TUTORIAL

Create a MEAN app with Angular and I Compose

Angular

By
Updated on September 15, 2020

Introduction

Note: Update: 30/03/2019

This article has been updated based on the updates to both `docker` article was written. The current version of `angular` is 7, the updates docker volume to the `angular` client so that you don't need to run `d` every time.

Docker allows us to run applications inside containers. These conta communicate with each other.

> Docker containers wrap a piece of software in a comp contains everything needed to run: code, runtime, sys

> libraries – anything that can be installed on a server.
> the software will always run the same, regardless of i

We'll build an angular app in one container, point it to an Express A
which connects to MongoDB in another container.

If you haven't worked with Docker before, this would be a good sta
explain every step covered, in some detail.

Why Use Docker

- Docker images usually include only what your application nee
don't have to worry about having a whole operating system wi
use. This results in smaller images of your application.
- Platform Independent - I bet you've heard of the phrase 'It wo
doesn't work on the server'. With Docker, all either environmer
Docker Engine or the Docker Daemon, and when we have a su
image, it should run anywhere.
- Once you have an image of your application built, you can eas
anyone who wants to run your application. They need not wor
or setting up their individual environments. All they need to ha
installed.
- Isolation - You'll see from the article that I try to separate the
become independent, and only point to each other. The reaso
part of our entire application should be somewhat independer
own. Docker in this instance would make scaling these individ
spinning up another instance of their images. This concept of
independently scalable parts of an entire system are what is c
Approach. You can read more about it in Introduction to Micro
- Docker images usually have tags, referring to their versions. Th
versioned builds of your image, enabling you to roll back to a p
something unexpected break.

Prerequisites

You need to have docker and docker-compose installed in your set
installing docker in your given platform can be found here.

Instructions for installing docker-compose can be found here.

Verify your installation by running:

```
$ docker  -v
```

```
Output
Docker version 18.09.2, build 6247962
```

```
$ docker-compose -v
```

```
Output
docker-compose version 1.23.2, build 1110ad01
```

```
$ node -v
```

```
Output
v11.12.0
```

Next, you need to know how to build a simple Angular app and an E
using the Angular CLI to build a simple app.

Single Builds Approach

We'll now separately build out these three parts of our app. The ap
take is building the app in our local environment, then dockerizing

Once these are running, we'll connect the three docker containers.
building two containers, Angular and the Express/Node API. The thi
a MongoDB image that we'll just pull from the Docker Hub.

> Docker Hub is a repository for docker images. It's whe
> official docker images such as MongoDB, NodeJs, Ubu
> create custom images and push them to Docker Hub
> pull and use.

Let's create a directory for our whole setup, we'll call it `mean-docke`

```
$ mkdir mean-docker
```

Angular Client App

Next, we'll create an Angular app and make sure it runs in a `docker`

Create a directory called `angular-client` inside the `mean-docker` di
above, and initialize an Angular App with the Angular CLI.

We'll use `npx`, a tool that allows us to run CLI apps without installi
It comes preinstalled when you install Node.js since version 5.2.0

```
$ npx @angular/cli new angular-client
```

```
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use? CSS
```

This scaffolds an Angular app, and npm installs the app's depender
structure should be like this

```
└── mean-docker
    └── angular-client
        ├── README.md
        ├── angular.json
        ├── e2e
        ├── node_modules
        ├── package.json
        ├── package-lock.json
        ├── src
        ├── tsconfig.json
        └── tslint.json
```

Running npm start, inside the `angular-client` directory should star
`http://localhost:4200`.

Dockerizing Angular 2 Client App

To dockerize any app, we usually need to write a Dockerfile

> A Dockerfile is a text document that contains all the
> could call on the command line to assemble an imag

To quickly brainstorm on what our angular app needs in order to ru

- We need an image with Node.js installed on it
- We could have the Angular CLI installed on the image, but the
  as a dependency, so it's not a requirement.
- We can add our Angular app to the image and install its deper
- It needs to expose port 4200 so that we can access it from ou
  `localhost:4200`.
- If all these requirements are met, we can run `npm start` in the
  runs `ng serve` since it's a script in the `package.json` file, creat
  our app should run.

Those are the exact instructions we are going to write in our Docke

mean-docker/angular-client/Dockerfile

CONTENTS

RELATED

*ngFor Directive in Angular

Tutorial

*ngIf Directive in Angular 2

```
# Create image based on the official Node 10 image from dockerh
FROM node:10

# Create a directory where our app will be placed
RUN mkdir -p /app

# Change directory so that our commands run inside this new dir
WORKDIR /app

# Copy dependency definitions
COPY package*.json /app/

# Install dependencies
RUN npm install

# Get all the code needed to run the app
COPY . /app/

# Expose the port the app runs in
EXPOSE 4200

# Serve the app
CMD ["npm", "start"]
```

I've commented on the file to show what each instruction clearly d

> **Note:** Before we build the image, if you are keen, you may have notic
> `/app/` copies our whole directory into the container, including `node_`
> and other files that are irrelevant to our container, we can add a `.do`
> what is to be ignored. This file is usually sometimes identical to the

Create a `.dockerignore` file.

mean-docker/angular-client/.dockerignore

```
node_modules/
```

One last thing we have to do before building the image is to ensure
from the host created by the `docker` image. To ensure this, go into
change the `start` script to:

mean-docker/angular-client/package.json

```
{
  ...
  "scripts": {
    "start": "ng serve --host 0.0.0.0",
    ...
  },
```

```
  ...
}
```

To build the image we will use `docker build` command. The syntax

```
$ docker build -t <image_tag>:<tag> <directory_with_Dockerfile
```

Make sure you are in the `mean_docker/angular-client` directory, the

```
$ cd angular-client
```

```
$ docker build -t angular-client:dev .
```

`-t` is a shortform of `--tag`, and refers to the name or tag given to
this case the tag will be `angular-client:dev`.

The `.` (dot) at the end refers to the current directory. Docker will l
our current directory and use it to build an image.

This could take a while depending on your internet connection.

Now that the image is built, we can run a container based on that i

```
$ docker run -d --name <container_name> -p <host-port:exposed-
```

The `-d` flag tells `docker` to run the container in `detached` mode. M
you back to your host, without going into the container.

```
$ docker run -d --name angular-client -p 4200:4200 angular-cli
```

`--name` refers to the name that will be assigned to the container.

`-p` or `--port` refers to which port our host machine should point t
container. In this case, `localhost:4200` should point to `dockerhost:`
syntax `4200:4200`.

Visit `localhost:4200` in your host browser should be serving the `an`
container.

You can stop the container running with:

```
$ docker stop angular-client
```

Dockerize the Express Server API

We've containerized the `angular` app, we are now two steps away f

Containerizing an express app should now be straightforward. Crea
`mean-docker` directory called `express-server`.

```
$ mkdir express-server
```

Add the following `package.json` file inside the app.

mean-docker/express-server/package.json

```
{
  "name": "express-server",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "body-parser": "~1.15.2",
    "express": "~4.14.0"
  }
}
```

Then, we'll create a simple express app inside it. Create a file `serv`

```
$ cd express-serve
```

```
$ touch server.js
```

```
$ mkdir routes && cd routes
```

```
$ touch api.js
```

mean-docker/express-server/server.js

```
// Get dependencies
const express = require('express');
const path = require('path');
const http = require('http');
const bodyParser = require('body-parser');

// Get our API routes
const api = require('./routes/api');

const app = express();
```

```
// Parsers for POST data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// Set our api routes
app.use('/', api);

// Get port from environment and store in Express.
const port = process.env.PORT || '3000';
app.set('port', port);

// Create HTTP server.
const server = http.createServer(app);

// Listen on provided port, on all network interfaces.
server.listen(port, () => console.log(`API running on localhost
```

```
[labe mean-docker/express-server/routes/api.js]
const express = require('express');
const router = express.Router();

// GET api listing.
router.get('/', (req, res) => {
    res.send('api works');
});

module.exports = router;
```

This is a simple `express` app, install the dependencies and start th

```
$ npm install
```

```
$ npm start
```

Going to `localhost:3000` in your browser should serve the app.

To run this app inside a Docker container, we'll also create a Docke
pretty similar to what we already have for the `angular-client`.

mean-docker/express-server/Dockerfile

```
# Create image based on the official Node 6 image from the dock
FROM node:6

# Create a directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our commands run inside this new dir
WORKDIR /usr/src/app
```

```
# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependencies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 3000

# Serve the app
CMD ["npm", "start"]
```

You can see the file is pretty much the same as the `angular-client` the exposed port.

You could also add a `.dockerignore` file to ignore files we do not ne

mean-docker/express-server/.dockerignore

```
node_modules/
```

We can then build the image and run a container based on the ima

```
$ docker build -t express-server:dev .
```

```
$ docker run -d --name express-server -p 3000:3000 express-ser
```

Going to `localhost:3000` in your browser should serve the API.

Once you are done, you can stop the container with

```
$ docker stop express-server
```

MongoDB container

The last part of our MEAN setup, before we connect them all toget we can't have a Dockerfile to build a MongoDB image, because one Docker Hub. We only need to know how to run it.

Assuming we had a MongoDB image already, we'd run a container b

```
$ docker run -d --name mongodb -p 27017:27017 mongo
```

The image name in this instance is `mongo`, the last parameter, and
be `mongodb`.

Docker will check to see if you have a mongo image already downlo
will look for the image in the Dockerhub. If you run the above comm
`mongodb` instance running inside a container.

To check if MongoDB is running, simply go to `http://localhost:2701`
you should see this message. `It looks like you are trying to acc`
`the native driver port.`

Alternatively, if you have mongo installed in your host machine, sim
terminal. And it should run and give you the mongo shell, without a

Docker Compose

To connect and run multiple containers with docker, we use Docke

> Compose is a tool for defining and running multi-cont
> applications. With Compose, you use a Compose file t
> application's services. Then, using a single command,
> all the services from your configuration.

`docker-compose` is usually installed when you install `docker`. So to s
it installed, run:

```
$ docker-compose
```

You should see a list of commands from docker-compose. If not, yo
installation here

Note: Ensure that you have docker-compose version 1.6 and above by
`compose -v`

Create a `docker-compose.yml` file at the root of our setup.

```
$ touch docker-compose.yml
```

Our directory tree should now look like this.

```
.
├── angular-client
├── docker-compose.yml
└── express-server
```

Then edit the `docker-compose.yml` file

mean-docker/docker-compose.yml

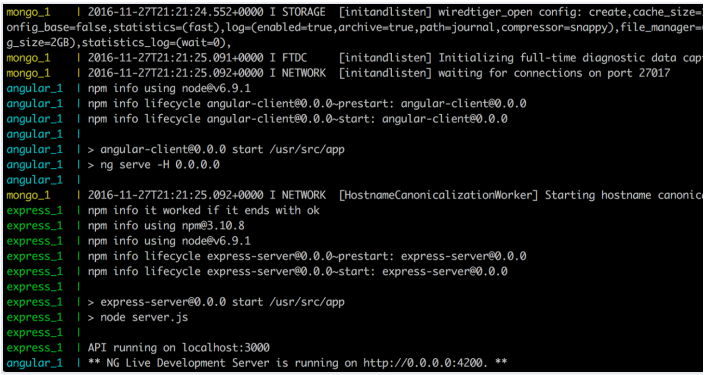
```
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  angular: # name of the first service
    build: angular-client # specify the directory of the Docker
    ports:
      - "4200:4200" # specify port forewarding

  express: #name of the second service
    build: express-server # specify the directory of the Docker
    ports:
      - "3000:3000" #specify ports forwarding

  database: # name of the third service
    image: mongo # specify image to build container from
    ports:
      - "27017:27017" # specify port forwarding
```

The `docker-compose.yml` file is a simple configuration file telling `do` containers to build. That's pretty much it.

Now, to run containers based on the three images, simply run

```
$ docker-compose up
```

This will build the images if not already built, and run them. Once i terminal looks something like this.



You can visit all three apps: `http://localhost:4200`, `http://localho` `mongodb://localhost:27017`. And you'll see that all three containers

Connecting the 3 Docker containers

Finally, the fun part.

Express and MongoDB

We now finally need to connect the three containers. We'll first cre
feature in our API using mongoose. You can go through Easily Devel
Apps with Mongoose to get a more detailed explanation of mongoo

First of all, add `mongoose` to your `express` server `package.json`

mean-docker/express-server/package.json

```
{
  "name": "express-server",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "body-parser": "~1.15.2",
    "express": "~4.14.0",
    "mongoose": "^4.7.0"
  }
}
```

We need to update our API to use MongoDB:

mean-docker/express-server/routes/api.js

```
// Import dependencies
const mongoose = require('mongoose');
const express = require('express');
const router = express.Router();

// MongoDB URL from the docker-compose file
const dbHost = 'mongodb://database/mean-docker';

// Connect to mongodb
mongoose.connect(dbHost);

// create mongoose schema
const userSchema = new mongoose.Schema({
  name: String,
  age: Number
});

// create mongoose model
const User = mongoose.model('User', userSchema);
```

```
// GET api listing.
router.get('/', (req, res) => {
        res.send('api works');
});

// GET all users.
router.get('/users', (req, res) => {
    User.find({}, (err, users) => {
        if (err) res.status(500).send(error)

        res.status(200).json(users);
    });
});

// GET one users.
router.get('/users/:id', (req, res) => {
    User.findById(req.param.id, (err, users) => {
        if (err) res.status(500).send(error)

        res.status(200).json(users);
    });
});

// Create a user.
router.post('/users', (req, res) => {
    let user = new User({
        name: req.body.name,
        age: req.body.age
    });

    user.save(error => {
        if (error) res.status(500).send(error);

        res.status(201).json({
            message: 'User created successfully'
        });
    });
});

module.exports = router;
```

Two main differences, first of all, our connection to MongoDB is in
`'mongodb://database/mean-docker';`. This `database` is the same as tl
created in the `docker-compose` file.

We've also added rest routes `GET /users`, `GET /users/:id` and `POST`

Update the `docker-compose` file, telling the express service to link t

mean-docker/docker-compose.yml

```
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  angular: # name of the first service
    build: angular-client # specify the directory of the Docker
    ports:
      - "4200:4200" # specify port forewarding
    volumes:
      - ./angular-client:/app # this will enable changes made t

  express: #name of the second service
    build: express-server # specify the directory of the Docker
    ports:
      - "3000:3000" #specify ports forewarding
    links:
      - database

  database: # name of the third service
    image: mongo # specify image to build container from
    ports:
      - "27017:27017" # specify port forwarding
```

The `links` property of the docker-compose file creates a connectic with the name of the service as the hostname. In this case `databas` Meaning, to connect to it from the `express` service, we should use why we made the `dbHost` equal to `mongodb://database/mean-docker`

Also, I've added a volume to the `angular` service. This will enable c Angular App to automatically trigger recompilation in the container

Angular and Express

The last part is to connect the Angular app to the express server. T make some modifications to our `angular` app to consume the `expr`

Add the Angular HTTP Client.

mean-docker/angular-client/src/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http'; // adc

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
```

```
    BrowserModule,
    HttpClientModule // import http client module
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

mean-docker/angular-client/src/app/app.componer

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'app works!';

  // Link to our api, pointing to localhost
  API = 'http://localhost:3000';

  // Declare empty list of people
  people: any[] = [];

  constructor(private http: HttpClient) {}

  // Angular 2 Life Cycle event when component has been initial
  ngOnInit() {
    this.getAllPeople();
  }

  // Add one person to the API
  addPerson(name, age) {
    this.http.post(`${this.API}/users`, {name, age})
      .subscribe(() => {
        this.getAllPeople();
      })
  }

  // Get all users from the API
  getAllPeople() {
    this.http.get(`${this.API}/users`)
      .subscribe((people: any) => {
        console.log(people)
        this.people = people
      })
  }
}
```

Angular best practices guides usually recommend separating most
service/provider. We've placed all the code in the component here

We've imported the `OnInit` interface, to call events when the comp
added two methods `AddPerson` and `getAllPeople`, that call the API.

Notice that this time around, our `API` is pointing to `localhost`. This
Angular 2 app will be running inside the container, it's served to the
browser is the one that makes requests. It will thus make a reques
API. As a result, we don't need to link Angular and Express in the `d`

Next, we need to make some changes to the template. I first added
the `index.html`

mean-docker/angular-client/src/app/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular Client</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-sc

  <!-- Bootstrap CDN -->
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/

  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root>Loading ... </app-root>
</body>
</html>
```

Then update the `app.component.html` template

mean-docker/angular-client/src/app/app.component

```
<!-- Bootstrap Navbar -->
<nav class="navbar navbar-light bg-faded">
  <div class="container">
    <a class="navbar-brand" href="#">Mean Docker</a>
  </div>
</nav>

<div class="container">
    <h3>Add new person</h3>
    <form>
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="name" #name
```

```
        </div>
        <div class="form-group">
          <label for="age">Age</label>
          <input type="number" class="form-control" id="age" #age
        </div>
        <button type="button" (click)="addPerson(name.value, age.
      </form>

      <h3>People</h3>
      <!-- Bootstrap Card -->
      <div class="card card-block col-md-3" *ngFor="let person of
        <h4 class="card-title">{{person.name}}  {{person.age}}</h
      </div>
    </div>
```

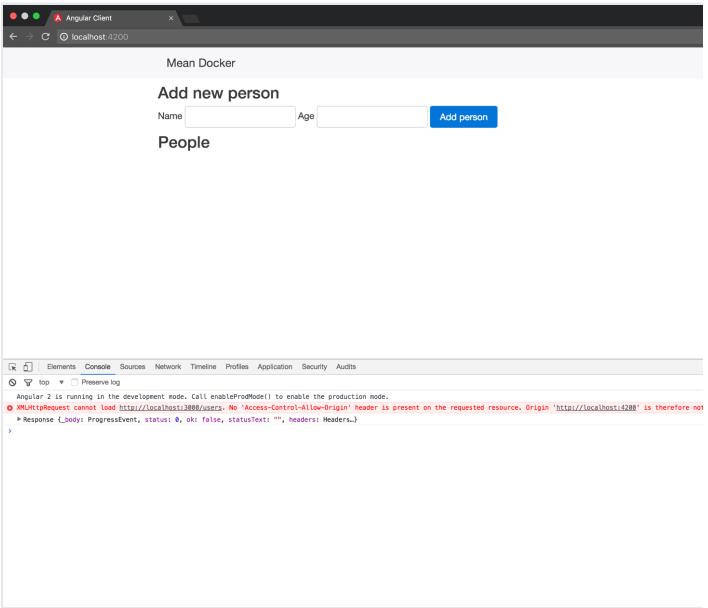
The above template shows the components' properties and binding

Since we've made changes to our code, we need to do a build for c

```
$ docker-compose up --build
```

The `--build` flag tells `docker compose` that we've made changes an build of our images.

Once this is done, go to `localhost:4200` in your browser,



We are getting a `No 'Access-Control-Allow-Origin'` error. To quickly enable `Cross-Origin` in our `express` app. We'll do this with a simple

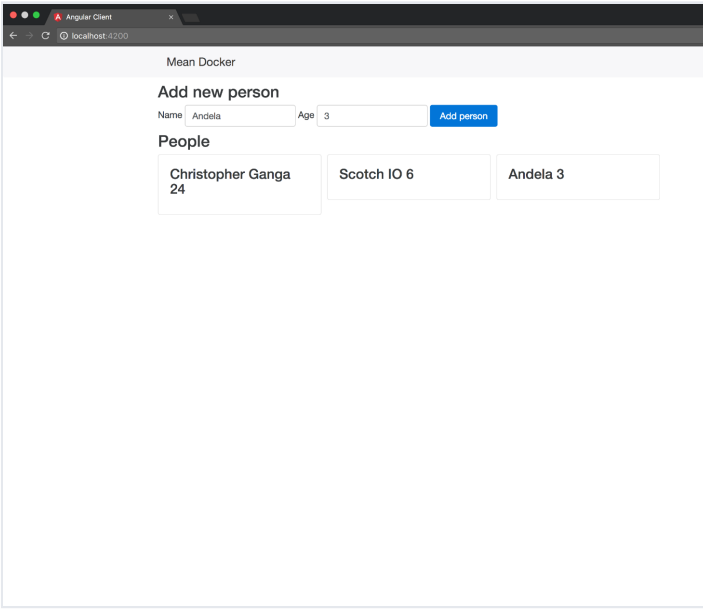
mean-docker/express-server/server.js

```
// Code commented out for brevity

// Parsers for POST data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// Cross Origin middleware
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*")
  res.header("Access-Control-Allow-Headers", "Origin, X-Request
  next()
})

// Set our api routes
app.use('/', api);

// Code commented out for brevity
```

We can now run `docker-compose` again with the `build` flag. You sho `docker` directory.

```
$ docker-compose up --build
```

Going to `localhost:4200` on the browser.



Conclusion

Note: I added an attached volume to the `docker-compose` file, and w
rebuild the service every time we make a change.

I bet you've learned a thing or two about MEAN or `docker` and `dock`

The problem with our set up however is that any time we make cha
`angular` app or the `express` API, we need to run `docker-compose up`

This can get tedious or even boring over time. We'll look at this in a

About the authors

Developer and author at
DigitalOcean.

Still looking for an answer?

Ask a question

Search for

Join the DigitalOcean Community

Join 1M+ other developers and:

- Get help and share knowledge in Q&A
- Subscribe to topics of interest
- Get courses & tools that help you grow as a developer or sma

Join Now

RELATED

*ngFor Directive in Angular

Tutorial

*ngIf Directive in Angular 2

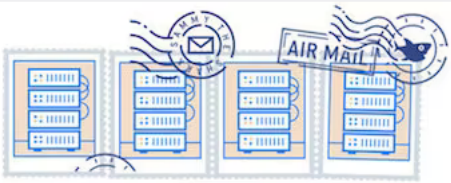
Tutorial

Comments

Leave a comment

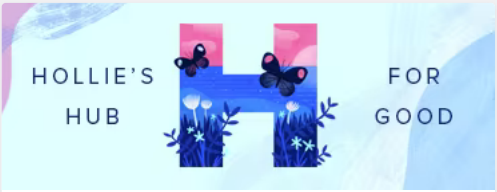
Leave a comment ...

Sign in to Comment



GET OUR BIWEEKLY NEWSLETTER

Sign up for Infrastructure as a Newsletter.



HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

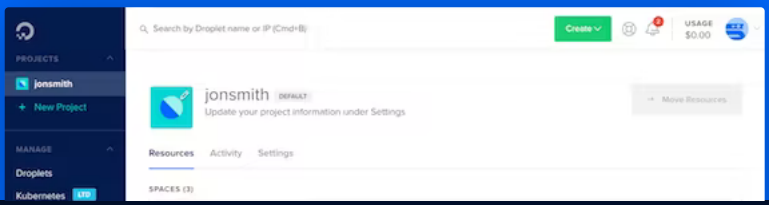
Featured on Community Kubernetes Course Learn Python 3 Machine Learning in Python Getting started with Go Intro to Kubernetes

DigitalOcean Products Virtual Machines Managed Databases Managed Kubernetes Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you're running one virtual machine or ten thousand.

Learn More



Company

About
Leadership
Blog
Careers
Partners