**React Resume Builder Documentation**

**Overview**

The **resume-builder-js** project is a Resume Builder app built with React and powered by Vite for fast development and build processes. It uses **react-hook-form** for efficient form management and validation, with **Zod** handling schema-based validation. Styling is done with **TailwindCSS**, and the app includes **ESLint** for linting to ensure clean code. The package includes necessary dependencies such as **react-icons** for UI elements and **@hookform/resolvers** for integration with Zod. The project follows modern JavaScript module practices, providing scripts for starting the server (vite), building the project (vite build), and previewing the build (vite preview). It is structured for scalability, maintaining a clean, modular, and reusable approach throughout the app.

* **App.js**: Initializes the form using react-hook-form, sets up context, and renders the main form and live preview.
* **MainForm.js**: Contains the form structure, including personal information, professional summary, and work experience sections.
* **LivePreview.js**: Displays real-time form data by watching the form's state.

**Component Breakdown**

**1. App.js**

**Responsibilities:**

**Key Code:**



**Flow:**

****

**2. MainForm.js******

****

**3. LivePreview.js**

****

**Data Flow**

****

**State Management**

****

**Validation**

****

**How to Add New Sections**

1. Define the schema in formTypes.js.
2. Add new form fields in **MainForm.js**.
3. Update the combined schema in **App.js**.
4. Reflect the new data in **LivePreview.js**.

**Running the App**



**Conclusion**

This React app is designed with extensibility in mind, utilizing **react-hook-form** for efficient form management, ensuring minimal re-renders and a smooth user experience. **Zod** is employed for robust, schema-driven validation, providing a clear structure for handling input constraints and errors. Shared state management is handled through **React Context**, which allows for seamless data flow across components without the need for prop drilling. The app follows an **atomic design approach**, which breaks down the UI into smaller, reusable components, promoting modularity and making it easy to scale and maintain. This approach not only enhances code reusability but also makes it simpler for new developers to contribute to the project. By adhering to a schema-driven validation structure and leveraging atomic design principles, new developers can easily extend the form and its functionality, ensuring that future updates remain consistent and manageable