TechFlow

Full-Stack Contact Management Platform

Project Report

Field	Detail
Project	TechFlow – Contact Management Platform
Author	Akshay Raj
Assignment	Web Development Internship
Version	1.0.0
Repository	https://github.com/AkshayRaj367/Internship-project
Live Client	https://techflow367.vercel.app
Live API	https://techflow-server-internship.onrender.com/api
Date	2025

Table of Contents

1. Introduction

TechFlow is a full-stack web application built to demonstrate modern enterprise-grade development practices. It serves as a contact management platform where users can submit contact inquiries through a public-facing form, create an account using email/password or Google OAuth, view and manage their submitted contacts in a real-time dashboard, track contact status through a lifecycle (New → Read → Replied → Archived), and export contacts to CSV for external processing.

The application showcases a complete development lifecycle – from a responsive frontend with smooth animations to a secure, scalable backend with real-time WebSocket communication, all deployed across cloud platforms using industry-standard practices.

2. Objectives

#	Objective	Status
1	Build a responsive React frontend	✅ Done
2	Implement secure REST API backend	✅ Done
3	Dual authentication (email + Google)	✅ Done
4	Real-time updates via WebSockets	✅ Done
5	Per-user data isolation	✅ Done
6	CRUD operations with search/filter	✅ Done
7	CSV export functionality	✅ Done
8	Production deployment	✅ Done
9	Comprehensive security middleware	✅ Done
10	Automated setup for developers	✅ Done

3. Technology Choices & Justification

Frontend Technologies

- **React 18:** Industry-standard UI library with concurrent features and excellent developer tooling
- **TypeScript:** Adds static typing to catch bugs at compile time and provides self-documenting code
- **Vite 4.5:** Ultra-fast build tool with Hot Module Replacement, significantly faster than Webpack
- **Tailwind CSS:** Utility-first CSS framework that eliminates context-switching between HTML and CSS
- **Framer Motion:** Production-ready animation library with declarative API and physics-based transitions
- **Socket.IO Client:** Reliable WebSocket communication with automatic reconnection

Backend Technologies

- **Express 4.18:** Most popular Node.js framework with mature ecosystem and middleware-based architecture
- **Mongoose 8:** Elegant MongoDB ODM with schema validation and TypeScript support
- **Passport.js:** Modular authentication library supporting 500+ strategies for Google OAuth and JWT
- **Socket.IO:** Battle-tested WebSocket library with rooms, namespaces, and automatic reconnection
- **Winston:** Enterprise logging library with multiple transports and log levels
- **Helmet:** Sets various HTTP headers to protect against web vulnerabilities

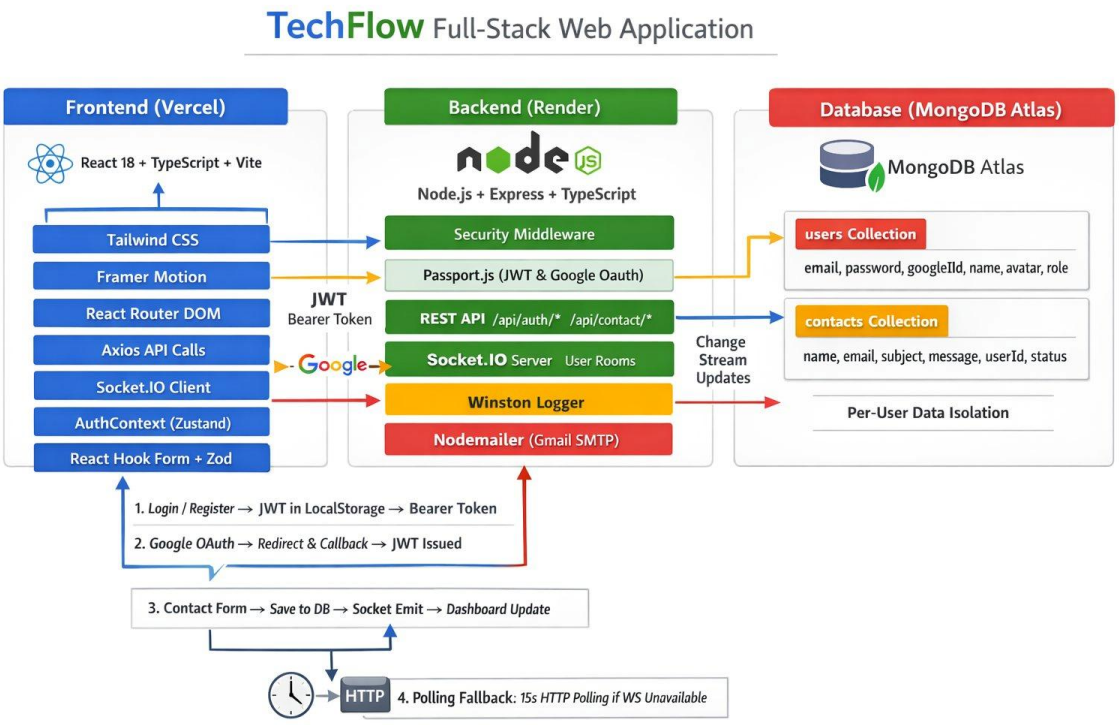
Database & Deployment

- **MongoDB Atlas:** Cloud-hosted NoSQL database with built-in replication and Change Streams
- **Vercel:** Optimized for React/Vite with global CDN and automatic deployments
- **Render:** Supports persistent Node.js processes required for Socket.IO

4. System Architecture

The application follows a three-tier architecture consisting of:

Figure 1: Complete System Architecture



Presentation Tier

React Single Page Application hosted on Vercel (CDN). Includes pages for Home, Login, Register, and Dashboard. Communicates via REST API using Axios and WebSocket using Socket.IO.

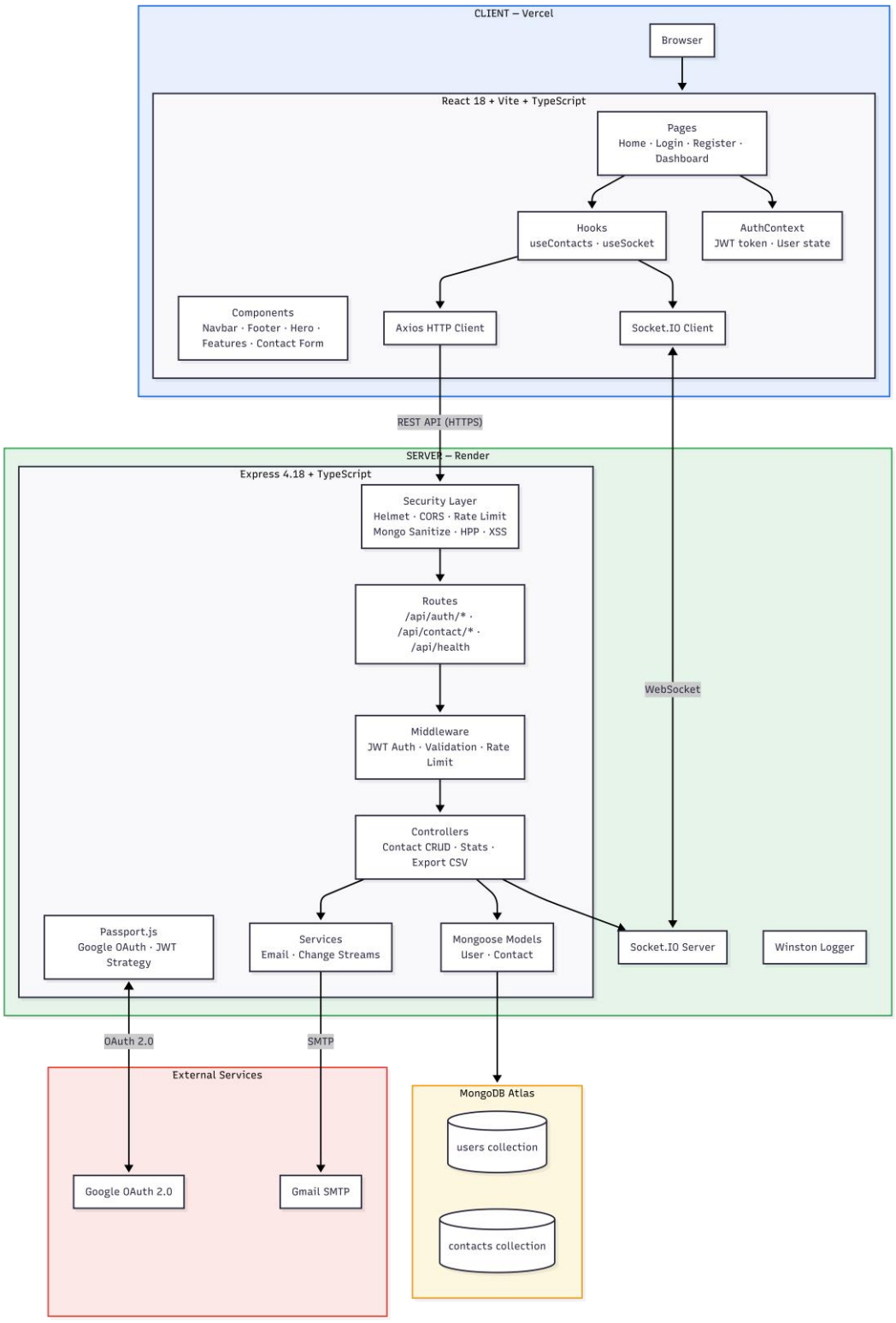
Application Tier

Express API server hosted on Render. Features a security middleware stack, route handlers, controllers, and models. Includes Socket.IO server and MongoDB Change Stream watcher for real-time updates.

Data Tier

MongoDB Atlas (Cloud M0 Free) with two main collections: users and contacts. Provides secure data storage with proper indexing.

Figure 2: Detailed System Components



Request Lifecycle

- Browser makes HTTP request to Express server
- Request passes through security middleware (Helmet, CORS, Rate Limiter, Mongo Sanitize)
- Route handler matches path and invokes middleware chain
- Auth middleware extracts and verifies JWT from Authorization header
- Validation middleware checks request body/params against Joi schemas
- Controller processes business logic and interacts with Mongoose models
- Response sent back; Socket.IO event emitted if needed for real-time updates

5. Feature Breakdown

5.1 Landing Page (Public)

The home page consists of three animated sections:

- **Hero Section:** Welcoming headline with call-to-action buttons and Framer Motion entrance animations
- **Features Section:** Card grid showcasing key capabilities with scroll-triggered animations
- **Contact Form:** Interactive form with subject dropdown, real-time Zod validation, and toast notifications

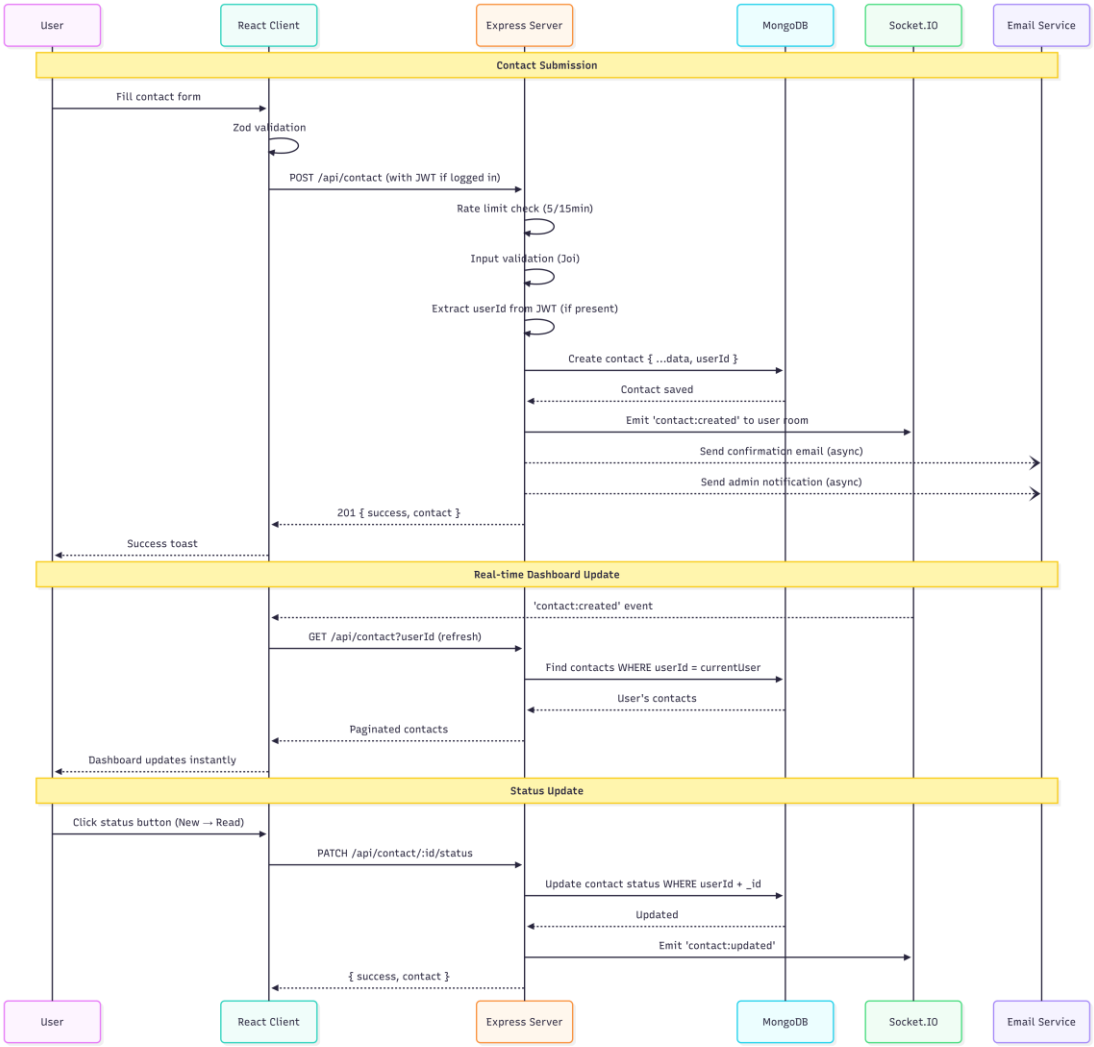
5.2 Authentication

- **Registration:** Name, email, password form with validation. Passwords hashed with bcrypt
- **Login:** Email/password authentication returning a JWT token
- **Google OAuth:** One-click Google sign-in using Passport's Google OAuth 2.0 strategy
- **Route Protection:** ProtectedRoute component wraps Dashboard, redirecting unauthenticated users

5.3 Dashboard (Authenticated)

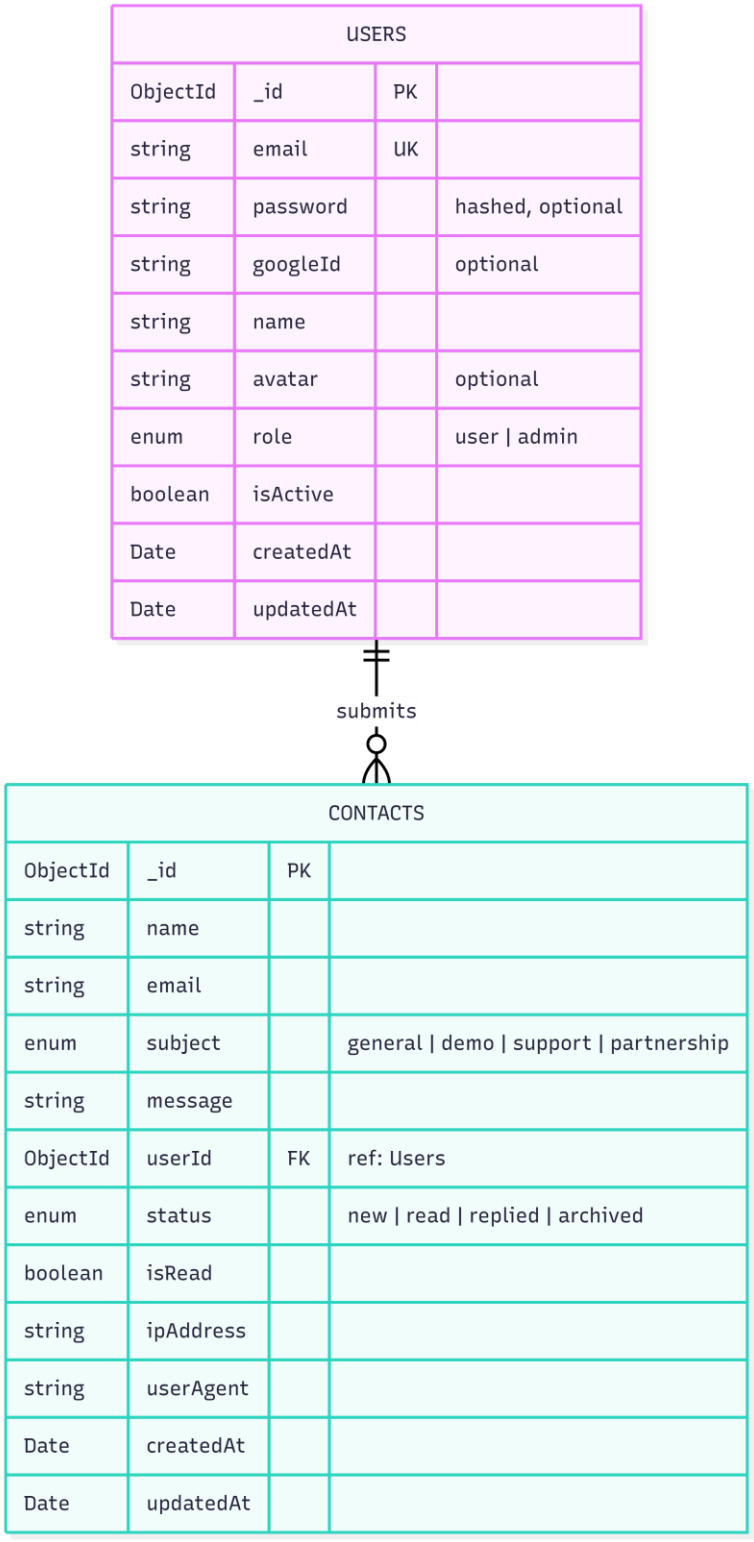
- **Statistics Cards:** Four cards showing Total, New, Read, and Replied contact counts with animated counters
- **Search & Filter:** Text search across contact name, email, and message body with status filter dropdown
- **Contact Table:** Paginated list of contacts with status badges, timestamps, and action buttons
- **Status Management:** Click to cycle through status: New → Read → Replied → Archived
- **CSV Export:** One-click download of all contacts as a spreadsheet-compatible CSV file
- **Real-Time Updates:** Dashboard updates instantly without page refresh via Socket.IO

Figure 3: Contact Lifecycle Flow



6. Database Design

Figure 4: Database Schema



Users Collection

The users collection stores authentication and profile information:

- _id: ObjectId (Primary key)
- email: String (Unique, lowercase, indexed)
- password: String (bcrypt hash, excluded from queries by default)
- googleId: String (Google OAuth identifier, optional)
- name: String (Display name)
- role: String (user | admin, default: user)
- createdAt, updatedAt: Date (Auto-managed by Mongoose)

Contacts Collection

The contacts collection stores all contact form submissions:

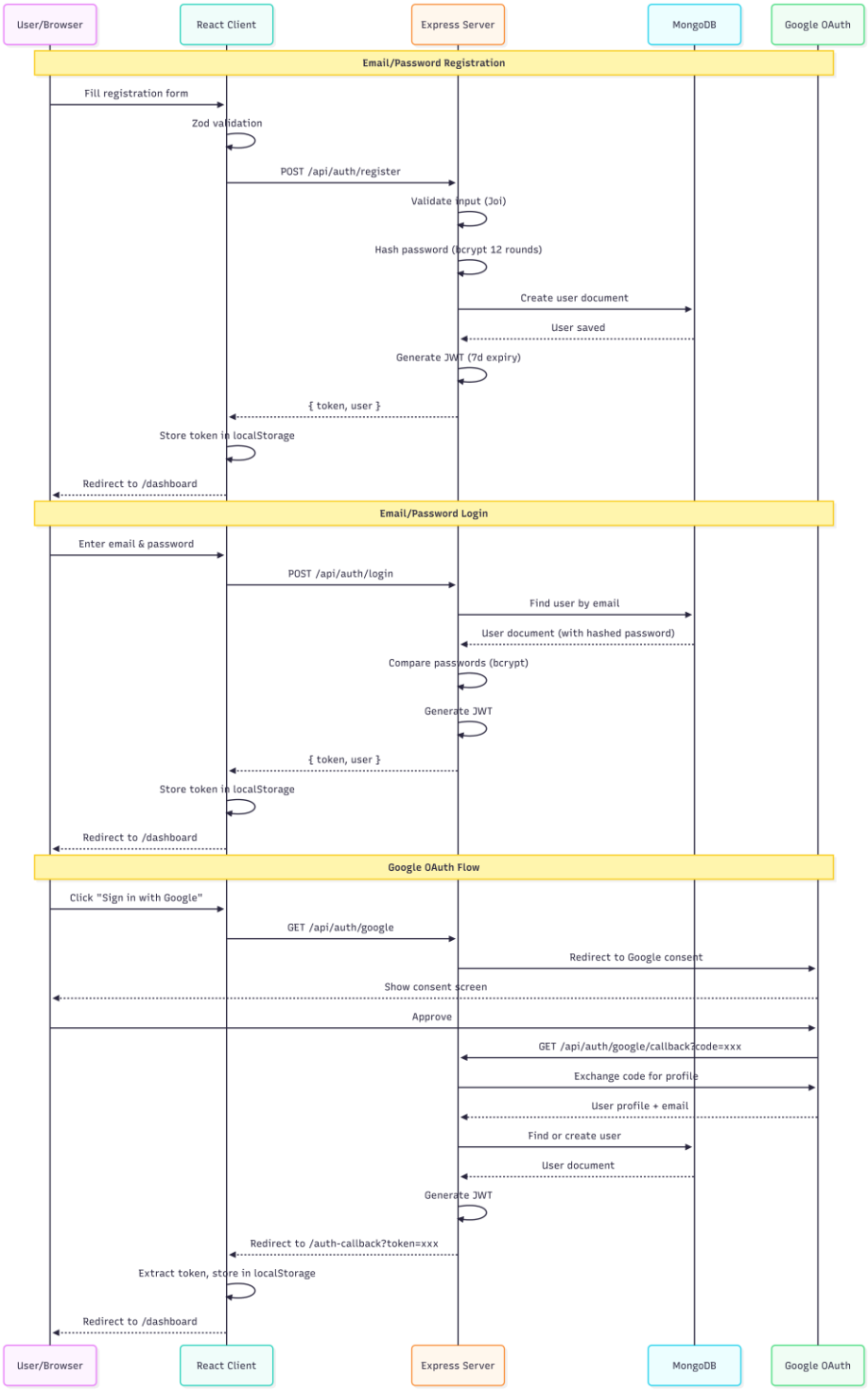
- _id: ObjectId (Primary key)
- name, email, subject, message: String (Contact details)
- userId: ObjectId (Reference to Users collection)
- status: String (new | read | replied | archived)
- ipAddress, userAgent: String (Audit information)
- createdAt, updatedAt: Date (Auto-managed)

Data Isolation

Every database query in the Contact controller is scoped to the authenticated user's ID. All CRUD queries include { userId: req.user._id } filter, ensuring User A can never see User B's contacts, even if they guess the contact's _id.

7. Authentication System

Figure 5: Authentication Flow



JWT Token Flow

Registration/Login process:

- Request received → Validate credentials → Hash/Compare password
- Generate JWT token → Return { token, user } to client

Protected Endpoints process:

- Extract Bearer token → Verify JWT signature → Decode userId
- Fetch user from database → Attach to req.user → Continue to controller

Token Configuration

- Algorithm: HS256 (HMAC-SHA256)
- Expiry: 7 days (configurable via JWT_EXPIRE)
- Payload: { userId, email, role }
- Storage: Client localStorage
- Transmission: Authorization: Bearer <token> header

Google OAuth Flow

- Client navigates to /api/auth/google
- Server redirects to Google consent screen
- User approves → Google sends authorization code to callback URL
- Server exchanges code for user profile (email, name, photo)
- Server finds or creates user in MongoDB (by googleId or email)
- Server generates JWT token and redirects to client

8. Real-Time Communication

Socket.IO Architecture

Server Side:

- Initializes Socket.IO with CORS matching Express CORS config
- On connection: client joins 'dashboard' room and 'user:<userId>' room
- emitToUser(userId, event, data) – sends to specific user's room
- emitToDashboard(event, data) – broadcasts to all connected dashboards

Client Side:

- useSocket() hook manages connection lifecycle
- Joins user-specific room on authentication
- Listens for contact:created, contact:updated, contact:deleted events
- Triggers data refresh on receipt

Fallback Mechanism:

useContacts() hook starts 15-second HTTP polling when socket is disconnected. Polling stops automatically when socket reconnects. This ensures data stays current even during Render free tier cold starts.

Why User-Specific Rooms?

Without rooms, all users would receive all contact events. By having each user join user:<theirId>, the server only emits events for contacts belonging to that user. This maintains data isolation in real-time, not just in REST API queries.

9. Security Implementation

Middleware Stack

#	Middleware	Purpose
1	Helmet	Sets security HTTP headers (X-Frame-Options, X-Content-Type, etc.)
2	CORS	Restricts origins to configured CLIENT_URL
3	Rate Limiter	General: 100 requests per 15 minutes per IP
4	Auth Rate Limiter	Login/Register: 20 requests per 15 minutes per IP
5	Contact Rate Limiter	Contact form: 5 submissions per 15 minutes per IP
6	express-mongo-sanitize	Prevents NoSQL injection by stripping $ and . from inputs
7	HPP	Prevents HTTP Parameter Pollution attacks
8	XSS-clean	Sanitizes user input to prevent Cross-Site Scripting
9	compression	Gzip compression for response bodies

Password Security

- Hashing: bcryptjs with 12 salt rounds
- Passwords stored as irreversible hashes
- Password field excluded from all queries by default (select: false)
- Comparison done via user.comparePassword() using bcrypt.compare

Input Validation

- Server-side: Joi schemas validate all incoming request bodies and parameters
- Client-side: Zod schemas provide real-time form validation with React Hook Form
- Both layers ensure defense-in-depth against malformed input

10. API Documentation

Base URLs

- Local: http://localhost:5000/api
- Production: https://techflow-server-internship.onrender.com/api

Authentication Endpoints

- POST /api/auth/register – Create new user account
- POST /api/auth/login – Authenticate and receive JWT token
- GET /api/auth/google – Initiate Google OAuth flow
- GET /api/auth/me [JWT Required] – Get current user profile
- POST /api/auth/logout – Clear session

Contact Endpoints

- POST /api/contact [Optional Auth] – Submit contact form
- GET /api/contact [JWT Required] – Get all user's contacts with pagination and filters
- GET /api/contact/stats [JWT Required] – Get contact statistics
- GET /api/contact/export [JWT Required] – Export contacts as CSV file
- GET /api/contact/:id [JWT Required] – Get specific contact
- PATCH /api/contact/:id/status [JWT Required] – Update contact status
- DELETE /api/contact/:id [JWT Required] – Delete contact

Error Response Format

All errors follow a consistent JSON format: { success: false, error: { message: 'Description', code: 'ERROR_CODE' } }

11. Frontend Implementation

Component Architecture

The application follows a hierarchical component structure:

- App.tsx – Root component with routing and layout
- Routes: Home (/), Login (/login), Register (/register), Dashboard (/dashboard)
- Home sections: Hero, Features, Contact form
- Shared components: Navbar, Footer, ProtectedRoute

State Management

- **AuthContext:** Manages user authentication state, JWT token, and auth functions
- **useContacts Hook:** Manages contact data fetching, real-time updates, and CSV export
- **useSocket Hook:** Manages Socket.IO connection lifecycle and event listening

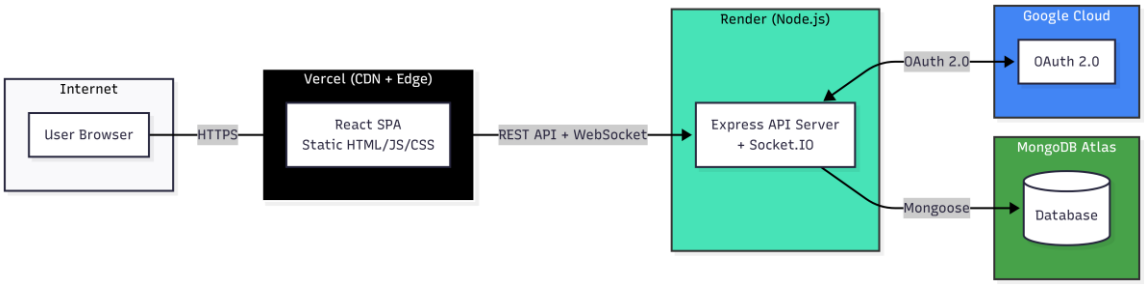
UI Design System

The application uses a Google-inspired design system with:

- Primary blue (#1a73e8) for CTAs and links
- Green (#34a853) for success states
- Yellow/amber (#f59e0b) for warnings
- Red (#ea4335) for errors and destructive actions
- Clean white backgrounds with subtle shadows and rounded corners

12. Deployment Strategy

Figure 6: Deployment Architecture



Deployment Architecture

The application uses a dual-platform deployment strategy:

- GitHub Repository: Source code versioned with Git
- Vercel (Frontend): Auto-deploys from main branch, builds with Vite, serves as static SPA with global CDN
- Render (Backend): Auto-deploys from main branch, runs as persistent Node.js process enabling Socket.IO

Why Render for Backend?

Vercel uses serverless functions where each request spins up a fresh function instance. This breaks Socket.IO because WebSockets require a persistent, long-lived server process. Render provides a traditional Node.js hosting environment where the server stays running, enabling real-time WebSocket connections.

Production Build Process

Server Build:

- npm install --include=dev (Install all dependencies including TypeScript)
- npm run build (Compile TypeScript to dist/ directory)
- npm start (Run compiled JavaScript with module-alias support)

Client Build:

- npm install (Install dependencies)
- npm run build (TypeScript type-check + Vite build to static assets)

Environment Variables

Key production environment variables:

- MONGODB_URI – MongoDB Atlas connection string
- JWT_SECRET, SESSION_SECRET – Secure random strings
- CLIENT_URL / CORS_ORIGIN – Vercel client URL
- VITE_API_URL – Render server URL + /api
- GOOGLE_CALLBACK_URL – Points to Render server callback

13. How to Use the Application

For End Users

- **Step 1:** Visit https://techflow367.vercel.app
- **Step 2:** Browse the landing page and explore Hero, Features, and Contact sections
- **Step 3:** Submit a contact inquiry using the form (works without an account)
- **Step 4:** Create an account by clicking 'Get Started' or 'Sign Up' (email/password or Google)
- **Step 5:** Access the Dashboard – after login, you're redirected to your personal dashboard
- **Step 6:** View contacts – all submitted contacts appear in a table with status badges
- **Step 7:** Search & filter – use search bar and status dropdown to find specific contacts
- **Step 8:** Update status – click status badge to change state (New → Read → Replied → Archived)
- **Step 9:** Export to CSV – click export button to download all contacts as spreadsheet
- **Step 10:** Logout – click user avatar or logout button in navbar

For Developers

- **Step 1:** Clone repository: git clone https://github.com/AkshayRaj367/Intership-project.git
- **Step 2:** Run setup wizard: node setup.js (handles all configuration automatically)
- **Step 3:** API testing: Use Postman or curl against http://localhost:5000/api
- **Step 4:** Health check: GET http://localhost:5000/api/health to verify server and database status

14. Project Structure

The project is organized into two main directories:

Root Level

- setup.js – Automated setup wizard
- README.md – Project documentation
- docs/ – Architecture diagrams and this report

Client Directory (Frontend)

- src/App.tsx – Root component with routing
- src/components/ – Reusable UI components (auth, layout, sections, ui)
- src/contexts/AuthContext.tsx – Authentication state management
- src/hooks/ – Custom hooks (useContacts, useSocket)
- src/lib/ – Axios configuration and utilities
- src/pages/ – Page components (Home, Dashboard, Login, Register)
- src/schemas/ – Zod validation schemas

Server Directory (Backend)

- src/server.ts – HTTP server and Socket.IO bootstrap
- src/app.ts – Express application class
- src/config/ – Database, Email, Passport, Socket configuration
- src/controllers/ – Contact controller (CRUD operations)
- src/middlewares/ – Auth, Security, Validation middleware
- src/models/ – User and Contact Mongoose models
- src/routes/ – Auth and Contact route definitions
- src/services/ – Change stream service for real-time updates

Total: ~40 TypeScript/TSX source files

15. Challenges & Solutions

Challenge 1: Workspace Path with Apostrophe

Problem: Problem: The project directory contained an apostrophe (Akshay's Stuff), causing ts-node-dev to crash on Windows.

Solution: Solution: Switched from ts-node-dev to nodemon with ts-node for development server, which handles special characters correctly.

Challenge 2: MongoDB Stale Indexes

Problem: Problem: An old username_1 unique index caused duplicate key errors during Google OAuth sign-ups, even though username field was removed.

Solution: Solution: Added automatic stale index detection and cleanup in database.ts – scans for indexes referencing non-existent fields and drops them on connection.

Challenge 3: Module Path Aliases in Production

Problem: Problem: TypeScript @/* path aliases compiled to require('@/config/database') in JavaScript output, which Node.js cannot resolve.

Solution: Solution: Settled on module-alias for runtime path resolution. Added _moduleAliases config to package.json mapping @ to dist/.

Challenge 4: Serverless WebSocket Limitation

Problem: Problem: Vercel's serverless functions cannot maintain persistent WebSocket connections, breaking Socket.IO.

Solution: Solution: Deployed server on Render (persistent Node.js) and added 15-second HTTP polling fallback in useContacts.ts for when Socket.IO is unavailable.

Challenge 5: Build Dependencies on Render

Problem: Problem: Render's production npm install skips devDependencies, but TypeScript and type definitions needed for build step.

Solution: Solution: Changed Render's build command to 'npm install --include=dev && npm run build' to ensure all compilation tools are available.

16. Future Enhancements

Enhancement	Description
Dark Mode	Toggle between light/dark themes (flags already in .env)
Admin Panel	Separate admin view to see all users' contacts
Email Templates	Rich HTML email templates for notifications
Contact Replies	Reply directly to contacts from the dashboard
File Attachments	Allow file uploads with contact submissions
Analytics Dashboard	Charts and graphs for contact trends over time
Two-Factor Auth	TOTP-based 2FA for additional security
Audit Logging	Detailed logs of all user actions for compliance
Webhook Integrations	Notify external services (Slack, Discord) on new contacts
Mobile App	React Native companion application

17. Conclusion

TechFlow successfully demonstrates the ability to design, build, and deploy a production-grade full-stack web application. The project covers all critical aspects of modern web development:

- **Frontend Excellence:** A responsive, animated React SPA with TypeScript safety and clean component architecture
- **Backend Robustness:** A secure Express API with layered middleware, dual authentication, and real-time capabilities
- **Database Design:** Proper data modeling with per-user isolation and efficient querying
- **Security:** Defense-in-depth with rate limiting, input sanitization, CORS, and encrypted credentials
- **DevOps:** Cloud deployment across Vercel and Render with automated CI/CD from GitHub
- **Developer Experience:** Interactive setup wizard, comprehensive documentation, and clean project structure

The application is fully functional in production and ready for further iteration and enhancement. It serves as a strong demonstration of full-stack development capabilities, modern best practices, and the ability to deliver enterprise-grade software solutions.

Report generated for Web Development Internship Assignment by Akshay Raj.