Containers

Containers are the building blocks for Kubernetes-based cloud native applications. If we make a comparison with OOP and Java, container images are like classes, and containers are like objects. The same way we can extend classes to reuse and alter behavior, we can have container images that extend other container images to reuse and alter behavior. The same way we can do object composition and use functionality, we can do container compositions by putting containers into a Pod and using collaborating containers.

If we continue the comparison, Kubernetes would be like the JVM but spread over multiple hosts, and it would be responsible for running and managing the containers. Init containers would be something like object constructors; DaemonSets would be similar to daemon threads that run in the background (like the Java Garbage Collector, for example). A Pod would be something similar to an Inversion of Control (IoC) context (Spring Framework, for example), where multiple running objects share a managed lifecycle and can access one another directly.

The parallel doesn't go much further, but the point is that containers play a fundamental role in Kubernetes, and creating modularized, reusable, single-purpose container images is fundamental to the long-term success of any project and even the containers' ecosystem as a whole. Apart from the technical characteristics of a container image that provide packaging and isolation, what does a container represent, and what is its purpose in the context of a distributed application? Here are a few suggestions on how to look at containers:

- A container image is the unit of functionality that addresses a single concern.
- A container image is owned by one team and has its own release cycle.
- A container image is self-contained and defines and carries its runtime dependencies.
- A container image is immutable, and once it is built, it does not change; it is configured.
- A container image defines its resource requirements and external dependencies.
- A container image has well-defined APIs to expose its functionality.
- A container typically runs as a single Unix process.
- A container is disposable and safe to scale up or down at any moment.

In addition to all these characteristics, a proper container image is modular. It is parameterized and created for reuse in the different environments in which it is going to run. Having small, modular, and reusable container images leads to the creation of more specialized and stable container images in the long term, similar to a great reusable library in the programming language world.

Pods

Looking at the characteristics of containers, we can see that they are a perfect match for implementing the microservices principles. A container image provides a single unit of functionality, belongs to a single team, has an independent release cycle, and provides deployment and runtime isolation. Most of the time, one microservice corresponds to one container image.

However, most cloud native platforms offer another primitive for managing the lifecycle of a group of containers—in Kubernetes, it is called a Pod. A *Pod* is an atomic unit of scheduling, deployment, and runtime isolation for a group of containers. All containers in a Pod are always scheduled to the same host, are deployed and scaled together, and can also share filesystem, networking, and process namespaces. This joint lifecycle allows the containers in a Pod to interact with one another over the filesystem or through networking via localhost or host interprocess communication mechanisms if desired (for performance reasons, for example). A Pod also represents a security boundary for an application. While it is possible to have containers with varying security parameters in the same Pod, typically all containers would have the same access level, network segmentation, and identity.

As you can see in Figure 1-2, at development and build time, a microservice corresponds to a container image that one team develops and releases. But at runtime, a microservice is represented by a Pod, which is the unit of deployment, placement, and scaling. The only way to run a container—whether for scale or migration—is through the Pod abstraction. Sometimes a Pod contains more than one container. In one such example, a containerized microservice uses a helper container at runtime, as Chapter 16, "Sidecar", demonstrates.

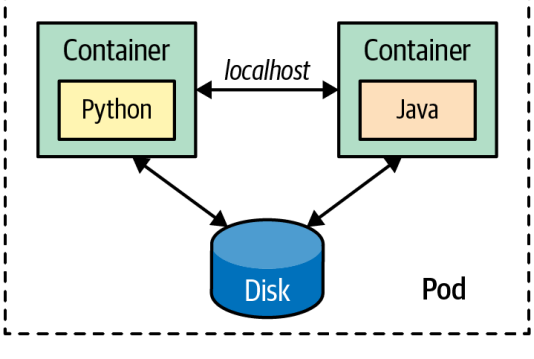

Figure 1-2. A Pod as the deployment and management unit

Containers, Pods, and their unique characteristics offer a new set of patterns and principles for designing microservices-based applications. We saw some of the characteristics of well-designed containers; now let's look at some characteristics of a Pod:

- A Pod is the atomic unit of scheduling. That means the scheduler tries to find a host that satisfies the requirements of all containers that belong to the Pod (we cover some specifics around init containers in Chapter 15, "Init Container"). If you create a Pod with many containers, the scheduler needs to find a host that has enough resources to satisfy all container demands combined. This scheduling process is described in Chapter 6, "Automated Placement".
- A Pod ensures colocation of containers. Thanks to the colocation, containers in the same Pod have additional means to interact with one another. The most common ways of communicating include using a shared local filesystem for exchanging data, using the localhost network interface, or using some host interprocess communication (IPC) mechanism for high-performance interactions.
- A Pod has an IP address, name, and port range that are shared by all containers belonging to it. That means containers in the same Pod have to be carefully configured to avoid port clashes, in the same way that parallel, running Unix processes have to take care when sharing the networking space on a host.

A Pod is the atom of Kubernetes where your application lives, but you don't access Pods directly—that is where Services enter the scene.

Services

Pods are ephemeral. They come and go at any time for all sorts of reasons (e.g., scaling up and down, failing container health checks, node migrations). A Pod IP address is known only after it is scheduled and started on a node. A Pod can be rescheduled to a different node if the existing node it is running on is no longer healthy. This means the Pod's network address may change over the life of an application, and there is a need for another primitive for discovery and load balancing.

That's where the Kubernetes Services come into play. The Service is another simple but powerful Kubernetes abstraction that binds the Service name to an IP address and port number permanently. So a Service represents a named entry point for accessing an application. In the most common scenario, the Service serves as the entry point for a set of Pods, but that might not always be the case. The Service is a generic primitive, and it may also point to functionality provided outside the Kubernetes cluster. As such, the Service primitive can be used for Service discovery and load balancing, and it allows altering implementations and scaling without affecting Service consumers. We explain Services in detail in Chapter 13, "Service Discovery".

Labels

We have seen that a microservice is a container image at build time but is represented by a Pod at runtime. So what is an application that consists of multiple microservices? Here, Kubernetes offers two more primitives that can help you define the concept of an application: labels and namespaces.

Before microservices, an application corresponded to a single deployment unit with a single versioning scheme and release cycle. There was a single file for an application in a *.war*, *.ear*, or some other packaging format. But then, applications were split into microservices, which are independently developed, released, run, restarted, or scaled. With microservices, the notion of an application diminishes, and there are no key artifacts or activities that we have to perform at the application level. But if you still need a way to indicate that some independent services belong to an application, *labels* can be used. Let's imagine that we have split one monolithic application into three microservices and another one into two microservices.

We now have five Pod definitions (and maybe many more Pod instances) that are independent of the development and runtime points of view. However, we may still need to indicate that the first three Pods represent an application and the other two Pods represent another application. Even the Pods may be independent, to provide a business value, but they may depend on one another. For example, one Pod may contain the containers responsible for the frontend, and the other two Pods are responsible for providing the backend functionality. If either of these Pods is down, the application is useless from a business point of view. Using label selectors gives us the ability to query and identify a set of Pods and manage it as one logical unit. Figure 1-3 shows how you can use labels to group the parts of a distributed application into specific subsystems.

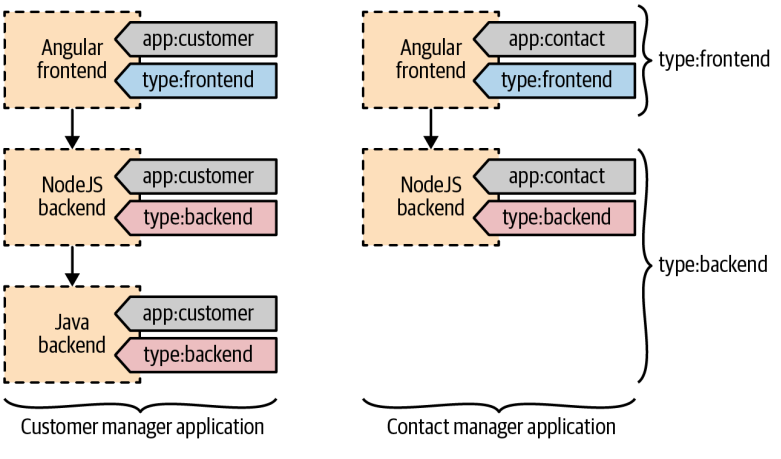

Figure 1-3. Labels used as an application identity for Pods

Here are a few examples where labels can be useful:

- Labels are used by ReplicaSets to keep some instances of a specific Pod running. That means every Pod definition needs to have a unique combination of labels used for scheduling.
- Labels are also heavily used by the scheduler. The scheduler uses labels for colocating or spreading Pods to the nodes that satisfy the Pods' requirements.
- A label can indicate a logical grouping of a set of Pods and give an application identity to them.
- In addition to the preceding typical use cases, labels can be used to store metadata. It may be difficult to predict what a label could be used for, but it is best to have enough labels to describe all important aspects of the Pods. For example, having labels to indicate the logical group of an application, the business characteristics and criticality, the specific runtime platform dependencies such as hardware architecture, or location preferences are all useful.

Later, these labels can be used by the scheduler for more fine-grained scheduling, or the same labels can be used from the command line for managing the matching Pods at scale. However, you should not go overboard and add too many labels in advance. You can always add them later if needed. Removing labels is much riskier as there is no straightforward way of finding out what a label is used for and what unintended effect such an action may cause.

Annotations

Another primitive very similar to labels is the *annotation*. Like labels, annotations are organized as a map, but they are intended for specifying nonsearchable metadata and for machine usage rather than human.

The information on the annotations is not intended for querying and matching objects. Instead, it is intended for attaching additional metadata to objects from various tools and libraries we want to use. Some examples of using annotations include build IDs, release IDs, image information, timestamps, Git branch names, pull request numbers, image hashes, registry addresses, author names, tooling information, and more. So while labels are used primarily for query matching and performing actions on the matching resources, annotations are used to attach metadata that can be consumed by a machine.

Namespaces

Another primitive that can also help manage a group of resources is the Kubernetes *namespace*. As we have described, a namespace may seem similar to a label, but in reality, it is a very different primitive with different characteristics and purposes.

Kubernetes namespaces allow you to divide a Kubernetes cluster (which is usually spread across multiple hosts) into a logical pool of resources. Namespaces provide scopes for Kubernetes resources and a mechanism to apply authorizations and other policies to a subsection of the cluster. The most common use case of namespaces is representing different software environments such as development, testing, integration testing, or production. Namespaces can also be used to achieve multitenancy and provide isolation for team workspaces, projects, and even specific applications. But ultimately, for a greater isolation of certain environments, namespaces are not enough, and having separate clusters is common. Typically, there is one nonproduction Kubernetes cluster used for some environments (development, testing, and integration testing) and another production Kubernetes cluster to represent performance testing and production environments.

Let's look at some of the characteristics of namespaces and how they can help us in different scenarios:

- A namespace is managed as a Kubernetes resource.
- A namespace provides scope for resources such as containers, Pods, Services, or ReplicaSets. The names of resources need to be unique within a namespace but not across them.
- By default, namespaces provide scope for resources, but nothing isolates those resources and prevents access from one resource to another. For example, a Pod from a development namespace can access another Pod from a production namespace as long as the Pod IP address is known. "Network isolation across namespaces for creating a lightweight multitenancy solution is described in Chapter 24, "Network Segmentation".
- Some other resources, such as namespaces, nodes, and PersistentVolumes, do not belong to namespaces and should have unique cluster-wide names.
- Each Kubernetes Service belongs to a namespace and gets a corresponding Domain Name Service (DNS) record that has the namespace in the form of `<service-name>.<namespace-name>.svc.cluster.local`. So the namespace name is in the URL of every Service belonging to the given namespace. That's one reason it is vital to name namespaces wisely.
- ResourceQuotas provide constraints that limit the aggregated resource consumption per namespace. With ResourceQuotas, a cluster administrator can control the number of objects per type that are allowed in a namespace. For example, a

developer namespace may allow only five ConfigMaps, five Secrets, five Services, five ReplicaSets, five PersistentVolumeClaims, and ten Pods.

- ResourceQuotas can also limit the total sum of computing resources we can request in a given namespace. For example, in a cluster with a capacity of 32 GB RAM and 16 cores, it is possible to allocate 16 GB RAM and 8 cores for the production namespace, 8 GB RAM and 4 cores for the staging environment, 4 GB RAM and 2 cores for development, and the same amount for testing namespaces. The ability to impose resource constraints decoupled from the shape and the limits of the underlying infrastructure is invaluable.

Discussion

We've only briefly covered a few of the main Kubernetes concepts we use in this book. However, there are more primitives used by developers on a day-by-day basis. For example, if you create a containerized service, there are plenty of Kubernetes abstractions you can use to reap all the benefits of Kubernetes. Keep in mind, these are only a few of the objects used by application developers to integrate a containerized service into Kubernetes. There are plenty of other concepts used primarily by cluster administrators for managing Kubernetes. Figure 1-4 gives an overview of the main Kubernetes resources that are useful for developers.

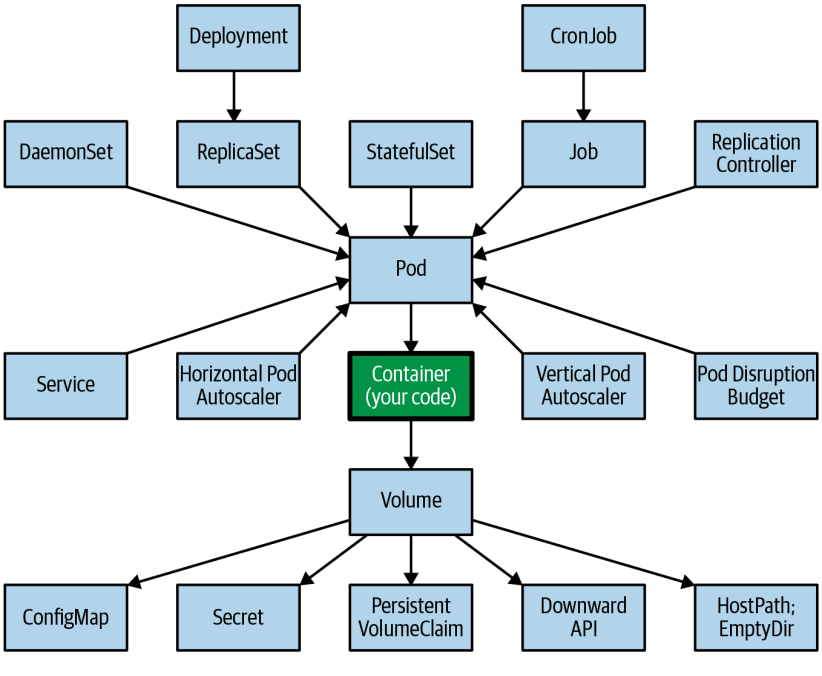

Figure 1-4. Kubernetes concepts for developers

With time, these new primitives give birth to new ways of solving problems, and some of these repetitive solutions become patterns. Throughout this book, rather than describing each Kubernetes resource in detail, we will focus on concepts that are proven as patterns.

More Information

- The Twelve-Factor App
- CNCF Cloud Native Definition v1.0
- Hexagonal Architecture
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*
- Best Practices for Writing Dockerfiles
- Principles of Container-Based Application Design
- General Container Image Guidelines

PART I
Foundational Patterns

Foundational patterns describe a number of fundamental principles that containerized applications must comply with in order to become good cloud-native citizens. Adhering to these principles will help ensure that your applications are suitable for automation in cloud-native platforms such as Kubernetes.

The patterns described in the following chapters represent the foundational building blocks of distributed container-based Kubernetes-native applications:

- Chapter 2, "Predictable Demands", explains why every container should declare its resource requirements and stay confined to the indicated resource boundaries.
- Chapter 3, "Declarative Deployment", describes the different application deployment strategies that can be expressed in a declarative way.
- Chapter 4, "Health Probe", dictates that every container should implement specific APIs to help the platform observe and maintain the application healthily.
- Chapter 5, "Managed Lifecycle", explains why a container should have a way to read the events coming from the platform and conform by reacting to those events.
- Chapter 6, "Automated Placement", introduces the Kubernetes scheduling algorithm and the ways to influence the placement decisions from the outside.

CHAPTER 2

Predictable Demands

The foundation of successful application deployment, management, and coexistence on a shared cloud environment is dependent on identifying and declaring the application resource requirements and runtime dependencies. This *Predictable Demands* pattern indicates how you should declare application requirements, whether they are hard runtime dependencies or resource requirements. Declaring your requirements is essential for Kubernetes to find the right place for your application within the cluster.

Problem

Kubernetes can manage applications written in different programming languages as long as the application can be run in a container. However, different languages have different resource requirements. Typically, a compiled language runs faster and often requires less memory compared to just-in-time runtimes or interpreted languages. Considering that many modern programming languages in the same category have similar resource requirements, from a resource consumption point of view, more important aspects are the domain, the business logic of an application, and the actual implementation details.

Besides resource requirements, application runtimes also have dependencies on platform-managed capabilities like data storage or application configuration.

Solution

Knowing the runtime requirements for a container is important mainly for two reasons. First, with all the runtime dependencies defined and resource demands envisaged, Kubernetes can make intelligent decisions about where to place a container on the cluster for the most efficient hardware utilization. In an environment with shared resources among a large number of processes with different priorities, the only way to

ensure a successful coexistence is to know the demands of every process in advance. However, intelligent placement is only one side of the coin.

Container resource profiles are also essential for capacity planning. Based on the particular service demands and the total number of services, we can do some capacity planning for different environments and come up with the most cost-effective host profiles to satisfy the entire cluster demand. Service resource profiles and capacity planning go hand in hand for successful cluster management in the long term.

Before diving into resource profiles, let's look at declaring runtime dependencies.

Runtime Dependencies

One of the most common runtime dependencies is file storage for saving application state. Container filesystems are ephemeral and are lost when a container is shut down. Kubernetes offers volume as a Pod-level storage utility that survives container restarts.

The most straightforward type of volume is `emptyDir`, which lives as long as the Pod lives. When the Pod is removed, its content is also lost. The volume needs to be backed by another kind of storage mechanism to survive Pod restarts. If your application needs to read or write files to such long-lived storage, you must declare that dependency explicitly in the container definition using volumes, as shown in Example 2-1.

Example 2-1. Dependency on a PersistentVolume

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - mountPath: "/logs"
      name: log-volume
  volumes:
  - name: log-volume
    persistentVolumeClaim:  ❶
      claimName: random-generator-log
```

❶ Dependency of a PersistentVolumeClaim (PVC) to be present and bound.

The scheduler evaluates the kind of volume a Pod requires, which affects where the Pod gets placed. If the Pod needs a volume that is not provided by any node on

the cluster, the Pod is not scheduled at all. Volumes are an example of a runtime dependency that affects what kind of infrastructure a Pod can run and whether the Pod can be scheduled at all.

A similar dependency happens when you ask Kubernetes to expose a container port on a specific port on the host system through `hostPort`. The usage of a `hostPort` creates another runtime dependency on the nodes and limits where a Pod can be scheduled. `hostPort` reserves the port on each node in the cluster and is limited to a maximum of one Pod scheduled per node. Because of port conflicts, you can scale to as many Pods as there are nodes in the Kubernetes cluster.

Configurations are another type of dependency. Almost every application needs some configuration information, and the recommended solution offered by Kubernetes is through ConfigMaps. Your services need to have a strategy for consuming settings—either through environment variables or the filesystem. In either case, this introduces a runtime dependency of your container to the named ConfigMaps. If not all of the expected ConfigMaps are created, the containers are scheduled on a node, but they do not start up.

Similar to ConfigMaps, Secrets offer a slightly more secure way of distributing environment-specific configurations to a container. The way to consume a Secret is the same as it is for ConfigMaps, and using a Secret introduces the same kind of dependency from a container to a namespace.

ConfigMaps and Secrets are explained in more detail in Chapter 20, "Configuration Resource", and Example 2-2 shows how these resources are used as runtime dependencies.

Example 2-2. Dependency on a ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef:  ❶
          name: random-generator-config
          key: pattern
```

❶ Mandatory dependency on the ConfigMap `random-generator-config`.

While the creation of ConfigMap and Secret objects are simple deployment tasks we have to perform, cluster nodes provide storage and port numbers. Some of these dependencies limit where a Pod gets scheduled (if anywhere at all), and other dependencies may prevent the Pod from starting up. When designing your containerized applications with such dependencies, always consider the runtime constraints they will create later.

Resource Profiles

Specifying container dependencies such as ConfigMap, Secret, and volumes is straightforward. We need some more thinking and experimentation for figuring out the resource requirements of a container. Compute resources in the context of Kubernetes are defined as something that can be requested by, allocated to, and consumed from a container. The resources are categorized as *compressible* (i.e., can be throttled, such as CPU or network bandwidth) and *incompressible* (i.e., cannot be throttled, such as memory).

Making the distinction between compressible and incompressible resources is important. If your containers consume too many compressible resources such as CPU, they are throttled, but if they use too many incompressible resources (such as memory), they are killed (as there is no other way to ask an application to release allocated memory).

Based on the nature and the implementation details of your application, you have to specify the minimum amount of resources that are needed (called `requests`) and the maximum amount it can grow up to (the `limits`). Every container definition can specify the amount of CPU and memory it needs in the form of a request and limit. At a high level, the concept of `requests`/`limits` is similar to soft/hard limits. For example, similarly, we define heap size for a Java application by using the `-Xms` and `-Xmx` command-line options.

The `requests` amount (but not `limits`) is used by the scheduler when placing Pods to nodes. For a given Pod, the scheduler considers only nodes that still have enough capacity to accommodate the Pod and all of its containers by summing up the requested resource amounts. In that sense, the `requests` field of each container affects where a Pod can be scheduled or not. Example 2-3 shows how such limits are specified for a Pod.

Example 2-3. Resource limits

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    resources:
      requests:          ❶
        cpu: 100m
        memory: 200Mi
      limits:            ❷
        memory: 200Mi
```

❶ Initial resource request for CPU and memory.

❷ Upper limit until we want our application to grow at max. We don't specify CPU limits by intention.

The following types of resources can be used as keys in the `requests` and `limits` specification:

`memory`
: This type is for the heap memory demands of your application, including volumes of type `emptyDir` with the configuration `medium: Memory`. Memory resources are incompressible, so containers that exceed their configured memory limit will trigger the Pod to be evicted; i.e., it gets deleted and recreated potentially on a different node.

`cpu`
: The `cpu` type is used to specify the range of needed CPU cycles for your application. However, it is a compressible resource, which means that in an overcommit situation for a node, all assigned CPU slots of all running containers are throttled relative to their specified requests. Therefore, it is highly recommended that you set `requests` for the CPU resource but *no* `limits` so that they can benefit from all excess CPU resources that otherwise would be wasted.

`ephemeral-storage`
: Every node has some filesystem space dedicated for ephemeral storage that holds logs and writable container layers. `emptyDir` volumes that are not stored in a memory filesystem also use ephemeral storage. With this request and limit type, you can specify the application's minimal and maximal needs. `ephemeral-storage` resources are not compressible and will cause a Pod to be evicted from the node if it uses more storage than specified in its `limit`.

`hugepage-<size>`
: *Huge pages* are large, contiguous pre-allocated pages of memory that can be mounted as volumes. Depending on your Kubernetes node configuration, several sizes of huge pages are available, like 2 MB and 1 GB pages. You can specify a request and limit for how many of a certain type of huge pages you want to consume (e.g., `hugepages-1Gi: 2Gi` for requesting two 1 GB huge pages). Huge pages can't be overcommitted, so the request and limit must be the same.

Depending on whether you specify the `requests`, the `limits`, or both, the platform offers three types of Quality of Service (QoS):

Best-Effort
: Pods that do not have any requests and limits set for its containers have a QoS of *Best-Effort*. Such a *Best-Effort* Pod is considered the lowest priority and is most likely killed first when the node where the Pod is placed runs out of incompressible resources.

Burstable
: A Pod that defines an unequal amount for `requests` and `limits` values (and `limits` is larger than `requests`, as expected) are tagged as *Burstable*. Such a Pod has minimal resource guarantees but is also willing to consume more resources up to its `limit` when available. When the node is under incompressible resource pressure, these Pods are likely to be killed if no *Best-Effort* Pods remain.

Guaranteed
: A Pod that has an equal amount of `request` and `limit` resources belongs to the *Guaranteed* QoS category. These are the highest-priority Pods and are guaranteed not to be killed before *Best-Effort* and *Burstable* Pods. This QoS mode is the best option for your application's memory resources, as it entails the least surprise and avoids out-of-memory triggered evictions.

So the resource characteristics you define or omit for the containers have a direct impact on its QoS and define the relative importance of the Pod in the event of resource starvation. Define your Pod resource requirements with this consequence in mind.

Recommendations for CPU and Memory Resources

While you have many options for declaring the memory and CPU needs of your applications, we and others recommend the following rules:

- For memory, always set `requests` equal to `limits`.
- For CPU, set `requests` but no `limits`.

See the blog post "For the Love of God, Stop Using CPU Limits on Kubernetes" for a more in-depth explanation of why you should not use `limits` for the CPU, and see the blog post "What Everyone Should Know About Kubernetes Memory Limits" for more details about the recommended memory settings.

Pod Priority

We explained how container resource declarations also define Pods' QoS and affect the order in which the Kubelet kills the container in a Pod in case of resource starvation. Two other related concepts are Pod priority and preemption. *Pod priority* allows you to indicate the importance of a Pod relative to other Pods, which affects the order in which Pods are scheduled. Let's see that in action in Example 2-4.

Example 2-4. Pod priority

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority  ❶
value: 1000            ❷
globalDefault: false   ❸
description: This is a very high-priority Pod class
---
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    env: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  priorityClassName: high-priority  ❹
```

❶ The name of the priority class object.

❷ The priority value of the object.

❸ `globalDefault` set to `true` is used for Pods that do not specify a `priorityClassName`. Only one PriorityClass can have `globalDefault` set to `true`.

❹ The priority class to use with this Pod, as defined in PriorityClass resource.

We created a PriorityClass, a non-namespaced object for defining an integer-based priority. Our PriorityClass is named `high-priority` and has a priority of 1,000.

Now we can assign this priority to Pods by its name as `priorityClassName: high-priority`. PriorityClass is a mechanism for indicating the importance of Pods relative to one another, where the higher value indicates more important Pods.

Pod priority affects the order in which the scheduler places Pods on nodes. First, the priority admission controller uses the `priorityClassName` field to populate the priority value for new Pods. When multiple Pods are waiting to be placed, the scheduler sorts the queue of pending Pods by highest priority first. Any pending Pod is picked before any other pending Pod with lower priority in the scheduling queue, and if there are no constraints preventing it from scheduling, the Pod gets scheduled.

Here comes the critical part. If there are no nodes with enough capacity to place a Pod, the scheduler can preempt (remove) lower-priority Pods from nodes to free up resources and place Pods with higher priority. As a result, the higher-priority Pod might be scheduled sooner than Pods with a lower priority if all other scheduling requirements are met. This algorithm effectively enables cluster administrators to control which Pods are more critical workloads and place them first by allowing the scheduler to evict Pods with lower priority to make room on a worker node for higher-priority Pods. If a Pod cannot be scheduled, the scheduler continues with the placement of other lower-priority Pods.

Suppose you want your Pod to be scheduled with a particular priority but don't want to evict any existing Pods. In that case, you can mark a PriorityClass with the field `preemptionPolicy: Never`. Pods assigned to this priority class will not trigger any eviction of running Pods but will still get scheduled according to their priority value.

Pod QoS (discussed previously) and Pod priority are two orthogonal features that are not connected and have only a little overlap. QoS is used primarily by the Kubelet to preserve node stability when available compute resources are low. The Kubelet first considers QoS and then the PriorityClass of Pods before eviction. On the other hand, the scheduler eviction logic ignores the QoS of Pods entirely when choosing preemption targets. The scheduler attempts to pick a set of Pods with the lowest priority possible that satisfies the needs of higher-priority Pods waiting to be placed.

When Pods have a priority specified, it can have an undesired effect on other Pods that are evicted. For example, while a Pod's graceful termination policies are respected, the PodDisruptionBudget as discussed in Chapter 10, "Singleton Service", is not guaranteed, which could break a lower-priority clustered application that relies on a quorum of Pods.

Another concern is a malicious or uninformed user who creates Pods with the highest possible priority and evicts all other Pods. To prevent that, ResourceQuota has been extended to support PriorityClass, and higher-priority numbers are reserved for critical system-Pods that should not usually be preempted or evicted.

In conclusion, Pod priorities should be used with caution because user-specified numerical priorities that guide the scheduler and Kubelet about which Pods to place or to kill are subject to gaming by users. Any change could affect many Pods and could prevent the platform from delivering predictable service-level agreements.

Project Resources

Kubernetes is a self-service platform that enables developers to run applications as they see suitable on the designated isolated environments. However, working in a shared multitenanted platform also requires the presence of specific boundaries and control units to prevent some users from consuming all the platform's resources. One such tool is ResourceQuota, which provides constraints for limiting the aggregated resource consumption in a namespace. With ResourceQuotas, the cluster administrators can limit the total sum of computing resources (CPU, memory) and storage consumed. It can also limit the total number of objects (such as ConfigMaps, Secrets, Pods, or Services) created in a namespace. Example 2-5 shows an instance that limits the usage of certain resources. See the official Kubernetes documentation on Resource Quotas for the full list of supported resources for which you can restrict usage with ResourceQuotas.

Example 2-5. Definition of resource constraints

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: default    ❶
spec:
  hard:
    pods: 4             ❷
    limits.memory: 5Gi  ❸
```

❶ Namespace to which resource constraints are applied.

❷ Allow four active Pods in this namespace.

❸ The sum of all memory limits of all Pods in this namespace must not be more than 5 GB.

Another helpful tool in this area is LimitRange, which allows you to set resource usage limits for each type of resource. In addition to specifying the minimum and maximum permitted amounts for different resource types and the default values for these resources, it also allows you to control the ratio between the `requests` and `limits`, also known as the *overcommit level*. Example 2-6 shows a LimitRange and the possible configuration options.

Example 2-6. Definition of allowed and default resource usage limits

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
  namespace: default
spec:
  limits:
  - min:                          ❶
      memory: 250Mi
      cpu: 500m
    max:                          ❷
      memory: 2Gi
      cpu: 2
    default:                      ❸
      memory: 500Mi
      cpu: 500m
    defaultRequest:               ❹
      memory: 250Mi
      cpu: 250m
    maxLimitRequestRatio:         ❺
      memory: 2
      cpu: 4
    type: Container               ❻
```

❶ Minimum values for requests and limits.

❷ Maximum values for requests and limits.

❸ Default values for limits when no limits are specified.

❹ Default values for requests when no requests are specified.

❺ Maximum ratio limit/request, used to specify the allowed overcommit level. Here, the memory limit must not be larger than twice the memory request, and the CPU limit can be as high as four times the CPU request.

❻ Type can be `Container`, `Pod`, (for all containers combined), or `PersistentVolumeClaim` (to specify the range for a request persistent volume).

LimitRanges help control the container resource profiles so that no containers require more resources than a cluster node can provide. LimitRanges can also prevent cluster users from creating containers that consume many resources, making the nodes not allocatable for other containers. Considering that the `requests` (and not `limits`) are the primary container characteristic the scheduler uses for placing, LimitRequestRatio allows you to control the amount of difference between the `requests` and `limits` of containers. A big combined gap between `requests` and `limits` increases the chances of overcommitting on the node and may degrade application performance when many containers simultaneously require more resources than initially requested.

Keep in mind that other shared node-level resources such as process IDs (PIDs) can be exhausted before hitting any resource limits. Kubernetes allows you to reserve a number of node PIDs for the system use and ensure that they are never exhausted by user workloads. Similarly, Pod PID limits allow a cluster administrator to limit the number of processes running in a Pod. We are not reviewing these in details here as they are set as Kubelet configurations options by cluster administrators and are not used by application developers.

Capacity Planning

Considering that containers may have different resource profiles in different environments, and a varied number of instances, it is evident that capacity planning for a multipurpose environment is not straightforward. For example, for best hardware utilization, on a nonproduction cluster, you may have mainly *Best-Effort* and *Burstable* containers. In such a dynamic environment, many containers are starting up and shutting down at the same time, and even if a container gets killed by the platform during resource starvation, it is not fatal. On the production cluster, where we want things to be more stable and predictable, the containers may be mainly of the *Guaranteed* type, and some may be *Burstable*. If a container gets killed, that is most likely a sign that the capacity of the cluster should be increased.

Table 2-1 presents a few services with CPU and memory demands.

Table 2-1. Capacity planning example

Pod	CPU request	Memory request	Memory limit	Instances
A	500 m	500 Mi	500 Mi	4
B	250 m	250 Mi	1000 Mi	2
C	500 m	1000 Mi	2000 Mi	2
D	500 m	500 Mi	500 Mi	1
Total	**4000 m**	**5000 Mi**	**8500 Mi**	**9**

Of course, in a real-life scenario, the more likely reason you are using a platform such as Kubernetes is that there are many more services to manage, some of which are about to retire, and some of which are still in the design and development phase. Even if it is a continually moving target, based on a similar approach as described previously, we can calculate the total amount of resources needed for all the services per environment.

Keep in mind that in the different environments, there are different numbers of containers, and you may even need to leave some room for autoscaling, build jobs, infrastructure containers, and more. Based on this information and the infrastructure provider, you can choose the most cost-effective compute instances that provide the required resources.

Discussion

Containers are useful not only for process isolation and as a packaging format. With identified resource profiles, they are also the building blocks for successful capacity planning. Perform some early tests to discover the resource needs for each container, and use that information as a base for future capacity planning and prediction.

Kubernetes can help you here with the *Vertical Pod Autoscaler* (VPA), which monitors the resource consumption of your Pod over time and gives a recommendation for requests and limits. The VPA is described in detail in "Vertical Pod Autoscaling" on page 325.

However, more importantly, resource profiles are the way an application communicates with Kubernetes to assist in scheduling and managing decisions. If your application doesn't provide any `requests` or `limits`, all Kubernetes can do is treat your containers as opaque boxes that are dropped when the cluster gets full. So it is more or less mandatory for every application to think about and provide these resource declarations.

Now that you know how to size our applications, in Chapter 3, "Declarative Deployment", you will learn multiple strategies to install and update our applications on Kubernetes.

More Information

- Predictable Demands Example
- Configure a Pod to Use a ConfigMap
- Kubernetes Best Practices: Resource Requests and Limits
- Resource Management for Pods and Containers
- Manage HugePages

- Configure Default Memory Requests and Limits for a Namespace
- Node-Pressure Eviction
- Pod Priority and Preemption
- Configure Quality of Service for Pods
- Resource Quality of Service in Kubernetes
- Resource Quotas
- Limit Ranges
- Process ID Limits and Reservations
- For the Love of God, Stop Using CPU Limits on Kubernetes
- What Everyone Should Know About Kubernetes Memory Limits

CHAPTER 3

Declarative Deployment

The heart of the *Declarative Deployment* pattern is the 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 place the applications 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 not 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, "Predictable Demands", we saw 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 5, "Managed Lifecycle") and also provide health-check endpoints as described in Chapter 4, "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 were implemented on the client side with the `kubectl rolling-update` command. In Kubernetes 1.18, `rolling-update` was removed in favor of a `rollout` command for `kubectl`. The difference is that `kubectl rollout` manages an application update on the server side by updating the Deployment *declaration* and leaving it to Kubernetes to perform the update. The `kubectl rolling-update` command, in contrast, was *imperative*: the client `kubectl` told 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`
: Pauses a rolling update so that multiple changes can be applied to a Deployment without 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 can find usage examples for `kubectl rollout` commands 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 you to shape the update process behavior with strategies such as `RollingUpdate` (default) and `Recreate`. Example 3-1 shows the important bits for configuring a Deployment for a rolling update strategy.

Example 3-1. 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", "/tmp/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 maximum of four replicas.

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

❹ Duration in seconds of all readiness probes for a rolled-out Pod needs to be healthy until the rollout continues.

❺ Readiness probes that are very important for a rolling deployment to ensure zero downtime—don't forget them (see Chapter 4, "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 the 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 behavior is `minReadySeconds`. This field specifies the duration in seconds that the readiness probes of a Pod need to be successful until the Pod itself is considered to be available in a rollout. Increasing this value guarantees that your application Pod is successfully running for some time 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 are larger.

Figure 3-1 shows the rolling update process.

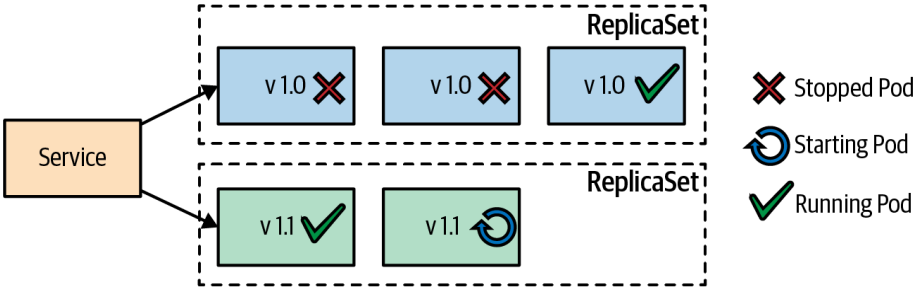

Figure 3-1. Rolling deployment

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 repository, which demonstrates the usage of these commands and shows you how to monitor or roll back an upgrade with `kubectl rollout`.

In addition to addressing the drawbacks of the imperative way of deploying services, the Deployment has 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 specifies how the deployed state should look rather than the steps necessary to get there.
- The Deployment definition is an executable object and more than just documentation. It can be 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, you can use the `Recreate` strategy, which is illustrated in Figure 3-2.

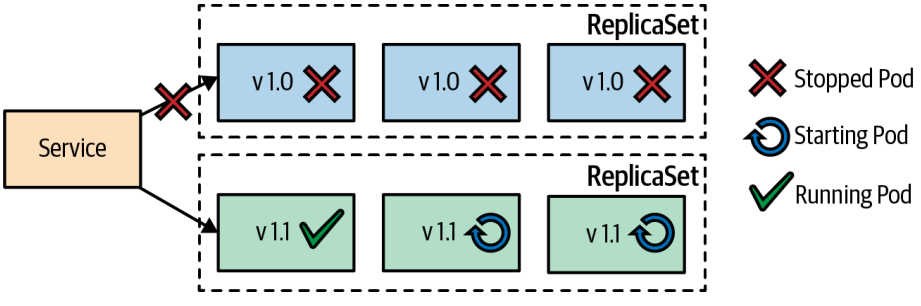

Figure 3-2. Fixed deployment using a `Recreate` strategy

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 is that downtime occurs while all containers with old versions are stopped, and no new containers are ready to handle incoming requests. On the positive side, two different versions of the containers won't be running at the same time, so service consumers can connect 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. The 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.

A Blue-Green deployment needs to be done manually if no extensions like a service mesh or Knative are 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 from the original Deployment (called *blue*) 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. You can do this in Kubernetes by updating the Service selector to match the new containers (labeled with green). As demonstrated in Figure 3-3, once the green (v1.1) containers handle all the traffic, the blue (v1.0) containers can be deleted and the resources freed for future Blue-Green deployments.

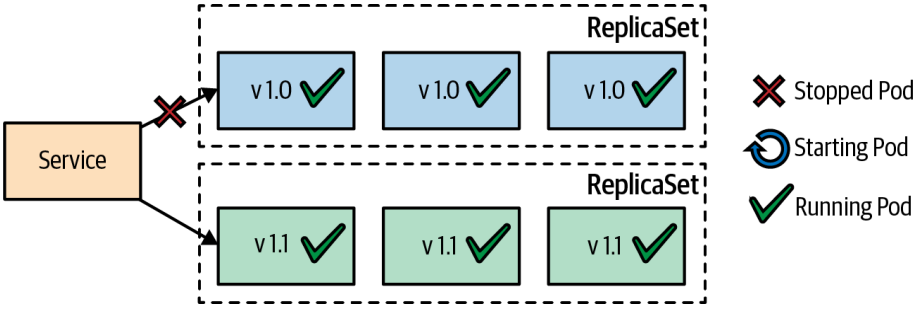

Figure 3-3. Blue-Green release

A benefit of the Blue-Green approach is that only one version of the application is serving requests at a time, 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, significant complications can occur 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're 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 3-4 shows a canary release in action.

In Kubernetes, this technique can be implemented by creating a new 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 the new ReplicaSet works as expected, we scale the new ReplicaSet up, and the old ReplicaSet down to zero. In a way, we're performing a controlled and user-tested incremental rollout.

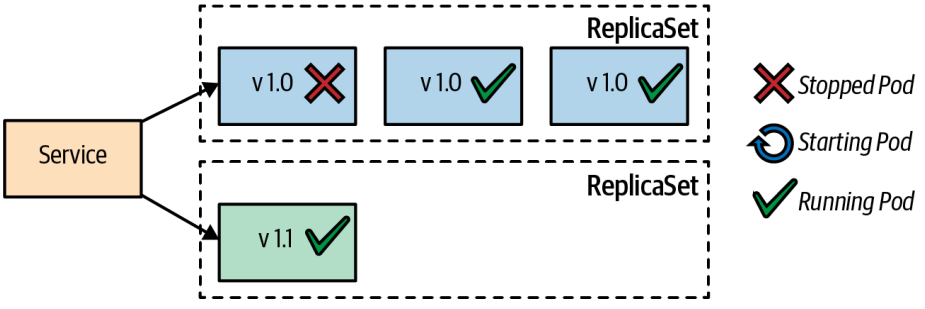

Figure 3-4. Canary release

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.

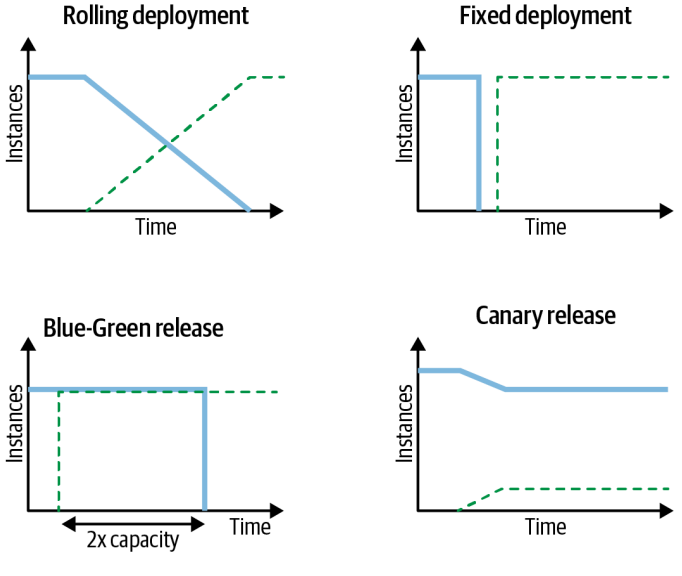

Figure 3-5. Deployment and release strategies

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.

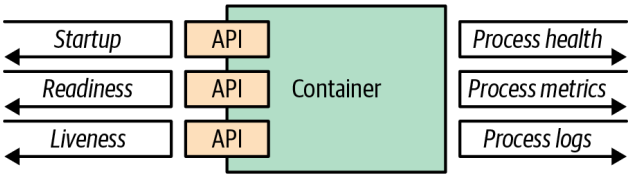

Figure 4-1. Container observability options

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.

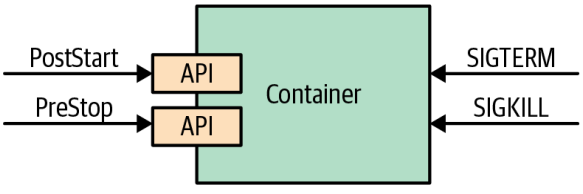

Figure 5-1. Managed container lifecycle

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.

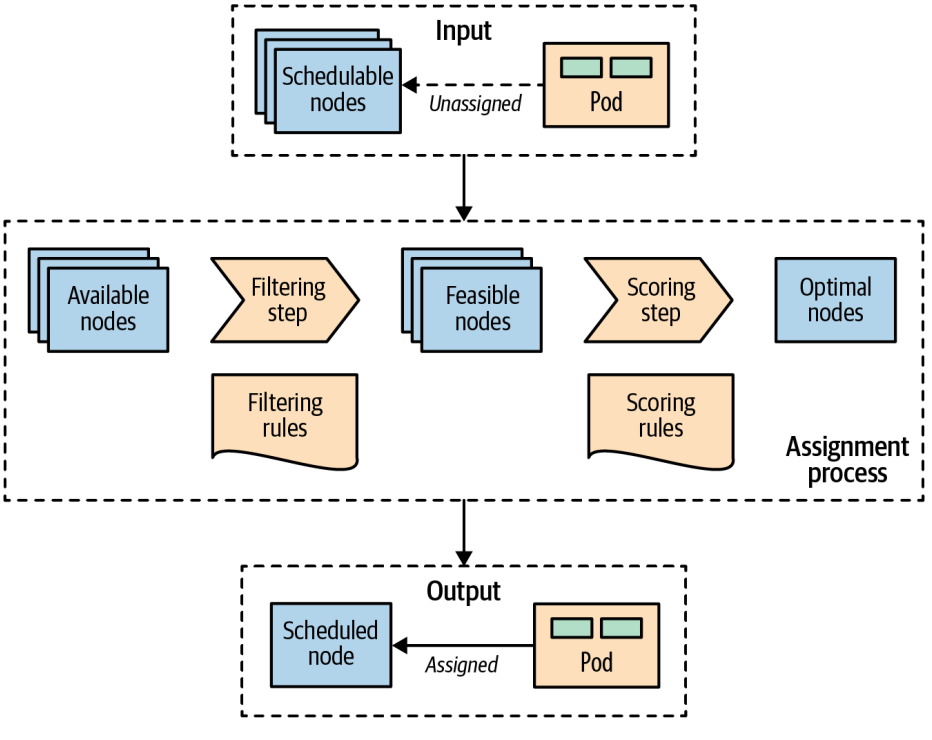

Figure 6-1. A Pod-to-node assignment process

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.

Node Affinity

Kubernetes supports many more flexible ways to configure the scheduling processes. One such feature is *node affinity*, which is a more expressive way of the node selector approach described previously that allows specifying rules as either required or preferred. *Required rules* must be met for a Pod to be scheduled to a node, whereas preferred rules only imply preference by increasing the weight for the matching nodes without making them mandatory. In addition, the node affinity feature greatly expands the types of constraints you can express by making the language more expressive with operators such as `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, or `Lt`. Example 6-4 demonstrates how node affinity is declared.

Example 6-4. Pod with node affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:    ❶
        nodeSelectorTerms:
        - matchExpressions:                              ❷
          - key: numberCores
            operator: Gt
            values: [ "3" ]
      preferredDuringSchedulingIgnoredDuringExecution:   ❸
      - weight: 1
        preference:
          matchFields:
          - key: metadata.name
            operator: NotIn
            values: [ "control-plane-node" ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Hard requirement that the node must have more than three cores (indicated by a node label) to be considered in the scheduling process. The rule is not reevaluated during execution if the conditions on the node change.

❷ Match on labels. In this example, all nodes are matched that have a label `numberCores` with a value greater than 3.

❸ Soft requirements, which is a list of selectors with weights. For every node, the sum of all weights for matching selectors is calculated, and the highest-valued node is chosen, as long as it matches the hard requirement.

Pod Affinity and Anti-Affinity

Pod affinity is a more powerful way of scheduling and should be used when `nodeSelector` is not enough. This mechanism allows you to constrain which nodes a Pod can run based on label or field matching. It doesn't allow you to express dependencies among Pods to dictate where a Pod should be placed relative to other Pods. To express how Pods should be spread to achieve high availability, or be packed and colocated together to improve latency, you can use Pod affinity and anti-affinity.

Node affinity works at node granularity, but Pod affinity is not limited to nodes and can express rules at various topology levels based on the Pods already running on a node. Using the `topologyKey` field, and the matching labels, it is possible to enforce more fine-grained rules, which combine rules on domains like node, rack, cloud provider zone, and region, as demonstrated in Example 6-5.

Example 6-5. Pod with Pod affinity

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  ❶
      - labelSelector:                                 ❷
          matchLabels:
            confidential: high
        topologyKey: security-zone                     ❸
    podAntiAffinity:                                   ❹
      preferredDuringSchedulingIgnoredDuringExecution: ❺
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchLabels:
              confidential: none
          topologyKey: kubernetes.io/hostname
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Required rules for the Pod placement concerning other Pods running on the target node.

❷ Label selector to find the Pods to be colocated with.

❸ The nodes on which Pods with labels `confidential=high` are running are supposed to carry a `security-zone` label. The Pod defined here is scheduled to a node with the same label and value.

❹ Anti-affinity rules to find nodes where a Pod would *not* be placed.

❺ Rule describing that the Pod should not (but could) be placed on any node where a Pod with the label `confidential=none` is running.

Similar to node affinity, there are hard and soft requirements for Pod affinity and anti-affinity, called `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`, respectively. Again, as with node affinity, the `IgnoredDuringExecution` suffix is in the field name, which exists for future extensibility reasons. At the moment, if the labels on the node change and affinity rules are no longer valid, the Pods continue running,[1] but in the future, runtime changes may also be taken into account.

Topology Spread Constraints

Pod affinity rules allow the placement of unlimited Pods to a single topology, whereas Pod anti-affinity disallows Pods to colocate in the same topology. Topology spread constraints give you more fine-grained control to evenly distribute Pods on your cluster and achieve better cluster utilization or high availability of applications.

Let's look at an example to understand how topology spread constraints can help. Let's suppose we have an application with two replicas and a two-node cluster. To avoid downtime and a single point of failure, we can use Pod anti-affinity rules to prevent the coexistence of the Pods on the same node and spread them into both nodes. While this setup makes sense, it will prevent you from performing rolling upgrades because the third replacement Pod cannot be placed on the existing nodes because of the Pod anti-affinity constraints. We will have to either add another node or change the Deployment strategy from rolling to recreate. Topology spread constraints would be a better solution in this situation as they allow you to tolerate some degree of uneven Pod distribution when the cluster is running out of resources. Example 6-6 allows the placement of the third rolling deployment Pod on one of the two nodes because it allows imbalances—i.e., a skew of one Pod.

1 However, if node labels change and allow for unscheduled Pods to match their node affinity selector, these Pods are scheduled on this node.

Example 6-6. Pod with topology spread constraints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: bar
spec:
  topologySpreadConstraints:                    ❶
  - maxSkew: 1                                  ❷
    topologyKey: topology.kubernetes.io/zone    ❸
    whenUnsatisfiable: DoNotSchedule            ❹
    labelSelector:                              ❺
      matchLabels:
        app: bar
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ Topology spread constraints are defined in the `topologySpreadConstraints` field of the Pod spec.

❷ `maxSkew` defines the maximum degree to which Pods can be unevenly distributed in the topology.

❸ A topology domain is a logical unit of your infrastructure. And a `topologyKey` is the key of the Node label where identical values are considered to be in the same topology.

❹ The `whenUnsatisfiable` field defines what action should be taken when `maxSkew` can't be satisfied. `DoNotSchedule` is a hard constraint preventing the scheduling of Pods, whereas `ScheduleAnyway` is a soft constraint that gives scheduling priority to nodes that reduce cluster imbalance.

❺ `labelSelector` Pods that match this selector are grouped together and counted when spreading them to satisfy the constraint.

Topology spread constraints is a feature that is still evolving at the time of this writing. Built-in cluster-level topology spread constraints allow certain imbalances based on default Kubernetes labels and give you the ability to honor or ignore node affinity and taint policies.

Taints and Tolerations

A more advanced feature that controls where Pods can be scheduled and allowed to run is based on taints and tolerations. While node affinity is a property of Pods that allows them to choose nodes, taints and tolerations are the opposite. They allow the nodes to control which Pods should or should not be scheduled on them. A *taint* is a characteristic of the node, and when it is present, it prevents Pods from scheduling onto the node unless the Pod has toleration for the taint. In that sense, taints and tolerations can be considered an *opt-in* to allow scheduling on nodes that by default are not available for scheduling, whereas affinity rules are an *opt-out* by explicitly selecting on which nodes to run and thus exclude all the nonselected nodes.

A taint is added to a node by using `kubectl`: `kubectl taint nodes control-plane-node node-role.kubernetes.io/control-plane="true":NoSchedule`, which has the effect shown in Example 6-7. A matching toleration is added to a Pod as shown in Example 6-8. Notice that the values for `key` and `effect` in the `taints` section of Example 6-7 and the `tolerations` section in Example 6-8 are the same.

Example 6-7. Tainted node

```
apiVersion: v1
kind: Node
metadata:
  name: control-plane-node
spec:
  taints:                                    ❶
  - effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
    value: "true"
```

❶ Mark this node as unschedulable except when a Pod tolerates this taint.

Example 6-8. Pod tolerating node taints

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  tolerations:
  - key: node-role.kubernetes.io/control-plane ❶
    operator: Exists
    effect: NoSchedule                         ❷
```

❶ Tolerate (i.e., consider for scheduling) nodes, which have a taint with key `node-role.kubernetes.io/control-plane`. On production clusters, this taint is set on the control plane node to prevent scheduling of Pods on this node. A toleration like this allows this Pod to be installed on the control plane node nevertheless.

❷ Tolerate only when the taint specifies a `NoSchedule` effect. This field can be empty here, in which case the toleration applies to every effect.

There are hard taints that prevent scheduling on a node (`effect=NoSchedule`), soft taints that try to avoid scheduling on a node (`effect=PreferNoSchedule`), and taints that can evict already-running Pods from a node (`effect=NoExecute`).

Taints and tolerations allow for complex use cases like having dedicated nodes for an exclusive set of Pods, or force eviction of Pods from problematic nodes by tainting those nodes.

You can influence the placement based on the application's high availability and performance needs, but try not to limit the scheduler too much and back yourself into a corner where no more Pods can be scheduled and there are too many stranded resources. For example, if your containers' resource requirements are too coarse-grained, or nodes are too small, you may end up with stranded resources in nodes that are not utilized.

In Figure 6-2, we can see node A has 4 GB of memory that cannot be utilized as there is no CPU left to place other containers. Creating containers with smaller resource requirements may help improve this situation. Another solution is to use the Kubernetes *descheduler*, which helps defragment nodes and improve their utilization.

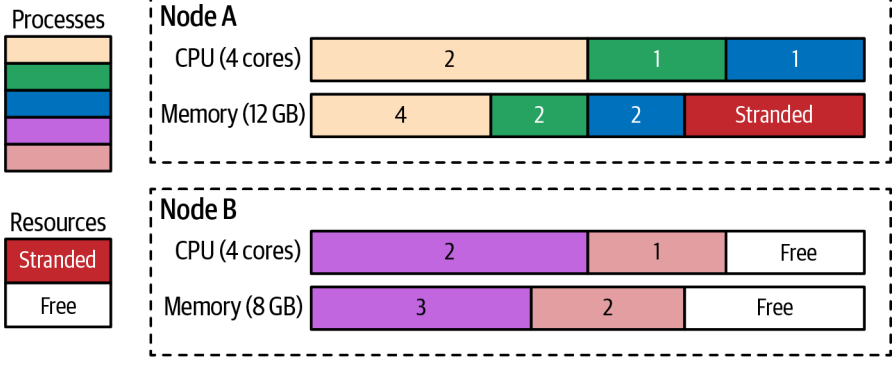

Figure 6-2. Processes scheduled to nodes and stranded resources

Once a Pod is assigned to a node, the job of the scheduler is done, and it does not change the placement of the Pod unless the Pod is deleted and recreated without a node assignment. As you have seen, with time, this can lead to resource fragmentation and poor utilization of cluster resources. Another potential issue is that the scheduler decisions are based on its cluster view at the point in time when a new Pod is scheduled. If a cluster is dynamic and the resource profile of the nodes changes or new nodes are added, the scheduler will not rectify its previous Pod placements. Apart from changing the node capacity, you may also alter the labels on the nodes that affect placement, but past placements are not rectified.

All of these scenarios can be addressed by the descheduler. The Kubernetes descheduler is an optional feature that is typically run as a Job whenever a cluster administrator decides it is a good time to tidy up and defragment a cluster by rescheduling the Pods. The descheduler comes with some predefined policies that can be enabled and tuned or disabled.

Regardless of the policy used, the descheduler avoids evicting the following:

- Node- or cluster-critical Pods
- Pods not managed by a ReplicaSet, Deployment, or Job, as these Pods cannot be recreated
- Pods managed by a DaemonSet
- Pods that have local storage
- Pods with PodDisruptionBudget, where eviction would violate its rules
- Pods that have a non-nil `DeletionTimestamp` field set
- Deschedule Pod itself (achieved by marking itself as a critical Pod)

Of course, all evictions respect Pods' QoS levels by choosing *Best-Efforts* Pods first, then *Burstable* Pods, and finally *Guaranteed* Pods as candidates for eviction. See Chapter 2, "Predictable Demands", for a detailed explanation of these QoS levels.

Discussion

Placement is the art of assigning Pods to nodes. You want to have as minimal intervention as possible, as the combination of multiple configurations can be hard to predict. In simpler scenarios, scheduling Pods based on resource constraints should be sufficient. If you follow the guidelines from Chapter 2, "Predictable Demands", and declare all the resource needs of a container, the scheduler will do its job and place the Pod on the most feasible node possible.

However, in more realistic scenarios, you may want to schedule Pods to specific nodes according to other constraints such as data locality, Pod colocality, application

high availability, and efficient cluster resource utilization. In these cases, there are multiple ways to steer the scheduler toward the desired deployment topology.

Figure 6-3 shows one approach to thinking and making sense of the different scheduling techniques in Kubernetes.

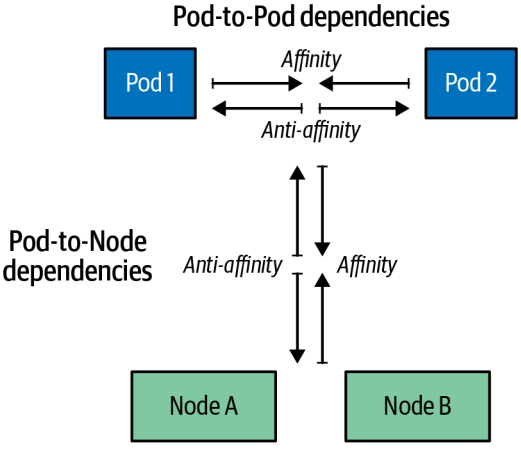

Figure 6-3. Pod-to-Pod and Pod-to-Node and dependencies

Start by identifying the forces and dependencies between the Pod and the nodes (for example, based on dedicated hardware capabilities or efficient resource utilization). Use the following node affinity techniques to direct the Pod to the desired nodes, or use anti-affinity techniques to steer the Pod away from the undesired nodes:

nodeName
: This field provides the simplest form of hard wiring a Pod to a node. This field should ideally be populated by the scheduler, which is driven by policies rather than manual node assignment. Assigning a Pod to a node through this approach prevents the scheduling of the Pod to any other node. If the named node has no capacity, or the node doesn't exist, the Pod will never run. This throws us back into the pre-Kubernetes era, when we explicitly needed to specify the nodes to run our applications. Setting this field manually is not a Kubernetes best practice and should be used only as an exception.

nodeSelector
: A node selector is a label map. For the Pod to be eligible to run on a node, the Pod must have the indicated key-value pairs as the label on the node. Having put some meaningful labels on the Pod and the node (which you should do anyway), a node selector is one of the simplest recommended mechanisms for controlling the scheduler choices.

Node affinity
: This rule improves the manual node assignment approaches and allows a Pod to express dependency toward nodes using logical operators and constraints that provides fine-grained control. It also offers soft and hard scheduling requirements that control the strictness of node affinity constraints.

Taints and tolerations
: Taints and tolerations allow the node to control which Pods should or should not be scheduled on them without modifying existing Pods. By default, Pods that don't have tolerations for the node taint will be rejected or evicted from the node. Another advantage of taints and tolerations is that if you expand the Kubernetes cluster by adding new nodes with new labels, you don't need to add the new labels on all Pods but only on those that should be placed on the new nodes.

Once the desired correlation between a Pod and the nodes is expressed in Kubernetes terms, identify the dependencies between different Pods. Use Pod affinity techniques for Pod colocation for tightly coupled applications, and use Pod anti-affinity techniques to spread Pods on nodes and avoid a single point of failure:

Pod affinity and anti-affinity
: These rules allow scheduling based on Pods' dependencies on other Pods rather than nodes. Affinity rules help for colocating tightly coupled application stacks composed of multiple Pods on the same topology for low-latency and data locality requirements. The anti-affinity rule, on the other hand, can spread Pods across your cluster among failure domains to avoid a single point of failure, or prevent resource-intensive Pods from competing for resources by avoiding placing them on the same node.

Topology spread constraints
: To use these features, platform administrators have to label nodes and provide topology information such as regions, zones, or other user-defined domains. Then, a workload author creating the Pod configurations must be aware of the underlying cluster topology and specify the topology spread constraints. You can also specify multiple topology spread constraints, but all of them must be satisfied for a Pod to be placed. You must ensure that they do not conflict with one another. You can also combine this feature with NodeAffinity and NodeSelector to filter nodes where evenness should be applied. In that case, be sure to understand the difference: multiple topology spread constraints are about calculating the result set independently and producing an AND-joined result, while combining it with NodeAffinity and NodeSelector, on the other hand, filters results of node constraints.

In some scenarios, all of these scheduling configurations might not be flexible enough to express bespoke scheduling requirements. In that case, you may have to customize and tune the scheduler configuration or even provide a custom scheduler implementation that can understand your custom needs:

Scheduler tuning
: The default scheduler is responsible for the placement of new Pods onto nodes within the cluster, and it does it well. However, it is possible to alter one or more stages in the filtering and prioritization phases. This mechanism with extension points and plugins is specifically designed to allow small alterations without the need for a completely new scheduler implementation.

Custom scheduler
: If none of the preceding approaches is good enough, or if you have complex scheduling requirements, you can also write your own custom scheduler. A custom scheduler can run instead of, or alongside, the standard Kubernetes scheduler. A hybrid approach is to have a "scheduler extender" process that the standard Kubernetes scheduler calls out to as a final pass when making scheduling decisions. This way, you don't have to implement a full scheduler but only provide HTTP APIs to filter and prioritize nodes. The advantage of having your scheduler is that you can consider factors outside of the Kubernetes cluster like hardware cost, network latency, and better utilization while assigning Pods to nodes. You can also use multiple custom schedulers alongside the default scheduler and configure which scheduler to use for each Pod. Each scheduler could have a different set of policies dedicated to a subset of the Pods.

To sum up, there are lots of ways to control the Pod placement, and choosing the right approach or combining multiple approaches can be overwhelming. The takeaway from this chapter is this: size and declare container resource profiles, and label Pods and nodes for the best resource-consumption-driven scheduling results. If that doesn't deliver the desired scheduling outcome, start with small and iterative changes. Strive for a minimal policy-based influence on the Kubernetes scheduler to express node dependencies and then inter-Pod dependencies.

More Information

- Automated Placement Example
- Assigning Pods to Nodes
- Scheduler Configuration
- Pod Topology Spread Constraints
- Configure Multiple Schedulers
- Descheduler for Kubernetes