Measuring The pulse of Prosperity:

An index of Economic Freedom Analysis

1. Introduction

- **Project Title:** Measuring The Pulse of Prosperity: An index of Economic Freedom Analysis
- **Team Members:** G. Pooja Sithrubi, M. Gayatri, Kotte Munikumar, K. Pavani

2. Project Overview

- **Purpose**:

  Economic freedom is the fundamental right of every human to control his or her own labour and property. In an economically free society, individuals are free to work, produce, consume, and invest in any way they please. In economically free societies, governments allow labour, capital, and goods to move freely, and refrain from coercion or constraint of liberty beyond the extent necessary to protect and maintain liberty itself.

- **Features**

  We measure economic freedom based on 12 quantitative and qualitative factors, grouped into four broad categories, or pillars, of economic freedom:
  Rule of lawproperty rights, government integrity, judicial effectiveness)
  Government size (government spending, tax burden, fiscal health)
  Regulatory efficiency (business freedom, labor freedom, monetary freedom)
  Open markets (trade freedom, investment freedom, financial freedom)

3. Architecture

- **Frontend:** Front end architecture using React

Objective: To visualize, compare, and explore the Index of Economic Freedom data by country, category, and year — empowering users with insight through interactive charts, maps, and dashboards.

```
React Frontend

│

├── App Component (Root)

│   ├── Layout (Navbar, Footer, Sidebar)

│   ├── Routes (React Router)

│   │   ├── Home

│   │   ├── Rankings

│   │   ├── Country Profile
│   │   ├── Compare Countries
│   │   ├── About
│   │   └── Error / Not Found
    │   └── State Management (Zustand / Redux Toolkit or Context API)
```

- **Backend:** Backend architecture using NodeJs and ExpressJS

  Objective: Build a secure, modular, and scalable REST API that powers the React frontend with real-time economic freedom data.

```
/backend
│
├── /src
│   ├── /controllers      → Logic for handling API requests
│   ├── /routes           → All route definitions (REST APIs)
│   ├── /models           → MongoDB schemas or SQL models
│   ├── /services         → Business logic and utilities
│   ├── /middlewares      → Error handling, logging, auth
│   ├── /config           → DB, environment, external APIs
│   ├── /utils            → Helpers (e.g., formatter, validator)
│   └── server.js         → Express app entry point
│
├── .env                  → Environment variables
├── package.json
    └── README.md
```

- **Database:** Database schema and interaction with MongoDB

```
// models/Country.js
const mongoose = require('mongoose');

const scoreSchema = new mongoose.Schema({
  year: { type: Number, required: true },
  totalScore: { type: Number, required: true },
  pillars: {
    ruleOfLaw: {
      propertyRights: Number,
      governmentIntegrity: Number,
      judicialEffectiveness: Number
    },
    governmentSize: {
      governmentSpending: Number,
      taxBurden: Number,
      fiscalHealth: Number
    },
    regulatoryEfficiency: {
      businessFreedom: Number,
      laborFreedom: Number,
      monetaryFreedom: Number
    },
    openMarkets: {
      tradeFreedom: Number,
      investmentFreedom: Number,
      financialFreedom: Number
    }
  }
});

const countrySchema = new mongoose.Schema({
  name: { type: String, required: true },
  code: { type: String, required: true }, // ISO 2 or 3
  region: String,
  data: [scoreSchema] // Historical data for multiple years
});

      module.exports = mongoose.model('Country', countrySchema);
```

4. Setup Instructions

- **Prerequisites:**

To build the full-stack Economic Freedom Index application, you'll need a set of software dependencies for both the frontend and backend. On the frontend, the application uses React along with core libraries like `react-dom` and `react-router-dom` for routing. For fetching data from the backend, `axios` is used, while visualization is powered by `recharts`, `chart.js`, or `react-simple-maps` with `d3-geo` for interactive maps. Styling is handled using either `tailwindcss` or `Material UI`, and data fetching optimization can be managed through `react-query` or `swr`. The project can be bootstrapped using either Vite or Create React App.

On the backend, Node.js and Express.js form the foundation. Dependencies such as `mongoose` are used to model and interact with the MongoDB database. Middleware like `cors` enables cross-origin requests, while `dotenv` manages environment variables. For request security and validation, packages like `helmet`, `express-validator`, and `body-parser` are included. If the application has admin login functionality, `bcryptjs` is used for password hashing, and `jsonwebtoken` is used for secure token-based authentication. Development tools such as `nodemon` enable auto-reloading, and `morgan` logs HTTP requests.

For the database, MongoDB is used as the NoSQL data store, and it can be hosted locally or through MongoDB Atlas for cloud access. Interaction with MongoDB is managed through the `mongoose` ODM. Optional tools like the MongoDB CLI can assist in importing/exporting data dumps.

For testing and development, the stack can include `jest` and `supertest` for unit and integration testing, and `eslint` with `prettier` for consistent code formatting. If desired, `swagger-ui-express` can be used to automatically generate API documentation. For deployment, frontend and backend can be hosted using platforms like

Netlify, Vercel, or Render, depending on whether you prefer serverless or container-based hosting.

- **Installation:**

To get started with the Economic Freedom Index application, first clone the project repository using git clone and navigate into the main folder. The project is typically divided into two directories: client for the React frontend and server for the Node.js backend. Begin by setting up the frontend. Navigate into the client directory and run npm install to install all required dependencies such as React, Axios, Recharts, and Tailwind CSS. Next, create a .env file in the same folder and define the environment variable VITE_API_BASE_URL, pointing to your backend API, for example: http://localhost:5000/api. You can then start the frontend server using npm run dev, which will usually launch on http://localhost:5173. After setting up the frontend, switch to the backend by navigating to the server directory. Run npm install to install backend dependencies like Express, Mongoose, CORS, Helmet, and Dotenv. Create a .env file here as well, and add key variables such as PORT, MONGODB_URI (your MongoDB Atlas or local connection string), and JWT_SECRET for authentication tokens. Once the .env file is configured, start the backend using npm run dev, which should launch the API server on http://localhost:5000. To confirm everything is working, visit the frontend in your browser and verify that it successfully fetches data from the backend by checking the network requests. Optionally, you can use a tool like concurrently to run both servers in parallel using a single command from the root folder, which simplifies development. This setup ensures the frontend and backend are properly connected and ready for development or deployment.

5. Folder Structure

- **Client:**

The React frontend of the Economic Freedom Index application is structured in a modular and scalable way to support multiple pages, components, and interactions efficiently. At the root level, the `App.jsx` or `App.tsx` file serves as the main entry point, setting up the layout and routing using `react-router-dom`. Common components like the `Navbar`, `Footer`, and a consistent page layout are included here to wrap around all route views. The frontend follows a clear directory-based organization under a `/src` folder, which typically includes subfolders like `/components`, `/pages`, `/services`, `/hooks`, and `/assets`.

The `/components` folder contains reusable UI elements such as `CountryCard`, `PillarChart`, `RadarChart`, `DataTable`, and `MapView`, each built as small, maintainable functional components. The `/pages` directory includes main route-based views like `Home`, `Rankings`, `CountryProfile`, `Compare`, and `About`, each one responsible for rendering a complete page experience. For state and API management, a `/services` folder holds API utility functions, often built using Axios, while `react-query` or SWR can be used to handle fetching, caching, and loading states.

Styling is handled using `Tailwind CSS` or another preferred UI library, with styles applied directly in JSX or through utility classes. If needed, context providers or global state management (like Zustand or Redux Toolkit) are set up in a `/store` or `/context` folder to share data like selected countries or filters across components.

Routes are declared in a `Routes.jsx` or inside `App.jsx`, mapping URLs to their respective page components, and `BrowserRouter` is used to manage navigation.

The frontend interacts with the backend via environment variables defined in a `.env` file (like `VITE_API_BASE_URL`), which enables dynamic API endpoint management. Overall, the structure is designed to be clean, scalable, and easy to maintain, supporting efficient development and future feature expansion.

- **Server:**

The Node.js backend is organized in a modular structure using Express.js to handle API routing and server logic. At the root of the server folder, there is typically a main entry file named server.js or index.js, which initializes the Express app, sets up middleware like CORS and JSON parsing, and starts the server by listening on a specified port. The backend follows a clean MVC (Model-View-Controller) pattern and includes dedicated folders such as /routes, /controllers, /models, /middlewares, /services, and /config.
The /routes folder contains all route definitions, grouped by feature. For example, countryRoutes.js handles endpoints like /api/countries and /api/compare, and is responsible for directing requests to the correct controller functions. These controller functions live in the /controllers folder, where each file contains logic for handling specific API operations like retrieving data, filtering results, or updating records. For instance, countryController.js might include functions like getCountryByCode or compareCountries.
The /models folder contains MongoDB schemas defined using Mongoose. For example, Country.js defines the schema for countries and their yearly economic freedom data, including nested objects for pillars such as rule of law and open markets. If user authentication is needed, a User.js schema is also included. The /middlewares folder contains reusable functions like authentication checks, error handling, and logging utilities. In the /services folder, business logic that doesn't belong directly in the controllers—like data transformation or external API integration—is abstracted for cleaner code.
The /config folder is used to manage configuration settings, such as connecting to MongoDB using a separate db.js file and loading environment variables using dotenv. These variables (like PORT, MONGODB_URI, and JWT_SECRET) are stored in a .env file at the root level. Logging tools like morgan may also be set up here for development diagnostics.
This organization allows the backend to be modular, scalable, and maintainable, with each concern separated logically. It supports a well-structured API for the frontend to interact with, while making the backend easy to extend—for example, to add user roles, file uploads, or additional data sources.

6. Running the Application

- Provide commands to start the frontend and backend servers locally.
  - **Frontend:** `npm start` in the client directory.

1. Open terminal.
2. Navigate to the client folder- cd client
3. Start the development server:- npm start

  - **Backend:** `npm start` in the server directory.

1. Open terminal.
2. Navigate to the server directory: cd server
3. Then run: npm start

7. API Documentation

he backend of the Economic Freedom Index application exposes a set of RESTful API endpoints built with Node.js and Express.js. The `/api/countries` endpoint allows clients to retrieve a list of all countries along with their basic information, and it supports optional query parameters such as `region` and `year` for filtering. To view detailed information about a specific country, including its economic freedom data over the years, the client can use the `/api/countries/:code` endpoint, where `:code` is the ISO country code (like `IN` for India). Historical economic scores for a single country can be fetched using the `/api/countries/:code/history` endpoint. For side-by-side comparisons, the backend offers the `/api/compare` endpoint, which takes two query parameters, `c1` and `c2`, representing the ISO codes of the countries to compare, and returns their overall scores and pillar breakdowns. In addition, the `/api/metrics/summary` endpoint returns global statistics such as top-performing countries, average scores, and the lowest scores for that year. For dynamic UI filtering, the `/api/filters` endpoint provides available options like regions, years, and economic pillars. An optional secured admin endpoint, `/api/admin/upload`, allows authorized users to upload or update country-level economic data via a POST request containing the country code, year, total score, and detailed pillar values. Finally, the backend includes a fallback route to handle undefined endpoints, which returns a standard 404 error response. If authentication is implemented, additional routes like `/api/auth/login` and `/api/auth/register` would support user login and account creation. Overall, this set of endpoints is designed to support data-driven visualizations and interactions on the frontend while keeping the backend clean, modular, and secure.

The backend API of the Economic Freedom Index application, built using Node.js and Express.js, provides a RESTful interface with multiple endpoints for accessing and managing economic data. The GET /api/countries endpoint retrieves a list of all available countries, optionally filtered by query parameters such as region (e.g., Asia) and year (e.g., 2022). For instance, calling /api/countries?region=Asia&year=2022 returns a JSON array of countries with basic details like { "code": "IN", "name": "India", "region": "Asia" }. To access detailed data for a single country, the GET /api/countries/:code endpoint accepts a path parameter (:code) such as IN and responds with an object containing the country's name, code, region, and yearly data with pillar-wise scores. For example, /api/countries/IN would return an object with yearly breakdowns like total score and the four main economic pillars.
To get a historical overview, GET /api/countries/:code/history fetches a timeline of scores for a given country, returning an array such as [{ "year": 2020, "totalScore": 55.0 }, { "year": 2022, "totalScore": 56.5 }]. For comparing two countries, the GET /api/compare?c1=IN&c2=US endpoint returns a JSON object comparing both nations' economic freedom scores, formatted as { "country1": { "code": "IN", "totalScore": 56.5 }, "country2": { "code": "US", "totalScore": 76.3 } }. The GET /api/metrics/summary endpoint provides high-level insights such as the top-performing countries, average scores, and lowest score globally, useful for overview dashboards. For building dynamic filters on the frontend, the GET /api/filters endpoint returns arrays of available years, regions, and economic pillars (e.g., "regions": ["Asia", "Europe"], "years": [2020, 2021, 2022]).
To update or insert new data, especially for administrative purposes, the backend exposes a secured POST /api/admin/upload endpoint. This expects a JSON body with fields such as code, year, totalScore, and a nested pillars object (e.g., "pillars": { "ruleOfLaw": { "propertyRights": 60 } }). The response confirms success with { "message": "Data inserted/updated successfully." }. If user authentication is required, optional endpoints like POST /api/auth/login and POST /api/auth/register can be implemented to handle login and account creation. All undefined routes are gracefully handled with a 404 fallback response:

{ "error": "Route not found" }. These endpoints are designed to support rich data visualizations, comparisons, and administrative operations in a scalable and secure way.

8. Authentication

In this project, authentication and authorization are implemented using JWT (JSON Web Tokens) to ensure secure access to protected backend routes. When a user registers through the POST /api/auth/register endpoint, their credentials — typically username, email, and password — are securely stored in MongoDB, with the password hashed using bcrypt for protection. Upon login via POST /api/auth/login, the server verifies the user's credentials and, if valid, generates a signed JWT token using a secret key stored in environment variables. This token is then sent to the client, which stores it (usually in localStorage or as an HTTP-only cookie) and includes it in the Authorization header (Bearer <token>) with subsequent requests to protected routes.

On the backend, a custom authentication middleware intercepts these requests and verifies the JWT token. If the token is valid, the user's decoded information is attached to the request object (req.user), allowing downstream controllers to identify the requester. Authorization is handled based on user roles (like "admin" or "editor") defined in the user schema. Specific routes, such as POST /api/admin/upload, check whether the req.user.role matches the required permission level before allowing access. If authentication fails or the user lacks the proper role, the server responds with a 401 Unauthorized or 403 Forbidden error, respectively. This token-based approach ensures that only authenticated users — and only those with appropriate privileges — can access sensitive data or perform administrative actions. The project uses JWT (JSON Web Token)-based stateless authentication, which eliminates the need for server-side sessions and enables secure communication between the frontend and backend. When a user successfully logs in via POST /api/auth/login, the backend generates a signed JWT using a secret key defined in environment variables (e.g., JWT_SECRET). This token typically contains a payload with the user's ID, email, and role (e.g., admin, viewer) and is signed with an expiration time (exp), such as 1 hour. The token is sent to the client, which stores it either in localStorage (if security is less strict) or in HTTP-only cookies (if CSRF protection is needed). In every subsequent request to protected endpoints (e.g., /api/admin/upload), the client includes the token in the Authorization header in the format: Authorization: Bearer <token>.
The backend includes a JWT verification middleware that inspects this token on each request. If valid and unexpired, the middleware decodes the token and attaches the user object (e.g., req.user = { id, email, role }) to the request, enabling role-based access checks in route handlers. Unlike traditional session-based systems that store session data in memory or databases, this token-based system keeps the backend stateless and more scalable. For token expiration and refresh strategies, the system can optionally implement a refresh token flow where short-lived access tokens are complemented with long-lived refresh tokens stored securely and used to issue new access tokens. This entire approach ensures secure, scalable, and efficient handling of authentication and authorization throught the application

9. Testing

The testing strategy for this project follows a layered approach, covering unit testing, integration testing, and end-to-end (E2E) testing to ensure robustness and reliability across both frontend and backend components. On the backend (Node.js + Express) side, Jest is used as the primary testing framework for unit tests, where individual functions and services like authentication logic, data parsing, and utility functions are tested in isolation. For integration testing—especially routes and controllers—Supertest is used to simulate HTTP requests and validate API responses, status codes, and authentication flows. On the frontend (React), React Testing Library is utilized for testing UI components in a way that reflects how users interact with them, ensuring elements render correctly and respond properly to events like clicks and form inputs. For mock data and functions, Mock Service Worker (MSW) is used to simulate API calls during component tests.

To cover end-to-end scenarios, tools like Cypress are recommended, which allow developers to simulate real user interactions across the full stack—for example, logging in, navigating to the country comparison page, or uploading data through the admin panel. Test coverage is monitored using coverage reports generated by Jest, ensuring that critical logic and user flows are tested. Additionally, tests are organized in `__tests__` directories and run automatically via npm scripts, which can be integrated with CI/CD pipelines like GitHub Actions or GitLab CI for automated validation during deployments. This layered strategy ensures code quality, catches regressions early, and provides confidence in the application's functionality and security.

10. Known Issues

As of the current version, there are a few known issues and bugs that are being tracked for future resolution. On the frontend, when switching rapidly between different country comparisons, the chart components occasionally flash outdated data due to React state lag, which can be mitigated by introducing debouncing or useEffect cleanup. In mobile view, certain UI elements — such as dropdown filters or the radar chart — may overflow or misalign due to limited responsive styling, particularly on smaller devices. On the backend, some older entries imported via the /api/admin/upload endpoint may fail validation if optional fields like monetaryFreedom or fiscalHealth are missing, leading to partial data ingestion. There's also an intermittent issue where MongoDB queries using special characters in country names (e.g., Côte d'Ivoire) can throw encoding-related errors unless sanitized properly. Additionally, while authentication via JWT is functional, expired tokens currently do not trigger a user-friendly logout or redirect on the frontend, requiring users to refresh manually. These issues are under review, and contributors are encouraged to check the GitHub issues tab or contact the maintainer team before reporting duplicate bugs.

11. Future Enhancements

To extend the capabilities and improve user engagement, several future features and enhancements are planned for the Economic Freedom Index project. One key addition would be an interactive global map using libraries like react-leaflet or D3.js, allowing users to visually explore scores by country and region. Another improvement is the integration of data trend visualization, enabling users to track a country's economic freedom over time with dynamic line charts. The platform could also benefit from user accounts with saved views, where users can bookmark favorite countries or create custom comparisons. For administrative purposes, an advanced data management dashboard with bulk upload, CSV export, and role-based permissions can streamline backend operations. On the technical side, implementing refresh token-based authentication, rate limiting, and input sanitization would enhance security and scalability. Lastly, integrating internationalization (i18n) for multilingual support and deploying a progressive web app (PWA) version for offline access can make the platform more accessible globally. These features are aimed at improving usability, performance, and global impact while maintaining a strong, secure architecture.