Indian Institute of Information Technology Surat

Lab Report on

Object Oriented Technology(CS 404)

Submitted by

Anjali(UI23CS07)

Course faculty

Dr. Reema Patel

Mr. Rishi Sharma

Department of Computer Science and Engineering

Indian Institute of Information Technology Surat

Gujarat-394190, India

Month-Year

Index

Project Link:- https://github.com/Anjali140600/OOT-Final-Project

Problem Statement: Job portal web application

(Mern stack)

Objective:

To build a responsive and dynamic job portal web application that allows:

- **Job seekers** to search, apply, and manage job applications.
- **Recruiters** to post, update, and manage job listings.
- **Admins** to monitor platform activities and manage users and companies.

Key Features:

For Job Seekers:

- User registration & authentication
- Profile creation and resume upload
- Job search with filters (location, role, salary, etc.)
- Apply to jobs and track application status
- Save jobs for later

For Recruiters:

- Company profile setup
- Post new jobs with detailed descriptions
- Update or delete existing job postings
- View applications received per job
- Shortlist or reject candidates

For Admin:

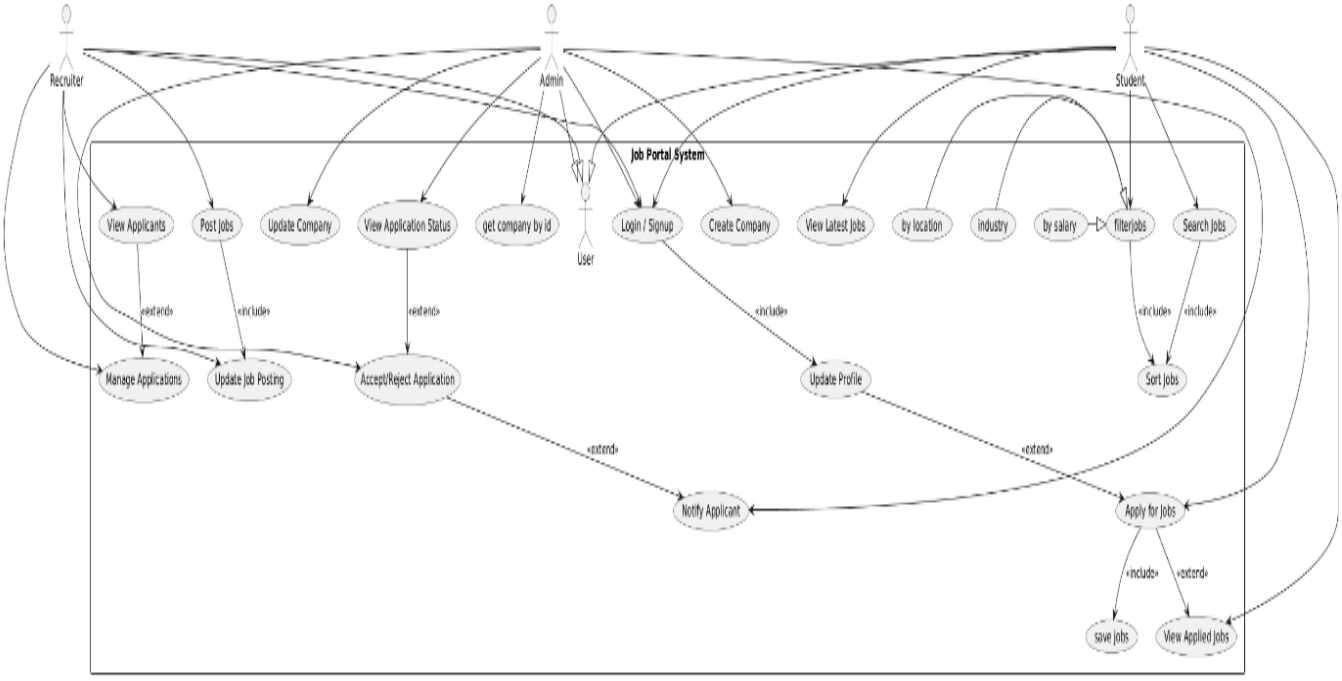
- Manage user roles (students, recruiters)
- View platform statistics (e.g., number of jobs, users)
- Moderate job posts and reported users
- Add/update/remove company details

Technology Stack:

- **Frontend**: React.js, Tailwind CSS
- **Backend**: Node.js, Express.js
- **Database**: MongoDB
- **Authentication**: JWT + bcrypt
- **Others**: Cloudinary (for file uploads), Redux (optional)

Diagrams

*Use Case Diagram



Explanation:

Actors:

- Student: Seeks jobs and applies.
- Recruiter: Posts jobs and manages applications.
- Admin: Manages companies and application statuses.

Key Use Cases:

1. Login / Signup: Common to all actors.
2. Update Profile: Can be updated by all users.

3. Search Jobs: Students search for jobs, can filter and sort results.
4. Apply for Jobs: Students apply for jobs, can save them for future reference.
5. View Applied Jobs: Students view jobs they've applied to.
6. Post Jobs: Recruiters post job listings.
7. Update Job Posting: Recruiters update posted jobs.
8. View Applicants: Recruiters view applicants for their posted jobs.
9. Manage Applications: Recruiters manage (accept/reject) applicants.
10. Create/Update Company: Admin can create and update companies.
11. View Application Status: Admin checks application statuses.

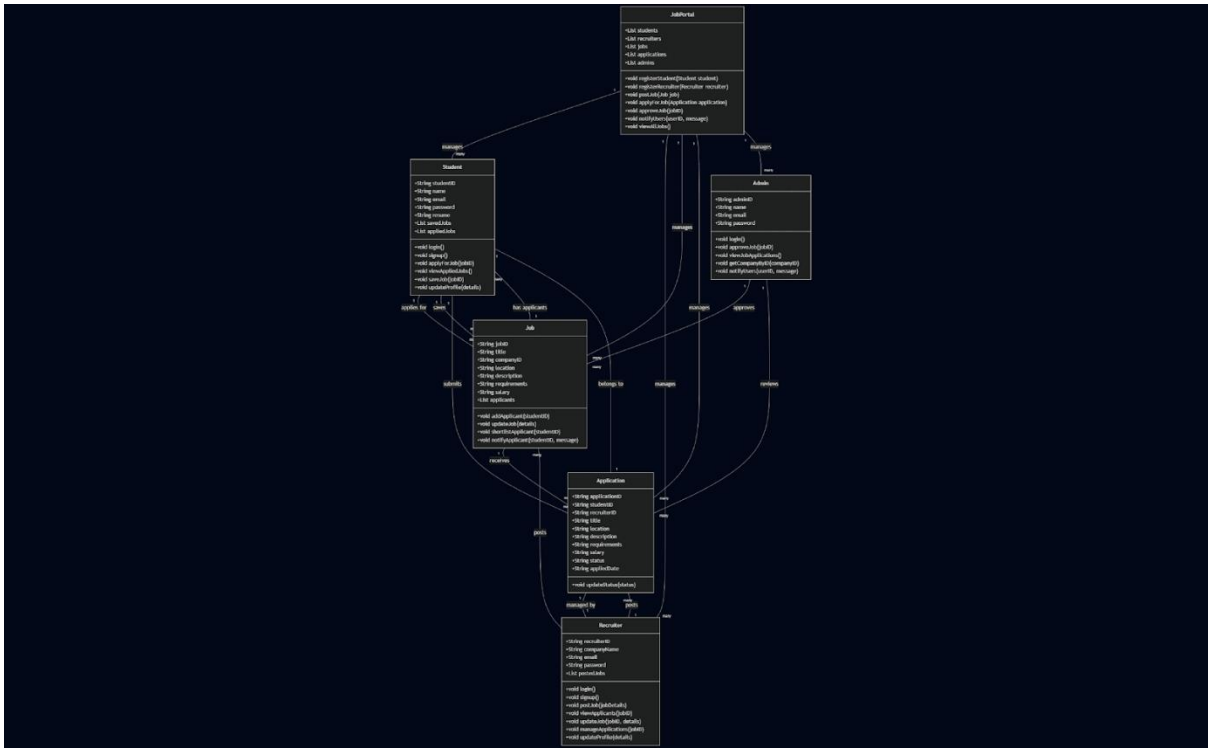
Relationships:

- Generalization:
    - Student, Recruiter, and Admin inherit from a common User.
- Includes:
    - Sort Jobs is part of Search Jobs and Filter Jobs.
    - Update Profile is included in the Login/Signup process.
- Extends:
    - Update Profile extends Apply for Jobs.
    - View Applied Jobs extends Apply for Jobs.
    - Manage Applications extends View Applicants.
    - Notify Applicant extends Accept/Reject Application.
    - Accept/Reject Application extends View Application Status.

Key Extensions:

- Filters (like By Salary, By Location, etc.) extend the Filter Jobs use case.
- Notify Applicant extends after an application is accepted or rejected.

*Class Diagram



Explanation:

1. Student Class

- Attributes:
  - studentID, name, email, password: Basic user identity fields
  - resume: A reference or link to the student’s resume
  - savedJobs, appliedJobs: Lists to store job IDs that are saved/applied
- Methods:
  - login(), signup(): For authentication
  - applyForJob(jobID): Applies for a specific job
  - viewAppliedJobs(): Views the list of jobs they've applied to
  - saveJob(jobID): Saves job for later
  - updateProfile(details): Update personal info/resume

Relationships:

- Applies for many jobs
- Saves many jobs
- Submits many applications
- Each application belongs to one student

2. Recruiter Class

◇ Attributes:

- recruiterID, companyName, email, password
- postedJobs: List of job IDs the recruiter has posted

◇ Methods:

- login(), signup(): Auth functionality
- postJob(jobDetails): Posts a new job
- viewApplicants(jobID): Views list of applicants for a job
- updateJob(jobID, details): Edits an existing job
- manageApplications(jobID): Manages all applications for a job
- updateProfile(details): Edits recruiter profile

Relationships:

- Posts many jobs
- Manages many applications
- Each application is managed by one recruiter

3. Admin Class

◇ Attributes:

- adminID, name, email, password

◇ Methods:

- login(): Admin login
- approveJob(jobID): Approves job before posting

- viewJobApplications(): Views all job applications
- getCompanyByID(companyID): Fetch company info
- notifyUsers(userID, message): Send notifications to users

Relationships:

- Approves many jobs
- Reviews many applications

4. Job Class

◇ Attributes:

- jobID, title, companyID, location, description, requirements, salary
- applicants: List of student IDs who applied

◇ Methods:

- addApplicant(studentID): Adds student to applicant list
- updateJob(details): Updates job info
- shortlistApplicant(studentID): Shortlist applicant
- notifyApplicant(studentID, message): Notify about status

Relationships:

- Has many applicants (Students)
- Receives many applications
- Is posted by one recruiter
- Is approved by admin

5. Application Class

◇ Attributes:

- applicationID, studentID, recruiterID
- Job-related details: title, location, description, requirements, salary
- status: e.g., "Pending", "Accepted", "Rejected"

- appliedDate: Timestamp

◇ Methods:

- updateStatus(status): Update application status

Relationships:

- Each application is linked to one student and one recruiter
- Belongs to a single job (indirectly)
- Admin can review applications

6.Job portal class

◇ Attributes:

* students: A list (or map) of all registered **Student** objects.

* recruiters: A list of all **Recruiter** objects.

* jobs: A list of all posted **Job** objects.

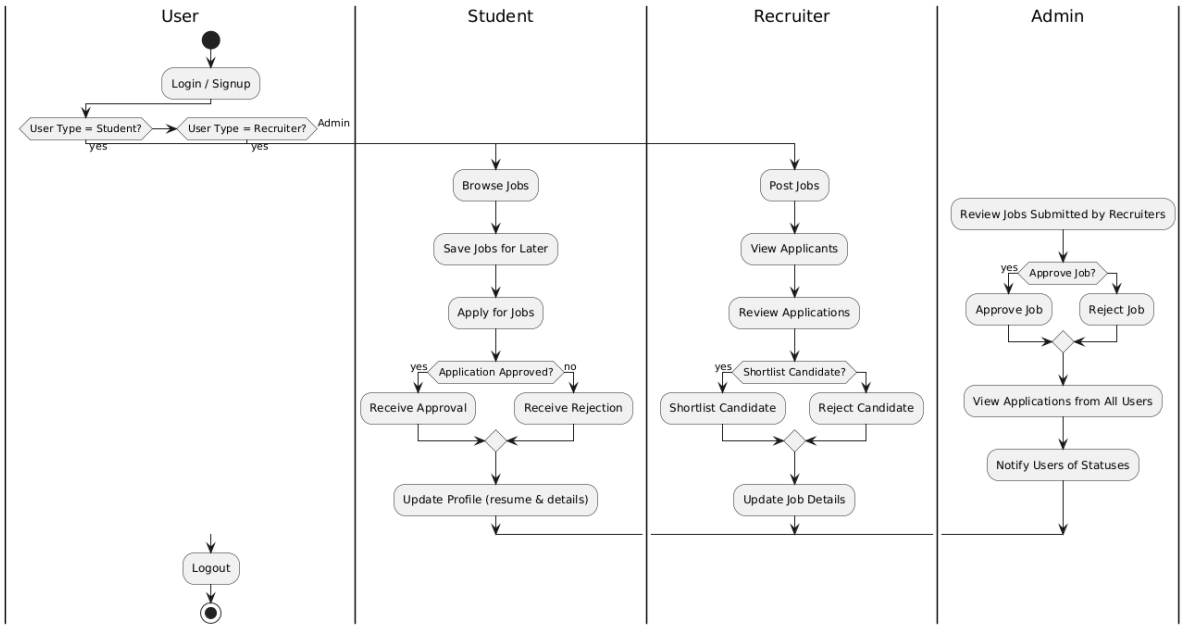
* applications: A list of all **Application** objects.

* admins: A list of all **Admin** users.

◇ Methods:

1. registerStudent(Student student)
2. registerRecruiter(Recruiter recruiter)
3. postJob(Job job)
4. applyToJob(string studentID, string jobID)
5. approveApplication(string applicationID)
6. rejectApplication(string applicationID)
7. sendNotification(string recipientID, string message)

*Activity Diagram



Explanation:

1. Swimlane: User
    1. Start
    2. Activity: Login / Signup
    3. Decision: Determine “User Type”
        - Student Recruiter Admin
2. Student Branch
    1. Activity: Browse Jobs
    2. Activity: Save Jobs for Later
    3. Activity: Apply for Jobs
    4. Decision: Was the application approved?

- Yes → Activity: Receive Approval
- No → Activity: Receive Rejection

5. Activity: Update Profile (resume & details)
6. Merge: Flow rejoins back to User swimlane

3. Recruiter Branch

Activity: Post Jobs

1. Activity: View Applicants
2. Activity: Review Applications
3. Decision: Shortlist candidate?
    - Yes → Activity: Shortlist Candidate
    - No → Activity: Reject Candidate
4. Activity: Update Job Details
5. Merge: Flow rejoins back to User swimlane

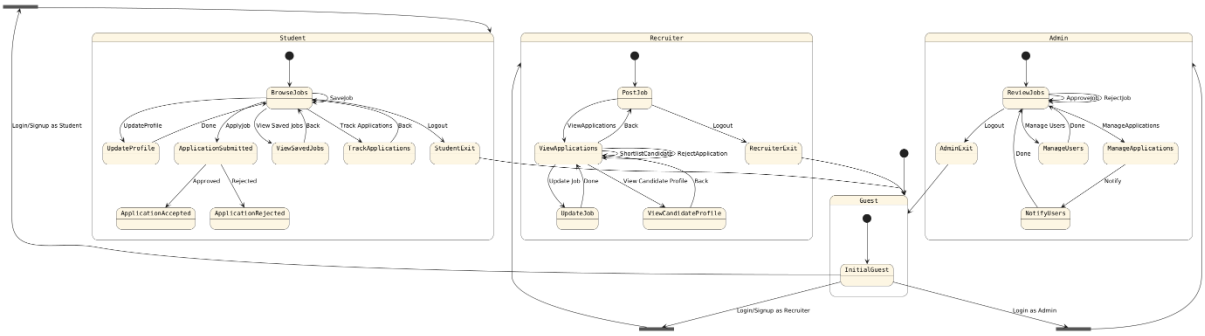
4. Admin Branch

1. Activity: Review Jobs Submitted by Recruiters
2. Decision: Approve job?
    - Yes → Activity: Approve Job
    - No → Activity: Reject Job
3. Activity: View Applications from All Users
4. Activity: Notify Users of Statuses
5. Merge: Flow rejoins back to User swimlane

5. Swimlane: User (convergence)

1. Activity: Logout
2. Stop

*State Diagram



Explanation:

Guest

- InitialGuest:
    - Login/Signup as Student → goes to StudentFork
    - Login/Signup as Recruiter → goes to RecruiterFork
    - Login as Admin → goes to AdminFork

Student Flow

- BrowseJobs:
    - Save a job
    - Apply for jobs → leads to ApplicationSubmitted (approved/rejected)
    - Track Applications
    - Update Profile

- Logout → returns to Guest

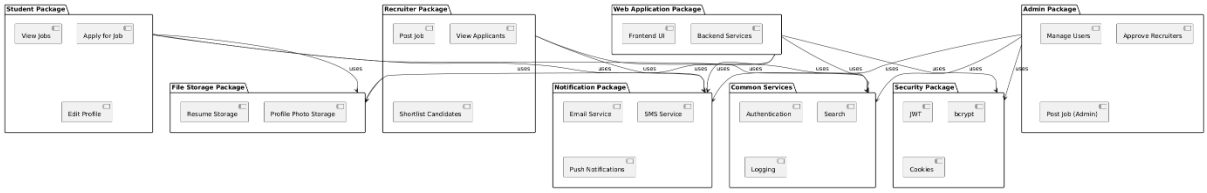
Recruiter Flow

- PostJob:
  - View Applications → shortlist/reject candidates
  - Update Job
  - Logout → returns to Guest

Admin Flow

- ReviewJobs:
  - Approve/Reject Jobs
  - Manage Applications → Notify users
  - Manage Users
  - Logout → returns to Guest

*Package Diagram:



Explanation:

1. Common Services

- Components: Authentication, Search, Logging
- Role: Provides shared services utilized across multiple packages.

2.Security Package

- Components: JWT, bcrypt, Cookies
- Role: Handles security aspects like authentication tokens, password hashing, and session management.

3.Notification Package

- Components: Email Service, SMS Service, Push Notifications
- Role: Manages the delivery of notifications to users through various channels.

4.File Storage Package

- Components: Resume Storage, Profile Photo Storage
- Role: Handles storage and retrieval of user-uploaded files.

5.Student Package

- Components: View Jobs, Apply for Job, Edit Profile
- Role: Encapsulates functionalities available to student users.

6.Recruiter Package

- Components: Post Job, View Applicants, Shortlist Candidates
- Role: Encapsulates functionalities available to recruiter users.

7.Admin Package

- Components: Manage Users, Approve Recruiters, Post Job (Admin)
- Role: Encapsulates administrative functionalities for managing the system.

8.Web Application Package

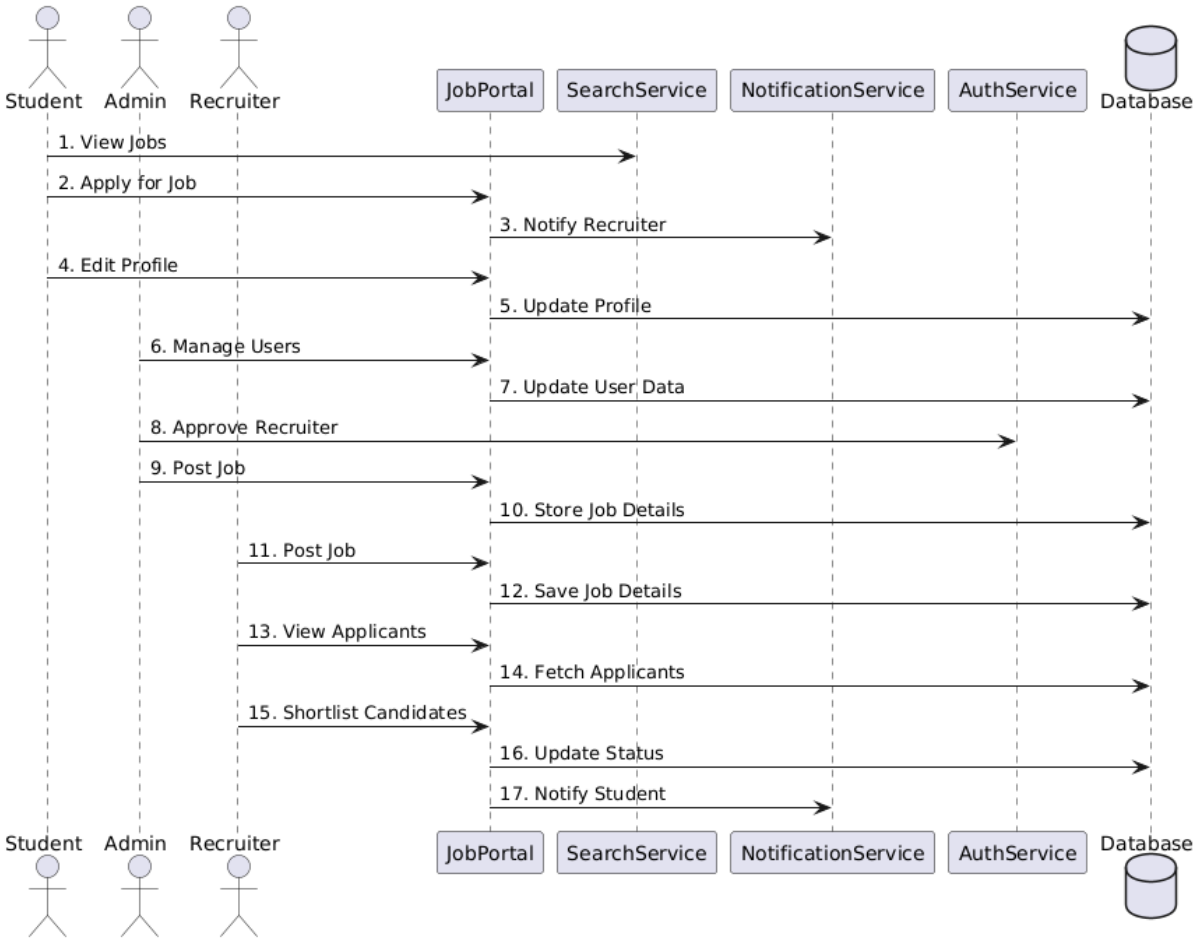
- Components: Frontend UI, Backend Services
- Role: Represents the user interface and backend logic of the application.

Relationships Between Packages:

The diagram uses arrows to depict dependencies and interactions between packages:

- Usage Relationships: Indicated by solid arrows (-->) labeled with uses, showing that one package utilizes the services or components of another.

*Communication Diagram:



Explanation:

Main Components

Actors

- Student: A user looking for jobs.
- Admin: A system administrator who manages users and jobs.
- Recruiter: A user who posts jobs and reviews applicants.

System Components

- JobPortal: Main application logic that handles requests.
- SearchService: Service that helps in finding jobs.
- NotificationService: Sends notifications (email/SMS/in-app).
- AuthService: Manages login/signup, especially for recruiters.

- Database: Stores persistent data (users, jobs, applications, etc.)

Message Flow Explanation

Student Interactions

1. Student -> SearchService: Student views jobs using the search functionality.
2. Student -> JobPortal: Applies for a selected job.
3. JobPortal -> NotificationService: The recruiter is notified of a new application.
4. Student -> JobPortal: Updates their profile (e.g., resume, info).
5. JobPortal -> Database: The new profile data is saved in the database.

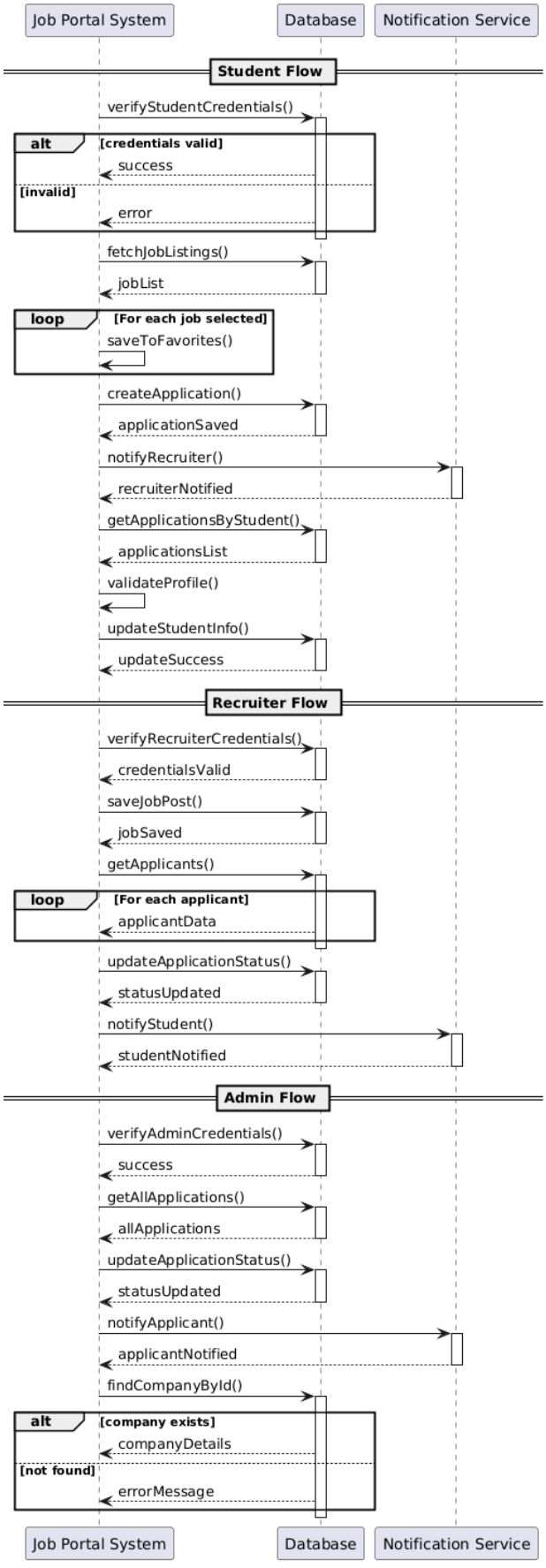
Admin Interactions

6. Admin -> JobPortal: Manages users via the main portal (like suspending or approving).
7. JobPortal -> Database: Updates user records accordingly.
8. Admin -> AuthService: Approves a recruiter account (authentication task)
9. Admin -> JobPortal: Admin manually posts a job on behalf of someone.
10. JobPortal -> Database: The job data is stored in the database.

Recruiter Interactions

11. Recruiter -> JobPortal: Posts a job from their side.
12. JobPortal -> Database: The job details are stored.
13. Recruiter -> JobPortal: Views list of applicants.
14. JobPortal -> Database: Fetches the applicant data.
15. Recruiter -> JobPortal: Shortlists candidates.
16. JobPortal -> Database: Updates status of application (shortlisted/rejected).
17. JobPortal -> NotificationService: Notifies student about their application status.

*Sequence Diagram



Explanation:

1. Student Flow

a. Login & Validation

- verifyStudentCredentials() → Database
    - If valid: return success
    - If invalid: return error

b. Job Interaction

- fetchJobListings() → Database → returns jobList
- Loop through selected jobs: saveToFavorites(job) → Database

c. Application Process

- createApplication() → Database → returns applicationSaved
- notifyRecruiter() → Notification Service → recruiterNotified

d. Application Tracking & Profile Management

- getApplicationsByStudent() → Database → applicationsList
- validateProfile() (likely internal validation)
- updateStudentInfo() → Database → updateSuccess

2. Recruiter Flow

a. Login & Job Posting

- verifyRecruiterCredentials() → Database → credentialsValid
- saveJobPost() → Database → jobSaved

b. Application Management

- getApplicants() → Database
- Loop: for each applicant → return applicantData
- updateApplicationStatus() → Database → statusUpdated
- notifyStudent() → Notification Service → studentNotified

3. Admin Flow

a. Login & Global Application View

- verifyAdminCredentials() → Database → success
- getAllApplications() → Database → allApplications

b. Manage Applications

- updateApplicationStatus() → Database → statusUpdated
- notifyApplicant() → Notification Service → applicantNotified

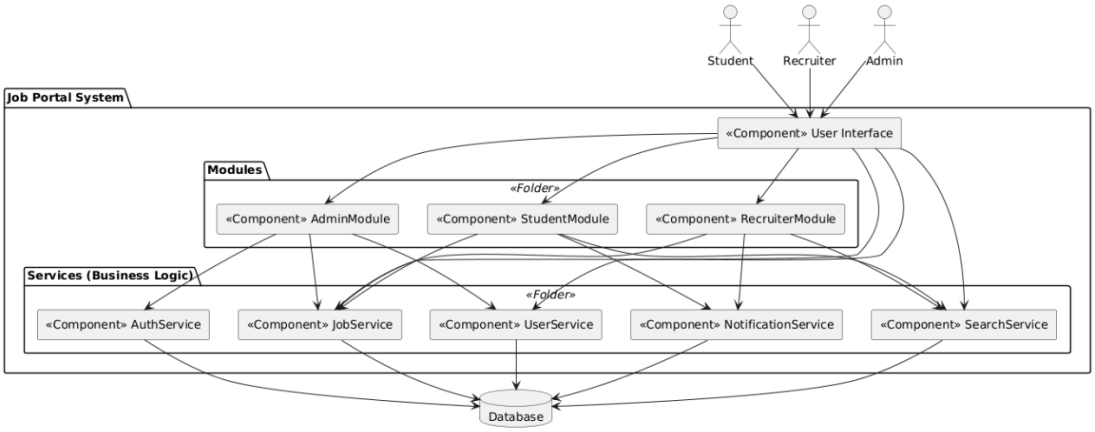
c. Company Lookup

- findCompanyById() → Database
    - If exists: return companyDetails
    - If not found: return errorMessage

System Components Involved

- Job Portal System: Orchestrates user actions.
- Database: Stores user data, job info, applications, etc.
- Notification Service: Sends updates (e.g., application status).

*Component Diagram:



Explanation:

Main System: Job Portal System

This is the core system housing all components.

Users

- Student
- Recruiter
- Admin

All three users interact via the UI component.

User Interface (UI)

- Acts as the entry point for users
- Interacts with StudentModule, AdminModule, RecruiterModule, and core services like JobService, UserService, SearchService

Modules (Feature-based)

- **StudentModule**
  - Uses:
    - JobService (for applying, viewing jobs)
    - SearchService (to search/filter jobs)
    - NotificationService (to receive updates)
- **AdminModule**

- Uses:
    - UserService (managing users)
    - JobService (job validation/moderation)
    - AuthService (credentials verification)

◇ **RecruiterModule**

- Uses:
    - JobService (posting, updating jobs)
    - NotificationService (notifying applicants)
    - SearchService (to search applicants or job stats)

Services (Business Logic)

AuthService

- Handles authentication
- Accesses Database for credential checks

JobService

- Central service for job-related operations (post, update, delete, fetch)
- Talks to Database

UserService

- Manages users' data and roles (Students, Admins, Recruiters)
- Communicates with Database

NotificationService

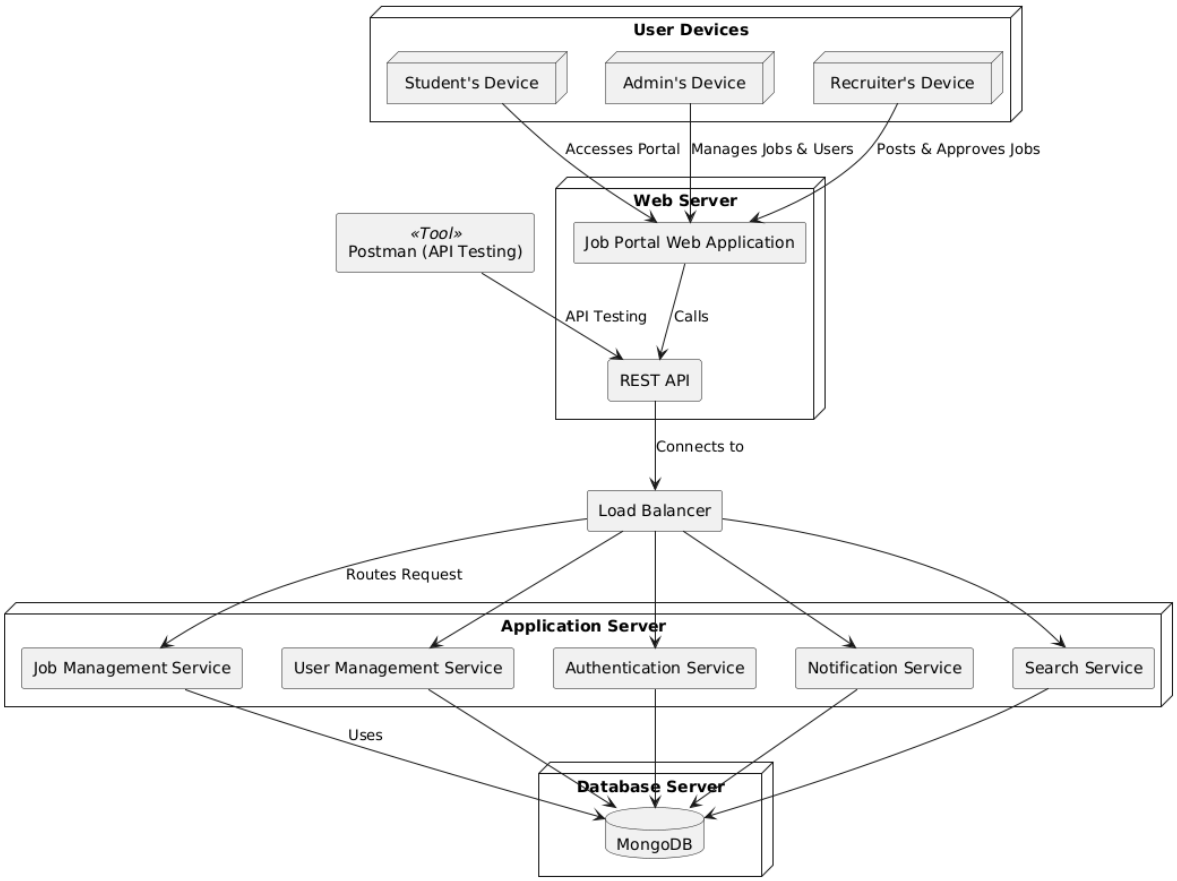
- Sends notifications (e.g., application status, new job alerts)
- Pulls recipient data from Database

SearchService

- Enables searching/filtering of jobs or candidates
- Uses Database for data retrieval

Database: The **shared resource** that supports all services for data persistence.

*Deployement Diagram



Explanation:

The diagram is divided into five main parts:

1. User Devices (Clients)
2. Web Server
3. Application Server
4. Database Server
5. Supporting Infrastructure (Load Balancer and Cache)

1. User Devices

These represent clients accessing the system.

- Student’s Device, Recruiter’s Device, and Admin’s Device

    - Access the web app via browsers
    - Role-based usage:
        - Students: view/search jobs, apply
        - Recruiters: post/view applicants
        - Admins: manage jobs & users
- **Postman**
    - Used by developers/testers for API testing
    - Directly communicates with the REST API

2. Web Server

Hosts the Job Portal Web Application and the REST API.

- WebApp:
    - Frontend/UI served to user browsers
    - Handles user interactions
- API:
    - Backend REST endpoints
    - Interacts with the Load Balancer to route API requests

3. Load Balancer

A routing component that distributes incoming API calls to the appropriate microservices in the Application Server for performance and reliability.

4. Application Server

Hosts the core microservices of the system:

- JobService: Job postings, updates, applications
- UserService: User profiles and role management
- AuthService: Login, registration, password handling

- NotificationService: Sends updates/alerts to users
- SearchService: Filters and fetches job/applicant data

These services:

- Access the MongoDB database

5. Database Server

- MongoDB: NoSQL database storing all persistent data (users, jobs, applications, etc.)

Job Portal mern stack project code in object oriented way in cpp:-

Code:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class User {
protected:
    string name, email, password, profilePhoto;
public:
    User(string n, string e, string p, string photo) : name(n), email(e), password(p), profilePhoto(photo) {}
    virtual void viewProfile() const {
        cout << "Name: " << name << "\nEmail: " << email << "\nProfile Photo: " << profilePhoto << "\n";
    }
    virtual void updateProfile() {
        cout << "Enter new name: "; cin >> name;
        cout << "Enter new profile photo: "; cin >> profilePhoto;
    }
    string getEmail() const { return email; }
    string getPassword() const { return password; }
};

class Job {
public:
    int id;
```

```
    string title, description, location, jobType, experience, position, requirement, role;
    int salary;

    Job(int i, string t, string desc, string l, string jt, string exp, string pos, string req, string r, int
s)
        : id(i), title(t), description(desc), location(l), jobType(jt), experience(exp),
          position(pos), requirement(req), role(r), salary(s) {}

    void display() const {
        cout << "\nID: " << id
             << "\nTitle: " << title
             << "\nDescription: " << description
<< "\nLocation: " << location
             << "\nJob Type: " << jobType
             << "\nExperience: " << experience
             << "\nPosition: " << position
             << "\nRequirement: " << requirement
             << "\nRole: " << role
             << "\nSalary: " << salary << "\n";
    }
};

class Student : public User {
public:
    string skills, bio, resume;
    vector<int> appliedJobs;
    Student(string n, string e, string p, string photo) : User(n, e, p, photo) {}
    void viewProfile() const override {
        User::viewProfile();
        cout << "Skills: " << skills << "\nBio: " << bio << "\nResume: " << resume << "\n";
```

```
    }
    void updateProfile() override {
        User::updateProfile();
        cout << "Enter skills: "; cin.ignore(); getline(cin, skills);
        cout << "Enter bio: "; getline(cin, bio);
        cout << "Enter resume path: "; getline(cin, resume);
    }
    void searchJobs(const vector<Job>& jobs, const string& keyword) {
        for (const auto& job : jobs) {
            if (job.title.find(keyword) != string::npos || job.role.find(keyword) != string::npos) {
                job.display();
            }
        }
    }
    void filterJobs(const vector<Job>& jobs, const string& location, const string& role, int
minSalary) {
        for (const auto& job : jobs) {
            if ((location.empty() || job.location == location) &&
                (role.empty() || job.role == role) &&
                (minSalary == 0 || job.salary >= minSalary)) {
                job.display();
                cout << "Apply to this job? (y/n): ";
                char ch; cin >> ch;
                if (ch == 'y') applyJob(job.id);
            }
        }
    }
    void applyJob(int jobId) {
        appliedJobs.push_back(jobId);
        cout << "Applied to job ID " << jobId << " successfully.\n";
```

```
    }
};

class Recruiter : public User {
public:
    vector<int> postedJobs;
    Recruiter(string n, string e, string p, string photo) : User(n, e, p, photo) {}

    void postJob(vector<Job>& jobs, int& jobIdCounter) {
        string title, description, location, jobType, experience, position, requirement, role;
        int salary;

        cin.ignore();
        cout << "Enter job title: "; getline(cin, title);
        cout << "Enter job description: "; getline(cin, description);
        cout << "Enter location: "; getline(cin, location);
        cout << "Enter job type (Full-time/Part-time/Internship): "; getline(cin, jobType);
        cout << "Enter required experience: "; getline(cin, experience);
        cout << "Enter position: "; getline(cin, position);
        cout << "Enter job requirement: "; getline(cin, requirement);
        cout << "Enter role: "; getline(cin, role);
        cout << "Enter salary: "; cin >> salary;

        jobs.emplace_back(jobIdCounter, title, description, location, jobType, experience,
position, requirement, role, salary);
        postedJobs.push_back(jobIdCounter);
        cout << "Job posted with ID " << jobIdCounter << "\n";
        jobIdCounter++;
    }
```

```
void updateJob(vector<Job>& jobs) {
    cout << "Your Posted Jobs:\n";
    for (int id : postedJobs) {
        for (auto& job : jobs) {
            if (job.id == id) {
                job.display();
                break;
            }
        }
    }

    int jobId;
    cout << "Enter Job ID to update: ";
    cin >> jobId;

    auto it = find(postedJobs.begin(), postedJobs.end(), jobId);
    if (it == postedJobs.end()) {
        cout << "You have not posted job ID " << jobId << ".\n";
        return;
    }

    for (auto& job : jobs) {
        if (job.id == jobId) {
            cin.ignore();
            cout << "Enter new title: "; getline(cin, job.title);
            cout << "Enter new description: "; getline(cin, job.description);
            cout << "Enter new location: "; getline(cin, job.location);
            cout << "Enter new job type: "; getline(cin, job.jobType);
            cout << "Enter new experience: "; getline(cin, job.experience);
```

```
            cout << "Enter new position: "; getline(cin, job.position);
            cout << "Enter new requirement: "; getline(cin, job.requirement);
            cout << "Enter new role: "; getline(cin, job.role);
            cout << "Enter new salary: "; cin >> job.salary;
            cout << "Job updated successfully.\n";
            return;
        }
    }
  }

  void reviewApplications(const vector<Student>& students) {
    for (const auto& s : students) {
      for (int jid : s.appliedJobs) {
        cout << s.getEmail() << " applied for job ID " << jid << ". Approve (a) or Reject (r)? ";
        char ch; cin >> ch;
        if (ch == 'a') cout << "Application approved.\n";
        else cout << "Application rejected.\n";
      }
    }
  }
};

class Company {
public:
  string name, location, description, website, logo;
  Company(string n, string l, string d, string w, string logo)
    : name(n), location(l), description(d), website(w), logo(logo) {}
  void display() const {
    cout << "Company: " << name << " | Location: " << location << "\n"
```

```
            << "Description: " << description << "\nWebsite: " << website
            << "\nLogo: " << logo << "\n";
    }
    void update() {
        cin.ignore();
        cout << "Enter new name: "; getline(cin, name);
        cout << "Enter new location: "; getline(cin, location);
        cout << "Enter new description: "; getline(cin, description);
        cout << "Enter new website: "; getline(cin, website);
        cout << "Enter new logo: "; getline(cin, logo);
    }
};

class Admin : public User {
public:
    Admin(string n, string e, string p, string photo) : User(n, e, p, photo) {}

    void createCompany(vector<Company>& companies) {
        string name, location, description, website, logo;
        cin.ignore();
        cout << "Enter company name: "; getline(cin, name);
        cout << "Enter location: "; getline(cin, location);
        cout << "Enter description: "; getline(cin, description);
        cout << "Enter website: "; getline(cin, website);
        cout << "Enter company logo path: "; getline(cin, logo);
        companies.emplace_back(name, location, description, website, logo);
        cout << "Company created successfully.\n";
    }
```

```
    void updateCompany(vector<Company>& companies) {
        for (int i = 0; i < companies.size(); ++i) {
            cout << i << ". "; companies[i].display();
        }
        int idx;
        cout << "Enter index of company to update: "; cin >> idx;
        if (idx >= 0 && idx < companies.size()) companies[idx].update();
        else cout << "Invalid index.\n";
    }

    void viewApplicationStats(const vector<Student>& students) {
        cout << "Application stats:\n";
        for (const auto& s : students) {
            cout << s.getEmail() << " applied to " << s.appliedJobs.size() << " job(s).\n";
        }
    }
};

bool login(const string& email, const string& password, const vector<Student>& students,
Student*& sPtr) {
    for (auto& s : students) {
        if (s.getEmail() == email && s.getPassword() == password) {
            sPtr = const_cast<Student*>(&s);
            return true;
        }
    }
    return false;
}
```

```
bool login(const string& email, const string& password, const vector<Recruiter>& recruiters,
Recruiter*& rPtr) {
    for (auto& r : recruiters) {
        if (r.getEmail() == email && r.getPassword() == password) {
            rPtr = const_cast<Recruiter*>(&r);
            return true;
        }
    }
    return false;
}

bool login(const string& email, const string& password, const vector<Admin>& admins,
Admin*& aPtr) {
    for (auto& a : admins) {
        if (a.getEmail() == email && a.getPassword() == password) {
            aPtr = const_cast<Admin*>(&a);
            return true;
        }
    }
    return false;
}

int main() {
    vector<Student> students;
    vector<Recruiter> recruiters;
    vector<Admin> admins;
    vector<Job> jobs;
    vector<Company> companies;
    int jobIdCounter = 1;
```

```
    while (true) {
        cout << "\n--- Job Portal ---\n1. Signup\n2. Login\n3. Exit\nChoose: ";
        int choice; cin >> choice;

        if (choice == 1) {
            string name, email, pass, role, photo;
            cout << "Name: "; cin >> name;
            cout << "Email: "; cin >> email;
            cout << "Password: "; cin >> pass;
            cout << "Role (student/recruiter/admin): "; cin >> role;
            cout << "Profile photo: "; cin >> photo;

            if (role == "student") {
                students.emplace_back(name, email, pass, photo);
                cout << "Student signed up.\n";
            } else if (role == "recruiter") {
                recruiters.emplace_back(name, email, pass, photo);
                cout << "Recruiter signed up.\n";
            } else if (role == "admin") {
                admins.emplace_back(name, email, pass, photo);
                Admin* admin = &admins.back();
                cout << "Admin signed up and logged in.\n";
                while (true) {
                    cout << "\nAdmin Menu:\n1. Create Company\n2. Update Company\n3. View
Application Stats\n4. Logout\nChoose: ";
                    int c; cin >> c;
                    if (c == 1) admin->createCompany(companies);
                    else if (c == 2) admin->updateCompany(companies);
                    else if (c == 3) admin->viewApplicationStats(students);
                    else break;
```

```
            }
        }

    } else if (choice == 2) {
        string email, pass;
        cout << "Email: "; cin >> email;
        cout << "Password: "; cin >> pass;

        Student* s = nullptr;
        Recruiter* r = nullptr;
        Admin* a = nullptr;

        if (login(email, pass, students, s)) {
            while (true) {
                cout << "\nStudent Menu:\n1. View Profile\n2. Update Profile\n3. Search Jobs\n4. Filter Jobs\n5. Apply for Job\n6. Logout\nChoose: ";
                int c; cin >> c;
                if (c == 1) s->viewProfile();
                else if (c == 2) s->updateProfile();
                else if (c == 3) {
                    string kw; cout << "Keyword: "; cin >> kw;
                    s->searchJobs(jobs, kw);
                } else if (c == 4) {
                    string loc, role; int sal;
                    cout << "Location: "; cin >> loc;
                    cout << "Role: "; cin >> role;
                    cout << "Min Salary: "; cin >> sal;
                    s->filterJobs(jobs, loc, role, sal);
                } else if (c == 5) {
                    int jid; cout << "Job ID: "; cin >> jid;
```

```
                s->applyJob(jid);
            } else break;
        }

    } else if (login(email, pass, recruiters, r)) {
        while (true) {
            cout << "\nRecruiter Menu:\n1. Post Job\n2. Review Applications\n3. Update Job\n4. Logout\nChoose: ";
            int c; cin >> c;
            if (c == 1) r->postJob(jobs, jobIdCounter);
            else if (c == 2) r->reviewApplications(students);
            else if (c == 3) r->updateJob(jobs);
            else break;
        }

    } else if (login(email, pass, admins, a)) {
        while (true) {
            cout << "\nAdmin Menu:\n1. Create Company\n2. Update Company\n3. View Application Stats\n4. Logout\nChoose: ";
            int c; cin >> c;
            if (c == 1) a->createCompany(companies);
            else if (c == 2) a->updateCompany(companies);
            else if (c == 3) a->viewApplicationStats(students);
            else break;
        }
    } else {
        cout << "Invalid login.\n";
    }

} else break;
```

```
    }
    return 0;
}
```

output:-

```
--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 1
Name: Anju
Email: anju@gmail.com
Password: 123
Role (student/recruiter/admin): student
Profile photo: C:\Users\anjal\OneDrive\Anjali.jpg
Student signed up.

--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 2
Email: anju@gmail.com
Password: 123

Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
4. Filter Jobs
5. Apply for Job
6. Logout
Choose: 1
Name: Anju
Email: anju@gmail.com
Profile Photo: C:\Users\anjal\OneDrive\Anjali.jpg
Skills:
Bio:
Resume:

Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
```

```
Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
4. Filter Jobs
5. Apply for Job
6. Logout
Choose: 2
Enter new name: angel
Enter new profile photo:

C:\Users\anjal\OneDrive\Anjali.jpg
Enter skills: react,nodejs
Enter bio: developer
Enter resume path: C:\Users\anjal\OneDrive\resume.jpg

Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
4. Filter Jobs
5. Apply for Job
6. Logout
Choose: 4
Location: mumbai
Role: frontend developer
```

```
Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
4. Filter Jobs
5. Apply for Job
6. Logout
Choose: 5
Job ID: 123456
Applied to job ID 123456 successfully.

Student Menu:
1. View Profile
2. Update Profile
3. Search Jobs
4. Filter Jobs
5. Apply for Job
6. Logout
Choose: 6

--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 3


=== Code Execution Successful ===
```

```
--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 1
Name: anju
Email: anju@gmail.com
Password: 123
Role (student/recruiter/admin): recruiter
Profile photo: "c/image.jpg"
Recruiter signed up.

--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 2
Email: anju@gmail.com
Password: 123

Recruiter Menu:
1. Post Job
2. Review Applications
3. Update Job
4. Logout
Choose: 1
Enter job title: frontend developer
Enter job description: dream job
Enter location: mumbai
Enter job type (Full-time/Part-time/Internship): Full-time
Enter required experience: 3
Enter position: 4
Enter job requirement: react,nodejs
Enter role: frontend developer
Enter salary: 9lpa
Job posted with ID 1
```

```
Recruiter Menu:
1. Post Job
2. Review Applications
3. Update Job
4. Logout
Choose: 3
Your Posted Jobs:

ID: 1
Title: full stack developer
Description: dream job
Location: mumbai
Job Type: full time
Experience: 4
Position: 6
Requirement: nodejs,react,mongodb
Role: full stack developer
Salary: 1000000
Enter Job ID to update: 1
Enter new title:
Enter new description:
Enter new location:
Enter new job type:
Enter new experience: 5
Enter new position:
Enter new requirement:
Enter new role:
Enter new salary:

00000

Job updated successfully.

Recruiter Menu:
1. Post Job
2. Review Applications
3. Update Job
4. Logout
```

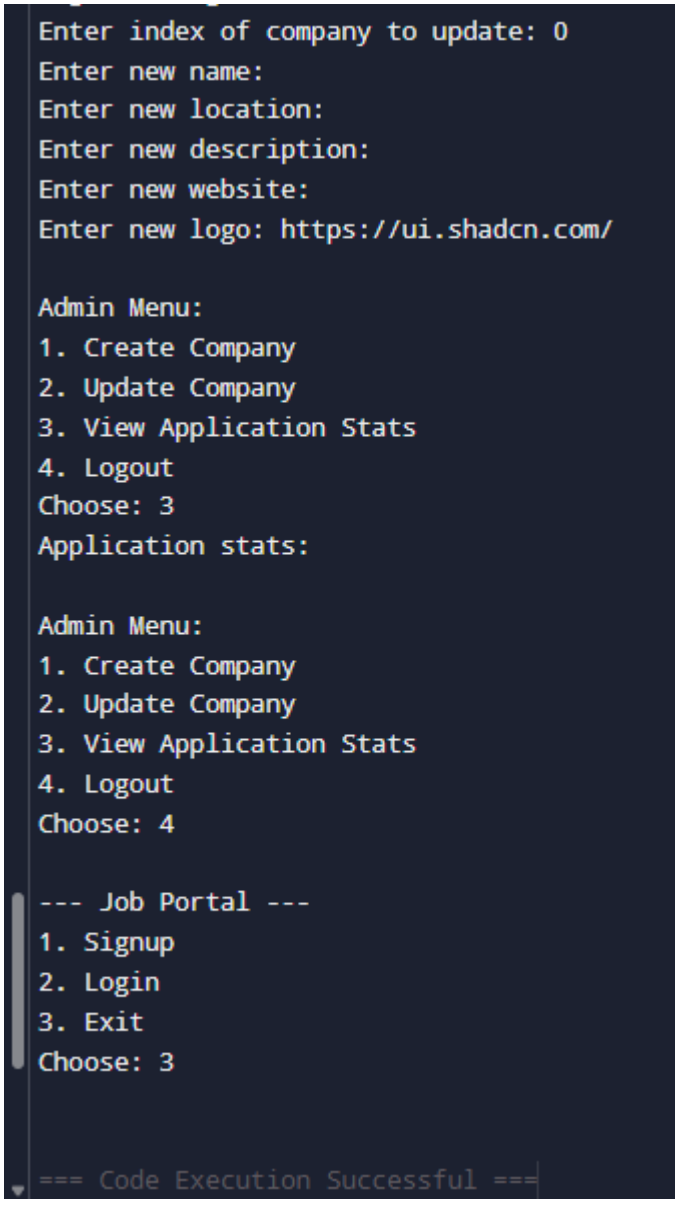
```
Recruiter Menu:
1. Post Job
2. Review Applications
3. Update Job
4. Logout
Choose: 4

--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 3
```

```
--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 1
Name: anju
Email: anju@gmail.com
Password: 123
Role (student/recruiter/admin): admi
Profile photo: "c/image.jpg"
Admin signed up and logged in.

Admin Menu:
1. Create Company
2. Update Company
3. View Application Stats
4. Logout
Choose: 1
Enter company name: google
Enter location: pune
Enter description: dream job
Enter website: https://ui.shadcn.com
Enter company logo path: "c/logo"
Company created successfully.

Admin Menu:
1. Create Company
2. Update Company
3. View Application Stats
4. Logout
Choose: 2
0. Company: google | Location: pune
Description: dream job
Website: https://ui.shadcn.com/
Logo: "c/logo"
Enter index of company to update: 0
Enter new name:
Enter new location:
Enter new description:
```


Enter index of company to update: 0
Enter new name:
Enter new location:
Enter new description:
Enter new website:
Enter new logo: https://ui.shadcn.com/
Admin Menu:
1. Create Company
2. Update Company
3. View Application Stats
4. Logout
Choose: 3
Application stats:
Admin Menu:
1. Create Company
2. Update Company
3. View Application Stats
4. Logout
Choose: 4
--- Job Portal ---
1. Signup
2. Login
3. Exit
Choose: 3
=== Code Execution Successful ===

Test case:

Test case for student class

Code for (googletest):

```
#include <gtest/gtest.h>
#include <iostream>
#include <vector>
#include <string>
using namespace std;
class User {
protected:
   string name, email, password, profilePhoto;
public:
   User(string n, string e, string p, string photo) : name(n), email(e), password(p), profilePhoto(photo) {}
   virtual void viewProfile() const {
      cout << "Name: " << name << "\nEmail: " << email << "\nProfile Photo: " << profilePhoto << "\n";
   }
   virtual void updateProfile() {
      cout << "Enter new name: "; cin >> name;
      cout << "Enter new profile photo: "; cin >> profilePhoto;
   }
   string getEmail() const { return email; }
   string getPassword() const { return password; }
};

class Student : public User {
```

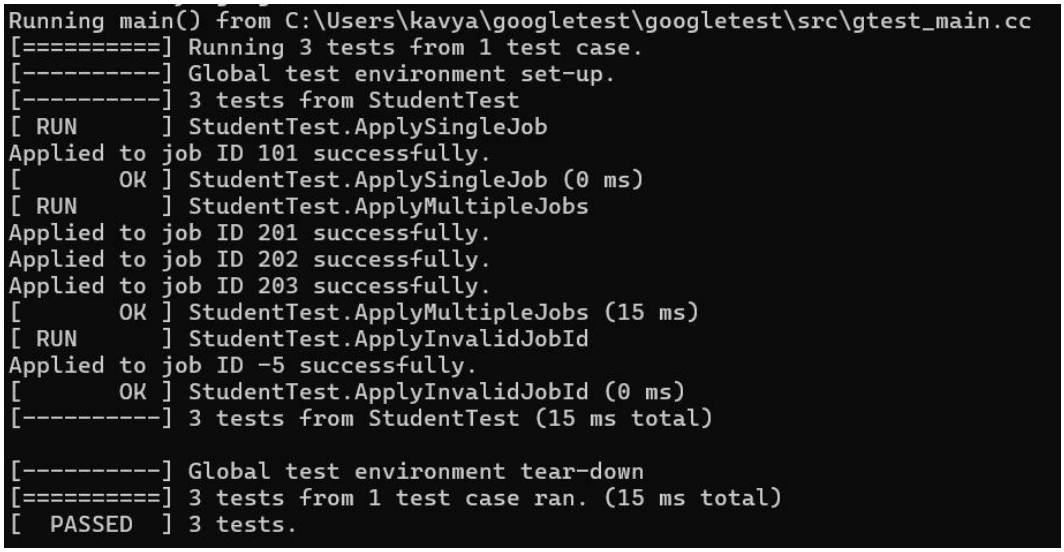
```
public:
    string skills, bio, resume;
    vector<int> appliedJobs;
    Student(string n, string e, string p, string photo) : User(n, e, p, photo) {}
    void viewProfile() const override {
        User::viewProfile();
        cout << "Skills: " << skills << "\nBio: " << bio << "\nResume: " << resume <<
"\n";
    }
    void updateProfile() override {
        User::updateProfile();
        cout << "Enter skills: "; cin.ignore(); getline(cin, skills);
        cout << "Enter bio: "; getline(cin, bio);
        cout << "Enter resume path: "; getline(cin, resume);
    }
    void applyJob(int jobId) {
        appliedJobs.push_back(jobId);
        cout << "Applied to job ID " << jobId << " successfully.\n";
    }
};
TEST(StudentTest, ApplySingleJob) {
    Student s("Alice", "alice@mail.com", "pass123", "photo.jpg");
    s.applyJob(101);
    ASSERT_EQ(s.appliedJobs.size(), 1);
    EXPECT_EQ(s.appliedJobs[0], 101);
}
```

```
TEST(StudentTest, ApplyMultipleJobs) {
    Student s("Bob", "bob@mail.com", "pass456", "pic.png");
    s.applyJob(201);
    s.applyJob(202);
    s.applyJob(203);

    ASSERT_EQ(s.appliedJobs.size(), 3);
    EXPECT_EQ(s.appliedJobs[0], 201);
    EXPECT_EQ(s.appliedJobs[1], 202);
    EXPECT_EQ(s.appliedJobs[2], 203);
}

TEST(StudentTest, ApplyInvalidJobId) {
    Student s("Charlie", "charlie@mail.com", "char123", "char.jpg");
    s.applyJob(-5);  // Currently allowed
    ASSERT_EQ(s.appliedJobs.size(), 1);
    ASSERT_EQ(s.appliedJobs[0], -5);
}
```

Output:



Herosection test: unit testing.

This test suite verifies the functionality and rendering of the HeroSection component, which serves as the main hero/banner section of a job portal application. The tests ensure that:

· Component Rendering

* Verifies all critical UI elements are displayed correctly:

*Main heading ("Search, Apply & Get Your Dream Jobs")

*Subheading tagline ("No. 1 Job Hunt Website")

* Description paragraph

* Search input field

* Search button with icon

· Search Functionality

Tests the search feature by:

* Simulating user input in the search field

* Verifying the input state updates correctly

* Confirming Redux actions are dispatched with the correct query

* Checking navigation to the browse page after submission

· Edge Cases

Validates behavior when:

* Submitting with empty search query

*Handling special text formatting (curly apostrophes in the description)

· Integration Points

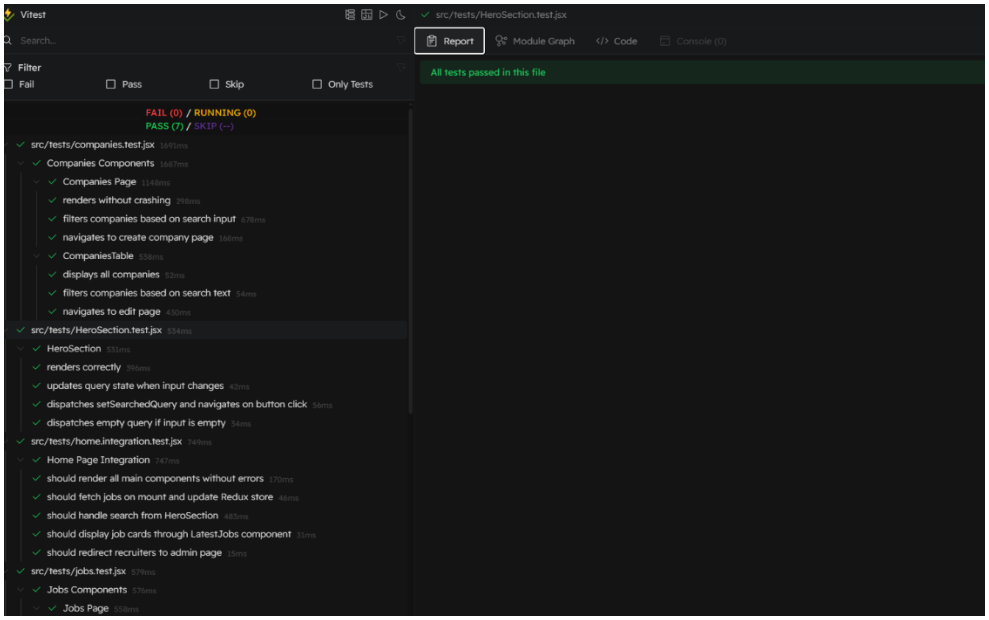
Mocks external dependencies to test integration with:

* Redux store (jobSlice reducer)

*React Router navigation

* UI components (Button, Search icon)

testing output:



Home page and latest job card test:- Integration testing

This integration test suite verifies the behavior of the Home Page and its components in a job portal application. Here's a concise breakdown:

What It Tests:

1. Rendering

a. Verifies all main components (Navbar, HeroSection, CategoryCarousel, Footer) render correctly.

2. Data Fetching & Redux State

a. Ensures jobs are fetched on mount and stored in Redux.

b. Checks if the LatestJobs component displays jobs from the Redux store.

3. User Interactions

a. Tests the search functionality in HeroSection (updates Redux + navigates to /browse).

b. Confirms recruiters are redirected to /admin/companies.

4. Edge Cases

a. Validates the UI shows "no job available" when the store is empty.

Key Features:

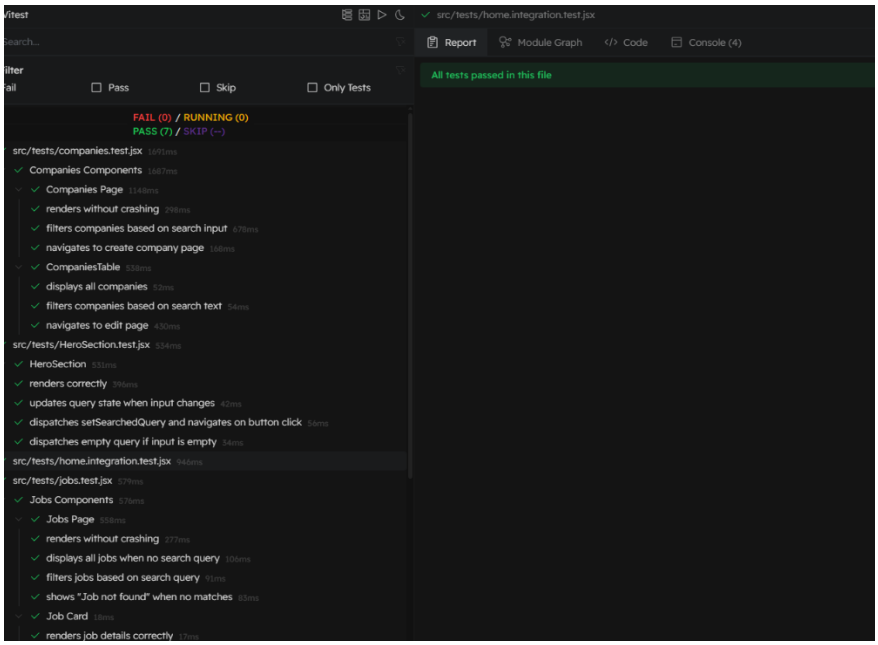
· Mocks: Axios (API calls), child components, and React Router.

· Async Handling: Uses act() and waitFor to manage state updates.

· Redux Integration: Tests state changes after API calls.

· UI Consistency: Checks if the DOM reflects the expected data.

Testing output:-



Job.test:-Unit+integration testing

Purpose of the Test

1. Jobs.jsx – the main listing page that displays filtered or full job data.
2. Job.jsx – an individual job card component.

· **Mocked dependencies:**

o Navbar, Bookmark, and Circle icons are mocked with static components.

o useNavigate from react-router-dom is mocked to prevent real navigation.

o useDispatch from react-redux is mocked to track action dispatching.

· Dummy data: dummyJobs is a sample job list used to simulate actual job data stored in Redux.

· Redux store: The test dynamically creates a mocked Redux store using @reduxjs/toolkit's configureStore.

Test Sections

◇ describe('Jobs Page', …)

* renders without crashing

Ensures the Jobs component loads and renders the mocked Navbar.

* displays all jobs when no search query

Renders all jobs if searchedQuery is empty.

* filters jobs based on search query

Simulates filtering: if searchedQuery is "frontend", only relevant jobs are shown.

*shows "Job not found" when no matches

If the query doesn’t match any job, it renders "Job not found".

◇ describe('Job Card', …)

* renders job details correctly

Tests that the Job component correctly displays job title, company, and icons for a single job

Testing output