THE UNIVERSITY OF CHICAGO

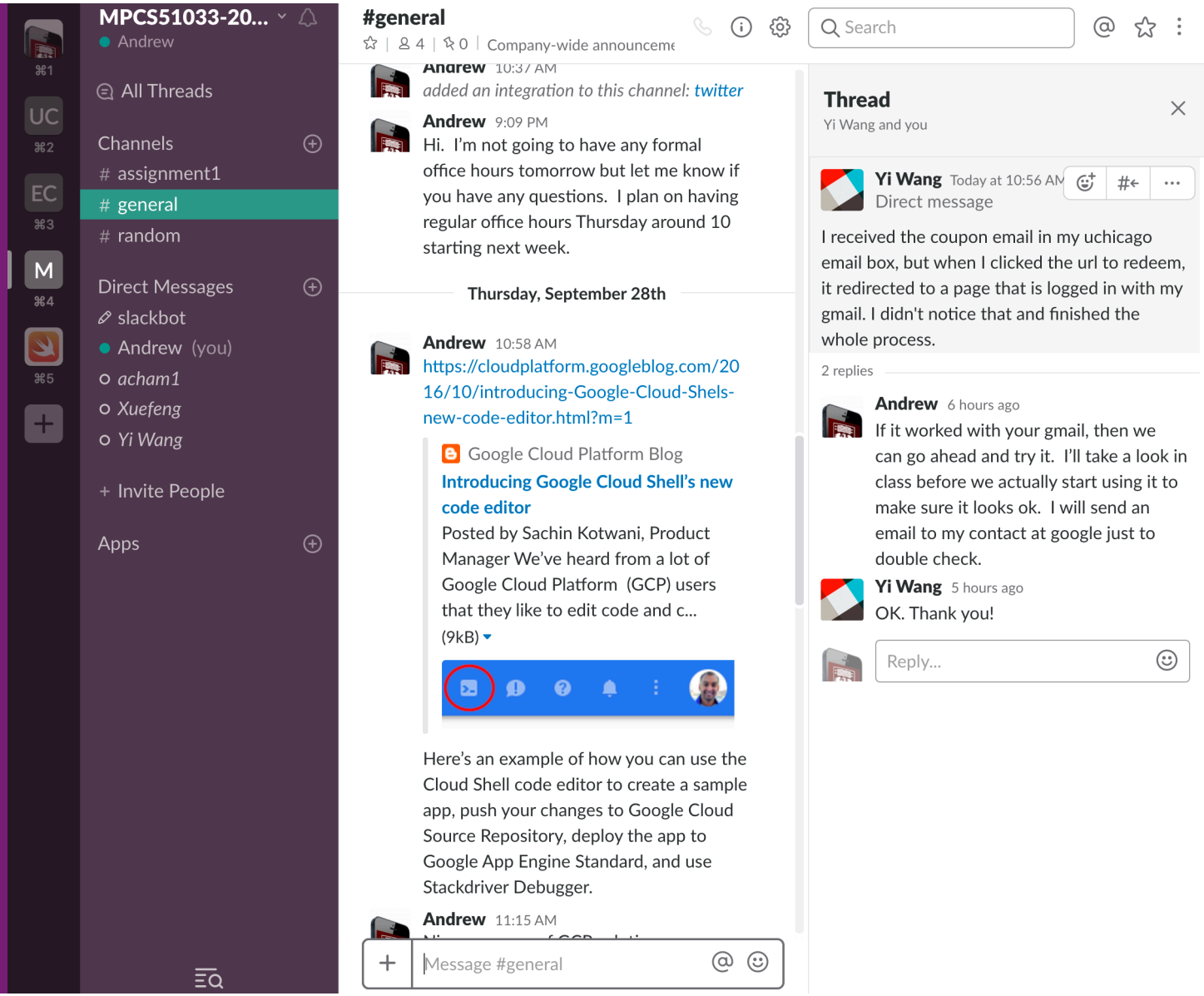
MPCS 51033 • AUTUMN 2017 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS

CLASS NEWS

CLASS NEWS

- Office hours Thursday 10AM in Young 308



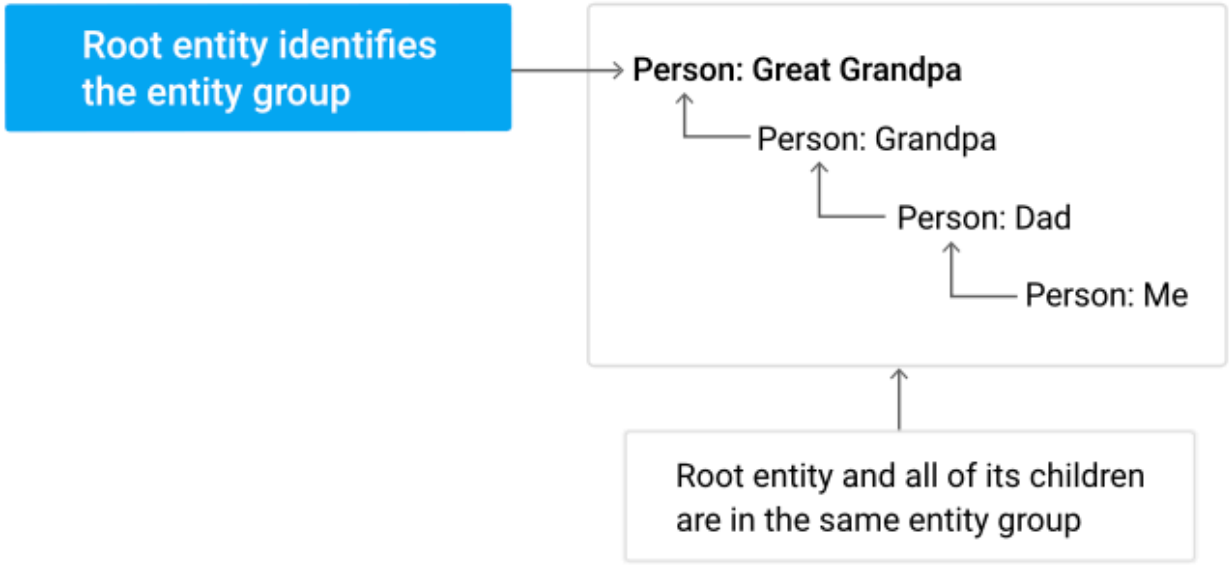
PARENTS JUST DON'T UNDERSTAND

PARENT JUST DON'T UNDERSTAND

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

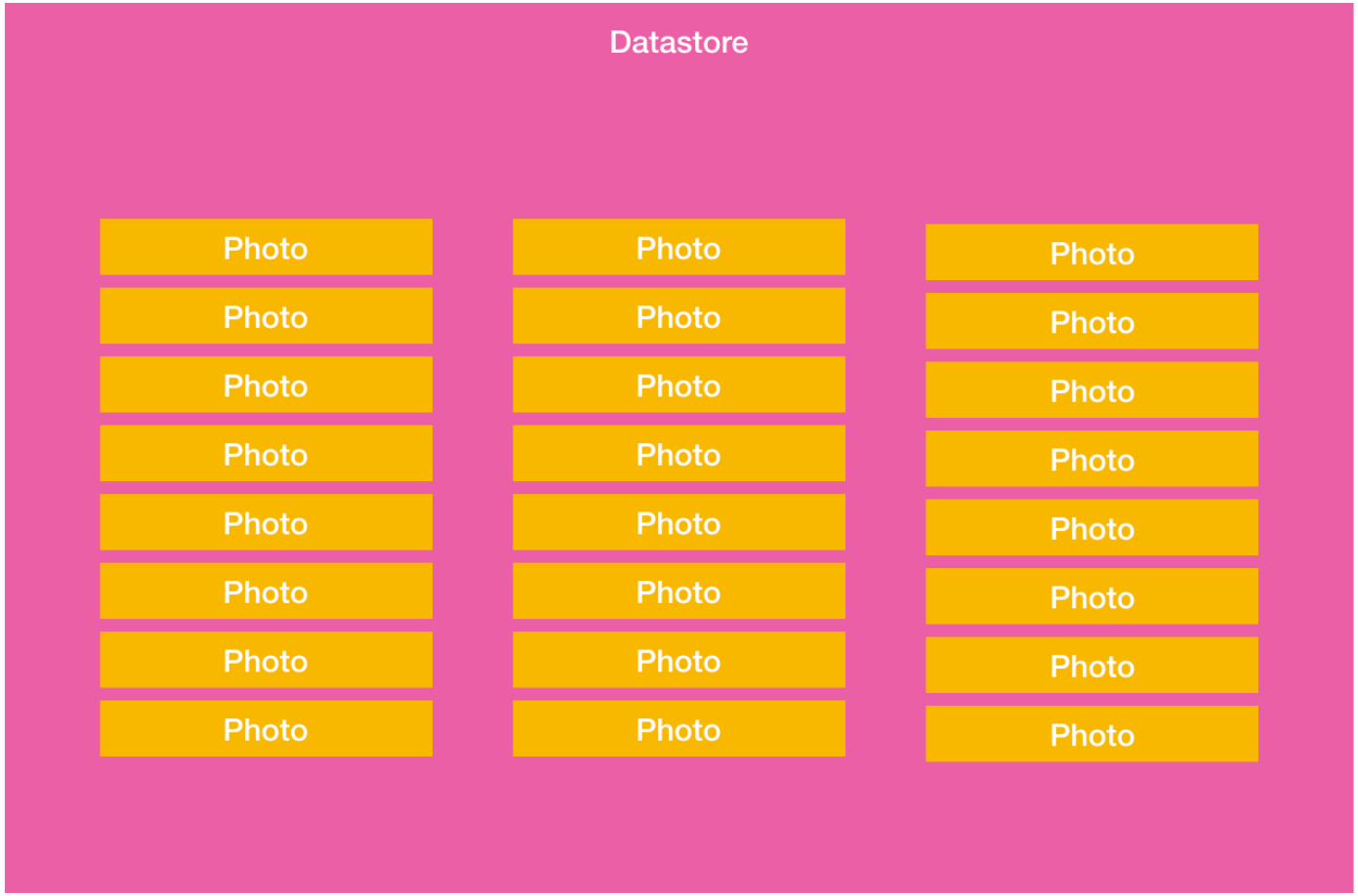
PARENT JUST DON'T UNDERSTAND

- Parent child relationship form a hierarchical structure



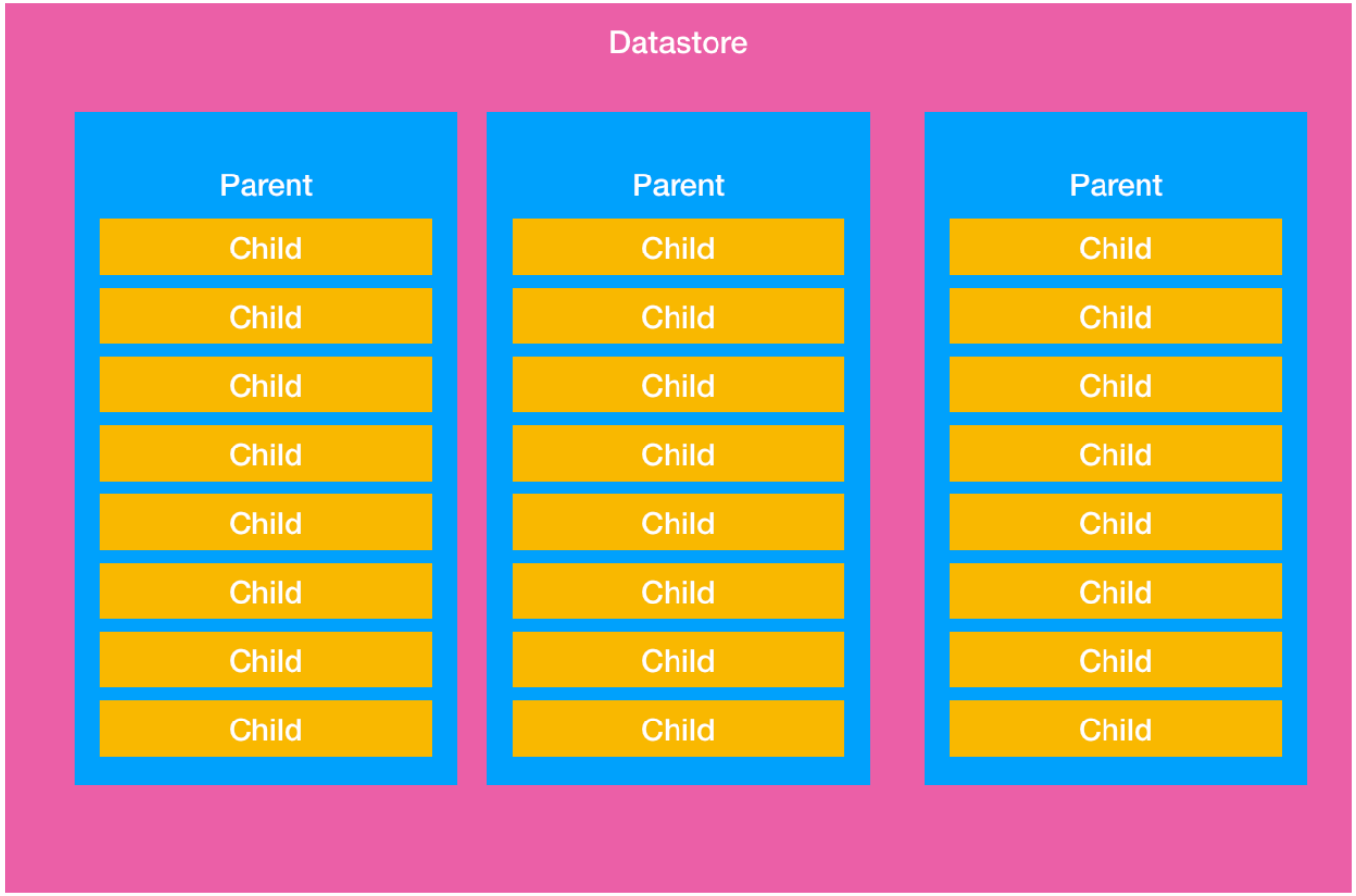
PARENT JUST DON'T UNDERSTAND

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



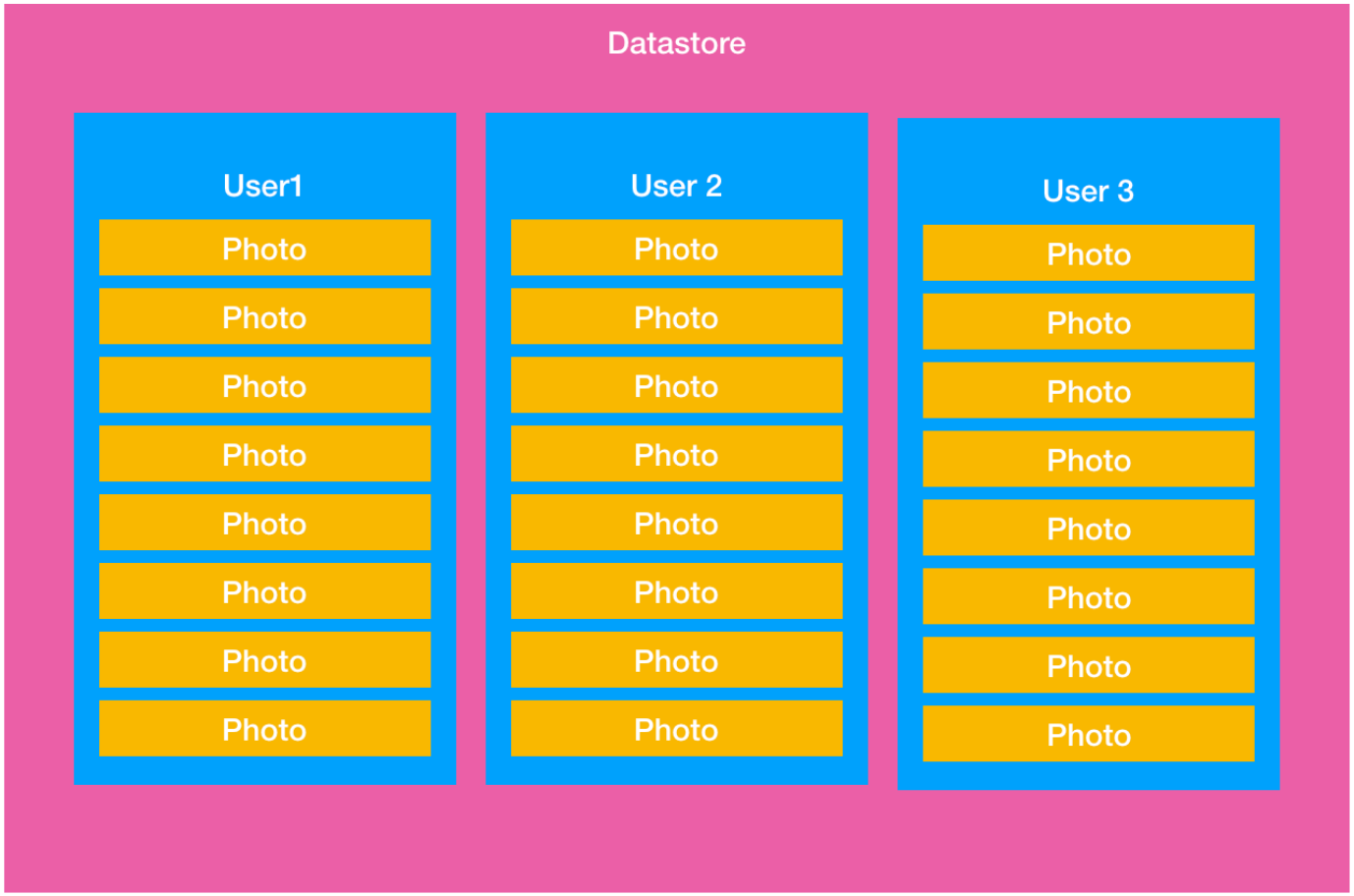
PARENT JUST DON'T UNDERSTAND

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



PARENT JUST DON'T UNDERSTAND

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



PARENT JUST DON'T UNDERSTAND

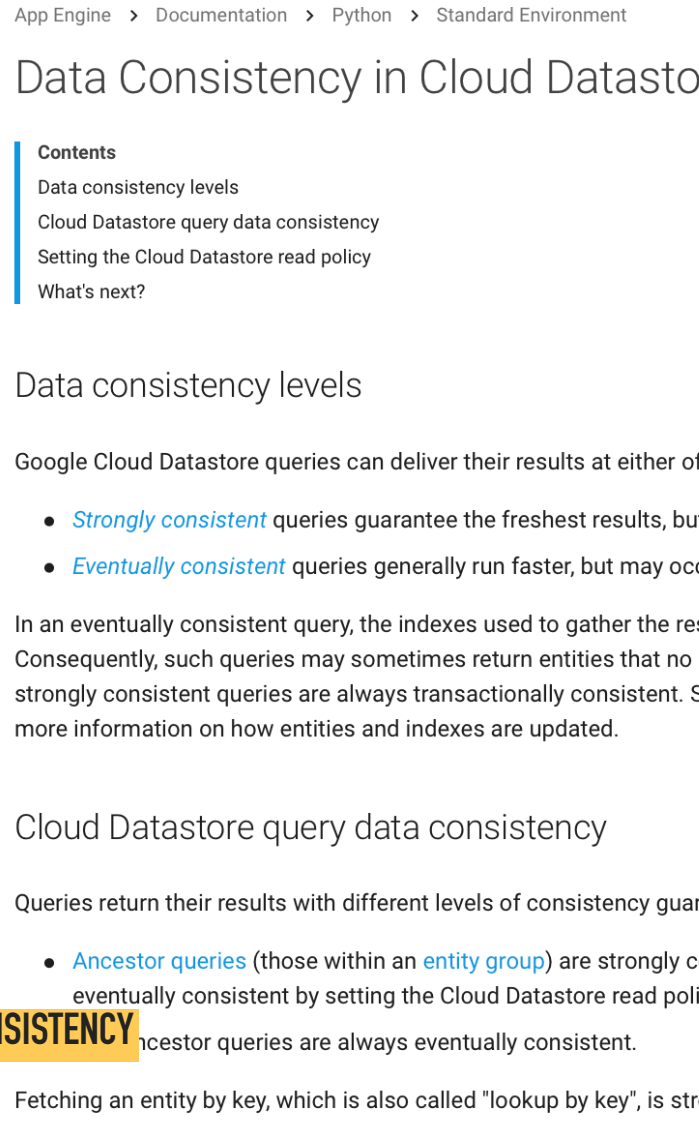
- Ancestor queries
- https://cloud.google.com/appengine/docs/standard/python/datastore/queries

CONSISTENCY

CONSISTENCY

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

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



App Engine › Documentation › Python › Standard Environment

Data Consistency in Cloud Datasto

Contents

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

Data consistency levels

Google Cloud Datastore queries can deliver their results at either of

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

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

Cloud Datastore query data consistency

Queries return their results with different levels of consistency guar

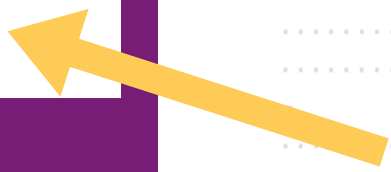
- Ancestor queries (those within an entity group) are strongly c
  eventually consistent by setting the Cloud Datastore read poli
  ncestor queries are always eventually consistent.

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

CONSISTENCY

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

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



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

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

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

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

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

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

CONSISTENCY

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

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

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

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

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

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

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

CONSISTENCY

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

App Engine > Documentation > Python > Standard Environment

Data Consistency in Cloud Datasto

Contents

Data consistency levels

Google Cloud Datastore queries can deliver their results at either of

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

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

Cloud Datastore query data consistency

Queries return their results with different levels of consistency guar

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

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

CONSISTENCY

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

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

done
```

CONSISTENCY

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

Structuring Data for Strong Consistency

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

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

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

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

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

CONSISTENCY

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

Structuring Data for Strong Consistency

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

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

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

USE MEMEACHE TO STORE PENDING WRITES

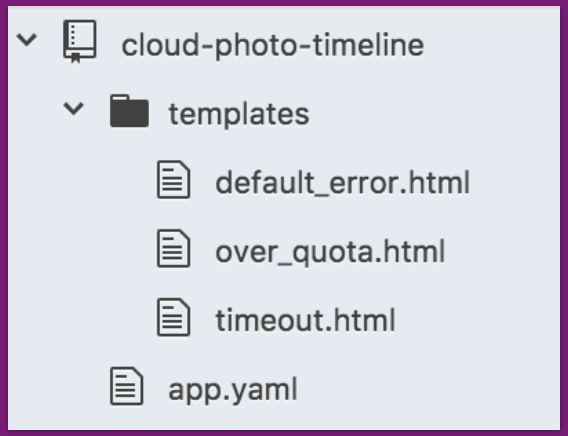
- The results transactions. This can occur because thes ensure that the Instead, they use whatever data is availabl at the time of q ost always less than a few seconds.
- A committed tra might appear to have been applied to some and not others. er appear to have been partially applied wit entity.
- The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

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

YAML MADNESS

YAML MADNESS

- Add custom error handlers to your application
- Update the .yaml file and add the appropriate .html files




```yaml
builtins:
- deferred: on
- appstats: on

# Available throughout the application in the os.environ dictionary
env_variables:
  PROJECT_NAME: 'cloud photo timeline'

error_handlers:
  - file: templates/default_error.html
  - error_code: over_quota
    file: templates/over_quota.html
  - error_code: timeout
    file: templates/timeout.html

# Files that will not be uploaded to app engine
skip_files:
- ^(.*/)?#.*#$
- ^(.*/)?.*~$
- ^(.*/)?.*\.py[co]$
- ^(.*/)?.*/RCS/.*$
- ^(.*/)?\..*$
- ^(.*/)?\.bak$
- ^(.*/)?\.txt$
- ^(.*/)?\.sql3$
```

YAML MADNESS

- env_variables
  - Key values available in os.environ dictionary

```
logging.info(os.environ)
```

```
builtins:
- deferred: on
- appstats: on

# Available throughout the application in the os.environ dictionary
env_variables:
  PROJECT_NAME: 'cloud photo timeline'

error_handlers:
  - file: templates/default_error.html
  - error_code: over_quota
    file: templates/over_quota.html
  - error_code: timeout
    file: templates/timeout.html


# Files that will not be uploaded to app engine
skip_files:
- ^(.*/)?#.*#$
- ^(.*/)?.*~$
- ^(.*/)?.*\.py[co]$
- ^(.*/)?.*/RCS/.*$
- ^(.*/)?\..*$
- ^(.*/)?\.bak$
- ^(.*/)?\.txt$
- ^(.*/)?\.sql3$
```

YAML MADNESS

```
logging.info(os.environ)
```

```
{'USER_NICKNAME': '', 'APPLICATION_ID': 'dev~None', 'wsgi.multiprocess': True,
'HTTP_COOKIE': '_ga=GA1.1.243577631.1504709156', 'SERVER_PROTOCOL': 'HTTP/1.1',
'SERVER_NAME': 'localhost', 'HTTPS': 'off', 'USER_EMAIL': '', 'INSTANCE_ID':
'3ab1534cc00d24e4c4a3b569eeb0cf7e5ebe', 'CURRENT_MODULE_ID': 'default', 'DATACENTER':
'us1', 'SERVER_PORT': '8080', 'HTTP_ACCEPT': 'text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8', 'APPENGINE_RUNTIME': 'python27', 'REQUEST_METHOD': 'GET',
'USER_ORGANIZATION': '', 'PROJECT_NAME': 'cloud photo timeline', 'REQUEST_ID_HASH':
'E9E4D4BC', 'QUERY_STRING': '', 'wsgi.errors':
<google.appengine.api.logservice.logservice._LogsStreamBuffer object at 0x107ca4d10>,
'HTTP_HOST': 'localhost:8080', 'SCRIPT_NAME': '', 'HTTP_USER_AGENT': 'Mozilla/5.0
(Macintosh; Intel Mac OS X 10_13) AppleWebKit/604.1.38 (KHTML, like Gecko) Version/11.0
Safari/604.1.38', 'SERVER_SOFTWARE': 'Development/2.0', 'wsgi.input': <cStringIO.StringI
object at 0x107c7e580>, 'HTTP_ACCEPT_LANGUAGE': 'en-us', 'wsgi.url_scheme': 'http',
'CURRENT_VERSION_ID': 'None.4802049877505O1526', 'TZ': 'UTC', 'REQUEST_LOG_ID':
'8eac5ecfee9a63770e5664a8f73ab1a3f2da483bf8abbea765bfca5f9b1a87f96a8ed34ac21f749ac9953ef488
f', 'wsgi.version': (1, 0), 'PATH_INFO': '/userdf/', 'USER_ID': '',
'HTTP_UPGRADE_INSECURE_REQUESTS': '1', 'wsgi.multithread': True, 'wsgi.run_once': False,
'REMOTE_ADDR': '::1', 'AUTH_DOMAIN': 'gmail.com', 'USER_IS_ADMIN': '0',
'HTTP_X_APPENGINE_COUNTRY': 'ZZ', 'DEFAULT_VERSION_HOSTNAME': 'localhost:8080'}
```

QUERY CURSORS

QUERY CURSORS

- Cursors are a NDB feature that simplifies results 'pagination'
- Won't want to query/return all results on mobile

The Cursor Class

Contents

The `Cursor` class provides a cursor in the current set search results, allowing you to retrieve the next set based on criteria that you specify. Using cursors improves the performance and consistency of pagination as indexes are updated.

The following shows how to use the cursor to get the next page of results:

```
# Get the first set of results, and return a cursor with that result set.
# The first cursor tells the API to return cursors in the SearchResults object.
results = index.search(search.Query(query_string='some stuff',
    options=search.QueryOptions(cursor=search.Cursor())))

# Get the next set of results with a cursor.
results = index.search(search.Query(query_string='some stuff',
    options=search.QueryOptions(cursor=results.cursor)))
```

If you want to continue search from any one of the `ScoredDocuments` in `SearchResults`, set `Cursor.per_result` to `True`:

```
# get the first set of results, the first cursor is used to specify
# that cursors are to be returned in the SearchResults.
results = index.search(search.Query(query_string='some stuff',
    options=search.QueryOptions(cursor=Cursor(per_result=True))))

# this shows how to access the per_document cursors returned from a search
per_document_cursor = None
for scored_document in results:
    per_document_cursor = scored_document.cursor

# get the next set of results
results = index.search(search.Query(query_string='some stuff',
    options=search.QueryOptions(cursor=per_document_cursor)))
```

QUERY CURSORS

```
from google.appengine.datastore.datastore_query import
Cursor

limit = 5
cursor = self.request.get('cursor')
q = Photo.query().order(-Photo.date)

if cursor == "":
  result,cursor,more = q.fetch_page(limit)
else:
   result,cursor,more = q.fetch_page(
                limit,
                start_cursor= Cursor(urlsafe=cursor)
                )
```

SET UP QUERY AS NORMAL

QUERY CURSORS

```
limit = 5
cursor = self.request.get('cursor')
q = Photo.query().order(-Photo.date)

if cursor == "":
  result,cursor,more = q.fetch_page(limit)
else:
   result,cursor,more = q.fetch_page(
                limit,
                start_cursor= Cursor(urlsafe=cursor)
                )
```

FIIRST QUERY RETURN A TUPLE WITH RESULTS, CURSOR AND BOOL IF THERE ARE MORE RESULTS

QUERY CURSORS

```
limit = 5
cursor = self.request.get('cursor')
q = Photo.query().order(-Photo.date)

if cursor == "":
  result,cursor,more = q.fetch_page(limit
else:
   result,cursor,more = q.fetch_page(
                limit,
                start_cursor= Cursor(urlsafe=cursor)
                )
```

SUBSEQUENT QUERY RETURN A TUPLE WITH RESULTS, CURSOR AND BOOL IF THERE ARE MORE RESULTS

QUERY CURSORS

```
limit = 5
cursor = self.request.get('cursor')
q = Photo.query().order(-Photo.date)

if cursor ==
  result,curs
else:
   result,curs
                limit,
                start_cursor= Cursor(urlsafe=cursor)
                )
```

ALSO END_CURSOR PARAMETER FOR MOVING BACKWARDS

QUERY CURSORS

```
    def web_results(self,photos,user,cursor):
        """Return html formatted from the datastore query.  Take an optional
        curson parameter for pagination.
        """
        limit = 5
        cursor = self.request.get('cursor')
        q = Photo.query().order(-Photo.date)

        # Even if the cursor was None, it will be passed around as an empty
        # string since it is coming from HTTP request
        if cursor == "":
            result,cursor,more = q.fetch_page(limit)
        else:
            result,cursor,more = q.fetch_page(limit,start_cursor= Cursor(urlsafe=cursor))

        # Write HTML to display onscreen
        html = ""
        for photo in result:
            html += '<div><hr><div><img src="/image/%s/" width="200" border="1"/></div>' % photo.key.urlsafe()
            html += '<div><blockquote>Caption: %s<br>User: %s<br>Date:%s</blockquote></div></div>' % (cgi.escape(photo.caption),user,str(photo.date))
        html += "<a href='/user/%s/web/?id_token=123&cursor=%s'>Next with cursor: %s</a>" % (user,cursor.urlsafe(),cursor.urlsafe())
        return html
```

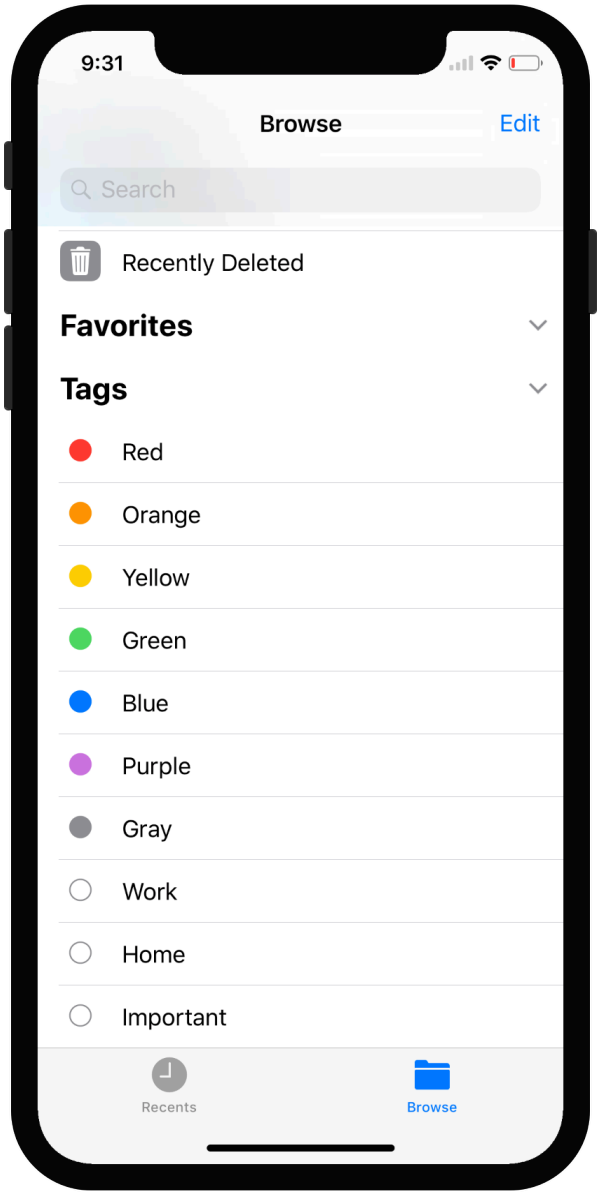
- Limit result to 5 and use cursor to paginate web results

QUERY CURSORS

MOBILE CURSOR STRATEGY #1

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

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

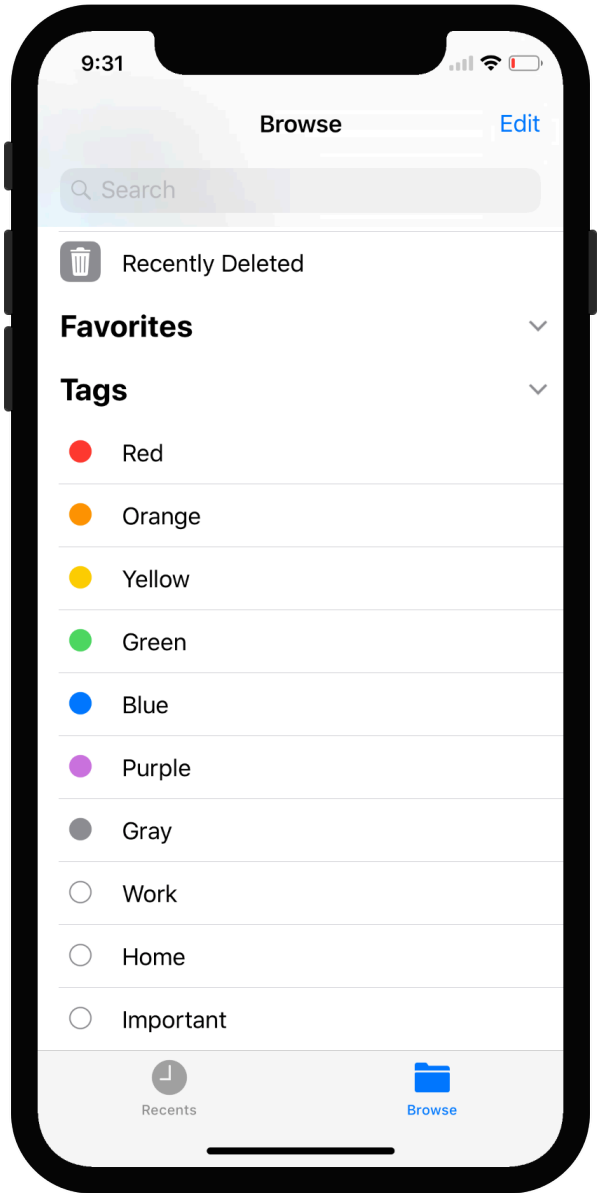


QUERY CURSORS

MOBILE CURSOR STRATEGY #1

- Add one more table view cell; monitor

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

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

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

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


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

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

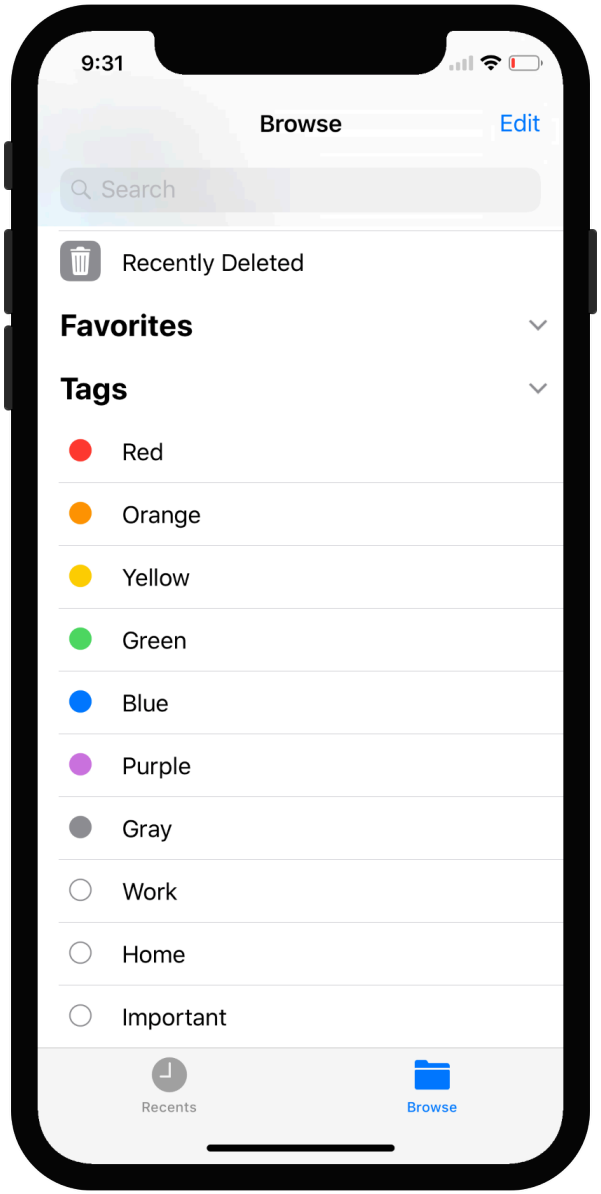


QUERY CURSORS

MOBILE CURSOR STRATEGY #3

- Monitor scroll view

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

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

                }
            }


    }
```



QUERY CURSORS

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

Limitations of cursors

Cursors are subject to the following limitations:

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

Cursors and data updates

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

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

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 singl
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()
```

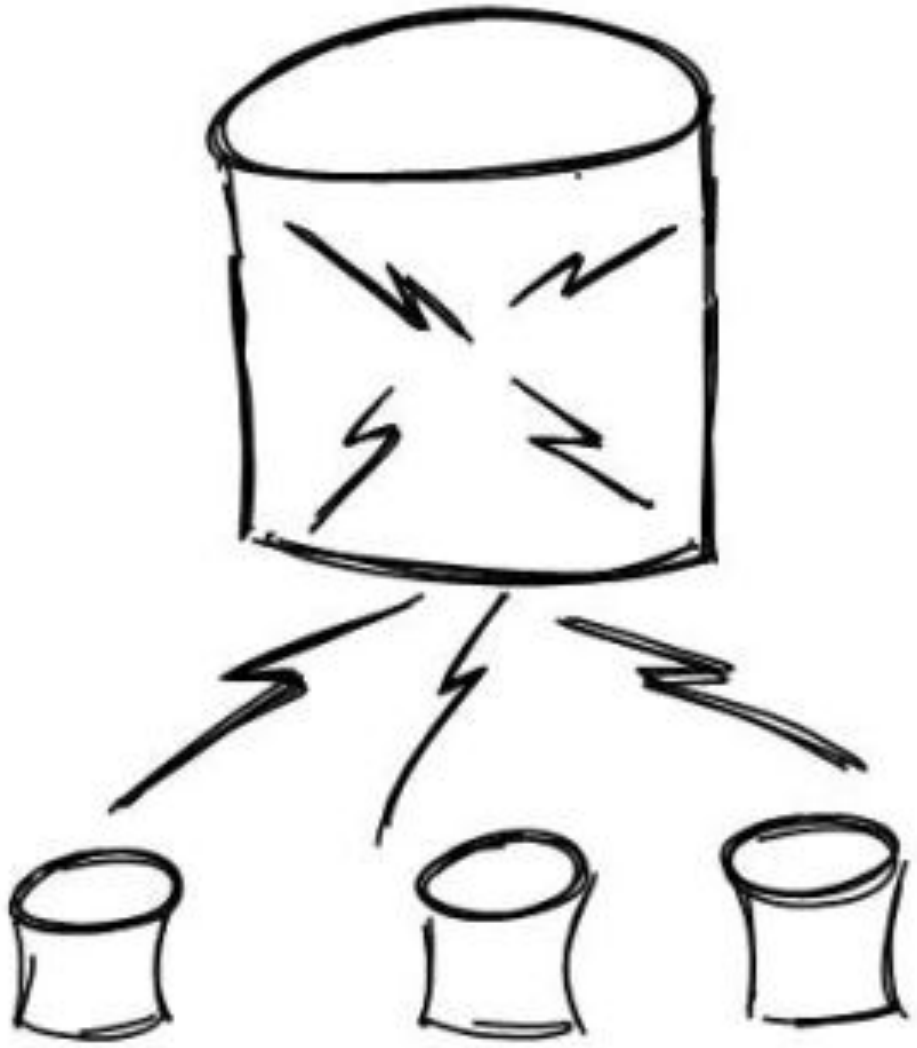
If you had a single entity that was the counter and the update rate was too fast, then you would
contention as the serialized writes would stack up and start to timeout. The way to solve this pr
little counter-intuitive if you are coming from a relational database; the solution relies on the fac
from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

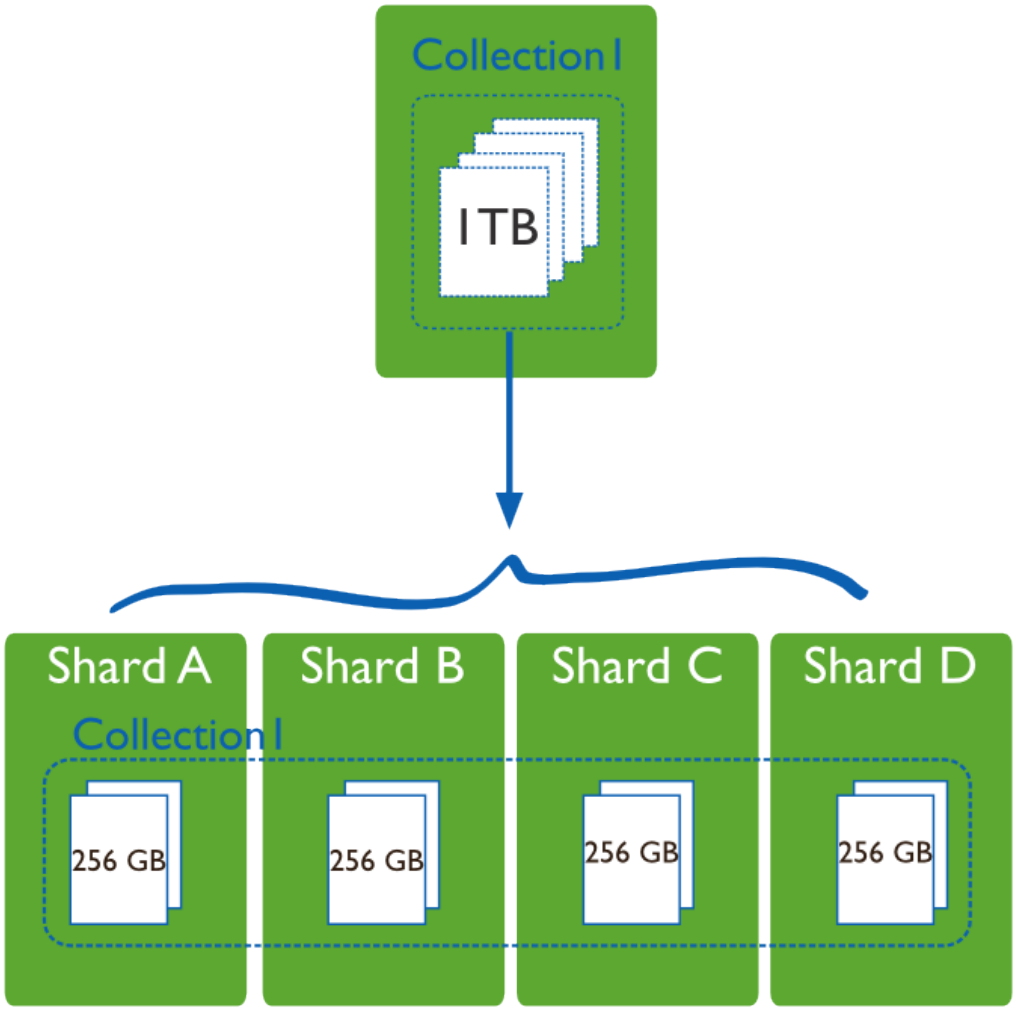
SHARING

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



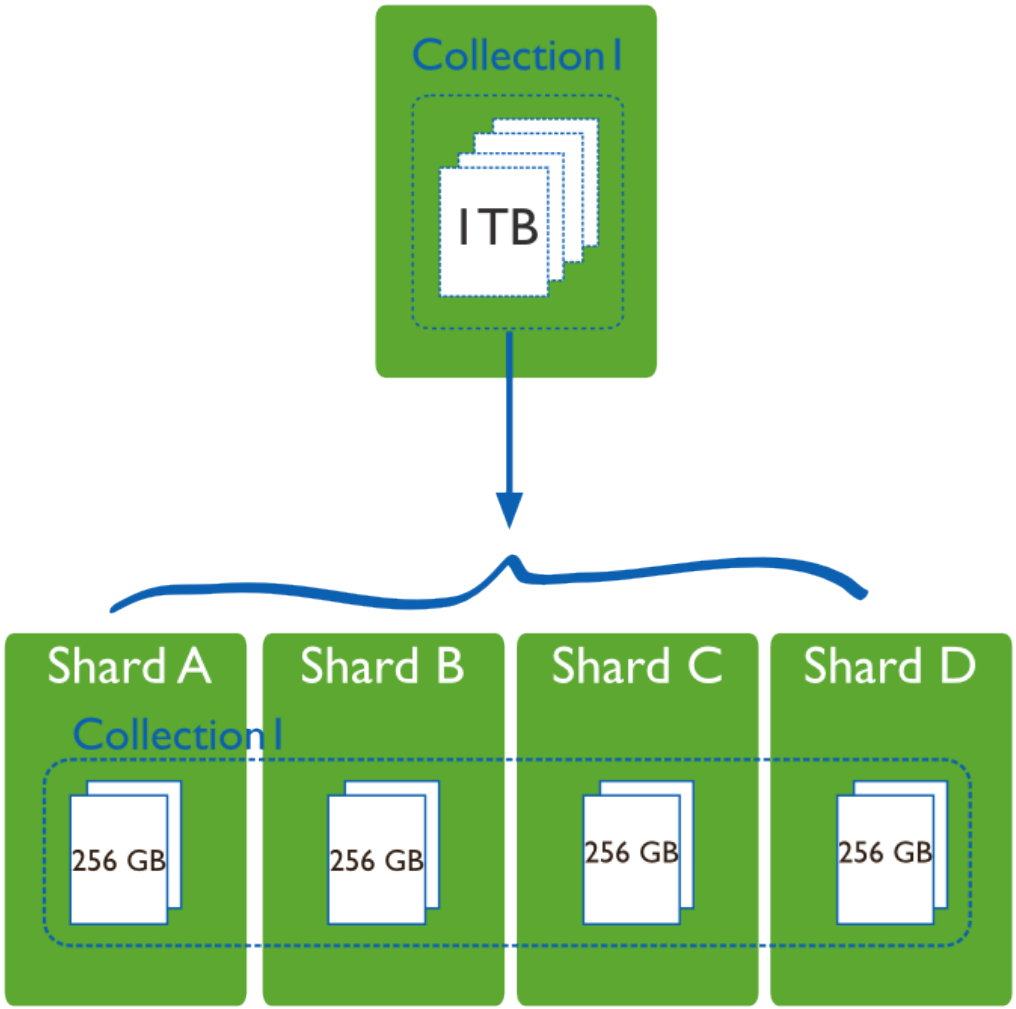
SHARING

- 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

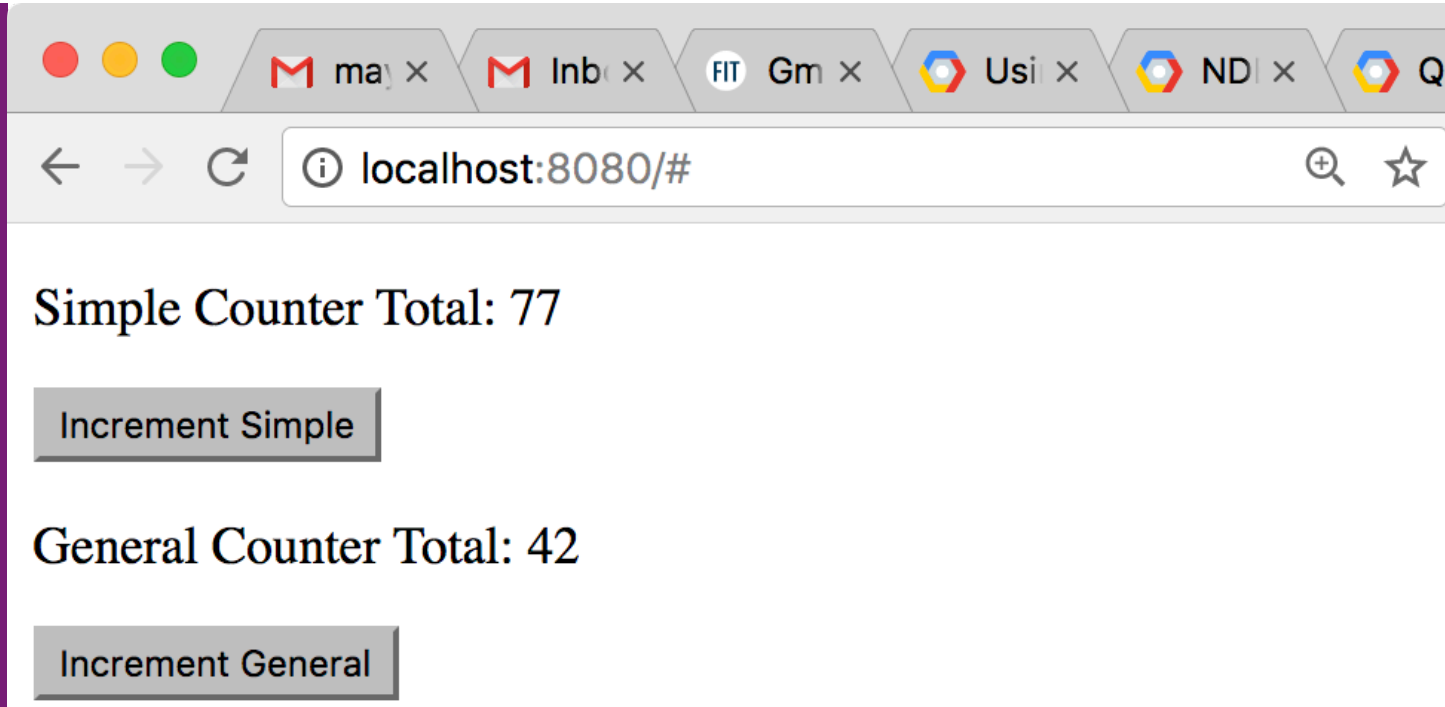
SHARDING

```
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

- Increment faster than rate limit



SHARDING

- Increment faster than rate limit by spreading over 20 entities

Entity Kind

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

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

Delete | Flush Memcache

SHARDING

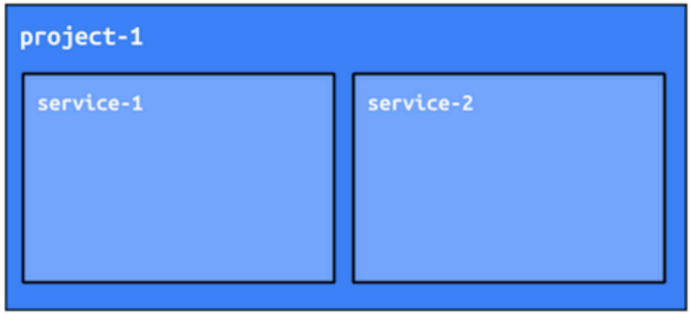
- More general purpose example also given
- Uses memcache to store the total

```
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)
```

MICROSERVICES

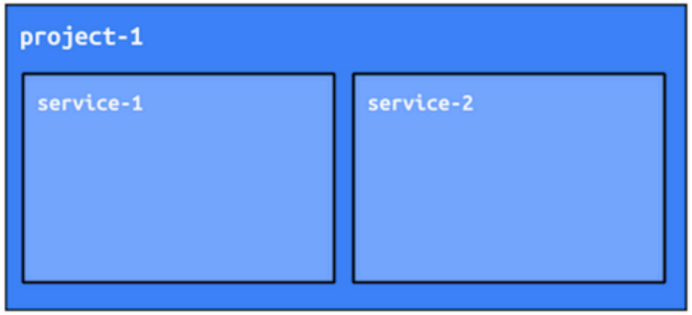
MICROSERVICES

- 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



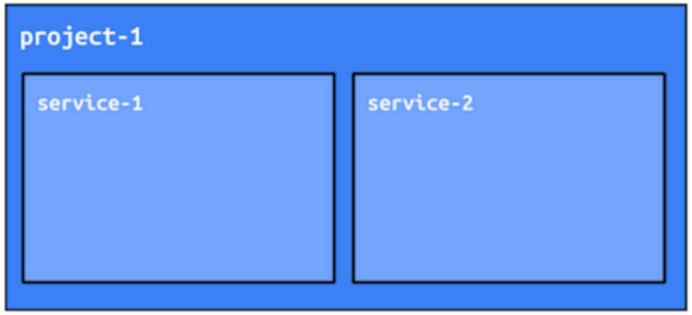
MICROSERVICES

- 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

- To serve a single user or API request, a microservices-based application can call many internal microservices to compose its response
- Some aspects may be more complex to manage, some less complex



MICROSERVICES

- Functionality in our photo-timeline example
  - Post and retrieve images
  - Delete photos
  - User 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

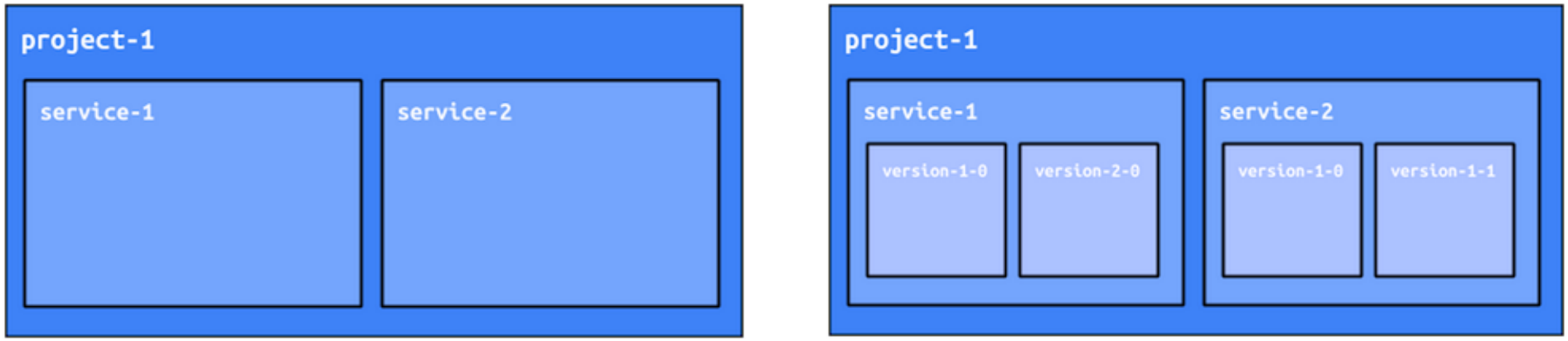
- Manageable now, but what happens when we add new features, tasks, schedule tasks?
- Compartmentalizing the function into logical segments may help
  - And minimize problems (potentially)

```
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

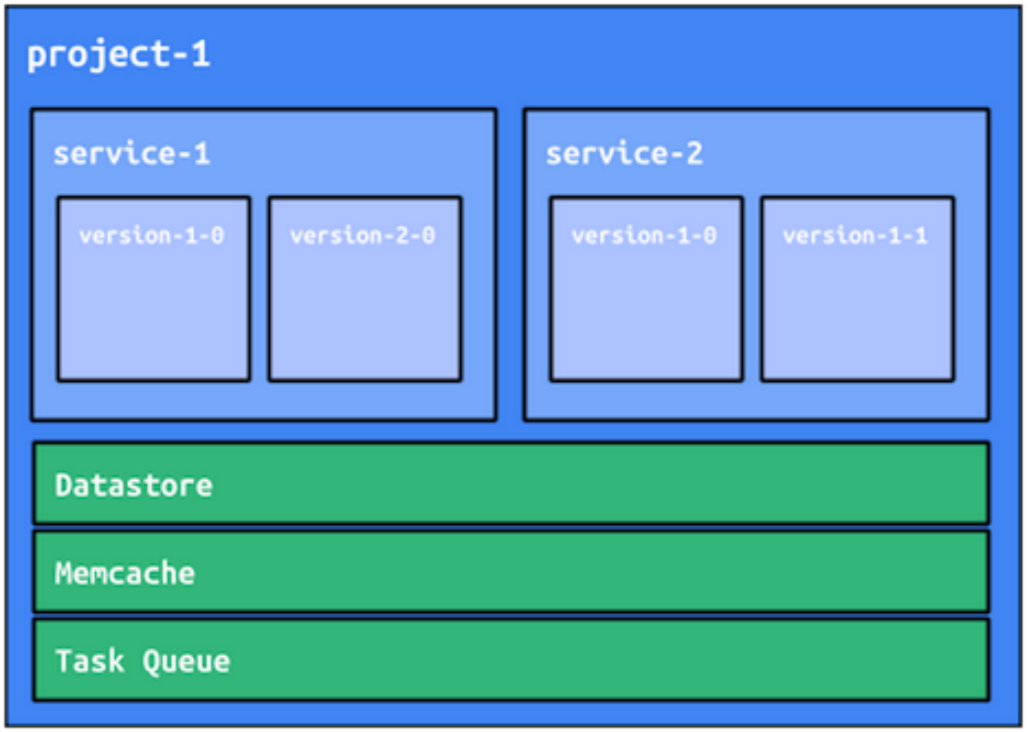
MICROSERVICES



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

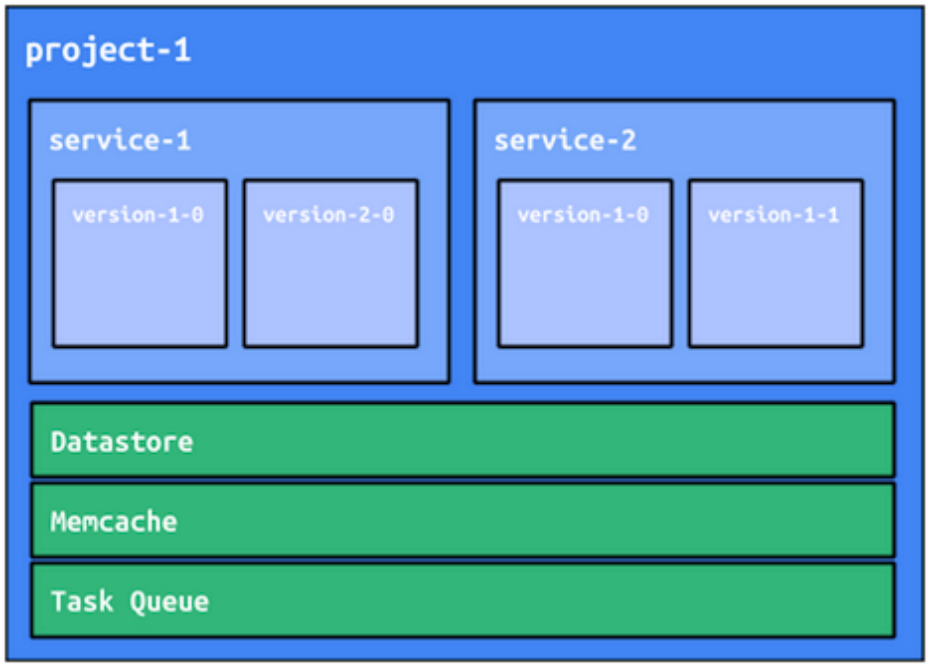
MICROSERVICES

- Some services share 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



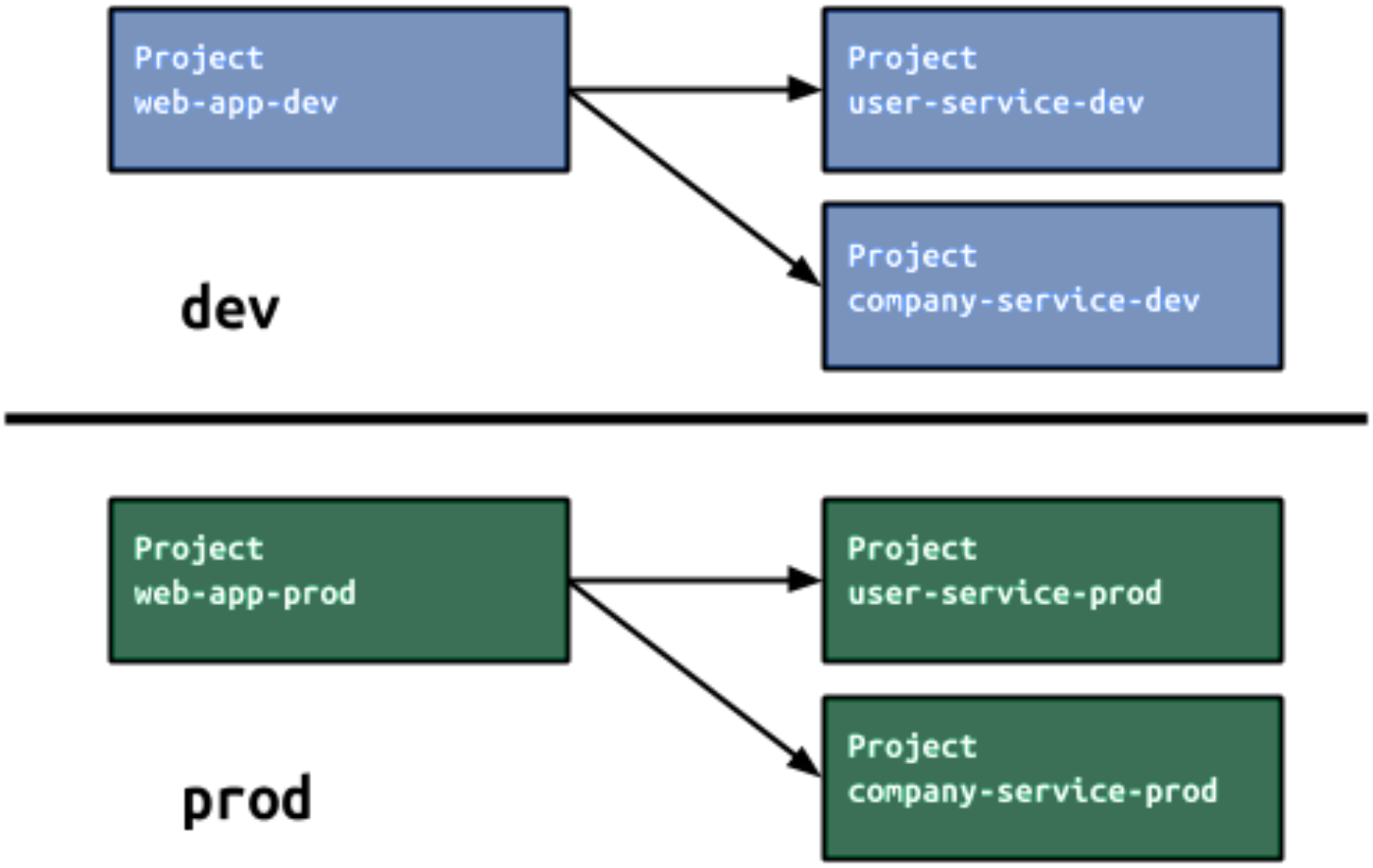
MICROSERVICES

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

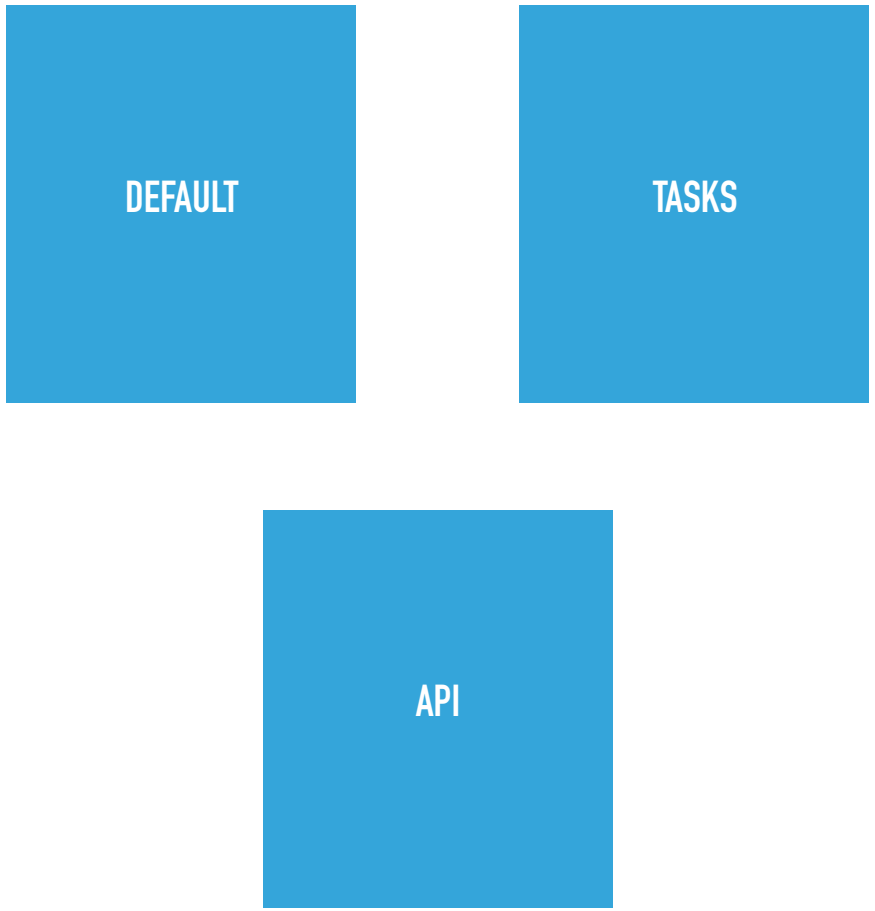
MIGRATING TO MICROSERVICES

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

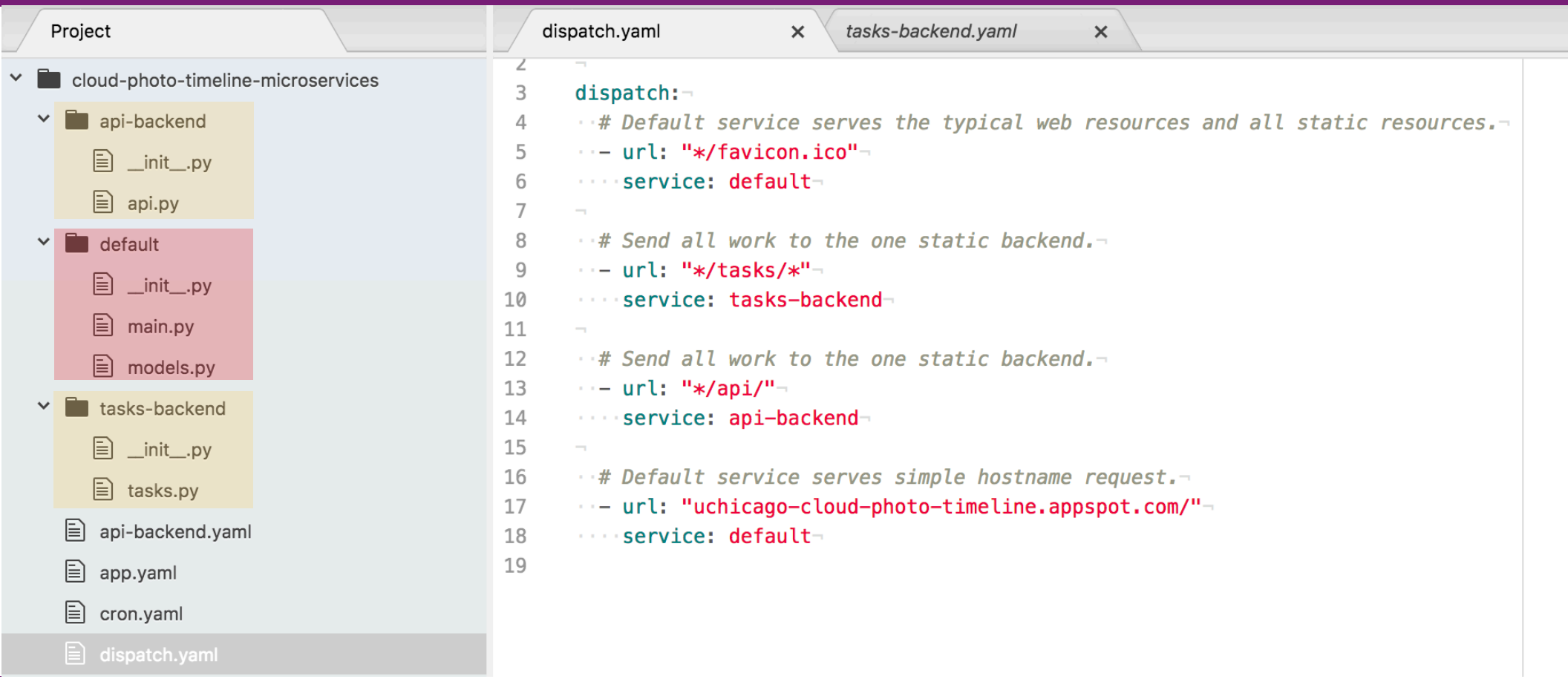
```
app = webapp2.WSGIApplication([
    ('/', HomeHandler),
    ('/email_task/', EmailTaskHandler),
    ('/tasks/summary_email/', CronTasksSummaryEmail),
    webapp2.Route('/logging/', handler=LoggingHandler),
    webapp2.Route('/image/<key>/', handler=ImageHandler),
    webapp2.Route('/post/<user>/',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

MIGRATING TO MICROSERVICES

- Our application has 3 logical separations
  - Default (web)
  - API
  - TASKS



MIGRATING TO MICROSERVICES



MIGRATING TO MICROSERVICES

- Each service needs
  - a .yaml file
  - Be listed in the dispatch.yaml file
  - Have a webapp2 application

```
dispatch:
  # Default service serves the typical web resources and all
  - 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
```

APP WAS DEFAULT

MIGRATING TO MICROSERVICES

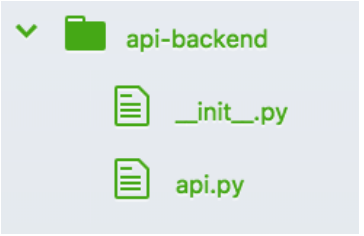
```
  service: default

# 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
```

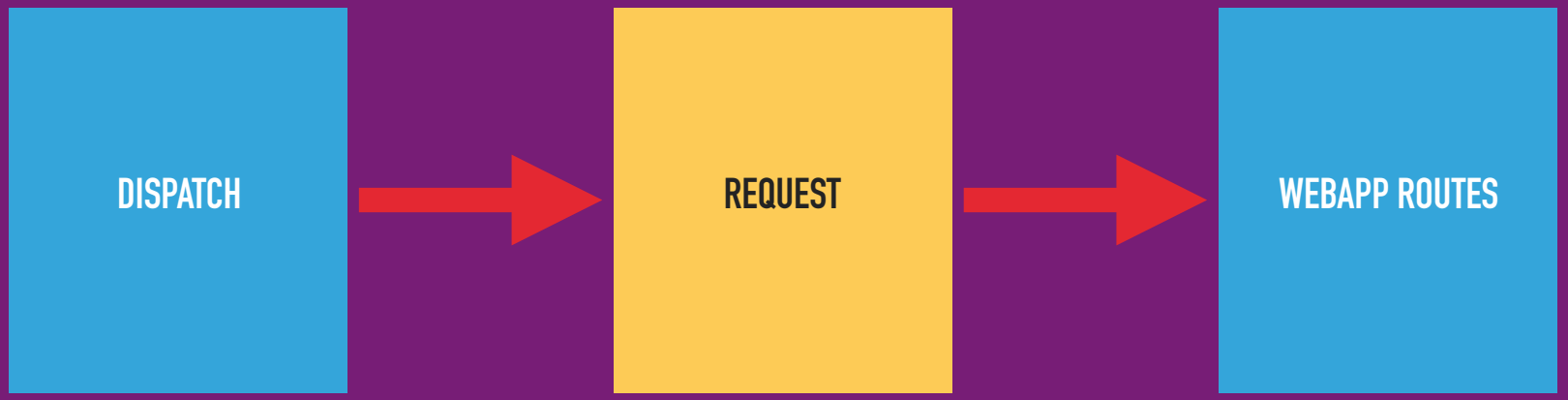


MIGRATING TO MICROSERVICES

- 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



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

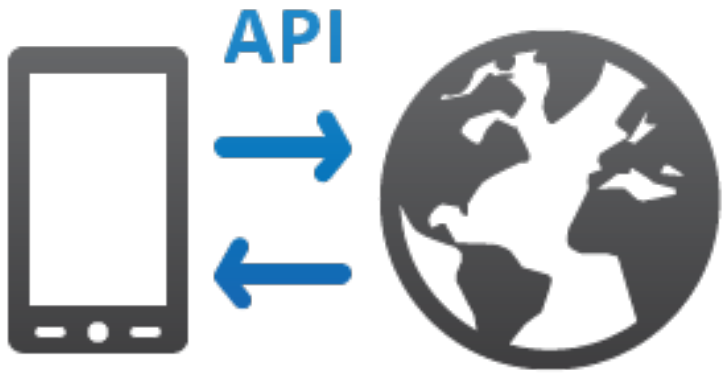
```
cron:
- description: daily summary job
  url: /summary_email/
  target: tasks-backend
  schedule: every 1 minutes
```

BREAK TIME 

API DESIGN

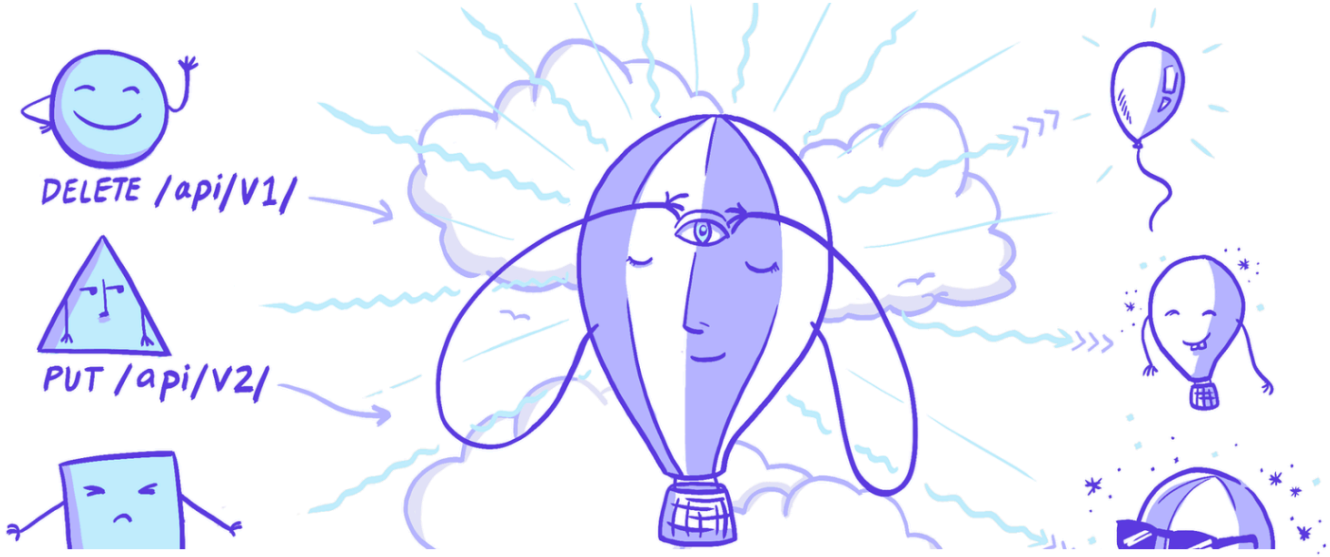
API DESIGN

- 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



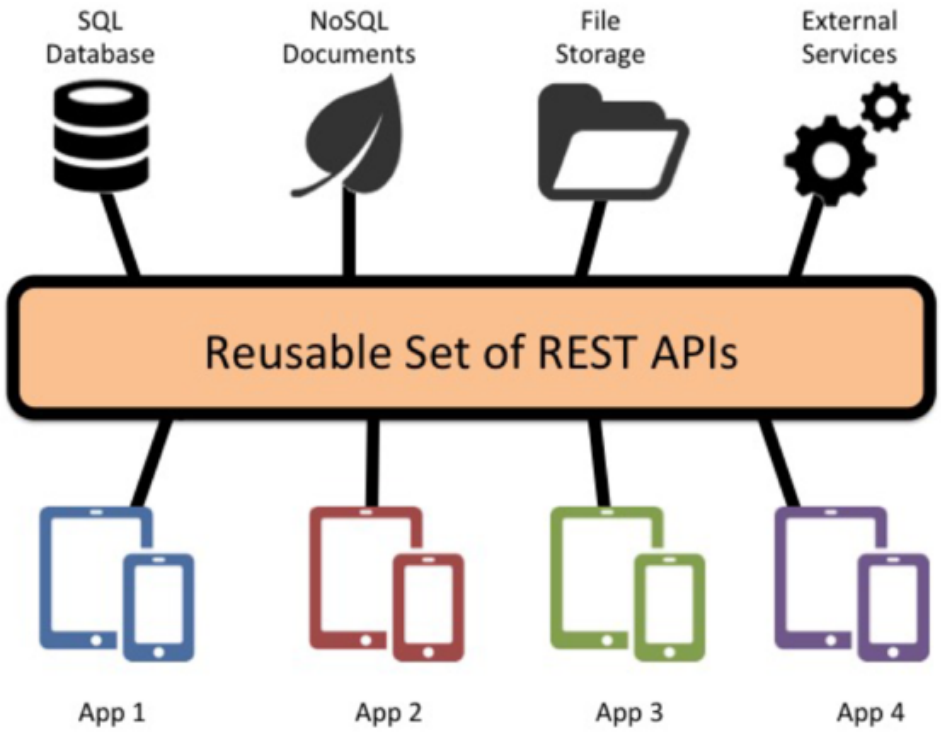
API DESIGN

- 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

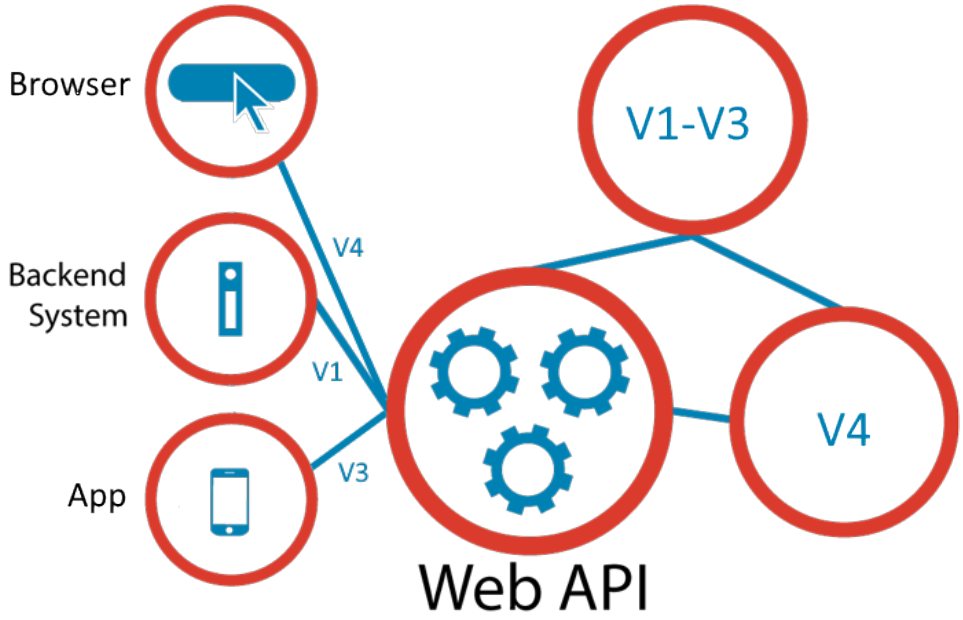


API DESIGN

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

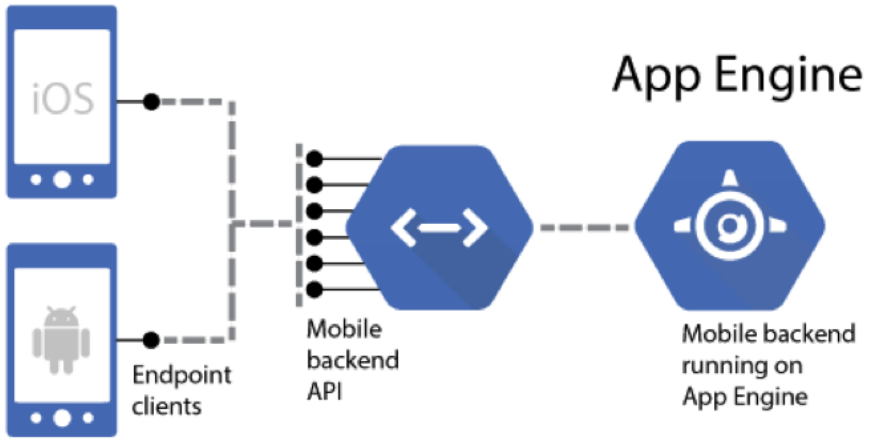
API DESIGN

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



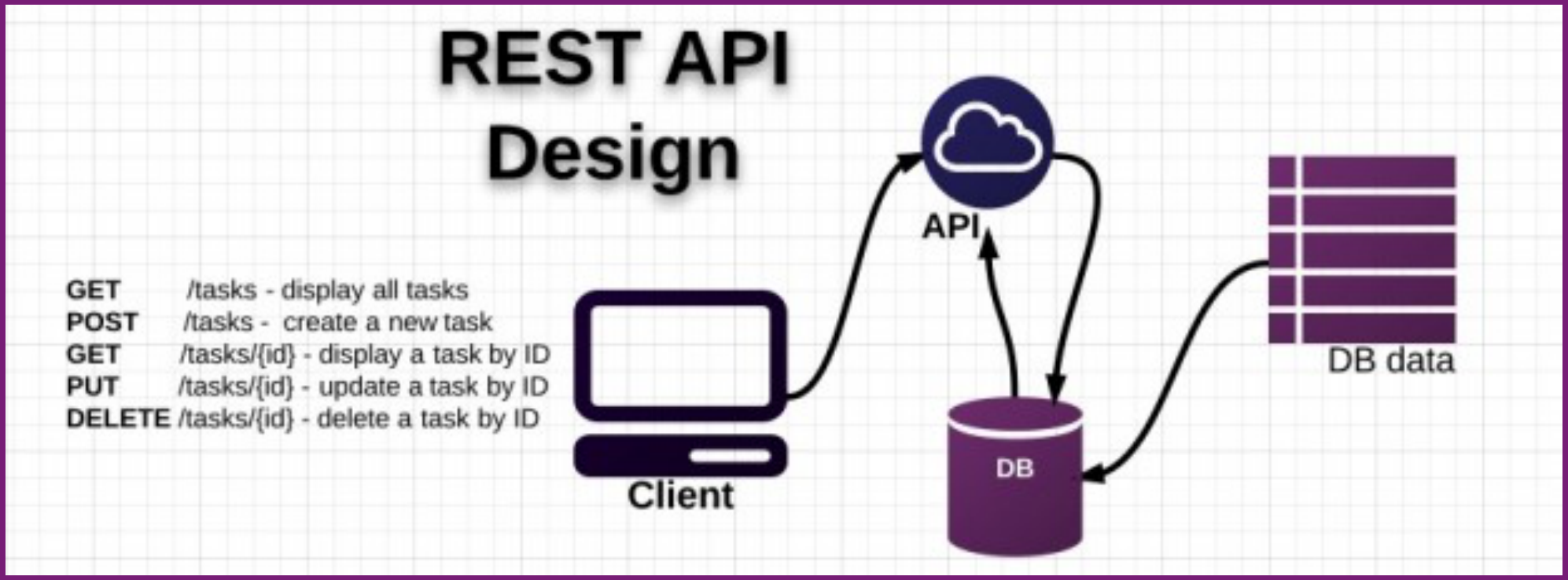
API DESIGN

- 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)

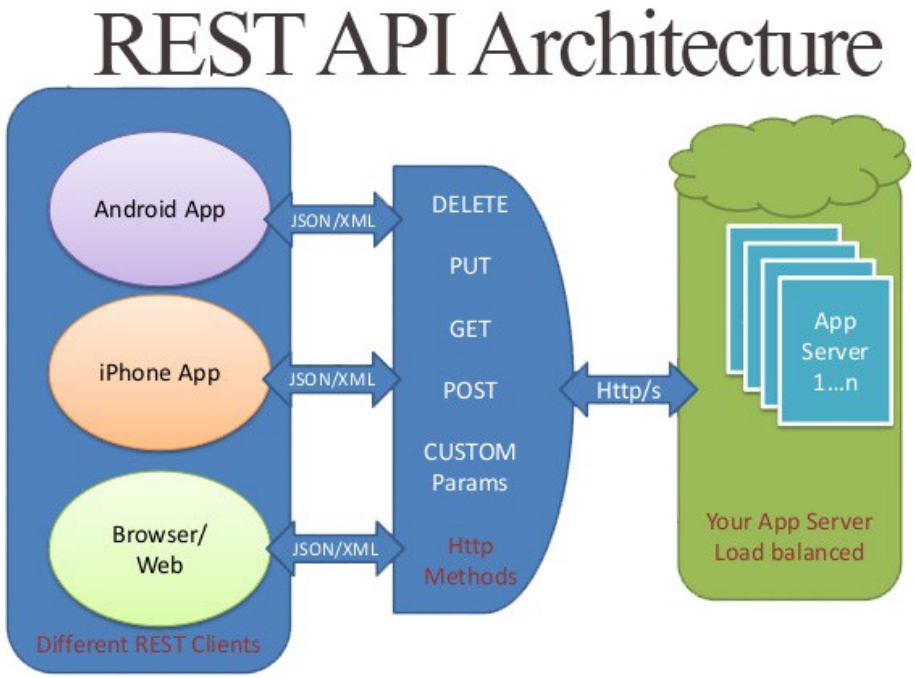
REST



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

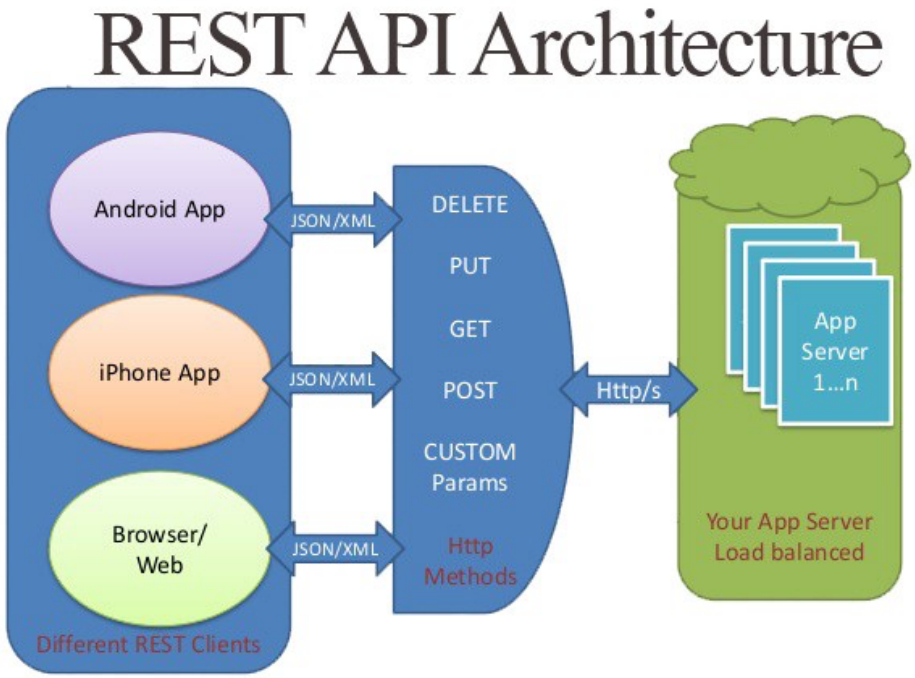
REST

- 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



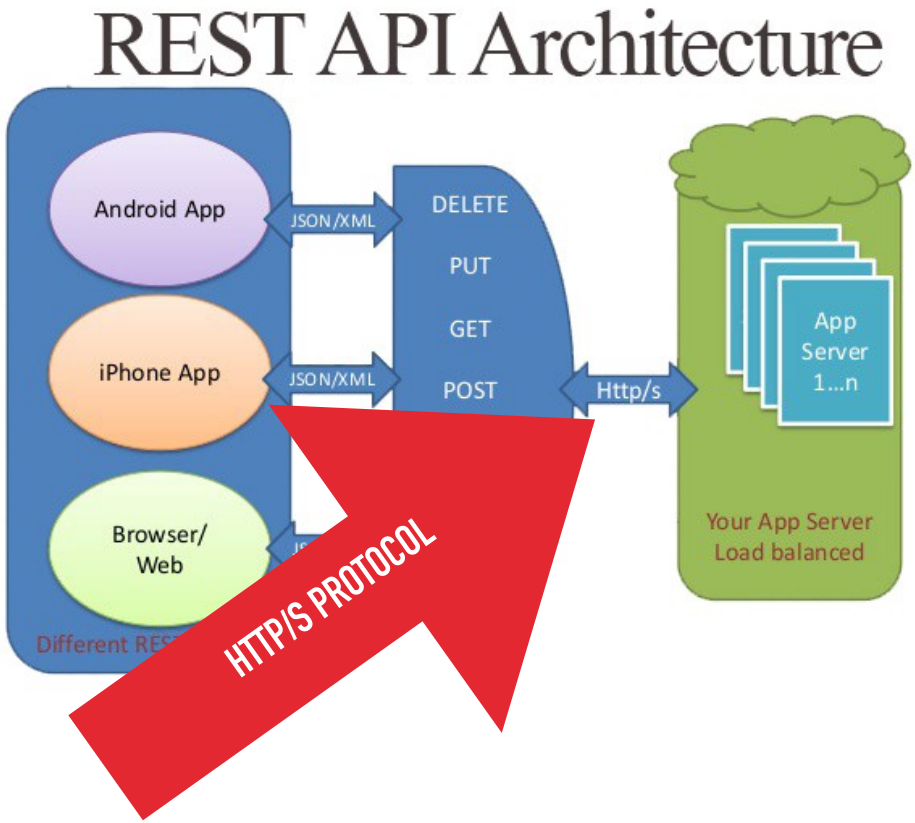
REST

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



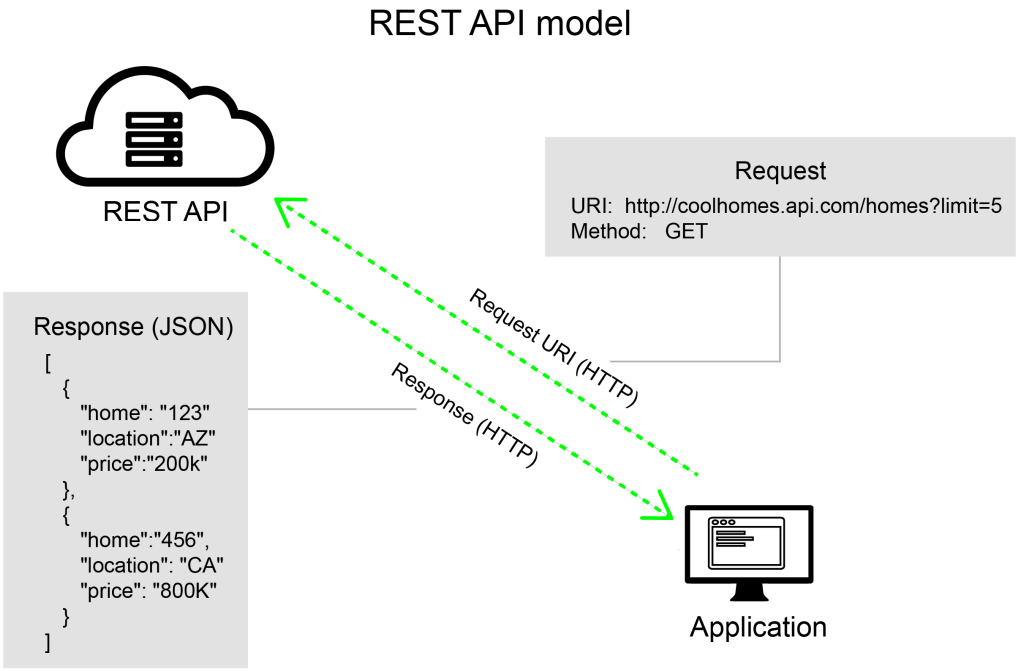
REST

- 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



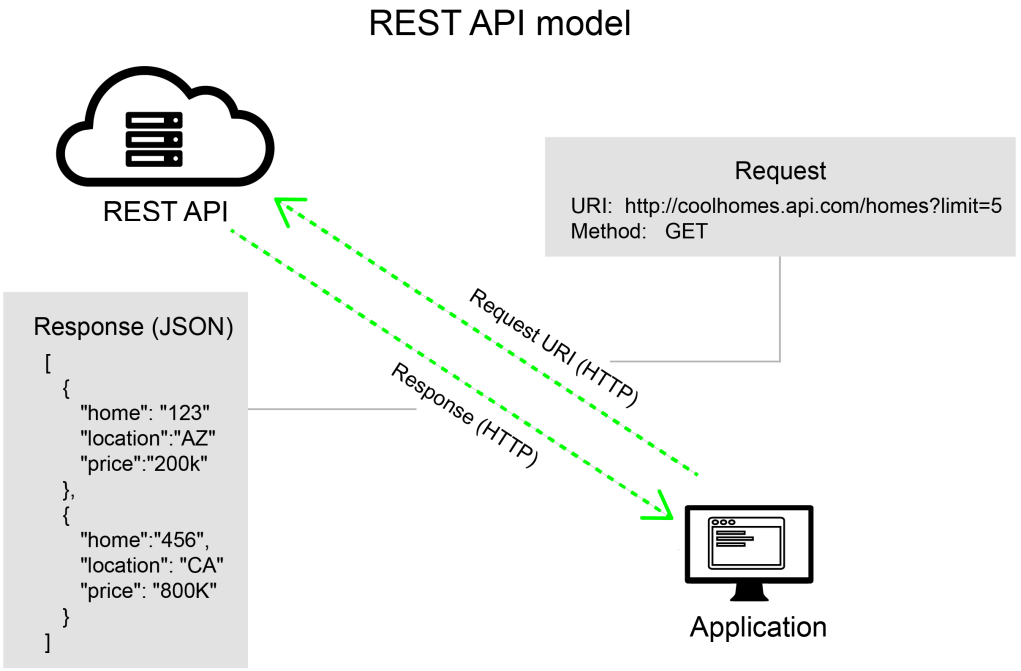
REST

- 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



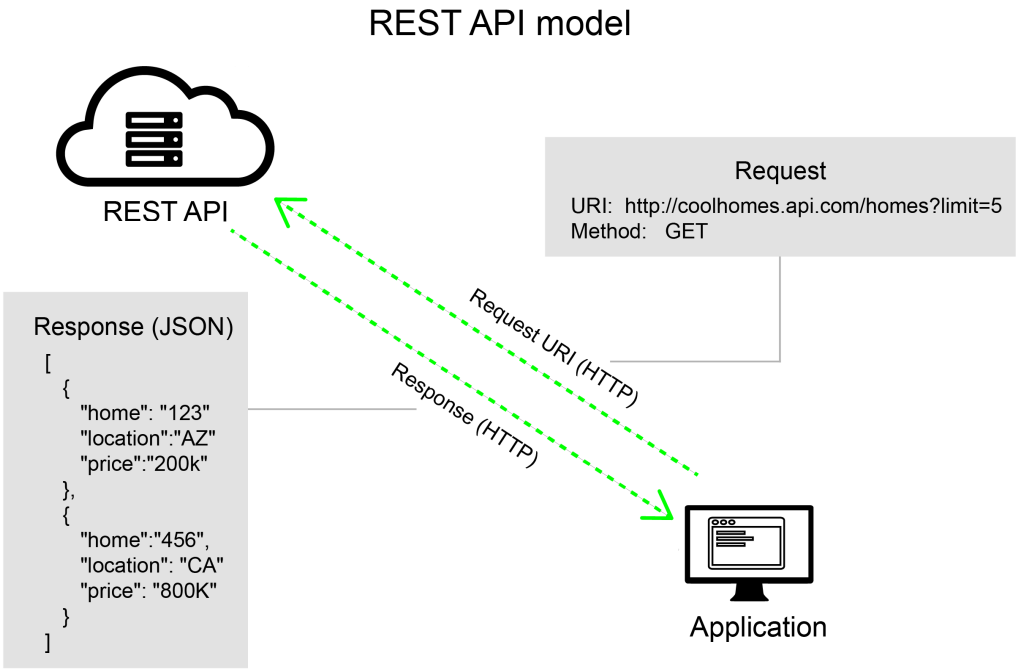
REST

- 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]

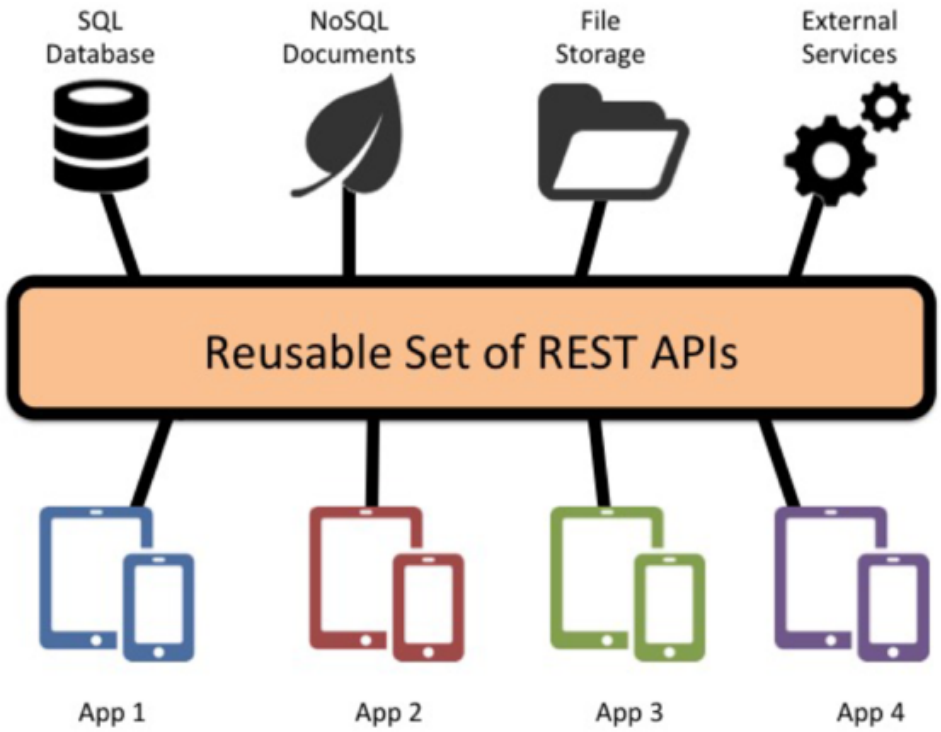
405 Method Not Allowed

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

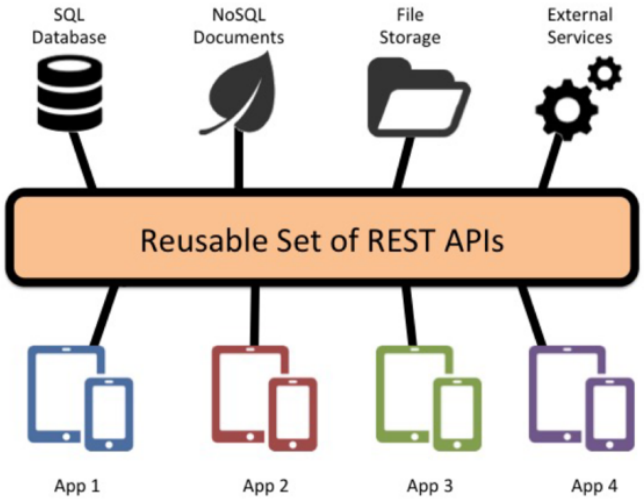
DESIGN OF A REST API

- 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



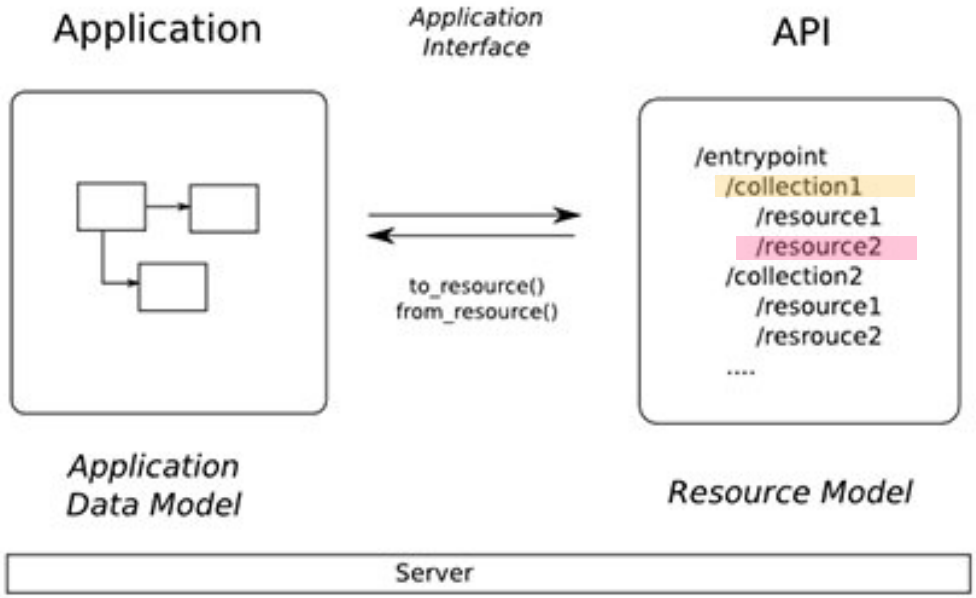
DESIGN OF A REST API

- Different than an API for classes in OOP of an API
  - 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
```

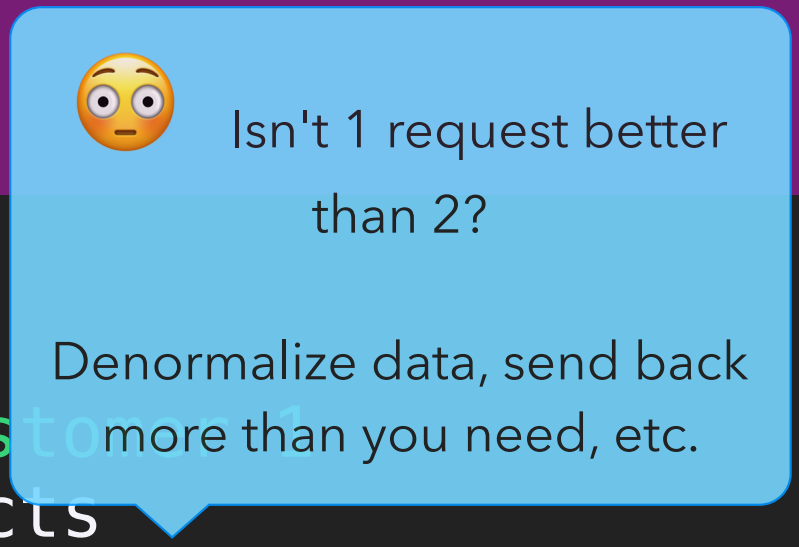
Do we want the customer information or the order information?

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 order
/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)`

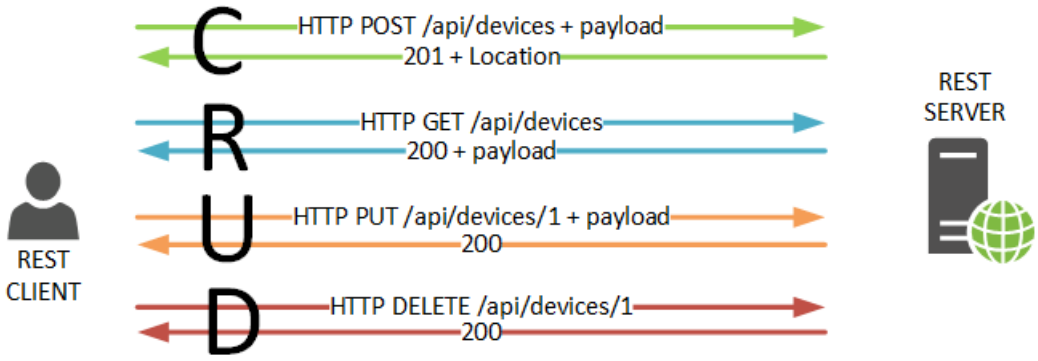
Called to handle an HTTP DELETE request. Overridden by handler subclasses.

`trace(*args)`

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

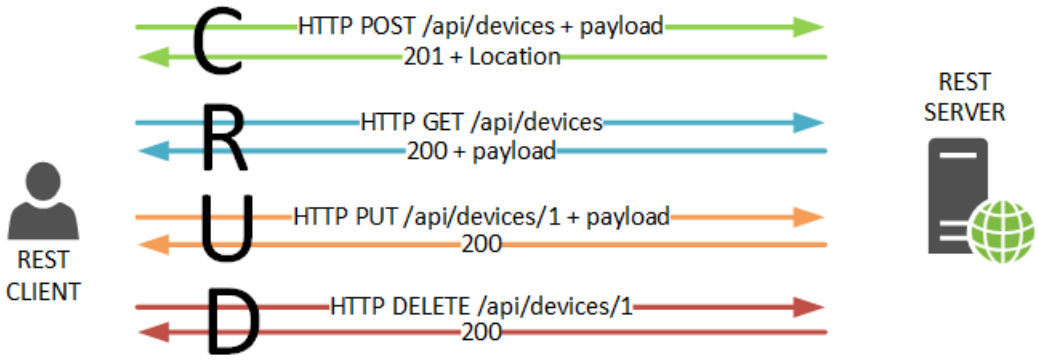
OPERATIONS IN TERMS OF HTTP METHODS

- 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



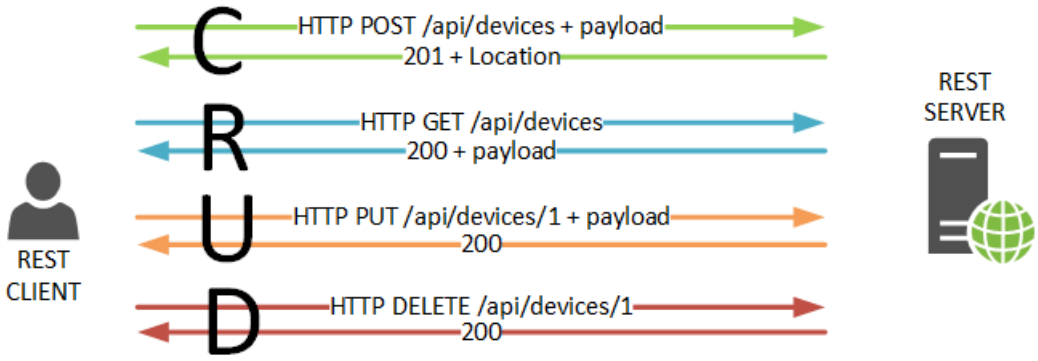
OPERATIONS IN TERMS OF HTTP METHODS

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



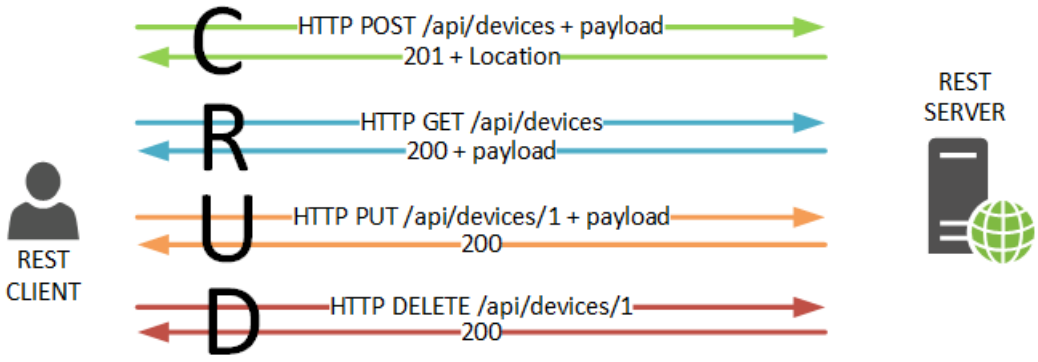
OPERATIONS IN TERMS OF HTTP METHODS

- 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

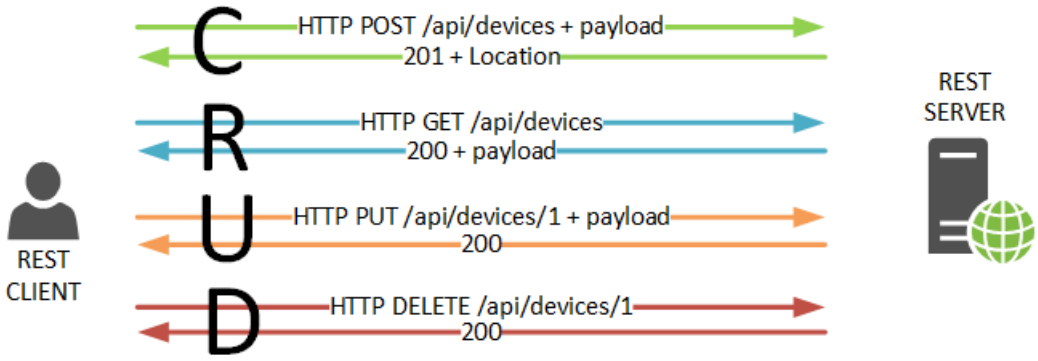
Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers (*if implemented*)	Remove all customers

```
# 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

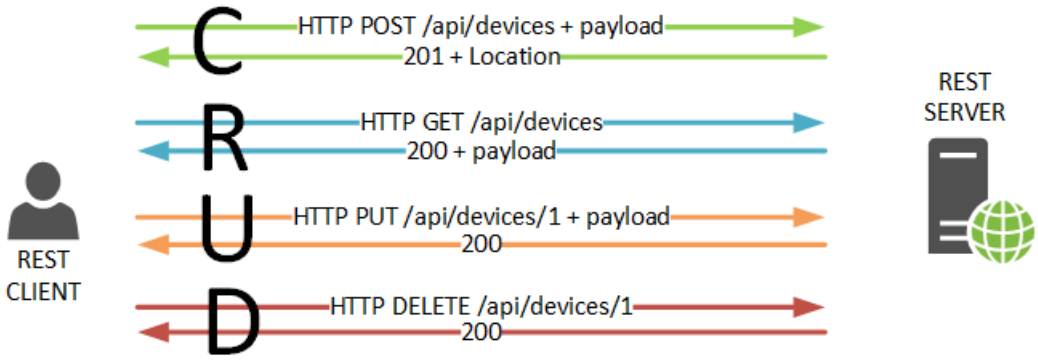
POST VS. PUT



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}
```

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
```

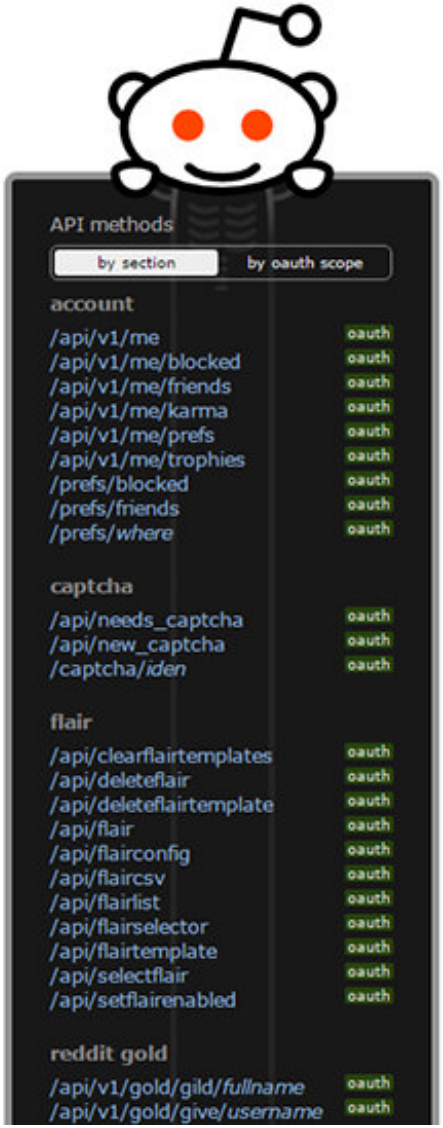
Alternative 200 (OK) 202 (Accepted)

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.
`DELETE`	Used for deleting resources.

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 playlist.
`list`	**`GET /playlistItems`**	Returns a collection of playlist items that match the API request parameters. You can retrieve all of the playlist items in a specified playlist or retrieve one or more playlist items by their unique IDs.
`update`	**`PUT /playlistItems`**	Modifies a playlist item. For example, you could update the item's position in the playlist.

SUMMARY

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

- "It depends...."

ASSIGNMENT 2

ASSIGNMENT 2

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

Assignment 2

As discussed in class, there are some potential issues with our photo timeline backend. In the fist part of the assignment you will update our existing code base (available here on Github) with the following changes.

Update the Data Model

THE UNIVERSITY OF CHICAGO

MPCS 51033 • AUTUMN 2017 • SESSION 3

BACKENDS FOR MOBILE APPLICATIONS