Jon Miller

Project Documentation



# Executive Summary

Elderly hospital patients recovering from hip fracture surgeries frequently develop post-surgical delirium, becoming extremely confused and agitated. Delirium is terrifying, leads to bad health outcomes, and costs hospitals millions of dollars. It is also severely underdiagnosed. Practitioners have developed an accurate screening tool to identify at-risk patients. This tool is a questionnaire that nurses fill out, consisting of questions designed to assess risk factors. At the end, the results can be used to calculate a risk score.

This tool has been shown to significantly improve hospitals' ability to identify at-risk patients. However, to-date, it has only existed in a limited capacity as a plugin to software that nurses already use for updating patient histories. Unfortunately, this severely limits its effectiveness. To get meaningful insights into the results, or even to calculate the risk scores, practitioners must tediously port data to excel spreadsheets.

This application addresses these challenges by storing the results of the patient questionnaire in a relational database. It provides practitioners with the ability to view patient responses and risk scores, as well as a dashboard on which to view cumulative results, such as average risk score and risk score trends over time. To ensure broad compatibility across systems, this application runs in a web browser.

The Delirium Assessment Tool empowers nurses working on orthopedic hospital units to more easily and quickly identify patients at risk of developing post-surgical delirium so that they can implement preventive measures.

# Home | Sign In or Sign Up

This screen greets users and prompts them to click a link to either sign into an existing account, or create a new one. See <https://delirium-assessment-tool.fly.dev/>.

A screenshot of a computer

Description automatically generated

# Sign In | Authenticate Using Existing Credentials

Users enter their email address and password to authenticate against their stored account credentials and gain access to the protected sections of the app. See <https://delirium-assessment-tool.fly.dev/login>.

A screenshot of a login form

Description automatically generated

# Sign Up | Create a New Account

Users create a new account by entering their email address and desired password. See <https://delirium-assessment-tool.fly.dev/signup>.

A screenshot of a login form

Description automatically generated

# Dashboard | See Recent Trends

Users can see charts illustrating the past thirty days of patient risk profile results. This includes a radar chart showing results by gender, and another one showing results by age group. The third chart is a stacked bar chart illustrating trends in risk profiles over the past thirty days. See <https://delirium-assessment-tool.fly.dev/home>.

A screenshot of a computer

Description automatically generated

# New Patient | Create a New Patient Record

Users can create a new patient record by entering that patient’s basic demographic information (name, gender, and date of birth). Clicking “Begin Assessment” starts the patient questionnaire.

A screenshot of a computer

Description automatically generated

# Patient Questions | Answer Questions About the Patient’s Mental State

Users iterate over a series of routes that present them with one question per screen to answer concerning the patient’s mental state.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# Patient Result | See a Summary of an Individual Patient’s Screening Result

After completing each question, the user is directed to the patient summary screen, where they can see their risk profile and a summary of their responses.

A screenshot of a computer

Description automatically generated

# All Patients | Access Patients from a Paginated List

Users can see a server-paginated list of all patients, showing their name, demographic information, and assessed risk level. Clicking on a patient’s name will bring the user to that patient’s summary screen.

A screenshot of a computer

Description automatically generated

# Reports

## Risk Type by Gender

This radar chart illustrates the risk type results for each gender over the past thirty days.

A screenshot of a graph

Description automatically generated

## Risk Type by Age Group

This radar chart illustrates the risk type results for each age group over the past thirty days.

A screenshot of a graph

Description automatically generated

## Risk Type Trends

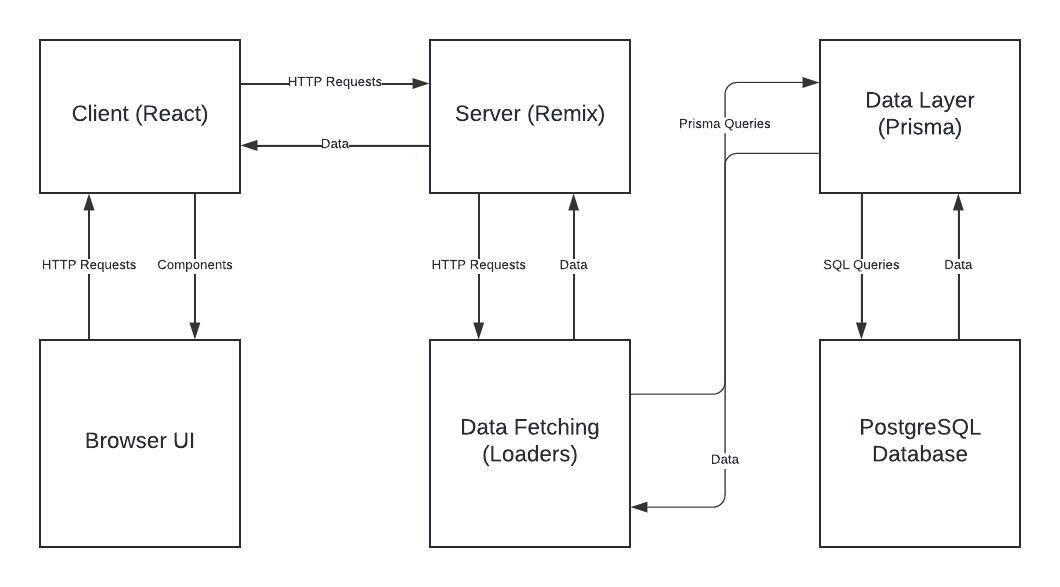
This stacked bar chart shows comparatively how each risk type trended over the past thirty days.

A graph with different colored lines

Description automatically generated

# System Architecture

This application was built with Remix, React, Prisma, and PostgreSQL. Applications built on this technology stack roughly follow a Model-View-Controller (MVC) pattern.



*System Architecture Diagram*

## Client-Side (React)

React (Frontend)The frontend is built with React, rendering dynamic user interfaces. React is responsible for handling user interactions and managing application state, using hooks like useState and useEffect. The React components are routed using Remix’s route-based navigation, which integrates seamlessly with server-side rendering (SSR).

React Router  
Handles client-side routing for navigation between different pages or components. It communicates with Remix to manage URL parameters and page data efficiently.

## Server-Side (Remix)

Remix (Backend Framework)  
Remix serves as the server-side rendering (SSR) framework. It handles the routing, provides data fetching (via loaders), and supports the rendering of React components both on the server and the client. Remix provides a flexible data loading mechanism, ensuring that server-side queries are preloaded and passed to the React components.

Loaders  
Each route can define a loader function in Remix to fetch data from the database or other sources before the page renders. Loaders are executed on the server side but their results are passed into the React components on the client side, ensuring efficient data fetching and SSR.

## Data Layer (Prisma)

Prisma ORMPrisma serves as the Object Relational Mapping (ORM) layer between the application and PostgreSQL. Prisma manages database queries, migrations, and schema management, making it easier to interact with the database using a JavaScript/TypeScript-friendly API. It simplifies the process of querying, inserting, and updating records in the PostgreSQL database.

Prisma Client  
The Prisma Client is generated based on the schema defined in Prisma. The client is used in Remix loaders and actions to interact with the database.

Database Migrations  
Prisma provides a mechanism for managing database schema migrations. It ensures that the structure of the PostgreSQL database is in sync with the application code.

## Database (PostgreSQL)

PostgreSQL  
PostgreSQL is the relational database system used for storing application data. It handles all the data management, including tables for users, patients, responses, and other entities related to the application. The database is accessed via Prisma, which abstracts SQL queries into easier-to-manage queries.

## Authentication and Authorization

Session/Cookie ManagementAuthentication and authorization utilize session cookies that are signed to ensure security. Remix handle cookies using the @remix-run/node library, enabling secure user login and access control mechanisms based on cookies stored in the browser.

## Source Code Structure

The following is a summary of the source code directories and their contents:

|  |  |  |
| --- | --- | --- |
| **Code Directory** | | |
| **Directory** | **Usage** | |
| app | Contains all application source code comprising the React application. | |
| app/assets | Non-code assets, such as logo. | |
| app/auth | Functions to manage cookie-based user authentication. | |
| app/components | Reusable React UI components. | |
| app/db | Code to instantiate a Prisma client for creating database queries. | |
| app/http | Utility functions to generate response bodies and codes in response to http requests. | |
| app/mui | Code to manage theming and provide context related to the Material UI (MUI) UI component library. | |
| app/providers | Wrapper components that provide access to shared dependencies using the React context api. | |
| app/queries | Database queries constructed using the generated Prisma client. | |
| app/routes | Routes have a file naming structure that defines their path in the web application. These files create the views, data fetching actions, and form submission actions associated with these routes. | |
| app/terms | String constants to be used throughout the app. | |
| app/validators | Code to validate user input. | |
| prisma | Contains all code relevant to the Prisma Object Relation Mapper, including database schema definitions, migrations, and seed data. | |
| prisma/migrations | The generated SQL code to run database migrations on the associated PostgreSQL database. | |
| public | Public assets to be served on the web, such as favicon. | |
| *Highlighted rows indicate directories containing source code.* | |

# Executables

There are no executables, as this is a web application. The runtime code is the bundled and minified Javascript compiled from the Typescript source files to the **build** directory.

# Code Architecture

## Frontend

## The front end is built using React, with routing and state management handled using React Router. The app structure includes components that focus on delivering a screen-by-screen workflow for user interactions. The UI is styled with Material-UI (MUI), and there are forms and input fields to collect patient data and responses.

## Backend

## The backend uses Prisma as an ORM to interface with the PostgreSQL database. The schema design supports relationships such as patients, risk assessments, and associated categories or question sets.

## Database

## The PostgreSQL database is designed to capture details about patients, their responses to risk assessment questions, and associated metadata such as categories and answer formats. The schema enforces a normalized structure with entity relations and lookup tables, and migrations are used to maintain database consistency.

A screenshot of a computer screen

Description automatically generated

*An Entity-Relationship-Diagram of the Delirium Assessment Tool Database*

### Database Tables

* **\_prisma\_migrations:** An auto-generated table that tracks the status of Prisma database migrations to keep schemas synced across environments.
* **Account:** Contains the email address and user id of registered users.
* **Password:** The salt and hash of the encrypted passwords of users, and their associated account ids.
* **Question:** The title, content, order, category, and format of questions for the user to answer in the patient questionnaire.
* **QuestionCategory:** Contains all the possible categories of questions. The number of positive responses from different categories determines the risk level of a patient.
* **AnswerFormat:** Indicates the format of the answer to a particular question. This is used to determine the correct UI component to use to present the question to the user.
* **Patient:** Patients’ demographic information and associated user id of the provider conducting their assessment.
* **Gender:** The possible gender options of a patient.
* **PatientResponse:** A patient’s response to a particular question.
* **ResponseOption:** A valid response to a particular question, presented as a radio button option in the UI.
* **RiskAssessment:** The assessed risk level of a particular patient after completing the questionnaire, e.g. “At Risk”, or, “Positive Diagnosis”.
* **RiskType:** The possible risk types of a risk assessment.
* **AgeGroup:** Demographic age groups for tracking results over time, eg. “0-17”, “18-30”, etc.
* **DailyResults:** Risk type results by demographic (gender and age range). Has a compound uniqueness constraint on date, gender, and age range so that counts can be compiled by all three fields for charting purposes.

Programming Language | Typescript

This project is written in Typescript. Typescript is a superset of JavaScript that adds aspects of strongly typed languages, such as types and generics. Typescript is a compiled language that compiles to plain JavaScript for use in NodeJS and web browser runtime environments.

The “ts” filename extension connotes a Typescript file. Additionally, there are files postfixed with a “tsx” filename extension. This is a type of Typescript file that supports JSX. JSX is a format used in React which allows the mixing of JavaScript and XML-style markup, used as shorthand for JavaScript functions that return React UI components.

The “prisma/migrations” folder contains several files written in SQL. These are generated files resulting from code migrations created using the Prisma ORM.

Project Classes

Classes within the project are used to abstract re-usable pieces of code. Classes are also used to group related values, known as properties. The project utilizes these classes:

### Client for Generating Prisma Database Queries | PrismaClient

The Prisma client is a JavaScript class generated from the Prisma database schema. It exposes all the utility methods needed to construct queries in JavaScript or Typescript, which are then executed at runtime as SQL commands to the PostgreSQL database.

Project Modules

Modules are used for procedural based code that does not require state data like class modules do. The application consists of reusable UI components, queries, terms, and validators.

## UI Components

### App Bar | AppAppBar.tsx

The shared top navigation bar for the application.

### App Container | AppContainer.tsx

The styled UI container used by each screen.

### Radar Chart Card | RadarChartCard.tsx

Renders a Radar-style chart contained in a styled card.

### Risk Type Trends Card | RiskTypeTrendsCard.tsx

Renders a stacked bar chart illustrating risk type trends over time.

### Queries

### Account Exists | accountExists.ts

Verifies if a user account exists for a given email address.

### Answer Question | answerQuestion.ts

Saves the response to a question.

### Create Account | createAccount.ts

Creates a new account and saves the credentials.

### Create Patient | createPatient.ts

Creates new patient demographic data.

### Create Risk Assessment | createRiskAssessment.ts

Generates and saves a risk assessment based on patient responses.

### Get Daily Results by Segment | getDailyResultsBySegment.ts

Retrieves risk type results by day, gender, and age group.

### Get Patient by Id | getPatientById.ts

Returns a patient’s demographic data, risk type, and responses by their id.

### Get Question by Order | getQuestionByOrder.ts

Retrieves a question by its order in the questionnaire, based on order url parameter.

### Get Risk Assessment from Responses | getRiskAssessmentFromResponses.ts

Assesses responses to determine the patient’s risk type for developing delirium.

### Get Risk Trends for Last *N* Days | getRiskTrendsForLastNDays.ts

Takes an integer *n*, and returns *n* days of counts by risk type.

### Login | login.ts

Compares a user’s supplied credentials against stored credentials and returns either a redirect on success or an error on failure.

## Terms

### Risk Assessment | riskAssessment.ts

An enumerable of possible risk type values, e.g. “At Risk”, “Positive Diagnosis”.

## Validators

### Login | login.ts

Validates that required login fields are included and that password requirements are met.

### New Patient | newPatient.ts

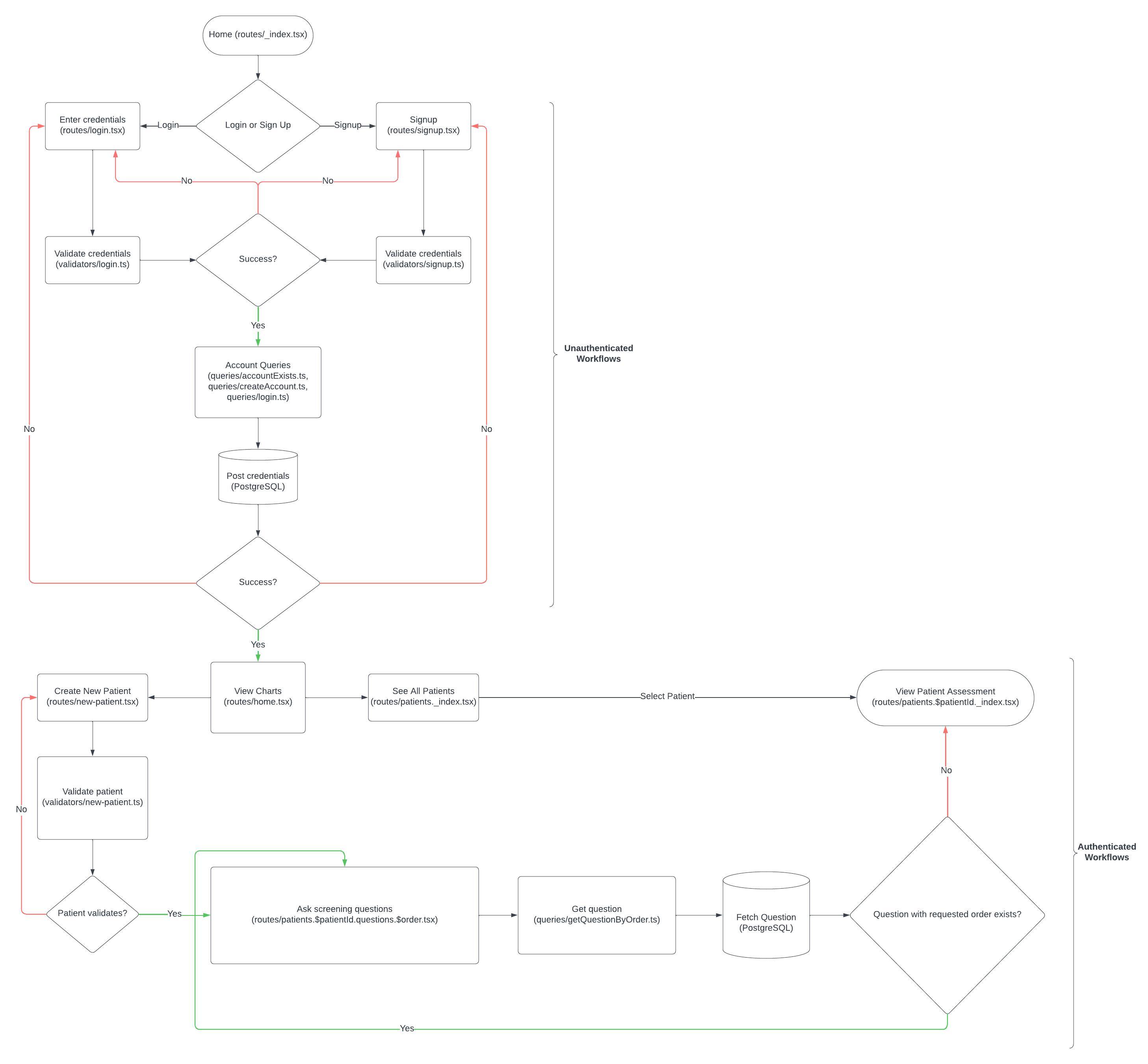
Validates a new patient, including first and last name length, and reasonable age range.

### Signup | validateSignup.ts

Validates the input for signing up a new user, including their email and password. Checks that the email is non-empty, includes an "@" symbol, and ensures the password is non-empty. It also verifies if an account already exists with the provided email.

Program Start and End Flow

The following chart illustrates the user flow, from login or signup through completing a patient screening. After authenticating, the user is presented with a dashboard containing risk trend charts. The user may then enter a new patient and answer screening questions. After completing the screening questions, a summary screen with the patient’s risk assessment is presented. These summaries are also available from the “All Patients” screen.



Summary

The Delirium Assessment Tool is a proof of concept demonstrating how a purpose-built screening tool for evaluating patients’ risk of developing delirium can assist healthcare providers in reducing incidence of delirium and mitigating its harmful effects. As a web application, it is accessible in most environments without the need for additional software. It relies on server-side-rendered React user interfaces, utilizing the full-stack Remix framework to provide security and performance. PostgreSQL was utilized to enforce sound data normalization. In addition to helping providers assess patients, it also provides them with compelling charts to help alert them to risk profile and demographic trends over time. The purpose of this effort is to show that healthcare software can be built in a way to empower providers to make data-driven decisions to ensure better health outcomes in their patient populations.

# APPENDIX A (BUILD AND RELEASE PROCESS)

## Deploying to Fly.io

A fly.toml file and a Dockerfile are included to manage the containerization and deployment of the app to its production environment, hosted on the cloud hosting provider Fly.io. Fly will run a deployment pipeline that installs all dependencies listed in package.json using **npm ci**, and subsequently run the **npm build** and **npm start** commands. Additionally, it will run the **npx prisma migrate deploy** command to execute any pending database schema migrations.

## Deploying Manually

The above process can be run manually at any time using the Fly.io command line tool “FlyCtl”. This tool must be installed to deploy manually (see <https://fly.io/docs/flyctl/install/>). Once it is installed, the app can be deployed using the following command:

1. fly deploy

## Github Actions

This app is also designed to auto-deploy to Fly.io on any pushes to the master branch of its git repository using Github Actions. These deployment steps are outlined in **.github/workflows/fly-deploy.yml**.

# APPENDIX B (CLIENT INSTALLATION INSTRUCTIONS)

## Client Installation Guide

### Environment

This app will run in any runtime environment that supports the “fetch” api. However, NodeJS is required to manage dependencies. To begin, install Node and NPM (see <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>).

### Dependencies

Install dependencies by running the following command:

1. npm ci

### Database Setup

This app uses **PostgreSQL** for its database. To set it up:

1. **Install PostgreSQL**  
   Install PostgreSQL on your server or connect to an existing PostgreSQL instance. (See <https://www.prisma.io/dataguide/postgresql/setting-up-a-local-postgresql-database>).
2. **Create a New Database**  
   Create a new database for the app.
3. **Set Up the .env File**  
   Create a .env file in the project root with the following variable:

1. DATABASE\_URL=postgresql://<username>:<password>@<host>:<port>/<database>

* + Replace <username>, <password>, <host>, <port>, and <database> with your PostgreSQL credentials and database details.
  + **Important**: Keep the .env file private and never commit it to version control.

1. **Apply the Prisma Schema**  
   Run the following commands to apply the Prisma schema and seed the database:

1. npx prisma migrate deploy

2. npx prisma db seed

### Building for Production

To build the app for production:

1. npm run build

This will create the bundled and minified production Javascript files in a “build” folder at the root of the project.

### Running the App in Production

After building, run the app with:

1. npm start

# APPENDIX C (DEVELOPER SETUP INSTRUCTIONS)

## Developer Guide

### Environment

This app requires a NodeJS environment to run locally and manage dependencies. To begin, install Node and NPM (see <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>).

### Dependencies

Install dependencies by running the following command:

1. npm i

### Database Setup

This app uses **PostgreSQL** for its database. To set it up:

1. **Install PostgreSQL**  
   Install PostgreSQL on your local machine or connect to an existing PostgreSQL instance. (See <https://www.prisma.io/dataguide/postgresql/setting-up-a-local-postgresql-database>).
2. **Create a New Database**  
   Create a new database for the app.
3. **Set Up the .env File**  
   Create a .env file in the project root with the following variable:

1. DATABASE\_URL=postgresql://<username>:<password>@<host>:<port>/<database>

* + Replace <username>, <password>, <host>, <port>, and <database> with your PostgreSQL credentials and database details.
  + **Important**: Keep the .env file private and never commit it to version control.

1. **Apply the Prisma Schema**  
   Run the following commands to apply the Prisma schema and seed the database:

1. npx prisma migrate dev

2. npx prisma db seed

### Running in Development Mode

Run the app in development mode:

1. npm run dev

This will hot-reload the app as you make changes and run on port 5173.

### Running Lint Checks

To check for any linting issues, use:

1. npm run lint

### Type Checking

To ensure type safety with TypeScript:

1. npm run typecheck

# APPENDIX D (DOCUMENTATION VIDEOS)

* [Delirium Assessment Tool Walkthrough - Application](https://www.youtube.com/watch?v=VISmz4nOeTM)
* [Delirium Assessment Tool Walkthrough - Database](https://www.youtube.com/watch?v=kmFFe0i79fk)
* [Delirium Assessment Tool Walkthrough – Reusable Code](https://www.youtube.com/watch?v=o7lnN2tAZ-A)
* [Delirium Assessment Tool Walkthrough - Validation](https://www.youtube.com/watch?v=LudYePb1DxM)