projects

Norah Jones

2025-06-29

Table of contents

Preface

Notes on various projects

Part I

MiniPost

1 MiniPost - Introduction

MiniPost is a minimal full-stack web application designed as a learning project to explore how modern web systems work end-to-end. It features a Single Page Application (SPA) frontend, a GraphQL API backend, secure user authentication using JWTs and bcrypt, and persistent data storage with PostgreSQL. The goal is to gain practical experience with key architectural patterns, tools, and development workflows used in real-world applications — all while maintaining a clean and modular codebase.

2 Project Overview

2.1 MiniPost — A Minimal Blog Platform with Auth and GraphQL

Project Origin & Purpose

This project builds on your in-depth exploration of web technologies — from HTTP and browser internals to SPAs, GraphQL, and authentication. **MiniPost** is your chance to tie all of that together in a single, coherent full-stack application.

What It Will Teach You

- SPA structure and frontend routing
- How GraphQL APIs communicate with clients
- Secure user authentication with hashed passwords and JWTs
- SQL database modeling and persistence
- Full request–response flow from frontend to backend to database

---

Architecture Options

Layer	Technology	Alternatives / Notes
Frontend	React + Vite *(or SvelteKit)*	SPA handles routing, forms, and dynamic DOM
API	Apollo Server *(or GraphQL Yoga)*	GraphQL schema + resolvers
Server	Node.js *(via Express or Fastify)*	Hosts GraphQL endpoint
Database	PostgreSQL	Real SQL schema: `users`, `posts`
Auth	JWT (via `jsonwebtoken`)	Stored in cookie or `localStorage`
Password	`bcrypt`	Secure password hashing
DB Access	`pg`, Knex.js, or Prisma ORM	Choose your abstraction level

---

Core Functionality (Milestones)

Stage 1 — Read-Only Blog (Public)

- Setup DB with `posts` table and seed data
- GraphQL schema exposes `posts: [Post]` query
- Frontend fetches and displays posts

Stage 2 — Authentication

- `users` table in PostgreSQL
- Mutations: `signup(email, password)`, `login(email, password)`
- Use `bcrypt` for hashing, `jsonwebtoken` for issuing tokens
- Frontend stores token and uses `Authorization: Bearer ...`

Stage 3 — Authenticated Posting

- Mutation: `createPost(title, body)` (auth required)
- JWT middleware to protect mutation
- Conditional frontend rendering based on auth state

Stage 4 — Polish & Extras (Optional)

- Add timestamps, pagination
- Apollo Client caching and local state
- Logout button, post editing/deletion, error messages

Development Stack Suggestions

Task	Tools
Frontend	React + Vite *(or SvelteKit)*
GraphQL API	Apollo Server *(or GraphQL Yoga)*
Auth	`bcrypt`, `jsonwebtoken`
DB Integration	`pg`, Knex.js, or Prisma
SQL Database	PostgreSQL
Dev Environment	Docker Compose *(optional)*

---

3 Plan

3.1 Long-Term Development Phases for MiniPost

Each phase builds on the previous one, moving from a basic read-only blog to a secure, interactive full-stack app with user management and refinement.

Phase 1 – Read-Only Blog (Public)

Focus: **Database + Backend + Frontend integration**

- Set up PostgreSQL with seeded `posts`
- Create a minimal GraphQL API with a `posts` query
- Build a frontend that queries and displays posts
- No user accounts, no mutations, no auth

Goal: Get a working end-to-end application with read-only content

---

Phase 2 – Authentication System

Focus: **User accounts, secure login, and token management**

- Add a `users` table to the database
- Create `signup` and `login` mutations
- Hash passwords with bcrypt
- Issue JWT tokens on login
- Frontend stores the token (e.g., in localStorage)
- Attach token to authenticated requests

Goal: Enable user accounts and secure access to protected features

---

Phase 3 – Authenticated Posting

Focus: **User-generated content + authorization**

- Add a `createPost` mutation (requires valid token)
- Associate posts with `author_id`
- Restrict post creation to logged-in users
- Update frontend with a post form (shown only when logged in)
- Validate and sanitize input

Goal: Allow logged-in users to create and view their posts

Phase 4 – Polish and Enhancements (Optional but valuable)

Focus: **Refinement, UX improvements, scalability**

- Add pagination for posts
- Include `created_at` timestamps in schema
- Add `editPost` and `deletePost` mutations (only for author)
- Add logout button on the frontend
- Improve error handling and form validation
- Use Apollo Client features (e.g., cache, local state)
- Prepare for production deployment (build scripts, Docker Compose prod profile)

Goal: Turn a functional prototype into a more realistic app

3.2 Future (Optional) Phases

If you're enjoying the project, you can go further:

- **Phase 5: User profiles** – profile pages, avatars, bios
- **Phase 6: Comments system** – nested content, moderation
- **Phase 7: Admin dashboard** – view users/posts, manage content
- **Phase 8: Deploy to cloud** – e.g., Fly.io, Railway, or Docker VPS

Let me know if you'd like this roadmap saved in a markdown block for direct pasting into your Quarto project, or if you want to adjust the scope or add more phases.

3.3 Phase 1 – Read-Only Public Blog (No Auth)

This phase builds confidence and sets a foundation by developing a minimal working app without authentication.

Step 1: Set up PostgreSQL with Docker (database only)

- Run a local PostgreSQL container using Docker
- Create a `minipost` database
- Define a `posts` table
- Insert seed data manually or using an SQL script

At this stage, no backend or frontend is needed — just ensure the database runs correctly.

Step 2: Build the backend API (GraphQL)

- Scaffold a basic Node.js backend
  - Use `apollo-server-express` or `graphql-yoga`
  - Connect to PostgreSQL via `pg`, Knex, or Prisma
- Define a `Post` type and a `posts: [Post]` query
- Implement a resolver that fetches data from the database

This teaches how the backend communicates with the database and exposes a GraphQL endpoint.

Step 3: Create the frontend SPA

- Use React + Vite or SvelteKit
- Add Apollo Client to consume the GraphQL API
- Query and display blog posts

At this point, you will have a minimal but complete full-stack app (still without auth).

Tooling Overview (Phase 1)

Component	Toolset
Database	PostgreSQL (Docker container)
Backend	Node.js + GraphQL (Apollo Server or Yoga) + pg/ORM
Frontend	React + Vite or SvelteKit
API URL	`http://localhost:4000/graphql`

Starter Template Option

I can generate a minimal project scaffold with this structure:

```
minipost/
  backend/
    index.js
    schema.js
    package.json
  frontend/
    (React/Vite app)
  docker-compose.yml
  README.md
```

If you prefer, we can focus only on the backend and database setup first.

What Comes After Phase 1

Once the read-only blog is working, the next steps will be:

- Add a `users` table
- Implement `signup` and `login` mutations
- Use bcrypt for password hashing and issue JWTs
- Protect the `createPost` mutation and add it to the frontend UI

Next Step: Choose Your Starting Point

Please pick one of the following:

1. Set up PostgreSQL in Docker (Phase 1, Step 1)
2. Scaffold the backend API with GraphQL (Phase 1, Step 2)
3. Generate the full starter template with folder structure and Compose file
4. Something else — you can modify or reorder the plan

Let me know, and I'll guide you through the selected step.