NextHire

Your one-step solution for job hunting and recruitment

Slok Tulsyan

10 December 2025

To get started with the project, first create two main folders -

1. Client (for frontend)
2. Services (for different backend services)

Auth Service

Now, create an **auth** folder inside the services folder to get started with the auth service. In this directory, run below given commands:

```
npm init -y
npm i -g typescript
npx tsc –init
npm i express dotenv bcrypt jsonwebtoken multer axios
datauri kafkajs redis cors
npm i -D @types/express @types/dotenv @types/bcrypt
@types/jsonwebtoken @types/multer @types/kafkajs @types/
redis @types/cors
npm i -D typescript nodemon
```

Modify the given keys in the `tsconfig.json` file:

```
"rootDir": "./src",
"outDir": "./dist",
"module": "nodenext",
"target": "es2020",
"esModuleInterop": true,
"forceConsistentCasingInFileNames": true,
"strict": true,
"skipLibCheck": true,
```

Now, start with creating given files with `src` folder:

1. index.ts
2. app.ts
3. .env

Now, add the below scripts in the `package.json` file:

```
"build": "npm install && tsc",
"start": "node dist/index.js",
"dev": "concurrently \"tsc -w\" \"nodemon dist/
index.js\"",
```

After creating these files, run `npm run dev` to build `dist` folder and run the **auth** service.

Connecting Database

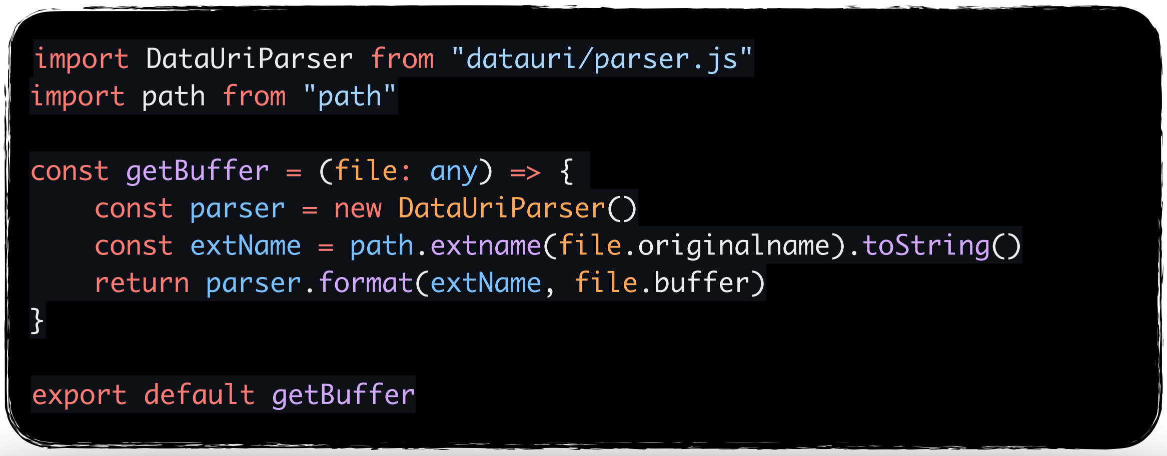
First, create a new project in the `Neon` and then get the connection URL from there and write it in the `.env` file. After that, run the given command for installation:

```
npm i @neondatabase/serverless
```

Now, create a `db.ts` file within `utils` folder to connect with the database. After that create a function to initialize database and write `sql` queries for creating `user_role`, `users`, `skills` and `user_skills` tables in the `index.ts` file.

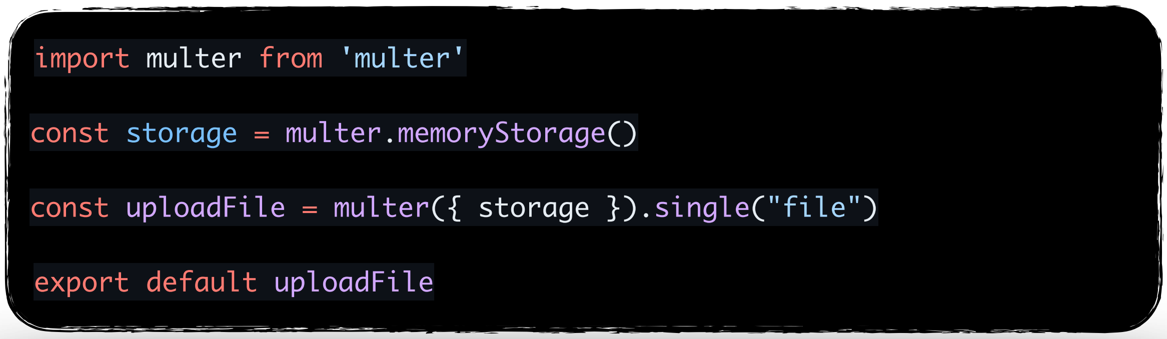
Setting up Multer

To store files like resume of job seekers, we need to setup `multer.` First, we need to set up `datauri` for getting buffer of a file using the following function:



```
import DataUriParser from "datauri/parser.js"
import path from "path"

const getBuffer = (file: any) => {
    const parser = new DataUriParser()
    const extName = path.extname(file.originalname).toString()
    return parser.format(extName, file.buffer)
}

export default getBuffer
```

Set up multer using the following code:



```
import multer from 'multer'

const storage = multer.memoryStorage()

const uploadFile = multer({ storage }).single("file")

export default uploadFile
```

Creating Endpoints

Moving ahead, create `routes` and `controllers` folders along with common **error handling** and **try-catch** functions for optimized code.

Now, start creating endpoints for authentication service:

1. **Register User: (POST)**

Create an endpoint for registering new users according to their role. As we know, there are two roles - `recruiter` and `jobseeker`. It will take `uploadFile` function as middleware to upload resume and profile picture and call the **utils** service for uploading them to the cloudinary.

```
http://localhost:5000/api/auth/register
```

After registering the user, generate a `jwt-token` for future validation while login the user.

2. Login User: (POST)

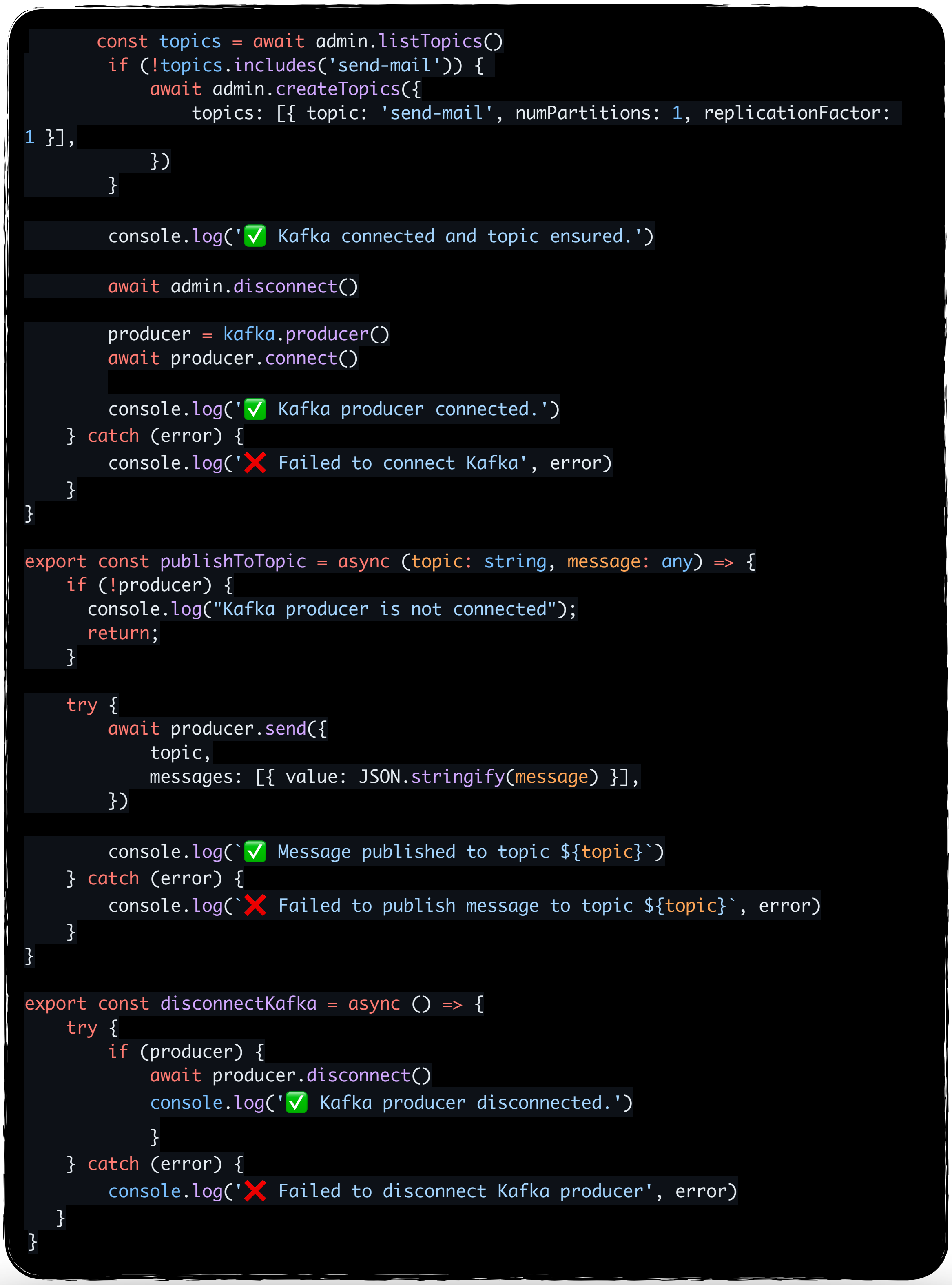
Now create an endpoint for login user using their email and password. While login, we have to `join` all three tables to get the list of skills user have.

```
http://localhost:5000/api/auth/login
```

3. Forget Password: (POST)

First, create a Kafka producer that connects with the mail-service under the **utils service** as below given:

```
import { Kafka, Producer, Admin } from 'kafkajs'
import dotenv from 'dotenv'

dotenv.config()

let producer: Producer
let admin: Admin

export const connectKafka = async () => {
    try {
        const kafka = new Kafka({
            clientId: 'auth-service',
            brokers: [process.env.KAFKA_BROKER || 'localhost:9092'],
        })

        admin = kafka.admin()

        await admin.connect()
```

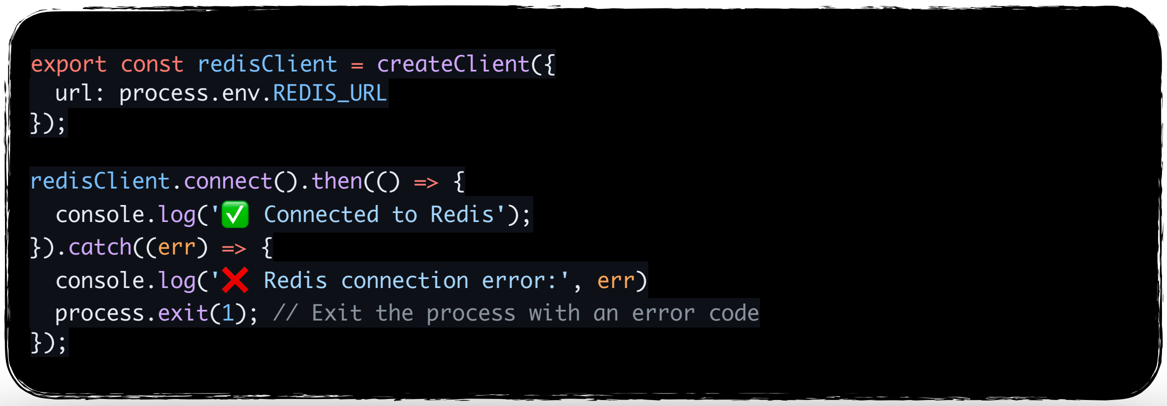


```
        const topics = await admin.listTopics()
        if (!topics.includes('send-mail')) {
            await admin.createTopics({
                topics: [{ topic: 'send-mail', numPartitions: 1, replicationFactor:
1 }],
            })
        }

        console.log('✅ Kafka connected and topic ensured.')

        await admin.disconnect()

        producer = kafka.producer()
        await producer.connect()

        console.log('✅ Kafka producer connected.')
    } catch (error) {
        console.log('❌ Failed to connect Kafka', error)
    }
}

export const publishToTopic = async (topic: string, message: any) => {
    if (!producer) {
      console.log("Kafka producer is not connected");
      return;
    }

    try {
        await producer.send({
            topic,
            messages: [{ value: JSON.stringify(message) }],
        })

        console.log(`✅ Message published to topic ${topic}`)
    } catch (error) {
        console.log(`❌ Failed to publish message to topic ${topic}`, error)
    }
}

export const disconnectKafka = async () => {
    try {
        if (producer) {
            await producer.disconnect()
            console.log('✅ Kafka producer disconnected.')
            }
    } catch (error) {
        console.log('❌ Failed to disconnect Kafka producer', error)
   }
}
```

After that, create an endpoint forgot password for getting the reset link on email. Along with this, create an HTML template for reset email.

http://localhost:5000/api/auth/forgot-password

Setup upstash for using `redis`. Create `redisClient` first and then use it to save the reset token in the `forgotPassword` function.



```
export const redisClient = createClient({
  url: process.env.REDIS_URL
});

redisClient.connect().then(() => {
  console.log('✅ Connected to Redis');
}).catch((err) => {
  console.log('❌ Redis connection error:', err)
  process.exit(1); // Exit the process with an error code
});
```

4. Reset Password: (POST)

Now, create an endpoint for reseting password using the token generated and sent on the email. For this, use `redisClient` to verify the stored token with the sent token.

http://localhost:5000/api/auth/reset-password/:token

Utils Service

Create a **utils** folder inside the services folder to get started with the utils service. Start with basic installation of some required packages using the given commands:

```
npm init -y
npm i express dotenv cors cloudinary
npm i -D @types/express @types/dotenv @types/cors
npm i -D typescript nodemon
npm i -D concurrently
```

Similar to the auth service, create a root folder named `src`, after that create `index.ts` and `routes.ts` files. Also, setup the `package.json` file for scripts as done in the auth service.

Setting up Cloudinary

Start setting up cloudinary by creating upload service using `buffer`. The following code can be used for this:

```
import express from 'express'
import cloudinary from 'cloudinary'

const router = express.Router()

router.post("/upload", async(req, res) => {
    try {
        const { buffer, public_id } = req.body
        if (public_id) {
            await cloudinary.v2.uploader.destroy(public_id)
        }
        const cloud = await cloudinary.v2.uploader.upload(buffer)
        res.json({
            url: cloud.secure_url,
            public_id: cloud.public_id
        })
    } catch (error: any) {
        res.status(500).json({ message: error.message })
    }
})

export default router
```

Setting up Kafka (Email Service)

User will receive an email with the reset link during forget password setup and this should be done in background so that it won't delay the whole website. Hence, we will use **kafka** here. For that, first install docker desktop and pull the Kafka image using the following command:

```
docker pull apache/kafka:4.1.0
```

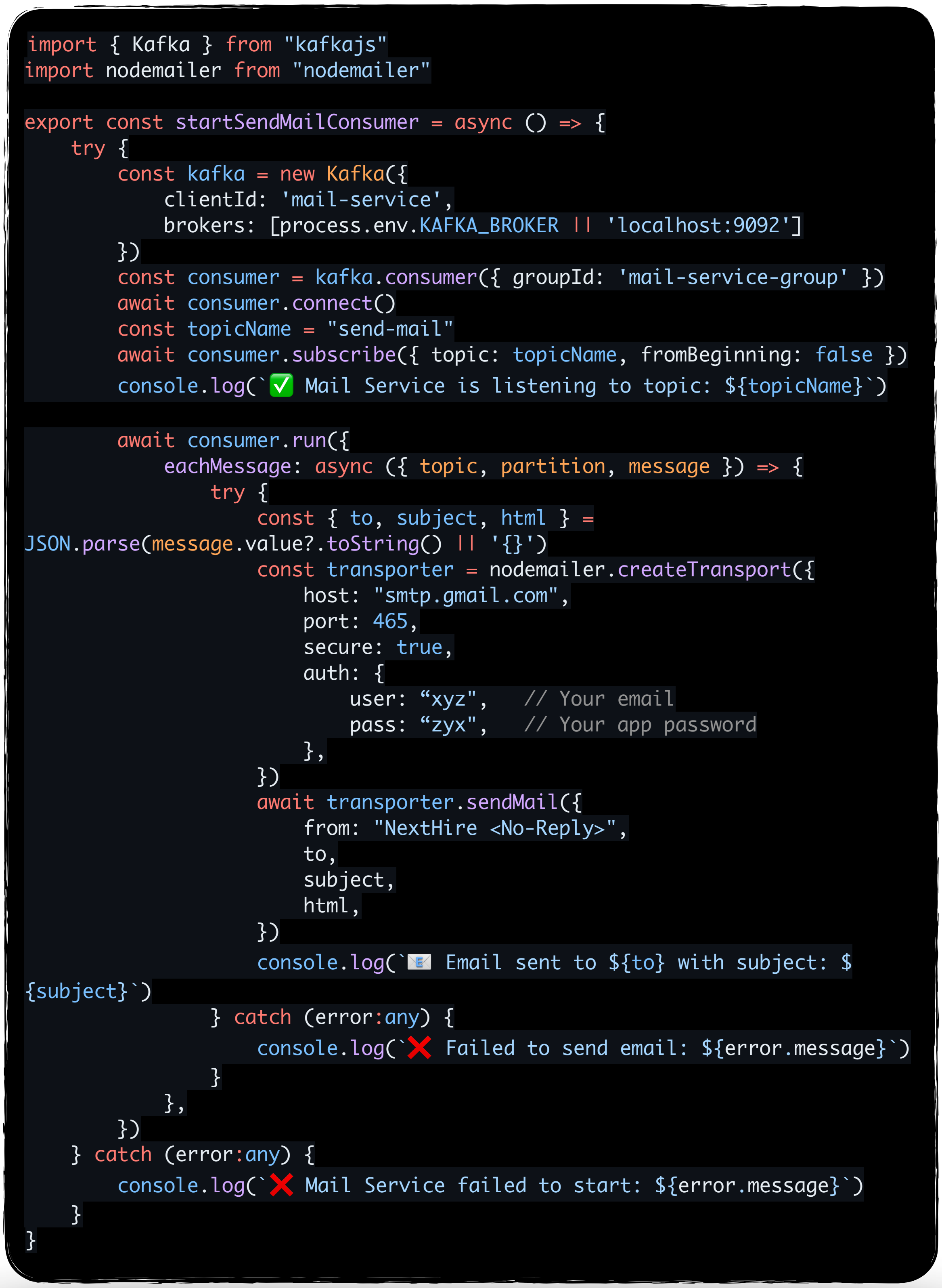
After this, we need to start a docker container which runs Kafka. Do this using the given command:

```
docker run -d --name kafka -p 9092:9092 -e
KAFKA_NODE_ID=1 -e KAFKA_PROCESS_ROLES=broker,controller
-e
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,
CONTROLLER:PLAINTEXT -e
KAFKA_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093 -e
KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092 -e
KAFKA_CONTROLLER_LISTENER_NAMES=CONTROLLER -e
KAFKA_CONTROLLER_QUORUM_VOTERS=1@localhost:9093 -e
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1 -e
KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS=0 apache/
kafka:4.1.0
```

It will return a container id, showing that a container is running kafka inside it. After that, install some packages:

```
npm i nodemailer kafkajs
npm i -D @types/nodemailer @types/kafkajs
```

Now, start creating consumer using kafka and implement mail-service using node mailer and gmail.



```
import { Kafka } from "kafkajs"
import nodemailer from "nodemailer"

export const startSendMailConsumer = async () => {
    try {
        const kafka = new Kafka({
            clientId: 'mail-service',
            brokers: [process.env.KAFKA_BROKER || 'localhost:9092']
        })
        const consumer = kafka.consumer({ groupId: 'mail-service-group' })
        await consumer.connect()
        const topicName = "send-mail"
        await consumer.subscribe({ topic: topicName, fromBeginning: false })
        console.log(`✅ Mail Service is listening to topic: ${topicName}`)

        await consumer.run({
            eachMessage: async ({ topic, partition, message }) => {
                try {
                    const { to, subject, html } =
JSON.parse(message.value?.toString() || '{}')
                    const transporter = nodemailer.createTransport({
                        host: "smtp.gmail.com",
                        port: 465,
                        secure: true,
                        auth: {
                            user: “xyz",    // Your email
                            pass: “zyx",    // Your app password
                        },
                    })
                    await transporter.sendMail({
                        from: "NextHire <No-Reply>",
                        to,
                        subject,
                        html,
                    })
                    console.log(`📧 Email sent to ${to} with subject: $
{subject}`)
                } catch (error:any) {
                    console.log(`❌ Failed to send email: ${error.message}`)
                }
            },
        })
    } catch (error:any) {
        console.log(`❌ Mail Service failed to start: ${error.message}`)
    }
}
```

Setting up Gemini

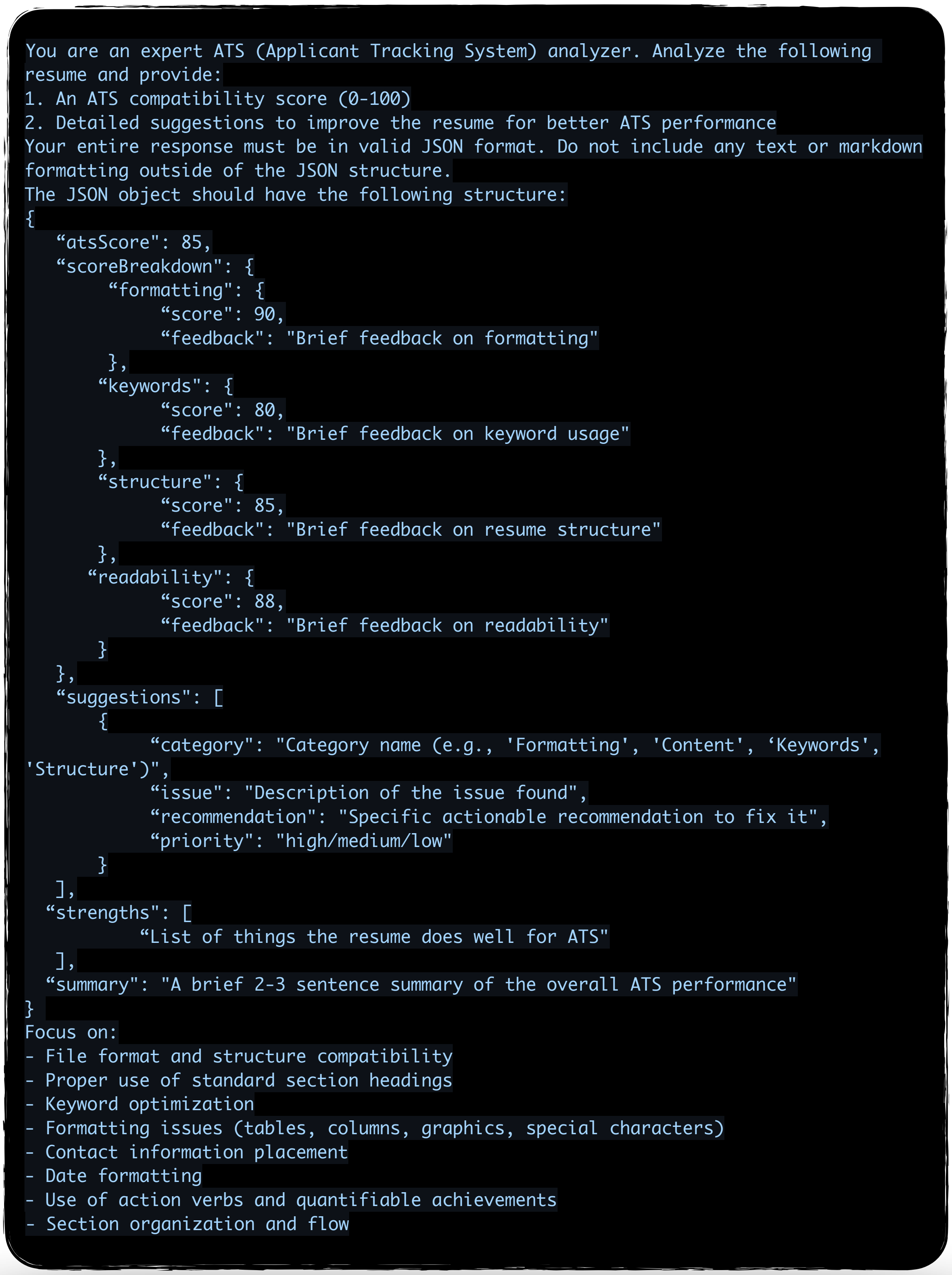
Now, start implementing AI features using **Google Gemini**. For that first create an API key of gemini and paste it in the `.env` file. After that install some packages to use it:

```
npm i @google/generative-ai
```

Now, create an endpoint for getting career analysis and use this prompt to get result in the fixed pattern from AI:

```
Based on the following skills: ${skills}.
Please act as a career advisor and generate a career path suggestion.
Your entire response must be in a valid JSON format. Do not include any text or
markdown formatting outside of the JSON structure.
The JSON object should have the following structure:
{
  “summary": "A brief, encouraging summary of the user's skill set and their general
job title.",
  “jobOptions": [
     {
        “title": "The name of the job role.",
        “responsibilities": "A description of what the user would do in this role.",
        “why": "An explanation of why this role is a good fit for their skills."
      }
   ],
  “skillsToLearn": [
      {
        “category": "A general category for skill improvement (e.g., 'Deepen Your
Existing Stack Mastery', 'DevOps & Cloud').",
        “skills": [
           {
              “title": "The name of the skill to learn.",
              “why": "Why learning this skill is important.",
              “how": "Specific examples of how to learn or apply this skill."
            }
         ]
      }
   ],
  “learningApproach": {
       “title": "How to Approach Learning",
       “points": ["A bullet point list of actionable advice for learning."]
   }
}
```

Similarly, create an endpoint for resume analysis and use this prompt to get result in the fixed pattern from AI:



```
You are an expert ATS (Applicant Tracking System) analyzer. Analyze the following
resume and provide:
1. An ATS compatibility score (0-100)
2. Detailed suggestions to improve the resume for better ATS performance
Your entire response must be in valid JSON format. Do not include any text or markdown
formatting outside of the JSON structure.
The JSON object should have the following structure:
{
   “atsScore": 85,
   “scoreBreakdown": {
        “formatting": {
             “score": 90,
             “feedback": "Brief feedback on formatting"
        },
       “keywords": {
             “score": 80,
             “feedback": "Brief feedback on keyword usage"
       },
       “structure": {
             “score": 85,
             “feedback": "Brief feedback on resume structure"
       },
      “readability": {
             “score": 88,
             “feedback": "Brief feedback on readability"
       }
   },
   “suggestions": [
       {
            “category": "Category name (e.g., 'Formatting', 'Content', ‘Keywords',
'Structure')",
            “issue": "Description of the issue found",
            “recommendation": "Specific actionable recommendation to fix it",
            “priority": "high/medium/low"
       }
   ],
  “strengths": [
          “List of things the resume does well for ATS"
   ],
  “summary": "A brief 2-3 sentence summary of the overall ATS performance"
}
Focus on:
- File format and structure compatibility
- Proper use of standard section headings
- Keyword optimization
- Formatting issues (tables, columns, graphics, special characters)
- Contact information placement
- Date formatting
- Use of action verbs and quantifiable achievements
- Section organization and flow
```

User Service

Create a **user** folder inside the services folder to get started with the user service. Start with basic installation of some required packages using the given commands:

```
npm init -y
npm i express dotenv cors jsonwebtoken @neondatabase/
serverless axios multer kafka
npm i -D @types/express @types/dotenv @types/cors @types/
jsonwebtoken @types/multer @types/kafka
npm i -D typescript concurrently nodemon
```

Similar to the auth service, create a root folder named `src`, after that create `index.ts` and `routes.ts` files. Also, setup the `package.json` file for scripts as done in the auth service.

Creating Endpoints

Moving ahead, create `routes` and `controllers` folders along with common **error handling** and **try-catch** functions for optimized code. Further more, create a middleware to get the user data from the auth token.

```
export const isAuth = async (req: AuthenticatedRequest, res: Response, next:
NextFunction): Promise<void> => {
    try {
        const authHeader = req.headers.authorization;
        if (!authHeader || !authHeader.startsWith('Bearer ')) {
            res.status(401).json({ message: 'Unauthorized' });
            return;
        }

        const token = authHeader.split(' ')[1];
        const decodedToken = jwt.verify(token, process.env.JWT_SECRET as string) as
JwtPayload;

        if(!decodedToken || !decodedToken.user_id) {
            res.status(401).json({ message: 'Invalid token' });
            return;
        }

        const users = await sql`SELECT u.user_id, u.name, u.email, u.phone_number,
u.role, u.bio, u.resume, u.profile_pic, u.subscription, ARRAY_AGG(s.name) FILTER(WHERE
s.name IS NOT NULL) as skills FROM users u LEFT JOIN user_skills us ON u.user_id =
us.user_id LEFT JOIN skills s ON us.skill_id = s.skill_id WHERE u.user_id = $
{decodedToken.user_id} GROUP BY u.user_id`;

        if (users.length === 0) {
            res.status(401).json({ message: 'User assosciated with this token no
longer exists' });
            return;
        }

        const user = users[0] as User;
        user.skills = user.skills || []
        req.user = user;
        next();

    } catch (error) {
        res.status(401).json({ message: 'Authentication failed. Please login again',
error: (error as Error).message });
    }
}
```

Now, start creating endpoints for user service:

1. **My Profile: (GET)**

Create an endpoint to fetch the data of the logged in user using the authenticated token via middleware.

```
http://localhost:5002/api/user/me
```

2. **User Profile: (GET)**

Now, create an endpoint to fetch the data of any other user using it’s `userId`.

```
http://localhost:5002/api/user/:userId
```

3. Update Profile: (PUT)

Now, create an endpoint to update the user data including the `name`, `phone_number` and `bio`.

```
http://localhost:5002/api/user/update-profile
```

4. Update Profile Picture: (PUT)

Now, create an endpoint to update the user's profile picture by deleting the previous one (if exists) and adding new one.

```
http://localhost:5002/api/user/update-profile-picture
```

5. Update Resume: (PUT)

Now, create an endpoint to update the user's resume by deleting the previous one (if exists) and adding new one.

```
http://localhost:5002/api/user/update-resume
```

6. Add Skills: (POST)

Now, create an endpoint for adding new skills in the user's profile. It should first check the existence of the skill in the `skills` table and then create new one.

```
http://localhost:5002/api/user/skill/add
```

7. Remove Skills: (DELETE)

Now, create an endpoint for removing skills from the user's profile.

```
http://localhost:5002/api/user/skill/remove
```

8. Get All Skills: (GET)

Now, create an endpoint for fetching all the existing skills so that user can select from them or add new one if not exists.

```
http://localhost:5002/api/user/skill
```

9. Search Skills: (GET)

Now, create an endpoint for searching skills among the existing one.

```
http://localhost:5002/api/user/skill/search
```

```
 const { query } = req.query;

const searchTerm = query.trim();

const skills = await sql`
    SELECT skill_id, name
    FROM skills
    WHERE LOWER(name) LIKE LOWER(${"%" + searchTerm + "%"})
    ORDER BY
      CASE
        WHEN LOWER(name) = LOWER(${searchTerm}) THEN 1
        WHEN LOWER(name) LIKE LOWER(${searchTerm + "%"}) THEN 2
        ELSE 3
      END,
      name
    LIMIT 10
  `;
```

10. Apply for Jobs: (POST)

Now, create an endpoint for applying for new jobs ensuring no repeated applications after the **job service** is created along with sending email for confirmation.

```
http://localhost:5002/api/user/apply/:jobId
```

11. Get My Applications: (GET)

Now, create an endpoint for fetching all the applications along with the job details.

```
http://localhost:5002/api/user/applications/me
```

Job Service

Create a **job** folder inside the services folder to get started with the job service. Start with basic installation of some required packages using the given commands:

```
npm init -y
npm i express dotenv cors jsonwebtoken @neondatabase/
serverless datauri multer axios kafkajs
npm i -D @types/express @types/dotenv @types/cors @types/
jsonwebtoken @types/multer @types/kafkajs
npm i -D typescript concurrently nodemon
```

Similar to the auth service, create a root folder named `src`, after that create `index.ts` and `routes.ts` files. Also, setup the `package.json` file for scripts as done in the auth service.

Connecting Database

Create a function to initialize database and write `sql` queries for creating `companies`, `jobs` and `applications` tables in the `index.ts` file.

Creating Endpoints

Moving ahead, create `routes` and `controllers` folders along with common **error handling** and **try-catch** functions for optimized code. Further more, create a middleware to get the user data from the auth token.

Now, start creating endpoints for user service:

1. **Create Company: (POST)**

Create an endpoint to add new company via authenticated recruiter.

```
http://localhost:5003/api/jobs/company/new
```

2. **Delete Company: (DELETE)**

Create an endpoint to delete any company along with deleting the jobs and applications associated with it.

```
http://localhost:5003/api/jobs/company/:companyId
```

3. **Create Job: (POST)**

Create an endpoint to add new job openings with all the required details.

```
http://localhost:5003/api/jobs/new
```

4. **Update Job: (PUT)**

Create an endpoint to update the existing jobs having an extra field `is_active`.

```
http://localhost:5003/api/jobs/:jobId
```

5. **Get Company By Recruiter: (GET)**

Create an endpoint to get all the companies by `recruiter_id` as one recruiter can add almost 3 companies.

```
http://localhost:5003/api/jobs/company/by-recruiter
```

6. **Get Company Details: (GET)**

Create an endpoint to get all the details of the company along with the jobs offered by that company.

```
http://localhost:5003/api/jobs/company/:companyId
```

7. Get All Jobs: (GET)

Create an endpoint to get all the available jobs based on filter with pagination.

```
http://localhost:5003/api/jobs
```

8. Get Applications for Jobs: (GET)

Create an endpoint to get all the applications for each job using `job_id`.

```
http://localhost:5003/api/jobs/applications/:jobId
```

9. Update Application Status: (PUT)

Create an endpoint to update the status of the application as `hired` or `rejected` and send the email regarding that to the applier.

```
http://localhost:5003/api/jobs/application/applicationId
```

10.Get Job Details: (GET)

Create an endpoint to fetch the details of a job using its `job_id`.

```
http://localhost:5003/api/jobs/:jobId
```

Client

Start setting up `Next.js` and UI libraries using the following commands:

```
npx create-next-app@latest .
npx shadcn@latest init
npm i next-themes axios react-hot-toast js-cookie
npm i @types/js-cookie
npx shadcn@latest add button popover avatar dropdown-menu
dialog label input card textarea select separator switch
checkbox alert tabs
```

Now, start with the home page designing. For that, create three components:

1. Hero Component
2. Career Guidance Component
3. Resume Analyser Component

After that design the about page as it is very much simpler.

Now, start with creating the `context provider` that will allow using `user`, `loading`, `isAuth`, `btnLoading` globally. Go ahead with the creation of login and register page.

Now, create my profile page and user profile page which will have all the details of the user including resume and profile picture which should be editable. Also, there should a skill section (only for job seekers) which allows user to add or remove skills. Also, there should be a companies section (only for recruiters). For each company, there will be a page where all the jobs of that company will be listed and for each job, there will be a page where all the applications will be listed. There will be a page for displaying all the available jobs with a filter panel.

Deployment

Go to the auth service and create `Dockerfile` & .dockerignore files.

```
FROM node:22-alpine AS builder

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install

COPY tsconfig.json ./
COPY src ./src

RUN npm run build

FROM node:22-alpine

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install --only=production

COPY --from=builder /app/dist ./dist

CMD ["node", "dist/index.js"]
```

Paste these two files in all other services as well. After that, run the below commands in each service directory:

```
docker login
docker build -t <service-name> .
docker tag <service-name> <docker-username>/<service-
name>:latest
docker push <docker-username>/<service-name>:latest
```

You can try running the docker image in your local machine using the `.env` file:

```
docker run —env-file .env <service-name>
```

Now, similarly create both files in the frontend (client).

```
FROM node:22-alpine AS deps

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install

FROM node:22-alpine AS builder

WORKDIR /app

COPY --from=deps /app/node_modules ./node_modules
COPY . .

RUN npm run build

FROM node:22-alpine AS runner

WORKDIR /app
ENV NODE_ENV=production

COPY --from=builder /app/public ./public
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json

EXPOSE 3000

CMD ["npm", "start"]
```

```
docker build -t client .
docker tag client <docker-username>/client:latest
docker push <docker-username>/client:latest
```

Now, get the public IPv4 from the **AWS** update all the env files as localhost -> <your IPv4>. Then, edit inbound rules in the AWS and add some new ports 9092, 9093, 5000, 5001, 5002, 5003 & 3000. After that connect with your AWS instance using **ssh** and your public IPv4.

After connecting successfully, create five directories - four for the four services and one for the client. Now, go to each service folder one by one, copy all the environment variables and run the following commands:

```
cd <service-name>
docker pull <docker-username>/<service-name>:latest
vim .env
docker run -d --name <service-name> --env-file .env -p
<service-port>:<service-port> --restart=always <docker-
username>/<service-name>:latest
```

Now, you can visit **http://<your-public-ip>:3000** to see your working and deployed web app.