

**SWE PROJECT REPORT**

***DEPARTMENT OF INFORMATION TECHNOLOGY***

***3rd Year***

**COLLABORATIVE ONLINE JOB PORTAL**

PREPARED BY:

**Name:** SAYAN MUNSHI **| Roll No.:** 27 **| Enrollment No.:** 12020002004031

**Name:** PRATYUGNA MANNA **| Roll No.:** 29 **| Enrollment No.:** 12020002004033

**Name:** SARVESH BHUSHAN **| Roll No.:** 31 **| Enrollment No.:** 12020002004035

**Name:** SHIVAM PANDEY **| Roll No.:** 33 **| Enrollment No.:** 12020002004037

**Group No.: 8**

**CERTIFICATE**

This is to certify that, **Group No. 8** of **3rd year IT,** consisting of students **Sayan Munshi, Pratyugna Manna, Sarvesh Bhushan, and Shivam Pandey** have successfully completed their **Software Engineering Project** titled **Collaborative Online Job Portal** under the guidance of **“Prof. Avijit Bose”.**

**Prof. Avijit Bose Prof Dr. Moutushi Singh,**

**HOD, IT**

TABLE OF CONTENT:

* Feasibility Report (Technical Feasibility)
* Planning (Technology and Tools)
* Decision Tree and Table
* Data Flow Diagram
* UML Diagrams
  + Use case Diagram
  + Class Diagram
  + Activity Diagram
  + Sequence Diagram
* Test Report and Results
* Main module codes and screenshots
* Bug Report
* Further Development
* Conclusion

TECHNICAL FEASIBILITY

Project **COLLABORATIVE INTERNAL JOB PORTAL** is a complete web 2.0 based collaborative intranet environment which facilitates internal sourcing. The main technologies and tools that are associated with OES are:-

* UI/UX DESIGN TOOLS: FIGMA
* HTML
* CSS
* TAILWIND CSS
* JAVASCRIPT
* REACT JS
* NODE JS
* EXPRESS JS
* MONGODB

Each of the technologies is freely available and the technical skills required are manageable. Time limitations of the product development and the ease of implementing using these technologies are synchronized.

The website will be hosted in a free domain. The bandwidth required in this application will be very low as it will be developed using the most up-to-date technology in an optimized way.

From these, it’s clear that the project COLLABORATIVE **INTERNAL JOB PORTAL** is technically feasible

PLANNING

**Technologies:**

* For designing our v1.0 we will use Figma Software as a UI/UX design tool.
* For developing our **COLLABORATIVE INTERNAL JOB PORTAL** we will be using the **MERN** stack as the base framework which comprises MongoDB, ExpressJS, ReactJS, and NodeJS.
* For developing the front end of our application we will use ReactJS as the base framework. Vanilla CSS and Tailwind CSS will be used to design our application's front end. ReactJS will use Javascript as its base programming language.
* For the backend framework we will use Node JS and Express JS.
* We will use MongoDB as the database.

**Tools:**

* Windows will be the preferred OS.
* VS Code will be our preferred text editor.
* We will use MongoDB Compass as the platform for managing the Database.

**Contributions:**

* To provide a platform that can be used by HRs and PMs to post internal job requirements.
* To share Job Postings and comments across different communities.
* To provide a one-stop point for all employees to apply or view job postings, comment on postings,                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           post job requirements, etc.

**Life Cycle Mode:**

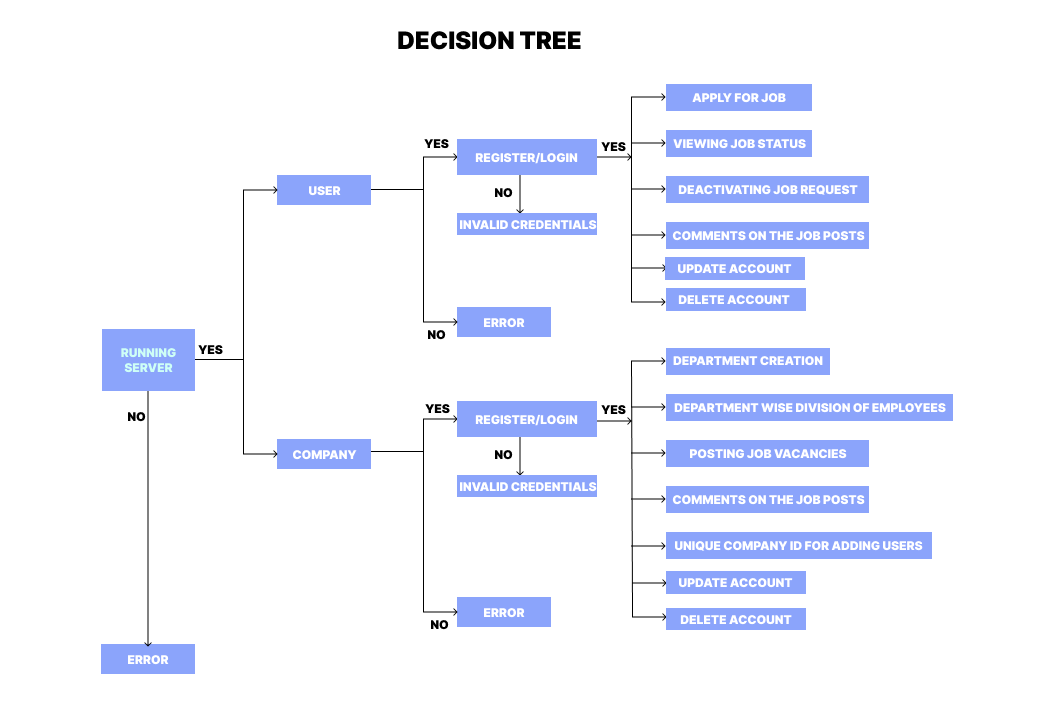
We will be using the **Agile** model.

**Reason:** Agile method break tasks into smaller iterations that do not directly involve long time planning.



DECISION TREE

A decision tree is a graph that uses a branching method to illustrate every possible output for a specific input. Decision trees can be drawn by hand or created with a graphics program or specialized software. Informally, decision trees are useful for focusing discussion when a group must make a decision.

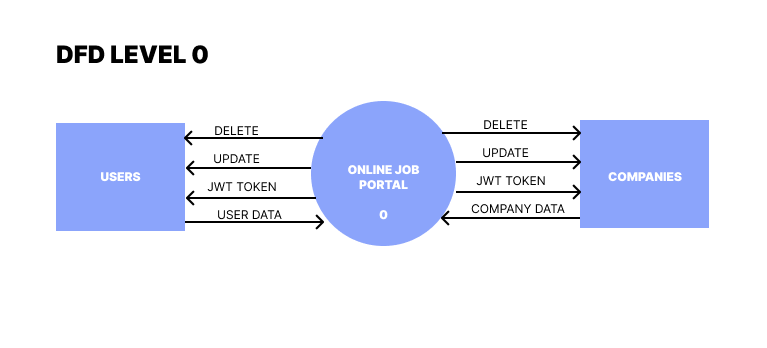


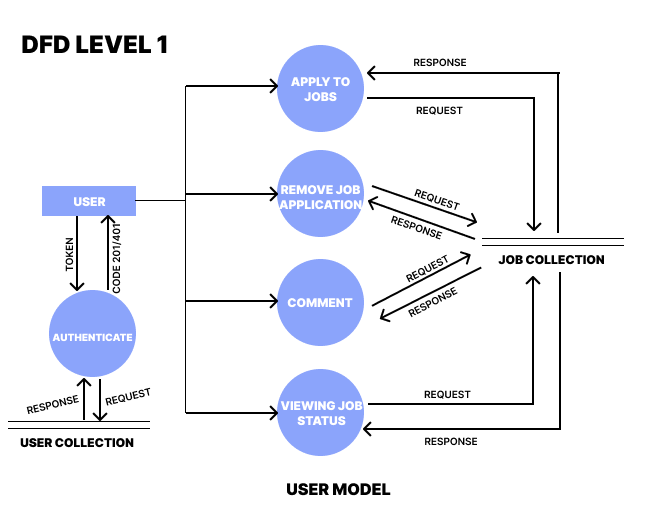
**DECISION TABLE:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **FUNCTIONS** | **T1** | **T2** | **T3** | **T4** | **T5** |
| RUNNING SERVER | **✔** | **✔** | **✔** |  | **X** |
| REGISTER/LOGIN (USER) | **✔** |  |  |  | **X** |
| UPDATE ACCOUNT (USER) |  |  | **✔** |  | **X** |
| DELETE ACCOUNT (USER) |  |  |  | **✔** | **X** |
| REGISTER/LOGIN (COMPANY) |  | **✔** |  |  | **X** |
| UPDATE ACCOUNT (COMPANY) |  |  | **✔** |  | **X** |
| DELETE ACCOUNT (COMPANY) |  |  |  | **✔** | **X** |
| APPLY FOR JOB | **✔** |  |  |  | **X** |
| VIEWING JOB STATUS | **✔** |  |  |  | **X** |
| CANCELLING APPLICATIONS | **✔** |  |  |  | **X** |
| COMMENTING ON POSTS | **✔** |  |  |  | **X** |
| CREATING DIVISIONS |  | **✔** |  |  | **X** |
| POSTING JOB VACANCIES |  | **✔** |  |  | **X** |
| COMPANY ID |  | **✔** |  |  | **X** |
| COMMENT |  | **✔** |  |  | **X** |
| ERROR |  |  |  |  | **✔** |

DATA FLOW DIAGRAM

A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It can be manual, automated, or a combination of both.

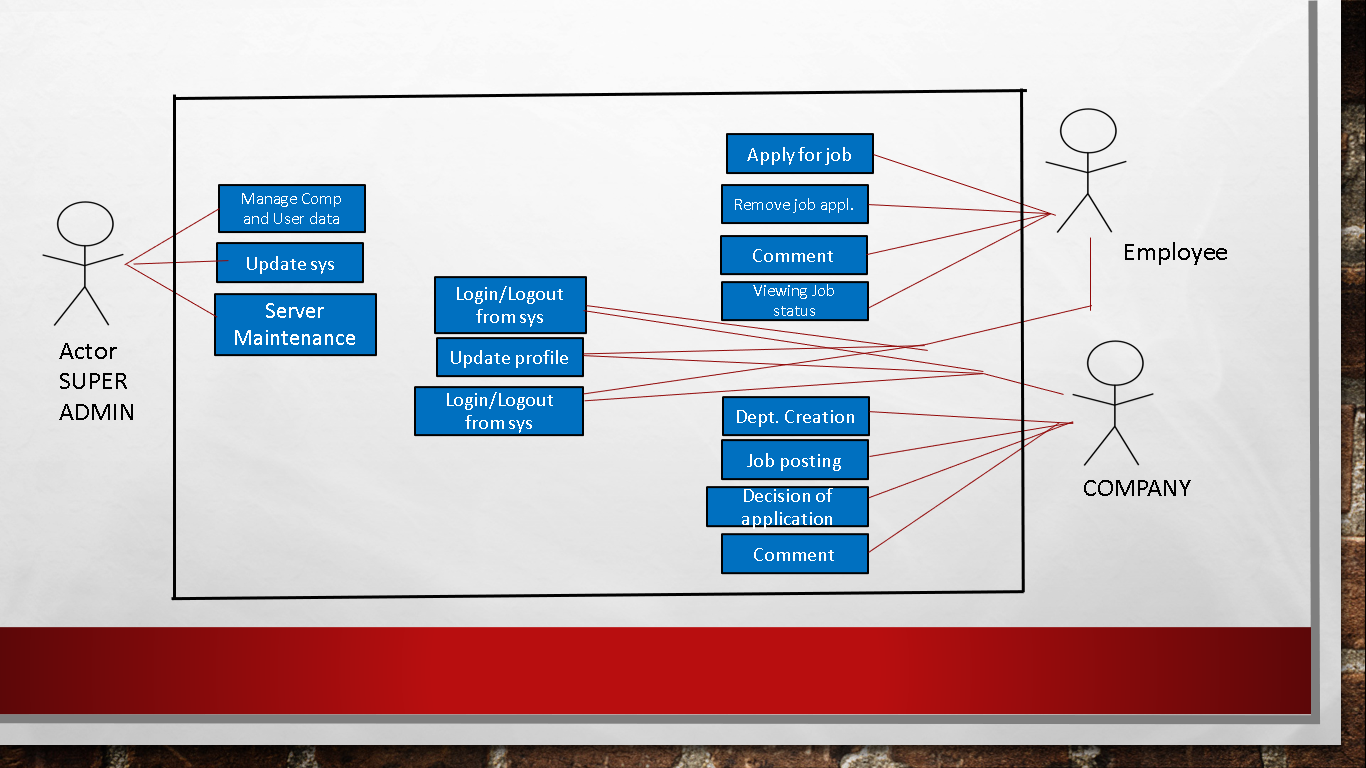
The objective of a DFD is to show the scope and boundaries of a system. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesignin



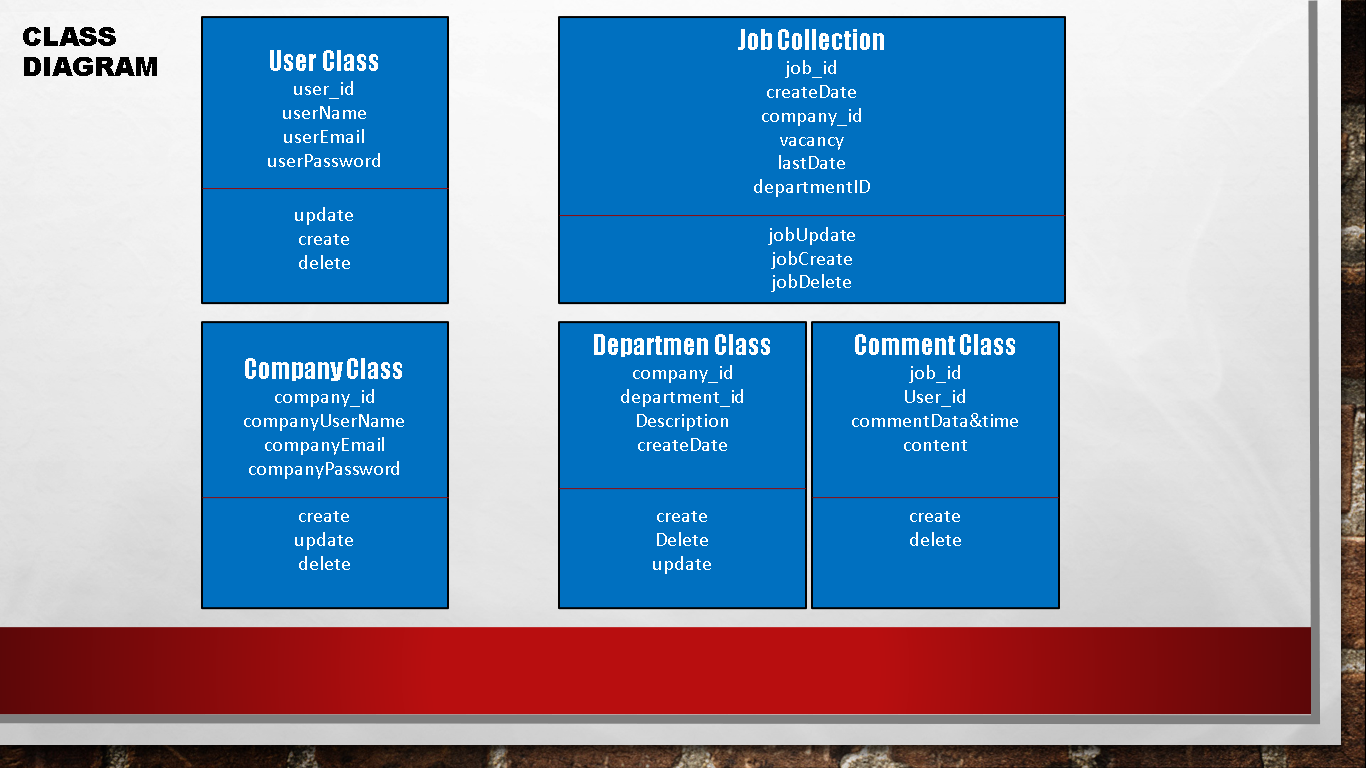
UML DIAGRAMS

A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts, or classes, in order to better understand, alter, maintain, or document information about the system.

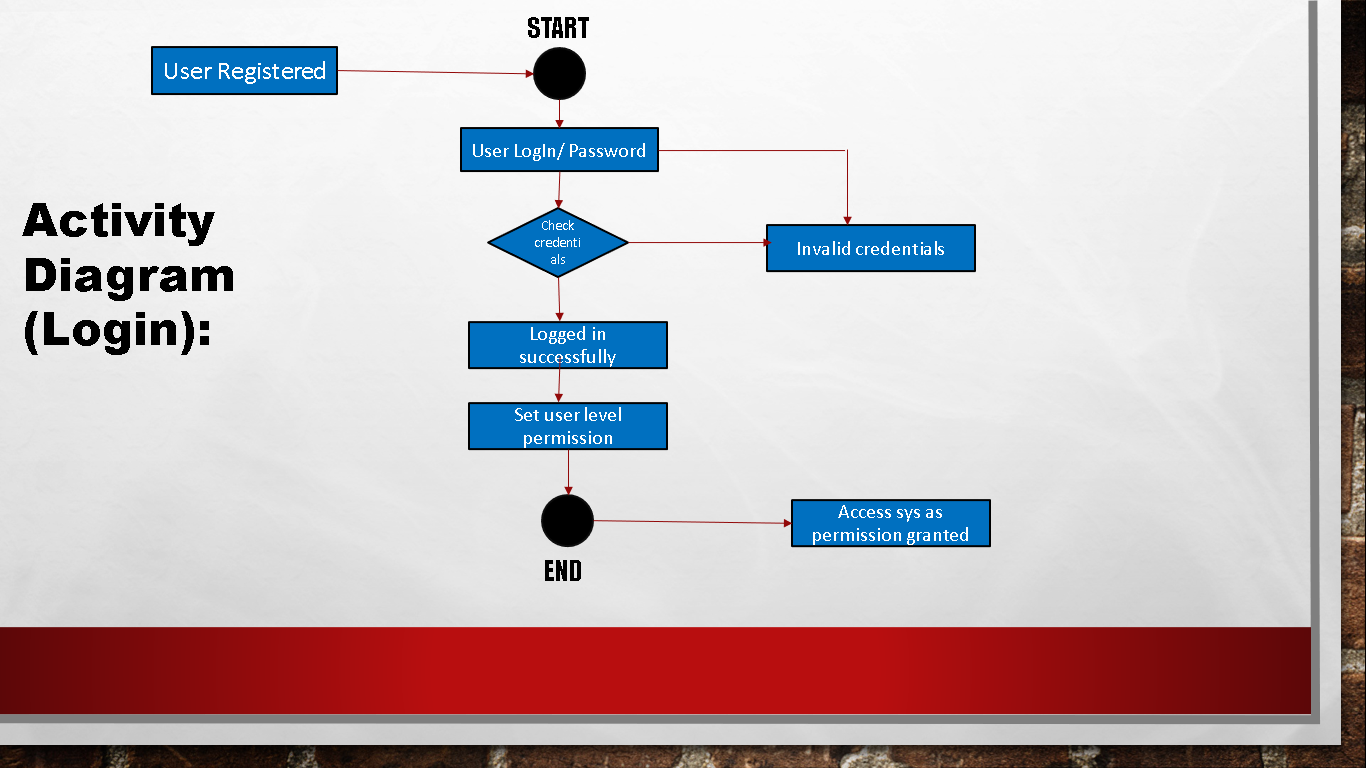
**USE CASE DIAGRAM**:-

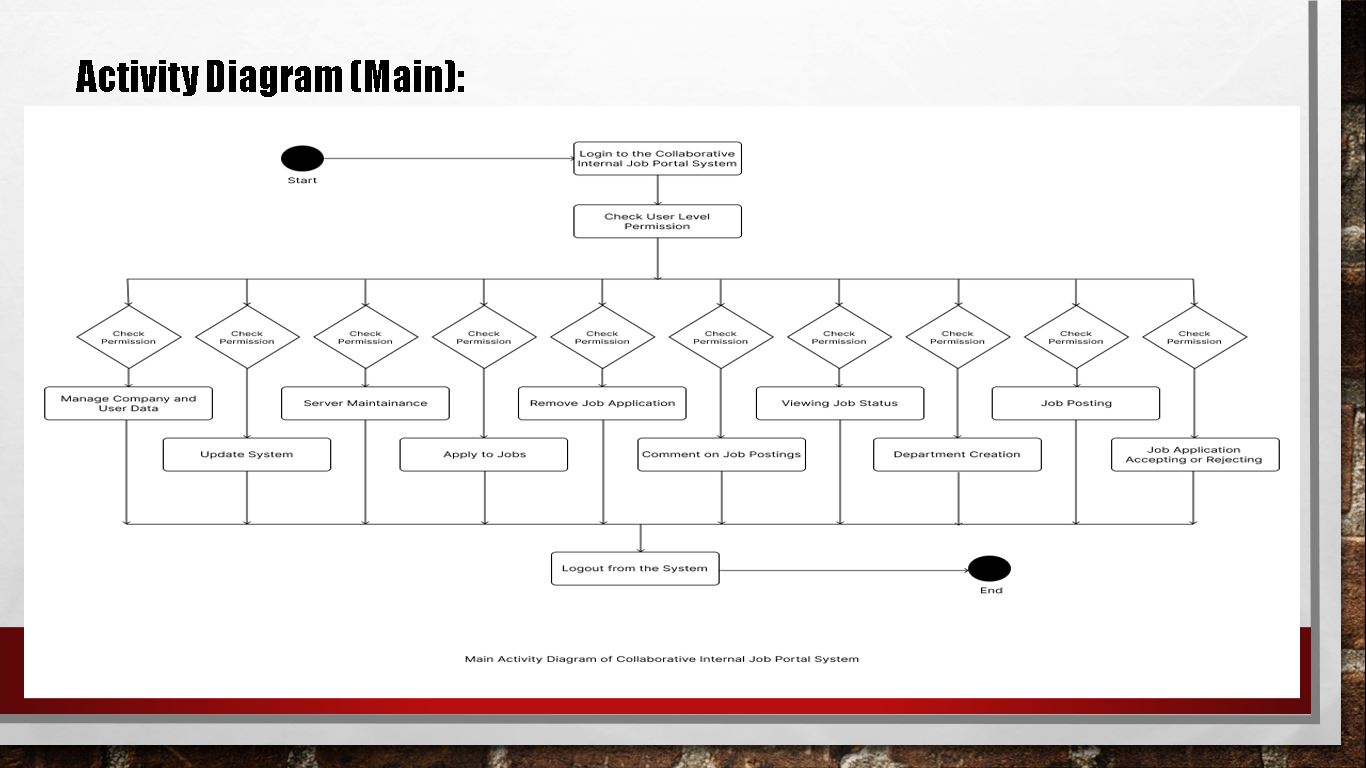


**CLASS DIAGRAM**:-

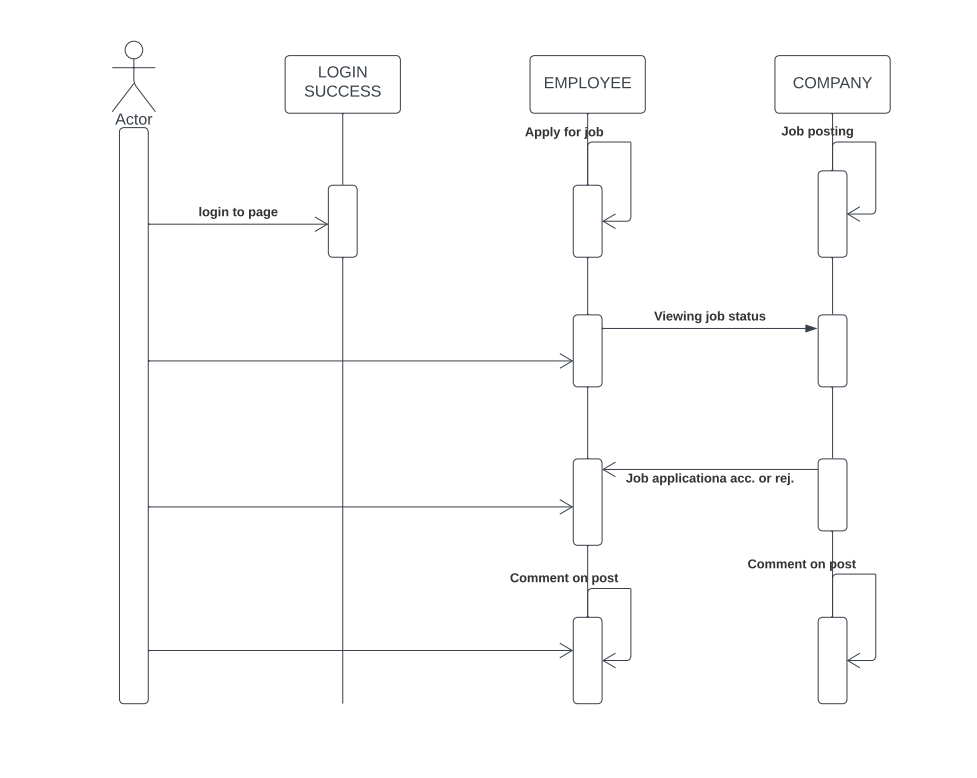


**ACTIVITY DIAGRAM**:-





**SEQUENCE DIAGRAM**:-



TEST REPORT AND RESULTS

The Collaborative Online Job Portal is a web application via which companies can post job applications and employees can apply for that particular job. The resume and profile of the applicants will be visible to the company and they can shortlist the applicants.

The application is built using the MERN stack and thorough testing is done. There are few bugs regarding uploading the resume and profile pictures which will be resolved in the further versions.

As for the initial version, we can create profile as an employee or as a company. Companies can successfully post job applications with full details and employees can apply to such job openings.

MAIN MODULE CODES AND SCREENSHOTS

The source code for backend ->

**server.js file ->**

const express = require("express");

const bodyParser = require("body-parser");

const mongoose = require("mongoose");

const passportConfig = require("./lib/passportConfig");

const cors = require("cors");

const fs = require("fs");

// MongoDB

mongoose

.connect("mongodb://localhost:27017/jobPortal", {

useNewUrlParser: true,

useUnifiedTopology: true,

useCreateIndex: true,

useFindAndModify: false,

})

.then((res) => console.log("Connected to DB"))

.catch((err) => console.log(err));

// initialising directories

if (!fs.existsSync("./public")) {

fs.mkdirSync("./public");

}

if (!fs.existsSync("./public/resume")) {

fs.mkdirSync("./public/resume");

}

if (!fs.existsSync("./public/profile")) {

fs.mkdirSync("./public/profile");

}

const app = express();

const port = 4444;

app.use(bodyParser.json()); // support json encoded bodies

app.use(bodyParser.urlencoded({ extended: true })); // support encoded bodies

// Setting up middlewares

app.use(cors());

app.use(express.json());

app.use(passportConfig.initialize());

// Routing

app.use("/auth", require("./routes/authRoutes"));

app.use("/api", require("./routes/apiRoutes"));

app.use("/upload", require("./routes/uploadRoutes"));

app.use("/host", require("./routes/downloadRoutes"));

app.listen(port, () => {

console.log(`Server started on port ${port}!`);

});

**package.json ->**

{

"name": "job-portal-backend",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "npx nodemon server.js"

},

"author": "",

"license": "ISC",

"dependencies": {

"bcrypt": "^5.0.0",

"body-parser": "^1.19.0",

"connect-flash": "^0.1.1",

"connect-mongo": "^3.2.0",

"cors": "^2.8.5",

"crypto": "^1.0.1",

"express": "^4.17.1",

"express-session": "^1.17.1",

"jsonwebtoken": "^8.5.1",

"mongoose": "^5.11.11",

"mongoose-type-email": "^1.1.2",

"multer": "^2.0.0-rc.2",

"passport": "^0.4.1",

"passport-jwt": "^4.0.0",

"passport-local": "^1.0.0",

"uuid": "^8.3.2"

}

}

**apiRoutes.js ->**

const express = require("express");

const mongoose = require("mongoose");

const jwtAuth = require("../lib/jwtAuth");

const User = require("../db/User");

const JobApplicant = require("../db/JobApplicant");

const Recruiter = require("../db/Recruiter");

const Job = require("../db/Job");

const Application = require("../db/Application");

const Rating = require("../db/Rating");

const router = express.Router();

// To add new job

router.post("/jobs", jwtAuth, (req, res) => {

const user = req.user;

if (user.type != "recruiter") {

res.status(401).json({

message: "You don't have permissions to add jobs",

});

return;

}

const data = req.body;

let job = new Job({

userId: user.\_id,

title: data.title,

maxApplicants: data.maxApplicants,

maxPositions: data.maxPositions,

dateOfPosting: data.dateOfPosting,

deadline: data.deadline,

skillsets: data.skillsets,

jobType: data.jobType,

duration: data.duration,

salary: data.salary,

rating: data.rating,

});

job

.save()

.then(() => {

res.json({ message: "Job added successfully to the database" });

})

.catch((err) => {

res.status(400).json(err);

});

});

// to get all the jobs [pagination] [for recruiter personal and for everyone]

router.get("/jobs", jwtAuth, (req, res) => {

let user = req.user;

let findParams = {};

let sortParams = {};

// const page = parseInt(req.query.page) ? parseInt(req.query.page) : 1;

// const limit = parseInt(req.query.limit) ? parseInt(req.query.limit) : 10;

// const skip = page - 1 >= 0 ? (page - 1) \* limit : 0;

// to list down jobs posted by a particular recruiter

if (user.type === "recruiter" && req.query.myjobs) {

findParams = {

...findParams,

userId: user.\_id,

};

}

if (req.query.q) {

findParams = {

...findParams,

title: {

$regex: new RegExp(req.query.q, "i"),

},

};

}

if (req.query.jobType) {

let jobTypes = [];

if (Array.isArray(req.query.jobType)) {

jobTypes = req.query.jobType;

} else {

jobTypes = [req.query.jobType];

}

console.log(jobTypes);

findParams = {

...findParams,

jobType: {

$in: jobTypes,

},

};

}

if (req.query.salaryMin && req.query.salaryMax) {

findParams = {

...findParams,

$and: [

{

salary: {

$gte: parseInt(req.query.salaryMin),

},

},

{

salary: {

$lte: parseInt(req.query.salaryMax),

},

},

],

};

} else if (req.query.salaryMin) {

findParams = {

...findParams,

salary: {

$gte: parseInt(req.query.salaryMin),

},

};

} else if (req.query.salaryMax) {

findParams = {

...findParams,

salary: {

$lte: parseInt(req.query.salaryMax),

},

};

}

if (req.query.duration) {

findParams = {

...findParams,

duration: {

$lt: parseInt(req.query.duration),

},

};

}

if (req.query.asc) {

if (Array.isArray(req.query.asc)) {

req.query.asc.map((key) => {

sortParams = {

...sortParams,

[key]: 1,

};

});

} else {

sortParams = {

...sortParams,

[req.query.asc]: 1,

};

}

}

if (req.query.desc) {

if (Array.isArray(req.query.desc)) {

req.query.desc.map((key) => {

sortParams = {

...sortParams,

[key]: -1,

};

});

} else {

sortParams = {

...sortParams,

[req.query.desc]: -1,

};

}

}

console.log(findParams);

console.log(sortParams);

// Job.find(findParams).collation({ locale: "en" }).sort(sortParams);

// .skip(skip)

// .limit(limit)

let arr = [

{

$lookup: {

from: "recruiterinfos",

localField: "userId",

foreignField: "userId",

as: "recruiter",

},

},

{ $unwind: "$recruiter" },

{ $match: findParams },

];

if (Object.keys(sortParams).length > 0) {

arr = [

{

$lookup: {

from: "recruiterinfos",

localField: "userId",

foreignField: "userId",

as: "recruiter",

},

},

{ $unwind: "$recruiter" },

{ $match: findParams },

{

$sort: sortParams,

},

];

}

console.log(arr);

Job.aggregate(arr)

.then((posts) => {

if (posts == null) {

res.status(404).json({

message: "No job found",

});

return;

}

res.json(posts);

})

.catch((err) => {

res.status(400).json(err);

});

});

// to get info about a particular job

router.get("/jobs/:id", jwtAuth, (req, res) => {

Job.findOne({ \_id: req.params.id })

.then((job) => {

if (job == null) {

res.status(400).json({

message: "Job does not exist",

});

return;

}

res.json(job);

})

.catch((err) => {

res.status(400).json(err);

});

});

// to update info of a particular job

router.put("/jobs/:id", jwtAuth, (req, res) => {

const user = req.user;

if (user.type != "recruiter") {

res.status(401).json({

message: "You don't have permissions to change the job details",

});

return;

}

Job.findOne({

\_id: req.params.id,

userId: user.id,

})

.then((job) => {

if (job == null) {

res.status(404).json({

message: "Job does not exist",

});

return;

}

const data = req.body;

if (data.maxApplicants) {

job.maxApplicants = data.maxApplicants;

}

if (data.maxPositions) {

job.maxPositions = data.maxPositions;

}

if (data.deadline) {

job.deadline = data.deadline;

}

job

.save()

.then(() => {

res.json({

message: "Job details updated successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

});

// to delete a job

router.delete("/jobs/:id", jwtAuth, (req, res) => {

const user = req.user;

if (user.type != "recruiter") {

res.status(401).json({

message: "You don't have permissions to delete the job",

});

return;

}

Job.findOneAndDelete({

\_id: req.params.id,

userId: user.id,

})

.then((job) => {

if (job === null) {

res.status(401).json({

message: "You don't have permissions to delete the job",

});

return;

}

res.json({

message: "Job deleted successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

});

// get user's personal details

router.get("/user", jwtAuth, (req, res) => {

const user = req.user;

if (user.type === "recruiter") {

Recruiter.findOne({ userId: user.\_id })

.then((recruiter) => {

if (recruiter == null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

res.json(recruiter);

})

.catch((err) => {

res.status(400).json(err);

});

} else {

JobApplicant.findOne({ userId: user.\_id })

.then((jobApplicant) => {

if (jobApplicant == null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

res.json(jobApplicant);

})

.catch((err) => {

res.status(400).json(err);

});

}

});

// get user details from id

router.get("/user/:id", jwtAuth, (req, res) => {

User.findOne({ \_id: req.params.id })

.then((userData) => {

if (userData === null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

if (userData.type === "recruiter") {

Recruiter.findOne({ userId: userData.\_id })

.then((recruiter) => {

if (recruiter === null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

res.json(recruiter);

})

.catch((err) => {

res.status(400).json(err);

});

} else {

JobApplicant.findOne({ userId: userData.\_id })

.then((jobApplicant) => {

if (jobApplicant === null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

res.json(jobApplicant);

})

.catch((err) => {

res.status(400).json(err);

});

}

})

.catch((err) => {

res.status(400).json(err);

});

});

// update user details

router.put("/user", jwtAuth, (req, res) => {

const user = req.user;

const data = req.body;

if (user.type == "recruiter") {

Recruiter.findOne({ userId: user.\_id })

.then((recruiter) => {

if (recruiter == null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

if (data.name) {

recruiter.name = data.name;

}

if (data.contactNumber) {

recruiter.contactNumber = data.contactNumber;

}

if (data.bio) {

recruiter.bio = data.bio;

}

recruiter

.save()

.then(() => {

res.json({

message: "User information updated successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

JobApplicant.findOne({ userId: user.\_id })

.then((jobApplicant) => {

if (jobApplicant == null) {

res.status(404).json({

message: "User does not exist",

});

return;

}

if (data.name) {

jobApplicant.name = data.name;

}

if (data.education) {

jobApplicant.education = data.education;

}

if (data.skills) {

jobApplicant.skills = data.skills;

}

if (data.resume) {

jobApplicant.resume = data.resume;

}

if (data.profile) {

jobApplicant.profile = data.profile;

}

console.log(jobApplicant);

jobApplicant

.save()

.then(() => {

res.json({

message: "User information updated successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

}

});

// apply for a job [todo: test: done]

router.post("/jobs/:id/applications", jwtAuth, (req, res) => {

const user = req.user;

if (user.type != "applicant") {

res.status(401).json({

message: "You don't have permissions to apply for a job",

});

return;

}

const data = req.body;

const jobId = req.params.id;

// check whether applied previously

// find job

// check count of active applications < limit

// check user had < 10 active applications && check if user is not having any accepted jobs (user id)

// store the data in applications

Application.findOne({

userId: user.\_id,

jobId: jobId,

status: {

$nin: ["deleted", "accepted", "cancelled"],

},

})

.then((appliedApplication) => {

console.log(appliedApplication);

if (appliedApplication !== null) {

res.status(400).json({

message: "You have already applied for this job",

});

return;

}

Job.findOne({ \_id: jobId })

.then((job) => {

if (job === null) {

res.status(404).json({

message: "Job does not exist",

});

return;

}

Application.countDocuments({

jobId: jobId,

status: {

$nin: ["rejected", "deleted", "cancelled", "finished"],

},

})

.then((activeApplicationCount) => {

if (activeApplicationCount < job.maxApplicants) {

Application.countDocuments({

userId: user.\_id,

status: {

$nin: ["rejected", "deleted", "cancelled", "finished"],

},

})

.then((myActiveApplicationCount) => {

if (myActiveApplicationCount < 10) {

Application.countDocuments({

userId: user.\_id,

status: "accepted",

}).then((acceptedJobs) => {

if (acceptedJobs === 0) {

const application = new Application({

userId: user.\_id,

recruiterId: job.userId,

jobId: job.\_id,

status: "applied",

sop: data.sop,

});

application

.save()

.then(() => {

res.json({

message: "Job application successful",

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.status(400).json({

message:

"You already have an accepted job. Hence you cannot apply.",

});

}

});

} else {

res.status(400).json({

message:

"You have 10 active applications. Hence you cannot apply.",

});

}

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.status(400).json({

message: "Application limit reached",

});

}

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.json(400).json(err);

});

});

// recruiter gets applications for a particular job [pagination] [todo: test: done]

router.get("/jobs/:id/applications", jwtAuth, (req, res) => {

const user = req.user;

if (user.type != "recruiter") {

res.status(401).json({

message: "You don't have permissions to view job applications",

});

return;

}

const jobId = req.params.id;

// const page = parseInt(req.query.page) ? parseInt(req.query.page) : 1;

// const limit = parseInt(req.query.limit) ? parseInt(req.query.limit) : 10;

// const skip = page - 1 >= 0 ? (page - 1) \* limit : 0;

let findParams = {

jobId: jobId,

recruiterId: user.\_id,

};

let sortParams = {};

if (req.query.status) {

findParams = {

...findParams,

status: req.query.status,

};

}

Application.find(findParams)

.collation({ locale: "en" })

.sort(sortParams)

// .skip(skip)

// .limit(limit)

.then((applications) => {

res.json(applications);

})

.catch((err) => {

res.status(400).json(err);

});

});

// recruiter/applicant gets all his applications [pagination]

router.get("/applications", jwtAuth, (req, res) => {

const user = req.user;

// const page = parseInt(req.query.page) ? parseInt(req.query.page) : 1;

// const limit = parseInt(req.query.limit) ? parseInt(req.query.limit) : 10;

// const skip = page - 1 >= 0 ? (page - 1) \* limit : 0;

Application.aggregate([

{

$lookup: {

from: "jobapplicantinfos",

localField: "userId",

foreignField: "userId",

as: "jobApplicant",

},

},

{ $unwind: "$jobApplicant" },

{

$lookup: {

from: "jobs",

localField: "jobId",

foreignField: "\_id",

as: "job",

},

},

{ $unwind: "$job" },

{

$lookup: {

from: "recruiterinfos",

localField: "recruiterId",

foreignField: "userId",

as: "recruiter",

},

},

{ $unwind: "$recruiter" },

{

$match: {

[user.type === "recruiter" ? "recruiterId" : "userId"]: user.\_id,

},

},

{

$sort: {

dateOfApplication: -1,

},

},

])

.then((applications) => {

res.json(applications);

})

.catch((err) => {

res.status(400).json(err);

});

});

// update status of application: [Applicant: Can cancel, Recruiter: Can do everything] [todo: test: done]

router.put("/applications/:id", jwtAuth, (req, res) => {

const user = req.user;

const id = req.params.id;

const status = req.body.status;

// "applied", // when a applicant is applied

// "shortlisted", // when a applicant is shortlisted

// "accepted", // when a applicant is accepted

// "rejected", // when a applicant is rejected

// "deleted", // when any job is deleted

// "cancelled", // an application is cancelled by its author or when other application is accepted

// "finished", // when job is over

if (user.type === "recruiter") {

if (status === "accepted") {

// get job id from application

// get job info for maxPositions count

// count applications that are already accepted

// compare and if condition is satisfied, then save

Application.findOne({

\_id: id,

recruiterId: user.\_id,

})

.then((application) => {

if (application === null) {

res.status(404).json({

message: "Application not found",

});

return;

}

Job.findOne({

\_id: application.jobId,

userId: user.\_id,

}).then((job) => {

if (job === null) {

res.status(404).json({

message: "Job does not exist",

});

return;

}

Application.countDocuments({

recruiterId: user.\_id,

jobId: job.\_id,

status: "accepted",

}).then((activeApplicationCount) => {

if (activeApplicationCount < job.maxPositions) {

// accepted

application.status = status;

application.dateOfJoining = req.body.dateOfJoining;

application

.save()

.then(() => {

Application.updateMany(

{

\_id: {

$ne: application.\_id,

},

userId: application.userId,

status: {

$nin: [

"rejected",

"deleted",

"cancelled",

"accepted",

"finished",

],

},

},

{

$set: {

status: "cancelled",

},

},

{ multi: true }

)

.then(() => {

if (status === "accepted") {

Job.findOneAndUpdate(

{

\_id: job.\_id,

userId: user.\_id,

},

{

$set: {

acceptedCandidates: activeApplicationCount + 1,

},

}

)

.then(() => {

res.json({

message: `Application ${status} successfully`,

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.json({

message: `Application ${status} successfully`,

});

}

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.status(400).json({

message: "All positions for this job are already filled",

});

}

});

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

Application.findOneAndUpdate(

{

\_id: id,

recruiterId: user.\_id,

status: {

$nin: ["rejected", "deleted", "cancelled"],

},

},

{

$set: {

status: status,

},

}

)

.then((application) => {

if (application === null) {

res.status(400).json({

message: "Application status cannot be updated",

});

return;

}

if (status === "finished") {

res.json({

message: `Job ${status} successfully`,

});

} else {

res.json({

message: `Application ${status} successfully`,

});

}

})

.catch((err) => {

res.status(400).json(err);

});

}

} else {

if (status === "cancelled") {

console.log(id);

console.log(user.\_id);

Application.findOneAndUpdate(

{

\_id: id,

userId: user.\_id,

},

{

$set: {

status: status,

},

}

)

.then((tmp) => {

console.log(tmp);

res.json({

message: `Application ${status} successfully`,

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.status(401).json({

message: "You don't have permissions to update job status",

});

}

}

});

// get a list of final applicants for current job : recruiter

// get a list of final applicants for all his jobs : recuiter

router.get("/applicants", jwtAuth, (req, res) => {

const user = req.user;

if (user.type === "recruiter") {

let findParams = {

recruiterId: user.\_id,

};

if (req.query.jobId) {

findParams = {

...findParams,

jobId: new mongoose.Types.ObjectId(req.query.jobId),

};

}

if (req.query.status) {

if (Array.isArray(req.query.status)) {

findParams = {

...findParams,

status: { $in: req.query.status },

};

} else {

findParams = {

...findParams,

status: req.query.status,

};

}

}

let sortParams = {};

if (!req.query.asc && !req.query.desc) {

sortParams = { \_id: 1 };

}

if (req.query.asc) {

if (Array.isArray(req.query.asc)) {

req.query.asc.map((key) => {

sortParams = {

...sortParams,

[key]: 1,

};

});

} else {

sortParams = {

...sortParams,

[req.query.asc]: 1,

};

}

}

if (req.query.desc) {

if (Array.isArray(req.query.desc)) {

req.query.desc.map((key) => {

sortParams = {

...sortParams,

[key]: -1,

};

});

} else {

sortParams = {

...sortParams,

[req.query.desc]: -1,

};

}

}

Application.aggregate([

{

$lookup: {

from: "jobapplicantinfos",

localField: "userId",

foreignField: "userId",

as: "jobApplicant",

},

},

{ $unwind: "$jobApplicant" },

{

$lookup: {

from: "jobs",

localField: "jobId",

foreignField: "\_id",

as: "job",

},

},

{ $unwind: "$job" },

{ $match: findParams },

{ $sort: sortParams },

])

.then((applications) => {

if (applications.length === 0) {

res.status(404).json({

message: "No applicants found",

});

return;

}

res.json(applications);

})

.catch((err) => {

res.status(400).json(err);

});

} else {

res.status(400).json({

message: "You are not allowed to access applicants list",

});

}

});

// to add or update a rating [todo: test]

router.put("/rating", jwtAuth, (req, res) => {

const user = req.user;

const data = req.body;

if (user.type === "recruiter") {

// can rate applicant

Rating.findOne({

senderId: user.\_id,

receiverId: data.applicantId,

category: "applicant",

})

.then((rating) => {

if (rating === null) {

console.log("new rating");

Application.countDocuments({

userId: data.applicantId,

recruiterId: user.\_id,

status: {

$in: ["accepted", "finished"],

},

})

.then((acceptedApplicant) => {

if (acceptedApplicant > 0) {

// add a new rating

rating = new Rating({

category: "applicant",

receiverId: data.applicantId,

senderId: user.\_id,

rating: data.rating,

});

rating

.save()

.then(() => {

// get the average of ratings

Rating.aggregate([

{

$match: {

receiverId: mongoose.Types.ObjectId(data.applicantId),

category: "applicant",

},

},

{

$group: {

\_id: {},

average: { $avg: "$rating" },

},

},

])

.then((result) => {

// update the user's rating

if (result === null) {

res.status(400).json({

message: "Error while calculating rating",

});

return;

}

const avg = result[0].average;

JobApplicant.findOneAndUpdate(

{

userId: data.applicantId,

},

{

$set: {

rating: avg,

},

}

)

.then((applicant) => {

if (applicant === null) {

res.status(400).json({

message:

"Error while updating applicant's average rating",

});

return;

}

res.json({

message: "Rating added successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

// you cannot rate

res.status(400).json({

message:

"Applicant didn't worked under you. Hence you cannot give a rating.",

});

}

})

.catch((err) => {

res.status(400).json(err);

});

} else {

rating.rating = data.rating;

rating

.save()

.then(() => {

// get the average of ratings

Rating.aggregate([

{

$match: {

receiverId: mongoose.Types.ObjectId(data.applicantId),

category: "applicant",

},

},

{

$group: {

\_id: {},

average: { $avg: "$rating" },

},

},

])

.then((result) => {

// update the user's rating

if (result === null) {

res.status(400).json({

message: "Error while calculating rating",

});

return;

}

const avg = result[0].average;

JobApplicant.findOneAndUpdate(

{

userId: data.applicantId,

},

{

$set: {

rating: avg,

},

}

)

.then((applicant) => {

if (applicant === null) {

res.status(400).json({

message:

"Error while updating applicant's average rating",

});

return;

}

res.json({

message: "Rating updated successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

}

})

.catch((err) => {

res.status(400).json(err);

});

} else {

// applicant can rate job

Rating.findOne({

senderId: user.\_id,

receiverId: data.jobId,

category: "job",

})

.then((rating) => {

console.log(user.\_id);

console.log(data.jobId);

console.log(rating);

if (rating === null) {

console.log(rating);

Application.countDocuments({

userId: user.\_id,

jobId: data.jobId,

status: {

$in: ["accepted", "finished"],

},

})

.then((acceptedApplicant) => {

if (acceptedApplicant > 0) {

// add a new rating

rating = new Rating({

category: "job",

receiverId: data.jobId,

senderId: user.\_id,

rating: data.rating,

});

rating

.save()

.then(() => {

// get the average of ratings

Rating.aggregate([

{

$match: {

receiverId: mongoose.Types.ObjectId(data.jobId),

category: "job",

},

},

{

$group: {

\_id: {},

average: { $avg: "$rating" },

},

},

])

.then((result) => {

if (result === null) {

res.status(400).json({

message: "Error while calculating rating",

});

return;

}

const avg = result[0].average;

Job.findOneAndUpdate(

{

\_id: data.jobId,

},

{

$set: {

rating: avg,

},

}

)

.then((foundJob) => {

if (foundJob === null) {

res.status(400).json({

message:

"Error while updating job's average rating",

});

return;

}

res.json({

message: "Rating added successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

} else {

// you cannot rate

res.status(400).json({

message:

"You haven't worked for this job. Hence you cannot give a rating.",

});

}

})

.catch((err) => {

res.status(400).json(err);

});

} else {

// update the rating

rating.rating = data.rating;

rating

.save()

.then(() => {

// get the average of ratings

Rating.aggregate([

{

$match: {

receiverId: mongoose.Types.ObjectId(data.jobId),

category: "job",

},

},

{

$group: {

\_id: {},

average: { $avg: "$rating" },

},

},

])

.then((result) => {

if (result === null) {

res.status(400).json({

message: "Error while calculating rating",

});

return;

}

const avg = result[0].average;

console.log(avg);

Job.findOneAndUpdate(

{

\_id: data.jobId,

},

{

$set: {

rating: avg,

},

}

)

.then((foundJob) => {

if (foundJob === null) {

res.status(400).json({

message: "Error while updating job's average rating",

});

return;

}

res.json({

message: "Rating added successfully",

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

})

.catch((err) => {

res.status(400).json(err);

});

}

})

.catch((err) => {

res.status(400).json(err);

});

}

});

// get personal rating

router.get("/rating", jwtAuth, (req, res) => {

const user = req.user;

Rating.findOne({

senderId: user.\_id,

receiverId: req.query.id,

category: user.type === "recruiter" ? "applicant" : "job",

}).then((rating) => {

if (rating === null) {

res.json({

rating: -1,

});

return;

}

res.json({

rating: rating.rating,

});

});

});

// Application.findOne({

// \_id: id,

// userId: user.\_id,

// })

// .then((application) => {

// application.status = status;

// application

// .save()

// .then(() => {

// res.json({

// message: `Application ${status} successfully`,

// });

// })

// .catch((err) => {

// res.status(400).json(err);

// });

// })

// .catch((err) => {

// res.status(400).json(err);

// });

// router.get("/jobs", (req, res, next) => {

// passport.authenticate("jwt", { session: false }, function (err, user, info) {

// if (err) {

// return next(err);

// }

// if (!user) {

// res.status(401).json(info);

// return;

// }

// })(req, res, next);

// });

module.exports = router;

**Application.js ->**

const mongoose = require("mongoose");

let schema = new mongoose.Schema(

{

userId: {

type: mongoose.Schema.Types.ObjectId,

required: true,

},

recruiterId: {

type: mongoose.Schema.Types.ObjectId,

required: true,

},

jobId: {

type: mongoose.Schema.Types.ObjectId,

required: true,

},

status: {

type: String,

enum: [

"applied", // when a applicant is applied

"shortlisted", // when a applicant is shortlisted

"accepted", // when a applicant is accepted

"rejected", // when a applicant is rejected

"deleted", // when any job is deleted

"cancelled", // an application is cancelled by its author or when other application is accepted

"finished", // when job is over

],

default: "applied",

required: true,

},

dateOfApplication: {

type: Date,

default: Date.now,

},

dateOfJoining: {

type: Date,

validate: [

{

validator: function (value) {

return this.dateOfApplication <= value;

},

msg: "dateOfJoining should be greater than dateOfApplication",

},

],

},

sop: {

type: String,

validate: {

validator: function (v) {

return v.split(" ").filter((ele) => ele != "").length <= 250;

},

msg: "Statement of purpose should not be greater than 250 words",

},

},

},

{ collation: { locale: "en" } }

);

// schema.virtual("applicationUser", {

// ref: "JobApplicantInfo",

// localField: "userId",

// foreignField: "userId",

// justOne: true,

// });

// schema.virtual("applicationRecruiter", {

// ref: "RecruiterInfo",

// localField: "recruiterId",

// foreignField: "userId",

// justOne: true,

// });

// schema.virtual("applicationJob", {

// ref: "jobs",

// localField: "jobId",

// foreignField: "\_id",

// justOne: true,

// });

module.exports = mongoose.model("applications", schema);

**Jobs.js ->**

const mongoose = require("mongoose");

let schema = new mongoose.Schema(

{

userId: {

type: mongoose.Schema.Types.ObjectId,

required: true,

},

title: {

type: String,

required: true,

},

maxApplicants: {

type: Number,

validate: [

{

validator: Number.isInteger,

msg: "maxApplicants should be an integer",

},

{

validator: function (value) {

return value > 0;

},

msg: "maxApplicants should greater than 0",

},

],

},

maxPositions: {

type: Number,

validate: [

{

validator: Number.isInteger,

msg: "maxPostions should be an integer",

},

{

validator: function (value) {

return value > 0;

},

msg: "maxPositions should greater than 0",

},

],

},

activeApplications: {

type: Number,

default: 0,

validate: [

{

validator: Number.isInteger,

msg: "activeApplications should be an integer",

},

{

validator: function (value) {

return value >= 0;

},

msg: "activeApplications should greater than equal to 0",

},

],

},

acceptedCandidates: {

type: Number,

default: 0,

validate: [

{

validator: Number.isInteger,

msg: "acceptedCandidates should be an integer",

},

{

validator: function (value) {

return value >= 0;

},

msg: "acceptedCandidates should greater than equal to 0",

},

],

},

dateOfPosting: {

type: Date,

default: Date.now,

},

deadline: {

type: Date,

validate: [

{

validator: function (value) {

return this.dateOfPosting < value;

},

msg: "deadline should be greater than dateOfPosting",

},

],

},

skillsets: [String],

jobType: {

type: String,

required: true,

},

duration: {

type: Number,

min: 0,

validate: [

{

validator: Number.isInteger,

msg: "Duration should be an integer",

},

],

},

salary: {

type: Number,

validate: [

{

validator: Number.isInteger,

msg: "Salary should be an integer",

},

{

validator: function (value) {

return value >= 0;

},

msg: "Salary should be positive",

},

],

},

rating: {

type: Number,

max: 5.0,

default: -1.0,

validate: {

validator: function (v) {

return v >= -1.0 && v <= 5.0;

},

msg: "Invalid rating",

},

},

},

{ collation: { locale: "en" } }

);

module.exports = mongoose.model("jobs", schema);

The source code for frontend ->

BUG REPORT

For the initial version of the web application, as for bugs there are problems with the profile picture uploading and resume uploading which will be resolved in the next versions of the applications.

FURTHER DEVELOPMENT

As for further development, we plan on ->

* Improving the UI of the application.
* Resolve existing bugs.
* Organizing the User Data with addition of more collections.

CONCLUSION

Collaborative Online Job portal is a MERN stack Web Application that showcases the CRUD (Create Read Update Delete) functionality. Every team member has put in their best effort to make the first version of the application up and running. Special gratitude to our respected project guide for pinpointing the short comings in our project.

**THANK YOU**