THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2019 • SESSION 2

BACKENDS FOR MOBILE APPLICATIONS

CLASS NEWS

CLASS NEWS

- Welcome Hannah and check out office hours

TA Intro & Office Hours #4

Open **hbennett766** opened this issue 5 days ago · 0 comments

hbennett766 commented 5 days ago

Hey everyone 👋 I'm Hannah, your TA this quarter. This is my first time working with the backends class, but I've been TA-ing the iOS classes for a few years.

Here's my resume, if you're interested.

Office hours this week will be virtual and on demand. Email me at hbennett766@gmail.com to set a google hangout.

My in-person office hours will be held on Sundays. Below is the schedule. Unless otherwise noted, they will be from 10a-12p at Dollop Coffee. It's right on the L, they've got lots of tables, and of course coffee.

October 13, 2-4p, at the Harold Washington Library (week 2)
October 20 (week 3)
Virtual and on-demand office hours (week 4), I'm out of town Oct 25-27
November 3 (week 5)
November 10 (week 6)
November 17 (week 7)
November 24 (week 8)
December 1 (week 9)
December 8 (week 10)

If you've got strong positive or negative opinions on this plan, be it time, location, or whatever, let me know. I look forward to seeing you all on Tuesday!

MOBILE APP BACKEND SERVICES WITH GCP

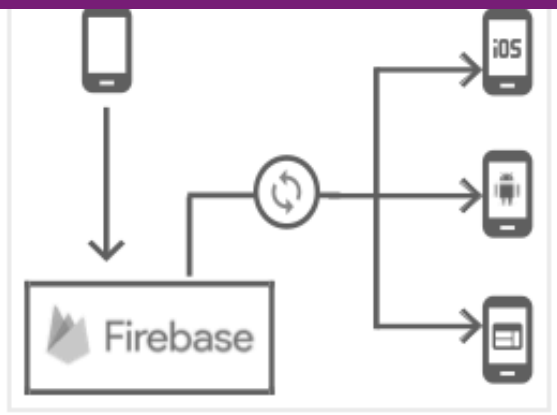
BACKEND SERVICES WITH GCP

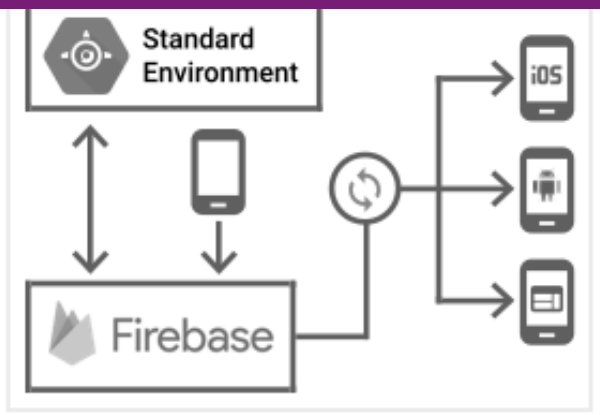
- Apple's trend is to do almost everything on device. Why do we need a backend?
- Most mobile apps and games need a backend service for things that can’t be done solely on-device
  - Sharing and processing data from multiple users
  - Storing large files
  - Group analytics statistics (leaderboard)

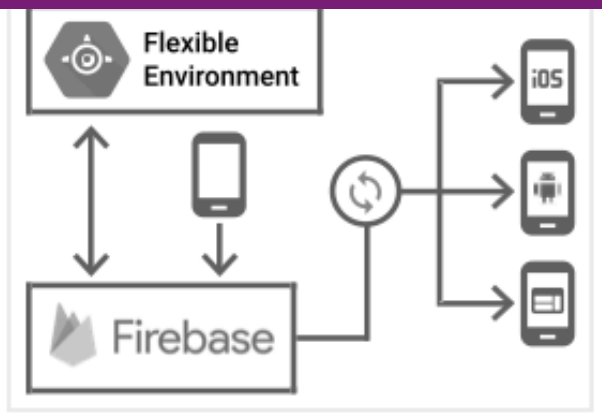
BACKEND SERVICES WITH GCP

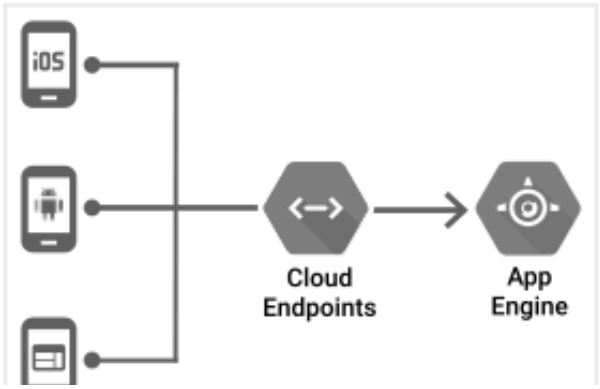
- Building a backend service for a mobile app is similar to building a web-based service, with some additional considerations
  - Limit on-device data storage
  - Synchronize data across multiple devices
  - Handle the online/offline transition gracefully
  - Send notifications and messages
  - Minimize battery drain (be a good citizen)

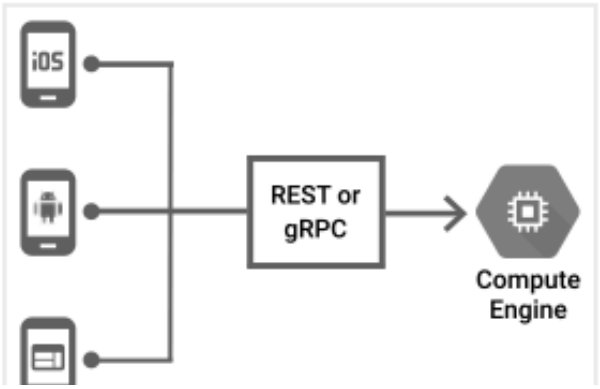
BACKEND SERVICES WITH GCP







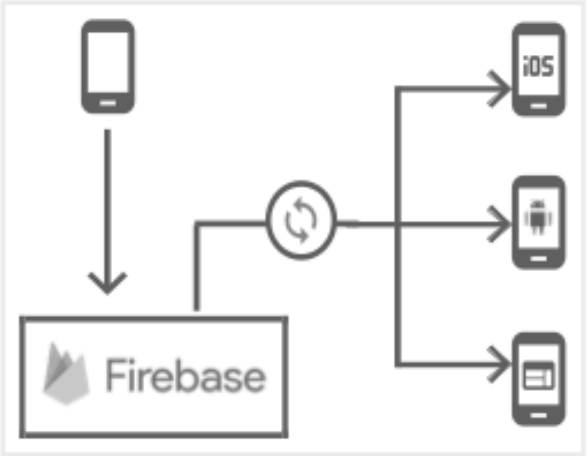


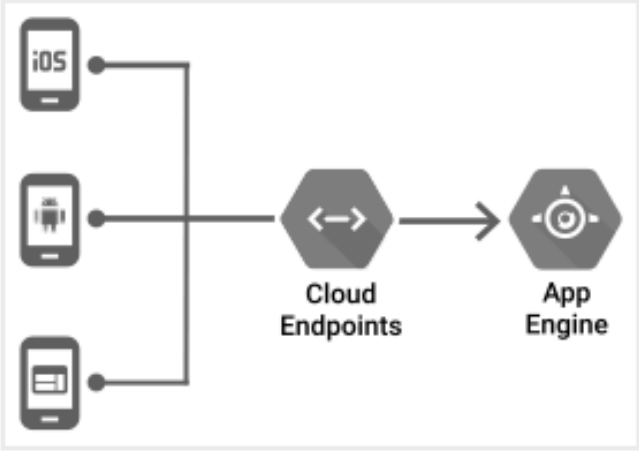


DESIGN PATTERNS USING GCP FOR BACKENDS SERVICES

BACKEND SERVICES WITH GCP

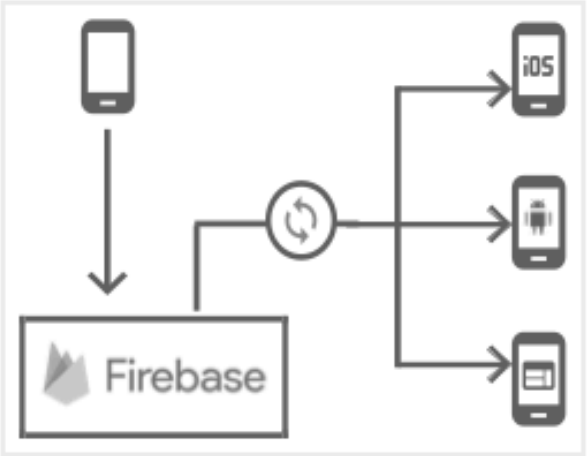
- Two-tier architecture with Firebase
  - Mobile app and Firebase manipulate data directly
  - Security and data validation handled in console

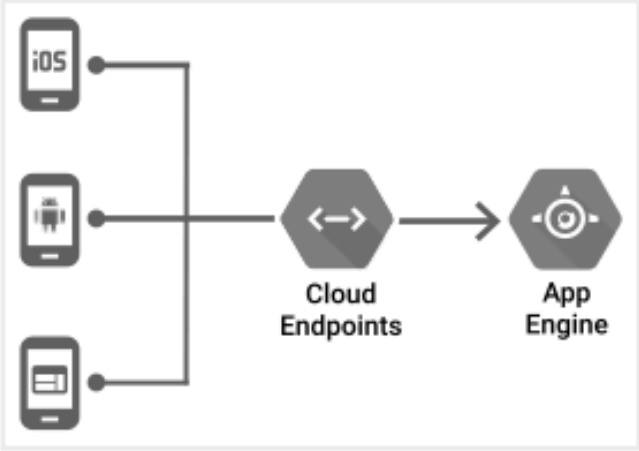




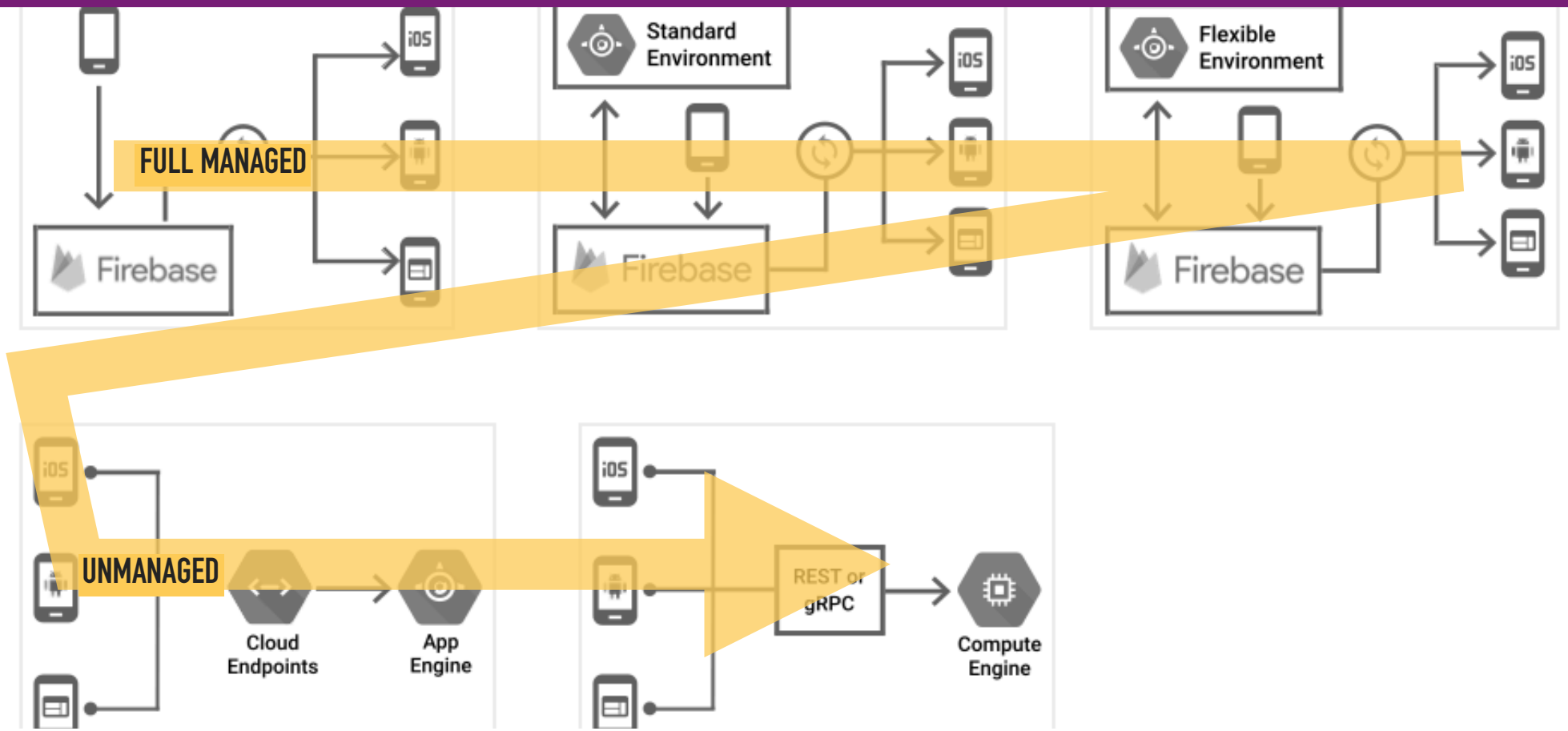
BACKEND SERVICES WITH GCP

- Three-tier architecture with App Engine
  - Communication layer between app and backend
  - Responsible for authentication and data-validation



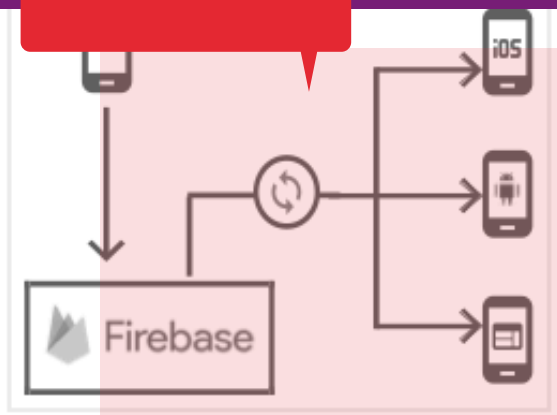


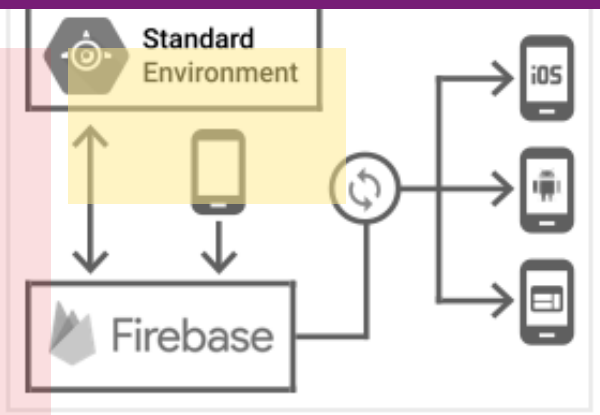
BACKEND SERVICES WITH GCP

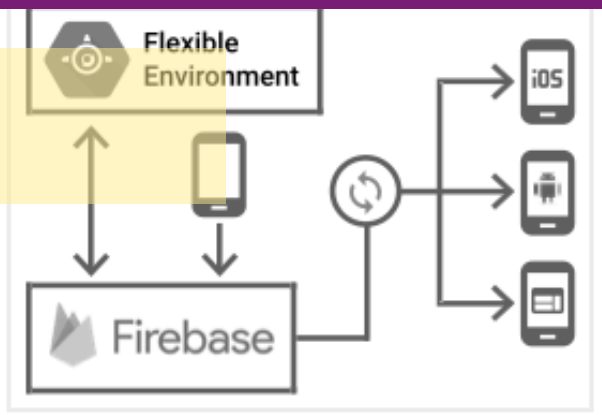


BACKEND SERVICES WITH GCP

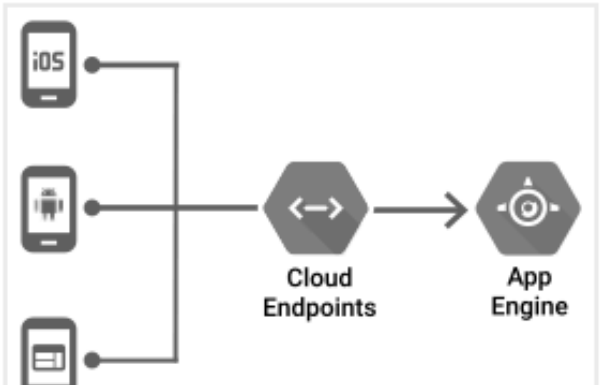
LATER IN THE COURSE

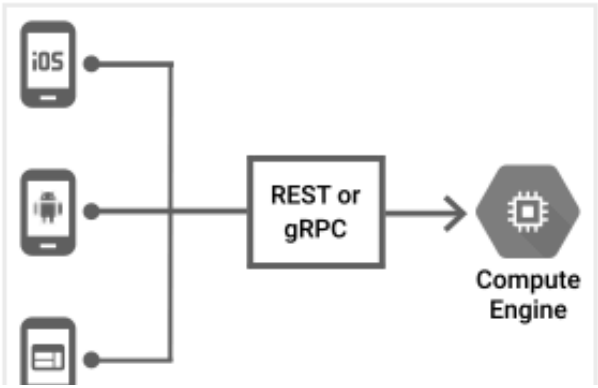






HYBRID PROVIDES MIX OF EASE OF USE AND FLEXIBILITY



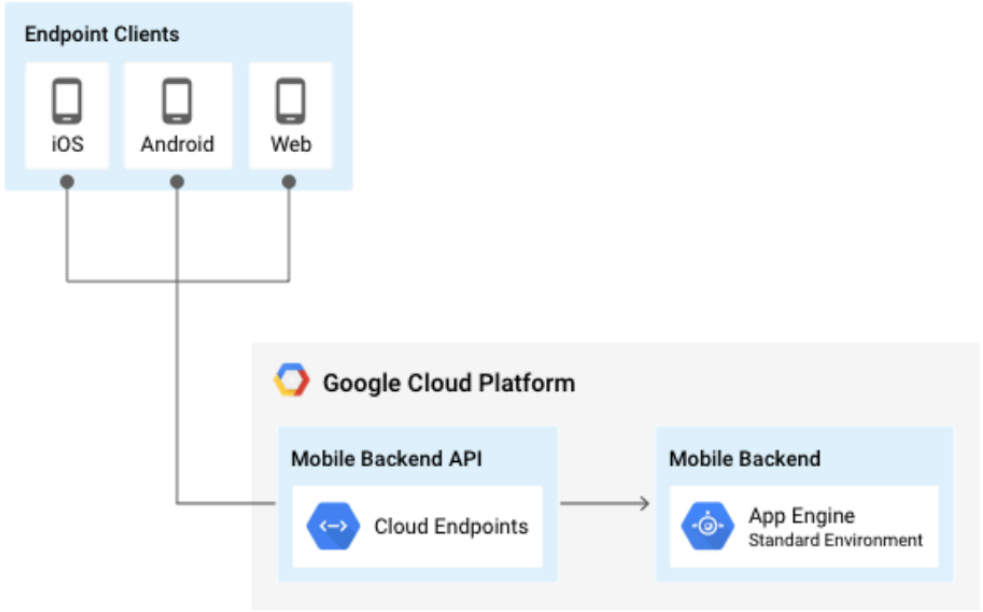


CLOUD ENDPOINTS

BACKEND SERVICES WITH GCP

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
- Client libraries are generated by Cloud Endpoints to make direct API calls from your mobile app



BACKEND SERVICES WITH GCP

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.

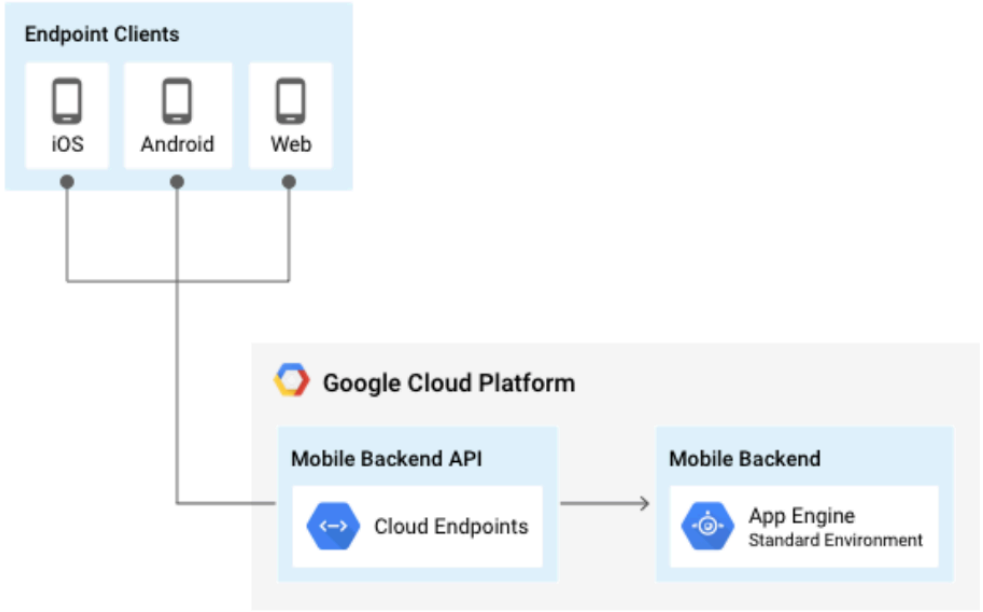
Cloud Endpoints Frameworks supports the following API management features provided by Cloud Endpoints:

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

BACKEND SERVICES WITH GCP

CLOUD ENDPOINTS

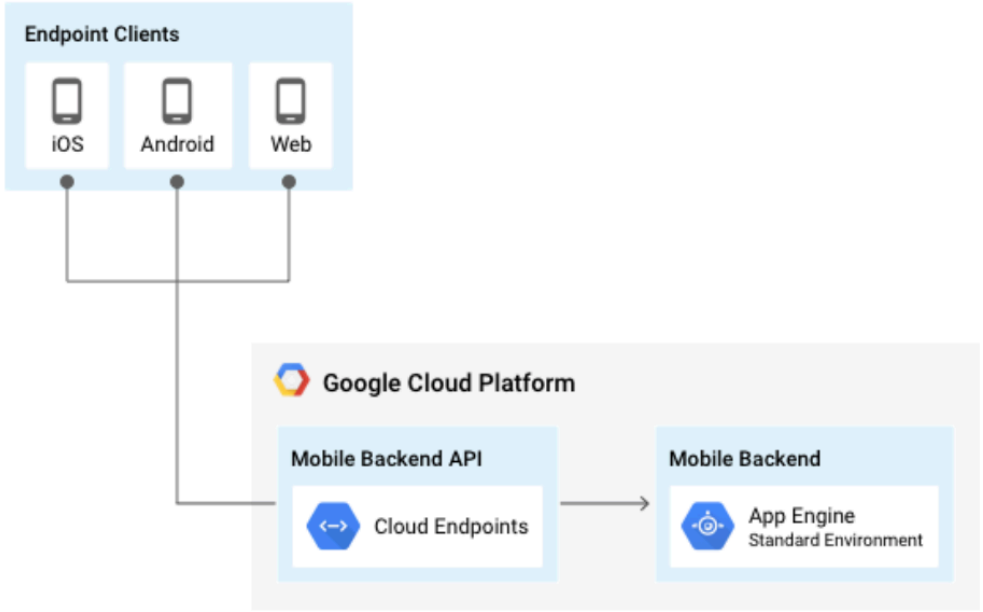
- Recommended use
  - Automated generation of client libraries that apps can use to call the backend service directly (ie ease of use)
  - Reducing on-device storage by moving files to Cloud Storage
  - Sending notifications by calling Cloud Messaging



BACKEND SERVICES WITH GCP

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.



BACKEND SERVICES WITH GCP

CLOUD ENDPOINTS

Cloud Endpoints > Documentation

Cloud Endpoints Quickstart

Contents

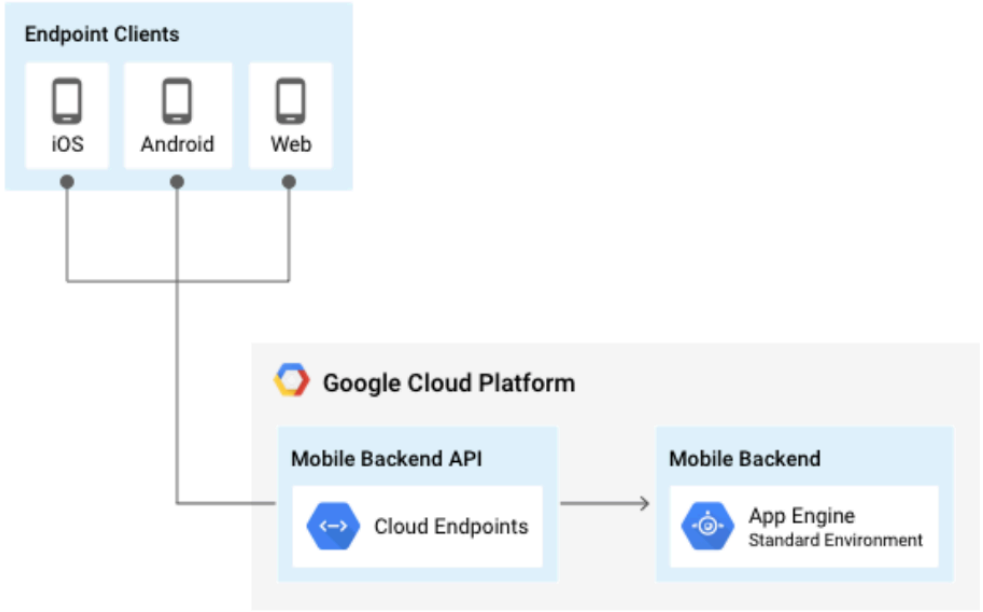
HTTPS://CLOUD.GOOGLE.COM/ENDPOINTS/DOCS/QUICKSTART-ENDPOINTS

This QuickStart walks you through deploying a sample API which Endpoints manages. The

BACKEND SERVICES WITH GCP

CLOUD ENDPOINTS

- Summary
  - If you are only going to use this endpoint

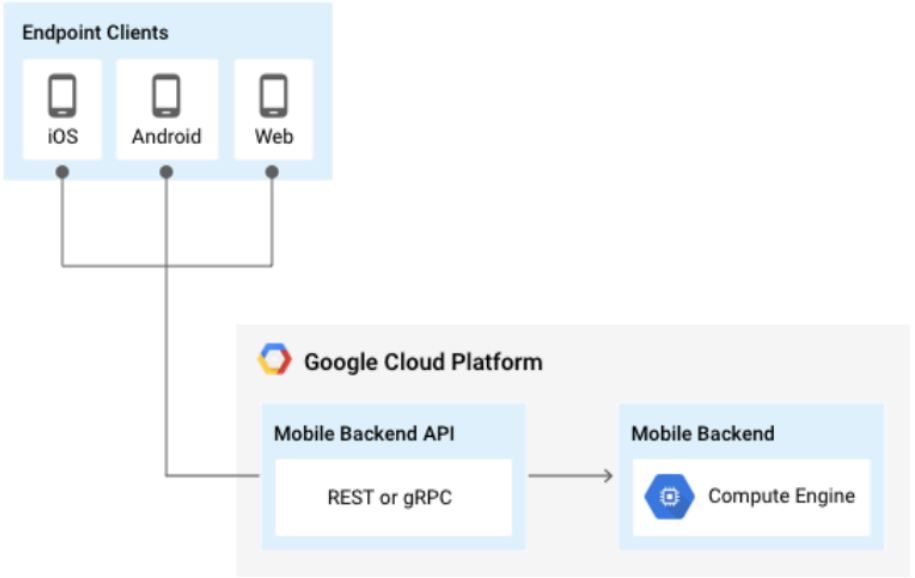


COMPUTE ENGINE

BACKEND SERVICES WITH GCP

COMPUTE ENGINE

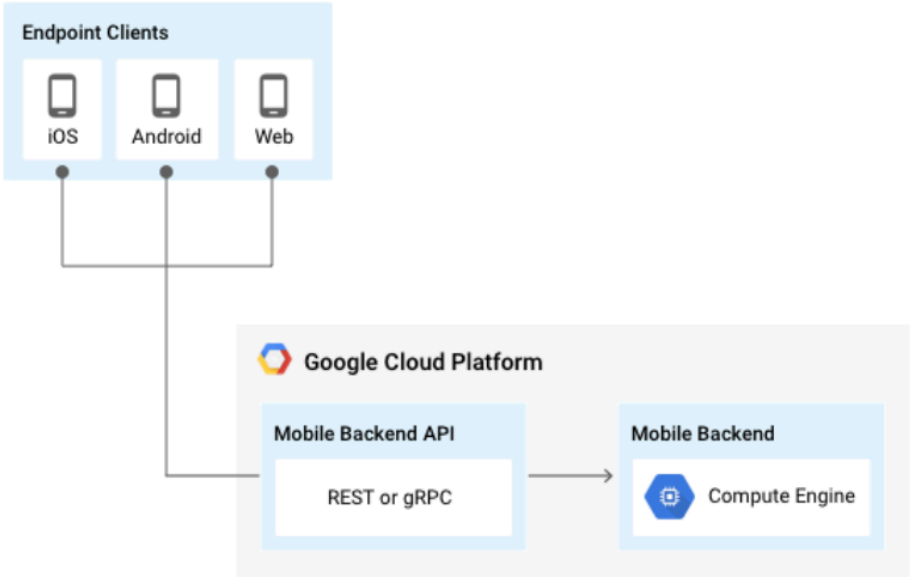
- Computer Engine lets you create and run virtual machines on Google infrastructure
- You have administrator rights to the server and full control over its configuration
  - You are responsible for updates and maintenance :(



BACKEND SERVICES WITH GCP

COMPUTE ENGINE

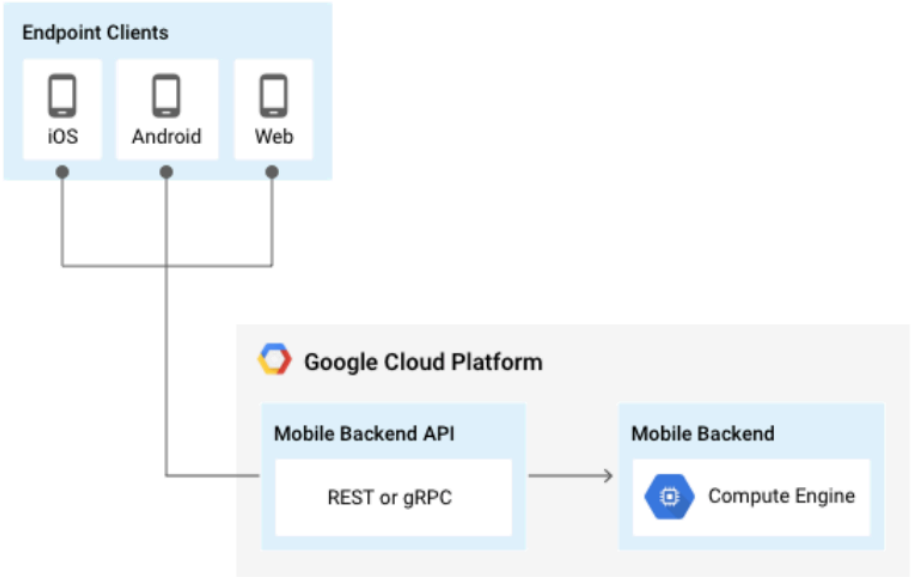
- Recommended for:
  - Porting an existing backend service running on an on-premise server or a virtual machine.
  - Backend services that require a custom server or third-party libraries.



BACKEND SERVICES WITH GCP

COMPUTE ENGINE

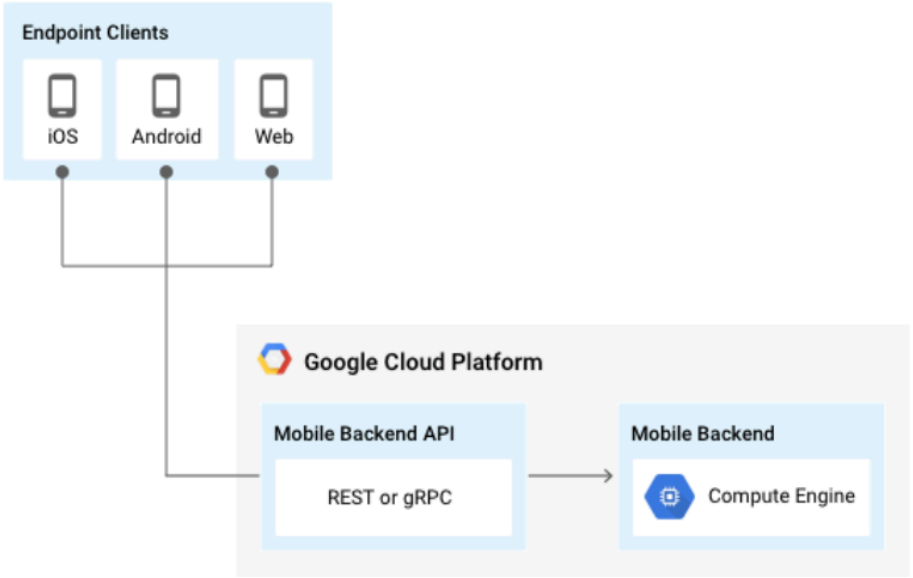
- Not recommended for:
  - Apps that require automatic real-time data synchronization across devices.
  - Automatic maintenance; you must maintain and upgrade the server yourself.
  - Automatic scaling; you must manually configure and manage an autoscaler.



BACKEND SERVICES WITH GCP

COMPUTE ENGINE

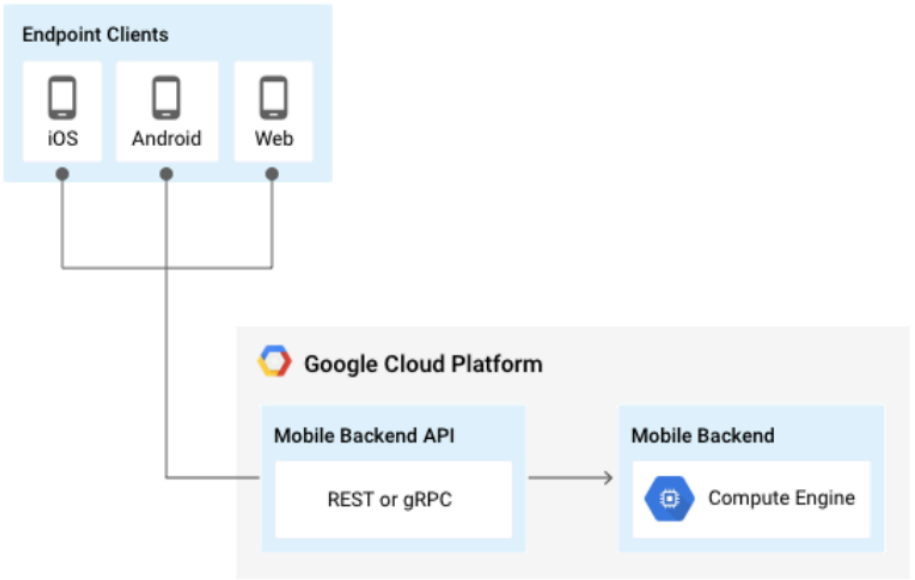
- Protocols used to connect to compute engine
  - REST - architecture based on HTTP
  - gRPC - framework using http/2



APP ENGINE

BACKEND SERVICES WITH GCP

- Everything that follows



APP WALK THROUGH: PHOTO TIMELINE

PHOTO TIMELINE

- Backend for a photo timeline application
- App behaviors
  - Users post pictures with a comment
  - Users retrieves feed of pictures
  - User sees top rated photos

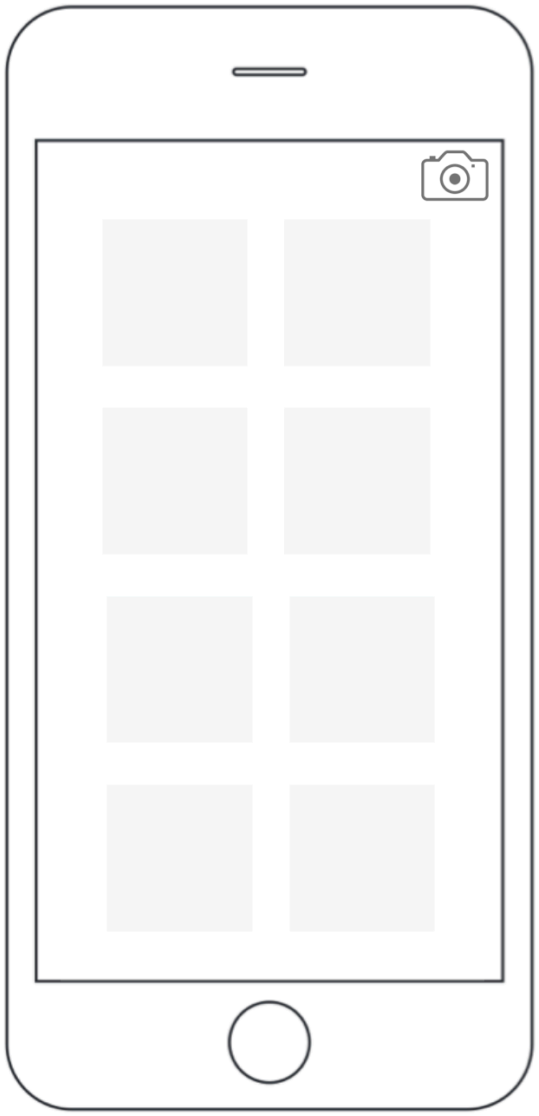


PHOTO TIMELINE

- Server behavior
  - Upload photos and store them in Google Cloud Storage
  - Store photo information and metadata in Datastore
  - Process with Google Vision APIs
  - Retrieve datastore data
  - Retrieve photos
  - Retrieve photo data in JSON

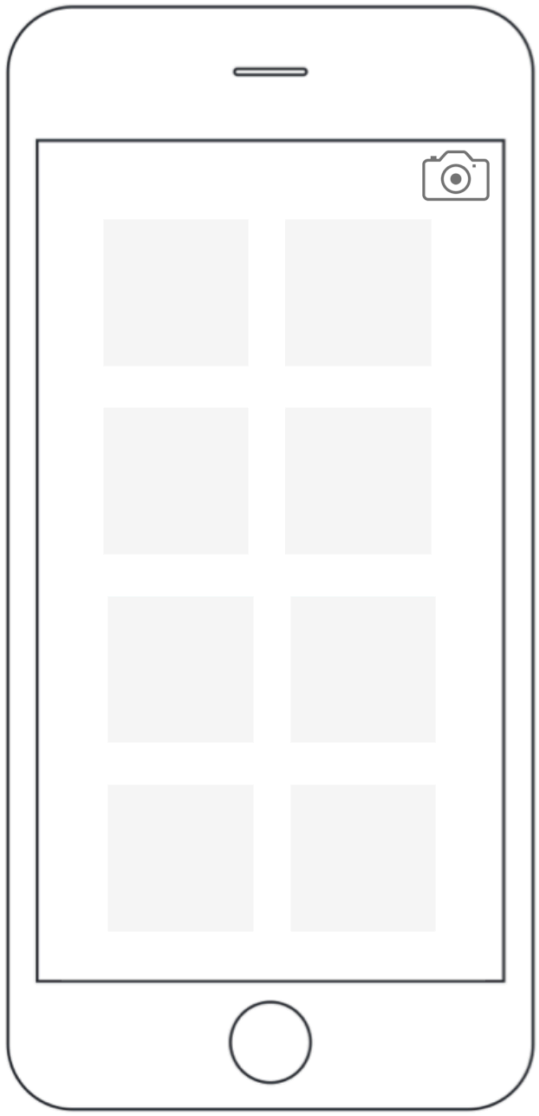


PHOTO TIMELINE

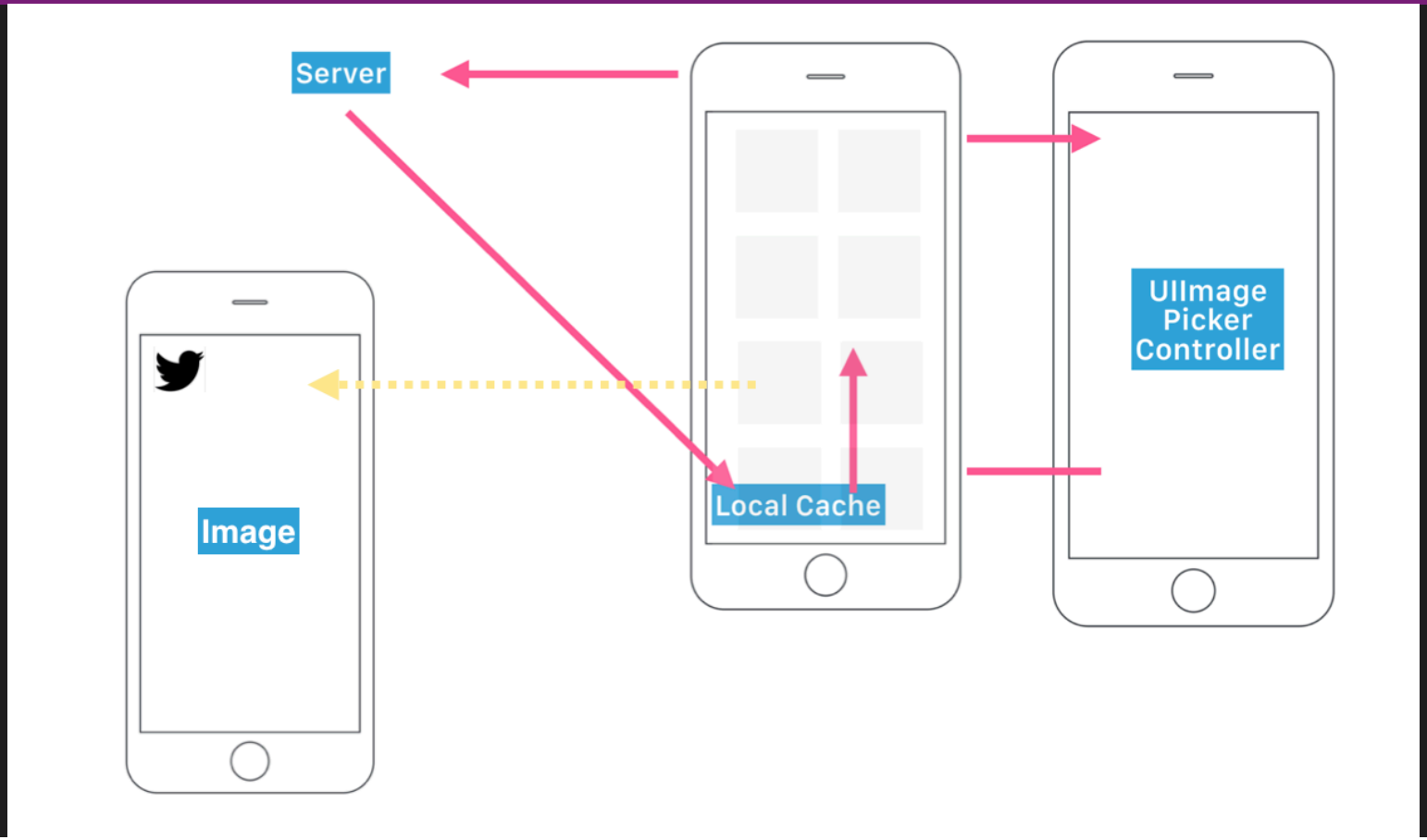


PHOTO TIMELINE

- Application files

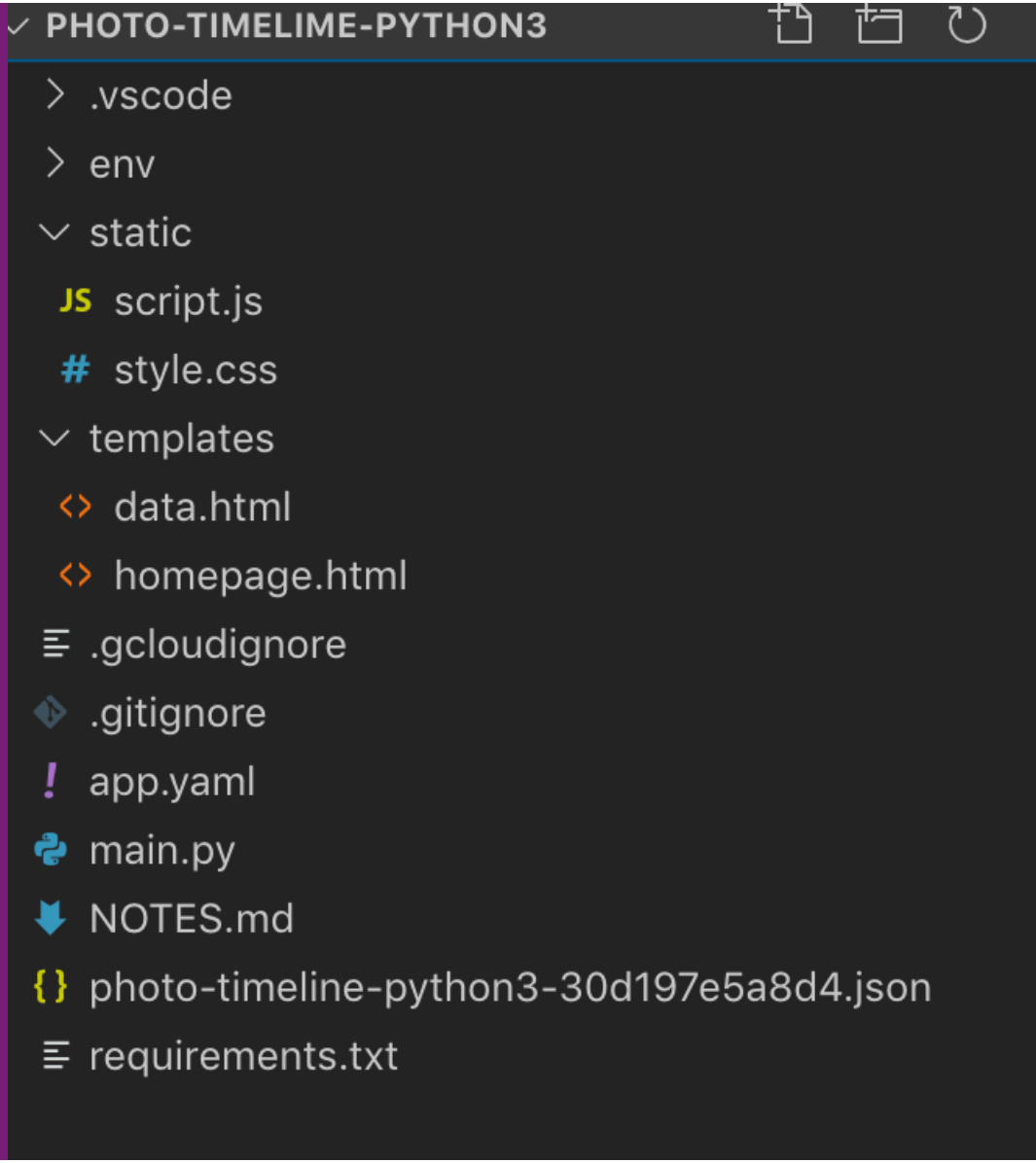


PHOTO TIMELINE

```
templates/
  homepage.html    /* HTML template that uses Jinja2 */
app.yaml           /* App Engine application configuration file */
main.py            /* Python Flask web application */
requirements.txt   /* List of dependencies for the project */
```

- Application files (needed to run the application)

PHOTO TIMELINE

- Supporting files
  - .gcloudignore
  - .gitignore
  - xxxx.json
  - Templates
  - Static

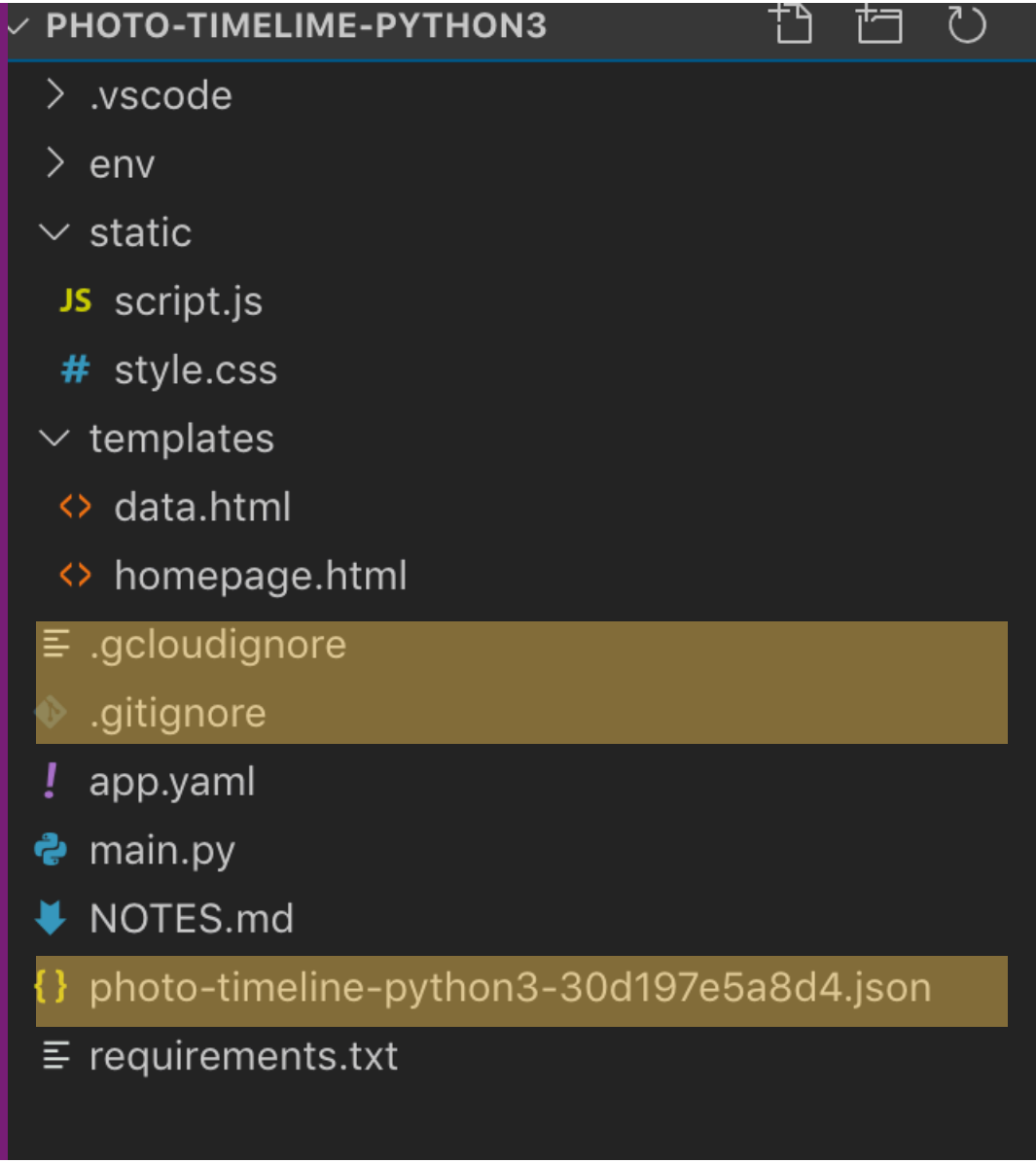
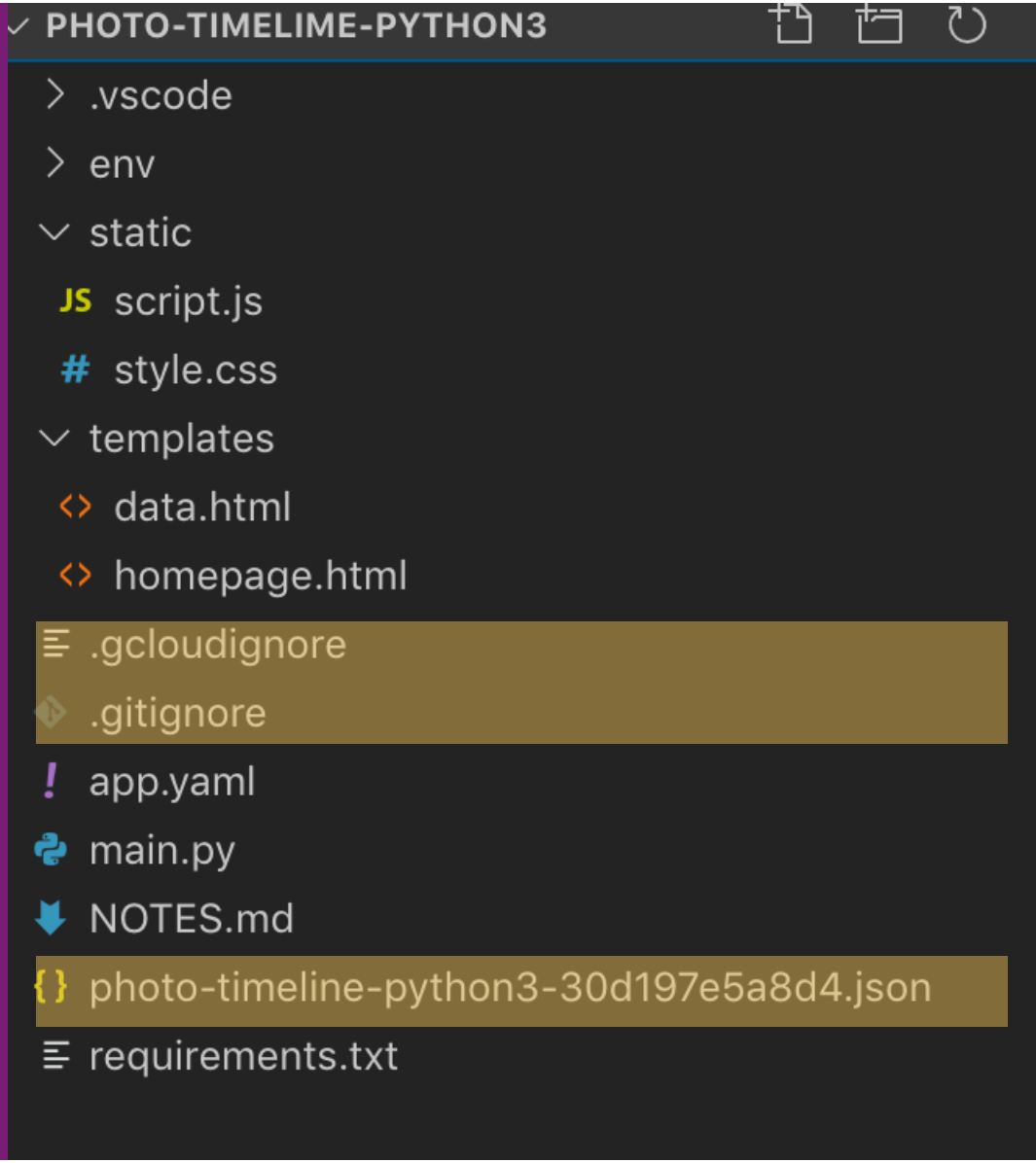


PHOTO TIMELINE

- We are building a web interface to speed development



APP.YAML

PHOTO TIMELINE

APP.YAML

- app.yml file
- More options
  - Specify machines
  - Global variables

How should the application route requests

app.yaml

```
1   runtime: python37
2
3   handlers:
4     # This configures Google App Engine to serve the files in the app's static
5     # directory.
6   - url: /static
7     static_dir: static
8
9     # This handler routes all requests not caught above to your main app. It is
10    # required when static routes are defined, but can be omitted (along with
11    # the entire handlers section) when there are no static files defined.
12  - url: /.*
13    script: auto
14
15
16  env_variables:
17    CLOUD_STORAGE_BUCKET: photo-timeline-python3
18
19
20
```

Runtime

Environment Variables

MAIN.PY

PHOTO TIMELINE

MAIN.PY

- Python Flask web application
  - Werkzeug and Jinja roots
- Availables on App Engine Flex and now in standard environment for Python3

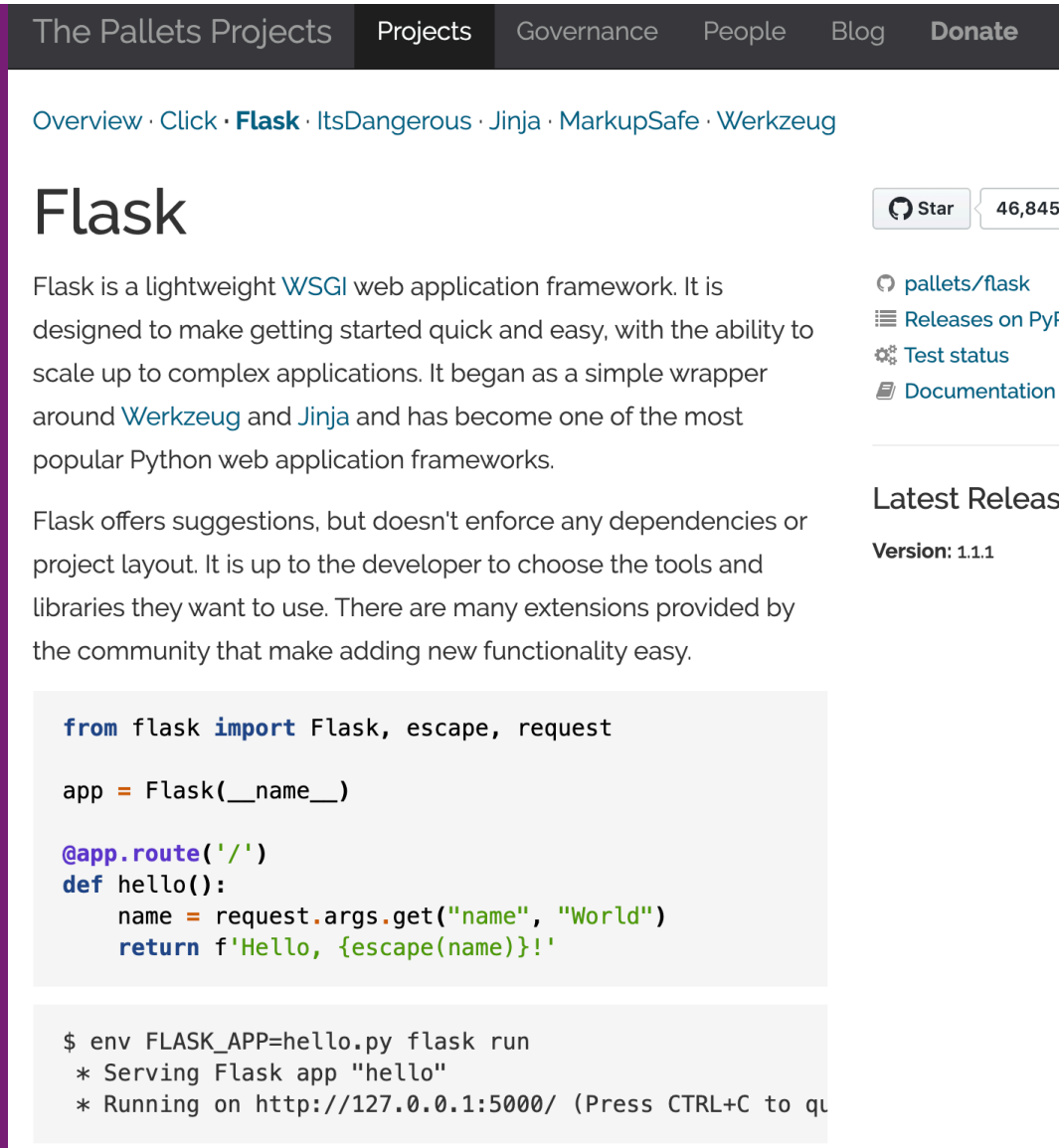


PHOTO TIMELINE

MAIN.PY

- Imports all required python and CGP modules
- Declare some global variables
- Starts the application
- Handles routing

```python
import datetime as datetime
import logging
import os
import json

from flask import Flask, redirect, render_template, request

from google.cloud import datastore
from google.cloud import storage

CLOUD_STORAGE_BUCKET = os.environ.get('CLOUD_STORAGE_BUCKET')

# 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():…


@app.errorhandler(500)
def server_error(e):…


if __name__ == '__main__':
    # This is used when running locally. Gunicorn is used to run the
    # application on Google App Engine. See entrypoint in app.yaml.
```

PHOTO TIMELINE

MAIN.PY

- Imports libraries

```python
import datetime as datetime
import logging
import os
import json

from flask import Flask, redirect, render_template, request

from google.cloud import datastore
from google.cloud import storage

CLOUD_STORAGE_BUCKET = os.environ.get('CLOUD_STORAGE_BUCKET')

# 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():…


@app.errorhandler(500)
def server_error(e):…


if __name__ == '__main__':
    # This is used when running locally. Gunicorn is used to run the
    # application on Google App Engine. See entrypoint in app.yaml.
```

PHOTO TIMELINE

MAIN.PY

- Declare environmental variables
- App Engine will read these directly from the app
- Set them locally to keep the same code base

```python
import datetime as datetime
import logging
import os
import json

from flask import Flask, redirect, render_template, request

from google.cloud import datastore
from google.cloud import storage

CLOUD_STORAGE_BUCKET = os.environ.get('CLOUD_STORAGE_BUCKET')

# 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():…


@app.errorhandler(500)
def server_error(e):…


if __name__ == '__main__':
    # This is used when running locally. Gunicorn is used to run the
    # application on Google App Engine. See entrypoint in app.yaml.
```

PHOTO TIMELINE

MAIN.PY

- Route requests to appropriate functions

URL Routing

```
# 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():…
```

PHOTO TIMELINE

MAIN.PY

http://localhost:8080/

```
# 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():…
```

PHOTO TIMELINE

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():…
```

http://localhost:8080/data

PHOTO TIMELINE

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():…
```

http://localhost:8080/
upload_photos

PHOTO TIMELINE

MAIN.PY

example.com?arg1=value1&arg2=value2

```
arg1 : value1

arg2 : value2
```

PHOTO TIMELINE

MAIN.PY

http://10.1.1.1:5000/login?
username=alex&password=pw1

```
from flask import request
@app.route('/login', methods=['GET'])
    def login():
        username = request.args.get('username')
        print(username)
        password= request.args.get('password')
        print(password)
```

PHOTO TIMELINE

MAIN.PY

```
@app.route('/form-example', methods=['GET', 'POST']) #allow both GET and POST requests
def form_example():
    if request.method == 'POST':  #this block is only entered when the form is submitted
        language = request.form.get('language')
        framework = request.form['framework']

        return '''<h1>The language value is: {}</h1>
                  <h1>The framework value is: {}<8080/h1>'''.format(language, framework)

    return '''<form method="POST">
                  Language: <input type="text" name="language"><br>
                  Framework: <input type="text" name="framework"><br>
                  <input type="submit" value="Submit"><br>
              </form>'''
```

Use method as condition

PHOTO TIMELINE

MAIN.PY

```
@app.route('/user/<username>')
def profile(username):
    ...

@app.route('/<int:year>/<int:month>/<title>')
def article(year, month, title):
    ...
```

PHOTO TIMELINE

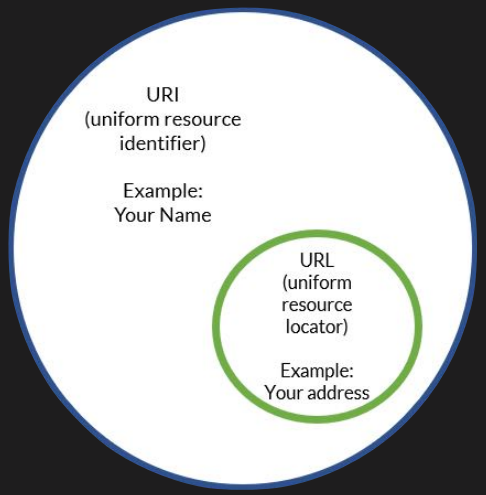
The URI (uniform resource identifiers) is a matter of style

```
/users/<user>/json/?user=<user>&style=json
```

PHOTO TIMELINE

The URI (uniform resource identifiers) is a matter of style

```
/users/<user>/json/?user=<user>&style=json
```

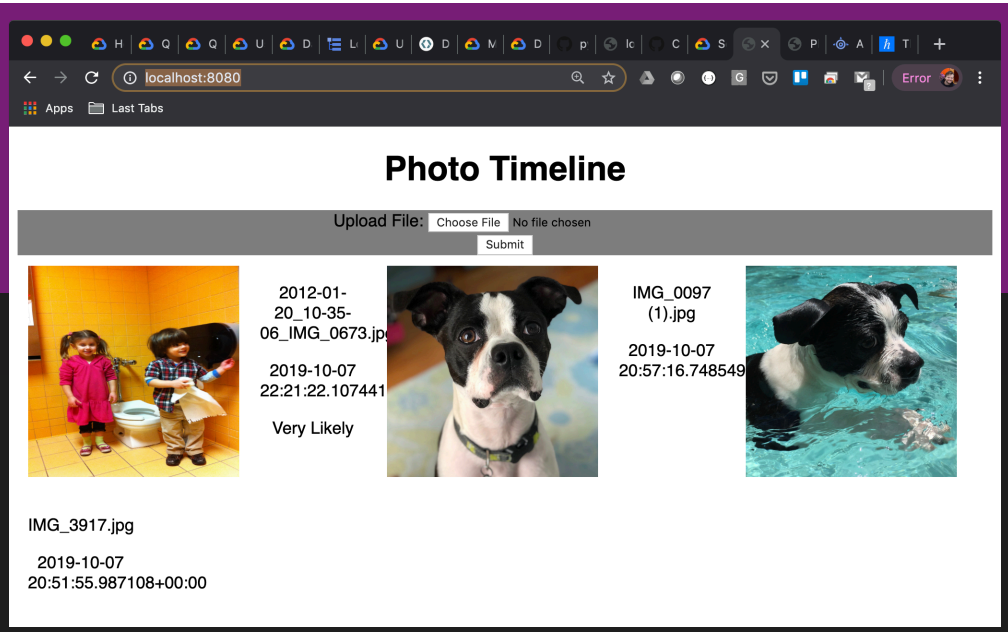


FOR STARTERS, URI STANDS FOR UNIFORM RESOURCE IDENTIFIER AND URL STANDS FOR UNIFORM RESOURCE LOCATOR. MOST OF THE CONFUSION WITH THESE TWO IS BECAUSE THEY ARE RELATED. YOU SEE, A URI CAN BE A NAME, LOCATOR, OR BOTH FOR AN ONLINE RESOURCE WHERE A URL IS JUST THE LOCATOR. URLS ARE A SUBSET OF URIS.

MAIN.PY ROUTES

PHOTO TIMELINE

MAIN.PY




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

    # Use the Cloud Datastore client to fetch information from Datastore about each
    query = datastore_client.query(kind='Faces')
    image_entities = list(query.fetch())

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

PHOTO TIMELINE

MAIN.PY

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

    # Use the Cloud Datastore client to fetch information from Datastore about each
    query = datastore_client.query(kind='Faces')
    image_entities = list(query.fetch())

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

Query datastore

Render template with image_entities

PHOTO TIMELINE

MAIN.PY

```python
@app.route('/upload_photo', methods=['GET', 'POST'])
def upload_photo():
    photo = request.files['file']

    # Create a Cloud Storage client.
    storage_client = storage.Client()

    # Get the bucket that the file will be uploaded to.
    bucket = storage_client.get_bucket(CLOUD_STORAGE_BUCKET)

    # Create a new blob and upload the file's content.
    blob = bucket.blob(photo.filename)
    blob.upload_from_string(
            photo.read(), content_type=photo.content_type)

    # Make the blob publicly viewable.
    blob.make_public()
```

Upload the file to storage

Set permissions

PHOTO TIMELINE

MAIN.PY

```
# Create a Cloud Datastore client.
datastore_client = datastore.Client()

# Fetch the current date / time.
current_datetime = datetime.datetime.now()

# The kind for the new entity.
kind = 'Faces'

# The name/ID for the new entity.
name = blob.name

# Create the Cloud Datastore key for the new entity.
key = datastore_client.key(kind, name)
```

Connect to datastore

Create a key based on name

PHOTO TIMELINE

MAIN.PY

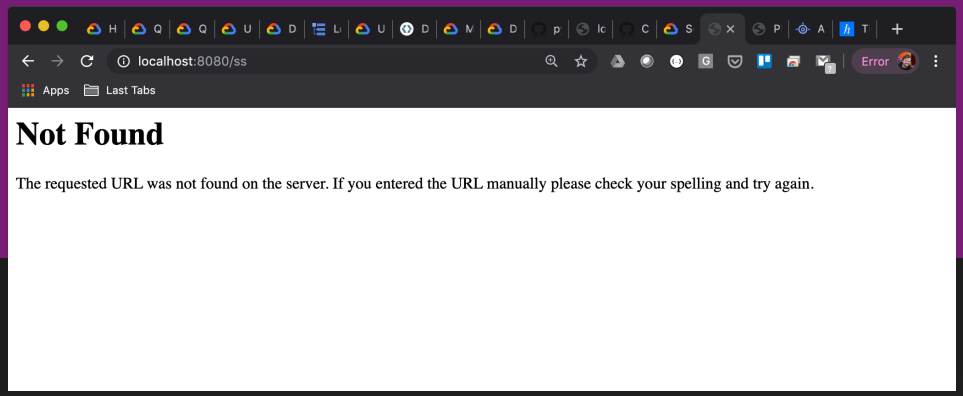
```
# Construct the new entity using the key. Set dictionary values for entity
# keys blob_name, storage_public_url, timestamp
entity = datastore.Entity(key)
entity['blob_name'] = blob.name
entity['image_public_url'] = blob.public_url
entity['timestamp'] = current_datetime

# Save the new entity to Datastore.
datastore_client.put(entity)

# Redirect to the home page.
return redirect('/')
```

Set entity values

Redirect to homepage

PHOTO TIMELINE

MAIN.PY

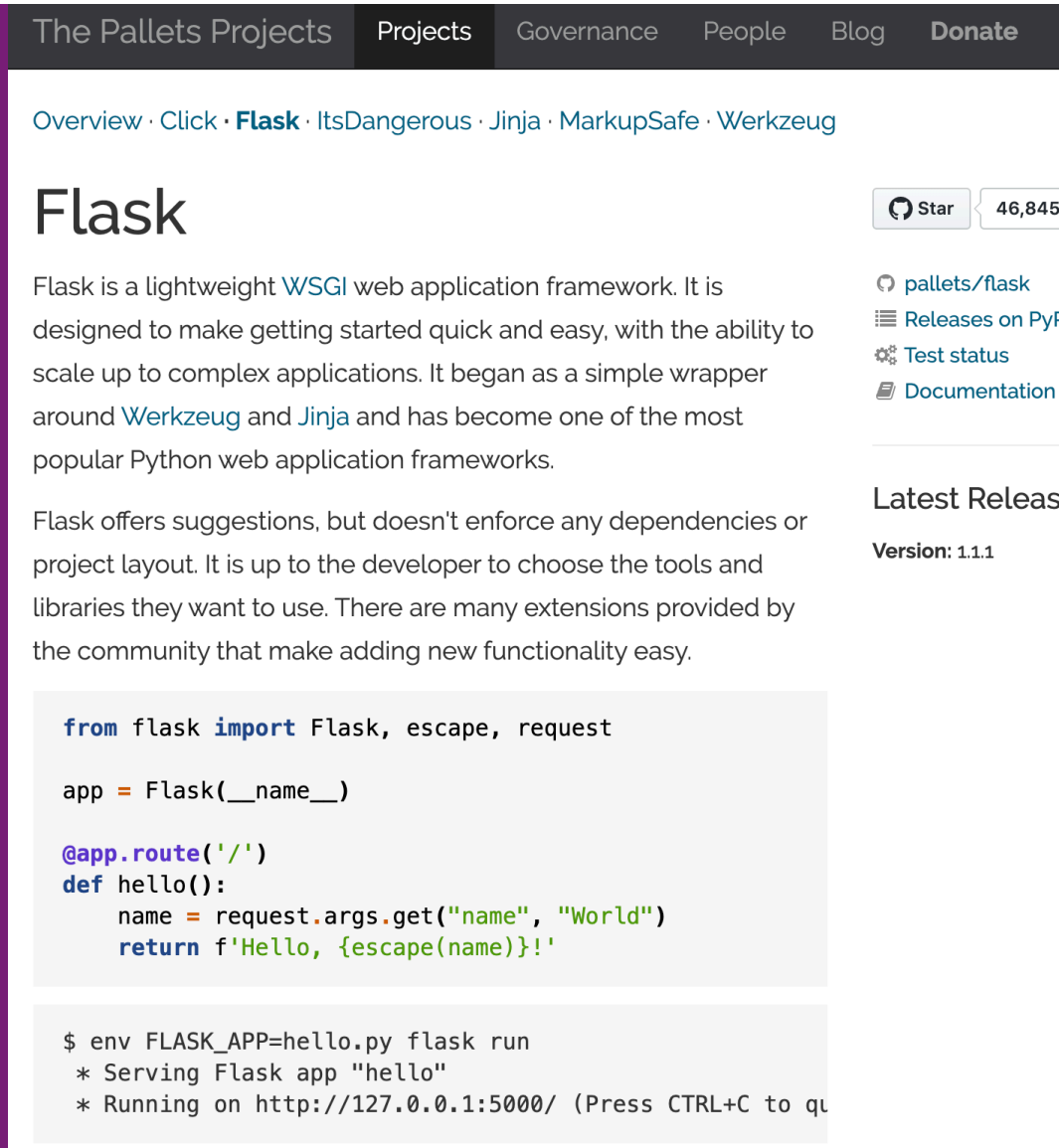



```
@app.errorhandler(500)
def server_error(e):
    logging.exception('An error occurred during a request.')
    return """
    An internal error occurred: <pre>{}</pre>
    See logs for full stacktrace.
    """.format(e), 500
```

PHOTO TIMELINE

MAIN.PY

- Python Flask web application
  - Werkzeug and Jinja roots
- Availables on App Engine Flex and now in standard environment for Python3



REQUIREMENTS.TXT

PHOTO TIMELINE

REQUIREMENTS.TXT

- Install the dependencies required for your application

```
Flask==1.0.2
google-cloud-datastore==1.7.3
google-cloud-storage==1.13.2
```

PHOTO TIMELINE

REQUIREMENTS.TXT

- Create a virtual environment for your application
- Helps to keep the correct versions and dependencies for your specific applicaiton

```
virtualenv -p python3 env
source env/bin/activate
```

PHOTO TIMELINE

REQUIREMENTS.TXT

- Install the required dependencies

```
virtualenv -p python3 env
source env/bin/activate
pip install -r requirements.txt
```

ENABLE APIS

PHOTO TIMELINE

ENABLE APIS

- Enable APIs you are going to use for the project
- Some are on by default

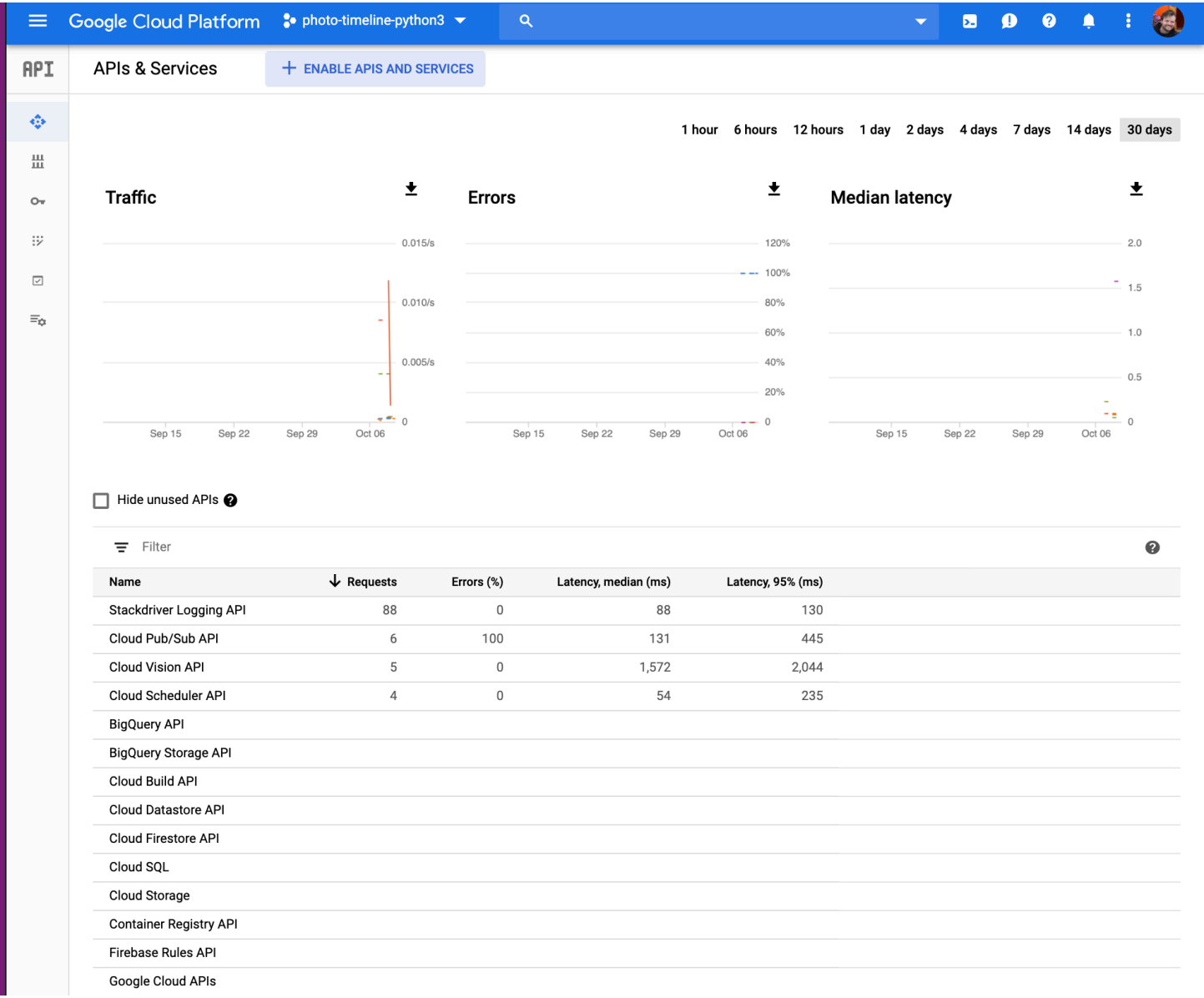


PHOTO TIMELINE

ENABLE APIS

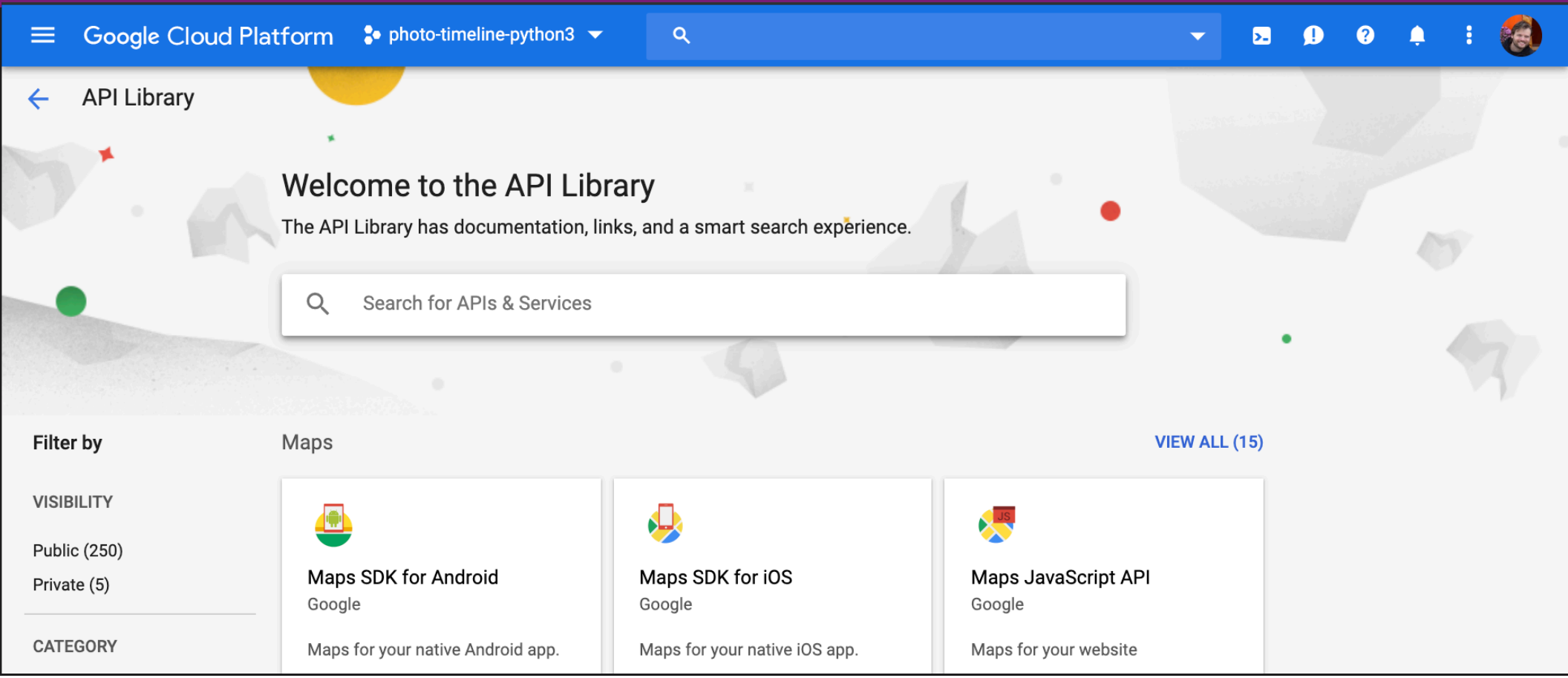


PHOTO TIMELINE

ENABLE APIS

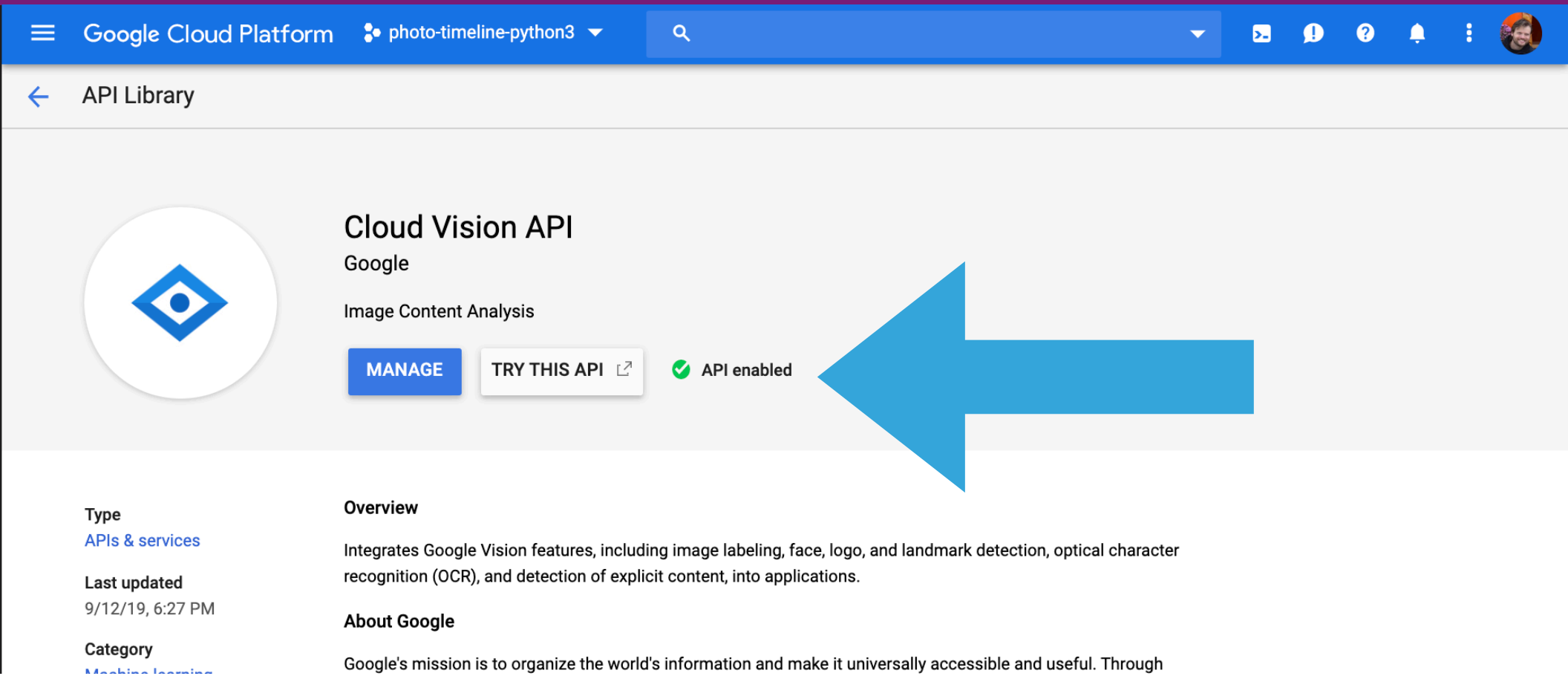


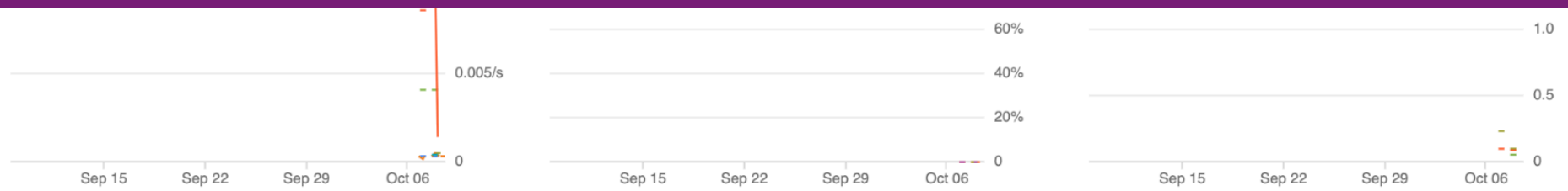
PHOTO TIMELINE

ENABLE APIS

```
gcloud services enable storage-component.googleapis.com

gcloud services enable datastore.googleapis.com

gcloud services enable vision.googleapis.com
```

PHOTO TIMELINE

ENABLE APIS



Hide unused APIs

Filter

Name	Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Stackdriver Logging API	88	0	88	130
Cloud Pub/Sub API	6	100	131	445
Cloud Vision API	5	0	1,572	2,044
Cloud Scheduler API	4	0	54	235

(photo-timeline-python3)

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to photo-timeline-python3.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
abinkowski@cloudshell:~ (photo-timeline-python3)$ gcloud services enable vision.googleapis.com
abinkowski@cloudshell:~ (photo-timeline-python3)$
```

SERVICES ACCOUNT

PHOTO TIMELINE

SERVICES ACCOUNT

- Create a services account
- Needed to run some APIs
- Owner not always the same as the developer

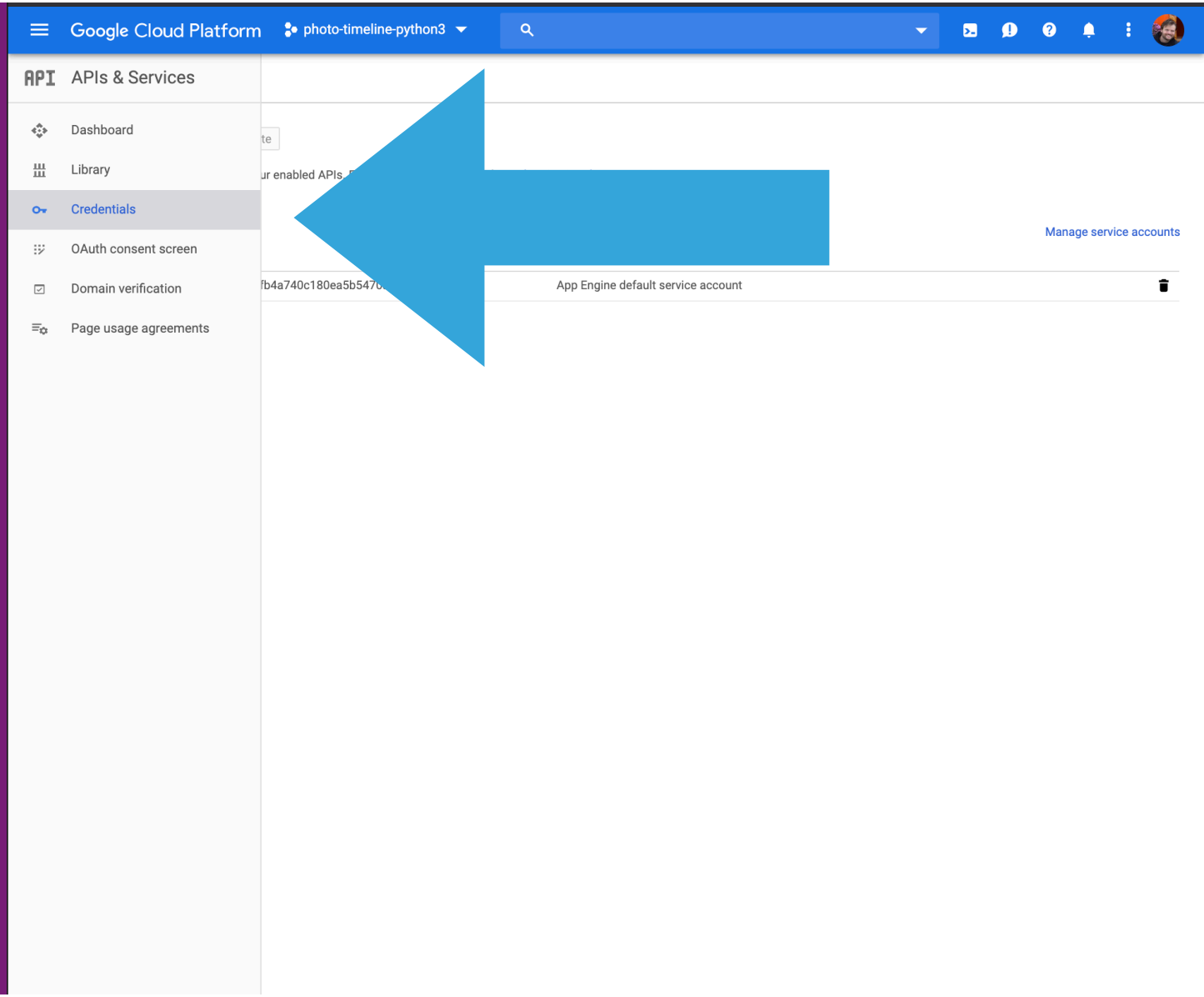
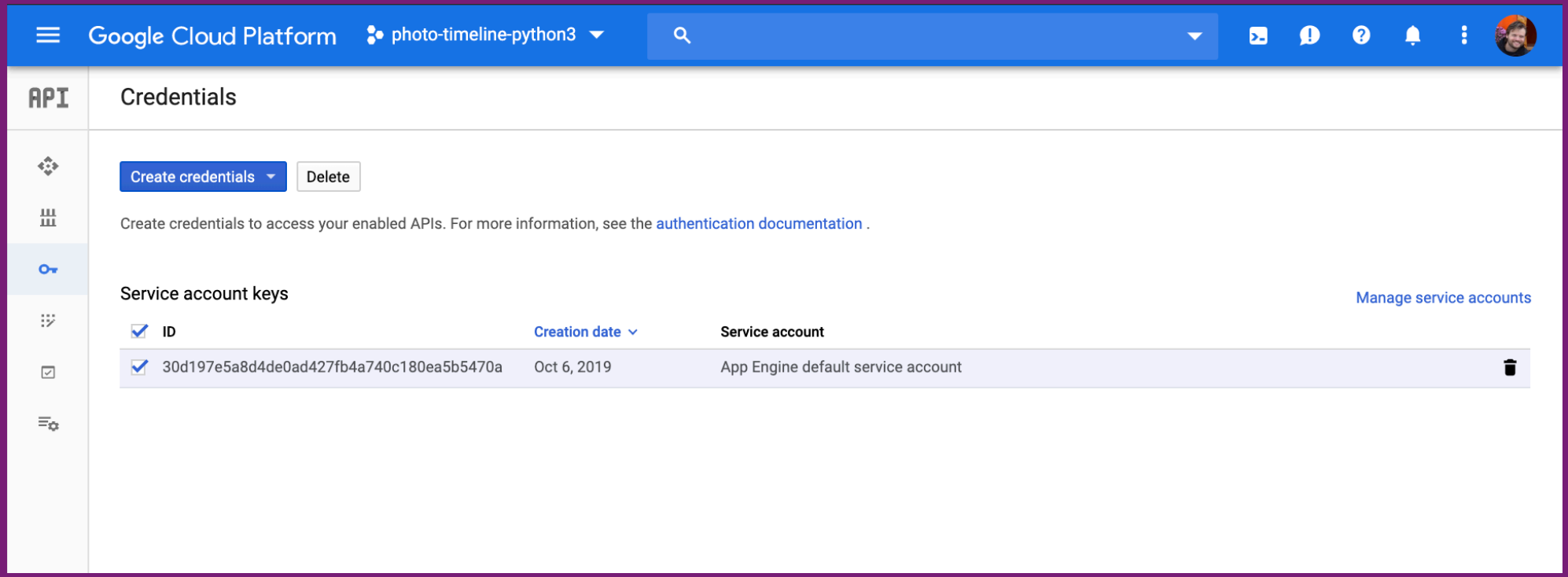


PHOTO TIMELINE

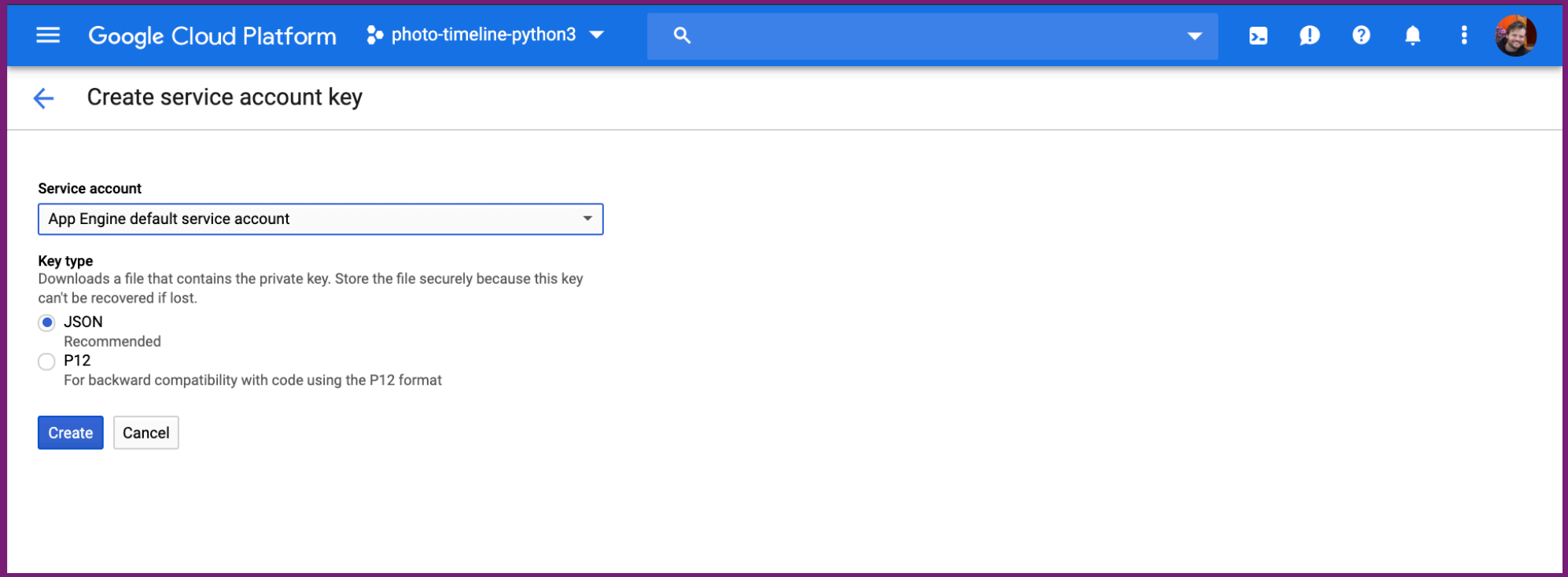
SERVICES ACCOUNT



- Should have a default service account created

PHOTO TIMELINE

SERVICES ACCOUNT



- Create a Service Account key (a .json file)

PHOTO TIMELINE

SERVICES ACCOUNT

- Service account key file

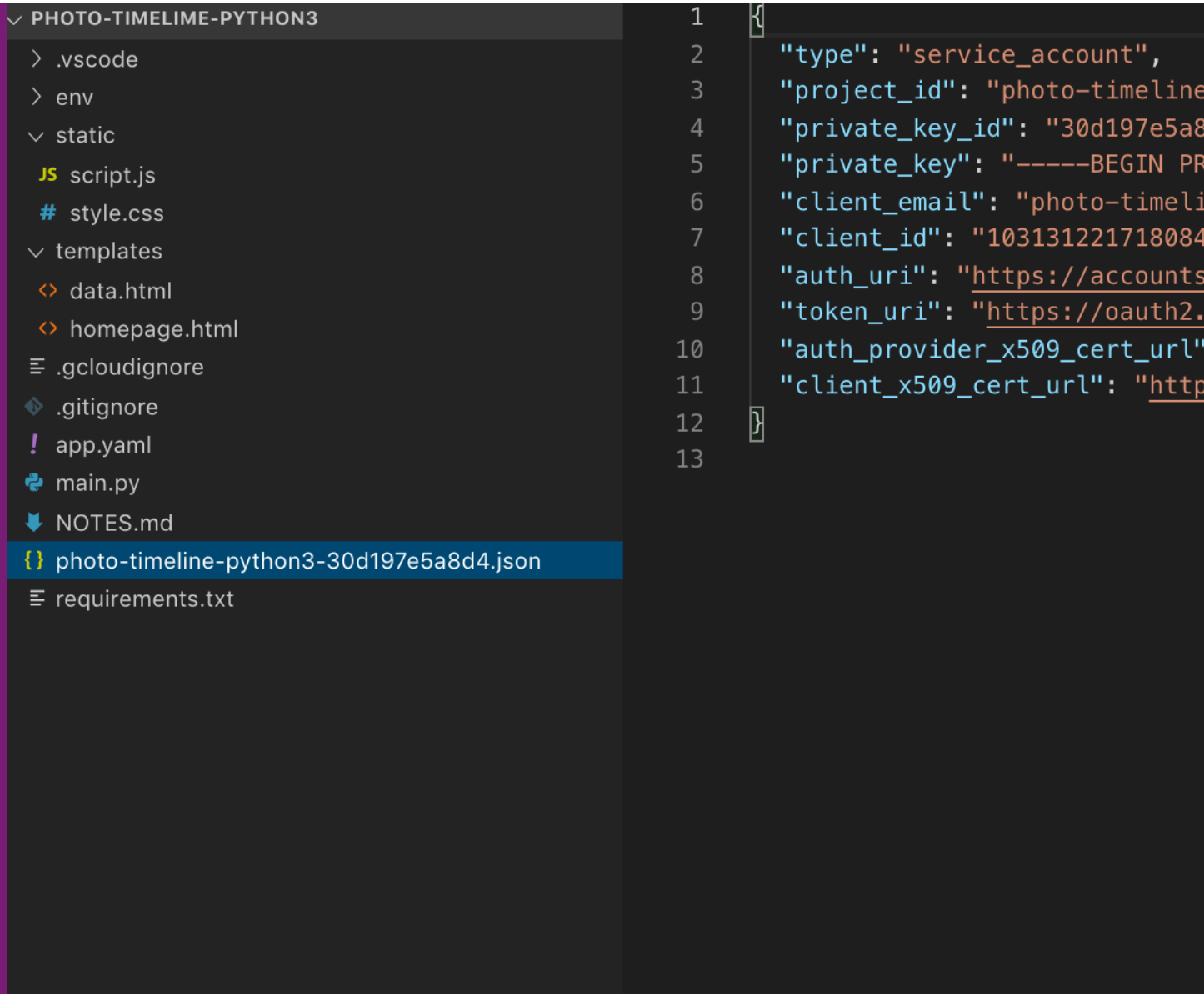


PHOTO TIMELINE

SERVICE ACCOUNT

Application will need this to run some APIs

```
export GOOGLE_APPLICATION_CREDENTIALS=[PATH]/key.json"
```

RUNNING YOUR APPLICATION

PHOTO TIMELINE

RUN LOCALLY

- Install the required dependencies

```
virtualenv -p python3 env
source env/bin/activate
pip install -r requirements.txt
```

PHOTO TIMELINE

RUN LOCALLY

Create project

```
# Create the project
gcloud projects create photo-timeline-python3
gcloud projects describe photo-timeline-python3

# Create App Engine service
gcloud app create --project=photo-timeline-python3
```

Need to create an App Engine service

PHOTO TIMELINE

RUN LOCALLY

```
# Change the default project
gcloud config set project photo-timeline-python3

# Set a environment variable of the project; this is
# done automatically on your GCP project
export PROJECT_ID=photo-timeline-python3
```

PHOTO TIMELINE

RUN LOCALLY

Storage bucket name has to match

```
# Set a environment variable of the project; this is
# done automatically on your GCP project
export PROJECT_ID=photo-timeline-python3

# Create the storage bucket
gsutil mb gs://${PROJECT_ID}

export CLOUD_STORAGE_BUCKET=${PROJECT_ID}
```

PHOTO TIMELINE

RUN LOCALLY

```
# Run the application on http://localhost:8080
python main.py
```

PHOTO TIMELINE

RUN LOCALLY

```
507 % python main.py
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 186-675-326
```

PHOTO TIMELINE

RUN LOCALLY

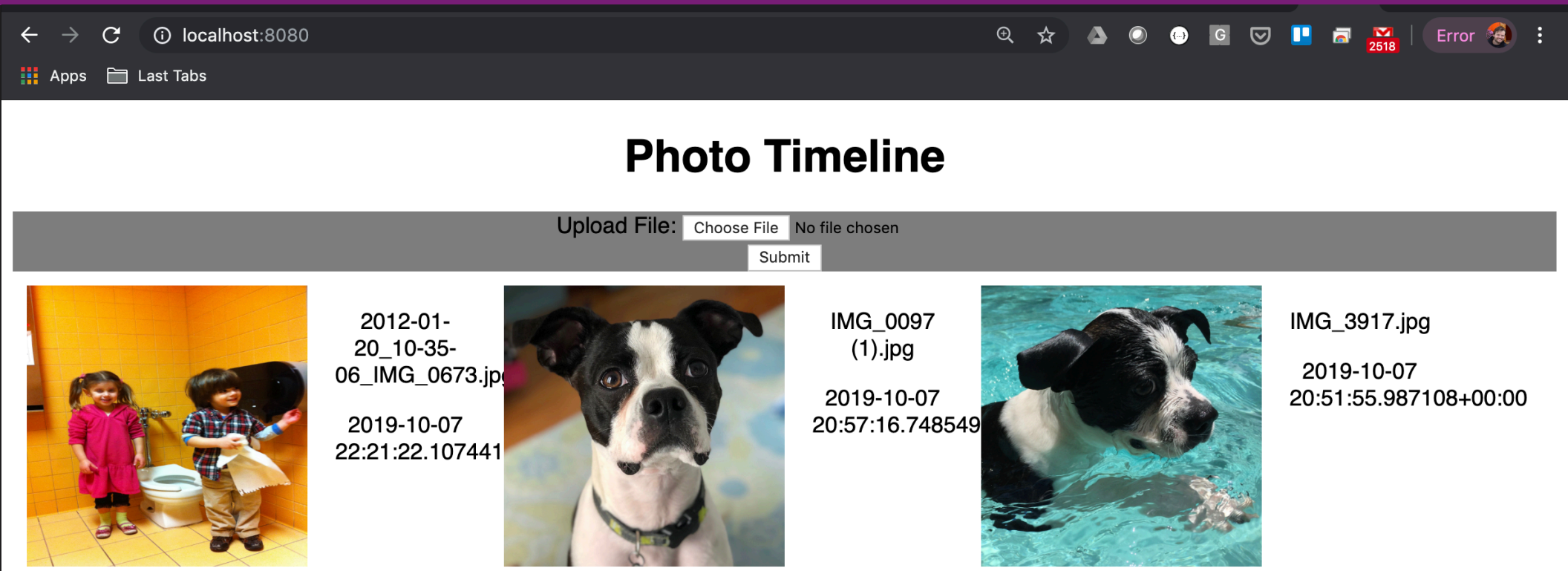


PHOTO TIMELINE

RUN LOCALLY

```
# Run the application on http://localhost:8080
python main.py
```

Using actual API services so local data and production should match...not always good

DEVELOPMENT AND LOGGING

DEVELOPMENT AND LOGGING

- Different levels of logging
- Appear different in the console
- Used to filter logs

```
import logging


logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')

try:
    raise ValueError('This is a sample value error.')
except ValueError:
    logging.exception('A example exception log.')
```

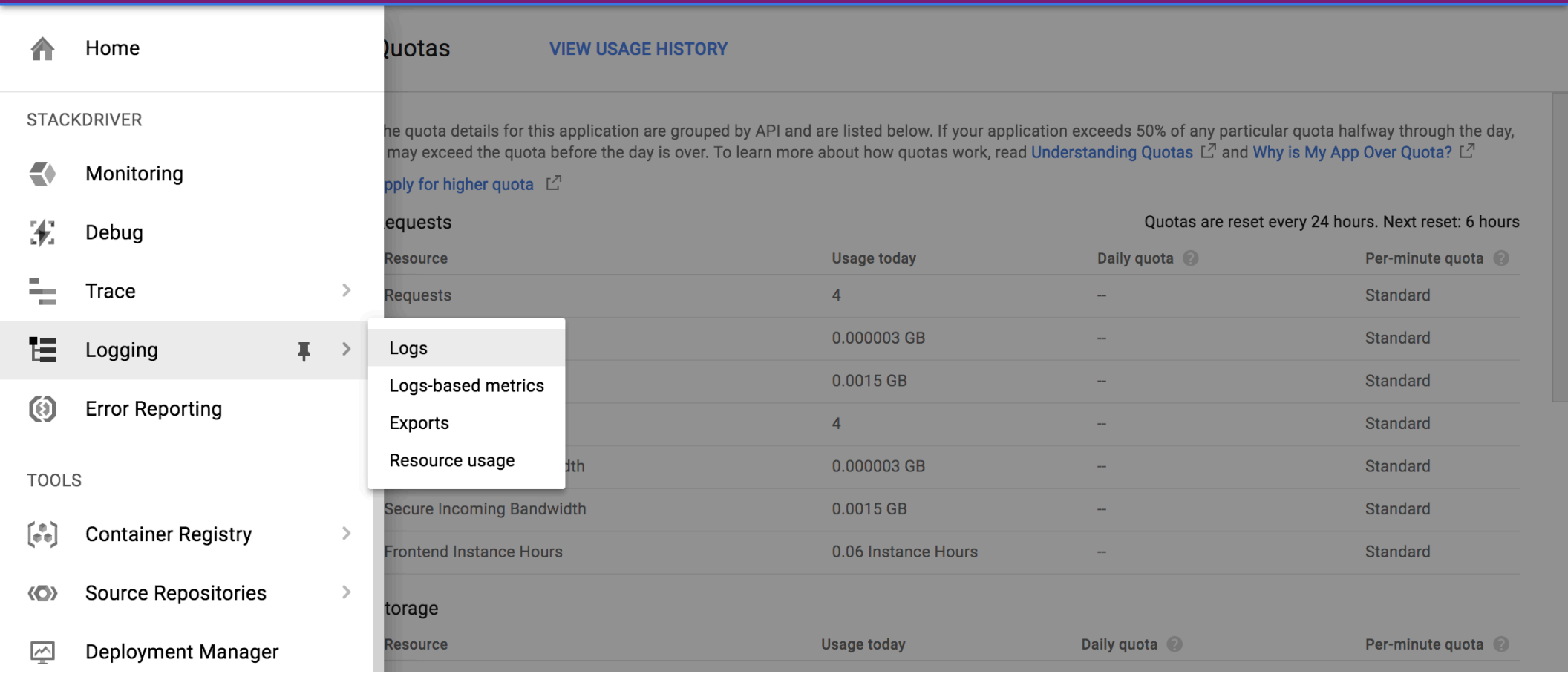
DEVELOPMENT AND LOGGING

```
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')

try:
    raise ValueError('This is a sample value erro
except ValueError:
    logging.exception('A example exception log.')
```

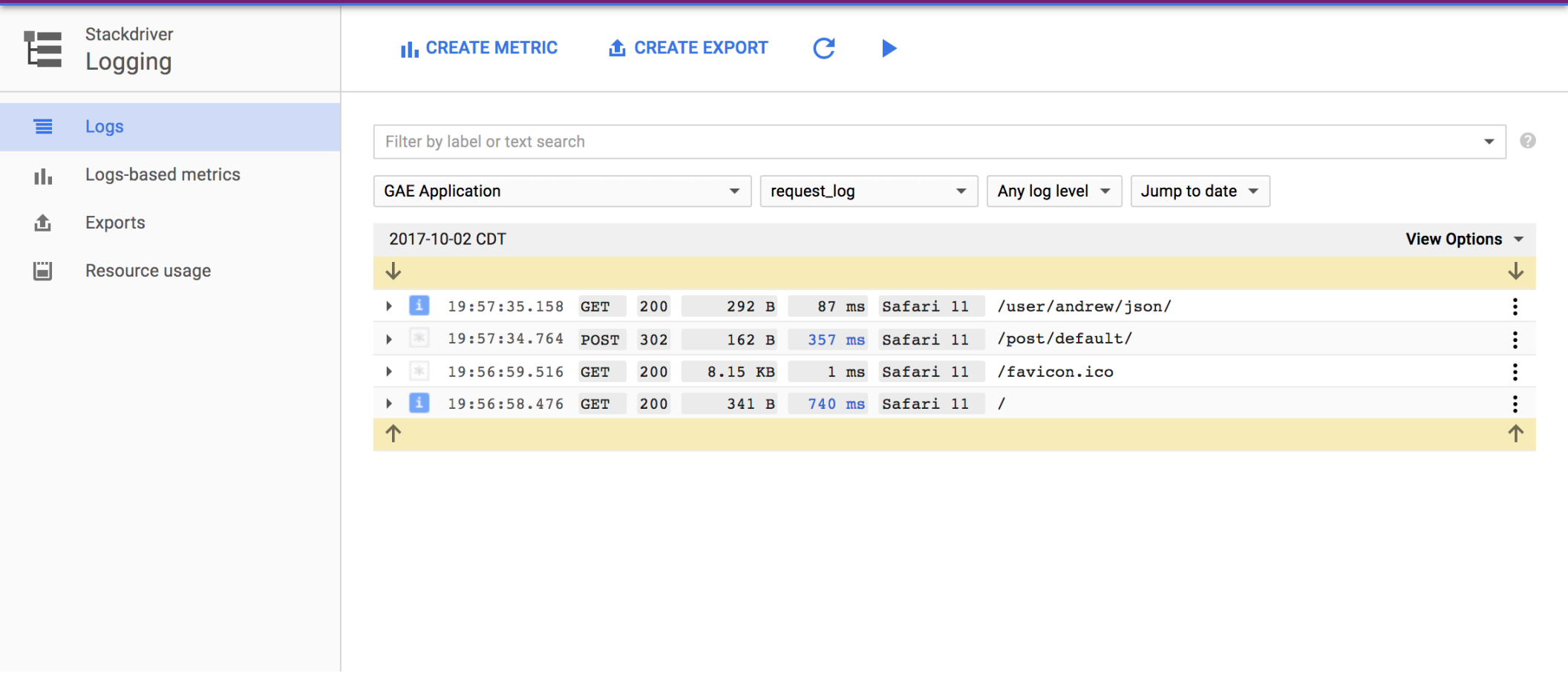
session2-cloud-photo-timeline — python

```
TTP/1.1" 200 188
INFO     2017-10-03 00:20:33,178 module.py:4
  /Users/tabinkowski/Google Drive/g-Teaching
mpcs51033-2017-autumn-playground-private/ses
INFO     2017-10-03 00:39:24,441 main.py:153] This is an info message
WARNING  2017-10-03 00:39:24,442 main.py:154] This is a warning message
ERROR    2017-10-03 00:39:24,442 main.py:155] This is an error message
CRITICAL 2017-10-03 00:39:24,442 main.py:156] This is a critical message
ERROR    2017-10-03 00:39:24,442 main.py:161] A example exception log.
Traceback (most recent call last):
  File "/Users/tabinkowski/Google Drive/g-Teaching/uchicago.cloud/mpcs51033-2017-a
utumn/mpcs51033-2017-autumn-playground-private/session2-cloud-photo-timeline/main.
py", line 159, in get
    raise ValueError('This is a sample value error.')
ValueError: This is a sample value error.
INFO     2017-10-03 00:39:24,474 module.py:821] default: "GET /logging/ HTTP/1.1"
200 16
```

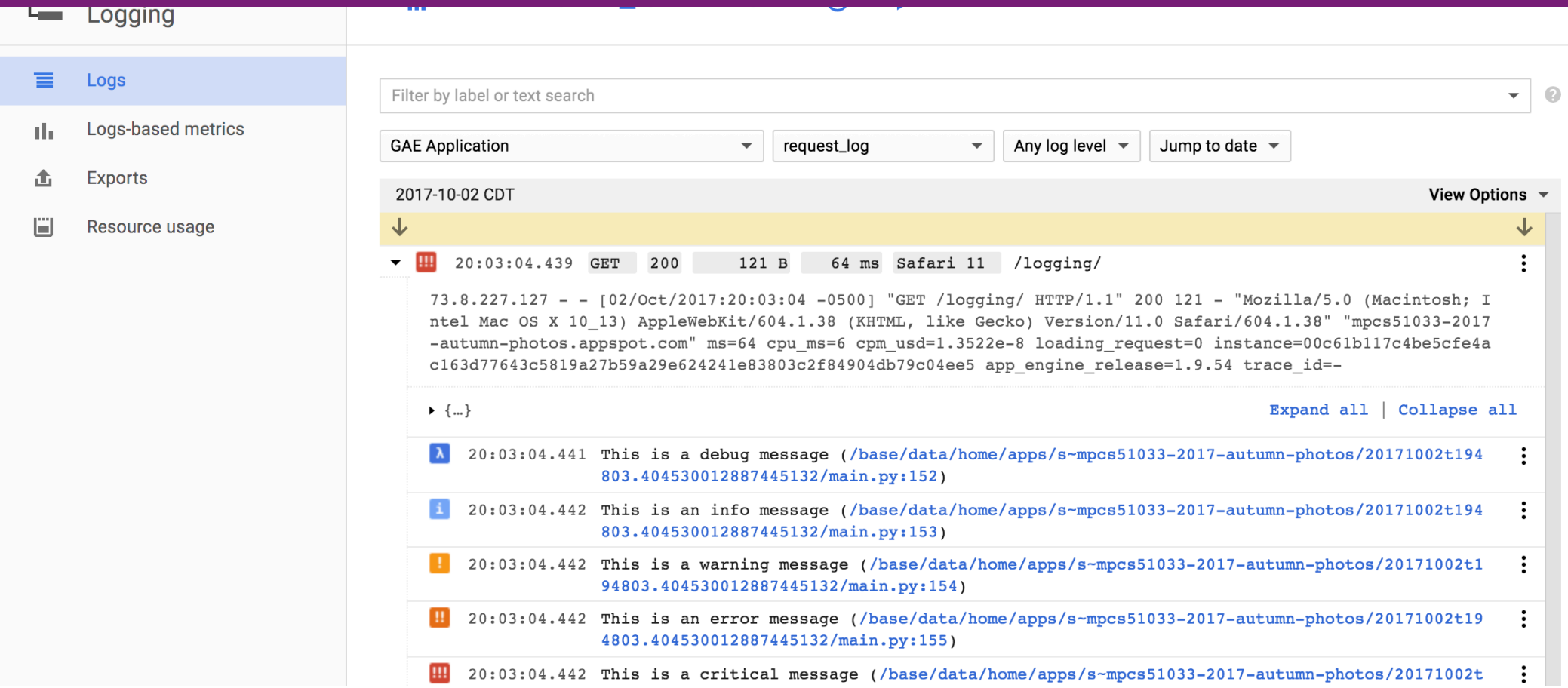
DEVELOPMENT AND LOGGING



DEVELOPMENT AND LOGGING



DEVELOPMENT AND LOGGING



DEVELOPMENT AND LOGGING

```
# Download the logs from a running application

% gcloud app logs read -s default
```

DEPLOY TO APP ENGINE

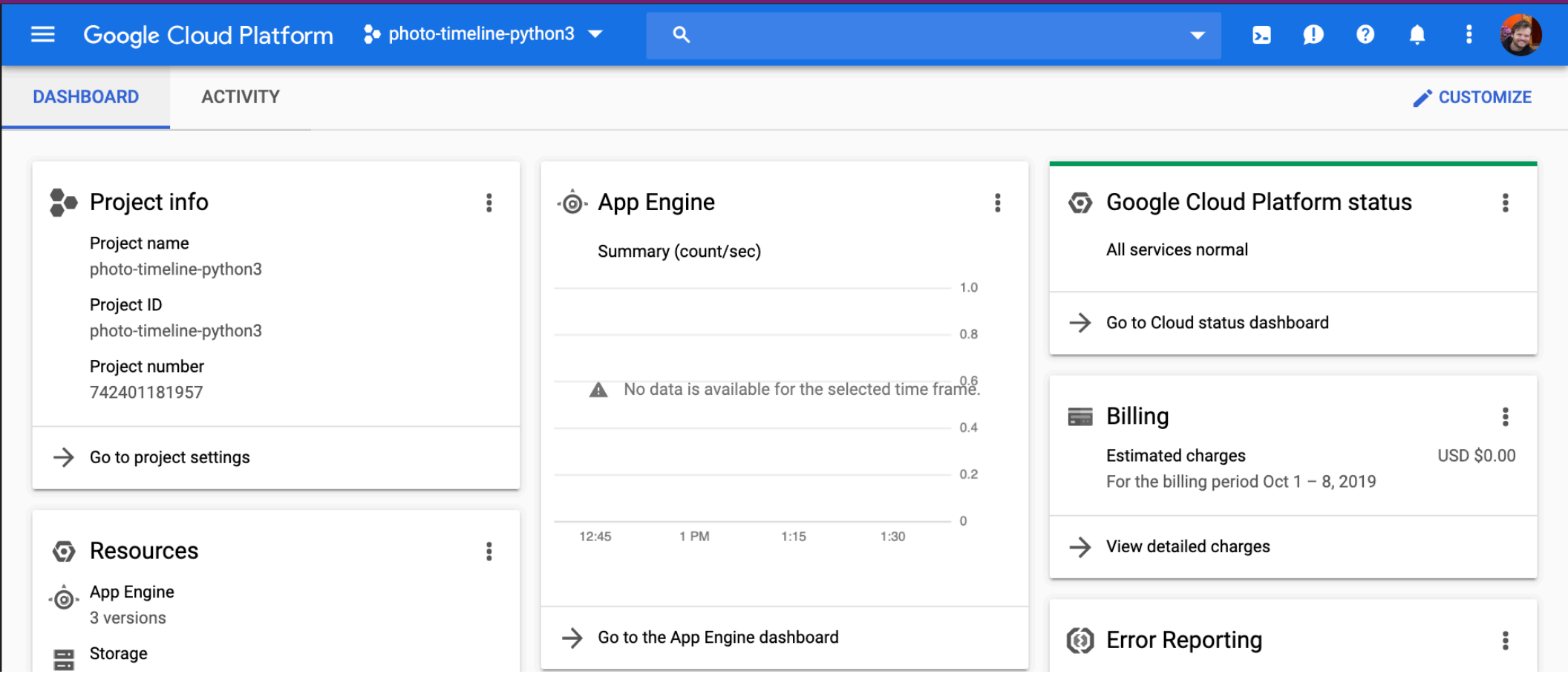
DEPLOY TO APP ENGINE

```
# Deploy
gcloud app deploy
```

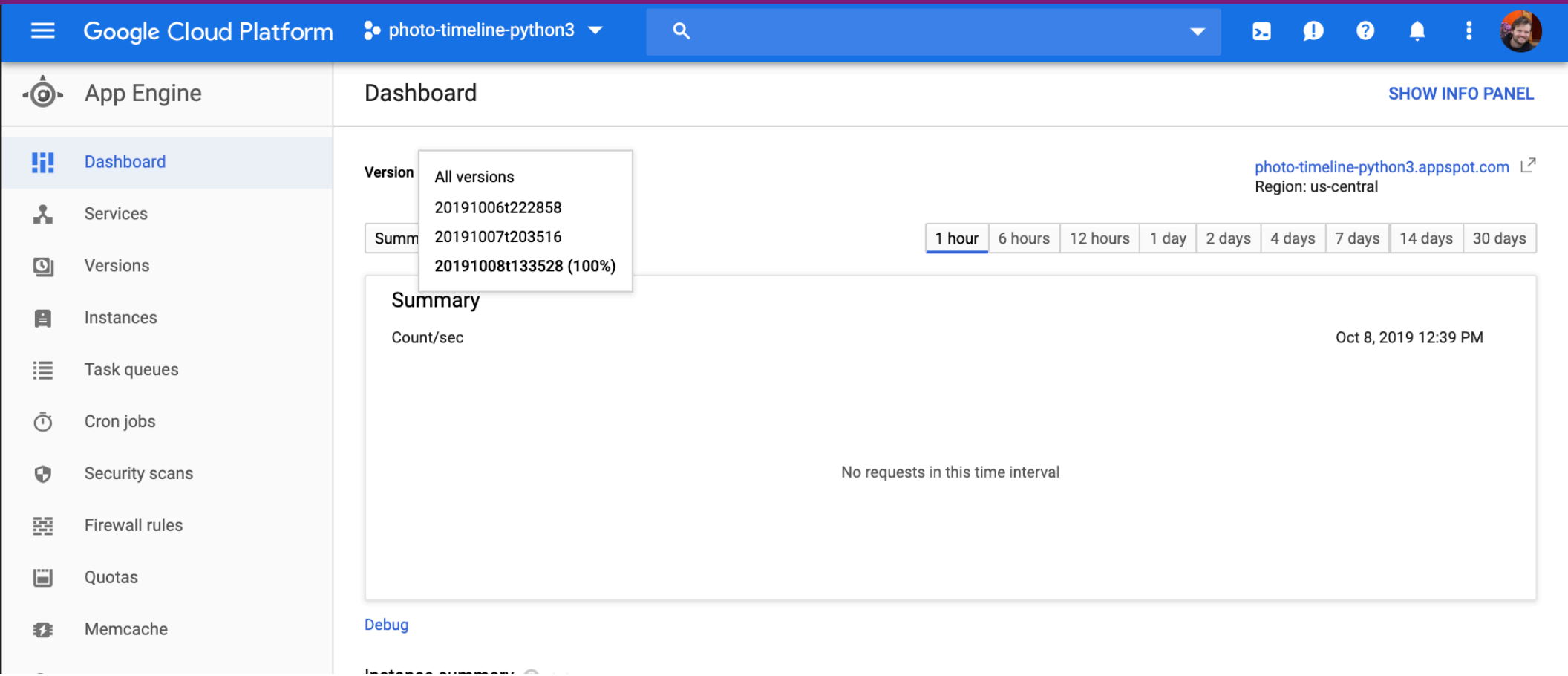
DEPLOY TO APP ENGINE

```
[508 % gcloud app deploy
Services to deploy:

descriptor:      [/Users/tabinkowski/Google Drive/g-Teaching/uchicago.codes/uchicago.codes-courses/mpcs50
101/mpcs50101-2019-autumn/mpcs50101-2019-autumn-code/photo-timelime-python3/app.yaml]
source:          [/Users/tabinkowski/Google Drive/g-Teaching/uchicago.codes/uchicago.codes-courses/mpcs50
101/mpcs50101-2019-autumn/mpcs50101-2019-autumn-code/photo-timelime-python3]
target project:  [photo-timeline-python3]
target service:  [default]
target version:  [20191008t133528]
target url:      [https://photo-timeline-python3.appspot.com]


Do you want to continue (Y/n)?  Y

Beginning deployment of service [default]...
╔════════════════════════════════════════════════════════════╗
╠═ Uploading 66 files to Google Cloud Storage               ═╣
╚════════════════════════════════════════════════════════════╝
File upload done.
Updating service [default]...done.
Setting traffic split for service [default]...done.
Deployed service [default] to [https://photo-timeline-python3.appspot.com]
```

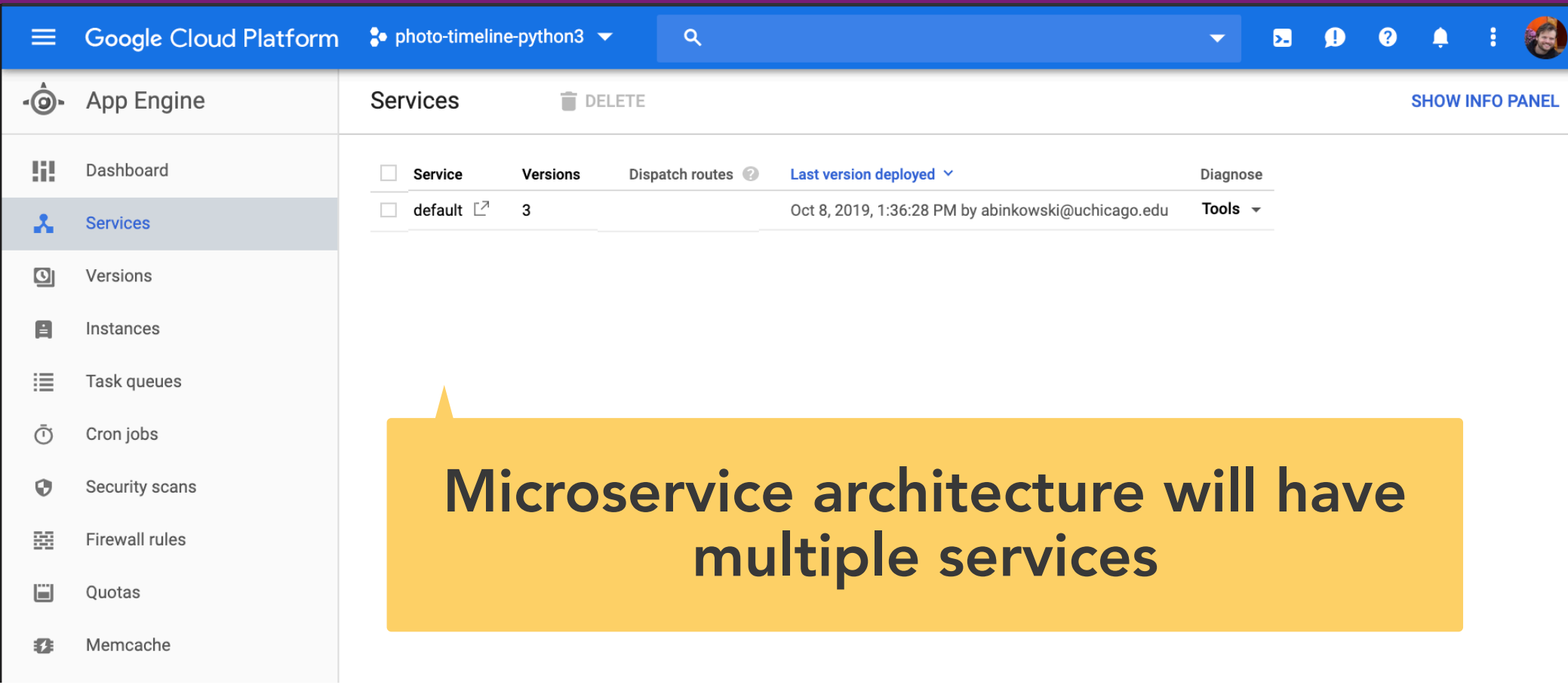
DEPLOY TO APP ENGINE



DEPLOY TO APP ENGINE

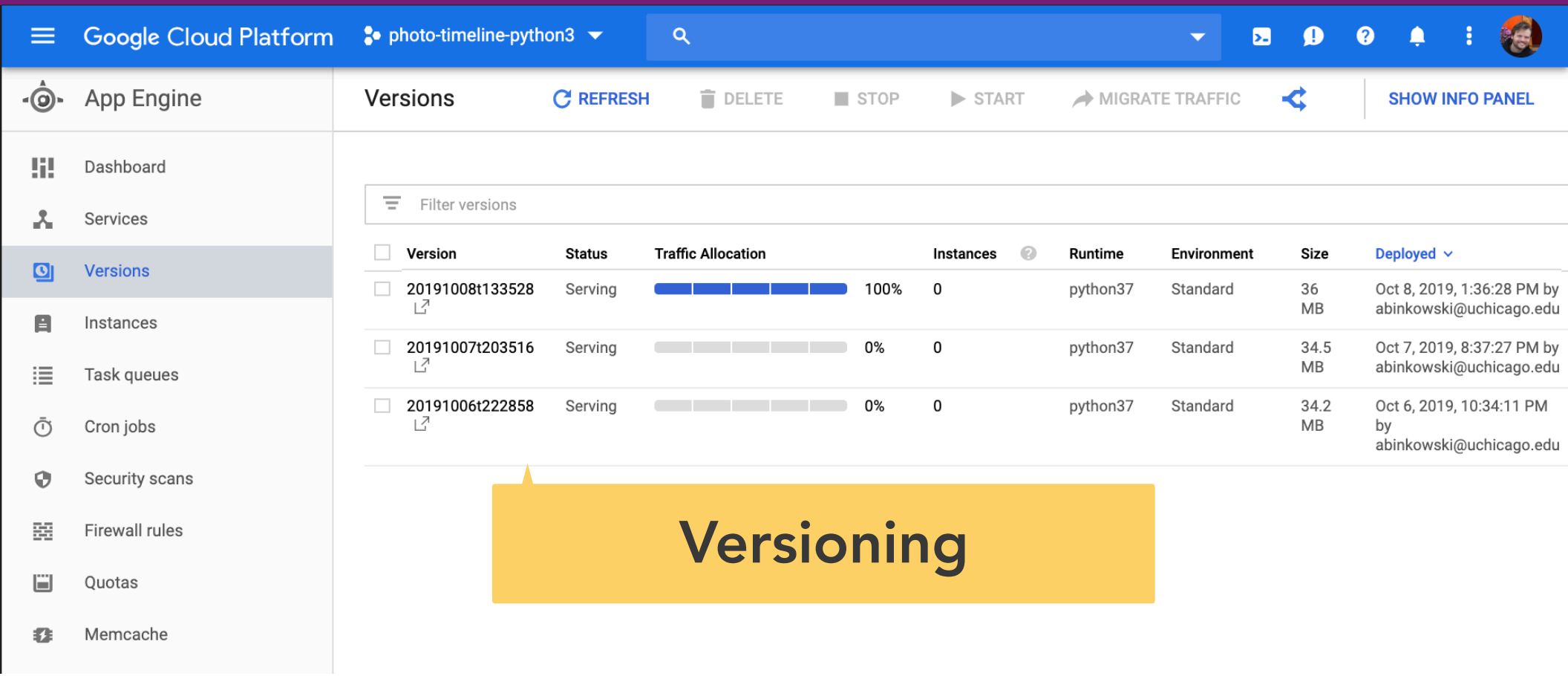


DEPLOY TO APP ENGINE

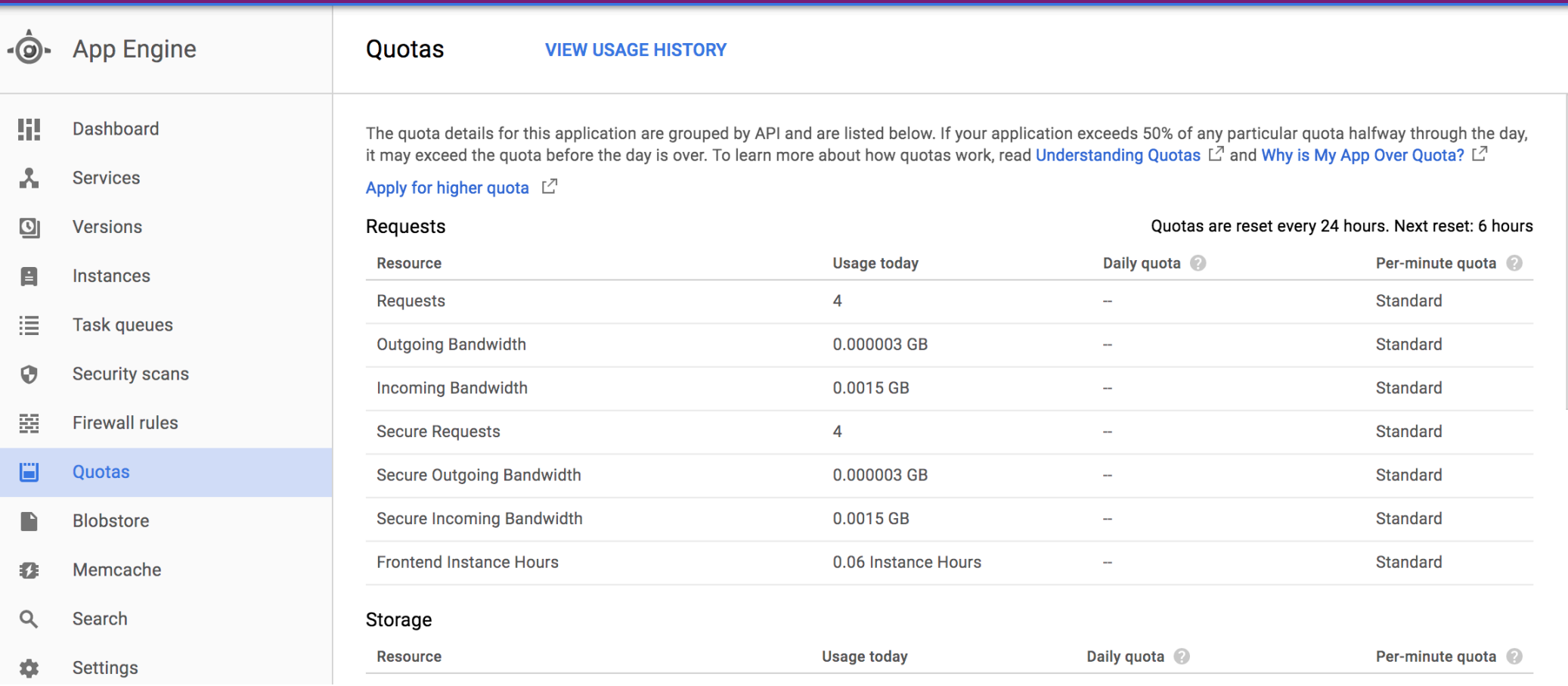


Microservice architecture will have multiple services

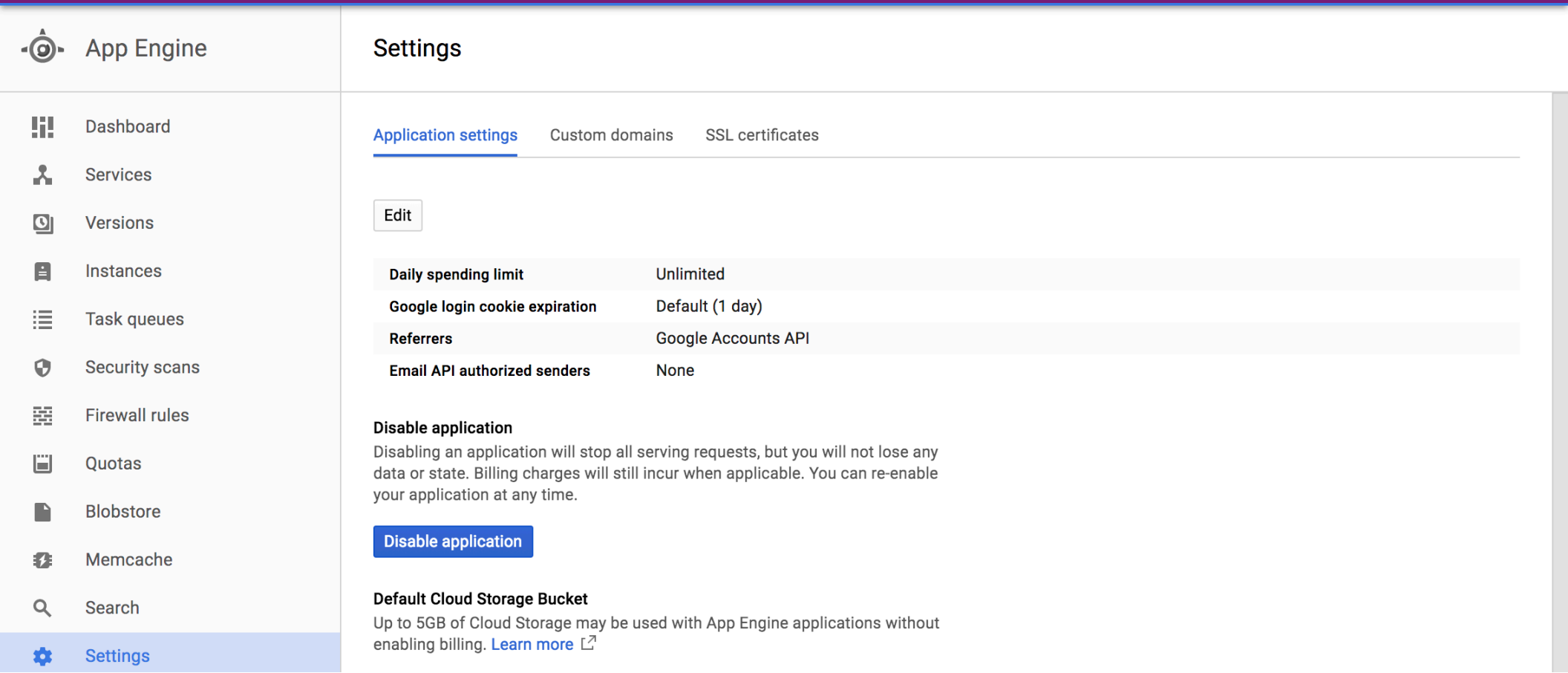
DEPLOY TO APP ENGINE



DEPLOY TO APP ENGINE



DEPLOY TO APP ENGINE



CLOUD FIRESTORE (DATASTORE)

CLOUD FIRESTORE

- NoSQL document database
- Highly structured mutable data
- Designed and implemented for scaling

Cloud Datastore Overview

SEND FEEDBACK

Python | Java | PHP | Go

Google Cloud Datastore is a NoSQL document database built for automatic scaling, high performance, and ease of application development. Cloud Datastore features include:

- **Atomic transactions**. Cloud Datastore can execute a set of operations where either all succeed, or none occur.
- **High availability of reads and writes**. Cloud Datastore runs in Google data centers, which use redundancy to minimize impact from points of failure.
- **Massive scalability with high performance**. Cloud Datastore uses a distributed architecture to automatically manage scaling. Cloud Datastore uses a mix of indexes and query constraints so your queries scale with the size of your result set, not the size of your data set.
- **Flexible storage and querying of data**. Cloud Datastore maps naturally to object-oriented and scripting languages, and is exposed to applications through multiple clients. It also provides a SQL-like query language.
- **Balance of strong and eventual consistency**. Cloud Datastore ensures that entity lookups by key and ancestor queries always receive strongly consistent data. All other queries are eventually consistent. The consistency models allow your application to deliver a great user experience while handling large amounts of data and users.
- **Encryption at rest**. Cloud Datastore automatically encrypts all data before it is written to disk and automatically decrypts the data when read by an authorized user. For more information, see Server-Side Encryption.
- **Fully managed with no planned downtime**. Google handles the administration of the Cloud Datastore service so you can focus on your application. Your application can still use Cloud Datastore when the service receives a planned upgrade.

Comparison with traditional databases

While the Cloud Datastore interface has many of the same features as traditional databases, as a NoSQL database it differs from them in the way it describes relationships between data objects. Here's a high-level comparison of Cloud Datastore and relational database concepts:

Concept	Cloud Datastore	Relational database
Category of object	Kind	Table
One object	Entity	Row
Individual data for an object	Property	Field
Unique ID for an object	Key	Primary key

CLOUD FIRESTORE

- Could Firestore has "replaced" datastore
  - Datastore mode for original functionality

Choosing between Native Mode and Datastore Mode

SEND FEEDBACK

When you create a new Cloud Firestore database, you can configure the database instance to run in *Datastore mode* which makes the database backwards-compatible with Cloud Datastore. This page helps you understand the difference between the two Cloud Firestore database modes: **Native mode** and **Datastore mode**.

Cloud Firestore in Native mode

Cloud Firestore is the next major version of Cloud Datastore and a re-branding of the product. Taking the best of Cloud Datastore and the Firebase Realtime Database, Cloud Firestore is a NoSQL document database built for automatic scaling, high performance, and ease of application development.

Cloud Firestore introduces new features such as:

- A new, strongly consistent storage layer
- A collection and document data model
- Real-time updates
- Mobile and Web client libraries

Cloud Firestore is backwards compatible with Cloud Datastore, but the new data model, real-time updates, and mobile and web client library features are not. To access all of the new Cloud Firestore features, you must use Cloud Firestore in Native mode.

CLOUD FIRESTORE

- Remember just one of many options for store data
- Available via API through all GPC services

Cloud Datastore is ideal for applications that rely on highly available structured data at scale. You can use Cloud Datastore to store and query all of the following types of data:

- Product catalogs that provide real-time inventory and product details for a retailer.
- User profiles that deliver a customized experience based on the user's past activities and preferences.
- Transactions based on ACID properties, for example, transferring funds from one bank account to another.

Other storage options

Cloud Datastore is not ideal for every use case. For example, Cloud Datastore is not a relational database, and it is not an effective storage solution for analytic data.

Here are some common scenarios where you should probably consider an alternative to Cloud Datastore:

- If you need a relational database with full SQL support for an online transaction processing (OLTP) system, consider Google Cloud SQL.
- If you don't require support for ACID transactions or if your data is not highly structured, consider Cloud Bigtable.
- If you need interactive querying in an online analytical processing (OLAP) system, consider Google BigQuery.
- If you need to store large immutable blobs, such as large images or movies, consider Google Cloud Storage.

For more information about other storage options, see the Choosing a Storage Option guide.

Connecting to Cloud Datastore with App Engine

App Engine's Python standard runtime connects to Cloud Datastore using the NDB Client Library. The NDB Client Library provides persistent storage in a schemaless object datastore. It supports automatic caching, sophisticated queries, and atomic transactions.

CLOUD FIRESTORE

- Entities have properties that have more or more values

```
entity = datastore.Entity(key)
entity['blob_name'] = blob.name
entity['image_public_url'] = blob.public_url
entity['timestamp'] = current_datetime
```

CLOUD FIRESTORE

- Data types available
- Some very specialized for applications

https://cloud.google.com/datastore/docs/concepts/entities#properties_and_value_types

- Protocol buffer
  - field name: `array_value`
  - type: an `ArrayValue` message that contains one or more `Value` messages
- Sort order: None
- Notes: Cannot contain another array value. The value instance must not set `meaning` or `exclude_from_indexes`.

Boolean

- JSON
  - field name: `booleanValue`
  - type: `true` or `false`
- Protocol buffer
  - field name: `boolean_value`
  - type: `bool`
- Sort order: `false` < `true`

Blob

- JSON
  - field name: `blobValue`
  - type: string. Must be base64-encoded.
- Protocol buffer
  - field name: `blob_value`
  - type: `bytes`
- Sort order: Byte order
- Notes: Up to 1,500 bytes if property is indexed, up to 1,048,487 bytes (1 MiB - 89 bytes) otherwise.

Date and time

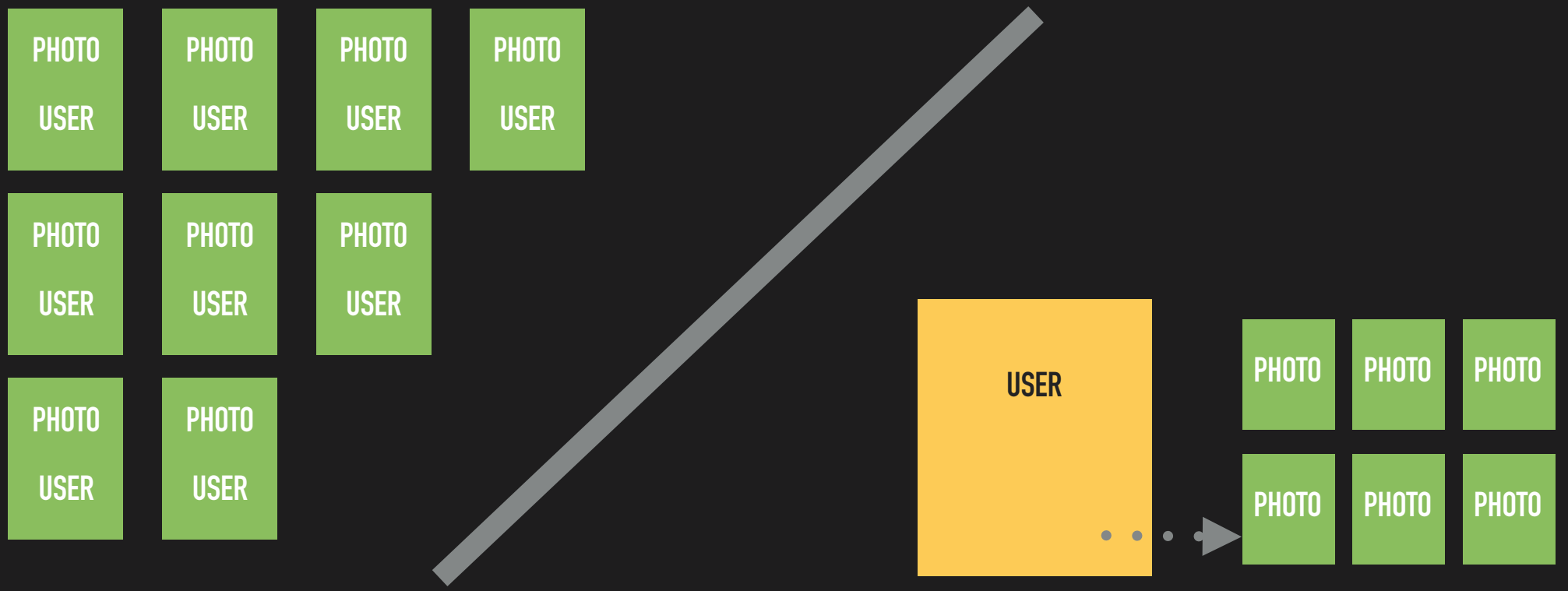
CLOUD FIRESTORE

- Model in our photo timeline app
- Objects with same ancestor have different behaviors

```
entity = datastore.Entity(key)
entity['blob_name'] = blob.name
entity['image_public_url'] = blob.public_url
entity['timestamp'] = current_datetime
```

CLOUD FIRESTORE

PHOTO TIMELINE DATA MODEL



POTENTIAL PROBLEMS WITH THE CURRENT ARCHITECTURE

PHOTO TIMELINE

FORESEEABLE PROBLEMS

- This solution will not scale very well once we have more users
- No way to identify who is uploading images
- Even if we added a 'user' field, every entity would have to be searched to retrieve the photos for a given user name
  - This could be very costly

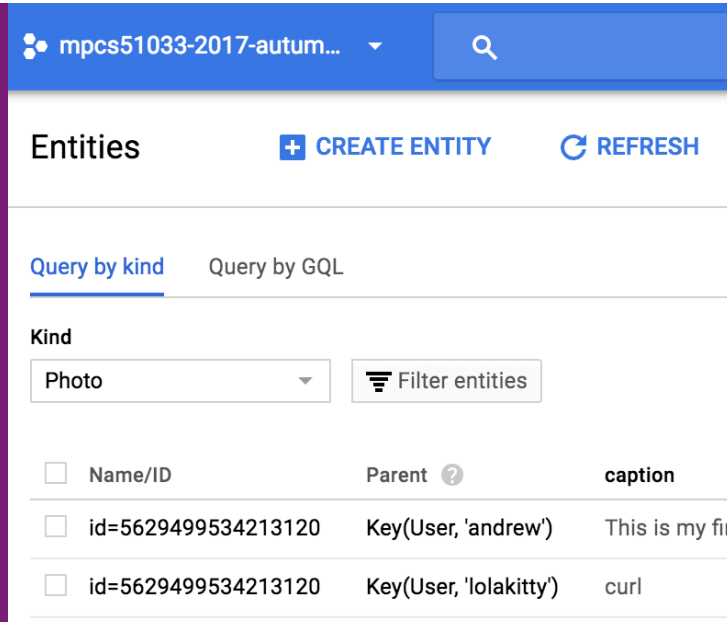


PHOTO TIMELINE

FORESEEABLE PROBLEMS

- No security
  - Anyone can post or retrieve with just a username
  - No way to uniquely identify a user

PHOTO TIMELINE

FORESEEABLE PROBLEMS

- No API to delete embarrassing photos

PHOTO TIMELINE

FORESEEABLE PROBLEMS

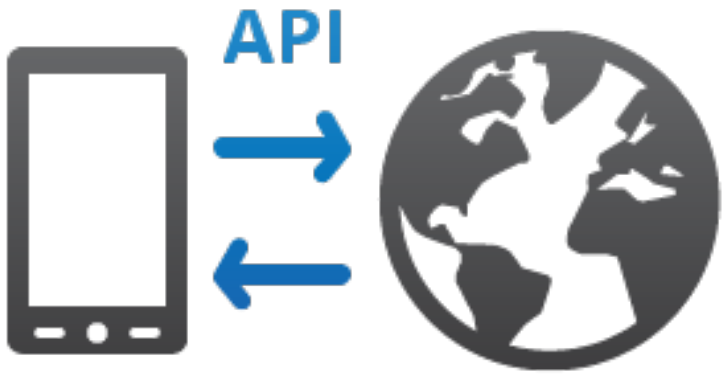
- No API to delete embarrassing photos
- No API to interact with a mobile device

BREAK TIME 

API DESIGN

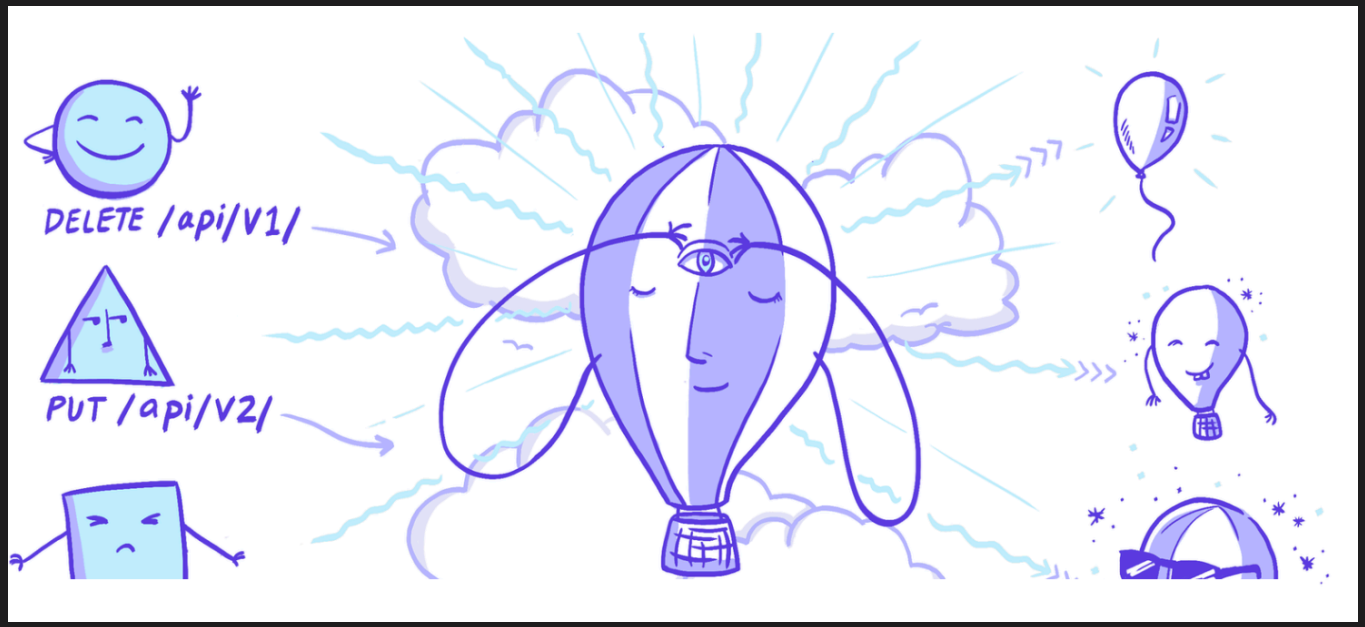
API DESIGN

- APIs expose web services hosted by web servers
- Consumers may be developers or end users
- The operations that a web service exposes constitute the public API



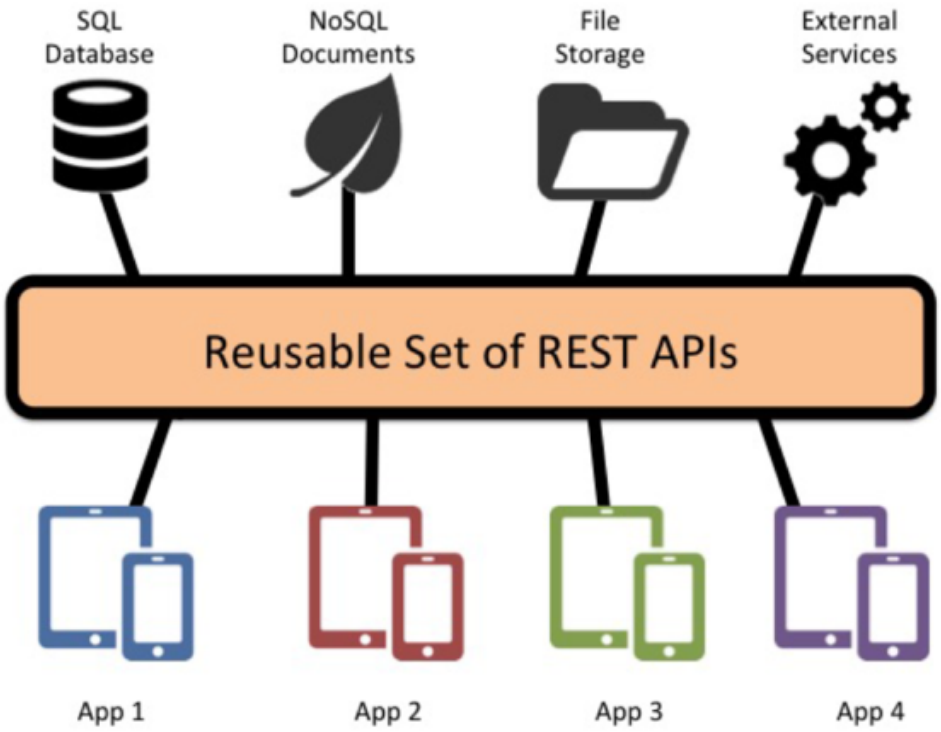
API DESIGN

- Well-designed APIS should aim to support
  - Platform independence
  - Service evolution



API DESIGN

- Platform independence
  - Clients should be able to interact with API regardless of platform (unless as specified)
  - API should have commons standards of interaction across all clients
  - Data formats and structure should be consistent

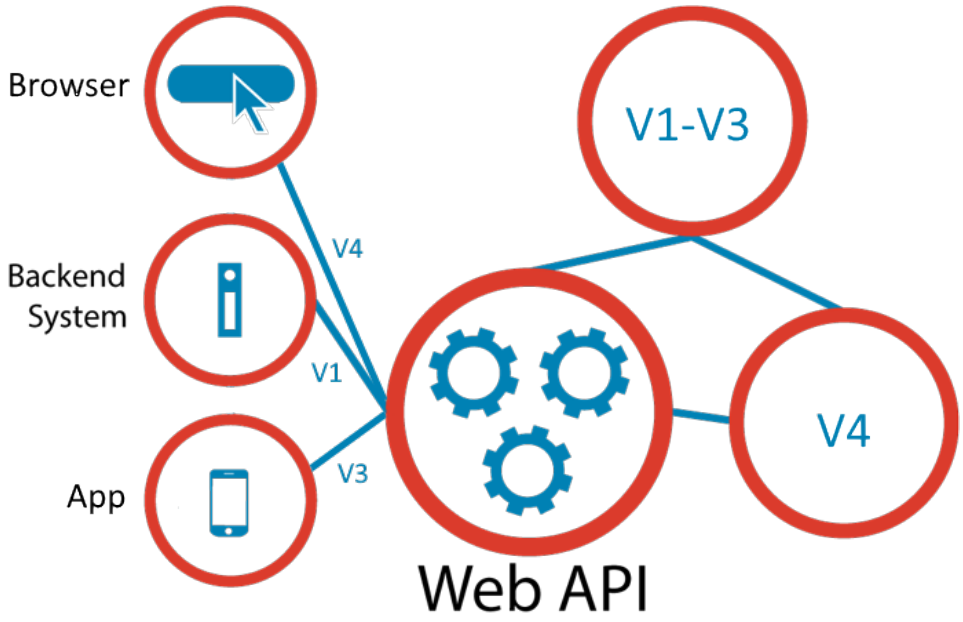


API DESIGN

- The same API should work for every client
- Keep it (as) simple (as possible)

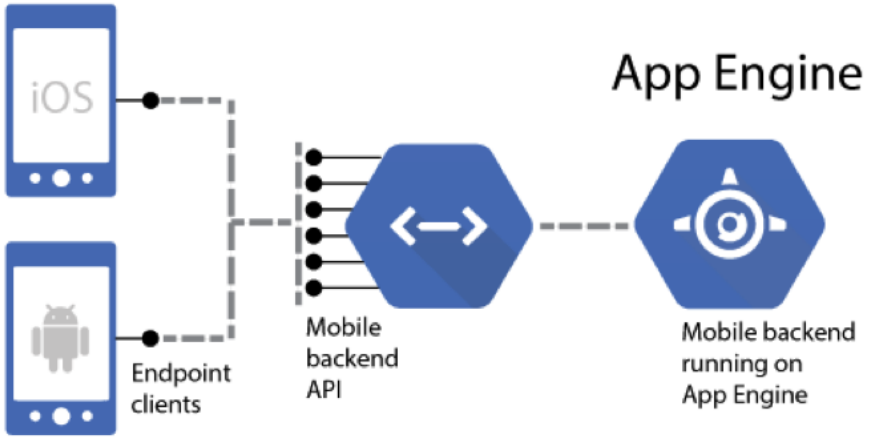
API DESIGN

- Service evolution
  - Web series should be able to evolve (add/remove) functionality
  - Independently of the client
  - Existing clients should continue to work



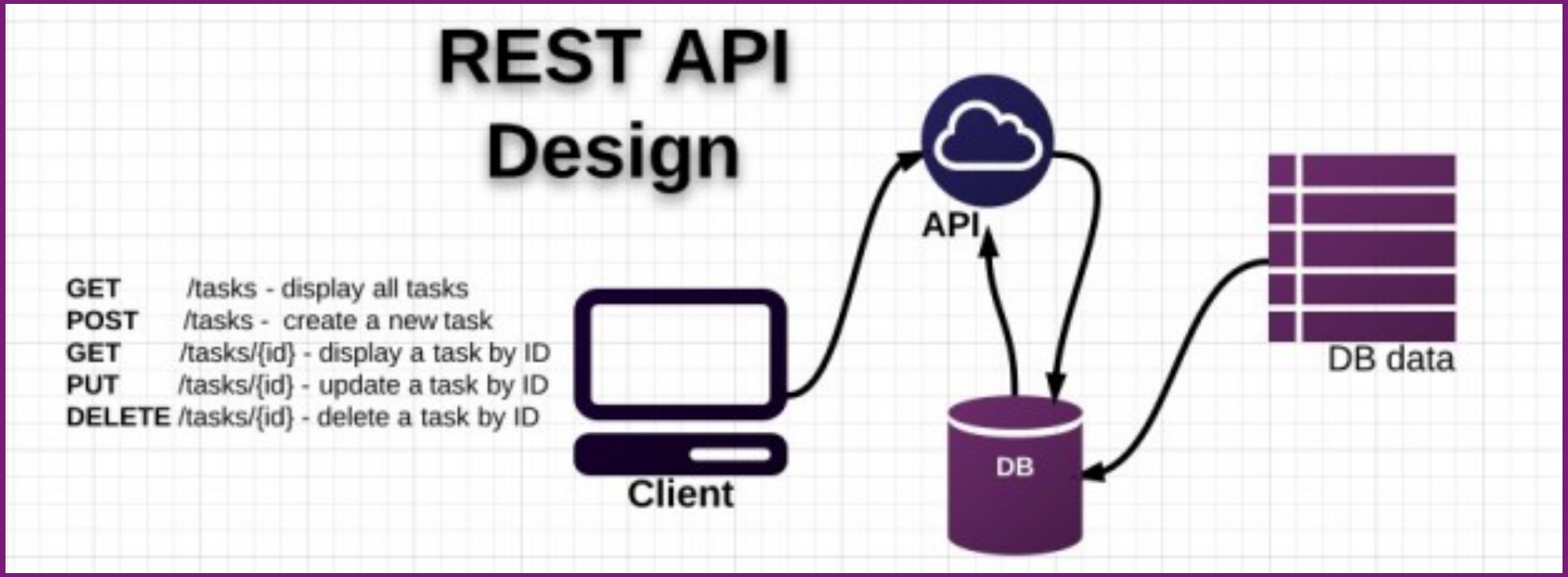
API DESIGN

- Special considerations for mobile
  - Backend and API can/should change more frequently than mobile app
  - Think about what work you can/should offload
  - Feature parity



REPRESENTATIONAL STATE TRANSFER (REST)

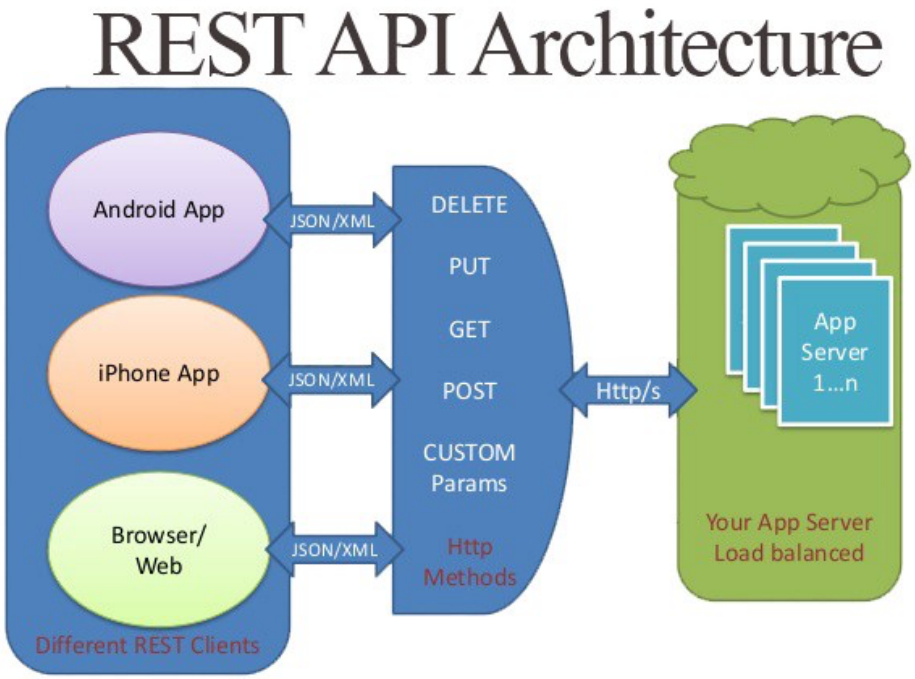
REST



- REST is architectural style first described in 2000 by Roy Fielding as an architectural approach to structuring the operations exposed by web services

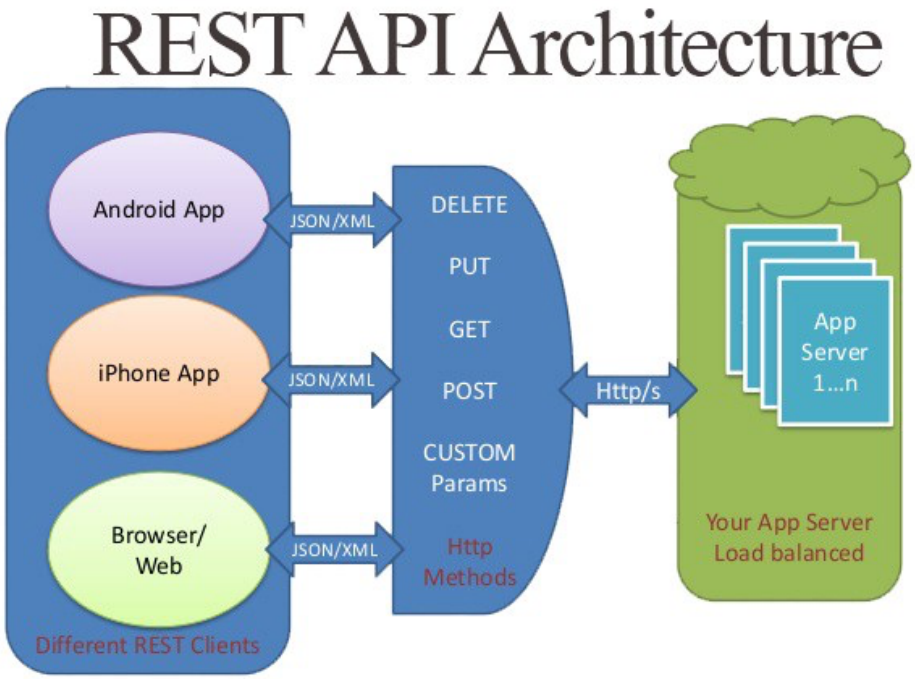
REST

- REST is described for building distributed systems based on hypermedia (eg. http)
  - Rest is not tied to HTTP, but is just the most common implementation of it



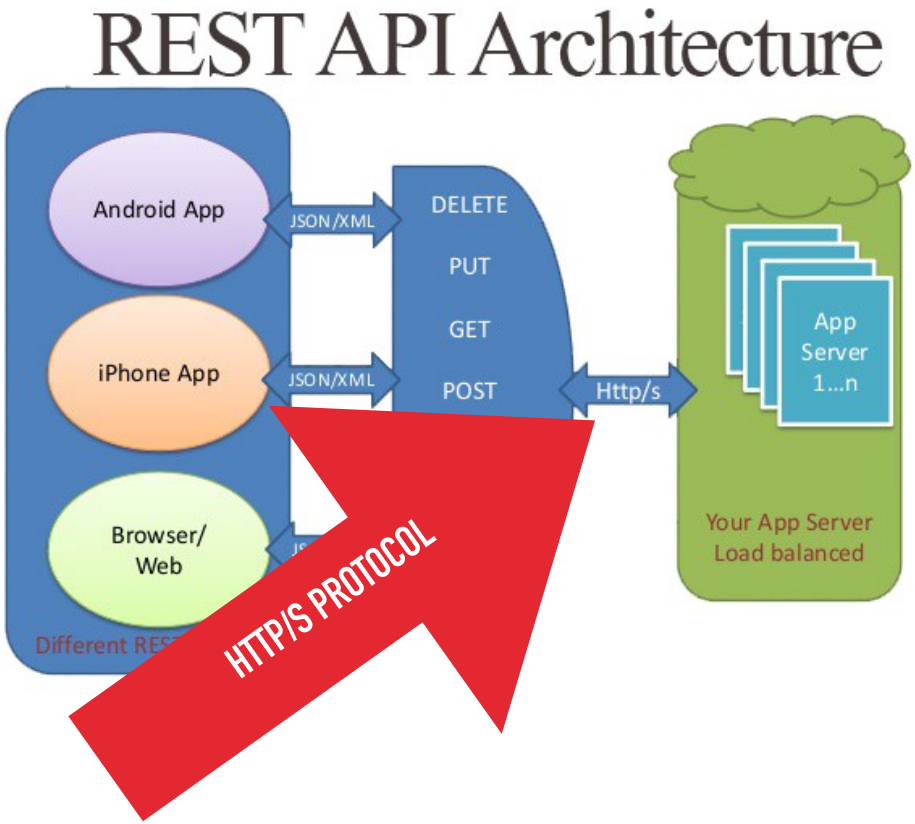
REST

- Based on open standards standards and is not tied to any language or toolset



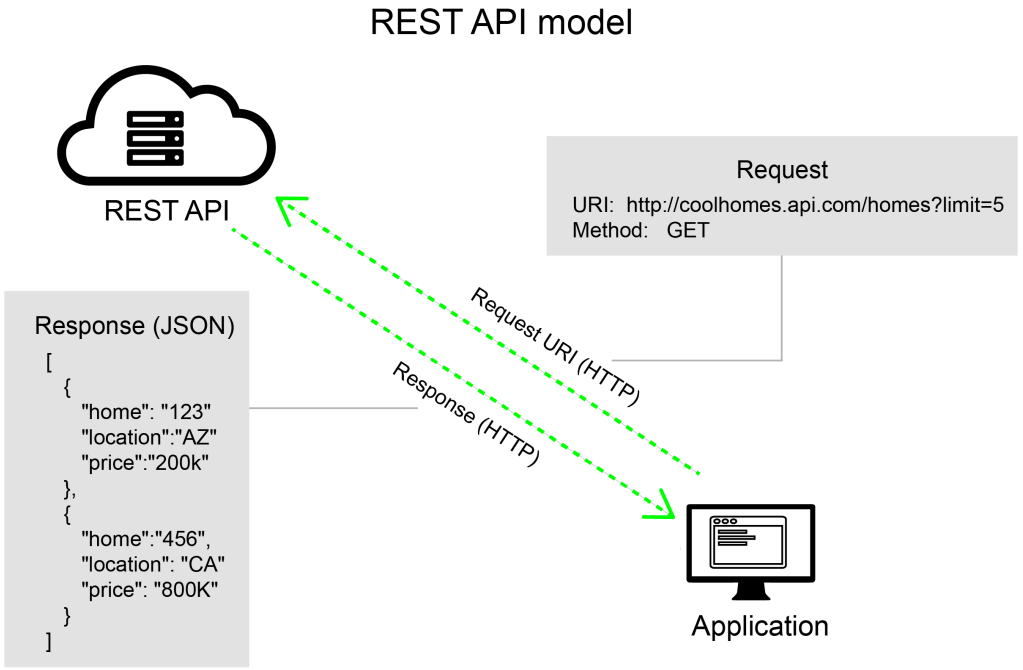
REST

- The REST model uses a navigational scheme to represent objects and services over a network
  - Referred to as resources
- Most systems that implement REST use the HTTP protocol to transmit requests to access these resources



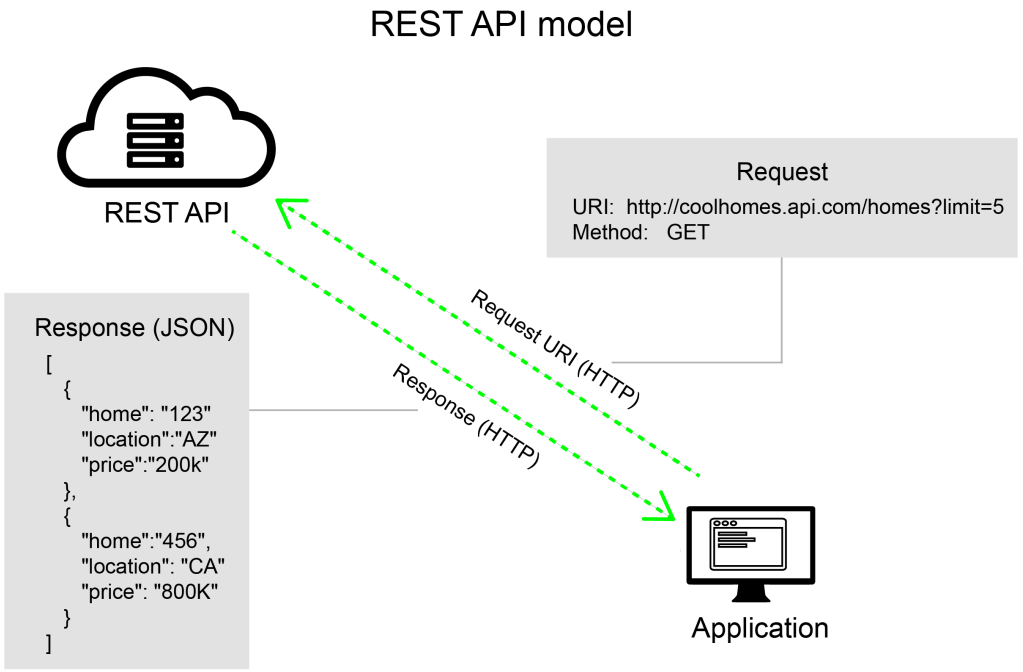
REST

- A client application request form
  - URI (Uniform Resource Identifier) that identifies a resource
  - HTTP method (the most common being GET, POST, PUT, or DELETE) that indicates the operation to be performed on that resource
  - The body of the HTTP request contains the data required to perform the operation



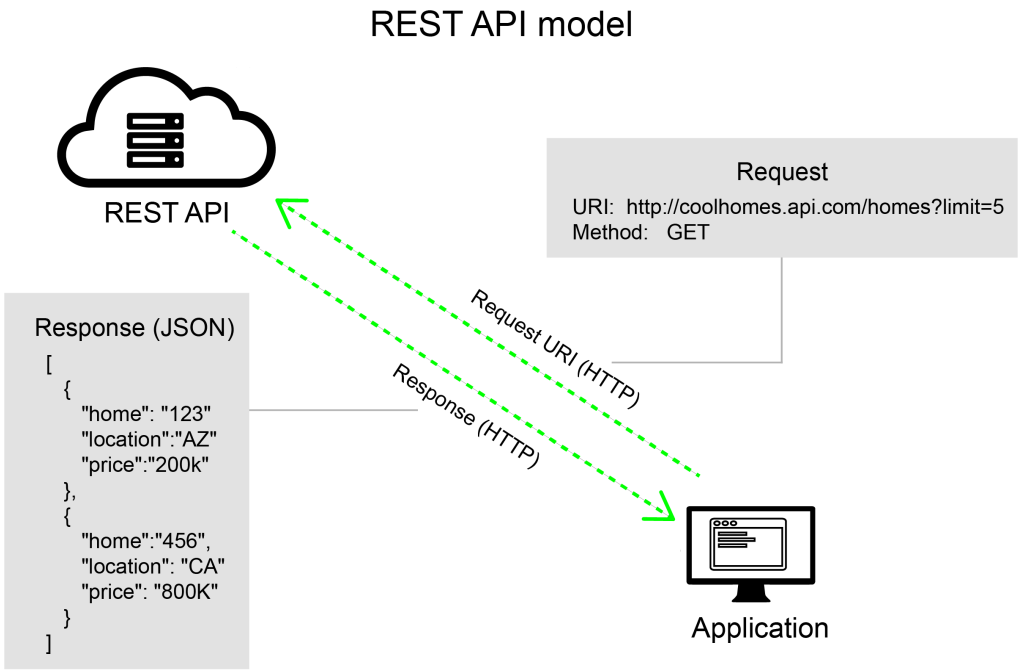
REST

- REST defines a stateless request model
- HTTP requests should be independent and may occur in any order
  - Retaining transient state information between requests is not feasible
- Information is only stored is in the resources themselves
  - Each request should be an atomic operation



REST

- Always bear in mind that the data on the server will not always be the same for all users at the same time
  - Consistency
- Does it matter?
  - If so, design for it 🤔
  - If not, don't worry about it 😃



REST

- An effective REST model defines the relationships between resources to which the model provides access
  - Collections
  - Relationships

```
# Collections in a shopping cart
# API


# URI for customers
/customers


# URI for orders
/orders
```

REST

```
# Issuing an HTTP GET URI to get collection of
# customers
GET http://site.com/customers  HTTP/1.1


# Issuing an HTTP GET URI to get collection of
# orders
GET http://site.com/orders  HTTP/1.1
```

REST

```
# Issuing an HTTP GET URI to get collection of
# orders
GET http://site.com/orders  HTTP/1.1
```

```
HTTP/1.1 200 OK
Date: Fri, 22 Aug 2014 08:49:02 GMT
Content-Length: ...
[
{"orderId":1,"orderValue":99.90,"productId":1,"quantity":1},
{"orderId":2,"orderValue":10.00,"productId":4,"quantity":2}
]
```

Response to GET request to /orders is an array of orders👌

REST

Resources should have unique identifier

```
# To fetch an individual order requires
# specifying the identifier for the order from
# the orders resource, such as /orders/2

GET http://site.com/orders/2 HTTP/1.1


HTTP/1.1 200 OK
...
Date: Fri, 22 Aug 2014 08:49:02 GMT
Content-Length: ...
{"orderId":2,"orderValue":10.00,"productId":4,"quantity":2}
```

REST

- Responses can return any type of data supported by HTTP
  - Text, binary encoded, encrypted
- The `content-type` should be set in the response header
- Request can also include a "accept" header

Common examples [edit]

- `application/javascript`
- `application/json`
- `application/x-www-form-urlencoded`
- `application/xml`
- `application/zip`
- `application/pdf`
- `audio/mpeg`
- `audio/vorbis`
- `multipart/form-data`
- `text/css`
- `text/html`
- `text/plain`
- `image/png`
- `image/jpeg`
- `image/gif`

REST

- REST requests should used standard HTTP response codes
  - 200s OK
  - 300s Redirect
  - 400s Client error
  - 500s Server error

400 Bad Request

The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, too large size, invalid request message framing, or deceptive request routing).[31]

401 Unauthorized (RFC 7235)

Similar to *403 Forbidden*, but specifically for use when authentication is req provided. The response must include a WWW-Authenticate header field co requested resource. See Basic access authentication and Digest access au means "unauthenticated",[33] i.e. the user does not have the necessary cred

Note: Some sites issue HTTP 401 when an IP address is banned from the w and that specific address is refused permission to access a website.

402 Payment Required

Reserved for future use. The original intention was that this code might be u or micropayment scheme, but that has not happened, and this code is not u uses this status if a particular developer has exceeded the daily limit on req

403 Forbidden

The request was valid, but the server is refusing action. The user might not resource.

404 Not Found

The requested resource could not be found but may be available in the futu are permissible.[35]

405 Method Not Allowed

A request method is not supported for the requested resource; for example, data to be presented via POST, or a PUT request on a read-only resource.

REST

- REST requests shoul
  standard HTTP respo
  codes
  - 200s OK
  - 300s Redirect
  - 400s Client error
  - 500s Server error

THERE ARE VERY SPECIFIC CODES WHERE THE INTERPRETATION IS ALWAYS CLEAR

FORBIDDEN / UNAUTHORIZED

400 Bad Request

The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, too large size, invalid request message framing, or deceptive request routing).[31]

401 Unauthorized (RFC 7235)

Similar to *403 Forbidden*, but specifically for use when authentication is requ provided. The response must include a WWW-Authenticate header field cor requested resource. See Basic access authentication and Digest access au means "unauthenticated",[33] i.e. the user does not have the necessary cred

Note: Some sites issue HTTP 401 when an IP address is banned from the w and that specific address is refused permission to access a website.

402 Payment Required

Reserved for future use. The original intention was that this code might be u or micropayment scheme, but that has not happened, and this code is not u uses this status if a particular developer has exceeded the daily limit on requ

403 Forbidden

The request was valid, but the server is refusing action. The user might not resource.

404 Not Found

The requested resource could not be found but may be available in the futu are permissible.[35]

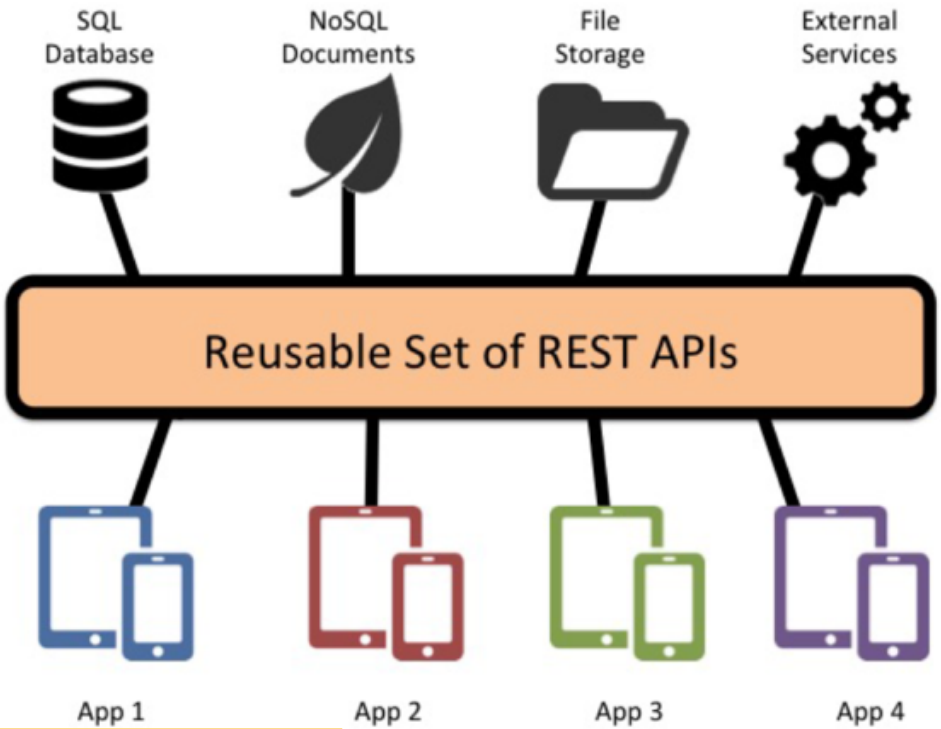
405 Method Not Allowed

A request method is not supported for the requested resource; for example, data to be presented via POST, or a PUT request on a read-only resource.

DESIGN AND STRUCTURE OF A REST WEB API

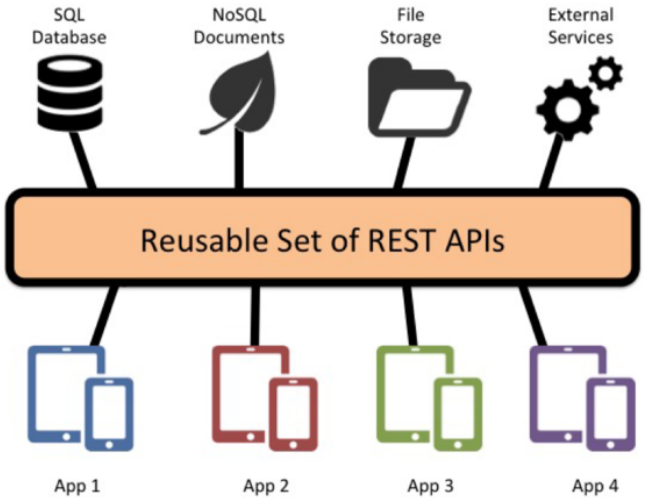
DESIGN OF A REST API

- Goal is to make it easy to build client applications
  - Exposing connected resources
  - Provide core operations that enable application to navigate and manipulate resources
- Most important thing is consistency and simplicity



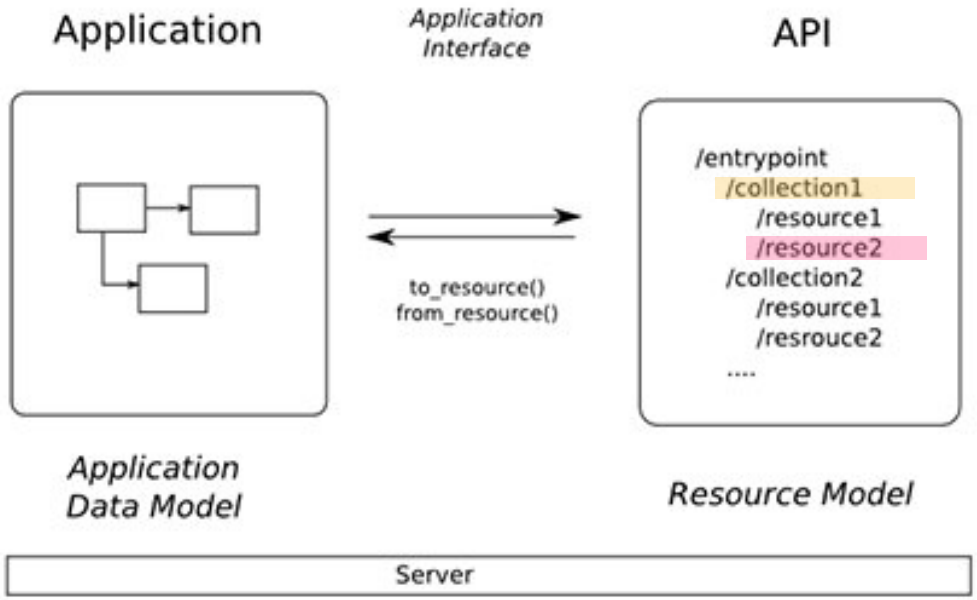
DESIGN OF A REST API

- Different than an API for classes in OOP
  - Motivated by behavior and classes to store and manipulate data
- RESTful web API should be stateless
  - Async nature of requests



DESIGN OF A REST API

- Organize API around resources
  - Focus on the entities (eg. Customers, orders, products, stock)
  - A resource does not have to represent a single entity; can be a combination of information
    - Example: Product, id, stock level, ship date, price, recommended products



DESIGN OF A REST API

- Multiple actions can be taken by a resource
- Example: POST a new order
  - Add to list of orders
  - Check stock levels
  - Bill the customer
  - The response can indicate if the order was successful or not

DESIGN OF A REST API

- URIs should be based on nouns representing the data
- Not verbs describing what the application can do with the data

```
# Nouns
/customers
/orders
/products

# Not verbs
/list/customers
/add/orders
/remove/products
```

DESIGN OF A REST API

- Each entity and collection are resources and should have their own unique URI
- Organize hierarchical manner

```
/customers     # All customers
/customers/5   # Customer id 5

/orders        # All orders
/orders/5      # Order number 5
```

DESIGN OF A REST API

- Each entity and collection are resources and should have their own unique URI
- Organize hierarchical manner

Use plural nouns for collections

```
/customers      # All customers
/customers/5    # Customer id 5

/orders         # All orders
/orders/5       # Order number 5
```

DESIGN OF A REST API

```
# The relationship between different types of
# resources

# All orders for customer 5
/customers/5/orders

# Customer order 999
/orders/999/customer
```

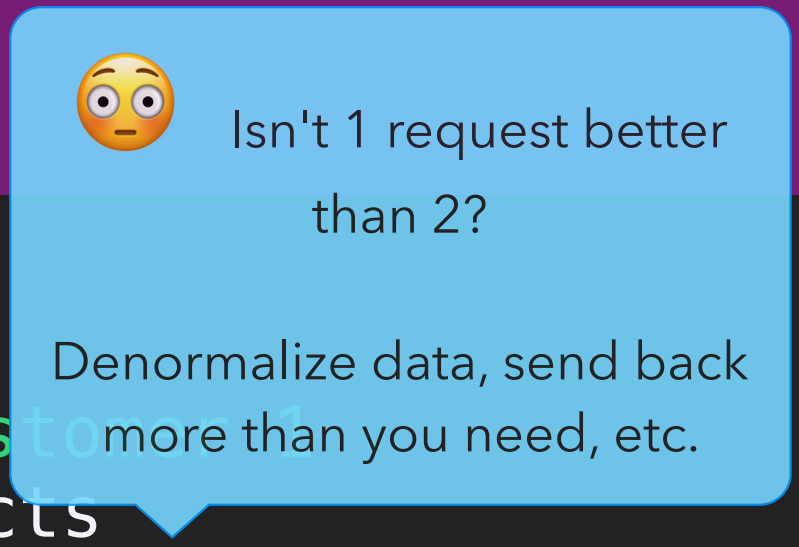
Do we want the customer information or the order information?

DESIGN OF A REST API

Do we need to expose this API?

```
# Products in order 99 by customer 1
/customers/1/orders/99/products

# Simplified
/customers/1/orders  # Order information
/orders/99/products  # Product information
```

DESIGN OF A REST API




```
# Products in order 99 by customer 1
/customers/1/orders/99/products

# Simplified
/customers/1/orders  # Order information
/orders/99/products  # Product information
```

DESIGN OF A REST API

- Denormalizing data is a strategy to optimize resources at the expense of data redundancy
  - Firebase 🔥

```
# Products in orde
/customers/1/orders

# Simplified
/customers/1/orders
/orders/99/products
```

🤔

What should I optimized my data for?

OPERATIONS IN TERMS OF HTTP METHODS

OPERATIONS IN TERMS OF HTTP METHODS

- The HTTP protocol defines a number of methods that assign semantic meaning to a request
  - GET
  - POST
  - PUT
  - DELETE

Instance Methods

Subclasses of the RequestHandler class inherit or override the following methods:

`get(*args)`

Called to handle an HTTP GET request. Overridden by handler subclasses.

`post(*args)`

Called to handle an HTTP POST request. Overridden by handler subclasses.

`put(*args)`

Called to handle an HTTP PUT request. Overridden by handler subclasses.

`head(*args)`

Called to handle an HTTP HEAD request. Overridden by handler subclasses.

`options(*args)`

Called to handle an HTTP OPTIONS request. Overridden by handler subclasses.

`delete(*args)`

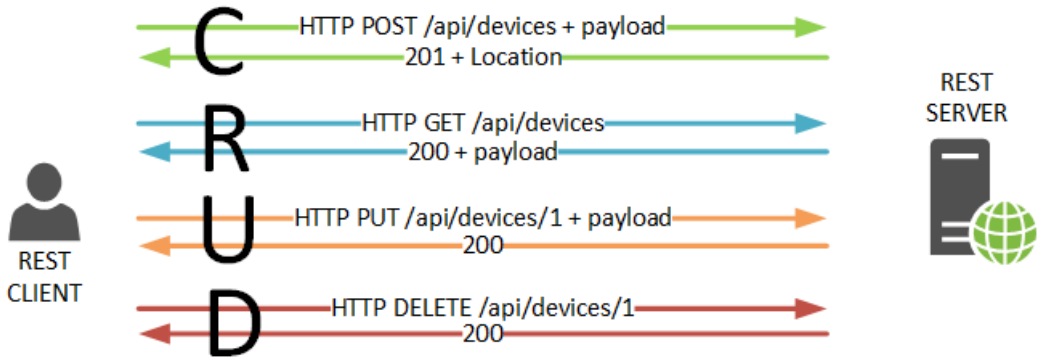
Called to handle an HTTP DELETE request. Overridden by handler subclasses.

`trace(*args)`

Called to handle an HTTP TRACE request. Overridden by handler subclasses.

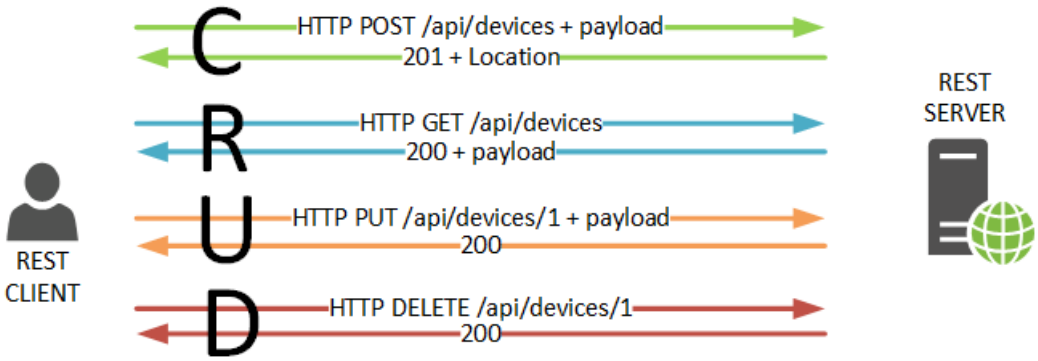
OPERATIONS IN TERMS OF HTTP METHODS

- POST
  - Create a new resource at the specified URI
  - The body of the request message provides the details of the new resource
  - Note that POST can also be used to trigger operations that don't actually create resources



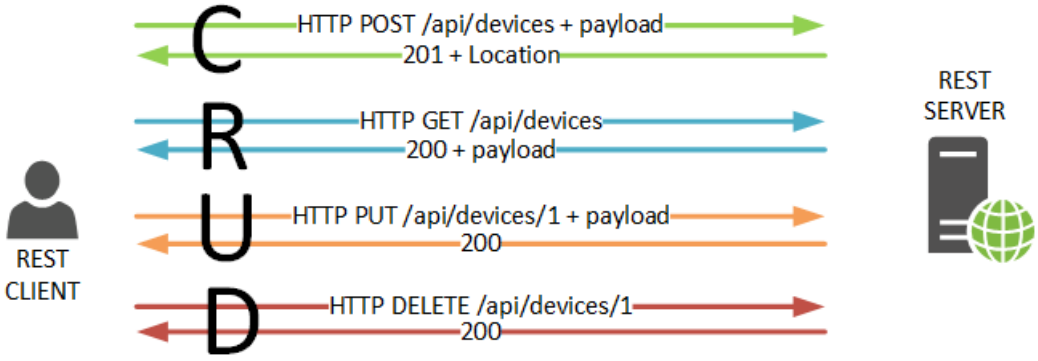
OPERATIONS IN TERMS OF HTTP METHODS

- GET
  - Retrieve a copy of the resource at the specified URI
  - The body of the response message contains the details of the requested resource



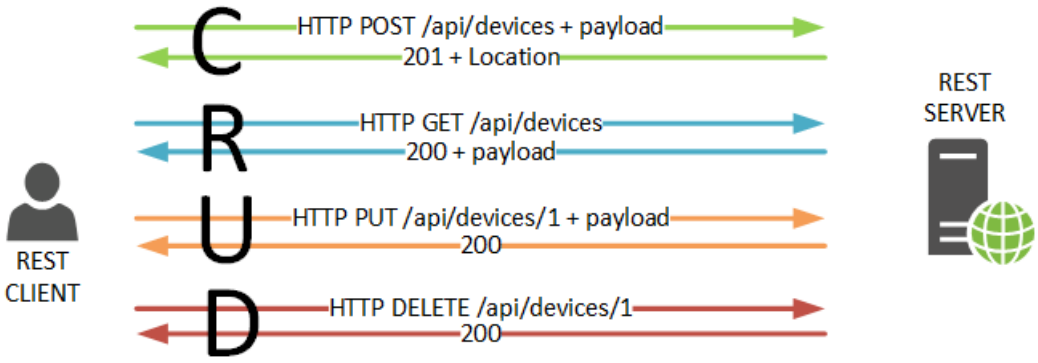
OPERATIONS IN TERMS OF HTTP METHODS

- PUT
  - To replace or update the resource at the specified URI
  - The body of the request message specifies the resource to be modified and the values to be applied



OPERATIONS IN TERMS OF HTTP METHODS

- DELETE
  - To remove the resource at the specified URI



OPERATIONS IN TERMS OF HTTP METHODS

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers (*if implemented*)	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists, otherwise return an error	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1 (*if implemented*)	Remove all orders for customer 1(*if implemented*)

DESIGN OF A REST API

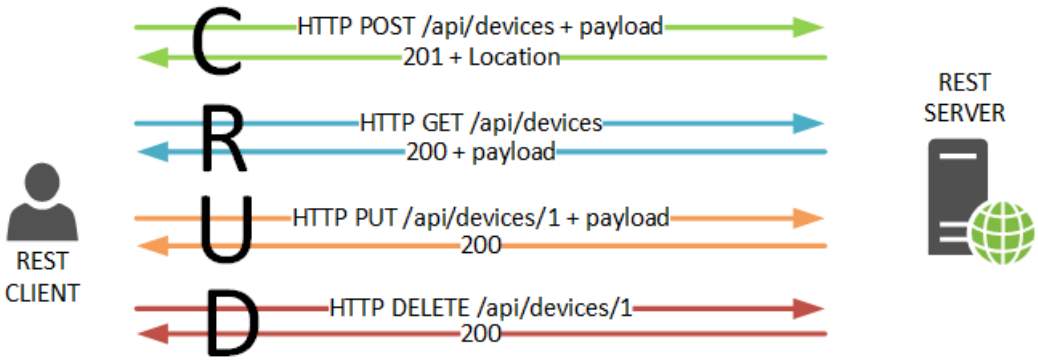
Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers (*if implemented*)	Remove all customers

```
# The old way (before we knew any better)
/customers/add/
/customers/update/
/customers/delete/


# One URI for all customer related resources
# by meaningfully using the HTTP methods
/customers/
```

OPERATIONS IN TERMS OF HTTP METHODS

- POST request to create a new resource with data provided in the body of the request
  - Apply POST requests to resources that are collections
  - New resource is added to the collection

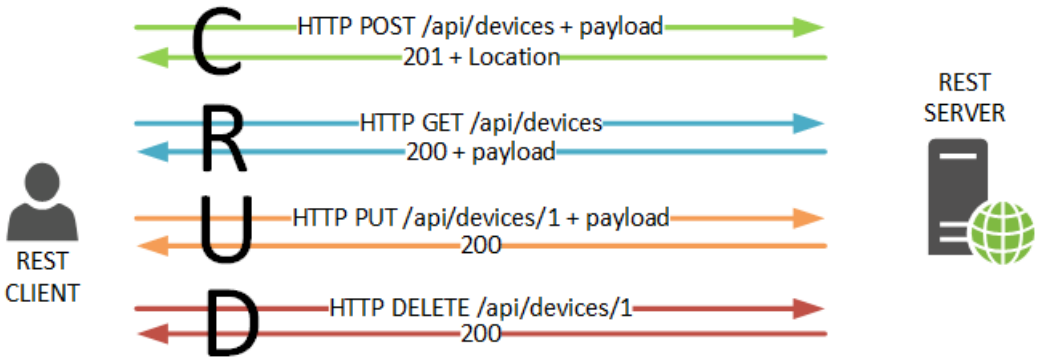
POST VS. PUT



OPERATIONS IN TERMS OF HTTP METHODS

- PUT request to modify an existing resource
  - If the specified resource does not exist, the PUT request could return an error
  - Applied to resources that are individual items (such as a specific customer or order)

POST VS. PUT



OPERATIONS IN TERMS OF HTTP METHODS

- POST vs PUT
  - A POST request should create a new resource with data provided in the body of the request
    - Apply POST requests to resources that are collections; the new resource is added to the collection
  - PUT request is intended to modify an existing resource
    - If the specified resource does not exist, the PUT request could return an error
    - Applied to resources that are individual items (such as a specific customer or order)

HTTP **`PATCH`** request to update a property in a resource; rarely used

PROCESSING HTTP REQUESTS

PROCESSING HTTP REQUESTS

- Requests can specify the types of media they expect to results to be returned
- Doesn't have to be honored
- If you control both ends...

```
GET http://site.com/orders/2

Accept: application/json
```

Pass in header

PROCESSING HTTP REQUESTS

- Requests can specify the types of media they expect to results to be returned
- Doesn't have to be honored
- If you control both ends...

```
GET http://site.com/orders/2

Accept: application/json
```

Could send HTTP 415 unsupported media type

PROCESSING HTTP REQUESTS

```
GET http://site.com/orders/2

Accept: application/json
```

Response declares content type

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

PROCESSING HTTP REQUESTS

- PUT request with data to be modified
- Urlencoded key/ value pairs seperated by "&"

```
# Request
PUT http://site.com/orders/1 HTTP/1.1


Content-Type: application/x-www-form-urlencoded


Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
ProductID=3&Quantity=5&OrderValue=250
```

PROCESSING HTTP REQUESTS

- Successful PUT returns 204 with no data in body
- Location of resource is in header

```
# Request
PUT http://site.com/orders/1 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
ProductID=3&Quantity=5&OrderValue=250
```

```
# Response
HTTP/1.1 204 No Content

Location: http://site.com/orders/1
Date: Fri, 22 Aug 2014 09:18:37 GMT
```

PROCESSING HTTP REQUESTS

- Successful POST should return 201 (created)
- The location should contain the URI
- Body should contain a copy of resource
  - Alternatively 200 with no body

```
# Request
POST http://site.com/orders HTTP/1.1

...
Content-Type: application/x-www-form-
urlencoded
...

Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
productID=5&quantity=15&orderValue=400
```

PROCESSING HTTP REQUESTS

```
# Request
POST http://site.com/orders HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
productID=5&quantity=15&orderValue=400

# Response
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: http://site.com/orders/99
Date: Fri, 22 Aug 2014 09:18:37 GMT
Content-Length: ...
{"orderID":99,"productID":5,"quantity":15,"orderValue":400}
```

RETURNS A COPY OF THE RESOURCE

PROCESSING HTTP REQUESTS

- To remove a resource, an HTTP DELETE request simply provides the URI of the resource to be deleted
- Respond with 204

```
# The following example attempts to
# remove order 99 (specified in params)
DELETE http://site.com/orders/ HTTP/1.1


HTTP/1.1 204 No Content
Date: Fri, 22 Aug 2014 09:18:37 GMT
```

Alternative 200 (OK) 202 (Accepted)

PROCESSING HTTP REQUESTS

- To remove a resource, an HTTP DELETE request simply provides the URI of the resource to be deleted
- Respond with 204

```
# The following example attempts to
remove order 99
DELETE http://site.com/orders/99 HTTP/1.1


HTTP/1.1 204 No Content
Date: Fri, 22 Aug 2014 09:18:37 GMT
```

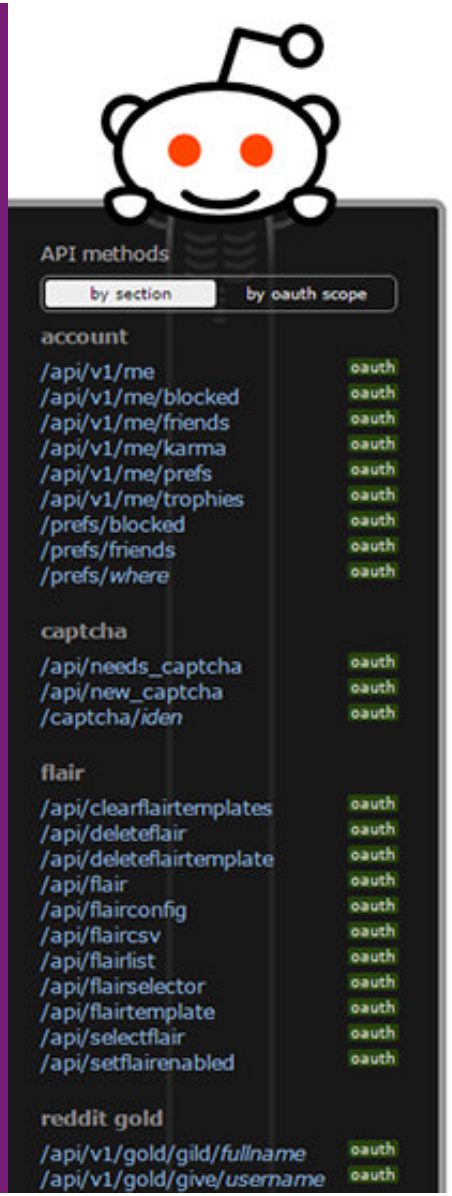
Alternative 200 (OK) 202 (Accepted)

PROCESSING HTTP REQUESTS

- HTTP status code of 400 (Bad Request)
  - Send when the request is invalid

PROCESSING HTTP REQUESTS

- Look at some existing APIs



This is automatically-generated documentation for the reddit API.

The reddit API and code are open source. Found a mistake or interested in helping us improve? Have a gander at api.py and send us a pull request.

Please take care to respect our API access rules.

overview

listings

Many endpoints on reddit use the same protocol for controlling pagination and filtering. These endpoints are called Listings and share five common parameters: `after` / `before`, `limit`, `count`, and `show`.

Listings do not use page numbers because their content changes so frequently. Instead, they allow you to view slices of the underlying data. Listing JSON responses contain `after` and `before` fields which are equivalent to the "next" and "prev" buttons on the site and in combination with `count` can be used to page through the listing.

The common parameters are as follows:

- `after` / `before` - only one should be specified. these indicate the fullname of an item in the listing to use as the anchor point of the slice.
- `limit` - the maximum number of items to return in this slice of the listing.
- `count` - the number of items already seen in this listing. on the html site, the builder uses this to determine when to give values for `before` and `after` in the response.
- `show` - optional parameter; if `all` is passed, filters such as "hide links that I have voted on" will be disabled.

To page through a listing, start by fetching the first page without specifying values for `after` and `count`. The response will contain an `after` value which you can pass in the next request. It is a good idea, but not required, to send an updated value for `count` which should be the number of items already fetched.

modhashes

A modhash is a token that the reddit API requires to help prevent CSRF. Modhashes can be obtained via the /api/me.json call or in response data of listing endpoints.

VERSIONING A REST API

VERSIONING A REST API

- Structure and organization of resources may change or be amended
- If you are in control of both sides can be a non-issue
- Simplest way to handle versioning is not to version if the changes are non breaking
  - Add new fields to response

VERSIONING A REST API

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: ...
{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond
WA 98053"}
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: ...
{"id":3,"name":"Contoso
LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":"1
Microsoft Way Redmond WA 98053"}
```

VERSIONING A REST API

- URI versioning schemes
- Developer has to support them

```
http://site.com/v1/customers/3

http://site.com/v2/customers/3

# Default to latest
http://site.com/customers/3
```

VERSIONING A REST API

- Header versioning

```
# Version 2
GET http://site.com/customers/3

Custom-Header: api-version=1
```

```
# Version 2
GET http://site.com/customers/3

Custom-Header: api-version=2
```

SUMMARY

SUMMARY

- Guidelines for developing a stable API
- Well-designed APIs should aim to support
  - Platform independence
  - Service evolution

SUMMARY

- Implementations vary
  - PUT, DELETE, PATCH
- Look at existing APIs for best use practices

SUMMARY

- Github API

HTTP Verbs

Where possible, API v3 strives to use appropriate HTTP verbs for each action.

Verb	Description
`HEAD`	Can be issued against any resource to get just the HTTP header info.
`GET`	Used for retrieving resources.
`POST`	Used for creating resources.
`PATCH`	Used for updating resources with partial JSON data. For instance, an Issue resource has `title` and `body` attributes. A PATCH request may accept one or more of the attributes to update the resource. PATCH is a relatively new and uncommon HTTP verb, so resource endpoints also accept `POST` requests.
`PUT`	Used for replacing resources or collections. For `PUT` requests with no `body` attribute, be sure to set the `Content-Length` header to zero.

SUMMARY

- YouTube API

Method	HTTP request	Description
URIs relative to **https://www.googleapis.com/youtube/v3**		
delete	**DELETE /playlistItems**	Deletes a playlist item.
insert	**POST /playlistItems**	Adds a resource to a pla
list	**GET /playlistItems**	Returns a collection of p the API request parame of the playlist items in a retrieve one or more pla IDs.
update	**PUT /playlistItems**	Modifies a playlist item. update the item's positi

SUMMARY

- Implementations vary
  - PUT, DELETE, PATCH
- Look at existing APIs for best use practices

- "It depends...."

SUMMARY


SWAGGER
SPECIFICATION
TOOLS
SUPPORT
BLOG
DOCS
SWAGGER
THE WORLD'S MOST POPULAR API TOOLING
Swagger is the world's largest framework of API developer tools for the OpenAPI Specification(OAS), enabling development across the entire API lifecycle, from design and documentation, to test and deployment.

ASSIGNMENT 2

ASSIGNMENT 2

HTTP://UCHICAGO.CLOUD/SESSIONS/SESSION2/

Our data model works fine for a small number of users, but if (when) our app goes viral, querying for a users photos will be inefficient and potentially costly. While we can count on the built-in query memcache and our own memcache solution, we can head off the problem by making changes to our data model.

- Add a `User` model to the application with the following properties:
  - `name`
  - `email`
  - `unique_id` (you can choose to use the `Key` property or use a custom uuid)

THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2019 • SESSION 2

BACKENDS FOR MOBILE APPLICATIONS