# Kubernetes controller configuration patterns

## Table of Contents

* [Introduction](#introduction)
* [Basics](#basics)
  + [1. Antipattern: Configuration with command line arguments](#X509acca89c455d6278361c579a14230c5aa989b)
  + [2. Configuration with environment variables](#X05cd225a824dd7a705287414481fb955706292f)
  + [3. Antipattern: configuration with NFS volume mounts](#Xa3420e7677d47770298f8b3988bd14962cae39b)
  + [4. Antipattern: configuration with init containers](#X13ac4d65d5eebdcb7a6fdaa4052fbab3a4a05fc)
  + [5. Configuration with configmap mounted as file](#Xf3063bd64ac1f6b5701a688e80c31241be98c50)
  + [6. Configuration with configmaps mounted as multiple files](#Xbd55305ab56657f8d658d9ba6e27f9a460cc890)
  + [7. Configuration with configmaps used as a source for environment variables](#X03771a6a07cc61268adddc3202ddba057d7f505)
  + [8. Configuration with secrets](#X3aeb702c13d9a61071475c8f635ddabdcf7ab06)
* [Advanced patterns](#advanced-patterns)
  + [1. Configuration with global configmaps](#X58a1e1112f35268e3c71b0ce59b3e1577bfc7be)
  + [2. Configuration with global and namespaced configmaps](#X527707b68d70d7b2168cf392126093c226342b6)
  + [3. Configuration with custom resources](#X53cc7af7c5b303591c47c04a0b0df4c64fa37bf)
  + [4. Configuration with custom resources, falling back to configmaps](#Xe614999ffaafce07b4e2c0020572939f80640e5)
  + [5. References to configmaps in custom resources](#X0a1ab6483914cadc48d24ed4198f3e7f7cd45d7)
  + [6. Using configmaps with labels](#X4b0eb5a780d7075833aae4771d4d2025ef3b27c)

## Introduction

Notes: - Only deployments are shown, but same with other PodSpecables like Pods, DaemonSets etc. - For brevity, fields like image etc. are omitted

## Basics

This section describes patterns that are basics to configuring an application running on Kubernetes.

### 1. Antipattern: Configuration with command line arguments

Kubernetes allows passing command line arguments to containers and this can be facilitated to provide configuration to the applications.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 args:  
 - "gravity=10"  
 - "colorMode=dark"

The command line arguments gravity and colorMode will be passed to the application running in the container. The application program needs to read the command line arguments and use them.

This pattern is not very flexible and things are mostly hardcoded in the YAML.

Except migrating the legacy software into containers that use command line arguments for configuration, any of the alternatives in this document is a better option in terms of flexibility.

### 2. Configuration with environment variables

Kubernetes allows setting environment variables on containers. Applications in containers may use this environment variables as configuration options.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 env:  
 - name: GRAVITY  
 value: "10"  
 - name: COLOR\_MODE  
 value: "dark"

The environment variables GRAVITY and COLOR\_MODE will be passed to the application running in the container. The application program needs to read the environment variables and use them.

This pattern is more flexible although in the example above things are hardcoded in the YAML. As seen in the later sections of this document, ability to pass environment variables is supported to get values from different sources such as Secrets, ConfigMaps and references.

Also a nice feature is that the deployment will be rolled out when there is a change in the environment variables.

If there is no need to share the config among different deployments or containers, this pattern provides the simplest and most straight forward configuration method.

### 3. Antipattern: configuration with NFS volume mounts

A volume with a configuration file in it can be mounted to a container. Later, the application running in the container can read the configuration file from the file system.

To create such volume, a network file system (NFS) volume can be used.

Container  
 +--------------------------------------+  
 |--------------------------------------|  
 || ||  
 || Container image file system ||  
 || / ||  
 || └── bin ||  
 || ├── foo ||  
 || └── bar ||  
 Volume with configuration || ||  
+-----------------------------------+ |--------------------------------------|  
| | |--------------------------------------|  
| | || ||  
| / | || Volume mount ||  
| └── configs | || / ||  
| └── game-server +--------------------------> || └── etc ||  
| ├── physics.properties | || └── config ||  
| └── ui.properties | || ├── physics.properties ||  
| | || └── ui.properties ||  
| | |--------------------------------------|  
+-----------------------------------+ +--------------------------------------+

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 volumes:  
 - name: nfs-volume  
 nfs:  
 server: nfs.example.com  
 path: /configs/game-server/  
 containers:  
 - name: server  
 volumeMounts:  
 - name: nfs-volume  
 mountPath: /etc/config

In this example, the game-server container will be able to read the files under /etc/config/ directory and that directory will be a reflection of the /configs/game-server/ path in the NFS server defined.

The configuration file or files must be separately created manually in the NFS, before creating the deployment on Kubernetes.

This pattern is very similar to [Configuration with configmap mounted as file](#Xf3063bd64ac1f6b5701a688e80c31241be98c50) pattern, except the need of NFS and need of the creation of the configuration file manually in the NFS.

This is an antipattern because of the necessities above however, in some cases this pattern is useful. One of the cases is when the configuration is very large, such as machine learning models, if it is OK to consider them as *configuration*. Another useful case is when the configuration has to be immutable.

### 4. Antipattern: configuration with init containers

In this pattern, the configuration lives in a separate image which is run by an *init container* and that container copies the configuration it holds into a volume shared with other containers.

Pod  
 +-----------------------------------------------------------------------+  
 | |  
 | |  
 | Init container |  
 | +-----------------------------------------+ |  
 | | | |  
 | copies the config | Container image file system | |  
 Shared Volume | into shared volume | / | |  
+---------------------------------------+ | +------------+ └── config-src | |  
| | | | | └── configuration.properties | |  
| | | | | | |  
| | | | | | |  
| / | | | +-----------------------------------------+ |  
| └── etc | | | |  
| └── config <-----------------------------------------+ |  
| └── configuration.properties | | |  
| | | Container |  
| | | +-----------------------------------------+ |  
| +---------------------+ |-----------------------------------------| |  
| | | | || || |  
+---------------------------------------+ | | || Container image file system || |  
 | | || / || |  
 | | || └── bin || |  
 | | || ├── foo || |  
 | | || └── bar || |  
 | | || || |  
 | | |-----------------------------------------| |  
 | | |-----------------------------------------| |  
 | | || || |  
 | | || Volume mount || |  
 | | mounted || || |  
 | | volume || / || |  
 | +-------------> || └── etc || |  
 | || └── config || |  
 | || └───configuration.properties || |  
 | || || |  
 | |-----------------------------------------| |  
 +-----------------------------------------------------------------------+

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 volumes:  
 - name: config-directory  
 emptyDir: {}  
 initContainers:  
 - name: init  
 image: initcontainer.example.com  
 volumeMounts:  
 - name: config-directory  
 mountPath: /etc/config  
 containers:  
 - name: server  
 volumeMounts:  
 - name: config-directory  
 mountPath: /etc/config

In this pattern, the *init container* would hold the configuration in it and later it would copy it over to the volume that is shared with the *server* container.

The init container has to be built that way. For example:

FROM busybox  
ADD configuration.properties /config-src/configuration.properties  
ENTRYPOINT [ "sh", "-c", "cp /config-src/\* /etc/config", "--" ]

The configuration.properties file is the configuration file that is added to the container image during its build. Later, the file is copied with cp command, in a simple way. So, the model should be available during the build time of the init container.

Obviously, any changes done to the configuration file in the shared volume by the *server* container will be transient. This pattern can be used, similarly to the NFS volume pattern, when the configuration file is too large or if an immutable configuration is targeted. This is considered as an antipattern except the cases above.

### 5. Configuration with configmap mounted as file

It is possible to mount a Kubernetes configmap into a container as a volume.

Container  
 +--------------------------------------+  
 |--------------------------------------|  
 || ||  
 || Container image file system ||  
 || / ||  
 || └── bin ||  
 Configmap || ├── foo ||  
+------------------------------------+ || └── bar ||  
| | |--------------------------------------|  
| game.properties: | |--------------------------------------|  
| gravity=10 +<---------+ || ||  
| colorMode=dark | | || Configmap mount ||  
| | | || ||  
+------------------------------------+ +---------| / ||  
 || └── etc ||  
 || └── config ||  
 || └── game.properties ||  
 || ||  
 |--------------------------------------|  
 +--------------------------------------+

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: game-config  
 namespace: my-game  
data:  
 game.properties: |  
 gravity=10  
 colorMode=dark  
  
---  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 volumeMounts:  
 - mountPath: /etc/config  
 name: game-config-volume  
 volumes:  
 - name: game-config-volume  
 configMap:  
 name: game-config

The data inside the configmap will be mounted to the deployment as a file in a volume. The key in data, game.properties, is used as the file name and the value of that key will be the file content. The configuration file path will be /etc/config/game.properties. Then the game application can read and process the .properties file.

It is easy to understand and manage this configuration pattern. Configuration can be read as a whole file in the application container, which might be simpler and faster. Also, same configmap can be mounted to different containers easily, compared to setting values from [configmaps as environment variables](TODO:%20link).

However, this pattern needs file reading operation within the application running in the container, which may not be ideal in every case. Also, when using this pattern, the configuration that the deployment is using is not visible directly on the deployment, nor the pods created for the deployment. An operator needs to go and check the configmap to see the configuration values.

When there is a lot of configuration options, configmap mount makes sense to use because the configuration is available as a whole with a single volumeMount.

Also, when a specific file format is used, such as the .properties file in example above, the file content can be fed to a properties file parser, which is available in many languages.

By default, the deployment will not be restarted when there is a change in the mounted configmap. However, there are techniques to accomplish that.

When there is not many configuration options, alternative patterns like [setting environment variables from configmaps](TODO:link) can be used to avoid the file reading operation and to avoid the volume mount.

Despite the fact that it is possible to make a mounted configmap readonly, it will not really immutable. Cluster admins will be able to change the configmap, thus the configuration for an application. If immutability is needed, use [configuration with NFS volume mounts](TODO:link) or [configuration with init containers](TODO:link)patterns.

### 6. Configuration with configmaps mounted as multiple files

This pattern is same as the [Configuration with configmap mounted as file](TODO) pattern, except it uses multiple files for separation of the configuration for different applications.

Container  
 +--------------------------------------+  
 |--------------------------------------|  
 || ||  
 || Container image file system ||  
 || / ||  
 || └── bin ||  
 Configmap || ├── foo ||  
+------------------------------------+ || └── bar ||  
| | |--------------------------------------|  
| game.properties: | |--------------------------------------|  
| gravity=10 +<---------+ || ||  
| | | || Configmap mounts ||  
| ui.properties: | | || ||  
| colorMode=dark | +---------| / ||  
| | || └── etc ||  
+------------------------------------+ || └── config ||  
 || ├── game.properties ||  
 || └── ui.properties ||  
 |--------------------------------------|  
 +--------------------------------------+

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: game-config  
 namespace: my-game  
data:  
 game.properties: |  
 gravity=10  
 ui.properties: |  
 colorMode=dark  
  
---  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 volumeMounts:  
 - mountPath: /etc/config  
 name: game-config-volume  
 volumes:  
 - name: game-config-volume  
 configMap:  
 name: game-config

As in [Configuration with configmap mounted as file](TODO) pattern, the data inside the configmap will be mounted to the deployment as files in a volume. This time though, there will be 2 files: /etc/config/game.properties and /etc/config/ui.properties

This separation is useful in some cases like when modules of an application reads their configuration independently, or, simply, when the configuration has too many options that it makes sense to split into multiple logical parts.

### 7. Configuration with configmaps used as a source for environment variables

This pattern is preferred when there is not many configuration options and when IO operations are undesired.

Container  
 +--------------------------------------+  
 |--------------------------------------|  
 || ||  
 || Container image file system ||  
 || / ||  
 || └── bin ||  
 Configmap || ├── foo ||  
+------------------------------------+ || └── bar ||  
| | || ||  
| data: | || ||  
| gravity: 10 +<---------+ || ||  
| colorMode: dark | | || Container environment ||  
| | || - GRAVITY: 10 ||  
+------------------------------------+ +---------| - COLOR\_MODE: dark ||  
 || ||  
 || ||  
 || ||  
 || ||  
 |--------------------------------------|  
 +--------------------------------------+

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: game-config  
 namespace: my-game  
data:  
 gravity: "10"  
 colorMode: "dark"

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 env:  
 - name: GRAVITY  
 valueFrom:  
 configMapKeyRef:  
 name: game-config  
 key: gravity  
 - name: COLOR\_MODE  
 valueFrom:  
 configMapKeyRef:  
 name: game-config  
 key: colorMode

The environment variables will be set with the values from configmap as GRAVITY=10 and COLOR\_MODE=dark. The application running in the container needs to read the environment variables and use them as the configuration.

This pattern provides a nice way for binding the options in the configmap to the configuration options that are actually used in the application. There might be more options in the configmap, but not every application needs all options.

With this pattern, application does not need doing file IO operations.

### 8. Configuration with secrets

Anything that can be done is also possible with secrets. It is a better practice to store credentials, certificates and similar things in a secret, rather than in a configmap. Secrets are more secure compared to configmaps. That subject is not in the scope of this document.

For example, to mount a secret as a file on a volume, following YAML can be used:

apiVersion: v1  
kind: Secret  
type: Opaque  
metadata:  
 name: game-secret  
 namespace: my-game  
data:  
 databaseUsername: Zm9v  
 databasePassword: YmFy  
---  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: game-server  
 name: game-server-deployment  
 namespace: my-game  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: game-server  
 template:  
 metadata:  
 labels:  
 app: game-server  
 spec:  
 containers:  
 - name: server  
 volumeMounts:  
 - mountPath: /etc/config  
 name: game-secret-volume  
 volumes:  
 - name: game-secret-volume  
 configMap:  
 name: game-secret

This will create 2 files that are available to the *server* container: /etc/config/databaseUsername and /etc/config/databasePassword.

Other patterns that use a configmap for configuration will not be repeated for secrets in this document. As seen in the example above, simply replacing the configmap with secret will work.

## Advanced patterns

This group of patterns are for more advanced scenarios where the basic Kubernetes features are not enough such as not being able to mount a configmap from another namespace into a pod.

These patterns will require coding against Kubernetes API on the application.

### 1. Configuration with global configmaps

It is not possible to mount a configmap to a deployment/pod/daemonset/replicaset/etc. when it is in another namespace.

In some cases though, it is needed to have a single central configuration store for components that run in different namespaces.

Namespace: user-ns-1  
 +--------------------------------------------+  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | Container |  
 | +--------------------------------------+ |  
 | | | |  
 | | | |  
 Namespace: logcollector +---------+ Running executable | |  
+----------------------------------------------+ | | | | |  
| | | | | | |  
| | | | +--------------------------------------+ |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| Configmap | | | |  
| +------------------------------------+ | | | |  
| | | | watches | +--------------------------------------------+  
| | data: | | reads |  
| | buffer: 2048 +<-------------------+  
| | target: file | | | Namespace: user-ns-2  
| | | | | +--------------------------------------------+  
| +------------------------------------+ | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | Container |  
+----------------------------------------------+ | | +--------------------------------------+ |  
 | | | | |  
 | | | | |  
 +---------+ Running executable | |  
 | | | |  
 | | | |  
 | +--------------------------------------+ |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 +--------------------------------------------+

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: logcollector-config  
 namespace: logcollector  
data:  
 buffer: "2048"  
 target: "file"  
  
---  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: logcollector  
 name: logcollector-deployment  
 namespace: user-ns-1  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: logcollector  
 template:  
 metadata:  
 labels:  
 app: logcollector  
 spec:  
 containers:  
 - name: server

In the example above, imagine there is a log collector system for pods, and a deployment for log collector agent is created per namespace. That is why the namespaces of the configmap and the deployment are different. The deployment per namespace for log collector is created by another controller, which is not shown above.

In the log collector applications within each namespace, the application goes and reads the *central* configmap for configuration. Please note that the configmap is not mounted to the container. The application needs to use Kubernetes API to read the configmap.

package main  
  
import ...  
  
func main(){  
 if \_, err := kubeclient.Get(ctx).CoreV1().ConfigMaps("logcollector").Get("logcollector-config", metav1.GetOptions{}); err == nil {  
 watchConfigmap("logcollector" ,"logcollector-config", func(configMap \*v1.ConfigMap) {  
 updateConfig(configMap)  
 })  
 } else if apierrors.IsNotFound(err) {  
 log.Fatal("Global ConfigMap 'logcollector-config' in namespace 'logcollector' does not exist")  
 } else {  
 log.Fatal("Error reading global ConfigMap 'logcollector-config' in namespace 'logcollector'")  
 }  
}

Pseudo code above gets the configmap, starts watching it and if there is any change, it updates the config. It is not necessary to roll out a new deployment.

The RBAC settings for getting, reading and watching configmaps should be handled.

### 2. Configuration with global and namespaced configmaps

[Configuration with global configmaps](TODO:link) provides the ability to use a central configuration, but often it is needed to override some part of the configuration per namespace.

In this case, a separate watch is needed for the namespaced configmap.

Namespace: user-ns-1  
 +---------------------------------------------------------------------------------------------------+  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | Container Configmap |  
 | +--------------------------------------+ +------------------------------------+ |  
 | | | watches | | |  
 | | | reads | data: | |  
 Namespace: logcollector +---------+ Running executable +--------------->+ buffer: 1024 | |  
+----------------------------------------------+ | | | | | target: file | |  
| | | | | | | | |  
| | | | +--------------------------------------+ +------------------------------------+ |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| Configmap | | | |  
| +------------------------------------+ | | | |  
| | | | watches | +---------------------------------------------------------------------------------------------------+  
| | data: | | reads |  
| | buffer: 2048 +<-------------------+  
| | target: file | | | Namespace: user-ns-2  
| | | | | +---------------------------------------------------------------------------------------------------+  
| +------------------------------------+ | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | |  
| | | | Container Configmap |  
+----------------------------------------------+ | | +--------------------------------------+ +------------------------------------+ |  
 | | | | watches | | |  
 | | | | reads | data: | |  
 +---------+ Running executable +--------------->+ buffer: 4096 | |  
 | | | | target: remote | |  
 | | | | | |  
 | +--------------------------------------+ +------------------------------------+ |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 | |  
 +---------------------------------------------------------------------------------------------------+

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: logcollector-config  
 namespace: logcollector  
data:  
 buffer: "2048"  
  
---  
apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: logcollector-config  
 namespace: user-ns-1  
data:  
 buffer: "1024"  
 target: "file"  
  
---  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 labels:  
 app: logcollector  
 name: logcollector-deployment  
 namespace: user-ns-1  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: logcollector  
 template:  
 metadata:  
 labels:  
 app: logcollector  
 spec:  
 containers:  
 - name: server  
 env:  
 - name: NAMESPACE  
 valueFrom:  
 fieldRef:  
 fieldPath: metadata.namespace

There are 2 configmaps now, one in logcollector namespace that is considered the *global* configuration. There is another one in the same namespace as the log collector instance in user-ns-1 namespace.

package main  
  
import ...  
  
func main(){  
 if namespace := os.Getenv("NAMESPACE"); ns == "" {  
 panic("Unable to determine application namespace")  
 }  
  
 var global \*v1.ConfigMap  
 var local \*v1.ConfigMap  
  
 if \_, err := kubeclient.Get(ctx).CoreV1().ConfigMaps("logcollector").Get("logcollector-config", metav1.GetOptions{}); err == nil {  
 watchConfigmap("logcollector" ,"logcollector-config", func(configMap \*v1.ConfigMap) {  
 global = configMap  
 config := mergeConfig(global, local)  
 updateConfig(config)  
 })  
 } else if apierrors.IsNotFound(err) {  
 log.Fatal("Global ConfigMap 'logcollector-config' in namespace 'logcollector' does not exist")  
 } else {  
 log.Fatal("Error reading global ConfigMap 'logcollector-config' in namespace 'logcollector'")  
 }  
  
 if \_, err := kubeclient.Get(ctx).CoreV1().ConfigMaps(namespace).Get("logcollector-config", metav1.GetOptions{}); err == nil {  
 watchConfigmap(namespace ,"logcollector-config", func(configMap \*v1.ConfigMap) {  
 global = configMap  
 config := mergeConfig(global, local)  
 updateConfig(config)  
 })  
 } else if apierrors.IsNotFound(err) {  
 log.Infof("Local ConfigMap 'logcollector-config' in namespace '%s' does not exist", namespace)  
 } else {  
 log.Fatalf("Error reading local ConfigMap 'logcollector-config' in namespace '%s'", namespace)  
 }  
}  
  
func mergeConfig(global, local \*v1.ConfigMap) (...){  
 // merge 2 configmaps here and return the result  
}

The application will first check the global config, then the namespaced config. The configuration in the namespaced configmap will have precedence.

It is ok to not have any local configmaps. Unlike when the global configmap is missing, the program will not stop execution when the local configmap does not exist.

It would also be possible to simply mount the namespaced configmap into the log collector container, however, in that case, the ability to reload the config without any restarts would be lost.

The application needs to know the container namespace it is running because the local configmap is not mounted using the configmap mounting and the application needs to GET the local configmap. It needs to tell Kubernetes API the namespace it is looking for the configmap in.

### 3. Configuration with custom resources

Custom resources are a way to extend Kubernetes API.

apiVersion: apiextensions.k8s.io/v1  
kind: CustomResourceDefinition  
metadata:  
 name: eventconsumers.example.com  
spec:  
 group: example.com  
 versions:  
 - name: v1  
 ...  
 scope: Namespaced  
 names:  
 plural: eventconsumers  
 singular: eventconsumer  
 kind: EventConsumer  
 ...

After applying the custom resource definition above, it will be possible to create a EventConsumer resource on Kubernetes:

apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-1  
 namespace: user-ns-1  
spec:  
 source: source1.foobar.com:443  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-2  
 namespace: user-ns-2  
spec:  
 source: source2.foobar.com:443

Custom resources can be used as configuration that are to be read by controllers.

t=0  
...................................................................................................................  
  
  
 EventConsumer custom resource  
 consumer-1  
 +------------------------------+  
 | spec: |  
 | source: | fed to  
 | source1.foobar.com:443 +------------+  
 | | |  
 +------------------------------+ | Event consumer controller  
 | +--------------------------+  
 | | |  
 +------> |  
 EventConsumer custom resource | | |  
 consumer-2 | +--------------------------+  
 +------------------------------+ |  
 | spec: | |  
 | source: | |  
 | source2.foobar.com:443 +------------+  
 | | fed to  
 +------------------------------+  
  
  
  
t=1  
....................................................................................................................  
  
  
 Event consumer app instance source1.foobar.com:443  
 +------------------------+ +----------------------+  
 creates | | pulls events from | |  
 +---------> +-----------------------> |  
 | | | | |  
 | +------------------------+ +----------------------+  
 Event consumer controller |  
+--------------------------+ |  
| | |  
| +---+  
| | |  
+--------------------------+ |  
 |  
 |  
 | Event consumer app instance source2.foobar.com:443  
 | +------------------------+ +----------------------+  
 | | | pulls events from | |  
 +---------> +-----------------------> |  
 creates | | | |  
 +------------------------+ +----------------------+

In this case, the custom resource EventConsumer provides configuration for the event consumption applications that are created by the event constumer controller.

Note the scope: Namespaced key and value in custom resource definition. This tells Kubernetes that the custom resources created for this definition will live in a namespace.

Having a strong API for fields in the config is a nice feature. With rules defined in the custom resource definition, we can leverage Kubernetes’ validation without any hassle.

However, there are drawbacks too. The first one is that the custom resource needs to be maintained carefully, and you cannot simply delete fields or update fields in an incompatible way without releasing a new version of the custom resource. Adding new fields is good.

The second drawback is that not everything can be specified strongly in advance. Imagine a program that can use one of the supported databases. The database clients will have different configs and the config shapes will be different. It will not be possible to simply have fields in the custom resource to pass them as a whole to the database client.

### 4. Configuration with custom resources, falling back to configmaps

This pattern is an extended version of the [Configuration with custom resources](#X53cc7af7c5b303591c47c04a0b0df4c64fa37bf) pattern.

In this case, the configuration options that are certain and different in every scenario are put into the custom resource as strongly typed API fields. The options that could be common between different custom resources are not put in the custom resources but in configmaps.

t=0  
...............................................................................................................................................................  
  
 Namespace user-ns-1 Namespace eventconsumer  
+-------------------------------------------+ +------------------------------------------------------------------------------------------------------+  
| | | |  
| EventConsumer custom resource | | |  
| consumer-1 | | |  
| +------------------------------+ | | |  
| | spec: | | | |  
| | source: | fed to| | |  
| | source1.foobar.com:443 +---------------+ | |  
| | | | | | |  
| +------------------------------+ | | | |  
| | | | |  
| Configmap | | | Event consumer controller |  
| +------------------------------+ | | | reads +--------------------------+ |  
| | | | | | watches | | |  
| | data: <----------------------------------+ | |  
| | buffer:4096 | | | | | | Configmap |  
| | | | | | | | +-------------------------+ |  
| +------------------------------+ | | | | | | | |  
| | | | | | reads | data: | |  
| | | | | | watches | buffer:2048 | |  
+-------------------------------------------+ | | | +-------------------------> privateKey: | |  
 | | | | | MIICdQI | |  
 +------------------> | | | |  
 | | | | +-------------------------+ |  
 | | +--------------------------+ |  
 | | |  
 | | |  
 Namespace user-ns-2 | | |  
+--------------------------------------------+ | | |  
| | | | |  
| EventConsumer custom resource | | | |  
| consumer-2 | | | |  
| +------------------------------+ | | | |  
| | spec: | | | | |  
| | source: | | | | |  
| | source2.foobar.com:443 +---------------+ | |  
| | privateKey: | fed to | | |  
| | foobarbaz | | | |  
| | | | | |  
| +------------------------------+ | | |  
| | | |  
| | | |  
| | | |  
| | | |  
| | | |  
| | | |  
| | | |  
| | | |  
+--------------------------------------------+ +------------------------------------------------------------------------------------------------------+  
  
  
  
  
  
  
  
  
  
t=1  
...............................................................................................................................................................  
  
 Namespace user-ns-1  
 +-----------------------------------+  
 | |  
 | Event consumer instance | source1.foobar.com:443  
 | +------------------------+ | +----------------------+  
 creates | | | | pulls events from | |  
 +---------------> +-------------------------------------------> |  
 | | | | | using buffer: 4096 | |  
 | | +------------------------+ | using privateKey: MIICdQI +----------------------+  
 Event consumer controller | | |  
 +--------------------------+ | | |  
 | | | +-----------------------------------+  
 | +---+  
 | | |  
 +--------------------------+ | Namespace user-ns-2  
 | +-----------------------------------+  
 | | |  
 | | Event consumer instance | source2.foobar.com:443  
 | | +------------------------+ | +----------------------+  
 | | | | | pulls events from | |  
 +---------------> +-------------------------------------------> |  
 creates | | | | using buffer: 2048 | |  
 | +------------------------+ | using privateKey: foobarbaz +----------------------+  
 | |  
 | |  
 +-----------------------------------+

apiVersion: apiextensions.k8s.io/v1  
kind: CustomResourceDefinition  
metadata:  
 name: eventconsumers.example.com  
spec:  
 group: example.com  
 versions:  
 - name: v1  
 ...  
 scope: Namespaced  
 names:  
 plural: eventconsumers  
 singular: eventconsumer  
 kind: EventConsumer  
 ...  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-1  
 namespace: user-ns-1  
spec:  
 source: source1.foobar.com:443  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-2  
 namespace: user-ns-2  
spec:  
 source: source2.foobar.com:443  
 privateKey: |  
 -----BEGIN PRIVATE KEY-----  
 foobarbazfoobar  
 barfoobarbazfoo  
 bazfoobarfoobar  
 -----END PRIVATE KEY-----  
  
---  
  
apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: eventconsumer-config  
 namespace: eventconsumer  
data:  
 buffer: "2048"  
 privateKey: |  
 -----BEGIN PRIVATE KEY-----  
 MIICdQIBADANBgk  
 nC45zqIvd1QXloq  
 bokumO0HhjqI12a  
 -----END PRIVATE KEY-----  
---  
apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: eventconsumer-config  
 namespace: user-ns-1  
data:  
 buffer: "4096"  
---

In the example above, EventConsumer consumer-1 is connecting to source1.foobar.com:8443 URL to consume events. consumer-2 though, is connecting to another URL.

The global configmap in eventconsumer-config contains the buffer size as well as the private key. These will be used by the event consumer controllers for EventConsumer custom resources unless there are overrides in local configmap or in overrides in the custom resources.

In user-ns-1 namespace, there is also a relevant configmap, which overrides the buffer size for the EventConsumer in that namespace. No local configmap exists for consumer-2 and no buffer size override exists in the custom resource, so the default buffer size in the global configmap will be used for that.

In consumer-2 custom resource, there’s also a privateKey defined, so the private key defined in global configmap is not used.

This pattern is a mix of [Configuration with custom resources](#X53cc7af7c5b303591c47c04a0b0df4c64fa37bf) and [Configuration with global and namespaced configmaps](#X527707b68d70d7b2168cf392126093c226342b6) patterns.

The pattern helps with sharing common information for custom resources. The buffer size or the private key could have been hardcoded in the application, but instead, in this case even the defaults are configurable.

Similar to the note in [Configuration with global and namespaced configmaps](#X527707b68d70d7b2168cf392126093c226342b6) pattern, a better approach could be creating a separate global configuration custom resource that is cluster scoped and that contains the shared configuration. That might then additionally bring the namespaced configuration custom resources and so on. If there are a lot of options and if they are complex to build and validate, it might be a better way to choose the global configuration custom resource approach as Kubernetes OpenAPI validation can be leveraged.

### 5. References to configmaps in custom resources

The pattern [Configuration with custom resources, falling back to configmaps](#Xe614999ffaafce07b4e2c0020572939f80640e5) helps with sharing the configuration for multiple custom resources, however, it does not provide the ability to specify multiple configurations for custom resources in the same namespace.

This pattern provides that by keeping the complicated and long configuration in configmaps, which is not desired to keep in the custom resource spec.

t=0  
...............................................................................................................................................................  
  
  
 EventConsumer custom resource Event consumer controller  
 consumer-1-a +--------------------------+  
 +----------------------------------+ | |  
 references |spec: | | |  
 +----------------+ source: source1a.foobar.com:443 | | |  
 | | connectionConfig:onfig: +---------------+ | |  
 | | ref: throttled | | | |  
 | | | | | |  
 | +----------------------------------+ | | |  
 | | | |  
 | +------------------> |  
 | EventConsumer custom resource | | |  
 | consumer-1-b | | |  
 | +----------------------------------+ | | |  
 | |spec: | | | |  
 | references | source: source1b.foobar.com:443 | fed to | | |  
 +----------------+ connectionConfig: +---------------+ | |  
 | | ref: throttled | | |  
 | | | | |  
 | +----------------------------------+ | |  
 | | |  
 | | |  
 | Configmap throttled | |  
 | +-----------------------+ reads | |  
 | | | watches | |  
 +----------------------> data: <---------------------------------------+ |  
 | lotsOfConfig:here | | |  
 | | | |  
 +-----------------------+ | |  
 | |  
 | |  
 | |  
 EventConsumer custom resource | |  
 consumer-2 | |  
 +----------------------------------+ | |  
 |spec: | | |  
 references | source: source2.foobar.com:443 | fed to | |  
 +----------------+ connectionConfig: +----------------------------------> |  
 | | ref: unlimited | | |  
 | | | | |  
 | +----------------------------------+ | |  
 | | |  
 | | |  
 | Configmap unlimited | |  
 | +---------------------------+ reads | |  
 | | | watches | |  
 +-----------------------> data: <----------------------------------+ |  
 | otherTypeOfConfig: here | | |  
 | | | |  
 +---------------------------+ | |  
 | |  
 | |  
 +--------------------------+  
  
t=1  
......................................................................................................................................................  
  
  
  
  
 Event consumer instance source1a.foobar.com:443  
 +------------------------+ +----------------------+  
 creates | | pulls events from | |  
 +---------------> +-------------------------------------------> |  
 | | | using connection settings | |  
 | +------------------------+ for throttled connection +----------------------+  
 Event consumer controller |  
 +--------------------------+ |  
 | | |  
 | | |  
 | | | Event consumer instance source1b.foobar.com:443  
 | | | +------------------------+ +----------------------+  
 | | | | | pulls events from | |  
 | +-------------------> +-------------------------------------------> |  
 | | | | | using connection settings | |  
 | | | +------------------------+ for throttled connection +----------------------+  
 | | |  
 | | |  
 | | |  
 +--------------------------+ |  
 | Event consumer instance source2.foobar.com:443  
 | +------------------------+ +----------------------+  
 | | | pulls events from | |  
 +---------------> +-------------------------------------------> |  
 creates | | using connection settings | |  
 +------------------------+ for unlimited connection +----------------------+

apiVersion: apiextensions.k8s.io/v1  
kind: CustomResourceDefinition  
metadata:  
 name: eventconsumers.example.com  
spec:  
 group: example.com  
 versions:  
 - name: v1  
 ...  
 scope: Namespaced  
 names:  
 plural: eventconsumers  
 singular: eventconsumer  
 kind: EventConsumer  
 ...  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-1-a  
spec:  
 source: source1a.foobar.com:443  
 connectionConfig:  
 ref: throttled  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-1-b  
spec:  
 source: source1b.foobar.com:443  
 connectionConfig:  
 ref: throttled  
  
---  
  
apiVersion: example.com/v1  
kind: EventConsumer  
metadata:  
 name: consumer-2  
spec:  
 source: source2.foobar.com:443  
 connectionConfig:  
 ref: unlimited  
  
---  
  
apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: throttled  
data:  
 lotsOfConfig: here  
  
---  
  
apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: unlimited  
data:  
 otherTypeOfConfig: here  
  
---

The custom resource references the configmap in it. The configmap can be referenced in multiple custom resources in the same namespace while it is possible to use reference different configmaps in the custom resources that are in the same namespace.

Furthermore, the configuration in the configmap can even be referenced in custom resources of different types. For example, the TLS settings can be kept in a configmap and referenced in a Kafka client component custom resource and in a CouchDB client component custom resource.

### 6. Using configmaps with labels

TBA

This is about getting the configuration from configmaps, similar to patterns above. However, instead of getting fetching configmaps by their names, we fetch them by labels. That also provides using multiple configmaps locally.

One example is, creating consumer pods per configmap. e.g. instead of using CRs to get some controller create consumer pods, use configmaps. So, a controller will watch the configmaps with these labels, creates consumer pods for them. This is normally done with CRs but in case of credentials, one still need to create secrets because you can’t put credentials in CRs. So, with CRs, there would be a CR and a secret per consumer pod.