**Lab: Create a React Quiz App**

1. **Set up questions.json and json-server**

Let’s extend the lab to include **using json-server** to simulate a real backend for your quiz questions, and explain how to **use questions.json from the public folder** as a fallback.

**Install json-server**

npm install --save-dev json-server

**Load Quiz Questions Using json-server or Public Folder**

We will:

* Create a questions.json file containing quiz data
* Place it in the public/ folder for easy access via fetch
* Optionally use json-server to simulate a backend API (localhost:5000/questions)

**File: public/questions.json**

Create the file:

[

{

"id": 1,

"question": "What hook is used to handle state in a functional component?",

"options": ["useState", "useEffect", "useReducer", "useContext"],

"answer": "useState"

},

{

"id": 2,

"question": "Which package is used for routing in React?",

"options": ["react-dom", "react-router-dom", "redux", "react-navigation"],

"answer": "react-router-dom"

}

// Add more questions here...

]

**Call with json-server public/questions.json**

**Load from public/questions.json. In your QuizPage.jsx coming up, you will have:**

useEffect(() => {

fetch("/questions.json")

.then((res) => res.json())

.then((data) => setQuestions(data))

.catch((error) => console.error("Error loading questions:", error));

}, []);

The browser will load /questions.json from the public folder at build time.

1. **Make a Home Page for Your React Quiz App**

The page will include a heading, a brief description, and a **Start Quiz** button that navigates to the quiz itself.

**Instructions:**

1. **Create a new file named** Home.jsx inside your src/pages directory.
2. **Set up the component structure** as a functional component using React.
3. **Use the useNavigate() hook** from react-router-dom to programmatically navigate to the quiz route when the button is clicked.
4. **Display instructional text** that tells the user what to expect when they begin the quiz.

The following code -

* Imports useNavigate from react-router-dom, which allows us to navigate to different routes programmatically.
* Creates a simple functional component named Home.
* Displays a heading and a short instructional paragraph.
* Includes a button that takes the user to /quiz when clicked.

**Example Code:**

import React from "react";

import { useNavigate } from "react-router-dom";

const Home = () => {

const navigate = useNavigate();

return (

<div>

<h1>React Quiz Instructions</h1>

<p>Answer all questions and see your score at the end.</p>

<button onClick={() => navigate("/quiz")}>Start Quiz</button>

</div>

);

};

export default Home;

**Check Your Understanding:**

* What is useNavigate()?

**Answers**

**Q1: What does the useNavigate() hook replace from React Router v5?**

In React Router v5, navigation was handled using the useHistory() hook and calling history.push('/quiz').  
In React Router v6, useHistory() was replaced by useNavigate(), and you call it like this:

**2. Build the Main Quiz Page**

Build the **Quiz Page** for your React Quiz Application. This component will:

* Load questions from a JSON file
* Display one question at a time
* Let the user select an answer
* Track their score
* Navigate to the results page when finished

**Instructions:**

1. **Create a new file** called QuizPage.jsx inside the src/pages folder.
2. **Import the following:**  
     
   import React, { useState, useEffect } from “react";

import { useNavigate } from "react-router-dom";

1. **Set up your component with this structure:**
   * questions: state to store quiz questions.
   * currentQuestionIndex: tracks which question the user is on.
   * selectedAnswer: tracks the user’s current choice.
   * score: tracks the user’s total correct answers.
   * useNavigate: to move to the results screen at the end.
2. **Fetch the quiz questions from a file called questions.json:**
   * Use fetch() inside a useEffect() to load them once on component mount.
3. **Display the current question and its answer choices.**
   * Each option is a button.
   * Once a user selects an option, disable the rest until they click **Next**.
4. **Evaluate their selection and update score if correct.**
   * After the last question, navigate to the /results page with the final score.

**Example Code:**

import React, { useState, useEffect } from "react";

import { useNavigate } from "react-router-dom";

const QuizPage = () => {

const [questions, setQuestions] = useState([]);

const [currentQuestionIndex, setCurrentQuestionIndex] = useState(0);

const [selectedAnswer, setSelectedAnswer] = useState(null);

const [score, setScore] = useState(0);

const navigate = useNavigate();

useEffect(() => {

fetch("/questions.json")

.then((res) => {

if (!res.ok) throw new Error(`HTTP error! Status: ${res.status}`);

return res.json();

})

.then((data) => setQuestions(data))

.catch((error) => console.error("Error loading questions:", error));

}, []);

if (!questions.length) return <p>Loading...</p>;

const handleAnswer = (answer) => {

setSelectedAnswer(answer);

};

const nextQuestion = () => {

if (selectedAnswer === questions[currentQuestionIndex].answer) {

setScore(score + 1);

}

setSelectedAnswer(null);

if (currentQuestionIndex + 1 < questions.length) {

setCurrentQuestionIndex(currentQuestionIndex + 1);

} else {

navigate("/results", { state: { score, total: questions.length } });

}

};

return (

<div>

<h2>Question {currentQuestionIndex + 1}</h2>

<p>{questions[currentQuestionIndex].question}</p>

{questions[currentQuestionIndex].options.map((option) => (

<button

key={option}

onClick={() => handleAnswer(option)}

disabled={selectedAnswer !== null}

>

{option}

</button>

))}

<br />

<button onClick={nextQuestion} disabled={!selectedAnswer}>

Next

</button>

</div>

);

};

export default QuizPage;

**Check Your Understanding:**

**1. What would happen if you didn't reset selectedAnswer to null in nextQuestion()?**  
*Hint: Think about what controls whether the answer buttons are disabled.*

**2. How does the app know when to end the quiz and navigate to the results page?**

**3. How would you prevent the user from skipping questions without answering?**

**Answers:**

**What would happen if you didn't reset selectedAnswer to null in nextQuestion()?**

If you don’t reset selectedAnswer to null, the answer buttons in the next question will **remain disabled**, because of this line in the component:

disabled={selectedAnswer !== null}

That logic disables buttons once an answer is selected. If selectedAnswer stays set, the next question will load but the user won't be able to select a new answer — effectively freezing the quiz.

**How does the app know when to end the quiz and navigate to the results page?**

The app checks whether the user is on the last question using this condition:

if (currentQuestionIndex + 1 < questions.length) {

setCurrentQuestionIndex(currentQuestionIndex + 1);

} else {

navigate("/results", { state: { score, total: questions.length } });

}

If the **next question index** would be **equal to or greater than the total number of questions**, the app knows the quiz is finished and uses navigate() to go to the results page with the user's score and total questions.

**How would you prevent the user from skipping questions without answering?**

You already have this line:

<button onClick={nextQuestion} disabled={!selectedAnswer}>

This disables the **Next** button unless the user has selected an answer (selectedAnswer is not null).

**3. Create a Results Page for the React Quiz App**

Create a simple **Results Page** that shows the user's quiz score and provides a button to restart the quiz. You will use React Router to receive data passed from the quiz and navigate back to the home screen.

**Instructions:**

1. **Create a file named** ResultsPage.jsx inside your src/pages folder.
2. **Import the following React Router hooks:**  
     
   import { useLocation, useNavigate } from "react-router-dom";
3. **Use useLocation()** to access the score and total number of questions that were passed from the quiz component.
4. **Use useNavigate()** to allow the user to click a **Try Again** button that returns them to the home page.

**Example Code:**

import React from "react";

import { useLocation, useNavigate } from "react-router-dom";

const ResultsPage = () => {

const location = useLocation();

const navigate = useNavigate();

const { score, total } = location.state || { score: 0, total: 0 };

return (

<div>

<h1>Quiz Completed!</h1>

<p>Your Score: {score} / {total}</p>

<button onClick={() => navigate("/")}>Try Again</button>

</div>

);

};

export default ResultsPage;

**What This Code Does:**

* useLocation() reads the state object passed from the quiz.
* score and total are safely pulled from location.state, or default to 0 if no state is found.
* A button allows the user to navigate back to the home page to take the quiz again.

**Check Your Understanding:**

**1. What happens if the user refreshes the Results page? Why might score and total show as 0?**

**2. What would you need to do if you wanted to show a custom message based on how well the user scored (e.g., “Great job!” or “Try again!”)?**

**Answers:**

**What happens if the user refreshes the Results page?**

The state passed via navigate("/results", { state }) is stored **in memory**, not in the URL or local storage. On page refresh, the memory is lost, so location.state becomes undefined, and the page shows score: 0 / 0.

**How would you show a custom message based on the score?**

You can add conditional rendering like:

{score / total >= 0.8 ? <p>Great job!</p> : <p>Try again!</p>}

Place this below the score display to give feedback based on performance.

**(Optional: Unit Tests)** Here are **unit tests** for each of your three pages (Home, QuizPage, ResultsPage) using **Vitest** and **React Testing Library**.

**Setup:**

npm install --save-dev vitest @testing-library/react @testing-library/jest-dom jsdom

Also, add this to your vite.config.js if not already present:

test: {

environment: 'jsdom',

},

**1. Home.jsx – Home Page Tests**

**Home.test.jsx**

import { render, screen, fireEvent } from '@testing-library/react';

import { BrowserRouter } from 'react-router-dom';

import Home from './Home';

import { vi } from 'vitest';

vi.mock('react-router-dom', async () => {

const actual = await vi.importActual('react-router-dom');

return {

...actual,

useNavigate: () => vi.fn(),

};

});

test('renders title and instructions', () => {

render(

<BrowserRouter>

<Home />

</BrowserRouter>

);

expect(screen.getByText(/React Quiz Instructions/i)).toBeInTheDocument();

expect(screen.getByText(/Answer all questions/i)).toBeInTheDocument();

expect(screen.getByRole('button', { name: /Start Quiz/i })).toBeInTheDocument();

});

**2. QuizPage.jsx – Quiz Page Tests**

**QuizPage.test.jsx**

import { render, screen, fireEvent, waitFor } from '@testing-library/react';

import QuizPage from './QuizPage';

import { vi } from 'vitest';

import { BrowserRouter } from 'react-router-dom';

// Mock fetch and useNavigate

global.fetch = vi.fn(() =>

Promise.resolve({

ok: true,

json: () => Promise.resolve([

{

id: 1,

question: 'What is React?',

options: ['Library', 'Language'],

answer: 'Library',

},

]),

})

);

vi.mock('react-router-dom', async () => {

const actual = await vi.importActual('react-router-dom');

return {

...actual,

useNavigate: () => vi.fn(),

};

});

test('loads and displays a question', async () => {

render(

<BrowserRouter>

<QuizPage />

</BrowserRouter>

);

expect(screen.getByText(/Loading.../i)).toBeInTheDocument();

await waitFor(() =>

expect(screen.getByText(/What is React\?/i)).toBeInTheDocument()

);

expect(screen.getByRole('button', { name: 'Library' })).toBeInTheDocument();

expect(screen.getByRole('button', { name: 'Language' })).toBeInTheDocument();

});

test('selects an answer and enables Next button', async () => {

render(

<BrowserRouter>

<QuizPage />

</BrowserRouter>

);

await waitFor(() => screen.getByText(/What is React/));

fireEvent.click(screen.getByText('Library'));

const nextButton = screen.getByRole('button', { name: /Next/i });

expect(nextButton).toBeEnabled();

});

**3. ResultsPage.jsx – Results Page Tests**

**ResultsPage.test.jsx**

import { render, screen, fireEvent } from '@testing-library/react';

import ResultsPage from './ResultsPage';

import { vi } from 'vitest';

import { MemoryRouter } from 'react-router-dom';

vi.mock('react-router-dom', async () => {

const actual = await vi.importActual('react-router-dom');

return {

...actual,

useLocation: () => ({

state: { score: 3, total: 5 },

}),

useNavigate: () => vi.fn(),

};

});

test('shows final score and Try Again button', () => {

render(

<MemoryRouter>

<ResultsPage />

</MemoryRouter>

);

expect(screen.getByText(/Quiz Completed/i)).toBeInTheDocument();

expect(screen.getByText(/Your Score: 3 \/ 5/i)).toBeInTheDocument();

expect(screen.getByRole('button', { name: /Try Again/i })).toBeInTheDocument();

});

**Test Coverage:**

|  |  |
| --- | --- |
| **Page** | **Test** |
| Home | Title, instructions, button |
| QuizPage | Fetch, render, select, next |
| ResultsPage | Shows score, try again button |

**Here's how Vitest works with each part:**

|  |  |
| --- | --- |
| **Feature** | **Explanation** |
| vitest | Core test runner |
| @testing-library/react | For rendering components & simulating DOM interaction |
| @testing-library/jest-dom | For matchers like toBeInTheDocument() |
| vi.mock() | Mocks react-router-dom and fetch in tests |
| jsdom | Enables browser-like environment in Vitest |

**Additional Setup:**

**2. Update vite.config.js:**

/// vite.config.js

import { defineConfig } from 'vite';

import react from '@vitejs/plugin-react';

export default defineConfig({

plugins: [react()],

test: {

environment: 'jsdom',

globals: true,

setupFiles: './setupTests.js', // optional

},

});

**3. Add a file, setupTests.js (optional but recommended):**

// setupTests.js

import '@testing-library/jest-dom';

**Run Tests with Vitest**

To run all tests:

npx vitest

To run in watch mode:

npx vitest --watch

**Optional:**

Here’s how to **add code coverage reporting** using **Vitest**, with full setup and explanation.

**What is Coverage?**

Coverage tells you **how much of your code is tested**. It shows:

* Which **functions**, **branches**, and **lines** were executed during tests
* Which parts of the code were **missed** (not covered)

**Enable Coverage in Vitest**

**1. Install Coverage Tool**

Vitest uses **c8** under the hood for coverage. (If you're already using Vitest, just install c8.)

npm install --save-dev c8

**2. Update vite.config.js**

Enable coverage reporting in your test config:

// vite.config.js

import { defineConfig } from 'vite';

import react from '@vitejs/plugin-react';

export default defineConfig({

plugins: [react()],

test: {

environment: 'jsdom',

globals: true,

coverage: {

provider: 'c8', // or 'v8' (but c8 is more stable)

reporter: ['text', 'html'], // add 'lcov' for CI tools

reportsDirectory: './coverage',

},

},

});

**3. Run Tests with Coverage**

npx vitest run --coverage

This will:

* Run all your tests
* Generate a coverage report in the coverage/ folder
* Print a summary in the terminal
* Create an index.html file for detailed viewing

**4. View HTML Coverage Report**

After running tests with coverage, open the report:

open coverage/index.html # Mac

xdg-open coverage/index.html # Linux

start coverage/index.html # Windows

You’ll see color-coded files:

* ✅ **Green** = covered lines
* ⚠️ **Yellow** = partially covered branches
* ❌ **Red** = not tested

Want to **exclude test files** from coverage?  
Add this in vite.config.js:  
  
coverage: {

exclude: ['\*\*/test/\*\*', '\*\*/\*.test.jsx'],

}

**What is Test Coverage?**

**Test coverage** is a metric that tells you **how much of your source code is exercised by your tests**.

It answers questions like:

* Are we testing **every line** of this function?
* Are we covering **every condition** in this if statement?
* Are we ignoring some parts of the app completely?

**Types of Coverage**

|  |  |  |
| --- | --- | --- |
| **Type** | **What it measures** | **Example** |
| **Line coverage** | How many **lines of code** ran during tests | Did return x + y; run? |
| **Function coverage** | Which **functions** were executed | Did calculateTotal() get called? |
| **Branch coverage** | Whether all **branches** of a conditional ran | Did both if and else blocks run? |
| **Statement coverage** | How many **statements** ran | Did every instruction get evaluated? |

**Why Use Coverage**

* You know whether your tests are truly exercising your app.
* Uncovered code is more likely to break silently.
* Find dead code that’s never run.
* Enforce minimum coverage in CI to prevent untested code from being merged.

**Code Coverage Example**

function isAdult(age) {

if (age >= 18) {

return true;

} else {

return false;

}

}

**If your test only does:**

expect(isAdult(20)).toBe(true);

* Line coverage: 100%
* Branch coverage: 50% (you didn’t test the else)

expect(isAdult(10)).toBe(false);

Now branch coverage is 100%.

**How To See Coverage**

After running tests with --coverage, you’ll see a terminal summary:

File | % Lines | % Branches | % Functions | % Statements

--------------------|---------|------------|-------------|--------------

src/utils.js | 100 | 100 | 100 | 100

src/app.js | 80 | 75 | 90 | 85

And a visual report in coverage/index.html.

In summary, the test for:

**Home Page: Home.jsx Had Test file: Home.test.jsx that:**

* Checks that the **title, instructions, and "Start Quiz" button** appear
* Mocks useNavigate

**Quiz Page: QuizPage.jsx Uses Test file: QuizPage.test.jsx**

* Mocks fetch() to load questions
* Mocks useNavigate()
* Confirms:
  + Questions are fetched and displayed
  + Options render
  + Selecting an answer enables the **Next** button

**Results Page: ResultsPage.jsx Uses Test file: ResultsPage.test.jsx**

* Mocks useLocation() with score/total
* Checks that the **score and "Try Again" button** appear

**Note - What is vi.importActual() in Vitest?**

vi.importActual() is a **Vitest utility** that lets you:

Import the **real (non-mocked)** version of a module — even when you're mocking part of it.

**Use Case: Partial Mocking**

Sometimes you want to mock **just one part** of a module (like useNavigate) while keeping the rest real. That’s where vi.importActual() helps.

**Example from our test:**

vi.mock('react-router-dom', async () => {

const actual = await vi.importActual('react-router-dom');

return {

...actual, // keep all the real exports

useNavigate: () => vi.fn(), // override just this one

};

});

Here’s what it does in the test:

* Loads the **real react-router-dom**
* Spreads in all of its exports
* Replaces just useNavigate() with a mock

Without vi.importActual(), you'd have to mock the entire module manually — which is error-prone and unnecessary if you only want to mock one function.

**Why not just do vi.fn() alone?**

Because vi.mock('react-router-dom', () => ({ useNavigate: vi.fn() })) would:

* Erase everything else (BrowserRouter, useLocation, etc.)
* Break your app unless you re-create all exports manually

Let me know if you want:

* A test where useNavigate tracks **what page was navigated to**
* A helper function for mocking React Router cleanly across tests

**Add Rich Features to the Quiz Page**

We will implement a fully interactive quiz page in React with the following features:

* Timed questions
* A visual progress bar
* Disabled answer buttons once selected
* Score tracking and result navigation
* Questions fetched from an external JSON file
* Styling for active answer and progress

**Features We Will Implement:**

|  |  |
| --- | --- |
| **Feature** | **Implemented?** |
| Load questions via fetch() |  |
| useState, useEffect, and useNavigate |  |
| **Timer (60 seconds/question)** |  |
| **Progress bar** |  |
| **Answer button highlighting** |  |
| **Next button disabled until answer is selected** |  |

**Objectives:**

1. In the file: src/pages/QuizPage.jsx
2. Add the component and use useState and useEffect to manage:
   * Questions
   * Timer
   * Selected answers
   * Score
   * Progress
3. Display the current question and answers
4. On answer selection:
   * Highlight the selected answer
   * Disable all answer buttons
5. Add a countdown timer: if it hits 0, auto-advance
6. Add a visual progress bar using inline styles
7. On quiz completion, navigate to /results

**Final Code with Features: QuizPage.jsx**

import React, { useState, useEffect } from "react";

import { useNavigate } from "react-router-dom";

const QuizPage = () => {

const [questions, setQuestions] = useState([]);

const [currentQuestionIndex, setCurrentQuestionIndex] = useState(0);

const [selectedAnswer, setSelectedAnswer] = useState(null);

const [score, setScore] = useState(0);

const [timeLeft, setTimeLeft] = useState(60);

const navigate = useNavigate();

// Shuffle questions

const shuffleArray = (array) => {

return array.sort(() => Math.random() - 0.5);

};

useEffect(() => {

fetch("/questions.json")

.then((res) => {

if (!res.ok) throw new Error(`HTTP error! Status: ${res.status}`);

return res.json();

})

.then((data) => setQuestions(shuffleArray(data)))

.catch((error) => console.error("Error loading questions:", error));

}, []);

useEffect(() => {

if (timeLeft === 0) {

nextQuestion();

return;

}

const timer = setTimeout(() => setTimeLeft(timeLeft - 1), 1000);

return () => clearTimeout(timer);

}, [timeLeft]);

if (!questions.length) return <p>Loading...</p>;

const handleAnswer = (answer) => {

setSelectedAnswer(answer);

};

const nextQuestion = () => {

if (selectedAnswer === questions[currentQuestionIndex].answer) {

setScore(score + 1);

}

setSelectedAnswer(null);

setTimeLeft(60); // Reset timer

if (currentQuestionIndex + 1 < questions.length) {

setCurrentQuestionIndex(currentQuestionIndex + 1);

} else {

navigate("/results", { state: { score, total: questions.length } });

}

};

const progress = ((currentQuestionIndex + 1) / questions.length) \* 100;

return (

<div style={{ padding: "20px", maxWidth: "600px", margin: "auto" }}>

{/\* Progress Bar \*/}

<div style={{ width: "100%", backgroundColor: "#ccc", height: "10px", marginBottom: "10px" }}>

<div style={{ width: `${progress}%`, backgroundColor: "#4caf50", height: "10px" }}></div>

</div>

{/\* Timer \*/}

<h3>Time Left: {timeLeft}s</h3>

<h2>Question {currentQuestionIndex + 1}</h2>

<p>{questions[currentQuestionIndex].question}</p>

{questions[currentQuestionIndex].options.map((option) => (

<button

key={option}

onClick={() => handleAnswer(option)}

disabled={selectedAnswer !== null}

style={{

backgroundColor: selectedAnswer === option ? "#d3d3d3" : "",

display: "block",

width: "100%",

margin: "5px 0",

padding: "10px",

}}

>

{option}

</button>

))}

<br />

<button onClick={nextQuestion} disabled={!selectedAnswer} style={{ marginTop: "10px" }}>

Next

</button>

</div>

);

};

export default QuizPage;

**Set Up Routing for Your Quiz App with App.jsx**

**Objective:**

In this lab, you will configure the main App.jsx file of your React application to use **React Router**. This will allow navigation between the **Home**, **Quiz**, and **Results** pages.

**Remember:**

* How to use react-router-dom
* How to define multiple Routes inside a Router
* How to connect different pages/components together

**Instructions for File: src/App.jsx**

1. **Import Required Modules**

Import React and routing components from react-router-dom:  
  
import React from "react";

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

1. **Import Your Pages**

These are your three main views:  
  
import Home from "./pages/Home";

import QuizPage from "./pages/QuizPage";

import ResultsPage from "./pages/ResultsPage";

1. **Set Up the Router**
   * Wrap your <Routes> in <Router> to enable routing
   * Define routes for /, /quiz, and /results

**Final Code: App.jsx**

import React from "react";

import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

import Home from "./pages/Home";

import QuizPage from "./pages/QuizPage";

import ResultsPage from "./pages/ResultsPage";

import './App.css';

function App() {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/quiz" element={<QuizPage />} />

<Route path="/results" element={<ResultsPage />} />

</Routes>

</Router>

);

}

export default App;

**Check Your Understanding:**

Review: The purpose of BrowserRouter (aliased as Router) in a React app is to enable client-side routing using the HTML5 History API. It allows your application to have multiple pages or views without requiring a full page reload.

Key Responsibilities of BrowserRouter:

* Listens to URL changes: It watches the browser’s address bar and keeps the UI in sync with the current location (window.location).
* Manages navigation: It allows you to navigate between components using <Link>, <NavLink>, or programmatically via useNavigate, all without reloading the page.
* Provides routing context: It makes the current route and navigation functions available to child components through React context.

**What happens if you navigate to a path not listed in Routes? How could you add a “Not Found” page to handle invalid URLs?**

**Bonus Challenges:**

* Add a <Route path="\*" element={<NotFound />} /> for unmatched routes.
* Wrap your layout with a global <Layout> component (e.g., header/footer).