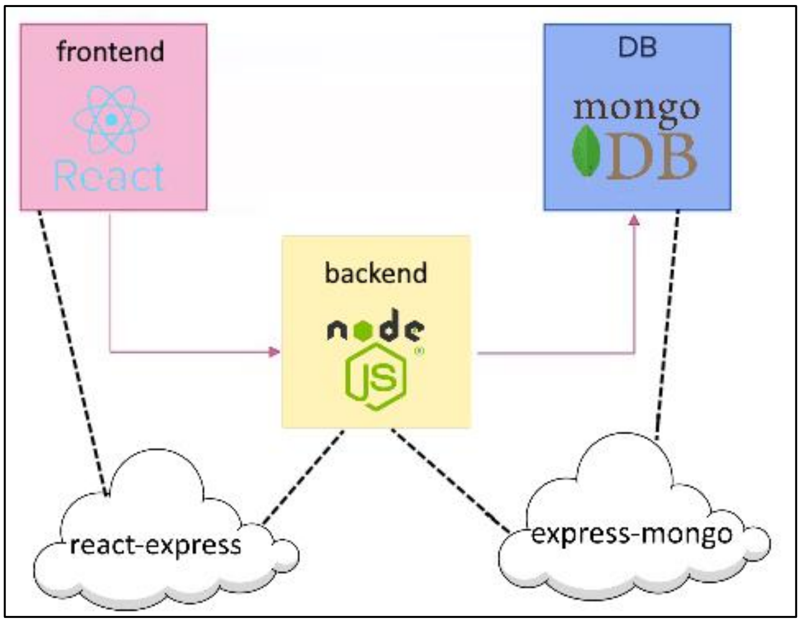
Workshop

Workshop for the ["Software Engineering and DevOps" course @ SoftUni](#).

1. ToDo App

The **TODO app** (provided in the **resources**) is a simple app for **adding tasks**. It is a **React application** with a **NodeJS backend** and a **MongoDB database**. We should **create separate Docker containers** and **connect them in two networks** as shown below to make the **three containers work together**:



Then, we should set up a **workflow**, in which **whenever** we make **changes** to our **codebase** and **push** those **changes** to our **GitHub repository**, the following should happen:

- **GitHub Actions** will **detect** the **push event** and **trigger** the **workflow**.
- The **workflow checks** out our **codebase**, sets up **Docker Buildx**, and **logs** into **Docker Hub**.
- For **each job** (**`build-frontend`** and **`build-backend`**), the workflow will build a **Docker image** using the **Dockerfile** provided in the context directory (**`./frontend`** or **`./backend`**).
- After building the Docker images, the workflow will push these images to the Docker Hub under your username and with the tag frontend:latest or backend:latest respectively.

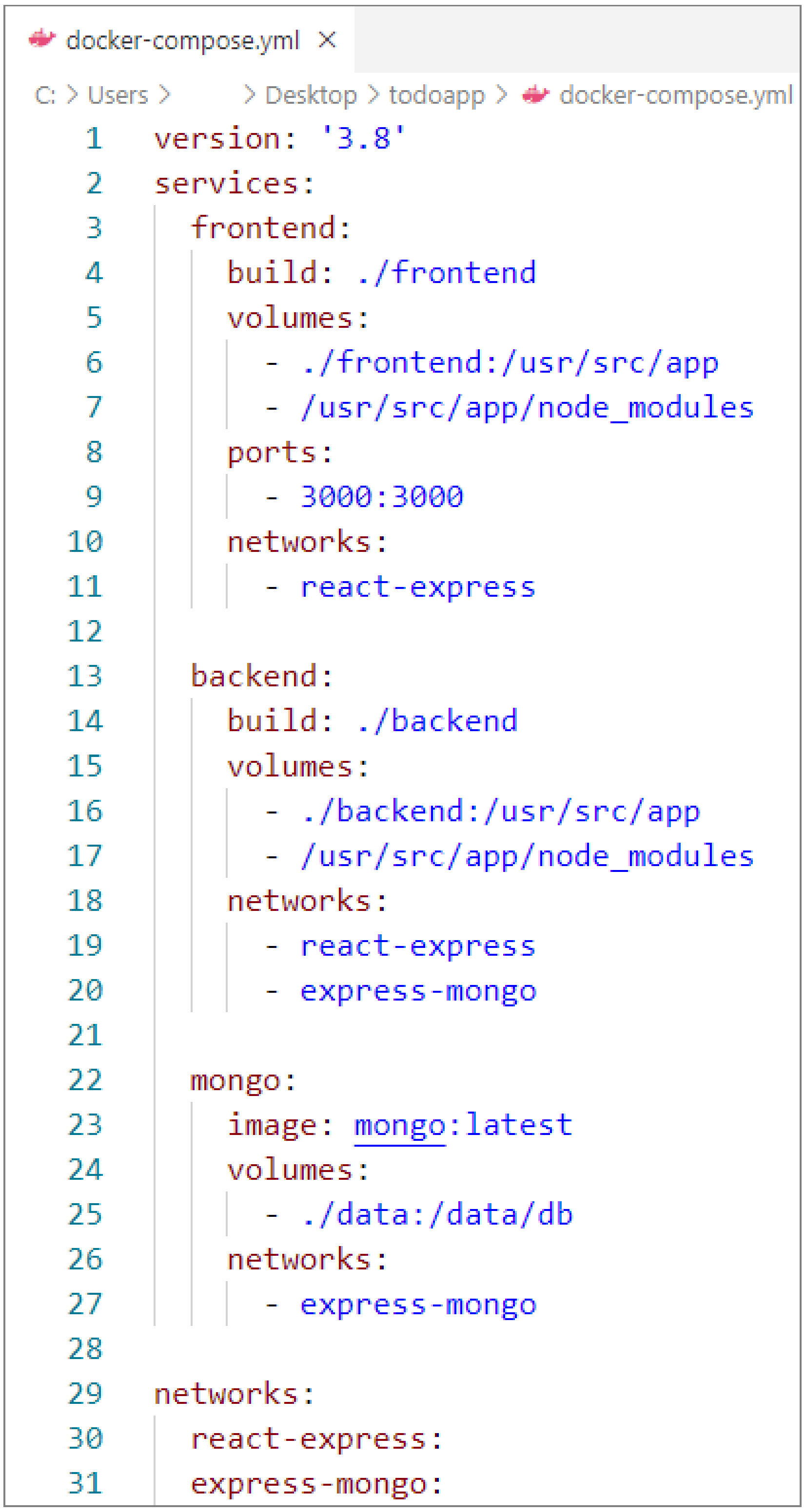
This way, we can be sure that our Docker images on Docker Hub always represent the latest state of our codebase on the **`main`** branch.

Now, let's start with the **`docker-compose`** file. We have some requirements that we have to follow:

- Specify the version of Docker Compose file format being used;
- List the services (containers) that make up our application;
  - frontend
  - backend
  - mongo
- Build the Docker images for the **frontend** and **backend** services using the **Dockerfiles** found in the **`./frontend`** and **`./backend`** directories
- Mount directories from our host machine into the Docker container;
  - Map the **`./frontend`** directory on our host to **`/usr/src/app`** in your **frontend** container;

  - Map the **`./backend`** on our host to **`/usr/src/app`** in your backend container;
- Map ports between the Docker host and containers.
  - For the frontend service, it's mapping port **`3000`** on our host to port **`3000`** in the container.
- List the networks the service should connect to.
  - The **frontend** service is connected to the **`react-express`** network;
  - The **backend** service is connected to both **`react-express`** and express-mongo networks;
  - **MongoDB** is connected to **`express-mongo`** network.
- Use the **`mongo:latest`** image directly from Docker Hub, instead of building from a local Dockerfile.
  - Mount the **`./data`** directory from our host to **`/data/db`** inside the container, where MongoDB stores its data.
- Define the networks to be used by services.

The file should look like this:

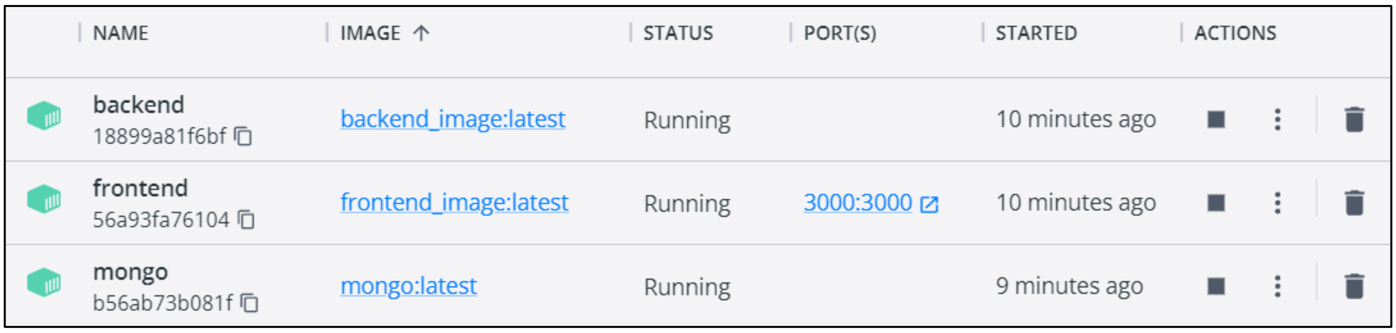


```yaml
version: '3.8'
services:
  frontend:
    build: ./frontend
    volumes:
      - ./frontend:/usr/src/app
      - /usr/src/app/node_modules
    ports:
      - 3000:3000
    networks:
      - react-express

  backend:
    build: ./backend
    volumes:
      - ./backend:/usr/src/app
      - /usr/src/app/node_modules
    networks:
      - react-express
      - express-mongo

  mongo:
    image: mongo:latest
    volumes:
      - ./data:/data/db
    networks:
      - express-mongo

networks:
  react-express:
  express-mongo:
```

Now let's test if everything works correctly. Open a terminal in the project directory and run the following command:

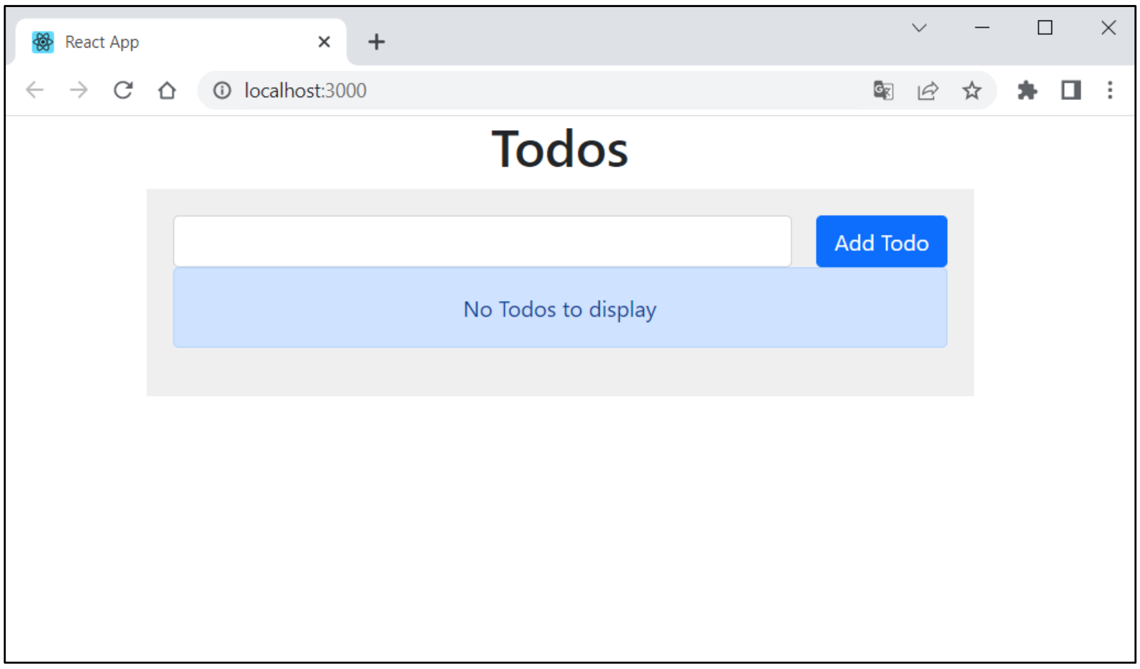
```
PS C:\Users\      \Desktop\todoapp> docker-compose up -d
```

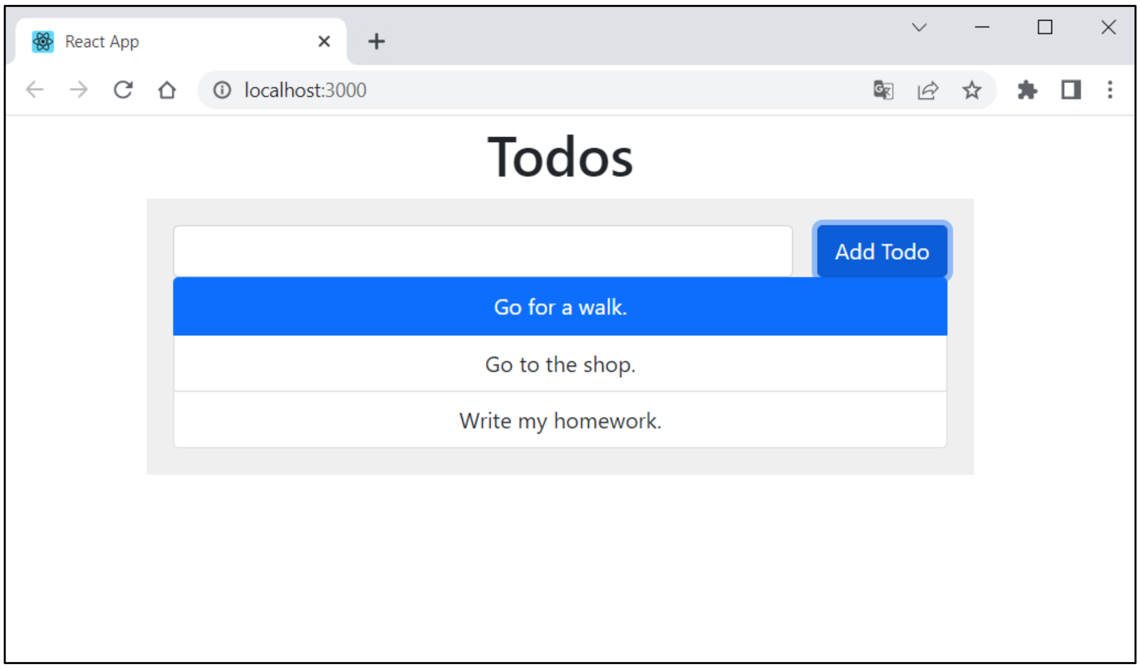
The containers should appear in Docker:



NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
backend 18899a81f6bf	backend_image:latest	Running		10 minutes ago	
frontend 56a93fa76104	frontend_image:latest	Running	3000:3000	10 minutes ago	
mongo b56ab73b081f	mongo:latest	Running		9 minutes ago	

When ready, we should be able to **add tasks** to the **TODO list in the app**:



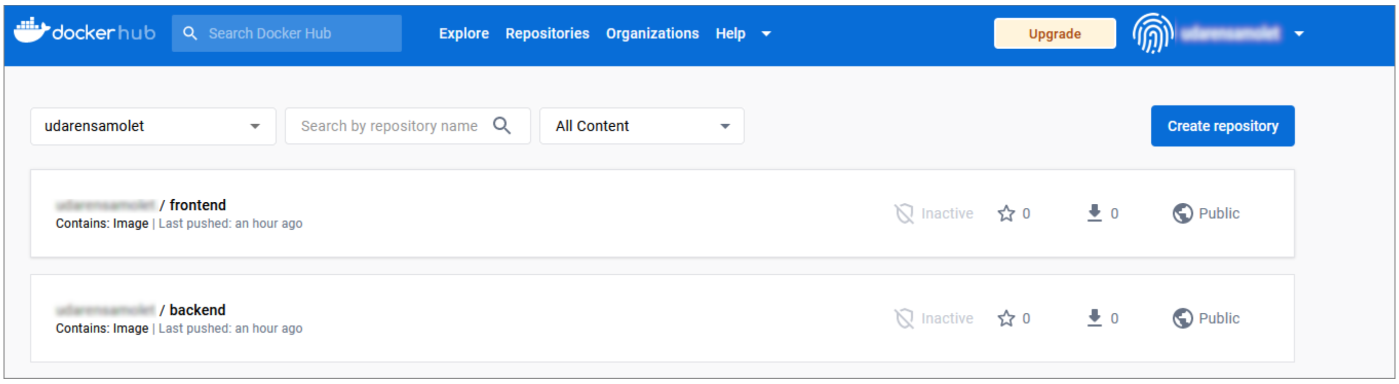


Now, let's define the CI/CD workflow in GitHub actions.

First, create a **GitHub** repo that will hold the folders for the **frontend** and **backend** services.

Then, go to Docker Hub and if you don't have an account, create one.

After you have set up your account, create two repos – one for each service:

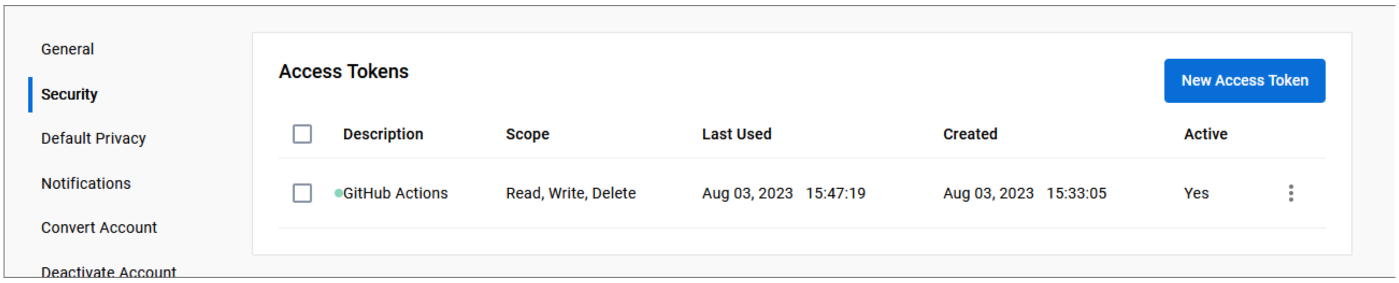


docker hub | Search Docker Hub | Explore | Repositories | Organizations | Help | Upgrade | [redacted]

udarensamolet | Search by repository name | All Content | Create repository

[redacted] / frontend Contains: Image | Last pushed: an hour ago	Inactive	0	0	Public
[redacted] / backend Contains: Image | Last pushed: an hour ago	Inactive	0	0	Public

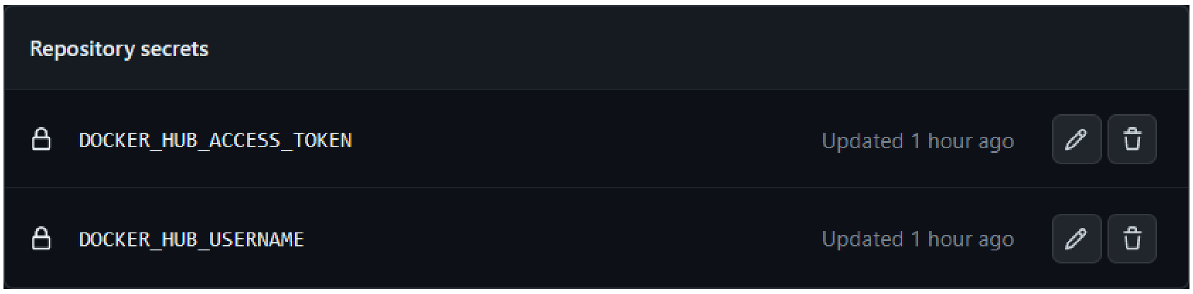
Then, while still in Docker Hub, go to your **Account** → **Account Settings** → **Security** → **Access Tokens** and create a new access token that we will use for our GitHub repo:



Access Tokens

Description	Scope	Last Used	Created	Active
GitHub Actions	Read, Write, Delete	Aug 03, 2023 15:47:19	Aug 03, 2023 15:33:05	Yes

Now, go to your GitHub repo and add this access token as a secret. You should add another secret, containing your Docker account:



Repository secrets

DOCKER_HUB_ACCESS_TOKEN	Updated 1 hour ago
DOCKER_HUB_USERNAME	Updated 1 hour ago

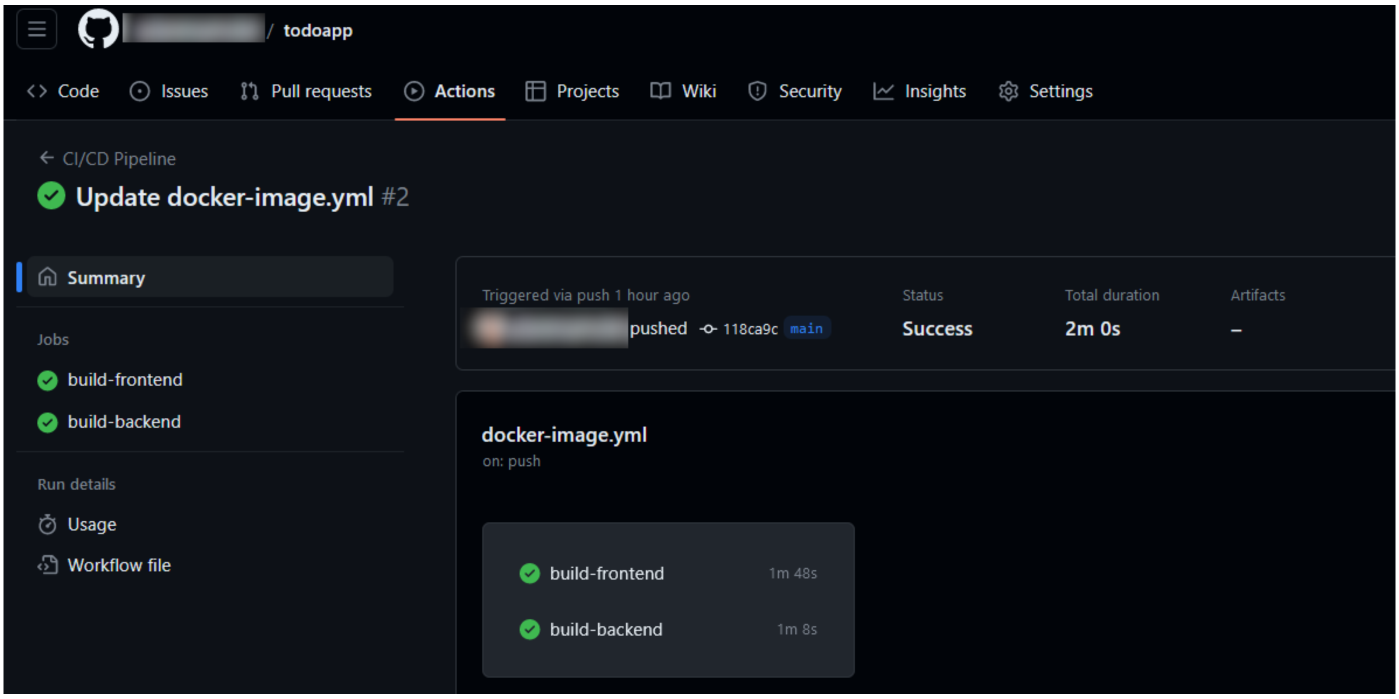
Now, let's set up the workflow:

- Define the name of the workflow;
- Specify that the workflow should be triggered when changes are pushed to the main branch of the repository;
- Define the jobs:
  - One job for each service;
  - Set the type of runner;
  - Define the steps to be taken in order to execute the tasks in the job:
    - Check out the repo
    - Set up QEMU
    - Set up Docker Buildx
    - Login to Docker Hub
    - Build and push

The **`main.yaml`** file should look like this:

```yaml
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]

jobs:
  build-frontend:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up QEMU
      uses: docker/setup-qemu-action@v1

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v1

    - name: Login to DockerHub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_HUB_USERNAME }}
        password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}

    - name: Build and push
      uses: docker/build-push-action@v2
      with:
        context: ./frontend
        file: ./frontend/Dockerfile
        push: true
        tags: [redacted]/frontend:latest
  build-backend:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v2

    - name: Set up QEMU
      uses: docker/setup-qemu-action@v1

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v1

    - name: Login to DockerHub
      uses: docker/login-action@v1
      with:
        username: ${{ secrets.DOCKER_HUB_USERNAME }}
        password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}

    - name: Build and push
      uses: docker/build-push-action@v2
      with:
        context: ./backend
        file: ./backend/Dockerfile
        push: true
        tags: [redacted]/backend:latest
```

Finally, the build should be successful:



[redacted] / todoapp

Code | Issues | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

← CI/CD Pipeline

Update docker-image.yml #2

Summary

Jobs
- build-frontend
- build-backend

Run details
- Usage
- Workflow file

Triggered via push 1 hour ago — [redacted] pushed 118ca9c main

Status: Success

Total duration: 2m 0s

Artifacts: –

docker-image.yml
on: push

build-frontend 1m 48s

build-backend 1m 8s

This way we've set up **two services**, **frontend** and **backend**, each in its own **Docker container**. If you have more microservices, you can follow a similar approach for each.

Then, we used the **docker-compose** file to **orchestrate** these services, defining how they should run together. In this case, we have the frontend and backend services communicating over one network and the backend and MongoDB communicating over another.

The GitHub Actions workflows we've defined are the CI/CD system. On every push to the main branch, it automatically builds the Docker images for your frontend and backend. In a real-life scenario, we would likely also have some testing steps in this workflow to ensure our code is working as expected before it's built into an image and pushed to Docker Hub.

As we're deploying in a local Docker environment, this could be as simple as running **`docker-compose up`** with the updated images.