

SB Works

Freelancing Platform

SDLC Phase 4

IMPLEMENTATION REPORT

Project Name	SB Works — Online Freelancing Platform
Phase	Phase 4: Implementation (Coding)
Prepared By	I. Sai Ganesh — Lead Developer
Duration	21 days of active coding
Date	January – February 2024

1. Implementation Overview

1.1 What Happened in This Phase?

Phase 4 is where all the planning, requirements, and design finally become real software. This is the coding phase — the longest and most intensive part of the project, lasting 21 days. Every database schema, API endpoint, React component, and real-time Socket.io event was written, tested, and refined during this period.

The implementation followed a logical order: first the backend was built (database models, API routes, authentication, real-time server), then the frontend was connected to the backend (services, context, pages). This order made sense because the frontend needs the API to exist before it can make calls to it.

- **Implementation Approach:** The project was coded feature by feature, not layer by layer. That means for each feature (e.g., "proposal system"), both the backend API and the frontend component were built together before moving to the next feature.

1.2 Development Order

Week	What Was Built	Key Files Created
Week 1 Days 1-3	Project setup, folder structure, MongoDB connection, basic Express server	index.js, server.js, .env, package.json
Week 1 Days 4-5	User model, OTP authentication API (get-otp, check-otp)	user.js (model), userAuth.controller.js
Week 1 Days 6-7	JWT cookie system, middleware for auth and roles	user.middleware.js, functions.js
Week 2 Days 8-10	Project model, CRUD API for projects, category model	project.js, project.controller.js
Week 2 Days 11-12	Proposal model and API — submit, list, accept, reject	proposal.js, proposal.controller.js
Week 2 Days 13-14	Socket.io real-time server, message model and API	server.js (socket), message.controller.js
Week 3 Days 15-16	Submission model and API — submit, review, approve	submission.js, submission.controller.js
Week 3 Days 17-18	Review model, notification model and their APIs	review.js, notification.controller.js
Week 3 Days 19-20	Admin routes — user management, category management	admin.routes.js, admin.user.controller.js
Week 3 Day 21	Seed script, environment configuration, backend testing	seed.js, .env, all validators

2. Backend Implementation Details

2.1 Authentication System Implementation

The authentication system was one of the most important parts of the backend to get right. Instead of storing passwords (which creates security risks), SB Works uses OTP (One-Time Password) authentication. Here is exactly how it works:

1. User submits their phone number via POST /api/user/get-otp. The server generates a random 6-digit number and saves it to the user's record in MongoDB with a timestamp.
2. The OTP expires 90 seconds after it is created. If the user tries to verify after 90 seconds, they get an error and must request a new OTP.
3. In development mode (NODE_ENV=development), the OTP is printed directly to the server console so developers can test without SMS. In production, Twilio API sends it as an SMS.
4. User submits their phone + OTP via POST /api/user/check-otp. Server checks the code matches and hasn't expired.
5. On success, two JWT tokens are created: an access token (valid 1 day) and a refresh token (valid 1 year). Both are set as signed HttpOnly cookies.
6. HttpOnly cookies cannot be read by JavaScript in the browser, making them safe against XSS (Cross-Site Scripting) attacks.
7. Every subsequent API request includes these cookies automatically. The verifyAccessToken middleware checks the token on each protected route.

2.2 Middleware Stack

Middleware functions run between the request and the controller. SB Works uses four custom middlewares:

Middleware	File	What It Does
verifyAccessToken	user.middleware.js	Reads JWT from cookie, verifies signature, attaches user to request
isVerifiedUser	user.middleware.js	Checks user account is active (not rejected, status not 0)
authorize(roles)	user.middleware.js	Checks user's role matches required role for the route
validateRequest(schema)	user.middleware.js	Runs Joi validation on request body before reaching controller

2.3 Real-time Features with Socket.io

Socket.io allows the server to push data to connected browsers in real-time without the browser having to keep asking (polling). This is used for two features: live chat and push notifications.

When a user logs in, their browser establishes a WebSocket connection to the server and joins a personal "room" named by their user ID. When someone sends them a message or when a notification needs to be delivered (e.g., "Your proposal was accepted!"), the server emits an event to that specific room — and only that user receives it.

Event	Who Emits	Who Receives	Triggered By
join	Browser (on login)	Server	User connects — joins their room
sendMessage	Sender browser	Server → Receiver	User sends a chat message
receiveMessage	Server	Receiver browser	New message in their conversation
receiveNotification	Server	Target user browser	Proposal, approval, review events

3. Frontend Implementation Details

3.1 React Application Architecture

The frontend is a React Single Page Application (SPA). This means the browser loads one HTML file and React handles all page transitions without full page reloads, making the app feel fast and smooth like a native desktop application.

The most important piece is the `AuthContext` — a React Context that wraps the entire app and provides the current user's data (name, role, status) and the `Socket.io` connection to every component that needs it. This means any page can check "who is logged in?" or "send a notification to this user" without passing data through multiple layers of props.

3.2 Axios Service Layer

All API calls are organized into service files — one per feature area. Instead of writing `fetch()` calls scattered throughout components, each page imports from a clean service:

Service File	Feature	Example Functions
<code>authService.js</code>	Login, profile, logout	<code>sendOTP()</code> , <code>verifyOTP()</code> , <code>completeProfile()</code> , <code>logout()</code>
<code>projectService.js</code>	Project CRUD	<code>getProjects()</code> , <code>createProject()</code> , <code>deleteProject()</code>
<code>proposalService.js</code>	Proposals	<code>submitProposal()</code> , <code>getMyProposals()</code> , <code>updateProposal()</code>
<code>messageService.js</code>	Chat	<code>sendMessage()</code> , <code>getConversation()</code> , <code>getConversations()</code>
<code>submissionService.js</code>	Work submission	<code>submitWork()</code> , <code>getSubmission()</code> , <code>reviewSubmission()</code>
<code>reviewService.js</code>	Reviews	<code>addReview()</code> , <code>getUserReviews()</code>
<code>notificationService.js</code>	Notifications	<code>getNotifications()</code> , <code>markAllRead()</code>
<code>categoryService.js</code>	Categories	<code>getCategories()</code>

3.3 Key Implementation Challenges and Solutions

Challenge Faced	How It Was Solved
Windows PowerShell blocked npm commands	Used Command Prompt (cmd) instead, or set ExecutionPolicy in PowerShell
Twilio SMS costs money in development	Built dev mode that auto-detects missing Twilio credentials and prints OTP to console
Phone validation regex too strict (US only)	Changed to flexible validation accepting 7-20 chars, any format
Users got 403 errors after login	Fixed middleware to allow both status=1 (pending) and status=2 (approved)
Proposals missing project reference	Added project field to <code>ProposalSchema</code> , saved in controller on creation

MongoDB Atlas bad auth errors	Fixed username encoding (%40 for @), recreated DB user with simple password
Frontend hardcoded localhost URLs	Used Vite proxy to forward all /api calls to backend, removed hardcoded URLs
No test data in database	Created seed.js that creates 10 categories and permanent admin account

4. Code Quality and Standards

4.1 Coding Standards Followed

- All controllers follow the same pattern: validate input → process business logic → return JSON response with success flag.
- Error handling is done centrally: all controllers are wrapped in try/catch and return standardized error responses.
- Sensitive configuration (MongoDB URI, JWT secrets, Twilio credentials) is stored in .env file, never hardcoded.
- Database queries use Mongoose methods with proper populate() calls to avoid multiple round-trips.
- All protected routes go through middleware in order: verifyAccessToken → isVerifiedUser → authorize(role) → validateRequest.
- Frontend services return clean data and handle errors with React Hot Toast notifications.
- Component naming follows PascalCase (e.g., BrowseProjects.jsx), service functions follow camelCase.

4.2 Implementation Statistics

Metric	Count
Total backend files created	35+ files
Total frontend files created	30+ files
Database models (Mongoose schemas)	8 models
API endpoints implemented	39 endpoints
React pages built	15 pages
React components created	5 reusable components
Axios service files	8 service files
Middleware functions	4 middlewares
Joi validation schemas	4 validators
Socket.io real-time events	6 events
Project categories seeded	10 categories
Total lines of code (approx)	3,500+ lines

■ Phase 4 Outcome: All 39 API endpoints were implemented and manually tested using Postman. All 15 frontend pages were built and connected to the backend. The full project lifecycle (login → post project → propose → hire → submit work → approve → review) was working end-to-end by the end of Phase 4.