# **Busybee: Job Application Tracking System**

## **Technical Report**

*Team Members: Mason Luna, Mason Krause, Jacqueline Justice, Jason Moss*

## **Table of Contents**

1. Introduction
   * Problem Statement
   * Solution Overview
2. Technology
   * Technology Stack
   * Technology Adoption Experience
   * Development Challenges and Advantages
   * Technology Decisions
3. Design
   * System Architecture
   * Frontend Components
   * Backend Services
   * Database Schema
   * Data Models
   * User Interface
4. Deployment Instructions
   * Prerequisites
   * Repository Setup
   * Supabase Configuration
   * Frontend Deployment
   * Database Setup
   * Final Configuration
5. Known Issues
   * Functionality Gaps
   * UI/UX Issues
   * Performance Concerns
   * Security Considerations
   * Planned Improvements

## **1. Introduction**

### **Problem Statement**

The job search process in today's market is complex and multi-faceted. Job seekers typically use multiple platforms simultaneously (LinkedIn, Indeed, Glassdoor, Monster, company websites) to find opportunities, which makes tracking applications challenging. Without a centralized system to monitor application statuses, follow-up dates, and interview schedules, job seekers often lose track of critical information, miss opportunities for follow-ups, and lack data-driven insights about their job search effectiveness.

Students, recent graduates, and active job seekers in competitive markets need a systematic way to organize their job search activities. The current solutions are often fragmented, requiring job seekers to maintain separate spreadsheets, notes, or documents, which is both time-consuming and prone to error.

### **Solution Overview**

Busybee addresses these challenges by providing a comprehensive job application tracking system specifically designed for job seekers. The application serves as a centralized dashboard where users can:

1. Track all job applications across multiple platforms in a consolidated view
2. Manage application status updates throughout the hiring process
3. Filter and sort job applications based on various criteria
4. Organize applications into custom groups for better categorization
5. View statistics about application success rates and activity levels

The system empowers users to structure their job search in a deliberate, efficient, and data-driven way. By providing visibility into metrics such as application-to-interview and interview-to-offer ratios, Busybee helps job seekers identify areas for improvement in their application materials or interview skills.

The web-based application is built with a responsive design, making it accessible on both desktop and mobile devices, ensuring users can update their job search information regardless of their location or device preference.

## **2. Technology**

### **Technology Stack**

Busybee is built using modern web development technologies that promote scalability, maintainability, and user experience. The key technologies utilized include:

1. **Frontend:**
   * **React.js**: A JavaScript library for building user interfaces with component-based architecture
   * **TypeScript**: A strongly typed programming language that builds on JavaScript
   * **HTML/CSS**: Standard web technologies for structure and styling
2. **Backend:**
   * **Supabase**: An open-source Firebase alternative providing ready-made backend services
   * **PostgreSQL**: A powerful, open-source relational database system
3. **Authentication:**
   * **Supabase Auth**: Secure authentication system with email/password support and password reset capabilities
4. **Storage:**
   * **Supabase Storage**: Cloud storage solution for document management (PDF and DOCX files)
5. **Hosting/Deployment:**
   * **Vercel**: Cloud platform for frontend hosting and serverless functions
   * **GitHub**: Version control and source code management
6. **Development Tools:**
   * **Visual Studio Code**: Code editor for development
   * **Chrome DevTools**: Browser-based tools for testing and debugging
   * **Discord**: Team communication platform

The choice of these technologies was deliberate and strategic:

* **React.js** was selected for its component reusability, virtual DOM for performance, and extensive ecosystem of libraries and community support. It allows for the creation of a dynamic single-page application with efficient updates.
* **TypeScript** was implemented to enhance code quality through static typing, improving maintainability and reducing runtime errors during development.
* **Supabase** was chosen as a comprehensive backend solution that eliminates the need for custom server development. It provides pre-built authentication, database, and storage services that accelerate development.
* **PostgreSQL** was selected for its robustness, reliability, and advanced relational database features that support complex queries necessary for the application's use cases.
* **Vercel** was chosen for its seamless integration with GitHub for continuous deployment and its optimized hosting environment for React applications.

### **Technology Adoption Experience**

The learning curve for the technologies varied among team members:

* **React.js and TypeScript**: Team members with previous JavaScript experience found the transition to React relatively smooth, though TypeScript's strict typing initially slowed development. However, once familiar with the typing system, it improved code consistency and reduced errors.
* **Supabase**: This technology presented the steepest learning curve for the team. Understanding the Supabase ecosystem, including authentication flows, database access policies, and storage configuration, required significant study of documentation and experimentation. The team spent approximately two weeks becoming proficient with Supabase's API and security model.
* **Vercel**: Deployment with Vercel was straightforward and required minimal learning. Its GitHub integration made continuous deployment simple to set up and maintain.

### **Development Challenges and Advantages**

**Advantages:**

1. **React Component Reusability**: The component-based architecture of React allowed the team to create reusable UI elements that maintained consistency throughout the application.
2. **Supabase Integration**: Despite the learning curve, Supabase eliminated the need to develop custom backend services, significantly reducing development time.
3. **TypeScript Type Safety**: The static typing system caught many potential bugs during development, resulting in fewer runtime errors and more robust code.
4. **Vercel Deployment Simplicity**: The automated deployment pipeline streamlined the release process, allowing the team to focus on feature development.

**Challenges:**

1. **State Management Complexity**: Managing application state across multiple components proved challenging, particularly for the job entry forms and filtering mechanisms.
2. **Supabase Row-Level Security**: Configuring proper security policies in Supabase to ensure data privacy and access control required careful planning and testing.
3. **TypeScript Learning Curve**: The strict typing system initially slowed development as team members adapted to explicitly defining types and interfaces.
4. **Mobile Responsiveness**: Ensuring a consistent user experience across desktop and mobile required additional design considerations and CSS adjustments.

### **Technology Decisions**

The team originally considered a more traditional MERN (MongoDB, Express.js, React, Node.js) stack but opted for a Supabase/React combination for several reasons:

1. **Development Speed**: Supabase's ready-made backend services accelerated development by eliminating the need for custom API routes and authentication systems.
2. **PostgreSQL vs. MongoDB**: The structured nature of the application data, with clear relationships between entities (users, jobs, groups), made a relational database like PostgreSQL more appropriate than MongoDB's document-oriented approach.
3. **Maintenance Considerations**: Supabase's managed services reduced the operational burden of maintaining separate backend servers and databases.
4. **Authentication Security**: Using Supabase's battle-tested authentication system provided better security than implementing a custom solution given the team's time constraints.

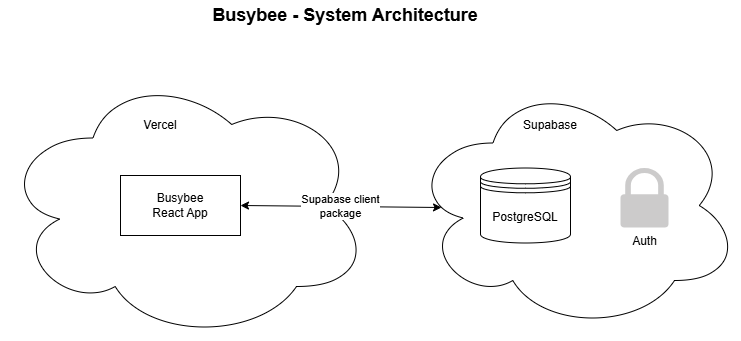
In retrospect, the decision to use Supabase over a traditional MERN stack was correct, as it allowed the team to focus on frontend functionality and user experience rather than backend infrastructure. The time saved on backend development was redirected to improving the dashboard's features and reliability.

## **3. Design**

### **System Architecture**

Busybee follows a client-server architecture with a clear separation between the frontend React application and the Supabase backend services. Figure 1 illustrates the high-level system architecture.

*Figure 1: Busybee System Architecture*

**

The application is structured as follows:

1. **Client Layer**: React TypeScript application with component-based UI
2. **Service Layer**: API services that communicate with Supabase
3. **Data Layer**: PostgreSQL database and Supabase storage

The system employs a unidirectional data flow pattern where:

* User actions trigger service calls to the backend
* Services update the Supabase database
* Component state is updated with the new data
* UI re-renders based on the updated state

### **Frontend Components**

The frontend is organized into several key modules:

#### **Core Pages**

* **LandingPage**: Entry point that checks authentication and directs users
* **LoginPage**: User authentication interface
* **SignUpPage**: New user registration interface
* **MainDashboard**: Central dashboard with navigation to other sections
* **StaticPages**: About, FAQ, and Contact information pages

#### **Dashboard Components**

* **SummaryDashboard**: Overview of recent activity and statistics
* **JobsDashboard**: Management of job applications
* **StatDashboard**: Visualization of job search statistics
* **GroupDashboard**: Organization of jobs into custom groups
* **DocumentsDashboard**: Document management for resumes and cover letters

#### **UI Components**

* **Header**: Navigation and user information display
* **Footer**: Site navigation and resources
* **EasyNav**: Left-side navigation between dashboard sections
* **JobList**: Sortable, filterable list of job applications
* **Statistics**: Various representations of job search metrics
* **GroupTabs**: Tab-based interface for group management
* **ErrorMessage**: Standardized error display component

#### **Form Components**

* **LoginForm**: User authentication form
* **SignUpForm**: User registration form
* **JobForm**: Create/edit job application details
* **GroupForm**: Create/edit group information

### **Backend Services**

The backend functionality is implemented through several service modules:

#### **Authentication Services**

* **supabaseService.tsx**: Handles user login, registration, password reset

export async function login(email: string, password: string){

try{

const { data, error } = await supabase.auth.signInWithPassword({

email: email,

password: password

});

// ...authentication logic

}

catch(err) {

console.log('Unexpected error sending login request... ', err);

}

return null;

}

#### **Data Services**

* **supabaseService.tsx**: CRUD operations for jobs, groups, and documents
* **objectConversionService.tsx**: Data transformation between frontend and backend formats
* **statsCalculationService.tsx**: Computation of job search statistics

Example of job creation service:

export async function createJob(jobInsertDto: JobInsertDto) {

try {

const { data, error } = await supabase

.from('jobs')

.insert([jobInsertDto])

.select();

// ... error handling and response processing

} catch (error) {

console.error("An unexpected error occurred while creating the job:", error);

}

return null;

}

#### **Context Providers**

* **DashboardDataProvider**: Central state management for dashboard data
* **useDashboardContext**: Custom hook for accessing dashboard state

### **Database Schema**

The database consists of the following tables:

1. **users**
   * user\_id: uuid (Primary Key)
   * first\_name: string
   * last\_name: string
   * email: string
   * date\_created: timestamp
   * last\_accessed: timestamp
   * monthly\_goal: integer
2. **jobs**
   * job\_id: integer (Primary Key)
   * user\_id: uuid (Foreign Key)
   * company\_name: string
   * job\_title: string
   * remote: boolean
   * job\_city: string (nullable)
   * job\_state: string (nullable)
   * job\_country: string (nullable)
   * date\_posted: date (nullable)
   * date\_applied: date (nullable)
   * platform: string (nullable)
   * estimated\_annual\_salary: integer (nullable)
   * status\_id: integer (Foreign Key)
   * notes: string (nullable)
3. **status**
   * status\_id: integer (Primary Key)
   * status\_name: string
4. **groups**
   * group\_id: integer (Primary Key)
   * user\_id: uuid (Foreign Key)
   * group\_name: string
   * group\_start\_date: date
   * group\_end\_date: date (nullable)
5. **group\_jobs** (Junction Table)
   * group\_job\_id: integer (Primary Key)
   * group\_id: integer (Foreign Key)
   * job\_id: integer (Foreign Key)
6. **documents**
   * document\_id: integer (Primary Key)
   * user\_id: uuid (Foreign Key)
   * filename: string
   * file\_path: string
   * upload\_date: timestamp
   * document\_type: string

The database implements row-level security policies in Supabase to ensure users can only access their own data.

### **Data Models**

The application uses TypeScript interfaces to define the structure of data:

// User model

export interface User {

user\_id: string;

first\_name: string;

last\_name: string;

email: string;

date\_created: string;

last\_accessed: string;

monthly\_goal: number;

}

// Job model

export interface Job {

job\_id: number;

company\_name: string;

job\_title: string;

remote: boolean;

job\_city: string | null;

job\_state: string | null;

job\_country: string | null;

date\_posted: string | null;

date\_applied: string | null;

platform: string | null;

estimated\_annual\_salary: number | null;

status\_id: number;

notes: string | null;

user\_id: string;

}

// Job Data Transfer Object

export interface JobDto {

job\_id: number;

company\_name: string;

job\_title: string;

remote: boolean;

job\_city: string | null;

job\_state: string | null;

job\_country: string | null;

date\_posted: string | null;

date\_applied: string | null;

platform: string | null;

estimated\_annual\_salary: number | null;

status\_name: string;

notes: string | null;

user\_id: string;

}

// Group model

export interface Group {

group\_id: number;

group\_name: string;

group\_start\_date: string;

group\_end\_date: string | null;

user\_id: string;

}

// GroupJob junction model

export interface GroupJob {

group\_job\_id: number;

group\_id: number;

job\_id: number;

}

// Statistics model

export interface UserStats {

totalApps: number;

totalInterviews: number;

totalOffers: number;

appsPerMonth: number;

appsThisMonth: number;

appsNeededForGoal: number;

interviewRate: string;

offerRate: string;

interviewSuccessRate: string;

}

### **User Interface**

The user interface is designed to be intuitive and accessible. Key screens include:

1. **Login Screen**: Authentication interface with email/password inputs and "Forgot Password" functionality
2. **Dashboard Home**: Overview with quick access to recently added jobs, documents, and key statistics
3. **Jobs Dashboard**: Comprehensive job listing with filtering, sorting, and detailed view options
4. **Statistics Dashboard**: Visual representations of application metrics and success rates
5. **Groups Dashboard**: Tab-based interface for organizing jobs into custom categories

*Figure 2: Dashboard Layout*

The UI implements a consistent layout with:

* Header with user information and sign-out button
* Left sidebar for navigation between dashboard sections
* Main content area with section-specific functionality
* Footer with links to resources and information pages

The application uses a responsive design approach to ensure usability on both desktop and mobile devices.

## **4. Deployment Instructions**

### **Prerequisites**

Before deploying Busybee, ensure you have:

1. **GitHub Account**: For repository cloning and version control
2. **Supabase Account**: For backend services (Free tier available)
3. **Vercel Account**: For frontend hosting (Free tier available)
4. **Node.js and npm**: Latest stable version installed locally

### **Repository Setup**

Clone the Busybee repository from GitHub:  
  
 git clone https://github.com/Masonluna/BusyBee.git

cd busybee

Install dependencies:  
  
 npm install

### **Supabase Configuration**

1. Create a new Supabase project from the Supabase dashboard.

Set up database tables according to the schema defined in Section 3. You can use the following SQL to create the necessary tables:  
  
 -- Create tables

CREATE TABLE public.users (

user\_id UUID PRIMARY KEY REFERENCES auth.users(id),

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE,

date\_created TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

last\_accessed TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

monthly\_goal INTEGER DEFAULT 10

);

CREATE TABLE public.status (

status\_id SERIAL PRIMARY KEY,

status\_name TEXT NOT NULL UNIQUE

);

CREATE TABLE public.jobs (

job\_id SERIAL PRIMARY KEY,

user\_id UUID REFERENCES public.users(user\_id) NOT NULL,

company\_name TEXT NOT NULL,

job\_title TEXT NOT NULL,

remote BOOLEAN DEFAULT FALSE,

job\_city TEXT,

job\_state TEXT,

job\_country TEXT,

date\_posted DATE,

date\_applied DATE,

platform TEXT,

estimated\_annual\_salary INTEGER,

status\_id INTEGER REFERENCES public.status(status\_id) NOT NULL,

notes TEXT

);

CREATE TABLE public.groups (

group\_id SERIAL PRIMARY KEY,

user\_id UUID REFERENCES public.users(user\_id) NOT NULL,

group\_name TEXT NOT NULL,

group\_start\_date DATE NOT NULL,

group\_end\_date DATE

);

CREATE TABLE public.group\_jobs (

group\_job\_id SERIAL PRIMARY KEY,

group\_id INTEGER REFERENCES public.groups(group\_id) NOT NULL,

job\_id INTEGER REFERENCES public.jobs(job\_id) NOT NULL,

UNIQUE(group\_id, job\_id)

);

CREATE TABLE public.documents (

document\_id SERIAL PRIMARY KEY,

user\_id UUID REFERENCES public.users(user\_id) NOT NULL,

filename TEXT NOT NULL,

file\_path TEXT NOT NULL,

upload\_date TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

document\_type TEXT NOT NULL

);

Populate the status table with predefined values:  
  
 INSERT INTO public.status (status\_name) VALUES

('Applied'),

('Assessment'),

('Interview'),

('Offer'),

('Counter Offer'),

('Rejected'),

('No Response'),

('Offer Accepted');

Configure Row Level Security (RLS) policies:  
  
 -- Enable RLS on all tables

ALTER TABLE public.users ENABLE ROW LEVEL SECURITY;

ALTER TABLE public.jobs ENABLE ROW LEVEL SECURITY;

ALTER TABLE public.groups ENABLE ROW LEVEL SECURITY;

ALTER TABLE public.group\_jobs ENABLE ROW LEVEL SECURITY;

ALTER TABLE public.documents ENABLE ROW LEVEL SECURITY;

-- Create policies for users table

CREATE POLICY "Users can view their own data" ON public.users

FOR SELECT USING (auth.uid() = user\_id);

CREATE POLICY "Users can update their own data" ON public.users

FOR UPDATE USING (auth.uid() = user\_id);

-- Create policies for jobs table

CREATE POLICY "Users can view their own jobs" ON public.jobs

FOR SELECT USING (auth.uid() = user\_id);

CREATE POLICY "Users can insert their own jobs" ON public.jobs

FOR INSERT WITH CHECK (auth.uid() = user\_id);

CREATE POLICY "Users can update their own jobs" ON public.jobs

FOR UPDATE USING (auth.uid() = user\_id);

CREATE POLICY "Users can delete their own jobs" ON public.jobs

FOR DELETE USING (auth.uid() = user\_id);

-- Similar policies for groups, group\_jobs, and documents tables

1. Configure authentication settings:
   * Enable email/password sign-up
   * Configure email templates for password reset
   * Set redirect URLs for password reset to point to your frontend application
2. Get your Supabase URL and anon key from the project settings.

### **Frontend Deployment**

Create a .env file in the project root with your Supabase credentials:  
  
 REACT\_APP\_SUPABASE\_URL=https://your-supabase-project.supabase.co

REACT\_APP\_SUPABASE\_ANON\_KEY=your-anon-key

Configure the Supabase client in src/utils/supabase.ts:  
  
 import { createClient } from '@supabase/supabase-js';

const supabaseUrl = process.env.REACT\_APP\_SUPABASE\_URL!;

const supabaseAnonKey = process.env.REACT\_APP\_SUPABASE\_ANON\_KEY!;

const supabase = createClient(supabaseUrl, supabaseAnonKey);

export default supabase;

1. Deploy the frontend to Vercel:  
   * Connect your GitHub repository to Vercel
   * Configure the build settings:
     + Build Command: npm run build
     + Output Directory: build
   * Add environment variables for Supabase credentials
   * Deploy the application

### **Database Setup**

After deploying the frontend, create a trigger to sync new auth users with the public users table:  
  
 -- Create a function to handle new user registration

CREATE OR REPLACE FUNCTION public.handle\_new\_user()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO public.users (user\_id, first\_name, last\_name, email)

VALUES (new.id, '', '', new.email);

RETURN new;

END;

$$ LANGUAGE plpgsql SECURITY DEFINER;

-- Create a trigger that calls this function when a new user signs up

CREATE TRIGGER on\_auth\_user\_created

AFTER INSERT ON auth.users

FOR EACH ROW EXECUTE FUNCTION public.handle\_new\_user();

1. Test the authentication flow by creating a new user and verifying that:  
   * The user is created in the auth.users table
   * The trigger creates a corresponding record in the public.users table

### **Final Configuration**

1. Configure your application domain in Supabase:  
   * Go to Authentication > Settings
   * Add your Vercel-deployed domain to the Site URL and Additional Redirect URLs
2. Test the full application by:  
   * Registering a new user
   * Creating job entries
   * Uploading documents
   * Creating and managing groups
   * Verifying that statistics are calculated correctly
3. Monitor application logs and database performance as users begin to use the system.

## **5. Known Issues**

### **Functionality Gaps**

1. **Document Management**: In the initial project design, we planned to allow users to upload their documents (cover letters and resumes) along with their job applications or independently. Unfortunately, due to the time constraints, we were unable to implement the functionality for handling document uploads.
2. **Limited Mobile Experience**: Although the application is responsive, the mobile interface requires optimization for smaller screens, particularly for the job entry form and statistics dashboards. Dedicated mobile-specific layouts should be developed to improve usability on smartphones.

### **UI/UX Issues**

1. **Form Validation Feedback**: The current form validation provides basic error messages, but feedback could be improved with inline validation and more descriptive guidance. The team plans to implement real-time validation with visual indicators as users complete forms.
2. **Dashboard Loading States**: The application lacks proper loading indicators when fetching data, which can make the interface appear unresponsive. Adding loaders and progress indicators would improve the perceived performance.
3. **Accessibility Compliance**: The application has not been fully tested for accessibility compliance with WCAG standards. Screen reader compatibility and keyboard navigation need improvement to ensure the application is usable by people with disabilities.

### **Performance Concerns**

1. **Inefficient Query Patterns**: Some database queries retrieve more data than necessary, particularly when loading the dashboard components. Implementing pagination and more specific queries would reduce data transfer and improve load times.
2. **State Management Complexity**: The current state management approach uses context providers, which can lead to unnecessary re-renders. Implementing a more optimized state management solution, such as Redux Toolkit or Recoil, would improve performance for complex state interactions.

### **Security Considerations**

1. **Password Policy Enforcement**: The current password requirements (minimum 6 characters) are insufficient by modern security standards. The team plans to enhance password policies to require stronger passwords with a combination of character types.
2. **Incomplete Input Sanitization**: While basic input validation is implemented, more comprehensive sanitization is needed to protect against injection attacks, particularly in free-text fields like notes. The team will implement additional server-side validation and sanitization.
3. **Session Management**: The current session management relies entirely on Supabase's default behavior. Implementing additional session security features, such as device fingerprinting and suspicious login detection, would enhance security.

### **Planned Improvements**

1. **Enhanced Analytics**: The team plans to expand the statistics dashboard with more detailed metrics and predictive analytics to help users optimize their job search strategy based on historical patterns.
2. **Email Notifications**: Implementing an email notification system would help users stay on top of follow-up deadlines and application status changes.
3. **Integration with Job Platforms**: Future development would include API integrations with popular job platforms to automatically import job listings and track application status updates.
4. **Interview Preparation Tools**: Adding features to help users prepare for interviews, such as company research tools, common question repositories, and interview scheduling, would extend the application's value throughout the job search lifecycle.
5. **Collaborative Features**: Implementing features that allow career counselors or mentors to provide feedback on a user's job search progress would enhance the application's utility for educational institutions and career services departments.
6. **Sortable Job Entries**: Adding a sort by feature would greatly improve the user experience as they navigate through the jobs page. The feature should allow for sorting by any viewable entry in the job application submission form.

The team is committed to addressing these known issues in future updates, prioritizing those that directly impact user experience and core functionality. The modular architecture of the application makes it well-suited for incremental improvements without requiring significant refactoring.