Project Documentation

Overview:

This project implements a data sourcing and visualization web application that fulfills the original assignment requirements:

1. Task Creation Frontend: Users can create a new “fetch” or “analysis” task by providing filtering parameters for two external data sources.

2. Job Queue Simulation: The system simulates an in-memory queue where tasks are assigned statuses (pending → in_progress → completed) with artificial delays to mimic real-world data gathering.

3. Database Integration (SQLite): After data is fetched, it’s merged into a relational database, where each row is associated with a task_name

4. Visualization: The frontend uses React and D3.js to display interactive charts (bar charts, multiple grouped charts, legends, etc.) based on the data.

This documentation explains how to set up technologies used, and how each part of the application works. Where relevant, it calls back to the assignment’s requirements (e.g., job queue, unified schema, and data storage).

Technologies Used

Backend

- **Python 3.8+**
- **FastAPI**: For building RESTful endpoints.
- **SQLAlchemy**: ORM for database interactions (SQLite in this example).
- **SQLite**: Lightweight relational database to store fetched/merged data.
- **queue / list**: Python’s in-memory data structure to simulate a job queue.

Frontend

- **React (JSX)**: Main framework for building the user interface.
- **Bootstrap**: For layout & utility classes (responsiveness).

- **Material UI**: Used selectively to add modern UI components (like MUI <Select> and advanced styling).
- **D3.js**: For custom, interactive data visualizations (grouped bar charts, tooltips, legends).

How to Set Up & Run:

Backend Setup:

1. **Install Python Dependencies**

```
cd backend
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt
```

2. **Run the Server**

```
uvicorn src.main:app --reload
```

The API is now available at http://localhost:8000.

Key Libraries:

fastapi for building APIs.

sqlalchemy for ORM logic.

queue or a simple list to simulate the job queue.

Endpoints:

- POST /api/v1/create_task/{task_name}:

Accepts filter parameters in the request body (ProviderA, ProviderB).

Starts a job in the queue (status: pending) → after delay, fetch data → status: in_progress → after final delay, store data → status: completed.

```
POST http://localhost:8000/api/v1/create_task/queue_test
Content-Type: application/json

{
 "provider_A": {
   "year_from": "2024",
   "year_to": "2025",
   "countries": ["USA"],
   "threat_levels": [4, 5]
 },
 "provider_B": {
   "year_from": "2024",
   "year_to": "2025",
   "countries": ["China", "India"],
   "severity": [5]
 }
}
```

- GET /api/v1/get_task/{task_name}: → Returns all data rows for a given task name.
- GET /api/v1/get_task_names: → Returns all distinct tasks stored in the DB.

Database:

db_threat.db (SQLite file).

Table structure (example: table_threat) for storing:

```
CREATE TABLE table_threat (
  country TEXT,
  discovery_date TEXT,
  source TEXT,
  risk_level INTEGER,
  task_num TEXT
);
```

In the code, we automatically create or reflect tables if needed.

Job Queue Simulation

```
tasks_queue = []
```

- When a new task is submitted:
    1. Append to `tasks_queue` with status = `pending`.
    2. After 5s, filter the data and move to `in_progress`.
    3. Fetch data, store in DB, after next delay → `completed`.

Frontend Setup:

1. **Install Node Dependencies**

```
cd frontend
npm install
```

2. **Run the React App**

```
npm run dev
```

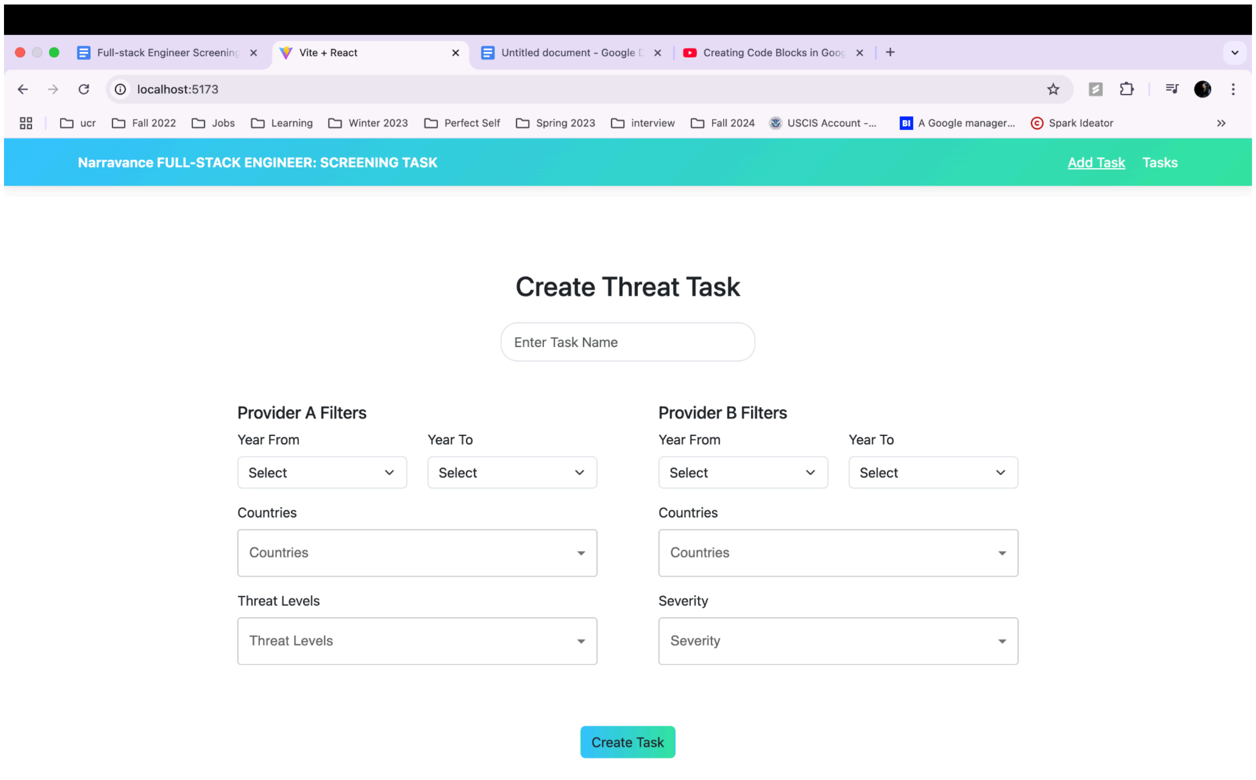
Your app runs at `http://localhost:3000`.

Key Libraries:

- **React**: For creating components & pages.
- **Bootstrap**: Utility classes & grid system.
- **Material UI**: Modern components
- **D3.js**: Data-driven visualizations with grouped bar charts, legends, and tooltips.

Important Components & Pages:

- **CreateTaskPage.jsx**:
    - Lets user input task_name and filters for Provider A & B.
    - On submit → calls createTask(taskName, filterPayload).
- **TasksPage.jsx**:
    - Displays a sidebar with all tasks from GET /api/v1/get_task_names.
    - On selecting a task, calls GET /api/v1/get_task/{task_name}.
    - Visualizes the returned data with **D3** charts (grouped bar for "Country vs Year," multiple bar charts for "Country vs Risk Level," etc.)



Narravance FULL-STACK ENGINEER: SCREENING TASK

Add Task Tasks

Create Threat Task

Enter Task Name

Provider A Filters

Year From: Select

Year To: Select

Countries: Countries

Threat Levels: Threat Levels

Provider B Filters

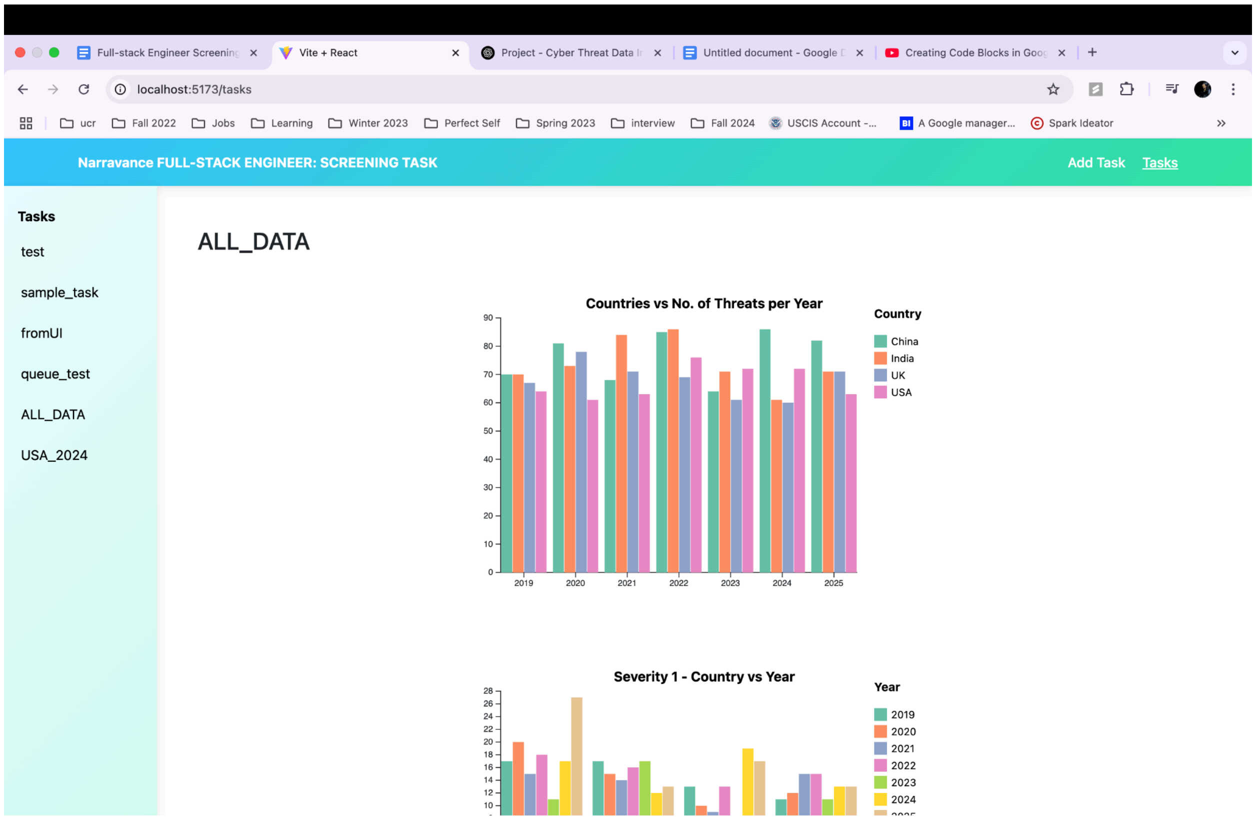
Year From: Select

Year To: Select

Countries: Countries

Severity: Severity

Create Task



Narravance FULL-STACK ENGINEER: SCREENING TASK

Add Task Tasks

Tasks

test

sample_task

fromUI

queue_test

ALL_DATA

USA_2024

ALL_DATA

Countries vs No. of Threats per Year

Severity 1 - Country vs Year

How the Requirements Are Met

1. **Task Creation Frontend**
    - The user picks filters (year range, countries, severity, etc.) for **Provider A** and **Provider B**, similar to the assignment's requirement (e.g., "sales from 2023 to 2025" with brand restrictions).
    - This effectively merges data from two sources (JSON + CSV for "threat alerts").
2. **Job Queue Simulation**
    - A simple list or `queue.Queue()` is used to simulate the "pending → in_progress → completed" pipeline with artificial delays.
    - During that time, I also fetched from external data sources or local CSV/JSON files.
3. **Database**
    - Data retrieved from the two sources is merged and stored in a single relational schema (e.g., `table_threat`) under a `task_name`.
    - `SQLAlchemy` is used to define or reflect tables, and to insert the filtered/merged data.
4. **Visualization**
    - Once the data is in the DB, the user can browse tasks in the "Tasks" view.
    - On selection, the data is fetched from the backend and displayed in **D3** bar charts:
        - "Country vs. Year" with grouped bars by country
        - "Country vs. Risk Level" with multiple sub-charts or grouped structure