



Reports System

By:Jawad Maani

Introduction

The "**Reports System**" is a full-stack **Next.js** application built with **TypeScript**, **React Query**, and **Zod** for type-safe data validation. It demonstrates the development of a complete reporting platform that supports CRUD operations, efficient state management, and robust validation.

The system provides users with the ability to create, edit, delete, and view reports through a modern and responsive interface. By leveraging cutting-edge web technologies, this project highlights best practices in frontend development, scalability, and user experience.

Requirements

1. Technical Requirements

- **Node.js** (v18 or higher)
- **npm**
- **Next.js** framework
- **TypeScript** for type safety
- **React Query** for data fetching and caching
- **Zod** for validation
- **Tailwind CSS + Shadcn/UI** for styling

2. Functional Requirements

- Ability to **create new reports**
- Ability to **edit existing reports**
- Ability to **delete reports**
- Ability to **view a list of reports**
- Ability to **view report details**
- **Map integration** to show report locations
- **Validation** to ensure correct data entry

Architecture

The "Reports System" follows a modular architecture that separates concerns into clear layers:

1. Frontend (Next.js + TypeScript)

- Provides the user interface and routing.
- Uses the **App Router** structure (`src/app`) for pages like:
 - `/reports` → list of reports
 - `/reports/[reportSlug]` → single report details
 - `/create-report` → create new report

2. Components Layer

- Reusable UI components stored in `src/components`.
- Examples:
 - `report-form.tsx` → form for creating/editing reports
 - `report-list.tsx` → list of all reports
 - `report-map.tsx` → map integration
 - `main-header.tsx` → navigation header

3. State Management (React Query)

- Handles fetching, caching, and updating report data.

4. Validation (Zod)

- Ensures all report data is type-safe and valid before submission.
- Prevents invalid or incomplete reports from being saved.

5. Styling (Tailwind CSS + Shadcn/UI)

- Provides a responsive and modern design.
- Ensures accessibility and consistency across components.

Components Overview

1-Route Components in (`src/app`)

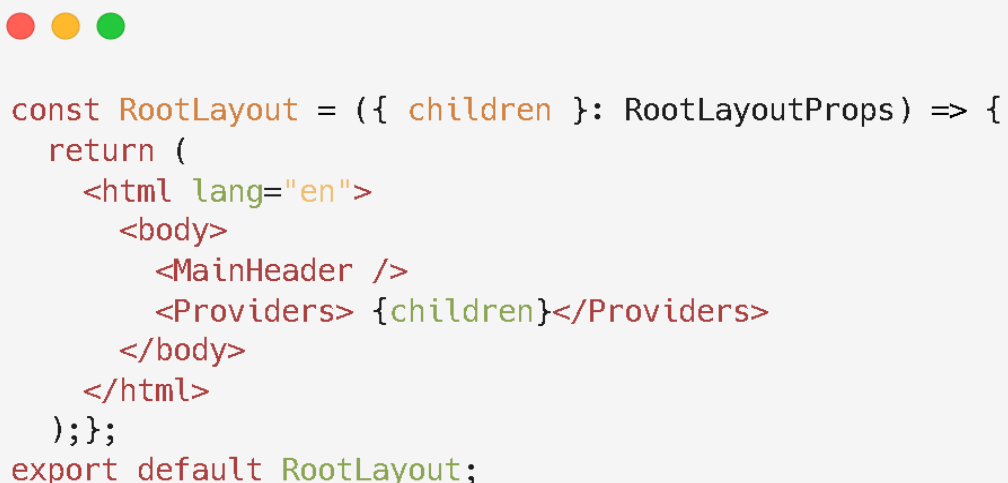
1-RootLayout Component (`app/layout.tsx`)

The RootLayout component defines the global structure of the application. It wraps all pages with a consistent layout and ensures that shared elements are always visible.

In this project, the layout includes:

- **MainHeader**: a navigation header displayed on every page.
- **Providers**: a wrapper that sets up React Query by initializing a `QueryClient` and making it available to all child components. This allows every page to use React Query for data fetching, caching, and synchronization without extra setup.

By using `RootLayout`, the application maintains a unified design and ensures that React Query's global client is accessible across the entire app, enabling efficient and consistent data handling.



```
const RootLayout = ({ children }: RootLayoutProps) => {
  return (
    <html lang="en">
      <body>
        <MainHeader />
        <Providers> {children}</Providers>
      </body>
    </html>
  );
};
export default RootLayout;
```

2-Providers Component (app/providers.tsx)

The Providers component sets up React Query for the whole application. It creates a **QueryClient** and makes it available to all pages and components through the **QueryClientProvider**. This allows every part of the app to use React Query without extra setup.

Key points:

- **Data fetching**: Makes it easy to get data from the server.
- **Caching**: Saves data so the app doesn't need to reload it every time.
- **Consistency**: Keeps one shared source of data for all components.
- **Performance**: Reduces unnecessary requests and makes the app faster.

In short, this component is the bridge between **React Query** and the **app**, ensuring smooth and efficient data handling everywhere.



```
const Providers = ({ children }: { children: ReactNode }) => {  
  const [queryClient] = useState(() => new QueryClient());  
  
  return (  
    <QueryClientProvider client={queryClient}>{children}</QueryClientProvider>  
  );  
};  
  
export default Providers;
```

3-ReportsPageComponent (app/reports/page)

The **ReportsPage** component is responsible for displaying the list of reports. It uses React Query to fetch data.

How it works:

- Data fetching: Uses `useQuery` with the key `["reports"]` to call **`fetchDummyReports()`**.
- Data validation: The fetched data is validated with `reportsSchema` (Zod) before rendering.
- UI rendering: Passes the validated data to the **ReportsGrid** component to display reports in a structured layout.
- Navigation: Includes a button that links to **`/reports/create-report`** for adding a new report.

```
export default function ReportsPage() {
  const { data, isPending, isError, error } = useQuery({
    queryKey: ["reports"],
    queryFn: () => fetchDummyReports(),
  });
  return (
    <div className="max-w-6xl mx-auto px-4 py-8">
      <h1 className="text-2xl font-bold mb-6">Reports</h1>
      <ReportsGrid reports={parseData} />
      <div className="flex justify-end">
        <Button asChild>
          <Link href="/reports/create-report">Create New Report</Link>
        </Button>
      </div>
    </div>
  );
}
```

4-CreateReportPage(/create-report/page)

The **CreateReportPage** component provides the interface for adding a new report. It integrates a form (**ReportForm**) with React Query **mutations** to handle the creation process and update the reports list.

How it works:

- **React Query Mutation:** Uses `useMutation` with the `addReport` function to send new report data.
- **Cache Invalidation:** On success, it calls `queryClient.invalidateQueries(["reports"])` to refresh the reports list automatically.
- **Navigation:** After a successful submission, the user is redirected back to `/reports`.
- **Form Handling:**
 - Captures form data with `FormData`.
 - Parses and validates the input using `parseReportFormData`.
 - Calls `mutate(parsedData)` to trigger the mutation.

```
const CreateReportPage = () => {
  const queryClient = useQueryClient();
  const navigate = useRouter();

  const { mutate, isPending, isError, error } = useMutation({
    mutationFn: addReport,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ["reports"] });
      navigate.push("/reports");
    },
    onError: (error) => {
      console.error("Error adding report:", error);
    },
  });
};
```

5-ReportsDetailsPage (reports/[reportSlug]/page)

The **ReportsDetailsPage** component is a dynamic route that displays the details of a single report. It uses **React Query** to fetch, update, and delete reports, while also providing interactive features like editing and map visualization.

How it works:

- **Dynamic Routing:** The page is accessed through a unique `reportSlug` parameter (e.g., `/reports/123`).
- **Data Fetching:** Uses `useQuery` to fetch the report data by its slug.
- **Data Validation:** The fetched data is validated with `reportSchema` (Zod) before rendering.
- **Delete Functionality:**
 - Uses `useMutation` with `deleteReport`.
 - On success, invalidates the reports list and redirects the user back to `/reports`.
 - Includes a confirmation modal before deletion.
- **Update Functionality:**
 - Integrates with `ReportEditPage` to allow inline editing of the report.
 - Updates the cached report data using `queryClient.setQueryData`.

```
const ReportsDetailsPage = ({ params }: ReportsDetailsPageProps) => {
  const navigate = useRouter();
  const queryClient = useQueryClient();
  const [isDeleting, setIsDeleting] = useState(false);
  const [showMap, setShowMap] = useState(false);

  const { data, isLoading, isError, error } = useQuery({
    queryKey: ["report", params.reportSlug],
    queryFn: () => fetchDummyReports(params.reportSlug),
  });

  const { mutate } = useMutation({
    mutationFn: deleteReport,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ["reports"] });
      navigate.push("/reports");
    },
  });

  const handleUpdate = (updatedReport: Report) => {
    queryClient.setQueryData(["report", params.reportSlug], updatedReport);
  };
};
```


Types and Validation (**src/types**)

The **types** folder defines the data structure and validation rules for reports. It uses **Zod**, a TypeScript-friendly validation library, to ensure that all report data is safe and consistent before being used in the application.

Key elements:

- **reportSchema**
 - Defines the structure of a single report.
 - Fields include:
 - **id**: unique identifier (string).
 - **title**: report title (cannot be empty).
 - **location**: object with **lat** and **lng** coordinates.
 - **importance**: must be one of "low", "medium", or "high".
 - **type**: must be one of "trafficLight", "roadwork", "accident", or "other".
 - **description**: text description (cannot be empty).
 - **createdAt**: date and time in ISO format.
- **reportsSchema**
 - An array of **reportSchema**.
 - Used when validating multiple reports at once (e.g., fetching a list).
- **Report type**
 - A TypeScript type automatically inferred from **reportSchema**.
 - Ensures type safety across the project when working with reports.

Why this is important:

- Guarantees that all report data follows the same structure.
- Prevents invalid or incomplete data from being processed.
- Improves developer experience by combining **runtime validation (Zod)** with **compile-time safety (TypeScript)**.



```
import { z } from "zod";

export const reportSchema = z.object({
  id: z.string(),
  title: z.string().min(1, "Title cannot be empty"),
  location: z.object({
    lat: z.number(),
    lng: z.number(),
  }),
  importance: z.enum(["low", "medium", "high"]),
  type: z.enum(["trafficLight", "roadwork", "accident", "other"]),
  description: z.string().min(1, "Description cannot be empty"),
  createdAt: z.string().datetime(),
});

export const reportsSchema = z.array(reportSchema);

export type Report = z.infer<typeof reportSchema>;
```

parseReportFormData

(utils/parseReportFormData.ts)

The **parseReportFormData** function is a utility that converts raw form input into a validated **Report** object. It ensures that all data submitted through the report form is properly structured and safe before being used in the application.

How it works:

- **Input:**
 - Accepts a **FormData** object (from the report form).
 - Optionally accepts an **existingReport** (used when editing a report).
- **Data Transformation:**
 - Converts the form data into a plain object.
 - Builds a **Report** object with the required fields:
 - **id**: reuses the existing ID if editing, otherwise generates a new one.
 - **createdAt**: keeps the original date if editing, otherwise sets the current timestamp.
 - **title**, **description**, **importance**, **type**: taken directly from the form.
 - **location**: parses latitude and longitude values as numbers.
- **Validation:**
 - Uses **reportSchema.safeParse** (Zod) to validate the constructed object.
 - If validation fails, returns **null**.
 - If validation succeeds, returns the validated **Report**.

Key points:

- Guarantees that **all form submissions** match the required report structure.
- Prevents invalid or incomplete data from being saved.
- Supports both **creating new reports** and **editing existing ones**.
- Combines **data parsing** with **runtime validation** for reliability.

```
import { reportSchema } from "@types/types";
import { Report } from "@types/types";

export const parseReportFormData = (
  formData: FormData,
  existingReport?: Report
): Report | null => {
  const data = Object.fromEntries(formData);

  const reportData: Report = {
    id: existingReport ? existingReport.id : Date.now().toString(),
    createdAt: existingReport
      ? existingReport.createdAt
      : new Date().toISOString(),
    title: data.title as string,
    location: {
      lat: parseFloat(data.lat as string),
      lng: parseFloat(data.lng as string),
    },
    importance: data.importance as "low" | "medium" | "high",
    type: data.type as "trafficLight" | "roadwork" | "accident" | "other",
    description: data.description as string,
  };

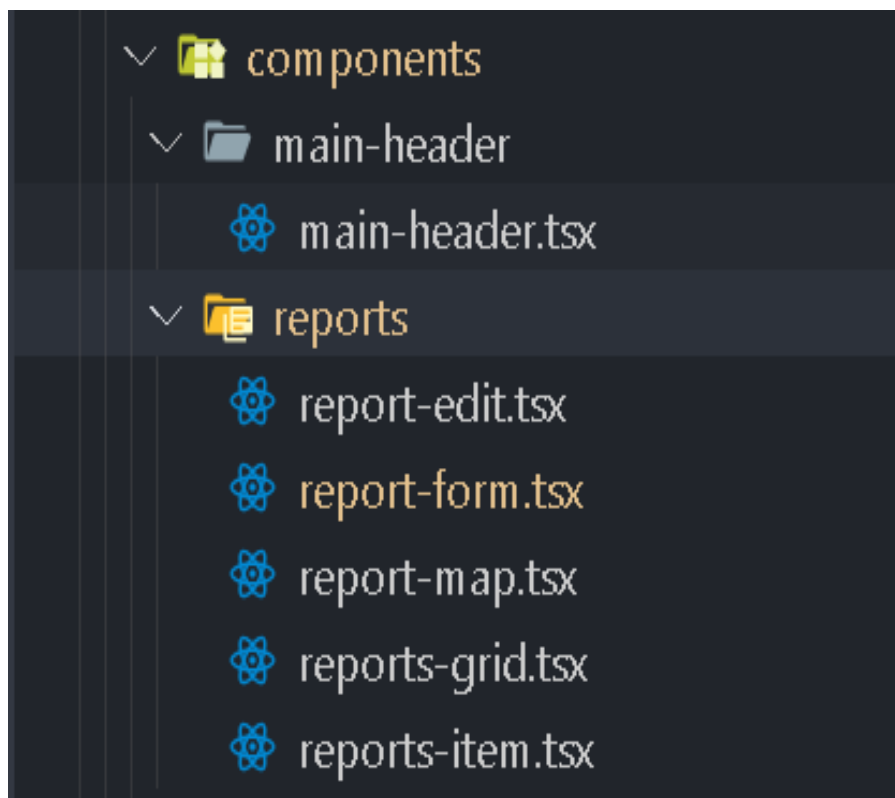
  const parsed = reportSchema.safeParse(reportData);
  if (!parsed.success) return null;
  return parsed.data;
};
```

Components (**src/components**)

The **src/components** folder contains reusable UI components that support the main application pages. These components are relatively simple and straightforward, focusing on presentation and reusability. Since they do not involve complex logic, they are not explained in detail in this report.

For a full view of the components' implementation, you can explore the project repository on GitHub:

[🔗 Reports System – GitHub Repository](#)



THE END