



Team Members:

Jason Surya Sandjaya 001202300102
Jason Anthony Wibowo 001202300216

Distributed and Parallel System
President University
2025

1. Introduction

The **Task Manager** is a simple CRUD web application developed to demonstrate containerized deployment using Docker and Docker Compose. It consists of a Python Flask backend providing REST APIs and a frontend built with HTML, CSS, and JavaScript. Both components run in separate containers, allowing users to create, update, and manage tasks through a clean web interface. This project showcases the core principles of building and deploying distributed applications efficiently.

2. Project Background

Developing modern applications often involves significant challenges, particularly in managing multiple interconnected services and ensuring consistent deployment across different environments. Traditional development workflows frequently encounter "it works on my machine" scenarios, where discrepancies between developer setups and production servers lead to frustrating issues. Our Task Manager project was initiated to create a practical solution for these common problems. It aims to demonstrate how a simple, yet effective, web application for task management can be built with a clear separation of concerns, providing a robust and manageable system. This project serves as a tangible example of how a modular application, focusing on core task management functionalities, addresses the complexities inherent in multi-service development, laying a foundation for more sophisticated distributed systems.

3. Objectives

This Task Manager project aims to deliver a web application that provides core task management functionality, allowing users to efficiently create, read, update, and delete tasks with clear titles, descriptions, and statuses. A key objective is to ensure an intuitive user experience through a clean HTML/CSS/JavaScript frontend, featuring visual status indicators and responsive interactions. Complementing this, a robust Python Flask REST API will be established to reliably handle all task data operations, thereby showcasing effective full-stack integration.

4. System Architecture

The Task Manager follows a **two-tier architecture**, separating the backend API and frontend interface into independent services. Each component runs inside its own Docker container, allowing for modular deployment and easier management.

Components:

1. Backend Service

- Built with Python Flask.
- Provides RESTful API endpoints to handle CRUD operations for tasks.

- Stores data **in memory**, meaning all task information is lost when the container stops.
- 2. **Frontend Service**
 - Built with HTML, CSS, and JavaScript.
 - Communicates with the backend API to fetch and update task data.
 - Served by an Nginx container (or optionally a simple static server).
- 3. **Docker Compose**
 - Used to define and run both services together.
 - Automatically creates a shared network for containers to communicate.

How it Works:

- When the user interacts with the frontend (e.g., adding a task), the frontend makes HTTP requests to the Flask API.
- The backend processes the request and updates the in-memory data structure.
- The frontend then fetches the updated task list and refreshes the view.

5. Features

The Task Manager provides the following core features:

1. **Web Interface**

Users can view all tasks in a clean, responsive dashboard. The interface displays a list of tasks with their titles, descriptions, and current status.
2. **Add New Tasks**

Users can create new tasks by entering a title, description, and selecting a status (Pending, In-Progress, or Completed). The new task appears immediately in the list.
3. **Edit or Delete Tasks**

Existing tasks can be edited to update their details or deleted if they are no longer needed.
4. **Status Display**

Each task displays a visual badge indicating its status. This makes it easy to track progress at a glance.
5. **In-Memory Data Handling**

All task data is stored in memory within the backend service. As a result, data is temporary and resets when the container is stopped or restarted.

6. Technologies Used

The project was developed using the following technologies and tools:

1. **Python Flask**

Used to build the backend REST API that handles all CRUD operations and serves task data to the frontend.

2. **HTML, CSS, and JavaScript**

Used to create the frontend web interface, allowing users to interact with the application in the browser.

3. **Docker**

Employed to containerize the backend and frontend components, ensuring consistent environments and simplifying deployment.

4. **Docker Compose**

Used to define and orchestrate multiple containers, making it easy to build, start, and manage all services together.

7. Front End

The frontend of the Task Manager was developed using standard web technologies: **HTML**, **CSS**, and **JavaScript**. It serves as the user interface, allowing users to interact with the task management system in a simple and intuitive way.

7.1 Interface Overview

The frontend provides a clean dashboard where users can:

- View a list of existing tasks
- Add new tasks using a form
- Edit or delete existing tasks
- See the status of each task via colored badges (e.g., Pending, In-Progress, Completed)

7.2 Functionality

1. **Task Display**

On page load, the frontend fetches the list of tasks from the backend API and displays them in a table or card layout. Each task includes a title, description, and status badge.

2. **Add Task**

A form is provided where users can input a title, description, and select a status from a dropdown. Upon submission, the task is sent to the backend via a POST request.

3. **Edit Task**

Users can click an "Edit" button next to any task to modify its details. The updated task is sent to the backend via a PUT request.

4. **Delete Task**

Each task also includes a "Delete" button. When clicked, the task is removed via a DELETE request to the backend.

5. **Responsive Design**

Basic styling with CSS ensures that the interface is responsive and works across different screen sizes.

7.3 Frontend Dockerfile Explanation

```
1  # Use nginx to serve static files
2  FROM nginx:alpine
3
4  # Copy static files to nginx html directory
5  COPY . /usr/share/nginx/html/
6
7  # Copy custom nginx configuration
8  COPY nginx.conf /etc/nginx/conf.d/default.conf
9
10 # Expose port 80
11 EXPOSE 80
12
13 # Start nginx
14 CMD ["nginx", "-g", "daemon off;"]
```

1. Base Image

FROM nginx:alpine

This line uses a lightweight Alpine-based Nginx image as the base. Nginx is commonly used to serve static web content efficiently.

2. Copy Static Files

COPY . /usr/share/nginx/html/

This copies all files from the build directory of your frontend project (e.g., HTML, CSS, JavaScript) into the default Nginx HTML directory where they will be served.

3. Custom Nginx Configuration

COPY nginx.conf /etc/nginx/conf.d/default.conf

This replaces the default Nginx configuration with a custom configuration file (**nginx.conf**). This allows us to customize settings such as:

- CORS headers
- Default index page
- Routing rules
- MIME types

4. Expose Port

EXPOSE 80

This declares that the container listens on port 80, which is the default HTTP port used by Nginx. Docker Compose or **docker run** will map this to a host port so we can access the frontend via our browser.

5. Start Nginx

CMD ["nginx", "-g", "daemon off;"]

This tells the container to start the Nginx process in the foreground (**daemon off**). Running it in the foreground ensures the container stays alive as long as Nginx is running.

7.4 Frontend Nginx Configuration

```
1  server {
2      listen 80;
3      server_name localhost;
4
5      root /usr/share/nginx/html;
6      index index.html;
7
8      # Handle static files (and SPA client-side routing)
9      location / {
10         try_files $uri $uri/ /index.html;
11     }
12
13     # IMPORTANT ADDITION: Proxy API requests to your backend service
14     # The 'task-manager-backend' part should match the service name
15     # of your backend in your docker-compose.yml file.
16     # The ':8000' should match the port your backend service is listening on inside its container.
17     location /api/ {
18         proxy_pass http://task-manager-backend:8000; # <--- Adjust this line!
19         proxy_set_header Host $host;
20         proxy_set_header X-Real-IP $remote_addr;
21         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
22         proxy_set_header X-Forwarded-Proto $scheme; # Important for HTTPS if you ever add it
23     }
24
25     # Enable gzip compression
26     gzip on;
27     gzip_types text/plain text/css application/json application/javascript text/xml application/xml+rss text/javascript;
28
29     # Cache static assets
30     location ~* \.(css|js|png|jpg|jpeg|gif|ico|svg)$ {
31         expires 1y;
32         add_header Cache-Control "public, immutable";
33     }
34 }
```

1. Server Block

```
server {
```

```
    listen 80;
```

```
    server_name localhost;
```

- Configures Nginx to listen on port 80 (HTTP) and respond to requests for **localhost**.

2. Serving Static Files

```
    root /usr/share/nginx/html;
```

```
    index index.html;
```

- Sets the root directory where your HTML, CSS, and JavaScript files are copied in the Docker image.
- Defines **index.html** as the default file to serve.

3. Single Page Application (SPA) Routing

```
location / {
    try_files $uri $uri/ /index.html;
}
```

- If a requested file does not exist, Nginx falls back to serving **index.html**.
- This ensures client-side routing works properly (e.g., refreshing a route like /tasks).

4.API Proxying

```
location /api/ {
    proxy_pass http://task-manager-backend:8000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

- For any request starting with **/api/**, Nginx forwards the request to the backend service container.
- **task-manager-backend** is the **Docker Compose service name**, and **8000** is the backend port.
- **The proxy_set_header** directives preserve the original request headers (host, IP, protocol).

5. Gzip Compression

```
gzip on;

gzip_types text/plain text/css application/json application/javascript text/xml
application/xml+rss text/javascript;
```

- Enables gzip compression to reduce the size of files sent to the browser.

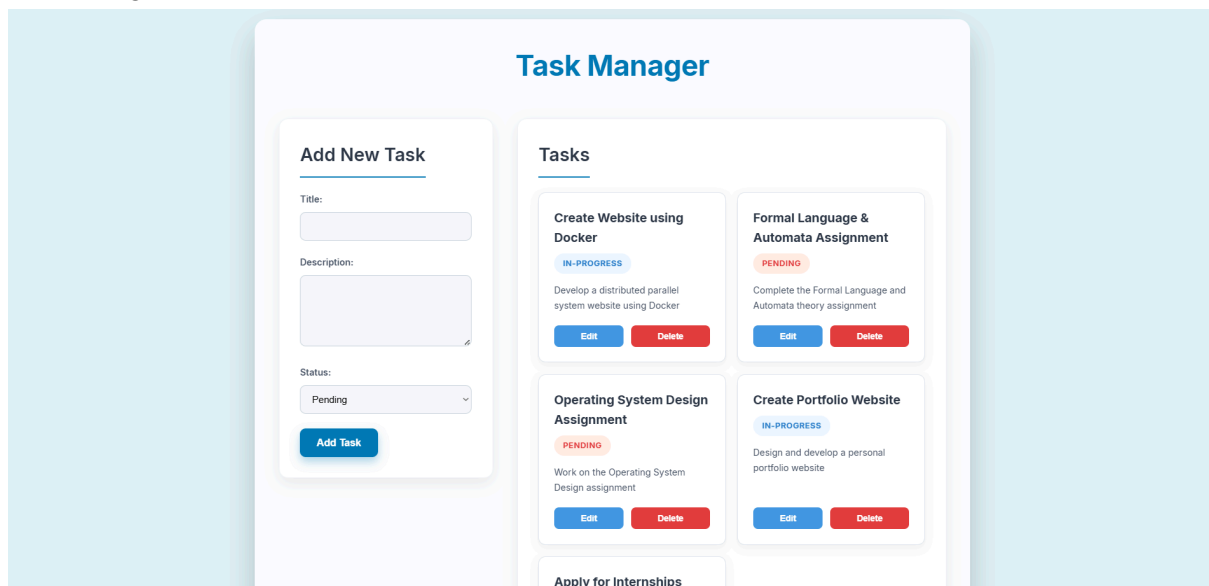
6. Caching Static Assets

```
location ~* \.(css|js|png|jpg|jpeg|gif|ico|svg)$ {  
  
    expires 1y;  
  
    add_header Cache-Control "public, immutable";  
  
}
```

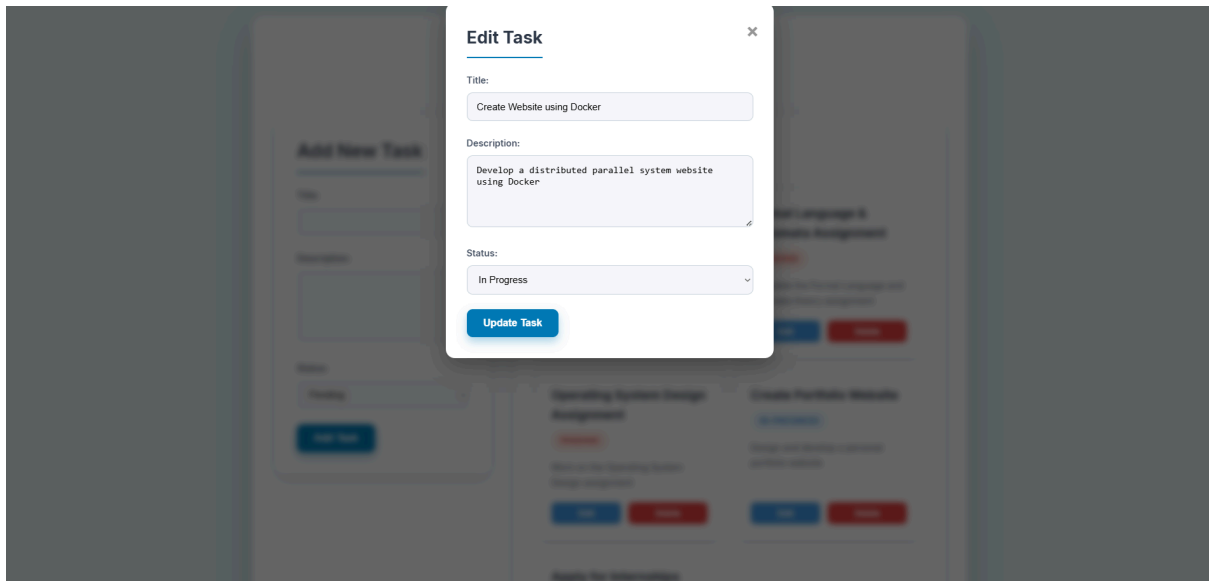
- Sets long-term caching headers for static assets (CSS, JS, images) to improve load performance.

8. User Interface and User Experience (UI/UX)

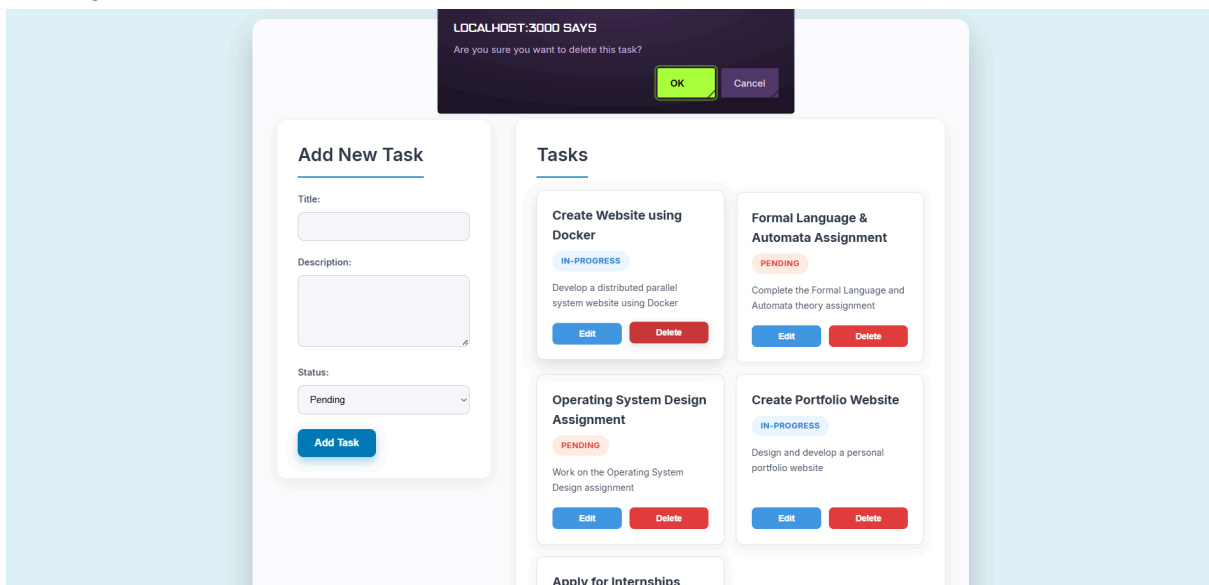
Default Page:



Editing a task:



Deleting a task:



The Task Manager uses a **clean, two-panel layout**:

- **Add New Task Panel (Left):**
Users enter the title, description, and status, then click **Add Task** to create a new entry.
- **Task List Panel (Right):**
Displays all tasks as cards showing the title, description, status badge, and **Edit/Delete** buttons.

Design Highlights:

- **Color-coded status badges** (green for Completed, blue for In-Progress, red for Pending).
- **Minimalist styling** with clear typography and blue action buttons.

- **Responsive design** that works on various screen sizes.
- **Instant updates** after adding or editing tasks, without page reload.

9. Backend

Dockerfile

```
# Use Python 3.11 slim image
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

# Install system dependencies
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
        gcc \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements first for better caching
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create non-root user for security
RUN adduser --disabled-password --gecos '' appuser && chown -R \
    appuser:appuser /app
USER appuser

# Expose port
EXPOSE 8000

# Health check
HEALTHCHECK --interval=30s --timeout=30s --start-period=5s --retries=3 \
```

```
CMD python -c "import requests;
requests.get('http://localhost:8000/') " || exit 1

# Run the application
CMD ["python", "app.py"]
```

1. FROM Instruction

Specifies the base image (python:3.11-slim), choosing a minimal variant to reduce image size, speed up downloads, and enhance security by including only essential components for Python.

2. WORKDIR Instruction

Sets the working directory inside the container to /app. This simplifies path references for subsequent instructions, making the Dockerfile cleaner.

3. ENV Instructions

Define environment variables within the image:

PYTHONDONTWRITEBYTECODE=1: Prevents Python from generating .pyc bytecode, reducing image size and avoiding potential issues.

PYTHONUNBUFFERED=1: Ensures Python's output is unbuffered, allowing real-time log visibility in Docker.

4. First RUN Instruction (System Dependencies)

Installs system-level packages (gcc) by first updating apt lists. The gcc compiler is often needed for Python libraries with C extensions. This single RUN command includes a critical step (rm -rf /var/lib/apt/lists/*) to clean up cached files, minimizing the layer's size and the final image.

5. First COPY Instruction (requirements.txt)

Copies only requirements.txt into the container. This is a key build cache optimization, allowing Docker to reuse the Python dependency installation layer if only application code changes, significantly speeding up subsequent builds.

6. Second RUN Instruction (Python Dependencies)

Executes pip install to install all Python packages from requirements.txt. The --no-cache-dir flag prevents pip from storing unnecessary download archives, further reducing the final image size.

7. Second COPY Instruction (Application Code)

Copies the remaining application code into the container. Its placement after dependency installations leverages Docker's build cache; changes here only invalidate subsequent layers, not the time-consuming dependency layers.

8. First RUN Instruction (User Creation)

Creates a dedicated non-root user (appuser) and changes ownership of the /app directory. This is a vital security measure, reducing the impact of potential compromises by avoiding root privileges within the container.

9. USER Instruction

Switches the user context for all subsequent instructions and the container's runtime to appuser, ensuring the application operates with reduced privileges.

10. EXPOSE Instruction

Serves as documentation, indicating that the container intends to listen on port 8000. It does not publish the port; explicit mapping is required via docker run -p or docker-compose.yml.

11. HEALTHCHECK Instruction

Defines a health check to monitor the container's status. It periodically runs a command (HTTP GET to localhost:8000) to confirm application responsiveness, with configurable intervals, timeouts, retries, and a start period. This aids orchestration systems in managing container lifecycle.

12. CMD Instruction

Specifies the default command (python app.py) to execute when the container starts. Using the "exec form" (JSON array) is preferred for proper signal handling and graceful application shutdown.

10. Docker-compose

```
services:
  # Backend service
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: task-manager-backend
    ports:
      - "8000:8000"
    environment:
      - FLASK_ENV=production
      - PYTHONUNBUFFERED=1
    volumes:
      - ./backend:/app
    networks:
      - task-manager-network
    restart: unless-stopped
    healthcheck:
```

```

    test: ["CMD", "python", "-c", "import requests;
requests.get('http://localhost:8000/')"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 40s

# Frontend service
frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  container_name: task-manager-frontend
  ports:
    - "3000:80"
  depends_on:
    - backend
  networks:
    - task-manager-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "wget", "--quiet", "--tries=1", "--spider",
"http://localhost:80"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 40s

networks:
  task-manager-network:
    driver: bridge
    name: task-manager-network

# Optional: Add volumes for persistent data if needed in the future
volumes:
  task-data:
    driver: local

```

1. Service Definitions

The services section defines the core components of the application: the backend API and the frontend web interface.

1.1 Backend Service (backend)

This service represents the Python Flask REST API responsible for task management.

Build Configuration: Specifies that the backend's Docker image is built from a Dockerfile located within the `./backend` directory.

Container Naming: Assigns a clear, static name (`task-manager-backend`) to the running container for easy identification and management.

Port Mapping: Exposes port 8000 from the container to port 8000 on the host machine, allowing external access to the backend API (e.g., for testing or by the frontend via the Nginx proxy).

Environment Variables: Configures runtime settings for the Flask application, setting `FLASK_ENV` to `production` for optimized performance and `PYTHONUNBUFFERED` to `1` to ensure real-time log output.

Volume Mounting: Utilizes a bind mount to synchronize the `./backend` directory on the host with the `/app` directory inside the container. This facilitates rapid development by reflecting local code changes instantly within the running container without requiring a rebuild.

Networking: Connects the backend container to the `task-manager-network`, enabling private communication with other services on the same network using their service names.

Restart Policy: Implements an `unless-stopped` restart policy, ensuring the backend container automatically restarts if it crashes or if the Docker daemon is restarted, thereby enhancing service availability.

Health Check: Defines an internal health check to monitor the backend's operational status. This check periodically attempts an HTTP GET request to the Flask application's internal port. Parameters are configured for interval (30s), timeout (10s), retries (3), and a start period (40s) to allow the application to fully initialize.

1.2 Frontend Service (frontend)

This service manages the static HTML, CSS, and JavaScript files for the web interface, typically served by Nginx.

Build Configuration: Directs Docker Compose to build the frontend's image using a Dockerfile found in the `./frontend` directory.

Container Naming: Provides a descriptive name (`task-manager-frontend`) for the frontend container.

Port Mapping: Maps port 3000 on the host machine to port 80 inside the container (Nginx's default listening port), making the frontend accessible via `http://localhost:3000`.

Service Dependency: Specifies `depends_on: backend`, ensuring that the backend service is initiated before the frontend service attempts to start. It's important to note that this only guarantees start order, not backend health.

Networking: Connects the frontend container to the shared task-manager-network, enabling it to communicate with the backend via service discovery facilitated by Docker Compose.

Restart Policy: Applies an unless-stopped restart policy, ensuring the frontend service's resilience by automatically restarting it upon unexpected termination.

Health Check: Configures a health check for the frontend using wget. This check verifies that the web server is responsive by attempting to "spider" its internal port 80, with similar parameters for interval, timeout, retries, and start period to ensure accurate status reporting.

2. Network Definitions

The networks section defines the custom network for inter-service communication.

task-manager-network: This custom bridge network is created to provide an isolated and secure communication channel between the backend and frontend containers. Services connected to this network can communicate with each other using their service names as hostnames, simplifying internal DNS resolution. The driver: bridge is the default and most common network driver for isolated container groups.

3. Volume Definitions

The volumes section declares named volumes for data persistence.

task-data: This named volume is defined with a local driver, meaning its data will be stored on the Docker host's filesystem. While currently declared, this volume is not yet attached to any service. In future enhancements, it would be used to persist application data (e.g., a database's data files) independently of the container's lifecycle, ensuring data is retained even if containers are removed or recreated.

11. Deployment Steps Using Docker

To deploy and run this application, you will need the following installed on your system:

- **Git:** For cloning the repository.
- **Docker Desktop:** This includes both Docker Engine and Docker Compose, which are essential for containerizing and orchestrating the application's services. Ensure Docker Desktop is running before proceeding (you should see the Docker whale icon in your system tray).
- **Web Browser:** Any modern web browser (e.g., Chrome, Firefox, Edge) to access the application.

For this application to function correctly, the project's directory structure is critical. After cloning, your project directory should look like this:

```
todo/
├── backend/
│   ├── Dockerfile
│   ├── app.py           (Your Flask application entry point)
│   └── requirements.txt (Python dependencies)
├── frontend/
│   ├── Dockerfile
│   ├── index.html       (Your main HTML file)
│   ├── style.css         (Your CSS file)
│   ├── script.js         (Your JavaScript logic)
│   └── nginx.conf        (Nginx configuration for frontend)
└── docker-compose.yml    (Your Docker Compose orchestration file)
```

Follow these steps in your terminal:

Step 1: Clone the Repository

First, clone the project repository from GitHub to your local machine. Replace your-github-username/your-repo-name with the actual path to your repository.

```
git clone https://github.com/your-github-username/your-repo-name.git
```

Step 2: Navigate to the Project Root Directory

After cloning, navigate into the newly created project directory. This directory contains the docker-compose.yml file.

```
cd your-repo-name
```

Step 3: Build and Run the Application with Docker Compose

Execute: **docker-compose up --build**

```
docker-compose up --build
```

This command will instruct Docker Compose to read your docker-compose.yml file, build the necessary Docker images for both the backend and frontend services, and then start them.

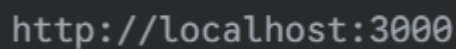
- up: Creates and starts containers, networks, and volumes as defined in docker-compose.yml.

- **--build:** This flag is crucial as it tells Docker Compose to build (or re-build) the Docker images for your services. This ensures that any recent changes in your Dockerfiles or application code are incorporated.

Step 4: Verify the Application Deployment

Once the `docker-compose up` command has finished execution (you'll see logs from both services, and the command prompt won't return until you stop it), your application should be running.

1. **Open your web browser:** Navigate to the local address where your frontend service is exposed. As configured in `docker-compose.yml`, it's <http://localhost:3000>



```
http://localhost:3000
```

2. **Check the Frontend:** You should now see the "Task Manager" web interface, styled according to the CSS and interactive via JavaScript.
3. **Interact with the Application:** Test the core functionalities:
 - Add new tasks.
 - Edit existing tasks.
 - Delete tasks.
 - Observe the task card colors changing based on their status (Pending, In-Progress, Completed).
4. **Stopping the Application:** To stop and remove the running Docker containers, return to your terminal (where `docker-compose up` is running) and press `Ctrl + C`. This will gracefully shut down the services. After the graceful shutdown, you can optionally run the following command to remove the containers, networks, and anonymous volumes created by Docker Compose: **`docker-compose down`**



```
docker-compose down
```

12. Conclusion

This project successfully demonstrates the development and deployment of a web application with both frontend and backend components in a standardized, containerized environment.

By creating a backend API with Python Flask and a frontend interface using HTML, CSS, and JavaScript, the project fulfills the requirement to build both parts of the system independently.

Dockerfiles were implemented for each service to ensure consistent builds across different environments. Additionally, a **`docker-compose.yml`** file was created to simplify the deployment process and manage both containers together.

Overall, this container-based approach provides a reproducible, scalable solution that meets the project objectives of handling the full stack in a unified workflow.

13. Github Repository

<https://github.com/Pollux14/taskmanager>