Live Service Monitoring Dashboard

Objective

To assess your ability to design and build a polished, robust, responsive, and modern frontend application. This task holistically evaluates your approach to frontend development, from architectural patterns for data fetching, caching, and state management in a real-world scenario to the execution of a professional, production-ready user interface.

The Scenario

You are tasked with building a proof-of-concept for a new internal dashboard at "MonitoCorp." This dashboard will be used by our Site Reliability Engineering (SRE) team to monitor the health of our various microservices. The key requirement is that the dashboard must feel **live** and **highly responsive** and **professionally designed**, providing engineers with up-to-the-second information without requiring them to manually refresh the page.

Core Feature Requirements

1. **Main Dashboard View:**
    - Display a list of all monitored services in a clean, well-organized table.
    - Each row should show the service's name, its type (e.g., API, Database), and its current `status` (e.g., Online, Offline, Degraded). Use visual cues (like status badges or icons) to make the status instantly recognisable.
    - The application should feel fast. Navigating back and forth between views should be seamless, and repeated views of the service list should be instantaneous, leveraging caching to avoid unnecessary network requests for data that hasn't changed.
2. **Live Status Updates:**
    - The `status` of a service is volatile and can change at any moment. The dashboard must automatically poll for status updates for all visible services periodically (e.g., every 15 seconds), updating the UI without a full-page re-render.
    - The core list of *configured services* changes infrequently. Your data fetching strategy should reflect this, avoiding refetching the entire list of services on every status poll.
    - When a user navigates away from the browser tab and later returns, the data displayed must be immediately refreshed to ensure they are seeing the most current state.
3. **Service Management (CRUD):**
    - Provide an intuitive way for users to **add**, **edit**, and **delete** services. These actions could be triggered from a modal or a dedicated form view.
    - After any of these actions (Create, Update, Delete), the main service list must reflect the changes **immediately and optimistically**, without requiring a full page reload. If an operation fails, the UI should revert to its previous state and notify the user.
4. **Service Details & Historical Events:**
    - Clicking on a service in the list should navigate the user to a dedicated "Service Details" view with a smooth transition.
    - This view should display the service's details and a list of its historical status events (e.g., "Service went offline at 2023-10-27 10:00:00 UTC").
    - The list of historical events can be very long. Implement it with **infinite scrolling** to ensure high performance, complete with appropriate loading indicators.

    - **UX Enhancement:** While fetching fresh data for the details view, avoid a blank loading screen. Show the last-known (potentially stale) data for the service while updating in the background.
5. **Filtering and Searching:**
    - The main dashboard should allow users to filter the list of services by their `status` and include a text input to search by `name`...
    - Filtering and searching should be performant and update the view without a full page reload.

Technical Specifications

- **Framework:** Use **React/Next**
- **API:** You are not expected to build a backend. Use a library like `msw` (Mock Service Worker) or create a fake API module that simulates network latency (300-1000ms) and potential errors.
- **Styling:** Your choice of styling library is open (e.g., Material-UI, Ant Design, Tailwind CSS, Styled Components), but the final result must demonstrate a high level of polish. We expect a clean, intuitive, and aesthetically pleasing interface that would be suitable for a production environment. Attention to layout, spacing, typography, and visual hierarchy is crucial.
- **State Management:** This is the core of the evaluation. You have complete freedom to choose your tools and patterns.

Mock API Endpoints

Simulate these endpoints. Introduce a random delay (e.g., 300ms - 1000ms) and a ~5% chance of failure for any API call to test error handling and loading states.

- `GET /api/services`: Get a list of all services.
    - Supports pagination query params: `?page=1&limit=10`
    - Supports filtering: `?status=Online` or `?name_like=user`
- `GET /api/services/:id`: Get a single service by its ID.
- `POST /api/services`: Create a new service. Takes `{ name, type }`. Returns the new service object with an `id` and default `status`.
- `PUT /api/services/:id`: Update a service. Takes `{ name, type }`. Returns the updated service.
- `DELETE /api/services/:id`: Delete a service. Returns a `200 OK` empty response.
- `GET /api/services/:id/events`: Get historical events for a service.
    - Supports pagination query params: `?page=1&limit=20`

Evaluation Criteria

We will be evaluating your submission based on the following:

1. **State Management Architecture:**
    - Your overall strategy for managing server state vs. client state.
    - How you handle data fetching, caching, background updates, and synchronisation.
    - The efficiency and elegance of your solution in meeting the “live” data requirements. This is the most important criterion.
2. **UI/UX Design & Execution:**
    - The quality of the final user interface. We are looking for a professional, intuitive, and clean interface.
    - This includes thoughtful use of layout, spacing, typography, loading states, error states, and micro-interactions that contribute to a seamless user experience.

3. **Code Quality & Organization:**
    - Clarity, maintainability, and organization of your code.
    - Appropriate use of modern JavaScript/TypeScript and React features, industry standard design patterns.
    - Component design and separation of concerns.
4. **Documentation:**
    - A `README.md` file that clearly explains your architectural decisions, the libraries you chose (and why), and instructions on how to run the application locally.

Submission

Please provide a link to a public Git repository (e.g., on GitHub) containing your solution. Ensure the `README.md` file is comprehensive.