# Docker Compose

### What is Docker Compose?

## Overview

Compose is a tool for Docker that allows you to define and run multiple Docker containers with a single command. It does this through a single configuration file where we can specify the deployment completely.

It is an excellent tool for defining and creating collections of containers and is particularly handy for spinning up microservice applications

## Objective

* Understand what Docker Compose is
* Why Docker Compose is used

## Configuring Microservice Applications

Consider a 4-part microservice application that consists of a reverse proxy, a frontend, a backend and a persistent database, each in their own container.

In order for this application to work it needs:

* Four containers, each with their own image and environment variables set
* A volume to persist the data in the database
* A bind mount to overwrite the **nginx.conf** file, allowing us to define its behaviour as a reverse proxy
* Relevant ports published
* A network to allow our services to communicate with one another

This would be a relatively large task considering it is a simple microservice application. This is where **Docker Compose** comes in. It allows us to streamline this process through the use of configuration files.

Docker Compose makes use of configuration files that allow us to declare what Docker resources we want rather than manually creating them.  
It does this using YAML files called **docker-compose.yaml** or   
**docker-compose.yml**

Once you have written the file, you simply need to enter the command **docker-compose up -d** and all your containers will be created and connected to a bridge network for you with the exact configuration you specified.

### Let’s get started!

In this part of the lab, you will create a simple Docker Compose configuration which creates an NGINX container and scales the application by making multiple replicas of it.

### Docker Compose Configuration File

1. **mkdir** docker-compose-nginx-**tutorial**
2. **cd** docker-compose-nginx-**tutorial**
3. Create a file called **docker-compose.yaml**

version: "3.8"

services:

nginx:

image: nginx:alpine

ports:

- target: 80

protocol: tcp

### Run Your Configuration

Type: docker-compose up -d

### View the Running Containers Using Compose

Type the following command and view the result: **docker-compose ps**

Connect to the container in a browser using the port number that you see.

### Scale Your Application

Compose will allow you to scale your application depending on the machine that you are using and the resources that are used by each of the containers.

In this instance we should be fine scaling to three NGINX containers:

**docker-compose up -d --scale nginx=3**

View the running containers there should be 3 instances of NGINX:

docker-compose ps

Name Command State Ports

-------------------------------------------------------------------

test\_nginx\_1 nginx -g daemon off; Up 0.0.0.0:32768->80/tcp

test\_nginx\_2 nginx -g daemon off; Up 0.0.0.0:32769->80/tcp

test\_nginx\_3 nginx -g daemon off; Up 0.0.0.0:32770->80/tc

#### **Clean Up!**

We can now stop and remove all the containers and images used by running this command:

**docker-compose down *--rmi all***

## Part 2 – A real-life example of using docker-compose

Let’s try another practical application which displays a static web page that uses a node server to get data from a MongoDb database.

In this example, you’ll install Node + MySQL and MongoDb databases using one configuration file.

1. Create a directory for this new application
2. Start a Terminal session in the new directory
3. Type the following command to create the necessary file: **docker init**

A screen shot of a computer program

Description automatically generated

1. Press Enter to accept the general purpose starting point. This option will create the following files:  
   A black background with white text

   Description automatically generated
2. You will soon fill these with code and configuration settings.
3. Create a directory called **public** to place your default **index.html** file.   
   We’ll come back to this file a bit later.
4. Copy the following text to the Dockerfile

# Use the official Node.js image

FROM node:14

# Create and change to the app directory

WORKDIR /usr/src/app

# Copy package.json and package-lock.json

COPY package\*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application

COPY . .

# Expose the port the app runs on

EXPOSE 3000

# Define the command to run the app

CMD ["node", "index.js"]

1. Copy the following text to your package.json

{

  "name": "static-web-app",

  "version": "1.0.0",

  "description": "An static web app using MongoDB and Docker",

  "main": "index.js",

  "scripts": {

    "start": "node index.js",

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "author": "Mike B",

  "license": "QA",

  "dependencies": {

    "express": "^4.17.1",

    "mongoose": "^6.0.12",

    "body-parser": "^1.19.0"

  }

}

1. The above file starts a node server by calling **node index.js** and sets up all the dependencies.
2. Copy the following text to the **docker-compose.yaml** file

services:

  web:

    build: .

    ports:

      - "3000:3000"

    volumes:

      - .:/usr/src/app

      - /usr/src/app/node\_modules

    depends\_on:

      - mongo

  mongo:

    image: mongo:latest

    ports:

      - "27017:27017"

    volumes:

      - mongo-data:/data/db

volumes:

  mongo-data:

The **volumes** setting is added because this component stores and manages persistent data generated by your containers, ensuring that data remains intact even after containers are removed or updated. This persistence accelerates development by allowing frequent updates during routine operations without data loss.

1. Finally add the code below to the index.js file for the node part of the application.  
   The code creates a connection to a MongoDb database on port 27017.

Let’s put it all together!

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const app = express();

const mongoURL = process.env.MONGO\_URL || 'mongodb://mongo:27017/testdb';

mongoose.connect(mongoURL, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log('Connected to MongoDB'))

  .catch(err => console.error('Could not connect to MongoDB', err));

const itemSchema = new mongoose.Schema({

  name: String,

  message: String,

});

const Item = mongoose.model('Item', itemSchema);

app.use(bodyParser.json());

app.use(express.static('public'));

app.post('/api/items', async (req, res) => {

  const item = new Item({ name: req.body.name, message:req.body.message });

  await item.save();

  res.status(201).send(item);

});

app.get('/api/items', async (req, res) => {

  const items = await Item.find();

  res.send(items);

});

const port = process.env.PORT || 3000;

app.listen(port, () => {

  console.log(`Server is running on port ${port}`);

});

The above code sets up a schema (view the **const itemSchema** setting) to define what it stores in a database called **testdb** which stores **name** and **message** strings.   
  
The database stores its data in a collection called “Item”

const Item = mongoose.model('Item', itemSchema);

The code returns all the data (names and messages) using the following Api

const Item = mongoose.model('Item', itemSchema);

Data is saved using a post call

app.post('/api/items', async (req, res) => {

  const item = new Item({ name: req.body.name, message:req.body.message });

  await item.save();

1. You will make calls to load and save data using the file **index.html** in the **public** folder  
    See the code below which will creates the following form  
   A screenshot of a computer

   Description automatically generated
2. Copy the following code to **index.html** in the **public** folder

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Updated Static Web App with MongoDB</title>

  <script>

    async function fetchItems() {

      const response = await fetch('/api/items');

      const items = await response.json();

      const itemsList = document.getElementById('items-list');

      itemsList.innerHTML = '';

      items.forEach(item => {

        const listItem = document.createElement('li');

        listItem.textContent = `${item.name}:${item.message}`;

        itemsList.appendChild(listItem);

      });

    }

    async function addItem(event) {

      event.preventDefault();

      const itemName = document.getElementById('item-name').value;

      const itemMessage = document.getElementById('item-message').value;

      let data = JSON.stringify({name: itemName,message:itemMessage});

      if (itemName) {

        await fetch('/api/items', {

          method: 'POST',

          headers: {

            'Content-Type': 'application/json',

          },

          body: data

        });

        document.getElementById('item-name').value = '';

        fetchItems();

      }

    }

    window.onload = function () {

      fetchItems();

      document.getElementById('item-form').onsubmit = addItem;

    }

  </script>

</head>

<body>

  <h1>Updated Static Web App with MongoDB</h1>

  <form id="item-form">

    Name:<input type="text" id="item-name" placeholder="Enter item name" required><br />

    Message:<input type="text" id="item-message" placeholder="Enter a message" required><br />

    <button type="submit">Add Item</button>

  </form>

  <h2>Items List</h2>

  <ul id="items-list"></ul>

</body>

1. </html>
2. Start the server by typing: **docker-compose up**  
   You may have to wait a few minutes for the first time.
3. Navigate to <http://localhost:3000/> in a browser to test the app.
4. Create and save a few records (name and message) using the form in the HTML page.
5. You can now examine the database (see the next page) and run few NoSQL commands

Let’s examine the MongoDb database   
  
Run the following command to get to know the MongoDb database you created earlier.  
Type docker ps to get the ID of the MongoDB and then stat a shell   
C:\>**docker ps** (to get the id of your mongo database. See example below)

CONTAINER ID IMAGE COMMAND CREATED STATUS …  
**7409b4291ca2 mongo:latest** …..

C:\>**docker exec -it 740 mongosh**

When in the shell, type the following basic command:

**show** dbs

**use** testdb

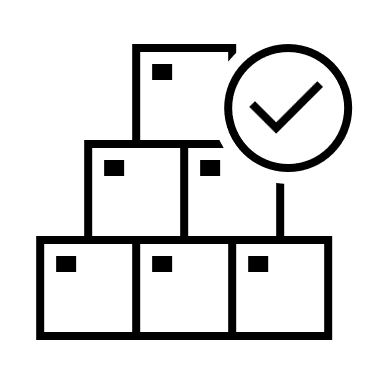
**show collections**

Toview all the records type: **db.**item.**find**()

To delete all the records type: **db.**item**.deleteMany({})**

To manually insert a few records type:  
**db**.item.**insertMany**([{name: "Max", message:"Hi"}, {name: "Sasha", message:"Hello"}])  
  
To update a record type:   
**db**.item.**updateOne**({name: "Sasha"},{$set:{message: "Hello again!"}})

Type **exit** to exit this shell and return to your Terminal window.

Congratulations, you have successfully created a Docker to host a Node application that persists data in a MongoDB database.

# Time to experiment!

# Scientist male outlineConsider developing an application that collects data from a customer, such as items in a shopping basket, and stores this information in a MongoDB database using an appropriate schema.