Развертывание приложения Node.js с помощью Docker: инструкция

timeweb.cloud/tutorials/nodejs/razvertyvanie-prilozheniya-node-js-s-pomoshchyu-docker

Миша Курушин 5 мая 2023 г.

Вы когда-нибудь пытались развернуть собственное приложение где-то за пределами вашей локальной машины? Запустить разработанный продукт (например, сервер на Node.js) на другом компьютере — иногда непростая задача.

Программные зависимости, переменные среды, файлы конфигурации — необходимо все это настроить, чтобы запустить даже самое простое приложение. И делать это вручную — рутинная и ненадежная работа. Нужна автоматизация.

Множество современных технологий стремятся решить проблему различных сред. Контейнеризация — одно из таких направлений. И именно Docker здесь — самый часто используемый инструмент.

Зачем нужен Docker?

Docker позволяет упаковать приложение, окружение и зависимости в так называемый *контейнер*.

Сперва создается образ приложения — код, библиотеки, файлы конфигурации, переменные среды и окружение. Все, что находится внутри образа, необходимо для сборки и запуска приложения.

Контейнером же называется непосредственно экземпляр этого образа. Если провести аналогию из языков программирования, то образ — это класс, а контейнер — экземпляр этого класса.

В отличие от виртуальной машины, контейнер является лишь процессом операционной системы.

По сути, Docker создает абстракцию над низкоуровневыми инструментами операционной системы, позволяя запускать один или несколько контейнерных процессов внутри виртуализированных экземпляров операционной системы Linux.

Несмотря на то, что Docker отнюдь не панацея в вопросах автоматизации деплоя, он решает множество важных задач:

- Быстро развертывает приложения
- Обеспечивает переносимость между машинами
- Имеет контроль версий
- Позволяет строить гибкую архитектуру с использованием компонентов
- Уменьшает накладные расходы при обслуживание за счет своей компактности

Что необходимо установить?

Прежде чем начать настройку, убедитесь что вы установили все необходимые программы на ваш компьютер. А именно:

- Node.js
- Docker

У нас есть инструкции по установке Docker на Ubuntu, а для Node.js — инструкции для разных операционных систем.

Этот материал предполагает, что читатель уже имеет опыт работы с платформой Node.js, а возможно и знаком с Docker.

Шаг 1 — Создание приложения Node.js

Конфигурация и зависимости

Сперва нужно создать каталог, в котором будут находиться исходные файлы приложения. Назовем его `node_app`:

```
mkdir node_app
```

Теперь можно перейти в этот каталог. В нашем случае он будет считаться корневым:

```
cd node_app
```

Как и в любом проекте на Node.js, нам понадобится конфигурационный файл. Создадим и откроем его. В Linux это можно сделать через `nano`:

```
nano package.json
```

Информация о проекте стандартная:

```
{
  "name": "node-app-by-timeweb",
  "description": "node with docker",
  "version": "1.0.0",
  "main": "timeweb.js",
  "keywords": [
    "nodejs",
    "express",
    "docker"
  ],
  "dependencies": {
    "express": "^4.16.4"
  },
  "scripts": {
    "start": "node timeweb.js"
  }
}
```

Как вы знаете, этот файл включает в себя общую информацию о проекте, авторе и лицензии. Все это нужно для в пакетного менеджера NPM, который отвечает за установку зависимостей и публикацию проектов в официальную библиотеку.

Обратите внимание, что самые важные параметры в этом `package.json`:

- Точка входа `main` для приложения — файл `timeweb.js`
- В зависимостях `dependencies` указан сетевой фреймворк `Express`, на котором построен этот пример

Теперь можно сохранить и закрыть файл. Осталось только установить зависимости:

```
npm install
```

Исходный код приложения

Примером будет простое серверное приложение, выводящее статичную веб-страницу по запросу пользователя — `index.html`.

Структура файлов такая:

- `timeweb.js` — точка входа, которая обрабатывает запросы и выполняет роутинг;
- `index.html` — разметка веб-страницы.

Стоит отметить, что CSS-стили для упрощения примера мы напишем сразу в HTML. Разумеется, в реальных проектах визуальное описание веб-страницы располагается в отдельных файлах вроде `style.css` — часто с применением транспиляторов SASS, LESS или SCSS.

Как и прежде, с помощью `nano`, создадим и откроем `timeweb.js`:

```
nano timeweb.js
```

Он будет содержать только самый минимальный код для запуска веб-сервера:

```
  const express = require('express'); // подключаем фреймворк Express (модуль)

const app = express(); // создаем экземпляр приложения
const router = express.Router(); // создаем экземпляр роутера

const path = __dirname; // записываем путь до рабочего каталога
const port = 8080; // записываем порт сервера

// выводим в консоль HTTP METHOD при каждом запросе
router.use(function (req,res,next) {
  console.log('/' + req.method);
  next();
});

// отвечаем на запрос главной страницы файлом index.html
router.get('/', function(req,res){
  res.sendFile(path + 'index.html');
});

// подключаем роутер к приложению
app.use('/', router);

// начинаем прослушивать порт 8080, тем самым запуская http-сервер
app.listen(port, function () {
  console.log('Listening on port 8080')
})
```

Более подробное описание всего функционала фреймворка, а также примеры его использования можно найти в официальной документации Express.

Файл HTML-разметки `index.html` выглядит довольно тривиально:

```
  <!DOCTYPE html>
<html lang="en">

<head>
    <title>NodeJS app with Docker by TimeWeb</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>
    <div>Hello World from TimeWeb!</div>
</body>

<style>
body
{
   height: 100vh;
   display: flex;
   align-items: center;
   justify-content: center;
}

body > div
{
   padding: 12px;
   color: white;
   font-weight: bold;
   background: black;
}
</style>
</html>
```

Чтобы убедиться, что все отображается корректно, вы можете открыть файл `index.html` в браузере. Фраза “Hello World from TimeWeb!” должна отобразиться в центральной части страницы, вместе с темным обрамлением.

На этом наше импровизированное приложение можно считать законченным. Теперь можно перейти к самой докеризации.

Шаг 2 — Создание Dockerfile

Dockerfile — это такой текстовый документ, который содержит инструкции по сборке Docker-образа.

Все инструкции выполняются ровно в том порядке, в каком они записаны в этом файле. Формат записи прост — указывается название инструкции и ее аргументы. Это чем-то похоже на функции в языках программирования. Комментарии пишутся после `#`.

```
  # comment
INSTRUCTION arguments
```

Хотя имена инструкций не чувствительны к регистру, их принято писать большими буквами, чтобы они визуально не сливались с аргументами.

Давайте создадим и откроем `Dockerfile`, после чего можно перейти к его редактированию:

```
  nano Dockerfile
```

Установка образа Node.js

Docker будет последовательно выполнять инструкции из Dockerfile каждый раз, когда конечный пользователь будет выполнять развертывание вашего приложения.

Поэтому первое, что ему понадобится — сам Node. Соответствующую инструкцию необходимо добавить в Dockerfile:

```
  FROM node:19-alpine
```

В данном случае команда `FROM` устанавливает на машину официальный образ Node.js Alpine Linux 19 версии.

На всякий случай — у Docker есть официальная библиотекаDocker Hub, в которой хранятся образы контейнеров от разработчиков со всего мира. Разумеется, Node.js там тоже представлен.

> Если Docker Hub недоступен, можно использовать наш бесплатный прокси, который возобновляет этот доступ.

Кстати, если посмотреть на код Node.js на GitHub, то можно заметить аналогичный Dockerfile, который выполняет всю работу по настройке среды для запуска Node на машине пользователя.

Если провести очень простую аналогию, то `Dockerfile` в Docker — это почти то же самое, что `package.json` в NPM. Он настраивает проект и «тащит» за собой все зависимости, причем рекурсивно — Dockerfile уровня выше устанавливает образ с Dockerfile-ом уровня ниже и так далее.

Установка рабочего каталога

Образу Docker (который впоследствии превратится в контейнер) нужно указать, в каком каталоге нужно выполнять остальные команды, которые будут оперировать файлами и папками. Например, команды `RUN`, `CMD`, `ENTRYPOINT`, `COPY`или `ADD`.

Для этого есть инструкция `WORKDIR`, которой в качестве аргумента передается путь каталога:

```
WORKDIR /app
```

Копирование конфигурационных файлов

С помощью команды `COPY` нужно скопировать файлы `package.json` и `package-lock.json` из каталога проекта на локальном компьютере в файловую систему контейнера, а точнее в указанный ранее каталог:

```
COPY package.json package-lock.json ./
```

Из-за того, что Dockerfile находится в каталоге проекта, образ контейнера содержит в себе все необходимые файлы. Однако образ — не контейнер. Поэтому с помощью команды `COPY` мы сообщаем Docker какие конкретно файлы нужно перенести в «виртуальное пространство» контейнера.

Установка зависимостей NPM

Поскольку установленный ранее каталог приложения уже содержит `package.json` и `package-lock.json`, можно загрузить необходимые зависимости из реестра NPM.

Для этих целей обычно выполняется команда `npm install`. Чтобы Docker сделал это автоматически нужно указать инструкцию `RUN`:

```
RUN npm install
```

Docker выполнит эту команду в ранее указанном каталоге `/app`.

Обратите внимание, что инструкция `RUN` выполняет команды во время установки образа (а не запуска контейнера), который впоследствии будет существовать как контейнер. Кстати, команды можно указывать в виде последовательной цепочки:

```
RUN [“command1”, “command2”, “command3”]
```

Копирование остальных файлов

После установки всех зависимостей можно скопировать все остальные файлы проекта в каталог `/app`. Для этого используется та же команда `COPY`, но с указанием всей директории, а не конкретных файлов:

```
COPY . ./
```

Запуск приложения

Теперь можно указать команду, которая будет запускать само приложение. Для этого нужно использовать инструкцию `CMD`. От инструкции `RUN` она отличается тем, что выполняет указанные команды уже во время выполнения контейнера, а не в момент установки образа:

```
CMD npm start
```

Не забудьте, что в `package.json` у вас уже определена команда `start`:

```
  "scripts": {
  "start": "node timeweb.js"
}
```

Итоговая конфигурация образа в Dockerfile

Итак, после указания полной последовательности действий, возложенных на Docker, полный код `Dockerfile` должен выглядеть следующим образом:

```
  # устанавливаем официальный образ Node.js
FROM node:19-alpine

# указываем рабочую (корневую) директорию
WORKDIR /app

# копируем основные файлы приложения в рабочую директорию
COPY package.json package-lock.json ./

# устанавливаем указанные зависимости NPM на этапе установки образа
RUN npm install

# после установки копируем все файлы проекта в корневую директорию
COPY . ./

# запускаем основной скрипт в момент запуска контейнера
CMD npm start
```

Все! Минимальный набор инструкций указан. Теперь можно попробовать создать образ и на его основе запустить контейнер.

Немного про файл .dockerignore

`.dockerignore` — это еще один конфигурационный файл, содержащий каталоги, которые необходимо исключить при создании образа Docker. Скажем так, в папке вашего проекта может быть много файлов, которые никак не связаны с создаваемым образом, хотя и важны при разработке.

На самом деле `.dockerignore` гораздо важнее, чем может показаться на первый взгляд — он предотвращает попадание слишком больших или конфиденциальных файлов в образ. Он также ограничивает действие команд `ADD` или `COPY`, используемых в `Dockerfile`.

Например, каждый раз, когда вы используете команду `docker build`, Docker сверяет кеш образа с состоянием файловой системы. Если есть изменения — сборка выполняется заново.

Однако, если некоторые файлы в вашем каталоге довольно часто обновляются, но при это не нужны для построения образа — их следует исключить, чтобы не выполнять бессмысленную пересборку.

Создание и редактирование .dockerignore

Файл `.dockerignore` создается в корневом каталоге вашего проекта. Внутри него на каждой новой строчке указываются названия файлов и директорий для исключения.

```
  # it's a comment
README.md
```

Как и `Dockerfile`, символ `#` обозначает начало комментария. Кстати, есть и способы и более общего указания файлов:

```
  */folder
```

В данном случае все директории (или файлы без расширения) с именем `folder` в любом каталоге на один уровень ниже будут исключены из сборки.

Впрочем, можно игнорировать директории и файлы рекурсивно — в корневом и во всех уровнях ниже:

```
  **/folder
```

При этом, с помощью `!` файл с конкретным именем можно исключить из исключения. В данном случае кроме `README.md` файлы с расширением `.md` не попадут в сборку:

```
  *.md
!README.md
```

Шаг 3 — Сборка образа Docker

Образ Docker создается на основе описания в `Dockerfile`. Для этого есть соответствующая команда, которая запускается из корня проекта — там, где расположен `Dockerfile`:

```
docker build . -t nodeproject
```

Флаг `-t` необходим для установки имени тега нового образа. Впоследствии на него можно будет ссылаться через `nodeproject:latest`.

После этого можно удостовериться в том, что образ был действительно создан:

```
docker images nodeproject:latest
```

Эта команда выводит информацию о конкретном образе Docker:

```
REPOSITORY     TAG         IMAGE ID             CREATED            SIZE
nodeproject      latest      gk8orf8fre489     3 minutes ago    15MB
```

Соответственно, если не указывать конкретное название, то в консоли выйдет информация обо всех образах на компьютере.

Шаг 4 — Запуск контейнера Docker

Каждый созданный образ можно запускать в виде контейнера. Для этого указывается его имя:

```
docker run nodeproject
```

Запущенный в виде контейнера образ Docker является типичным процессом операционной системы, в котором файловая система, сеть и дерево процессов отделены от хост-компьютера.

Все консольные выводы вашего приложения Node.js будут выводиться в том же терминале, в котором был запущен контейнер. Однако, привязка процесса контейнера к конкретному экземпляру терминала — не лучшее решение.

Поэтому более разумной практикой является запуск контейнера в фоновом режиме с помощью специального флага `--detach` или `-d`.

```
docker run -d nodeproject
```

Docker запустит контейнер в автономном режиме, написав в терминале специальный идентификатор. Его можно будет использовать для доступа к контейнеру в последующих командах:

```
9341f8b2532b121e9dea8aeb55563c24302df96710c411c699a612e794e89ee4
```

Кстати, стоит сказать, что перед каждым запуском лучше всегда проверять, не был ли контейнер уже запущен — если только не предполагается обратное. Для этого в Docker есть команда, выводящая список всех запущенных на компьютере контейнеров:

```
docker ps
```

Таким образом можно увидеть идентификатор контейнера, образ на основе которого запущен контейнер, команду, используемую для запуска контейнера, время его создания, текущий статус, порты, предоставляемые контейнером, и само имя контейнера. Кстати, по умолчанию Docker присваивает контейнеру случайное имя, но его можно изменить с помощью флага `--name`.

Обратите внимание, речь идет именно про имя контейнера, а не образа. Предположим, вы запускаете контейнер с именем `myname`:

```
docker run -d --name myname nodeproject
```

Теперь вы сможете его остановить, указав имя:

```
docker stop myname
```

А также удалить:

```
docker rm myname
```

Логи автономного контейнера

Запущенный в фоновом режиме контейнер не показывает выводы в консоль в явном виде. Однако, они по прежнему существуют. Их можно увидеть так:

```
docker logs myname
```

Теперь все, что успело вывести ваше приложение в консоль, будет напечатано в терминале.

Заключение

Эта статья очень коротко рассказывает, что такое Docker, как он работает и почему он может быть полезен при разработке Node-приложений.

Понимая как правильно форматировать `Dockerfile` и запускать приложение Node.js с помощью Docker, вы можете автоматизировать процесс развертывания программных продуктов на машинах конечных пользователей.

Подобные решения наиболее актуальный в DevOps-разработке, в частности при построении CI/CD-пайплайнов — непрерывной интеграции и развертывании.