Richard MAY

Software Engineer

London

+445552055055

richardmay@gmail.com

Frontend Developer

Application Date	Hiring Status	Time Taken
August 25, 2025	Scored	17 min

Candidate's Score

92/100

Assessment Summary

No	Question Title	Tab Switch	Time Taken	Score
1	Word Transformation Wizardry	-	7m 43s	10/13

In the mystical lands of Verbatim, young wizards train in the art of word transformation. Using their magic, they can alter texts by performing three types of spells: inserting a rune (character), removing a rune, or changing one rune into another. The challenge for these apprentices is to transform a given incantation **(word1)** into a desired charm **(word2)** using the fewest spells possible. Your task is to help these budding wizards by determining the minimal number of spells required for the transformation.

Example 1:
Input: word1 = "light", word2 = "night"
Output: 1
Explanation:
Replace 'l' in "light" with 'n', changing it to "night".
A single, precise spell is enough to cast the night.

Example 2:
Input: word1 = "silent", word2 = "listen"
Output: 2
Explanation:
Remove 's' from "silent", resulting in "ilent".
Insert 's' after 'i', changing "ilent" to "listen".
A tweak and a shuffle realize the listening charm.

```
function minDistance(word1:string, word2:string) : number {

    const m = word1.length;
    const n = word2.length;

    const dp: number[][] = Array.from({length: m +1}, () => Array(n+1).fill(0));
    for (let i = 0; i<=m; i++) dp[i][0] = i;
    for (let j = 0; j<=n; j++) dp[j][0] = j;

    for(let i = 1; i<=m; i++){
        for (let j = 1; j <=n; j++){
        if(word1[i-1] === word2[j-1]){
            dp[i][j] = dp[i-1][j-1];
        }else{
            dp[i][j] = Math.min(
                dp[i-1][j]+1,
                dp[i][j-1]+1,
                dp[i-1][j-1]+1
            );
        }
        }

    }
    return dp[m][n]

    // Insert your code here
}
```

CODING · HARD

PARTIALLY CORRECT

2 Identifying Performance Bottlenecks in a Web Application - 2m 10s 3/7

A web-based dashboard in the financial industry is experiencing slow load times and sluggish UI interactions when displaying large sets of real-time data. Without writing code, explain how you would use performance profiling tools and techniques to identify and prioritize the main areas causing these performance issues.

Browser devtools performance profiling, network analysis, memory profiling, rendering bottlenecks, audit with lighthouse

FREE TEXT · MEDIUM

PARTIALLY CORRECT

3 TypeScript Type Assertions vs. Type Guards - 56s 7/7

An engineer is working with a TypeScript function that receives a value of type

`unknown`

from an external source. The engineer wants to access the
`length`

property only if the value is a string or an array. Which approach below correctly implements both compile-time safety and runtime correctness?

○
```
function processValue(val: unknown) {
  if ((val as string | any[]).length !== undefined) {
    console.log((val as any).length);
  }
}
```

○
```
function processValue(val: unknown) {
  if (typeof val === "object" || typeof val === "string") {
    console.log((val as any).length);
  }
}
```

●
```
function processValue(val: unknown) {
  if (typeof val === "string" || Array.isArray(val)) {
    console.log(val.length);
  }
}
```

○
```
function processValue(val: unknown) {
  if ((val as any).length >= 0) {
    console.log((val as any).length);
  }
}
```

○
```
function processValue(val: unknown) {
  if (val instanceof Array || val instanceof String) {
    console.log((val as any).length);
  }
}
```

MULTIPLE CHOICE · EASY CORRECT

4 React useEffect and Redux Integration - 37s 7/7

A React component is connected to a Redux store and must fetch user data whenever the userId prop changes. Which approach correctly ensures that the fetchUserData action is dispatched only when userId changes, and not on every render?

- ○ Call the fetchUserData action directly inside the function body of the component without using any React hooks.
- ● Use the useEffect hook, pass [userId] as the dependency array, and dispatch fetchUserData inside the useEffect callback.
- ○ Use the useEffect hook with an empty dependency array so that fetchUserData is only dispatched on initial mount.
- ○ Call fetchUserData inside the useSelector function to ensure it triggers on Redux state changes.
- ○ Invoke fetchUserData in a callback function passed to onClick for manual triggering.

MULTIPLE CHOICE · EASY CORRECT

5 Isolating Side Effects in Component Tests - 33s 7/7

A frontend team develops a React component that subscribes to a global event bus on mount and unsubscribes on unmount. During component testing with a popular test runner, flaky test results are observed, with some subsequent tests unexpectedly receiving events. What is the BEST practice to avoid such side effects in component tests?

- Rely on the test runner's default cleanup to handle unsubscriptions after each test
- Manually trigger a re-render of the component after each test
- [✓ Reset all global event bus subscribers in the test framework's afterEach lifecycle hook]
- Use spies or mocks for the event bus methods only in the first test to avoid polluting global state
- Increase test timeout to allow subscriptions to settle between tests

MULTIPLE CHOICE · EASY CORRECT

6 Redux Reducer Principles - 27s 7/7

An engineer is designing a Redux reducer for a React application with the following user-related state:

```
{
  user: {
    id: 23,
    name: 'Alex',
    isAdmin: false
  }
}
```

Given the Redux guidelines, which of the following rules MUST be followed when implementing a reducer for updating the user object?

- The reducer must mutate the existing state object directly to apply updates for performance optimization.
- [✓ The reducer should always return a new state object even if only one property is changed to maintain immutability.]
- Reducers should fetch latest user details from an API before updating the state.
- The reducer should return undefined if the action type is not matched, ensuring a reset to initial state.
- Reducers must serialize the state to JSON on each update for debugging purposes.

MULTIPLE CHOICE · EASY CORRECT

7 Micro Frontend Isolation Strategies - 20s 7/7

A large-scale enterprise adopts a micro frontend architecture where teams independently deploy their applications into shared pages. During testing, a new micro frontend causes a global stylesheet collision, changing button styles across other teams' applications. Given industry best practices for micro frontend isolation, which is the MOST robust approach to prevent such cross-application style and JavaScript conflicts?

- Assign unique class names using CSS-in-JS libraries and trust all teams to follow the naming conventions.
- Configure webpack to output each micro frontend in its own JavaScript namespace without changing CSS handling.
- [✓ Wrap each micro frontend in a Web Component using the Shadow DOM for both style and DOM encapsulation.]
- Enforce global CSS resets and provide a shared, centrally managed stylesheet for all micro frontends to use.

🟥 Limit each team's micro frontend to a single page application (SPA) and avoid direct DOM manipulation.

MULTIPLE CHOICE · MEDIUM CORRECT

8 Optimizing UI Rendering with Virtualization - 33s 7/7

A frontend team is facing performance problems with a data-intensive React application. The main issue is long render times when displaying large lists of data (over 10,000 items). Based on current frontend best practices, which solution BEST mitigates rendering lag while maintaining usability and accessibility?

🟥 Increase browser memory limits and use pagination controls for list navigation

🟥 Leverage React's PureComponent or memoization to reduce unnecessary renders for all list items

🟩 Implement windowing or list virtualization so that only visible items are rendered while using ARIA roles and keyboard navigation support for accessibility

🟥 Move list rendering from the client to server-side rendering (SSR) for all requests

🟥 Use the useEffect hook to defer rendering with setTimeout whenever the list changes

MULTIPLE CHOICE · MEDIUM CORRECT

9 Advanced Git Workflow: Interactive Rebasing for Feature Branch Integration 1 41s 6/7

A team is working on a large-scale software project using Git with a trunk-based development workflow. The team wants to ensure clean, linear history for all feature branches merged into main, minimize merge conflicts, and maintain traceability of original commits. Feature branches may have multiple contributors and are long-lived. According to advanced Git practices for such a workflow, what is the most appropriate workflow for a team lead to integrate a feature branch into main upon completion?

🟥 Squash all feature branch commits into a single commit using`git merge --squash`then merge into main and push

🟥 Merge the feature branch into main using a standard merge commit with`git merge --no-ff feature-branch` to preserve branch topology

🟩 Rebase the feature branch interactively onto origin/main, resolving conflicts and rewriting commits for clarity, then fast-forward merge the rebased branch into main

🟥 Cherry-pick each commit from the feature branch onto main in commit order, then push and close the feature branch

🟥 Force push the feature branch to overwrite main, ensuring the branch commits are preserved as-is

MULTIPLE CHOICE · MEDIUM PARTIALLY CORRECT

10 TypeScript Utility Types Deep Understanding - 41s 7/7

An engineer is creating a complex, recursive, and immutable state shape using TypeScript. They want to ensure that an arbitrary nested object type becomes deeply readonly, such that all nested objects, arrays, and fields cannot be mutated at any depth. Which of the following utility types MOST correctly enforces deep immutability for any generic type T?

- ○
```
type DeepReadonly<T> = Readonly<T>;
```
- ○
```
type DeepReadonly<T> = {
  readonly [K in keyof T]: DeepReadonly<T[K]>;
};
```
- ○
```
type DeepReadonly<T> = ReadonlyArray<DeepReadonly<T>>;
```
- [✓]
```
type DeepReadonly<T> = {
  readonly [K in keyof T]: T[K] extends object
    ? DeepReadonly<T[K]>
    : T[K];
};
```
- ○
```
type DeepReadonly<T> = T;
```

MULTIPLE CHOICE · MEDIUM CORRECT

11 Micro Frontend Integration Strategy - 29s 7/7

A large enterprise project is adopting a micro frontend architecture to allow different teams to deploy and update distinct parts of the user interface independently. The application consists of Angular, React, and Vue.js micro frontends, all deployed on the same domain and rendered within a shell application. The teams want to achieve seamless routing, global state sharing, and isolated deployments—but need to avoid version conflicts and minimize runtime overhead. Which integration approach best addresses these requirements according to current industry best practices?

- ○ Mount each micro frontend in an iframe to provide total isolation and load cross-frame JavaScript messaging for routing and state sharing.
- [✓ Use a runtime JavaScript module loader such as SystemJS in conjunction with the Module Federation plugin in Webpack, allowing each micro frontend to expose and consume modules as needed.]
- ○ Build each micro frontend as a static HTML fragment and have the shell application insert them using innerHTML when navigating routes.
- ○ Bundle all micro frontends into a single monolithic JavaScript file at deploy time in the shell application and use component-level conditional rendering for route-based display.
- ○ Enforce that all micro frontends use the same JavaScript framework and share a single instance of the framework in the global window scope.

MULTIPLE CHOICE · HARD CORRECT

12 E2E Testing Race Conditions - 24s 7/7

A product team is experiencing non-deterministic failures in their JavaScript end-to-end (E2E) test suite that uses Cypress for UI automation against a production-like frontend. Complex tests include asynchronous user actions (such as drag-and-drop, modal confirmation, and batch-saving) that occasionally fail due to race conditions between UI transitions and backend responses. Given the current state of best practices for E2E test stability, which approach is MOST effective for eliminating race conditions while maintaining test maintainability and reliability?

- 🟥 Introduce explicit time-based waits (e.g., cy.wait(1000)) after each asynchronous user action to allow frontend and backend states to settle.
- 🟩 Refactor tests to always interact with the DOM using Cypress commands that automatically wait on explicit UI state assertions (e.g., cy.get(selector).should("be.visible")), and use network request interception (cy.intercept) to wait for specific API responses before proceeding.
- 🟥 Run tests in parallel across multiple browsers and use test retries to decrease the overall test flakiness, without altering asynchronous action handling.
- 🟥 Disable frontend UI transitions (such as animations and modals) in the test environment to decrease non-determinism, relying only on Cypress's built-in implicit wait mechanisms.
- 🟥 Add custom JavaScript polling in test code to query backend endpoints directly, and halt test execution until expected data states are present in the API responses.

MULTIPLE CHOICE · HARD CORRECT

13 Advanced Redux Selector Optimization - 48s 7/7

A React team notices performance bottlenecks in a component tree subscribing to complex derived state from a large Redux store. The selectors used are recomputing on every dispatch, causing unnecessary re-renders, even when the relevant slices of state are unchanged. Considering best practices, how should the team optimize these selectors to reduce unnecessary recomputations while keeping derived data consistent and performant?

- 🟥 Wrap each selector in a useMemo hook within the React component to cache the results per render.
- 🟩 Use the reselect library to create memoized selectors, ensuring selectors only recompute when their relevant input state changes.
- 🟥 Decompose all selectors into primitive functions and manually cache outputs in a module-scoped object.
- 🟥 Remove all selectors and access state directly in mapStateToProps to avoid abstraction overhead.
- 🟥 Dispatch an additional, unrelated Redux action after each successful selector call to trigger cache invalidation.

MULTIPLE CHOICE · HARD CORRECT

14 Bitbucket Webhook Security for DevOps Pipelines - 38s 7/7

A DevOps team utilizes Bitbucket to manage source code and automate deployments through a CI/CD pipeline. The pipeline triggers on any update to the main branch via a Bitbucket webhook. The team is concerned about protecting the pipeline from unauthorized or spoofed webhook requests. According to Bitbucket's advanced security features and industry best practices, which approach best ensures that only genuine Bitbucket events trigger the CI/CD pipeline?

- 🟩 Restrict incoming webhook requests to only allow connections from Bitbucket's documented IP address ranges and validate event payload signatures using a webhook secret known only to Bitbucket and the pipeline endpoint.

- ○ Configure JWT token validation on the pipeline endpoint, requiring the webhook payload to contain a user-generated token that the pipeline can verify against Bitbucket user accounts.
- ○ Require all webhook requests to use HTTPS and validate the presence of a "User-Agent: Bitbucket-Webhooks/2.0" header before allowing the trigger.
- ○ Use HTTP Basic Authentication with credentials stored in Bitbucket Repository Variables and transmitted on every webhook event.
- ○ Configure branch permissions in Bitbucket so that only authorized users can modify the main branch, preventing unauthorized webhook event generation.

MULTIPLE CHOICE · HARD CORRECT