Kubernetes Quick Guide for Developers

The Elements Of Orchestration

by Sandra Parsick

Get started on automated deployments via Kubernetes as your orchestration platform. Sandra Parsick addresses components and concepts from a developer's perspective. Learn to handle your first deployment and run a web application on Kubernetes.

- Kubernetes: Components and Concepts
- Deploying web applications on Kubernetes

A Kubernetes Quick Start for Developers

Kubernetes: Components and Concepts

Kubernetes is omnipresent; either applications are already running on Kubernetes, or they are intended to in the future. Developers might think, “But what does the Kubernetes infrastructure have to do with me?” With the introduction of the platform, however, they are often tasked with writing deployment scripts in the form of Kubernetes descriptors alongside the application code. For this, they need a basic understanding of the components. So here is an introduction.

We will go through components and concepts mainly from the perspective of a developer who wants to deploy her web application on Kubernetes. The series consists of three parts: We start by explaining what Kubernetes is, provide an overview of which Kubernetes components developers need to handle for deployment, and what prerequisites they must establish. The second part introduces the most important Kubernetes components a developer needs to deploy their web application on Kubernetes so that the outside world can communicate with the application. The third and final part of the series describes additional useful Kubernetes components.

Kubernetes: What Is It, and How Does It Work?

Kubernetes is a platform that allows operations to run applications in a standardized way. Kubernetes describes itself as an open-source system where applications in containers can be deployed, scaled, and managed automatically.

We can think of Kubernetes as an abstraction layer over many different machines that provides a unified interface for deploying our application. The platform handles the distribution and management of the application based on the information we provide when calling the interface. For this purpose, Kubernetes divides the machines assigned to it into two groups (Figure 1):

- Control Plane
- Nodes

TRACK

Build Once. Scale Everywhere. Real Developer Experience

PLATFORM ENGINEERING SUMMIT by DevOpsCon

What You'll Learn About Platform Engineering:

- **Treat internal platforms as products** to align with developer needs and deliver measurable DevEx outcomes
- **Reduce cognitive load** using modular design, workflow mapping, and user-centric architectures
- **Build scalable IDPs** with CNCF-backed tools and reusable Infrastructure as Code (IaC) templates
- **Automate security and compliance** in CI/CD pipelines using governance automation, policy-as-code, and secrets management
- **Measure and prove platform ROI** with DevEx metrics, adoption data, and iterative feedback loops
- **Apply proven platform patterns from** real-world enterprise-scale architectures
- **Align platform strategy** with AI governance through MLOps pipelines and secure RAG systems

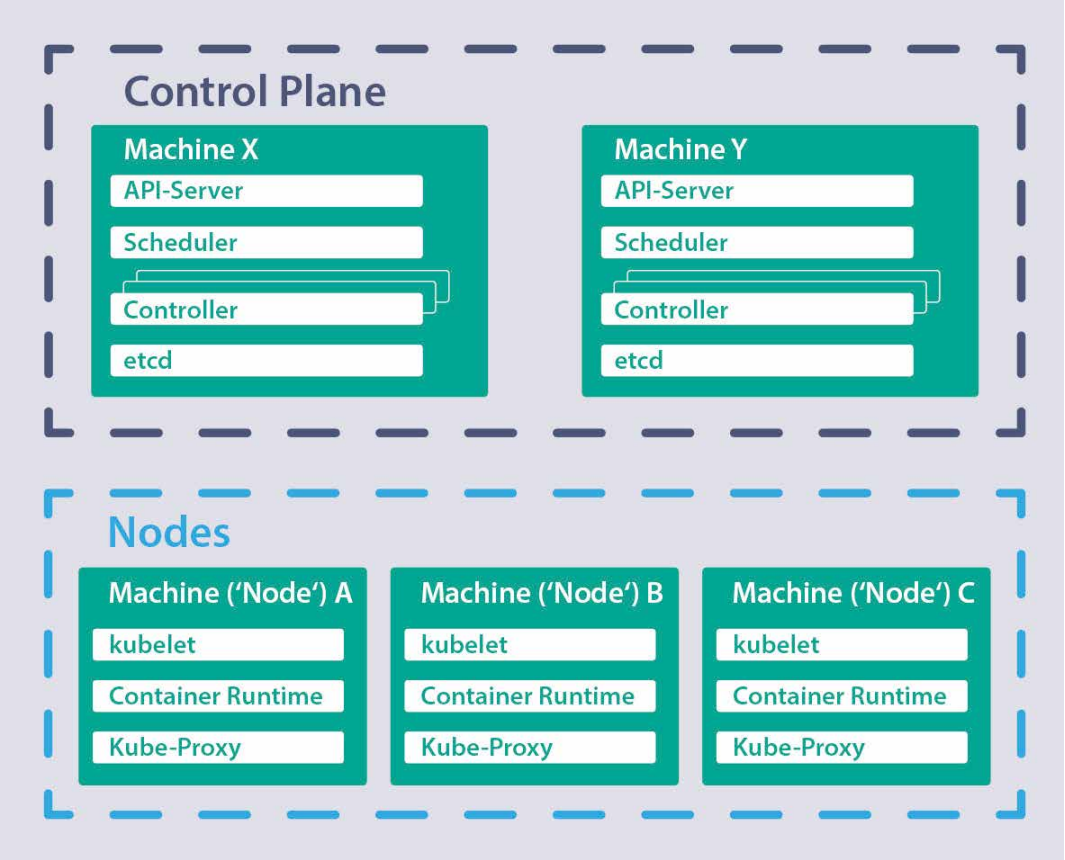


FIGURE 1 Distribution of Machines

The machines in the Control Plane group are responsible for managing and distributing applications to the machines in the Nodes group. For this, the operations team installs the following components on each device in the Control Plane group: API server, scheduler, various controllers, and etcd. In theory, it is sufficient to assign only one machine to this group. In practice, there should be at least two to increase fault tolerance.

The machines in the Nodes group run the actual applications. Each machine is also referred to as a Node. To allow applications to be distributed to them, the operations team installs the components kubelet, container runtime, and kube-proxy on them.

A developer deploys an application to the cluster by sending a description to the API server component, which informs the other components what is needed to run the application in the cluster (Figure 2).

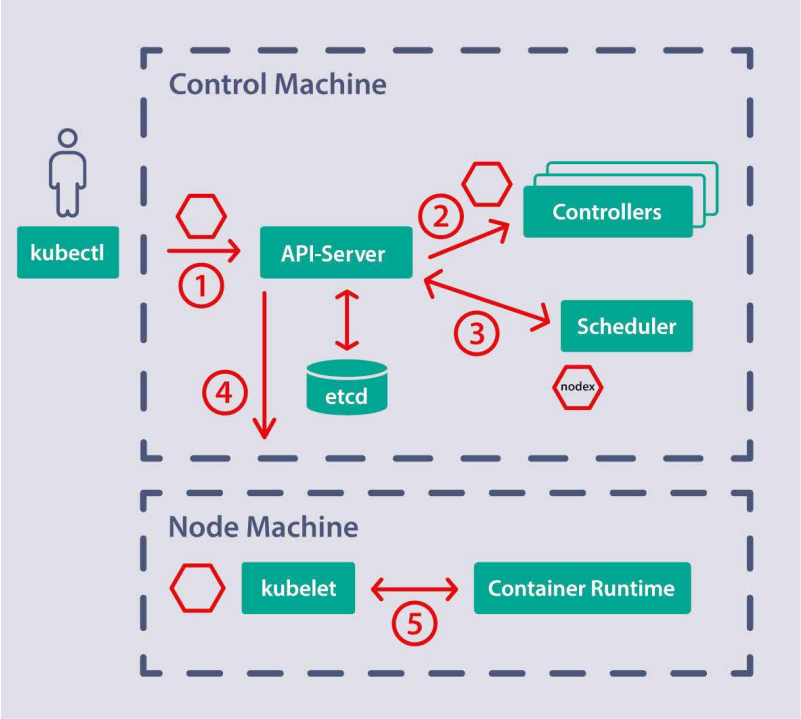


FIGURE 2 This is how the creation of Kubernetes objects works

The API server decides, based on the received description, which controller to forward the request to. Additionally, it asks the scheduler which node the object can run on. The scheduler, in turn, decides, based on various criteria (e.g., CPU and RAM usage), which node the pod and containers are assigned to. The scheduler then augments the object with information about which node it should run on.

With this information, the API server contacts kubelet on the selected node, which ensures that the object runs on that node. If the objects contain container information, kubelet passes this information to the container runtime, which manages the container lifecycle on the node.

Communication with the Kubernetes cluster is handled via the command-line tool kubectl – a wrapper around defined REST calls to the API server. The description is done using Kubernetes descriptors in YAML format.

Using descriptors, the developer informs the cluster which Kubernetes objects the components in the cluster must create for the application to run.

Each descriptor is divided into four areas:

- apiVersion defines the API version for the described Kubernetes object.
- kind defines which Kubernetes object should be created.
- metadata contains metadata such as the name of the Kubernetes object.
- spec specifies the Kubernetes object defined under kind. It is essentially the payload of the REST call.

Listing 1

```
[source, yaml]
---
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-demo
spec:
---
```

kubectl extracts from the description in Listing 1 which API (apiVersion and kind) it must call in the cluster and sends it the spec description, which informs it which Kubernetes objects to create.

Why Should I, as a Developer, Deal with Kubernetes Objects?

The previous introduction to Kubernetes is very operations-focused, and a developer might justifiably ask why they should engage more closely with Kubernetes. Ideally, they would like to focus on application development and use Kubernetes as a black box.
However, for Kubernetes to know how to best distribute a given application across its assigned machines, it needs information to decide how and where to run the application.

TRACK

DevOpsCon

Kubernetes & Cloud Native:
Containerize for a cloud-native approach.

Master containerization fundamentals, deployment strategies, and advanced orchestration to manage and scale containerized apps with tools like Kubernetes, Docker Swarm, and Apache Mesos. Gain expertise in running secure, cloud-native applications across public clouds, private clouds, or on-premises environments.

Learn from Industry Leaders about:

- **Containerization Basics:** Create and manage containers with Docker and Containerd for cloud-native deployments.
- **Kubernetes Core Concepts:** Master Kubernetes for orchestrating and scaling containerized applications.
- **Advanced Orchestration:** Explore multi-cluster management and high-availability setups with Kubernetes, Docker Swarm, and Apache Mesos
- **Deployment Techniques:** Learn rolling updates, blue-green, and canary deployments for containers.
- **Security Essentials:** Implement strategies to secure containerized environments across cloud and on-premises.
- **Scaling Containers:** Use Kubernetes Autoscaler and KEDA for smart, demand-driven scaling.

The members of the development team are those who best know what the application they are creating does and what requirements it has for its environment. Depending on the application's purpose, it requires different Kubernetes objects to run well in the Kubernetes cluster.

Of course, a developer could take the position that an operations team exists to provide the Kubernetes objects. This applies to objects for which the development team has no information or objects shared with other teams (e.g., network settings, storage requirements). However, for objects where the team already needs to provide information for proper configuration, the team can just create them themselves, with support from the Kubernetes operations team.

The Kubernetes descriptor is the ideal tool for Dev and Ops teams to communicate about the deployment and the needs of an application. Without this exchange, we encounter the familiar and frustrating problems caused by silos between Dev and Ops departments.

Kubernetes Objects Needed from an Application Perspective

Next, we look at web applications on their way into the cluster and which Kubernetes objects are needed. The example application consists of a deployable artifact that provides an HTTP interface outside the cluster or a server-side rendered interface. In this section, we provide a rough overview of the components (Figure 3). In Part 2 of the series, we present the components in detail.

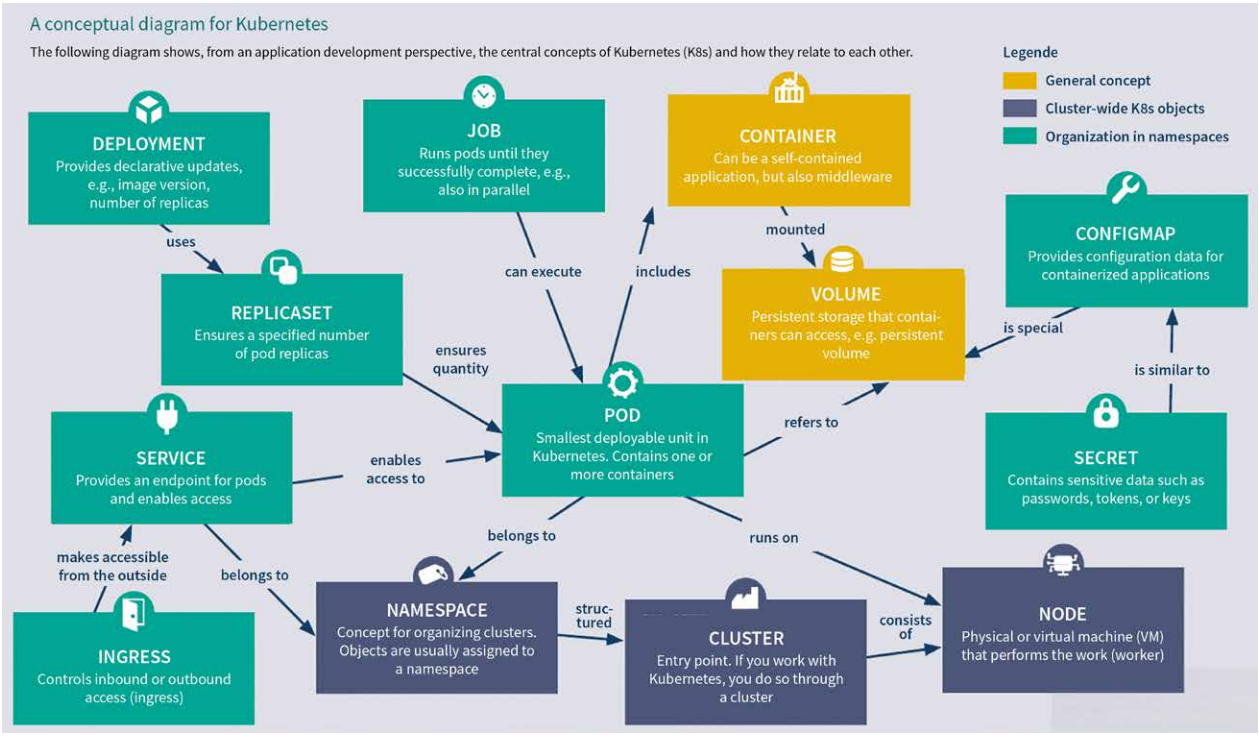


FIGURE 3 Overview of Components

The smallest unit in a Kubernetes cluster is the Pod object. It serves as a wrapper around a container in which the application runs. Therefore, the developer must first ensure that the application is packaged into a container. We will look at this more closely later.

In the Pod, the developer defines which resources the application requires, such as CPU, RAM, or disk space. With this information, the scheduler can decide which node is best suited to run the application (Figure 4).

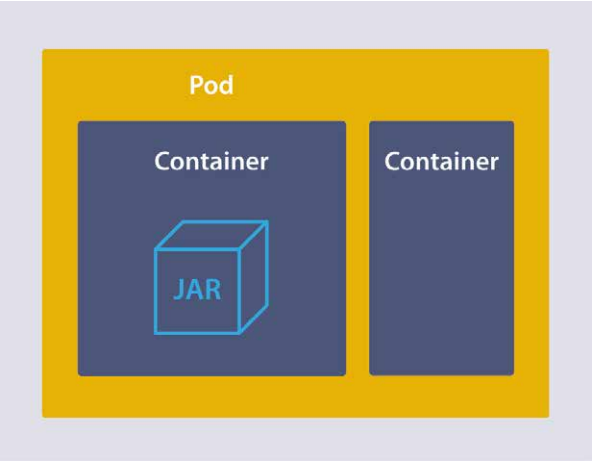


FIGURE 4 The Pod

Thanks to the Pod object, the application runs in Kubernetes but is not accessible inside or outside the cluster. The Service Kubernetes object is responsible for this. Its task is to represent a set of Pods offering the same service under a stable and consistent address (Figure 5).

The Service object can be used in different modes. For example, it can make the service available via a consistent address only within the cluster or via a consistent address outside the cluster.

The latter approach is sufficient when running few applications on the cluster and exposing them outside. However, when running many applications, this approach quickly reaches its limits, as each service receives its own public IP address, complicating IP management.

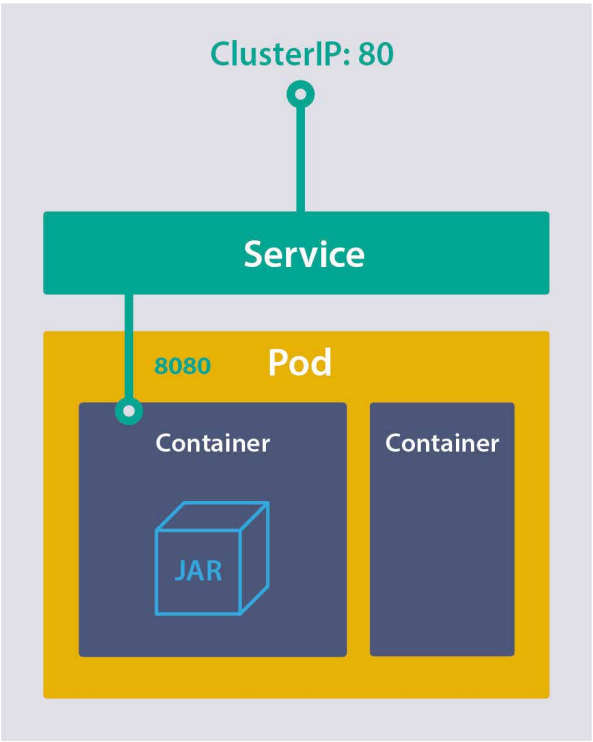


FIGURE 5 The Service

With the Ingress Kubernetes object, we can make all applications in the cluster accessible from outside via a single IP address. Ingress acts as a load balancer and reverse proxy, routing HTTP and HTTPS requests to the appropriate service based on rules (Figure 6).

Before we can look at the individual components in detail in Part 2, we need the foundation on which applications can run in a Kubernetes cluster: the container.

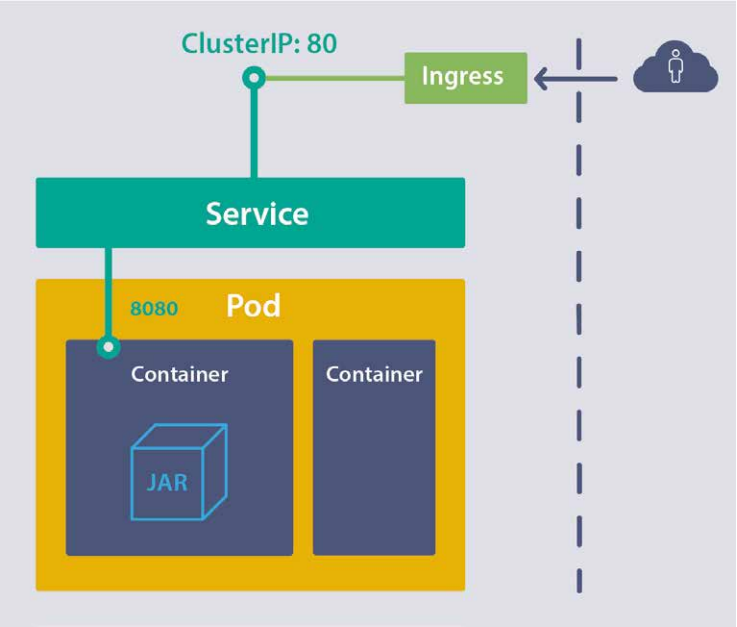


FIGURE 6 The Ingress Object

The Foundation: The Container

Although the smallest unit in Kubernetes is a Pod, the prerequisite for running an application in a Kubernetes cluster is the container. A Pod is a wrapper around one (or more) containers. But what exactly is a container?

Depending on the programming language the development team used, the application requires an environment at runtime that provides certain libraries or even entire runtime environments. On a traditional server, creating an environment that works for multiple applications can be challenging.

At the latest, when a developer wants to update the libraries, they must ensure that the applications are compatible with the new version. If even one application is incompatible, the environment cannot be updated.

There are, of course, solutions to this problem, but they are error-prone, and each programming ecosystem offers its own solutions, so combining them introduces additional complexity. Containers provide a unified solution.

Containers create a dedicated environment for each application, isolated from other processes at runtime. From the operating system's perspective, a container runs as a single process. Inside this container process runs the application process. Containers decouple applications from shared libraries, runtimes, and resources.

The technology is based on two Linux kernel features: namespaces and cgroups. A namespace is an isolated environment where a defined set of processes can access defined resources (files, network, CPU, etc.). It defines which resources processes can access. Cgroups define how many resources processes in a namespace are allowed to use.

The kernel manages all processes and resources in a tree structure. Thanks to namespaces, container implementations can simulate multiple processes on the host as one process, and inside the container process, it appears to the processes as if they are running in their own environment. To the kernel, these are all individual processes in a tree (Figure 7).

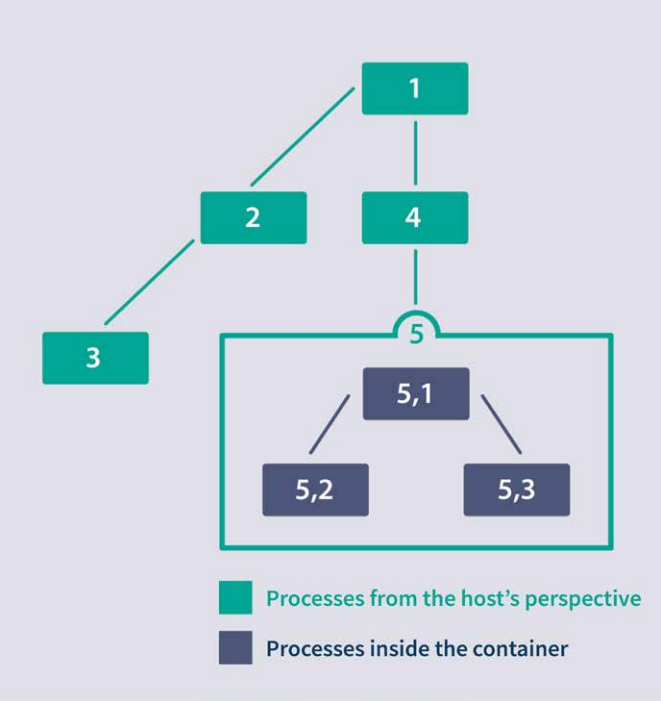


FIGURE 7 Container Processes

The next question a developer will ask is: How do I get my application into the container? The simplest way is to start a container process, install and run the application inside it, and "freeze" (commit) the process into a container image. This process is difficult to automate and reproduce. Therefore, it is possible to describe a container image using a build file (for Docker, the so-called Dockerfile) (Listing 2).

Listing 2

```
[source, docker]
----
FROM gcr.io/distroless/java21-debian12
WORKDIR /application
EXPOSE 8080
COPY target/*.jar application.jar
ENTRYPOINT ["java", "-jar", "application.jar"]
----
```

In the example, the developer references an existing container image and builds a new one based on it (FROM gcr.io/distroless/java21-debian12). In the following instructions, the developer describes how Docker should package the Java application into the container image (COPY copies a JAR file from the host into the container and the directory set with WORKDIR) and how the application should be executed at container runtime (ENTRYPOINT).

With this Dockerfile, the developer can build a container image (docker build Dockerfile -t myContainer) and start it (docker run myContainer).

There are many other ways to get an application into a container. These are described in the article “Container Images Deep Dive – 101 Ways to Build and Deploy” [1].

In the next chapter, we will discuss how the container is deployed on a K8s cluster.

Links & Literature

[1] Parsick, Sandra: „Container-Images Deep Dive – 101 Wege zum Bauen und Bereitstellen“: https://www.informatik-aktuell.de/entwicklung/methoden/container-images-deep-dive-101-wege-zum-bauen-und-bereitstellen.html



A Kubernetes Quick Start for Developers

Deploying web applications on Kubernetes

In the previous chapter, we looked at an overview of what Kubernetes is and which components are important for us as developers. Now, we will take a closer look at the individual components. A web application that provides an HTTP interface outside the cluster is used here as our example. Instead of the interface, it could also be a server-side rendered interface.

In the following sections, we will take a detailed look at the most important Kubernetes components that the application needs in order to run in the cluster and be accessible outside the cluster. Essentially, these are the Kubernetes components Pod, Service, and Ingress.

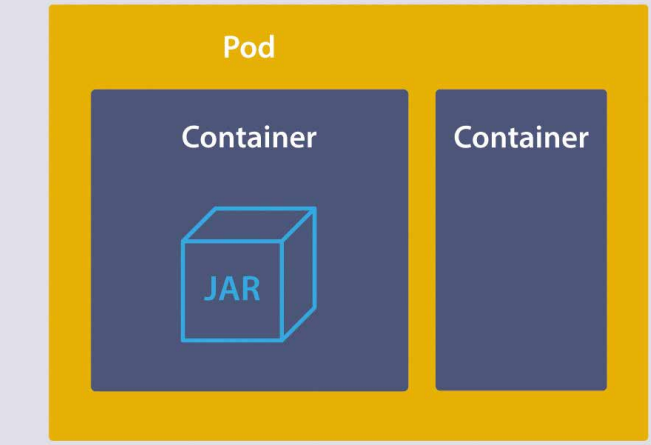


FIGURE 1 The Pod's object

The Pod

The smallest unit in a Kubernetes cluster is the pod object. A pod combines a number of containers. In our scenario, it is sufficient for us to consider a pod as a wrapper around a container (Listing 1). We looked at how the developer builds a container with the application in the first part of the series (Figure. 1).

Listing 1: Kubernetes descriptor for a pod with one container

```
[source, yaml]
----
apiVersion: v1
kind: Pod
metadata:
  name: single-pod-demo
spec:
  containers:
  - name: hero-app
    image: sparsick/spring-boot-demo:1.5.0
    ports:
    - containerPort: 8080
----
```

A descriptor is used to describe each Kubernetes object. The developer enters the following information in the descriptor for the Pod object (Listing 1): Under metadata.name, they specify the name of the Pod (here: single-pod-demo). This is part of the Pod metadata.

Under spec, they specify which containers this pod should serve as a wrapper for. In our case, it is for one container. Therefore, the containers section contains an entry that specifies the container. It consists of the name of the container (name: hero-app), the container image on which it is created (image: sparsick/spring-boot-demo:1.5.0), and the ports that the container is allowed to access (containerPort: 8080).

The developer has thus defined a minimalist pod that can been be deployed to the Kubernetes cluster:

```
[source, shell]
----
$ kubectl apply -f 01-single-pod.yaml
pod/single-pod-demo created
```

To ensure that the pod is also deployed on the cluster, it queries the cluster for the corresponding pods (Listing 2).

Listing 2: CLI command to view all deployed pods

```
source, shell]
----
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
single-pod-demo     1/1     Running   0          31s
----
```

It can take a little longer to start the application, especially for larger applications. The developer can provide a container in a pod with additional information. Kubernetes uses this information to determine whether the application in the container is ready. This is particularly helpful for web applications, as it allows Kubernetes to determine whether it can send requests to the pod. This information is called readiness probes.

In our case, the `httpGet` probe is a good choice. It sends an HTTP `GET` request to the container and uses the HTTP response status to determine whether the container is ready or not. The developer defines the readiness probes for each container in the pod definition.

In addition to the configurations of the individual readiness probes, the developer specifies the intervals at which the check should be performed (initialDelaySeconds, periodSeconds, and timeoutSeconds).

Besides readiness, the cluster also wants to know when an application is no longer running and when the cluster should restart it if necessary.

The liveness probe is responsible for this check. The liveness probe checks whether the application is still alive at the configured intervals. As soon as the timeout is reached and the check fails, the cluster restarts the pod. The developer configures it in the same way as the readiness probe.

Listing 3 : Pod definition with liveness and readiness probe source, shell]

```
[source,yaml]
----
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-probes-demo
spec:
  containers:
  - name: hero-app
    image: sparsick/spring-boot-demo:1.5.0
    ports:
    - name: container-http
      containerPort: 8080
      protocol: TCP
    livenessProbe:
      httpGet:
        path: /actuator/health/liveness
        port: container-http
      initialDelaySeconds: 15
      periodSeconds: 10
      timeoutSeconds: 30
    readinessProbe:
      httpGet:
        path: /actuator/health/readiness
        port: container-http
      initialDelaySeconds: 15
      periodSeconds: 10
      timeoutSeconds: 30
----
```

The developer can define how much CPU and memory each container in the pod will need. This helps the scheduler make a better decision about which node to deploy the pod and its containers to. They can define how much CPU and memory the container needs (requests) and what the absolute limit is (limits). The CPU and memory requirements and limits of a pod are then calculated based on the sum of the requests and limits for all containers in a pod.

In the container specification, the developer can specify the CPU and memory requirements (requests) and limits (limits) under resources (Listing 4).

Listing 4: Pod definition with information on resource requirements

```
[source,yaml]
----
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-probes
spec:
  containers:
  - name: hero-app
    image: sparsick/spring-boot-demo:1.5.0
    ports:
    - name: container-http
      containerPort: 8080
      protocol: TCP
   resources:
      requests:
        memory: “1Gi”
        cpu: “250m”
      limits:
        memory: “2Gi”
        cpu: “500m”
----
```

When the scheduler is tasked with distributing a new pod, it looks at both the CPU and memory requirements of the new pod and the requirements of the pods running on the nodes. It is interested in the requested requirements of the pods and not the currently used requirements (Figure 2).

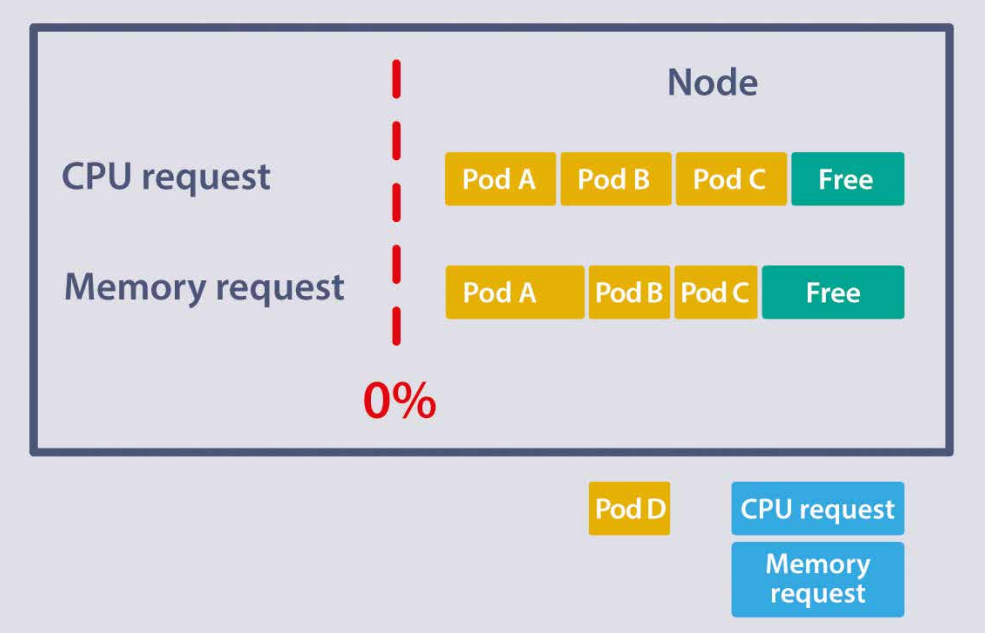


FIGURE 2
Resource Request

The developer specifies the maximum amount of CPU and memory that a container may consume with the limit specifications. If the container requires more CPU than specified in the limit, then Kubernetes throttles this container and the process in the container runs slower. If the container requires more memory than specified in the limit, then Kubernetes terminates the container with an "out of memory kill."

Now we can ask ourselves why we should specify resource requirements at all. If we do not specify any requirements, the containers will consume more resources than they need, thus increasing the costs for the Kubernetes cluster. Additionally, the resource specifications allow the development team to control the priority of the pods in the cluster (more on this in the Kubernetes documentation at [1]).

Service

Thanks to the Pod object, our application runs in Kubernetes. However, the application cannot be accessed either inside or outside the cluster. The Kubernetes Service object solves this problem. It represents a set of Pods that offer the same service under a stable and uniform address (Figure 3).

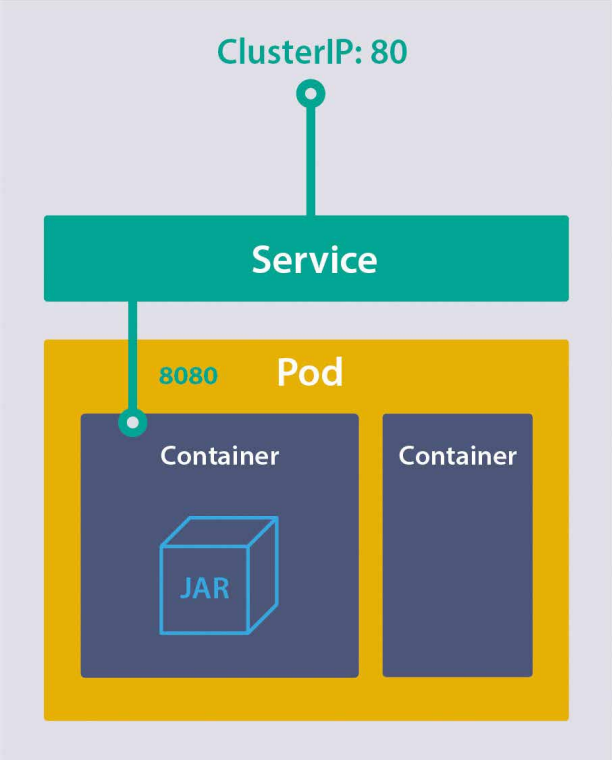


FIGURE 3 The Service Object

The service bundles all pods based on their labels (selector) and defines which port of the pod should be accessible under which port.

The developer can assign so-called labels to each Kubernetes object under metadata. These are self-defined key-value pairs under which Kubernetes objects can be grouped. Others (such as the service) can use these labels to access specific Kubernetes objects that are grouped under the same label.

In spec under ports, the developer defines which port of the pod (targetPort) should be accessible under which port in the service (port). In addition, the port should get a name (name) and it should also be determined which protocol (protocol) should be used to address it (Listing 5).

Listing 5: Kubernetes descriptor for the object service and for the pod with labels

```
[source, yaml]
----
apiVersion: v1
kind: Service
metadata:
  name: cluster-ip-demo-srv
spec:
  type: ClusterIP
  selector:
    app.kubernetes.io/name: hero
  ports:
  - name: hero-http
    protocol: TCP
    port: 80
    targetPort: 8080
---
apiVersion: v1
kind: Pod
metadata:
  name: spring-boot-demo
  labels:
     app.kubernetes.io/name: hero
spec:
  containers:
  - name: hero-app
    image: sparsick/spring-boot-demo:1.5.0
    ports:
    - name: container-http
      containerPort: 8080
      protocol: TCP
----
```

The service supports multiple operating modes (type). In our case, the erating mode ClusterIP is relevant. In this mode, the service is only acces-sible internally in the cluster under a uniform address (Listing 6).

Listing 6: CLI command for service deployment

```
[source, shell]
---
$ kubectl apply -f 05-pod-with-service-clusterip
pod/spring-boot-demo created
service/cluster-ip-demo-srv created
$ kubectl get service
NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)  AGE
kubernetes            ClusterIP   10.96.0.1       <none>        443/TCP  2d1h
cluster-ip-demo-srv   ClusterIP   10.104.173.54   <none>        80/TCP   77s
```

This means that if I start a temporary pod (Listing 7), I can access the application via the IP address listed under ClusterIP (Listing 6).

Listing 7: Launching a temporary pod

```
[source, shell]
----
$ kubectl run -it --rm --restart=Never busybox --image=busybox sh
f you don't see a command prompt, try pressing enter.
/ #
/ #
/ # wget http://10.104.173.54/hero -O -
Connecting to 10.104.173.54 (10.104.173.54:80)
writing to stdout
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
  <title>Hero Demo</title>
  <link rel="stylesheet" href="/webjars/bootstrap/5.2.3/css/bootstrap.min.
                                                                    css">
  <link rel="stylesheet" href="/webjars/font-awesome/6.3.0/css/all.css">
</head>
<body>
<main role="main" class="container">
  <h1>Hero Search Machine</h1>
  <div class="row-cols-2">
    <form action="/hero/list" method="get">
      <div class="input-group mb-3">
        <div class="input-group-prepend">
          <button class="btn btn-outline-secondary" type="submit"
                                                    id="button-search">
```

```
            <i class='fas fa-search'></i> Search
          </button>
        </div>
        <input type="text" class="form-control" name="search" id="search"
                                                              value=""/>
      </div>
    </form>
  </div>
  <div class="row-cols-2">
    <p>You find a new hero? Fill Expand database</p>
  </div>
  <div class="row-cols-6">
    <a id="button-add" role="button" class="btn btn-outline-secondary"
       href="/hero/new"><i class='fas fa-user-shield'></i> Add new hero</a>
  </div>
  <div class="row-cols-2">
    <p> Data are coming from <span>10.244.1.8</span></p>
  </div>
</main>
</body>
</html>
-                   100% |*********************************************
 ************************************************************|  1342  0:00:00
                                                              ETA
written to stdout
/ #
----
```

Ingress

If the development team wants to make the application accessible outside the cluster, it needs another Kubernetes component: Ingress. The Ingress object makes all applications in the cluster accessible from the outside via a single IP address. Ingress acts as a load balancer and reverse proxy, routing HTTP and HTTPS requests to the respective service based on rules (Figure 4).

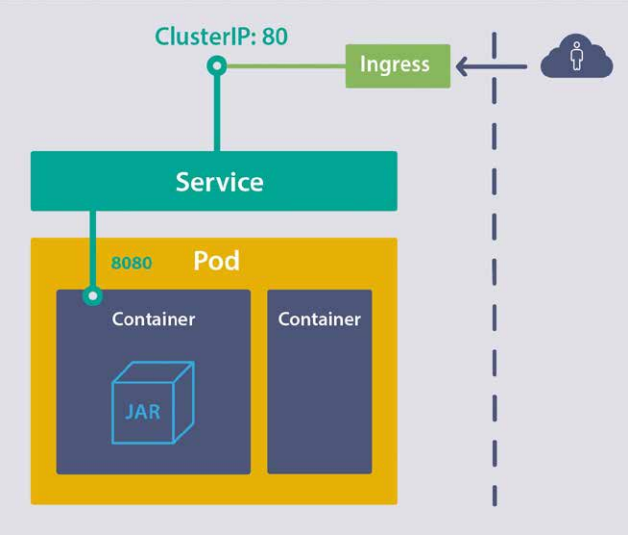


FIGURE 4 The Ingress Object

Listing 8: Kubernetes descriptor for Ingress

```
[source, yaml]
----
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: spring-demo-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/x-forwarded-prefix: “/”
spec:
  rules:
    - host: spring-boot-demo.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: spring-boot-demo-srv
                port:
                  number: 80
----
```

When deploying the object, an ingress controller handles the processing (Listing 8). There are different implementations of this controller that offer additional configuration options beyond the ingress specification. Therefore, the developer has the option of specifying implementation-specific configurations under metadata.annotations in the ingress object. In the example, the developer defines the rewrite rules for the ingress controller implementation nginx.

Under spec, the developer defines the forwarding rules. Each rule consists of an optional host specification (host) and a list of paths (http.paths). For each request, the ingress object analyzes the URL of the request (example: http://spring-boot-demo.local/hero). First, it checks whether there is a rule for the host (in the example, spring-boot-demo.local).

If the host is not explicitly specified in a rule, then that rule applies to all requests.

If the Ingress object finds a matching host rule, it then checks the associated path list to determine to which service (backend.service.name) and which port (backend.service.port.number or, alternatively, the port name defined in the service, backend.service.port.name) the request should be forwarded to. In the example, it is the service spring-boot-demo-srv.

The paths can be defined in different ways (pathType). The developer can define whether the URL path must exactly match the path rules (Exact) or whether only the prefix of the URL path must match the path rules (Prefix). The developer can also specify their own implementation for path determination (ImplementationSpecific). Details on the path rules can be found in the Kubernetes documentation [2].

We have now looked at the most important components that the development team needs to deploy a web application on Kubernetes. This should get us started on our own Kubernetes deployments.

TRACK

DevOpsCon

Kubernetes & Cloud Native:
Containerize for a cloud-native approach.

Master containerization fundamentals, deployment strategies, and advanced orchestration to manage and scale containerized apps with tools like Kubernetes, Docker Swarm, and Apache Mesos. Gain expertise in running secure, cloud-native applications across public clouds, private clouds, or on-premises environments.

Learn from Industry Leaders about:

- **Containerization Basics:** Create and manage containers with Docker and Containerd for cloud-native deployments.
- **Kubernetes Core Concepts:** Master Kubernetes for orchestrating and scaling containerized applications.
- **Advanced Orchestration:** Explore multi-cluster management and high-availability setups with Kubernetes, Docker Swarm, and Apache Mesos
- **Deployment Techniques:** Learn rolling updates, blue-green, and canary deployments for containers.
- **Security Essentials:** Implement strategies to secure containerized environments across cloud and on-premises.
- **Scaling Containers:** Use Kubernetes Autoscaler and KEDA for smart, demand-driven scaling.

Links & Literature

[1] The Kubernetes Documentation for Pod Quality of Service Classes: https://kubernetes.io/docs/concepts/workloads/pods/pod-qos

[2] The Kubernetes Documentation for Ingress: https://kubernetes.io/docs/concepts/services-networking/ingress



Sandra Parsick is a Java Champion and works as a freelance software developer and consultant in the Java environment. She has been involved in agile software development in various roles since 2008. She specialises in Java enterprise applications, the cloud, software craftsmanship and the automation of software development processes. She enjoys writing articles and speaking at conferences on these topics. In her spare time, Sandra Parsick is involved in various programme committees and community groups.

DevOpsCon

THE GLOBAL CONFERENCE SERIES FOR **CI/CD, KUBERNETES, PLATFORM ENGINEERING & DEVSECOPS**

