ChatGPT

Job Portal Web Application – Software Requirements Specification (SRS)

1. Introduction

1.1 Purpose

This SRS document defines the goals, features, and current status of the Job Portal web application. The purpose is to describe **what** the software will do and **how** it will behave, serving as a blueprint for both development and stakeholder review. As defined by industry standards, an SRS "is a document that describes what the software will do and how it will be expected to perform" [1]. In this case, the job portal is intended to let users **search for jobs, read related content, and interact with employers**, while providing an administrative interface for content management. The project aligns with the client's vision of a career site where people can find job listings, read categorized articles, create accounts, and manage postings [2] [1]. Key objectives include: - Enabling job seekers to browse and search jobs by title, location, or category.
- Allowing content consumption (e.g. articles, courses, migration guides) and subscriptions to updates.
- Providing secure user registration and login (initially with email/password).
- Offering an admin panel for managing jobs, courses, stories, migrations, and user accounts.

1.2 Scope

The Job Portal is a standalone web application comprising a public-facing site and a protected admin interface. **In scope** are features such as: user account creation and login; job listings (create, read, update, delete by admins); searchable job directory; informational pages (About, Course list, Story list, etc.); and an admin dashboard for content oversight. The application will run in browsers (desktop/mobile web) and use a Node/Express backend with a MongoDB database. **Out of scope** for the initial release are complex features like mobile SMS/OTP authentication and a paid subscription/payment system, as these are planned for later phases [3] [4]. In other words, the MVP focuses on core job-posting and user management functionality; advanced features (payments, two-factor auth) are deferred to future iterations.

1.3 Intended Audience

This document is intended for all stakeholders of the Job Portal project, including:
- **Project Sponsors and Managers:** to verify that the system meets business goals and to track project progress.
- **Development Team (Front-end & Back-end):** to guide implementation of features and architecture.
- **Quality Assurance/Testers:** to understand expected behaviors and features for validation.
- **UI/UX Designers:** to ensure design aligns with requirements (e.g. color scheme, layout references).
- **Business/Non-Technical Stakeholders:** such as marketing or operations staff, who need a high-level understanding of functionality and status.

1.4 Definitions and Acronyms

- **SRS:** *Software Requirements Specification* – a document detailing software functionality and performance [1].
- **UI:** *User Interface*.
- **API:** *Application Programming Interface*.
- **OTP:** *One-Time Passcode* (SMS-based login code, planned but not yet implemented) [4].
- **MERN:** A software stack (MongoDB, Express.js, React.js, Node.js) used by this project.
- **JWT:** *JSON Web Token* – a format for secure authentication tokens (planned for login) [5].
- **CRUD:** Operations to Create, Read, Update, and Delete data.
- **Vite:** A modern build tool for front-end projects (used to bootstrap the React app) [6].
- **Axios:** A JavaScript HTTP client used to call backend APIs [7].
- **Context API:** A React feature for managing global state (used optionally in this project) [8].

2. Overall Description

2.1 Product Perspective

The job portal is a new, standalone web system. It does not rely on any existing software but follows a standard *client-server* architecture: the React-based front end communicates via APIs with a Node.js/Express backend, which stores data in a MongoDB Atlas database [9]. The application is similar in purpose to commercial job boards (e.g. Indeed or TopJob), but it also includes career-related content (courses, migration guides, stories) as part of the offering. The front end runs in users' web browsers (desktop or mobile), and the server is designed to be hosted online (e.g. via Vercel). The product expects internet connectivity and a modern browser; no special hardware is needed beyond a standard web hosting environment. All features are self-contained within this system (future integrations like SMS gateways or payment processors are not part of the current scope).

2.2 Product Functions

Key functions of the Job Portal include:
- **User Account Management:** Visitors can register (email/password) and log in to their accounts. (Future: mobile OTP login will be added [4].)
- **Job Browsing:** Users can view and search job listings. The Jobs page provides a search bar and displays job cards with title, company, location, category, etc. [10].
- **Job Content Consumption:** Users can read job-related articles (stories) and browse course listings. Currently, the *Stories*, *Courses*, and *Categories* pages are placeholders showing static content [11].
- **Informational Pages:** An *About* page presents mission, statistics, and values [12]. A *Home* page highlights major features (search banner, categories grid, sample jobs/courses/stories) [13].
- **Subscription/Notifications:** Users can subscribe to updates (e.g. newsletter or job alerts). (Feature design noted, implementation pending.)
- **Search and Filtering:** On the Jobs page, users can filter or search jobs by keywords, location, or category [14].
- **Admin Dashboard:** An admin user has access to a Dashboard page showing key metrics (total jobs, users, courses, migrations) [15] [16].
- **Content Management (Admin):** Admin can *Create, Update, Delete* job postings, courses, migration guides, and stories. Each content type has a management interface (e.g. Manage Jobs, Manage Courses) listing

items and providing edit/delete buttons [17] [18].
- **User Management (Admin):** Admin can view all registered users, search/filter by name/email/phone, and deactivate or flag accounts [19].
- **Security:** The system will enforce authentication and authorization rules (planned via JWT tokens and role-based access) [5].

Each of these functions will be documented in detail as features are completed.

2.3 User Classes and Characteristics

- **Visitor (Guest):** Unauthenticated users who can browse public content (Home, About, Jobs listings, Courses list, Stories) and view job details. They can register or log in to access protected features.
- **Registered User (Job Seeker):** A user who has created an account. This user can log in, personalize settings, save favorite jobs or subscribe to alerts. They can browse all job listings and content but cannot edit site content. They use typical consumer-grade devices (desktop/mobile browser, internet-savvy but not necessarily technical).
- **Administrator:** A staff member with access to the admin interface. The admin manages site content and users. Tasks include posting new jobs/courses/stories, editing existing ones, reviewing users, and monitoring site statistics. Admin users are assumed to be comfortable with web-based dashboards and basic data entry.
- **System (Developer/Maintainer):** Not an end-user of the site, but the technical role that uses this SRS. They set up environments and deploy the application; their needs are addressed under implementation constraints and architecture rather than user functions.

2.4 Operating Environment

- **Client (Frontend):** Runs in modern web browsers (Chrome, Firefox, Safari, Edge) on Windows, macOS, or mobile OS. The React.js application is built with Vite [6] [7].
- **Server (Backend):** Node.js environment (e.g. Node 18+) running an Express.js application. Locally it listens on port 5000 by convention [20], but in production it will be hosted on a cloud platform (e.g. Vercel Serverless or Render).
- **Database:** MongoDB Atlas (cloud database) accessed over the internet. The backend uses Mongoose ORM to connect to Atlas [21].
- **Development:** The development setup uses npm/Yarn, with project directories for client and server. Code is version-controlled (e.g. Git).
- **Deployment:** Frontend is deployed to Vercel (or similar) for static hosting [22], and the backend is deployed either as serverless functions on Vercel or on a Node-capable host.

2.5 Design and Implementation Constraints

- **Technology Stack:** Must use React.js for the frontend and Node.js/Express for the backend [23] [7]. MongoDB Atlas is mandated as the database [24]. These choices constrain library compatibility and architectural decisions.
- **Styling:** The UI must use the specified color scheme (blue and yellow accents) and a clean, professional design. Although Tailwind CSS was initially included, the final design uses plain CSS/SASS for styling [25]. All pages must be responsive for mobile devices (e.g. collapsing the navigation into a mobile menu) [26].

- **Data Models:** The content types (Jobs, Courses, Migrations, Stories, Users) have defined fields (e.g. jobs have title, company, location, description, image [17]; courses have title, image, subtitles/content [18]). These schemas constrain the database design.
- **Authentication:** Security will use industry-standard practices: passwords hashed with bcrypt, authentication via JWT tokens [5]. Role-based access must protect admin routes. Without a paid OTP service yet, SMS-based login is not currently active.
- **API Design:** The backend will expose RESTful APIs (e.g. `/api/jobs`) to be consumed by the frontend. These APIs must handle JSON and enforce CORS. Any third-party APIs (email/SMS) are not included initially.
- **Hosting Limitations:** Using Vercel's serverless functions means no long-running processes or in-memory sessions; the app must be stateless. Environment variables (for DB connection, JWT secrets) must be managed securely.
- **Browser Compatibility:** The app must work on major browsers; polyfills or fallbacks may be needed for older browser support (per user needs).

2.6 Assumptions and Dependencies

- **User Assumptions:** Users have internet access and use modern browsers. They understand basic web navigation. Mobile/tablet support is assumed but not all mobile-specific gestures are required.
- **Email/Notifications:** An email service (SMTP) will be available for account verification or OTP in the future. Currently, we assume email/password login only [3] [4].
- **Third-Party Services:** MongoDB Atlas is assumed to be set up and accessible. Vercel (or chosen host) is available for deployment. Any use of external APIs (e.g. for maps or analytics) is low-priority and assumed to be added later.
- **Development Tools:** Developers have Node.js and npm installed. GitHub (or similar) is used for version control. Node packages like express, mongoose, bcryptjs, jsonwebtoken, cors, and dotenv are assumed available [27].
- **Content Assets:** Logos, images, and content text are provided or will be sourced under proper licenses. The `/assets` folder holds UI images/icons [9].
- **Project Scope:** It's assumed that additional features (e.g. multi-language support, mobile app) may be added later but are not dependencies for the current SRS.

3. System Features (Implemented or In Progress)

3.1 Frontend Features

- **Home Page:** Includes a hero banner with a job search bar (to query job listings), followed by a grid of category cards (each with image, title, description) [13]. It also shows sample job cards, story cards, and course cards (static examples) in sections. A standard footer appears at the bottom. The layout is fully responsive.
- **Login/Register Page:** A single page (`Auth.jsx`) with toggleable tabs or panels for "Login" and "Register". The login form collects Email and Password, plus a "Forgot Password" link; the registration form collects Name, Email, Password, and Confirm Password [4] [28]. The design uses the site's primary colors (blue #0B669A and yellow #FDC62C) for buttons and accents [29]. On larger screens, a welcome illustration or text is shown alongside the form [30].
- **Jobs Page:** Displays a search/filter bar at the top and a list of JobCard components showing job details (title, company, location, etc.). This page uses the same header and footer. (Currently, job cards are static data; filtering is planned.)

- **About Page:** Contains site/company information: mission statement, key statistics (e.g. number of jobs, users), and core values [12].
- **Content Pages:** The Courses, Stories, and Categories pages exist as placeholders. They currently display static example cards or text, indicating future dynamic content. (For example, `Courses.jsx` shows a list of course cards with title and image.)
- **Reusable Components:** The app uses a common `Header.jsx` (with navigation links and a responsive menu) and `Footer.jsx` across all pages [31]. Card components (`JobCard.jsx`, `CourseCard.jsx`, `StoryCard.jsx`) are used to render lists of jobs, courses, and stories in a uniform style [31]. A `Categories` component shows the grid of category tiles on the Home page.
- **Responsive Design:** All pages use CSS flex/grid layouts to adapt to different screen sizes. For example, the header collapses into a mobile menu icon on narrow screens. (Some responsiveness tweaks remain, such as menu icon behavior [32].)

3.2 Admin Features

- **Admin Layout:** Under the `src/admin/` directory, a dedicated React layout (`AdminLayout.jsx`) wraps all admin pages with a sidebar navigation. The `AdminSidebar.jsx` component lists links: Dashboard, Manage Jobs, Manage Courses, Manage Migrations, Manage Stories, Manage Users, and Settings [33] [34]. The active page is highlighted dynamically. The sidebar is positioned vertically, while the main content appears to the right.
- **Dashboard:** Shows an overview of site metrics: total number of job postings, registered users, courses, and migrations (guides) [15] [16]. Each metric is displayed as a card or tile. (Currently the counts are static or zero; dynamic data is pending backend integration.)
- **Manage Jobs:** An interface for administrators to add, edit, or delete job postings [17]. The page lists existing jobs (with only the title visible, by default), and each list item has “Edit” and “Delete” buttons. Admins can search the job list by name, location, or category [14]. When adding or editing a job, a form collects fields: title, company, location, category, description (body), and image URL. (Implementation in progress.)
- **Manage Courses:** Allows admins to add/edit/delete courses [18]. Each course has a title, an image, and multiple “subtitles” (sections) each with content. The course list view shows course titles with Edit/Delete actions.
- **Manage Migrations:** Similar to courses, this section manages migration-related articles [35]. Each migration entry has a title, image, and multiple content sections. The list view and forms mirror the Courses layout.
- **Manage Stories:** Lets admins add/edit/delete story entries [36]. Stories have a title, image, and body content. The list shows story titles with action buttons.
- **Manage Users:** Displays a table of registered users [19]. Admins can search or filter by name, email, or phone number. Each user entry has options to deactivate or flag the account. (Currently this is a static UI; integration with the User model is planned.)
- **Settings:** A placeholder page for future settings (e.g. site configuration, account settings).
- **Components:** Common admin components include `AdminHeader.jsx` (optional) and `AdminSidebar.jsx`. The `admin.css` file holds styles for admin pages.

3.3 Pages and Components Structure

- **Client/Server Directories:** The project has two main folders: `/client` for front-end code and `/server` for back-end code [9].
- **Client (`/client`):** Contains the React application. Within `src/`, there are subdirectories:

- `/src/pages/` – page-level components (Home.jsx, Jobs.jsx, About.jsx, Auth.jsx, etc.).
- `/src/components/` – reusable UI components (Header.jsx, Footer.jsx, JobCard.jsx, etc.).
- `/src/admin/` – admin interface components and pages (Dashboard.jsx, ManageJobs.jsx, etc.) [34]. Includes subfolder `/admin/components` with `AdminSidebar.jsx`, `AdminHeader.jsx`, and `admin.css`.
- `/src/context/` – global state providers (if using Context API).
- `/src/hooks/` – custom React hooks.
- `/src/services/` – modules for making API calls via Axios.
- `/src/styles/` – global CSS/SASS styles.
- `App.jsx` – main app component configuring routes (with React Router).
- `main.jsx` – entry point rendering the App.
- **Server (`/server`):** Contains the Node/Express API code. Key subdirectories (as generated) are: [37]
- `/server/config/` – e.g. `db.js` for database connection settings.
- `/server/controllers/` – business logic for handling requests (e.g. `authController.js`, `jobController.js`).
- `/server/models/` – Mongoose schema definitions (`User.js`, `Job.js`, etc.).
- `/server/routes/` – API route definitions (`authRoutes.js`, `jobRoutes.js`, etc.).
- `/server/utils/` – helper modules (e.g. `asyncHandler.js`, `errorResponse.js`).
- `server.js` – main application entry point that sets up Express, middleware, and listens on a port.
- `.env` – environment variables (database URL, JWT secret).
- **Data Flow:** The frontend makes HTTP requests to endpoints like `/api/v1/jobs`, `/api/v1/auth/register`, etc. The server handles these and performs database operations. This clear separation of directories reflects the client-server architecture [9].

4. Technologies Used

- **Frontend:** React.js (bootstrapped with Vite) for building the UI [7]. React Router DOM handles client-side page routing, and Axios is used for AJAX calls to the backend [7]. React Context API may be used for global state (e.g. user auth state) [8]. CSS is used for styling; initially Tailwind CSS was installed, but the final UI employs handcrafted CSS (with global styles and component-specific styles) [25]. Icons and images come from the `/assets` directory.
- **Backend:** Node.js with the Express framework serves as the server. Key libraries include Mongoose (for MongoDB object modeling), bcryptjs (for password hashing), jsonwebtoken (for JWT authentication), cors (for cross-origin resource sharing), and dotenv (for environment variables) [27]. The server enforces RESTful routes under `/api/v1/`.
- **Database:** MongoDB (Atlas cloud) is the data store for all persistent data (users, jobs, courses, etc.) [21]. Mongoose schemas define the data models.
- **Authentication:** Uses JWT tokens and bcrypt password hashing (standard libraries) [5]. The system is set up for email/password auth now, with a future plan for mobile OTP.
- **Development Tools:** Vite (frontend build tool), npm/Yarn for package management, and Git for version control.
- **Deployment:** Frontend is planned to be deployed on Vercel (or similar static host) [22]. The backend may be hosted via Vercel Serverless Functions or a Node hosting service (e.g. Render) [22]. The MongoDB Atlas DB is cloud-hosted.
- **Other:** JWT for secure API authentication; bcryptjs for hashing; Axios for HTTP; Context API for app state; any other NPM libraries as needed (e.g. react-icons for iconography).

5. System Architecture and Folder Structure

The system uses a traditional client-server architecture: a React front end (client) communicates with a Node/Express back end (server), which in turn accesses the MongoDB database.

- **Client-Server Overview:** When a user interacts with the UI (e.g. searches for jobs), React components dispatch API requests (via Axios) to Express routes. The server controllers process these requests, perform CRUD operations using Mongoose, and return JSON responses. This stateless interaction uses HTTP/HTTPS. Authentication (login) yields a JWT, which the client stores (e.g. in localStorage) and sends in headers for protected API calls.

- **Server Folder (`/server`):** As shown below, the server directory has separate concerns. [37]

- `config/db.js` – sets up the MongoDB connection.

- `controllers/` – functions to handle requests (e.g. register user, create job).
- `models/` – Mongoose schemas (`User.js`, `Job.js`, etc.).
- `routes/` – route files (e.g. `authRoutes.js` handles `/api/v1/auth/*`, `jobRoutes.js` handles `/api/v1/jobs/*`).
- `utils/` – utility code (error handling, async middleware).
- `server.js` – main Express app (loads middleware, routes, and starts the HTTP server).

- `.env`, `.gitignore` – configuration and ignore files.

- **Client Folder (`/client`):** The client directory (React app) is structured as follows [38]:

- `src/pages/` – Page components like `Home.jsx`, `Jobs.jsx`, `About.jsx`, `Auth.jsx`, etc. Each page corresponds to a route.

- `src/components/` – Shared UI components (`Header.jsx`, `Footer.jsx`, `JobCard.jsx`, `CourseCard.jsx`, `StoryCard.jsx`, etc.).
- `src/admin/` – Admin panel components. Under `/src/admin/` we have pages like `Dashboard.jsx`, `ManageJobs.jsx`, plus an `components/` folder (with `AdminSidebar.jsx`, `AdminHeader.jsx`) and `admin.css` for admin styles [34].
- `src/context/` – For any React Context providers (e.g. AuthContext).
- `src/services/` – API service modules (e.g. `jobService.js`, `authService.js`) that wrap Axios calls.
- `src/styles/` – Global stylesheet(s), e.g. for resetting or theme.
- `App.jsx` – Main component setting up React Router routes and context providers.

- `main.jsx` – The application entry point that renders `<App />`.

- **Admin Folder:** Notably, the admin UI is separated under `src/admin/` as shown [34]. This keeps admin pages and components isolated from the public pages. For example, `src/admin/Dashboard.jsx` is distinct from `src/pages/Dashboard.jsx`. There is a top-level `admin.css` for admin-specific styling.

- **Assets:** Images and icons (e.g. logo, illustration) are placed in an `/assets` or `/public` directory and referenced by the React components.
- **Architecture Diagram (conceptual):**

(Client React UI) → **Axios/API** → (Express Server) → **Mongoose/DB** → (MongoDB Atlas)

This architecture ensures clear separation between the presentation layer and the data/API layer [9].

6. Current Progress

- **Frontend Pages Completed:** The public-facing UI pages have been built in React. This includes: Home, About, Jobs, and Auth (Login/Register). These pages use static example data (no live API yet) but demonstrate the intended layout and components [13] [28]. The Home page has a working search bar (non-functional placeholder) and category/job/course/story sections. The About page displays site info (mission, stats) [12]. The Auth page's forms collect user input correctly [4].
- **Components Completed:** A set of reusable components are in place: `Header.jsx` (responsive navigation), `Footer.jsx`, `JobCard.jsx`, `CourseCard.jsx`, `StoryCard.jsx`, and a `Categories` grid component [31]. These components are integrated into the pages. Basic CSS styling (colors, spacing, shadows) has been applied according to the design.
- **Admin UI in Progress:** The admin interface's framework is set up. The `AdminSidebar.jsx` navigation works and highlights the active link. The **Dashboard** page is implemented and shows static count widgets [16]. The **Manage Jobs** page has been started (React form and list layout) [39]. Other admin pages (Courses, Migrations, Stories, Users) have placeholders ready but need forms and functionality; they are marked "upcoming" in the code [40].
- **Routing and Layout:** React Router is configured for all routes (`/`, `/jobs`, `/about`, `/auth`, `/admin/*`). Public pages use the main header/footer layout; admin pages use the AdminLayout with sidebar. Context (Auth context) is scaffolded but not fully implemented.
- **Backend Setup:** The server directory structure has been created with `npm init` and dependencies installed (express, mongoose, etc.) [27]. However, API endpoints are not yet implemented, and the database connection is prepared but not live. User authentication routes (register/login) are pending.
- **Authentication & Security:** The frontend registration and login forms exist. Client-side validation (e.g. password confirmation) is in place. No real JWT handling or password hashing is done yet (planned next). Mobile OTP login is noted as a future feature [3] [4].
- **Styling and Responsiveness:** The site uses the agreed color palette (blue and yellow) and card-based UI. The login/register page is styled with rounded corners and soft shadows as requested [30]. Pages are generally responsive on desktop; minor mobile layout bugs (like the header menu) were identified and are being fixed.
- **Known Limitations / Pending Items:**
- **Backend Integration:** No API is functional yet – all data is currently hardcoded. Full CRUD endpoints (jobs, courses, etc.) need to be developed. User data is not saved to a database yet [5].
- **Authentication Logic:** Server-side auth (JWT, password hashing, route protection) is not in place. The planned OTP flow is not integrated [5].
- **Admin Pages:** Manage Courses/Migrations/Stories/Users pages are UI-only placeholders. Editing and deletion forms need implementation.
- **Search/Filter:** The job search/filter UI exists but needs the logic to query the database.

- **Subscription/Payments:** Features like user subscriptions to alerts and any paid or premium functionality are not implemented.
- **Testing:** Automated tests (unit/integration) have not been written yet.
- **Deployment:** The app is not deployed; deployment scripts and configuration (Vercel or other) are still to be set up.

Overall, the project has a complete front-end scaffold and design, with the core look-and-feel established. The next major steps are to implement the backend APIs, connect to the database, and enable real data flow between client and server.

7. Appendix

7.1 Color Scheme

- **Primary Colors:** Blue (#0B669A) and Yellow (#FDC62C) [29]. These colors are used for buttons, headers, and highlights to create a professional accent.
- **Neutral Colors:** White background for content areas; dark gray/black text for readability. Soft shadows and light gray (#F5F5F5) backgrounds are used on cards to give depth. Rounded corners (e.g. 4-8px radius) are applied to buttons and cards as per modern design.
- **Typography:** (Assumed) Sans-serif fonts for a clean look. Font sizes follow a scalable hierarchy (e.g. 16px base).
- **Spacing and Layout:** Consistent margins/padding (e.g. 16px grid) ensures a clean layout. Forms and cards have uniform styling.

7.2 Key UI References

- **Home Page Layout:** A hero section with search bar (similar to popular job sites), followed by a responsive grid of category cards (image + title + short desc) [13]. Job, story, and course sections use uniform card components with images.
- **Login/Register Design:** Two-panel form (on desktop) with welcome graphic or text on the left and form on the right [4] [30]. The forms use the primary blue/yellow accents (e.g. blue “Login” button, yellow “Register” highlight).
- **Admin Dashboard:** Displays metric cards (Jobs, Users, Courses, Migrations) in a row or grid [15] [16]. The sidebar uses clear labels/icons for each section. This follows standard admin dashboard UI patterns.
- **Card Components:** Jobs, courses, and stories are presented as cards with an image, title, and brief info. All cards share a consistent look (white background, slight shadow, bottom-aligned buttons) [13] [31].
- **Icons/Imagery:** Minimal icons (e.g. search icon in the search bar) and illustrative images (e.g. hero graphic, content images) are used to make the UI friendly. Placeholder images are in `/assets`.

These references ensure the UI has a consistent, user-friendly design aligned with the project’s branding and requirements [30] [13].

Sources: This specification is based on the project’s documented design and current implementation notes [2] [17] [7], industry best practices for SRS documents [1], and the development team’s progress reports.

---

1 How to Write an SRS Document (Software Requirements Specification Document) | Perforce Software
https://www.perforce.com/blog/alm/how-write-software-requirements-specification-srs-document

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 job_portal_documentation.pdf
file://file-XvAzBWKLhkgNpjw4PakoXP