



Understanding Basic Kubernetes Concepts

This guide will help you understand basic Kubernetes concepts.

From the Author

In the first chapter I'll explain the concepts of Pods, Labels, and ReplicaSets. In the second chapter we'll talk about Deployments. The third chapter explains the Services concept and in the fourth we'll look at Secrets and ConfigMaps. In the final chapter we'll talk about DaemonSets and Jobs.

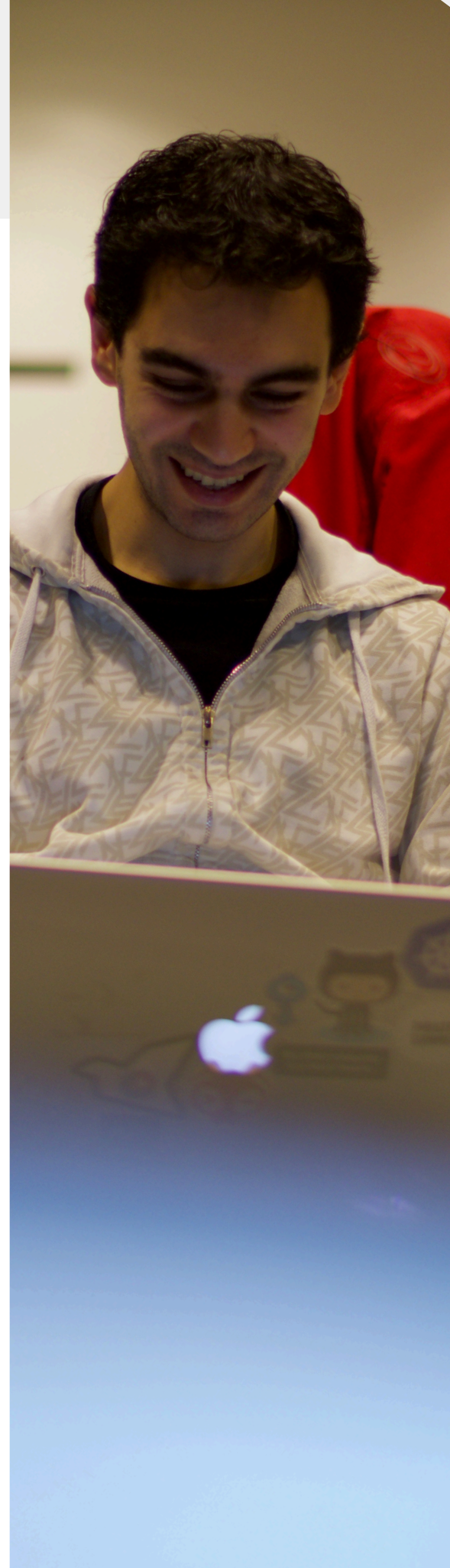
There's lots of introductions and tutorials out there that help you get started with Kubernetes. However, most of them focus on showing how the commands work and how to get stuff running.

Don't get me wrong, trying out the commands is important, but to really get into working productively with Kubernetes you'll need to go deeper and understand the concepts and their functionalities so you can actually use them in the way they were intended.

This is especially hard coming from vanilla Docker, as the concepts of Docker don't directly translate to Kubernetes - at least if you want to use Kubernetes the right way.

Puja Abbassi

Developer Advocate at Giant Swarm



An Introduction to Pods, Labels, and Replicas

Let's start with the most basic concepts: Pods, Labels & Selectors, and ReplicaSets. I won't include any usage instructions, but focus solely on explaining the concepts, their functionality, and what you should use them for. These are only the most basic concepts and you'll need some more to get a full overview of Kubernetes. But rest assured I will explore further concepts in following chapters.

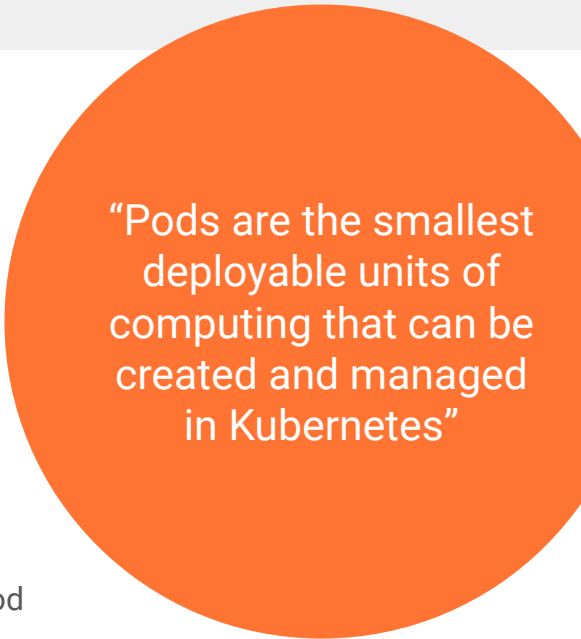
Pods

"Pods are the smallest deployable units of computing that can be created and managed in Kubernetes." say the official Kubernetes docs for Pods. This sometimes leads to the confusion that Pods are single containers, as that's what people are used to from Docker. While Pods can contain one single container, they are not limited to one and can contain as many containers as needed.

What makes these containers a Pod, is that all containers in a Pod run (nearly) as if they would have been running on a single host in pre-container world. Thus, they share a set of Linux namespaces and do not run isolated from each other. This results in them sharing an IP address and port space, and being able to find each other over localhost or communicate over the IPC namespace. Furthermore, all containers in a Pod have access to shared volumes, that is they can mount and work on the same volumes if needed.

In order to gain all this functionality a Pod is a single deployable unit. Each single instance or as we call it replica of a pod (with all its containers) is always scheduled together.

Now, for the typical Docker user this concept is quite new. For some it might sound like going back from the isolated "one process, per container" to "deploying your whole LAMP stack together". However, this is not the intended use case for Pods. The main motivation for the Pod concept is supporting co-located, co-managed helper containers next to the application container. These include things like: logging or monitoring agents, backup tooling, data change watchers, event publishers, proxies, etc.. If you are not sure what to use Pods for in the beginning, you can for now use them with single containers like you might be used to from Docker.



"Pods are the smallest deployable units of computing that can be created and managed in Kubernetes"

The Pod is the most basic concept in Kubernetes. By itself, it is ephemeral and won't be rescheduled to a new node once it dies. If we want to keep one or more instances of a Pod alive we need another concept: ReplicaSets. But before that we need to understand what Labels and Selectors are.

Labels & Selectors

Labels are key/value pairs that can be attached to objects, such as Pods, but also any other object in Kubernetes, even nodes. They should be used to specify identifying attributes of objects that are meaningful and relevant to users. You can attach Labels to objects at creation time and modify them or add new ones later.

Labels can be used to organize and select subsets of objects. They are often used for example to identify releases (beta, stable), environments (dev, prod), or tiers (frontend, backend), but are flexible to support many other cases, too. They help get some order into the multi-dimensionality of modern deployment pipelines.

Labels are a key concept of Kubernetes as they are used together with selectors to manage objects or groups thereof. This is done without the actual need for any specific information about the object(s), not even the number of objects that exist.

Especially the fact that the number of objects is unknown should be kept in mind when working with Label Selectors. In general, you should expect many objects to carry the same Label(s).

Currently, there are two types of Selectors in Kubernetes: equality-based and set-based. The former use key value pairs to filter based on basic equality (or inequality). The latter are a bit more powerful and allow filtering keys according to a set of values.

Using Label Selectors a client or user can identify and subsequently manage a group of objects. This is the core grouping primitive of Kubernetes and used in many places. One example of its use is working with ReplicaSets.

ReplicaSets

As mentioned in this guide a pod by itself is ephemeral and won't be rescheduled if it gets deleted or the node it is running on goes down. This is where the ReplicaSet comes in and ensures that a specific number of pod instances (or replicas) are running at any given time. Thus, if you want your pod to stay alive you make sure you have an according ReplicaSet specifying at least one replica for that pod. The ReplicaSet then takes care of (re)scheduling your instances for you.

As indicated above the ReplicaSet ensures a specific number of replicas are running. By modifying the number of replicas in the set's definition you can scale your Pods up and down.

You can include the definition of the pod directly in the definition of the ReplicaSet, so you can manage them together. However, the ReplicaSet will manage all Pods that match its selector. You should in most cases refrain from manually creating such Pods and also take care that there's not other resources creating similarly labeled Pods as the ReplicaSet will think it created those pds and start managing them.

However, there is a higher level concept called a Deployment, which in most cases manages replica sets for you. Therefore, you usually won't need to create or manipulate ReplicaSet objects directly. It's still important knowing about this concept as without it you won't understand the specific workings of how Kubernetes will help you run and manage your applications. I will explain Deployments and the many features they bring with them in the next chapter. For now suffice to say that they will manage your replica sets for you.

Get Your Hands Dirty

Now that you learned a bit about the basic concepts you can read more about them and their usage in the official Kubernetes docs, try to play around with them and then maybe move on to a [more advanced example](#) and try it out on a [local environment](#).

Official Kubernetes Docs Reference:

- [Pods](#)
- [Labels & Selectors](#)
- [Replica Sets](#)

Deployments

The Imperative vs. The Declarative Way

The Imperative Way

The imperative way is usually the one you use to try out things and get something working. You manually tweak it and at some point it is to your liking. However, if you keep on using this imperative style for deploying and managing your software you will encounter several problems (even if you automate the steps).

- If you want to know what has been last deployed, you can only resort to checking the currently deployed version, which is not necessarily what was planned to be deployed.
- If you change something manually, someone else might overwrite your changes or they might get reverted. For example because you only started new Pods manually and didn't change the controller.
- Keeping track of how a service has developed over time is hard and needs to be documented somewhere separately. This is especially important when working in teams.
- You need several commands to get one service to the desired state. Even if these steps get automated, they might lead to different outcomes when run in different environments.

Thinking declaratively is an important best practice in the Kubernetes world.

The Declarative Way

The declarative way on the other hand is what you should come up with, once you go into actually deploying and managing your software - especially when going towards production or integrating with continuous delivery pipelines. You might have tried out stuff the imperative way before, but once you know how it should look like, you sit down and "make it official" by writing it into a declarative definition. This avoids the above-mentioned problems and even brings some added benefits.

- It makes you think and plan how you want things to look once they are running (plan the state).
- It describes the way things should look like when running and not the steps that are needed to get there (define the desired state not the process).
- Changes can be easily documented by keeping track of (or versioning) the declarative definition, which makes work and communication in teams easier, but is also a good/clean approach for single devs.

Deployments in Kubernetes

Deployments are the main primitive we have for deploying and managing our software in Kubernetes, so it is important to understand what they do and what you can use them for.

Before Deployments, there were Replication Controllers, which managed Pods and ensured a certain number of them were running. With deployments we moved to ReplicaSets, which replaced Replication Controllers later on.

We don't usually manage ReplicaSets, but they get managed by the Deployments we define. Thus, the chain is like following: Deployment -> ReplicaSet -> Pod(s). And we only have to take care of the first.

Additional to what ReplicaSets offer, Deployments give you declarative control over the update strategy used for the Pods. This replaces the old kubectl rolling-update way of updating, but offers the same flexibility in terms of defining maxSurge and maxUnavailable, i.e. how many additional and how many unavailable Pods are allowed. Defining this in a Deployment enables you to "spec once use many times", which helps even more when working in teams or managing a multitude of Deployments.

Deployments manage your updates for you. They even go as far as to check whether or not a new revision that is being rolled out is working and stop the rollout in case it is not.

You can additionally define a wait time (minReadySeconds) that a Pod needs to be ready without any of its containers crashing before it is considered available, which again helps against "bad updates" and gives your containers a bit more time to get ready for traffic.

Furthermore, Deployments keep a history of their revisions, which is used in rollback situations, as well as an event log, which you can use to audit releases and changes to your Deployment.

Getting Started With Deployments

You can learn the usage of Deployments by [reading up on them in the official documentation](#) on Deployments and then writing a manifest that works well for you. Try updating and rolling back your Deployment and play around with scaling it up and down.

Services

By now you should have a basic understanding of some of the fundamental primitives of Kubernetes. However, there are some concepts missing, still.

One of the very central ones, especially when working with microservices, sets an abstraction layer on top of Pods (and other services) so you can communicate with them without having to track every single Pod as it starts, dies, and gets rescheduled. It is a basic building block for service discovery in Kubernetes.

Enter Services

As you might have heard (or read) me saying before, in the Kubernetes community it is sometimes hard to find the best way to do things. Furthermore, a lot of beginner tutorials let users run a list of kubectl commands to run and manage pods on Kubernetes. This, at least in parts, results in an imperative form of managing software. It involves manual work and can be pretty opaque if you work in a team.

Mainly to make Pod management easier and more controlled, the declarative Deployment object was added in Kubernetes 1.2. However, it took quite a while for people to commonly use it and it just recently (with Kubernetes 1.9) graduated from the beta API.

Services provide an abstraction layer and are an important concept to understand.



How Services Work

Services work by defining a logical set of Pods and a policy by which to access them. The selection of Pods is based on Label Selectors (which we talked about in the first chapter). In case you select multiple Pods, the Service automatically takes care of simple load balancing and assigns them a single virtual Service IP (which you can also set manually).

You can use the Selector to choose a group of Pods and define a targetPort on which to access them. Further helping with abstraction, this targetPort can also refer to the name of a port, which gives you more freedom to implement the actual Pods behind the Service. Even the port of each Pod could be different as long as they carry the same name.

This comes in handy when you abstract away certain kinds of backends that differ between stages. You can for example abstract away a database behind a Service. This way you can then use a simple local database for your development environment and a professionally managed database cluster in production without having to change the way that your other services talk to that database service.

You can also use Services if some of your workloads are running outside of your Kubernetes cluster, i.e. either on another Kubernetes cluster (or namespace) or even completely outside of Kubernetes. The latter is especially interesting if you are just starting to migrate workloads.

Get started with Services

Now that you know a bit more about the concept of Services, you should read up on their usage in the [official documentation](#). Then go ahead and build some Deployments that talk to each other over Services.

Secrets and ConfigMaps

In Kubernetes we have two separate primitives for storing configuration information. The first is Secrets, which as the name suggest is for storing sensitive information. The second one is ConfigMaps, which you can use for storing general configuration.

Secrets & ConfigMaps are quite similar in usage and support a variety of use cases.

Secrets and ConfigMaps

In the 12 factor app methodology, the third factor is called Config and describes why you should store configuration in the environment instead of in your code. This is based on the fact that an application's configuration can change between environments (e.g. development, staging, production, etc.) and that you want your applications to be portable. Thus, you should store the config outside of the application itself.

Now with Docker and containers, this means we should try to keep configuration out of the container image. This is even more needed when working with sensitive information, such as passwords, keys, auth tokens, because we might not want them to be available in a registry, even if that registry is private.

In Docker we would used to use `--env` or `--env-file` for both configuration and sensitive information. For the latter, there's the `docker secret` command in newer versions of Docker.



Secrets

Secrets can (and should) be used for storing small amounts (less than 1MB each) of sensitive information like passwords, keys, tokens, etc.. Kubernetes creates and uses some secrets automatically (e.g. Service Account tokens for accessing the API from a Pod). You can also create your own easily. Using Secrets is quite straightforward. You reference them in a Pod and can then use them either as files at your specified mount points or as environment variables in your Pod. Keep in mind that each container in your Pod that needs to access the Secret needs to request it explicitly. There's no implicit sharing of Secrets inside the Pod.

There's also a special type of Secret called `imagePullSecrets`. Using these you can pass a Docker (or other) container image registry login to the kubelet, so it can pull a private image for your Pod.

When updating Secrets that are used by already running Pods you need to be careful, as running Pods won't automatically pull the updated Secret. You need to explicitly update your Pods (for example using the rolling update functionality of Deployments explained above or by restarting/recreating them).

Further keep in mind that a Secret is namespaced, i.e. lives in a specific namespace, and only Pods in the same namespace can access that Secret.

A Note on the Security of Secrets

Secrets are kept in tmpfs and only on nodes that run Pods that use those Secrets. The tmpfs keeps Secrets from coming to rest on the node. However, they are transmitted to and from the API server in plain text, thus, be sure to have SSL/TLS protected connections between user and API server, but also between API server and kubelets.

To further increase security for secrets you can (and should) choose to encrypt them at rest (in etcd). For another added layer of security, you should enable Node Authorization in Kubernetes, so that a kubelet can only request Secrets of Pods pertaining to its node. This decreases the blast radius of a security breach on a node.

ConfigMaps

ConfigMaps are similar to Secrets, only that they are designed to more conveniently support working with strings that do not contain sensitive information. They can be used to store individual properties in form of key-value pairs. However, the values can also be entire config files or JSON blobs to store more information.

This configuration data can then be used as:

- Environment variables
- Command-line arguments for a container
- Config files in a volume
- Good use cases for ConfigMaps are for example storing config files for tools like redis or prometheus. This way you can change the configuration of these without having to rebuild the container.

A difference to the Secrets concept is that ConfigMaps actually get updated without the need to restart the Pods that use them. However, depending on how you use the configs provided, you might need to reload the configs, e.g. with an API call to prometheus to reload. This is often done through a sidecar container in the same Pod watching for changes of the config file.

Towards more portable containers

ConfigMaps are similar to Secrets, only that they are designed Once, you're using Secrets and ConfigMaps, it's easy to differentiate between environments like dev, test, and prod. You can just use different Secrets and ConfigMaps to configure your containers for the respective environment.

These two concepts also make your containers more versatile in that they keep out some of the specificities and let different users deploy them in different ways. Thus, you can foster better re-use between teams or even outside of your organization.

Secrets (and in some use cases also ConfigMaps) are especially helpful when sharing with other teams and organizations or even more when sharing publicly. You can freely share your images (and manifests), maybe even keep them in a public repository, without having to worry about any company-specific or sensitive data being published.

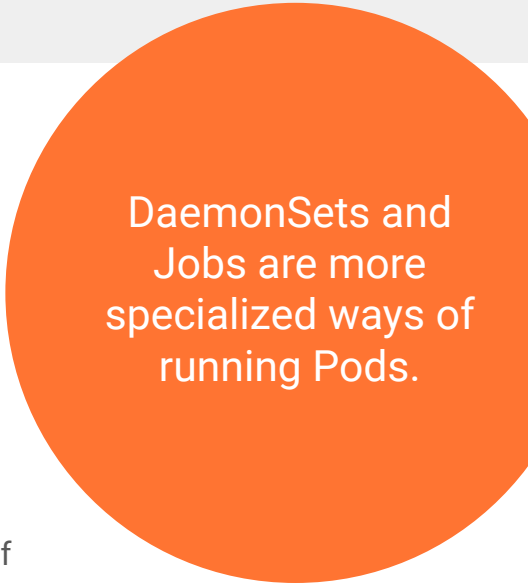
DaemonSets and Jobs

In the previous chapters we looked at the basics of how to run our applications as Pods in Kubernetes. There's two more ways to run Pods that are a bit more specialized. One is the DaemonSet and the other is called Jobs.

Daemon Sets

As the name suggests you can use DaemonSets for running daemons (and other tools) that need to run on all nodes of a cluster. These can be things like cluster storage daemons (e.g. Quobyte, glusterd, ceph, etc.), log collectors (e.g. fluentd or filebeat), or monitoring daemons (e.g. Prometheus Node Exporter, collectd, New Relic agent, etc.) The simplest use case is deploying a daemon to all nodes. However, you might want to split that up to multiple daemon sets for example if you have a cluster with nodes of varying hardware, which might need adaptation in the memory and/or cpu requests you might include for the daemon.

In other cases you might want different logging, monitoring, or storage solutions on different nodes of your cluster. For these cases where you want to deploy the daemons only to a specific set of nodes instead of all, you can use a nodeSelector to specify a subset of nodes for the DaemonSet. Note that for this to work you need to have labeled your nodes accordingly.



DaemonSets and Jobs are more specialized ways of running Pods.

There are four ways to communicate with your daemons

Push

The Pods are configured to push data to a service, so they do not have clients that need to find them.

NodeIP and known port

The Pods use a hostPort and clients can access them via this port on each NodeIP (in the range of nodes they are deployed to).

DNS

The Pods can be reached through a headless service by either using the endpoints resource or getting multiple A records from DNS.

Service

The Pods are reachable through a standard service. Clients can access a daemon on a random node using that service. Note that this option doesn't offer a way to reach a specific node..

Jobs

Unlike the typical Pod that you use for long-running processes, Jobs let you manage Pods that are supposed to terminate and not be restarted. A Job creates one or more Pods and ensures a specified number of them terminate with success.

You can use Jobs for the typical batch-job (e.g. a backup of a database), but also for workers that need to work off a certain queue (e.g. image or video converters).

Non-parallel Jobs

For non-parallel Jobs usually only one Pod gets started and the Job is considered complete once the Pod terminates successfully. If the Pod fails another one gets started in its place.

Parallel Jobs with a fixed completion count

For parallel Jobs with a fixed completion count the Job is complete when there is one successful Pod for each value between 1 and the number of completions specified.

Parallel Jobs with a work queue

For parallel Jobs with a work queue, you need to take care that no Pod terminates with success unless the work queue is empty. That is, even if the worker did its job it should only terminate successfully if it knows that all its peers are also done. Once a Pod exits with success, then all other Pods should also be exited or in the process of exiting.

For parallel Jobs you can define the requested parallelism. By default it is set to 1 (only a single Pod at any time). If parallelism is set to 0, the job is basically paused until it is increased. Keep in mind that parallel Jobs are not designed to support use cases that need closely-communicating parallel processes like for example in scientific computations, but rather for working off a specific amount of work that can be parallelized.

CronJobs

If you want a Job to be scheduled once or repeatedly at a specific point in time you can use CronJobs. These are similar to Jobs, with the addition of a schedule in Cron format

The End?

While this guide should give you a good first understanding of the basic concepts and get you going with your cluster, this doesn't mean that having read this makes you a Kubernetes master.

First, this guide only explains the basic concepts and does not go into detail on all options and features each of these primitives hold. Kubernetes resources are often times very complex so it is good to start with just the basics, but you might need to go deeper with time and make use of more features to get your applications deployed.

Second, the primitives introduced do not cover the whole range of resources available in Kubernetes. There's many more and with new versions even new ones might get added. All resources are there for a reason and often times knowing about them makes your life easier in the long run, when having to manage more complex workloads.

Furthermore, just reading this guide and maybe looking through the Kubernetes documentation and blog posts gives you a good foundation. However, you need to actually go and try it out and find ways to use the primitives to run and manage actual applications to get proficient in their usage. And there's no excuse that you don't have a cluster at hand, just try it out on a local environment.