Job Scheduler Dashboard

About the Developer

Developer Profile

- **Name:** Thadaka SanjayKumar
- **Role:** Full Stack Developer (MERN Stack)[Fresher]
- **Education:** B.Tech in Computer Science Engineering(2025)
- **Specialization:** Web Development, Backend Systems, UI Engineering
- **Technologies:** React.js, Node.js, Express.js, SQLite, REST APIs

Professional Summary

I am a passionate **Full Stack Developer** with a strong interest in building scalable, real-world web applications. I enjoy working across both **frontend and backend layers**, focusing on clean architecture, maintainable code, and user-centric design.

This project, **Job Scheduler Dashboard**, was developed as a demonstration of my ability to:

- Design **end-to-end systems**
- Implement **asynchronous background processing**
- Build **enterprise-style dashboards**
- Follow **industry-standard architectural patterns**

ABSTRACT

The **Job Scheduler Dashboard** is a full-stack web application developed to manage and monitor background jobs in a structured and efficient manner. The system provides a centralized platform where users can create, execute, update, and delete jobs while tracking their lifecycle in real time.

In modern software systems, many operations such as data processing, notifications, and scheduled tasks are executed asynchronously as background jobs. Managing these jobs effectively requires clear visibility, controlled execution, and reliable state tracking. This project addresses those requirements by implementing a job lifecycle model that transitions through **Pending**, **Running**, and **Completed** states.

The backend of the application is built using **Node.js and Express**, with **SQLite** as the persistence layer to store job data and execution metadata. Business logic is organized using a controller–service architecture to ensure scalability and maintainability. A webhook mechanism is integrated to notify external systems when a job completes, demonstrating an event-driven design approach commonly used in enterprise systems.

The frontend is developed using **React**, with a modern glassmorphism-based user interface. It provides real-time job monitoring through a polling mechanism, responsive layouts for desktop and mobile devices, inline editing capabilities, and intuitive filtering and search options. Theme management is implemented using the Context API, allowing seamless switching between light and dark modes.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

In modern software applications, many operations cannot be executed instantly within a single user request. Tasks such as data processing, report generation, notifications, and scheduled operations are commonly handled as background jobs. Managing these jobs efficiently is essential to ensure system reliability, performance, and transparency.

The **Job Scheduler Dashboard** is designed to address this need by providing a centralized platform to manage background job execution. It allows users to create, execute, monitor, update, and delete jobs while maintaining complete visibility into each job's lifecycle. The system simplifies job management by offering a clean and intuitive interface backed by a robust backend architecture.

This project simulates a real-world job scheduling system used in enterprise environments. It demonstrates how background jobs transition through different states—**Pending**, **Running**, and **Completed**—and how these transitions are tracked and reflected in real time on the user interface. By doing so, the application bridges the gap between backend processing and frontend visibility.

The backend is implemented using **Node.js and Express**, following a layered architecture to separate concerns such as request handling, business logic, and database operations. Job data is stored persistently using **SQLite**, ensuring reliability and data consistency. Additionally, a webhook mechanism is incorporated to notify external systems when a job completes, highlighting an event-driven approach commonly used in distributed systems.

On the frontend, the application is built with **React**, focusing on component reusability, responsiveness, and user experience. Features such as inline editing, filtering, real-time updates through polling, and theme support enhance usability and demonstrate modern UI development practices.

Overall, the Job Scheduler Dashboard serves as a comprehensive example of full-stack application development, combining asynchronous processing, RESTful API design, responsive user interfaces, and professional software architecture. The project is suitable for enterprise-level evaluation and provides a strong foundation for understanding job scheduling and monitoring systems.

1.2 Technology Stack Used:

Frontend Technologies:

React.js

- Used to build a **component-based, dynamic user interface**
- Enables efficient state management and re-rendering
- Supports reusable UI components such as tables, cards, filters, and modals

Styled-Components

- Used for **CSS-in-JS styling**
- Allows dynamic styling based on theme (light/dark mode)
- Prevents global CSS conflicts by scoping styles to components

Framer Motion

- Provides smooth and consistent **UI animations**
- Used for transitions, modals, hero sections, and interactive components
- Enhances user experience with professional motion effects

Radix UI (Select Component)

- Used to implement accessible and customizable dropdowns
- Ensures consistent behavior across browsers
- Improves dark-theme visibility and UI quality

React Router DOM

- Handles client-side routing
- Enables navigation between:
    - Create Job Page
    - Jobs Dashboard Page
- Supports URL-based job search (job ID in query parameters)

Backend Technologies:

Node.js

- Provides a non-blocking, event-driven runtime
- Suitable for handling asynchronous background job execution

Express.js

- Used to create RESTful APIs
- Manages routes for:
    - Job creation
    - Job execution
    - Job updates
    - Job deletion
- Implements middleware for request parsing and error handling

---

Database

SQLite

- Lightweight, file-based relational database
- Stores job-related data such as:
    - Task name
    - Priority
    - Status
    - Timestamps
- Ideal for assessment and prototype-level applications
- Demonstrates SQL-based data modeling and persistence

Webhook Integration

Webhook (External Event Notification)

- Used to notify external systems when a job completes
- Sends structured job completion data to a configured webhook URL
- Demonstrates event-driven architecture and system integration concepts

---

1.3 PROJECT STRUCTURE

The **Job Scheduler Dashboard** follows a clean **frontend–backend separation** with modular architecture. This structure improves **readability, maintainability, scalability, and team collaboration**, which is expected in enterprise-level applications.

Frontend Structure (React)

```
frontend/
├── src/
│   ├── animations/
│   │   └── motionPresets.js
│   │
│   ├── components/
│   │   ├── Button.jsx
│   │   ├── CapsuleTabs.jsx
│   │   ├── Filters.jsx
│   │   ├── GlassSelect.jsx
│   │   ├── JobCard.jsx
│   │   ├── JobForm.jsx
│   │   ├── JobTable.jsx
```

```
│   │   ├── SuccessMessage.jsx
│   │   └── ThemeToggle.jsx
│   │
│   ├── context/
│   │   └── ThemeContext.jsx
│   │
│   ├── layout/
│   │   ├── Navbar.jsx
│   │   ├── MobileNavbar.jsx
│   │   └── Footer.jsx
│   │
│   ├── pages/
│   │   ├── CreateJobPage.jsx
│   │   └── JobsPage.jsx
│   │
│   ├── services/
│   │   └── jobApi.js
│   │
│   ├── styles/
│   │   ├── GlobalStyles.js
│   │   ├── theme.js
│   │   └── glass.js
│   │
│   ├── App.jsx
│   └── main.jsx
│
├── public/
│   └── index.html
│
└── package.json
```

- **Frontend Folder Explanation**

  animations/

  - Contains reusable animation presets using Framer Motion
  - Ensures consistent motion behavior across the application

components/

- Reusable UI components
- Includes buttons, tables, cards, filters, dropdowns, and modals
- Keeps UI logic modular and maintainable

context/

- Global state management (Theme Context)
- Handles light/dark mode across the entire app

layout/

- Shared layout components like Navbar and Footer
- Ensures consistent layout on all pages

pages/

- Route-level components
- Each page represents a major screen in the application

services/

- Centralized API communication layer
- Handles all HTTP requests using Axios

styles/

- Global styles and theme definitions
- Manages dark/light theme configurations

Backend Structure (Node.js + Express)

```
backend/
├── controllers/
│   └── jobController.js
│
├── database/
│   └── db.js
```

```
│
├── routes/
│   └── jobRoutes.js
│
├── services/
│   ├── jobService.js
│   └── webhookService.js
│
├── app.js
├── server.js
├── .env
└── package.json
```

- **Backend Folder Explanation**

controllers/

- Handles request–response logic
- Validates inputs and returns appropriate HTTP responses

database/

- Manages SQLite database connection
- Initializes tables and schema

routes/

- Defines REST API endpoints
- Maps routes to corresponding controller functions

services/

- Contains business logic and database queries
- Separates core logic from controllers (clean architecture)

app.js

- Express application setup
- Registers middleware and routes

server.js

- Entry point of the backend
- Starts the server and listens on a specified port

.env

- Stores environment variables (e.g., webhook URL)

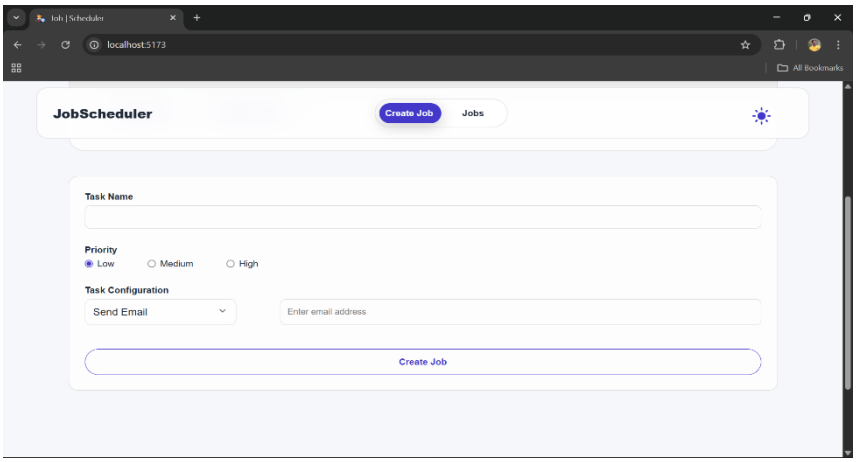
Database Structure

CHAPTER-2

Functional Overview of the Job Scheduler Dashboard

2.1 Create Job Screen (Home Page):

Purpose:
Allows users to create a new background job by providing required details.



Functionality:

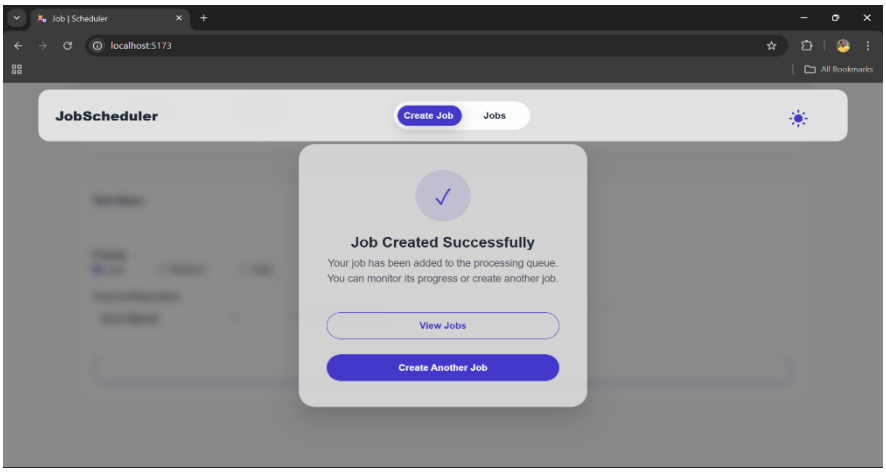
- User enters:
  - **Task Name**
  - **Priority (Low / Medium / High)**
  - **Optional Payload**
- On clicking **Create Job**:
  - Data is sent to the backend via POST /jobs
  - Job is stored in the database with status pending
- A **success popup modal** appears after job creation

Key Features:

- Form validation (required fields enforced)

- Clean glassmorphism UI
- Responsive layout for mobile & desktop

2.2 Success Popup (After Job Creation):



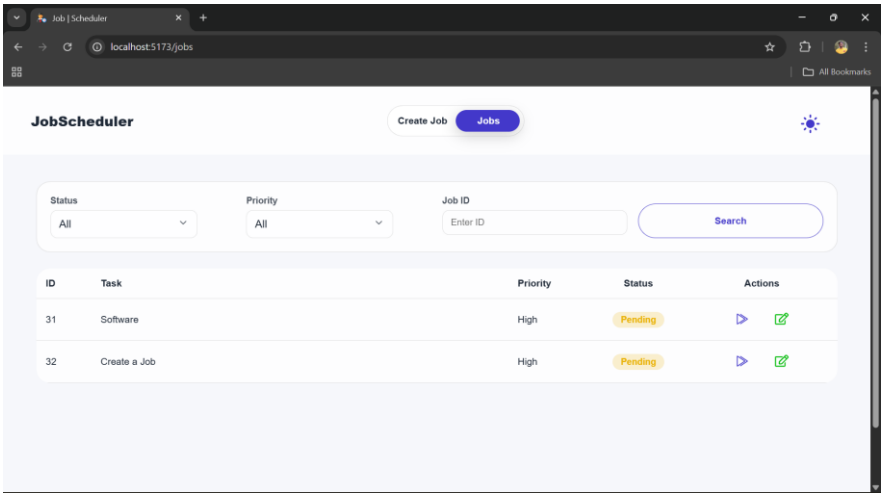
Functionality:

- Appears as a **centered modal popup**
- Prevents background interaction (desktop)
- Displays:
    - Confirmation message
    - Two action buttons:
        - **View Jobs**
        - **Create Another Job**

User Actions:

- **View Jobs** → navigates to Jobs Dashboard
- **Create Another Job** → resets form

2.3 Jobs Dashboard (List View):



Purpose:

Displays all jobs with their current status and actions.

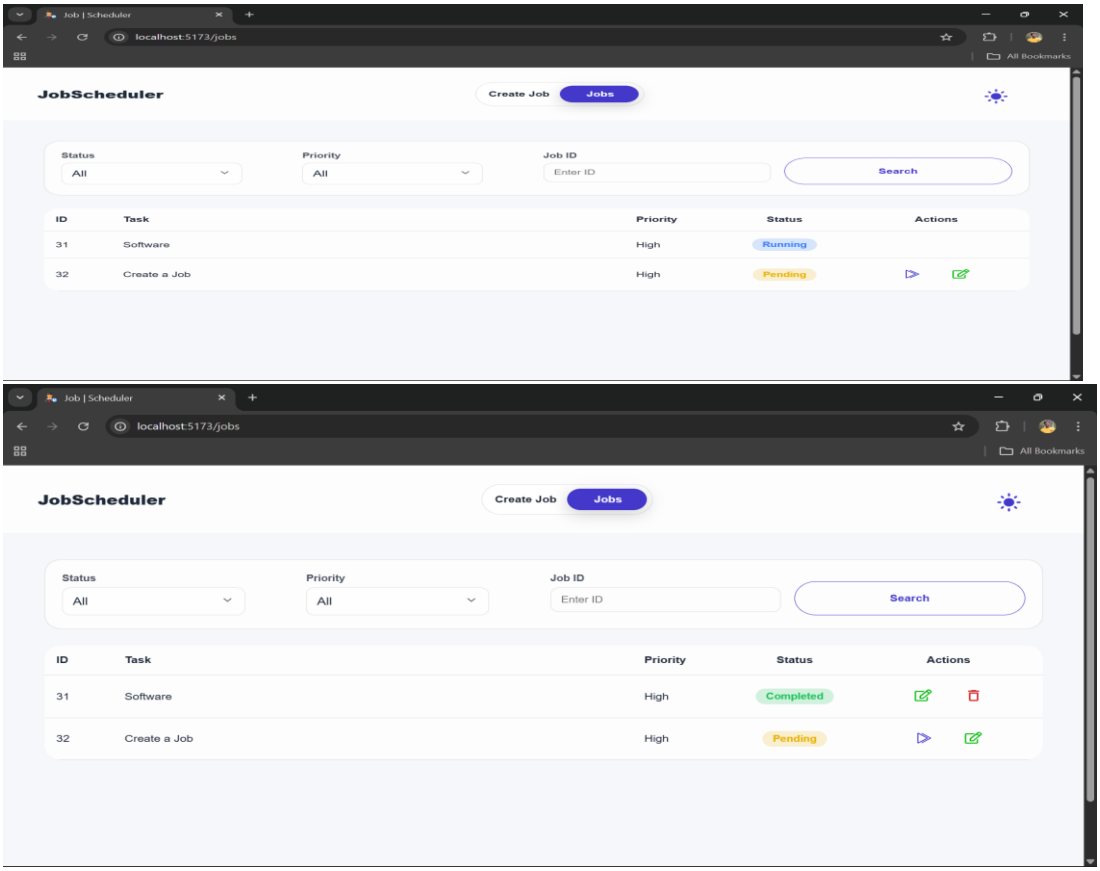
Columns:

- Job ID
- Task Name
- Priority
- Status
- Actions

Status Flow:

Status	Meaning
Pending	Job created but not executed
Running	Job currently processing
Completed	Job finished successfully

2.4 Run Job Functionality:



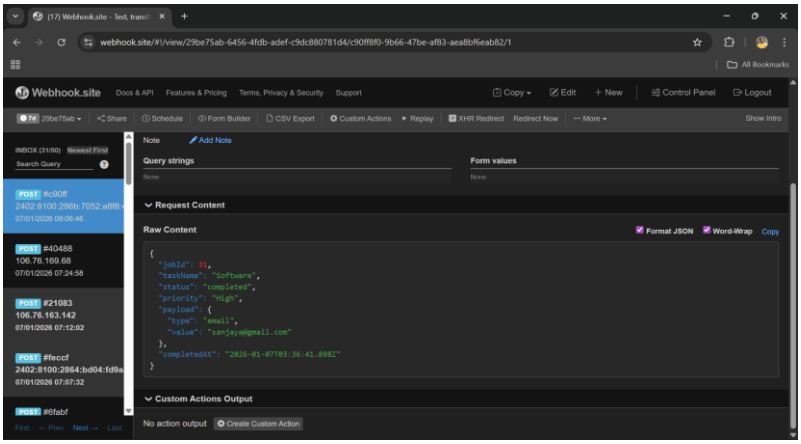
Functionality:

- **Run button visible only for pending jobs**
- Clicking **Run**:
  - Calls POST /jobs/:id/run
  - Backend updates status to running
  - Simulated execution using setTimeout
  - After completion:
    - Status updated to completed
    - Webhook triggered

Why This Matters:

- Demonstrates background processing
- Simulates real async job execution
- Shows status lifecycle management

2.5 Webhook Trigger (Post Completion):



Purpose:

Notifies external systems when a job completes.

Trigger Condition:

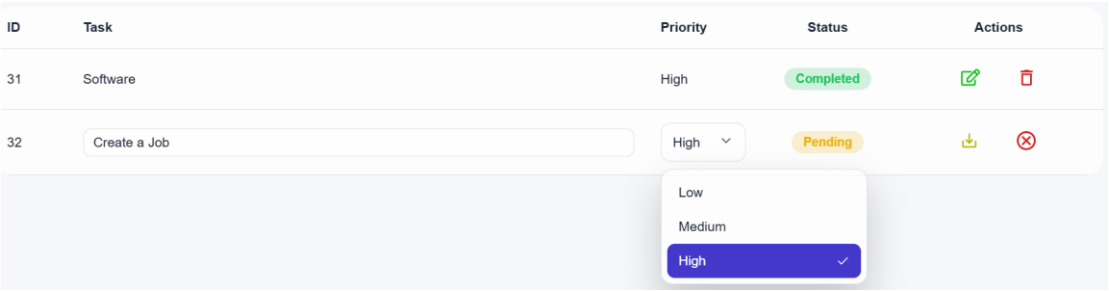
- Only fires when:
    - Job status changes to completed

Payload Sent:

- Job ID
- Task Name
- Priority
- Status
- Completion timestamp
- Payload (if provided)

Webhook is **NOT triggered** on update/delete
It is strictly for **job completion events**

2.6 Edit Job Inline (Table Edit):



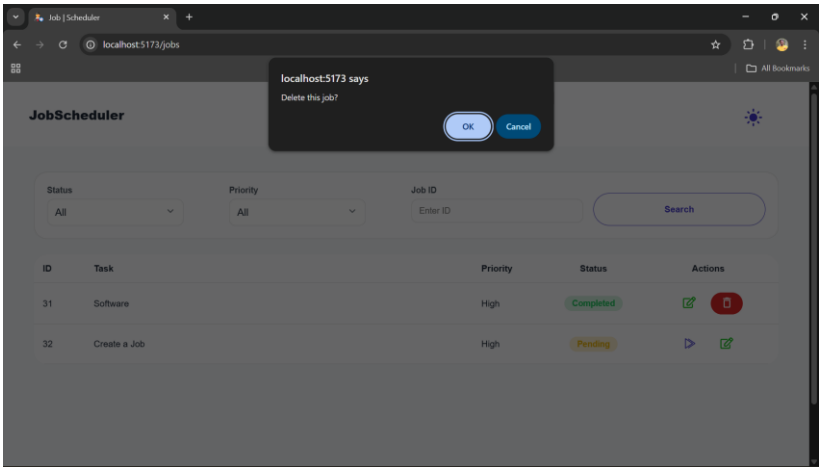
Functionality:

- Clicking **Edit**:
  - Converts row into editable inputs
  - Button changes to **Save**
- User can edit:
  - Task Name
  - Priority
- Clicking **Save**:
  - Calls PATCH /jobs/:id
  - Updates database
  - UI refreshes automatically

Business Rule:

- Running jobs **cannot be edited**
- Completed jobs are locked

2.7 Delete Job:



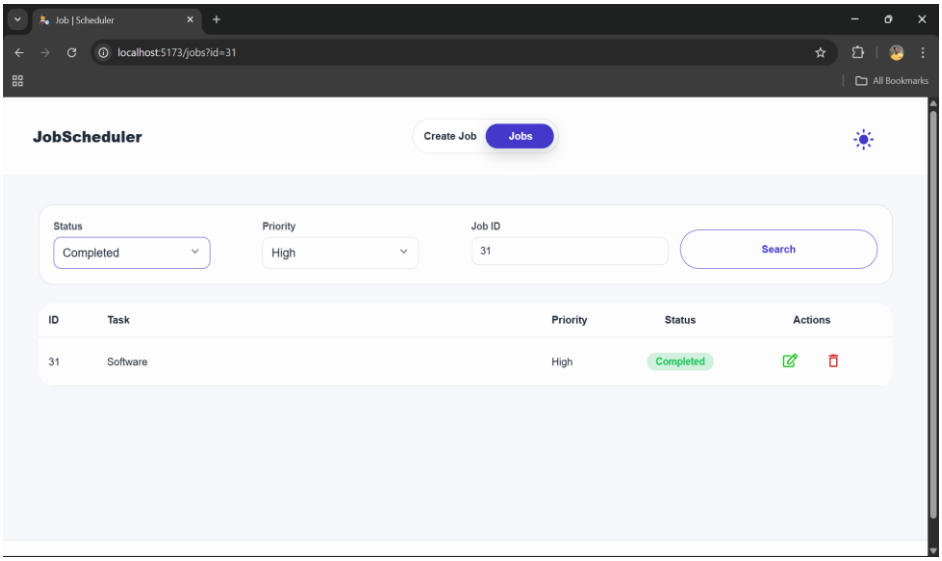
Functionality:

- Delete button visible only for **completed jobs**
- Confirmation dialog shown before deletion
- Calls DELETE /jobs/:id
- Job removed permanently from database

Safety Rules:

- ❌ Running jobs cannot be deleted
- ✔ Completed jobs only

2.8 Filters & Search:



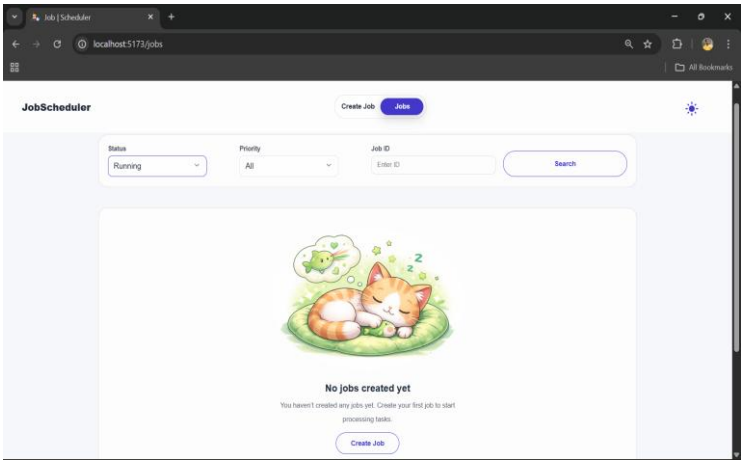
Filters Available:

- Status (Pending / Running / Completed)
- Priority (Low / Medium / High)
- Job ID Search

Behavior:

- Status & Priority → client-side filtering
- Job ID Search:
    - Calls GET /jobs/:id
    - Updates URL query param
    - Supports direct URL access

2.9 Empty State (No Jobs):



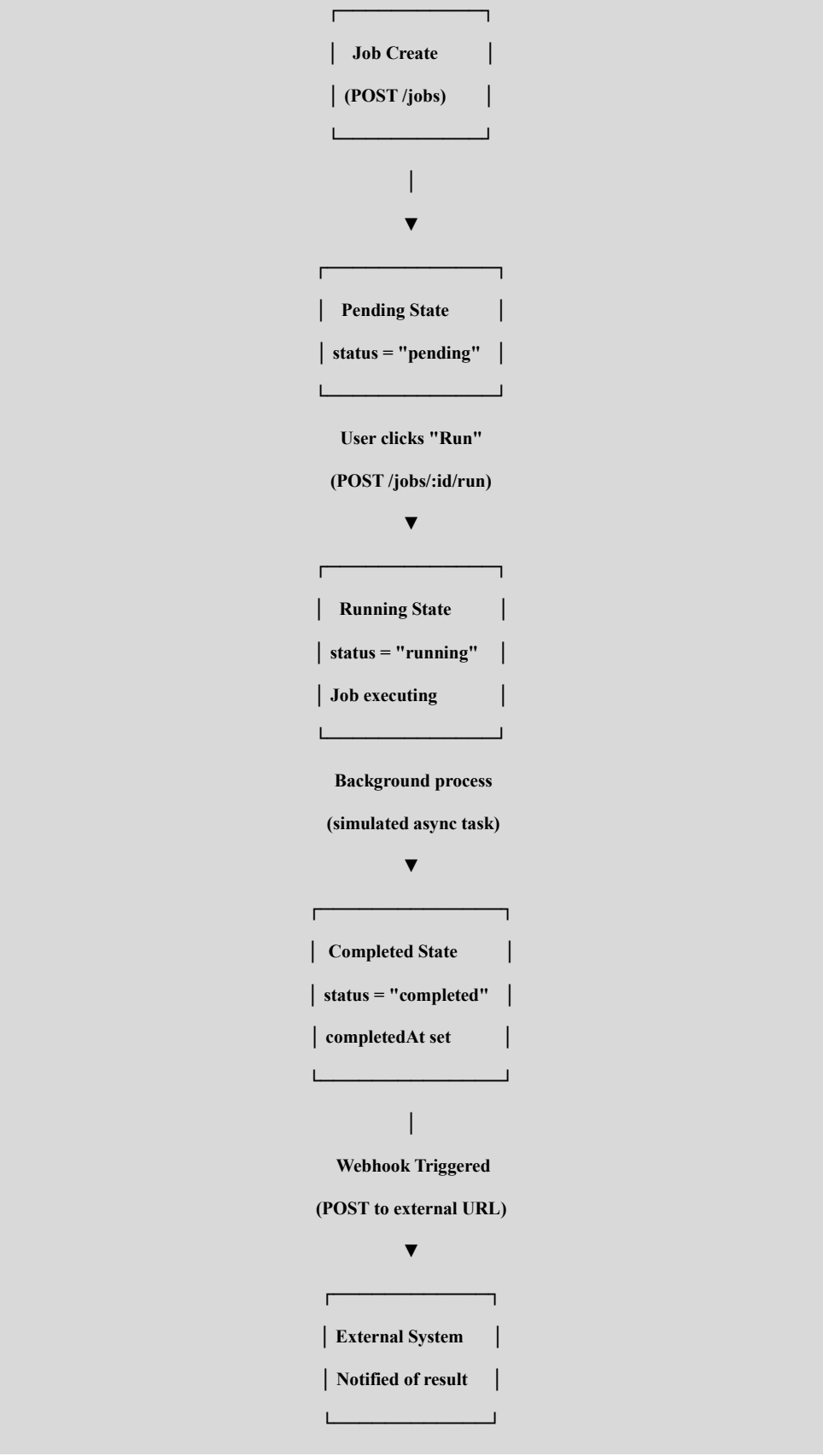
Purpose:

Displayed when job list is empty.

UX Improvements:

- Friendly illustration
- Clear guidance message
- Matches table width for visual consistency
- Supports dark & light themes

2.10 Job Life Cycle Diagram:



CHAPTER-3

Life Cycle Explanation (Simple & Clear)

3.1 Job Creation

- User submits job details from UI
- Backend stores job in database
- Initial status is set to **pending**

API Used:

POST /jobs

3.2 Pending State

- Job exists but has **not started**
- User can:
    - Edit job
    - Run job
    - View job details

Why pending exists:

Allows control before execution (real-world schedulers behave the same).

3.3 Running State

- Triggered when user clicks **Run**
- Backend updates status to **running**
- Job execution happens asynchronously

API Used:

POST /jobs/:id/run

Rules:

- Running jobs **cannot be edited**
- Running jobs **cannot be deleted**

3.4 Completed State

- Job execution finishes
- Backend updates:
    - status → completed
    - completedAt timestamp

Why update is required:

So frontend & database reflect final job state correctly.

3.5 Webhook Trigger

- Once job reaches completed
- Backend sends job data to external system

Purpose of webhook:

- Notify other systems
- Enable event-driven architecture
- Simulate enterprise integrations

Triggered ONLY when:

✓ Job is completed

❌ Not on edit

❌ Not on delete

Post-Completion Rules

Action	Allowed
Edit job	❌ No
Run job again	❌ No
Delete job	☑ Yes
Webhook	☑ Already triggered

Why This Life Cycle Is Important

- Demonstrates **state management**
- Shows **asynchronous job execution**
- Implements **real-world job schedulers**
- Uses **event-driven webhook integration**
- Enforces **business rules properly**

CHAPTER-4

Version Control & Deployment Process

4.1 Version Control Using Git & GitHub

Purpose

Version control is used to:

- Track code changes
- Maintain history
- Enable collaboration
- Ensure code safety and rollback capability

This project uses **Git** as the version control system and **GitHub** as the remote repository.

Repository Structure

The project is organized into two main folders:

job-scheduler/

├── frontend/

├── backend/

Each module is versioned independently but maintained in a single repository for simplicity.

Git Workflow Followed

The following Git workflow was followed during development:

1. **Initialize Repository**

git init

2. **Add Remote Repository**

git remote add origin https://github.com/Sanjay-Kumar-Git/job-scheduler.git

3. **Stage Changes**

git add .

4. **Commit Changes**

git commit -m "Initial project setup"

5. **Push to GitHub**

git push origin main

Each feature (UI, API, database, webhook) was committed incrementally to maintain clarity and traceability.

Commit Practices

- Meaningful commit messages
- Logical grouping of changes
- Incremental updates instead of large commits

Example:

feat: add job run API

fix: priority filter issue

ui: improve glass select styling

4.2 Deployment Process – Frontend (Vercel)

Platform Used

- **Vercel** (Frontend hosting)

Reason for Choosing Vercel

- Seamless React deployment
- GitHub integration
- Automatic CI/CD
- Fast global CDN
- Zero-config deployment

Frontend Deployment Steps

1. Login to **Vercel Dashboard**
2. Click **New Project**
3. Import GitHub repository
4. Select **frontend folder** as root
5. Configure environment variables if needed
6. Click **Deploy**

Vercel automatically:

- Installs dependencies
- Builds the project
- Deploys it globally
- Assigns a public URL

CI/CD with Vercel

Every time a commit is pushed to the main branch:

- Vercel automatically triggers a new build
- Deploys the updated frontend
- Ensures latest code is live

This provides **continuous integration and deployment**.

4.3 Backend Deployment (Local / Cloud)

Current Setup

For assessment purposes, the backend is:

- Run using **Node.js**
- Hosted locally on port 5000

npm install

npm start

API Base URL

http://localhost:5000

Production-Ready Enhancement (Optional)

In real-world scenarios, the backend can be deployed using:

- Render
- Railway
- AWS EC2
- DigitalOcean
- Fly.io

The same API structure can be reused without frontend changes.

4.4 Environment Configuration

Environment variables are managed using a .env file.

Example:

WEBHOOK_URL=https://webhook.site/#!/view/29be75ab-6456-4fdb-adef-c9dc880781d4/c90ff8f0-9b66-47be-af83-aea8bf6eab82/1

Benefits:

- Secure sensitive values
- Easy configuration changes
- Environment isolation

4.5 Webhook Testing

The project uses **Webhook.site** for testing webhook events.

When Webhook is Triggered:

- Only when a job transitions to completed

Payload Sent:

- Job ID
- Task Name
- Status
- Priority
- Payload
- Completion timestamp

4.6 Deployment Summary

Component	Platform	Purpose
Frontend	Vercel	UI hosting
Backend	Node.js	API server
Database	SQLite	Data persistence
Webhook	Webhook.site	Event testing
Repo	GitHub	Version control

4.7 Benefits of This Deployment Strategy

- 🚀 Fast and reliable frontend hosting
- 🔁 Automatic updates via GitHub
- 🔐 Secure environment handling
- 🧪 Easy webhook testing
- 📈 Scalable architecture

CHAPTER-5

Problem Statement

In modern applications, background job processing is a critical requirement. Tasks such as data processing, report generation, notifications, and integrations often need to be executed asynchronously without blocking the user interface. However, many systems lack:

- Proper job lifecycle management
- Real-time status tracking
- Centralized job monitoring
- Clear feedback to users
- Integration hooks for external systems

This creates challenges in tracking task execution, debugging failures, and scaling systems effectively.

CHAPTER-6

Objective of the Project

The objective of this project is to design and implement a Job Scheduler Dashboard that:

- Allows users to create background jobs
- Enables execution and monitoring of jobs
- Tracks job lifecycle states (Pending → Running → Completed)
- Stores job data persistently
- Notifies external systems upon job completion
- Provides a modern, user-friendly interface

The system aims to simulate real-world enterprise job scheduling behavior in a simplified yet scalable manner.

CHAPTER-7

Project Description

The Job Scheduler Dashboard is a full-stack web application built using modern technologies.

Key Features:

- Create background jobs with priority and payload
- View all jobs in a dashboard
- Execute jobs manually
- Edit job details
- Delete completed jobs
- Filter jobs by status, priority, and ID
- Real-time updates using polling
- Webhook integration for job completion events
- Dark/Light theme support
- Responsive design (Desktop & Mobile)

The application follows a client–server architecture, where the frontend communicates with the backend via REST APIs, and the backend manages job execution and persistence.

CHAPTER-8

Advantages of the System

Technical Advantages

- Clean separation of frontend and backend
- Scalable and modular architecture
- Proper REST API design
- Centralized job state management
- Persistent storage using SQLite
- Webhook-based event notification

User Experience Advantages

- Modern glassmorphism UI
- Clear job status indicators
- Responsive layout
- Friendly empty state visuals
- Smooth animations and transitions

Business / Real-World Relevance

- Mimics enterprise job schedulers
- Demonstrates async processing
- Shows event-driven integration via webhooks
- Useful for task automation platforms

CHAPTER-9

Future Improvements & Enhancements

The project can be extended with the following features:

- Authentication & authorization (Admin/User roles)
- Job retry mechanism on failure
- Job failure states & logs
- Real background workers (Redis + BullMQ)
- WebSocket-based real-time updates (instead of polling)
- Cron-based scheduled jobs
- Multi-tenant job management
- Analytics dashboard for job statistics

These improvements would make the system production-ready for large-scale usage.

CHAPTER-10

Conclusion

The Job Scheduler Dashboard successfully demonstrates how background jobs can be created, managed, and monitored in a full-stack application.

By combining:

- A modern React-based frontend
- A robust Node.js backend
- Persistent database storage
- Event-driven webhook notifications

The project delivers a realistic simulation of enterprise job scheduling systems while maintaining simplicity and clarity.

This system serves as a strong foundation for learning and showcasing:

- Asynchronous processing
- RESTful API design
- State management
- UI/UX best practices
- Backend business logic

CHAPTER-11

Regards & Acknowledgements

Acknowledgement

I would like to express my sincere gratitude to everyone who supported and guided me throughout the development of the Job Scheduler Dashboard project.

This project has been an important learning milestone, allowing me to apply both frontend and backend concepts in a real-world, production-style application.

Guidance & Learning Support

I would like to acknowledge:

- Mentors & Instructors

  For providing conceptual clarity in:

  - Full Stack Development
  - REST API design
  - Database handling
  - Real-time job processing concepts
- Learning Platforms & Documentation
  - React & Styled Components documentation
  - Node.js & Express.js documentation
  - SQLite documentation
  - Radix UI & Framer Motion references

These resources helped me design a clean, scalable, and maintainable system.

Tools & Technologies Appreciation

I sincerely appreciate the open-source community for providing powerful tools that made this project possible:

- React.js for frontend development
- Node.js & Express.js for backend services
- SQLite for lightweight database management
- Styled Components for modern UI styling
- Vercel for seamless frontend deployment
- Webhook.site for webhook testing

Each tool contributed significantly to the success of this project.

Personal Growth & Learning Outcome

Through this project, I gained hands-on experience in:

- Full-stack application architecture
- API lifecycle management
- Job execution workflows
- UI/UX design for dashboards
- Error handling and validation
- Real-time status updates
- Deployment and version control

This project strengthened my confidence in building production-ready applications.

Closing Note

I consider this project a strong foundation for my journey as a Full Stack Developer. The knowledge gained here will be instrumental in tackling more complex systems in the future.

Thank you to everyone who contributed directly or indirectly to this learning experience.

Prepared By

Thadaka Sanjay Kumar

MERN Full Stack Developer

Job Scheduler Dashboard Project