National University of Computer and Emerging Sciences

Laboratory Manual

for

Web Engineering
(SL3003)

Lab Instructors	Sana Ejaz, Hassan Raza
Section	6C 1,2
Semester	Spring 2025

Department of Software Engineering

FAST-NU, Lahore, Pakistan

Project Nightingale

Building a Performant Post Analysis Dashboard

Scenario:

Imagine you're a junior dev at a startup. Your lead, Sarah, just dropped this task on your desk: "We need a quick internal dashboard, let's call it 'Project Nightingale', to sift through blog posts from an external source. Users need to filter posts by title instantly, and sometimes they want to quickly pinpoint the first relevant post from their filtered view. The catch? The data source *could* eventually send us thousands of posts, so this thing needs to be snappy. No janky filtering or sluggish interactions allowed! Show me you understand how to keep React performant under pressure."

Core Task:

Build the "Project Nightingale" dashboard component. It needs to fetch posts, display them, allow real-time title filtering, highlight the top filtered result on demand, and crucially, use React's optimization hooks (useMemo, useCallback) correctly to prevent performance bottlenecks. You'll also handle focus management (useRef) and the initial data load (useEffect with Axios).

API Endpoint: Use the trusty JSONPlaceholder API for posts:

`https://jsonplaceholder.typicode.com/posts`

Part 1: Getting Off the Ground (Foundation & Data)

1. **Component Scaffolding:** Create the ProjectNightingaleDashboard functional component.
2. **Essential State:** Figure out what pieces of information need to be tracked in the component's state using useState (think: the posts themselves, loading status, any potential errors, what the user is typing to filter).
3. **Fetching the Goods:** Implement a useEffect hook. This effect should only fire *once* when the dashboard first loads. Inside, use Axios to grab the posts. Make sure you handle both success (update state, turn off loading indicator) and failure (show an error, turn off loading indicator) gracefully. No one likes a mysteriously blank screen.
4. **Initial Display:** Render the basic UI. Show a loading message initially. If an error

occurs, display it clearly. Once data arrives, list out the post titles. An unordered list (ul) seems appropriate.

Part 2: The "Instant Filter" Feature (Smart Calculation)

1. **Filter Input:** Add a text input field. Hook it up to the relevant piece of state so you always know what the user is typing.
2. **The Potential Bottleneck:** Filtering is the core feature here. On every keystroke in the filter input, you'll need to decide which posts match. If we had thousands of posts, doing this naively on every single render could grind things to a halt.
3. **Memoization Magic:** How can you ensure this filtering logic *only* runs when the underlying data it depends on actually changes (i.e., the original posts list or the filter text)? Implement a solution using the appropriate React hook to memoize the *result* of your filtering calculation. Your filtering should be case-insensitive.
4. **Displaying the Right Stuff:** Update your rendering logic to show only the *filtered* posts based on the memoized result.

Part 3: Smooth User Experience (Focus Management)

1. **Guiding the Eye:** When the dashboard loads and is ready, the user will likely want to start filtering immediately. Make it easy for them. Use useRef to get a direct handle on the filter input element.
2. **Auto-Focus:** Enhance your useEffect logic. After the initial data fetch is complete (or perhaps right on mount, your call!), automatically set the browser's focus to the filter input field. Make sure this only happens *once* on the initial load.
    - **Nuance:** Think about *when* exactly the input element is available in the DOM relative to your data fetching. Calling .focus() too early won't work.

Part 4: Optimizing Interactions (Callbacks that Don't Over-React)

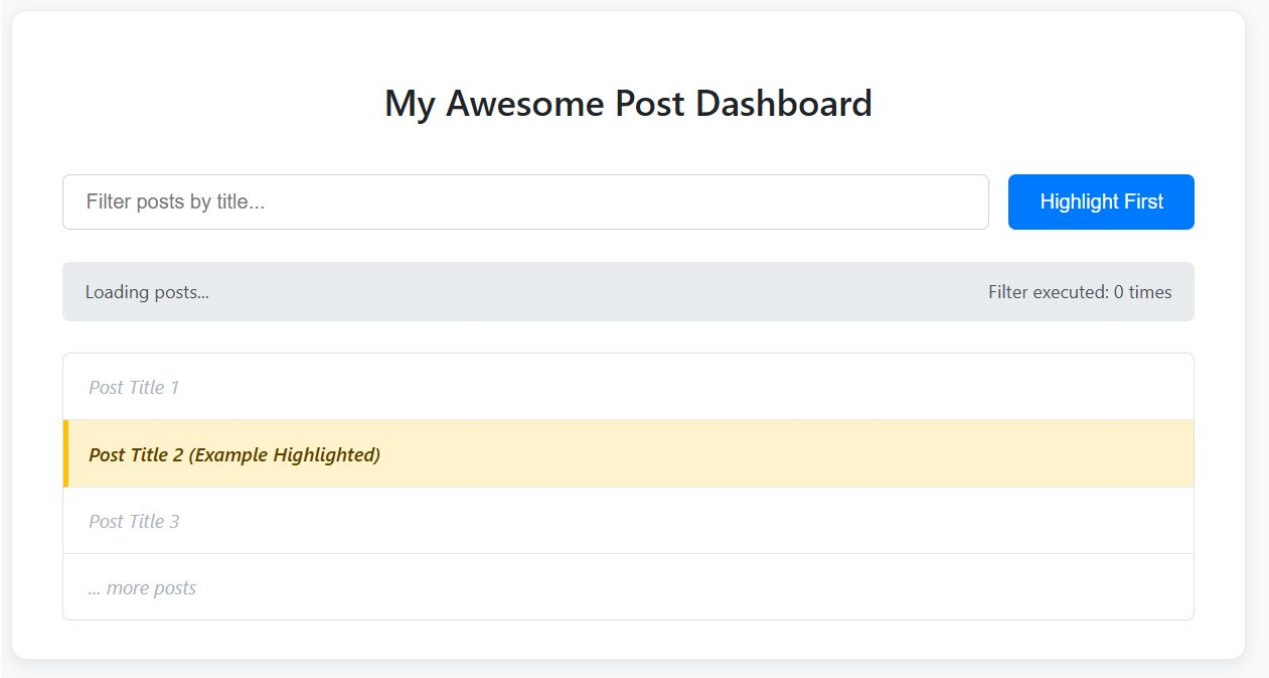
1. **"Pinpoint" Button:** Add a button, maybe labelled "Highlight Top Result". Clicking this should visually highlight the *first* post currently showing in the filtered list.
2. **Tracking the Highlight:** You'll need another piece of state to keep track of *which* post (if any) is currently highlighted. An ID seems like a good way to track it.
3. **The Handler Function:** Write the function that executes when the "Highlight Top Result" button is clicked. It needs to look at the *current filtered posts*, grab the ID of the

first one (if the list isn't empty), and update the highlight state.

4. **Callback Consideration:** Now, think about re-renders. If the dashboard re-renders for *any* reason (even unrelated to this button), should this handler function be recreated from scratch every single time? Probably not. Wrap its definition in useCallback, making sure you correctly identify any dependencies it needs from the component's scope.
5. **Visual Feedback:** Modify how you render the list items (li). If a post's ID matches the currently highlighted ID in your state, give it a distinct visual style (a background color, bold text, etc.).
6. **The "Why":** Briefly consider: If you were passing a function like this down as a prop to many child components (e.g., individual PostItem components), why would using useCallback be even *more* critical for performance? (Mental exercise, no need to build PostItem).
    - **Hint:** Unnecessary function recreation can break optimizations in child components.

Evaluation Notes:

Sarah will be looking for: Correct data fetching and state management; efficient, memoized filtering; smooth focus behaviour; optimized callback functions; accurate tracking of filter calculations; and clean, readable code that shows you understand *why* these hooks are being used, not just how. Good luck, Dev!



Here’s an UI that I quickly put together to give you an idea of what your final submission could look like.