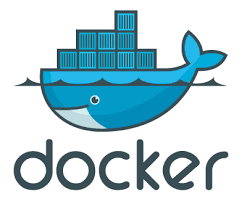
Лабораторна робота №5



Використання Docker, для розгортання додатку Node.JS

Docker являє собою систему управління контейнерами. Вона дозволяє «упакувати» додаток або веб-сайт з усім його оточенням і залежностями в контейнер, яким в подальшому можна легко і просто керувати: переносити на інший сервер, масштабувати, оновлювати.

Docker був написаний на мові програмування Go і випущений в 2013 році. Спочатку він працював тільки з Linux-системами, проте на даний момент його можна використовувати також в Windows і macOS. Незважаючи на те, що проект є відносно новим, Докер широко використовується багатьма фахівцями і продовжує завойовувати популярність.

Контейнери (containers) являють собою засоби інкапсуляції додатку разом з його залежностями. На перший погляд контейнери можуть здаватися усього лише спрощеною формою віртуальних машин (virtual machines - VM) - як і віртуальна машина, контейнер містить ізольований екземпляр операціонної системи (ОС), який можна використовувати для запуску додатків. Але контейнери мають деякі переваги, що забезпечують такі варіанти використання, які важко або неможливо реалізувати в звичайних віртуальних машинах:

* контейнери спільно використовують ресурси основної ОС, що робить їх напорядок більш ефективними. Контейнери можна запускати і зупиняти за долі секунди. Для додатків, що запускаються в контейнерах, накладні витрати мінімальні або взагалі відсутні, в порівнянні з додатками, що запускаються безпосередньо під управлінням основної ОС;
* переносимість контейнерів забезпечує потенційну можливість усунення цілого класу програмних помилок, що викликаються незначнимними змінами робочого середовища, це позбавляє обґрунтування позиції розробника: «*але це працює на моєму комп'ютері*»;
* спрощена сутність контейнера означає, що розробники можуть одночасно запускати десятки контейнерів, що дає можливість імітації роботи промислової розподіленої системи. Інженери по експлуатації можуть запустити на одному хості набагато більше контейнерів, ніж при використанні окремих віртуальних машин;
* окрім того, контейнери надають переваги кінцевим користувачам і розробникам без необхідності розгортання програми в хмарі. Користувачі можуть завантажувати і запускати складні додатки без багатогодинної метушні з конфігурацією і проблемами при установціі, при цьому не турбуватися про будь-які зміни в їх локальних системах. У свою чергу, розробники подібних додатків можуть уникнути проблем, пов'язаних з відмінностями в конфігураціях призначених для користувача середовищ і з доступністю залежностей для цих додатків.

І що більш важливо, існують принципові відмінності у меті використання віртуальних машин і контейнерів.

Метою застосування ***віртуальної машини*** є повна емуляція чужорідного програмного (операційного) середовища, тоді як мета застосування ***контейнера*** - зробити додатки такими, що переносяться і самодостатніми.

Розглянемо три основні терміни в екосистемі Docker:

**Образ (image)** - шаблон, який використовується для створення контейнерів. Являє собою зліпок файлової системи, в якому розташований код програми та його оточення.

**Реєстр (registry)** - репозиторій образів. Docker Hub, - це публічний репозиторій, де зберігається величезна кількість образів.

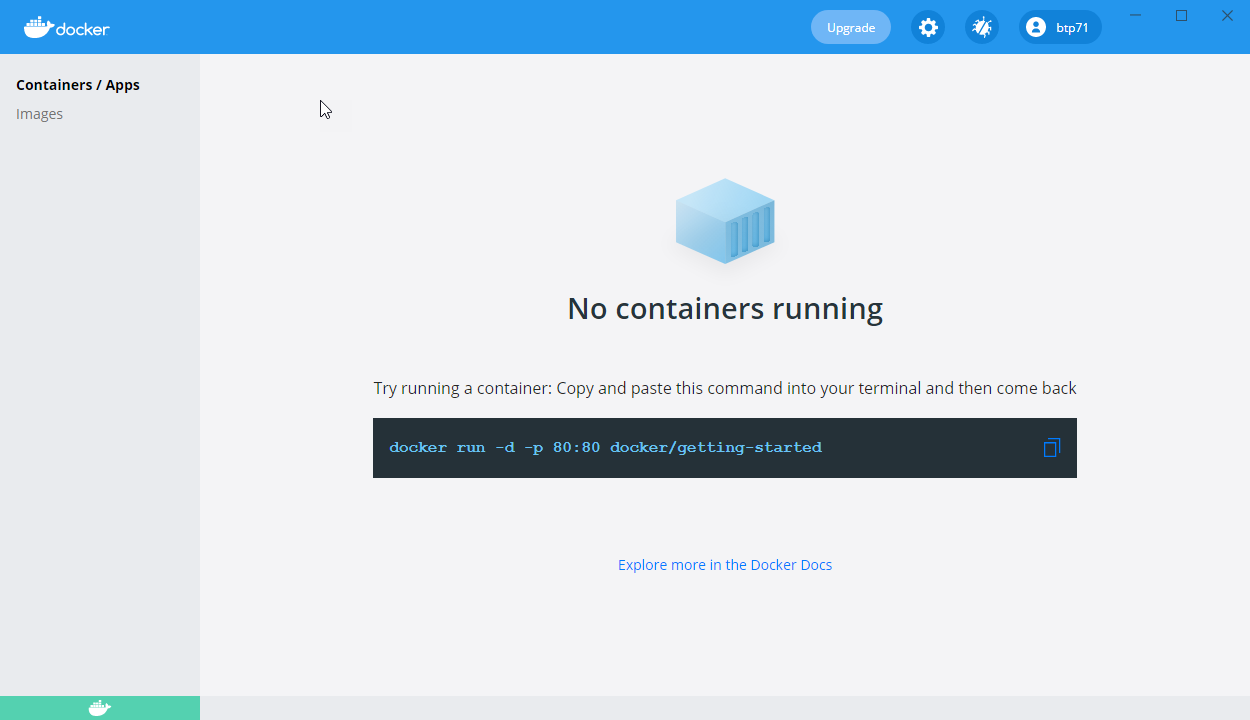
**Контейнер (container)** - запущений додаток, тобто сукупність процесів і образу.

**Встановлення Docker**

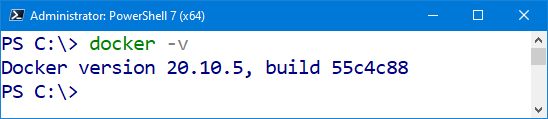
Детальні керівництва по встановленню Docker на різні ОС знаходяться на офіційному сайті [https://www.docker.com](https://www.docker.com/)

Ми будемо використовувати Docker під операційною системою Windows.

Для цього нам знадобиться програмний пакет **Docker Desktop**, який потрібно завантажити за адресою <https://www.docker.com/products/docker-desktop>. Після встановлення і запуску цього пакету ми маємо змогу повноцінно використовувати Docker у нашій операційній системі.



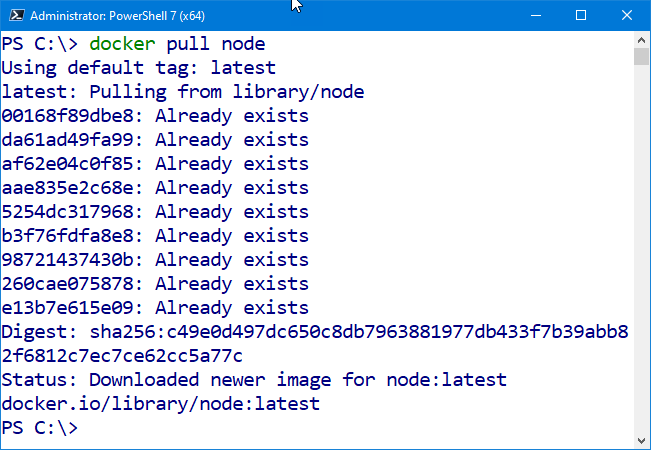
Також можно перевірити версію Docker за допомогою терміналу.



Важливою частиною екосистеми Docker є **Docker Hub** <https://hub.docker.com/> - відкритий репозиторій образів контейнерів. У ньому можна знайти десятки готових додатків від офіційних розробників. Серед них - nginx, MySQL, Apache, Gitlab, Redmine, Elasticsearch, Jenkins і інші.

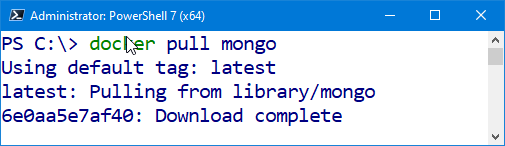
Для того, щоб завантажити контейнер з Docker Hub скористуємося командою:

docker pull node



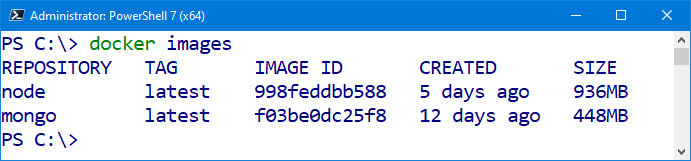
Таким чином ми завантажили образ з вже встановленною Node.JS. При завантаженні ми не вказали версію Node.JS, тому за замовченням був завантажений образ з останньою стабільною версією **node:latest**. Після двокрапки маємо змогу вказувати конкретну потрібну версію.

Аналогічно завантажимо образ з MongoDB.

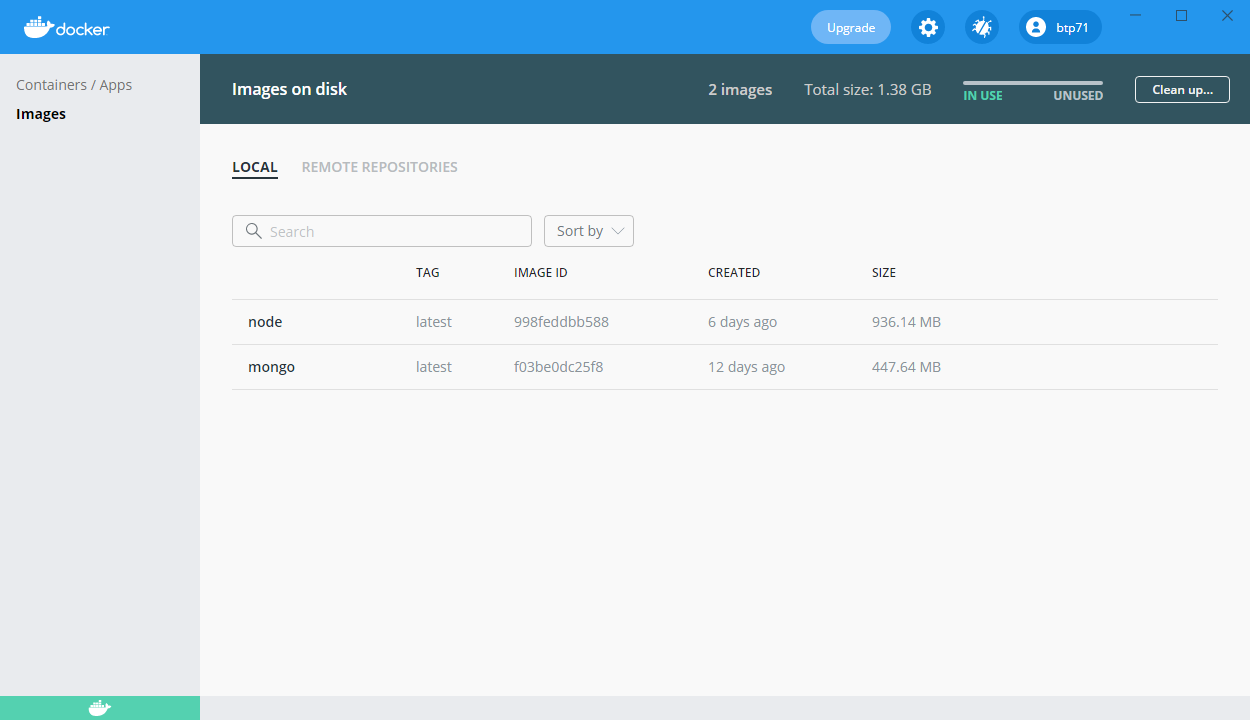


Для перегляду образів що завантажені скористуємося командою:

docker images



Відповідно бачимо список завантажених образів. Аналогічний список образів відображується у панелі Docker Desktop на вкладці Images.



Керувати контейнерами та образами можливо як за допомогою візуального інтерфейсу Docker Desktop, так і запускаючи відповідні команди у терміналі. В подальшому будемо розглядати тільки варіант з терміналом.

Підказку по командам завжди можливо подивитися запускаючи команду

docker --help

Запустимо контейнер з MongoDB наступною кмандою

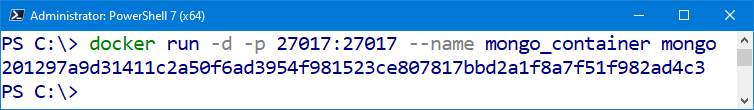
docker run -d -p 27017:27017 --name mongo\_container mongo

-d - запустити контейнер у фоновому режимі та вивести его ID;

-p - прокинути порт в контейнер. (у даному випадку прокидується 27017 порт);

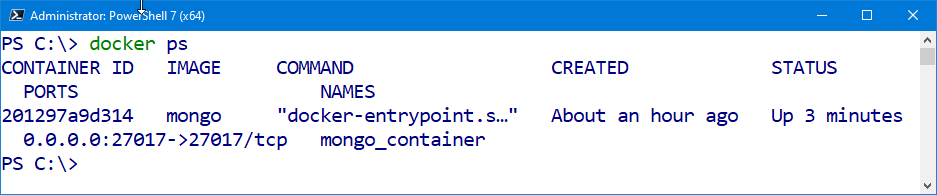
--name - призначити ім’я контейнеру.

mongo - ім’я образу, на основі якого потрібно створити контейнер.



Виведемо список запущених контейнерів:

docker ps



Зупинимо контейнер використовуючи його ID, причому достатньо набрати тільки перші символи CONTAINER ID:

docker stop 201

PS D:\> docker stop 201

201

PS D:\>

Перевіремо список усіх існуючих контейнерів:

docker ps -a

PS D:\> docker ps -a

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

201297a9d314 mongo "docker-entrypoint.s…" 2 hours ago Exited (0) About a minute ago mongo\_container

PS D:\>

Видалимо контейнер командою:

docker rm 201

PS D:\> docker rm 201

201

PS D:\>

Пересвідчимося, що всі контейнери видалено:

PS D:\> docker ps -a

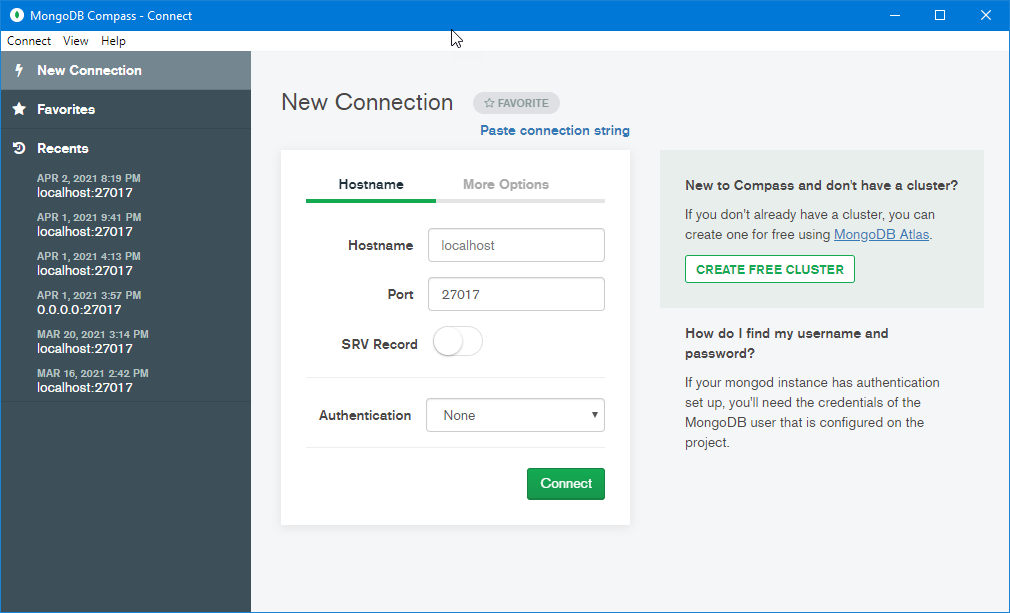
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

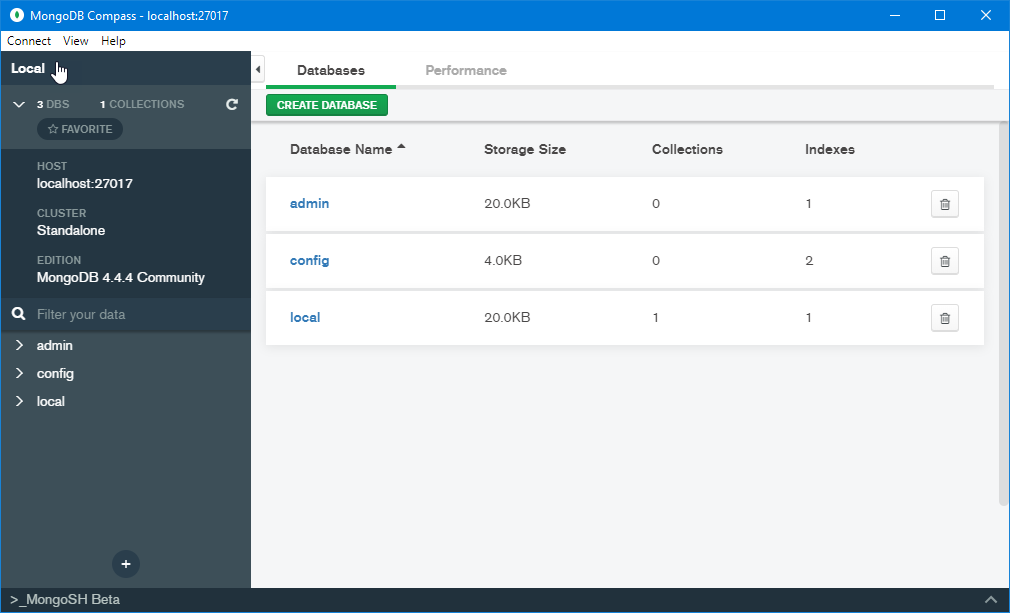
PS D:\>

Знову запустимо контейнер з MongoDB

docker run -d -p 27017:27017 --name mongo\_container mongo

Пересвідчитися, що контейнер працює можливо за допомогою MongoDBCompass. Запустивши MongoDBCompass підключаємося до MongoDB за адресою <http://localhost:27017>

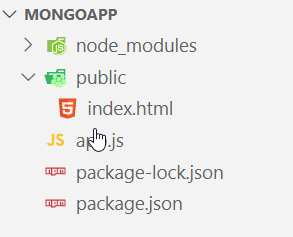




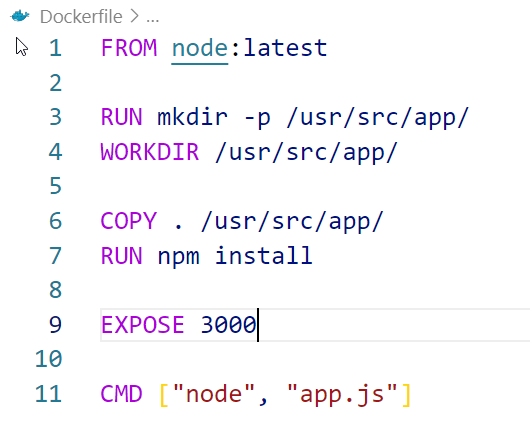
Висновок: MongoDB запущена у контейнері і ми маємо змогу підключатися до неї з нашого додатку з попередніх лабораторних робот.

**Створення власного образу. Dockerfile**

Для створення власного образу скористуємося проектом з попередньої роботи. Зараз структура нашого проекту виглядає наступним чином:



Створювати новий власний образ мибудемо за допомогою конфігураційного файлу, який має назву **Dockerfile.** Отож у корені проекту створимо такий файл з наступним вмістом:



Таблиця. Пояснення до змісту файлу Dockerfile

|  |  |
| --- | --- |
| FROM node:latest | Вказуємо базовий образ (на основі якого створюємо новий). |
| RUN mkdir -p /usr/src/app/ | В операційній системі нового образу (це **linux**) створюємо відповідну теку. |
| WORKDIR /usr/src/app/ | Призначаємо цю теку робочою для нашого проекту |
| COPY . /usr/src/app/ | Копіюємо файли з поточної теки нашого проекту до ОС образу. |
| RUN npm install | Запускаємо команду для встановлення необхідних модулів, які прописані у файлі **package.json** |
| EXPOSE 3000 | Вказуємо що буде використовуватися порт 3000 |
| CMD ["node", "app.js"] | Після запуску контейнера на основі цього образу виконати таку команду для запуску нашого додатку |

Для запобігання копіювання зайвої інформації до нашого образу у корені нашого проекту створимо файл .dockerignore наступного вмісту:

node\_modules

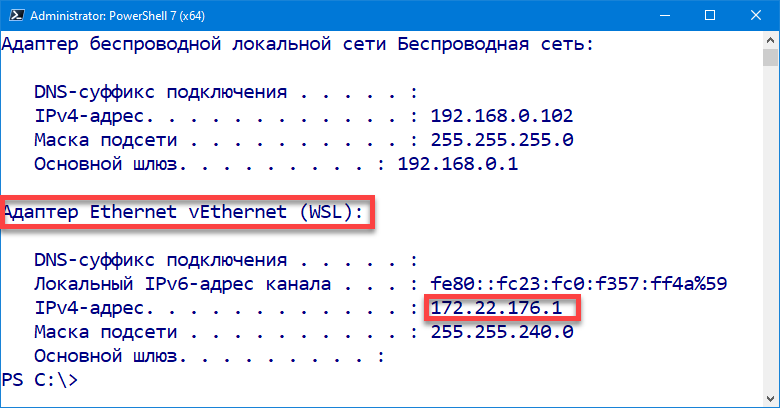
Dockerfile

package-lock.json

Перед створенням власного образу потрібно з’ясувати на якому IP працює Docker. Запустимо в терміналі команду:

ipconfig

Результат буде приблизно таким:



Цю ір адресу необхідно вказати у конфігураційному файлі MongoDB, який знаходиться за шляхом c:\Program Files\MongoDB\Server\4.4\bin\, якщо ви не встановили MongoDB у іншу теку. Відредагуємо файл mongod.cfg. Знайдіть параметр:

# network interfaces

net:

port: 27017

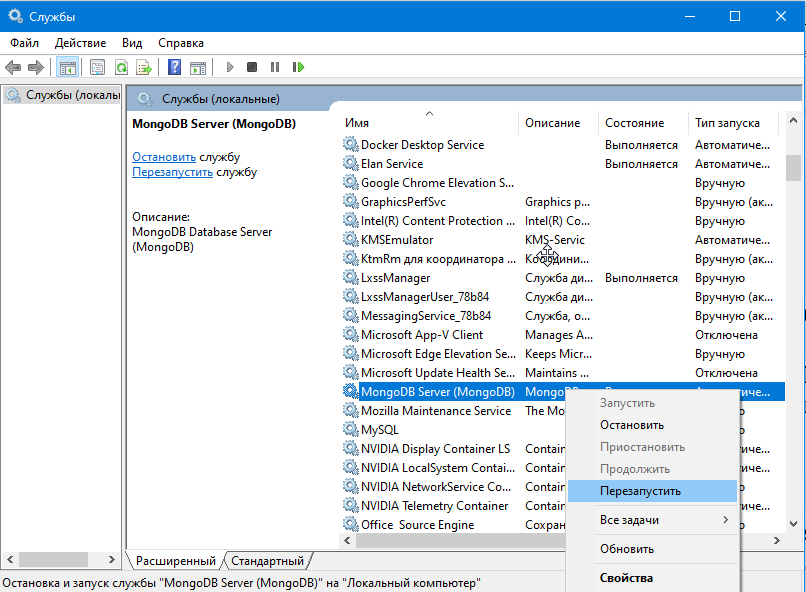
bindIp: 127.0.0.1

Замінемо адресу bindIp на Ір Docker (*на Вашому хості це може буде інша адреса*)

bindIp: 172.22.176.1

Перезавантажемо службу MongoDB:

*Панель управления->Администрирование->Службы*



Відредагуємо файл app.js нашого додатку, зокрема рядок підключення до Mongo DB:

mongoose.connect("mongodb://172.22.176.1:27017/usersdb", { useUnifiedTopology: true, useNewUrlParser: true, useFindAndModify: false }, function(err){

    if(err) return console.log(err);

    app.listen(3000, function(){

        console.log("Сервер ожидает подключения...");

    });

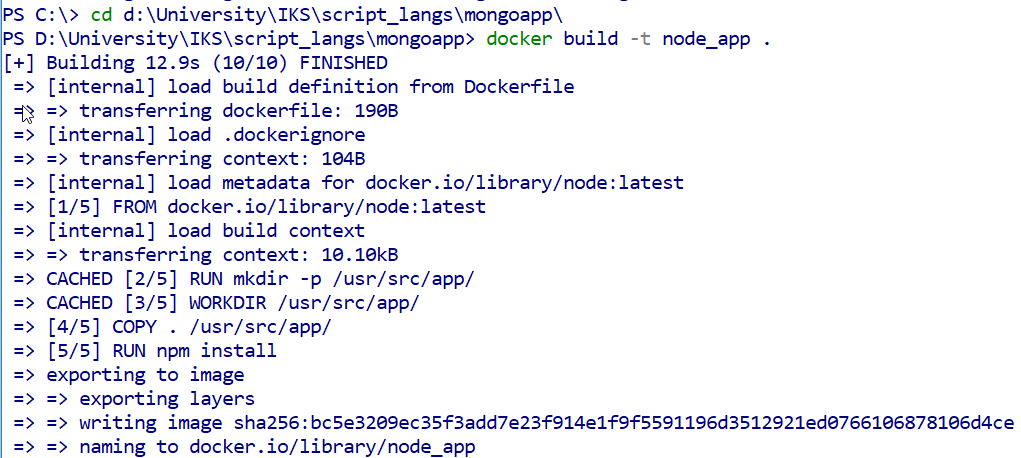
});

Створимо новий образ командою (запускати потрібно з кореневого каталогу нашого додатку):

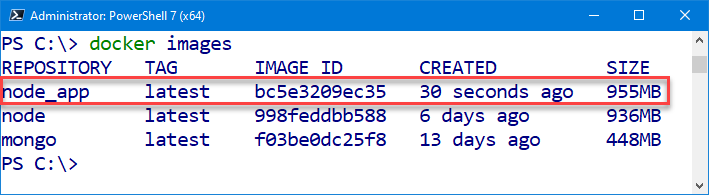
docker build -t node\_app .

-t node\_app – вказує назву нашого образу;

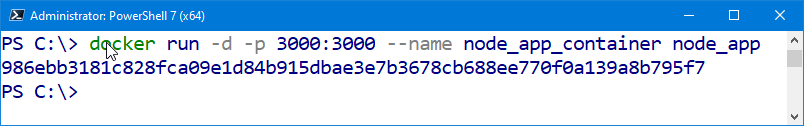
. – крапка вказує на теку, де знаходиться Dockerfile



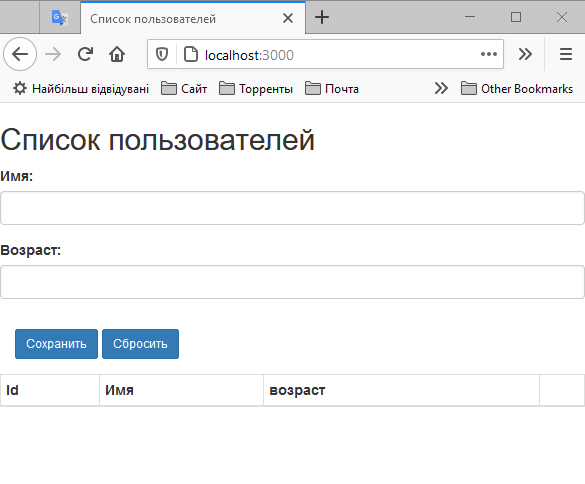
Пересвідчимося що маємо новий образ:



Запустимо новий контейнер на базі нашого нового образу:

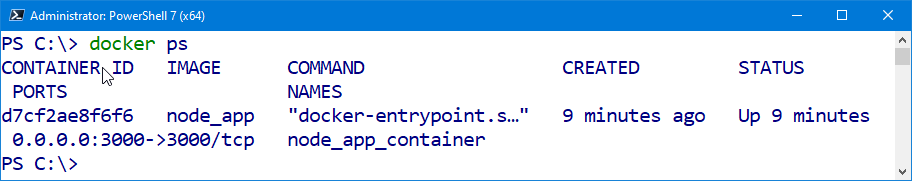


В браузері за адресою <http://localhost:3000> запустимо наш додаток.



Тобто наш додаток запущений в контейнері має змогу взаємодіяти з MongoDB, яка запущена на локальному хості.

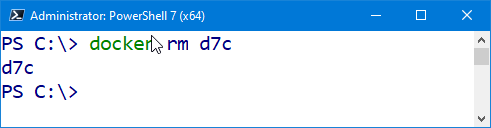
Переглянемо список запущених контейнерів



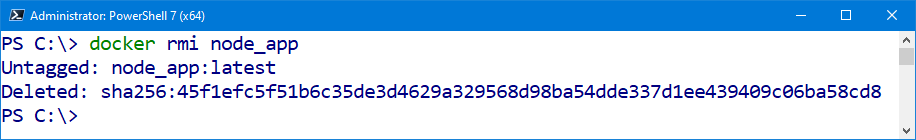
Зупинемо наш контейнер використовуючи його ID (*можна використати тільки перші символи*):



І видалимо його:



Також видалимо наш образ:



Відновимо конфігурацію MongoDB (редагуємо файл mongod.cfg).

# network interfaces

net:

port: 27017

bindIp: 127.0.0.1

І перезавантажимо службу MongoDB.

**Взаємодія контейнерів. Утіліта docker-compose**

Для спрощення роботи можна додати ще одну невелику функцію автоматизації. Інструмент **Docker Compose** (<http://docs.docker.com/compose/>) призначений для швидкого налаштування і запуску різних варіантів середовищ розробки Docker. Важливо відзначити, що Compose використовує файли YAML (YAML Ain’t Markup Language; мова сериалізації даних) для зберігання конфігурації груп контейнерів, дозволяючи розробникам позбутися від рутинної ручної роботи і не витрачати сил на дублювання вже існуючого рішення. На даний момент гранична простота нашого додатки не дозволяє по справжньому оцінити переваги автоматизації, але додатки зазвичай дуже швидко стають великими і складними. **Compose** звільнить нас від необхідності супроводжувати всі допоміжні скрипти, призначені для організації роботи, включаючи запуск, встановлення з'єднань, оновлення і зупинку контейнерів.

Якщо Docker встановлений за допомогою Docker Desktop, то Compose вже доступний у вашій ОС. В іншому випадку виконайте інструкції по установці Compose з веб-сайту Docker (<http://docs.docker.com/compose/install/>).

Наше завдання створити два контейнери. Один з сервером баз даних MongoDB, другий наш додаток на Node.JS, та налаштувати їхню взаємодію.

У корені нашого додатку створимо файл docker-compose.yaml наступного змісту:

version: "3"

volumes:

    mongo\_db\_data:

services:

    my\_mongodb:

        image: mongo:latest

        volumes:

            - mongo\_db\_data:/data/db



app:

        build: .

        ports:

            - 3000:3000

        environment:

            - MONGO\_DB\_HOSTNAME=my\_mongodb

            - MONGO\_DB\_PORT=27017

            - MONGO\_DB=usersdb

Таблиця. Поясненя до змісту файлу docker-compose.yaml

|  |  |
| --- | --- |
| version: "3" | Версія формату файлу docker-compose.yaml |
| services: | У цій секції визначається список служб (або контейнерів), які потрібно використовувати як частину нашого додатку. |
| app: | Секція, яка визначає директиви для контейнеру нашого додатку |
| build: . | Ключ build повідомляє Compose, що образ для даного контейнера повинен бути створений з Dockerfile, розташованого в поточному каталозі (.). Кожне визначення контейнеру повинно містити або ключ build, або ключ image. Для ключів image задається тег або ідентифікатор способу, з якого створюється контейнер, по аналогії з аргументом image в команді docker run. |
| ports:       - "3000:3000" | Ключ ports повністю аналогічний аргументу -p в команді docker run і пизначений для оголошення відкритих портів. Тут зв'язується порт 3000 в контейнері з портом 3000 на хості. Лапки не обов'язкові, але рекомендується їх використання, щоб уникнути плутанини при синтаксичному розборі YAML-виразів, наприклад 56:56 як числа в шестидесятковій системі. |
| environment:   - MONGO\_DB\_HOSTNAME=my\_mongodb   - MONGO\_DB\_PORT=27017   - MONGO\_DB=usersdb | Ключ environment повністю аналогічний аргументу -e в команді docker run і призначений для визначення значень змінних середовища в контейнері. |
| my\_mongodb: | Секція, яка визначає контейнер для роботи з MongoDB |
| image: mongo:latest | Образ на базі якого створюється контейнер |
| volumes:      - mongo\_db\_data:/data/db | Ключ volumes повністю аналогічний аргументу -v в команді docker run і призначений для визначення томів збереження даних. Тут визначено монтування volumes з назвою mongo\_db\_data до текі /data/db всередині контейнеру |
| volumes:      mongo\_db\_data: | Зазначення з яким volumes ми працюємо |

Також додаємо файл docker-compose.yaml у файл .dockerignore

node\_modules

docker-compose.yaml

Dockerfile

package-lock.json

У файлі package.json визначемо пакету dotenv для використання змінних середовища.

{

    "name": "mongoapp",

    "version": "1.0.0",

    "dependencies": {

        "dotenv": "^8.2.0",

        "express": "^4.17.1",

        "mongodb": "^3.6.4",

        "mongoose": "^5.12.1"

    }

}

Змінемо зміст файлу app.js наступним чином:

const mongoose = require("mongoose");

const express = require("express");

const Schema = mongoose.Schema;

const app = express();

const jsonParser = express.json();

const {

    MONGO\_DB\_HOSTNAME,

    MONGO\_DB\_PORT,

    MONGO\_DB

} = process.env

const options = {

    useUnifiedTopology: true,

    useNewUrlParser: true,

    useFindAndModify: false

}

const url = `mongodb://${MONGO\_DB\_HOSTNAME}:${MONGO\_DB\_PORT}/${MONGO\_DB}`;

const userScheme = new Schema({name: String, age: Number}, {versionKey: false});

const User = mongoose.model("User", userScheme);

app.use(express.static(\_\_dirname + "/public"));

mongoose.connect(url, options, function(err){

    if(err) return console.log(err);

    app.listen(3000, function(){

        console.log("Сервер ожидает подключения...");

    });

});

app.get("/api/users", function(req, res){

    User.find({}, function(err, users){

        if(err) return console.log(err);

        res.send(users)

    });

});

app.get("/api/users/:id", function(req, res){

    const id = req.params.id;

    User.findOne({\_id: id}, function(err, user){

        if(err) return console.log(err);

        res.send(user);

    });

});

app.post("/api/users", jsonParser, function (req, res) {

    if(!req.body) return res.sendStatus(400);

    const userName = req.body.name;

    const userAge = req.body.age;

    const user = new User({name: userName, age: userAge});

    user.save(function(err){

        if(err) return console.log(err);

        res.send(user);

    });

});

app.delete("/api/users/:id", function(req, res){

    const id = req.params.id;

    User.findByIdAndDelete(id, function(err, user){

        if(err) return console.log(err);

        res.send(user);

    });

});

app.put("/api/users", jsonParser, function(req, res){

    if(!req.body) return res.sendStatus(400);

    const id = req.body.id;

    const userName = req.body.name;

    const userAge = req.body.age;

    const newUser = {age: userAge, name: userName};

    User.findOneAndUpdate({\_id: id}, newUser, {new: true}, function(err, user){

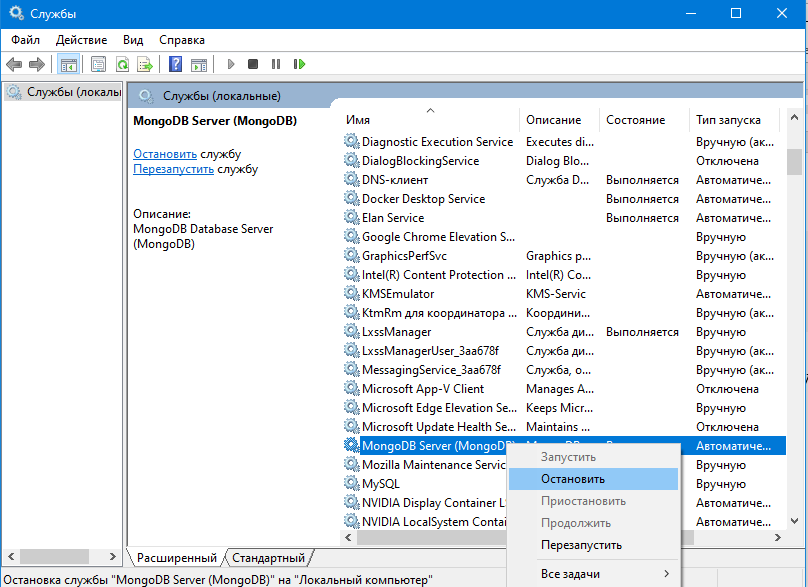
        if(err) return console.log(err);

        res.send(user);

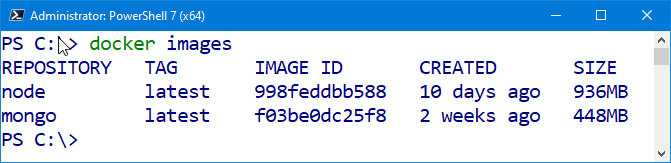
    });

});

Зупинемо службу серверу MongoDB у ОС.



Видаліть усі існуючи контейнери, якщо вони є. Також потрібно видалити образ нашого додатку, який був створений у попредній частині роботи. Тобто у нас повинні залишитися два образи яки ми завантажили з DockerHub:



У принципі, навіть ці образи не потрібні на локальному хості. Якщо їх не буде, вони автоматично завантажуються з DockerHub.

З поточного каталогу нашого додатку виконаємо команду:

docker-compose up -d

У бразузері запустимо наш додаток <http://localhost:3000/>

Таким чином ми отримали повністю ізольований додаток, який створений на базі двох контейнерів.

**Посилання**

<https://www.docker.com/> Сайт Docker

<https://docs.docker.com/> Документація Docker