**🐳 Building and Deploying a Multi-Container Node.js Application with Docker**

This project demonstrates how to build and deploy a **multi-container Node.js application** using **Docker and Docker Compose**. The goal is to deploy an application that records tasks/goals and access the application via http://localhost:3000.

**🚀 Application Architecture**

This project includes three Docker containers:

* **MongoDB container** – stores data from the application
* **Backend container** – hosts a Node.js REST API
* **Frontend container** – serves a React-based single-page application (SPA)

All containers are placed on the same Docker **network** for internal communication. Persistent **volumes** are used for MongoDB data and source code mounting (for hot reloading in development).

A screenshot of a computer

AI-generated content may be incorrect.

To streamline the process of building and deploying the application, I’m using a **Docker Compose file**. Docker Compose is a tool that allows you to define and manage **multi-container applications** using a single YAML configuration file. Rather than manually starting each container and configuring networks and volumes separately, all services and their dependencies are declared in one place, making setup and orchestration more efficient and consistent.

**Database Container**

We’ll now set up the MongoDB container by adding its configuration to the Docker Compose file.

A screenshot of a computer

AI-generated content may be incorrect.

* **Version: “3.8”** - tells Docker Compose which version of the Compose file format to use.
* **Service:** - This is the section where you define all your application’s containers.
* **Mongodb:** - This is the name of the service. Here we’re creating a service named mongodb.
* **Image: ‘mongo:4.4’** - This tells Compose which image to run for this service. Here we want the official MongoDB image from Docker Hub, version **4.4**.
* **Volumes: (- data:/data/db)** - This mounts a volume named data into the container’s /data/db. That’s where MongoDB stores its data files.

By mounting this volume:

* + - 1. MongoDB’s data persists outside the container
      2. Even if the container is removed or updated, your data stays safe
* **Env\_file: (- ./env/mongo.env)** - This tells Compose to load environment variables from a file.

**Note:** Inside the main project directory (multi-container-app), I’ve created an env folder to store environment variable files. The mongodb.env file contains the MongoDB credentials, as shown below.A screenshot of a computer

AI-generated content may be incorrect.

The credentials defined above will initialize a username and password for the MongoDB database when the container is first created.

**Note:** At the bottom of your docker-compose.yml file, add a volumes: section (aligned with services: at the root level). This tells Docker Compose to automatically create and manage a named data volume, ensuring your MongoDB data persists across container restarts.

A screenshot of a computer

AI-generated content may be incorrect.

**Backend Container**

* To set up the backend container, we’ll need to create three key files:
* A Dockerfile – defines how the backend image is built
* An environment file (backend.env) – stores environment variables
* A service definition – added to the docker-compose.yml file

Start by creating a file named Dockerfile inside the backend directory. Add the following configuration to define the container build process:

* **FROM node** - Base image. This tells Docker to use the official node image from Docker Hub as the base environment.
* **WORKDIR /app** - Set the working directory. Inside the container, all commands will now run relative to /app.
* **COPY package.json .** - Copy the dependency file. This copies the package.json file from your local directory into the container's current working directory (/app).
* **RUN npm install** - Install dependencies. This runs npm install inside the container so all packages listed in package.json will be installed into node\_modules.
* **COPY . .** - Copy the rest of the files. This copies everything in your current directory (.) into the container's /app.
* **EXPOSE 80** - Document the listening port. This tells Docker (and readers) that the container expects the app to listen on port 80.
* **ENV MONGODB\_USERNAME=root and ENV MONGODB\_PASSWORD=secret** - Set environment variables. Defines MONGODB\_USERNAME and MONGODB\_PASSWORD in the container's environment.
* **CMD [“npm”, “start”]** - Set the default container command. This tells Docker to run npm start when you start the container.

**Note:** The following is the expected structure of your Dockerfile.A screenshot of a computer

AI-generated content may be incorrect.

Next, let’s create the backend.env file.  
Inside the env directory, add a new file named backend.env. This file will store the MongoDB username and password used by the backend service to connect to the database.

Example:

A screenshot of a computer

AI-generated content may be incorrect.

Finally, let’s define the **backend service** in the docker-compose.yml file.

A screenshot of a computer

AI-generated content may be incorrect.

* **Backend:** - This is the service name.
* **Build: ./backend** - Tells Docker Compose to build an image for this service using the Dockerfile found in the ./backend directory.
* **./backend** is the path on your machine where the Dockerfile is located.
* **Ports: ’80:80’** - Maps port 80 on your host to port 80 inside the container. It Allows you to visit the backend on http://localhost:80 on your computer.
* **Volumes:** Volumes allow sharing files or directories between your host and the container.
  + - **Logs:/app/logs** - Uses a named volume called logs (managed by Docker), mounted at /app/logs in the container. This keeps log files persistent between container restarts.
    - **./backend:/app** - Mounts your host machine’s ./backend folder into /app in the container. Enables real-time code changes on your host to be reflected inside the container.
    - **/app/node\_modules** - Creates an anonymous volume for /app/node\_modules. Prevents your container’s node\_modules from being overwritten by the host bind mount. Essentially, this isolates the container’s node\_modules directory, so you don’t have to rebuild it every time you change code.
* **Env\_file: ./env/backend.env** - Loads environment variables into the container from the file ./env/backend.env. This keeps secrets and config separate from your source code.
* **Depends\_on: mongodb** - Makes sure the mongodb service is started before this backend service.

**Frontend Container**

The **frontend container setup** consists of two main files:

1. A **Dockerfile** – to define how the frontend image is built
2. A **docker-compose.yml** entry – to configure and run the container

Let’s begin by creating the **Dockerfile** for the frontend service:

* **FROM node** - This sets the base image for your Docker container. It pulls the official Node.js image from Docker Hub. This image has Node.js and npm already installed.
* **WORKDIR /app** - Sets the working directory inside the container. All subsequent commands (like COPY, RUN, etc.) will be run from this directory (/app). If the directory doesn't exist, it will be created.
* **COPY package.json .** - Copies the package.json file into the container. Only the package.json file is copied at this stage. This is a best practice to take advantage of Docker layer caching, so dependencies aren’t reinstalled unnecessarily when the rest of the code changes.
* **RUN npm install** - Installs project dependencies listed in package.json . It runs npm install inside the container to install all Node modules. These will be saved inside the container's file system.
* **COPY . .** - Copies the rest of your application files into the container. The ‘.’ means copy everything from your project directory on your host machine into the working directory (/app) in the container.
* **EXPOSE 3000** - Documents that the container listens on port 3000. This doesn’t publish the port; it just tells Docker what port your app will use. When you run the container with -p 3000:3000, it maps this port to your host.
* **CMD [“npm”, “start”]** - Specifies the default command to run when the container starts. This tells the container to run npm start, which usually runs your main Node.js app (like node index.js or whatever is in your package.json under "start").

Note: Place the above Dockerfile inside your frontend folder.

A screenshot of a computer

AI-generated content may be incorrect.

Finally, add the frontend configuration to your Docker Compose file. The configuration should include the following:

* **Frontend** - **Defines a service** named frontend. This is typically a React, Vue, or other frontend framework containerized to run in development or production.
* **Build:** ./frontend - Instructs Docker to build an image from the Dockerfile located in the ./frontend directory (relative to the location of the docker-compose.yml file). This will refer to the dockerfile we made above.
* **Ports:**
  + **‘3000:3000’** - Maps port 3000 on the host machine to port 3000 inside the container. Useful if your frontend app runs on port 3000 (as React apps do by default), so you can access it by going to http://localhost:3000 on your browser.
* **Volumes:** 
  + **./frontend/src:/app/src** - Mounts a volume from the host's ./frontend/src directory into the container’s /app/src directory. This allows live code changes on the host to be reflected immediately in the container (hot reloading). This is used in development setups to avoid rebuilding the image on every code change.
* **Stdin\_open:** true - Keeps the stdin (standard input) open even if not attached. This is needed if the container needs to run in interactive mode, often helpful during development or debugging.
* **Tty: true** - Allocates a pseudo-TTY (terminal). Like stdin\_open, it helps with interactive sessions, especially if you're running a development server that logs interactively.
* **Depends\_on:** 
  + Backend - Specifies that the frontend service depends on the backend service. This means Docker Compose will start the backend container first.

**Note:** Your complete Docker Compose file should look like the following:A screenshot of a computer

AI-generated content may be incorrect.

**Application deployment**

Now that we’ve prepared all the essential files for deploying our application, we can proceed with the deployment and access the app via localhost:3000 in your browser.

**Step 1: Log in to Docker**

Before deploying, ensure you are logged into Docker. Here are two ways to log in:

1. **Via VS Code Terminal:**
   * Run docker login
   * You will be prompted to enter your Docker Hub username and password.
2. **Via Docker Desktop:**
   * Open Docker Desktop
   * Click the profile icon in the top-right corner
   * Select **Sign in**
   * Enter your Docker Hub credentials

**Note:** Once logged in, VS Code will recognize your Docker session as long as it is connected to the same Docker daemon.

**Step 2: Deploy the Application**

After logging in, run the following command in the VS Code terminal:

‘**docker-compose up -d’**

Here’s what happens when you run docker-compose up -d:

1. **Reads the docker-compose.yml file**  
   Docker Compose looks for the docker-compose.yml file in the current directory.
2. **Builds images**  
   For any service with a build: key, Docker builds the image using the specified Dockerfile.
3. **Creates containers**  
   Containers are created for each service defined in the compose file.
4. **Starts containers in detached mode**  
   The -d flag runs the containers in the background, so logs won’t be displayed in the terminal.

Once this process completes, you should see output similar to the following in your terminal:

A screenshot of a computer

AI-generated content may be incorrect.

You can also run **‘docker ps’** in the terminal to see all active and running containers. See example below:A screenshot of a computer

AI-generated content may be incorrect.

Now for the best part — open your browser and navigate to localhost:3000. The application shown below should appear.

A screenshot of a computer

AI-generated content may be incorrect.

To shut down all your containers, run docker-compose down in the terminal. This command will stop and remove all the containers.A screenshot of a computer

AI-generated content may be incorrect.

Another important point to remember is that we added volumes to our backend application. This means that if you shut down all containers and recreate them, your stored data will persist within the application. See below:A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**✅ Final Notes**

* This project demonstrates how to use Docker Compose to deploy a full-stack Node.js app.
* Containers are loosely coupled, making it easy to scale or replace services independently.
* This setup supports live code updates (via bind mounts) and data persistence (via volumes).