Stupid Simple

Kubernetes

by Zoltán Czakó

Welcome to Stupid Simple Kubernetes

In software development, the single constant is that everything changes fast. Good developers are always prepared for change: the framework you're working on today could be outdated in a few months. One way to prepare for change is to create loosely coupled, independent components that can be easily replaced.

As software and tools change, so do application architectures. Recently we've witnessed an evolution from traditional monolithic architecture, where all components of the application are packed as a single atonomous unit, to service-oriented architecture, to today's microservices architecture.

Microservices architectures have sprung up because of changes in tools, programming languages and development environments. To keep up with new technologies, you need a way to add new, independent components at any time. These components must be free to use whatever technology they like, so they can be built using different programming languages and frameworks, databases or even communication protocols.

In this e-book, we will show you how to build a stable, easily manageable, highly available microservices architecture. In the first part, we will introduce Kubernetes, its components and building blocks. Then, we will build a small sample application based on a microservices architecture. We'll define the Kubernetes scripts to create the Deployments, Services, Ingress Controllers and Persistent Volumes and then deploy this ecosystem in Azure Cloud.

In the second part of this book, we will dive into scalability and we will define different Kubernetes configuration files for Horizontal and Vertical Pod Autoscaling and also for Cluster Autoscaling.

In the last part of the book, we will present different solutions for easily handling all the cross-cutting concerns that we presented when using Service Meshes. We'll build our own Service Mesh using Envoy proxies and then use Istio to handle all these concerns automatically.

Ready to get started with Kubernetes? Let's go.

Chapter 1

Everything You Need to Know to Start Using Kubernetes 4

Chapter 2

Deployments, Services and Ingresses Explained 14

Chapter 3

Persistent Volumes Explained 25

Chapter 4

Device Plugins Explained 36

Chapter 5

Create an Azure Infrastructure for Microservices 40

Chapter 6

Stupid Simple Scability 50

Chapter 7

Stupid Simple Service Mesh - What, When, Why 57

Chapter 8

Stupid Simple Service Mesh in Kubernetes 66

Conclusion

Become a Microservices Master 77

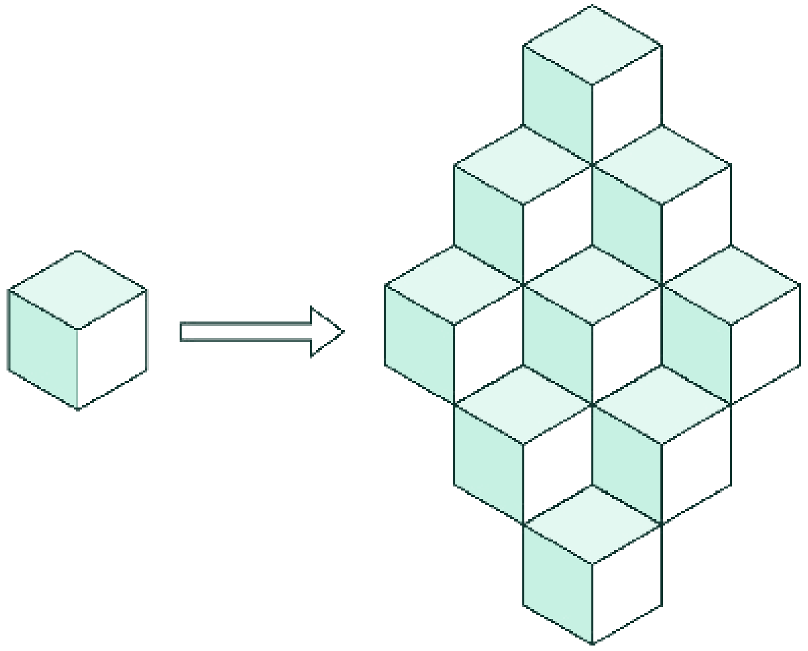
Chapter 1

Everything You Need to Know to Start Using Kubernetes

In the era of **Microservices**, **Cloud Computing** and **Serverless** architecture, it's useful to understand **Kubernetes** and learn how to use it. However, the official Kubernetes documentation can be hard to decipher, especially for newcomers. In this book, I will present a **simplified view of Kubernetes** and give examples of how to use it for deploying microservices using different cloud providers, including **Azure, Amazon, Google Cloud** and even **IBM**.

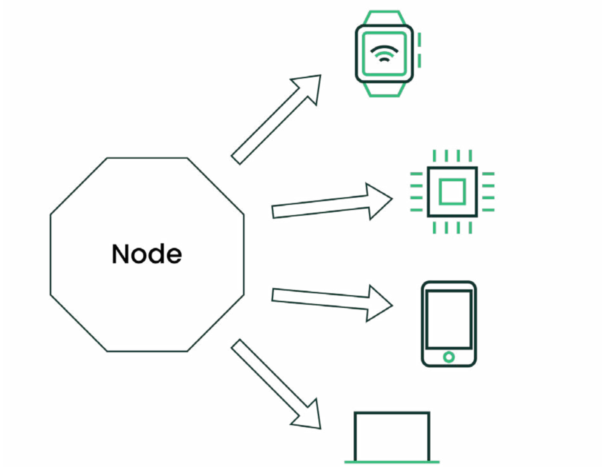
In this first chapter, we'll talk about the most important concepts used in Kubernetes. Later in the book, we'll learn how to **write configuration files**, use Helm as a package manager, **create a cloud infrastructure**, easily **orchestrate our services** using Kubernetes and create a **CI/CD pipeline** to automate the whole workflow. With this information, you can spin up any kind of project and create a solid infrastructure/architecture.

First, I'd like to mention that using containers has multiple benefits, from **increased deployment velocity** to delivery consistency with a **greater horizontal scale**. Even so, you should not use containers for everything because just putting any part of your application in a container comes with **overhead, like maintaining a container orchestration layer**. So, don't jump to conclusions. Instead, create a cost/benefit analysis at the start of the project.



Now, let's start our journey in the world of Kubernetes.

Kubernetes Hardware Structure



Nodes

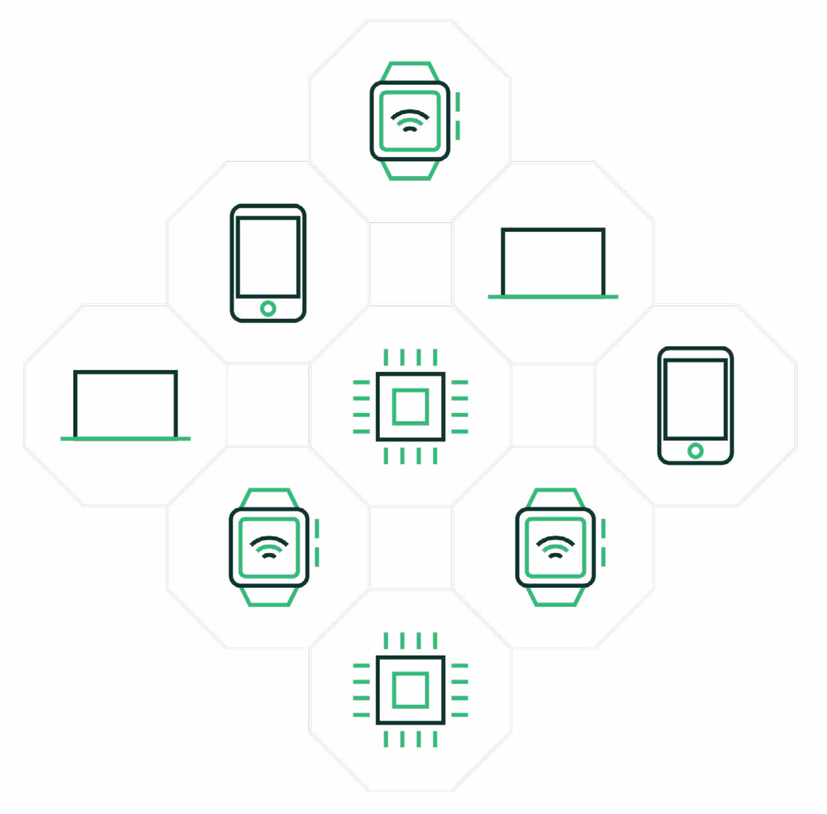
Nodes are worker machines in Kubernetes, which can be any device that has CPU and RAM. For example, a node can be anything, from a smartwatch, smartphone, or laptop to a Raspberry Pi. When we work with cloud providers, a node is a *virtual machine (VM)*. So, a node is an abstraction over a single device.

As you will see in the next chapter, the beauty of this abstraction is that we don't need to know the underlying hardware structure. We will just use nodes; this way, our infrastructure is platform independent.

Cluster

A cluster is **a group of nodes**. When you deploy programs onto the cluster, it automatically handles the distribution of work to the individual nodes. If more resources are required (for example, we need more memory), new nodes can be added to the cluster, and the work will be redistributed automatically.

We run our code on a cluster, and we shouldn't care about which node. The distribution of the workw is automatic.



Persistent Volumes

Because our code can be relocated from one node to another (for example, a node doesn't have enough memory, so the work is rescheduled on a different node with enough memory), **data saved on a node is volatile**. But there are cases when we want to save our data persistently. In this case, we should use Persistent Volumes. A persistent volume is like an **external hard drive**; you can plug it in and save your data on it.

Google developed Kubernetes as a **platform for stateless applications** with persistent data stored elsewhere. As the project matured, many organizations wanted to leverage it for their stateful applications, so the developers added persistent volume management. Much like the early days of virtualization, database servers are not typically the first group of servers to move into this new architecture. That's because the database is the core of many applications and may contain valuable information, so on-premises database systems still largely run in VMs or physical servers.

So, the question is, *when should we use Persistent Volumes?* To answer that question, first, we should understand the different types of database applications.

We can classify the data management solutions into two classes:

1. **Vertically scalable**—includes traditional RDMS solutions such as MySQL, PostgreSQL and SQL Server
2. **Horizontally scalable**—includes "NoSQL" solutions such as ElasticSearch or Hadoop-based solutions

Vertical scalable solutions like MySQL, Postgres and Microsoft SQL **should not go in containers**. These database platforms require high I/O, shared disks, block storage, etc., and do not (by design) handle the loss of a node in a cluster gracefully, which often happens in a container-based ecosystem.

For **horizontally scalable** applications (Elastic, Cassandra, Kafka, etc.), use containers. They can withstand the loss of a node in the database cluster, and the database application can independently rebalance.

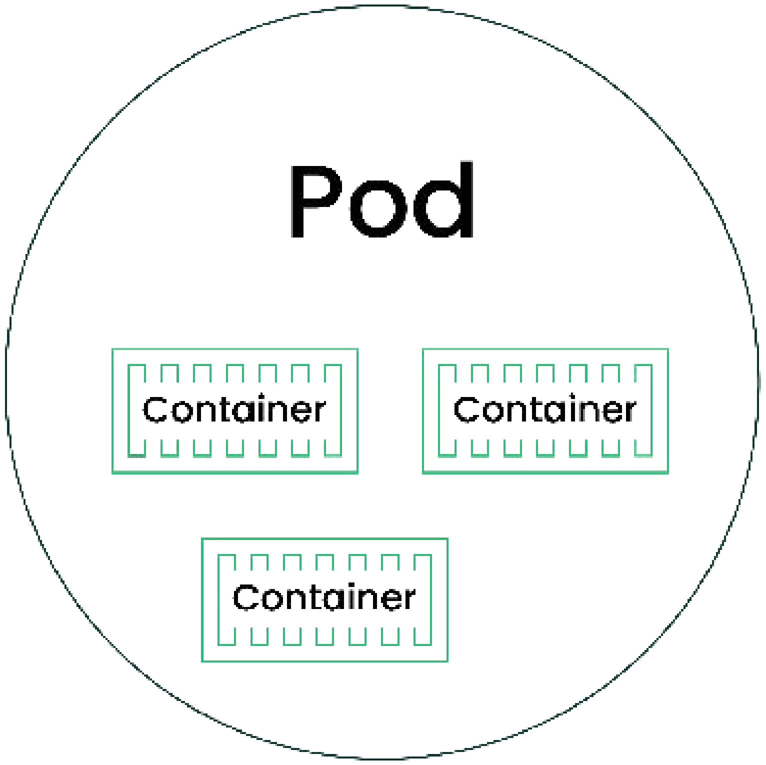
Usually, **you can and should containerize distributed databases that use redundant storage techniques and can withstand a node's loss** in the database cluster (ElasticSearch is a good example).

Kubernetes Software Components

Container

One of the goals of modern software development is to keep applications on the same host or cluster **isolated**. *Virtual machines* are one solution to this problem. But virtual machines require their own OS, so they are typically gigabytes in size.

Containers, by contrast, isolate application execution environments from one another but share the underlying OS kernel. So, a container is like a box where we store everything needed to run an application: code, runtime, system tools, system libraries, settings, etc. They're typically measured in megabytes, use far fewer resources than VMs and start up almost immediately.

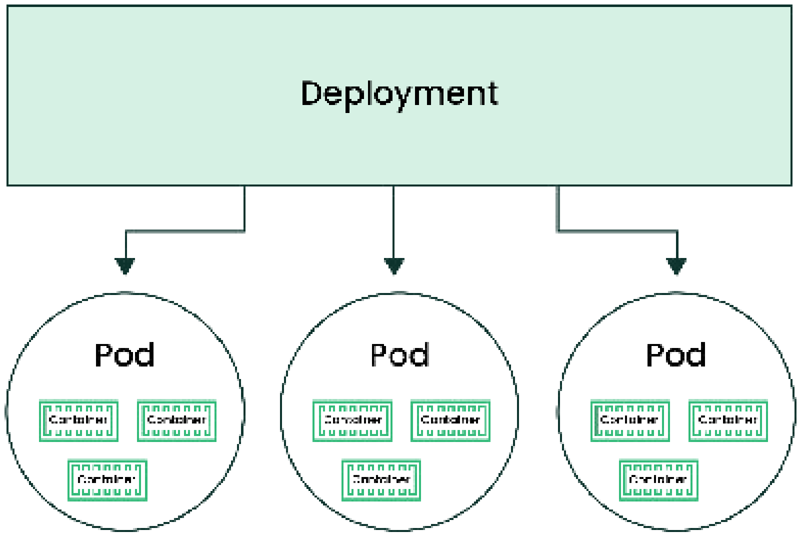


Pods

A pod is a **group of containers**. In Kubernetes, the **smallest unit of work** is a pod. A pod can contain multiples containers, but usually, we use one container per pod because the **replication unit in Kubernetes is the pod**. If we want to scale each container independently, we add one container in a pod.

Deployments

The primary role of **deployment** is to provide declarative updates to both the pod and the ReplicaSet (a set in which the same pod is replicated multiple times). Using the deployment, we can specify **how many replicas of the same pod should be running at any time**. The deployment is like a manager for the pods; it automatically spins up the number of pods requested, monitors the pods and recreates the pods in case of failure. Deployments are helpful because you don't have to create and manage each pod separately.



Stateful Sets

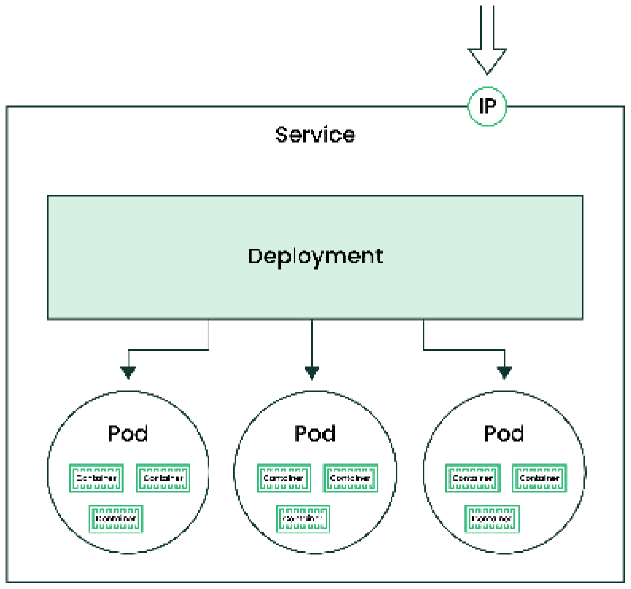
StatefulSet is a new concept in Kubernetes, and it is a resource used to manage stateful applications. It manages the deployment and scaling of a set of pods and guarantees these pods' ordering and uniqueness. It is similar to deployment; the only difference is that the deployment creates a set of pods with random pod names and the order of the pods is not important, while the StatefulSet creates pods with a unique naming convention and order. So, if you want to create three replicas of a pod called example, the StatefulSet will create pods with the following names: *example-0, example-1, example-2.* In this case, the most important benefit is that you can rely on the name of the pods.

DaemonSets

A **DaemonSet** ensures that the pod runs on all the nodes of the cluster. If a node is added/removed from a cluster, DaemonSet automatically adds/deletes the pod. This is useful for **monitoring** and **logging** because you can monitor every node and don't have to monitor the cluster manually.

Services

While deployment is responsible for keeping a set of pods running, the service is **responsible for enabling network access to a set of pods**. Services provide standardized features across the cluster: load balancing, service discovery between applications and zero-downtime application deployments. Each service has a unique IP address and a DNS hostname. Applications that consume a service can be manually configured to use either the IP address or the hostname and the traffic will be load balanced to the correct pods. In the ***External Traffic*** section, we will learn more about the service types and how we can communicate between our internal services and the external world.



ConfigMaps

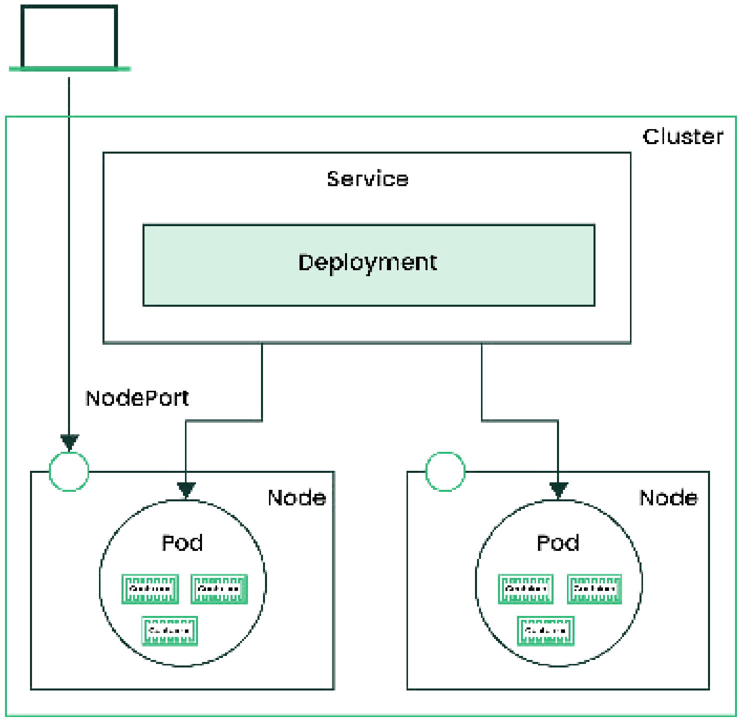
If you want to **deploy to multiple environments**, like staging, dev and prod, it's a bad practice to bake the configs into the application because of environmental differences. Ideally, you'll want to **separate configurations** to match the deploy environment. This is where ConfigMap comes into play. **ConfigMaps allow you to decouple configuration artifacts from image content** to keep containerized applications portable.

External Traffic

Now that you've got the services running in your cluster, how do you get external traffic into your cluster? There are three different service types for handling external traffic: **ClusterIP**, **NodePort** and **LoadBalancer**. The 4th solution is to add another layer of abstraction, called **Ingress Controller**.

ClusterIP

ClusterIP is the default service type in Kubernetes and lets you **communicate with other services inside your cluster**. While ClusterIP is **not meant for external access**, with a little hack using a proxy, external traffic can hit our service. Don't use this solution in production, but only for debugging. **Services declared as ClusterIP should NOT be directly visible from the outside**.

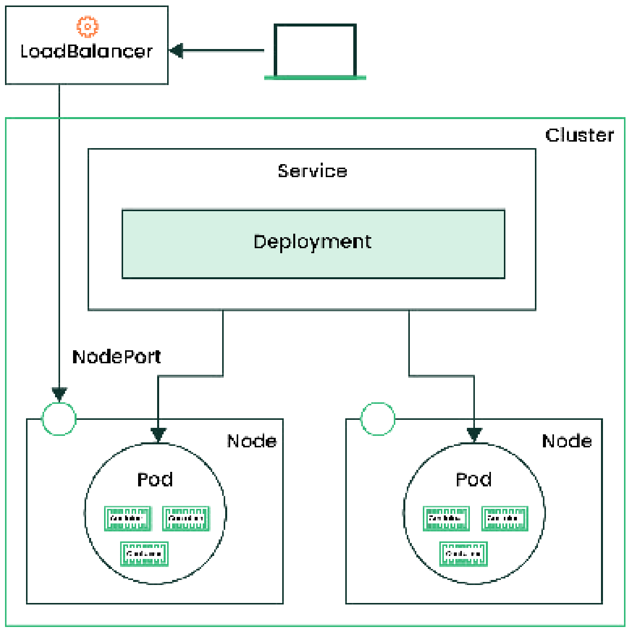


NodePort

As we saw in the first part of this chapter, pods are running on nodes. Nodes can be different devices, like laptops or virtual machines (when working in the cloud). Each node has a fixed IP address. By declaring a service as NodePort, the service will expose the **node's IP address** so that you can access it from the outside. You can use NodePort in production, but for large applications, where you have many services, manually managing all the different IP addresses can be cumbersome.

LoadBalancer

Declaring a service of type LoadBalancer exposes it externally **using a cloud provider's load balancer**. How the external load balancer routes traffic to the Service pods depends on the cluster provider. With this solution, you don't have to manage all the IP addresses of every node of the cluster, but you will have one load balancer per service. The downside is that every service has a separate load balancer and you will be billed per load balancer instance.

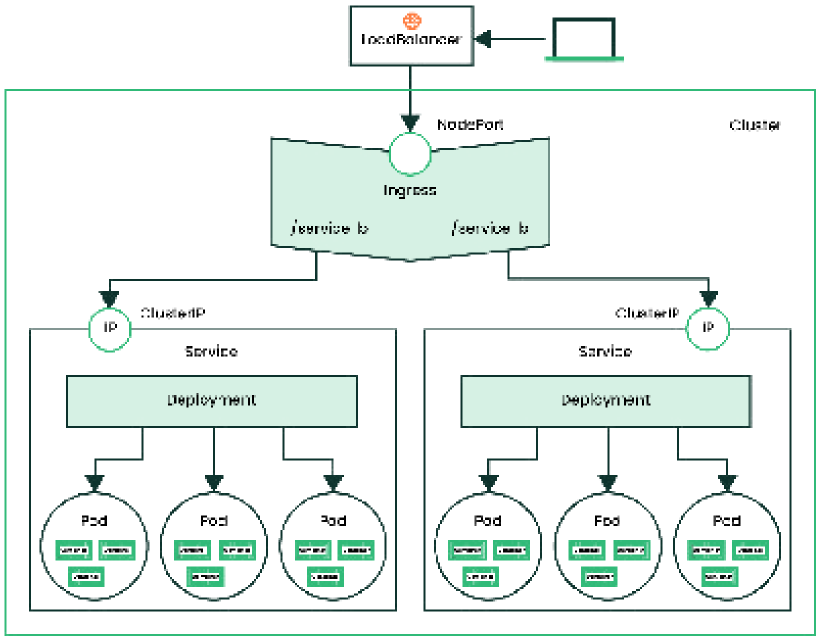


This solution is good for production, but it can be a little bit expensive. Let's look at a less expensive solution.

Ingress

Ingress is not a service but an API object that manages external access to a cluster's services. **It acts as a reverse proxy and single entry-point to your cluster that routes the request to different services**. I usually use NGINX Ingress Controller, which takes on reverse proxy while also functioning as SSL. The best production-ready solution to expose the ingress is to use a load balancer.

With this solution, you can expose any number of services using a single load balancer, so you can keep your bills as low as possible.



Next Steps

In this chapter, we learned about the **basic concepts used in Kubernetes** and its hardware structure. We also discussed the different software components including **Pods**, **Deployments**, **StatefulSets** and **Services**, and saw how to **communicate between services and with the outside world**.

In the **next chapter**, we'll set up a **cluster on Azure** and create an infrastructure with a **LoadBalancer**, **an Ingress Controller and two Services** and use **two Deployments** to spin up **three Pods** per Service.

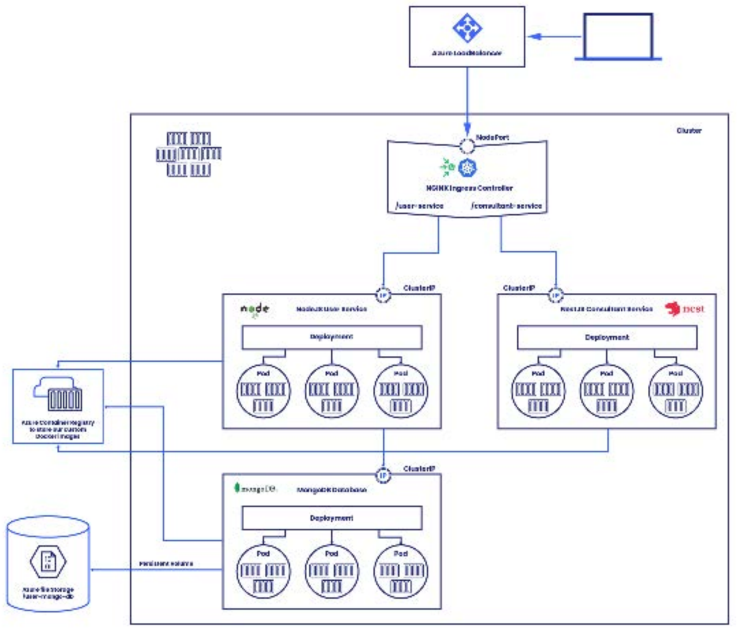
Chapter 2

Deployments, Services and Ingresses Explained

In the first chapter, we learned about the basic concepts used in Kubernetes, its hardware structure, the different software components like **Pods, Deployments, StatefulSets, Services, Ingresses and Persistent Volumes** and saw how to communicate between services and with the outside world.

In this chapter, we will:

- Create a **NodeJS backend** with a MongoDB database
- Write the **Dockerfile** to containerize our application
- Create the **Kubernetes Deployment scripts** to spin up the Pods
- Create the **Kubernetes Service scripts** to define the communication interface between the containers and the outside world
- Deploy an **Ingress Controller** for request routing
- Write the **Kubernetes Ingress scripts** to define the communication with the outside world.



Because our code can be relocated from one node to another (for example, a node doesn't have enough memory, so the work will be rescheduled on a different node with enough memory), **data saved on a node is volatile (so our MongoDB data will be volatile, too)**. In the next chapter, we will talk about the problem of data persistence and how to use **Kubernetes Persistent Volumes** to safely store our persistent data.

In this tutorial, we will use NGINX as an Ingress Controller and **Azure** Container Registry to store our custom Docker images. All the scripts written in this book can be found in my StupidSimpleKubernetes git repository. If you like it, **please leave a star!**

NOTE: the scripts are **platform agnostic**, so you can follow the tutorial using other types of cloud providers or a local cluster with K3s. I suggest using **K3s** because it is very lightweight, packed in a single binary less than 40MB. What's more, it's a highly available, certified Kubernetes distribution designed for production workloads in resource-constrained environments. For more information, you can take a look over its well-written and easy-to-follow **documentation**.

I would like to recommend another great article about basic Kubernetes concepts: *Explain By Example: Kubernetes*.

Requirements

Before starting this tutorial, please make sure that you have installed **Docker**. **Kubectl** will be installed with Docker. (If not, please install it from here).

The Kubectl commands used throughout this tutorial can be found in the **Kubectl Cheat Sheet**.

Through this tutorial, we will use **Visual Studio Code**, but this is not mandatory.

Creating a Production-Ready Microservices Architecture

Containerize the app

The first step is to **create the Docker image of our NodeJS backend**. After creating the image, we will **push it in to the container registry**, where it will be accessible and can be pulled by the Kubernetes service (in this case, **Azure Kubernetes Service** or **AKS**).

```
The Docker file for NodeJS:
FROM node:13.10.1
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
# Bundle app source
COPY . .
EXPOSE 3000
CMD [ “node”, “index.js” ]
```

In the *first line*, we need to define from what image we want to build our backend service. In this case, we will use the official node image with version 13.10.1 from **Docker Hub**.

In *line 3* we create a directory to hold the application code inside the image. This will be the working directory for your application.

This image comes with **Node.js and NPM already installed** so the next thing we need to do is to install your app dependencies using the *npm* command.

Note that to install the required dependencies, we **don’t have to copy the whole directory, only the package.json**, which allows us to take advantage of **cached Docker layers** (more info about efficient Dockerfiles **here**).

In *line 9* we copy our source code into the working directory and on line 11 we expose it on port 3000 (you can choose another port if you want, but make sure to change in the Kubernetes Service script, too.)

Finally, on *line 13* we define the command to run the application (inside the Docker container). Note that there should **only be one CMD instruction** in each Dockerfile. If you include more than one, **only the last will take effect**.

Now that we have defined the Dockerfile, we will build an image from it using the following Docker command (using the Terminal of the Visual Studio Code or for example using the CMD on Windows):

```
docker build -t node-user-service:dev .
```

Note the little *dot* from the end of the Docker command, it means that we are building our image from the current directory, so please make sure that you are in the same folder, where the Dockerfile is located (in this case the root folder of the repository).

To run the image locally, we can use the following command:

```
docker run -p 3000:3000 node-user-service:dev
```

To push this image to our Azure Container Registry, we have to tag it using the following format *<container_registry_log-in_service>/<image_name>:<tag>*, so in our case:

```
docker build -t node-user-service:dev .
```

The *last step* is to push it to our container registry using the following Docker command:

```
docker push stupidsimplekubernetescontainerregistry.
azurecr.io/node-user-service:dev
```

Create Pods using Deployment scripts

NodeJs backend

The next step is to define the **Kubernetes Deployment script**, which automatically manages the Pods for us.

```
piVersion: apps/v1
kind: Deployment
metadata:
  name: node-user-service-deployment
spec:
  selector:
    matchLabels:
      app: node-user-service-pod
  replicas: 3
  template:
    metadata:
      labels:
        app: node-user-service-pod
    spec:
      containers:
        - name: node-user-service-container
          image: stupidsimplekubernetescontainerregistry.
azurecr.io/node-user-service:dev
          resources:
            limits:
              memory: “256Mi”
              cpu: “500m”
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
```

The **Kubernetes API** lets you query and manipulates the state of objects in the Kubernetes Cluster (for example, Pods, Namespaces, ConfigMaps, etc.). The current stable version of this API is 1, as we specified in the *first line*.

In each Kubernetes .yml script we have to define the Kubernetes resource type (Pods, Deployments, Services, etc.) using the kind keyword. In this case, in *line 2* we defined that we would like to use the *Deployment* resource.

Kubernetes lets you add some *metadata* to your resources. This way it’s easier to identify, filter and in general to refer to your resources.

From *line 5* we define the **specifications** of this resource. In *line 8* we specified that this Deployment should be applied only to the resources with the label *app:node-user-service-pod* and in *line 9* we said that we want to create *3 replicas* of the **same pod**.

The template (starting from line 10) defines the Pods. Here we add the label *app:node-user-service-pod* to each Pod. This way they will be identified by the Deployment. In *lines 16 and 17* we define what kind of Docker Container should be run inside the pod. As you can see in *line 17*, we will use the Docker Image from our *Azure Container Registry* which was built and pushed in the previous section.

We can also define the **resource limits** for the Pods, avoiding **Pod starvation** (when a Pod uses all the resources and other Pods don't get a chance to use them). Furthermore, when you specify the resource *request* for Containers in a Pod, the scheduler uses this information to decide **which node to place the Pod on**. When you specify a resource *limit* for a Container, the **kubelet** enforces those limits so that the running container **is not allowed to use more of that resource than the limit you set**. The kubelet also **reserves at least the request amount of that system resource** specifically for that container to use. Be aware that if you don't have enough hardware resources (like CPU or memory), the pod won't be scheduled -- ever.

The last step is to define the port used for communication. In this case, we used port 3000. **This port number should be the same as the port number exposed in the Dockerfile**.

MongoDB

The Deployment script for the MongoDB database is quite similar. The only difference is that we have to specify the **volume mounts** (the folder on the node where the data will be saved).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-db-deployment
spec:
  selector:
    matchLabels:
      app: user-db-app
  replicas: 1
  template:
    metadata:
      labels:
        app: user-db-app
    spec:
      containers:
        - name: mongo
          image: mongo:3.6.4
          command:
            - mongod
            - “--bind_ip_all”
            - “--directoryperdb”
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: data
              mountPath: /data/db
          resources:
            limits:
              memory: “256Mi”
              cpu: “500m”
      volumes:
        - name: data
          persistentVolumeClaim:
           claimName: static-persistence-volume-claim-mongo
```

In this case, we used the official MongoDB image directly from the DockerHub (*line 17*). The volume mounts are defined in *line 24*. The last *four lines* will be explained in the **next chapter** when we will talk about **Kubernetes Persistent Volumes**.

Create the Services for Network Access

Now that we have the Pods up and running, we should define the communication between the containers and with the outside world. For this, we need to define a **Service**. The relation between a Service and a Deployment is 1-to-1, so for each Deployment, we should have a Service. The Deployment manages the lifecycle of the Pods and it is also responsible for monitoring them, while the **Service is responsible for enabling network access to a set of Pods** (as we saw in Chapter One).

```
apiVersion: v1
kind: Service
metadata:
  name: node-user-service
spec:
  type: ClusterIP
  selector:
    app: node-user-service-pod
  ports:
    - port: 3000
      targetPort: 3000
```

The **important part of this .yml script is the selecto**r, which defines how to identify the Pods (created by the Deployment) to which we want to refer from this Service. As you can see in *line 8*, the selector is *app:node-user-service-pod*, because the Pods from the previously defined Deployment are labeled like this. Another important thing is to define the **mapping between the container port and the Service port**. In this case, the incoming request will use the 3000 port, defined on *line 10* and they will be routed to the port defined in *line 11*.

The Kubernetes Service script for the MongoDB pods is very similar. We just have to update the selector and the ports.

```
apiVersion: v1
kind: Service
metadata:
  name: user-db-service
spec:
  clusterIP: None
  selector:
    app: user-db-app
  ports:
    - port: 27017
      targetPort: 27017
```

Configure the External Traffic

To communicate with the outside world, we need to define an **Ingress Controller** and **specify the routing rules using an Ingress Kubernetes Resource**.

To configure an **NGINX Ingress Controller** we will use the script that can be found here.

This is a generic script that can be applied without modifications (explaining the NGINX Ingress Controller is out of scope for this book).

The next step is to define the **Load Balancer**, which will be used to route external traffic using a public IP address (the cloud provider provides the load balancer).

```
kind: Service
apiVersion: v1
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  externalTrafficPolicy: Local
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: https
      port: 443
      targetPort: https
```

Now that we have the Ingress Controller and the Load Balancer up and running, we can define the **Ingress Kubernetes Resource for specifying the routing rules**.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: node-user-service-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
    - host: stupid-simple-kubernetes.eastus2.cloudapp.
azure.com
      http:
        paths:
          - backend:
              serviceName: node-user-service
              servicePort: 3000
            path: /user-api(/|$)(.*)
          # - backend:
          #     serviceName: nestjs-i-consultant-service
          #     servicePort: 3001
          #   path: /i-consultant-api(/|$)(.*)
```

In *line 6* we define the Ingress Controller type (it's a Kubernetes predefined value; Kubernetes as a project currently supports and maintains GCE and nginx controllers).

In line 7 we define the rewrite target rules (more information here) and in line 10 we define the hostname.

For each service that should be accessible from the outside world, we should add an entry in the paths list (starting from line 13). In this example, we added only one entry for the NodeJS user service backend, which will be accessible using port 3000. The /user-api uniquely identifies our service, so any request that starts with stupid-simple-kubernetes.eastus2.cloudapp.azure.com/user-api will be routed to this NodeJS backend. If you want to add other services, then you have to update this script (as an example see the commented out code).

Apply the .yml scripts

To apply these scripts, we will use the **kubectl**. The kubectl command to apply files is the following:

```
kubectl apply -f <file_name>
```

So in our case, if you are in the root folder of the StupidSimpleKubernetes repository, you will write the following commands:

```
kubectl apply -f .\manifest\kubernetes\deployment.yml
kubectl apply -f .\manifest\kubernetes\service.yml
kubectl apply -f .\manifest\kubernetes\ingress.yml
kubectl apply -f .\manifest\ingress-controller\nginx-in-
gress-controller-deployment.yml
kubectl apply -f .\manifest\ingress-controller\ng-
nix-load-balancer-setup.yml
```

After applying these scripts, we will have everything in place, so we can call our backend from the outside world (for example by using Postman).

Conclusion

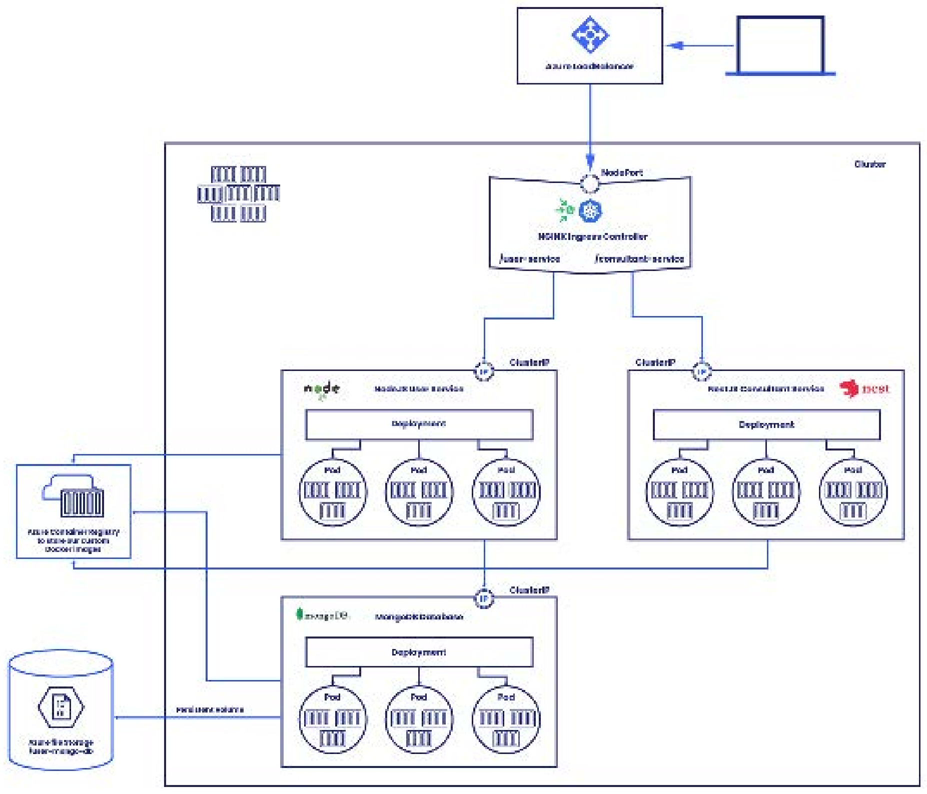
In this tutorial, we learned how to create different kinds of resources in Kubernetes, like **Pods, Deployments, Services, Ingresses** and **Ingress Controller**. We created a **NodeJS backend with a MongoDB database** and we **containerized and deployed** the NodeJS and MongoDB containers using **replication of 3 pods**.

In the next chapter, we will approach the problem of saving data persistently and we will learn about **Persistent Volumes in Kubernetes**.

Chapter 3

Persistent Volumes Explained

Welcome back to our series, where we introduce you to the basic concepts of Kubernetes. In the first chapter, we provided a brief introduction to **Persistent Volumes**. Here, we'll dig into this topic: we will learn how to set up data persistency and will write Kubernetes scripts to connect our Pods to a Persistent Volume. In this example, we will use **Azure File Storage** to store the data from our **MongoDB** database, but you can use any kind of volume to achieve to same results (such as Azure Disk, GCE Persistent Disk, AWS Elastic Block Store, etc.)



NOTE: the scripts provided are **platform agnostic**, so you can follow the tutorial using other types of cloud providers or using a local cluster with K3s. I suggest using **K3s** because it is very lightweight, packed in a single binary with a size less than 40MB. It is also a highly available, certified Kubernetes distribution designed for production workloads in resource-constrained environments. For more information, take a look at its well-written and easy-to-follow documentation.

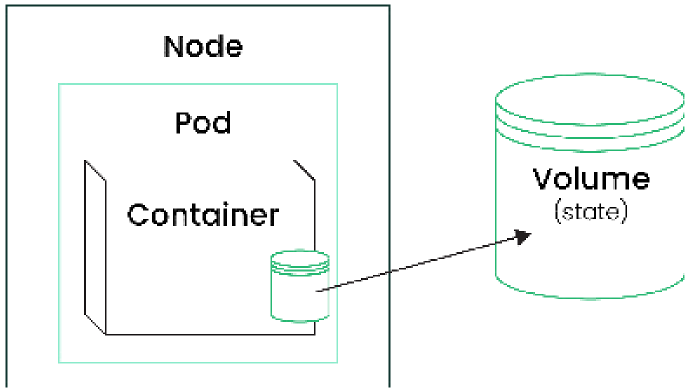
Requirements

Before starting this tutorial, please make sure that you have installed Docker. Kubectl will install with Docker (if not, please install it from here).

The Kubectl commands used throughout this tutorial can be found in the Kubectl Cheat Sheet.

Through this tutorial, we will use Visual Studio Code, but this is not mandatory.

What Problem Does Kubernetes Volume Solve?



Remember that we have a **Node** (an actual hardware device or a virtual machine) and **inside the Nodes, we have a Pod** (or multiple Pods) and **inside the Pod, we have the Container. Pods are ephemeral**, so they can come and go very often (they can be deleted, rescheduled, etc.). In this case, if you have data that you must keep even if the Pod goes down you have to **move it outside the Pod**. This way it can exist independently of any Pod. This external place is called **Volume** and it is an **abstraction of a storage system**. Using the Volume, you can persist state across multiple Pods.

When to Use Persistent Volumes

When containers became popular, they were designed to support stateless workloads with persistent data stored elsewhere. Since then, a lot of effort has been made to support stateful applications in the container ecosystem.

Every project needs some kind of data persistency, so usually, you need a database to store the data. But **in a clean design, you don't want to depend on concrete implementations**; you want to write an application as reusable and platform independent as possible.

There has always been a need to hide the details of storage implementation from the applications. But now, in the era of cloud-native applications, cloud providers create environments where applications or users who want to access the data need to integrate with a specific storage system. For example, many applications are directly using specific storage systems like Amazon S3, Azure File or Blog storage, etc. which create an unhealthy dependency. Kubernetes is trying to change this by creating an **abstraction called Persistent Volume**, which allows cloud-native applications to connect to a wide variety of cloud storage systems without having to create an explicit dependency with those systems. This can make the consumption of cloud storage much more seamless and eliminate integration costs. It can also **make it much easier to migrate between clouds and adopt multi-cloud strategies**.

Even if sometimes, because of material constraints like money, time or manpower (which are closely related) you have to make some compromises and directly couple your app with a specific platform or provider, you should try to avoid as many direct dependencies as possible. One way of decoupling your application from the actual database implementation (there are other solutions, but those solutions require more effort) is by **using containers** (and Persistent Volumes to prevent data loss). This way, **your app will rely on abstraction instead of a specific implementation**.

Now the real question is, *should we always use a containerized database with Persistent Volume, or what are the storage system types which should NOT be used in containers?*

There is no golden rule of when you should and shouldn't use Persistent Volumes, but as a starting point, you should **have in mind scalability and the handling of the loss of node in the cluster**.

Based on scalability, we can have two types of storage systems:

1. **Vertically scalable**—includes traditional RDMS solutions such as MySQL, PostgreSQL and SQL Server
2. **Horizontally scalable**—includes "NoSQL" solutions such as ElasticSearch or Hadoop based solution

Vertically scalable solutions like MySQL, Postgres, Microsoft SQL, etc. **should NOT go in containers**. These database platforms require high I/O, shared disks, block storage, etc., and were not designed to handle the loss of a node in a cluster gracefully, which often happens in a container-based ecosystem.

For **horizontally scalable** applications (Elastic, Cassandra, Kafka, etc.), you **should use containers**, because they can withstand the loss of a node in the database cluster and the database application can independently re-balance.

Usually, **you can and should containerize distributed databases that use redundant storage techniques and withstand the loss of a node** in the database cluster (*ElasticSearch* is a really good example).

Types of Kubernetes Volumes

We can categorize the Kubernetes Volumes based on their **lifecycle** and **the way they are provisioned**.

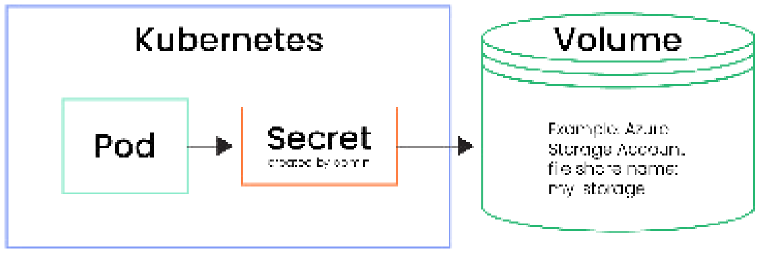
Considering the **lifecycle** of the volumes, we can have:

1. **Ephemeral Volumes**, which are tightly coupled with the lifetime of the Node (for example emptyDir, or hostPath) and they are deleted if the Node goes down.
2. **Persistent Volumes**, which are meant for long-term storage and are independent of the Pods or Nodes lifecycle. These can be cloud volumes (like gcePersistentDisk, awsElasticBlockStore, azureFile or azureDisk), NFS (Network File Systems) or Persistent Volume Claims (a series of abstraction to connect to the underlying cloud provided storage volumes).

Based on the **way the volumes are provisioned**, we can have:

1. **Direct access**
2. **Static provisioning**
3. **Dynamic provisioning**

Direct Access Persistent Volumes



In this case, the pod will be **directly coupled with the volume**, so it will know the storage system (for example, the Pod will be coupled with the Azure Storage Account). This solution is **not cloud-agnostic** and it **depends on a concrete implementation and not an abstraction**. So if possible, please avoid this solution. The only advantage is that it is easy and fast. Create the Secret in the Pod and specify the Secret and the exact storage type that should be used.

The script for creating a **Secret** is as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: static-persistence-secret
type: Opaque
data:
  azurestorageaccountname: “base64StorageAccountName”
  azurestorageaccountkey: “base64StorageAccountKey”
```

As in any Kubernetes script, on *line 2* we specify the type of the resource -- in this case, Secret. On *line 4*, we give it a name (we called it static because it is manually created by the Admin and not automatically generated). The *Opaque type*, from Kubernetes' point of view, means that the content (data) of this Secret is unstructured (it can contain arbitrary key-value pairs). To learn more about Kubernetes Secrets, see the Secrets design document and Configure Kubernetes Secrets.

In the data section, we have to specify the account name (in Azure, it is the name of the Storage Account) and the access key (in Azure, select the Storage Account under Settings, Access key). Don't forget that both should be **encoded using Base64**.

The next step is to modify our **Deployment** script to use the Volume (in this case the volume is the Azure File Storage).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-db-deployment
spec:
  selector:
    matchLabels:
      app: user-db-app
  replicas: 1
  template:
    metadata:
      labels:
        app: user-db-app
    spec:
      containers:
        - name: mongo
          image: mongo:3.6.4
          command:
            - mongod
            - “--bind_ip_all”
            - “--directoryperdb”
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: data
              mountPath: /data/db
          resources:
            limits:
              memory: “256Mi”
              cpu: “500m”
      volumes:
        - name: data
          azureFile:
            secretName: static-persistence-secret
            shareName: user-mongo-db
            readOnly: false
```

As you can see, the only difference is that from *line 32* we specify the used volume, give it a name and specify the exact details of the underlying storage system. The secretName must be the name of the previously created Secret.

Kubernetes Storage Class

To understand the **Static** or **Dynamic provisioning**, first we have to understand the **Kubernetes Storage Class**.

With **StorageClass**, administrators can offer **Profiles** or “**classes**” regarding the available storage. Different classes might map to quality-of-service levels, or backup policies or arbitrary policies determined by the cluster administrators.

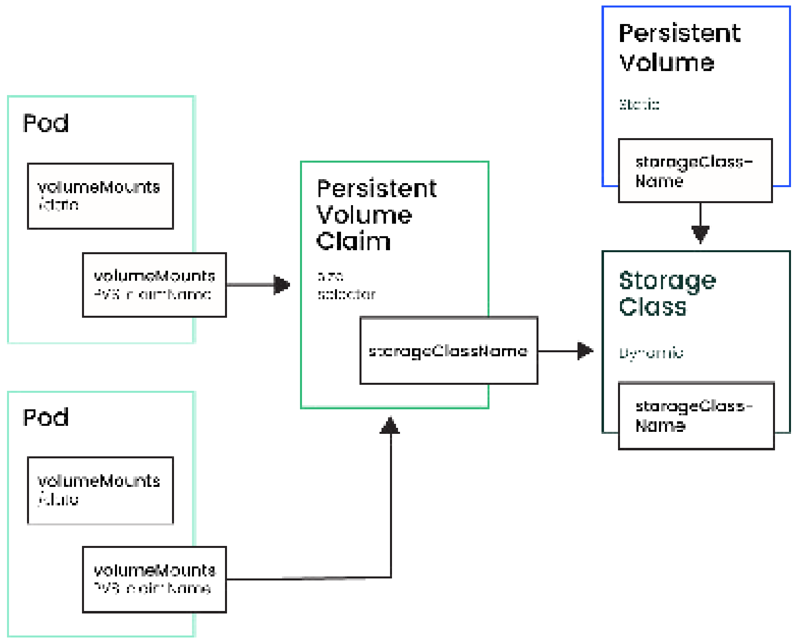
For example, you could have a profile to store data on an HDD named slow-storage or a profile to store data on an SSD named fast-storage. The kind of storage is determined by the Provisioner. For Azure, there are two kinds of provisioners: **AzureFile** and **AzureDisk** (the difference is that AzureFile can be used with ReadWriteMany access mode, while AzureDisk supports only ReadWriteOnce access, which can be a disadvantage when you want to use multiple pods simultaneously). You can learn more about the different types of StorageClasses here.

The script for our **StorageClass**:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefilestorage
provisioner: kubernetes.io/azure-file
parameters:
  storageAccount: storageaccountname
reclaimPolicy: Retain
allowVolumeExpansion: true
```

Kubernetes predefines the value for the provisioner property (see Kubernetes Storage Classes). The ***Retain*** reclaim policy means that after we delete the *PVC* and *PV*, **the actual storage medium is NOT purged**. We can set it to ***Delete*** and with this setting, **as soon as a PVC is deleted**, **it also triggers the removal of the corresponding PV along with the actual storage medium** (here the actual storage is the Azure File Storage).

Persistent Volume and Persistent Volume Claim



Kubernetes has a matching primitive for each of the traditional storage operational activities (provisioning/configuring/attaching). **Persistent Volume is Provisioning**, **Storage Class is Configuring** and **Persistent Volume Claim is Attaching**.

From the original documentation:

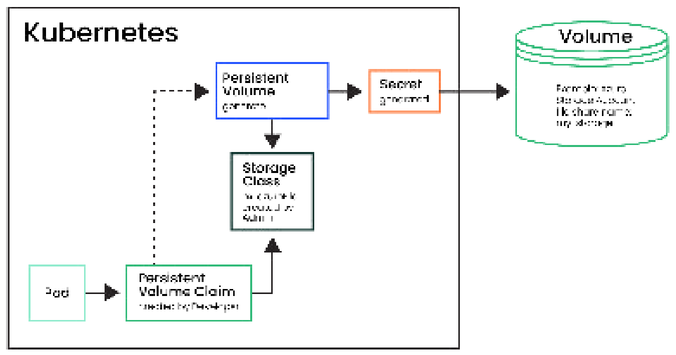
A *PersistentVolume* (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

A *PersistentVolumeClaim* (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and memory). Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).

This means that the **Admin** will create the **Persistent Volume** to specify the type of storage that can be used by the Pods, the size of the storage, and the access mode. The **Developer** will create a **Persistent Volume Claim** asking for a piece of volume, access permission and the type of storage. This way there is a clear separation between "Dev" and "Ops." **Devs** are responsible for **asking for the necessary volume (PVC)** and **Ops** are **responsible for preparing and provisioning the requested volume (PV)**.

The **difference between Static and Dynamic provisioning** is that if **there isn't a PersistentVolume** and a Secret created manually by the Admin, Kubernetes will try to automatically create these resources.

Dynamic Provisioning



In this case, **there is NO PersistentVolume and Secret created manually**, so Kubernetes will try to **generate** them. The StorageClass is mandatory and we will use the one created earlier.

The script for the **PersistentVolumeClaim** can be found below:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: persistent-volume-claim-mongo
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: azurefilestorage
```

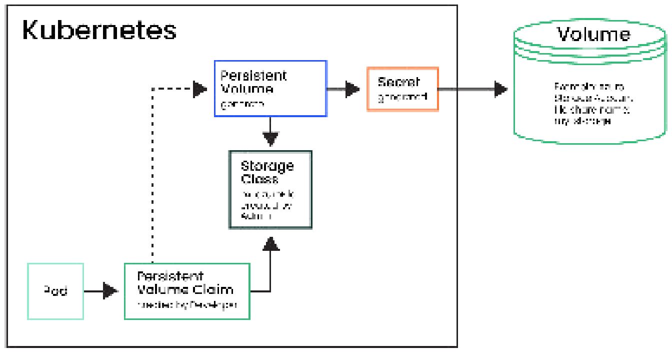
And our **updated Deployment** script:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-db-deployment
spec:
  selector:
    matchLabels:
      app: user-db-app
  replicas: 1
  template:
    metadata:
      labels:
        app: user-db-app
    spec:
      containers:
        - name: mongo
          image: mongo:3.6.4
          command:
            - mongod
            - "--bind_ip_all"
            - "--directoryperdb"
          ports:
            - containerPort: 27017
          volumeMounts:
            - name: data
              mountPath: /data/db
          resources:
            limits:
              memory: "256Mi"
              cpu: "500m"
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: persistent-volume-claim-mongo
```

As you can see, in *line 34* we referenced the previously created PVC by name. In this case, we didn't create a PersistenVolume or a Secret for it, so it will be created automatically.

The most important **advantage** of this approach is that you **don't have to create the PV and the Secret manually** and the Deployment is **cloud agnostic**. The underlying detail of the storage is not present in the Pod's specs. But there are also some **disadvantages**: you **cannot configure the Storage Account or the File Share** because they are auto-generated and you cannot reuse the PV or the Secret — they will be **regenerated for each new Claim**.

Dynamic Provisioning



The only difference between Static and Dynamic provisioning is that **we manually create the PersistentVolume and the Secret** in Static Provisioning. This way we have full control over the resource that will be created in our cluster.

The **PersistentVolume** script is below:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: static-persistent-volume-mongo
  labels:
    storage: azurefile
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  storageClassName: azurefilestorage
  azureFile:
    secretName: static-persistence-secret
    shareName: user-mongo-db
    readOnly: false
```

It is important that in *line 12* we reference the StorageClass by name. Also, in *line 14* we reference the Secret, which is used to access the underlying storage system.

I recommend this solution, even if it requires more work, because it is **cloud-agnostic**. It also lets you apply **separation of concerns regarding roles** (Cluster Administrator vs. Developers) and gives you control of **naming and resource creation**.

Conclusion

In this tutorial, we learned how to **persist data and state using Volumes**. We presented three different ways of setting up your system, **Direct Access**, **Dynamic Provisioning and Static Provisioning** and discussed the **advantages** and **disadvantages** of each.

In *chapter 5*, we will talk about **CI/CD pipelines** to automate the deployment of Microservices.

Chapter 4

Device Plugins Explained

As discussed in Chapter 1, **Nodes** are worker machines in Kubernetes. They can be any device that has CPU and RAM, and can be a physical machine or a virtual machine. In other words, a node is an abstraction over a single device.

Modern server, storage or networking devices (available as either physical or virtual resources) may have access to a rich set of hardware resources. These can range from hardware accelerators (such as GPUs or SmartNICs) to non-volatile memory resources, to name a few. For example, AI and ML workloads often benefit from GPU accelerators being available to the workload. Kubernetes enables the creation of clusters with a variety of different types of work nodes, including instruction set architectures (ISAs) such as x86_64 and others.

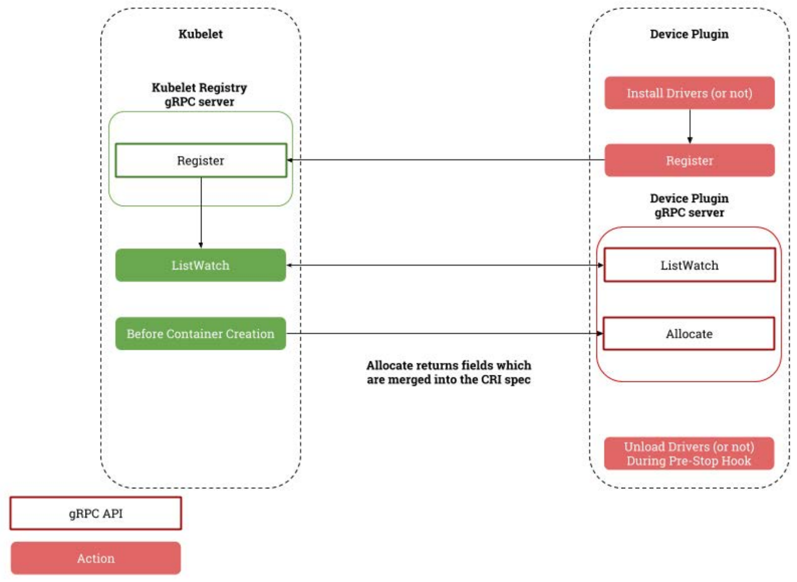
Hardware architecture (processors, graphics, storage, networks, etc.) remains as relevant as ever in cloud-native environments. Customers' performance requirements still need to be addressed and in many instances (for example AI and ML), hardware accelerators will play a key role.

Now, how does a Kubernetes cluster become 'aware' of the different hardware capabilities at its disposal (beyond CPU and memory)? And how do we ensure that these resources are made available to those workloads that can benefit from it? The answer is: *Kubernetes Device Plugins*.

Kubernetes Device Plugins

Kubernetes Device Plugins provide vendors with a mechanism to announce their resources (for example GPU) to Kubelet and monitor them without additional Kubernetes core code changes. For users, it provides consistency when it comes to the consumption of hardware resources.

"As a refresher, Kubelet is an agent that runs on each node of the cluster. It makes sure that containers are running in a pod".


Kubelet
Kubelet Registry gRPC server
Register
ListWatch
Before Container Creation
Device Plugin
Install Drivers (or not)
Register
Device Plugin gRPC server
ListWatch
Allocate
Allocate returns fields which are merged into the CRI spec
Unload Drivers (or not) During Pre-Stop Hook
gRPC API
Action

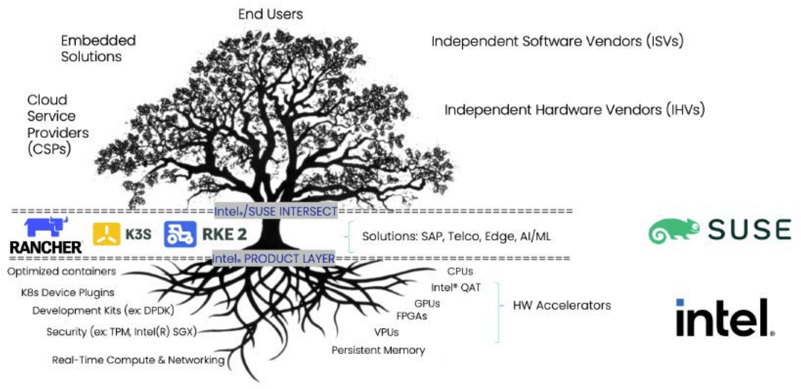
Device Plugins Overview Diagram

Once the device plugin is registered with the Kubernetes cluster, it sends Kubelet the list of devices that it manages. Kubelet, in turn, advertises the availability of these resources. Users then request devices as part of a pod specification.

Device Plugins Example – Intel®

Intel® is an industry leader in the design, manufacture and sale of Silicon-based hardware and software solutions. While most of us immediately associate Intel with processors (CPUs), the company provides a large ecosystem of hardware technologies that are key to today's digital life. These include (but are not limited to) processors as Intel® Xeon™ scalable processors, memory, GPUs, crypto accelerators, secure enclaves and FPGAs to name a few.

To enable optimized consumption of Intel's hardware technologies in cloud-native environments, Intel provides several device plugins. These are usually made available in the form of Kubernetes Operators (collecting several device plugins together) or drivers (in the case of storage. Like the roots in a tree, Intel's work is often not 'visible', but the 'tree' (in this case, the cloud-native ecosystem) cannot function without it.


End Users
Embedded Solutions
Independent Software Vendors (ISVs)
Cloud Service Providers (CSPs)
Independent Hardware Vendors (IHVs)
Intel®/SUSE INTERSECT
RANCHER
K3S
RKE 2
Solutions: SAP, Telco, Edge, AI/ML
SUSE
Intel® PRODUCT LAYER
Optimized containers
K8s Device Plugins
Development Kits (ex: DPDK)
Security (ex: TPM, Intel(R) SGX)
Real-Time Compute & Networking
CPUs
Intel® QAT
GPUs
FPGAs
VPUs
Persistent Memory
HW Accelerators
intel.

Positioning of Intel and SUSE cloud-native components enabling the rest of the ecosystem.

For illustration purposes, we will introduce Intel's Device Plugin Operator and the Intel Operator for Permanent Memory (PMEM) via CSI driver.

Intel® Device Plugins Operator

Intel's Device Plugins Operator provides a collection of device plugins advertising Intel-specific hardware resources to the kubelet. The operator provides unified implementation for:

- Data streaming accelerator (DSA) device plugin.
- Dynamic Load Balancer (DLB) device plugin.

- FPGA device plugin for Intel® Arria® and Stratix® 10 devices.
- GPU device plugin to access discrete (Intel® Iris® Xe MAX) and integrated GPU hardware device files.
- Intel® Analytics accelerator (IAA) device plugin.
- Intel® Quick Assist Technology (Intel QAT) device plugin (cryptography acceleration and compression capabilities).
- Intel® Software Guard Extensions (Intel SGX) device plugin.

Instructions and tests on how to install the Intel Device Plugins Operator can be found at: https://github.com/intel/intel-device-plugins-for-kubernetes/blob/main/cmd/operator/README.md

Intel® Operator for PMEM-CSI driver

Intel PMEM-CSI is a container storage interface (CSI) driver for container orchestrators like Kubernetes. It enables local persistent memory (PMEM – for example Intel® Optane™) available as a filesystem volume to container-based applications.

PMEM is a project under active development. For up-to-date information on feature status and how to install (including a demo), visit: https://intel.github.io/pmem-csi/latest/README.html

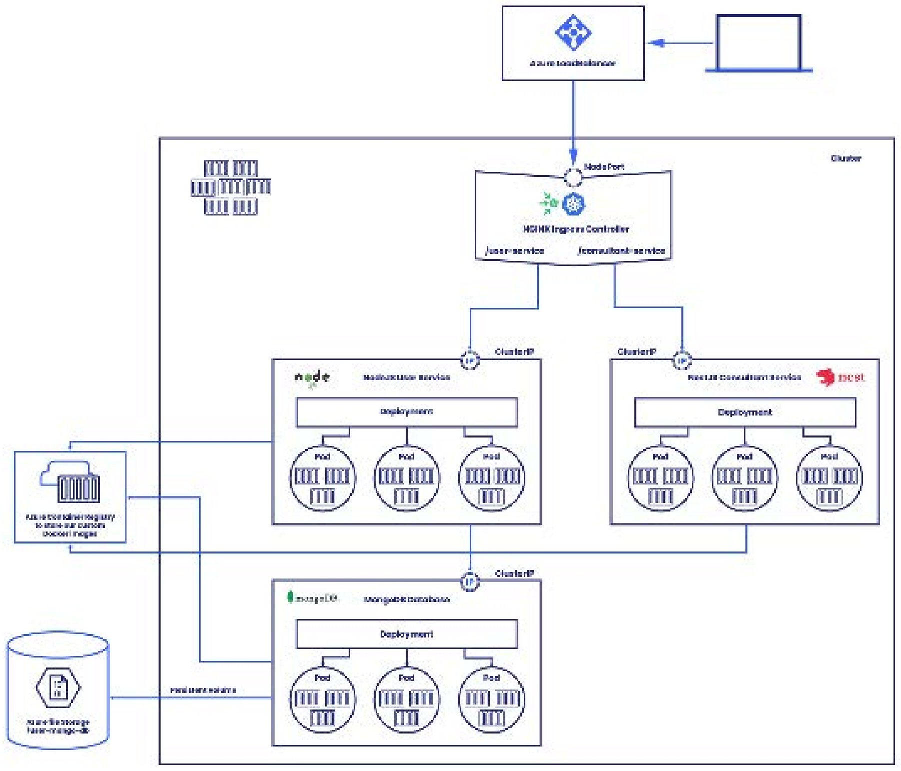
Chapter 5

Create an Azure Infrastructure for Microservices

In the **first chapter**, we learned about the basic concepts used in Kubernetes and its hardware structure. We talked about the different software components, including Pods, Deployments, StatefulSets and Services and how to communicate between services and with the outside world.

In this chapter, we're getting practical. We will create all the necessary configuration files to deploy multiple microservices in different languages using MongoDB as data storage. We will also learn about Azure Kubernetes Service (AKS) and will present the infrastructure used to deploy our services.

The code used in this chapter can be found in my StupidSimpleKubernetes-AKS git repository. If you like it, please leave a star!



NOTE: the scripts provided are **platform agnostic**, so you can follow the tutorial using other types of cloud providers or a local cluster with K3s. I suggest using **K3s** because it is very lightweight, packed in a single binary with a size less than 40MB. Furthermore, it is a highly available, certified Kubernetes distribution designed for production workloads in resource-constrained environments. For more information, you can take a look over its well-written and easy-to-follow **documentation**.

Requirements

Before starting this tutorial, please make sure that you have installed Docker and Azure CLI. Kubectl will be installed with Docker (if not, please install it from here).

You will also need an Azure Account. Azure offers a 30-day free trial that gives you $200 in credit, which will be more than enough for our tutorial.

Through this tutorial, we will use Visual Studio Code, but this is not mandatory.

Creating a Production Ready Azure Infrastructure for Microservices

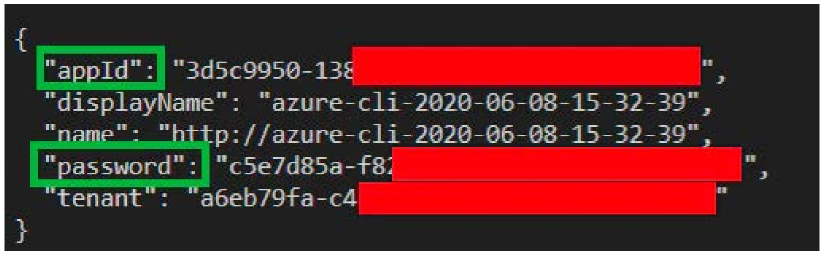
To have a fast setup, I've created an ARM Template, which will automatically spin up all the Azure resources needed for this tutorial. You can read more about ARM Templates here.

We will run all the scripts in the VS Code Terminal.

The first step is to log in to your Azure account from the VS Code Terminal. For this run *az login*. This will open a new tab in your default browser, where you can enter your credentials.

For the Azure Kubernetes Service, we need to set up a Service Principal. For this, I've created a PowerShell script called create-service-principal.ps1. Just run this script in the VS Code Terminal or PowerShell.

After running the code, it will return a JSON response with the following structure:



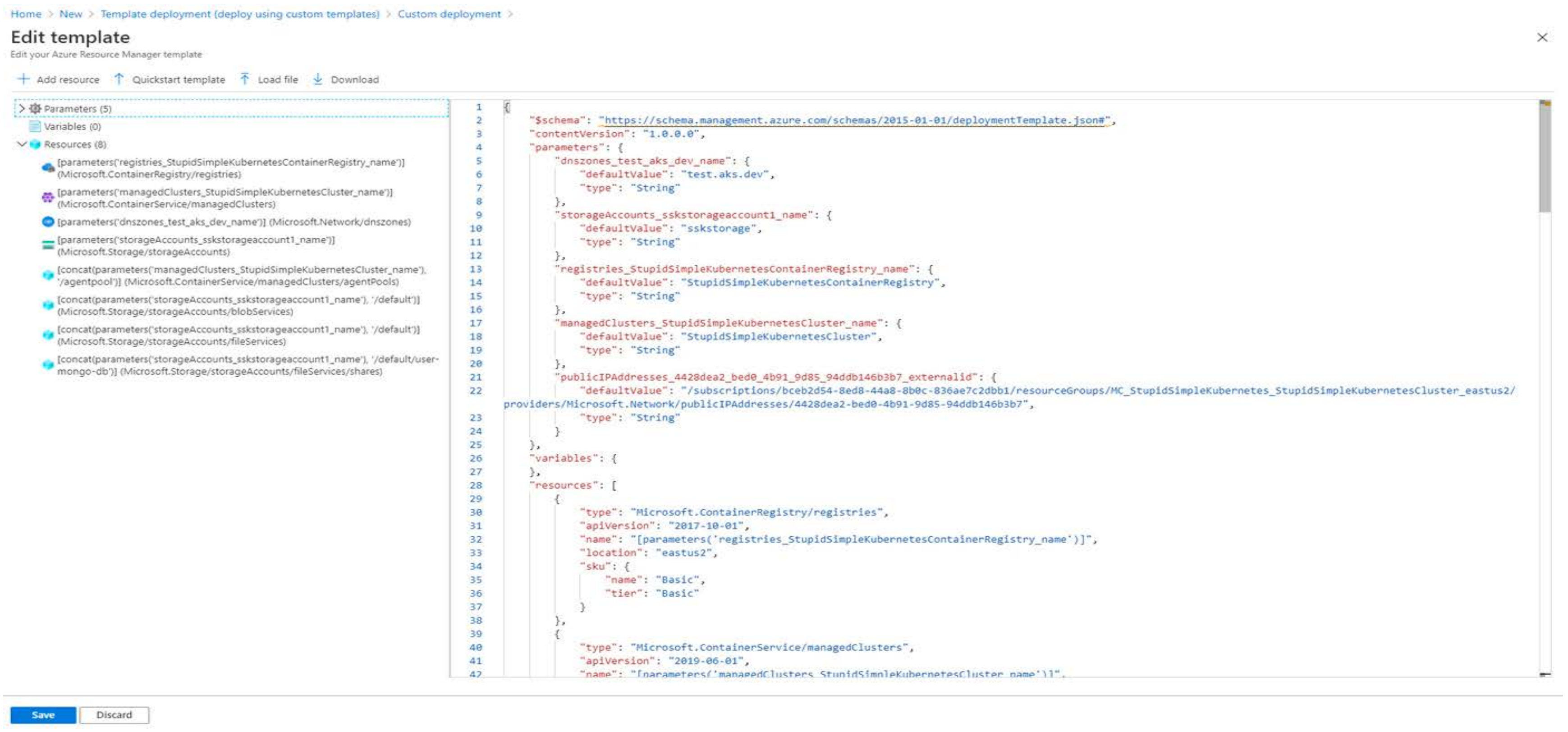
```
{
  "appId": "3d5c9950-138[redacted]",
  "displayName": "azure-cli-2020-06-08-15-32-39",
  "name": "http://azure-cli-2020-06-08-15-32-39",
  "password": "c5e7d85a-f8[redacted]",
  "tenant": "a6eb79fa-c4[redacted]"
}
```

Based on this information, you will have to update the ARM Template to use your Service Principal. For this, please copy the appId from the returned JSON to the clientId in the ARM Template. Also, copy the password and paste it into the ARM Template's secret field.

In the next step, you should create a new Resource Group called "StupidSimpleKubernetes" in your Azure Portal and import the ARM template to it.

To import the ARM template, in the Azure Portal, click on the Create a resource button, search for Template Deployment and select Build your own template in the editor. Copy and paste the template code from our Git repository to the Azure template editor. Now you should see something like in the following picture:

Hit the save button, select the StupidSimpleKubernetes resource group, and hit Purchase. This will take a while and it will create all the necessary Azure resources for a production-ready microservices infrastructure.


Home > New > Template deployment (deploy using custom templates) > Custom deployment >
Edit template
Edit your Azure Resource Manager template
Add resource
Quickstart template
Load file
Download
Parameters (5)
Variables (0)
Resources (8)
Save
Discard

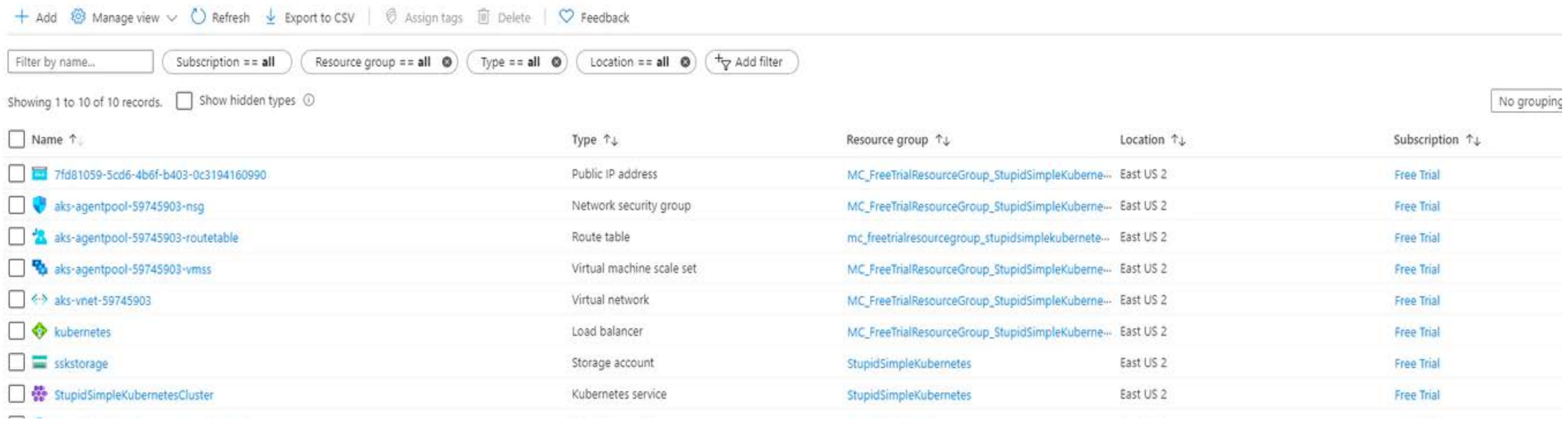
[parameters('registries_StupidSimpleKubernetesContainerRegistry_name')] (Microsoft.ContainerRegistry/registries)
[parameters('managedClusters_StupidSimpleKubernetesCluster_name')] (Microsoft.ContainerService/managedClusters)
[parameters('dnszones_test_aks_dev_name')] (Microsoft.Network/dnszones)
[parameters('storageAccounts_sskstorageaccount1_name')] (Microsoft.Storage/storageAccounts)
[concat(parameters('managedClusters_StupidSimpleKubernetesCluster_name'), '/agentpool')] (Microsoft.ContainerService/managedClusters/agentPools)
[concat(parameters('storageAccounts_sskstorageaccount1_name'), '/default')] (Microsoft.Storage/storageAccounts/blobServices)
[concat(parameters('storageAccounts_sskstorageaccount1_name'), '/default')] (Microsoft.Storage/storageAccounts/fileServices)
[concat(parameters('storageAccounts_sskstorageaccount1_name'), '/default/user-mongo-db')] (Microsoft.Storage/storageAccounts/fileServices/shares)

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "dnszones_test_aks_dev_name": {
            "defaultValue": "test.aks.dev",
            "type": "String"
        },
        "storageAccounts_sskstorageaccount1_name": {
            "defaultValue": "sskstorage",
            "type": "String"
        },
        "registries_StupidSimpleKubernetesContainerRegistry_name": {
            "defaultValue": "StupidSimpleKubernetesContainerRegistry",
            "type": "String"
        },
        "managedClusters_StupidSimpleKubernetesCluster_name": {
            "defaultValue": "StupidSimpleKubernetesCluster",
            "type": "String"
        },
        "publicIPAddresses_4428dea2_bed0_4b91_9d85_94ddb146b3b7_externalid": {
            "defaultValue": "/subscriptions/bceb2d54-8ed8-44a8-8b0c-836ae7c2dbb1/resourceGroups/MC_StupidSimpleKubernetes_StupidSimpleKubernetesCluster_eastus2/
providers/Microsoft.Network/publicIPAddresses/4428dea2-bed0-4b91-9d85-94ddb146b3b7",
            "type": "String"
        }
    },
    "variables": {
    },
    "resources": [
        {
            "type": "Microsoft.ContainerRegistry/registries",
            "apiVersion": "2017-10-01",
            "name": "[parameters('registries_StupidSimpleKubernetesContainerRegistry_name')]",
            "location": "eastus2",
            "sku": {
                "name": "Basic",
                "tier": "Basic"
            }
        },
        {
            "type": "Microsoft.ContainerService/managedClusters",
            "apiVersion": "2019-06-01",
            "name": "[parameters('managedClusters_StupidSimpleKubernetesCluster_name')]",
```

You can also apply the ARM Template using the Azure CLI, by running the following command in the root folder of our git repository:

```
az deployment group create --name testtemplate --re-
source-group StupidSimpleKubernetes --template-file .\mani-
fest\arm-templates\template.json
```

After the ARM Template Deployment is done, we should have the following Azure resources:



Showing 1 to 10 of 10 records.

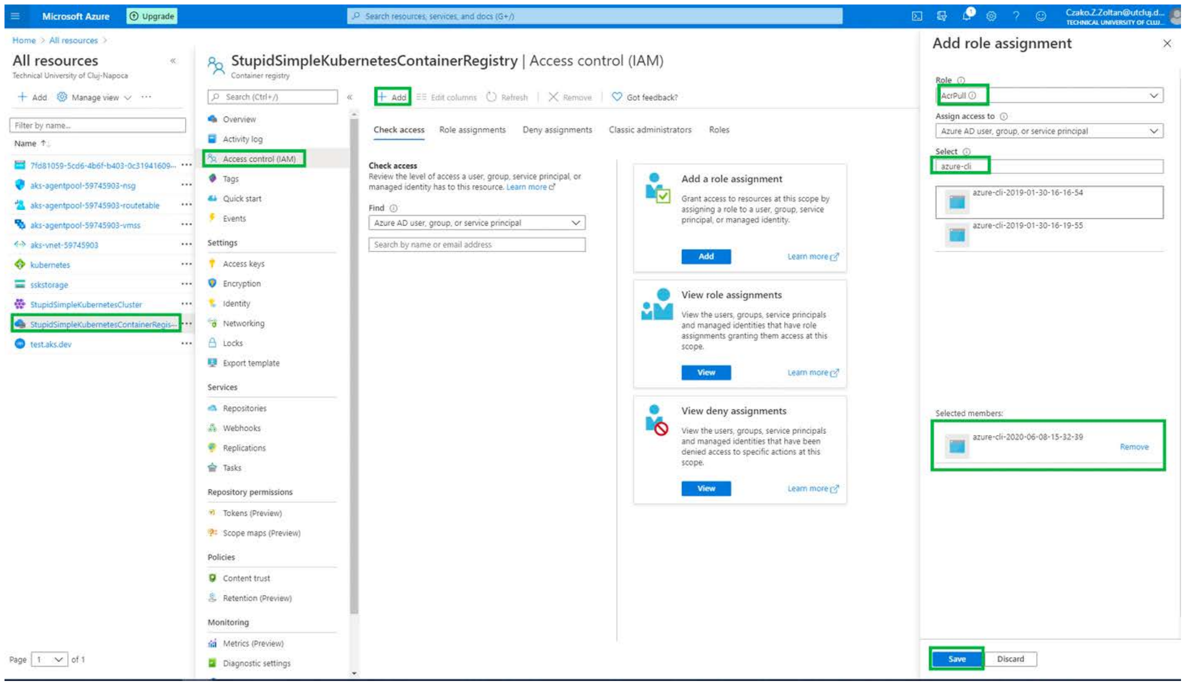
Name	Type	Resource group	Location	Subscription
7fd81059-5cd6-4b6f-b403-0c3194160990	Public IP address	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
aks-agentpool-59745903-nsg	Network security group	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
aks-agentpool-59745903-routetable	Route table	mc_freetrialresourcegroup_stupidsimplekubernete...	East US 2	Free Trial
aks-agentpool-59745903-vmss	Virtual machine scale set	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
aks-vnet-59745903	Virtual network	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
kubernetes	Load balancer	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
sskstorage	Storage account	StupidSimpleKubernetes	East US 2	Free Trial
StupidSimpleKubernetesCluster	Kubernetes service	StupidSimpleKubernetes	East US 2	Free Trial

The next step is to authorize our Kubernetes service to pull images from the Container Registry. For this, select the container registry, select the Access Control (IAM) menu option from the left menu, click on the Add button and select Role Assignment.

In the right menu, search for the correct Service Principal (use the *Z* from the returned JSON object—see the Service Principal image above).

After this step, our Kubernetes Service will be able to pull the right Docker images from the Azure Container Registry. We will store all our custom Docker images in this Azure Container Registry.

We are almost ready! In the last step, we will set up the NGNIX Ingress Controller and add a RecordSet to our DNS. This assigns a human-readable hostname to our services instead of using the IP:PORT of the Load Balancer.



Role: AcrPull

Assign access to: Azure AD user, group, or service principal

Select: azure-cli

azure-cli-2019-01-30-16-16-54

azure-cli-2019-01-30-16-19-55

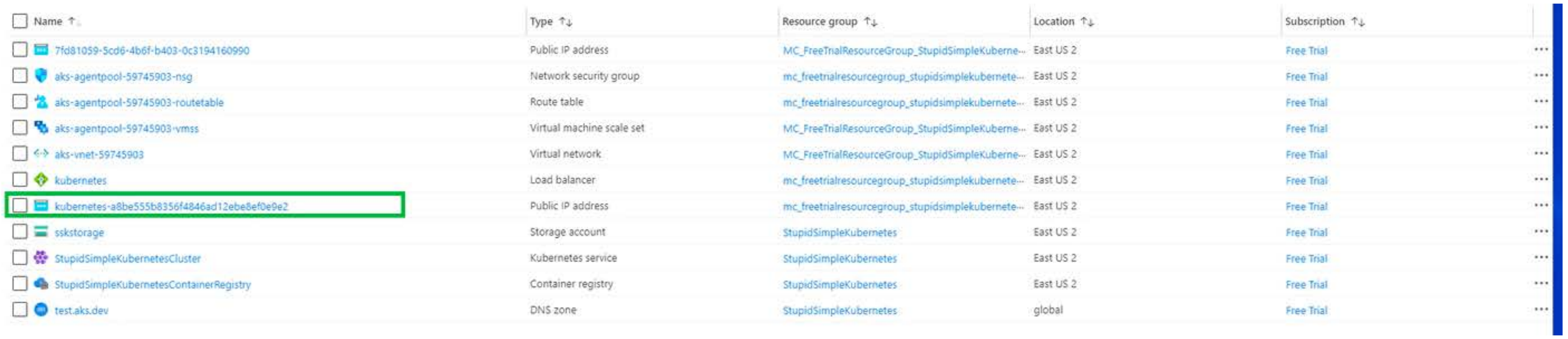
Selected members:

azure-cli-2020-06-08-15-32-39 Remove

To set up the NGINX Ingress Controller, run the following two commands in the root folder of the repository, one by one:

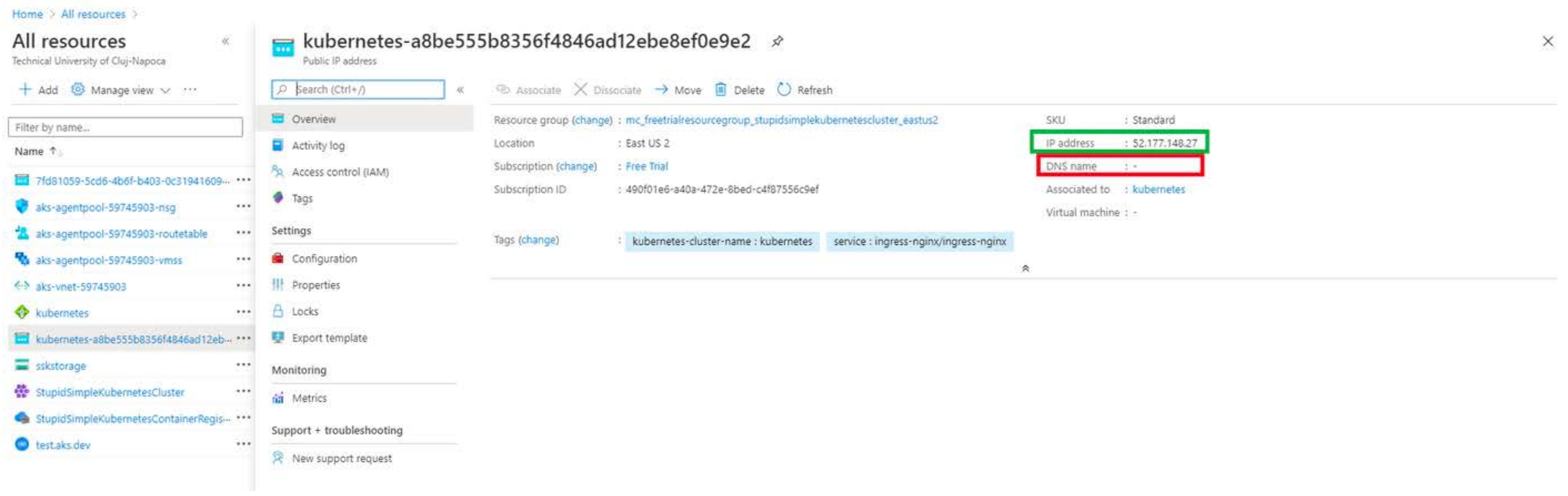
```
kubectl apply -f .\manifest\ingress-controller\nginx-in-
gress-controller-deployment.yml
kubectl apply -f .\manifest\ingress-controller\ng-
nix-load-balancer-setup.yml
```

This will create a new public IP, which you can see in the Azure Portal:



Name	Type	Resource group	Location	Subscription
7fd81059-5cd6-4b6f-b403-0c3194160990	Public IP address	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
aks-agentpool-59745903-nsg	Network security group	mc_freetrialresourcegroup_stupidsimplekubernete...	East US 2	Free Trial
aks-agentpool-59745903-routetable	Route table	mc_freetrialresourcegroup_stupidsimplekubernete...	East US 2	Free Trial
aks-agentpool-59745903-vmss	Virtual machine scale set	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
aks-vnet-59745903	Virtual network	MC_FreeTrialResourceGroup_StupidSimpleKuberne...	East US 2	Free Trial
kubernetes	Load balancer	mc_freetrialresourcegroup_stupidsimplekubernete...	East US 2	Free Trial
kubernetes-a8be555b8356f4846ad12ebe8ef0e9e2	Public IP address	mc_freetrialresourcegroup_stupidsimplekubernete...	East US 2	Free Trial
sskstorage	Storage account	StupidSimpleKubernetes	East US 2	Free Trial
StupidSimpleKubernetesCluster	Kubernetes service	StupidSimpleKubernetes	East US 2	Free Trial
StupidSimpleKubernetesContainerRegistry	Container registry	StupidSimpleKubernetes	East US 2	Free Trial
test.aks.dev	DNS zone	StupidSimpleKubernetes	global	Free Trial

If we take a look over the details of this new Public IP resource, we can see that it does NOT have a DNS name.



7fd81059-5cd6-4b6f-b403-0c31941609...
aks-agentpool-59745903-nsg
aks-agentpool-59745903-routetable
aks-agentpool-59745903-vmss
aks-vnet-59745903
kubernetes
kubernetes-a8be555b8356f4846ad12eb...
sskstorage
StupidSimpleKubernetesCluster
StupidSimpleKubernetesContainerRegis...
test.aks.dev

kubernetes-a8be555b8356f4846ad12ebe8ef0e9e2
Public IP address

Resource group (change) : mc_freetrialresourcegroup_stupidsimplekubernetescluster_eastus2
Location : East US 2
Subscription (change) : Free Trial
Subscription ID : 490f01e6-a40a-472e-8bed-c4f87556c9ef
Tags (change) : kubernetes-cluster-name : kubernetes service : ingress-nginx/ingress-nginx

SKU : Standard
IP address : 52.177.148.27
DNS name : -
Associated to : kubernetes
Virtual machine : -

To assign a human-readable DNS name to this Public IP, please run the following PowerShell script (just replace the IP address with the correct IP address from your Public IP resource):

This assigns a DNS name to the public IP of your NGINX Ingress Controller.

Now we are ready to deploy our Microservices to the Azure Kubernetes Cluster.

Conclusion

In this tutorial, we learned how to create a production-ready Azure infrastructure to deploy our microservices. We used an ARM Template to automatically set up the Azure Kubernetes Service, the Azure Container Registry, the Azure Load Balancer, Azure File Storage (which will be used for persistent data storage) and to add a DNS Zone. We applied some configuration files to authorize Kubernetes to pull Docker images from the Azure Container Registry, configure the NGINX Ingress Controller and set up a DNS Hostname for our Ingress Controller.

Chapter 6

Stupid Simple Scalability

This post will define and explain software scalability in Kubernetes and look at different scalability types. Then we will present three autoscaling methods in Kubernetes: **HPA** (Horizontal Pod Autoscaler), **VPA** (Vertical Pod Autoscaler), and **CA** (Cluster Autoscaler).

Scalability Explained

To understand the different concepts in software scalability, let's take a real-life example.

Suppose you've just opened a coffee shop, you bought a simple coffee machine, which can make three coffees per minute, and you hired an employee who serves the clients.

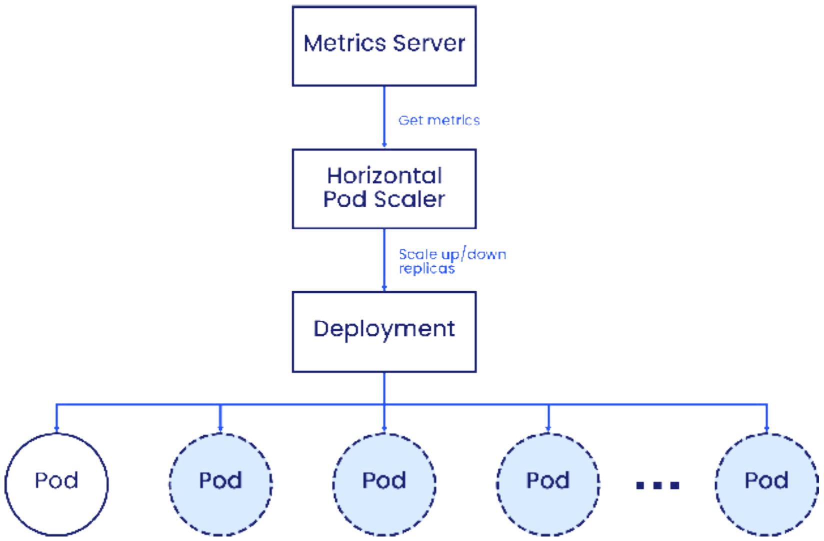
At first, you have a few clients: everything is going well, and all the people are happy about the coffee and the service because they don't have to wait too long to get their delicious coffee. As time goes by, your coffee shop becomes famous in town, and more and more people are buying their coffee from you. But there is a problem. You have only one employee and too many clients, so the waiting time gets considerably higher and people are starting to complain about your service. The coffee machine could make three coffees per minute, but the employee can handle only one client per minute. You decide to hire two more employees. With this, you've solved the problem for a while.

After some time, near the coffee shop, the city opens a fun park, so more and more tourists are coming and drinking their coffee in your famous coffee shop. So you decide to hire more people, but even with more employees, the waiting time is almost the same. The problem is that your coffee machine can make three coffees per minute, so now your employees are waiting for the coffee machine. The solution is to buy a new coffee machine. Another problem is that the clients tend to buy coffee from employees that they already know. As a result, some employees have a lot of work, and others are idle. This is when you decide you need to hire another employee who will greet the clients and redirect them to the employee who is free or has fewer orders to prepare.

Analyzing your income and expenses, you realize that you have many more clients during the summer than in the winter, so you decide to hire seasonal workers. Now you have three employees working full-time and the other employees are working for you only during the summer. This way, you can increase your income and decrease expenses. Furthermore, you can rent some coffee machines during the summer and give them back during the winter to minimize the costs. This way, you won’t have idle coffee machines.

To translate this short story to software scalability in Kubernetes, we can replace the coffee machines with nodes, the employees with pods, the coffee shop is the cluster, and the employee who greets the clients and redirects them is the load balancer. Adding more employees means Horizontal Pod Scaling; adding more coffee machines means Cluster Scaling. Seasonal workers and renting coffee machines only for the summer season means Autoscaling because when the load is higher, we have more pods to serve the clients and more nodes to be used by pods. When the load drops (during the winter), we have fewer expenses. In this analogy, Vertical Pod Scaling would be hiring a more experienced employee who can serve more clients in the same amount of time (high performing employee). The trigger for the Autoscaling would be the season; we scale up during the summer and scale down during the winter.

Horizontal Pod Autoscaling (HPA)



Horizontal scaling or **scaling out** means that the number of running pods dynamically increases or decreases as your application usage changes. To know exactly when to increase or decrease the number of replicas, Kubernetes uses triggers based on the observed metrics (average CPU utilization, average memory utilization, or custom metrics defined by the user). HPA, a Kubernetes resource, runs in a loop (the loop duration can be configured, by default, it is set to 15 seconds) and fetches the resource metrics from the resource metrics API for each pod. Using these metrics, it calculates the actual resource utilization values based on the mean values of all the pods and compares them to the metrics defined in the HPA definition. To calculate the desired number of replicas, HPA uses the following formula:

```
desiredReplicas = ceil[currentReplicas*(currentMet-
ricValue/desiredMetricValue)]
```

To understand this formula, let's take the following configuration:

```
spec:
  containers:
      - name: php-apache
        image: k8s.gcr.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
```

The unit suffix **m** stands for “thousandth of a core,” so this resources object specifies that the container process needs 200/1000 of a core (20%) and is allowed to use, at most, 500/1000 of a core (50 percent).

With the following command, we can create an HPA that maintains between 1 and 10 replicas. It will increase or decrease the number of replicas to maintain an average CPU usage of 50 percent, or in this concrete example, 100 milli-cores.

```
kubectl autoscale deployment deployment_name
--cpu-percent=50 --min=1 --max=10
```

Suppose that the CPU usage has increased to 210 percent; this means that we will have **nrReplicas = ceil[1 * (210 / 50)] = ceil[4.2] = 5 replicas.**

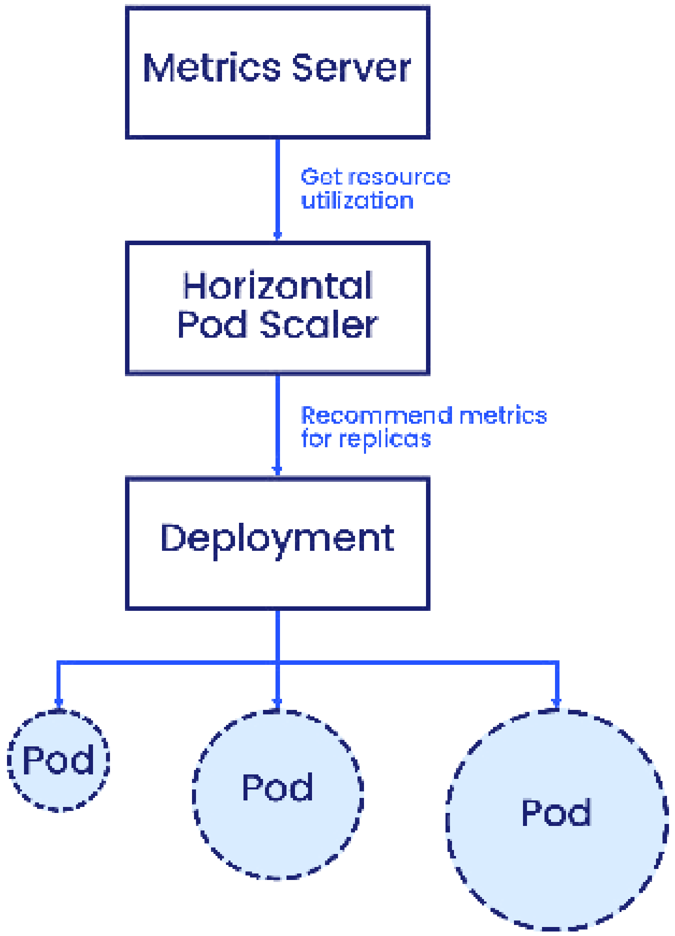
Now the CPU usage drops to 25 percent when having 5 replicas, so the HPA will decrease the number of replicas to **nrReplicas = ceil[5 * (25 / 50)] = ceil[2.5] = 3 replicas.**

For more examples, read Autoscaling in Kubernetes using HPA and VPA or HPA Walkthrough.

When configuring HPA, make sure that:

1. **All pods have resource requests and limits configured -** this will be taken into consideration when HPA takes scaling decisions
2. **Use custom metrics or observed metrics -** external metrics can be a security risk because they can provide access to a large number of metrics
3. **Use HPA together with CA whenever possible**

Vertical Pod Autoscaling (VPA)

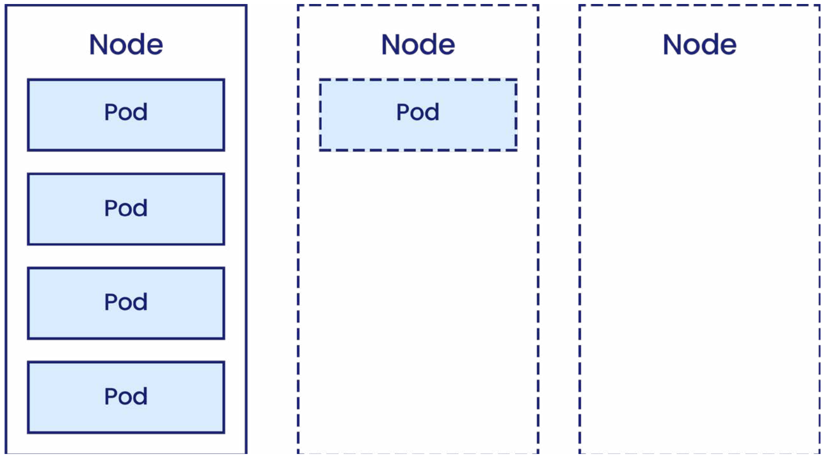


VPA recommends optimized CPU and memory requests/limits values (and automatically updates them for you so that the cluster resources are efficiently used). **VPA won't add more replicas of a Pod, but it increases the memory or CPU limits.** This is useful when adding more replicas won't help your solution. For example, sometimes you can't scale a database (read Chapter Three, Persistent Volumes Explained) just by adding more Pods. Still, you can make the database handle more connections by increasing the memory or CPU. You can use the VPA when your application serves heavyweight requests, which requires higher resources.

HPA can be useful when, for example, your application serves a large number of lightweight (i.e., low resource-consuming) requests. In that case, scaling the number of replicas can distribute the workload on each pod. The VPA, on the other hand, can be useful when your application serves heavyweight requests, which require higher resources.

HPA and VPA are incompatible. Do not use both together for the same set of pods. HPA uses the resource request and limits to trigger scaling, and in the meantime, VPA modifies those limits, so it will be a mess unless you configure the HPA to use either custom or external metrics. Read more about VPA and HPA here.

Cluster Autoscaling (CA)



While HPA scales the number of Pods, the **CA changes the number of nodes.** When your cluster runs low on resources, the CA provision a new computation unit (physical or virtual machine) and adds it to the cluster. If there are too many empty nodes, the CA will remove them to reduce costs.

Learn more about Cluster Autoscaling in Architecting Kubernetes Clusters—Choosing the Best Autoscaling Strategy.

Conclusion

In the first part of this chapter, we provided a **real-life example** to explain the different concepts used in **software scalability.** Then we defined and presented the three **scalability methods provided by Kubernetes, HPA** (Horizontal Pod Autoscaler), **VPA** (Vertical Pod Autoscaler), and **CA** (Cluster Autoscaler).

Chapter 7

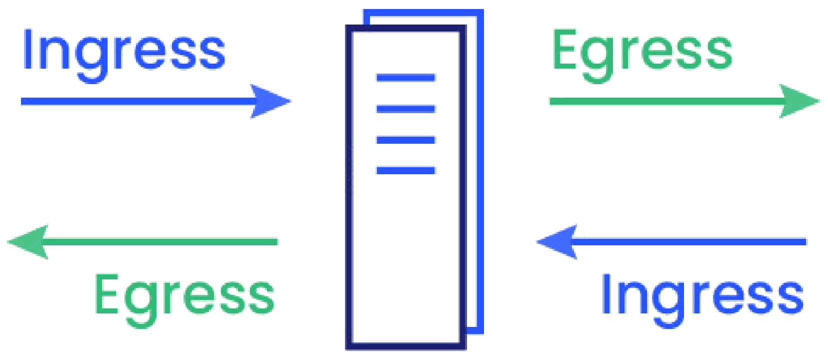
Stupid Simple Service Mesh - What, When, Why

Recently microservices-based applications became very popular and with the rise of microservices, the concept of Service Mesh also became a very hot topic. Unfortunately, there are only a few articles about this concept and most of them are hard to digest.

In this section, we will try to **demystify the concept of Service Mesh using "Stupid Simple" explanations**, diagrams, and examples to make this concept more transparent and accessible for everyone. In the **first chapter**, we will talk about **the basic building blocks of a Service Mesh** and we will implement a **sample application** to have a practical example of each theoretical concept. In the **next chapter**, based on this sample app, we will touch more advanced topics, like **Service Mesh in Kubernetes**, and we will talk about some more **advanced Service Mesh implementations like Istio**, **Linkerd**, etc.

To understand the concept of Service Mesh, the first step is to understand **what problems** it solves and **how it solves them**.

Software architecture has evolved a lot in a short time, from a classical monolithic architecture to microservices. Although many praise the microservice architecture as the holy grail of software development, it introduces some serious challenges.



Overview of the sample application

For one, a microservices-based architecture means that we have a **distributed system**. Every distributed system has challenges such as *transparency, security, scalability, troubleshooting, and identifying the root cause of issues*. In a monolithic system, we can find the root cause of a failure by tracing. But in a microservice-based system, each service can be written in different languages, so tracing is no trivial task. Another challenge is *service-to-service communication*. Instead of focusing on business logic, developers need to take care of *service discovery, handle connection errors, detect latency, retry logic*, etc. Applying SOLID principles on the **architecture level** means that these kinds of network problems **should be abstracted away and not mixed with the business logic**. This is why we need Service Mesh.

Ingress Controller vs. API Gateway vs. Service Mesh

As I mentioned above, we need to apply SOLID **principles on an architectural level**. For this, it is important to set the boundaries between Ingress Controller, API Gateway, and Service Mesh and understand each one's role and responsibility.

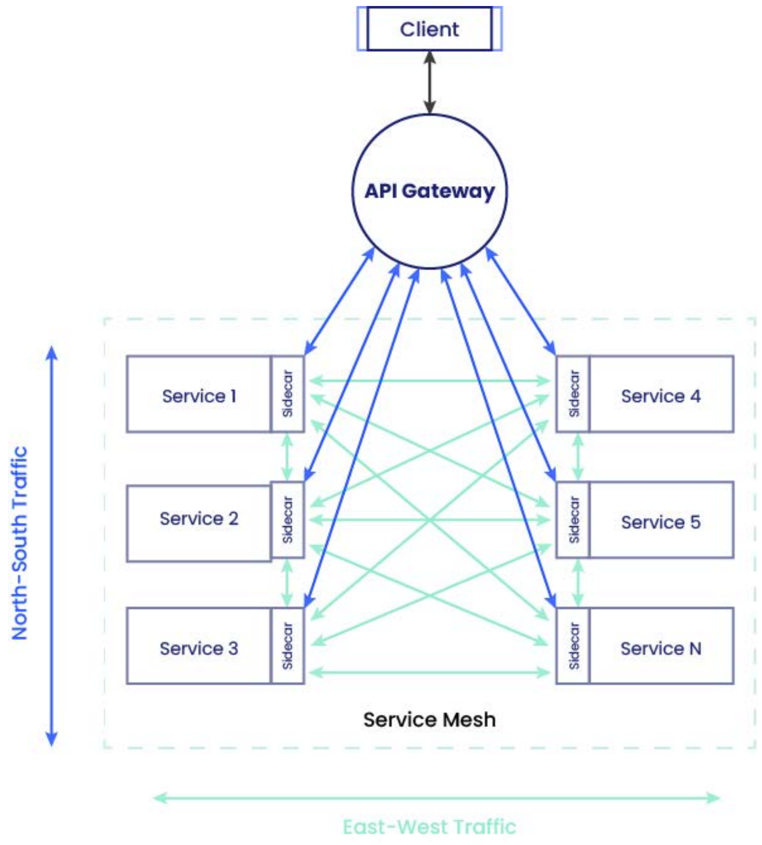
On a stupid simple and oversimplified level, these are the responsibilities of each concept:

1. Ingress Controller: allows a **single IP port to access all services from the cluster**, so its main responsibilities are path mapping, routing and simple load balancing, like a reverse proxy

2. API Gateway: **aggregates and abstracts away APIs**; other responsibilities are rate-limiting, authentication, and security, tracing, etc. In a microservices-based application, you need a way to distribute the requests to different services, gather the responses from multiple/all microservices, and then prepare the final response to be sent to the caller. This is what an API Gateway is meant to do. It is responsible for **client-to-service communication**, north-south traffic.

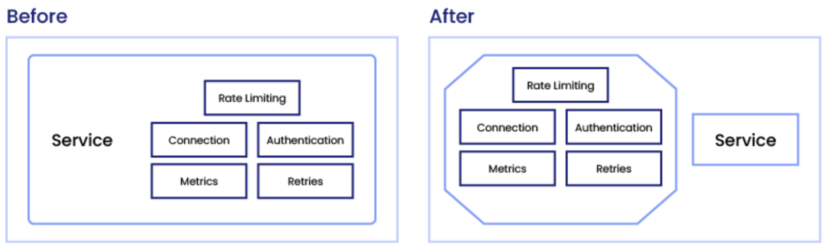
3. Service Mesh: responsible for **service-to-service communication**, east-west traffic. We'll dig more into the concept of Service Mesh in the next section.

Service Mesh and **API Gateway** have overlapping functionalities, such as rate limiting, security, service discovery, tracing, etc. but they work on different levels and solve different problems. **Service Mesh** is responsible for the **flow of requests between services**. **API Gateway** is responsible for the **flow of requests between the client and the services**, aggregating multiple services and creating and sending the final response to the client.



The main responsibility of an API gateway is to accept traffic from outside your network and distribute it internally, while the main responsibility of a service mesh is to route and manage traffic within your network. They are complementary concepts and a well-defined microservices-based system should combine them to ensure application uptime and resiliency while ensuring that your applications are easily consumable.

What does a Service Mesh Solve?



As an oversimplified and stupid simple definition, a Service Mesh is an *abstraction layer hiding away and separating networking-related logic from business logic*. This way developers can focus only on implementing business logic. We implement this abstraction using a **proxy**, which sits in the front of the service. It takes care of all the network-related problems. This allows the service to focus on what is really important:

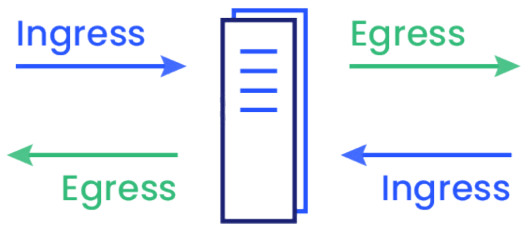
the business logic. In a microservice-based architecture, we have multiple services and each service has a proxy. Together, these *proxies are called Service Mesh*.

As best practices suggest, *proxy and service should be in separate containers*, so each container has a **single responsibility**. In the world of Kubernetes, the container of the proxy is implemented as a **sidecar**. This means that each service has a sidecar containing the proxy. A single Pod will contain two containers: the service and the sidecar. Another implementation is to use one proxy for multiple pods. In this case, the proxy can be implemented as a Deamonset. The most common solution is using sidecars. Personally, I prefer sidecars over Deamonsets, because they keep the logic of the proxy as simple as possible.

There are multiple Service Mesh solutions, including Istio, Linkerd, Consul, Kong, and Cilium. Let's focus on the basics and understand the concept of Service Mesh, starting with **Envoy**. This is a high-performance proxy and not a complete framework or solution for Service Meshes (in this tutorial, we will build our own Service Mesh solution). Some of the Service Mesh solutions use Envoy in the background (like Istio), so before starting with these higher-level solutions, it's a good idea to understand the low-level functioning.

Understanding Envoy

Ingress and Egress

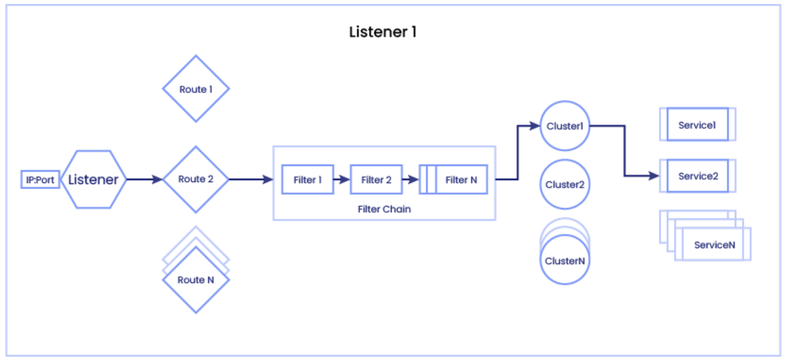


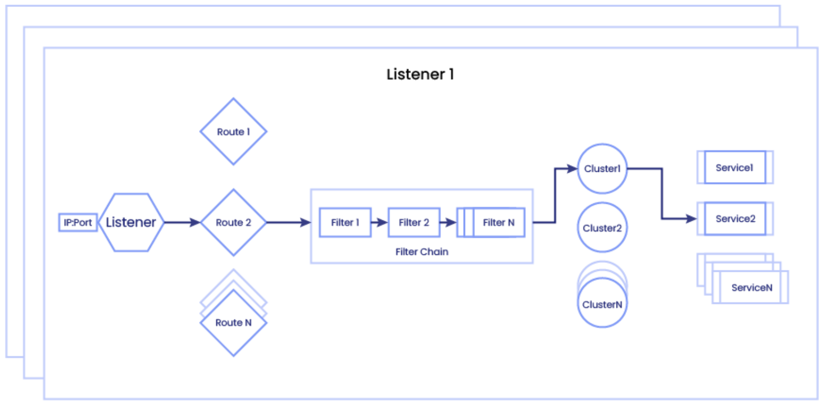
Simple definitions:

- Any traffic sent to the server (service) is called **ingress**.
- Any traffic sent from the server (service) is called **egress**.

The **Ingress** and the **Egress** rules should be added to the **configuration of the Envoy proxy**, so the sidecar will take care of these. This means that *any traffic* to the service *will first go to the Envoy sidecar*. Then the *Envoy proxy redirects the traffic to the real service*. Vice-versa, any traffic from this service will go to the Envoy proxy first and Envoy resolves the destination service using Service Discovery. By intercepting the inbound and outbound traffic, Envoy can implement service discovery, circuit breaker, rate limiting, etc.

The Structure of an Envoy Proxy Configuration File





Every Envoy configuration file has the following components:

1. **Listeners**: where we configure the IP and the Portnumber that the Envoy proxy listens to

2. **Routes**: the received request will be routed to a cluster based on rules. For example, we can have path matching rules and prefix rewrite rules to select the service that should handle a request for a specific path/subdomain. Actually, the **route is just another type of filter**, which is mandatory. Otherwise, the proxy doesn't know where to route our request.

3. **Filters**: Filters can be chained and are used to enforce different rules, such as rate-limiting, route mutation, manipulation of the requests, etc.

4. **Clusters**: act as a manager for a group of logically similar services (the cluster has similar responsibility as a service in Kubernetes; it defines the way a service can be accessed), and acts as a load balancer between the services.

5. **Service/Host**: the concrete service that handles and responds to the request

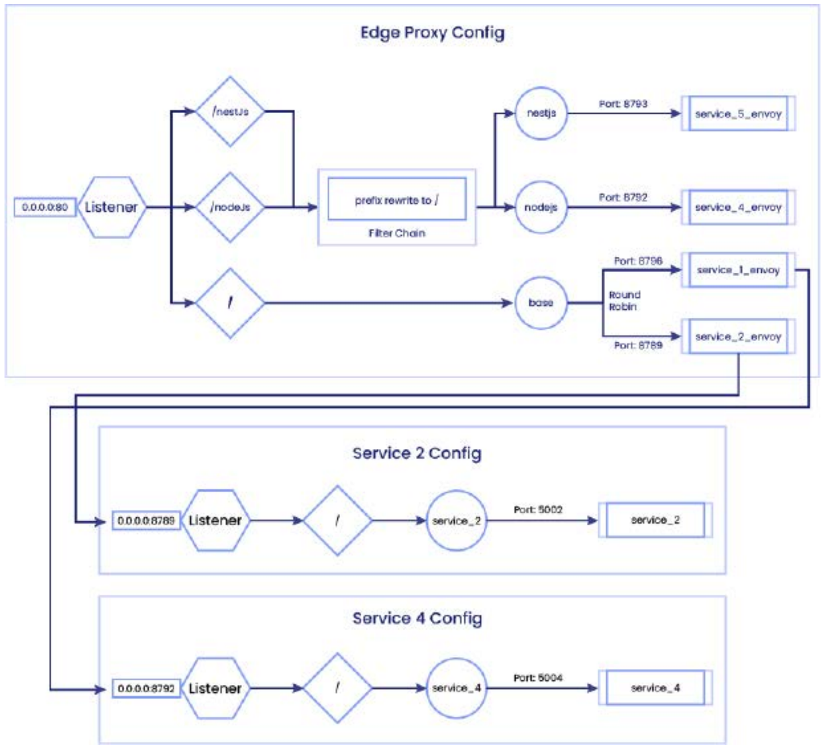
Here is an example of an Envoy configuration file:

```
---
admin:
  access_log_path: “/tmp/admin_access.log”
  address:
    socket_address:
      address: “127.0.0.1”
      port_value: 9901
static_resources:
  listeners:
   -
      name: “http_listener”
      address:
        socket_address:
          address: “0.0.0.0”
          port_value: 80
      filter_chains:
        filters:
          -
            name: “envoy.http_connection_manager”
            config:
              stat_prefix: “ingress”
              codec_type: “AUTO”
              generate_request_id: true
              route_config:
                name: “local_route”
                virtual_hosts:
                  -
                    name: “http-route”
                    domains:
                      - “*”
                    routes:
                      -
                        match:
                          prefix: “/nestjs”
                        route:
                          prefix_rewrite: “/”
                          cluster: “nestjs”
                      -
                        match:
                            prefix: “/nodejs”
                          route:
                            prefix_rewrite: “/”
                            cluster: “nodejs”
                       -
```

```
                       match:
                         path: “/”
                       route:
                         cluster: “base”
            http_filters:
              -
                name: “envoy.router”
                config: {}

clusters:
  -
    name: “base”
    connect_timeout: “0.25s”
    type: “strict_dns”
    lb_policy: “ROUND_ROBIN”
    hosts:
      -
        socket_address:
          address: “service_1_envoy”
          port_value: 8786
      -
        socket_address:
          address: “service_2_envoy”
          port_value: 8789
  -
    name: “nodejs”
    connect_timeout: “0.25s”
    type: “strict_dns”
    lb_policy: “ROUND_ROBIN”
    hosts:
      -
        socket_address:
          address: “service_4_envoy”
          port_value: 8792
  -
    name: “nestjs”
    connect_timeout: “0.25s”
    type: “strict_dns”
    lb_policy: “ROUND_ROBIN”
    hosts:
      -
        socket_address:
          address: “service_5_envoy”
          port_value: 8793
```

The configuration file above translates into the following diagram:



This diagram did not include all configuration files for all the services, but it is enough to understand the basics. You can find this code in my Stupid Simple Service Mesh repository.

As you can see, between *lines 10-15* we defined the **Listener** for our Envoy proxy. Because we are working in Docker, the host is 0.0.0.0.

After configuring the listener, between *lines 15-52* we define the **Filters**. For simplicity we used only the basic filters, **to match the routes and to rewrite the target routes**. In this case, if the subdomain is "host:port/nodeJs," the router will choose the *nodejs* cluster and the URL will be rewritten to "host:port/" (this way the request for the concrete service won't contain the /nodesJs part). The logic is the same also in the case of "host:port/nestJs". If we don't have a subdomain in the request, then the request will be routed to the cluster called *base* without prefix rewrite filter.

Between lines 53-89 we defined the **clusters**. The *base* cluster will have two services and the chosen load balancing strategy is *round-robin*. Other available strategies can be found here. The other two clusters (*nodejs* and *nestjs*) are simple, with only a single service.

The **complete code** for this tutorial can be found in my Stupid Simple Service Mesh git repository.

Conclusion

In this chapter, we learned about the **basic concepts of Service Mesh**. In the first part, we understood the **responsibilities and differences between the Ingress Controller**, **API Gateway, and Service Mesh**. Then we talked about what **Service Mesh** is and **what problems it solves**. In the second part, we introduced Envoy, a performant and popular proxy, which we used to build our Service Mesh example. We learned about the **different parts of the Envoy configuration files** and **created a Service Mesh with five example services** and a **front-facing edge proxy**.

In the **next chapter**, we will look at how to use **Service Mesh with Kubernetes** and will create an example project that can be used as a **starting point in any project using microservices**.

Chapter 8

Stupid Simple Service Mesh in Kubernetes

Stupid Simple Service Mesh in Kubernetes

We covered the what, when and why of Service Mesh in an earlier chapter. Now I'd like to talk about why they are critical in Kubernetes.

To understand the importance of using service meshes when working with microservices-based applications, let's start with a story.

Suppose that you are working on a big microservices-based banking application, where any mistake can have serious impacts. One day the development team receives a feature request to add a rating functionality to the application. The solution is obvious: create a new microservice that can handle user ratings. Now comes the hard part. The team must come up with a reasonable time estimate to add this new service.

The team estimates that the rating system can be finished in 4 sprints. The manager is angry. He cannot understand why it is so hard to add a simple rating functionality to the app.

To understand the estimate, let's understand what we need to do in order to have a functional rating microservice. The CRUD (Create, Read, Update, Delete) part is easy -- just simple coding. But adding this new project to our microservices-based application is not trivial. First, we have to implement authentication and authorization, then we need some kind of tracing to understand what is happening in our application. Because the network is not reliable (unstable connections can result in data loss), we have to think about solutions for retries, circuit breakers, timeouts, etc.

We also need to think about deployment strategies. Maybe we want to use shadow deployments to test our code in production without impacting the users. Maybe we want to add A/B testing capabilities or canary deployments. So even if we create just a simple microservice, there are lots of cross-cutting concerns that we have to keep in mind.

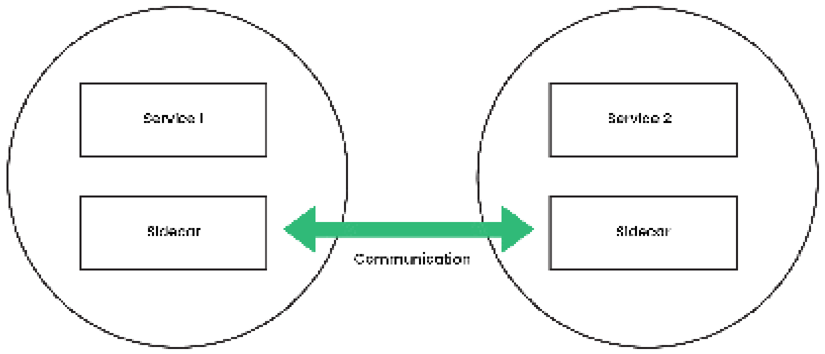
Sometimes it is much easier to add a new functionality to an existing service, than create a new service and add it to our infrastructure. It can take a lot of time to deploy a new service, to add authentication and authorization, to configure tracing, to create CI/CD pipelines, to implement retry mechanisms and more. But adding the new feature to an existing service will make the service too big. It will also break the rule of single responsibility, and like many existing microservices projects, it will be transformed into a set of connected macroservices or monoliths.

We call this the cross-cutting concerns burden — the fact that in each microservice you must reimplement the cross-cutting concerns, such as authentication, authorization, retry mechanisms and rate limiting.

What is the solution for this burden? Is there a way to implement all these concerns once and inject them into every microservice, so the development team can focus on producing business value? The answer is Istio.

Set Up a Service Mesh in Kubernetes using Istio

Istio solves these issues using **sidecars**, which it automatically injects into your pods. Your services won't communicate directly with each other — they'll communicate through sidecars. The sidecars will handle all the cross-cutting concerns. You define the rules once, and these rules will be injected automatically into all of your pods.

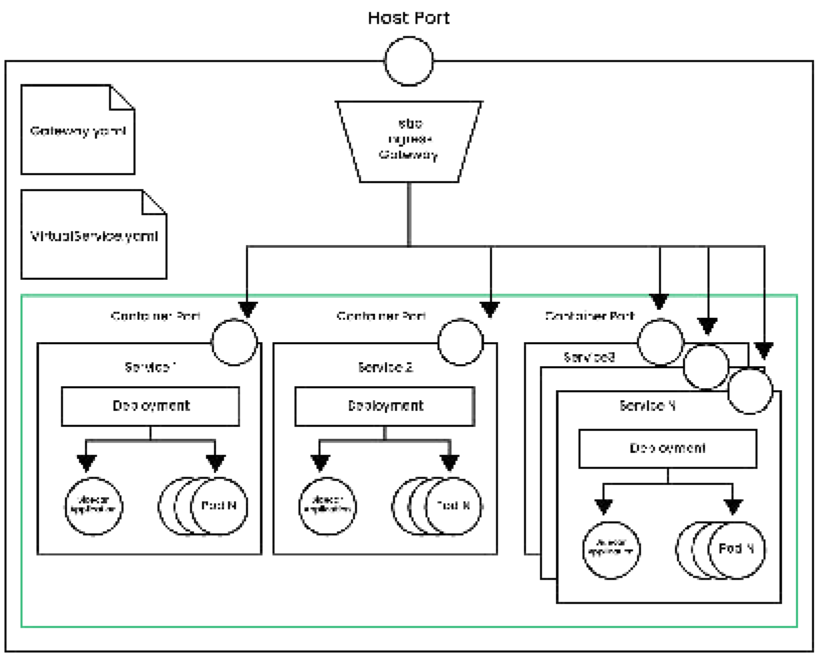


Samples Application

Let's put this idea into practice. We'll build a sample application to explain the basic functionalities and structure of Istio.

In the previous chapter, we created a service mesh by hand, using envoy proxies. In this tutorial, we will use the same services, but we will configure our Service Mesh using **Istio** and **Kubernetes**.

The image below depicts that application architecture.



Requirements

To work along with this tutorial, you will need to install the following tools:

1. Kubernetes (we used the 1.21.3 version in this tutorial)
2. Helm (we used the v2)
3. Istio (we used 1.1.17) - setup tutorial
4. Minikube, K3s or Kubernetes cluster enabled in Docker

Git Repository

My **Stupid Simple Service Mesh in Kubernetes** repository contains all the scripts for this tutorial. Based on these scripts you can configure any project.

Running our Microservices-Based Project using Istio and Kubernetes

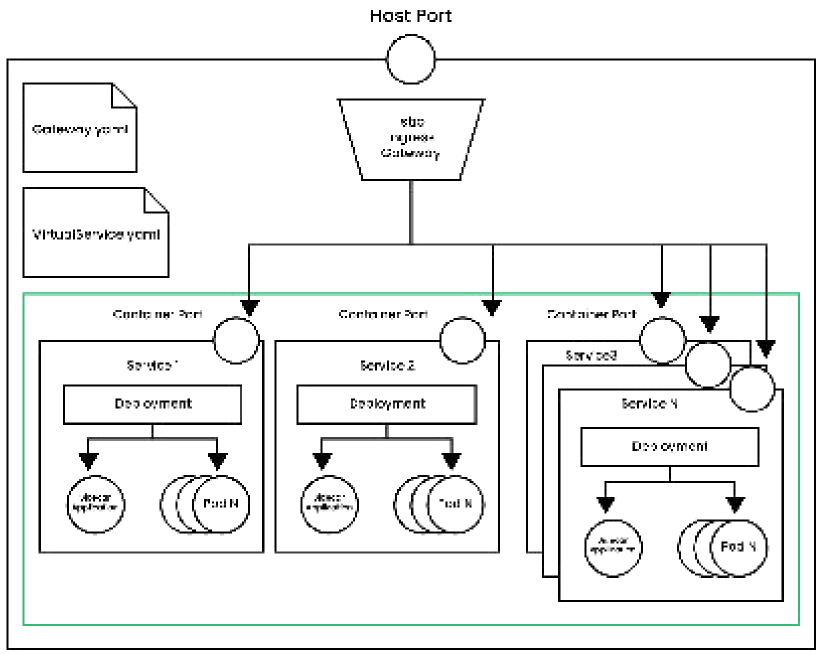
As I mentioned above, step one is to configure Istio to inject the sidecars into each of your pods from a namespace. We will use the default namespace. This can be done using the following command:

```
kubectl label namespace default istio-injection=enabled
```

In the second step, we navigate into the /kubernetes folder from the downloaded repository, and we apply the configuration files for our services:

```
kubectl apply -f service1.yaml
kubectl apply -f service2.yaml
kubectl apply -f service3.yaml
```

After these steps, we will have the green part up and running:



For now, we can't access our services from the browser. In the next step, we will configure the **Istio Ingress and Gateway**, allowing traffic from the exterior.

The **gateway configuration** is as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
    name: http-gateway
spec:
    selector:
        istio: ingressgateway
    servers:
        - port:
            number: 80
            name: http
            protocol: HTTP
        hosts:    - “*”
```

Using the selector *istio: ingressgateway*, we specify that we would like to use the default ingress gateway controller, which was automatically added when we installed Istio. As you can see, the gateway allows traffic on port 80, but it doesn't know where **to route the requests**. To define the routes, **we need a so-called VirtualService**, which is another custom Kubernetes resource defined by Istio.

```
apiVersion: networking.istio.io/v1b
kind: VirtualService
metadata:
    name: sssm-virtual-services
spec:
    hosts:  - “*”
    gateways:  - http-gateway
    http:
        - match:
            - uri:
                prefix: /service1
            route:
                - destination:
                    host: service1
                    port:
                        number: 80
        - match:
            - uri:
                prefix: /service2
            route:
                - destination:
                    host: service2
                    port:
                        number: 80
```

The code above shows an example configuration for the VirtualService. In line 7, we specified that the virtual service applies to the requests coming from the gateway called http-gateway and from line 8 we define the rules to match the services where the requests should be sent. Every request with /service1 will be routed to the service1 container while every requests with /service2 will be routed to the service2 container.

At this step, we have a working application. Until now there is nothing special about Istio — you can get the same architecture with a simple Kubernetes Ingress controller, without the burden of sidecars and gateway configuration.

Now let’s see what we can do using Istio rules.

Security in Istio

Without Istio, every microservice must implement authentication and authorization. Istio removes the responsibility of adding authentication and authorization from the main container (so developers can focus on providing business value) and moves these responsibilities into its sidecars. The sidecars can be configured to request the access token at each call, making sure that only authenticated requests can reach our services.

```
apiVersion: authentication.istio.io/v1beta1
kind: Policy
metadata:
    name: auth-policy
spec:
    targets:
        - name: service1
        - name: service2
        - name: service3
        - name: service4
        - name: service5
    origins:
    - jwt:
        issuer: “{YOUR_DOMAIN}”
        jwksUri: “{YOUR_JWT_URI}”
    principalBinding: USE_ORIGIN
```

As an identity and access management server, you can use Auth0, Okta or other **OAuth** providers. You can learn more about authentication and authorization using Auth0 with Istio in this article.

Traffic Management using Destination Rules

Istio's official documentation says that the **DestinationRule** "*defines policies that apply to traffic intended for a service after routing has occurred.*" This means that the DestionationRule resource is **situated** somewhere **between the Ingress controller and our services**. Using DestinationRules, we can define policies for *load balancing, rate limiting or even outlier detection* to detect unhealthy hosts.

Shadowing

Shadowing, also called Mirroring, is useful when you want to test your changes in production silently, without affecting end users. All the requests sent to the main service are mirrored (a copy of the request) to the secondary service that you want to test.

Shadowing is easily achieved by defining a destination rule using subsets and a virtual service defining the mirroring route. The destination rule will be defined as follows:

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
    name: service2
spec:
    host: service2
    subsets:
    - name: v1
      labels:
          version: v1
    - name: v2
      labels:
          version: v2
```

As we can see above, we defined two subsets for the two versions.

Now we define the virtual service with mirroring configuration, like in the script below:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
    name: service2
spec:
    hosts:
    - service2
    http:
    - route:
        - destination:
          host: service2
          subset: v1
        mirror:
            host: service2
            subset: v2
```

In this virtual service, we defined the main destination route for service2 version v1. The mirroring service will be the same service, but with the v2 version tag. This way the end user will interact with the v1 service, while the request will also be sent also to the v2 service for testing.

Traffic Splitting

Traffic splitting is a technique used to test your new version of a service by **letting only a small part (a subset) of users to interact** with the new service. This way, if there is a bug in the new service, only a small subset of end users will be affected.

This can be achieved by modifying our virtual service as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
    name: service2
spec:
    hosts:
    - service2
    http:
    - route:
        - destination:
              host: service2
              subset: v1
         weight: 90
         - destination:
               host: service2
               subset: v2
         weight: 10
```

The most important part of the script is the **weight** tag, which defines the percentage of the requests that will reach that specific service instance. In our case, 90 percent of the request will go to the v1 service, while only 10 percent of the requests will go to v2 service.

Canary Deployments

In canary deployments, newer versions of services are **incrementally rolled** out to users to minimize the risk and impact of any bugs introduced by the newer version.

This can be achieved by **gradually decreasing the weight of the old version while increasing the weight of the new version**.

A/B Testing

This technique is used when we have two or more different user interfaces and we would like to test which one offers a better user experience. We deploy all the different versions and we collect metrics about the user interaction. A/B testing can be configured using a load balancer based on consistent hashing or by using subsets.

In the first approach, we define the load balancer like in the following script:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
    name: service2
spec:
    host: service2
    trafficPolicy:
        loadBalancer:
            consistentHash:
                httpHeaderName: version
```

As you can see, the consistent hashing is based on the version tag, so this tag must be added to our service called "service2", like this (in the repository you will find two files called service2_v1 and service2_v2 for the two different versions that we use):

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: service2-v2
    labels:
        app: service2
spec:
    selector:
        matchLabels:
            app: service2
    strategy:
        type: Recreate
    template:
        metadata:
            labels:
                app: service2
                version: v2
        spec:
            containers:
            - image: zoliczako/sssm-service2:1.0.0
              imagePullPolicy: Always
              name: service2
              ports:
              - containerPort: 5002
              resources:
                  limits:
                      memory: "256Mi"
                      cpu: "500m"
```

The most important part to notice is the spec -> template -> metadata -> version: v2. The other service has the version: v1 tag.

The other solution is based on subsets.

Retry Management

Using Istio, we can easily define the maximum number of attempts to connect to a service if the initial attempt fails (for example, in case of overloaded service or network error).

The retry strategy can be defined by adding the following lines to the end of our virtual service:

```
retries:
    attempts: 5
    perTryTimeout: 10s
```

With this configuration, our service2 will have five retry attempts in case of failure and it will wait 10 seconds before returning a timeout.

Learn more about traffic management in **this article**. You'll find a great workshop to configure an end-to-end service mesh using Istio **here**.

Conclusion

In this chapter, we learned how to set up and configure a service mesh in Kubernetes using Istio. First, we configured an ingress controller and gateway and then we learned about traffic management using destination rules and virtual services.

Conclusion

Become a Microservices Master

You've made it through our Stupid Simple Kubernetes e-book. Congratulations! You are well on your way to becoming a microservices master.

There are many more resources available to further your learning Microservices, including the **Microservices.io** website. Similarly, there are many Kubernetes resources out there. One of our favorites is **The Illustrated Children's Guide to Kubernetes video**.

I strongly encourage you to get hands on and continue your learning. The SUSE & Rancher Community is a great place to start – and is welcoming to learners at all levels. Whether you are interested in an introductory Kubernetes class or ready to go deeper with a mutli-week class on K3s, they've got it all. **Join the free community today**!

Keep learning and keep it simple!

Zoltán Czakó

Zoltán Czakó is a software developer experienced in backend, frontend, DevOps, artificial intelligence and machine Learning. He is the founder of HumindZ, a company focused on making Artificial Intelligence and Machine Learning accessible for everyone, providing services to improve everyday life using the power of AI/ML.

He is also a research assistant at the Technical University of Cluj-Napoca in Romania, where he is applying his skills to create a platform that combines No-Code AI with AutoAI. Using this platform, the research team creates automated solutions mainly for healthcare, automating the diagnosis of different diseases, this way helping to improve the lives of thousands of people.

During his career, Zoltán has worked on multiple microservices-based projects. He wrote this book to help others get started with microservices and to make Kubernetes simple and accessible for everyone.