Tech Stack Choices

Q1. What frontend framework did you use and why? (React, Vue, etc.)

React 18 was used as the frontend framework. React was chosen because:

1. Component-based architecture enables code reusability and maintainability
2. Fast development with Vite for hot reloading and building
3. Strong ecosystem with React Router for navigation and Axios for API calls
4. Excellent state management with hooks for real-time UI updates
5. Large community support and extensive documentation

Q2. What backend framework did you choose and why? (Express, Flask, Django, etc.)

Express.js was chosen as the backend framework because:

1. Lightweight and flexible Node.js framework
2. Easy to set up RESTful APIs quickly
3. Great middleware support (Multer for file uploads, CORS for cross-origin requests)
4. Integrates seamlessly with JavaScript frontend
5. Minimal configuration required for local development

Q3. What database did you choose and why? (SQLite vs PostgreSQL vs others)

SQLite3 with the better-sqlite3 package was selected because:

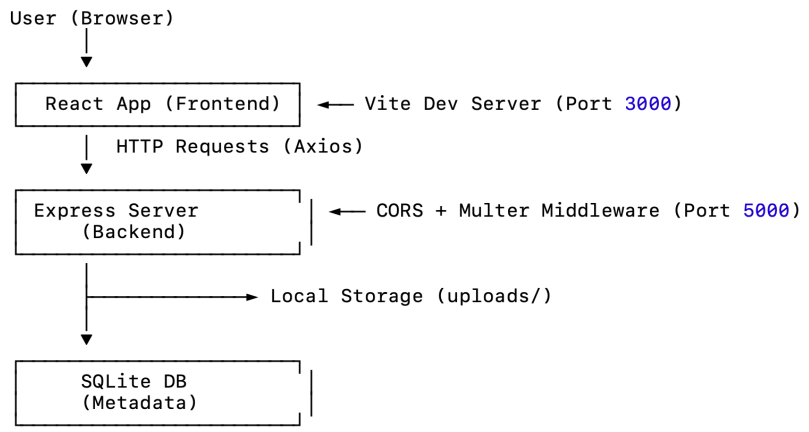
1. Serverless and lightweight - perfect for local development
2. No complex setup or configuration required
3. Single file storage makes deployment simple
4. Synchronous operations for better performance
5. Ideal for single-user applications with moderate data needs

Q4. If you were to support 1,000 users, what changes would you consider?

1. Database: Migrate to PostgreSQL for better concurrency and ACID compliance
2. Authentication: Implement JWT-based user authentication and authorization
3. File Storage: Move to cloud storage (AWS S3) with CDN for better performance
4. Caching: Add Redis for session management and frequently accessed data
5. Load Balancing: Deploy multiple server instances behind a load balancer
6. Monitoring: Implement logging, error tracking, and performance monitoring
7. Security: Add rate limiting, input validation, and file encryption
8. Infrastructure: Deploy on cloud platforms with auto-scaling capabilities

2. Architecture Overview

Q1. Draw or describe the flow between frontend, backend, database, and file storage.



Data Flow Description

1. User Interaction: User interacts with React components (upload, view, delete files)

2. Frontend to Backend: React app sends HTTP requests via Axios to Express server

3. Backend Processing:
   - Express receives requests and applies middleware (CORS, Multer)
   - Validates file types and sizes
   - Processes business logic

4. Data Storage:
   - File Storage: PDF files saved to local uploads directory with timestamped names
   - Database: File metadata (filename, filepath, size, timestamp) stored in SQLite

5. Response Flow: Backend sends JSON responses back to frontend, which updates the UI in real-time

3. API Specification

For each of the following endpoints, provide:

- URL and HTTP method
- Sample request & response
- Brief description

POST /documents/upload

URL & method: POST http://localhost:5000/documents/upload
Description: Accept a multipart/form-data request with a single PDF, validate (MIME/extension, size ≤ 10MB), store file, insert metadata into DB, return saved metadata.

GET /documents

URL & method: GET http://localhost:5000/documents
Description: Return a JSON array of all uploaded document metadata (id, filename, storedName/filepath, filesize, created_at).

GET /documents/:id

URL & method: GET http://localhost:5000/documents/:id
Description: Stream the stored PDF for download. Backend looks up metadata by id, verifies file exists, sends file with appropriate headers.

DELETE /documents/:id

URL & method: DELETE http://localhost:5000/documents/:id
Description: Delete the file from storage and remove its metadata row from the DB. Returns confirmation on success.

4. Data Flow Description

Q5. Describe the step-by-step process of what happens when a file is uploaded and when it is downloaded.

Data flow (upload)

1. User selects a PDF in the React UI and clicks "Upload".
2. Frontend: Axios sends POST /documents/upload with multipart/form-data (field name "file").
3. Backend (Express + Multer): Multer parses request and writes temp file.
4. Backend validation:
    - Check MIME type and .pdf extension
    - Check file size <= 10MB
    - Sanitize original filename to prevent path traversal
5. If validation fails → respond 400/413 with error.
6. If valid → move/rename temp file into uploads/ (e.g., timestamped storedName).
7. Insert metadata into SQLite (documents table): filename, filepath, filesize, created_at → get inserted id.
8. Respond 201 with JSON metadata ({ id, filename, storedName, filepath, filesize, created_at }).
9. Frontend refreshes list (GET /documents) and updates UI.

Data flow (download)

1. User clicks "Download" in the UI for a specific document.
2. Frontend: Axios or direct link issues GET /documents/:id.
3. Backend: query SQLite for document metadata by id.
4. If no record → respond 404.
5. If record found → verify file exists at stored filepath. If missing → respond 404/500.
6. Set response headers:
    - Content-Type: application/pdf
    - Content-Disposition: attachment; filename="original-filename.pdf"
7. Stream file to client (res.sendFile or stream via fs.createReadStream).
8. Client receives binary stream and saves/opens PDF.

5. Assumptions

Q6. What assumptions did you make while building this? (e.g., file size limits, authentication, concurrency)

- Allowed file types: only PDFs (checked by MIME and .pdf extension).
- Max upload size: 10 MB (configurable).
- Storage: local filesystem uploads/ directory (no cloud/S3).
- DB: SQLite3 (better-sqlite3) single-file DB for local/dev use.
- Concurrency: low; better-sqlite3 synchronous operations are acceptable for prototype but not heavy concurrent writes.
- Authentication/authorization: none implemented (single-user / prototype).
- Security controls omitted: no virus scanning, no file encryption at rest, basic filename sanitization only.
- Backups/replication/CDN: not included.
- CORS: allowed for dev origin (http://localhost:3000).