can access the same volume to share data. We use the same mechanism to share the cloned files from the init container to the application container.

Example 15-1 shows an init container that copies data into an empty volume.

Example 15-1. Init Container

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
  - name: download
    image: bitnami/git
    command:                                    ❶
    - git
    - clone
    - https://github.com/mdn/beginner-html-site-scripted
    - /var/lib/data
    volumeMounts:                               ❷
    - mountPath: /var/lib/data
      name: source
  containers:
  - name: run
    image: centos/httpd
    ports:
    - containerPort: 80
    volumeMounts:                               ❷
    - mountPath: /var/www/html
      name: source
  volumes:                                      ❸
  - emptyDir: {}
    name: source
```

❶ Clone an external Git repository into the mounted directory.

❷ Shared volume used by both init container and the application container.

❸ Empty directory used on the node for sharing data.

We could have achieved the same effect by using ConfigMap or PersistentVolumes but want to demonstrate how init containers work here. This example illustrates a typical usage pattern of an init container sharing a volume with the main container.

For debugging the outcome of init containers, it helps if the command of the application container is replaced temporarily with a dummy `sleep` command so that you have time to examine the situation. This trick is particularly useful if your init container fails to start up and your application fails to start because the configuration is missing or broken. The following command within the Pod declaration gives you an hour to debug the volumes mounted by entering the Pod with `kubectl exec -it <pod> sh`:

```
command:
- /bin/sh
- "-c"
- "sleep 3600"
```

A similar effect can be achieved by using a sidecar, as described next in Chapter 16, “Sidecar”, where the HTTP server container and the Git container are running side by side as application containers. But with the sidecar approach, there is no way of knowing which container will run first, and sidecar is meant to be used when containers run side by side continuously. We could also use a sidecar and init container together if both a guaranteed initialization and a constant update of the data are required.

More Initialization Techniques

As you have seen, an init container is a Pod-level construct that gets activated after a Pod has been started. A few other related techniques used to initialize Kubernetes resources are different from init containers and are worth listing here for completeness:

Admission controllers
: This set of plugins intercepts every request to the Kubernetes API Server before persistence of the object and can mutate or validate it. There are many admission controllers for applying checks, enforcing limits, and setting default values, but all are compiled into the `kube-apiserver` binary and configured by a cluster administrator when the API Server starts up. This plugin system is not very flexible, which is why admission webhooks were added to Kubernetes.

Admission webhooks
: These components are external admission controllers that perform HTTP callbacks for any matching request. There are two types of admission webhooks: the *mutating webhook* (which can change resources to enforce custom defaults) and the *validating webhook* (which can reject resources to enforce custom admission policies). This concept of external controllers allows admission webhooks to be developed out of Kubernetes and configured at runtime.

There used to be other techniques for initializing Kubernetes resources, such as Initializers and PodPresets, which were eventually deprecated and removed. Nowadays other projects such as Metacontroller and Kyverno use admission webhooks or the *Operator* pattern to mutate Kubernetes resources and intervene in the initialization process. These techniques differ from init containers because they validate and mutate resources at creation time.

In contrast, the *Init Container* pattern discussed in this chapter is something that activates and performs its responsibilities during startup of the Pod. You could use admission webhooks, for example, to inject an init container into any Pod that doesn't have one already. For example, Istio, which is a popular service mesh project, uses a combination of techniques discussed in this chapter to inject its proxies into application Pods. Istio uses Kubernetes mutating admission webhooks for automatic sidecar and init container injection into the Pod definition at Pod definition creation time. When such a Pod is starting up, Istio's init container configures the Pod environment to redirect inbound and outbound traffic from the application to the Envoy proxy sidecar. The init container runs before any other container and configures iptable rules to insert the Envoy proxy in the request path of the application before any traffic reaches the application. This separation of containers is good for lifecycle management and also because the init container in this case requires elevated permissions to configure traffic redirection, which can pose a security threat. This is an example of how many initialization activities can be performed before an application container starts up.

In the end, the most significant difference is that init containers can be used by developers deploying on Kubernetes, whereas admission webhooks help administrators and various frameworks control and alter the container initialization process.

Discussion

So why separate containers in a Pod into two groups? Why not just use an application container with a bit of scripting in a Pod for initialization if required? The answer is that these two groups of containers have different lifecycles, purposes, and even authors in some cases.

Having init containers run before application containers, and more importantly, having init containers run in stages that progress only when the current init container completes successfully, means you can be sure at every step of the initialization that the previous step has completed successfully, and you can progress to the next stage. Application containers, in contrast, run in parallel and do not provide similar guarantees as init containers. With this distinction in hand, we can create containers focused on initialization or application-focused tasks, and reuse them in different contexts by organizing them in Pods with predictable guarantees.

More Information

- Init Container Example
- Init Containers
- Configuring Pod Initialization
- Admission Controllers Reference
- Dynamic Admission Control
- Metacontroller
- Kyverno
- Demystifying Istio's Sidecar Injection Model
- Object Initialization in Swift

CHAPTER 16
Sidecar

A sidecar container extends and enhances the functionality of a preexisting container without changing it. The *Sidecar* pattern is one of the fundamental container patterns that allows single-purpose containers to cooperate closely together. In this chapter, you'll learn all about the basic sidecar concept. The specialized follow-up patterns, *Adapter* and *Ambassador*, are discussed in Chapters 17 and 18, respectively.

Problem

Containers are a popular packaging technology that allow developers and system administrators to build, ship, and run applications in a unified way. A container represents a natural boundary for a unit of functionality with a distinct runtime, release cycle, API, and team owning it. A proper container behaves like a single Linux process—solves one problem and does it well—and is created with the idea of replaceability and reuse. This last part is essential as it allows us to build applications more quickly by leveraging existing specialized containers.

Today, to make an HTTP call, we don't have to write a client library but can use an existing one. In the same way, to serve a website, we don't have to create a container for a web server but can use an existing one. This approach allows developers to avoid reinventing the wheel and create an ecosystem with a smaller number of better-quality containers to maintain. However, having single-purpose reusable containers requires ways of extending the functionality of a container and a means for collaboration among containers. The sidecar pattern describes this kind of collaboration, where a container enhances the functionality of another preexisting container.

Solution

In Chapter 1, we described how the Pod primitive allows us to combine multiple containers into a single unit. Behind the scenes, at runtime, a Pod is a container as well, but it starts as a paused process (literally with the `pause` command) before all other containers in the Pod. It is not doing anything other than holding all the Linux namespaces the application containers use to interact throughout the Pod's lifetime. Apart from this implementation detail, what is more interesting is all the characteristics that the Pod abstraction provides.

The Pod is such a fundamental primitive that it is present in many cloud native platforms under different names but always with similar capabilities. A Pod as the deployment unit puts certain runtime constraints on the containers belonging to it. For example, all containers end up deployed to the same node, and they share the same Pod lifecycle. In addition, a Pod allows its containers to share volumes and communicate over the local network or host IPC. These are the reasons users put a group of containers into a Pod. *Sidecar* (sometimes also called *Sidekick*) is used to describe the scenario of a container being put into a Pod to extend and enhance another container's behavior.

A typical example demonstrating this pattern is of an HTTP server and a Git synchronizer. The HTTP server container is focused only on serving files over HTTP and does not know how or where the files are coming from. Similarly, the Git synchronizer container's only goal is to sync data from a Git server to the local filesystem. It does not care what happens once synced—its only concern is keeping the local folder in sync with the remote Git server. Example 16-1 shows a Pod definition with these two containers configured to use a volume for file exchange.

Example 16-1. Pod with a sidecar

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: centos/httpd                  ❶
    volumeMounts:
    - mountPath: /var/www/html           ❸
      name: git
  - name: poll
    image: bitnami/git                   ❷
    volumeMounts:
    - mountPath: /var/lib/data           ❸
      name: git
    env:
```

```
    - name: GIT_REPO
      value: https://github.com/mdn/beginner-html-site-scripted
    command: [ "sh", "-c" ]
    args:
    - |
      git clone $(GIT_REPO) .
      while true; do
        sleep 60
        git pull
      done
    workingDir: /var/lib/data
  volumes:
  - emptyDir: {}
    name: git
```

❶ Main application container serving files over HTTP.

❷ Sidecar container running in parallel and pulling data from a Git server.

❸ Shared location for exchanging data between the sidecar and main application container as mounted in the `app` and `poll` containers, respectively.

This example shows how the Git synchronizer enhances the HTTP server's behavior with content to serve and keeps it synchronized. We could also say that both containers collaborate and are equally important, but in a *Sidecar* pattern, there is a main container and a helper container that enhance the collective behavior. Typically, the main container is the first one listed in the containers list, and it represents the default container (e.g., when we run the command `kubectl exec`).

This simple pattern, illustrated in Figure 16-1, allows runtime collaboration of containers and at the same time enables separation of concerns for both containers, which might be owned by separate teams, using different programming languages, with different release cycles, etc. It also promotes replaceability and reuse of containers as the HTTP server, and the Git synchronizer can be reused in other applications and different configuration either as a single container in a Pod or again in collaboration with other containers.

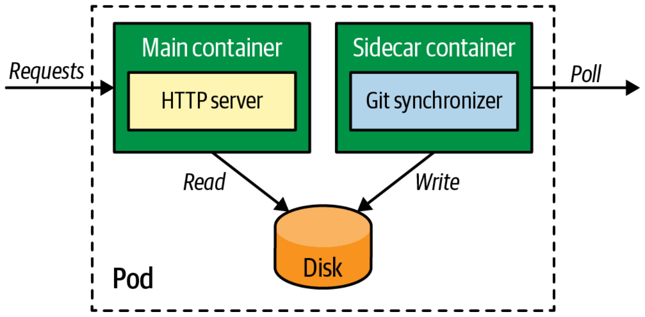

Figure 16-1. Sidecar pattern

Discussion

Previously we said that container images are like classes, and containers are like the objects in object-oriented programming (OOP). If we continue this analogy, extending a container to enhance its functionality is similar to inheritance in OOP, and having multiple containers collaborating in a Pod is similar to composition in OOP. While both approaches allow code reuse, inheritance involves tighter coupling between containers and represents an "is-a" relationship between containers.

On the other hand, a composition in a Pod represents a "has-a" relationship, and it is more flexible because it doesn't couple containers together at build time, giving you the ability to later swap containers in the Pod definition. With the composition approach, you have multiple containers (processes) running, health checked, restarted, and consuming resources, as the main application container does. Modern sidecar containers are small and consume minimal resources, but you have to decide whether it is worth running a separate process or whether it is better to merge it into the main container.

We see two dominating approaches for using sidecars: transparent sidecars that are invisible to the application, and explicit sidecars that the main application interacts with over well-defined APIs. Envoy proxy is an example of a transparent sidecar that runs alongside the main container and abstracts the network by providing common features such as Transport Layer Security (TLS), load balancing, automatic retries, circuit breaking, global rate limiting, observability of L7 traffic, distributed tracing, and more. All of these features become available to the application by transparently attaching the sidecar container and intercepting all the incoming and outgoing traffic to the main container. This is similar to aspect-oriented programming, in that with additional containers, we introduce orthogonal capabilities to the Pod without touching the main container.

An example of an explicit proxy that uses the sidecar architecture is Dapr. A Dapr sidecar container is injected into a Pod and offers features such as reliable service invocation, publish-subscribe, bindings to external systems, state abstraction, observability, distributed tracing, and more. The primary difference between Dapr and Envoy proxy is that Dapr does not intercept all the networking traffic going in and out of the application. Rather, Dapr features are exposed over HTTP and gRPC APIs, which the application invokes or subscribes to.

More Information

- Sidecar Example
- Pods
- Design Patterns for Container-Based Distributed Systems
- Prana: A Sidecar for Your Netflix PaaS-Based Applications and Services
- Tin-Can Phone: Patterns to Add Authorization and Encryption to Legacy Applications
- Envoy
- Dapr
- The Almighty Pause Container
- Sidecar Pattern

CHAPTER 17

Adapter

The *Adapter* pattern takes a heterogeneous containerized system and makes it conform to a consistent, unified interface with a standardized and normalized format that can be consumed by the outside world. The *Adapter* pattern inherits all its characteristics from the *Sidecar* pattern but has the single purpose of providing adapted access to the application.

Problem

Containers allow us to package and run applications written in different libraries and languages in a unified way. Today, it is common to see multiple teams using different technologies and creating distributed systems composed of heterogeneous components. This heterogeneity can cause difficulties when all components have to be treated in a unified way by other systems. The *Adapter* pattern offers a solution by hiding the complexity of a system and providing unified access to it.

Solution

The best way to illustrate the *Adapter* pattern is through an example. A major prerequisite for successfully running and supporting distributed systems is providing detailed monitoring and alerting. Moreover, if we have a distributed system composed of multiple services we want to monitor, we may use an external monitoring tool to poll metrics from every service and record them.

However, services written in different languages may not have the same capabilities and may not expose metrics in the same format expected by the monitoring tool. This diversity creates a challenge for monitoring such a heterogeneous application from a single monitoring solution that expects a unified view of the whole system. With the *Adapter* pattern, it is possible to provide a unified monitoring interface by exporting metrics from various application containers into one standard format and protocol. In Figure 17-1, an adapter container translates locally stored metrics information into the external format the monitoring server understands.

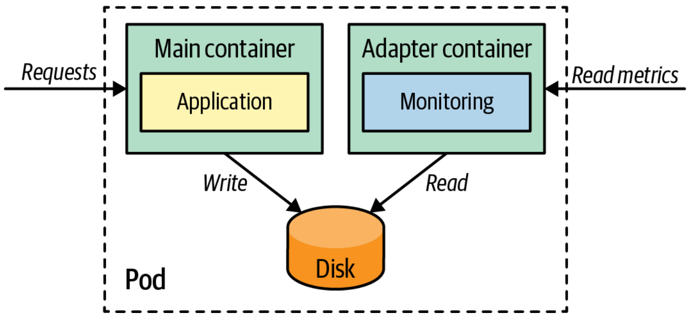

Figure 17-1. Adapter pattern

With this approach, every service represented by a Pod, in addition to the main application container, would have another container that knows how to read the custom application-specific metrics and expose them in a generic format understandable by the monitoring tool. We could have one adapter container that knows how to export Java-based metrics over HTTP and another adapter container in a different Pod that exposes Python-based metrics over HTTP. For the monitoring tool, all metrics would be available over HTTP and in a common, normalized format.

For a concrete implementation of this pattern, let's add the adapter shown in Figure 17-1 to our sample random generator application. When appropriately configured, it writes out a log file with the random-number generator and includes the time it took to create the random number. We want to monitor this time with Prometheus. Unfortunately, the log format doesn't match the format Prometheus expects. Also, we need to offer this information over an HTTP endpoint so that a Prometheus server can scrape the value.

For this use case, an adapter is a perfect fit: a sidecar container starts a small HTTP server and on every request, reads the custom log file and transforms it into a Prometheus-understandable format. Example 17-1 shows a Deployment with such an adapter. This configuration allows a decoupled Prometheus monitoring setup without the main application needing to know anything about Prometheus. The full example in the book's GitHub repository demonstrates this setup together with a Prometheus installation.

Example 17-1. Adapter delivering Prometheus-conformant output

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0         ❶
        name: random-generator
        env:
        - name: LOG_FILE                                 ❷
          value: /logs/random.log
        ports:
        - containerPort: 8080
          protocol: TCP
        volumeMounts:                                    ❸
        - mountPath: /logs
          name: log-volume
      # --------------------------------------------
      - image: k8spatterns/random-generator-exporter ❹
        name: prometheus-adapter
        env:
        - name: LOG_FILE                                 ❺
          value: /logs/random.log
        ports:
        - containerPort: 9889
          protocol: TCP
        volumeMounts:                                    ❻
        - mountPath: /logs
          name: log-volume
      volumes:
      - name: log-volume                                 ❼
        emptyDir: {}
```

❶ Main application container with the random generator service exposed on 8080.

❷ Path to the log file containing the timing information about random-number generation.

❸ Directory shared with the Prometheus Adapter container.

❹ Prometheus exporter image, exporting on port 9889.

❺ Path to the same log file to which the main application is logging.

❻ Shared volume is also mounted in the adapter container.

❼ Files are shared via an `emptyDir` volume from the node's filesystem.

Another use of this pattern is logging. Different containers may log information in different formats and levels of detail. An adapter can normalize that information, clean it up, enrich it with contextual information by using the *Self Awareness* pattern described in Chapter 14, and then make it available for pickup by the centralized log aggregator.

Discussion

The *Adapter* is a specialization of the *Sidecar* pattern explained in Chapter 16. It acts as a reverse proxy to a heterogeneous system by hiding its complexity behind a unified interface. Using a distinct name different from the generic *Sidecar* pattern allows us to more precisely communicate the purpose of this pattern.

In the next chapter, you'll get to know another sidecar variation: the *Ambassador* pattern, which acts as a proxy to the outside world.

More Information

- Adapter Example

CHAPTER 18

Ambassador

The *Ambassador* pattern is a specialized sidecar responsible for hiding external complexities and providing a unified interface for accessing services outside the Pod. In this chapter, you will see how the *Ambassador* pattern can act as a proxy and decouple the main container from directly accessing external dependencies.

Problem

Containerized services don't exist in isolation and very often have to access other services that may be difficult to reach in a reliable way. The difficulty in accessing other services may be due to dynamic and changing addresses, the need for load balancing of clustered service instances, an unreliable protocol, or difficult data formats. Ideally, containers should be single-purposed and reusable in different contexts. But if we have a container that provides some business functionality and consumes an external service in a specialized way, the container will have more than one responsibility.

Consuming the external service may require a special service discovery library that we do not want to put in our container. Or we may want to swap different kinds of services by using different kinds of service-discovery libraries and methods. This technique of abstracting and isolating the logic for accessing other services in the outside world is the goal of this *Ambassador* pattern.

Solution

To demonstrate the pattern, we will use a cache for an application. Accessing a local cache in the development environment may be a simple configuration, but in the production environment, we may need a client configuration that can connect to the different shards of the cache. Another example is consuming a service by looking it up in a registry and performing client-side service discovery. A third example is consuming a service over a nonreliable protocol such as HTTP, so to protect our application, we have to use circuit-breaker logic, configure timeouts, perform retries, and more.

In all of these cases, we can use an ambassador container that hides the complexity of accessing the external services and provides a simplified view and access to the main application container over localhost. Figures 18-1 and 18-2 show how an ambassador Pod can decouple access to a key-value store by connecting to an ambassador container listening on a local port. In Figure 18-1, we see how data access can be delegated to a fully distributed remote store like etcd.

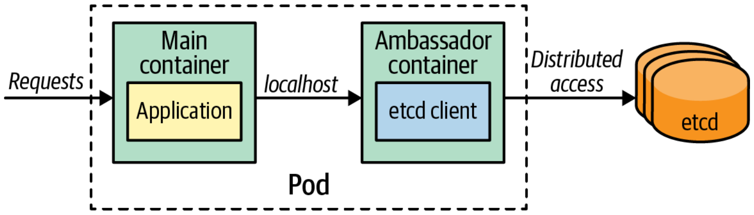

Figure 18-1. Ambassador for accessing a remote distributed cache

For development purposes, this ambassador container can be easily exchanged with a locally running in-memory key-value store like memcached (as shown in Figure 18-2).

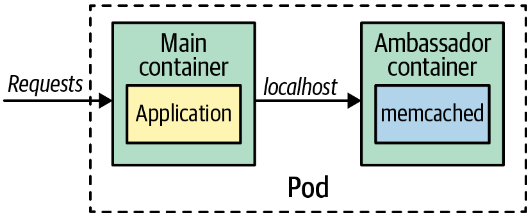

Figure 18-2. Ambassador for accessing a local cache

Example 18-1 shows an ambassador that runs parallel to a REST service. Before returning its response, the REST service logs the generated data by sending it to a fixed URL: *http://localhost:9009*. The ambassador process listens in on this port and processes the data. In this example, it prints the data out just to the console, but it could also do something more sophisticated like forward the data to a full logging infrastructure. For the REST service, it doesn't matter what happens to the log data, and you can easily exchange the ambassador by reconfiguring the Pod without touching the main container.

Example 18-1. Ambassador processing log output

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0            ❶
    name: main
    env:
    - name: LOG_URL                                    ❷
      value: http://localhost:9009
    ports:
    - containerPort: 8080
      protocol: TCP
  - image: k8spatterns/random-generator-log-ambassador ❸
    name: ambassador
```

❶ Main application container providing a REST service for generating random numbers.

❷ Connection URL for communicating with the ambassador via localhost.

❸ Ambassador running in parallel and listening on port 9009 (which is not exposed to the outside of the Pod).

Discussion

At a higher level, the *Ambassador* pattern is a *Sidecar* pattern. The main difference between ambassador and sidecar is that an ambassador does not enhance the main application with additional capability. Instead, it acts merely as a smart proxy to the outside world (this pattern is sometimes referred to as the *Proxy* pattern). This pattern can be useful for legacy applications that are difficult to modify and extend with modern networking concepts such as monitoring, logging, routing, and resiliency patterns.

The benefits of the *Ambassador* pattern are similar to those of the *Sidecar* pattern—both allow you to keep containers single-purposed and reusable. With such a pattern, our application container can focus on its business logic and delegate the responsibility and specifics of consuming the external service to another specialized container. This also allows you to create specialized and reusable ambassador containers that can be combined with other application containers.

More Information

- Ambassador Example
- How to Use the Ambassador Pattern to Dynamically Configure Services on CoreOS
- Modifications to the CoreOS Ambassador Pattern

PART IV

Configuration Patterns

Every application needs to be configured, and the easiest way to do this is by storing configurations in the source code. However, this approach has the side effect of code and configuration living and dying together. We need the flexibility to adapt configurations without modifying the application and recreating its container image. In fact, mixing code and configuration is an antipattern for a continuous delivery approach, where the application is created once and then moves unaltered through the various stages of the deployment pipeline until it reaches production. The way to achieve this separation of code and configuration is by using external configuration data, which is different for each environment. The patterns in the following chapters are all about customizing and adapting applications with external configurations for various environments:

- Chapter 19, "EnvVar Configuration", uses environment variables to store configuration data.
- Chapter 20, "Configuration Resource", uses Kubernetes resources like ConfigMaps or Secrets to store configuration information.
- Chapter 21, "Immutable Configuration", brings immutability to large configuration sets by putting them into containers linked to the application at runtime.
- Chapter 22, "Configuration Template", is useful when large configuration files need to be managed for multiple environments that differ only slightly.

CHAPTER 19

EnvVar Configuration

In this *EnvVar Configuration* pattern, we look into the simplest way to configure applications. For small sets of configuration values, the easiest way to externalize configuration is by putting them into universally supported environment variables. We'll see different ways of declaring environment variables in Kubernetes but also the limitations of using environment variables for complex configurations.

Problem

Every nontrivial application needs some configuration for accessing data sources, external services, or production-level tuning. And we knew well before the twelve-factor app manifesto that it is a bad thing to hardcode configurations within the application. Instead, the configuration should be *externalized* so that we can change it even after the application has been built. That provides even more value for containerized applications that enable and promote sharing of immutable application artifacts. But how can this be done best in a containerized world?

Solution

The twelve-factor app manifesto recommends using environment variables for storing application configurations. This approach is simple and works for any environment and platform. Every operating system knows how to define environment variables and how to propagate them to applications, and every programming language also allows easy access to these environment variables. It is fair to claim that environment variables are universally applicable. When using environment variables, a typical usage pattern is to define hardcoded default values during build time, which we can then overwrite at runtime. Let's see some concrete examples of how this works in Docker and Kubernetes.

For Docker images, environment variables can be defined directly in Dockerfiles with the ENV directive. You can define them line by line or all in a single line, as shown in Example 19-1.

Example 19-1. Example Dockerfile with environment variables

```
FROM openjdk:11
ENV PATTERN "EnvVar Configuration"
ENV LOG_FILE "/tmp/random.log"
ENV SEED "1349093094"

# Alternatively:
ENV PATTERN="EnvVar Configuration" LOG_FILE=/tmp/random.log SEED=1349093094
...
```

Then a Java application running in such a container can easily access the variables with a call to the Java standard library, as shown in Example 19-2.

Example 19-2. Reading environment variables in Java

```
public Random initRandom() {
  long seed = Long.parseLong(System.getenv("SEED"));
  return new Random(seed);       ❶
}
```

❶ Initializes a random-number generator with a seed from an EnvVar.

Directly running such an image will use the default hardcoded values. But in most cases, you want to override these parameters from outside the image.

When running such an image directly with Docker, environment variables can be set from the command line by calling Docker, as in Example 19-3.

Example 19-3. Set environment variables when starting a Docker container

```
docker run -e PATTERN="EnvVarConfiguration" \
           -e LOG_FILE="/tmp/random.log" \
           -e SEED="147110834325" \
           k8spatterns/random-generator:1.0
```

For Kubernetes, these types of environment variables can be set directly in the Pod specification of a controller like Deployment or ReplicaSet (as shown in Example 19-4).

Example 19-4. Deployment with environment variables set

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: LOG_FILE
      value: /tmp/random.log                    ❶
    - name: PATTERN
      valueFrom:
        configMapKeyRef:                        ❷
          name: random-generator-config         ❸
          key: pattern                          ❹
    - name: SEED
      valueFrom:
        secretKeyRef:                           ❺
          name: random-generator-secret
          key: seed
```

❶ EnvVar with a literal value.

❷ EnvVar from a ConfigMap.

❸ ConfigMap's name.

❹ Key within the ConfigMap to look for the EnvVar value.

❺ EnvVar from a Secret (lookup semantic is the same as for a ConfigMap).

In such a Pod template, you not only can attach values directly to environment variables (as for `LOG_FILE`), but also can use a delegation to Kubernetes Secrets and ConfigMaps. The advantage of ConfigMap and Secret indirection is that the environment variables can be managed independently from the Pod definition. Secret and ConfigMap and their pros and cons are explained in detail in Chapter 20, "Configuration Resource".

In the preceding example, the `SEED` variable comes from a Secret resource. While that is a perfectly valid use of Secret, it is also important to point out that environment variables are not secure. Putting sensitive, readable information into environment variables makes this information easy to read, and it may even leak into logs.

About Default Values

Default values make life easier, as they take away the burden of selecting a value for a configuration parameter you might not even know exists. They also play a significant role in the *convention over configuration* paradigm. However, defaults are not always a good idea. Sometimes they might even be an antipattern for an evolving application.

This is because *changing* default values retrospectively is a difficult task. First, changing default values means replacing them within the code, which requires a rebuild. Second, people relying on defaults (either by convention or consciously) will always be surprised when a default value changes. We have to communicate the change, and the user of such an application probably has to modify the calling code as well.

Changes in default values, however, often make sense, because it is hard to get default values right from the very beginning. It's essential that we consider a change in a default value as a *major change*, and if semantic versioning is in use, such a modification justifies a bump in the major version number. If unsatisfied with a given default value, it is often better to remove the default altogether and throw an error if the user does not provide a configuration value. This will at least break the application early and prominently instead of it doing something different and unexpected silently.

Considering all these issues, it is often the best solution to *avoid default values* from the very beginning if you cannot be 90% sure that a reasonable default will last for a long time. Passwords or database connection parameters are good candidates for not providing default values, as they depend highly on the environment and often cannot be reliably predicted. Also, if we do not use default values, the configuration information has to be provided explicitly, which serves as documentation too.

Instead of individually referring to configuration values from Secrets or ConfigMaps, you can also import *all* values of a particular Secret or ConfigMap with `envFrom`. We explain this field in Chapter 20, "Configuration Resource", when we talk about ConfigMaps and Secrets in detail.

Two other valuable features that can be used with environment variables are the downward API and *dependent variables*. You learned all about the downward API in Chapter 14, "Self Awareness", so let's have a look at dependent variables in Example 19-5 that allow you to reference previously defined variables in the value definition of other entries.

Example 19-5. Dependent environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: PORT
      value: "8181"
    - name: IP                                       ❶
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MY_URL
      value: "https://$(IP):$(PORT)"                 ❷
```

❶ Use the downward API to pick up the Pod's IP. The downward API is discussed in detail in Chapter 14, "Self Awareness".

❷ Include the previously defined environment variables `IP` and `PORT` to build up a URL.

With a `$(...)` notation, you can reference environment variables defined earlier in the `env` list or coming from an `envFrom` import. Kubernetes will resolve those references during the startup of the container. Be careful about the ordering, though: if you reference a variable defined later in the list, it will not be resolved, and the `$(...)` reference will be taken over literally. In addition, you can also reference environment variables with this syntax for Pod commands, as shown in Example 19-6.

Example 19-6. Using environment variables in a container's command definition

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - name: random-generator
      image: k8spatterns/random-generator:1.0
      command: [ "java", "RandomRunner", "$(OUTPUT_FILE)", "$(COUNT)" ] ❶
      env:                              ❷
        - name: OUTPUT_FILE
          value: "/numbers.txt"
        - name: COUNT
          valueFrom:
            configMapKeyRef:
              name: random-config
              key: RANDOM_COUNT
```

❶ Reference environment variables for the startup command of a container.

❷ Definition of the environment variables substituted in the commands.

Discussion

Environment variables are easy to use, and everybody knows about them. This concept maps smoothly to containers, and every runtime platform supports environment variables. But environment variables are not secure, and are good only for a decent number of configuration values. And when there are a lot of different parameters to configure, the management of all these environment variables becomes unwieldy.

In these cases, many people use an extra level of indirection and put configuration into various configuration files, one for each environment. Then a single environment variable is used to select one of these files. *Profiles* from Spring Boot are an example of this approach. Since these profile configuration files are typically stored within the application itself, which is within the container, it couples the configuration tightly with the application. This often leads to configuration for development and production ending up side by side in the same Docker image, which requires an image rebuild for every change in either environment. We do not recommend this setup (configuration should always be external to the application), but this solution indicates that environment variables are suitable for small to medium sets of configurations only.

The patterns *Configuration Resource*, *Immutable Configuration*, and *Configuration Template* described in the following chapters are good alternatives when more complex configuration needs come up.

Environment variables are universally applicable, and because of that, we can set them at various levels. This option leads to fragmentation of the configuration definitions and makes it hard to track for a given environment variable where it is set. When there is no central place where all environments variables are defined, it is hard to debug configuration issues.

Another disadvantage of environment variables is that they can be set only *before* an application starts, and we cannot change them later. On the one hand, it's a drawback that you can't change configuration "hot" during runtime to tune the application. However, many see this as an advantage, as it promotes *immutability* even to the configuration. Immutability here means you throw away the running application container and start a new copy with a modified configuration, very likely with a smooth Deployment strategy like rolling updates. That way, you are always in a defined and well-known configuration state.

Environment variables are simple to use, but are applicable mainly for simple use cases and have limitations for complex configuration requirements. The next patterns show how to overcome those limitations.

More Information

- EnvVar Configuration Example
- The Twelve-Factor App
- Expose Pod Information to Containers Through Environment Variables
- Define Dependent Environment Variables
- Spring Boot Profiles for Using Sets of Configuration Values

CHAPTER 20

Configuration Resource

Kubernetes provides native configuration resources for regular and confidential data, which allows you to decouple the configuration lifecycle from the application lifecycle. The *Configuration Resource* pattern explains the concepts of ConfigMap and Secret resources and how we can use them, as well as their limitations.

Problem

One significant disadvantage of the *EnvVar Configuration* pattern, discussed in Chapter 19, is that it's suitable for only a handful of variables and simple configurations. Another disadvantage is that because environment variables can be defined in various places, it is often hard to find the definition of a variable. And even if you find it, you can't be entirely sure it won't be overridden in another location. For example, environment variables defined within a OCI image can be replaced during runtime in a Kubernetes Deployment resource.

Often, it is better to keep all the configuration data in a single place and not scattered around in various resource definition files. But it does not make sense to put the content of a whole configuration file into an environment variable. So some extra indirection would allow more flexibility, which is what Kubernetes configuration resources offer.

Solution

Kubernetes provides dedicated configuration Resources that are more flexible than pure environment variables. These are the ConfigMap and Secret objects for general-purpose and sensitive data, respectively.