

# MAIN PROJECT REPORT

Jayam Shanmukha Bhuvan - IMT2020506

---

## TakHub - an Intuitive Task Manager Application



**Github repo link:** <https://github.com/Bhuvan506/TaskHub>

### 1. Introduction

The project started with the analysis of the requirements and user stories provided by the client. Based on this, a detailed project plan was created, outlining the different development stages and milestones. The frontend development began with the creation of wireframes and mockups to define the application's layout and user interface. React components were then implemented using the Tailwind CSS framework to ensure a responsive design that adapts to different screen sizes and devices. Redux was used for state management, allowing for efficient data flow between components.

The backend development involved setting up a Node.js server using the Express framework. The server handled incoming requests and communicated with the MongoDB database to fetch and store task data. RESTful APIs were designed to allow users to perform CRUD (Create, Read, Update, Delete) operations on boards and tasks.

---

---

Authentication and authorization were implemented using JSON Web Tokens (JWT) to ensure secure access to the application.

Version control was managed using Git, with different branches for feature development and bug fixes. Continuous integration was achieved using Jenkins, which automated the building and testing process of different branches. To achieve containerization and easy deployment, Docker was used to package the application along with its dependencies into containers. The containers were then deployed to production environments using Ansible automation, ensuring a consistent deployment process across different servers.

Logging of server requests and errors was handled using Morgan, a middleware library, which generated log files. These logs were then processed and aggregated in an ELK (Elasticsearch, Logstash, Kibana) stack, allowing for monitoring and analysis of the application's performance.

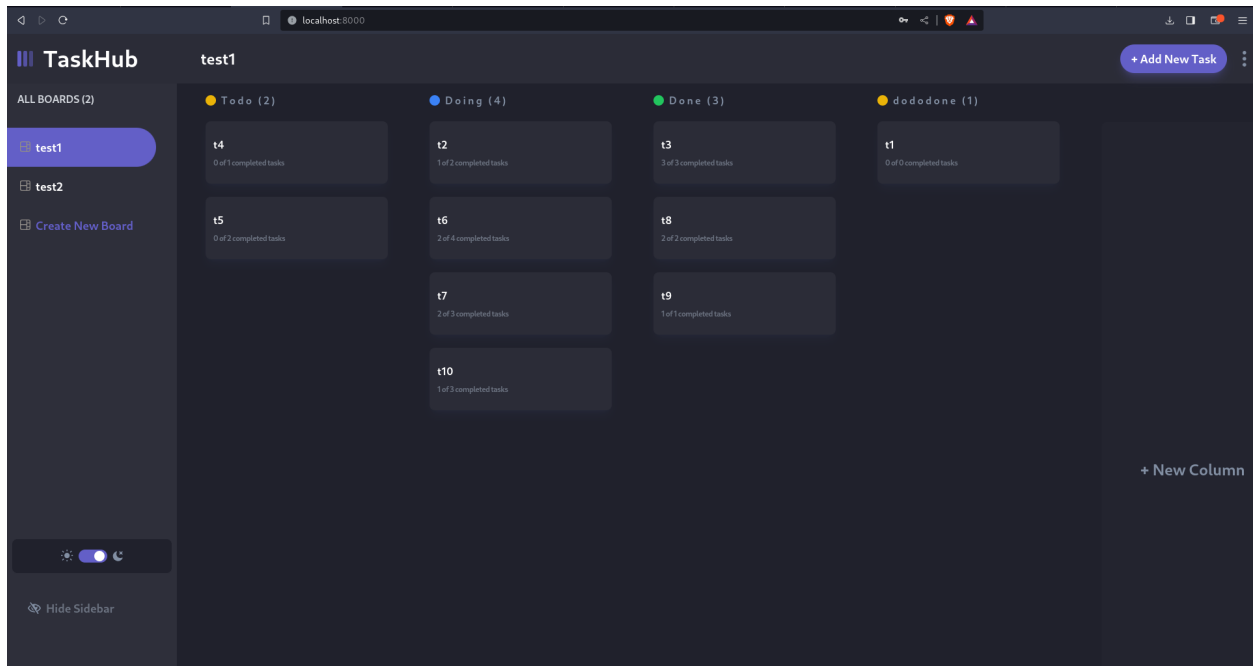
Extensive testing was conducted throughout the development process. Unit tests were written using Jest for testing React components, while integration tests were performed to ensure the proper functioning of the APIs. Documentation was provided for both end users and API consumers. This included user manuals with step-by-step instructions on how to use the application, as well as API documentation outlining the available endpoints, request/response formats, and authentication requirements.

Overall, this project report outlines the development process of a task management application from start to finish, covering the frontend and backend implementation, version control, continuous integration and deployment, containerization, logging and monitoring, testing, and documentation.

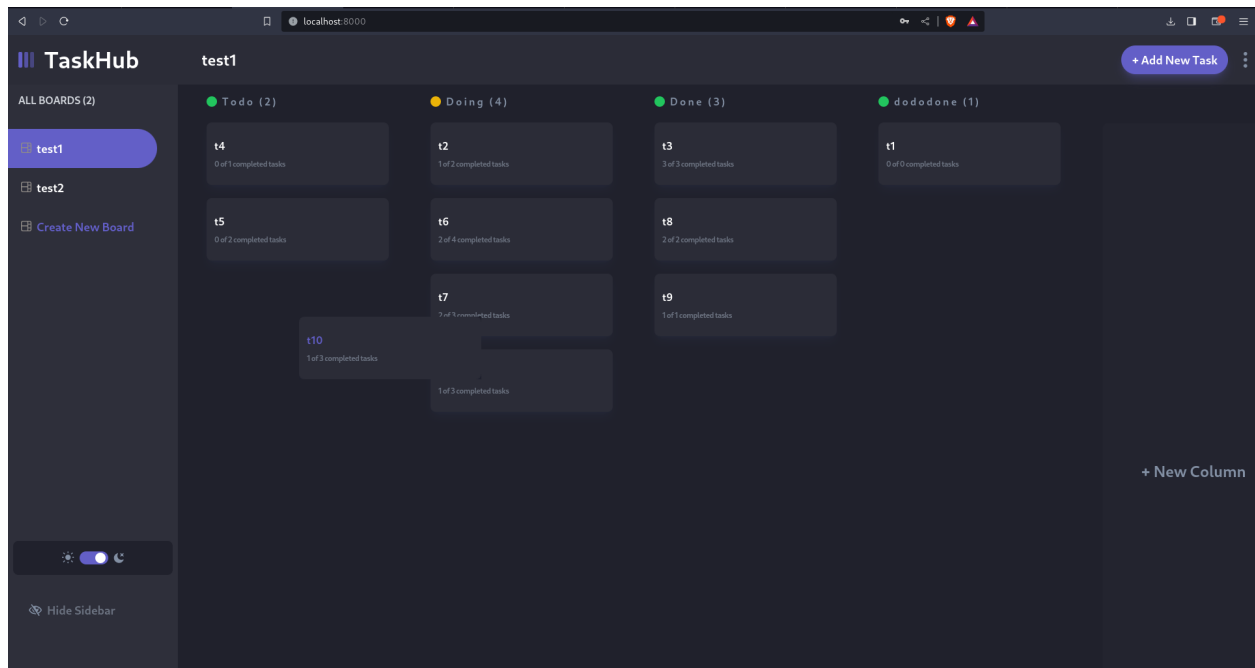
---

## 2. Project Idea and Innovation

The idea of this project came from a desire to streamline task management for both individuals and small teams. As someone who juggles many competing priorities on a daily basis, I found existing applications lacking in key areas such as intuitive interfaces and cross-device compatibility. The overarching goal was to develop a web application that could serve as a one-stop solution for organizing work, no matter the context or environment.



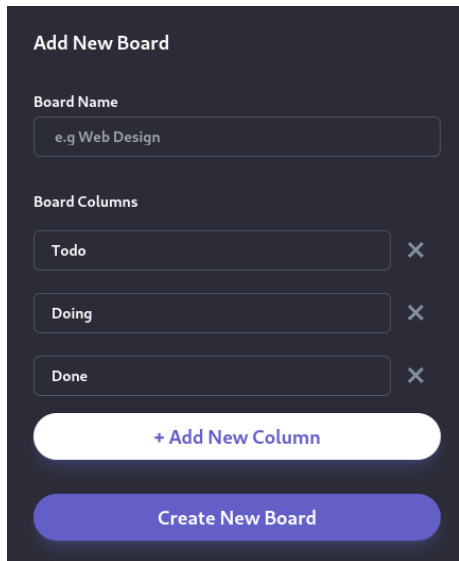
The target users are envisioned as knowledge workers, freelancers, small businesses and project teams who rely on workflows. Common usage scenarios include individuals tracking personal to-dos, freelancers coordinating client work, small marketing agencies collaborating on campaigns, and startup founders managing product development. The application aims to support both individual and shared board usage, empowering effective task management in a variety of professional and organizational settings.



What sets this project apart is the emphasis placed on user experience and productivity. Inspired by applications like Trello and Jira, an intuitive drag-and-drop interface allows users to structure their work visually and update status with ease. The responsive design ensures usability across any device while collaborative sharing capabilities foster teamwork. Advanced features like subtasks, attachments and custom fields lay the foundation for capturing diverse workstreams in a single, organized digital workspace. Ultimately, the goal was to develop a best-in-class solution optimized for productivity and satisfaction.

---

## 3. User Interface

A dark-themed form titled "Add New Board". It has a "Board Name" input field with the placeholder "e.g Web Design". Below it is a "Board Columns" section with three input fields: "Todo", "Doing", and "Done", each with a close button (X). At the bottom of the columns section is a button "+ Add New Column". At the very bottom is a large blue button "Create New Board".

Add New Board

Board Name

e.g Web Design

Board Columns

Todo X

Doing X

Done X

+ Add New Column

Create New Board

The user interface was designed with a focus on simplicity, visual clarity and productivity. Inspired by applications like Trello, the core experience revolves around boards containing columns for different task statuses.

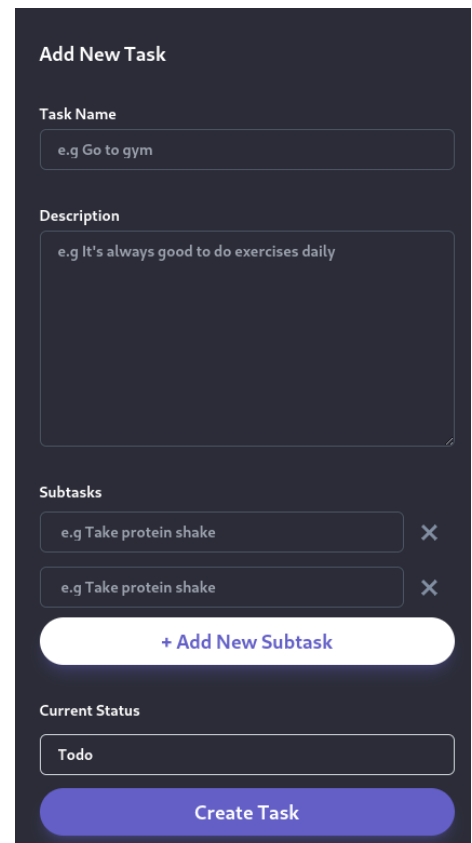
Users are greeted with a clean home screen displaying their boards. Tapping on a board title smoothly transitions to the board view. Here, tasks are represented by easily scannable cards stretching across columns for a "To Do", "In Progress" and "Done".

All major interactions were made intuitive through gestures. Tasks can be dragged between columns to update status with a simple flick. Cards also support vertical scrolling to accommodate detailed content.

Additional features are accessible through lean, text-based controls. Creating a new card opens a full-screen form allowing rich task details like descriptions, assignees, dates and file attachments. Users have full control to customize their workflow through board settings.

A consistent responsive design ensures the interface translates seamlessly across devices. On mobile, interactions were optimized for touch with large hit areas while preserving the familiar board structure.

Overall, the interface aims to keep users focused on their work through simple and fluid interactions built upon methodology. Visual cues help optimize productivity and satisfaction.

A dark-themed form titled "Add New Task". It has a "Task Name" input field with the placeholder "e.g Go to gym". Below it is a "Description" text area with the placeholder "e.g It's always good to do exercises daily". At the bottom of the description area is a close button (X). Below the description is a "Subtasks" section with two input fields: "e.g Take protein shake" and "e.g Take protein shake", each with a close button (X). At the bottom of the subtasks section is a button "+ Add New Subtask". At the very bottom is a "Current Status" input field with the placeholder "Todo". At the very bottom is a large blue button "Create Task".

Add New Task

Task Name

e.g Go to gym

Description

e.g It's always good to do exercises daily

Subtasks

e.g Take protein shake X

e.g Take protein shake X

+ Add New Subtask

Current Status

Todo

Create Task

---

## 4.Features and Functionalities

Users should be able to:

- View the app's ideal layout based on the screen size of their smartphone.
- View all of the page's interactive elements' hover states.
- Make, read, edit, remove, and manage tasks and boards.
- When attempting to add or edit boards and tasks, get form validations
- Move jobs between columns and mark subtasks as completed.
- Turn the board sidebar on or off.

Boards:

- The selected board will appear when you click on any of the other boards in the sidebar.
- The "Add New Board" popup appears when you click "Create New Board" on the sidebar.
- Modifying details is possible in the "Edit Board" model that appears when you click on the "Edit Board" dropdown menu.
- For the Add/Edit Board modals, columns are added and removed.
- A board that is deleted also loses all of its tasks and columns and has to be confirmed.

Columns:

- Before tasks may be added, a board must have at least one column. The header's "Add New Task" button is inactive if there are no columns.
- To add columns, click "Add New Column" to enter the "Edit Board" box.

Tasks:

- A new job is added at the bottom of the appropriate column.
- When a task's status is updated, it gets moved to the appropriate column.

Bonus:

- It is possible to drag and drop the jobs into a different column.

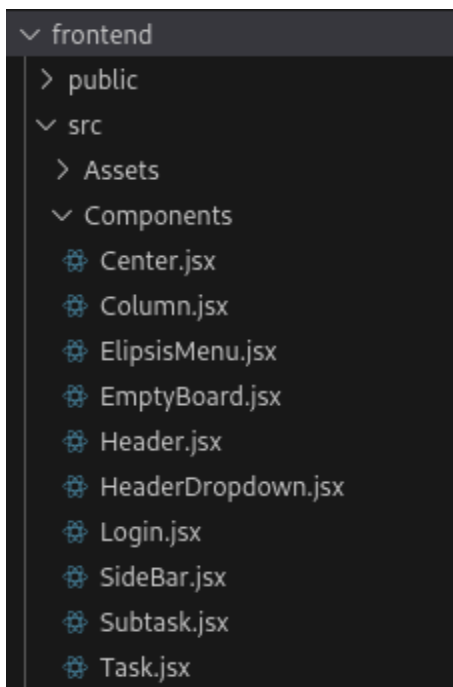
---

## 5. Architecture and Technologies

### 5.1 Frontend



On the frontend, React handles component-based views and Tailwind CSS provides responsive design. Users interact with an intuitive UI without page reloads for a smooth experience.



```
frontend > src > # index.css > ...
1  @import "tailwindcss/base";
2  @import "tailwindcss/components";
3  @import "tailwindcss/utilities";
4
5  body {
6    margin: 0;
7    font-family: system-ui, -apple-system, BlinkMacSystemFont,
8    -webkit-font-smoothing: antialiased;
9    -moz-osx-font-smoothing: grayscale;
10 }
11
12 code {
13   font-family: source-code-pro, Menlo, Monaco, Consolas, 'Cousine',
14   monospace;
15 }
16
17 @layer components {
18   .button {
19     @apply bg-[#635fc7] py-2 px-4 rounded-full text-white text-sm;
20   }
21 }
22
23 .dropdown{
24   background-color: #00000080;
25 }
26
```

The user interface of the frontend is created using HTML, CSS, and JavaScript. Component styling and layout on the page are accomplished with CSS. JavaScript manages user interactions and DOM manipulation. To create reusable user interface elements, a component-based methodology is used. As a result, the code is readable, modular, and manageable even as the application's size and complexity increase.

The application uses Redux Toolkit for state management. Redux is made simpler by abstracting away a large portion of the boilerplate code required for common Redux use cases. Action types and handler methods can be automatically generated based on the

---

state structure thanks to features like `createSlice`. This lowers the possibility of errors in manual code and enhances the developer experience. The store provides centralized access to the state from any location. When necessary, selectors are used to get particular state slices. All things considered, Redux Toolkit facilitates the development of scalable Redux apps.

```
frontend > src > Redux > JS boardsSlice.js > ...
1  import { createSlice, current } from "@reduxjs/toolkit";
2  import data from "../Data/data.json";
3
4  const boardsSlice = createSlice({
5    name: "boards",
6    initialState: data.boards,
7    reducers: {
8 >     addBoard: (state, action) => { ...
18    },
19 >     editBoard: (state, action) => { ...
24    },
25 >     deleteBoard: (state) => { ...
28    },
29 >     setBoardActive: (state, action) => { ...
36    },
37 >     addTask: (state, action) => { ...
44    },
45 >     editTask: (state, action) => { ...
66    },
67 >     dragTask: (state, action) => { ...
73    },
74 >     setSubtaskCompleted: (state, action) => { ...
81    },
82 >     setTaskStatus: (state, action) => { ...
93    },
94 >     deleteTask: (state, action) => { ...
99    },
100 >     returnData: (state, action) => { ...
102    }
103  },
104  });
105
106  export default boardsSlice;
107  |
```



---

## 5.2 Backend



Express framework and Node.js are used in the backend development process. It offers a REST API that may be used by the frontend. Controller functions are used to manage requests, and routes are defined using Express. It makes it possible to concentrate on the business logic by abstracting away the difficulties of HTTP. Asynchronous code that doesn't block is used to process several requests at once without compromising performance.

```
const server = express();

var accessLogStream = fs.createWriteStream(path.join(__dirname, 'logs.log'), {flags: 'a'});

server.use(cors());
server.use(morgan('combined', {stream: accessLogStream}));
server.use(bodyParser.json());
```

Using Morgan middleware, HTTP requests can be recorded. It offers helpful details like the URL, request method, and response status code, among others. Debugging, performance monitoring, and security all benefit from this. For examination, the logs can be streamed to a file, database, or outside services.

To add, read, update, and remove tasks, use the CRUD operations exposed by the API endpoints. Morgan logging increases insight into the use of the API.

```
server.post('/users', async (req,res) => {
  let user = new User();
  user.email = req.body.email;
  user.password = req.body.password;
  const doc = await user.save();
  res.json(doc);
})

server.get('/users', async (req,res) => {
  const docs = await User.find({});
  res.json(docs);
})
```

```
> server.get('/users/:id/data', async (req,res) => { ...
  })

> server.post('/users/:id/data', async (req,res) => { ...
  })

server.post('/users/:id/data1', async (req,res) => {
  const doc = await Data.findOneAndReplace({ _userId: req.body._userId}, {
    data: req.body.data,
    _userId: req.body._userId
  })
  res.json(doc);
})
```

---

## 5.3 Database



A MongoDB database is used to store application data in JSON-like documents. It offers a flexible schema that can accommodate changes easily as requirements evolve. Embedded documents help reduce the need for multiple queries to fetch related data. MongoDB scales well as it supports horizontal scaling by adding more machines. It can handle a large number of requests and a large dataset. Integrating with Node.js also provides a seamless development experience.

```
const UserSchema = new mongoose.Schema({
  email: String,
  password: String
})

const taskSchema = new mongoose.Schema({
  title: String,
  description: String,
  status: String,
  subtasks: [{
    title: String,
    isCompleted: Boolean
  }]
});

const columnSchema = new mongoose.Schema({
  name: String,
  tasks: [taskSchema]
});
```

```
const boardSchema = new mongoose.Schema({
  name: String,
  isActive: Boolean,
  columns: [columnSchema]
});

const DataSchema = new mongoose.Schema({
  data: [boardSchema],
  userId: {
    type: mongoose.Types.ObjectId,
    required: true
  }
});
```

Overall, this modern stack leverages the strengths of each technology: React and Tailwind for the UI, Node.js/Express for API logic, MongoDB for data, and various tools to streamline the development process. The modular architecture also makes the application extensible. The frontend, backend and database components work in harmony to deliver responsive performance.

---

## 6. Version Control - GitHub



***Github repository link:***

<https://github.com/Bhuvan506/TaskHub>

Git is used for version control and collaboration on the project. A local Git repository tracks changes to code which can then be shared to a remote repository on GitHub.

```
git init
git remote add origin https://github.com/Bhuvan506/TaskHub.git
```

Commit messages provide descriptive summaries of changes at a high level.

```
git add <files>
git commit -m "commit message"
git pull origin master
git push origin master
```

This helps maintain an audit trail and understand the purpose of changes.

The master branch represents the mainline of development. Feature branches enable working on new functionality in isolation. When complete, code is merged back into master after pull requests and review.

```
git checkout master
git checkout -b "<branch_name>"
```

Tagging important releases using semantic versioning provides checkpoints that can be easily referred back to. The GitHub repository facilitates collaboration and code reviews by multiple developers. Issues can be logged, discussed and tracked on the platform.

Overall, Git and GitHub provide robust version control capabilities for the project. Code history is preserved, changes can be easily compared and rolled back if needed. A clean branching model supports parallel development while maintaining a stable master branch.

---

## 7. Testing



# SUPERTEST

In order to guarantee code quality and avoid regressions, extensive testing is essential. Jest is used for both unit and integration testing on the backend.

Jest enables the writing of tests using test blocks and descriptive explanation blocks that resemble code. This facilitates the writing and comprehension of tests.

```
describe('User routes', () => {  
  it('Creates a new user', async () => {  
    const res = await request(server)  
      .post('/users')  
      .send({  
        email: 'test@test.com',  
        password: 'password'  
      });  
    expect(res.statusCode).toEqual(200);  
  }, 10000);  
  
  it('Gets all users', async () => {  
    const res = await request(server)  
      .get('/users');  
    expect(res.statusCode).toEqual(200);  
    expect(res.body).toBeDefined();  
    uid = res.body._id;  
  }, 10000);  
});
```

```
describe('Data routes', () => {  
  it('Gets data for a user', async () => {  
    const res = await request(server)  
      .get('/users/:id/data');  
    expect(res.statusCode).toEqual(200);  
  }, 10000);  
  
  it('Creates data for a user', async () => {  
    const res = await request(server)  
      .post('/users/:id/data1')  
      .send({  
        _userId: uid,  
        data: []  
      });  
    expect(res.statusCode).toEqual(200);  
  }, 10000);  
});
```

Unit tests, as shown in the diagram, isolate and test specific functions or class methods from outside dependencies. This ensures that prior to integration, business logic is implemented correctly. By simulating external dependencies, mocking helps to avoid test failures caused by variables not related to the function under test.

The tests are broken into two describe blocks - one for the user routes and one for the data routes.

The user routes tests:

- Create a new user to get an ID
- Get all users to retrieve the newly created user ID

---

The data routes tests:

- Get data for a user by ID
- Create new data for a user by ID

Before and after hooks are used to initialize and close the MongoDB connection. This allows the tests to interact with the database.

The expect statements verify the response status codes and body contents match what is expected.

By testing across routes and layers, this validates the end-to-end functionality and interactions between the API, database and models. The use of `async/await` and `timeouts` also handles asynchronous operations.

```
jayam@DESKTOP-PB8TL0M: /mnt/d/TaskHub$ cd node-server
jayam@DESKTOP-PB8TL0M: /mnt/d/TaskHub/node-server$ npm test

> server@1.0.0 test
> jest --forceExit

(node:7020) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
console.log
  Server is running on port 4000

    at Server.log (index.js:107:13)

console.log
  db connected

    at log (index.js:20:13)

PASS ./auth.test.js (8.633 s)
  User routes
    ✓ Creates a new user (348 ms)
    ✓ Gets all users (57 ms)
  Data routes
    ✓ Gets data for a user (20 ms)
    ✓ Creates data for a user (14 ms)

Test Suites: 1 passed, 1 total
Tests: 4 passed, 4 total
Snapshots: 0 total
Time: 9.216 s, estimated 11 s
Ran all test suites.
Force exiting Jest: Have you considered using `--detectOpenHandles` to detect async operations that kept running after all tests finished?
jayam@DESKTOP-PB8TL0M: /mnt/d/TaskHub/node-server$
```

Integration tests combine units to verify cross-layer functionality, including database interactions and API routes.

The test suite is run on every code change using a CI/CD pipeline to catch errors early. Over time, tests serve as living documentation that also prevents accidental regressions.

Overall, testing with Jest ensures high code quality, reliability and saves debugging time by verifying changes did not break existing functionality.

## 8. Containerization



The application stack is containerised using Docker to enable simple deployment and uniform runtimes across environments. The build procedure and runtime dependencies are declaratively defined in a Dockerfile. This makes it possible to recreate the identical environment from scratch wherever.

```
node-server > Dockerfile > ...
1 FROM node:latest
2 COPY . .
3 WORKDIR /app
4 RUN npm install
5 CMD ["npm", "start"]
```

```
frontend > Dockerfile > ...
1 FROM node:latest
2 WORKDIR /app
3 # ENV PATH /app/node_modules/.bin:$PATH
4 COPY package.json .
5 # COPY package-lock.json .
6 RUN npm install
7 COPY . .
8 EXPOSE 8000
9 CMD ["npm", "run", "dev"]
```

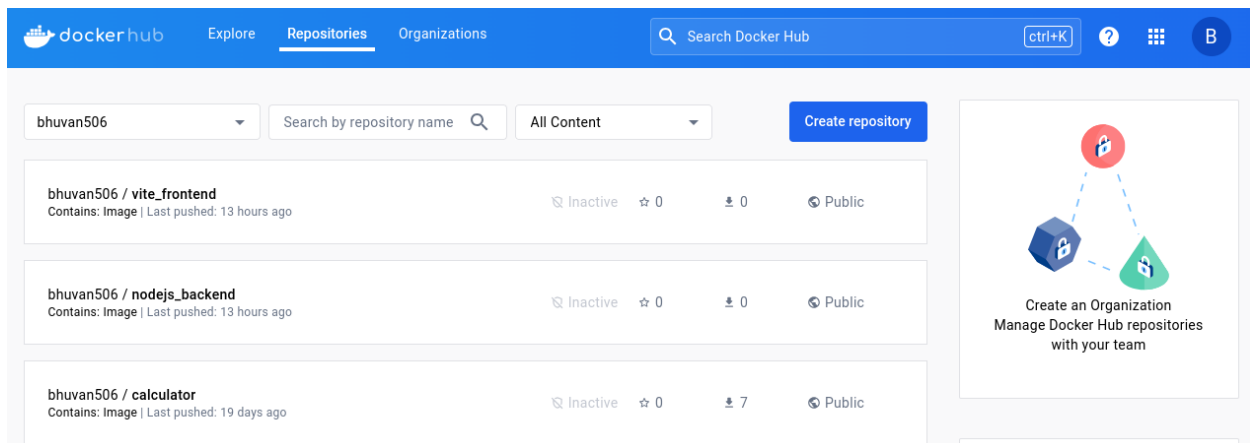
For smaller final images, the multi-stage Dockerfile separates building from operating. Dependencies and Node.js are installed independently of the layers of the production environment.

```
home > bhuvan > spe_main_project_ansible > docker-compose.yml
1 version: '3.8'
2 services:
3   mongodb:
4     image: mongo
5     container_name: database
6     volumes:
7       - /home/bhuvan/Desktop/spe/database:/data/db
8     restart: unless-stopped
9   server:
10    image: nodejs_backend
11    container_name: backend
12    ports:
13      - '4000:4000'
14    restart: unless-stopped
15   client:
16    image: vite_frontend
17    container_name: frontend
18    ports:
19      - '8000:8000'
20    restart: unless-stopped
```

Docker Compose is used to connect services; it establishes dependencies and interactions between containers via YAML configuration files.

---

Version numbers are applied to images in order to enable repeatable deployments. Docker Hub receives the production image deployment for simple pulls and distribution.



Dependencies and the application are isolated via containers. Resources are automatically assigned and decommissioned. Application deployment is more secure, scalable, and portable thanks to containerisation.

Docker Compose is used to run three containers - one each for the backend, frontend and database. It leverages Docker's ability to run multiple isolated processes/services on the same host.

The docker-compose.yml file defines the three containers - backend, frontend and mongo-db (the database). This allows the three components to run as independent containers but still communicate seamlessly through the network. The frontend fetches data by calling the backend API. The backend accesses the database.

By linking the containers, Docker Compose handles starting, restarting and stopping them as a cohesive unit. It also scales services horizontally by increasing replicas.

---

## 9. Deployment



The application containers' automated deployment to distant servers is accomplished with Ansible. YAML playbooks are used to provide and configure infrastructure as code.

Playbooks specify how to deploy an application in three different environments: development, staging, and production. For reusability, roles abstract common tasks.

```
deployment > ! deploy.yml
1  ---
2  - name: Pull Docker Images
3    hosts: all
4    vars:
5      ansible_python_interpreter: /usr/bin/python3
6    tasks:
7      - name: Pull server image
8        docker_image:
9          name: bhuvan506/nodejs_backend:latest
10         source: pull
11      - name: Pull client image
12        docker_image:
13          name: bhuvan506/vite_frontend:latest
14          source: pull
15      - name: Create new directory
16        shell: mkdir -p spe_main_project_ansible
17      - name: Copy Docker Compose
18        copy:
19          src: /var/lib/jenkins/workspace/SPE-Project/docker-compose.yml
20          dest: /home/bhuvan/spe_main_project_ansible/docker-compose.yml
21      - name: Stop Docker Compose
22        command: docker-compose down
23        args:
24          chdir: /home/bhuvan/spe_main_project_ansible
25      - name: Remove old images
26        shell: docker image rm bhuvan506/vite_frontend bhuvan506/nodejs_backend
27      - name: Run Docker Compose
28        command: docker-compose up -d
29        args:
30          chdir: /home/bhuvan/spe_main_project_ansible
31
```

Roles are defined to run the containers, pull images from the registry, install Docker, and configure the server .



---

Variables separate configurations unique to a certain environment, such as database connection strings, target hosts, and image tags. This facilitates the deployment of the same codebase across several environments.

```
deployment > ≡ inventory
1  localhost ansible_user=bhuvan
2  
```

Ansible ensures a secure deployment by operating over SSH without the need for an agent on distant devices. Idempotent playbooks only execute commands in response to changes in the infrastructure.

Using Git hooks, the automated pipeline initiates fresh deployments in response to code changes. Rollbacks are simple and involve going back to earlier configurations.

In conclusion, Ansible deployment makes it easier to release features, maintains consistency in the infrastructure, and enables quick provisioning of extra servers for growth.

---

## 10. Continuous Integration/Deployment



# Jenkins

The Continuous Integration and Delivery process is implemented by this Jenkins pipeline, which builds, tests, and deploys the application code automatically.

Jenkins uses the specified webhook to identify code changes that are pushed to the GitHub repository. The build process begins with this.

```
stage("Step 1: Git Clone"){
  steps{
    git branch: "master", url: "https://github.com/Bhuvan506/TaskHub.git"
  }
}
```

The configured Git branch's most recent code is examined in the first step. Before constructing containers, code is compiled and put through unit tests in the following step to ensure it passes testing. This aids in finding errors or malfunctions early on.

```
stage("Step 2: Testing"){
  steps{
    dir('node-server') {
      sh 'npm i'
      sh 'npm test'
    }
  }
}
```

When testing is successful, Docker is used to create container images for the frontend and backend codebases. Every project's Dockerfile specifies how to create deployment-ready, optimized images.

```
stage("Step 3: Build docker image"){
  steps{
    dir('node-server'){
      script{
        docker_image_server = docker.build "bhuvan506/nodejs_backend:latest"
      }
    }
    dir('frontend'){
      script{
        docker_image_client = docker.build "bhuvan506/vite_frontend:latest"
      }
    }
  }
}
```

---

After that, the images are uploaded up to the Docker Hub registry, enabling simple deployment across other locations. In this step, the most recent version is applied to the photos.







```
stage("Step 4: Push Docker image to hub"){
  steps{
    script{
      docker.withRegistry("", "DockerHubCred"){
        docker_image_server.push()
        docker_image_client.push()
      }
    }
  }
}
```

The most recent container images are automatically and consistently deployed to production servers via an Ansible script. It takes care of the necessary configuration adjustments as well as setting up and launching the containers.

```
stage("Step 5: Ansible Deployment"){
  steps{
    ansiblePlaybook becomeUser: null,
    colorized: true,
    credentialsId: 'localhost',
    disableHostKeyChecking: true,
    installation: 'Ansible',
    inventory: 'deployment/inventory',
    playbook: 'deployment/deploy.yml',
    sudoUser: null
  }
}
```

By putting this continuous process in place, developers can be confident that every deployment is tested and validated before it reaches users and they also receive rapid feedback on their code modifications. It enables dependable, quick software updates at This content is available for use at any moment thanks to automation.

## Credentials

T	P	Store ↓	Domain	ID	Name
		System	(global)	DockerHubCred	bhuvan506/***** (Docker Hub Credentials)
		System	(global)	localhost	bhuvan/***** (Localhost user login credentials)
		System	(global)	Github Personal Access Token	Personal Access Token

---

**Github Credentials:** For accessing private Github repositories.

**Docker Hub Credentials:** For logging into Docker Hub account and pushing Docker images.

#### Stage View

		Step 1: Git Clone	Step 3: Build docker image	Step 4: Push Docker image to hub	Step 5: Ansible Deployment
Average stage times: (Average full run time: ~4min 37s)		6s	3min 23s	1min 25s	2min 6s
#49 Dec 13 20:56	No Changes	4s	1min 30s	46s	40s
#48 Dec 13 20:38	1 commit	8s	3min 21s	2min 6s	1min 32s
#46 Dec 13 19:19	No Changes	9s	37s	26s	2min 2s

Running docker compose after build:

```
bhuvan@localhost:/media/bhuvan/DATA/TaskHub$ docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
87e04d121d8b   vite_frontend  "docker-entrypoint.s..." 13 hours ago   Up 13 hours   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp   frontend
f851e2ca6b7b   nodejs_backend "docker-entrypoint.s..." 13 hours ago   Up 13 hours   0.0.0.0:4000->4000/tcp, :::4000->4000/tcp   backend
6f11503b28eb   mongo         "docker-entrypoint.s..." 13 hours ago   Up 13 hours   27017/tcp                               database
```

## 11. Logging and Monitoring



Logging provides a multitude of information for tracking the performance of applications and infrastructure. The Node.js backend sends request logs to the Morgan middleware, which records information such as the HTTP method, URL, response status, and more.

A distributed, RESTful search and analytics engine is called Elasticsearch. Logs may be searched using free-text, time intervals, or fields. This aids in problem diagnosis and isolation.

Kibana uses the Elasticsearch index to provide customisable tables, dashboards, histograms, and other views for the purpose of visualizing logs. Developers can obtain valuable insights by utilizing robust analytics and searching features.

All things considered, the ELK stack centralizes logs from many sources for effective analytics, searching, and visualization. When paired with Morgan, it offers complete application monitoring and transparency.

### Log file stats(data fields):

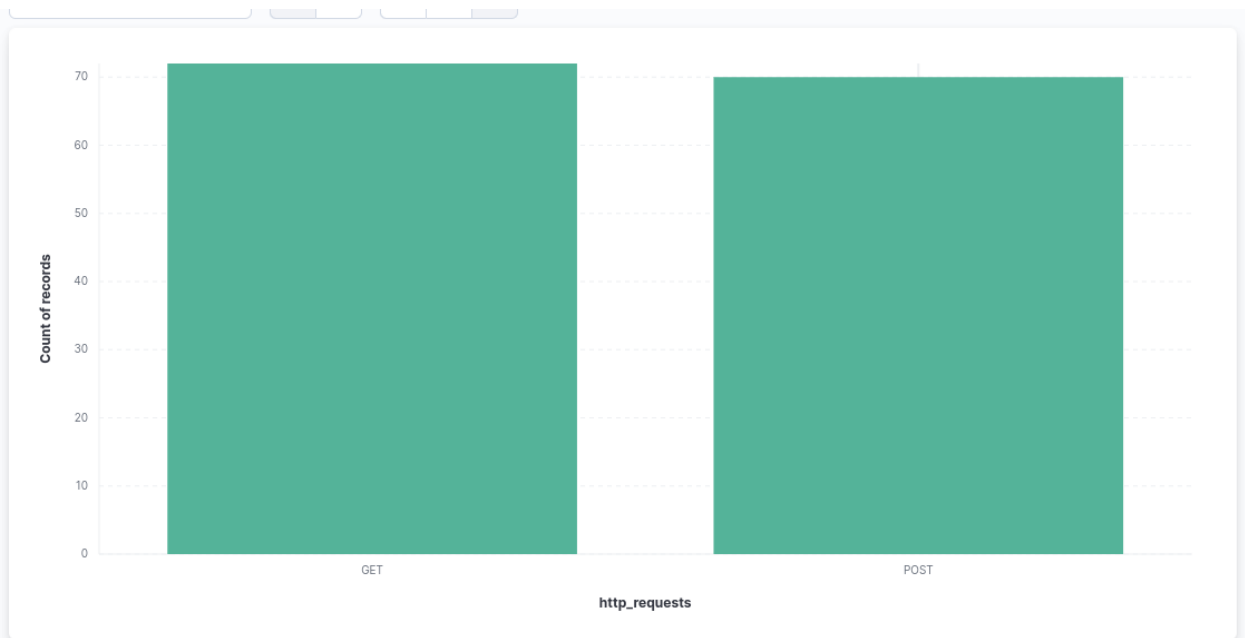
File stats			
All fields 12 of 12 total		Number fields 3 of 3 total	
		Field name 12	Field type 5
Type	Name	Documents (%)	Distinct values
agent	agent	141 (100%)	3
auth	auth	141 (100%)	1
bytes	bytes	129 (91.49%)	38
clientip	clientip	141 (100%)	2
httpversion	httpversion	141 (100%)	1
ident	ident	141 (100%)	1
message	message	141 (100%)	141
referrer	referrer	141 (100%)	2
request	request	141 (100%)	3
response	response	141 (100%)	2
timestamp	timestamp	141 (100%)	56
verb	verb	141 (100%)	2

## Visualizations:

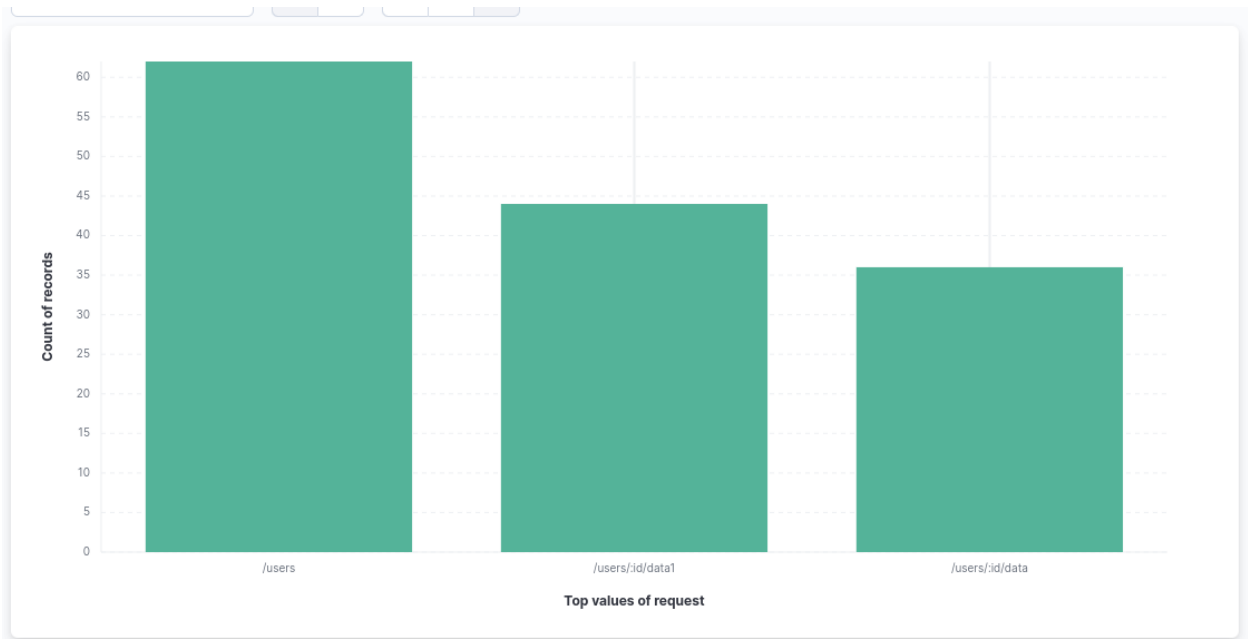
### 1) Time series view



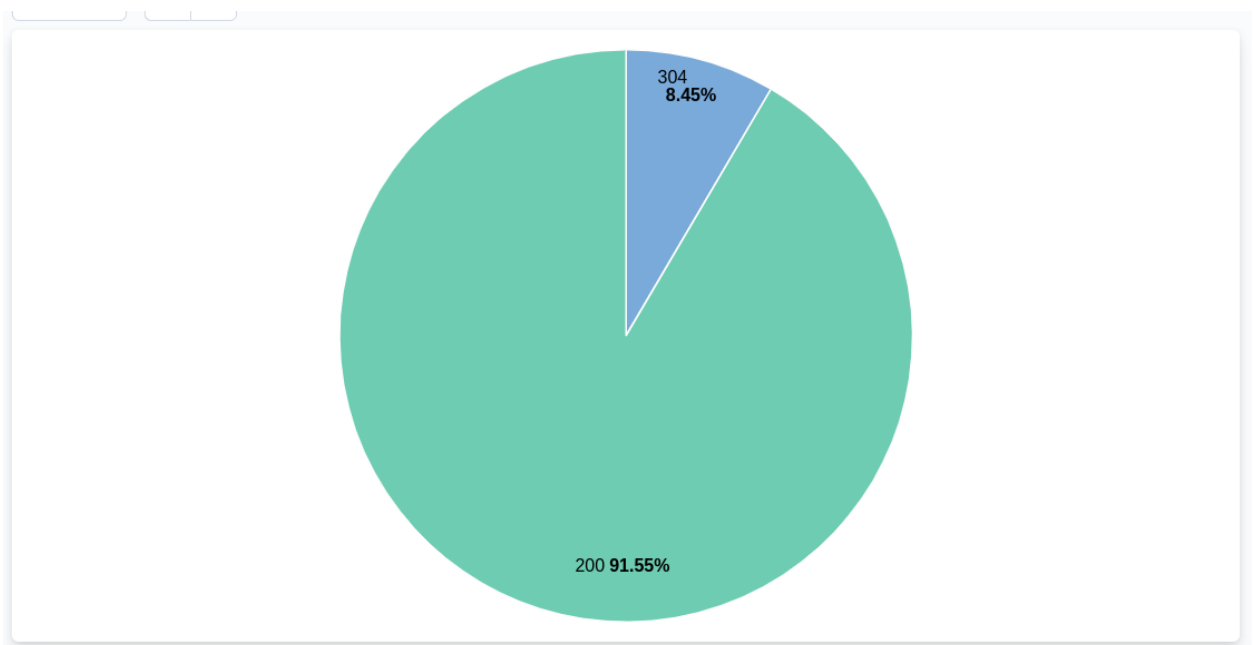
### 2) http\_requests



### 3) Path fields



### 4) Status code



---

# API Documentation

Endpoint	HTTP Method	Input	Description
/users	POST	email, password	The user registration process is handled via this route. With the email address and password supplied in the request body, a new User document is saved to the database.
/users	GET	-	Pulls all user documents out of the database and sends them back in the reply. helpful for viewing all registered users on an admin interface.
/users/:id/data	GET	-	Obtains every Data document linked to the user ID that was supplied in the route parameter. This gives the user's individual board and task data back.
/users/:id/data	POST	userId	Enables the database to store a new Data document. The information is linked to the user ID that was supplied in the request body and route parameter. This enables a user to save the original board data.
/users/:id/data1	POST	userId, data	Locates and replaces the user ID-matching existing data document. This enables the updating or changing of task data in the database and a user's board.



---

# Result

*Sign Up page:*

TaskHub

Welcome

Email address

Password

Confirm Password

Sign Up

Already registered?Login

*Sign In page:*

TaskHub

Welcome

Email address

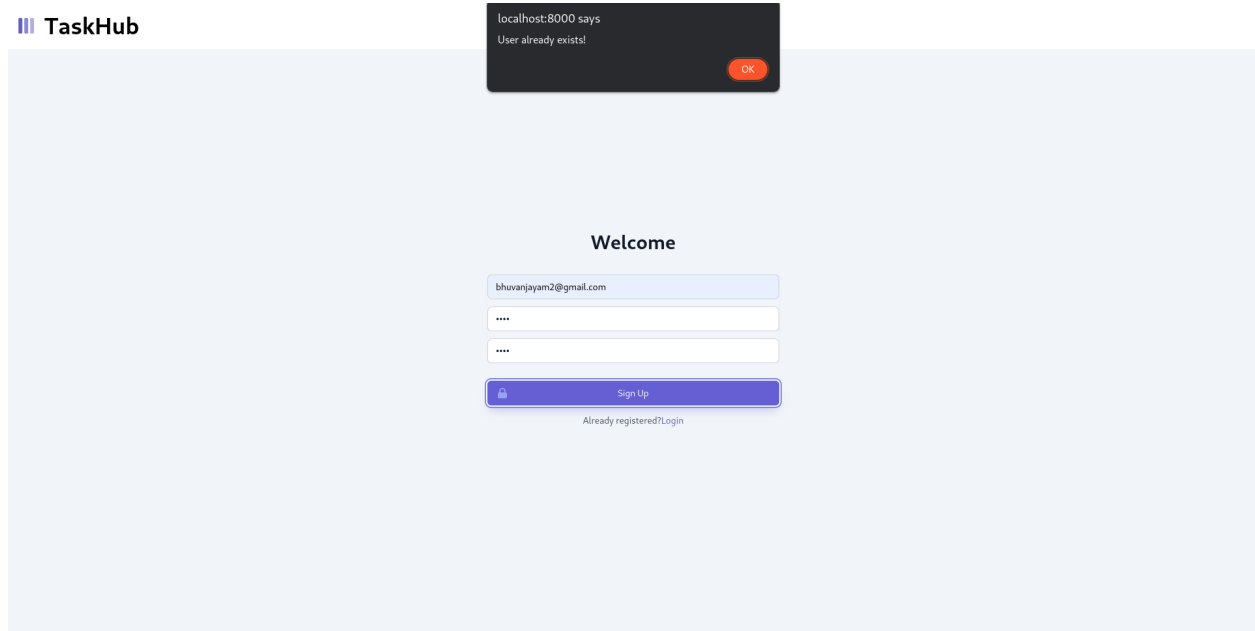
Password

Forgot your password?

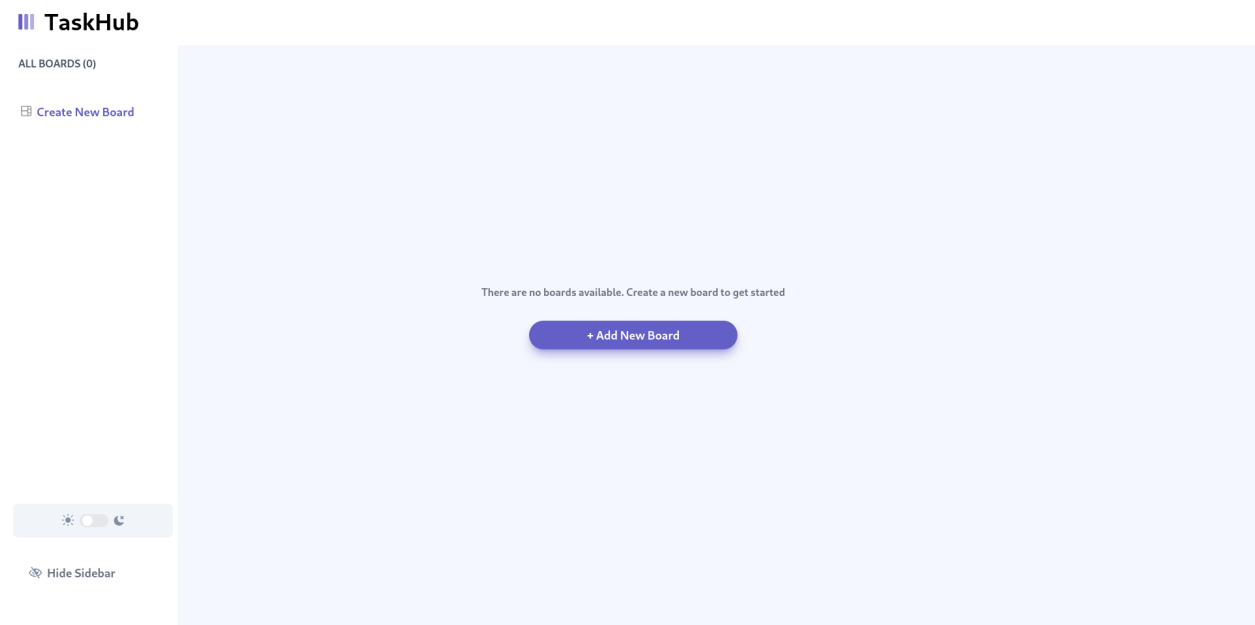
Login

Don't have an account?Sign Up

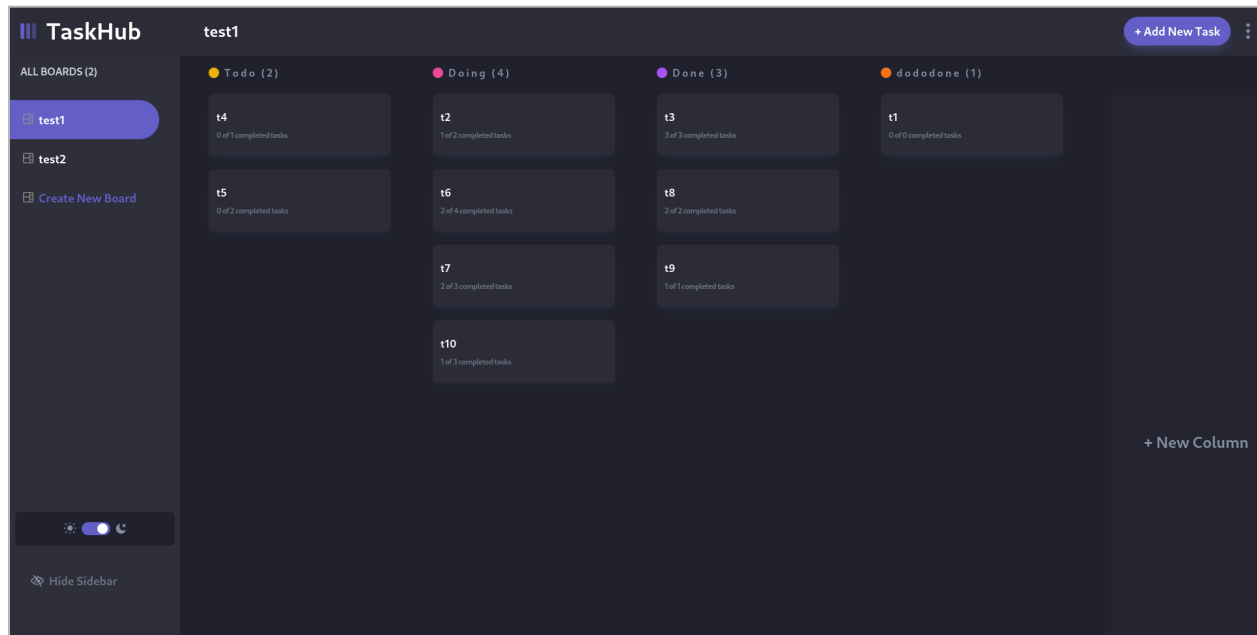
*Alerts: (wrong password, user already exists, passwords don't match, etc.):*



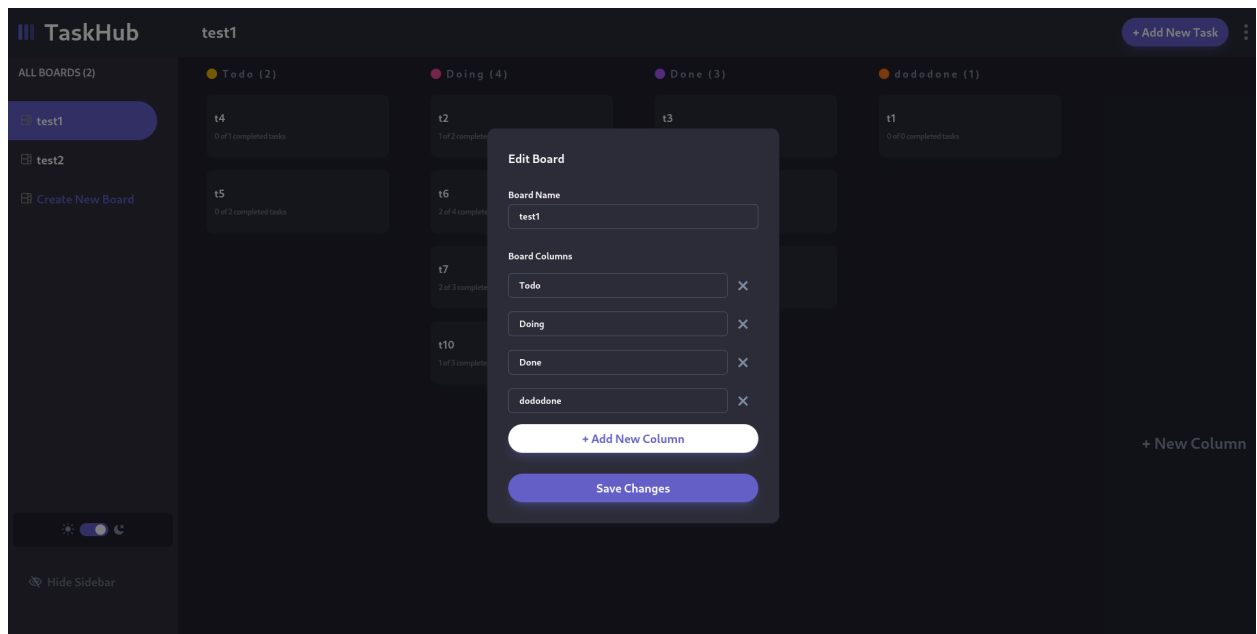
*Empty Board page:*



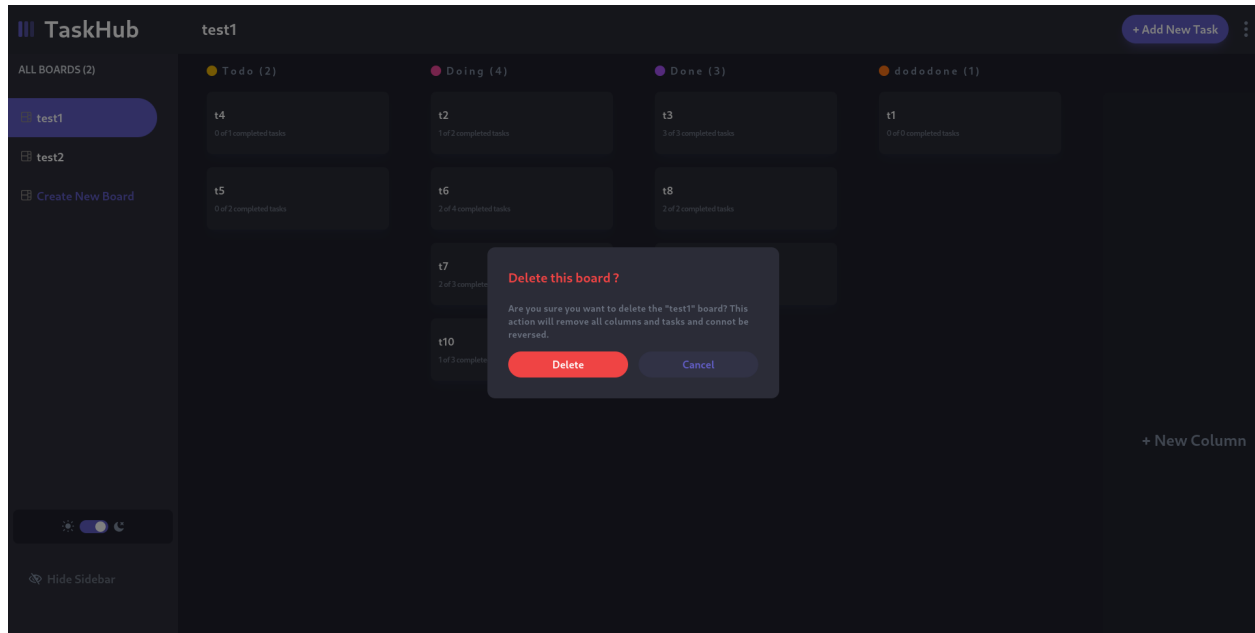
## Board view:



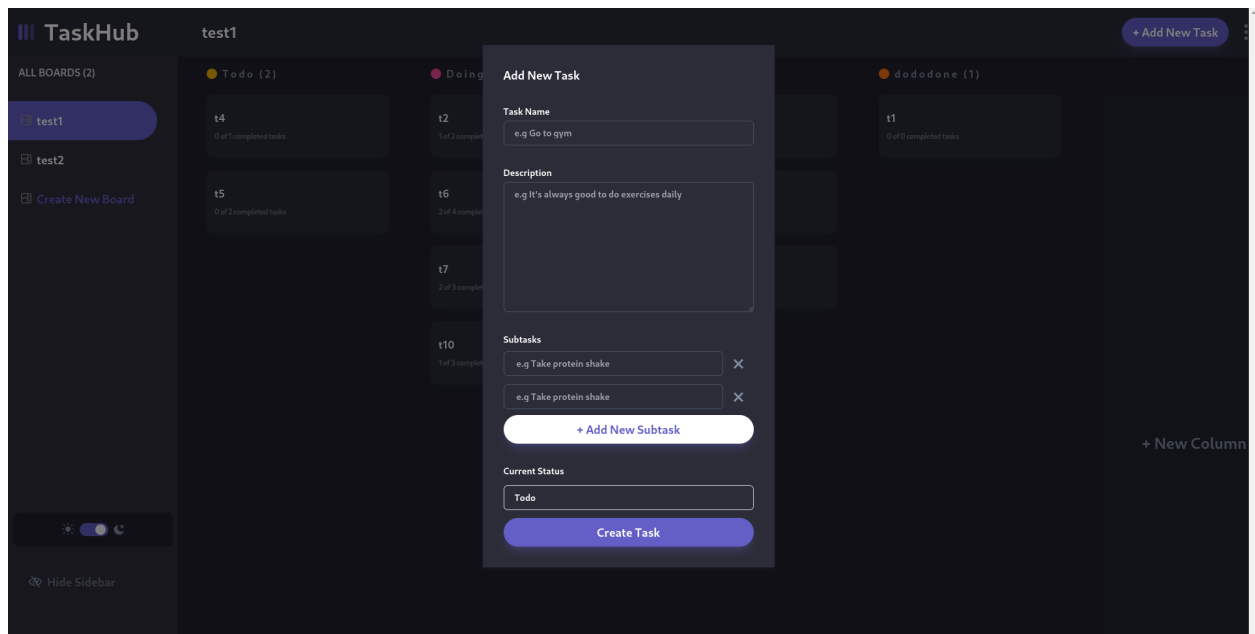
## Add/Edit Board:



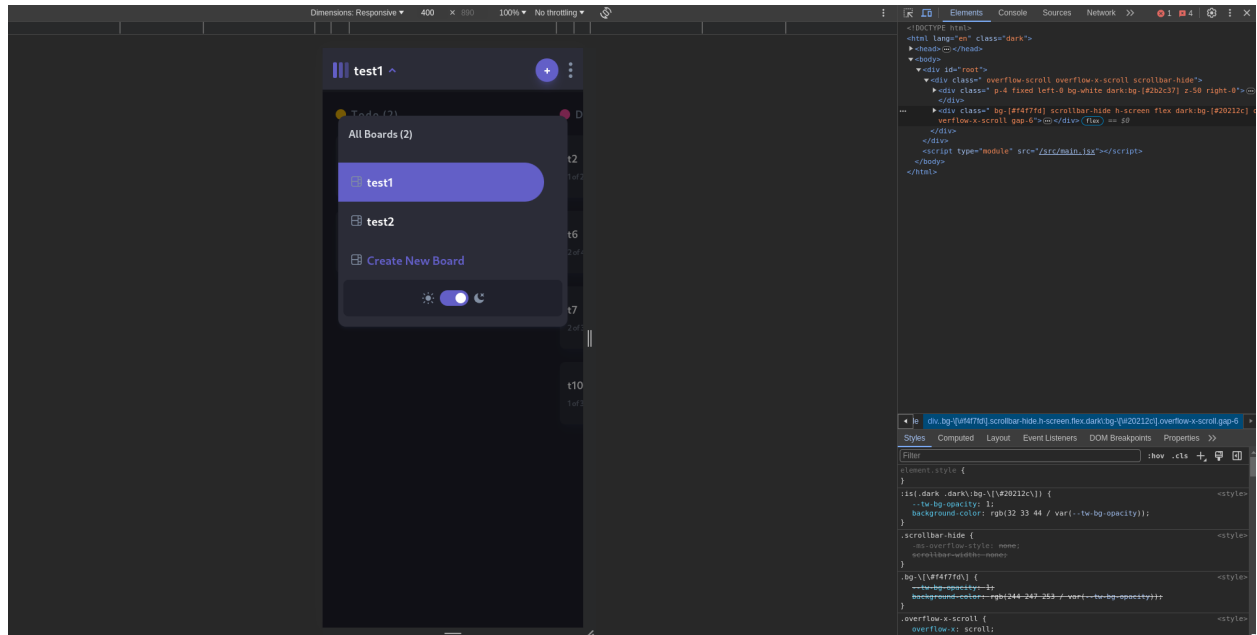
Delete board/task:



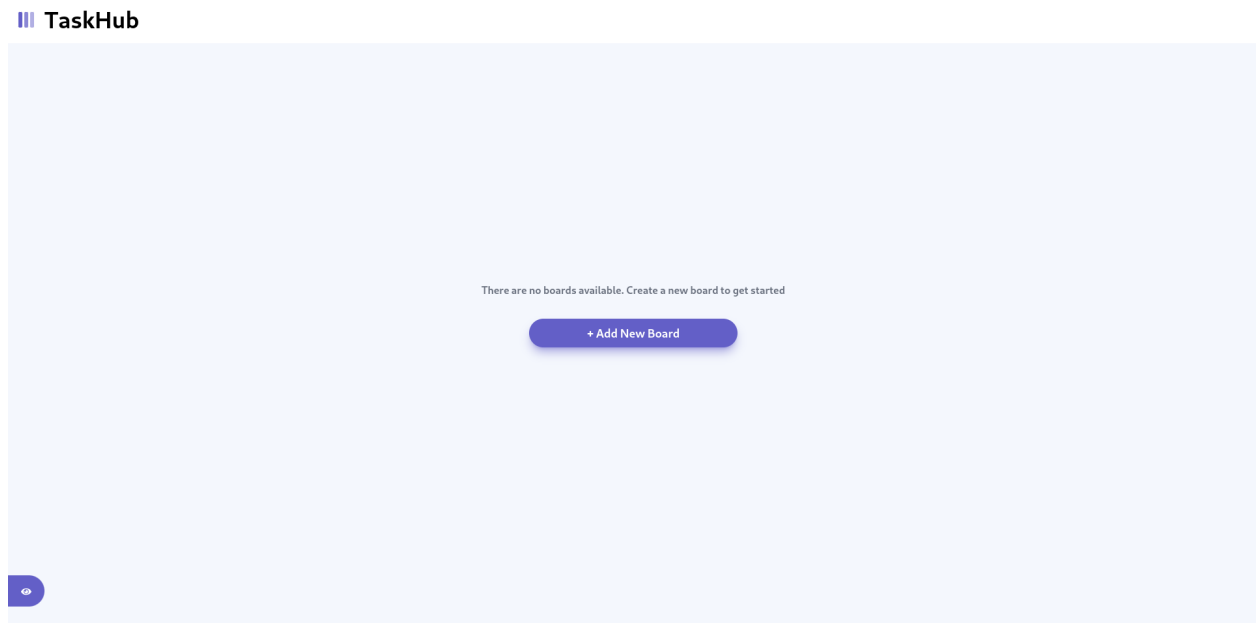
Add/Edit task:



*Dropdown Menu(instead of sidebar) for small screens:*



*Sidebar closed view:*



---

## Additional Tools Used

### 1) **Redux toolkit:**

Redux Toolkit made state management easier by giving Redux an easier-to-use API. The creation of action types and handler methods was automated by features like createSlice and createAsyncThunk. This lessened the possibility of bugs and enhanced the developer experience.

*Ref link:*

<https://redux-toolkit.js.org/>

### 2) **HTML Drag and Drop API:**

The HTML5 Drag and Drop API made it possible to reorganize jobs in the browser with ease using drag and drop. Because native browser capabilities were leveraged, no plugins or libraries were needed. To adjust a task's priority, users only need to drag it up or down.

*Ref link:*

[https://developer.mozilla.org/en-US/docs/Web/API/HTML\\_Drag\\_and\\_Drop\\_API](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API)

### 3) **Tailwind CSS:**

Tailwind CSS provided a utility-first method for quickly styling elements with borders, spacing, text styles, and other features. In most cases, it eliminated the requirement for specific CSS rules. Its approach to low-level atomic design allowed for rapid iteration.

*Ref link:*

<https://tailwindcss.com/>

---

## Future Scope

Improving collaboration with tools like in-app notifications, task attachments, and comments are near-term priorities. Users would find it easier to find pertinent information from their workflow history with advanced search options. By using APIs to integrate third-party solutions, services may be able to work together more effectively.

In terms of technology, high volume utilization would be enabled via a scalable architecture that makes use of microservices. Using a framework such as GraphQL could simplify the process of retrieving data for intricate queries. Users would be able to view their boards even on the go using a dedicated mobile app.

Deep insights may be gained through data analytics and reporting capabilities. Continuous improvement would be enabled via dashboards displaying indicators such as team productivity, bottleneck locations, and task completion rates. Planning from the top down and the bottom up would be made easier by integration with project management tools.

In the long run, AI-driven fields such as predictive modeling have chances to suggest the optimum course of action based on past performance. Paper documents could be digitalized and made searchable with the help of computer vision APIs. The way people engage with information could be completely changed by virtual and augmented reality.

Although lofty, there are lots of opportunities in front of you. Long-term value delivery from this platform will be ensured by properly utilizing the newest technology and remaining sensitive to changing user needs. Coming up with creative solutions is a never-ending path of problem-solving and learning.

---

## Conclusion

A well-rounded tech stack is demonstrated by this full-stack task management application, which can be used to create scalable and reliable online services. The frontend interface and backend API were constructed using the MERN stack, which consists of MongoDB, Express, React, and Node.js.

Best practices were followed in the implementation of important features including user authentication, data modeling, routing, testing, and deployment. Users may collaboratively track progress over time and graphically arrange activities on customisable boards thanks to core features.

The programme and associated dependencies are packaged for simple deployment thanks to Docker containerisation. Jenkins continuous integration automates the build, test, and release processes. ELK is used to gather application logs and analytics for performance tracking and problem-solving.

All things considered, the project provides a strong base and a point of entry for the addition of new features such as reports, permissions, notifications, and integrations. It also emphasizes how utilizing contemporary technologies and development methodologies results in long-term productivity, maintainability, and high-quality code. Future enhancements will concentrate on increasing its usefulness, improving accessibility, and optimizing performance.