MERN JOB BOARD REPORT

Project Overview

This project implements a full-stack job board application using the MERN stack (MongoDB, Express, React, Node.js), enhanced with AWS Cognito for secure user authentication. The backend exposes RESTful API endpoints to handle job postings and resume submissions, while the frontend provides a user interface for admins to post jobs and for users to apply. The entire application is deployed using an AWS EC2 instance for the backend and Ngrok for frontend HTTPS tunneling during deployment. MongoDB Atlas is used for cloud-hosted database storage.

Objectives

- Develop a RESTful API with POST and GET endpoints
- Store and retrieve job and resume data using MongoDB Atlas
- Host the backend on an AWS EC2 instance
- Build frontend using React and Vite
- Implement user and admin authentication using AWS Cognito
- Conditionally render pages and components based on user roles

Technology Stack

Component	Tools Used	Description
Frontend	React 19, Vite, React Router	Builds the user interface and handles page routing
Backend	Node.js, Express	Handles API logic and server-side operations
Database	MongoDB Atlas (Cloud)	A cloud database that stores application data, integrates with Node.js
Authentication	AWS Cognito	Manages user login and role-based access
Deployment	AWS EC2 (Linux)	Hosts the backend on a cloud server

System Architecture

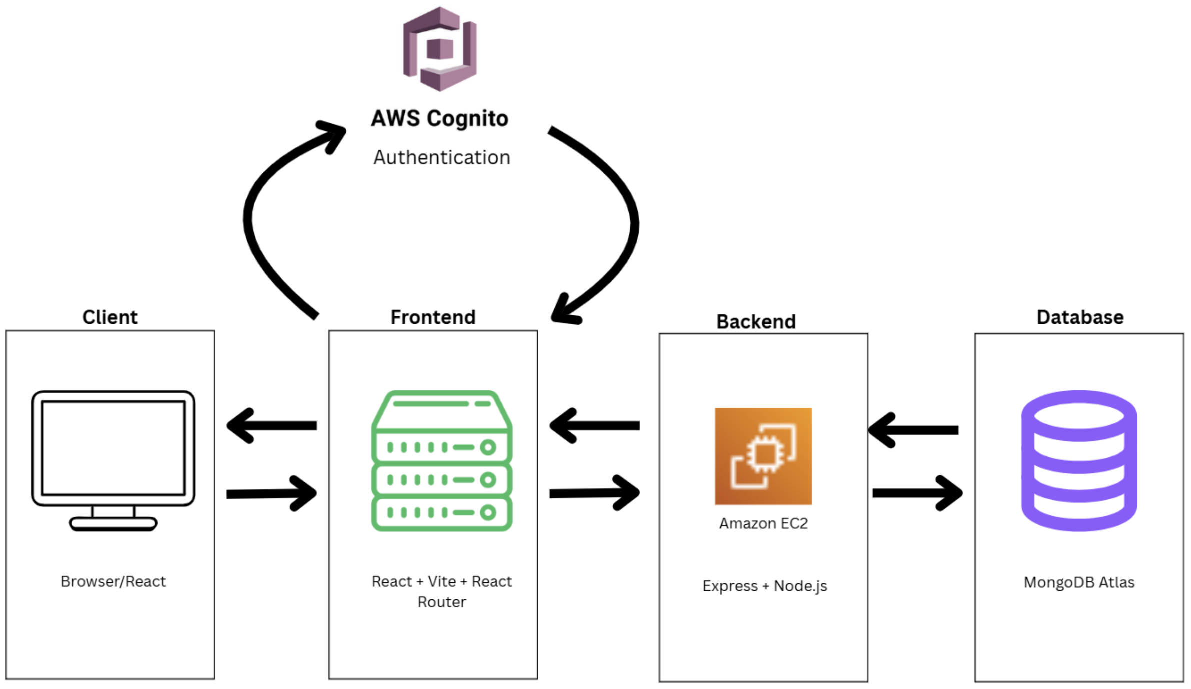


Fig. 1 – System Architecture Diagram

- Client interacts with the Frontend, triggering API requests and navigation.
- Frontend sends HTTP requests to Backend on EC2.
- Backend queries and updated MongoDB Atlas using Mongoose.
- Client also redirects to AWS Cognito for login, then receives a token which is exchange for user data.
- Deployment layers include Vite for frontend and EC2 + Node.js for backend.

Backend

1. Project Setup

- Created a folder structure to separate backend (server) and frontend (client) code
- Initialized the backend project using:

```
npm init -y
```

- Installed necessary dependencies:

```
npm install express mongodb cors
```

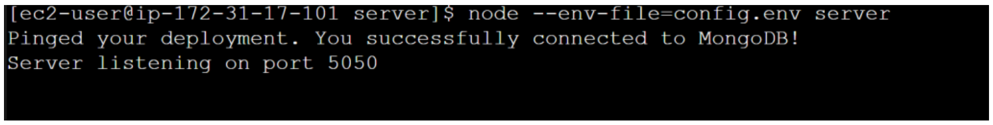
2. Environment Configuration

- Created a file named **config.env** in the **server** directory
- Stored credentials and configuration variables:

```
ATLAS_URI=mongodb+srv://<username>:<password>@<cluster>.mongo
db.net/employees?retryWrites=true&w=majority
```

3. Database Connection

- Created a **db/connection.js** file to handle the MongoDB Atlas connection.
- Used the **mongodb** driver to connect MongoDB atlas, initialize database named **employees** and confirm successful connection with a ping.



```
[ec2-user@ip-172-31-17-101 server]$ node --env-file=config.env server
Pinged your deployment. You successfully connected to MongoDB!
Server listening on port 5050
```

Fig. 2 – Connecting database to port 5050 for listening

4. Server Initialization

- Created **server.js** in the **server** directory
- Setup Express application
- Started the server using port **5050** defined in **config.env**

5. API Routes

- Created a **routes/record.js** file for handling all record operations
- Implemented the following endpoints:

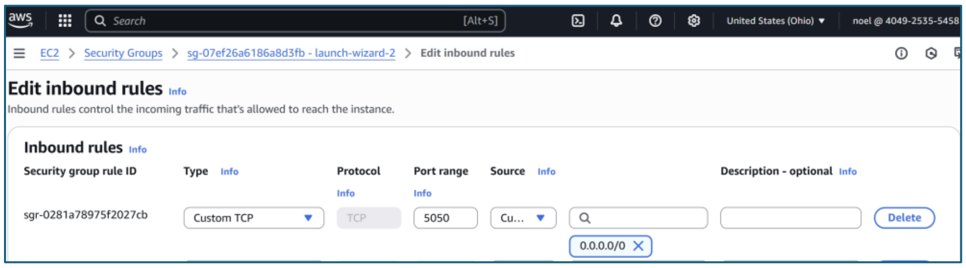
Method	**Endpoint**	**Description**
GET	**/record**	Get all records
GET	**/record/id**	Get a specific record
POST	**/record**	Create a new record
DELETE	**/record/id**	Delete a record

6. Deploying AWS EC2

- Launched a Linux EC2 Instance
- Installed Node.js (v20) and npm
- Transferred backend project files to EC2
- Installed project dependencies with npm install
- Ran the server with:

```
node --env-file=config.env server.js
```

- Opened **port 5050** in the EC2 security group for public access



EC2 > Security Groups > sg-07ef26a6186a8d3fb - launch-wizard-2 > Edit inbound rules

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules Info

Security group rule ID	Type	Protocol	Port range	Source		Description - optional
sgr-0281a78975f2027cb	Custom TCP	TCP	5050	Cu...	0.0.0.0/0	

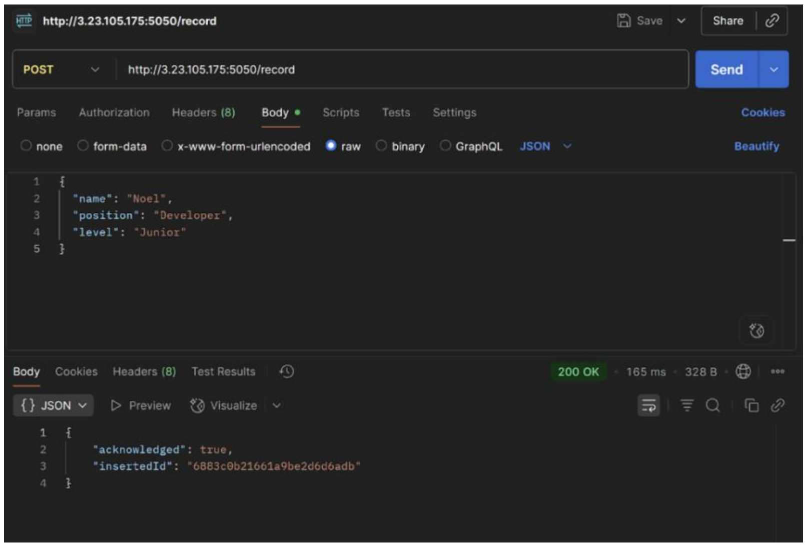
Fig. 3 – Opening Port 5050 on Amazon EC2 security group to ensure connectivity.

7. Testing and Verification

- Used Postman to send **POST** requests to create records and **GET** requests to fetch records.
- Verified successful responses and database changes in MongoDB Atlas.

Example **POST** (JSON):

```
{
  "name": "Noel",
  "position": "Developer",
  "level": "Junior"
}
```



http://3.23.105.175:5050/record

POST http://3.23.105.175:5050/record

```
{
  "name": "Noel",
  "position": "Developer",
  "level": "Junior"
}
```

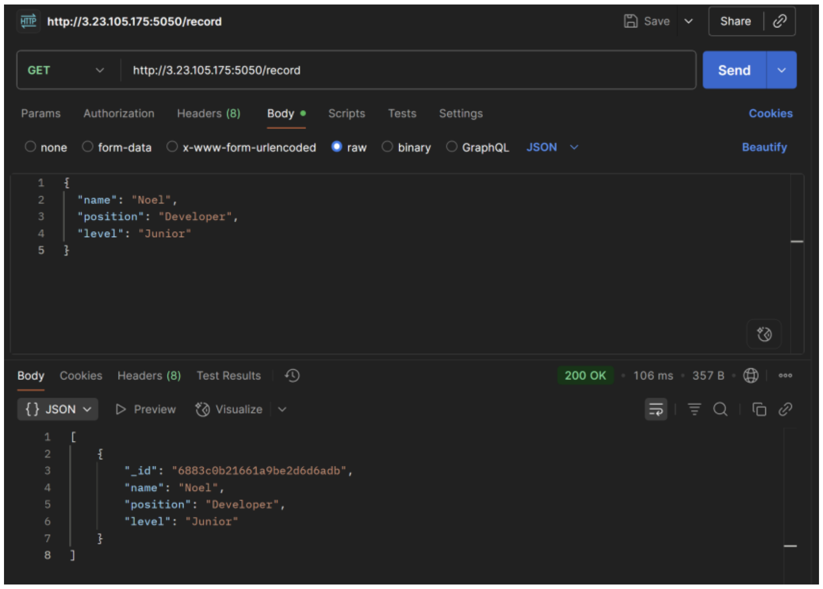
200 OK · 165 ms · 328 B

```
{
    "acknowledged": true,
    "insertedId": "6883c0b21661a9be2d6d6adb"
}
```

Fig. 4 – Verifying POST method

Example **GET** URL:

http://3.23.105.175:5050/record



http://3.23.105.175:5050/record

GET http://3.23.105.175:5050/record

```
{
  "name": "Noel",
  "position": "Developer",
  "level": "Junior"
}
```

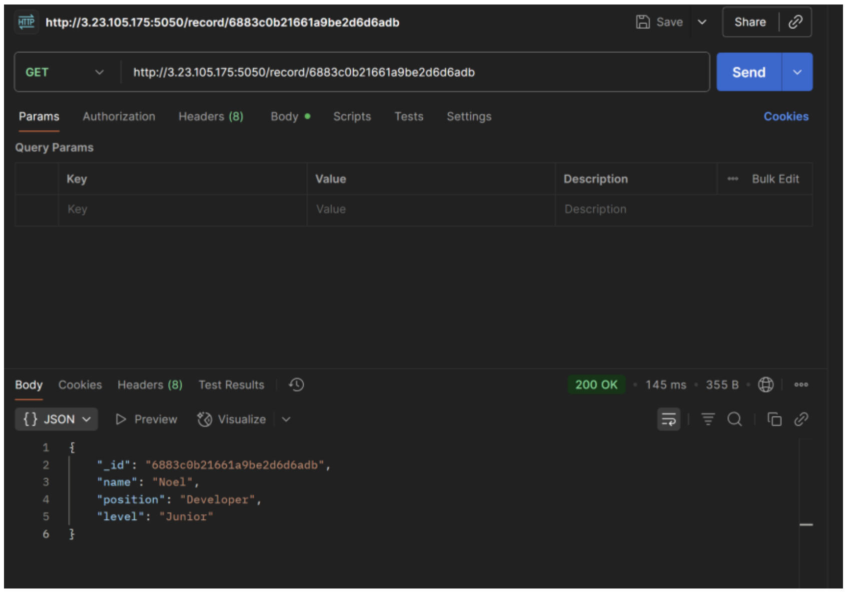
200 OK · 106 ms · 357 B

```
[
    {
        "_id": "6883c0b21661a9be2d6d6adb",
        "name": "Noel",
        "position": "Developer",
        "level": "Junior"
    }
]
```

Fig. 5 – Verifying GET method

Example **GET** URL **using ID**:

http://3.23.105.175:5050/record/6883c0b21661a9be2d6d6adb



http://3.23.105.175:5050/record/6883c0b21661a9be2d6d6adb

GET http://3.23.105.175:5050/record/6883c0b21661a9be2d6d6adb

Query Params

Key	Value	Description
Key	Value	Description

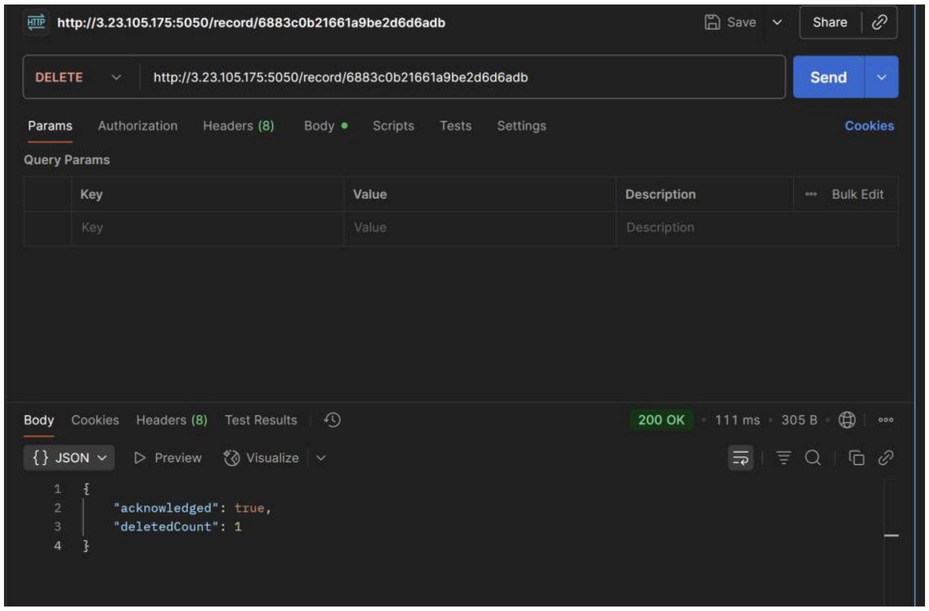
200 OK · 145 ms · 355 B

```
{
  "_id": "6883c0b21661a9be2d6d6adb",
  "name": "Noel",
  "position": "Developer",
  "level": "Junior"
}
```

Fig. 6 – Verifying GET method using ID

Example **DELETE** URL:

http://3.23.105.175:5050/record/6883c0b21661a9be2d6d6adb



```
{
    "acknowledged": true,
    "deletedCount": 1
}
```

Fig. 7 – Verifying POST method

Conclusion and Reflection

This project provided valuable hands on experience in building and deploying a complete backend service using the MERN stack. I learned how to:

- Set up and manage an EC2 instance on AWS
- Use Node.js and Express to build RESTful API endpoints
- Connect a cloud-hosted MongoDB Atlas database to create and retrieve data.

Challenges

- Troubleshooting MongoDB Atlas connection issues due to TLS and not whitelisting my IP.
- Configuring security groups and firewall settings for public access on EC2

Overcoming these challenges has improved my confidence in full-stack development and cloud deployment. This experience has also shown me the importance of reading documentation carefully.

The final outcome is a functional, testable backend system that meets the original project specifications and can be used as a foundation for future endeavors with full-stack.

Frontend

1. **Project Setup**

   The frontend was created using React 19 and Vite for fast development and build optimization. The folder was initialized separately within the project root using:

```
npm create vite@latest client -- --template react
cd client
npm install
```

2. **Environment Configuration**

   A **.env** file was created to manage environment variables securely, such as the Ngrok callback URL for AWS Cognito:

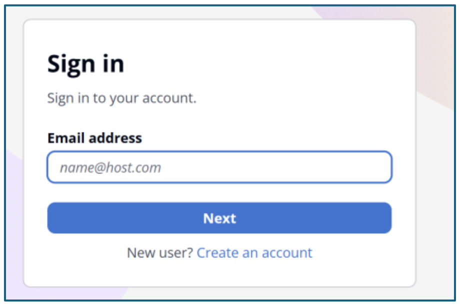
```
VITE_NGROK_URL=https://your-ngrok-url.ngrok-free.app
VITE_NGROK_HOST=your-ngrok-url.ngrok-free.app
```

   These variables are used in Vite's config and for AWS Cognito redirection as it requires HTTPS.

3. **Authentication with AWS Cognito**

   The application uses **AWS Cognito** for user authentication. On clicking "Login", the user is redirected to the Cognito-hosted UI. After successful login, Cognito returns a code to the frontend, which is exchanged for access and ID tokens.

   The token is then decoded to extract the user information and roles, this determines the accessibility of UI.



Sign in

Sign in to your account.

Email address

name@host.com

Next

New user? Create an account

Fig. 8 – Upon selecting 'Login' the user will be redirected to AWS Cognito to Sign In or Create an account.

4. App Routing and Components

React Router is used to manage the pages:

/	→	Displays job listings
/login	→	Triggers Cognito login
/post	→	Post job (admin only)
/resume/:jobId	→	Submit resume (user only)
/applications/:jobId	→	View resumes (admin only)

Components include:

- Home.jsx – Lists Job's
- PostJob.jsx – Form for admins to post new jobs
- SubmitResume.jsx – Allows for users to apply for jobs
- Applications.jsx – Lists submitted resumes (admin only)

5. Integration with Backend

The frontend uses **fetch()** calls to communicate with the backend API hosted on AWS EC2(e.g., http://3.23.105.175:5050/api/record). Routes include:

- **GET /record** to fetch jobs
- **POST /record** to create jobs/resumes

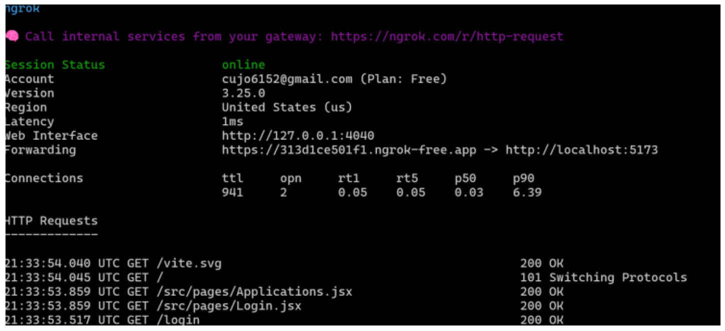
The type field helps determine if an entry is either a "job" or "resume" record in MongoDB.

6. HTTPS Access via Ngrok

Since AWS Cognito requires a secure 'redirect_uri', **Ngrok** was used to expose the local frontend (3.23.105.175:5173) as HTTPS:

```
Ngrok http 5173
```

This allowed Cognito to return the auth code to a valid HTTPS endpoint during development.



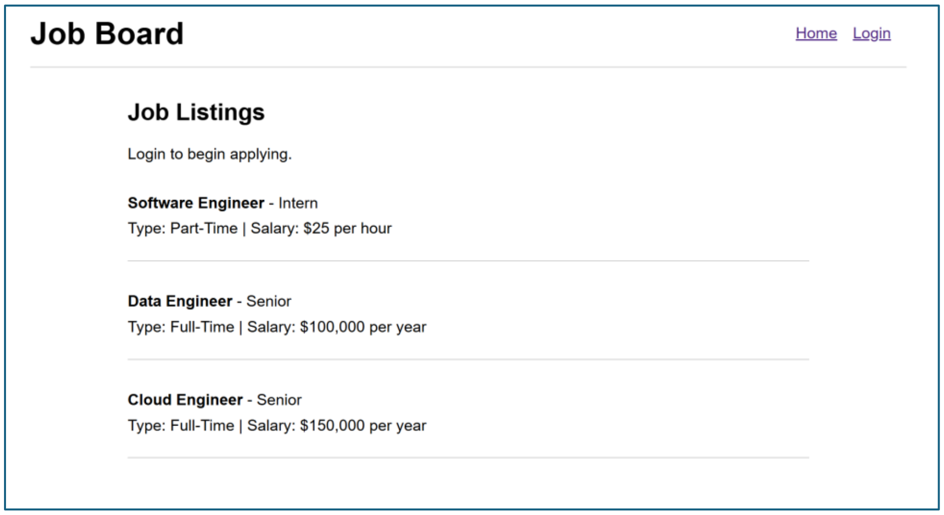
```
ngrok

Call internal services from your gateway: https://ngrok.com/r/http-request

Session Status                online
Account                       cujo6152@gmail.com (Plan: Free)
Version                       3.25.0
Region                        United States (us)
Latency                       1ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://313d1ce501f1.ngrok-free.app -> http://localhost:5173

Connections                   ttl     opn     rt1     rt5     p50     p90
                              941     2       0.05    0.05    0.03    6.39

HTTP Requests
-------------

21:33:54.040 UTC GET /vite.svg                                  200 OK
21:33:54.045 UTC GET /                                          101 Switching Protocols
21:33:53.859 UTC GET /src/pages/Applications.jsx                200 OK
21:33:53.859 UTC GET /src/pages/Login.jsx                       200 OK
21:33:53.517 UTC GET /login                                     200 OK
```

Fig. 9 – Ngrok setup to begin redirecting AWS Cognito back to application

7. Testing and UI Verification

- Login was tested with users in both 'admin' and 'user' role.
- Conditional UI was verified to only show Post Job and View Applications for 'admin' roles only.
- Success/Failure messages are shown directly on the application.
- Salary formatting was handled dynamically based on user input.

Results



Job Board

Job Listings

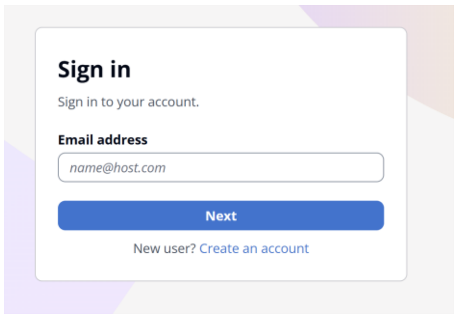
Login to begin applying.

Software Engineer - Intern
Type: Part-Time | Salary: $25 per hour

Data Engineer - Senior
Type: Full-Time | Salary: $100,000 per year

Cloud Engineer - Senior
Type: Full-Time | Salary: $150,000 per year

Fig. 10 – Upon loading application, users must login to apply for jobs



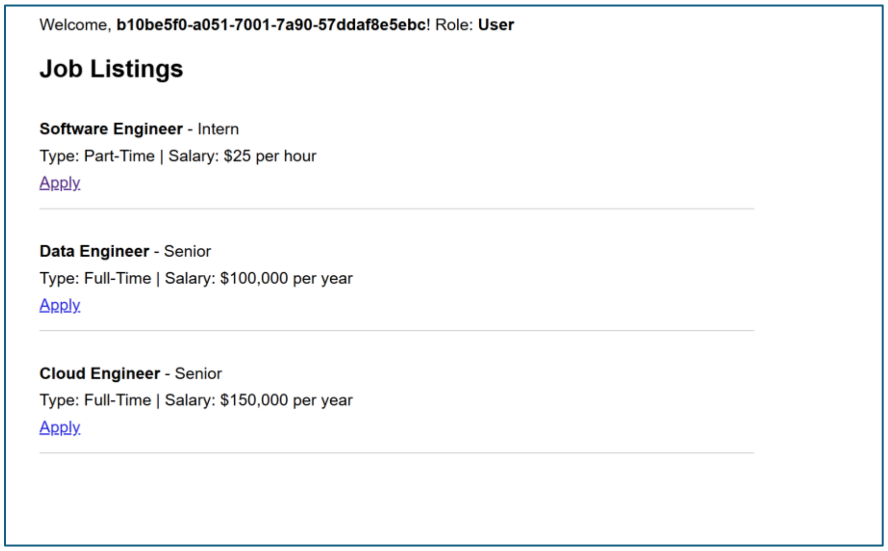
Sign in

Sign in to your account.

Email address

New user? Create an account

Fig. 11 – Upon selecting 'Login' the user is redirected to AWS Cognito to Sign In or Sign Up if needed.



Welcome, **b10be5f0-a051-7001-7a90-57ddaf8e5ebc**! Role: **User**

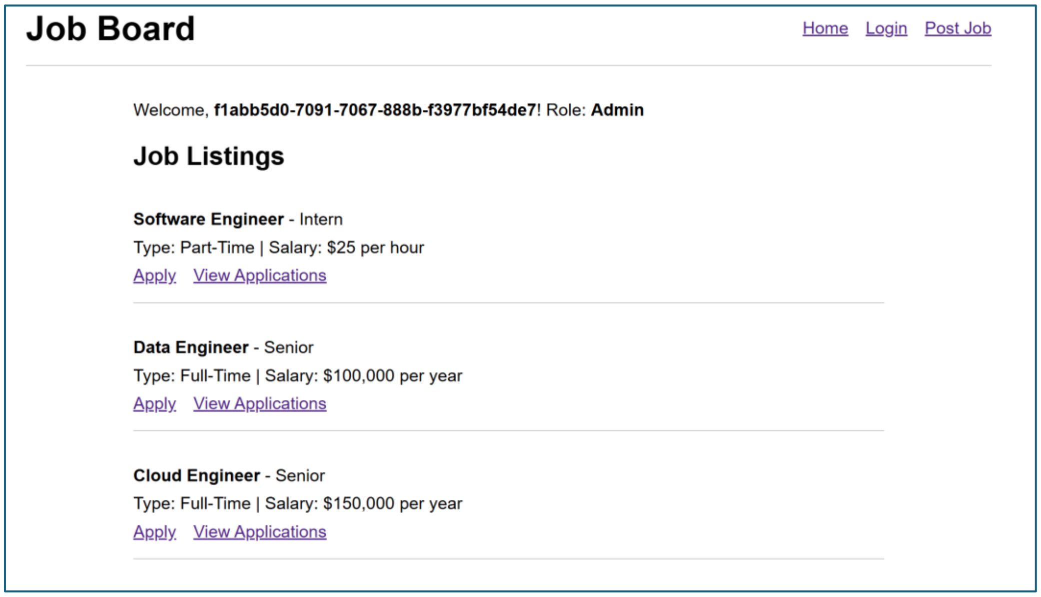
Job Listings

Software Engineer - Intern
Type: Part-Time | Salary: $25 per hour
Apply

Data Engineer - Senior
Type: Full-Time | Salary: $100,000 per year
Apply

Cloud Engineer - Senior
Type: Full-Time | Salary: $150,000 per year
Apply

Fig. 12 – Upon Successful login, a role will be given based on AWS Cognito of either 'admin' or 'user'. Users are limited to viewing the Job Listings and being able to Apply to those listings.



Job Board

Home Login Post Job

Welcome, **f1abb5d0-7091-7067-888b-f3977bf54de7**! Role: **Admin**

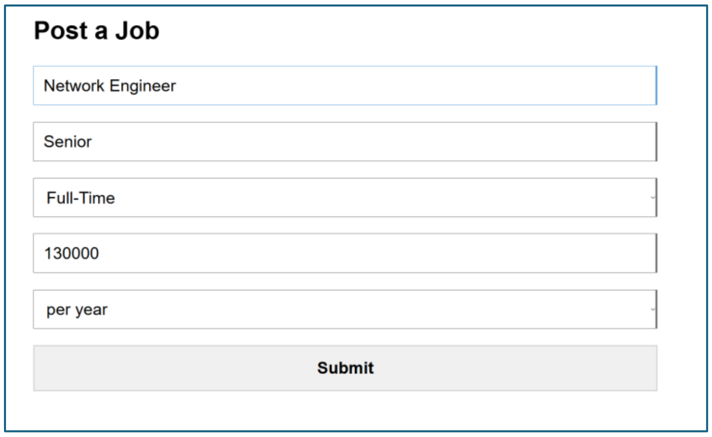
Job Listings

Software Engineer - Intern
Type: Part-Time | Salary: $25 per hour
Apply View Applications

Data Engineer - Senior
Type: Full-Time | Salary: $100,000 per year
Apply View Applications

Cloud Engineer - Senior
Type: Full-Time | Salary: $150,000 per year
Apply View Applications

Fig. 13 – The Admin UI is able to Post Job's on the top right page, and View Applications for job postings.


Post a Job

Network Engineer

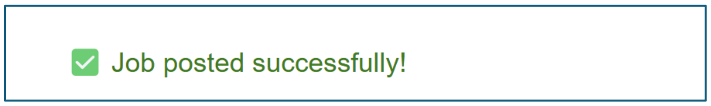
Senior

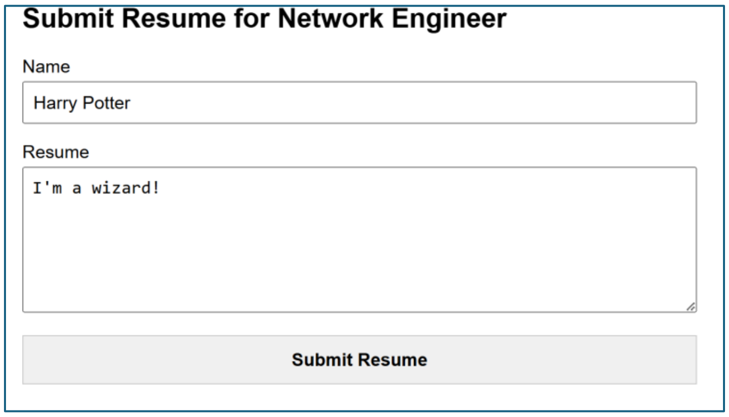
Full-Time

130000

per year

Fig. 14 – This is the Job Posting page, upon submission, the webpage will display if the submission was successful or not.

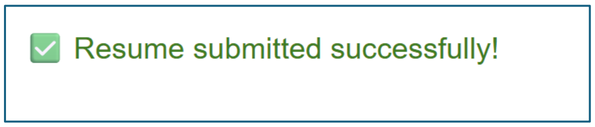

✅ Job posted successfully!

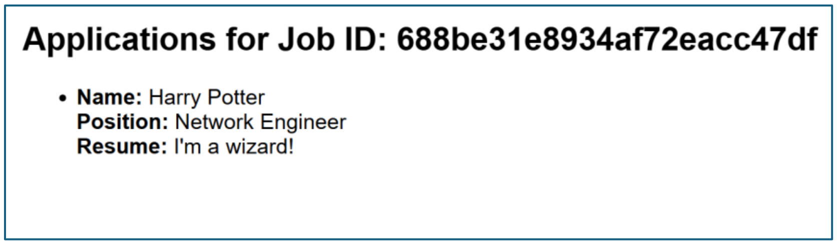

Submit Resume for Network Engineer

Name: Harry Potter

Resume: I'm a wizard!

Fig. 15 – users can apply for these job listings where they will fill in their names and input their resume. Similarly to posting a job, once a resume is submitted, the user will be promoted to a successful or failed submission.


✅ Resume submitted successfully!



Applications for Job ID: 688be31e8934af72eacc47df

- **Name:** Harry Potter
  Position: Network Engineer
  Resume: I'm a wizard!

Fig. 16 – Admin's can then view applications for a given job posting.

Conclusion and Reflection

The frontend was successfully integrated with AWS Cognito and Node.js/Express backend, following MERN stack principles. It delivers role-based access, dynamic UI, and interacts clearly with a cloud-hosted backend.

Challenges Faced

- Token exchange and decoding from Cognito required understanding OAuth2 flow.
- Ensuring HTTPS compatibility using Ngrok was necessary for successful redirection with AWS Cognito.
- Handling conditional rendering for user roles required token parsing and error handling.
- Keeping component logic organized while expanding frontend functionality was difficult as some variable names were consistent, resulting in unforeseen errors.