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.

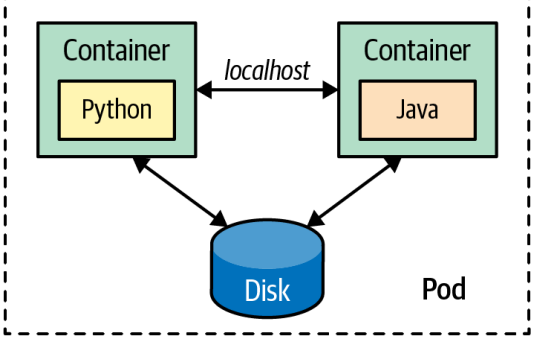

Figure 1-2. A Pod as the deployment and management unit

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.

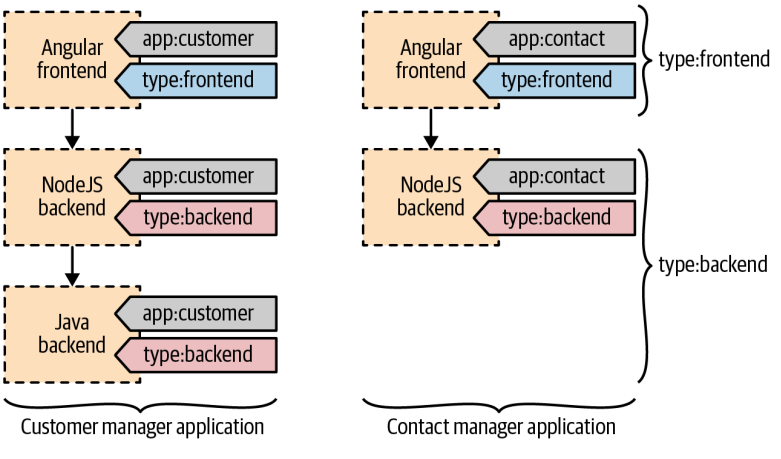

Figure 1-3. Labels used as an application identity for Pods

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.

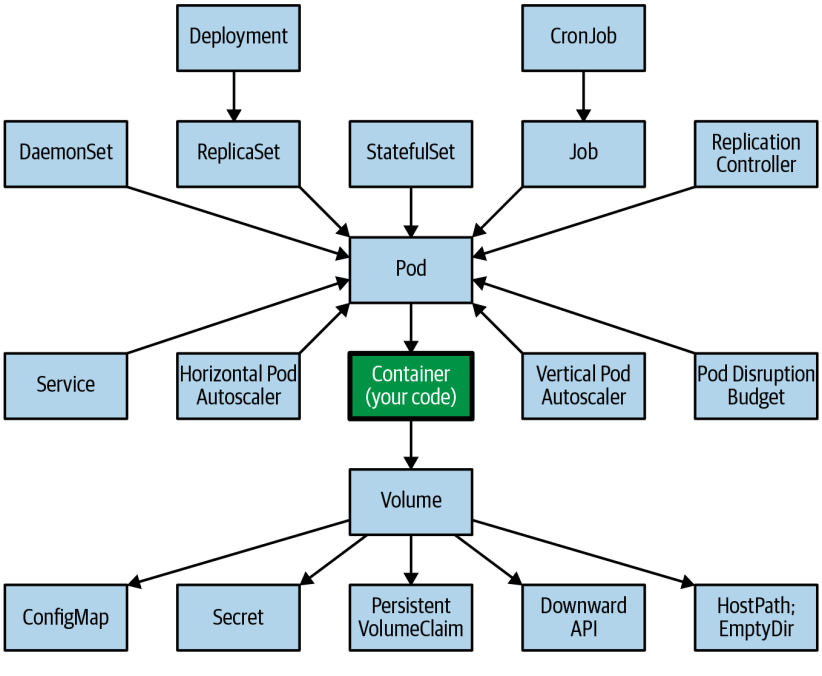

Figure 1-4. Kubernetes concepts for developers

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.

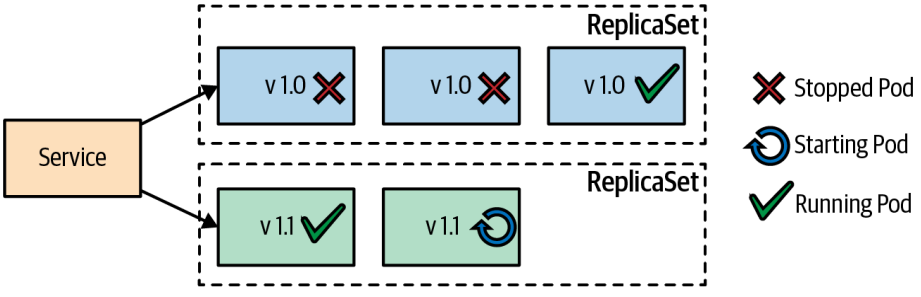

Figure 3-1. Rolling deployment

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.

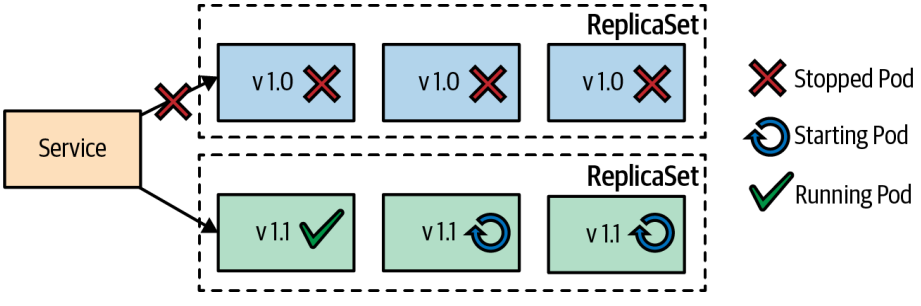

Figure 3-2. Fixed deployment using a `Recreate` strategy

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.

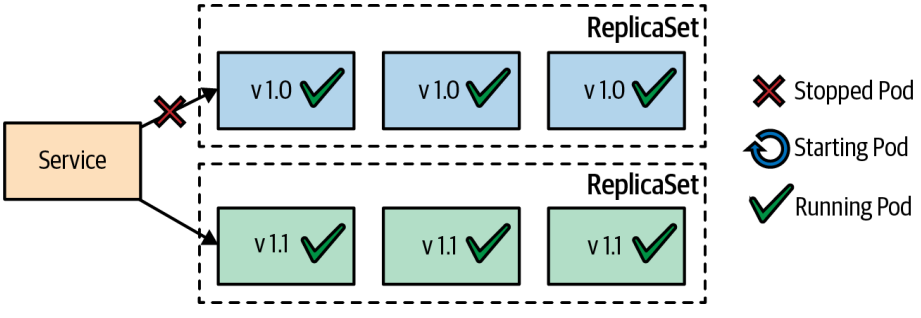

Figure 3-3. Blue-Green release

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.