A black background with text and a picture of a person

AI-generated content may be incorrect. A black and white logo

AI-generated content may be incorrect.

**COLLEGE CODE: 9111**

**COLLEGE NAME: SRM Madurai college for engineering and technology**

**DEPARTMENT: Computer Science and Engineering**

**STUDENT NM-ID: 4BF0134536EB1DB9BD0436F56619B6E2**

**ROLL NO: 911123104010**

**DATE: 22/09/2025**

**Completed the project named Phase 3**

**TECHNOLOGY PROJECT NAME:** IBM-FE-Blog Site with Comment Section

**SUBMITTED BY,**

**NAME:** GOKULARAM M

**MOBILE NO: 8870213343**

Blog Project – Phase 3 Full Implementation

# Introduction

This document explains the **full implementation of the Blog Project (Phase 3)**, which integrates:

* A backend built with **Express.js** and **SQLite**
* A frontend built with **React**
* Functionality to manage **posts and comments**
* Instructions to run the project and set up version control

The goal of this phase is to deliver a working blogging platform with post creation, comment submission, and

persistent storage using SQLite.

# Project Structure

The project is organized into two main folders: blog-project/

■ blog-backend/

* server.js
* db.js
* package.json
* blog.db (auto-created)

■ blog-frontend/

■ src/App.jsx

■ package.json

* blog-backend/: REST API server with database.
* blog-frontend/: User interface with React.
* blog.db: SQLite database file auto-generated on first run.

**Core Features Implementation**

1. **View All Post:**
   * Users can see a list of all blog posts (title only) on the homepage.
   * Backend provides a lightweight endpoint (GET /api/posts) to fetch posts.
   * Frontend displays them using React state and a simple list.
2. **View Single Post with Comments**
   * Users can click a post to see its full content and all associated comments.
   * Backend fetches the post and its comments from SQLite (GET /api/posts/:id).
   * Frontend renders post content and iterates through comments dynamically.
3. **Add Comments**
   * Users can add a comment to any post.
   * Backend stores the comment in the comments table with a timestamp.
   * Frontend handles form input, validates non-empty comments, and updates the comment list in real-time

.

1. **Add New Posts (Optional)**
   * Users can submit a new post (title + body).
   * Backend inserts the new post into the posts table (POST /api/posts).
   * Frontend can include a form to create posts dynamically (optional in Phase 3 demo).
2. **Real-Time Updates**
   * Newly added comments are immediately displayed without reloading the page.
   * React state and hooks (useState, useEffect) handle dynamic updates.
3. **Error Handling**
   * Backend returns proper HTTP status codes and error messages if required fields are missing.
   * Frontend shows user-friendly error messages for failed API calls or empty comments.

7.**Responsive UI**

* + Frontend layout adapts to screen sizes using CSS (or optional Tailwind).
  + Posts list and post details appear side-by-side on large screens and stacked on small screens.

# Backend: blog-backend

The backend uses:

* express: REST API framework
* cors: Enable frontend–backend communication
* body-parser: Parse JSON request bodies
* sqlite3: Lightweight relational database

To install:

cd blog-backend npm install

Database: db.js initializes the SQLite database with two tables:

* posts: Stores blog post title & body
* comments: Stores comments linked to posts with timestamps

API Endpoints:

* GET /api/posts  Fetch all post titles
* GET /api/posts/:id  Fetch one post with comments
* POST /api/posts  Add a new post
* POST /api/posts/:id/comments  Add a comment to a post

# Backend Code (server.js)

const express = require("express"); const cors = require("cors");

const bodyParser = require("body-parser"); const db = require("./db");

const app = express(); app.use(cors()); app.use(bodyParser.json());

app.get("/api/posts", (req, res) => {

db.all("SELECT \* FROM posts", [], (err, rows) => {

if (err) return res.status(500).json({ error: err.message }); res.json(rows);

});

});

app.get("/api/posts/:id", (req, res) => { const postId = req.params.id;

db.get("SELECT \* FROM posts WHERE id = ?", [postId], (err, post) => { if (err) return res.status(500).json({ error: err.message });

db.all("SELECT \* FROM comments WHERE postId = ?", [postId], (err2, comments) => { if (err2) return res.status(500).json({ error: err2.message });

res.json({ ...post, comments });

});

});

});

app.post("/api/posts", (req, res) => { const { title, body } = req.body;

db.run("INSERT INTO posts (title, body) VALUES (?, ?)", [title, body], function (err) { if (err) return res.status(500).json({ error: err.message });

res.json({ id: this.lastID, title, body });

});

});

app.post("/api/posts/:id/comments", (req, res) => { const { name, text } = req.body;

const postId = req.params.id;

db.run("INSERT INTO comments (postId, name, text) VALUES (?, ?, ?)", [postId, name, text], functio if (err) return res.status(500).json({ error: err.message });

res.json({ id: this.lastID, postId, name, text });

});

});

app.listen(4000, () => console.log("Backend running on [http://localhost:4000"));](http://localhost:4000/)



# Frontend: blog-frontend

Frontend written in React provides:

* A list of posts
* Ability to view a post’s details
* A comment submission form
* Error handling and loading states

Main component: App.jsx

* Manages state (posts, selectedPost, comments, error messages)
* Fetches posts from API
* Renders UI with Tailwind CSS

UI Flow:

1. Homepage  Displays posts list
2. Post View  Shows post title, body, and comments
3. Comment Form  Add comments with name (optional)

# Frontend Code (App.jsx)

import { useState, useEffect } from "react";

export default function App() {

const [posts, setPosts] = useState([]);

const [selectedPost, setSelectedPost] = useState(null); const [comments, setComments] = useState([]);

const [commentText, setCommentText] = useState(""); const [error, setError] = useState("");

useEffect(() => { [fetch("http://localhost:4000/api/posts")](http://localhost:4000/api/posts)

.then(res => res.json())

.then(setPosts)

.catch(() => setError("Failed to load posts"));

}, []);

const fetchPost = (id) => { [fetch(`http://localhost:4000/api/posts/${id}`)](http://localhost:4000/api/posts/%24)

.then(res => res.json())

.then(data => { setSelectedPost(data); setComments(data.comments || []);

})

.catch(() => setError("Failed to load post"));

};

const submitComment = () => {

if (!commentText.trim()) return setError("Comment cannot be empty"); [fetch(`http://localhost:4000/api/posts/${selectedPost.id}/comments`,](http://localhost:4000/api/posts/%24) {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ name: "Anonymous", text: commentText })

})

.then(res => res.json())

.then(newComment => { setComments([...comments, newComment]); setCommentText("");

})

.catch(() => setError("Failed to submit comment"));

};

return (

<div className="p-4">

<h1 className="text-2xl font-bold mb-4">Blog</h1>

{error && <p className="text-red-500">{error}</p>}

{!selectedPost ? (

<ul>

{posts.map(post => (

<li key={post.id}>

<button onClick={() => fetchPost(post.id)} className="text-blue-600 underline">

{post.title}

</button>

</li>

))}

</ul>

) : (

<div>

<button onClick={() => setSelectedPost(null)} className="mb-2 text-gray-600 underline">

 Back

</button>

<h2 className="text-xl font-semibold">{selectedPost.title}</h2>

<p>{selectedPost.body}</p>

<h3 className="mt-4 font-bold">Comments</h3>

<ul className="mb-4">

{comments.map(c => (

<li key={c.id} className="border-b py-1">{c.text}</li>

))}

</ul>

<textarea value={commentText} onChange={(e) => setCommentText(e.target.value)} className="

<button onClick={submitComment} className="bg-blue-500 text-white px-3 py-1 rounded"> Submit Comment

</button>

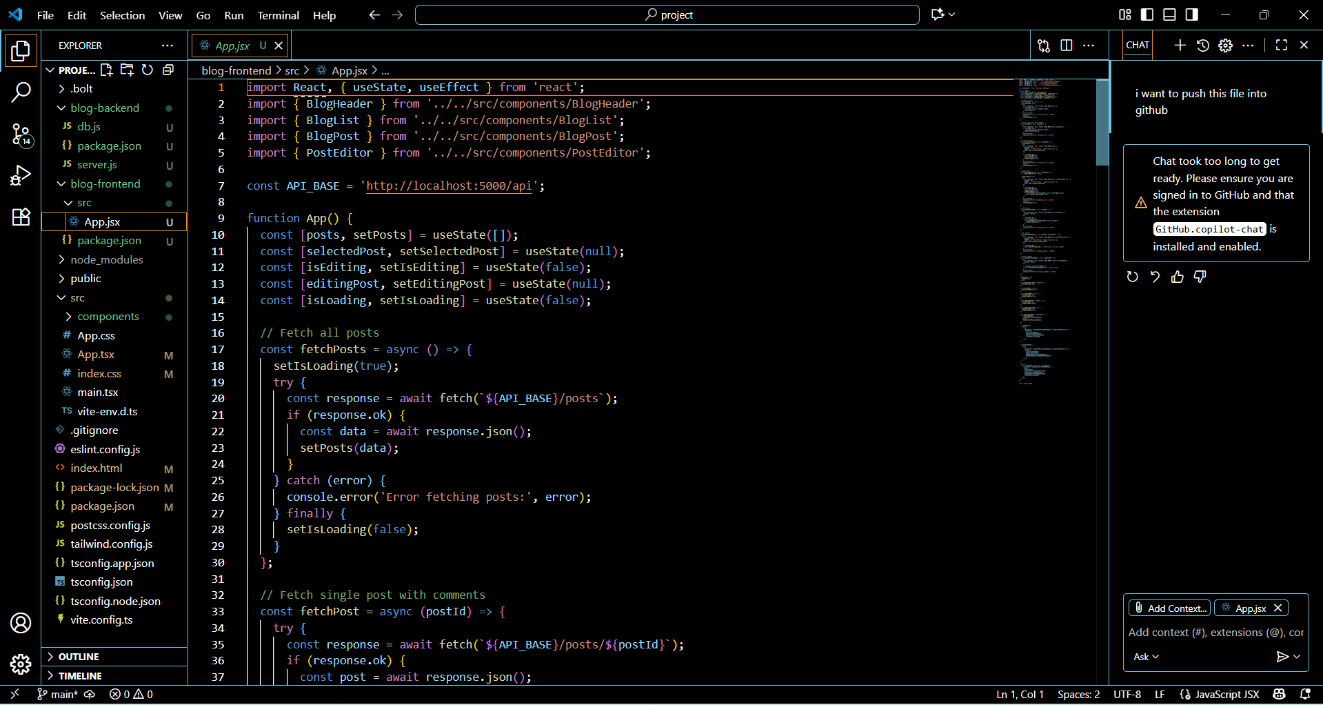
</div>

)}

</div>

);

}



# Running the Project

Backend:

cd blog-backend node server.js

Frontend:

cd blog-frontend npm run dev # if Vite npm start # if CRA

Backend URL: [http://localhost:4000](http://localhost:4000/)

Frontend URL: [http://localhost:5173](http://localhost:5173/) or [http://localhost:3000](http://localhost:3000/)

**Database Setup**

db.js initializes the SQLite database with two tables:

* **posts** → Stores blog post title & body
* **comments** → Stores comments linked to posts with timestamps

Sample seed posts are automatically inserted if the database is empty.

*(Diagram idea: ER diagram showing posts (id, title, body) → comments (id, postId, name, text, createdAt) with a one-to-many relation.)*

**API Endpoints**

The backend exposes the following endpoints:

* GET /api/posts → Fetch all post titles
* GET /api/posts/:id → Fetch one post with comments
* POST /api/posts → Add a new post
* POST /api/posts/:id/comments → Add a comment to a post

This ensures full CRUD support for posts and interaction through comments.

# GitHub Version Control

Initialize repository:

git init git add .

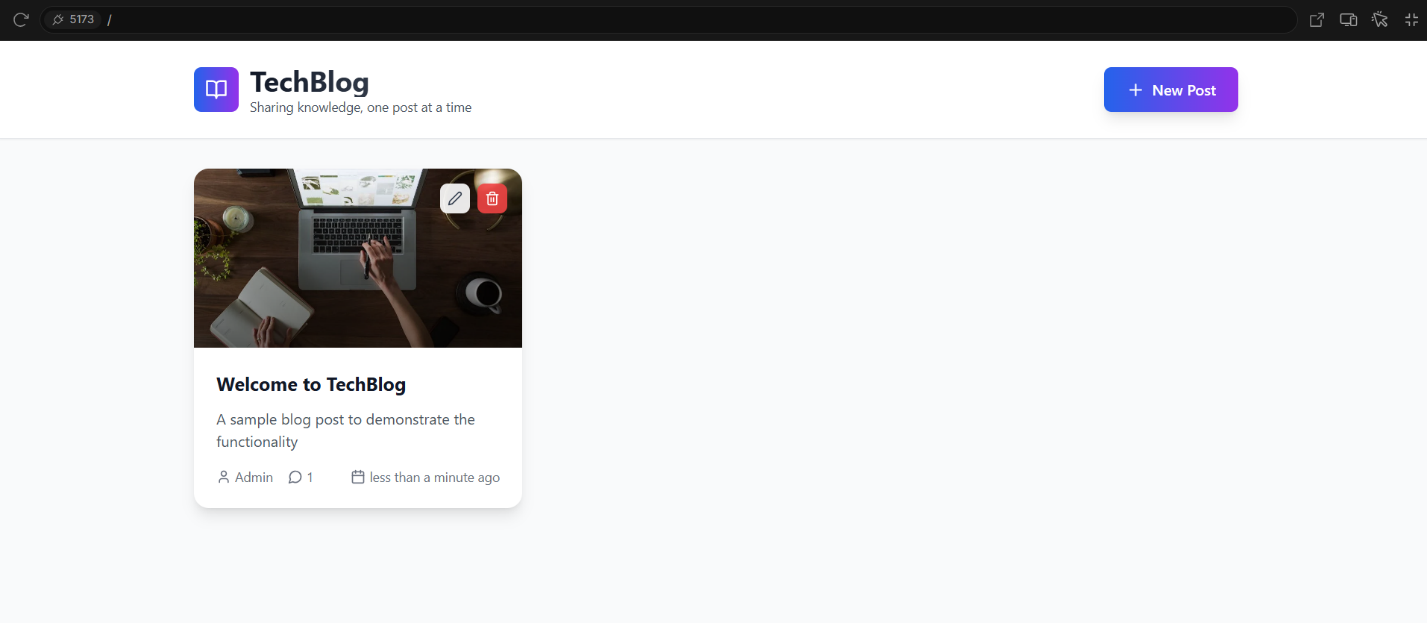
git commit -m "Phase 3 full implementation"

Push to GitHub:

git remote add origin https://github.com/YOUR\_USERNAME/blog-with-comments.git git branch -M main

git push -u origin main

output



# 

# Conclusion

Phase 3 completes the end-to-end blog system:

* SQLite database for persistence
* Express backend for APIs
* React frontend for UI
* Commenting system for interaction
* GitHub integration for version control

Possible extensions:

* User authentication
* Edit/delete posts or comments
* Cloud deployment