DSCI 551 Final Project: Emulating Firebase

Final Report
JobMatch

Team members:
Lavina Agarwal
Aishwarya Krishnakumar
Syeda Masooma Naqvi

Table of Contents:

Code Links

GitHub: https://github.com/Lavina4/dsci551-emulate-firebase
Google Drive:
https://drive.google.com/drive/folders/1VMt1Aaf3-shcL4R9irsgAypgqs6udkFp?usp=sharing

Overview

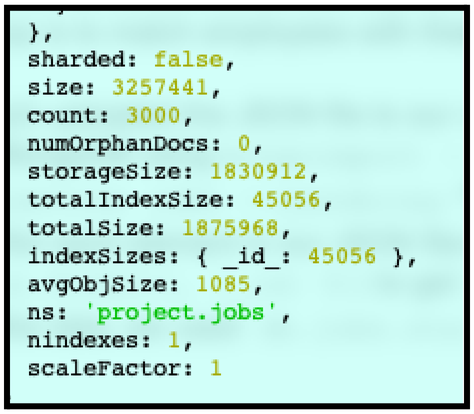
Introduction

In this project, we have developed a web application called JobMatch that was created using the Streamlit Python library. JobMatch aims to connect potential employees with employers. As students who will soon be entering the job market ourselves, we felt that creating a platform that streamlines the job search process would be incredibly useful for ourselves and our peers.

JobMatch features a variety of filters and search options that allow users to easily find job listings that meet their specific criteria. Users can search for jobs based on location, job type, company name, industry, and more, making the process of finding and applying for jobs more efficient and tailored to individual needs.

Dataset

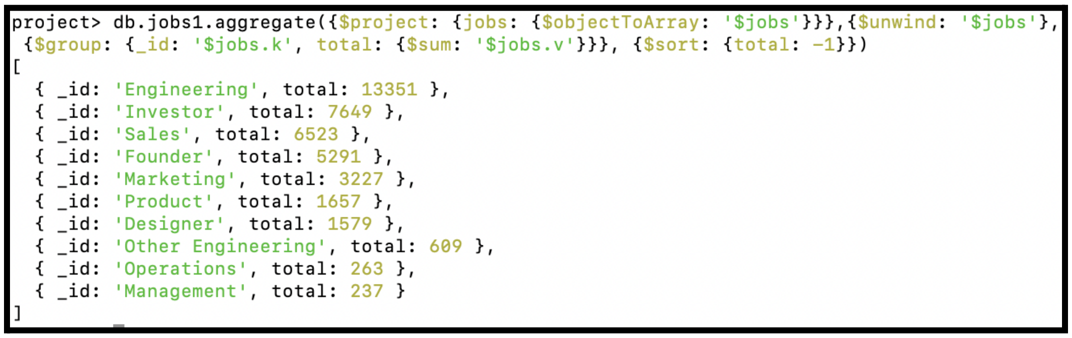
We used a JSON dataset called ‘Top Tech Startups Hiring 2023’. The 3.4 MB dataset contains job listings about 3,000 tech startups with features including Company Name, Company Tags, Company Headline, Website, Locations, Industry, # of Jobs Available, # of Employees, etc. This dataset was well-maintained by the owner. It didn’t have missing or mismatched data, which was helpful when it came to us preprocessing the data.



```
},
sharded: false,
size: 3257441,
count: 3000,
numOrphanDocs: 0,
storageSize: 1830912,
totalIndexSize: 45056,
totalSize: 1875968,
indexSizes: { _id_: 45056 },
avgObjSize: 1085,
ns: 'project.jobs',
nindexes: 1,
scaleFactor: 1
```

Methodologies

One preprocessing step that we needed to take was changing the already existing id field to _id in the JSON file. The reasoning behind this is that we wanted to use the id in the JSON file as the _id field in MongoDB. In order to set up the start of our project, we each installed MongoDB on our EC2. Then, we imported the preprocessed dataset, which we named as jobs_data.json. We then placed it into a collection. Our first step was to understand more about the data. So, we ran the stats command to get an overview. We confirmed the size of the dataset, along with learning more about the different elements. For example, as you can see below, we found how many jobs there were in each type.



```
project> db.jobs1.aggregate({$project: {jobs: {$objectToArray: '$jobs'}}},{$unwind: '$jobs'},
 {$group: {_id: '$jobs.k', total: {$sum: '$jobs.v'}}}, {$sort: {total: -1}})
[
  { _id: 'Engineering', total: 13351 },
  { _id: 'Investor', total: 7649 },
  { _id: 'Sales', total: 6523 },
  { _id: 'Founder', total: 5291 },
  { _id: 'Marketing', total: 3227 },
  { _id: 'Product', total: 1657 },
  { _id: 'Designer', total: 1579 },
  { _id: 'Other Engineering', total: 609 },
  { _id: 'Operations', total: 263 },
  { _id: 'Management', total: 237 }
]
```

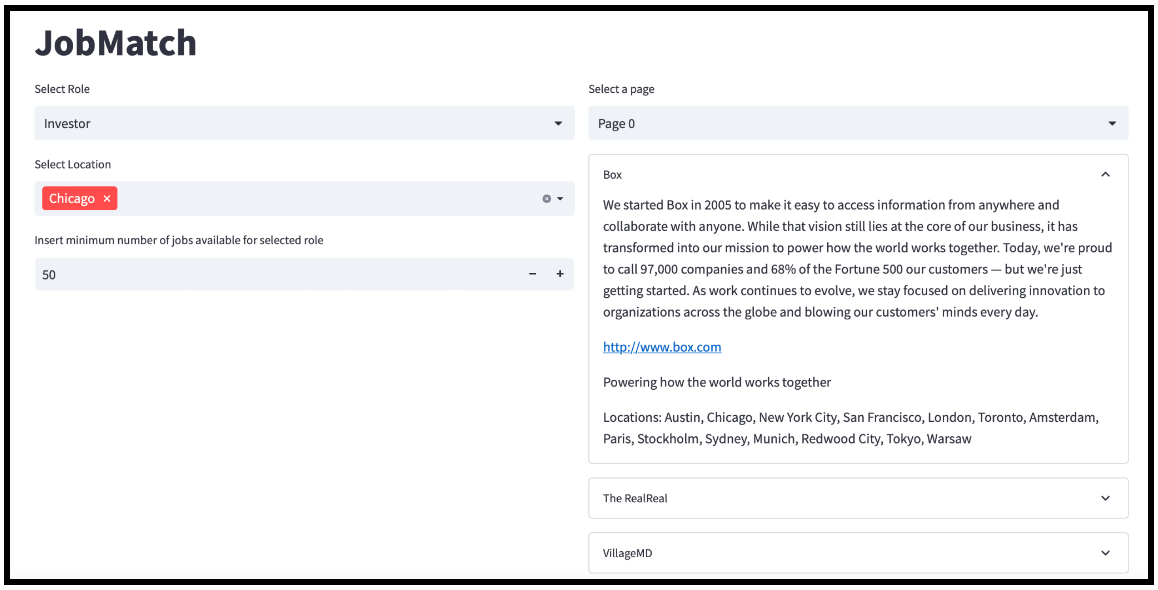
After performing data analysis on data in MongoDB installed on EC2 we moved to MongoDB Community Edition.

Model

We used Flask to create a prototype system that supports functions in Firebase RESTful API including GET, POST, PUT, PATCH, DELETE and filtering functions. We used MongoDB to store our JSON data. In addition, we utilized WebSockets to enable real-time updates of the data, which ensures that users always have access to the latest job listings and can receive notifications when new jobs that match their search criteria become available.

Web Application

Our goal with the web application was to create a page where users can search for jobs, which is automatically populated from our dataset. Another feature we added was that the employer, for example, can use CURL commands to change the dataset. For example, if an employer no longer has 30 available jobs in France, they can change that number, the country, the job type, or one of the other filters.



JobMatch

Select Role: Investor

Select a page: Page 0

Select Location: Chicago ×

Insert minimum number of jobs available for selected role: 50

Box

We started Box in 2005 to make it easy to access information from anywhere and collaborate with anyone. While that vision still lies at the core of our business, it has transformed into our mission to power how the world works together. Today, we're proud to call 97,000 companies and 68% of the Fortune 500 our customers — but we're just getting started. As work continues to evolve, we stay focused on delivering innovation to organizations across the globe and blowing our customers' minds every day.

http://www.box.com

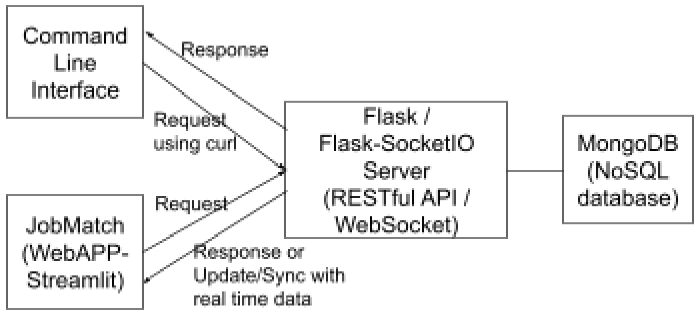
Powering how the world works together

Locations: Austin, Chicago, New York City, San Francisco, London, Toronto, Amsterdam, Paris, Stockholm, Sydney, Munich, Redwood City, Tokyo, Warsaw

The RealReal

VillageMD

Implementation



Command Line Interface
Response
Request using curl
Flask / Flask-SocketIO Server (RESTful API / WebSocket)
MongoDB (NoSQL database)
Request
JobMatch (WebAPP-Streamlit)
Response or Update/Sync with real time data

Data Collection: Dataset

Data Preprocessing

Changed the id to _id in the JSON file. We want to use the id in the json file as the _id field in MongoDB.

Importing data in MongoDB Community Edition

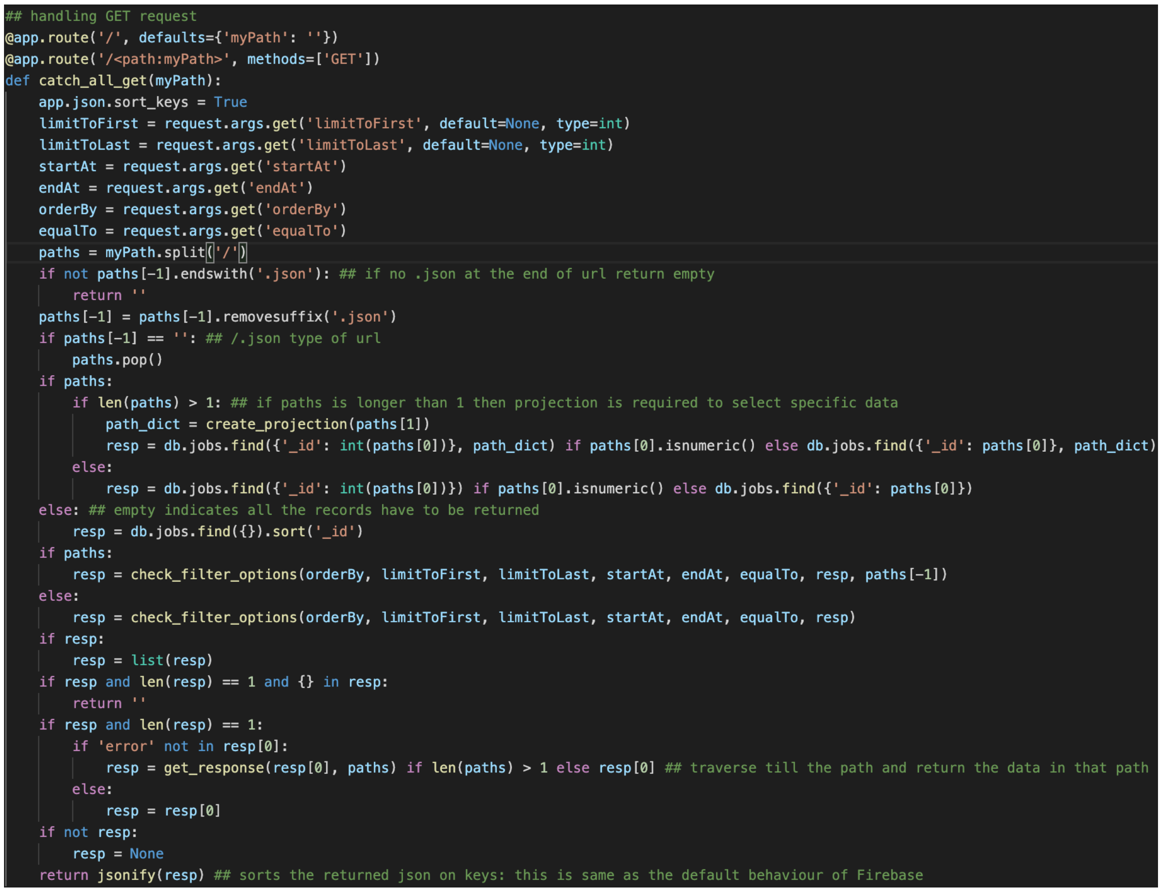
Import data to MongoDB using `mongoimport --file jobs_data.json --db project --collection jobs --jsonArray`. This creates a 'project' database and adds a collection 'jobs' with the data from jobs_data.json.

Implementing RESTFUL API

MongoClient from Pymongo library used to connect Flask with MongoDB database.

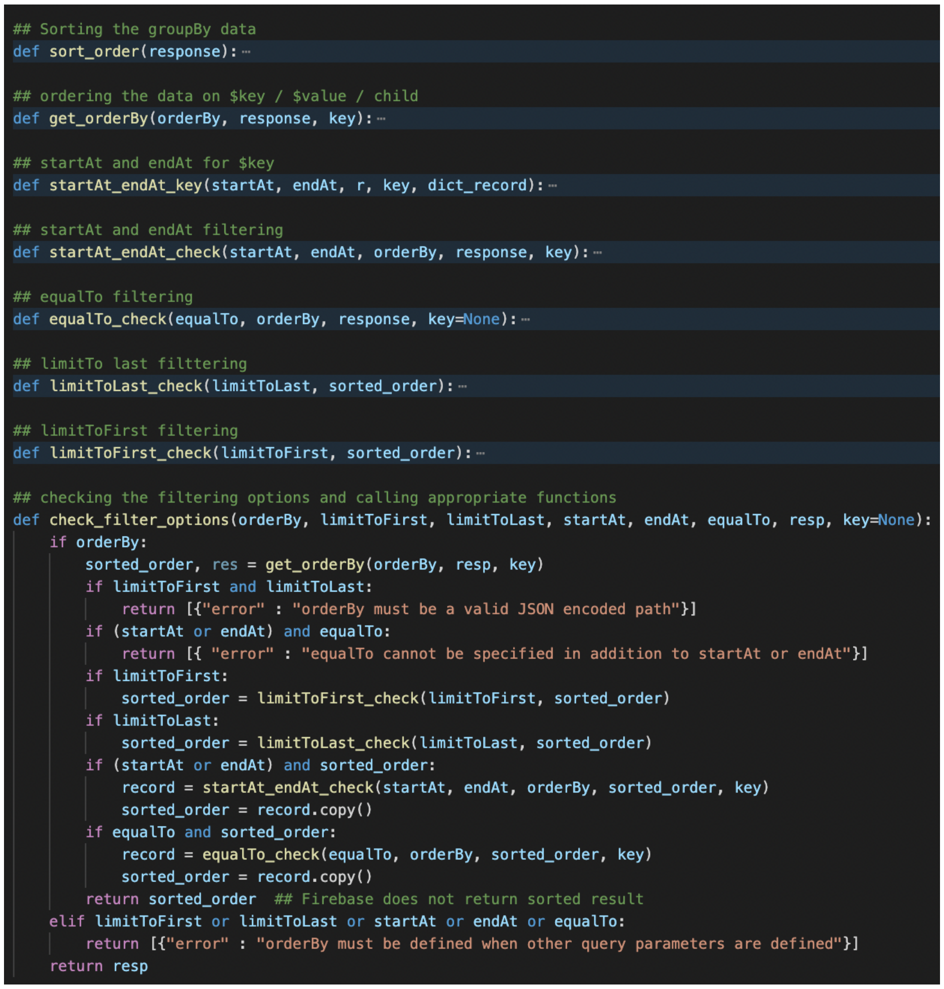
GET

Get the filtering options which are the arguments from the GET request. Get the details of data to find from the path. Find the data in MongoDB using the find method of MongoClient.



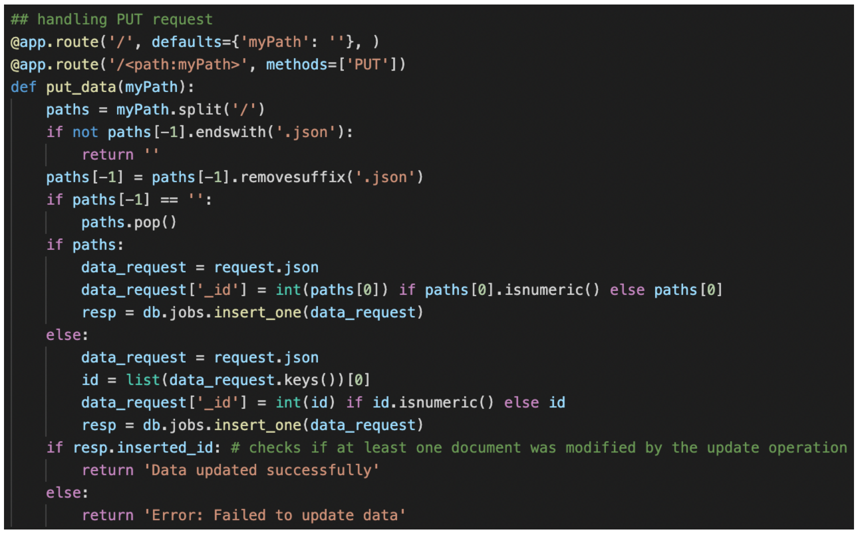
```python
## handling GET request
@app.route('/', defaults={'myPath': ''})
@app.route('/<path:myPath>', methods=['GET'])
def catch_all_get(myPath):
    app.json.sort_keys = True
    limitToFirst = request.args.get('limitToFirst', default=None, type=int)
    limitToLast = request.args.get('limitToLast', default=None, type=int)
    startAt = request.args.get('startAt')
    endAt = request.args.get('endAt')
    orderBy = request.args.get('orderBy')
    equalTo = request.args.get('equalTo')
    paths = myPath.split('/')
    if not paths[-1].endswith('.json'): ## if no .json at the end of url return empty
        return ''
    paths[-1] = paths[-1].removesuffix('.json')
    if paths[-1] == '': ## /.json type of url
        paths.pop()
    if paths:
        if len(paths) > 1: ## if paths is longer than 1 then projection is required to select specific data
            path_dict = create_projection(paths[1])
            resp = db.jobs.find({'_id': int(paths[0])}, path_dict) if paths[0].isnumeric() else db.jobs.find({'_id': paths[0]}, path_dict)
        else:
            resp = db.jobs.find({'_id': int(paths[0])}) if paths[0].isnumeric() else db.jobs.find({'_id': paths[0]})
    else: ## empty indicates all the records have to be returned
        resp = db.jobs.find({}).sort('_id')
    if paths:
        resp = check_filter_options(orderBy, limitToFirst, limitToLast, startAt, endAt, equalTo, resp, paths[-1])
    else:
        resp = check_filter_options(orderBy, limitToFirst, limitToLast, startAt, endAt, equalTo, resp)
    if resp:
        resp = list(resp)
    if resp and len(resp) == 1 and {} in resp:
        return ''
    if resp and len(resp) == 1:
        if 'error' not in resp[0]:
            resp = get_response(resp[0], paths) if len(paths) > 1 else resp[0] ## traverse till the path and return the data in that path
        else:
            resp = resp[0]
    if not resp:
        resp = None
    return jsonify(resp) ## sorts the returned json on keys: this is same as the default behaviour of Firebase
```

Check to see if orderBy exists.The possible values for this are ‘$key’, ‘$value’ or child key name. If it exists then orderBy based on it’s value and check all the other filtering options. Return the final filtered data. If there are other filtering options without orderBy return error.
After performing filtering, return the data using jsonify method which will serialize the data to JSON format and return Response object with the serialized data.
Check and filter based on the values of filtering options as shown below:


```python
## Sorting the groupBy data
def sort_order(response):⋯

## ordering the data on $key / $value / child
def get_orderBy(orderBy, response, key):⋯

## startAt and endAt for $key
def startAt_endAt_key(startAt, endAt, r, key, dict_record):⋯

## startAt and endAt filtering
def startAt_endAt_check(startAt, endAt, orderBy, response, key):⋯

## equalTo filtering
def equalTo_check(equalTo, orderBy, response, key=None):⋯

## limitTo last filttering
def limitToLast_check(limitToLast, sorted_order):⋯

## limitToFirst filtering
def limitToFirst_check(limitToFirst, sorted_order):⋯

## checking the filtering options and calling appropriate functions
def check_filter_options(orderBy, limitToFirst, limitToLast, startAt, endAt, equalTo, resp, key=None):
    if orderBy:
        sorted_order, res = get_orderBy(orderBy, resp, key)
        if limitToFirst and limitToLast:
            return [{"error" : "orderBy must be a valid JSON encoded path"}]
        if (startAt or endAt) and equalTo:
            return [{ "error" : "equalTo cannot be specified in addition to startAt or endAt"}]
        if limitToFirst:
            sorted_order = limitToFirst_check(limitToFirst, sorted_order)
        if limitToLast:
            sorted_order = limitToLast_check(limitToLast, sorted_order)
        if (startAt or endAt) and sorted_order:
            record = startAt_endAt_check(startAt, endAt, orderBy, sorted_order, key)
            sorted_order = record.copy()
        if equalTo and sorted_order:
            record = equalTo_check(equalTo, orderBy, sorted_order, key)
            sorted_order = record.copy()
        return sorted_order  ## Firebase does not return sorted result
    elif limitToFirst or limitToLast or startAt or endAt or equalTo:
        return [{"error" : "orderBy must be defined when other query parameters are defined"}]
    return resp
```

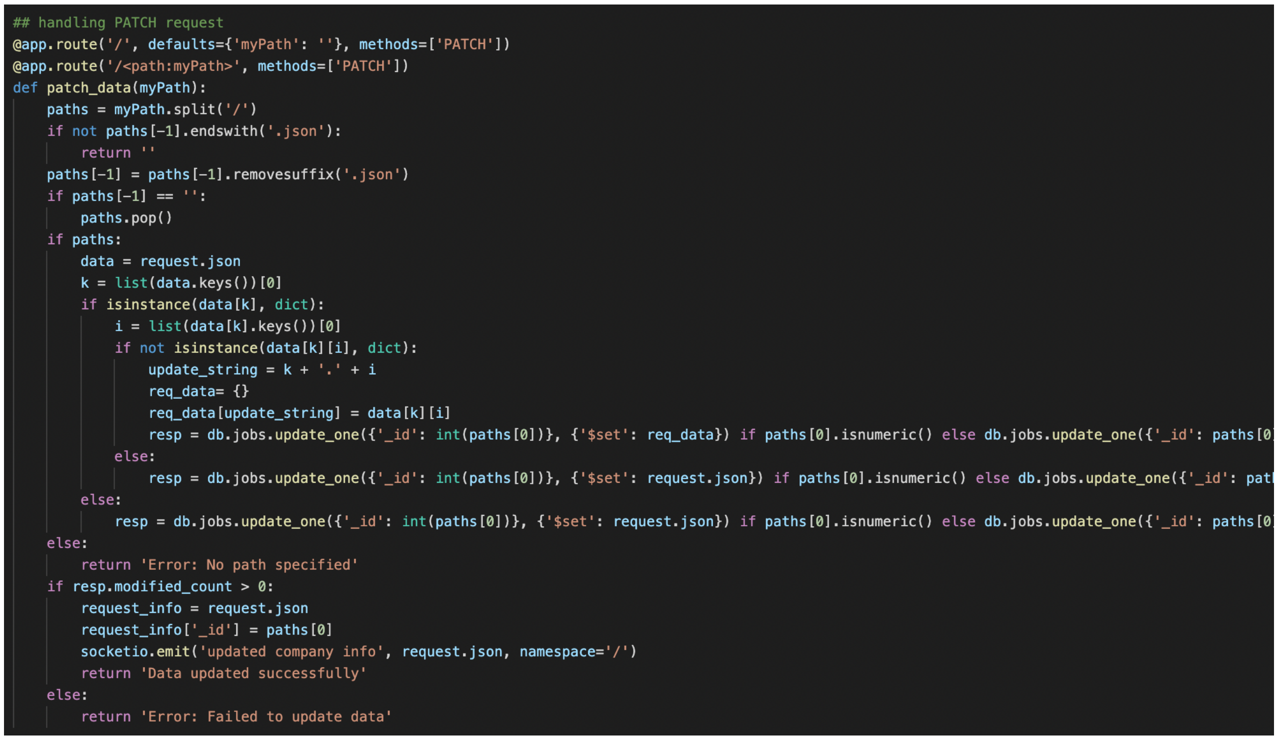
PUT

Get the id from the path. This is the _id value. Create the data_request dictionary with_id set. Put the data_request in MongoDB using insert_one method of MongoClient. Return the status of PUT - Success or Failure.


```python
## handling PUT request
@app.route('/', defaults={'myPath': ''}, )
@app.route('/<path:myPath>', methods=['PUT'])
def put_data(myPath):
    paths = myPath.split('/')
    if not paths[-1].endswith('.json'):
        return ''
    paths[-1] = paths[-1].removesuffix('.json')
    if paths[-1] == '':
        paths.pop()
    if paths:
        data_request = request.json
        data_request['_id'] = int(paths[0]) if paths[0].isnumeric() else paths[0]
        resp = db.jobs.insert_one(data_request)
    else:
        data_request = request.json
        id = list(data_request.keys())[0]
        data_request['_id'] = int(id) if id.isnumeric() else id
        resp = db.jobs.insert_one(data_request)
    if resp.inserted_id: # checks if at least one document was modified by the update operation
        return 'Data updated successfully'
    else:
        return 'Error: Failed to update data'
```

<u>PATCH</u>

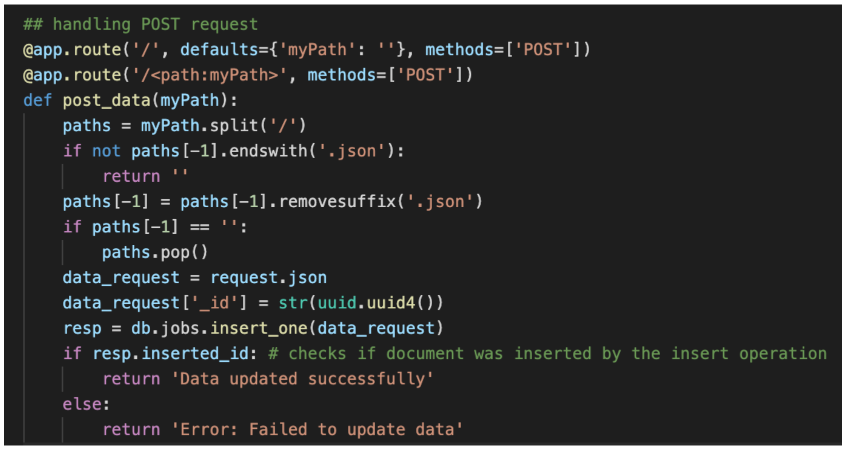
Get the id from the path. This is the _id value. This represents the data to be updated. Update the data in MongoDB using update_one method of MongoClient while taking care of nested json updates . Return the status of PATCH - Success or Failure.



```python
## handling PATCH request
@app.route('/', defaults={'myPath': ''}, methods=['PATCH'])
@app.route('/<path:myPath>', methods=['PATCH'])
def patch_data(myPath):
    paths = myPath.split('/')
    if not paths[-1].endswith('.json'):
        return ''
    paths[-1] = paths[-1].removesuffix('.json')
    if paths[-1] == '':
        paths.pop()
    if paths:
        data = request.json
        k = list(data.keys())[0]
        if isinstance(data[k], dict):
            i = list(data[k].keys())[0]
            if not isinstance(data[k][i], dict):
                update_string = k + '.' + i
                req_data= {}
                req_data[update_string] = data[k][i]
                resp = db.jobs.update_one({'_id': int(paths[0])}, {'$set': req_data}) if paths[0].isnumeric() else db.jobs.update_one({'_id': paths[0
            else:
                resp = db.jobs.update_one({'_id': int(paths[0])}, {'$set': request.json}) if paths[0].isnumeric() else db.jobs.update_one({'_id': pat
        else:
            resp = db.jobs.update_one({'_id': int(paths[0])}, {'$set': request.json}) if paths[0].isnumeric() else db.jobs.update_one({'_id': paths[0
    else:
        return 'Error: No path specified'
    if resp.modified_count > 0:
        request_info = request.json
        request_info['_id'] = paths[0]
        socketio.emit('updated company info', request.json, namespace='/')
        return 'Data updated successfully'
    else:
        return 'Error: Failed to update data'
```

<u>POST</u>

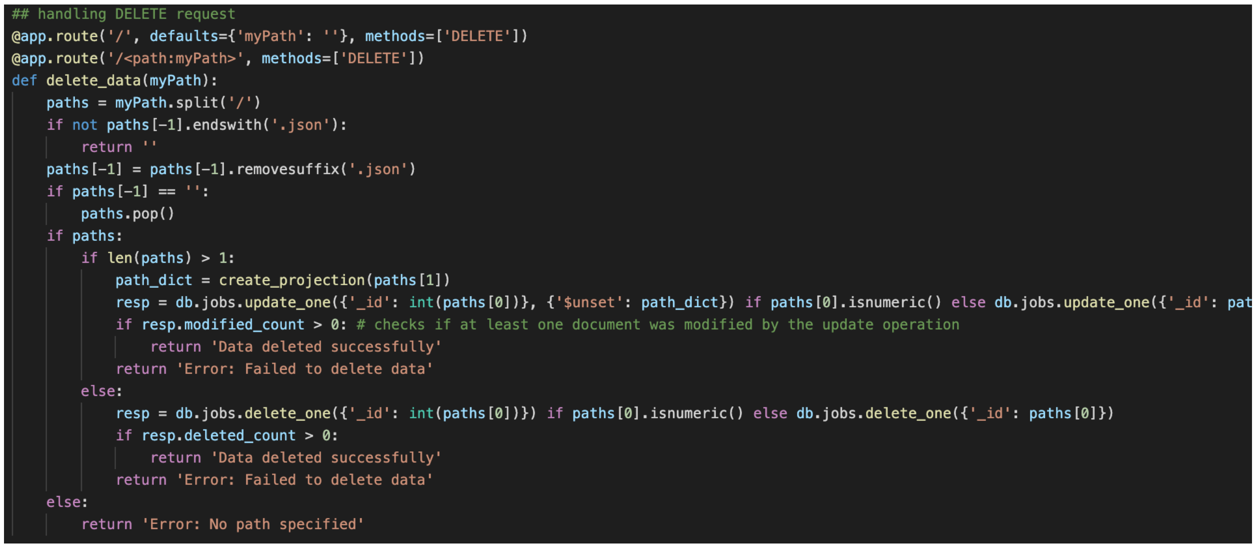
Insert the data in MongoDB using inser_one method from MongoClient with new _id created using uuid.uuid4() . Return the status of POST - Success or Failure.



```python
## handling POST request
@app.route('/', defaults={'myPath': ''}, methods=['POST'])
@app.route('/<path:myPath>', methods=['POST'])
def post_data(myPath):
    paths = myPath.split('/')
    if not paths[-1].endswith('.json'):
        return ''
    paths[-1] = paths[-1].removesuffix('.json')
    if paths[-1] == '':
        paths.pop()
    data_request = request.json
    data_request['_id'] = str(uuid.uuid4())
    resp = db.jobs.insert_one(data_request)
    if resp.inserted_id: # checks if document was inserted by the insert operation
        return 'Data updated successfully'
    else:
        return 'Error: Failed to update data'
```

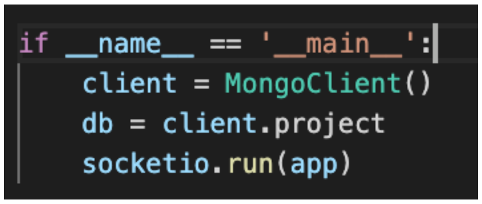
DELETE

Delete the data using delete_one method of MongoClient. Return the status of delete - Success or Failure.



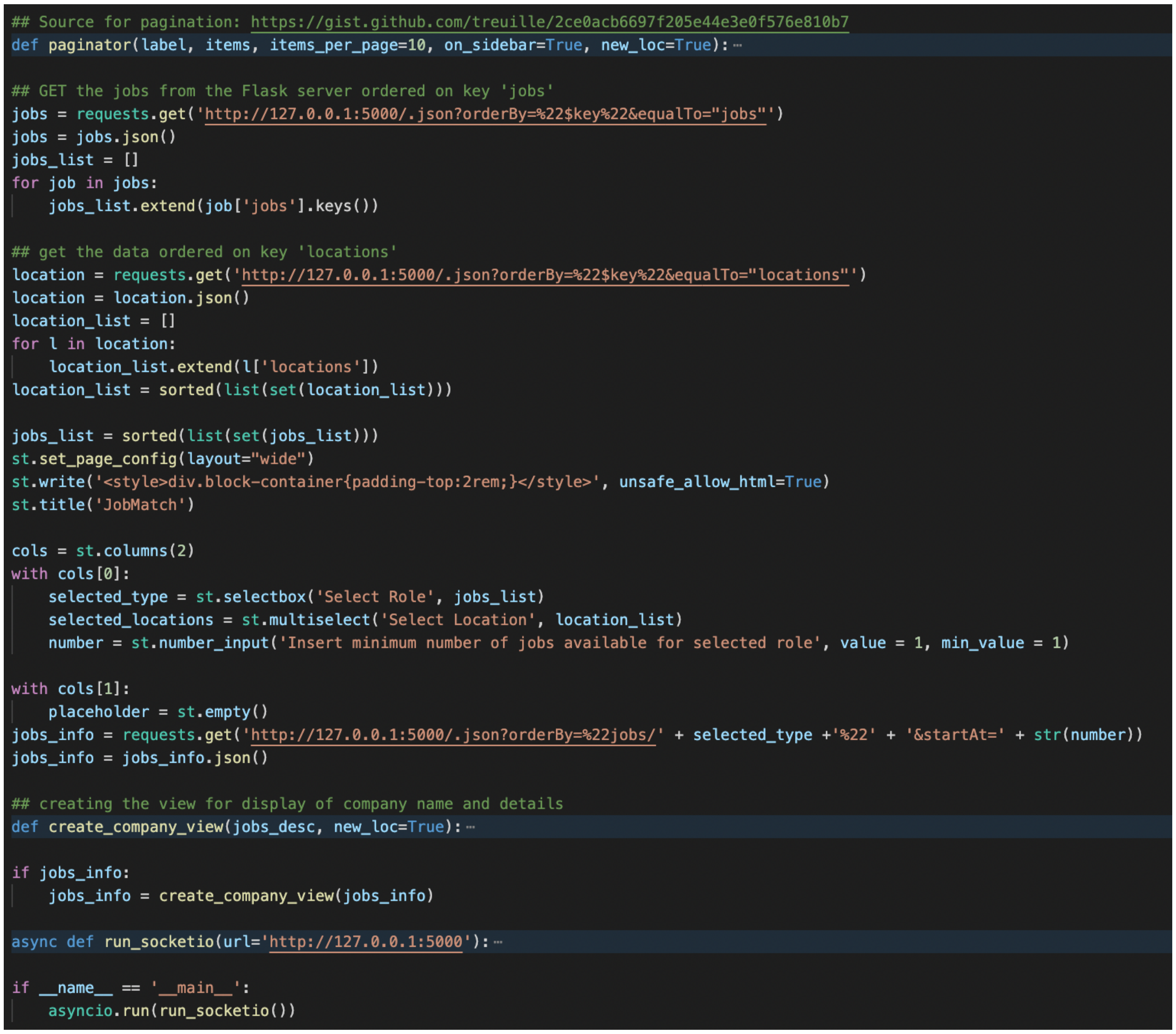
```python
## handling DELETE request
@app.route('/', defaults={'myPath': ''}, methods=['DELETE'])
@app.route('/<path:myPath>', methods=['DELETE'])
def delete_data(myPath):
    paths = myPath.split('/')
    if not paths[-1].endswith('.json'):
        return ''
    paths[-1] = paths[-1].removesuffix('.json')
    if paths[-1] == '':
        paths.pop()
    if paths:
        if len(paths) > 1:
            path_dict = create_projection(paths[1])
            resp = db.jobs.update_one({'_id': int(paths[0])}, {'$unset': path_dict}) if paths[0].isnumeric() else db.jobs.update_one({'_id': pat
            if resp.modified_count > 0: # checks if at least one document was modified by the update operation
                return 'Data deleted successfully'
            return 'Error: Failed to delete data'
        else:
            resp = db.jobs.delete_one({'_id': int(paths[0])}) if paths[0].isnumeric() else db.jobs.delete_one({'_id': paths[0]})
            if resp.deleted_count > 0:
                return 'Data deleted successfully'
            return 'Error: Failed to delete data'
    else:
        return 'Error: No path specified'
```

Run the Flask-socketio server



```python
if __name__ == '__main__':
    client = MongoClient()
    db = client.project
    socketio.run(app)
```

Implementing Frontend - Streamlit Framework



```python
## Source for pagination: https://gist.github.com/treuille/2ce0acb6697f205e44e3e0f576e810b7
def paginator(label, items, items_per_page=10, on_sidebar=True, new_loc=True):⋯

## GET the jobs from the Flask server ordered on key 'jobs'
jobs = requests.get('http://127.0.0.1:5000/.json?orderBy=%22$key%22&equalTo="jobs"')
jobs = jobs.json()
jobs_list = []
for job in jobs:
    jobs_list.extend(job['jobs'].keys())

## get the data ordered on key 'locations'
location = requests.get('http://127.0.0.1:5000/.json?orderBy=%22$key%22&equalTo="locations"')
location = location.json()
location_list = []
for l in location:
    location_list.extend(l['locations'])
location_list = sorted(list(set(location_list)))

jobs_list = sorted(list(set(jobs_list)))
st.set_page_config(layout="wide")
st.write('<style>div.block-container{padding-top:2rem;}</style>', unsafe_allow_html=True)
st.title('JobMatch')

cols = st.columns(2)
with cols[0]:
    selected_type = st.selectbox('Select Role', jobs_list)
    selected_locations = st.multiselect('Select Location', location_list)
    number = st.number_input('Insert minimum number of jobs available for selected role', value = 1, min_value = 1)

with cols[1]:
    placeholder = st.empty()
jobs_info = requests.get('http://127.0.0.1:5000/.json?orderBy=%22jobs/' + selected_type +'%22' + '&startAt=' + str(number))
jobs_info = jobs_info.json()

## creating the view for display of company name and details
def create_company_view(jobs_desc, new_loc=True):⋯

if jobs_info:
    jobs_info = create_company_view(jobs_info)

async def run_socketio(url='http://127.0.0.1:5000'):⋯

if __name__ == '__main__':
    asyncio.run(run_socketio())
```

Get the jobs data required from the Flask server and retrieve the required fields from it. Create 2 columns:

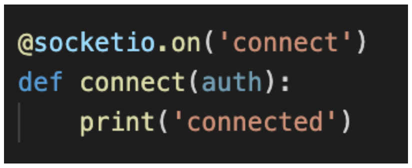
1. Filtering options - Job Role, Job Location and Number of jobs available in the specified role in the company
2. Display the Companies as per filtering options and paginate them

Run the socketio to listen for updates from the server.

Run Web App - streamlit run frontend.py

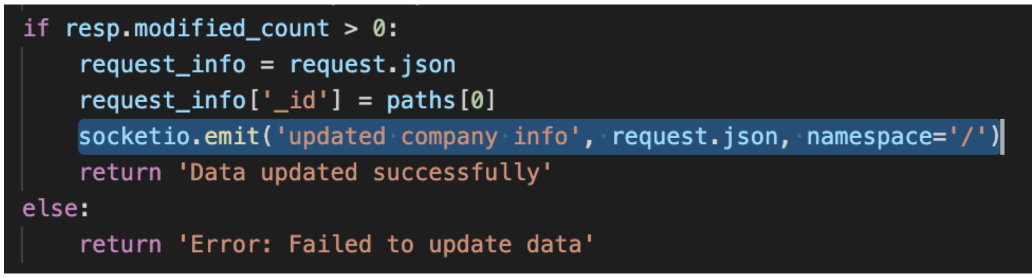
Implementing Websocket

Server - Flask-SocketIO



```
@socketio.on('connect')
def connect(auth):
    print('connected')
```

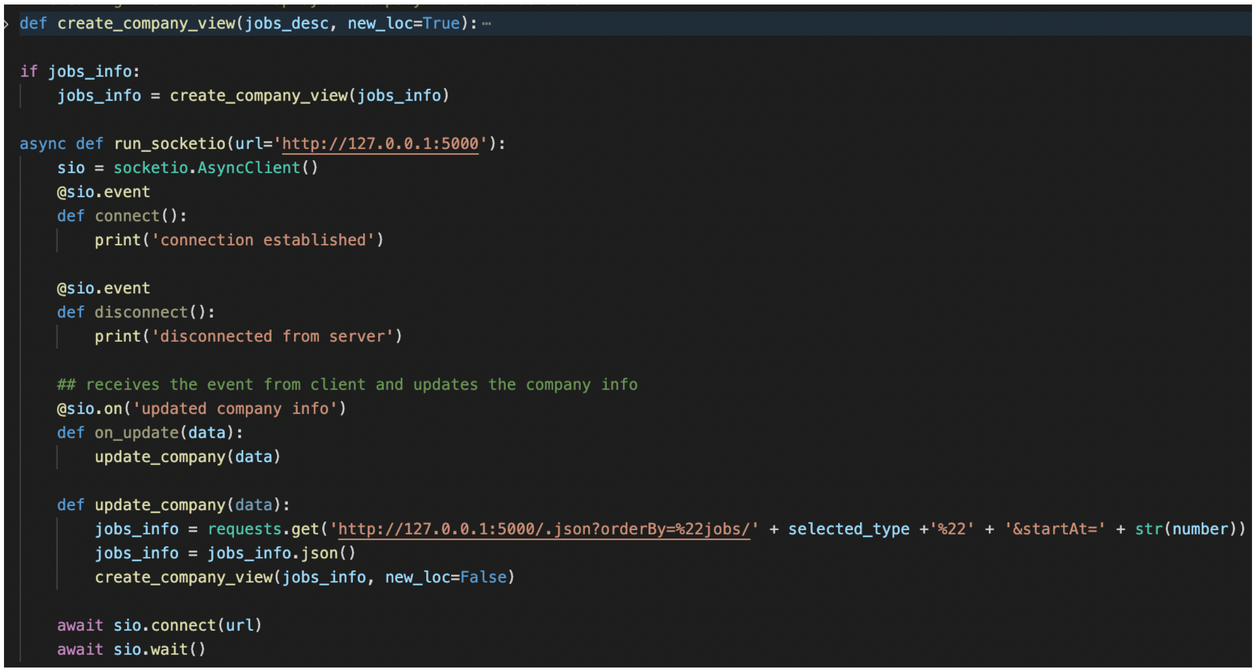
Printing connected wherever a connect request received.



```
if resp.modified_count > 0:
    request_info = request.json
    request_info['_id'] = paths[0]
    socketio.emit('updated company info', request.json, namespace='/')
    return 'Data updated successfully'
else:
    return 'Error: Failed to update data'
```

If an update is successful from PATCH request ‘updated company info’ event is emitted.

Client - socketio



```
def create_company_view(jobs_desc, new_loc=True):…

if jobs_info:
    jobs_info = create_company_view(jobs_info)

async def run_socketio(url='http://127.0.0.1:5000'):
    sio = socketio.AsyncClient()
    @sio.event
    def connect():
        print('connection established')

    @sio.event
    def disconnect():
        print('disconnected from server')

    ## receives the event from client and updates the company info
    @sio.on('updated company info')
    def on_update(data):
        update_company(data)

    def update_company(data):
        jobs_info = requests.get('http://127.0.0.1:5000/.json?orderBy=%22jobs/' + selected_type +'%22' + '&startAt=' + str(number))
        jobs_info = jobs_info.json()
        create_company_view(jobs_info, new_loc=False)

    await sio.connect(url)
    await sio.wait()
```

The client connects to the Flask-socketio server and listens for the ‘updated company info’ event. When the event is received it updates the webpage.
Use case: A PATCH request to increase the number of jobs to 460 in the ‘Investor’ role is received from ‘Wise’ company. If the update is successful, the client emits an ‘updated company info’ event. The client receives it and GET the updated data from Flask and using this data adds another expander in the display column for filtering options set to
Role - Investor
Minimum number of jobs available for selected role: 450

Learning Experiences & Future Improvements

Throughout this project, we not only learned about data management as a whole, but also researched about cutting-edge data management concepts, systems, and techniques. We learned more about Firebase and MongoDB by choosing this topic.

One thing that we can improve upon our Web App is to add more filters, and perhaps add in a map to show the locations. While those features go beyond the scope of this course, it may be interesting to further develop this project.