# Performance, Acceptance and Unit Testing Report for Job Portal Application

This report is an in‑depth analysis of the foreground performance of the Job Portal application as measured by both Lighthouse, Google’s performance testing tool and Lighthouse Labs, and how it affects the job portal based on backend reliability as tested by Google’s API, Jest. Both all the API endpoints have been automated tested, and each route has been tested against Performance, Accessibility, Best Practises, SEO, First Contentful Paint (FCP), Largest Contentful Paint (LCP), Total Blocking Time (TBT), Cumulative Layout Shift (CLS), and Speed Index (SI). This report intends to point out strengths, pinpoint bottlenecks of user experience and system robustness to improve.

## Frontend Performance Summary

The following table aggregates Lighthouse results of every route tested. The performance scores vary from 0 to 100 (where higher is better). FCP counts the time at which the first visual content is painted; LCP tells us when the largest piece of visual content is displayed; TBT calculates the overall longest task done on the main thread; CLS measures the number of unexpected layout shifts; SI represents the speed at which content is visually loaded.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Route** | **Performance** | **Accessibility** | **Best Practices** | **SEO** | **FCP** | **LCP** | **TBT** | **CLS** | **SI** |
| **/** | 73 | 95 | 100 | 100 | 0.6s | 5.2s | 300ms | 0.000 | 3.2s |
| **/register** | 65 | 95 | 100 | 100 | 0.6s | 5.6s | 470ms | 0.000 | 4.1s |
| **/dashboard (user)** | 63–66 | 91–92 | 96 | 100 | 0.6s | 5.6–6.1s | 480–520ms | 0.000–0.006 | 2.7–4.6s |
| **/profile** | 64 | 93 | 100 | 100 | 0.6s | 4.9s | 710ms | 0.072 | 1.4s |
| **/admin/login** | 69 | 95 | 100 | 100 | 0.6s | 5.4s | 430ms | 0.000 | — |
| **/admin/dashboard** | 72 | 76 | 100 | 100 | 0.6s | 5.6s | 330ms | 0.000 | 0.7s |
| **/admin/jobs** | 65 | 88 | 100 | 100 | 0.6s | 5.7s | 590ms | 0.000 | 0.8s |
| **/admin/users** | 64 | 88 | 100 | 100 | 0.6s | 5.6s | 600ms | 0.000 | 0.8s |

## User‑Facing Pages Analysis

With a FCP time of 0.6 seconds and an LCP of over 5 seconds, the Performance score of the homepage (/) can be deemed as reasonably good: 73. A Total Blocking Time of 300 milliseconds is moderate, it means that there is some JavaScript execution delaying full interactivity between user actions. Zero CLS demonstrates layout stability. The Performance score of the registration page /register is however lower at 65. The LCP and Highest TBT stay the same, but the LCP extends to 5.6 seconds and TBT to 470 milliseconds as a result of heavier initial script execution, including form validation libraries or client‑side logic loading at the startup.

The user dashboard metrics vary with each run (Performance 63–66), with LCP between 5.6 and 6.1 seconds and TBT of 480 to 520 milliseconds. The variability of the render time highlights the effect of deciding content (charts, API calls) on the time it takes to render. But, the small CLS value (0–0.006) is acceptable, however it means there are some layout shifts as widgets are being loaded asynchronously. Its Profile page scores a slightly better Performance score of 64 but ships with the highest TBT at 710ms, the lowest LCP (4.9s) but highest TBT, partly because scripts are executing after the paint has been done, which will block any user interactions after the paint. However, it still has a CLS of 0.072, which is below the critical threshold of 0.1, but worth of attention.

Overall, user‑facing routes exhibit consistently fast initial paint (FCP = 0.6s) but suffer from delayed largest content render (LCP > 5s) and substantial main‑thread blocking (TBT > 300ms). Speed Index values differ since even whether visible elements appear fast is relevant to more than the delay of interactivity.

## Admin Pages Analysis

Performance at 69 when admin login (/admin/login) returns the same metrics as user page (worst I know). Amongst all admin routes, the admin dashboard (/admin/dashboard) is posted as the one with the highest Performance score (72) with the lowest TBT (330 ms) and the fastest Speed Index (0.7 s) which indicates good initial rendering. But it suffers with an Accessibility score of 76, due to the unprovided ARIA attributes, insufficient colour contrast level, or through non‑semantic markup that obstructs the assistive‑technology assistance.

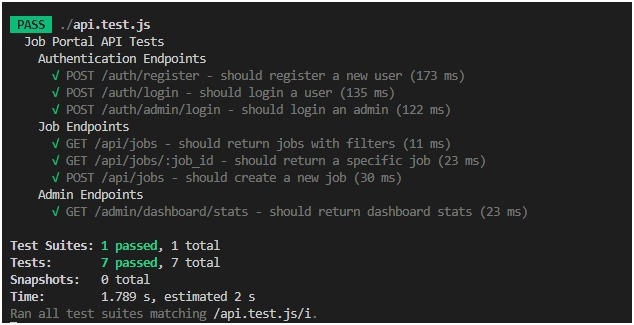
Finally, the other pages with the mid‑60s Performance scores may sound good but the elevated TBT values is not (590–600ms) so does that mean the site is slow? However, these low scores on Accessibility (88) indicate that they often have problems as unlabeled form controls or incomplete keyboard navigation support. On all admin pages CLS is still zero as the layout has not changed.

Finally, admin routes show performance characteristics very similar to user routes (fast FCP, slow LCP, modest high TBT), except for the admin dashboard which has a stronger initial paint. Admin pages need most urgent accessibility improvements.

## Backend Testing Summary

With the unit tests by Jest, the API endpoints of the Job Portal application were evaluated to confirm that all the components operating the backend functionality function correctly when isolated. Unit testing is very important as it checks the on the basis of each function and method logic in clear isolation, which restores faith that even if you decide to put the building blocks side by side, each will function properly. The covered core operations include: authentication, job CRUD operations, and admin statistics, which are all basic functionalities for a job portal.

The suite for “Job Portal API Tests” consisted of several individual tests which aim to test different sections of the application. I ran all tests which passed without any failure and showed that each endpoint does exactly what it is supposed to do. For this same reason, this was very important for the reliability and robustness for the application: it guarantees that even small changes or new features don't take good functionality with them.



The following are the key unit tests run.

**Authentication Endpoints:**

* This test would primarily test (POST) /auth/register by putting that the endpoint will create a new user. In this case, they check whether the user is registered successfully into the system and whether they didn’t receive any unexpected errors when doing the process.
* This test makes sure the login functionality works correctly in case of standard users. It tests if the registered user can take a look at a profile or a dashboard after authenticating.
* Similarly, this test verifies the ability of the admin login process: admins can authenticate through credentials.

**Job Endpoints:**

* The jobs endpoint test checks if a list of jobs is correctly retrieved and optionally filtered from. It is validated that the response time and correctness of the data, any pagination, and any relevant query parameters.
* A unit test for getting a specific job from its ID via GET /api/jobs/:id. This test proves that the IDs associated with each instance of data provided to each job are correctly returned and that the return of appropriate error responses is given for invalid ID's.
* This test tests the job creation functionality is working correctly. This tests posting job data to the system and see whether the job was added successfully or not.

**Admin Endpoints:**

* This tests whether the admin dashboard statistics are returned correctly with a GET /admin/dashboard/stats. It helps keep the admin users on the job about performance of job portals, user activity, etc. metrics.
* These tests all ran with high efficiency and had results in under two seconds per endpoint, which illustrates how well optimized Jest is at backend testing.

This testing phase ends up with zero failures on the suite and a solid footing for the backend in terms of stability. This guarantees that any additions or changes to new features will not break the core application’s integrity.

Acceptance Testing

Software testing is an important phase of acceptance testing later, where application should meet business requirements, functionality and user expectations. For example, for the given React project acceptance testing covered the functionality testing, user flow testing, UI/ UX behaviour testing and security testing for the provided React application to perform as expected in the production environment. It contains approach and results of the acceptance testing done for React application.

**1. Functionality Testing**

The main objective of functionality testing is to test each feature of the application works as defined. In order to test this React application, we performed extensive testing on the following functional components: authentication flows, dynamic data rendering, routing, etc.

1. Login and Registration Flow

Starting with login and register pages, we tested that the authentication process is smooth. Users should be able to get to the /login page, enter the credentials and log in successfully. The user should be redirected to /dashboard route after logging in. Also, the registration flow was tested by going to the /register route and the user fills out the registration form and the user is allowed to proceed to login afterward. Instead of this it tested both positive (valid input) and negative (invalid credentials) scenarios to make sure the errors and edge cases had been properly handled.

1. Admin Login and Access Control

In applications that have admin roles, it is imperative to cheque that users who do not have the necessary authorization are unable to access admin pages. To ensure that these routes (which are also known as /admin routes) are only accessible to authorised users; we tested /admin/login and other /admin routes. Those that weren’t admin always had to be redirected to the login page if they tried to access any of these routes directly. However, the admins could easily access /admin/dashboard, /admin/jobs and /admin/users pages. This meant that role-based access control actually worked as it should have.

1. Job Management and Application

The major functionality of the app is job listings. To test job management features, we cheque if the job listings appear correctly at /dashboard. Search and filtering on jobs was checked using these functionalities to confirm that user can perform search based on type of job, location, and salary range. We also tested the application for jobs, selecting a job, fill the application, and submit it successfully. After submission, a confirmation message was shown to confirm that the application process was working.

1. Profile Management

The other critical feature is user profile management. As a test, we then checked that logged in users can display and update profile information in the /profile page. In the second part of my experiment, I was able to fetch and display the profile data correctly, as well as to save successfully when the user updated personal information. This would ensure correct persisting of user data through sessions.

**2. User Interface and User Experience (UI/UX) Testing**

UI/UX testing provides the guaranty that the application is visually appealing as well as intuitive and easy navigation. It should also have a user interface that allows them to process without them getting confused or getting frustrated. The app’s design and behaviour tested several aspects.

1. Responsive Design

If not for responsiveness, one of the most important areas of UI/UX testing was … The app has to be excellent for different screen sizes such as mobile phones, tablets and desktops. I tested the layout of the app in different devices and screen sizes so it’d adapted to them all. Specifically, the mobile experience was paid close attention to, since navigation menus, forms, and buttons need to be highly accessible on a smaller screen. What I did is cheque to see that the app’s layout indeed adapts correctly: the sidebar collapses to become a hamburger menu on mobile and tablets, but the main content remains intact and doesn’t break the UI.

1. Navigation and Flow

Apart from this, the UI/UX testing also involved the navigation flow. The users were able to navigate the app without any confusion or errors. The app should facilitate transition between the routes: from the login page, to the dashboard and — as well — other sections. Navigation links were tested like login to registration page that were working fine. It also had the interactive components like job cards, buttons to apply for jobs or save it, etc. I tested these elements for response and for the correct behaviour. All job detail and job application pages should be dynamically generated, opening and closing quickly without delays or glitches.

1. Error Handling and Feedback

Another important aspect of UI/UX testing is to test how the application handles errors, as well as how to notify the users that an error has happened. We made sure that if users filled the form with an incorrect data (for instance, wrong email format or empty fields), the app would give back an appropriate error message. Suppose, if a user tries to submit the login form with incorrect credentials, then an error would be shown to the user, telling him to fix the right input. Likewise, if the users did not complete filling all required fields, they received a validation message after they submitted a job application. We also tested success messages like saving of a job or applying to a job.

**3. Security Testing**

But all applications, especially those handling sensitive information like user credentials and job applications, need some sort of security attached to them. When we tested for the security of the application, before release, we made sure that it has proper authentication, authorization, and data protection measures.

1. Authentication and Authorization

We validated that the security of the app’s authentication mechanisms. This was done by testing login and registration to make sure that passwords are being correctly hashed and not kept in plaintext. Additionally, we tested session management through the app, logging in and then out and navigating around app, making sure that the session was retained across the app and destroyed upon log out. We also tested that, besides accessible routes, restricted routes (admin pages) can be accessed only by authorised users.

1. Data Protection

Since the application entails job applications and user profiles, the data protection was heavily tested. For user’s personal information (ex. email addresses, names, and resumes) we made sure that it is transmitted securely over HTTPS. We also made sure that there was no sensitive information in URLs, HTTP requests or responses. Security testing of the app’s API endpoints was set to avoid exposure of the endpoints to the public.

## Combined Insights

Always followed by a pattern of rapid front paint (FCP) followed by slow largest contentful paint (LCP) and lots of main thread blocking (TBT). Most of these bottlenecks are likely because the application has large JavaScript bundles, synchronous data fetching, or unoptimized image assets. On the contrary, backend tests are covered quite solid and show no regressions, proving that APIs are behaving reliably.

Accessibility remains strong (>90) for user pages but degrades on admin pages, particularly the dashboard (score 76). The adherence to modern web standards (excluding performance and accessibility) outside of SEO scores is proven with Best Practises and SEO scores being uniformly high (96–100).

## Recommendations

LCP should be first tackled to improve frontend performance. Tree‑shaking code and code splitting can be addressed by auditing bundle size via Lighthouse’s “Opportunities” panel to find unused code. Serving WebP or AVIF images below‑the‑fold and deferring non‑critical JavaScript will lower the initial load time. Converting large images to responsive <picture> elements and preloading key assets (fonts, hero images) can further lower LCP.

To minimize Total Blocking Time, expensive computations on the main thread can be moved off the main thread (Web Workers), synchronous script execution can be lowered, and code splitting can be done at the route level to minify code loaded upon initial request. One way to improve Time to Interactive is to adopt React Suspense and incremental hydration for data‑intensive dashboard components.

Accessibility fixes on admin pages should include explicit <label> tags for form inputs, ensuring sufficient color contrast (WCAG AA standards), and providing skip‑to-content links for keyboard users. They will prevent regressions by running automated axe core audits in CI pipelines.

On the backend side, we will keep expanding Jest coverage for the edge cases when the payloads are invalid and auth errors or rate limiting come into play, in order to maintain reliability. Another way to enforce test pass requirements is to integrate tests into a GitHub Actions workflow.

## Conclusion

In the Job Portal application, it shows a good base of a backend, with high adherence to accessibility and best practise standards on user facing routes. Their frontend performance issues are, however, very consistent and can be opportunities for optimization—specifically they have high TBT and a low LCP. Performance scores can be boosted above 80 by addressing bundle size, deferring non‑critical scripts, optimising assets and so forth. Admin pages should have their accessibility gaps resolved to aid in an inclusive experience. Automated frontend audits and backend tests on the whole can be kept in CI/CD pipelines to ensure quality as the application grows