

# MOBILE SENSING LEARNING



## CS5323 & 7323

Mobile Sensing and Learning

tornado, pymongo, and http requests

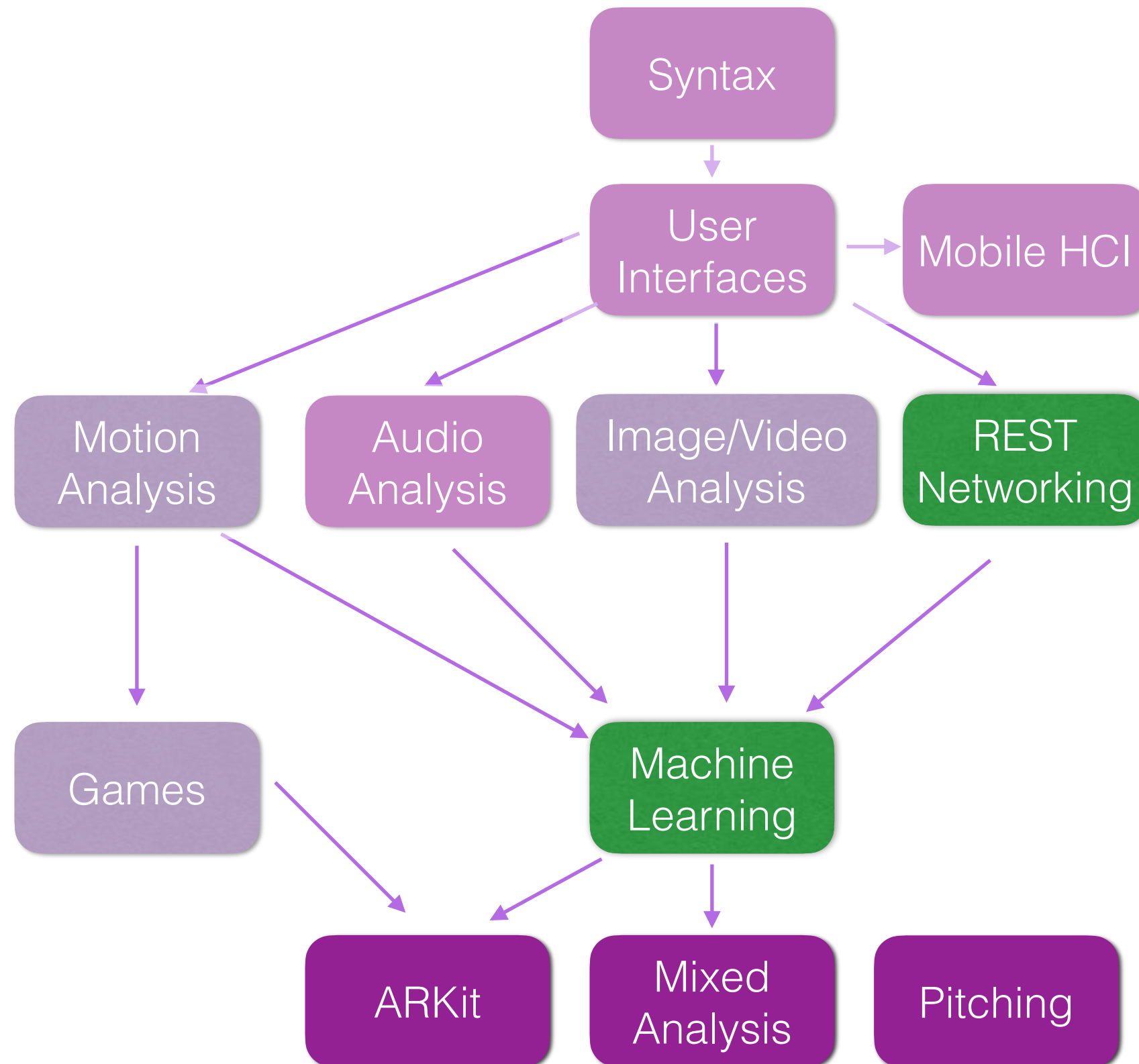
Eric C. Larson, Lyle School of Engineering,  
Computer Science, Southern Methodist University

# course logistics/agenda

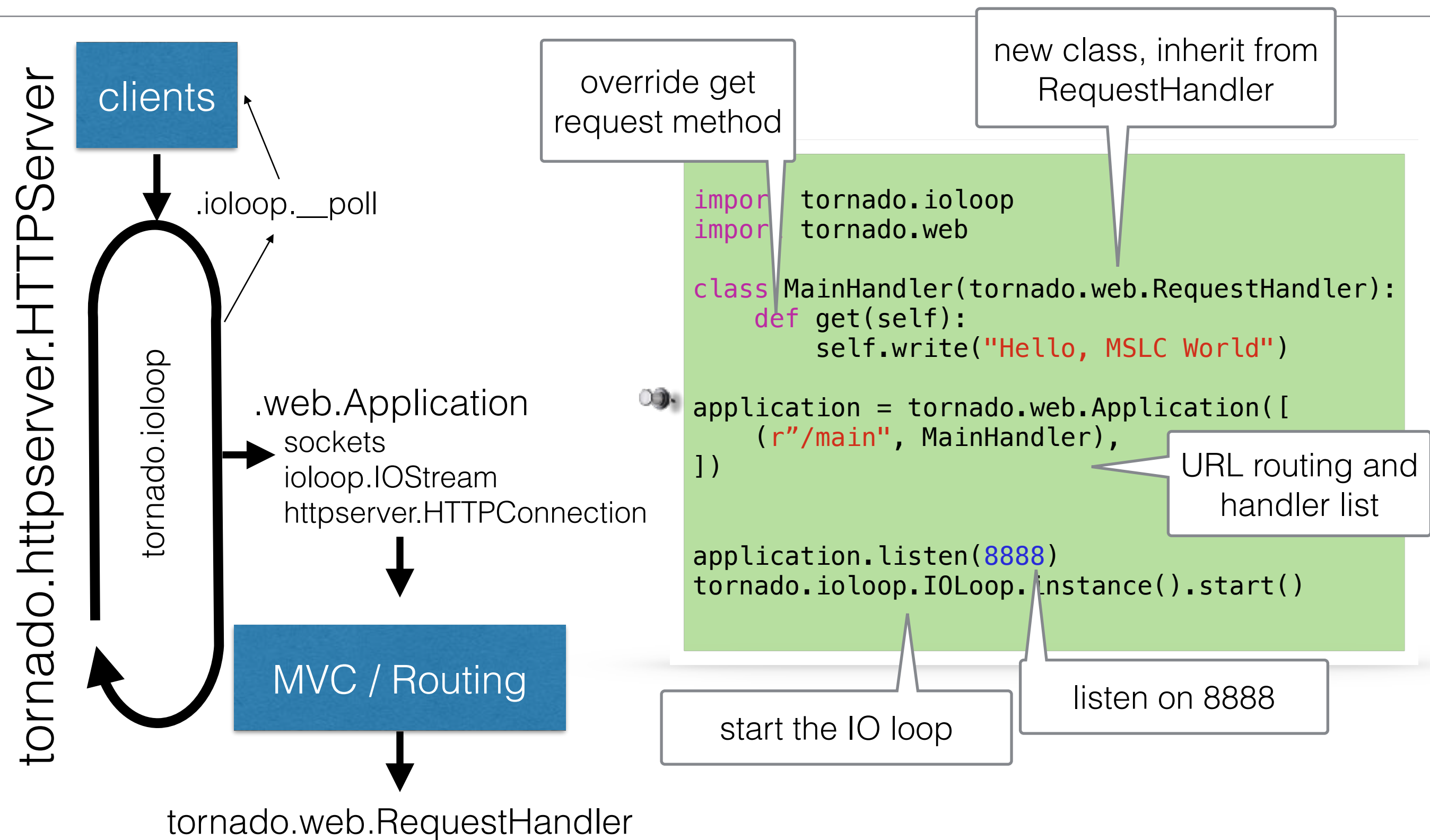
---

- grading update
- start to think about the final project you want to propose
- agenda:
  - continue tornado
  - pymongo

# class overview



# tornado review



# 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 concurrent
  - 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



```
# 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.TestHandler),
                    (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

more to come in a moment



# post versus get

## Compare GET vs. POST

The following table compares the two HTTP methods: GET and POST.

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL  Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

credit: [w3schools.com](http://w3schools.com)



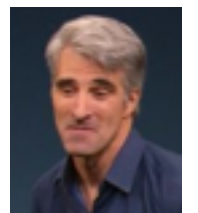
# BaseHandler



- 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 versus get

- if we are sending data to a server for processing
  - of unknown length
  - of many different formats
  - possibly in a multi-part file
- should we use post or get requests?
- why?

# post requests



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

convenience function written for you!

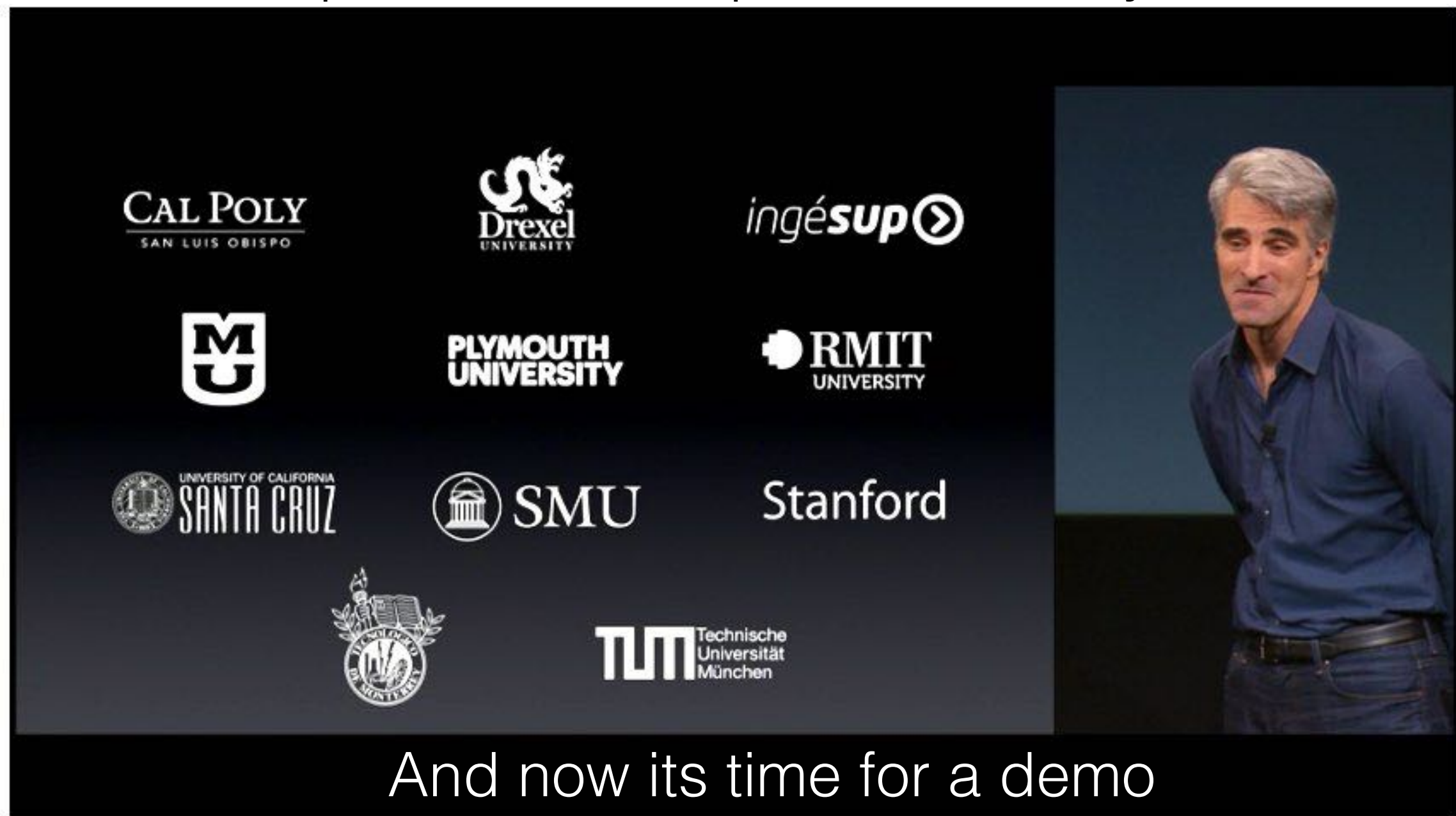
```
class PostHandler(BaseHandler):  
    def get(self):  
        '''respond with arg1*2'''  
  
        arg1 = self.get_float_arg("arg1", default="none");  
        self.write("Get from Post Handler? " + str(arg1*2));  
  
    def post(self):  
        '''Respond with arg1 and arg1*4'''  
  
        arg1 = self.get_float_arg("arg1", default=1.0);  
        self.write_json({"arg1":arg1, "arg2":4*arg1});
```

convenience function written for you!

# tornado examples

ifconfig | grep "inet "

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



# mongodb

---

- hum**mong**ous 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

- install it
- <http://www.mongodb.org/downloads>
- to run single server database:
  - make a directory for the db
    - like `data/db`
  - run `mongodb`
    - `./mongod --dbpath "<path to db>"`
  - its running! localhost
- you can also **run as a service** (`./mongo`)



make sure this exists!



# 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",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```



← field: value  
← field: value  
← field: value  
← 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

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(  
    {"key1":values,"key2":more_values,  
     "coolkey":with_cool_values}
```

doc to insert

unique key, \_id

- 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

# pymongo (get data)



- find one datum in database

could be list of keys!

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

access the result

sort with this key

return last element

- iterate through many results

```
f=[];  
for a in db.some_collect.find({"keyIWant":valueOfKeyIWant}):  
    a['key1'] # entire document is available here  
    f.append( str(a['keyToGrabDataWith']) )
```

- 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({"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({"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(insert_list)
```

```
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

- want to store binary data more than 16MB?

- use `gridfs`, its real simple

- use `put()` and `get()` instead of `insert()` and `find()` object, `fs` 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)

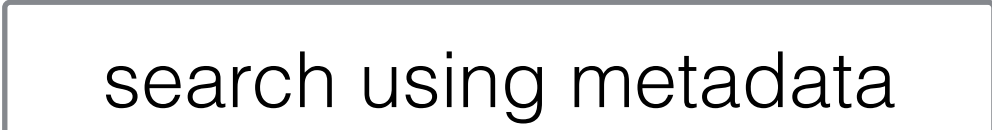

> a = fs.put("hello world")
> fs.get(a).read()
'hello world'
```

`put/read` used like file  
object id, "a" is like file pointer

```
> 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(...)
```

[current/examples/gridfs.html](http://current/examples/gridfs.html)

# mongodb and binary data

- 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- search using metadata  in chunks

```
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

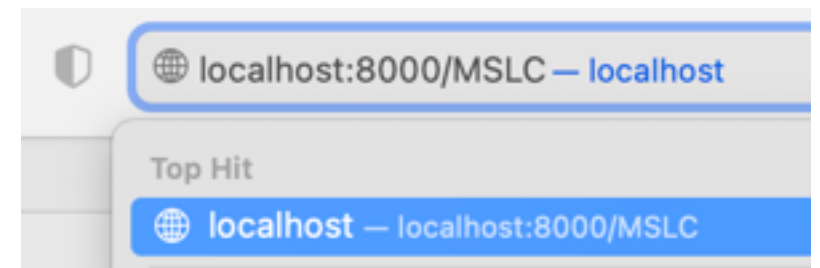
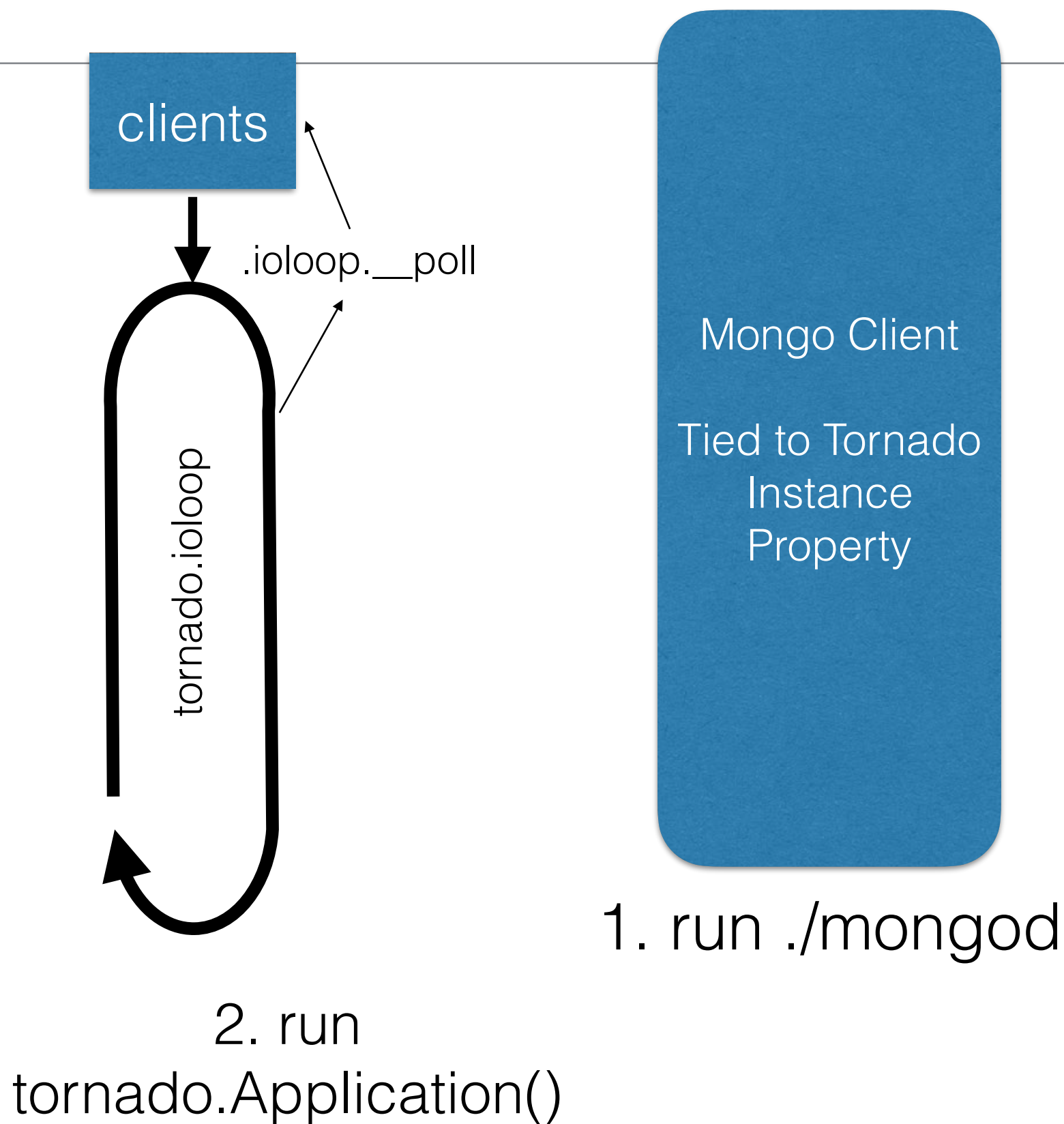
- we will use pymongo and tornado
  - mongodb runs localhost, tornado mediates access
  - good for learning
  - but real product would need load balancing (nginx)
  - and asynchronous calls (decorators or async/await)

```
>>> 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>





3. make local queries

`ifconfig | grep "inet "`

4. make queries  
from external sources  
via external facing IP

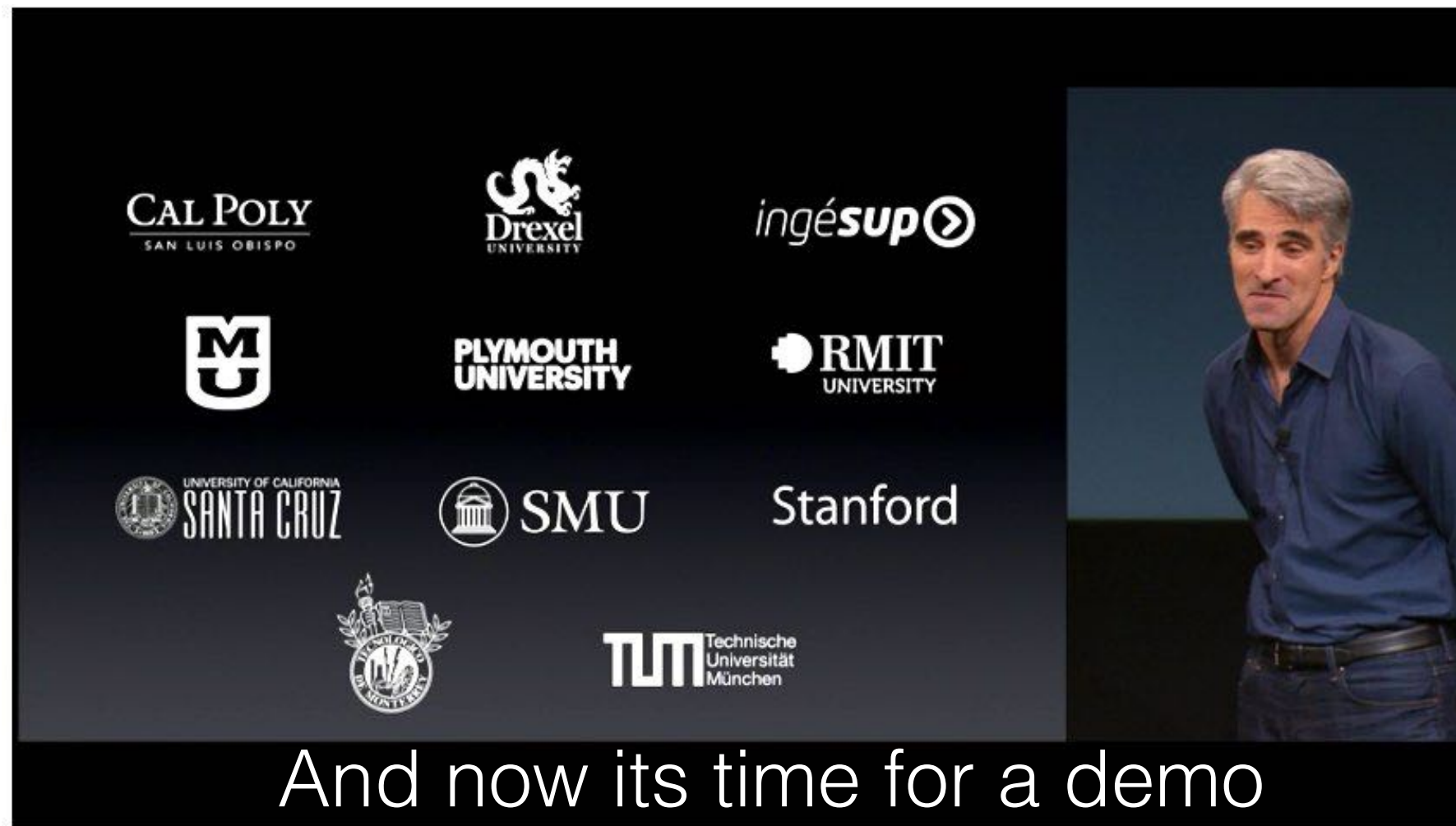




# mongodb + tornado

ifconfig | grep "inet "

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



and add ***something*** to it