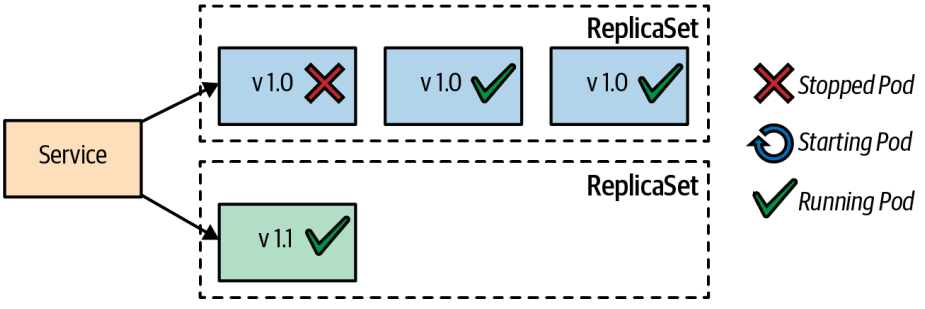

Figure 3-4. Canary release

Discussion

The Deployment primitive is an example of Kubernetes turning the tedious process of manually updating applications into a declarative activity that can be repeated and automated. The out-of-the-box deployment strategies (rolling and recreate) control the replacement of old containers by new ones, and the advanced release strategies (Blue-Green and canary) control how the new version becomes available to service consumers. The latter two release strategies are based on a human decision for the transition trigger and as a consequence are not fully automated by Kubernetes but require human interaction. Figure 3-5 summarizes of the deployment and release strategies, showing instance counts during transitions.

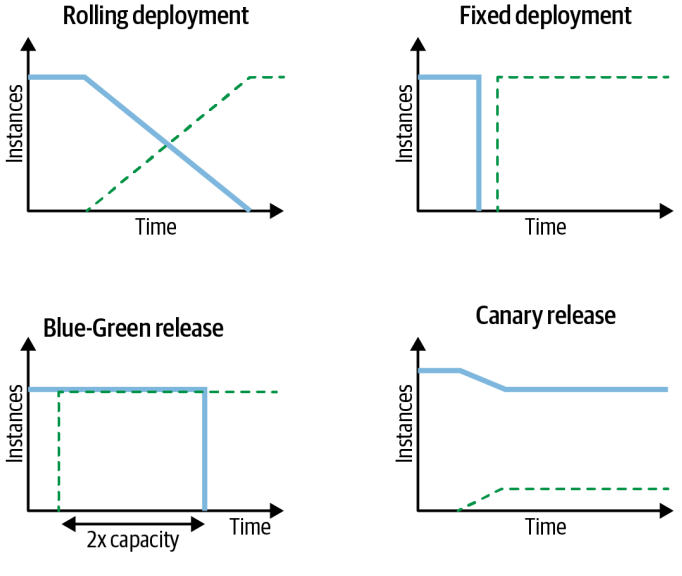

Figure 3-5. Deployment and release strategies

All software is different, and deploying complex systems usually requires additional steps and checks. The techniques discussed in this chapter cover the Pod update process, but do not include updating and rolling back other Pod dependencies such as ConfigMaps, Secrets, or other dependent services.

Pre and Post Deployment Hooks

In the past, there has been a proposal for Kubernetes to allow hooks in the deployment process. Pre and Post hooks would allow the execution of custom commands before and after Kubernetes has executed a deployment strategy. Such commands could perform additional actions while the deployment is in progress and would additionally be able to abort, retry, or continue a deployment. Those hooks are a good step toward new automated deployment and release strategies. Unfortunately, this effort has been stalled for some years (as of 2023), so it is unclear whether this feature will ever come to Kubernetes.

One approach that works today is to create a script to manage the update process of services and their dependencies using the Deployment and other primitives discussed in this book. However, this imperative approach that describes the individual update steps does not match the declarative nature of Kubernetes.

As an alternative, higher-level declarative approaches have emerged on top of Kubernetes. The most important platforms are described in the sidebar that follows. Those techniques work with operators (see Chapter 28, "Operator") that take a declarative description of the rollout process and perform the necessary actions on the server side, some of them also including automatic rollbacks in case of an update error. For advanced, production-ready rollout scenarios, it is recommended to look at one of those extensions.

Higher-Level Deployments

The Deployment resource is a good abstraction over ReplicaSets and Pods to allow a simple declarative rollout that a handful of parameters can tune. However, as we have seen, Deployment does not support more sophisticated strategies like canary or Blue-Green deployments directly. There are higher-level abstractions that enhance Kubernetes by introducing new resource types, enabling the declaration of more flexible deployment strategies. Those extensions all leverage the *Operator* pattern described in Chapter 28 and introduce their own custom resources for describing the desired rollout behavior.

As of 2023, the most prominent platforms that support higher-level Deployments include the following:

Flagger
: Flagger implements several deployment strategies and is part of the Flux CD GitOps tools. It supports canary and Blue-Green deployments and integrates with many ingress controllers and service meshes to provide the necessary traffic split between your app's old and new versions. It can also monitor the status of the rollout process based on a custom metric and detect if the rollout fails so that it can trigger an automatic rollback.

Argo Rollouts
: The focus on this part of the Argo family of tools is on providing a comprehensive and opinionated continuous delivery (CD) solution for Kubernetes. Argo Rollouts support advanced deployment strategies, like Flagger, and integrate into many ingress controllers and service meshes. It has very similar capabilities to Flagger, so the decision about which one to use should be based on which CD solution you prefer, Argo or Flux.

Knative
: Knative a serverless platform on top of Kubernetes. A core feature of Knative is traffic-driven autoscaling support, which is described in detail in Chapter 29, "Elastic Scale". Knative also provides a simplified deployment model and traffic splitting, which is very helpful for supporting high-level deployment rollouts. The support for rollout or rollbacks is not as advanced as with Flagger or Argo Rollouts but is still a substantial improvement over the rollout capabilities of Kubernetes Deployments. If you are using Knative anyway, the intuitive way of splitting traffic between two application versions is a good alternative to Deployments.

Like Kubernetes, all of these projects are part of the Cloud Native Computing Foundation (CNCF) project and have excellent community support.

Regardless of the deployment strategy you are using, it is essential for Kubernetes to know when your application Pods are up and running to perform the required sequence of steps to reach the defined target deployment state. The next pattern, *Health Probe*, in Chapter 4 describes how your application can communicate its health state to Kubernetes.

More Information

- Declarative Deployment Example
- Performing a Rolling Update
- Deployments
- Run a Stateless Application Using a Deployment
- Blue-Green Deployment

- Canary Release
- Flagger: Deployment Strategies
- Argo Rollouts
- Knative: Traffic Management

CHAPTER 4

Health Probe

The *Health Probe* pattern indicates how an application can communicate its health state to Kubernetes. To be fully automatable, a cloud native application must be highly observable by allowing its state to be inferred so that Kubernetes can detect whether the application is up and whether it is ready to serve requests. These observations influence the lifecycle management of Pods and the way traffic is routed to the application.

Problem

Kubernetes regularly checks the container process status and restarts it if issues are detected. However, from practice, we know that checking the process status is not sufficient to determine the health of an application. In many cases, an application hangs, but its process is still up and running. For example, a Java application may throw an `OutOfMemoryError` and still have the JVM process running. Alternatively, an application may freeze because it runs into an infinite loop, deadlock, or some thrashing (cache, heap, process). To detect these kinds of situations, Kubernetes needs a reliable way to check the health of applications—that is, not to understand how an application works internally, but to check whether the application is functioning as expected and capable of serving consumers.

Solution

The software industry has accepted the fact that it is not possible to write bug-free code. Moreover, the chances for failure increase even more when working with distributed applications. As a result, the focus for dealing with failures has shifted from avoiding them to detecting faults and recovering. Detecting failure is not a simple task that can be performed uniformly for all applications, as everyone has different

definitions of a failure. Also, various types of failures require different corrective actions. Transient failures may self-recover, given enough time, and some other failures may need a restart of the application. Let's look at the checks Kubernetes uses to detect and correct failures.

Process Health Checks

A *process health check* is the simplest health check the Kubelet constantly performs on the container processes. If the container processes are not running, the container is restarted on the node to which the Pod is assigned. So even without any other health checks, the application becomes slightly more robust with this generic check. If your application is capable of detecting any kind of failure and shutting itself down, the process health check is all you need. However, for most cases, that is not enough, and other types of health checks are also necessary.

Liveness Probes

If your application runs into a deadlock, it is still considered healthy from the process health check's point of view. To detect this kind of issue and any other types of failure according to your application business logic, Kubernetes has *liveness probes*—regular checks performed by the Kubelet agent that asks your container to confirm it is still healthy. It is important to have the health check performed from the outside rather than in the application itself, as some failures may prevent the application watchdog from reporting its failure. Regarding corrective action, this health check is similar to a process health check, since if a failure is detected, the container is restarted. However, it offers more flexibility regarding which methods to use for checking the application health, as follows:

HTTP probe
: Performs an HTTP GET request to the container IP address and expects a successful HTTP response code between 200 and 399.

TCP Socket probe
: Assumes a successful TCP connection.

Exec probe
: Executes an arbitrary command in the container's user and kernel namespace and expects a successful exit code (0).

gRPC probe
: Leverages gRPC's intrinsic support for health checks.

In addition to the probe action, the health check behavior can be influenced with the following parameters:

`initialDelaySeconds`
: Specifies the number of seconds to wait until the first liveness probe is checked.

`periodSeconds`
: The interval in seconds between liveness probe checks.

`timeoutSeconds`
: The maximum time allowed for a probe check to return before it is considered to have failed.

`failureThreshold`
: Specifies how many times a probe check needs to fail in a row until the container is considered to be unhealthy and needs to be restarted.

An example HTTP-based liveness probe is shown in Example 4-1.

Example 4-1. Container with a liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-liveness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: DELAY_STARTUP
      value: "20"
    ports:
    - containerPort: 8080
      protocol: TCP
    livenessProbe:
      httpGet:                    ❶
        path: /actuator/health
        port: 8080
      initialDelaySeconds: 30     ❷
```

❶ HTTP probe to a health-check endpoint.

❷ Wait 30 seconds before doing the first liveness check to give the application some time to warm up.

Depending on the nature of your application, you can choose the method that is most suitable for you. It is up to your application to decide whether it considers itself healthy or not. However, keep in mind that the result of not passing a health check is that your container will restart. If restarting your container does not help, there is no benefit to having a failing health check as Kubernetes restarts your container without fixing the underlying issue.

Readiness Probes

Liveness checks help keep applications healthy by killing unhealthy containers and replacing them with new ones. But sometimes, when a container is not healthy, restarting it may not help. A typical example is a container that is still starting up and is not ready to handle any requests. Another example is an application that is still waiting for a dependency like a database to be available. Also, a container can be overloaded, increasing its latency, so you want it to shield itself from the additional load for a while and indicate that it is not ready until the load decreases.

For this kind of scenario, Kubernetes has *readiness probes*. The methods (HTTP, TCP, Exec, gRPC) and timing options for performing readiness checks are the same as for liveness checks, but the corrective action is different. Rather than restarting the container, a failed readiness probe causes the container to be removed from the service endpoint and not receive any new traffic. Readiness probes signal when a container is ready so that it has some time to warm up before getting hit with requests from the service. It is also useful for shielding the container from traffic at later stages, as readiness probes are performed regularly, similarly to liveness checks. Example 4-2 shows how a readiness probe can be implemented by probing the existence of a file the application creates when it is ready for operations.

Example 4-2. Container with readiness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    readinessProbe:
      exec:  ❶
        command: [ "stat", "/var/run/random-generator-ready" ]
```

❶ Check for the existence of a file the application creates to indicate it's ready to serve requests. `stat` returns an error if the file does not exist, letting the readiness check fail.

Again, it is up to your implementation of the health check to decide when your application is ready to do its job and when it should be left alone. While process health checks and liveness checks are intended to recover from the failure by restarting the container, the readiness check buys time for your application and expects it to recover by itself. Keep in mind that Kubernetes tries to prevent your container from receiving new requests (when it is shutting down, for example), regardless of whether the readiness check still passes after having received a SIGTERM signal.

Custom Pod Readiness Gates

Readiness probes work on a per-container level, and a Pod is considered ready to serve requests when all containers pass their readiness probes. In some situations, this is not good enough—for example, when an external load balancer like the AWS LoadBalancer needs to be reconfigured and ready too. In this case, the `readinessGates` field of a Pod's specification can be used to specify extra conditions that need to be met for the Pod to become ready. Example 4-3 shows a readiness gate that will introduce an additional condition, `k8spatterns.io/load-balancer-ready`, to the Pod's `status` sections.

Example 4-3. Readiness gate for indicating the status of an external load balancer

```
apiVersion: v1
kind: Pod
...
spec:
  readinessGates:
  - conditionType: "k8spatterns.io/load-balancer-ready"
...
status:
  conditions:
  - type: "k8spatterns.io/load-balancer-ready" ❶
    status: "False"
    ...
  - type: Ready                                ❷
    status: "False"
    ...
```

❶ New condition introduced by Kubernetes and set to `False` by default. It needs to be switched to `True` externally, e.g., by a controller, as desribed in Chapter 27, "Controller", when the load balancer is ready to serve.

❷ The Pod is "ready" when all containers' readiness probes are passing and the readiness gates' conditions are `True`; otherwise, as here, the Pod is marked as nonready.

> Pod readiness gates are an advanced feature that are not supposed to be used by the end user but by Kubernetes add-ons to introduce additional dependencies on the readiness of a Pod.

In many cases, liveness and readiness probes are performing the same checks. However, the presence of a readiness probe gives your container time to start up. Only by passing the readiness check is a Deployment considered to be successful, so that, for example, Pods with an older version can be terminated as part of a rolling update.

For applications that need a very long time to initialize, it's likely that failing liveness checks will cause your container to be restarted before the startup is finished. To prevent these unwanted shutdowns, you can use *startup probes* to indicate when the startup is finished.

Startup Probes

Liveness probes can also be used exclusively to allow for long startup times by stretching the check intervals, increasing the number of retries, and adding a longer delay for the initial liveness probe check. This strategy, however, is not optimal since these timing parameters will also apply for the post-startup phase and will prevent your application from quickly restarting when fatal errors occur.

When applications take minutes to start (for example, Jakarta EE application servers), Kubernetes provides *startup probes*.

Startup probes are configured with the same format as liveness probes but allow for different values for the probe action and the timing parameters. The `periodSeconds` and `failureThreshold` parameters are configured with much larger values compared to the corresponding liveness probes to factor in the longer application startup. Liveness and readiness probes are called only after the startup probe reports success. The container is restarted if the startup probe is not successful within the configured failure threshold.

While the same probe action can be used for liveness and startup probes, a successful startup is often indicated by a marker file that is checked for existence by the startup probe.

Example 4-4 is a typical example of a Jakarta EE application server that takes a long time to start.

Example 4-4. Container with a startup and liveness probe

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-startup-check
spec:
  containers:
  - image: quay.io/wildfly/wildfly ❶
    name: wildfly
    startupProbe:
      exec:
        command: [ "stat", "/opt/jboss/wildfly/standalone/tmp/startup-marker" ] ❷
      initialDelaySeconds: 60 ❸
      periodSeconds: 60
      failureThreshold: 15
    livenessProbe:
      httpGet:
        path: /health
        port: 9990
        periodSeconds: 10 ❹
        failureThreshold: 3
```

❶ JBoss WildFly Jakarta EE server that will take its time to start.

❷ Marker file that is created by WildFly after a successful startup.

❸ Timing parameters that specify that the container should be restarted when it has not been passing the startup probe after 15 minutes (60-second pause until the first check, then maximal 15 checks with 60-second intervals).

❹ Timing parameters for the liveness probes are much smaller, resulting in a restart if subsequent liveness probes fail within 20 seconds (three retries with 10-second pauses between each).

The liveness, readiness, and startup probes are fundamental building blocks of the automation of cloud native applications. Application frameworks such as Quarkus SmallRye Health, Spring Boot Actuator, WildFly Swarm health check, Apache Karaf health check, or the MicroProfile spec for Java provide implementations for offering health probes.

Discussion

To be fully automatable, cloud native applications must be highly observable by providing a means for the managing platform to read and interpret the application health, and if necessary, take corrective actions. Health checks play a fundamental role in the automation of activities such as deployment, self-healing, scaling, and others. However, there are also other means through which your application can provide more visibility about its health.

The obvious and old method for this purpose is through logging. It is a good practice for containers to log any significant events to system out and system error and have these logs collected to a central location for further analysis. Logs are not typically used for taking automated actions but rather to raise alerts and further investigations. A more useful aspect of logs is the postmortem analysis of failures and detection of unnoticeable errors.

Apart from logging to standard streams, it is also a good practice to log the reason for exiting a container to */dev/termination-log*. This location is the place where the container can state its last will before being permanently vanished.[1] Figure 4-1 shows the possible options for how a container can communicate with the runtime platform.

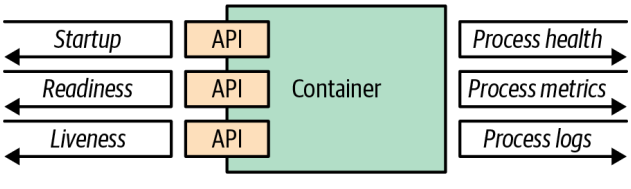

Figure 4-1. Container observability options

Containers provide a unified way for packaging and running applications by treating them like opaque systems. However, any container that is aiming to become a cloud native citizen must provide APIs for the runtime environment to observe the container health and act accordingly. This support is a fundamental prerequisite for automation of the container updates and lifecycle in a unified way, which in turn improves the system's resilience and user experience. In practical terms, that means, as a very minimum, your containerized application must provide APIs for the different kinds of health checks (liveness and readiness).

Even-better-behaving applications must also provide other means for the managing platform to observe the state of the containerized application by integrating with

1 Alternatively, you could change the `.spec.containers.terminationMessagePolicy` field of a Pod to `FallbackToLogsOnError`, in which case the last line of the log is used for the Pod's status message when it terminates.

tracing and metrics-gathering libraries such as OpenTracing or Prometheus. Treat your application as an opaque system, but implement all the necessary APIs to help the platform observe and manage your application in the best way possible.

The next pattern, *Managed Lifecycle*, is also about communication between applications and the Kubernetes management layer, but coming from the other direction. It's about how your application gets informed about important Pod lifecycle events.

More Information

- Health Probe Example
- Configure Liveness, Readiness, and Startup Probes
- Kubernetes Best Practices: Setting Up Health Checks with Readiness and Liveness Probes
- Graceful Shutdown with Node.js and Kubernetes
- Kubernetes Startup Probe—Practical Guide
- Improving Application Availability with Pod Readiness Gates
- Customizing the Termination Message
- SmallRye Health
- Spring Boot Actuator: Production-Ready Features
- Advanced Health Check Patterns in Kubernetes

CHAPTER 5

Managed Lifecycle

Containerized applications managed by cloud native platforms have no control over their lifecycle, and to be good cloud native citizens, they have to listen to the events emitted by the managing platform and adapt their lifecycles accordingly. The *Managed Lifecycle* pattern describes how applications can and should react to these lifecycle events.

Problem

In Chapter 4, "Health Probe", we explained why containers have to provide APIs for the different health checks. Health-check APIs are read-only endpoints the platform is continually probing to get application insight. It is a mechanism for the platform to extract information from the application.

In addition to monitoring the state of a container, the platform sometimes may issue commands and expect the application to react to them. Driven by policies and external factors, a cloud native platform may decide to start or stop the applications it is managing at any moment. It is up to the containerized application to determine which events are important to react to and how to react. But in effect, this is an API that the platform is using to communicate and send commands to the application. Also, applications are free to either benefit from lifecycle management or ignore it if they don't need this service.

Solution

We saw that checking only the process status is not a good enough indication of the health of an application. That is why there are different APIs for monitoring the health of a container. Similarly, using only the process model to run and stop a process is not good enough. Real-world applications require more fine-grained

interactions and lifecycle management capabilities. Some applications need help to warm up, and some applications need a gentle and clean shutdown procedure. For this and other use cases, some events, as shown in Figure 5-1, are emitted by the platform that the container can listen to and react to if desired.

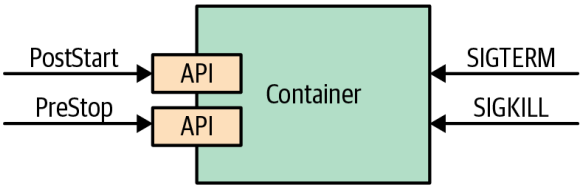

Figure 5-1. Managed container lifecycle

The deployment unit of an application is a Pod. As you already know, a Pod is composed of one or more containers. At the Pod level, there are other constructs such as init containers, which we cover in Chapter 15, "Init Container", that can help manage the container lifecycle. The events and hooks we describe in this chapter are all applied at an individual container level rather than the Pod level.

SIGTERM Signal

Whenever Kubernetes decides to shut down a container, whether that is because the Pod it belongs to is shutting down or simply because a failed liveness probe causes the container to be restarted, the container receives a SIGTERM signal. SIGTERM is a gentle poke for the container to shut down cleanly before Kubernetes sends a more abrupt SIGKILL signal. Once a SIGTERM signal has been received, the application should shut down as quickly as possible. For some applications, this might be a quick termination, and some other applications may have to complete their in-flight requests, release open connections, and clean up temp files, which can take a slightly longer time. In all cases, reacting to SIGTERM is the right moment to shut down a container in a clean way.

SIGKILL Signal

If a container process has not shut down after a SIGTERM signal, it is shut down forcefully by the following SIGKILL signal. Kubernetes does not send the SIGKILL signal immediately but waits 30 seconds by default after it has issued a SIGTERM signal. This grace period can be defined per Pod via the `.spec.terminationGracePeriodSeconds` field, but it cannot be guaranteed as it can be overridden while issuing commands to Kubernetes. The aim should be to design and implement containerized applications to be ephemeral with quick startup and shutdown processes.

PostStart Hook

Using only process signals for managing lifecycles is somewhat limited. That is why additional lifecycle hooks such as `postStart` and `preStop` are provided by Kubernetes. A Pod manifest containing a `postStart` hook looks like the one in Example 5-1.

Example 5-1. A container with `postStart` hook

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      postStart:
        exec:
          command:  ❶
          - sh
          - -c
          - sleep 30 && echo "Wake up!" > /tmp/postStart_done
```

❶ The `postStart` command waits 30 seconds. `sleep` is just a simulation for any lengthy startup code that might run at this point. Also, it uses a trigger file to sync with the main application, which starts in parallel.

The `postStart` command is executed after a container is created, asynchronously with the primary container's process. Even if much of the application initialization and warm-up logic can be implemented as part of the container startup steps, `postStart` still covers some use cases. The `postStart` action is a blocking call, and the container status remains *Waiting* until the `postStart` handler completes, which in turn keeps the Pod status in the *Pending* state. This nature of `postStart` can be used to delay the startup state of the container while allowing time for the main container process to initialize.

Another use of `postStart` is to prevent a container from starting when the Pod does not fulfill certain preconditions. For example, when the `postStart` hook indicates an error by returning a nonzero exit code, Kubernetes kills the main container process.

The `postStart` and `preStop` hook invocation mechanisms are similar to the health probes described in Chapter 4, "Health Probe", and support these handler types:

exec
: Runs a command directly in the container

httpGet
: Executes an HTTP GET request against a port opened by one Pod container

You have to be very careful what critical logic you execute in the `postStart` hook as there are no guarantees for its execution. Since the hook is running in parallel with the container process, it is possible that the hook may be executed before the container has started. Also, the hook is intended to have at-least-once semantics, so the implementation has to take care of duplicate executions. Another aspect to keep in mind is that the platform does not perform any retry attempts on failed HTTP requests that didn't reach the handler.

PreStop Hook

The `preStop` hook is a blocking call sent to a container before it is terminated. It has the same semantics as the SIGTERM signal and should be used to initiate a graceful shutdown of the container when reacting to SIGTERM is not possible. The `preStop` action in Example 5-2 must complete before the call to delete the container is sent to the container runtime, which triggers the SIGTERM notification.

Example 5-2. A container with a `preStop` hook

```
apiVersion: v1
kind: Pod
metadata:
  name: pre-stop-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      preStop:
        httpGet:  ❶
          path: /shutdown
          port: 8080
```

❶ Call out to a `/shutdown` endpoint running within the application.

Even though `preStop` is blocking, holding on it or returning an unsuccessful result does not prevent the container from being deleted and the process killed. The `preStop` hook is only a convenient alternative to a SIGTERM signal for graceful application shutdown and nothing more. It also offers the same handler types and guarantees as the `postStart` hook we covered previously.

Other Lifecycle Controls

In this chapter, so far we have focused on the hooks that allow you to execute commands when a container lifecycle event occurs. But another mechanism that is not at the container level but at the Pod level allows you to execute initialization instructions.

We describe the *Init Container* pattern in Chapter 15 in depth, but here we describe it briefly to compare it with lifecycle hooks. Unlike regular application containers, init containers run sequentially, run until completion, and run before any of the application containers in a Pod start up. These guarantees allow you to use init containers for Pod-level initialization tasks. Both lifecycle hooks and init containers operate at a different granularity (at the container level and Pod level, respectively) and can be used interchangeably in some instances, or complement one another in other cases. Table 5-1 summarizes the main differences between the two.

Table 5-1. Lifecycle hooks and init containers

Aspect	Lifecycle hooks	Init containers
Activates on	Container lifecycle phases.	Pod lifecycle phases.
Startup phase action	A `postStart` command.	A list of `initContainers` to execute.
Shutdown phase action	A `preStop` command.	No equivalent feature.
Timing guarantees	A `postStart` command is executed at the same time as the container's `ENTRY POINT`.	All init containers must be completed successfully before any application container can start.
Use cases	Perform noncritical startup/shutdown cleanups specific to a container.	Perform workflow-like sequential operations using containers; reuse containers for task executions.

If even more control is required to manage the lifecycle of your application containers, there is an advanced technique for rewriting the container entrypoints, sometimes also referred to as the *Commandlet* pattern. This pattern is especially useful when the main containers within a Pod have to be started in a certain order and need an extra level of control. Kubernetes-based pipeline platforms like Tekton and Argo CD require the sequential execution of containers that share data and support the inclusion of additional sidecar containers running in parallel (we talk more about sidecars in Chapter 16, "Sidecar").

For these scenarios, a sequence of init containers is not good enough because init containers don't allow sidecars. As an alternative, an advanced technique called *entrypoint rewriting* can be used to allow fine-grained lifecycle control for the Pod's main containers. Every container image defines a command that is executed by default when the container starts. In a Pod specification, you can also define this command directly in the Pod spec. The idea of entrypoint rewriting is to replace this command with a generic wrapper command that calls the original command and takes care of lifecycle concerns. This generic command is injected from another container image before the application container starts.

This concept is best explained by an example. Example 5-3 shows a typical Pod declaration that starts a single container with the given arguments.

Example 5-3. Simple Pod starting an image with a command and arguments

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-random-generator
spec:
  restartPolicy: OnFailure
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    command:
    - "random-generator-runner"  ❶
    args:                        ❷
    - "--seed"
    - "42"
```

❶ The command executed when the container starts.

❷ Additional arguments provided to the entrypoint command.

The trick is now to wrap the given command `random-generator-runner` with a generic supervisor program that takes care of lifecycle aspects, like reacting on `SIGTERM` or other external signals. Example 5-4 demonstrates a Pod declaration that includes an init container for installing a supervisor, which is then started to monitor the main application.

Example 5-4. Pod that wraps the original entrypoint with a supervisor

```
apiVersion: v1
kind: Pod
metadata:
  name: wrapped-random-generator
spec:
  restartPolicy: OnFailure
  volumes:
  - name: wrapper                          ❶
    emptyDir: { }
  initContainers:
  - name: copy-supervisor                  ❷
    image: k8spatterns/supervisor
    volumeMounts:
    - mountPath: /var/run/wrapper
      name: wrapper
    command: [ cp ]
    args: [ supervisor, /var/run/wrapper/supervisor ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - mountPath: /var/run/wrapper
      name: wrapper
    command:
    - "/var/run/wrapper/supervisor"        ❸
    args:                                  ❹
    - "random-generator-runner"
    - "--seed"
    - "42"
```

❶ A fresh `emptyDir` volume is created to share the supervisor daemon.

❷ Init container used for copying the supervisor daemon to the application containers.

❸ The original command `randomGenerator` as defined in Example 5-3 is replaced with supervisor daemon from the shared volume.

❹ The original command specification becomes the arguments for the supervisor commands.

This entrypoint rewriting is especially useful for Kubernetes-based applications that create and manage Pods programmatically, like Tekton, which creates Pods when running a continuous integration (CI) pipeline. That way, they gain much better control of when to start, stop, or chain containers within a Pod.

There are no strict rules about which mechanism to use except when you require a specific timing guarantee. We could skip lifecycle hooks and init containers entirely and use a bash script to perform specific actions as part of a container's startup or shutdown commands. That is possible, but it would tightly couple the container with the script and turn it into a maintenance nightmare. We could also use Kubernetes lifecycle hooks to perform some actions, as described in this chapter. Alternatively, we could go even further and run containers that perform individual actions using init containers or inject supervisor daemons for even more sophisticated control. In this sequence, the options require increasingly more effort, but at the same time offer stronger guarantees and enable reuse.

Understanding the stages and available hooks of containers and Pod lifecycles is crucial for creating applications that benefit from being managed by Kubernetes.

Discussion

One of the main benefits the cloud native platform provides is the ability to run and scale applications reliably and predictably on top of potentially unreliable cloud infrastructure. These platforms provide a set of constraints and contracts for an application running on them. It is in the interest of the application to honor these contracts to benefit from all of the capabilities offered by the cloud native platform. Handling and reacting to these events ensures that your application can gracefully start up and shut down with minimal impact on the consuming services. At the moment, in its basic form, that means the containers should behave as any well-designed POSIX process should. In the future, there might be even more events giving hints to the application when it is about to be scaled up or asked to release resources to prevent being shut down. It is essential to understand that the application lifecycle is no longer in the control of a person but is fully automated by the platform.

Besides managing the application lifecycle, the other big duty of orchestration platforms like Kubernetes is to distribute containers over a fleet of nodes. The next pattern, *Automated Placement*, explains the options to influence the scheduling decisions from the outside.

More Information

- Managed Lifecycle Example
- Container Lifecycle Hooks
- Attach Handlers to Container Lifecycle Events
- Kubernetes Best Practices: Terminating with Grace
- Graceful Shutdown of Pods with Kubernetes
- Argo and Tekton: Pushing the Boundaries of the Possible on Kubernetes
- Russian Doll: Extending Containers with Nested Processes

CHAPTER 6

Automated Placement

Automated Placement is the core function of the Kubernetes scheduler for assigning new Pods to nodes that match container resource requests and honor scheduling policies. This pattern describes the principles of the Kubernetes scheduling algorithm and how to influence the placement decisions from the outside.

Problem

A reasonably sized microservices-based system consists of tens or even hundreds of isolated processes. Containers and Pods do provide nice abstractions for packaging and deployment but do not solve the problem of placing these processes on suitable nodes. With a large and ever-growing number of microservices, assigning and placing them individually to nodes is not a manageable activity.

Containers have dependencies among themselves, dependencies to nodes, and resource demands, and all of that changes over time too. The resources available on a cluster also vary over time, through shrinking or extending the cluster or by having it consumed by already-placed containers. The way we place containers impacts the availability, performance, and capacity of the distributed systems as well. All of that makes scheduling containers to nodes a moving target.

Solution

In Kubernetes, assigning Pods to nodes is done by the scheduler. It is a part of Kubernetes that is highly configurable, and it is still evolving and improving. In this chapter, we cover the main scheduling control mechanisms, driving forces that affect the placement, why to choose one or the other option, and the resulting consequences. The Kubernetes scheduler is a potent and time-saving tool. It plays a fundamental role in the Kubernetes platform as a whole, but similar to other

Kubernetes components (API Server, Kubelet), it can be run in isolation or not used at all.

At a very high level, the main operation the Kubernetes scheduler performs is to retrieve each newly created Pod definition from the API Server and assign it to a node. It finds the most suitable node for every Pod (as long as there is such a node), whether that is for the initial application placement, scaling up, or when moving an application from an unhealthy node to a healthier one. It does this by considering runtime dependencies, resource requirements, and guiding policies for high availability; by spreading Pods horizontally; and also by colocating Pods nearby for performance and low-latency interactions. However, for the scheduler to do its job correctly and allow declarative placement, it needs nodes with available capacity and containers with declared resource profiles and guiding policies in place. Let's look at each of these in more detail.

Available Node Resources

First of all, the Kubernetes cluster needs to have nodes with enough resource capacity to run new Pods. Every node has capacity available for running Pods, and the scheduler ensures that the sum of the container resources requested for a Pod is less than the available allocatable node capacity. Considering a node dedicated only to Kubernetes, its capacity is calculated using the following formula in Example 6-1.

Example 6-1. Node capacity

```
Allocatable [capacity for application pods] =
    Node Capacity [available capacity on a node]
        - Kube-Reserved [Kubernetes daemons like kubelet, container runtime]
        - System-Reserved [Operating System daemons like sshd, udev]
        - Eviction Thresholds [Reserved memory to prevent system OOMs]
```

If you don't reserve resources for system daemons that power the OS and Kubernetes itself, the Pods can be scheduled up to the full capacity of the node, which may cause Pods and system daemons to compete for resources, leading to resource starvation issues on the node. Even then, memory pressure on the node can affect all Pods running on it through OOMKilled errors or cause the node to go temporarily offline. OOMKilled is an error message displayed when the Linux kernel's Out-of-Memory (OOM) killer terminates a process because the system is out of memory. Eviction thresholds are the last resort for the Kubelet to reserve memory on the node and attempt to evict Pods when the available memory drops below the reserved value.

Also keep in mind that if containers are running on a node that is not managed by Kubernetes, the resources used by these containers are not reflected in the node capacity calculations by Kubernetes. A workaround is to run a placeholder Pod that doesn't do anything but has only resource requests for CPU and memory

corresponding to the untracked containers' resource use amount. Such a Pod is created only to represent and reserve the resource consumption of the untracked containers and helps the scheduler build a better resource model of the node.

Container Resource Demands

Another important requirement for an efficient Pod placement is to define the containers' runtime dependencies and resource demands. We covered that in more detail in Chapter 2, "Predictable Demands". It boils down to having containers that declare their resource profiles (with `request` and `limit`) and environment dependencies such as storage or ports. Only then are Pods optimally assigned to nodes and can run without affecting one another and facing resource starvation during peak usage.

Scheduler Configurations

The next piece of the puzzle is having the right filtering or priority configurations for your cluster needs. The scheduler has a default set of predicate and priority policies configured that is good enough for most use cases. In Kubernetes versions before v1.23, a scheduling policy can be used to configure the predicates and priorities of a scheduler. Newer versions of Kubernetes moved to scheduling profiles to achieve the same effect. This new approach exposes the different steps of the scheduling process as an extension point and allows you to configure plugins that override the default implementations of the steps. Example 6-2 demonstrates how to override the `PodTopologySpread` plugin from the `score` step with custom plugins.

Example 6-2. A scheduler configuration

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:                          ❶
        disabled:
        - name: PodTopologySpread     ❷
        enabled:
        - name: MyCustomPlugin        ❸
          weight: 2
```

❶ The plugins in this phase provide a score to each node that has passed the filtering phase.

❷ This plugin implements topology spread constraints that we will see later in the chapter.

❸ The disabled plugin in the previous step is replaced by a new one.

Scheduler plugins and custom schedulers should be defined only by an administrator as part of the cluster configuration. As a regular user deploying applications on a cluster, you can just refer to predefined schedulers.

By default, the scheduler uses the default-scheduler profile with default plugins. It is also possible to run multiple schedulers on the cluster, or multiple profiles on the scheduler, and allow Pods to specify which profile to use. Each profile must have a unique name. Then when defining a Pod, you can add the field `.spec.schedulerName` with the name of your profile to the Pod specification, and the Pod will be processed by the desired scheduler profile.

Scheduling Process

Pods get assigned to nodes with certain capacities based on placement policies. For completeness, Figure 6-1 visualizes at a high level how these elements get together and the main steps a Pod goes through when being scheduled.

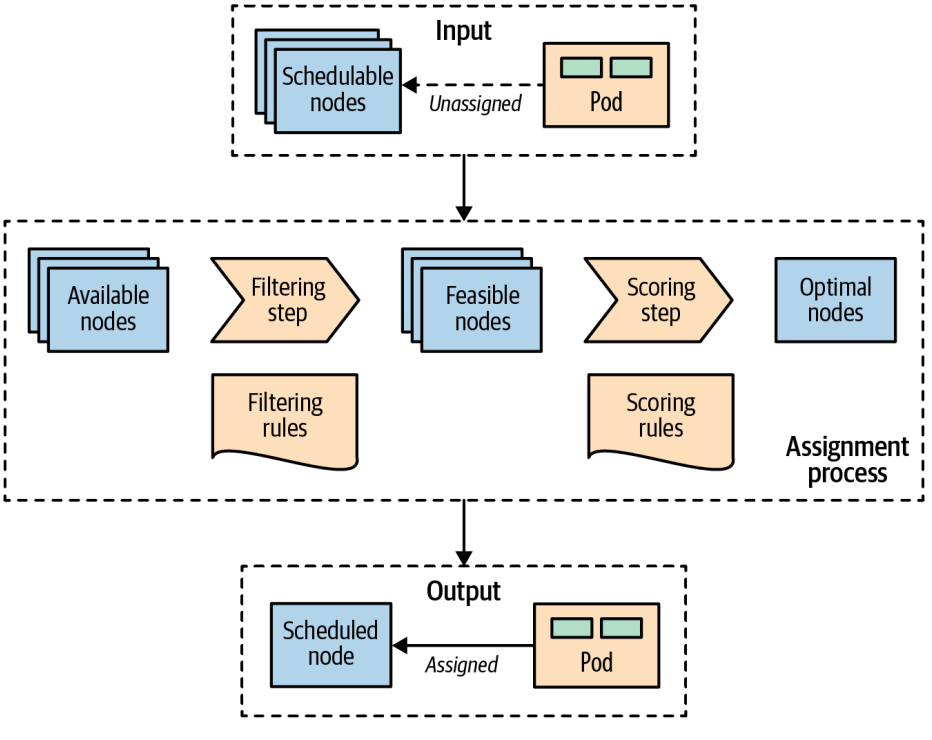

Figure 6-1. A Pod-to-node assignment process

As soon as a Pod is created that is not assigned to a node yet, it gets picked by the scheduler together with all the available nodes and the set of filtering and priority policies. In the first stage, the scheduler applies the filtering policies and removes all nodes that do not qualify. Nodes that meet the Pod's scheduling requirements are called *feasible nodes*. In the second stage, the scheduler runs a set of functions to score the remaining feasible nodes and orders them by weight. In the last stage, the scheduler notifies the API server about the assignment decision, which is the primary outcome of the scheduling process. This whole process is also referred to as *scheduling*, *placement*, *node assignment*, or *binding*.

In most cases, it is better to let the scheduler do the Pod-to-node assignment and not micromanage the placement logic. However, on some occasions, you may want to force the assignment of a Pod to a specific node or group of nodes. This assignment can be done using a node selector. The `.spec.nodeSelector` Pod field specifies a map of key-value pairs that must be present as labels on the node for the node to be eligible to run the Pod. For example, let's say you want to force a Pod to run on a specific node where you have SSD storage or GPU acceleration hardware. With the Pod definition in Example 6-3 that has `nodeSelector` matching `disktype: ssd`, only nodes that are labeled with `disktype=ssd` will be eligible to run the Pod.

Example 6-3. Node selector based on type of disk available

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  nodeSelector:
    disktype: ssd       ❶
```

❶ Set of node labels a node must match to be considered the node of this Pod.

In addition to specifying custom labels to your nodes, you can use some of the default labels that are present on every node. Every node has a unique `kubernetes.io/hostname` label that can be used to place a Pod on a node by its hostname. Other default labels that indicate the OS, architecture, and instance type can be useful for placement too.