New Relic

The Fundamentals of Kubernetes

From managing cluster capacity to container health to storage volumes, build your foundational knowledge of Kubernetes with these five fundamentals

Table of Contents

Introduction

A recent survey from Cloud Native Computing Foundation (CNCF) found that 83% of respondents are using Kubernetes in production, up from 58% in 2018. When it comes to automating the deployment, scaling, and management of containerized applications, teams are clearly finding significant value in Kubernetes. But that doesn't mean the technology is simple to comprehend. As teams move to adopt Kubernetes, they may find that the sheer breadth of the platform can be overwhelming.

The five chapters in this ebook explore intro-level Kubernetes fundamentals—including managing cluster capacity, using health checks, organizing clusters, and beyond. If you're a Kubernetes novice, this information is essential, but we suspect even some pros will find value in reviewing these basic elements for successful Kubernetes management.

Chapter 1: How to Manage Cluster Capacity with Requests and Limits

While Kubernetes manages the nodes in your cluster, you have to first define the resource requirements for your applications. Understanding how Kubernetes manages resources, especially during peak times, is important to keep your containers running smoothly.

This chapter looks at how Kubernetes manages CPU and memory using requests and limits.

How requests and limits work

Every node in a Kubernetes cluster has an allocated amount of memory (RAM) and computational power (CPU) that can be used to run containers.

Kubernetes defines a logical grouping of one or more containers into Pods. Pods, in turn, can be deployed and managed on top of the nodes. When you create a Pod, you normally specify the storage and networking that containers share within that Pod. The Kubernetes scheduler will then look for nodes that have the resources required to run the Pod.

To help the scheduler, you can specify a lower and upper RAM and CPU limits for each container using requests and limits. These two keywords enable you to specify the following:

- By specifying a request on a container, you are setting the minimum amount of RAM or CPU required for that container. Kubernetes will roll all container requests into a total Pod request. The scheduler will use this total request to ensure the Pod can be deployed on a node with enough resources.
- By specifying a limit on a container, you are setting the maximum amount of RAM or CPU that the container can consume. Kubernetes translates the limits to the container service (Docker, for instance) that enforces the limit. If a container exceeds its memory limit, it may be terminated and restarted, if possible. CPU limits are less strict and can generally be exceeded for extended periods of time.

Let's see how requests and limits are used.

Setting CPU requests and limits

Requests and limits on CPU are measured in CPU units. In Kubernetes, a single CPU unit equals a virtual CPU (vCPU) or core for cloud providers, or a single thread on bare metal processors.

Under certain circumstances, one full CPU unit can still be considered a lot of resources for a container, particularly in regard to microservices. This is why Kubernetes supports CPU fractions.

While you can enter fractions of the CPU as decimals—for example, 0.5 of a CPU—Kubernetes uses the "millicpu" notation, where 1,000 millicpu (or 1,000m) equals 1 CPU unit.

When you submit a request for a CPU unit, or a fraction of it, the Kubernetes scheduler will use this value to find a node within a cluster that the Pod can run on. For instance, if a Pod contains a single container with a CPU request of 1 CPU, the scheduler will ensure the node it places this Pod on has 1 CPU resource free. For a Docker container, Kubernetes uses the CPU share constraint to proportion the CPU.

If you specify a limit, Kubernetes will try to set the container's upper CPU usage limit. As mentioned earlier, this is not a hard limit, and a container may or may not exceed this limit depending on the containerization technology. For a Docker container, Kubernetes uses the CPU period constraint to set the upper bounds of CPU usage. This allows Docker to restrict the percentage of runtime over 100 milliseconds the container can use.

Below is a simple example of a Pod configuration YAML file with a CPU request of 0.5 units and a CPU limit of 1.5 units.

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-request-limit-example
spec:
  containers:
  - name: cpu-request-limit-container
    image: images.example/app-image
    resources:
      requests:
        cpu: "500m"
      limits:
        cpu: "1500m"
```

This configuration defines a single container called "cpu-request-limit-container" with the image limits specified in the resources section. In that section, you specify your requests and limits. In this case, you are requesting 500 millicpu (0.5 or 50% of a CPU unit) and limiting the container to 1500 millicpu (1.5 or 150% of a CPU unit).

Setting memory requests and limits

Memory requests and limits are measured in bytes, with some standard short codes to specify larger amounts, such as kilobytes (K) or 1,000 bytes, megabytes (M) or 1,000,000 bytes, and gigabytes (G) or 1,000,000,000 bytes. There are also power-of-two equivalents for these shortcuts. For example, Ki (1,024 bytes), Mi, and Gi. Unlike CPU units, there are no fractions for memory as the smallest unit is a byte.

The Kubernetes scheduler uses memory requests to find a node within your cluster that has enough memory for the Pod to run on. Memory limits work in a similar way to CPU limits, except they

are enforced more strictly. If a container exceeds a memory limit, it might be terminated and potentially restarted with an "out of memory" error.

The simple example of a Pod configuration YAML file below contains a memory request of 256 megabytes and a memory limit of 512 megabytes.

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-request-limit-example
spec:
  containers:
  - name: memory-request-limit-container
    image: images.example/app-image
    resources:
      requests:
        memory: "256M"
      limits:
        memory: "512M"
```

This configuration defines a single container called "memory-request-limit-container" with the image limits specified in the resources section. You have specified the memory request of 256M, and limited the container to 512M.

Setting limits via namespaces

If you have several developers, or teams of developers, working within the same large Kubernetes cluster, a good practice is to set common resource requirements to ensure resources are not consumed inadvertently. With Kubernetes, you can define different namespaces for teams and use resource quotas to enforce quotas on these namespaces.

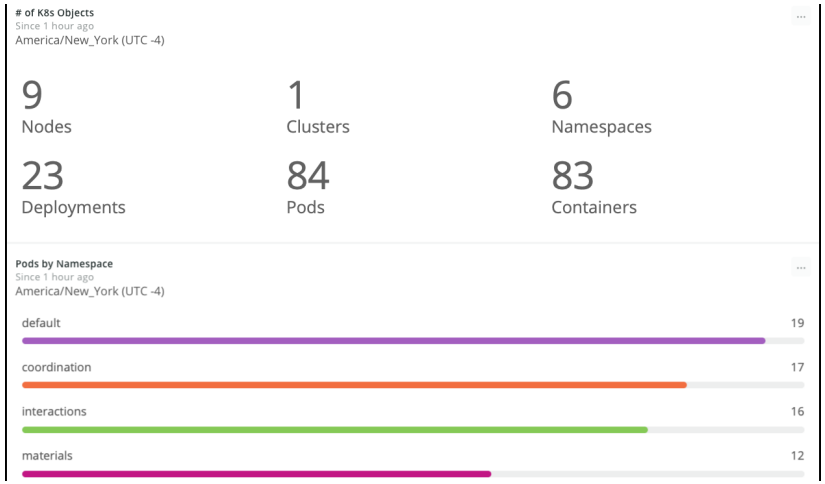
For instance, you may have a Kubernetes cluster that has 64 CPU units and 256 gigabytes of RAM spread over eight nodes. You might create three namespaces—one for each of your development teams—with the resource quota of 10 CPU units and 80 Gigabytes of memory. This would allow each development team to create any number of Pods up to that limit, with some CPU and memory left in reserve.

For more information on specifying resource quotas for namespaces, refer to the Resource Quotas section of the Kubernetes documentation.

The importance of monitoring cluster capacity

Setting requests and limits on both containers and namespaces can go a long way to ensure your Kubernetes cluster does not run out of resources. Monitoring, however, still plays an important role in maintaining the health of individual services, as well as the overall health of your cluster.

When you have large clusters with many services running within Kubernetes Pods, health and error monitoring can be difficult. Observability tools, such as New Relic, offer an easy way to monitor your Kubernetes cluster and the services running within it. It helps you make sure that requests and limits you are setting at the container and across the cluster are appropriate.



Having a good understanding of how Kubernetes handles CPU and memory resources, as well as enabling configuration to manage these resources, is critical to ensure your Kubernetes clusters have enough capacity at all times. As we've seen, setting CPU and memory requests and limits is easy—and now you know how to do it. By adding a layer of monitoring, you will go a long way to ensuring that Pods are not fighting for resources on your cluster.

Chapter 2: How to Use Health Checks

To manage containers effectively, Kubernetes needs a way to check container health to ensure that they are working correctly and receiving traffic. Kubernetes uses health checks—also known as probes—to determine if instances of your app are running and responsive.

This chapter discusses the different probe types and the various ways to use them.

Why probes are important

Distributed systems can be hard to manage. Because the separate components work independently, each part will keep running even after other components have failed. At some point, an application might crash. Or an application might be still in the initialization stage and not yet ready to receive and process requests.

You can only assert the system's health if all of its components are working. Using probes, you can determine whether a container is dead or alive, and decide if Kubernetes should temporarily prevent other containers from accessing it. Kubernetes verifies individual containers' health to determine the overall Pod health.

Types of probes

As you deploy and operate distributed applications, containers are created, started, run, and terminated. To check a container's health in the different stages of its life cycle, Kubernetes uses different types of probes:

Liveness probes allow Kubernetes to check if your app is alive. The kubelet agent that runs on each node uses the liveness probes to ensure that the containers are running as expected. If a container app is no longer serving requests, kubelet will intervene and restart the container.

For example, if an application is not responding and cannot make progress because of a deadlock, the liveness probe detects that it is faulty. Kubelet then terminates and restarts the container. Even if the application carries defects that cause further deadlocks, the restart will increase the container's availability. It also gives your developers time to identify the defects and resolve them later.

Readiness probes run during the entire life cycle of the container. Kubernetes uses this probe to know when the container is ready

to start accepting traffic. If a readiness probe fails, Kubernetes will stop routing traffic to the Pod until the probe passes again.

For example, a container may need to perform initialization tasks, including unzipping and indexing files and populating database tables. Until the startup process is completed, the container will not be able to receive or serve traffic. During this time, the readiness probe will fail, so Kubernetes will route requests to other containers.

A Pod is considered ready when all of its containers are ready. That helps Kubernetes control which Pods are used as backends for services. If it's not ready, a Pod is removed from service load balancers.

Startup probes are used to determine when a container application has been initialized successfully. If a startup probe fails, the Pod is restarted.

When Pod containers take too long to become ready, readiness probes might fail repeatedly. In this case, containers risk being terminated by kubelet before they are up and running. This is where the startup probe comes to the rescue.

The startup probe forces liveness and readiness checks to wait until it succeeds, so that the application startup is not compromised. That is especially beneficial for slow-starting legacy applications.

Creating probes

To create health check probes, you must issue requests against a container. There are three ways of implementing Kubernetes liveness, readiness, and startup probes:

1. Sending an HTTP request
2. Running a command
3. Opening a TCP socket

HTTP REQUESTS

An HTTP request is a common and straightforward mechanism for creating a liveness probe. To expose an HTTP endpoint, you can implement any lightweight HTTP server in your container.

A Kubernetes probe will perform an HTTP GET request against your endpoint at the container's IP to verify whether your service is alive. If your endpoint returns a success code, kubelet will consider the container alive and healthy. Otherwise, kubelet will terminate and restart the container.

Suppose you have a container based on an image named k8s.gcr.io/liveness. In that case, if you define a liveness probe that uses an HTTP GET request, your YAML configuration file would look similar to this snippet:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

The configuration defines a single-container Pod with initialDelaySeconds and periodSeconds properties that tell kubelet to execute a liveness probe every three seconds then wait three seconds before performing the first probe. Kubelet will check whether the container is alive and healthy by sending requests to the /healthz path on `8080` port and expect a success result code.

COMMANDS

When the HTTP requests are not suitable, you can use command probes.

Once you have a command probe configured, kubelet executes the `cat /tmp/healthy` command in the target container. Kubelet considers your container alive and healthy if the command succeeds. Otherwise, Kubernetes terminates and restarts the container.

This is how your YAML configuration would look for a new Pod that runs a container based on the k8s.gcr.io/busybox image:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
```

```
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/
healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

The above configuration defines a single container Pod with the `initialDelaySeconds` and the periodSeconds keys tell kubelet to perform a liveness probe every five seconds then wait five seconds before the first probe is completed.

Kubelet will run the `cat /tmp/healthy` command in the container to execute a probe.

TCP CONNECTIONS

When a TCP socket probe is defined, Kubernetes tries to open a TCP connection on your container's specified port. If Kubernetes succeeds, the container is considered healthy. TCP probes are helpful when HTTP or command probes are not adequate. Scenarios in which containers can benefit from TCP probes include gRPC and FTP services, where the TCP protocol infrastructure already exists.

With the following configuration, kubelet will try to open a socket to your container on the specified port.

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

The above configuration is similar to the HTTP check. It defines a readiness and a liveness probe. When the container starts, kubelet will wait five seconds to send the first readiness probe. After that, kubelet will keep checking the container readiness every 10 seconds.

Monitoring Kubernetes health

Probes tell Kubernetes whether your containers are healthy, but they don't tell you anything.

When you have many services running in Kubernetes Pods deployed across many nodes, health and error monitoring can be difficult.

As a developer or a DevOps specialist working with the Kubernetes platform, you might find New Relic an excellent tool for checking Kubernetes' health, gathering insights, and troubleshooting container issues.

Health checks via probes are essential to ensure that your containers are good citizens in a cluster. Kubernetes uses liveness, readiness, and startup probes to decide when a container needs to be restarted, or a Pod needs to be removed from service. That helps you keep your distributed system services reliable and available.

Chapter 3: How to Use Kubernetes Secrets

Containerized applications running in Kubernetes frequently need access to external resources that often require secrets, passwords, keys, or tokens to gain access. Kubernetes Secrets lets you securely store these items, removing the need to store them in Pod definitions or container images.

This chapter addresses various ways to create and use secrets in Kubernetes, with an aim to help you select the best approach for your environment.

Creating Kubernetes Secrets

Kubernetes Secrets offers three methods to create and store secrets:

- Via the command line
- In a configuration file
- With a generator

Let's take a look at creating some secrets with these methods.

Creating Kubernetes Secrets from the command line

You can create a secret via the Kubernetes administrator command line tool, `kubectl`. This tool allows you to use files or pass in literal strings from your local machine, package them into secrets, and create objects on the cluster server using an API. It's important to note that secret objects must be in the form of a DNS subdomain name.

For username and password secrets, use this command line pattern:

```
kubectl create secret generic <secret-object-name>
<flags>
```

For secrets using Transport Layer Security (TLS) from a given public/private key pair, use this command line pattern:

```
kubectl create secret tls <secret-object-name>
--cert=<cert-path> --key=<key-file-path>
```

You can also create a generic secret using a username and password combination for a database. This example applies the literal flag to specify the username and password at the command prompt:

```
kubectl create secret generic sample
-db-secret --from-literal=username=admin
--from-literal=password='7f3,F9D^LJz37]!W'
```

The command creates a new secret called sample-db-secret, with a username value of admin and a password value of `7f3,F9D^LJz37]!W`. It is worth noting that strong, complex passwords often have characters that need to be escaped. To avoid this, you can put all your usernames and passwords in text files and use the following flags:

```
kubectl create secret generic sample-db-secret
--from-file=username.txt --from-file=password.txt
```

This command drops the username and password keys as it provides the name of a file containing this information. You can add this back into the `--from-file` switch the same way as the `--from-literal` switch if the key is different from the file name.

```
kubectl create secret generic sample-db-secret
--from-file=username=123.txt --from-file=pass-
word=xyz.txt
```

Setting Kubernetes Secrets in a configuration file

Another option is to create your secret using a JSON or YAML configuration file. A secret created in a configuration file has two data maps: data and stringData. The former requires your values to be base64-encoded, whereas the latter allows you to provide values as unencoded strings.

The following template is used for secrets in YAML files:

```
apiVersion: v1
kind: Secret
metadata:
    name: <secret name>
type: Opaque
data:
    <key>: <base64 Value>
stringData:
    <key>: <string value>
```

You apply the template using the `kubectl apply -f ./<file-name>.yaml` command. As an example, here is a YAML file for an application that requires a number of secret values:

```
apiVersion: v1
kind: Secret
metadata:
```

```
    name: my-example-app
type: Opaque
data:
    app-user: YWRtaW5pc3RyYXRvcg==
    app-password: cGFzc3dvcmQ=
stringData:
    Dbconnection:
Server=tcp:myserver.database.net,1433;Database=myD-
B;User ID=mylogin@myserver;Password=myPass-
word;Trusted_Connection=False;Encrypt=True;
    config.yaml: |-
        LogLevel: Warning
        API_TOKEN: NcNIMcMYMAMg.MGwjPnPfEBgqMl8Q
        API_URI: https://www.myapp.com/api
```

The above YAML file contains a number of values, including:

- The secret name (my-example-app)
- An `app-user` ("administrator," base64-encoded)
- An `app-pasword` ("password," base64-encoded)
- A `dbconnection` string
- The `config.yaml` file with data

This is a good way for packaging many secrets and potentially sensitive configuration information into a single configuration file.

Creating Kubernetes Secrets with a generator

The third option for creating secrets is to use Kustomize, a stand-alone tool for customizing Kubernetes objects using a configuration file called `kustomization.yaml`.

Kustomize allows you to generate secrets in a fashion similar to the command line by specifying secrets in files (with a key/value pair on each line), or as literals within the configuration file.

For secrets, the following structure is used within a kustomization.yaml file:

```
secretGenerator:
    name: <secret-name>
    files:
        <filename>
    literals:
        <key>=<value>
```

When you have created the kustomization.yaml file and included all the linked files in a directory, you can use the kubectl kustomize <directory> command, then apply the configuration using the kubectl apply -k <directory> command.

The following example `kustomization.yaml` file creates a secret with two literal key/values (API_TOKEN and API_URI), as well as a config.yaml file:

```
secretGenerator:
    name: example-app-secrets
    files:
        passwords.txt
    literals:
        API_TOKEN: NcNIMcMYMAMg.MGwjPnPfEBgqMl8Q
        API_URI: https://www.myapp.com/api
```

The `config.yaml` file referenced in this example could be the config file for an application, for instance.

Which method for creating Kubernetes Secrets is the best?

Each of the methods we've discussed is "the best" under specific circumstances.

The command line is most useful when you have one or two secrets you want to add to a Pod—for instance a username and password—and you have them in a local file or want to pass them in as literals.

Configuration files are great when handling a handful of secrets that you want to include in a Pod all at one time.

The Kustomize configuration file is the preferred option when you have one or more configuration files and secrets you want to deploy to multiple Pods.

Once you have created your secrets, you can access them in two main ways:

- Via files (volume-mounted)
- Via environment variables

The first option is similar to accessing configuration files as part of the application process. The second option loads the secrets as environment variables for the application to access. We are going to explore both methods.

Accessing volume-mounted Kubernetes secrets

To access secrets loaded in a volume, first you need to add the secret to the Pod under `spec[].[]volumes[].secret.secretName`. You then add a volume to each container under `spec[].containers[].volumeMounts`, where the name of the volume is the same as that of the secret, and where readOnly is set to `"true"`.

There are also a number of additional options you can specify. Let's have a look at an example Pod with a single container:

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
spec:
```

```
    containers:
        - name: myapp
        image: ubuntu
        volumeMounts:
            - name: secrets
            mountPath: "/etc/secrets"
            readOnly: true
volumes:
    - name: secrets
    secret:
        secretName: mysecret
        defaultMode: 0400
        items:
            - key: username
            path: my-username
```

This configuration file specifies a single Pod (`mypod`) with a single container (`myapp`). In the volumes section for the Pod, you have a volume named secrets, which is shared by all containers. This volume is of type secret, and it loads the secret called `mysecret`. It loads the volume using the Unix file permissions `0400`, which gives read access to the owner (`root`), and no access to other users.

The secret also contains the items list, which casts only specific secret keys (in this case, `username`) and an appended path (`my-username`). In your container (`myapp`), you map the volume in volumeMounts (name is `secrets`) to the mountPath as read only.

Because you're casting the username to my-username, the directory `/etc/secrets` in the container will look something like this:

```
lrwxrwxrwx 1 root root 2 September 20 19:18 my-username -> ..data/username
```

Because you're casting a single value and changing the path, the key has changed. If you follow symlink, you will see that the permissions are set correctly:

```
-r-------- 1 root root  2 September 20 19:18 username
```

All files that reside on the secret volume will contain the base64-decoded values of the secrets.

The advantage of loading your secrets into a volume is that, when the secrets are updated or modified, the volume is eventually updated as well, allowing your applications to re-read the secrets. It also makes parsing secret files (such as sensitive configuration files), as well as referencing multiple secrets, a lot easier.

Accessing Kubernetes Secrets as environment variables

You can project your secrets into a container using environment variables. To do this, you add an environment variable for every

secret key you wish to add using `env[].valueFrom.secretKeyRef`. Let's have a look at an example Pod specification:

```
apiVersion: v1
kind: Pod
metadata:
    name: mypod
spec:
    containers:
        - name: myapp
        image: ubuntu
        env:
            - name: USERNAME
            valueFrom:
                secretKeyRef:
                    name: mysecret
                    key: username
            - name: PASSWORD
            valueFrom:
                secretKeyRef:
                    name: mysecret
                    key: password
```

This configuration file passes two keys (`username` and `password`) from the mysecret secret to the container as environment variables. These values are the base64-decoded values of the secrets.

If you logged into the container and ran the echo `$USERNAME` command, you would get the decoded value of the `username` secret (for example, "admin").

One of the biggest advantages of casting secrets this way is that you can be very specific with the secret value. Besides, for some applications, reading environment variables is easier than parsing configuration files.

Alternatives to Kubernetes Secrets

While secret management in a Kubernetes cluster is relatively simple, fairly secure, and can meet most requirements, it does have some downsides. In particular, secrets use namespaces such as Pods, so if secrets and Pods are in the same namespace, all Pods can read the secrets.

The other major downside is that keys are not rotated automatically. You need to manually rotate secrets.

To address these issues and provide a more centralized secret management, you can use an alternative configuration, such as:

- Integrating a cloud vendor secrets management tool, such as Hashicorp Vault or AWS Secrets Manager. These tools typically use Kubernetes service accounts to grant access to the vault for secrets and mutating webhooks to mount the secrets into the Pod.
- Integrating a cloud vendor Identity and Access Management (IAM) tool, such as AWS Identity and Access Manager. This type

of integration uses a method similar to OpenID Connect for web applications, which allows Kubernetes to utilize tokens from a Secure Token Service.

- Running a third-party secrets manager, such as Conjur loaded into Pods as a sidecar.

Secure storage of secrets is critical to running containers in Kubernetes because almost all applications require access to external resources—databases, services, and so on. Using Kubernetes Secrets allows you to manage sensitive application information across your cluster, minimizing the risks of maintaining secrets in a non-centralized fashion.

Chapter 4: How to Organize Clusters

Kubernetes was designed to scale. A team can start a small cluster and progressively expand its installation. After a while, the cluster may be running dozens of Pods and hundreds of containers, or even more.

However, without organization, the number of deployed services and objects can quickly get out of control, leading to performance, security, and other issues.

This chapter examines three key tools you can use to help keep your Kubernetes cluster organized: namespaces, labels, and annotations.

How namespaces work

By default, Kubernetes provides just one workable namespace on top of a physical cluster in which you create all Kubernetes objects. Eventually, though, your project is likely to grow to a point where a single namespace will become a limitation.

Luckily, you can think of a namespace as a virtual cluster, and Kubernetes supports multiple virtual clusters. Configuring multiple namespaces creates a sophisticated façade, freeing your teams from working with a single namespace and improving manageability, security, and performance.

Different companies will adopt different namespace strategies, depending on factors such as team size, structure, and the complexity of their projects. A small team working with a few microservices can easily deploy all services into the default namespace. But for a rapidly growing company, with far more services, a single namespace could make it hard to coordinate the team's work. In this case, the company could create sub-teams, with a separate namespace for each.

In larger companies, too, teams may be widely dispersed, often working on projects that other teams aren't aware of, making it hard to keep up with frequent changes. Third-party companies might also contribute to the platform, further increasing complexity. Coordinating so many resources is an administrative challenge, and for developers it becomes impossible to run the entire stack on the local machine. In addition to technologies such as service mesh and multi-cloud continuous delivery, multiple namespaces are essential to managing large-scale scenarios.

When different teams deploy projects to the same namespace, they risk affecting one another's work. By providing isolation and team-based access security, separate namespaces help ensure teams work on their own without disrupting others. You can also set a resource quota per namespace, so that a resource-hungry application doesn't exhaust the cluster capacity, impacting other teams' resources.

Using namespaces

When you create a cluster, Kubernetes provides three namespaces out of the box. To list the namespaces that come with the cluster, run the following command:

```
$kubectl get namespaces
NAME          STATUS    AGE
default       ACTIVE    2d
kube-system   ACTIVE    2d
kube-public   ACTIVE    2d
```

The `kube-system` namespace is reserved for the Kubernetes engine and is not meant for your use. The `kube-public` namespace is where public access data is stored, such as cluster information.

The default namespace is where you create apps and services. Whenever you create a component and don't specify a namespace, Kubernetes creates it in the default namespace. But using the default namespace is suitable only when you're working on small systems.

You can create a Kubernetes namespace with a single `kubectl` command:

```
kubectl create namespace test
```

Alternatively, you can create namespaces with a YAML configuration file, which might be preferable if you want to leave a history in your configuration file repository of the objects that have been created in a cluster. The following `demo.yaml` file shows how to create a namespace with a configuration file:

```
kind: Namespace
apiVersion: v1
metadata:
  name: demo
  labels:
    name: demo
kubectl apply -f demo.yaml
```

Imagine having three projects—sales, billing, and shipping. You wouldn't want to deploy all of them into a single default namespace, for the reasons presented earlier, so you'd start by creating one namespace per project.

The problem is, each project has its own life cycle and you don't want to mix development and production resources. So, as your

projects get more complicated, your cluster needs a more sophisticated namespace solution. You could further split your cluster into development, staging, and production environments:

```
kind: Namespace
apiVersion: v1
metadata:
  name: dev
  labels:
    name: dev
kubectl apply -f dev.yaml
```

```
kind: Namespace
apiVersion: v1
metadata:
  name: staging
  labels:
    name: staging
kubectl apply -f staging.yaml
```

```
kind: Namespace
apiVersion: v1
metadata:
  name: prod
  labels:
    name: prod
kubectl apply -f prod.yaml
```

Here's a list of potential namespaces you might employ, depending on the needs of your projects:

- sales-dev
- sales-staging
- sales-prod
- billing-dev
- billing-staging
- billing-prod
- shipping-dev
- shipping-staging
- shipping-prod

There are two ways to explicitly tell Kubernetes in which namespace you want to create your resources.

You can specify the `namespace` flag when creating the resource with the `kubectl` apply command:

```
kubectl apply -f pod.yaml --namespace=demo
```

You can also modify the YAML configuration file metadata to include the destination namespace attribute:

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  namespace: demo
  labels:
    name: testpod
```

```
spec:
  containers:
  - name: testpod
    image: nginx
```

If you predefine the namespace in the YAML declaration, the resource will always be created in that namespace. If you try to run the `kubectl apply` command with the namespace flag to set a different `namespace` for this resource, the command will fail.

Using labels

As the number of objects in your cluster grows, it can be hard to find and organize them. The increased complexity of the projects means their multidimensional components may cross boundaries and challenge rigid cluster structures. Labels let you attach meaningful and relevant metadata to cluster objects so they can be categorized, found, and operated on in bulk.

Labels are key/value structures assigned to objects. An object can be assigned one or more label pairs or no label at all. Labels can be useful for:

- Determining whether a Pod is part of a production or a canary deployment
- Differentiating between stable and alpha releases
- Specifying to which layer (e.g., UI, business logic, database) an object belongs
- Identifying whether a Pod is frontend or backend
- Specifying an object's release version (e.g., V1.0, V2.0, V2.1)

For example, the following configuration file defines a Pod that has two labels: layer: interface and version: stable.

```
apiVersion: v1
kind: Pod
metadata:
  name: app-gateway
  labels:
    layer: interface
    version: stable
```

Once the labels are in place, you can use label selectors to select Kubernetes objects according to criteria you define.

Let's say you have some Pods in a cluster and they've been assigned labels. The following command will get you all Pods and their labels:

```
kubectl get pods --show-labels
```

```
NAME         READY STATUS  RESTARTS AGE LABELS
app-gateway 1/1   Running 0        1m
layer=interface,version=stable
micro-svc1  1/1   Running 0        1m
layer=business,version=stable
```

```
micro-svc2   1/1    Running 0          1m
layer=business,version=alpha
```

Kubernetes lets you use label selectors to run the same `kubectl get pods` command and retrieve only Pods with the specified labels. The following `-L` selector allows you to display only the layer label:

```
kubectl get pods -L=layer

NAME         READY STATUS   RESTARTS AGE LABELS
app-gateway  1/1    Running 0          1m  interface
micro-svc1   1/1    Running 0          1m  business
micro-svc2   1/1    Running 0          1m  business
```

If you want to filter the results and retrieve just the interface pods, you can use the -l and specify this condition:

```
kubectl get pods -l=layer=interface --show-labels

NAME         READY STATUS   RESTARTS AGE LABELS
app-gateway  1/1    Running 0          1m
layer=interface,version=stable
```

For more on labels and label selectors, refer to the Kubernetes Labels and Selectors page.

Using annotations

Annotations are similar to labels. They're also structured as key/value pairs, but unlike labels, they're not intended to be used in searches.

So why should you bother to use annotations, when you can use labels?

Imagine having to organize a warehouse where boxes are stored. These boxes have small labels attached outside that include important data to help you identify, group, and arrange them. Now imagine that each of these boxes contains information. Think of these contents as annotations that you can retrieve when you open the box, but that you don't need to be visible from the outside.

Unlike labels, annotations can't be used to select or identify Kubernetes objects, so you can't use selectors to query them. You could store this kind of metadata in external databases, but that would be complicated. Instead, you can conveniently attach annotations to the object itself. Once you access the object, you can read the annotations.

Annotations can be helpful for a number of use cases, including:

- The Docker registry where a Pod's containers are stored
- The git repo from which a container is built
- Pointers to logging, monitoring, analytics, or audit repositories

- Debugging-related information, such as name, version, and build information
- System information, such as URLs of related objects from other ecosystem components
- Rollout metadata, such as config or checkpoints
- Phone numbers or email addresses of people in charge of the component's project
- The team's website URL

In the following example, a Pod configuration file has the information on the git repo from which a container is built, as well as the team manager's phone number:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-test
  annotations:
    repo: "https://git.your-big-company.com.br/lms/
new-proj"
    phone: 800-555-1212
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

You can also add annotations to an existing Pod with the `annotate` command:

```
kubectl annotate pod annotations-test
phone=800-555-1212
```

To access Pod annotations, you can use:

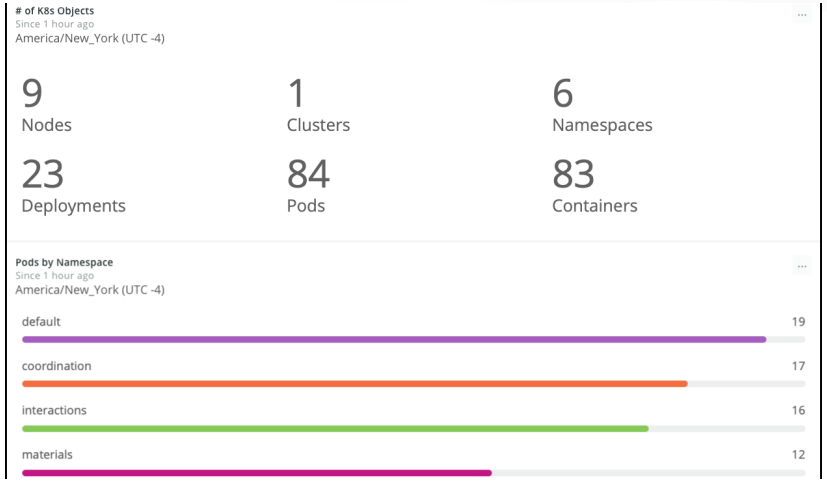
```
kubectl describe pod your-pod-name
```

This will give you a full description of your Pod, or you can use `kubectl get pods` command with the `JSONPath` template to read just the annotation data from Pods:

```
kubectl get pods -o=jsonpath="{.items[*]['metadata.
annotations']}"
```

Organizing your cluster, organizing your journey

Namespaces, labels, and annotations are handy tools for keeping your Kubernetes cluster organized and manageable.



A New Relic One dashboard showing Kubernetes information and a breakdown of Pods by namespace

None of these tools are hard to use. As with most things in Kubernetes, the individual concepts are easy to learn—there are just a lot of them to learn.

But now you're further along your Kubernetes journey because you understand namespaces, labels, and annotations—and how to use them.

Chapter 5: Working With Kubernetes Volumes

There are many advantages to using containers to run applications. However, ease of storage is certainly not one of them. To do its job, a container must have a temporary file system. But when a container shuts down, any changes made to its file system are lost. A side effect of easily fungible containers is that they lack an inherent concept of persistence.

While Docker has solved this issue with mount points from the host, on Kubernetes you face more difficulties along the way. As you've learned, the smallest deployable unit of computing in Kubernetes is a Pod. Multiple instances of a Pod may be hosted on multiple physical machines. Even worse, different containers might run in the same Pod but access the same storage.

This chapter covers two tools Kubernetes offers to help solve storage issues: volumes and persistent volumes. We'll cover how and why you'd use each.

About Kubernetes volumes

Volumes offer storage shared between all containers in a Pod. This allows you to reliably use the same mounted file system with multiple services running in the same Pod. This is, however, not automatic. Containers that need to use a volume have to specify which volume they want to use, and where to mount it in the container's file system.

Additionally, volumes come with a clearly defined life cycle. They are bound to the life cycle of the Pod they belong to. As long as the Pod is active, the volume is there, too. However, when you restart the Pod, the volume gets reset. If this is not what you want, you should either use persistent volumes (discussed in the next section) or change your application's logic to accommodate this behavior appropriately.

While Kubernetes cares about only the formal definition of a volume, you also need to have a real (physical) file system allocated somewhere. This is where Kubernetes goes beyond what Docker offers. While Docker only maps a path from the host to the con-

tainer, Kubernetes allows essentially anything as long as there is a proper provider for the storage.

You could use cloud options such as Amazon Elastic Block Store (EBS) or Microsoft Azure Blob Storage, or an open source solution such as Ceph. Using something as simple and generic as NFS is possible, too. If you want to use something similar to Docker's mount path, you can fall back to the `hostPath` volume type.

So how do you create these volumes? You do so in the Pod definition.

Working with volumes

For example, consider creating a new Pod called `sharedvolumeexample` using two containers—both just sleeping. Using the `volumes` key, you can describe your volumes to be used within the containers.

```
kind: Pod
apiVersion: v1
metadata:
  name: sharedvolumeexample
spec:
  containers:
  - name: c1
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  - name: c2
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    emptyDir: {}
```

To use a volume in a container, you need to specify `volumeMounts` as shown above. The `mountPath` key describes the volume access path.

To demonstrate how this shares the volume between the two containers, let's run a little test. First, you should create the Pod from the spec (for example, `sharedvolumeexample.yml`):

```
kubectl apply -f sharedvolumeexample.yml
```

Then, you can access the terminal on the first container, c1, using `kubectl`:

```
kubectl exec -it sharedvolumeexample -c c1 -- bash
```

Next, write some data into a file under the `/tmp/xchange` mount point:

```
echo 'some data' > /tmp/xchange/file.txt
```

Let's open another terminal, connecting to the container called c2:

```
kubectl exec -it sharedvolumeexample -c c2 -- bash
```

The difference is that this time you read from its mounted storage at `/tmp/data`:

```
cat /tmp/data/file.txt
```

This yields "some data," as expected. Now you can remove the Pod:

```
kubectl delete pod/sharedvolumeexample
```

Working with persistent volumes

When (regular) volumes don't meet your needs, you can switch to a persistent volume.

A persistent volume is a storage object that lives at the cluster level. As a result, its life cycle isn't tied to that of a single Pod, but rather to the cluster itself. A persistent volume makes it possible to share data among Pods.

One advantage of a persistent volume is that it can be shared not only among containers of a single Pod, but also among multiple Pods. This means persistent volumes can be scaled by expanding their size. Reducing size, however, is not possible.

A persistent volume offers the same options for selecting the physical provider as a regular volume. Provisioning, however, is a bit different.

There are two ways to provision a persistent volume:

- **Statically:** You already allocated everything on the storage side, so there's nothing to be done. The physical storage behind will always be the same.
- **Dynamically:** You might want to extend the available storage space when the demand grows. The demand is settled via a volume claim resource, which we'll discuss in a bit. To enable dynamic storage provisioning, you have to enable the `DefaultStorageClass` admission controller on the Kubernetes API server.

For growing systems with demand increase backed by scalable resources, dynamic provisioning makes more sense. Otherwise, we recommend staying with the simpler static provisioning.

Let's try to create a persistent volume for a `hostPath`-backed storage. Note that instead of configuring kind as Pod, you instead configure as `PersistentVolume`:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: persvolumeexample
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

As with Pods, these resources are created using the `kubectl` tool:

```
kubectl apply -f persvolumeexample.yml
```

In this example, you created a new persistent volume named `persvolumeexample`, with the maximum storage capacity of 10GB. As for the different access modes, you could specify `ReadWriteOnce`, `ReadOnlyMany`, and `ReadWriteMany`, although not all of these modes are available for every storage provider. For instance, AWS EBS only supports `ReadWriteOnce`.

You can use the created persistent volume via another resource: `PersistentVolumeClaim`. The claim ensures that there is enough space available. This might fail even if, during dynamic provisioning, Kubernetes actively tries to allocate more space.

Let's create a claim for provisioning 3GB:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim-1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

The provisioning requires the use of `kubectl`:

```
kubectl apply -f myclaim-1.yml
```

When you run this command, Kubernetes looks for a persistent volume that matches the claim. Using the claim is simple:

```
kind: Pod
apiVersion: v1
metadata:
```

```
  name: volumeexample
spec:
  containers:
  - name: c1
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  - name: c2
    image: centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    persistentVolumeClaim:
      claimName: myclaim-1
```

If you compare this example with the previous one, you'll see that only the `volumes` section has changed, nothing else.

The claim manages only a fraction of the volume. To free this fraction, you'd have to delete the claim. The reclaim policy for a persistent volume tells Kubernetes what to do with the volume after it has been released of its claim. The options are `Retain`, `Recycle` (deprecated in preference of dynamic provisioning), and `Delete`.

To set the reclaim policy, you need to define the `persistentVolumeReclaimPolicy` option in the spec section of the `PersistentVolume` config. For instance, in the previous config this would look like:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: persvolumeexample
  labels:
    type: local
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: "/tmp/data"
```

Choose your volume wisely

Both volumes and persistent volumes allow you to add data storage that survives container restarts. While volumes are bound to the life cycle of the Pod, persistent volumes can be defined independently of a specific Pod. They can then be used in any Pod.

The one you choose depends on your needs. A volume is deleted when the containing Pod shuts down, yet it is perfect when you need to share data between containers running in a Pod.

Because persistent volumes outlive individual Pods, they're ideal when you have data that must survive Pod restarts or has to be shared among Pods.

Both types of storage are easy to set up and use in a cluster.

Hopefully, the approaches we've documented help you gain some control over your Kubernetes environment, so you can get busy shipping more perfect software. When you're ready for a deep dive into Kubernetes monitoring, check out A Complete Introduction to Monitoring Kubernetes with New Relic.

Kubernetes Monitoring

Get complete observability into your clusters.

Learn More