SPE Major Project

Harsh Shah — IMT2020006
Prem Shah — IMT2020044

Instructor: Prof. B. Thangaraju *Date:* 11 December 2023

SAC ELECTION PORTAL

1 REPOSITORIES

- Github link: Repo
- Dockerhub link:
  - Backend Image Repo
  - Frontend Image Repo

2 ABSTRACT

Voting is a very important part of our lives. Voting enables us to make basic decisions that affect our lives. Voting is an integral part of a democratic society. In our college particularly, voting is involved for selecting members for SAC. Students need to physically go to designated voting locations in order to cast their vote. This can lead to lesser accessibility and in turn lesser vote percentage. On the other hand, online portals allow voters to participate from anywhere with an internet connection, eliminating the need to travel to physical polling stations. This, in particular, is helpful for those who are unable to visit polling stations in person.

3 INTRODUCTION

The SAC Elections Portal is a custom-built website that offers comprehensive information about ongoing elections, including details about participating candidates, eligible student voters, and the status of votes cast. This platform serves as a centralized hub for hosting SAC elections, providing students with the ability to conveniently cast their votes.

3.1 Functionalities

- The homepage of the SAC Elections Portal showcases all presently ongoing elections, presenting diverse statistics like candidate count, voter participation, and registered vote counts.

Live standings for each election are also available, providing up-to-the-minute updates. Additionally, a convenient button is included to facilitate the process of adding a new vote.

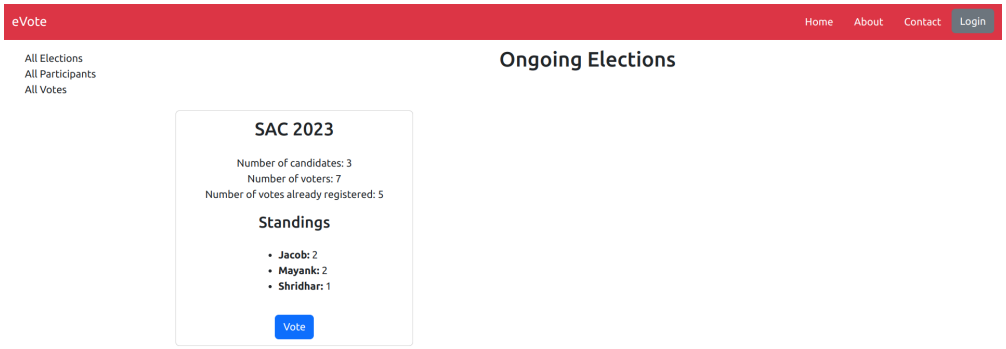


Figure 1: Home Page

- Each election comprises a dynamic list featuring eligible student voters and the candidates vying for positions within that specific election. This list is continually updated, allowing for the addition or removal of new participants as necessary.
- To cast a vote in an election, users are required to input their roll number and select their preferred candidate from a list provided. Upon submission, the backend system verifies the user's roll number to ensure its validity before registering their vote. This verification process ensures that only eligible users can successfully cast their votes.

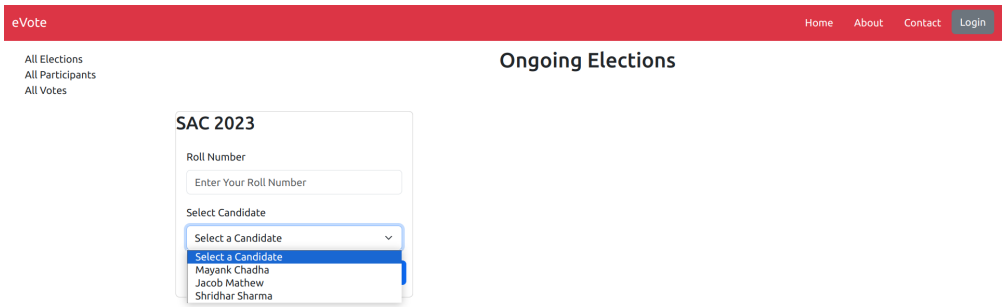


Figure 2: Vote cast

- The "All Participants" page exhibits a comprehensive list encompassing both students and candidates across all ongoing elections. This centralized platform enables users to add new students or candidates and delete existing entries, providing efficient management capabilities for participants across multiple elections.

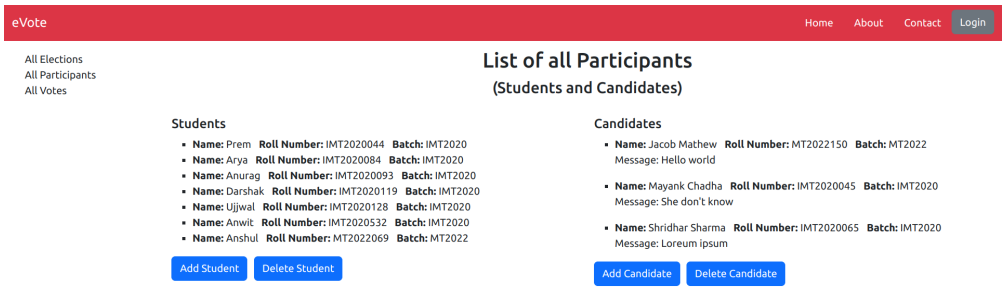


Figure 3: Participants involved in voting

- A dedicated page exists to display the comprehensive list of all votes registered to date. Users are granted the option to edit their votes should they reconsider their choice of candidate, allowing for the selection of an alternate candidate in case of a change of preference.

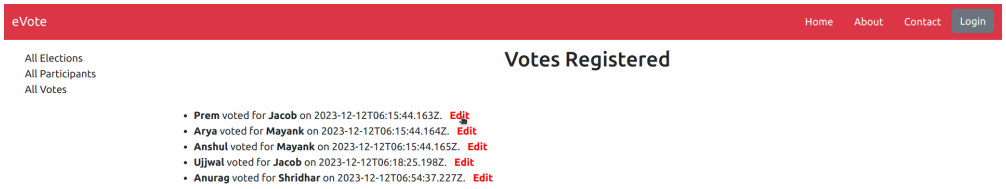


Figure 4: List of all votes

3.2 Tech Stack

- Front end for the web application was built with ReactJS, framework for Javscript.
- Back end was built using Node.js and Express.js framework.
- Database used was MongoDB, a NoSQL database.

3.3 Tools used

- Version Control: Git
- CI/CD Pipeline: Jenkins
- Testing: Mocha and Chai
- Build/Packaging: Node Package Manager(npm)
- Containerization: Docker
- Deployment: Ansible
- Continuous Monitoring: ELK Stack

4 WHAT IS DEVOPS

DevOps represents a transformative approach to software development and IT operations, driven by collaboration, automation, and cultural alignment. It's a philosophy that fosters a harmonious relationship between traditionally separate development and operations teams, promoting shared goals and responsibilities throughout the software delivery life cycle. By breaking down silos and encouraging open communication, DevOps aims to streamline processes, increase agility, and accelerate the delivery of high-quality software. This approach prioritizes automation, employing tools and practices to automate repetitive tasks.

5 SOFTWARE DEVELOPMENT LIFE CYCLE

5.1 Installations

5.1.1 NodeJS

Choosing NodeJS for the backend of the project was driven by the aim to maintain language consistency across both the frontend and backend components. NodeJS, with its JavaScript-based runtime, enables developers to use the same language (JavaScript) for both client-side and server-side scripting.

To install NodeJS on your system:

1. Open the terminal and enter the command `sudo apt install nodejs`
2. Verify the installation using `node --version`
3. For this project, we need to install a node package manager. Run the command `sudo apt install npm`

5.1.2 MongoDB

To install MongoDB on your system:

1. Run the command `sudo apt install -y mongodb`
2. To start MongoDB service, run `sudo systemctl start mongodb`
3. To stop MongoDB service, run `sudo systemctl stop mongodb`

5.2 Version Control System

Version control is a system that tracks changes to files over time, enabling multiple contributors to collaborate on a project by managing different versions of files, facilitating collaboration, and providing the ability to revert to previous states or branches. For our project, we have used Git as our version control system.

5.2.1 CI Pipeline

Start Jenkins from terminal and open the url localhost:8080 in your browser. Create a new project. To make jenkins clone your git repository, add the following script.

```
pipeline {
    environment {
        docker_image=""
    }
    agent any
    stages {
        stage('Stage 1: Git Clone') {
            steps {
                git branch: 'main', url: 'https://github.com/harsh788/SAC_Election_Portal.git'
            }
        }
    }
}
```

5.3 Docker

Docker is a platform that simplifies the deployment and management of applications by using containers—lightweight, portable, and self-sufficient environments that package applications and their dependencies, allowing them to run consistently across different environments. It abstracts away the underlying infrastructure complexities, enabling easier development, testing, and deployment of software.

We will use Docker for building docker images by making use of the Dockerfile. Dockerfile will contain steps to pull a base image from DockerHub, installing the dependencies mentioned in package.json file by running the command `npm install`, copying the source code, building and packaging the react app source code using the command `npm run build`, etc.

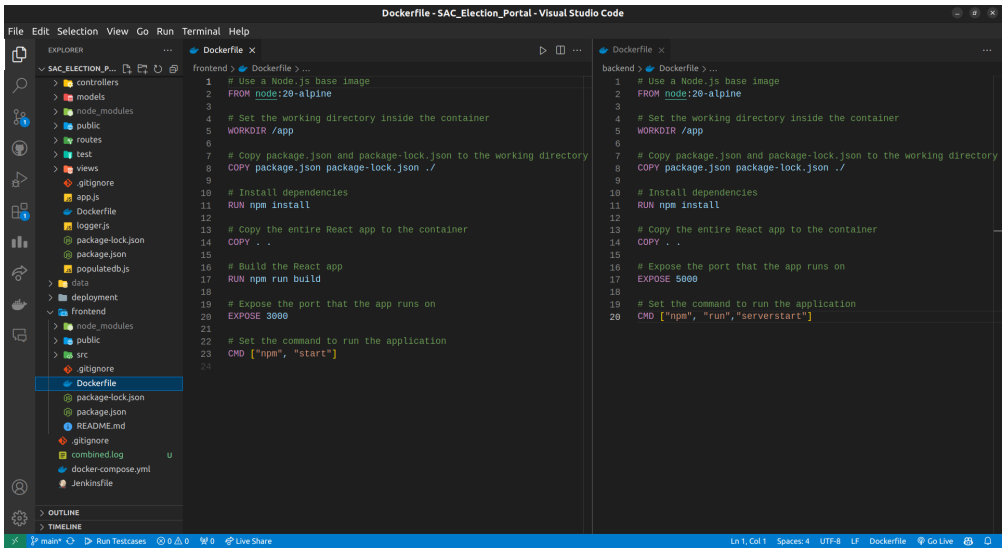


Figure 5: Dockerfile

5.3.1 CI Pipeline

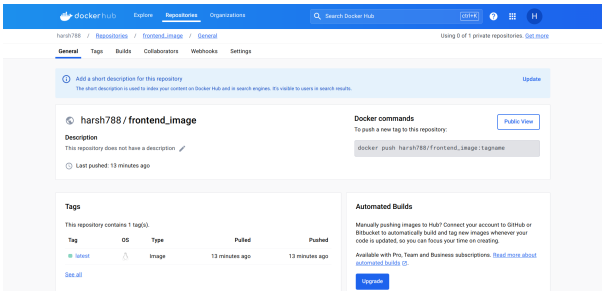
To automate the process of creating Docker images of backend and frontend folders with jenkins, we will add the following script in the pipeline:

```
stage('Stage 2: Build Docker Image'){
    steps{
        dir('backend')
        {
            script{
                docker_image = docker.build "harsh788/backend_image:
   latest"
            }
        }
        dir('frontend')
        {
            script{
                docker_image = docker.build "harsh788/frontend_image:
   latest"
            }
        }
    }
}
```

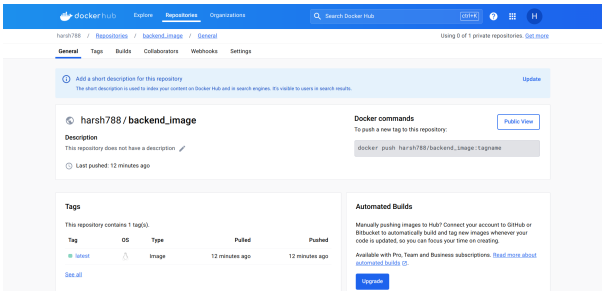
To create a mongodb container for our database, we first need to build an image for it. Pull the official base version of mongo image from DockerHub. This step needs to be performed before creating frontend and backend images because the containers of these iamges will depend on mongodb container. Add the following code at the top of the Jenkins pipeline:

```
stage('Stage 0: Pull mongo docker image')
{
    steps{
        script{
            sh 'docker rm -f mongo_container frontend_container backend_container'
            docker.withRegistry('', 'DockerHubCred') {
                docker.image('mongo').pull()
            }
        }
    }
}
```

We will also push these newly created docker images to DockerHub so that anyone can pull them and use it in their own systems. To do so, add the following stage in the pipeline:



(a) Front end image repository



(b) Back end image repository

Figure 6: DockerHub

```
stage('Stage 3: Push docker image to hub') {
    steps{
        script{
            docker.withRegistry('', 'DockerHubCred'){
                // docker_image.push()
                sh 'docker push harsh788/frontend_image:latest'
                sh 'docker push harsh788/backend_image:latest'
            }
        }
    }
}
```

Once you run this pipeline multiple times, you will notice duplicate instances of docker images getting created every time. These are nothing but older versions of the docker images which are no longer useful. To remove these from our system, we can create another pipleine stage as follows:

```
stage('Stage 4: Clean docker images'){
    steps{
        script{
            sh 'docker container prune -f'
            sh 'docker image prune -f'
        }
    }
}
```

5.4 Docker-Compose

As one might have noticed till now, we are working with multiple docker containers, one for the database and one for each of frontend and backend. To manage all of their activities, like starting or stoping multiple containers at once, we make of the docker-compose file. It is a yaml file which contains instructions to build docker containers from images on specified ports, and to allow them to communicate with each other over a shared network. Communication between these containers is a must, as frontend container will send api requests to the backend container, which in turn will query the database container before sending a response. To achieve data persistance in our mongodb container, we make use of volumes. Volumes are a feature that allows containers to persist and manage data beyond the container's life cycle.

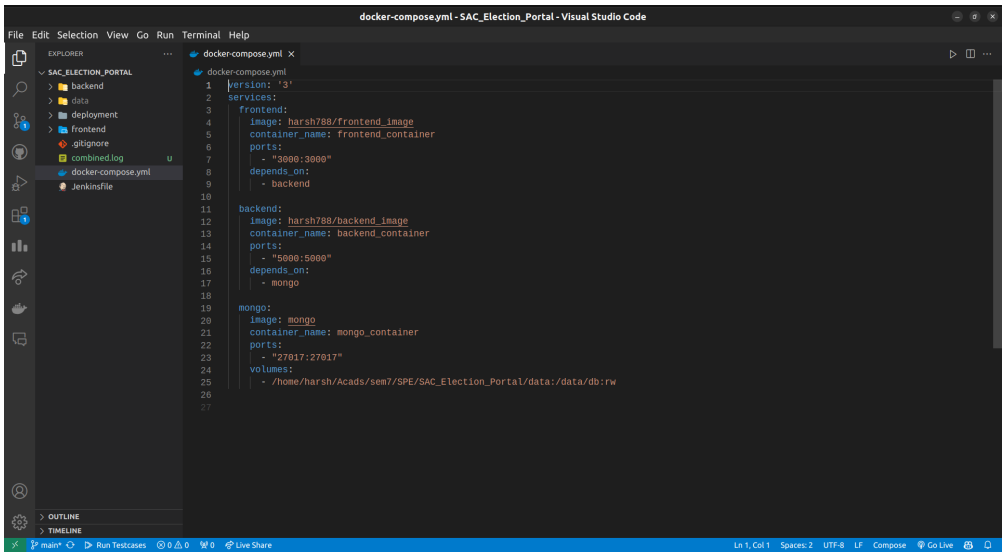


Figure 7: docker-compose file

5.5 Deployment-Ansible

Ansible is an open-source automation tool used for configuration management, application deployment, and orchestration. It simplifies complex tasks by allowing the automation of repetitive system administration tasks across multiple servers, streamlining infrastructure management

through a declarative language and agentless architecture, making it popular for IT automation and DevOps practices.

Ansible works against multiple managed nodes or "hosts" in your infrastructure at the same time, using a list or group of lists known as inventory. Once your inventory is defined, you use patterns to select the hosts or groups you want Ansible to run against.

Playbooks contain the steps which the user wants to execute on a particular machine. Playbooks are run sequentially. Playbooks are the building blocks for all the use cases of Ansible.

deployment > deploy.yml

```yaml
---
- name: Deploy docker images
  hosts: localhost

  tasks:
    - name: Copy Docker Compose file from host machine to remote host
      copy:
        src: ../docker-compose.yml
        dest: ./

    - name: Pull the Docker images specifed in docker-compose
      shell:
        cmd: docker-compose pull
        chdir: ./

    - name: Run the pulled Docker images in detached mode
      command: docker-compose up -d --build
```

(a) Deploy.yml

deployment > inventory

```
[localhost]
127.0.0.1 ansible_connection=local ansible_user=harsh
```

(b) Inventory

Figure 8: Deployment files

The complete workflow for Ansible goes like this:

- Initialization: Ansible reads the inventory file to identify the target hosts for deployment.
- Execution: The deploy.yml files contains the following tasks: Copying the docker-compose file to remote host, pulling the images mentioned in it, running the docker-compose file which will start the three containers.

5.5.1 CI Pipeline

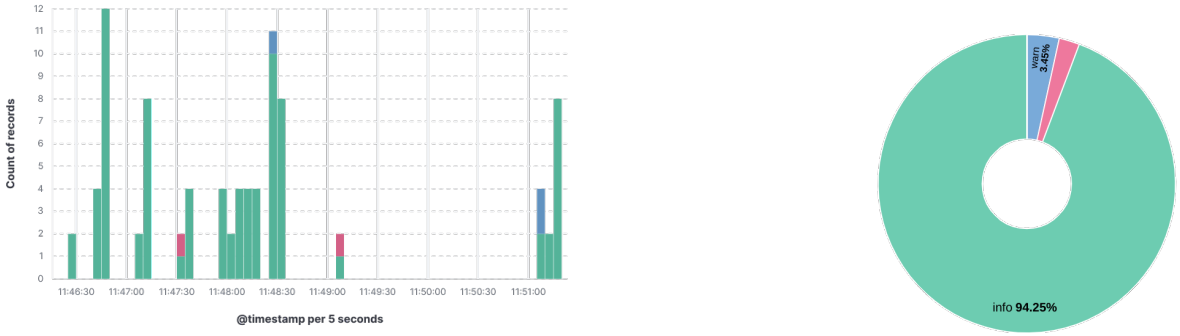
To add the above mentioned tasks in jenkins for automation, we update the jenkins pipeline by adding the following stage:

```
stage('Step 5: Ansible Deployment'){
    steps{
        ansiblePlaybook becomeUser: null,
        colorized: true,
        credentialsId: 'localhost',
        disableHostKeyChecking: true,
        installation: 'Ansible',
        inventory: 'deployment/inventory',
        playbook: 'deployment/deploy.yml',
        sudoUser: null
    }
}
```

5.6 Continuous Monitoring

Continuous monitoring is a practice of consistently observing and analyzing systems, applications, and networks to detect and respond to potential issues or threats in real-time. It involves gathering metrics, logs, and other relevant data to gain insights into the health, performance, and security of an environment, ensuring continuous availability and reliability. Various log messages have been included in the project. On execution, these logs will be added to a file in the container. We need to extract this file. We can do so by running the command: `sudo docker cp backend_container:/app/combined.log /your_path`. Once we have the log file, we can upload it to Kibana. To do so, we need to start the Elasticsearch and Kibana service. Run the following commands:

- `sudo systemctl start elasticsearch`
- `sudo systemctl start kibana`



(a) Stacked bar chart (b) Pie chart

Figure 9: Log file visualizations using Kibana

5.7 Testing

5.7.1 Backend Testing

The goal is to ensure that each unit of code behaves as expected and to catch any potential bugs or issues early in the development process. Unit testing in backend JavaScript involves testing individual units of code, typically functions or methods, in isolation from the rest of the application.

Framework Used

1. **Mocha**: Mocha is a flexible and feature-rich test framework that provides a structure for organizing and running tests.
2. **Chai**: Chai is an assertion library that works seamlessly with Mocha (and other test frameworks). It provides a set of expressive and chainable APIs for making assertions about the behavior of code.

Sample Test Case

```
// test vote_list function
describe('/GET vote_list', () => {
    it('it should display list of all votes', (done) => {
        chai.request('http://localhost:5000')
        .get('/dashboard/votes')
        .end((err, res) => {
            if(err){
                console.error(err);
                logger.error('Error during test: ' + toString(err));
                logger.error('Test failed');
                return done(err);
            }

            if(typeof res.body === 'object' && res.body !== null){
                expect(res).to.have.status(200);
                expect(res.body).to.have.property('title').eql('List of all the votes');
                expect(res.body).to.have.property('vote_list');
                expect(res.body).to.have.property('candidate_list');
                expect(res.body).to.have.property('student_list');
                done();

                logger.info('Test passed');
            }
            else{
                done(new Error('Unexpected response body format'));
                logger.error('Unexpected response body format');
                logger.error('Test failed');
            }
        });
    });
```

Figure 10: Test case to test '/dashboard/votes' api

Test Results

- Execute the test using 'mocha' command in the terminal: `mocha Test.js`
- OR Go to the 'backend' directory and run the command `npm test`.

```
(base) prem@LAPTOP-NORT8EO5:/mnt/c/acads/Sem7/Nodejs/SAC_Election_Portal/backend$ npm test

> final-project@0.0.0 test
> mocha ./test/Test.js



  Vote Controller
    /GET vote_list
      ✓ it should display list of all votes
    /GET vote_update_get/:id
      ✓ it should GET updated vote details

  Student Controller
    /GET studennt_list
      ✓ it should GET the list of all students
    /GET student_detail/:id
      ✓ it should GET the detail page for a student with valid ID
      ✓ it should return a 500 status for an invalid ID

  Candidate Controller
    /GET candidate_list
      ✓ it should GET a list of all candidates


  6 passing (108ms)
```

Figure 11: Test results

6 API DOCUMENTATION

6.1 Schemas

There are four schemas used in this project. They are as follows:

- **Student**: It contains three fields: name, roll_number, batch
- **Candidate**: It contains five fields: first_name, last_name, roll_number, batch, message
- **Vote**: It contains three fields: voter, timestamp, selection
- **Election**: It contains four fields: title, voter_list, votes, candidates

```
const Schema = mongoose.Schema;

const studentSchema = new Schema({
    name: {type: String, required: true, maxLength: 100},
    roll_number: {type: String, required: true, minLength: 9, maxLength: 10},
    batch: {
        type: String,
        required: true,
        enum: ["IMT2020", "IMT2019", "MT2022"],
        default: "IMT2020",
    },
});
```

(a) Student Schema

```
const candidateSchema = new Schema({
    first_name: {type: String, required: true, maxLength: 100},
    last_name: {type: String, required: true, maxLength: 100},
    roll_number: {type: String, required: true, minLength: 9, maxLength: 10},
    batch: {
        type: String,
        required: true,
        enum: ["IMT2020", "IMT2019", "MT2022"],
        default: "IMT2020",
    },
    message: {type: String, required: true},
});
```

(b) Candidate Schema

```
const Schema = mongoose.Schema;

const voteSchema = new Schema({
    voter: {type: Schema.Types.ObjectId, ref: "Student", required: true},
    timestamp: {type: Date, required: true},
    selection: {type: Schema.Types.ObjectId, ref: "Candidate", required: true},
});

// Virtual for vote url
voteSchema.virtual("url").get(function () {
    return `/dashboard/vote/${this._id}`;
});
```

(c) Vote Schema

```
const Schema = mongoose.Schema;

const electionSchema = new Schema({
    title: {type: String, required: true},
    voter_list: [{type: Schema.Types.ObjectId, ref: "Student"}],
    votes: [{type: Schema.Types.ObjectId, ref: "Vote"}],
    candidates: [{type: Schema.Types.ObjectId, ref: "Candidate"}],
});

electionSchema.virtual("url").get(function() {
    return `/dashboard/election/${this._id}`;
});
```

(d) Election Schema

Figure 12: Various Schemas

6.2 API Endpoints

The various API endpoints exposed by the server, and where the frontend can make a fetch request are as follows:

- **HomePage:** *http://localhost:5000/* or *http://lcoalhost:5000/dashboard* [GET]
- **Casting a vote to an election:** *http://localhost:5000/dashboard/election/:id* [GET POST]
- **List of all Students:** *http://localhost:5000/dashboard/students* [GET]
- **Adding a new Student:** *http://localhost:5000/dashboard/student/create* [GET POST]
- **Deleting an existing Student:** *http://localhost:5000/dashboard/student/:id/delete* [GET POST]
- **List of all Candidates:** *http://localhost:5000/dashboard/candidates* [GET]
- **Adding a new Candidate:** *http://localhost:5000/dashboard/candidate/create* [GET POST]
- **Deleting an existing Candidate:** *http://localhost:5000/dashboard/candidate/:id/delete* [GET POST]
- **List of all Votes:** *http://localhost:5000/dashboard/votes* [GET]
- **Updating an existing Vote:** *http://localhost:5000/dashboard/vote/:id/update* [GET POST]

7 FUTURE WORK

- **User authorization**: Currently there are no roles assigned to a user. Any user can perform any operation. A role based mechanism can be implemented with exclusive rights to admin.
- **User authentication:** When separate roles are assigned to users, there needs to be an authentication mechanism so as to validate users with the help of username and password.
- **Add more CRUD operations:** Election Create, Update, Delete; Student Update; Candidate Update; Vote Delete operations are yet to be implemented.

8 REFERENCES

1. **Backend Testing**: https://medium.com/spidernitt/testing-with-mocha-and-chai-b8da8d2e10f2
2. **MongoDB**: https://www.mongodb.com/docs/drivers/node/current/
3. **Express.js**: https://expressjs.com/en/guide/routing.html