PROJECT REPORT

Development of the CareerCanvas Job Portal

SUBMITTED BY:

NASIRA RIAZ

SUBMITTED TO:

MA'AM AIMAN

SUBMITTED ON:

AUGUST 09, 2025

Contents

Introduction

The objective of this project was to develop a modern, fully functional job portal application named CareerCanvas. The primary goal was to create an intuitive and responsive user interface where users can search for, filter, and view detailed job listings. The application was built as a single-page application (SPA) using the React.js library and integrates with a live, third-party API to source its job data.

Project Setup and Architecture

The project was initialized using create-react-app to establish a standard development environment. From the beginning, I focused on creating a well-organized and scalable file structure to ensure the project was easy to manage. The core application logic within the src directory was organized into several key folders:

- **/components:** This folder was used for all reusable UI elements. I created a subfolder, /home, for components that were only used on the homepage, while general components like Header.js, Footer.js, and JobCard.js were kept in the main components directory.
- **/pages:** This folder contains the top-level component for each route in the application, such as Home.js, Jobs.js, and JobDetails.js.
- **/hooks:** This folder was created to hold custom React Hooks that manage specific pieces of logic, particularly for handling API interactions.
- **/assets:** This folder served as a central location for all static files, including all images and SVG icons used throughout the project.

Development Steps and Task Implementation

The development process was approached in a structured, task-oriented manner, starting with the visual layout and progressively adding complexity and functionality.

Task 1: Building the Static UI and Core Components

The first step was to build the main visual structure of the application without any dynamic data. I began by creating the primary layout components, Header.js and Footer.js, which are used on every page. Following this, I constructed the static version of the homepage, building out the

individual sections such as the Hero search area, the Categories grid, and the HowItWorks section. This allowed me to focus entirely on styling and layout using CSS, ensuring the visual foundation was strong before introducing any JavaScript logic.

Task 2: Integrating the JSearch API

With the static UI in place, I moved on to integrating the live JSearch API to fetch real-time job listings. I used the **Axios** library for handling the HTTP requests. To keep the data-fetching logic clean and reusable, I created two custom React Hooks:

1. **useJobSearch:** This hook was designed to handle API calls to the main search endpoint. It takes search queries and filter options as arguments and manages its own loading and error states.
2. **useJobDetails:** This hook was created to fetch the detailed information for a single job by taking a job ID as its argument.

To maintain security, the API key was stored in a .env file and accessed through environment variables, preventing it from being exposed in the public source code.

Task 3: Displaying Job Data Dynamically

Once I could fetch data, the next task was to display it on the screen. I created the JobCard.js component to render a single job listing in a card format. On the main Jobs.js page, I utilized the useJobSearch hook to get the array of job data. I then used the JavaScript .map() method to iterate over this array and render a JobCard component for each job. I also implemented logic to show a loading spinner while the API call was in progress and to display an error message if the fetch failed.

Task 4: Developing the Advanced Filtering System

This was a key feature of the application. I built the Filters.js component to manage all the different user inputs, including text fields for keywords, checkboxes for job types, and radio buttons for the date posted. The component uses its own state to track what the user has selected. When the "Apply Filters" button is clicked, a callback function (onApplyFilters) sends the current state of the filters up to the parent Jobs.js page. The Jobs.js page then uses this information to construct a new, more specific query for the useJobSearch hook, which triggers a new API call.

Task 5: Implementing Client-Side Routing

To create a seamless single-page application experience, I used the **React Router** library. I configured the main App.js file to define the different routes for the application, such as /, /jobs, /about, and /contact. I also created a dynamic route, /job/:id, for the Job Details page. This route uses the ID from the URL as a parameter to fetch the correct job data using the useJobDetails hook.

Task 6: Finalizing Styles and Ensuring Responsiveness

Throughout the development process, I wrote custom CSS for each component. I established a consistent theme by defining global color and font variables in the main index.css file. I used modern CSS layout techniques like **Flexbox** and **Grid** to structure the pages. Finally, I added **media queries** to the CSS files to ensure that the layout would adjust properly on different screen sizes, resulting in a fully responsive design that works well on both desktop and mobile browsers.

Challenges Faced and Solutions

During the project, I encountered a few technical challenges that required problem-solving:

1. **CSS Specificity Conflicts:** When I first implemented the job cards, the button styles were not applying correctly because the default styles from the Bootstrap library were overriding my custom CSS. I used the browser's developer tools to diagnose this specificity issue. I resolved it by making my CSS selectors more specific (e.g., using .job-card .btn-primary instead of just .btn-primary), which gave my rules a higher priority.
2. **GitHub Large File Upload:** My initial screen recording of the application was over 200 MB, which exceeded GitHub's 100 MB file size limit. My first attempt to push the code failed. To solve this, I had to learn and implement **Git LFS (Large File Storage)**, which is the official tool for handling large files in a Git repository. I installed it, configured it to track .mp4 files, and was then able to successfully push the large video file to the repository.

Conclusion

The CareerCanvas project was a valuable exercise in building a complete, data-driven web application with React. Through this project, I have demonstrated my ability to structure a scalable application, manage component state, integrate with external APIs, and create a polished, responsive user interface. The process of overcoming challenges, such as CSS conflicts and Git LFS configuration, has significantly improved my practical problem-solving skills as a developer.