Kenshi Resumes: AI-Powered Resume Builder Documentation

- **Project:** Kenshi Resumes (Frontend & Backend)
- **Audience:** Job seekers and general users, Developers, and Recruiters
- **License:** MIT (see *License & Credits*)

Table of Contents

Project Overview

Kenshi Resumes is an **AI-powered resume builder** designed to help **job seekers** create professional, ATS-friendly resumes quickly and easily. Leveraging Google's cutting-edge **Gemini Flash 2.0** language model API, the application generates tailored resume content in seconds, while offering a live editing interface for users to customize their resume in real time. The project comprises a modern **React** frontend and a **Node.js/Express** backend with a **PostgreSQL** database, making it developer-friendly and scalable.

Purpose & Value: The main goal is to streamline resume creation: job applicants can input basic details (name, email, job title, etc.) and have high-quality resume sections generated by AI. It also provides **ATS (Applicant Tracking System) scoring and recommendations**, helping applicants optimize resumes to pass automated screening. Recruiters and HR professionals benefit indirectly: candidates using Kenshi Resumes will likely submit more polished, keyword-optimized resumes, improving the quality of applications they receive. The Telegram bot integration allows convenient resume delivery via chat.

Intended Users:

- **Job Seekers / End Users:** Those looking to build or improve a resume with AI assistance, using an intuitive web editor and optional ATS feedback.
- **Recruiters / Employers:** Although not direct users of the app, recruiters are addressed through features like ATS scoring (explained below) that ensure incoming resumes meet screening criteria. They can also share Kenshi Resumes with candidates to accelerate application processing.
- **Developers:** Tech-savvy contributors interested in extending or integrating the system. The codebase uses popular technologies (React, Node, Tailwind, PostgreSQL) and includes clear setup instructions (see *Setup & Installation*).

Key Features & Use Cases

- **AI-Generated Content:** Uses Google Gemini Flash 2.0 to write high-quality resume sections (summary, skills, experience, etc.) based on user input. This saves time and brings expertise to resume writing.
- **Live Resume Editor:** A React-powered interface allows users to edit their resume in real time with instant preview. Changes are reflected immediately, making customization easy.
- **Mobile-Responsive Design:** The frontend is built with React and Tailwind CSS, ensuring a clean, responsive layout on all devices (desktops, tablets, phones).
- **Secure & Private:** No third-party sharing of data – all resume data is stored in the user's own database (PostgreSQL) via the backend. As the README notes, *"Your data stays safe and confidential."*.
- **ATS Checker & Recommendations:** Kenshi Resumes analyzes the resume with ATS in mind. It provides an **ATS score** and suggestions (via the Flash Recommendations API) to improve keyword usage and formatting. An ATS (Applicant Tracking System) is software that *"scans resumes for essentials like contact info, job titles, and keywords"* and filters applications. By highlighting missing keywords or sections, Kenshi Resumes helps job seekers craft resumes that are more likely to pass ATS filtering.
- **Telegram Bot:** An optional companion bot (Kenshi Resumes Bot) is available on Telegram. Users can send their resume to this bot to receive it via chat. The backend includes routes for Telegram integration, so a user can get their generated resume delivered through the messaging app.

- **Resume Storage & Retrieval:** Every generated resume is saved in the database. Users can retrieve previously generated resumes by querying their email, allowing them to save and reuse work.

Example Use Case (Job Seeker): Jane enters her name, email, and a desired job title. Kenshi Resumes calls the AI API to generate sections like a summary and skills. Jane edits the content in the live editor and sees how it will look in the final PDF. She then checks the ATS score; the app suggests adding a few more keywords related to her field to improve the score. Finally, she saves her resume and optionally sends it to her Telegram account via the bot for easy access.

Example Use Case (Recruiter): A recruiter shares Kenshi Resumes with candidates as a recommended tool. Candidates craft resumes tailored to the job description, increasing their fit. The recruiter can trust that submissions generated this way are ATS-optimized, reducing manual filtering. Additionally, recruiters can quickly view recommendations provided by the ATS checker to understand how candidates' resumes were improved.

Tech Stack

- **Frontend:** React (Vite) for building the interactive resume editor and UI. React is a popular open-source library for creating user interfaces. Vite is used for fast development builds. Tailwind CSS provides utility-first styling, ensuring a polished design and responsive layout.
- **Backend:** Node.js and Express.js. Node.js is an open-source, cross-platform JavaScript runtime for server-side code. Express.js is a minimal and flexible web framework for Node.js that simplifies building APIs and web services.
- **AI Engine:** Google **Gemini Flash 2.0** API (via the AI Engine layer) for generating resume text and analyzing content. Gemini is Google's latest state-of-the-art multimodal language model (released March 2025) with advanced reasoning abilities. Kenshi Resumes uses Gemini Flash 2.0 to power its AI content creation.
- **Database:** PostgreSQL – a powerful, open-source relational database. It stores user resume records, user info (email, name), and any generated content or metadata.
- **Styling:** Tailwind CSS for design and layout (utility classes enable rapid UI development).
- **Deployment Platforms:** Frontend is designed to be deployed on Vercel (a cloud platform for frontend hosting and deployment). The backend is deployed on Render (a cloud platform for hosting and deploying web services). These platforms offer seamless CI/CD: pushing code to GitHub can trigger automated deployments.

The tech stack balances modern development speed with stability. (E.g., React and Express are widely used in industry, while Gemini provides cutting-edge AI capability.) The diagram below outlines the high-level architecture:

- *User (browser)* → React UI (calls) → Express API → Gemini AI & PostgreSQL → responses to UI.

Architecture & Folder Structure

This section outlines the high-level organization of the code for both frontend and backend.

Frontend Structure

```
kenshi-resumes/
├── src/               # React source files (components, pages, utilities)
├── public/            # Static files (index.html, images, favicon)
├── server/            # Legacy Strapi Backend
├── .env               # Environment variables config (API keys, endpoints)
├── package.json       # Project metadata and scripts
└── README.md          # Documentation
```

- `src/` contains the React application code. Components like the resume editor, form inputs, and preview are implemented here.
- `public/` holds static assets and the base `index.html`.
- `.env` holds configuration values (e.g. `VITE_API_URL` for the backend API, `GEMINI_API_KEY`).
- **Routing and Interactions:** The app likely has a main `App` component that renders pages (e.g., a Home/Editor page). Components communicate via React state or context. On user actions (e.g., "Generate Resume" button), the frontend sends HTTP requests to the backend API. The editor updates the view in real time.
- **Deprecated `server/` folder:** The `README.md` mentions a `server/` directory for a Strapi backend, but notes the current backend is a separate Node.js service (the **Kenshi-Resumes-Backend** repo). In practice, the frontend's `server/` directory is legacy and not used in the new architecture.

Backend Structure

```
kenshi-resumes-backend/
├── index.js               # Express server entry point
├── flashAI.js            # AI-driven resume generation logic
├── flashRecommendations.js # ATS recommendations logic
├── bot.js                # Telegram bot integration
├── sharedData.js         # Shared constants/helpers
├── queries.sql           # SQL script to create needed tables
├── package.json          # Metadata and scripts
├── steps.txt             # Development & deployment steps (notes)
└── README.md             # Project documentation
```

- `index.js`: Sets up the Express server, routes, middleware (e.g., JSON parsing), and database connection. Defines API endpoints (as documented below).
- `flashAI.js`: Contains functions to call the Gemini API and process the input to generate resume content.
- `flashRecommendations.js`: Implements the logic for ATS scoring and suggestions. It likely parses the resume or uses an AI prompt to identify missing elements.
- `bot.js`: Handles the Telegram bot: receiving commands/requests, fetching resumes from the database, and sending files to the Telegram user.

- `sharedData.js`: Exports constants such as SQL queries, helper functions, or common API keys.
- `queries.sql`: SQL script defining the database schema (tables for users, resumes, etc.) – run once to initialize the PostgreSQL database.
- `steps.txt`: Contains notes on development and deployment process (e.g., connecting to Railway).
- `package.json`: Lists dependencies (Express, pg, etc.) and scripts like `start`.

Backend API Documentation

All API routes are prefixed by `/api`. The backend does **not** enforce user authentication; instead, it uses user-provided identifiers (like email or document ID) to fetch records. (For security, a real deployment might add token-based auth, but it is not implemented here.Instaed on client side, clerk is used.)

API Endpoints-

Base path

All endpoints are rooted at:

```
/api
```

Quick Reference Table

Endpoint	Method	Description
`/api/user-resumes/upload/:id/:teleUser`	**POST**	Upload a resume file (Telegram user). Expects multipart `files`.
`/api/user-resumes`	**POST**	Create a new resume entry (inserts into DB). **Response:** fixed `{ data: { documentId: 1 } }` (see note).
`/api/user-resumes?userEmail=<email>`	**GET**	Get all resumes for a user.
`/api/user-resumes/:id?populate=*`	**GET**	Get a resume and its related sections (experience, education, skills).
`/api/user-resumes/:id`	**PUT**	Update a specific resume section (`summery`, `personalDetails`, `experience`, `education`, `skills`, or theme color).

`/api/user-resumes/:id`	DELETE	Delete a resume and related DB rows (resume, experience, education, skills, telegramusers).
`/api/user-resumes/ats/:id`	POST	Upload a resume file for ATS processing. Expects multipart `files`.
`/api/user-resumes/fetchScore/:id`	GET	Return ATS score JSON after `flashAI` has populated `dataForAts`.
`/api/user-resumes/fetchRecommendations/:id`	GET	Return recommendations array after ATS processing and `fetchRecommendations()` runs.
`/api/feedbacks`	POST	Submit feedback.
`/api/feedbacks`	GET	Get all feedbacks (stored in server memory).

Detailed Endpoints

1. Upload Resume (Telegram User)

POST `/api/user-resumes/upload/:id/:teleUser`

Description: Uploads a PDF (or other file) for a Telegram user. The route uses `multer` with memory storage and expects the file field named `files`. The file buffer is inserted into the `telegramusers` table. The server also updates `sharedData` (buffer, id, fileName).

Path params:

- `:id` — documentId to store in DB (string)
- `:teleUser` — telegram username (string)

Request headers:

```
Content-Type: multipart/form-data
```

Request body (multipart):

- `files` — file binary (uploaded file)

Behavior & responses (per code):

- If no file is sent:
  - **Status:** `400 Bad Request`

  - **Body (JSON):**

    ```
    { "error": "No file uploaded" }
    ```

- If file is received and DB insert succeeds:
  - **Status:** `200 OK`
  - **Body (plain text):** (exact string)

    ```
    File uploaded successfully with id <id>
    ```

  - Example: `File uploaded successfully with id 12345`
- If DB insertion / other error occurs:
  - The code logs the error to console; there is no explicit error `res.status(...)` other than the 400 case — but failures will be visible in server logs.

Notes:

- The code stores the file buffer to DB column `pdf` and filename into `fileName` for `telegramusers` table.
- `sharedData` outer export is updated with buffer, id and fileName (server-side).

---

2. Create Resume entry

POST `/api/user-resumes`

Description: Inserts a row into `resume` table with (`title`, `documentId`, `userEmail`, `userName`) using `req.body.data`. Important: due to how the code is written, the response sent is a fixed object declared before the DB insertion (see below).

Request headers:

```
Content-Type: application/json
```

Request body (expected by insertion):

```
{
  "data": {
    "title": "Software Engineer Resume",
    "documentId": 12345,
    "userEmail": "john@example.com",
    "userName": "John Doe"
```

```
  }
}
```

Behavior & responses (per code):

- The server attempts to `INSERT INTO resume (title, "documentId", "userEmail", "userName")` using `req.body.data` values.
- **Response sent by the code:** the function returns the pre-declared `data` object:

  ```
  {
    "data": {
      "documentId": 1
    }
  }
  ```

  This is because there is an outer `const data = { data: { documentId: documentId } }` (which is `1`) and an inner `const data = req.body.data` inside the `try` block; the inner one is block-scoped and the `res.send(data)` outside the try refers to the outer `data`. Therefore, **the response is always `{ "data": { "documentId": 1 } }`** per the current code, regardless of the inserted values. (The DB insert still runs using `req.body.data`.)
- If DB insertion throws, the code logs the error; no error response is explicitly returned (client will still receive the outer `data` response unless the server crashes).

---

3. Get All Resumes (by user email)

GET `/api/user-resumes?userEmail=<email>`

Description: Fetches all rows from `resume` table where `"userEmail" = $1` (value from query param). Response is JSON with `data` key containing DB rows.

Query params:

- `userEmail` — the email used to filter resumes (required)

Response (success): `200 OK` with JSON:

```
{
  "data": [
    {
      "id": <db id>,
      "title": "...",
      "documentId": ...,
      "userEmail": "...",
      "userName": "...",
      // other columns present in the resume table row
```

```
    }
    // ...more rows
  ]
}
```

Failure: On DB error, server logs error and responds:

- **Status:** `500 Internal Server Error`
- **Body (plain text):** `Error fetching data from database`

---

4. Get Resume with Details (experience, education, skills)

GET `/api/user-resumes/:id?populate=*`

Description: When `populate=*` query is present exactly, the route fetches the `resume` row for given `documentId` and then loads related `experience`, `education`, and `skills` rows for that `documentId`. It transforms DB columns that are arrays (in the DB) into structured arrays of objects in the response.

Path params:

- `:id` — the `documentId` to fetch

Query params:

- `populate` — must be exactly `*` to return populated data; otherwise the handler responds with an error (see below).

Response (success) — structure produced by code:

```
{
  "data": {
    // fields from the resume row (as returned by DB), for example:
    "documentId": 12345,
    "title": "Software Engineer Resume",
    "summery": "Experienced dev...",
    // plus three populated arrays:
    "Experience": [
      {
        "title": "...",
        "companyName": "...",
        "city": "...",
        "state": "...",
        "startDate": "...",
        "endDate": "...",
        "workSummery": "..."
      }
      // ... for each experience entry
    ],
    "education": [
      {
        "degree": "...",
        "university": "...",
        "major": "...",
        "startDate": "...",
        "endDate": "...",
        "description": "..."
      }
      // ... for each education entry
    ],
    "skills": [
      {
        "name": "...",
        "rating": "..."
      }
      // ... for each skill entry
    ]
  }
}
```

Failure cases:

- If `populate` query is missing or not exactly `*`:
  - **Status:** `500 Internal Server Error`
  - **Body (plain text):** `No populate query found`
- If DB fetching fails for any query, code logs the error and sends:
  - **Status:** `500 Internal Server Error`
  - **Body (plain text):** `Error fetching data from database`

Notes about transformation logic:

- `experience`, `education`, and `skills` rows are expected to have array columns (`title`, `companyName`, `startDate`, etc.). The code iterates the arrays and builds a list of objects per entry.

---

5. Update Resume Section

PUT `/api/user-resumes/:id`

Description: Updates one of several sections of a resume based on `req.body.section`. The handler expects `req.body.data` containing the data for that section and `req.params.id` for `documentId`. After performing DB updates/inserts, the handler responds by echoing the entire `req.body`.

Request headers:

```
Content-Type: application/json
```

Request body – required fields:

```
{
  "section": "<one of: summery, personalDetails, experience, education, skills, (otherwi
  "data": { ... }
}
```

Supported `section` values and required `data` shapes (as used by code):

- `summery`
  - `data` object must contain `summery` string.
  - DB query: `UPDATE resume SET "summery" = $1 WHERE "documentId" = $2`

  Example request body:

  ```
  {
    "section": "summery",
    "data": { "summery": "Experienced full-stack developer..." }
  }
  ```

- `personalDetails`
  - `data` must contain `firstName`, `lastName`, `jobTitle`, `address`, `phone`, `email`.
  - DB query updates corresponding columns in `resume` table.

  Example:

  ```
  {
    "section": "personalDetails",
    "data": {
      "firstName": "John",
      "lastName": "Doe",
      "jobTitle": "Software Engineer",
      "address": "Lucknow, India",
      "phone": "+91 9876543210",
      "email": "john@example.com"
    }
  }
  ```

- `experience`

  - `data` shape per code: `{ "Experience": [ { title, companyName, city, state, startDate, endDate, workSummery }, ... ] }`
  - The handler extracts arrays from the provided `Experience` array and writes them into the `experience` table as array columns. It either `UPDATE`s an existing `experience` row for the `documentId` or `INSERT`s a new row and then updates it.

  Example:

  ```
  {
    "section": "experience",
    "data": {
      "Experience": [
        {
          "title": "Software Engineer",
          "companyName": "Tech Corp",
          "city": "Mumbai",
          "state": "MH",
          "startDate": "2022-06-01",
          "endDate": "2024-06-01",
          "workSummery": "Worked on APIs..."
        }
      ]
    }
  }
  ```

- `education`
  - `data` shape: `{ "education": [ { degree, university, major, startDate, endDate, description }, ... ] }`
  - The handler converts these into parallel arrays and updates/inserts into `education` table.

  Example:

  ```
  {
    "section": "education",
    "data": {
      "education": [
        {
          "degree": "B.Tech",
          "university": "Amity University",
          "major": "CSE",
          "startDate": "2022",
          "endDate": "2026",
          "description": "High GPA"
        }
      ]
    }
  }
  ```

- `skills`
  - `data` shape: `{ "skills": [ { name, rating }, ... ] }`
  - Handler converts to `name[]` and `rating[]` arrays and updates/inserts into `skills` table.

  Example:

  ```
  {
    "section": "skills",
    "data": {
      "skills": [
        { "name": "JavaScript", "rating": 5 },
        { "name": "React", "rating": 4 }
      ]
    }
  }
  ```

- **Fallback (theme color)**
  - If `section` is none of the above, code treats the request as themeColor update. It expects `data.themeColor` and runs:
    `UPDATE resume SET "themeColor" = $1 WHERE "documentId" = $2`

  Example:

  ```
  {
    "section": "theme",
    "data": { "themeColor": "#6EE7B7" }
  }
  ```

Response (per code): server returns the exact `req.body` as-is:

- **Status:** `200 OK` (implied)
- **Body:** the same JSON object sent in the request (echo).
  Example echo response for a skills update will be the same JSON as the request body above.

Errors: On DB error the code logs error; no explicit error payload is returned by the handler (it still ends with `res.send(req.body)` irrespective of success/failure).

---

6. Delete Resume and related rows

DELETE `/api/user-resumes/:id`

Description: Deletes rows matching `documentId` from `resume`, `experience`, `education`, `skills`, and `telegramusers` tables.

Path params:

- `:id` — `documentId` to delete

Success response:

- **Status:** `200 OK`
- **Body (plain text):** (exact string)

```
Document deleted successfully with id <id>
```

Failure: If any DB error occurs:

- **Status:** `500 Internal Server Error`
- **Body (plain text):** `Error deleting data from database`

---

7. Upload Resume for ATS processing

POST `/api/user-resumes/ats/:id`

Description: Uploads file (field `files`), stores buffer into shared data via `setBuffer()` and calls `fetchATS()` (imported from `flashAI.js`). After calling `fetchATS()` the route responds with a plain-text message.

Request headers:

```
Content-Type: multipart/form-data
```

Request body (multipart):

- `files` — uploaded file

Response:

- **Status:** `200 OK`
- **Body (plain text):**

```
ATS score fetched successfully with id <id>
```

Note: `fetchATS()` runs asynchronously and the route does not wait for a separate result besides awaiting the function call — the actual ATS results are later available via `GET /api/user-resumes/fetchScore/:id` which reads `dataForAts` exported from `sharedData.js`.

---

8. Fetch ATS Score

GET `/api/user-resumes/fetchScore/:id`

Description: Waits until `dataForAts.fetched` becomes truthy, then returns the `score` and `fetched` flags from `dataForAts` object (imported from `sharedData.js`). The route polls every 3 seconds while waiting.

Path params:

- `:id` — passed but not used to index the stored `dataForAts` in code; response uses global `dataForAts` object.

Response (per code):

- **Status:** `200 OK`
- **Body (JSON):**

  ```
  {
    "score": <dataForAts.score>,
    "fetched": <dataForAts.fetched>
  }
  ```

  Example if `dataForAts.score` is `78` and `fetched` is `true`:

  ```
  { "score": 78, "fetched": true }
  ```

Notes: The code polls until `dataForAts.fetched` is `true`. There is no timeout in the handler; a long wait will keep the request open until the flag changes.

---

9. Fetch Recommendations

GET `/api/user-resumes/fetchRecommendations/:id`

Description: Waits until `dataForAts.fetched` is true, then calls `fetchRecommendations()` (which populates the exported `recommendations` variable from `flashRecommendations.js`) and returns that `recommendations` array in JSON.

Path params:

- `:id` — accepted but not used to select which recommendations to return; the handler returns the module-level `recommendations` value.

Response:

- **Status:** `200 OK`
- **Body (JSON):**

```
{
  "recommendations": [ /* contents of imported `recommendations` array */ ]
}
```

Notes: The endpoint calls `await fetchRecommendations()` after waiting for `dataForAts.fetched`, so it triggers generation/refresh of `recommendations` before returning them.

🔟 Submit Feedback

POST `/api/feedbacks`

Description: Pushes `req.body` into the server in-memory `feedback` array (exported) if present, and responds with a plain text confirmation.

Request headers:

```
Content-Type: application/json
```

Request body: any JSON object representing feedback. Example:

```
{
  "rating": 5,
  "feedback": "Amazing resume builder!"
}
```

Response (per code):

- **Status:** `200 OK`
- **Body (plain text):**

  ```
  Feedback received successfully
  ```

Notes: `feedback` array begins with a default entry:

```
[
  {
    "rating": -1,
    "feedback": "No ratings given yet!"
  }
]
```

New feedback items are `push`ed into that array in memory (not persisted to DB).

1️⃣ 1️⃣ Get All Feedbacks

GET `/api/feedbacks`

Description: Returns the server in-memory `feedback` array as JSON.

Response:

- **Status:** `200 OK`
- **Body (JSON):** current `feedback` array, e.g.

```
[
  { "rating": -1, "feedback": "No ratings given yet!" },
  { "rating": 5, "feedback": "Amazing resume builder!" }
]
```

Root / Static route

GET `/` (no `/api` prefix) — Serves `public/index.html` using `res.sendFile(__dirname + "/public/index.html")`.

Implementation notes & future improvement-(AFTER REVIEW BY DEVELOPERS)

- Several handlers return plain text strings (e.g. upload success, delete success, ATS upload success, feedback confirmation) — they are **not JSON** unless explicitly using `res.json`. Keep that in mind if a client expects JSON.
- `POST /api/user-resumes` responds with a fixed `{ "data": { "documentId": 1 } }` due to how variables are scoped in the handler (outer `data` object is sent). This behavior is faithful to the provided code.
- `GET /api/user-resumes/fetchScore/:id` and `/fetchRecommendations/:id` rely on module-level state (`dataForAts` and `recommendations`) — the routes wait (poll) until `dataForAts.fetched` is true. There is no per-user documentId mapping in these handlers: they return the module-level values.
- Error paths often just log and return generic text or HTTP 500; consider adding consistent JSON error payloads in future for API clarity.

AI Model Modules

> This document describes the AI-related modules implemented in the backend logic.

Environment

- **Required environment variable:** `GOOGLE_API_KEY`
  The code instantiates `new GoogleGenAI({ apiKey: process.env.GOOGLE_API_KEY })`. If this variable is missing or invalid, the Google GenAI client will not authenticate correctly.

Files & Modules Covered

- `flashAI.js` — Generates an ATS score using Google GenAI and sets it via `setScore`.
- `flashRecommendations.js` — Generates resume improvement recommendations using Google GenAI and exports `recommendations`.
- `sharedData.js` — Shared state between modules: `dataForAts`, `setBuffer(buffer)`, `setScore(score)`.

---

1) `flashAI.js` (AI ATS scorer)

Exports / API

- **Default export:** `async function main()` — when invoked it:
  - reads `dataForAts.buffer`
  - if buffer is null -> logs error and returns
  - otherwise constructs `contents` for `ai.models.generateContent()` and calls the Google GenAI client
  - cleans the returned `response.text`, parses it as JSON, reads `ats_score`, and calls `setScore(score)`.

Important code behavior

- `const ai = new GoogleGenAI({ apiKey: process.env.GOOGLE_API_KEY });`
- `const pdfResp = dataForAts.buffer;`
  - If `pdfResp === null`, the function logs `"Buffer is null. Buffer not intialized yet."` and `return`s.
- `contents` is an array with two elements:
  i. A `text` item containing the instruction:

```
Give ats score for the resume returning a json(not markdown) as follows and do
{
"ats_score": ats score in percentage
}
```

ii. An `inlineData` object:

```
{
  inlineData: {
    mimeType: 'application/pdf',
    data: Buffer.from(pdfResp).toString("base64")
  }
}
```

- The code calls:

```
const response = await ai.models.generateContent({
  model: "gemini-2.0-flash",
  contents: contents
});
```

- Post-processing:
  - `response.text` is cleaned by removing any occurrences of "`json" or "`" via ``response.text.replace(/```json|```/g, '').trim();``
  - The function uses `JSON.parse(cleaned)` (exactly) and expects the result to be an object with an `ats_score` property.
  - It logs `JSON.parse(cleaned)` and then `console.log("ATS score:", score.ats_score);`
  - Finally calls `setScore(score.ats_score);`

Expected AI output format (required by the code)

The AI `response.text` must contain valid JSON (possibly wrapped in code fences). After removing code fences the string must parse to:

```
{
  "ats_score": <number>
}
```

The code will parse this and pass the numeric `ats_score` value to `setScore()`.

Error handling & notes for future improvements (AFTER REVIEW BY DEVELOPERS)

- If the AI returns malformed JSON that cannot be parsed by `JSON.parse`, the code will throw and crash unless the exception is caught elsewhere. The code does not wrap `JSON.parse` in a `try/catch` in the snippet you provided.
- The `main()` function is **not** invoked automatically in the file (it's commented out). It must be imported and called by another module (e.g., the server) to run.

---

2) `flashRecommendations.js` (AI recommendations generator)

Exports / API

- **Default export:** `async function main()` — when invoked:
  - reads `dataForAts.buffer`
  - if buffer is null -> logs error and returns
  - otherwise constructs `contents` for `ai.models.generateContent()` and calls the Google GenAI client
  - processes `response.text` into a cleaned `recommendations` string (module-level)
- **Named export:** `export { recommendations }` — `recommendations` is a module-level `let` that holds the cleaned `response.text`.

Important code behavior

- `const ai = new GoogleGenAI({ apiKey: process.env.GOOGLE_API_KEY });`
- `let recommendations = '';` (module-level variable)
- The function constructs `contents`:
  - i. A `text` item: `"Give recommendations to improve this resume's ATS Scores."`
  - ii. An `inlineData` object with the PDF buffer base64 identical to `flashAI.js`.
- Calls:

  ```
  const response = await ai.models.generateContent({
    model: "gemini-2.0-flash",
    contents: contents
  });
  ```

- Post-processing of `response.text` (exact sequence):
  - i. `console.log(response.text);`
  - ii. `recommendations = response.text`
    - `.replace(/\*\*(.*?)\*\*/g, (_, text) => text.toUpperCase())`
    - `.replace(/\*(.*?)\*/g, (_, text) => text)`
    - `.replace(/

      ?

      {2,}/g, '

') ` - `.replace(/[`#_>~]/g, '') ` - `.trim();`

- The `recommendations` variable holds the final cleaned string and is exported.

Expected AI output format

- The code treats the AI reply as plain text (not JSON). The reply can be in Markdown or plain text; the code attempts to clean Markdown-like characters and normalize spacing. The final exported `recommendations` is a string.

Error handling & notes for future improvements(AFTER REVIEW BY DEVELOPERS)

- If `dataForAts.buffer` is `null`, the function logs `"Buffer is null for recommendations. Buffer not intialized yet."` and returns.
- The `main()` function is **not** invoked automatically (commented out). Another module must import and call it.
- The code does not validate or parse the AI text into structured JSON — it leaves `recommendations` as a cleaned string.

---

3) `sharedData.js` (shared state)

Exports / API (exact)

- **Exported object:** `export let dataForAts = { buffer: null, score: 0, fetched: false };`
- **Exported function:** `export function setBuffer(buffer)`
  - Sets `dataForAts.buffer = buffer;`
  - Resets `dataForAts.score = 0;`
  - Sets `dataForAts.fetched = false;`
- **Exported function:** `export function setScore(score)`
  - Sets `dataForAts.score = score;`
  - Sets `dataForAts.fetched = true;`

Behavior

- `dataForAts` is a shared module-level object used by both AI modules and the server.
- Typical flow intended by the code:
  - i. Some code (e.g., server route) calls `setBuffer()` with a PDF buffer (from upload).
  - ii. `flashAI.main()` reads `dataForAts.buffer`, generates an ATS score, and calls `setScore()`.
  - iii. `dataForAts.fetched` becomes `true` and `dataForAts.score` holds the ATS numeric score.
  - iv. Other parts of the server poll or read `dataForAts` (for example `GET /api/user-resumes/fetchScore/:id` waits for `dataForAts.fetched`).

Notes & caveats (AFTER REVIEW BY DEVELOPERS)

- `dataForAts` is process-global per Node process. It is **not** persisted across restarts.
- No concurrency locks or queuing are implemented — if multiple buffers are set concurrently, the last one wins.
- `setBuffer()` resets `score` and `fetched` to initial values; callers must wait for `setScore()` to be called to see `fetched: true`.

Usage summary

- The server sets the PDF buffer via `setBuffer(req.file.buffer)` (server code does this in `/api/user-resumes/ats/:id`).
- After `setBuffer()` the server calls `await fetchATS()` (which is the default export of `flashAI.js`) — this triggers the GenAI call and eventually `setScore()`.
  - In your server code:

    ```
    setBuffer(req.file.buffer);
    await fetchATS();
    ```

- Later, server route `GET /api/user-resumes/fetchScore/:id` polls `dataForAts.fetched` and returns `dataForAts.score` and `dataForAts.fetched` as JSON (server code handles this).
- For recommendations, the server calls `await fetchRecommendations()` (default export of `flashRecommendations.js`) and then returns the exported `recommendations` array/string.

Implementation caveats & exact behavior to be aware of (AFTER REVIEW BY DEVELOPERS)

- **AI response parsing is strict in `flashAI.js`**: the code expects JSON text that parses to an object with `ats_score`. The code will `JSON.parse` without try/catch.
- **`flashRecommendations.js` returns a plain string:** the server returns `recommendations` as whatever cleaned text the AI returned — there is no structure enforced.
- **No per-document mapping in `sharedData`**: `dataForAts` is a single global buffer/score. The `:id` path params in server endpoints are not used to index multiple `dataForAts` objects.
- **Both `main()` functions are not auto-executed** (their `main()` calls are commented out in the provided snippets). They must be invoked by the server or another orchestrator to run.

Exact strings logged / expected messages

- `"Buffer is null. Buffer not intialized yet."` — from `flashAI.js` when buffer is null
- `"Buffer is null for recommendations. Buffer not intialized yet."` — from `flashRecommendations.js` when buffer is null

- The server code prints `"ATS score:"` and the numeric value after parsing in `flashAI.js`.

Telegram Bot

> This document describes the Telegram bot module implemented in the backend logic.

Environment

- **Required environment variable:** `TELEGRAM_BOT_TOKEN`
  The bot is instantiated with:

  ```
  const token = process.env.TELEGRAM_BOT_TOKEN;
  const bot = new TelegramBot(token, { polling: true });
  ```

Main behavior & listeners

1) `bot.onText(/\greet/, (msg, match) => { ... })`

Description:

- The code registers an `onText` handler using the regular expression literal `/\greet/`.
- Inside the handler:
  - `const chatId = msg.chat.id;`
  - `const resp = match[1];` // `match[1]` is used
  - Sends a reply:

    ```
    bot.sendMessage(chatId, "Hello, I am your bot! How can I assist you today?
    Answered to:" + resp);
    ```

Notes & caveats (AFTER REVIEW BY DEVELOPERS):

- The regex `/\greet/` does not contain a capture group (i.e., there is no `( ...)` group), so `match[1]` will be `undefined` unless the regex engine or node-telegram-bot-api provides a different `match` shape. The code uses `match[1]` directly — the resulting `resp` may therefore be `undefined`.
- The regex literal includes a backslash before `g` (`\g`) which is not a standard escape sequence; depending on JS engine it may be interpreted as `g` or cause unexpected behavior. This is a factual observation of the code — the code does not alter the regex.

---

2) `bot.onText(/\/sendpdf/, async (msg, match) => { ... })`

Description:

- Handler activated when incoming text matches `/\/sendpdf/` (a literal `/sendpdf` command).
- Steps performed (per code):
   i. `const chatId = msg.chat.id;`
   ii. `const userName = msg.from.username;`
   iii. Logs `sharedData`.
   iv. Executes DB query:

   ```
   const result = await db.query(`SELECT * FROM telegramusers WHERE "userName" = $
   ```

   v. If `result.rows.length > 0`, the code iterates `result.rows.forEach((row) => { ... })` and for each `row`:
   - `const pdfData = row.pdf;` (assigned as `const`)
   - Checks:

     ```
     if (!Buffer.isBuffer(pdfData)) {
         console.log("pdfData is not a buffer. Converting...");
         pdfData = Buffer.from(pdfData, 'binary');
     }
     ```

   - Sends document:

     ```
     bot.sendDocument(chatId, pdfData, {
         caption: 'Here is your AI-generated Resume!',
     }, {
         filename: row.fileName,
         contentType: 'application/pdf',
     })
     .catch((error) => {
         console.error('Error sending document:', error);
         bot.sendMessage(chatId, 'Sorry, there was an error sending the PDF.');
     });
     ```

   vi. If no rows found, sends:

   ```
   bot.sendMessage(chatId, 'Sorry, no records found!');
   ```

   vii. If DB query throws, `catch` block logs error and sends:

   ```
   bot.sendMessage(chatId, 'Sorry, you are not authorized to access this document.
   ```

Notes & exact caveats visible in code:(AFTER REVIEW BY DEVELOPERS)

- `pdfData` is declared with `const pdfData = row.pdf;` and later reassigned with `pdfData = Buffer.from(...)` inside the `if` block. Reassigning a `const` will throw a runtime `TypeError` (`Assignment to constant variable`) — this would crash the handler at runtime unless changed to `let`. The code as written performs this reassignment.
- The `Buffer.isBuffer(pdfData)` check is present; if `row.pdf` is not a Buffer (for example a binary string), the code attempts to convert it to a Buffer (but due to `const` it will error as described).
- `bot.sendDocument` is called with four arguments in the code: `(chatId, pdfData, { caption }, { filename, contentType })`. The node-telegram-bot-api `sendDocument` signature usually expects `(chatId, doc, options)` where `options` may include `caption` and `filename`. The code supplies two separate option objects — the code will pass these literally to the library as written; behavior depends on the library's parameter handling.
- The database query uses `userName` taken from `msg.from.username`. If `username` is `undefined` (user hasn't set a Telegram username), the query will run with `[undefined]` and likely return no rows.

---

3) `bot.on('message', (msg) => { ... })`

Description (exact):

- Listens for any incoming message.
- Inside handler:
  - `const chatId = msg.chat.id;`
  - Logs the message: `console.log("Response from user", msg);`
  - Sends a fixed reply:

    ```
    bot.sendMessage(chatId, 'Hello,Kenshin Commander reporting! Have a pathetic day
    ```

Notes:

- The reply string is exactly as in code and will be sent for every message received (including commands) unless other handlers intercept earlier. The string contains wording that may be considered informal; documentation records the exact text.

Permissions & setup notes

- Bot uses **polling** (not webhooks):

  ```
  const bot = new TelegramBot(token, { polling: true });
  ```

- Ensure `TELEGRAM_BOT_TOKEN` is set in environment before running.

Potential runtime issues — exact observations from code(AFTER REVIEW BY DEVELOPERS)

- Reassigning `const pdfData` will throw at runtime when `row.pdf` is not already a Buffer and the conversion code attempts `pdfData = Buffer.from(...)`.
- `bot.onText(/\greet/, ...)` uses `match[1]` though the regex has no capture group — `resp` will likely be `undefined`.
- `bot.sendDocument` invocation uses two option objects; depending on the bot library's implementation this may or may not behave as intended.
- The DB query and document send assume `userName` exists in `msg.from.username`; if absent, the query may find no rows.

Exact strings logged / sent by the bot

- Logged: `console.log("received data--->", sharedData);` (in sendpdf handler)
- Sent as reply in greet handler:

  ```
  Hello, I am your bot! How can I assist you today?
  Answered to: <resp>
  ```

  (where `<resp>` equals `match[1]` per code; possibly `undefined`.)
- Sent when sending a PDF: caption `'Here is your AI-generated Resume!'`
- Sent on DB error or no records: `'Sorry, no records found!'` or `'Sorry, you are not authorized to access this document.'`
- Default message for any message:

  ```
  Hello,Kenshin Commander reporting! Have a pathetic day baby!
  ```

Recommendation (code-level) — purely informational(AFTER REVIEW BY DEVELOPERS)

> The following suggestions are *observations* based on the exact code and are provided only to help avoid runtime errors if you decide to modify the code:
>
> - Change `const pdfData = row.pdf;` to `let pdfData = row.pdf;` before attempting to reassign it.

> - Use a regex with a capture group when you intend to access `match[1]`, for example `/\/greet (.+)/` (if you actually want a captured parameter), or use the provided `msg.text` directly.
> - Consolidate `sendDocument` options into a single options object as the library expects, for example:
>
>   ```
>   bot.sendDocument(chatId, pdfData, { caption: '...', filename: row.fileName, con
>   ```
>
> These changess may be applied to the current code in future — only suggestions after review are mentioned.

Database Connection

> This document documents the database connection and related usage implemented in the Node.js/Express codebase.

Overview

The application uses the `pg` package and instantiates a single `pg.Client` configured from environment variables. The code expects a managed Postgres-compatible service (the code prints messages referencing "Neon Database"). All queries are executed using `db.query(...)` on the single `Client` instance which is exported as the default export.

The code also:

- uses an **in-memory multer storage** (`multer.memoryStorage()`) to accept uploaded PDF files and then stores the binary data into the database (`bytea` / binary column).
- relies on parameterized queries (`$1, $2, ...`) to avoid SQL injection.

Environment variables

The following environment variables are used by the connection logic and server configuration:

```
PORT            # optional, server port (defaults to 3000)
DB_USER         # database username
DB_HOST         # database host (hostname)
DB_NAME         # database name
DB_PASSWORD     # database password
DB_PORT         # database port (numeric)
```

Example `.env` (do not commit to source control)

```
PORT=3000
DB_USER=your_db_user
DB_HOST=db.your-host.example
DB_NAME=your_db_name
DB_PASSWORD=supersecretpassword
DB_PORT=5432
```

> The code calls `dotenv.config()` at the top, so these variables are read from the environment or `.env` file.

Connection setup

The code creates a `pg.Client` like this (paraphrased):

- `user`: `process.env.DB_USER`
- `host`: `process.env.DB_HOST`
- `database`: `process.env.DB_NAME`
- `password`: `process.env.DB_PASSWORD`
- `port`: `process.env.DB_PORT`
- `ssl`: `{ rejectUnauthorized: false }`

The client is then connected with `db.connect((err) => { ... })`. The callback prints either an error message or a success message and — on success — starts a 4-minute interval to run a simple `SELECT NOW()` query to keep the managed DB (Neon) from idling.

Important detail from the code:

- The code sets `ssl: { rejectUnauthorized: false }` (the comment says "for production, uncomment the ssl part" — but the current code already includes `ssl`. If you switch environments, ensure SSL settings match your DB provider requirements).

Wake-up / keep-alive behavior

After successful connection the code executes a periodic wake-up request every **4 minutes** (4 * 60 * 1000 ms):

```
setInterval(async () => {
  try {
    const awakeTime = await db.query('SELECT NOW()');
    console.log("NeonDB is awake!");
    console.log("Checked at:", awakeTime.rows[0].now.toLocaleString());
  } catch (error) {
    console.error("Error waking up NeonDB:", error);
    console.error("Checked at:", new Date().toLocaleString());
  }
}, 4 * 60 * 1000);
```

This is used to prevent the managed DB from going idle. Keep this interval if you rely on a serverless / autoscaling DB product that idles connections.

Error handling & reconnect logic (commented in code)

The repository contains a commented-out `db.on('error', ...)` block that demonstrates a reconnect pattern used by the author. Key points from that block:

- It logs that NeonDB went idle and prints the time.
- Calls `db.end()` to close the current connection, then creates a new `pg.Client` with the same configuration and calls `db.connect()` again.
- Prints a reconnect message and time.

Note: That reconnect logic is commented out. If you plan to use it, test carefully — re-creating `db` like that requires careful scoping and you must ensure all modules that import `db` continue to reference the re-created client. Consider using `pg.Pool` for robust connection handling in production.

How the `db` client is used in routes

The code performs standard parameterized queries across many routes. Highlights from the code (the exact queries are implemented in the application):

- `INSERT` PDF file into `telegramusers` with `pdf` column (binary) and `fileName`:

  ```
  INSERT INTO telegramusers("documentId","userName", pdf, "fileName") VALUES($1,$2,$3,
  ```

  where `req.file.buffer` is passed for `$3`.

- Create `resume` record:

  ```
  INSERT INTO resume (title,"documentId","userEmail","userName") VALUES ($1, $2, $3, $
  ```

- Fetch resumes by user email:

```
SELECT * FROM resume WHERE "userEmail" = $1
```

- Fetch a full document by `documentId` and populate related arrays from `experience`, `education`, `skills`. The code reads array columns and maps them into JSON objects before returning to client.
- Update `resume` fields like `summery`, personal details or `themeColor` using `UPDATE resume SET ... WHERE "documentId" = $1`.
- For collections (`experience`, `education`, `skills`) the code treats columns as arrays (e.g., `title`, `companyName`, `degree`, `name`, `rating`, etc.), updates them by replacing entire arrays via `UPDATE ... SET column = $1 WHERE "documentId" = $2`, and when missing inserts a skeleton row with `INSERT INTO experience ("documentId") VALUES ($1)` then updates.
- Deletion of a document cascades to multiple tables via separate `DELETE FROM ... WHERE "documentId" = $1` statements.

Concurrency & transactions: the current code runs separate queries sequentially with `await`. If you have multi-step operations that must be atomic (for example creating + updating related tables), consider wrapping them in a transaction (`BEGIN` / `COMMIT`) — the current code does not use explicit transactions.

Database schema

```
-- resume table
CREATE TABLE resume (
  id serial PRIMARY KEY,
  title text,
  "documentId" integer UNIQUE NOT NULL,
  "userEmail" text,
  "userName" text,
  summery text,
  "firstName" text,
  "lastName" text,
  "jobTitle" text,
  address text,
  phone text,
  email text,
  "themeColor" text
);

-- experience table: arrays for each field
CREATE TABLE experience (
  id serial PRIMARY KEY,
  "documentId" integer UNIQUE NOT NULL,
  title text[],
  "companyName" text[],
  city text[],
  state text[],
  "startDate" text[],
  "endDate" text[],
  "workSummery" text[]
);

-- education table: arrays
CREATE TABLE education (
  id serial PRIMARY KEY,
  "documentId" integer UNIQUE NOT NULL,
  degree text[],
  university text[],
  major text[],
  "startDate" text[],
  "endDate" text[],
  description text[]
);

-- skills table: arrays
CREATE TABLE skills (
  id serial PRIMARY KEY,
  "documentId" integer UNIQUE NOT NULL,
  name text[],
  rating integer[]
);

-- telegramusers table: store uploaded pdfs
CREATE TABLE telegramusers (
  id serial PRIMARY KEY,
  "documentId" integer NOT NULL,
  "userName" text,
  pdf bytea,
  "fileName" text
);
```

Notes:

- Columns like `title`, `companyName`, `degree`, `name`, etc. are stored as PostgreSQL arrays (e.g., `text[]`) because the code expects to iterate through `rows[0].title.length` and index into them.
- `pdf` is stored as `bytea` to hold binary file buffers.
- `documentId` is treated as the canonical identifier across tables. The code expects exactly one row per `documentId` in `resume`, `experience`, `education`, and `skills`.

Best practices & recommendations (compatible with the current code)

1. **Use connection pooling for production:** The code uses a single `pg.Client`. For higher throughput and robust reconnects, prefer `pg.Pool` and `pool.connect()` / `pool.query()`.

2. **Use transactions for multi-step operations:** If you `INSERT` then `UPDATE` related rows, use `BEGIN` / `COMMIT` to keep operations atomic.

3. **Avoid storing large files directly in the DB if you expect heavy traffic:** Storing PDFs in `bytea` is acceptable for small use, but for scale consider object storage (S3/MinIO) and store references in the DB.

4. **Validate inputs:** The code trusts `req.body` and `req.params`. Add validation (e.g., `Joi`, `zod`) before querying the DB.

5. **Be explicit about SSL in production:** Keep `ssl: { rejectUnauthorized: false }` only if your provider requires it. Prefer trusting CA or configure correctly for better security.

6. **Add indexes if queries become slow:** Primary keys and an index on `"userEmail"` and `"documentId"` should help.

7. **Be careful with the reconnect pattern:** Replacing the exported `db` object at runtime can break module references. Use pooling or a reconnect helper that maintains the same exported interface.

Troubleshooting

- **`Error connecting to Neon Database` logged:** verify env vars, network, and SSL settings. Check that `DB_HOST` and `DB_PORT` are correct and that the credentials are valid.
- **Neon / serverless DB idling:** if you still get idle disconnects, either keep the wake-up `SELECT NOW()` interval (as the code does) or use a connection pool with a long-lived process.
- **Binary data insertion fails:** ensure the column is `bytea` and that you pass `req.file.buffer` directly (as code does). For older `pg` versions you may need to use `Buffer.from(...)`.
- **Arrays not behaving as expected in queries:** confirm the table columns are declared as `text[]` / `integer[]` as shown in the schema above.

Where to find the `db` object in code

- The database client is exported as `export default db;` at the bottom of the file. Other modules may import it as the default import.

- Two additional named exports are present in the same file and used by other parts of the app:

  - `export const sharedData = { buffer: null, fileName: null, id: null }` — used to temporarily store uploaded file buffers and identifiers.
  - `export const feedback = [{ rating: -1, feedback: "No ratings given yet!" }];`

Quick checklist to deploy successfully

- ☐ Set environment variables (`DB_USER`, `DB_HOST`, `DB_NAME`, `DB_PASSWORD`, `DB_PORT`, `PORT`).
- ☐ Ensure network rules (allowlist) on the DB provider allow connections from your server.
- ☐ Confirm SSL requirements for your DB host and adjust `ssl` config accordingly.
- ☐ Run the SQL DDL to create the tables (or use your migrations).
- ☐ Test uploads and `SELECT NOW()` keep-alive once deployed.

Frontend Components & Interactions

The React frontend is structured around user interaction with resume data. While the exact component file names may vary, the logical flow is:

1. **Landing / Editor Page:** When a user visits Kenshi Resumes, they see a form with fields (Name, Email, Job Title, etc.) and a “Generate Resume” button. This is likely implemented as a React component (e.g. `EditorPage`).
2. **State Management:** As the user fills form fields, state (probably using React `useState`) is updated.
3. **Generate Action:** On clicking “Generate,” the app sends a `POST /api/user-resumes` request (via `fetch` or Axios) with the user’s data.
4. **Loading/Feedback:** The UI shows a spinner or message while the AI content is being generated.
5. **Display Resume:** Once the response arrives, the generated resume content (probably returned in the API response or fetched via another call) is displayed in a live preview pane. This could be done with a component like `ResumePreview` that takes JSON data and renders formatted resume sections.
6. **Editing:** The user can edit any part of the resume in a rich text editor (e.g. `contentEditable` fields or separate input fields) and see changes in real-time. Components for each section (e.g. `ExperienceSection`, `EducationSection`) allow inline editing.
7. **ATS Score & Recommendations:** The user can click “Check ATS Score.” This triggers a request to `/api/user-resumes/fetchRecommendations/:id` which returns tips. The app displays these tips, perhaps highlighting missing keywords.
8. **Save/Download:** The user can save the resume to their browser (e.g. download a PDF) or send it via Telegram. The “Send to Telegram” button might instruct the bot to send the resume to the user’s Telegram account, using the POST `/upload` route behind the scenes.
9. **Navigation/Routes:** The frontend may use React Router (or a single-page layout) for navigation. For example, there might be routes for `/editor`, `/preview`, `/about`, etc. However, since the repo is a single application, it might just be one page with conditional rendering.

Interactions Summary: Components communicate via props/state. Key interactions are form submission (to backend), content editing (in the browser), and the dynamic preview. All UI actions that require data (like generating or getting recommendations) are implemented with `fetch` calls to the backend API routes listed above.

Setup & Installation

Follow these steps to set up the project locally for development (each repo must be set up separately):

1. **Prerequisites:** Ensure you have Node.js (v16 or later) and PostgreSQL (v12+) installed. If deploying on Railway, you can use their PostgreSQL add-on.

2. **Clone the Repositories:**

   - Frontend: `git clone https://github.com/Kenshi2727/Kenshi-Resumes-AI-Powered.git`
   - Backend: `git clone https://github.com/Kenshi2727/Kenshi-Resumes-Backend.git`

3. **Install Dependencies:** For each project folder, run:

   ```
   cd kenshi-resumes-ai-powered   # frontend directory
   npm install
   cd ../Kenshi-Resumes-Backend   # backend directory
   npm install
   ```

4. **Configure Environment Variables:**

   - In **frontend**, create a `.env` in the root (you may copy from `.env.example` if available). Set `VITE_API_URL` to point to your backend (e.g. `http://localhost:5000`).

   - In **backend**, create a `.env` file with the following keys:

     ```
     PORT=5000
     DATABASE_URL=postgresql://<user>:<password>@<host>:<port>/<database>
     GEMINI_API_KEY=<your_google_gemini_api_key>
     TELEGRAM_BOT_TOKEN=<your_telegram_bot_token>   # optional, if using Telegram
     ```

     Replace `<user>`, `<password>`, etc., with your PostgreSQL credentials. `GEMINI_API_KEY` is required to call the Gemini API.

5. **Initialize the Database:** Using your PostgreSQL client (psql or a GUI), run the SQL script provided:

   ```
   psql $DATABASE_URL -f queries.sql
   ```

   This creates the necessary tables for users, resumes, etc.

6. **Run the Servers:**

   - **Backend:** In the `Kenshi-Resumes-Backend` folder, start the server:

```
npm start
```

The Express API should now be listening (e.g. on `http://localhost:5000`).

- **Frontend:** In the `Kenshi-Resumes-AI-Powered` folder, start the development server:

  ```
  npm run dev
  ```

  Vite will host the React app (commonly on `http://localhost:5173`). Open this in your browser.

Now you can use the app locally. When you click “Generate,” the frontend will call the local backend and populate the resume.

Deployment

Frontend (Vercel)

Kenshi Resumes frontend is designed for deployment on **Vercel**, an end-to-end platform for hosting and deploying web applications. To deploy:

1. Push your frontend code to a GitHub repository.
2. Create a Vercel account (free tier available).
3. Import the GitHub repo into Vercel. It will auto-detect the React/Vite setup and create a production build.
4. Set environment variables in the Vercel dashboard: e.g. `VITE_API_URL` (pointing to your backend URL) and `GEMINI_API_KEY`.
5. Deploy. Vercel will provide a URL (like `https://kenshi-resumes-ai-powered.vercel.app`) where the app is live.

Vercel supports real-time previews for pull requests and handles HTTPS certificates automatically. It can be configured via the vercel.json file included in the repo.

Backend (Render)

Render Deployment & CORS Patch —

Part A — CORS patch (ready to paste)

Below is a small code patch you can paste into your backend entry file (the file that contains your Express `app` and `app.use(cors(...))` call). It adds your Render service URL via an environment variable `RENDER_URL` while keeping existing allowed origins. Paste the replacement block **in place of** the existing `app.use(cors({ ... }))` block in your code.

```
--- original
-app.use(cors({
-    origin: ['https://kenshi-resumes-ai-powered.vercel.app', 'https://www.kenshi-res.ap
-    // origin: 'http://localhost:5173', // origins for development
-    methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
-    allowedHeaders: ['Content-Type', 'Authorization'],
-    //   credentials: true, // if using cookies/auth headers
-})); // Allow cross-origin requests
+// Allow CORS from known frontends and the Render backend URL (if provided)
+// Add RENDER_URL as an environment variable on Render (e.g. https://your-backend.onren
+const allowedOrigins = [
+  process.env.RENDER_URL, // Render service URL (set this in Render's Environment setti
+  'https://kenshi-resumes-ai-powered.vercel.app',
+  'https://www.kenshi-res.app'
+].filter(Boolean); // remove any undefined/empty values
+
+app.use(cors({
+    origin: allowedOrigins,
+    // origin: 'http://localhost:5173', // uncomment during local development if needed
+    methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
+    allowedHeaders: ['Content-Type', 'Authorization'],
+    // credentials: true, // enable if you use cookies/auth headers
+})); // Allow cross-origin requests
```

Notes:

- Set an environment variable `RENDER_URL` in Render to your service URL (e.g. `https://your-backend.onrender.com`). The patch reads `process.env.RENDER_URL` at runtime.
- The `.filter(Boolean)` ensures `undefined` or empty values are removed so CORS doesn't get `[""]` as an origin.
- After pasting, redeploy on Render so the new CORS origins take effect.
- If your code file uses `app.use(express.static('public'))` and other middleware order matters, keep this replacement in the same place as the original `app.use(cors(...))` line (do not move it above `app.use(express.static())` if you relied on the previous order).

Part B — Render deployment checklist (fill secrets in Render dashboard)

This checklist helps you deploy or update the Kenshi Resumes backend on **Render**, and includes exact environment variable names your code expects.

1. Repo & Service

- Push your backend code to GitHub (branch you want to deploy).
- In Render dashboard: create a new Web Service and connect the GitHub repo, then select the branch to deploy.
- Build command: `npm install` (or your preferred installer).
- Start command: `npm start` (or `node index.js` / `node server.js` depending on your repo).

2. Required environment variables (add these under the service's Environment in Render)

- `DB_USER` — Postgres username (from managed DB)
- `DB_HOST` — Postgres host
- `DB_NAME` — Postgres database name
- `DB_PASSWORD` — Postgres password
- `DB_PORT` — Postgres port (usually `5432`)
- `TELEGRAM_BOT_TOKEN` — Telegram bot token (if using bot features)
- `GOOGLE_API_KEY` — Google GenAI API key (AI modules)
- `PORT` — (optional) Render sets `$PORT` automatically; you can leave this unset
- `RENDER_URL` — (recommended) the public Render URL for this service (e.g. `https://your-backend.onrender.com`). Used by the CORS patch.

Important: Your code reads individual DB env vars (not `DATABASE_URL`). Set them exactly as listed above.

3. Managed Postgres (optional but recommended)

- In Render: create a managed PostgreSQL instance.
- After creation, copy the DB connection values into the service environment variables listed above.
- Run your database schema (`queries.sql`) against the Render Postgres instance. Use Render's database connection info and any client (psql, pgAdmin, DBeaver). Example `psql` command:

```
psql -h <DB_HOST> -U <DB_USER> -d <DB_NAME> -p <DB_PORT> -f queries.sql
```

- Verify tables: `resume`, `experience`, `education`, `skills`, `telegramusers` exist and columns match your queries file.

4. CORS setup

- Add `RENDER_URL` env var (your Render service URL) so the code's allowed origins include it (see Part A above). Alternatively, open the allowed origin list inline in code before deployment.
- If you have multiple frontend origins, list them in the code's allowedOrigins or use an env var like `ALLOWED_ORIGINS` (comma-separated) and parse it in code:

```
const allowedOrigins = (process.env.ALLOWED_ORIGINS || '').split(',').map(s => s.trim())
```

5. Telegram bot & AI modules

- Ensure `TELEGRAM_BOT_TOKEN` and `GOOGLE_API_KEY` are set. Bot uses polling; Render processes keep-alive are needed — Render supports background workers, but polling in a web service may still work. Monitor for long-term stability (webhooks are more robust for bots in production).
- Verify the AI modules can access `process.env.GOOGLE_API_KEY` at runtime.

6. Deployment & verification

- Enable Auto-Deploy (optional): Render can auto-deploy on Git push to the connected branch.
- After deploy, Render exposes a public URL like `https://your-backend.onrender.com`.
- Test endpoints (replace `<URL>` with your Render URL):
    - `GET https://<URL>/` → Should return your `public/index.html` content.
    - `GET https://<URL>/api/user-resumes?userEmail=<email>` → returns JSON array
    - `POST https://<URL>/api/user-resumes/upload/:id/:teleUser` (multipart/form-data) → test upload
    - `POST https://<URL>/api/user-resumes/ats/:id` (multipart/form-data) → triggers AI processing (requires GOOGLE_API_KEY)
    - `GET https://<URL>/api/user-resumes/fetchScore/:id` → returns `{ score, fetched }`
    - `GET https://<URL>/api/user-resumes/fetchRecommendations/:id` → returns `{ recommendations }`
    - `POST https://<URL>/api/feedbacks` → test feedback push
    - `GET https://<URL>/api/feedbacks` → returns current feedback array

7. Debugging tips

- Use Render logs (service dashboard) to view console output (e.g., DB connection logs, AI model logs, bot logs).
- If DB connection fails, confirm `DB_HOST` and `DB_PORT`. If you use SSL, the `pg.Client` in your code already sets `ssl: { rejectUnauthorized: false }`.
- If `fetchScore` endpoint hangs, check that `flashAI` actually sets `dataForAts.fetched = true` and that `GOOGLE_API_KEY` is valid and allowed to call the model.
- For Telegram bot polling issues, ensure the bot token is correct and there are no constraints on Render's execution model (consider running the bot in a dedicated background worker or switch to webhooks).

8. Post-deploy (optional)

- Set up monitoring/alerts in Render for service errors.
- Add a health-check endpoint if you want automatic restarts on failures.

Quick checklist to paste in Render service notes

- □ Push code to GitHub branch
- □ Create Web Service in Render
- □ Create/Postgres (if required) and set DB env vars
- □ Add TELEGRAM_BOT_TOKEN and GOOGLE_API_KEY env vars
- □ Add RENDER_URL env var (public service URL)
- □ Paste CORS patch and redeploy
- □ Run `queries.sql` on DB
- □ Test endpoints and bot functionality

Contribution Guidelines

Contributions from the community are welcome! Please follow these best practices:

1. **Fork the Repositories:** Create a personal copy of each repo on GitHub (frontend and/or backend) and clone it locally.

2. **Create a Feature Branch:** Use a descriptive branch name. For example:

   ```
   git checkout -b feature/add-ATS-feedback
   ```

3. **Make Changes:** Implement your feature or bugfix. For backend, follow the existing code style (ESLint is configured). For frontend, follow the React component and Tailwind CSS conventions.

4. **Testing:** Ensure the app runs locally and that your changes do not break existing functionality.

5. **Commit and Push:** Write clear commit messages. Push your branch to your GitHub fork.

6. **Pull Request:** Open a Pull Request against the `main` branch of the original repo. Describe your changes and why they're needed. For backend PRs, also mention any new API changes or DB migrations.

7. **Review:** The maintainer will review your PR. Respond to feedback or update as needed.

Some best practices:

- Keep code modular and well-documented.
- Handle errors gracefully (e.g. invalid input on backend, UI error messages).
- Respect the existing coding conventions (indentation, naming).
- Do not commit sensitive data (never hard-code API keys or passwords).

For major changes, it's a good idea to open an issue first to discuss plans with the maintainer. This ensures everyone is aligned and avoids duplicate work.

> *Support:* If you use this project, consider starring the repos on GitHub and reporting any issues you encounter. The author welcomes questions and suggestions via email or GitHub Discussions.

License & Third-Party Credits

- **License:** Kenshi Resumes (both frontend and backend) is released under the **MIT License**. This allows anyone to use, copy, and modify the code freely, as long as the license notice is included.
- **Third-Party Software:** The project depends on several open-source libraries (React, Express, Tailwind, etc.) as listed in `package.json`. These libraries are each under their own licenses (MIT, BSD, etc.).
- **AI Model:** Kenshi Resumes uses Google's Gemini Flash 2.0 via API. Gemini is a proprietary model by Google DeepMind.
- **Platforms:** The deployed app uses Vercel (for frontend) and Railway (for backend). Both are credited for providing hosting infrastructure.
- **Inspirations:** No direct forks, but features like live editing and AI generation are influenced by modern resume builder tools. The Telegram bot uses the standard Bot API.

Please ensure credit is given if you reuse substantial parts of this project. The MIT License text is included in each repo's `LICENSE` file.

Sources & References: Kenshi Resumes documentation is based on the project's GitHub READMEs and additional information about technologies (React, Node.js, Express, PostgreSQL, Gemini AI, Vercel, Railway) and ATS systems. All cited lines refer to these sources.