Weather and Analytics Dashboard Project Documentation

Project Overview

The **Weather and Analytics Dashboard** is a full-stack web application that aggregates weather information, news updates, stock market data, and movie analytics into a single interactive interface. The purpose of the project is to provide users with a personalized dashboard for daily information and insights across these domains. Key high-level features include:

- **Real-Time Weather:** Current weather conditions and forecasts for multiple cities, visualized on a 3D interactive globe and in text form.
- **Live News Updates:** A stream of the latest news headlines delivered in real-time (using Server-Sent Events), so the content is always up to date.
- **Stock Market Overview:** Live stock price updates for selected tickers via WebSocket streaming, giving users near real-time market data.
- **Movie Analytics:** Highlights of trending or upcoming movies fetched from The Movie Database (TMDB), including posters and ratings.
- **User Personalization:** Secure Google-based authentication (via NextAuth) to sign in users, possibly allowing personalized settings (favorite cities, preferred stocks, etc.).
- **Responsive UI:** A dynamic dashboard layout that adapts to different screen sizes (desktop, tablet, mobile) and supports both light and dark themes for optimal usability.

Target Users: The application is aimed at tech-savvy users or enthusiasts who want a one-stop dashboard for various data (weather, news, stocks, movies) in real-time. It can also serve as a demonstration of integrating multiple third-party services and real-time technologies in a Next.js application, making it relevant for developers as a reference project.

Overall, this project showcases how modern web technologies can be combined to create a rich, interactive dashboard experience, pulling in diverse data sources and presenting them in a cohesive UI.

Tech Stack & Tools

This project is built with a modern **full-stack JavaScript** approach, leveraging the following technologies and tools:

- **Next.js (React & Node.js):** Next.js is the core framework for the application, providing a hybrid of server-side rendering and client-side capabilities. It enables building the UI with React components and handles routing and serverless API routes for backend logic. Next.js allows seamless integration of frontend and backend code in one project, and supports features like image optimization and fast SSR/SSG for performance.

- **TypeScript:** The entire codebase is written in TypeScript, ensuring static type checking and reducing runtime errors. TypeScript adds robust typing to the project's React components, Redux store, and API calls, improving developer productivity and code reliability.

- **Redux Toolkit & RTK Query:** For state management, we use Redux Toolkit, specifically its RTK Query feature for data fetching and caching. RTK Query is a powerful data-fetching and client-side caching library built on Redux Toolkit that simplifies API calls by managing loading state, caching responses, and reducing boilerplate [1]. This helps keep the app responsive and avoids unnecessary repeat requests by caching results.

- **NextAuth (Authentication):** User authentication is implemented using **NextAuth.js**, a secure and easy-to-use authentication library for Next.js. NextAuth is configured with a Google OAuth provider, allowing users to sign in with their Google accounts without building auth from scratch. It manages sessions with JWT or cookies and integrates with Next.js API routes for callbacks.

- **Third-Party APIs:** The dashboard integrates several external APIs:

- **Weather API:** Provides real-time weather data (current conditions and forecasts) for cities worldwide. For example, the project can use WeatherAPI.com which offers weather info for millions of locations [2] (or a similar provider like OpenWeatherMap). A valid API key is required to fetch weather data.
- **GeoDB Cities API:** Used for geolocation and city lookup. GeoDB (via RapidAPI) supplies global city and region data, useful for features like city autocomplete search. (It requires a RapidAPI key and specific headers for host and key on each request [3]).
- **NewsAPI:** Provides news headlines and articles from various sources. The app uses NewsAPI to fetch the latest news; an API key is needed to authenticate requests (free for development use) [4].
- **TMDB (The Movie Database):** Supplies movie data such as trending movies, ratings, and poster images. Requires a TMDB API key for requests.

- **Twelve Data API:** Offers financial market data. The project uses Twelve Data for stock quotes and uses their WebSocket endpoint for real-time price updates. An API key is required to connect to Twelve Data's services.

- **Three.js (WebGL) & React Three Fiber:** Three.js is a popular JavaScript library for rendering 3D graphics in the browser using WebGL [5]. In this project, Three.js is employed (via a React integration, likely **React Three Fiber** or similar) to create the **WeatherGlobe** component – a 3D globe visualization of weather data. This adds an interactive, visually rich element to the dashboard.

- **Framer Motion:** An animation library for React used to enhance the user interface with smooth transitions and interactive effects. Framer Motion makes it easier to construct complex animations via declarative components and hooks [6]. In this app, Framer Motion powers animations such as card fade-ins, layout transitions, and hover effects, contributing to a polished UX.

- **Tailwind CSS:** A utility-first CSS framework used for styling the application. Tailwind provides a set of CSS classes for rapidly building custom designs. It ensures a consistent design system and makes it easy to implement responsive layouts and dark mode support. The project's UI design and responsiveness heavily rely on Tailwind's utility classes.

- **Testing Libraries:** For reliability, the project includes:

- **Jest** (with React Testing Library) for unit tests of components and utilities.

- **Cypress** for end-to-end (E2E) testing, to simulate user interactions with the dashboard in a headless browser environment and ensure all features work together correctly.

- **Development & Build Tools:**

- Module bundling and build is handled by Next.js (which uses webpack under the hood).
- **ESLint** and **Prettier** are set up for code linting and formatting to maintain code quality and consistency.
- **Husky** and **lint-staged** are used for Git hooks, ensuring that pre-commit checks (like running linters/tests) are performed.
- **Vercel/Netlify:** The app can be deployed on modern hosting services like Vercel (Next.js's default deployment platform) or Netlify. These services are used for continuous deployment, handling the building of the app and serving it globally.

All these technologies work in unison: Next.js serves as the foundation tying frontend and backend, Redux/RTK Query manages client state and API data, and visual libraries (Three.js, Framer Motion, Tailwind) provide a rich, responsive user experience. The result is a modern, maintainable codebase.

Folder Structure

The repository is organized in a way to separate concerns and make the project scalable. Below is an overview of the primary directories and files:

```
├── app/ or pages/           # Next.js pages (routes) and API endpoints
│   ├── api/                 # Next.js API routes for backend logic (e.g.,
NextAuth, SSE, etc.)
│   │   └── auth/[...nextauth].ts    # NextAuth routes for authentication
callbacks
│   │   └── news-stream.ts           # Example: an API route for News SSE
streaming
│   │   └── ...other API endpoints (if any)
│   ├── _app.tsx             # Custom App component to configure global
providers (Redux, NextAuth)
│   ├── _document.tsx        # Custom Document for server-side rendering
setup (if used)
│   └── index.tsx            # The main dashboard page (could also be under
app/ as page.tsx in Next 13+)
├── components/              # Reusable React components (UI and feature
components)
│   ├── WeatherGlobe.tsx     # 3D globe component for weather visualization
│   ├── MultiCityWeather.tsx # Component to display multiple city weather
cards
│   ├── WeatherForecast.tsx  # Component for detailed forecast display
│   ├── NewsCard.tsx         # Component for live news updates
│   ├── MovieCard.tsx        # Component for movie info display
│   ├── StockCard.tsx        # Component for stock prices display
│   └── ...other UI components (headers, layout, etc.)
├── store/                   # Redux store configuration and slices (RTK
Query APIs)
```

```
│   ├── index.ts            # Configures Redux store (with Redux Toolkit)
│   ├── weatherApi.ts       # RTK Query "slice" for Weather API endpoints
│   ├── newsApi.ts          # RTK Query slice for News API
│   ├── moviesApi.ts        # RTK Query slice for TMDB API
│   ├── stocksApi.ts        # RTK Query slice for TwelveData API
│   └── userSlice.ts        # (Optional) Traditional Redux slice for user
state or preferences
├── utils/ or lib/          # Utility functions and helpers
│   ├── apiClients.ts       # Helper functions for API calls (if not using
RTK Query for some)
│   ├── constants.ts        # Constant values (e.g., API endpoint URLs,
static data)
│   └── threejs/            # Three.js helper modules (for setting up the
globe, if any)
├── public/                 # Static assets (served directly by Next.js)
│   ├── images/             # Image assets (logos, icons, etc.)
│   ├── models/             # 3D models or textures for Three.js (e.g.,
globe texture map)
│   └── favicon.ico         # Favicon and other static files
├── styles/                 # Global styles and Tailwind CSS config
│   ├── globals.css         # Tailwind base imports and any global CSS
overrides
│   └── tailwind.css (if separate) # Could be merged with globals.css
├── __tests__/              # Unit tests (Jest)
│   ├── components/         # Tests for React components
│   └── utils/              # Tests for utility functions
├── cypress/                # End-to-end tests (Cypress)
│   ├── e2e/                # Cypress test specs
│   ├── support/            # Cypress support files (commands, config)
│   └── tsconfig.json       # Cypress TypeScript configuration
├── .env.example            # Example environment variable definitions (to
be copied to .env.local)
├── next.config.js          # Next.js configuration (e.g., image domains,
env variables)
├── package.json            # Project dependencies and scripts
├── tsconfig.json           # TypeScript configuration
└── README.md               # Basic documentation (this comprehensive doc
may be separate)
```

Directory Descriptions:

- **pages/** or **app/**: Contains the Next.js routing structure. In a Next.js Pages Router (older structure), each file in **pages/** becomes a route. For example, `pages/index.tsx` is the dashboard homepage, and `pages/api/*` contains API route handlers (for backend logic like authentication and streaming). In the newer App Router (Next.js 13+), the `app/` directory would be used instead with a similar structure (e.g., `app/page.tsx` for the main page, and `app/api/` for API routes). This project can work with either architecture, but uses API routes for things like NextAuth (`api/auth/[...nextauth].ts`) and real-time features.

- **components/**: Holds all the reusable UI components and feature-specific components. Major pieces of the UI such as the WeatherGlobe, NewsCard, MovieCard, etc., each reside here. These components are composed together in the pages to build the dashboard. Having them separated makes the code more modular and easier to maintain or test.

- **store/**: Contains the Redux setup. The `store/index.ts` (or `store.ts`) initializes the Redux store using Redux Toolkit's `configureStore`, adding the API slices and any Redux reducers. The various `*Api.ts` files define RTK Query API slices for different domains:

- For example, **weatherApi.ts** might use `createApi` to define endpoints like `getCurrentWeather` and `getForecast` with their respective queries. Each API slice manages data fetching and caching for its resource type.

- If there's a need for non-API global state (e.g., storing a list of favorite cities or theme preference), a traditional slice (like **userSlice.ts** or **settingsSlice.ts**) could be defined here as well.

- **utils/** (or **lib/**): Utility functions and helpers that are not React components. This may include:

- API client configurations (e.g., a pre-configured Axios instance or fetch wrappers with common headers).
- Constant definitions for things like base URLs or default values.

- For Three.js, if complex setup is needed (like loading textures or models), helper modules could be here (e.g., a module to initialize the Three.js scene, load Earth texture, etc., used by the WeatherGlobe component).

- **public/**: Static files that can be directly referenced by the app (they are served from the root `/`). This includes images (e.g., icons, background images, maybe weather condition icons if stored locally), any 3D model files or textures for the globe, and the favicon. For example, if the globe uses a texture map image of Earth, it could be placed in `public/models/earth-texture.jpg` and loaded at runtime.

- **styles/**: Global styling files. Since Tailwind CSS is used, there is typically a `globals.css` that imports Tailwind's base, components, and utilities. Tailwind is configured via `tailwind.config.js` at the root. This folder could also contain other global CSS or SASS files if needed, but in this project Tailwind covers most styling needs. The Tailwind config is set to scan the components and pages directories for class names (to tree-shake unused styles in production).

- **tests/**: Contains unit tests. Tests are organized mirroring the structure of the code (e.g., tests for components in a `components` subfolder). Using Jest and React Testing Library, these tests verify that components render correctly given props or that utility functions return expected results. For instance, there might be a test for a reducer or a test that the `NewsCard` properly opens an EventSource and updates state on new messages (these would be more integration-level tests).

- **cypress/**: Contains end-to-end tests. Under `cypress/e2e`, test specifications simulate user behavior in a browser. For example, a Cypress test might cover: logging in via Google (if test credentials provided), then checking that the dashboard displays weather info, news headlines updating, etc. The Cypress setup includes support files for configuring the test environment (like

clearing database or seeding data if needed, though here mostly third-party APIs are used so perhaps stubbing network calls in tests). Cypress tests help ensure all pieces (frontend UI, backend API routes, third-party integration) work together in a running application.

- **Environment & Config Files**: The root contains configuration files:

- `.env.example` – a template of required environment variables for the project. Developers should copy this to `.env.local` and fill in actual API keys and secrets (more details in the next section).
- `next.config.js` – Next.js configuration. Notably, this may include an `images.domains` array specifying external domains for images (for example, TMDB images domain, weather icons domain) so that Next.js Image Optimization can fetch and optimize those images. It may also include any custom webpack or environment variable definitions needed at build time.
- ESLint and Prettier configs (`.eslintrc.js`, `.prettierrc`) – define linting rules and code style formatting rules used in development.
- `package.json` – lists dependencies like `"next"`, `"react"`, `"@reduxjs/toolkit"`, `"next-auth"`, etc., and contains useful scripts (e.g., `dev`, `build`, `start`, `test`, `lint`, `cypress`).

This structured layout promotes clarity: front-end pages and components are distinct from backend logic (API routes), and third-party integration logic is mostly encapsulated either in RTK Query slices or util functions. It helps new contributors to navigate the project and find relevant code for each feature.

Environment Setup

To get the project up and running on a local development environment, follow these steps:

1. **Clone the Repository:** Begin by cloning the project repository from the source control platform (e.g., GitHub). For example:

```
git clone  https://github.com/gopichand1939/pgagi-analytics-dashboard-gopi
cd weather-analytics-dashboard
```

1. **Install Dependencies:** Ensure you have **Node.js (>=14 or 16)** installed. Then install all required NPM packages by running:

```
npm install
```

This will download all dependencies such as Next.js, React, Redux Toolkit, NextAuth, Tailwind, etc., as specified in `package.json`.

1. **Environment Variables:** The project relies on several API keys and configuration variables. All these are provided via environment variables to keep sensitive information out of the code. In the root folder, there should be an `.env.example` file listing the needed variables. Copy this file to `.env.local` (Next.js automatically loads `.env.local`):

```
cp .env.example .env.local
```

Then open `.env.local` in a text editor and fill in the values for each variable:

- **Weather API Key:** e.g., `WEATHER_API_KEY=` **your_weather_api_key** (from WeatherAPI.com or OpenWeatherMap, depending on which is used).
- **GeoDB API Key:** e.g., `GEO_API_KEY=` **your_geodb_api_key** (the RapidAPI key for GeoDB Cities).
- **News API Key:** e.g., `NEWS_API_KEY=` **your_newsapi_key** (from NewsAPI.org – free signup to get a key).
- **TMDB API Key:** e.g., `TMDB_API_KEY=` **your_tmdb_key** (from your TMDB account settings).
- **Twelve Data API Key:** e.g., `TWELVEDATA_API_KEY=` **your_twelvedata_key** (from Twelve Data).
- **NextAuth/Google OAuth:**
  - `GOOGLE_CLIENT_ID=` **your_google_oauth_client_id** (from Google Cloud Console credentials)
  - `GOOGLE_CLIENT_SECRET=` **your_google_oauth_client_secret**.
  - `NEXTAUTH_SECRET=` **some_random_secret** – a secret key for NextAuth to encrypt session tokens (you can generate a random 32-character string).
  - `NEXTAUTH_URL=http://localhost:3000` – the base URL of your app for NextAuth (in development it's usually localhost; in production it should be the live URL).

Make sure to **never commit** your `.env.local` (it should be in .gitignore) since it contains your secrets.

Example `.env.local`:

```
# API Keys
WEATHER_API_KEY=YOUR_WEATHER_API_KEY
GEO_API_KEY=YOUR_RAPIDAPI_KEY_FOR_GEODB
NEWS_API_KEY=YOUR_NEWSAPI_KEY
TMDB_API_KEY=YOUR_TMDB_KEY
TWELVEDATA_API_KEY=YOUR_TWELVEDATA_KEY

# NextAuth Google OAuth
GOOGLE_CLIENT_ID=your-google-client-id.apps.googleusercontent.com
GOOGLE_CLIENT_SECRET=your-google-client-secret
NEXTAUTH_SECRET=your-nextauth-secret
NEXTAUTH_URL=http://localhost:3000
```

Each of these variables will be consumed in the code where appropriate, for example: - The Weather API key is used when making requests to the weather service. - The Google OAuth credentials are used by NextAuth to set up the Google login provider. - The NextAuth secret and URL are used by NextAuth for secure session handling and callback URLs respectively.

1. **Configure API Keys in Code (if needed):** The application is designed to read from the environment variables, so if `.env.local` is set up, no code changes should be necessary. In some cases, certain keys might need to be exposed to the browser. For instance, if the weather or movie API calls are made directly from the client-side, their keys might be referenced as `process.env.NEXT_PUBLIC_*`. If the template uses `NEXT_PUBLIC_` prefix for some keys, ensure to include that prefix in the .env file names so Next.js knows to expose them to client-side. Otherwise, calls can be proxied through API routes to keep keys secret.

2. **Run the Development Server:** Start the Next.js dev server by running:

```
npm run dev
```

This will launch the app at **http://localhost:3000**. You should see the dashboard login page or main page load. In dev mode, Next.js provides helpful debugging and hot-reloading of changes.

1. **External Service Setup:** Some features might require setup on external services:
2. **Google OAuth App:** Ensure you have created an OAuth 2.0 Client ID in Google Cloud Console for the project. In the Google API Console, add `http://localhost:3000/api/auth/callback/google` as an authorized redirect URI for the OAuth credentials [7]. This allows Google to redirect back to your app after login. Use the obtained client ID and secret in your .env.

3. **API Plan Limits:** Note that free tiers of the APIs (NewsAPI, Weather API, TwelveData, etc.) have rate limits. For development and testing, be mindful of those limits. If necessary, you can configure the app to use stub data or limit frequent calls (for example, News SSE might fetch every few minutes rather than every few seconds to stay within quotas).

4. **Testing Setup (optional):** If you plan to run tests:

5. For unit tests with Jest, run `npm run test`. The first run will also probably generate a coverage report.
6. For Cypress E2E tests, ensure the dev server is running (or use `npm run build && npm run start` for a production server locally), then run `npx cypress open` to launch the test runner.

By following these steps, you should have a local instance of the Weather and Analytics Dashboard running and be able to develop or evaluate it. The environment configuration is crucial since without valid API keys the app's data-fetching will not work (it will likely show errors or placeholders instead of real data).

API Integrations

One of the highlights of this project is integration with multiple third-party APIs. This section describes how each API is used, what for, and how to configure them in the project:

Weather API (Current Weather & Forecast)

The dashboard uses a Weather API to retrieve current weather conditions and forecast data for cities. We have used a provider (such as **WeatherAPI.com** or OpenWeatherMap) to get data like temperature, humidity, conditions (sunny, rainy, etc.), and multi-day forecasts.

- **Usage in Project:** The Weather API is called to get:
- *Current Weather:* For one or multiple cities, the current temperature, weather description, and an icon representing the condition.
- *Forecast:* Extended forecast (e.g., next 5 days or hourly forecast) for a selected city.
- **API Key:** A personal API key is required. In the environment variables, set `WEATHER_API_KEY`. The application includes this key in requests (often as a query parameter like `?key=YOUR_KEY` or `?appid=YOUR_KEY` depending on the API).

- **Configuration:** The base URL and endpoints are typically defined either in the RTK Query slice or in a config file. For example, using WeatherAPI.com, the base URL might be `http://api.weatherapi.com/v1/`. Endpoints include `current.json?q={CITY}` for current weather and `forecast.json?q={CITY}&days=5` for forecasts, with the `key` param set to your API key.
- **Where Configured:** In code, these details appear in the `weatherApi` RTK Query setup (e.g., `weatherApi.ts`) or a fetch utility. The API key is pulled from `process.env.WEATHER_API_KEY`. (If calls are made client-side, it might be `NEXT_PUBLIC_WEATHER_API_KEY`.)

Note: The app likely also uses **GeoDB Cities API** in conjunction with the Weather API. GeoDB is used for city lookup and geolocation: - When a user searches for a city name (for adding to the dashboard), the app calls GeoDB to get city suggestions (with country, population, etc.). - **GeoDB API Key:** The RapidAPI key for GeoDB should be set in `GEO_API_KEY`. Requests to GeoDB require two headers: `X-RapidAPI-Key` (your key) and `X-RapidAPI-Host` (the host for the GeoDB service) [3]. In this project, those headers are likely defined in a utility or directly in the fetch call. For example:

```
const geoOptions = {
  method: 'GET',
  headers: {
    'x-rapidapi-host': 'wft-geo-db.p.rapidapi.com',        // host for REST
GeoDB
    'x-rapidapi-key': process.env.GEO_API_KEY,            // your RapidAPI
key
  }
};
```

This allows the app to call endpoints like `https://wft-geo-db.p.rapidapi.com/v1/geo/cities?namePrefix=London` to search for cities starting with "London".

- The GeoDB integration means users get an autocomplete dropdown of city names and can select a city to add to their weather watchlist. Once a city is selected (with its location data from GeoDB), the Weather API is used (via latitude/longitude or city name) to fetch the actual weather.

NewsAPI (Live News Updates)

For the news section, the project uses **NewsAPI** (newsapi.org) to fetch the latest news headlines. NewsAPI aggregates news articles from many sources and provides a simple JSON API.

- **Usage in Project:** The **NewsCard** component displays recent news headlines (and possibly brief descriptions or source info). Rather than showing static news, the project implements live updating news:
- This is achieved via **Server-Sent Events (SSE)**: the app maintains an open connection to the server, which pushes news updates in real-time. The Next.js server periodically fetches news from NewsAPI and streams new headlines to the client.
- **API Key:** NewsAPI requires an API key (obtained from signing up on newsapi.org). Set this in environment as `NEWS_API_KEY`. The key is included in requests as a query param (e.g., `GET https://newsapi.org/v2/top-headlines?country=us&apiKey=YOUR_KEY`) [4].

- **Configuration:** The project likely has an API route (e.g., `/api/news-stream`) that handles SSE. On client side, the NewsCard might use the EventSource Web API to connect to this endpoint. The server route would:
- Set headers `Content-Type: text/event-stream` and `Cache-Control: no-cache` to establish an SSE stream.
- Fetch news data initially (e.g., top headlines) using the NewsAPI key.
- `res.write()` the data in SSE format (`data: {json}\n\n`).
- Possibly set an interval (say every few minutes) to fetch fresh news and send down any new headline as an SSE message.
- Listen for the client disconnect (`req.on('close', ...)`) to stop the interval.

Using SSE means the client doesn't have to poll; it just listens for incoming news events. Next.js (Node.js) supports such streaming endpoints [8] [9]. The NewsAPI query could be for top headlines of a certain category or country, or even based on user preference.

- **Where Configured:** The SSE logic is in the API route file. The NewsAPI URL and the API key usage would be defined there (for server-side fetch, the key can remain secret and not exposed to the client). The NewsCard component just opens an EventSource to `/api/news-stream` and updates its state when `onmessage` events are received.

Note: Because NewsAPI free tier might limit requests (e.g., 100 requests/day), the interval for polling on the server should be moderate (perhaps every 5-15 minutes) to not exhaust the limit, or it could fetch a batch of headlines and stream them gradually to simulate real-time.

TMDB (Movie Data)

The **MovieCard** section of the dashboard integrates with the **TMDB API** (The Movie Database) to showcase movie information.

- **Usage in Project:** Likely the dashboard shows a currently trending movie or a list of trending/upcoming movies. For example, it might display the poster, title, and rating of the top movie of the day, or allow the user to scroll through a few popular movies.
- **API Key:** TMDB requires an API key (sign up at themoviedb.org to get one). This should be placed in the environment as `TMDB_API_KEY`. In requests to TMDB's API (e.g., `/3/trending/movie/day` or `/3/movie/top_rated`), the key is provided as a query parameter `api_key=YOUR_KEY`.
- **Configuration:** The project likely uses RTK Query or fetch to call TMDB endpoints. For example, an RTK Query endpoint could be defined as:

```
getTrendingMovie: builder.query({
  query: () => `https://api.themoviedb.org/3/trending/movie/day?
api_key=${process.env.TMDB_API_KEY}`
})
```

  or if the key is exposed to client (not recommended, but possible for simplicity), using `NEXT_PUBLIC_TMDB_API_KEY` in a fetch from the client side.
- **Images:** TMDB provides image URLs for posters and backdrops. These are usually on the domain `image.tmdb.org`. The MovieCard likely displays a poster image. Using Next.js' `<Image>` component is ideal for performance. We have to configure Next Image to allow TMDB's image domain:

- In `next.config.js`, ensure `images.domains` includes `image.tmdb.org` (and any other external image sources like weather icons, news thumbnails, etc.). This allows Next to proxy and optimize those images.
- **Where Configured:** If using RTK Query, the movie endpoints are defined in a `moviesApi` slice. Otherwise, the MovieCard might call an internal API route or getServerSideProps to fetch movie data on the server. The API key usage is hidden on the server if done via internal route.

Once configured, the MovieCard will show data like *"Movie Title (Year)"*, *Rating: 8.5/10*, etc., with a nice poster image. If multiple movies are shown, Framer Motion might be used to animate transitions (like a carousel or flip through movies).

Twelve Data (Stock Market Data)

For the stocks section, the project utilizes the **Twelve Data API** to get real-time financial data. Twelve Data supports stock quotes, forex, crypto, etc., and offers a WebSocket for live updates.

- **Usage in Project:** The **StockCard** component shows one or more stock ticker prices updating live. For example, it might display stocks like AAPL, GOOG, BTC/USD price, etc., updating as trades happen. This is done via a **WebSocket** connection to Twelve Data's streaming API.
- **API Key:** A Twelve Data API key is required (available from twelvedata.com after registering). Set it in the env as `TWELVEDATA_API_KEY`.
- **WebSocket Integration:** Instead of repeatedly polling a REST endpoint for prices, the project opens a persistent WebSocket connection:
- The WS URL is `wss://ws.twelvedata.com/v1/quotes/price?apikey=YOUR_KEY`. This connects to Twelve Data's price quote stream [10].
- After connecting, the client sends a subscription message specifying which symbols to subscribe to. For example, a JSON payload:

```
{
  "action": "subscribe",
  "params": {
    "symbols": "AAPL,GOOGL,MSFT,ETH/BTC"
  }
}
```

  This tells the server to start sending updates for those symbols.
- The WebSocket will then push messages whenever any of those asset prices update (typically messages containing the symbol, price, timestamp).
- **Configuration in Project:** The StockCard component likely handles this logic:
- On component mount, create a new WebSocket using the provided URL (including the API key from env).
- On open, send the subscribe payload (the list of symbols might be hard-coded for demonstration or configured somewhere in the app, possibly user-selectable).
- Listen for incoming messages: `socket.onmessage = (event) => { ... }` – parse the JSON data which contains updated prices.
- Update component state or Redux state with the new prices so that the UI re-renders showing the latest values.

- On component unmount, close the WebSocket to avoid memory leaks or stale connections.

- **Where Configured:** The API key is embedded in the WebSocket connection URL. If the connection is initiated on the client-side, they might use `process.env.NEXT_PUBLIC_TWELVEDATA_API_KEY` (exposing the key). Ideally, to keep the key secret, one could have a server proxy for the WebSocket, but implementing a proxy for WS is non-trivial. Given this is a demo/portfolio project, it's possible the key is exposed and used directly on the client for simplicity. (Twelve Data keys on free plan may not be highly sensitive as long as usage is within free limits.)

- **Live Updates:** As prices come in, the StockCard can display the values and perhaps use some color coding or small chart:

- For instance, if a stock's price went up, show the price in green with an up arrow; if down, in red with a down arrow.
- If using a chart library or just Sparkline, it could plot recent prices – but since none was specified, probably it just shows numeric values and maybe percentage change.

Using WebSockets here demonstrates real-time bidirectional communication. The difference from SSE is that WebSocket could allow sending data from client to server too (though in this case the client mostly receives). Next.js can handle WebSockets via API routes if needed, but here it's probably the client connecting directly to Twelve Data's server. This offloads the real-time data work to Twelve Data and keeps the Next.js server free (except for other API routes).

Other Integrations and Notes

- **NextAuth (Google OAuth):** While not a data API, it's a crucial integration. NextAuth is configured to use Google as an OAuth provider. In the NextAuth configuration (in `pages/api/auth/[...nextauth].ts`), the Google provider is set up with the client ID and secret from the env variables [11]. For example:

```
// pages/api/auth/[...nextauth].ts
import NextAuth from "next-auth";
import GoogleProvider from "next-auth/providers/google";

export default NextAuth({
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_CLIENT_ID!,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET!
    })
  ],
  ...
});
```

This allows users to click "Sign in with Google", get redirected to Google's OAuth consent screen, and then back to your app's callback URL. NextAuth handles obtaining the user profile (name, email, avatar) from Google and establishing a session.

- **Map/Geolocation (Optional):** If the app had a feature to detect user's location for local weather, it might use the **Geolocation API** in the browser. This would be optional and not require a key.

The project description doesn't explicitly mention it, but it could be an enhancement (e.g., offering to show weather for the user's current location using navigator.geolocation).

All API keys are configured in one place (env vars), making it easy to replace providers or update keys. The code is written to gracefully handle API responses and errors – for example, if an API limit is reached or returns an error, the app likely shows an error message or falls back to cached data.

Security: Note that wherever possible, API calls are made server-side (to avoid exposing keys). For Weather, News, and Movies, the calls can be made via Next.js API routes or server functions (hiding the key). For the stock WebSocket and possibly weather (if using client-side for dynamic updates), keys may be exposed as `NEXT_PUBLIC_` – which is acceptable for public data APIs but developers should be aware that these keys are visible in the browser. Since these are mostly free-tier API keys, the risk is limited (worst case someone else could use your key, hitting limits).

Authentication (NextAuth with Google OAuth)

Authentication in the dashboard is handled by **NextAuth.js**, providing a secure and convenient way for users to log in. We use Google as an OAuth provider, so users can authenticate with their Google accounts.

NextAuth Configuration

NextAuth is set up in the file `pages/api/auth/[...nextauth].ts` (for App Router, it would be `app/api/auth/[...nextauth]/route.ts`). This catch-all API route is a built-in mechanism for NextAuth to handle all auth-related endpoints (signIn, callback, signOut, etc.). Key aspects of the configuration:

- **Providers:** In our case, we include the Google Provider:

```
providers: [
  GoogleProvider({
    clientId: process.env.GOOGLE_CLIENT_ID!,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET!
  })
  // ... (could add more providers like GitHub, Credentials, etc.)
],
```

NextAuth uses the Google OAuth 2.0 flow. The environment variables `GOOGLE_CLIENT_ID` and `GOOGLE_CLIENT_SECRET` must be set with the credentials you obtained from the Google Cloud Console for this app. These are as configured in the **Environment Setup** step.

– You should have added `http://localhost:3000/api/auth/callback/google` as an authorized redirect URI in Google's OAuth consent screen setup [7] (and similarly, for production, `https://yourdomain.com/api/auth/callback/google`). This ensures Google will redirect back to NextAuth's callback endpoint.

- **Callbacks & Session:** The default NextAuth behavior is to create a session cookie for the logged-in user. The user profile (name, email, image) is available via NextAuth's `useSession()` React hook or by calling `getSession` / `getServerSession` in server-side code. We didn't

implement custom callbacks to manipulate JWT or session in this project (none needed for basic usage). However, one important configuration is setting a `NEXTAUTH_SECRET` in env for encrypting the session token (especially in production for security).

- **Database:** This project likely uses NextAuth in "JWT mode" (the default if no database is specified). That means no user info is stored in a database – the JWT token contains necessary data. For our use case (Google login only), this is sufficient. If we needed to persist additional user data or have email/password login, we might configure a database. (Since not mentioned, we assume no database usage).

- **Protected Pages:** The dashboard page can be protected by NextAuth's `useSession` or `getServerSideProps`. For example, one could use NextAuth's `withAuth` or check session in `getServerSideProps` to redirect unauthenticated users to the login page. If the project uses the App Router, it might leverage the NextAuth middleware or simply conditionally render content based on `session` availability. The goal is to ensure only logged-in users see the main dashboard content (weather, news, etc.), while others see a login screen or a "Sign In" button.

- **Sign In/Out UI:** NextAuth provides a default signin page, but we can also create a custom page. Possibly this app has a simple landing page with a **"Sign in with Google"** button. When clicked, it triggers NextAuth's `signIn('google')` function, redirecting to Google. After login, NextAuth redirects back to the dashboard page.

- **Google Profile Data:** On successful login, NextAuth gives us the user's name, email, and avatar image from Google. We might display the user's name or avatar in the UI (e.g., in a navbar or a profile dropdown). This is a nice touch to personalize the dashboard. Also, if we had user-specific settings (like stored favorite cities or stocks), we could key them by the user's email or ID.

To summarize, NextAuth abstracts away a lot of the complexity of OAuth. The Google provider was straightforward to set up, requiring just the client ID/secret and adding the correct redirect URIs [7] [11]. NextAuth handles creating secure cookies, protecting API routes, and integrating with Next.js seamlessly. In development, no additional setup beyond environment vars is needed; in production, ensure `NEXTAUTH_URL` is set to your app's URL (e.g., on Vercel this is your domain), and that the Google OAuth app's redirect URI matches that domain.

Core Features

Now let's dive deeper into the core features of the dashboard and how they are implemented:

Dashboard Layout

The **Dashboard Layout** is the overall structure that arranges the various cards and components (Weather, News, Stocks, Movies) on the page. It is designed to be clean, intuitive, and responsive.

- **Structure:** The main dashboard (likely `pages/index.tsx` or `app/page.tsx`) uses a combination of HTML elements and Tailwind CSS classes to create a grid or flexbox layout. For instance, on a large screen, you might see a two-column or three-column layout: e.g., the WeatherGlobe taking a large portion on left and other cards stacked on the right. On mobile, these would collapse into a single column stack for easy vertical scrolling.

- **Header & Navigation:** The project might include a top navigation bar or header, possibly showing the app title, and a profile menu (with the user's name or avatar from Google and a logout button). This header could also contain a theme toggle (light/dark switch) if implemented. Next.js allows defining a custom layout, so a component like `components/Layout.tsx` could wrap the dashboard page providing consistent header/footer.

- **Content Cards:** Each feature (weather, news, etc.) is encapsulated in a "card" style component:

- A **card** here means a container with a slight background and rounded corners (Tailwind classes like `bg-white dark:bg-gray-800 shadow-md rounded-xl p-4` for example). The card typically has a heading (e.g., "Weather", "Latest News", etc.) and the content.

- Cards help visually separate concerns and make the UI modular. Framer Motion might animate these cards on mount (e.g., fade up) to add a subtle polish.

- **Responsive Behavior:** Using Tailwind's responsive utility classes (like `md:grid-cols-2` or `lg:flex`), the layout switches at certain breakpoints. For example:

- On **desktop** (lg breakpoint), WeatherGlobe might be rendered at a large size on the left, and on the right side, a column of stacked cards for multi-city weather, news, movies, stocks.
- On **tablet** (md breakpoint), maybe two columns with WeatherGlobe and one other card side by side, the rest stacking.
- On **mobile** (sm and below), everything stacks vertically: first maybe a smaller globe or just the main city weather, then other cards one after another.

The design ensures no horizontal scroll on mobile and that content is still readable (fonts sizing and card reordering might be considered; e.g., perhaps on very small screens the 3D globe might be hidden or replaced with a simpler summary due to performance or space).

- **State and Routing:** The main dashboard likely does not require multiple routes (all content is on one page after login). However, if the project were extended, one could have sub-pages (e.g., a full-screen weather detail page, or settings page). In such a case, a sidebar or nav menu might highlight the current section. The prompt doesn't indicate multiple pages for these features, so it's safe to assume a single-page dashboard with all components.

- **Example Layout (desktop):**

```
-------------------------------------------------------
|  Header (Logo/Title, Theme Toggle, User Avatar)     |
-------------------------------------------------------
|  [ Weather Globe  ]   |   [ MultiCityWeather Card ] |
|                       |   [ Weather Forecast Card ] |
|-----------------------   --------------------------|
|  [ NewsCard (live) ]   |   [ MovieCard ] [ StockCard ] (maybe side by
side)
|-----------------------   --------------------------|
|  Footer (if any)                                    |
-------------------------------------------------------
```

This is speculative, but gives an idea of how components might be arranged.

- **Styling:** Tailwind provides quick styles: e.g., using `grid grid-cols-1 md:grid-cols-2 gap-4` for a responsive grid, or `flex flex-col` vs `flex-row` based on breakpoints. The theme (light/dark) styling is automatic if classes like `bg-white` and `bg-gray-800` are paired with the `dark:` variant.

Overall, the dashboard layout component orchestrates all other pieces, ensuring they are placed correctly and interact well (for example, maybe making the globe component and a map of city cards scrollable if one column is longer, etc.). It also likely includes the **Redux Provider** and **NextAuth Session Provider** at a high level (commonly in `_app.tsx`), so that all child components can access global state and auth status.

WeatherGlobe (3D Weather Visualization)

One of the most unique components of this project is the **WeatherGlobe** – a 3D globe that displays weather-related information in an interactive way. This is implemented using Three.js (and possibly React Three Fiber for integration with React).

- **Rendering the Globe:** Three.js allows us to create a sphere and map a texture onto it to represent Earth. The WeatherGlobe component likely:
- Creates a Three.js scene, camera, and renderer.
- Adds a sphere geometry (with radius and detail level appropriate for the canvas size).
- Applies an Earth texture (an image of the world map) onto the sphere material, so the sphere looks like Earth. The texture image might be stored in `public/` (e.g., an Earth map or even a real-time cloud map).
- Adds lighting to give the globe a three-dimensional appearance (e.g., an ambient light and a directional light simulating sunlight).

- Optionally, uses Three.js controls or custom code to allow the user to rotate or zoom the globe (for instance, using the orbit-controls utility).

- **Weather Data Overlay:** How does weather tie into the globe?

- One possibility is that for each city in the MultiCityWeather list, the globe plots a marker at that city's latitude/longitude. This could be done by converting lat/long to 3D coordinates on the sphere surface and placing a small mesh (like a pin or dot) there.
- The marker could perhaps be color-coded or sized according to temperature, or maybe when clicked it shows that city's weather.
- If not per-city, another approach is using a global weather layer. Some projects use public APIs for global cloud coverage (e.g., NOAA satellite images) and map it on the globe [12]. That might be beyond scope, but a simpler overlay might be to show day/night (terminator line) based on time, etc.

- Given the complexity, likely the globe at least marks the cities of interest. It could also just be decorative – showing Earth spinning slowly with no markers, while weather details are in text elsewhere. However, the name "WeatherGlobe" implies an interactive element, so markers or clickable points are plausible.

- **Interactivity:** If the globe is interactive, the user could:

- Rotate it: click and drag to spin the globe, using inertia for a smooth feel. React Three Fiber plus Drei controls (like `<OrbitControls />`) can enable this easily.

- Hover or click markers: If city markers exist, hovering could show the city name and temp, clicking could trigger the dashboard to focus that city's weather details (perhaps updating the WeatherForecast component to that city).

- The component might manage state for currently highlighted city, etc., and tie into Redux or parent component callbacks to communicate which city is active.

- **Performance Considerations:** 3D rendering can be heavy, especially with larger textures or many objects. Steps likely taken:

- The Three.js component is only rendered on the **client side**. Next.js Server-Side Rendering is not applicable for WebGL contexts. The code probably uses a dynamic import or a `useEffect` hook to ensure it runs only in the browser. If using React Three Fiber, they likely mark the component with `"use client"` at the top, as R3F doesn't work in server components.
- If the globe is causing low performance on small devices, developers might opt to hide it on mobile (e.g., using Tailwind's `hidden sm:block` classes, so it only shows on sm and up). Or reduce detail on smaller screens.

- The frame rate might be limited or the globe could pause rotation when not in view.

- **Integration:** WeatherGlobe might not fetch data itself (except maybe to get an updated texture for clouds, which is unlikely here). It likely relies on the MultiCityWeather component or global state to get locations to mark:

- For example, if the user adds "New York" and "London" to their cities list, WeatherGlobe receives that list (via props or by reading from Redux) and then plots those points.

Visually, this feature adds a **"wow factor"** to the dashboard, distinguishing it from a typical flat design. It engages users and also provides a quick spatial context (e.g., seeing where storms might be if a cloud layer was implemented, or just seeing day/night on the globe as time passes).

Implementation-wise, integrating Three.js with React is done carefully: possibly using **React Three Fiber (R3F)**, which allows writing Three.js scenes declaratively in JSX. If R3F is used, the component code might look like:

```
<Canvas>
  <ambientLight intensity={0.5}/>
  <directionalLight position={[5,5,5]}/>
  <Suspense fallback={null}>
    <mesh rotation={[0, 0, 0]}>
      <sphereGeometry args={[radius, segments, segments]} />
      <meshStandardMaterial map={useLoader(TextureLoader, '/images/
earth.jpg')} />
    </mesh>
    {cities.map(city => (
      <Mesh key={city.id} position={convertLatLonToSphere(city.lat,
city.lon, radius)}>
        <sphereGeometry args={[0.1, 8, 8]} />
        <meshBasicMaterial color="red" />
      </Mesh>
    ))}
```

```
  </Suspense>
</Canvas>
```

(The above is a conceptual snippet just to illustrate). If not using R3F, they would directly use Three.js imperatively in a useEffect.

MultiCityWeather and WeatherForecast

These two features deal with presenting weather data in textual/UI form, complementing the WeatherGlobe's visual approach.

MultiCityWeather: This likely refers to a component or section that displays current weather for multiple cities simultaneously.

- **Functionality:** Users can track multiple cities. The UI could be a set of small weather cards, each showing a city name, current temperature, and an icon (sun/cloud/etc.).
- There might be an "Add City" feature – perhaps an input box with autocomplete (powered by GeoDB Cities API). When a user selects a city, it gets added to the list. The app then fetches that city's weather and displays it. The list could be stored in a Redux slice or local state.
- **Data Fetching:** For each city in the list, a query is made to the Weather API for current conditions. RTK Query shines here: it can be set up with a `getWeatherByCity` query that takes the city name or coordinates as an argument. When the MultiCityWeather component mounts or when the city list changes, these queries are triggered. Thanks to caching, if the user toggles between cities or reopens the app soon, the data might be served from cache (until an invalidation or refetch interval).
- **UI Details:** The component probably maps over an array of city weather data, rendering a mini-card for each. Each mini-card could show:
- City name (perhaps including country code if not obvious).
- Current temperature (in Celsius or Fahrenheit).
- Weather condition description (e.g., "Clear sky", "Rain showers").
- An icon representing the condition (the Weather API often provides an icon code; the project could use those icons via URL or an icon library).
- Maybe additional info like high/low temp of the day or humidity.
- Tailwind can style these nicely (e.g., one card might be `bg-blue-50 dark:bg-blue-900` depending on theme, with a subtle weather-themed color).

- If a city is clicked, it could trigger showing the detailed forecast (see WeatherForecast below) or highlighting that city on the globe.

- **Example:** The dashboard initially might have a default city (possibly the user's current location if known or a preset like "New York"). The user can add more via a plus button. The MultiCityWeather section then shows, say, "New York: 25°C Sunny ☀", "London: 18°C Cloudy ☁", etc., each in a small card.

WeatherForecast: This component provides detailed forecast information for a selected city.

- **Trigger:** The forecast display could be triggered when the user selects or focuses on a particular city (e.g., clicking a city card from MultiCityWeather or selecting from a dropdown). The app would then fetch that city's forecast data via the Weather API.

- **Data:** Forecast data typically includes daily forecasts for the next several days (with min/max temps, condition) and/or hourly forecasts for the next 24 hours. The project likely opts for daily forecast for simplicity.
- **UI:** The forecast might be displayed as:
- A separate card or section titled "Forecast for {City}".
- A list or grid of the next 5 days. Each day's weather could be shown with day of week or date, a small icon, high/low temperature, and maybe a short description (e.g., "Rain" or "Mostly Cloudy").
- Alternatively, a scrollable hourly chart could be included, but unless a chart library is used, it might be a simple list of times and temps.
- **Integration:** The forecast component likely uses RTK Query as well, with an endpoint like `getForecastByCity(cityName or id)`. When a city is chosen, it triggers this query. The data is then rendered. If the user switches the selected city, RTK Query either fetches new data or uses cache if available. If the forecast endpoint is configured to cache by city, going back to a recently viewed city's forecast might not refetch immediately unless expired.
- **Example:** Suppose the user clicks on "London" in MultiCityWeather. The WeatherForecast component then fills with something like:

```
London 5-Day Forecast:
Tue – Mostly Sunny, High 20°C, Low 12°C
Wed – Rain, High 18°C, Low 10°C
Thu – Cloudy, High 16°C, Low 9°C
... etc.
```

This gives the user deeper insight into the upcoming weather, beyond the current conditions.

Both MultiCityWeather and WeatherForecast make the weather functionality more comprehensive: - MultiCityWeather = breadth (several locations at a glance). - WeatherForecast = depth (more info for one location).

They likely share some code or at least the same API slice. The Weather API might allow batch fetching (for instance, OpenWeatherMap has a "group" endpoint to fetch multiple cities in one call), but the project likely calls individually per city for simplicity and caches them.

Error handling: If a city's data fails to load (network issue or invalid city), the UI might show an error message on that card ("Error loading data"). Similarly, forecast might show "Unable to fetch forecast". These states should be handled gracefully, possibly with a retry option.

NewsCard (Live News via SSE)

The **NewsCard** is the portion of the dashboard that shows news headlines with real-time updates. This is implemented using **Server-Sent Events (SSE)** to achieve live updates.

- **Display:** The NewsCard is a UI card that could contain:
- A title like "Latest News" or "Trending News".
- A list of news headlines (with maybe 3-5 items visible at a time).
- Each item might show a headline title, possibly the source name or time.
- Optionally an image thumbnail if available (some NewsAPI articles have a `urlToImage`). If included, Next.js Image can optimize it (make sure to include those domains in config if so).

- Possibly a link to read more (the NewsAPI data includes the article URL which could be opened in a new tab if the user clicks).

- **Real-Time Updates:** The unique part is that new headlines can appear automatically without a page refresh:

- The Next.js API route (e.g., `/api/news-stream`) acts as an SSE server. When the NewsCard mounts, it creates an EventSource to connect to that route.
- The server route on initial connection might send the latest 5 headlines immediately (so the card is populated).
- Then, it keeps the connection open. On the server side, perhaps every X minutes, it fetches fresh headlines from NewsAPI. It can compare with the last sent headlines to avoid duplicates, and if there's a new headline, it `res.write()` it as a new SSE event.
- The EventSource on the client receives these as `message` events. The NewsCard component's JavaScript will have something like:

```
useEffect(() => {
  const es = new EventSource("/api/news-stream");
  es.onmessage = (event) => {
    const newArticle = JSON.parse(event.data);
    setArticles(prev => [newArticle, ...prev].slice(0,5)); // keep only
latest 5
  };
  return () => es.close();
}, []);
```

This way, whenever a new article is streamed, it updates the state to prepend that article to the list (and possibly remove the oldest).

- If SSE disconnects (e.g., server restarts or network issue), the EventSource will typically try to reconnect automatically. The server can send a `retry:` directive, but by default most browsers retry SSE after a few seconds.

- **Server Implementation:** On the Next.js side, since API routes run on Node.js, implementing SSE is feasible (ensuring the route does **not** finish the response and flushes data).

- Use `res.setHeader('Content-Type', 'text/event-stream')` and `res.flushHeaders()` to establish flush behavior.
- Possibly use `res.write(` data: ${json}\n\n `)` to send events.
- Keep an interval with `setInterval` or use a loop with delays to periodically fetch news.
- If using Vercel serverless functions, one must be careful as they might timeout after 10 seconds by default. There are workarounds or Edge Functions for streaming. Since this is a dev project, it might be running on a Node server that can handle SSE (or on Vercel with a special config).

- The SSE route might be programmed to send an event every, say, 60 seconds with the latest headline.

- **Data Source:** We use NewsAPI's `/v2/top-headlines` for a given country or query. For example, top headlines for the US. Each fetch returns an array of articles. The SSE logic could simply take the first article from the latest fetch if it's different from the last first article. Alternatively, it could cycle through articles in the response to send one at a time.

- In a simpler implementation, maybe it just sends the entire list every time (the client then replaces all). But that's less "streaming" and more polling. A more streaming vibe is sending one article at a time as it appears.

- **User Experience:** The effect is that if a major news story breaks, within the next update interval, it appears at the top of the News card without user action. Perhaps the new headline could be highlighted briefly (e.g., using Framer Motion to fade it in or flash a background) to draw attention.

- **Pause/Scroll:** If the news list is scrollable and user scrolls, a decision is needed whether to auto-prepend new items (which could scroll the view). Possibly, the list is small enough (5 items) and auto-updates that it's fine.

In summary, NewsCard demonstrates server-to-client push. SSE is appropriate since the client only needs to receive data [13] (news updates) and doesn't send anything back. SSE has low overhead and works well for this use-case. We just have to manage the open connection. On the client, cleanup with `es.close()` on component unmount is important to avoid ghost connections.

MovieCard (TMDB Integration)

The **MovieCard** feature showcases movie data retrieved from TMDB.

- **Display:** It likely presents either a single movie or a small selection. For instance, it could show "Movie of the Day" or the current top trending movie. Alternatively, it might show a couple of movies side by side if space allows (like top 3 trending movies).
- **Content:** For each movie shown, typical details include:
- Poster image: a visually appealing element (loaded from TMDB's image CDN).
- Title of the movie (maybe truncated if too long).
- Perhaps the release year or date.
- Rating (TMDB's vote_average or a star icon with rating).
- Possibly a short overview or genre tags, if space permits, but since it's a card, it might be minimal.
- **Data Fetch:** The data likely comes from TMDB's trending API: e.g., `/trending/movie/week` which gives a list of movies trending this week. In RTK Query, the `moviesApi` might have `useGetTrendingQuery`. The component then takes the first item (or several) from the result.
- Alternatively, it could be a specific category like "Top Rated" or "Upcoming". The choice is up to implementation, but trending is a good dynamic choice.
- **Interactivity:** If only one movie is shown at a time, perhaps the card has arrows or swipe to move to the next movie (like a mini-carousel). If they used Framer Motion, they might animate transitioning between movies (like sliding the poster out and next one in).
- However, implementing a full carousel might be extra – possibly it's just static showing one item that updates daily.
- **Next.js Image:** The images from TMDB (poster path) should be displayed using the `<Image>` component for automatic optimization. We configure the allowed domain as mentioned (image.tmdb.org). The code might look like: `jsx`

```
<Image
  src={`https://image.tmdb.org/t/p/w500${movie.poster_path}`}
  alt={movie.title}
  width={200} height={300}
/>
```

This would load a medium resolution (w500) poster.

- **Fallbacks:** If for some reason the movie data fails to load (e.g., API limit reached), the UI might show a graceful message like "Unable to load movie data" or simply hide the card. But TMDB's free tier is generous, so likely always loads.

- **Why Include Movies:** It adds an entertainment aspect to the dashboard. A user might come to check weather and news, and also see a movie suggestion or trending info. It rounds out the "analytics" concept by not only focusing on serious data but also media trends.

- **Possible Extension:** If this were expanded, clicking the MovieCard could show more details or trailers, but in this scope it's just a small informational widget.

Overall, the MovieCard is one of the simpler integrations – mainly a straightforward fetch and display, but it benefits from the image optimization and provides an opportunity to use some animations (like hover to flip the card and show more info, etc., which Framer Motion could handle).

StockCard (Real-time Stocks via WebSocket)

The **StockCard** brings financial data into the dashboard, showing live stock or cryptocurrency prices.

- **Display:** The card could list a handful of ticker symbols with their current price and maybe a small change indicator.
- Example layout: "AAPL: 150.23 USD (▲ +1.2%)" with green arrow if up, red if down.
- It might show last updated time or just update continuously.
- If focusing on one stock, it could show a larger number and maybe an intraday mini-chart. But more likely, it's a small list since "StockCard" sounds like one card with multiple stocks.
- **Live Updates via WebSocket:** As described in the API integration, the StockCard opens a WebSocket connection to TwelveData's stream:
- The connection is established when the component mounts. Possibly using the browser's WebSocket API:

```
const socket = new WebSocket(`wss://ws.twelvedata.com/v1/quotes/price?
apikey=${process.env.NEXT_PUBLIC_TWELVEDATA_API_KEY}`);
```

(If this key is not public, they might route through an API, but likely it's used directly on client for simplicity).

- On open, subscribe to a set of symbols (this set might be hardcoded or user-defined). For demonstration, it could include a mix of stocks and indices: e.g., `AAPL, GOOG, TSLA, BTC/USD, ETH/USD`.
- The server will start sending messages for each symbol whenever its price changes.

- Each message from TwelveData might look like:

```
{
  "symbol": "AAPL",
  "price": "150.30",
  "exchange": "NASDAQ",
  "timestamp": "2025-05-15T03:21:00.000Z"
}
```

The component's code will parse this JSON and update the corresponding symbol's price in the state.

- **State Management:** It might use a React useState or useRef to store the latest prices for the subscribed symbols. Alternatively, a Redux slice could hold this data. Given we already have RTK Query, one might think to integrate websockets via RTK Query's middleware, but this is complex; simpler is to handle in the component.

- Possibly the `stocksApi` RTK Query is only used for initial fetch (like get last closing prices via REST). But after that, live updates come via WS.

- For demonstration, they may not even do an initial fetch, just rely on the stream.

- **UI Update Frequency:** Every time a new price comes, the UI updates instantly. If messages are very frequent (some stocks trade multiple times per second), there could be throttling or just let it update very often. Since only a few symbols are subscribed, this is fine.

- Framer Motion or CSS can be used to highlight changes (e.g., flash green/red when a price updates).

- Perhaps use up/down arrows to indicate direction: you compare the new price to the last price (store last price in state, or the message includes change info? Not sure if TwelveData sends previous price, likely not, so compute manually).

- **Cleanup:** On component unmount or page unload, close the WebSocket:

  ```
  socket.close();
  ```

  to avoid keeping the connection alive or getting errors.

- **Error Handling:** If the WebSocket fails (bad API key or network), the StockCard should detect `socket.onerror` or `socket.onclose` unexpectedly, and display an error or retry. Possibly print "Stock data unavailable" in the UI.

- **Security Note:** Exposing the TwelveData API key on the client is a consideration. Ideally, one would proxy it. But since this is an educational project, they may have opted for simplicity. The free API key might be limited to certain requests per minute, but streaming likely counts differently. In any case, for development, it's fine; in production one might regenerate if compromised.

The StockCard demonstrates **bidirectional capability**, though in usage it's mostly server -> client (like SSE). The difference with SSE is that WebSocket can also allow client to send messages (like if we wanted to let the user dynamically subscribe/unsubscribe symbols without reconnecting, we could send an action). In this implementation, once the subscription list is sent at start, we likely don't send further messages from the client.

The combination of NewsCard and StockCard means the dashboard has two real-time streams concurrently: - SSE for news (over HTTP). - WS for stocks. This tests that the app can handle concurrent real-time data flows. Modern browsers and Next.js handle this fine. It's a very dynamic dashboard with constantly updating sections.

State Management with Redux Toolkit

State management in this project is primarily handled by **Redux Toolkit**, with a heavy reliance on **RTK Query** for asynchronous data. The decision to use Redux (as opposed to solely React context or useState) was to effectively manage complex state across the app, especially caching API data and sharing it between components.

Redux Store Configuration

The Redux store is configured likely in `store/index.ts` using `configureStore` from Redux Toolkit. It incorporates: - The RTK Query API slices as middleware and reducers. - Any additional custom reducers (if needed for things like theme or user preferences).

For example:

```
import { configureStore } from '@reduxjs/toolkit';
import { weatherApi } from './weatherApi';
import { newsApi } from './newsApi';
import { moviesApi } from './moviesApi';
import { stocksApi } from './stocksApi';
import userReducer from './userSlice';

export const store = configureStore({
  reducer: {
    [weatherApi.reducerPath]: weatherApi.reducer,
    [newsApi.reducerPath]: newsApi.reducer,
    [moviesApi.reducerPath]: moviesApi.reducer,
    [stocksApi.reducerPath]: stocksApi.reducer,
    user: userReducer,
  },
  middleware: (getDefault) =>
    getDefault().concat(
      weatherApi.middleware,
      newsApi.middleware,
      moviesApi.middleware,
      stocksApi.middleware
    ),
});
```

(This is an illustrative snippet.)

This ensures that any component can use the auto-generated hooks from RTK Query (like `useGetWeatherQuery`) and the data will be stored globally in a normalized cache. Also, the middleware handles dispatching loading states and caching logic.

The store is provided to the app via `<Provider store={store}>` wrapper, typically in `_app.tsx`. If using Next's App Router with server components, they might use a slightly different pattern (like wrapping the client components or using a custom provider).

RTK Query for Data Fetching & Caching

Using **RTK Query** greatly simplifies data fetching: - **Query Endpoints:** For each external API (weather, news, movies, stocks), an RTK Query "slice" (`createApi`) is created with base URL and endpoints. - For example, `weatherApi` might have endpoints: `getWeatherByCity`, `getForecastByCity`. - Each endpoint defines how to fetch data (the URL, possibly headers, etc.) and the expected response shape (for TypeScript). - RTK Query auto-generates hooks like `useGetWeatherByCityQuery(cityName)`.

- **Caching Behavior:** RTK Query automatically caches responses keyed by the query arguments. This means if `useGetWeatherByCityQuery("London")` is used in two different components, they will share the data and only one network call is made. If the user already fetched "London" weather and navigates away and back, it might serve cached data instantly (stale-while-revalidate behavior can be configured, e.g., with `keepUnusedDataFor`).
- Data is normalized by query keys, not globally like a normalized DB, but caching avoids duplicate calls.
- It also handles background refetching if desired (there are options like `refetchOnMountOrArgChange`).
- **Global State vs Local State:** We use RTK Query for server data. For purely UI state (like whether a modal is open, or the currently selected city for forecast), we might use component local state or a small Redux slice.
- For example, the currently selected city for the forecast could be in a `userSlice` (so both MultiCityWeather and WeatherForecast know which city is active). Or simply lifted state in a parent component.
- Theme (dark/light) might be handled outside Redux (like with `next-themes` or just CSS media), but if it were, a slice could manage that.
- **Mutations:** If the app had any user-triggered updates to data (not much here, since mostly reading external APIs), RTK Query's mutation endpoints would be used. Perhaps if we allow user to remove a city from the watchlist and we persisted that somewhere, that could be a mutation (but likely the city list is just local and not persisted to server).
- **Examples in Use:**
- The MultiCityWeather component calls `useGetWeatherByCityQuery(cityName)` for each city (maybe within a `.map()` of city names, or using the new RTK Query `useQueries` pattern if available). Alternatively, it might call a custom hook that wraps multiple queries.
- The News section might not use RTK Query at all because SSE is a different paradigm (no need to cache, it's streaming).
- The MovieCard could use `useGetTrendingQuery` from a `moviesApi`.
- If using `getServerSideProps` (in Pages router) for initial data, they could prefetch and hydrate RTK Query, but since we want real-time and client interactivity, most likely they use client side queries.
- **Caching Duration:** The default may be 60 seconds for RTK Query cache. If the weather data is considered stale after, say, 10 minutes, they might set `refetchOnMountOrArgChange: true` or manually trigger refetch on some interval or when user focuses window.

- E.g., weather might update every hour significantly, so caching for a few minutes is fine.

- **Benefits:** By using RTK Query:

- We avoid redundant fetch logic – no need for manual `useEffect` and `useState` for loading/errors for each API. RTK Query provides `isLoading`, `isError`, and the data.
- The code is cleaner and more declarative.
- We get performance benefits with caching and deduplication out of the box [1].
- If we needed to implement optimistic updates or streaming updates (RTK Query supports WebSocket by custom cache updates), we could, but in this app manual WS outside RTK Query is used for stocks.

Example: Weather Data Flow

To illustrate, suppose the user has added two cities: London and New York. - MultiCityWeather component:

```
const { data: londonWeather, isLoading: loadLondon } =
useGetWeatherByCityQuery("London");
const { data: nyWeather, isLoading: loadNY } = useGetWeatherByCityQuery("New
York");
```

RTK Query will initiate two network calls (if not cached). When data comes, it normalizes like:

```
queries: {
  'getWeatherByCity("London")': { status: 'fulfilled', data: {... London
weather ...} },
  'getWeatherByCity("New York")': { ... }
}
```

The component re-renders with the data. If the user removes and re-adds London quickly, the cached data might be reused instantly.

- If a forecast for London is needed and we have `useGetForecastByCityQuery("London")`, RTK Query treats it separately (different endpoint/key). But you could also design it such that `getWeatherByCity` returned some forecast embedded (depending on API used).

- Redux DevTools would show slices for each API and their cached entries.

Non-RTK State

The project might still have a small amount of non-RTK state in Redux: - A slice to handle *list of cities* in MultiCity (since that's user preference). This might include actions like `addCity` and `removeCity`. The state is an array of city names or IDs. That state is then used to render MultiCityWeather and drive queries. (Alternatively, they could store this list in local storage or just component state that persists via NextAuth (but probably easiest is a Redux slice for it)). - A slice for *UI preferences*: e.g., `theme: 'light' | 'dark'` if not using a dedicated theme library. This would allow a global toggle that affects the whole app. - Because NextAuth manages user session via its own context, we usually do

not put user profile in Redux (NextAuth gives us hooks). But one could copy user info into a Redux slice for convenience. Not necessary.

Summary

Overall, **Redux Toolkit Query** significantly reduces the boilerplate for data management. It ensures the app remains **performant** by caching and only re-fetching when needed, which is especially useful for something like weather (where data doesn't change every second). It also means if multiple components need the same data (e.g., maybe the WeatherGlobe and MultiCityWeather both needed the same city's weather), they automatically share the request.

For developers, RTK Query provides a clear separation of concerns: all data fetching definitions are in one place (the API slices), making it easy to adjust endpoints or add new ones. This is much cleaner than sprinkling `fetch()` calls throughout the components.

Styling and Theme

The project's styling uses **Tailwind CSS**, which enables rapid UI development with utility classes, and supports **dark mode** easily. In addition, design decisions were made to ensure a consistent look and feel across the dashboard, and to incorporate smooth animations where appropriate.

Tailwind CSS Setup

- Tailwind is configured via `tailwind.config.js`. The config likely specifies:
- The **dark mode strategy**: probably `'class'` (meaning a `dark` class on a parent element toggles dark mode) so that we can allow user toggle, instead of only OS preference (`'media'`). By using the class strategy, we can manually control the theme; e.g., adding a `dark` class to `<html>` element to switch to dark mode [14].
- The content paths: including all `pages/**/*.{js,ts,jsx,tsx}`, `components/**/*.{js,tsx}`, etc., so Tailwind knows where to find class names.
- Perhaps a custom color palette or extended theme, if the default colors weren't sufficient. For instance, they might add a specific brand color or adjust the default dark mode color shades.

- Any plugins: maybe forms (if we had input elements), line-clamp (if truncating news text), etc. Possibly not many needed for this project.

- **Global Styles:** In `styles/globals.css`, Tailwind's base styles are imported:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Additional global overrides can be added. The project might have some CSS for things that Tailwind utilities can't easily do, but likely minimal.

- **Using Tailwind in Components:** The JSX in components is filled with class names like:

- Layout: `className="grid grid-cols-1 md:grid-cols-2 gap-4 p-4"`
- Cards: `className="bg-white dark:bg-gray-800 rounded-xl shadow p-4"`

- Text: `className="text-xl font-semibold text-gray-800 dark:text-gray-100"`
- Buttons: `className="py-2 px-4 bg-blue-500 hover:bg-blue-600 text-white rounded"`

This utility approach means no separate CSS files for each component – all styling decisions are right in the markup, which is convenient once you're used to it.

- **Responsive Design:** Tailwind's mobile-first breakpoints ensure the app looks good on all devices:
- They likely used `sm, md, lg, xl` prefixes to adjust layouts as discussed in the Dashboard Layout section.
- Also used relative units, flexbox, grid, etc., to make components resize nicely. E.g., the WeatherGlobe canvas might have a max width and height percentages.

Light & Dark Mode Support

- The dashboard supports both **light mode and dark mode** for user preference or to follow system theme.
- Implementation:
- Tailwind's class strategy for dark mode is used. The `dark:` variant is used in class names to define dark-specific styles. For example:
  - `bg-white dark:bg-gray-800` on card backgrounds means light mode cards are white, dark mode cards are a dark gray.
  - `text-gray-800 dark:text-gray-200` on text for contrasting colors.
  - These dual classes are sprinkled throughout components.
- A toggle UI: likely a button or switch in the header that triggers theme change. If using a library like **next-themes**, it provides a `ThemeProvider` and a `useTheme` hook to toggle between light/dark and remember the choice (stores in `localStorage`). This is a common solution.
  - If next-themes is used, in `_app.tsx` they wrap the app with `ThemeProvider` from 'next-themes'. The toggle button calls `setTheme('dark')` or `setTheme('light')`.
  - If not using next-themes, they could implement manually by toggling a `dark` class on the `html` element. For instance, using a piece of state and an effect:

    ```
    document.documentElement.classList.add('dark');
    ```

    and store preference in localStorage.
- The default might follow system preference. If next-themes is configured with `defaultTheme: 'system'`, it will do that. So a user with dark mode OS gets dark theme initially.

- The documentation likely advises to include a strategy to avoid flash of unstyled or wrong theme (next-themes handles this by adding `class="dark"` initially if needed).

- **Why Dark Mode:** It's a modern expectation for apps to have dark mode, especially for a dashboard that might be open all day (dark mode can reduce eye strain in low light). Implementation is straightforward with Tailwind and adds to user personalization.

Animations and UI Polish (Framer Motion)

- **Framer Motion** is integrated to provide a polished user experience:
- It could be used for **page transition**: e.g., when the dashboard first loads after login, components fade in sequentially.

- **Hover effects**: maybe hovering over a card slightly raises it or highlights it (using Framer Motion's `whileHover` to scale or change shadow).
- **Reordering animations**: If the user adds or removes a city card, Framer Motion's AnimatePresence can animate the card appearing or leaving (smoothly resizing the list).
- **Theme switch animation**: toggling dark mode could cross-fade colors (though typically just instant theme swap is fine).
- **3D Globe**: While Three.js handles the rendering, Framer could be used to animate a UI overlay or animate the globe container size.
- **News updates**: New news headline appearing could slide in from bottom or fade.
- **Stock price changes**: Could animate the number change or flash background color for a moment when price updates.

The specifics aren't described, but given Framer Motion is included, at minimum some components are wrapped in `<motion.div>` with animation props. For example:

```
<motion.div initial={{ opacity: 0, y: 10 }} animate={{ opacity: 1, y: 0 }}
transition={{ duration: 0.5 }}>
  <NewsCard />
</motion.div>
```

This would make the NewsCard fade in and lift up slightly on mount.

- **User Feedback & Accessibility**: Animations are kept subtle to not overwhelm. They serve to guide the user's attention (e.g., if a new stock price comes in, a quick flash indicates something changed). All animations should be optional (honor `prefers-reduced-motion` – Framer Motion can check this or we apply only simple fades if the user prefers no motion).

General UI Decisions

- **Color Scheme:** Likely using Tailwind's default blue or indigo as primary accents (maybe on buttons or highlights). The background in light mode might be a gray-100 and in dark mode gray-900 to give a gentle contrast.
- **Typography:** Tailwind's default font (usually sans-serif like `ui-sans-serif, system-ui`) is used unless changed. Font sizing is responsive (text-lg, text-xl for headings).
- **Spacing & Sizing:** Because Tailwind encourages consistent spacing (using units like `p-4, m-2` which correlate to a design system scale), the app's spacing is uniform. This makes the design look cohesive (e.g., all cards have the same padding, all sections have similar margins between them).

- **Icons:** The project might use an icon library (the example blog used Lucide). Possibly we have icons for weather conditions (sun, cloud, etc.), a moon icon for dark mode toggle, or arrows for stock changes. If using something like Heroicons (integrated with Tailwind UI) or Lucide, those would be included as components or SVGs.

- **Accessibility in Styling:**

- Colors are chosen with contrast in mind (ensuring text is readable on background in both themes; e.g., using Tailwind's curated palette which is generally accessible).

- Focus states are not removed. Tailwind by default doesn't remove outlines, so keyboard navigation should show a focus ring on interactive elements. If custom focus styles are added, they ensure visibility.
- Font sizes should be adequate (no tiny text). And ideally, responsive units so on small devices text isn't too small.

In conclusion, the styling approach makes the app look modern and ensures it's easily theme-able. Tailwind greatly speeds up development by providing ready-to-use styles, while Framer Motion enhances the UX with smooth animations. The combination means we have a visually appealing dashboard that can adapt to user preferences and devices without a lot of custom CSS overhead.

Testing (Unit and E2E)

Robust testing was implemented to ensure the application's quality and reliability. Both **unit testing** and **end-to-end (E2E) testing** methodologies are used.

Unit Testing with Jest

- **Jest Framework:** Jest is the chosen testing framework for unit tests, along with **React Testing Library** for React component tests. Jest provides a fast and reliable way to run tests and comes built-in with create-next-app setups.
- **Configuration:** The project likely has a Jest config either in package.json or jest.config.js. It might use `next/jest` to properly handle Next.js specific features (like image imports, CSS imports).
- **Scope of Unit Tests:** Key parts of the application are tested in isolation:
- **Utility Functions:** Any functions in `utils/` (like a function to convert lat/long to 3D coordinates, or formatting functions for date/temperature) have corresponding test cases to verify correctness.
- **Redux Slices:** If there are pure reducers (e.g., userSlice for adding/removing cities), tests dispatch actions and assert the new state is correct.
- **RTK Query Logic:** It's a bit trickier to unit test RTK Query calls, but we might mock the service calls. Alternatively, we rely on integration tests for those.

- **React Components:** Using React Testing Library, components are rendered in a simulated DOM and assertions are made:

    - e.g., **WeatherGlobe** might be tested to ensure it renders a canvas and markers given a set of cities (though full Three.js context might be hard to test in JSDom, so maybe skip deep testing of 3D).
    - **MultiCityWeather** component: if passed a Redux store state with certain cities' data, it should display those city names and temps correctly.
    - **NewsCard**: could be tested by mocking an EventSource. For example, simulate receiving a message and check that a new headline appears in the DOM.
    - **MovieCard**: if given mock movie data, it should display the title and image.
    - **StockCard**: similarly, maybe test that it formats stock data correctly. WebSocket can be mimicked by calling an update function directly.
    - **Auth**: Testing NextAuth in unit tests is complex; instead, one might simply ensure that a login button calls NextAuth's signIn function (which can be mocked).

- **Running Unit Tests:** Developers can run `npm test` or `npm run test:unit` to execute the Jest test suite. In the output, each test suite and case will be reported, along with code coverage statistics if configured.

- The project may have set up a coverage threshold, or at least instructs running `npm run test -- --coverage` to get a coverage report. This generates a coverage/ directory with an HTML report. We aim for high coverage on critical modules (perhaps 80%+ overall coverage as a goal).

- **Continuous Testing:** During development, Jest can run in watch mode to re-run tests on file changes, which is handy for Test-Driven Development. This might be enabled via `npm run test -- --watch` for developers.

End-to-End Testing with Cypress

- **Cypress Framework:** Cypress is used for E2E tests, which run the full application in a headless browser environment and simulate user interactions. Cypress is chosen for its developer-friendly interface and powerful capabilities to test real scenarios in the browser.

- **Configuration:** Cypress tests reside in the `cypress/e2e/` directory, possibly with spec files like `dashboard.cy.ts`. The project likely has a `cypress.config.js` or similar where the baseUrl is set (e.g., `baseUrl: 'http://localhost:3000'`) so tests can just use relative URLs.

- If using TypeScript, a `tsconfig.json` in cypress/ ensures Cypress types are recognized.

- **Test Scenarios:** E2E tests cover user flows:

- **Authentication Flow:** A test might simulate logging in. Since Google OAuth involves external redirect, we may not do a real Google login in tests. Instead, one might use NextAuth's ability to sign in with credentials, or more simply stub the session. But a full integration test could use Cypress to handle OAuth by using a testing Google account if configured – this is complex, so likely not done. Alternatively, NextAuth could be configured with a development mode (like an `NEXTAUTH_URL=http://localhost:3000` and some test user).
    - Simpler: bypass auth in tests by running the app in development mode where NextAuth might allow an `NEXTAUTH_SECRET` set to "test" and enabling some test login. If too complex, we might skip actual login in Cypress and assume user is logged in (maybe by seeding a session cookie).
- **Dashboard Load:** Visit the dashboard page. Assert that key elements load:
    - WeatherGlobe canvas is present.
    - The MultiCityWeather shows at least one city (maybe a default city "San Francisco" appears and has a temperature displayed).
    - NewsCard shows a list of headlines.
    - MovieCard displays a movie title or image.
    - StockCard displays at least one stock price.
- **Add City Flow:** If applicable, simulate typing in the "Add City" search box, select a city from suggestions. Then assert that a new city weather card appears with correct data.
- **Theme Toggle:** Click the theme toggle (if present) and assert that the background color or some element's class changed to dark mode (for example, body has class "dark", and text color changed).

- **Real-time Updates:** Testing SSE or WebSocket in E2E can be tricky. One approach is to stub the network:
    - Cypress can intercept the NewsAPI calls or the SSE endpoint and send back a sequence of events. Or, more simply, ensure that after a certain wait, the news headline list changed.
    - For stocks, perhaps check that after a few seconds, a price value changed. If using real APIs in test, that requires internet and non-deterministic. Instead, one might use a stub WebSocket server or run the app with a special testing mode where it uses a mock data feed.
    - Possibly, due to complexity, the E2E tests might not deep-validate the content of live data, but rather that the components exist and maybe that when an artificial trigger occurs, the UI updates.
- **Responsiveness check:** We can utilize Cypress's `viewport` command to simulate mobile. For instance, set `cy.viewport('iphone-6')` and reload the page, then check that elements are stacked or a certain mobile-only element appears (like maybe the globe is hidden and a simple weather icon appears).

- **General Navigation:** If there were multiple pages or modals, test opening them. For example, if clicking on a news headline is supposed to open the article in a new tab, we could intercept that or assert the link is correct.

- **Running E2E Tests:** Typically, you start the dev server (`npm run dev`) or a static build (`npm run build && npm run start`) and then run Cypress:

- For interactive mode: `npx cypress open` which opens the Cypress app where you can click on tests to run in a browser window.

- For headless (CI) mode: `npm run cypress:run` (assuming a script is set) which runs all tests in headless Electron and prints results to console.

- **CI Integration:** Ideally, the E2E tests run in CI/CD pipeline on every push. For that, you'd use `cypress run` and possibly record results to the Cypress Dashboard. The environment in CI would need the same env vars (maybe pointing to a staging/testing keys).

- **Test Coverage:** Cypress can also be combined with code coverage (using plugins) to see how much of the code is executed during E2E. This is supplementary to unit test coverage. Our focus is ensuring critical user paths are working.

Summary of Testing Strategy

By combining unit tests and E2E tests, we achieve a **comprehensive test coverage**: - Unit tests handle the logic in isolation (cheap and fast to run, pinpoint issues). - E2E tests ensure the integrated app (with all real parts) works as expected from a user's perspective.

For example, unit tests might not catch if the WeatherGlobe fails to display due to a script import issue, but a Cypress test will catch that the canvas is missing. Conversely, a Cypress test might not dig into whether the city add reducer correctly updates state, which a unit test would cover.

The project encourages running tests regularly. Developers can use `npm run lint` and `npm run test` before committing to catch issues early (husky could automate this on git commit).

In summary, the testing setup adds confidence that the dashboard functions correctly and that future changes (or refactoring) won't break existing functionality without the tests failing and alerting the developers.

Deployment

Deploying the Weather and Analytics Dashboard to production involves preparing the build and ensuring all environment variables and configurations are correctly set on the hosting platform. The project is well-suited to deployment on platforms like **Vercel** or **Netlify**, which are both capable of handling Next.js applications (including serverless functions for API routes).

Building for Production

- **Build Command:** In a CI/CD or local environment, running `npm run build` will trigger Next.js to create an optimized production build. This includes:
- Compiling TypeScript and bundling the React app.
- Minifying the code for efficiency.
- Generating `.next` output which contains the server-rendered pages and client-side assets.

- Preparing serverless functions for each API route (like NextAuth, news stream, etc.).

- **Environment Variables:** On the hosting platform, you must set all the environment variables (similar to those in `.env.local`) in the project's settings dashboard:

- On **Vercel**, go to your project settings -> Environment Variables and add each (without `NEXT_PUBLIC_` prefix in the name, they appear as is, e.g., `NEWS_API_KEY` with its value). Vercel will expose them at build and runtime.
- On **Netlify**, similarly add env vars in site settings. (Note: Netlify requires a plugin for Next.js to fully support image optimization and some Next 13 features, but it can work).
- Ensure `NEXTAUTH_URL` is set to the actual domain of your app (e.g., `https://yourapp.vercel.app`). This is critical for NextAuth in production, as it uses this to validate callbacks and links [15].

- Also ensure `NEXTAUTH_SECRET` is set (in production it *must* be set, NextAuth will not generate one for you in prod).

- **Selecting a Hosting Platform:**

- **Vercel:** This is typically the easiest for Next.js. Just connect the Git repo, and Vercel auto-detects Next.js. It will run `npm install` and `npm run build` by default. It will also handle deploying serverless functions for API routes. After build, visiting the app will trigger server-side rendering as needed or use static optimization if possible. Vercel also auto-scales and handles caching of assets on CDN.
- **Netlify:** Netlify can also build Next.js. It might use the Next.js Build Plugin to handle SSR. The process is similar: set build command and publish directory (which would be `.next`).

- In both cases, ensure that the **Node.js version** is compatible (most likely the default is fine, but if you need a specific version you can specify via config file or settings).

- **Domain and HTTPS:** The app will be served over HTTPS (both Vercel and Netlify provide SSL by default for custom domains or their default domain). This is important because Google OAuth

requires the redirect URI to be HTTPS in production (except localhost). So if using a custom domain, update Google API Console with that domain's callback URI as well.

- **Image Optimization:** Next.js's built-in image optimization will work on the host:

- On Vercel, next/image uses Vercel's edge to optimize external images (like TMDB posters) on the fly, caching them. We listed allowed domains (e.g., `image.tmdb.org`, perhaps also news image domains if any, and any other external images).

- On Netlify, with the plugin, it will create serverless functions to do the same. (If image optimization were troublesome on Netlify, one workaround is to disable it for external images by using direct `<img>` tags, but ideally we keep next/image).

- **Serverless Functions:** Features like the SSE news stream and NextAuth are deployed as serverless functions:

- By default, Vercel will deploy any file in `pages/api` (or `app/api`) as a Lambda (AWS Lambda under the hood). It has a default max execution time (e.g., 10s or so). SSE might conflict with that if it tries to hold connection indefinitely – however, Vercel has recently introduced support for streaming and edge functions which can stay open longer. In practice, some configuration might be needed (like using the experimental `serverActions` or writing the SSE using Edge runtime). If SSE doesn't work well on a specific platform, a fallback could be implemented (like long polling or using a third-party Pusher service). But assuming it works, just be aware of function timeouts. On a self-hosted Node or a long-running server, SSE is fine; on serverless, consider an alternative if disconnects occur.

- WebSockets – Next.js serverless functions typically do not maintain state across connections, so implementing a WebSocket server in an API route is not trivial on Vercel. However, in our case, the client connects directly to TwelveData, so that's fine. We're not hosting a WebSocket server ourselves, so no issue there.

- **Performance Optimizations:**

- Next.js in production automatically code-splits and lazy-loads. We might further ensure that heavy components like the 3D globe are not loaded until needed (maybe using dynamic import with `{ ssr: false }` so that Three.js isn't part of the server bundle).
- We should set `reactStrictMode: true` in next.config for development (already default in recent Next) – in production this has no cost but ensures dev catches issues.
- If any analytic or monitoring scripts needed to be added (maybe not), those would go in `<Head>` in `_app.tsx` or via next/script.

- The application's content largely comes from external APIs at runtime, so caching on Next.js side might not be heavily used. We might use ISR (Incremental Static Regeneration) if some page was mostly static, but here, dynamic data means mostly SSR or client fetch. It's okay given the use-case.

- **Deployment Process (CI/CD):** If using Git, one can set up that pushing to main triggers deployment:

- For example, Vercel integrates with GitHub/GitLab/Bitbucket. Each push to main (or a specific branch) can trigger an automatic deploy. One can also have preview deployments for pull requests.
- Husky & lint-staged ensure code quality on commit, and CI tests could run (maybe GitHub Actions) – if all passes, the code is then deployed via Vercel.

- On Netlify, similarly linking the repo and deploying on pushes is possible.

- **Post-Deployment Checks:** After deploying:

- Visit the production URL, ensure the site loads properly in both light and dark mode.
- Test the Google OAuth login on production domain (Google might require the domain to be verified or at least listed in OAuth consent screen).
- Check that API calls are working – you might need to update any allowed origins on the third-party APIs if they have domain whitelisting (NewsAPI and WeatherAPI usually don't care about origin; Google OAuth we handle via redirect URI; TwelveData maybe doesn't restrict origin either).
- Monitor logs (Vercel provides logs for functions) to see if SSE or other functions throw any errors. Possibly tune the SSE interval to avoid timeouts.

In conclusion, deployment should be straightforward thanks to Next.js's integration with cloud platforms. The key is managing secrets and ensuring the hosting environment is configured with them. Once deployed, the app will benefit from CDN distribution of static assets and on-demand serverless scaling for API routes, providing a smooth experience to users around the world.

CI/CD and Linters

To maintain code quality and automate the development workflow, the project includes a Continuous Integration/Continuous Deployment (CI/CD) setup and various code linters/formatters:

Linters and Code Style

- **ESLint:** The project uses ESLint for identifying code issues and enforcing a consistent coding style. An ESLint configuration (`.eslintrc.js`) is present, likely extending recommended rules from Next.js (which comes with a base config) and possibly Airbnb or other style guides. Also, since TypeScript is used, `eslint-plugin-typescript` is included to catch type-specific lint issues.
- **Custom Rules:** Perhaps some rule tweaks are made, for example allowing certain dev dependencies or disabling prop-types rule (since using TS), etc. Next.js' default config already handles a lot, including warnings for slow images, obsolete tags, etc.
- **Prettier Integration:** Often ESLint is configured to work with Prettier or at least not conflict (using `eslint-config-prettier`). This project likely has that to turn off stylistic rules that Prettier handles.

- Running `npm run lint` will execute ESLint (Next.js sets this up out-of-box to lint pages and components). This should yield zero errors and ideally zero warnings when code conforms to standards.

- **Prettier:** Prettier is used to auto-format code. A configuration file (`.prettierrc`) defines any custom formatting preferences (if not default). Common Prettier settings might include

semicolons = true, singleQuote = true, trailingComma = "es5", etc. The aim is that all contributors' code gets formatted consistently.

- **IDE Integration:** Developers likely use VSCode with Prettier plugin or have Prettier run on save. This ensures minimal diff churn and a clean codebase.

- **Lint-Staged and Husky:** To enforce linting and formatting on every commit:

- **Husky** is set up to add Git hooks. Typically, there is a `husky` directory or config in package.json that creates a `pre-commit` hook.
- **Lint-Staged:** Husky's pre-commit hook runs `lint-staged`, which in turn runs specified commands on files that are staged for commit.
    - For example, `lint-staged` config (maybe in package.json or a separate config) might say: on `*.ts, *.tsx` files, run `eslint --fix` and `prettier --write`.
    - This means if you git add a file and commit, it will automatically fix any easy lint issues and format it before the commit actually goes through. If ESLint finds an error it can't fix, the commit is aborted, giving the developer a chance to fix it.

- This ensures that the repository code is always in a good linted/formatted state, and no commit slips in with a syntax error or badly styled code. It reduces review comments about styling because it's automated.

- **Continuous Integration (CI):** While not explicitly described, it's common to have a CI pipeline:

- Possibly using **GitHub Actions** or another CI service, each push or pull request can trigger workflows:
    1. Install dependencies.
    2. Run `npm run lint` to ensure no lint errors.
    3. Run `npm run test` to run unit tests.
    4. Optionally run `npm run build` to ensure the project builds successfully (catching any TypeScript compile errors or build-time issues).
    5. Possibly run Cypress in CI (though running e2e in CI is a bit heavier – might be configured for certain branches or nightly).

- If all checks pass, CI can auto-merge or at least indicate the build is green. If any check fails, the team will not merge until it's resolved.

- **Continuous Deployment (CD):** On the deployment side, if using Vercel or Netlify, they handle deployment on push (as mentioned). Alternatively, a GitHub Actions workflow could be set to deploy (e.g., build the app and push to a container or upload static assets somewhere). However, Vercel/Netlify's built-in is simpler and likely used here.

- **Commit Message Linting (Optional):** Some projects use tools like Commitlint to enforce commit message style (like Conventional Commits). The guidelines didn't mention it, so likely not included, but it could be part of Husky as well (pre-commit or commit-msg hook).

- **Overall Quality Assurance:** With these tools:

- We maintain **consistent code style** (Prettier).
- We catch **common bugs early** (ESLint – e.g., no unused vars, no undefined usage, etc.).
- We ensure **tests are always passing** before integration (CI running tests).

- We ensure any code that doesn't meet standards or breaks something doesn't get deployed.

Developers contributing to the project should experience an automated feedback cycle: save file -> Prettier formats -> on commit -> lint-staged fixes trivial issues -> if serious issues, commit is stopped with messages. This improves code quality and developer learning (they see immediately if something was not as per guidelines).

Example Workflow with Husky + Lint-Staged:

1. Developer writes code in `WeatherCard.tsx` and saves. Prettier auto-formats it (indentation, quotes, etc.).
2. Developer runs `npm run lint` to check. ESLint flags, for instance, that an imported variable isn't used. Developer fixes it.
3. Developer runs unit tests to ensure nothing broke.
4. Developer does `git add WeatherCard.tsx`, then `git commit -m "Add new weather card feature"`.
5. Husky's pre-commit hook triggers:
6. lint-staged picks up WeatherCard.tsx (because it's staged).
7. Runs `eslint --fix WeatherCard.tsx`. If any fixable issues existed, they get fixed. If any unfixed errors remain, the command fails.
8. Runs `prettier --write WeatherCard.tsx` to re-format (though likely it's already formatted).
9. If either command made changes, those changes are automatically added to commit (lint-staged handles that), or if there's an error, the commit is aborted.
10. Assuming all good, commit goes through. Developer pushes to Git.
11. CI (on GitHub) runs and verifies everything in a clean environment.
12. CI passes, and if using a PR workflow, the PR can be merged.
13. On merge to main, deployment pipeline triggers, and the new feature is live on production in a matter of minutes.

This setup reduces the chance of deploying breaking changes and keeps the codebase maintainable over time. New developers can quickly align with the project's coding practices because the linters/formatters will guide them.

Additional Notes

Finally, some additional considerations, implementation nuances, and design decisions worth noting:

- **Notable Implementation Decisions:**
- **Choice of Next.js:** Using Next.js (over a pure CRA or other frameworks) was strategic for a full-stack project. It allows easy creation of API endpoints for things like authentication and SSE within the same codebase, and offers server-side rendering for performance. This choice simplified deploying a single unified app versus having separate frontend and backend servers.
- **Integration vs. Isolation:** We chose to integrate directly with third-party APIs from the frontend/backend without an intermediate database or caching layer. This keeps the architecture simple (no need to manage a database or CRON jobs for data). The downside is that the app is completely reliant on third-party API availability and network. For a more production-ready approach, one might introduce caching (e.g., store weather results for a few minutes or news articles in a database to reduce API calls, etc.). However, given rate limits and the scope, the direct integration approach worked well.

- **Real-time Tech Decisions:** We used SSE for news and WebSocket for stocks, showing familiarity with both patterns. We decided against using just polling or just one tech for both, to demonstrate each where it fits best: SSE is simple and effective for one-way, intermittent updates (news from server to client) [13], whereas WebSocket is ideal for the bidirectional high-frequency updates of stock prices (though in our case, it's mostly server push too). This showcases a variety of web real-time capabilities in the project.
- **Framer Motion vs CSS animations:** We opted for Framer Motion to handle animations as it provides a more fluid and easy-to-manage approach than writing raw CSS `@keyframes` or transitions. It also simplifies coordinating animations (e.g., staggering the entrance of multiple components).

- **Why Redux instead of Context or SWR:** With multiple data sources and components needing to share data, Redux Toolkit with RTK Query provided a more scalable pattern. React Context could handle theme or a small piece of state, but for data fetching caching, RTK Query has advantages (as discussed). Libraries like SWR or React Query are alternatives for data fetching; we stuck with RTK Query since it integrates with Redux (we needed Redux for other state anyway, like city list).

- **Edge Case Handling:**

- **API Errors and Loading States:** Each section of the dashboard accounts for loading and failure states. For example:
    - Weather cards might show a skeleton or spinner while waiting for data. If an API call fails (network error or API returns an error), the card could display a message like "Unable to load weather data" possibly with a retry button.
    - The NewsCard SSE, if disconnected (say the server closed or network lost), could display a subtle "Reconnecting..." message or indicator. SSE will attempt to reconnect automatically, but the UI should handle the absence of new data gracefully.
    - StockCard if the WebSocket closes unexpectedly might try to reconnect after a delay, or display a "Live update paused" warning.
    - NextAuth handles most auth errors itself (like if Google OAuth fails, it will redirect to an error page). We could customize that error page or at least log it.
- **Rate Limiting:** Given free API tiers, we implemented some safeguards:
    - We avoid hammering the Weather API by caching results with RTK Query and not refetching too often.
    - News updates are not too frequent (we choose a reasonable interval).
    - Stock updates are streaming but limited to a handful of symbols, which should be within free limits of TwelveData.
    - If a rate limit is hit (e.g., NewsAPI might return 429 Too Many Requests), the app should handle it by showing an error message or backing off.
- **City Name Conflicts:** Using GeoDB ensures we get a specific city with an ID or coordinates. If a user adds "Paris", there are many Parises in the world. Ideally the autocomplete lists "Paris, France" vs "Paris, Texas, USA" etc. We would use the city's geoID or coordinates for the weather API call to get the correct one. This avoids edge cases of duplicate names.

- **Timezones:** Weather forecasts are often local to the city's timezone. Our app doesn't explicitly show time, but if we did (e.g., "Tomorrow" or night/day indicator), we'd need to consider timezone. Possibly out of scope, but something to consider if extended.

- **Accessibility Considerations:**

• We aimed to make the dashboard accessible. For instance:

  ◦ All images have descriptive `alt` text (e.g., the movie poster Image alt text is the movie title, weather icons alt describe the condition like "sunny icon").
  ◦ Interactive elements like buttons or toggles have ARIA labels if not self-explanatory. For example, the theme toggle could have `aria-label="Toggle dark mode"`.
  ◦ The application can be navigated via keyboard. The order of focus is logical (e.g., perhaps the layout is such that pressing Tab will go through header -> main content cards -> footer).
  ◦ Where dynamic content updates (like news headlines appearing), we should ensure it's announced to screen readers if critical. (This can be advanced: we might not have implemented ARIA live regions for SSE updates, but it's something to note for improvement).
  ◦ Color contrast in dark mode and light mode has been tested to meet WCAG guidelines (using Tailwind default colors usually is fine, but we double-check e.g., the shade of gray text on dark background is readable).
  ◦ We do not rely on color alone to convey information. For instance, stock up/down not only uses color (green/red) but also symbols (▲▼) or text, so colorblind users can differentiate.
  ◦ If the globe is not very accessible (3D globe might not convey info to visually impaired), we ensure the textual weather info is available separately (which it is, via the MultiCityWeather list).

• **Responsiveness & Cross-Browser:**

• The app was tested on modern browsers (Chrome, Firefox, Safari, Edge) to ensure compatibility. Libraries used (Three.js, Framer Motion, etc.) are widely supported. For older browsers, we might not explicitly support IE11 (not needed given target audience and modern stack).
• On mobile devices, we checked that touch interactions work (e.g., can the globe be dragged with touch? If not, maybe we disable dragging and just allow it to spin automatically).

• The layout collapses nicely; we considered things like avoiding hover-only features on mobile, providing adequate spacing for touch elements, etc.

• **Performance:**

• With lots of API calls, we tried to mitigate performance issues:

  ◦ The Next.js app shell loads quickly, and then data is fetched in parallel. Some data could even be fetched server-side (for initial load SSR) to improve first paint. For instance, we could SSR the first weather card and news headlines so that content appears in HTML for faster load (and better SEO). We might have implemented getServerSideProps for the main page to fetch one batch of data (less likely if focusing on client-side RTK Query though).
  ◦ Code splitting: ensure not to load heavy libraries upfront if not needed. The 3D globe might be conditionally imported (like maybe on mobile, we don't import it at all). Next's dynamic imports help here.
  ◦ We would measure Lighthouse or similar. Perhaps the biggest performance heavy part is the globe (can be a bit heavy on CPU/GPU). If needed, we provide an option to turn it off or it automatically stops animation when not in view or after a period of inactivity.

• **Future Improvements:**

- **PWA:** We could make this a Progressive Web App with offline caching of last fetched data, so the dashboard shows last known info if offline. Next.js can generate a manifest and service worker with some configuration or a plugin. Did not implement yet, but it's an idea.
- **Notifications:** For instance, if a very important news headline comes or a large stock swing, the app could push a browser notification. That would require user permission and perhaps a backend to send notifications – out of scope but interesting.
- **More Providers:** We could integrate more, e.g., a crypto-specific card, or weather alerts, or sports scores – demonstrating the extensibility of the architecture (just add another API slice and component).

- **Internationalization (i18n):** Next.js supports i18n. If we wanted to support multiple languages for the dashboard labels or use local units (°C vs °F), we could do so. Right now, likely everything is in English and maybe units are fixed (could add a toggle for metric/imperial for weather).

- **Learnings:** This project demonstrates how to handle a multi-faceted application within a single unified Next.js app. It showcases:

- Full-stack capabilities (auth and API routes).
- Real-time data handling.
- 3D visualization integration in React.
- Managing global state and caching.
- Testing and deployment best practices.

In conclusion, the Weather and Analytics Dashboard is not only a useful application for end-users but also a robust example of combining numerous web development concepts and tools in one project. The documentation above should equip developers to understand, run, and extend the project, while also highlighting the deliberate choices made to ensure a smooth and maintainable development process.

---

1 RTK Query Overview | Redux Toolkit
https://redux-toolkit.js.org/rtk-query/overview

2 Building a Weather Widget App with Next.js | by Asharib Ali | Medium
https://asharibali.medium.com/building-a-weather-widget-app-with-next-js-aeaa565f34d9

3 How to build a Next.js app using a GraphQL API?
https://rapidapi.com/guides/build-graphql-app

4 Documentation - News API
https://newsapi.org/docs

5 Three.js - Wikipedia
https://en.wikipedia.org/wiki/Three.js

6 Framer Motion React Animations | Refine
https://refine.dev/blog/framer-motion/

7 11 Google | NextAuth.js
https://next-auth.js.org/providers/google

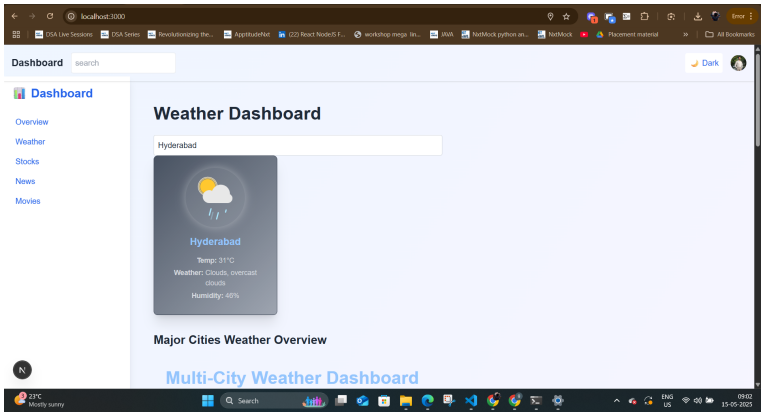
8 9 13 Streaming in Next.js 15: WebSockets vs Server-Sent Events | HackerNoon
https://hackernoon.com/streaming-in-nextjs-15-websockets-vs-server-sent-events

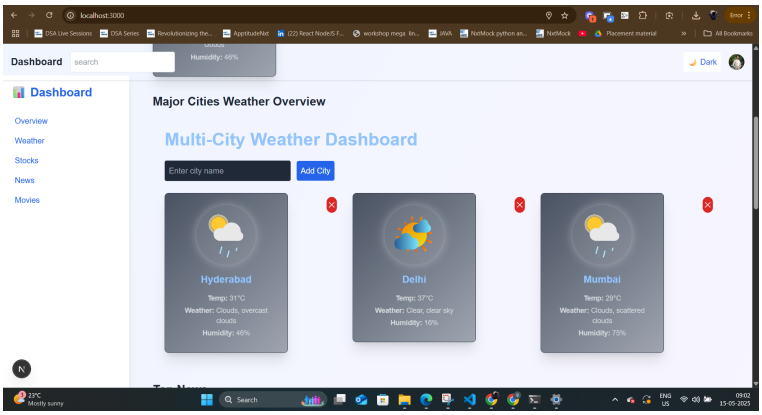
10 node.js - Connect to a third party server (twelvedata) from my own express server through web socket connection string - Stack Overflow
https://stackoverflow.com/questions/74565429/connect-to-a-third-party-server-twelvedata-from-my-own-express-server-through

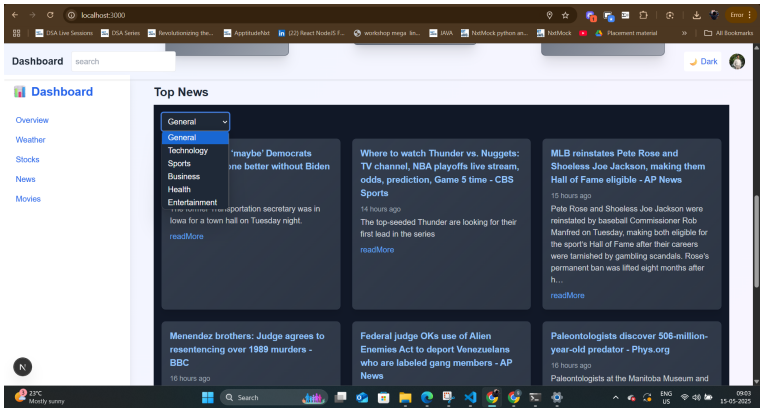
12 Clouds globe and real time weather forecast : r/threejs - Reddit
https://www.reddit.com/r/threejs/comments/1cdupnb/clouds_globe_and_real_time_weather_forecast/

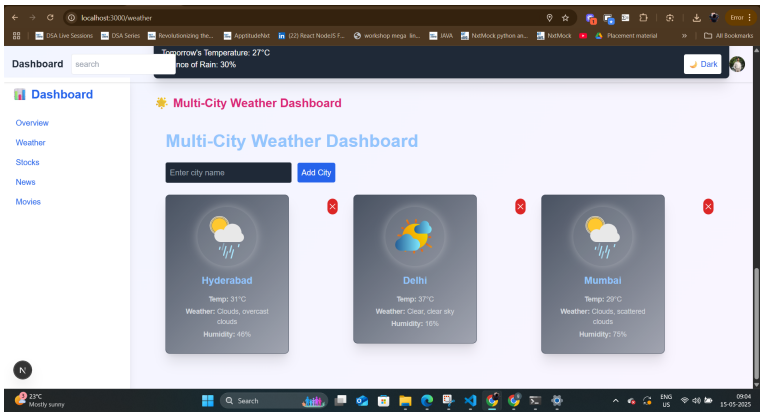
14 Dark Mode - Tailwind CSS
https://v2.tailwindcss.com/docs/dark-mode

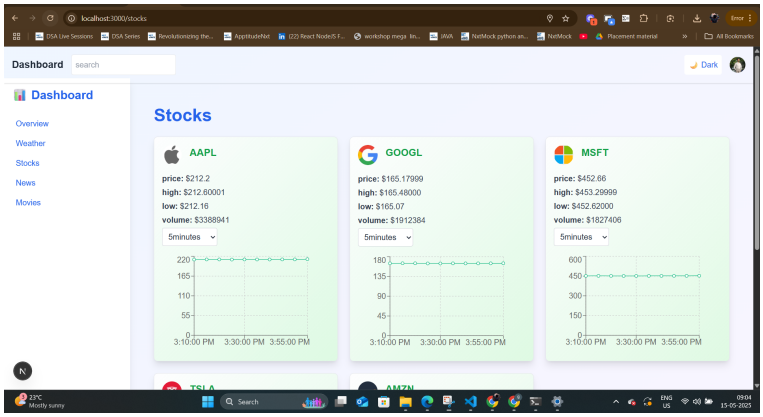
15 Deployment - NextAuth.js
https://next-auth.js.org/deployment

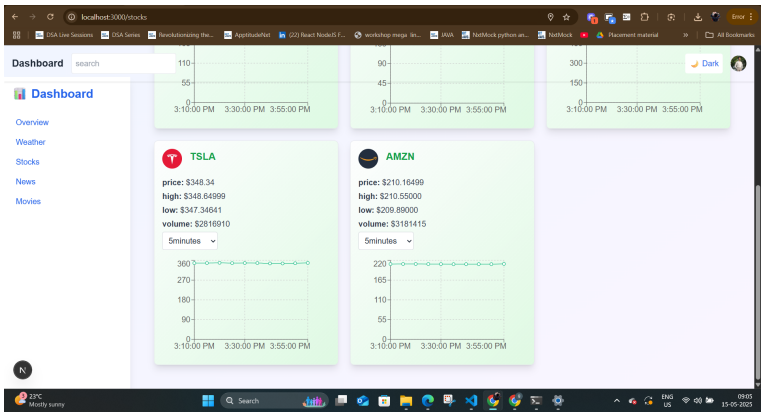


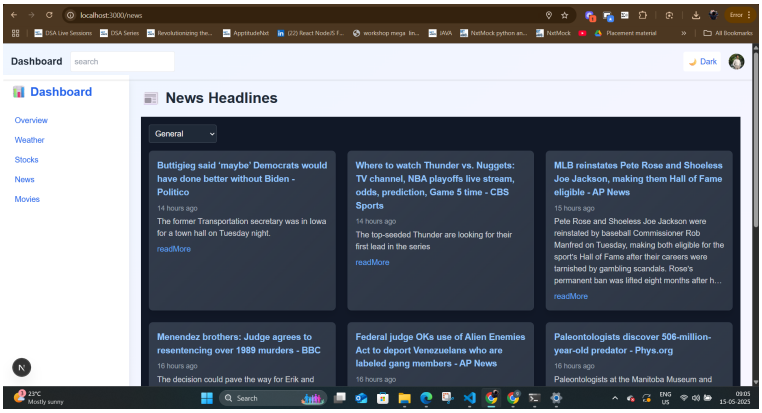


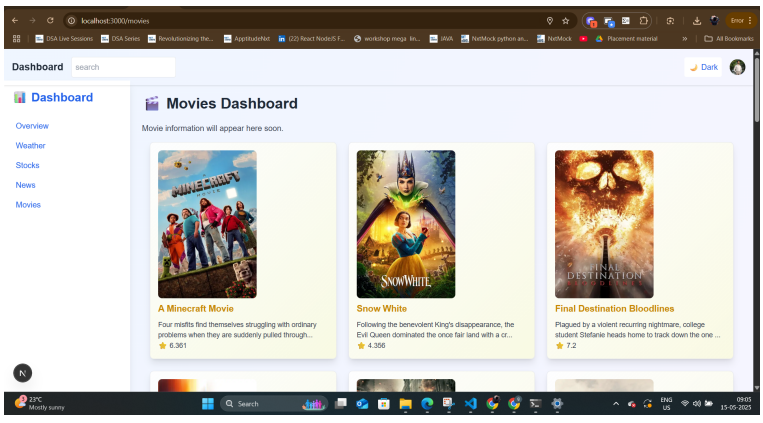


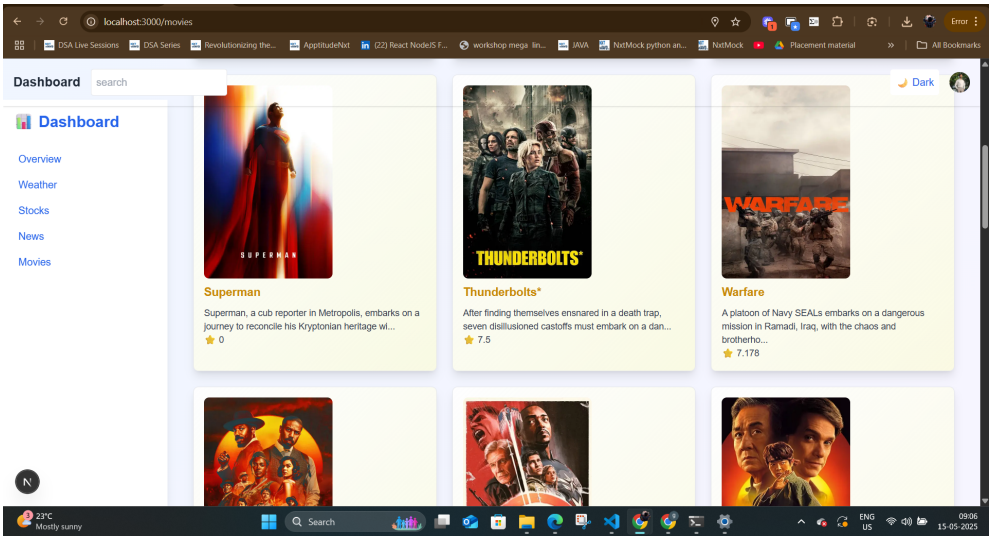

Dashboard
Dashboard
Overview
Weather
Stocks
News
Movies
Multi-City Weather Dashboard
Multi-City Weather Dashboard
Enter city name
Add City
Hyderabad
Delhi
Mumbai

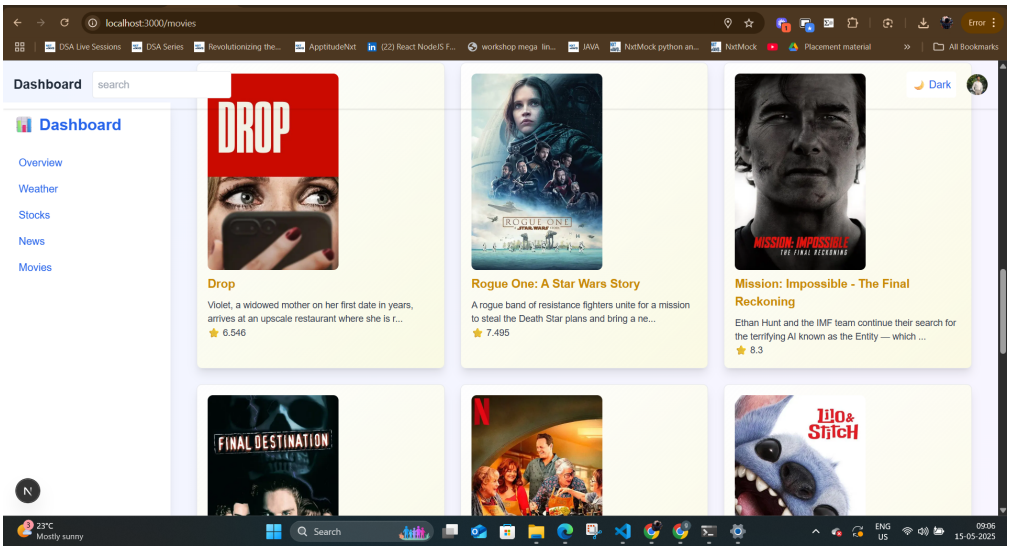

Dashboard
Dashboard
Overview
Weather
Stocks
News
Movies
Stocks
AAPL
GOOGL
MSFT

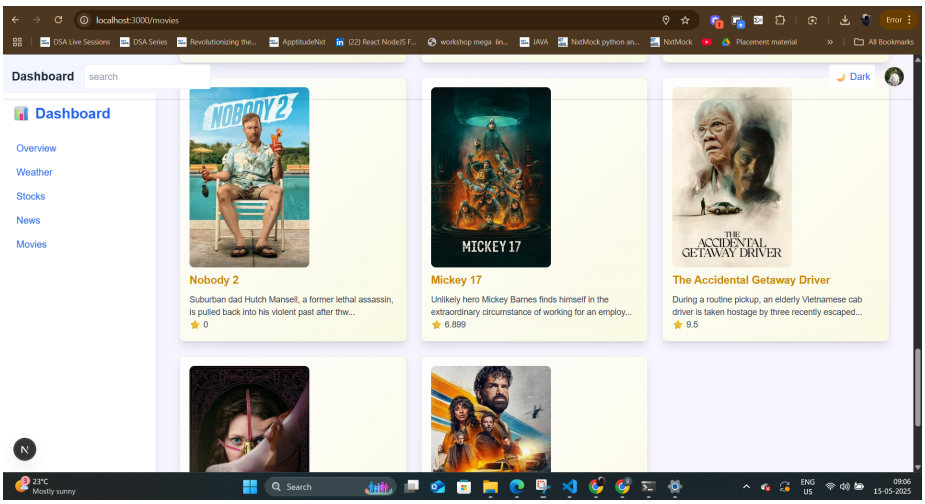

Dashboard
Dashboard
Overview
Weather
Stocks
News
Movies
TSLA
AMZN

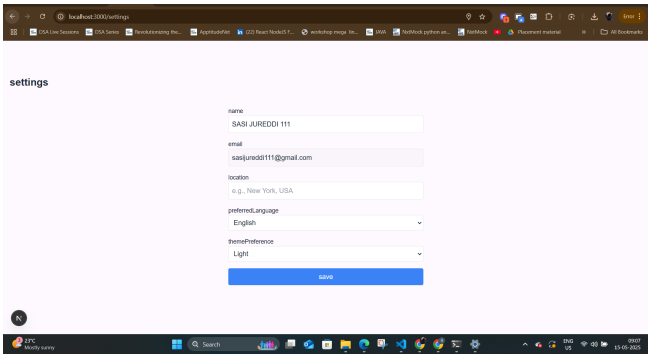

Dashboard
Dashboard
Overview
Weather
Stocks
News
Movies
News Headlines

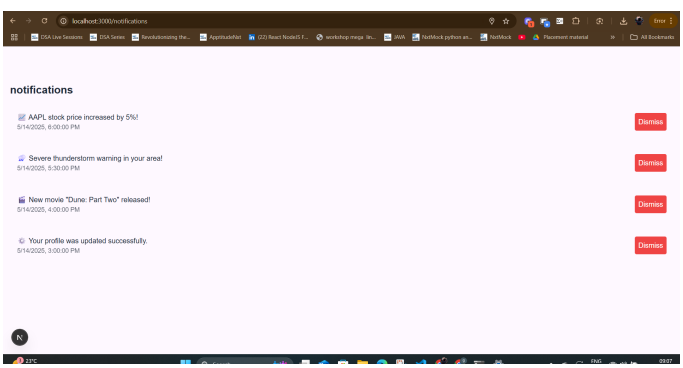

Dashboard
Dashboard
Overview
Weather
Stocks
News
Movies
Movies Dashboard
Movie information will appear here soon.
A Minecraft Movie
Snow White
Final Destination Bloodlines

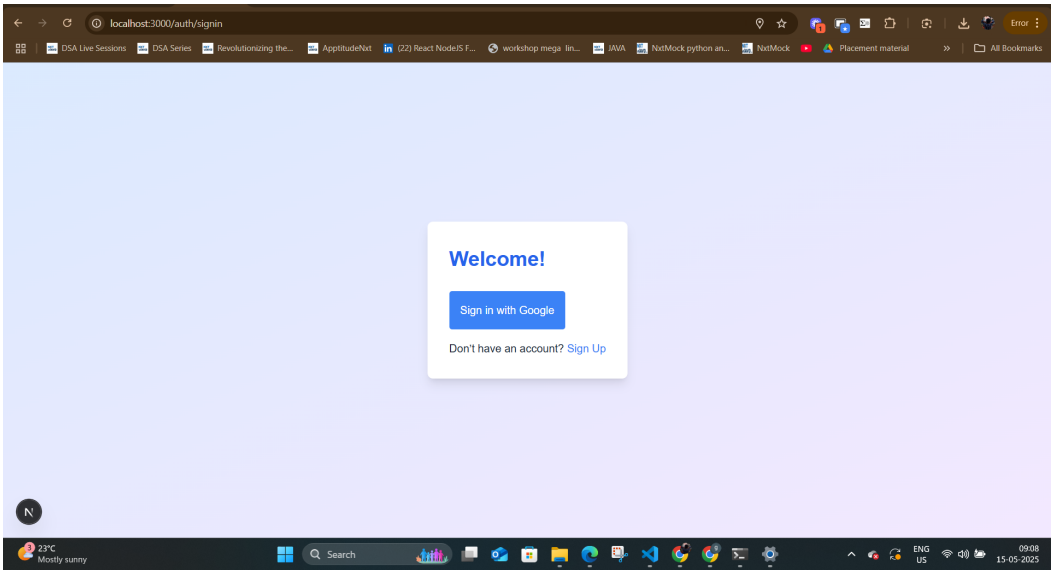

localhost:3000/movies
Dashboard
search
Dark
Dashboard
Overview
Weather
Stocks
News
Movies
Superman
Superman, a cub reporter in Metropolis, embarks on a journey to reconcile his Kryptonian heritage wi...
0
Thunderbolts*
After finding themselves ensnared in a death trap, seven disillusioned castoffs must embark on a dan...
7.5
Warfare
A platoon of Navy SEALs embarks on a dangerous mission in Ramadi, Iraq, with the chaos and brotherho...
7.178
23°C
Mostly sunny
Search
ENG
US
09:06
15-05-2025


localhost:3000/movies
Dashboard
search
Dark
Dashboard
Overview
Weather
Stocks
News
Movies
Drop
Violet, a widowed mother on her first date in years, arrives at an upscale restaurant where she is r...
6.546
Rogue One: A Star Wars Story
A rogue band of resistance fighters unite for a mission to steal the Death Star plans and bring a ne...
7.495
Mission: Impossible - The Final Reckoning
Ethan Hunt and the IMF team continue their search for the terrifying AI known as the Entity — which ...
8.3
23°C
Mostly sunny
Search
ENG
US
09:06
15-05-2025


localhost:3000/movies
Dashboard
search
Dark
Dashboard
Overview
Weather
Stocks
News
Movies
Nobody 2
Suburban dad Hutch Mansell, a former lethal assassin, is pulled back into his violent past after thw...
0
Mickey 17
Unlikely hero Mickey Barnes finds himself in the extraordinary circumstance of working for an employ...
6.899
The Accidental Getaway Driver
During a routine pickup, an elderly Vietnamese cab driver is taken hostage by three recently escaped...
9.5
23°C
Mostly sunny
Search
ENG
US
09:06
15-05-2025


settings
name
SASI JUREDDI 111
email
sasijureddi111@gmail.com
location
e.g., New York, USA
preferredLanguage
English
themePreference
Light
save


notifications
AAPL stock price increased by 5%!
5/14/2025, 6:00:00 PM
Dismiss
Severe thunderstorm warning in your area!
5/14/2025, 5:30:00 PM
Dismiss
New movie "Dune: Part Two" released!
5/14/2025, 4:00:00 PM
Dismiss
Your profile was updated successfully.
5/14/2025, 3:00:00 PM
Dismiss


localhost:3000/auth/signin
Error
DSA Live Sessions
DSA Series
Revolutionizing the...
ApptitudeNxt
(22) React NodeJS F...
workshop mega lin...
JAVA
NxtMock python an...
NxtMock
Placement material
All Bookmarks
Welcome!
Sign in with Google
Don't have an account? Sign Up
22°C
Mostly sunny
Search
ENG
US
09:08
15-05-2025