Priority Task Scheduling Dashboard - PTSD

Technical Documentation

Author: Antoni Iwan

April 27, 2025

Contents

1 High-Level Overview

1.1 Project Summary:

The **Priority Task Scheduling Dashboard (PTSD)** is an application designed to create and manage task lists. The app offers two interfaces: a web interface and a terminal interface, both allowing users to add new tasks to the list. The application operates in a Docker environment, ensuring easy installation and deployment. MongoDB is used to store the list of tasks, providing an efficient and scalable solution for data storage.

1.2 Components:

- **Backend:** Flask (Python) - The server-side component that handles the API, allowing the addition of new tasks to the task list.
- **CLI Client:** Python Terminal App - A terminal-based application that allows users to add tasks from the command line interface.
- **Web Client:** React with Vite - A Web interface through which users can manage their tasks.
- **Database:** MongoDB - A NoSQL database used to store task data.
- **Infrastructure:** Docker Compose - A containerized environment that simplifies the application's setup and execution, both locally and in the cloud.

1.3 Limitations:

- **No Editing or Deleting Tasks:** Users can only add new tasks; there is no option to edit or delete existing tasks.
- **No Filtering or Sorting:** The application does not support filtering or sorting tasks based on any criteria. The task list is presented as-is, with no advanced functionality for managing tasks in a more granular way.

1.4 Environment:

The application is designed to run in Docker containers, providing an easy setup and deployment process. It is fully functional in both local and cloud environments. To run the application, users need to have Docker and Docker Compose installed.

2 Business Requirements

2.1 User Features:

- **Account creation:** Users can create accounts to provide connection between the web version and the terminal version.
- **Task list creation:** Users can add new task list to provide more organization between tasks
- **Task Creation:** Users can add new tasks to the task list, helping them keep track of things to do.
- **Simple User Interface:** The application provides two interfaces: a web interface and a terminal interface. Users can choose which method best suits them to manage tasks.

2.2 Use Cases:

- **Creating an Account:** A user provides a username and password to create an account and then logs in. After logging in, the user gains the ability to create a task list and add tasks within it.
- **Adding a New Task:** A user enters a title for a new task and adds it to the task list. Once the task is added, it becomes available for review, but editing is not possible. The task is simply appended to the list without any modifications.
- **Deleting a Task:** A user can select a task to delete it, both in the terminal and web versions of the application.
- **Marking and Unmarking Tasks:** A user can mark a task as completed and also unmark it, allowing for flexible management of task statuses.

2.3 User Interface Overview:

- **Web Interface Main Screen:** The main screen in the web interface displays the task list, with an option to add new tasks. Users can easily navigate through the tasks, but cannot modify them beyond adding new entries and deleting existing ones.
- **Terminal Interface:** The terminal interface allows users to enter commands for adding new tasks directly from the command line. This alternative interface caters to users who prefer a terminal-based workflow. Additionally, the terminal version provides a weather-check feature to help users plan tasks more effectively by considering external factors such as weather conditions.

2.4 Interfaces:

The application provides two key interfaces:

- **Web Interface:** A simple, responsive interface built using React and Vite. It displays the list of tasks and provides a user-friendly way to add new tasks.

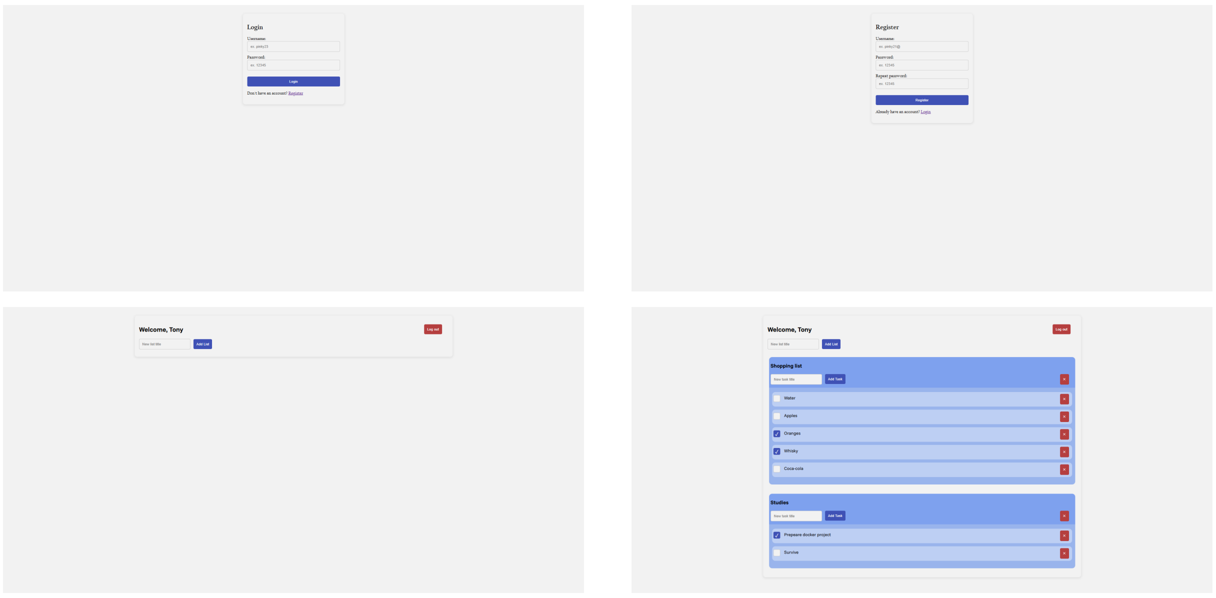
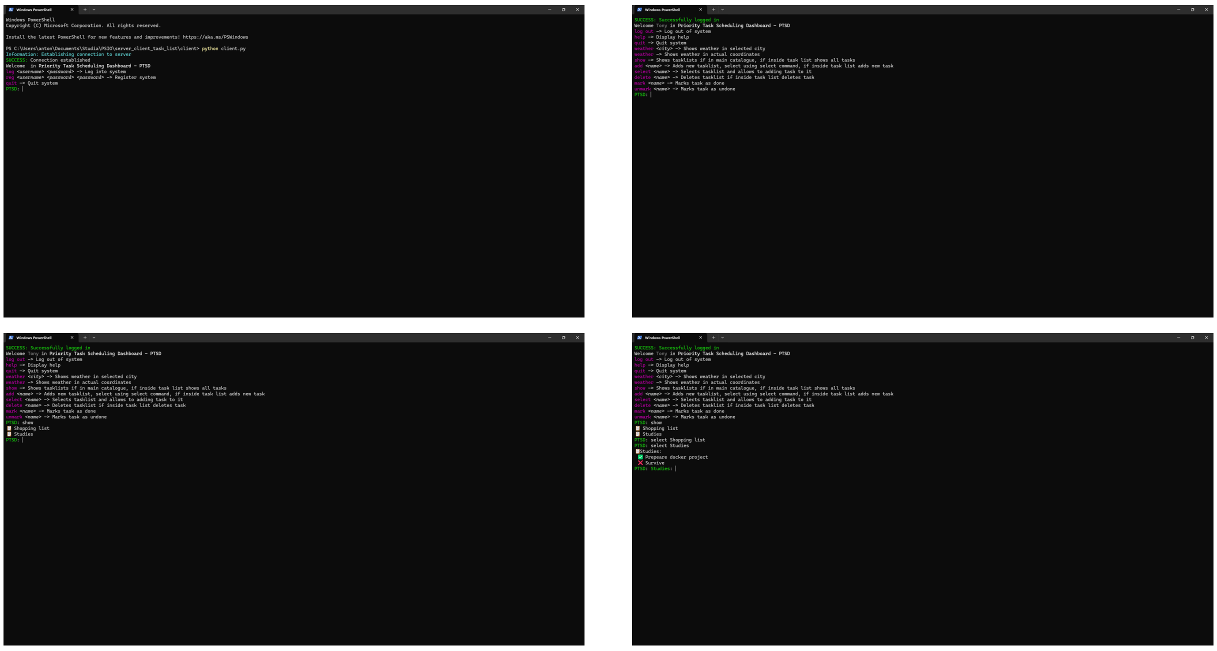


Figure 1: Web application views: login, register, dashboard empty and dashboard with tasks

- **CLI Interface:** A lightweight command-line interface that allows users to interact with the application via terminal commands.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\anton\Documents\Studia\PSIO\server_client_task_list\client> python client.py
Information: Establishing connection to server
SUCCESS: Connection established
Welcome  in Priority Task Scheduling Dashboard - PTSD
log <username> <password> -> Log into system
reg <username> <password> <password> -> Register system
quit -> Quit system
PTSD: |
```

```
SUCCESS: Successfully logged in
Welcome Tony in Priority Task Scheduling Dashboard - PTSD
log out -> Log out of system
help -> Display help
quit -> Quit system
weather <city> -> Shows weather in selected city
weather -> Shows weather in actual coordinates
show -> Shows tasklists if in main catalogue, if inside task list shows all tasks
add <name> -> Adds new tasklist, select using select command, if inside task list adds new task
select <name> -> Selects tasklist and allows to adding task to it
delete <name> -> Deletes tasklist if inside task list deletes task
mark <name> -> Marks task as done
unmark <name> -> Marks task as undone
PTSD: |
```

```
SUCCESS: Successfully logged in
Welcome Tony in Priority Task Scheduling Dashboard - PTSD
log out -> Log out of system
help -> Display help
quit -> Quit system
weather <city> -> Shows weather in selected city
weather -> Shows weather in actual coordinates
show -> Shows tasklists if in main catalogue, if inside task list shows all tasks
add <name> -> Adds new tasklist, select using select command, if inside task list adds new task
select <name> -> Selects tasklist and allows to adding task to it
delete <name> -> Deletes tasklist if inside task list deletes task
mark <name> -> Marks task as done
unmark <name> -> Marks task as undone
PTSD: show
📋 Shopping list
📋 Studies
PTSD: |
```

```
SUCCESS: Successfully logged in
Welcome Tony in Priority Task Scheduling Dashboard - PTSD
log out -> Log out of system
help -> Display help
quit -> Quit system
weather <city> -> Shows weather in selected city
weather -> Shows weather in actual coordinates
show -> Shows tasklists if in main catalogue, if inside task list shows all tasks
add <name> -> Adds new tasklist, select using select command, if inside task list adds new task
select <name> -> Selects tasklist and allows to adding task to it
delete <name> -> Deletes tasklist if inside task list deletes task
mark <name> -> Marks task as done
unmark <name> -> Marks task as undone
PTSD: show
📋 Shopping list
📋 Studies
PTSD: select Shopping list
PTSD: select Studies
📋Studies:
 ✅ Prepeare docker project
 ❌ Survive
PTSD: Studies: |
```

Figure 2: CLI application views: entry point, after registration, listing all lists and listing tasks

3 Technical Design

3.1 Architecture Overview

3.1.1 Data Flow Diagram

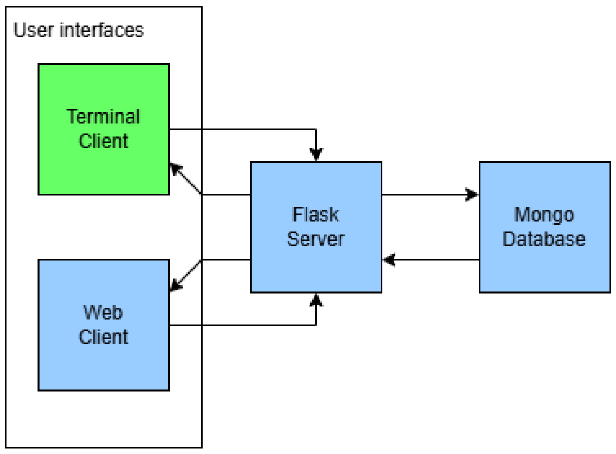


Figure 3: Data Flow Diagram

The system architecture is composed of multiple components working together to deliver a seamless experience across different user interfaces.

There are two main user-facing interfaces: a web interface and a command-line interface (CLI). The web interface is a modern, responsive application that can be accessed through any standard web browser, while the terminal version provides a lightweight, fast way to interact with the system via the command line. Both interfaces communicate with the backend server through HTTP requests.

The backend server is built using Flask, a lightweight and flexible Python web framework. It acts as the central component, managing business logic, authentication, data validation, and communication with the database. The server exposes a set of RESTful API endpoints that are consumed by both the web and terminal clients.

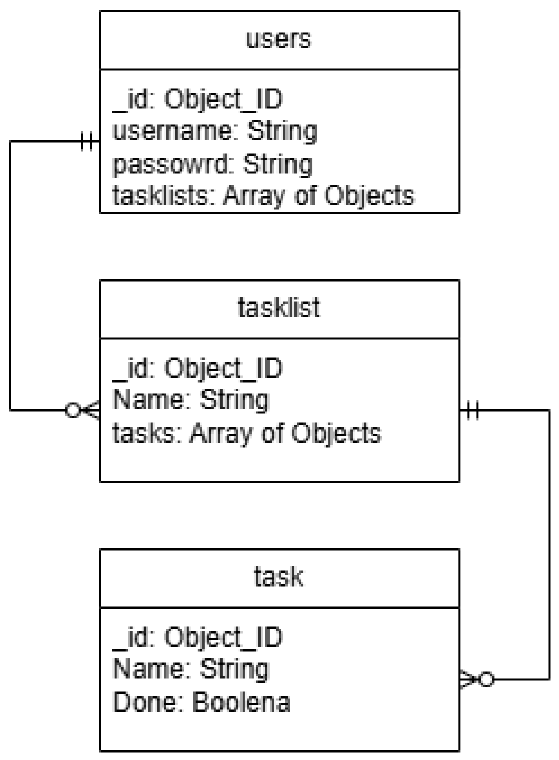
Data is persisted in a MongoDB database, which stores information about users, tasks, and other entities. MongoDB's document-oriented model provides flexibility and scalability, allowing the application to easily evolve over time.

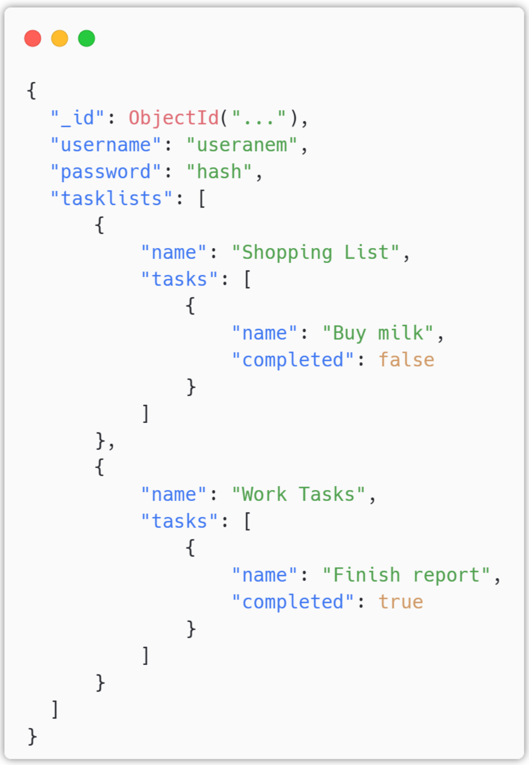
Each major component of the system—the database, the backend server, and the web client—is encapsulated in a separate Docker container. This ensures isolation, portability, and consistency across different deployment environments. In the data flow diagram, containers are represented with blue figures to clearly distinguish them from other system components.

The communication flow follows a client-server pattern. User actions in the web or terminal interface trigger HTTP requests to the Flask server. The server processes these requests, interacts with the MongoDB database when necessary, and returns appropriate responses back to the clients.

Docker Compose is used to orchestrate the system, ensuring that services are correctly networked and initialized. This setup simplifies deployment and development, allowing the whole application to be run with minimal configuration effort.

3.1.2 Database schema





```
{
  "_id": ObjectId("..."),
  "username": "useranem",
  "password": "hash",
  "tasklists": [
      {
          "name": "Shopping List",
          "tasks": [
              {
                  "name": "Buy milk",
                  "completed": false
              }
          ]
      },
      {
          "name": "Work Tasks",
          "tasks": [
              {
                  "name": "Finish report",
                  "completed": true
              }
          ]
      }
  ]
}
```

Figure 4: Database structure (UML), Database document

The schema represents a task management system where each user can have multiple task lists, and each task list contains multiple tasks. The schema is designed to be flexible and scalable, using MongoDB's NoSQL format, which allows for nested structures and dynamic data types.

- *users* Collection: Stores information about the users, including their username, password (hashed), and associated tasks lists.
- *tasklist* Collection (Part of the User Document): Represents task lists that belong to the user. Each task list has a name and a set of tasks.
- *task* Collection (Part of the Tasklist Document): Represents individual tasks within a task list. Each task has a name and a completion status.

3.1.3 Backend Description

The application has three main blueprints responsible for handling different aspects of the system: Users, Tasks, and Weather, as shown in Figure 5.

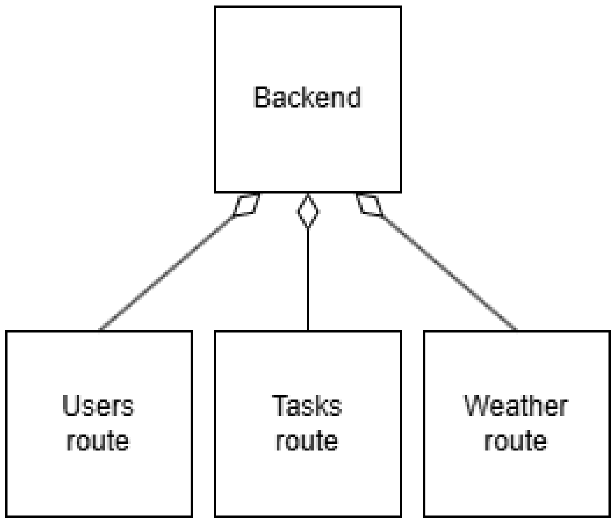


Figure 5: Blueprint schema

The backend architecture is organized into three main components, each encapsulated in its own blueprint, as described below:

- **Users Blueprint:**
  The `users` blueprint is responsible for managing user-related actions such as registration, login, and user data retrieval. The URL prefix for this blueprint is `/users`.
  - **GET /users**
    Returns the list of all users in the system. This endpoint retrieves all user records from the database.
  - **POST /delete_user**
    Deletes a specified user from the database. Requires the IDs of both the administrator and the user to be deleted. Admin users are the only ones authorized to perform this operation.
  - **POST /get_user_id**
    Given a username and password, returns the corresponding user ID. This endpoint is used to authenticate users and retrieve their unique identifier.
  - **POST /user_exist**
    Checks if a user with the given username exists in the system. Returns a boolean value indicating the presence of the user.
  - **POST /log_in**
    Authenticates a user by checking their username and password. Returns a response indicating whether the login was successful, along with the user's ID if authenticated.
  - **POST /register**
    Registers a new user with a specified username and password. The password is hashed before being stored. If the username already exists, it returns an error message.
- **Tasks Blueprint:**
  The `tasks` blueprint is responsible for managing tasks associated with users. It

allows for creating, reading, updating, and deleting tasks. The URL prefix for this blueprint is `/tasks`.

- **Tasks Blueprint:**
  The `tasks` blueprint is responsible for managing tasks associated with users. It allows for creating, reading, updating, and deleting tasks. The URL prefix for this blueprint is `/tasks`.
  - **POST /<`user_id`>/tasklists**
    Adds a new tasklist for the specified user. Requires a title for the tasklist to be included in the request body.
  - **POST /<`user_id`>/tasklists/<`list_title`>/tasks**
    Adds a new task to the specified tasklist. Requires a title for the task and marks the task as not done by default.
  - **PATCH /<`user_id`>/tasklists/<`list_title`>/tasks/<`task_title`>/done**
    Marks the specified task as done. The task is identified by its title within the tasklist.
  - **PATCH /<`user_id`>/tasklists/<`list_title`>/tasks/<`task_title`>/undone**
    Marks the specified task as not done (undo). The task is identified by its title within the tasklist.
  - **DELETE /<`user_id`>/tasklists/<`list_title`>**
    Deletes the specified tasklist for the given user. The tasklist is identified by its title.
  - **DELETE /<`user_id`>/tasklists/<`list_title`>/tasks/<`task_title`>**
    Deletes the specified task from the tasklist. The task is identified by its title within the tasklist.
  - **GET /<`user_id`>/tasklists**
    Retrieves all tasklists for the specified user. If the user does not exist, returns an error message.
- **Weather Blueprint:**
  The `weather` blueprint is responsible for retrieving weather data. It fetches data from external weather APIs. The URL prefix for this blueprint is `/weather`.
  - **GET /weather/current**
    Retrieves the current weather data for a specified location.
  - **GET /weather/forecast**
    Retrieves the weather forecast for a specified location.

3.2 Docker Configuration:

Docker is used to containerize the entire application, including the backend, frontend, and database services. This allows for a consistent development environment, easy deployment, and scalability. The configuration includes defining services, ports, and volumes for each component.

- **Backend Service (Flask Application):**
  The backend service is containerized using Docker and runs the Flask application.

The container listens on port 5000 and depends on the MongoDB container for data storage. It exposes the necessary API endpoints for interacting with the tasks and user data.

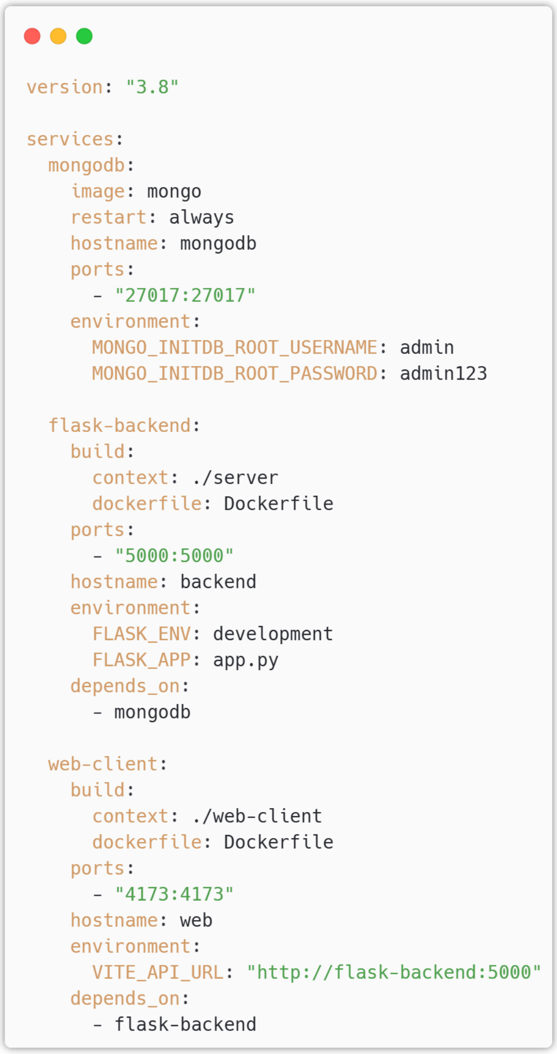
  - **Dockerfile:** The backend service is built from a `Dockerfile`, which specifies the base image (e.g., `python:3.8`), installs dependencies, copies the application code into the container, and sets up the environment variables.
  - **Ports:** The Flask application inside the container listens on port 5000, which is mapped to the same port on the host machine to allow external access.

- **Frontend Service (React Application):**
  The frontend is built using React and is also containerized using Docker. The React app is served on port 4173 by default. The frontend interacts with the backend API to display and manage tasks.
  - **Dockerfile:** The frontend container is built from a `Dockerfile` that sets up Node.js, installs the required npm packages, and runs the React development server.
  - **Ports:** The React application listens on port 4173, which is mapped to port 4173 on the host machine.

- **Database Service (MongoDB):**
  MongoDB is used as the database to store user and task data. It is containerized as a separate service and is configured to persist data using Docker volumes.
  - **MongoDB Container:** The MongoDB service is based on the official `mongo` Docker image and runs on port 27017. The data is persisted in a Docker volume, so it remains available even if the container is stopped or removed.
  - **Volumes:** The MongoDB data is stored in a volume that is mounted to the container's data directory to persist data across container restarts.

- **Docker Compose:**
  To simplify the management of the different services, Docker Compose is used to define and run multi-container Docker applications. The `docker-compose.yml` file specifies the services, their configurations, ports, volumes, and dependencies between them.
  - **Services:** The `docker-compose.yml` file defines three services: `backend`, `frontend`, and `database`.
  - **Networks:** A custom Docker network is defined to allow communication between the services. The backend and frontend services communicate over this network with MongoDB.
  - **Ports:** The necessary ports for each service are exposed to the host machine, so the backend and frontend can be accessed externally.
  - **Volumes:** Persistent storage for MongoDB is configured via Docker volumes to ensure data persists even when containers are removed or restarted.

- **Example `docker-compose.yml` File:**
  Here is an example of the `docker-compose.yml` configuration for the application:



```yaml
version: "3.8"

services:
  mongodb:
    image: mongo
    restart: always
    hostname: mongodb
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: admin123

  flask-backend:
    build:
      context: ./server
      dockerfile: Dockerfile
    ports:
      - "5000:5000"
    hostname: backend
    environment:
      FLASK_ENV: development
      FLASK_APP: app.py
    depends_on:
      - mongodb

  web-client:
    build:
      context: ./web-client
      dockerfile: Dockerfile
    ports:
      - "4173:4173"
    hostname: web
    environment:
      VITE_API_URL: "http://flask-backend:5000"
    depends_on:
      - flask-backend
```

Figure 6: docker-compose.yml

This configuration ensures that all components of the application are containerized, and the backend, frontend, and database services can communicate with each other efficiently. Docker Compose simplifies the process of starting and stopping the entire application stack with a single command. This setup allows for easy scaling, management, and deployment of the application in different environments.

3.3 Dependencies:

The application relies on several key dependencies to handle the backend, frontend, and interactions with the database. These dependencies are crucial for the development, deployment, and maintenance of the application. Below are the primary dependencies for each part of the system:

- **Backend Dependencies (Flask and MongoDB):**
  - **Flask:** Flask is a lightweight web framework for Python. It is used to create the backend API for managing tasks, user data, and interaction with the MongoDB database. The Flask application handles routing, request processing, and response formatting.

  - **python-dotenv:** python-dotenv is a library for reading key-value pairs from .env files. It is used to manage environment variables for the backend, allowing secure and flexible configuration (e.g., database credentials, Flask environment settings).
  - **flask-cors:** Flask-CORS is a Flask extension that allows cross-origin requests, enabling the frontend (which may be served from a different domain or port) to make API requests to the backend.
  - **pymongo:** PyMongo is a Python library used for interacting with MongoDB. It allows the Flask backend to connect to and manage the MongoDB database, perform CRUD operations, and handle data persistence.
  - **bcrypt:** bcrypt is a library for hashing passwords securely. It is used for hashing user passwords before storing them in the database and for verifying user credentials during login.
  - **requests:** Requests is a simple HTTP library for Python. It is used in the Flask backend for making HTTP requests to external services, such as fetching data or interacting with other APIs.

- **Frontend Dependencies (React and Axios):**
  - **React:** React is a JavaScript library for building user interfaces. It is used to create the frontend of the application, where users can view and manage their task lists. React is component-based, making it easy to build interactive UIs with reusable components.
  - **react-dom:** This package is the entry point to the DOM and enables rendering of React components in the browser. It works alongside React to manage the DOM updates and application state.
  - **react-router-dom:** React Router is used for handling navigation and routing within the React application. It allows users to navigate between different views without refreshing the entire page, ensuring a single-page application (SPA) experience.
  - **Vite:** Vite is a fast and modern build tool for frontend development. It is used to serve the React application during development and compile the code for production. Vite offers features such as fast hot module replacement (HMR) and optimized builds for production.

- **Database Dependencies (MongoDB):**
  - **MongoDB:** MongoDB is a NoSQL document-based database. It stores the data for tasks and user task lists in a flexible, schema-less format. MongoDB allows for fast querying and scalability, making it suitable for applications with evolving data structures.

- **Development Dependencies:**
  - **Docker:** Docker is a platform for containerizing applications. It is used to create isolated environments for the backend, frontend, and database services. Docker ensures consistency across different environments (development, testing, and production) and simplifies the deployment process.

  - **Docker Compose:** Docker Compose is used to define and manage multi-container Docker applications. It allows for the orchestration of services (backend, frontend, database) and simplifies the management of dependencies between them.
  - **pytest:** pytest is a testing framework for Python. It is used to write and run unit tests for the backend Flask application to ensure code quality and correctness. Tests can include API endpoint checks, database interactions, and other business logic.
  - **ESLint and Prettier:** ESLint and Prettier are tools used for code linting and formatting in the frontend (React) project. ESLint helps identify and fix JavaScript issues, while Prettier enforces a consistent code style.

The dependencies are managed using `pip` for the backend (Python) and `npm` (or `yarn`) for the frontend (JavaScript). The `requirements.txt` file for the backend and `package.json` for the frontend list all the required dependencies for each environment.