Article

Convert your app from Cloud Foundry with Docker and Kubernetes

Learn how to containerize a CF application and deploy it to Kubernetes

By Marc Velasco

Updated April 29, 2019 | Published April 29, 2019

Cloud Foundry, Docker, and Kubernetes are very popular options for cloud native applications, but they each approach providing an application in slightly different ways. In this article, we'll examine how you can containerize a cloud foundry application and how to deploy a containerized application into Kubernetes.

1.

Starting from Cloud Foundry

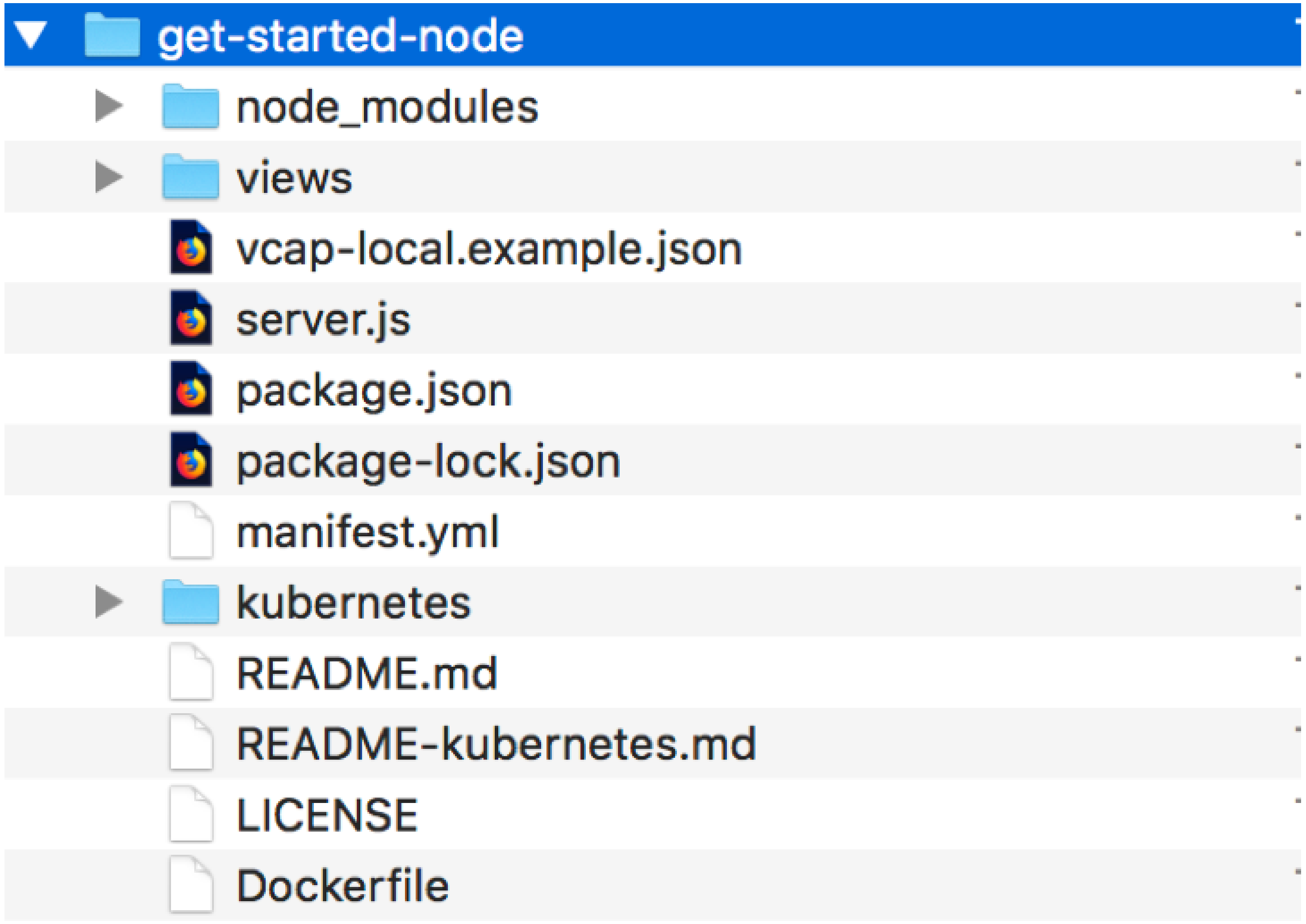
Cloud Foundry applications

When you develop a Cloud Foundry application, your application sits in the repository where you initiated a deployment by using the Cloud Foundry CLIs to package and deploy your application.

A Cloud Foundry application is comprised primarily of your application and a manifest, which defines deployment parameters for your application. When you deploy an application to Cloud Foundry, the tools will search through your directory or specified location, therefore gathering all of your application's artifacts and deployment based on the parameters that you specified in your manifest or utilized in your buildpack.

This knowledge allows you to keep your code separate from your deployment, and when you initiate a `cf` push, the Cloud Foundry tools will simply take your code and do the deployment for you, including packaging.

Let's look at an example of a Cloud Foundry application. Start by setting up and running the nodejs get-started-node sample app as documented here: https://cloud.ibm.com/docs/runtimes/nodejs/getting-started.html#getting-started



```
get-started-node
    node_modules
    views
    vcap-local.example.json
    server.js
    package.json
    package-lock.json
    manifest.yml
    kubernetes
    README.md
    README-kubernetes.md
    LICENSE
    Dockerfile
```

This sample application can run on both Cloud Foundry or Docker. For now, we'll focus on Cloud Foundry.

The main piece we need for Cloud Foundry is the manifest.yml, (seen in the above image of the files and folders):

```
applications:
 - name: GetStartedNode
   random-route: true
   memory: 256M
```

Show more

This file tells Cloud Foundry how to structure the application; we have one application named GetStartedNode that runs with memory 256M.

The code for the Node.js application is very simple. A package.json defines the main entry point as server.js:

```
"name": "get-started-node",
"main": "server.js",
"description": "An introduction to developing Node.js apps on the IBM Cloud platform",
"version": "0.1.1",
```

Show more

While `server.js` contains the code for the main entrypoint, it also controls the overall function of the web app and incorporates the express framework, namely the view render.

You can deploy this application by using the IBM Cloud command line tools as documented here: https://cloud.ibm.com/docs/runtimes/nodejs/getting-started.html#deploy

Here is what the output should look like:

```
ibmcloud cf push
Invoking 'cf push'...
```

```
Pushing from manifest to org test@ibm.com / space dev as test@ibm.com...
Using manifest file get-started-node/manifest.yml
Getting app info...
Creating app with these attributes...
+ name:       GetStartedNode
  path:       /get-started-node
+ memory:     256M
  routes:
+   getstartednode-fluent-reedbuck.mybluemix.net

Creating app GetStartedNode...
```

Show more

Let's see if our app is running:

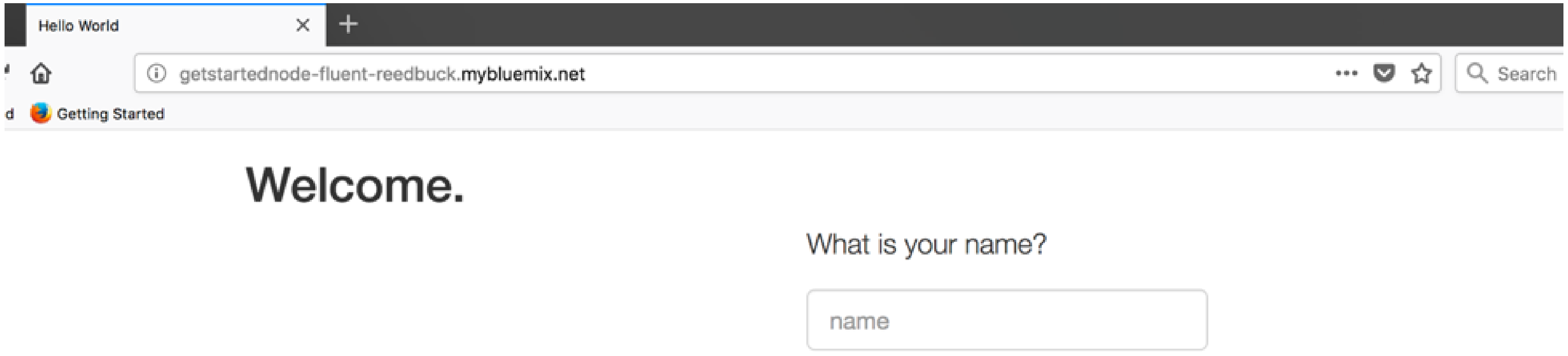
```
ibmcloud cf apps
Invoking 'cf apps'...

Getting apps in org mvelasc@us.ibm.com / space dev as mvelasc@us.ibm.com...
OK

name             requested state   instances   memory   disk   urls
GetStartedNode   started           1/1         256M     1G     getstartednode-fluent-reedbuck.mybluemix.net
```

Show more

Now let's actually go to the running application from the listed URL, getstartednode-fluent-reedbuck.myblumix.net:



So we can see that we easily deployed a sample application for Cloud Foundry that contains the code for the app (Node dependencies and server.js).

Cloud Foundry buildpacks

In Cloud Foundry, the developer is typically only concerned with the application and not the runtime and dependencies. That is because the responsiblity would fall on the buildpack. Cloud Foundry buildpacks are designed to handle the runtime and dependencies that are needed by your applications. Buildpacks exist for all kinds of application runtimes.

As a developer, you can specify a specific buildpack to use when you're performing a cf push; you can also specify it as an attribute in your manifest or the default action. By not specifying a buildpack to use, the Cloud Foundry platform will automatically pick a buildpack for you based on the contents of your application.

So in Cloud Foundry, the notion of runtime and dependencies is abstracted from the actual application development.

How Cloud Foundry is different from Docker

In Docker, that abstraction of runtimes and dependencies does not exist. When it comes to Docker, the runtime and application are combined. Just as a Docker container encapsulates your application, runtime, and dependencies, so too does the definition of that Docker container.

Each Docker container is built from a Dockerfile, a file that specifies how a container is built and the components that it holds. That is the application, the runtime, and the dependencies. This is the big difference between Cloud Foundry and Docker applications. With Docker, a developer must be aware of the runtime and dependencies in the Docker container that Cloud Foundry used to handle for you automatically. However this isn’t as bad as it seems. Keep in mind containers should be fairly lightweight, and short of huge monolithic applications, a docker container is usually relatively small and nimble.

Applying this to our Cloud Foundry application, let’s look at a Dockerfile that builds a Docker container around our application:

```
FROM node:6-alpine

ADD views /app/views
ADD package.json /app
ADD server.js /app

RUN cd /app; npm install

ENV NODE_ENV production
ENV PORT 8080
EXPOSE 8080
```

```
WORKDIR "/app"
CMD [ "npm", "start" ]
```

Show more

In this case, we take the same code base since the application itself does not change; instead we encapsulate the application into the Docker container. Specifically, the first line identifies this container we'll create based on the "node" container from Docker, which you can see here.

The tag "6-alpine" is a specific revision of the node container in the Docker store to use as our base, which means someone a container with node v 6.0.0 on Alpine Linux. We will use this as the base of our container.

The three ADD lines will add the files from our source directory into the container, which loads our application files into the container. After the container is created a copy of the application will live within the container itself.

The `RUN` command installs the node dependencies for our application prior to the container starting up for the first time, this allows us to preload all dependencies so when an instance of our container is instantiated it can start immediately with no dependency load.

The `ENV` and `EXPOSE` tags set the port information for the container to respond to user requests; this is needed as a Docker container generally does not expose ports but instead by the application developer to decide which ports should be exposed and opened for the application to provide service.

The `WORKDIR` changes the context to the directory of our application so that we can run the "CMD" command on container startup, this is the command that executes as soon as the container starts, which in our example starts the node application.

Now let's create the container. First you'll need to do a docker login as we are pulling an image from Docker Hub:

```
$docker login
Authenticating with existing credentials...
Stored credentials invalid or expired
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over
Username (test@ ibm.com): test
Password:
Login Succeeded
```

Show more

Now create the container by using the Dockerfile in our current directory. Note that each line in the Dockerfile represents a step in building the Docker container. When you finish creating the container, you will have created a Docker container `dockerdemo`, with a tag of `test1`:

```
$docker build .
Sending build context to Docker daemon   12.7MB
Step 1/10 : FROM node:6-alpine
 ---> 7c9d8e1567b1
Step 2/10 : ADD views /app/views
 ---> cc32b8c96c8a
Step 3/10 : ADD package.json /app
 ---> 577d5c8eedf8
Step 4/10 : ADD server.js /app
 ---> 8d54bf8ac53b
Step 5/10 : RUN cd /app; npm install
 ---> Running in 7c8ee1dc7074
get-started-node@0.1.1 /app
+-- @cloudant/cloudant@3.0.2
| +-- @types/request@2.48.1
```

Show more

We can see in Docker that the images are available now, including our new container and node that we used as our base:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
dockerdemo          test1               5a6307ac6e7e        32 seconds ago      79.9MB
node                6-alpine            7c9d8e1567b1        3 weeks ago         55.6MB
```

Show more

Now let’s deploy an instance of our new container:

```
$docker run -d -p 8080:8080 dockerdemo:test1
3cca2823db352b9e8931e3e4ce3882de1099473765f630d139c45d51e8ecbbe4
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
3cca2823db35        dockerdemo:test1    "npm start"         44 seconds ago      Up 43 seconds
```

Show more

Now let’s verify with our browser. You should see the following:

verify with browser

2. On to Kubernetes

At this point, you should have a Docker container for your application. The next step in our Kubernetes journey is to deploy this container in a configuration that meets your needs. The key part is to determine what types of deployment and what attributes that you’ll need to provide for the deployment. But first, let’s understand what deploying to Kubernetes actually means.

What’s involved in deploying to Kubernetes?

To deploy into Kubernetes, you’ll need to push your Docker container, with your application in it, to a Docker registry. This can be a public registry, like the Docker store or a private Docker registry. Note that most private Kubernetes distributions will include a Docker registry as a part of their services.

Our goal in using Kubernetes here is to deploy a Kubernetes pod, which is the unit of encapsulation for one to many Docker containers. We define the parameters, attributes, and container information for this pod in a yaml file that is used with the Kubernetes CLI to create the pod. The container information is the Docker registry and container image name that Kubernetes will pull when it builds the pod.

The yaml file also defines network resources that provide access to your pod. In Kubernetes, the pod just controls the availability of a set of containers. In order to access a container, you need to create a corresponding network resource that provides routing to a container in the pod. Imagine if you have four containers and a user tries to connect: the user needs to be directed to a container in the pod; and the network service controls the list and location of containers, as containers are automatically restarted or killed and recreated.

3. Tying it all together

Pushing to the Container Registry

Now let’s confirm that we still have our Docker container that we just created:

```
$docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
dockerdemo          test1               5a6307ac6e7e        11 days ago         79.9MB
```

Show more

Now we'll connect and push this Docker image to a container repository, because when we deploy to Kubernetes we'll need to refer to this container registry from which to pull the container image. In this instance, we'll use the Container Registry as an example.

First, determine the Container Registry information:

```
ibmcloud cr namespace-add getnode
Adding namespace 'getnode'...

Successfully added namespace 'getnode'

OK
$ ibmcloud cr info

Container Registry                  us.icr.io
Container Registry API endpoint     https://us.icr.io/api

IBM Cloud Container Registry is adopting new icr.io domain names to align with the rebranding of IBM Cloud :

OK
```

Show more

Next we will log in to the Container Registry:

```
ibmcloud cr login
Logging in to 'registry.ng.bluemix.net'...
```

```
Logged in to 'registry.ng.bluemix.net'.

IBM Cloud Container Registry is adopting new icr.io domain names to align with the rebranding of IBM Cloud :

Logging in to 'us.icr.io'...
Logged in to 'us.icr.io'.

IBM Cloud Container Registry is adopting new icr.io domain names to align with the rebranding of IBM Cloud :
```

Show more

Now we can push our container into the container repository. First, we tag the image locally with the repository and tag metadata that we want the container to hold in the new repository, then we push it to the new repository:

```
docker tag dockerdemo:test1  us.icr.io/getnode/dockerdemo:test1
docker push us.icr.io/getnode/dockerdemo:test1
The push refers to repository [us.icr.io/getnode/dockerdemo]
ef27c95b3262: Pushed
52e3122c6402: Pushed
76de4fb8c77b: Pushed
2d0f88d2dd7b: Pushed
d8b9f4ebf971: Pushed
4a38d89e6259: Pushed
d9ff549177a9: Pushed
test1: digest: sha256:515cf2d0be96071f099f3c36c839039551ece4d91bc23e0d83d41ad68c57cafb size: 178
```

Show more

Prerequisites for setup

This application uses Cloudant DB as a resource, thus a Kubernetes instance and a Cloudant db resource will need to be configured for when you deploy; this can be done by following instructions here:

- Create a Kubernetes cluster
- Create Cloudant DB

Deploy to Kubernetes

When you’re ready to deploy to Kubernetes, your yaml file will define the following:

- The container and Docker repository information to use to pull the container(s)
- The network services to create
- Attributes to control failure detection and are self-healing
- The Docker registry information from which to pull our container
- A Cloudant url secret describing the Cloudant DB resource this application is using

Now let’s look at our original Cloud Foundry application that is now a Docker container. Our yaml file to define the deployment to Kubernetes should be written as:

```
# Update <REGISTRY> <NAMESPACE> values before use
apiVersion: apps/v1
kind: Deployment
metadata:
  name: get-started-node
  labels:
    app: get-started-node
spec:
  replicas: 1
  selector:
    matchLabels:
      app: get-started-node
  template:
    metadata:
```

Show more

With this file, we can now deploy our containerized application to Kubernetes as seen here in our example:

```
kubectl create -f deployment.yaml
deployment.apps/get-started-node created

kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
get-started-node-7b45c6cf8b-7zbnd   1/1     Running   0          33s
```

Show more

Now if we tell Kubernetes to describe the pod, we can see the information that we specified in the deployment yaml in the form of the Container Registry, Cloudant secret, and port number:

```
kubectl describe pod get-started-node-7b45c6cf8b-7zbnd
Name:               get-started-node-7b45c6cf8b-7zbnd
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               10.76.215.122/10.76.215.122
Start Time:         Thu, 11 Apr 2019 23:48:25 -0700
Labels:             app=get-started-node
                    pod-template-hash=7b45c6cf8b
Annotations:        kubernetes.io/psp: ibm-privileged-psp
Status:             Running
IP:                 172.30.151.135
Controlled By:      ReplicaSet/get-started-node-7b45c6cf8b
Containers:
  get-started-node:
```

Show more

Now let’s create a network service to access the container:

```
kubectl expose deployment get-started-node --type NodePort --port 8080 --target-port 8080
service/get-started-node exposed
```

Show more

Examine the service to determine the port that is exposed. We can see that port `31936` will map to port `8080`, so that will be our external port:

```
kubectl get service
NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
get-started-node   NodePort    172.21.182.149   <none>        8080:31936/TCP   6m13s
kubernetes         ClusterIP   172.21.0.1       <none>        443/TCP          7h59m
```

Show more

Now let’s find the public IP of our worker node so that we can verify our app is running:

```
ibmcloud cs workers cftokube

Plugin version 0.2.99 is now available. To update run: ibmcloud plugin update container-service -r Bluemix

OK
ID                                                  Public IP        Private IP      Machine Type   State
kube-hou02-pa23af55b0d7c84af5b7b942ed48b35717-w1   173.193.112.17   10.76.215.122   free           normal
marcs-mbp:kubernetes mvelasc@us.ibm.com$
```

Show more

Our full URL is: http://173.193.112.17:30124

db hello db hello name

Conclusion

We've seen how we can take an application from Cloud Foundry, build a Docker container, and then deploy it to Kubernetes. The whole process required no change to the application, as most of our time was spent customizing the container or Kubernetes environment that encapsulated the container. Starting from Cloud Foundry does not have to entail a great amount of effort to port to a Docker container, and once a Docker container is generated it is a trivial task to extend that to a Kuberenetes environment.

What's next?

So you built a container and deployed it to Kubernetes. Why not try out an alternative to starting out with Docker by going through this tutorial, "Containerization: Starting with Docker and IBM Cloud." Or see if you can walk through our code patterns, "Use a Kubernetes cluster to deploy a Fabric network smart contract onto blockchain" and "Deploy and use a multi-framework deep learning platform on Kubernetes."

Need some more basic Kubernetes tutorials? Then make sure to check out our beginner's Learning Path to Kubernetes.

Deployment Models (1)

Cloud

Products & Services (3)

Technologies (1)

Table of Contents

1 Starting from Cloud Foundry

Cloud Foundry applications

Cloud Foundry buildpacks

How Cloud Foundry is different from Docker

2 On to Kubernetes

What's involved in deploying to Kubernetes?

3 Tying it all together

Pushing to the Container Registry

Prerequisites for setup

Deploy to Kubernetes

Conclusion

What's next?

Resources

Recommended

Tutorial

IBM Blockchain 101: Quick-start guide for developers

May 30, 2019

Code Pattern

Create a banking chatbot

Aug 06, 2019

Code Pattern

Manage microservices traffic using Istio

Jul 27, 2017

Explore

Newsletters

Code patterns

Articles

Tutorials

Open source projects

Videos

Events

Answers

Cities

IBM Developer

English

中文

日本語

Русский

Português

Español

한글

Community Privacy Terms of use Accessibility Cookie Preferences