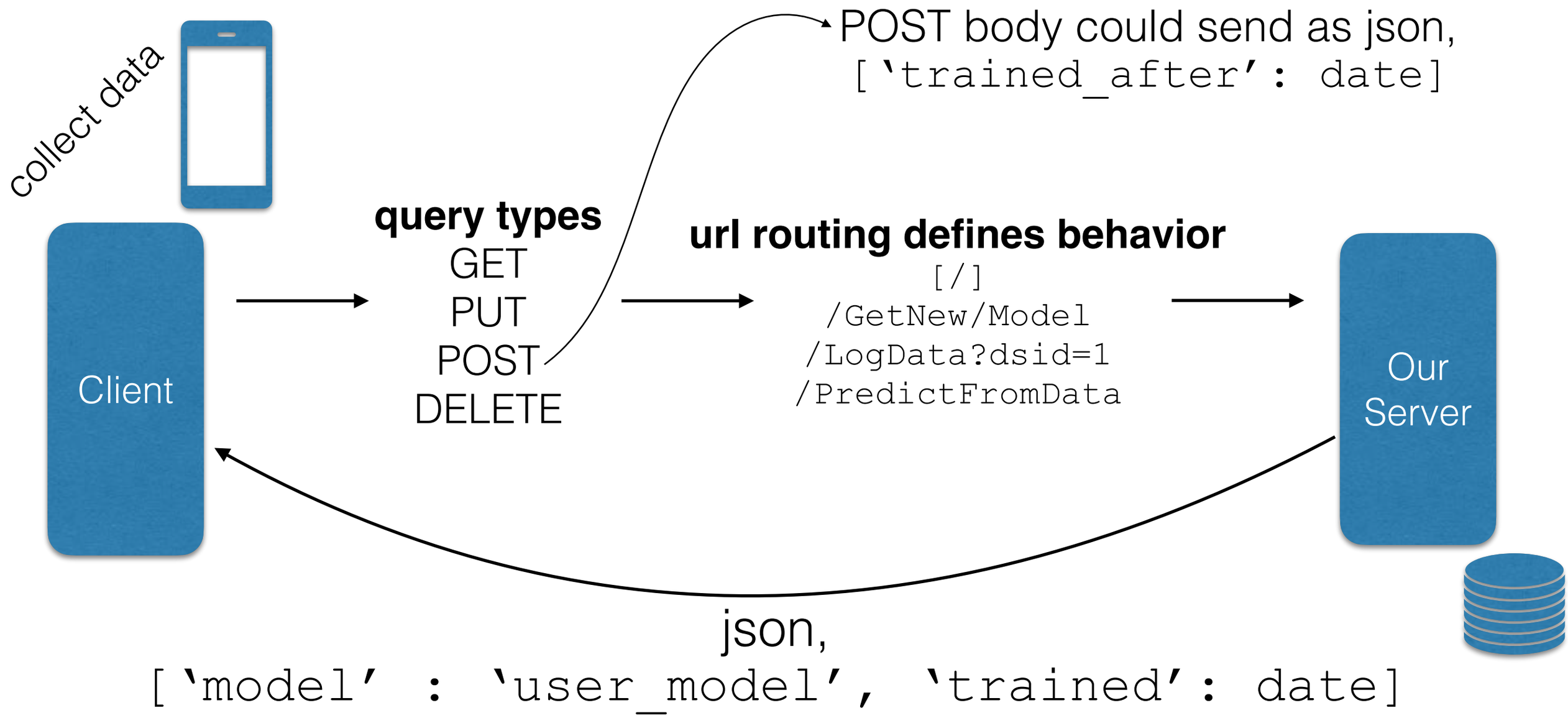
GET: Retrieves data from the server. No other effect.
PUT: Replaces target resource with the request payload. Update or create a new resource.
POST: Performs *resource-specific processing* on the payload. Can be used for different actions including creating a new resource, uploading a file, or training an ML model.
DELETE: Removes data from the server.

•Specifies a design for how we can interact remotely with a server to post and query data. REST does one thing at a time… We will abuse this a little in our implementation.

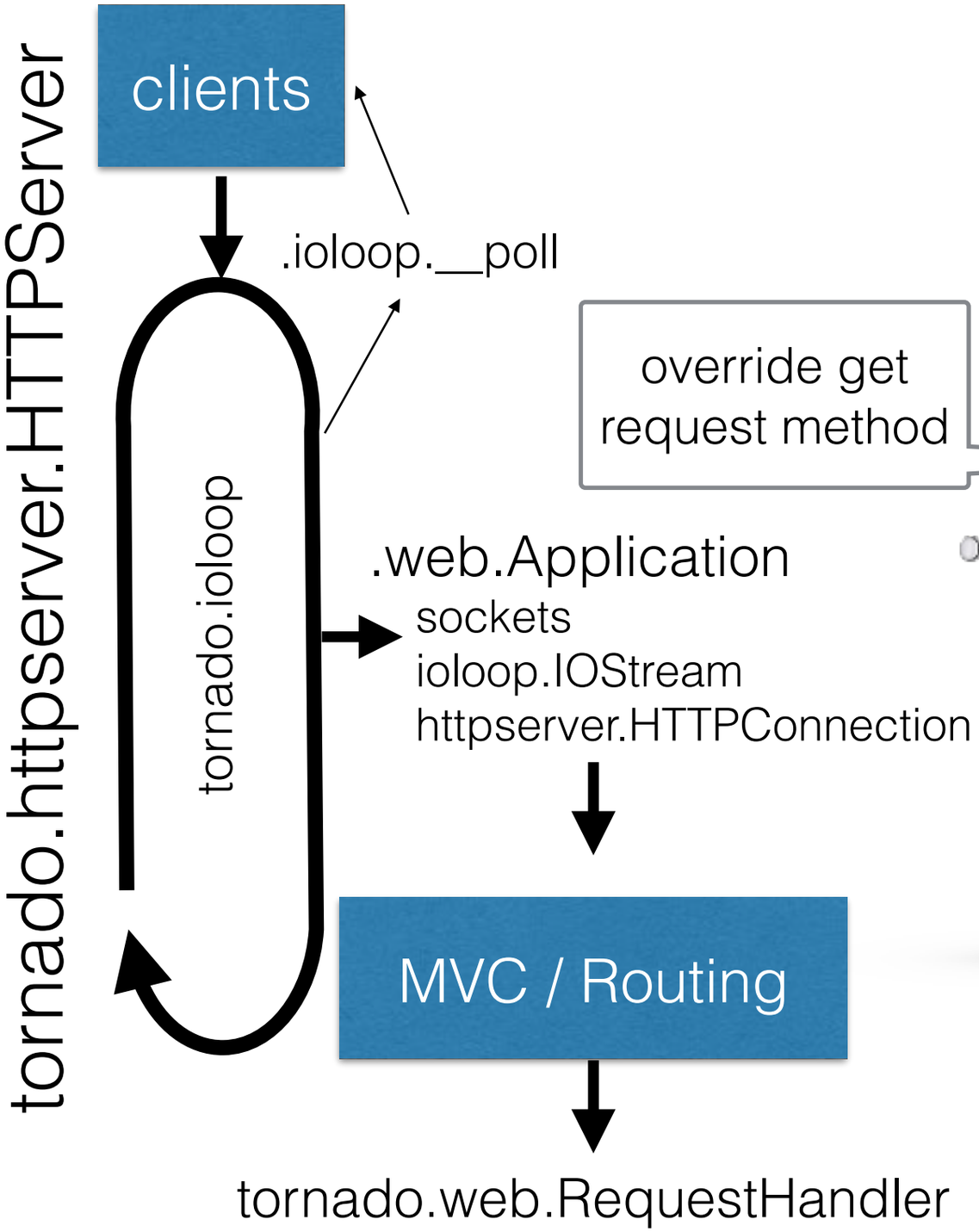


tornado web

https://www.tornadoweb.org/en/stable/

- non-blocking web server
  - built for short-lived requests (pipelined)
  - and long lived connections
- built to scale
  - an attempt to solve the 10k concurrent problem
- has a python wrapper implementation
  - open sourced by Facebook after acquiring friendfeed.com
  - originally developed by the developers of gmail and google maps (the original releases)
- uses IOLoop and callback model

tornado




```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, MSLC World")

application = tornado.web.Application([
    (r"/main", MainHandler),
])


application.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

new class, inherit from RequestHandler

override get request method

URL routing and handler list

start the IO loop

listen on 8888

tornado: get request

- get requests with arguments

```
class GetExampleHandler(tornado.web.RequestHandler):
   def get(self):
      arg = self.get_argument("arg", None, True) # get arg
      if arg is None:
         self.write("No 'arg' in query")
      else:
         self.write(str(arg)) # spit back out the argument
```

- how many connections?
  - one front end of Tornado~3,000 in 1 second
  - with nginx and four instances of tornado
    - anywhere from 9,000-17,000
  - caveat: as long as you do not block the thread!

sub-classing application

```
# tornado imports
import tornado.web
from tornado.web import HTTPError
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from tornado.options import define, options

# Setup information for tornado class
define("port", default=8000,
       help="run on the given port", type=int)

CUSTOM CLASSES AND DEFINITIONS

def main():
    '''Create server, begin IOLoop
    '''
    tornado.options.parse_command_line()
    http_server = HTTPServer(Application(), xheaders=True)
    http_server.listen(options.port)
    IOLoop.instance().start()

if __name__ == "__main__":
    main()
```

We need to write the Application class to meet desired functionality

sub-classing application

```
# custom imports
from basehandler import BaseHandler
import examplehandlers
# Utility to be used when creating the Tornado server
# Contains the handlers and the database connection
class Application(tornado.web.Application):
    def __init__(self):
        '''Store necessary handlers,
           connect to database
        '''

        handlers = [(r"/[/]?",
                     BaseHandler),
                    (r"/Test[/]?",
                     examplehandlers.PostHandler),
                    (r"/DoPost[/]?",
                    MORE HANDLERS AND URL PATHS
                    ]

        settings = {'debug':True}
        tornado.web.Application.__init__(self, handlers, **settings)

        SETUP DATABASE

    def __exit__(self):
        self.client.close()
```

I wrote the base handler for you

handlers should subclass it

call the super class init

more to come in a moment

BaseHandler

- check out what it does
  - built for analyzing and writing back json
  - implements both get and post requests
  - put this in the main python file to access these:

```
# custom imports
from basehandler import BaseHandler
import examplehandlers
```

we will explore this more in the demo to come!

post requests

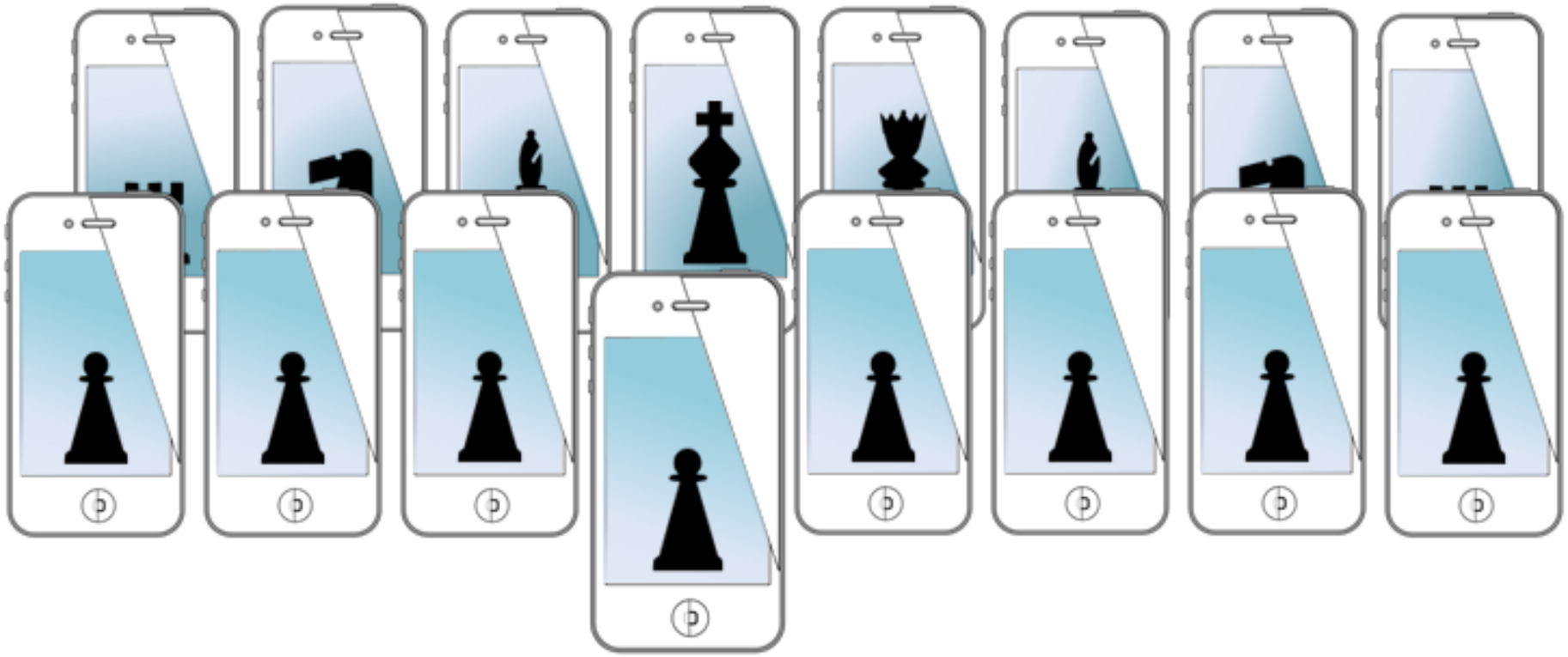
- identical handling code in python
- in our implementation, return json

```
class PostHandler(BaseHandler):
  def get(self):
   '''respond with arg1*2
   '''
   arg1 = self.get_float_arg(“arg1",default=0.0);
   self.write("Get from Post Handler? " + str(arg1*2));

  def post(self):
   '''Respond with arg1 and arg1*4
   '''
   arg1 = self.get_float_arg("arg1",default=1.0);
   self.write_json({"arg1":arg1,"arg2":4*arg1});
```

client sent GET to URL

client sent POST to URL

MOBILE SENSING LEARNING



CS5323 & 7323

Mobile Sensing and Learning

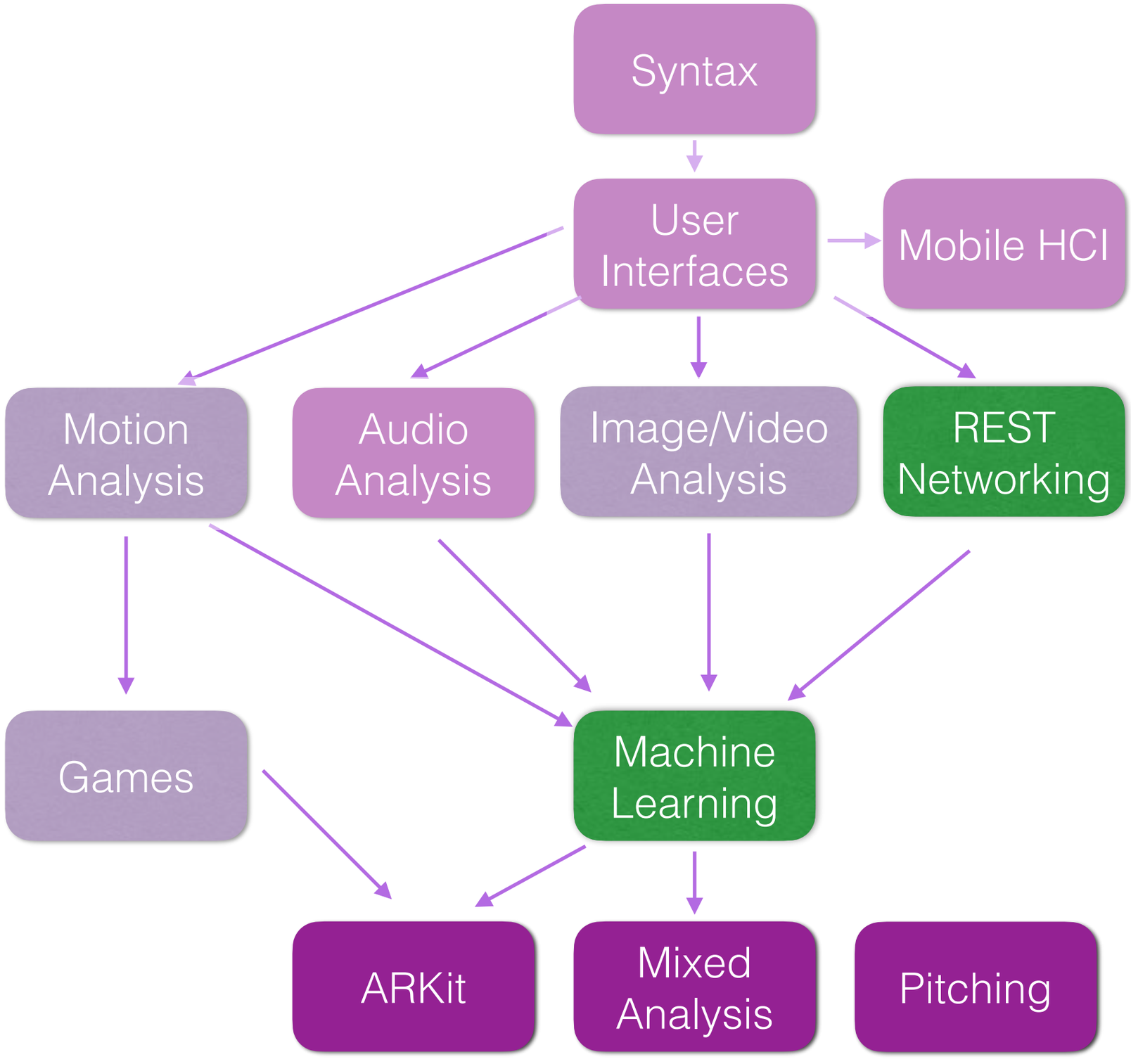
tornado, pymongo, and http requests

Eric C. Larson, Lyle School of Engineering,
Computer Science, Southern Methodist University

course logistics/agenda

- grading update (nearly up to date)
- start to think about the final project you want to propose
- agenda:
  - continue tornado
  - pymongo
  - Apple ML begin (if time)

class overview



what to install for this class?

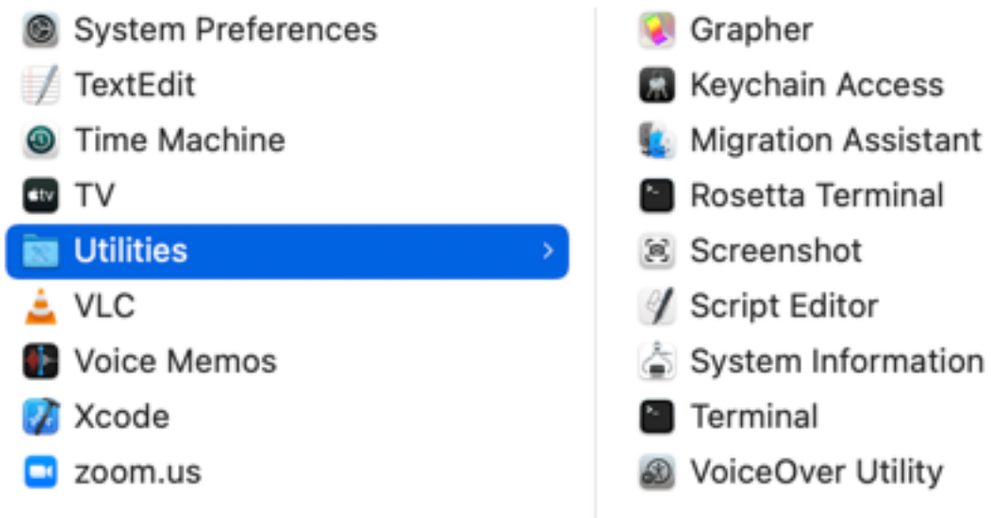
- look at installation packages list in tornado branch
- https://github.com/SMU-MSLC/tornado_bare/blob/turi_create_example/InstallPythonEnvironment.txt

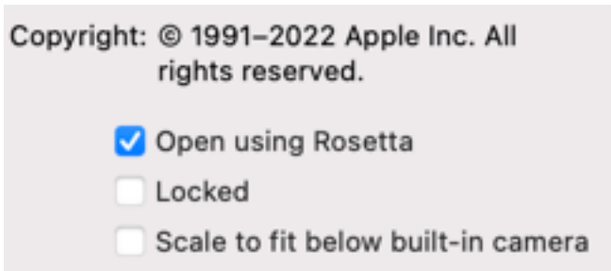
From the Rosetta terminals:
CONDA_SUBDIR=osx-64 conda create "python38env" python=3.8

conda create -n "python38env" python=3.8

Instructions for all Macs:
Note that numpy must be an older version to be compatible with Turi ...

```
conda activate python38env
python3 -m pip install --upgrade pip
pip3 install numpy==1.23.1
pip3 install pandas
pip3 install matplotlib
pip3 install scikit-learn
pip3 install seaborn
pip3 install jupyter
pip3 install coremltools
pip3 install turicreate
pip3 install pymongo
```

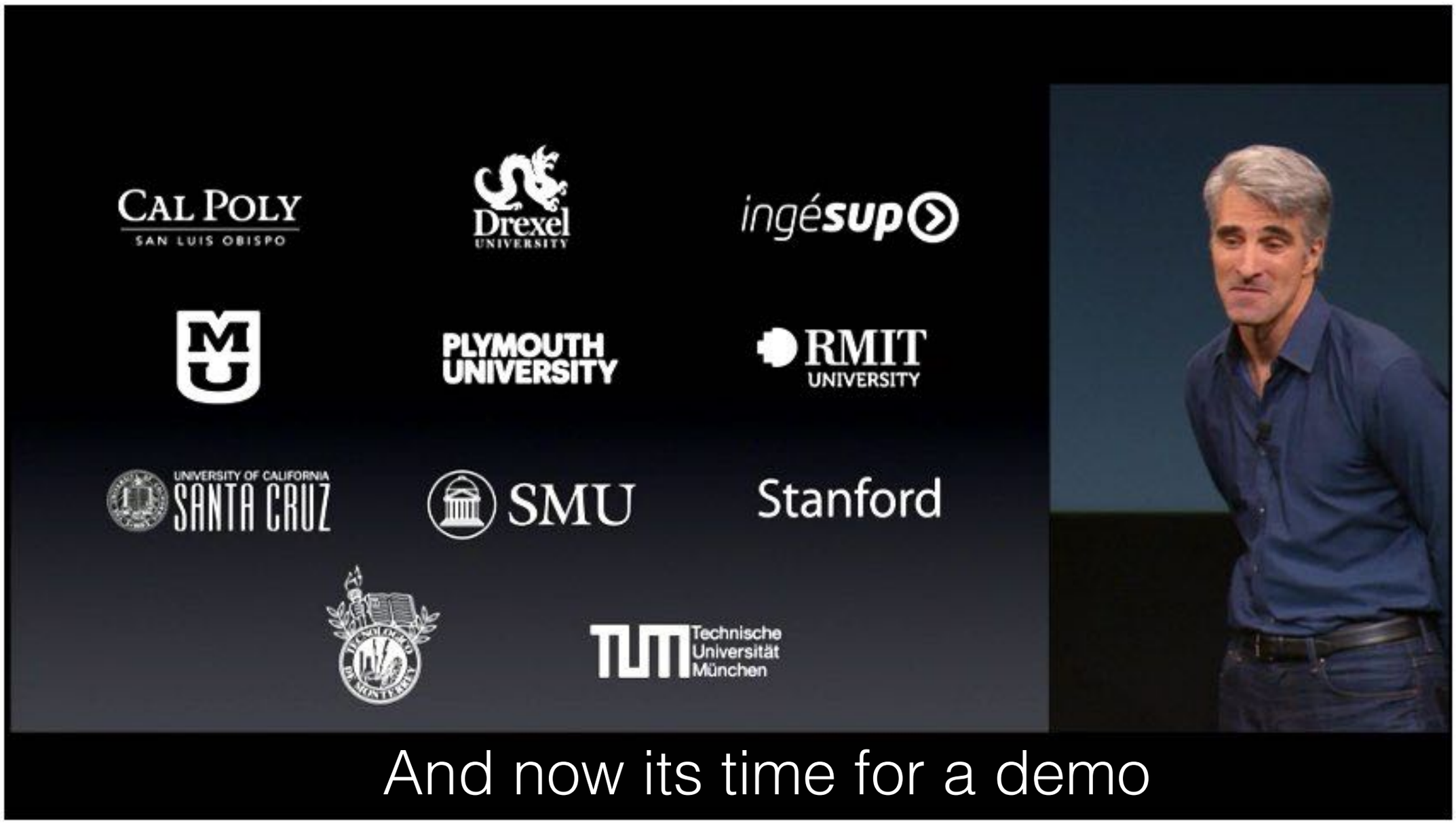




tornado examples

ifconfig | grep "inet "

- with everything, except the database
- note that quick database queries are “okay” to block on



mongodb

- hu**mongo**us data
- NoSQL database (vs relational database)
  - its a document database
- everything stored as a document
  - more or less json
  - key: value/array
- schema is dynamic
  - the key advantage of NoSQL

mongodb install

Instructions also in Repository
`InstallMongoDB.txt`

- install it
  - `brew tap mongodb/brew`
  - `brew update`
  - `brew install mongodb-community@6.0`
- you can also **run as a service** (`./mongo`)
  - `brew services start mongodb-community@6.0`
  - its running! localhost
  - `brew services stop mongodb-community@6.0`

mongodb

- a document, as stated by mongodb

Document Database

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

```
{
  name:  "sue",                          ← field: value
  age: 26,                               ← field: value
  status: "A",                           ← field: value
  groups: [ "news", "sports" ]           ← field: value
}
```

A MongoDB document.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

docs and collections

Database: MSLC_creations

default limit on size of each document: **16MB**

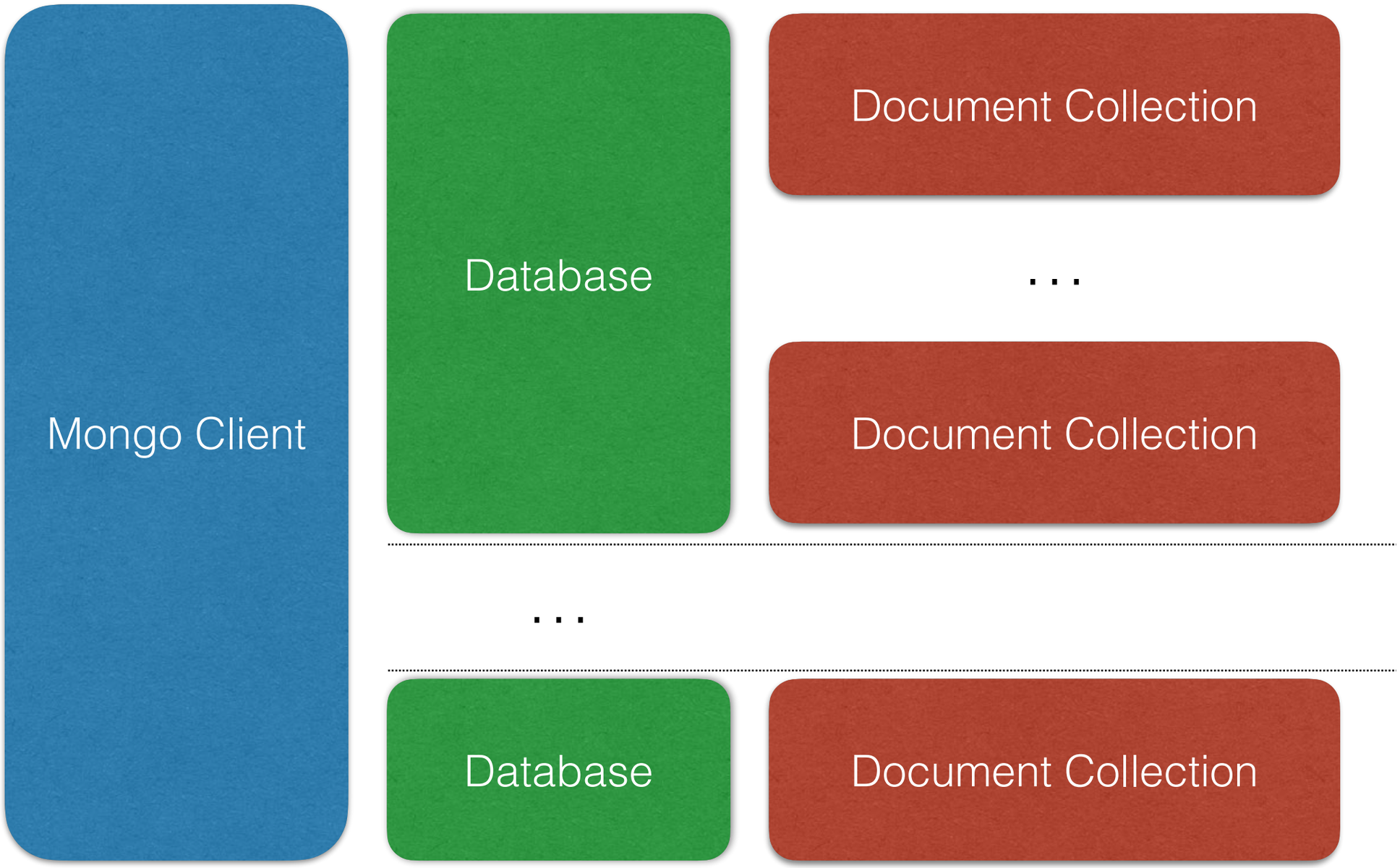
apps_collection

```
{
  app: "mongoApp",
  users: 100005,
}

{
  app: “StepCount",
  users: 45,
  rating: 2.6,
}
....

{
  app: “Trench",
  users: 4050000,
  rating: 5,
}
```

teams_collection

```
{
  team: "mongo",
  members: [ "Eric", “Ringo”, “Paul” ],
  numApps: 21,
  website: “teammongo.org",
}
{
  team: “ran off",
  members: [ "John", “Yoko” ],
  website: “flewthecoop.org",
}
....
{
  team: “21 Pilots",
  members: [ "Tyler", “Nick” ],
  numApps: 4,
  website: “RollingStone.com”,
}
```

Mongo Clients



Interface to outside world
listen on large port number

Organizational Structure in MongoDB

pymongo

- python wrapper for using mongo db

```
client  = MongoClient() # localhost, default port
db = client.some_database # access database
```

create this database, if it does not exist

```
collect = client.some_database.some_collection # access a collection
```

Mongo Client | Database | Document Collection

nothing is **created** until the first insert!!!

```
db.collection_names()
[u'system.indexes', u'some_collection']
```

get collections

pymongo (add data)

- insertion

```
dbid = db.some_collect.insert_one(
        {“key1”:values,"key2":more_values,
          “coolkey”:with_cool_values}
        )
```

unique key, _id

doc to insert

- update

```
db.some_collect.update(  {“thiskey":keyValue},
        {  "$set": {"keyToSet":valueToSet}  },
        upsert=True)
```

where ever this key is…

equal to this

set

this key to this value

insert if it does not exist (put/post)

pymongo (get data)

- find one datum in database

```
a = db.some_collect.find_one(sort=[(“sortOnThisKey", -1)])
newData = float( a[‘sortOnThisKey'] );
```

could be list of keys!

access the result

sort with this key

return last element

- iterate through many results

```
f=[];
for doc in db.some_collect.find({“keyIWant":valueOfKeyIWant}):
   doc[‘key1’] # entire document, is available
   f.append( str(doc[‘keyToGrabDataWith’]) )
```

return iterator to loop over

each iteration gives one document

- lots of advanced queries are possible

https://api.mongodb.org/python/current/

teams example

```
>>> from pymongo import MongoClient
>>> client = MongoClient()

>>> db = client.some_database
>>> collect1 = db.some_collection
>>> collect1.insert_one({“team”:"TeamFit","members":["Matt","Mark","Rita","Gavin"]})
ObjectId('53396a80291ebb9a796a8af1')

>>> db.collection_names()
[u'system.indexes', u'some_collection']

>>> db.some_collection.find_one()
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'],
u'team': u'TeamFit'}

>>> collect1.insert_one({“team":"Underscore","members":["Carly","Lauryn","Cameron"]})
ObjectId('53396c80291ebb9a796a8af2')

>>> db.some_collection.find_one()
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'],
u'team': u'TeamFit'}

>>> db.some_collection.find_one({"team":"Underscore"})
{u'_id': ObjectId('53396c80291ebb9a796a8af2'), u'members': [u'Carly', u'Lauryn', u'Cameron'],
u'team': u'Underscore'}
```

bulk operations

```
from pymongo import MongoClient

client = MongoClient()
db=client.some_database
collect1 = db.some_collection

insert_list = [{"team":"MCVW","members":["Matt","Rowdy","Jason"]},
               {“team":"CHC", "members":["Hunter","Chelsea","Conner"]}]

obj_ids=collect1.insert_many(insert_list)
```

anything iterable

```
for document in collect1.find({"members":"Matt"}):
  print(document)
```

```
{u'_id': ObjectId('53396a80291ebb9a796a8af1'), u'members': [u'Matt', u'Mark', u'Rita', u'Gavin'], u'team': u'TeamFit'}
{u'_id': ObjectId('53397331291ebb9afdd3cd2f'), u'members': [u'Matt', u'Rowdy', u'Jason'], u'team': u'MCVW'}
```

```
document = collect1.find_one({"members":"Matt","team":"MCVW"})
print (document)
```

```
{u'_id': ObjectId('53397331291ebb9afdd3cd2f'), u'members': [u'Matt', u'Rowdy', u'Jason'], u'team': u'MCVW'}
```

mongodb and binary data

reference slide

- want to store binary data more than 16MB?
- use `gridfs`, its real simple
- use `put()` and `get()` instead of

can also add metadata for easier search

wrap db in GridFS object

```
> from pymongo import MongoClient
> import gridfs
> db = MongoClient().gridfs_ex
> fs = gridfs.GridFS(db)

> a = fs.put("hello world”)
> fs.get(a).read()
'hello world'
```

```
> b = fs.put("hello world”,
      filename="foo", bar="baz")
> out = fs.get(b)
> out.read()          'hello world'
> out.filename          u'foo'
> out.bar               u'baz'
> out.upload_date
datetime.datetime(...)
```

put/read used like file object id, “a” is like file pointer

/current/examples/gridfs.html

mongodb and binary data

reference slide

- want to store binary data more than 16MB?
- use `gridfs`, its real simple
- use `put()` and `get()` instead of `insert()` and `find()`
  - `get()` returns a "file- ... d in chunks

search using metadata

```
for grid_out in fs.find({"filename": "foo.txt"},
                        no_cursor_timeout=True):
    data = grid_out.read()

most_recent_three = fs.find().sort(
                    "uploadDate", -1).limit(3)
```

get three most recent files

http://api.mongodb.com/python/current/examples/gridfs.html

mongodb + tornado

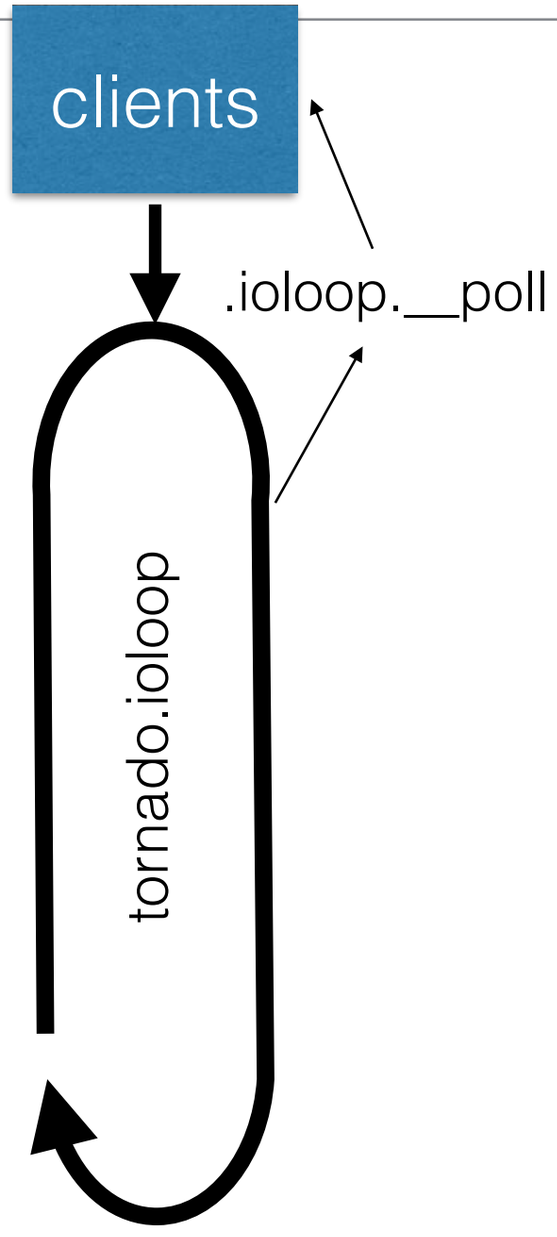
- we will use pymongo and tornado
  - mongodb runs localhost, tornado mediates access
  - good for learning
  - but real product would need load balancing (nginx)
  - and asynchronous calls (decorators or async/await)

```
>>> async def do_find_one():
...     document = await db.test_collection.find_one({'i': {'$lt': 1}})
...     pprint.pprint(document)
...
>>> IOLoop.current().run_sync(do_find_one)
{'_id': ObjectId('...'), 'i': 0}
```

the motor package is great for this!

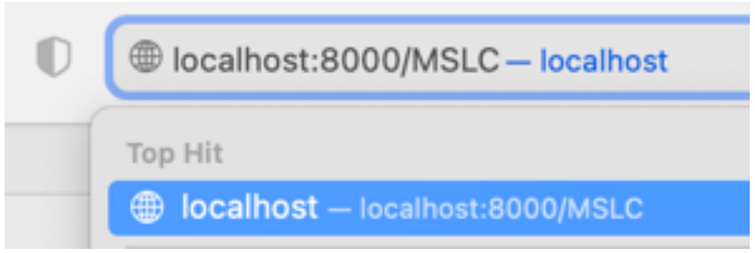
https://motor.readthedocs.io/en/stable/tutorial-tornado.html

1. `brew services start`



Mongo Client

Tied to Tornado Instance Property



3. test local queries in a browser
(if you want)

ifconfig | grep "inet "

4. make queries from external sources via external facing IP

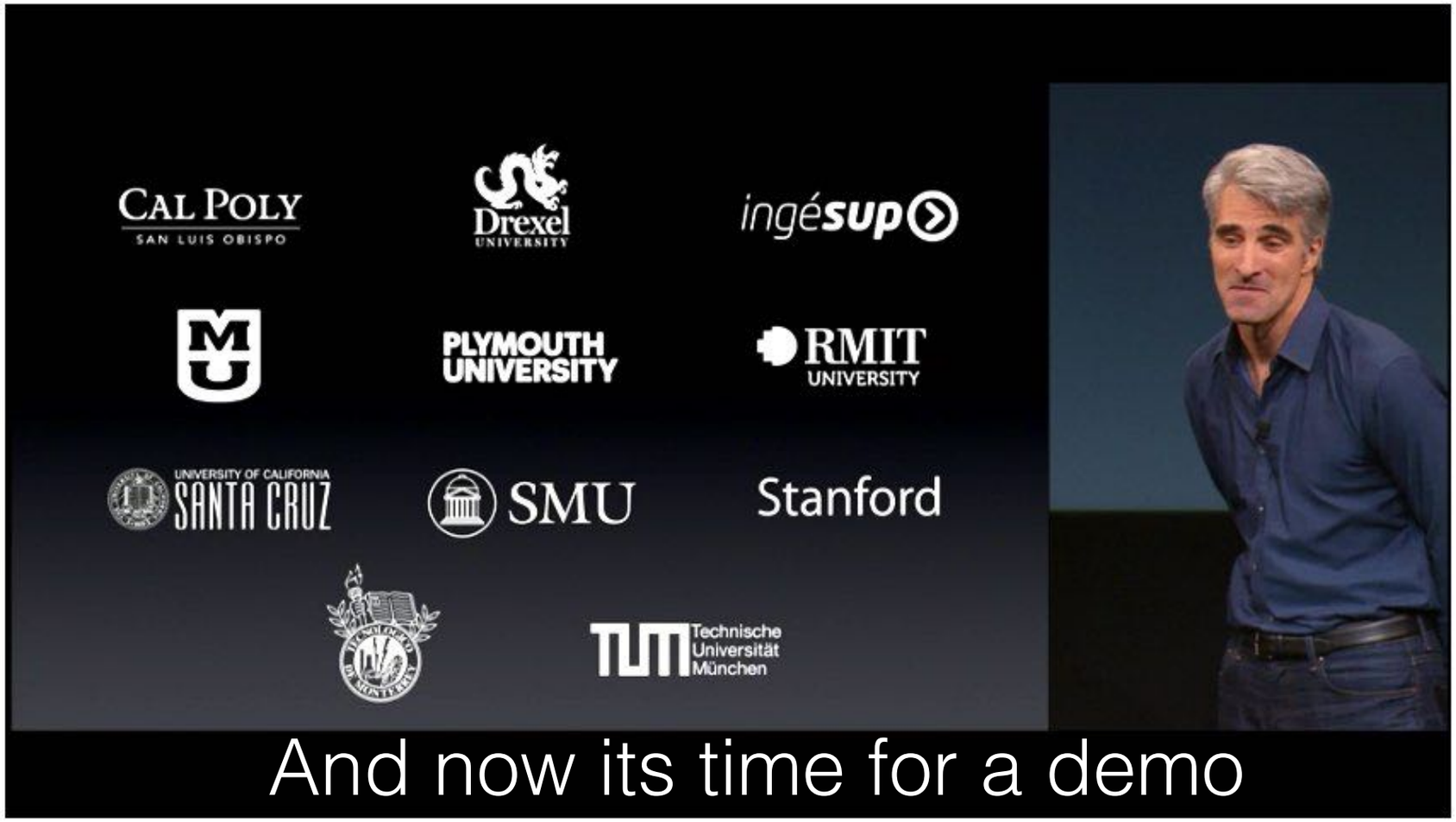
2. run tornado.Application()

`python tornado_script.py`

mongodb + tornado

ifconfig | grep "inet "

- demo:
  - store data inside mongodb with each http request



and add ***something*** to it