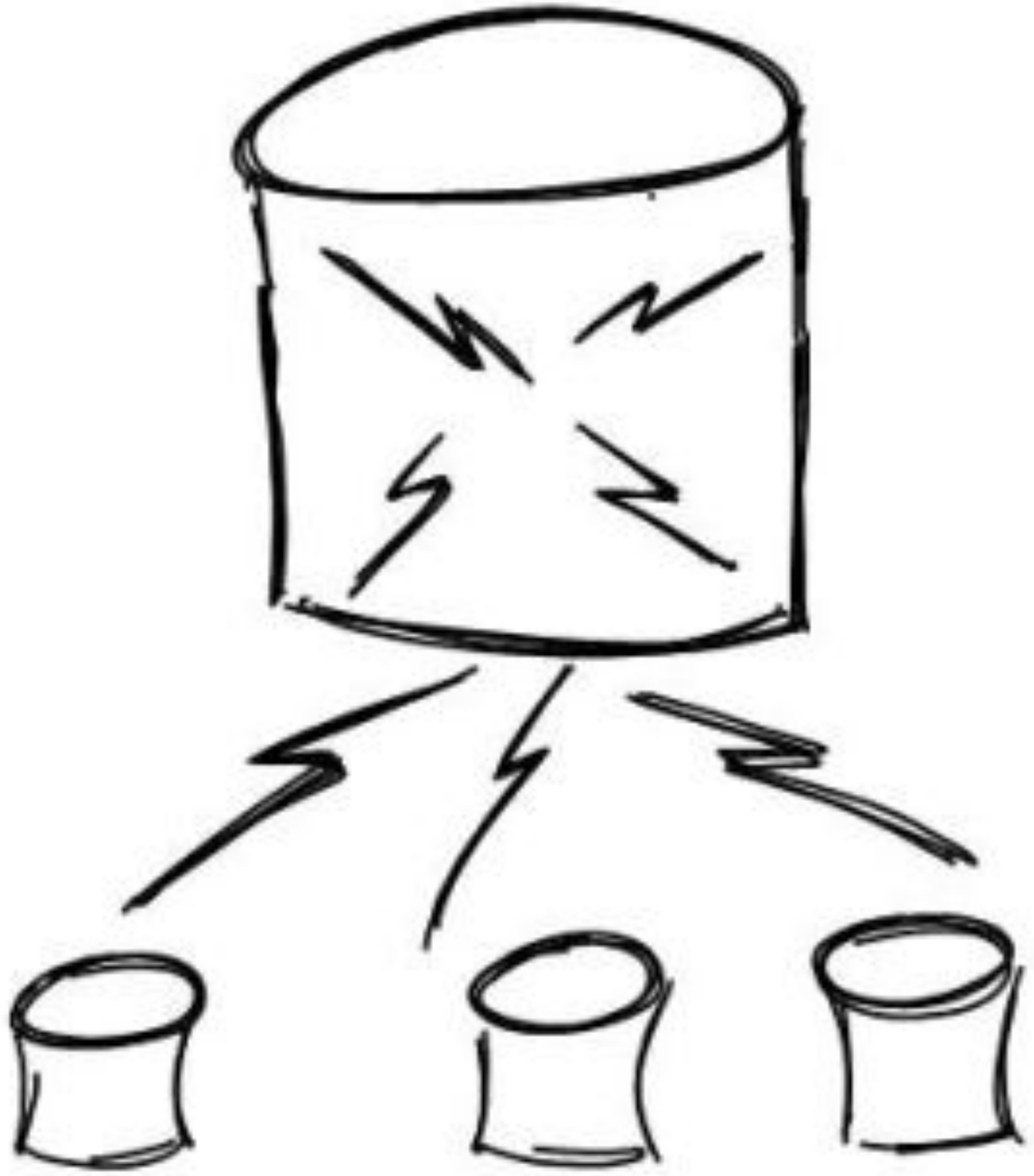
- So what if you want to update more than 5 times a second?
  - Voting/liking
  - Analytics
  - Viral app 😉
  - . . .

SHARING

- So what if you want to update more than 5 times a second?
  - Contention between writes
  - Timeout
  - Loose data

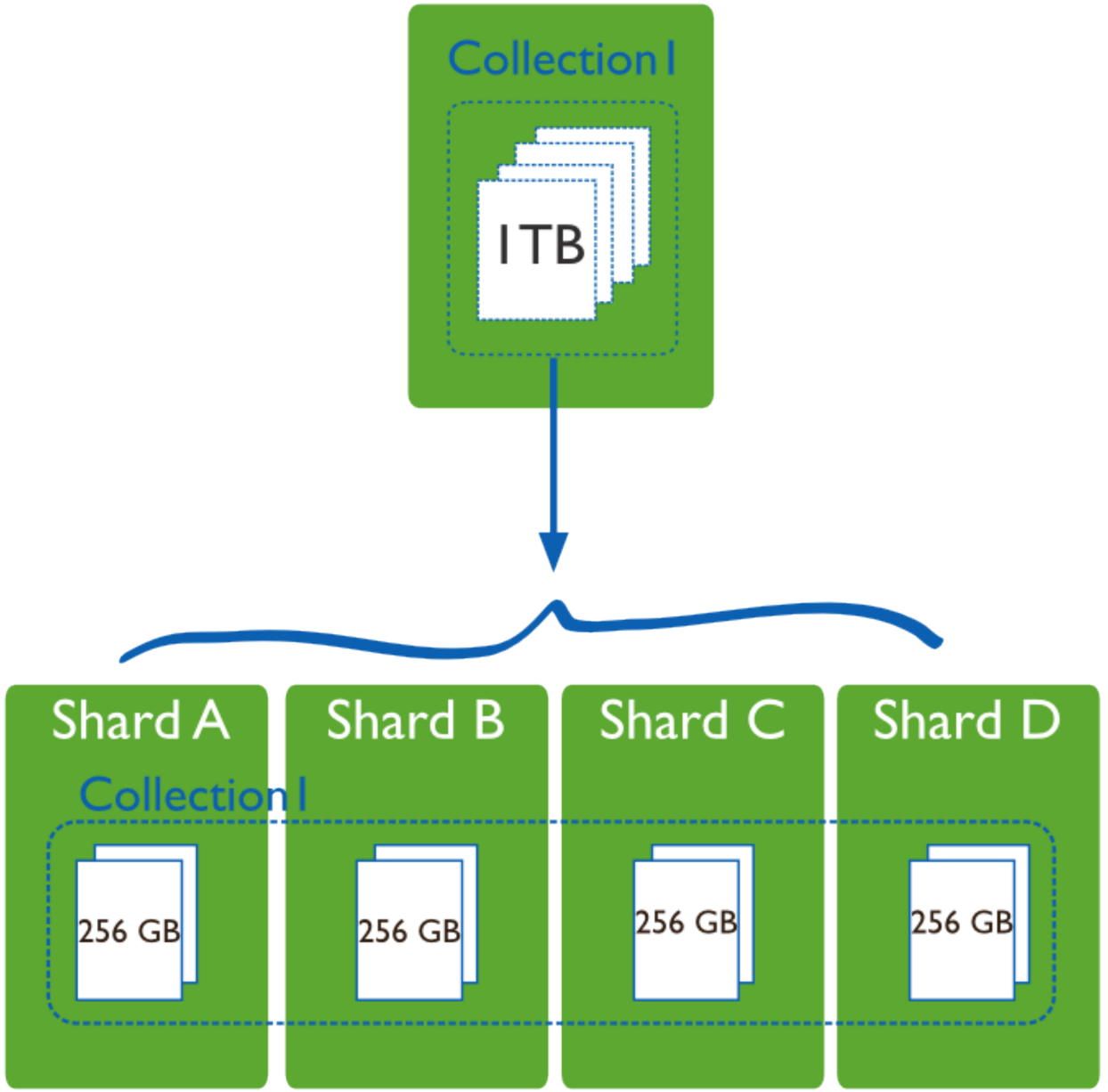
SHARING

- So what if you want to update more than 5 times a second?
  - Contention between writes
  - Timeout
  - Loose data



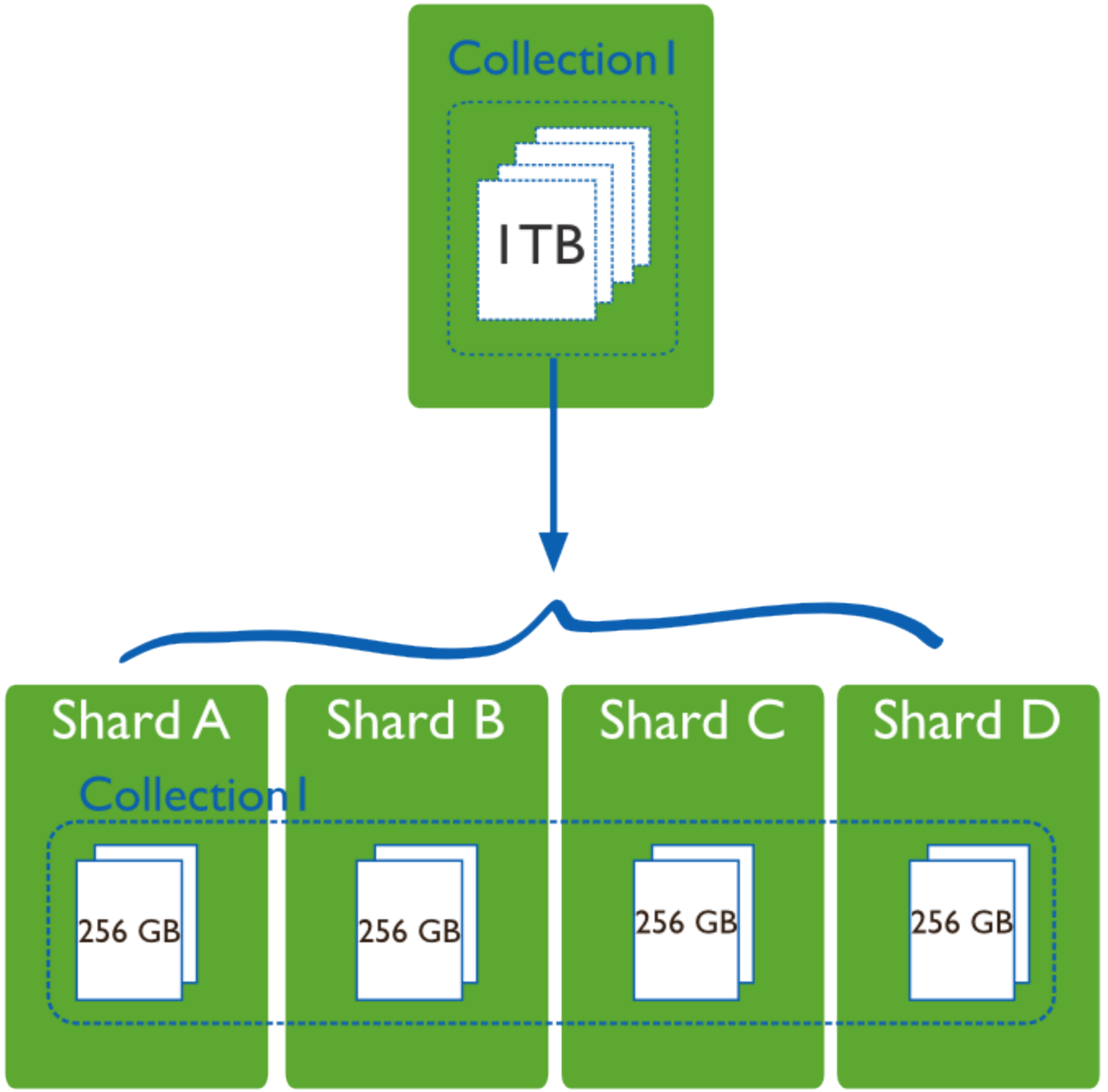
SHARING

- Sharding is a type of database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards
- The word shard means a small part of a whole



SHARING

- Shards are created to best match the performance of the system
- Data can be coallated on demand or stored/updated separately



COUNTER EXAMPLE

SHARDING

- Google example:
  - appengine-sharded-counters

```
import webapp2
from webapp2_extras import jinja2

import general_counter
import simple_counter


DEFAULT_COUNTER_NAME = 'TOUCHES'


class CounterHandler(webapp2.RequestHandler):
    """Handles displaying counter values and requests to increment a counter.
    Uses a simple and general counter and allows either to be updated.
    """
    @webapp2.cached_property
    def jinja2(self):
        return jinja2.get_jinja2(app=self.app)

    def render_response(self, template, **context):
        rendered_value = self.jinja2.render_template(template, **context)
        self.response.write(rendered_value)

    def get(self):
        """GET handler for displaying counter values."""
        simple_total = simple_counter.get_count()
        general_total = general_counter.get_count(DEFAULT_COUNTER_NAME)
        self.render_response('counter.html', simple_total=simple_total,
                             general_total=general_total)

    def post(self):
        """POST handler for updating a counter which is specified in payload."""
        counter = self.request.get('counter')
        if counter == 'simple':
            simple_counter.increment()
        else:
            general_counter.increment(DEFAULT_COUNTER_NAME)
        self.redirect('/')


APPLICATION = webapp2.WSGIApplication([('/', CounterHandler)],
                                      debug=True)
```

SHARDING

```
    def get(self):
        """GET handler for displaying counter values."""
        simple_total = simple_counter.get_count()
        general_total = general_counter.get_count(DEFAULT_COUNTER_NAME)
        self.render_response('counter.html', simple_total=simple_total,
                             general_total=general_total)

    def post(self):
        """POST handler for updating a counter which is specified in payload."""
        counter = self.request.get('counter')
        if counter == 'simple':
            simple_counter.increment()
        else:
            general_counter.increment(DEFAULT_COUNTER_NAME)
        self.redirect('/')


APPLICATION = webapp2.WSGIApplication([('/', CounterHandler)],
                                      debug=True)
```

WEB APP

POST A TOUCH

SHARDING

```python
import random

from google.appengine.ext import ndb

NUM_SHARDS = 20

class SimpleCounterShard(ndb.Model):
    """Shards for the counter."""
    count = ndb.IntegerProperty(default=0)

def get_count():
    """Retrieve the value for a given sharded counter.
    Returns:
        Integer; the cumulative count of all sharded counters.
    """
    total = 0
    for counter in SimpleCounterShard.query():
        total += counter.count
    return total


@ndb.transactional
def increment():
    """Increment the value for a given sharded counter."""
    shard_string_index = str(random.randint(0, NUM_SHARDS - 1))
    counter = SimpleCounterShard.get_by_id(shard_string_index)
    if counter is None:
        counter = SimpleCounterShard(id=shard_string_index)
    counter.count += 1
    counter.put()
```

COUNTER CLASS WITH ONE PROPERTY

READ

INCREMENT

SHARDING

ATOMIC WRITES
60 SEC MAX

```
@ndb.transactional
def increment():
    """Increment the value for a given sharded counter."""
    shard_string_index = str(random.randint(0, NUM_SHARDS - 1))
    counter = SimpleCounterShard.get_by_id(shard_string_index)
    if counter is None:
        counter = SimpleCounterShard(id=shard_string_index)
    counter.count += 1
    counter.put()
```

PICK RANDOM NUMBER,
CREATE ID WITH NUMBER,
GET (OR CREATE) ENTITY,
INCREMENT THAT ONE

SHARDING

Entity Kind

SimpleCounterShard | List Entities | Create New Entity | Select a different namespace

	Key	Write Ops	ID	Key Name	count
☐	aghkZXZ-...	4		0	4
☐	aghkZXZ-...	4		1	7
☐	aghkZXZ-...	4		10	12
☐	aghkZXZ-...	4		11	5
☐	aghkZXZ-...	4		12	8
☐	aghkZXZ-...	4		13	2
☐	aghkZXZ-...	4		14	4
☐	aghkZXZ-...	4		15	5
☐	aghkZXZ-...	4		16	3
☐	aghkZXZ-...	4		17	6
☐	aghkZXZ-...	4		18	9
☐	aghkZXZ-...	4		19	4
☐	aghkZXZ-...	4		2	2
☐	aghkZXZ-...	4		3	7
☐	aghkZXZ-...	4		4	5
☐	aghkZXZ-...	4		5	10
☐	aghkZXZ-...	4		6	3
☐	aghkZXZ-...	4		7	5
☐	aghkZXZ-...	4		8	10
☐	aghkZXZ-...	4		9	6

20 COUNTERS

SHARDING

```
def get_count():
    """Retrieve the value for a given sharded counter.
    Returns:
        Integer; the cumulative count of all sharded counters.
    """
    total = 0
    for counter in SimpleCounterShard.query():
        total += counter.count
    return total
```

GET ALL ENTITIES,
SUM THEIR COUNTS
RETURN

SHARDING

- More general purpose example also given
- Uses memcache to store the total

```python
import random

from google.appengine.api import memcache
from google.appengine.ext import ndb


SHARD_KEY_TEMPLATE = 'shard-{}-{:d}'


class GeneralCounterShardConfig(ndb.Model):
    """Tracks the number of shards for each named counter."""
    num_shards = ndb.IntegerProperty(default=20)

    @classmethod
    def all_keys(cls, name):
        """Returns all possible keys for the counter name given the config.

        Args:
            name: The name of the counter.

        Returns:
            The full list of ndb.Key values corresponding to all the possible
                counter shards that could exist.
        """
        config = cls.get_or_insert(name)
        shard_key_strings = [SHARD_KEY_TEMPLATE.format(name, index)
                             for index in range(config.num_shards)]
        return [ndb.Key(GeneralCounterShard, shard_key_string)
                for shard_key_string in shard_key_strings]


class GeneralCounterShard(ndb.Model):
    """Shards for each named counter."""
    count = ndb.IntegerProperty(default=0)
```

ASSIGNMENT 2

ASSIGNMENT 2

- Part 1
  - Address the problems we discussed in class
- Part 2
  - Develop a mobile analytics backend

PART 1

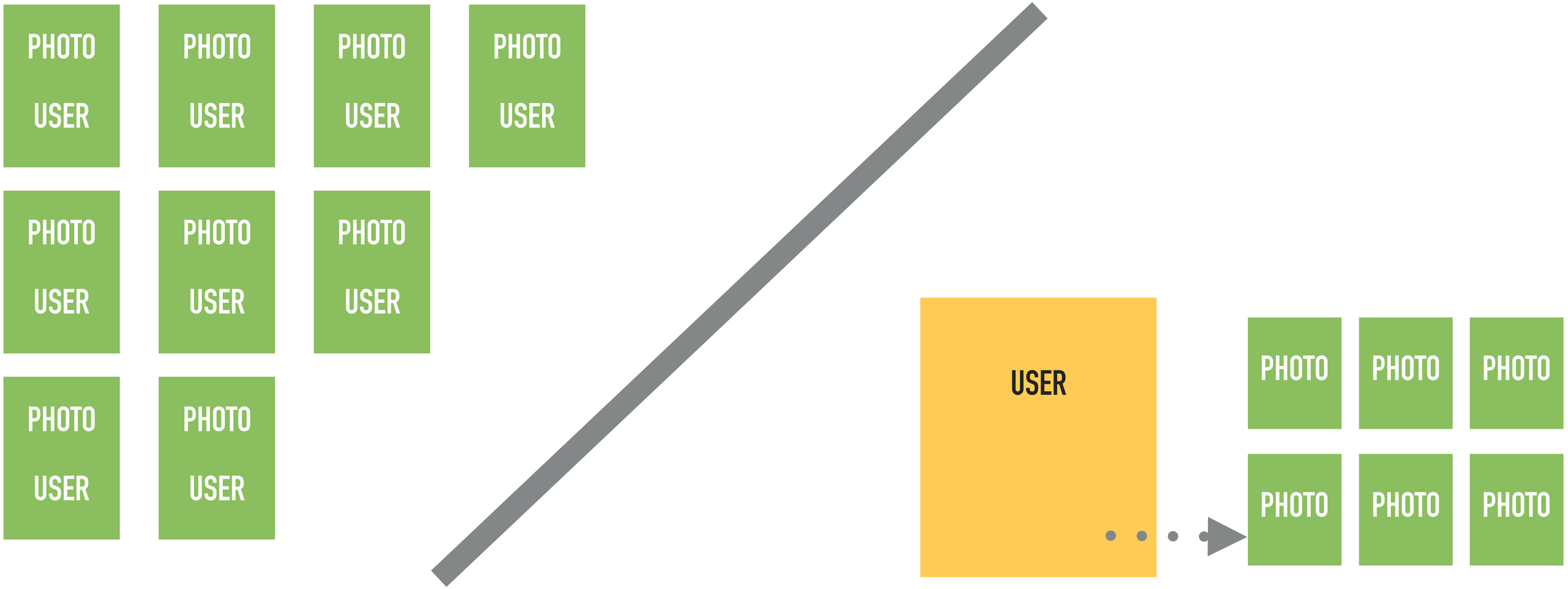
ASSIGNMENT 2 - PART 1

ADDRESSING THE ISSUES

- Address some of the concerns we have with our existing solution
  - Changes to model
  - Increased security
  - Improved functionality

ASSIGNMENT 2 - PART 1

DATA MODEL



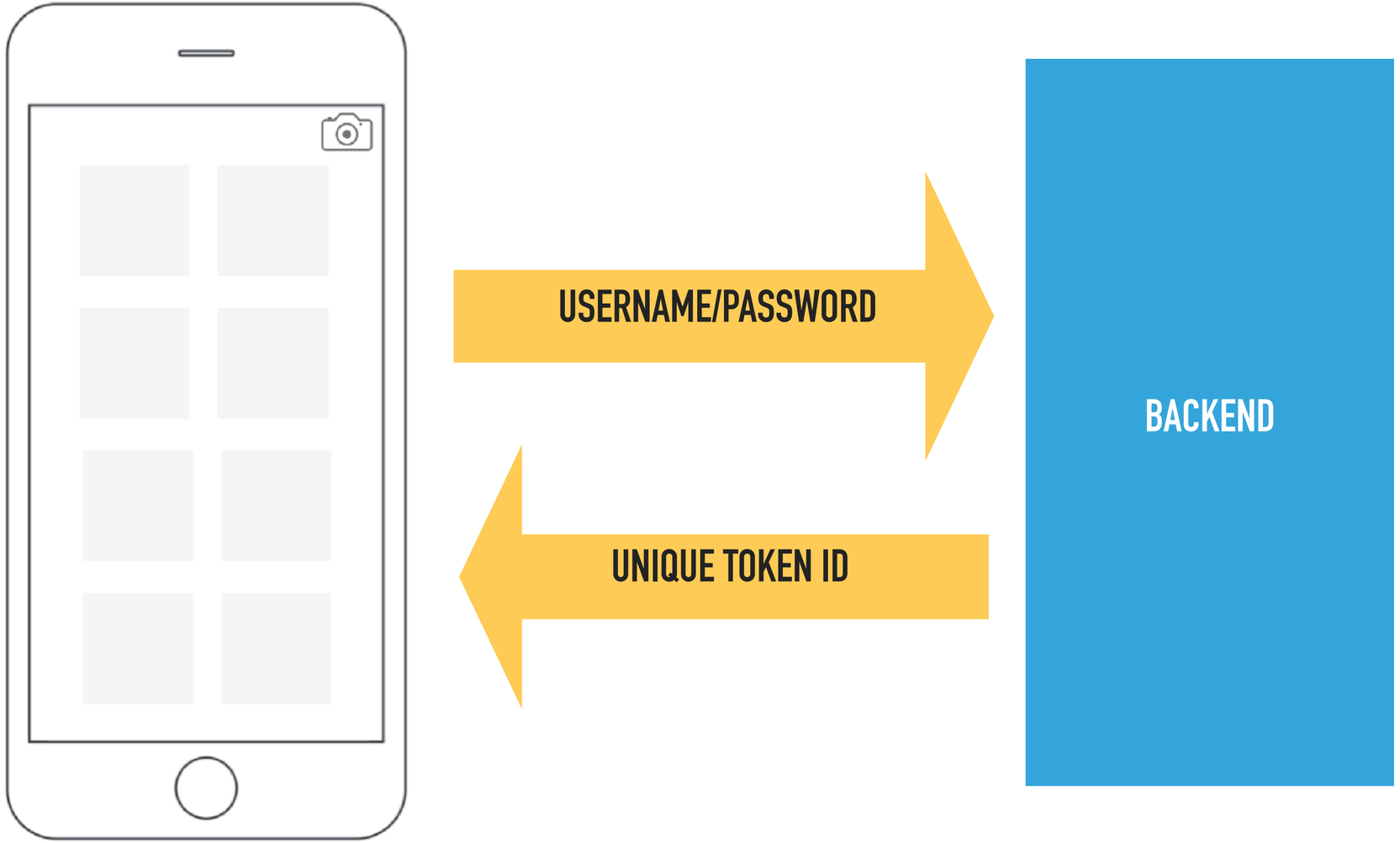
ASSIGNMENT 2 - PART 1

DATA MODEL

- Add a "User" model to the application
  - name (ex. johnny)
  - email (ex. johnny@xx.com)
  - uniqued_id (ex. can be user entity key)
  - photos (all the photos uploaded to this account)
- Update all the queries to take advantage of the new User model
- Update the Photo model to reflect the new User model

ASSIGNMENT 2 - PART 1

SECURITY



ASSIGNMENT 2 - PART 1

SECURITY

```
http://--.appspot.com/user/authenticate/?
username=XXX&password=XXXX

# This should return a unique id token that would be stored
# on the device
```

ASSIGNMENT 2 - PART 1

SECURITY

```
# Get a json list of most recent submitted pictures
http://--.appspot.com/user/<USERNAME>/json/?id_token=XXXX


# See a list of the most recent on a web page (useful for
debugging
http://--.appspot.com/user/<USERNAME>/web/?id_token=XXXX


# Endpoint for posting images to server. There is an
optional "caption" parameter that you can use.
http://--.appspot.com/post/<USERNAME>/?id_token=XXXX
```

ASSIGNMENT 2 - PART 1

FUNCTIONALITY

```
# Add ability to delete a photo
http://--.appspot.com/image/<key>/delete/?id_token=XXX
```

PART 2

ASSIGNMENT 2 - PART 2

MOBILE ANALYTICS PLATFORM

- Create a mobile analytics platform
  - Track unique users - visits, time duration
  - Track events - clicks, navigation
  - Touch heat map - all touches (for a reason)
  - Daily summary email

ASSIGNMENT 2 - PART 2

MOBILE ANALYTICS PLATFORM

- You will design all the aspects of the analytics platform using Google App Engine
- Simple companion iOS app (reuse old app)

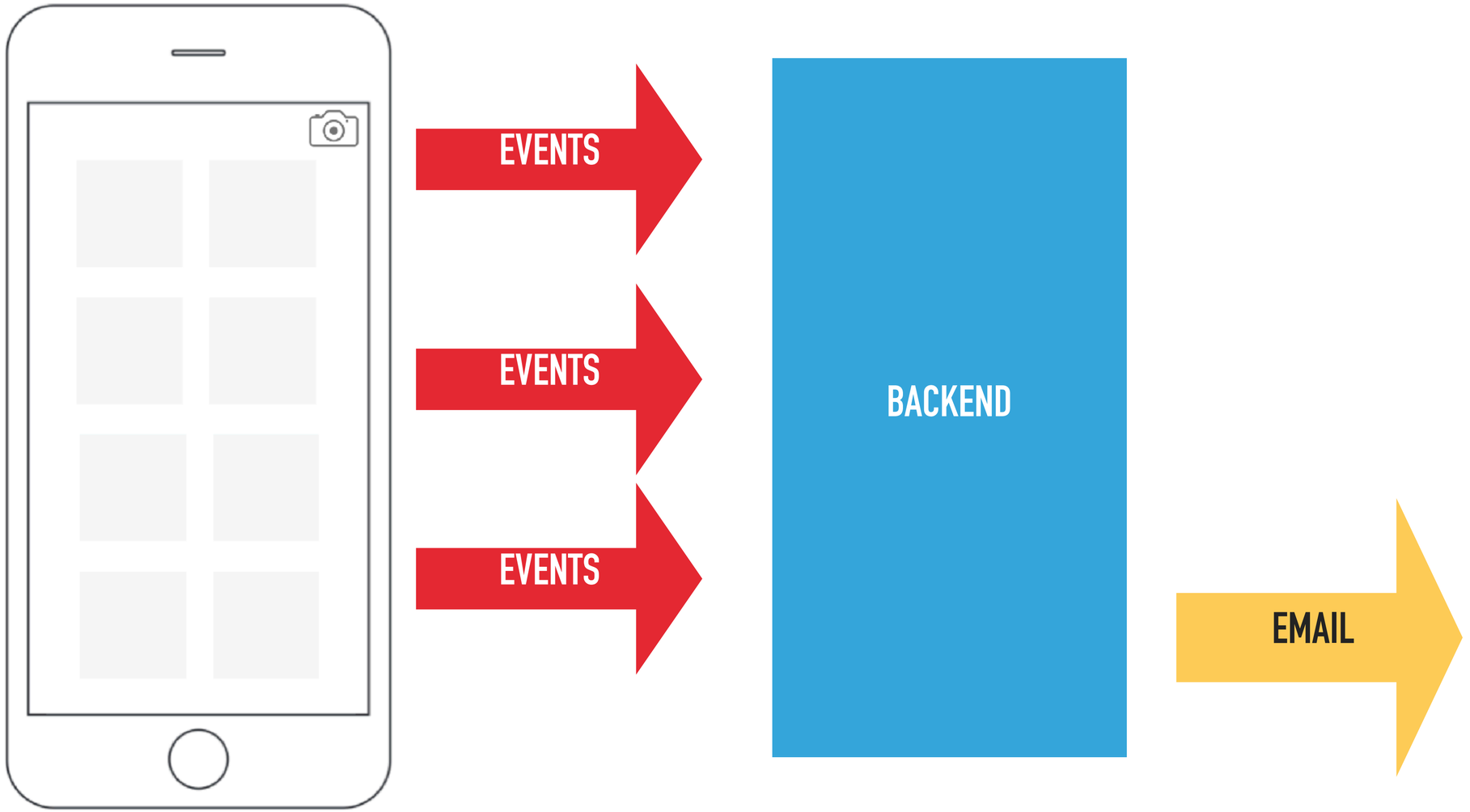
ASSIGNMENT 2 - PART 2

MOBILE ANALYTICS PLATFORM

- Next week
  - Sharding
  - Cron
  - Task queue
  - Microservices
  - Email

ASSIGNMENT 2 - PART 1

SECURITY



LAB

LAB

```
# 1 to many relationships is Datastore?

class Photo(ndb.Model):
 image = nab.BlobProperty()

class User(ndb.Model):
 name = ndb.StringProperty()

 photos = ndb.KeyProperty(kind='Photo', repeated=True)
 # or
 photos = nab.StructuredProperty(kind='Photo',
                                 repeated=True)
```

LAB

- Although the Photo instances are defined using the same syntax as for model classes they are not full-fledged entities
- They don't have their own keys in the Datastore
- They cannot be retrieved independently of the User entity to which they belong

```
# 1 to many relationships is Datastore?

class Photo(ndb.Model):
 image = nab.BlobProperty()

class User(ndb.Model):
 name = ndb.StringProperty()

 photos = ndb.KeyProperty(kind='Photo', repeated=True)
 # or #
 photos = nab.StructuredProperty(kind='Photo',
                                 repeated=True)
```

THE UNIVERSITY OF CHICAGO

MPCS 51033 • SPRING 2017 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS