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 Railway (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.)

Resume Endpoints

- **POST `/api/user-resumes`** – *Generate and save a new resume.*
  - **Request Body:** JSON object with a `data` field. Example:

    ```
    {
      "data": {
        "title": "Senior Software Engineer",
        "documentId": "some-unique-uuid",
        "userEmail": "user@example.com",
        "userName": "Jane Doe"
      }
    }
    ```

  - **Function:** The server uses these details (name, email, title) to prompt Gemini and generate resume content. It saves the generated resume in the database (along with `documentId` and metadata).
  - **Response:** On success, returns `{ data: { title, documentId, userEmail, userName } }` (echoing the saved info).
  - *Example Response:*

    ```
    {
      "data": {
        "title": "Senior Software Engineer",
        "documentId": "generated-uuid",
        "userEmail": "jane@example.com",
        "userName": "Jane Doe"
      }
    }
    ```

- **GET `/api/user-resumes`** – *Retrieve a previously generated resume.*
  - **Query Parameters:** `userEmail` (e.g. `/api/user-resumes?userEmail=jane@example.com`).
  - **Function:** Looks up the resume(s) associated with that email in the database.
  - **Response:** Returns the stored resume data (e.g. title, documentId, userName, plus the AI-generated content) for the user. (If multiple resumes per email are allowed, it might return all or the latest; refer to backend code for specifics.)

Flash Recommendations

- **GET `/api/user-resumes/fetchRecommendations/:id`** – *Get ATS improvement tips.*
  - **Route Parameters:** `id` is the document ID of the resume (the same `documentId` saved earlier).
  - **Function:** The server analyzes the specified resume (by ID) and returns suggestions to improve the ATS score (e.g., add keywords, reformat sections).
  - **Response:** Likely a JSON with an array of recommendation strings or tips.

Telegram Bot Routes (Optional)

These endpoints are used by the **Kenshi Resumes Telegram Bot** to deliver resumes to users on Telegram. They are not needed by the web frontend, but are documented for completeness.

- **POST `/api/user-resumes/upload/:id/:teleUser`** – *Send resume to Telegram user.*
  - **Route Parameters:** `id` (document ID of resume), `teleUser` (Telegram user ID or chat).
  - **Function:** Triggered by the bot when a user requests their resume. The server fetches the resume by ID, then uses the Telegram Bot API to send the resume file to the given user.
  - **Response:** Likely a status message indicating success/failure.

Each endpoint returns standard HTTP status codes (200 for success, 4xx for client errors, etc.). As no authentication is enforced, the backend trusts the provided `userEmail`, `documentId`, etc. In a production scenario, you might add token checks (e.g. JWT in headers) to protect these routes.

For reference, see the API spec from the README.

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 (Railway)

The backend can be deployed on **Railway**, a cloud platform for application deployment. Steps:

1. Push the backend code to GitHub.
2. Sign up at Railway (free tier available) and create a new project.
3. Connect the GitHub repo. Railway will detect the Node.js project and install dependencies.
4. In Railway, add a PostgreSQL plugin or set `DATABASE_URL` to your database. Railway can provision a managed Postgres instance, and will set the `DATABASE_URL` environment variable for you.
5. Add environment variables in Railway (GEMINI_API_KEY, TELEGRAM_BOT_TOKEN if needed).
6. Deploy: Railway will build and start the app (e.g. running `npm start`). It provides a public URL like `https://yourapp.up.railway.app`.

Railway supports continuous deployment (push to GitHub → auto-deploy), environment management (storing API keys), and includes features like automatic SSL. This matches Kenshi Resumes' needs for zero-downtime hosting and easy updates.

> **Tip:** The backend repo's README (see *Steps.txt*) may contain additional deployment notes. Ensure that the `queries.sql` is run on the Railway Postgres instance (you can use the Railway console or PGAdmin).

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.