Лабораторна робота 1
Тема: Створення Docker контейнеру для backend додатку

Концепції та варіанти використання Docker для розробників

Якщо подивитися на причини створення Docker та його популярність, то це здебільшого завдяки його здатності скорочувати час налаштування середовищ, де працюють та розробляються програми.

Просто подивіться, скільки часу потрібно налаштувати середовище, де у вас є API Node та Express для бекенду, якому також потрібен Mongo.

А потім, якщо ви додасте свої програми або фронтенд, вам, можливо, доведеться налаштувати там також кілька елементів. І це лише початок. Потім, коли ваша команда зростає, і у вас є кілька розробників, які працюють над одним фронтендом, бекендом, і тому їм потрібно налаштувати однакові ресурси у своїх локальних середовищах для тестування, як ви можете гарантувати, що кожен розробник запускатиме одне й те саме середовище, ресурси, не кажучи вже про однакові версії?

Усі ці сценарії добре підкреслюють сильні сторони Docker, де його цінність полягає у встановленні контейнерів із певними налаштуваннями, середовищами та навіть версіями ресурсів. І просто введіть кілька команд, щоб Docker автоматично налаштував, встановив та запустив ваші ресурси, і найчастіше набагато швидше.

Тож давайте коротко розглянемо основні компоненти.

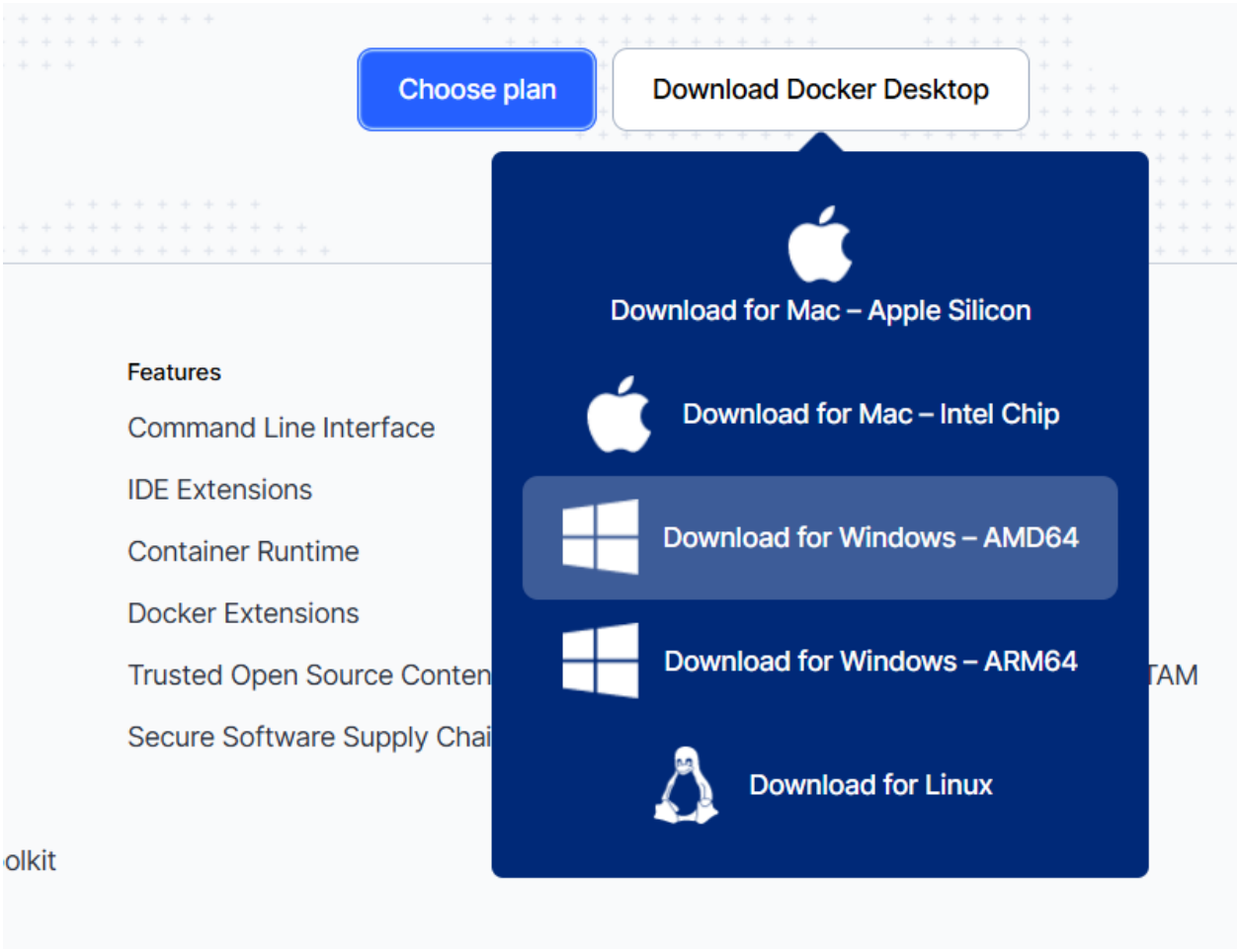
Контейнер – це, по суті, місце, де знаходиться ваша програма або певний ресурс. Наприклад, у вас може бути база даних Mongo в одному контейнері, потім фронтенд-програма React, і, нарешті, ваш сервер Node/Express у третьому контейнері. Потім у вас є образ, з якого побудовано контейнер. Образ містить всю інформацію , необхідну нашому контейнеру для створення контейнера точно так само в будь-яких системах. Це як рецепт. Потім у вас є томи, які зберігають дані ваших контейнерів. Тож, якщо ваші програми знаходяться в контейнерах, які є статичними та незмінними, дані, що змінюються, знаходяться на томах. І, нарешті, елементи, які дозволяють усім цим елементам взаємодіяти, – це мережа. Так, це звучить просто, але зрозумійте, що кожен контейнер у Docker не має уявлення про існування один одного. Вони повністю ізольовані. Тож, якщо ми не налаштуємо мережу в Docker, вони не матимуть уявлення, як підключитися один до одного.

Концепції та варіанти використання Docker для розробників

Отже, почнемо з Docker. Перше, що вам потрібно зробити, це встановити Docker, і це дуже простий процес.

Просто перейдіть на головний веб-сайт, який називається docker.com.

Як тільки ви туди потрапите, там є кнопка «Почати». Натисніть на цю кнопку, а потім прокрутіть униз, поки не побачите Docker Desktop.

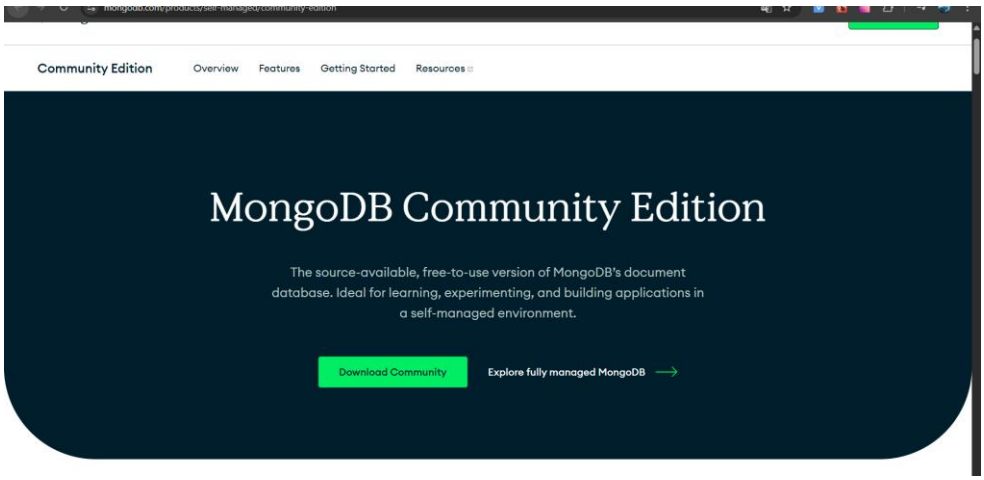


Потім виберіть операційну систему, яку ви використовуєте, завантажте та встановіть.

Після встановлення ви повинні побачити тут логотип або значок кита, і це, по суті, Docker працює. Тож, якщо ви бачите, що Docker Desktop працює, це означає, що все готово.

Ще я б рекомендував створити обліковий запис , а потім переконатися, що ви ввійшли за допомогою свого облікового запису, оскільки ми збираємося використовувати деякі речі, які вимагають входу.

Тож, як тільки ви це налаштуєте, перейдемо до наступного кроку. Крім того, більшість наших програм потребують бази даних MongoDB. Тож, щоб переконатися, і якщо ви також хочете їх протестувати, обов'язково дотримуйтесь інструкцій щодо встановлення MongoDB. Як їх знайти? Ви заходите в Google, вводите mongodb community edition , а потім переходите до «Встановити MongoDB Community Edition». Отже, як тільки ви все це встановили у своїй системі, все готово.



Тож давайте перейдемо до файлів у системі мудл і натиснемо «Resources». І тут у вас є три програми. Одна з них — простий бекенд. Інша — простий фронтенд, тобто, по суті, React-застосунок. А потім у вас є повний бекенд. І це, по суті, програми, які ми будемо використовувати для розгортання.

Ми почнемо з простого бекенду, і ми будемо використовувати ці програми за допомогою Docker. Вона має дуже просте підключення до бази даних і початковий запуск нашого сервера.

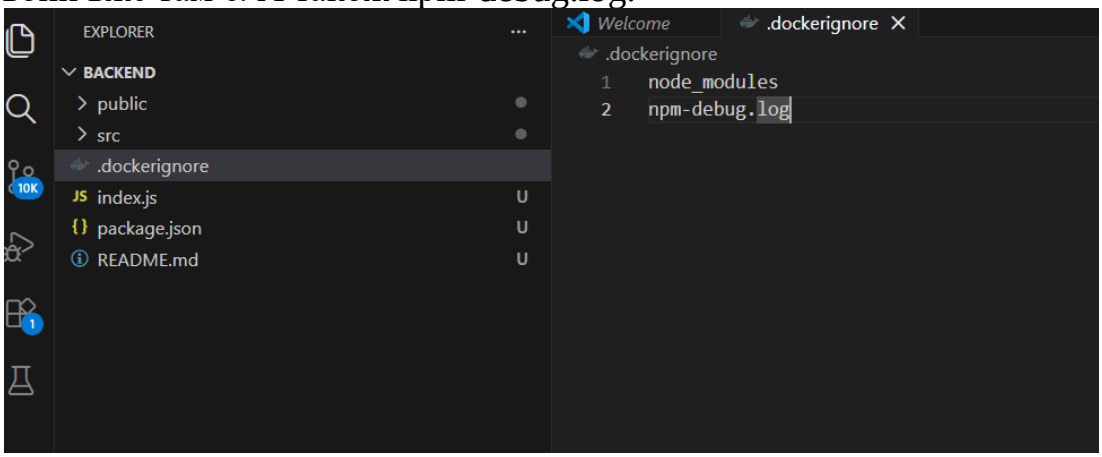
Створення вашого першого образу Docker

Ми розмістимо образ Docker та контейнер, використовуючи проект бекенду в ресурсах.

Перше, що ми зробимо, це створимо нову папку на нашому робочому столі та використаємо проект, який у нас є у файлі вправи backend. А потім відкриємо VS Code в цьому випадку, якщо ви використовуєте щось інше, це також добре.

Створимо два нових файли в цьому проекті. Одним з них буде .dockerignore, а .dockerignore— це, по суті, файл, який повідомляє вам, що не включати в контейнер.

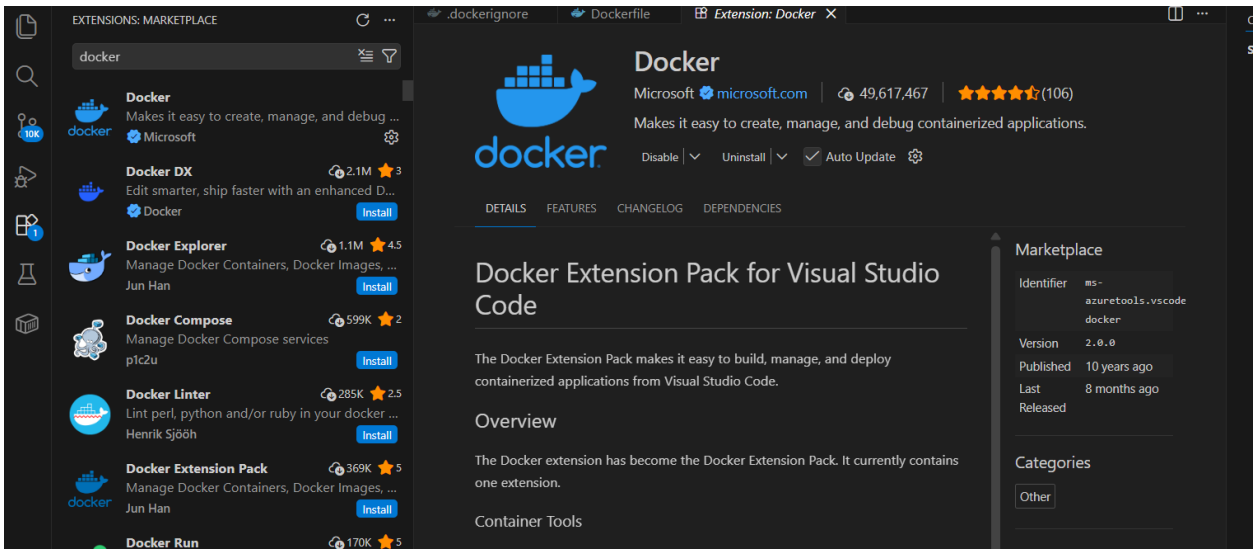
А потім я додам node_modules у файл. Я не хочу, щоб вони були у файлі Docker, тому що він автоматично встановить їх. Тому я не хочу включати їх, якщо вони вже там є. А також npm-debug.log.



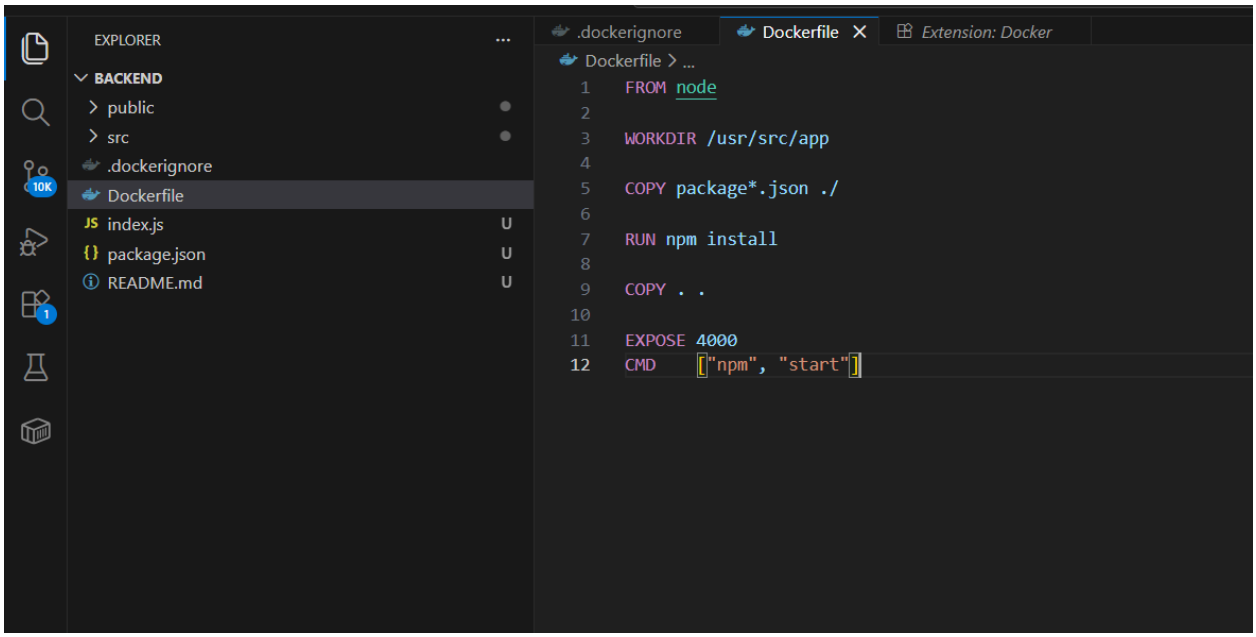
Отже, як тільки ми це зробимо, ми можемо зберегти, і закрити цей файл, а потім ми будемо працювати над найважливішим файлом для цього проекту, яким є файл Docker.

Тож, по суті, метою файлу Docker є створення образу на основі інструкцій вашого файлу Dockerfile.

Є розширення, яке допомагає вам під час створення Docker-файлу. Воно виконує деякі поради та подібні речі під час створення Docker-файлів. Тож встановіть це розширення, якщо ви працюєте з VS-кодом, а якщо ні, то встановіть будь-які інструменти, які можуть допомогти вам з Docker-файлом.



Отже, як тільки ми потрапляємо у файл Docker, Ми збираємося виконати інструкції для образу. Отже, по суті, кожна інструкція, яку ми включимо в цей файл, наказуватиме Docker створити образ.



Кожна команда в цьому Dockerfile має свою роль у створенні контейнера для Node.js застосунку. Розберемо їх по черзі:

1. FROM node
- Вказує базовий образ, з якого буде створено контейнер.
- Тут використовується офіційний образ Node.js з Docker Hub.
- Це означає, що всередині контейнера вже буде встановлений Node.js та npm.

2. WORKDIR /usr/src/app
- Задає робочу директорію всередині контейнера.
- Усі наступні команди (COPY, RUN, CMD) будуть виконуватись відносно цієї директорії.
- Якщо директорії не існує, Docker автоматично її створить.

3. COPY package*.json ./
- Копіює файли package.json та package-lock.json (якщо він є) з локальної машини в контейнер у робочу директорію.
- Це робиться перед копіюванням всього коду, щоб використати кешування шарів Docker: якщо залежності не змінились, npm install не буде виконуватись повторно.

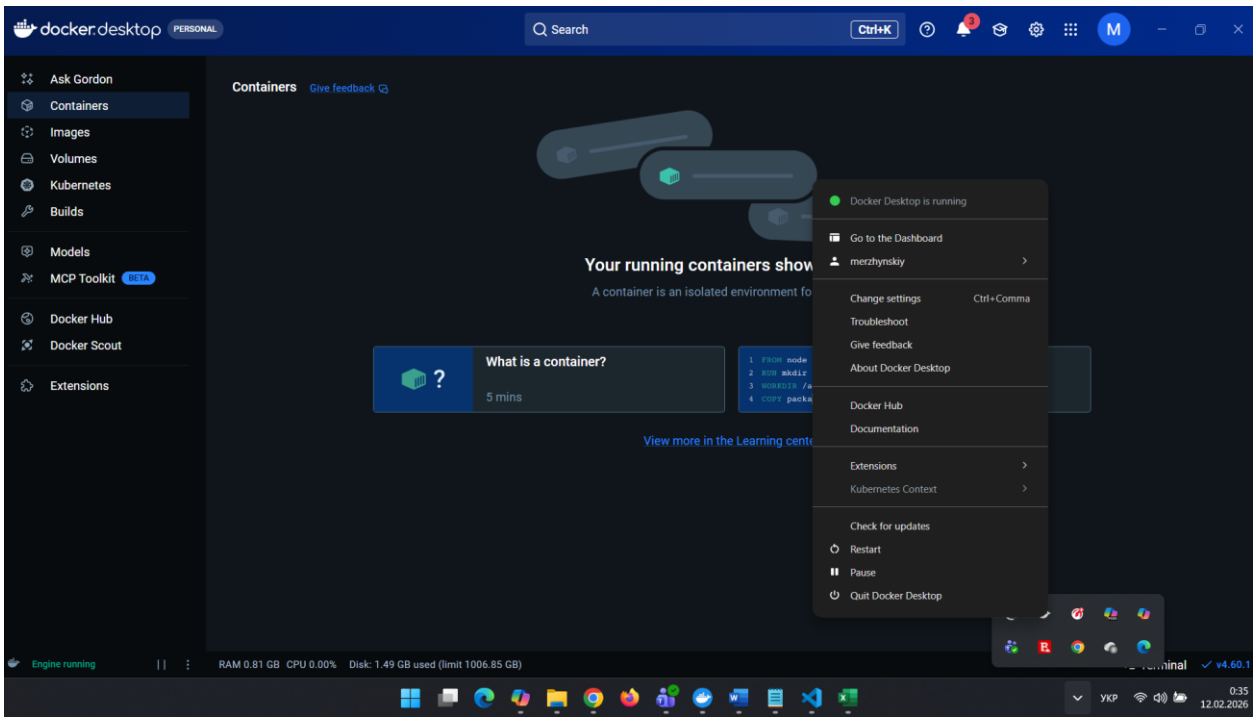
4. RUN npm install
- Виконує встановлення залежностей, описаних у package.json.
- Усі пакети будуть встановлені всередині контейнера.
- Це створює шар образу з готовим node_modules.

5. COPY . .
- Копіює весь проєкт (усі файли з поточної директорії) у контейнер у робочу директорію /usr/src/app.
- Таким чином, код застосунку потрапляє всередину контейнера.

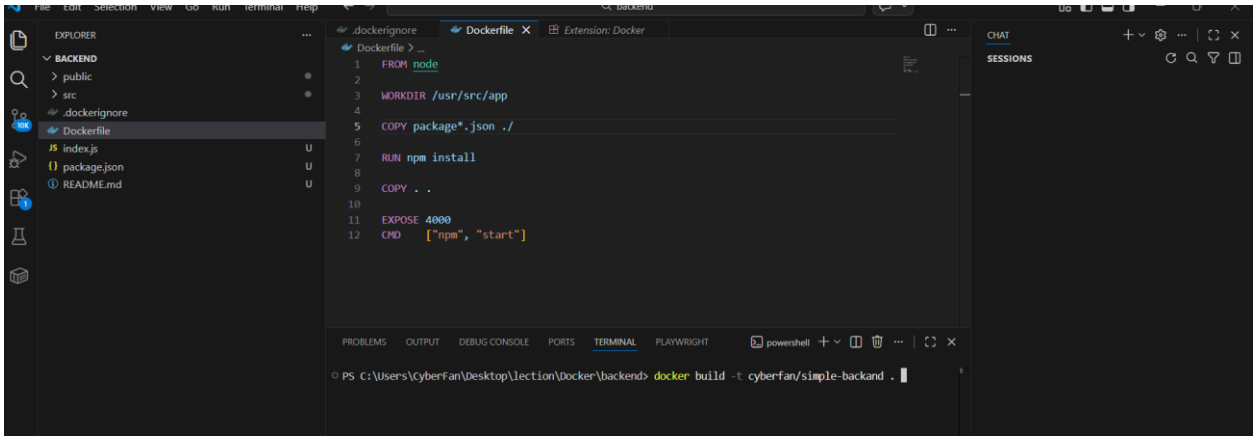
6. EXPOSE 4000
- Декларує, що контейнер слухає порт 4000.
- Це не відкриває порт автоматично, але служить документацією та підказкою для інших сервісів чи оркестраторів (наприклад, Docker Compose або Kubernetes).

7. CMD ["npm", "start"]
- Вказує команду, яка буде виконана при запуску контейнера.
- У цьому випадку запускається npm start, що зазвичай означає старт Node.js застосунку (наприклад, node server.js або react-scripts start, залежно від того, що прописано у package.json).

Тепер, коли у нас є створений Docker-файл для нашого тестового проєкту, давайте створимо з нього образ і протестуємо його. Перше, що вам потрібно зробити, це переконатися, що Docker працює. Якщо у вас немає значка кита у вашій системі, і коли ви його відкриваєте, Docker desktop працює, переконайтеся, що він працює і що він зелений.



Після цього також переконайтеся, що ви ввійшли в систему і все готово. Тож давайте відкриємо термінал у VS code або будь-якому іншому редакторі, який ви використовуєте, або ви можете відкрити свій власний термінал окремо.



Запускаємо таку команду:

docker build -t cyberfan/simple-backend .

Детальний розбір
- docker build
Запускає процес створення Docker-образу на основі інструкцій у Dockerfile.
- -t cyberfan/simple-backend
Опція -t (tag) задає ім’я та тег для створюваного образу.
- cyberfan — це "repository" або namespace (зазвичай ім’я користувача чи організації).

- simple-backend — назва образу.
Разом вони утворюють тег cyberfan/simple-backend:latest (якщо не вказати версію, за замовчуванням використовується latest).
- . (крапка)
Вказує контекст збірки — директорію, з якої Docker бере файли.
У цьому випадку це поточна директорія, де знаходиться Dockerfile та весь код проєкту.
Docker копіює цю директорію у внутрішній "build context" і виконує інструкції з Dockerfile.

Далі запустимо команду:

docker images

Вона використовується для **перегляду списку всіх Docker-образів**, які збережені локально на твоїй машині.
Що саме вона показує:

- **REPOSITORY** — назва репозиторію (наприклад, cyberfan/simple-backand).
- **TAG** — тег образу (якщо не вказати, за замовчуванням буде latest).
- **IMAGE ID** — унікальний ідентифікатор образу.
- **CREATED** — коли образ був створений.
- **SIZE** — розмір образу.

Приклад виводу:
REPOSITORY TAG IMAGE ID CREATED SIZE cyberfan/simple-backand latest a1b2c3d4e5f6 2 minutes ago 150MB node latest 123456789abc 3 days ago 120MB
Таким чином, ця команда дозволяє перевірити, які образи вже є у тебе локально після виконання docker build.

Отже, як запустити контейнер з цього образу? Це дуже проста команда, знову ж таки, як це зробити, в основному, знову ж таки, Docker, завжди починайте з docker

docker run -p 4000:4000 cyberfan/simple-backend

Розбір по частинах

- **docker run**
  Створює та запускає новий контейнер з вказаного образу.
- **-p 4000:4000**
  Опція для пробросу портів (port mapping).
  - Перше число (4000) — порт на твоїй локальній машині (host).
  - Друге число (4000) — порт всередині контейнера, який був оголошений у Dockerfile через EXPOSE 4000.
    Це означає: коли ти відкриєш http://localhost:4000 у браузері,

запит буде перенаправлений у контейнер на його внутрішній порт 4000.

- **cyberfan/simple-backend**
  Назва образу, з якого створюється контейнер.
  Саме цей образ ти зібрав раніше командою docker build.

Що відбувається в результаті

1. Docker створює контейнер із образу cyberfan/simple-backend.
2. Запускає команду CMD ["npm", "start"], яка прописана у Dockerfile (тобто стартує твій Node.js сервер).
3. Сервер всередині контейнера слухає порт 4000.
4. Завдяки -p 4000:4000, ти можеш звертатись до нього з хоста через http://localhost:4000.

Команда

docker ps

виводить список запущених контейнерів у твоїй системі.
Що саме показує:
- CONTAINER ID — унікальний ідентифікатор контейнера.
- IMAGE — образ, з якого контейнер був створений (наприклад, cyberfan/simple-backend).
- COMMAND — команда, яка виконується всередині контейнера (у твоєму випадку npm start).
- CREATED — коли контейнер був запущений.
- STATUS — поточний стан (наприклад, Up 5 minutes).
- PORTS — які порти проброшені (наприклад, 0.0.0.0:4000->4000/tcp).
- NAMES — ім’я контейнера (Docker автоматично генерує випадкове ім’я, якщо ти не задав своє через --name).

```
PS C:\Users\CyberFan\Desktop\lection\Docker\backend> docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS         P
ORTS                                        NAMES
2db2afde4066   cyberfan/simple-backend   "docker-entrypoint.s…"   2 minutes ago   Up 2 minutes   0
.0.0.0:4000->4000/tcp, [::]:4000->4000/tcp   frosty_wiles
PS C:\Users\CyberFan\Desktop\lection\Docker\backend>
```

Команди **docker stop** та **docker kill** обидві використовуються для зупинки контейнерів, але роблять це по-різному:
docker stop
- Надсилає контейнеру сигнал SIGTERM (коректне завершення).
- Це дає процесу всередині контейнера час на "чисте" завершення роботи: закрити з’єднання, зберегти дані, виконати фіналізацію.
- Якщо контейнер не завершився за певний таймаут (за замовчуванням 10 секунд), тоді Docker надсилає сигнал SIGKILL і примусово зупиняє його.

- Використовується у більшості випадків, коли потрібно акуратно зупинити застосунок.
docker kill
- Одразу надсилає сигнал SIGKILL (примусове завершення).
- Контейнер зупиняється миттєво, без можливості виконати будь-які завершальні дії.
- Застосовується у випадках, коли контейнер "завис" або не реагує на docker stop.

Налаштуємо файл конфігурації **docker-compose.yml**

```yaml
version: '1'
services:
  app:
    container_name: app
    restart: always
    build: .
    ports:
      - "4000:4000"
    links:
      - mongo
  mongo:
    container_name: mongo
    image: mongo
    volumes:
      - ./data:/data/db
    ports:
     - "27017:27017"
```

Файл **docker-compose.yml** — це конфігураційний файл, який використовується інструментом **Docker Compose** для опису та запуску багатоконтейнерних застосунків.

Для чого він потрібен на бекенді:

- У бекенд-проєктах часто є кілька сервісів:
  - **Node.js / Express API**
  - **MongoDB база даних**
  - (іноді) **React фронтенд** або інші допоміжні сервіси
- Кожен із них запускається у своєму контейнері.
- docker-compose.yml дозволяє описати всі ці сервіси в одному файлі та керувати ними як єдиною системою.

Основні можливості:

- **Опис сервісів**: визначає, які образи використовувати, які порти відкривати, які змінні середовища передати.

- **Мережа між контейнерами**: автоматично створює спільну мережу, щоб бекенд міг звертатись до бази даних за ім’ям сервісу (наприклад, mongo).
- **Томи (volumes)**: дозволяє зберігати дані поза контейнером (наприклад, щоб база не втрачала дані після перезапуску).
- **Єдиний запуск**: замість того щоб вручну запускати кілька docker run, можна просто виконати docker-compose up і підняти всі сервіси разом.

Що він робить:

- **version: '1'**
  Вказує формат файлу. Сьогодні рекомендують використовувати 3.x, бо він підтримує сучасні можливості.
- **services:**
  Описує набір контейнерів, які треба запустити разом.

Сервіс app:

- **container_name: app** — ім’я контейнера буде app.
- **restart: always** — контейнер автоматично перезапускається, якщо впаде.
- **build: .** — збирає образ із Dockerfile у поточній директорії.
- **ports: "4000:4000"** — проброс порту: локальний 4000 → контейнерний 4000.
- **links: - mongo** — створює зв’язок із контейнером mongo (старий спосіб, зараз краще використовувати networks).

Сервіс mongo:

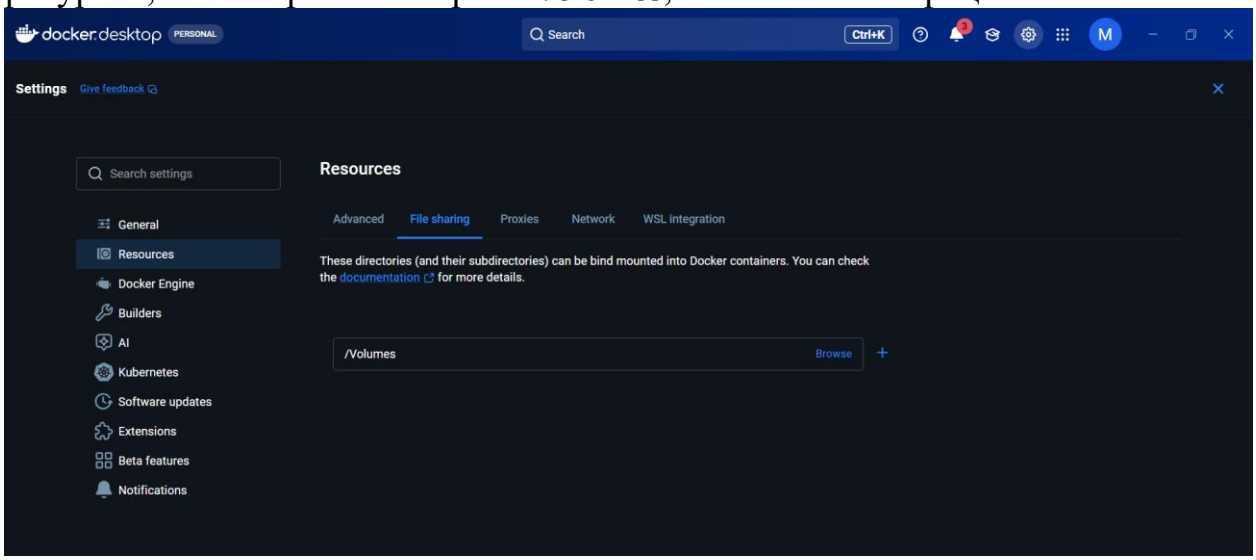
- **container_name: mongo** — ім’я контейнера буде mongo.
- **image: mongo** — використовує офіційний образ MongoDB.
- **volumes: ./data:/data/db** — зберігає дані бази у локальній папці ./data, щоб вони не зникали після перезапуску.
- **ports: "27017:27017"** — проброс порту для доступу до MongoDB з хоста.

Навіщо він потрібен:

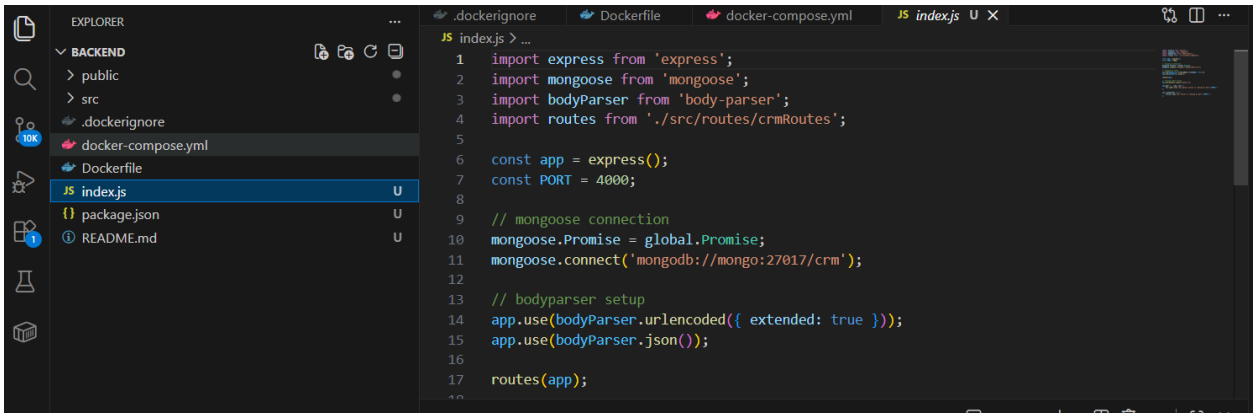
- Дає можливість **запустити бекенд і базу даних разом** однією командою:
  docker-compose up
- Забезпечує **однакове середовище** для всієї команди: кожен розробник отримує Node.js API та MongoDB, налаштовані однаково.
- Автоматично створює мережу між контейнерами, щоб бекенд міг звертатись до MongoDB за ім’ям mongo.

Перш ніж ми продовжимо та протестуємо наш новий Docker-бекенд за допомогою Compose, переконайтеся, що у вас є спільні локальні диски для томів, які знадобляться Docker. Щоб перевірити це, нам потрібно лише зайти в Docker, потім у Параметри , а потім у Resources, File sharing.

Отже, якщо у вас немає нічого подібного до volumes у цих конкретних ресурсах, вам потрібно створити volumes, інакше він не працюватиме.



Також, інша річ, яку обов'язково потрібно зробити у файлі index.js, це переконатися, що підключаєтеся до mongo, тому що якщо ви не підключитеся до mongo, він шукатиме container_name: mongo, але не знайде його. Тож переконайтеся, що ви дійсно використовуєте це підключення таким самим чином:



Запустимо команду:

docker-compose build

вона запускає процес **збирання образів** для всіх сервісів, які описані у файлі docker-compose.yml.

Детальний розбір:

- **docker-compose** — інструмент, який дозволяє керувати багатоконтейнерними застосунками, описаними у docker-compose.yml.
- **build** — інструкція, яка каже Docker зібрати образи для сервісів, де в конфігурації вказано build: . або інший шлях до Dockerfile.

Що відбувається:

1. Docker читає docker-compose.yml.

2. Для кожного сервісу, де є build:, він виконує збірку образу на основі відповідного Dockerfile.
3. Якщо образ уже існує і не змінився контекст (файли, залежності), Docker використає кешовані шари.
4. Після завершення у тебе будуть готові локальні образи, які можна запускати через docker-compose up.

Далі введемо команду:

docker-compose up -d mongo

docker-compose up використовується для **запуску всіх сервісів**, описаних у файлі docker-compose.yml. А команда **docker-compose up -d mongo** запустить лише один сервіс — mongo з твого docker-compose.yml.
Що відбувається:

1. Docker Compose читає конфігурацію з docker-compose.yml.
2. Якщо для сервісів потрібна збірка (build:), він перевіряє, чи є готові образи. Якщо немає — виконує docker-compose build.
3. Створює та запускає контейнери для кожного сервісу (наприклад, app і mongo).
4. Автоматично створює спільну мережу, щоб контейнери могли взаємодіяти між собою за іменами сервісів.
5. Проброс портів і монтування томів налаштовуються так, як описано у файлі.

Варіанти використання:

- **docker-compose up** — запускає сервіси у поточному терміналі, показуючи їхні логи.
- **docker-compose up -d** — запускає сервіси у фоновому режимі (detached mode), щоб термінал залишався вільним.

Отже, якщо **docker-compose build готує образи**, то **docker-compose up запускає контейнери і піднімає всю систему в робочий стан**.

Далі введемо команду:

docker-compose up -d app

Якщо у вас порт вже зайнятий то треба зупинити контейнер

docker ps

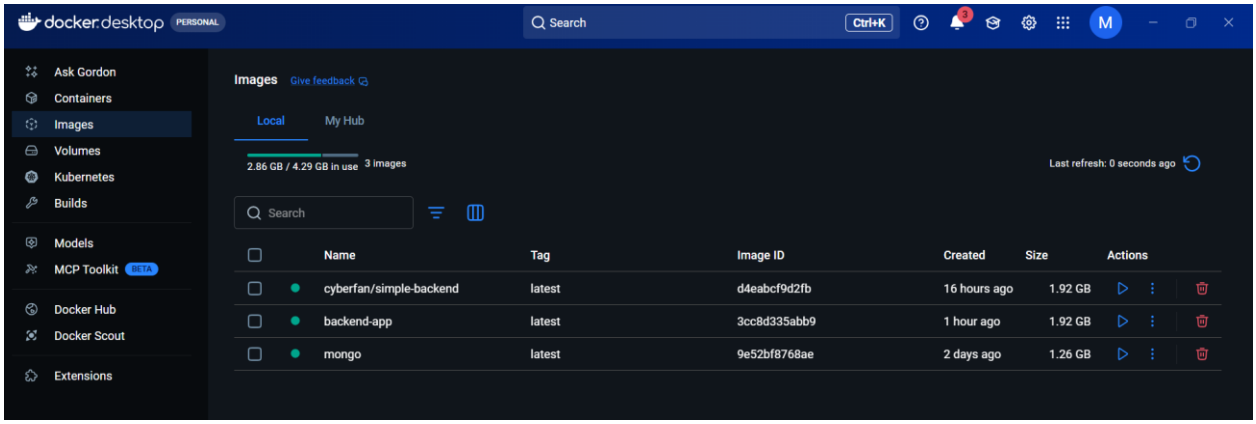
docker stop <container_id>

Командой docker ps ми можемо перевірити які контейнери зараз запушені

Команда яка дозволяє переглянути лог файли контейнера:

docker logs <container_id>

Також в docker можна глянути які наразі запушені образи



Для того щоб зупинити всі контейнери команда:

docker-copmose stop