We can use both in the same way, as both provide storage and management of key-value pairs. When we are describing ConfigMaps, the same can be applied most of the time to Secrets too. Besides the actual data encoding (which is Base64 for Secrets), there is no technical difference for the use of ConfigMaps and Secrets.

Once a ConfigMap is created and holding data, we can use the keys of a ConfigMap in two ways:

- As a reference for *environment variables*, where the key is the name of the environment variable.
- As *files* that are mapped to a volume mounted in a Pod. The key is used as the filename.

The file in a mounted ConfigMap volume is updated when the ConfigMap is updated via the Kubernetes API. So, if an application supports hot reload of configuration files, it can immediately benefit from such an update. However, with ConfigMap entries used as environment variables, updates are not reflected because environment variables can't be changed after a process has been started.

In addition to ConfigMap and Secret, another alternative is to store configuration directly in external volumes that are then mounted.

The following examples concentrate on ConfigMap usage, but they can also be used for Secrets. There is one big difference, though: values for Secrets have to be Base64 encoded.

A ConfigMap resource contains key-value pairs in its `data` section, as shown in Example 20-1.

Example 20-1. ConfigMap resource

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: random-generator-config
data:
  PATTERN: Configuration Resource   ❶
  application.properties: |
    # Random Generator config
    log.file=/tmp/generator.log
    server.port=7070
  EXTRA_OPTIONS: "high-secure,native"
  SEED: "432576345"
```

❶ ConfigMaps can be accessed as environment variables and as a mounted file. We recommend using uppercase keys in the ConfigMap to indicate an EnvVar usage and proper filenames when used as mounted files.

We see here that a ConfigMap can also carry the content of complete configuration files, like the Spring Boot `application.properties` in this example. You can imagine that for a nontrivial use case, this section could get quite large!

Instead of manually creating the full resource descriptor, we can use `kubectl` to create ConfigMaps or Secrets too. For the preceding example, the equivalent `kubectl` command looks like that in Example 20-2.

Example 20-2. Create a ConfigMap from a file

```
kubectl create cm spring-boot-config \
   --from-literal=PATTERN="Configuration Resource" \
   --from-literal=EXTRA_OPTIONS="high-secure,native" \
   --from-literal=SEED="432576345" \
   --from-file=application.properties
```

This ConfigMap then can be read in various places—everywhere environment variables are defined, as demonstrated Example 20-3.

Example 20-3. Environment variable set from ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef:
          name: random-generator-config
          key: PATTERN
....
```

If a ConfigMap has many entries that you want to consume as environment variables, using a certain syntax can save a lot of typing. Rather than specifying each entry individually, as shown in the preceding example in the `env` section, `envFrom` allows you to expose all ConfigMap entries that have a key that also can be used as a valid environment variable. We can prepend this with a prefix, as shown in Example 20-4. Any key that cannot be used as an environment variable is ignored (e.g., `"illeg.al"`). When multiple ConfigMaps are specified with duplicate keys, the last entry in `env From` takes precedence. Also, any same-named environment variable set directly with `env` has higher priority.

Example 20-4. Setting all entries of a ConfigMap as environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    envFrom:                       ❶
    - configMapRef:
        name: random-generator-config
      prefix: CONFIG_              ❷
```

❶ Pick up all keys from the ConfigMap `random-generator-config` that can be used as environment variable names.

❷ Prefix all suitable ConfigMap keys with `CONFIG_`. With the ConfigMap defined in Example 20-1, this leads to three exposed environment variables: `CONFIG_PATTERN_NAME`, `CONFIG_EXTRA_OPTIONS`, and `CONFIG_SEED`.

Secrets, as with ConfigMaps, can also be consumed as environment variables, either per entry or for all entries. To access a Secret instead of a ConfigMap, replace `configMapKeyRef` with `secretKeyRef`.

When a ConfigMap is used as a volume, its complete content is projected into this volume, with the keys used as filenames. See Example 20-5.

Example 20-5. Mount a ConfigMap as a volume

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:  ❶
      name: random-generator-config
```

❶ A ConfigMap-backed volume will contain as many files as entries, with the map's keys as filenames and the map's values as file content.

The configuration in Example 20-1 that is mounted as a volume results in four files in the */config* folder: an *application.properties* file with the content defined in the ConfigMap and the files *PATTERN*, *EXTRA_OPTIONS*, and *SEED*, each with a single line of content.

The mapping of configuration data can be fine-tuned more granularly by adding additional properties to the volume declaration. Rather than mapping all entries as files, you can also individually select every key that should be exposed, the filename, and permissions under which it should be available. Example 20-6 demonstrates how you can granularly select which parts of a ConfigMap are exposed as volumes.

Example 20-6. Expose ConfigMap entries selectively as volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap:
      name: random-generator-config
      items:                          ❶
      - key: application.properties   ❷
        path: spring/myapp.properties
        mode: 0400
```

❶ List of ConfigMap entries to expose as volumes.

❷ Expose only `application.properties` from the ConfigMap under the path `spring/myapp.properties` with file mode 0400.

As you have seen, changes to a ConfigMap are directly reflected in a projected volume that contains the ConfigMap's content as files. An application can watch those files and immediately pick up any changes. This hot reload is very useful to avoid a redeployment of an application, which can cause an interruption of the service. On the other hand, such live changes are not tracked anywhere and can easily get lost during a restart. These ad hoc chances can cause configuration drift that is hard to detect and analyze. That is one of the reasons many people prefer an *immutable configuration* that stays constant once deployed. We have dedicated a whole pattern in

Chapter 21, "Immutable Configuration", to this paradigm, but there is a cheap way to easily achieve this with ConfigMap and Secrets too.

How Secure Are Secrets?

Secrets hold Base64-encoded data and decode it before passing it to a Pod either as environment variables or mounted volume. This is very often confused as a security feature. Base64 encoding is not an encryption method, and from a security perspective, it is considered the same as plain text. Base64 encoding in Secrets allows you to store binary data, so why are Secrets considered more secure than ConfigMaps? There are a number of other implementation details of Secrets that make them secure. Constant improvements are occurring in this area, but the main implementation details currently are as follows:

- A Secret is distributed only to nodes running Pods that need access to the Secret.
- On the nodes, Secrets are stored in memory in a `tmpfs` and never written to physical storage, and they are removed when the Pod is removed.
- In etcd, the backend storage for the Kubernetes API, Secrets can be stored in encrypted form.

Regardless of all that, there are still ways to get access to Secrets as a root user, or even by creating a Pod and mounting a Secret. You can apply role-based access control (RBAC) to Secrets (as you can do to ConfigMaps or other resources) and allow only certain Pods with predefined service accounts to read them. We explain RBAC in great length in Chapter 26, "Access Control". But users who have the ability to create Pods in a namespace can still escalate their privileges within that namespace by creating Pods. They can run a Pod under a greater-privileged service account and still read Secrets. A user or a controller with Pod-creation access in a namespace can impersonate any service account and access all Secrets and ConfigMaps in that namespace. Thus, additional encryption of sensitive information is often done at the application level too. In Chapter 25, "Secure Configuration", you'll learn several ways to make Secrets more secure, especially in a GitOps context.

Since version 1.21, Kubernetes supports an `immutable` field for ConfigMaps and Secrets that, if set to `true`, prevents the resource from being updated once created. Besides preventing unwanted updates, using immutable ConfigMaps and Secrets considerably improves a cluster's performance as the Kubernetes API server does not need to monitor changes on those immutable objects. Example 20-7 shows how to declare a Secret immutable. The only way to change such a Secret after it has been stored on the cluster is to delete and recreate the updated Secret. Any running Pod referencing this secret needs to be restarted too.

Example 20-7. Immutable Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: random-config
data:
  user: cm9sYW5k
immutable: true ❶
```

❶ Boolean flag declaring the mutability of the Secret (default is `false`).

Discussion

ConfigMaps and Secrets allow you to store configuration information in dedicated resource objects that are easy to manage with the Kubernetes API. The most significant advantage of using ConfigMaps and Secrets is that they decouple the *definition* of configuration data from its *usage*. This decoupling allows us to manage the objects that use the configuration independently of the configuration definition. Another benefit of ConfigMaps and Secrets is that they are intrinsic features of the platform. No custom construct like that in Chapter 21, "Immutable Configuration", is required.

However, these configuration resources also have their restrictions: with a 1 MB size limit for Secrets, they can't store arbitrarily large data and are not well suited for nonconfiguration application data. You can also store binary data in Secrets, but since they have to be Base64 encoded, you can use only around 700 KB data for it. Real-world Kubernetes clusters also put an individual quota on the number of ConfigMaps that can be used per namespace or project, so ConfigMap is not a golden hammer.

The next two chapters show how to deal with large configuration data by using the *Immutable Configuration* and *Configuration Template* patterns.

More Information

- Configuration Resource Example
- Configure a Pod to Use a ConfigMap
- Secrets
- Encrypting Secret Data at Rest
- Distribute Credentials Securely Using Secrets
- Immutable Secrets
- How to Create Immutable ConfigMaps and Secrets
- Size Limit for a ConfigMap

CHAPTER 21

Immutable Configuration

The *Immutable Configuration* pattern offers two ways to make configuration data immutable so that your application's configuration is always in a well-known and recorded state. With this pattern, we can not only use immutable and versioned configuration data, but also overcome the size limitation of configuration data stored in environment variables or ConfigMaps.

Problem

As you saw in Chapter 19, "EnvVar Configuration", environment variables provide a simple way to configure container-based applications. And although they are easy to use and universally supported, as soon as the number of environment variables exceeds a certain threshold, managing them becomes hard.

This complexity can be handled to some degree by using *Configuration Resources*, as described in Chapter 20, "Configuration Resource", which since Kubernetes 1.21 can be declared as *immutable*. However, ConfigMaps still have a size limitation, so if you work with large configuration data (like precomputed data models in a machine learning context), then ConfigMaps are not suitable even when marked as immutable.

Immutability here means that we can't change the configuration after the application has started, in order to ensure that we always have a well-defined state for our configuration data. In addition, immutable configuration can be put under version control and follow a change control process.

Solution

There are several options to address the concern of configuration immutability. The simplest and preferred option is to use ConfigMaps or Secrets that are marked as immutable in their declaration. You learned about immutable ConfigMaps in Chapter 20. ConfigMaps should be the first choice if your configuration fits into a ConfigMap and is reasonably easy to maintain. In real-world scenarios, however, the amount of configuration data can increase quickly. Although a WildFly application server configuration might still fit in a ConfigMap, it is quite huge. It becomes really ugly when you have to nest XML or YAML within YAML—i.e., when the content of your configuration is also YAML and you embed this as within the ConfigMaps YAML section. Editor support for such use cases is limited, so you have to be very careful about the indentation, and even then, you will probably mess it up more than once (believe us!). Another nightmare is having to maintain tens or hundreds of entries in a single ConfigMap because your application requires many different configuration files. Although this pain can be mitigated to some degree with good tooling, large configuration data sets like pretrained machine learning data models are just impossible with ConfigMap because of the backend size restriction of 1 MB.

To address the concern of complex configuration data, we can put all environment-specific configuration data into a single, passive data image that we can distribute as a regular container image. During runtime, the application and the data image are linked together so that the application can extract the configuration from the data image. With this approach, it is easy to craft different configuration data images for various environments. These images then combine all configuration information for specific environments and can be versioned like any other container image.

Creating such a data image is trivial, as it is a simple container image that contains only data. The challenge is the linking step during startup. We can use various approaches, depending on the platform.

Docker Volumes

Before looking at Kubernetes, let's go one step back and consider the vanilla Docker case. In Docker, it is possible for a container to expose a *volume* with data from the container. With a `VOLUME` directive in a Dockerfile, you can specify a directory that can be shared later. During startup, the content of this directory within the container is copied over to this shared directory. As shown in Figure 21-1, this volume linking is an excellent way to share configuration information from a dedicated configuration container with another application container.

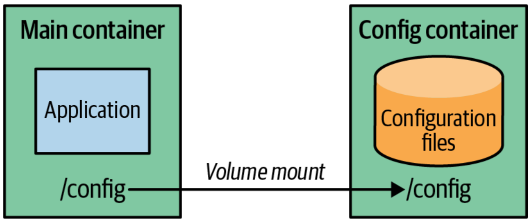

Figure 21-1. Immutable configuration with Docker volume

Let's have a look at an example. For the development environment, we create a Docker image that holds the developer configuration and creates a volume with mount point */config*. We can create such an image with `Dockerfile-config`, as in Example 21-1.

Example 21-1. Dockerfile for a configuration image

```
FROM scratch
ADD app-dev.properties /config/app.properties   ❶
VOLUME /config                                  ❷
```

❶ Add specified property.

❷ Create volume and copy property into it.

We now create the image itself and the Docker container with the Docker CLI in Example 21-2.

Example 21-2. Building the configuration Docker image

```
docker build -t k8spatterns/config-dev-image:1.0.1 -f Dockerfile-config .
docker create --name config-dev k8spatterns/config-dev-image:1.0.1 .
```

The final step is to start the application container and connect it to this configuration container (Example 21-3).

Example 21-3. Start application container with config container linked

```
docker run --volumes-from config-dev k8spatterns/welcome-servlet:1.0
```

The application image expects its configuration files to be within a */config* directory, the volume exposed by the configuration container. When you move this application from the development environment to the production environment, all you have to do is change the startup command. There is no need to alter the application image itself. Instead, you simply volume-link the application container with the production configuration container, as seen in Example 21-4.

Example 21-4. Use different configuration for production environment

```
docker build -t k8spatterns/config-prod-image:1.0.1 -f Dockerfile-config .
docker create --name config-prod k8spatterns/config-prod-image:1.0.1 .
docker run --volumes-from config-prod k8spatterns/welcome-servlet:1.0
```

Kubernetes Init Containers

In Kubernetes, volume sharing within a Pod is perfectly suited for this kind of linking of configuration and application containers. However, if we want to transfer this technique of Docker volume linking to the Kubernetes world, we will find that there is currently no support for container volumes in Kubernetes. Considering the age of the discussion and the complexity of implementing this feature versus its limited benefits, it's likely that container volumes will not arrive anytime soon.

So containers can share (external) volumes, but they cannot yet directly share directories located within the containers. To use immutable configuration containers in Kubernetes, we can use the *Init Containers* pattern from Chapter 15 that can initialize an empty shared volume during startup.

In the Docker example, we base the configuration Docker image on `scratch`, an empty Docker image with no operating system files. We don't need anything else because we only want the configuration data shared via Docker volumes. But for Kubernetes init containers, we need help from the base image to copy over the configuration data to a shared Pod volume. A good choice for this is `busybox`, which is still small but allows us to use a plain Unix `cp` command for this task.

So how does the initialization of shared volumes with configuration work under the hood? Let's have a look at an example. First, we need to create a configuration image again with a Dockerfile, as in Example 21-5.

Example 21-5. Development configuration image

```
FROM busybox
ADD dev.properties /config-src/demo.properties
ENTRYPOINT [ "sh", "-c", "cp /config-src/* $1", "--" ]  ❶
```

❶ Using a shell here in order to resolve wildcards.

The only difference from the vanilla Docker case in Example 21-1 is that we have a different base image and we add an `ENTRYPOINT` that copies the properties file to the directory given as an argument when the container image starts. This image can now be referenced in an init container within a Deployment's `.template.spec` (see Example 21-6).

Example 21-6. Deployment that copies configuration to destination in init container

```yaml
initContainers:
- image: k8spatterns/config-dev:1
  name: init
  args:
  - "/config"
  volumeMounts:
  - mountPath: "/config"
    name: config-directory
containers:
- image: k8spatterns/demo:1
  name: demo
  ports:
  - containerPort: 8080
    name: http
    protocol: TCP
  volumeMounts:
  - mountPath: "/var/config"
    name: config-directory
volumes:
  - name: config-directory
    emptyDir: {}
```

The Deployment's Pod template specification contains a single volume and two containers:

- The volume `config-directory` is of the type `emptyDir`, so it's created as an empty directory on the node hosting this Pod.
- The init container Kubernetes calls during startup is built from the image we just created, and we set a single argument, `/config`, used by the image's `ENTRYPOINT`. This argument instructs the init container to copy its content to the specified directory. The directory */config* is mounted from the volume `config-directory`.
- The application container mounts the volume `config-directory` to access the configuration that was copied over by the init container.

Figure 21-2 illustrates how the application container accesses the configuration data created by an init container over a shared volume.

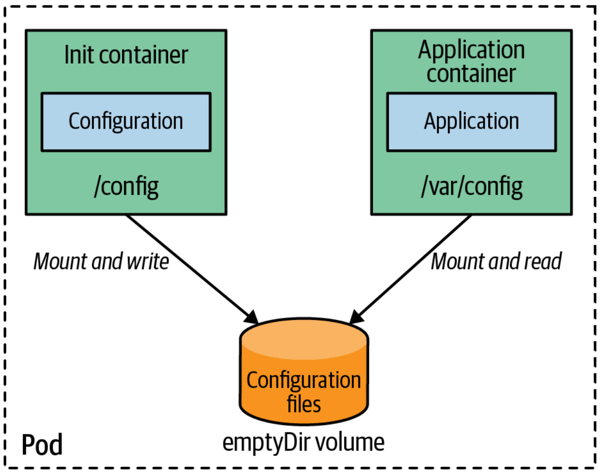

Figure 21-2. Immutable configuration with an init container

Now to change the configuration from the development to the production environment, all we need to do is exchange the image of the init container. We can do this either by changing the YAML definition or by updating with `kubectl`. However, it is not ideal to have to edit the resource descriptor for each environment. If you are on Red Hat OpenShift, an enterprise distribution of Kubernetes, *OpenShift Templates* can help address this. OpenShift Templates can create different resource descriptors for the different environments from a single template.

OpenShift Templates

OpenShift Templates are regular resource descriptors that are parameterized. As seen in Example 21-7, we can easily use the configuration image as a parameter.

Example 21-7. OpenShift Template for parameterizing config image

```
apiVersion: v1
kind: Template
metadata:
  name: demo
parameters:
  - name: CONFIG_IMAGE                              ❶
    description: Name of configuration image
    value: k8spatterns/config-dev:1
objects:
- apiVersion: apps/v1
  kind: Deployment
    // ....
    spec:
```

```
  template:
    metadata:
      // ....
      spec:
        initContainers:
        - name: init
          image: ${CONFIG_IMAGE}          ❷
          args: [ "/config" ]
          volumeMounts:
          - mountPath: /config
            name: config-directory
        containers:
        - image: k8spatterns/demo:1
          // ...
          volumeMounts:
          - mountPath: /var/config
            name: config-directory
      volumes:
      - name: config-directory
        emptyDir: {}
```

❶ Template parameter `CONFIG_IMAGE` declaration.

❷ Use of the template parameter.

We show here only a fragment of the full descriptor, but you can quickly recognize the parameter `CONFIG_IMAGE` we reference in the init container declaration. If we create this template on an OpenShift cluster, we can instantiate it by calling `oc`, as in Example 21-8.

Example 21-8. Applying OpenShift template to create new application

```
oc new-app demo -p CONFIG_IMAGE=k8spatterns/config-prod:1
```

Detailed instructions for running this example, as well as the full Deployment descriptors, can be found as usual in our example Git repository.

Discussion

Using data containers for the *Immutable Configuration* pattern is admittedly a bit involved. Use these only if immutable ConfigMaps and Secret are not suitable for your use case.

Data containers have some unique advantages:

- Environment-specific configuration is sealed within a container. Therefore, it can be versioned like any other container image.

- Configuration created this way can be distributed over a container registry. The configuration can be examined even without accessing the cluster.
- The configuration is immutable, as is the container image holding the configuration: a change in the configuration requires a version update and a new container image.
- Configuration data images are useful when the configuration data is too complex to put into environment variables or ConfigMaps, since it can hold arbitrarily large configuration data.

As expected, the *Immutable Configuration* pattern also has certain drawbacks:

- It has higher complexity, because extra container images need to be built and distributed via registries.
- It does not address any of the security concerns around sensitive configuration data.
- Since no image volume support is actually available for Kubernetes workloads, the technique described here is still limited for use cases where the overhead of copying over data from init containers to a local volume is acceptable. We hope that eventually mounting container images directly as volumes will be possible in the future, but as of 2023, only experimental CSI support is available.
- Extra init container processing is required in the Kubernetes case, and hence we need to manage different Deployment objects for different environments.

All in all, you should carefully evaluate whether such an involved approach is really required.

Another approach for dealing with large configuration files that differ only slightly from environment to environment is described with the *Configuration Template* pattern, the topic of the next chapter.

More Information

- Immutable Configuration Example
- How to Mimic `--volumes-from` in Kubernetes
- Immutable ConfigMaps
- Feature Request: Image Volumes and Container Volumes
- docker-flexvol: A Kubernetes Driver That Supports Docker Volumes
- Red Hat OpenShift: Using Templates
- Kubernetes CSI Driver for Mounting Images

CHAPTER 22

Configuration Template

The *Configuration Template* pattern enables you to create and process large and complex configurations during application startup. The generated configuration is specific to the target runtime environment as reflected by the parameters used in processing the configuration template.

Problem

In Chapter 20, "Configuration Resource", you saw how to use the Kubernetes native resource objects ConfigMap and Secret to configure applications. But sometimes configuration files can get large and complex. Putting the configuration files directly into ConfigMaps can be problematic since they have to be correctly embedded in the resource definition. We need to be careful and avoid using special characters like quotes and breaking the Kubernetes resource syntax. The size of configurations is another consideration, as there is a limit on the sum of all values of ConfigMaps or Secrets, which is 1 MB (a limit imposed by the underlying backend store etcd).

Large configuration files typically differ only slightly for the different execution environments. This similarity leads to a lot of duplication and redundancy in the ConfigMaps because each environment has mostly the same data. The *Configuration Template* pattern we explore in this chapter addresses these specific use-case concerns.

Solution

To reduce duplication, it makes sense to store only the *differing* configuration values like database connection parameters in a ConfigMap or even directly in environment variables. During startup of the container, these values are processed with configuration templates to create the full configuration file (like a WildFly *standalone.xml*).

There are many tools like *Tiller* (Ruby) or *Gomplate* (Go) for processing templates during application initialization. Figure 22-1 is a configuration template example filled with data coming from environment variables or a mounted volume, possibly backed by a ConfigMap.

Before the application is started, the fully processed configuration file is put into a location where it can be directly used like any other configuration file.

There are two techniques for how such live processing can happen during runtime:

- We can add the template processor as part of the `ENTRYPOINT` to a Dockerfile so the template processing becomes directly part of the container image. The entry point here is typically a script that first performs the template processing and then starts the application. The parameters for the template come from environment variables.
- With Kubernetes, a better way to perform initialization is with an init container of a Pod in which the template processor runs and creates the configuration for the application containers in the Pod. The *Init Container* pattern is described in detail in Chapter 15.

For Kubernetes, the init container approach is the most appealing because we can use ConfigMaps directly for the template parameters. This technique is illustrated in Figure 22-1.

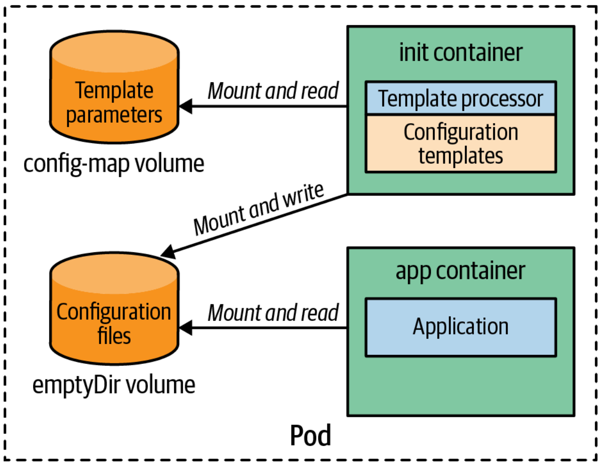

Figure 22-1. Configuration template

The application's Pod definition consists of at least two containers: one init container for the template processing and one for the application container. The init container contains not only the template processor but also the configuration templates themselves. In addition to the containers, this Pod also defines two volumes: one volume for the template parameters, backed by a ConfigMap, and an `emptyDir` volume used to share the processed templates between the init container and the application container.

With this setup, the following steps are performed during startup of this Pod:

1. The init container is started, and it runs the template processor. The processor takes the templates from its image, and the template parameters from the mounted ConfigMap volume, and stores the result in the `emptyDir` volume.
2. After the init container has finished, the application container starts up and loads the configuration files from the `emptyDir` volume.

The following example uses an init container for managing a full set of WildFly configuration files for two environments: a development environment and a production environment. Both are very similar to each other and differ only slightly. In fact, in our example, they differ only in the way logging is performed: each log line is prefixed with `DEVELOPMENT:` or `PRODUCTION:`, respectively.

You can find the full example along with complete installation instructions in the book's example GitHub repo. (We show only the main concept here; for the technical details, refer to the source repo.)

The log pattern in Example 22-1 is stored in *standalone.xml*, which we parameterize by using the Go template syntax.

Example 22-1. Log configuration template

```
....
<formatter name="COLOR-PATTERN">
  <pattern-formatter pattern="{{(datasource "config").logFormat}}"/>
</formatter>
....
```

Here we use Gomplate as a template processor, which uses the notion of a *data source* for referencing the template parameters to be filled in. In our case, this data source comes from a ConfigMap-backed volume mounted to an init container. Here, the ConfigMap contains a single entry with the key `logFormat`, from where the actual format is extracted.

With this template in place, we can now create the Docker image for the init container. The Dockerfile for the image *k8spatterns/example-configuration-template-init* is very simple (Example 22-2).

Example 22-2. Simple Dockerfile for template image

```
FROM k8spatterns/gomplate
COPY in /in
```

The base image *k8spatterns/gomplate* contains the template processor and an entrypoint script that uses the following directories by default:

- */in* holds the WildFly configuration templates, including the parameterized *standalone.xml*. These are added directly to the image.
- */params* is used to look up the Gomplate data sources, which are YAML files. This directory is mounted from a ConfigMap-backed Pod volume.
- */out* is the directory into which the processed files are stored. This directory is mounted in the WildFly application container and used for the configuration.

The second ingredient of our example is the ConfigMap holding the parameters. In Example 22-3, we just use a simple file with key-value pairs.

Example 22-3. Create ConfigMap with values to fill into the configuration template

```
kubectl create configmap wildfly-cm \
       --from-literal='config.yml=logFormat: "DEVELOPMENT: %-5p %s%e%n'
```

Finally, we need the Deployment resource for the WildFly server (Example 22-4).

Example 22-4. Deployment with template processor as init container

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    example: cm-template
  name: wildfly-cm-template
spec:
  replicas: 1
  template:
    metadata:
      labels:
        example: cm-template
    spec:
      initContainers:
      - image: k8spatterns/example-config-cm-template-init ❶
        name: init
        volumeMounts:
        - mountPath: "/params"                              ❷
          name: wildfly-parameters
        - mountPath: "/out"                                 ❸
          name: wildfly-config
      containers:
      - image: jboss/wildfly:10.1.0.Final
        name: server
        command:
        - "/opt/jboss/wildfly/bin/standalone.sh"
        - "-Djboss.server.config.dir=/config"
        volumeMounts:
        - mountPath: "/config"                              ❹
          name: wildfly-config
      volumes:                                              ❺
      - name: wildfly-parameters
        configMap:
          name: wildfly-cm
      - name: wildfly-config
        emptyDir: {}
```

❶ Image holding the configuration templates that has been created from Example 22-2.

❷ Parameters are mounted from a volume `wildfly-parameters` declared in ❺.

❸ The target directory for writing out processed templates. This is mounted from an empty volume.

❹ The directory holding the generated full configuration files is mounted as `/config`.

❺ Volume declaration for the parameters' ConfigMap and the empty directory used for sharing the processed configuration.

This declaration is quite a mouthful, so let's drill down: the Deployment specification contains a Pod with our init container, the application container, and two internal Pod volumes:

- The first volume, `wildfly-parameters`, references the ConfigMap `wildfly-cm` with the parameter values that we created in Example 22-3.
- The other volume is an empty directory initially and is shared between the init container and the WildFly container.

If you start this Deployment, the following will happen:

- An init container is created, and its command is executed. This container takes the *config.yml* from the ConfigMap volume, fills in the templates from the */in* directory in an init container, and stores the processed files in the */out* directory. The */out* directory is where the volume `wildfly-config` is mounted.
- After the init container is done, a WildFly server starts with an option so that it looks up the complete configuration from the */config* directory. Again, */config* is the shared volume `wildfly-config` containing the processed template files.

It is important to note that we do *not* have to change these Deployment resource descriptors when going from the development to the production environment. Only the ConfigMap with the template parameters is different.

With this technique, it is easy to create a DRY configuration without copying and maintaining duplicated large configuration files.[1] For example, when the WildFly configuration changes for all environments, only a single template file in the init container needs to be updated. This approach has, of course, significant advantages on maintenance as there is no danger of configuration drift.

When working with Pods and volumes, as in this pattern, it is not obvious how to debug if things don't work as expected. So if you want to examine the processed templates, check out the directory */var/lib/kubelet/pods/{podid}/volumes/kubernetes.io~empty-dir/* on the node, as it contains the content of an `emptyDir` volume. Alternatively, just `kubectl exec` into the Pod when it is running, and examine the mounted directory (*/config* in our example) for any created files.

Discussion

The *Configuration Template* pattern builds on top of the *Configuration Resource* pattern and is especially suited when we need to operate applications in different environments with similar complex configurations. However, the setup with configuration templates is more complicated and has more moving parts that can go wrong. Use it only if your application requires huge configuration data. Such applications often require a considerable amount of configuration data from which only a small fraction is dependent on the environment. Even when copying over the whole configuration directly into the environment-specific ConfigMap works initially, it puts a

1 DRY is an acronym for "Don't Repeat Yourself."

burden on the maintenance of that configuration because it is doomed to diverge over time. For such a situation, this template approach is perfect.

If you are running on top of Red Hat OpenShift, an enterprise Kubernetes distribution, you have an alternative by using OpenShift templates for parameterizing resource descriptors. This approach does not solve the challenge of large configuration sets but is still very helpful for applying the same deployment resources to slightly varying environments.

More Information

- Configuration Template Example
- Tiller Template Engine
- Gomplate
- Go Template Syntax

PART V

Security Patterns

Security is a broad topic that has implications for all stages of the software development lifecycle, from development practices, to image scanning at build time, to cluster hardening through admission controllers at deployment time, to threat detection at runtime. Security also touches all the layers of the software stack, from cloud infrastructure security, to cluster security, to container security, to code security, also known as the 4C's of cloud native security. In this section, we focus on the intersection of an application with Kubernetes from the security point of view, as demonstrated in Figure V-1.

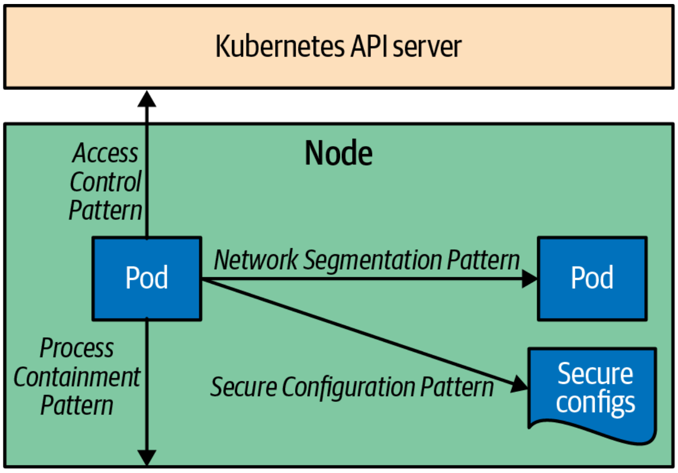
Kubernetes API server
Node
Access Control Pattern
Pod
Network Segmentation Pattern
Pod
Process Containment Pattern
Secure Configuration Pattern
Secure configs

Figure V-1. Security patterns

We start by describing the *Process Containment* pattern to contain and limit the actions an application is allowed to perform on the node it is running on. Then we explore the techniques to limit what other Pods a Pod can talk to by doing *Network Segmentation*. In the *Secure Configuration* pattern, we discuss how an application within a Pod can access and use configurations in a secure way. And finally, we describe the *Access Control* pattern—how an application can authenticate and talk to the Kubernetes API server and interact with it in more advanced scenarios. These give you an overview of the main security dimensions of an application running on Kubernetes, and we discuss the resulting patterns in the following chapters:

- Chapter 23, "Process Containment", describes the ways to contain a process to the least privileges it is entitled to.
- Chapter 24, "Network Segmentation", applies network controls to limit the traffic a Pod is allowed to participate in.
- Chapter 25, "Secure Configuration", helps keep and use sensitive configuration data securely and safely.
- Chapter 26, "Access Control", allows users and application workloads to authenticate and interact with the Kubernetes API server.

CHAPTER 23

Process Containment

This chapter describes techniques that help apply the principle of least privilege to constrain a process to the minimum privileges it needs to run. The *Process Containment* pattern helps make applications more secure by limiting the attack surface and creating a line of defense. It also prevents any rogue process from running out of its designated boundary.

Problem

One of the primary attack vectors for Kubernetes workloads is through the application code. Many techniques can help improve code security. For example, static code analysis tools can check the source code for security flaws. Dynamic scanning tools can simulate malicious attackers with the goal of breaking into the system through well-known service attacks such as SQL injection (SQLi), cross-site request forgery (CSRF), and cross-site scripting (XSS). Then there are tools for regularly scanning the application's dependencies for security vulnerabilities. As part of the image build process, the containers are scanned for known vulnerabilities. This is usually done by checking the base image and all its packages against a database that tracks vulnerable packages. These are only a few of the steps involved in creating secure applications and protecting against malicious actors, compromised users, unsafe container images, or dependencies with vulnerabilities.

Regardless of how many checks are in place, new code and new dependencies can introduce new vulnerabilities, and there is no way to guarantee the complete absence of risks. Without runtime process-level security controls in place, a malicious actor can breach the application code and attempt to take control of the host or the entire Kubernetes cluster. The mechanisms we will explore in this chapter demonstrate how to limit a container only to the permissions it needs to run and apply the least-privilege principle. This way, Kubernetes configurations act as another line of defense, containing any rogue process and preventing it from running outside its designated boundary.

Solution

Typically, a container runtime such as Docker assigns the default runtime permissions a container will have. When the container is managed by Kubernetes, the security configurations that will be applied to a container are controlled by Kubernetes and exposed to the user through the security context configurations of the Pod and the container specs. The Pod-level configurations apply to the Pod's volumes and all containers in the Pod, whereas container-level configurations apply to a single container. When the same configurations are set at both Pod and container levels, the values in the container spec take precedence.

As a developer creating cloud native applications, you typically should not need to deal with many fine-grained security configurations but instead have them validated and enforced as global policy. Fine-grained tuning is usually required when creating specialized infrastructure containers such as build systems and other plugins that need broader access to the underlying nodes. Therefore, we will review only the common security configurations that would be useful for running typical cloud native applications on Kubernetes.

Running Containers with a Non-Root User

Container images have a user, and can optionally have a group, to run the container process. These users and groups are used to control access to files, directories, and volume mounts. With some other containers, no user is created and the container image runs as root by default. In others, a user is created in the container image, but it is not set as the default user to run. These situations can be rectified by overriding the user at runtime using `securityContext`, as shown in Example 23-1.

Example 23-1. Setting a user and group for the containers of a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  securityContext:
    runAsUser: 1000    ❶
    runAsGroup: 2000   ❷
  containers:
  - name: app
    image: k8spatterns/random-generator:1.0
```

❶ Indicates the UID to run the container process.

❷ Specifies the GID to run the container process.

The configuration forces any container in the Pod to run with user ID 1000 and group ID 2000. This is useful when you want to swap the user that is specified in the container image. But there is also a danger in setting these values and making runtime decisions about which user to run the image. Often the user is set in conjunction with the directory structure containing files that have the same ownership IDs specified in the container image. To avoid having runtime failures due to lack of permissions, you should check the container image file and run the container with the user ID and group ID defined. This is one way to prevent a container from running as root, and matching it to the expected user in the image.

Instead of specifying a user ID to ensure that a container is not running as root, a less intrusive way is to set the `.spec.securityContext.runAsNonRoot` flag to `true`. When set, the Kubelet will validate at runtime and prevent any container from starting with a root user—that is, a user with UID 0. This latter mechanism doesn't change the user, but only ensures that a container is running as a non-root user. If you need to run as root to access files or volumes in the container, you can limit the exposure to root by running an init container that can run as root for a short time, and you can change the file access modes, before applications containers start up as non-root.

A container may not run as root, but it is possible to obtain root-like capabilities through privilege escalation. This is most similar to using the `sudo` command on Linux and executing commands with the root privileges. The way to prevent this in containers is by setting `.spec.containers[].securityContext.allowPrivilegeEscalation` to `false`. This configuration typically has no side effects because if an application is designed to run as non-root, it should not require privilege escalation during its lifetime.

The root user has special permissions and privileges in a Linux system, and preventing the root user from owning container processes, escalating privileges to become root, or limiting the root user lifetime with init containers will help prevent container breakout attacks and ensure adherence to the general security practices.

Restricting Container Capabilities

In essence, a container is a process that runs on a node, and it can have the same privileges a process can have. If the process requires a kernel-level call, it needs to have the privileges to do so in order to succeed. You can do this either by running the container as root, which grants all privileges to the container, or by assigning specific capabilities required for the application to function.

Containers with the .spec.containers[].securityContext.privileged flag set are essentially equivalent to root on the host and bypass the kernel permission checks. From a security point of view, this option bundles your container with the host system rather than isolating it. Therefore, this flag is typically set for containers with administrative capabilities—for example, to manipulate the network stack or access hardware devices. It is a better approach to avoid using privileged containers altogether and give specific kernel capabilities to containers that need them. In Linux, the privileges traditionally associated with the root user are divided into distinct capabilities, which can be independently enabled and disabled. Finding out what capabilities your container has is not straightforward. You can employ a whitelisting approach and start your container without any capabilities and gradually add capabilities when needed for every use case within the container. You might need the help of your security team, or you can use tools such as SELinux in permissive mode and check the audit logs of your application to discover what capabilities it needs, if any.

To make containers more secure, you should provide them with the least amount of privileges needed to run. The container runtime assigns a set of default privileges (capabilities) to the container. Contrary to what you might expect, if the .spec.containers[].securityContext.capabilities section is left empty, the default set of capabilities defined by the container runtime are far more generous than most processes need, opening them up to exploits. A good security practice for locking down the container attack surface is to drop all privileges and add only the ones you need, as shown in Example 23-2.

Example 23-2. Setting Pod permissions

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: docker.io/centos/httpd
    securityContext:
      capabilities:
        drop: [ 'ALL' ]          ❶
        add: ['NET_BIND_SERVICE'] ❷
```

❶ Removes all default capabilities assigned to the container by the container runtime.

❷ Adds back only the NET_BIND_SERVICE capability.

In this example, we drop all the capabilities and add back only the NET_BIND_SERVICE capability, which allows binding to privileged ports with numbers lower than 1024. An alternative approach for addressing this scenario is to replace the container with one that binds to an unprivileged port number.

A Pod is more likely to be compromised if its Security Context is not configured or is too permissive. Limiting the capabilities of containers to the very minimum acts as an additional line of defense against known attacks. A malicious actor who breaches an application would have a harder time taking control of the host when the container process is not privileged or when the capabilities are severely limited.

Avoiding a Mutable Container Filesystem

In general, containerized applications should not be able to write to the container filesystem because containers are ephemeral and any state will be lost upon restart. As discussed in Chapter 11, "Stateless Service", state should be written to external persistence methods such as database or filesystems. Logs should be written to stdout or forward to a remote log collector. Such an application can limit the attack surface of the container further by having a read-only container filesystem. A read-only filesystem will prevent any rogue user from tampering with the application configuration or installing additional executables on the disk that can be used for further exploits. The way to do that is to set `.spec.containers[].securityContext.readOnlyRootFile` to `true`, which will mount the container's root filesystem as read-only. This prevents any writes to the container's root filesystem at runtime and enforces the principle of immutable infrastructure.

The complete list of values in the `securityContext` field has many more items and can vary between Pod and container configurations. It is beyond the scope of this book to cover all security configurations. The two other must-check security context options are `seccompProfile` and `seLinuxOptions`. The first one is a Linux kernel feature that can be used to limit the process running in a container to call only a subset of the available system calls. These system calls are configured as profiles and applied to a container or Pod.

The latter option, `seLinuxOptions`, can assign custom SELinux labels to all containers within the Pod as well as the volume. SELinux uses policies to define which processes can access other labeled objects in the system. In Kubernetes, it is typically used to label the container image in such a way as to restrict the process to access only files within the image. When SELinux is supported on the host environment, it can be strictly enforced to deny access, or it can be configured in permissive mode to log access violations.