COLLEGE CODE:9111

COLLEGE NAME:SRM Madurai College For Engineering And Technology

DEPARTMENT:B.E.CSE(CYBER SECURITY)

STUDENT NM ID:

Shai Barath.M	**07E0646C3B533AE801F48ACF09330E35**	**911123149043**
Lokesh Kumar.A	**D68C451B15F39EB9D044A8655E618D24**	**911123149025**
Rohith.S.R	**300113A10D6D6A6A97F4444B57A07884**	**911123149036**
Syed Ashik Ahamed.S.C	**190187529AEBD4D89A648024BC2FB217**	**911123149051**
Hari Ram.R	**4FB458E6A2D967CAFB64EF5B7349373A**	**911123149014**

Completed the project named as Phase 5

SINGLE PAGE APPLICATION

SHAI BARATH.M
LOKESH KUMAR.A
SYED ASHIK AHAMED.S.C
ROHITH.S.R
HARI RAM.R

Phase 5 — Project Demonstration & Documentation

This is the **final phase** of a project lifecycle, where the focus shifts from development and deployment to **presenting, documenting, and submitting** the completed work. At this stage, you are essentially "wrapping up" the project, showcasing what has been built, and providing clear documentation so others (evaluators, teammates, or future developers) can understand, run, and extend your work.

1. Final Demo Walkthrough

Problem Statement

The **Problem Statement** defines the core issue your SPA aims to solve. It should be specific, user-centric, and explain **why** the application is needed.

Example:

Users find it time-consuming and inefficient to manage daily tasks across multiple platforms. There is no centralized, real-time interface that allows easy task management, collaboration, and progress tracking with minimal load time and seamless navigation.

We need a responsive, browser-based application that offers a fluid, desktop-like experience without frequent page reloads.

In SPA context:

- The problem is solved by reducing latency and improving user experience via client-side routing.
- Data syncs via API calls, keeping the user on the same page

Users & Stakeholders

Clearly identify the primary **end users** and **project stakeholders**.

☎ *Users:*

- Regular users: Individuals using the app for its core functionality.
- Admins: Users with elevated permissions to manage content, users, or data.
- Guest users: Limited-access or read-only users (optional).

💼 *Stakeholders:*

- Product Owner: Sets priorities and defines the product vision.
- Project Manager: Oversees timelines, milestones.
- UI/UX Designers: Design SPA layouts and user flows.
- Frontend Developers: Build the SPA using React, Vue, or Angular.
- Backend Developers: Build REST/GraphQL APIs.
- QA Engineers: Validate functionality and acceptance criteria.

User Stories

User stories define **features from a user's perspective**, usually following this format:

As a [user role], *I want* [goal], *so that* [benefit].

Example User Stories:

- As a **task manager**, I want to create a new task, so that I can track my to-dos.
- As a **team lead**, I want to assign tasks to team members, so that responsibilities are clear.
- As an **admin**, I want to manage user accounts, so

that I can restrict access if needed.

In an SPA:

- Each user story maps to a **dynamic view or component**, not a page reload.
- Data is fetched using APIs and rendered without leaving the main page.

MVP Features

MVP (Minimum Viable Product) features are the essential core functionalities needed to launch the application with just enough value for early adopters.

MVP Features Example (Task Management SPA):

Feature	Description
User Authentication	Login, Register, and Logout (JWT or OAuth)
Create/Edit/Delete Tasks	Full CRUD on tasks
Task List View	Display all tasks with filters and sorting
Assign Tasks	Assign tasks to users
Real-time Sync	Tasks update across devices without reloads (WebSockets or polling)
Responsive UI	SPA works on desktop and mobile

Wireframes / API Endpoint List

Wireframes:

Visual mockups of key SPA views such as:

- Login Screen
- Dashboard

- Task Detail Page
- Admin Panel

Tools: Figma, Adobe XD, Balsamiq

📡 *API Endpoint List (for RESTful backend):*

Endpoint	Method	Description
/api/login	POST	User login
/api/register	POST	User registration
/api/tasks	GET	Get list of tasks
/api/tasks/:id	GET	Get task details
/api/tasks	POST	Create new task
/api/tasks/:id	PUT	Update a task
/api/tasks/:id	DELETE	Delete a task

Acceptance Criteria

These are **conditions that must be met** for a feature or user story to be considered complete. They serve as the **basis for testing and validation**.

Example (for "Create Task" story):

- ☐ User can access a “Create Task” modal or page.
- ☐ Form includes Title, Description, Assignee, and Due Date.
- ☐ Validation ensures all required fields are filled.
- ☐ On submission, task is added to the task list instantly (without full reload).
- ☐ API returns success and task is saved to the backend.
- ☐ UI displays a success message and resets

the form. In SPAs:

- Acceptance criteria often focus on **client-side behavior**, **state changes**, **API communication**, and **UI responsiveness**.

1) Tech stack (recommended)

- *Frontend (mobile + web):*
  - React Native (single codebase for iOS/Android) OR React (web) + React Native for mobile if you prefer separate UIs.
  - UI: Tailwind / NativeBase / React Native Paper for quick, consistent components.
- *Backend / API:*
  - Node.js + Express (fast to iterate) or Python + FastAPI (type hints, great docs). I'll use **Node.js + Express** examples below.
- *Database:*
  - PostgreSQL (relational data, strong joins for users/projects/tasks). Use UUID primary keys.
- *Realtime / Notifications:*
  - WebSockets via Socket.IO for realtime updates and push notification orchestration.
  - Push notifications via Firebase Cloud Messaging (FCM) for Android and APNs for iOS.
- *Background jobs:*
  - Redis + BullMQ (Node) or RQ/Celery (Python) for scheduled reminders, email digests.
- *Caching / rate limiting:*
  - Redis for caching and rate limiting.
- *File storage:*
  - S3-compatible object storage (AWS S3 / DigitalOcean Spaces) for attachments.
- *Auth / Security:*
  - JWT access tokens + refresh tokens (store refresh tokens server-side or rotate), optionally OAuth2 social logins.
- *Observability & CI/CD:*
  - Logging: Winston/Logstash; Monitoring: Prometheus + Grafana; CI: GitHub Actions; Deploy: Docker + Kubernetes or managed container service.

2) UI Structure / API Schema Design

UI structure (screen map, top-level)

- Auth: Login, Signup, Forgot Password
- Home / Dashboard: Project list, quick add task
- Board/List View: Project → columns or list of tasks
- Task Details: title, description, assignee(s), due date, comments, attachments, activity log
- Calendar View: date-based tasks
- Team / Members: invite, role management
- Settings: notification preferences, profile

API Schema (RESTful endpoints + example request/response shapes)

Notes: use `Authorization: Bearer <token>` for protected endpoints. Timestamps in ISO 8601.

Auth

- `POST /auth/signup`
  Request: `{ "name","email","password" }`
  Response: `{ "user": {id,name,email}, "accessToken", "refreshToken" }`
- `POST /auth/login`
  Request: `{ "email", "password" }`
  Response: same as signup
- `POST /auth/refresh`
  Request: `{ "refreshToken" }`
  Response: `{ accessToken, refreshToken }`

Users

- `GET /users/:id` → get profile
  Response: `{ id, name, email, avatarUrl, createdAt }`
- `PUT /users/:id` → update profile

Projects / Boards

- `GET /projects` → list projects user belongs to
  Response: `[ { id, name, role, lastUpdated } ]`
- `POST /projects`
  Request: `{ name, description, visibility: "private|team" }`
  Response: `{ id, name, ... }`
- `POST /projects/:id/invite`
  Request: `{ email, role }`
  Response: `{ invitedUserId, status }`

Tasks

- `GET /projects/:projectId/tasks?view=list|board&assignee=&status=&dueBefore=`
  Response: `[ { id, title, status, assignees:[userIds], dueDate, priority } ]`
- `POST /projects/:projectId/tasks`

Request:

```arduino
{
  "title": "Write spec",
  "description": "Create Phase 1 doc",
  "assignees": ["user-uuid", ...],
  "dueDate": "2025-10-01T12:00:00Z",
  "status": "todo|in_progress|done",
  "priority": "low|medium|high",
  "attachments": []
}
```

- Response: newly created task
- `PUT /tasks/:taskId` → update task fields
- `DELETE /tasks/:taskId`

Comments / Activity

- `POST /tasks/:taskId/comments`
  Request: `{ content }`
  Response: comment object with author and createdAt

Attachments

- `POST /tasks/:taskId/attachments`
  (multipart/form-data) returns attachment meta `{ id, url, filename }`

Notifications

- `GET /notifications`
- `POST /users/:id/notification-preferences`

Realtime

- Socket events: `task:created`, `task:updated`, `task:commented`, `project:invited`

3) Data handling approach

Data modeling (core tables)

- **users** `(id UUID PK, name, email unique, password_hash, avatar_url, created_at, updated_at)`
- **projects** `(id, name, description, visibility, owner_id FK users, created_at)`

- **project_members** (`project_id FK, user_id FK, role, joined_at`) — many-to-many
- **tasks** (`id, project_id FK, title, description, status, priority, due_date, created_by, created_at, updated_at`)
- **task_assignees** (`task_id FK, user_id FK`) — many-to-many
- **comments** (`id, task_id FK, author_id FK, content, created_at`)
- **attachments** (`id, task_id FK, url, filename, uploaded_by, created_at`)
- **notifications** (`id, user_id, type, payload JSON, read boolean, created_at`)

Use indexes on `tasks(project_id, status, due_date)` and `task_assignees(user_id)` for fast lookups.

Transactions & consistency

- Use DB transactions for multi-table operations (e.g., create task + assign + notify).
- Soft deletes (e.g., `deleted_at`) if undo is needed.

Caching & performance

- Cache frequently read but rarely updated items (project list, user profile) in Redis with short TTL.
- Use pagination for lists (`limit/offset` or cursor-based pagination).

Background jobs

- **Reminder jobs:** schedule jobs to send push/email reminders before dueDate (e.g., 24h, 1h). Use BullMQ (Redis) to schedule.
- **Digest jobs:** daily summary emails for each user.
- **Attachment processing:** virus scan / thumbnail generation as background tasks.

Realtime & push notifications

- Use Socket.IO rooms per project to broadcast real-time updates.
- For offline users send push notifications via FCM/APNs; track device tokens in DB.

Backup & retention

- Daily DB backups, WAL archiving for PostgreSQL. Retain backups per RTO/RPO requirements.

Security & rate limiting

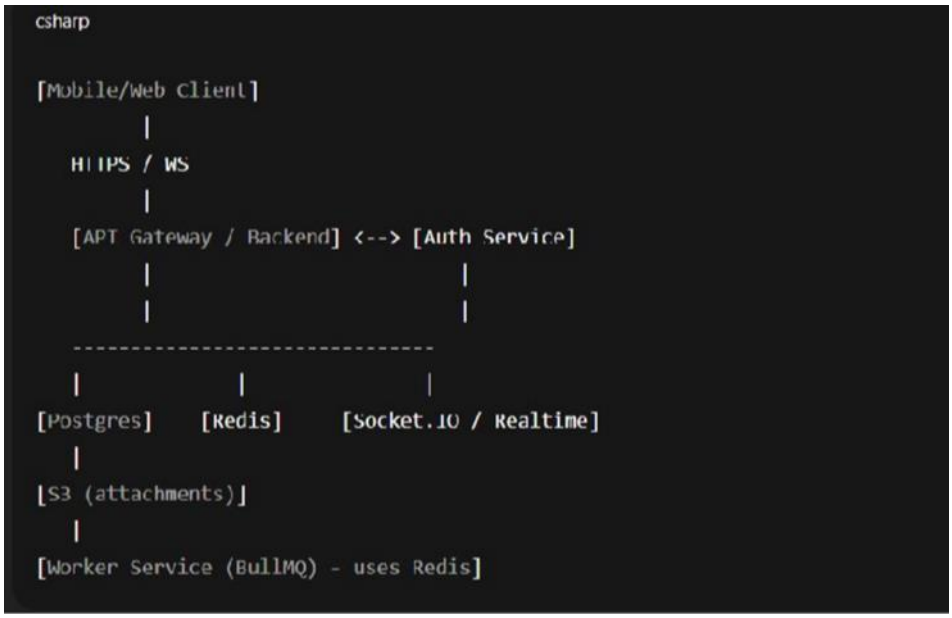
- Rate-limit sensitive endpoints (login, invite) with Redis.
- Encrypt sensitive data at rest? Only store hashed passwords (bcrypt/argon2). Use TLS for in-transit.

4) Component / Module diagram (textual + ASCII)

High-level modules:

- **Client Apps:** Mobile (React Native), Web (React)
- **API Gateway / Backend:** Express app with controllers
- **Auth Service:** token issuance, social login integration
- **Realtime Service:** Socket.IO server (can be same as backend or separate)
- **Worker Service:** background workers (BullMQ) for reminders, attachments
- **Database:** PostgreSQL
- **Cache / Queue:** Redis (caching + BullMQ)
- **File Storage:** S3
- **Push Service:** Integrates with FCM/APNs
- **Monitoring / Logging**

ASCII diagram:



- Backend handles REST and emits socket events to Realtime. Worker service processes scheduled jobs and pushes notifications via Push Service.

5) Basic flow diagram (core flows — stepwise)

A. User sign up + first project

1. Client `POST /auth/signup` (email/password)
2. Backend validates, hashes password, creates `users` row, issues tokens.
3. Backend may create default "Personal" project for user.
4. Client navigates to Dashboard, requests `/projects` and `/projects/:id/tasks`.

B. Create a task & assign

1. Client `POST /projects/:projectId/tasks` with task payload.
2. Backend starts DB transaction: insert into `tasks`, insert into `task_assignees` rows.
3. Backend writes activity log and enqueues job for notifications (BullMQ).
4. Backend emits `task:created` to Socket.IO room `project:{id}`; connected members receive realtime update.
5. Worker picks up notification job and:
    - Sends push notification to assignees (FCM/APNs) if offline.
    - Creates `notifications` DB entries for persistence.

C. Reminder before due date

1. On task creation, schedule a delayed job to run at `dueDate - reminderOffset`.
2. Worker executes job:
    - Query assignees & user preferences.
    - Send push/email; create notification rows.

D. Comment on task

1. Client `POST /tasks/:taskId/comments`.
2. Backend inserts comment and emits `task:commented` event.
3. Realtime clients update UI; offline users receive push notifications.

Extra: Scalability & deployment notes (quick)

- Start monolith (API + Realtime) for speed. Split services (Realtime, Workers) as load grows.
- Use horizontal scaling for stateless backend behind load balancer. Sticky sessions not required if Socket.IO uses Redis adapter for pub/sub.
- Use connection pooling for Postgres and limit long transactions.

Project Setup

What it means: create the foundation for development so everyone can work efficiently and reproducibly.

Concrete tasks / checklist

- Create repository (GitHub): initialize repo, add `README.md`, `.gitignore`, `LICENSE`.
- Add project metadata: purpose, tech stack, high-level architecture diagram in README.
- Define branch & release rules (protected `main`/`master`, PR required).
- Scaffold project structure:
    - Frontend: `src/`, components, pages, styles, test.
    - Backend: `src/`, routes/controllers/services/models, migrations, tests.
    - `infra/` or `ops/` for Docker, terraform, k8s manifests (if used).
- Environment management:
    - `.env.example` with required vars; `.env` for local.
    - Instructions for secrets (Vault, GitHub Secrets).
- Tooling & quality of life:
    - Linter + formatter (ESLint + Prettier for JS, flake8/black for Python).
    - Type system (TypeScript or typed Python).

    - Dependency manager (npm/yarn/pnpm, pipenv/poetry).
    - Testing framework installed and smoke test that runs.
- Containerization & local dev:
    - `Dockerfile` and `docker-compose.yml` to run app + DB locally.
- CI pipeline (basic):
    - CI runs lint → tests → build. (GitHub Actions, CircleCI, etc.)
- Initial tickets / backlog in project board (GitHub Projects, Jira, Trello).

Example commands / first-steps

- `git init && gh repo create my-mvp --public`
- `npm init -y && npx create-react-app frontend --template typescript`
- `docker-compose up -d` (after writing compose)

Core Features Implementation

What it means: implement the smallest set of features that deliver meaningful value to users — the "minimum" in MVP.

How to decide what's core

- Translate product goals into user stories (e.g., "As a user I can sign up so I can save my settings").
- Prioritize by impact vs effort — pick features that validate the product hypothesis.

- Example core set (depending on product) — for many apps: Authentication, Primary Content CRUD, Search/Browse, Basic Settings, Notifications.

Implementation workflow

1. **Define feature spec**: acceptance criteria, wireframe/screens, API contract (request/response).
2. **Break into tasks**: backend endpoints, DB model, frontend components, tests, docs.
3. **Implement incrementally**: basic happy-path first, then edge cases & validation.
4. **Add observability**: logs and basic metrics for each feature.
5. **Demo & iterate**: demo to stakeholder, collect feedback, refine.

Checklist for each feature

- ✅ User story and acceptance criteria
- ✅ API contract documented (OpenAPI/Swagger)
- ✅ DB schema / migration created
- ✅ Backend endpoint + business logic
- ✅ Frontend UI + state handling
- ✅ Unit & integration tests
- ✅ E2E test (if core for user flow)
- ✅ Performance/security checks for that flow

Data Storage (Local State / Database)

What it means: decide where data lives (client-side ephemeral state vs persistent server DB) and implement the storage strategy.

Local state (client-side)

- Use for UI-only ephemeral state: form inputs, modal open/close, UI flags.
- Options: component state, React Context, Redux/RTK, Zustand.
- For short persistence across reloads: `localStorage` or `sessionStorage` (but keep security in mind — don't store sensitive tokens unless encrypted).

When to persist to server DB

- Any user data that must survive between devices or sessions.
- Anything multiple users need to access.
- Sensitive or business-critical data.

Server/database choices

- **Relational (Postgres/MySQL)** — strong consistency, relational data. Great for transactions and structured schemas.

- **NoSQL (MongoDB, DynamoDB)** — flexible schemas, good for document-like data or rapid iteration.
- **Key-value / caching (Redis)** — sessions, rate-limiting, transient caches.
- **Object storage (S3)** — user uploads, static assets.

Schema & migrations

- Design normalized schema (or lightweight denormalized model if read-perf matters).
- Use migrations & version control for schema changes (Prisma Migrate, Alembic, Rails migrations).
- Add indexes for common queries.

Operational concerns

- Connection pooling, limits, and retries.
- Backups and restore strategy.
- Encryption at rest and in transit.
- Access controls, least privilege for DB users.
- Plan for scaling: read replicas, partitioning, or a move to managed DB.

Example minimal data model (generic app)

- `users` (id, email, hashed_password, created_at)
- `items` (id, owner_id, title, body, created_at)
- `transactions` or `events` as needed

Testing Core Features

What it means: verify the MVP actually works — from unit logic to end-to-end user flows.

Types of tests

- **Unit tests** — isolated functions/components (Jest, pytest).
- **Integration tests** — server routes + DB interactions (supertest, pytest + test DB).
- **End-to-end (E2E)** — full user flows in a browser (Cypress, Playwright).
- **Contract tests** — if you have services interacting (Pact).
- **Performance / load tests** — basic smoke load test for critical endpoints (k6).
- **Security scans** — dependency scanning (Snyk), static code analysis.

Testing practices

- Write acceptance tests for each user story (maps to the acceptance criteria).
- Use test doubles where appropriate (mocks, fakes).
- Use test fixtures/factories to create test data.
- Keep tests fast and reliable — flaky tests undermine confidence.
- Run tests on each PR in CI; block merge on failed tests.

Sample test matrix (for a core auth + item flow)

- Unit: hashing function, input validators.
- Integration: signup → DB has user, login → token issued.
- E2E: user signs up → creates item → item visible in list.

Targets

- Aim for meaningful coverage (e.g., >70% for business logic), but prioritize high-value tests over coverage %.

Version Control (GitHub)

What it means: use Git + GitHub to manage code, review changes, and run automation.

Repository & workflow

- Branch strategy: feature branches (`feature/xxx`), PRs into `main` (or `develop` → `main` if you prefer).
- PR process:
  - PR template with checklist: tests, lint, docs, screenshots.
  - Required reviews (1–2 reviewers).
  - Auto-run CI: lint → tests → build.
- Protect `main`: require PRs, pass status checks, enforce up-to-date branches.

Operational GitHub features

- **Issues & Projects**: track tasks and milestones (Week 8).
- **Milestones**: group issues for the MVP release.
- **Labels**: triage and priority (bug, feature, high-priority).
- **Actions**: CI/CD workflows:
  - Run tests and linters on PR.
  - Build artifacts on push to main → deploy to staging.
  - Run security scans, dependency updates (Dependabot).
- **Releases & changelog**: tag semantic versions, publish release notes.

PR & review best practices

- Small PRs (~1 feature/task).
- Write clear description: what changed, why, how to test.
- Link to issue/ticket and include screenshots for UI changes.
- Review checklist: code style, tests present, no secrets, performance considerations.

This is typically the stage of a software project where the initial product (MVP) has already been built and tested, and now improvements, refinements, and deployment to a production environment take place.

1. **Additional Features: -**

- After the core product has been developed and tested in earlier phases, this stage focuses on adding any extra functionality that wasn't part of the minimum viable product (MVP). These could be user-requested features, competitive upgrades, or backlog items.
- **Examples:**
  - Adding a search filter in an e-commerce app.
  - Enabling push notifications in a mobile app.
  - Introducing multi-language support for global users.
- Enhances the product's usability and value proposition, making it more attractive to end-users and stakeholders.

2. **UI/UX Improvements: -**

- UI (User Interface) refers to how the product looks, while UX (User Experience) is about how users feel when using it. This stage focuses on polishing the design and making the application more intuitive.
- **Examples:**

    - Improving button placements for better navigation.
    - Adding animations for smoother transitions.
    - Enhancing accessibility (contrast, font size, screen reader support).
- A great UI/UX increases user satisfaction, reduces errors, and makes the application easier to adopt.

3. <u>API Enhancements</u>: -

- APIs (Application Programming Interfaces) are the backbone of most modern applications, enabling communication between frontend, backend, and third-party services. Enhancements at this stage involve making APIs more efficient, secure, and feature-rich.
- **Examples:**
    - Adding new endpoints for reporting or analytics.
    - Improving response times by optimizing queries.
    - Strengthening authentication with OAuth2.0 or JWT.
- Enhances integration capabilities, improves performance, and supports scalability.

4. <u>Performance & Security Checks</u>: -

- Before deployment, the product must be tested for speed, responsiveness, and security vulnerabilities. This ensures that the application performs well under load and is protected against attacks.
- **Examples:**
    - Load testing to simulate 1,000+ users accessing the app simultaneously.
    - Vulnerability scanning for SQL injection, XSS, and CSRF.
    - Encrypting sensitive user data.
- A fast, secure application improves trust and prevents costly downtime or breaches.

5. <u>Testing of Enhancements</u>: -

- Once new features and improvements are added, they must be thoroughly tested to ensure nothing breaks and all enhancements work as intended.
- **Examples:**
  - Regression testing to ensure old features still work.
  - Automated unit tests for new functions.
  - Manual user acceptance testing (UAT) for overall user experience.
- Testing ensures product stability and avoids issues in production.

6. Deployment (Netlify, Vercel, or Cloud Platform): -

- Deployment is the process of making the application live and accessible to end-users. Depending on the project, this could involve hosting on a cloud platform (AWS, Azure, GCP) or using frontend hosting providers like Netlify or Vercel.
- **Examples:**
  - Deploying a static website on **Netlify**.
  - Hosting a Next.js app on **Vercel**.
  - Deploying a backend on **AWS EC2** or **Heroku**.
- Deployment makes the product usable in a real-world environment, marking the transition from development to delivery.

Visual Code: -

1) package.json: -

```
{
  "name": "phase4-enhancements-deployment",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
```

```
    "start": "next start",
    "lint": "next lint",
    "test": "jest --watchAll=false",
    "test:ci": "jest --runInBand",
    "format": "prettier --write .",
    "perf": "lighthouse-ci autorun",
    "analyze": "cross-env ANALYZE=true next build"
  },
  "dependencies": {
    "next": "^13.4.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "jsonwebtoken": "^9.0.0"
  },
  "devDependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^14.0.0",
    "babel-jest": "^29.0.0",
    "cross-env": "^7.0.3",
    "eslint": "8.40.0",
    "jest": "^29.0.0",
    "lighthouse": "^11.0.0",
    "lighthouse-ci": "^0.10.0",
    "prettier": "^2.8.0"
  }
}
```

2) pages/index.js (UI/UX improvements & additional features sample): -

```
// pages/index.js
import Head from "next/head";
import FeatureCard from "../components/FeatureCard";

export default function Home() {
  const features = [
    { title: "Search", desc: "Fast client-side search with filters" },
    { title: "Notifications", desc: "Push-style notification demo" },
    { title: "Localization", desc: "Example of multi-language support" }
  ];

  return (
    <>
      <Head>
        <title>Phase 4 — Enhancements & Deployment</title>
        <meta name="description" content="Enhancements and deployment example" />
      </Head>

      <main style={{ maxWidth: 900, margin: "3rem auto", fontFamily: "system-ui, sans-serif" }}>
        <h1 style={{ fontSize: 28 }}>Phase 4 — Enhancements & Deployment</h1>
        <p style={{ color: "#333", lineHeight: 1.5 }}>
          This demo showcases additional features, UI/UX improvements, API enhancements,
          performance & security checks, testing, and deployment configuration.
```

```
      </p>

      <section aria-labelledby="features-heading" style={{ marginTop: 24 }}>
        <h2 id="features-heading">Additional Features</h2>
        <div style={{ display: "grid", gridTemplateColumns: "repeat(auto-fit, minmax(220px, 1fr))", gap: 16 }}>
          {features.map((f) => (
            <FeatureCard key={f.title} title={f.title} description={f.desc} />
          ))}
        </div>
      </section>

      <section aria-labelledby="api-heading" style={{ marginTop: 32 }}>
        <h2 id="api-heading">API Example</h2>
        <p>Call <code>/api/hello</code> to see a secure-ish API route (JWT optional).</p>
      </section>
    </main>
  </>
 );
}
```

3) components/FeatureCard.js (accessible component): -

```
// components/FeatureCard.js
export default function FeatureCard({ title, description }) {
 return (
  <article
   role="article"
```

```
      aria-label={title}
      style={{
      padding: 16,
        borderRadius: 12,
        boxShadow: "0 6px 18px rgba(0,0,0,0.06)",
        background: "#fff"
      }}
    >
      <h3 style={{ marginTop: 0 }}>{title}</h3>
      <p style={{ marginBottom: 0 }}>{description}</p>
    </article>
  );
}
```

4) GitHub Actions — Deploy to Netlify: -

```
name: Deploy to Netlify

on:
  push:
    branches: [ main ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
```

```
    - name: Setup Node
      uses: actions/setup-node@v4
      with:
        node-version: 18

    - name: Install deps
      run: npm ci

    - name: Run tests
      run: npm run test:ci

    - name: Build
      run: npm run build

    - name: Deploy to Netlify
      uses: nwtgck/actions-netlify@v1.2
      with:
        publish-dir: ".next"
        production-branch: "main"
        github-token: ${{ secrets.GITHUB_TOKEN }}
      deploy-message: "Auto deploy from GitHub Actions"
      env:
        NETLIFY_AUTH_TOKEN: ${{ secrets.NETLIFY_AUTH_TOKEN
}}
        NETLIFY_SITE_ID: ${{ secrets.NETLIFY_SITE_ID }}
```

11) GitHub Actions — Deploy to Vercel: -

```
name: Deploy to Vercel
```

```yaml
on:
  push:
    branches: [ main ]

jobs:
  vercel-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: 18

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm run test:ci

      - name: Deploy to Vercel
        uses: amondnet/vercel-action@v20
        with:
          vercel-token: ${{ secrets.VERCEL_TOKEN }}
          vercel-args: "--prod"
```

working-directory: .

5) README.md (quick steps & env variables): -

```
# Phase 4 — Enhancements & Deployment (Demo)

## Quick start
1. `npm install`
2. `npm run dev`
3. Open `http://localhost:3000`

## Testing
- `npm run test`
- `npm run perf` (runs lighthouse-ci; will start the dev server automatically)

## Environment variables
- `API_JWT_SECRET` (for API JWT test)
- For Netlify/GitHub Actions:
  - `NETLIFY_AUTH_TOKEN`, `NETLIFY_SITE_ID`
- For Vercel:
  - `VERCEL_TOKEN`

## CI/CD
- Use `.github/workflows/deploy-netlify.yml` or `deploy-vercel.yml`.
- Tests run in CI before deployment.

## Notes on enhancements
- UI/UX improvements in `components/` and `styles/`.
- API enhancements in `pages/api/hello.js`.
```

- Security headers in `next.config.js`.
- Performance checks via `lighthouse-ci` config.

2. Project Report

- A structured written document summarizing the entire project from start to finish. It acts as the official record of your work.

- **Typical contents:**
  - **Introduction:** Problem statement, objectives, and scope.
  - **Methodology:** Tools, technologies, and frameworks used.
  - **System Design:** Architecture diagrams, workflows, data models.
  - **Implementation:** Key modules, code overview.
  - **Results:** Features achieved, performance outcomes.
  - **Conclusion & Future Work:** What was accomplished and potential improvements.
- Helps evaluators, peers, or future developers understand the **what, why, and how** of your project.

3. Screenshots / API Documentation

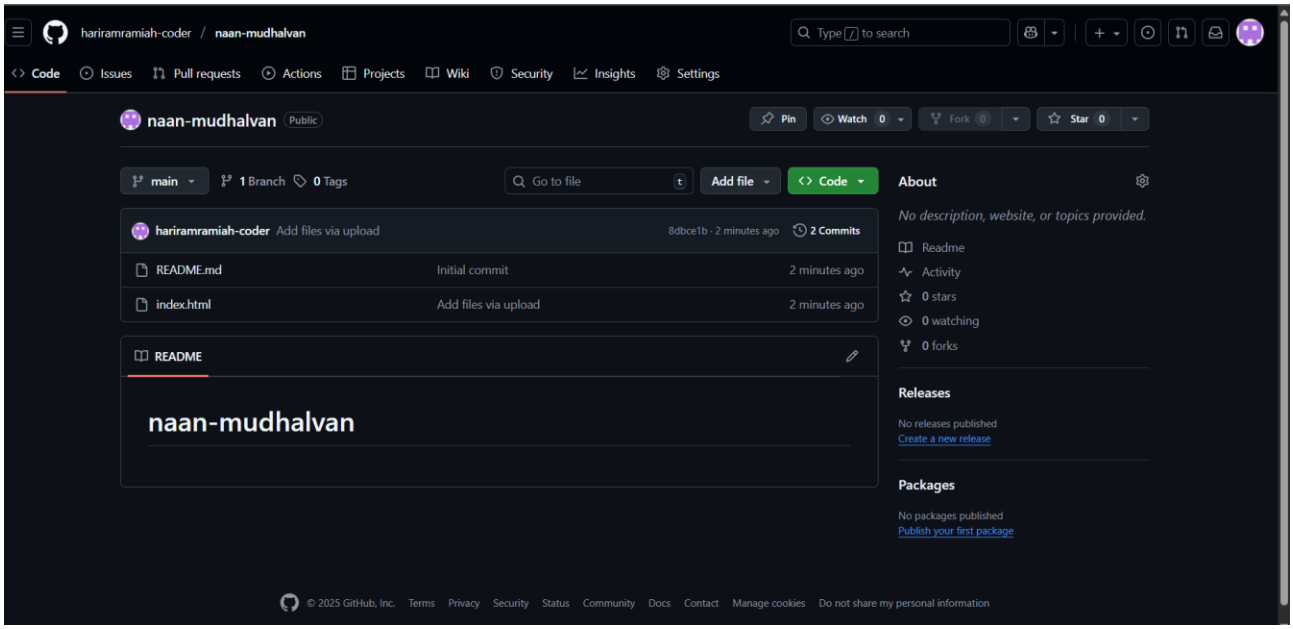
- Visual and technical evidence of your project's implementation. Screenshots highlight the UI and workflows, while API documentation provides technical details of the backend endpoints or integrations.
- **Examples:**
  - Screenshots of login, dashboard, or reports.
  - API endpoint details like GET /users with request/response format.
  - Sample curl/Postman commands.
- Provides quick proof of features and helps other developers (or evaluators) test your project without reading the entire codebase.

4. Challenges & Solutions

- A section that highlights the **problems faced during development** and how your team solved them. It reflects problem-solving and critical thinking skills.
- **Examples of challenges:**
  - API integration issues due to authentication errors.
  - Performance problems with large datasets.
  - Deployment hurdles (e.g., dependency conflicts).
- **Examples of solutions:**
  - Switching to JWT authentication.
  - Optimizing queries with indexing.

    - Using Docker for environment consistency.
- Shows evaluators your ability to overcome obstacles and your technical adaptability.

5. GitHub README & Setup Guide



6. Final Submission (Repo + Deployed Link)

- The official handover of your project, including the **GitHub repository** and the **live deployed application link** (on Netlify, Vercel, AWS, etc.).
- **Checklist for submission:**
    - GitHub repo should have clean, well-structured code.
    - README file should be complete.
    - Deployed link should be working and accessible.
    - Make sure repo and link are both tested before submission.
- This marks the **completion of your project** and gives evaluators the ability to check both the codebase and the live system.

Github Link: - https://github.com/hariramamiah-coder/naan-mudhalvan