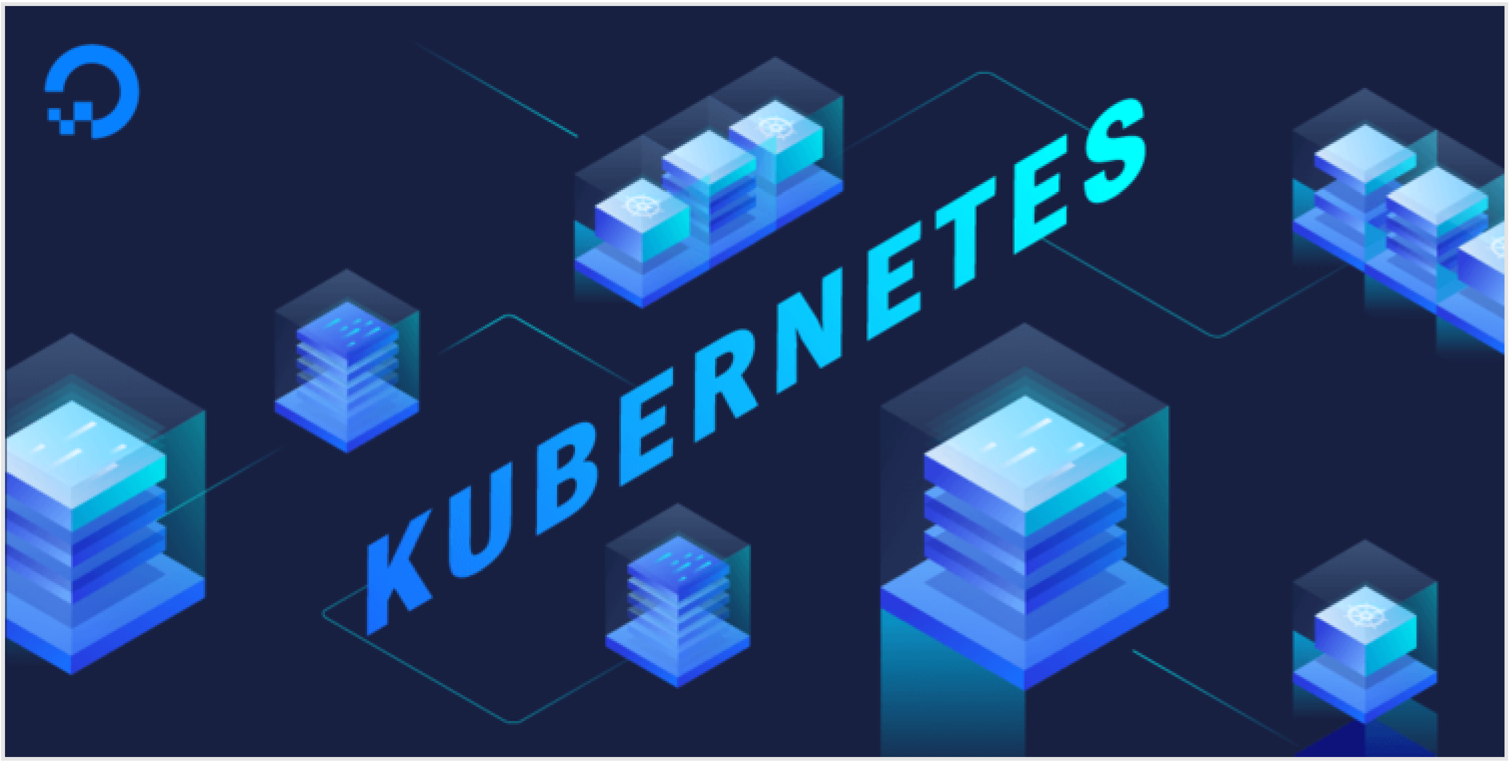

Modernizing Applications for Kubernetes

Posted September 12, 2018 21.8k KUBERNETES CONCEPTUAL

6

By: Hanif Jetha

Introduction

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

```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.

By: Hanif Jetha

Upvote (6) Subscribe Share

Related Tutorials

Architecting Applications for Kubernetes

An Introduction to Service Meshes

How To Deploy a PHP Application with Kubernetes on Ubuntu 16.04

How To Set Up Multi-Node Deployments With Rancher 2.1, Kubernetes, and Docker Machine on Ubu...

How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes

0 Comments

Leave a comment...

Log In to Comment