Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2021, 8(7):118-123

Research Article ISSN: 2394 - 658X

Automatic Update of Kubernetes Pods When ConfigMap is Updated

Pallavi Priya Patharlagadda

Pallavipriya527.p@gmail.com
United States of America

ABSTRACT

Kubernetes has become very popular in the last few years and is gaining a lot of momentum. While adoption is increasing, more users are becoming aware of the gaps in Kubernetes. The lifecycle management of application configuration is one such topic. The ConfigMap is a Kubernetes construct used to store the configuration. ConfigMaps can be accessed using Pods or Deployments, and environment variables or files via volumes can be used to inject values into the container. Nevertheless, these ConfigMaps' lifecycles are entirely independent of the containers that utilize them. This means that when configuration is updated, the container won't pick up the most recent version and the Pod or Deployment that uses it won't be activated. The only way to get configuration updates would be to recreate the containers.

When mounting configuration to volumes, the files on the disk are "eventually updated as well" when a ConfigMap is modified. However, this results in unreliable deployments. The application might be broken by the configuration that is altered "at some point," but rollbacks are more difficult to execute because this is an out-of-bound activity from the deployment. It also contradicts the idea of immutable containers. In this paper, we provide a solution on how to overcome this problem without manual intervention.

Keywords: Kubernetes, ConfigMap

PROBLEM STATEMENT

Runtime or application initialization are two times when many applications rely on configuration. Value adjustments for setup settings are typically necessary. You can inject configuration data into application pods using the Kubernetes ConfigMaps. With the ConfigMap idea, you can maintain the portability of containerized apps by separating configuration artifacts from image content. For instance, you can spin up containers for local development, system testing, or managing an active end-user workload by downloading and executing the same container image. But if the configmap is updated with the latest configuration, then the pod should also be updated to reflect the latest configuration. These ConfigMaps lifecycles are entirely independent of the containers that utilize them. But in the usual scenario, pods/deployments don't get updated automatically. The only way to get configuration updates would be to recreate the containers. In this paper, we will provide you with a solution on how to make sure the deployment gets updated when the config map is updated.

INTRODUCTION

Every application requires configuration. It is common to refer to "special" data, like tokens, API keys, and other confidential information. Configuration options, such as a PHP.ini file, environment variables, and flags that alter your app's logic can all be used to modify your application. It can be tempting to incorporate these references into your application logic using hard coding. This might work in a small standalone application, but in any app that is substantially sized, it quickly becomes unmanageable. The workaround for this is to keep configuration files and environment variables in a "central location" that our app can access. Simply edit the file or modify the environment variable to make any configuration changes, and you're ready to go. In Kubernetes, this can be achieved using Config Maps.

Let's do a deep dive into Kubernetes ConfigMap and Kubernetes Helm:

KUBERNETES ConfigMap

Data can be stored as key-value pairs using an API object called a Kubernetes ConfigMap. ConfigMaps are available for usage by Kubernetes pods as environment variables, configuration files, or command-line arguments.
Advantages:
Applications can be made portable by separating environment-specific configurations from containers using ConfigMaps.
Disadvantages:
1. ConfigMaps don't use any kind of encryption. Hence, anybody with file access permissions can see all of the data inside. To store sensitive data, you can utilize Kubernetes secrets.
2. The requirement that files be no more than 1MB is another possible disadvantage of ConfigMaps. Bigger datasets might need other storage techniques, including distinct file mounts, file services, or databases.
Creating a ConfigMap:
Use the kubectl command to build a new ConfigMap

```
kubectl create configmap <name> <data-source>
```

where <data-source> denotes the directory, file, or literal value from which the data is to be drawn, and <map-name> is the name you choose to give to the ConfigMap. A ConfigMap object's name needs to be a legitimate DNS subdomain name.
The basename of the file is the default value for the key in the <data-source> when building a ConfigMap based on a file, and the file content is the default value.
You can use kubectl describe or kubectl get to retrieve information about a ConfigMap
Here is the sample config.yml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-cm
data:
  sample-key: sample
  example-key: example
```

ConfigMap contents can be displayed using below command.

```
kubectl describe configmaps nginx-cm
```

```
Name:         example-cm
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
sample-key: sample
example-key: example
```

There are two primary ways to access the information stored in a configmap. The first step is to mount the configmap as a volume in the application container, which can then be read using standard file I/O methods. The second step is to get the configmap from the Kubernetes API. In this article, I'll go over how to mount the configuration map as a volume.
Create a Deployment with Volume Mounts using ConfigMap:
Add the ConfigMap name to the volume section of the Pod specification. This loads the ConfigMap data into the directory indicated by volumeMounts.mountPath (in this case, /etc/config/cm). The command section displays directory files whose names match the keys in ConfigMap.

```
apiVersion: v1
kind: Deployment
metadata:
  name: dapi-test-pod
spec:
  template:
    spec:
      containers:
        - name: test-container
        image: registry.k8s.io/busybox
        command: [ "/bin/sh", "-c", "ls /etc/config/cm" ]
        volumeMounts:
        - name: config-volume
              mountPath: /etc/config/cm
    volumes:
       - name: config-volume
        configMap:
              # Provide the name of the ConfigMap containing the files you want
              # to add to the container
              name: example-cm
```

kubectl create -f deployment-configmap-volume.yaml
When the deployment runs, it would display the output as below
sample-key: sample
example-key: example
In the preceding example, we set our mount point as /etc/config/cm, which is the name by which your program would refer to this configmap when accessing it through the filesystem. One important consideration is that mount points are usually directories. So, the application will recognize the configmap as a directory containing one or more files. As a result, each key in your configmap is exposed as its file under the configmap's mount point. Even if the value is a single, relatively short string, as in our example-cm config map, each key will appear as a separate file, with the contents of the file representing the key's value in the original configmap.

HELM

Helm is a Kubernetes package management. It enables the grouping of Kubernetes objects into a bundled application that is easy to download, install, and customize for individual use. These packages are known as charts in Helm (much like debs or rpms). Helm creates a Kubernetes cluster in the background.
Helm is structured on several important ideas:
● A chart consists of a set of preconfigured Kubernetes resources.
● Releases are particular instances of charts that have been Helm-deployed to the cluster.
● A collection of publicly available charts is called a repository.
Kubernetes With the release of Kubernetes 1.4 in 2016, Google and Deis unveiled Helm. Due to Helm's recent inception, there aren't many public repositories for Helm packages; hub.kubeapps.com is one such repository.
What Problems Does Kubernetes Helm Fix?
One well-known complicated platform with a challenging learning curve is Kubernetes. Kubernetes Helm facilitates quicker and simpler Kubernetes use:
● **Enhances productivity:** Developers may spend more time building their apps and less time deploying test environments to test their Kubernetes clusters by using a Helm chart to install a pre-tested app.
● **Existing Helm Charts:** Enable developers to quickly and easily install a functional CMS, big data platform, database, etc. for their application with just one click. To automate production, testing, or development processes, developers might make custom charts or change pre-existing ones.
● **Less complicated to start using Kubernetes**: Learning how to implement production-grade apps with Kubernetes might be challenging at first. Even if you lack significant container knowledge, Helm's one-click app deployment makes it a lot simpler to get going and launch your first app.
● **Decreased complexity:** The implementation of applications coordinated by Kubernetes may be quite intricate. Deployments might fail due to misconfigured configuration files or improper app rollout from YAML templates. By establishing fixed values and appropriate defaults for adjustable parameters, Helm Charts enables the community to preconfigure applications and offer a uniform interface for modifying configuration. This removes deployment mistakes by locking out wrong settings and drastically decreases complexity.
● **Production ready**: It is challenging and error-prone to operate Kubernetes in production with all of its components (pods, namespaces, deployments, etc.). Users can lower the difficulty of managing a Kubernetes App Catalog and deploy to production with confidence when they have a tested and reliable Helm chart.
● **No wasted effort:** a chart may be utilized by several organizations both inside and outside of an organization once a developer has generated, tested, and stabilized it. It used to be significantly harder, but still doable, to duplicate and exchange Kubernetes apps between environments.

HELM ARCHITECTURE

Helm has two main components:

● **Helm Client:** Developers may communicate with the tiller server, maintain chart repositories, and create new charts using the Helm Client.
● **Tiller:** Within the Kubernetes cluster is the Tiller Server. communicates with the Helm client and converts Kubernetes API instructions to chart definitions and configurations. Tiller builds a release by combining a chart and its settings. Upgrading charts or removing and discarding them from the Kubernetes cluster are under Tiller's purview as well

After Helm is installed, the helm init command adds the Tiller server to your Kubernetes cluster. It is then feasible to install charts on the cluster.

Helm templates and Values:

Go is used to create the Helm chart templates. The templates/ folder of a chart contains all template files. Every file in that directory is rendered using the template engine whenever Helm accesses a chart. To supply default values for a template in a particular chart, you can include a values.yaml file inside the chart. Users of charts can provide a values-containing YAML file. The command "helm install" can supply this.

Through the usage of a chart template, the designer of the chart can provide variables that users can change when the chart is installed. These variables are referred to as values, and to guarantee that the chart installs correctly right out of the box, all values must have suitable defaults defined for them. Below is the sample template file that reads three files to our config map.

config1.toml:

```
message = "Hello from config 1"
```

config2.toml:

```
message = " Hello from config 2"
```

config3.toml:

```
message = " Hello from config 3"
```

These are all just plain TOML files, like the outdated Windows INI files. Since we are aware of the names of these files, we can loop through them and add their contents to our ConfigMap by using a range function.

When you run this template, all three files' contents will be combined into a single ConfigMap.

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-configmap
data:
  config1.toml: |-
    message = "Hello from config 1"

  config2.toml: |-
    message = " Hello from config 2"

  config3.toml: |-
    message = " Hello from config 3"
```

UPDATING A DEPLOYMENT THAT USES CONFIGMAP:

Let us consider a scenario where a config map is updated. Let's check if the deployment also gets updated per the config map. We will take an example of a nginx. Below is the basic chart. I have the following ConfigMap manifest within that chart:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-cm
data:
  default.conf: {{.Files.Get "files/default.conf" | printf "%s" | indent 4}}
```

As you can see, a file named default.conf is reading the data from the ConfigMap, which goes by the name nginx-cm. This file, default.conf, is used to configure nginx.

```
server {
  listen       80;
  server_name  localhost;

  location / {
    proxy_pass http://<project-url>.com/;
  }
  error_page   500 502 503 504  /50x.html;
  location = /50x.html {
    root   /usr/share/nginx/html;
  }
}
```

I now want to use this nginx-cm ConfigMap for my application. I will thus use Volume Mounts to expose it in my app's deployment manifest

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/conf.d/
      volumes:
        - name: config-volume
          configMap:
            name: nginx-cm
```

We must add the ConfigMap to the Volumes section and give it a special name (config-volume, as demonstrated in the example manifest), as indicated in the manifest above. Next, we add this volume in the containers section under volume mounts. The precise place within the container where the configuration file will be made available to the container is indicated by the volumeMounts.mountPath parameter.
We can thus use these manifests to launch an application that utilizes the contents of the nginx configuration file that the ConfigMap makes available.
Assume for the moment that the nginx configuration file needs to be updated. If this configuration file is changed, the ConfigMap should also be updated; otherwise, the new content won't be used by our app that makes use of the ConfigMap.
The kubectl update command is undoubtedly capable of updating the ConfigMap. An update to the deployment ought to come after this. Will the deployment also be supported by kubectl update?

When I attempted, I received the below message.

```
Deployment  "kube-nginx-deployment" remains unchanged
```

This is because even after updating the configMap resource, the deployment's spec.template section remains unchanged. The deployment's spec.template remained unchanged despite the changes made to the ConfigMap's data section. One way around this is to use the deployment to create new pods using the updated configMap after deleting every pod it is currently managing. This method wasn't accepted well with me, though, because you have to manually remove each pod. I then set out to find better answers.

A sha256 hash of a string can be generated using the Helm templating language. We can write a one-liner that generates this hash for us directly in the template thanks to this and the ability to load templates as strings.

Below is the updated deployment file with added annotation.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx/conf.d/
      volumes:
        - name: config-volume
          configMap:
            name: nginx-cm
```

You can input the path to your configmap file to the sha256 sum method, as you can see under annotations. Every time the configmap file is modified, this modifies the annotation section, which updates the deployment's spec.template. This was useful because the contents of the configuration files might change quite a bit. Helm makes sure that the program will continue to update quickly to reflect those changes because of this approach.
The fact that no further scripting is needed is a benefit over the prior solution. Now, the logic that is carried out is visibly and unambiguously visible while reading the YAML template.

CONCLUSION

If you are already using Helm, I strongly advise you to use the sha256sum approach as advised to make sure that any modifications to the configuration map file update the deployment's annotation section. If you are not utilizing Helm, creating and writing the hash yourself will allow you to simply implement the functionality. It would make it obvious that a change to ConfigMap could potentially cause a restart of any Pods that use it, given that ConfigMap is effectively a dependency for a pod. I believe Kubernetes would provide support for it in the future.

REFERENCES

[1]. https://www.rancher.cn/blog/2018/2018-07-10-helm-tips-and-tricks-updating-app-with-configmap/
[2]. https://github.com/mrajashree/helm-cm1/blob/master/templates/deployment.yaml
[3]. https://v2.helm.sh/docs/charts_tips_and_tricks/#automatically-roll-deployments-when-configmaps-or-secrets-change
[4]. https://sanderknape.com/2019/03/kubernetes-helm-configmaps-changes-deployments/
[5]. https://blog.palark.com/configmaps-in-kubernetes-how-they-work-and-what-you-should-remember/
[6]. https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-configmap/
[7]. https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-helm/
[8]. https://v2.helm.sh/docs/developing_charts/#chart-dependencies
[9]. https://v2.helm.sh/docs/chart_template_guide/
[10]. https://v2.helm.sh/docs/developing_charts/#templates-and-values