Note: the document is based on Mac OS X.

Install and Run MongoDB

- Install Homebrew

  You can download to install Homebrew, or if you already installed it, just update it with following command:

  ```
  $ brew update
  ```

- Install MongoDB

  To install the MongoDB, issue the following command in the terminal:

  ```
  $ brew install mongodb
  ```

- Run MongoDB

  Note: By default the brew would config the MongoDB path, you can execute the `sudo mongod` to test it. If the command is unknown, you may have to refer to this step's guidance.

First, add the mongodb path to $PATH, issue the following command:

```
$ vi ~/.bash_profile
```

Add the following code at the end of the file:

```
export PATH=$PATH:<PATH-FOR-MONGODB-PATH>
```

Then execute the following command to update the .bash_profile file:

```
$ source ~/.bash_profile
```

Before start MongoDB for the first time, you need to check if there is /data/db folder, if it has this folder, you can run the following command:

```
 $ sudo mongod
```

If the folder do not exist, you need to create this folder, then run the start command:

```
$ mkdir -p /data/db
$ sudo mongod
```

- Use mongo Shell

  Once you have [installed and started]() MongoDB, you can connect to your MongoDB instance via mongo shell. You can execute SQLs for the common database operation.

  On the same system where the MongoDB is running, open a terminal window and run the mongo shell with the following command:

  $ ./mongo

  ```
  show dbs
  WallExplorer 0.000GB
  connTest 0.000GB
  local 0.000GB
  test 0.000GB
  ```

  Note: Ensure that MongoDB is running before attempting to launch the mongo shell.

MongoDB Common Operation

Data Init

Use `mongoimport` to import data from json/csv files to database, `mongoexport` to export data from database as json/csv files.

In the following examples, mongoimport imports data from the file json or csv into the database marketing's contacts collection. As the database was hosted remote, connection must have authentication.

Case1: Import JSON

```
  $ mongoimport --host mongodb1.example.net --port 37017 \
 --username user --password pass --collection contacts --db marketing \
 --file /opt/ backups/mdb1-examplenet.json
```

Case2: Import CSV

```
   $ mongoimport -db marketing -collection contacts --type csv \
-headerline --file /opt/backups/contacts.csv
```

Connection

Methods

There are some connection methods using to provide connection

- Mongo.getDB() - provides access to database objects from the mongo shell or from a JavaScript file, returns a database object.
- Mongo.getReadPrefMode() - return the current read preference mode, as in the following example:

  ```
  db.getMongo().getReadPrefMode()
  ```

- Mongo.getReadPrefTagSet() - used to return the current read preference tag set for the MongoDB connection.

  ```
  db.getMongo().getReadPrefTagSet()
  ```

- Mongo.setReadPref() - set the read preference for the MongoDB connection, the following operation can set a read preference mode.

  ```
  db.getMongo().setReadPref('primaryPreferred')
  ```

- Mongo.setSlaveOk() - permits read operations from non-master(i.e slave or secondary) instances. Practically, use this method in the following form:

  ```
  db.getMongo().setSlaveOk()
  ```

- Mongo() - used to create a new connection object.

- connect() - used to connect to a MongoDB instance and to a specified database on that instance.

Connect with Python

PyMongo is a Python package used to interact with MongoDB database.

- Prerequisites

  Before start, make sure that you have the PyMongo distribution installed. In the Python shell, the following should run success without raising an exception:

  ```
  >>> import pymongo
  ```

  This tutorial also assumes that a MongoDB instance is running on the default host and port. Assuming you have downloaded and installed MongoDB, you can start it like so:

  ```
  $ mongod
  ```

- Create a Connection

  The first step when working with PyMongo is to create a MongoClient to the running mongod instance:

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

  The above code will connect on the default host and port. You can also specify the host and port explicitly, as follow:

  ```
  >>> client = MongoClient('localhost', 27017)
  ```

Or use the MongoDB URI format:

```
>>> client = MongoClient('mongodb://localhost:27017/')
```

- Get a Database

  A single instance of MongoDB can support multiple independent databases. When working with PyMongo you can access databases as below command, the test_database is the database name.

  ```
  >>> db = client.test_database
  ```

Insert

In MongoDB, the db.collection.insert() method adds new documents into a collection.

Insert a Document

- Insert a document into a collection

  The operation will create the collection inventory if the collection does not currently exist.

  ```
  db.inventory.insert(
      {
          item: "ABC1",
          details: {
              model: "14Q3",
              manufacturer: "XYZ Company"
          },
          stock: [{size:"S", qty:25}, {size:"M", qty:50}],
          category:"clothing"
      }
  )
  ```

  The operation returns a WriteResult object with the status of the operation. A successful insert of the document returns the following object:

  ```
    WriteResult({"nInserted":1})
  ```

  - Review the inserted document

    If the insert operation is successful, verify the insertion by querying the collection.

```
db.inventory.find()
```

The document you inserted should return:

```
{ "_id" : ObjectId("53d98f133bb604791249ca99"),
"item" : "ABC1", "details" : { "model" : "14Q3", "manufacturer" : "XYZ Company" },
"stock" : [ { "size" : "S", "qty" : 25 }, { "size" : "M", "qty" : 50 } ],
"category" : "clothing" }
```

Insert an Array of Documents

You can pass an array of documents to the db.collection.insert() method to insert multiple documents.

- Create an array of documents

  Define a variable mydocuments that holds an array of documents to insert.

```
var mydocuments =
[
    {
        item: "ABC2",
        details:{model:"14Q3", manufacturer: "M1 Corporation"},
        stock: [{size: "M", qty: 50}],
        category: "clothing"
    },
    {
        item: "MN02",
        details:{model:"14Q3", manufacturer: "ABC Company"},
        stock: [{size: "S", qty: 5}, {size: "M", qty: 5}, {size: "L",qty: 1}],
        category: "clothing"
    },
    {
        item: "IJK2",
        details: {model: "14Q2", manufacturer: "M5 Corporation"},
        stock: [{size: "S", qty: 5}, {size: "L", qty: 1}],
        category: "houseware"
    }
];
```

- Insert the documents

  Pass the mydocuments array to the db.collection.insert() to perform a bulk insert.

The method returns a BulkWriteResult object with the status of the operation. A successful insert of the documents returns the following object:

~~~
```
BulkWriteResult({
    "writeErrors": [ ],
    "writeConcernErrors": [ ],
    "nInserted": 3,
    "nUpserted": 0,
    "nMatched": 0,
    "nModified": 0,
    "nRemoved": 0,
    "upserted": [ ]
})

```
~~~

Insert Multiple Documents with Bulk

- Initialize a Bulk Operations Builder

  Initialize a Bulk operations builder for the collection inventory.

  ```
  var bulk = db.inventory.initializeUnorderedBulkOp();
  ```

  - Add Insert Operations to the Bulk Object.

  Add two insert operations to the bulk object using the Bulk.insert() method.

  ```
   bulk.insert(
      {
          item: "BE10",
          details: {model:"14Q2", manufacturer: "XYZ Company"},
          stock: [{size: "L", qty: 5}],
          category: "clothing"
      }
   );
   bulk.insert(
      {
          item: "ZYT1",
          details: {model:"14Q1", manufacturer: "ABC Company"},
          stock: [{size: "S", qty: 5}, {size: "M", qty: 5}],
          category: "houseware"
      }
   );
  ```

- Execute the Bulk Operation

Call the execute() method on the bulk object to execute the operations in its list:

```
bulk.execute();
```

The method returns a BulkWriteResult object with the status of the operation. A successful insert of the documents returns the following object:

```
BulkWriteResult({
    "writeErrors": [ ],
    "writeConcernErrors": [ ],
    "nInserted": 2,
    "nUpserted": 0,
    "nMatched": 0,
    "nModified": 0,
    "nRemoved": 0,
    "upserted": [ ]
})
```

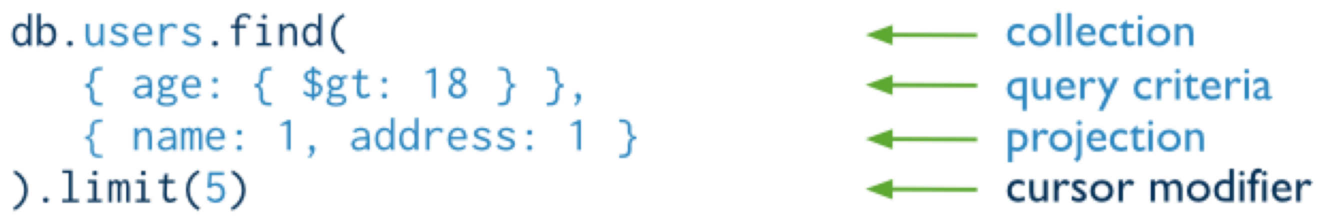
Query

A query retrieves data stored in the database and may include a projection that specifies the fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network.

- Query Interface

  For query operations, MongoDB provides a db.collection.find() method. The method accepts both the query criteria and projections and returns a cursor to the matching documents.

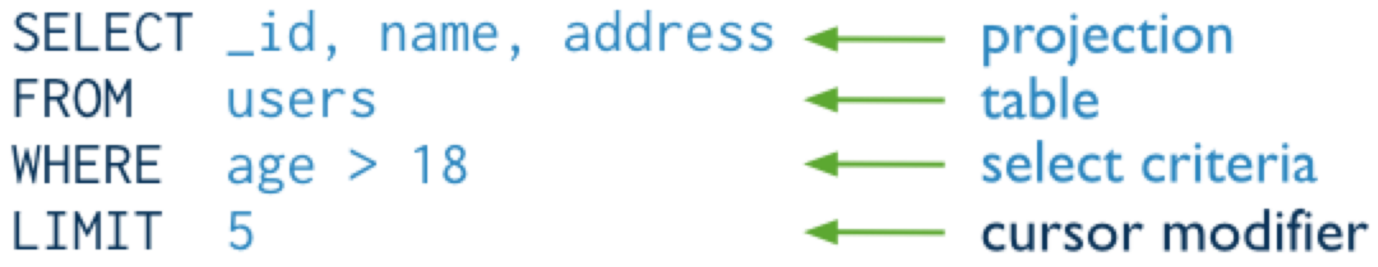
  The following diagram highlights the components of a MongoDB query operation:



```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

Name and address are projection which represents filter columns, 1 means exist and need to be listed, 0 means not exist and don't need to be listed.

The next diagram shows the same query in SQL:


```
SELECT _id, name, address  ←  projection
FROM   users               ←  table
WHERE  age > 18            ←  select criteria
LIMIT  5                   ←  cursor modifier
```

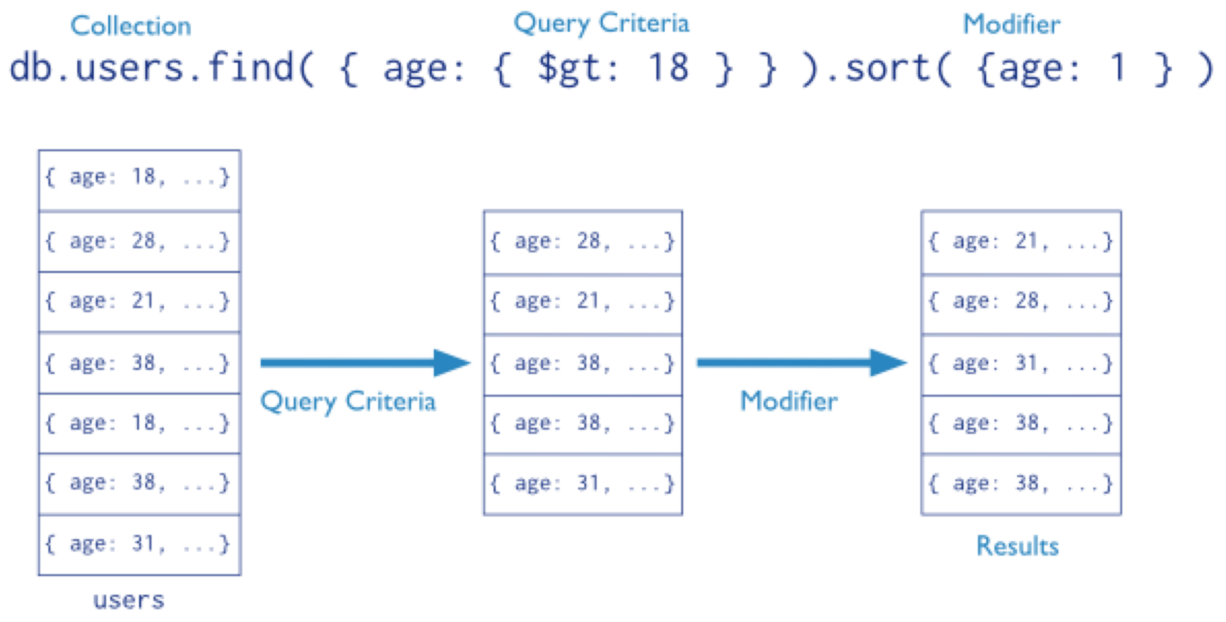
- Query Behavior MongoDB queries exhibit the following behavior:

```
- All queries in MongoDB address a single collection
- You can modify the query to impose limits, skips, and sort orders
- The order of documents returned by a query is not defined unless specify a sort()
- Operations that modify existing docuements use the same query syntax as queries to select documents to update.
- In aggregation pipleline, the $match pipeline stage provides access to MongoDB queries.
```

MongoDB provides a db.collection.findOne() method as a special case of find() that returns a single document.

- Query Statements

  Consider the following diagram of the query process that specifies a query criteria and a sort modifier:



In the diagram, the query selects documents from the users collection. Using a query selection operator to define the conditions for matching documents, the query selects documents that have age greater than 18. Then the sort() modifier sorts the results by age in ascending order.

Python Code for MongoDB Operation

Using Pymongo to connect MongoDB

- Install pymongo

  install pymongo through pip

  ```
  $ sudo pip install pymongo
  ```

- Create a json file

  Create a json file named data.json to store the data, the following is an example:

```
{
    "name": "xiaoming",
    "age": 20,
    "birthplace": {
```

```
        "province": "Shanghai",
        "city": "Shanghai",
        "area": "Pudong new area"
    }
}
```

- Create database and import the json data into database

  Import data into the collection. Go to the folder where the json file has been put, and issue the following command:

  ```
  $ mongoimport --db connTest --collection person --drop --file data.json
  ```

- Connect to mongodb via Python

  create a python file named connTest.py to import pymongo as following:

  ```
  from pymongo import MongoClient

  client = MongoClient()
  db = client.connTest

  result = db.person.insert_one({
          "name": "xiaozhang",
          "age": 22,
          "birthplace": {
          "province": "zhejiang",
          "city": "Hangzhou",
          "area": "Xiaoshan area"
      }
  })

  persons = db.person.find()

  for person in persons:
      print("name:", person['name'], 'age:', person['age'])

  client.close()
  ```

- Test the execute result through command

  Run the `python connTest.py`, if it runs correctly and prints the following information, it says that connect to a MongoDB database via Python successfully.

  ```
  ('name:', u'xiaoming' 'age:', 20)
  ('name:', u'xiaozhang' 'age:',22)
  ```

Application Examples

HD-Wall-Explorer

The example of this project which have used MongoDB as the database, path for this code is /HD-Wall-Explorer/app/controllers/apps.py

```
from app import app
from pymongo import MongoClient
from bson import json_util
from flask.ext.cors import CORS, cross_origin
from flask import json,request,send_file

client = MongoClient('localhost', 27017)
db = client.WallExplorer

def getMaxId():
    existedApps = db.apps.find().sort( [("id", -1)] ).limit(1)
    return existedApps[0]['id']

    #...

@app.route('/HDWallExplorer/api/v1.0/apps',methods=['POST'])
def save_app():
    jsondata = request.data
    app = json.loads(jsondata)
    apphash = {}

    if('id' not in app): #Create New Comment: id is NULL
        if 'description' not in app:
            app['description']  = ''
        maxid = getMaxId()
        apphash = {
                "id":maxid + 1,
                "name":app['name'],
              "path":"http://www.abc.com",
              "description":app['description'],
              "thumbnail":app['thumbnail'],
              "isdeleted": False
        }
        db.apps.insert(apphash)
    elif('isdeleted' in app and app['isdeleted'] == True): #Delete Comment: id is NOT NULL && isdeleted is TRUE
        id = app['id']
        db.apps.update({'id':int(id)},{'$set':{'isdeleted' : True}})
    else: #Update Comment: id is NOT NULL && isdeleted is FALSE
```

```
        id = app['id']
        print 'What the fucking update:  ', jsondata
        db.apps.update({'id':int(id)},{'$set':{
                'name':app['name'],
                'path':app['path'],
                'description':app['description'],
                'thumbnail':app['thumbnail']
                }
            }
        )
    return json_util.dumps(apphash, sort_keys=True, indent=4, separators=(',',':'))
```

Divvy

The example of this project which have used MongoDB as the database，path for this code is /divvy/app/controllers/routes.py

```
from app import app
from pymongo import MongoClient
from bson import json_util
from flask.ext.cors import CORS, cross_origin
from flask import json,request,send_file

client = MongoClient('localhost', 27017)
db = client.Divvy

@app.route('/divvy/api/v1.0/routes/start=<start_id>&end=<end_id>',methods=['GET'])
@cross_origin(origin='*', methods=['GET', 'POST', 'OPTIONS'], headers=['X-Requested-With', 'Content-Type', 'Origin'])
def get_routes_byid(start_id,end_id):
    routes = db.routes.find({'id':{'$gte': int(start_id),'$lte': int(end_id)}},{'_id': False})
    return json_util.dumps(routes, sort_keys=True, indent=4, separators=(',',':'))

# ...

@app.route('/divvy/api/v1.0/routes/Update',methods=['PUT'])
def update_mass_routes():
    jsondata = request.form['jsondata']
    routes = json.loads(jsondata)
    for route in routes:
        id = route['id']
        geo_data = route['geo_data']
        db.routes.update({'id':int(id)},{'$set':{'geo_data' : geo_data}})
    return json_util.dumps(routes, sort_keys=True, indent=4, separators=(',',':'))
```

Compliance360-API

In this project, it has used MongoDB as the database, different from the first two projects, this project has used MongoAlchemy to increase the client model definition. Path for this code is /Compliance360-API/app/models.py:

```
from mongoalchemy.document import Document
from mongoalchemy.fields import *
session = Session(MongoClient('localhost', 27017)).connect('Compliance360')

#...

class KYC(Document):
    UNIQUE_ID = IntField()
    CUSTOMER_NAME = StringField()
    KYC_ACCOUNT_COUNTRY = StringField()
    KYC_COUNTRY_DOMICILE_INC = StringField()
    #...

    def __str__(self):
        return '%s' % (self.UNIQUE_ID)

    def __getitem__(self, key):
        return self.__getattribute__(key)

    @staticmethod
    def getStatics():
        statistics = {'geos': set(), 'categories': set(), 'industries': set()}

        for k in session.query(KYC):
            statistics['geos'].add(k.KYC_ACCOUNT_COUNTRY)
            statistics['categories'].add(k.KYC_INDUSTRY_TYPE)
            statistics['industries'].add(k.CLIENT_CATEGORY)
        return statistics

    @staticmethod
    def queryByParams(geographies, industries, categories):
        kyc_ids, kyc_subset = [], []

        for item in session.query(KYC) \
                .in_(KYC.KYC_ACCOUNT_COUNTRY, *geographies) \
                .in_(KYC.KYC_INDUSTRY_TYPE, *industries) \
                .in_(KYC.CLIENT_CATEGORY, *categories):
            kyc_subset.append(item)
            kyc_ids.append(item.UNIQUE_ID)
```