Kubernetes for Full-Stack Developers

Compiled by **Jamon Camisso, Hanif Jetha,** and **Kathleen Juell**

ISBN 978-0-9997730-3-1

Kubernetes for Full-Stack Developers

Compiled by Jamon Camisso, Hanif Jetha, Katherine Juell

DigitalOcean, New York City, New York, USA

2020-01

Kubernetes for Full-Stack Developers

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Preface - Getting Started with this Book

We recommend that you begin with a set of three or more clean, new servers to start learning about Kubernetes. You can also run Kubernetes locally in a development environment using a tool like Minikube. Finally, you can choose to use a managed Kubernetes solution as long as you have administrator access to the cluster. The examples in this book will work with any system running Ubuntu 18.04 and should also work on Debian 9 and 10 systems, from a laptop to a remote server running in a cloud provider's environment.

Chapter 2 of this book goes into detail about how to create a Kubernetes cluster from three new servers. It will be helpful to prepare in advance and ensure that you can connect to the servers that you will use to create a cluster. To connect to your servers with a terminal, use one of these guides based on your computer's operating system.

- Linux and macOS users: How to Connect to Droplets with SSH
- Windows users: If you have Bash on Windows or Windows Subsystem for Linux, you can use the guide above. Otherwise you can use PuTTY on Windows to connect to your Ubuntu server.

Once you have connected to your servers, everything should be ready for you to start following along with the examples in this book.

Introduction

Cloud computing is a paradigm shift away from traditional self-managed infrastructure to highly-available, on-demand, and scalable computing resources. Instead of building physical servers and managing hardware for individual applications or customers, cloud computing relies on managing computing capacity programmatically on public or private clouds. This approach to running applications means that resources can be provisioned immediately when they are needed, and only with the resources relevant for an application.

Kubernetes is a tool that is designed to manage cloud infrastructure and the applications running on it. It reduces the complexity of coordinating many physical and virtual machines, networks, and environments so that developers can focus on building and deploying their applications. Kubernetes also allows system administrators to focus on application delivery, availability, and scalability instead of managing individual servers and networks. Many applications can run simultaneously on a cloud that is managed with Kubernetes, and each application will only use the memory, CPU, and network resources that Kubernetes allocates to them.

This approach of using Kubernetes to programmatically manage and deploy applications on a cloud requires some familiarity with containers and building applications that can take advantage of the resilience and scalability that a tool like Kubernetes offers. The goal of this book is to familiarize you with Kubernetes, containers, microservices architectures, and managing applications on a Kubernetes cluster.

Introduction to this book

This book is designed to help newcomers and experienced users alike learn about Kubernetes. Its chapters are designed to introduce core Kubernetes concepts, and to build on them to a level where running an application on a production cluster is a familiar, repeatable, and automated process.

From there, more advanced topics are introduced, like how to manage a Kubernetes cluster itself. There are numerous tools, networking configurations, and processes that can be used to deploy, monitor, and run a cluster. This book will examine each topic in turn so that anyone who follows along will be able to build, manage, and monitor their own cluster.

This book is based on the Kubernetes for Full-Stack Developers curriculum found on DigitalOcean Community. It is structured around a few central topics:

1. Learning Kubernetes core concepts
2. Modernizing applications to work with containers
3. Containerizing applications
4. Deploying applications to Kubernetes
5. Managing cluster operations

You should not feel obliged to follow the topics in any particular order. If one section is more interesting or relevant to you, explore it and come back to the others later if you prefer. Likewise, if you are already familiar with the concepts and tools in a given section, feel free to skip that one and focus on other topics.

What You'll Learn

In terms of concrete learning outcomes, if you follow along with this book from the beginning, you will be able to:

- Explain how containers, pods, and services are used in a Kubernetes cluster
- Determine if containers are appropriate for running an application
- Describe and compose application components in terms of a microservices architecture
- Run an application in a standalone Docker container
- Modernize an example application to use containers and run on Kubernetes
- Upload and use container images hosted on public or private image registries
- Deploy an application into a Kubernetes cluster manually, using Helm for dependencies
- Monitor an application's health in a Kubernetes cluster
- Monitor internal cluster health using Prometheus, Grafana, and Alertmanager
- Build a Continuous Integration and Deployment (CI/CD) pipeline to work with Kubernetes
- Create a Kubernetes cluster from scratch using Ansible

By the end of this book, you will have created your own Kubernetes cluster and deployed multiple containerized applications to it. Your applications will be designed around a microservices architecture so that you can individually manage and scale each as components in a larger application. You will also be able to set up and customize monitoring for your cluster, and the applications within it. These outcomes are just a small sample of what you can accomplish when using Kubernetes to build and manage cloud native applications.

An Introduction to Kubernetes

Written by Justin Ellingwood

Kubernetes is a powerful open-source system that manages containerized applications in a clustered environment. It is designed to manage distributed applications and services across varied infrastructure.

In this guide, we'll discuss basic Kubernetes concepts. We will talk about its system architecture, the problems it solves, and the model that it uses to handle containerized deployments and scaling.

After reading this guide, you should be familiar with core Kubernetes concepts like the kube-apiserver, Nodes, Pods, Services, Deployments, and Volumes.

Other tutorials in this curriculum explore each of these components and their different use cases in further depth.

Kubernetes is a powerful open-source system, initially developed by Google, for managing containerized applications in a clustered environment. It aims to provide better ways of managing related, distributed components and services across varied infrastructure.

In this guide, we'll discuss some of Kubernetes' basic concepts. We will talk about the architecture of the system, the problems it solves, and the model that it uses to handle containerized deployments and scaling.

What is Kubernetes?

Kubernetes, at its basic level, is a system for running and coordinating containerized applications across a cluster of machines. It is a platform designed to completely manage the life cycle of containerized applications and services using methods that provide predictability, scalability, and high availability.

As a Kubernetes user, you can define how your applications should run and the ways they should be able to interact with other applications or the outside world. You can scale your services up or down, perform graceful rolling updates, and switch traffic between different versions of your applications to test features or rollback problematic deployments. Kubernetes provides interfaces and composable platform primitives that allow you to define and manage your applications with high degrees of flexibility, power, and reliability.

Kubernetes Architecture

To understand how Kubernetes is able to provide these capabilities, it is helpful to get a sense of how it is designed and organized at a high level. Kubernetes can be visualized as a system built in layers, with each higher layer abstracting the complexity found in the lower levels.

At its base, Kubernetes brings together individual physical or virtual machines into a cluster using a shared network to communicate between each server. This cluster is the physical platform where all Kubernetes components, capabilities, and workloads are configured.

The machines in the cluster are each given a role within the Kubernetes ecosystem. One server (or a small group in highly available deployments) functions as the master server. This server acts as a gateway and brain for the cluster by exposing an API for users and clients, health checking other servers, deciding how best to split up and assign work (known as "scheduling"), and orchestrating communication between other components. The master server acts as the primary point of contact with the cluster and is responsible for most of the centralized logic Kubernetes provides.

The other machines in the cluster are designated as nodes: servers responsible for accepting and running workloads using local and external resources. To help with isolation, management, and flexibility, Kubernetes runs applications and services in containers, so each node needs to be equipped with a container runtime (like Docker or rkt). The node receives work instructions from the master server and creates or destroys containers accordingly, adjusting networking rules to route and forward traffic appropriately.

As mentioned above, the applications and services themselves are run on the cluster within containers. The underlying components make sure that the desired state of the applications matches the actual state of the cluster. Users interact with the cluster by communicating with the main API server either directly or with clients and libraries. To start up an application or service, a declarative plan is submitted in JSON or YAML defining what to create and how it should be managed. The master server then takes the plan and figures out how to run it on the infrastructure by examining the requirements and the current state of the system. This group of user-defined applications running according to a specified plan represents Kubernetes' final layer.

Master Server Components

As we described above, the master server acts as the primary control plane for Kubernetes clusters. It serves as the main contact point for administrators and users, and also provides many cluster-wide systems for the relatively unsophisticated worker nodes. Overall, the components on the master server work together to accept user requests, determine the best ways to schedule workload containers, authenticate clients and nodes, adjust cluster-wide networking, and manage scaling and health checking responsibilities.

These components can be installed on a single machine or distributed across multiple servers. We will take a look at each of the individual components associated with master servers in this section.

etcd

One of the fundamental components that Kubernetes needs to function is a globally available configuration store. The etcd project, developed by the team at CoreOS, is a lightweight, distributed key-value store that can be configured to span across multiple nodes.

Kubernetes uses `etcd` to store configuration data that can be accessed by each of the nodes in the cluster. This can be used for service discovery and can help components configure or reconfigure themselves according to up-to-date information. It also helps maintain cluster state with features like leader election and distributed locking. By providing a simple HTTP/JSON API, the interface for setting or retrieving values is very straight forward.

Like most other components in the control plane, `etcd` can be configured on a single master server or, in production scenarios, distributed among a number of machines. The only requirement is that it be network accessible to each of the Kubernetes machines.

kube-apiserver

One of the most important master services is an API server. This is the main management point of the entire cluster as it allows a user to configure Kubernetes' workloads and organizational units. It is also responsible for making sure that the `etcd` store and the service details of deployed containers are in agreement. It acts as the bridge between various components to maintain cluster health and disseminate information and commands.

The API server implements a RESTful interface, which means that many different tools and libraries can readily communicate with it. A client called kubectl is available as a default method of interacting with the Kubernetes cluster from a local computer.

kube-controller-manager

The controller manager is a general service that has many responsibilities. Primarily, it manages different controllers that regulate the state of the cluster, manage workload life cycles, and perform routine tasks. For instance, a replication controller ensures that the number of replicas (identical copies) defined for a pod matches the number currently deployed on the cluster. The details of these operations are written to `etcd`, where the controller manager watches for changes through the API server.

When a change is seen, the controller reads the new information and implements the procedure that fulfills the desired state. This can involve scaling an application up or down, adjusting endpoints, etc.

kube-scheduler

The process that actually assigns workloads to specific nodes in the cluster is the scheduler. This service reads in a workload’s operating requirements, analyzes the current infrastructure environment, and places the work on an acceptable node or nodes.

The scheduler is responsible for tracking available capacity on each host to make sure that workloads are not scheduled in excess of the available resources. The scheduler must know the total capacity as well as the resources already allocated to existing workloads on each server.

cloud-controller-manager

Kubernetes can be deployed in many different environments and can interact with various infrastructure providers to understand and manage the state of resources in the cluster. While Kubernetes works with generic representations of resources like attachable storage and load balancers, it needs a way to map these to the actual resources provided by non-homogeneous cloud providers.

Cloud controller managers act as the glue that allows Kubernetes to interact providers with different capabilities, features, and APIs while maintaining relatively generic constructs internally. This allows Kubernetes to update its state information according to information gathered from the cloud provider, adjust cloud resources as changes are needed in the system, and create and use additional cloud services to satisfy the work requirements submitted to the cluster.

Node Server Components

In Kubernetes, servers that perform work by running containers are known as nodes. Node servers have a few requirements that are necessary for communicating with master components, configuring the container networking, and running the actual workloads assigned to them.

A Container Runtime

The first component that each node must have is a container runtime. Typically, this requirement is satisfied by installing and running Docker, but alternatives like rkt and runc are also available.

The container runtime is responsible for starting and managing containers, applications encapsulated in a relatively isolated but lightweight operating environment. Each unit of work on the cluster is, at its basic level, implemented as one or more containers that must be deployed. The container runtime on each node is the component that finally runs the containers defined in the workloads submitted to the cluster.

kubelet

The main contact point for each node with the cluster group is a small service called kubelet. This service is responsible for relaying information to and from the control plane services, as well as interacting with the `etcd` store to read configuration details or write new values.

The `kubelet` service communicates with the master components to authenticate to the cluster and receive commands and work. Work is received in the form of a manifest which defines the workload and the operating parameters. The `kubelet` process then assumes responsibility for maintaining the state of the work on the node server. It controls the container runtime to launch or destroy containers as needed.

kube-proxy

To manage individual host subnetting and make services available to other components, a small proxy service called kube-proxy is run on each node server. This process forwards requests to the correct containers, can do primitive load balancing, and is generally responsible for making sure the networking environment is predictable and accessible, but isolated where appropriate.

Kubernetes Objects and Workloads

While containers are the underlying mechanism used to deploy applications, Kubernetes uses additional layers of abstraction over the container interface to provide scaling, resiliency, and life cycle management features. Instead of managing containers directly, users define and interact with instances composed of various primitives provided by the Kubernetes object model. We will go over the different types of objects that can be used to define these workloads below.

Pods

A pod is the most basic unit that Kubernetes deals with. Containers themselves are not assigned to hosts. Instead, one or more tightly coupled containers are encapsulated in an object called a pod.

A pod generally represents one or more containers that should be controlled as a single application. Pods consist of containers that operate closely together, share a life cycle, and should always be scheduled on the same node. They are managed entirely as a unit and share their environment, volumes, and IP space. In spite of their containerized implementation, you should generally think of pods as a single, monolithic application to best conceptualize how the cluster will manage the pod's resources and scheduling.

Usually, pods consist of a main container that satisfies the general purpose of the workload and optionally some helper containers that facilitate closely related tasks. These are programs that benefit from being run and managed in their own containers, but are tightly tied to the main application. For example, a pod may have one container running the primary application server and a helper container pulling down files to the shared filesystem when changes are detected in an external repository. Horizontal scaling is generally discouraged on the pod level because there are other higher level objects more suited for the task.

Generally, users should not manage pods themselves, because they do not provide some of the features typically needed in applications (like sophisticated life cycle management and scaling). Instead, users are encouraged to work with higher level objects that use pods or pod templates as base components but implement additional functionality.

Replication Controllers and Replication Sets

Often, when working with Kubernetes, rather than working with single pods, you will instead be managing groups of identical, replicated pods. These are created from pod templates and can be horizontally scaled by controllers known as replication controllers and replication sets.

A replication controller is an object that defines a pod template and control parameters to scale identical replicas of a pod horizontally by increasing or decreasing the number of running copies. This is an easy way to distribute load and increase availability natively within Kubernetes. The replication controller knows how to create new pods as needed because a template that closely resembles a pod definition is embedded within the replication controller configuration.

The replication controller is responsible for ensuring that the number of pods deployed in the cluster matches the number of pods in its configuration. If a pod or underlying host fails, the controller will start new pods to compensate. If the number of replicas in a controller's configuration changes, the controller either starts up or kills containers to match the desired number. Replication controllers can also perform rolling updates to roll over a set of pods to a new version one by one, minimizing the impact on application availability.

Replication sets are an iteration on the replication controller design with greater flexibility in how the controller identifies the pods it is meant to manage. Replication sets are beginning to replace replication controllers because of their greater replica selection capabilities, but they are not able to do rolling updates to cycle backends to a new version like replication controllers can. Instead, replication sets are meant to be used inside of additional, higher level units that provide that functionality.

Like pods, both replication controllers and replication sets are rarely the units you will work with directly. While they build on the pod design to add horizontal scaling and reliability guarantees, they lack some of the fine grained life cycle management capabilities found in more complex objects.

Deployments

Deployments are one of the most common workloads to directly create and manage. Deployments use replication sets as a building block, adding flexible life cycle management functionality to the mix.

While deployments built with replications sets may appear to duplicate the functionality offered by replication controllers, deployments solve many of the pain points that existed in the implementation of rolling updates. When updating applications with replication controllers, users are required to submit a plan for a new replication controller that would replace the current controller. When using replication controllers, tasks like tracking history, recovering from network failures during the update, and rolling back bad changes are either difficult or left as the user's responsibility.

Deployments are a high level object designed to ease the life cycle management of replicated pods. Deployments can be modified easily by changing the configuration and Kubernetes will adjust the replica sets, manage transitions between different application versions, and optionally maintain event history and undo capabilities automatically. Because of these features, deployments will likely be the type of Kubernetes object you work with most frequently.

Stateful Sets

Stateful sets are specialized pod controllers that offer ordering and uniqueness guarantees. Primarily, these are used to have more fine-grained control when you have special requirements related to deployment ordering, persistent data, or stable networking. For instance, stateful sets are often associated with data-oriented applications, like databases, which need access to the same volumes even if rescheduled to a new node.

Stateful sets provide a stable networking identifier by creating a unique, number-based name for each pod that will persist even if the pod needs to be moved to another node. Likewise, persistent storage volumes can be transferred with a pod when rescheduling is necessary. The volumes persist even after the pod has been deleted to prevent accidental data loss.

When deploying or adjusting scale, stateful sets perform operations according to the numbered identifier in their name. This gives greater predictability and control over the order of execution, which can be useful in some cases.

Daemon Sets

Daemon sets are another specialized form of pod controller that run a copy of a pod on each node in the cluster (or a subset, if specified). This is most often useful when deploying pods that help perform maintenance and provide services for the nodes themselves.

For instance, collecting and forwarding logs, aggregating metrics, and running services that increase the capabilities of the node itself are popular candidates for daemon sets. Because daemon sets often provide fundamental services and are needed throughout the fleet, they can bypass pod scheduling restrictions that prevent other controllers from assigning pods to certain hosts. As an example, because of its unique responsibilities, the master server is frequently configured to be unavailable for normal pod scheduling, but daemon sets have the ability to override the restriction on a pod-by-pod basis to make sure essential services are running.

Jobs and Cron Jobs

The workloads we've described so far have all assumed a long-running, service-like life cycle. Kubernetes uses a workload called jobs to provide a more task-based workflow where the running containers are expected to exit successfully after some time once they have completed their work. Jobs are useful if you need to perform one-off or batch processing instead of running a continuous service.

Building on jobs are cron jobs. Like the conventional `cron` daemons on Linux and Unix-like systems that execute scripts on a schedule, cron jobs in Kubernetes provide an interface to run jobs with a scheduling component. Cron jobs can be used to schedule a job to execute in the future or on a regular, reoccurring basis. Kubernetes cron jobs are basically a reimplementation of the classic cron behavior, using the cluster as a platform instead of a single operating system.

Other Kubernetes Components

Beyond the workloads you can run on a cluster, Kubernetes provides a number of other abstractions that help you manage your applications, control networking, and enable persistence. We will discuss a few of the more common examples here.

Services

So far, we have been using the term "service" in the conventional, Unix-like sense: to denote long-running processes, often network connected, capable of responding to requests. However, in Kubernetes, a service is a component that acts as a basic internal load balancer and ambassador for pods. A service groups together logical collections of pods that perform the same function to present them as a single entity.

This allows you to deploy a service that can keep track of and route to all of the backend containers of a particular type. Internal consumers only need to know about the stable endpoint provided by the service. Meanwhile, the service abstraction allows you to scale out or replace the backend work units as necessary. A service's IP address remains stable regardless of changes to the pods it routes to. By deploying a service, you easily gain discoverability and can simplify your container designs.

Any time you need to provide access to one or more pods to another application or to external consumers, you should configure a service. For instance, if you have a set of pods running web servers that should be accessible from the internet, a service will provide the necessary abstraction. Likewise, if your web servers need to store and retrieve data, you would want to configure an internal service to give them access to your database pods.

Although services, by default, are only available using an internally routable IP address, they can be made available outside of the cluster by choosing one of several strategies. The NodePort configuration works by opening a static port on each node's external networking interface. Traffic to the external port will be routed automatically to the appropriate pods using an internal cluster IP service.

Alternatively, the LoadBalancer service type creates an external load balancer to route to the service using a cloud provider's Kubernetes load balancer integration. The cloud controller manager will create the appropriate resource and configure it using the internal service service addresses.

Volumes and Persistent Volumes

Reliably sharing data and guaranteeing its availability between container restarts is a challenge in many containerized environments. Container runtimes often provide some mechanism to attach storage to a container that persists beyond the lifetime of the container, but implementations typically lack flexibility.

To address this, Kubernetes uses its own volumes abstraction that allows data to be shared by all containers within a pod and remain available until the pod is terminated. This means that tightly coupled pods can easily share files without complex external mechanisms. Container failures within the pod will not affect access to the shared files. Once the pod is terminated, the shared volume is destroyed, so it is not a good solution for truly persistent data.

Persistent volumes are a mechanism for abstracting more robust storage that is not tied to the pod life cycle. Instead, they allow administrators to configure storage resources for the cluster that users can request and claim for the pods they are running. Once a pod is done with a persistent volume, the volume's reclamation policy determines whether the volume is kept around until manually deleted or removed along with the data immediately. Persistent data can be used to guard against node-based failures and to allocate greater amounts of storage than is available locally.

Labels and Annotations

A Kubernetes organizational abstraction related to, but outside of the other concepts, is labeling. A label in Kubernetes is a semantic tag that can be attached to Kubernetes objects to mark them as a part of a group. These can then be selected for when targeting different instances for management or routing. For instance, each of the controller-based objects use labels to identify the pods that they should operate on. Services use labels to understand the backend pods they should route requests to.

Labels are given as simple key-value pairs. Each unit can have more than one label, but each unit can only have one entry for each key. Usually, a “name” key is used as a general purpose identifier, but you can additionally classify objects by other criteria like development stage, public accessibility, application version, etc.

Annotations are a similar mechanism that allows you to attach arbitrary key-value information to an object. While labels should be used for semantic information useful to match a pod with selection criteria, annotations are more free-form and can contain less structured data. In general, annotations are a way of adding rich metadata to an object that is not helpful for selection purposes.

Conclusion

Kubernetes is an exciting project that allows users to run scalable, highly available containerized workloads on a highly abstracted platform. While Kubernetes’ architecture and set of internal components can at first seem daunting, their power, flexibility, and robust feature set are unparalleled in the open-source world. By understanding how the basic building blocks fit together, you can begin to design systems that fully leverage the capabilities of the platform to run and manage your workloads at scale.

How To Create a Kubernetes Cluster Using Kubeadm on Ubuntu 18.04

Written by bsder

In this guide, you will set up a Kubernetes cluster from scratch using Ansible and Kubeadm, and then deploy a containerized Nginx application to it. You will be able to use the cluster that you create in this tutorial in subsequent tutorials.

While the first tutorial in this curriculum introduces some of the concepts and terms that you will encounter when running an application in Kubernetes, this tutorial focuses on the steps required to build a working Kubernetes cluster.

This tutorial uses Ansible to automate some of the more repetitive tasks like user creation, dependency installation, and network setup in the cluster. If you would like to create a cluster manually, the tutorial provides a list of resources that includes the official Kubernetes documentation, which you can use instead of Ansible.

By the end of this tutorial you should have a functioning Kubernetes cluster that consists of three Nodes (a master and two worker Nodes). You will also deploy Nginx to the cluster to confirm that everything works as intended.

The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.

Kubernetes is a container orchestration system that manages containers at scale. Initially developed by Google based on its experience running containers in production, Kubernetes is open source and actively developed by a community around the world.

Note: This tutorial uses version 1.14 of Kubernetes, the official supported version at the time of this article's publication. For up-to-date information on the latest version, please see the current release notes in the official Kubernetes documentation.

Kubeadm automates the installation and configuration of Kubernetes components such as the API server, Controller Manager, and Kube DNS. It does not, however, create users or handle the installation of operating-system-level dependencies and their configuration. For these preliminary tasks, it is possible to use a configuration management tool like Ansible or SaltStack. Using these tools makes creating additional clusters or recreating existing clusters much simpler and less error prone.

In this guide, you will set up a Kubernetes cluster from scratch using Ansible and Kubeadm, and then deploy a containerized Nginx application to it.

Goals

Your cluster will include the following physical resources:

- One master node
  The master node (a node in Kubernetes refers to a server) is responsible for managing the state of the cluster. It runs Etcd, which stores cluster data among components that schedule workloads to worker nodes.
- Two worker nodes

Worker nodes are the servers where your workloads (i.e. containerized applications and services) will run. A worker will continue to run your workload once they're assigned to it, even if the master goes down once scheduling is complete. A cluster's capacity can be increased by adding workers.

After completing this guide, you will have a cluster ready to run containerized applications, provided that the servers in the cluster have sufficient CPU and RAM resources for your applications to consume. Almost any traditional Unix application including web applications, databases, daemons, and command line tools can be containerized and made to run on the cluster. The cluster itself will consume around 300-500MB of memory and 10% of CPU on each node.

Once the cluster is set up, you will deploy the web server Nginx to it to ensure that it is running workloads correctly.

Prerequisites

- An SSH key pair on your local Linux/macOS/BSD machine. If you haven't used SSH keys before, you can learn how to set them up by following this explanation of how to set up SSH keys on your local machine.
- Three servers running Ubuntu 18.04 with at least 2GB RAM and 2 vCPUs each. You should be able to SSH into each server as the root user with your SSH key pair.
- Ansible installed on your local machine. If you're running Ubuntu 18.04 as your OS, follow the "Step 1 - Installing Ansible" section in How to Install and Configure Ansible on Ubuntu 18.04 to install

Ansible. For installation instructions on other platforms like macOS or CentOS, follow the official Ansible installation documentation.

- Familiarity with Ansible playbooks. For review, check out Configuration Management 101: Writing Ansible Playbooks.
- Knowledge of how to launch a container from a Docker image. Look at “Step 5 — Running a Docker Container” in How To Install and Use Docker on Ubuntu 18.04 if you need a refresher.

Step 1 — Setting Up the Workspace Directory and Ansible Inventory File

In this section, you will create a directory on your local machine that will serve as your workspace. You will configure Ansible locally so that it can communicate with and execute commands on your remote servers. Once that’s done, you will create a `hosts` file containing inventory information such as the IP addresses of your servers and the groups that each server belongs to.

Out of your three servers, one will be the master with an IP displayed as **`master_ip`**. The other two servers will be workers and will have the IPs **`worker_1_ip`** and **`worker_2_ip`**.

Create a directory named `~/kube-cluster` in the home directory of your local machine and `cd` into it:

```
mkdir ~/kube-cluster
cd ~/kube-cluster
```

This directory will be your workspace for the rest of the tutorial and will contain all of your Ansible playbooks. It will also be the directory inside which you will run all local commands.

Create a file named `~/kube-cluster/hosts` using `nano` or your favorite text editor:

```
nano ~/kube-cluster/hosts
```

Add the following text to the file, which will specify information about the logical structure of your cluster:

~/kube-cluster/hosts

```
[masters]
master ansible_host=master_ip ansible_user=root

[workers]
worker1 ansible_host=worker_1_ip ansible_user=root
worker2 ansible_host=worker_2_ip ansible_user=root

[all:vars]
ansible_python_interpreter=/usr/bin/python3
```

You may recall that inventory files in Ansible are used to specify server information such as IP addresses, remote users, and groupings of servers to target as a single unit for executing commands. `~/kube-cluster/hosts` will be your inventory file and you've added two Ansible groups (masters and workers) to it specifying the logical structure of your cluster.

In the masters group, there is a server entry named "master" that lists the master node's IP (**`master_ip`**) and specifies that Ansible should run remote commands as the root user.

Similarly, in the workers group, there are two entries for the worker servers (**`worker_1_ip`** and **`worker_2_ip`**) that also specify the `ansible_user` as root.

The last line of the file tells Ansible to use the remote servers' Python 3 interpreters for its management operations.

Save and close the file after you've added the text.

Having set up the server inventory with groups, let's move on to installing operating system level dependencies and creating configuration settings.

Step 2 — Creating a Non-Root User on All Remote Servers

In this section you will create a non-root user with sudo privileges on all servers so that you can SSH into them manually as an unprivileged user. This can be useful if, for example, you would like to see system information with commands such as `top/htop`, view a list of running containers, or change configuration files owned by root. These operations are routinely performed during the maintenance of a cluster, and using a non-root user for such tasks minimizes the risk of modifying or deleting important files or unintentionally performing other dangerous operations.

Create a file named `~/kube-cluster/initial.yml` in the workspace:

```
nano ~/kube-cluster/initial.yml
```

Next, add the following play to the file to create a non-root user with sudo privileges on all of the servers. A play in Ansible is a collection of steps to be performed that target specific servers and groups. The following play will create a non-root sudo user:

~/kube-cluster/initial.yml

```
- hosts: all
  become: yes
  tasks:
    - name: create the 'ubuntu' user
      user: name=ubuntu append=yes state=present
createhome=yes shell=/bin/bash

    - name: allow 'ubuntu' to have passwordless
sudo
      lineinfile:
        dest: /etc/sudoers
        line: 'ubuntu ALL=(ALL) NOPASSWD: ALL'
        validate: 'visudo -cf %s'

    - name: set up authorized keys for the ubuntu
user
      authorized_key: user=ubuntu key="{{item}}"
      with_file:
        - ~/.ssh/id_rsa.pub
```

Here’s a breakdown of what this playbook does:

- Creates the non-root user `ubuntu`.
- Configures the `sudoers` file to allow the `ubuntu` user to run `sudo` commands without a password prompt.
- Adds the public key in your local machine (usually `~/.ssh/id_rsa.pub`) to the remote `ubuntu` user’s authorized

key list. This will allow you to SSH into each server as the `ubuntu` user.

Save and close the file after you’ve added the text.

Next, execute the playbook by locally running:

```
ansible-playbook -i hosts ~/kube-
cluster/initial.yml
```

The command will complete within two to five minutes. On completion, you will see output similar to the following:

Output

```
PLAY [all] ****

TASK [Gathering Facts] ****
ok: [master]
ok: [worker1]
ok: [worker2]

TASK [create the 'ubuntu' user] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [allow 'ubuntu' user to have passwordless
sudo] ****
changed: [master]
changed: [worker1]
```

```
changed: [worker2]

TASK [set up authorized keys for the ubuntu user]
****
changed: [worker1] => (item=ssh-rsa AAAAB3...)
changed: [worker2] => (item=ssh-rsa AAAAB3...)
changed: [master] => (item=ssh-rsa AAAAB3...)

PLAY RECAP ****
master                     : ok=5    changed=4
unreachable=0    failed=0
worker1                    : ok=5    changed=4
unreachable=0    failed=0
worker2                    : ok=5    changed=4
unreachable=0    failed=0
```

Now that the preliminary setup is complete, you can move on to installing Kubernetes-specific dependencies.

Step 3 — Installing Kubernetetes' Dependencies

In this section, you will install the operating-system-level packages required by Kubernetes with Ubuntu's package manager. These packages are:

- Docker - a container runtime. It is the component that runs your containers. Support for other runtimes such as rkt is under active development in Kubernetes.

- `kubeadm` - a CLI tool that will install and configure the various components of a cluster in a standard way.
- `kubelet` - a system service/program that runs on all nodes and handles node-level operations.
- `kubectl` - a CLI tool used for issuing commands to the cluster through its API Server.

Create a file named `~/kube-cluster/kube-dependencies.yml` in the workspace:

```
nano ~/kube-cluster/kube-dependencies.yml
```

Add the following plays to the file to install these packages to your servers:

~/kube-cluster/kube-dependencies.yml

```
- hosts: all
  become: yes
  tasks:
   - name: install Docker
     apt:
       name: docker.io
       state: present
       update_cache: true

   - name: install APT Transport HTTPS
     apt:
       name: apt-transport-https
       state: present
   - name: add Kubernetes apt-key
     apt_key:
       url:
https://packages.cloud.google.com/apt/doc/apt-
key.gpg
       state: present

   - name: add Kubernetes' APT repository
     apt_repository:
      repo: deb http://apt.kubernetes.io/
kubernetes-xenial main
      state: present
      filename: 'kubernetes'

   - name: install kubelet
     apt:
       name: kubelet=1.14.0-00
       state: present
       update_cache: true

   - name: install kubeadm
     apt:
       name: kubeadm=1.14.0-00
       state: present

- hosts: master
```

```
become: yes
tasks:
 - name: install kubectl
   apt:
     name: kubectl=1.14.0-00
     state: present
     force: yes
```

The first play in the playbook does the following:

- Installs Docker, the container runtime.
- Installs `apt-transport-https`, allowing you to add external HTTPS sources to your APT sources list.
- Adds the Kubernetes APT repository's apt-key for key verification.
- Adds the Kubernetes APT repository to your remote servers' APT sources list.
- Installs `kubelet` and `kubeadm`.

The second play consists of a single task that installs `kubectl` on your master node.

Note: While the Kubernetes documentation recommends you use the latest stable release of Kubernetes for your environment, this tutorial uses a specific version. This will ensure that you can follow the steps successfully, as Kubernetes changes rapidly and the latest version may not work with this tutorial.

Save and close the file when you are finished.

Next, execute the playbook by locally running:

```
ansible-playbook -i hosts ~/kube-cluster/kube-
dependencies.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [all] ****

TASK [Gathering Facts] ****
ok: [worker1]
ok: [worker2]
ok: [master]

TASK [install Docker] ****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [install APT Transport HTTPS] *****
ok: [master]
ok: [worker1]
changed: [worker2]

TASK [add Kubernetes apt-key] *****
changed: [master]
changed: [worker1]
changed: [worker2]
```

```
TASK [add Kubernetes' APT repository] *****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [install kubelet] *****
changed: [master]
changed: [worker1]
changed: [worker2]

TASK [install kubeadm] *****
changed: [master]
changed: [worker1]
changed: [worker2]

PLAY [master] *****

TASK [Gathering Facts] *****
ok: [master]

TASK [install kubectl] ******
ok: [master]

PLAY RECAP ****
master                      : ok=9    changed=5
unreachable=0    failed=0
worker1                     : ok=7    changed=5
```

```
unreachable=0    failed=0
worker2                          : ok=7    changed=5
unreachable=0    failed=0
```

After execution, Docker, `kubeadm`, and `kubelet` will be installed on all of the remote servers. `kubectl` is not a required component and is only needed for executing cluster commands. Installing it only on the master node makes sense in this context, since you will run `kubectl` commands only from the master. Note, however, that `kubectl` commands can be run from any of the worker nodes or from any machine where it can be installed and configured to point to a cluster.

All system dependencies are now installed. Let’s set up the master node and initialize the cluster.

Step 4 — Setting Up the Master Node

In this section, you will set up the master node. Before creating any playbooks, however, it’s worth covering a few concepts such as Pods and Pod Network Plugins, since your cluster will include both.

A pod is an atomic unit that runs one or more containers. These containers share resources such as file volumes and network interfaces in common. Pods are the basic unit of scheduling in Kubernetes: all containers in a pod are guaranteed to run on the same node that the pod is scheduled on.

Each pod has its own IP address, and a pod on one node should be able to access a pod on another node using the pod’s IP. Containers on a single node can communicate easily through a local interface. Communication between pods is more complicated, however, and requires a separate networking component that can transparently route traffic from a pod on one node to a pod on another.

This functionality is provided by pod network plugins. For this cluster, you will use Flannel, a stable and performant option.

Create an Ansible playbook named `master.yml` on your local machine:

```
nano ~/kube-cluster/master.yml
```

Add the following play to the file to initialize the cluster and install Flannel:

~/kube-cluster/master.yml

```
- hosts: master
  become: yes
  tasks:
    - name: initialize the cluster
      shell: kubeadm init --pod-network-
cidr=10.244.0.0/16 >> cluster_initialized.txt
      args:
        chdir: $HOME
        creates: cluster_initialized.txt

    - name: create .kube directory
      become: yes
      become_user: ubuntu
      file:
        path: $HOME/.kube
        state: directory
        mode: 0755

    - name: copy admin.conf to user's kube config
      copy:
        src: /etc/kubernetes/admin.conf
        dest: /home/ubuntu/.kube/config
        remote_src: yes
        owner: ubuntu

    - name: install Pod network
      become: yes
      become_user: ubuntu
      shell: kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/a
70459be0084506e4ec919aa1c114638878db11b/Documentat
ion/kube-flannel.yml >> pod_network_setup.txt
      args:
        chdir: $HOME
        creates: pod_network_setup.txt
```

Here’s a breakdown of this play:

- The first task initializes the cluster by running `kubeadm init`. Passing the argument `--pod-network-cidr=10.244.0.0/16` specifies the private subnet that the pod IPs will be assigned from. Flannel uses the above subnet by default; we’re telling `kubeadm` to use the same subnet.

- The second task creates a `.kube` directory at `/home/ubuntu`. This directory will hold configuration information such as the admin key files, which are required to connect to the cluster, and the cluster's API address.
- The third task copies the `/etc/kubernetes/admin.conf` file that was generated from `kubeadm init` to your non-root user's home directory. This will allow you to use `kubectl` to access the newly-created cluster.
- The last task runs `kubectl apply` to install `Flannel`. `kubectl apply -f descriptor.[yml|json]` is the syntax for telling `kubectl` to create the objects described in the `descriptor.[yml|json]` file. The `kube-flannel.yml` file contains the descriptions of objects required for setting up `Flannel` in the cluster.

Save and close the file when you are finished.

Execute the playbook locally by running:

```
ansible-playbook -i hosts ~/kube-
cluster/master.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [master] ****

TASK [Gathering Facts] ****
ok: [master]
```

```
TASK [initialize the cluster] ****
changed: [master]

TASK [create .kube directory] ****
changed: [master]

TASK [copy admin.conf to user's kube config] *****
changed: [master]

TASK [install Pod network] *****
changed: [master]

PLAY RECAP ****
master                           : ok=5    changed=4
unreachable=0    failed=0
```

To check the status of the master node, SSH into it with the following command:

```
ssh ubuntu@master_ip
```

Once inside the master node, execute:

```
kubectl get nodes
```

You will now see the following output:

Output

```
NAME      STATUS    ROLES     AGE       VERSION
master    Ready     master    1d        v1.14.0
```

The output states that the `master` node has completed all initialization tasks and is in a `Ready` state from which it can start accepting worker nodes and executing tasks sent to the API Server. You can now add the workers from your local machine.

Step 5 — Setting Up the Worker Nodes

Adding workers to the cluster involves executing a single command on each. This command includes the necessary cluster information, such as the IP address and port of the master's API Server, and a secure token. Only nodes that pass in the secure token will be able join the cluster.

Navigate back to your workspace and create a playbook named `workers.yml`:

```
nano ~/kube-cluster/workers.yml
```

Add the following text to the file to add the workers to the cluster:

~/kube-cluster/workers.yml

```
- hosts: master
  become: yes
  gather_facts: false
  tasks:
    - name: get join command
      shell: kubeadm token create --print-join-
command
      register: join_command_raw

    - name: set join command
      set_fact:
        join_command: "{{
join_command_raw.stdout_lines[0] }}"
```

```
- hosts: workers
  become: yes
  tasks:
    - name: join cluster
      shell: "{{ hostvars['master'].join_command
}} >> node_joined.txt"
      args:
        chdir: $HOME
        creates: node_joined.txt
```

Here's what the playbook does:

- The first play gets the join command that needs to be run on the worker nodes. This command will be in the following format:`kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash>`. Once it gets the actual command with the proper token and hash values, the task sets it as a fact so that the next play will be able to access that info.
- The second play has a single task that runs the join command on all worker nodes. On completion of this task, the two worker nodes will be part of the cluster.

Save and close the file when you are finished.

Execute the playbook by locally running:

```
ansible-playbook -i hosts ~/kube-
cluster/workers.yml
```

On completion, you will see output similar to the following:

Output

```
PLAY [master] ****

TASK [get join command] ****
changed: [master]

TASK [set join command] *****
ok: [master]

PLAY [workers] *****

TASK [Gathering Facts] *****
ok: [worker1]
ok: [worker2]

TASK [join cluster] *****
changed: [worker1]
changed: [worker2]

PLAY RECAP *****
master                    : ok=2    changed=1
unreachable=0    failed=0
worker1                   : ok=2    changed=1
```

```
unreachable=0    failed=0
worker2                          : ok=2    changed=1
unreachable=0    failed=0
```

With the addition of the worker nodes, your cluster is now fully set up and functional, with workers ready to run workloads. Before scheduling applications, let's verify that the cluster is working as intended.

Step 6 — Verifying the Cluster

A cluster can sometimes fail during setup because a node is down or network connectivity between the master and worker is not working correctly. Let's verify the cluster and ensure that the nodes are operating correctly.

You will need to check the current state of the cluster from the master node to ensure that the nodes are ready. If you disconnected from the master node, you can SSH back into it with the following command:

```
ssh ubuntu@master_ip
```

Then execute the following command to get the status of the cluster:

```
kubectl get nodes
```

You will see output similar to the following:

Output

```
NAME       STATUS     ROLES      AGE        VERSION
master     Ready      master     1d         v1.14.0
worker1    Ready      <none>     1d         v1.14.0
worker2    Ready      <none>     1d         v1.14.0
```

If all of your nodes have the value `Ready` for `STATUS`, it means that they're part of the cluster and ready to run workloads.

If, however, a few of the nodes have `NotReady` as the `STATUS`, it could mean that the worker nodes haven't finished their setup yet. Wait for around five to ten minutes before re-running `kubectl get nodes` and inspecting the new output. If a few nodes still have `NotReady` as the status, you might have to verify and re-run the commands in the previous steps.

Now that your cluster is verified successfully, let's schedule an example Nginx application on the cluster.

Step 7 — Running An Application on the Cluster

You can now deploy any containerized application to your cluster. To keep things familiar, let's deploy Nginx using Deployments and Services to see how this application can be deployed to the cluster. You can use the commands below for other containerized applications as well, provided you change the Docker image name and any relevant flags (such as `ports` and `volumes`).

Still within the master node, execute the following command to create a deployment named `nginx`:

```
kubectl create deployment nginx --image=nginx
```

A deployment is a type of Kubernetes object that ensures there's always a specified number of pods running based on a defined template, even if the pod crashes during the cluster's lifetime. The above deployment will create a pod with one container from the Docker registry's Nginx Docker Image.

Next, run the following command to create a service named `nginx` that will expose the app publicly. It will do so through a NodePort, a scheme that will make the pod accessible through an arbitrary port opened on each node of the cluster:

```
kubectl expose deploy nginx --port 80 --target-
port 80 --type NodePort
```

Services are another type of Kubernetes object that expose cluster internal services to clients, both internal and external. They are also capable of load balancing requests to multiple pods, and are an integral component in Kubernetes, frequently interacting with other components.

Run the following command:

```
kubectl get services
```

This will output text similar to the following:

Output

```
NAME         TYPE        CLUSTER-IP
EXTERNAL-IP           PORT(S)             AGE
kubernetes   ClusterIP   10.96.0.1        <none>
443/TCP             1d
nginx        NodePort    10.109.228.209   <none>
80:nginx_port/TCP   40m
```

From the third line of the above output, you can retrieve the port that Nginx is running on. Kubernetes will assign a random port that is greater than `30000` automatically, while ensuring that the port is not already bound by another service.

To test that everything is working, visit `http://worker_1_ip:nginx_port` or `http://worker_2_ip:nginx_port` through a browser on your local machine. You will see Nginx's familiar welcome page.

If you would like to remove the Nginx application, first delete the `nginx` service from the master node:

```
kubectl delete service nginx
```

Run the following to ensure that the service has been deleted:

```
kubectl get services
```

You will see the following output:

Output

```
NAME           TYPE         CLUSTER-IP
EXTERNAL-IP           PORT(S)        AGE
kubernetes     ClusterIP    10.96.0.1          <none>
443/TCP          1d
```

Then delete the deployment:

```
kubectl delete deployment nginx
```

Run the following to confirm that this worked:

```
kubectl get deployments
```

Output

```
No resources found.
```

Conclusion

In this guide, you've successfully set up a Kubernetes cluster on Ubuntu 18.04 using Kubeadm and Ansible for automation.

If you're wondering what to do with the cluster now that it's set up, a good next step would be to get comfortable deploying your own applications and services onto the cluster. Here's a list of links with further information that can guide you in the process:

- Dockerizing applications - lists examples that detail how to containerize applications using Docker.
- Pod Overview - describes in detail how Pods work and their relationship with other Kubernetes objects. Pods are ubiquitous in Kubernetes, so understanding them will facilitate your work.
- Deployments Overview - provides an overview of deployments. It is useful to understand how controllers such as deployments work since they are used frequently in stateless applications for scaling and the automated healing of unhealthy applications.
- Services Overview - covers services, another frequently used object in Kubernetes clusters. Understanding the types of services and the options they have is essential for running both stateless and stateful applications.

Other important concepts that you can look into are Volumes, Ingresses and Secrets, all of which come in handy when deploying production applications.

Kubernetes has a lot of functionality and features to offer. The Kubernetes Official Documentation is the best place to learn about concepts, find task-specific guides, and look up API references for various objects.

Webinar Series: A Closer Look at Kubernetes

Written by Janakiram MSV

In this webinar based tutorial, you will learn how Kubernetes primitives work together as you deploy a Pod in Kubernetes, expose it a Service, and scale it through a Replication Controller.

This article supplements a webinar series on deploying and managing containerized workloads in the cloud. The series covers the essentials of containers, including managing container lifecycles, deploying multi-container applications, scaling workloads, and working with Kubernetes. It also highlights best practices for running stateful applications.

This article supplements the fourth session in the series, A Closer Look at Kubernetes.

Kubernetes is an open source container orchestration tool for managing containerized applications. In the previous tutorial in this series, you configured Kubernetes on DigitalOcean. Now that the cluster is up and running, you can deploy containerized applications on it.

In this tutorial, you will learn how these primitives work together as you deploy a Pod in Kubernetes, expose it as a Service, and scale it through a Replication Controller.

Prerequisites

To complete this tutorial, you should first complete the previous tutorial in this series, Getting Started with Kubernetes.

Step 1 – Understanding Kubernetes Primitives

Kubernetes exposes an API that clients use to create, scale, and terminate applications. Each operation targets one of more objects that Kubernetes manages. These objects form the basic building blocks of Kubernetes. They are the primitives through which you manage containerized applications.

The following is a summary of the key API objects of Kubernetes:

- Clusters: Pool of compute, storage, and network resources.
- Nodes: Host machines running within the cluster.
- Namespaces: Logical partitions of a cluster.
- Pods: Units of deployment.
- Labels and Selectors: Key-Value pairs for identification and service discovery.
- Services: Collection of Pods belonging to the same application.
- Replica Set: Ensures availability and scalability.
- Deployment: Manages application lifecycle.

Let’s look at these in more detail.

The Nodes that run a Kubernetes cluster are also treated as objects. They can be managed like any other API objects of Kubernetes. To enable logical separation of applications, Kubernetes supports creation of Namespaces. For example, an organization may logically partition a Kubernetes cluster for running development, test, staging, and production environment. Each environment can be placed into a dedicated Namespace that is managed independently. Kubernetes exposes its API through the Master Node.

Though Kubernetes runs Docker containers, these containers cannot be directly deployed. Instead, the applications need to be packaged in a format that Kubernetes understands. This format enables Kubernetes to manage containerized applications efficiently. These applications may contain one or more containers that need to work together.

The fundamental unit of packaging and deployment in Kubernetes is called a Pod. Each Pod may contain one or more containers that need to be managed together. For example, a web server (Nginx) container and a cache (Redis) container can be packaged together as a Pod. Kubernetes treats all the containers that belong to a Pod as a logical unit. Each time a new Pod is created, it results in the creation of all the containers declared in the Pod definition. All the containers in a Pod share the same context such as the IP address, hostname, and storage. They communicate with each other through interprocess communication (IPC) rather than remote calls or REST APIs.

Once the containers are packaged and deployed on Kubernetes, they need to be exposed for internal and external access. Certain containers like databases and caches do not need to be exposed to the outside world. Since APIs and web frontends will be accessed directly by other consumers and end-users, they will have to be exposed to the public. In Kubernetes, containers are exposed internally or externally based on a policy. This mechanism will reduce the risks of exposing sensitive workloads such as databases to the public.

Pods in Kubernetes are exposed through Services. Each Service is declared as an internal or external endpoint along with the port and protocol information. Internal consumers including other Pods and external consumers such as API clients rely on Kubernetes Services for basic interaction. Services support TCP and UDP protocols.

Each object in Kubernetes, such as a Pod or Service, is associated with additional metadata called Labels and Selectors. Labels are key/value pairs attached to a Kubernetes object. These labels uniquely identify one or more API objects. Selectors associate one Kubernetes object with another. For example, a Selector defined in a Service helps Kubernetes find all the Pods with a Label that match the value of the Selector. This association enables dynamic discovery of objects. New objects that are created at runtime with the same Labels will be instantly discovered and associated with the corresponding Selectors. This service discovery mechanism enables efficient dynamic configuration such as scale-in and scale-out operations.

One of the advantages of switching to containers is rapid scaling. Because containers are lightweight when compared to virtual machines, you can scale them in a few seconds. For a highly-available and scalable setup, you will need to deploy multiple instances of your applications and ensure a minimum number of instances of these application are always running. To address this configuration of containerized applications, Kubernetes introduced the concept of Replica Sets, which are designed to run one or more Pods all the time. When multiple instances of Pods need to run in a cluster, they are packaged as Replica Sets. Kubernetes will ensure that the number of Pods defined in the Replica Set are always in a running mode. If a Pod is terminated due to a hardware or configuration issue, the Kubernetes control plane will immediately launch another Pod.

A Deployment object is a combination of Pods and Replica Sets. This primitive brings PaaS-like capabilities to Kubernetes applications. It lets you perform a rolling upgrade of an existing deployment with minimal downtime. Deployments also enable patterns such as canary deploys and blue/green deployments. They handle the essential parts of application lifecycle management (ALM) of containerized applications.

Step 2 – Listing Kubernetes Nodes and Namespaces

Assuming you have followed the steps to set up the Kubernetes Cluster in DigitalOcean, run the following commands to list all the Nodes and available Namespaces:

```
kubectl get nodes
```

Output

```
NAME                   STATUS    ROLES     AGE
VERSION
spc3c97hei-master-1    Ready     master    10m
v1.8.7
spc3c97hei-worker-1    Ready     <none>    4m
v1.8.7
spc3c97hei-worker-2    Ready     <none>    4m
v1.8.7
kubectl get namespaces
```

Output

```
NAME                 STATUS    AGE
default              Active    11m
kube-public          Active    11m
```

```
kube-system          Active    11m
stackpoint-system    Active    4m
```

When no Namespace is specified, `kubectl` targets the default Namespace.

Now let's launch an application.

Step 3– Creating and Deploying a Pod

Kubernetes objects are declared in YAML files and submitted to Kubernetes via the `kubectl` CLI. Let's define a Pod and deploy it.

Create a new YAML file called `Simple-Pod.yaml`:

```
nano Simple-Pod.yaml
```

Add the following code which defines a Pod with one container based on the Nginx web server. It is exposed on port `80` over the TCP protocol. Notice that the definition contains the labels `name` and `env`. We'll use those labels to identify and configure specific Pods.

Simple-Pod.yaml

```
apiVersion: "v1"
kind: Pod
metadata:
  name: web-pod
  labels:
    name: web
    env: dev
spec:
  containers:
    - name: myweb
```

```
      image: nginx
      ports:
        - containerPort: 80
          name: http
          protocol: TCP
```

Run the following command to create a Pod.

```
kubectl create -f Simple-Pod.yaml
```

Output

```
pod "web-pod" created
```

Let’s verify the creation of the Pod.

```
kubectl get pods
```

Output

```
NAME       READY     STATUS     RESTARTS    AGE
web-pod    1/1       Running    0           2m
```

In the next step, we will make this Pod accessible to the public Internet.

Step 4 – Exposing Pods through a Service

Services expose a set of Pods either internally or externally. Let’s define a Service that makes the Nginx pod publicly available. We’ll expose Nginx through a NodePort, a scheme that makes the Pod accessible through an arbitrary port opened on each Node of the cluster.

Create a new file called `Simple-Service.yaml` that contains this code which defines the service for Nginx:

Simple-Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: web-svc
  labels:
    name: web
    env: dev
spec:
  selector:
    name: web
  type: NodePort
  ports:
    - port: 80
      name: http
      targetPort: 80
      protocol: TCP
```

The Service discovers all the Pods in the same Namespace that match the Label with `name: web`. The selector section of the YAML file explicitly defines this association.

We specify that the Service is of type NodePort through type: NodePort declaration.

Then use **kubectl** to submit it to the cluster.

```
kubectl create -f Simple-Service.yml
```

You’ll see this output indicating the service was created successfully:

Output

```
service "web-svc" created
```

Let’s get the port on which the Pod is available.

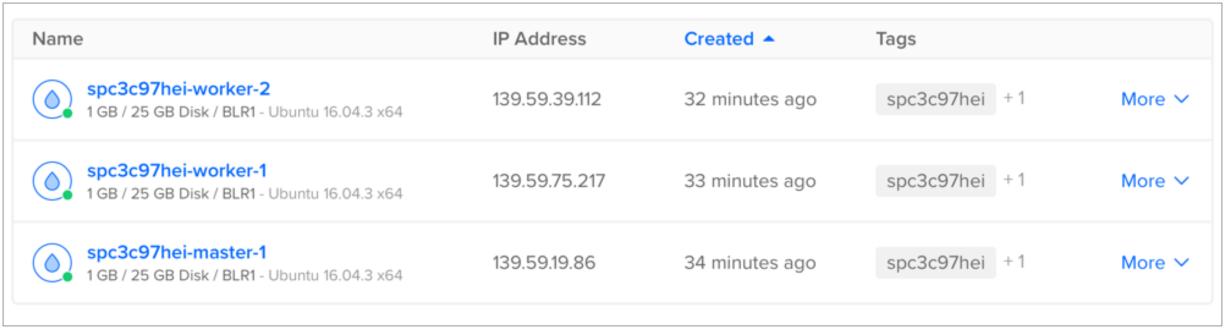
```
kubectl get services
```

Output

```
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)         AGE
kubernetes    ClusterIP   10.3.0.1      <none>
443/TCP         28m
web-svc       NodePort    10.3.0.143    <none>
80:32097/TCP    38s
```

From this output, we see that the Service is available on port `32097`. Let’s try to connect to one of the Worker Nodes.

Use the DigitalOcean Console to get the IP address of one of the Worker Nodes.



Name	IP Address	Created	Tags	
spc3c97hei-worker-2 1 GB / 25 GB Disk / BLR1 - Ubuntu 16.04.3 x64	139.59.39.112	32 minutes ago	spc3c97hei +1	More
spc3c97hei-worker-1 1 GB / 25 GB Disk / BLR1 - Ubuntu 16.04.3 x64	139.59.75.217	33 minutes ago	spc3c97hei +1	More
spc3c97hei-master-1 1 GB / 25 GB Disk / BLR1 - Ubuntu 16.04.3 x64	139.59.19.86	34 minutes ago	spc3c97hei +1	More

The Droplets in the DigitalOcean console associated with your Kubernetes Cluster.

Use the `curl` command to make an HTTP request to one of the nodes on port `31930`.

```
curl http://your_worker_1_ip_address:32097
```

You’ll see the response containing the Nginx default home page:

Output

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to nginx!</title>
...
     Commercial support is available at
     <a href="http://nginx.com/">nginx.com</a>.
</p>
    <p><em>Thank you for using nginx.</em></p>
  </body>
</html>
```

You've defined a Pod and a Service. Now let's look at scaling with Replica Sets.

Step 5 – Scaling Pods through Replica Set

A Replica Set ensures that at least a minimum number of Pods are running in the cluster. Let's delete the current Pod and recreate three Pods through the Replica Set.

First, delete the existing Pod.

```
kubectl delete pod web-pod
```

Output

```
pod "web-pod" deleted
```

Now create a new Replica Set declaration. The definition of the Replica Set is identical to a Pod. The key difference is that it contains the replica element that defines the number of Pods that need to run. Like a Pod, it also contains Labels as metadata that help in service discovery.

Create the file `Simple-RS.yml` and add this code to the file:

Simple-RS.yml

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: web-rs
  labels:
    name: web
    env: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      name: web
  template:
    metadata:
      labels:
        name: web
        env: dev
    spec:
      containers:
      - name: myweb
        image: nginx
          ports:
            - containerPort: 80
              name: http
              protocol: TCP
```

Save and close the file.

Now create the Replica Set:

```
kubectl create -f Simple-RS.yml
```

Output

```
replicaset "web-rs" created
```

Then check the number of Pods:

```
kubectl get pods
```

Output

```
NAME           READY     STATUS    RESTARTS   AGE
web-rs-htb58   1/1       Running   0          38s
web-rs-khtld   1/1       Running   0          38s
web-rs-p5lzg   1/1       Running   0          38s
```

When we access the Service through the NodePort, the request will be sent to one of the Pods managed by the Replica Set.

Let’s test the functionality of a Replica Set by deleting one of the Pods and seeing what happens:

```
kubectl delete pod web-rs-p5lzg
```

Output

```
pod "web-rs-p5lzg" deleted
```

Look at the pods again:

```
kubectl get pods
```

Output

```
NAME             READY     STATUS
RESTARTS    AGE
web-rs-htb58    1/1       Running              0
2m
web-rs-khtld    1/1       Running              0
2m
web-rs-fqh2f    0/1       ContainerCreating    0
2s
web-rs-p5lzg    1/1       Running              0
2m
web-rs-p5lzg    0/1       Terminating          0
2m
```

As soon as the Pod is deleted, Kubernetes has created another one to ensure the desired count is maintained.

Now let’s look at Deployments.

Step 6 – Dealing with Deployments

Though you can deploy containers as Pods and Replica Sets, Deployments make upgrading and patching your application easier. You can upgrade a Pod in-place using a Deployment, which you cannot do with a Replica Set. This makes it possible to roll out a new version of an application with minimal downtime. They bring PaaS-like capabilities to application management.

Delete the existing Replica Set before creating a Deployment. This will also delete the associated Pods:

```
kubectl delete rs web-rs
```

Output

```
replicaset "web-rs" deleted
```

Now define a new Deployment. Create the file `Simple-Deployment.yaml` and add the following code:

Simple-Deployment.yaml

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: web-dep
  labels:
    name: web
    env: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      name: web
  template:
    metadata:
      labels:
        name: web
    spec:
      containers:
      - name: myweb
```

```
        image: nginx
        ports:
        - containerPort: 80
```

Create a deployment and verify the creation.

```
kubectl create -f Simple-Deployment.yml
```

Output

```
deployment "web-dep" created
```

View the deployments:

```
kubectl get deployments
```

Output

```
NAME      DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
web-dep   3         3         3            3
1m
```

Since the Deployment results in the creation of Pods, there will be three Pods running as per the replicas declaration in the YAML file.

```
kubectl get pods
```

Output

```
NAME                           READY     STATUS
RESTARTS   AGE
web-dep-8594f5c765-5wmrb   1/1       Running   0
2m
web-dep-8594f5c765-6cbsr   1/1       Running   0
2m
```

```
web-dep-8594f5c765-sczf8   1/1       Running   0
2m
```

The Service we created earlier will continue to route the requests to the Pods created by the Deployment. That's because of the Labels that contain the same values as the original Pod definition.

Clean up the resources by deleting the Deployment and Service.

```
kubectl delete deployment web-dep
```

Output

```
deployment "web-dep" deleted
kubectl delete service web-svc
```

Output

```
service "web-svc" deleted
```

For more details on Deployments, refer to the Kubernetes documentation.

Conclusion

In this tutorial, you explored the basic building blocks of Kubernetes as you deployed an Nginx web server using a Pod, a Service, a Replica Set, and a Deployment.

In the next part of this series, you will learn how to package, deploy, scale, and manage a multi-container application.

An Introduction to Helm, the Package Manager for Kubernetes

Written by Brian Boucheron

Setting up and running an application on a Kubernetes cluster can involve creating multiple interdependent Kubernetes resources. Each Pod, Service, Deployment, and ReplicaSet requires its own YAML manifest file that must be authored and tested before an application is made available in a cluster.

Helm is a package manager for Kubernetes that allows developers and operators to more easily package, configure, and deploy applications and services onto Kubernetes clusters. Helm packages are called charts, which consist of YAML configuration files and templates that reduce or eliminate the need to write YAML manifests from scratch to deploy an application.

By the end of this tutorial, you should be familiar with Helm charts, and be able to decide if using a chart to deploy an application requires more or less work than writing YAML files directly.

Deploying applications to Kubernetes – the powerful and popular container-orchestration system – can be complex. Setting up a single application can involve creating multiple interdependent Kubernetes resources – such as pods, services, deployments, and replicasets – each requiring you to write a detailed YAML manifest file.

Helm is a package manager for Kubernetes that allows developers and operators to more easily package, configure, and deploy applications and services onto Kubernetes clusters.

Helm is now an official Kubernetes project and is part of the Cloud Native Computing Foundation, a non-profit that supports open source projects in and around the Kubernetes ecosystem.

In this article we will give an overview of Helm and the various abstractions it uses to simplify deploying applications to Kubernetes. If you are new to Kubernetes, it may be helpful to read An Introduction to Kubernetes first to familiarize yourself with the basics concepts.

An Overview of Helm

Most every programming language and operating system has its own package manager to help with the installation and maintenance of software. Helm provides the same basic feature set as many of the package managers you may already be familiar with, such as Debian's `apt`, or Python's `pip`.

Helm can:

- Install software.
- Automatically install software dependencies.
- Upgrade software.
- Configure software deployments.
- Fetch software packages from repositories.

Helm provides this functionality through the following components:

- A command line tool, `helm`, which provides the user interface to all Helm functionality.

- A companion server component, `tiller`, that runs on your Kubernetes cluster, listens for commands from `helm`, and handles the configuration and deployment of software releases on the cluster.
- The Helm packaging format, called charts.
- An [official curated charts repository]() with prepackaged charts for popular open-source software projects.

We'll investigate the charts format in more detail next.

Charts

Helm packages are called charts, and they consist of a few YAML configuration files and some templates that are rendered into Kubernetes manifest files. Here is the basic directory structure of a chart:

Example chart directory

```
package-name/
  charts/
  templates/
  Chart.yaml
  LICENSE
  README.md
  requirements.yaml
  values.yaml
```

These directories and files have the following functions:

- charts/: Manually managed chart dependencies can be placed in this directory, though it is typically better to use `requirements.yaml` to dynamically link dependencies.

- templates/: This directory contains template files that are combined with configuration values (from `values.yaml` and the command line) and rendered into Kubernetes manifests. The templates use the Go programming language's template format.
- Chart.yaml: A YAML file with metadata about the chart, such as chart name and version, maintainer information, a relevant website, and search keywords.
- LICENSE: A plaintext license for the chart.
- README.md: A readme file with information for users of the chart.
- requirements.yaml: A YAML file that lists the chart's dependencies.
- values.yaml: A YAML file of default configuration values for the chart.

The `helm` command can install a chart from a local directory, or from a `.tar.gz` packaged version of this directory structure. These packaged charts can also be automatically downloaded and installed from chart repositories or repos.

We'll look at chart repositories next.

Chart Repositories

A Helm chart repo is a simple HTTP site that serves an `index.yaml` file and `.tar.gz` packaged charts. The `helm` command has subcommands available to help package charts and create the required `index.yaml` file. These files can be served by any web server, object storage service, or a static site host such as GitHub Pages.

Helm comes preconfigured with a default chart repository, referred to as stable. This repo points to a Google Storage bucket at `https://kubernetes-charts.storage.googleapis.com`. The source for the stable repo can be found in the helm/charts Git repository on GitHub.

Alternate repos can be added with the `helm repo add` command. Some popular alternate repositories are:

- The official incubator repo that contains charts that are not yet ready for stable. Instructions for using incubator can be found on the official Helm charts GitHub page.
- Bitnami Helm Charts which provide some charts that aren't covered in the official stable repo.

Whether you're installing a chart you've developed locally, or one from a repo, you'll need to configure it for your particular setup. We'll look into configs next.

Chart Configuration

A chart usually comes with default configuration values in its `values.yaml` file. Some applications may be fully deployable with default values, but you'll typically need to override some of the configuration to meet your needs.

The values that are exposed for configuration are determined by the author of the chart. Some are used to configure Kubernetes primitives, and some may be passed through to the underlying container to configure the application itself.

Here is a snippet of some example values:

values.yaml

```
service:
  type: ClusterIP
  port: 3306
```

These are options to configure a Kubernetes Service resource. You can use `helm inspect values` **`chart-name`** to dump all of the available configuration values for a chart.

These values can be overridden by writing your own YAML file and using it when running `helm install`, or by setting options individually on the command line with the `--set` flag. You only need to specify those values that you want to change from the defaults.

A Helm chart deployed with a particular configuration is called a release. We will talk about releases next.

Releases

During the installation of a chart, Helm combines the chart's templates with the configuration specified by the user and the defaults in `value.yaml`. These are rendered into Kubernetes manifests that are then deployed via the Kubernetes API. This creates a release, a specific configuration and deployment of a particular chart.

This concept of releases is important, because you may want to deploy the same application more than once on a cluster. For instance, you may need multiple MySQL servers with different configurations.

You also will probably want to upgrade different instances of a chart individually. Perhaps one application is ready for an updated MySQL server but another is not. With Helm, you upgrade each release individually.

You might upgrade a release because its chart has been updated, or because you want to update the release's configuration. Either way, each upgrade will create a new revision of a release, and Helm will allow you to easily roll back to previous revisions in case there's an issue.

Creating Charts

If you can't find an existing chart for the software you are deploying, you may want to create your own. Helm can output the scaffold of a chart directory with `helm create` **`chart-name`**. This will create a folder with the files and directories we discussed in the Charts section above.

From there, you'll want to fill out your chart's metadata in `Chart.yaml` and put your Kubernetes manifest files into the `templates` directory. You'll then need to extract relevant configuration variables out of your manifests and into `values.yaml`, then include them back into your manifest templates using the templating system.

The `helm` command has many subcommands available to help you test, package, and serve your charts. For more information, please read the official Helm documentation on developing charts.

Conclusion

In this article we reviewed Helm, the package manager for Kubernetes. We overviewed the Helm architecture and the individual `helm` and `tiller` components, detailed the Helm charts format, and looked at chart repositories. We also looked into how to configure a Helm chart and how configurations and charts are combined and deployed as releases on

Kubernetes clusters. Finally, we touched on the basics of creating a chart when a suitable chart isn't already available.

For more information about Helm, take a look at the official Helm documentation. To find official charts for Helm, check out the official helm/charts Git repository on GitHub.

How To Install Software on Kubernetes Clusters with the Helm Package Manager

Written by Brian Boucheron

The previous Helm tutorial introduced the concept of package management in a Kubernetes cluster. In this hands-on tutorial, we will set up Helm and use it to install, reconfigure, rollback, then delete an instance of the Kubernetes Dashboard application.

By the end of this tutorial, you will have a working Kubernetes dashboard that you can use to administer your cluster. You will also have Helm set up so that you can install any of the supported open source applications in Helm's official chart repository, as well as your own custom Helm charts.

Helm is a package manager for Kubernetes that allows developers and operators to more easily configure and deploy applications on Kubernetes clusters.

In this tutorial we will set up Helm and use it to install, reconfigure, rollback, then delete an instance of the Kubernetes Dashboard application. The dashboard is an official web-based Kubernetes GUI.

For a conceptual overview of Helm and its packaging ecosystem, please read our article An Introduction to Helm.

Prerequisites

For this tutorial you will need:

- A Kubernetes 1.8+ cluster with role-based access control (RBAC) enabled.
- The `kubectl` command-line tool installed on your local machine, configured to connect to your cluster. You can read more about installing `kubectl` [in the official documentation]().

  You can test your connectivity with the following command:

  ```
  kubectl cluster-info
  ```

  If you see no errors, you're connected to the cluster. If you access multiple clusters with `kubectl`, be sure to verify that you've selected the correct cluster context:

  ```
  kubectl config get-contexts
  [secondary_label Output]
  CURRENT   NAME                                    CLUSTER
  AUTHINFO                              NAMESPACE
  *         do-nyc1-k8s-example     do-nyc1-k8s-
  example           do-nyc1-k8s-example-admin
            docker-for-desktop        docker-for-
  desktop-cluster   docker-for-desktop
  ```

  In this example the asterisk (`*`) indicates that we are connected to the `do-nyc1-k8s-example` cluster. To switch clusters run:

  ```
  kubectl config use-context context-name
  ```

When you are connected to the correct cluster, continue to Step 1 to begin installing Helm.

Step 1 — Installing Helm

First we'll install the `helm` command-line utility on our local machine. Helm provides a script that handles the installation process on MacOS, Windows, or Linux.

Change to a writable directory and download the script from Helm's GitHub repository:

```
cd /tmp
curl
https://raw.githubusercontent.com/kubernetes/helm/
master/scripts/get > install-helm.sh
```

Make the script executable with `chmod`:

```
chmod u+x install-helm.sh
```

At this point you can use your favorite text editor to open the script and inspect it to make sure it's safe. When you are satisfied, run it:

```
./install-helm.sh
```

You may be prompted for your password. Provide it and press `ENTER`.

Output

```
helm installed into /usr/local/bin/helm
Run 'helm init' to configure helm.
```

Next we will finish the installation by installing some Helm components on our cluster.

Step 2 — Installing Tiller

Tiller is a companion to the `helm` command that runs on your cluster, receiving commands from `helm` and communicating directly with the Kubernetes API to do the actual work of creating and deleting resources.

To give Tiller the permissions it needs to run on the cluster, we are going to make a Kubernetes `serviceaccount` resource.

Note: We will bind this `serviceaccount` to the cluster-admin cluster role. This will give the `tiller` service superuser access to the cluster and allow it to install all resource types in all namespaces. This is fine for exploring Helm, but you may want a more locked-down configuration for a production Kubernetes cluster.

Please refer to the official Helm RBAC documentation for more information on setting up different RBAC scenarios for Tiller.

Create the tiller `serviceaccount`:

```
kubectl -n kube-system create serviceaccount
tiller
```

Next, bind the tiller `serviceaccount` to the cluster-admin role:

```
kubectl create clusterrolebinding tiller --
clusterrole cluster-admin --serviceaccount=kube-
system:tiller
```

Now we can run `helm init`, which installs Tiller on our cluster, along with some local housekeeping tasks such as downloading the stable repo details:

```
helm init --service-account tiller
```

Output

```
. . .

Tiller (the Helm server-side component) has been
installed into your Kubernetes Cluster.
```

```
Please note: by default, Tiller is deployed with
an insecure 'allow unauthenticated users' policy.
For more information on securing your installation
see: https://docs.helm.sh/using_helm/#securing-
your-helm-installation
Happy Helming!
```

To verify that Tiller is running, list the pods in thekube-system namespace:

```
kubectl get pods --namespace kube-system
```

Output

```
NAME                                    READY
STATUS     RESTARTS    AGE
. . .
kube-dns-64f766c69c-rm9tz               3/3
Running    0           22m
kube-proxy-worker-5884                  1/1
Running    1           21m
kube-proxy-worker-5885                  1/1
Running    1           21m
kubernetes-dashboard-7dd4fc69c8-c4gwk   1/1
Running    0           22m
tiller-deploy-5c688d5f9b-lccsk          1/1
Running    0           40s
```

The Tiller pod name begins with the prefix `tiller-deploy-`.

Now that we’ve installed both Helm components, we’re ready to use `helm` to install our first application.

Step 3 — Installing a Helm Chart

Helm software packages are called charts. Helm comes preconfigured with a curated chart repository called stable. You can browse the available charts in their GitHub repo. We are going to install the Kubernetes Dashboard as an example.

Use `helm` to install the `kubernetes-dashboard` package from the `stable` repo:

```
helm install stable/kubernetes-dashboard --name
dashboard-demo
```

Output

```
NAME:   dashboard-demo
LAST DEPLOYED: Wed Aug  8 20:11:07 2018
NAMESPACE: default
STATUS: DEPLOYED


. . .
```

Notice the `NAME` line, highlighted in the above example output. In this case we specified the name `dashboard-demo`. This is the name of our release. A Helm release is a single deployment of one chart with a specific configuration. You can deploy multiple releases of the same chart with, each with its own configuration.

If you don't specify your own release name using `--name`, Helm will create a random name for you.

We can ask Helm for a list of releases on this cluster:

```
helm list
```

Output

```
NAME                REVISION    UPDATED
STATUS      CHART                           NAMESPACE
dashboard-demo      1           Wed Aug  8 20:11:11
2018    DEPLOYED    kubernetes-dashboard-0.7.1
default
```

We can now use `kubectl` to verify that a new service has been deployed on the cluster:

```
kubectl get services
```

Output

```
NAME                                       TYPE
CLUSTER-IP      EXTERNAL-IP    PORT(S)    AGE
dashboard-demo-kubernetes-dashboard       ClusterIP
10.32.104.73    <none>         443/TCP    51s
kubernetes                                 ClusterIP
10.32.0.1       <none>         443/TCP    34m
```

Notice that by default the service name corresponding to our release is a combination of the Helm release name and the chart name.

Now that we've deployed the application, let's use Helm to change its configuration and update the deployment.

Step 4 — Updating a Release

The `helm upgrade` command can be used to upgrade a release with a new or updated chart, or update the it's configuration options.

We're going to make a simple change to our `dashboard-demo` release to demonstrate the update and rollback process: we'll update the name of the dashboard service to just `dashboard`, instead of `dashboard-demo-kubernetes-dashboard`.

The `kubernetes-dashboard` chart provides a `fullnameOverride` configuration option to control the service name. Let's run `helm upgrade` with this option set:

```
helm upgrade dashboard-demo stable/kubernetes-
dashboard --set fullnameOverride="dashboard"
```

You'll see output similar to the initial `helm install` step.

Check if your Kubernetes services reflect the updated values:

```
kubectl get services
```

Output

```
NAME                    TYPE        CLUSTER-IP
EXTERNAL-IP   PORT(S)   AGE
kubernetes              ClusterIP   10.32.0.1
<none>        443/TCP   36m
dashboard               ClusterIP   10.32.198.148
<none>        443/TCP   40s
```

Our service name has been updated to the new value.

Note: At this point you may want to actually load the Kubernetes Dashboard in your browser and check it out. To do so, first run the following command:

```
kubectl proxy
```

This creates a proxy that lets you access remote cluster resources from your local computer. Based on the previous instructions your dashboard service is named `kubernetes-dashboard` and it's running in the `default` namespace. You may now access the dashboard at the following url:

```
http://localhost:8001/api/v1/namespaces/default/se
rvices/https:dashboard:/proxy/
```

If necessary, substitute your own service name and namespace for the highlighted portions. Instructions for actually using the dashboard are out of scope for this tutorial, but you can read the official Kubernetes Dashboard docs for more information.

Next we'll look at Helm's ability to roll back releases.

Step 5 — Rolling Back a Release

When we updated our **`dashboard-demo`** release in the previous step, we created a second revision of the release. Helm retains all the details of previous releases in case you need to roll back to a prior configuration or chart.

Use `helm list` to inspect the release again:

```
helm list
```

Output

```
NAME            REVISION    UPDATED
STATUS      CHART                               NAMESPACE
dashboard-demo  2         Wed Aug  8 20:13:15 2018
DEPLOYED    kubernetes-dashboard-0.7.1  default
```

The `REVISION` column tells us that this is now the second revision.

Use `helm rollback` to roll back to the first revision:

```
helm rollback dashboard-demo 1
```

You should see the following output, indicating that the rollback succeeded:

Output

```
Rollback was a success! Happy Helming!
```

At this point, if you run `kubectl get services` again, you will notice that the service name has changed back to its previous value. Helm has re-deployed the application with revision 1's configuration.

Next we'll look into deleting releases with Helm.

Step 6 — Deleting a Release

Helm releases can be deleted with the `helm delete` command:

```
helm delete dashboard-demo
```

Output

```
release "dashboard-demo" deleted
```

Though the release has been deleted and the dashboard application is no longer running, Helm saves all the revision information in case you want to re-deploy the release. If you tried to `helm install` a new `dashboard-demo` release right now, you'd get an error:

```
Error: a release named dashboard-demo already
exists.
```

If you use the `--deleted` flag to list your deleted releases, you'll see that the release is still around:

```
helm list --deleted
```

Output

```
NAME                REVISION    UPDATED
STATUS   CHART                              NAMESPACE
dashboard-demo   3           Wed Aug  8 20:15:21
2018     DELETED kubernetes-dashboard-0.7.1
default
```

To really delete the release and purge all old revisions, use the `--purge` flag with the `helm delete` command:

```
helm delete dashboard-demo --purge
```

Now the release has been truly deleted, and you can reuse the release name.

Conclusion

In this tutorial we installed the `helm` command-line tool and its `tiller` companion service. We also explored installing, upgrading, rolling back, and deleting Helm charts and releases.

For more information about Helm and Helm charts, please see the official Helm documentation.

Architecting Applications for Kubernetes

Written by Justin Ellingwood

How you architect and design your applications will determine how you build and deploy them to Kubernetes. One design method that works well for applications that run on Kubernetes is called The Twelve-Factor App. It is a useful framework for building applications that will run on Kubernetes. Some of its core principles include separating code from configuration, making applications stateless, ensuring app processes are disposable (can be started and stopped with no side effects), and facilitating easy scaling. This tutorial will guide you through designing, scaling, and containerizing your applications using Twelve-Factor as a framework.

Designing and running applications with scalability, portability, and robustness in mind can be challenging, especially as system complexity grows. The architecture of an application or system significantly impacts how it must be run, what it expects from its environment, and how closely coupled it is to related components. Following certain patterns during the design phase and adhering to certain operational practices can help counter some of the most common problems that applications face when running in highly distributed environments.

While software design patterns and development methodologies can produce applications with the right scaling characteristics, the infrastructure and environment influence the deployed system's operation. Technologies like Docker and Kubernetes help teams package software and then distribute, deploy, and scale on platforms of distributed computers. Learning how to best harness the power of these tools can help you manage applications with greater flexibility, control, and responsiveness.

In this guide, we will discuss some of the principles and patterns you may wish to adopt to help you scale and manage your workloads on Kubernetes. While Kubernetes can run many types of workloads, choices you make can affect the ease of operation and the possibilities available on deployment. How you architect and build your applications, package your services within containers, and configure life cycle management and behavior within Kubernetes can each influence your experience.

Designing for Application Scalability

When producing software, many requirements affect the patterns and architecture you choose to employ. With Kubernetes, one of the most important factors is the ability to scale horizontally, adjusting the number of identical copies of your application to distribute load and increase availability. This is an alternative to vertical scaling, which attempts to manipulate the same factors by deploying on machines with greater or fewer resources.

In particular, microservices is a software design pattern that works well for scalable deployments on clusters. Developers create small, composable applications that communicate over the network through well-defined REST APIs instead of larger compound programs that communicate through through internal programming mechanisms. Decomposing monolithic applications into discrete single-purpose components makes it possible to scale each function independently. Much of the complexity and composition that would normally exist at the application level is transferred to the operational realm where it can be managed by platforms like Kubernetes.

Beyond specific software patterns, cloud native applications are designed with a few additional considerations in mind. Cloud native applications are programs that follow a microservices architecture pattern with built-in resiliency, observability, and administrative features to adapt to the environment provided by clustered platforms in the cloud.

For example, cloud native applications are constructed with health reporting metrics to enable the platform to manage life cycle events if an instance becomes unhealthy. They produce (and make available for export) robust telemetry data to alert operators to problems and allow them to make informed decisions. Applications are designed to handle regular restarts and failures, changes in backend availability, and high load without corrupting data or becoming unresponsive.

Following 12 Factor Application Philosophy

One popular methodology that can help you focus on the characteristics that matter most when creating cloud-ready web apps is the Twelve-Factor App philosophy. Written to help developers and operations teams understand the core qualities shared by web services designed to run in the cloud, the principles apply very well to software that will live in a clustered environment like Kubernetes. While monolithic applications can benefit from following these recommendations, microservices architectures designed around these principles work particularly well.

A quick summary of the Twelve Factors are:

1. Codebase: Manage all code in version control systems (like Git or Mercurial). The codebase comprehensively dictates what is deployed.
2. Dependencies: Dependencies should be managed entirely and explicitly by the codebase, either vendored (stored with the code) or version pinned in a format that a package manager can install from.
3. Config: Separate configuration parameters from the application and define them in the deployment environment instead of baking them into the application itself.
4. Backing services: Local and remote services are both abstracted as network-accessible resources with connection details set in configuration.
5. Build, release, run: The build stage of your application should be completely separate from your application release and operations processes. The build stage creates a deployment artifact from source code, the release stage combines the artifact and configuration, and the run stage executes the release.
6. Processes: Applications are implemented as processes that should not rely on storing state locally. State should be offloaded to a backing service as described in the fourth factor.
7. Port binding: Applications should natively bind to a port and listen for connections. Routing and request forwarding should be handled externally.
8. Concurrency: Applications should rely on scaling through the process model. Running multiple copies of an application concurrently, potentially across multiple servers, allows scaling without adjusting application code.

9. Disposability: Processes should be able to start quickly and stop gracefully without serious side effects.
10. Dev/prod parity: Your testing, staging, and production environments should match closely and be kept in sync. Differences between environments are opportunities for incompatibilities and untested configurations to appear.
11. Logs: Applications should stream logs to standard output so external services can decide how to best handle them.
12. Admin processes: One-off administration processes should be run against specific releases and shipped with the main process code.

By adhering to the guidelines provided by the Twelve Factors, you can create and run applications using a model that fits the Kubernetes execution environment. The Twelve Factors encourage developers to focus on their application's primary responsibility, consider the operating conditions and interfaces between components, and use inputs, outputs, and standard process management features to run predictably in Kubernetes.

Containerizing Application Components

Kubernetes uses containers to run isolated, packaged applications across its cluster nodes. To run on Kubernetes, your applications must be encapsulated in one or more container images and executed using a container runtime like Docker. While containerizing your components is a requirement for Kubernetes, it also helps reinforce many of the principles from the twelve factor app methodology discussed above, allowing easy scaling and management.

For instance, containers provide isolation between the application environment and the external host system, support a networked, service-oriented approach to inter-application communication, and typically take configuration through environmental variables and expose logs written to standard error and standard out. Containers themselves encourage process-based concurrency and help maintain dev/prod parity by being independently scalable and bundling the process's runtime environment. These characteristics make it possible to package your applications so that they run smoothly on Kubernetes.

Guidelines on Optimizing Containers

The flexibility of container technology allows many different ways of encapsulating an application. However, some methods work better in a Kubernetes environment than others.

Most best practices on containerizing your applications have to do with image building, where you define how your software will be set up and run from within a container. In general, keeping image sizes small and composable provides a number of benefits. Size-optimized images can reduce the time and resources required to start up a new container on a cluster by keeping footprint manageable and reusing existing layers between image updates.

A good first step when creating container images is to do your best to separate your build steps from the final image that will be run in production. Building software generally requires extra tooling, takes additional time, and produces artifacts that might be inconsistent from container to container or unnecessary to the final runtime environment depending on the environment. One way to cleanly separate the build process from the runtime environment is to use Docker multi-stage builds. Multi-stage build configurations allow you to specify one base image to use during your build process and define another to use at runtime. This makes it possible to build software using an image with all of the build tools installed and copy the resulting artifacts to a slim, streamlined image that will be used each time afterwards.

With this type of functionality available, it is usually a good idea to build production images on top of a minimal parent image. If you wish to completely avoid the bloat found in "distro"-style parent layers like `ubuntu:16.04` (which includes a rather complete Ubuntu 16.04 environment), you could build your images with `scratch` — Docker's most minimal base image — as the parent. However, the `scratch` base layer doesn't provide access to many core tools and will often break assumptions about the environment that some software holds. As an alternative, the Alpine Linux `alpine` image has gained popularity by being a solid, minimal base environment that provides a tiny, but fully featured Linux distribution.

For interpreted languages like Python or Ruby, the paradigm shifts slightly since there is no compilation stage and the interpreter must be available to run the code in production. However, since slim images are still ideal, many language-specific, optimized images built on top of Alpine Linux are available on Docker Hub. The benefits of using a smaller image for interpreted languages are similar to those for compiled languages: Kubernetes will be able to quickly pull all of the necessary container images onto new nodes to begin doing meaningful work.

Deciding on Scope for Containers and Pods

While your applications must be containerized to run on a Kubernetes cluster, pods are the smallest unit of abstraction that Kubernetes can manage directly. A pod is a Kubernetes object composed of one or more closely coupled containers. Containers in a pod share a life cycle and are managed together as a single unit. For example, the containers are always scheduled on the same node, are started or stopped in unison, and share resources like filesystems and IP space.

At first, it can be difficult to discover the best way to divide your applications into containers and pods. This makes it important to understand how Kubernetes handles these components and what each layer of abstraction provides for your systems. A few considerations can help you identify some natural points of encapsulation for your application with each of these abstractions.

One way to determine an effective scope for your containers is to look for natural development boundaries. If your systems operate using a microservices architecture, well-designed containers are frequently built to represent discrete units of functionality that can often be used in a variety of contexts. This level of abstraction allows your team to release changes to container images and then deploy this new functionality to any environment where those images are used. Applications can be built by composing individual containers that each fulfill a given function but may not accomplish an entire process alone.

In contrast to the above, pods are usually constructed by thinking about which parts of your system might benefit most from independent management. Since Kubernetes uses pods as its smallest user-facing abstraction, these are the most primitive units that the Kubernetes tools and API can directly interact with and control. You can start, stop, and restart pods, or use higher level objects built upon pods to introduce replication and life cycle management features. Kubernetes doesn’t allow you to manage the containers within a pod independently, so you should not group containers together that might benefit from separate administration.

Because many of Kubernetes’ features and abstractions deal with pods directly, it makes sense to bundle items that should scale together in a single pod and to separate those that should scale independently. For example, separating your web servers from your application servers in different pods allows you to scale each layer independently as needed. However, bundling a web server and a database adaptor into the same pod can make sense if the adaptor provides essential functionality that the web server needs to work properly.

Enhancing Pod Functionality by Bundling Supporting Containers

With this in mind, what types of containers should be bundled in a single pod? Generally, a primary container is responsible for fulfilling the core functions of the pod, but additional containers may be defined that modify or extend the primary container or help it connect to a unique deployment environment.

For instance, in a web server pod, an Nginx container might listen for requests and serve content while an associated container updates static files when a repository changes. It may be tempting to package both of these components within a single container, but there are significant benefits to implementing them as separate containers. Both the web server container and the repository puller can be used independently in different contexts. They can be maintained by different teams and can each be developed to generalize their behavior to work with different companion containers.

Brendan Burns and David Oppenheimer identified three primary patterns for bundling supporting containers in their paper on design patterns for container-based distributed systems. These represent some of the most common use cases for packaging containers together in a pod:

- Sidecar: In this pattern, the secondary container extends and enhances the primary container's core functionality. This pattern involves executing non-standard or utility functions in a separate container. For example, a container that forwards logs or watches for updated configuration values can augment the functionality of a pod without significantly changing its primary focus.
- Ambassador: The ambassador pattern uses a supplemental container to abstract remote resources for the main container. The primary container connects directly to the ambassador container which in turn connects to and abstracts pools of potentially complex external resources, like a distributed Redis cluster. The primary container does not have to know or care about the actual deployment environment to connect to external services.
- Adaptor: The adaptor pattern is used to translate the primary container's data, protocols, or interfaces to align with the standards expected by outside parties. Adaptor containers enable uniform access to centralized services even when the applications they serve may only natively support incompatible interfaces.

Extracting Configuration into ConfigMaps and Secrets

While application configuration can be baked into container images, it’s best to make your components configurable at runtime to support deployment in multiple contexts and allow more flexible administration. To manage runtime configuration parameters, Kubernetes offers two objects called ConfigMaps and Secrets.

ConfigMaps are a mechanism used to store data that can be exposed to pods and other objects at runtime. Data stored within ConfigMaps can be presented as environment variables or mounted as files in the pod. By designing your applications to read from these locations, you can inject the configuration at runtime using ConfigMaps and modify the behavior of your components without having to rebuild the container image.

Secrets are a similar Kubernetes object type used to securely store sensitive data and selectively allow pods and other components access as needed. Secrets are a convenient way of passing sensitive material to applications without storing them as plain text in easily accessible locations in your normal configuration. Functionally, they work in much the same way as ConfigMaps, so applications can consume data from ConfigMaps and Secrets using the same mechanisms.

ConfigMaps and Secrets help you avoid putting configuration directly in Kubernetes object definitions. You can map the configuration key instead of the value, allowing you to update configuration on the fly by modifying the ConfigMap or Secret. This gives you the opportunity to alter the active runtime behavior of pods and other Kubernetes objects without modifying the Kubernetes definitions of the resources.

Implementing Readiness and Liveness Probes

Kubernetes includes a great deal of out-of-the-box functionality for managing component life cycles and ensuring that your applications are always healthy and available. However, to take advantage of these features, Kubernetes has to understand how it should monitor and interpret your application's health. To do so, Kubernetes allows you to define liveness and readiness probes.

Liveness probes allow Kubernetes to determine whether an application within a container is alive and actively running. Kubernetes can periodically run commands within the container to check basic application behavior or can send HTTP or TCP network requests to a designated location to determine if the process is available and able to respond as expected. If a liveness probe fails, Kubernetes restarts the container to attempt to reestablish functionality within the pod.

Readiness probes are a similar tool used to determine whether a pod is ready to serve traffic. Applications within a container may need to perform initialization procedures before they are ready to accept client requests or they may need to reload upon being notified of a new configuration. When a readiness probe fails, instead of restarting the container, Kubernetes stops sending requests to the pod temporarily. This allows the pod to complete its initialization or maintenance routines without impacting the health of the group as a whole.

By combining liveness and readiness probes, you can instruct Kubernetes to automatically restart pods or remove them from backend groups. Configuring your infrastructure to take advantage of these capabilities allows Kubernetes to manage the availability and health of your applications without additional operations work.

Using Deployments to Manage Scale and Availability

Earlier, when discussing some pod design fundamentals, we also mentioned that other Kubernetes objects build on these primitives to provide more advanced functionality. A deployment, one such compound object, is probably the most commonly defined and manipulated Kubernetes object.

Deployments are compound objects that build on other Kubernetes primatives to add additional capabilities. They add life cycle management capabilities to intermediary objects called replicasets, like the ability to perform rolling updates, rollback to earlier versions, and transition between states. These replicasets allow you to define pod templates to spin up and manage multiple copies of a single pod design. This helps you easily scale out your infrastructure, manage availability requirements, and automatically restart pods in the event of failure.

These additional features provide an administrative framework and self-healing capabilities to the relatively simple pod abstraction. While pods are the units that ultimately run the workloads you define, they are not the units that you should usually be provisioning and managing. Instead, think of pods as a building block that can run applications robustly when provisioned through higher-level objects like deployments.

Creating Services and Ingress Rules to Manage Access to Application Layers

Deployments allow you to provision and manage sets of interchangeable pods to scale out your applications and meet user demands. However, routing traffic to the provisioned pods is a separate concern. As pods are swapped out as part of rolling updates, restarted, or moved due to host failures, the network addresses previously associated with the running group will change. Kubernetes services allow you to manage this complexity by maintaining routing information for dynamic pools of pods and controlling access to various layers of your infrastructure.

In Kubernetes, services are specific mechanisms that control how traffic gets routed to sets of pods. Whether forwarding traffic from external clients or managing connections between several internal components, services allow you to control how traffic should flow. Kubernetes will then update and maintain all of the information needed to forward connections to the relevant pods, even as the environment shifts and the networking landscape changes.

Accessing Services Internally

To effectively use services, you first must determine the intended consumers for each group of pods. If your service will only be used by other applications deployed within your Kubernetes cluster, the clusterIP service type allows you to connect to a set of pods using a stable IP address that is only routable from within the cluster. Any object deployed on the cluster can communicate with the group of replicated pods by sending traffic directly to the service's IP address. This is the simplest service type, which works well for internal application layers.

An optional DNS addon enables Kubernetes to provide DNS names for services. This allows pods and other objects to communicate with services by name instead of by IP address. This mechanism does not change service usage significantly, but name-based identifiers can make it simpler to hook up components or define interactions without knowing the service IP address ahead of time.

Exposing Services for Public Consumption

If the interface should be publicly accessible, your best option is usually the load balancer service type. This uses your specific cloud provider's API to provision a load balancer, which serves traffic to the service pods through a publicly exposed IP address. This allows you to route external requests to the pods in your service, offering a controlled network channel to your internal cluster network.

Since the load balancer service type creates a load balancer for every service, it can potentially become expensive to expose Kubernetes services publicly using this method. To help alleviate this, Kubernetes ingress objects can be used to describe how to route different types of requests to different services based on a predetermined set of rules. For instance, requests for "example.com" might go to service A, while requests for "sammytheshark.com" might be routed to service B. Ingress objects provide a way of describing how to logically route a mixed stream of requests to their target services based on predefined patterns.

Ingress rules must be interpreted by an ingress controller — typically some sort of load balance, like Nginx — deployed within the cluster as a pod, which implements the ingress rules and forwards traffic to Kubernetes services accordingly. Currently, the ingress object type is in beta, but there are several working implementations that can be used to minimize the number of external load balancers that cluster owners are required to run.

Using Declarative Syntax to Manage Kubernetes State

Kubernetes offers quite a lot of flexibility in defining and controlling the resources deployed to your cluster. Using tools like `kubectl`, you can imperatively define ad-hoc objects to immediately deploy to your cluster. While this can be useful for quickly deploying resources when learning Kubernetes, there are drawbacks to this approach that make it undesirable for long-term production administration.

One of the major problems with imperative management is that it does not leave any record of the changes you've deployed to your cluster. This makes it difficult or impossible to recover in the event of failures or to track operational changes as they're applied to your systems.

Fortunately, Kubernetes provides an alternative declarative syntax that allows you to fully define resources within text files and then use `kubectl` to apply the configuration or change. Storing these configuration files in a version control repository is a simple way to monitor changes and integrate with the review processes used for other parts of your organization. File-based management also makes it simple to adapt existing patterns to new resources by copying and editing existing definitions. Storing your Kubernetes object definitions in versioned directories allows you to maintain a snapshot of your desired cluster state at each point in time. This can be invaluable during recovery operations, migrations, or when tracking down the root cause of unintended changes introduced to your system.

Conclusion

Managing the infrastructure that will run your applications and learning how to best leverage the features offered by modern orchestrations environments can be daunting. However, many of the benefits offered by systems like Kubernetes and technologies like containers become more clear when your development and operations practices align with the concepts the tooling is built around. Architecting your systems using the patterns Kubernetes excels at and understanding how certain features can alleviate some of the challenges associated with highly complex deployments can help improve your experience running on the platform.

Modernizing Applications for Kubernetes

Written by Hanif Jetha

The previous tutorial explored key ideas and application design techniques to build applications that will run effectively on Kubernetes. This guide will focus on modernizing an existing application to run on Kubernetes. To prepare for migration, there are some important application-level changes to implement that will maximize your app's portability and observability in Kubernetes.

You will learn how to extract configuration data from code and externalize application state using databases and data stores for persistent data. You will also build in health checks and code instrumentation for logging and monitoring, thereby creating an infrastructure to identify errors in your cluster more effectively. After covering application logic, this tutorial examines some best practices for containerizing your app with Docker.

Finally, this guide discusses some core Kubernetes components for managing and scaling your app. Specifically, you will learn how to use Pods, ConfigMaps, Secrets, and Services to deploy and manage a modernized application on Kubernetes.

Modern stateless applications are built and designed to run in software containers like Docker, and be managed by container clusters like Kubernetes. They are developed using Cloud Native and Twelve Factor principles and patterns, to minimize manual intervention and maximize portability and redundancy. Migrating virtual-machine or bare metal-based applications into containers (known as “containerizing”) and deploying them inside of clusters often involves significant shifts in how these apps are built, packaged, and delivered.

Building on Architecting Applications for Kubernetes, in this conceptual guide, we’ll discuss high-level steps for modernizing your applications, with the end goal of running and managing them in a Kubernetes cluster. Although you can run stateful applications like databases on Kubernetes, this guide focuses on migrating and modernizing stateless applications, with persistent data offloaded to an external data store. Kubernetes provides advanced functionality for efficiently managing and scaling stateless applications, and we’ll explore the application and infrastructure changes necessary for running scalable, observable, and portable apps on Kubernetes.

Preparing the Application for Migration

Before containerizing your application or writing Kubernetes Pod and Deployment configuration files, you should implement application-level changes to maximize your app’s portability and observability in Kubernetes. Kubernetes is a highly automated environment that can automatically deploy and restart failing application containers, so it’s important to build in the appropriate application logic to communicate with the container orchestrator and allow it to automatically scale your app as necessary.

Extract Configuration Data

One of the first application-level changes to implement is extracting application configuration from application code. Configuration consists of any information that varies across deployments and environments, like service endpoints, database addresses, credentials, and various parameters and options. For example, if you have two environments, say `staging` and `production`, and each contains a separate database, your application should not have the database endpoint and credentials explicitly declared in the code, but stored in a separate location, either as variables in the running environment, a local file, or external key-value store, from which the values are read into the app.

Hardcoding these parameters into your code poses a security risk as this config data often consists of sensitive information, which you then check in to your version control system. It also increases complexity as you now have to maintain multiple versions of your application, each consisting of the same core application logic, but varying slightly in configuration. As applications and their configuration data grow, hardcoding config into app code quickly becomes unwieldy.

By extracting configuration values from your application code, and instead ingesting them from the running environment or local files, your app becomes a generic, portable package that can be deployed into any environment, provided you supply it with accompanying configuration data. Container software like Docker and cluster software like Kubernetes have been designed around this paradigm, building in features for managing configuration data and injecting it into application containers. These features will be covered in more detail in the Containerizing and Kubernetes sections.

Here's a quick example demonstrating how to externalize two config values `DB_HOST` and `DB_USER` from a simple Python Flask app's code.

We'll make them available in the app's running environment as env vars, from which the app will read them:

hardcoded_config.py

```
from flask import Flask

DB_HOST = 'mydb.mycloud.com'
DB_USER = 'sammy'

app = Flask(__name__)

@app.route('/')
def print_config():
    output = 'DB_HOST: {} -- DB_USER:
{}'.format(DB_HOST, DB_USER)
    return output
```

Running this simple app (consult the Flask Quickstart to learn how) and visiting its web endpoint will display a page containing these two config values.

Now, here's the same example with the config values externalized to the app's running environment:

env_config.py

```
import os

from flask import Flask
```

```
DB_HOST = os.environ.get('APP_DB_HOST')
DB_USER = os.environ.get('APP_DB_USER')

app = Flask(__name__)

@app.route('/')
def print_config():
    output = 'DB_HOST: {} -- DB_USER:
{}'.format(DB_HOST, DB_USER)
    return output
```

Before running the app, we set the necessary config variables in the local environment:

```
export APP_DB_HOST=mydb.mycloud.com
export APP_DB_USER=sammy
flask run
```

The displayed web page should contain the same text as in the first example, but the app's config can now be modified independently of the application code. You can use a similar approach to read in config parameters from a local file.

In the next section we'll discuss moving application state outside of containers.

Offload Application State

Cloud Native applications run in containers, and are dynamically orchestrated by cluster software like Kubernetes or Docker Swarm. A given app or service can be load balanced across multiple replicas, and any individual app container should be able to fail, with minimal or no disruption of service for clients. To enable this horizontal, redundant scaling, applications must be designed in a stateless fashion. This means that they respond to client requests without storing persistent client and application data locally, and at any point in time if the running app container is destroyed or restarted, critical data is not lost.

For example, if you are running an address book application and your app adds, removes and modifies contacts from an address book, the address book data store should be an external database or other data store, and the only data kept in container memory should be short-term in nature, and disposable without critical loss of information. Data that persists across user visits like sessions should also be moved to external data stores like Redis. Wherever possible, you should offload any state from your app to services like managed databases or caches.

For stateful applications that require a persistent data store (like a replicated MySQL database), Kubernetes builds in features for attaching persistent block storage volumes to containers and Pods. To ensure that a Pod can maintain state and access the same persistent volume after a restart, the StatefulSet workload must be used. StatefulSets are ideal for deploying databases and other long-running data stores to Kubernetes.

Stateless containers enable maximum portability and full use of available cloud resources, allowing the Kubernetes scheduler to quickly scale your app up and down and launch Pods wherever resources are available. If you don’t require the stability and ordering guarantees provided by the StatefulSet workload, you should use the Deployment workload to manage and scale and your applications.

To learn more about the design and architecture of stateless, Cloud Native microservices, consult our Kubernetes White Paper.

Implement Health Checks

In the Kubernetes model, the cluster control plane can be relied on to repair a broken application or service. It does this by checking the health of application Pods, and restarting or rescheduling unhealthy or unresponsive containers. By default, if your application container is running, Kubernetes sees your Pod as “healthy.” In many cases this is a reliable indicator for the health of a running application. However, if your application is deadlocked and not performing any meaningful work, the app process and container will continue to run indefinitely, and by default Kubernetes will keep the stalled container alive.

To properly communicate application health to the Kubernetes control plane, you should implement custom application health checks that indicate when an application is both running and ready to receive traffic. The first type of health check is called a readiness probe, and lets Kubernetes know when your application is ready to receive traffic. The second type of check is called a liveness probe, and lets Kubernetes know when your application is healthy and running. The Kubelet Node agent can perform these probes on running Pods using 3 different methods:

- HTTP: The Kubelet probe performs an HTTP GET request against an endpoint (like `/health`), and succeeds if the response status is between 200 and 399
- Container Command: The Kubelet probe executes a command inside of the running container. If the exit code is 0, then the probe succeeds.
- TCP: The Kubelet probe attempts to connect to your container on a specified port. If it can establish a TCP connection, then the probe succeeds.

You should choose the appropriate method depending on the running application(s), programming language, and framework. The readiness and liveness probes can both use the same probe method and perform the same check, but the inclusion of a readiness probe will ensure that the Pod doesn't receive traffic until the probe begins succeeding.

When planning and thinking about containerizing your application and running it on Kubernetes, you should allocate planning time for defining what "healthy" and "ready" mean for your particular application, and development time for implementing and testing the endpoints and/or check commands.

Here's a minimal health endpoint for the Flask example referenced above:

env_config.py

```
. . .
@app.route('/')
def print_config():
    output = 'DB_HOST: {} -- DB_USER:
{}'.format(DB_HOST, DB_USER)
    return output

@app.route('/health')
def return_ok():
    return 'Ok!', 200
```

A Kubernetes liveness probe that checks this path would then look something like this:

pod_spec.yaml

```
. . .
  livenessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 2
```

The `initialDelaySeconds` field specifies that Kubernetes (specifically the Node Kubelet) should probe the `/health` endpoint after waiting 5 seconds, and `periodSeconds` tells the Kubelet to probe `/health` every 2 seconds.

To learn more about liveness and readiness probes, consult the Kubernetes documentation.

Instrument Code for Logging and Monitoring

When running your containerized application in an environment like Kubernetes, it's important to publish telemetry and logging data to monitor and debug your application's performance. Building in features to publish performance metrics like response duration and error rates will help you monitor your application and alert you when your application is unhealthy.

One tool you can use to monitor your services is Prometheus, an open-source systems monitoring and alerting toolkit, hosted by the Cloud Native Computing Foundation (CNCF). Prometheus provides several client libraries for instrumenting your code with various metric types to count events and their durations. For example, if you're using the Flask

Python framework, you can use the Prometheus Python client to add decorators to your request processing functions to track the time spent processing requests. These metrics can then be scraped by Prometheus at an HTTP endpoint like `/metrics`.

A helpful method to use when designing your app's instrumentation is the RED method. It consists of the following three key request metrics:

- Rate: The number of requests received by your application
- Errors: The number of errors emitted by your application
- Duration: The amount of time it takes your application to serve a response

This minimal set of metrics should give you enough data to alert on when your application's performance degrades. Implementing this instrumentation along with the health checks discussed above will allow you to quickly detect and recover from a failing application.

To learn more about signals to measure when monitoring your applications, consult Monitoring Distributed Systems from the Google Site Reliability Engineering book.

In addition to thinking about and designing features for publishing telemetry data, you should also plan how your application will log in a distributed cluster-based environment. You should ideally remove hardcoded configuration references to local log files and log directories, and instead log directly to stdout and stderr. You should treat logs as a continuous event stream, or sequence of time-ordered events. This output stream will then get captured by the container enveloping your application, from which it can be forwarded to a logging layer like the

EFK (Elasticsearch, Fluentd, and Kibana) stack. Kubernetes provides a lot of flexibility in designing your logging architecture, which we'll explore in more detail below.

Build Administration Logic into API

Once your application is containerized and up and running in a cluster environment like Kubernetes, you may no longer have shell access to the container running your app. If you've implemented adequate health checking, logging, and monitoring, you can quickly be alerted on, and debug production issues, but taking action beyond restarting and redeploying containers may be difficult. For quick operational and maintenance fixes like flushing queues or clearing a cache, you should implement the appropriate API endpoints so that you can perform these operations without having to restart containers or `exec` into running containers and execute series of commands. Containers should be treated as immutable objects, and manual administration should be avoided in a production environment. If you must perform one-off administrative tasks, like clearing caches, you should expose this functionality via the API.

Summary

In these sections we've discussed application-level changes you may wish to implement before containerizing your application and moving it to Kubernetes. For a more in-depth walkthrough on building Cloud Native apps, consult Architecting Applications for Kubernetes.

We'll now discuss some considerations to keep in mind when building containers for your apps.

Containerizing Your Application

Now that you've implemented app logic to maximize its portability and observability in a cloud-based environment, it's time to package your app inside of a container. For the purposes of this guide, we'll use Docker containers, but you should use whichever container implementation best suits your production needs.

Explicitly Declare Dependencies

Before creating a Dockerfile for your application, one of the first steps is taking stock of the software and operating system dependencies your application needs to run correctly. Dockerfiles allow you to explicitly version every piece of software installed into the image, and you should take advantage of this feature by explicitly declaring the parent image, software library, and programming language versions.

Avoid `latest` tags and unversioned packages as much as possible, as these can shift, potentially breaking your application. You may wish to create a private registry or private mirror of a public registry to exert more control over image versioning and to prevent upstream changes from unintentionally breaking your image builds.

To learn more about setting up a private image registry, consult Deploy a Registry Server from the Docker official documentation and the Registries section below.

Keep Image Sizes Small

When deploying and pulling container images, large images can significantly slow things down and add to your bandwidth costs.

Packaging a minimal set of tools and application files into an image provides several benefits:

- Reduce image sizes
- Speed up image builds
- Reduce container start lag
- Speed up image transfer times
- Improve security by reducing attack surface

Some steps you can consider when building your images:

- Use a minimal base OS image like `alpine` or build from `scratch` instead of a fully featured OS like `ubuntu`
- Clean up unnecessary files and artifacts after installing software
- Use separate "build" and "runtime" containers to keep production application containers small
- Ignore unnecessary build artifacts and files when copying in large directories

For a full guide on optimizing Docker containers, including many illustrative examples, consult Building Optimized Containers for Kubernetes.

Inject Configuration

Docker provides several helpful features for injecting configuration data into your app's running environment.

One option for doing this is specifying environment variables and their values in the Dockerfile using the `ENV` statement, so that configuration data is built-in to images:

Dockerfile

```
...
ENV MYSQL_USER=my_db_user
...
```

Your app can then parse these values from its running environment and configure its settings appropriately.

You can also pass in environment variables as parameters when starting a container using `docker run` and the `-e` flag:

```
docker run -e MYSQL_USER='my_db_user' IMAGE[:TAG]
```

Finally, you can use an env file, containing a list of environment variables and their values. To do this, create the file and use the `--env-file` parameter to pass it in to the command:

```
docker run --env-file var_list IMAGE[:TAG]
```

If you're modernizing your application to run it using a cluster manager like Kubernetes, you should further externalize your config from the image, and manage configuration using Kubernetes' built-in ConfigMap and Secrets objects. This allows you to separate configuration from image manifests, so that you can manage and version it separately from your application. To learn how to externalize configuration using ConfigMaps and Secrets, consult the ConfigMaps and Secrets section below.

Publish Image to a Registry

Once you've built your application images, to make them available to Kubernetes, you should upload them to a container image registry. Public registries like Docker Hub host the latest Docker images for popular open source projects like Node.js and nginx. Private registries allow you publish your internal application images, making them available to developers and infrastructure, but not the wider world.

You can deploy a private registry using your existing infrastructure (e.g. on top of cloud object storage), or optionally use one of several Docker registry products like Quay.io or paid Docker Hub plans. These registries can integrate with hosted version control services like GitHub so that when a Dockerfile is updated and pushed, the registry service will automatically pull the new Dockerfile, build the container image, and make the updated image available to your services.

To exert more control over the building and testing of your container images and their tagging and publishing, you can implement a continuous integration (CI) pipeline.

Implement a Build Pipeline

Building, testing, publishing and deploying your images into production manually can be error-prone and does not scale well. To manage builds and continuously publish containers containing your latest code changes to your image registry, you should use a build pipeline.

Most build pipelines perform the following core functions:

- Watch source code repositories for changes
- Run smoke and unit tests on modified code
- Build container images containing modified code
- Run further integration tests using built container images
- If tests pass, tag and publish images to registry

- (Optional, in continuous deployment setups) Update Kubernetes Deployments and roll out images to staging/production clusters

There are many paid continuous integration products that have built-in integrations with popular version control services like GitHub and image registries like Docker Hub. An alternative to these products is Jenkins, a free and open-source build automation server that can be configured to perform all of the functions described above. To learn how to set up a Jenkins continuous integration pipeline, consult How To Set Up Continuous Integration Pipelines in Jenkins on Ubuntu 16.04.

Implement Container Logging and Monitoring

When working with containers, it’s important to think about the logging infrastructure you will use to manage and store logs for all your running and stopped containers. There are multiple container-level patterns you can use for logging, and also multiple Kubernetes-level patterns.

In Kubernetes, by default containers use the `json-file` Docker logging driver, which captures the stdout and stderr streams and writes them to JSON files on the Node where the container is running. Sometimes logging directly to stderr and stdout may not be enough for your application container, and you may want to pair the app container with a logging sidecar container in a Kubernetes Pod. This sidecar container can then pick up logs from the filesystem, a local socket, or the systemd journal, granting you a little more flexibility than simply using the stderr and stdout streams. This container can also do some processing and then stream enriched logs to stdout/stderr, or directly to a logging backend. To learn more about Kubernetes logging patterns, consult the Kubernetes logging and monitoring section of this tutorial.

How your application logs at the container level will depend on its complexity. For simple, single-purpose microservices, logging directly to stdout/stderr and letting Kubernetes pick up these streams is the recommended approach, as you can then leverage the `kubectl logs` command to access log streams from your Kubernetes-deployed containers.

Similar to logging, you should begin thinking about monitoring in a container and cluster-based environment. Docker provides the helpful `docker stats` command for grabbing standard metrics like CPU and memory usage for running containers on the host, and exposes even more metrics through the Remote REST API. Additionally, the open-source tool cAdvisor (installed on Kubernetes Nodes by default) provides more advanced functionality like historical metric collection, metric data export, and a helpful web UI for sorting through the data.

However, in a multi-node, multi-container production environment, more complex metrics stacks like Prometheus and Grafana may help organize and monitor your containers' performance data.

Summary

In these sections, we briefly discussed some best practices for building containers, setting up a CI/CD pipeline and image registry, as well as some considerations for increasing observability into your containers.

- To learn more about optimizing containers for Kubernetes, consult Building Optimized Containers for Kubernetes.

- To learn more about CI/CD, consult An Introduction to Continuous Integration, Delivery, and Deployment and An Introduction to CI/CD Best Practices.

In the next section, we'll explore Kubernetes features that allow you to run and scale your containerized app in a cluster.

Deploying on Kubernetes

At this point, you've containerized your app and implemented logic to maximize its portability and observability in Cloud Native environments. We'll now explore Kubernetes features that provide simple interfaces for managing and scaling your apps in a Kubernetes cluster.

Write Deployment and Pod Configuration Files

Once you've containerized your application and published it to a registry, you can now deploy it into a Kubernetes cluster using the Pod workload. The smallest deployable unit in a Kubernetes cluster is not a container but a Pod. Pods typically consist of an application container (like a containerized Flask web app), or an app container and any "sidecar" containers that perform some helper function like monitoring or logging. Containers in a Pod share storage resources, a network namespace, and port space. They can communicate with each other using `localhost` and can share data using mounted volumes. Addtionally, the Pod workload allows you to define Init Containers that run setup scripts or utilities before the main app container begins running.

Pods are typically rolled out using Deployments, which are Controllers defined by YAML files that declare a particular desired state. For example, an application state could be running three replicas of the Flask web app container and exposing port 8080. Once created, the control plane gradually brings the actual state of the cluster to match the desired state declared in the Deployment by scheduling containers onto Nodes as required. To scale the number of application replicas running in the cluster, say from 3 up to 5, you update the `replicas` field of the Deployment configuration file, and then `kubectl apply` the new configuration file. Using these configuration files, scaling and deployment operations can all be tracked and versioned using your existing source control services and integrations.

Here's a sample Kubernetes Deployment configuration file for a Flask app:

flask_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
  labels:
    app: flask-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask
        image: sammy/flask_app:1.0
        ports:
        - containerPort: 8080
```

This Deployment launches 3 Pods that run a container called `flask` using the `sammy/flask_app` image (version `1.0`) with port `8080` open. The Deployment is called `flask-app`.

To learn more about Kubernetes Pods and Deployments, consult the Pods and Deployments sections of the official Kubernetes documentation.

Configure Pod Storage

Kubernetes manages Pod storage using Volumes, Persistent Volumes (PVs) and Persistent Volume Claims (PVCs). Volumes are the Kubernetes abstraction used to manage Pod storage, and support most cloud provider block storage offerings, as well as local storage on the Nodes hosting the running Pods. To see a full list of supported Volume types, consult the Kubernetes documentation.

For example, if your Pod contains two NGINX containers that need to share data between them (say the first, called `nginx` serves web pages, and the second, called `nginx-sync` fetches the pages from an external location and updates the pages served by the `nginx` container), your Pod spec would look something like this (here we use the `emptyDir` Volume type):

pod_volume.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: nginx-web
      mountPath: /usr/share/nginx/html

  - name: nginx-sync
    image: nginx-sync
    volumeMounts:
    - name: nginx-web
      mountPath: /web-data

  volumes:
  - name: nginx-web
    emptyDir: {}
```

We use a `volumeMount` for each container, indicating that we'd like to mount the `nginx-web` volume containing the web page files at `/usr/share/nginx/html` in the `nginx` container and at `/web-data` in the `nginx-sync` container. We also define a `volume` called `nginx-web` of type `emptyDir`.

In a similar fashion, you can configure Pod storage using cloud block storage products by modifying the `volume` type from `emptyDir` to the relevant cloud storage volume type.

The lifecycle of a Volume is tied to the lifecycle of the Pod, but not to that of a container. If a container within a Pod dies, the Volume persists and the newly launched container will be able to mount the same Volume and access its data. When a Pod gets restarted or dies, so do its Volumes, although if the Volumes consist of cloud block storage, they will simply be unmounted with data still accessible by future Pods.

To preserve data across Pod restarts and updates, the PersistentVolume (PV) and PersistentVolumeClaim (PVC) objects must be used.

PersistentVolumes are abstractions representing pieces of persistent storage like cloud block storage volumes or NFS storage. They are created separately from PersistentVolumeClaims, which are demands for pieces of storage by developers. In their Pod configurations, developers request persistent storage using PVCs, which Kubernetes matches with available PV Volumes (if using cloud block storage, Kubernetes can dynamically create PersistentVolumes when PersistentVolumeClaims are created).

If your application requires one persistent volume per replica, which is the case with many databases, you should not use Deployments but use the StatefulSet controller, which is designed for apps that require stable network identifiers, stable persistent storage, and ordering guarantees. Deployments should be used for stateless applications, and if you define a PersistentVolumeClaim for use in a Deployment configuration, that PVC will be shared by all the Deployment's replicas.

To learn more about the StatefulSet controller, consult the Kubernetes documentation. To learn more about PersistentVolumes and

PersistentVolume claims, consult the Kubernetes storage documentation.

Injecting Configuration Data with Kubernetes

Similar to Docker, Kubernetes provides the `env` and `envFrom` fields for setting environment variables in Pod configuration files. Here's a sample snippet from a Pod configuration file that sets the `HOSTNAME` environment variable in the running Pod to `my_hostname` :

sample_pod.yaml

```
...
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        env:
        - name: HOSTNAME
          value: my_hostname
...
```

This allows you to move configuration out of Dockerfiles and into Pod and Deployment configuration files. A key advantage of further externalizing configuration from your Dockerfiles is that you can now modify these Kubernetes workload configurations (say, by changing the `HOSTNAME` value to `my_hostname_2`) separately from your application container definitions. Once you modify the Pod configuration file, you can then redeploy the Pod using its new environment, while the underlying container image (defined via its Dockerfile) does not need to be rebuilt, tested, and pushed to a repository. You can also version these Pod and Deployment configurations separately from your Dockerfiles, allowing you to quickly detect breaking changes and further separate config issues from application bugs.

Kubernetes provides another construct for further externalizing and managing configuration data: ConfigMaps and Secrets.

ConfigMaps and Secrets

ConfigMaps allow you to save configuration data as objects that you then reference in your Pod and Deployment configuration files, so that you can avoid hardcoding configuration data and reuse it across Pods and Deployments.

Here's an example, using the Pod config from above. We'll first save the `HOSTNAME` environment variable as a ConfigMap, and then reference it in the Pod config:

```
kubectl create configmap hostname --from-
literal=HOSTNAME=my_host_name
```

To reference it from the Pod configuration file, we use the the `valueFrom` and `configMapKeyRef` constructs:

sample_pod_configmap.yaml

```
...
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        env:
        - name: HOSTNAME
          valueFrom:
            configMapKeyRef:
              name: hostname
              key: HOSTNAME
...
```

So the `HOSTNAME` environment variable's value has been completely externalized from configuration files. We can then update these variables across all Deployments and Pods referencing them, and restart the Pods for the changes to take effect.

If your applications use configuration files, ConfigMaps additionally allow you to store these files as ConfigMap objects (using the `--from-file` flag), which you can then mount into containers as configuration files.

Secrets provide the same essential functionality as ConfigMaps, but should be used for sensitive data like database credentials as the values are base64-encoded.

To learn more about ConfigMaps and Secrets consult the Kubernetes documentation.

Create Services

Once you have your application up and running in Kubernetes, every Pod will be assigned an (internal) IP address, shared by its containers. If one of these Pods is removed or dies, newly started Pods will be assigned different IP addresses.

For long-running services that expose functionality to internal and/or external clients, you may wish to grant a set of Pods performing the same function (or Deployment) a stable IP address that load balances requests across its containers. You can do this using a Kubernetes Service.

Kubernetes Services have 4 types, specified by the `type` field in the Service configuration file:

- `ClusterIP`: This is the default type, which grants the Service a stable internal IP accessible from anywhere inside of the cluster.
- `NodePort`: This will expose your Service on each Node at a static port, between 30000-32767 by default. When a request hits a Node at its Node IP address and the `NodePort` for your service, the request will be load balanced and routed to the application containers for your service.
- `LoadBalancer`: This will create a load balancer using your cloud provider's load balancing product, and configure a `NodePort` and `ClusterIP` for your Service to which external requests will be routed.
- `ExternalName`: This Service type allows you to map a Kubernetes Service to a DNS record. It can be used for accessing external services from your Pods using Kubernetes DNS.

Note that creating a Service of type `LoadBalancer` for each Deployment running in your cluster will create a new cloud load balancer for each Service, which can become costly. To manage routing external requests to multiple services using a single load balancer, you can use an Ingress Controller. Ingress Controllers are beyond the scope of this article, but to learn more about them you can consult the Kubernetes documentation. A popular simple Ingress Controller is the NGINX Ingress Controller.

Here's a simple Service configuration file for the Flask example used in the Pods and Deployments section of this guide:

flask_app_svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: flask-svc
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: flask-app
  type: LoadBalancer
```

Here we choose to expose the `flask-app` Deployment using this `flask-svc` Service. We create a cloud load balancer to route traffic from load balancer port `80` to exposed container port `8080`.

To learn more about Kubernetes Services, consult the Services section of the Kubernetes docs.

Logging and Monitoring

Parsing through individual container and Pod logs using `kubectl logs` and `docker logs` can get tedious as the number of running applications grows. To help you debug application or cluster issues, you should implement centralized logging. At a high level, this consists of agents running on all the worker nodes that process Pod log files and streams, enrich them with metadata, and forward the logs off to a backend like Elasticsearch. From there, log data can be visualized, filtered, and organized using a visualization tool like Kibana.

In the container-level logging section, we discussed the recommended Kubernetes approach of having applications in containers log to the stdout/stderr streams. We also briefly discussed logging sidecar containers that can grant you more flexibility when logging from your application. You could also run logging agents directly in your Pods that capture local log data and forward them directly to your logging backend. Each approach has its pros and cons, and resource utilization tradeoffs (for example, running a logging agent container inside of each Pod can become resource-intensive and quickly overwhelm your logging backend). To learn more about different logging architectures and their tradeoffs, consult the Kubernetes documentation.

In a standard setup, each Node runs a logging agent like Filebeat or Fluentd that picks up container logs created by Kubernetes. Recall that Kubernetes creates JSON log files for containers on the Node (in most installations these can be found at `/var/lib/docker/containers/`). These should be rotated using a tool like logrotate. The Node logging agent should be run as a DaemonSet Controller, a type of Kubernetes Workload that ensures that every Node runs a copy of the DaemonSet Pod. In this case the Pod would contain the logging agent and its configuration, which processes logs from files and directories mounted into the logging DaemonSet Pod.

Similar to the bottleneck in using `kubectl logs` to debug container issues, eventually you may need to consider a more robust option than simply using `kubectl top` and the Kubernetes Dashboard to monitor Pod resource usage on your cluster. Cluster and application-level monitoring can be set up using the Prometheus monitoring system and time-series database, and Grafana metrics dashboard. Prometheus works using a “pull” model, which scrapes HTTP endpoints (like `/metrics/cadvisor` on the Nodes, or the `/metrics` application REST API endpoints) periodically for metric data, which it then processes and stores. This data can then be analyzed and visualized using Grafana dashboard. Prometheus and Grafana can be launched into a Kubernetes cluster like any other Deployment and Service.

For added resiliency, you may wish to run your logging and monitoring infrastructure on a separate Kubernetes cluster, or using external logging and metrics services.

Conclusion

Migrating and modernizing an application so that it can efficiently run in a Kubernetes cluster often involves non-trivial amounts of planning and architecting of software and infrastructure changes. Once implemented, these changes allow service owners to continuously deploy new versions of their apps and easily scale them as necessary, with minimal amounts of manual intervention. Steps like externalizing configuration from your app, setting up proper logging and metrics publishing, and configuring health checks allow you to fully take advantage of the Cloud Native paradigm that Kubernetes has been designed around. By building portable containers and managing them using Kubernetes objects like Deployments and Services, you can fully use your available compute infrastructure and development resources.

How To Build a Node.js Application with Docker

Written by Kathleen Juell

This tutorial is a first step towards writing an example Node.js application that will run on Kubernetes. When building and scaling an application on Kubernetes, the starting point is typically creating a Docker image, which you can then run as a Pod in a Kubernetes cluster. The image includes your application code, dependencies, environment variables, and application runtime environment. Using an image ensures that the environment in your container is standardized and contains only what is necessary to build and run your application.

In this tutorial, you will create an application image for a static website that uses the Express Node.js framework and Bootstrap front-end library. You will then push the image to Docker Hub for future use and then run a container using that image. Finally, you will pull the stored image from your Docker Hub repository and run another container, demonstrating how you can quickly recreate and scale your application. As you move through this curriculum, subsequent tutorials will expand on this initial image until it is up and running directly on Kubernetes.

The Docker platform allows developers to package and run applications as containers. A container is an isolated process that runs on a shared operating system, offering a lighter weight alternative to virtual machines. Though containers are not new, they offer benefits — including process isolation and environment standardization — that are growing in importance as more developers use distributed application architectures.

When building and scaling an application with Docker, the starting point is typically creating an image for your application, which you can then run in a container. The image includes your application code, libraries, configuration files, environment variables, and runtime. Using an image ensures that the environment in your container is standardized and contains only what is necessary to build and run your application.

In this tutorial, you will create an application image for a static website that uses the Express framework and Bootstrap. You will then build a container using that image and push it to Docker Hub for future use. Finally, you will pull the stored image from your Docker Hub repository and build another container, demonstrating how you can recreate and scale your application.

Prerequisites

To follow this tutorial, you will need: - One Ubuntu 18.04 server, set up following this Initial Server Setup guide. - Docker installed on your server, following Steps 1 and 2 of How To Install and Use Docker on Ubuntu 18.04. - Node.js and npm installed, following these instructions on installing with the PPA managed by NodeSource. - A Docker Hub account. For an overview of how to set this up, refer to this introduction on getting started with Docker Hub.

Step 1 — Installing Your Application Dependencies

To create your image, you will first need to make your application files, which you can then copy to your container. These files will include your application's static content, code, and dependencies.

First, create a directory for your project in your non-root user's home directory. We will call ours **`node_project`**, but you should feel free to replace this with something else:

```
mkdir node_project
```

Navigate to this directory:

```
cd node_project
```

This will be the root directory of the project.

Next, create a [`package.json`](package.json) file with your project's dependencies and other identifying information. Open the file with `nano` or your favorite editor:

```
nano package.json
```

Add the following information about the project, including its name, author, license, entrypoint, and dependencies. Be sure to replace the author information with your own name and contact details:

~/node_project/package.json

```
{
  "name": "nodejs-image-demo",
  "version": "1.0.0",
  "description": "nodejs image demo",
  "author": "Sammy the Shark <sammy@example.com>",
  "license": "MIT",
  "main": "app.js",
  "keywords": [
    "nodejs",
    "bootstrap",
    "express"
  ],
  "dependencies": {
    "express": "^4.16.4"
  }
}
```

This file includes the project name, author, and license under which it is being shared. Npm recommends making your project name short and descriptive, and avoiding duplicates in the npm registry. We've listed the MIT license in the license field, permitting the free use and distribution of the application code.

Additionally, the file specifies: - `"main"`: The entrypoint for the application, `app.js`. You will create this file next. - `"dependencies"`: The project dependencies — in this case, Express 4.16.4 or above.

Though this file does not list a repository, you can add one by following these guidelines on adding a repository to your `package.json` file. This is a good addition if you are versioning your application.

Save and close the file when you've finished making changes.

To install your project's dependencies, run the following command:

```
npm install
```

This will install the packages you've listed in your `package.json` file in your project directory.

We can now move on to building the application files.

Step 2 — Creating the Application Files

We will create a website that offers users information about sharks. Our application will have a main entrypoint, `app.js`, and a `views` directory that will include the project's static assets. The landing page, `index.html`, will offer users some preliminary information and a link to a page with more detailed shark information, `sharks.html`. In the `views` directory, we will create both the landing page and `sharks.html`.

First, open `app.js` in the main project directory to define the project's routes:

```
nano app.js
```

The first part of the file will create the Express application and Router objects, and define the base directory and port as constants:

~/node_project/app.js

```
const express = require('express');
const app = express();
const router = express.Router();

const path = __dirname + '/views/';
const port = 8080;
```

The `require` function loads the `express` module, which we then use to create the `app` and `router` objects. The `router` object will perform the routing function of the application, and as we define HTTP method routes we will add them to this object to define how our application will handle requests.

This section of the file also sets a couple of constants, `path` and `port`:
- `path`: Defines the base directory, which will be the `views` subdirectory

within the current project directory. - `port`: Tells the app to listen on and bind to port `8080`.

Next, set the routes for the application using the `router` object:

~/node_project/app.js

```
...

router.use(function (req,res,next) {
  console.log('/' + req.method);
  next();
});

router.get('/', function(req,res){
  res.sendFile(path + 'index.html');
});

router.get('/sharks', function(req,res){
  res.sendFile(path + 'sharks.html');
});
```

The `router.use` function loads a middleware function that will log the router's requests and pass them on to the application's routes. These are defined in the subsequent functions, which specify that a GET request to the base project URL should return the `index.html` page, while a GET request to the `/sharks` route should return `sharks.html`.

Finally, mount the `router` middleware and the application's static assets and tell the app to listen on port `8080`:

~/node_project/app.js

```
...

app.use(express.static(path));
app.use('/', router);

app.listen(port, function () {
  console.log('Example app listening on port
8080!')
})
```

The finished `app.js` file will look like this:

~/node_project/app.js

```
const express = require('express');
const app = express();
const router = express.Router();

const path = __dirname + '/views/';
const port = 8080;

router.use(function (req,res,next) {
  console.log('/' + req.method);
  next();
});

router.get('/', function(req,res){
  res.sendFile(path + 'index.html');
});

router.get('/sharks', function(req,res){
  res.sendFile(path + 'sharks.html');
});

app.use(express.static(path));
app.use('/', router);

app.listen(port, function () {
  console.log('Example app listening on port
8080!')
})
```

Save and close the file when you are finished.

Next, let’s add some static content to the application. Start by creating the `views` directory:

```
mkdir views
```

Open the landing page file, `index.html`:

```
nano views/index.html
```

Add the following code to the file, which will import Boostrap and create a jumbotron component with a link to the more detailed `sharks.html` info page:

~/node_project/views/index.html

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

```
<head>
    <title>About Sharks</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1">
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap
/4.1.3/css/bootstrap.min.css" integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkF
OJwJ8ERdknLPMO" crossorigin="anonymous">
    <link href="css/styles.css" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?
family=Merriweather:400,700" rel="stylesheet"
type="text/css">
</head>

<body>
    <nav class="navbar navbar-dark bg-dark navbar-
static-top navbar-expand-md">
        <div class="container">
            <button type="button" class="navbar-
toggler collapsed" data-toggle="collapse" data-
target="#bs-example-navbar-collapse-1" aria-
expanded="false"> <span class="sr-only">Toggle
navigation
            </button> <a class="navbar-brand"
href="#">Everything Sharks</a>
```

```
            <div class="collapse navbar-collapse"
id="bs-example-navbar-collapse-1">
                <ul class="nav navbar-nav mr-
auto">
                    <li class="active nav-item"><a
href="/" class="nav-link">Home</a>
                    </li>
                    <li class="nav-item"><a
href="/sharks" class="nav-link">Sharks</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="jumbotron">
        <div class="container">
            <h1>Want to Learn About Sharks?</h1>
            <p>Are you ready to learn about
sharks?</p>
            <br>
            <p><a class="btn btn-primary btn-lg"
href="/sharks" role="button">Get Shark Info</a>
            </p>
        </div>
    </div>
    <div class="container">
        <div class="row">
```

```
            <div class="col-lg-6">
                <h3>Not all sharks are alike</h3>
                <p>Though some are dangerous,
sharks generally do not attack humans. Out of the
500 species known to researchers, only 30 have
been known to attack humans.
                </p>
            </div>
            <div class="col-lg-6">
                <h3>Sharks are ancient</h3>
                <p>There is evidence to suggest
that sharks lived up to 400 million years ago.
                </p>
            </div>
        </div>
    </div>
</body>

</html>
```

The top-level navbar here allows users to toggle between the Home and Sharks pages. In the `navbar-nav` subcomponent, we are using Bootstrap’s `active` class to indicate the current page to the user. We’ve also specified the routes to our static pages, which match the routes we defined in `app.js`:

~/node_project/views/index.html

```
...
<div class="collapse navbar-collapse" id="bs-
example-navbar-collapse-1">
   <ul class="nav navbar-nav mr-auto">
      <li class="active nav-item"><a href="/"
class="nav-link">Home</a>
      </li>
      <li class="nav-item"><a href="/sharks"
class="nav-link">Sharks</a>
      </li>
   </ul>
</div>
...
```

Additionally, we’ve created a link to our shark information page in our jumbotron’s button:

~/node_project/views/index.html

```
...
<div class="jumbotron">
   <div class="container">
      <h1>Want to Learn About Sharks?</h1>
      <p>Are you ready to learn about sharks?</p>
      <br>
      <p><a class="btn btn-primary btn-lg"
href="/sharks" role="button">Get Shark Info</a>
      </p>
   </div>
```

```
</div>
...
```

There is also a link to a custom style sheet in the header:

~/node_project/views/index.html

```
...
<link href="css/styles.css" rel="stylesheet">
...
```

We will create this style sheet at the end of this step.

Save and close the file when you are finished.

With the application landing page in place, we can create our shark information page, `sharks.html`, which will offer interested users more information about sharks.

Open the file:

```
nano views/sharks.html
```

Add the following code, which imports Bootstrap and the custom style sheet and offers users detailed information about certain sharks:

~/node_project/views/sharks.html

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

<head>
    <title>About Sharks</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1">
```

```
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap
/4.1.3/css/bootstrap.min.css" integrity="sha384-
MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkF
OJwJ8ERdknLPMO" crossorigin="anonymous">
    <link href="css/styles.css" rel="stylesheet">
    <link href="https://fonts.googleapis.com/css?
family=Merriweather:400,700" rel="stylesheet"
type="text/css">
</head>
<nav class="navbar navbar-dark bg-dark navbar-
static-top navbar-expand-md">
    <div class="container">
        <button type="button" class="navbar-
toggler collapsed" data-toggle="collapse" data-
target="#bs-example-navbar-collapse-1" aria-
expanded="false"> <span class="sr-only">Toggle
navigation
        </button> <a class="navbar-brand"
href="/">Everything Sharks</a>
        <div class="collapse navbar-collapse"
id="bs-example-navbar-collapse-1">
            <ul class="nav navbar-nav mr-auto">
                <li class="nav-item"><a href="/"
class="nav-link">Home</a>
                </li>
                <li class="active nav-item"><a
```

```
href="/sharks" class="nav-link">Sharks</a>
                </li>
            </ul>
        </div>
    </div>
</nav>
<div class="jumbotron text-center">
    <h1>Shark Info</h1>
</div>
<div class="container">
    <div class="row">
        <div class="col-lg-6">
            <p>
                <div class="caption">Some sharks
are known to be dangerous to humans, though many
more are not. The sawshark, for example, is not
considered a threat to humans.
                </div>
                <img
src="https://assets.digitalocean.com/articles/dock
er_node_image/sawshark.jpg" alt="Sawshark">
            </p>
        </div>
        <div class="col-lg-6">
            <p>
                <div class="caption">Other sharks
are known to be friendly and welcoming!</div>
```

```
                <img
src="https://assets.digitalocean.com/articles/dock
er_node_image/sammy.png" alt="Sammy the Shark">
            </p>
        </div>
    </div>
</div>

</html>
```

Note that in this file, we again use the `active` class to indicate the current page.

Save and close the file when you are finished.

Finally, create the custom CSS style sheet that you've linked to in `index.html` and `sharks.html` by first creating a `css` folder in the `views` directory:

```
mkdir views/css
```

Open the style sheet:

```
nano views/css/styles.css
```

Add the following code, which will set the desired color and font for our pages:

~/node_project/views/css/styles.css

```
.navbar {
    margin-bottom: 0;
}

body {
    background: #020A1B;
    color: #ffffff;
    font-family: 'Merriweather', sans-serif;
}

h1,
h2 {
    font-weight: bold;
}

p {
    font-size: 16px;
    color: #ffffff;
}

.jumbotron {
    background: #0048CD;
    color: white;
    text-align: center;
}

.jumbotron p {
    color: white;
    font-size: 26px;
}

.btn-primary {
    color: #fff;
    text-color: #000000;
    border-color: white;
    margin-bottom: 5px;
}

img,
video,
audio {
    margin-top: 20px;
    max-width: 80%;
}

div.caption: {
    float: left;
    clear: both;
}
```

In addition to setting font and color, this file also limits the size of the images by specifying a `max-width` of 80%. This will prevent them from taking up more room than we would like on the page.

Save and close the file when you are finished.

With the application files in place and the project dependencies installed, you are ready to start the application.

If you followed the initial server setup tutorial in the prerequisites, you will have an active firewall permitting only SSH traffic. To permit traffic to port `8080` run:

```
sudo ufw allow 8080
```

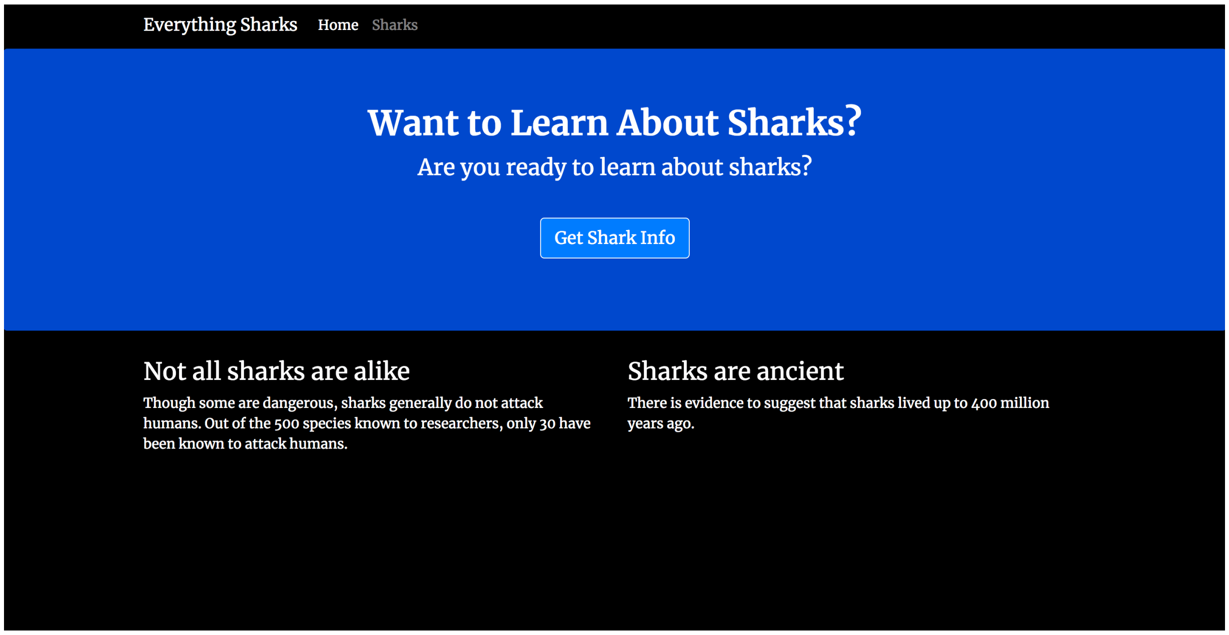
To start the application, make sure that you are in your project's root directory:

```
cd ~/node_project
```

Start the application with `node app.js`:

```
node app.js
```

Navigate your browser to `http://your_server_ip:8080`. You will see the following landing page:



Everything Sharks Home Sharks

Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

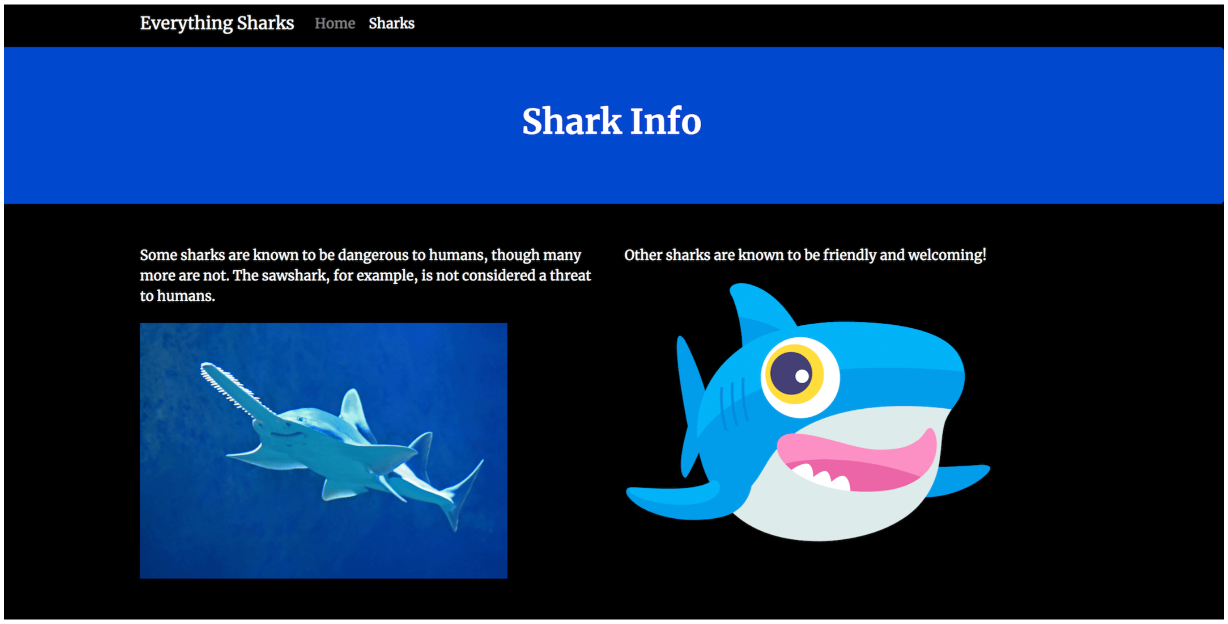
Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Application Landing Page

Click on the Get Shark Info button. You will see the following information page:



Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Shark Info Page

You now have an application up and running. When you are ready, quit the server by typing `CTRL+C`. We can now move on to creating the Dockerfile that will allow us to recreate and scale this application as desired.

Step 3 — Writing the Dockerfile

Your Dockerfile specifies what will be included in your application container when it is executed. Using a Dockerfile allows you to define your container environment and avoid discrepancies with dependencies or runtime versions.

Following these guidelines on building optimized containers, we will make our image as efficient as possible by minimizing the number of image layers and restricting the image's function to a single purpose — recreating our application files and static content.

In your project's root directory, create the Dockerfile:

```
nano Dockerfile
```

Docker images are created using a succession of layered images that build on one another. Our first step will be to add the base image for our application that will form the starting point of the application build.

Let's use the `node:10-alpine` image, since at the time of writing this is the recommended LTS version of Node.js. The `alpine` image is derived from the Alpine Linux project, and will help us keep our image size down. For more information about whether or not the `alpine` image is the right choice for your project, please see the full discussion under the Image Variants section of the Docker Hub Node image page.

Add the following `FROM` instruction to set the application's base image:

~/node_project/Dockerfile

```
FROM node:10-alpine
```

This image includes Node.js and npm. Each Dockerfile must begin with a `FROM` instruction.

By default, the Docker Node image includes a non-root node user that you can use to avoid running your application container as root. It is a recommended security practice to avoid running containers as root and to restrict capabilities within the container to only those required to run its processes. We will therefore use the node user's home directory as the working directory for our application and set them as our user inside the container. For more information about best practices when working with the Docker Node image, see this best practices guide.

To fine-tune the permissions on our application code in the container, let's create the `node_modules` subdirectory in `/home/node` along with the `app` directory. Creating these directories will ensure that they have the permissions we want, which will be important when we create local node modules in the container with `npm install`. In addition to creating these directories, we will set ownership on them to our node user:

~/node_project/Dockerfile

```
...
RUN mkdir -p /home/node/app/node_modules && chown
-R node:node /home/node/app
```

For more information on the utility of consolidating `RUN` instructions, see this discussion of how to manage container layers.

Next, set the working directory of the application to `/home/node/app`:

~/node_project/Dockerfile

```
...
WORKDIR /home/node/app
```

If a `WORKDIR` isn't set, Docker will create one by default, so it's a good idea to set it explicitly.

Next, copy the `package.json` and `package-lock.json` (for npm 5+) files:

~/node_project/Dockerfile

```
...
COPY package*.json ./
```

Adding this `COPY` instruction before running `npm install` or copying the application code allows us to take advantage of Docker's caching mechanism. At each stage in the build, Docker will check to see if it has a layer cached for that particular instruction. If we change `package.json`, this layer will be rebuilt, but if we don't, this instruction will allow Docker to use the existing image layer and skip reinstalling our node modules.

To ensure that all of the application files are owned by the non-root node user, including the contents of the `node_modules` directory, switch the user to node before running `npm install`:

~/node_project/Dockerfile

```
...
USER node
```

After copying the project dependencies and switching our user, we can run `npm install`:

~/node_project/Dockerfile

```
...
RUN npm install
```

Next, copy your application code with the appropriate permissions to the application directory on the container:

~/node_project/Dockerfile

```
...
COPY --chown=node:node . .
```

This will ensure that the application files are owned by the non-root node user.

Finally, expose port `8080` on the container and start the application:

~/node_project/Dockerfile

```
...
EXPOSE 8080

CMD [ "node", "app.js" ]
```

`EXPOSE` does not publish the port, but instead functions as a way of documenting which ports on the container will be published at runtime. `CMD` runs the command to start the application — in this case, `node app.js`. Note that there should only be one `CMD` instruction in each Dockerfile. If you include more than one, only the last will take effect.

There are many things you can do with the Dockerfile. For a complete list of instructions, please refer to Docker's Dockerfile reference documentation.

The complete Dockerfile looks like this:

~/node_project/Dockerfile

```
FROM node:10-alpine

RUN mkdir -p /home/node/app/node_modules && chown
-R node:node /home/node/app

WORKDIR /home/node/app

COPY package*.json ./

USER node

RUN npm install
```

```
COPY --chown=node:node . .

EXPOSE 8080

CMD [ "node", "app.js" ]
```

Save and close the file when you are finished editing.

Before building the application image, let's add a `.dockerignore` file. Working in a similar way to a `.gitignore` file, `.dockerignore` specifies which files and directories in your project directory should not be copied over to your container.

Open the `.dockerignore` file:

```
nano .dockerignore
```

Inside the file, add your local node modules, npm logs, Dockerfile, and `.dockerignore` file:

~/node_project/.dockerignore

```
node_modules
npm-debug.log
Dockerfile
.dockerignore
```

If you are working with Git then you will also want to add your `.git` directory and `.gitignore` file.

Save and close the file when you are finished.

You are now ready to build the application image using the `docker build` command. Using the `-t` flag with `docker build` will allow you to tag the image with a memorable name. Because we are going to push the image to Docker Hub, let's include our Docker Hub username in the tag. We will tag the image as **nodejs-image-demo**, but feel free to replace this with a name of your own choosing. Remember to also replace **your_dockerhub_username** with your own Docker Hub username:

```
docker build -t your_dockerhub_username/nodejs-
image-demo .
```

The `.` specifies that the build context is the current directory.

It will take a minute or two to build the image. Once it is complete, check your images:

```
docker images
```

You will see the following output:

Output

```
REPOSITORY
TAG                    IMAGE ID               CREATED
SIZE
your_dockerhub_username/nodejs-image-demo
latest                 1c723fb2ef12           8 seconds
ago         73MB
node
10-alpine              f09e7c96b6de           3 weeks
ago          70.7MB
```

It is now possible to create a container with this image using [docker run](https://docs.docker.com/engine/reference/run/). We will include three flags with this command: - `-p`: This publishes the port on the container and maps it to a port on our host. We will use port `80` on the host, but you should feel free to modify this as necessary if you have another process running on that port. For more information about how this works, see this discussion in the Docker docs on port binding. - -d: This runs the container in the background. - --name: This allows us to give the container a memorable name.

Run the following command to build the container:

```
docker run --name nodejs-image-demo -p 80:8080 -d
your_dockerhub_username/nodejs-image-demo
```

Once your container is up and running, you can inspect a list of your running containers with `docker ps`:

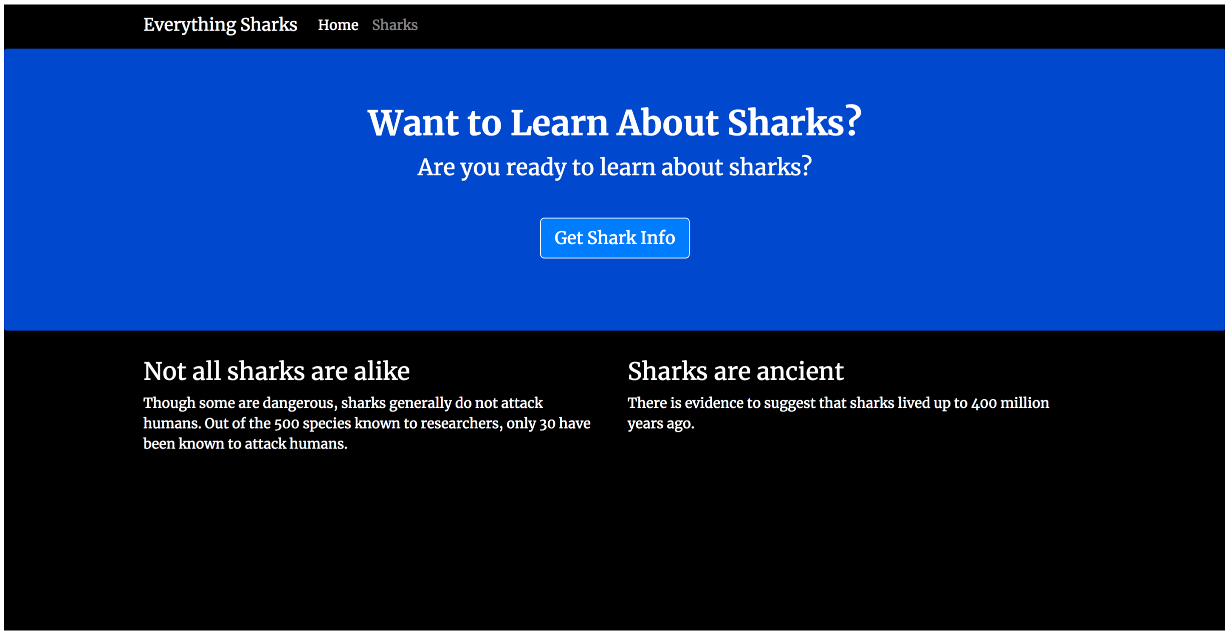
```
docker ps
```

You will see the following output:

Output

```
CONTAINER ID        IMAGE
COMMAND             CREATED             STATUS
PORTS                  NAMES
e50ad27074a7
your_dockerhub_username/nodejs-image-demo
"node app.js"       8 seconds ago       Up 7
seconds        0.0.0.0:80->8080/tcp   nodejs-
image-demo
```

With your container running, you can now visit your application by navigating your browser to `http://your_server_ip`. You will see your application landing page once again:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Application Landing Page

Now that you have created an image for your application, you can push it to Docker Hub for future use.

Step 4 — Using a Repository to Work with Images

By pushing your application image to a registry like Docker Hub, you make it available for subsequent use as you build and scale your containers. We will demonstrate how this works by pushing the application image to a repository and then using the image to recreate our container.

The first step to pushing the image is to log in to the Docker Hub account you created in the prerequisites:

```
docker login -u your_dockerhub_username
```

When prompted, enter your Docker Hub account password. Logging in this way will create a `~/.docker/config.json` file in your user's home directory with your Docker Hub credentials.

You can now push the application image to Docker Hub using the tag you created earlier, **your_dockerhub_username**/**nodejs-image-demo**:

```
docker push your_dockerhub_username/nodejs-image-demo
```

Let's test the utility of the image registry by destroying our current application container and image and rebuilding them with the image in our repository.

First, list your running containers:

```
docker ps
```

You will see the following output:

Output

```
CONTAINER ID          IMAGE
COMMAND               CREATED               STATUS
PORTS                    NAMES
e50ad27074a7
your_dockerhub_username/nodejs-image-demo   "node
app.js"       3 minutes ago       Up 3 minutes
0.0.0.0:80->8080/tcp   nodejs-image-demo
```

Using the `CONTAINER ID` listed in your output, stop the running application container. Be sure to replace the highlighted ID below with your own `CONTAINER ID`:

```
docker stop e50ad27074a7
```

List your all of your images with the `-a` flag:

```
docker images -a
```

You will see the following output with the name of your image, **your_dockerhub_username/nodejs-image-demo**, along with the `node` image and the other images from your build:

Output

```
REPOSITORY
TAG                    IMAGE ID            CREATED
SIZE
your_dockerhub_username/nodejs-image-demo
latest                 1c723fb2ef12        7 minutes
ago        73MB
<none>
<none>                 2e3267d9ac02        4 minutes
ago        72.9MB
<none>
<none>                 8352b41730b9        4 minutes
ago        73MB
<none>
<none>                 5d58b92823cb        4 minutes
ago        73MB
<none>
<none>                 3f1e35d7062a        4 minutes
ago        73MB
<none>
<none>                 02176311e4d0        4 minutes
ago        73MB
<none>
```

```
<none>                  8e84b33edcda         4 minutes
ago        70.7MB
<none>
<none>                  6a5ed70f86f2         4 minutes
ago        70.7MB
<none>
<none>                  776b2637d3c1         4 minutes
ago        70.7MB
node
10-alpine               f09e7c96b6de         3 weeks
ago          70.7MB
```

Remove the stopped container and all of the images, including unused or dangling images, with the following command:

```
docker system prune -a
```

Type `y` when prompted in the output to confirm that you would like to remove the stopped container and images. Be advised that this will also remove your build cache.

You have now removed both the container running your application image and the image itself. For more information on removing Docker containers, images, and volumes, please see How To Remove Docker Images, Containers, and Volumes.

With all of your images and containers deleted, you can now pull the application image from Docker Hub:

```
docker pull your_dockerhub_username/nodejs-image-demo
```

List your images once again:

```
docker images
```

You will see your application image:

Output

```
REPOSITORY                                              TAG
IMAGE ID                CREATED                 SIZE
your_dockerhub_username/nodejs-image-demo
latest                  1c723fb2ef12            11 minutes
ago         73MB
```

You can now rebuild your container using the command from Step 3:

```
docker run --name nodejs-image-demo -p 80:8080 -d
your_dockerhub_username/nodejs-image-demo
```

List your running containers:

```
docker ps
```

Output

```
CONTAINER ID            IMAGE
COMMAND                 CREATED                 STATUS
PORTS                      NAMES
f6bc2f50dff6
your_dockerhub_username/nodejs-image-demo
"node app.js"           4 seconds ago           Up 3
seconds            0.0.0.0:80->8080/tcp   nodejs-
image-demo
```

Visit `http://your_server_ip` once again to view your running application.

Conclusion

In this tutorial you created a static web application with Express and Bootstrap, as well as a Docker image for this application. You used this image to create a container and pushed the image to Docker Hub. From there, you were able to destroy your image and container and recreate them using your Docker Hub repository.

If you are interested in learning more about how to work with tools like Docker Compose and Docker Machine to create multi-container setups, you can look at the following guides: - How To Install Docker Compose on Ubuntu 18.04. - How To Provision and Manage Remote Docker Hosts with Docker Machine on Ubuntu 18.04.

For general tips on working with container data, see: - How To Share Data between Docker Containers. - How To Share Data Between the Docker Container and the Host.

If you are interested in other Docker-related topics, please see our complete library of Docker tutorials.

Containerizing a Node.js Application for Development With Docker Compose

Written by Kathleen Juell

This tutorial is a second step towards writing an example Node.js application that will run on Kubernetes. Building on the previous tutorial, you will create two containers — one for the Node.js application and another for a MongoDB database — and coordinate running them with Docker Compose.

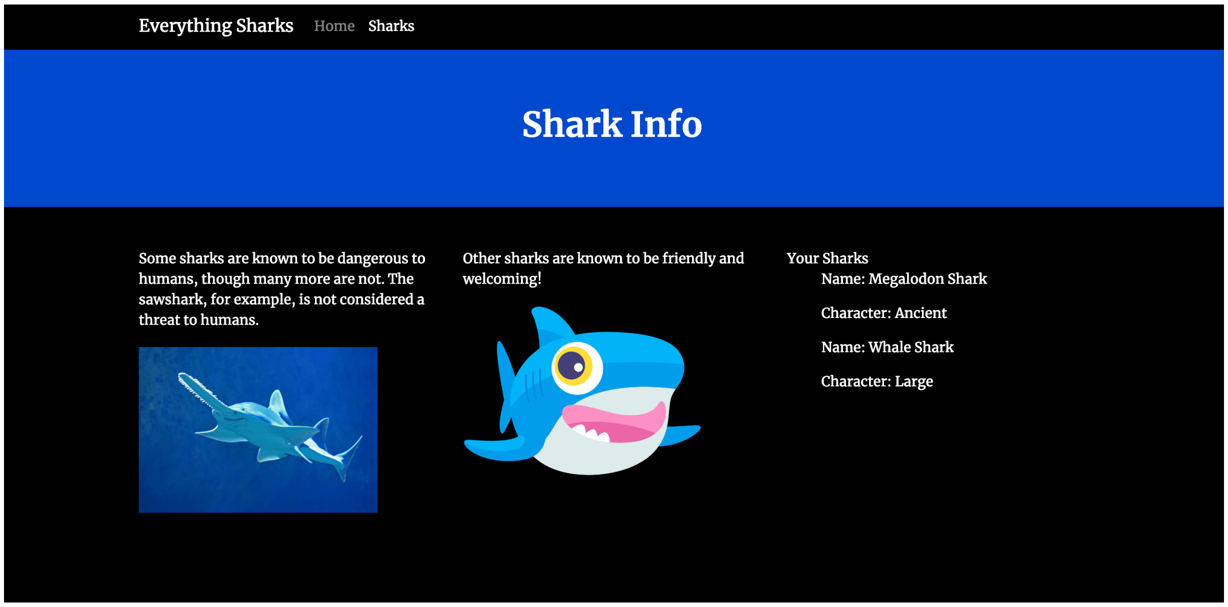
This tutorial demonstrates how to use multiple containers with persistent data. It also highlights the importance of separating the application code from the data store. This design will ensure that the final Docker image for the Node.js application is stateless and that it will be ready to run on Kubernetes by the end of this curriculum.

If you are actively developing an application, using Docker can simplify your workflow and the process of deploying your application to production. Working with containers in development offers the following benefits: - Environments are consistent, meaning that you can choose the languages and dependencies you want for your project without worrying about system conflicts. - Environments are isolated, making it easier to troubleshoot issues and onboard new team members. - Environments are portable, allowing you to package and share your code with others.

This tutorial will show you how to set up a development environment for a Node.js application using Docker. You will create two containers — one for the Node application and another for the MongoDB database —

with Docker Compose. Because this application works with Node and MongoDB, our setup will do the following: - Synchronize the application code on the host with the code in the container to facilitate changes during development. - Ensure that changes to the application code work without a restart. - Create a user and password-protected database for the application's data. - Persist this data.

At the end of this tutorial, you will have a working shark information application running on Docker containers:



Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Your Sharks
Name: Megalodon Shark
Character: Ancient
Name: Whale Shark
Character: Large

Complete Shark Collection

Prerequisites

To follow this tutorial, you will need: - A development server running Ubuntu 18.04, along with a non-root user with `sudo` privileges and an active firewall. For guidance on how to set these up, please see this Initial Server Setup guide. - Docker installed on your server, following Steps 1 and 2 of How To Install and Use Docker on Ubuntu 18.04. - Docker Compose installed on your server, following Step 1 of How To Install Docker Compose on Ubuntu 18.04.

Step 1 — Cloning the Project and Modifying Dependencies

The first step in building this setup will be cloning the project code and modifying its `package.json` file, which includes the project's dependencies. We will add `nodemon` to the project's `devDependencies`, specifying that we will be using it during development. Running the application with `nodemon` ensures that it will be automatically restarted whenever you make changes to your code.

First, clone the `nodejs-mongo-mongoose` repository from the DigitalOcean Community GitHub account. This repository includes the code from the setup described in How To Integrate MongoDB with Your Node Application, which explains how to integrate a MongoDB database with an existing Node application using Mongoose.

Clone the repository into a directory called **`node_project`**:

```
git clone https://github.com/do-community/nodejs-mongo-mongoose.git node_project
```

Navigate to the **`node_project`** directory:

```
cd node_project
```

Open the project's `package.json` file using `nano` or your favorite editor:

```
nano package.json
```

Beneath the project dependencies and above the closing curly brace, create a new `devDependencies` object that includes `nodemon`:

~/node_project/package.json

```
...
"dependencies": {
    "ejs": "^2.6.1",
    "express": "^4.16.4",
    "mongoose": "^5.4.10"
  },
  "devDependencies": {
    "nodemon": "^1.18.10"
  }
}
```

Save and close the file when you are finished editing.

With the project code in place and its dependencies modified, you can move on to refactoring the code for a containerized workflow.

Step 2 — Configuring Your Application to Work with Containers

Modifying our application for a containerized workflow means making our code more modular. Containers offer portability between environments, and our code should reflect that by remaining as decoupled from the underlying operating system as possible. To achieve this, we will refactor our code to make greater use of Node's process.env property, which returns an object with information about your user environment at runtime. We can use this object in our code to dynamically assign configuration information at runtime with environment variables.

Let’s begin with `app.js`, our main application entrypoint. Open the file:

```
nano app.js
```

Inside, you will see a definition for a `port` constant, as well a `listen` function that uses this constant to specify the port the application will listen on:

~/home/node_project/app.js

```
...
const port = 8080;
...
app.listen(port, function () {
  console.log('Example app listening on port 8080!');
});
```

Let’s redefine the `port` constant to allow for dynamic assignment at runtime using the `process.env` object. Make the following changes to the constant definition and `listen` function:

~/home/node_project/app.js

```
...
const port = process.env.PORT || 8080;
...
app.listen(port, function () {
  console.log(`Example app listening on ${port}!`);
});
```

Our new constant definition assigns `port` dynamically using the value passed in at runtime or `8080`. Similarly, we've rewritten the `listen` function to use a template literal, which will interpolate the port value when listening for connections. Because we will be mapping our ports elsewhere, these revisions will prevent our having to continuously revise this file as our environment changes.

When you are finished editing, save and close the file.

Next, we will modify our database connection information to remove any configuration credentials. Open the `db.js` file, which contains this information:

```
nano db.js
```

Currently, the file does the following things: - Imports Mongoose, the Object Document Mapper (ODM) that we're using to create schemas and models for our application data. - Sets the database credentials as constants, including the username and password. - Connects to the database using the `mongoose.connect` method.

For more information about the file, please see Step 3 of How To Integrate MongoDB with Your Node Application.

Our first step in modifying the file will be redefining the constants that include sensitive information. Currently, these constants look like this:

~/node_project/db.js

```
...
const MONGO_USERNAME = 'sammy';
const MONGO_PASSWORD = 'your_password';
const MONGO_HOSTNAME = '127.0.0.1';
const MONGO_PORT = '27017';
const MONGO_DB = 'sharkinfo';
...
```

Instead of hardcoding this information, you can use the `process.env` object to capture the runtime values for these constants. Modify the block to look like this:

~/node_project/db.js

```
...
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;
...
```

Save and close the file when you are finished editing.

At this point, you have modified `db.js` to work with your application's environment variables, but you still need a way to pass these variables to your application. Let's create an `.env` file with values that you can pass to your application at runtime.

Open the file:

```
nano .env
```

This file will include the information that you removed from `db.js`: the username and password for your application's database, as well as the port setting and database name. Remember to update the username, password, and database name listed here with your own information:

~/node_project/.env

```
MONGO_USERNAME=sammy
MONGO_PASSWORD=your_password
MONGO_PORT=27017
MONGO_DB=sharkinfo
```

Note that we have removed the host setting that originally appeared in `db.js`. We will now define our host at the level of the Docker Compose file, along with other information about our services and containers.

Save and close this file when you are finished editing.

Because your `.env` file contains sensitive information, you will want to ensure that it is included in your project's `.dockerignore` and `.gitignore` files so that it does not copy to your version control or containers.

Open your `.dockerignore` file:

```
nano .dockerignore
```

Add the following line to the bottom of the file:

~/node_project/.dockerignore

```
...
.gitignore
.env
```

Save and close the file when you are finished editing.

The `.gitignore` file in this repository already includes `.env`, but feel free to check that it is there:

```
nano .gitignore
```

~~/node_project/.gitignore

```
...
.env
...
```

At this point, you have successfully extracted sensitive information from your project code and taken measures to control how and where this information gets copied. Now you can add more robustness to your database connection code to optimize it for a containerized workflow.

Step 3 — Modifying Database Connection Settings

Our next step will be to make our database connection method more robust by adding code that handles cases where our application fails to connect to our database. Introducing this level of resilience to your application code is a recommended practice when working with containers using Compose.

Open `db.js` for editing:

```
nano db.js
```

You will see the code that we added earlier, along with the `url` constant for Mongo's connection URI and the Mongoose `connect` method:

~/node_project/db.js

```
...
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;
```

```
const url =
`mongodb://${MONGO_USERNAME}:${MONGO_PASSWORD}@${M
ONGO_HOSTNAME}:${MONGO_PORT}/${MONGO_DB}?
authSource=admin`;

mongoose.connect(url, {useNewUrlParser: true});
```

Currently, our `connect` method accepts an option that tells Mongoose to use Mongo's new URL parser. Let's add a few more options to this method to define parameters for reconnection attempts. We can do this by creating an `options` constant that includes the relevant information, in addition to the new URL parser option. Below your Mongo constants, add the following definition for an `options` constant:

~/node_project/db.js

```
...
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;

const options = {
  useNewUrlParser: true,
  reconnectTries: Number.MAX_VALUE,
  reconnectInterval: 500,
  connectTimeoutMS: 10000,
};
...
```

The `reconnectTries` option tells Mongoose to continue trying to connect indefinitely, while `reconnectInterval` defines the period between connection attempts in milliseconds. `connectTimeoutMS` defines 10 seconds as the period that the Mongo driver will wait before failing the connection attempt.

We can now use the new `options` constant in the Mongoose `connect` method to fine tune our Mongoose connection settings. We will also add a promise to handle potential connection errors.

Currently, the Mongoose `connect` method looks like this:

~/node_project/db.js

```
...
mongoose.connect(url, {useNewUrlParser: true});
```

Delete the existing `connect` method and replace it with the following code, which includes the `options` constant and a promise:

~/node_project/db.js

```
...
mongoose.connect(url, options).then( function() {
  console.log('MongoDB is connected');
})
  .catch( function(err) {
```

```
  console.log(err);
});
```

In the case of a successful connection, our function logs an appropriate message; otherwise it will catch and log the error, allowing us to troubleshoot.

The finished file will look like this:

~/node_project/db.js

```
const mongoose = require('mongoose');

const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;

const options = {
  useNewUrlParser: true,
  reconnectTries: Number.MAX_VALUE,
  reconnectInterval: 500,
  connectTimeoutMS: 10000,
};

const url =
`mongodb://${MONGO_USERNAME}:${MONGO_PASSWORD}@${M
ONGO_HOSTNAME}:${MONGO_PORT}/${MONGO_DB}?
authSource=admin`;

mongoose.connect(url, options).then( function() {
  console.log('MongoDB is connected');
})
  .catch( function(err) {
  console.log(err);
});
```

Save and close the file when you have finished editing.

You have now added resiliency to your application code to handle cases where your application might fail to connect to your database. With this code in place, you can move on to defining your services with Compose.

Step 4 — Defining Services with Docker Compose

With your code refactored, you are ready to write the `docker-compose.yml` file with your service definitions. A service in Compose is a running container, and service definitions — which you will include in your `docker-compose.yml` file — contain information about how each container image will run. The Compose tool allows you to define multiple services to build multi-container applications.

Before defining our services, however, we will add a tool to our project called [`wait-for`]() to ensure that our application only attempts to connect to our database once the database startup tasks are complete. This wrapper script uses [`netcat`]() to poll whether or not a specific host and port are accepting TCP connections. Using it allows you to control your application's attempts to connect to your database by testing whether or not the database is ready to accept connections.

Though Compose allows you to specify dependencies between services using the `depends_on` option, this order is based on whether or not the container is running rather than its readiness. Using `depends_on` won't be optimal for our setup, since we want our application to connect only when the database startup tasks, including adding a user and password to the `admin` authentication database, are complete. For more information on using `wait-for` and other tools to control startup order, please see the relevant recommendations in the Compose documentation.

Open a file called `wait-for.sh`:

```
nano wait-for.sh
```

Paste the following code into the file to create the polling function:

~/node_project/app/wait-for.sh

```
#!/bin/sh

# original script:
https://github.com/eficode/wait-
for/blob/master/wait-for

TIMEOUT=15
QUIET=0

echoerr() {
  if [ "$QUIET" -ne 1 ]; then printf "%s\n" "$*"
1>&2; fi
```

```
}

usage() {
  exitcode="$1"
  cat << USAGE >&2
Usage:
  $cmdname host:port [-t timeout] [-- command
args]
  -q | --quiet                          Do not
output any status messages
  -t TIMEOUT | --timeout=timeout        Timeout in
seconds, zero for no timeout
  -- COMMAND ARGS                       Execute
command with args after the test finishes
USAGE
  exit "$exitcode"
}

wait_for() {
  for i in `seq $TIMEOUT` ; do
    nc -z "$HOST" "$PORT" > /dev/null 2>&1

    result=$?
    if [ $result -eq 0 ] ; then
      if [ $# -gt 0 ] ; then
        exec "$@"
      fi
      exit 0
    fi
    sleep 1
  done
  echo "Operation timed out" >&2
  exit 1
}

while [ $# -gt 0 ]
do
  case "$1" in
    *:* )
    HOST=$(printf "%s\n" "$1"| cut -d : -f 1)
    PORT=$(printf "%s\n" "$1"| cut -d : -f 2)
    shift 1
    ;;
    -q | --quiet)
    QUIET=1
    shift 1
    ;;
    -t)
    TIMEOUT="$2"
    if [ "$TIMEOUT" = "" ]; then break; fi
    shift 2
    ;;
    --timeout=*)
    TIMEOUT="${1#*=}"
    shift 1
    ;;
    --)
    shift
    break
    ;;
    --help)
    usage 0
    ;;
    *)
    echoerr "Unknown argument: $1"
    usage 1
    ;;
  esac
done

if [ "$HOST" = "" -o "$PORT" = "" ]; then
  echoerr "Error: you need to provide a host and
port to test."
  usage 2
fi

wait_for "$@"
```

Save and close the file when you are finished adding the code.

Make the script executable:

```
chmod +x wait-for.sh
```

Next, open the `docker-compose.yml` file:

```
nano docker-compose.yml
```

First, define the `nodejs` application service by adding the following code to the file:

~/node_project/docker-compose.yml

```
version: '3'

services:
  nodejs:
    build:
      context: .
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
      - MONGO_PASSWORD=$MONGO_PASSWORD
      - MONGO_HOSTNAME=db
      - MONGO_PORT=$MONGO_PORT
      - MONGO_DB=$MONGO_DB
    ports:
      - "80:8080"
    volumes:
      - .:/home/node/app
      - node_modules:/home/node/app/node_modules
    networks:
      - app-network
    command: ./wait-for.sh db:27017 --
/home/node/app/node_modules/.bin/nodemon app.js
```

The `nodejs` service definition includes the following options: - `build`: This defines the configuration options, including the `context` and `dockerfile`, that will be applied when Compose builds the application image. If you wanted to use an existing image from a registry like Docker Hub, you could use the `image` instruction instead, with information about your username, repository, and image tag. - `context`: This defines the build context for the image build — in this case, the current project directory. - `dockerfile`: This specifies the `Dockerfile` in your current project directory as the file Compose will use to build the application image. For more information about this file, please see How To Build a Node.js Application with Docker. - `image`, `container_name`: These apply names to the image and container. - `restart`: This defines the restart policy. The default is `no`, but we have set the container to restart unless it is stopped. - `env_file`: This tells Compose that we would like to add environment variables from a file called `.env`, located in our build context. - `environment`: Using this option allows you to add the Mongo connection settings you defined in the `.env` file. Note that we are not setting `NODE_ENV` to `development`, since this is Express's default behavior if `NODE_ENV` is not set. When moving to production, you can set this to `production` to enable view caching and less verbose error messages. Also note that we have specified the `db` database container as the host, as discussed in Step 2. - `ports`: This maps port `80` on the host to port `8080` on the container. - `volumes`:

We are including two types of mounts here: - The first is a bind mount that mounts our application code on the host to the `/home/node/app` directory on the container. This will facilitate rapid development, since any changes you make to your host code will be populated immediately in the container. - The second is a named volume, `node_modules`. When Docker runs the `npm install` instruction listed in the application `Dockerfile`, `npm` will create a new `node_modules` directory on the container that includes the packages required to run the application. The bind mount we just created will hide this newly created `node_modules` directory, however. Since `node_modules` on the host is empty, the bind will map an empty directory to the container, overriding the new `node_modules` directory and preventing our application from starting. The named `node_modules` volume solves this problem by persisting the contents of the `/home/node/app/node_modules` directory and mounting it to the container, hiding the bind.

```
**Keep the following points in mind when using
this approach**:
- Your bind will mount the contents of the
`node_modules` directory on the container to the
host and this directory will be owned by `root`,
since the named volume was created by Docker.
- If you have a pre-existing `node_modules`
directory on the host, it will override the
`node_modules` directory created on the container.
The setup that we're building in this tutorial
assumes that you do **not** have a pre-existing
`node_modules` directory and that you won't be
```

```
working with `npm` on your host. This is in
keeping with a [twelve-factor approach to
application development](https://12factor.net/),
which minimizes dependencies between execution
environments.
```

- `networks`: This specifies that our application service will join the `app-network` network, which we will define at the bottom on the file.
- `command`: This option lets you set the command that should be executed when Compose runs the image. Note that this will override the `CMD` instruction that we set in our application `Dockerfile`. Here, we are running the application using the `wait-for` script, which will poll the `db` service on port `27017` to test whether or not the database service is ready. Once the readiness test succeeds, the script will execute the command we have set, `/home/node/app/node_modules/.bin/nodemon app.js`, to start the application with `nodemon`. This will ensure that any future changes we make to our code are reloaded without our having to restart the application.

Next, create the `db` service by adding the following code below the application service definition:

~/node_project/docker-compose.yml

```
...
  db:
    image: mongo:4.1.8-xenial
    container_name: db
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_INITDB_ROOT_USERNAME=$MONGO_USERNAME
      - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASSWORD
    volumes:
      - dbdata:/data/db
    networks:
      - app-network
```

Some of the settings we defined for the `nodejs` service remain the same, but we've also made the following changes to the `image`, `environment`, and `volumes` definitions: - `image`: To create this service, Compose will pull the `4.1.8-xenial` Mongo image from Docker Hub. We are pinning a particular version to avoid possible future conflicts as the Mongo image changes. For more information about version pinning, please see the Docker documentation on Dockerfile best practices. - `MONGO_INITDB_ROOT_USERNAME`, `MONGO_INITDB_ROOT_PASSWORD`: The `mongo` image makes these environment variables available so that you can modify the initialization of your database instance. `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` together create a `root` user in the `admin` authentication database and ensure that authentication is enabled when the container starts. We have set `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` using the values from our `.env` file, which we pass to the `db` service using the `env_file` option. Doing this means that our `sammy` application user will be a `root` user on the database instance, with access to all of the administrative and operational privileges of that role. When working in production, you will want to create a dedicated application user with appropriately scoped privileges.

Note: Keep in mind that these variables will not take effect if you start the container with an existing data directory in place.

- `dbdata:/data/db`: The named volume `dbdata` will persist the data stored in Mongo's default data directory, `/data/db`. This will ensure that you don't lose data in cases where you stop or remove containers.

We've also added the `db` service to the `app-network` network with the `networks` option.

As a final step, add the volume and network definitions to the bottom of the file:

~/node_project/docker-compose.yml

```
...
networks:
  app-network:
    driver: bridge

volumes:
  dbdata:
  node_modules:
```

The user-defined bridge network `app-network` enables communication between our containers since they are on the same Docker daemon host. This streamlines traffic and communication within the application, as it opens all ports between containers on the same bridge network, while exposing no ports to the outside world. Thus, our `db` and `nodejs` containers can communicate with each other, and we only need to expose port `80` for front-end access to the application.

Our top-level `volumes` key defines the volumes `dbdata` and `node_modules`. When Docker creates volumes, the contents of the volume are stored in a part of the host filesystem, `/var/lib/docker/volumes/`, that's managed by Docker. The contents of each volume are stored in a directory under `/var/lib/docker/volumes/` and get mounted to any container that uses the volume. In this way, the shark information data that our users will create will persist in the `dbdata` volume even if we remove and recreate the `db` container.

The finished `docker-compose.yml` file will look like this:

~/node_project/docker-compose.yml

```
version: '3'

services:
  nodejs:
    build:
      context: .
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
      - MONGO_PASSWORD=$MONGO_PASSWORD
      - MONGO_HOSTNAME=db
      - MONGO_PORT=$MONGO_PORT
      - MONGO_DB=$MONGO_DB
    ports:
      - "80:8080"
    volumes:
      - .:/home/node/app
      - node_modules:/home/node/app/node_modules
    networks:
      - app-network
    command: ./wait-for.sh db:27017 --
/home/node/app/node_modules/.bin/nodemon app.js

  db:
    image: mongo:4.1.8-xenial
    container_name: db
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_INITDB_ROOT_USERNAME=$MONGO_USERNAME
      - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASSWORD
    volumes:
      - dbdata:/data/db
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  dbdata:
  node_modules:
```

Save and close the file when you are finished editing.

With your service definitions in place, you are ready to start the application.

Step 5 — Testing the Application

With your `docker-compose.yml` file in place, you can create your services with the [`docker-compose up`] command. You can also test that your data will persist by stopping and removing your containers with [`docker-compose down`].

First, build the container images and create the services by running `docker-compose up` with the `-d` flag, which will then run the `nodejs` and `db` containers in the background:

```
docker-compose up -d
```

You will see output confirming that your services have been created:

Output

```
...
Creating db ... done
Creating nodejs ... done
```

You can also get more detailed information about the startup processes by displaying the log output from the services:

```
docker-compose logs
```

You will see something like this if everything has started correctly:

Output

```
...
nodejs    | [nodemon] starting `node app.js`
nodejs    | Example app listening on 8080!
nodejs    | MongoDB is connected
...
db        | 2019-02-22T17:26:27.329+0000 I ACCESS
[conn2] Successfully authenticated as principal
sammy on admin
```

You can also check the status of your containers with docker-compose ps:

```
docker-compose ps
```

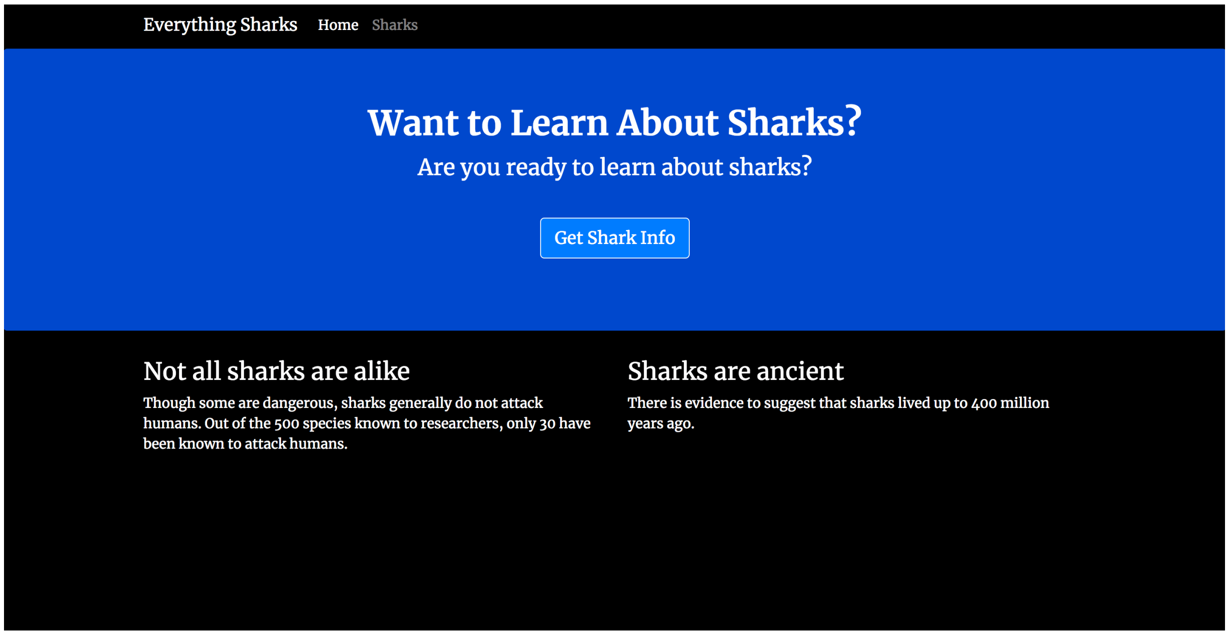
You will see output indicating that your containers are running:

Output

```
 Name                 Command                      State
Ports
--------------------------------------------------
--------------------
```

```
db        docker-entrypoint.sh mongod      Up
27017/tcp
nodejs    ./wait-for.sh db:27017 --  ...   Up
0.0.0.0:80->8080/tcp
```

With your services running, you can visit `http://your_server_ip` in the browser. You will see a landing page that looks like this:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

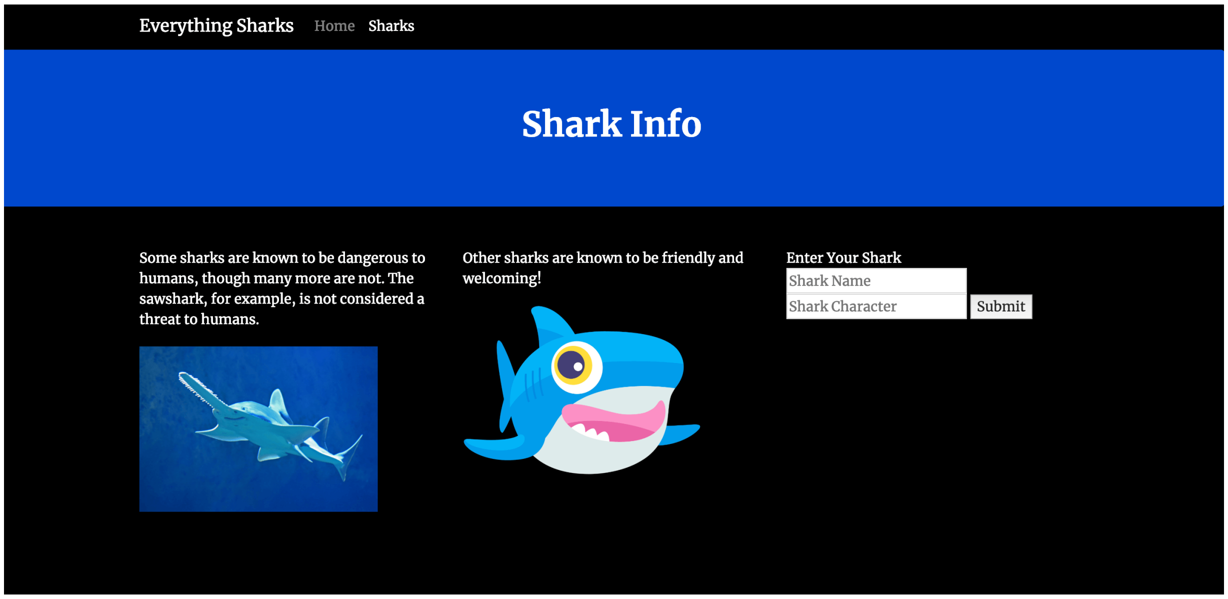
Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Application Landing Page

Click on the Get Shark Info button. You will see a page with an entry form where you can enter a shark name and a description of that shark's general character:


Everything Sharks Home Sharks

Shark Info

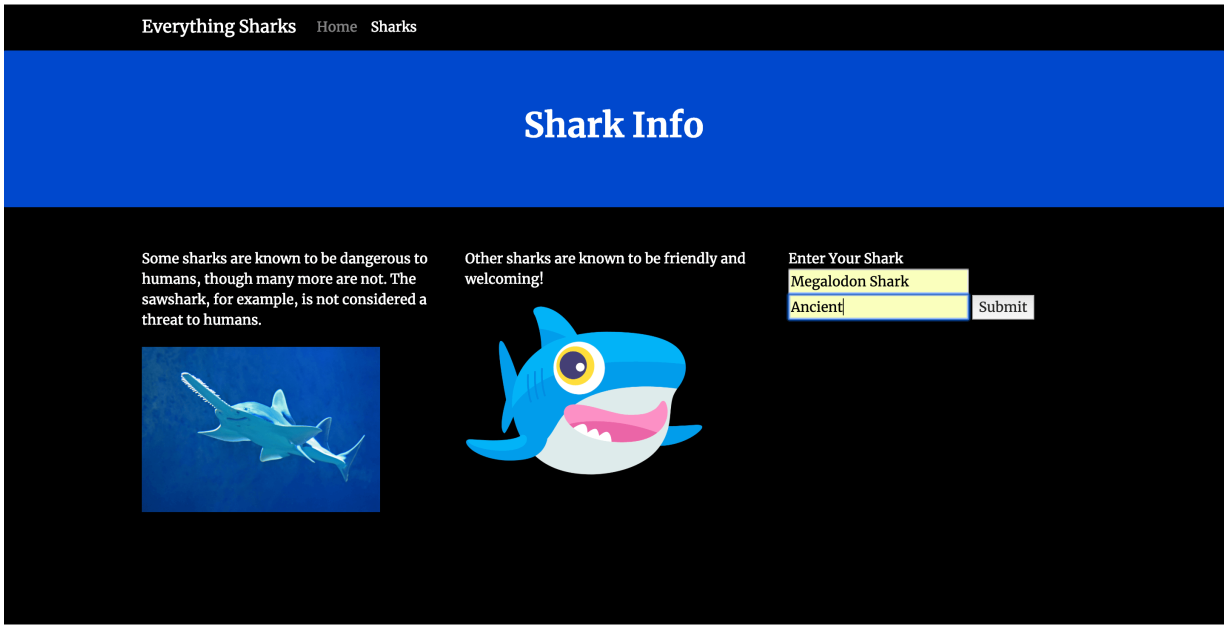
Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

Shark Info Form

In the form, add a shark of your choosing. For the purpose of this demonstration, we will add `Megalodon Shark` to the Shark Name field, and `Ancient` to the Shark Character field:



Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

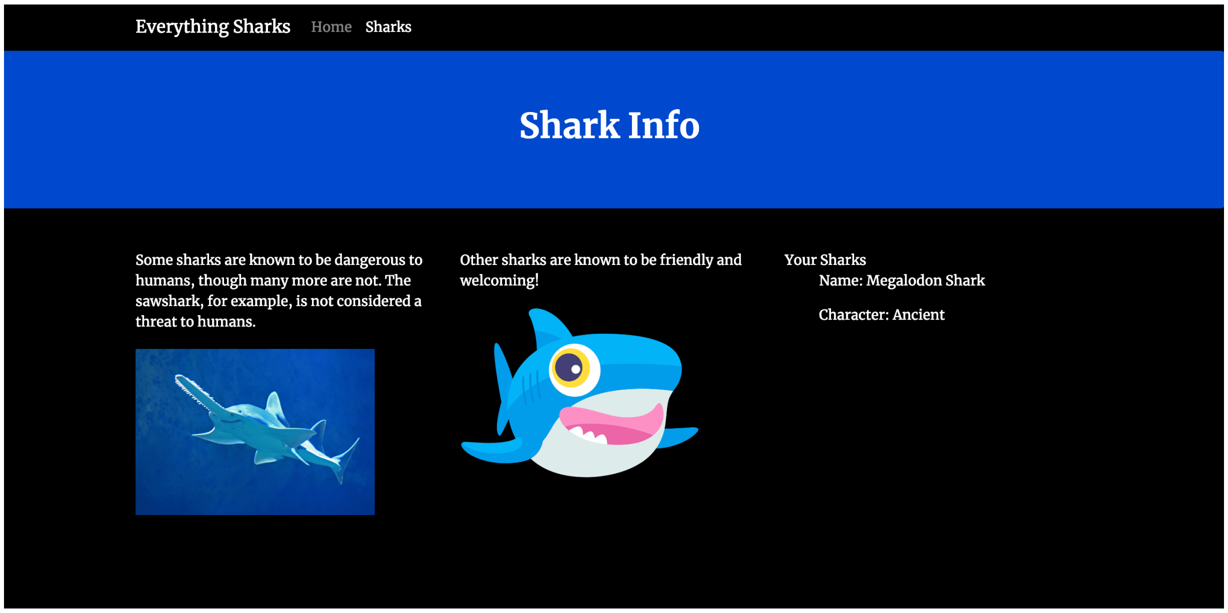
Enter Your Shark

Megalodon Shark

Ancient

Filled Shark Form

Click on the Submit button. You will see a page with this shark information displayed back to you:



Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Your Sharks

Name: Megalodon Shark

Character: Ancient

Shark Output

As a final step, we can test that the data you've just entered will persist if you remove your database container.

Back at your terminal, type the following command to stop and remove your containers and network:

```
docker-compose down
```

Note that we are not including the `--volumes` option; hence, our `dbdata` volume is not removed.

The following output confirms that your containers and network have been removed:

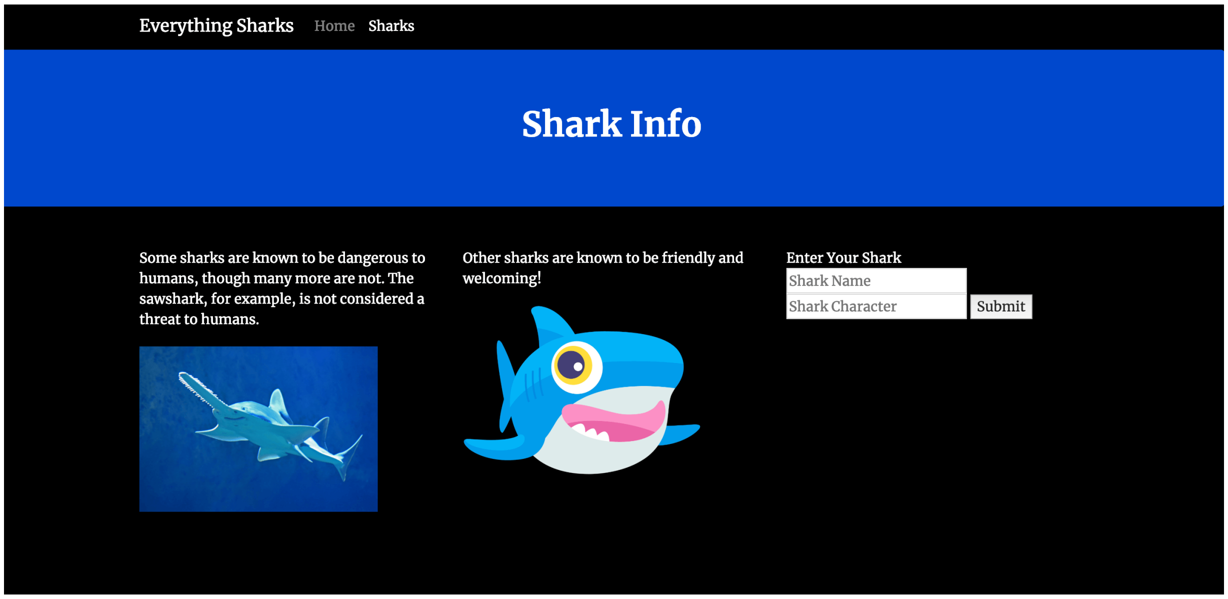
Output

```
Stopping nodejs ... done
Stopping db     ... done
Removing nodejs ... done
Removing db     ... done
Removing network node_project_app-network
```

Recreate the containers:

```
docker-compose up -d
```

Now head back to the shark information form:


Everything Sharks Home Sharks

Shark Info

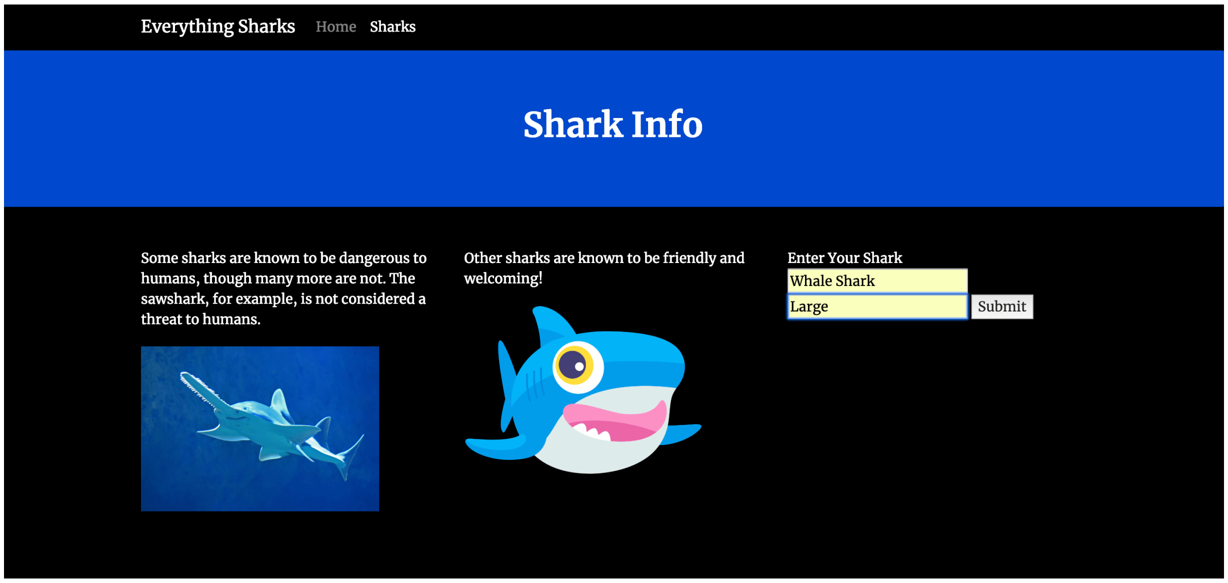
Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

Shark Info Form

Enter a new shark of your choosing. We'll go with `Whale Shark` and `Large`:


Everything Sharks Home Sharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

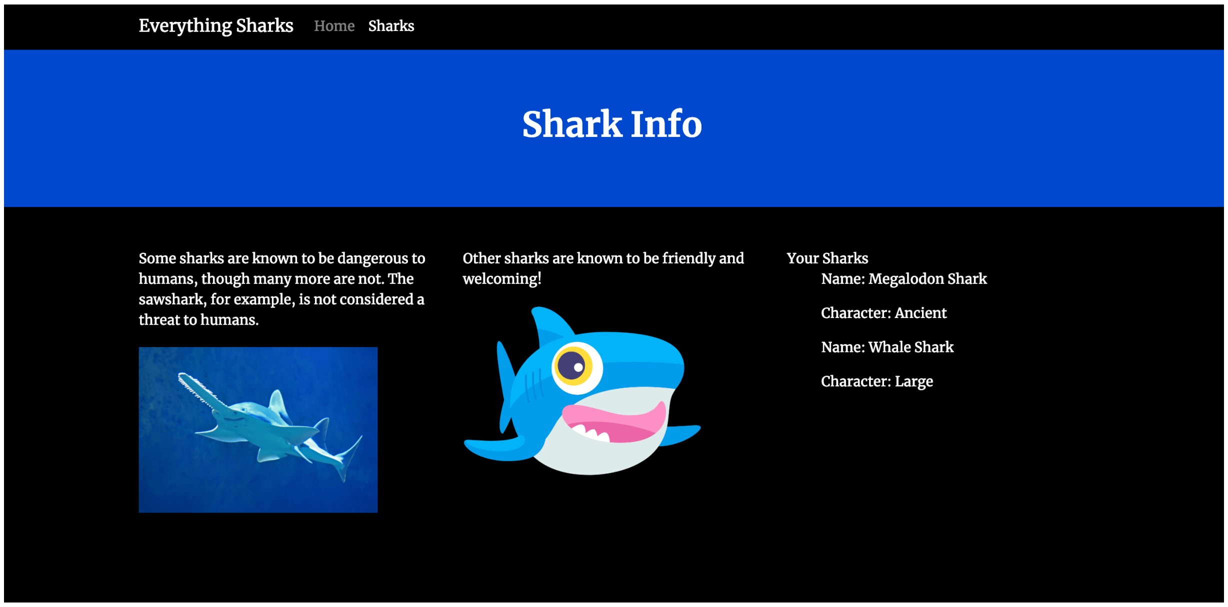
Enter Your Shark

Whale Shark

Large

Enter New Shark

Once you click Submit, you will see that the new shark has been added to the shark collection in your database without the loss of the data you've already entered:



Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Your Sharks
Name: Megalodon Shark
Character: Ancient
Name: Whale Shark
Character: Large

Complete Shark Collection

Your application is now running on Docker containers with data persistence and code synchronization enabled.

Conclusion

By following this tutorial, you have created a development setup for your Node application using Docker containers. You've made your project more modular and portable by extracting sensitive information and decoupling your application's state from your application code. You have also configured a boilerplate `docker-compose.yml` file that you can revise as your development needs and requirements change.

As you develop, you may be interested in learning more about designing applications for containerized and Cloud Native workflows. Please see Architecting Applications for Kubernetes and Modernizing Applications for Kubernetes for more information on these topics.

To learn more about the code used in this tutorial, please see How To Build a Node.js Application with Docker and How To Integrate MongoDB with Your Node Application. For information about deploying a Node application with an Nginx reverse proxy using containers, please see How To Secure a Containerized Node.js Application with Nginx, Let’s Encrypt, and Docker Compose.

How to Set Up DigitalOcean Kubernetes Cluster Monitoring with Helm and Prometheus Operator

Written by Eddie Zaneski and Hanif Jetha

Along with tracing and logging, monitoring and alerting are essential components of a Kubernetes observability stack. Setting up monitoring for your Kubernetes cluster allows you to track your resource usage and analyze and debug application errors.

One popular monitoring solution is the open-source Prometheus, Grafana, and Alertmanager stack. In this tutorial you will learn how to use the Helm package manager for Kubernetes to install all three of these monitoring tools into your Kubernetes cluster.

By the end of this tutorial, you will have cluster monitoring set up with a standard set of dashboards to view graphs and health metrics for your cluster, Prometheus rules for collecting health data, and alerts to notify you when something is not performing or behaving properly.

Along with tracing and logging, monitoring and alerting are essential components of a Kubernetes observability stack. Setting up monitoring for your Kubernetes cluster allows you to track your resource usage and analyze and debug application errors.

A monitoring system usually consists of a time-series database that houses metric data and a visualization layer. In addition, an alerting layer creates and manages alerts, handing them off to integrations and external services as necessary. Finally, one or more components generate or expose the metric data that will be stored, visualized, and processed for alerts by this monitoring stack.

One popular monitoring solution is the open-source Prometheus, Grafana, and Alertmanager stack:

- Prometheus is a time series database and monitoring tool that works by polling metrics endpoints and scraping and processing the data exposed by these endpoints. It allows you to query this data using PromQL, a time series data query language.
- Grafana is a data visualization and analytics tool that allows you to build dashboards and graphs for your metrics data.
- Alertmanager, usually deployed alongside Prometheus, forms the alerting layer of the stack, handling alerts generated by Prometheus and deduplicating, grouping, and routing them to integrations like email or PagerDuty.

In addition, tools like kube-state-metrics and node_exporter expose cluster-level Kubernetes object metrics as well as machine-level metrics like CPU and memory usage.

Implementing this monitoring stack on a Kubernetes cluster can be complicated, but luckily some of this complexity can be managed with the Helm package manager and CoreOS's Prometheus Operator and kube-prometheus projects. These projects bake in standard configurations and dashboards for Prometheus and Grafana, and abstract away some of the lower-level Kubernetes object definitions. The Helm `prometheus-operator` chart allows you to get a full cluster monitoring solution up and running by installing Prometheus Operator and the rest of the components listed above, along with a default set of dashboards, rules, and alerts useful for monitoring Kubernetes clusters.

In this tutorial, we will demonstrate how to install the `prometheus-operator` Helm chart on a DigitalOcean Kubernetes cluster. By the end of the tutorial, you will have installed a full monitoring stack into your cluster.

Prerequisites

To follow this tutorial, you will need:

- A DigitalOcean Kubernetes cluster.
- The `kubectl` command-line interface installed on your local machine and configured to connect to your cluster. You can read more about installing and configuring `kubectl` in its official documentation.
- The Helm package manager (2.10+) installed on your local machine and Tiller installed on your cluster, as detailed in How To Install Software on Kubernetes Clusters with the Helm Package Manager.

Step 1 — Creating a Custom Values File

Before we install the `prometheus-operator` Helm chart, we'll create a custom values file that will override some of the chart's defaults with DigitalOcean-specific configuration parameters. To learn more about overriding default chart values, consult the Helm Install section of the Helm docs.

To begin, create and open a file called `custom-values.yaml` on your local machine using `nano` or your favorite editor:

```
nano custom-values.yaml
```

Copy and paste in the following custom values, which enable persistent storage for the Prometheus, Grafana, and Alertmananger components, and disable monitoring for Kubernetes control plane components not exposed on DigitalOcean Kubernetes:

custom-values.yaml

```
# Define persistent storage for Prometheus (PVC)
prometheus:
  prometheusSpec:
    storageSpec:
      volumeClaimTemplate:
        spec:
          accessModes: ["ReadWriteOnce"]
          storageClassName: do-block-storage
          resources:
            requests:
              storage: 5Gi

# Define persistent storage for Grafana (PVC)
grafana:
  # Set password for Grafana admin user
  adminPassword: your_admin_password
  persistence:
    enabled: true
    storageClassName: do-block-storage
    accessModes: ["ReadWriteOnce"]
    size: 5Gi

# Define persistent storage for Alertmanager (PVC)
alertmanager:
  alertmanagerSpec:
    storage:
      volumeClaimTemplate:
        spec:
          accessModes: ["ReadWriteOnce"]
          storageClassName: do-block-storage
          resources:
            requests:
              storage: 5Gi

# Change default node-exporter port
prometheus-node-exporter:
  service:
    port: 30206
    targetPort: 30206

# Disable Etcd metrics
kubeEtcd:
  enabled: false

# Disable Controller metrics
kubeControllerManager:
  enabled: false

# Disable Scheduler metrics
kubeScheduler:
  enabled: false
```

In this file, we override some of the default values packaged with the chart in its values.yaml file.

We first enable persistent storage for Prometheus, Grafana, and Alertmanager so that their data persists across Pod restarts. Behind the scenes, this defines a `5 Gi` Persistent Volume Claim (PVC) for each component, using the DigitalOcean Block Storage storage class. You should modify the size of these PVCs to suit your monitoring storage needs. To learn more about PVCs, consult Persistent Volumes from the official Kubernetes docs.

Next, replace **`your_admin_password`** with a secure password that you'll use to log in to the Grafana metrics dashboard with the admin user.

We'll then configure a different port for node-exporter. Node-exporter runs on each Kubernetes node and provides OS and hardware metrics to Prometheus. We must change its default port to get around the DigitalOcean Kubernetes firewall defaults, which will block port 9100 but allow ports in the range 30000-32767. Alternatively, you can configure a custom firewall rule for node-exporter. To learn how, consult How to Configure Firewall Rules from the official DigitalOcean Cloud Firewalls docs.

Finally, we'll disable metrics collection for three Kubernetes control plane components that do not expose metrics on DigitalOcean Kubernetes:

the Kubernetes Scheduler and Controller Manager, and etcd cluster data store.

To see the full list of configurable parameters for the `prometheus-operator` chart, consult the Configuration section from the chart repo README or the default values file.

When you're done editing, save and close the file. We can now install the chart using Helm.

Step 2 — Installing the `prometheus-operator` Chart

The `prometheus-operator` Helm chart will install the following monitoring components into your DigitalOcean Kubernetes cluster:

- Prometheus Operator, a Kubernetes Operator that allows you to configure and manage Prometheus clusters. Kubernetes Operators integrate domain-specific logic into the process of packaging, deploying, and managing applications with Kubernetes. To learn more about Kubernetes Operators, consult the CoreOS Operators Overview. To learn more about Prometheus Operator, consult this introductory post on the Prometheus Operator and the Prometheus Operator GitHub repo. Prometheus Operator will be installed as a Deployment.
- Prometheus, installed as a StatefulSet.
- Alertmanager, a service that handles alerts sent by the Prometheus server and routes them to integrations like PagerDuty or email. To learn more about Alertmanager, consult Alerting from the Prometheus docs. Alertmanager will be installed as a StatefulSet.

- Grafana, a time series data visualization tool that allows you to visualize and create dashboards for your Prometheus metrics. Grafana will be installed as a Deployment.
- node-exporter, a Prometheus exporter that runs on cluster nodes and provides OS and hardware metrics to Prometheus. Consult the node-exporter GitHub repo to learn more. node-exporter will be installed as a DaemonSet.
- kube-state-metrics, an add-on agent that listens to the Kubernetes API server and generates metrics about the state of Kubernetes objects like Deployments and Pods. You can learn more by consulting the kube-state-metrics GitHub repo. kube-state-metrics will be installed as a Deployment.

By default, along with scraping metrics generated by node-exporter, kube-state-metrics, and the other components listed above, Prometheus will be configured to scrape metrics from the following components:

- kube-apiserver, the Kubernetes API server.
- CoreDNS, the Kubernetes cluster DNS server.
- kubelet, the primary node agent that interacts with kube-apiserver to manage Pods and containers on a node.
- cAdvisor, a node agent that discovers running containers and collects their CPU, memory, filesystem, and network usage metrics.

On your local machine, let's begin by installing the `prometheus-operator` Helm chart and passing in the custom values file we created above:

```
helm install --namespace monitoring --name doks-
cluster-monitoring -f custom-values.yaml
stable/prometheus-operator
```

Here we run `helm install` and install all components into the `monitoring` namespace, which we create at the same time. This allows us to cleanly separate the monitoring stack from the rest of the Kubernetes cluster. We name the Helm release `doks-cluster-monitoring` and pass in the custom values file we created in Step 1. Finally, we specify that we'd like to install the `prometheus-operator` chart from the Helm `stable` directory.

You should see the following output:

Output

```
NAME:   doks-cluster-monitoring
LAST DEPLOYED: Mon Apr 22 10:30:42 2019
NAMESPACE: monitoring
STATUS: DEPLOYED

RESOURCES:
==> v1/PersistentVolumeClaim
NAME                                 STATUS   VOLUME
CAPACITY  ACCESS MODES  STORAGECLASS  AGE
doks-cluster-monitoring-grafana  Pending  do-
block-storage  10s

==> v1/ServiceAccount
NAME
```

```
SECRETS  AGE
doks-cluster-monitoring-grafana
1        10s
doks-cluster-monitoring-kube-state-metrics
1        10s

. . .

==> v1beta1/ClusterRoleBinding
NAME
AGE
doks-cluster-monitoring-kube-state-metrics
9s
psp-doks-cluster-monitoring-prometheus-node-
exporter  9s



NOTES:
The Prometheus Operator has been installed. Check
its status by running:
  kubectl --namespace monitoring get pods -l
"release=doks-cluster-monitoring"

Visit https://github.com/coreos/prometheus-
operator for instructions on how
to create & configure Alertmanager and Prometheus
instances using the Operator.
```

This indicates that Prometheus Operator, Prometheus, Grafana, and the other components listed above have successfully been installed into your DigitalOcean Kubernetes cluster.

Following the note in the `helm install` output, check the status of the release's Pods using `kubectl get pods`:

```
kubectl --namespace monitoring get pods -l
"release=doks-cluster-monitoring"
```

You should see the following:

Output

```
NAME
READY   STATUS    RESTARTS   AGE
doks-cluster-monitoring-grafana-9d7f984c5-hxnw6
2/2     Running   0          3m36s
doks-cluster-monitoring-kube-state-metrics-
dd8557f6b-9rl7j   1/1     Running   0
3m36s
doks-cluster-monitoring-pr-operator-9c5b76d78-
9kj85          1/1     Running   0          3m36s
doks-cluster-monitoring-prometheus-node-exporter-
2qvxw       1/1     Running   0          3m36s
doks-cluster-monitoring-prometheus-node-exporter-
7brwv       1/1     Running   0          3m36s
doks-cluster-monitoring-prometheus-node-exporter-
jhdgz       1/1     Running   0          3m36s
```

This indicates that all the monitoring components are up and running, and you can begin exploring Prometheus metrics using Grafana and its preconfigured dashboards.

Step 3 — Accessing Grafana and Exploring Metrics Data

The `prometheus-operator` Helm chart exposes Grafana as a `ClusterIP` Service, which means that it's only accessible via a cluster-internal IP address. To access Grafana outside of your Kubernetes cluster, you can either use `kubectl patch` to update the Service in place to a public-facing type like `NodePort` or `LoadBalancer`, or `kubectl port-forward` to forward a local port to a Grafana Pod port.

In this tutorial we'll forward ports, but to learn more about `kubectl patch` and Kubernetes Service types, you can consult [Update API Objects in Place Using kubectl patch]() and [Services]() from the official Kubernetes docs.

Begin by listing running Services in the `monitoring` namespace:

```
kubectl get svc -n monitoring
```

You should see the following Services:

Output

```
NAME
TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)
AGE
alertmanager-operated
ClusterIP   None            <none>
9093/TCP,6783/TCP   34m
doks-cluster-monitoring-grafana
ClusterIP   10.245.105.130  <none>         80/TCP
34m
```

```
doks-cluster-monitoring-kube-state-metrics
ClusterIP   10.245.140.151   <none>
8080/TCP            34m
doks-cluster-monitoring-pr-alertmanager
ClusterIP   10.245.197.254   <none>
9093/TCP            34m
doks-cluster-monitoring-pr-operator
ClusterIP   10.245.14.163    <none>
8080/TCP            34m
doks-cluster-monitoring-pr-prometheus
ClusterIP   10.245.201.173   <none>
9090/TCP            34m
doks-cluster-monitoring-prometheus-node-exporter
ClusterIP   10.245.72.218    <none>
30206/TCP           34m
prometheus-operated
ClusterIP   None             <none>
9090/TCP            34m
```

We are going to forward local port `8000` to port `80` of the `doks-cluster-monitoring-grafana` Service, which will in turn forward to port `3000` of a running Grafana Pod. These Service and Pod ports are configured in the `stable/grafana` Helm chart values file:

```
kubectl port-forward -n monitoring svc/doks-
cluster-monitoring-grafana 8000:80
```

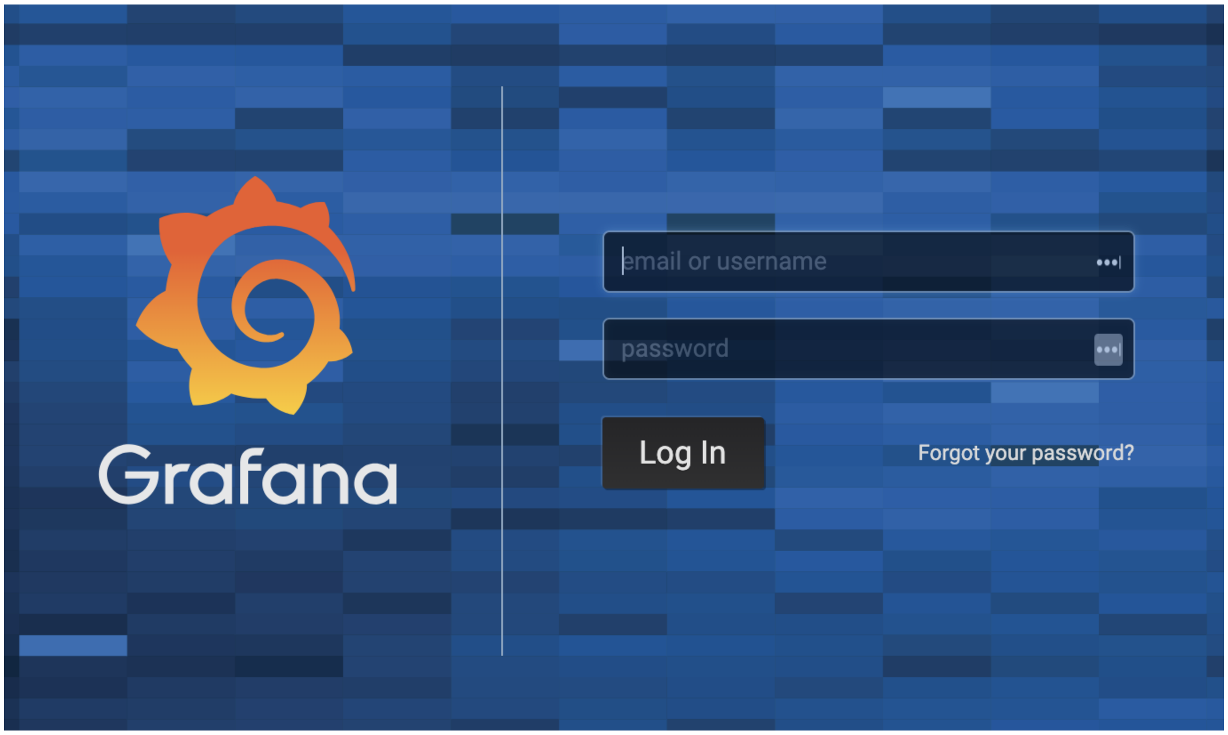
You should see the following output:

Output

```
Forwarding from 127.0.0.1:8000 -> 3000
Forwarding from [::1]:8000 -> 3000
```

This indicates that local port `8000` is being forwarded successfully to a Grafana Pod.

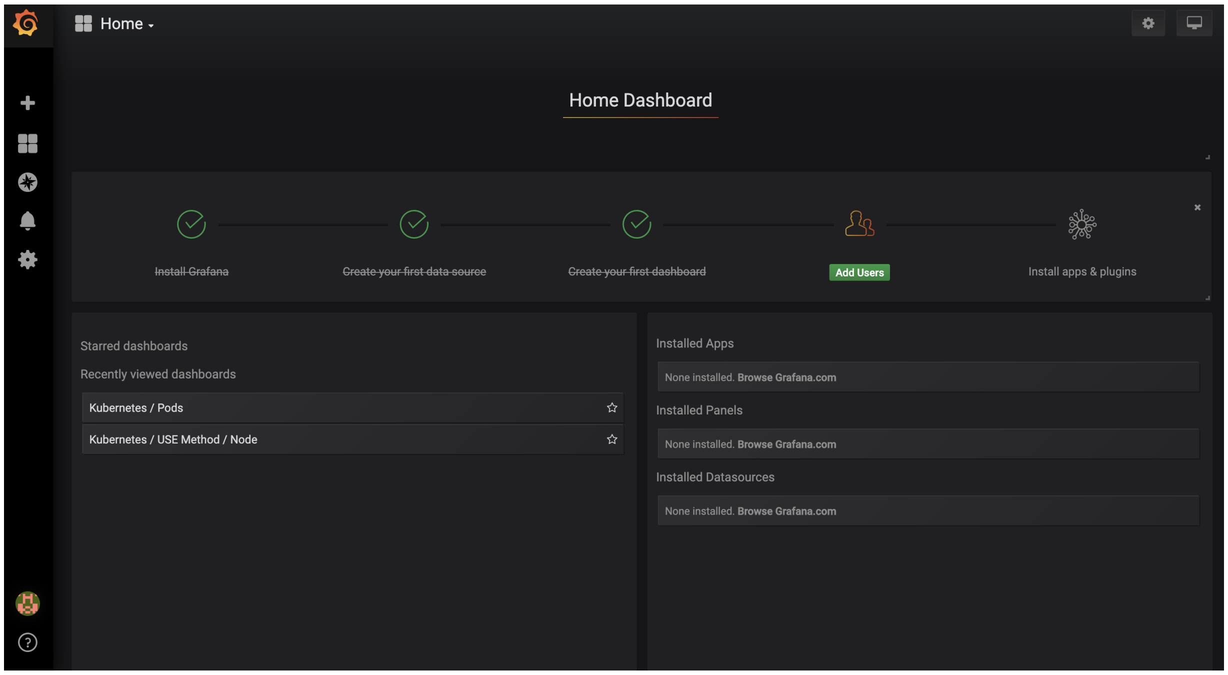
Visit `http://localhost:8000` in your web browser. You should see the following Grafana login page:



Grafana Login Page

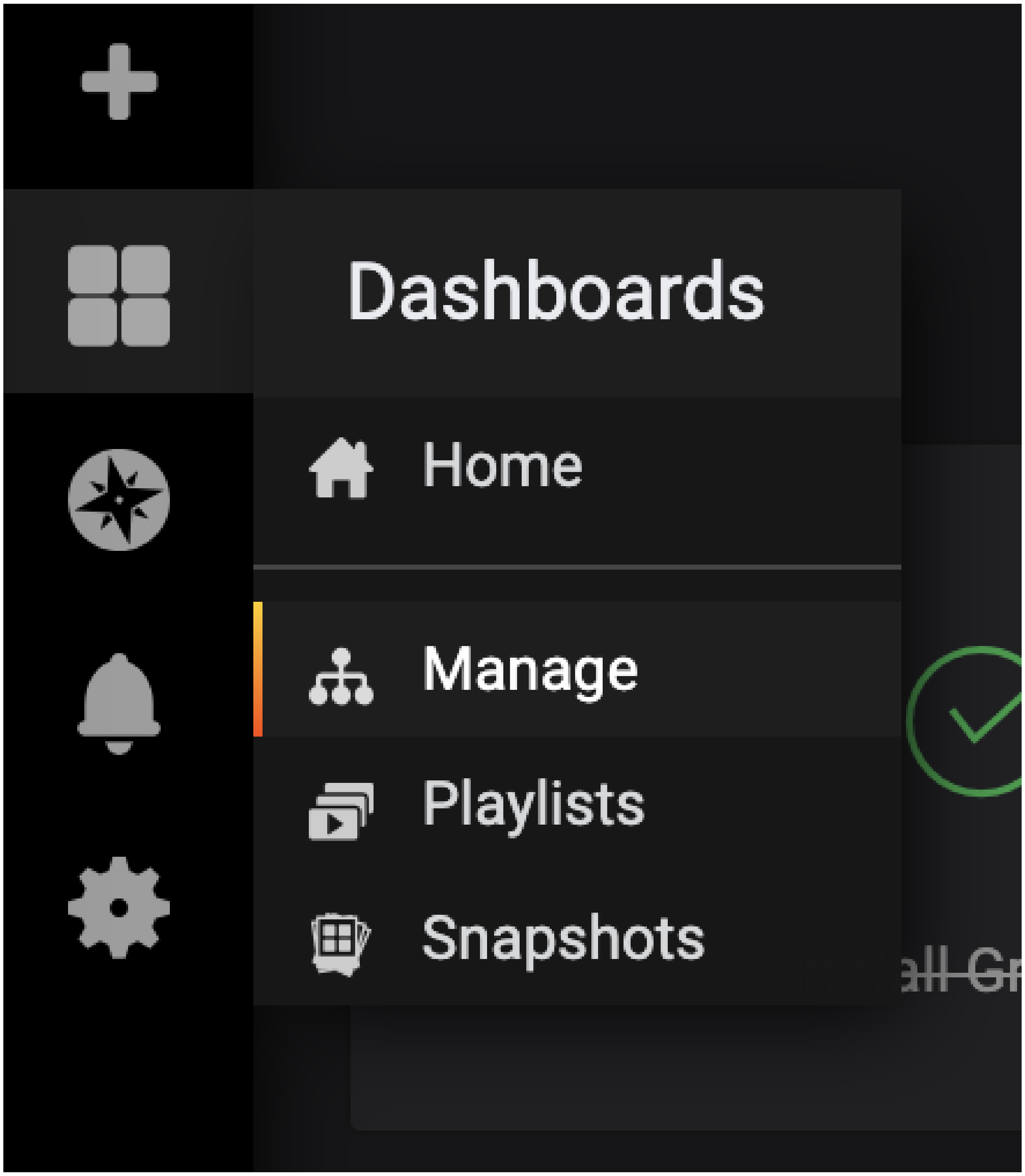
Enter admin as the username and the password you configured in `custom-values.yaml`. Then, hit Log In.

You’ll be brought to the following Home Dashboard:


Home
Home Dashboard
~~Install Grafana~~
~~Create your first data source~~
~~Create your first dashboard~~
Add Users
Install apps & plugins
Starred dashboards
Recently viewed dashboards
Kubernetes / Pods
Kubernetes / USE Method / Node
Installed Apps
None installed. Browse Grafana.com
Installed Panels
None installed. Browse Grafana.com
Installed Datasources
None installed. Browse Grafana.com

Grafana Home Page

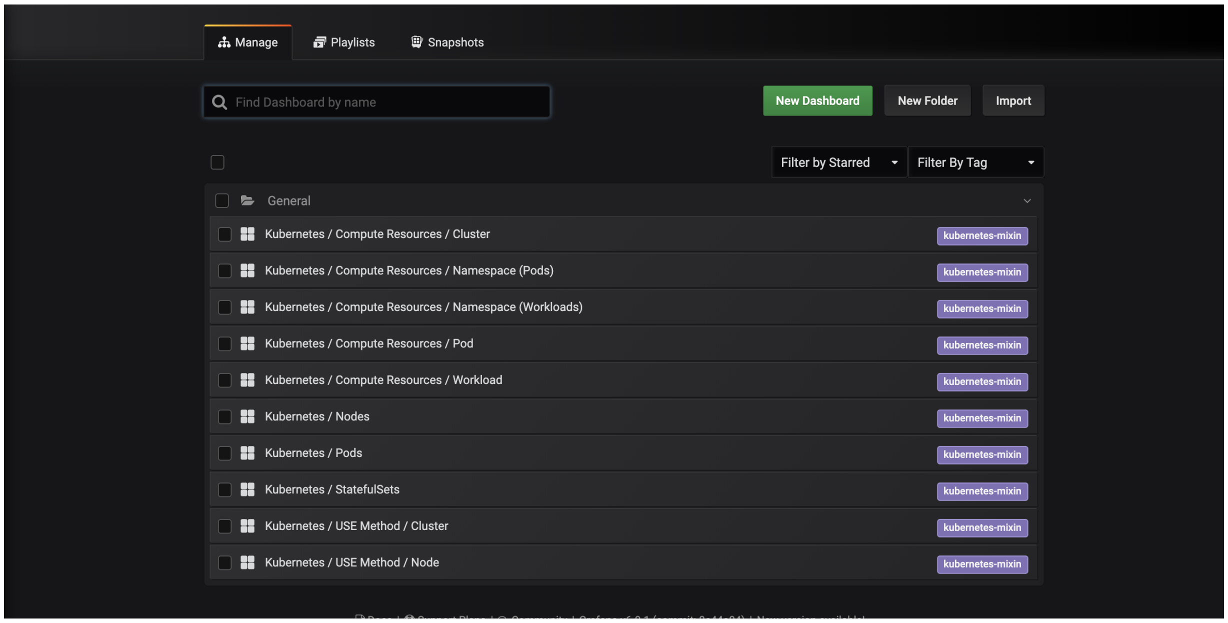
In the left-hand navigation bar, select the Dashboards button, then click on Manage:



Grafana Dashboard Tab

You'll be brought to the following dashboard management interface, which lists the dashboards installed by the `prometheus-operator`

Helm chart:

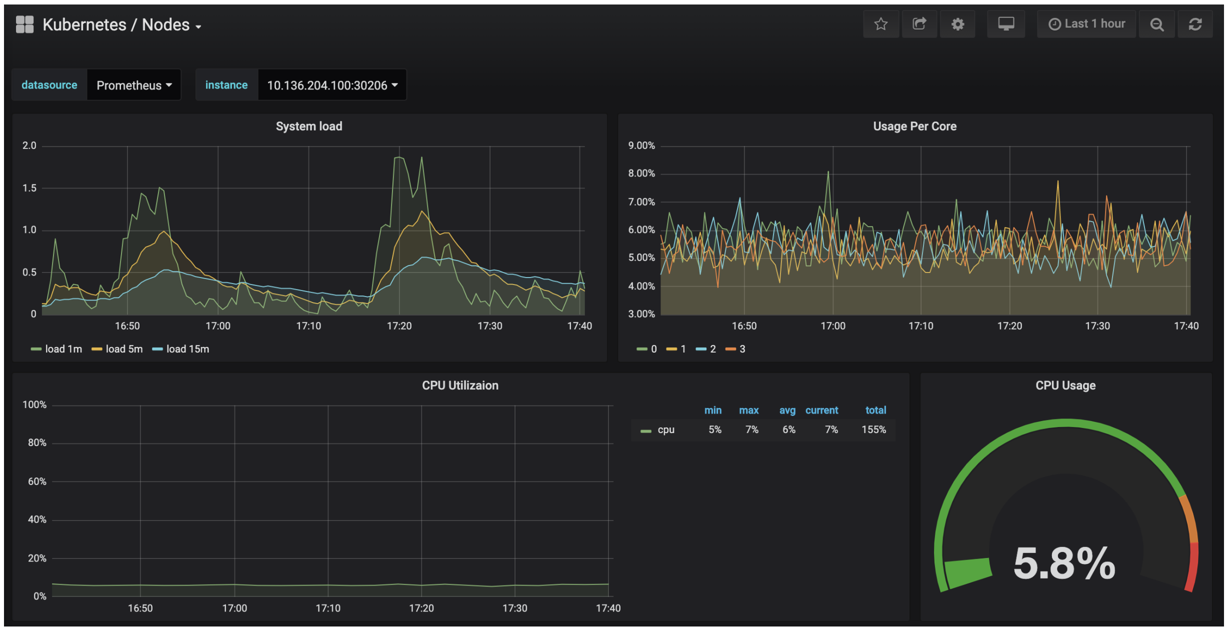


Kubernetes / Compute Resources / Cluster	kubernetes-mixin
Kubernetes / Compute Resources / Namespace (Pods)	kubernetes-mixin
Kubernetes / Compute Resources / Namespace (Workloads)	kubernetes-mixin
Kubernetes / Compute Resources / Pod	kubernetes-mixin
Kubernetes / Compute Resources / Workload	kubernetes-mixin
Kubernetes / Nodes	kubernetes-mixin
Kubernetes / Pods	kubernetes-mixin
Kubernetes / StatefulSets	kubernetes-mixin
Kubernetes / USE Method / Cluster	kubernetes-mixin
Kubernetes / USE Method / Node	kubernetes-mixin

Grafana Dashboard List

These dashboards are generated by `kubernetes-mixin`, an open-source project that allows you to create a standardized set of cluster monitoring Grafana dashboards and Prometheus alerts. To learn more, consult the Kubernetes Mixin GitHub repo.

Click in to the Kubernetes / Nodes dashboard, which visualizes CPU, memory, disk, and network usage for a given node:



	min	max	avg	current	total
cpu	5%	7%	6%	7%	155%

Grafana Nodes Dashboard

Describing each dashboard and how to use it to visualize your cluster's metrics data goes beyond the scope of this tutorial. To learn more about the USE method for analyzing a system's performance, you can consult Brendan Gregg's The Utilization Saturation and Errors (USE) Method page. Google's SRE Book is another helpful resource, in particular Chapter 6: Monitoring Distributed Systems. To learn how to build your own Grafana dashboards, check out Grafana's Getting Started page.

In the next step, we'll follow a similar process to connect to and explore the Prometheus monitoring system.

Step 4 — Accessing Prometheus and Alertmanager

To connect to the Prometheus Pods, we once again have to use `kubectl port-forward` to forward a local port. If you're done exploring

Grafana, you can close the port-forward tunnel by hitting `CTRL-C`. Alternatively you can open a new shell and port-forward connection.

Begin by listing running Services in the `monitoring` namespace:

```
kubectl get svc -n monitoring
```

You should see the following Services:

Output

```
NAME
TYPE         CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
alertmanager-operated
ClusterIP   None             <none>
9093/TCP,6783/TCP   34m
doks-cluster-monitoring-grafana
ClusterIP   10.245.105.130   <none>        80/TCP
34m
doks-cluster-monitoring-kube-state-metrics
ClusterIP   10.245.140.151   <none>
8080/TCP            34m
doks-cluster-monitoring-pr-alertmanager
ClusterIP   10.245.197.254   <none>
9093/TCP            34m
doks-cluster-monitoring-pr-operator
ClusterIP   10.245.14.163    <none>
8080/TCP            34m
doks-cluster-monitoring-pr-prometheus
ClusterIP   10.245.201.173   <none>
```

```
9090/TCP            34m
doks-cluster-monitoring-prometheus-node-exporter
ClusterIP   10.245.72.218    <none>
30206/TCP           34m
prometheus-operated
ClusterIP   None             <none>
9090/TCP            34m
```

We are going to forward local port `9090` to port `9090` of the `doks-cluster-monitoring-pr-prometheus` Service:

```
kubectl port-forward -n monitoring svc/doks-
cluster-monitoring-pr-prometheus 9090:9090
```

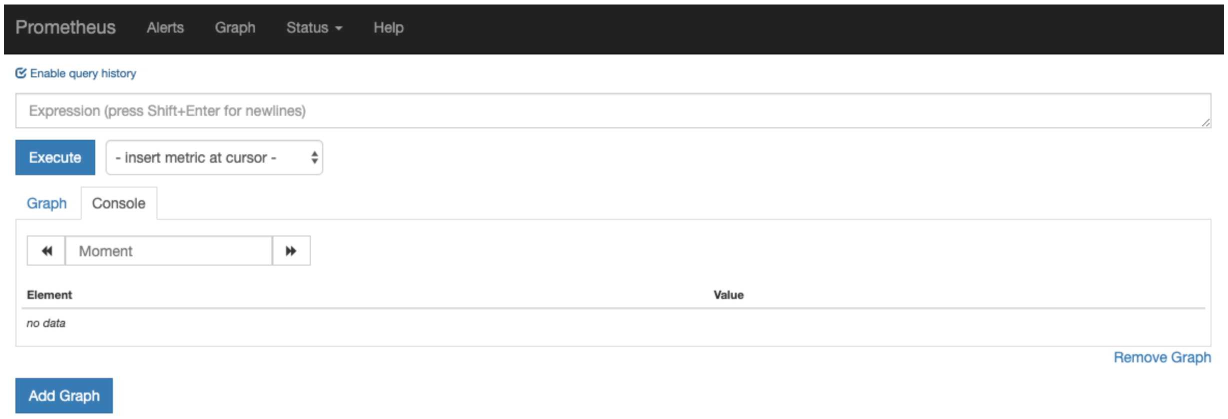
You should see the following output:

Output

```
Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
```

This indicates that local port `9090` is being forwarded successfully to a Prometheus Pod.

Visit `http://localhost:9090` in your web browser. You should see the following Prometheus Graph page:



Prometheus Graph Page

From here you can use PromQL, the Prometheus query language, to select and aggregate time series metrics stored in its database. To learn more about PromQL, consult Querying Prometheus from the official Prometheus docs.

In the Expression field, type `machine_cpu_cores` and hit Execute. You should see a list of time series with the metric `machine_cpu_cores` that reports the number of CPU cores on a given node. You can see which node generated the metric and which job scraped the metric in the metric labels.

Finally, in the top navigation bar, click on Status and then Targets to see the list of targets Prometheus has been configured to scrape. You should see a list of targets corresponding to the list of monitoring endpoints described at the beginning of Step 2.

To learn more about Promtheus and how to query your cluster metrics, consult the official Prometheus docs.

We'll follow a similar process to connect to AlertManager, which manages Alerts generated by Prometheus. You can explore these Alerts by clicking into Alerts in the Prometheus top navigation bar.

To connect to the Alertmanager Pods, we will once again use `kubectl port-forward` to forward a local port. If you're done exploring Prometheus, you can close the port-forward tunnel by hitting `CTRL-C`. Alternatively you can open a new shell and port-forward connection.

Begin by listing running Services in the `monitoring` namespace:

```
kubectl get svc -n monitoring
```

You should see the following Services:

Output

```
NAME
TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
alertmanager-operated
ClusterIP   None            <none>
9093/TCP,6783/TCP   34m
doks-cluster-monitoring-grafana
ClusterIP   10.245.105.130   <none>        80/TCP
34m
doks-cluster-monitoring-kube-state-metrics
ClusterIP   10.245.140.151   <none>
8080/TCP            34m
doks-cluster-monitoring-pr-alertmanager
ClusterIP   10.245.197.254   <none>
9093/TCP            34m
doks-cluster-monitoring-pr-operator
ClusterIP   10.245.14.163    <none>
```

```
8080/TCP            34m
doks-cluster-monitoring-pr-prometheus
ClusterIP   10.245.201.173   <none>
9090/TCP            34m
doks-cluster-monitoring-prometheus-node-exporter
ClusterIP   10.245.72.218    <none>
30206/TCP           34m
prometheus-operated
ClusterIP   None             <none>
9090/TCP            34m
```

We are going to forward local port `9093` to port `9093` of the `doks-cluster-monitoring-pr-alertmanager` Service.

```
kubectl port-forward -n monitoring svc/doks-
cluster-monitoring-pr-alertmanager 9093:9093
```

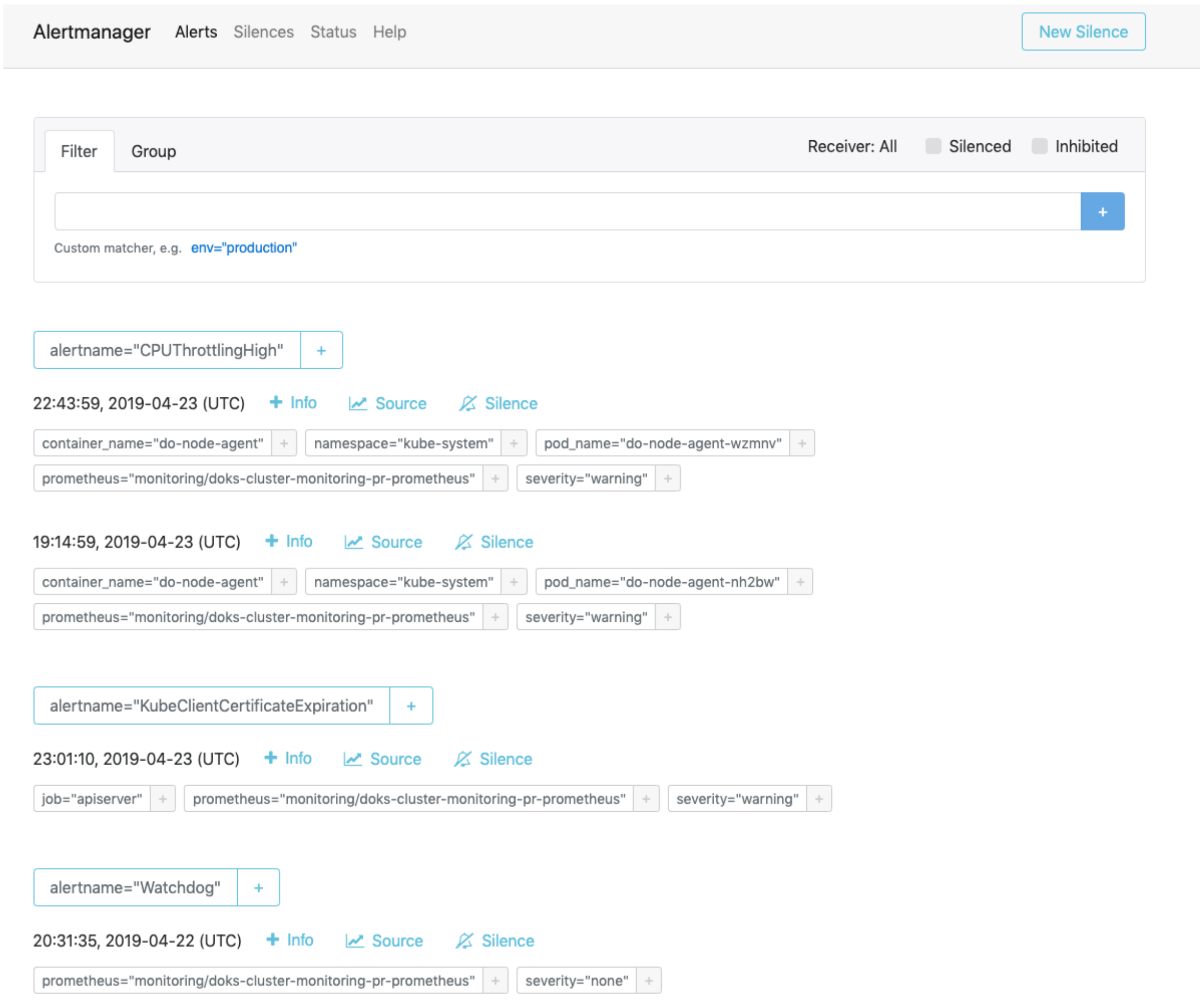
You should see the following output:

Output

```
Forwarding from 127.0.0.1:9093 -> 9093
Forwarding from [::1]:9093 -> 9093
```

This indicates that local port `9093` is being forwarded successfully to an Alertmanager Pod.

Visit `http://localhost:9093` in your web browser. You should see the following Alertmanager Alerts page:



alertname="CPUThrottlingHigh"

22:43:59, 2019-04-23 (UTC) Info Source Silence

container_name="do-node-agent" namespace="kube-system" pod_name="do-node-agent-wzmnv" prometheus="monitoring/doks-cluster-monitoring-pr-prometheus" severity="warning"

19:14:59, 2019-04-23 (UTC) Info Source Silence

container_name="do-node-agent" namespace="kube-system" pod_name="do-node-agent-nh2bw" prometheus="monitoring/doks-cluster-monitoring-pr-prometheus" severity="warning"

alertname="KubeClientCertificateExpiration"

23:01:10, 2019-04-23 (UTC) Info Source Silence

job="apiserver" prometheus="monitoring/doks-cluster-monitoring-pr-prometheus" severity="warning"

alertname="Watchdog"

20:31:35, 2019-04-22 (UTC) Info Source Silence

prometheus="monitoring/doks-cluster-monitoring-pr-prometheus" severity="none"

Alertmanager Alerts Page

From here, you can explore firing alerts and optionally silencing them. To learn more about Alertmanager, consult the official Alertmanager documentation.

Conclusion

In this tutorial, you installed a Prometheus, Grafana, and Alertmanager monitoring stack into your DigitalOcean Kubernetes cluster with a standard set of dashboards, Prometheus rules, and alerts. Since this was done using Helm, you can use `helm upgrade`, `helm rollback`, and `helm delete` to upgrade, roll back, or delete the monitoring stack. To learn more about these functions, consult How To Install Software on Kubernetes Clusters with the Helm Package Manager.

The `prometheus-operator` chart helps you get cluster monitoring up and running quickly using Helm. You may wish to build, deploy, and configure Prometheus Operator manually. To do so, consult the Prometheus Operator and kube-prometheus GitHub repos.

How To Set Up Laravel, Nginx, and MySQL with Docker Compose

Written by Faizan Bashir

In this tutorial, you will build a web application using the Laravel framework, with Nginx as the web server and MySQL as the database, all inside Docker containers. You will define the entire stack configuration in a docker-compose file, along with configuration files for PHP, MySQL, and Nginx.

The author selected The FreeBSD Foundation to receive a donation as part of the Write for DOnations program.

Over the past few years, Docker has become a frequently used solution for deploying applications thanks to how it simplifies running and deploying applications in ephemeral containers. When using a LEMP application stack, for example, with PHP, Nginx, MySQL and the Laravel framework, Docker can significantly streamline the setup process.

Docker Compose has further simplified the development process by allowing developers to define their infrastructure, including application services, networks, and volumes, in a single file. Docker Compose offers an efficient alternative to running multiple `docker container create` and `docker container run` commands.

In this tutorial, you will build a web application using the Laravel framework, with Nginx as the web server and MySQL as the database, all inside Docker containers. You will define the entire stack configuration in a `docker-compose` file, along with configuration files for PHP, MySQL, and Nginx.

Prerequisites

Before you start, you will need:

- One Ubuntu 18.04 server, and a non-root user with `sudo` privileges. Follow the [Initial Server Setup with Ubuntu 18.04] tutorial to set this up.
- Docker installed, following Steps 1 and 2 of [How To Install and Use Docker on Ubuntu 18.04].
- Docker Compose installed, following Step 1 of [How To Install Docker Compose on Ubuntu 18.04].

Step 1 — Downloading Laravel and Installing Dependencies

As a first step, we will get the latest version of Laravel and install the dependencies for the project, including [Composer], the application-level package manager for PHP. We will install these dependencies with Docker to avoid having to install Composer globally.

First, check that you are in your home directory and clone the latest Laravel release to a directory called **`laravel-app`**:

```
cd ~
git clone https://github.com/laravel/laravel.git
laravel-app
```

Move into the **`laravel-app`** directory:

```
cd ~/laravel-app
```

Next, use Docker's `composer` image to mount the directories that you will need for your Laravel project and avoid the overhead of installing Composer globally:

```
docker run --rm -v $(pwd):/app composer install
```

Using the `-v` and `--rm` flags with `docker run` creates an ephemeral container that will be bind-mounted to your current directory before being removed. This will copy the contents of your `~/`**`laravel-app`** directory to the container and also ensure that the `vendor` folder Composer creates inside the container is copied to your current directory.

As a final step, set permissions on the project directory so that it is owned by your non-root user:

```
sudo chown -R $USER:$USER ~/laravel-app
```

This will be important when you write the Dockerfile for your application image in Step 4, as it will allow you to work with your application code and run processes in your container as a non-root user.

With your application code in place, you can move on to defining your services with Docker Compose.

Step 2 — Creating the Docker Compose File

Building your applications with Docker Compose simplifies the process of setting up and versioning your infrastructure. To set up our Laravel application, we will write a `docker-compose` file that defines our web server, database, and application services.

Open the file:

```
nano ~/laravel-app/docker-compose.yml
```

In the docker-compose file, you will define three services: app, webserver, and db. Add the following code to the file, being sure to replace the root password for MYSQL_ROOT_PASSWORD, defined as an environment variable under the db service, with a strong password of your choice:

~/laravel-app/docker-compose.yml

```
version: '3'
services:

  #PHP Service
  app:
    build:
      context: .
      dockerfile: Dockerfile
    image: digitalocean.com/php
    container_name: app
    restart: unless-stopped
    tty: true
    environment:
      SERVICE_NAME: app
      SERVICE_TAGS: dev
    working_dir: /var/www
    networks:
      - app-network
```

```
#Nginx Service
webserver:
  image: nginx:alpine
  container_name: webserver
  restart: unless-stopped
  tty: true
  ports:
    - "80:80"
    - "443:443"
  networks:
    - app-network

#MySQL Service
db:
  image: mysql:5.7.22
  container_name: db
  restart: unless-stopped
  tty: true
  ports:
    - "3306:3306"
  environment:
    MYSQL_DATABASE: laravel<^>
    MYSQL_ROOT_PASSWORD: your_mysql_root_password
    SERVICE_TAGS: dev
    SERVICE_NAME: mysql
  networks:
    - app-network
```

```
        app-network

#Docker Networks
networks:
  app-network:
    driver: bridge
```

The services defined here include:

- `app`: This service definition contains the Laravel application and runs a custom Docker image, `digitalocean.com/php`, that you will define in Step 4. It also sets the `working_dir` in the container to `/var/www`.
- `webserver`: This service definition pulls the [`nginx:alpine` image](#) from Docker and exposes ports `80` and `443`.
- `db`: This service definition pulls the [`mysql:5.7.22` image](#) from Docker and defines a few environmental variables, including a database called **`laravel`** for your application and the root password for the database. You are free to name the database whatever you would like, and you should replace **`your_mysql_root_password`** with your own strong password. This service definition also maps port `3306` on the host to port `3306` on the container.

Each `container_name` property defines a name for the container, which corresponds to the name of the service. If you don't define this property, Docker will assign a name to each container by combining a historically famous person’s name and a random word separated by an underscore.

To facilitate communication between containers, the services are connected to a bridge network called `app-network`. A bridge network uses a software bridge that allows containers connected to the same bridge network to communicate with each other. The bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other. This creates a greater level of security for applications, ensuring that only related services can communicate with one another. It also means that you can define multiple networks and services connecting to related functions: front-end application services can use a `frontend` network, for example, and back-end services can use a `backend` network.

Let’s look at how to add volumes and bind mounts to your service definitions to persist your application data.

Step 3 — Persisting Data

Docker has powerful and convenient features for persisting data. In our application, we will make use of volumes and bind mounts for persisting the database, and application and configuration files. Volumes offer flexibility for backups and persistence beyond a container’s lifecycle, while bind mounts facilitate code changes during development, making changes to your host files or directories immediately available in your containers. Our setup will make use of both.

Warning: By using bind mounts, you make it possible to change the host filesystem through processes running in a container, including creating, modifying, or deleting important system files or directories. This is a powerful ability with security implications, and could impact non-Docker processes on the host system. Use bind mounts with care.

In the `docker-compose` file, define a volume called `dbdata` under the `db` service definition to persist the MySQL database:

~/laravel-app/docker-compose.yml

```
...
#MySQL Service
db:
  ...
    volumes:
      - dbdata:/var/lib/mysql
    networks:
      - app-network
  ...
```

The named volume `dbdata` persists the contents of the `/var/lib/mysql` folder present inside the container. This allows you to stop and restart the `db` service without losing data.

At the bottom of the file, add the definition for the `dbdata` volume:

~/laravel-app/docker-compose.yml

```
...
#Volumes
volumes:
  dbdata:
    driver: local
```

With this definition in place, you will be able to use this volume across services.

Next, add a bind mount to the `db` service for the MySQL configuration files you will create in Step 7:

~/laravel-app/docker-compose.yml

```
...
#MySQL Service
db:
  ...
    volumes:
      - dbdata:/var/lib/mysql
      - ./mysql/my.cnf:/etc/mysql/my.cnf
  ...
```

This bind mount binds `~/laravel-app/mysql/my.cnf` to `/etc/mysql/my.cnf` in the container.

Next, add bind mounts to the `webserver` service. There will be two: one for your application code and another for the Nginx configuration definition that you will create in Step 6:

~/laravel-app/docker-compose.yml

```
#Nginx Service
webserver:
  ...
  volumes:
      - ./:/var/www
      - ./nginx/conf.d/:/etc/nginx/conf.d/
  networks:
      - app-network
```

The first bind mount binds the application code in the `~/laravel-app` directory to the `/var/www` directory inside the container. The configuration file that you will add to `~/laravel-app/nginx/conf.d/` will also be mounted to `/etc/nginx/conf.d/` in the container, allowing you to add or modify the configuration directory's contents as needed.

Finally, add the following bind mounts to the `app` service for the application code and configuration files:

~/laravel-app/docker-compose.yml

```
#PHP Service
app:
  ...
  volumes:
      - ./:/var/www
      - ./php/local.ini:/usr/local/etc/php/conf.d/local.ini
  networks:
      - app-network
```

The `app` service is bind-mounting the `~/laravel-app` folder, which contains the application code, to the `/var/www` folder in the container. This will speed up the development process, since any changes made to your local application directory will be instantly reflected inside the container. You are also binding your PHP configuration file,

`~/laravel-app/php/local.ini`, to `/usr/local/etc/php/conf.d/local.ini` inside the container. You will create the local PHP configuration file in Step 5.

Your `docker-compose` file will now look like this:

~/laravel-app/docker-compose.yml

```
version: '3'
services:

  #PHP Service
  app:
    build:
      context: .
      dockerfile: Dockerfile
    image: digitalocean.com/php
    container_name: app
    restart: unless-stopped
    tty: true
    environment:
      SERVICE_NAME: app
      SERVICE_TAGS: dev
    working_dir: /var/www
    volumes:
      - ./:/var/www
      - ./php/local.ini:/usr/local/etc/php/conf.d/l

    networks:
```

```yaml
  networks:
    - app-network

#Nginx Service
webserver:
  image: nginx:alpine
  container_name: webserver
  restart: unless-stopped
  tty: true
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - ./:/var/www
    - ./nginx/conf.d/:/etc/nginx/conf.d/
  networks:
    - app-network

#MySQL Service
db:
  image: mysql:5.7.22
  container_name: db
  restart: unless-stopped
  tty: true
  ports:
    - "3306:3306"

  environment:
      MYSQL_DATABASE: laravel<^>
      MYSQL_ROOT_PASSWORD: your_mysql_root_password
      SERVICE_TAGS: dev
      SERVICE_NAME: mysql
    volumes:
      - dbdata:/var/lib/mysql/
      - ./mysql/my.cnf:/etc/mysql/my.cnf
    networks:
      - app-network

#Docker Networks
networks:
  app-network:
    driver: bridge
#Volumes
volumes:
  dbdata:
    driver: local
```

Save the file and exit your editor when you are finished making changes.

With your `docker-compose` file written, you can now build the custom image for your application.

Step 4 — Creating the Dockerfile

Docker allows you to specify the environment inside of individual containers with a Dockerfile. A Dockerfile enables you to create custom images that you can use to install the software required by your application and configure settings based on your requirements. You can push the custom images you create to [Docker Hub]() or any private registry.

Our `Dockerfile` will be located in our `~/laravel-app` directory. Create the file:

```
nano ~/laravel-app/Dockerfile
```

This `Dockerfile` will set the base image and specify the necessary commands and instructions to build the Laravel application image. Add the following code to the file:

~/laravel-app/php/Dockerfile

```
FROM php:7.2-fpm

# Copy composer.lock and composer.json
COPY composer.lock composer.json /var/www/

# Set working directory
WORKDIR /var/www

# Install dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    mysql-client \
    libpng-dev \
    libjpeg62-turbo-dev \
    libfreetype6-dev \
    locales \
    zip \
    jpegoptim optipng pngquant gifsicle \
    vim \
    unzip \
    git \
    curl

# Clear cache
RUN apt-get clean && rm -rf /var/lib/apt/lists/*

# Install extensions
RUN docker-php-ext-install pdo_mysql mbstring zip
exif pcntl
RUN docker-php-ext-configure gd --with-gd --with-
freetype-dir=/usr/include/ --with-jpeg-
dir=/usr/include/ --with-png-dir=/usr/include/
RUN docker-php-ext-install gd

# Install composer
RUN curl -sS https://getcomposer.org/installer |
php -- --install-dir=/usr/local/bin --
filename=composer

# Add user for laravel application
RUN groupadd -g 1000 www
RUN useradd -u 1000 -ms /bin/bash -g www www

# Copy existing application directory contents
COPY . /var/www

# Copy existing application directory permissions
COPY --chown=www:www . /var/www

# Change current user to www
USER www

# Expose port 9000 and start php-fpm server
EXPOSE 9000
CMD ["php-fpm"]
```

First, the Dockerfile creates an image on top of the `php:7.2-fpm` Docker image. This is a Debian-based image that has the PHP FastCGI implementation PHP-FPM installed. The file also installs the prerequisite packages for Laravel: `mcrypt`, `pdo_mysql`, `mbstring`, and `imagick` with `composer`.

The `RUN` directive specifies the commands to update, install, and configure settings inside the container, including creating a dedicated user and group called www. The `WORKDIR` instruction specifies the `/var/www` directory as the working directory for the application.

Creating a dedicated user and group with restricted permissions mitigates the inherent vulnerability when running Docker containers, which run by default as root. Instead of running this container as root, we've created the www user, who has read/write access to the `/var/www` folder thanks to the `COPY` instruction that we are using with the `--chown` flag to copy the application folder's permissions.

Finally, the `EXPOSE` command exposes a port in the container, `9000`, for the `php-fpm` server. `CMD` specifies the command that should run once the container is created. Here, `CMD` specifies `"php-fpm"`, which will start the server.

Save the file and exit your editor when you are finished making changes.

You can now move on to defining your PHP configuration.

Step 5 — Configuring PHP

Now that you have defined your infrastructure in the `docker-compose` file, you can configure the PHP service to act as a PHP processor for incoming requests from Nginx.

To configure PHP, you will create the `local.ini` file inside the `php` folder. This is the file that you bind-mounted to `/usr/local/etc/php/conf.d/local.ini` inside the container in Step 2. Creating this file will allow you to override the default `php.ini` file that PHP reads when it starts.

Create the `php` directory:

```
mkdir ~/laravel-app/php
```

Next, open the `local.ini` file:

```
nano ~/laravel-app/php/local.ini
```

To demonstrate how to configure PHP, we'll add the following code to set size limitations for uploaded files:

~/laravel-app/php/local.ini

```
upload_max_filesize=40M
post_max_size=40M
```

The `upload_max_filesize` and `post_max_size` directives set the maximum allowed size for uploaded files, and demonstrate how you can set `php.ini` configurations from your `local.ini` file. You can put any PHP-specific configuration that you want to override in the `local.ini` file.

Save the file and exit your editor.

With your PHP `local.ini` file in place, you can move on to configuring Nginx.

Step 6 — Configuring Nginx

With the PHP service configured, you can modify the Nginx service to use PHP-FPM as the FastCGI server to serve dynamic content. The FastCGI server is based on a binary protocol for interfacing interactive programs with a web server. For more information, please refer to this article on Understanding and Implementing FastCGI Proxying in Nginx.

To configure Nginx, you will create an `app.conf` file with the service configuration in the `~/laravel-app/nginx/conf.d/` folder.

First, create the `nginx/conf.d/` directory:

```
mkdir -p ~/laravel-app/nginx/conf.d
```

Next, create the `app.conf` configuration file:

```
nano ~/laravel-app/nginx/conf.d/app.conf
```

Add the following code to the file to specify your Nginx configuration:

~/laravel-app/nginx/conf.d/app.conf

```
server {
    listen 80;
    index index.php index.html;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/public;
    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass app:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO
$fastcgi_path_info;
    }
    location / {
        try_files $uri $uri/ /index.php?
$query_string;
        gzip_static on;
    }
}
```

The server block defines the configuration for the Nginx web server with the following directives: - `listen`: This directive defines the port on which the server will listen to incoming requests. - `error_log` and `access_log`: These directives define the files for writing logs. - `root`:

This directive sets the root folder path, forming the complete path to any requested file on the local file system.

In the `php` location block, the `fastcgi_pass` directive specifies that the `app` service is listening on a TCP socket on port `9000`. This makes the PHP-FPM server listen over the network rather than on a Unix socket. Though a Unix socket has a slight advantage in speed over a TCP socket, it does not have a network protocol and thus skips the network stack. For cases where hosts are located on one machine, a Unix socket may make sense, but in cases where you have services running on different hosts, a TCP socket offers the advantage of allowing you to connect to distributed services. Because our `app` container is running on a different host from our `webserver` container, a TCP socket makes the most sense for our configuration.

Save the file and exit your editor when you are finished making changes.

Thanks to the bind mount you created in Step 2, any changes you make inside the `nginx/conf.d/` folder will be directly reflected inside the `webserver` container.

Next, let's look at our MySQL settings.

Step 7 — Configuring MySQL

With PHP and Nginx configured, you can enable MySQL to act as the database for your application.

To configure MySQL, you will create the `my.cnf` file in the `mysql` folder. This is the file that you bind-mounted to `/etc/mysql/my.cnf` inside the container in Step 2. This bind mount allows you to override the `my.cnf` settings as and when required.

To demonstrate how this works, we'll add settings to the `my.cnf` file that enable the general query log and specify the log file.

First, create the `mysql` directory:

```
mkdir ~/laravel-app/mysql
```

Next, make the `my.cnf` file:

```
nano ~/laravel-app/mysql/my.cnf
```

In the file, add the following code to enable the query log and set the log file location:

~/laravel-app/mysql/my.cnf

```
[mysqld]
general_log = 1
general_log_file = /var/lib/mysql/general.log
```

This `my.cnf` file enables logs, defining the `general_log` setting as `1` to allow general logs. The `general_log_file` setting specifies where the logs will be stored.

Save the file and exit your editor.

Our next step will be to start the containers.

Step 8 — Running the Containers and Modifying Environment Settings

Now that you have defined all of your services in your `docker-compose` file and created the configuration files for these services, you can start the containers. As a final step, though, we will make a copy of the `.env.example` file that Laravel includes by default and name the copy `.env`, which is the file Laravel expects to define its environment:

```
cp .env.example .env
```

We will configure the specific details of our setup in this file once we have started the containers.

With all of your services defined in your `docker-compose` file, you just need to issue a single command to start all of the containers, create the volumes, and set up and connect the networks:

```
docker-compose up -d
```

When you run `docker-compose up` for the first time, it will download all of the necessary Docker images, which might take a while. Once the images are downloaded and stored in your local machine, Compose will create your containers. The `-d` flag daemonizes the process, running your containers in the background.

Once the process is complete, use the following command to list all of the running containers:

```
docker ps
```

You will see the following output with details about your `app`, `webserver`, and `db` containers:

Output

```
CONTAINER ID        NAMES                   IMAGE
STATUS              PORTS
c31b7b3251e0        db
```

```
mysql:5.7.22                          Up 2 seconds
0.0.0.0:3306->3306/tcp
ed5a69704580       app
digitalocean.com/php                  Up 2 seconds
9000/tcp
5ce4ee31d7c0       webserver
nginx:alpine                          Up 2 seconds
0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
```

The `CONTAINER ID` in this output is a unique identifier for each container, while `NAMES` lists the service name associated with each. You can use both of these identifiers to access the containers. `IMAGE` defines the image name for each container, while `STATUS` provides information about the container's state: whether it's running, restarting, or stopped.

You can now modify the `.env` file on the `app` container to include specific details about your setup.

Open the file using `docker-compose exec`, which allows you to run specific commands in containers. In this case, you are opening the file for editing:

```
docker-compose exec app nano .env
```

Find the block that specifies `DB_CONNECTION` and update it to reflect the specifics of your setup. You will modify the following fields: - `DB_HOST` will be your `db` database container. - `DB_DATABASE` will be the **`laravel`** database. - `DB_USERNAME` will be the username you will use for your database. In this case, we will use **`laraveluser`**. - `DB_PASSWORD` will be the secure password you would like to use for this user account.

/var/www/.env

```
DB_CONNECTION=mysql
DB_HOST=db
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=laraveluser
DB_PASSWORD=your_laravel_db_password
```

Save your changes and exit your editor.

Next, set the application key for the Laravel application with the `php artisan key:generate` command. This command will generate a key and copy it to your `.env` file, ensuring that your user sessions and encrypted data remain secure:

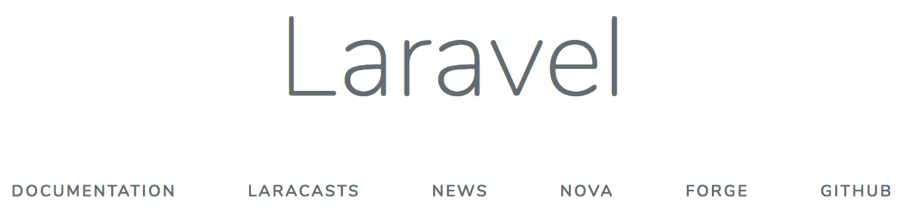
```
docker-compose exec app php artisan key:generate
```

You now have the environment settings required to run your application. To cache these settings into a file, which will boost your application's load speed, run:

```
docker-compose exec app php artisan config:cache
```

Your configuration settings will be loaded into `/var/www/bootstrap/cache/config.php` on the container.

As a final step, visit `http://your_server_ip` in the browser. You will see the following home page for your Laravel application:



Laravel Home Page

With your containers running and your configuration information in place, you can move on to configuring your user information for the `laravel` database on the `db` container.

Step 9 — Creating a User for MySQL

The default MySQL installation only creates the root administrative account, which has unlimited privileges on the database server. In general, it's better to avoid using the root administrative account when interacting with the database. Instead, let's create a dedicated database user for our application's Laravel database.

To create a new user, execute an interactive bash shell on the `db` container with `docker-compose exec`:

```
docker-compose exec db bash
```

Inside the container, log into the MySQL root administrative account:

```
[environment second]
mysql -u root -p
```

You will be prompted for the password you set for the MySQL root account during installation in your `docker-compose` file.

Start by checking for the database called **`laravel`**, which you defined in your `docker-compose` file. Run the `show databases` command to check for existing databases:

```
[environment second]
show databases;
```

You will see the **`laravel`** database listed in the output:

Output

```
[environment second]
+--------------------+
| Database           |
+--------------------+
| information_schema |
| laravel            |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

Next, create the user account that will be allowed to access this database. Our username will be **laraveluser**, though you can replace this with another name if you'd prefer. Just be sure that your username and password here match the details you set in your `.env` file in the previous step:

```
[environment second]
GRANT ALL ON laravel.* TO 'laraveluser'@'%'
IDENTIFIED BY 'your_laravel_db_password';
```

Flush the privileges to notify the MySQL server of the changes:

```
[environment second]
FLUSH PRIVILEGES;
```

Exit MySQL:

```
[environment second]
EXIT;
```

Finally, exit the container:

```
[environment second]
exit
```

You have configured the user account for your Laravel application database and are ready to migrate your data and work with the Tinker console.

Step 10 — Migrating Data and Working with the Tinker Console

With your application running, you can migrate your data and experiment with the `tinker` command, which will initiate a PsySH console with Laravel preloaded. PsySH is a runtime developer console and interactive debugger for PHP, and Tinker is a REPL specifically for Laravel. Using the `tinker` command will allow you to interact with your Laravel application from the command line in an interactive shell.

First, test the connection to MySQL by running the Laravel `artisan migrate` command, which creates a `migrations` table in the database from inside the container:

```
docker-compose exec app php artisan migrate
```

This command will migrate the default Laravel tables. The output confirming the migration will look like this:

Output

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating:
2014_10_12_100000_create_password_resets_table
Migrated:
2014_10_12_100000_create_password_resets_table
```

Once the migration is complete, you can run a query to check if you are properly connected to the database using the `tinker` command:

```
docker-compose exec app php artisan tinker
```

Test the MySQL connection by getting the data you just migrated:

```
\DB::table('migrations')->get();
```

You will see output that looks like this:

Output

```
=> Illuminate\Support\Collection {#2856
     all: [
       {#2862
         +"id": 1,
         +"migration":
"2014_10_12_000000_create_users_table",
         +"batch": 1,
       },
       {#2865
         +"id": 2,
         +"migration":
"2014_10_12_100000_create_password_resets_table",
         +"batch": 1,
       },
     ],
   }
```

You can use `tinker` to interact with your databases and to experiment with services and models.

With your Laravel application in place, you are ready for further development and experimentation. ## Conclusion

You now have a LEMP stack application running on your server, which you've tested by accessing the Laravel welcome page and creating MySQL database migrations.

Key to the simplicity of this installation is Docker Compose, which allows you to create a group of Docker containers, defined in a single file, with a single command. If you would like to learn more about how to do CI with Docker Compose, take a look at How To Configure a Continuous

Integration Testing Environment with Docker and Docker Compose on Ubuntu 16.04. If you want to streamline your Laravel application deployment process then How to Automatically Deploy Laravel Applications with Deployer on Ubuntu 16.04 will be a relevant resource.

How To Migrate a Docker Compose Workflow to Kubernetes

Written by Kathleen Juell

To run your services on a distributed platform like Kubernetes, you will need to translate your Docker Compose service definitions to Kubernetes objects. Kompose is a conversion tool that helps developers move their Docker Compose workflows to container clusters like Kubernetes.

In this tutorial, you will translate your Node.js application's Docker Compose services into Kubernetes objects using kompose. You will use the object definitions that kompose provides as a starting point and make adjustments to ensure that your setup will use Secrets, Services, and PersistentVolumeClaims in the way that Kubernetes expects. By the end of the tutorial, you will have a single-instance Node.js application with a MongoDB database running on a Kubernetes cluster.

When building modern, stateless applications, containerizing your application's components is the first step in deploying and scaling on distributed platforms. If you have used Docker Compose in development, you will have modernized and containerized your application by: - Extracting necessary configuration information from your code. - Offloading your application's state. - Packaging your application for repeated use.

You will also have written service definitions that specify how your container images should run.

To run your services on a distributed platform like Kubernetes, you will need to translate your Compose service definitions to Kubernetes objects. This will allow you to scale your application with resiliency. One tool that can speed up the translation process to Kubernetes is kompose, a conversion tool that helps developers move Compose workflows to container orchestrators like Kubernetes or OpenShift.

In this tutorial, you will translate Compose services to Kubernetes objects using kompose. You will use the object definitions that kompose provides as a starting point and make adjustments to ensure that your setup will use Secrets, Services, and PersistentVolumeClaims in the way that Kubernetes expects. By the end of the tutorial, you will have a single-instance Node.js application with a MongoDB database running on a Kubernetes cluster. This setup will mirror the functionality of the code described in Containerizing a Node.js Application with Docker Compose and will be a good starting point to build out a production-ready solution that will scale with your needs.

Prerequisites

- A Kubernetes 1.10+ cluster with role-based access control (RBAC) enabled. This setup will use a DigitalOcean Kubernetes cluster, but you are free to create a cluster using another method.
- The `kubectl` command-line tool installed on your local machine or development server and configured to connect to your cluster. You can read more about installing `kubectl` in the official documentation.
- Docker installed on your local machine or development server. If you are working with Ubuntu 18.04, follow Steps 1 and 2 of How To Install

and Use Docker on Ubuntu 18.04; otherwise, follow the official documentation for information about installing on other operating systems. Be sure to add your non-root user to the `docker` group, as described in Step 2 of the linked tutorial.
- A Docker Hub account. For an overview of how to set this up, refer to this introduction to Docker Hub.

Step 1 — Installing kompose

To begin using kompose, navigate to the project's GitHub Releases page, and copy the link to the current release (version **1.18.0** as of this writing). Paste this link into the following `curl` command to download the latest version of kompose:

```
curl -L
https://github.com/kubernetes/kompose/releases/down
load/v1.18.0/kompose-linux-amd64 -o kompose
```

For details about installing on non-Linux systems, please refer to the installation instructions.

Make the binary executable:

```
chmod +x kompose
```

Move it to your `PATH`:

```
sudo mv ./kompose /usr/local/bin/kompose
```

To verify that it has been installed properly, you can do a version check:

```
kompose version
```

If the installation was successful, you will see output like the following:

Output

```
1.18.0 (06a2e56)
```

With `kompose` installed and ready to use, you can now clone the Node.js project code that you will be translating to Kubernetes.

Step 2 — Cloning and Packaging the Application

To use our application with Kubernetes, we will need to clone the project code and package the application so that the `kubelet` service can pull the image.

Our first step will be to clone the node-mongo-docker-dev repository from the DigitalOcean Community GitHub account. This repository includes the code from the setup described in Containerizing a Node.js Application for Development With Docker Compose, which uses a demo Node.js application to demonstrate how to set up a development environment using Docker Compose. You can find more information about the application itself in the series From Containers to Kubernetes with Node.js.

Clone the repository into a directory called **`node_project`**:

```
git clone https://github.com/do-community/node-
mongo-docker-dev.git node_project
```

Navigate to the **`node_project`** directory:

```
cd node_project
```

The **`node_project`** directory contains files and directories for a shark information application that works with user input. It has been modernized to work with containers: sensitive and specific configuration information has been removed from the application code and refactored to be injected at runtime, and the application's state has been offloaded to a MongoDB database.

For more information about designing modern, stateless applications, please see Architecting Applications for Kubernetes and Modernizing Applications for Kubernetes.

The project directory includes a `Dockerfile` with instructions for building the application image. Let's build the image now so that you can push it to your Docker Hub account and use it in your Kubernetes setup.

Using the `docker build` command, build the image with the `-t` flag, which allows you to tag it with a memorable name. In this case, tag the image with your Docker Hub username and name it **`node-kubernetes`** or a name of your own choosing:

```
docker build -t your_dockerhub_username/node-kubernetes .
```

The `.` in the command specifies that the build context is the current directory.

It will take a minute or two to build the image. Once it is complete, check your images:

```
docker images
```

You will see the following output:

Output
```
REPOSITORY                                  TAG
IMAGE ID            CREATED             SIZE
your_dockerhub_username/node-kubernetes     latest
9c6f897e1fbc        3 seconds ago       90MB
node                                        10-alpine
94f3c8956482        12 days ago         71MB
```

Next, log in to the Docker Hub account you created in the prerequisites:

```
docker login -u your_dockerhub_username
```

When prompted, enter your Docker Hub account password. Logging in this way will create a `~/.docker/config.json` file in your user's home directory with your Docker Hub credentials.

Push the application image to Docker Hub with the `docker push` command. Remember to replace **`your_dockerhub_username`** with your own Docker Hub username:

```
docker push your_dockerhub_username/node-kubernetes
```

You now have an application image that you can pull to run your application with Kubernetes. The next step will be to translate your application service definitions to Kubernetes objects.

Step 3 — Translating Compose Services to Kubernetes Objects with kompose

Our Docker Compose file, here called `docker-compose.yaml`, lays out the definitions that will run our services with Compose. A service in Compose is a running container, and service definitions contain information about how each container image will run. In this step, we will translate these definitions to Kubernetes objects by using kompose to create `yaml` files. These files will contain specs for the Kubernetes objects that describe their desired state.

We will use these files to create different types of objects: Services, which will ensure that the Pods running our containers remain accessible; Deployments, which will contain information about the desired state of our Pods; a PersistentVolumeClaim to provision storage for our database data; a ConfigMap for environment variables injected at runtime; and a Secret for our application's database user and password. Some of these definitions will be in the files kompose will create for us, and others we will need to create ourselves.

First, we will need to modify some of the definitions in our `docker-compose.yaml` file to work with Kubernetes. We will include a reference to our newly-built application image in our `nodejs` service definition and remove the bind mounts, volumes, and additional commands that we used to run the application container in development with Compose. Additionally, we'll redefine both containers' restart policies to be in line with the behavior Kubernetes expects.

Open the file with `nano` or your favorite editor:

```
nano docker-compose.yaml
```

The current definition for the `nodejs` application service looks like this:

~/node_project/docker-compose.yaml

```
...
services:
  nodejs:
    build:
      context: .
      dockerfile: Dockerfile
    image: nodejs
    container_name: nodejs
    restart: unless-stopped
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
```

```
        - MONGO_PASSWORD=$MONGO_PASSWORD
        - MONGO_HOSTNAME=db
        - MONGO_PORT=$MONGO_PORT
        - MONGO_DB=$MONGO_DB
    ports:
        - "80:8080"
    volumes:
        - .:/home/node/app
        - node_modules:/home/node/app/node_modules
    networks:
        - app-network
    command: ./wait-for.sh db:27017 --
/home/node/app/node_modules/.bin/nodemon app.js
...
```

Make the following edits to your service definition: - Use your **node-kubernetes** image instead of the local `Dockerfile`. - Change the container `restart` policy from `unless-stopped` to `always`. - Remove the `volumes` list and the `command` instruction.

The finished service definition will now look like this:

~/node_project/docker-compose.yaml

```
...
services:
  nodejs:
    image: your_dockerhub_username/node-kubernetes
    container_name: nodejs
    restart: always
    env_file: .env
    environment:
      - MONGO_USERNAME=$MONGO_USERNAME
      - MONGO_PASSWORD=$MONGO_PASSWORD
      - MONGO_HOSTNAME=db
      - MONGO_PORT=$MONGO_PORT
      - MONGO_DB=$MONGO_DB
    ports:
      - "80:8080"
    networks:
      - app-network
...
```

Next, scroll down to the `db` service definition. Here, make the following edits: - Change the `restart` policy for the service to `always`. - Remove the `.env` file. Instead of using values from the `.env` file, we will pass the values for our `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` to the database container using the Secret we will create in Step 4.

The `db` service definition will now look like this:

~/node_project/docker-compose.yaml

```
...
  db:
    image: mongo:4.1.8-xenial
    container_name: db
    restart: always
    environment:
      - MONGO_INITDB_ROOT_USERNAME=$MONGO_USERNAME
      - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASSWORD
    volumes:
      - dbdata:/data/db
    networks:
      - app-network
...
```

Finally, at the bottom of the file, remove the `node_modules` volumes from the top-level `volumes` key. The key will now look like this:

~/node_project/docker-compose.yaml

```
...
volumes:
  dbdata:
```

Save and close the file when you are finished editing.

Before translating our service definitions, we will need to write the `.env` file that kompose will use to create the ConfigMap with our non-sensitive information. Please see Step 2 of Containerizing a Node.js Application for Development With Docker Compose for a longer explanation of this file.

In that tutorial, we added `.env` to our `.gitignore` file to ensure that it would not copy to version control. This means that it did not copy over when we cloned the node-mongo-docker-dev repository in Step 2 of this tutorial. We will therefore need to recreate it now.

Create the file:

```
nano .env
```

kompose will use this file to create a ConfigMap for our application. However, instead of assigning all of the variables from the `nodejs` service definition in our Compose file, we will add only the `MONGO_DB` database name and the `MONGO_PORT`. We will assign the database username and password separately when we manually create a Secret object in Step 4.

Add the following port and database name information to the `.env` file. Feel free to rename your database if you would like:

~/node_project/.env

```
MONGO_PORT=27017
MONGO_DB=sharkinfo
```

Save and close the file when you are finished editing.

You are now ready to create the files with your object specs. kompose offers multiple options for translating your resources. You can: - Create `yaml` files based on the service definitions in your `docker-compose.yaml` file with `kompose convert`. - Create Kubernetes objects directly with `kompose up`. - Create a Helm chart with `kompose convert -c`.

For now, we will convert our service definitions to `yaml` files and then add to and revise the files kompose creates.

Convert your service definitions to `yaml` files with the following command:

```
kompose convert
```

You can also name specific or multiple Compose files using the `-f` flag.

After you run this command, kompose will output information about the files it has created:

Output

```
INFO Kubernetes file "nodejs-service.yaml" created
INFO Kubernetes file "db-deployment.yaml" created
INFO Kubernetes file "dbdata-
persistentvolumeclaim.yaml" created
INFO Kubernetes file "nodejs-deployment.yaml"
created
INFO Kubernetes file "nodejs-env-configmap.yaml"
created
```

These include `yaml` files with specs for the Node application Service, Deployment, and ConfigMap, as well as for the `dbdata` PersistentVolumeClaim and MongoDB database Deployment.

These files are a good starting point, but in order for our application's functionality to match the setup described in Containerizing a Node.js Application for Development With Docker Compose we will need to make a few additions and changes to the files kompose has generated.

Step 4 — Creating Kubernetes Secrets

In order for our application to function in the way we expect, we will need to make a few modifications to the files that kompose has created. The first of these changes will be generating a Secret for our database user and password and adding it to our application and database Deployments. Kubernetes offers two ways of working with environment variables: ConfigMaps and Secrets. kompose has already created a ConfigMap with the non-confidential information we included in our `.env` file, so we will now create a Secret with our confidential information: our database username and password.

The first step in manually creating a Secret will be to convert your username and password to base64, an encoding scheme that allows you to uniformly transmit data, including binary data.

Convert your database username:

```
echo -n 'your_database_username' | base64
```

Note down the value you see in the output.

Next, convert your password:

```
echo -n 'your_database_password' | base64
```

Take note of the value in the output here as well.

Open a file for the Secret:

```
nano secret.yaml
```

Note: Kubernetes objects are typically defined using YAML, which strictly forbids tabs and requires two spaces for indentation. If you would like to check the formatting of any of your `yaml` files, you can use a linter or test the validity of your syntax using `kubectl create` with the `--dry-run` and `--validate` flags:

```
kubectl create -f your_yaml_file.yaml --dry-run --
validate=true
```

In general, it is a good idea to validate your syntax before creating resources with `kubectl`.

Add the following code to the file to create a Secret that will define your `MONGO_USERNAME` and `MONGO_PASSWORD` using the encoded values you just created. Be sure to replace the dummy values here with your encoded username and password:

~/node_project/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
data:
  MONGO_USERNAME: your_encoded_username
  MONGO_PASSWORD: your_encoded_password
```

We have named the Secret object **mongo-secret**, but you are free to name it anything you would like.

Save and close this file when you are finished editing. As you did with your `.env` file, be sure to add `secret.yaml` to your `.gitignore` file to keep it out of version control.

With `secret.yaml` written, our next step will be to ensure that our application and database Pods both use the values we added to the file. Let's start by adding references to the Secret to our application Deployment.

Open the file called `nodejs-deployment.yaml`:

```
nano nodejs-deployment.yaml
```

The file's container specifications include the following environment variables defined under the `env` key:

~/node_project/nodejs-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
...
    spec:
      containers:
```

```
    - env:
      - name: MONGO_DB
        valueFrom:
          configMapKeyRef:
            key: MONGO_DB
            name: nodejs-env
      - name: MONGO_HOSTNAME
        value: db
      - name: MONGO_PASSWORD
      - name: MONGO_PORT
        valueFrom:
          configMapKeyRef:
            key: MONGO_PORT
            name: nodejs-env
      - name: MONGO_USERNAME
```

We will need to add references to our Secret to the `MONGO_USERNAME` and `MONGO_PASSWORD` variables listed here, so that our application will have access to those values. Instead of including a `configMapKeyRef` key to point to our `nodejs-env` ConfigMap, as is the case with the values for `MONGO_DB` and `MONGO_PORT`, we'll include a `secretKeyRef` key to point to the values in our **`mongo-secret`** secret.

Add the following Secret references to the `MONGO_USERNAME` and `MONGO_PASSWORD` variables:

~/node_project/nodejs-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
```

```
...
    spec:
      containers:
      - env:
        - name: MONGO_DB
          valueFrom:
            configMapKeyRef:
              key: MONGO_DB
              name: nodejs-env
        - name: MONGO_HOSTNAME
          value: db
        - name: MONGO_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: MONGO_PASSWORD
        - name: MONGO_PORT
          valueFrom:
            configMapKeyRef:
              key: MONGO_PORT
              name: nodejs-env
        - name: MONGO_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: MONGO_USERNAME
```

Save and close the file when you are finished editing.

Next, we'll add the same values to the `db-deployment.yaml` file.

Open the file for editing:

```
nano db-deployment.yaml
```

In this file, we will add references to our Secret for following variable keys: `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD`. The `mongo` image makes these variables available so that you can modify the initialization of your database instance. `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` together create a `root` user in the `admin` authentication database and ensure that authentication is enabled when the database container starts.

Using the values we set in our Secret ensures that we will have an application user with `root` privileges on the database instance, with access to all of the administrative and operational privileges of that role. When working in production, you will want to create a dedicated application user with appropriately scoped privileges.

Under the `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` variables, add references to the Secret values:

~/node_project/db-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
...
    spec:
      containers:
      - env:
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: MONGO_PASSWORD
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: MONGO_USERNAME
        image: mongo:4.1.8-xenial
...
```

Save and close the file when you are finished editing.

With your Secret in place, you can move on to creating your database Service and ensuring that your application container only attempts to connect to the database once it is fully set up and initialized.

Step 5 — Creating the Database Service and an Application Init Container

Now that we have our Secret, we can move on to creating our database Service and an Init Container that will poll this Service to ensure that our application only attempts to connect to the database once the database startup tasks, including creating the `MONGO_INITDB` user and password, are complete.

For a discussion of how to implement this functionality in Compose, please see Step 4 of Containerizing a Node.js Application for Development with Docker Compose.

Open a file to define the specs for the database Service:

```
nano db-service.yaml
```

Add the following code to the file to define the Service:

~/node_project/db-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: kompose convert
    kompose.version: 1.18.0 (06a2e56)
  creationTimestamp: null
  labels:
    io.kompose.service: db
  name: db
spec:
  ports:
  - port: 27017
    targetPort: 27017
  selector:
    io.kompose.service: db
status:
  loadBalancer: {}
```

The `selector` that we have included here will match this Service object with our database Pods, which have been defined with the label `io.kompose.service: db` by kompose in the `db-deployment.yaml` file. We've also named this service `db`.

Save and close the file when you are finished editing.

Next, let's add an Init Container field to the `containers` array in `nodejs-deployment.yaml`. This will create an Init Container that we can use to delay our application container from starting until the `db` Service has been created with a Pod that is reachable. This is one of the possible uses for Init Containers; to learn more about other use cases, please see the official documentation.

Open the `nodejs-deployment.yaml` file:

```
nano nodejs-deployment.yaml
```

Within the Pod spec and alongside the `containers` array, we are going to add an `initContainers` field with a container that will poll the `db` Service.

Add the following code below the `ports` and `resources` fields and above the `restartPolicy` in the `nodejs` `containers` array:

~/node_project/nodejs-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
...
    spec:
      containers:
      ...
        name: nodejs
        ports:
        - containerPort: 8080
        resources: {}
      initContainers:
      - name: init-db
        image: busybox
        command: ['sh', '-c', 'until nc -z
db:27017; do echo waiting for db; sleep 2; done;']
      restartPolicy: Always
...
```

This Init Container uses the BusyBox image, a lightweight image that includes many UNIX utilities. In this case, we'll use the `netcat` utility to poll whether or not the Pod associated with the `db` Service is accepting TCP connections on port `27017`.

This container `command` replicates the functionality of the `wait-for` script that we removed from our `docker-compose.yaml` file in Step 3. For a longer discussion of how and why our application used the `wait-for` script when working with Compose, please see Step 4 of Containerizing a Node.js Application for Development with Docker Compose.

Init Containers run to completion; in our case, this means that our Node application container will not start until the database container is running and accepting connections on port `27017`. The `db` Service definition allows us to guarantee this functionality regardless of the exact location of the database container, which is mutable.

Save and close the file when you are finished editing.

With your database Service created and your Init Container in place to control the startup order of your containers, you can move on to checking the storage requirements in your PersistentVolumeClaim and exposing your application service using a LoadBalancer.

Step 6 — Modifying the PersistentVolumeClaim and Exposing the Application Frontend

Before running our application, we will make two final changes to ensure that our database storage will be provisioned properly and that we can expose our application frontend using a LoadBalancer.

First, let's modify the `storage` `resource` defined in the PersistentVolumeClaim that kompose created for us. This Claim allows us to dynamically provision storage to manage our application's state.

To work with PersistentVolumeClaims, you must have a StorageClass created and configured to provision storage resources. In our case, because we are working with DigitalOcean Kubernetes, our default StorageClass `provisioner` is set to `dobs.csi.digitalocean.com` — DigitalOcean Block Storage.

We can check this by typing:

```
kubectl get storageclass
```

If you are working with a DigitalOcean cluster, you will see the following output:

Output

```
NAME                             PROVISIONER
AGE
do-block-storage (default)
dobs.csi.digitalocean.com   76m
```

If you are not working with a DigitalOcean cluster, you will need to create a StorageClass and configure a `provisioner` of your choice. For details about how to do this, please see the official documentation.

When kompose created `dbdata-persistentvolumeclaim.yaml`, it set the `storage resource` to a size that does not meet the minimum size requirements of our `provisioner`. We will therefore need to modify our PersistentVolumeClaim to use the minimum viable DigitalOcean Block Storage unit: 1GB. Please feel free to modify this to meet your storage requirements.

Open `dbdata-persistentvolumeclaim.yaml`:

```
nano dbdata-persistentvolumeclaim.yaml
```

Replace the `storage` value with **`1Gi`**:

~/node_project/dbdata-persistentvolumeclaim.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: dbdata
  name: dbdata
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
status: {}
```

Also note the `accessMode`: `ReadWriteOnce` means that the volume provisioned as a result of this Claim will be read-write only by a single node. Please see the documentation for more information about different access modes.

Save and close the file when you are finished.

Next, open `nodejs-service.yaml`:

```
nano nodejs-service.yaml
```

We are going to expose this Service externally using a DigitalOcean Load Balancer. If you are not using a DigitalOcean cluster, please consult the relevant documentation from your cloud provider for information about their load balancers. Alternatively, you can follow the official Kubernetes documentation on setting up a highly available cluster with `kubeadm`, but in this case you will not be able to use PersistentVolumeClaims to provision storage.

Within the Service spec, specify `LoadBalancer` as the Service `type`:

~/node_project/nodejs-service.yaml

```
apiVersion: v1
kind: Service
...
spec:
  type: LoadBalancer
  ports:
...
```

When we create the `nodejs` Service, a load balancer will be automatically created, providing us with an external IP where we can access our application.

Save and close the file when you are finished editing.

With all of our files in place, we are ready to start and test the application.

Step 7 — Starting and Accessing the Application

It’s time to create our Kubernetes objects and test that our application is working as expected.

To create the objects we’ve defined, we’ll use `kubectl create` with the `-f` flag, which will allow us to specify the files that kompose created for us, along with the files we wrote. Run the following command to create the Node application and MongoDB database Services and Deployments, along with your Secret, ConfigMap, and PersistentVolumeClaim:

```
kubectl create -f nodejs-service.yaml,nodejs-
deployment.yaml,nodejs-env-configmap.yaml,db-
service.yaml,db-deployment.yaml,dbdata-
persistentvolumeclaim.yaml,secret.yaml
```

You will see the following output indicating that the objects have been created:

Output

```
service/nodejs created
deployment.extensions/nodejs created
configmap/nodejs-env created
service/db created
```

```
deployment.extensions/db created
persistentvolumeclaim/dbdata created
secret/mongo-secret created
```

To check that your Pods are running, type:

```
kubectl get pods
```

You don't need to specify a Namespace here, since we have created our objects in the `default` Namespace. If you are working with multiple Namespaces, be sure to include the `-n` flag when running this command, along with the name of your Namespace.

You will see the following output while your `db` container is starting and your application Init Container is running:

Output

```
NAME                          READY   STATUS
RESTARTS   AGE
db-679d658576-kfpsl           0/1     ContainerCreating
0          10s
nodejs-6b9585dc8b-pnsws       0/1     Init:0/1
0          10s
```

Once that container has run and your application and database containers have started, you will see this output:

Output

```
NAME                          READY   STATUS
RESTARTS   AGE
db-679d658576-kfpsl           1/1     Running   0
54s
```

```
nodejs-6b9585dc8b-pnsws   1/1     Running   0
54s
```

The `Running` `STATUS` indicates that your Pods are bound to nodes and that the containers associated with those Pods are running. `READY` indicates how many containers in a Pod are running. For more information, please consult the documentation on Pod lifecycles.

Note: If you see unexpected phases in the `STATUS` column, remember that you can troubleshoot your Pods with the following commands:

```
kubectl describe pods your_pod
kubectl logs your_pod
```

With your containers running, you can now access the application. To get the IP for the LoadBalancer, type:

```
kubectl get svc
```

You will see the following output:

Output

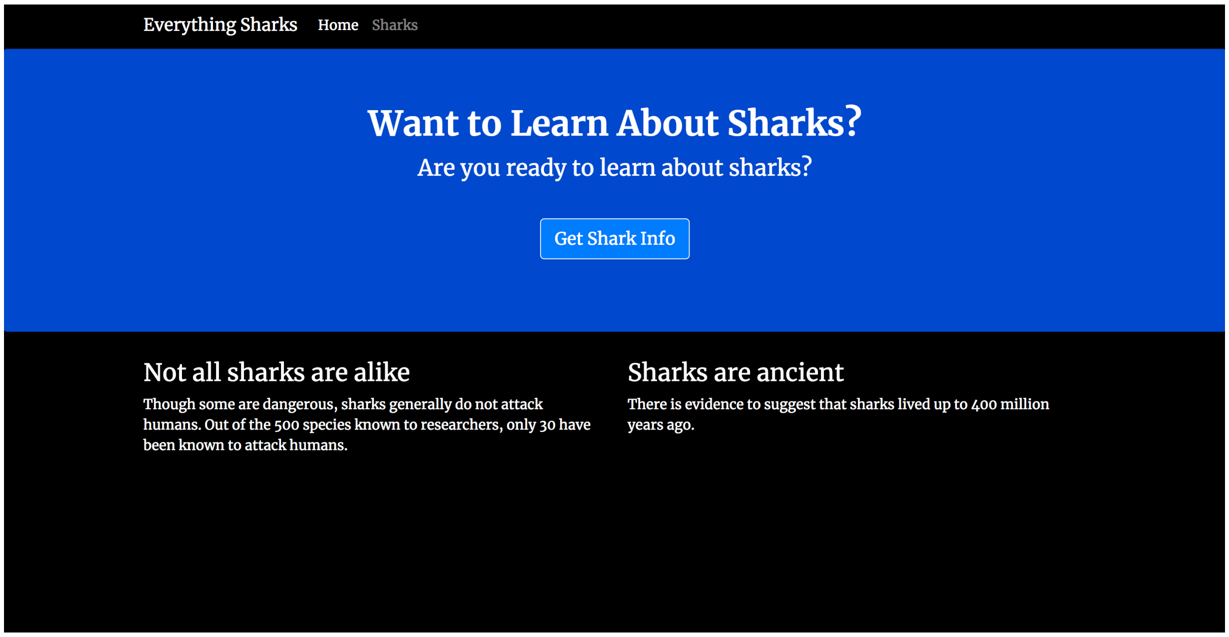
```
NAME          TYPE           CLUSTER-IP
EXTERNAL-IP       PORT(S)         AGE
db            ClusterIP      10.245.189.250   <none>
27017/TCP       93s
kubernetes    ClusterIP      10.245.0.1       <none>
443/TCP         25m12s
nodejs        LoadBalancer   10.245.15.56
your_lb_ip        80:30729/TCP   93s
```

The `EXTERNAL_IP` associated with the `nodejs` service is the IP address where you can access the application. If you see a `<pending>`

status in the `EXTERNAL_IP` column, this means that your load balancer is still being created.

Once you see an IP in that column, navigate to it in your browser: `http://your_lb_ip`.

You should see the following landing page:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

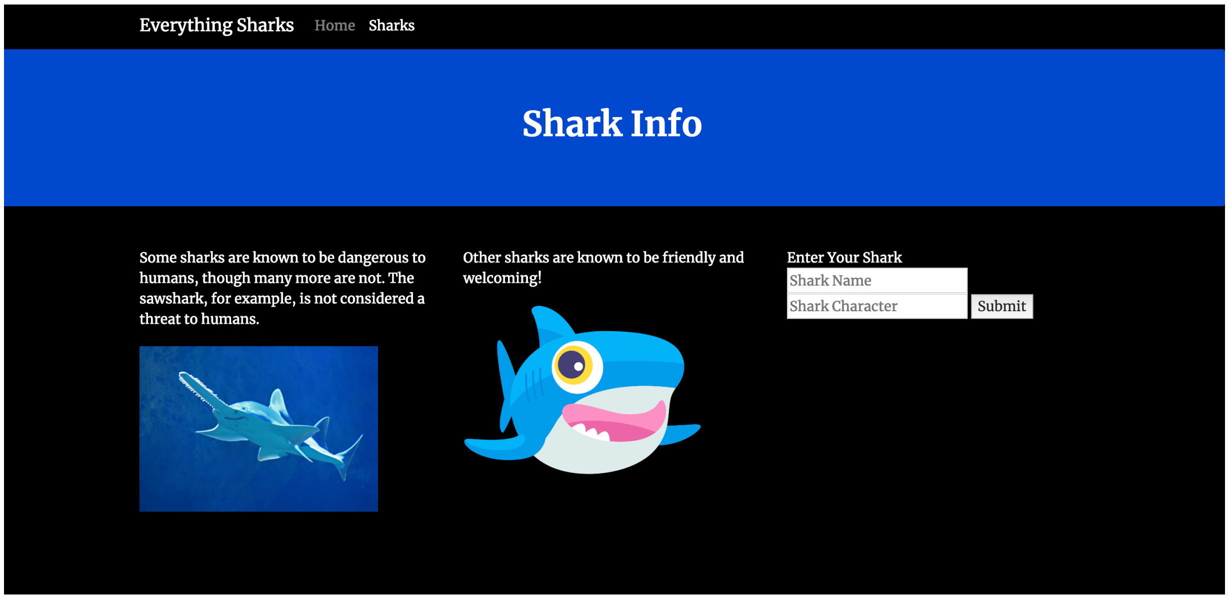
Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Application Landing Page

Click on the Get Shark Info button. You will see a page with an entry form where you can enter a shark name and a description of that shark's general character:



Shark Info

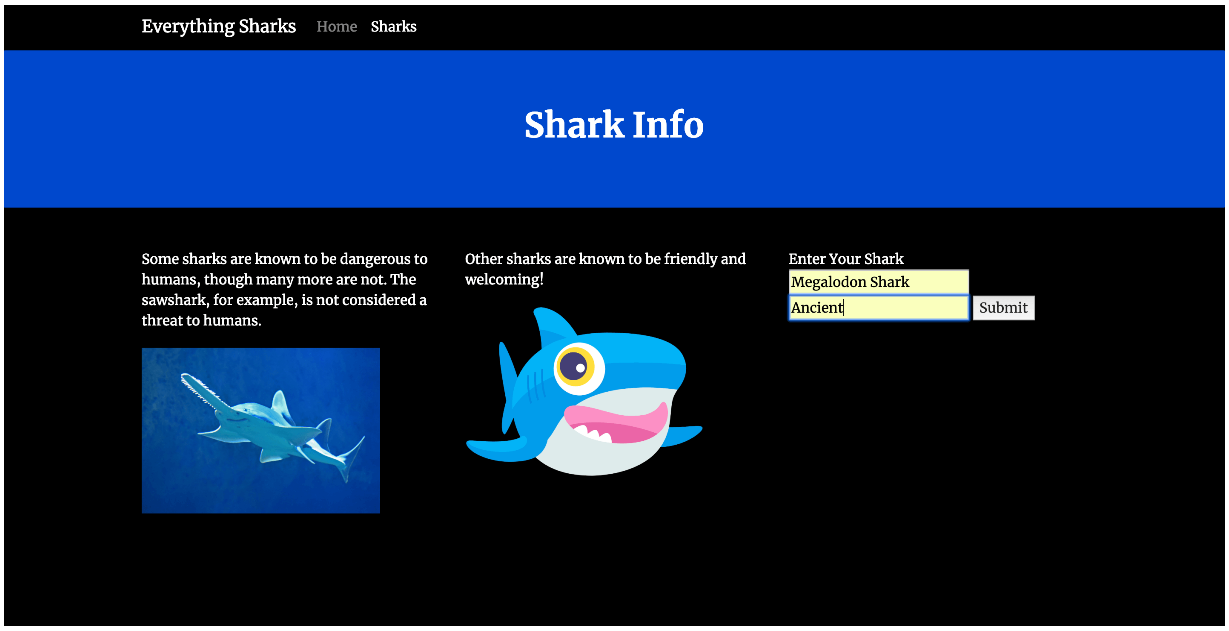
Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

Shark Info Form

In the form, add a shark of your choosing. To demonstrate, we will add `Megalodon Shark` to the Shark Name field, and `Ancient` to the Shark Character field:



Everything Sharks Home Sharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

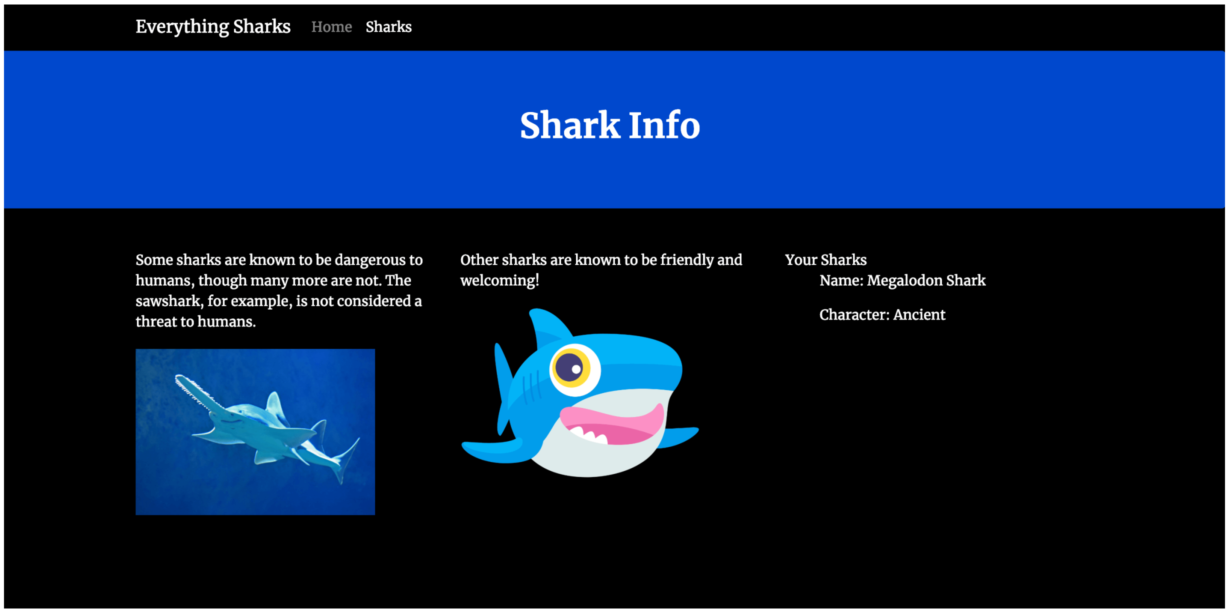
Megalodon Shark

Ancient

Submit

Filled Shark Form

Click on the Submit button. You will see a page with this shark information displayed back to you:



Everything Sharks Home Sharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Your Sharks
Name: Megalodon Shark

Character: Ancient

Shark Output

You now have a single instance setup of a Node.js application with a MongoDB database running on a Kubernetes cluster.

Conclusion

The files you have created in this tutorial are a good starting point to build from as you move toward production. As you develop your application, you can work on implementing the following: - Centralized logging and monitoring. Please see the relevant discussion in Modernizing Applications for Kubernetes for a general overview. You can also look at How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes to learn how to set up a logging stack with Elasticsearch, Fluentd, and Kibana. Also check out An Introduction to Service Meshes for information about how service meshes like Istio implement this functionality. - Ingress Resources to route traffic to your cluster. This is a good alternative to a LoadBalancer in cases where you are running multiple Services, which each require their own LoadBalancer, or where you would like to implement application-level routing strategies (A/B & canary tests, for example). For more information, check out How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes and the related discussion of routing in the service mesh context in An Introduction to Service Meshes. - Backup strategies for your Kubernetes objects. For guidance on implementing backups with Velero (formerly Heptio Ark) with DigitalOcean’s Kubernetes product, please see How To Back Up and Restore a Kubernetes Cluster on DigitalOcean Using Heptio Ark.

Building Optimized Containers for Kubernetes

Written by Justin Ellingwood

In this article you will learn some strategies for creating high-quality images and explore a few general goals to help guide your decisions when containerizing applications. The focus is on building images intended to be run on Kubernetes, but many of the suggestions apply equally to running containers on other orchestration platforms and in other contexts.

There are a number of suggestions and best practices that you will learn about in this tutorial. Some of the more important ones are:

1. Use minimal, shareable parent images to build application images. This strategy will ensure fast image builds and fast container start-up times in a cluster.
2. Combine Dockerfile instructions to create clean image layers and avoid image caching mistakes.
3. Containerize applications by isolating discrete functionality, and design Pods based on applications with a single, focused responsibility.
4. Bundle helper containers to enhance the main container’s functionality or to adapt it to the deployment environment.
5. Run applications as the primary processes in containers so Kubernetes can manage lifecycle events.

Container images are the primary packaging format for defining applications within Kubernetes. Used as the basis for pods and other objects, images play an important role in leveraging Kubernetes' features to efficiently run applications on the platform. Well-designed images are secure, highly performant, and focused. They are able to react to configuration data or instructions provided by Kubernetes and also implement endpoints the orchestration system uses to understand internal application state.

In this article, we'll introduce some strategies for creating high quality images and discuss a few general goals to help guide your decisions when containerizing applications. We will focus on building images intended to be run on Kubernetes, but many of the suggestions apply equally to running containers on other orchestration platforms or in other contexts.

Characteristics of Efficient Container Images

Before we go over specific actions to take when building container images, we will talk about what makes a good container image. What should your goals be when designing new images? Which characteristics and what behavior are most important?

Some qualities to aim for are:

A single, well-defined purpose

Container images should have a single discrete focus. Avoid thinking of container images as virtual machines, where it can make sense to package related functionality together. Instead, treat your container images like Unix utilities, maintaining a strict focus on doing one small thing well.

Applications can be coordinated outside of the container scope to compose complex functionality.

Generic design with the ability to inject configuration at runtime

Container images should be designed with reuse in mind when possible. For instance, the ability to adjust configuration at runtime is often required to fulfill basic requirements like testing your images before deploying to production. Small, generic images can be combined in different configurations to modify behavior without creating new images.

Small image size

Smaller images have a number of benefits in clustered environments like Kubernetes. They download quickly to new nodes and often have a smaller set of installed packages, which can improve security. Pared down container images make it simpler to debug problems by minimizing the amount of software involved.

Externally managed state

Containers in clustered environments experience a very volatile life cycle including planned and unplanned shutdowns due to resource scarcity, scaling, or node failures. To maintain consistency, aid in recovery and availability of your services, and to avoid losing data, it is critical that you store application state in a stable location outside of the container.

Easy to understand

It is important to try to keep container images as simple and easy to understand as possible. When troubleshooting, being able to easily reason about the problem by viewing container image configuration or testing container behavior can help you reach a resolution faster. Thinking of container images as a packaging format for your application instead of a machine configuration can help you strike the right balance.

Follow containerized software best practices

Images should aim to work within the container model instead of acting against it. Avoid implementing conventional system administration practices, like including full init systems and daemonizing applications. Log to standard out so Kubernetes can expose the data to administrators instead of using an internal logging daemon. Each of these differs from best practices for full operating systems.

Fully leverage Kubernetes features

Beyond conforming to the container model, it’s important to understand and reconcile with the environment and tooling that Kubernetes provides. For example, providing endpoints for liveness and readiness checks or adjusting operation based on changes in the configuration or environment can help your applications use Kubernetes’ dynamic deployment environment to their advantage.

Now that we’ve established some of the qualities that define highly functional container images, we can dive deeper into strategies that help you achieve these goals.

Reuse Minimal, Shared Base Layers

We can start off by examining the resources that container images are built from: base images. Each container image is built either from a parent image, an image used as a starting point, or from the abstract `scratch` layer, an empty image layer with no filesystem. A base image is a container image that serves as a foundation for future images by defining the basic operating system and providing core functionality. Images are comprised of one or more image layers built on top of one another to form a final image.

No standard utilities or filesystem are available when working directly from `scratch`, which means that you only have access to extremely limited functionality. While images created directly from `scratch` can be very streamlined and minimal, their main purpose is in defining base images. Typically, you want to build your container images on top of a parent image that sets up a basic environment that your applications run in so that you do not have to construct a complete system for every image.

While there are base images for a variety of Linux distributions, it's best to be deliberate about which systems you choose. Each new machine will have to download the parent image and any additional layers you've added. For large images, this can consume a significant amount of bandwidth and noticeably lengthen the startup time of your containers on their first run. There is no way to pare down an image that's used as a parent downstream in the container build process, so starting with a minimal parent is a good idea.

Feature rich environments like Ubuntu allow your application to run in an environment you're familiar with, but there are some tradeoffs to consider. Ubuntu images (and similar conventional distribution images) tend to be relatively large (over 100MB), meaning that any container images built from them will inherit that weight.

Alpine Linux is a popular alternative for base images because it successfully packages a lot of functionality into a very small base image (~ 5MB). It includes a package manager with sizable repositories and has most of the standard utilities you would expect from a minimal Linux environment.

When designing your applications, it’s a good idea to try to reuse the same parent for each image. When your images share a parent, machines running your containers will download the parent layer only once. Afterwards, they will only need to download the layers that differ between your images. This means that if you have common features or functionality you’d like to embed in each image, creating a common parent image to inherit from might be a good idea. Images that share a lineage help minimize the amount of extra data you need to download on fresh servers.

Managing Container Layers

Once you’ve selected a parent image, you can define your container image by adding additional software, copying files, exposing ports, and choosing processes to run. Certain instructions in the image configuration file (a `Dockerfile` if you are using Docker) will add additional layers to your image.

For many of the same reasons mentioned in the previous section, it’s important to be mindful of how you add layers to your images due to the resulting size, inheritance, and runtime complexity. To avoid building large, unwieldy images, it’s important to develop a good understanding of how container layers interact, how the build engine caches layers, and how subtle differences in similar instructions can have a big impact on the images you create.

Understanding Image Layers and Build Cache

Docker creates a new image layer each time it executes a `RUN`, `COPY`, or `ADD` instruction. If you build the image again, the build engine will check each instruction to see if it has an image layer cached for the operation. If it finds a match in the cache, it uses the existing image layer rather than executing the instruction again and rebuilding the layer.

This process can significantly shorten build times, but it is important to understand the mechanism used to avoid potential problems. For file copying instructions like `COPY` and `ADD`, Docker compares the checksums of the files to see if the operation needs to be performed again. For `RUN` instructions, Docker checks to see if it has an existing image layer cached for that particular command string.

While it might not be immediately obvious, this behavior can cause unexpected results if you are not careful. A common example of this is updating the local package index and installing packages in two separate steps. We will be using Ubuntu for this example, but the basic premise applies equally well to base images for other distributions:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update
RUN apt -y install nginx
. . .
```

Here, the local package index is updated in one `RUN` instruction (`apt -y update`) and Nginx is installed in another operation. This works without issue when it is first used. However, if the Dockerfile is updated later to install an additional package, there may be problems:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update
RUN apt -y install nginx php-fpm
. . .
```

We've added a second package to the installation command run by the second instruction. If a significant amount of time has passed since the previous image build, the new build might fail. That's because the package index update instruction (`RUN apt -y update`) has not changed, so Docker reuses the image layer associated with that instruction. Since we are using an old package index, the version of the `php-fpm` package we have in our local records may no longer be in the repositories, resulting in an error when the second instruction is run.

To avoid this scenario, be sure to consolidate any steps that are interdependent into a single `RUN` instruction so that Docker will re-execute all of the necessary commands when a change occurs:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update && apt -y install nginx php-fpm
. . .
```

The instruction now updates the local package cache whenever the package list changes.

Reducing Image Layer Size by Tweaking RUN Instructions

The previous example demonstrates how Docker's caching behavior can subvert expectations, but there are some other things to keep in mind with how `RUN` instructions interact with Docker's layering system. As mentioned earlier, at the end of each `RUN` instruction, Docker commits the changes as an additional image layer. In order to exert control over the scope of the image layers produced, you can clean up unnecessary files in the final environment that will be committed by paying attention to the artifacts introduced by the commands you run.

In general, chaining commands together into a single `RUN` instruction offers a great deal of control over the layer that will be written. For each command, you can set up the state of the layer (`apt -y update`), perform the core command (`apt install -y nginx php-fpm`), and remove any unnecessary artifacts to clean up the environment before it's committed. For example, many Dockerfiles chain `rm -rf /var/lib/apt/lists/*` to the end of `apt` commands, removing the downloaded package indexes, to reduce the final layer size:

Package installation example Dockerfile

```
FROM ubuntu:18.04
RUN apt -y update && apt -y install nginx php-fpm
&& rm -rf /var/lib/apt/lists/*
. . .
```

To further reduce the size of the image layers you are creating, trying to limit other unintended side effects of the commands you're running can be helpful. For instance, in addition to the explicitly declared packages, `apt` also installs "recommended" packages by default. You can include `--no-install-recommends` to your `apt` commands to remove this behavior. You may have to experiment to find out if you rely on any of the functionality provided by recommended packages.

We've used package management commands in this section as an example, but these same principles apply to other scenarios. The general idea is to construct the prerequisite conditions, execute the minimum viable command, and then clean up any unnecessary artifacts in a single `RUN` command to reduce the overhead of the layer you'll be producing.

Using Multi-stage Builds

Multi-stage builds were introduced in Docker 17.05, allowing developers to more tightly control the final runtime images they produce. Multi-stage builds allow you to divide your Dockerfile into multiple sections representing distinct stages, each with a `FROM` statement to specify separate parent images.

Earlier sections define images that can be used to build your application and prepare assets. These often contain build tools and development files that are needed to produce the application, but are not necessary to run it. Each subsequent stage defined in the file will have access to artifacts produced by previous stages.

The last `FROM` statement defines the image that will be used to run the application. Typically, this is a pared down image that installs only the necessary runtime requirements and then copies the application artifacts produced by previous stages.

This system allows you worry less about optimizing `RUN` instructions in the build stages since those container layers will not be present in the final runtime image. You should still pay attention to how instructions interact with layer caching in the build stages, but your efforts can be directed towards minimizing build time rather than final image size. Paying attention to instructions in the final stage is still important in reducing image size, but by separating the different stages of your container build, it’s easier to to obtain streamlined images without as much Dockerfile complexity.

Scoping Functionality at the Container and Pod Level

While the choices you make regarding container build instructions are important, broader decisions about how to containerize your services often have a more direct impact on your success. In this section, we’ll talk a bit more about how to best transition your applications from a more conventional environment to running on a container platform.

Containerizing by Function

Generally, it is good practice to package each piece of independent functionality into a separate container image.

This differs from common strategies employed in virtual machine environments where applications are frequently grouped together within the same image to reduce the size and minimize the resources required to run the VM. Since containers are lightweight abstractions that don’t virtualize the entire operating system stack, this tradeoff is less compelling on Kubernetes. So while a web stack virtual machine might bundle an Nginx web server with a Gunicorn application server on a single machine to serve a Django application, in Kubernetes these might be split into separate containers.

Designing containers that implement one discrete piece of functionality for your services offers a number of advantages. Each container can be developed independently if standard interfaces between services are established. For instance, the Nginx container could potentially be used to proxy to a number of different backends or could be used as a load balancer if given a different configuration.

Once deployed, each container image can be scaled independently to address varying resource and load constraints. By splitting your applications into multiple container images, you gain flexibility in development, organization, and deployment.

Combining Container Images in Pods

In Kubernetes, pods are the smallest unit that can be directly managed by the control plane. Pods consist of one or more containers along with additional configuration data to tell the platform how those components should be run. The containers within a pod are always scheduled on the same worker node in the cluster and the system automatically restarts failed containers. The pod abstraction is very useful, but it introduces another layer of decisions about how to bundle together the components of your applications.

Like container images, pods also become less flexible when too much functionality is bundled into a single entity. Pods themselves can be scaled using other abstractions, but the containers within cannot be managed or scaled independently. So, to continue using our previous example, the separate Nginx and Gunicorn containers should probably not be bundled together into a single pod so that they can be controlled and deployed separately.

However, there are scenarios where it does make sense to combine functionally different containers as a unit. In general, these can be categorized as situations where an additional container supports or enhances the core functionality of the main container or helps it adapt to its deployment environment. Some common patterns are:

- Sidecar: The secondary container extends the main container’s core functionality by acting in a supporting utility role. For example, the sidecar container might forward logs or update the filesystem when a remote repository changes. The primary container remains focused on its core responsibility, but is enhanced by the features provided by the sidecar.
- Ambassador: An ambassador container is responsible for discovering and connecting to (often complex) external resources. The primary container can connect to an ambassador container on well-known interfaces using the internal pod environment. The ambassador abstracts the backend resources and proxies traffic between the primary container and the resource pool.
- Adaptor: An adaptor container is responsible for normalizing the primary containers interfaces, data, and protocols to align with the properties expected by other components. The primary container can operate using native formats and the adaptor container translates and normalizes the data to communicate with the outside world.

As you might have noticed, each of these patterns support the strategy of building standard, generic primary container images that can then be deployed in a variety contexts and configurations. The secondary containers help bridge the gap between the primary container and the specific deployment environment being used. Some sidecar containers can also be reused to adapt multiple primary containers to the same environmental conditions. These patterns benefit from the shared filesystem and networking namespace provided by the pod abstraction while still allowing independent development and flexible deployment of standardized containers.

Designing for Runtime Configuration

There is some tension between the desire to build standardized, reusable components and the requirements involved in adapting applications to their runtime environment. Runtime configuration is one of the best methods to bridge the gap between these concerns. Components are built to be both general and flexible and the required behavior is outlined at runtime by providing the software with additional configuration information. This standard approach works for containers as well as it does for applications.

Building with runtime configuration in mind requires you to think ahead during both the application development and containerization steps. Applications should be designed to read values from command line parameters, configuration files, or environment variables when they are launched or restarted. This configuration parsing and injection logic must be implemented in code prior to containerization.

When writing a Dockerfile, the container must also be designed with runtime configuration in mind. Containers have a number of mechanisms for providing data at runtime. Users can mount files or directories from the host as volumes within the container to enable file-based configuration. Likewise, environment variables can be passed into the internal container runtime when the container is started. The `CMD` and

`ENTRYPOINT` Dockerfile instructions can also be defined in a way that allows for runtime configuration information to be passed in as command parameters.

Since Kubernetes manipulates higher level objects like pods instead of managing containers directly, there are mechanisms available to define configuration and inject it into the container environment at runtime. Kubernetes ConfigMaps and Secrets allow you to define configuration data separately and then project the values into the container environment as environment variables or files at runtime. ConfigMaps are general purpose objects intended to store configuration data that might vary based on environment, testing stage, etc. Secrets offer a similar interface but are specifically designed for sensitive data, like account passwords or API credentials.

By understanding and correctly using the runtime configuration options available throughout each layer of abstraction, you can build flexible components that take their cues from environment-provided values. This makes it possible to reuse the same container images in very different scenarios, reducing development overhead by improving application flexibility.

Implementing Process Management with Containers

When transitioning to container-based environments, users often start by shifting existing workloads, with few or no changes, to the new system. They package applications in containers by wrapping the tools they are already using in the new abstraction. While it is helpful to use your usual patterns to get migrated applications up and running, dropping in previous implementations within containers can sometimes lead to ineffective design.

Treating Containers like Applications, Not Services

Problems frequently arise when developers implement significant service management functionality within containers. For example, running systemd services within the container or daemonizing web servers may be considered best practices in a normal computing environment, but they often conflict with assumptions inherent in the container model.

Hosts manage container life cycle events by sending signals to the process operating as PID (process ID) 1 inside the container. PID 1 is the first process started, which would be the init system in traditional computing environments. However, because the host can only manage PID 1, using a conventional init system to manage processes within the container sometimes means there is no way to control the primary application. The host can start, stop, or kill the internal init system, but can't manage the primary application directly. The signals sometimes propagate the intended behavior to the running application, but this adds complexity and isn't always necessary.

Most of the time, it is better to simplify the running environment within the container so that PID 1 is running the primary application in the foreground. In cases where multiple processes must be run, PID 1 is responsible for managing the life cycle of subsequent processes. Certain applications, like Apache, handle this natively by spawning and managing workers that handle connections. For other applications, a wrapper script or a very simple init system like dumb-init or the included tini init system can be used in some cases. Regardless of the implementation you choose, the process running as PID 1 within the container should respond appropriately to `TERM` signals sent by Kubernetes to behave as expected.

Managing Container Health in Kubernetes

Kubernetes deployments and services offer life cycle management for long-running processes and reliable, persistent access to applications, even when underlying containers need to be restarted or the implementations themselves change. By extracting the responsibility of monitoring and maintaining service health out of the container, you can leverage the platform's tools for managing healthy workloads.

In order for Kubernetes to manage containers properly, it has to understand whether the applications running within containers are healthy and capable of performing work. To enable this, containers can implement liveness probes: network endpoints or commands that can be used to report application health. Kubernetes will periodically check defined liveness probes to determine if the container is operating as expected. If the container does not respond appropriately, Kubernetes restarts the container in an attempt to reestablish functionality.

Kubernetes also provides readiness probes, a similar construct. Rather than indicating whether the application within a container is healthy, readiness probes determine whether the application is ready to receive traffic. This can be useful when a containerized application has an initialization routine that must complete before it is ready to receive connections. Kubernetes uses readiness probes to determine whether to add a pod to or remove a pod from a service.

Defining endpoints for these two probe types can help Kubernetes manage your containers efficiently and can prevent container life cycle problems from affecting service availability. The mechanisms to respond to these types of health requests must be built into the application itself and must be exposed in the Docker image configuration.

Conclusion

In this guide, we’ve covered some important considerations to keep in mind when running containerized applications in Kubernetes. To reiterate, some of the suggestions we went over were:

- Use minimal, shareable parent images to build images with minimal bloat and reduce startup time
- Use multi-stage builds to separate the container build and runtime environments
- Combine Dockerfile instructions to create clean image layers and avoid image caching mistakes
- Containerize by isolating discrete functionality to enable flexible scaling and management
- Design pods to have a single, focused responsibility
- Bundle helper containers to enhance the main container’s functionality or to adapt it to the deployment environment
- Build applications and containers to respond to runtime configuration to allow greater flexibility when deploying
- Run applications as the primary processes in containers so Kubernetes can manage life cycle events
- Develop health and liveness endpoints within the application or container so that Kubernetes can monitor the health of the container

Throughout the development and implementation process, you will need to make decisions that can affect your service's robustness and effectiveness. Understanding the ways that containerized applications differ from conventional applications, and learning how they operate in a managed cluster environment can help you avoid some common pitfalls and allow you to take advantage of all of the capabilities Kubernetes provides.

How To Scale a Node.js Application with MongoDB on Kubernetes Using Helm

Written by Kathleen Juell

In this tutorial, you will deploy your Node.js shark application with a MongoDB database onto a Kubernetes cluster using Helm charts. You will use the official Helm MongoDB replica set chart to create a StatefulSet object consisting of three Pods, a Headless Service, and three PersistentVolumeClaims. You will also create a chart to deploy a multi-replica Node.js application using a custom application image.

By the end of this tutorial you will have deployed a replicated, highly-available shark information application on a Kubernetes cluster using Helm charts. This demo application and the workflow outlined in this tutorial can act as a starting point as you build custom charts for your application and take advantage of Helm's stable repository and other chart repositories.

Kubernetes is a system for running modern, containerized applications at scale. With it, developers can deploy and manage applications across clusters of machines. And though it can be used to improve efficiency and reliability in single-instance application setups, Kubernetes is designed to run multiple instances of an application across groups of machines.

When creating multi-service deployments with Kubernetes, many developers opt to use the Helm package manager. Helm streamlines the process of creating multiple Kubernetes resources by offering charts and templates that coordinate how these objects interact. It also offers pre-packaged charts for popular open-source projects.

In this tutorial, you will deploy a Node.js application with a MongoDB database onto a Kubernetes cluster using Helm charts. You will use the official Helm MongoDB replica set chart to create a StatefulSet object consisting of three Pods, a Headless Service, and three PersistentVolumeClaims. You will also create a chart to deploy a multi-replica Node.js application using a custom application image. The setup you will build in this tutorial will mirror the functionality of the code described in Containerizing a Node.js Application with Docker Compose and will be a good starting point to build a resilient Node.js application with a MongoDB data store that can scale with your needs.

Prerequisites

To complete this tutorial, you will need: - A Kubernetes 1.10+ cluster with role-based access control (RBAC) enabled. This setup will use a DigitalOcean Kubernetes cluster, but you are free to create a cluster using another method. - The `kubectl` command-line tool installed on your local machine or development server and configured to connect to your cluster. You can read more about installing `kubectl` in the official documentation. - Helm installed on your local machine or development server and Tiller installed on your cluster, following the directions outlined in Steps 1 and 2 of How To Install Software on Kubernetes Clusters with the Helm Package Manager. - Docker installed on your local machine or development server. If you are working with Ubuntu 18.04, follow Steps 1 and 2 of How To Install and Use Docker on Ubuntu 18.04; otherwise, follow the official documentation for information about installing on other operating systems. Be sure to add your non-root user to the `docker` group, as described in Step

2 of the linked tutorial. - A Docker Hub account. For an overview of how to set this up, refer to this introduction to Docker Hub.

Step 1 — Cloning and Packaging the Application

To use our application with Kubernetes, we will need to package it so that the `kubelet agent` can pull the image. Before packaging the application, however, we will need to modify the MongoDB connection URI in the application code to ensure that our application can connect to the members of the replica set that we will create with the Helm `mongodb-replicaset` chart.

Our first step will be to clone the node-mongo-docker-dev repository from the DigitalOcean Community GitHub account. This repository includes the code from the setup described in Containerizing a Node.js Application for Development With Docker Compose, which uses a demo Node.js application with a MongoDB database to demonstrate how to set up a development environment with Docker Compose. You can find more information about the application itself in the series From Containers to Kubernetes with Node.js.

Clone the repository into a directory called **`node_project`**:

```
git clone https://github.com/do-community/node-
mongo-docker-dev.git node_project
```

Navigate to the **`node_project`** directory:

```
cd node_project
```

The **`node_project`** directory contains files and directories for a shark information application that works with user input. It has been modernized to work with containers: sensitive and specific configuration information has been removed from the application code and refactored to be injected at runtime, and the application's state has been offloaded to a MongoDB database.

For more information about designing modern, containerized applications, please see Architecting Applications for Kubernetes and Modernizing Applications for Kubernetes.

When we deploy the Helm `mongodb-replicaset` chart, it will create: - A StatefulSet object with three Pods — the members of the MongoDB replica set. Each Pod will have an associated PersistentVolumeClaim and will maintain a fixed identity in the event of rescheduling. - A MongoDB replica set made up of the Pods in the StatefulSet. The set will include one primary and two secondaries. Data will be replicated from the primary to the secondaries, ensuring that our application data remains highly available.

For our application to interact with the database replicas, the MongoDB connection URI in our code will need to include both the hostnames of the replica set members as well as the name of the replica set itself. We therefore need to include these values in the URI.

The file in our cloned repository that specifies database connection information is called `db.js`. Open that file now using `nano` or your favorite editor:

```
nano db.js
```

Currently, the file includes constants that are referenced in the database connection URI at runtime. The values for these constants are injected using Node's `process.env` property, which returns an object with information about your user environment at runtime. Setting values dynamically in our application code allows us to decouple the code from the underlying infrastructure, which is necessary in a dynamic, stateless environment. For more information about refactoring application code in this way, see Step 2 of Containerizing a Node.js Application for Development With Docker Compose and the relevant discussion in The 12-Factor App.

The constants for the connection URI and the URI string itself currently look like this:

~/node_project/db.js

```
...
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB
} = process.env;


...

const url =
`mongodb://${MONGO_USERNAME}:${MONGO_PASSWORD}@${MON
GO_HOSTNAME}:${MONGO_PORT}/${MONGO_DB}?
authSource=admin`;
...
```

In keeping with a 12FA approach, we do not want to hard code the hostnames of our replica instances or our replica set name into this URI string. The existing `MONGO_HOSTNAME` constant can be expanded to include multiple hostnames — the members of our replica set — so we will leave that in place. We will need to add a replica set constant to the `options` section of the URI string, however.

Add MONGO_REPLICASET to both the URI constant object and the connection string:

~/node_project/db.js

```
...
const {
  MONGO_USERNAME,
  MONGO_PASSWORD,
  MONGO_HOSTNAME,
  MONGO_PORT,
  MONGO_DB,
  MONGO_REPLICASET
} = process.env;

...
const url =
`mongodb://${MONGO_USERNAME}:${MONGO_PASSWORD}@${MONGO_HOSTNAME}:${MONGO_PORT}/${MONGO_DB}?
replicaSet=${MONGO_REPLICASET}&authSource=admin`;
...
```

Using the replicaSet option in the options section of the URI allows us to pass in the name of the replica set, which, along with the hostnames defined in the MONGO_HOSTNAME constant, will allow us to connect to the set members.

Save and close the file when you are finished editing.

With your database connection information modified to work with replica sets, you can now package your application, build the image with the docker build command, and push it to Docker Hub.

Build the image with `docker build` and the `-t` flag, which allows you to tag the image with a memorable name. In this case, tag the image with your Docker Hub username and name it **`node-replicas`** or a name of your own choosing:

```
docker build -t your_dockerhub_username/node-
replicas .
```

The `.` in the command specifies that the build context is the current directory.

It will take a minute or two to build the image. Once it is complete, check your images:

```
docker images
```

You will see the following output:

Output

```
REPOSITORY                                  TAG
IMAGE ID            CREATED                 SIZE
your_dockerhub_username/node-replicas       latest
56a69b4bc882        7 seconds ago           90.1MB
node                                        10-alpine
aa57b0242b33        6 days ago              71MB
```

Next, log in to the Docker Hub account you created in the prerequisites:

```
docker login -u your_dockerhub_username
```

When prompted, enter your Docker Hub account password. Logging in this way will create a `~/.docker/config.json` file in your non-root user's home directory with your Docker Hub credentials.

Push the application image to Docker Hub with the [`docker push` command](). Remember to replace **`your_dockerhub_username`** with your own Docker Hub username:

```
docker push your_dockerhub_username/node-replicas
```

You now have an application image that you can pull to run your replicated application with Kubernetes. The next step will be to configure specific parameters to use with the MongoDB Helm chart.

Step 2 — Creating Secrets for the MongoDB Replica Set

The `stable/mongodb-replicaset` chart provides different options when it comes to using Secrets, and we will create two to use with our chart deployment: - A Secret for our replica set keyfile that will function as a shared password between replica set members, allowing them to authenticate other members. - A Secret for our MongoDB admin user, who will be created as a root user on the `admin` database. This role will allow you to create subsequent users with limited permissions when deploying your application to production.

With these Secrets in place, we will be able to set our preferred parameter values in a dedicated values file and create the StatefulSet object and MongoDB replica set with the Helm chart.

First, let’s create the keyfile. We will use the `openssl` command with the `rand` option to generate a 756 byte random string for the keyfile:

```
openssl rand -base64 756 > key.txt
```

The output generated by the command will be base64 encoded, ensuring uniform data transmission, and redirected to a file called `key.txt`, following the guidelines stated in the `mongodb-replicaset` chart authentication documentation. The key itself must be between 6 and 1024 characters long, consisting only of characters in the base64 set.

You can now create a Secret called **`keyfilesecret`** using this file with `kubectl create`:

```
kubectl create secret generic keyfilesecret --from-file=key.txt
```

This will create a Secret object in the `default` namespace, since we have not created a specific namespace for our setup.

You will see the following output indicating that your Secret has been created:

Output
```
secret/keyfilesecret created
```

Remove `key.txt`:

```
rm key.txt
```

Alternatively, if you would like to save the file, be sure restrict its permissions and add it to your `.gitignore` file to keep it out of version control.

Next, create the Secret for your MongoDB admin user. The first step will be to convert your desired username and password to base64.

Convert your database username:

```
echo -n 'your_database_username' | base64
```

Note down the value you see in the output.

Next, convert your password:

```
echo -n 'your_database_password' | base64
```

Take note of the value in the output here as well.

Open a file for the Secret:

```
nano secret.yaml
```

Note: Kubernetes objects are typically defined using YAML, which strictly forbids tabs and requires two spaces for indentation. If you would like to check the formatting of any of your YAML files, you can use a linter or test the validity of your syntax using `kubectl create` with the `--dry-run` and `--validate` flags:

```
kubectl create -f your_yaml_file.yaml --dry-run --validate=true
```

In general, it is a good idea to validate your syntax before creating resources with `kubectl`.

Add the following code to the file to create a Secret that will define a `user` and `password` with the encoded values you just created. Be sure to replace the dummy values here with your own encoded username and password:

~/node_project/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
data:
  user: your_encoded_username
  password: your_encoded_password
```

Here, we're using the key names that the `mongodb-replicaset` chart expects: `user` and `password`. We have named the Secret object **`mongo-secret`**, but you are free to name it anything you would like.

Save and close the file when you are finished editing.

Create the Secret object with the following command:

```
kubectl create -f secret.yaml
```

You will see the following output:

Output

```
secret/mongo-secret created
```

Again, you can either remove `secret.yaml` or restrict its permissions and add it to your `.gitignore` file.

With your Secret objects created, you can move on to specifying the parameter values you will use with the `mongodb-replicaset` chart and creating the MongoDB deployment.

Step 3 — Configuring the MongoDB Helm Chart and Creating a Deployment

Helm comes with an actively maintained repository called stable that contains the chart we will be using: `mongodb-replicaset`. To use this chart with the Secrets we've just created, we will create a file with configuration parameter values called `mongodb-values.yaml` and then install the chart using this file.

Our `mongodb-values.yaml` file will largely mirror the default `values.yaml` file in the `mongodb-replicaset` chart repository. We will, however, make the following changes to our file: - We will set the `auth` parameter to `true` to ensure that our database instances start with authorization enabled. This means that all clients will be required to authenticate for access to database resources and operations. - We will add information about the Secrets we created in the previous Step so that the chart can use these values to create the replica set keyfile and admin user. - We will decrease the size of the PersistentVolumes associated with each Pod in the StatefulSet to use the minimum viable DigitalOcean Block Storage unit, 1GB, though you are free to modify this to meet your storage requirements.

Before writing the `mongodb-values.yaml` file, however, you should first check that you have a StorageClass created and configured to provision storage resources. Each of the Pods in your database StatefulSet will have a sticky identity and an associated PersistentVolumeClaim, which will dynamically provision a PersistentVolume for the Pod. If a Pod is rescheduled, the PersistentVolume will be mounted to whichever node the Pod is scheduled on (though each Volume must be manually deleted if its associated Pod or StatefulSet is permanently deleted).

Because we are working with DigitalOcean Kubernetes, our default StorageClass `provisioner` is set to `dobs.csi.digitalocean.com` — DigitalOcean Block Storage — which we can check by typing:

```
kubectl get storageclass
```

If you are working with a DigitalOcean cluster, you will see the following output:

Output

```
NAME                             PROVISIONER
AGE
do-block-storage (default)
dobs.csi.digitalocean.com   21m
```

If you are not working with a DigitalOcean cluster, you will need to create a StorageClass and configure a `provisioner` of your choice. For details about how to do this, please see the official documentation.

Now that you have ensured that you have a StorageClass configured, open `mongodb-values.yaml` for editing:

```
nano mongodb-values.yaml
```

You will set values in this file that will do the following: - Enable authorization. - Reference your **`keyfilesecret`** and **`mongo-secret`** objects. - Specify `1Gi` for your PersistentVolumes. - Set your replica set name to `db`. - Specify `3` replicas for the set. - Pin the `mongo` image to the latest version at the time of writing: `4.1.9`.

Paste the following code into the file:

~/node_project/mongodb-values.yaml

```
replicas: 3
port: 27017
replicaSetName: db
podDisruptionBudget: {}
auth:
  enabled: true
  existingKeySecret: keyfilesecret
  existingAdminSecret: mongo-secret
imagePullSecrets: []
installImage:
  repository: unguiculus/mongodb-install
  tag: 0.7
  pullPolicy: Always
copyConfigImage:
  repository: busybox
  tag: 1.29.3
  pullPolicy: Always
image:
  repository: mongo
  tag: 4.1.9
  pullPolicy: Always
```

```
extraVars: {}
metrics:
  enabled: false
  image:
    repository: ssalaues/mongodb-exporter
    tag: 0.6.1
    pullPolicy: IfNotPresent
  port: 9216
  path: /metrics
  socketTimeout: 3s
  syncTimeout: 1m
  prometheusServiceDiscovery: true
  resources: {}
podAnnotations: {}
securityContext:
  enabled: true
  runAsUser: 999
  fsGroup: 999
  runAsNonRoot: true
init:
  resources: {}
  timeout: 900
resources: {}
nodeSelector: {}
affinity: {}
tolerations: []
extraLabels: {}
```

```
persistentVolume:
  enabled: true
  #storageClass: "-"
  accessModes:
    - ReadWriteOnce
  size: 1Gi
  annotations: {}
serviceAnnotations: {}
terminationGracePeriodSeconds: 30
tls:
  enabled: false
configmap: {}
readinessProbe:
  initialDelaySeconds: 5
  timeoutSeconds: 1
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
livenessProbe:
  initialDelaySeconds: 30
  timeoutSeconds: 5
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
```

The `persistentVolume.storageClass` parameter is commented out here: removing the comment and setting its value to `"-"` would disable dynamic provisioning. In our case, because we are leaving this value undefined, the chart will choose the default `provisioner` — in our case, `dobs.csi.digitalocean.com`.

Also note the `accessMode` associated with the `persistentVolume` key: `ReadWriteOnce` means that the provisioned volume will be read-write only by a single node. Please see the documentation for more information about different access modes.

To learn more about the other parameters included in the file, see the configuration table included with the repo.

Save and close the file when you are finished editing.

Before deploying the `mongodb-replicaset` chart, you will want to update the stable repo with the `helm repo update` command:

```
helm repo update
```

This will get the latest chart information from the stable repository.

Finally, install the chart with the following command:

```
helm install --name mongo -f mongodb-values.yaml
stable/mongodb-replicaset
```

Note: Before installing a chart, you can run `helm install` with the `--dry-run` and `--debug` options to check the generated manifests for your release:

```
helm install --name your_release_name -f
your_values_file.yaml --dry-run --debug your_chart
```

Note that we are naming the Helm release **mongo**. This name will refer to this particular deployment of the chart with the configuration options we've specified. We've pointed to these options by including the `-f` flag and our `mongodb-values.yaml` file.

Also note that because we did not include the `--namespace` flag with `helm install`, our chart objects will be created in the `default` namespace.

Once you have created the release, you will see output about its status, along with information about the created objects and instructions for interacting with them:

Output

```
NAME:   mongo
LAST DEPLOYED: Tue Apr 16 21:51:05 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                                DATA  AGE
mongo-mongodb-replicaset-init       1     1s
mongo-mongodb-replicaset-mongodb    1     1s
mongo-mongodb-replicaset-tests      1     0s
...
```

You can now check on the creation of your Pods with the following command:

```
kubectl get pods
```

You will see output like the following as the Pods are being created:

Output

```
NAME                             READY   STATUS
RESTARTS   AGE
mongo-mongodb-replicaset-0   1/1     Running    0
67s
```

```
mongo-mongodb-replicaset-1   0/1     Init:0/3   0
8s
```

The `READY` and `STATUS` outputs here indicate that the Pods in our StatefulSet are not fully ready: the Init Containers associated with the Pod's containers are still running. Because StatefulSet members are created in sequential order, each Pod in the StatefulSet must be `Running` and `Ready` before the next Pod will be created.

Once the Pods have been created and all of their associated containers are running, you will see this output:

Output

```
NAME                         READY   STATUS
RESTARTS   AGE
mongo-mongodb-replicaset-0   1/1     Running   0
2m33s
mongo-mongodb-replicaset-1   1/1     Running   0
94s
mongo-mongodb-replicaset-2   1/1     Running   0
36s
```

The `Running` `STATUS` indicates that your Pods are bound to nodes and that the containers associated with those Pods are running. `READY` indicates how many containers in a Pod are running. For more information, please consult the documentation on Pod lifecycles.

Note: If you see unexpected phases in the `STATUS` column, remember that you can troubleshoot your Pods with the following commands:

```
kubectl describe pods your_pod
kubectl logs your_pod
```

Each of the Pods in your StatefulSet has a name that combines the name of the StatefulSet with the ordinal index of the Pod. Because we created three replicas, our StatefulSet members are numbered 0-2, and each has a stable DNS entry comprised of the following elements: `$(`**`statefulset-name`**`)-$(`**`ordinal`**`).$(`**`service name`**`).$(`**`namespace`**`).svc.cluster.local`.

In our case, the StatefulSet and the Headless Service created by the `mongodb-replicaset` chart have the same names:

```
kubectl get statefulset
```

Output

```
NAME                          READY   AGE
mongo-mongodb-replicaset      3/3     4m2s
```

```
kubectl get svc
```

Output

```
NAME                                 TYPE
CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
kubernetes                           ClusterIP
10.245.0.1   <none>        443/TCP     42m
mongo-mongodb-replicaset             ClusterIP   None
<none>        27017/TCP   4m35s
mongo-mongodb-replicaset-client      ClusterIP   None
<none>        27017/TCP   4m35s
```

This means that the first member of our StatefulSet will have the following DNS entry:

```
mongo-mongodb-replicaset-0.mongo-mongodb-
replicaset.default.svc.cluster.local
```

Because we need our application to connect to each MongoDB instance, it's essential that we have this information so that we can communicate directly with the Pods, rather than with the Service. When we create our custom application Helm chart, we will pass the DNS entries for each Pod to our application using environment variables.

With your database instances up and running, you are ready to create the chart for your Node application.

Step 4 — Creating a Custom Application Chart and Configuring Parameters

We will create a custom Helm chart for our Node application and modify the default files in the standard chart directory so that our application can work with the replica set we have just created. We will also create files to define ConfigMap and Secret objects for our application.

First, create a new chart directory called **`nodeapp`** with the following command:

```
helm create nodeapp
```

This will create a directory called **`nodeapp`** in your `~/`**`node_project`** folder with the following resources: - A `Chart.yaml` file with basic information about your chart. - A `values.yaml` file that allows you to set specific parameter values, as you did with your MongoDB deployment. - A `.helmignore` file with file and directory patterns that will be ignored when packaging charts. - A `templates/` directory with the template files that will generate Kubernetes manifests. - A `templates/tests/` directory for test files. - A `charts/` directory for any charts that this chart depends on.

The first file we will modify out of these default files is `values.yaml`. Open that file now:

```
nano nodeapp/values.yaml
```

The values that we will set here include: - The number of replicas. - The application image we want to use. In our case, this will be the `node-replicas` image we created in Step 1. - The ServiceType. In this case, we will specify LoadBalancer to create a point of access to our application for testing purposes. Because we are working with a DigitalOcean Kubernetes cluster, this will create a DigitalOcean Load Balancer when we deploy our chart. In production, you can configure your chart to use Ingress Resources and Ingress Controllers to route traffic to your Services. - The targetPort to specify the port on the Pod where our application will be exposed.

We will not enter environment variables into this file. Instead, we will create templates for ConfigMap and Secret objects and add these values to our application Deployment manifest, located at `~/node_project/nodeapp/templates/deployment.yaml`.

Configure the following values in the `values.yaml` file:

~/node_project/nodeapp/values.yaml

```
# Default values for nodeapp.
# This is a YAML-formatted file.
# Declare variables to be passed into your
templates.

replicaCount: 3

image:
  repository: your_dockerhub_username/node-replicas
  tag: latest
  pullPolicy: IfNotPresent

nameOverride: ""
fullnameOverride: ""

service:
  type: LoadBalancer
  port: 80
  targetPort: 8080
...
```

Save and close the file when you are finished editing.

Next, open a `secret.yaml` file in the **nodeapp**/`templates` directory:

```
nano nodeapp/templates/secret.yaml
```

In this file, add values for your `MONGO_USERNAME` and `MONGO_PASSWORD` application constants. These are the constants that your application will expect to have access to at runtime, as specified in `db.js`, your database connection file. As you add the values for these constants, remember to the use the base64-encoded values that you used earlier in Step 2 when creating your **mongo-secret** object. If you need to recreate those values, you can return to Step 2 and run the relevant commands again.

Add the following code to the file:

~/node_project/nodeapp/templates/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-auth
data:
  MONGO_USERNAME: your_encoded_username
  MONGO_PASSWORD: your_encoded_password
```

The name of this Secret object will depend on the name of your Helm release, which you will specify when you deploy the application chart.

Save and close the file when you are finished.

Next, open a file to create a ConfigMap for your application:

```
nano nodeapp/templates/configmap.yaml
```

In this file, we will define the remaining variables that our application expects: `MONGO_HOSTNAME`, `MONGO_PORT`, `MONGO_DB`, and `MONGO_REPLICASET`. Our `MONGO_HOSTNAME` variable will include the DNS entry for each instance in our replica set, since this is what the MongoDB connection URI requires.

According to the Kubernetes documentation, when an application implements liveness and readiness checks, SRV records should be used when connecting to the Pods. As discussed in Step 3, our Pod SRV records follow this pattern: `$(statefulset-name)-$(ordinal).$(service name).$(namespace).svc.cluster.local`. Since our MongoDB StatefulSet implements liveness and readiness checks, we should use these stable identifiers when defining the values of the `MONGO_HOSTNAME` variable.

Add the following code to the file to define the `MONGO_HOSTNAME`, `MONGO_PORT`, `MONGO_DB`, and `MONGO_REPLICASET` variables. You are free to use another name for your `MONGO_DB` database, but your `MONGO_HOSTNAME` and `MONGO_REPLICASET` values must be written as they appear here:

~/node_project/nodeapp/templates/configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-config
data:
  MONGO_HOSTNAME: "mongo-mongodb-replicaset-0.mongo-
mongodb-replicaset.default.svc.cluster.local,mongo-
mongodb-replicaset-1.mongo-mongodb-
replicaset.default.svc.cluster.local,mongo-mongodb-
replicaset-2.mongo-mongodb-
replicaset.default.svc.cluster.local"
  MONGO_PORT: "27017"
  MONGO_DB: "sharkinfo"
  MONGO_REPLICASET: "db"
```

Because we have already created the StatefulSet object and replica set, the hostnames that are listed here must be listed in your file exactly as they appear in this example. If you destroy these objects and rename your MongoDB Helm release, then you will need to revise the values included in this ConfigMap. The same applies for `MONGO_REPLICASET`, since we specified the replica set name with our MongoDB release.

Also note that the values listed here are quoted, which is the expectation for environment variables in Helm.

Save and close the file when you are finished editing.

With your chart parameter values defined and your Secret and ConfigMap manifests created, you can edit the application Deployment template to use your environment variables.

Step 5 — Integrating Environment Variables into Your Helm Deployment

With the files for our application Secret and ConfigMap in place, we will need to make sure that our application Deployment can use these values. We will also customize the liveness and readiness probes that are already defined in the Deployment manifest.

Open the application Deployment template for editing:

```
nano nodeapp/templates/deployment.yaml
```

Though this is a YAML file, Helm templates use a different syntax from standard Kubernetes YAML files in order to generate manifests. For more information about templates, see the Helm documentation.

In the file, first add an `env` key to your application container specifications, below the `imagePullPolicy` key and above `ports`:

~/node_project/nodeapp/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
  spec:
    containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{
.Values.image.tag }}"
        imagePullPolicy: {{ .Values.image.pullPolicy
}}
```

```
        env:
        ports:
```

Next, add the following keys to the list of `env` variables:

~/node_project/nodeapp/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
  spec:
    containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{
.Values.image.tag }}"
        imagePullPolicy: {{ .Values.image.pullPolicy
}}
        env:
        - name: MONGO_USERNAME
          valueFrom:
            secretKeyRef:
              key: MONGO_USERNAME
              name: {{ .Release.Name }}-auth
        - name: MONGO_PASSWORD
          valueFrom:
            secretKeyRef:
              key: MONGO_PASSWORD
              name: {{ .Release.Name }}-auth
```

```
- name: MONGO_HOSTNAME
  valueFrom:
    configMapKeyRef:
      key: MONGO_HOSTNAME
      name: {{ .Release.Name }}-config
- name: MONGO_PORT
  valueFrom:
    configMapKeyRef:
      key: MONGO_PORT
      name: {{ .Release.Name }}-config
- name: MONGO_DB
  valueFrom:
    configMapKeyRef:
      key: MONGO_DB
      name: {{ .Release.Name }}-config
- name: MONGO_REPLICASET
  valueFrom:
    configMapKeyRef:
      key: MONGO_REPLICASET
      name: {{ .Release.Name }}-config
```

Each variable includes a reference to its value, defined either by a `secretKeyRef` key, in the case of Secret values, or `configMapKeyRef` for ConfigMap values. These keys point to the Secret and ConfigMap files we created in the previous Step.

Next, under the `ports` key, modify the `containerPort` definition to specify the port on the container where our application will be exposed:

~/node_project/nodeapp/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
  spec:
    containers:
    ...
      env:
    ...
      ports:
        - name: http
          containerPort: 8080
          protocol: TCP
      ...
```

Next, let’s modify the liveness and readiness checks that are included in this Deployment manifest by default. These checks ensure that our application Pods are running and ready to serve traffic: - Readiness probes assess whether or not a Pod is ready to serve traffic, stopping all requests to the Pod until the checks succeed. - Liveness probes check basic application behavior to determine whether or not the application in the container is running and behaving as expected. If a liveness probe fails, Kubernetes will restart the container.

For more about both, see the relevant discussion in Architecting Applications for Kubernetes.

In our case, we will build on the `httpGet` request that Helm has provided by default and test whether or not our application is accepting requests on the /sharks endpoint. The kubelet service will perform the probe by sending a GET request to the Node server running in the application Pod's container and listening on port 8080. If the status code for the response is between 200 and 400, then the kubelet will conclude that the container is healthy. Otherwise, in the case of a 400 or 500 status, kubelet will either stop traffic to the container, in the case of the readiness probe, or restart the container, in the case of the liveness probe.

Add the following modification to the stated path for the liveness and readiness probes:

~/node_project/nodeapp/templates/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
  spec:
    containers:
    ...
      env:
    ...
      ports:
        - name: http
          containerPort: 8080
          protocol: TCP
      livenessProbe:
        httpGet:
          path: /sharks
          port: http
      readinessProbe:
        httpGet:
          path: /sharks
          port: http
```

Save and close the file when you are finished editing.

You are now ready to create your application release with Helm. Run the following `helm install` command, which includes the name of the release and the location of the chart directory:

```
helm install --name nodejs ./nodeapp
```

Remember that you can run `helm install` with the `--dry-run` and `--debug` options first, as discussed in Step 3, to check the generated manifests for your release.

Again, because we are not including the `--namespace` flag with `helm install`, our chart objects will be created in the `default` namespace.

You will see the following output indicating that your release has been created:

Output

```
NAME:   nodejs
LAST DEPLOYED: Wed Apr 17 18:10:29 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME           DATA  AGE
nodejs-config  4     1s
```

```
==> v1/Deployment
NAME              READY  UP-TO-DATE  AVAILABLE  AGE
nodejs-nodeapp    0/3    3            0          1s

...
```

Again, the output will indicate the status of the release, along with information about the created objects and how you can interact with them.

Check the status of your Pods:

```
kubectl get pods
```

Output

```
NAME                                   READY    STATUS
RESTARTS   AGE
mongo-mongodb-replicaset-0             1/1      Running
0          57m
mongo-mongodb-replicaset-1             1/1      Running
0          56m
mongo-mongodb-replicaset-2             1/1      Running
0          55m
nodejs-nodeapp-577df49dcc-b5fq5        1/1      Running
0          117s
nodejs-nodeapp-577df49dcc-bkk66        1/1      Running
0          117s
nodejs-nodeapp-577df49dcc-lpmt2        1/1      Running
0          117s
```

Once your Pods are up and running, check your Services:

```
kubectl get svc
```

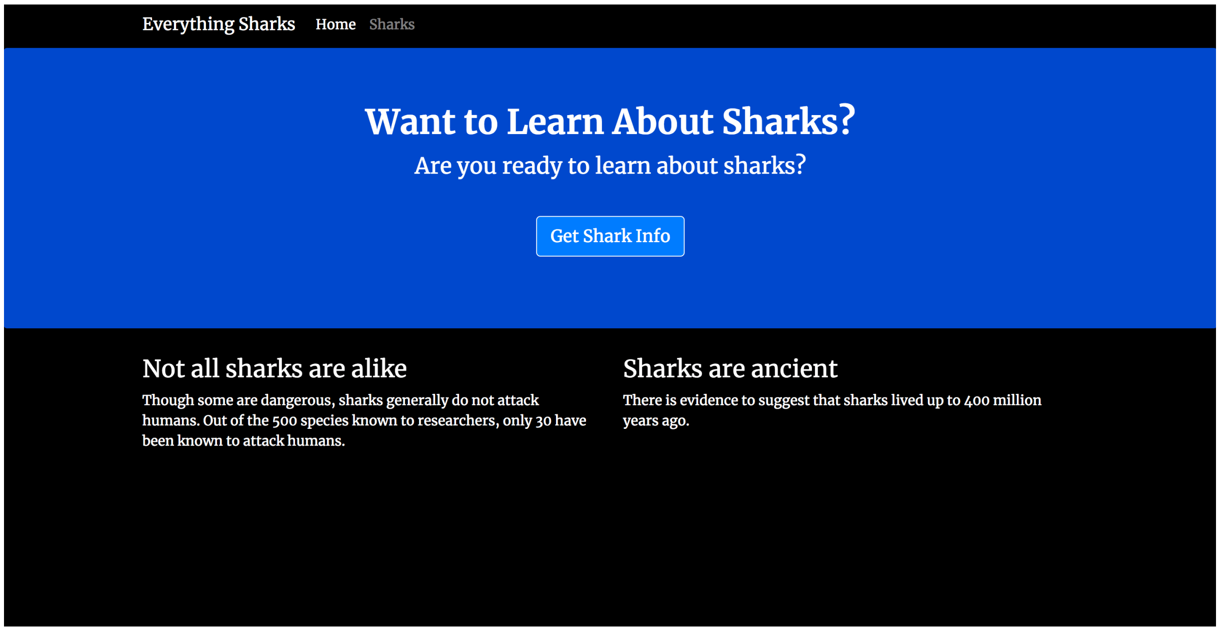
Output

```
NAME                                 TYPE
CLUSTER-IP      EXTERNAL-IP         PORT(S)          AGE
kubernetes                           ClusterIP
10.245.0.1      <none>              443/TCP          96m
mongo-mongodb-replicaset             ClusterIP
None            <none>              27017/TCP        58m
mongo-mongodb-replicaset-client      ClusterIP
None            <none>              27017/TCP        58m
nodejs-nodeapp                       LoadBalancer
10.245.33.46    your_lb_ip          80:31518/TCP
3m22s
```

The `EXTERNAL_IP` associated with the `nodejs-nodeapp` Service is the IP address where you can access the application from outside of the cluster. If you see a `<pending>` status in the `EXTERNAL_IP` column, this means that your load balancer is still being created.

Once you see an IP in that column, navigate to it in your browser: `http://`**`your_lb_ip`**.

You should see the following landing page:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

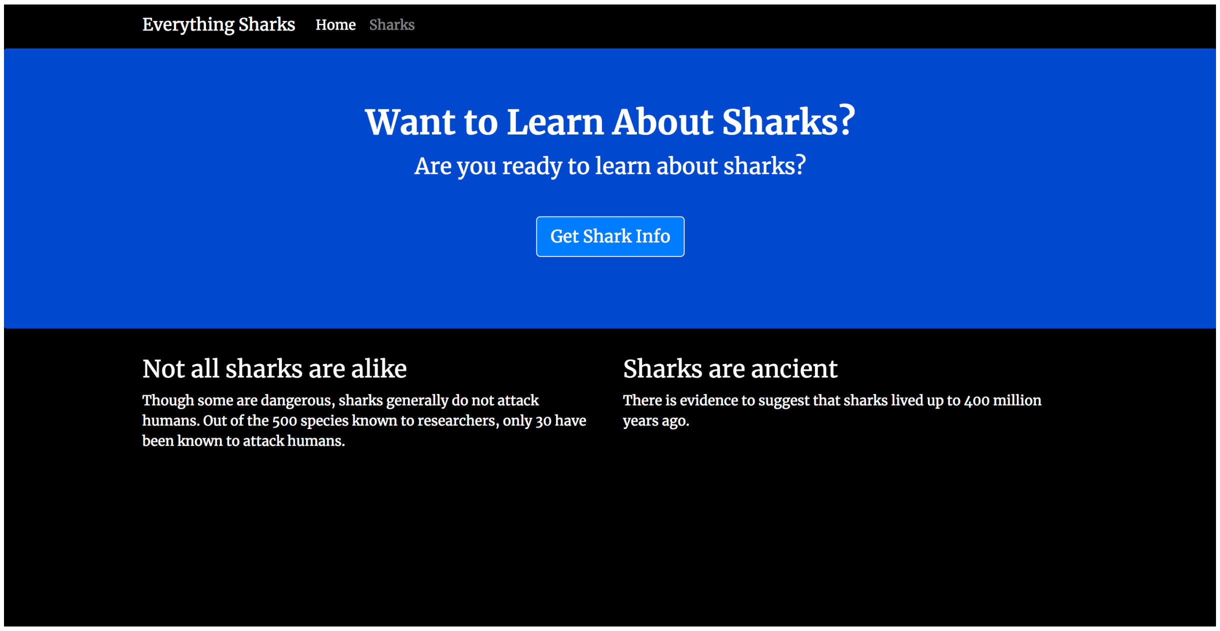
Application Landing Page

Now that your replicated application is working, let's add some test data to ensure that replication is working between members of the replica set.

Step 6 — Testing MongoDB Replication

With our application running and accessible through an external IP address, we can add some test data and ensure that it is being replicated between the members of our MongoDB replica set.

First, make sure you have navigated your browser to the application landing page:



Want to Learn About Sharks?

Are you ready to learn about sharks?

Not all sharks are alike

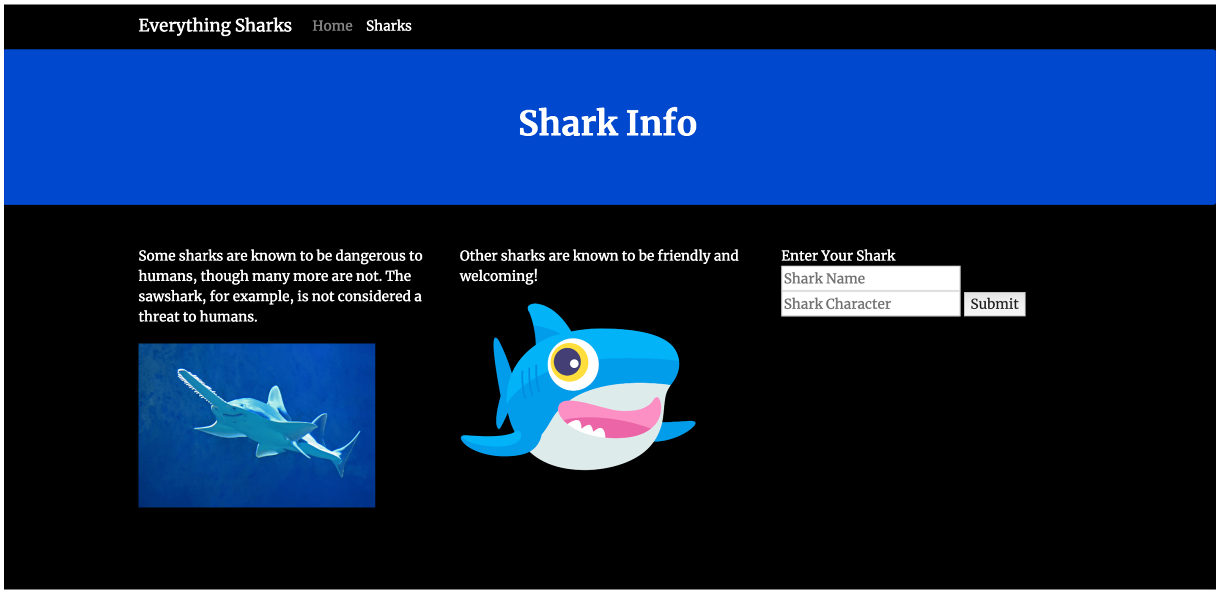
Though some are dangerous, sharks generally do not attack humans. Out of the 500 species known to researchers, only 30 have been known to attack humans.

Sharks are ancient

There is evidence to suggest that sharks lived up to 400 million years ago.

Application Landing Page

Click on the Get Shark Info button. You will see a page with an entry form where you can enter a shark name and a description of that shark's general character:


Everything Sharks Home Sharks

Shark Info

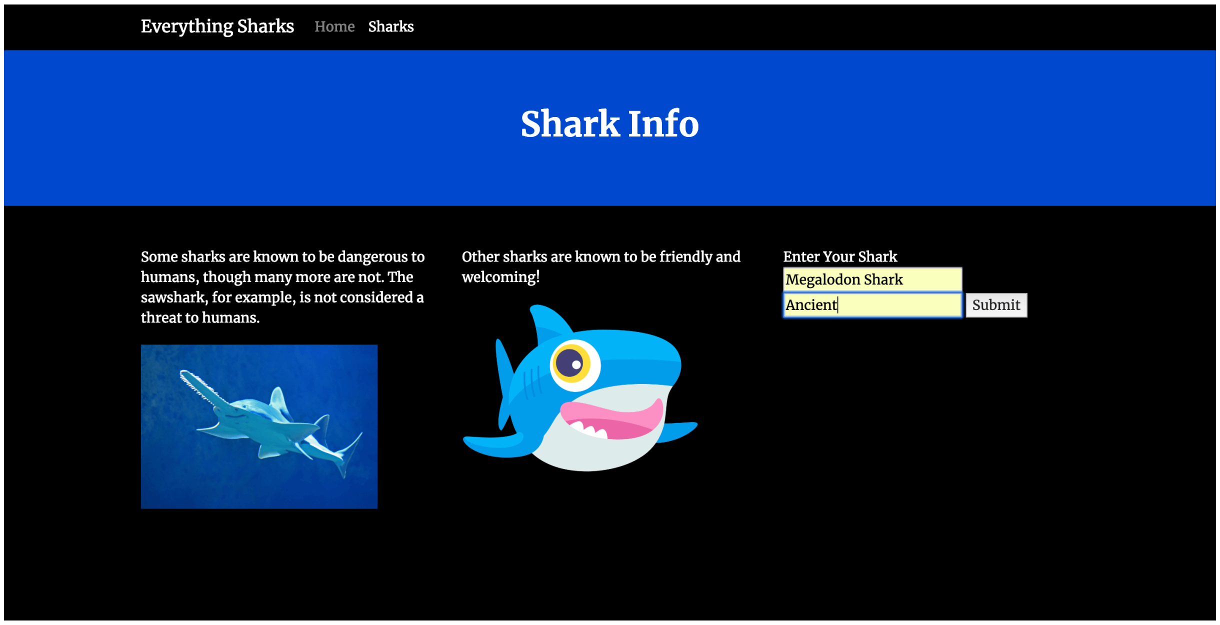
Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

Shark Info Form

In the form, add an initial shark of your choosing. To demonstrate, we will add `Megalodon Shark` to the Shark Name field, and `Ancient` to the Shark Character field:



Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

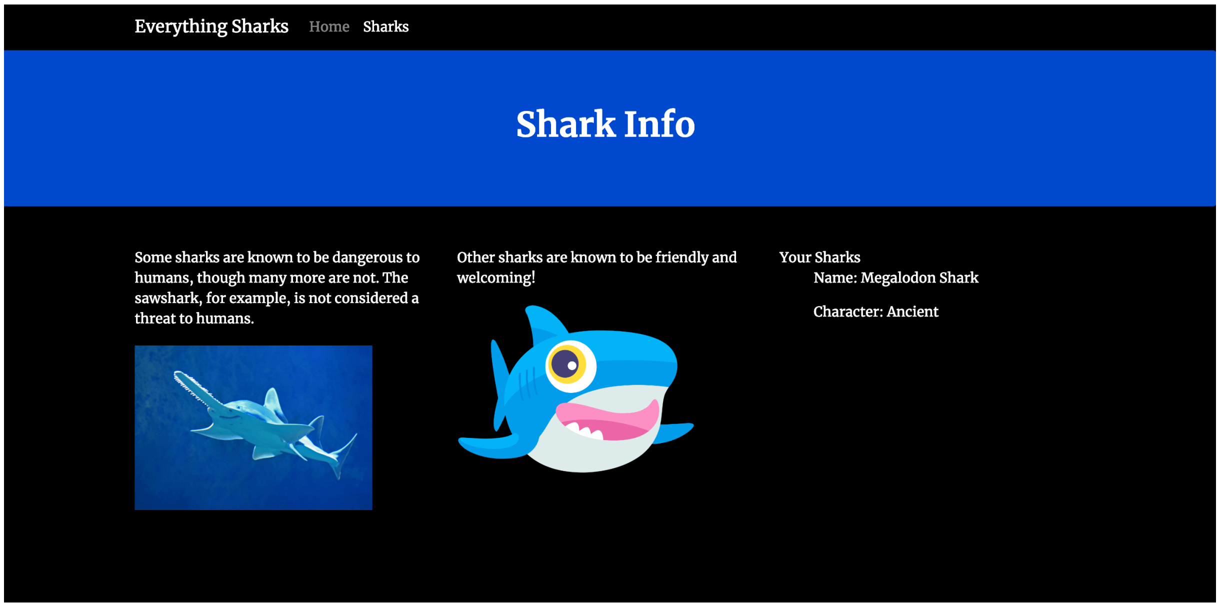
Enter Your Shark

Megalodon Shark

Ancient

Filled Shark Form

Click on the Submit button. You will see a page with this shark information displayed back to you:



Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

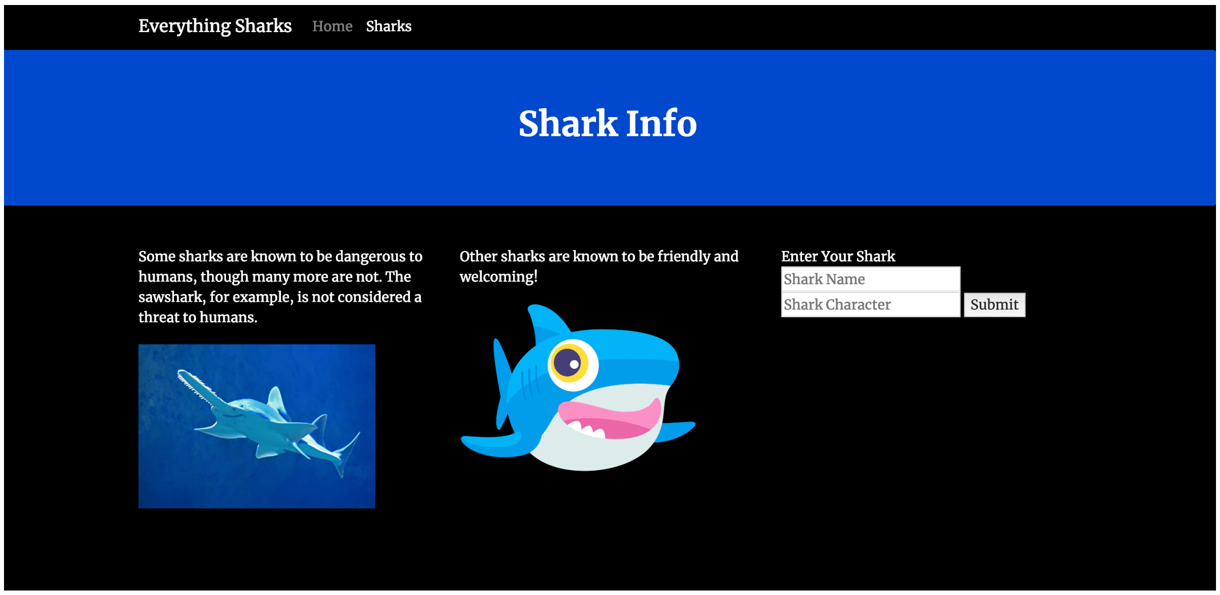
Your Sharks

Name: Megalodon Shark

Character: Ancient

Shark Output

Now head back to the shark information form by clicking on Sharks in the top navigation bar:


Everything Sharks Home Sharks

Shark Info

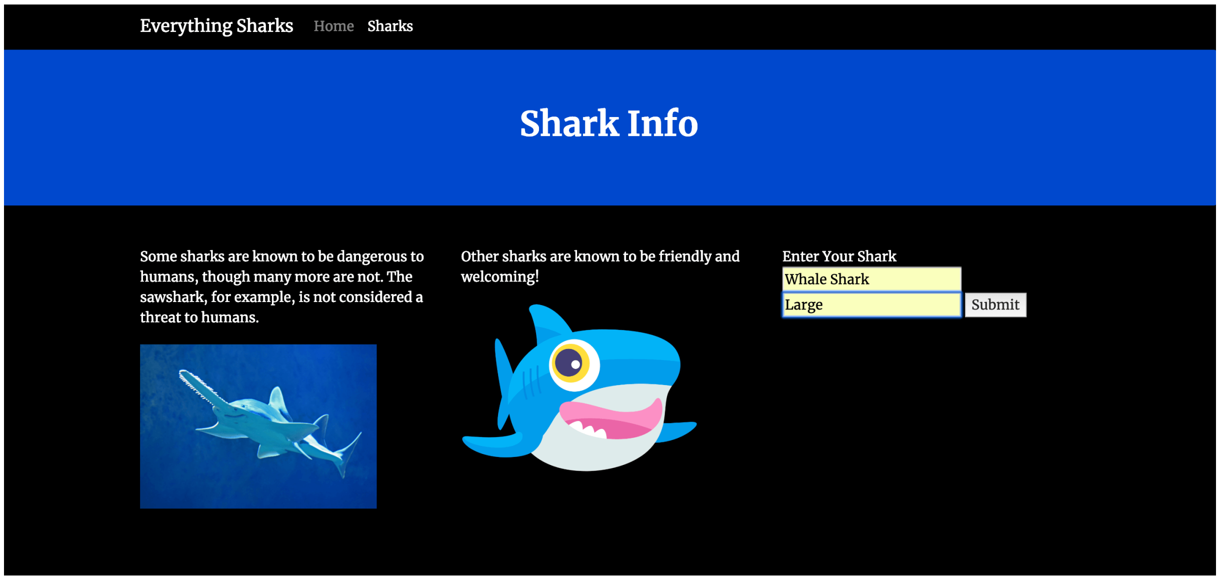
Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Enter Your Shark

Shark Info Form

Enter a new shark of your choosing. We'll go with `Whale Shark` and `Large`:



Everything Sharks Home Sharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

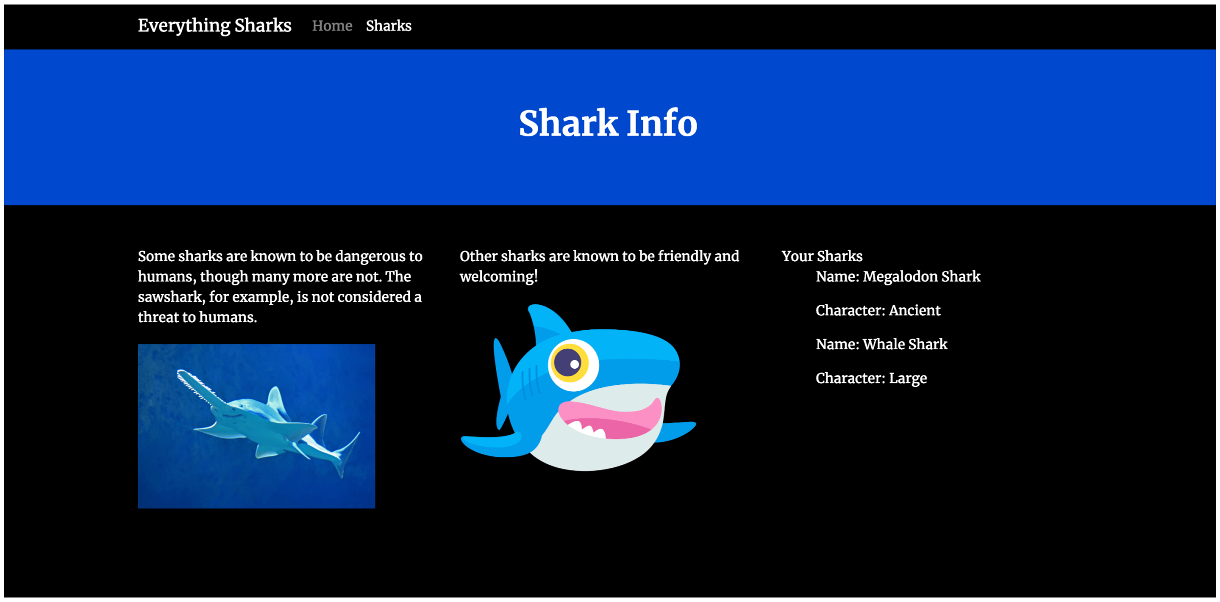
Enter Your Shark

Whale Shark

Large

Enter New Shark

Once you click Submit, you will see that the new shark has been added to the shark collection in your database:



Everything Sharks Home Sharks

Shark Info

Some sharks are known to be dangerous to humans, though many more are not. The sawshark, for example, is not considered a threat to humans.

Other sharks are known to be friendly and welcoming!

Your Sharks

Name: Megalodon Shark

Character: Ancient

Name: Whale Shark

Character: Large

Complete Shark Collection

Let’s check that the data we’ve entered has been replicated between the primary and secondary members of our replica set.

Get a list of your Pods:

```
kubectl get pods
```

Output

```
NAME                                 READY    STATUS
RESTARTS    AGE
mongo-mongodb-replicaset-0           1/1      Running
0           74m
mongo-mongodb-replicaset-1           1/1      Running
0           73m
mongo-mongodb-replicaset-2           1/1      Running
0           72m
nodejs-nodeapp-577df49dcc-b5fq5      1/1      Running
0           5m4s
nodejs-nodeapp-577df49dcc-bkk66      1/1      Running
0           5m4s
nodejs-nodeapp-577df49dcc-lpmt2      1/1      Running
0           5m4s
```

To access the `mongo` shell on your Pods, you can use the `kubectl exec` command and the username you used to create your **`mongo-secret`** in Step 2. Access the `mongo` shell on the first Pod in the StatefulSet with the following command:

```
kubectl exec -it mongo-mongodb-replicaset-0 -- mongo
-u your_database_username -p --
authenticationDatabase admin
```

When prompted, enter the password associated with this username:

Output

```
MongoDB shell version v4.1.9
Enter password:
```

You will be dropped into an administrative shell:

Output

```
MongoDB server version: 4.1.9
Welcome to the MongoDB shell.
...

db:PRIMARY>
```

Though the prompt itself includes this information, you can manually check to see which replica set member is the primary with the [`rs.isMaster()` method](#):

```
rs.isMaster()
```

You will see output like the following, indicating the hostname of the primary:

Output

```
db:PRIMARY> rs.isMaster()
{
        "hosts" : [
                "mongo-mongodb-replicaset-0.mongo-
mongodb-replicaset.default.svc.cluster.local:27017",
                "mongo-mongodb-replicaset-1.mongo-
mongodb-replicaset.default.svc.cluster.local:27017",
                "mongo-mongodb-replicaset-2.mongo-
mongodb-replicaset.default.svc.cluster.local:27017"
```

```
        ],
        ...
        "primary" : "mongo-mongodb-replicaset-
0.mongo-mongodb-
replicaset.default.svc.cluster.local:27017",
        ...
```

Next, switch to your **sharkinfo** database:

```
use sharkinfo
```

Output

```
switched to db sharkinfo
```

List the collections in the database:

```
show collections
```

Output

```
sharks
```

Output the documents in the collection:

```
db.sharks.find()
```

You will see the following output:

Output

```
{ "_id" : ObjectId("5cb7702c9111a5451c6dc8bb"),
"name" : "Megalodon Shark", "character" : "Ancient",
"__v" : 0 }
{ "_id" : ObjectId("5cb77054fcdbf563f3b47365"),
"name" : "Whale Shark", "character" : "Large", "__v"
: 0 }
```

Exit the MongoDB Shell:

```
exit
```

Now that we have checked the data on our primary, let's check that it's being replicated to a secondary. `kubectl exec` into `mongo-mongodb-replicaset-1` with the following command:

```
kubectl exec -it mongo-mongodb-replicaset-1 -- mongo
-u your_database_username -p --
authenticationDatabase admin
```

Once in the administrative shell, we will need to use the `db.setSlaveOk()` method to permit read operations from the secondary instance:

```
db.setSlaveOk(1)
```

Switch to the **sharkinfo** database:

```
use sharkinfo
```

Output

```
switched to db sharkinfo
```

Permit the read operation of the documents in the `sharks` collection:

```
db.setSlaveOk(1)
```

Output the documents in the collection:

```
db.sharks.find()
```

You should now see the same information that you saw when running this method on your primary instance:

Output

```
db:SECONDARY> db.sharks.find()
{ "_id" : ObjectId("5cb7702c9111a5451c6dc8bb"),
"name" : "Megalodon Shark", "character" : "Ancient",
"__v" : 0 }
```

```
{ "_id" : ObjectId("5cb77054fcdbf563f3b47365"),
"name" : "Whale Shark", "character" : "Large", "__v"
: 0 }
```

This output confirms that your application data is being replicated between the members of your replica set.

Conclusion

You have now deployed a replicated, highly-available shark information application on a Kubernetes cluster using Helm charts. This demo application and the workflow outlined in this tutorial can act as a starting point as you build custom charts for your application and take advantage of Helm’s stable repository and other chart repositories.

As you move toward production, consider implementing the following: - Centralized logging and monitoring. Please see the relevant discussion in Modernizing Applications for Kubernetes for a general overview. You can also look at How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes to learn how to set up a logging stack with Elasticsearch, Fluentd, and Kibana. Also check out An Introduction to Service Meshes for information about how service meshes like Istio implement this functionality. - Ingress Resources to route traffic to your cluster. This is a good alternative to a LoadBalancer in cases where you are running multiple Services, which each require their own LoadBalancer, or where you would like to implement application-level routing strategies (A/B & canary tests, for example). For more information, check out How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes and the related discussion of routing in the service mesh context in An Introduction to Service Meshes. - Backup strategies for your Kubernetes objects. For guidance on implementing backups with Velero (formerly Heptio Ark) with DigitalOcean’s Kubernetes product, please see How To Back Up and Restore a Kubernetes Cluster on DigitalOcean Using Heptio Ark.

To learn more about Helm, see An Introduction to Helm, the Package Manager for Kubernetes, How To Install Software on Kubernetes Clusters with the Helm Package Manager, and the Helm documentation.

How To Set Up a Private Docker Registry on Top of DigitalOcean Spaces and Use It with DigitalOcean Kubernetes

Written by Savic

In this tutorial, you'll deploy a private Docker registry to your Kubernetes cluster using Helm. A self-hosted Docker Registry lets you privately store, distribute, and manage your Docker images. While this tutorial focuses on using DigitalOcean's Kubernetes and Spaces products, the principles of running your own Registry in a cluster apply to any Kubernetes stack.

At the end of this tutorial, you'll have a secure, private Docker registry that uses DigitalOcean Spaces (or another S3-compatible object storage system) to store your images. Your Kubernetes cluster will be configured to use the self-hosted registry so that your containerized applications remain private and secure.

The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.

A Docker registry is a storage and content delivery system for named Docker images, which are the industry standard for containerized applications. A private Docker registry allows you to securely share your images within your team or organization with more flexibility and control when compared to public ones. By hosting your private Docker registry directly in your Kubernetes cluster, you can achieve higher speeds, lower latency, and better availability, all while having control over the registry.

The underlying registry storage is delegated to external drivers. The default storage system is the local filesystem, but you can swap this for a cloud-based storage driver. DigitalOcean Spaces is an S3-compatible object storage designed for developer teams and businesses that want a scalable, simple, and affordable way to store and serve vast amounts of data, and is very suitable for storing Docker images. It has a built-in CDN network, which can greatly reduce latency when frequently accessing images.

In this tutorial, you'll deploy your private Docker registry to your DigitalOcean Kubernetes cluster using Helm, backed up by DigitalOcean Spaces for storing data. You'll create API keys for your designated Space, install the Docker registry to your cluster with custom configuration, configure Kubernetes to properly authenticate with it, and test it by running a sample deployment on the cluster. At the end of this tutorial, you'll have a secure, private Docker registry installed on your DigitalOcean Kubernetes cluster.

Prerequisites

Before you begin this tutorial, you'll need:

- Docker installed on the machine that you'll access your cluster from. For Ubuntu 18.04 visit How To Install and Use Docker on Ubuntu 18.04. You only need to complete the first step. Otherwise visit Docker's website for other distributions.
- A DigitalOcean Kubernetes cluster with your connection configuration configured as the `kubectl` default. Instructions on how to configure `kubectl` are shown under the Connect to your

Cluster step shown when you create your cluster. To learn how to create a Kubernetes cluster on DigitalOcean, see Kubernetes Quickstart.

- A DigitalOcean Space with API keys (access and secret). To learn how to create a DigitalOcean Space and API keys, see How To Create a DigitalOcean Space and API Key.
- The Helm package manager installed on your local machine, and Tiller installed on your cluster. Complete steps 1 and 2 of the How To Install Software on Kubernetes Clusters with the Helm Package Manager. You only need to complete the first two steps.
- The Nginx Ingress Controller and Cert-Manager installed on the cluster. For a guide on how to do this, see How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes.
- A domain name with two DNS A records pointed to the DigitalOcean Load Balancer used by the Ingress. If you are using DigitalOcean to manage your domain's DNS records, consult How to Manage DNS Records to create A records. In this tutorial, we'll refer to the A records as **`registry.example.com`** and **`k8s-test.example.com`**.

Step 1 — Configuring and Installing the Docker Registry

In this step, you will create a configuration file for the registry deployment and install the Docker registry to your cluster with the given config using the Helm package manager.

During the course of this tutorial, you will use a configuration file called `chart_values.yaml` to override some of the default settings for the Docker registry Helm chart. Helm calls its packages, charts; these are sets of files that outline a related selection of Kubernetes resources. You'll edit the settings to specify DigitalOcean Spaces as the underlying storage system and enable HTTPS access by wiring up Let's Encrypt TLS certificates.

As part of the prerequisite, you would have created the `echo1` and `echo2` services and an `echo_ingress` ingress for testing purposes; you will not need these in this tutorial, so you can now delete them.

Start off by deleting the ingress by running the following command:

```
kubectl delete -f echo_ingress.yaml
```

Then, delete the two test services:

```
kubectl delete -f echo1.yaml && kubectl delete -f
echo2.yaml
```

The kubectl `delete` command accepts the file to delete when passed the `-f` parameter.

Create a folder that will serve as your workspace:

```
mkdir ~/k8s-registry
```

Navigate to it by running:

```
cd ~/k8s-registry
```

Now, using your text editor, create your `chart_values.yaml` file:

```
nano chart_values.yaml
```

Add the following lines, ensuring you replace the highlighted lines with your details:

chart_values.yaml

```
ingress:
  enabled: true
  hosts:
    - registry.example.com
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
    nginx.ingress.kubernetes.io/proxy-body-size:
"30720m"
  tls:
    - secretName: letsencrypt-prod
      hosts:
        - registry.example.com

storage: s3

secrets:
  htpasswd: ""
  s3:
    accessKey: "your_space_access_key"
    secretKey: "your_space_secret_key"

s3:
  region: your_space_region
  regionEndpoint:
your_space_region.digitaloceanspaces.com
```

```
  secure: true
  bucket: your_space_name
```

The first block, `ingress`, configures the Kubernetes Ingress that will be created as a part of the Helm chart deployment. The Ingress object makes outside HTTP/HTTPS routes point to internal services in the cluster, thus allowing communication from the outside. The overridden values are:

- `enabled`: set to `true` to enable the Ingress.
- `hosts`: a list of hosts from which the Ingress will accept traffic.
- `annotations`: a list of metadata that provides further direction to other parts of Kubernetes on how to treat the Ingress. You set the Ingress Controller to `nginx`, the Let's Encrypt cluster issuer to the production variant (`letsencrypt-prod`), and tell the `nginx` controller to accept files with a max size of 30 GB, which is a sensible limit for even the largest Docker images.
- `tls`: this subcategory configures Let's Encrypt HTTPS. You populate the `hosts` list that defines from which secure hosts this Ingress will accept HTTPS traffic with our example domain name.

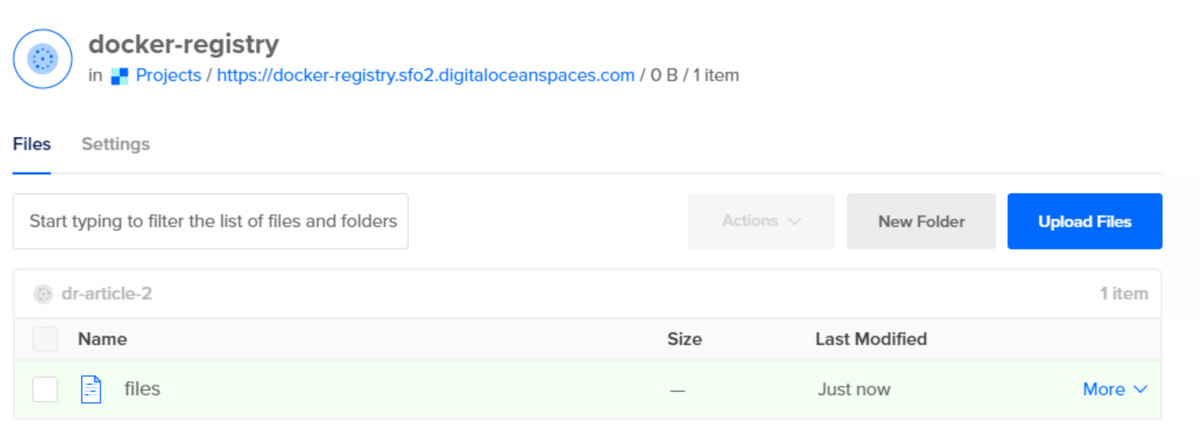
Then, you set the file system storage to `s3` — the other available option would be `filesystem`. Here `s3` indicates using a remote storage system compatible with the industry-standard Amazon S3 API, which DigitalOcean Spaces fulfills.

In the next block, `secrets`, you configure keys for accessing your DigitalOcean Space under the `s3` subcategory. Finally, in the `s3` block, you configure the parameters specifying your Space.

Save and close your file.

Now, if you haven't already done so, set up your A records to point to the Load Balancer you created as part of the Nginx Ingress Controller installation in the prerequisite tutorial. To see how to set your DNS on DigitalOcean, see [How to Manage DNS Records]().

Next, ensure your Space isn't empty. The Docker registry won't run at all if you don't have any files in your Space. To get around this, upload a file. Navigate to the Spaces tab, find your Space, click the Upload File button, and upload any file you'd like. You could upload the configuration file you just created.


docker-registry

in Projects / https://docker-registry.sfo2.digitaloceanspaces.com / 0 B / 1 item

dr-article-2 1 item

Name	Size	Last Modified	
files	—	Just now	More

Empty file uploaded to empty Space

Before installing anything via Helm, you need to refresh its cache. This will update the latest information about your chart repository. To do this run the following command:

```
helm repo update
```

Now, you'll deploy the Docker registry chart with this custom configuration via Helm by running:

```
helm install stable/docker-registry -f
chart_values.yaml --name docker-registry
```

You'll see the following output:

Output

```
NAME:   docker-registry
...
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                    DATA  AGE
docker-registry-config  1     1s

==> v1/Pod(related)
NAME                                  READY  STATUS
RESTARTS  AGE
docker-registry-54df68fd64-l26fb  0/1
ContainerCreating  0         1s
```

```
==> v1/Secret
NAME                     TYPE    DATA  AGE
docker-registry-secret  Opaque  3     1s

==> v1/Service
NAME              TYPE       CLUSTER-IP
EXTERNAL-IP  PORT(S)   AGE
docker-registry  ClusterIP  10.245.131.143  <none>
5000/TCP  1s

==> v1beta1/Deployment
NAME              READY  UP-TO-DATE  AVAILABLE  AGE
docker-registry  0/1    1           0          1s

==> v1beta1/Ingress
NAME              HOSTS                   ADDRESS
PORTS  AGE
docker-registry  registry.example.com  80, 443  1s


NOTES:
1. Get the application URL by running these
commands:
  https://registry.example.com/
```

Helm lists all the resources it created as a result of the Docker registry chart deployment. The registry is now accessible from the domain name you specified earlier.

You’ve configured and deployed a Docker registry on your Kubernetes cluster. Next, you will test the availability of the newly deployed Docker registry.

Step 2 — Testing Pushing and Pulling

In this step, you’ll test your newly deployed Docker registry by pushing and pulling images to and from it. Currently, the registry is empty. To have something to push, you need to have an image available on the machine you’re working from. Let’s use the `mysql` Docker image.

Start off by pulling `mysql` from the Docker Hub:

```
sudo docker pull mysql
```

Your output will look like this:

Output

```
Using default tag: latest
latest: Pulling from library/mysql
27833a3ba0a5: Pull complete
...
e906385f419d: Pull complete
Digest:
sha256:a7cf659a764732a27963429a87eccc8457e6d4af0ee
9d5140a3b56e74986eed7
Status: Downloaded newer image for mysql:latest
```

You now have the image available locally. To inform Docker where to push it, you’ll need to tag it with the host name, like so:

```
sudo docker tag mysql registry.example.com/mysql
```

Then, push the image to the new registry:

```
sudo docker push registry.example.com/mysql
```

This command will run successfully and indicate that your new registry is properly configured and accepting traffic — including pushing new images. If you see an error, double check your steps against steps 1 and 2.

To test pulling from the registry cleanly, first delete the local `mysql` images with the following command:

```
sudo docker rmi registry.example.com/mysql && sudo
docker rmi mysql
```

Then, pull it from the registry:

```
sudo docker pull registry.example.com/mysql
```

This command will take a few seconds to complete. If it runs successfully, that means your registry is working correctly. If it shows an error, double check what you have entered against the previous commands.

You can list Docker images available locally by running the following command:

```
sudo docker images
```

You'll see output listing the images available on your local machine, along with their ID and date of creation.

Your Docker registry is configured. You've pushed an image to it and verified you can pull it down. Now let's add authentication so only certain people can access the code.

Step 3 — Adding Account Authentication and Configuring Kubernetes Access

In this step, you'll set up username and password authentication for the registry using the `htpasswd` utility.

The `htpasswd` utility comes from the Apache webserver, which you can use for creating files that store usernames and passwords for basic authentication of HTTP users. The format of `htpasswd` files is `username:hashed_password` (one per line), which is portable enough to allow other programs to use it as well.

To make `htpasswd` available on the system, you'll need to install it by running:

```
sudo apt install apache2-utils -y
```

Note: If you're running this tutorial from a Mac, you'll need to use the following command to make `htpasswd` available on your machine:

```
docker run --rm -v ${PWD}:/app -it httpd htpasswd
-b -c /app/htpasswd_file sammy password
```

Create it by executing the following command:

```
touch htpasswd_file
```

Add a username and password combination to **`htpasswd_file`**:

```
htpasswd -B htpasswd_file username
```

Docker requires the password to be hashed using the bcrypt algorithm, which is why we pass the `-B` parameter. The bcrypt algorithm is a password hashing function based on Blowfish block cipher, with a work factor parameter, which specifies how expensive the hash function will be.

Remember to replace **`username`** with your desired username. When run, `htpasswd` will ask you for the accompanying password and add the combination to `htpasswd_file`. You can repeat this command for as many users as you wish to add.

Now, show the contents of `htpasswd_file` by running the following command:

```
cat htpasswd_file
```

Select and copy the contents shown.

To add authentication to your Docker registry, you'll need to edit `chart_values.yaml` and add the contents of **`htpasswd_file`** in the `htpasswd` variable.

Open `chart_values.yaml` for editing:

```
nano chart_values.yaml
```

Find the line that looks like this:

chart_values.yaml

```
  htpasswd: ""
```

Edit it to match the following, replacing **`htpasswd\_file\_contents`** with the contents you copied from the **`htpasswd_file`**:

chart_values.yaml

```
  htpasswd: |-
    htpasswd_file_contents
```

Be careful with the indentation, each line of the file contents must have four spaces before it.

Once you've added your contents, save and close the file.

To propagate the changes to your cluster, run the following command:

```
helm upgrade docker-registry stable/docker-
registry -f chart_values.yaml
```

The output will be similar to that shown when you first deployed your Docker registry:

Output

```
Release "docker-registry" has been upgraded. Happy
Helming!
LAST DEPLOYED: ...
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                    DATA  AGE
docker-registry-config  1     3m8s

==> v1/Pod(related)
NAME                              READY  STATUS
RESTARTS  AGE
docker-registry-6c5bb7ffbf-ltnjv  1/1    Running
0         3m7s

==> v1/Secret
NAME                    TYPE    DATA  AGE
docker-registry-secret  Opaque  4     3m8s

==> v1/Service
NAME             TYPE       CLUSTER-IP
```

```
EXTERNAL-IP  PORT(S)   AGE
docker-registry  ClusterIP  10.245.128.245  <none>
5000/TCP  3m8s

==> v1beta1/Deployment
NAME              READY  UP-TO-DATE  AVAILABLE  AGE
docker-registry  1/1    1           1
3m8s

==> v1beta1/Ingress
NAME              HOSTS                ADDRESS
PORTS    AGE
docker-registry  registry.example.com
159.89.215.50  80, 443  3m8s


NOTES:
1. Get the application URL by running these
commands:
  https://registry.example.com/
```

This command calls Helm and instructs it to upgrade an existing release, in your case `docker-registry`, with its chart defined in `stable/docker-registry` in the chart repository, after applying the `chart_values.yaml` file.

Now, you'll try pulling an image from the registry again:

```
sudo docker pull registry.example.com/mysql
```

The output will look like the following:

Output
```
Using default tag: latest
Error response from daemon: Get
https://registry.example.com/v2/mysql/manifests/la
test: no basic auth credentials
```

It correctly failed because you provided no credentials. This means that your Docker registry authorizes requests correctly.

To log in to the registry, run the following command:

```
sudo docker login registry.example.com
```

Remember to replace **registry.example.com** with your domain address. It will prompt you for a username and password. If it shows an error, double check what your `htpasswd_file` contains. You must define the username and password combination in the `htpasswd_file`, which you created earlier in this step.

To test the login, you can try to pull again by running the following command:

```
sudo docker pull registry.example.com/mysql
```

The output will look similar to the following:

Output
```
Using default tag: latest
latest: Pulling from mysql
Digest:
sha256:f2dc118ca6fa4c88cde5889808c486dfe94bccecd01
ca626b002a010bb66bcbe
```

```
Status: Image is up to date for
registry.example.com/mysql:latest
```

You’ve now configured Docker and can log in securely. To configure Kubernetes to log in to your registry, run the following command:

```
sudo kubectl create secret generic regcred --from-
file=.dockerconfigjson=/home/sammy/.docker/config.
json --type=kubernetes.io/dockerconfigjson
```

You will see the following output:

Output

```
secret/regcred created
```

This command creates a secret in your cluster with the name `regcred`, takes the contents of the JSON file where Docker stores the credentials, and parses it as `dockerconfigjson`, which defines a registry credential in Kubernetes.

You’ve used `htpasswd` to create a login config file, configured the registry to authenticate requests, and created a Kubernetes secret containing the login credentials. Next, you will test the integration between your Kubernetes cluster and registry.

Step 4 — Testing Kubernetes Integration by Running a Sample Deployment

In this step, you’ll run a sample deployment with an image stored in the in-cluster registry to test the connection between your Kubernetes cluster and registry.

In the last step, you created a secret, called `regcred`, containing login credentials for your private registry. It may contain login credentials for multiple registries, in which case you'll have to update the Secret accordingly.

You can specify which secret Kubernetes should use when pulling containers in the pod definition by specifying `imagePullSecrets`. This step is necessary when the Docker registry requires authentication.

You'll now deploy a sample Hello World image from your private Docker registry to your cluster. First, in order to push it, you'll pull it to your machine by running the following command:

```
sudo docker pull paulbouwer/hello-kubernetes:1.5
```

Then, tag it by running:

```
sudo docker tag paulbouwer/hello-kubernetes:1.5
registry.example.com/paulbouwer/hello-
kubernetes:1.5
```

Finally, push it to your registry:

```
sudo docker push
registry.example.com/paulbouwer/hello-
kubernetes:1.5
```

Delete it from your machine as you no longer need it locally:

```
sudo docker rmi
registry.example.com/paulbouwer/hello-
kubernetes:1.5
```

Now, you'll deploy the sample Hello World application. First, create a new file, `hello-world.yaml`, using your text editor:

```
nano hello-world.yaml
```

Next, you'll define a Service and an Ingress to make the app accessible to outside of the cluster. Add the following lines, replacing the highlighted lines with your domains:

hello-world.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-kubernetes-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: k8s-test.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: hello-kubernetes
          servicePort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
spec:
  type: NodePort
  ports:
```

```
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
      - name: hello-kubernetes
        image:
registry.example.com/paulbouwer/hello-
kubernetes:1.5
        ports:
        - containerPort: 8080
      imagePullSecrets:
      - name: regcred
```

First, you define the Ingress for the Hello World deployment, which you will route through the Load Balancer that the Nginx Ingress Controller owns. Then, you define a service that can access the pods created in the deployment. In the actual deployment spec, you specify the `image` as the one located in your registry and set `imagePullSecrets` to `regcred`, which you created in the previous step.

Save and close the file. To deploy this to your cluster, run the following command:

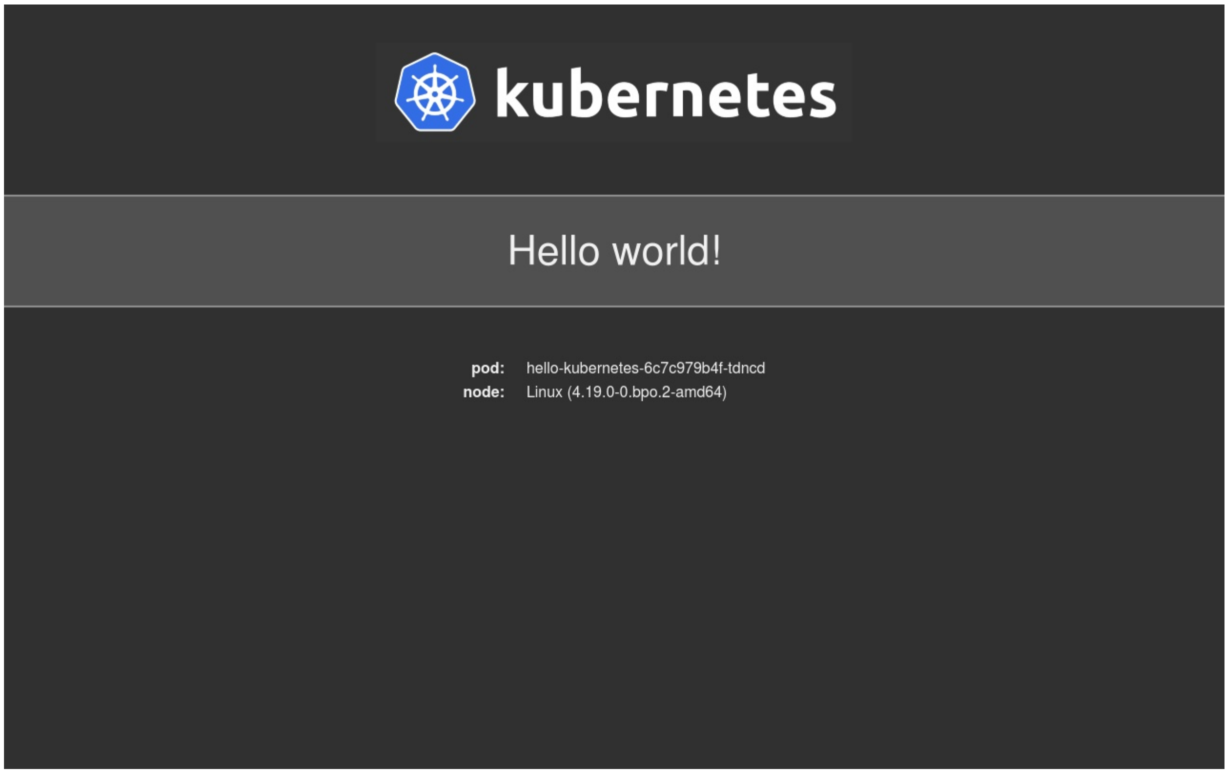
```
kubectl apply -f hello-world.yaml
```

You’ll see the following output:

Output

```
ingress.extensions/hello-kubernetes-ingress
created
service/hello-kubernetes created
deployment.apps/hello-kubernetes created
```

You can now navigate to your test domain — the second A record, **`k8s-test.example.com`** in this tutorial. You will see the Kubernetes Hello world! page.



```
pod:   hello-kubernetes-6c7c979b4f-tdncd
node:  Linux (4.19.0-0.bpo.2-amd64)
```

Hello World page

The Hello World page lists some environment information, like the Linux kernel version and the internal ID of the pod the request was served from. You can also access your Space via the web interface to see the images you've worked with in this tutorial.

If you want to delete this Hello World deployment after testing, run the following command:

```
kubectl delete -f hello-world.yaml
```

You've created a sample Hello World deployment to test if Kubernetes is properly pulling images from your private registry.

Conclusion

You have now successfully deployed your own private Docker registry on your DigitalOcean Kubernetes cluster, using DigitalOcean Spaces as the storage layer underneath. There is no limit to how many images you can store, Spaces can extend infinitely, while at the same time providing the same security and robustness. In production, though, you should always strive to optimize your Docker images as much as possible, take a look at the How To Optimize Docker Images for Production tutorial.

How To Deploy a PHP Application with Kubernetes on Ubuntu 18.04

Written by Amitabh Dhiwal

In this tutorial, you will deploy a PHP application on a Kubernetes cluster with Nginx and PHP-FPM running in separate Pods. You will also learn how to keep your configuration files and application code outside the container image using DigitalOcean’s Block Storage system. This approach will allow you to reuse the Nginx image for any application that needs a web/proxy server by passing a configuration volume, rather than rebuilding the image.

The author selected Electronic Frontier Foundation to receive a donation as part of the Write for DOnations program.

Kubernetes is an open source container orchestration system. It allows you to create, update, and scale containers without worrying about downtime.

To run a PHP application, Nginx acts as a proxy to PHP-FPM. Containerizing this setup in a single container can be a cumbersome process, but Kubernetes will help manage both services in separate containers. Using Kubernetes will allow you to keep your containers reusable and swappable, and you will not have to rebuild your container image every time there’s a new version of Nginx or PHP.

In this tutorial, you will deploy a PHP 7 application on a Kubernetes cluster with Nginx and PHP-FPM running in separate containers. You will also learn how to keep your configuration files and application code outside the container image using DigitalOcean's Block Storage system. This approach will allow you to reuse the Nginx image for any application that needs a web/proxy server by passing a configuration volume, rather than rebuilding the image.

Prerequisites

- A basic understanding of Kubernetes objects. Check out our Introduction to Kubernetes article for more information.
- A Kubernetes cluster running on Ubuntu 18.04. You can set this up by following the How To Create a Kubernetes 1.14 Cluster Using Kubeadm on Ubuntu 18.04 tutorial.
- A DigitalOcean account and an API access token with read and write permissions to create our storage volume. If you don't have your API access token, you can create it from here.
- Your application code hosted on a publicly accessible URL, such as Github.

Step 1 — Creating the PHP-FPM and Nginx Services

In this step, you will create the PHP-FPM and Nginx services. A service allows access to a set of pods from within the cluster. Services within a cluster can communicate directly through their names, without the need for IP addresses. The PHP-FPM service will allow access to the PHP-FPM pods, while the Nginx service will allow access to the Nginx pods.

Since Nginx pods will proxy the PHP-FPM pods, you will need to tell the service how to find them. Instead of using IP addresses, you will take advantage of Kubernetes' automatic service discovery to use human-readable names to route requests to the appropriate service.

To create the service, you will create an object definition file. Every Kubernetes object definition is a YAML file that contains at least the following items:

- `apiVersion`: The version of the Kubernetes API that the definition belongs to.
- `kind`: The Kubernetes object this file represents. For example, a `pod` or `service`.
- `metadata`: This contains the `name` of the object along with any `labels` that you may wish to apply to it.
- `spec`: This contains a specific configuration depending on the kind of object you are creating, such as the container image or the ports on which the container will be accessible from.

First you will create a directory to hold your Kubernetes object definitions.

SSH to your master node and create the `definitions` directory that will hold your Kubernetes object definitions.

```
mkdir definitions
```

Navigate to the newly created `definitions` directory:

```
cd definitions
```

Make your PHP-FPM service by creating a `php_service.yaml` file:

```
nano php_service.yaml
```

Set `kind` as `Service` to specify that this object is a service:

php_service.yaml

```
apiVersion: v1
kind: Service
```

Name the service `php` since it will provide access to PHP-FPM:

php_service.yaml

```
...
metadata:
  name: php
```

You will logically group different objects with labels. In this tutorial, you will use labels to group the objects into “tiers”, such as frontend or backend. The PHP pods will run behind this service, so you will label it as `tier: backend`.

php_service.yaml

```
...
  labels:
    tier: backend
```

A service determines which pods to access by using `selector` labels. A pod that matches these labels will be serviced, independent of whether the pod was created before or after the service. You will add labels for your pods later in the tutorial.

Use the `tier: backend` label to assign the pod into the back-end tier. You will also add the `app: php` label to specify that this pod runs PHP. Add these two labels after the `metadata` section.

php_service.yaml

```
...
spec:
  selector:
    app: php
    tier: backend
```

Next, specify the port used to access this service. You will use port `9000` in this tutorial. Add it to the `php_service.yaml` file under `spec`:

php_service.yaml

```
...
  ports:
    - protocol: TCP
      port: 9000
```

Your completed `php_service.yaml` file will look like this:

php_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: php
  labels:
    tier: backend
spec:
  selector:
    app: php
    tier: backend
  ports:
```

```
  - protocol: TCP
    port: 9000
```

Hit `CTRL + O` to save the file, and then `CTRL + X` to exit `nano`.

Now that you've created the object definition for your service, to run the service you will use the `kubectl apply` command along with the `-f` argument and specify your `php_service.yaml` file.

Create your service:

```
kubectl apply -f php_service.yaml
```

This output confirms the service creation:

Output

```
service/php created
```

Verify that your service is running:

```
kubectl get svc
```

You will see your PHP-FPM service running:

Output

```
NAME         TYPE        CLUSTER-IP      EXTERNAL-
IP    PORT(S)     AGE
kubernetes   ClusterIP   10.96.0.1       <none>
443/TCP    10m
php          ClusterIP   10.100.59.238   <none>
9000/TCP   5m
```

There are various service types that Kubernetes supports. Your `php` service uses the default service type, `ClusterIP`. This service type assigns an internal IP and makes the service reachable only from within the cluster.

Now that the PHP-FPM service is ready, you will create the Nginx service. Create and open a new file called `nginx_service.yaml` with the editor:

```
nano nginx_service.yaml
```

This service will target Nginx pods, so you will name it `nginx`. You will also add a `tier: backend` label as it belongs in the back-end tier:

nginx_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    tier: backend
```

Similar to the `php` service, target the pods with the selector labels `app: nginx` and `tier: backend`. Make this service accessible on port `80`, the default HTTP port.

nginx_service.yaml

```
...
spec:
  selector:
    app: nginx
    tier: backend
  ports:
  - protocol: TCP
    port: 80
```

The Nginx service will be publicly accessible to the internet from your Droplet's public IP address. **`your_public_ip`** can be found from your [DigitalOcean Control Panel](). Under `spec.externalIPs`, add:

nginx_service.yaml

```
...
spec:
  externalIPs:
  - your_public_ip
```

Your `nginx_service.yaml` file will look like this:

nginx_service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    tier: backend
spec:
  selector:
    app: nginx
    tier: backend
  ports:
  - protocol: TCP
    port: 80
  externalIPs:
  - your_public_ip
```

Save and close the file. Create the Nginx service:

```
kubectl apply -f nginx_service.yaml
```

You will see the following output when the service is running:

Output

```
service/nginx created
```

You can view all running services by executing:

```
kubectl get svc
```

You will see both the PHP-FPM and Nginx services listed in the output:

Output

```
NAME             TYPE         CLUSTER-IP       EXTERNAL-
IP    PORT(S)     AGE
kubernetes    ClusterIP    10.96.0.1        <none>
443/TCP     13m
nginx         ClusterIP    10.102.160.47
your_public_ip 80/TCP      50s
php           ClusterIP    10.100.59.238    <none>
9000/TCP    8m
```

Please note, if you want to delete a service you can run:

```
kubectl delete svc/service_name
```

Now that you've created your PHP-FPM and Nginx services, you will need to specify where to store your application code and configuration files.

Step 2 — Installing the DigitalOcean Storage Plug-In

Kubernetes provides different storage plug-ins that can create the storage space for your environment. In this step, you will install the DigitalOcean storage plug-in to create block storage on DigitalOcean. Once the installation is complete, it will add a storage class named `do-block-storage` that you will use to create your block storage.

You will first configure a Kubernetes Secret object to store your DigitalOcean API token. Secret objects are used to share sensitive information, like SSH keys and passwords, with other Kubernetes objects within the same namespace. Namespaces provide a way to logically separate your Kubernetes objects.

Open a file named `secret.yaml` with the editor:

```
nano secret.yaml
```

You will name your Secret object `digitalocean` and add it to the `kube-system` namespace. The `kube-system` namespace is the default namespace for Kubernetes' internal services and is also used by the DigitalOcean storage plug-in to launch various components.

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: digitalocean
  namespace: kube-system
```

Instead of a `spec` key, a Secret uses a `data` or `stringData` key to hold the required information. The `data` parameter holds base64 encoded data that is automatically decoded when retrieved. The `stringData` parameter holds non-encoded data that is automatically encoded during creation or updates, and does not output the data when retrieving Secrets. You will use `stringData` in this tutorial for convenience.

Add the `access-token` as `stringData`:

secret.yaml

```
...
stringData:
  access-token: your-api-token
```

Save and exit the file.

Your `secret.yaml` file will look like this:

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: digitalocean
  namespace: kube-system
stringData:
  access-token: your-api-token
```

Create the secret:

```
kubectl apply -f secret.yaml
```

You will see this output upon Secret creation:

Output

```
secret/digitalocean created
```

You can view the secret with the following command:

```
kubectl -n kube-system get secret digitalocean
```

The output will look similar to this:

Output

```
NAME             TYPE      DATA      AGE
digitalocean     Opaque    1         41s
```

The `Opaque` type means that this Secret is read-only, which is standard for `stringData` Secrets. You can read more about it on the Secret design spec. The `DATA` field shows the number of items stored in this Secret. In this case, it shows `1` because you have a single key stored.

Now that your Secret is in place, install the DigitalOcean block storage plug-in:

```
kubectl apply -f
https://raw.githubusercontent.com/digitalocean/csi
-
digitalocean/master/deploy/kubernetes/releases/csi
-digitalocean-v1.1.0.yaml
```

You will see output similar to the following:

Output

```
csidriver.storage.k8s.io/dobs.csi.digitalocean.com
created
customresourcedefinition.apiextensions.k8s.io/volu
mesnapshotclasses.snapshot.storage.k8s.io created
customresourcedefinition.apiextensions.k8s.io/volu
mesnapshotcontents.snapshot.storage.k8s.io created
customresourcedefinition.apiextensions.k8s.io/volu
mesnapshots.snapshot.storage.k8s.io created
storageclass.storage.k8s.io/do-block-storage
created
```

```
statefulset.apps/csi-do-controller created
serviceaccount/csi-do-controller-sa created
clusterrole.rbac.authorization.k8s.io/csi-do-
provisioner-role created
clusterrolebinding.rbac.authorization.k8s.io/csi-
do-provisioner-binding created
clusterrole.rbac.authorization.k8s.io/csi-do-
attacher-role created
clusterrolebinding.rbac.authorization.k8s.io/csi-
do-attacher-binding created
clusterrole.rbac.authorization.k8s.io/csi-do-
snapshotter-role created
clusterrolebinding.rbac.authorization.k8s.io/csi-
do-snapshotter-binding created
daemonset.apps/csi-do-node created
serviceaccount/csi-do-node-sa created
clusterrole.rbac.authorization.k8s.io/csi-do-node-
driver-registrar-role created
clusterrolebinding.rbac.authorization.k8s.io/csi-
do-node-driver-registrar-binding created
error: unable to recognize
"https://raw.githubusercontent.com/digitalocean/cs
i-
digitalocean/master/deploy/kubernetes/releases/csi
-digitalocean-v1.1.0.yaml": no matches for kind
"VolumeSnapshotClass" in version
"snapshot.storage.k8s.io/v1alpha1"
```

For this tutorial, it is safe to ignore the errors.

Now that you have installed the DigitalOcean storage plug-in, you can create block storage to hold your application code and configuration files.

Step 3 — Creating the Persistent Volume

With your Secret in place and the block storage plug-in installed, you are now ready to create your Persistent Volume. A Persistent Volume, or PV, is block storage of a specified size that lives independently of a pod's life cycle. Using a Persistent Volume will allow you to manage or update your pods without worrying about losing your application code. A Persistent Volume is accessed by using a `PersistentVolumeClaim`, or PVC, which mounts the PV at the required path.

Open a file named `code_volume.yaml` with your editor:

```
nano code_volume.yaml
```

Name the PVC `code` by adding the following parameters and values to your file:

code_volume.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code
```

The `spec` for a PVC contains the following items:

- `accessModes` which vary by the use case. These are:
    - `ReadWriteOnce` – mounts the volume as read-write by a single node

    - `ReadOnlyMany` – mounts the volume as read-only by many nodes
    - `ReadWriteMany` – mounts the volume as read-write by many nodes

- `resources` – the storage space that you require

DigitalOcean block storage is only mounted to a single node, so you will set the `accessModes` to `ReadWriteOnce`. This tutorial will guide you through adding a small amount of application code, so 1GB will be plenty in this use case. If you plan on storing a larger amount of code or data on the volume, you can modify the `storage` parameter to fit your requirements. You can increase the amount of storage after volume creation, but shrinking the disk is not supported.

code_volume.yaml

```
...
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Next, specify the storage class that Kubernetes will use to provision the volumes. You will use the `do-block-storage` class created by the DigitalOcean block storage plug-in.

code_volume.yaml

```
...
  storageClassName: do-block-storage
```

Your `code_volume.yaml` file will look like this:

code_volume.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: code
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: do-block-storage
```

Save and exit the file.

Create the `code` PVC using `kubectl`:

```
kubectl apply -f code_volume.yaml
```

The following output tells you that the object was successfully created, and you are ready to mount your 1GB PVC as a volume.

Output

```
persistentvolumeclaim/code created
```

To view available Persistent Volumes (PV):

```
kubectl get pv
```

You will see your PV listed:

Output

```
NAME
CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM          STORAGECLASS       REASON    AGE
pvc-ca4df10f-ab8c-11e8-b89d-12331aa95b13   1Gi
RWO            Delete           Bound
default/code   do-block-storage             2m
```

The fields above are an overview of your configuration file, except for `Reclaim Policy` and `Status`. The `Reclaim Policy` defines what is done with the PV after the PVC accessing it is deleted. `Delete` removes the PV from Kubernetes as well as the DigitalOcean infrastructure. You can learn more about the `Reclaim Policy` and `Status` from the Kubernetes PV documentation.

You've successfully created a Persistent Volume using the DigitalOcean block storage plug-in. Now that your Persistent Volume is ready, you will create your pods using a Deployment.

Step 4 — Creating a PHP-FPM Deployment

In this step, you will learn how to use a Deployment to create your PHP-FPM pod. Deployments provide a uniform way to create, update, and manage pods by using ReplicaSets. If an update does not work as expected, a Deployment will automatically rollback its pods to a previous image.

The Deployment `spec.selector` key will list the labels of the pods it will manage. It will also use the `template` key to create the required pods.

This step will also introduce the use of Init Containers. Init Containers run one or more commands before the regular containers specified under the pod's `template` key. In this tutorial, your Init Container will fetch a sample `index.php` file from GitHub Gist using `wget`. These are the contents of the sample file:

index.php

```
<?php
echo phpinfo();
```

To create your Deployment, open a new file called `php_deployment.yaml` with your editor:

```
nano php_deployment.yaml
```

This Deployment will manage your PHP-FPM pods, so you will name the Deployment object `php`. The pods belong to the back-end tier, so you will group the Deployment into this group by using the `tier: backend` label:

php_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php
  labels:
    tier: backend
```

For the Deployment `spec`, you will specify how many copies of this pod to create by using the `replicas` parameter. The number of `replicas` will vary depending on your needs and available resources. You will create one replica in this tutorial:

php_deployment.yaml

```
...
spec:
  replicas: 1
```

This Deployment will manage pods that match the `app: php` and `tier: backend` labels. Under `selector` key add:

php_deployment.yaml

```
...
  selector:
    matchLabels:
      app: php
      tier: backend
```

Next, the Deployment `spec` requires the `template` for your pod's object definition. This template will define specifications to create the pod from. First, you will add the labels that were specified for the `php` service `selectors` and the Deployment's `matchLabels`. Add `app: php` and `tier: backend` under `template.metadata.labels`:

php_deployment.yaml

```
...
  template:
    metadata:
      labels:
        app: php
        tier: backend
```

A pod can have multiple containers and volumes, but each will need a name. You can selectively mount volumes to a container by specifying a mount path for each volume.

First, specify the volumes that your containers will access. You created a PVC named `code` to hold your application code, so name this volume `code` as well. Under `spec.template.spec.volumes`, add the following:

php_deployment.yaml

```
...
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
```

Next, specify the container you want to run in this pod. You can find various images on [the Docker store](), but in this tutorial you will use the `php:7-fpm` image.

Under `spec.template.spec.containers`, add the following:

php_deployment.yaml

```
...
      containers:
      - name: php
        image: php:7-fpm
```

Next, you will mount the volumes that the container requires access to. This container will run your PHP code, so it will need access to the `code` volume. You will also use `mountPath` to specify `/code` as the mount point.

Under `spec.template.spec.containers.volumeMounts`, add:

php_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

Now that you have mounted your volume, you need to get your application code on the volume. You may have previously used FTP/SFTP or cloned the code over an SSH connection to accomplish this, but this step will show you how to copy the code using an Init Container.

Depending on the complexity of your setup process, you can either use a single `initContainer` to run a script that builds your application, or you can use one `initContainer` per command. Make sure that the volumes are mounted to the `initContainer`.

In this tutorial, you will use a single Init Container with `busybox` to download the code. `busybox` is a small image that contains the `wget` utility that you will use to accomplish this.

Under `spec.template.spec`, add your `initContainer` and specify the `busybox` image:

php_deployment.yaml

```
...
      initContainers:
      - name: install
        image: busybox
```

Your Init Container will need access to the `code` volume so that it can download the code in that location. Under `spec.template.spec.initContainers`, mount the volume `code` at the `/code` path:

php_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

Each Init Container needs to run a `command`. Your Init Container will use `wget` to download the code from Github into the `/code` working directory. The `-O` option gives the downloaded file a name, and you will name this file `index.php`.

Note: Be sure to trust the code you're pulling. Before pulling it to your server, inspect the source code to ensure you are comfortable with what the code does.

Under the `install` container in `spec.template.spec.initContainers`, add these lines:

php_deployment.yaml

```
...
        command:
```

```
        - wget
        - "-O"
        - "/code/index.php"
        - https://raw.githubusercontent.com/do-community/php-kubernetes/master/index.php
```

Your completed `php_deployment.yaml` file will look like this:

php_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php
  labels:
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: php
      tier: backend
  template:
    metadata:
      labels:
        app: php
        tier: backend
    spec:
      volumes:
```

```
      - name: code
        persistentVolumeClaim:
          claimName: code
      containers:
      - name: php
        image: php:7-fpm
        volumeMounts:
        - name: code
          mountPath: /code
      initContainers:
      - name: install
        image: busybox
        volumeMounts:
        - name: code
          mountPath: /code
        command:
        - wget
        - "-O"
        - "/code/index.php"
        - https://raw.githubusercontent.com/do-
community/php-kubernetes/master/index.php
```

Save the file and exit the editor.

Create the PHP-FPM Deployment with `kubectl`:

```
kubectl apply -f php_deployment.yaml
```

You will see the following output upon Deployment creation:

Output

```
deployment.apps/php created
```

To summarize, this Deployment will start by downloading the specified images. It will then request the `PersistentVolume` from your `PersistentVolumeClaim` and serially run your `initContainers`. Once complete, the containers will run and mount the `volumes` to the specified mount point. Once all of these steps are complete, your pod will be up and running.

You can view your Deployment by running:

```
kubectl get deployments
```

You will see the output:

Output

```
NAME       DESIRED    CURRENT    UP-TO-DATE
AVAILABLE    AGE
php        1          1          1            0
19s
```

This output can help you understand the current state of the Deployment. A `Deployment` is one of the controllers that maintains a desired state. The `template` you created specifies that the `DESIRED` state will have 1 `replicas` of the pod named `php`. The `CURRENT` field indicates how many replicas are running, so this should match the `DESIRED` state. You can read about the remaining fields in the Kubernetes Deployments documentation.

You can view the pods that this Deployment started with the following command:

```
kubectl get pods
```

The output of this command varies depending on how much time has passed since creating the Deployment. If you run it shortly after creation, the output will likely look like this:

Output
```
NAME                   READY     STATUS
RESTARTS   AGE
php-86d59fd666-bf8zd   0/1       Init:0/1   0
9s
```

The columns represent the following information:

- `Ready`: The number of `replicas` running this pod.
- `Status`: The status of the pod. `Init` indicates that the Init Containers are running. In this output, 0 out of 1 Init Containers have finished running.
- `Restarts`: How many times this process has restarted to start the pod. This number will increase if any of your Init Containers fail. The Deployment will restart it until it reaches a desired state.

Depending on the complexity of your startup scripts, it can take a couple of minutes for the status to change to `podInitializing`:

Output
```
NAME                   READY     STATUS
RESTARTS   AGE
php-86d59fd666-lkwgn   0/1       podInitializing
0          39s
```

This means the Init Containers have finished and the containers are initializing. If you run the command when all of the containers are running, you will see the pod status change to `Running`.

Output

```
NAME                      READY     STATUS
RESTARTS   AGE
php-86d59fd666-lkwgn   1/1       Running   0
1m
```

You now see that your pod is running successfully. If your pod doesn't start, you can debug with the following commands:

- View detailed information of a pod:

```
kubectl describe pods pod-name
```

- View logs generated by a pod:

```
kubectl logs pod-name
```

- View logs for a specific container in a pod:

```
kubectl logs pod-name container-name
```

Your application code is mounted and the PHP-FPM service is now ready to handle connections. You can now create your Nginx Deployment.

Step 5 — Creating the Nginx Deployment

In this step, you will use a ConfigMap to configure Nginx. A ConfigMap holds your configuration in a key-value format that you can reference in other Kubernetes object definitions. This approach will grant you the flexibility to reuse or swap the image with a different Nginx version if needed. Updating the ConfigMap will automatically replicate the changes to any pod mounting it.

Create a `nginx_configMap.yaml` file for your ConfigMap with your editor:

```
nano nginx_configMap.yaml
```

Name the ConfigMap `nginx-config` and group it into the `tier: backend` micro-service:

nginx_configMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  labels:
    tier: backend
```

Next, you will add the `data` for the ConfigMap. Name the key `config` and add the contents of your Nginx configuration file as the value. You can use the example Nginx configuration from this tutorial.

Because Kubernetes can route requests to the appropriate host for a service, you can enter the name of your PHP-FPM service in the `fastcgi_pass` parameter instead of its IP address. Add the following to your `nginx_configMap.yaml` file:

nginx_configMap.yaml

```
...
data:
  config : |
    server {
      index index.php index.html;
      error_log  /var/log/nginx/error.log;
      access_log /var/log/nginx/access.log;
      root /code;

      location / {
          try_files $uri $uri/ /index.php?
$query_string;
      }

      location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)
(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO
$fastcgi_path_info;
```

```
        }
    }
```

Your `nginx_configMap.yaml` file will look like this:

nginx_configMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  labels:
    tier: backend
data:
  config : |
    server {
      index index.php index.html;
      error_log  /var/log/nginx/error.log;
      access_log /var/log/nginx/access.log;
      root /code;

      location / {
          try_files $uri $uri/ /index.php?
$query_string;
      }

      location ~ \.php$ {
          try_files $uri =404;
          fastcgi_split_path_info ^(.+\.php)
(/.+)$;
          fastcgi_pass php:9000;
          fastcgi_index index.php;
          include fastcgi_params;
          fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
          fastcgi_param PATH_INFO
$fastcgi_path_info;
        }
    }
```

Save the file and exit the editor.

Create the ConfigMap:

```
kubectl apply -f nginx_configMap.yaml
```

You will see the following output:

Output

```
configmap/nginx-config created
```

You've finished creating your ConfigMap and can now build your Nginx Deployment.

Start by opening a new `nginx_deployment.yaml` file in the editor:

```
nano nginx_deployment.yaml
```

Name the Deployment `nginx` and add the label `tier: backend`:

nginx_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    tier: backend
```

Specify that you want one `replicas` in the Deployment `spec`. This Deployment will manage pods with labels `app: nginx` and `tier: backend`. Add the following parameters and values:

nginx_deployment.yaml

```
...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      tier: backend
```

Next, add the pod `template`. You need to use the same labels that you added for the Deployment `selector.matchLabels`. Add the following:

nginx_deployment.yaml

```
...
  template:
    metadata:
      labels:
        app: nginx
        tier: backend
```

Give Nginx access to the `code` PVC that you created earlier. Under `spec.template.spec.volumes`, add:

nginx_deployment.yaml

```
...
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
```

Pods can mount a ConfigMap as a volume. Specifying a file name and key will create a file with its value as the content. To use the ConfigMap, set `path` to name of the file that will hold the contents of the `key`. You want to create a file `site.conf` from the key `config`. Under `spec.template.spec.volumes`, add the following:

nginx_deployment.yaml

```
...
      - name: config
        configMap:
          name: nginx-config
          items:
          - key: config
            path: site.conf
```

Warning: If a file is not specified, the contents of the `key` will replace the `mountPath` of the volume. This means that if a path is not explicitly specified, you will lose all content in the destination folder.

Next, you will specify the image to create your pod from. This tutorial will use the `nginx:1.7.9` image for stability, but you can find other Nginx images on the Docker store. Also, make Nginx available on the port `80`. Under `spec.template.spec` add:

nginx_deployment.yaml

```
...
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Nginx and PHP-FPM need to access the file at the same path, so mount the `code` volume at `/code`:

nginx_deployment.yaml

```
...
        volumeMounts:
        - name: code
          mountPath: /code
```

The `nginx:1.7.9` image will automatically load any configuration files under the `/etc/nginx/conf.d` directory. Mounting the `config` volume in this directory will create the file `/etc/nginx/conf.d/site.conf`. Under `volumeMounts` add the following:

nginx_deployment.yaml

```
...
        - name: config
          mountPath: /etc/nginx/conf.d
```

Your `nginx_deployment.yaml` file will look like this:

nginx_deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      tier: backend
  template:
    metadata:
      labels:
        app: nginx
        tier: backend
    spec:
      volumes:
      - name: code
        persistentVolumeClaim:
          claimName: code
      - name: config
        configMap:
          name: nginx-config
          items:
          - key: config
            path: site.conf
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        volumeMounts:
        - name: code
          mountPath: /code
        - name: config
          mountPath: /etc/nginx/conf.d
```

Save the file and exit the editor.

Create the Nginx Deployment:

```
kubectl apply -f nginx_deployment.yaml
```

The following output indicates that your Deployment is now created:

Output

```
deployment.apps/nginx created
```

List your Deployments with this command:

```
kubectl get deployments
```

You will see the Nginx and PHP-FPM Deployments:

Output

```
NAME       DESIRED   CURRENT   UP-TO-DATE
AVAILABLE   AGE
nginx     1         1         1            0
16s
php       1         1         1            1
7m
```

List the pods managed by both of the Deployments:

```
kubectl get pods
```

You will see the pods that are running:

Output

```
NAME                        READY     STATUS
RESTARTS   AGE
nginx-7bf5476b6f-zppml   1/1       Running   0
32s
php-86d59fd666-lkwgn     1/1       Running   0
7m
```

Now that all of the Kubernetes objects are active, you can visit the Nginx service on your browser.

List the running services:

```
kubectl get services -o wide
```

Get the External IP for your Nginx service:

Output

```
NAME          TYPE         CLUSTER-IP       EXTERNAL-
IP   PORT(S)     AGE         SELECTOR
```

```
kubernetes   ClusterIP   10.96.0.1       <none>
443/TCP    39m        <none>
nginx        ClusterIP   10.102.160.47
your_public_ip 80/TCP     27m
app=nginx,tier=backend
php          ClusterIP   10.100.59.238   <none>
9000/TCP   34m        app=php,tier=backend
```

On your browser, visit your server by typing in `http://your_public_ip`. You will see the output of `php_info()` and have confirmed that your Kubernetes services are up and running.

Conclusion

In this guide, you containerized the PHP-FPM and Nginx services so that you can manage them independently. This approach will not only improve the scalability of your project as you grow, but will also allow you to efficiently use resources as well. You also stored your application code on a volume so that you can easily update your services in the future.

How To Automate Deployments to DigitalOcean Kubernetes with CircleCI

Written by Jonathan Cardoso

Having an automated deployment process is a requirement for a scalable and resilient application. Tools like CircleCI allow you to test and deploy your code automatically every time you make a change to your source code repository. When this kind of CI/CD is combined with the flexibility of Kubernetes infrastructure, you can build an application that scales easily with changing demand.

In this article you will use CircleCI to deploy a sample application to a DigitalOcean Kubernetes cluster. After reading this tutorial, you'll be able to apply these same techniques to deploy other CI/CD tools that are buildable as Docker images.

The author selected the Tech Education Fund to receive a donation as part of the Write for DOnations program.

Having an automated deployment process is a requirement for a scalable and resilient application, and GitOps, or Git-based DevOps, has rapidly become a popular method of organizing CI/CD with a Git repository as a "single source of truth." Tools like CircleCI integrate with your GitHub repository, allowing you to test and deploy your code automatically every time you make a change to your repository. When this kind of CI/CD is combined with the flexibility of Kubernetes infrastructure, you can build an application that scales easily with changing demand.

In this article you will use CircleCI to deploy a sample application to a DigitalOcean Kubernetes (DOKS) cluster. After reading this tutorial, you'll be able to apply these same techniques to deploy other CI/CD tools that are buildable as Docker images.

Prerequisites

To follow this tutorial, you'll need to have:

- A DigitalOcean account, which you can set up by following the Sign up for a DigitalOcean Account documentation.
- Docker installed on your workstation, and knowledge of how to build, remove, and run Docker images. You can install Docker on Ubuntu 18.04 by following the tutorial on How To Install and Use Docker on Ubuntu 18.04.
- Knowledge of how Kubernetes works and how to create deployments and services on it. It's highly recommended to read the Introduction to Kubernetes article.
- The `kubectl` command line interface tool installed on the computer from which you will control your cluster.
- An account on Docker Hub to be used to store your sample application image.
- A GitHub account and knowledge of Git basics. You can follow the tutorial series Introduction to Git: Installation, Usage, and Branches and How To Create a Pull Request on GitHub to build this knowledge.

For this tutorial, you will use Kubernetes version `1.13.5` and `kubectl` version `1.10.7`.

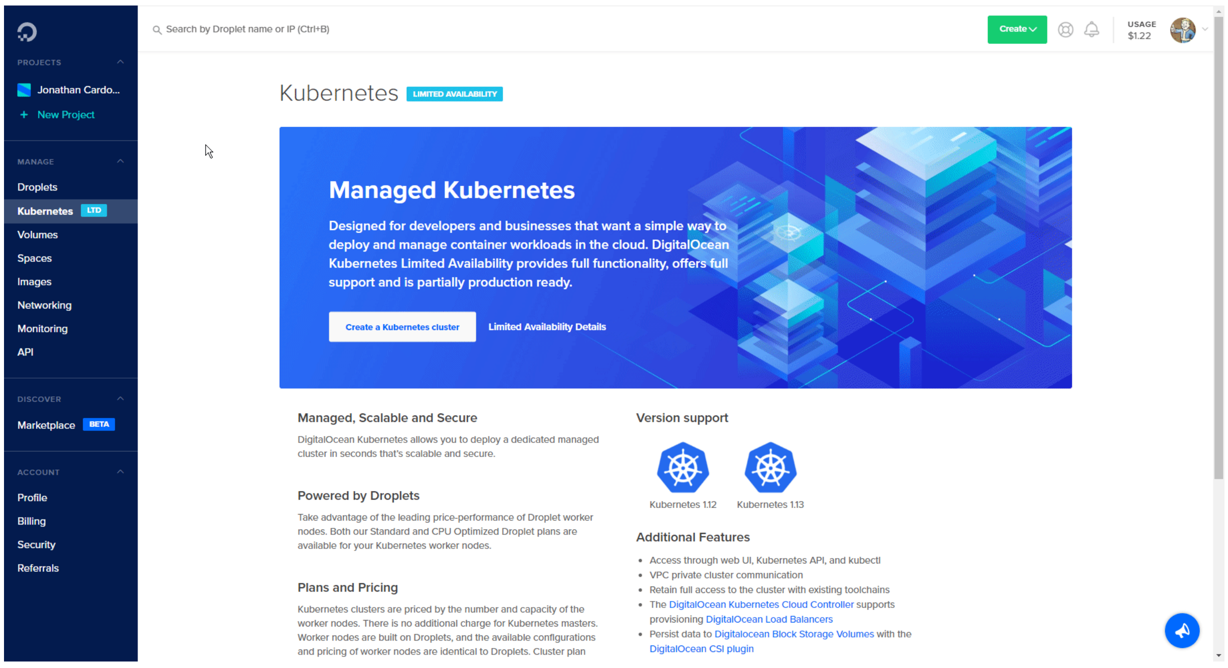
Step 1 — Creating Your DigitalOcean Kubernetes Cluster

Note: You can skip this section if you already have a running DigitalOcean Kubernetes cluster.

In this first step, you will create the DigitalOcean Kubernetes (DOKS) cluster from which you will deploy your sample application. The `kubectl` commands executed from your local machine will change or retrieve information directly from the Kubernetes cluster.

Go to the Kubernetes page on your DigitalOcean account.

Click Create a Kubernetes cluster, or click the green Create button at the top right of the page and select Clusters from the dropdown menu.



Managed Kubernetes

Designed for developers and businesses that want a simple way to deploy and manage container workloads in the cloud. DigitalOcean Kubernetes Limited Availability provides full functionality, offers full support and is partially production ready.

Managed, Scalable and Secure

DigitalOcean Kubernetes allows you to deploy a dedicated managed cluster in seconds that's scalable and secure.

Powered by Droplets

Take advantage of the leading price-performance of Droplet worker nodes. Both our Standard and CPU Optimized Droplet plans are available for your Kubernetes worker nodes.

Plans and Pricing

Kubernetes clusters are priced by the number and capacity of the worker nodes. There is no additional charge for Kubernetes masters. Worker nodes are built on Droplets, and the available configurations and pricing of worker nodes are identical to Droplets. Cluster plan

Version support

Additional Features

- Access through web UI, Kubernetes API, and kubectl
- VPC private cluster communication
- Retain full access to the cluster with existing toolchains
- The DigitalOcean Kubernetes Cloud Controller supports provisioning DigitalOcean Load Balancers
- Persist data to Digitalocean Block Storage Volumes with the DigitalOcean CSI plugin

Creating a Kubernetes Cluster on DigitalOcean

The next page is where you are going to specify the details of your cluster. On Select a Kubernetes version pick version 1.13.5-do.0. If this one is not available, choose a higher one.

For Choose a datacenter region, choose the region closest to you. This tutorial will use San Francisco - 2.

You then have the option to build your Node pool(s). On Kubernetes, a node is a worker machine, which contains the services necessary to run pods. On DigitalOcean, each node is a Droplet. Your node pool will consist of a single Standard node. Select the 2GB/1vCPU configuration and change to 1 Node on the number of nodes.

You can add extra tags if you want; this can be useful if you plan to use DigitalOcean API or just to better organize your node pools.

On Choose a name, for this tutorial, use `kubernetes-deployment-tutorial`. This will make it easier to follow throughout while reading the next sections. Finally, click the green Create Cluster button to create your cluster.

After cluster creation, there will be a button on the UI to download a configuration file called Download Config File. This is the file you will be using to authenticate the `kubectl` commands you are going to run against your cluster. Download it to your `kubectl` machine.

The default way to use that file is to always pass the `--kubeconfig` flag and the path to it on all commands you run with `kubectl`. For example, if you downloaded the config file to `Desktop`, you would run the `kubectl get pods` command like this:

```
kubectl --kubeconfig ~/Desktop/kubernetes-
deployment-tutorial-kubeconfig.yaml get pods
```

This would yield the following output:

Output

```
No resources found.
```

This means you accessed your cluster. The `No resources found.` message is correct, since you don't have any pods on your cluster.

If you are not maintaining any other Kubernetes clusters you can copy the kubeconfig file to a folder on your home directory called `.kube`. Create that directory in case it does not exist:

```
mkdir -p ~/.kube
```

Then copy the config file into the newly created `.kube` directory and rename it `config`:

```
cp current_kubernetes-deployment-tutorial-
kubeconfig.yaml_file_path ~/.kube/config
```

The config file should now have the path `~/.kube/config`. This is the file that `kubectl` reads by default when running any command, so there is no need to pass `--kubeconfig` anymore. Run the following:

```
kubectl get pods
```

You will receive the following output:

Output

```
No resources found.
```

Now access the cluster with the following:

```
kubectl get nodes
```

You will receive the list of nodes on your cluster. The output will be similar to this:

Output

```
NAME                                       STATUS
ROLES     AGE       VERSION
```

```
kubernetes-deployment-tutorial-1-7pto   Ready
<none>    1h         v1.13.5
```

In this tutorial you are going to use the `default` namespace for all `kubectl` commands and manifest files, which are files that define the workload and operating parameters of work in Kubernetes. Namespaces are like virtual clusters inside your single physical cluster. You can change to any other namespace you want; just make sure to always pass it using the `--namespace` flag to `kubectl`, and/or specifying it on the Kubernetes manifests metadata field. They are a great way to organize the deployments of your team and their running environments; read more about them in the official Kubernetes overview on Namespaces.

By finishing this step you are now able to run `kubectl` against your cluster. In the next step, you will create the local Git repository you are going to use to house your sample application.

Step 2 — Creating the Local Git Repository

You are now going to structure your sample deployment in a local Git repository. You will also create some Kubernetes manifests that will be global to all deployments you are going to do on your cluster.

Note: This tutorial has been tested on Ubuntu 18.04, and the individual commands are styled to match this OS. However, most of the commands here can be applied to other Linux distributions with little to no change needed, and commands like `kubectl` are platform-agnostic.

First, create a new Git repository locally that you will push to GitHub later on. Create an empty folder called `do-sample-app` in your home directory and `cd` into it:

```
mkdir ~/do-sample-app
cd ~/do-sample-app
```

Now create a new Git repository in this folder with the following command:

```
git init .
```

Inside this repository, create an empty folder called `kube`:

```
mkdir ~/do-sample-app/kube/
```

This will be the location where you are going to store the Kubernetes resources manifests related to the sample application that you will deploy to your cluster.

Now, create another folder called `kube-general`, but this time outside of the Git repository you just created. Make it inside your home directory:

```
mkdir ~/kube-general/
```

This folder is outside of your Git repository because it will be used to store manifests that are not specific to a single deployment on your cluster, but common to multiple ones. This will allow you to reuse these general manifests for different deployments.

With your folders created and the Git repository of your sample application in place, it's time to arrange the authentication and authorization of your DOKS cluster.

Step 3 — Creating a Service Account

It's generally not recommended to use the default admin user to authenticate from other Services into your Kubernetes cluster. If your keys on the external provider got compromised, your whole cluster would become compromised.

Instead you are going to use a single Service Account with a specific Role, which is all part of the RBAC Kubernetes authorization model.

This authorization model is based on Roles and Resources. You start by creating a Service Account, which is basically a user on your cluster, then you create a Role, in which you specify what resources it has access to on your cluster. Finally, you create a Role Binding, which is used to make the connection between the Role and the Service Account previously created, granting to the Service Account access to all resources the Role has access to.

The first Kubernetes resource you are going to create is the Service Account for your CI/CD user, which this tutorial will name `cicd`.

Create the file `cicd-service-account.yml` inside the `~/kube-general` folder, and open it with your favorite text editor:

```
nano ~/kube-general/cicd-service-account.yml
```

Write the following content on it:

~/kube-general/cicd-service-account.yml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cicd
  namespace: default
```

This is a YAML file; all Kubernetes resources are represented using one. In this case you are saying this resource is from Kubernetes API version `v1` (internally `kubectl` creates resources by calling Kubernetes HTTP APIs), and it is a `ServiceAccount`.

The `metadata` field is used to add more information about this resource. In this case, you are giving this `ServiceAccount` the name `cicd`, and creating it on the `default` namespace.

You can now create this Service Account on your cluster by running `kubectl apply`, like the following:

```
kubectl apply -f ~/kube-general/
```

You will recieve output similar to the following:

Output

```
serviceaccount/cicd created
```

To make sure your Service Account is working, try to log in to your cluster using it. To do that you first need to obtain their respective access token and store it in an environment variable. Every Service Account has an access token which Kubernetes stores as a Secret.

You can retrieve this secret using the following command:

```
TOKEN=$(kubectl get secret $(kubectl get secret |
grep cicd-token | awk '{print $1}') -o
jsonpath='{.data.token}' | base64 --decode)
```

Some explanation on what this command is doing:

```
$(kubectl get secret | grep cicd-token | awk
'{print $1}')
```

This is used to retrieve the name of the secret related to our `cicd` Service Account. `kubectl get secret` returns the list of secrets on the default namespace, then you use `grep` to search for the lines related to your `cicd` Service Account. Then you return the name, since it is the first thing on the single line returned from the `grep`.

```
kubectl get secret preceding-command -o
jsonpath='{.data.token}' | base64 --decode
```

This will retrieve only the secret for your Service Account token. You then access the token field using `jsonpath`, and pass the result to `base64 --decode`. This is necessary because the token is stored as a Base64 string. The token itself is a JSON Web Token.

You can now try to retrieve your pods with the `cicd` Service Account. Run the following command, replacing **`server-from-kubeconfig-file`** with the server URL that can be found after `server:` in `~kube/config`. This command will give a specific error that you will learn about later in this tutorial:

```
kubectl --insecure-skip-tls-verify --
kubeconfig="/dev/null" --server=server-from-
kubeconfig-file --token=$TOKEN get pods
```

`--insecure-skip-tls-verify` skips the step of verifying the certificate of the server, since you are just testing and do not need to verify this. `--kubeconfig="/dev/null"` is to make sure `kubectl` does not read your config file and credentials but instead uses the token provided.

The output should be similar to this:

Output

```
Error from server (Forbidden): pods is forbidden:
User "system:serviceaccount:default:cicd" cannot
```

```
list resource "pods" in API group "" in the
namespace "default"
```

This is an error, but it shows us that the token worked. The error you received is about your Service Account not having the neccessary authorization to list the resource `secrets`, but you were able to access the server itself. If your token had not worked, the error would have been the following one:

Output

```
error: You must be logged in to the server
(Unauthorized)
```

Now that the authentication was a success, the next step is to fix the authorization error for the Service Account. You will do this by creating a role with the necessary permissions and binding it to your Service Account.

Step 4 — Creating the Role and the Role Binding

Kubernetes has two ways to define roles: using a `Role` or a `ClusterRole` resource. The difference between the former and the latter is that the first one applies to a single namespace, while the other is valid for the whole cluster.

As you are using a single namespace on this tutorial, you will use a `Role`.

Create the file `~/kube-general/cicd-role.yml` and open it with your favorite text editor:

```
nano ~/kube-general/cicd-role.yml
```

The basic idea is to grant access to do everything related to most Kubernetes resources in the `default` namespace. Your `Role` would look like this:

~/kube-general/cicd-role.yml

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cicd
  namespace: default
rules:
  - apiGroups: ["", "apps", "batch", "extensions"]
    resources: ["deployments", "services",
"replicasets", "pods", "jobs", "cronjobs"]
    verbs: ["*"]
```

This YAML has some similarities with the one you created previously, but here you are saying this resource is a `Role`, and it's from the Kubernetes API `rbac.authorization.k8s.io/v1`. You are naming your role `cicd`, and creating it on the same namespace you created your `ServiceAccount`, the `default` one.

Then you have the `rules` field, which is a list of resources this role has access to. In Kubernetes resources are defined based on the API group they belong to, the resource kind itself, and what actions you can do on then, which is represented by a verb. Those verbs are similar to the HTTP ones.

In our case you are saying that your `Role` is allowed to do everything, `*`, on the following resources: `deployments`, `services`, `replicasets`, `pods`, `jobs`, and `cronjobs`. This also applies to those resources belonging to the following API groups: `""` (empty string), `apps`, `batch`, and `extensions`. The empty string means the root API group. If you use `apiVersion: v1` when creating a resource it means this resource is part of this API group.

A `Role` by itself does nothing; you must also create a [`RoleBinding`](), which binds a `Role` to something, in this case, a `ServiceAccount`.

Create the file `~/kube-general/cicd-role-binding.yml` and open it:

```
nano ~/kube-general/cicd-role-binding.yml
```

Add the following lines to the file:

~/kube-general/cicd-role-binding.yml

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cicd
  namespace: default
subjects:
  - kind: ServiceAccount
    name: cicd
    namespace: default
roleRef:
  kind: Role
  name: cicd
  apiGroup: rbac.authorization.k8s.io
```

Your `RoleBinding` has some specific fields that have not yet been covered in this tutorial. `roleRef` is the `Role` you want to bind to something; in this case it is the `cicd` role you created earlier. `subjects` is the list of resources you are binding your role to; in this case it's a single `ServiceAccount` called `cicd`.

Note: If you had used a `ClusterRole`, you would have to create a `ClusterRoleBinding` instead of a `RoleBinding`. The file would be almost the same. The only difference would be that it would have no `namespace` field inside the `metadata`.

With those files created you will be able to use `kubectl apply` again. Create those new resources on your Kubernetes cluster by running the following command:

```
kubectl apply -f ~/kube-general/
```

You will receive output similar to the following:

Output

```
rolebinding.rbac.authorization.k8s.io/cicd created
role.rbac.authorization.k8s.io/cicd created
serviceaccount/cicd created
```

Now, try the command you ran previously:

```
kubectl --insecure-skip-tls-verify --
kubeconfig="/dev/null" --server=server-from-
kubeconfig-file --token=$TOKEN get pods
```

Since you have no pods, this will yield the following output:

Output

```
No resources found.
```

In this step, you gave the Service Account you are going to use on CircleCI the necessary authorization to do meaningful actions on your cluster like listing, creating, and updating resources. Now it's time to create your sample application.

Step 5 — Creating Your Sample Application

Note: All commands and files created from now on will start from the folder `~/do-sample-app` you created earlier. This is becase you are now creating files specific to the sample application that you are going to deploy to your cluster.

The Kubernetes `Deployment` you are going to create will use the Nginx image as a base, and your application will be a simple static HTML page. This is a great start because it allows you to test if your deployment works by serving a simple HTML directly from Nginx. As you will see later on, you can redirect all traffic coming to a local `address:port` to your deployment on your cluster to test if it's working.

Inside the repository you set up earlier, create a new `Dockerfile` file and open it with your text editor of choice:

```
nano ~/do-sample-app/Dockerfile
```

Write the following on it:

~/do-sample-app/Dockerfile

```
FROM nginx:1.14


COPY index.html /usr/share/nginx/html/index.html
```

This will tell Docker to build the application container from an `nginx` image.

Now create a new `index.html` file and open it:

```
nano ~/do-sample-app/index.html
```

Write the following HTML content:

~/do-sample-app/index.html

```
<!DOCTYPE html>
<title>DigitalOcean</title>
<body>
  Kubernetes Sample Application
</body>
```

This HTML will display a simple message that will let you know if your application is working.

You can test if the image is correct by building and then running it.

First, build the image with the following command, replacing **`dockerhub-username`** with your own Docker Hub username. You must specify your username here so when you push it later on to Docker Hub it will just work:

```
docker build ~/do-sample-app/ -t dockerhub-
username/do-kubernetes-sample-app
```

Now run the image. Use the following command, which starts your image and forwards any local traffic on port `8080` to the port `80` inside the image, the port Nginx listens to by default:

```
docker run --rm -it -p 8080:80 dockerhub-
username/do-kubernetes-sample-app
```

The command prompt will stop being interactive while the command is running. Instead you will see the Nginx access logs. If you open `localhost:8080` on any browser it should show an HTML page with the content of `~/do-sample-app/index.html`. In case you don't have a browser available, you can open a new terminal window and use the following `curl` command to fetch the HTML from the webpage:

```
curl localhost:8080
```

You will receive the following output:

Output

```
<!DOCTYPE html>
<title>DigitalOcean</title>
<body>
  Kubernetes Sample Application
</body>
```

Stop the container (`CTRL` + `C` on the terminal where it's running), and submit this image to your Docker Hub account. To do this, first log in to Docker Hub:

```
docker login
```

Fill in the required information about your Docker Hub account, then push the image with the following command (don't forget to replace the **`dockerhub-username`** with your own):

```
docker push dockerhub-username/do-kubernetes-
sample-app
```

You have now pushed your sample application image to your Docker Hub account. In the next step, you will create a Deployment on your DOKS cluster from this image.

Step 6 — Creating the Kubernetes Deployment and Service

With your Docker image created and working, you will now create a manifest telling Kubernetes how to create a Deployment from it on your cluster.

Create the YAML deployment file `~/do-sample-app/kube/do-sample-deployment.yml` and open it with your text editor:

```
nano ~/do-sample-app/kube/do-sample-deployment.yml
```

Write the following content on the file, making sure to replace **`dockerhub-username`** with your Docker Hub username:

~/do-sample-app/kube/do-sample-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: do-kubernetes-sample-app
  namespace: default
  labels:
    app: do-kubernetes-sample-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: do-kubernetes-sample-app
  template:
    metadata:
      labels:
        app: do-kubernetes-sample-app
    spec:
      containers:
        - name: do-kubernetes-sample-app
          image: dockerhub-username/do-kubernetes-
sample-app:latest
          ports:
            - containerPort: 80
              name: http
```

Kubernetes deployments are from the API group `apps`, so the `apiVersion` of your manifest is set to `apps/v1`. On `metadata` you added a new field you have not used previously, called `metadata.labels`. This is useful to organize your deployments. The field `spec` represents the behavior specification of your deployment. A deployment is responsible for managing one or more pods; in this case it's going to have a single replica by the `spec.replicas` field. That is, it's going to create and manage a single pod.

To manage pods, your deployment must know which pods it's responsible for. The `spec.selector` field is the one that gives it that information. In this case the deployment will be responsible for all pods with tags `app=do-kubernetes-sample-app`. The `spec.template` field contains the details of the `Pod` this deployment will create. Inside the template you also have a `spec.template.metadata` field. The `labels` inside this field must match the ones used on `spec.selector`. `spec.template.spec` is the specification of the pod itself. In this case it contains a single container, called `do-kubernetes-sample-app`. The image of that container is the image you built previously and pushed to Docker Hub.

This YAML file also tells Kubernetes that this container exposes the port `80`, and gives this port the name `http`.

To access the port exposed by your `Deployment`, create a Service. Make a file named `~/do-sample-app/kube/do-sample-service.yml` and open it with your favorite editor:

```
nano ~/do-sample-app/kube/do-sample-service.yml
```

Next, add the following lines to the file:

~/do-sample-app/kube/do-sample-service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: do-kubernetes-sample-app
  namespace: default
  labels:
    app: do-kubernetes-sample-app
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: http
      name: http
  selector:
    app: do-kubernetes-sample-app
```

This file gives your `Service` the same labels used on your deployment. This is not required, but it helps to organize your applications on Kubernetes.

The service resource also has a `spec` field. The `spec.type` field is responsible for the behavior of the service. In this case it’s a `ClusterIP`, which means the service is exposed on a cluster-internal IP, and is only reachable from within your cluster. This is the default `spec.type` for services. `spec.selector` is the label selector criteria that should be used when picking the pods to be exposed by this service. Since your pod has the tag `app: do-kubernetes-sample-app`, you used it here. `spec.ports` are the ports exposed by the pod’s containers that you want to expose from this service. Your pod has a single container which exposes port `80`, named `http`, so you are using it here as `targetPort`. The service exposes that port on port `80` too, with the same name, but you could have used a different port/name combination than the one from the container.

With your `Service` and `Deployment` manifest files created, you can now create those resources on your Kubernetes cluster using `kubectl`:

```
kubectl apply -f ~/do-sample-app/kube/
```

You will receive the following output:

Output

```
deployment.apps/do-kubernetes-sample-app created
service/do-kubernetes-sample-app created
```

Test if this is working by forwarding one port on your machine to the port that the service is exposing inside your Kubernetes cluster. You can do that using `kubectl port-forward`:

```
kubectl port-forward $(kubectl get pod --
selector="app=do-kubernetes-sample-app" --output
jsonpath='{.items[0].metadata.name}') 8080:80
```

The subshell command `$(kubectl get pod --selector="app=do-kubernetes-sample-app" --output jsonpath='{.items[0].metadata.name}')` retrieves the name of the pod matching the tag you used. Otherwise you could have retrieved it from the list of pods by using `kubectl get pods`.

After you run `port-forward`, the shell will stop being interactive, and will instead output the requests redirected to your cluster:

Output
```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Opening `localhost:8080` on any browser should render the same page you saw when you ran the container locally, but it's now coming from your Kubernetes cluster! As before, you can also use `curl` in a new terminal window to check if it's working:

```
curl localhost:8080
```

You will receive the following output:

Output
```
<!DOCTYPE html>
<title>DigitalOcean</title>
<body>
  Kubernetes Sample Application
</body>
```

Next, it's time to push all the files you created to your GitHub repository. To do this you must first create a repository on GitHub called `digital-ocean-kubernetes-deploy`.

In order to keep this repository simple for demonstration purposes, do not initialize the new repository with a `README`, `license`, or `.gitignore` file when asked on the GitHub UI. You can add these files later on.

With the repository created, point your local repository to the one on GitHub. To do this, press `CTRL` + `C` to stop `kubectl port-forward` and get the command line back, then run the following commands to add a new remote called `origin`:

```
cd ~/do-sample-app/
git remote add origin https://github.com/your-github-account-username/digital-ocean-kubernetes-deploy.git
```

There should be no output from the preceding command.

Next, commit all the files you created up to now to the GitHub repository. First, add the files:

```
git add --all
```

Next, commit the files to your repository, with a commit message in quotation marks:

```
git commit -m "initial commit"
```

This will yield output similar to the following:

Output

```
[master (root-commit) db321ad] initial commit
 4 files changed, 47 insertions(+)
 create mode 100644 Dockerfile
 create mode 100644 index.html
 create mode 100644 kube/do-sample-deployment.yml
 create mode 100644 kube/do-sample-service.yml
```

Finally, push the files to GitHub:

```
git push -u origin master
```

You will be prompted for your username and password. Once you have entered this, you will see output like this:

Output

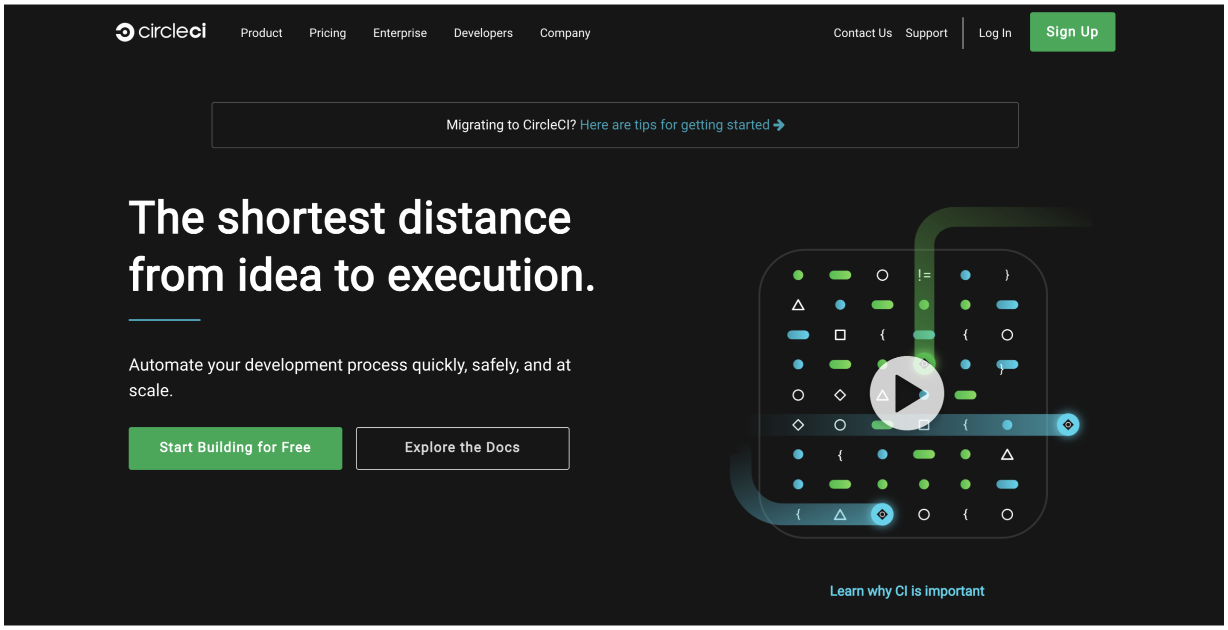
```
Counting objects: 7, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 907 bytes | 0
bytes/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To github.com:your-github-account-
username/digital-ocean-kubernetes-deploy.git
 * [new branch]      master -> master
Branch master set up to track remote branch master
from origin.
```

If you go to your GitHub repository page you will now see all the files there. With your project up on GitHub, you can now set up CircleCI as your CI/CD tool.

Step 7 — Configuring CircleCI

For this tutorial, you will use CircleCI to automate deployments of your application whenever the code is updated, so you will need to log in to CircleCI using your GitHub account and set up your repository.

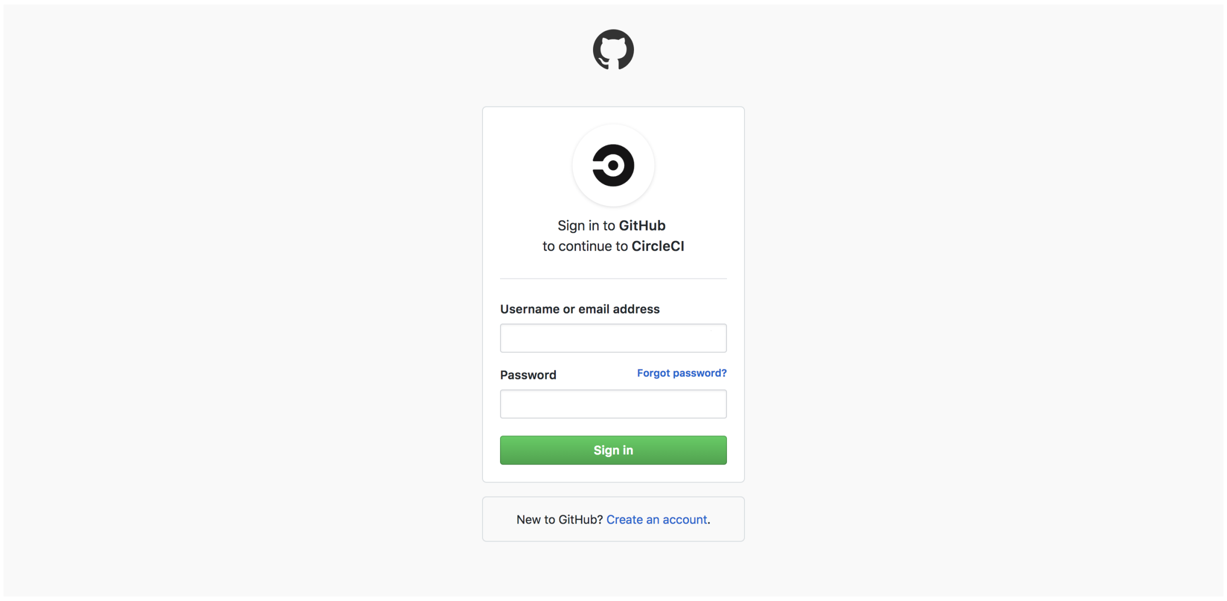
First, go to their homepage https://circleci.com, and press Sign Up.


circleci
Product
Pricing
Enterprise
Developers
Company
Contact Us
Support
Log In
Sign Up
Migrating to CircleCI? Here are tips for getting started
The shortest distance from idea to execution.
Automate your development process quickly, safely, and at scale.
Start Building for Free
Explore the Docs
Learn why CI is important

circleci-home-page

You are using GitHub, so click the green Sign Up with GitHub button.

CircleCI will redirect to an authorization page on GitHub. CircleCI needs some permissions on your account to be able to start building your projects. This allows CircleCI to obtain your email, deploy keys and permission to create hooks on your repositories, and add SSH keys to your account. If you need more information on what CircleCI is going to do with your data, check their documentation about GitHub integration.


Sign in to GitHub
to continue to CircleCI

Username or email address

Password

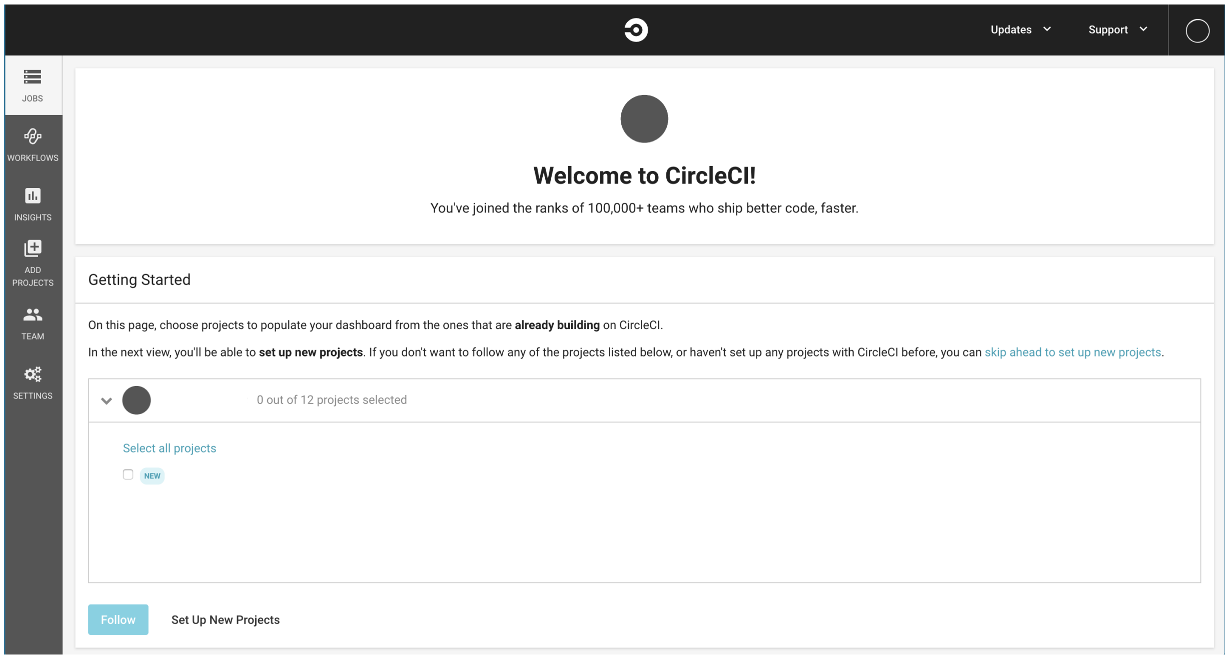
Forgot password?

Sign in

New to GitHub? Create an account.

circleci-github-authorization

After authorizing CircleCI you will be redirected to their dashboard.



Welcome to CircleCI!

You've joined the ranks of 100,000+ teams who ship better code, faster.

Getting Started

On this page, choose projects to populate your dashboard from the ones that are **already building** on CircleCI.

In the next view, you'll be able to **set up new projects**. If you don't want to follow any of the projects listed below, or haven't set up any projects with CircleCI before, you can skip ahead to set up new projects.

0 out of 12 projects selected

Select all projects

NEW

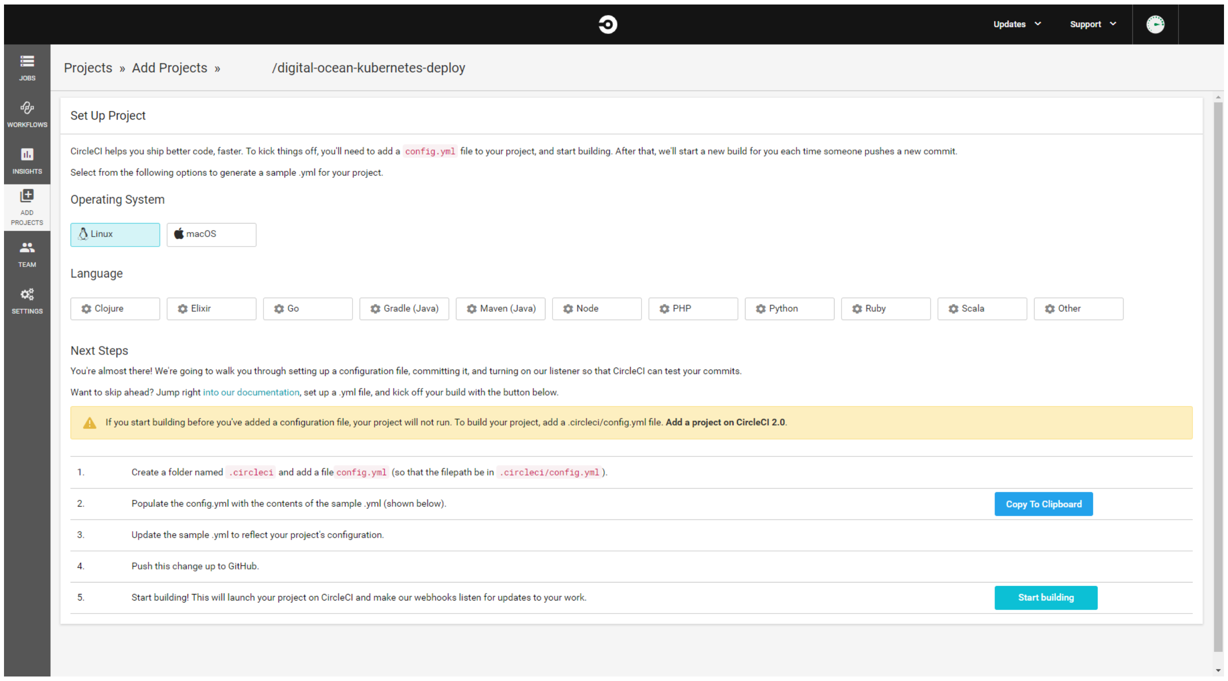
Follow

Set Up New Projects

circleci-project-dashboard

Next, set up your GitHub repository in CircleCI. Click on Set Up New Projects from the CircleCI Dashboard, or as a shortcut, open the following link changing the highlighted text with your own GitHub username: `https://circleci.com/setup-project/gh/`**`your-github-username`**`/digital-ocean-kubernetes-deploy`.

After that press Start Building. Do not create a config file in your repository just yet, and don't worry if the first build fails.



Projects » Add Projects » /digital-ocean-kubernetes-deploy

Set Up Project

CircleCI helps you ship better code, faster. To kick things off, you'll need to add a `config.yml` file to your project, and start building. After that, we'll start a new build for you each time someone pushes a new commit.

Select from the following options to generate a sample .yml for your project.

Operating System

[✓ Linux] macOS

Language

Clojure Elixir Go Gradle (Java) Maven (Java) Node PHP Python Ruby Scala Other

Next Steps

You're almost there! We're going to walk you through setting up a configuration file, committing it, and turning on our listener so that CircleCI can test your commits.

Want to skip ahead? Jump right into our documentation, set up a .yml file, and kick off your build with the button below.

If you start building before you've added a configuration file, your project will not run. To build your project, add a .circleci/config.yml file. **Add a project on CircleCI 2.0.**

1. Create a folder named `.circleci` and add a file `config.yml` (so that the filepath be in `.circleci/config.yml`).
2. Populate the config.yml with the contents of the sample .yml (shown below). Copy To Clipboard
3. Update the sample .yml to reflect your project's configuration.
4. Push this change up to GitHub.
5. Start building! This will launch your project on CircleCI and make our webhooks listen for updates to your work. Start building

circleci-start-building

Next, specify some environment variables in the CircleCI settings. You can find the settings of the project by clicking on the small button with a cog icon on the top right section of the page then selecting Environment Variables, or you can go directly to the environment variables page by using the following URL (remember to fill in your username):

`https://circleci.com/gh/`**`your-github-username`**`/digital-ocean-kubernetes-deploy/edit#env-vars`. Press Add Variable to create new environment variables.

First, add two environment variables called `DOCKERHUB_USERNAME` and `DOCKERHUB_PASS` which will be needed later on to push the image to Docker Hub. Set the values to your Docker Hub username and password, respectively.

Then add three more: `KUBERNETES_TOKEN`, `KUBERNETES_SERVER`, and `KUBERNETES_CLUSTER_CERTIFICATE`.

The value of `KUBERNETES_TOKEN` will be the value of the local environment variable you used earlier to authenticate on your Kubernetes cluster using your Service Account user. If you have closed the terminal, you can always run the following command to retrieve it again:

```
kubectl get secret $(kubectl get secret | grep
cicd-token | awk '{print $1}') -o
jsonpath='{.data.token}' | base64 --decode
```

`KUBERNETES_SERVER` will be the string you passed as the `--server` flag to `kubectl` when you logged in with your `cicd` Service Account. You can find this after `server:` in the `~/.kube/config` file, or in the file `kubernetes-deployment-tutorial-kubeconfig.yaml` downloaded from the DigitalOcean dashboard when you made the initial setup of your Kubernetes cluster.

`KUBERNETES_CLUSTER_CERTIFICATE` should also be available on your `~/.kube/config` file. It’s the `certificate-authority-data` field on the `clusters` item related to your cluster. It should be a long string; make sure to copy all of it.

Those environment variables must be defined here because most of them contain sensitive information, and it is not secure to place them directly on the CircleCI YAML config file.

With CircleCI listening for changes on your repository, and the environment variables configured, it's time to create the configuration file.

Make a directory called `.circleci` inside your sample application repository:

```
mkdir ~/do-sample-app/.circleci/
```

Inside this directory, create a file named `config.yml` and open it with your favorite editor:

```
nano ~/do-sample-app/.circleci/config.yml
```

Add the following content to the file, making sure to replace **`dockerhub-username`** with your Docker Hub username:

~/do-sample-app/.circleci/config.yml

```
version: 2.1
jobs:
  build:
    docker:
      - image: circleci/buildpack-deps:stretch
    environment:
      IMAGE_NAME: dockerhub-username/do-
kubernetes-sample-app
    working_directory: ~/app
    steps:
      - checkout
      - setup_remote_docker
      - run:
          name: Build Docker image
          command: |
            docker build -t $IMAGE_NAME:latest .
      - run:
          name: Push Docker Image
          command: |
            echo "$DOCKERHUB_PASS" | docker login
-u "$DOCKERHUB_USERNAME" --password-stdin
            docker push $IMAGE_NAME:latest
workflows:
  version: 2
  build-master:
    jobs:
      - build:
          filters:
            branches:
              only: master
```

This sets up a Workflow with a single job, called `build`, that runs for every commit to the `master` branch. This job is using the image `circleci/buildpack-deps:stretch` to run its steps, which is an image from CircleCI based on the official `buildpack-deps` Docker image, but with some extra tools installed, like Docker binaries themselves.

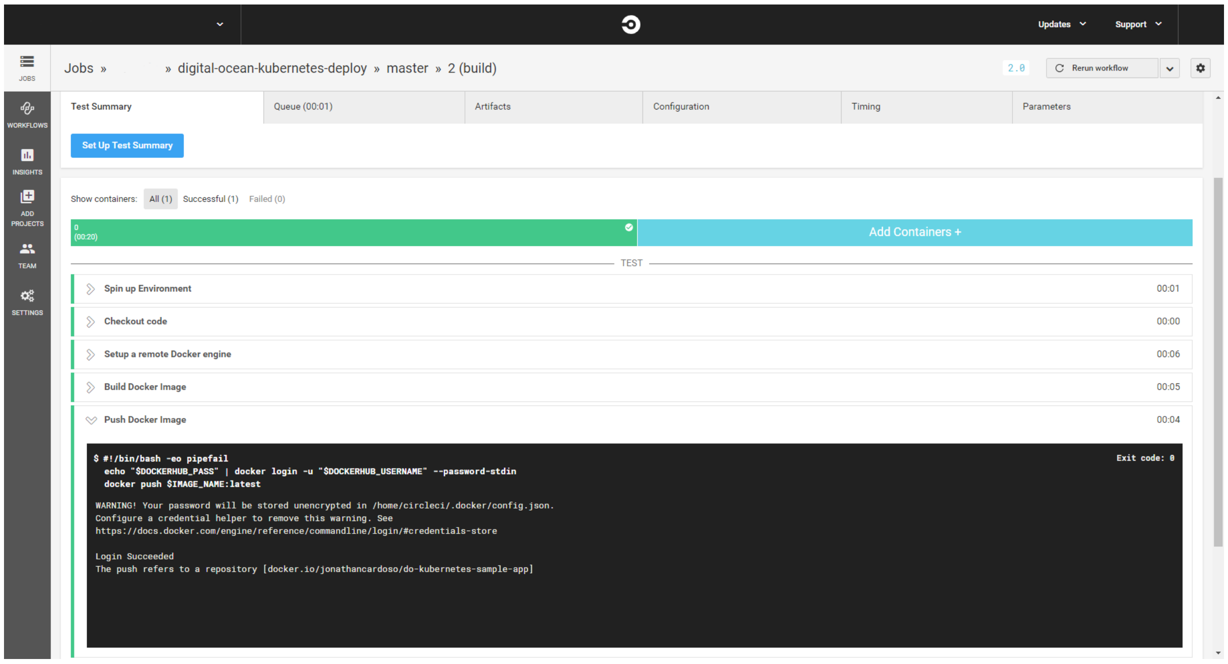
The workflow has four steps:

- `checkout` retrieves the code from GitHub.
- `setup_remote_docker` sets up a remote, isolated environment for each build. This is required before you use any `docker` command inside a job step. This is necessary because as the steps are running inside a docker image, `setup_remote_docker` allocates another machine to run the commands there.
- The first `run` step builds the image, as you did previously locally. For that you are using the environment variable you declared in `environment:`, `IMAGE_NAME` (remember to change the highlighted section with your own information).
- The last `run` step pushes the image to Dockerhub, using the environment variables you configured on the project settings to authenticate.

Commit the new file to your repository and push the changes upstream:

```
cd ~/do-sample-app/
git add .circleci/
git commit -m "add CircleCI config"
git push
```

This will trigger a new build on CircleCI. The CircleCI workflow is going to correctly build and push your image to Docker Hub.



```
$ #!/bin/bash -eo pipefail                                                    Exit code: 0
  echo "$DOCKERHUB_PASS" | docker login -u "$DOCKERHUB_USERNAME" --password-stdin
  docker push $IMAGE_NAME:latest

WARNING! Your password will be stored unencrypted in /home/circleci/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
The push refers to a repository [docker.io/jonathancardoso/do-kubernetes-sample-app]
```

CircleCI build page with success build info

Now that you have created and tested your CircleCI workflow, you can set your DOKS cluster to retrieve the up-to-date image from Docker Hub and deploy it automatically when changes are made.

Step 8 — Updating the Deployment on the Kubernetes Cluster

Now that your application image is being built and sent to Docker Hub every time you push changes to the `master` branch on GitHub, it's time to update your deployment on your Kubernetes cluster so that it retrieves the new image and uses it as a base for deployment.

To do that, first fix one issue with your deployment: it's currently depending on an image with the `latest` tag. This tag does not tell us which version of the image you are using. You cannot easily lock your deployment to that tag because it’s overwritten everytime you push a new image to Docker Hub, and by using it like that you lose one of the best things about having containerized applications: Reproducibility.

You can read more about that on this article about why depending on Docker latest tag is a anti-pattern.

To correct this, you first must make some changes to your `Push Docker Image` build step in the `~/do-sample-app/.circleci/config.yml` file. Open up the file:

```
nano ~/do-sample-app/.circleci/config.yml
```

Then add the highlighted lines to your `Push Docker Image` step:

~/do-sample-app/.circleci/config.yml:16-22

```
...
      - run:
          name: Push Docker Image
          command: |
            echo "$DOCKERHUB_PASS" | docker login
-u "$DOCKERHUB_USERNAME" --password-stdin
            docker tag $IMAGE_NAME:latest
$IMAGE_NAME:$CIRCLE_SHA1
            docker push $IMAGE_NAME:latest
            docker push $IMAGE_NAME:$CIRCLE_SHA1
...
```

Save and exit the file.

CircleCI has some special environment variables set by default. One of them is `CIRCLE_SHA1`, which contains the hash of the commit it’s building. The changes you made to `~/do-sample-`

app/.circleci/config.yml will use this environment variable to tag your image with the commit it was built from, always tagging the most recent build with the latest tag. That way, you always have specific images available, without overwriting them when you push something new to your repository.

Next, change your deployment manifest file to point to that file. This would be simple if inside ~/do-sample-app/kube/do-sample-deployment.yml you could set your image as **dockerhub-username**/do-kubernetes-sample-app:$COMMIT_SHA1, but kubectl doesn't do variable substitution inside the manifests when you use kubectl apply. To account for this, you can use envsubst. envsubst is a cli tool, part of the GNU gettext project. It allows you to pass some text to it, and if it finds any variable inside the text that has a matching environment variable, it's replaced by the respective value. The resulting text is then returned as their output.

To use this, you will create a simple bash script which will be responsible for your deployment. Make a new folder called scripts inside ~/do-sample-app/:

```
mkdir ~/do-sample-app/scripts/
```

Inside that folder create a new bash script called ci-deploy.sh and open it with your favorite text editor:

```
nano ~/do-sample-app/scripts/ci-deploy.sh
```

Inside it write the following bash script:

~/do-sample-app/scripts/ci-deploy.sh

```
#! /bin/bash
# exit script when any command ran here returns
```

```
with non-zero exit code
set -e

COMMIT_SHA1=$CIRCLE_SHA1

# We must export it so it's available for envsubst
export COMMIT_SHA1=$COMMIT_SHA1

# since the only way for envsubst to work on files
is using input/output redirection,
#  it's not possible to do in-place substitution,
so we need to save the output to another file
#  and overwrite the original with that one.
envsubst <./kube/do-sample-deployment.yml
>./kube/do-sample-deployment.yml.out
mv ./kube/do-sample-deployment.yml.out ./kube/do-
sample-deployment.yml

echo "$KUBERNETES_CLUSTER_CERTIFICATE" | base64 --
decode > cert.crt

./kubectl \
  --kubeconfig=/dev/null \
  --server=$KUBERNETES_SERVER \
  --certificate-authority=cert.crt \
  --token=$KUBERNETES_TOKEN \
  apply -f ./kube/
```

Let's go through this script, using the comments in the file. First, there is the following:

```
set -e
```

This line makes sure any failed command stops the execution of the bash script. That way if one command fails, the next ones are not executed.

```
COMMIT_SHA1=$CIRCLE_SHA1
export COMMIT_SHA1=$COMMIT_SHA1
```

These lines export the CircleCI `$CIRCLE_SHA1` environment variable with a new name. If you had just declared the variable without exporting it using `export`, it would not be visible for the `envsubst` command.

```
envsubst <./kube/do-sample-deployment.yml
>./kube/do-sample-deployment.yml.out
mv ./kube/do-sample-deployment.yml.out ./kube/do-
sample-deployment.yml
```

`envsubst` cannot do in-place substitution. That is, it cannot read the content of a file, replace the variables with their respective values, and write the output back to the same file. Therefore, you will redirect the output to another file and then overwrite the original file with the new one.

```
echo "$KUBERNETES_CLUSTER_CERTIFICATE" | base64 --
decode > cert.crt
```

The environment variable `$KUBERNETES_CLUSTER_CERTIFICATE` you created earlier on CircleCI's project settings is in reality a Base64 encoded string. To use it with `kubectl` you must decode its contents and save it to a file. In this case you are saving it to a file named `cert.crt` inside the current working directory.

```
./kubectl \
  --kubeconfig=/dev/null \
  --server=$KUBERNETES_SERVER \
  --certificate-authority=cert.crt \
  --token=$KUBERNETES_TOKEN \
  apply -f ./kube/
```

Finally, you are running `kubectl`. The command has similar arguments to the one you ran when you were testing your Service Account. You are calling `apply -f ./kube/`, since on CircleCI the current working directory is the root folder of your project. `./kube/` here is your `~/do-sample-app/kube` folder.

Save the file and make sure it's executable:

```
chmod +x ~/do-sample-app/scripts/ci-deploy.sh
```

Now, edit `~/do-sample-app/kube/do-sample-deployment.yml`:

```
nano ~/do-sample-app/kube/do-sample-deployment.yml
```

Change the tag of the container image value to look like the following one:

~/do-sample-app/kube/do-sample-deployment.yml

```
      # ...
      containers:
        - name: do-kubernetes-sample-app
          image: dockerhub-username/do-kubernetes-
sample-app:$COMMIT_SHA1
          ports:
```

```
          - containerPort: 80
            name: http
```

Save and close the file. You must now add some new steps to your CI configuration file to update the deployment on Kubernetes.

Open `~/do-sample-app/.circleci/config.yml` on your favorite text editor:

```
nano ~/do-sample-app/.circleci/config.yml
```

Write the following new steps, right below the `Push Docker Image` one you had before:

~/do-sample-app/.circleci/config.yml

```
...
      - run:
          name: Install envsubst
          command: |
            sudo apt-get update && sudo apt-get -y
install gettext-base
      - run:
          name: Install kubectl
          command: |
            curl -LO
https://storage.googleapis.com/kubernetes-
release/release/$(curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/linux/amd64/kubect
l
            chmod u+x ./kubectl
      - run:
          name: Deploy Code
          command: ./scripts/ci-deploy.sh
```

The first two steps are installing some dependencies, first `envsubst`, and then `kubectl`. The `Deploy Code` step is responsible for running our deploy script.

To make sure the changes are really going to be reflected on your Kubernetes deployment, edit your `index.html`. Change the HTML to something else, like:

~/do-sample-app/index.html

```
<!DOCTYPE html>
<title>DigitalOcean</title>
<body>
  Automatic Deployment is Working!
</body>
```

Once you have saved the above change, commit all the modified files to the repository, and push the changes upstream:

```
cd ~/do-sample-app/
git add --all
git commit -m "add deploy script and add new steps
to circleci config"
git push
```

You will see the new build running on CircleCI, and successfully deploying the changes to your Kubernetes cluster.

Wait for the build to finish, then run the same command you ran previously:

```
kubectl port-forward $(kubectl get pod --
selector="app=do-kubernetes-sample-app" --output
jsonpath='{.items[0].metadata.name}') 8080:80
```

Make sure everything is working by opening your browser on the URL `localhost:8080` or by making a `curl` request to it. It should show the updated HTML:

```
curl localhost:8080
```

You will receive the following output:

Output

```
<!DOCTYPE html>
<title>DigitalOcean</title>
<body>
  Automatic Deployment is Working!
</body>
```

Congratulations, you have set up automated deployment with CircleCI!

Conclusion

This was a basic tutorial on how to do deployments to DigitalOcean Kubernetes using CircleCI. From here, you can improve your pipeline in many ways. The first thing you can do is create a single `build` job for multiple deployments, each one deploying to different Kubernetes clusters or different namespaces. This can be extremely useful when you have different Git branches for development/staging/production environments, ensuring that the deployments are always separated.

You could also build your own image to be used on CircleCI, instead of using `buildpack-deps`. This image could be based on it, but could already have `kubectl` and `envsubst` dependencies installed.

If you would like to learn more about CI/CD on Kubernetes, check out the tutorials for our CI/CD on Kubernetes Webinar Series, or for more information about apps on Kubernetes, see Modernizing Applications for Kubernetes.

How To Set Up a CD Pipeline with Spinnaker on DigitalOcean Kubernetes

Written by Savic

In this tutorial, you'll deploy Spinnaker, an open-source resource management and continuous delivery application, to your Kubernetes cluster. Spinnaker enables automated application deployments to many platforms and can integrate with other DevOps tools, like Jenkins and TravisCI. Additionally, it can be configured to monitor code repositories and Docker registries for completely automated Continuous Delivery development and deployment processes.

By the end of this tutorial you will be able to manage applications and development processes on your Kubernetes cluster using Spinnaker. You will automate the start of your deployment pipelines using triggers, such as, when a new Docker image has been added to your private registry, or when new code is pushed to a git repository.

The author selected the Free and Open Source Fund to receive a donation as part of the Write for DOnations program.

Spinnaker is an open-source resource management and continuous delivery application for fast, safe, and repeatable deployments, using a powerful and customizable pipeline system. Spinnaker allows for automated application deployments to many platforms, including DigitalOcean Kubernetes. When deploying, you can configure Spinnaker to use built-in deployment strategies, such as Highlander and Red/black, with the option of creating your own deployment strategy. It can integrate with other DevOps tools, like Jenkins and TravisCI, and can be configured to monitor GitHub repositories and Docker registries.

Spinnaker is managed by Halyard, a tool specifically built for configuring and deploying Spinnaker to various platforms. Spinnaker requires external storage for persisting your application's settings and pipelines. It supports different platforms for this task, like DigitalOcean Spaces.

In this tutorial, you'll deploy Spinnaker to DigitalOcean Kubernetes using Halyard, with DigitalOcean Spaces as the underlying back-end storage. You'll also configure Spinnaker to be available at your desired domain, secured using Let's Encrypt TLS certificates. Then, you will create a sample application in Spinnaker, create a pipeline, and deploy a `Hello World` app to your Kubernetes cluster. After testing it, you'll introduce authentication and authorization via GitHub Organizations. By the end, you will have a secured and working Spinnaker deployment in your Kubernetes cluster.

Note: This tutorial has been specifically tested with Spinnaker `1.13.5`.

Prerequisites

- Halyard installed on your local machine, according to the official instructions. Please note that using Halyard on Ubuntu versions higher than 16.04 is not supported. In such cases, you can use it via Docker.
- A DigitalOcean Kubernetes cluster with your connection configured as the `kubectl` default. The cluster must have at least 8GB RAM and 4 CPU cores available for Spinnaker (more will be required in the

case of heavier use). Instructions on how to configure `kubectl` are shown under the Connect to your Cluster step shown when you create your cluster. To create a Kubernetes cluster on DigitalOcean, see the Kubernetes Quickstart.

- An Nginx Ingress Controller and cert-manager installed on the cluster. For a guide on how to do this, see How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes.
- A DigitalOcean Space with API keys (access and secret). To create a DigitalOcean Space and API keys, see How To Create a DigitalOcean Space and API Key.
- A domain name with three DNS A records pointed to the DigitalOcean Load Balancer used by the Ingress. If you're using DigitalOcean to manage your domain's DNS records, consult How to Create DNS Records to create A records. In this tutorial, we'll refer to the A records as **`spinnaker.example.com`**, **`spinnaker-api.example.com`**, and **`hello-world.example.com`**.
- A GitHub account, added to a GitHub Organization with admin permissions and public visibility. The account must also be a member of a Team in the Organization. This is required to complete Step 5.

Step 1 — Adding a Kubernetes Account with Halyard

In this section, you will add a Kubernetes account to Spinnaker via Halyard. An account, in Spinnaker's terms, is a named credential it uses to access a cloud provider.

As part of the prerequisite, you created the `echo1` and `echo2` services and an `echo_ingress` ingress for testing purposes; you will not need these in this tutorial, so you can now delete them.

Start off by deleting the ingress by running the following command:

```
kubectl delete -f echo_ingress.yaml
```

Then, delete the two test services:

```
kubectl delete -f echo1.yaml && kubectl delete -f
echo2.yaml
```

The `kubectl delete` command accepts the file to delete when passed the `-f` parameter.

Next, from your local machine, create a folder that will serve as your workspace:

```
mkdir ~/spinnaker-k8s
```

Navigate to your workspace by running the following command:

```
cd ~/spinnaker-k8s
```

Halyard does not yet know where it should deploy Spinnaker. Enable the Kubernetes provider with this command:

```
hal config provider kubernetes enable
```

You'll receive the following output:

Output

```
+ Get current deployment
  Success
+ Edit the kubernetes provider
  Success
Problems in default.provider.kubernetes:
- WARNING Provider kubernetes is enabled, but no
accounts have been
  configured.
```

```
+ Successfully enabled kubernetes
```

Halyard logged all the steps it took to enable the Kubernetes provider, and warned that no accounts are defined yet.

Next, you'll create a Kubernetes service account for Spinnaker, along with RBAC. A service account is a type of account that is scoped to a single namespace. It is used by software, which may perform various tasks in the cluster. RBAC (Role Based Access Control) is a method of regulating access to resources in a Kubernetes cluster. It limits the scope of action of the account to ensure that no important configurations are inadvertently changed on your cluster.

Here, you will grant Spinnaker `cluster-admin` permissions to allow it to control the whole cluster. If you wish to create a more restrictive environment, consult the official Kubernetes documentation on RBAC.

First, create the `spinnaker` namespace by running the following command:

```
kubectl create ns spinnaker
```

The output will look like:

Output

```
namespace/spinnaker created
```

Run the following command to create a service account named **`spinnaker-service-account`**:

```
kubectl create serviceaccount spinnaker-service-
account -n spinnaker
```

You've used the `-n` flag to specify that `kubectl` create the service account in the `spinnaker` namespace. The output will be:

Output

```
serviceaccount/spinnaker-service-account created
```

Then, bind it to the `cluster-admin` role:

```
kubectl create clusterrolebinding spinnaker-service-account --clusterrole cluster-admin --serviceaccount=spinnaker:spinnaker-service-account
```

You will see the following output:

Output

```
clusterrolebinding.rbac.authorization.k8s.io/spinnaker-service-account created
```

Halyard uses the local kubectl to access the cluster. You'll need to configure it to use the newly created service account before deploying Spinnaker. Kubernetes accounts authenticate using usernames and tokens. When a service account is created, Kubernetes makes a new secret and populates it with the account token. To retrieve the token for the **`spinnaker-service-account`**, you'll first need to get the name of the secret. You can fetch it into a console variable, named `TOKEN_SECRET`, by running:

```
TOKEN_SECRET=$(kubectl get serviceaccount -n spinnaker spinnaker-service-account -o jsonpath='{.secrets[0].name}')
```

This gets information about the **`spinnaker-service-account`** from the namespace `spinnaker`, and fetches the name of the first secret it contains by passing in a JSON path.

Fetch the contents of the secret into a variable named `TOKEN` by running:

```
TOKEN=$(kubectl get secret -n spinnaker
$TOKEN_SECRET -o jsonpath='{.data.token}' | base64
--decode)
```

You now have the token available in the environment variable `TOKEN`. Next, you'll need to set credentials for the service account in kubectl:

```
kubectl config set-credentials spinnaker-token-
user --token $TOKEN
```

You will see the following output:

Output

```
User "spinnaker-token-user" set.
```

Then, you'll need to set the user of the current context to the newly created **`spinnaker-token-user`** by running the following command:

```
kubectl config set-context --current --user
spinnaker-token-user
```

By setting the current user to **`spinnaker-token-user`**, kubectl is now configured to use the **`spinnaker-service-account`**, but Halyard does not know anything about that. Add an account to its Kubernetes provider by executing:

```
hal config provider kubernetes account add
spinnaker-account --provider-version v2
```

The output will look like this:

Output

```
+ Get current deployment
  Success
+ Add the spinnaker-account account
```

```
  Success
+ Successfully added account spinnaker-account for
provider
  kubernetes.
```

This commmand adds a Kubernetes account to Halyard, named `spinnaker-account`, and marks it as a service account.

Generally, Spinnaker can be deployed in two ways: distributed installation or local installation. Distributed installation is what you're completing in this tutorial—you're deploying it to the cloud. Local installation, on the other hand, means that Spinnaker will be downloaded and installed on the machine Halyard runs on. Because you're deploying Spinnaker to Kubernetes, you'll need to mark the deployment as `distributed`, like so:

```
hal config deploy edit --type distributed --
account-name spinnaker-account
```

Since your Spinnaker deployment will be building images, it is necessary to enable `artifacts` in Spinnaker. You can enable them by running the following command:

```
hal config features edit --artifacts true
```

Here you've enabled `artifacts` to allow Spinnaker to store more metadata about the objects it creates.

You've added a Kubernetes account to Spinnaker, via Halyard. You enabled the Kubernetes provider, configured RBAC roles, and added the current kubectl config to Spinnaker, thus adding an account to the provider. Now you'll set up your back-end storage.

Step 2 — Configuring the Space as the Underlying Storage

In this section, you will configure the Space as the underlying storage for the Spinnaker deployment. Spinnaker will use the Space to store its configuration and pipeline-related data.

To configure S3 storage in Halyard, run the following command:

```
hal config storage s3 edit --access-key-id
your_space_access_key --secret-access-key --
endpoint spaces_endpoint_with_region_prefix --
bucket space_name --no-validate
```

Remember to replace **your_space_access_key** with your Space access key and **spaces_endpoint_with_region_prefix** with the endpoint of your Space. This is usually **region-id**.digitaloceanspaces.com, where **region-id** is the region of your Space. You can replace **space_name** with the name of your Space. The --no-validate flag tells Halyard not to validate the settings given right away, because DigitalOcean Spaces validation is not supported.

Once you've run this command, Halyard will ask you for your secret access key. Enter it to continue and you'll then see the following output:

Output

```
+ Get current deployment
  Success
+ Get persistent store
  Success
+ Edit persistent store
  Success
+ Successfully edited persistent store "s3".
```

Now that you've configured `s3` storage, you'll ensure that your deployment will use this as its storage by running the following command:

```
hal config storage edit --type s3
```

The output will look like this:

Output

```
+ Get current deployment
  Success
+ Get persistent storage settings
  Success
+ Edit persistent storage settings
  Success
+ Successfully edited persistent storage.
```

You've set up your Space as the underlying storage that your instance of Spinnaker will use. Now you'll deploy Spinnaker to your Kubernetes cluster and expose it at your domains using the Nginx Ingress Controller.

Step 3 — Deploying Spinnaker to Your Cluster

In this section, you will deploy Spinnaker to your cluster using Halyard, and then expose its UI and API components at your domains using an Nginx Ingress. First, you'll configure your domain URLs: one for Spinnaker's user interface and one for the API component. Then you'll pick your desired version of Spinnaker and deploy it using Halyard. Finally you'll create an ingress and configure it as an Nginx controller.

First, you'll need to edit Spinnaker's UI and API URL config values in Halyard and set them to your desired domains. To set the API endpoint to your desired domain, run the following command:

```
hal config security api edit --override-base-url
https://spinnaker-api.example.com
```

The output will look like:

Output

```
+ Get current deployment
  Success
+ Get API security settings
  Success
+ Edit API security settings
  Success
...
```

To set the UI endpoint to your domain, which is where you will access Spinnaker, run:

```
hal config security ui edit --override-base-url
https://spinnaker.example.com
```

The output will look like:

Output

```
+ Get current deployment
  Success
+ Get UI security settings
  Success
+ Edit UI security settings
  Success
+ Successfully updated UI security settings.
```

Remember to replace **`spinnaker-api.example.com`** and **`spinnaker.example.com`** with your domains. These are the domains you have pointed to the Load Balancer that you created during the Nginx Ingress Controller prerequisite.

You've created and secured Spinnaker's Kubernetes account, configured your Space as its underlying storage, and set its UI and API endpoints to your domains. Now you can list the available Spinnaker versions:

```
hal version list
```

Your output will show a list of available versions. At the time of writing this article `1.13.5` was the latest version:

Output

```
+ Get current deployment
  Success
+ Get Spinnaker version
  Success
+ Get released versions
  Success
+ You are on version "", and the following are
available:
 - 1.11.12 (Cobra Kai):
   Changelog: https://gist.GitHub.com/spinnaker-
release/29a01fa17afe7c603e510e202a914161
   Published: Fri Apr 05 14:55:40 UTC 2019
   (Requires Halyard >= 1.11)
 - 1.12.9 (Unbreakable):
   Changelog: https://gist.GitHub.com/spinnaker-
```

```
release/7fa9145349d6beb2f22163977a94629e
   Published: Fri Apr 05 14:11:44 UTC 2019
   (Requires Halyard >= 1.11)
 - 1.13.5 (BirdBox):
   Changelog: https://gist.GitHub.com/spinnaker-
release/23af06bc73aa942c90f89b8e8c8bed3e
   Published: Mon Apr 22 14:32:29 UTC 2019
   (Requires Halyard >= 1.17)
```

To select a version to install, run the following command:

```
hal config version edit --version 1.13.5
```

It is recommended to always select the latest version, unless you encounter some kind of regression.

You will see the following output:

Output

```
+ Get current deployment
  Success
+ Edit Spinnaker version
  Success
+ Spinnaker has been configured to update/install
version "version".
  Deploy this version of Spinnaker with `hal
deploy apply`.
```

You have now fully configured Spinnaker's deployment. You'll deploy it with the following command:

```
hal deploy apply
```

This command could take a few minutes to finish.

The final output will look like this:

Output

```
+ Get current deployment
  Success
+ Prep deployment
  Success
+ Preparation complete... deploying Spinnaker
+ Get current deployment
  Success
+ Apply deployment
  Success
+ Deploy spin-redis
  Success
+ Deploy spin-clouddriver
  Success
+ Deploy spin-front50
  Success
+ Deploy spin-orca
  Success
+ Deploy spin-deck
  Success
+ Deploy spin-echo
  Success
+ Deploy spin-gate
  Success
+ Deploy spin-rosco
```

```
  Success
...
```

Halyard is showing you the deployment status of each of Spinnaker's microservices. Behind the scenes, it calls kubectl to install them.

Kubernetes will take some time—ten minutes on average—to bring all of the containers up, especially for the first time. You can watch the progress by running the following command:

```
kubectl get pods -n spinnaker -w
```

You've deployed Spinnaker to your Kubernetes cluster, but it can't be accessed beyond your cluster.

You'll be storing the ingress configuration in a file named **`spinnaker-ingress.yaml`**. Create it using your text editor:

```
nano spinnaker-ingress.yaml
```

Add the following lines:

spinnaker-ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: spinnaker-ingress
  namespace: spinnaker
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
spec:
  tls:
```

```
  - hosts:
    - spinnaker-api.example.com
    - spinnaker.example.com
    secretName: spinnaker
  rules:
  - host: spinnaker-api.example.com
    http:
      paths:
      - backend:
          serviceName: spin-gate
          servicePort: 8084
  - host: spinnaker.example.com
    http:
      paths:
      - backend:
          serviceName: spin-deck
          servicePort: 9000
```

Remember to replace **`spinnaker-api.example.com`** with your API domain, and **`spinnaker.example.com`** with your UI domain.

The configuration file defines an ingress called `spinnaker-ingress`. The annotations specify that the controller for this ingress will be the Nginx controller, and that the `letsencrypt-prod` cluster issuer will generate the TLS certificates, defined in the prerequisite tutorial.

Then, it specifies that TLS will secure the UI and API domains. It sets up routing by directing the API domain to the `spin-gate` service (Spinnaker's API containers), and the UI domain to the `spin-deck` service (Spinnaker's UI containers) at the appropriate ports `8084` and `9000`.

Save and close the file.

Create the Ingress in Kubernetes by running:

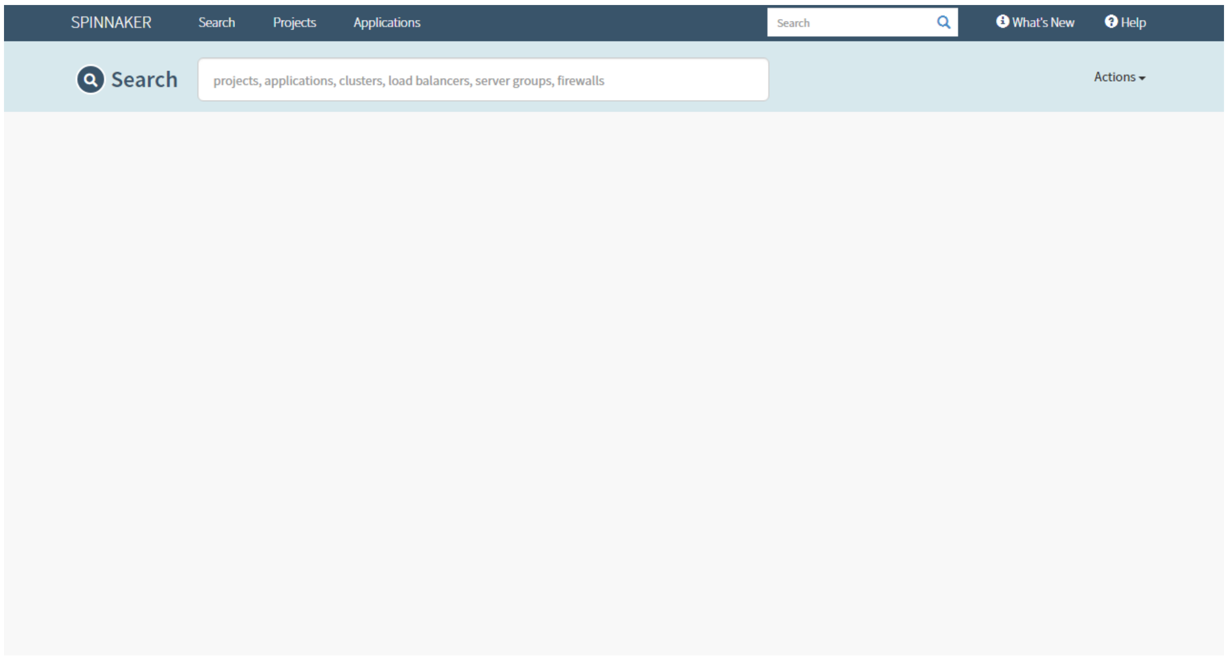
```
kubectl create -f spinnaker-ingress.yaml
```

You'll see the following output:

Output

```
ingress.extensions/spinnaker-ingress created
```

Wait a few minutes for Let's Encrypt to provision the TLS certificates, and then navigate to your UI domain, **`spinnaker.example.com`**, in a browser. You will see Spinnaker's user interface.


SPINNAKER
Search
Projects
Applications
Search
What's New
Help
Search
projects, applications, clusters, load balancers, server groups, firewalls
Actions

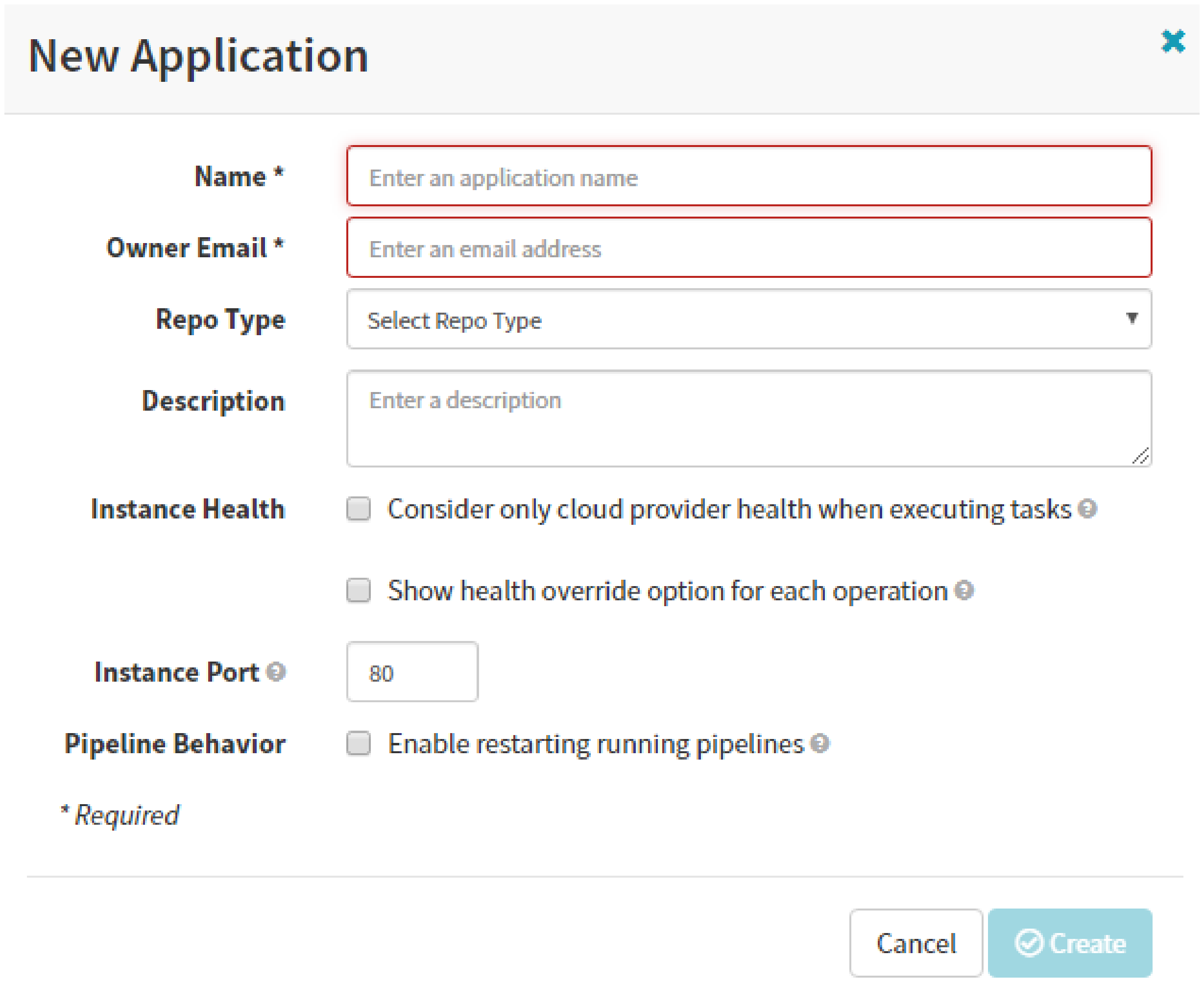
Spinnaker's home page

You've deployed Spinnaker to your cluster, exposed the UI and API components at your domains, and tested if it works. Now you'll create an application in Spinnaker and run a pipeline to deploy the `Hello World` app.

Step 4 — Creating an Application and Running a Pipeline

In this section, you will use your access to Spinnaker at your domain to create an application with it. You'll then create and run a pipeline to deploy a `Hello World` app, which can be found at [paulbouwer/hello-kubernetes](). You'll access the app afterward.

Navigate to your domain where you have exposed Spinnaker's UI. In the upper right corner, press on Actions, then select Create Application. You will see the New Application form.


New Application

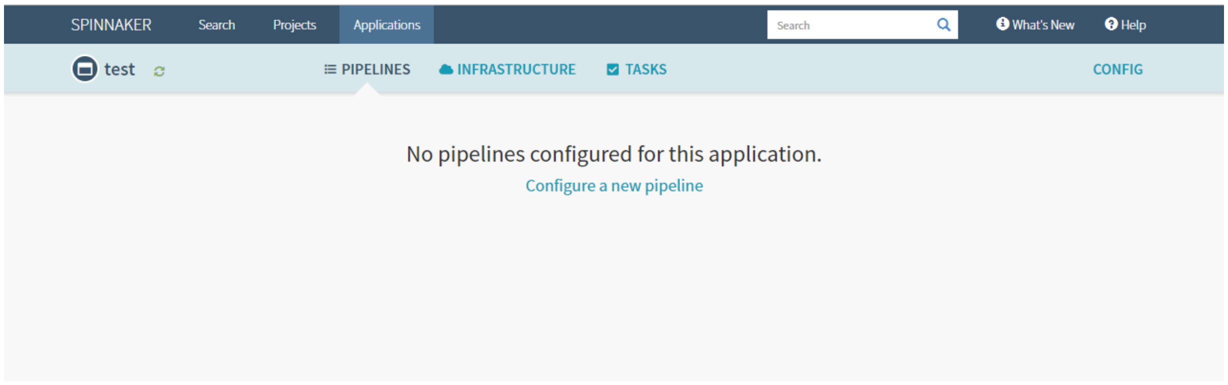
Name *	Enter an application name
Owner Email *	Enter an email address
Repo Type	Select Repo Type
Description	Enter a description
Instance Health	☐ Consider only cloud provider health when executing tasks
	☐ Show health override option for each operation
Instance Port	80
Pipeline Behavior	☐ Enable restarting running pipelines

** Required*

Creating a new Application in Spinnaker

Type in `hello-world` as the name, input your email address, and press Create.

When the page loads, navigate to Pipelines by clicking the first tab in the top menu. You will see that there are no pipelines defined yet.

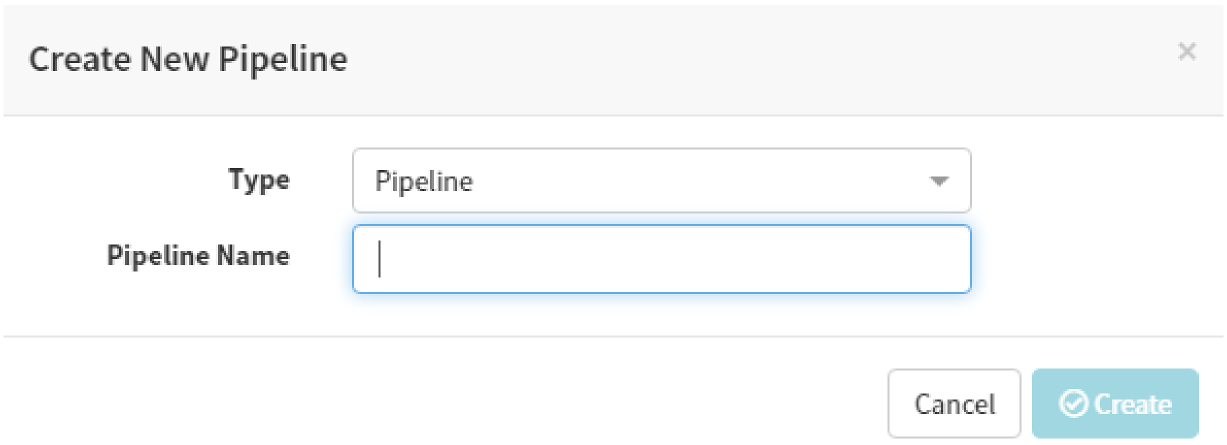


No pipelines configured for this application.

Configure a new pipeline

No pipelines defined in Spinnaker

Press on Configure a new pipeline and a new form will open.



Create New Pipeline

Type: Pipeline

Pipeline Name:

Creating a new Pipeline in Spinnaker

Fill in `Deploy Hello World Application` as your pipeline's name, and press Create.

On the next page, click the Add Stage button. As the Type, select Deploy (Manifest), which is used for deploying Kubernetes manifests you specify. For the Stage Name, type in `Deploy Hello World`. Scroll down, and in the textbox under Manifest Configuration, enter the following lines:

Manifest Configuration

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  namespace: spinnaker
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
spec:
  tls:
  - hosts:
    - hello-world.example.com
    secretName: hello-world
  rules:
  - host: hello-world.example.com
    http:
      paths:
      - backend:
          serviceName: hello-kubernetes
          servicePort: 80
---
apiVersion: v1
kind: Service
metadata:
```

```yaml
  name: hello-kubernetes
  namespace: spinnaker
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
  namespace: spinnaker
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello-kubernetes
  template:
    metadata:
      labels:
        app: hello-kubernetes
    spec:
      containers:
      - name: hello-kubernetes
```

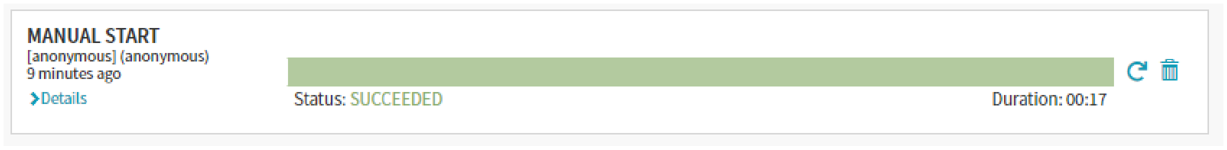
```
        image: paulbouwer/hello-kubernetes:1.5
        ports:
        - containerPort: 8080
```

Remember to replace **hello-world.example.com** with your domain, which is also pointed at your Load Balancer.

In this configuration, you define a `Deployment`, consisting of three replicas of the `paulbouwer/hello-kubernetes:1.5` image. You also define a `Service` to be able to access it and an Ingress to expose the `Service` at your domain.

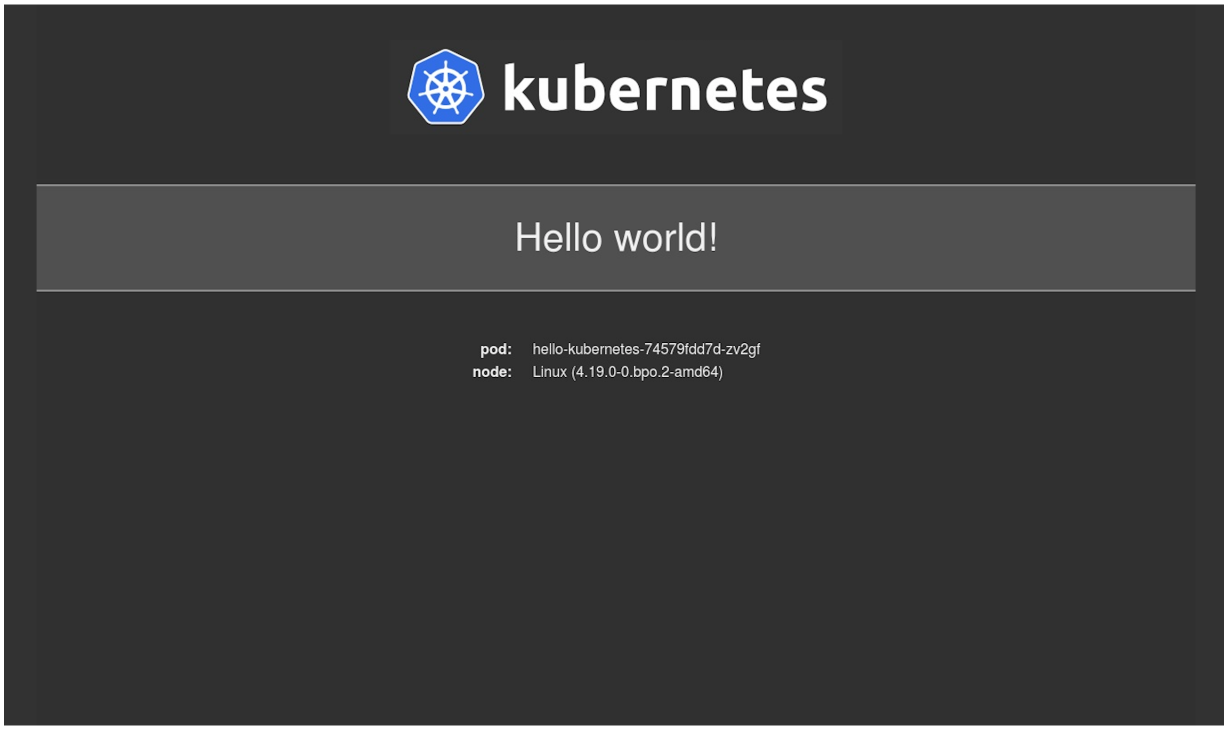
Press Save Changes in the bottom right corner of the screen. When it finishes, navigate back to Pipelines. On the right side, select the pipeline you just created and press the Start Manual Execution link. When asked to confirm, press Run.

This pipeline will take a short time to complete. You will see the progress bar complete when it has successfully finished.


MANUAL START
[anonymous] (anonymous)
9 minutes ago
Details
Status: SUCCEEDED
Duration: 00:17

Successfully ran a Pipeline

You can now navigate to the domain you defined in the configuration. You will see the `Hello World` app, which Spinnaker just deployed.



pod: hello-kubernetes-74579fdd7d-zv2gf
node: Linux (4.19.0-0.bpo.2-amd64)

Hello World App

You've created an application in Spinnaker, ran a pipeline to deploy a `Hello World` app, and accessed it. In the next step, you will secure Spinnaker by enabling GitHub Organizations authorization.

Step 5 — Enabling Role-Based Access with GitHub Organizations

In this section, you will enable GitHub OAuth authentication and GitHub Organizations authorization. Enabling GitHub OAuth authentication forces Spinnaker users to log in via GitHub, therefore preventing anonymous access. Authorization via GitHub Organizations restricts access only to those in an Organization. A GitHub Organization can contain Teams (named groups of members), which you will be able to use to restrict access to resources in Spinnaker even further.

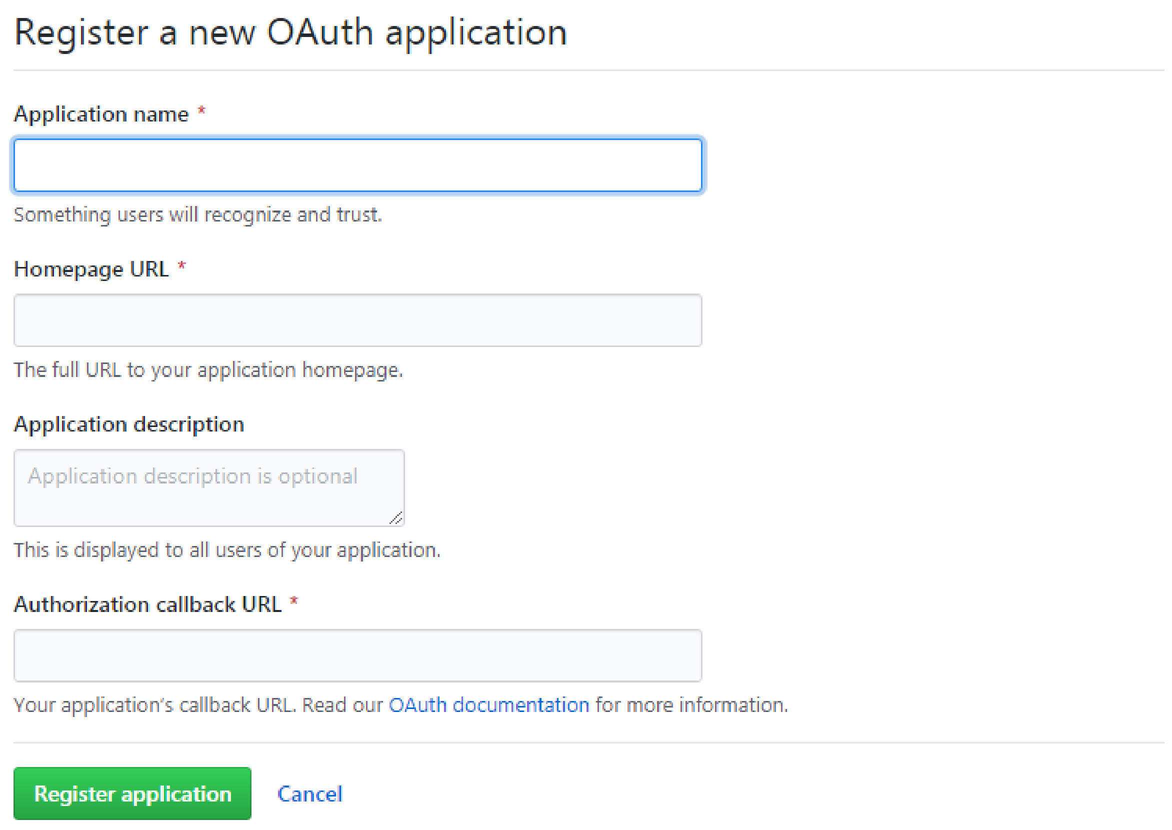
For OAuth authentication to work, you'll first need to set up the authorization callback URL, which is where the user will be redirected after authorization. This is your API domain ending with `/login`. You need to specify this manually to prevent Spinnaker and other services from guessing. To configure this, run the following command:

```
hal config security authn oauth2 edit --pre-
established-redirect-uri https://spinnaker-
api.example.com/login
```

You will see this output:

Output
```
+ Get current deployment
  Success
+ Get authentication settings
  Success
+ Edit oauth2 authentication settings
  Success
+ Successfully edited oauth2 method.
```

To set up OAuth authentication with GitHub, you'll need to create an OAuth application for your Organization. To do so, navigate to your Organization on GitHub, go to Settings, click on Developer Settings, and then select OAuth Apps from the left-hand menu. Afterward, click the New OAuth App button on the right. You will see the Register a new OAuth application form.



Register a new OAuth application

Application name *

Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

Application description is optional

This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our OAuth documentation for more information.

Creating a new OAuth App on GitHub

Enter **`spinnaker-auth`** as the name. For the Homepage URL, enter `https://`**`spinnaker.example.com`**, and for the Authorization callback URL, enter `https://`**`spinnaker-api.example.com`**`/login`. Then, press Register Application.

You'll be redirected to the settings page for your new OAuth app. Note the Client ID and Client Secret values—you'll need them for the next command.

With the OAuth app created, you can configure Spinnaker to use the OAuth app by running the following command:

```
hal config security authn oauth2 edit --client-id
client_id --client-secret client_secret --provider
GitHub
```

Remember to replace **`client_id`** and **`client_secret`** with the values shown on the GitHub settings page.

You output will be similar to the following:

Output

```
+ Get current deployment
  Success
+ Get authentication settings
  Success
+ Edit oauth2 authentication settings
  Success
Problems in default.security.authn:
- WARNING An authentication method is fully or
partially
  configured, but not enabled. It must be enabled
to take effect.

+ Successfully edited oauth2 method.
```

You've configured Spinnaker to use the OAuth app. Now, to enable it, execute:

```
hal config security authn oauth2 enable
```

The output will look like:

Output

```
+ Get current deployment
  Success
+ Edit oauth2 authentication settings
  Success
+ Successfully enabled oauth2
```

You’ve configured and enabled GitHub OAuth authentication. Now users will be forced to log in via GitHub in order to access Spinnaker. However, right now, everyone who has a GitHub account can log in, which is not what you want. To overcome this, you’ll configure Spinnaker to restrict access to members of your desired Organization.

You’ll need to set this up semi-manually via local config files, because Halyard does not yet have a command for setting this. During deployment, Halyard will use the local config files to override the generated configuration.

Halyard looks for custom configuration under `~/.hal/default/profiles/`. Files named **`service-name-*.yml`** are picked up by Halyard and used to override the settings of a particular service. The service that you’ll override is called `gate`, and serves as the API gateway for the whole of Spinnaker.

Create a file under `~/.hal/default/profiles/` named `gate-local.yml`:

```
nano ~/.hal/default/profiles/gate-local.yml
```

Add the following lines:

gate-local.yml

```
security:
 oauth2:
   providerRequirements:
     type: GitHub
     organization: your_organization_name
```

Replace **your_organization_name** with the name of your GitHub Organization. Save and close the file.

With this bit of configuration, only members of your GitHub Organization will be able to access Spinnaker.

Note: Only those members of your GitHub Organization whose membership is set to Public will be able to log in to Spinnaker. This setting can be changed on the member list page of your Organization.

Now, you'll integrate Spinnaker with an even more particular access-rule solution: GitHub Teams. This will enable you to specify which Team(s) will have access to resources created in Spinnaker, such as applications.

To achieve this, you'll need to have a GitHub Personal Access Token for an admin account in your Organization. To create one, visit Personal Access Tokens and press the Generate New Token button. On the next page, give it a description of your choice and be sure to check the read:org scope, located under admin:org. When you are done, press Generate token and note it down when it appears—you won't be able to see it again.

To configure GitHub Teams role authorization in Spinnaker, run the following command:

```
hal config security authz github edit --
accessToken access_token --organization
organization_name --baseUrl https://api.github.com
```

Be sure to replace **access_token** with your personal access token you generated and replace **organization_name** with the name of the Organization.

The output will be:

Output
```
+ Get current deployment
  Success
+ Get GitHub group membership settings
  Success
+ Edit GitHub group membership settings
  Success
+ Successfully edited GitHub method.
```

You've updated your GitHub group settings. Now, you'll set the authorization provider to GitHub by running the following command:

```
hal config security authz edit --type github
```

The output will look like:

Output
```
+ Get current deployment
  Success
+ Get group membership settings
  Success
+ Edit group membership settings
  Success
+ Successfully updated roles.
```

After updating these settings, enable them by running:

```
hal config security authz enable
```

You’ll see the following output:

Output

```
+ Get current deployment
  Success
+ Edit authorization settings
  Success
+ Successfully enabled authorization
```

With all the changes in place, you can now apply the changes to your running Spinnaker deployment. Execute the following command to do this:

```
hal deploy apply
```

Once it has finished, wait for Kubernetes to propagate the changes. This can take quite some time—you can watch the progress by running:

```
kubectl get pods -n spinnaker -w
```

When all the pods’ states become `Running` and availability `1/1`, navigate to your Spinnaker UI domain. You will be redirected to GitHub and asked to log in, if you’re not already. If the account you logged in with is a member of the Organization, you will be redirected back to Spinnaker and logged in. Otherwise, you will be denied access with a message that looks like this:

```
{"error":"Unauthorized", "message":"Authentication
```

The effect of GitHub Teams integration is that Spinnaker now translates them into roles. You can use these roles in Spinnaker to incorporate additional restrictions to access for members of particular teams. If you try to add another application, you'll notice that you can now also specify permissions, which combine the level of access—read only or read and write—with a role, for that application.

You've set up GitHub authentication and authorization. You have also configured Spinnaker to restrict access to members of your Organization, learned about roles and permissions, and considered the place of GitHub Teams when integrated with Spinnaker.

Conclusion

You have successfully configured and deployed Spinnaker to your DigitalOcean Kubernetes cluster. You can now manage and use your cloud resources more easily, from a central place. You can use triggers to automatically start a pipeline; for example, when a new Docker image has been added to the registry. To learn more about Spinnaker's terms and architecture, visit the official documentation. If you wish to deploy a private Docker registry to your cluster to hold your images, visit How To Set Up a Private Docker Registry on Top of DigitalOcean Spaces and Use It with DO Kubernetes.

Kubernetes Networking Under the Hood

Written by Brian Boucheron

This tutorial discusses how data moves inside a Pod, between Pods, and between Nodes. It also shows how a Kubernetes Service can provide a single static IP address and DNS entry for an application, easing communication with services that may be distributed among multiple constantly scaling and shifting Pods. This tutorial also includes detailed hop-by-hop explanations of the different journeys that packets can take depending on the network configuration.

Kubernetes is a powerful container orchestration system that can manage the deployment and operation of containerized applications across clusters of servers. In addition to coordinating container workloads, Kubernetes provides the infrastructure and tools necessary to maintain reliable network connectivity between your applications and services.

The Kubernetes cluster networking documentation states that the basic requirements of a Kubernetes network are:

- *all containers can communicate with all other containers without NAT*
- *all nodes can communicate with all containers (and vice-versa) without NAT*
- *the IP that a container sees itself as is the same IP that others see it as*

In this article we will discuss how Kubernetes satisfies these networking requirements within a cluster: how data moves inside a pod, between pods, and between nodes.

We will also show how a Kubernetes Service can provide a single static IP address and DNS entry for an application, easing communication with services that may be distributed among multiple constantly scaling and shifting pods.

If you are unfamiliar with the terminology of Kubernetes pods and nodes or other basics, our article An Introduction to Kubernetes covers the general architecture and components involved.

Let’s first take a look at the networking situation within a single pod.

Pod Networking

In Kubernetes, a pod is the most basic unit of organization: a group of tightly-coupled containers that are all closely related and perform a single function or service.

Networking-wise, Kubernetes treats pods similar to a traditional virtual machine or a single bare-metal host: each pod receives a single unique IP address, and all containers within the pod share that address and communicate with each other over the lo loopback interface using the localhost hostname. This is achieved by assigning all of the pod’s containers to the same network stack.

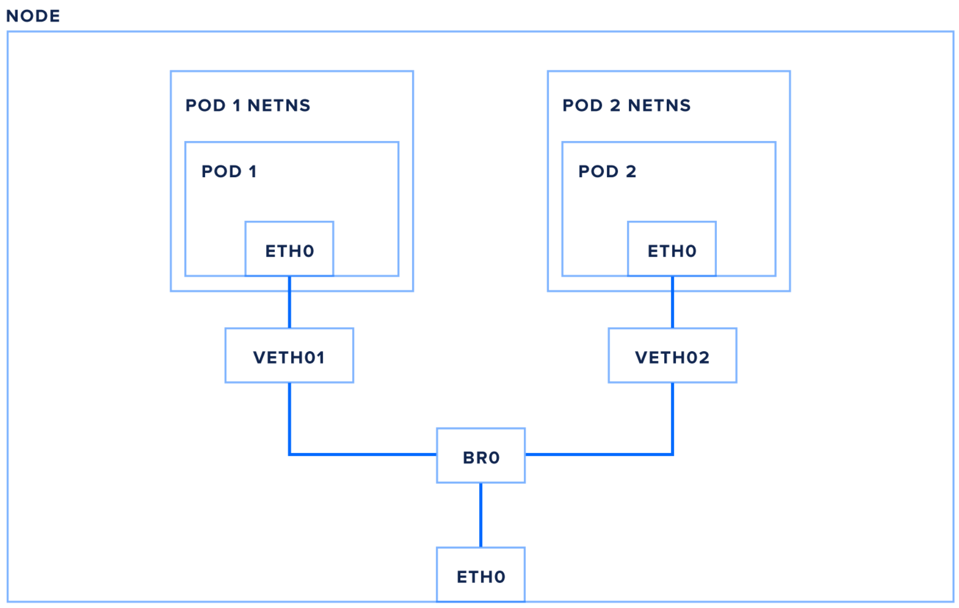
This situation should feel familiar to anybody who has deployed multiple services on a single host before the days of containerization. All the services need to use a unique port to listen on, but otherwise communication is uncomplicated and has low overhead.

Pod to Pod Networking

Most Kubernetes clusters will need to deploy multiple pods per node. Pod to pod communication may happen between two pods on the same node, or between two different nodes.

Pod to Pod Communication on One Node

On a single node you can have multiple pods that need to communicate directly with each other. Before we trace the route of a packet between pods, let's inspect the networking setup of a node. The following diagram provides an overview, which we will walk through in detail:



Networking overview of a single Kubernetes node

Each node has a network interface – eth0 in this example – attached to the Kubernetes cluster network. This interface sits within the node's root network namespace. This is the default namespace for networking devices on Linux.

Just as process namespaces enable containers to isolate running applications from each other, network namespaces isolate network devices such as interfaces and bridges. Each pod on a node is assigned its own isolated network namespace.

Pod namespaces are connected back to the root namespace with a virtual ethernet pair, essentially a pipe between the two namespaces with an interface on each end (here we're using veth1 in the root namespace, and eth0 within the pod).

Finally, the pods are connected to each other and to the node's eth0 interface via a bridge, br0 (your node may use something like cbr0 or docker0). A bridge essentially works like a physical ethernet switch, using either ARP (address resolution protocol) or IP-based routing to look up other local interfaces to direct traffic to.

Let's trace a packet from pod1 to pod2 now:

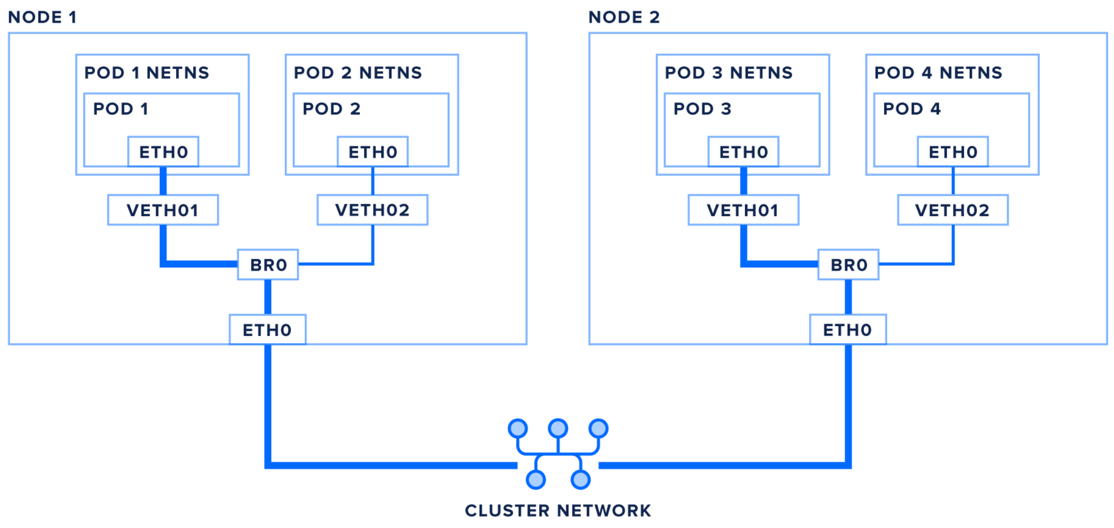
- pod1 creates a packet with pod2's IP as its destination
- The packet travels over the virtual ethernet pair to the root network namespace
- The packet continues to the bridge br0
- Because the destination pod is on the same node, the bridge sends the packet to pod2's virtual ethernet pair
- the packet travels through the virtual ethernet pair, into pod2's network namespace and the pod's eth0 network interface

Now that we've traced a packet from pod to pod within a node, let's look at how pod traffic travels between nodes.

Pod to Pod Communication Between Two Nodes

Because each pod in a cluster has a unique IP, and every pod can communicate directly with all other pods, a packet moving between pods on two different nodes is very similar to the previous scenario.

Let's trace a packet from pod1 to pod3, which is on a different node:



Networking diagram between two Kubernetes nodes

- pod1 creates a packet with pod3's IP as its destination
- The packet travels over the virtual ethernet pair to the root network namespace

- The packet continues to the bridge br0
- The bridge finds no local interface to route to, so the packet is sent out the default route toward eth0
- Optional: if your cluster requires a network overlay to properly route packets to nodes, the packet may be encapsulated in a VXLAN packet (or other network virtualization technique) before heading to the network. Alternately, the network itself may be set up with the proper static routes, in which case the packet travels to eth0 and out the the network unaltered.
- The packet enters the cluster network and is routed to the correct node.
- The packet enters the destination node on eth0
- Optional: if your packet was encapsulated, it will be de-encapsulated at this point
- The packet continues to the bridge br0
- The bridge routes the packet to the destination pod's virtual ethernet pair
- The packet passes through the virtual ethernet pair to the pod's eth0 interface

Now that we are familiar with how packets are routed via pod IP addresses, let's take a look at Kubernetes services and how they build on top of this infrastructure.

Pod to Service Networking

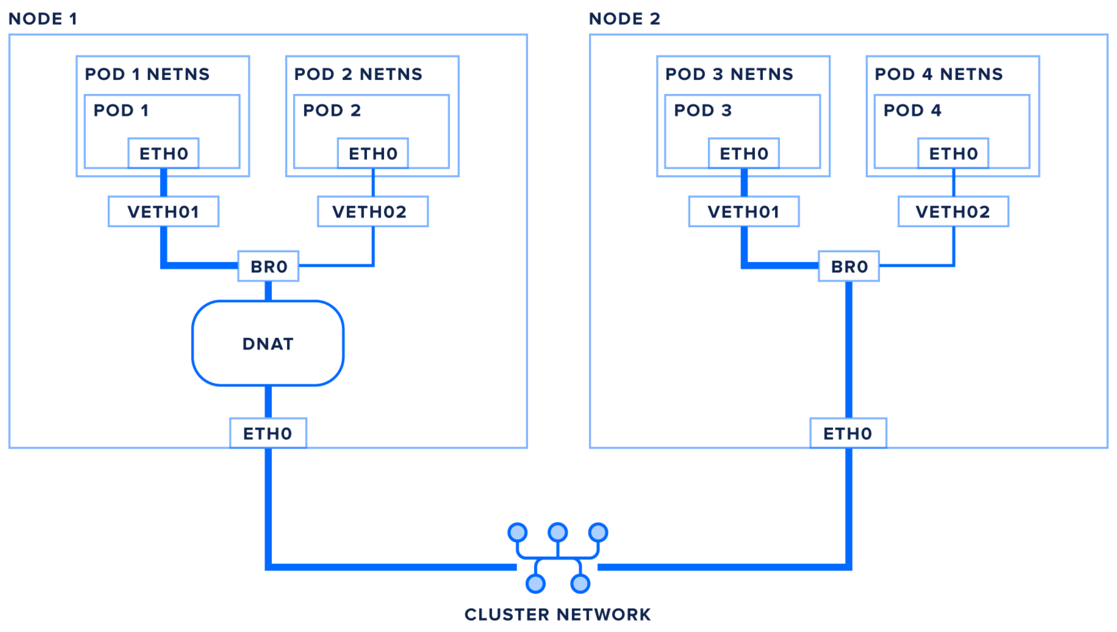
It would be difficult to send traffic to a particular application using just pod IPs, as the dynamic nature of a Kubernetes cluster means pods can be moved, restarted, upgraded, or scaled in and out of existence. Additionally, some services will have many replicas, so we need some way to load balance between them.

Kubernetes solves this problem with Services. A Service is an API object that maps a single virtual IP (VIP) to a set of pod IPs. Additionally, Kubernetes provides a DNS entry for each service's name and virtual IP, so services can be easily addressed by name.

The mapping of virtual IPs to pod IPs within the cluster is coordinated by the `kube-proxy` process on each node. This process sets up either iptables or IPVS to automatically translate VIPs into pod IPs before sending the packet out to the cluster network. Individual connections are tracked so packets can be properly de-translated when they return. IPVS and iptables can both do load balancing of a single service virtual IP into multiple pod IPs, though IPVS has much more flexibility in the load balancing algorithms it can use.

Note: this translation and connection tracking processes happens entirely in the Linux kernel. kube-proxy reads from the Kubernetes API and updates iptables ip IPVS, but it is not in the data path for individual packets. This is more efficient and higher performance than previous versions of kube-proxy, which functioned as a user-land proxy.

Let's follow the route a packet takes from a pod, pod1 again, to a service, service1:



Networking diagram between two Kubernetes nodes, showing DNAT translation of virtual IPs

- pod1 creates a packet with service1's IP as its destination
- The packet travels over the virtual ethernet pair to the root network namespace
- The packet continues to the bridge br0
- The bridge finds no local interface to route the packet to, so the packet is sent out the default route toward eth0
- Iptables or IPVS, set up by `kube-proxy`, match the packet's destination IP and translate it from a virtual IP to one of the service's

pod IPs, using whichever load balancing algorithms are available or specified

- Optional: your packet may be encapsulated at this point, as discussed in the previous section
- The packet enters the cluster network and is routed to the correct node.
- The packet enters the destination node on eth0
- Optional: if your packet was encapsulated, it will be de-encapsulated at this point
- The packet continues to the bridge br0
- The packet is sent to the virtual ethernet pair via veth1
- The packet passes through the virtual ethernet pair and enters the pod network namespace via its eth0 network interface

When the packet returns to node1 the VIP to pod IP translation will be reversed, and the packet will be back through the bridge and virtual interface to the correct pod.

Conclusion

In this article we've reviewed the internal networking infrastructure of a Kubernetes cluster. We've discussed the building blocks that make up the network, and detailed the hop-by-hop journey of packets in different scenarios.

For more information about Kubernetes, take a look at our Kubernetes tutorials tag and the official Kubernetes documentation.

How To Inspect Kubernetes Networking

Written by Brian Boucheron

Maintaining network connectivity between all the containers in a cluster requires some advanced networking techniques. Thankfully Kubernetes does all of the work to set up and maintain its internal networking. However, when things do not work as expected, tools like kubectl, Docker, nsenter, and iptables are invaluable for inspecting and troubleshooting a Kubernetes cluster's network setup. These tools are useful for debugging routing and connectivity issues, investigating network throughput problems, and generally exploring Kubernetes to learn how it operates.

Kubernetes is a container orchestration system that can manage containerized applications across a cluster of server nodes. Maintaining network connectivity between all the containers in a cluster requires some advanced networking techniques. In this article, we will briefly cover some tools and techniques for inspecting this networking setup.

These tools may be useful if you are debugging connectivity issues, investigating network throughput problems, or exploring Kubernetes to learn how it operates.

If you want to learn more about Kubernetes in general, our guide An Introduction to Kubernetes covers the basics. For a networking-specific overview of Kubernetes, please read Kubernetes Networking Under the Hood.

Getting Started

This tutorial will assume that you have a Kubernetes cluster, with `kubectl` installed locally and configured to connect to the cluster.

The following sections contain many commands that are intended to be run on a Kubernetes node. They will look like this:

```
echo 'this is a node command'
```

Commands that should be run on your local machine will have the following appearance:

```
echo 'this is a local command'
```

Note: Most of the commands in this tutorial will need to be run as the root user. If you instead use a sudo-enabled user on your Kubernetes nodes, please add `sudo` to run the commands when necessary.

Finding a Pod’s Cluster IP

To find the cluster IP address of a Kubernetes pod, use the `kubectl get pod` command on your local machine, with the option `-o wide`. This option will list more information, including the node the pod resides on, and the pod’s cluster IP.

```
kubectl get pod -o wide
```

Output

```
NAME                               READY     STATUS
RESTARTS   AGE       IP             NODE
hello-world-5b446dd74b-7c7pk   1/1       Running
0          22m       10.244.18.4   node-one
hello-world-5b446dd74b-pxtzt   1/1       Running
0          22m       10.244.3.4    node-two
```

The IP column will contain the internal cluster IP address for each pod.

If you don’t see the pod you’re looking for, make sure you’re in the right namespace. You can list all pods in all namespaces by adding the flag `--all-namespaces`.

Finding a Service’s IP

We can find a Service IP using `kubectl` as well. In this case we will list all services in all namespaces:

```
kubectl get service --all-namespaces
```

Output

```
NAMESPACE     NAME                          TYPE
CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
default       kubernetes                    ClusterIP
10.32.0.1       <none>        443/TCP          6d
kube-system   csi-attacher-doplugin         ClusterIP
10.32.159.128   <none>        12345/TCP        6d
kube-system   csi-provisioner-doplugin      ClusterIP
10.32.61.61     <none>        12345/TCP        6d
kube-system   kube-dns                      ClusterIP
10.32.0.10      <none>        53/UDP,53/TCP    6d
kube-system   kubernetes-dashboard          ClusterIP
10.32.226.209   <none>        443/TCP          6d
```

The service IP can be found in the CLUSTER-IP column.

Finding and Entering Pod Network Namespaces

Each Kubernetes pod gets assigned its own network namespace. Network namespaces (or netns) are a Linux networking primitive that provide isolation between network devices.

It can be useful to run commands from within a pod's netns, to check DNS resolution or general network connectivity. To do so, we first need to look up the process ID of one of the containers in a pod. For Docker, we can do that with a series of two commands. First, list the containers running on a node:

```
docker ps
```

Output

```
CONTAINER ID        IMAGE
COMMAND                  CREATED
STATUS              PORTS               NAMES
173ee46a3926        gcr.io/google-samples/node-
hello         "/bin/sh -c 'node se…"    9 days ago
Up 9 days                               k8s_hello-
world_hello-world-5b446dd74b-
pxtzt_default_386a9073-7e35-11e8-8a3d-
bae97d2c1afd_0
11ad51cb72df        k8s.gcr.io/pause-amd64:3.1
"/pause"                 9 days ago          Up 9
days                                   k8s_POD_hello-
world-5b446dd74b-pxtzt_default_386a9073-7e35-11e8-
8a3d-bae97d2c1afd_0
. . .
```

Find the container ID or name of any container in the pod you're interested in. In the above output we're showing two containers:

- The first container is the `hello-world` app running in the `hello-world` pod
- The second is a pause container running in the `hello-world` pod. This container exists solely to hold onto the pod's network namespace

To get the process ID of either container, take note of the container ID or name, and use it in the following `docker` command:

```
docker inspect --format '{{ .State.Pid }}'
container-id-or-name
```

Output

```
14552
```

A process ID (or PID) will be output. Now we can use the `nsenter` program to run a command in that process's network namespace:

```
nsenter -t your-container-pid -n ip addr
```

Be sure to use your own PID, and replace `ip addr` with the command you'd like to run inside the pod's network namespace.

Note: One advantage of using `nsenter` to run commands in a pod's namespace – versus using something like `docker exec` – is that you have access to all of the commands available on the node, instead of the typically limited set of commands installed in containers.

Finding a Pod's Virtual Ethernet Interface

Each pod's network namespace communicates with the node's root netns through a virtual ethernet pipe. On the node side, this pipe appears as a device that typically begins with `veth` and ends in a unique identifier, such as `veth77f2275` or `veth01`. Inside the pod this pipe appears as `eth0`.

It can be useful to correlate which `veth` device is paired with a particular pod. To do so, we will list all network devices on the node, then list the devices in the pod's network namespace. We can then correlate device numbers between the two listings to make the connection.

First, run `ip addr` in the pod's network namespace using `nsenter`. Refer to the previous section Finding and Entering Pod Network Namespaces for details on how to do this:

```
nsenter -t your-container-pid -n ip addr
```

Output

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc
noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:f4:03:04 brd
ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.3.4/24 brd 10.244.3.255 scope
global eth0
       valid_lft forever preferred_lft forever
```

The command will output a list of the pod's interfaces. Note the `if11` number after `eth0@` in the example output. This means this pod's `eth0` is linked to the node’s 11th interface. Now run `ip addr` in the node’s default namespace to list out its interfaces:

```
ip addr
```

Output

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc
noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

. . .

7: veth77f2275@if6:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue master docker0 state UP group default
    link/ether 26:05:99:58:0d:b9 brd
ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::2405:99ff:fe58:db9/64 scope link
       valid_lft forever preferred_lft forever
9: vethd36cef3@if8:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue master docker0 state UP group default
    link/ether ae:05:21:a2:9a:2b brd
```

```
ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::ac05:21ff:fea2:9a2b/64 scope link
       valid_lft forever preferred_lft forever
11: veth4f7342d@if10:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue master docker0 state UP group default
    link/ether e6:4d:7b:6f:56:4c brd
ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::e44d:7bff:fe6f:564c/64 scope link
       valid_lft forever preferred_lft forever
```

The 11th interface is `veth4f7342d` in this example output. This is the virtual ethernet pipe to the pod we're investigating.

Inspecting Conntrack Connection Tracking

Prior to version 1.11, Kubernetes used iptables NAT and the conntrack kernel module to track connections. To list all the connections currently being tracked, use the `conntrack` command:

```
conntrack -L
```

To watch continuously for new connections, use the `-E` flag:

```
conntrack -E
```

To list conntrack-tracked connections to a particular destination address, use the `-d` flag:

```
conntrack -L -d 10.32.0.1
```

If your nodes are having issues making reliable connections to services, it's possible your connection tracking table is full and new connections are being dropped. If that's the case you may see messages like the following in your system logs:

/var/log/syslog

```
Jul 12 15:32:11 worker-528 kernel: nf_conntrack:
table full, dropping packet.
```

There is a sysctl setting for the maximum number of connections to track. You can list out your current value with the following command:

```
sysctl net.netfilter.nf_conntrack_max
```

Output

```
net.netfilter.nf_conntrack_max = 131072
```

To set a new value, use the `-w` flag:

```
sysctl -w net.netfilter.nf_conntrack_max=198000
```

To make this setting permanent, add it to the `sysctl.conf` file:

/etc/sysctl.conf

```
. . .
net.ipv4.netfilter.ip_conntrack_max = 198000
```

Inspecting Iptables Rules

Prior to version 1.11, Kubernetes used iptables NAT to implement virtual IP translation and load balancing for Service IPs.

To dump all iptables rules on a node, use the `iptables-save` command:

```
iptables-save
```

Because the output can be lengthy, you may want to pipe to a file (`iptables-save > output.txt`) or a pager (`iptables-save | less`) to more easily review the rules.

To list just the Kubernetes Service NAT rules, use the `iptables` command and the `-L` flag to specify the correct chain:

```
iptables -t nat -L KUBE-SERVICES
```

Output

```
Chain KUBE-SERVICES (2 references)
target     prot opt source
destination
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere
10.32.0.10           /* kube-system/kube-dns:dns
cluster IP */ udp dpt:domain
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere
10.32.0.10           /* kube-system/kube-dns:dns-
tcp cluster IP */ tcp dpt:domain
KUBE-SVC-XGLOHA7QRQ3V22RZ  tcp  --  anywhere
10.32.226.209        /* kube-system/kubernetes-
dashboard: cluster IP */ tcp dpt:https
. . .
```

Querying Cluster DNS

One way to debug your cluster DNS resolution is to deploy a debug container with all the tools you need, then use `kubectl` to exec `nslookup` on it. This is described in the official Kubernetes documentation.

Another way to query the cluster DNS is using `dig` and `nsenter` from a node. If `dig` is not installed, it can be installed with `apt` on Debian-based Linux distributions:

```
apt install dnsutils
```

First, find the cluster IP of the kube-dns service:

```
kubectl get service -n kube-system kube-dns
```

Output

```
NAME         TYPE         CLUSTER-IP    EXTERNAL-IP
PORT(S)          AGE
kube-dns     ClusterIP    10.32.0.10    <none>
53/UDP,53/TCP    15d
```

The cluster IP is highlighted above. Next we'll use `nsenter` to run `dig` in the a container namespace. Look at the section Finding and Entering Pod Network Namespaces for more information on this:

```
nsenter -t 14346 -n dig
kubernetes.default.svc.cluster.local @10.32.0.10
```

This `dig` command looks up the Service's full domain name of **service-name.namespace**.svc.cluster.local and specifics the IP of the cluster DNS service IP (`@10.32.0.10`).

Looking at IPVS Details

As of Kubernetes 1.11, `kube-proxy` can configure IPVS to handle the translation of virtual Service IPs to pod IPs. You can list the translation table of IPs with `ipvsadm`:

```
ipvsadm -Ln
```

Output

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight
ActiveConn InActConn
TCP  100.64.0.1:443 rr
  -> 178.128.226.86:443           Masq    1      0
0
TCP  100.64.0.10:53 rr
  -> 100.96.1.3:53                Masq    1      0
0
  -> 100.96.1.4:53                Masq    1      0
0
UDP  100.64.0.10:53 rr
  -> 100.96.1.3:53                Masq    1      0
0
  -> 100.96.1.4:53                Masq    1      0
0
```

To show a single Service IP, use the `-t` option and specify the desired IP:

```
ipvsadm -Ln -t 100.64.0.10:53
```

Output

```
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight
ActiveConn InActConn
TCP  100.64.0.10:53 rr
```

```
  -> 100.96.1.3:53                Masq    1      0
0
  -> 100.96.1.4:53                Masq    1      0
0
```

Conclusion

In this article we've reviewed some commands and techniques for exploring and inspecting the details of your Kubernetes cluster's networking. For more information about Kubernetes, take a look at our Kubernetes tutorials tag and the official Kubernetes documentation.

An Introduction to Service Meshes

Written by Kathleen Juell

A service mesh is an infrastructure layer that allows you to manage communication between your application’s microservices. Service meshes are designed to facilitate service-to-service communication through service discovery, routing and internal load balancing, traffic configuration, encryption, authentication and authorization, and metrics and monitoring.

This tutorial will use Istio’s Bookinfo sample application — four microservices that together display information about particular books — as a concrete example to illustrate how service meshes work.

A service mesh is an infrastructure layer that allows you to manage communication between your application’s microservices. As more developers work with microservices, service meshes have evolved to make that work easier and more effective by consolidating common management and administrative tasks in a distributed setup.

Taking a microservice approach to application architecture involves breaking your application into a collection of loosely-coupled services. This approach offers certain benefits: teams can iterate designs and scale quickly, using a wider range of tools and languages. On the other hand, microservices pose new challenges for operational complexity, data consistency, and security.

Service meshes are designed to address some of these challenges by offering a granular level of control over how services communicate with one another. Specifically, they offer developers a way to manage:

- Service discovery
- Routing and traffic configuration
- Encryption and authentication/authorization
- Metrics and monitoring

Though it is possible to do these tasks natively with container orchestrators like Kubernetes, this approach involves a greater amount of up-front decision-making and administration when compared to what service mesh solutions like Istio and Linkerd offer out of the box. In this sense, service meshes can streamline and simplify the process of working with common components in a microservice architecture. In some cases they can even extend the functionality of these components.

Why Services Meshes?

Service meshes are designed to address some of the challenges inherent to distributed application architectures.

These architectures grew out of the three-tier application model, which broke applications into a web tier, application tier, and database tier. At scale, this model has proved challenging to organizations experiencing rapid growth. Monolithic application code bases can grow to be unwieldy “big balls of mud”, posing challenges for development and deployment.

In response to this problem, organizations like Google, Netflix, and Twitter developed internal “fat client” libraries to standardize runtime operations across services. These libraries provided load balancing, circuit breaking, routing, and telemetry — precursors to service mesh capabilities. However, they also imposed limitations on the languages developers could use and required changes across services when they themselves were updated or changed.

A microservice design avoids some of these issues. Instead of having a large, centralized application codebase, you have a collection of discretely managed services that represent a feature of your application. Benefits of a microservice approach include: - Greater agility in development and deployment, since teams can work on and deploy different application features independently. - Better options for CI/CD, since individual microservices can be tested and redeployed independently. - More options for languages and tools. Developers can use the best tools for the tasks at hand, rather than being restricted to a given language or toolset. - Ease in scaling. - Improvements in uptime, user experience, and stability.

At the same time, microservices have also created challenges: - Distributed systems require different ways of thinking about latency, routing, asynchronous workflows, and failures. - Microservice setups cannot necessarily meet the same requirements for data consistency as monolithic setups. - Greater levels of distribution necessitate more complex operational designs, particularly when it comes to service-to-service communication. - Distribution of services increases the surface area for security vulnerabilities.

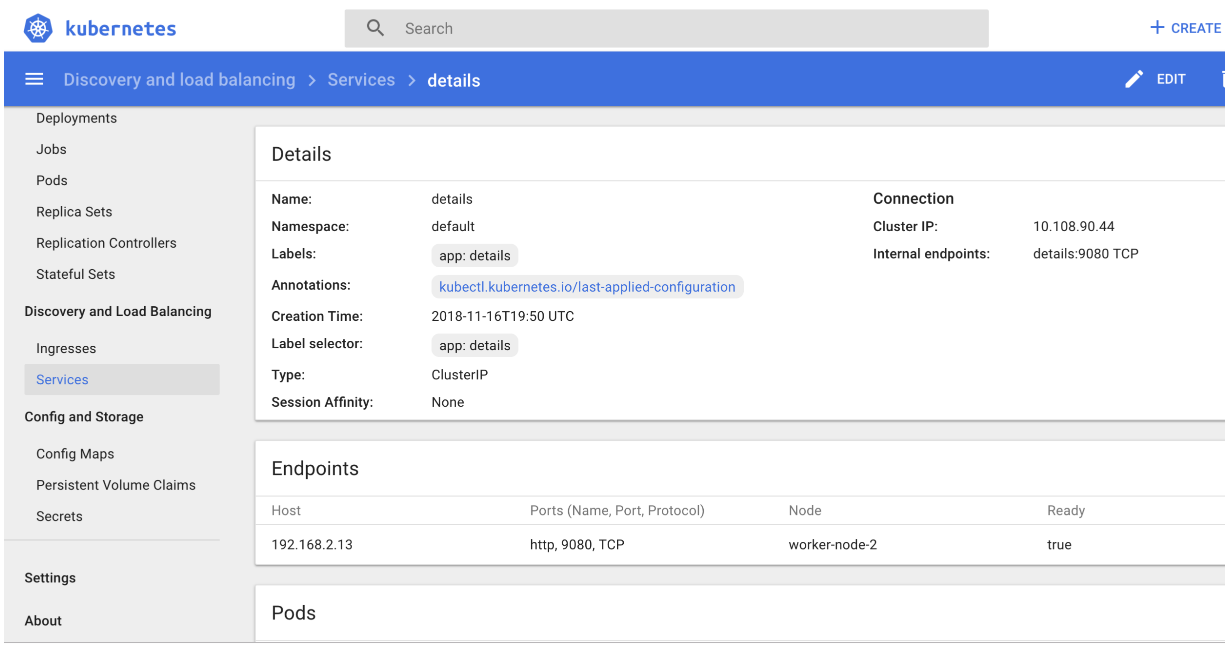
Service meshes are designed to address these issues by offering coordinated and granular control over how services communicate. In the sections that follow, we'll look at how service meshes facilitate service-to-service communication through service discovery, routing and internal load balancing, traffic configuration, encryption, authentication and authorization, and metrics and monitoring. We will use Istio's Bookinfo sample application — four microservices that together display information about particular books — as a concrete example to illustrate how service meshes work.

Service Discovery

In a distributed framework, it's necessary to know how to connect to services and whether or not they are available. Service instance locations are assigned dynamically on the network and information about them is constantly changing as containers are created and destroyed through autoscaling, upgrades, and failures.

Historically, there have been a few tools for doing service discovery in a microservice framework. Key-value stores like etcd were paired with other tools like Registrator to offer service discovery solutions. Tools like Consul iterated on this by combining a key-value store with a DNS interface that allows users to work directly with their DNS server or node.

Taking a similar approach, Kubernetes offers DNS-based service discovery by default. With it, you can look up services and service ports, and do reverse IP lookups using common DNS naming conventions. In general, an A record for a Kubernetes service matches this pattern: **`service`**`.`**`namespace`**`.svc.cluster.local`. Let's look at how this works in the context of the Bookinfo application. If, for example, you wanted information on the `details` service from the Bookinfo app, you could look at the relevant entry in the Kubernetes dashboard:



Details

Name:	details
Namespace:	default
Labels:	app: details
Annotations:	kubectl.kubernetes.io/last-applied-configuration
Creation Time:	2018-11-16T19:50 UTC
Label selector:	app: details
Type:	ClusterIP
Session Affinity:	None

Connection

Cluster IP:	10.108.90.44
Internal endpoints:	details:9080 TCP

Endpoints

Host	Ports (Name, Port, Protocol)	Node	Ready
192.168.2.13	http, 9080, TCP	worker-node-2	true

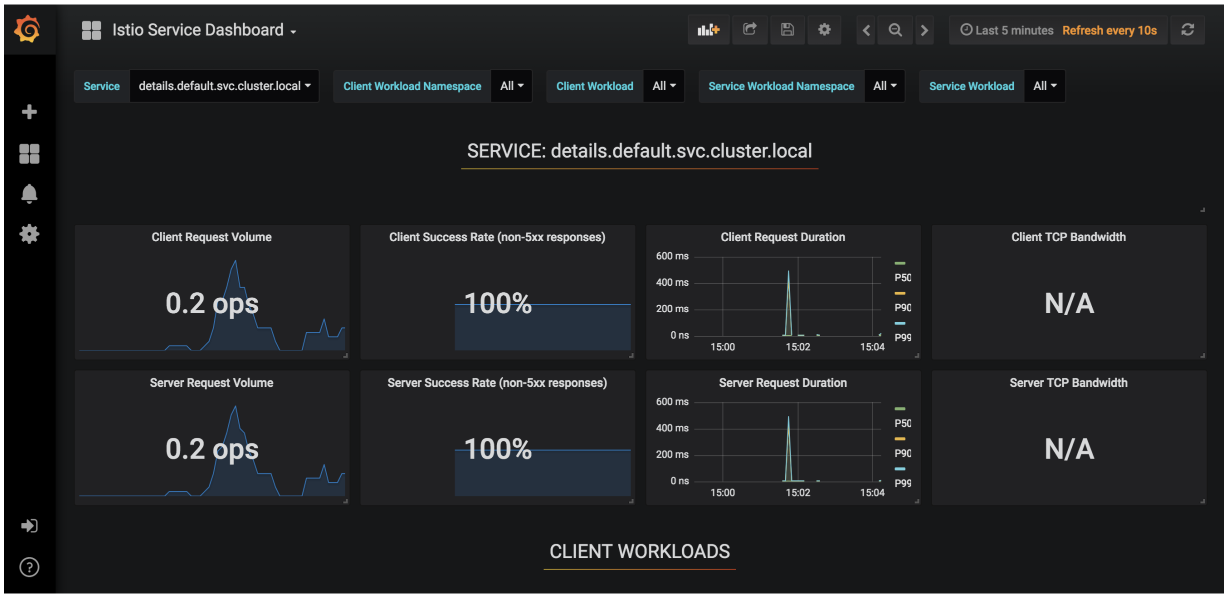
Pods

Details Service in Kubernetes Dash

This will give you relevant information about the Service name, namespace, and `ClusterIP`, which you can use to connect with your Service even as individual containers are destroyed and recreated.

A service mesh like Istio also offers service discovery capabilities. To do service discovery, Istio relies on communication between the Kubernetes API, Istio's own control plane, managed by the traffic management component Pilot, and its data plane, managed by Envoy sidecar proxies. Pilot interprets data from the Kubernetes API server to register changes in Pod locations. It then translates that data into a canonical Istio representation and forwards it onto the sidecar proxies.

This means that service discovery in Istio is platform agnostic, which we can see by using Istio's Grafana add-on to look at the `details` service again in Istio's service dashboard:



Details Service Istio Dash

Our application is running on a Kubernetes cluster, so once again we can see the relevant DNS information about the `details` Service, along with other performance data.

In a distributed architecture, it's important to have up-to-date, accurate, and easy-to-locate information about services. Both Kubernetes and service meshes like Istio offer ways to obtain this information using DNS conventions.

Routing and Traffic Configuration

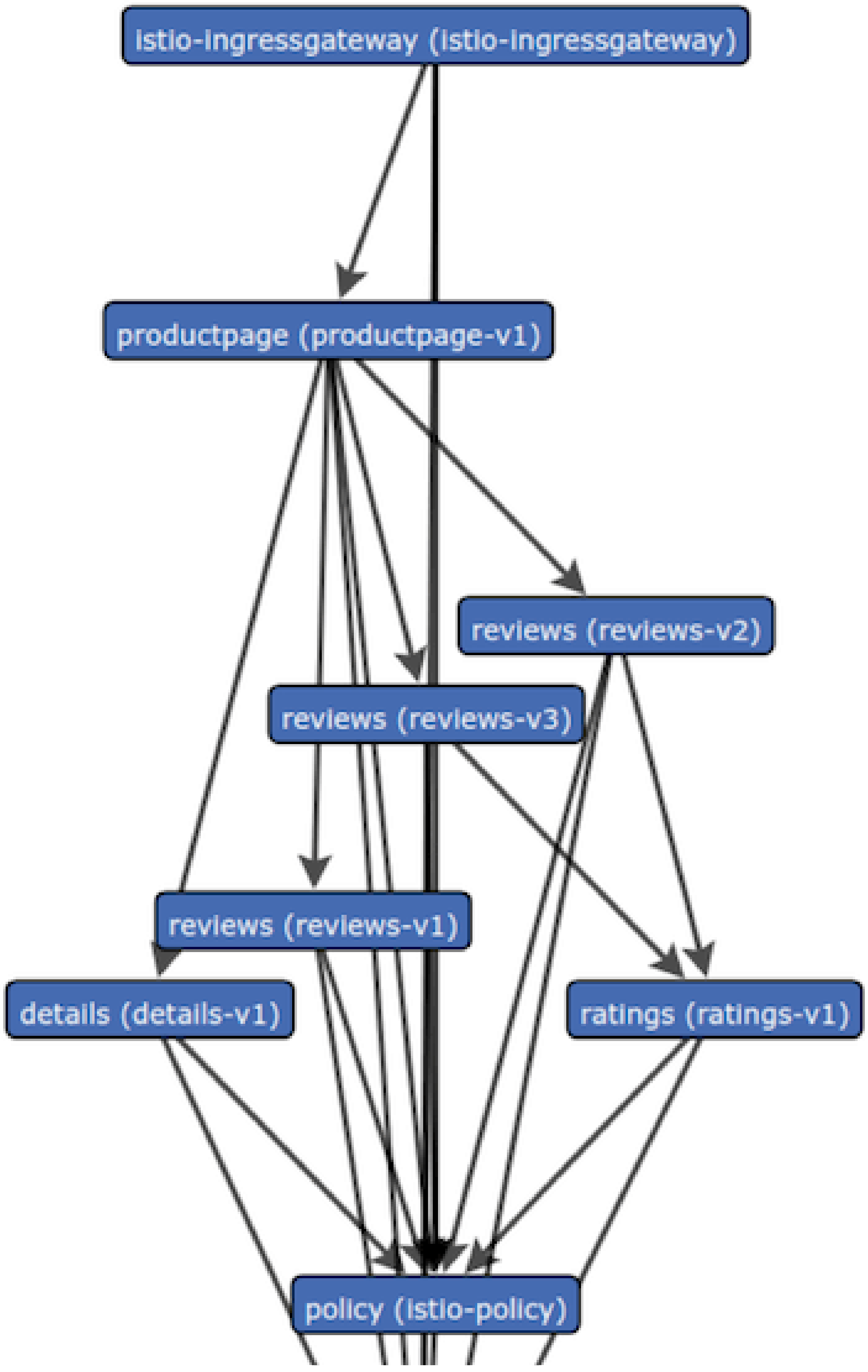
Managing traffic in a distributed framework means controlling how traffic gets to your cluster and how it's directed to your services. The more control and specificity you have in configuring external and internal traffic, the more you will be able to do with your setup. For example, in cases where you are working with canary deployments, migrating applications to new versions, or stress testing particular services through fault injection, having the ability to decide how much traffic your services are getting and where it is coming from will be key to the success of your objectives.

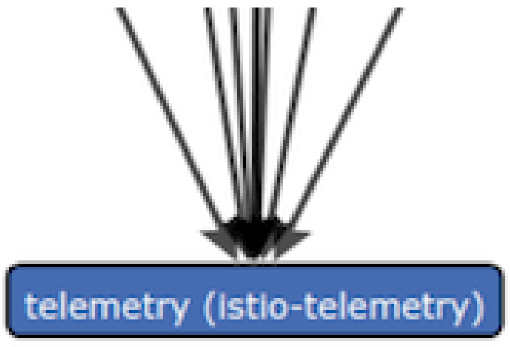
Kubernetes offers different tools, objects, and services that allow developers to control external traffic to a cluster: `kubectl proxy`, `NodePort`, Load Balancers, and Ingress Controllers and Resources. Both `kubectl proxy` and `NodePort` allow you to quickly expose your services to external traffic: `kubectl proxy` creates a proxy server that allows access to static content with an HTTP path, while `NodePort` exposes a randomly assigned port on each node. Though this offers quick access, drawbacks include having to run `kubectl` as an authenticated user, in the case of `kubectl proxy`, and a lack of flexibility in ports and node IPs, in the case of `NodePort`. And though a Load Balancer optimizes for flexibility by attaching to a particular Service, each Service requires its own Load Balancer, which can be costly.

An Ingress Resource and Ingress Controller together offer a greater degree of flexibility and configurability over these other options. Using an Ingress Controller with an Ingress Resource allows you to route external traffic to Services and configure internal routing and load balancing. To use an Ingress Resource, you need to configure your Services, the Ingress Controller and `LoadBalancer`, and the Ingress Resource itself, which will specify the desired routes to your Services. Currently, Kubernetes supports its own Nginx Controller, but there are other options you can choose from as well, managed by Nginx, Kong, and others.

Istio iterates on the Kubernetes Controller/Resource pattern with Istio Gateways and VirtualServices. Like an Ingress Controller, a Gateway defines how incoming traffic should be handled, specifying exposed ports and protocols to use. It works in conjunction with a VirtualService, which defines routes to Services within the mesh. Both of these resources communicate information to Pilot, which then forwards that information to the Envoy proxies. Though they are similar to Ingress Controllers and Resources, Gateways and VirtualServices offer a different level of control over traffic: instead of combining Open Systems Interconnection (OSI) layers and protocols, Gateways and VirtualServices allow you to differentiate between OSI layers in your settings. For example, by using VirtualServices, teams working with application layer specifications could have a separation of concerns from security operations teams working with different layer specifications. VirtualServices make it possible to separate work on discrete application features or within different trust domains, and can be used for things like canary testing, gradual rollouts, A/B testing, etc.

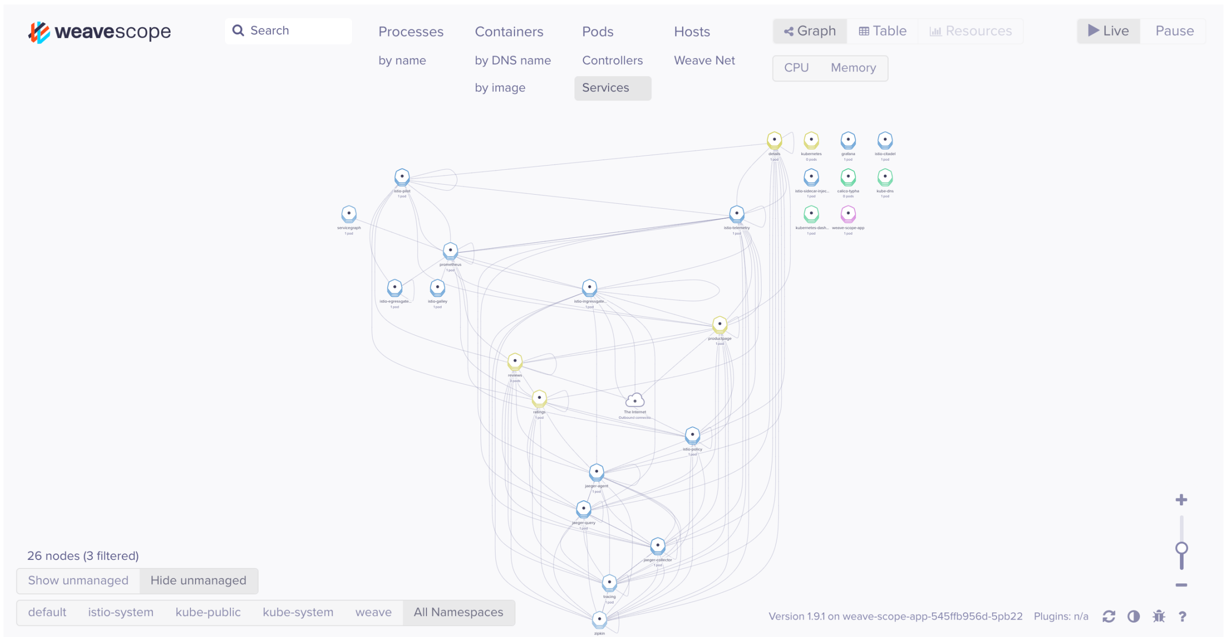
To visualize the relationship between Services, you can use Istio's Servicegraph add-on, which produces a dynamic representation of the relationship between Services using real-time traffic data. The Bookinfo application might look like this without any custom routing applied:


istio-ingressgateway (istio-ingressgateway)
productpage (productpage-v1)
reviews (reviews-v2)
reviews (reviews-v3)
reviews (reviews-v1)
details (details-v1)
ratings (ratings-v1)
policy (istio-policy)



Bookinfo service graph

Similarly, you can use a visualization tool like [Weave Scope]() to see the relationship between your Services at a given time. The Bookinfo application without advanced routing might look like this:



Weave Scope Service Map

When configuring application traffic in a distributed framework, there are a number of different solutions — from Kubernetes-native options to service meshes like Istio — that offer various options for determining how external traffic will reach your application resources and how these resources will communicate with one another.

Encryption and Authentication/Authorization

A distributed framework presents opportunities for security vulnerabilities. Instead of communicating through local internal calls, as they would in a monolithic setup, services in a microservice architecture communicate information, including privileged information, over the network. Overall, this creates a greater surface area for attacks.

Securing Kubernetes clusters involves a range of procedures; we will focus on authentication, authorization, and encryption. Kubernetes offers native approaches to each of these: - Authentication: API requests in Kubernetes are tied to user or service accounts, which need to be authenticated. There are several different ways to manage the necessary credentials: Static Tokens, Bootstrap Tokens, X509 client certificates, and external tools like OpenID Connect. - Authorization: Kubernetes has different authorization modules that allow you to determine access based on things like roles, attributes, and other specialized functions. Since all requests to the API server are denied by default, each part of an API request must be defined by an authorization policy. - Encryption: This can refer to any of the following: connections between end users and services, secret data, endpoints in the Kubernetes control plane, and communication between worker cluster components and master components. Kubernetes has different solutions for each of these: - Ingress Controllers and Resources, which can be used in conjunction with add-ons like cert-manager to manage TLS certificates. - Encryption of secret data at rest for encrypting the secrets resources in `etcd`. - TLS bootstrapping to bootstrap client certificates for kubelets and secure communication between worker nodes and the `kube-apisever`. You can also use an overlay network like Weave Net to do this.

Configuring individual security policies and protocols in Kubernetes requires administrative investment. A service mesh like Istio can consolidate some of these activities.

Istio is designed to automate some of the work of securing services. Its control plane includes several components that handle security: - Citadel: manages keys and certificates. - Pilot: oversees authentication and naming policies and shares this information with Envoy proxies. - Mixer: manages authorization and auditing.

For example, when you create a Service, Citadel receives that information from the `kube-apiserver` and creates SPIFFE certificates and keys for this Service. It then transfers this information to Pods and Envoy sidecars to facilitate communication between Services.

You can also implement some security features by enabling mutual TLS during the Istio installation. These include strong service identities for cross- and inter-cluster communication, secure service-to-service and user-to-service communication, and a key management system that can automate key and certificate creation, distribution, and rotation.

By iterating on how Kubernetes handles authentication, authorization, and encryption, service meshes like Istio are able to consolidate and extend some of the recommended best practices for running a secure Kubernetes cluster.

Metrics and Monitoring

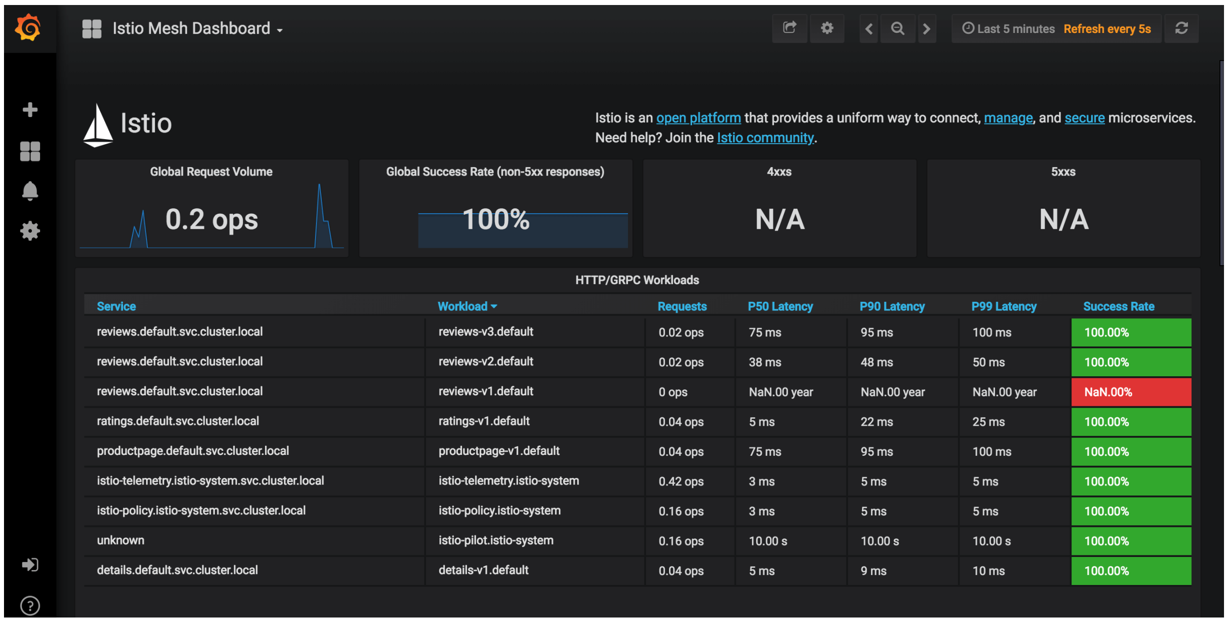
Distributed environments have changed the requirements for metrics and monitoring. Monitoring tools need to be adaptive, accounting for frequent changes to services and network addresses, and comprehensive, allowing for the amount and type of information passing between services.

Kubernetes includes some internal monitoring tools by default. These resources belong to its resource metrics pipeline, which ensures that the cluster runs as expected. The cAdvisor component collects network usage, memory, and CPU statistics from individual containers and nodes and passes that information to kubelet; kubelet in turn exposes that information via a REST API. The Metrics Server gets this information from the API and then passes it to the `kube-aggregator` for formatting.

You can extended these internal tools and monitoring capabilities with a full metrics solution. Using a service like Prometheus as a metrics aggregator allows you to build directly on top of the Kubernetes resource metrics pipeline. Prometheus integrates directly with cAdvisor through its own agents, located on the nodes. Its main aggregation service collects and stores data from the nodes and exposes it though dashboards and APIs. Additional storage and visualization options are also available if you choose to integrate your main aggregation service with backend storage, logging, and visualization tools like InfluxDB, Grafana, ElasticSearch, Logstash, Kibana, and others.

In a service mesh like Istio, the structure of the full metrics pipeline is part of the mesh's design. Envoy sidecars operating at the Pod level communicate metrics to Mixer, which manages policies and telemetry. Additionally, Prometheus and Grafana services are enabled by default (though if you are installing Istio with Helm you will need to specify `granafa.enabled=true` during installation). As is the case with the full metrics pipeline, you can also configure other services and deployments for logging and viewing options.

With these metric and visualization tools in place, you can access current information about services and workloads in a central place. For example, a global view of the BookInfo application might look like this in the Istio Grafana dashboard:



Istio is an open platform that provides a uniform way to connect, manage, and secure microservices.
Need help? Join the Istio community.

Global Request Volume	Global Success Rate (non-5xx responses)	4xxs	5xxs
0.2 ops	100%	N/A	N/A

HTTP/GRPC Workloads

Service	Workload	Requests	P50 Latency	P90 Latency	P99 Latency	Success Rate
reviews.default.svc.cluster.local	reviews-v3.default	0.02 ops	75 ms	95 ms	100 ms	100.00%
reviews.default.svc.cluster.local	reviews-v2.default	0.02 ops	38 ms	48 ms	50 ms	100.00%
reviews.default.svc.cluster.local	reviews-v1.default	0 ops	NaN.00 year	NaN.00 year	NaN.00 year	NaN.00%
ratings.default.svc.cluster.local	ratings-v1.default	0.04 ops	5 ms	22 ms	25 ms	100.00%
productpage.default.svc.cluster.local	productpage-v1.default	0.04 ops	75 ms	95 ms	100 ms	100.00%
istio-telemetry.istio-system.svc.cluster.local	istio-telemetry.istio-system	0.42 ops	3 ms	5 ms	5 ms	100.00%
istio-policy.istio-system.svc.cluster.local	istio-policy.istio-system	0.16 ops	3 ms	5 ms	5 ms	100.00%
unknown	istio-pilot.istio-system	0.16 ops	10.00 s	10.00 s	10.00 s	100.00%
details.default.svc.cluster.local	details-v1.default	0.04 ops	5 ms	9 ms	10 ms	100.00%

Bookinfo services from Grafana dash

By replicating the structure of a Kubernetes full metrics pipeline and simplifying access to some of its common components, service meshes like Istio streamline the process of data collection and visualization when working with a cluster.

Conclusion

Microservice architectures are designed to make application development and deployment fast and reliable. Yet an increase in inter-service communication has changed best practices for certain administrative tasks. This article discusses some of those tasks, how they are handled in a Kubernetes-native context, and how they can be managed using a service mesh — in this case, Istio.

For more information on some of the Kubernetes topics covered here, please see the following resources: - How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes. - How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes. - An Introduction to the Kubernetes DNS Service.

Additionally, the Kubernetes and Istio documentation hubs are great places to find detailed information about the topics discussed here.

How To Back Up and Restore a Kubernetes Cluster on DigitalOcean Using Velero

Written by Hanif Jetha and Jamon Camisso

In this tutorial you will learn how to back up and restore your Kubernetes cluster. First you will set up and configure the backup client on a local machine, and deploy the backup server into your Kubernetes cluster. You'll then deploy a sample Nginx app that uses a Persistent Volume for logging and simulate a disaster recovery scenario. After completing all the recovery steps you will have restored service to the test Nginx application.

Velero is a convenient backup tool for Kubernetes clusters that compresses and backs up Kubernetes objects to object storage. It also takes snapshots of your cluster's Persistent Volumes using your cloud provider's block storage snapshot features, and can then restore your cluster's objects and Persistent Volumes to a previous state.

The DigitalOcean Velero Plugin allows you to use DigitalOcean block storage to snapshot your Persistent Volumes, and Spaces to back up your Kubernetes objects. When running a Kubernetes cluster on DigitalOcean, this allows you to quickly back up your cluster's state and restore it should disaster strike.

In this tutorial we'll set up and configure the `velero` command line tool on a local machine, and deploy the server component into our Kubernetes cluster. We'll then deploy a sample Nginx app that uses a

Persistent Volume for logging and then simulate a disaster recovery scenario.

Prerequisites

Before you begin this tutorial, you should have the following available to you:

On your local computer: - The `kubectl` command-line tool, configured to connect to your cluster. You can read more about installing and configuring `kubectl` in the official Kubernetes documentation. - The `git` command-line utility. You can learn how to install `git` in Getting Started with Git.

In your DigitalOcean account: - A DigitalOcean Kubernetes cluster, or a Kubernetes cluster (version `1.7.5` or later) on DigitalOcean Droplets. - A DNS server running inside of your cluster. If you are using DigitalOcean Kubernetes, this is running by default. To learn more about configuring a Kubernetes DNS service, consult Customizing DNS Service from the official Kuberentes documentation. - A DigitalOcean Space that will store your backed-up Kubernetes objects. To learn how to create a Space, consult the Spaces product documentation. - An access key pair for your DigitalOcean Space. To learn how to create a set of access keys, consult How to Manage Administrative Access to Spaces. - A personal access token for use with the DigitalOcean API. To learn how to create a personal access token, consult How to Create a Personal Access Token. Ensure that the token you create or use has `Read/Write` permissions or snapshots will not work.

Once you have all of this set up, you’re ready to begin with this guide.

Step 1 — Installing the Velero Client

The Velero backup tool consists of a client installed on your local computer and a server that runs in your Kubernetes cluster. To begin, we'll install the local Velero client.

In your web browser, navigate to the Velero GitHub repo releases page, find the release corresponding to your OS and system architecture, and copy the link address. For the purposes of this guide, we'll use an Ubuntu 18.04 server on an x86-64 (or AMD64) processor as our local machine, and the Velero `v1.2.0` release.

Note: To follow this guide, you should download and install v1.2.0 of the Velero client.

Then, from the command line on your local computer, navigate to the temporary `/tmp` directory and `cd` into it:

```
cd /tmp
```

Use `wget` and the link you copied earlier to download the release tarball:

```
wget https://link_copied_from_release_page
```

Once the download completes, extract the tarball using `tar` (note the filename may differ depending on the release version and your OS):

```
tar -xvzf velero-v1.2.0-linux-amd64.tar.gz
```

The `/tmp` directory should now contain the extracted `velero-v1.2.0-linux-amd64` directory as well as the tarball you just downloaded.

Verify that you can run the `velero` client by executing the binary:

```
./velero-v1.2.0-linux-amd64/velero help
```

You should see the following help output:

Output

```
Velero is a tool for managing disaster recovery,
specifically for Kubernetes
cluster resources. It provides a simple,
configurable, and operationally robust
way to back up your application state and
associated data.

If you're familiar with kubectl, Velero supports a
similar model, allowing you to
execute commands such as 'velero get backup' and
'velero create schedule'. The same
operations can also be performed as 'velero backup
get' and 'velero schedule create'.

Usage:
  velero [command]

Available Commands:
  backup            Work with backups
  backup-location   Work with backup storage
locations
  bug               Report a Velero bug
  client            Velero client related commands
  completion        Output shell completion code
for the specified shell (bash or zsh)
  create            Create velero resources
  delete             Delete velero resources
  describe           Describe velero resources
  get                Get velero resources
  help               Help about any command
  install            Install Velero
  plugin             Work with plugins
  restic             Work with restic
  restore            Work with restores
  schedule           Work with schedules
  snapshot-location  Work with snapshot locations
  version            Print the velero version and
associated image
. . .
```

At this point you should move the `velero` executable out of the temporary `/tmp` directory and add it to your `PATH`. To add it to your `PATH` on an Ubuntu system, simply copy it to `/usr/local/bin`:

```
sudo mv velero-v1.2.0-linux-amd64/velero
/usr/local/bin/velero
```

You're now ready to configure secrets for the Velero server and then deploy it to your Kubernetes cluster.

Step 2 — Configuring Secrets

Before setting up the server component of Velero, you will need to prepare your DigitalOcean Spaces keys and API token. Again navigate to the temporary directory `/tmp` using the `cd` command:

```
cd /tmp
```

Now we'll download a copy of the Velero plugin for DigitalOcean. Visit the plugin's Github releases page and copy the link to the file ending in `.tar.gz`.

Use `wget` and the link you copied earlier to download the release tarball:

```
wget https://link_copied_from_release_page
```

Once the download completes, extract the tarball using `tar` (again note that the filename may differ depending on the release version):

```
tar -xvzf v1.0.0.tar.gz
```

The `/tmp` directory should now contain the extracted `velero-plugin-1.0.0` directory as well as the tarball you just downloaded.

Next we'll `cd` into the `velero-plugin-1.0.0` directory:

```
cd velero-plugin-1.0.0
```

Now we can save the access keys for our DigitalOcean Space and API token for use as a Kubernetes Secret. First, open up the `examples/cloud-credentials` file using your favorite editor.

```
nano examples/cloud-credentials
```

The file will look like this:

/tmp/velero-plugin-1.0.0/examples/cloud-credentials

```
[default]
aws_access_key_id=<AWS_ACCESS_KEY_ID>
aws_secret_access_key=<AWS_SECRET_ACCESS_KEY>
```

Edit the `<AWS_ACCESS_KEY_ID>` and `<AWS_SECRET_ACCESS_KEY>` placeholders to use your DigitalOcean Spaces keys. Be sure to remove the < and > characters.

The next step is to edit the `01-velero-secret.patch.yaml` file so that it includes your DigitalOcean API token. Open the file in your favourite editor:

```
nano examples/01-velero-secret.patch.yaml
```

It should look like this:

```
---
apiVersion: v1
kind: Secret
stringData:
digitalocean_token: <DIGITALOCEAN_API_TOKEN>
type: Opaque
```

Change the entire `<DIGITALOCEAN_API_TOKEN>` placeholder to use your DigitalOcean personal API token. The line should look something like `digitalocean_token: 18a0d730c0e0....`. Again, make sure to remove the < and > characters.

Step 3 — Installing the Velero Server

A Velero installation consists of a number of Kubernetes objects that all work together to create, schedule, and manage backups. The `velero` executable that you just downloaded can generate and install these objects for you. The `velero install` command will perform the preliminary set-up steps to get your cluster ready for backups. Specifically, it will:

- Create a `velero` Namespace.
- Add the `velero` Service Account.
- Configure role-based access control (RBAC) rules to grant permissions to the `velero` Service Account.

- Install Custom Resource Definitions (CRDs) for the Velero-specific resources: `Backup`, `Schedule`, `Restore`, `Config`.
- Register Velero Plugins to manage Block snapshots and Spaces storage.

We will run the `velero install` command with some non-default configuration options. Specifically, you will to need edit each of the following settings in the actual invocation of the command to match your Spaces configuration:

- `--bucket velero-backups`: Change the `velero-backups` value to match the name of your DigitalOcean Space. For example if you called your Space ‘backup-bucket’, the option would look like this: `--bucket backup-bucket`
- `--backup-location-config s3Url=https://`**`nyc3`**`.digitaloceanspaces.com,region=`**`nyc3`**: Change the URL and region to match your Space’s settings. Specifically, edit both `nyc3` portions to match the region where your Space is hosted. For example, if your Space is hosted in the `fra1` region, the line would look like this: `--backup-location-config s3Url=https://`**`fra1`**`.digitaloceanspaces.com,region=`**`fra1`**. The identifiers for regions are: `nyc3`, `sfo2`, `sgp1`, and `fra1`.

Once you are ready with the appropriate bucket and backup location settings, it is time to install Velero. Run the following command, substituting your values where required:

```
velero install \
  --provider velero.io/aws \
  --bucket velero-backups \
  --plugins velero/velero-plugin-for-
aws:v1.0.0,digitalocean/velero-plugin:v1.0.0 \
  --backup-location-config
s3Url=https://nyc3.digitaloceanspaces.com,region=n
yc3 \
  --use-volume-snapshots=false \
  --secret-file=./examples/cloud-credentials
```

You should see the following output:

Output

```
CustomResourceDefinition/backups.velero.io:
attempting to create resource
CustomResourceDefinition/backups.velero.io:
created
CustomResourceDefinition/backupstoragelocations.ve
lero.io: attempting to create resource
CustomResourceDefinition/backupstoragelocations.ve
lero.io: created
CustomResourceDefinition/deletebackuprequests.vele
ro.io: attempting to create resource
CustomResourceDefinition/deletebackuprequests.vele
ro.io: created
CustomResourceDefinition/downloadrequests.velero.i
o: attempting to create resource
```

```
CustomResourceDefinition/downloadrequests.velero.i
o: created
CustomResourceDefinition/podvolumebackups.velero.i
o: attempting to create resource
CustomResourceDefinition/podvolumebackups.velero.i
o: created
CustomResourceDefinition/podvolumerestores.velero.
io: attempting to create resource
CustomResourceDefinition/podvolumerestores.velero.
io: created
CustomResourceDefinition/resticrepositories.velero
.io: attempting to create resource
CustomResourceDefinition/resticrepositories.velero
.io: created
CustomResourceDefinition/restores.velero.io:
attempting to create resource
CustomResourceDefinition/restores.velero.io:
created
CustomResourceDefinition/schedules.velero.io:
attempting to create resource
CustomResourceDefinition/schedules.velero.io:
created
CustomResourceDefinition/serverstatusrequests.vele
ro.io: attempting to create resource
CustomResourceDefinition/serverstatusrequests.vele
ro.io: created
CustomResourceDefinition/volumesnapshotlocations.v
```

```
elero.io: attempting to create resource
CustomResourceDefinition/volumesnapshotlocations.v
elero.io: created
Waiting for resources to be ready in cluster...
Namespace/velero: attempting to create resource
Namespace/velero: created
ClusterRoleBinding/velero: attempting to create
resource
ClusterRoleBinding/velero: created
ServiceAccount/velero: attempting to create
resource
ServiceAccount/velero: created
Secret/cloud-credentials: attempting to create
resource
Secret/cloud-credentials: created
BackupStorageLocation/default: attempting to
create resource
BackupStorageLocation/default: created
Deployment/velero: attempting to create resource
Deployment/velero: created
Velero is installed! ⛵ Use 'kubectl logs
deployment/velero -n velero' to view the status.
```

You can watch the deployment logs using the `kubectl` command from the output. Once your deploy is ready, you can proceed to the next step, which is configuring the server. A successful deploy will look like this (with a different AGE column):

```
kubectl get deployment/velero --namespace velero
```

Output

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
velero    1/1     1            1           2m
```

At this point you have installed the server component of Velero into your Kubernetes cluster as a Deployment. You have also registered your Spaces keys with Velero using a Kubernetes Secret.

Note: You can specify the `kubeconfig` that the `velero` command line tool should use with the `--kubeconfig` flag. If you don't use this flag, `velero` will check the `KUBECONFIG` environment variable and then fall back to the `kubectl` default (`~/.kube/config`).

Step 4 — Configuring snapshots

When we installed the Velero server, the option `--use-volume-snapshots=false` was part of the command. Since we want to take snapshots of the underlying block storage devices in our Kubernetes cluster, we need to tell Velero to use the correct plugin for DigitalOcean block storage.

Run the following command to enable the plugin and register it as the default snapshot provider:

```
velero snapshot-location create default --provider
digitalocean.com/velero
```

You will see the following output:

Output

```
Snapshot volume location "default" configured
successfully.
```

Step 5 — Adding an API token

In the previous step we created block storage and object storage objects in the Velero server. We've registered the `digitalocean/velero-plugin:v1.0.0` plugin with the server, and installed our Spaces secret keys into the cluster.

The final step is patching the `cloud-credentials` Secret that we created earlier to use our DigitalOcean API token. Without this token the snapshot plugin will not be able to authenticate with the DigitalOcean API.

We could use the `kubectl edit` command to modify the Velero Deployment object with a reference to the API token. However, editing complex YAML objects by hand can be tedious and error prone. Instead, we'll use the `kubectl patch` command since Kubernetes supports patching objects. Let's take a quick look at the contents of the patch files that we'll apply.

The first patch file is the `examples/01-velero-secret.patch.yaml` file that you edited earlier. It is designed to add your API token to the `secrets/cloud-credentials` Secret that already contains your Spaces keys. `cat` the file:

```
cat examples/01-velero-secret.patch.yaml
```

It should look like this (with your token in place of the `<DIGITALOCEAN_API_TOKEN>` placeholder):

examples/01-velero-secret.patch.yaml

```
. . .
---
apiVersion: v1
kind: Secret
```

```
stringData:
  digitalocean_token: <DIGITALOCEAN_API_TOKEN>
type: Opaque
```

Now let’s look at the patch file for the Deployment:

```
cat examples/02-velero-deployment.patch.yaml
```

You should see the following YAML:

examples/02-velero-deployment.patch.yaml

```
. . .
---
apiVersion: v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - args:
        - server
        command:
        - /velero
        env:
        - name: DIGITALOCEAN_TOKEN
          valueFrom:
            secretKeyRef:
              key: digitalocean_token
              name: cloud-credentials
        name: velero
```

This file indicates that we're patching a Deployment's Pod spec that is called `velero`. Since this is a patch we do not need to specify an entire Kubernetes object spec or metadata. In this case the Velero Deployment is already configured using the `cloud-credentials` secret because the `velero install` command created it for us. So all that this patch needs to do is register the `digitalocean_token` as an environment variable with the already deployed Velero Pod.

Let's apply the first Secret patch using the `kubectl patch` command:

```
kubectl patch secret/cloud-credentials -p "$(cat
examples/01-velero-secret.patch.yaml)" --namespace
velero
```

You should see the following output:

Output

```
secret/cloud-credentials patched
```

Finally we will patch the Deployment. Run the following command:

```
kubectl patch deployment/velero -p "$(cat
examples/02-velero-deployment.patch.yaml") --
namespace velero
```

You will see the following if the patch is successful:

Output

```
deployment.apps/velero patched
```

Let's verify the patched Deployment is working using `kubectl get` on the `velero` Namespace:

```
kubectl get deployment/velero --namespace velero
```

You should see the following output:

Output

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
velero    1/1     1            1           12s
```

At this point Velero is running and fully configured, and ready to back up and restore your Kubernetes cluster objects and Persistent Volumes to DigitalOcean Spaces and Block Storage.

In the next section, we'll run a quick test to make sure that the backup and restore functionality works as expected.

Step 6 — Testing Backup and Restore Procedure

Now that we've successfully installed and configured Velero, we can create a test Nginx Deployment, with a Persistent Volume and Service. Once the Deployment is running we will run through a backup and restore drill to ensure that Velero is configured and working properly.

Ensure you are still working in the `/tmp/velero-plugin-1.0.0` directory. The `examples` directory contains a sample Nginx manifest called `nginx-example.yaml`.

Open this file using your editor of choice:

```
nano examples/nginx-example.yaml
```

You should see the following text:

Output

```
. . .
---
apiVersion: v1
```

```yaml
kind: Namespace
metadata:
  name: nginx-example
  labels:
    app: nginx

---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nginx-logs
  namespace: nginx-example
  labels:
    app: nginx
spec:
  storageClassName: do-block-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deploy
  namespace: nginx-example
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: nginx-logs
          persistentVolumeClaim:
           claimName: nginx-logs
      containers:
      - image: nginx:stable
        name: nginx
        ports:
        - containerPort: 80
        volumeMounts:
          - mountPath: "/var/log/nginx"
            name: nginx-logs
            readOnly: false
```

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx-svc
  namespace: nginx-example
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

In this file, we observe specs for:

- An Nginx namespace called `nginx-example`
- An Nginx Deployment consisting of a single replica of the `nginx:stable` container image
- A 5Gi Persistent Volume Claim (called `nginx-logs`), using the `do-block-storage` StorageClass
- A `LoadBalancer` Service that exposes port `80`

Create the objects using `kubectl apply`:

```
kubectl apply -f examples/nginx-example.yaml
```

You should see the following output:

Output

```
namespace/nginx-example created
persistentvolumeclaim/nginx-logs created
deployment.apps/nginx-deploy created
service/nginx-svc created
```

Check that the Deployment succeeded:

```
kubectl get deployments --namespace=nginx-example
```

You should see the following output:

Output

```
NAME           READY   UP-TO-DATE   AVAILABLE
AGE
nginx-deploy   1/1     1            1
1m23s
```

Once `Available` reaches 1, fetch the Nginx load balancer's external IP using `kubectl get`:

```
kubectl get services --namespace=nginx-example
```

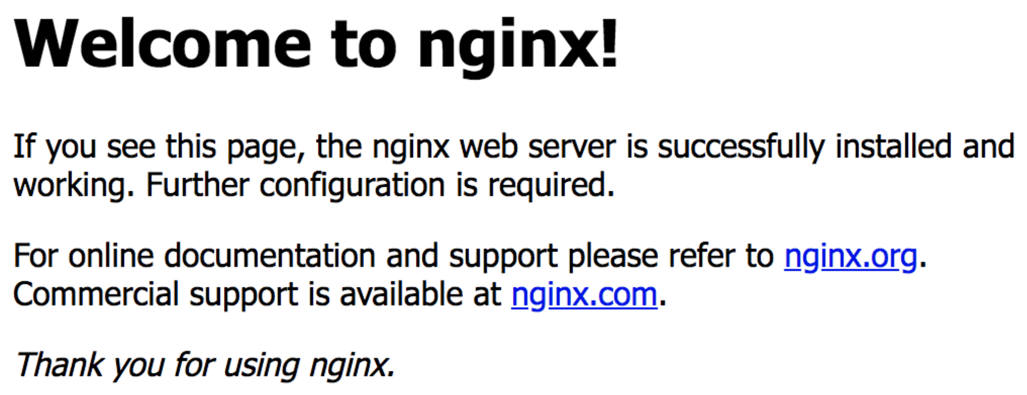
You should see both the internal `CLUSTER-IP` and `EXTERNAL-IP` for the `my-nginx` Service:

Output

```
NAME        TYPE           CLUSTER-IP
EXTERNAL-IP      PORT(S)        AGE
nginx-svc   LoadBalancer   10.245.147.61
159.203.48.191   80:30232/TCP   3m1s
```

Note the `EXTERNAL-IP` and navigate to it using your web browser.

You should see the following NGINX welcome page:


Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Nginx Welcome Page

This indicates that your Nginx Deployment and Service are up and running.

Before we simulate our disaster scenario, let's first check the Nginx access logs (stored on a Persistent Volume attached to the Nginx Pod):

Fetch the Pod's name using `kubectl get`:

```
kubectl get pods --namespace nginx-example
```

Output

```
NAME                              READY    STATUS
RESTARTS    AGE
nginx-deploy-694c85cdc8-vknsk     1/1      Running
0           4m14s
```

Now, `exec` into the running Nginx container to get a shell inside of it:

```
kubectl exec -it nginx-deploy-694c85cdc8-vknsk --
namespace nginx-example -- /bin/bash
```

Once inside the Nginx container, `cat` the Nginx access logs:

```
[environment second]
cat /var/log/nginx/access.log
```

You should see some Nginx access entries:

```
[environment second]
[secondary_label Output]
10.244.0.119 - - [03/Jan/2020:04:43:04 +0000] "GET
/ HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Linux
x86_64; rv:72.0) Gecko/20100101 Firefox/72.0" "-"
10.244.0.119 - - [03/Jan/2020:04:43:04 +0000] "GET
/favicon.ico HTTP/1.1" 404 153 "-" "Mozilla/5.0
(X11; Linux x86_64; rv:72.0) Gecko/20100101
Firefox/72.0" "-"
```

Note these down (especially the timestamps), as we will use them to confirm the success of the restore procedure. Exit the pod:

```
exit
```

We can now perform the backup procedure to copy all `nginx` Kubernetes objects to Spaces and take a Snapshot of the Persistent Volume we created when deploying Nginx.

We'll create a backup called `nginx-backup` using the `velero` command line client:

```
velero backup create nginx-backup --selector
app=nginx
```

The `--selector app=nginx` instructs the Velero server to only back up Kubernetes objects with the `app=nginx` Label Selector.

You should see the following output:

Output

```
Backup request "nginx-backup" submitted
successfully.
Run `velero backup describe nginx-backup` or
`velero backup logs nginx-backup` for more
details.
```

Running `velero backup describe nginx-backup --details` should provide the following output after a short delay:

Output

```
Name:          nginx-backup
Namespace:     velero
Labels:        velero.io/backup=nginx-backup
               velero.io/pv=pvc-6b7f63d7-752b-4537-
9bb0-003bed9129ca
               velero.io/storage-location=default
Annotations:   <none>

Phase:  Completed

Namespaces:
  Included:  *
  Excluded:  <none>
```

```
Resources:
  Included:        *
  Excluded:        <none>
  Cluster-scoped:  auto

Label selector:  app=nginx

Storage Location:  default

Snapshot PVs:  auto

TTL:  720h0m0s

Hooks:  <none>

Backup Format Version:  1

Started:    2020-01-02 23:45:30 -0500 EST
Completed:  2020-01-02 23:45:34 -0500 EST

Expiration:  2020-02-01 23:45:30 -0500 EST

Resource List:
  apps/v1/Deployment:
    - nginx-example/nginx-deploy
  apps/v1/ReplicaSet:
    - nginx-example/nginx-deploy-694c85cdc8
  v1/Endpoints:
    - nginx-example/nginx-svc
  v1/Namespace:
    - nginx-example
  v1/PersistentVolume:
    - pvc-6b7f63d7-752b-4537-9bb0-003bed9129ca
  v1/PersistentVolumeClaim:
    - nginx-example/nginx-logs
  v1/Pod:
    - nginx-example/nginx-deploy-694c85cdc8-vknsk
  v1/Service:
    - nginx-example/nginx-svc

Persistent Volumes:
  pvc-6b7f63d7-752b-4537-9bb0-003bed9129ca:
    Snapshot ID:        dfe866cc-2de3-11ea-9ec0-
0a58ac14e075
    Type:               ext4
    Availability Zone:
    IOPS:               <N/A>
```

This output indicates that `nginx-backup` completed successfully. The list of resources shows each of the Kubernetes objects that was included in the backup. The final section shows the PersistentVolume was also backed up using a filesystem snapshot.

To confirm from within the DigitalOcean Cloud Control Panel, navigate to the Space containing your Kubernetes backup files.

You should see a new directory called `nginx-backup` containing the Velero backup files.

Using the left-hand navigation bar, go to Images and then Snapshots. Within Snapshots, navigate to Volumes. You should see a Snapshot corresponding to the PVC listed in the above output.

We can now test the restore procedure.

Let’s first delete the `nginx-example` Namespace. This will delete everything in the Namespace, including the Load Balancer and Persistent Volume:

```
kubectl delete namespace nginx-example
```

Verify that you can no longer access Nginx at the Load Balancer endpoint, and that the `nginx-example` Deployment is no longer running:

```
kubectl get deployments --namespace=nginx-example
```

Output

```
No resources found in nginx-example namespace.
```

We can now perform the restore procedure, once again using the `velero` client:

```
velero restore create --from-backup nginx-backup
```

Here we use `create` to create a Velero `Restore` object from the `nginx-backup` object.

You should see the following output:

Output

```
Restore request "nginx-backup-20200102235032"
submitted successfully.
```

```
Run `velero restore describe nginx-backup-
20200102235032` or `velero restore logs nginx-
backup-20200102235032` for more details.
```

Check the status of the restored Deployment:

```
kubectl get deployments --namespace=nginx-example
```

Output

```
NAME              READY   UP-TO-DATE   AVAILABLE
AGE
nginx-deploy   1/1     1            1
58s
```

Check for the creation of a Persistent Volume:

```
 kubectl get pvc --namespace=nginx-example
```

Output

```
NAME           STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS       AGE
nginx-logs   Bound    pvc-6b7f63d7-752b-4537-9bb0-
003bed9129ca   5Gi        RWO            do-block-
storage   75s
```

The restore also created a LoadBalancer. Sometimes the Service will be re-created with a new IP address. You will need to find the `EXTERNAL-IP` address again:

```
kubectl get services --namespace nginx-example
```

Output

```
NAME         TYPE             CLUSTER-IP
EXTERNAL-IP      PORT(S)        AGE
```

```
nginx-svc   LoadBalancer   10.245.15.83
159.203.48.191   80:31217/TCP   97s
```

Navigate to the Nginx Service's external IP once again to confirm that Nginx is up and running.

Finally, check the logs on the restored Persistent Volume to confirm that the log history has been preserved post-restore.

To do this, once again fetch the Pod's name using `kubectl get`:

```
kubectl get pods --namespace nginx-example
```

Output

```
NAME                              READY   STATUS
RESTARTS   AGE
nginx-deploy-694c85cdc8-vknsk     1/1     Running
0          2m20s
```

Then `exec` into it:

```
kubectl exec -it nginx-deploy-694c85cdc8-vknsk --
namespace nginx-example -- /bin/bash
```

Once inside the Nginx container, `cat` the Nginx access logs:

```
[environment second]
cat /var/log/nginx/access.log
[environment second]
[secondary_label Output]
10.244.0.119 - - [03/Jan/2020:04:43:04 +0000] "GET
/ HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Linux
x86_64; rv:72.0) Gecko/20100101 Firefox/72.0" "-"
10.244.0.119 - - [03/Jan/2020:04:43:04 +0000] "GET
/favicon.ico HTTP/1.1" 404 153 "-" "Mozilla/5.0
```

```
(X11; Linux x86_64; rv:72.0) Gecko/20100101
Firefox/72.0" "-"
```

You should see the same pre-backup access attempts (note the timestamps), confirming that the Persistent Volume restore was successful. Note that there may be additional attempts in the logs if you visited the Nginx landing page after you performed the restore.

At this point, we've successfully backed up our Kubernetes objects to DigitalOcean Spaces, and our Persistent Volumes using Block Storage Volume Snapshots. We simulated a disaster scenario, and restored service to the test Nginx application.

Conclusion

In this guide we installed and configured the Velero Kubernetes backup tool on a DigitalOcean-based Kubernetes cluster. We configured the tool to back up Kubernetes objects to DigitalOcean Spaces, and back up Persistent Volumes using Block Storage Volume Snapshots.

Velero can also be used to schedule regular backups of your Kubernetes cluster for disaster recovery. To do this, you can use the `velero schedule` command. Velero can also be used to migrate resources from one cluster to another.

To learn more about DigitalOcean Spaces, consult the official Spaces documentation. To learn more about Block Storage Volumes, consult the Block Storage Volume documentation.

This tutorial builds on the README found in StackPointCloud's `ark-plugin-digitalocean` GitHub repo.

How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes

Written by Hanif Jetha

When running multiple services and applications on a Kubernetes cluster, a centralized, cluster-level logging stack can help you quickly sort through and analyze the heavy volume of log data produced by your Pods. In this tutorial, you will learn how to set up and configure Elasticsearch, Fluentd, and Kibana (the EFK stack) on your Kubernetes cluster.

To start, you will configure and launch a scalable Elasticsearch cluster on top of your Kubernetes cluster. From there you will create a Kubernetes Service and Deployment for Kibanaso that you can visualize and work with your logs. Finally, you will set up Fluentd as a Kubernetes DaemonSet so that it runs on every worker Node and collects logs from every Pod in your cluster.

When running multiple services and applications on a Kubernetes cluster, a centralized, cluster-level logging stack can help you quickly sort through and analyze the heavy volume of log data produced by your Pods. One popular centralized logging solution is the Elasticsearch, Fluentd, and Kibana (EFK) stack.

Elasticsearch is a real-time, distributed, and scalable search engine which allows for full-text and structured search, as well as analytics. It is commonly used to index and search through large volumes of log data, but can also be used to search many different kinds of documents.

Elasticsearch is commonly deployed alongside Kibana, a powerful data visualization frontend and dashboard for Elasticsearch. Kibana allows you to explore your Elasticsearch log data through a web interface, and build dashboards and queries to quickly answer questions and gain insight into your Kubernetes applications.

In this tutorial we'll use Fluentd to collect, transform, and ship log data to the Elasticsearch backend. Fluentd is a popular open-source data collector that we'll set up on our Kubernetes nodes to tail container log files, filter and transform the log data, and deliver it to the Elasticsearch cluster, where it will be indexed and stored.

We'll begin by configuring and launching a scalable Elasticsearch cluster, and then create the Kibana Kubernetes Service and Deployment. To conclude, we'll set up Fluentd as a DaemonSet so it runs on every Kubernetes worker node.

Prerequisites

Before you begin with this guide, ensure you have the following available to you:

- A Kubernetes 1.10+ cluster with role-based access control (RBAC) enabled

    - Ensure your cluster has enough resources available to roll out the EFK stack, and if not scale your cluster by adding worker nodes. We'll be deploying a 3-Pod Elasticsearch cluster (you can scale this down to 1 if necessary), as well as a single Kibana Pod. Every worker node will also run a Fluentd Pod. The cluster in this guide consists of 3 worker nodes and a managed control plane.

- The `kubectl` command-line tool installed on your local machine, configured to connect to your cluster. You can read more about installing `kubectl` in the official documentation.

Once you have these components set up, you're ready to begin with this guide.

Step 1 — Creating a Namespace

Before we roll out an Elasticsearch cluster, we'll first create a Namespace into which we'll install all of our logging instrumentation. Kubernetes lets you separate objects running in your cluster using a "virtual cluster" abstraction called Namespaces. In this guide, we'll create a `kube-logging` namespace into which we'll install the EFK stack components. This Namespace will also allow us to quickly clean up and remove the logging stack without any loss of function to the Kubernetes cluster.

To begin, first investigate the existing Namespaces in your cluster using `kubectl`:

```
kubectl get namespaces
```

You should see the following three initial Namespaces, which come preinstalled with your Kubernetes cluster:

Output

```
NAME            STATUS    AGE
default         Active    5m
```

```
kube-system   Active    5m
kube-public   Active    5m
```

The `default` Namespace houses objects that are created without specifying a Namespace. The `kube-system` Namespace contains objects created and used by the Kubernetes system, like `kube-dns`, `kube-proxy`, and `kubernetes-dashboard`. It's good practice to keep this Namespace clean and not pollute it with your application and instrumentation workloads.

The `kube-public` Namespace is another automatically created Namespace that can be used to store objects you'd like to be readable and accessible throughout the whole cluster, even to unauthenticated users.

To create the `kube-logging` Namespace, first open and edit a file called `kube-logging.yaml` using your favorite editor, such as nano:

```
nano kube-logging.yaml
```

Inside your editor, paste the following Namespace object YAML:

kube-logging.yaml

```
kind: Namespace
apiVersion: v1
metadata:
  name: kube-logging
```

Then, save and close the file.

Here, we specify the Kubernetes object's `kind` as a `Namespace` object. To learn more about `Namespace` objects, consult the Namespaces Walkthrough in the official Kubernetes documentation. We also specify the Kubernetes API version used to create the object (`v1`), and give it a `name`, `kube-logging`.

Once you've created the `kube-logging.yaml` Namespace object file, create the Namespace using `kubectl create` with the `-f` filename flag:

```
kubectl create -f kube-logging.yaml
```

You should see the following output:

Output

```
namespace/kube-logging created
```

You can then confirm that the Namespace was successfully created:

```
kubectl get namespaces
```

At this point, you should see the new `kube-logging` Namespace:

Output

```
NAME              STATUS    AGE
default           Active    23m
kube-logging      Active    1m
kube-public       Active    23m
kube-system       Active    23m
```

We can now deploy an Elasticsearch cluster into this isolated logging Namespace.

Step 2 — Creating the Elasticsearch StatefulSet

Now that we've created a Namespace to house our logging stack, we can begin rolling out its various components. We'll first begin by deploying a 3-node Elasticsearch cluster.

In this guide, we use 3 Elasticsearch Pods to avoid the "split-brain" issue that occurs in highly-available, multi-node clusters. At a high-level, “split-brain” is what arises when one or more nodes can’t communicate with the others, and several “split” masters get elected. With 3 nodes, if one gets disconnected from the cluster temporarily, the other two nodes can elect a new master and the cluster can continue functioning while the last node attempts to rejoin. To learn more, consult A new era for cluster coordination in Elasticsearch and Voting configurations.

Creating the Headless Service

To start, we’ll create a headless Kubernetes service called `elasticsearch` that will define a DNS domain for the 3 Pods. A headless service does not perform load balancing or have a static IP; to learn more about headless services, consult the official Kubernetes documentation.

Open a file called `elasticsearch_svc.yaml` using your favorite editor:

```
nano elasticsearch_svc.yaml
```

Paste in the following Kubernetes service YAML:

elasticsearch_svc.yaml

```
kind: Service
apiVersion: v1
metadata:
  name: elasticsearch
  namespace: kube-logging
  labels:
    app: elasticsearch
spec:
  selector:
    app: elasticsearch
  clusterIP: None
  ports:
    - port: 9200
      name: rest
    - port: 9300
      name: inter-node
```

Then, save and close the file.

We define a `Service` called `elasticsearch` in the `kube-logging` Namespace, and give it the `app: elasticsearch` label. We then set the `.spec.selector` to `app: elasticsearch` so that the Service selects Pods with the `app: elasticsearch` label. When we associate our Elasticsearch StatefulSet with this Service, the Service will return DNS A records that point to Elasticsearch Pods with the `app: elasticsearch` label.

We then set `clusterIP: None`, which renders the service headless. Finally, we define ports `9200` and `9300` which are used to interact with the REST API, and for inter-node communication, respectively.

Create the service using `kubectl`:

```
kubectl create -f elasticsearch_svc.yaml
```

You should see the following output:

Output

```
service/elasticsearch created
```

Finally, double-check that the service was successfully created using `kubectl get`:

```
kubectl get services --namespace=kube-logging
```

You should see the following:

Output

```
NAME             TYPE        CLUSTER-IP   EXTERNAL-
IP   PORT(S)             AGE
elasticsearch   ClusterIP   None         <none>
9200/TCP,9300/TCP   26s
```

Now that we’ve set up our headless service and a stable `.elasticsearch.kube-logging.svc.cluster.local` domain for our Pods, we can go ahead and create the StatefulSet.

Creating the StatefulSet

A Kubernetes StatefulSet allows you to assign a stable identity to Pods and grant them stable, persistent storage. Elasticsearch requires stable storage to persist data across Pod rescheduling and restarts. To learn more about the StatefulSet workload, consult the Statefulsets page from the Kubernetes docs.

Open a file called `elasticsearch_statefulset.yaml` in your favorite editor:

```
nano elasticsearch_statefulset.yaml
```

We will move through the StatefulSet object definition section by section, pasting blocks into this file.

Begin by pasting in the following block:

elasticsearch_statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: es-cluster
  namespace: kube-logging
spec:
  serviceName: elasticsearch
  replicas: 3
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
```

In this block, we define a StatefulSet called `es-cluster` in the `kube-logging` namespace. We then associate it with our previously created `elasticsearch` Service using the `serviceName` field. This ensures that each Pod in the StatefulSet will be accessible using the following DNS address: `es-cluster-[0,1,2].elasticsearch.kube-logging.svc.cluster.local`, where `[0,1,2]` corresponds to the Pod's assigned integer ordinal.

We specify 3 `replicas` (Pods) and set the `matchLabels` selector to `app: elasticseach`, which we then mirror in the `.spec.template.metadata` section. The

`.spec.selector.matchLabels` and `.spec.template.metadata.labels` fields must match.

We can now move on to the object spec. Paste in the following block of YAML immediately below the preceding block:

elasticsearch_statefulset.yaml

```
. . .
    spec:
      containers:
      - name: elasticsearch
        image:
docker.elastic.co/elasticsearch/elasticsearch:7.2.
0
        resources:
            limits:
              cpu: 1000m
            requests:
              cpu: 100m
        ports:
        - containerPort: 9200
          name: rest
          protocol: TCP
        - containerPort: 9300
          name: inter-node
          protocol: TCP
        volumeMounts:
        - name: data
```

```
          mountPath: /usr/share/elasticsearch/data
        env:
          - name: cluster.name
            value: k8s-logs
          - name: node.name
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: discovery.seed_hosts
            value: "es-cluster-0.elasticsearch,es-
cluster-1.elasticsearch,es-cluster-
2.elasticsearch"
          - name: cluster.initial_master_nodes
            value: "es-cluster-0,es-cluster-1,es-
cluster-2"
          - name: ES_JAVA_OPTS
            value: "-Xms512m -Xmx512m"
```

Here we define the Pods in the StatefulSet. We name the containers `elasticsearch` and choose the `docker.elastic.co/elasticsearch/elasticsearch:7.2.0` Docker image. At this point, you may modify this image tag to correspond to your own internal Elasticsearch image, or a different version. Note that for the purposes of this guide, only Elasticsearch `7.2.0` has been tested.

We then use the `resources` field to specify that the container needs at least 0.1 vCPU guaranteed to it, and can burst up to 1 vCPU (which limits the Pod's resource usage when performing an initial large ingest or dealing with a load spike). You should modify these values depending on your anticipated load and available resources. To learn more about resource requests and limits, consult the official Kubernetes Documentation.

We then open and name ports `9200` and `9300` for REST API and inter-node communication, respectively. We specify a `volumeMount` called `data` that will mount the PersistentVolume named `data` to the container at the path `/usr/share/elasticsearch/data`. We will define the VolumeClaims for this StatefulSet in a later YAML block.

Finally, we set some environment variables in the container:

- `cluster.name`: The Elasticsearch cluster's name, which in this guide is `k8s-logs`.
- `node.name`: The node's name, which we set to the `.metadata.name` field using `valueFrom`. This will resolve to `es-cluster-[0,1,2]`, depending on the node's assigned ordinal.
- `discovery.seed_hosts`: This field sets a list of master-eligible nodes in the cluster that will seed the node discovery process. In this guide, thanks to the headless service we configured earlier, our Pods have domains of the form `es-cluster-[0,1,2].elasticsearch.kube-logging.svc.cluster.local`, so we set this variable accordingly. Using local namespace Kubernetes DNS resolution, we can shorten this to `es-cluster-[0,1,2].elasticsearch`. To learn more about Elasticsearch discovery, consult the official Elasticsearch documentation.
- `cluster.initial_master_nodes`: This field also specifies a list of master-eligible nodes that will participate in the master

election process. Note that for this field you should identify nodes by their `node.name`, and not their hostnames.

- `ES_JAVA_OPTS`: Here we set this to `-Xms512m -Xmx512m` which tells the JVM to use a minimum and maximum heap size of 512 MB. You should tune these parameters depending on your cluster's resource availability and needs. To learn more, consult Setting the heap size.

The next block we'll paste in looks as follows:

elasticsearch_statefulset.yaml

```
. . .
      initContainers:
      - name: fix-permissions
        image: busybox
        command: ["sh", "-c", "chown -R 1000:1000
/usr/share/elasticsearch/data"]
        securityContext:
          privileged: true
        volumeMounts:
        - name: data
          mountPath: /usr/share/elasticsearch/data
      - name: increase-vm-max-map
        image: busybox
        command: ["sysctl", "-w",
"vm.max_map_count=262144"]
        securityContext:
          privileged: true
      - name: increase-fd-ulimit
        image: busybox
        command: ["sh", "-c", "ulimit -n 65536"]
        securityContext:
          privileged: true
```

In this block, we define several Init Containers that run before the main `elasticsearch` app container. These Init Containers each run to completion in the order they are defined. To learn more about Init Containers, consult the official Kubernetes Documentation.

The first, named `fix-permissions`, runs a `chown` command to change the owner and group of the Elasticsearch data directory to `1000:1000`, the Elasticsearch user's UID. By default Kubernetes mounts the data directory as `root`, which renders it inaccessible to Elasticsearch. To learn more about this step, consult Elasticsearch's "Notes for production use and defaults."

The second, named `increase-vm-max-map`, runs a command to increase the operating system's limits on mmap counts, which by default may be too low, resulting in out of memory errors. To learn more about this step, consult the official Elasticsearch documentation.

The next Init Container to run is `increase-fd-ulimit`, which runs the `ulimit` command to increase the maximum number of open file descriptors. To learn more about this step, consult the "Notes for Production Use and Defaults" from the official Elasticsearch documentation.

Note: The Elasticsearch Notes for Production Use also mentions disabling swapping for performance reasons. Depending on your

Kubernetes installation or provider, swapping may already be disabled. To check this, `exec` into a running container and run `cat /proc/swaps` to list active swap devices. If you see nothing there, swap is disabled.

Now that we've defined our main app container and the Init Containers that run before it to tune the container OS, we can add the final piece to our StatefulSet object definition file: the `volumeClaimTemplates`.

Paste in the following `volumeClaimTemplate` block:

elasticsearch_statefulset.yaml

```
. . .
  volumeClaimTemplates:
  - metadata:
      name: data
      labels:
        app: elasticsearch
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: do-block-storage
      resources:
        requests:
          storage: 100Gi
```

In this block, we define the StatefulSet's `volumeClaimTemplates`. Kubernetes will use this to create PersistentVolumes for the Pods. In the block above, we name it `data` (which is the `name` we refer to in the `volumeMounts` defined previously), and give it the same `app: elasticsearch` label as our StatefulSet.

We then specify its access mode as `ReadWriteOnce`, which means that it can only be mounted as read-write by a single node. We define the storage class as `do-block-storage` in this guide since we use a DigitalOcean Kubernetes cluster for demonstration purposes. You should change this value depending on where you are running your Kubernetes cluster. To learn more, consult the Persistent Volume documentation.

Finally, we specify that we'd like each PersistentVolume to be 100GiB in size. You should adjust this value depending on your production needs.

The complete StatefulSet spec should look something like this:

elasticsearch_statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: es-cluster
  namespace: kube-logging
spec:
  serviceName: elasticsearch
  replicas: 3
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
    spec:
      containers:
      - name: elasticsearch
        image:
docker.elastic.co/elasticsearch/elasticsearch:7.2.
0
        resources:
            limits:
              cpu: 1000m
            requests:
              cpu: 100m
        ports:
        - containerPort: 9200
          name: rest
          protocol: TCP
        - containerPort: 9300
          name: inter-node
          protocol: TCP
        volumeMounts:
        - name: data
          mountPath: /usr/share/elasticsearch/data
        env:
          - name: cluster.name
            value: k8s-logs
          - name: node.name
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: discovery.seed_hosts
            value: "es-cluster-0.elasticsearch,es-
cluster-1.elasticsearch,es-cluster-
2.elasticsearch"
          - name: cluster.initial_master_nodes
            value: "es-cluster-0,es-cluster-1,es-
cluster-2"
          - name: ES_JAVA_OPTS
            value: "-Xms512m -Xmx512m"
      initContainers:
      - name: fix-permissions
        image: busybox
        command: ["sh", "-c", "chown -R 1000:1000
/usr/share/elasticsearch/data"]
        securityContext:
          privileged: true
        volumeMounts:
        - name: data
          mountPath: /usr/share/elasticsearch/data
      - name: increase-vm-max-map
        image: busybox
        command: ["sysctl", "-w",
"vm.max_map_count=262144"]
        securityContext:
          privileged: true
      - name: increase-fd-ulimit
        image: busybox
        command: ["sh", "-c", "ulimit -n 65536"]
        securityContext:
          privileged: true
  volumeClaimTemplates:
  - metadata:
      name: data
      labels:
        app: elasticsearch
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: do-block-storage
      resources:
        requests:
          storage: 100Gi
```

Once you’re satisfied with your Elasticsearch configuration, save and close the file.

Now, deploy the StatefulSet using `kubectl`:

```
kubectl create -f elasticsearch_statefulset.yaml
```

You should see the following output:

Output

```
statefulset.apps/es-cluster created
```

You can monitor the StatefulSet as it is rolled out using `kubectl rollout status`:

```
kubectl rollout status sts/es-cluster --
namespace=kube-logging
```

You should see the following output as the cluster is rolled out:

Output

```
Waiting for 3 pods to be ready...
Waiting for 2 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 3 new pods have
been updated...
```

Once all the Pods have been deployed, you can check that your Elasticsearch cluster is functioning correctly by performing a request against the REST API.

To do so, first forward the local port `9200` to the port `9200` on one of the Elasticsearch nodes (`es-cluster-0`) using `kubectl port-forward`:

```
kubectl port-forward es-cluster-0 9200:9200 --
namespace=kube-logging
```

Then, in a separate terminal window, perform a `curl` request against the REST API:

```
curl http://localhost:9200/_cluster/state?pretty
```

You shoulds see the following output:

Output

```
{
  "cluster_name" : "k8s-logs",
  "compressed_size_in_bytes" : 348,
  "cluster_uuid" : "QD06dK7CQgids-GQZooNVw",
  "version" : 3,
  "state_uuid" : "mjNIWXAzQVuxNNOQ7xR-qg",
  "master_node" : "IdM5B7cUQWqFgIHXBp0JDg",
  "blocks" : { },
  "nodes" : {
    "u7DoTpMmSCixOoictzHItA" : {
      "name" : "es-cluster-1",
      "ephemeral_id" : "ZlBflnXKRMC4RvEACHIVdg",
      "transport_address" : "10.244.8.2:9300",
      "attributes" : { }
    },
    "IdM5B7cUQWqFgIHXBp0JDg" : {
      "name" : "es-cluster-0",
      "ephemeral_id" : "JTk1FDdFQuWbSFAtBxdxAQ",
      "transport_address" : "10.244.44.3:9300",
      "attributes" : { }
    },
    "R8E7xcSUSbGbgrhAdyAKmQ" : {
      "name" : "es-cluster-2",
      "ephemeral_id" : "9wv6ke71Qqy9vk2LgJTqaA",
      "transport_address" : "10.244.40.4:9300",
      "attributes" : { }
    }
  },
...
```

This indicates that our Elasticsearch cluster `k8s-logs` has successfully been created with 3 nodes: `es-cluster-0`, `es-cluster-1`, and `es-cluster-2`. The current master node is `es-cluster-0`.

Now that your Elasticsearch cluster is up and running, you can move on to setting up a Kibana frontend for it.

Step 3 — Creating the Kibana Deployment and Service

To launch Kibana on Kubernetes, we'll create a Service called `kibana`, and a Deployment consisting of one Pod replica. You can scale the number of replicas depending on your production needs, and optionally specify a `LoadBalancer` type for the Service to load balance requests across the Deployment pods.

This time, we'll create the Service and Deployment in the same file. Open up a file called `kibana.yaml` in your favorite editor:

```
nano kibana.yaml
```

Paste in the following service spec:

kibana.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: kube-logging
  labels:
    app: kibana
spec:
  ports:
  - port: 5601
  selector:
    app: kibana
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  namespace: kube-logging
  labels:
    app: kibana
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
    spec:
      containers:
      - name: kibana
        image:
docker.elastic.co/kibana/kibana:7.2.0
        resources:
          limits:
            cpu: 1000m
          requests:
            cpu: 100m
        env:
          - name: ELASTICSEARCH_URL
            value: http://elasticsearch:9200
        ports:
        - containerPort: 5601
```

Then, save and close the file.

In this spec we've defined a service called `kibana` in the `kube-logging` namespace, and gave it the `app: kibana` label.

We've also specified that it should be accessible on port `5601` and use the `app: kibana` label to select the Service's target Pods.

In the `Deployment` spec, we define a Deployment called `kibana` and specify that we'd like 1 Pod replica.

We use the `docker.elastic.co/kibana/kibana:7.2.0` image. At this point you may substitute your own private or public Kibana image to use.

We specify that we'd like at the very least 0.1 vCPU guaranteed to the Pod, bursting up to a limit of 1 vCPU. You may change these parameters depending on your anticipated load and available resources.

Next, we use the `ELASTICSEARCH_URL` environment variable to set the endpoint and port for the Elasticsearch cluster. Using Kubernetes DNS, this endpoint corresponds to its Service name `elasticsearch`. This domain will resolve to a list of IP addresses for the 3 Elasticsearch Pods. To learn more about Kubernetes DNS, consult DNS for Services and Pods.

Finally, we set Kibana's container port to `5601`, to which the `kibana` Service will forward requests.

Once you're satisfied with your Kibana configuration, you can roll out the Service and Deployment using `kubectl`:

```
kubectl create -f kibana.yaml
```

You should see the following output:

Output

```
service/kibana created
deployment.apps/kibana created
```

You can check that the rollout succeeded by running the following command:

```
kubectl rollout status deployment/kibana --
namespace=kube-logging
```

You should see the following output:

Output

```
deployment "kibana" successfully rolled out
```

To access the Kibana interface, we’ll once again forward a local port to the Kubernetes node running Kibana. Grab the Kibana Pod details using `kubectl get`:

```
kubectl get pods --namespace=kube-logging
```

Output

```
NAME                          READY     STATUS
RESTARTS   AGE
es-cluster-0                  1/1       Running   0
55m
es-cluster-1                  1/1       Running   0
54m
es-cluster-2                  1/1       Running   0
54m
```

```
kibana-6c9fb4b5b7-plbg2   1/1       Running   0
4m27s
```

Here we observe that our Kibana Pod is called `kibana-6c9fb4b5b7-plbg2`.

Forward the local port `5601` to port `5601` on this Pod:

```
kubectl port-forward kibana-6c9fb4b5b7-plbg2
5601:5601 --namespace=kube-logging
```

You should see the following output:

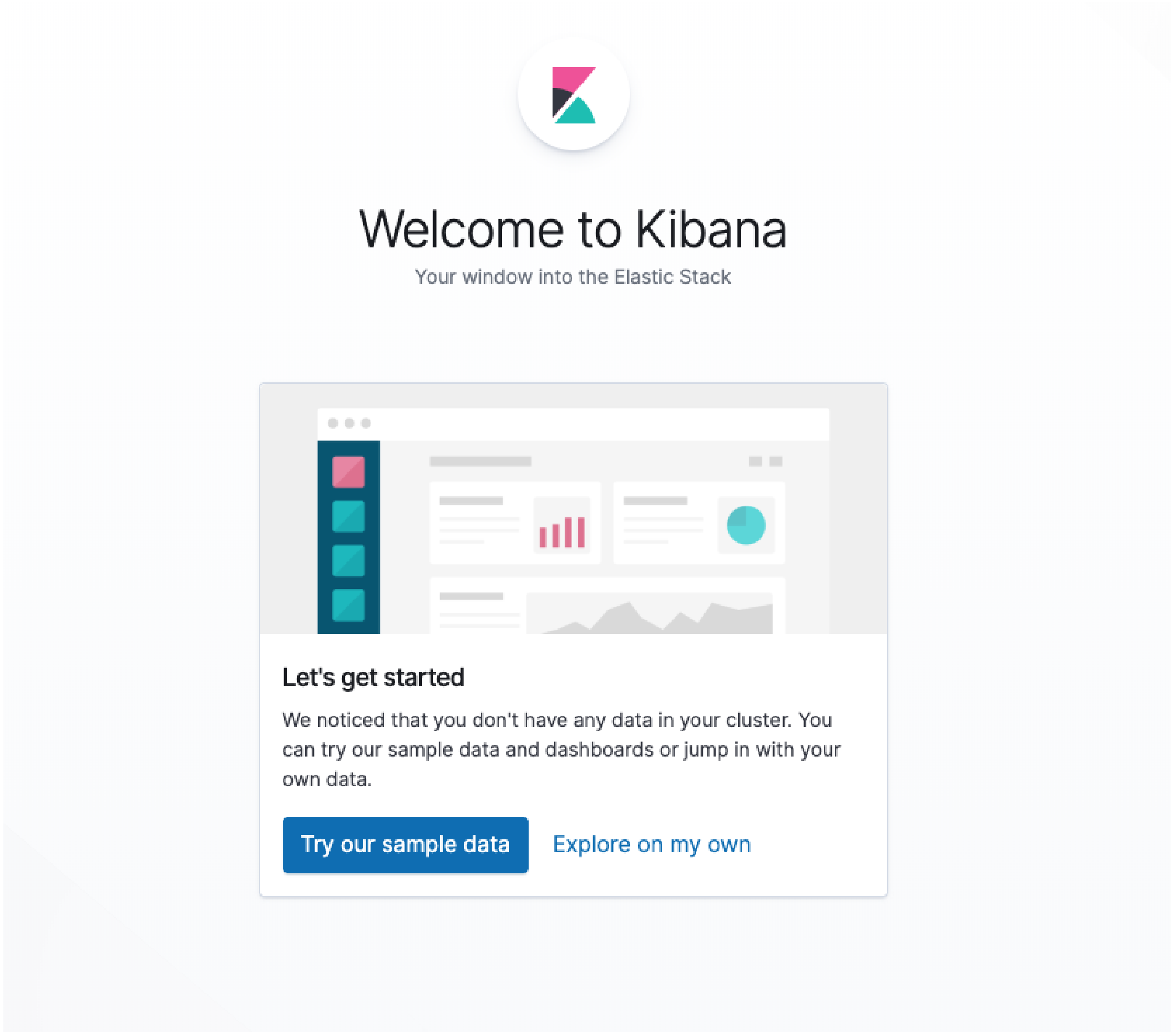
Output

```
Forwarding from 127.0.0.1:5601 -> 5601
Forwarding from [::1]:5601 -> 5601
```

Now, in your web browser, visit the following URL:

```
http://localhost:5601
```

If you see the following Kibana welcome page, you’ve successfully deployed Kibana into your Kubernetes cluster:


Welcome to Kibana

Your window into the Elastic Stack

Let's get started

We noticed that you don't have any data in your cluster. You can try our sample data and dashboards or jump in with your own data.

Kibana Welcome Screen

You can now move on to rolling out the final component of the EFK stack: the log collector, Fluentd.

Step 4 — Creating the Fluentd DaemonSet

In this guide, we'll set up Fluentd as a DaemonSet, which is a Kubernetes workload type that runs a copy of a given Pod on each Node in the Kubernetes cluster. Using this DaemonSet controller, we'll roll out a

Fluentd logging agent Pod on every node in our cluster. To learn more about this logging architecture, consult “Using a node logging agent” from the official Kubernetes docs.

In Kubernetes, containerized applications that log to `stdout` and `stderr` have their log streams captured and redirected to JSON files on the nodes. The Fluentd Pod will tail these log files, filter log events, transform the log data, and ship it off to the Elasticsearch logging backend we deployed in Step 2.

In addition to container logs, the Fluentd agent will tail Kubernetes system component logs like kubelet, kube-proxy, and Docker logs. To see a full list of sources tailed by the Fluentd logging agent, consult the `kubernetes.conf` file used to configure the logging agent. To learn more about logging in Kubernetes clusters, consult “Logging at the node level” from the official Kubernetes documentation.

Begin by opening a file called `fluentd.yaml` in your favorite text editor:

```
nano fluentd.yaml
```

Once again, we’ll paste in the Kubernetes object definitions block by block, providing context as we go along. In this guide, we use the Fluentd DaemonSet spec provided by the Fluentd maintainers. Another helpful resource provided by the Fluentd maintainers is Kuberentes Fluentd.

First, paste in the following ServiceAccount definition:

fluentd.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd
  namespace: kube-logging
  labels:
    app: fluentd
```

Here, we create a Service Account called `fluentd` that the Fluentd Pods will use to access the Kubernetes API. We create it in the `kube-logging` Namespace and once again give it the label `app: fluentd`. To learn more about Service Accounts in Kubernetes, consult Configure Service Accounts for Pods in the official Kubernetes docs.

Next, paste in the following `ClusterRole` block:

fluentd.yaml

```
. . .
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: fluentd
  labels:
    app: fluentd
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - namespaces
  verbs:
```

```
  - get
  - list
  - watch
```

Here we define a ClusterRole called `fluentd` to which we grant the `get`, `list`, and `watch` permissions on the `pods` and `namespaces` objects. ClusterRoles allow you to grant access to cluster-scoped Kubernetes resources like Nodes. To learn more about Role-Based Access Control and Cluster Roles, consult Using RBAC Authorization from the official Kubernetes documentation.

Now, paste in the following `ClusterRoleBinding` block:

fluentd.yaml

```
. . .
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: fluentd
roleRef:
  kind: ClusterRole
  name: fluentd
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: fluentd
  namespace: kube-logging
```

In this block, we define a `ClusterRoleBinding` called `fluentd` which binds the `fluentd` ClusterRole to the `fluentd` Service Account. This grants the `fluentd` ServiceAccount the permissions listed in the `fluentd` Cluster Role.

At this point we can begin pasting in the actual DaemonSet spec:

fluentd.yaml

```
. . .
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-logging
  labels:
    app: fluentd
```

Here, we define a DaemonSet called `fluentd` in the `kube-logging` Namespace and give it the `app: fluentd` label.

Next, paste in the following section:

fluentd.yaml

```
. . .
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      serviceAccount: fluentd
      serviceAccountName: fluentd
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd
        image: fluent/fluentd-kubernetes-
daemonset:v1.4.2-debian-elasticsearch-1.1
        env:
          - name:  FLUENT_ELASTICSEARCH_HOST
            value: "elasticsearch.kube-
logging.svc.cluster.local"
          - name:  FLUENT_ELASTICSEARCH_PORT
            value: "9200"
          - name: FLUENT_ELASTICSEARCH_SCHEME
            value: "http"
          - name: FLUENTD_SYSTEMD_CONF
            value: disable
```

Here, we match the `app: fluentd` label defined in `.metadata.labels` and then assign the DaemonSet the `fluentd` Service Account. We also select the `app: fluentd` as the Pods managed by this DaemonSet.

Next, we define a `NoSchedule` toleration to match the equivalent taint on Kubernetes master nodes. This will ensure that the DaemonSet also gets rolled out to the Kubernetes masters. If you don't want to run a Fluentd Pod on your master nodes, remove this toleration. To learn more about Kubernetes taints and tolerations, consult "Taints and Tolerations" from the official Kubernetes docs.

Next, we begin defining the Pod container, which we call `fluentd`.

We use the official v1.4.2 Debian image provided by the Fluentd maintainers. If you'd like to use your own private or public Fluentd image, or use a different image version, modify the `image` tag in the container spec. The Dockerfile and contents of this image are available in Fluentd's fluentd-kubernetes-daemonset Github repo.

Next, we configure Fluentd using some environment variables:

- `FLUENT_ELASTICSEARCH_HOST`: We set this to the Elasticsearch headless Service address defined earlier: `elasticsearch.kube-logging.svc.cluster.local`. This will resolve to a list of IP addresses for the 3 Elasticsearch Pods. The actual Elasticsearch host will most likely be the first IP address returned in this list. To distribute logs across the cluster, you will need to modify the configuration for Fluentd's Elasticsearch Output plugin. To learn more about this plugin, consult Elasticsearch Output Plugin.
- `FLUENT_ELASTICSEARCH_PORT`: We set this to the Elasticsearch port we configured earlier, `9200`.
- `FLUENT_ELASTICSEARCH_SCHEME`: We set this to `http`.
- `FLUENTD_SYSTEMD_CONF`: We set this to `disable` to suppress output related to `systemd` not being set up in the container.

Finally, paste in the following section:

fluentd.yaml

```
. . .
        resources:
          limits:
            memory: 512Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Here we specify a 512 MiB memory limit on the FluentD Pod, and guarantee it 0.1vCPU and 200MiB of memory. You can tune these resource limits and requests depending on your anticipated log volume and available resources.

Next, we mount the `/var/log` and `/var/lib/docker/containers` host paths into the container using the `varlog` and `varlibdockercontainers` `volumeMounts`. These `volumes` are defined at the end of the block.

The final parameter we define in this block is `terminationGracePeriodSeconds`, which gives Fluentd 30 seconds to shut down gracefully upon receiving a `SIGTERM` signal. After 30 seconds, the containers are sent a `SIGKILL` signal. The default value for `terminationGracePeriodSeconds` is 30s, so in most cases this parameter can be omitted. To learn more about gracefully terminating Kubernetes workloads, consult Google's "Kubernetes best practices: terminating with grace."

The entire Fluentd spec should look something like this:

fluentd.yaml
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd
  namespace: kube-logging
  labels:
    app: fluentd
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```yaml
metadata:
  name: fluentd
  labels:
    app: fluentd
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - namespaces
  verbs:
  - get
  - list
  - watch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: fluentd
roleRef:
  kind: ClusterRole
  name: fluentd
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: fluentd
  namespace: kube-logging
```

```
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-logging
  labels:
    app: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      serviceAccount: fluentd
      serviceAccountName: fluentd
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd
        image: fluent/fluentd-kubernetes-
daemonset:v1.4.2-debian-elasticsearch-1.1
        env:
          - name:  FLUENT_ELASTICSEARCH_HOST
            value: "elasticsearch.kube-
logging.svc.cluster.local"
          - name:  FLUENT_ELASTICSEARCH_PORT
            value: "9200"
          - name: FLUENT_ELASTICSEARCH_SCHEME
            value: "http"
          - name: FLUENTD_SYSTEMD_CONF
            value: disable
        resources:
          limits:
            memory: 512Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
```

```
        hostPath:
          path: /var/lib/docker/containers
```

Once you've finished configuring the Fluentd DaemonSet, save and close the file.

Now, roll out the DaemonSet using `kubectl`:

```
kubectl create -f fluentd.yaml
```

You should see the following output:

Output

```
serviceaccount/fluentd created
clusterrole.rbac.authorization.k8s.io/fluentd
created
clusterrolebinding.rbac.authorization.k8s.io/fluen
td created
daemonset.extensions/fluentd created
```

Verify that your DaemonSet rolled out successfully using `kubectl`:

```
kubectl get ds --namespace=kube-logging
```

You should see the following status output:

Output

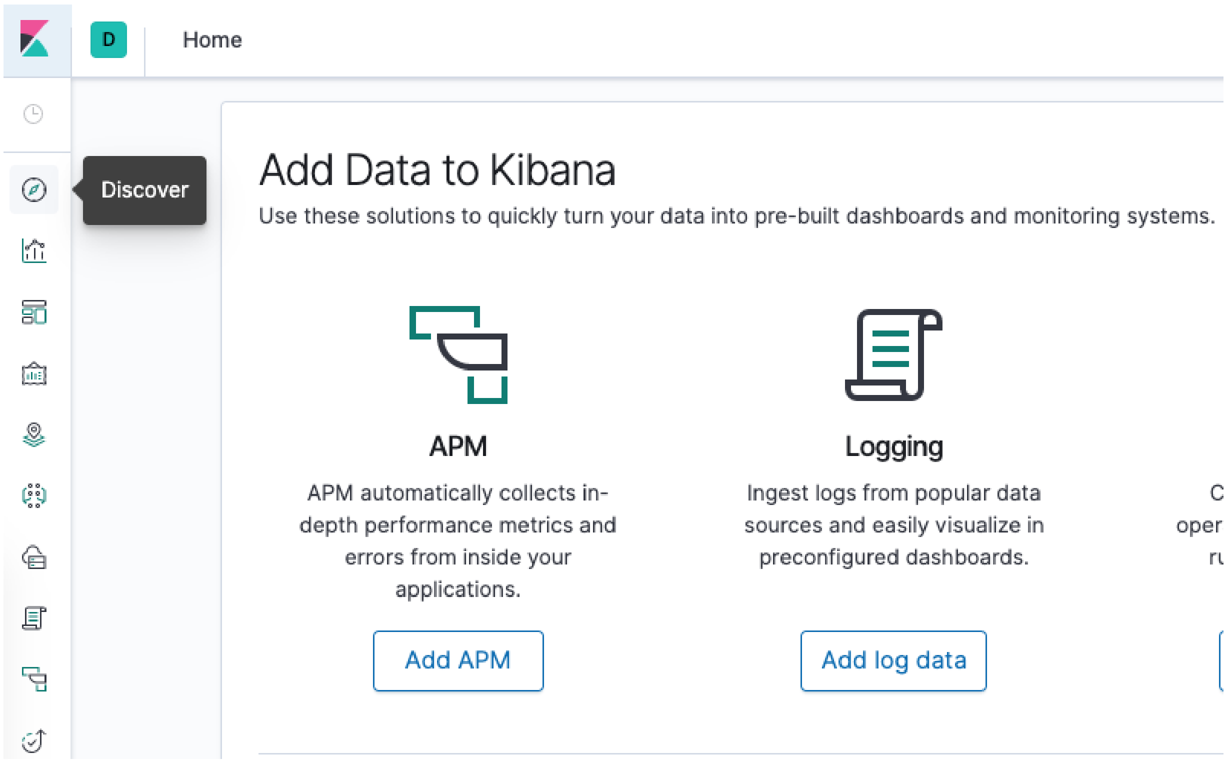
```
NAME      DESIRED   CURRENT   READY     UP-TO-DATE
AVAILABLE   NODE SELECTOR   AGE
fluentd   3         3         3         3
3           <none>          58s
```

This indicates that there are 3 `fluentd` Pods running, which corresponds to the number of nodes in our Kubernetes cluster.

We can now check Kibana to verify that log data is being properly collected and shipped to Elasticsearch.

With the `kubectl port-forward` still open, navigate to `http://localhost:5601`.

Click on Discover in the left-hand navigation menu:



Add Data to Kibana

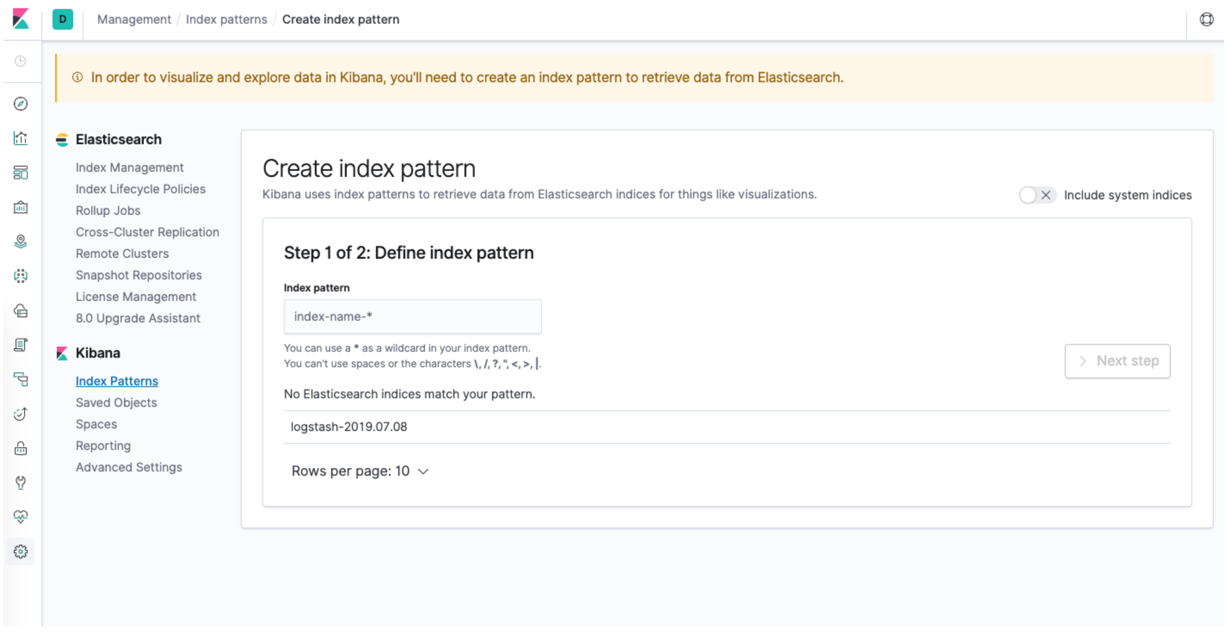
Use these solutions to quickly turn your data into pre-built dashboards and monitoring systems.

APM automatically collects in-depth performance metrics and errors from inside your applications.

Ingest logs from popular data sources and easily visualize in preconfigured dashboards.

Kibana Discover

You should see the following configuration window:



Management / Index patterns / Create index pattern

In order to visualize and explore data in Kibana, you'll need to create an index pattern to retrieve data from Elasticsearch.

Create index pattern

Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.

Include system indices

Step 1 of 2: Define index pattern

Index pattern: index-name-*

You can use a * as a wildcard in your index pattern.
You can't use spaces or the characters \, /, ?, ", <, >, |.

No Elasticsearch indices match your pattern.

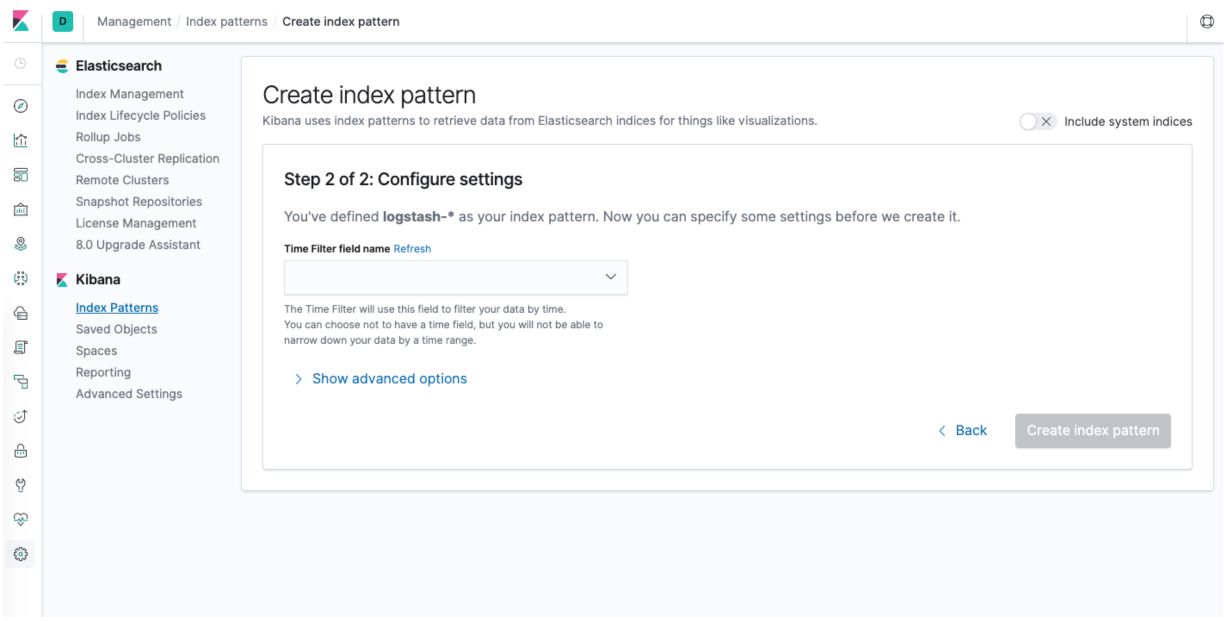
logstash-2019.07.08

Rows per page: 10

Kibana Index Pattern Configuration

This allows you to define the Elasticsearch indices you'd like to explore in Kibana. To learn more, consult Defining your index patterns in the official Kibana docs. For now, we'll just use the `logstash-*` wildcard pattern to capture all the log data in our Elasticsearch cluster. Enter `logstash-*` in the text box and click on Next step.

You'll then be brought to the following page:



Management / Index patterns / Create index pattern

Elasticsearch: Index Management, Index Lifecycle Policies, Rollup Jobs, Cross-Cluster Replication, Remote Clusters, Snapshot Repositories, License Management, 8.0 Upgrade Assistant

Kibana: Index Patterns, Saved Objects, Spaces, Reporting, Advanced Settings

Create index pattern

Kibana uses index patterns to retrieve data from Elasticsearch indices for things like visualizations.

Include system indices

Step 2 of 2: Configure settings

You've defined **logstash-*** as your index pattern. Now you can specify some settings before we create it.

Time Filter field name Refresh

The Time Filter will use this field to filter your data by time.
You can choose not to have a time field, but you will not be able to narrow down your data by a time range.

Show advanced options

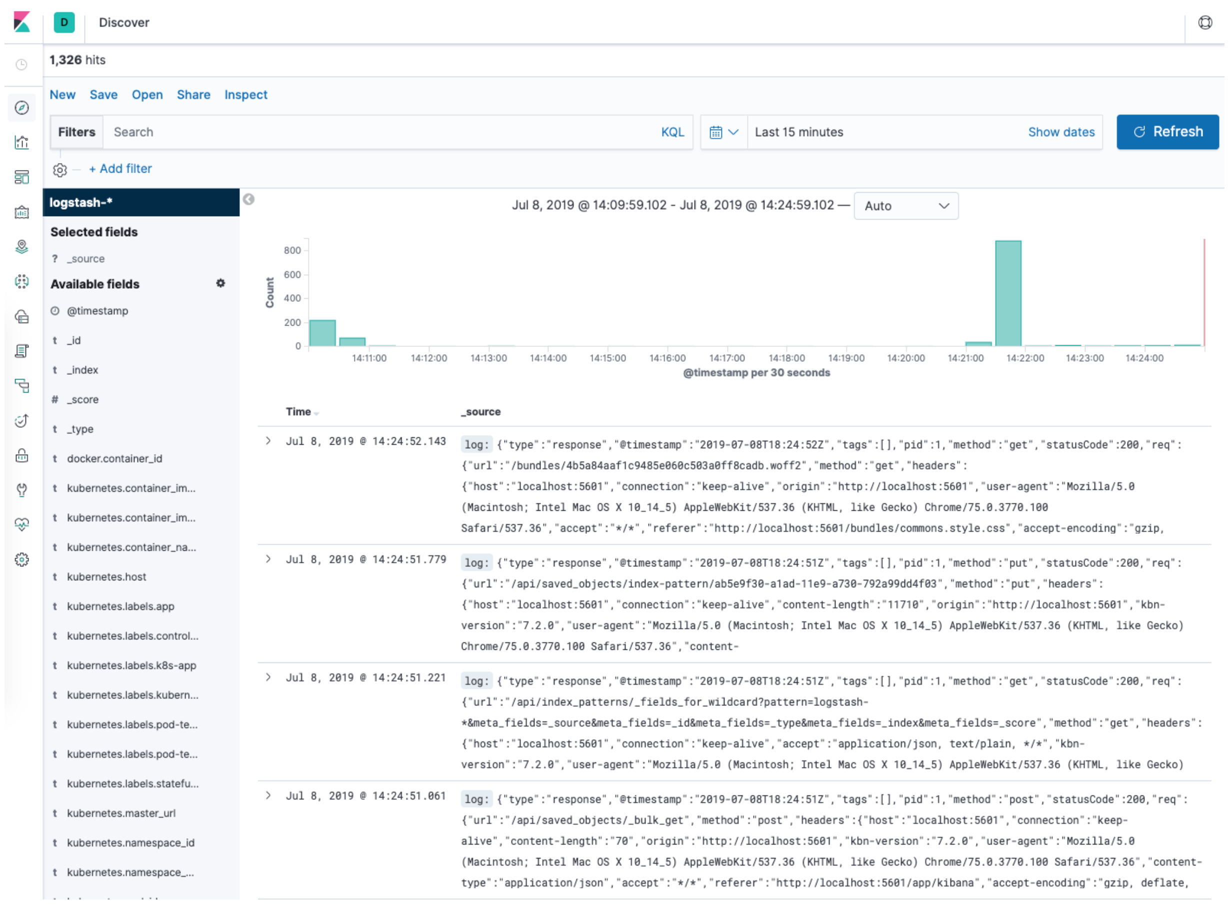
Back | Create index pattern

Kibana Index Pattern Settings

This allows you to configure which field Kibana will use to filter log data by time. In the dropdown, select the @timestamp field, and hit Create index pattern.

Now, hit Discover in the left hand navigation menu.

You should see a histogram graph and some recent log entries:


Discover
1,326 hits
New Save Open Share Inspect
Filters Search KQL Last 15 minutes Show dates Refresh
+ Add filter
logstash-*
Selected fields
_source
Available fields
@timestamp
_id
_index
_score
_type
docker.container_id
kubernetes.container_im...
kubernetes.container_im...
kubernetes.container_na...
kubernetes.host
kubernetes.labels.app
kubernetes.labels.control...
kubernetes.labels.k8s-app
kubernetes.labels.kubern...
kubernetes.labels.pod-te...
kubernetes.labels.pod-te...
kubernetes.labels.statefu...
kubernetes.master_url
kubernetes.namespace_id
kubernetes.namespace_...
Jul 8, 2019 @ 14:09:59.102 - Jul 8, 2019 @ 14:24:59.102 — Auto
Count
@timestamp per 30 seconds

```
Time            _source
Jul 8, 2019 @ 14:24:52.143  log: {"type":"response","@timestamp":"2019-07-08T18:24:52Z","tags":[],"pid":1,"method":"get","statusCode":200,"req":
{"url":"/bundles/4b5a84aaf1c9485e060c503a0ff8cadb.woff2","method":"get","headers":
{"host":"localhost:5601","connection":"keep-alive","origin":"http://localhost:5601","user-agent":"Mozilla/5.0
(Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.100
Safari/537.36","accept":"*/*","referer":"http://localhost:5601/bundles/commons.style.css","accept-encoding":"gzip,
Jul 8, 2019 @ 14:24:51.779  log: {"type":"response","@timestamp":"2019-07-08T18:24:51Z","tags":[],"pid":1,"method":"put","statusCode":200,"req":
{"url":"/api/saved_objects/index-pattern/ab5e9f30-a1ad-11e9-a730-792a99dd4f03","method":"put","headers":
{"host":"localhost:5601","connection":"keep-alive","content-length":"11710","origin":"http://localhost:5601","kbn-
version":"7.2.0","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/75.0.3770.100 Safari/537.36","content-
Jul 8, 2019 @ 14:24:51.221  log: {"type":"response","@timestamp":"2019-07-08T18:24:51Z","tags":[],"pid":1,"method":"get","statusCode":200,"req":
{"url":"/api/index_patterns/_fields_for_wildcard?pattern=logstash-
*&meta_fields=_source&meta_fields=_id&meta_fields=_type&meta_fields=_index&meta_fields=_score","method":"get","headers":
{"host":"localhost:5601","connection":"keep-alive","accept":"application/json, text/plain, */*","kbn-
version":"7.2.0","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko)
Jul 8, 2019 @ 14:24:51.061  log: {"type":"response","@timestamp":"2019-07-08T18:24:51Z","tags":[],"pid":1,"method":"post","statusCode":200,"req":
{"url":"/api/saved_objects/_bulk_get","method":"post","headers":{"host":"localhost:5601","connection":"keep-
alive","content-length":"70","origin":"http://localhost:5601","kbn-version":"7.2.0","user-agent":"Mozilla/5.0
(Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.100 Safari/537.36","content-
type":"application/json","accept":"*/*","referer":"http://localhost:5601/app/kibana","accept-encoding":"gzip, deflate,
```

Kibana Incoming Logs

At this point you've successfully configured and rolled out the EFK stack on your Kubernetes cluster. To learn how to use Kibana to analyze your log data, consult the Kibana User Guide.

In the next optional section, we'll deploy a simple counter Pod that prints numbers to stdout, and find its logs in Kibana.

Step 5 (Optional) — Testing Container Logging

To demonstrate a basic Kibana use case of exploring the latest logs for a given Pod, we'll deploy a minimal counter Pod that prints sequential numbers to stdout.

Let's begin by creating the Pod. Open up a file called `counter.yaml` in your favorite editor:

```
nano counter.yaml
```

Then, paste in the following Pod spec:

counter.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
            'i=0; while true; do echo "$i:
$(date)"; i=$((i+1)); sleep 1; done']
```

Save and close the file.

This is a minimal Pod called `counter` that runs a `while` loop, printing numbers sequentially.

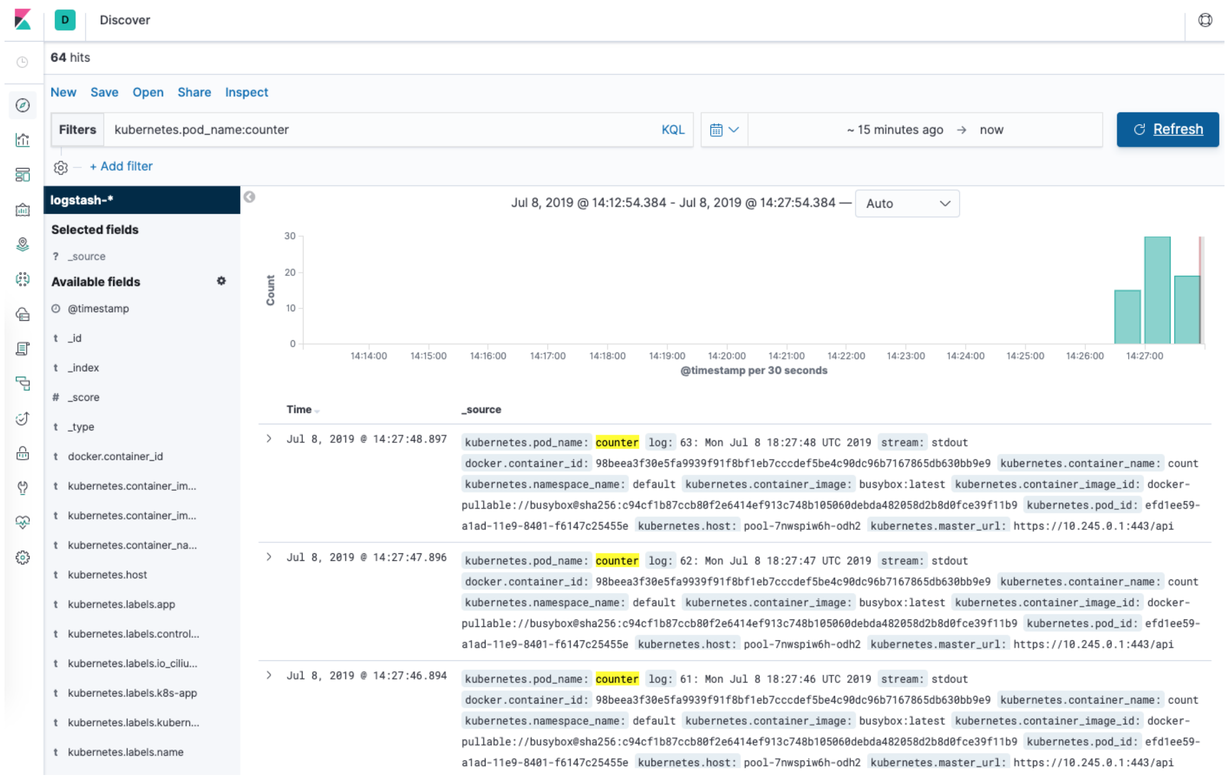
Deploy the `counter` Pod using `kubectl`:

```
kubectl create -f counter.yaml
```

Once the Pod has been created and is running, navigate back to your Kibana dashboard.

From the Discover page, in the search bar enter `kubernetes.pod_name:counter`. This filters the log data for Pods named `counter`.

You should then see a list of log entries for the `counter` Pod:



```
kubernetes.pod_name:counter
Jul 8, 2019 @ 14:12:54.384 - Jul 8, 2019 @ 14:27:54.384

Jul 8, 2019 @ 14:27:48.897  kubernetes.pod_name: counter log: 63: Mon Jul 8 18:27:48 UTC 2019 stream: stdout
docker.container_id: 98beea3f30e5fa9939f91f8bf1eb7cccdef5be4c90dc96b7167865db630bb9e9 kubernetes.container_name: count
kubernetes.namespace_name: default kubernetes.container_image: busybox:latest kubernetes.container_image_id: docker-
pullable://busybox@sha256:c94cf1b87ccb80f2e6414ef913c748b105060debda482058d2b8d0fce39f11b9 kubernetes.pod_id: efd1ee59-
a1ad-11e9-8401-f6147c25455e kubernetes.host: pool-7nwspiw6h-odh2 kubernetes.master_url: https://10.245.0.1:443/api

Jul 8, 2019 @ 14:27:47.896  kubernetes.pod_name: counter log: 62: Mon Jul 8 18:27:47 UTC 2019 stream: stdout
docker.container_id: 98beea3f30e5fa9939f91f8bf1eb7cccdef5be4c90dc96b7167865db630bb9e9 kubernetes.container_name: count
kubernetes.namespace_name: default kubernetes.container_image: busybox:latest kubernetes.container_image_id: docker-
pullable://busybox@sha256:c94cf1b87ccb80f2e6414ef913c748b105060debda482058d2b8d0fce39f11b9 kubernetes.pod_id: efd1ee59-
a1ad-11e9-8401-f6147c25455e kubernetes.host: pool-7nwspiw6h-odh2 kubernetes.master_url: https://10.245.0.1:443/api

Jul 8, 2019 @ 14:27:46.894  kubernetes.pod_name: counter log: 61: Mon Jul 8 18:27:46 UTC 2019 stream: stdout
docker.container_id: 98beea3f30e5fa9939f91f8bf1eb7cccdef5be4c90dc96b7167865db630bb9e9 kubernetes.container_name: count
kubernetes.namespace_name: default kubernetes.container_image: busybox:latest kubernetes.container_image_id: docker-
pullable://busybox@sha256:c94cf1b87ccb80f2e6414ef913c748b105060debda482058d2b8d0fce39f11b9 kubernetes.pod_id: efd1ee59-
a1ad-11e9-8401-f6147c25455e kubernetes.host: pool-7nwspiw6h-odh2 kubernetes.master_url: https://10.245.0.1:443/api
```

Counter Logs in Kibana

You can click into any of the log entries to see additional metadata like the container name, Kubernetes node, Namespace, and more.

Conclusion

In this guide we've demonstrated how to set up and configure Elasticsearch, Fluentd, and Kibana on a Kubernetes cluster. We've used a minimal logging architecture that consists of a single logging agent Pod running on each Kubernetes worker node.

Before deploying this logging stack into your production Kubernetes cluster, it's best to tune the resource requirements and limits as indicated throughout this guide. You may also want to set up X-Pack to enable built-in monitoring and security features.

The logging architecture we've used here consists of 3 Elasticsearch Pods, a single Kibana Pod (not load-balanced), and a set of Fluentd Pods rolled out as a DaemonSet. You may wish to scale this setup depending on your production use case. To learn more about scaling your Elasticsearch and Kibana stack, consult Scaling Elasticsearch.

Kubernetes also allows for more complex logging agent architectures that may better suit your use case. To learn more, consult Logging Architecture from the Kubernetes docs.

How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes

Written by Hanif Jetha

Kubernetes Ingresses allow you to flexibly route traffic from outside your Kubernetes cluster to Services inside of your cluster. This routing is accomplished using Ingress Resources, which define rules for routing HTTP and HTTPS traffic to Kubernetes Services, and Ingress Controllers, which implement the rules by load balancing traffic and routing it to the appropriate backend Services.

In this guide, you will set up the Kubernetes-maintained Nginx Ingress Controller, and create some Ingress Resources to route traffic to several dummy backend services. Once the Ingress is in place, you will install cert-manager into your cluster to manage and provision TLS certificates using Let's Encrypt for encrypting web traffic to your applications.

Kubernetes Ingresses allow you to flexibly route traffic from outside your Kubernetes cluster to Services inside of your cluster. This is accomplished using Ingress Resources, which define rules for routing HTTP and HTTPS traffic to Kubernetes Services, and Ingress Controllers, which implement the rules by load balancing traffic and routing it to the appropriate backend Services. Popular Ingress Controllers include Nginx, Contour, HAProxy, and Traefik. Ingresses provide a more efficient and flexible alternative to setting up multiple LoadBalancer services, each of which uses its own dedicated Load Balancer.

In this guide, we'll set up the Kubernetes-maintained Nginx Ingress Controller, and create some Ingress Resources to route traffic to several dummy backend services. Once we've set up the Ingress, we'll install cert-manager into our cluster to manage and provision TLS certificates for encrypting HTTP traffic to the Ingress. This guide does not use the Helm package manager. For a guide on rolling out the Nginx Ingress Controller using Helm, consult How To Set Up an Nginx Ingress on DigitalOcean Kubernetes Using Helm.

Prerequisites

Before you begin with this guide, you should have the following available to you:

- A Kubernetes 1.10+ cluster with role-based access control (RBAC) enabled
- The `kubectl` command-line tool installed on your local machine and configured to connect to your cluster. You can read more about installing `kubectl` in the official documentation.
- A domain name and DNS A records which you can point to the DigitalOcean Load Balancer used by the Ingress. If you are using DigitalOcean to manage your domain's DNS records, consult How to Manage DNS Records to learn how to create A records.
- The `wget` command-line utility installed on your local machine. You can install `wget` using the package manager built into your operating system.

Once you have these components set up, you're ready to begin with this guide.

Step 1 — Setting Up Dummy Backend Services

Before we deploy the Ingress Controller, we'll first create and roll out two dummy echo Services to which we'll route external traffic using the Ingress. The echo Services will run the `hashicorp/http-echo` web server container, which returns a page containing a text string passed in when the web server is launched. To learn more about `http-echo`, consult its GitHub Repo, and to learn more about Kubernetes Services, consult Services from the official Kubernetes docs.

On your local machine, create and edit a file called `echo1.yaml` using `nano` or your favorite editor:

```
nano echo1.yaml
```

Paste in the following Service and Deployment manifest:

echo1.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: echo1
spec:
  ports:
  - port: 80
    targetPort: 5678
  selector:
    app: echo1
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo1
spec:
  selector:
    matchLabels:
      app: echo1
  replicas: 2
  template:
    metadata:
      labels:
        app: echo1
    spec:
      containers:
      - name: echo1
        image: hashicorp/http-echo
        args:
        - "-text=echo1"
        ports:
        - containerPort: 5678
```

In this file, we define a Service called `echo1` which routes traffic to Pods with the `app: echo1` label selector. It accepts TCP traffic on port `80` and routes it to port `5678`,`http-echo`'s default port.

We then define a Deployment, also called `echo1`, which manages Pods with the `app: echo1` Label Selector. We specify that the Deployment should have 2 Pod replicas, and that the Pods should start a container called `echo1` running the `hashicorp/http-echo` image. We pass in the `text` parameter and set it to `echo1`, so that the `http-echo` web server returns `echo1`. Finally, we open port `5678` on the Pod container.

Once you're satisfied with your dummy Service and Deployment manifest, save and close the file.

Then, create the Kubernetes resources using `kubectl apply` with the `-f` flag, specifying the file you just saved as a parameter:

```
kubectl apply -f echo1.yaml
```

You should see the following output:

Output

```
service/echo1 created
deployment.apps/echo1 created
```

Verify that the Service started correctly by confirming that it has a ClusterIP, the internal IP on which the Service is exposed:

```
kubectl get svc echo1
```

You should see the following output:

Output

```
NAME       TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)    AGE
echo1      ClusterIP   10.245.222.129   <none>
80/TCP     60s
```

This indicates that the `echo1` Service is now available internally at `10.245.222.129` on port `80`. It will forward traffic to containerPort `5678` on the Pods it selects.

Now that the `echo1` Service is up and running, repeat this process for the `echo2` Service.

Create and open a file called `echo2.yaml`:

echo2.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: echo2
spec:
  ports:
  - port: 80
    targetPort: 5678
  selector:
    app: echo2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo2
spec:
  selector:
    matchLabels:
      app: echo2
  replicas: 1
  template:
    metadata:
      labels:
        app: echo2
    spec:
      containers:
      - name: echo2
        image: hashicorp/http-echo
        args:
        - "-text=echo2"
        ports:
        - containerPort: 5678
```

Here, we essentially use the same Service and Deployment manifest as above, but name and relabel the Service and Deployment `echo2`. In addition, to provide some variety, we create only 1 Pod replica. We ensure that we set the `text` parameter to `echo2` so that the web server returns the text `echo2`.

Save and close the file, and create the Kubernetes resources using `kubectl`:

```
kubectl apply -f echo2.yaml
```

You should see the following output:

Output

```
service/echo2 created
deployment.apps/echo2 created
```

Once again, verify that the Service is up and running:

```
kubectl get svc
```

You should see both the `echo1` and `echo2` Services with assigned ClusterIPs:

Output

```
NAME          TYPE        CLUSTER-IP
EXTERNAL-IP    PORT(S)    AGE
echo1         ClusterIP   10.245.222.129   <none>
80/TCP     6m6s
echo2         ClusterIP   10.245.128.224   <none>
80/TCP     6m3s
kubernetes    ClusterIP   10.245.0.1       <none>
443/TCP    4d21h
```

Now that our dummy echo web services are up and running, we can move on to rolling out the Nginx Ingress Controller.

Step 2 — Setting Up the Kubernetes Nginx Ingress Controller

In this step, we'll roll out **`v0.26.1`** of the Kubernetes-maintained Nginx Ingress Controller. Note that there are several Nginx Ingress Controllers; the Kubernetes community maintains the one used in this guide and Nginx Inc. maintains kubernetes-ingress. The instructions in this tutorial are based on those from the official Kubernetes Nginx Ingress Controller Installation Guide.

The Nginx Ingress Controller consists of a Pod that runs the Nginx web server and watches the Kubernetes Control Plane for new and updated Ingress Resource objects. An Ingress Resource is essentially a list of traffic routing rules for backend Services. For example, an Ingress rule can specify that HTTP traffic arriving at the path `/web1` should be directed towards the `web1` backend web server. Using Ingress Resources, you can also perform host-based routing: for example, routing requests that hit `web1.your_domain.com` to the backend Kubernetes Service `web1`.

In this case, because we're deploying the Ingress Controller to a DigitalOcean Kubernetes cluster, the Controller will create a LoadBalancer Service that spins up a DigitalOcean Load Balancer to which all external traffic will be directed. This Load Balancer will route external traffic to the Ingress Controller Pod running Nginx, which then forwards traffic to the appropriate backend Services.

We'll begin by first creating the Kubernetes resources required by the Nginx Ingress Controller. These consist of ConfigMaps containing the Controller's configuration, Role-based Access Control (RBAC) Roles to grant the Controller access to the Kubernetes API, and the actual Ingress Controller Deployment which uses v0.26.1 of the Nginx Ingress Controller image. To see a full list of these required resources, consult the manifest from the Kubernetes Nginx Ingress Controller's GitHub repo.

To create these mandatory resources, use `kubectl apply` and the `-f` flag to specify the manifest file hosted on GitHub:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingre
ss-nginx/nginx-0.26.1/deploy/static/mandatory.yaml
```

We use `apply` here so that in the future we can incrementally `apply` changes to the Ingress Controller objects instead of completely overwriting them. To learn more about `apply`, consult Managing Resources from the official Kubernetes docs.

You should see the following output:

Output

```
namespace/ingress-nginx created
configmap/nginx-configuration created
configmap/tcp-services created
configmap/udp-services created
serviceaccount/nginx-ingress-serviceaccount
created
clusterrole.rbac.authorization.k8s.io/nginx-
ingress-clusterrole created
role.rbac.authorization.k8s.io/nginx-ingress-role
created
rolebinding.rbac.authorization.k8s.io/nginx-
ingress-role-nisa-binding created
clusterrolebinding.rbac.authorization.k8s.io/nginx
-ingress-clusterrole-nisa-binding created
deployment.apps/nginx-ingress-controller created
```

This output also serves as a convenient summary of all the Ingress Controller objects created from the `mandatory.yaml` manifest.

Next, we'll create the Ingress Controller LoadBalancer Service, which will create a DigitalOcean Load Balancer that will load balance and route HTTP and HTTPS traffic to the Ingress Controller Pod deployed in the previous command.

To create the LoadBalancer Service, once again `kubectl apply` a manifest file containing the Service definition:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingre
```

```
ss-nginx/nginx-
0.26.1/deploy/static/provider/cloud-generic.yaml
```

You should see the following output:

Output

```
service/ingress-nginx created
```

Confirm that the Ingress Controller Pods have started:

```
kubectl get pods --all-namespaces -l
app.kubernetes.io/name=ingress-nginx
```

Output

```
NAMESPACE       NAME
READY   STATUS    RESTARTS   AGE
ingress-nginx   nginx-ingress-controller-
7fb85bc8bb-lnm6z   1/1     Running   0
2m42s
```

Now, confirm that the DigitalOcean Load Balancer was successfully created by fetching the Service details with `kubectl`:

```
kubectl get svc --namespace=ingress-nginx
```

After several minutes, you should see an external IP address, corresponding to the IP address of the DigitalOcean Load Balancer:

Output

```
NAME            TYPE           CLUSTER-IP
EXTERNAL-IP       PORT(S)                         AGE
ingress-nginx   LoadBalancer   10.245.247.67
203.0.113.0    80:32486/TCP,443:32096/TCP   20h
```

Note down the Load Balancer's external IP address, as you'll need it in a later step.

Note: By default the Nginx Ingress LoadBalancer Service has `service.spec.externalTrafficPolicy` set to the value `Local`, which routes all load balancer traffic to nodes running Nginx Ingress Pods. The other nodes will deliberately fail load balancer health checks so that Ingress traffic does not get routed to them. External traffic policies are beyond the scope of this tutorial, but to learn more you can consult A Deep Dive into Kubernetes External Traffic Policies and Source IP for Services with Type=LoadBalancer from the official Kubernetes docs.

This load balancer receives traffic on HTTP and HTTPS ports 80 and 443, and forwards it to the Ingress Controller Pod. The Ingress Controller will then route the traffic to the appropriate backend Service.

We can now point our DNS records at this external Load Balancer and create some Ingress Resources to implement traffic routing rules.

Step 3 — Creating the Ingress Resource

Let's begin by creating a minimal Ingress Resource to route traffic directed at a given subdomain to a corresponding backend Service.

In this guide, we'll use the test domain example.com. You should substitute this with the domain name you own.

We'll first create a simple rule to route traffic directed at echo1.**example.com** to the `echo1` backend service and traffic directed at echo2.**example.com** to the `echo2` backend service.

Begin by opening up a file called `echo_ingress.yaml` in your favorite editor:

```
nano echo_ingress.yaml
```

Paste in the following ingress definition:

echo_ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: echo-ingress
spec:
  rules:
  - host: echo1.example.com
    http:
      paths:
      - backend:
          serviceName: echo1
          servicePort: 80
  - host: echo2.example.com
    http:
      paths:
      - backend:
          serviceName: echo2
          servicePort: 80
```

When you've finished editing your Ingress rules, save and close the file.

Here, we've specified that we'd like to create an Ingress Resource called `echo-ingress`, and route traffic based on the Host header. An HTTP request Host header specifies the domain name of the target server. To learn more about Host request headers, consult the Mozilla Developer

Network definition page. Requests with host echo1.**example.com** will be directed to the `echo1` backend set up in Step 1, and requests with host echo2.**example.com** will be directed to the `echo2` backend.

You can now create the Ingress using `kubectl`:

```
kubectl apply -f echo_ingress.yaml
```

You'll see the following output confirming the Ingress creation:

Output

```
ingress.extensions/echo-ingress created
```

To test the Ingress, navigate to your DNS management service and create A records for `echo1.example.com` and `echo2.example.com` pointing to the DigitalOcean Load Balancer's external IP. The Load Balancer's external IP is the external IP address for the `ingress-nginx` Service, which we fetched in the previous step. If you are using DigitalOcean to manage your domain's DNS records, consult How to Manage DNS Records to learn how to create A records.

Once you've created the necessary `echo1.example.com` and `echo2.example.com` DNS records, you can test the Ingress Controller and Resource you've created using the `curl` command line utility.

From your local machine, `curl` the `echo1` Service:

```
curl echo1.example.com
```

You should get the following response from the `echo1` service:

Output

```
echo1
```

This confirms that your request to `echo1.example.com` is being correctly routed through the Nginx ingress to the `echo1` backend Service.

Now, perform the same test for the `echo2` Service:

```
curl echo2.example.com
```

You should get the following response from the `echo2` Service:

Output

```
echo2
```

This confirms that your request to `echo2.example.com` is being correctly routed through the Nginx ingress to the `echo2` backend Service.

At this point, you've successfully set up a minimal Nginx Ingress to perform virtual host-based routing. In the next step, we'll install cert-manager to provision TLS certificates for our Ingress and enable the more secure HTTPS protocol.

Step 4 — Installing and Configuring Cert-Manager

In this step, we'll install cert-manager into our cluster. cert-manager is a Kubernetes service that provisions TLS certificates from Let's Encrypt and other certificate authorities and manages their lifecycles. Certificates can be requested and configured by annotating Ingress Resources with the `cert-manager.io/issuer` annotation, appending a `tls` section to the Ingress spec, and configuring one or more Issuers or ClusterIssuers to specify your preferred certificate authority. To learn more about Issuer and ClusterIssuer objects, consult the official cert-manager documentation on Issuers.

Before we install cert-manager, we'll first create a Namespace for it to run in:

```
kubectl create namespace cert-manager
```

Next, we'll install cert-manager and its Custom Resource Definitions (CRDs) like Issuers and ClusterIssuers. Do this by `apply`ing the manifest directly from the cert-manager GitHub repository :

```
kubectl apply --validate=false -f
https://github.com/jetstack/cert-
manager/releases/download/v0.12.0/cert-
manager.yaml
```

You should see the following output:

Output

```
customresourcedefinition.apiextensions.k8s.io/cert
ificaterequests.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/cert
ificates.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/chal
lenges.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/clus
terissuers.cert-manager.io created
. . .
deployment.apps/cert-manager-webhook created
mutatingwebhookconfiguration.admissionregistration
.k8s.io/cert-manager-webhook created
validatingwebhookconfiguration.admissionregistrati
on.k8s.io/cert-manager-webhook created
```

To verify our installation, check the `cert-manager` Namespace for running pods:

```
kubectl get pods --namespace cert-manager
```

Output

```
NAME                                          READY
STATUS    RESTARTS   AGE
cert-manager-5c47f46f57-jknnx                 1/1
Running   0          27s
cert-manager-cainjector-6659d6844d-j8cbg      1/1
Running   0          27s
cert-manager-webhook-547567b88f-qks44         1/1
Running   0          27s
```

This indicates that the cert-manager installation succeeded.

Before we begin issuing certificates for our Ingress hosts, we need to create an Issuer, which specifies the certificate authority from which signed x509 certificates can be obtained. In this guide, we'll use the Let's Encrypt certificate authority, which provides free TLS certificates and offers both a staging server for testing your certificate configuration, and a production server for rolling out verifiable TLS certificates.

Let's create a test Issuer to make sure the certificate provisioning mechanism is functioning correctly. Open a file named `staging_issuer.yaml` in your favorite text editor:

```
nano staging_issuer.yaml
```

Paste in the following ClusterIssuer manifest:

staging_issuer.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
 name: letsencrypt-staging
 namespace: cert-manager
spec:
 acme:
   # The ACME server URL
   server: https://acme-staging-
v02.api.letsencrypt.org/directory
   # Email address used for ACME registration
   email: your_email_address_here
   # Name of a secret used to store the ACME
account private key
   privateKeySecretRef:
     name: letsencrypt-staging
   # Enable the HTTP-01 challenge provider
   solvers:
   - http01:
       ingress:
         class:  nginx
```

Here we specify that we'd like to create a ClusterIssuer object called `letsencrypt-staging`, and use the Let's Encrypt staging server. We'll later use the production server to roll out our certificates, but the production server may rate-limit requests made against it, so for testing purposes it's best to use the staging URL.

We then specify an email address to register the certificate, and create a Kubernetes Secret called `letsencrypt-staging` to store the ACME account's private key. We also enable the `HTTP-01` challenge mechanism.

To learn more about these parameters, consult the official cert-manager documentation on Issuers.

Roll out the ClusterIssuer using `kubectl`:

```
kubectl create -f staging_issuer.yaml
```

You should see the following output:

Output

```
clusterissuer.cert-manager.io/letsencrypt-staging
created
```

Now that we've created our Let's Encrypt staging Issuer, we're ready to modify the Ingress Resource we created above and enable TLS encryption for the `echo1.example.com` and `echo2.example.com` paths.

Open up `echo_ingress.yaml` once again in your favorite editor:

```
nano echo_ingress.yaml
```

Add the following to the Ingress Resource manifest:

echo_ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: echo-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    cert-manager.io/cluster-issuer: "letsencrypt-
staging"
spec:
  tls:
```

```
  - hosts:
    - echo1.hjdo.net
    - echo2.hjdo.net
    secretName: echo-tls
  rules:
  - host: echo1.hjdo.net
    http:
      paths:
      - backend:
          serviceName: echo1
          servicePort: 80
  - host: echo2.hjdo.net
    http:
      paths:
      - backend:
          serviceName: echo2
          servicePort: 80
```

Here we add some annotations to specify the `ingress.class`, which determines the Ingress Controller that should be used to implement the Ingress Rules. In addition, we define the `cluster-issuer` to be `letsencrypt-staging`, the certificate Issuer we just created.

Finally, we add a `tls` block to specify the hosts for which we want to acquire certificates, and specify a `secretName`. This secret will contain the TLS private key and issued certificate.

When you’re done making changes, save and close the file.

We'll now update the existing Ingress Resource using `kubectl apply`:

```
kubectl apply -f echo_ingress.yaml
```

You should see the following output:

Output

```
ingress.networking.k8s.io/echo-ingress configured
```

You can use `kubectl describe` to track the state of the Ingress changes you've just applied:

```
kubectl describe ingress
```

Output

```
Events:
  Type    Reason             Age
From                     Message
  ----    ------             ----                 ---
-                      -------
  Normal  CREATE             14m
nginx-ingress-controller  Ingress default/echo-
ingress
  Normal  CreateCertificate  67s   cert-manager
Successfully created Certificate "echo-tls"
  Normal  UPDATE             53s   nginx-ingress-
controller  Ingress default/echo-ingress
```

Once the certificate has been successfully created, you can run an additional `describe` on it to further confirm its successful creation:

```
kubectl describe certificate
```

You should see the following output in the `Events` section:

Output

```
Events:
  Type    Reason          Age   From
Message
  ----    ------          ----  ----              -----
--
  Normal  GeneratedKey  2m12s  cert-manager
Generated a new private key
  Normal  Requested     2m12s  cert-manager
Created new CertificateRequest resource "echo-tls-
3768100355"
  Normal  Issued        47s    cert-manager
Certificate issued successfully
```

This confirms that the TLS certificate was successfully issued and HTTPS encryption is now active for the two domains configured.

We're now ready to send a request to a backend `echo` server to test that HTTPS is functioning correctly.

Run the following `wget` command to send a request to `echo1.example.com` and print the response headers to `STDOUT`:

```
wget --save-headers -O- echo1.example.com
```

You should see the following output:

Output

```
. . .
HTTP request sent, awaiting response... 308
```

```
Permanent Redirect
. . .
ERROR: cannot verify echo1.example.com's
certificate, issued by ‘CN=Fake LE Intermediate
X1’:
  Unable to locally verify the issuer's authority.
To connect to echo1.example.com insecurely, use `-
-no-check-certificate'.
```

This indicates that HTTPS has successfully been enabled, but the certificate cannot be verified as it’s a fake temporary certificate issued by the Let’s Encrypt staging server.

Now that we’ve tested that everything works using this temporary fake certificate, we can roll out production certificates for the two hosts `echo1.example.com` and `echo2.example.com`.

Step 5 — Rolling Out Production Issuer

In this step we’ll modify the procedure used to provision staging certificates, and generate a valid, verifiable production certificate for our Ingress hosts.

To begin, we’ll first create a production certificate ClusterIssuer.

Open a file called `prod_issuer.yaml` in your favorite editor:

```
nano prod_issuer.yaml
```

Paste in the following manifest:

prod_issuer.yaml

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
```

```
metadata:
  name: letsencrypt-prod
  namespace: cert-manager
spec:
  acme:
    # The ACME server URL
    server: https://acme-
v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: your_email_address_here
    # Name of a secret used to store the ACME
account private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

Note the different ACME server URL, and the `letsencrypt-prod` secret key name.

When you’re done editing, save and close the file.

Now, roll out this Issuer using `kubectl`:

```
kubectl create -f prod_issuer.yaml
```

You should see the following output:

Output

```
clusterissuer.cert-manager.io/letsencrypt-prod
created
```

Update `echo_ingress.yaml` to use this new Issuer:

```
nano echo_ingress.yaml
```

Make the following change to the file:

echo_ingress.yaml

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: echo-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    cert-manager.io/cluster-issuer: "letsencrypt-
prod"
spec:
  tls:
  - hosts:
    - echo1.hjdo.net
    - echo2.hjdo.net
    secretName: echo-tls
  rules:
  - host: echo1.hjdo.net
    http:
      paths:
      - backend:
          serviceName: echo1
          servicePort: 80
  - host: echo2.hjdo.net
    http:
      paths:
      - backend:
          serviceName: echo2
          servicePort: 80
```

Here, we update the ClusterIssuer name to `letsencrypt-prod`.

Once you're satisfied with your changes, save and close the file.

Roll out the changes using `kubectl apply`:

```
kubectl apply -f echo_ingress.yaml
```

Output

```
ingress.networking.k8s.io/echo-ingress configured
```

Wait a couple of minutes for the Let's Encrypt production server to issue the certificate. You can track its progress using `kubectl describe` on the `certificate` object:

```
kubectl describe certificate echo-tls
```

Once you see the following output, the certificate has been issued successfully:

Output

```
Events:
  Type    Reason         Age                    From
Message
  ----    ------         ----                   ----
-------
```

```
  Normal  GeneratedKey  8m10s                cert-
manager  Generated a new private key
  Normal  Requested     8m10s                cert-
manager  Created new CertificateRequest resource
"echo-tls-3768100355"
  Normal  Requested     35s                  cert-
manager  Created new CertificateRequest resource
"echo-tls-4217844635"
  Normal  Issued        10s (x2 over 6m45s)  cert-
manager  Certificate issued successfully
```

We'll now perform a test using `curl` to verify that HTTPS is working correctly:

```
curl echo1.example.com
```

You should see the following:

Output

```
<html>
<head><title>308 Permanent Redirect</title></head>
<body>
<center><h1>308 Permanent Redirect</h1></center>
<hr><center>nginx/1.15.9</center>
</body>
</html>
```

This indicates that HTTP requests are being redirected to use HTTPS.

Run `curl` on `https://echo1.example.com`:

```
curl https://echo1.example.com
```

You should now see the following output:

Output

```
echo1
```

You can run the previous command with the verbose `-v` flag to dig deeper into the certificate handshake and to verify the certificate information.

At this point, you've successfully configured HTTPS using a Let's Encrypt certificate for your Nginx Ingress.

Conclusion

In this guide, you set up an Nginx Ingress to load balance and route external requests to backend Services inside of your Kubernetes cluster. You also secured the Ingress by installing the cert-manager certificate provisioner and setting up a Let's Encrypt certificate for two host paths.

There are many alternatives to the Nginx Ingress Controller. To learn more, consult Ingress controllers from the official Kubernetes documentation.

For a guide on rolling out the Nginx Ingress Controller using the Helm Kubernetes package manager, consult How To Set Up an Nginx Ingress on DigitalOcean Kubernetes Using Helm.

How to Protect Private Kubernetes Services Behind a GitHub Login with oauth2_proxy

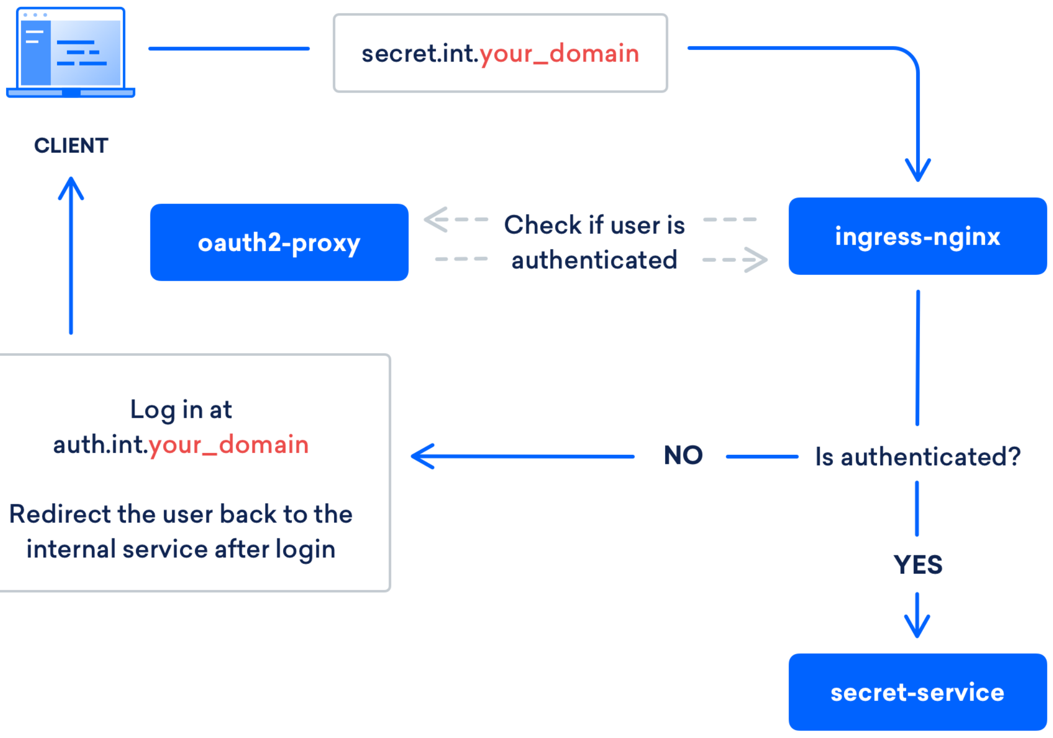
Written by Kamal Nasser

Kubernetes ingresses make it easy to expose web services to the internet. When it comes to private services, however, you will likely want to limit who can access them. In this tutorial you'll use oauth2_proxy with GitHub to protect your services. oauth2_proxy is a reverse proxy server that provides authentication using different providers, such as GitHub, and validates users based on their email address or other properties.

By the end of this tutorial you will have setup oauth2_proxy on your Kubernetes cluster and protected a private service behind a GitHub login. oauth2_proxy also supports other OAuth providers like Google and Facebook, so by following this tutorial you will be able to protect your services using the provider of your choice.

Kubernetes ingresses make it easy to expose web services to the internet. When it comes to private services, however, you will likely want to limit who can access them. oauth2_proxy can serve as a barrier between the public internet and private services. oauth2_proxy is a reverse proxy and server that provides authentication using different providers, such as GitHub, and validates users by their email address or other properties.

In this tutorial you'll use oauth2_proxy with GitHub to protect your services. When you're done, you will have an authorization system that looks like the one in the following diagram:


secret.int.your_domain
CLIENT
oauth2-proxy
Check if user is authenticated
ingress-nginx
Log in at auth.int.your_domain
Redirect the user back to the internal service after login
NO
Is authenticated?
YES
secret-service

A diagram of a request flow end-result

Prerequisites

To complete this tutorial, you'll need:

- A Kubernetes cluster with two web services running with an Nginx ingress and Let's Encrypt. This tutorial builds on How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes. Be sure to follow it to the very end in order to complete this tutorial.
- A GitHub account.
- Python installed on your local machine. If you do not have it installed, follow the installation instructions for your operating

system.

Step 1 — Configuring Your Domains

After following the tutorial linked in the Prerequisites section, you will have two web services running on your cluster: `echo1` and `echo2`. You will also have one ingress that maps `echo1.`**`your_domain`** and `echo2.`**`your_domain`** to their corresponding services.

In this tutorial, we will use the following conventions:

- All private services will fall under the `.int.`**`your_domain`** subdomain, like `service.int.`**`your_domain`**. Grouping private services under one subdomain is ideal because the authentication cookie will be shared across all `*.int.`**`your_domain`** subdomains.
- The login portal will be served on `auth.int.`**`your_domain`**.

Note: Be sure to replace **`your_domain`** with your own domain name wherever it appears in this tutorial.

To start, update the existing ingress definition to move the `echo1` and `echo2` services under `.int.`**`your_domain`**. Open `echo_ingress.yaml` in your text editor so you can change the domains:

```
nano echo_ingress.yaml
```

Rename all instances of `echo1.`**`your_domain`** to `echo1.int.`**`your_domain`**, and replace all instances of `echo2.`**`your_domain`** with `echo2.`**`int.your_domain`**:

echo_ingress.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: echo-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
spec:
  tls:
  - hosts:
    - echo1.int.your_domain
    - echo2.int.your_domain
    secretName: letsencrypt-prod
  rules:
  - host: echo1.int.your_domain
    http:
      paths:
      - backend:
          serviceName: echo1
          servicePort: 80
  - host: echo2.int.your_domain
    http:
      paths:
      - backend:
          serviceName: echo2
          servicePort: 80
```

Save the file and apply the changes:

```
kubectl apply -f echo_ingress.yaml
```

This will update the TLS certificates for your `echo1` and `echo2` services as well.

Now update your DNS configuration to reflect the changes you made. First, look up the IP address of your Nginx ingress by running the following command to print its details:

```
kubectl get svc --namespace=ingress-nginx
```

You will see the IP address under `EXTERNAL-IP` in the output:

Output

```
NAME               TYPE              CLUSTER-IP
EXTERNAL-IP          PORT(S)                          AGE
ingress-nginx     LoadBalancer     10.245.247.67
203.0.113.0     80:32486/TCP,443:32096/TCP    20h
```

Copy the external IP address to your clipboard. Browse to your DNS management service and locate the A records for `echo1-2.`**`your_domain`** to point to that external IP address. If you are using DigitalOcean to manage your DNS records, see How to Manage DNS Records for instructions.

Delete the records for `echo1` and `echo2`. Add a new `A` record for the hostname `*.int.`**`your_domain`** and point it to the External IP address of the ingress.

Now any request to any subdomain under `*.int.`**`your_domain`** will be routed to the Nginx ingress, so you can use these subdomains within your cluster.

Next you'll configure GitHub as your login provider.

Step 2 — Creating a GitHub OAuth Application

oauth2_proxy supports various login providers. In this tutorial, you will use the GitHub provider. To get started, create a new GitHub OAuth App.

In the OAuth Apps tab of the Developer settings page of your account, click the New OAuth App button.

The Application name and Homepage URL fields can be anything you want. In the Authorization callback URL field, enter `https://auth.int.your_domain/oauth2/callback`.

After registering the application, you will receive a Client ID and Secret. Note the two as you will need them in the next step.

Now that you've created a GitHub OAuth application, you can install and configure oauth2_proxy.

Step 3 – Setting Up the Login Portal

You'll use Helm to install oauth2_proxy onto the cluster. First, you'll create a Kubernetes secret to hold the GitHub application's Client ID and Secret, as well as an encryption secret for browser cookies set by oauth2_proxy.

Run the following command to generate a secure cookie secret:

```
python -c 'import os,base64; print
base64.b64encode(os.urandom(16))'
```

Copy the result to your clipboard

Then, create the Kubernetes secret, substituting the highlighted values for your cookie secret, your GitHub client ID, and your GitHub secret key:

```
kubectl -n default create secret generic oauth2-
proxy-creds \
--from-literal=cookie-secret=YOUR_COOKIE_SECRET \
--from-literal=client-id=YOUR_GITHUB_CLIENT_ID \
--from-literal=client-secret=YOUR_GITHUB_SECRET
```

You'll see the following output:

Output

```
secret/oauth2-proxy-creds created
```

Next, create a new file named `oauth2-proxy-config.yaml` which will contain the configuration for `oauth2_proxy`:

```
nano oauth2-proxy-config.yaml
```

The values you'll set in this file will override the Helm chart's defaults. Add the following code to the file:

oauth2-proxy-config.yaml

```
config:
  existingSecret: oauth2-proxy-creds

extraArgs:
  whitelist-domain: .int.your_domain
  cookie-domain: .int.your_domain
  provider: github

authenticatedEmailsFile:
  enabled: true
  restricted_access: |-
    allowed@user1.com
    allowed@user2.com

ingress:
  enabled: true
  path: /
  hosts:
    - auth.int.your_domain
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
  tls:
    - secretName: oauth2-proxy-https-cert
      hosts:
        - auth.int.your_domain
```

This code does the following:

1. Instructs oauth2_proxy to use the secret you created.
2. Sets the domain name and provider type.
3. Sets a list of allowed email addresses. If a GitHub account is associated with one of these email addresses, it will be allowed access to the private services.
4. Configures the ingress that will serve the login portal on `auth.int.`**`your_domain`** with a TLS certificate from Let’s

Encrypt.

Now that you have the secret and configuration file ready, you can install `oauth2_proxy`. Run the following command:

```
helm repo update \
&& helm upgrade oauth2-proxy --install
stable/oauth2-proxy \
--reuse-values \
--values oauth2-proxy-config.yaml
```

It might take a few minutes for the Let's Encrypt certificate to be issued and installed.

To test that the deployment was successful, browse to `https://auth.int.`**`your_domain`**. You'll see a page that prompts you to log in with GitHub.

With oauth2_proxy set up and running, all that is left is to require authentication on your services.

Step 4 — Protecting the Private Services

In order to protect a service, configure its Nginx ingress to enforce authentication via oauth2_proxy. Nginx and nginx-ingress support this configuration natively, so you only need to add a couple of annotations to the ingress definition.

Let's protect the `echo1` and `echo2` services that you set up in the prerequisite tutorial. Open `echo_ingress.yaml` in your editor:

```
nano echo_ingress.yaml
```

Add these two additional annotations to the file to require authentication:

echo_ingress.yaml

```
   annotations:
     kubernetes.io/ingress.class: nginx
     certmanager.k8s.io/cluster-issuer:
letsencrypt-prod
     nginx.ingress.kubernetes.io/auth-url:
"https://auth.int.your_domain/oauth2/auth"
     nginx.ingress.kubernetes.io/auth-signin:
"https://auth.int.your_domain/oauth2/start?
rd=https%3A%2F%2F$host$request_uri"
```

Save the file and apply the changes:

```
kubectl apply -f echo_ingress.yaml
```

Now when you browse to `https://echo1.int.`**`your_domain`**, you will be asked to log in using GitHub in order to access it. After logging in with a valid account, you will be redirected back to the `echo1` service. The same is true for `echo2`.

Conclusion

In this tutorial, you set up oauth2_proxy on your Kubernetes cluster and protected a private service behind a GitHub login. For any other services you need to protect, simply follow the instructions outlined in Step 4.

oauth2_proxy supports many different providers other than GitHub. To learn more about different providers, see the official documentation.

Additionally, there are many configuration parameters that you might need to adjust, although the defaults will suit most needs. For a list of parameters, see the Helm chart's documentation and oauth2_proxy's documentation.