International Institute of Information Technology, Bangalore

Software Project Engineering Mini Project

Telemedicine

Under the Guidance of
Prof. B. Thangaraju and
Teaching Assistant: Atibhi Agrawal

Group 10

Jaymeen Kachrola (MT2020035)
Parth Patel (MT2020057)

Tables of Content

1. Introduction

1.1 OverView

Our project idea was to design a system where doctors can meet/consult patients remotely over a video/audio call. As we know there are many already available solutions where doctors and patient can consult over a video call (Hangouts, Skype, Zoom, Google Meet etc.), what our design provides which is unique/different from these already available solutions ?

1.2 Features

Peer to Peer Connection

- For WebRTC connection, what we will need is a signaling server. The job of this signaling server would be to help the peers connect before establishing a peer to peer connection.
- Once the peer to peer connection is established, the signaling server will become obsolete and it won't be required. Even if you disconnect the signaling server while the video call is still going on, you won't face any kind of disconnections.
- Once the connection is established, the entire thing becomes peer to peer so we can have a secure video chat without worrying about the security issues.

Multi-Stream Video Conferences

- In this app, any user can add multiple streams over the same connection. Why would we require such a functionality ?
- With this functionality in the patient side there can be devices connected to his/her PC and doctor can see output of it while having conversation with the patient simultaneously.

No requirement of sharing ID's

- One another feature of this app is there no requirement to share ID's with which the doctors and patient gets connected.
- When the patient calls any doctor, the notification reaches to the doctor in real time (if he is logged in, in case the doctor is not logged in, the call will be stored in backend as pending and the notification will reach doctor when he logs in) and he/she has to just click a button to accept/reject calls.

2. System Configuration

2.1 Operating system

- Ubuntu 20.04 Focal Fossa

2.2 CPU and RAM

- 2 core processor and RAM 8 GB

2.3 Language

- JavaScript, TypeScript, Angular web framework, Mocha Testing Library and Winston Logging Library

2.4 Kernel Version

- Linux Machine 5.4.0-28-generic

2.5 Database

- MySQL 8.0.20 (MySQL Community Server - GPL)

2.6 Building Tools

- Node for build frontend

2.7 DevOps Tools

- Source Control Management - GitHub
- Continuous Integration - Jenkins
- Containerization – Docker Compose
- Continuous deployment - Ansible
- Monitoring - ELK Stack (Elastic Search, Logstash, Kibana)

3. Software Development life cycle

3.1 Source Code

We have following project structure. There are 4 folder namely ansible_deployment, Backend, Ng-frontend, WebRTC_Signaling_Server in entire project. Backend, Ng-frontend, and WebRTC_Signaling_Server contain main application code and ansible_deployment contain deployment related code. Let's try to understand each folder step by step.

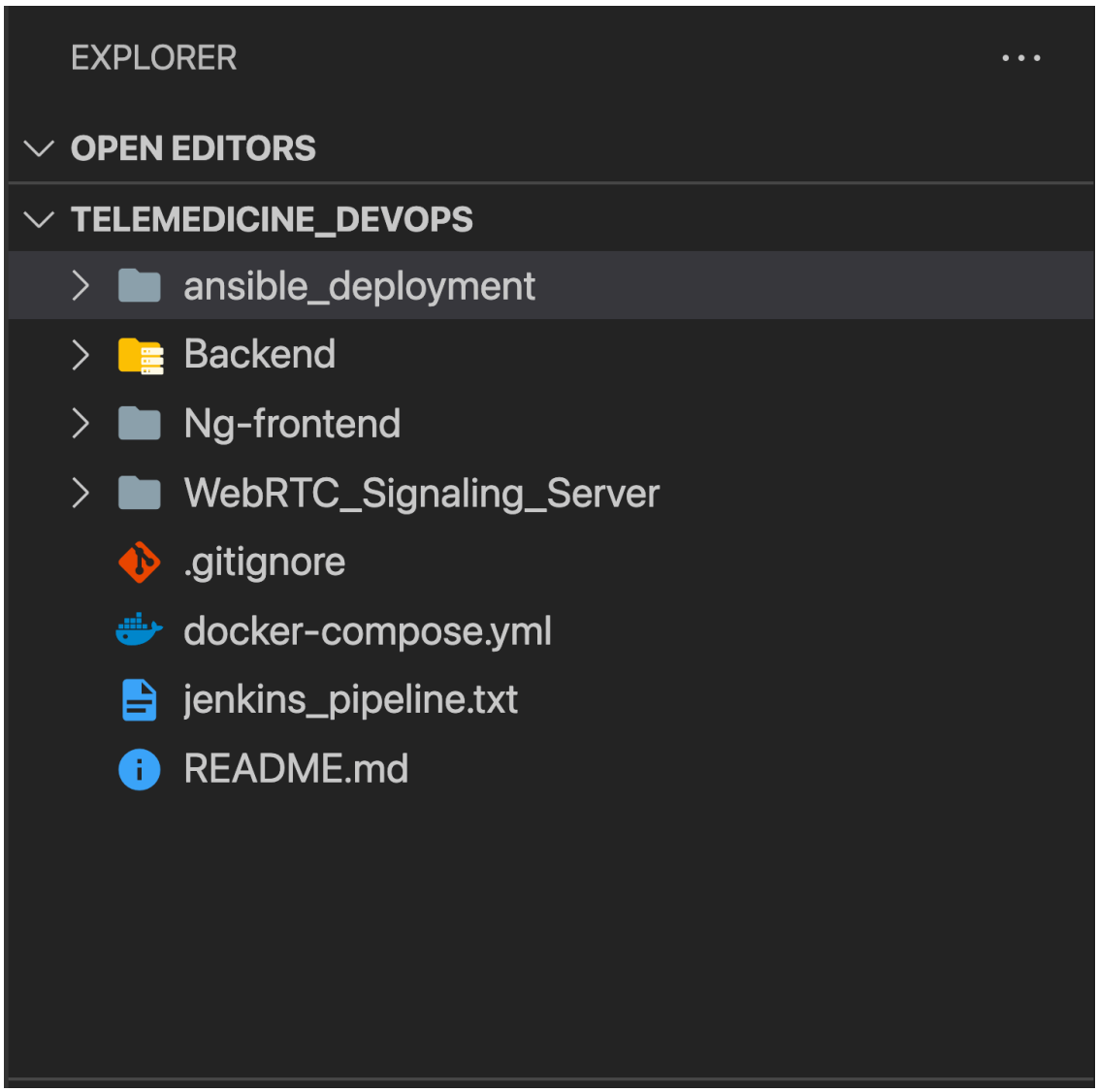

Figure 1 Project Structure

3.1.1 Frontend

Ng-frontend contains the code of frontend. Angular framework has been used for creating the frontend.

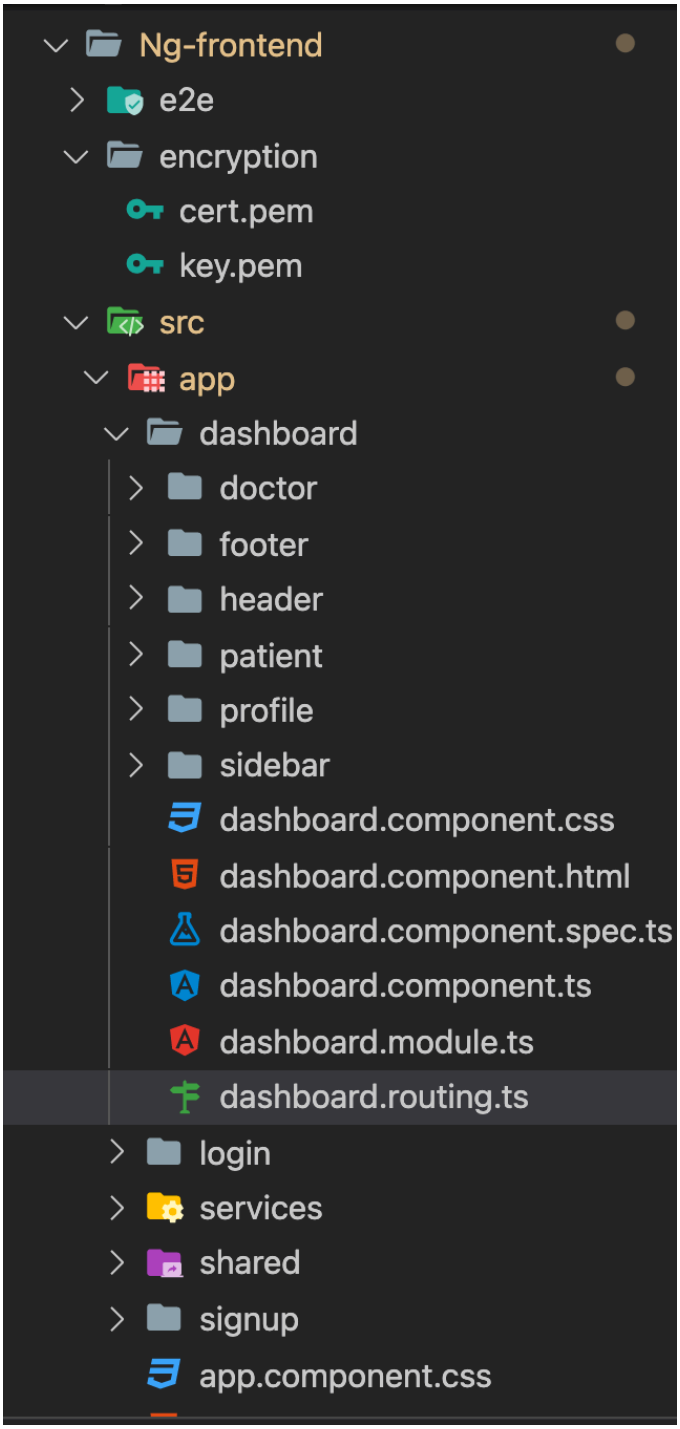

Figure 2 Frontend Components and Module

Above Image shows the list of component and modules that are there in the frontend. Also we have self singed ssl certificate which is essential for our application to work as it is accessing the Microphone and Web Camera of the computer. Following images show the different routes and the services of the frontend.

app.routing.ts M

Ng-frontend > src > app > app.routing.ts > ...

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { LoginComponent } from './login/login.component';
4  import { SignupComponent } from './signup/signup.component';
5
6  const routes: Routes = [
7    { path: '', redirectTo: 'login', pathMatch: 'full' },
8    { path: 'login', component: LoginComponent },
9    { path: 'signup', component: SignupComponent },
10 ];
11
12 @NgModule({
13   imports: [RouterModule.forRoot(routes)],
14   exports: [RouterModule]
15 })
16 export class AppRoutingModule { }
17
```

Figure 3 App Routing

dashboard.routing.ts

Ng-frontend > src > app > dashboard > dashboard.routing.ts > ...

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { AuthGuard } from '../services/auth.guard';
4  import { DashboardComponent } from './dashboard.component';
5  import { HomeComponent as PatientHomeComponent } from './patient/home/home.component';
6  import { HomeComponent as DoctorHomeComponent } from './doctor/home/home.component';
7  import { ProfileComponent } from './profile/profile.component';
8
9  const routes: Routes = [
10   {
11     path: 'dashboard',
12     component: DashboardComponent,
13     children: [
14       { path: '', redirectTo: 'home', pathMatch: 'full' },
15       { path: 'doctor/home', component: DoctorHomeComponent, canActivate: [AuthGuard] },
16       { path: 'patient/home', component: PatientHomeComponent, canActivate: [AuthGuard] },
17       { path: 'profile', component: ProfileComponent, canActivate: [AuthGuard] },
18     ]
19   },
20   { path: '**', redirectTo: '' }
21 ];
22
23 @NgModule({
24   imports: [RouterModule.forChild(routes)],
25   exports: [RouterModule]
26 })
27 export class DashboardRoutingModule { }
28
```

Figure 4 Dashboard Routing

auth-service.service.ts

Ng-frontend > src > app > services > auth-service.service.ts > ...

```typescript
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';
import { baseUrl, webrtcServerUrl } from 'src/environments/environment';
import { map } from 'rxjs/operators';
import { WebrtcService } from './webrtc.service';


@Injectable({
  providedIn: 'root'
})
export class AuthServiceService {

  private currentUserSubject: BehaviorSubject<any>;
  public currentUser: Observable<any>;

  constructor(private http: HttpClient) {
    this.currentUserSubject = new BehaviorSubject(JSON.parse(localStorage.getItem('currentUser')));
    this.currentUser = this.currentUserSubject.asObservable();
  }

  public get getcurrentUser() {
    this.currentUserSubject = new BehaviorSubject(JSON.parse(localStorage.getItem('currentUser')));
    this.currentUser = this.currentUserSubject.asObservable();
    return this.currentUserSubject.value;
  }

  validate(): Observable<any> {
    return this.http.get(`${baseUrl}users/validate`);
  }

  login(data): Observable<any> {
    return this.http.post(`${baseUrl}users/login/`, data)
      .pipe(map(results => {
        if (results['success']) {
```

Figure 5 Auth Service

users.service.ts

Ng-frontend > src > app > services > users.service.ts > ...

```typescript
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { baseUrl } from 'src/environments/environment';

@Injectable({
  providedIn: 'root'
})
export class UsersService {

  constructor(private http: HttpClient) { }

  update(data): Observable<any> {
    return this.http.patch(`${baseUrl}users/`, data)
      .pipe(map(data => {
        return data;
      }));
  }

  getDoctors(specialization) {
    return this.http.post(`${baseUrl}users/getdoctors/`, { 'specialization': specialization });
  }

  getWaitingPatients(doctor_id){
    return this.http.post(`${baseUrl}users/getwaitingpatients/`, { 'doctor_id': doctor_id });
  }

  removePatientFromWaitlist(patient_room_id){
    return this.http.post(`${baseUrl}users/removefromwaitlist/`, { 'room_id' : patient_room_id });
  }

  addPatientToWaitList(doctor_id, roomId){
      const currentUser = JSON.parse(localStorage.getItem('currentUser'));
      const patient_id = currentUser['id'];
```

Figure 6 User Service

webrtc.service.ts

Ng-frontend > src > app > services > webrtc.service.ts > ...

```ts
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { webrtcServerUrl } from 'src/environments/environment';

@Injectable({
  providedIn: 'root'
})
export class WebrtcService {

  constructor(private http: HttpClient) { }

 createRoom() {
    return this.http.get(`${webrtcServerUrl}/createRoom`);
  }

  joinRoom(roomId) {
    return this.http.get(`${webrtcServerUrl}/joinRoom?roomId=${roomId}`);
  }

  createNamespace(namespace_id) {
    return this.http.get(`${webrtcServerUrl}/createNamespace?namespace_id=${namespace_id}`);
  }

  removeNamespace(namespace_id) {
    return this.http.get(`${webrtcServerUrl}/removeNamespace?namespace_id=${namespace_id}`);
  }

  chekckAvailabilityOfDoctor(namespace_id){
      return this.http.get(`${webrtcServerUrl}/checkAvailability?namespace_id=${namespace_id}`);
  }

}

```

Figure 7 webrtc service

3.1.2 Backend

For the backend we have used express and node. Following figure shows the list of files and middleware (left panel) we have used in the backend and main app.js file. It also contains dockerfile which is used for docker containerization. Backend is used for Authentication and CRUD operation associated with the application.

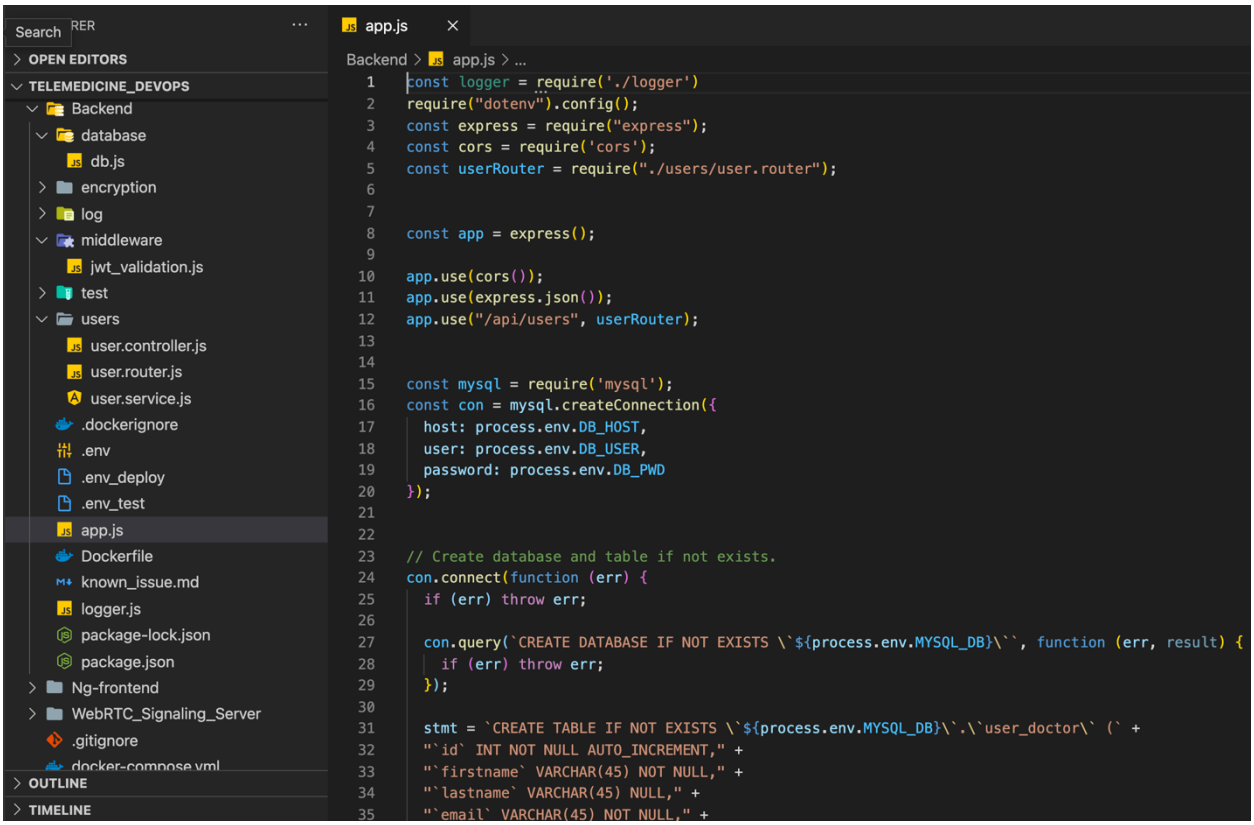

Figure 8 Backend and app.js

3.1.3 WebRTC Signaling Server

For the WebRTC Signaling Server, we have used express and node. It has a single file webrtc-server.js, which is useful for establishing the connection between peers (Doctor and Patient). It also contains dockerfile which is used for docker containerization.

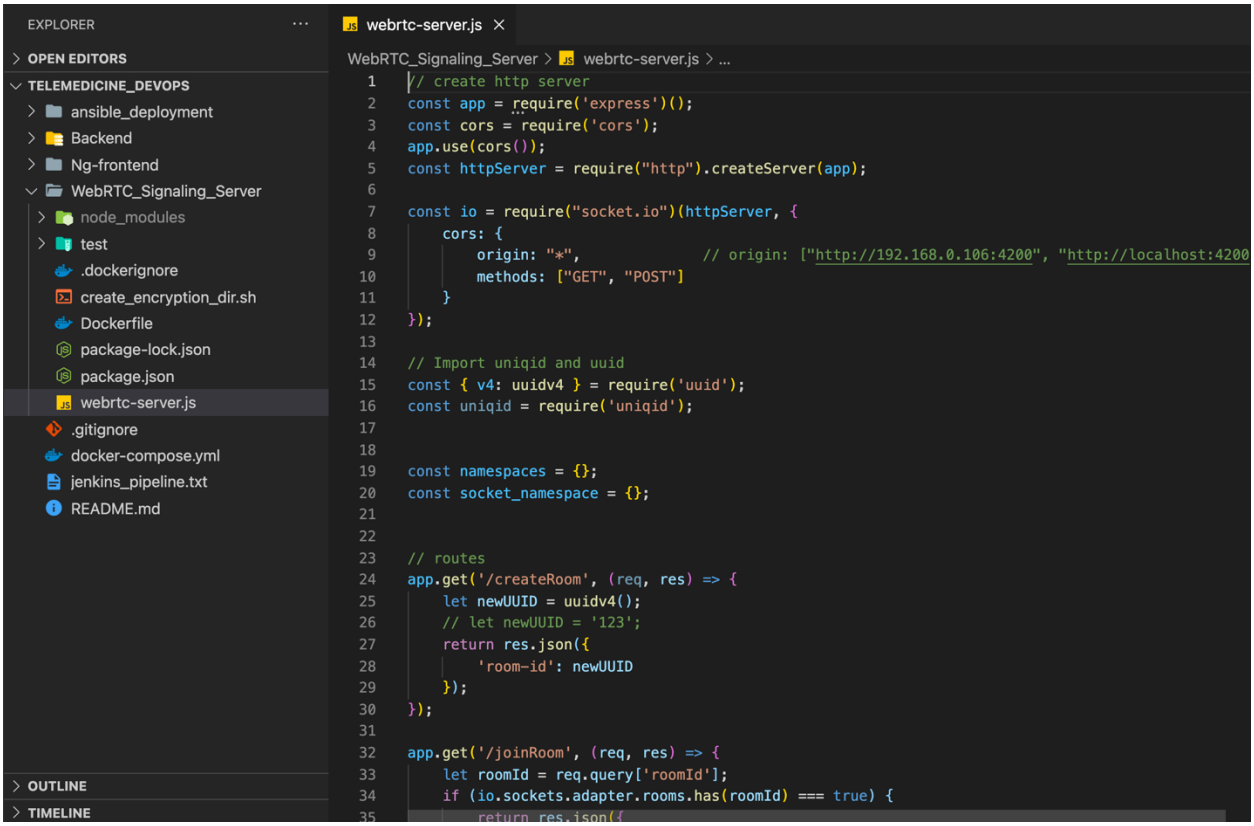

Figure 9 WebRTC signaling server

3.2 Source Code Management (SCM)

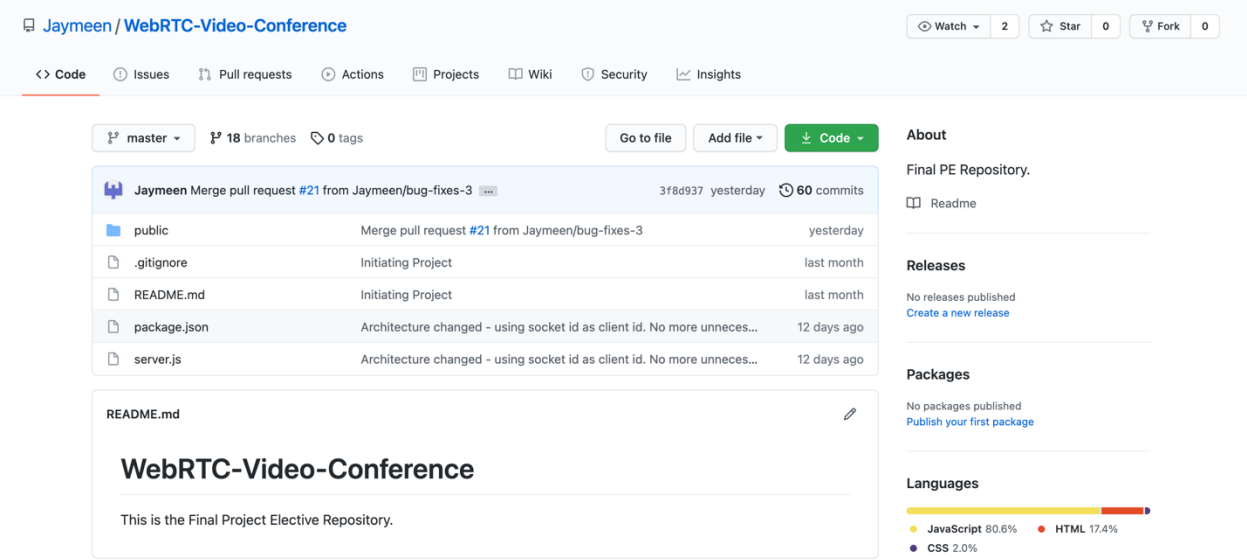

Figure 10 WebRTC-Video-Conference

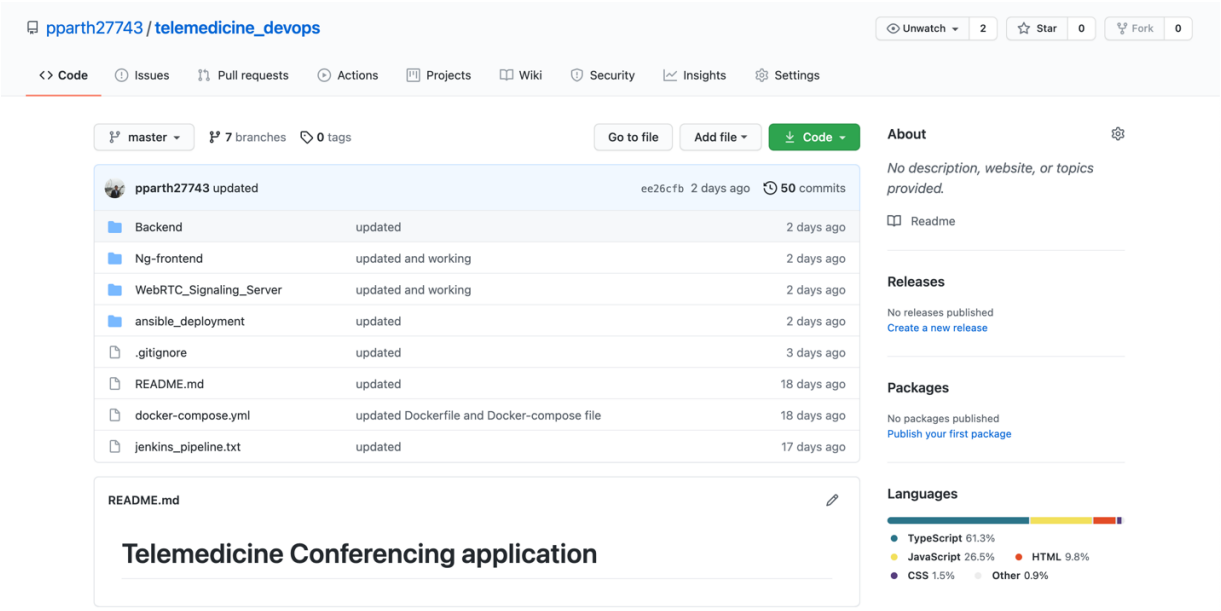

Figure 11 Full-Fledged Angular WebRTC Application

Following is the GitHub repository for of the Project,
WebRTC-Video-Conference: https://github.com/Jaymeen/WebRTC-Video-Conference
Full-Fledged Angular WebRTC Application:
https://github.com/pparth27743/telemedicine_devops

3.3 Building

We have used node for building our frontend application. Following figure shows the build files of frontend.

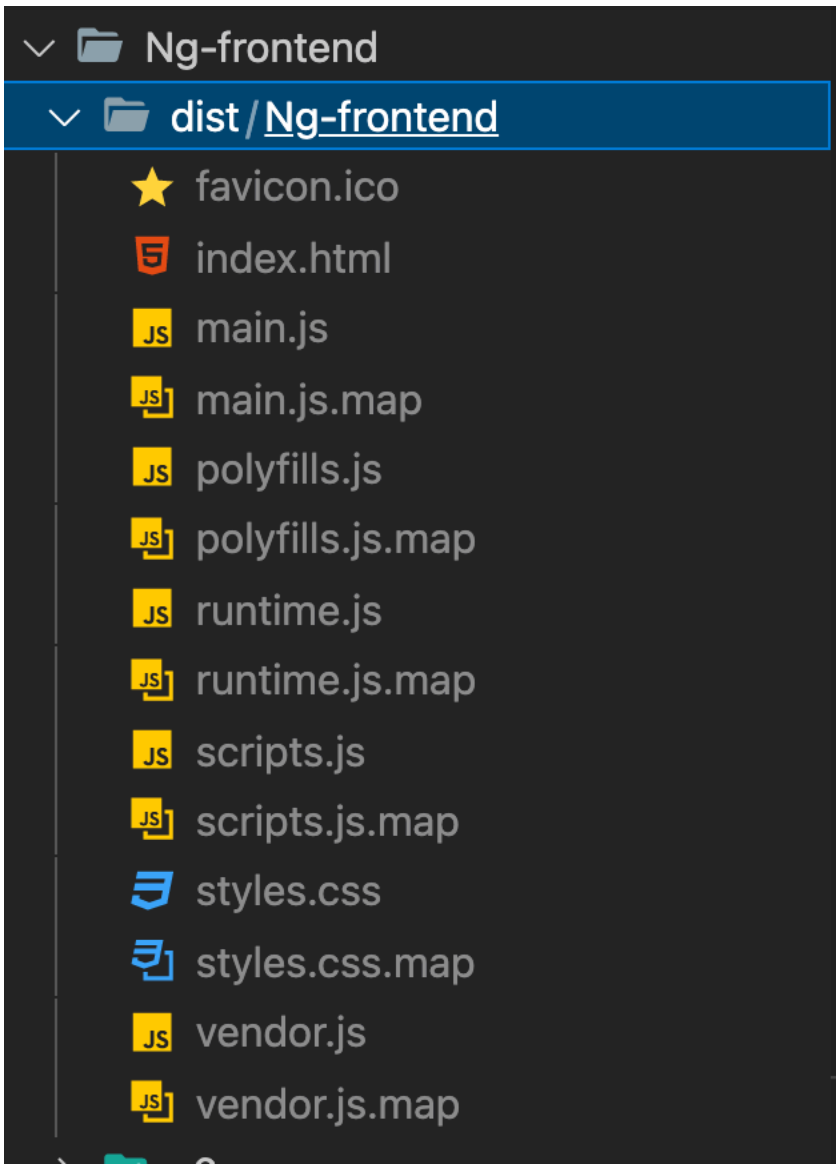

Figure 12 Build of frontend

3.4 Testing

We have used mocha which is a testing JavaScript library. We have done testing of various functionality related to CRUD operation and Authentication in the Backend and also testing of WebRTC server functionality such as create room, checking room availability, adding and removing namespace of socket.io etc,.

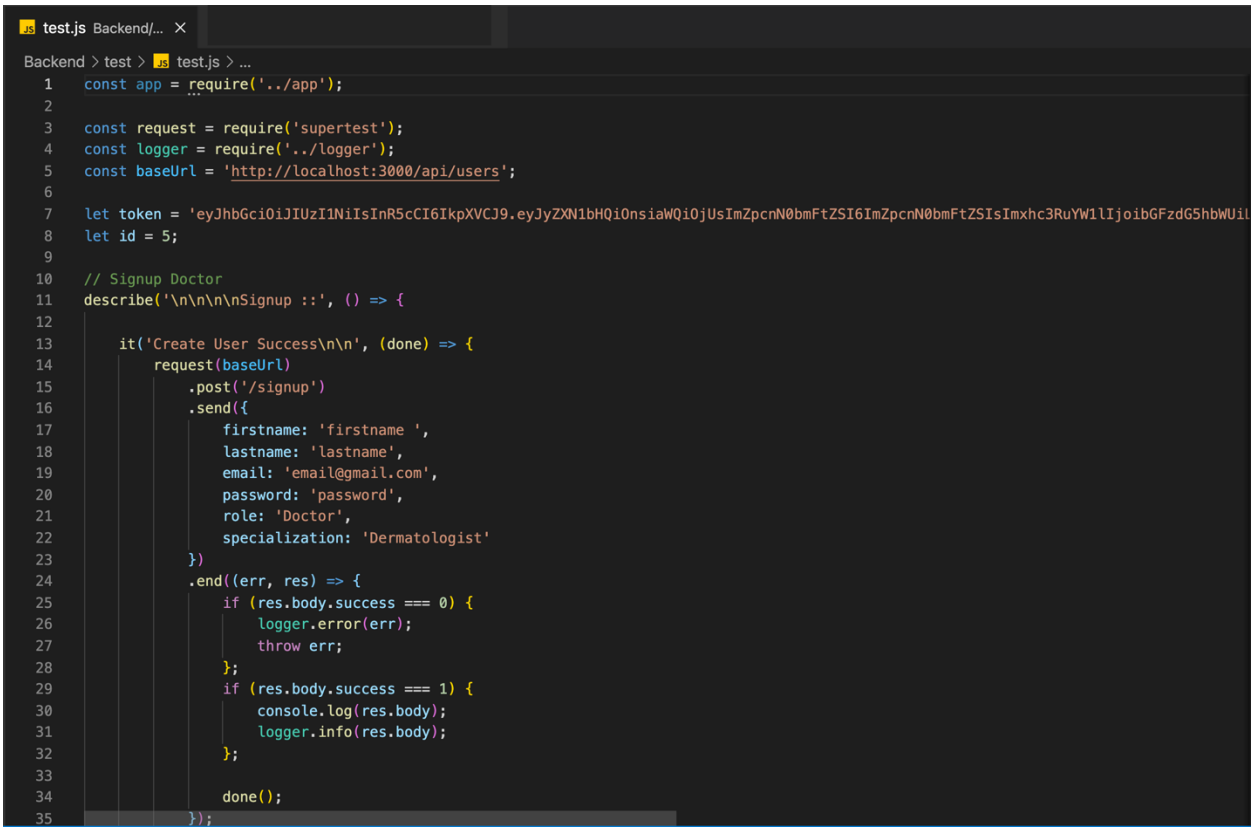

Figure 13 Backend test.js

test.js WebRTC_Signaling_Server/...

WebRTC_Signaling_Server > test > test.js > ...

```
const app = require('../webrtc-server');

const request = require('supertest');
const baseUrl = 'http://localhost:4440';

let roomId = '123';


// Get Room Id
describe('\n\nGet Room Id:: ', () => {
    it('Get Room Id \n', (done) => {

        request(baseUrl)
            .get('/createRoom')
            .end((err, res) => {
                if (!res.body['room-id']) throw err;
                else {
                    console.log(res.body);
                    roomId = res.body['room-id'];
                };
                done();
            });
    })
});

// Join Room
describe('\n\nJoin Room :: ', () => {

    it('Join Room with given Id \n', (done) => {
        request(baseUrl)
            .get(`/joinRoom?roomId=${roomId}`)
            .end((err, res) => {
                console.log(res.body);
                done();
            });
```

Figure 14 WebRTC Server test.js

3.5 Logging

We have used Winston, which is logging JavaScript library. We have logged each and every event of the Backend that is happing there. These log file created later used in ELK for monitoring of the application.

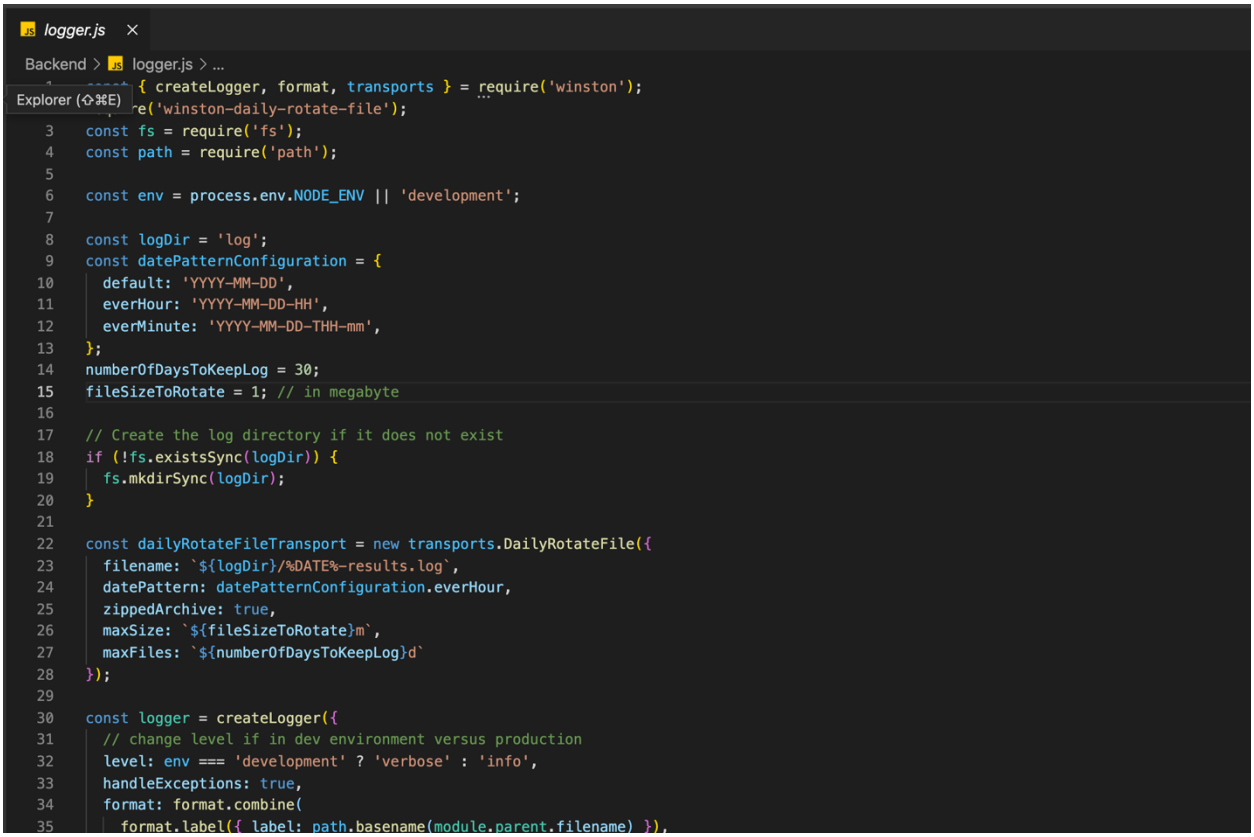

Figure 15 logger.js

3.6 Docker, Docker Compose and DokcerHub

Docker container is popularly known as a lightweight virtual machine that only contains the application and its dependencies. On a machine, we can run thousands of containers whereas fewer Virtual Machines can be created on the machine.
To create a docker container of our application we need to create a Docker Image and for that, we first need to create Dockerfile which has all the instructions and dependencies that are downloaded when the image is built.
There are 3 different Dockerfile we have created for frontend, backend and webrtc-signaling-server as follows.

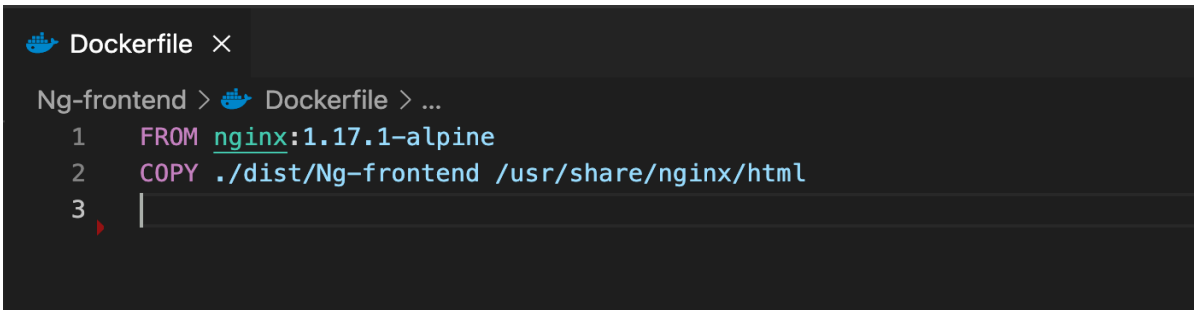

Figure 16 Dockerfile for frontend

Figure 17 Dockerfile for Backend

Dockerfile

WebRTC_Signaling_Server > Dockerfile > ...

```
# Base image
FROM node:14.15-alpine

WORKDIR /webrtc_server
COPY . .

RUN     npm i
CMD [ "npm", "start" ]

```

Figure 18 Dockerfile for WebRTC Server

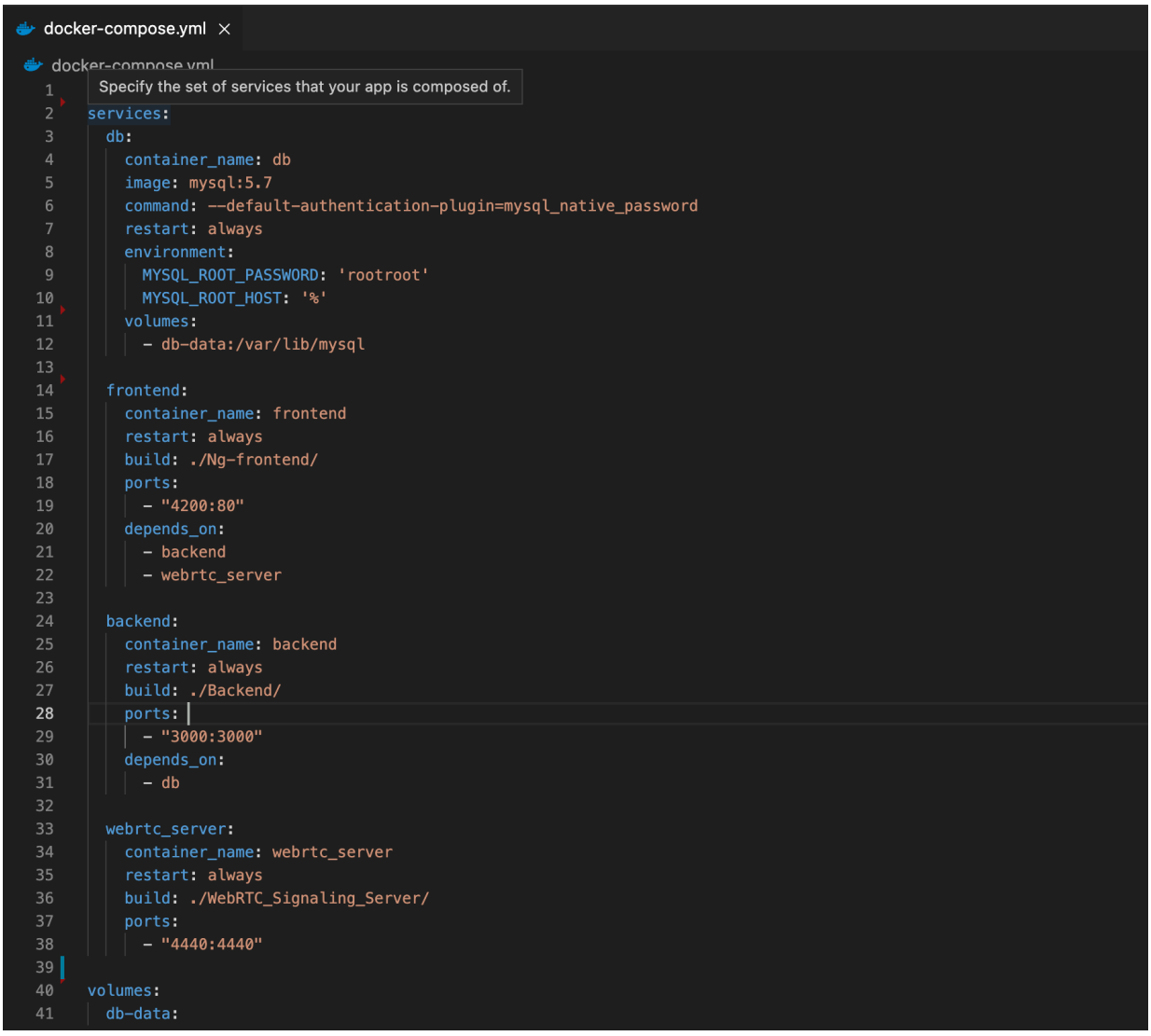

Figure 19 Docker Compose File

Docker compose is used to do docker containerization. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration[6].

Docker Compose file is creating 4 services namely db, frontend, backend and webrtc_server. Here we used docker volume for persistent storage which is attached to db (database) service.

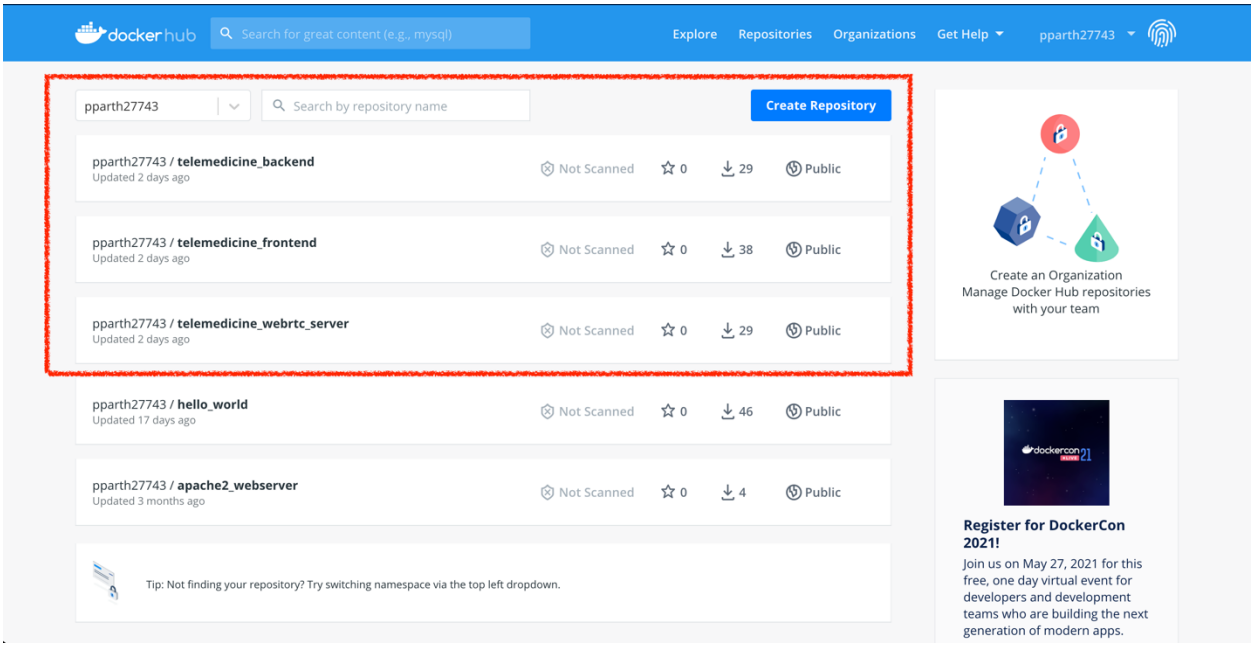

Figure 20 Docker Images uploaded on DockerHub

3.7 Ansible

Ansible is a simple open-source IT engine that automates application deployment, intra service orchestration, cloud provisioning, and many other IT tools.
Ansible is easy to deploy as it is agentless tool. It only needs to be installed on the controlled node. The managed node need not install ansible in it. Ansible uses a playbook to define tasks that need to be performed on the managed host and these tasks are known as play in an ansible world. Ansible uses ssh for logging into the managed host. And this host information is mentioned inside the inventory file. Ansible playbooks are in a very simple language that is YAML.

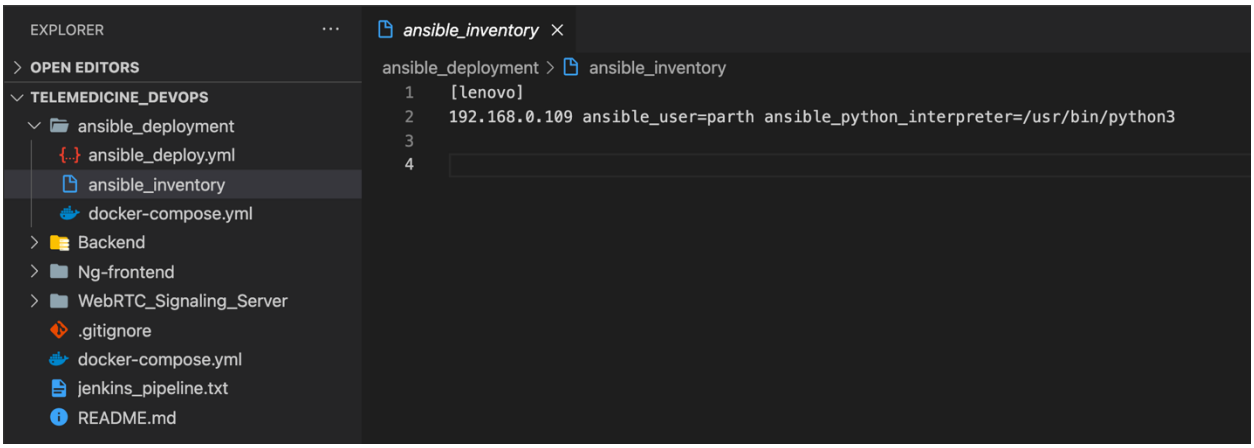

Figure 21 Ansible Inventory file

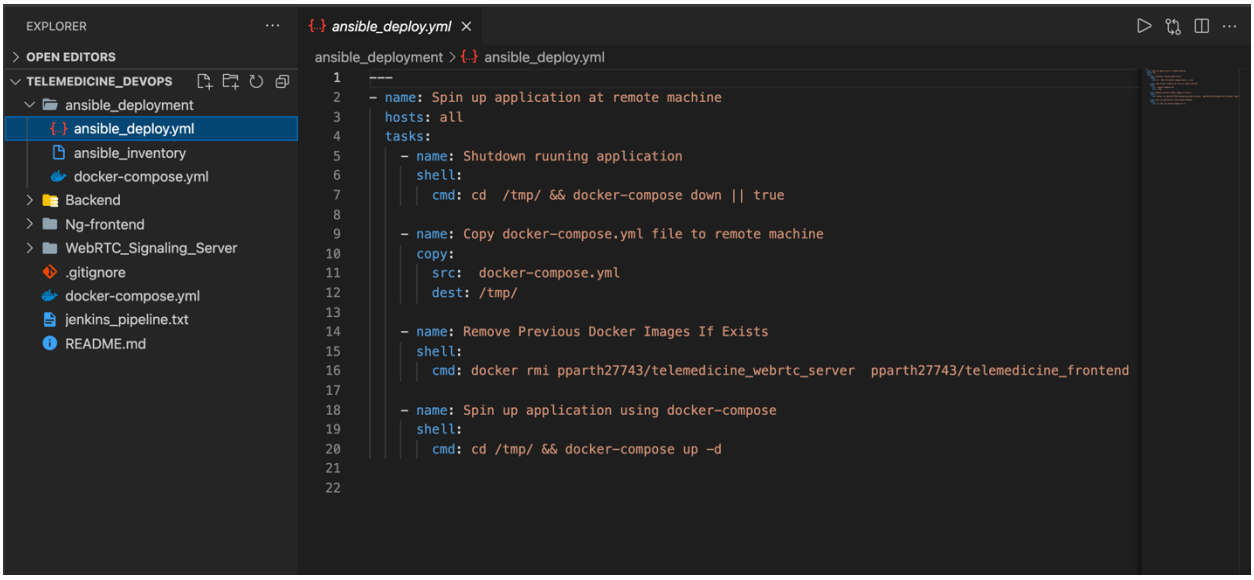

Figure 22 Ansible playbook

Ansible Playbook does series of operation on managed host staring from Shutting down already running application to deployment of updated application using docker compose file.

3.8 Jenkins and Pipeline

All the above steps can be done in an automated fashion by using Jenkins. Jenkins is is a free and open-source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. Jenkins by default runs on 8080 port.
We need to install the necessary plugins for Git, Ansible and need to store credentials to push docker image on to docker hub.

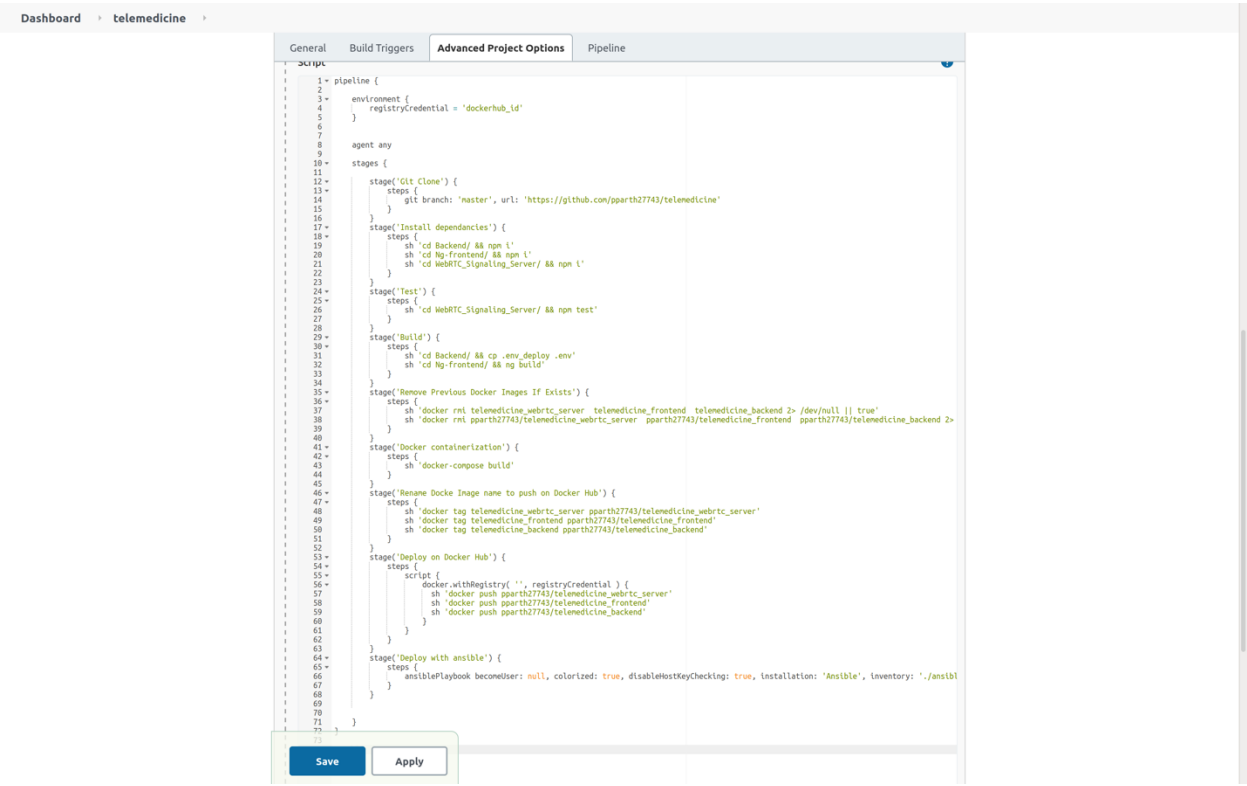

Figure 23 Pipeline Script

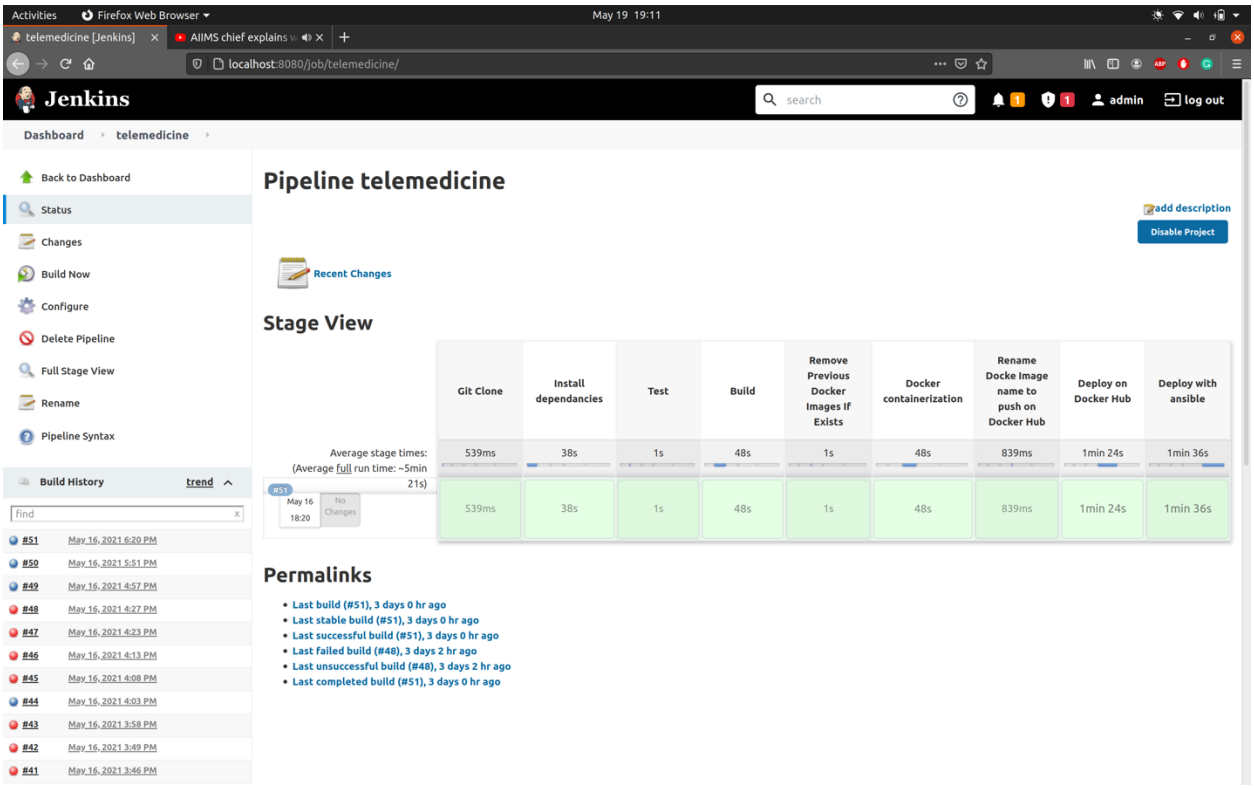

Figure 24 Build of Pipeline

3.9 Monitoring using ELK:

"ELK" is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. Elasticsearch is a search and analytics engine. Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch. Kibana lets users visualize data with charts and graphs in Elasticsearch. The Elastic Stack is the next evolution of the ELK Stack.

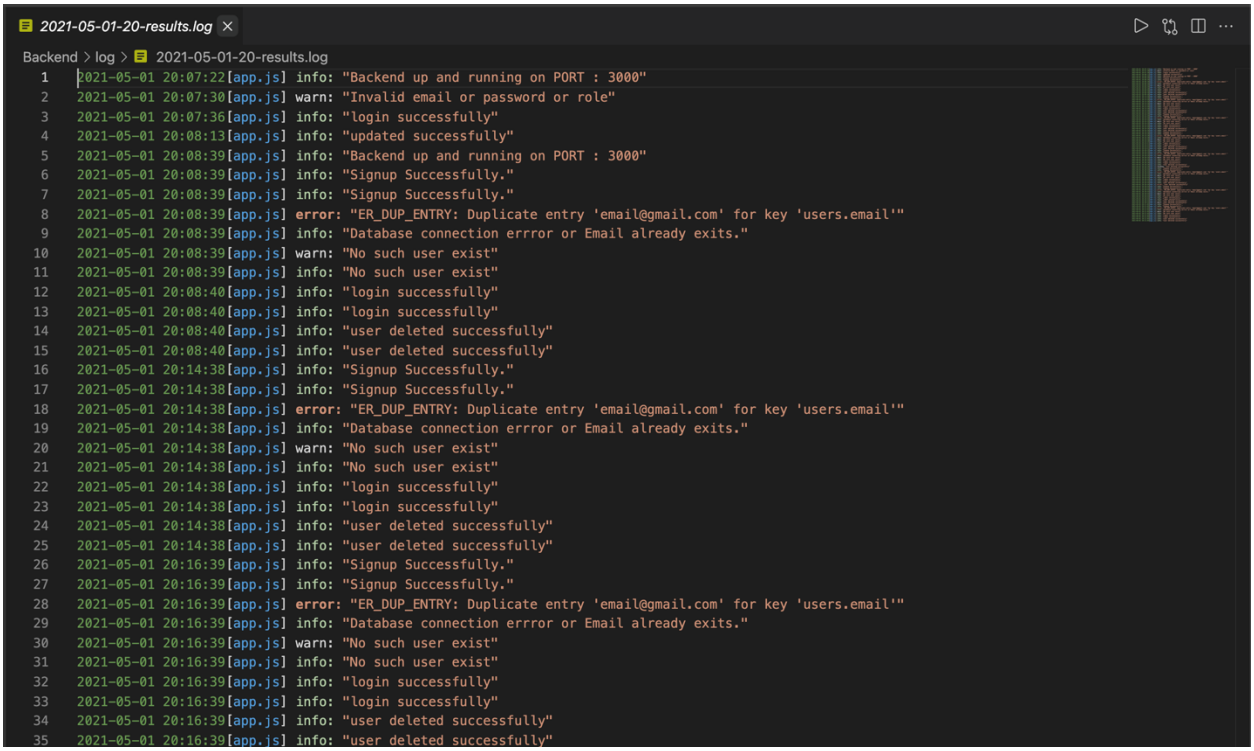

Figure 25 Log file

Now, this log file needs to be uploaded to https://cloud.elastic.co. It is an online service that provides ELK.

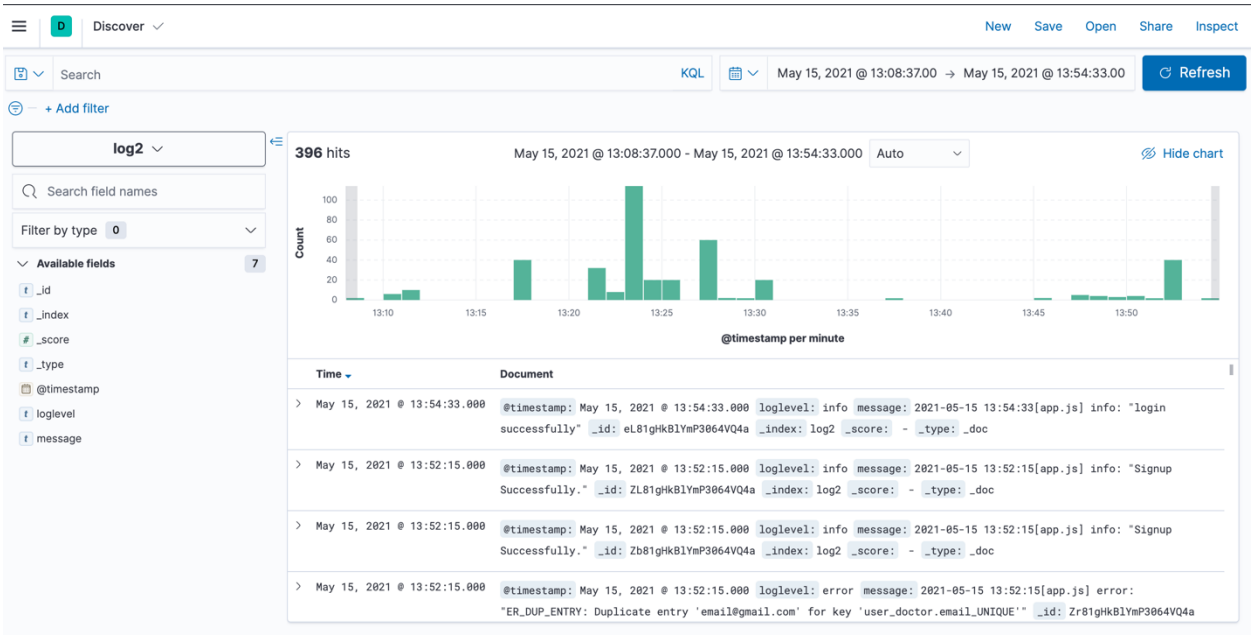

Figure 26 ELK logs

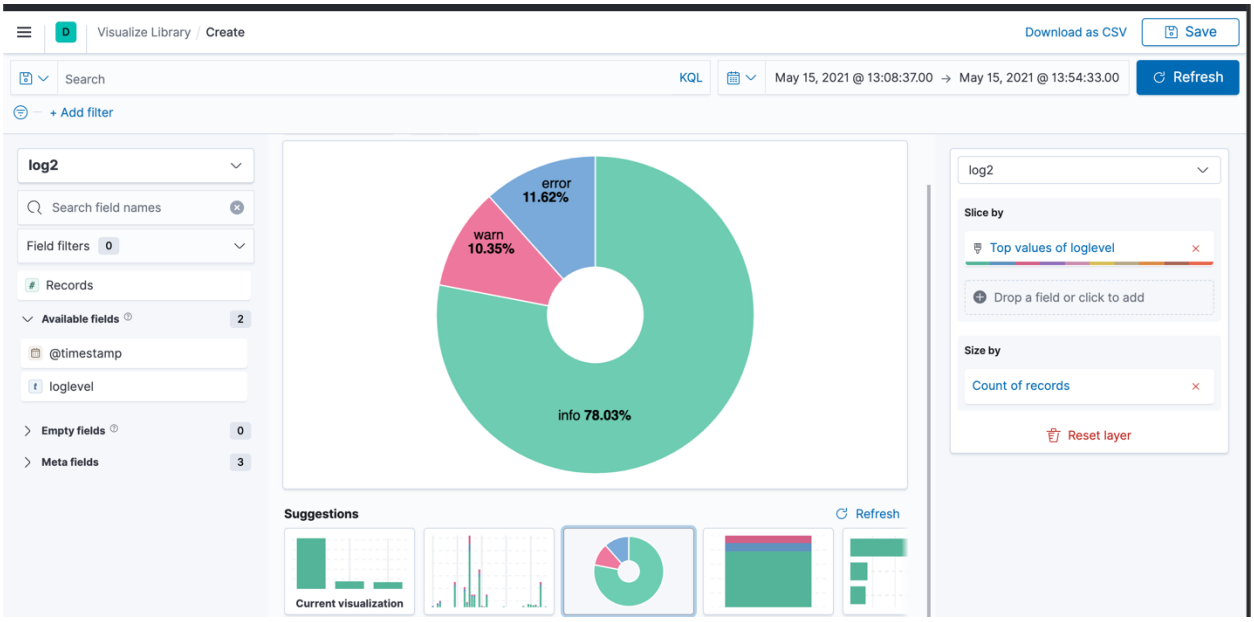

Figure 27 Graph Based on Log Level

4. Results and Discussion

4.1 Login Page

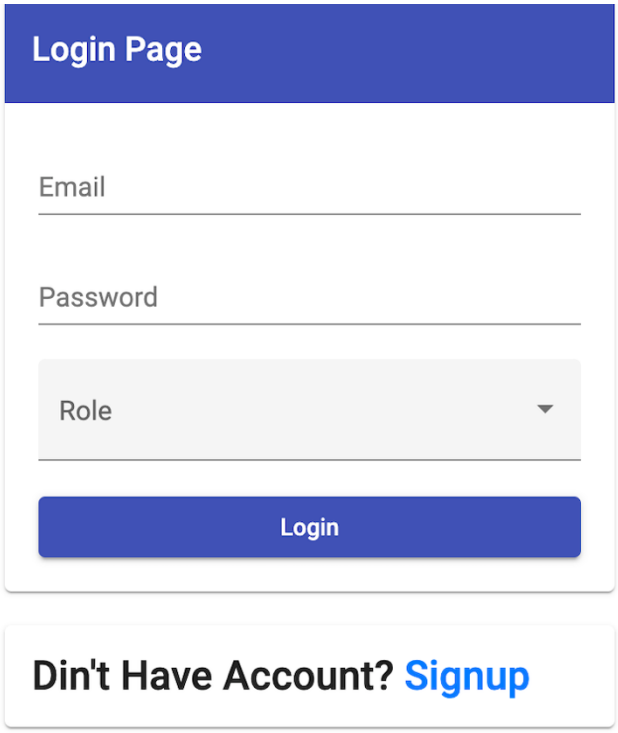

Figure 28 Login Page

4.2 Signup Page

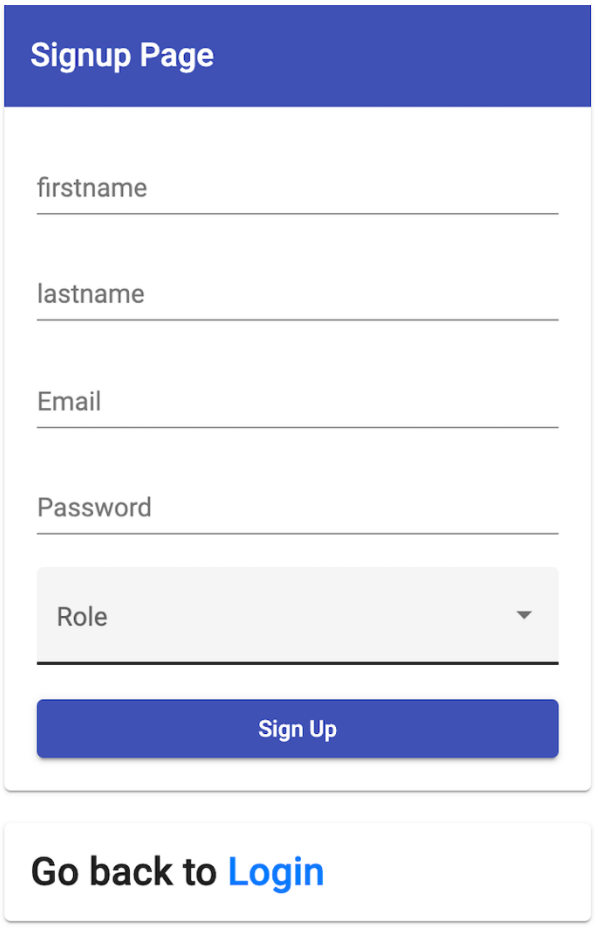

Figure 29 Signup Page

4.3 Doctor’s Home Page

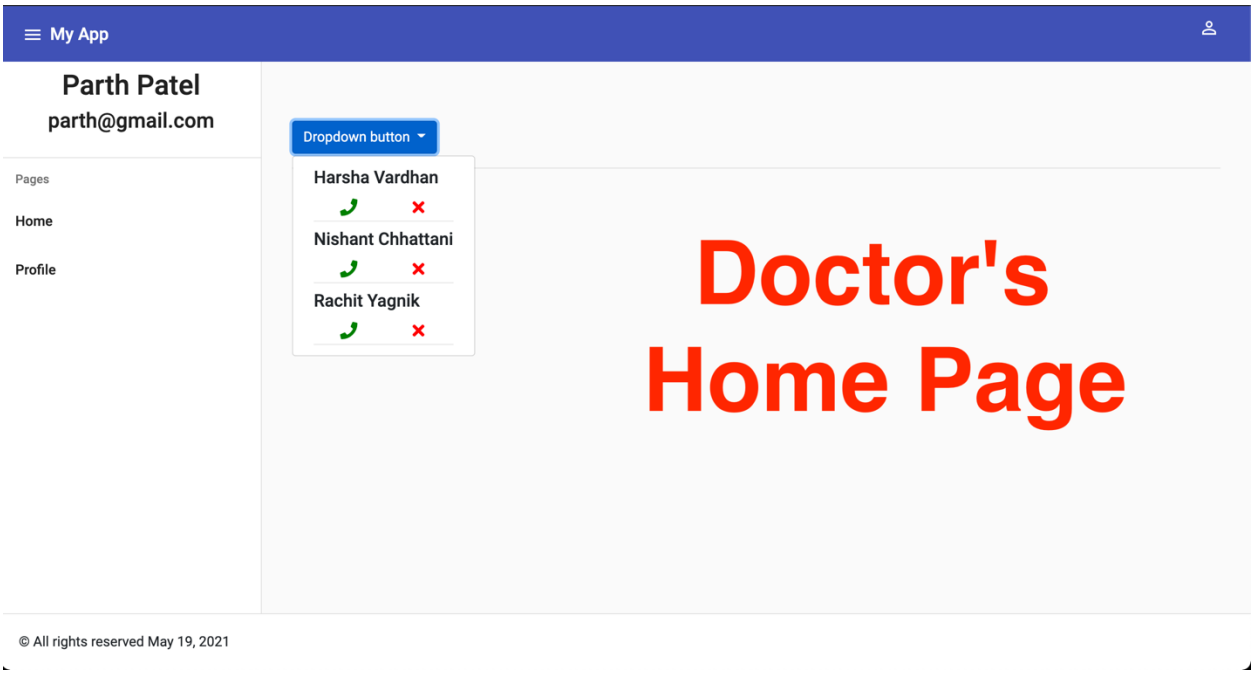

Figure 30 Doctor’s Home Page

4.4 Patient’s Home Page

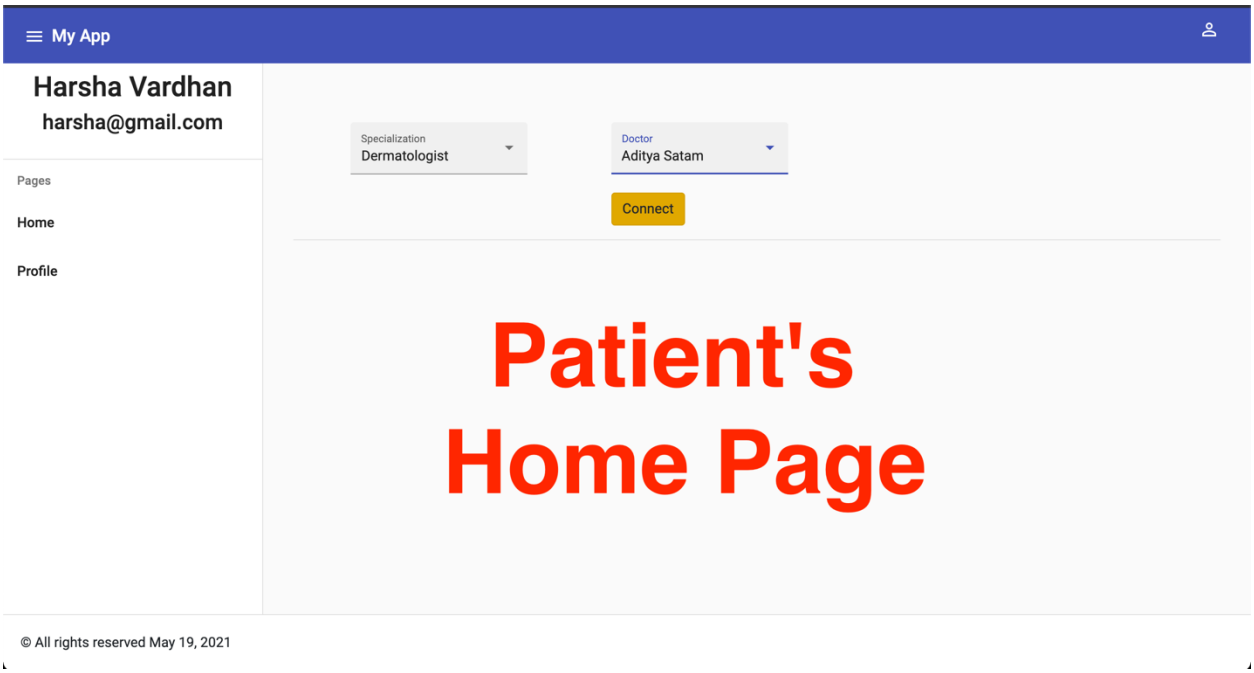

Figure 31 Patient's Home Page

4.5 Profile Update Page

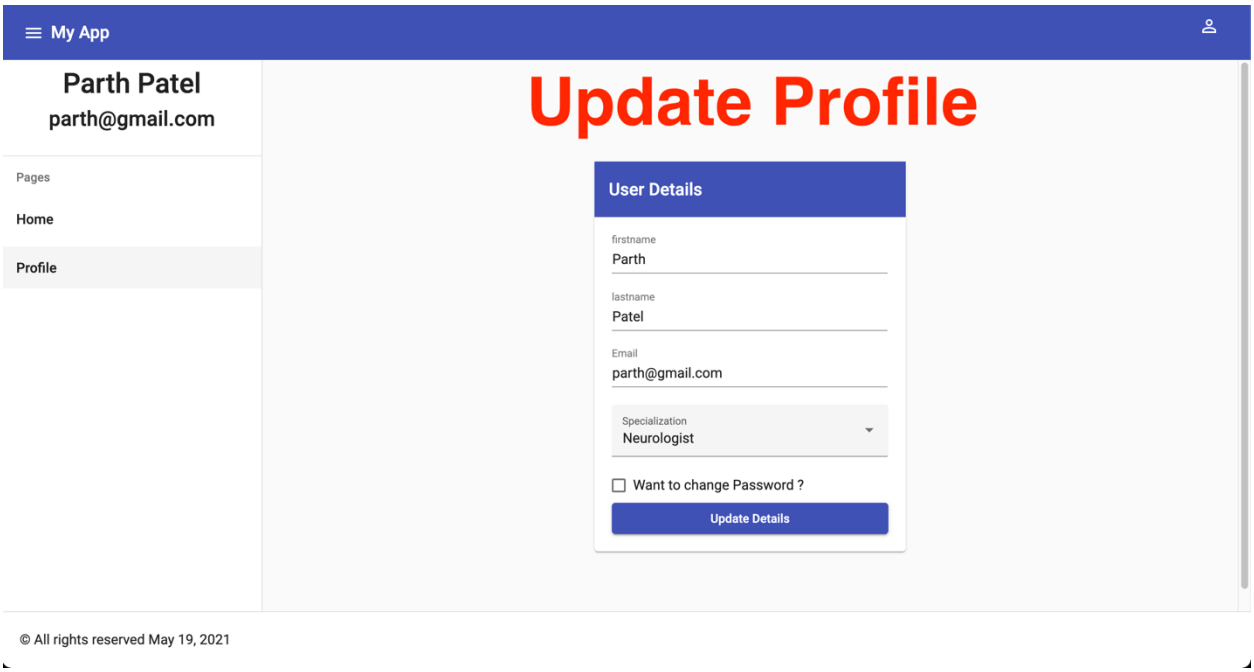

Figure 32 Profile Update Page

4.6 Doctor Consulting Patient

Here You can see Patient has added multiple steam in the call. Those stream are of Audio and Video through which Patient can send real-time heart rate or stethoscope audio so doctor can treat him/her precisely.

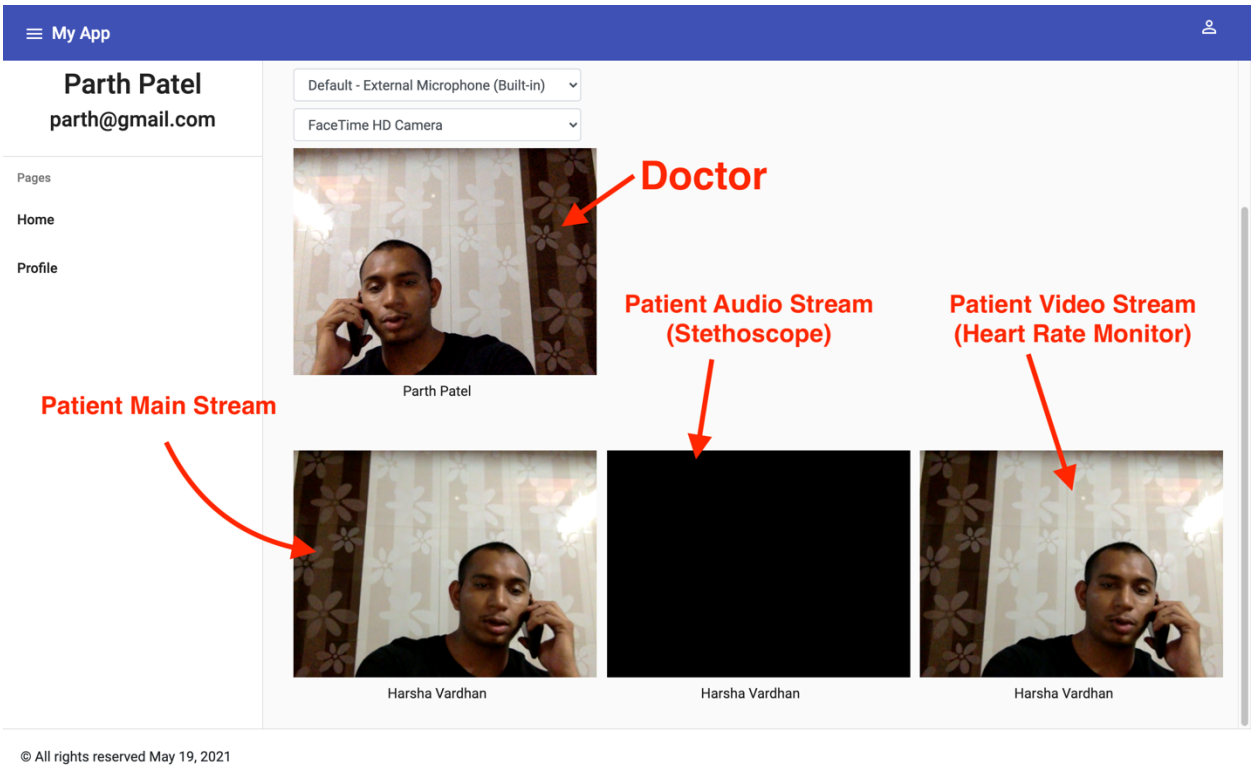

Figure 33 Doctor and Patient Having Call

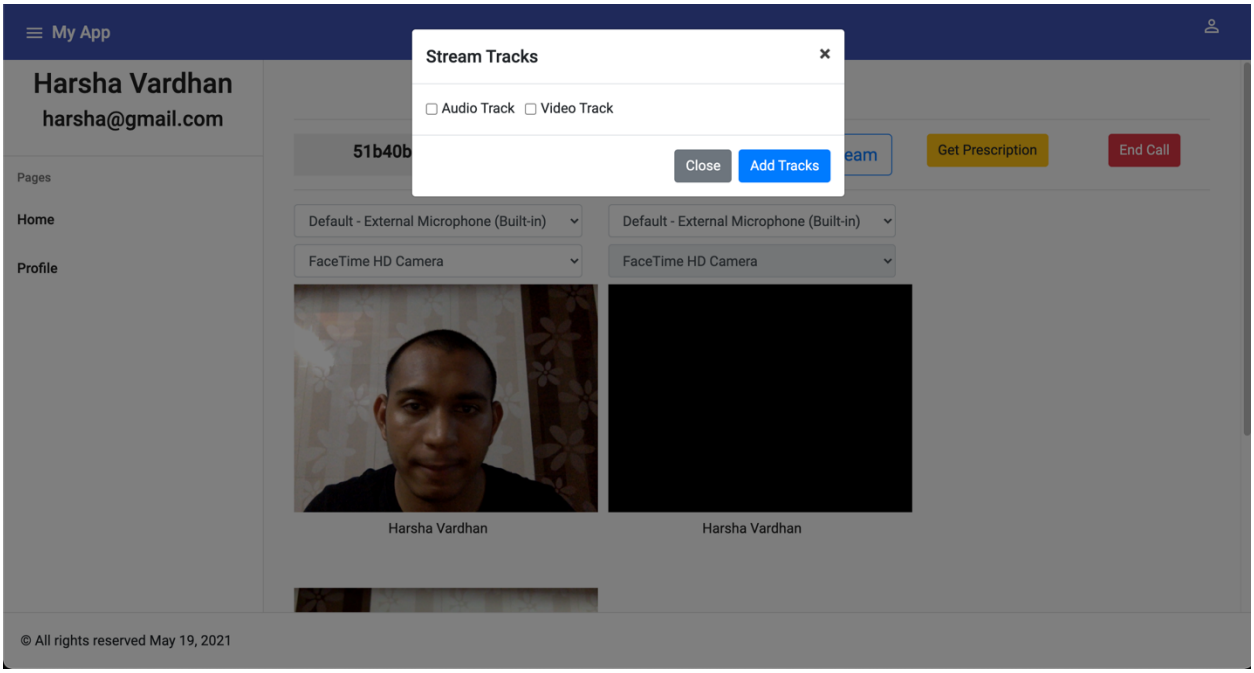

Figure 34 Add Stream

4.7 Doctor Writing Prescription for Patient

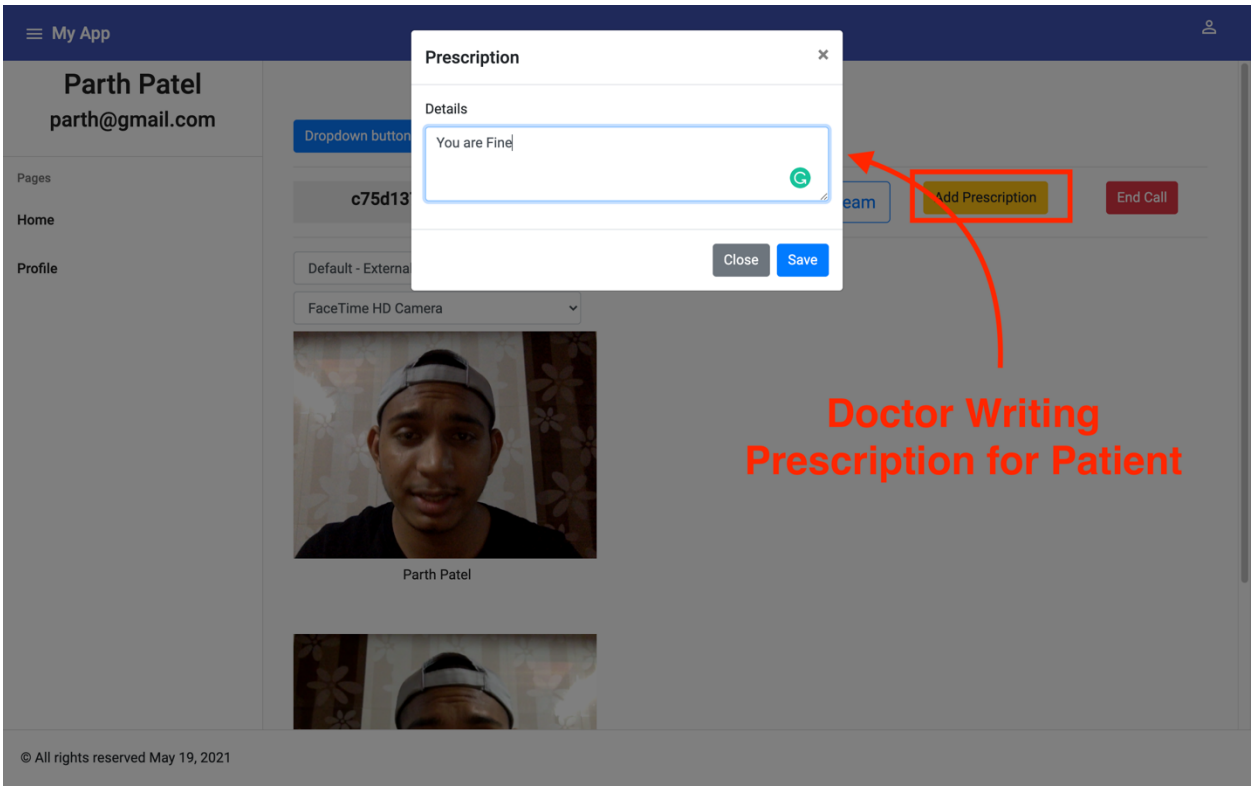

Figure 35 Doctor Writes Prescription

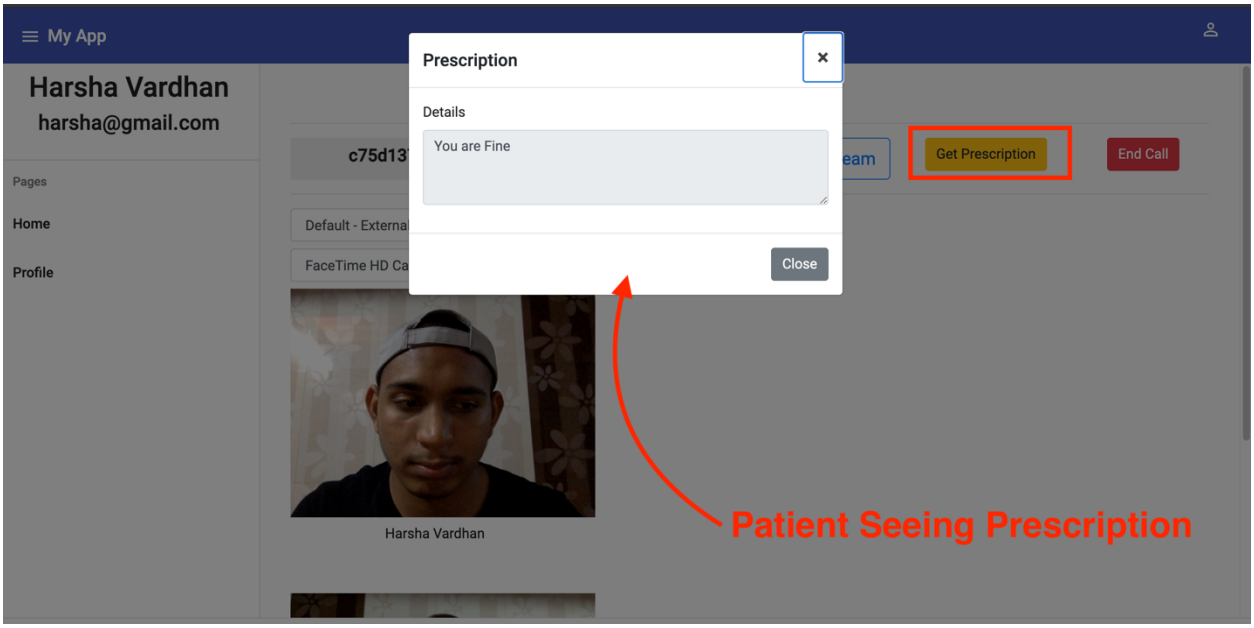

Figure 36 Patient Reads Prescription

5. Scope for future work

As This project is not complete in itself. There are many functionality can be added to this to make a production ready application. We just implemented Video Conferencing and Doctor writing prescription for the patient. But There are many such as text chat, file sharing, screen sharing, reminders etc. things can be added in this application. This was about functional features. But non functional things can be security enhancement, scalability, availability, efficiency can be added to the project as well.

6. Conclusion

We have successfully build a Telemedicine application through which Patient can take appointment online and consult Doctor remotely. In doing so, we have various DevOps tools like used GitHub, Jenkins, Docker, Docker Compose, Ansible. There tools are integrated using Jenkins to make entire SDLC cycle automated.

DevOps methodology and tools prove that it is better than Agile methodology for an application which is rich in features and required features to be deployed as soon as they are developed. And that's what modern enterprise required to stay in competition.

7. References:

[1] https://www.jenkins.io/doc/
[2] https://docs.ansible.com/ansible/latest/scenario_guides/guide_docker.html
[3] https://docs.github.com/en
[4] https://www.google.co.in/
[5] https://stackoverflow.com/
[6] https://docs.docker.com/compose/