**Distance Calculator Application Documentation**

**High-Level Application Design**

For this project, I used **monolithic architecture** for simplicity, as it's easier to develop, deploy, and manage for a small-scale project. However, if scalability demands increase, a **microservices** approach could be considered for modularity.

* **Backend Architecture:** A RESTful API using **Node.js with Express.js** to handle requests and interact with a database.
* The RESTful API structure with clear endpoints such as /history (To retrieve stored data from DB), /history/new (To post/store new search query into the DB). Alternatively, GraphQL could be considered for flexible data querying.
* **Database Choice:** **NoSQL (MongoDB)**
  + **Why?** Since addresses and location data vary in structure, and there are no complex relations between data, a document-based database like MongoDB provides flexibility. Also,

It supports geospatial queries, which are useful for location-based applications.

* **Scalability Considerations:**
  + Deploy via containerization (Docker) for easy scaling.
  + Including load balancing, horizontal scaling, caching mechanisms, and asynchronous processing using message queues (e.g., RabbitMQ, Kafka) can improve the scalability of the project. Since this is a very small-scale project, that means a smaller number of users = less number of incoming requests. So, I won’t think that scalability will be a blocker for this project. So, I didn’t implement these approaches.

**Development Workflow**

**Codebase Structure:**

* Follow MVC (Model-View-Controller) for clarity.
* **For Backend**: Separate concerns- /routes, /controllers, /models.
* **For Frontend**: Separate folder structure for each individual component (e.g.: /Heading/Heading.jsx, /Heading/Heading.css).

**Git branching strategy**:

* I followed a Git Flow branching strategy:

**For Backend:**

* **main** for production-ready code.
* **Mongo-Schema-Setup** for adding schema models DB.
* **Retrieve-GeoCodes** for implementing Nominatim API for retrieval of Geocodes.
* **Api-Routes** for implementation of GET & POST routes.
* **Error-handling** for implementing basic error handling and basic logging.
* **Code-Labeling** for adding description for every implementation.
* **Code-Restructure** for restructuring the code into MVC structure.

**For Frontend:**

* **main** for production-ready code.
* **calculator-component** implemented calculator component with all inputs components & buttons.
* **History-Page** added history page component to display the query data which is coming from backend DB.
* **Basic-Error-Handling** for implementation of error handlers in case of API failure or invalid address.
* **Code-Labeling** added necessary information about the functionality and logic.
* **Testing** implemented basic unit test cases for components like History, CalculatorBox, APP, and Heading components.

**Testing & Documentation:**

* Used **Vitest** for unit testing React frontend.
* Used **Postman for** backend REST APIs.
* Prepared documentation for project and step by step instructions to setup project.

**Security & Error Handling**

**Security Measures:**

* Implemented basic logic for Input validation. But there are other ways to validate input (e.g. Joi) to prevent SQL/NoSQL injection.

**Error Handling:**

* Use **Express middleware** to centralize error responses.
* Implemented **Express middleware** for basic logging for debugging.

Application Live link: <https://distancecalculatorapp.netlify.app/>