O'REILLY®

Second Edition

Kubernetes Patterns

Reusable Elements for Designing Cloud-Native Applications

Early Release
Raw & Unedited

Compliments of Red Hat

Bilgin Ibryam &
Roland Huß

Red Hat

Build **Smarter.**
Ship **Faster.**

To make the most of the cloud, IT needs to approach applications in new ways. Cloud-native development means packaging with containers, adopting modern architectures, and using agile techniques.

Red Hat can help you arrange your people, processes, and technologies to build, deploy, and run cloud-ready applications anywhere they are needed. Discover how with **cloud-native development solutions.**

Kubernetes Patterns

Reusable Elements for Designing Cloud-Native Applications

Bilgin Ibryam and Roland Huß

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes Patterns

by Bilgin Ibryam and Roland Huß

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Rita Fernando
Production Editor: Kate Galloway
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2019:	First Edition
May 2023:	Second Edition

Revision History for the Early Release

2022-09-01:	First Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=9781098131685* for release details.

978-1-098-13162-3

Table of Contents

CHAPTER 1

Declarative Deployment

The heart of the *Declarative Deployment* pattern is Kubernetes' Deployment resource. This abstraction encapsulates the upgrade and rollback processes of a group of containers and makes its execution a repeatable and automated activity.

Problem

We can provision isolated environments as namespaces in a self-service manner and have the services placed in these environments with minimal human intervention through the scheduler. But with a growing number of microservices, continually updating and replacing them with newer versions becomes an increasing burden too.

Upgrading a service to a next version involves activities such as starting the new version of the Pod, stopping the old version of a Pod gracefully, waiting and verifying that it has launched successfully, and sometimes rolling it all back to the previous version in the case of failure. These activities are performed either by allowing some downtime but no running concurrent service versions, or with no downtime, but increased resource usage due to both versions of the service running during the update process. Performing these steps manually can lead to human errors, and scripting properly can require a significant amount of effort, both of which quickly turn the release process into a bottleneck.

Solution

Luckily, Kubernetes has automated application upgrades as well. Using the concept of *Deployment*, we can describe how our application should be updated, using different strategies, and tuning the various aspects of the update process. If you consider that you do multiple Deployments for every microservice instance per release cycle (which, depending on the team and project, can span from minutes to several months), this is another effort-saving automation by Kubernetes.

In Chapter 2, we have seen that, to do its job effectively, the scheduler requires sufficient resources on the host system, appropriate placement policies, and containers with adequately defined resource profiles. Similarly, for a Deployment to do its job correctly, it expects the containers to be good cloud-native citizens. At the very core of a Deployment is the ability to start and stop a set of Pods predictably. For this to work as expected, the containers themselves usually listen and honor lifecycle events (such as SIGTERM; see Chapter 3, "Managed Lifecycle") and also provide health-check endpoints as described in Chapter 2, "Health Probe", which indicate whether they started successfully.

If a container covers these two areas accurately, the platform can cleanly shut down old containers and replace them by starting updated instances. Then all the remaining aspects of an update process can be defined in a declarative way and executed as one atomic action with predefined steps and an expected outcome. Let's see the options for a container update behavior.

Deployment updates with `kubectl rollout`

In previous versions of Kubernetes, rolling updates have been implemented on the client-side with the `kubectl rolling-update` command. In Kubernetes 1.18, `rolling-update` has been removed in favor of a `rollout` command for kubectl. The difference is that `kubectl rollout` will manage an application update on the server-side by updating the Deployment *declaration* and leaving it to Kubernetes to perform the update. `kubectl rolling-update`, in contrast, was *imperative* in nature; the client `kubectl` tells the server what to do for each update step.

A Deployment can be fully managed by updating the Kubernetes resources files. However, `kubectl rollout` comes in very handy for everyday rollout tasks:

- `kubectl rollout status` shows the current status of a Deployment's rollout

- `kubectl rollout pause` will pause a rolling update so that multiple changes can be applied to a Deployment with retriggering another rollout.
- `kubectl rollout resume` resumes a previously paused rollout.
- `kubectl rollout undo` performs a rollback to a prevision revision of a Deployment. A rollback is helpful in case of an error during the update.
- `kubectl rollout history` shows the available revisions of a Deployment.
- `kubectl rollout restart` does not perform an update but restarts the current set of Pods belonging to a Deployment using the configured rollout strategy.

You find usage examples for `kubectl rollouts` in the examples.

Rolling Deployment

The declarative way of updating applications in Kubernetes is through the concept of Deployment. Behind the scenes, the Deployment creates a ReplicaSet that supports set-based label selectors. Also, the Deployment abstraction allows shaping the update process behavior with strategies such as `RollingUpdate` (default) and `Recreate`. Example 1-1 shows the important bits for configuring a Deployment for a rolling update strategy.

Example 1-1. Deployment for a rolling update

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
spec:
  replicas: 3                 ❶
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1             ❷
      maxUnavailable: 1       ❸
  minReadySeconds: 60         ❹
selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        readinessProbe:     ❺
          exec:
            command: [ "stat", "/random-generator-ready" ]
```

❶ Declaration of three replicas. You need more than one replica for a rolling update to make sense.

❷ Number of Pods that can be run temporarily in addition to the replicas specified during an update. In this example, it could be a total of four replicas at maximum.

❸ Number of Pods that can be unavailable during the update. Here it could be that only two Pods are available at a time during the update.

❹ Duration how long all readiness probes for a rolled out Pod needs to be healthy until to continue with the rollout.

❺ Readiness probes are very important for a rolling deployment to provide zero downtime—don't forget them (see Chapter 2, "Health Probe").

`RollingUpdate` strategy behavior ensures there is no downtime during the update process. Behind the scenes, the Deployment implementation performs similar moves by creating new ReplicaSets and replacing old containers with new ones. One enhancement here is that with Deployment, it is possible to control the rate of a new container rollout. The Deployment object allows you to control the range of available and excess Pods through `maxSurge` and `maxUnavailable` fields.

These two fields can be either absolute numbers of Pods or relative percentages that are applied to configured number of replicas for the Deployment and are rounded up (`maxSurge`) or down (`maxUnavailable`) to the next integer value. By default `maxSurge` and `maxUnavailable` are both set to 25%.

Another important parameter that influences the rollout behaviour is `minReadySeconds`. This field specifies the duration in seconds how long the readiness probes of a Pod need to return success until the Pod itself is considered to be available in a rollout. Increasing this value guarantees that your application Pod is succesfully running for some time already before continuing with the rollout. Also, a larger `minReadySeconds` interval helps in debugging and exploring the new version. A `kubectl rollout pause` might be easier to leverage when the intervals between the update steps is larger.

Figure 1-1 shows the rolling update process.

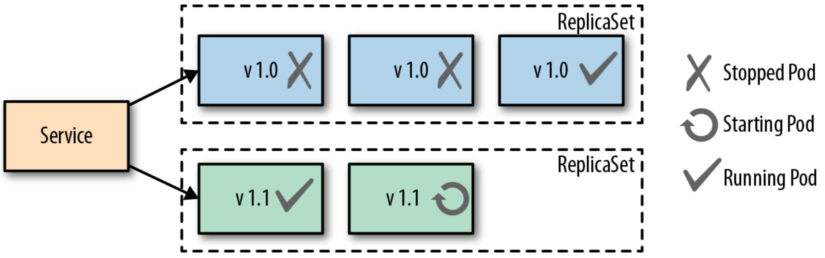

Figure 1-1. Rolling deployment

To trigger a declarative update, you have three options:

- Replace the whole Deployment with the new version's Deployment with `kubectl replace`.
- Patch (`kubectl patch`) or interactively edit (`kubectl edit`) the Deployment to set the new container image of the new version.
- Use `kubectl set image` to set the new image in the Deployment.

See also the full example in our example repository, which demonstrates the usage of these commands, and shows you how you can monitor or roll back an upgrade with `kubectl rollout`.

In addition to addressing the previously mentioned drawbacks of the imperative way of deploying services, the Deployment brings the following benefits:

- Deployment is a Kubernetes resource object whose status is entirely managed by Kubernetes internally. The whole update process is performed on the server side without client interaction.
- The declarative nature of Deployment makes you see how the deployed state should look rather than the steps necessary to get there.
- The Deployment definition is an executable object, tried and tested on multiple environments before reaching production.
- The update process is also wholly recorded, and versioned with options to pause, continue, and roll back to previous versions.

Fixed Deployment

A `RollingUpdate` strategy is useful for ensuring zero downtime during the update process. However, the side effect of this approach is that during the update process, two versions of the container are running at the same time. That may cause issues for the service consumers, especially when the update process has introduced backward-incompatible changes in the service APIs and the client is not capable of dealing with them. For this kind of scenario, there is the `Recreate` strategy, which is illustrated in Figure 1-2.

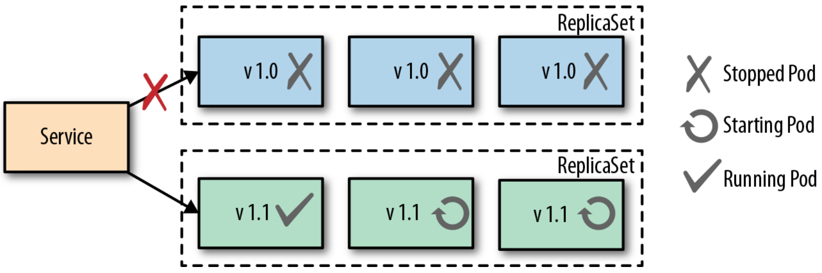

Figure 1-2. Fixed deployment using a Recreate strategy

The Recreate strategy has the effect of setting `maxUnavailable` to the number of declared replicas. This means it first kills all containers from the current version and then starts all new containers simultaneously when the old containers are evicted. The result of this sequence of actions is that there is some downtime while all containers with old versions are stopped, and there are no new containers ready to handle incoming requests. On the positive side, there won't be two versions of the containers running at the same time, simplifying the life of service consumers to handle only one version at a time.

Blue-Green Release

The *Blue-Green deployment* is a release strategy used for deploying software in a production environment by minimizing downtime and reducing risk. Kubernetes' Deployment abstraction is a fundamental concept that lets you define how Kubernetes transitions immutable containers from one version to another. We can use the Deployment primitive as a building block, together with other Kubernetes primitives, to implement this more advanced release strategy of a Blue-Green deployment.

A Blue-Green deployment needs to be done manually if no extensions like a Service Mesh or Knative is used, though. Technically it works by creating a second Deployment with the latest version of the containers (let's call it *green*) not serving any requests yet. At this stage, the old Pod replicas (called *blue*) from the original Deployment are still running and serving live requests.

Once we are confident that the new version of the Pods is healthy and ready to handle live requests, we switch the traffic from old Pod replicas to the new replicas. This activity in Kubernetes can be done by updating the Service selector to match the new containers (tagged as green). As demonstrated in Figure 1-3, once the green containers handle all the traffic, the blue containers can be deleted and the resources freed for future Blue-Green deployments.

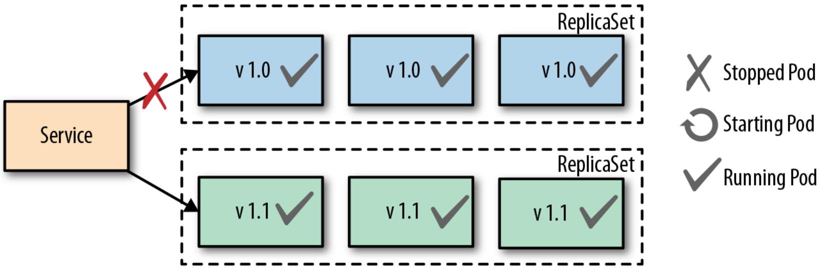

Figure 1-3. Blue-Green release

A benefit of the Blue-Green approach is that there's only one version of the application serving requests, which reduces the complexity of handling multiple concurrent versions by the Service consumers. The downside is that it requires twice the application capacity while both blue and green containers are up and running. Also, there can be significant complications with long-running processes and database state drifts during the transitions.

Canary Release

Canary release is a way to softly deploy a new version of an application into production by replacing only a small subset of old instances with new ones. This technique reduces the risk of introducing a new version into production by letting only some of the consumers reach the updated version. When we are happy with the new version of our service and how it performed with a small sample of users, we can replace all the old instances with the new version in an additional step after this canary release. Figure 1-4 shows a canary release in action.

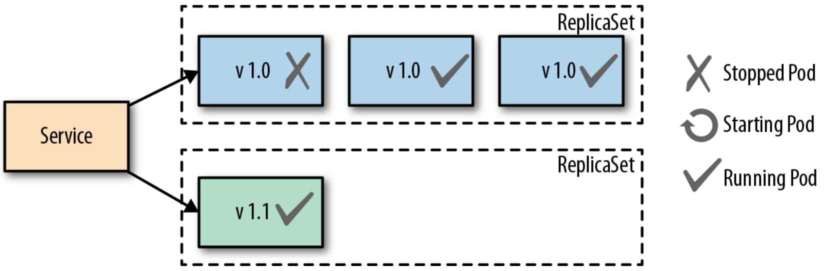

Figure 1-4. Canary release

In Kubernetes, this technique can be implemented by creating a new ReplicaSet for the new container version (preferably using a Deployment) with a small replica count that can be used as the Canary instance. At this stage, the Service should direct some of the consumers to the updated Pod instances. After the canary release and once we are confident that everything with new ReplicaSet works as expected, we scale the new ReplicaSet up, and the old ReplicaSet down to zero. In a way, we are performing a controlled and user-tested incremental rollout.

Discussion

The Deployment primitive is an example of where Kubernetes turns 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 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 but require human interaction. Figure 1-5 shows a summary of the deployment and release strategies, showing instance counts during transitions.

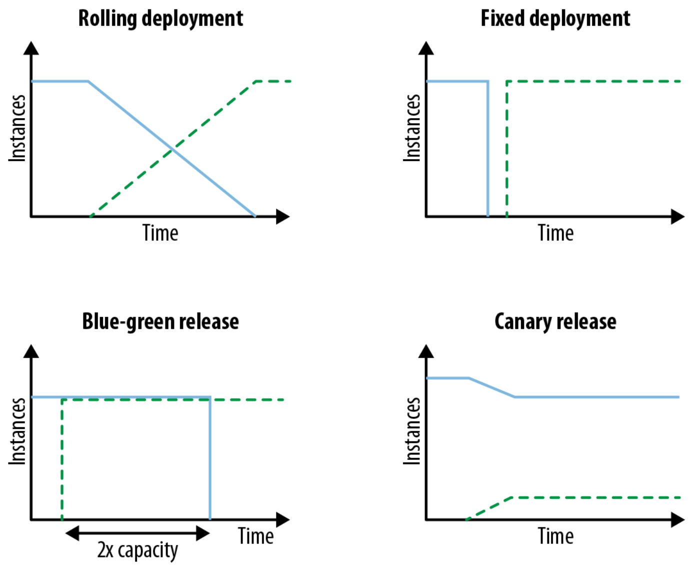

Figure 1-5. Deployment and release strategies

Every 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 2022), 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 "Higher-level Deployments" on page 15. Those techniques work with Operators (see Chapter 23) that take a declarative desciption 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. Higher-level abstractions do exist that extend Kubernetes with new resource types that allow such more advanced deployment strategies to be declared flexibly. Those extensions all leverage the Operator pattern and introduce own custom resources for describing the desired rollout behaviour.

As of 2022, the most prominent platforms that supports higher-level Deployments are:

- **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 some custom metric and detect if the rollout fails so that it can trigger an automatic rollback. Like Kubernetes, Flagger is a Cloud

Native Computing Foundation (CNCF) project with excellent community support.

- **Argo Rollouts** is part of the Argo family of tools (also part of the CNCF) whose focus 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 as Flagger, so the decision on which one to use should be made on which continuous delivery solution you prefer, Argo or Flux.
- **Knative** is a serverless platform on top of Kubernetes. It's main feature is traffic-driven autoscaling support, which is described in detail in Chapter 24. 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 still a substantial improvement over the rollout capabilities of Kubernetes Deployments. If you are using Knative anyway, then the intuitive way of splitting traffic between two application versions is a good alternative to Deployments.

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 reaching the defined target deployment state. The next pattern, *Health Probe*, in Chapter 2 describes how your application can communicate its health state to Kubernetes.

More Information

- Declarative Deployment Examples
- Rolling Update
- Deployments
- Run a Stateless Application Using a Deployment
- Blue-Green Deployment
- Canary Release
- Flagger Deployment Strategies
- Argo Rollouts
- Knative: Traffic Management

CHAPTER 2
Health Probe

The *Health Probe* pattern is about 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 decide about 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 a check that indicates 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 all have 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 see 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. 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 some 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 what 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.
- A TCP Socket probe assumes a successful TCP connection.

- An Exec probe executes an arbitrary command in the container kernel namespace and expects a successful exit code (0).
- A gRPC probe leverages gRPC's intrinsic support for health checks

Beside 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` is the interval in seconds between liveness probe checks.
- `timeoutSeconds` is the maximum time allowed for a probe check to return before it is considered to be failed.
- `failureThreshold` specifies how often a probe check needs to fail in a row until the container is considered to be unhealty and needs to be restarted.

An example HTTP-based liveness probe is shown in Example 2-1.

Example 2-1. 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 implementation to decide when your application is considered healthy or not. However, keep in mind that the result of not passing a health check is restarting of your container. 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 are useful for keeping applications healthy by killing unhealthy containers and replacing them with new ones. But sometimes a container may not be healthy, and restarting it may not help either. The most common example is when a container is still starting up and not ready to handle any requests yet. Or maybe a container is overloaded, and its latency is increasing, and you want it to shield itself from additional load for a while.

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 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 2-2 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 2-2. 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.

In many cases, you have liveness and readiness probes 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.

Custom Pod Readiness Gates

Readiness probes work on a per-container level and a Pod is considered to be "READY" to serve requests when all containers pass their readiness probes. In some situation this is not good enough. E.g. 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 2-3 shows a readiness gate that will introduce an additional condition `k8spatterns.io/load-balancer-ready` to the Pod's `status:` sections.

Example 2-3. Readiness Gate for indicating the status of an external load balancer

```
apiVersion: v1
...
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 that is set to `False` by default. It needs to be switched to `True` externally, e.g. by a *Controller* (Chapter 22) when the load balancer is ready to serve.

❷ The Pod is "ready" when all container's readiness probes are passing and the readiness gates' conditions are `True`, otherwise, like here, the Pod is marked as non-ready.

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

For applications that need a very long time to initialize it is not unlikely that failing liveness checks will cause your container to be restarted before the startup is finished. To prevent these unwanted shutdowns, *startup probes* can be used to indicate when the startup is finished.

Startup Probes

Liveness probes can be also 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 prevent your application to quickly restart when a fatal erros occur.

In the situation when applications take minutes to startup (for example JavaEE 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 as for the corresponding liveness probes to factor in the longer application startup. Liveness and readiness probes are only called 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, often a successful startup is indicated by a marker file that is checked for existance by the startup probe.

Example 2-4 is a typical example of a JavaEE application server that takes a long time to startup.

Example 2-4. 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 JavaEE server that will take its time to startup

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 seconds pause until the first check, then maximal 15 checks with 60 second intervals)

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

The liveness, readiness, and startup probes are fundamental building blocks of automation of cloud-native applications. Application frameworks such as Spring actuator, WildFly Swarm health check, Karaf health checks, 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 detecting 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. Figure 2-1 shows the possible options for how a container can communicate with the runtime platform.

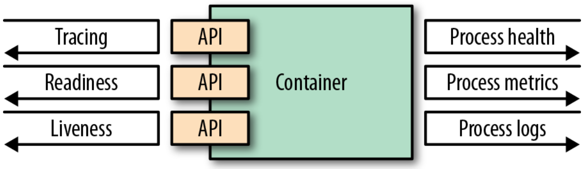

Figure 2-1. Container observability options

Containers provide a unified way for packaging and running applications by treating them like black boxes. 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 tracing and metrics-gathering libraries such as OpenTracing or Prometheus. Treat your application as a black box, 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
- Configuring Liveness, Readiness and Startup Probes
- Setting Up Health Checks Wth Readiness and Liveness Probes
- Graceful Shutdown with Node.js and Kubernetes
- Kubernetes Startup Probe - Practical Guide
- Improving Application Availability with Pod Readiness Gates
- Advanced Health-Check Patterns in Kubernetes

CHAPTER 3

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 2, "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 on these. 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 3-1, are emitted by the platform that the container can listen to and react to if desired.

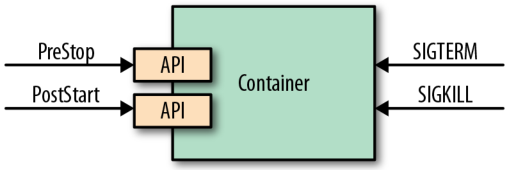

Figure 3-1. Managed container lifecycle

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 14 (and defer-containers, which is still at the proposal stage as of this writing) 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 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 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 for a grace period of 30 seconds by default after it has issued a SIGTERM signal. This grace period can be defined per Pod using the `.spec.terminationGracePeriodSeconds` field, but cannot be guaranteed as it can be overridden while issuing commands to Kubernetes. The aim here 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 there are additional lifecycle hooks such as `postStart` and `preStop` provided by Kubernetes. A Pod manifest containing a `postStart` hook looks like the one in Example 3-1.

Example 3-1. 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 here 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 many of the application initialization and warm-up logic can be implemented as part of the container startup steps, `post`

`Start` 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 giving time to 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, the main container process gets killed by Kubernetes.

`postStart` and `preStop` hook invocation mechanisms are similar to the *Health Probes* described in Chapter 2 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 3-2 must complete before the call to delete the container is sent to the container runtime, which triggers the SIGTERM notification.

Example 3-2. A container with a preStop hook

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

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

Even though `preStop` is blocking, holding on it or returning a nonsuccessful result does not prevent the container from being deleted and the process killed. `preStop` 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 executing commands when a container lifecycle event occurs. But another mechanism that is not at the container level but at a Pod level allows executing initialization instructions.

We describe init containers in Chapter 14, 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 using init containers for Pod-level initialization tasks. Both lifecycle hooks and init containers operate at a different granularity (at container level and Pod-level, respectively) and could be used interchangeably in some instances, or complement each other in other cases. Table 3-1 summarizes the main differences between the two.

Table 3-1. 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 `ENTRYPOINT`	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* (*https://youtu.be/iPRw_n_JV4o*). This pattern is especially useful for situations when the main containers within a Pod has to be started in a certain order and need some extra level of control. Kubernetes-based pipeline platforms like Tekton and Argo CD require a sequential ordering of container execution within a Pod that share data and allow for additional sidecar containers (we talk more about sidecars in Chapter 15).

For these scenarios a sequence of init container 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 3-3 shows a typical Pod declaration that starts a single container with the given arguments.

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

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-random-generator
spec:
  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 `SIG TERM` or other external signals.

Example 3-4. Pod that wraps the original entrypoint with a supervisor

```
apiVersion: v1
kind: Pod
metadata:
  name: wrapped-random-generator
spec:
  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 3-3 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 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 more effort increasingly, 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 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. 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 get into the mindset where the application lifecycle is no longer in the control of a person but 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
- Attaching Handlers to Container Lifecycle Events
- Terminating with Grace
- Graceful Shutdown of Pods with Kubernetes
- Argo and Tekton: Pushing the Boundaries of the Possible on Kubernetes describing entrypoint rewriting
- Russian Doll: Extending Containers with Nested Processes
- Defer Containers

CHAPTER 4

Batch Job

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *rfernando@oreilly.com*.

The *Batch Job* pattern is suited for managing isolated atomic units of work. It is based on Job abstraction, which runs short-lived Pods reliably until completion on a distributed environment.

Problem

The main primitive in Kubernetes for managing and running containers is the Pod. There are different ways of creating Pods with varying characteristics:

Bare Pod
: It is possible to create a Pod manually to run containers. However, when the node such a Pod is running on fails, the Pod is not restarted. Running Pods this way is discouraged except for development or testing purposes. This mechanism is also known under the names of *unmanaged* or *naked Pods*.

ReplicaSet
: This controller is used for creating and managing the lifecycle of Pods expected to run continuously (e.g., to run a web server container). It maintains a stable set of replica Pods running at any given time and guarantees the availability of a specified number of identical Pods.

DaemonSet
: A controller for running a single Pod on every node. Typically used for managing platform capabilities such as monitoring, log aggregation, storage containers, and others. See Chapter 9 for a detailed discussion on DaemonSets.

A common aspect of these Pods is the fact that they represent long-running processes that are not meant to stop after some time. However, in some cases there is a need to perform a predefined finite unit of work reliably and then shut down the container. For this task, Kubernetes provides the Job resource.

Solution

A Kubernetes Job is similar to a ReplicaSet as it creates one or more Pods and ensures they run successfully. However, the difference is that, once the expected number of Pods terminate successfully, the Job is considered complete and no additional Pods are started. A Job definition looks like Example 4-1.

Example 4-1. A Job specification

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-generator
spec:
  completions: 5                      ❶
  parallelism: 2                      ❷
  ttlSecondsAfterFinished: 300        ❸
  template:
    metadata:
      name: random-generator
    spec:
      restartPolicy: OnFailure        ❹
      containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
        command: [ "java", "-cp", "/", "RandomRunner", "/numbers.txt", "10000" ]
```

❶ Job should run five Pods to completion, which all must succeed.

❷ Two Pods can run in parallel.

❸ Keep Pods for five minutes (300 seconds) before garbage collecting them.

❹ Specifying the `restartPolicy` is mandatory for a Job. Possible values are `OnFailure` on `Never`.

One crucial difference between the Job and the ReplicaSet definition is the `.spec.template.spec.restartPolicy`. The default value for a ReplicaSet is `Always`, which makes sense for long-running processes that must always be kept running. The value `Always` is not allowed for a Job and the only possible options are either `OnFailure` or `Never`.

So why bother creating a Job to run a Pod only once instead of using bare Pods? Using Jobs provides many reliability and scalability benefits that make them the preferred option:

- A Job is not an ephemeral in-memory task, but a persisted one that survives cluster restarts.
- When a Job is completed, it is not deleted but kept for tracking purposes. The Pods that are created as part of the Job are also not deleted but available for examination (e.g., to check the container logs). This is also true for bare Pods, but only for a `restartPolicy: OnFailure`. You can still remove the Pods of a Job after a certain time by specifying `.spec.ttlSecondsAfterFinished`.
- A Job may need to be performed multiple times. Using the `.spec.completions` field it is possible to specify how many times a Pod should complete successfully before the Job itself is done.
- When a Job has to be completed multiple times (set through `.spec.completions`), it can also be scaled and executed by starting multiple Pods at the same time. That can be done by specifying the `.spec.parallelism` field.
- A Job can be suspended by setting the field `.spec.suspend` to `true`. In this case all active Pods are deleted and restarted if the Job is resumed (i.e. `.spec.suspend` set to `false` by the user).
- If the node fails or when the Pod is evicted for some reason while still running, the scheduler places the Pod on a new healthy node and reruns it. Bare Pods would remain in a failed state as existing Pods are never moved to other nodes.

All of this makes the Job primitive attractive for scenarios where some guarantees are required for the completion of a unit of work.

The two fields that play major roles in the behavior of a Job are:

`.spec.completions`
: Specifies how many Pods should run to complete a Job.

`.spec.parallelism`
: Specifies how many Pod replicas could run in parallel. Setting a high number does not guarantee a high level of parallelism and the actual number of Pods may still be less (and in some corner cases, more) than the desired number (e.g., due to throttling, resource quotas, not enough completions left, and other reasons). Setting this field to 0 effectively pauses the Job.

Figure 4-1 shows how the *Batch Job* defined in Example 4-1 with a completion count of five and a parallelism of two is processed.

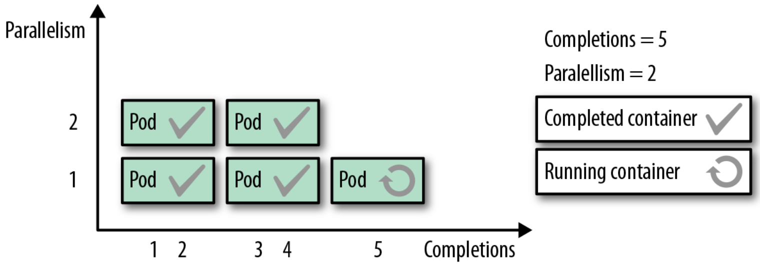

Figure 4-1. Parallel Batch Job with a fixed completion count

Based on these two parameters, there are the following types of Jobs:

Single Pod Job
: This type is selected when you leave out both `.spec.completions` and `.spec.parallelism` or set them to their default values of one. Such a Job starts only one Pod and is completed as soon as the single Pod terminates successfully (with exit code 0).

Fixed completion count Jobs
: When you specify `.spec.completions` with a number greater than one, this many Pods must succeed. Optionally, you can set `.spec.parallelism`, or leave it at the default value of one. Such a Job is considered completed after the `.spec.completions` number of Pods has completed successfully. Example 4-1 shows this mode in action and is the best choice when we know the number of work items in advance, and the processing cost of a single work item justifies the use of a dedicated Pod.

Work queue Jobs
: You have a work queue for parallel Jobs when you leave out `.spec.completions` and set `.spec.parallelism` to an integer greater than one. A work queue Job is considered completed when at least one Pod has terminated successfully, and all other Pods have terminated too. This setup requires the Pods to coordinate among themselves and determine what each one is working on so that they can

finish in a coordinated fashion. For example, when a fixed but unknown number of work items is stored in a queue, parallel Pods can pick these up one by one to work on them. The first Pod that detects that the queue is empty and exits with success indicates the completion of the Job. The Job controller waits for all other Pods to terminate too. Since one Pod processes multiple work items, this Job type is an excellent choice for granular work items—when the overhead for one Pod per work item is not justified.

Indexed Jobs
: Similar to *Work queue Jobs*, you can distribute work items to individual Jobs without needing an external work queue. When using a fixed completion count and setting the completion mode `.spec.completionMode` to `Indexed`, every Pod of the Job gets an associated index ranging from 0 to `.spec.completionMode` - 1. The assigned index is available to the containers through the Pod annotation `batch.kubernetes.io/job-completion-index` (see Chapter 13 for how this annotation can be accessed from your code) or directly via the environment variable `JOB_COMPLETION_INDEX` that is set to the index associated with this Pod. With this index at hand, the application can pick the associated work item without any external synchronization. Example 4-2 shows a Job that processes the lines of a single file individually by separate Pods. A more realistic example would be an indexed Job used for video processing where parallel Pods are processing a certain frame range calculated from the index, as shown in the example.

Example 4-2. An indexed Job selecting their work items based on a job index

```
apiVersion: batch/v1
kind: Job
metadata:
  name: file-split
spec:
  completionMode: Indexed                                           ❶
  completions: 5                                                    ❷
  parallelism: 5
  template:
    metadata:
      name: file-split
    spec:
      containers:
      - image: library/perl
        name: split
        command:
        - "perl"                                                    ❸
        - "-ne"
        - |
          BEGIN {
            $idx = $ENV{JOB_COMPLETION_INDEX};                      ❹
            open($fh,">","/logs/random-${idx}.txt");                ❺
          };
          print $fh $_ if $. >= $idx * 10000 && $. < ($idx+1) * 10000;   ❻
          END {
            close($fh)
          }
        - /logs/random.log
        volumeMounts:
        - mountPath: /logs                                               ❼
          name: log-volume
      restartPolicy: OnFailure
```

❶ Enable an indexed completion mode.

❷ Run 5 pods in parallel to completion.

❸ Execute a Perl scripts that prints out a range of line from a given file `/logs/random.log`. This file is expected to have 50000 lines of data.

❹ Remember the current completion index (0 … 4) in a Perl variable `$idx`.

❺ Open a new file that is based on the index name.

❻ Write out the line of the input file if it is in the range dedicated to the completion index (`$_` is the current line in the input file, `$.` the input line number).

❼ Mount the input data from an external volume. The volume is not shown here, you can find the full working definition in the example repository.

Partitioning the work

As you have seen, we have multiple options for processing many work items by fewer worker Pods. While *Work queue Jobs* can operate on an unknown but finite set of work items, they need support from an external system that provides the work items. In that case, the external system already has divided the work into appropriately sized work items, so the worker Pods have to process those and stop when there is nothing left to do. The alternative is to use *Indexed Jobs*, which do not rely on an external work queue but have to split up the work on their own so that each Pod can separately work on a portion of the overall task. Each Pod needs to know its own identity (provided by the environment variable `JOB_COMPLETION_INDEX`), the total amount of workers, and maybe the overall size of the work (like the size of a movie file to process). Unfortunately, the Job's application code cannot discover the total number of workers (i.e., the value specified in `.spec.completions`) for an indexed job. Therefore, a `JOB_COMPLETION_TOTAL` environment variable would be helpful to partition the work dynamically, but this is not supported as of 2022. However, there are two solutions to overcome this:

- Hardcode the knowledge of the total number of Pods working on a Job into the application code. While this might work for simple examples like in Example 4-2, it's generally an imperfect solution as it couples the code in your container to the Kubernetes declaration. That is, if you want to change the number of completions in your Job definition, you would also have to create a new container image for your Job logic with an updated value.
- Copy the value of `.spec.completions` to an environment variable in the Job's template specification so that your application code can access it. However, this means you have to update two places in the Job declaration if you plan to change the number of completions.

Both solutions have their drawbacks, but these are the best options for now. The community is considering to integrate such an environment variable for `.spec.completions` in *https://github.com/kubernetes/kubernetes/issues/106009* though, so we recommend following this page for updates.

If you have an unlimited stream of work items to process, other controllers like ReplicaSet are the better choice for managing the Pods processing these work items.

Discussion

The Job abstraction is a pretty basic but also fundamental primitive that other primitives such as CronJobs are based on. Jobs help turn isolated work units into a reliable and scalable unit of execution. However, a Job doesn't dictate how you should map individually processable work items into Jobs or Pods. That is something you have to determine after considering the pros and cons of each option:

One Job per work item
: This option has the overhead of creating Kubernetes Jobs, and also for the platform to manage a large number of Jobs that are consuming resources. This option is useful when each work item is a complex task that has to be recorded, tracked, or scaled independently.

One Job for all work items
: This option is right for a large number of work items that do not have to be independently tracked and managed by the platform. In this scenario, the work items have to be managed from within the application via a batch framework.

The Job primitive provides only the very minimum basics for scheduling of work items. Any complex implementation has to combine the Job primitive with a batch application framework (e.g., in the Java ecosystem we have Spring Batch and JBeret as standard implementations) to achieve the desired outcome.

Not all services must run all the time. Some services must run on demand, some on a specific time, and some periodically. Using Jobs can run Pods only when needed, and only for the duration of the task execution. Jobs are scheduled on nodes that have the required capacity, satisfy Pod placement policies, and other container dependency considerations. Using Jobs for short-lived tasks rather than using long-running abstractions (such as ReplicaSet) saves resources for other workloads on the platform. All of that makes Jobs a unique primitive, and Kubernetes a platform supporting diverse workloads.

More Information

- Batch Job Example
- Kubernetes Jobs documentation
- Parallel Processing Using Expansions
- Coarse Parallel Processing Using a Work Queue
- Fine Parallel Processing Using a Work Queue
- Indexed Job for Parallel Processing with Static Work Assignment
- Spring Batch on Kubernetes
- JBeret introduction

CHAPTER 5
Periodic Job

The *Periodic Job* pattern extends the *Batch Job* pattern by adding a time dimension and allowing the execution of a unit of work to be triggered by a temporal event.

Problem

In the world of distributed systems and microservices, there is a clear tendency toward real-time and event-driven application interactions using HTTP and lightweight messaging. However, regardless of the latest trends in software development, job scheduling has a long history, and it is still relevant. *Periodic Jobs* are commonly used for automating system maintenance or administrative tasks. They are also relevant to business applications requiring specific tasks to be performed periodically. Typical examples here are business-to-business integration through file transfer, application integration through database polling, sending newsletter emails, and cleaning up and archiving old files.

The traditional way of handling *Periodic Jobs* for system maintenance purposes has been the use of specialized scheduling software or cron. However, specialized software can be expensive for simple use cases, and cron jobs running on a single server are difficult to maintain and represent a single point of failure. That is why, very often, developers tend to implement solutions that can handle both the scheduling aspect and the business logic that needs to be performed. For example, in the Java world, libraries such as Quartz, Spring Batch, and custom implementations with the `ScheduledThreadPoolExecutor` class can run temporal tasks. But similarly to cron, the main difficulty with this approach is making the scheduling capability resilient and highly available, which leads to high resource consumption. Also, with this approach, the time-based job scheduler is part of the application, and to make the scheduler highly available, the whole application must be highly available. Typically, that involves running multiple instances of the application, and at the same time, ensuring that only a single instance is active and schedules jobs—which involves leader election and other distributed systems challenges.

In the end, a simple service that has to copy a few files once a day may end up requiring multiple nodes, a distributed leader election mechanism, and more. Kubernetes CronJob implementation solves all that by allowing scheduling of Job resources using the well-known cron format and letting developers focus only on implementing the work to be performed rather than the temporal scheduling aspect.

Solution

In Chapter 4, "Batch Job", we saw the use cases and the capabilities of Kubernetes Jobs. All of that applies to this chapter as well since the CronJob primitive builds on top of a Job. A CronJob instance is similar to one line of a Unix crontab (cron table) and manages the temporal aspects of a Job. It allows the execution of a Job periodically at a specified point in time. See Example 5-1 for a sample definition.

Example 5-1. A CronJob resource

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: random-generator
spec:
  # Every three minutes
  schedule: "*/3 * * * *"        ❶
  jobTemplate:
    spec:
      template:                  ❷
        spec:
          containers:
          - image: k8spatterns/random-generator:1.0
          name: random-generator
          command: [ "java", "-cp", "/", "RandomRunner", "/numbers.txt", "10000" ]
        restartPolicy: OnFailure
```

❶ Cron specification for running every three minutes.

❷ Job template that uses the same specification as a regular Job.

Apart from the Job spec, a CronJob has additional fields to define its temporal aspects:

`.spec.schedule`
: Crontab entry for specifying the Job's schedule (e.g., `0 * * * *` for running every hour). You can also use shortcuts like `@daily` or `@hourly`. Please refer to the CronJob documentation for all available options.

`.spec.startingDeadlineSeconds`
: Deadline (in seconds) for starting the Job if it misses its scheduled time. In some use cases, a task is valid only if it executed within a certain timeframe and is useless when executed late. For example, if a Job is not executed in the desired time because of a lack of compute resources or other missing dependencies, it might be better to skip an execution because the data it is supposed to process is obsolete already. Don't use a deadline smaller than ten seconds since Kubernetes only will check the Job status only every ten seconds.

`.spec.concurrencyPolicy`
: Specifies how to manage concurrent executions of Jobs created by the same CronJob. The default behavior `Allow` creates new Job instances even if the previous Jobs have not completed yet. If that is not the desired behavior, it is possible to skip the next run if the current one has not completed yet with `Forbid` or to cancel the currently running Job and start a new one with `Replace`.

`.spec.suspend`
: Field suspending all subsequent executions without affecting already started executions. Note that this is different from a Job's `.spec.suspend` as the start of new Jobs will be suspended, not the Jobs themselves.

`.spec.successfulJobsHistoryLimit` *and* `.spec.failedJobsHistoryLimit`
: Fields specifying how many completed and failed Jobs should be kept for auditing purposes.

CronJob is a very specialized primitive, and it applies only when a unit of work has a temporal dimension. Even if CronJob is not a general-purpose primitive, it is an excellent example of how Kubernetes capabilities build on top of each other and support noncloud-native use cases as well.

Discussion

As you can see, a CronJob is a pretty simple primitive that adds clustered, cron-like behavior to the existing Job definition. But when it is combined with other primitives such as Pods, container resource isolation, and other Kubernetes features such as those described in Chapter 6, or Chapter 2, "Health Probe", it ends up being a very powerful Job scheduling system. This enables developers to focus solely on the problem domain and implement a containerized application that is responsible only for the business logic to be performed. The scheduling is performed outside the application, as part of the platform with all of its added benefits such as high availability, resiliency, capacity, and policy-driven Pod placement. Of course, similar to the Job implementation, when implementing a CronJob container, your application has to consider all corner and failure cases of duplicate runs, no runs, parallel runs, or cancellations.

More Information

- Periodic Job Example
- Cron Jobs
- Cron
- Crontab specification
- Cron Expression Generator

About the Authors

Bilgin Ibryam (@bibryam) is a principal architect at Red Hat, a member of Apache Software Foundation, and committer to multiple open source projects. He is a regular blogger, open source evangelist, blockchain enthusiast, speaker, and the author of *Camel Design Patterns*. He has over a decade of experience building and designing highly scalable, resilient, distributed systems.

In his day-to-day job, Bilgin enjoys mentoring, coding, and leading enterprise companies to be successful with building open source solutions. His current work focuses on application integration, enterprise blockchains, distributed system design, microservices, and cloud-native applications in general.

Dr. Roland Huß (@ro14nd) is a principal software engineer at Red Hat who worked as tech lead on Fuse Online and landed recently in the serverless team for coding on Knative. He has been developing in Java for over 20 years now and recently found another love with Golang. However, he has never forgotten his roots as a system administrator. Roland is an active open source contributor, lead developer of the JMX-HTTP bridge Jolokia and some popular Java build tools for creating container images and deploying them on Kubernetes and OpenShift. Besides coding, he enjoys spreading the word about his work at conferences and through his writing.