API Discovery Document

This document details the EHR's local API endpoints, which are built using Next.js API Routes. The API is designed for managing all patient-related data, including records, appointments, and clinical information, in a local SQLite database. It also integrates with an external, read-only FHIR API.

1. Complete List of Endpoints Discovered

Base URL: http://localhost:3000/api (when testing locally)

Endpoint	Method	Description
`/patients`	GET	Fetches a list of all patient records. Supports client-side filtering via a search query parameter.
	POST	Creates a new patient record in the database.
`/patients/[id]`	GET	Retrieves a specific patient's complete record, including all related data.
	PUT	Updates an existing patient's details (e.g., name, gender, birth date).
	DELETE	Deletes a patient record and all associated data across the entire database, including allergies, appointments, etc.
`/allergies`	POST	Creates a new allergy record for a specified patient.
`/appointments`	GET	Retrieves all appointments. Supports optional filtering by date and provider.
	POST	Books a new appointment with details like patient, provider, and time.

`/appointments/[id]`	PUT	Updates the status or details of a specific appointment.
	DELETE	Deletes an appointment record.
`/conditions`	POST	Creates a new medical condition record for a patient.
`/encounters`	GET	Retrieves a list of all patient encounters. Supports an optional patientId query parameter.
	POST	Creates a new encounter record.
`/medications`	GET	Fetches a list of all medication records. Supports an optional patientId query parameter.
	POST	Creates a new medication record.
`/observations`	GET	Retrieves a list of all observations. Supports an optional patientId query parameter.
	POST	Creates a new observation record.
`/procedures`	GET	Fetches a list of all procedures. Supports an optional patientId query parameter.
	POST	Creates a new procedure record.
`/practitioners`	GET	Retrieves a list of all practitioners.
	POST	Creates a new practitioner record.

`/diagnostic-reports`	GET	Fetches all local diagnostic reports.

2. Capabilities and Limitations

- **Local Data**: The local API, built with Next.js and Prisma, provides full **CRUD** (Create, Read, Update, Delete) functionality for all data stored within the application. This data is managed in a SQLite file-based database.
- **External Data**: The system integrates with the public, read-only FHIR API (`https://hapi.fhir.org/baseR4`) to display diagnostic reports and observations. This data is purely for informational purposes and cannot be modified through the EHR dashboard.
- **Integration Approach**: The front-end of the application is responsible for calling both the local and external APIs directly. The architecture is a hybrid client-server model where the client orchestrates data fetching from multiple sources.

Implementation Guide

1. How the Integration Works

The core of the application's data flow is managed via `React` hooks in the client-side components. When a user navigates to a page, `useEffect` hooks trigger API calls to fetch the relevant data. On the Dashboard, `Promise.all` is used to fetch data from multiple sources concurrently to speed up the initial load time.

2. Command Processing Logic

User actions like adding a new patient from the `PatientForm` are handled by form submission handlers. These functions serialize the form data and send a `POST` request to the appropriate API route. The API route then uses Prisma's client to interact with the database.

3. State Management Approach

The application uses a simple, component-based state management approach with `useState` and `useEffect`. Data fetched from APIs is stored in local component state. A `refreshTrigger` state variable is used on the dashboard to force a re-fetch of all data when a user manually clicks the "Refresh" button.

4. Error Handling Strategies

All API routes are wrapped in `try...catch` blocks to prevent crashes. On an error, the server responds with a `500 Internal Server Error` and a JSON object containing an error message. The front-end handles this by setting a loading state to `false` and logging the error to the console.

5. Performance Optimizations Made

- **Debounced Search**: Input fields for searching records (e.g., on the Patients page) use a debounced search function to delay API calls until the user stops typing, reducing network load.
- **Parallel Data Fetching**: The dashboard uses `Promise.all` to fetch data from multiple endpoints simultaneously, improving the perceived performance of the page load.
- **Client-Side Filtering**: For pages like Encounters and Allergies, the entire dataset is fetched once, and subsequent filtering is performed on the client side, avoiding repeated API calls.