**PROJECT REPORT – TWITTER**

Group 7

Aswin Prasad

Farha Kauser

Jayasurya Pinaki

Maaz Sirkhot

Shubhangi Yadav

## Member Contribution:

* **Aswin Prasad**

Aswin has been mainly responsible for deployment of application and implementation of Kafka. From configuring multiple AWS EC2 instances for backend and frontend to testing the various scenarios in JMeter along with Jayasurya. A good collaborator he is, Aswin has worked with Farha to implement Analytics and complete user interface for the same. Also, Aswin emphasizes on achieving perfection whether in backend or frontend to ensure all the bugs are resolved.

* **Farha Kauser**

Farha contributed in developing backend APIs and handling complex tasks related to Analytics Dashboard, data management and extraction. Additionally, she has performed end to end Enzyme testing and regression testing to highlight critical bugs in the application. With great understanding of requirements, Farha ensures all requirements are accurately understood and communicated to all team members.

* **Jayasurya Pinaki**

Jayasurya has played key role in smooth functioning of team meetings, regular reviews, maintaining sprints, code integration and designing the frontend skeleton for the web interface. Well versed with both backend and frontend, he has been a bridge between identifying the processes and providing constant suggestions for optimizing the application. Frontend development has been efficient all due to his efforts and fluency in HTML and bootstrap.

* **Maaz Sirkhot**

Maaz has been key member for designing the database structure for SQL and MongoDB. With preference for backend development, Maaz has written code for critical APIs related to Messages, fetching tweets for various scenarios, delete user profile to delete data from each collection. Furthermore, Maaz has taken complete ownership of Redis implementation and designing Messages webpage on User Interface.

* **Shubhangi Yadav**

With expertise in backend technologies, Shubhangi prepared the code boilerplate, provided transition on better coding practices, following industry standards, managing authorization and authentication and most importantly assisted in designing the database structure along with Maaz. Shubhangi is also responsible for writing Lists APIs and input validations in backend to prevent incorrect data to be written into database

## GitLab URL:

<https://gitlab.com/jayasurya17/273-twitter> (Currently Private)

## Object Management Policy:

In this distributed application, there are four major objects namely Users, Tweets, Lists, and Messages. Each object is maintained with the help of a model schema storing the data in databases. These models define the fields each document will have as well as how they are related to one another. Objects are loosely coupled with each other thereby providing enough flexibility to act and modify independently as required.

Users:

This object is responsible for storing personal information, bookmarks and authentication information. User object is uniquely identified using the \_id, username and email/phone fields. These fields are unique. Although \_id is auto generated by MongoDB, other fields are maintained uniquely by utilities which are called every time data is written into the database. These utilities check the database to find if there is a similar entry and if there is, it doesn’t allow to write a duplicate entry into the database. Likewise, username field, email field, phone field are unique in the database.

The schema of User object contains following fields:

\_id, name, username, city, state, zipcode, imageURL, description, password (hashed), isActive, bookmarks, followingCount, followersCount, views, jwtToken, phone, email, dateofBirth.

Each field in the object is validated thoroughly. Especially the password field is hashed using bcrypt hashing mechanism. This is wrapped as a pre-condition for inserting a document in the database. This password is hashed with 10 rounds of salt thereby making it secure. This user object is continuously retrieved and checked for verifying the userId (\_id) and gain other user details irrespective of the task to be performed. Furthermore, JWT Passport strategy uses this object to verify the jwtToken each time API request is received. This strategy checks for jwt Tokens in the database and allows upto four simultaneous login sessions. In case a user tries to login from fifth session, the first session is terminated thus following the FIFO mechanism to manage sessions.

Tweets:

Tweets object stores the tweet information including the image uploaded in tweet, comments on tweets, tweet owner details etc. Tweets object is uniquely identified by only one field \_id which is used and referenced everywhere else required. The tweet object stores the tweet information, retweet information, likes count, comments. This object however does not store all the information related to tweets but some of the tweet information is stored in SQL database which keeps record of the users who liked the tweet. With frequent Tweet Like activity, the write operations are abundant however it is only on one field which makes retrieving and updating the document costly process. Therefore, the likes are maintained in a different table in relational database SQL. These two are bound together through the TweetId (\_id) which although isn’t a foreign key, it is manually checked in MongoDB before being written in SQL.

The schema of Tweet object contains following fields:

\_id, userId, userName, userImageURL, tweetDate, imageURL, isRetweet, isActive, originalTweetId, originalUserId, originalUserName, originalUserImageURL, originalBody, likeCount, commentCount, retweetCount, viewsCount, comments, isDeleted.

The tweet object can be accessed by all the APIs and the schema defines the types of each field and data that can be inserted or updated in the database. The tweet object is heavy weight and contains lots of information and at each given point of time, retrieving thousands of tweets means retrieving large amount of data. Managing this data is complex and we tried to keep minimal nested objects. Hence, only comments array is a list of nested objects which hold the comment data and basic information about the respective user. Tweets are retrieved mainly by referring the tweetId or in some cases other fields are used such as userName, isRetweet, tweetDate etc depending upon the requirement.

Messages:

Messages is a very independent and loosely linked object among all the objects. Rightly so, it is the least affected of the changes in other objects and very simple to update and manage. Retrieving data from Messages object is straight forward and doesn’t contain lot of fields. The ideology of having an inbox containing all the conversations is majorly replicated in the design of the object and storage of data. As a result, very less amount of data manipulations. In the messages object, the data is stored in two fields which contain array of objects. One stores list of participants in conversation and other stores the messages. Participants is array of objects where object contains basic user information like userName, userId and userImageURL. Body field stores the message text, sender’s userName and timestamp. Designing such a model allows to store messages and manage it in the same format it is required.

Messages schema fields:

Participants, Body

The Messages object also has a validation where the user information is first checked in user object before inserting it in the message schema. This ensures messages are created only for the valid users. In case a document is already present for a given set of participants, the further messages are simply pushed to the body of that document. Only when a participant pair is not present, new document is created.

Lists:

Lists is most similar to Tweet object and mainly stores the data related to personalized lists created by users. Users can create their own list and add members to the list. Hence, subscribers of this list will be able view tweets from members of the list. Therefore, this object contains data referenced from user as well as Tweet object. The lists also keep count of number of members in the list and their details as well as count of subscribers. Details of subscribers are not directly stored in the object but their userId is stored in SQL database similar to tweet’s ‘like’ data.

Data in lists can be updated by APIs and only the list owner is allowed to modify the members of the list. Hence, each time members are updated a check is performed to ensure the owner is modifying it. This is very important to maintain accuracy.

List schema contains following fields:

\_id, listName, listDescription, listOwnerId, ownerId, ownerImage, ownerUserName, ownerName, isActive, noofMembers, membersId, noofSubscribers

Similar to other objects, even for this object the data is validated before insertion. User data is checked for ownerId, ownerImage, ownerName etc. Even to add members in this, availability is checked thoroughly before creating an entry. A member can be added only once, and a subscriber can subscribe to list once. If a member is already present or a user has already subscribed, the same task cannot be executed twice.

## Heavyweight Resources:

The distributed system heavily lies on the functioning of API requests and database calls which utilize most resources. In this application, fetching data from databases is the most frequent task especially the user data. Although we have critical user data stored in all collections, we still must validate the data by referencing the User object. Therefore, inside each API we must fetch the user information. Fetching the user information every time causes more active connections and as a result slower response time. To tackle this, we have implemented Redis in-memory cache to store user profile information. Redis hash set stored the key value pair data stored for User object. Hence, for fetching user information and quick retrieval of data, Redis significantly reduces the response times.

Other heavyweight resources including uploading images for profile picture and in tweets. Storing of images is an overhead and as number of images increase, we need a dynamic storage which can be expanded as required. By default, NodeJS provides multer module in npm to store images in a static page on web server. However, due to size limitations and computation risks, we have implemented Amazon’s S3 bucket where all the images are stored and referenced. There is a similar library in NPM called Multer S3 and aws-sdk which together work to upload the file in request on AWS bucket and provide the link for the same. This link is therefore called as a resource for images on web interface.

Furthermore, Kafka is the heaviest of all resources and responsible for the high response time which causes lesser number of requests being processed. Kafka manages a messaging queue through which all the requests pass. Every request is received by Kafka first and according to the topic, it is sent to respective topic handler. This extra processing causes more higher response time, but this makes sure no request is timed out or dropped without a response. Each request is queued and as a result provided a response. However, although the response time has increased, the system reliability and request success rate has increased phenomenally. The distributed application can now process thousands of requests with immense efficiency which would have been difficult to achieve without Kafka.

## Database write operations:

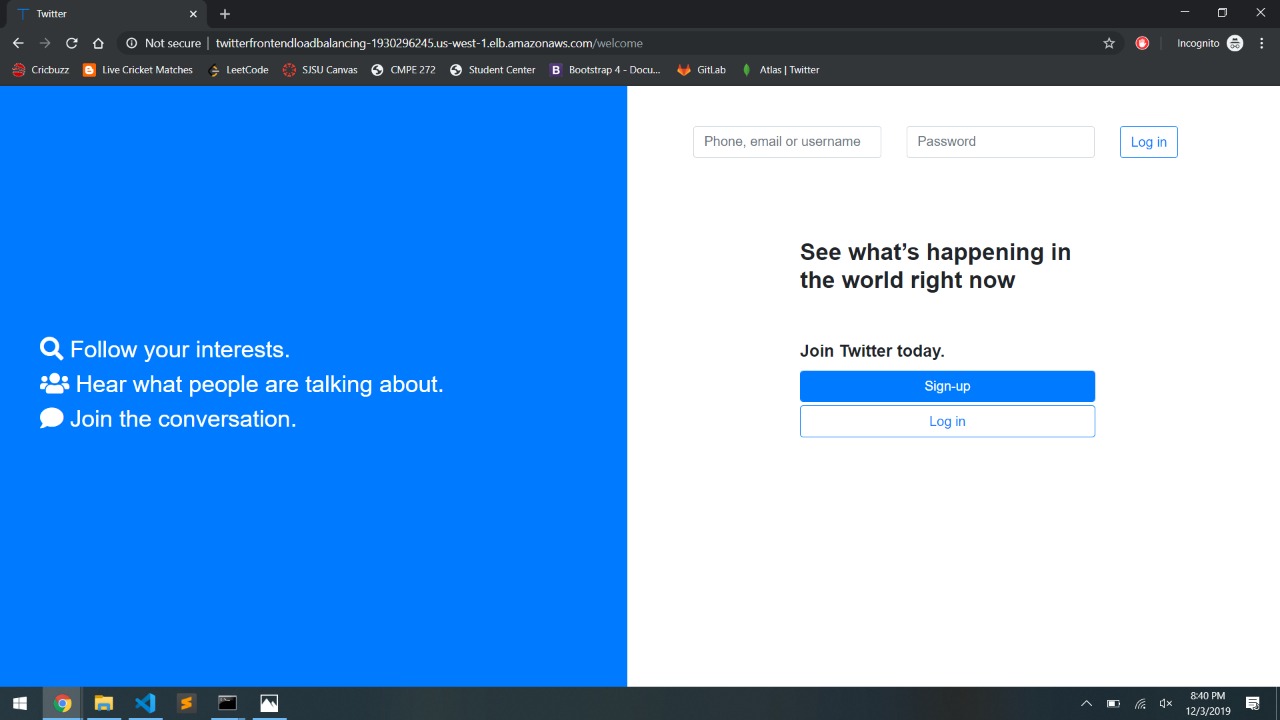
Inserting and updating data in the database requires resources and it can slow down or even crash the application if not performed correctly. Hence, it is of utmost importance to validate and organize the data before written into database. In the application, data validations are put not only on front end but also on the backend using Joi module of npm. This module helps in checking for required fields, data types and data format. In case the data is not received in the required format, an error is thrown, and failed response is provided. This prevents any further crashing of data and eventually the database. Hence, only validated data is written into database.

Additionally, the database design is such that frequent write operations are handled carefully. Wherever, frequent write operation in anticipated, the data is stored in MySQL which allows systematic and faster operation than MongoDB. Other data which is read frequently is stored in MongoDB and the user profile data which must be fetched for almost every API request, is cached in Redis.

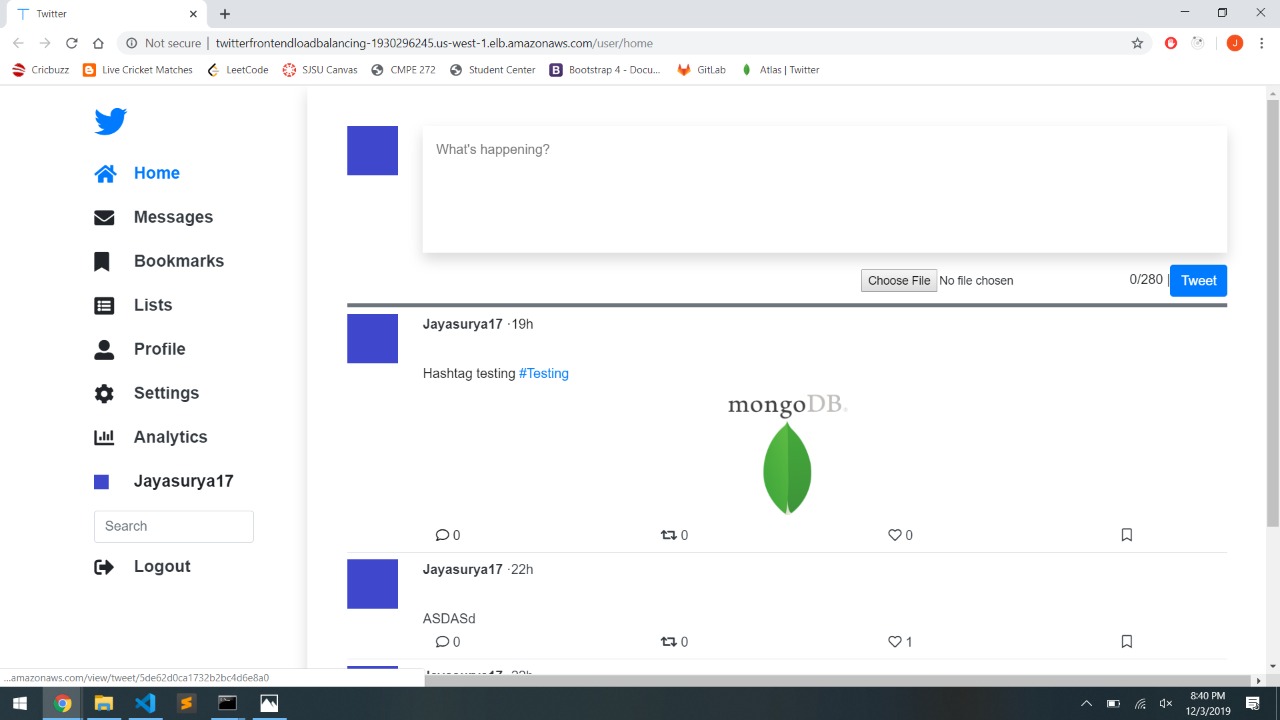
## Application Screenshots:

Some of the screenshots for user interface pages are given below.

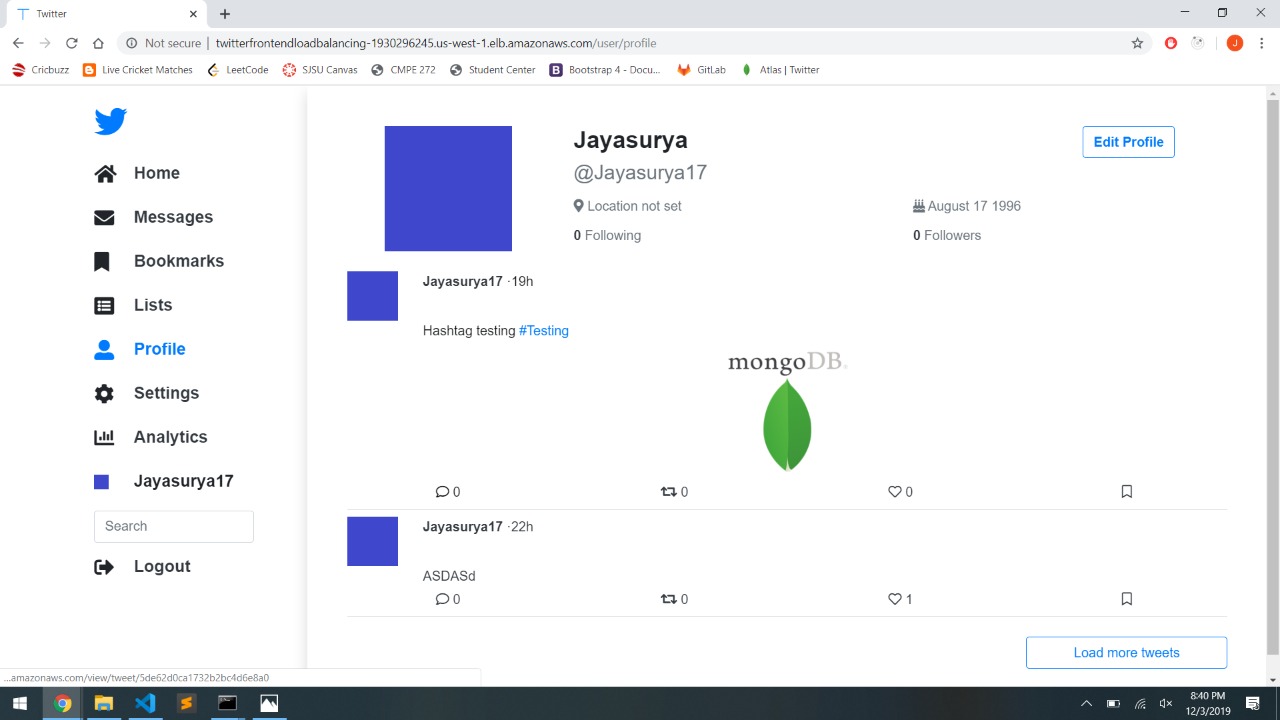
Welcome Page:



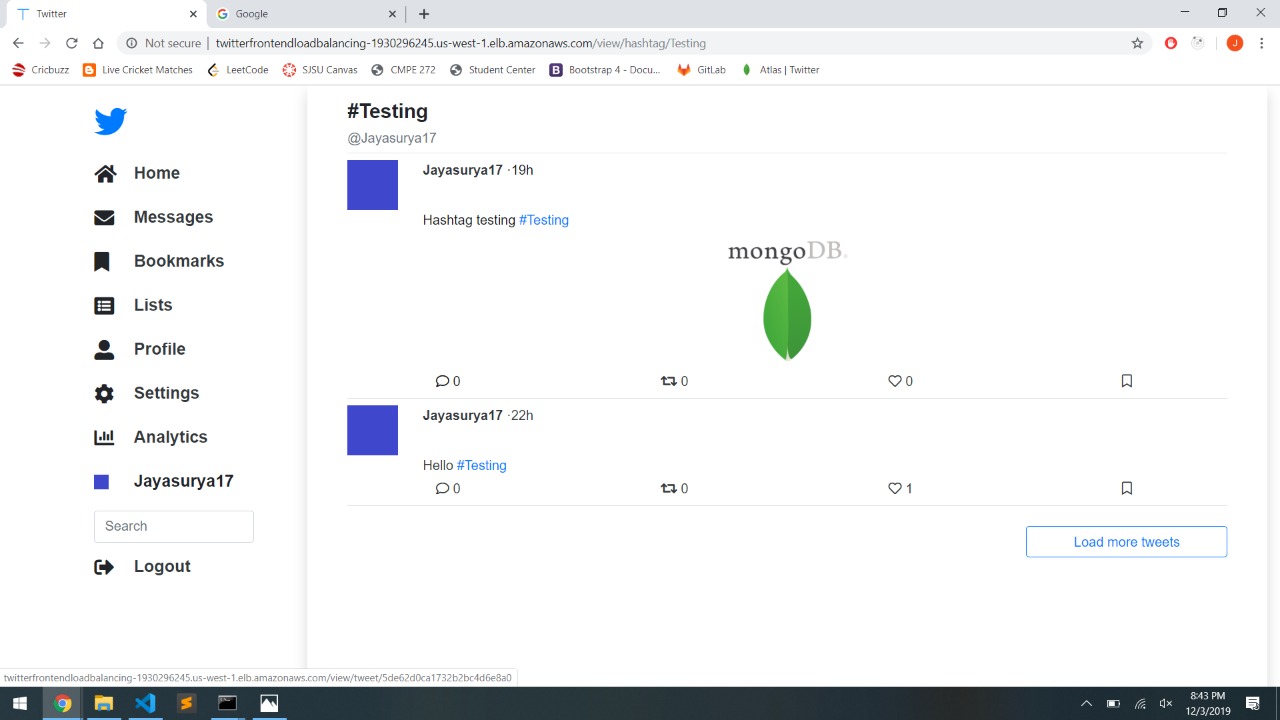
Home:



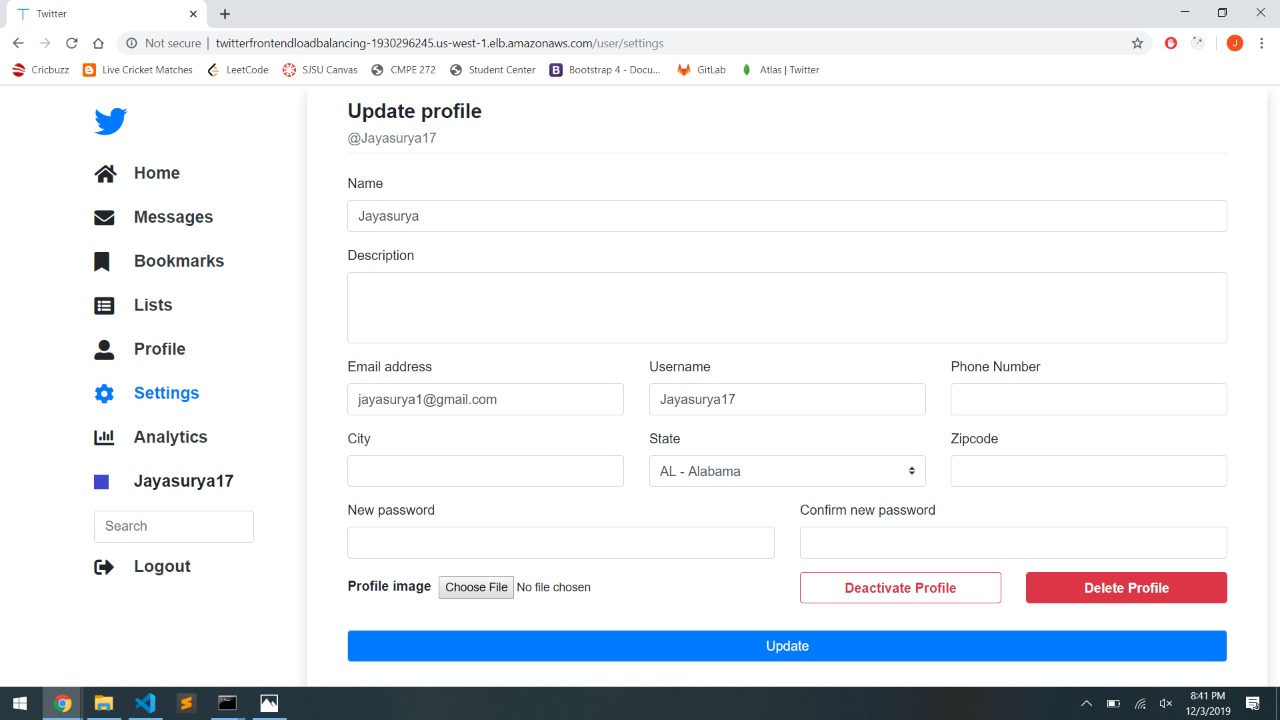
Profile:



Search Hashtag:

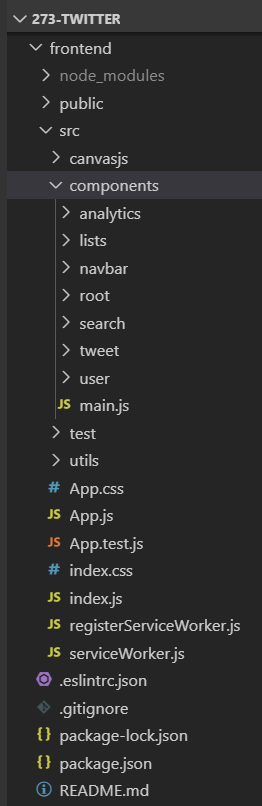
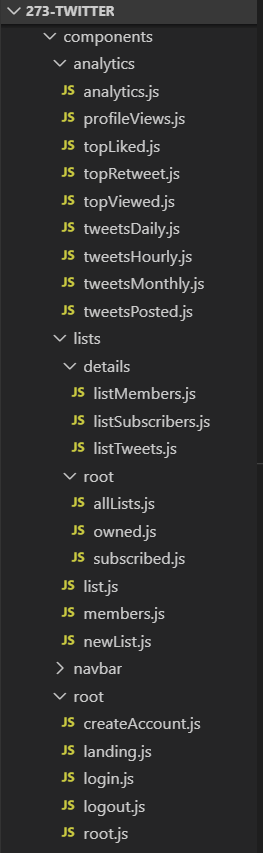
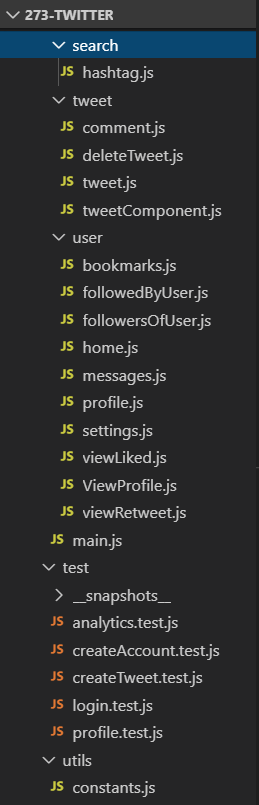


Settings:



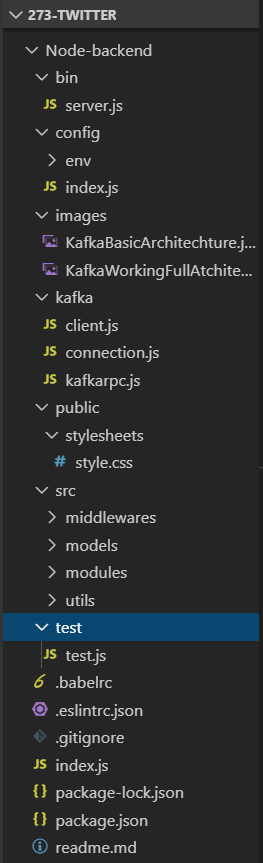
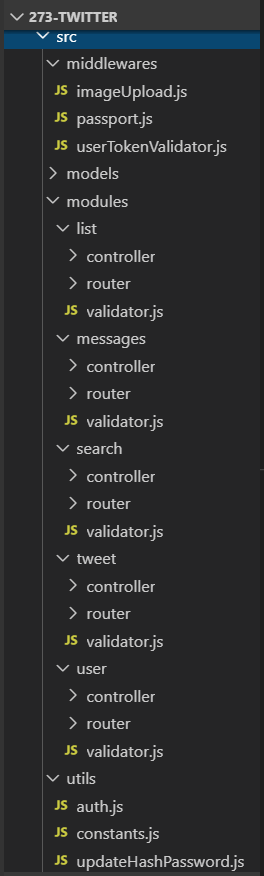
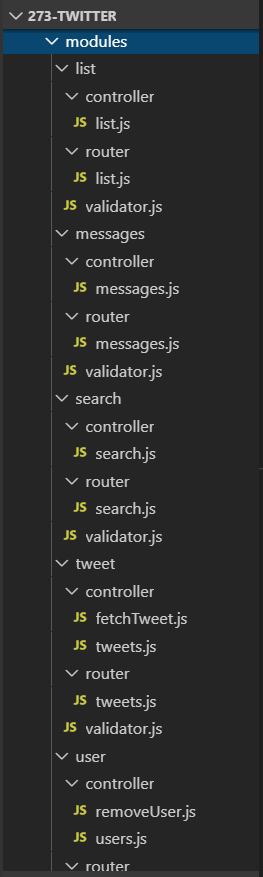
## Code Listing – Client:

Folder Structure:

## Code Listing – Server

Folder Structure:

## Code Listing – Server Implementation of entity objects:

**Kafka RPC:**

var crypto = require('crypto');

var conn = require('./connection');

var TIMEOUT=30000; //time to wait for response in ms

var self;

exports = module.exports =  KafkaRPC;

// It connects to kafka broker, fires request to kafka broker, gets response from broker, connects with consumer.

function KafkaRPC(){

    self = this;

    this.connection = conn;

    this.requests = {}; //hash to store request in wait for response

    this.response\_queue = false; //placeholder for the future queue

    this.producer = this.connection.getProducer();

}

KafkaRPC.prototype.makeRequest = function(topic\_name, content, callback){

    self = this;

    //generate a unique correlation id for this call

    var correlationId = crypto.randomBytes(16).toString('hex');

    //create a timeout for what should happen if we don't get a response

    var tId = setTimeout(function(corr\_id){

        //if this ever gets called we didn't get a response in a

        //timely fashion

        console.log('timeout');

        callback(new Error("timeout " + corr\_id));

        //delete the entry from hash

        delete self.requests[corr\_id];

    }, TIMEOUT, correlationId);

    //create a request entry to store in a hash

    var entry = {

        callback:callback,

        timeout: tId //the id for the timeout so we can clear it

    };

    //put the entry in the hash so we can match the response later

    self.requests[correlationId]=entry;

    //make sure we have a response topic

    self.setupResponseQueue(self.producer,topic\_name,function(){

        console.log('in response');

        //put the request on a topic

        var payloads = [

            { topic: topic\_name, messages: JSON.stringify({

                correlationId:correlationId,

                replyTo:'response\_topic',

                data:content})}

        ];

        console.log('in response1');

        console.log(self.producer.ready);

        self.producer.send(payloads, function(err, data){

            console.log('in response2');

            if(err)

                console.log(err);

            console.log(data);

        });

    });

};

KafkaRPC.prototype.setupResponseQueue = function(producer,topic\_name, next){

    //don't mess around if we have a queue

    if(this.response\_queue) return next();

    console.log('1');

    self = this;

    //subscribe to messages

    var consumer = self.connection.getConsumer('response\_topic');

    consumer.on('message', function (message) {

        console.log('msg received');

        var data = JSON.parse(message.value);

        //get the correlationId

        var correlationId = data.correlationId;

        //is it a response to a pending request

        if(correlationId in self.requests){

            //retrieve the request entry

            var entry = self.requests[correlationId];

            //make sure we don't timeout by clearing it

            clearTimeout(entry.timeout);

            //delete the entry from hash

            delete self.requests[correlationId];

            //callback, no err

            entry.callback(null, data.data);

        }

    });

    self.response\_queue = true;

    console.log('returning next');

    return next();

};

**Kafka Connection:**

var kafka = require('kafka-node');

function ConnectionProvider() {

    this.getConsumer = function(topic\_name) {

        this.client = new kafka.Client("localhost:2181");

        this.kafkaConsumerConnection = new kafka.Consumer(this.client,[ { topic: topic\_name, partition: 0} ]);

        this.client.on('ready', function () { console.log('client ready!') })

        return this.kafkaConsumerConnection;

    };

    //Code will be executed when we start Producer

    this.getProducer = function() {

        if (!this.kafkaProducerConnection) {

            this.client = new kafka.Client("localhost:2181");

            var HighLevelProducer = kafka.HighLevelProducer;

            this.kafkaProducerConnection = new HighLevelProducer(this.client);

            console.log('producer ready');

        }

        return this.kafkaProducerConnection;

    };

}

exports = module.exports = new ConnectionProvider;

Search object listing:

'use strict'

import Tweets from '../../../models/mongoDB/tweets'

import Users from '../../../models/mongoDB/users'

import mongoose from 'mongoose'

import constants from '../../../utils/constants'

import \_ from 'lodash'

const responseFormer = (status, message) => {

    return {status: status, message: message}

}

/\*\*

 \* Search for tweets with given hashtag.

 \* @param  {Object} req request object

 \* @param  {Object} res response object

 \*/

exports.hashtagSearch = async (req, res) => {

    try {

        const tweetsByHashtag = await Tweets.find({

            originalBody: {

                $regex: new RegExp('^(.\* )?#' + req.params.hashtag + '( .\*)?$', "i")

            } , isActive : true

        })

            .sort({ \_id: -1 })

            .skip(parseInt(req.query.start))

            .limit(parseInt(req.query.count))

        return responseFormer(constants.STATUS\_CODE.CREATED\_SUCCESSFULLY\_STATUS, tweetsByHashtag)

    } catch (error) {

        console.log(`Error while creating user ${error}`)

        return responseFormer(constants.STATUS\_CODE.INTERNAL\_SERVER\_ERROR\_STATUS, error.message)

    }

}

/\*\*

 \* Fetch user profile details based on userid and increase the view count of it.

 \* @param  {Object} req request object

 \* @param  {Object} res response object

 \*/

exports.fetchProfile = async (req, res) => {

    try {

        let details = await Users.findOne({ \_id: mongoose.Types.ObjectId(req.params.userId), isActive: true })

        if (details) {

            let views = details.views,

                today = new Date(),

                dd = String(today.getDate()).padStart(2, '0'),

                mm = String(today.getMonth() + 1).padStart(2, '0'),

                yyyy = today.getFullYear()

            today = mm + '/' + dd + '/' + yyyy;

            let viewCountObj = {},

                newDate = false,

                newCount

            if (views.length === 0) {

                viewCountObj.date = today

                viewCountObj.count = 1

                newDate = true

            } else {

                let found = \_.find(details.views, ['date', today])

                if (found) {

                    newCount = found.count + 1

                } else {

                    viewCountObj.date = today

                    viewCountObj.count = 1

                    newDate = true

                }

            }

            if (newDate) {

                await Users.findByIdAndUpdate(mongoose.Types.ObjectId(req.params.userId), {

                    $push: {

                        views: viewCountObj

                    }

                })

            } else {

                await Users.updateOne({ \_id: mongoose.Types.ObjectId(req.params.userId), 'views.date': today }, {

                    '$set': {

                        'views.$.count': newCount

                    }

                })

            }

            details = await Users.findById(mongoose.Types.ObjectId(req.params.userId))

            details = details.toJSON()

            delete details.password

            return responseFormer(200, details)

        } else {

            return responseFormer(204, null)

        }

    } catch (error) {

        console.log(`Error while fetching user profile details and increasing view count ${error}`)

        return responseFormer(constants.STATUS\_CODE.INTERNAL\_SERVER\_ERROR\_STATUS, error.message)

    }

}

## Code Listing of Server Implementation – Session Objects:

4 simultaneous login sessions allowed: (Login function)

exports.loginUser = async (req, res) => {

    console.log('-----------', 'innnnn', '--------------')

    try {

        var user

        var isAuth = false

        if (isNaN(req.body.loginId)) {

            user = await Users.findOne({

                $or: [

                    {

                        email: req.body.loginId

                    },

                    {

                        userName: req.body.loginId

                    }

                ]

            })

        } else {

            user = await Users.findOne({

                $or: [

                    {

                        phone: req.body.loginId

                    }

                ]

            })

        }

        if (user) {

            const validate = await user.validatePassword(req.body.password)

            if (validate) {

                const token = user.generateToken()

                user = user.toJSON()

                delete user.password

                user.token = token

                let tokenObj = {

                    token : token,

                    date: Date.now()

                }

                if (user.jwtToken.length === 4) {

                    await Users.findByIdAndUpdate(user.\_id, { $pop: { jwtToken: -1 } });

                }

                await Users.findByIdAndUpdate(user.\_id, {

                    $push: {

                        jwtToken: tokenObj

                    }, isActive: true

                });

                await Tweet.updateMany({

                    $or: [

                        {

                            userId: user.\_id

                        },

                        {

                            originalUserId: user.\_id

                        }

                    ]

                },{

                    isActive: true

                })

                await List.updateMany({

                    ownerId: user.\_id

                },{

                    isActive: true

                })

                isAuth = true

                return responseFormer(constants.STATUS\_CODE.SUCCESS\_STATUS, user)

            }

        }

        if (!isAuth) {

            return responseFormer(constants.STATUS\_CODE.UNAUTHORIZED\_ERROR\_STATUS, constants.MESSAGES.AUTHORIZATION\_FAILED)

        }

    } catch (error) {

        console.log(`Error while logging in user ${error}`)

        return responseFormer(constants.STATUS\_CODE.INTERNAL\_SERVER\_ERROR\_STATUS, error.message)

    }

}

userTokenValidator.js:

`use strict`

import Users from '../models/mongoDB/users'

import config from '../../config'

import { getToken } from '../utils/auth'

import { verify } from 'jsonwebtoken'

import constants from '../utils/constants'

export async function ensureUser(req, res, next) {

    const token = getToken(req)

    if (!token) {

        return res.status(constants.STATUS\_CODE.UNAUTHORIZED\_ERROR\_STATUS).send()

    }

    try {

        let decoded = verify(token, config.token)

        let tokenMatch = await Users.findOne({ \_id: decoded.id }, { jwtToken: { $elemMatch: { token: token } } })

        if (!tokenMatch) {

            return res.status(constants.STATUS\_CODE.UNAUTHORIZED\_ERROR\_STATUS).send()

        }

        //setting these fields to use in logout api

        // eslint-disable-next-line require-atomic-updates

        req.tokenToDelete = token

        // eslint-disable-next-line require-atomic-updates

        req.userId = decoded.id

    } catch (err) {

        console.log('in catch', err)

        return res.status(constants.STATUS\_CODE.UNAUTHORIZED\_ERROR\_STATUS).send()

    }

    return next()

}

Passport.js:

`use strict`

import passport from 'passport'

import Users from '../models/mongoDB/users'

import config from '../../config'

import mongoose from 'mongoose'

var JwtStrategy = require('passport-jwt').Strategy,

    ExtractJwt = require('passport-jwt').ExtractJwt;

var opts = {}

opts.jwtFromRequest = ExtractJwt.fromAuthHeaderAsBearerToken();

opts.secretOrKey = config.token;

// console.log("opts", opts.jwtFromRequest)

passport.use(new JwtStrategy(opts, function (jwt\_payload, done) {

    (async () => {

        // console.log("opts.jwtFromRequest", opts.jwtFromRequest)

        // console.log("jwt\_payloaksjdad.id", jwt\_payload.id)

        const user = await Users.findById(mongoose.Types.ObjectId(jwt\_payload.id))

        // console.log(user)

        if (user) {

            return done(null, true)

        } else {

            return done(null, false)

        }

    })()

}));

## Code Listing of Main server file:

Node Backend Main Server File: (/bin/server.js)

let createError = require('http-errors')

let logger = require('morgan')

var express = require('express');

var app = express();

var kafka = require('../kafka/client');

import config from '../config'

import cors from 'cors'

import constants from '../src/utils/constants'

// router for modules

let usersRouter = require('../src/modules/user/router/users')

let tweetsRouter = require('../src/modules/tweet/router/tweets')

let messageRouter = require('../src/modules/messages/router/messages');

let listRouter = require('../src/modules/list/router/list');

let searchRouter = require('../src/modules/search/router/search');

// database connections

require('../src/models/mongoDB/index')

let port = process.env.PORT || 9000

let frontendUrl = process.env.FRONTEND\_URL || "http://localhost:3000"

app.use(logger('dev'))

app.use(express.json())

app.use(express.urlencoded({ extended: false }))

app.use('/public/', express.static('./public/'));

// use cors to allow cross origin resource sharing

app.use(cors({ origin: '\*', credentials: false }));

// base routes for modules

app.use('/users', usersRouter)

app.use('/tweets', tweetsRouter)

app.use('/messages', messageRouter)

app.use('/lists', listRouter)

app.use('/search', searchRouter)

// Ping route to check health of instance for load balancer

app.get('/ping', (req, res) => {

    return res

      .status(constants.STATUS\_CODE.SUCCESS\_STATUS)

      .send()

})

// catch 404 and forward to error handler

app.use(function (req, res, next) {

    next(createError(404))

})

// error handler

app.use(function (err, req, res) {

    // set locals, only providing error in development

    res.locals.message = err.message

    res.locals.error = req.app.get('env') === 'development' ? err : {}

    // render the error page

    res.status(err.status || 500)

    res.render('error')

})

app.listen(config.port, () => console.log(`Twitter server listening on ${port}`))

module.exports = app

Kafka Backend Main Server File: (Kafka-Backend/bin/server.js)

require('babel-core/register')()

require('babel-polyfill')

var connection =  require('../kafka/Connection');

//topics files

//var signin = require('./services/signin.js');

var Books = require('../services/books');

var Users = require('../services/users');

var Tweets = require('../services/tweets');

var Lists = require('../services/lists');

var Message = require('../services/message');

var Search = require('../services/search');

require('../src/models/sqlDB/index')

require('../src/models/mongoDB/index')

function handleTopicRequest(topic\_name,fname){

    //var topic\_name = 'root\_topic';

    var consumer = connection.getConsumer(topic\_name);

    var producer = connection.getProducer();

    console.log('server is running ');

    consumer.on('message', function (message) {

        console.log('message received for ' + topic\_name +" ", fname);

        console.log(JSON.stringify(message.value));

        var data = JSON.parse(message.value);

        fname.handle\_request(data.data, function(err,res){

            console.log('after handle'+res);

            var payloads = [

                { topic: data.replyTo,

                    messages:JSON.stringify({

                        correlationId:data.correlationId,

                        data : res

                    }),

                    partition : 0

                }

            ];

            producer.send(payloads, function(err, data){

                console.log(data);

            });

            return;

        });

    });

}

// Add your TOPICs here

//first argument is topic name

//second argument is a function that will handle this topic request

handleTopicRequest("post\_book",Books)

handleTopicRequest("users",Users)

handleTopicRequest("tweets",Tweets)

handleTopicRequest("lists", Lists)

handleTopicRequest("messages", Message)

handleTopicRequest("search", Search)

## Code Listing of Database Access File:

MongoDB Connection File:

`use strict`

import config from '../../../config'

import mongoose from 'mongoose'

mongoose.connect(config.database.mongoDbUrl, { useNewUrlParser: true, useUnifiedTopology: true, poolSize: 5 })

.then(() => console.log('MongoDB Connected'))

mongoose.Promise = global.Promise

let db = mongoose.connection

db.on('error', console.error.bind(console, 'MongoDB connection error:'))

module.exports = db

SQL Connection File:

`use strict`

let fs = require('fs')

let path = require('path')

let Sequelize = require('sequelize')

import config from '../../../config'

var sequelize = new Sequelize(config.database.name, config.database.user, config.database.password, {

    host: config.database.host,

    port: config.database.port,

    dialect: config.database.dialect,

    pool: {

        max: 5,

        min: 0,

        idle: 10000

    }

})

sequelize

    .authenticate()

    .then(() => {

        console.log('MySQL Connected')

    })

    .catch(err => {

        console.error('Unable to connect to the mysql database:', err)

    })

var db = {}

fs.readdirSync(\_\_dirname)

    .filter(function (file) {

        return (file.indexOf('.') !== 0) && (file !== 'index.js')

    })

    .forEach(function (file) {

        var model = sequelize.import(path.join(\_\_dirname, file))

        db[model.name] = model

    })

Object.keys(db).forEach(function (modelName) {

    if ('associate' in db[modelName]) {

        db[modelName].associate(db)

    }

})

db.sequelize = sequelize

module.exports = db

Redis Connection File:

import redis from 'redis';

let client = redis.createClient();

client.on('connect', function(err){

    if(err){

        console.log("Error occurred while connecting to Redis");

    } else {

        console.log('Connected to Redis Client');

    }

});

module.exports = client;

## Code Listing of Test Class:

Mocha Testing:

var app = require('../bin/server');

var chai = require('chai');

chai.use(require('chai-http'));

var expect = require('chai').expect;

var agent = require('chai').request.agent(app);

var context = {}

describe('Twitter App', function () {

    it('should login', function (done) {

        agent.post('/users/login')

            .set('Accept', 'application/json')

            .send({

                'loginId': 'jayasurya1@gmail.com',

                'password': 'Test@1234',

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                context.token = res.body.token

                context.userId = res.body.\_id

                done();

            });

    });

    it('should get profile', function (done) {

        const { token, userId } = context

        agent.get(`/users/profile/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get tweet', function (done) {

        const { token } = context

        agent.get(`/tweets/fetchTweetByID/5ddddfd5d93f5437285eb551`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get followers of user', function (done) {

        const { token, userId } = context

        agent.get(`/users/followersOfUserId/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get users followed by user', function (done) {

        const { token, userId } = context

        agent.get(`/users/followedByUserId/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all lists not created by requesting user', function (done) {

        const { token, userId } = context

        agent.get(`/lists/all/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all lists owned by requesting user', function (done) {

        const { token, userId } = context

        agent.get(`/lists/owned/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all lists subscribed by requesting user', function (done) {

        const { token, userId } = context

        agent.get(`/lists/subscribed/${userId}`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all tweets and retweets posted by users followed by requesting user', function (done) {

        const { token, userId } = context

        agent.get(`/tweets/fetchTweetByUserID/${userId}/USERFEED?start=0&count=2`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all tweets and retweets posted by the user', function (done) {

        const { token, userId } = context

        agent.get(`/tweets/fetchTweetByUserID/${userId}/MYTWEETS?start=0&count=2`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

                done();

            });

    });

    it('should get all tweets and retweets bookmarked by user', function (done) {

        const { token, userId } = context

        agent.get(`/tweets/fetchTweetByUserID/${userId}/BOOKMARKEDTWEETS?start=0&count=2`)

            .set({

                'Accept': 'application/json',

                'Authorization': `Bearer ${token}`

            })

            .then(function (res) {

                expect(res.status).to.equal(200);

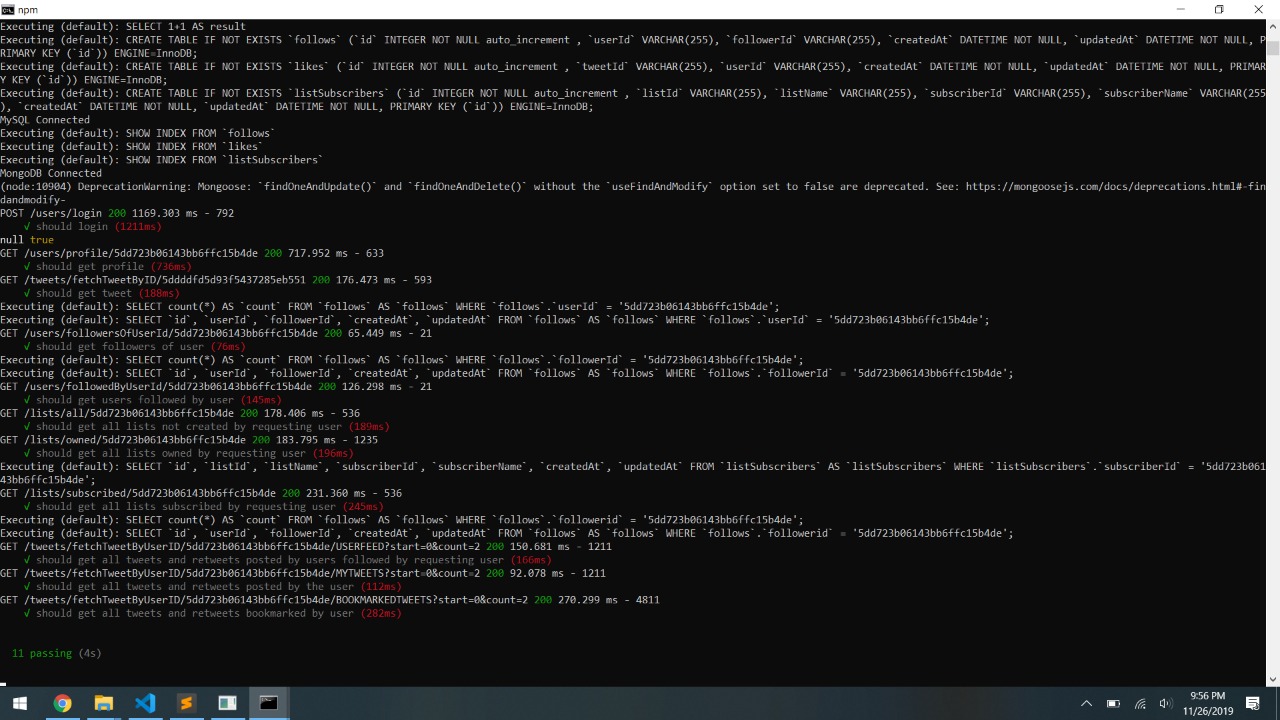
                done();

            });

    });

})

Output Screenshot for Mocha Testing:



Enzyme Testing Class:

Login Test:

import { mount } from 'enzyme'

import 'jest-localstorage-mock'

import React from 'react'

import { shallow } from 'enzyme'

import { configure } from 'enzyme'

import Adapter from 'enzyme-adapter-react-16'

import MyComponent from '../components/root/login'

import { MemoryRouter as Router, withRouter } from 'react-router-dom'

configure({ adapter: new Adapter() })

describe('MyComponent', () => {

  it('Incorrect Login', () => {

    const props = {

      component: () => {}

    }

    const wrapper = shallow(<MyComponent />)

    console.log('testing Invalid Login - Success')

    // const Btn = wrapper.find('button.userLogin')

    // Btn.simulate('click')

    // const text = wrapper.find('p').text()

    // expect(text).toEqual('Fiels cannot be empty')

    expect(

      wrapper

        .find('input')

        .at(2)

        .prop('value')

    ).toEqual('Login')

    expect(wrapper.find('input')).toHaveLength(3)

    expect(

      wrapper

        .find('#userLogin')

        .simulate('click', { preventDefault: () => undefined })

    )

    const text = wrapper

      .find('p')

      .at(0)

      .text()

    expect(text).toEqual(' Fields cannot be empty ')

  })

})

Profile Test:

import { mount } from 'enzyme'

import 'jest-localstorage-mock'

import React from 'react'

import { render } from 'enzyme'

import { shallow } from 'enzyme'

import { configure } from 'enzyme'

import Adapter from 'enzyme-adapter-react-16'

import MyComponent from '../components/user/profile'

import { MemoryRouter as Router, withRouter } from 'react-router-dom'

configure({ adapter: new Adapter() })

describe('MyComponent', () => {

  it('Render Analytical Data', () => {

    const wrapper = shallow(<MyComponent />)

    console.log(wrapper)

    expect(wrapper.find('Navbar').exists()).toBeTruthy()

    // expect(wrapper.find('ProfileViews').exists()).toBeTruthy()

  })

})

Create Tweet Test:

import { mount } from 'enzyme'

import 'jest-localstorage-mock'

import React from 'react'

import { render } from 'enzyme'

import { shallow } from 'enzyme'

import { configure } from 'enzyme'

import Adapter from 'enzyme-adapter-react-16'

import MyComponent from '../components/user/home'

import { MemoryRouter as Router, withRouter } from 'react-router-dom'

configure({ adapter: new Adapter() })

describe('MyComponent', () => {

  it('Create a New Tweet', () => {

    const wrapper = shallow(<MyComponent />)

    console.log(wrapper)

    expect(wrapper.find('#tweetContent').length).toBe(1)

    const tweet = wrapper.find('#tweetContent')

    tweet.value = 'New Tweet'

    expect(tweet.value).toBe('New Tweet')

    wrapper

      .find('#Tweet')

      .simulate('click', { preventDefault: () => undefined })

    const tweet1 = wrapper.find('#tweetContent').text()

    expect(tweet1).toBe('')

  })

})

Create Account Test:

import { mount } from 'enzyme'

import 'jest-localstorage-mock'

import { spy } from 'sinon'

import React from 'react'

import { shallow } from 'enzyme'

import { configure } from 'enzyme'

import Adapter from 'enzyme-adapter-react-16'

import MyComponent from '../components/root/createAccount'

configure({ adapter: new Adapter() })

describe('MyComponent', () => {

  it('Check the Create Account snapshot after sending props', () => {

    console.log('Matching snapshots with create account- Success')

    const props = {

      component: () => {}

    }

    const component = shallow(<MyComponent {...props} debug />)

    expect(component).toMatchSnapshot()

  })

})

Analytics Test:

import { mount } from 'enzyme'

import 'jest-localstorage-mock'

import React from 'react'

import { render } from 'enzyme'

import { shallow } from 'enzyme'

import { configure } from 'enzyme'

import Adapter from 'enzyme-adapter-react-16'

import MyComponent from '../components/analytics/analytics'

import { MemoryRouter as Router, withRouter } from 'react-router-dom'

configure({ adapter: new Adapter() })

describe('MyComponent', () => {

  it('Render Analytical Data', () => {

    const wrapper = shallow(<MyComponent />)

    console.log(wrapper)

    expect(wrapper.find('TopViewed').exists()).toBeTruthy()

    expect(wrapper.find('TopLiked').exists()).toBeTruthy()

    expect(wrapper.find('Navbar').exists()).toBeTruthy()

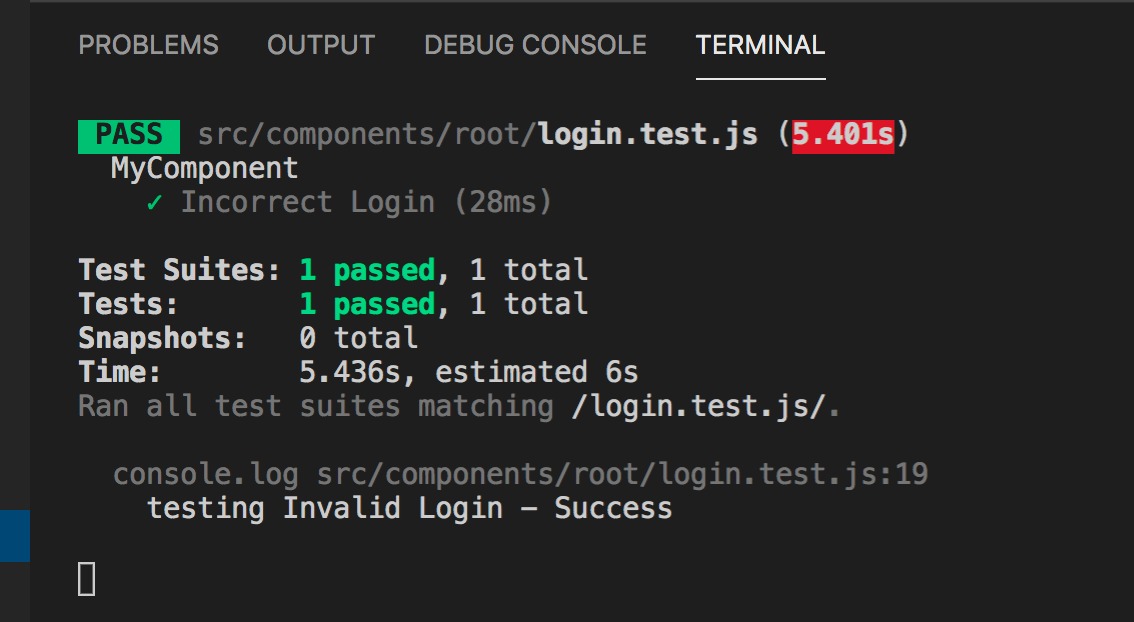
    // expect(wrapper.find('ProfileViews').exists()).toBeTruthy()

  })

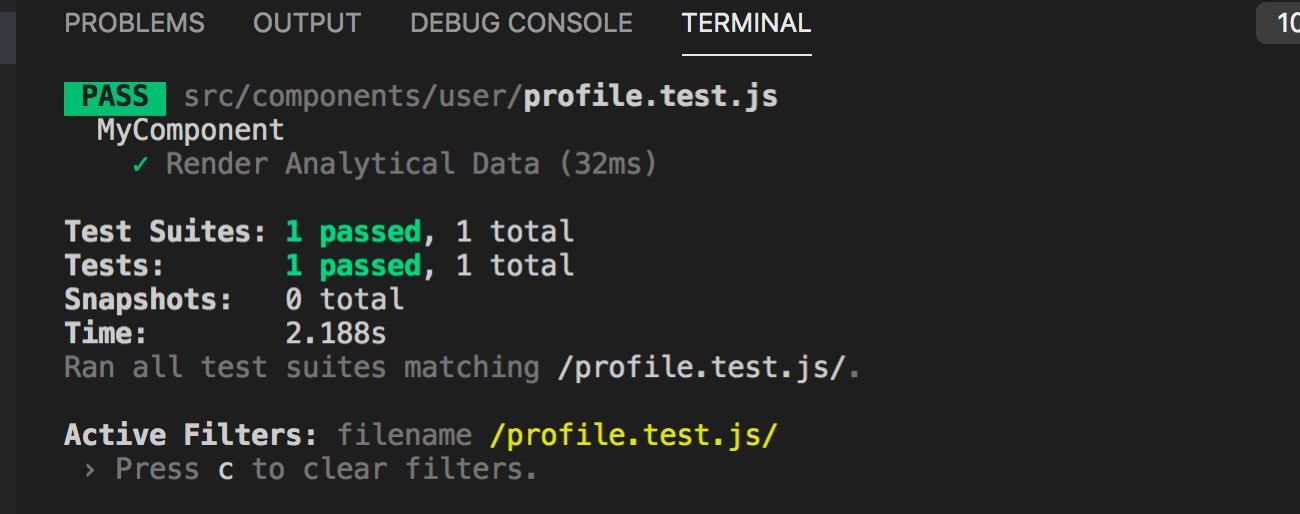
})

Enzyme Testing output screenshots:

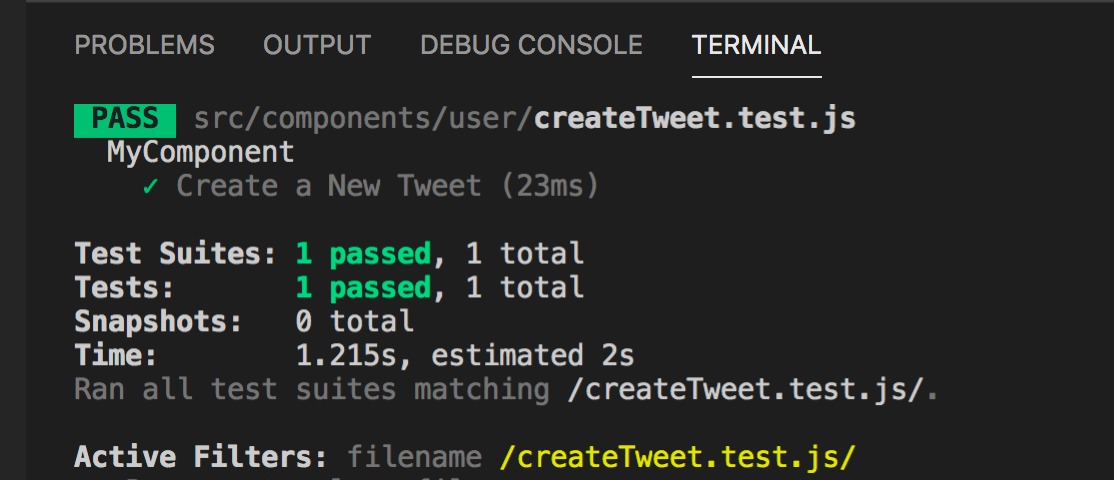
Login:



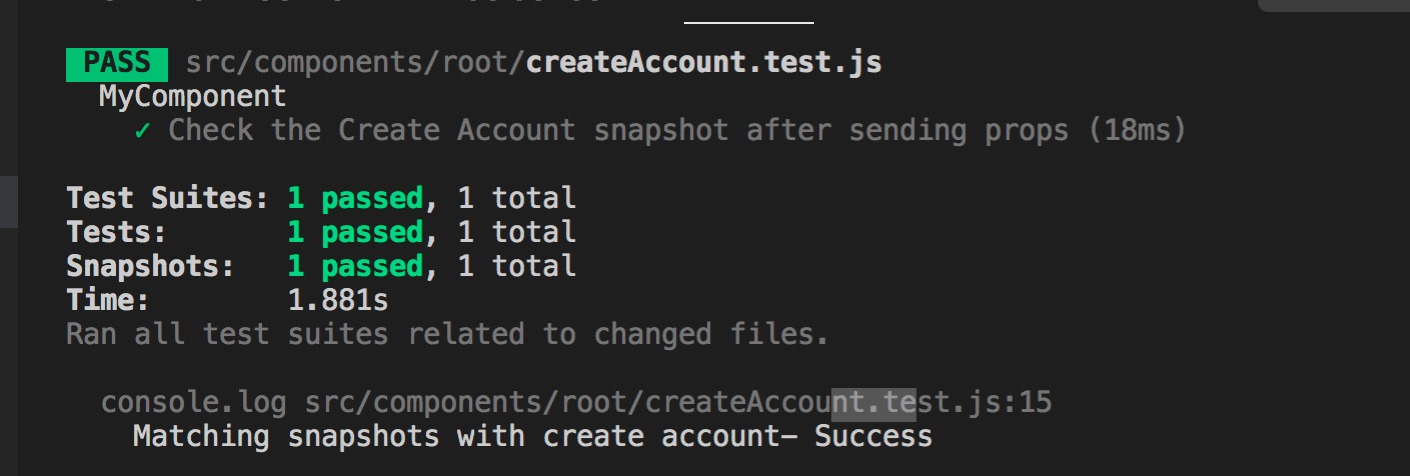
Profile Test:



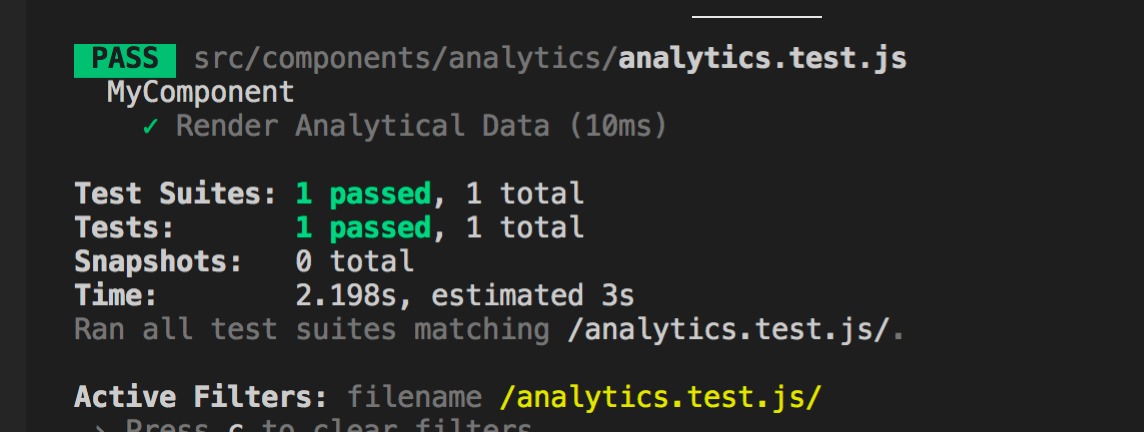
Create Tweet Test:



Create Account Test:



Analytics Test:



## Code Listing of Database Script:

const axios = require('axios');

let usrData = {

        password : "Test@1234",

        dateOfBirth : "January 1 1990"

    },

    tweet = {

        "userImageURL": " ",

    },

    URL = "http://sample-1116658264.us-east-2.elb.amazonaws.com",

    name = "TestUser",

    loginData = {},

    newTweet,

    userLogin,

    token,

    count,

    newUser,

    Date1,

    Date2

(async() => {

    for (count = 1; count <= 10000; count++) {

        Date1 = new Date()

        usrData.name = name + count.toString()

        usrData.email = name + count.toString() + "@gmail.com"

        newUser = await axios.post(URL + "/users/signup", usrData)

        // console.log(newUser.data)

        loginData.loginId = usrData.email

        loginData.password = "Test@1234"

        userLogin = await axios.post(URL + "/users/login", loginData)

        // console.log(userLogin.data)

        token = {

            headers: { 'Authorization': "bearer " + userLogin.data.token }

        }

        newUserName = newUser.data.userName

        tweet.userId = newUser.data.\_id

        tweet.userName = newUser.data.userName

        tweet.originalBody = "Tweet created from script for user " + newUser.data.userName + " with ID " + newUser.data.\_id

        newTweet = await axios.post(URL + "/tweets/createTweet", tweet, token)

        // console.log(newTweet.data)

        Date2 = new Date()

        console.log(Date2 - Date1)

        console.log(count)

    }

}) ()

## Code Listing (Utility Classes):

Image Upload:

'use strict'

import aws from 'aws-sdk';

import multer from 'multer';

import multerS3 from 'multer-s3';

import config from '../../config/env/development';

aws.config.update({

    secretAccessKey: config.awsS3Keys.AWS\_SECRET\_ACCESS,

    accessKeyId: config.awsS3Keys.AWS\_ACCESSKEY,

    region:config.awsS3Keys.REGION,

})

const s3 = new aws.S3();

var upload = multer({

  storage: multerS3({

    s3: s3,

    bucket: 'twitter-273-images',

    acl: 'public-read',

    metadata: function (req, file, cb) {

      cb(null, {fieldName: "Twitter-Images"});

    },

    key: function (req, file, cb) {

      cb(null, Date.now().toString())

    }

  })

})

module.exports = upload;

Kafka Services for 5 topics:

Users:

import UserController from '../src/modules/user/controller/users'

import DeleteController from '../src/modules/user/controller/removeUser'

async function handle\_request (req, callback) {

  console.log('Inside User kafka backend')

  console.log('------------', req.path, '----------------')

  let results

  switch (req.path) {

    case '/signup':

      results = await UserController.createUser(req)

      break

    case '/login':

      results = await UserController.loginUser(req)

      break

    case '/profile/:userId':

      results = await UserController.getUserProfile(req)

      break

    case '/profile':

      results = await UserController.updateUserProfile(req)

      break

    case '/deactivateAccount/:userId':

      results = await UserController.deactivateUserProfile(req)

      break

    case '/bookmarkTweet':

      results = await UserController.bookmarkTweet(req)

      break

    case '/follow':

      results = await UserController.followUser(req)

      break

    case '/unFollow':

      results = await UserController.unFollowUser(req)

      break

    case '/followersOfUserId/:userId':

      results = await UserController.followersOfUserId(req)

      break

    case '/followedByUserId/:userId':

      results = await UserController.followedByUserId(req)

      break

    case '/searchByName':

      results = await UserController.searchByName(req)

      break

    case '/searchByUserName':

      results = await UserController.searchByUserName(req)

      break

    case '/findUser/:userName':

      results = await UserController.findUser(req)

      break

    case '/viewCount/:userId':

      results = await UserController.viewCount(req)

      break

    case '/logout':

      results = await UserController.logout(req)

      break

    case '/deleteUser':

      results = await DeleteController.deleteUser(req)

      break

    case '/validate':

      results = await UserController.validate(req)

      break

  }

  callback(null, results)

  console.log('after callback')

}

exports.handle\_request = handle\_request

Lists:

import ListController from '../src/modules/list/controller/list'

async function handle\_request(req, callback){

    console.log("Inside List kafka backend");

    console.log('------------', req.path, '----------------');

    let results;

    switch(req.path) {

      case '/': results = await ListController.createList(req);

      break;

      case '/owned/:userId': results = await ListController.getOwnedList(req);

      break;

      case '/all/:userId': results = await ListController.getAllList(req);

      break;

      case '/subscribe': results = await ListController.subscribeList(req);

      break;

      case '/subscribed/:userId': results = await ListController.getSubscribedList(req);

      break;

      case '/members/:listId': results = await ListController.getMembersOfList(req);

      break;

      case '/subscribers/:listId': results = await ListController.getSubscribersOfList(req);

      break;

    }

    callback(null, results);

    console.log("after callback");

};

exports.handle\_request = handle\_request;

Tweets:

import TweetContoller from '../src/modules/tweet/controller/tweets'

import FetchTweetContoller from '../src/modules/tweet/controller/fetchTweet'

async function handle\_request(req, callback){

    console.log("Inside User kafka backend");

    console.log('------------', req.path, '----------------');

    let results;

    switch(req.path) {

      case '/createTweet': results = await TweetContoller.createTweet(req);

      break;

      case '/addComment': results = await TweetContoller.addComment(req);

      break;

      case '/:tweetId': results = await TweetContoller.deleteTweet(req);

      break;

      case '/fetchTweetById/:tweetId': results = await TweetContoller.fetchTweetById(req);

      break;

      case '/fetchTweetByUserID/:userId/:taskName': results = await FetchTweetContoller.getTweets(req);

      break;

      case '/topTweetsByLike/:userId': results = await TweetContoller.topTweetsByLike(req);

      break;

      case '/topTweetsByRetweets/:userId': results = await TweetContoller.topTweetsByRetweets(req);

      break;

      case '/topTweetsByViews/:userId': results = await TweetContoller.topTweetsByViews(req);

      break;

      case '/tweetsByMonth/:userId': results = await TweetContoller.tweetsByMonth(req);

      break;

      case '/tweetsByDay/:userId/:month/:year': results = await TweetContoller.tweetsByDay(req);

      break;

      case '/tweetsByHour/:userId/:day/:month/:year': results = await TweetContoller.tweetsByHour(req);

      break;

      case '/likeTweet': results = await TweetContoller.likeTweet(req);

      break;

      case '/fetchTweetForList/:listId': results = await FetchTweetContoller.getTweetsForList(req);

      break;

      case '/searchByHashTag': results = await TweetContoller.searchByHashTag(req);

      break;

    }

    callback(null, results);

    console.log("after callback");

};

exports.handle\_request = handle\_request;

Messages:

import MessageController from '../src/modules/messages/controller/messages'

async function handle\_request(req, callback){

    console.log("Inside User kafka backend");

    console.log('------------', req.path, '----------------');

    let results;

    switch(req.path) {

      case '/send': results = await MessageController.sendMessage(req);

      break;

      case '/newMessage': results = await MessageController.sendNewMessage(req);

      break;

      case '/inbox/:userName': results = await MessageController.getInbox(req);

      break;

      case '/conversation/:userName1/:userName2': results = await MessageController.getConversation(req);

      break;

    }

    callback(null, results);

    console.log("after callback");

};

exports.handle\_request = handle\_request;

Search:

import SearchController from '../src/modules/search/controller/search'

async function handle\_request(req, callback){

    console.log("Inside Search kafka backend");

    console.log('------------', req.path, '----------------');

    let results;

    switch(req.path) {

      case '/tweet/hashtag/:hashtag': results = await SearchController.hashtagSearch(req);

      break;

      case '/fetchProfile/:userId': results = await SearchController.fetchProfile(req);

      break;

    }

    callback(null, results);

    console.log("after callback");

};

exports.handle\_request = handle\_request;

Redis Caching for Profile Data:

client.hmset("profiledata\_" + userObj.userId, userObj, function (err, success) {

                if (err) {

                    console.log(err)

                }

                else {

                    console.log(success);

                    return res.status(200).send(userObj)

                }

            })

profileDetails = await client.hgetall("profiledata\_" + req.params.userId, function (err, success) {

            if (err || !success) {

                console.log(err, !success)

                return null;

            }

            else {

                console.log("Success is ", success);

                console.log("From Redis");

                delete success.password

                return success;

            }

        })

One of input validator file using JOI: (User validator)

`use strict`

import Joi from 'joi'

module.exports = {

    signup: {

        body: {

            name: Joi.string().required(),

            email: Joi.string().email(),

            phone: Joi.number(),

            dateOfBirth: Joi.string().required(),

            password: Joi.string().regex(/^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/).required()

        },

        model: "createUser",

        group: "User",

        description: "Create user and save details in database"

    },

    login: {

        body: {

            loginId: Joi.string().required(),

            password: Joi.string().required()

        },

        model: "loginUser",

        group: "User",

        description: "Login user and send auth token and user details in response"

    },

    getProfile: {

        path: {

            userId: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'getUserDetails',

        group: "User",

        description: "Get user profile details based on userid"

    },

    updateProfile: {

        body: {

            userId: Joi.string(),

            name: Joi.string(),

            userName: Joi.string().max(15),

            // city: Joi.string().optional(),

            // state: Joi.string(),

            // zipcode: Joi.string().regex(/^(?!0{5})(\d{5})(?!-?0{4})(|-\d{4})?$/),

            // description: Joi.string().max(160),

            // password: Joi.string().regex(/^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/),

            // phone: Joi.number(),

            email: Joi.string().email(),

        },

        payload: {

            maxBytes: 209715200,

            output: 'file',

            parse: true

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'updateUserDetails',

        group: "User",

        description: "Update user deatils based on userid"

    },

    deactivateProfile: {

        path: {

            userId: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'deactivateUserDetails',

        group: "User",

        description: "Deactivate user based on userid"

    },

    bookmarkTweet: {

        path: {

            userId: Joi.string().required(),

            tweetId: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'bookmarkTweet',

        group: "User",

        description: "Bookmark a tweet"

    },

    followUser : {

        body: {

            userId: Joi.string().required(),

            followerId: Joi.string().required().not(Joi.ref('userId')),

        },

        header: {

            authorization: Joi.string().required()

        },

        model : 'followUser',

        group: 'User',

        description: 'follow a user'

    },

    unFollowUser : {

        body: {

            userId: Joi.string().required(),

            followerId: Joi.string().required().not(Joi.ref('userId'))

        },

        header: {

            authorization: Joi.string().required()

        },

        model : 'unFollowUser',

        group: 'User',

        description: 'un-follow a user'

    },

    followersOfUserId : {

        params: {

            userId: Joi.string().required()

        },

        model: 'followersUserId',

        group: 'User',

        description: 'get followers based on userid'

    },

    followedByUserId: {

        params: {

            userId: Joi.string().required()

        },

        model: 'followerfollowedByUserIdsUserId',

        group: 'User',

        description: 'get the users followed by userid'

    },

    searchByName: {

        body: {

            keyword: Joi.string().required()

        },

        model: 'SearchByName',

        group: 'User',

        description: 'search by profile name'

    },

    searchByUserName: {

        body: {

            keyword: Joi.string().required().regex(/^@[a-zA-Z]+/)

        },

        model: 'SearchByUserName',

        group: 'User',

        description: 'search by user name'

    },

    findUser: {

        path: {

            userName: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'getUserDetails',

        group: "User",

        description: "Get user profile details based on userid"

    },

    viewCount: {

        path: {

            userId: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'getUserDetails',

        group: "User",

        description: "Get user profile details based on userid"

    },

    logout: {

        header: {

            authorization: Joi.string().required()

        },

        model: 'logout',

        group: "User",

        description: "Logout user and delete the token from database"

    },

    deleteUser: {

        path: {

            userId: Joi.string().required(),

            userName: Joi.string().required()

        },

        header: {

            authorization: Joi.string().required()

        },

        model: 'deleteUserDetails',

        group: "User",

        description: "Delete all user details"

    },

}

Routes Modularization:

User Routes:

`use strict`

import express from 'express'

let router = express.Router()

import userController from '../controller/users'

import userRemover from '../controller/removeUser'

import validator from '../validator'

import validation from 'express-validation'

require('../../../middlewares/passport')

import passport from 'passport'

import upload from '../../../middlewares/imageUpload'

import {ensureUser} from '../../../middlewares/userTokenValidator'

router.post('/signup', validation(validator['signup']), userController.createUser)

router.post('/login', validation(validator['login']), userController.loginUser)

router.get('/profile/:userId', validation(validator['getProfile']), passport.authenticate('jwt', { session: false }), ensureUser, userController.getUserProfile)

router.put('/profile/', upload.single('image'), validation(validator['updateProfile']) , passport.authenticate('jwt', { session: false }), ensureUser, userController.updateUserProfile)

router.delete('/deactivateAccount/:userId', validation(validator['deactivateProfile']) , passport.authenticate('jwt', { session: false }), ensureUser, userController.deactivateUserProfile)

router.post('/bookmarkTweet', validation(validator['bookmarkTweet']) , passport.authenticate('jwt', { session: false }), ensureUser, userController.bookmarkTweet)

router.post('/follow', validation(validator['followUser']), passport.authenticate('jwt', { session: false }), ensureUser, userController.followUser)

router.post('/unFollow', validation(validator['unFollowUser']), passport.authenticate('jwt', { session: false }), ensureUser, userController.unFollowUser)

router.get('/followersOfUserId/:userId', validation(validator['followersOfUserId']), passport.authenticate('jwt', { session: false }), ensureUser, userController.followersOfUserId)

router.get('/followedByUserId/:userId', validation(validator['followedByUserId']), passport.authenticate('jwt', { session: false }), ensureUser, userController.followedByUserId)

router.post('/searchByName',validation(validator['searchByName']), passport.authenticate('jwt', { session: false }), ensureUser, userController.searchByName)

router.post('/searchByUserName',validation(validator['searchByUserName']), passport.authenticate('jwt', { session: false }), ensureUser, userController.searchByUserName)

router.get('/findUser/:userName', validation(validator['getProfile']), passport.authenticate('jwt', { session: false }), ensureUser, userController.findUser)

router.get('/viewCount/:userId', validation(validator['viewCount']), passport.authenticate('jwt', { session: false }), ensureUser, userController.viewCount)

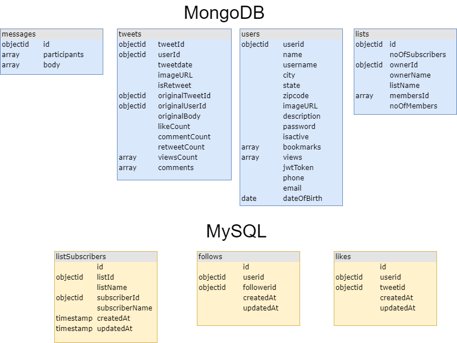
router.put('/logout',  validation(validator['logout']), passport.authenticate("jwt", { session: false }), passport.authenticate("jwt", { session: false }), ensureUser, userController.logout)

router.post('/deleteUser',/\* validation(validator['deleteUser']), passport.authenticate('jwt', { session: false }), ensureUser, \*/ userRemover.deleteUser)

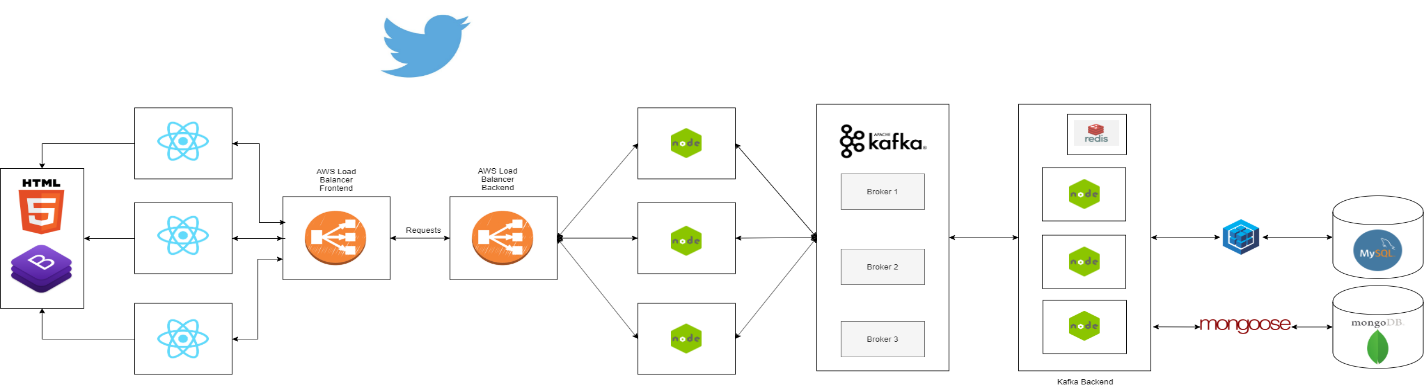
router.get('/validate', passport.authenticate('jwt', { session: false }), ensureUser, userController.validate)

module.exports = router

## Database Schema:



## Architecture Diagram:



## Observations and Learning:

**Distributed Architecture:**

Developing a distributed architecture is important when it comes to building large applications that can be used by several users across the geography. This means, the system must be such that it should be able to manage huge load and traffic at any given time. Moreover, the system should be fault tolerant and partitioned such that failure of single component at a single location doesn’t crash other components anywhere else. In order to build such a system, we need to implement Partition Tolerance and Availability among the three parts of CAP theorem. Building a robust and scalable architecture requires the system to be horizontally scalable i.e. we should be able to add any type of component to the system as opposed to increasing the number of components already present. Horizontal scaling improves the efficiency and performance of the system like no other. Furthermore, it is highly cost efficient as compared to vertical scaling.

**Modular Design:**

Throughout the development of the application, it was evident how important it is to write a modular code which is loosely coupled with each other. This allows easy modifications and smooth debugging. Not only that, modular design enhances code reusability. We realized soon how difficult it is to go back and change a piece of code as requirements changed. Hence, we put lot of efforts in designing the code structure in such a way that collaboration can be achieved easily, and code modification is a hassle-free process. Differentiating the purpose of tasks into multiple components helps to design the structure where each component serves only a single purpose and no other purpose is combined. Hence, if there is a change in single module, only that module will be changed keeping rest of the modules intact. This is better way to identify and fix bugs.

**Collaboration and Integration:**

One of the challenges of building a distributed application is integrating all pieces of puzzle to perfectly interact and communicate with each other to produce a single application. End users must be unaware of the distributed architecture and whole application should act like single model. Therefore, to integrate various components, we need to make sure the code style, input and output format, request handling, exception handling, validation, middleware must be uniform across all members. To achieve this, it is important to spend sufficient planning and defining the standards. It indeed is difficult to merge code written by various people into a single application however, overcoming this challenge is key towards building bug free code.

Overall, all the practices and learning fuse together to help build a robust application. If any one of the above is not achieved properly, it may have ripple effect on other processes and management. Continuous integration and reviews drive building software from scratch.

## Analysis Dashboard Screenshots:

