Lab 06: React Router & Navigation

Week 6 Assignment (16.02–22.02.2026)

Student Name: ____________________ Date: _______

Overview

Total Points: 100 (50 points per lab)
Required Reading: *React and React Native, Fifth Edition* by Sakhniuk & Boduch (Packt 2024)
Chapter 7: "Handling Navigation with Routes" (pp. 105–120)
Sections to read: Technical requirements (p. 105), Declaring routes (p. 106), Hello route (p. 106), Decoupling route declarations (p. 108), Handling route parameters (p. 110), Resource IDs in routes (p. 111), Query parameters (p. 114), Using link components (p. 116), Basic linking (p. 117), URL and query parameters (p. 118)

Key Vocabulary

- **createBrowserRouter**: Function that creates a router from an array of route configuration objects. Each route has a `path` and `element` property (Ch. 7, p. 106).
- **RouterProvider**: The top-level component that receives the router created by `createBrowserRouter` and renders the matched route's component (Ch. 7, p. 107).
- **Outlet**: A built-in react-router component placed inside a layout that will be replaced with the matched child route's element (Ch. 7, p. 109).
- **useParams**: Hook that returns an object of key/value pairs of the dynamic URL segments. E.g., for route `/users/:id`, calling `useParams()` returns `{id: "1"}` (Ch. 7, p. 112).
- **useSearchParams**: Hook that returns query string parameters as a `URLSearchParams` object. E.g., for URL `/?order=desc`, you call `search.get("order")` (Ch. 7, p. 115).
- **useLoaderData**: Hook that returns data loaded by a route's `loader` function, which runs before the component renders (Ch. 7, p. 112).
- **Link**: Component that navigates using the client-side router instead of sending a GET request to the server. Use `to` prop instead of `href` (Ch. 7, p. 117).
- **errorElement**: A fallback component rendered when a route's loader throws an error or the route is not found (Ch. 7, p. 111).

1. Lab 6.1: Multi-Page Application with React Router (50 Points)

Objective

Build a multi-page React application using `react-router-dom` v6. Students will learn to declare routes using `createBrowserRouter`, create a shared layout with `Outlet`, and navigate between pages using `Link` components. This aligns with Ch. 7, "Declaring routes" (p. 106) and "Using link components" (p. 116).

Background

Ch. 7 opens with the concept that a router's job is to render content that corresponds to a URL. The `react-router` package is the standard tool for this. Routes are declared as configuration objects, and the `RouterProvider` component connects root rendering to the router. The book emphasizes that each top-level feature of the application can define its own routes so it's clear which routes belong to which feature (p. 108). See **Appendix A** for the book's route declaration example and **Appendix B** for the Layout with `Outlet` pattern.

Problem Statement

Build a simple "Student Portal" with the following pages:

- **Home** (`/`) — welcome message and links to other pages
- **Courses** (`/courses`) — list of courses (hardcoded data is fine)
- **About** (`/about`) — about the university/program
- **Not Found** (`*`) — a "404 Not Found" page for any unmatched URL

All pages share a common navigation bar and footer rendered via a `Layout` component.

Tasks

Task 1: Project Setup and Route Configuration (15 points)

1. Create a new React TypeScript project and install react-router:

```
npm create vite@latest student-portal -- --template react-ts
cd student-portal && npm install
npm install react-router-dom
```

2. In `main.tsx`, declare routes using `createBrowserRouter` (Ch. 7, p. 106):

```
import { createBrowserRouter, RouterProvider }
  from "react-router-dom";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Layout />,
    children: [
      { index: true, element: <Home /> },
      { path: "courses", element: <Courses /> },
      { path: "about", element: <About /> },
      { path: "*", element: <NotFound /> },
    ],
  },
]);
```

```
ReactDOM.createRoot(
  document.getElementById("root")!
).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

3. Reference: Ch. 7, “Hello route” (p. 106), “Decoupling route declarations” (p. 108). See **Appendix A.**

Task 2: Layout with Navigation and Outlet (20 points)

1. Create a `Layout.tsx` component that renders:
   - A `<nav>` bar with `Link` components (not `<a>` tags!) pointing to `/`, `/courses`, and `/about`
   - An `<Outlet />` component where child routes will render (Ch. 7, p. 109)
   - A simple `<footer>`

```
import { Link, Outlet } from "react-router-dom";

function Layout() {
  return (
    <>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/courses">Courses</Link>
        <Link to="/about">About</Link>
      </nav>
      <main>
        <Outlet />  {/* child route renders here */}
      </main>
      <footer>Student Portal 2026</footer>
    </>
  );
}
```

2. **Important**: The book warns against using standard `<a>` elements because they “will try to locate the page on the backend by sending a GET request” instead of handling routes locally (Ch. 7, p. 116). Always use `<Link>`.

3. Add basic CSS styling to your navigation (flexbox or similar). Make the active page’s link visually distinct (optional: use `NavLink` with `className` callback).

4. Reference: Ch. 7, “Basic linking” (p. 117). See **Appendix B**.

Task 3: Page Components and 404 Handler (15 points)

1. Create simple page components in separate files:
   - `Home.tsx` — welcome text, maybe a hero section
   - `Courses.tsx` — a list of 3–5 hardcoded courses (use TypeScript interface for course data)
   - `About.tsx` — descriptive paragraph about the program

  - `NotFound.tsx` — "404 — Page Not Found" message with a link back to Home

2. Verify that navigating to an unknown URL (e.g., `/xyz`) displays the `NotFound` component.
3. All components must use TypeScript (`.tsx` files) with proper types (reuse knowledge from Lab 5).

Deliverable

- Vite + React TS project with: `main.tsx`, `Layout.tsx`, `Home.tsx`, `Courses.tsx`, `About.tsx`, `NotFound.tsx`
- Navigation works without full page reloads (client-side routing)
- 404 page renders for unknown URLs
- Brief README explaining your route structure

Lab 6.1 Assessment Rubric (50 points)

- **Task 1 — Route Configuration (15 pts)**: Vite TS project created (3), `react-router-dom` installed (2), `createBrowserRouter` used correctly (5), all 4 routes declared (5).
- **Task 2 — Layout & Navigation (20 pts)**: `Layout.tsx` with `<nav>` (5), `Link` components used (not `<a>`) (5), `<Outlet />` renders child routes (5), basic styling on nav (5).
- **Task 3 — Pages & 404 (15 pts)**: All 4 page components created (8), TypeScript types used (4), 404 route works for unknown URLs (3).

Grading Notes: App must compile and run. Using `<a>` instead of `<Link>` is **-5 pts**. Missing `.tsx` extension is **-3 pts**.

2. Lab 6.2: Route Parameters & Dynamic Pages (50 Points)

Objective

Extend the application from Lab 6.1 to include dynamic route parameters (`/courses/:id`) and query parameters (`?sort=asc`). Students will use the `useParams`, `useLoaderData`, and `useSearchParams` hooks. This aligns with Ch. 7, "Handling route parameters" (p. 110) and "Query parameters" (p. 114).

Background

Ch. 7 explains that "the `:` syntax marks the beginning of a URL variable" (p. 111). The variable is passed to the component and retrieved with the `useParams()` hook. For data loading, the book introduces `loader` functions that run *before* the component renders, providing data via the `useLoaderData()` hook (p. 111). Query parameters are handled by the `useSearchParams()` hook, which returns a `URLSearchParams` object (p. 115). See **Appendix C** for the book's `useParams` example and **Appendix D** for query parameters.

Problem Statement

Extend the "Student Portal" so that:

- The Courses page lists courses as clickable links
- Clicking a course navigates to `/courses/:id` showing details
- The course list supports sorting via query parameter `?sort=asc|desc`
- Invalid course IDs show a user-friendly error page

Tasks

Task 1: Dynamic Route with useParams (20 points)

1. Define TypeScript interfaces for your data:

```
interface Course {
  id: number;
  title: string;
  instructor: string;
  description: string;
}
```

2. Create a mock data file `data.ts` with at least 4 courses.

3. Add a new route with a dynamic segment:

```
{
  path: "courses/:id",
  element: <CourseDetail />,
  errorElement: <p>Course not found</p>,
  loader: async ({ params }) => {
    const course = getCourseById(
      Number(params.id)
    );
    if (!course)
      throw new Error("Course not found");
    return { course };
  },
}
```

4. In `CourseDetail.tsx`, use `useParams()` and `useLoaderData()` to display the course:

```
import { useParams, useLoaderData }
  from "react-router-dom";

function CourseDetail() {
  const { id } = useParams();
  const { course } = useLoaderData() as {
    course: Course;
  };
  return (
    <div>
      <h2>{course.title}</h2>
      <p>Instructor: {course.instructor}</p>
      <p>{course.description}</p>
      <p>Route ID parameter: {id}</p>
    </div>
  );
}
```

5. Reference: Ch. 7, "Resource IDs in routes" (p. 111). See **Appendix C**.

Task 2: Course List with Links (15 points)

1. Update `Courses.tsx` to render each course as a `Link` to `/courses/{id}`:

```
import { Link } from "react-router-dom";

function Courses() {
  return (
    <ul>
      {courses.map((c) => (
        <li key={c.id}>
          <Link to={`/courses/${c.id}`}>
            {c.title}
          </Link>
        </li>
      ))}
    </ul>
  );
}
```

2. The book shows that "constructing the dynamic segments of a path that is passed to `<Link>` involves string manipulation" using template literals (Ch. 7, p. 118). See **Appendix E**.
3. Verify that clicking a course link navigates to the detail page without a full page reload.
4. Verify that navigating to `/courses/999` (nonexistent) shows the error element.

Task 3: Query Parameters for Sorting (15 points)

1. Add sorting functionality to the Courses page using `useSearchParams` (Ch. 7, p. 114):

```
import { useSearchParams } from "react-router-dom";

function Courses() {
  const [searchParams, setSearchParams] =
    useSearchParams();
  const sortOrder = searchParams.get("sort")
    || "asc";

  const sorted = [...courses].sort((a, b) =>
    sortOrder === "desc"
      ? b.title.localeCompare(a.title)
      : a.title.localeCompare(b.title)
  );

  const toggleSort = () => {
    setSearchParams({
      sort: sortOrder === "asc" ? "desc" : "asc",
    });
  };

  return (
    <div>
      <button onClick={toggleSort}>
        Sort: {sortOrder.toUpperCase()}
      </button>
      <ul>
        {sorted.map((c) => (
          <li key={c.id}>
            <Link to={`/courses/${c.id}`}>
              {c.title}
            </Link>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

2. The book notes: “There is no special setup in the router for handling query parameters. It’s up to the component to handle any query strings provided to it” (Ch. 7, p. 115).
3. Verify: clicking the Sort button toggles the URL between `/courses?sort=asc` and `/courses?sort=desc`, and the list re-orders accordingly.
4. Reference: Ch. 7, “Query parameters” (p. 114). See **Appendix D**.

Deliverable

- Extended project from Lab 6.1 with: `data.ts`, `CourseDetail.tsx`, updated `Courses.tsx`, updated routes in `main.tsx`
- Dynamic route `/courses/:id` works correctly
- `?sort=asc|desc` query parameter works
- Invalid IDs show error element
- README explaining: what `useParams` vs `useSearchParams` is, when to use URL params vs query params

Lab 6.2 Assessment Rubric (50 points)

- **Task 1 — Dynamic Route (20 pts)**: TypeScript `Course` interface (3), mock data file (3), route with `:id` parameter (4), `loader` function (4), `useParams` + `useLoaderData` used correctly (3), error handling for invalid ID (3).
- **Task 2 — Course Links (15 pts)**: Courses rendered as `Link` list (6), template literal for dynamic `to` prop (4), navigation works without reload (3), error element for nonexistent course (2).
- **Task 3 — Query Parameters (15 pts)**: `useSearchParams` used (5), sort toggle button works (4), URL updates with query string (3), list re-orders visually (3).

Grading Notes: App must run. No `any`. All files `.tsx`. Using `window.location` instead of router hooks is **-10 pts**.

Submission Requirements and Regulations

OquLabs Protocol (30 points)

- **Full Screen Mode (10 points)**: Maintain full screen throughout. Violations: **-20 points**.
- **Active Typing (10 points)**: Code must be typed. Copy-paste detection: **-30 points**. Boilerplate templates allowed; logic must be typed.
- **Session Duration (10 points)**: Minimum 30 minutes. Under 30 min: **-10 points** and may count as absence.

Git Discipline (10 points)

- **Folder Structure (5 points)**: Use `Lab_06/task1/`, `Lab_06/task2/`. Flat dumps: **-10 points**.
- **Conventional Commits (5 points)**: Use `feat:`, `fix:`, `refactor:`. Bad messages: **-5 points**.
- **Incremental History**: At least 3 meaningful commits. Single dump: **-5 points**.

Code Quality Standards (20 points)

- **TypeScript Strict Mode (10 points)**: Ensure `"strict": true` in `tsconfig.json`. Missing: **-5 points**.
- **Zero `any` (10 points)**: Avoid the `any` keyword. Each `any`: **-5 points**.

AI Usage Policy

If you use AI tools:

- Submit `AI_REPORT` with: tool used, prompts, how you modified/verified code, what you learned.
- Missing report when AI used: **-100 points**. Incomplete report: **-20 points**.

Submission Instructions

1. **Lab 6.1**: Multi-page app (`Layout.tsx`, page components, route config) + README.
2. **Lab 6.2**: Extended app (`CourseDetail.tsx`, `data.ts`, updated routes) + README.
3. Use `Lab_06/task1/`, `Lab_06/task2/` structure.
4. Include name, student ID, date. Submit via course portal. Ensure OquLabs logs are available.

Comprehensive Assessment Summary

Category	Points	Assessment
Lab 6.1 Tasks	50	See Lab 6.1 Rubric
Lab 6.2 Tasks	50	See Lab 6.2 Rubric
Code compiles (zero TS errors)	20	[YES] / [NO] (-20)
TypeScript strict mode	10	[YES] / [NO] (-5)
Zero `any` usage	10	[YES] / [NO] (-5 per any)
OquLabs: Full screen	10	[YES] / [NO] (-20)
OquLabs: No copy-paste	10	[YES] / [NO] (-30)
OquLabs: Duration $>$ 30 min	10	[YES] / [NO] (-10)
Git: Folder structure	5	[YES] / [NO] (-10)
Git: Conventional commits	5	[YES] / [NO] (-5)
TOTAL	**100**	

A. Declaring Routes — createBrowserRouter (Ch. 7, p. 106–107)

The book introduces routing by creating a router with `createBrowserRouter`, passing an array of route objects. Each route has a `path` and `element` property.

Basic route declaration (p. 106)

```
import React from "react";
import ReactDOM from "react-dom/client";
import {
  createBrowserRouter,
  RouterProvider,
} from "react-router-dom";

function MyComponent() {
  return <h1>My Component</h1>;
}

const router = createBrowserRouter([
  {
    path: "/",
    element: <MyComponent />,
  },
]);

ReactDOM.createRoot(
  document.getElementById("root")!
).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Key concept (p. 107): “When the `path` property is matched against the active URL, the component is rendered... the router checks the current URL and returns the corresponding component from the `createBrowserRouter` declaration.”

B. Layout with Outlet and Nested Routes (Ch. 7, p. 108–109)

The book shows how to use a `Layout` component with an `<Outlet />` to render child routes within a shared template.

Layout with Outlet (p. 108–109)

```
import { Link, Outlet } from "react-router-dom";

function Layout() {
  return (
    <main>
      <nav>
        <Link to="/">Main</Link>
        <span> | </span>
        <Link to="/one">One</Link>
        <span> | </span>
        <Link to="/two">Two</Link>
      </nav>
      <Outlet />   {/* matched child renders here */}
    </main>
  );
}

const router = createBrowserRouter([
  {
    path: "/",
    element: <Layout />,
    children: [
      { index: true,
        element: <h1>Nesting Routes</h1> },
      routeOne,   // imported from feature module
      routeTwo,
    ],
  },
]);

export const App = () =>
  <RouterProvider router={router} />;
```

Key concept (p. 109): "`<Outlet />` is a built-in react-router component that will be replaced with matched route elements." The router only gets as big as the number of application features, not the number of routes.

C. useParams and Route Loaders (Ch. 7, p. 111–113)

The book demonstrates fetching a resource by its URL segment ID using a `loader` function and `useParams()`/`useLoaderData()` hooks.

Dynamic route with :id parameter (p. 111–112)

```
export type User = {
  first: string;
  last: string;
  age: number;
};

const router = createBrowserRouter([
  {
    path: "/",
    element: <UsersContainer />,
  },
  {
    path: "/users/:id",
    element: <UserContainer />,
    errorElement: <p>User not found</p>,
    loader: async ({ params }) => {
      const user = await fetchUser(
        Number(params.id)
      );
      return { user };
    },
  },
]);
```

Reading params in the component (p. 112)

```
import { useParams, useLoaderData }
  from "react-router-dom";

function UserContainer() {
  const params = useParams();
  const { user } = useLoaderData() as {
    user: User;
  };

  return (
    <div>
      User ID: {params.id}
      <UserData user={user} />
    </div>
  );
}
```

Key concept (p. 111): “The `:` syntax marks the beginning of a URL variable. The `id` variable will be passed to the `UserContainer` component.” The `loader` function is triggered before the component renders, fetching data asynchronously.

D. Query Parameters with useSearchParams (Ch. 7, p. 114–115)

The book shows how to use query strings for sorting. The router does not explicitly declare query parameters; the component reads them with `useSearchParams()`.

Sorting with query parameters (p. 115)

```
import { useState, useEffect } from "react";
import { useSearchParams }
  from "react-router-dom";

export type SortOrder = "asc" | "desc";

function UsersContainer() {
  const [users, setUsers] =
    useState<string[]>([]);
  const [search] = useSearchParams();

  useEffect(() => {
    const order =
      search.get("order") as SortOrder;
    fetchUsers(order).then((users) => {
      setUsers(users);
    });
  }, [search]);

  return <Users users={users} />;
}
```

Key concept (p. 115): "There is no special setup in the router for handling query parameters. It's up to the component to handle any query strings provided to it." Use `search.get("key")` to read parameters.

E. Dynamic Link Construction (Ch. 7, p. 118–119)

The book shows how to build dynamic links using template literals for URL parameters and `URLSearchParams` for query strings.

URL parameter link vs query parameter link (p. 118–119)

```
import { Link } from "react-router-dom";

const param = "From Param";
const query = new URLSearchParams({
  msg: "From Query",
});

export default function App() {
  return (
    <section>
      <p>
        <Link to={`echo/${param}`}>
          Echo param
        </Link>
      </p>
      <p>
        <Link to={`echo?${query.toString()}`}>
          Echo query
        </Link>
      </p>
    </section>
  );
}
```

Reading both param and query in one component (p. 119)

```
import { useParams, useSearchParams }
  from "react-router-dom";

function Echo() {
  const params = useParams();
  const [searchParams] = useSearchParams();
  return (
    <h1>
      {params.msg || searchParams.get("msg")}
    </h1>
  );
}
```

Key concept (p. 118): "Constructing the dynamic segments of a path that is passed to `<Link>` involves string manipulation. Everything that's part of the path goes to the `to` property."