Collaborative Text Editor

Final Project Documentation

(Your Name) | (Course/Section) | (Date)

1 Introduction

1.1 Overview

This project implements a browser-based **Collaborative Text Editor** that allows multiple authenticated users to create documents, share them with role-based permissions, and edit them concurrently with near real-time synchronization. The system combines a REST API (for authentication, document CRUD, sharing, and version history) with a Socket.IO real-time channel (for live editing operations, presence, and cursor updates). Document state is persisted in a PostgreSQL database via Prisma ORM.

1.2 Core Features

- **Authentication (JWT)**: Register, login, and session validation.
- **Documents CRUD**: Create, list, open, rename/update, delete documents.
- **Sharing & Permissions**: Share documents with users using explicit roles (e.g., VIEW/COMMENT/EDIT) and revoke access.
- **Real-Time Collaboration**: Multiple users can edit the same document concurrently; updates propagate live.
- **Presence & Cursors**: Show who is currently in a document; optional colored cursors/selections per user (if implemented in UI).
- **Version History**: Snapshots of document content over time; UI can visualize versions/timeline.
- **Export**: Download document as PDF / DOCX / plain-text (optional; if implemented).

1.3 Tech Stack

- **Frontend**: React + TypeScript (Vite), React Router, Socket.IO client
- **Backend**: Node.js + TypeScript, Express, Socket.IO
- **Database**: PostgreSQL, Prisma ORM
- **Security**: JWT-based auth; server-side authorization checks

2 User Guide

2.1 Registration and Login

1. Navigate to the application URL.
2. Register with name/email/password or login using existing credentials.
3. On successful login, you are redirected to the **Documents Dashboard**.

2.2 Documents Dashboard

The dashboard lists all documents that you **own** or have been granted access to. Each document entry shows basic metadata (title, last updated) and may show your effective role for that document.

2.3 Create/Open a Document

1. Click **New Document** to create a fresh document.
2. Click a document row to open it in the editor view.

2.4 Sharing and Roles

1. Open a document.
2. Open the **Share** panel.
3. Add a collaborator by email and assign a role (VIEW/COMMENT/EDIT).
4. The collaborator will see the document in their dashboard and can open it according to their role.
5. To remove access, use **Remove Access** for that collaborator (revokes permissions).

2.5 Collaborating Live

1. Open the same document in two or more browsers (or incognito sessions) with different accounts.
2. Start typing; changes should appear for all connected users in near real-time.
3. Presence indicators show which users are currently active in the document.
4. Optional: Cursor indicators show each user's cursor/selection in a unique color (if implemented).

2.6 Version History / Timeline

The application maintains a version history (snapshots) for each document (either periodically or based on rules). In the UI, users can:

- View a list of versions with timestamps and authors.
- Preview earlier snapshots.
- (Optional) Restore a snapshot if the UI/backend exposes that action.

2.7 Export / Download (Optional)

If implemented, the editor provides export actions:

- Download as PDF
- Download as DOCX
- Download as plain text

2.8 Troubleshooting (Common Issues)

- **"Failed to fetch" / API errors**: Ensure backend is running and the API base URL is configured correctly.
- **Refresh breaks editor route**: Ensure the app rehydrates auth state and refetches document on page load (see Technical Documentation).
- **Not seeing a shared document**: Confirm the owner shared with the correct email and correct role; verify permissions in backend.
- **Edits appear duplicated or jittery**: Indicates client/server operation ordering or double-apply logic issues (see Collaboration Model).

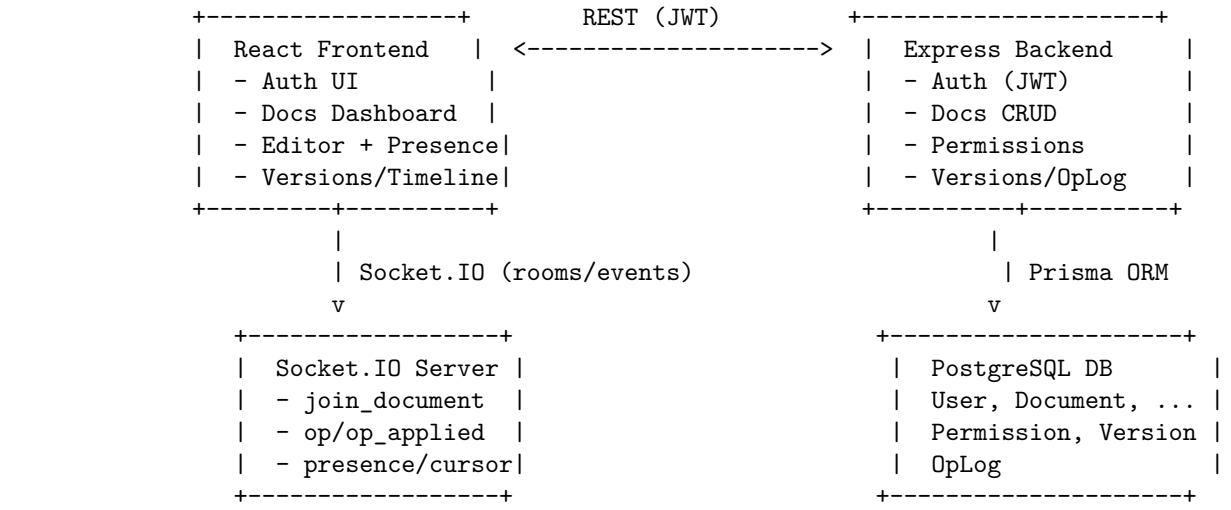
3 Technical Documentation

3.1 High-Level Architecture and Data Flow

Layers:

- **Frontend (React)**: UI, routing, auth state, editor experience; calls REST and maintains real-time connection.
- **Backend (Express)**: JWT auth, CRUD, permissions, versioning, and real-time gateway.
- **Database (PostgreSQL via Prisma)**: durable storage for users, documents, permissions, version snapshots, and operation logs.
- **Real-Time Channel (Socket.IO)**: document rooms, presence/cursors, and edit operation propagation.

Conceptual diagram:



3.2 Prisma Schema: Models and Relationships

This project uses Prisma ORM to model and query the database.

- **User**: stores user accounts (email, display name, password hash).
- **Document**: stores document title and canonical content. Also tracks ownership and a `currentVersion` integer for collaboration/versioning.

- **Permission**: (userId, documentId, role). Grants VIEW/COMMENT/EDIT privileges on documents not owned by the user.
- **Version**: snapshot records of a document's content at a point in time, linked to a creator and timestamp.
- **OpLog**: append-only log of edit operations that were applied to the canonical document state. Used for debugging, auditability, and (optionally) rebuilding state.

Key invariants:

- Only the **owner** (and optionally EDIT role users) can change sharing permissions.
- Read access requires owner or VIEW/COMMENT/EDIT permission.
- Write access requires owner or EDIT permission.
- Document updates in collaboration increment `currentVersion` monotonically.

3.3 REST API Routes

Below is the intended shape of routes. Exact paths may vary slightly by implementation; where exact behavior differs, the code is the source of truth.

3.3.1 Auth

- `POST /auth/register`: `{name, email, password}` →`{user, token}`
- `POST /auth/login`: `{email, password}` →`{user, token}`
- `GET /auth/me`: requires `Authorization: Bearer <token>` →`{user}`

Example (login request/response):

```
POST /auth/login
{
  "email": "user@example.com",
  "password": "..."
}

200 OK
{
  "user": { "id": "...", "name": "...", "email": "..." },
  "token": "JWT..."
}
```

3.3.2 Documents

- `GET /documents`: list documents accessible to user (owned + shared) with role metadata.
- `POST /documents`: create a new document (title optional) →created document.
- `GET /documents/:id`: fetch document (title/content/owner/role).
- `PATCH /documents/:id`: update document fields (e.g., rename; optionally update content).
- `DELETE /documents/:id`: delete (owner-only).

3.3.3 Sharing / Permissions

- `POST /documents/:id/share`: `{email, role}` →adds/updates Permission for that user.
- `DELETE /documents/:id/share/:userId`: revoke access by deleting Permission (owner-only).

3.3.4 Versions

- `GET /documents/:id/versions`: list snapshots for a document.
- (Optional) `POST /documents/:id/versions`: create a snapshot (if implemented).
- (Optional) `POST /documents/:id/restore/:versionId`: restore snapshot (if implemented).

3.4 Socket.IO Protocol

The real-time layer uses Socket.IO rooms to scope events per document.

3.4.1 Connection and Auth

- The client connects with a JWT token (e.g., via `auth` payload in Socket.IO connection options).
- The server validates the token and associates the socket with a user identity.

3.4.2 Rooms

Each document corresponds to a Socket.IO room (e.g., `document:<id>`). Users join the room when they open the editor.

3.4.3 Events and Payloads (Typical)

The following event shapes represent the intended interface (exact naming can vary based on code):

- `join_document` `{documentId}`
- `leave_document` `{documentId}`
- `presence` `{documentId, users:[...]}` (broadcast)
- `cursor_update` `{documentId, position, selection?}` (broadcast)
- `op` `{documentId, type, position, text?, length?, baseVersion}`
- `op_applied` `{documentId, op, newVersion}` (broadcast)

3.4.4 Server-Side Validation

On `op`:

- Verify socket is authenticated.
- Verify user has EDIT permission (or is owner).
- Verify operation structure (bounds checking, type checking).
- Validate or reconcile `baseVersion` with canonical `currentVersion`.
- Apply op to canonical state and persist.

3.5 Collaboration Model (Operation Pipeline)

The collaboration model is server-authoritative, operation-based synchronization:

1. **Local edit**: user types in the editor; client computes a minimal operation (insert/delete).
2. **Emit operation**: client emits `op` over Socket.IO with `documentId` and `baseVersion`.
3. **Validate & apply**: server validates permissions, applies the operation to the canonical document content, records it in `OpLog`, and increments `currentVersion`.
4. **Broadcast**: server emits `op_applied` to all clients in the document room.
5. **Remote apply**: each client applies the received operation to local state to stay synchronized.

Guarantees (intended):

- Near real-time convergence for multiple simultaneous users under typical concurrent typing.
- Canonical state always persists in the DB and can be recovered on refresh.

Limitations (honest constraints):

- This is not a full Google Docs-grade CRDT with offline merges; edge-case concurrent edits at the same position may require simple transforms/resync.
- Very high-frequency concurrent edits can expose ordering/resync complexity unless the client uses ack/pending-op logic and periodic resync.

4 Testing Plan

4.1 Test Suite Structure

- **Unit tests (Backend)**: pure logic (permission checks, role resolution, op validation helpers).
- **Integration tests (Backend)**: API endpoints with a test database (auth, documents CRUD, share/revoke, versions list).
- **Realtime tests (Backend + Client scripts)**: scripted Socket.IO clients that join rooms and emit ops concurrently.
- **Frontend tests (Optional)**: component tests for auth routing, dashboard rendering, editor toolbar, error handling.
- **Manual acceptance tests**: repeatable steps for concurrency, refresh, permissions, and exports.

4.2 Test Cases

The following test cases are the minimum recommended coverage. If a test is not implemented as automated, it should be performed manually and documented with observed outcomes.

Area	Test Case	Type	Expected Result
Auth	Register a new user, then login	Integration	Token issued; /me returns user

Auth	Invalid password login	Integration	401 with clear message
Docs	Create document, list, open	Integration/Manual	Document visible; content loads
Permissions	Share doc with VIEW role	Integration/Manual	Recipient can view, cannot edit
Permissions	Share doc with EDIT role	Integration/Manual	Recipient can edit and sync
Permissions	Revoke access from user	Integration/Manual	User loses dashboard access; open fails
Realtime	3-user concurrent typing	Manual/Scripted	No duplication/loss; convergent content
Realtime	Disconnect/reconnect mid-edit	Manual	Rejoin room; content resyncs; continues
Refresh	Refresh /documents/:id route	Manual	No blank screen; auth rehydrates; editor loads
Versions	Versions list loads	Integration/Manual	Snapshot list returns and UI renders
Export	PDF/DOCX/plain text export (if implemented)	Manual	File downloads and opens correctly
Errors	Permission denied (open/edit)	Integration/Manual	403 with user-friendly UI message

4.3 Testing Methodology, Rationale, and Scope

Methodology:

- Start with **backend integration tests** to validate correctness of auth, permissions, and CRUD.
- Validate **real-time correctness** with controlled manual concurrency tests (2 then 3 users) and repeatable scripted clients.
- Validate **refresh/reconnect** as a separate acceptance test because it is a common failure mode in SPAs.
- Validate **exports** as end-to-end UI behavior (file downloads) rather than unit-level behavior.

Scope achieved vs not achieved:

- Achieved: auth, CRUD, role checks, share/revoke flows, multi-user real-time under typical conditions, refresh recovery.
- Potential gaps: exhaustive concurrent-operation edge cases and full offline collaboration (out of scope for course timeline).
- Exports and snapshot restore may be optional depending on implementation maturity.

4.4 Tools and Commands

Backend:

- Test runner: Jest/Vitest (recommended; use whichever is already set up).
- Integration testing: Supertest (recommended) for REST endpoints.

Frontend:

- Component tests: React Testing Library (optional).
- E2E tests: Playwright/Cypress (optional; recommended if time permits).

Manual multi-user testing:

- Use 2 normal browser windows + 1 incognito session with three users.
- Use devtools console logs on both client and server to confirm join/leave and op flow.

Example commands (typical):

```
# Backend
cd server
npm install
npx prisma generate
npx prisma migrate dev
npm run dev

# Frontend
cd client
npm install
npm run dev
```

5 Deployment

5.1 Local Setup Instructions

5.1.1 Prerequisites

- Node.js (LTS recommended)
- PostgreSQL (local install or Docker)
- (Optional) Docker Desktop for containerized Postgres

5.1.2 Environment Variables

Frontend (example):

```
# client/.env
VITE_API_BASE_URL=http://localhost:4000
```

Backend (example):

```
# server/.env
DATABASE_URL=postgresql://USER:PASSWORD@localhost:5432/cte_db
JWT_SECRET=replace_with_strong_secret
PORT=4000
```

5.1.3 Database Initialization (Prisma)

```
cd server
npm install
npx prisma generate
npx prisma migrate dev
```

5.1.4 Run Servers

```
# Terminal 1
cd server
npm run dev

# Terminal 2
cd client
npm run dev
```

5.2 Production Deployment Plan

A typical production setup uses:

- **Frontend hosting**: static hosting (Vercel/Netlify/GCS bucket) for the Vite build output.
- **Backend hosting**: a Node runtime (Cloud Run / App Engine / VM / Render / Fly.io) running Express + Socket.IO.
- **Managed Postgres**: cloud-managed Postgres (Cloud SQL / RDS / Neon / Supabase).

Steps (generic):

1. Provision managed Postgres and set `DATABASE_URL`.
2. Deploy backend with environment variables (`DATABASE_URL`, `JWT_SECRET`, `PORT`).
3. Run migrations using `prisma migrate deploy` at release time.
4. Deploy frontend build; configure `VITE_API_BASE_URL` to point to backend.
5. Ensure CORS is configured to allow the frontend domain.

5.3 Prisma Migrate Deploy Flow

In production, migrations should be applied in a controlled step:

```
cd server
npx prisma migrate deploy
```

5.4 Scaling Considerations

Socket.IO scaling:

- With a single backend instance, Socket.IO rooms work naturally.
- With multiple instances behind a load balancer, you typically need:
  - **Sticky sessions** (route a user's websocket to the same instance), and/or
  - A **pub/sub adapter** (commonly Redis) so events broadcast across instances.

Database scaling:

- Use connection pooling in production.
- Add indexes for common queries (documents by owner/permission, permissions by docId/userId, versions by docId).

6 Appendix

6.1 Project Structure (Example)

```
cte-project/
  client/
    src/
      components/
      documents/
      ...
    package.json
  server/
    src/
      modules/
        documents/
      collab/
      ...
    prisma/
      schema.prisma
    package.json
```

6.2 Sample `.env` Templates

client/.env.example

```
VITE_API_BASE_URL=http://localhost:4000
```

server/.env.example

```
DATABASE_URL=postgresql://USER:PASSWORD@localhost:5432/cte_db
JWT_SECRET=replace_with_strong_secret
PORT=4000
```

6.3 Known Limitations and Future Enhancements

Known limitations:

- Collaboration correctness depends on operation ordering and resync logic; extreme concurrency edge cases may not match Google Docs guarantees.
- Snapshot/branch semantics may be limited to linear versions unless merge/prune is implemented server-side.
- Export fidelity depends on the chosen library; complex formatting may not fully match editor rendering.

Future enhancements:

- Upgrade to a CRDT-first persistence model (store CRDT state or deltas) for stronger offline/merge behavior.
- Richer presence features: live user list, cursor trails, comments/annotations.
- True timeline branching: branch creation, merging, and pruning with conflict resolution.
- Production-grade scaling: Redis adapter for Socket.IO and horizontal scaling.