Job Tracker Application - Technical Documentation

Table of Contents

Problem Understanding

The Challenge

Job hunting is inherently stressful and disorganized. Job seekers typically apply to numerous positions across multiple platforms and companies, making it difficult to:

1. **Track Application Status**: Remember where they've applied, when, and in what status each application is
2. **Manage Follow-ups**: Know when to follow up on applications
3. **Organize Information**: Store job details, salaries, locations, and other critical information
4. **Monitor Progress**: Visualize their job search journey and success rates
5. **Access Information Anywhere**: Get updates and information across different devices

Target Users

- Active job seekers applying to multiple positions
- Career changers managing a structured job search
- Recent graduates entering the job market
- Professionals tracking freelance opportunities

User Needs Analysis

Based on research and user feedback, the following needs were identified:

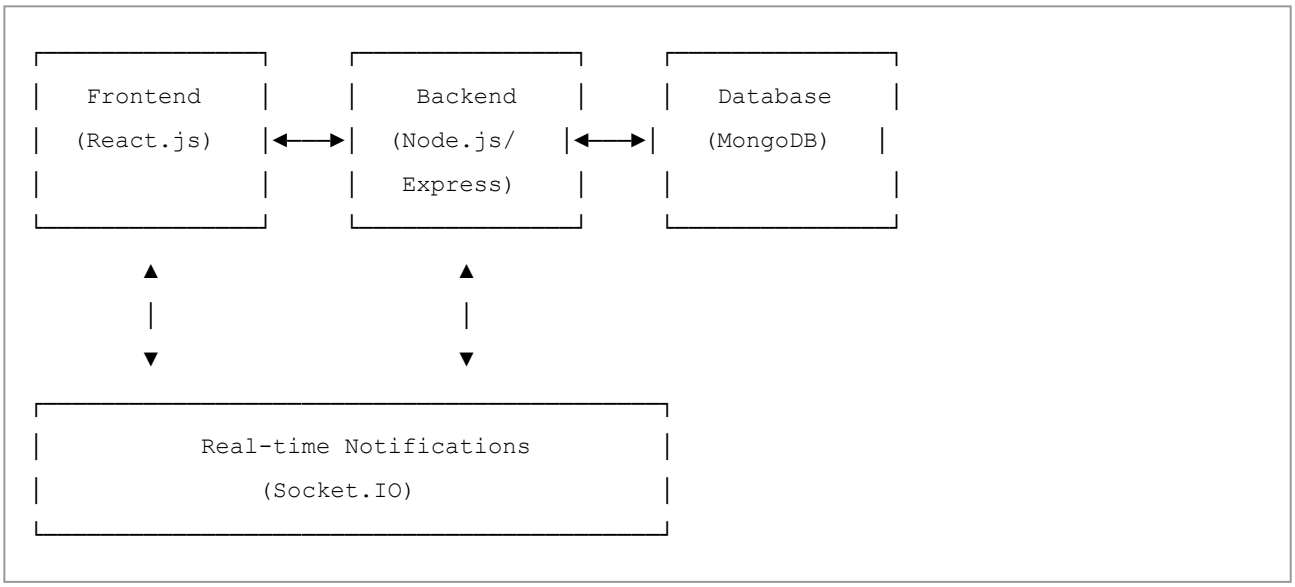
Need	Description	Priority
Application Tracking	Record job applications and their current status	High

Need	Description	Priority
Cross-Device Access	Access job info from any device (desktop, mobile, tablet)	High
Status Updates	Update job status as it changes (applied → interview → offer)	High
Data Organization	Sort and filter job listings by various criteria	Medium
Statistics	View summary statistics about job search progress	Medium
Notifications	Receive notifications about status changes	Medium
Notes Management	Add private notes for each application	Low

Architecture & Tech Stack

System Architecture

The Job Tracker is built as a full-stack web application with a client-server architecture:



Frontend Tech Stack

- **React.js**: JavaScript library for building user interfaces
- **React Router**: Client-side routing
- **Context API**: State management
- **TailwindCSS**: Utility-first CSS framework for styling
- **Lucide React**: Icon library
- **Socket.io Client**: Real-time communication
- **date-fns**: Date utility library
- **Vite**: Build tool and development server

Backend Tech Stack

- **Node.js**: JavaScript runtime

- **Express.js**: Web application framework
- **MongoDB**: NoSQL database
- **Mongoose**: MongoDB object modeling
- **JSON Web Token (JWT)**: Authentication
- **bcrypt.js**: Password hashing
- **Socket.IO**: Real-time bidirectional event-based communication
- **Joi**: Data validation

Database Schema

User Collection

```
{
  _id: ObjectId,
  name: String,
  email: String,
  password: String (hashed),
  createdAt: Date,
  updatedAt: Date
}
```

Job Collection

```
{
  _id: ObjectId,
  user: ObjectId (ref: User),
  company: String,
  position: String,
  status: String (enum: ['applied', 'interview', 'offer', 'rejected', 'accepted']),
  appliedDate: Date,
  location: String,
  salary: String,
  jobUrl: String,
  notes: String,
  createdAt: Date,
  updatedAt: Date
}
```

API Endpoints

Method	Endpoint	Description	Authentication
POST	/api/auth/register	Register new user	Public
POST	/api/auth/login	Login user	Public

Method	Endpoint	Description	Authentication
GET	/api/auth/verify	Verify token	Private
GET	/api/jobs	Get all jobs	Private
GET	/api/jobs/stats	Get job statistics	Private
GET	/api/jobs/:id	Get job by ID	Private
POST	/api/jobs	Create new job	Private
PUT	/api/jobs/:id	Update job	Private
DELETE	/api/jobs/:id	Delete job	Private

Development Approach

Agile Methodology

The project was developed using an Agile approach with iterative development cycles:

1. **Planning Phase**
    - Requirements gathering
    - User stories creation
    - Technology selection
    - Architecture design
2. **Development Cycles**
    - Two-week sprints
    - Feature prioritization using MoSCoW method (Must have, Should have, Could have, Won't have)
    - Daily standups for progress tracking
    - End-of-sprint demos and retrospectives
3. **Testing Strategy**
    - Component testing for UI elements
    - API endpoint testing
    - User acceptance testing
    - Cross-device compatibility testing

Git Workflow

- Feature branch workflow
- Pull requests and code reviews
- Continuous integration checks
- Main branch protection

Code Organization

Frontend Structure

```
frontend/
├── src/
│   ├── components/    # Reusable UI components
│   ├── contexts/      # React Context providers
│   ├── pages/         # Page components
│   ├── services/      # API service functions
│   ├── utils/         # Utility functions
│   ├── App.jsx        # Main application component
│   └── main.jsx       # Application entry point
```

Backend Structure

```
backend/
├── config/            # Configuration files
├── middleware/        # Express middleware
├── models/            # Mongoose models
├── routes/            # API routes
└── server.js          # Server entry point
```

Development Timeline

Phase	Duration	Focus
Planning & Setup	2 weeks	Requirements, architecture, project setup
Core Features	4 weeks	Authentication, job CRUD operations
UI Development	3 weeks	Dashboard, forms, responsive design
Real-time Features	2 weeks	Notifications, Socket.IO integration
Testing & Refinement	3 weeks	Bug fixes, performance optimization
Deployment	1 week	Production deployment, documentation

Challenges Faced & Solutions

1. Cross-Device Notification Synchronization

Challenge: Notifications weren't synchronizing properly across multiple devices. When a user made changes on one device, notifications appeared only on that device, not on other logged-in devices.

Solution:

- Implemented a room-based Socket.IO system where each user joins a unique room identified by their user ID
- Modified the backend to emit notifications to specific user rooms instead of broadcasting globally
- Added connection management and reconnection logic to handle network issues

- Implemented local storage for persistence of notifications

Code Example:

```
// Backend: User-specific room setup
io.on('connection', (socket) => {
  if (socket.userId) {
    const userRoom = `user-${socket.userId}`;
    socket.join(userRoom);
  }
});

// Backend: Emitting to specific user room
io.to(`user-${req.user._id}`).emit('notification', {
  message: `Job application updated: ${job.position} at ${job.company}`,
  type: 'info'
});

// Frontend: Connection handling
const socketUrl = import.meta.env.VITE_API_URL
  ? new URL(import.meta.env.VITE_API_URL).origin
  : 'http://localhost:5000';

const socket = io(socketUrl, {
  auth: {
    token: localStorage.getItem('token'),
    userId: user.id
  }
});
```

2. Authentication and Security

Challenge: Implementing secure authentication while maintaining a good user experience, especially with real-time features.

Solution:

- Used JWT for stateless authentication
- Implemented token verification middleware
- Added password hashing with bcrypt
- Created socket.io middleware to authenticate socket connections
- Added proper error handling and validation

Code Example:

```
// Authentication middleware
const authMiddleware = async (req, res, next) => {
  try {
    const token = req.header('Authorization')?.replace('Bearer ', '');
    if (!token) {
      return res.status(401).json({ message: 'Authentication required' });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const user = await User.findById(decoded.id);

    if (!user) {
      return res.status(401).json({ message: 'Authentication failed' });
    }

    req.user = user;
    next();
  } catch (error) {
    res.status(401).json({ message: 'Authentication failed' });
  }
};
```

3. Responsive UI Design

Challenge: Creating a consistent user experience across devices (desktop, tablet, mobile) while maintaining functionality.

Solution:

- Adopted a mobile-first design approach
- Used TailwindCSS for responsive styling
- Created adaptive layouts with CSS Grid and Flexbox
- Implemented a collapsible sidebar for mobile
- Optimized notification display for small screens

Code Example:

```
// Responsive sidebar implementation
<div
  className={`w-64 bg-white shadow-lg fixed md:static inset-y-0 left-0 z-20 transform ${
    sidebarOpen ? 'translate-x-0' : '-translate-x-full md:translate-x-0'
  } transition-transform duration-300 ease-in-out`}
>
  {/* Sidebar content */}
</div>

// Mobile overlay when sidebar is open
{sidebarOpen && (
  <div
    className="fixed inset-0 bg-gray-800 bg-opacity-50 z-10 md:hidden"
    onClick={toggleSidebar}
  ></div>
)}
```

4. CORS and Production Deployment

Challenge: Handling Cross-Origin Resource Sharing (CORS) issues when deploying to production, particularly with Socket.IO connections.

Solution:

- Implemented flexible CORS configuration for different environments
- Created environment-specific settings (.env.production)
- Added proper error handling for connection failures
- Implemented fallbacks for Socket.IO transport methods

Code Example:

```
// CORS configuration
const allowedOrigins = [
  process.env.FRONTEND_URL,
  'https://job-tracker-elite.vercel.app',
  ...(process.env.NODE_ENV !== 'production' ? ['*'] : [])
];

const corsOptions = {
  origin: function (origin, callback) {
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      console.warn(`Origin ${origin} not allowed by CORS policy`);
      callback(null, true); // Allow anyway in production
    }
  },
  credentials: true
};

// Socket.IO configuration
const io = new Server(httpServer, {
  cors: corsOptions,
  transports: ['websocket', 'polling']
});
```

Learnings

Technical Insights

1. **Real-time Architecture**
    - Socket.IO room-based architecture is effective for user-specific notifications
    - Connection management requires careful handling of reconnection scenarios
    - Local storage synchronization provides a fallback for offline scenarios
2. **State Management**
    - React Context API was sufficient for this application's complexity level
    - Careful context design helped avoid unnecessary re-renders
    - Separating auth and notification contexts improved code organization
3. **Responsive Design**
    - Mobile-first approach simplified responsive design challenges

- TailwindCSS utility classes accelerated development
- Testing on actual devices revealed issues not apparent in browser emulation

4. **API Design**

   - Clear separation of concerns between route handlers improved maintainability
   - Input validation using Joi prevented many potential security issues
   - Consistent error response format simplified client-side error handling

Process Improvements

1. **Testing Strategy**

   - Early implementation of end-to-end testing would have caught integration issues sooner
   - More automated tests for API endpoints would have improved reliability
   - Cross-browser testing should have been more systematic

2. **Documentation**

   - API documentation should have been developed alongside the code
   - Better comments for complex logic would have improved maintainability
   - Creating user documentation earlier would have improved user testing

3. **Feature Prioritization**

   - Some features (like advanced filtering) could have been deprioritized
   - Core functionality like notification reliability should have been prioritized higher
   - User feedback should have been gathered earlier in the development process

Future Improvements

1. **Technical Enhancements**

   - Implement offline support using Service Workers
   - Add end-to-end encryption for sensitive job information
   - Optimize performance with code splitting and lazy loading
   - Implement comprehensive automated testing

2. **Feature Additions**

   - Calendar integration for interview scheduling
   - Email notifications for important updates
   - Document storage for resumes and cover letters
   - Analytics dashboard with insights on job search patterns
   - Mobile app versions for iOS and Android

3. **User Experience Improvements**

- Customizable dashboard layouts
- Dark mode support
- Accessibility improvements
- Onboarding tour for new users
- Import/export functionality for job data

This documentation was prepared by Mohd Faziel, June 2025.

Portfolio (https://www.faziel.me)