Aplicação de contêineres Docker no desenvolvimento NodeJS utilizando docker-compose e volumes

Tiago B. Boeing[1]

[1] Universidade do Sul de Santa Catarina, Tubarão, Brasil

contato@tiagoboeing.com

Abstract. *This meta-article is a tutorial to based development in Docker containers using NodeJS programming language. This approach has benefits if compared with other "traditional" and is a good practice to guarantee sceneries homogeneous, production, test, or development.*

Resumo. *Este meta-artigo é um guia para o desenvolvimento baseado em contêineres Docker e da utilização da linguagem de programação NodeJS. Esta abordagem possui alguns benefícios se comparada ao "tradicional" e é uma boa prática por garantir cenários homogêneos, tanto de produção quanto testes ou desenvolvimento.*

1. Introdução

No desenvolvimento de software independente da linguagem de programação utilizada é comum que o fluxo de trabalho para *deploy* em diferentes ambientes seja relativamente similar. Muitas das dificuldades foram mapeadas pela comunidade de desenvolvimento ao longo dos anos, uma delas relacionada ao gerenciamento e reaproveitamento de recursos através da virtualização. O *Docker* surgiu em 13 de março de 2013 com o propósito de facilitar a criação, implantação e execução de aplicativos através da utilização de *containers*.

> "Os contêineres permitem que um desenvolvedor empacote um aplicativo com todas as partes necessárias, como bibliotecas e outras dependências, e implante-o como um pacote. Ao fazer isso, graças ao contêiner, o desenvolvedor pode ter certeza de que o aplicativo será executado em qualquer outra máquina Linux, independentemente das configurações personalizadas que a máquina possa ter e que possam diferir da máquina usada para escrever e testar o código." ["What is Docker?" 2020]

2. Contextualização

NodeJS é amplamente utilizado atualmente para desenvolvimento. Embora o fluxo de trabalho não seja muito complexo, aliado ao *Docker* e boas práticas é possível garantir uma aplicação previamente pronta para ambiente de produção (real). Tal cenário não se restringe apenas ao *NodeJS*.

> O Node.js® é um tempo de execução JavaScript criado no mecanismo JavaScript V8 do Chrome. O Node.js usa um modelo de E/S sem bloqueio, orientado a eventos, que o torna leve e eficiente. O ecossistema de pacotes do Node.js., npm, é o maior ecossistema de bibliotecas de código aberto do mundo. ["What exactly is Node.js?" 2020]

3. **Desenvolvimento**

Os pré-requisitos para as etapas seguintes basicamente são: a instalação prévia do *Docker* e *NodeJS.*

Inicie um projeto NodeJS criando uma pasta no diretório de arquivos e rodando o comando **npm init -y** . O -y é utilizado para que informações adicionais como licença, main script, version sejam solicitados.

```
tiagoboeing@TIAGOBOEING MINGW64 ~/Desktop/new-project
$ npm init -y
Wrote to C:\Users\tiagoboeing\Desktop\new-project\package.json:

{
  "name": "new-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Após a execução o arquivo **package.json** foi criado com as informações do mesmo.

Por padrão o comportamento do *Node* é de executar as instruções e finalizar a tarefa, o servidor HTTP garantirá que o mesmo permaneça rodando. Vamos instalar o *Express* para criar um. No terminal rode **npm install express --save**

Neste artigo vamos criar um **index.js** que conterá os seguintes comandos para fazer com que o *Express* funcione:

index.js > ...

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (req, res) {
5    res.send('Hello World!');
6  });
7
8  app.listen(3000, function () {
9    console.log('Example app listening on port 3000!');
10 });
```

Resta agora adicionar a execução na tag *scripts* do **package.json** . Adicione: **"start": "node index.js"** .

O **package.json** final ficará dessa forma:

package.json > ...

```
{
  "name": "new-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Rode no terminal **npm start** e perceba que o servidor estará disponível no endereço local: **http://localhost:3000** . Ao acessar visualizaremos o texto “Hello World”.

3.1. Dockerfile

Na raiz do projeto crie um arquivo **Dockerfile** e adicione o seguinte conteúdo:

```
FROM node:alpine

# cria diretório de trabalho
RUN mkdir /usr/app
WORKDIR /usr/app

COPY package*.json ./

# baixa dependências
RUN npm install

# copia todo conteúdo para pasta de trabalho
COPY . .
```

O Dockerfile é um arquivo que o *Docker* pesquisará no diretório para obter instruções quanto às operações necessárias para que tal container/image seja criado (a). A image *node:alpine* é utilizada visando diminuir o tamanho a ser baixado pelo Docker.

3.2. docker-compose

O arquivo docker-compose representa uma *stack* de serviços que compõem a aplicação. Em outras palavras a *stack* pode ser resumida em um conjunto de outros contêineres que são fundamentais para que a aplicação principal se torne operacional.

Um bom exemplo de uma *stack* é onde uma aplicação necessita de uma base de dados para que consiga se conectar. Através do *docker-compose* podemos configurar o contêiner irmão por assim dizer para que a base de dados seja executada no exato momento que a aplicação principal. Embora não seja exclusividade dessa técnica, a mesma deixa transparente através de um único arquivo e permite tirar proveito da rede interna do Docker para comunicação entre contêineres.

Ainda na raiz da aplicação vamos criar um arquivo **docker-compose.yml** com o seguinte conteúdo:

```yaml
version: "3.4"

services:
  app:
    build: . # indica pasta raiz para buscar Dockerfile
    command: npm start # script no package.json
    ports:
      - "3000:3000" # bind entre porta interna e externa do contêiner
    volumes:
      - .:/usr/app # workdir informado no Dockerfile
```

Neste momento temos a aplicação pronta para produção (ready to deploy).

3.3. Inicializando contêiner

Com o terminal na pasta do projeto, rode **docker-compose up -d** e verifique se o contêiner foi inicializado corretamente. O endereço **http://localhost:3000** deve estar acessível.

```
$ docker-compose up -d
Creating network "auth_default" with the default driver
Building app
Step 1/10 : FROM node:alpine
 ---> 483343d6c5f5
Step 2/10 : RUN apk add g++ make python
 ---> Using cache
 ---> 6186440bc058
Step 3/10 : WORKDIR /
 ---> Running in 17cd05849295
```

Perceba na interface do Docker desktop (Windows) como se encontra o contêiner em execução:

3.4. Atualizar arquivos no contêiner

Embora a aplicação esteja pronta para deploy em produção, ainda não há um fluxo de trabalho estabelecido, já que a cada alteração nos arquivos do projeto exigirá que o contêiner seja reiniciado de forma manual. Através de Docker Compose Extends é possível criar arquivos compose para diferentes ambientes e que utilizam variações do arquivo original, porém não vamos adentrar nesta questão no artigo.

Supondo que precisamos fazer com que o servidor HTTP seja reiniciado a cada alteração de arquivos e o contêiner não precise ser recriado novamente podemos utilizar o **nodemon** ou qualquer outra dependência com esta finalidade. Nesta etapa entra a questão de termos configurado os *volumes* no *docker-compose.yml* anteriormente.

```
volumes:
   - .:/usr/app
```

Perceba que mapeamos um volume a ser monitorado, isto significa que a cada alteração na pasta do projeto o conteúdo será copiado novamente para o contêiner. Por o mesmo estar com o **nodemon** iniciado, automaticamente o servidor será reiniciado.

Instale o **nodemon** utilizando **npm i nodemon --save** . Vamos editar o **package.json** com o comando de inicialização, neste exemplo sobrescrevendo o *script* start (não utilize esta abordagem para ambiente de produção). Em start adicione: **nodemon index.js**

```
"scripts": {
  "start": "nodemon index"
},
```

Tudo pronto! Volte a inicializar a aplicação utilizando **docker-compose up** sem utilizar o **-d** para ser possível testar se o **nodemon** está atuando. Ao ficar visível que o **nodemon** iniciou, basta realizar alteração em qualquer arquivo da pasta e salvar, automaticamente perceba que o **nodemon** terá reiniciado o servidor.

```
tiagoboeing@TIAGOBOEING:/mnt/d/new-project$ docker-compose up
Starting newproject_app_1 ...
Starting newproject_app_1 ... done
Attaching to newproject_app_1
app_1  |
app_1  | > new-project@1.0.0 start /usr/app
app_1  | > nodemon index
app_1  |
app_1  | [nodemon] 2.0.3
app_1  | [nodemon] to restart at any time, enter `rs`
app_1  | [nodemon] watching path(s): *.*
app_1  | [nodemon] watching extensions: js,mjs,json
app_1  | [nodemon] starting `node index index.js`
app_1  | Example app listening on port 3000
app_1  | [nodemon] restarting due to changes...
app_1  | [nodemon] starting `node index index.js`
app_1  | Example app listening on port 3000
```

4. **Conclusão**

O desenvolvimento aliado à utilização de contêineres *Docker* visa garantir que as aplicações sejam concebidas de uma maneira pronta para implantação em qualquer ambiente, além de poupar tempo com inúmeras configurações de serviços paralelos a cada novo desenvolver que integrará o projeto. É possível dessa maneira garantir também cenários homogêneos, significando que o ambiente de desenvolvimento ou o de produção possuem características em comum.

Embora a utilização de contêineres possa acabar adicionando certa complexidade a solução, os *volumes* do *Docker* foram desenvolvidos exatamente para a finalidade de monitorar determinados diretórios e permitir a comunicação entre contêineres.

5. **Referências**

What is Docker? (2020). https://opensource.com/resources/what-docker, [accessed on May 9].

What exactly is Node.js? (2020). https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/, [accessed on May 9].