ZACHARY FLOWER
FREELANCE WEB DEVELOPER AND WRITER

Continuous Deployment for Docker Apps to Kubernetes

Share this Codeship Guide

About the Author.

Zachary Flower is a freelance web developer, writer, and polymath. He has an eye for simplicity and usability, and strives to build products with both the end user and business goals in mind.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

CODESHIP

Share this Codeship Guide

Continuous Deployment for Docker Apps to Kubernetes

What is Kubernetes?

In a nutshell, Kubernetes is an open-source system for automating the management, deployment, and scaling of containerized applications like Docker. It is an incredibly powerful and cool tool which we will have a closer look at in this eBook.

ABOUT THIS EBOOK

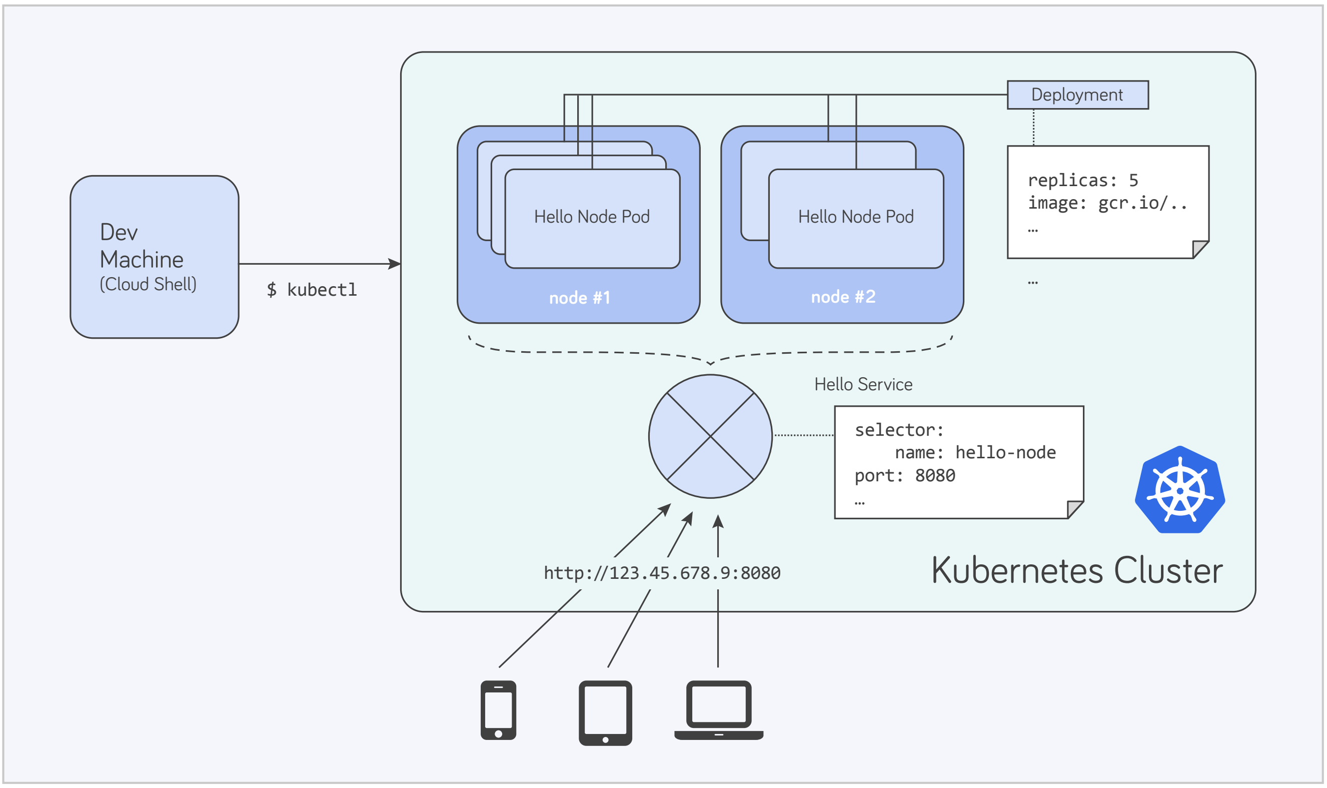
In this book you will learn how to set up Continuous Deployment to Kubernetes for your Docker Apps. In detail, we will look at automating the management, deployment and scaling of your containerized applications.

Introduction

According to the official website, Kubernetes is a system that groups containers into logical units, which makes management of containers across multiple nodes "as simple as managing containers on a single system." Kubernetes essentially acts as a digital datacenter, allowing you to seamlessly manage hundreds of servers across as many nodes without ever having to step foot inside an overly air-conditioned clean room.

Beyond simply managing a complex container architecture, Kubernetes also packs some powerful automated deployment and scaling functionality, giving you the ability to roll out new code and resize your datacenter with minimal configuration.

Because Kubernetes introduces a relatively new way to interact with a cluster of containers, there are likely some new terms that I will mention in this article. These new terms can be very ambiguous when you're just starting out, so it is important to be aware of them early on. To help visualize the definition of each of these terms, I'll borrow a diagram from the Kubernetes documentation.



In a nutshell, here's what this diagram is showing:

- A **Cluster** is a collection of physical and/or virtual machines called Nodes.
- Each **Node** is responsible for running a set of Pods.
- A **Pod** is a group of networked Docker-based containers.

Outside of the parent-child chain are **Deployments** (which I'll get to below) and **Services.** Services are logical sets of Pods with a defined policy by which to access them (read: microservice). A service can span multiple Nodes within a Kubernetes Cluster.

Deployments (uppercase) vs. deployments (lowercase)

When it comes to actually launching containers, Kubernetes provides tools to automatically roll out new code by updating Deployment definitions. It is important to mention here that the word "Deployment" in Kubernetes-speak is really just a fancy (and a bit ambiguous) word for a recipe that describes how containers should be configured and launched. Because this article deals with delivering and launching updated Docker images to Kubernetes using Codeship, there is bound to be some confusion over terminology, so to keep things clear(ish) I'll be using the lowercase "deployment" to refer to the act of delivering product, and the uppercase "Deployment" to refer to the Kubernetes definition of the word.

In Kubernetes, updating a Deployment involves rolling out an updated Docker image to a previously defined Deployment. Kubernetes makes it clear in their documentation that an automated rollout to a Deployment is only triggered when the defined label or container image is updated, which means that simply updating a Docker image in the registry won't trigger a Deployment update unless we specifically tell it to. Don't worry if this seems a bit confusing at first, I'll be going into more detail about how this whole process works later.

I should point out that, even though Deployment updates need to be triggered in a specific way, there is very little risk of downtime in a multi-container environment. Thanks to the way Deployments are built, Kubernetes will ensure that **no downtime is suffered** by bringing down only a fraction of the Pods at a time. While the load won't necessarily be as efficiently distributed during these updates, the consumers of your application won't suffer any outages.

Integrating Codeship with Kubernetes

So, given a functioning Kubernetes Deployment, how do we integrate it into our Codeship workflow? The answer to this question ultimately depends on your Kubernetes host, but because the official documentation uses Google Cloud as an example, this is the platform I'll address. Thankfully for us, Codeship has already built out some Google Cloud integrations into their CI Platform for Docker that we can use to authenticate and deploy new images to Google Cloud.

Before we can do anything, however, we need to create an encrypted environment file using Codeship's Jet CLI tool in order to authenticate to Google Cloud. Codeship already has an excellent tutorial of how to do this, so I won't go over it here, but the environment variables that need to be set are:

- a Google Cloud Key – `GOOGLE_AUTH_JSON`
- a Google Authentication Email – `GOOGLE_AUTH_EMAIL`
- and a Google Project ID – `GOOGLE_PROJECT_ID`

Once we have an encrypted environment file (and have saved our Google Cloud environment variables to `gc.env.encrypted`), we next need to define the Google Cloud service in the `codeship-services.yml` file.

CODESHIP-SERVICES.YML

```
google_cloud_deployment:
  image: zachflower/google-cloud-deployment
  add_docker: true
  encrypted_env_file: gc.env.encrypted
  volumes:
    - ./:/deploy
gcr_dockercfg:
  image: codeship/gcr-dockercfg-generator
  add_docker: true
  encrypted_env_file: gc.env.encrypted
```

Notice that there are two services defined, rather than one. This is because one is for interacting with Google Cloud services (`google_cloud_deployment`), while another is used to enable Docker image push functionality to the Google Cloud Registry (`gcr_dockercfg`). This is only half of the puzzle, however, because although it creates the necessary services for interacting with Google Cloud, it doesn't automatically deploy newly built images or update a Kubernetes Deployment.

Google Container Registry Pushing

Thanks to Codeship's built-in push steps, deploying a Docker image to a remote registry is a pretty painless process. Using the `gcr_dockercfg` service defined above, all we need to do is add a step to the `codeship-steps.yml` file with our Google Container Registry URL as the destination. It's important to remember here that we will be deploying our application image, so be sure to replace the app service name with the name of the service your own application is running on.

CODESHIP-STEPS.YML

```yaml
- service: app
  type: push
  image_name: gcr.io/project-name/app-name
  registry: https://gcr.io
  dockercfg_service: gcr_dockercfg
```

The parameters above should be pretty self-explanatory, but the basic idea is that the `app` image gets pushed up to the Google Container Registry using the previously defined `gcr_dockercfg` service for authentication.

While this step does push updated images to the registry, there is a problem with it as currently defined. Without a set Docker image tag, Codeship will push updated images to the `latest` tag. Now, this isn't a bad thing in and of itself (in fact, it's expected), but in order to trigger automatic Kubernetes Deployment updates, we need to be able to set a **distinct tag** for each push.

To accomplish this, Codeship provides an `image_tag` declaration that allows us to set any tag other than `latest` to push our image up to. Codeship has a nice list of variables that can be used for this declaration; however, to keep things simple let's use the current build's Unix timestamp because it is relatively unique and repeatable. With the new `image_tag` declaration, the previous step should now look like this:

CODE

```yaml
- service: app
  type: push
  image_name: gcr.io/project-name/app-name
  image_tag: "{{ .Timestamp }}"
  registry: https://gcr.io
  dockercfg_service: gcr_dockercfg
```

Now, when we push up our app image to the Google Container Registry, it will be tagged with the current build's Unix timestamp.

Updating Kubernetes Deployments

Once our push step is defined, we need to tell Kubernetes to update the appropriate Deployment to roll out the new image. This is where the previously defined `google_cloud_deployment` service comes into play. Thanks to this service, we are able to easily run authenticated commands against Google Cloud Platform without any additional overhead, which means that manipulating our Kubernetes platform from within Codeship is no different than working with it directly.

Before we set up the Codeship step, though, let's take a look at how updating a Kubernetes Deployment actually works. According to the Kubernetes documentation (and as touched upon above), triggering a Deployment update is as simple as updating the Deployment's defined label or container image. For now, let's assume that we already have a defined Deployment for an Nginx server as per the documentation. All we have to do to roll out an updated Docker image to the Deployment is to change the defined image using the kubectl command like so:

CODE

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

Because we went through the work of tagging our image pushes above, this type of update will be relatively easy for us to set up. But, this is just a command – It doesn't show us how to actually update Deployments from Codeship. All it takes to accomplish this is a small script to run the few necessary commands to authenticate to the Google Cloud Platform and trigger a Kubernetes Deployment update.

Thankfully, due to the hard work put in by Codeship already, the script we need to write consists of only a small handful of commands:

KUBERNETES-DEPLOY.SH

```
#!/bin/bash

set -e

# authenticate to google cloud
codeship_google authenticate

# set compute zone
gcloud config set compute/zone us-central1-a

# set kubernetes cluster
gcloud container clusters get-credentials cluster-name

# update kubernetes Deployment
GOOGLE_APPLICATION_CREDENTIALS=/keyconfig.json kubectl set image deployment/
deployment-name app=gcr.io/project-name/app-name:$CI_TIMESTAMP
```

You can find the complete repository including the `codeship-steps.yml` and the `codeship-services.yml` files here: https://github.com/codeship/codeship-kubernetes-demo

Let's step through the above script really quick. The first important command is the authentication piece. The `google_cloud_deployment` service needs to be authenticated with the Google Cloud Platform before we can run any commands. Since we set up the necessary environment variables already, all it takes to authenticate is to run the `codeship_google authenticate` command at the beginning of our script.

Next, we need to set the compute zone. This example shows `us-central1-a`, but you should change this to suit your needs. The next set of commands is the actual Kubernetes interactions. The first sets the Kubernetes cluster that we need to interact with, while the second is the actual Deployment update command. As you can see, it's not very different from the example provided by Kubernetes itself. It's important to note here that Codeship provides an environment variable of the current build's timestamp, which allows us to correlate the Kubernetes command with the registry push step above.

Now that we have our deployment script set up (I've saved mine to the root of my project as `deploy.sh`), all we have left to do is to add a step to the `codeship-steps.yml` file that calls it:

CODESHIP-STEPS.YML

```
- service: google_cloud_deployment
  command: /deploy/deploy.sh
```

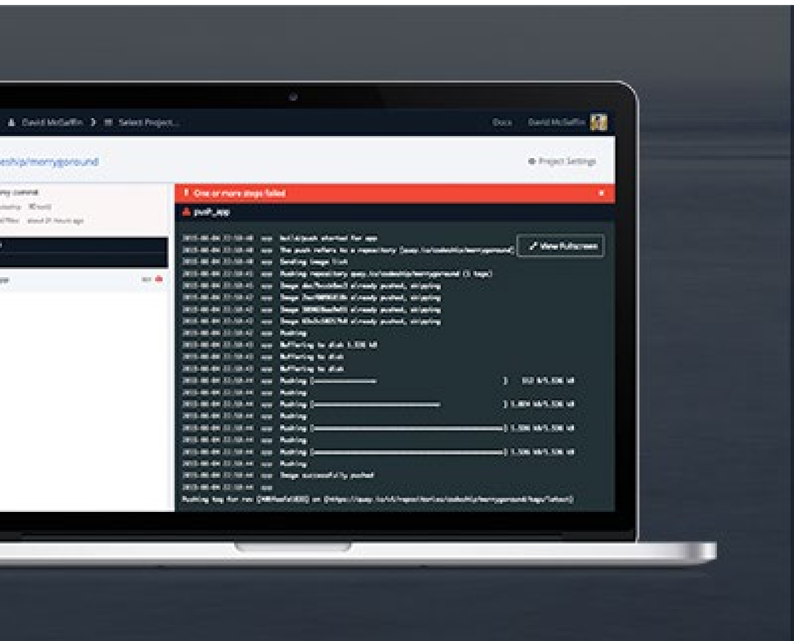
CODESHIP

Share this Codeship Guide

Conclusion

Fortunately for us, most of the heavy lifting for this integration has been done by Codeship already, which means that interacting with the Google Cloud Platform during our CI/CD process is **as simple as running a command.**

Thanks to the incredible flexibility of Codeship, interacting with any cloud platform that we choose is an incredibly straightforward process. Because we are only limited by the capabilities of Docker itself, our deployment workflows are completely customizable, so we can get our process just right.



Share this Codeship Guide

More Codeship Resources.

CODESHIP

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs.
Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.

- Works out of the box
- Preinstalled CI dependencies
- Optimized hosted infrastructure
- Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.

- Customizability & Full Autonomy
- Local CLI tool
- Dedicated single-tenant instances
- Deploy anywhere

LEARN MORE