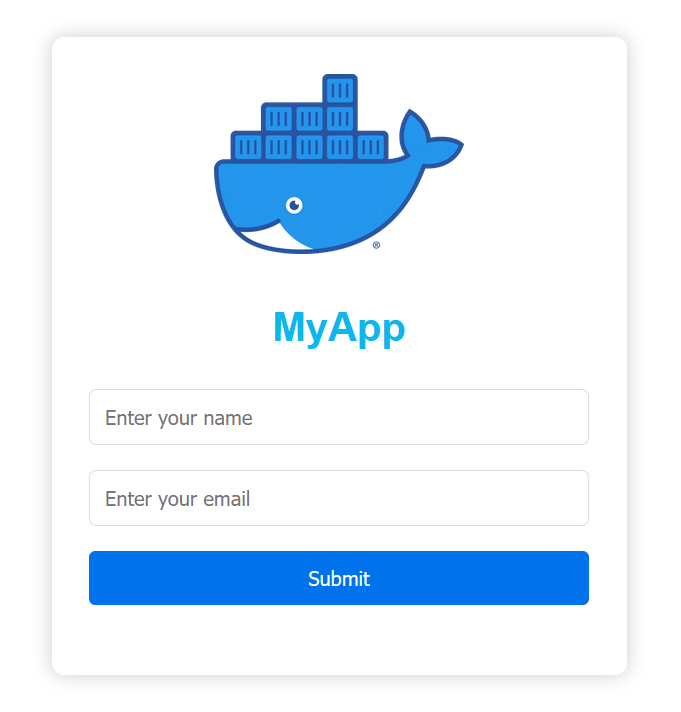
Dockify

|  |
| --- |
| Transform your local app into a Dockerized app |



**Dockify** is a hands-on project designed to showcase the **power of Docker** in building, containerizing, and deploying a full-stack web application from scratch to the cloud.

🔹 **Frontend**: Static HTML page to capture user input (Username and Email).  
🔹 **Backend**: Node.js server using Express.js framework.  
🔹 **Database**: MongoDB containerized via Docker for reliable data storage.

**🚀 Project Flow — All Through Docker:**

1. **Local Development & Initial Setup**:
   * Built and tested the Node.js app locally.
   * Used **Docker containers** to run MongoDB and Mongo Express UI during development.
2. **Containerizing the Application**:
   * Wrote a **Dockerfile** to containerize the Node.js backend.
   * Defined a **docker-compose.yml** to orchestrate **multi-container** setup (App + MongoDB + Mongo Express).
   * Built and ran all services together on **Docker Desktop**.
3. **Local Multi-Container Testing**:
   * Verified seamless interaction between the containers using **Docker networking**.
   * Ensured user data could be stored, retrieved, and viewed through Mongo Express — all within containers.
4. **Production Image Creation**:
   * Optimized the Docker image using **multi-stage builds** for smaller size.
   * Pushed the final production image to **AWS ECR (Elastic Container Registry)**.ss
5. **Cloud Deployment Using Docker and ECS**:
   * Deployed the Docker image as an **ECS Fargate service**.
   * Integrated the service with an **Application Load Balancer (ALB)** for public access.
   * Maintained MongoDB separately in a Dockerized environment on an EC2 instance.
6. **End Result**:
   * Fully containerized infrastructure from local machine to cloud.
   * Achieved a **cloud-native deployment** without managing servers manually — only containers!

Let's move on to the practical section;

**MyApp 1.0 - Setting Up Local Web Application**

**1. Create a Custom Docker Network**

Before starting any containers, created a custom Docker network to ensure clean and secure communication between containers.

Command:

***docker network create myapp-network***

✅ This creates an isolated network named myapp-network.

Benefits:

* MongoDB and future containers (like the Node.js app) can communicate using container names.
* Cleaner architecture compared to using the default bridge network.

Since both mongodb and mongo-express are inside myapp-network,  
containers can talk to each other using container names (like mongodb) instead of using IP addresses.

**2. Start MongoDB Container Separately**

Pulled and ran MongoDB Docker container with authentication enabled.

Command:

**docker run -d --name mongodb -p 27017:27017 \**

**--network myapp-network \**

**-e MONGO\_INITDB\_ROOT\_USERNAME=admin \**

**-e MONGO\_INITDB\_ROOT\_PASSWORD=password \**

**mongo**

* **-d:** Run container in detached mode.
* **--name mongodb:** Name the container.
* **-p 27017:27017:** Map local port 27017 to container port.
* **--network myapp-network :** Custom network.
* **-e:** Pass environment variables for MongoDB root user.

✅ MongoDB running locally at **localhost:27017**.

**3. Start Mongo Express (Optional Verification)**

Also launched **Mongo Express** to visually verify MongoDB connection.

Command:

**docker run -d --name mongo-express -p 8081:8081 \**

**-e ME\_CONFIG\_MONGODB\_ADMINUSERNAME=admin \**

**-e ME\_CONFIG\_MONGODB\_ADMINPASSWORD=password \**

**-e ME\_CONFIG\_MONGODB\_SERVER=mongodb \**

**-e ME\_CONFIG\_MONGODB\_URL=mongodb://admin:password@localhost:27017/admin \**

**-e ME\_CONFIG\_BASICAUTH="false"**

**mongo-express**

* Mongo Express accessible at **http://localhost:8081.**
* Verified database and collections through browser UI.

**4. Set Up Local Node.js Project**

**Initialize npm Project:**

Initialized a new Node.js project using npm to create a package.json file:

Commands:

**npm init -y**

* -y flag auto-generates a basic package.json with default settings.
* package.json is critical to **manage project dependencies** and scripts.

**Install Necessary Node.js Packages:**

Installed the required libraries for building the backend server:

Command:

**npm install express body-parser mongodb dotenv cors**

Dependency:

**express** : A fast, minimalist web framework for Node.js. It simplifies handling HTTP routes, requests, and responses.

**body-parser** : Middleware to parse incoming request bodies, especially JSON payloads from forms or APIs. Works with Express.

**mongodb** : The official MongoDB client for Node.js. It allows the server to connect to a MongoDB database, perform CRUD operations.

**cors** : Middleware to enable **Cross-Origin Resource Sharing** (CORS). It allows the server to accept requests from different origins (domains), which is necessary when frontend and backend are hosted separately.

**dotenv:** Loads environment variables from a .env file into process.env. Helps manage sensitive information like database URLs, passwords, and ports securely without hardcoding them into the code.

After installation:

* The node\_modules/ folder created (contains all packages).
* The package.json updated with the installed dependencies.

**Project Structure:**

myapp/

── index.html (Frontend page)

── server.js (Backend server)

── package.json (Node.js dependencies)

✅ **Important Fix:**

I got some connection issues between the mongodb and MyApp while submit the details;Fix using below steps;

TI have updated the MongoDB connection string to include **authSource=admin** for proper authentication.

**const url = process.env.MONGO\_URL || "mongodb://admin:password@localhost:27017/ admin";**

**Why localhost is used instead of mongodb in; mongodb://admin:password@localhost:27017**

In the connection string:

**mongodb://admin:password@localhost:27017**

localhost here refers to your local machine (your Windows or Linux system) — not the Docker container name.

Because at this stage:

* **MongoDB** is running as a Docker container, but **you mapped its port (27017) to your host machine** using:
* -p 27017:27017 : This **port binding** exposes the MongoDB service **directly to your host** at localhost:27017.
* So, your Node.js app — which is running **locally on your machine** (not inside a Docker container yet) —  
  must connect using:

**mongodb://admin:password@localhost:27017/admin**

**localhost** correctly points to MongoDB because the Docker container port is exposed to your host.

**When would we use mongodb instead of localhost?**

Later, when both Node.js (server.js – MyApp container) and MongoDB run inside the same Docker network,  
then they will communicate by their Docker container names.

**5. Run Backend Server**

Command:

**node server.js**

Server started at http://localhost:3000.

Logged: Connected to MongoDB and Server running.

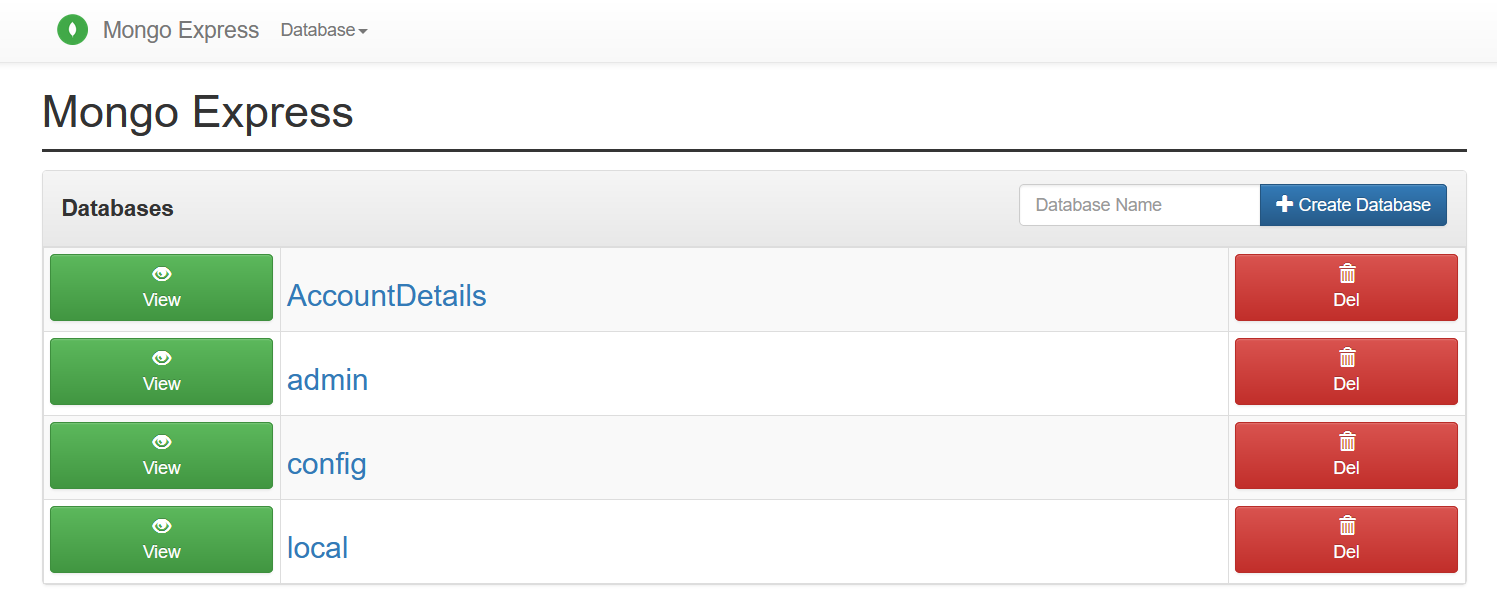
**6. Access Application in Browser**

* Opened http://localhost:3000.
* Displayed HTML form from index.html.
* Submitted form with Name and Email.
* Data inserted successfully into MongoDB.

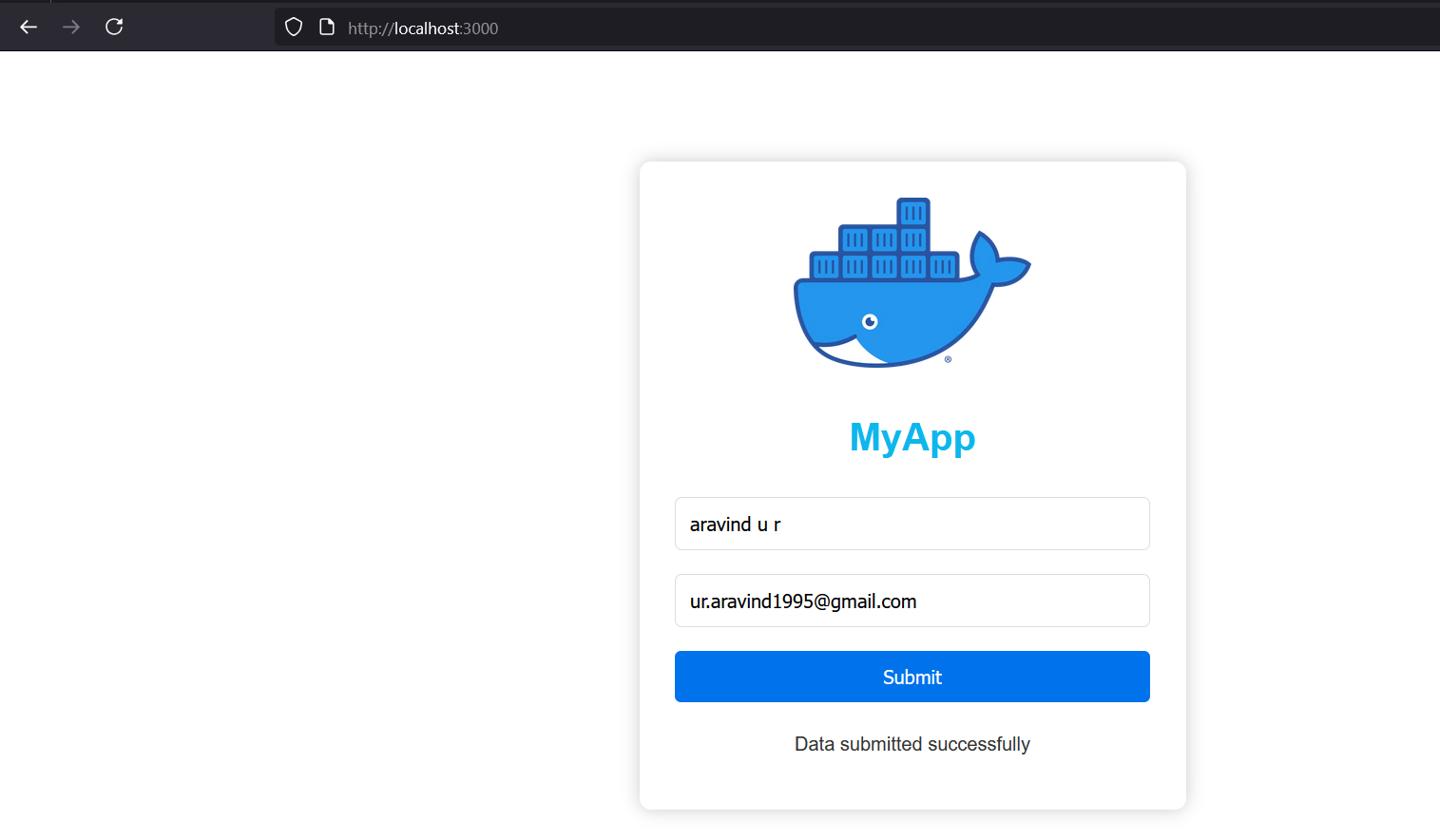
✅ Accessed http://localhost:3000/data to verify all inserted user records**.**

**Verify:**

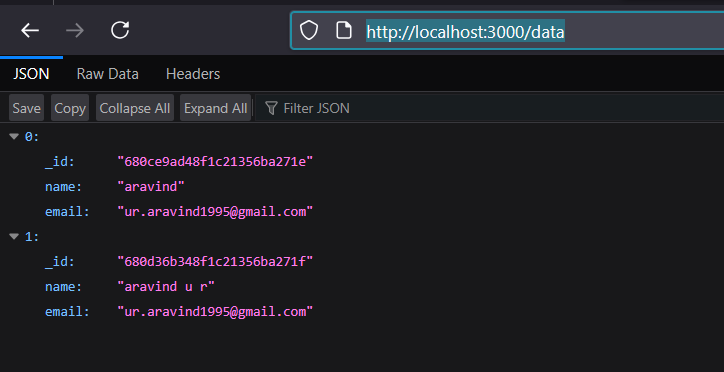
* Mongo Express UI showing AccountDetails database.



* Browser showing index.html form.



* Browser showing /data JSON response.



**MyApp 1.1 - Dockerizing Node.js App and Combining Services**

**1. Create Dockerfile to Dockerize Node.js App (MyApp)**

Package the **Node.js app** (server.js + index.html) into a **Docker image**.  
This ensures the app can run anywhere — laptop, server, or cloud — without setup issues.

**Dockerfile:**

***FROM node:18***

***WORKDIR /app***

***COPY package\*.json ./***

***RUN npm install***

***COPY . .***

***EXPOSE 3000***

***CMD ["node", "server.js"]***

|  |  |
| --- | --- |
| **Section** | **Purpose** |
| **FROM node:18** | Use official Node.js v18 image as the starting point (comes preinstalled with Node and npm). |
| **WORKDIR /app** | Sets /app as working directory inside the container. |
| ***COPY package.json ./*\*** | Copies package.json and package-lock.json first to optimize Docker layer caching. |
| **RUN npm install** | Installs all required Node.js dependencies. |
| **COPY . .** | Copies all project files (server.js, index.html, etc.) into the container. |
| **EXPOSE 3000** | Documents that the container listens on port 3000 (useful for running with port mappings). |
| **CMD ["node", "server.js"]** | Defines the command to start the app inside the container. |

The Dockerfile is the single source of truth.  
Whether you build manually (docker build) or using Compose (build: .),  
it is the same process and same Dockerfile — only the method of triggering the build differs.

✅ Dockerfile = Build instructions  
✅ docker-compose.yml = How to orchestrate multiple services (and optionally build)

**2. Create docker-compose.yml to Combine MongoDB + Mongo Express and build MyApp**

The docker-compose.yml file is created to **orchestrate multiple containers**   
MongoDB (Database), Mongo Express (DB UI), and MyApp (Node.js backend)   
in a single, automated, and networked environment.

✅ Instead of manually running multiple docker run commands,  
✅ docker-compose up starts **all services together** with correct connections, configurations, and dependencies.

**docker-compose.yml**

***version: '3.8'***

***services:***

***mongodb:***

***image: mongo:latest***

***container\_name: mongodb***

***ports:***

***- "27017:27017"***

***volumes:***

***- mongodb\_data:/data/db***

***networks:***

***- app-network***

***restart: always***

***environment:***

***MONGO\_INITDB\_ROOT\_USERNAME: admin***

***MONGO\_INITDB\_ROOT\_PASSWORD: password***

***mongo-express:***

***image: mongo-express:latest***

***container\_name: mongo-express***

***ports:***

***- "8081:8081"***

***restart: always***

***environment:***

***- ME\_CONFIG\_MONGODB\_ADMINUSERNAME=admin***

***- ME\_CONFIG\_MONGODB\_ADMINPASSWORD=password***

***- ME\_CONFIG\_MONGODB\_SERVER=mongodb***

***- ME\_CONFIG\_MONGODB\_URL=*** ***mongodb://admin:password@mongodb:27017/?authSource=admin***

***- ME\_CONFIG\_BASICAUTH=false***

***networks:***

***- app-network***

***depends\_on:***

***- mongodb***

***myapp:***

***build: .***

***container\_name: myapp***

***ports:***

***- "3000:3000"***

***networks:***

***- app-network***

***depends\_on :***

***- mongodb***

***environment :***

**- MONGO\_URL= mongodb://admin:password@mongodb:27017/?authSource=admin**

***networks:***

***app-network:***

***driver: bridge***

***volumes:***

***mongodb\_data:***

***driver: local***

**Core explanation:**

* **version: '3.8' :**Specifies the Compose file version.
* **services:**

This section defines multiple containers (services) that make up the application stack.

**mongodb Service**

|  |  |
| --- | --- |
| **Field** | **Explanation** |
| **image: mongo:latest** | Pulls the latest MongoDB image from Docker Hub. |
| **container\_name: mongodb** | Assigns a custom name to the MongoDB container. |
| **ports: "27017:27017"** | Maps MongoDB's default port 27017 from container to host, allowing local access. |
| **volumes: mongodb\_data:/data/db** | Mounts a persistent volume at /data/db inside the container for database storage. |
| **networks: app-network** | Connects MongoDB to a custom bridge network named app-network. |
| **restart: always** | Automatically restarts the container if it crashes or the Docker daemon restarts. |
| **environment:** | Sets environment variables for MongoDB authentication. Creates a root user with username admin and password password. |

MongoDB service acts as the database server for the application.

**mongo-express Service**

|  |  |
| --- | --- |
| **Field** | **Explanation** |
| **image: mongo-express:latest** | Pulls the latest Mongo Express image. |
| **container\_name: mongo-express** | Names the container mongo-express. |
| **ports: "8081:8081"** | Exposes port 8081 for browser access to MongoDB admin UI. |
| **restart: always** | Ensures auto-restart if the container fails. |
| **environment:** | Provides credentials and connection settings to MongoDB. |
| **depends\_on: mongodb** | Ensures MongoDB starts before Mongo Express. |
| **networks: app-network** | Connects Mongo Express to the same network as MongoDB. |

Mongo Express is a **web-based UI** to easily manage MongoDB databases through a browser (<http://localhost:8081>).

**myapp Service (Node.js App)**

|  |  |
| --- | --- |
| **Field** | **Explanation** |
| **build: .** | Builds the Node.js app image using the Dockerfile in the current directory. |
| **container\_name: myapp** | Names the container myapp. |
| **ports: "3000:3000"** | Exposes port 3000 to access the Node.js server through browser (http://localhost:3000). |
| **depends\_on: mongodb** | Ensures MongoDB is started first before MyApp. |
| **environment:** | Sets environment variables for MongoDB connection (MONGO\_URL). |
| **networks: app-network** | Connects MyApp to the same custom network for internal communication. |
| **networks: app-network** | Connects Mongo Express to the same network as MongoDB. |

MyApp is the backend server that connects to MongoDB and handles user form submissions.

* **networks:**

|  |  |
| --- | --- |
| **Field** | **Explanation** |
| **app-network** | Defines a custom network for the project. |
| **driver: bridge** | Uses Docker's default bridge networking driver, allowing container-to-container communication by container name. |

All services are attached to app-network, enabling internal name-based communication (mongodb, mongo-express, myapp) instead of relying on localhost/IPs.

* **volumes:**

|  |  |
| --- | --- |
| **field** | **Explanation** |
| **mongodb\_data** | Defines a named Docker volume. |
| **driver: local** | Stores the volume data on the host machine’s filesystem. |

Ensures MongoDB database files persist even if the MongoDB container is removed, rebooted, or re-created.

Critical for data durability.

**3. Run the Full Application (Using Docker Compose)**

Inside your project folder:

***docker-compose up --build***

✅ This will:

* Build the Docker image for myapp.
* Pull MongoDB image (if not already pulled).
* Start MongoDB container + MyApp container together.
* Set up the internal network (myapp-network).
* Map the ports to localhost (3000 for app, 27017 for MongoDB).

**Important Connection Note :**

In docker-compose.yml,  
we now use:

mongodb://admin:password@mongodb:27017/?authSource=admin

* because Node.js app and MongoDB are inside the same Docker network (myapp-network).
* so, they can talk using container names (mongodb).

**📈 Final Working Status After Stage 2**

|  |  |  |
| --- | --- | --- |
| **Service** | **URL** | **Purpose** |
| MyApp (Frontend + Backend) | http://localhost:3000 | Access form, submit data |
| MongoDB | localhost:27017 | Database, not directly browsed |
| Mongo Express (Optional if added) | http://localhost:8081 | Visual DB management |

We have successfully:

* Created a Dockerfile to containerize the Node.js backend
* Used Docker Compose to manage MyApp, MongoDB, and Mongo Express
* Built and tested the image locally
* Prepared to push the image to Docker Hub or AWS ECR for deployment

This sets the foundation for deploying the full-stack app using container orchestration platforms like AWS ECS or Kubernetes.

🔄 **Normal Docker Workflow**

When you're not using Docker Compose, here's what typically happens:

1. You create a Dockerfile for your app (e.g., a Node.js app).
2. You build the image manually using:

***docker build -t myapp .***

1. You run the container from the image using:

***docker run -p 3000:3000 myapp***

📦 **Docker Compose Workflow**

When you use Docker Compose, here's what changes:

1. You still create a Dockerfile for your app (this step doesn't change).
2. But instead of manually building and running, you write a docker-compose.yml file which:
   * Specifies the services, like MongoDB, your Node.js app (myapp), and Mongo Express.
   * For your app service, you just write:

***build: .***

This tells Docker Compose to use your Dockerfile to build the image automatically.

* + Compose also maps ports, sets environment variables, volumes, networks, etc.

1. Then you run everything together with:

***docker-compose up --build***

* + This builds the image (if not already built).
  + Starts all the containers.
  + Connects them via a shared network.
  + Applies volumes and other configurations.

|  |  |  |
| --- | --- | --- |
| **Action** | **Without Compose** | **With Compose** |
| Build image | docker build | build: . in docker-compose.yml |
| Run container | docker run | docker-compose up |
| Manage multiple services | Manual & complex | Centralized in Compose file |
| Networks/Volumes | Manual | Automatic/custom via Compose |

**MyApp1.2 : Dockerize the Entire App Using a Multi-Stage Dockerfile**

**Objective:**

Create a **Docker image** for the entire Node.js app using a **multi-stage build** to:

* Keep the final image **small and production-ready**
* Exclude unnecessary development tools and files
* Prepare for pushing to **AWS ECR**

**Dockerfile:**

# Stage 1: Build

**FROM node:18 as build**

#Create a work directory

**WORKDIR /app**

# Copy only package files first for better caching

**COPY package\*.json ./**

# Install only production dependencies (omit dev)

**RUN npm install --omit=dev**

# Stage 2: Final Lightweight Image

**FROM node:alpine**

#Create a work directory

**WORKDIR /app**

# Copy installed node\_modules from build stage

**COPY --from=build /app/node\_modules ./node\_modules**

# Copy all remaining app files

**COPY . .**

# Expose port used by the app

**EXPOSE 3000**

# Start the app

**CMD ["node", "server.js"]**

**Detailed Explanation: Multi-Stage Dockerfile:**

**# Stage 1: Build**

**FROM node:18 as build**

* Purpose: This is the build stage, using a full Node.js base image (node:18) with build tools like npm, node-gyp, and compilers.
* Why: It contains everything we need to install and compile packages—especially native modules if any.

**WORKDIR /app**

* Sets the working directory inside the container to /app.
* All subsequent commands will run in this folder.

**COPY package\*.json ./**

* Copies package.json and package-lock.json from your host to the container.
* Best practice for caching dependencies, so Docker doesn’t reinstall every time if the dependencies don’t change.

**RUN npm install --omit=dev**

* Installs only production dependencies (excluding dev tools like linters, test frameworks).
* --omit=dev ensures the image is lean and secure, and avoids extra packages that aren’t needed in production.

**# Stage 2: Final Lightweight Image**

**FROM node:alpine**

* Starts the second stage using the lightweight Alpine image (~5MB).
* Why: Keeps the final image very small by excluding build tools and using a tiny base.

**WORKDIR /app**

* Again, sets the working directory in this new final image.

**COPY --from=build /app/node\_modules ./node\_modules**

* Copies only the production-ready node\_modules folder from the build stage.
* Prevents us from reinstalling them again here.

**COPY . .**

* Copies all remaining files (like server.js, HTML files, etc.) from your local folder into the container.
* This includes your app code.

**EXPOSE 3000**

* Informs Docker and external tools (like ECS or Compose) that the app listens on port 3000.
* This doesn’t actually publish the port—docker run -p or Compose handles that.

**CMD ["node", "server.js"]**

* Sets the command to run when the container starts.
* In this case, it runs your app with Node.js by executing server.js.

**How Multi-Stage Build Reduces Image Size:**

| **Feature** | **Description** |
| --- | --- |
| **Separation of concerns** | Dependencies are installed in the first stage and only essential runtime files are copied to the final image |
| **No build tools in production** | The final node:alpine image doesn’t have npm, compilers, or dev tools—keeping the image smaller and safer |
| **Alpine base image** | Much smaller than Debian/Ubuntu-based images (5MB vs ~100MB) |
| **No devDependencies** | Skipping dev packages saves storage, improves performance, and reduces attack surface |

Then we can move to AWS Migration for our MyApp.

**MyApp1.2** : **Dockify App AWS Migration (Production deployment - Backend: Node.js + MongoDB)**

To migrate a Dockerized full-stack web application (Node.js backend serving HTML + MongoDB database) from local development to a production-grade deployment using:

* **Amazon ECS Fargate** (for MyApp backend)
* **Amazon EC2** (for MongoDB & Mongo Express)
* **Amazon ECR** (for storing container images)

✅ **Step 1: MongoDB & Mongo Express Setup on EC2**

**1. Launched an Ubuntu EC2 instance:**

* Type: t2.micro
* Security Group: Custom rules added
  + ✅ Port 22 (SSH)
  + ✅ Port 27017 (MongoDB)
  + ✅ Port 8081 (Mongo Express)
  + ✅ Temporarily allowed 0.0.0.0/0 for ECS testing

**2. Installed Docker & Ran Containers:**

Install the docker and docker-compose in EC2

Create a docker-compose.yml with below script.

***version: '3.8'***

***services:***

***mongodb:***

***image: mongo:latest***

***container\_name: mongodb***

***ports:***

***- "27017:27017"***

***volumes:***

***- mongodb\_data:/data/db***

***networks:***

***- app-network***

***restart: always***

***environment:***

***MONGO\_INITDB\_ROOT\_USERNAME: admin***

***MONGO\_INITDB\_ROOT\_PASSWORD: password***

***mongo-express:***

***image: mongo-express:latest***

***container\_name: mongo-express***

***ports:***

***- "8081:8081"***

***restart: always***

***environment:***

***- ME\_CONFIG\_MONGODB\_ADMINUSERNAME=admin***

***- ME\_CONFIG\_MONGODB\_ADMINPASSWORD=password***

***- ME\_CONFIG\_MONGODB\_SERVER=mongodb***

***- ME\_CONFIG\_MONGODB\_URL=mongodb://admin:password@mongodb:27017/?authSource=admin***

***- ME\_CONFIG\_BASICAUTH=false***

***networks:***

***- app-network***

***depends\_on:***

***- mongodb***

***networks:***

***app-network:***

***driver: bridge***

***volumes:***

***mongodb\_data:***

***driver: local***

**3. Verified:**

* Mongo Express accessible at browser on http://<EC2\_PUBLIC\_IP>:8081
* Created DB: AccountDetails
* Created collection: users

✅ **Step 2: MyApp Image Push to ECR**

In this stage, we push our optimized Docker image (created using multi-stage build) to Amazon Elastic Container Registry (ECR) so that it can be used in ECS or anywhere else via pull.

**1. Rebuilt image after editing:**

* index.html: Replaced localhost with ***window.location.origin***
* server.js:
  + Used MONGO\_URL with fallback string
  + Added authSource=admin
  + Added error handling for undefined DB

**🔍 Problem Summary:**

* Your index.html currently has:

***const response = await fetch("http://localhost:3000/submit", {***

* ✅ This works **locally**, but **not in AWS ECS**.
* ❗ After deploying to ECS (Fargate with Public IP), **you only get the ECS public IP after deployment**.
* But you need that IP **before** building and pushing the Docker image (which contains the updated HTML).

**✅ Solution Options : Use JavaScript to Dynamically Detect Host**

Inside index.html, use:

***const apiHost = window.location.origin;***

***const response = await fetch(`${apiHost}/submit`, {***

This dynamically uses the **domain or IP** serving the HTML to send the request. So, whether it's:

* http://localhost:3000
* http://<ECS\_PUBLIC\_IP>:3000
* http://<ALB\_DNS>

…it will **just work**.

✅ **Best for current setup**, especially since I have serving HTML from the **same Express server**.

**🔍 Why Use It?**

This makes your frontend **automatically match** whatever domain/IP/port it's being served from.  
Since your index.html is served from the **same backend Express server**, this guarantees that:

* GET / serves index.html
* POST /submit hits the same host

👉 **No need to hardcode any IP or domain.**

**2. Tagged & Pushed to ECR:**

• Create an ECR Repository:

* Go to the AWS ECR Console
* Click "Create repository".
* Choose Private repository.
* Set the repository name to: myapp

Leave other settings as default and click "Create repository".

• Tag the Docker Image:

Assume the image was built locally with:

***docker tag myapp:latest <aws\_account\_id>.dkr.ecr.<region>.amazonaws.com/myapp:latest*** *(use correct tag here)*

• Authenticate Docker to ECR:

Use AWS CLI:

***aws ecr get-login-password --region <region> \***

***| docker login --username AWS \***

***--password-stdin <aws\_account\_id>.dkr.ecr.<region>.amazonaws.com***

• Push the Image to ECR:

***docker push <aws\_account\_id>.dkr.ecr.<region>.amazonaws.com/myapp:latest***

Notes

• Only the MyApp image is pushed to ECR, not MongoDB or Mongo Express.

• Once pushed, this image can be pulled using Docker or ECS services.

• The existing docker-compose.yml file can be reused locally by replacing build: section with an image: tag.

myapp:

image: <aws\_account\_id>.dkr.ecr.<region>.amazonaws.com/myapp:latest

No need to use --build if you're pulling from ECR.

✅ **Step 3: ECS Fargate Deployment**

**Step 1: Create an ECS Cluster**

1. Go to **ECS Console** → “Clusters” → “Create Cluster”
2. Choose **"Networking only" (Fargate)**
3. Give it a name, e.g., myapp-cluster
4. Click **Create**

**Step 2: Create a Task Definition**

1. Go to **ECS → Task Definitions** → “Create new”
2. **Launch type**: Fargate
3. **Name**: myapp-task
4. **Task Role**: Leave default (or create IAM role if needed)
5. **Task Size**:
   * 0.5 vCPU
   * 1 GB Memory
6. **Add container**:
   * Name: myapp
   * Image: your\_ecr\_image\_url
   * Port Mapping: 3000
7. **Environment Variable**:
   * MONGO\_URL: ***mongodb://admin:password@<EC2\_PRIVATE\_IP>:27017/?authSource=admin***
8. **Click Create**

**Step 3: Run a Task (Service) with Public IP**

1. Go to **ECS → Clusters → myapp-cluster**
2. Click **“Create Service”**
   * Launch Type: Fargate
   * Task Definition: myapp-task
   * Service Name: myapp-service
   * Number of tasks: 1
3. **Networking**:
   * VPC: Choose default or existing
   * Subnets: Select at least 2
   * **Assign Public IP: ENABLED**
   * Security Group: Allow inbound traffic on port **3000**
4. Click **Next → Create Service**

**Testing & Debugging !!!**

After successfully deploying the MyApp container on **ECS Fargate** and MongoDB on **EC2**, the application frontend (index.html) loaded properly in the browser. However, the data submission process (form POST to /submit) failed consistently with a **500 Internal Server Error**.

**❌ Error 1: Form Submission Failing (500 Internal Server Error)**

**🔍 Symptom:**

* HTML form loaded correctly at:

***http://<ECS\_PUBLIC\_IP>:3000***

Upon submitting the form:

* Browser console showed:

***POST http://<ECS\_PUBLIC\_IP>:3000/submit 500 (Internal Server Error)***

***Uncaught (in promise): Could not establish connection.***

**🧪 Debugging Step 1: Inspect Browser Console**

Used DevTools → Console tab:

* Observed that the request was being sent correctly
* JavaScript code used window.location.origin (✅ as expected)

**🧪 Debugging Step 2: CloudWatch Logs**

Opened **ECS → Tasks → Logs** in CloudWatch. Found the following error:

***Error inserting data: TypeError: Cannot read properties of undefined (reading 'collection')***

***at /app/server.js:41:29***

**🔎 Diagnosis:**

* This line points to:

***db.collection('users').insertOne(...)***

Meaning: db is undefined, because MongoDB connection failed silently

**✅ Fix 1: Added DB Connection Check in server.js**

***if (!db) {***

***return res.status(500).json({ message: 'Database not connected' });***}

✔ Prevents crashing if MongoDB connection is not established

**❌ Error 2: MongoDB Connection Not Established**

Even after adding a safe fallback, logs now showed:

***MongoRuntimeError: Unable to parse <EC2\_PRIVATE\_IP>:27017 with URL***

**🧠 Root Cause:**

In ECS Task Definition, the environment variable was mistakenly left as:

env

***MONGO\_URL = mongodb://admin:password@<EC2\_PRIVATE\_IP>:27017/admin?authSource=admin***

➡️ ECS treated <EC2\_PRIVATE\_IP> as a literal string, not a variable

**✅ Fix 2: Updated ECS Task Definition**

* Created **new task revision (Rev 4)**
* Replaced placeholder with actual EC2 private IP (e.g., 172.31.27.251)
* Final value:

env

***MONGO\_URL = mongodb://admin:password@172.31.27.251:27017/admin?authSource=admin***

* Updated ECS Service to use this task revision

**🔍 Verification in Logs**

After redeploying, CloudWatch Logs confirmed:

Using MONGO\_URL: ***mongodb://admin:password@172.31.27.251:27017/admin?authSource=admin***

✅ Connected to MongoDB

🚀 Server running on port 3000

**❌ Error 3: ECS Service Create Failed (Initial Cluster Creation)**

Before any of the above:

* While creating ECS cluster:

CREATE\_FAILED: Unable to assume the service linked role

✅ Fix:

* Created the missing ECS-linked IAM role manually:

bash

aws iam create-service-linked-role --aws-service-name ecs.amazonaws.com

**🧪 Final Testing (After Fixes)**

* Opened browser → http://<ECS\_PUBLIC\_IP>:3000
* Filled the form → clicked submit
* ✅ Data was successfully inserted into MongoDB
* ✅ Verified via Mongo Express (http://<EC2\_PUBLIC\_IP>:8081)
* Collection users inside DB AccountDetails showed the new record

**✅ Debugging Summary Table**

| **Error** | **Diagnosis** | **Resolution** |
| --- | --- | --- |
| 500 Internal Server Error | db was undefined due to failed connection | Added guard if (!db) + logged errors |
| MongoRuntimeError: Unable to parse <EC2\_PRIVATE\_IP> | Used placeholder <EC2\_PRIVATE\_IP> in URL | Replaced with real EC2 private IP in Task Definition |
| ECS Cluster CREATE\_FAILED | Missing ECS service-linked role | Created role via AWS CLI |
| Form submission not reaching MongoDB | Security group misconfiguration (not mentioned earlier but checked) | Ensured EC2 SG allows 27017 from ECS CIDR |

This step ensured that:

* Backend was resilient
* Logging gave visibility
* ECS and EC2 network were properly configured
* All deployment variables were accurately injected

**Step 4: Final Deployment Confirmation**

* Form submission ✅
* Data stored in EC2-hosted MongoDB ✅
* http://<EC2 Public IP>/data and Mongo Express showed inserted records
* Verified all endpoints:
  + /submit
  + /data
  + Mongo Express GUI

**Conclusion:**

This project demonstrates a complete, end-to-end DevOps workflow by building, containerizing, and deploying a full-stack web application — from local development to a production-ready cloud infrastructure using Docker and AWS services.

**🐳 Docker Phase: Building and Containerizing the App**

In the initial phase, the focus was on learning, experimenting, and building the core application using:

* **Node.js** for the backend logic (server.js)
* A simple HTML page as the frontend (index.html)
* **MongoDB** as the database, with **Mongo Express** as an admin UI

Once the app logic was complete, Docker was used to:

* Create container images for each component
* Use docker-compose.yml to orchestrate:
  + MyApp (Node.js app)
  + MongoDB
  + Mongo Express

Key Docker milestones included:

* Multi-stage Dockerfile to optimize image size and isolate build tools from runtime
* Dynamic environment variable handling using MONGO\_URL
* Ensuring static frontend was served correctly from the Node.js backend
* Local testing with docker-compose up for rapid feedback

This hands-on Docker experience formed a strong foundation for understanding container networking, multi-container communication, image versioning, and troubleshooting logs from individual containers.

**☁️ AWS Phase: Migrating and Scaling the Application**

Once the app was stable locally, the second phase focused on **migrating the containerized application to AWS** using scalable and secure services:

**✅ EC2 Setup:**

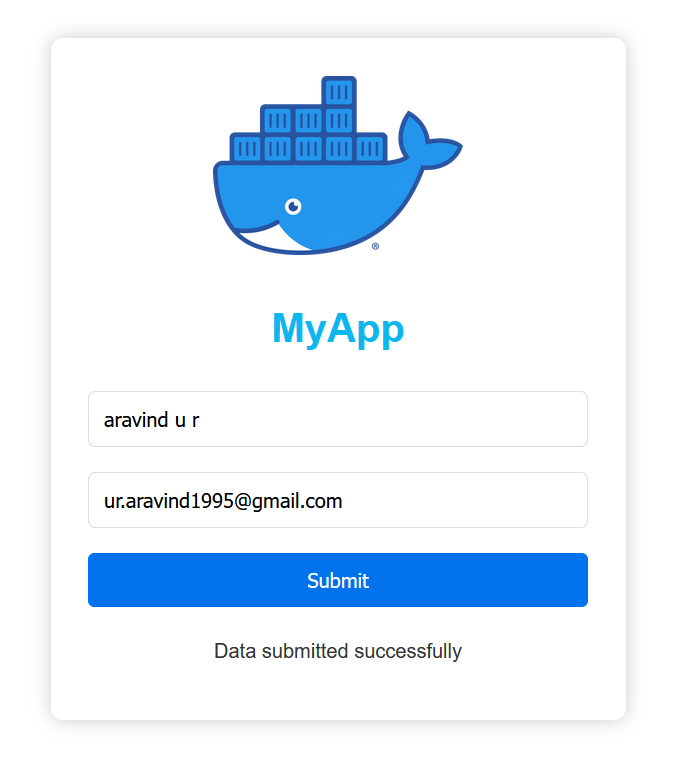
* MongoDB and Mongo Express were deployed as containers on an EC2 Ubuntu instance
* Configured Docker, Security Groups, and verified public/private access

**✅ ECR:**

* Final production-ready Docker image of MyApp was built and pushed to **Amazon Elastic Container Registry (ECR)**
* Multiple tags (1.0, 1.2) were used to track versions

**✅ ECS Fargate:**

* Deployed MyApp as a stateless container on **AWS ECS Fargate**
* Managed task definitions, environment variables, port mappings, and public IP exposure
* Applied networking best practices via Security Groups and Subnet configuration



--End--