Corndel DevOps Engineering Programme

in association with Softwire

Module 14: Advanced Immutable Infrastructure

Corndel Digital.

DB63

Introduction

In the previous module about immutable infrastructure we had a look at **containers** and how they provide a self-contained environment for an application and its dependencies that is isolated from other processes that are running on the host machine. This containerisation of an application makes it easy to create multiple instances of the application by spinning up new containers, which will behave in the same way as the existing instances of the application. This ability allows us to scale the application to meet demand.

We could manually monitor the utilisation of each application and manually spin up containers, but this would be inefficient, time-consuming and error-prone. Like most things we've encountered on this course, we will want to automate it. This automation of the management of containers is called **container orchestration**. However, it is not just limited to the scaling of a containerised application, but the automation of many other facets related to a container's lifecycle.

Imagine you are working for a large company that provides a global video-streaming platform. To maintain and develop all their systems they have 100s of engineers working for them. The platform is made up of various microservices, which enables a small team of developers to be responsible for each microservice rather than every developer needing to be able to work on the entire system. Each microservice is packaged and deployed as a container. In an organisation like this there can easily be hundreds of services: services responsible for authentication, user sign up, delivering video content, recording user favourites, etc. Each service may need multiple instances to handle demand, meaning that in a large organisation there can be hundreds, if not thousands, of containers that need to be managed.

These containers will be running on host machines, which could be virtual machines in the cloud or bare-metal machines in the company's own infrastructure. How do we ensure that the resources of these host machines are used efficiently? What happens if a host machine fails? How is availability of the services that were running on that machine maintained?

Each service may need to communicate with one or more other services within the platform. How does the service know where the instances of the other services are located and how to send messages to them? How is traffic distributed across all instances of that service so that one instance isn't overloaded compared to the others?

Each team within the organisation will be actively developing the service they are looking after, as well as updating dependencies, applying security patches and generally maintaining the services. This means that each team will be deploying changes regularly. Across the whole company this could be tens or hundreds of deployments each day that are happening. Each deployment must happen while ensuring that no disruption happens to the platform, making sure it is available throughout.

It is these problems, as well as scalability, that container orchestration attempts to solve.

Table of Contents

Core Material

Chapter 1

Container Orchestration

There are a few different container orchestration tools, which we will discuss later, but they all have the aim of trying to automate the deployment, scaling, networking and availability of containers. They mostly take configuration (normally a YAML or JSON file) that describes the desired state of the containers. This could include the number of required instances, any required resources, where to retrieve the container image from, and where to store logs.

Provisioning

Container orchestration tools tend to look after a collection of **host** machines, often referred to as a **cluster**. These hosts are the machines that run the containers themselves. They could be bare-metal or VMs, and they could be hosted locally or in the cloud.

When a user wishes to deploy a new container into the cluster, they add configuration to the orchestration tool. This configuration tells the orchestration tool where it can retrieve the container image, as well as any *host requirements* needed by the container. The orchestration tool then creates the desired numbers of containers on host machines that meet the requirements.

Typical host requirements include the amount of available RAM, the number of available CPU threads, and the number of container instances.

Monitoring application load and scaling

Once a set of containers has been deployed, the orchestration tool will monitor the load being handled by each host. The tool may move containers between hosts to make sure that the load is balanced between all hosts.

The tool will also monitor the load going to a particular application. If the load on an application is high, then the tool can add more container instances (*scaling up*). If the load is low, then the tool could remove containers to free up host resources (*scaling down*).

The rules for scaling are defined in the tool's configuration. This could specify the minimum and maximum number of instances, or the amount of resource utilisation of the current containers that should cause a new instance to be created.

Health monitoring

As well as monitoring the resource utilisation across the cluster, the orchestration tool may also monitor the health of both the host machines and the containers. If a host machine becomes faulty or goes down completely, then the orchestration tool will allocate that host's containers to other hosts, ensuring that they remain available.

High error rates on a host's containers can indicate a hardware fault, such as failing RAM.

Redundancy and availability

Most orchestration tools allow you to define rules about how you wish to spread instances of a container across host machines. For example, you could specify that you want 6 instances of a container, but you want them spread over at least 3 host machines. This approach allows your system to have redundancy in it, in case one of the host machines fails. This, combined with health monitoring of the hosts and the orchestration tool moving any containers to new hosts when a host machine fails, allows the system to maintain high availability.

One thing to be aware of is the availability of the orchestration tool itself. Some default setups of orchestration tools only use one machine to host the tool, creating a single point of failure, as the whole system could become unavailable if this machine fails. Although the default may be to run the tool on only one machine, most tools have the ability to run on a multi-machine setup that avoids this single point of failure. This must be considered when planning a system that requires a high amount of availability.

Container updates

As mentioned previously, in a large organisation there might be hundreds of changes to applications each day. These could range from code changes introducing new features, to applying security patches to dependencies within an application's container. Each change requires a new container to be created and deployed into the cluster, replacing the old container. However, this must be done in a seamless way that doesn't cause any downtime for the system. Two of the ways that container orchestration tools achieve this are **blue/green deployments** or **rolling updates**.

Blue-green deployments

When a new version of a container needs to be deployed, the orchestration tool creates the same number of instances of the new version as there are of the existing version. This leads to two groups of containers: the old group and the new group, which we'll call the blue group and green group, respectively.

We previously covered blue-green deployments in Module 05 and Module 08.

Initially, the blue group still receives user traffic. When the green group is up and running and has been tested, user traffic is directed to the green group instead, without any downtime.

During the next release, the new container version is deployed to the blue group. Once the blue group is ready, user traffic is routed to it instead of the green group. Essentially, there are two groups that take turns being the production group. If you experience problems with a newly deployed container version, you can switch the traffic back to the other group, ensuring that the system remains available.

In practice, you will often not need both groups to permanently exist within your cluster. Once you have switched traffic to the new group and are confident that it is working correctly, you can remove the old group until you need it for the next deployment.

Disadvantages of blue-green deployments

Blue-green deployments have a couple disadvantages:

- When deploying, you need the resources to run both blue and green groups of containers at the same time. This essentially doubles the resource requirements (although it may only be only for a short time).
- Every user is switched over to the new version at once. If there is a problem in the new version of the containers that hasn't been picked up by testing, then every user will be affected.

These are the problems that rolling updates attempts to resolve.

Rolling updates

Instead of creating a whole new group of containers to switch to in one go, the containers are replaced one-by-one. The orchestration tool first spins up an instance of the new container version. Once the new container is ready, a small percentage of traffic is directed to it and an existing container is spun down.

The tool then monitors the new container. If the container has issues then it can be rolled back to the old version. In the meantime, only the proportion of users directed to the new container instance are affected. If the new container instance has no issues and is stable, the tool then repeats the process for the next container. This continues until all containers have been replaced by the new version.

Service discovery and load balancing

Let’s say you have an application, call it 'Service A', which communicates with another application, 'Service B'. If Service B is installed directly on a machine, 'Machine Z', then it is quite easy to tell Service A that Service B is on Machine Z and that's where it should send any requests.

However, when using container orchestration there may be multiple instances of Service B running in containers on multiple different host machines. Additionally, the container orchestration tool may move, add, or remove containers for resource efficiency and scaling. This means that the locations of Service B's instances are not guaranteed, and may change frequently.

Container orchestration tools handle this by providing **service discovery**, which allows the services and applications hosted within the cluster to be located by other services.

As there can be multiple instances of the same container within a cluster, most container orchestration tools provide **load balancing**. It would be pointless to have multiple instances of a container if all traffic went to only one of them, so the responsibility of the load balancer is to make sure that traffic is spread out across the available instances.

Configuration and secrets management

Environment-specific configuration should be decoupled from an application and its container images, instead being passed at runtime. Most container orchestration tools provide mechanisms to define this configuration and pass it to containers.

Secrets are a subset of configuration that contain sensitive information and require extra security. Some container orchestration tools provide higher security mechanisms for passing these secret values to containers. These will be covered in more detail later in the module.

Database connection strings and API keys are common secrets.

Chapter 2

Options

There are a variety of different container orchestration tools. Some are open-source tools that can be run on your own infrastructure or on cloud provided VMs. The main cloud infrastructure providers offer managed versions of some of those open source tools, as well as their own proprietary platforms.

Choosing a container orchestration platform is often as much an organisational decision as a technical one. For example, an organisation may want to make sure the same orchestration technology is used throughout their organisation to prevent hosting fragmentation, where every team uses a different tool. This can help when developers move between teams and ensures that knowledge can be shared throughout the organisation.

Another consideration may be that an organisation wants to use an open source tool that can be transferred between different cloud providers, to prevent being locked-in to a particular vendor.

Open-source

- **Docker Swarm** - Docker's built-in orchestration tool. Allows a group of physical or virtual machines to be joined together in a cluster and Docker containers to be run on it.
- **Apache Mesos** and **Marathon** - Most container orchestration tools provide two distinct, but related, pieces of functionality. The first is to abstract away individual machines, normally into the concept of a *cluster* and the second is to manage the containers running within the cluster. Apache Mesos is a cluster management system that focuses on the first of these tasks, abstracting all the compute resources away from individual machines (VMs or physical), so they can be managed as a single pool of resources. Marathon is a framework that runs on Apache Mesos, which provides the second of the aforementioned tasks - the container management itself.
- **Kubernetes** - a popular container orchestration tool developed by Google. Kubernetes can be used with multiple different container runtimes, although Docker remains the most popular. We will be taking a deeper look at Kubernetes later in this module.

Managed Kubernetes services

Most major cloud infrastructure providers offer a managed Kubernetes service. These services tend to offer easier integration with other services provided by that cloud provider, such as role-based access control (RBAC), secrets management, application load-balancers and provisioning of VMs. You tend to get more 'out-of-the-box' with these managed services. However, compared to managing your own Kubernetes service, this comes at the expense of portability.

1. **Google Kubernetes Engine (GKE)**
2. **AWS Elastic Kubernetes Service (EKS)**
3. **Azure Kubernetes Service (AKS)**

Proprietary cloud-based services

Cloud infrastructure providers all tend to have their own proprietary container management tools, as well as their managed Kubernetes services. These container management tools are normally simpler to use than tools like Kubernetes. For smaller organisations with fewer applications/containers to manage, they can provide the advantages of container management without needing to invest a lot of time in learning and maintaining a more complex tool like Kubernetes.

- **AWS ECS** - Amazon Elastic Container Service can be used to run docker images in the cloud at scale, without having to manage the cluster of machines it is running on.
- **Azure App Service** - Microsoft's fully managed web app Platform as a Service (PaaS). Intended for hosting, at scale, web apps and APIs, which can be provided either as a container or as code. This means that it isn't a good fit if you are looking for a container orchestration system for generic containerised applications.

Chapter 3
Kubernetes

Kubernetes is one of the most popular container orchestration tools around at the moment, and every major cloud provider offers a managed version of it, so we shall be taking a deeper look at it.

Kubernetes is often abbreviated to K8s, where the '8' represents the 8 letters between the 'K' and the 's'.

Terminology

Nodes

A Node is a worker machine in Kubernetes, which may be either a virtual machine or a physical machine.

Cluster

When you deploy Kubernetes you get a cluster, which is a set of one or more Nodes.

Pods

A Pod represents a group of one or more application containers (such as Docker) and some shared resources, which includes:

- Shared storage
- Networking, including a unique IP address
- Specifications for how each container runs, such as the image version and ports to use

A Pod is the smallest building block of Kubernetes and is the object that Kubernetes manages the lifecycle of (rather than directly managing the containers themselves). Kubernetes assigns Pods onto Nodes, taking into account available resources and other specified rules, with Nodes usually running multiple Pods. Horizontal scaling is achieved in Kubernetes by running replicas of Pods, and this can lead to high availability when the replica Pods are then assigned to multiple Nodes.

Most of the time you will want to use a *one-container-per-Pod* approach, where the Pod is a "wrapper" for a single container. However, the *multi-container-Pod* approach can be useful in certain situations where you have tightly-coupled containers that need to share resources and work as a single, cohesive unit of service. Example use cases include proxies, logging, monitoring agents and data loaders.

Control plane

The control plane can be thought of as the "brains" of Kubernetes. It makes the global decisions about the cluster and manages the lifecycles of the Pods. The control plane is made up of a number of components. These components could be installed on any of the machines within the cluster, but they are usually installed on a *master* Node. This Node is responsible for just the control plane components and does not run any Pods.

In a production setup, to allow for high availability, it is usual to have Kubernetes use a multi-master setup that has replicas of the master Node. If the master Node fails then one of the replicas can take over and the cluster remains available.

Labels

Often a user of Kubernetes will want to attach their own metadata to resources, such as Pods. You can do this in Kubernetes using labels, which are key-value pairs.

Label selectors can then be used to filter resources by the labels, which may be useful for finding all Pods that run a particular application or all Pods for a certain customer.

Services

Consider a Pod that contains a containerised application that processes images. Within our Kubernetes cluster we may have multiple instances of that Pod running. Consumers of that image processing application need to know how to connect to it, which is a problem made harder by Pods being non-permanent resources, Kubernetes may destroy a Pod on a Node that has a lot of resource usage and then create a new instance on another Node.

Consumers of the image processing application should not need to care about how many instances of it there are and they should not have to track which Node the Pod has just been moved to. This is where **Services** come in.

A Service is an abstraction that defines a logical set of Pods and a way to access them. The most common way to define a Service is to use a label selector to select the Pods you want to be in the Service. For the example above, we could use a selector that selects all the instances of the image processing application.

When creating a Service, Kubernetes gives the Service a DNS name that can be used to connect to it. Kubernetes then routes each incoming request to one of the available Pod instances in the Service. The requests are load balanced across the available Pods in the Service.

The use of Services means that consumers of a Service do not need to worry about the inner workings of Kubernetes. They can interface at the Service level while Kubernetes handles load balancing and tracking where the instances of the Pods actually are.

Controllers

A controller is a process that monitors the state of a cluster. If a cluster is not in the desired state, then the controller will make changes in order to move the cluster to the desired state. For example, the *Node controller* is responsible for noticing when a Node goes down and responding to that situation.

Kubernetes Components

Kubernetes is made up of various components that work together to manage a Kubernetes cluster. Some of those components make up the control plane and, as mentioned above, usually run on their own master Nodes, whereas the other components live on the worker Nodes themselves.

Control plane components

kube-apiserver

This is the server that exposes the Kubernetes API. It is effectively the front end for the control plane. The Kubernetes API is an HTTP API that can be accessed directly using REST calls. However, most of the time you will likely use the Kubernetes command-line interface *kubectl* or other command-line tools to interface with it.

etcd

etcd is the consistent and highly-available key-value store that Kubernetes uses to store all of the data that it uses to manage the cluster.

kube-scheduler

Component that watches for newly created Pods that have not been assigned a Node, and selects a Node for them to run on, taking into account factors such as resource requirements, software constraints, etc. We will go into a bit more detail about the scheduling of Pods later.

kube-controller-manager

Each controller, such as the *Node controller* is logically a separate process. However, they are all compiled and run as a single binary, the *kube-controller-manager*.

cloud-controller-manager

The *cloud-controller-manager* component is used to link a cluster into a cloud provider's API, providing easier integration with its features. For example, the *Service controller* can create, update and delete cloud-provided load balancers. This component is only present in cloud-based Kubernetes installations.

Node components

kubelet

The kubelet component runs on each Node and is responsible for ensuring the containers that are specified for a Pod are running and healthy.

kube-proxy

kube-proxy is a network proxy that helps implement *Services* by maintaining the network rules on the Nodes, allowing network communication to the Pods from inside or outside of the cluster.

Container runtime

This is the software that is responsible for running the actual containers on the Node. Kubernetes supports a few different container runtimes, with the most popular being Docker.

Kubernetes objects

Kubernetes objects are persistent entities in Kubernetes. They are a "record of intent" representing the *desired state* of your cluster, which Kubernetes will constantly work to achieve. There are various different types of objects, which can describe:

1. What containerised applications are running
2. The resources available to them
3. Policies about how they behave, such as restart policies, upgrades and fault-tolerance

When using the *kubectl* command-line interface you will most often provide the information about the object in a YAML file. We're going to have a look at an example file that declares a *Deployment* object, which is an object that can represent an application running on your cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

There are four required fields when defining a Kubernetes object:

- `apiVersion`: The version of the Kubernetes API you're using to create this object.
- `kind`: The type of object you want to create.
- `metadata`: Data that helps uniquely identify the object.
- `spec`: The state you desire for the object. The format for this field is dependent on the type of Kubernetes object.

In the example above, the `spec` for the Deployment contains the following:

- `template`: This is the template for the Pod used in the Deployment. In this case, it is a Pod that consists of one container (which uses the `ngingx:1.14.2` container image). Each instance of this Pod will have a *label* attached to it with a key-value pair of `app:nginx`.
- `replicas`: This defines how many instances of the Pod we require in our cluster, in this case 2.
- `selector`: This is using a *label selector* to define which Pods belong to this *ReplicaSet*, i.e. Kubernetes will look to make sure that 2 instances of Pods labelled with the key-value pair `app:nginx` are running on the cluster, which is the label we've given to the Pods we're creating.

When querying existing objects in Kubernetes, they will have another field, `status`, which describes the *current state* of the object. Kubernetes monitors this `status` and if there is a difference between it and the `spec` it will try and correct this. For example, if we created a Deployment spec that required three replicas of an application, then Kubernetes would read that spec and then start up three instances, updating the status to match the spec. If one of those instances should fail (for example if the Node went down) then Kubernetes would notice the status change and respond to the difference between the current status and the desired spec by making a correction, which in this case would be starting a replacement instance.

Workloads

A workload is an application running on Kubernetes, which may be a single Pod or several that work together. Kubernetes provides *workload resources*, which are a type of object that manages on your behalf the set of Pods that make up an application. These resources configure *controllers*, which ensure your Pods match the state you specified.

Some of the Kubernetes built-in workload resources are:

- `Deployment`: This resource is good for managing a stateless application workload, where any Pod in the Deployment is interchangeable and can be replaced if needed. Deployments use another type of workload resource, `ReplicaSet`, under-the-hood. The ReplicaSet ensures the specified number of identical Pods are running on the cluster. It is recommended that you use Deployments instead of interacting directly with a ReplicaSet, as Deployments are a higher-level concept that provide declarative updates to Pods along with other useful features. This is the object that we had a look at in the YAML file above.
- `StatefulSet`: If your workload is not stateless and it needs to record data persistently you can use a StatefulSet. We'll come back to how storage works within Kubernetes later in this module.
- `Job` and `CronJob`: These define tasks that run to completion and then stop, with Jobs representing one-off tasks and CronJobs representing tasks that recur based on a schedule.

Scheduling

We've had a look at a few different concepts in Kubernetes, but how do they tie together to ensure that the cluster is in the desired state?

Let’s have a look at what happens within Kubernetes in a few scenarios.

Creating a workload resource

First of all, let's have a look at what happens when we create a new Deployment workload resource. Imagine we have a Deployment that contains one Pod that we want three replicas of. We define that in a YAML file and upload it to Kubernetes using the `kubectl` CLI. A controller is now responsible for trying to match the status of the cluster to that desired spec:

1. The controller knows we have zero of the three Pods that are required, so it creates three Pods. These Pods do not yet have containers within them at this stage and they are not assigned to a Node.
2. The *kube-scheduler* component watches for newly created Pods that have no Node assigned, so will detect the three Pods that have just been created and add them to its queue. The scheduler is now responsible for finding the best Node for each Pod to run on. When you define a Pod you can also define rules about the Nodes that it can be placed on. The scheduler will filter all the Nodes by these requirements and place the Pod on one of those Nodes. If there are no suitable Nodes then the Pod will remain unscheduled until the scheduler is able to place it.

   Node requirements could specify the type and amount of a required resource, or only allow Nodes that aren't already hosting an instance of that Pod. As well as "hard" requirements you can also indicate that a rule is "soft"/"preference". If the scheduler cannot find a Node that satisfies that rule, the Pod will still be scheduled.

3. The *kubelet* component running on the Node notices that it now has a new Pod assigned to it. *kubelet* will create the containers for the Pod using the specified container runtime (e.g. Docker).

Eviction

We now have the three replica Pods of the Deployment running on different Nodes, but what happens if one of the Nodes starts running out of a resource? Conceptually it is up to Kubernetes to 'move' one of the Pods on that Node to a Node that has more available resources. However, in Kubernetes a Pod that is scheduled to a Node will run on a Node until it is stops or it is terminated, and a Pod instance cannot be "rescheduled" to a different Node, so what mechanism is used to 'move' a Pod?

1. On each Node the *kubelet* monitors the resources used by the Pods and if a resource is being starved then the *kubelet* fails a Pod, terminating all of its containers.
2. The controller watching our Deployment notices that there are now only two Pods running, and not the desired three. It now creates a new Pod.
3. Using the same mechanism as when initially creating the Pods, the *kube-scheduler* then assigns this new Pod instance to a Node, where the *kubelet* creates the containers for that Pod.

A similar mechanism is also used if a Node goes down. If a Node does not respond then all Pods on that Node are deleted. The controllers responsible for those Pods will then create new instances of the Pods to be scheduled, to move the cluster back towards the desired state.

Updating a workload resource

Let's now consider what happens when we want to update the container image we are using in our Deployment to a new version. To maintain availability Kubernetes uses rolling updates, where it does not remove old Pods until a sufficient number of new Pods have been created and it does not create new Pods until a sufficient number of old Pods have been killed. A Deployment can be configured so that only a certain amount of Pods can be down, to maintain availability, and only a certain amount of Pods can be created above the desired number, to minimise resource usage. By default Kubernetes ensures that a least

75% of the desired Pods are up and at most 125% of the desired number of Pods are available.

During a rolling update there is no guarantee about the ordering of removing and adding Pods. For example, if the Deployment has 10 replicas running and the configuration is set to allow 10 as a maximum number of Pods during an update, and 7 the minimum, then Kubernetes would start off by removing a Pod/Pods before creating the new Pods, so it would not break these constraints.

To update to a new container version, we would first update the yaml file for the Deployment to use the new container image. We would then use the *kubectl* CLI to upload the new version of this file to Kubernetes. Kubernetes, via a controller, would notice that the cluster's status does not match the desired spec anymore. Kubernetes would then alternate between scheduling new, updated Pods to new Nodes via the scheduler, and removing existing Pods. This would continue until all the instances of the Pods running were of the new version.

When removing a Pod during a rolling update, the Pod is not immediately terminated. First of all, no new requests are routed to the Pod, then after a delay, which is configurable, the Pod is removed. This is to ensure that the Pod has a chance to finish handling any requests that have been directed to it before it is removed.

Networking

Each Pod in the cluster is given its own IP address. Every container *within* a Pod shares the network namespace, including the IP address and network ports and can communicate with each other using `localhost`. Containers must use IP networking to communicate with containers running in different Pods. To help facilitate this inter-Pod communication, Kubernetes has the concept of a *Service*, as explained previously. Using *Services* means that we do not need to care about where the underlying Pods are as it is abstracted away for us by Kubernetes.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

The above example is a yaml definition for a Service called `my-service`. It uses a *label selector* to define that any Pod that has a label `app:MyApp` will be included in this Service.

If DNS has been enabled in your cluster then a DNS name will be created for the Service, allowing other Pods to look up the Service's IP address. The Service will then load balance those requests to the Pods that make up the Service.

Ingress

Pods can communicate within the cluster with the help of Services, but we often need to expose the applications running in our Pods to outside the cluster. This is done using *Ingress*, which exposes HTTP and HTTPS routes from outside the cluster to Services within the cluster, and controls the traffic routing via configured rules. The Ingress may be configured to terminate SSL, give Services externally-reachable URLs and load balance traffic.

An *Ingress controller* is responsible for fulfilling the *Ingress*, so one must be set up within the Kubernetes cluster for the *Ingress* resource to take effect. It is usually a load balancer, but may also be responsible for configuring the firewall or gateway for your cluster. When using a cloud-managed version of Kubernetes, the Ingress controller is often a cloud-based load balancer, such as Amazon Application Load Balancer (ALB) or Azure Application Gateway.

Storage

Pods in Kubernetes are ephemeral by nature. They may be destroyed on one Node and recreated on another Node to balance resources, or terminated during a Pod update. Most systems, however, need to store data and not have it disappear every time a Pod was terminated.

One perfectly valid strategy, that can often be simpler, is to store data outside of Kubernetes in a cloud-hosted database as a Service offering, such as Azure SQL Database or Amazon Aurora.

However, Kubernetes does give you tools to be able to run stateful applications in a cluster, such as databases.

Volumes

In Kubernetes a *volume* is a directory that can contain data, accessible to a Pod. There are many types of volumes in Kubernetes, some use storage on the Nodes themselves, others use cloud-provided storage. Some types of volumes are empty when created, others are pre-populated with data, such as configuration or a git repository, making them a good method for exposing data to a container in a Pod. Some volume types persist their data, even when the Pod using them is removed from the Node, others are removed along with the Pod.

When specifying a volume for a Pod you can specify where to mount those volumes into containers.

There are many different types of Kubernetes Volumes available, including:

1. *emptyDir* - This volume is created when a Pod is first assigned to a Node. As is suggested by the name, this volume is empty of data when created. Once the Pod is removed from the Node, the data within it is erased. As all containers within a Pod have access to the same volumes, this type can be used to transfer data between those containers. Also, this type is

useful for containers to store temporary data in, as the data will not be lost if a container needs to be restarted by the *kubelet*.

2. *gcePersistentDisk*, *awsElasticBlockStore* and *azureDiskVolume* - These volumes mount cloud provided storage (for Google Cloud, Amazon AWS and Microsoft Azure respectively) to your Pod. The data remains intact in the storage when the Pod is removed from the Node.
3. *gitRepo* - This type is an example of a volume that can provide data to the containers in your Pod. It mounts an empty directory and then clones a specified git repository into it, which your Pod then has access to.
4. *secret* - A secret volume is used to pass sensitive data, such as passwords, to Pods. We'll have a more in-depth look later about configuration and Secrets in Kubernetes.

When defining volumes, they are intrinsically linked to the Pod that you defined the volume for. However, we often want our data to be decoupled from the Pods that act upon it. This is where *PersistentVolumes* are useful.

PersistentVolumes

A *PersistentVolume (PV)* is piece of storage in the cluster. It is a cluster resource, in the same way that a Node is. The API object captures the details of the implementation of the storage, such as whether it is NFS, iSCSI or a cloud-provider-specific storage system.

```
kind: PersistentVolume
apiVersion: v1
metadata:
   name: pv0001
   labels:
      type: local
spec:
   capacity:
      storage: 10Gi
   accessModes:
      - ReadWriteOnce
      hostPath:
         path: "/tmp/data01"
```

The above specifies a PV with 10Gi of storage space.

While a PersistentVolume is a resource within a cluster, a *PersistentVolumeClaim (PVC)* is a request for storage by a user, which can be satisfied by a PersistentVolume. It is similar to a Pod; Pods consume Node resources, while PVCs consume PV resources. A PVC allows a user to consume abstract, persistent storage resource, without having to care about exactly how that storage is allocated within the cluster.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
   name: my-claim
spec:
   accessModes:
      - ReadWriteOnce
   resources:
      requests:
         storage: 4Gi
```

The above is a PVC for 4Gi of space.

When declaring a Pod (e.g. in a Deployment), we can now tell it to use the PersistentVolumeClaim we have created, which is allocated somewhere in our cluster. Every Pod that uses that PVC will get access to the same bit of storage, including if a Pod is deleted and then restarted on a different Node. The storage is not linked to the lifecycle of any particular Pod.

StatefulSet

StatefulSet is a *workload resource* like a *Deployment*. While Deployments are a good fit for *stateless* workloads, a *StatefulSet* is useful when running Pods that track state in some way.

StatefulSets provides guarantees about the ordering and uniqueness of the Pods that it contains. Each Pod within the set is created from the same spec. However they are not interchangeable, as each has a persistent identifier that is maintained across any rescheduling. This persistent Pod identifier makes it easier to match existing volumes to new Pods that have replaced failed Pods.

Using a combination of PersistentVolumes and StatefulSets allows stateful applications, such as databases, to be managed by Kubernetes.

Configuration

When writing an application it is a very good idea to separate the configuration out from the code, with the configuration being anything that is likely to vary between deploys. Values like the canonical hostname for the deploy, database hostname or credentials for external Services fall into this category. In Kubernetes we will need to pass these configuration values to applications that are running inside the Pods on our cluster.

ConfigMaps

A *ConfigMap* is a Kubernetes object used to store non-confidential data in key-values pairs. Pods can then consume these ConfigMaps as environment variables or as configuration files attached to the Pod in a *volume*.

Secrets

Sensitive and confidential configuration, such as passwords, OAuth tokens, API credentials and SSH keys, should be stored using Kubernetes *Secrets*. Secrets can be used with a Pod in three ways:

1. As files in a volume.
2. As container environment variables.
3. Pod spec files can reference Secrets to use when pulling container images from a registry. This allows you to pass in credentials for private container registries.

   Caution: By default, Kubernetes Secrets are stored as unencrypted strings, which can be retrieved as plain text by anyone with API access. It is strongly recommended to not use Kubernetes Secrets in this default state, especially for a production system. Kubernetes allows Encryption at Rest to be enabled for Secrets as well as RBAC (role-based access control) rules that restrict reading and writing Secrets.

Scaling

When working with Kubernetes there's two aspects of scaling to consider: the scaling of the application and the scaling of the cluster itself. Scaling applications (provided they are written in a stateless way that allows it) can be achieved by altering the number of Pod replicas. However, you can only increase the number of Pods running in your cluster if the cluster has enough room to do so. If the cluster is running out of resources due to the number of Pods running, then the cluster itself may have to be expanded, through adding Nodes, to then allow the number of Pods to be increased.

Pod scaling

When defining a Deployment you can specify how many replicas of the Pods are running in that Deployment. One way to alter the number of replicas running, either up or down, is to manually update the number of replicas in the Deployment. This, however, relies on someone manually monitoring the load across the Pods and then reacting to it, which is not sustainable, especially when you have dozens or even hundreds of Deployments running on a cluster.

Instead, Kubernetes provides a method for automatically scaling the Pods in a Deployment, ReplicaSet or StatefulSet, called the *Horizontal Pod Autoscaler*. It is implemented as a controller that periodically adjusts the number of replicas based on observed metrics. The controller's behaviour is defined by a Kubernetes API resource.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: my-deployment
spec:
  scaleTargetRef:
    kind: DeploymentConfig
    name: my-deployment
    apiVersion: apps/v1
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

In the above example we define that we want the CPU usage of our Pods to be, on average, 50%. We also define that we want at least 1 replica running, and no more than 10 replicas.

To achieve this, the controller will periodically read metrics (the default is every 15 seconds, but this can be specified) from Kubernetes' Metrics API. These metrics can be per-Pod resource metrics, such as memory usage or CPU utilisation (as specified above), or they can be custom metrics provided on a per-Pod basis. It's also possible to just use a single, external metric that isn't Pod-specific.

The Kubernetes Metrics API is usually provided by the *Metrics Server*, which needs to be deployed within the cluster.

The controller will then scale the number of replicas up or down to try and reach the user-supplied target value for the metrics. The controller operates on the ratio between the desired value and the current value:

```
desiredReplicas = ceil[currentReplicas *
( currentMetricValue / desiredMetricValue )]
```

The controller is aiming to get the ratio between the desired metric and the current metric to be sufficiently close to 1.0, within a globally configured tolerance that is defaulted to 0.1.

Scaling behaviours

Lets say we are running a web API in Kubernetes that has a consistently high amount of traffic, and the traffic suddenly drops to half the usual level for only one minute before returning to the usual level. In that minute of low level traffic, the *Horizontal Pod Autoscaler* would detect that the utilisation had dropped, and scale down the amount of Pods by half. When the traffic returns to normal a minute later, all of the Pods would be over-utilised, so the Autoscaler would double the Pods back to the original level. As it takes a small amount of time for Pods to start up, the API might become overloaded when it returns to the normal traffic amount.

It is for this reason that Kubernetes lets you define scaling behaviours, which control the rate of change of replicas while scaling. Kubernetes lets you set the behaviours for when you are scaling up or scaling down independently of one-another, as often you will want the behaviour to differ when Pods are being added, to how it behaves when Pods are being removed. For example, it might be prudent to have replicas scale down slowly to smooth over temporary dips in traffic, but to allow replicas to scale up quickly so the system remains responsive.

```
behavior:
  scaleDown:
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
    selectPolicy: Max
```

The above shows a *scaling policy* for scaling down replicas. The `periodSeconds` field indicates the amount of time for which the policy must hold true. The first policy allows at most 4 replicas to be scaled down in one minute. The second policy allows at most 10% of the current replicas to be scaled down in one minute.

The `selectPolicy` specifies which policy of the two would be used. As the value is `Max` this means it will choose the policy that allows the largest change in replicas.

```
scaleDown:
  stabilizationWindowSeconds: 300
```

Another option you can set is the *stabilization window*. The Autoscaler will take into account the metrics from across the specified time period when deciding how much it should scale. So in the above example, if the current metrics indicated that the Pods should be scaled down then the metrics for the past 5 minutes would be taken into account, and if any metrics in that time period meant the Pods should not be scaled down then they would not be. This helps smooth over fluctuations in metrics.

Cluster scaling

Scaling Pods to meet demand only works if there is capacity within your Kubernetes cluster to handle the number of required Pods. If you are hosting Kubernetes on your own server infrastructure then expanding the cluster involves purchasing extra machines and adding them manually into the cluster. However, if you are using Kubernetes in the cloud there are a couple approaches that allow for the automated scaling (both up and down) of the cluster to meet demand.

Cluster Autoscaler

Kubernetes provides a *Cluster Autoscaler*, which most cloud-provider implementations of Kubernetes, such as Amazon EKS, Azure AKS and Google Kubernetes Engine, support. The *Cluster Autoscaler* will increase the size of the cluster when there are Pods that fail to be scheduled on any current Nodes due to insufficient resources and adding a Node similar to the Nodes currently present would help. These new Nodes are VMs provided by the cloud platform. The Autoscaler decreases the size of the cluster when Nodes are consistently unneeded (that is, they have low utilisation and all of their important Pods can be moved elsewhere) for a significant amount of time.

Serverless

Some cloud-hosting companies now provide serverless container services, such as AWS Fargate or Azure Container Instances, that integrate with Kubernetes. Instead of directly managing the Nodes that make up a cluster, the service will run the Pods you provide in their own isolated environment. This service will manage the scaling of the cluster for you. It is similar to the concept of functions-as-a-service, where you provide the function and do not need to care about the exact machine it is run on. With serverless containers you provide the container, or the Pod, and the service manages the servers they are hosted on for you.

Chapter 4

Helm

Helm is a package manager that allows you to install packages into your Kubernetes cluster, in the same way you might use Homebrew, Chocolatey or Apt to install applications on your own computer.

Helm uses a packaging format called **charts**, which contains all the resource definitions needed to run an application, tool or service inside your cluster.

A single chart might be used to deploy a simple set of pods or something a lot more complex, such as multiple different Deployments or StatefulSets. It could even extend to deploying multiple Services that are designed to work together, such as a full web app stack with HTTP servers, databases, and caches. Charts can also reference and depend on other charts, so a web app could have a dependency on a database without needing to duplicate the entire configuration.

Charts can be stored in **chart repositories**.

Artifact Hub (artifacthub.io) is the main website used to search public repositories for charts.

Some repositories are public, allowing anyone to download charts and use the contained software. Other repositories are private and just used by one organisation to store the definitions of their internal applications.

Installing Helm

Helm is distributed as a client-side binary, so can be installed through either the official scripts or through a variety of package managers (e.g. Homebrew). Helm communicates with clusters through the same means as the `kubectl` CLI tool, so `kubectl` needs to be installed and configured to access the cluster too.

Installing a public chart

To install a public chart (e.g. Apache) on a Kubernetes cluster, first add the repository containing the chart:

```
$ helm repo add bitnami https://charts.bitnami.com/
bitnami
```

Then run the `helm install` command:

```
$ helm install bitnami/apache --generate-name
```

When a chart is installed, a new instance of it is deployed to Kubernetes. Each instance is known as a release. The `--generate-name` flag can be used to automatically generate a name for a release. However, it's also possible to manually set the release name, e.g. `my-release`:

```
$ helm install my-release bitnami/apache
```

The `helm list` command can be used to view information about deployed releases. The `helm delete` command can be used to remove and uninstall a release from a cluster.

Overriding parameters

The behaviour of a chart can often be modified using parameters (e.g. `imagePullPolicy`). Charts typically have default values for parameters, which can then be overridden in the `helm install` command:

```
$ helm install my-release --set imagePullPolicy=Always
bitnami/apache
```

Pods host one or more containers, and each container is based on an image. When Kubernetes creates a Pod, it uses the Pod's `imagePullPolicy` to determine whether to pull each container's image from the registry, or use cached images instead.

It's also possible to override parameters by creating a YAML file and passing it to the `helm install` command:

```
# values.yaml
image:
  pullPolicy: Always
```

```
$ helm install my-release -f values.yaml bitnami/apache
```

Upgrading releases

An existing release can be upgraded to a new version by using the `helm upgrade` command. An upgrade can change the version of the chart, change the values passed to the chart, or both. Helm keeps a history of the different revisions of each release. This allows you to rollback to a previous revision if necessary.

In the following example, `my-release`'s current chart is replaced with the latest Apache chart in the repository. `values.yaml` is used to override the chart's parameters: the new Apache deployment will serve a static site cloned from a git repository.

```
# values.yaml
cloneHtdocsFromGit:
  enabled: true
  repository: https://github.com/mdn/beginner-html-site-styled.git
  branch: master
```

```
$ helm upgrade my-release -f values.yaml bitnami/apache
```

To see the history of a release, use `helm history`.

```
$ helm history my-release
REVISION        UPDATED                         STATUS          CHART           APP VERSION     DESCRIPTION
1               Wed May 26 17:43:11 2021        superseded      apache-8.5.4    2.4.46          Install complete
2               Wed May 26 17:44:04 2021        deployed        apache-8.5.4    2.4.46          Upgrade complete
```

To revert to a previous revision of a release, use `helm rollback`.

```
$ helm rollback my-release 1
Rollback was a success! Happy Helming!

$ helm history my-release
REVISION        UPDATED                         STATUS          CHART           APP VERSION     DESCRIPTION
1               Wed May 26 17:43:11 2021        superseded      apache-8.5.4    2.4.46          Install complete
2               Wed May 26 17:44:04 2021        superseded      apache-8.5.4    2.4.46          Upgrade complete
3               Wed May 26 17:46:03 2021        deployed        apache-8.5.4    2.4.46          Rollback to 1
```

In a CI setting, you may often want to upgrade an existing release or create it if it doesn't already exist. This can be achieved by using the `--install` flag:

```
$ helm upgrade my-release --install --set metrics.enabled=true bitnami/apache
```

In this example, the `metrics.enabled` parameter is set, so a Prometheus exporter will be included in the Apache deployment. If `my-release` already exists, then it will be upgraded to include the exporter. If no such release exists, then it will be created (along with the exporter).

Prometheus is a service that can be used to monitor metrics. A Prometheus exporter can be used to import metrics into Prometheus.

Chart structure

Charts are created as a directory of files in a structure specified by Helm, and can be packaged into versioned archives for distribution through repositories.

Helm documentation:
https://helm.sh/docs/topics/charts/

The top level directory of a chart has the same name as the chart (without any versioning). For example, a simple `devops` chart could have this structure:

```
devops/
  Chart.yaml
  values.yaml
  templates/
    deployment.yaml
    service.yaml
```

Chart.yaml

The `Chart.yaml` file contains metadata for the chart. The most important (and only mandatory) fields in this file are:

1. `apiVersion`
2. This should be set to `v2` for new Helm 3 charts, or `v1` for older/backwards compatible charts.
3. `name`
4. The name of the chart, e.g. `devops`.
5. `version`
6. A version number for the chart

A minimum viable `Chart.yaml` file might look like this:

```
# Chart.yaml
apiVersion: v2
name: myChart
version: 1.2.13
```

The version number specified here is used by Helm in several ways, so it is important to correctly keep it up to date. For example, it will automatically be included in the name of the chart archive produced by `helm package`.

Helm chart versions follow the Semantic Versioning 2.0.0 (SemVer 2) format. Non-semantic version numbers are specifically disallowed.

Templates

The `templates` directory consists of YAML templates that, when combined with the values from `values.yaml`, will generate a valid Kubernetes manifest file.

Helm documentation: https://helm.sh/docs/chart_template_guide/

The `values.yaml` and `templates/deployment.yaml` files below can be combined to produce a Deployment manifest. When the `devops` chart containing these files is installed on a cluster, a Kubernetes Deployment object is created using the manifest.

A Kubernetes manifest is a file that defines a Kubernetes object (e.g. a Deployment). Kubernetes manifests can be written in either JSON or YAML, but YAML is generally preferred for readability.

```
# values.yaml
nginx:
  version: 1.18.0
external:
  expose: true
  port: 8080

# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-nginx-deployment
spec:
  selector:
    matchLabels:
      app: {{ .Release.Name }}-nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}-nginx
    spec:
      containers:
      - name: nginx
        image: nginx:{{ .Values.nginx.version }}
        ports:
        - containerPort: 80
```

Values from `values.yaml` are accessed like `{{ .Values.nginx.version }}`. Helm also provides many built in values. For example, `{{ Release.Name }}` will be replaced with the name of the Helm release. This system allows the commonisation of shared values between template files in the chart.

The `values.yaml` file included in the chart sets default values, and end users can selectively override them at deployment. For example, to release the chart above with a different nginx version, a user could run:

```
$ helm install devops --set nginx.version=1.19.9 ./devops
```

Further templating

As well as directly placing values into templates, the Helm template language contains various features to transform the supplied data into different formats. For example, the `quote` function will wrap a string with quotes, and the `default` function replaces a non-existent value with a given default.

```
# values.yaml
serviceName: "my-service"

# templates/example.yaml
port: {{ default 8080 .Values.port }} # a "port" value is not provided, so
this renders as 8080
serviceName: {{ quote .Values.serviceName }} # renders as "my-service"
```

Functions can also be composed in a UNIX-style pipeline format:

```
# Equivalent to {{ default 8080 .Values.port }}
port: { { .Values.port | default 8080 } }

# Equivalent to {{ quote (default "my-devops-service" .Values.serviceName) }}
serviceName: { { .Values.serviceName | default "my-devops-service" | quote } }
```

This inverted, functional style is common in the Helm documentation as well as many real-world chart definitions.

As well as populating values, the Helm template language can be used to change the structure of manifests. Conditional blocks created with `if/else` allow us to selectively include blocks of text, and `range` provides looping capabilities.

For example, we could define the final template from the `devops` chart above as:

```
# templates/service.yaml
{{ if .Values.external.expose }}
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-service
spec:
  selector:
    app: {{ .Release.Name }}-nginx
  ports:
    - name: {{ .Values.external.portName | default "http" | quote }}
      protocol: TCP
      port: {{ .Values.external.port }}
      targetPort: 80
{{ end }}
```

If `external.expose` is set to `True`, then the block will render and Helm will deploy a Service to the cluster. Otherwise, the template will render as an empty file and no service will be deployed.

See the Helm documentation for a full list of functions and flow control structures.

Subcharts

A chart directory can contain a `charts` subdirectory, which contains *subcharts* - dependencies of the top level chart. These are standalone Helm charts which can have their values overridden from the parent chart.

Helm documentation: https://helm.sh/docs/chart_template_guide/subcharts_and_globals/

Suppose the `devops` chart has an `autoscaler` subchart, which provides a `HorizontalPodAutoscaler` for the application. The file structure would look like this:

A subchart cannot override a value from its parent.

```
devops/
  Chart.yaml
  values.yaml
  charts/
    autoscaler/
      Chart.yaml
      values.yaml
      templates/
        autoscaler.yaml
  templates/
    deployment.yaml
    service.yaml
```

Suppose that the `autoscaler` subchart is defined as follows:

```
# autoscaler/Chart.yaml
apiVersion: v2
name: autoscaler
version: 0.1.0

# autoscaler/values.yaml
minReplicas: 1
maxReplicas: 10
targetCpu: 70
```

Charts have a recursive structure; each subchart has the same structure as the parent chart.

```
# autoscaler/templates/autoscaler.yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ .Release.Name }}-autoscaler
spec:
  scaleTargetRef:
    kind: Deployment
    name: {{ .Release.Name }}-{{ .Values.scaledDeployment }}
    apiVersion: apps/v1
  minReplicas: {{ .Values.minReplicas }}
  maxReplicas: {{ .Values.maxReplicas }}
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: {{ .Values.targetCpu }}
```

The subchart can be configured from the `values.yaml` file of the parent chart. All values in the section matching the name of the subchart will be passed down. For example, we could modify `devops/values.yaml` as follows:

```
nginx:
  version: 1.18.0

external:
  expose: true
  port: 8080

autoscaler:
  maxReplicas: 20
  scaledDeployment: nginx-deployment
```

Note that the `autoscaler` section can both set required values (`scaledDeployment`) and override values with an existing default (`maxReplicas`).

By default, Helm doesn't distinguish between required and optional values. Values used in templates but not provided in `values.yaml` or as an override will render as the empty string. You can use the `required` Helm function to throw an explicit error if a value is not provided.

External dependencies

Subcharts can be used to break up a large parent chart into logical chunks. They can also be used to extract common functionality so that it can be shared. For this to work, the subchart needs to be stored separately to the parent. This can be achieved by adding a `dependencies` section to the parent `Chart.yaml`:

```
apiVersion: v2
name: devops
version: 1.0.0
dependencies:
  - name: autoscaler
    version: 1.1.0
    repository: https://example.com/charts
```

The dependency's repository can either be a Helm repository or a local path (e.g. `repository: file://../dependencies/autoscaler`).

Once all external subcharts are listed in `Chart.yaml`, they can be synced into the parent chart's `charts` folder by running:

```
$ helm dependency update
```

Creating a chart

Creating a Helm chart for your application has a few benefits over deploying the manifests directly:

- The templating system allows values to be commonised between manifest files easily, as well as providing various sophisticated YAML generation capabilities.
- Your application can easily integrate with the huge library of already existing Helm charts by including them as subcharts.
- Other applications can easily integrate your application by depending on it as a subchart.

To quickly scaffold a new Helm chart, run `helm create chart-name`. This will create the necessary file structure for you, as well as providing various example template files (which you can safely delete).

Chart repositories

Helm documentation:
https://helm.sh/docs/topics/chart_repository/

Pulling from a chart repository

Earlier, we installed the `bitnami/apache` chart from a public repository. To download a chart locally first (e.g. for inspection or verification), use the `helm pull` command:

```
$ helm pull bitnami/apache --untar # Download and
extract the bitnami/apache chart
```

`helm install` deploys a chart to a Kubernetes cluster without downloading a local copy.
`helm pull` downloads a local copy of a chart, but doesn't deploy it to a cluster.

Pushing to a chart repository

A repository is a web server containing an `index.yaml` file and a collection of chart archives. Pushing your chart to a repository allows it to be easily shared.

The `index.yaml` file contains information about each chart in the repository.

Before we can upload our `devops` chart to a repository, it has to be packaged for distribution.

```
# Create the chart archive devops-0.1.0.tgz
$ helm package devops
```

Next, the `index.yaml` file for the repository must be updated to include the `devops` chart.

```
$ mkdir devops-charts
$ mv devops-0.1.0.tgz devops-charts

# Generate an index.yaml file containing the existing
charts, as well as the newly created chart
$ helm repo index devops-charts --url https://my-helm-
repository.net
```

Finally, the `devops-0.1.0.tgz` archive and updated `index.yaml` file can be uploaded to the repository.

Collaborative chart repositories (e.g. k8s@home) generally have extended upload processes that are designed to handle large numbers of contributors and submissions. You can view the guidelines for contributing to k8s@home here.

Chapter 5

Security in Kubernetes

When running and maintaining Kubernetes clusters, security is a key concern. As Kubernetes manages a large number of components and a great deal of complexity, there are several areas to be conscious of when securing and managing workloads.

Secrets and credentials

Just like we've seen in previous modules, it is best practice in Kubernetes to separate *configuration* from *code*. This allows one container image to be built and then deployed across environments or with different configurations. However, splitting code and configuration requires us to have a secure way to manage that configuration and pass it in to the application at run time. The simplest way to pass configuration is to directly attach environment variables to a Deployment, for example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: environment-variables
spec:
  selector:
    matchLabels:
      app:  environment-variables
  replicas: 3
  template:
    metadata:
      labels:
        app:  environment-variables
    spec:
      containers:
      - name: environment-variables
        image: gcr...
        env:
        - name: SERVER_PORT
          value: 8080
        - name: SUPER_SECRET_PASSWORD
          value: "ImSuperSecret1$"
```

For lots of configuration this is both acceptable and clear. However, for sensitive values like database credentials or API keys it is worth using the Kubernetes built in primitive of `Secrets`.

Secrets have a separate lifecycle from Deployments, allowing keys to be rotated without changing Deployments. They also signal intent clearly, for example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: environment-variables
spec:
  selector:
    matchLabels:
      app:  environment-variables
  replicas: 3
  template:
    metadata:
      labels:
        app:  environment-variables
    spec:
      containers:
      - name: environment-variables
        image: gcr...
        env:
        - name: SUPER_SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret-name
              key: apiKey
---
apiVersion: v1
kind: Secret
metadata:
  name: secret-name
type: Opaque
data:
  apiKey: c20vjcmv
```

It's worth noting that while "Secrets" sound secure, they're not *that* secure on their own. Secrets are effectively base64 encoded to slightly obscure their values, but they're by default **not** encrypted at rest.

They're also available to anyone who can access Deployments, which is dependent on roles, explored further below.

The Twelve-Factor App

The twelve-factor app (12FA) is a methodology for building web apps that incorporates several DevOps best practices. It's structured as a list of factors, each aiming to improve portability, reliability, and scalability. 12FA is generalised, and each factor can be applied to apps written in any programming language.

Some of the characteristics of 12FA apps are:

- They have setup processes that are declarative and automated, allowing new developers to ramp up quickly.
- They aren't dependent on a specific operating system, increasing portability between environments.
- They are deployable on cloud platforms, removing the need to run and maintain servers.
- Their development and production environments are always closely in sync, making deployments safer.
- They can easily scale up, enabling them to meet demand with minimal developer input.

You can find a full list of the factors, along with explanations, here (https://12factor.net/).

There are other ways to inject secrets, such as cloud-provided secret storage (e.g. Azure KeyVault) or other specific tooling (e.g. HashiCorp Vault). Most external secret providers integrate through Kubernetes Container Storage Interface Secret Stores, which act as a bridge between external data stores and APIs that load Kubernetes secrets. This can provide better guarantees on the security of secrets, but adds complexity to the management of the cluster.

Securing the cluster

Outside of credentials, there are a variety of attack surfaces and risks within an environment as complex as Kubernetes.

A commonly used model for security within cloud-based services is the 4Cs, identifying four layers of security, and where threats can come from. The four Cs are Cloud, Cluster, Container and Code.

Cloud

Clouds are often assumed to be relatively secure, and cloud providers spend significant amounts of money and effort on securing their infrastructure. However, it's worth validating a few key areas, and ensuring that on self hosted clusters the following are taken into account.

- Access to the Control Plane/API Server should be restricted. The API server shouldn't be reachable over the public internet, or access should be restricted to a specific set of IP addresses required to administer the cluster. Similarly, if possible, Nodes shouldn't be directly accessible over the internet. Instead, they should be placed behind the cloud provider's load balancers. Restricting IP addresses and ports to those required by specific services is also recommended.
- Etcd, which is used as the underlying data store (including Secrets) for the cluster, should be secured and encrypted at rest. Access to etcd should be limited to the control plane only.
- When running Kubernetes within a cloud provider (e.g. Azure AKS or AWS EKS), various services that require access to the cloud provider's API are used (e.g. load balancers and VM management). This should be on the *principle of least privilege*, only allowing access to the resources that are actually required.

Cloud providers also often make specific recommendations for securing workloads, which should be followed.

Cluster

After the cloud and underlying infrastructure are secured, the next area of focus is on the Kubernetes cluster itself. Within the cluster, there are two areas of concern: the components of the cluster itself (those provided by Kubernetes) and the components running in the cluster (the workloads you are running).

Role-based Access Control

Role-based access control (RBAC) is a common method of managing access to computer and network resources, based on roles.

Kubernetes structures RBAC around four objects, `Role`, `ClusterRole`, `RoleBinding` and `ClusterRoleBinding`. These can be managed through the API or kubectl, in the same way as other Kubernetes objects. Role and ClusterRole both contain *additive* permissions. The difference is that Roles apply within a namespace and ClusterRoles apply outside a namespace. If you want to define roles within namespaces, define them with Roles. For cluster-level permissions or across namespaces, use ClusterRoles.

Additive permissions means that rules can only increase what a user has permission to do, i.e. there are no "deny" rules that could override permissions set at a higher level.

RoleBindings and ClusterRoleBindings connect (or *bind*) users or other **subjects** (e.g. groups, service accounts) to Roles and ClusterRoles. Once connected (or *bound*) to a Role or ClusterRole the subject will have the permissions granted by that object. RoleBindings grant access within a specific namespace, and ClusterRoleBindings grant access cluster-wide.

A RoleBinding can bind either a Role *or* a ClusterRole. A ClusterRole can only bind a ClusterRole.

Let’s look at an example. The first part of the YAML below creates a ClusterRole called `secret-read-only` that grants permissions to `get`, `watch`, and `list` secrets. The second part creates a RoleBinding called `read-secrets` that binds a subject, the user `alice`, and the `secret-read-only` ClusterRole defined earlier. The RoleBinding is scoped to the `development` namespace.

The overall effect is to grant `alice` permission to read secrets, but only within the `development` namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secret-read-only
rules:
- apiGroups: [""] # "" defines the core API group
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-secrets
  namespace: development
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-read-only
  apiGroup: rbac.authorization.k8s.io
```

Roles and RoleBindings should be fine-grained and follow the *principle of least privilege*, granting only the minimal set of permissions that the subject needs.

Kubernetes has a number of built-in roles, covering specific component needs and user roles. For example, `system:kube-scheduler` allows access to resources needed by the scheduler, while `cluster-admin` allows full access to every resource.

Network Policies

Network policies can be used to prevent access between pods and other networked components, through an *allow list* of pods and namespaces, and an IP blocking process.

A network interface is the software or hardware that connects a computer to a network. The Container Network Interface (CNI) defines a common network interface for Linux containers. There are several CNI implementations that can be used as a networking layer in Kubernetes (e.g. Calico). However, it's important to carefully choose the networking layer, as some CNI implementations don't support network policies.

The CNI specification is language and environment agnostic, making it highly adaptable.

Calico implements all the required features for Network Policies.

If multiple pods have a network policy applied, both the egress policy on the source and the ingress policy on the destination need to allow the traffic or it will be blocked.

In the example below, the network policy isolates pods matching `role=db`, in the `default` namespace for `Ingress` and `Egress` traffic. It allows traffic from any Pod in the `default` namespace with `role=frontend`, or IP addresses in the range `10.0.0.0/24`. Pods within the `default` Namespace matching `role=db` are allowed to connect on port 5978 to any IP in the range `10.0.0.0/24`.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.0.0.0/24
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Container

Containerisation of code provides a powerful way to manage and deploy application artefacts, but it adds additional attack surfaces. There are various practices which should be applied to reduce the risk of containers, including scanning containers, signing images, running with least privileges and reviewing the container runtime itself.

Container Scanning

As part of the build process for container applications, an automated container scan should be run using e.g. *Trivy* or *Clair*.

This provides a point-in-time view of any known security vulnerabilities within the base image, and any application level dependencies included that the scanning tool can use.

Scanning as part of a build process incorporates feedback early. However, applications may not be released regularly, so it is worth running scans on images used in production to understand if any new vulnerabilities might have been revealed in the lifetime of the service.

Image signing

Public container registries that are often used for convenience have the possibility to introduce vulnerabilities into a cluster. For example, a malicious update to a public container image could be published and then automatically used by the cluster.

Images can be signed using tools such as Docker Notary, and then verified when run within the cluster.

Minimal base images

Docker containers can contain large attack surfaces from large base images. For example `node`'s default image is built on Debian Stretch, a full distribution with a large number of tools and utilities that are rarely required for applications. Base images, such as those built on `alpine` or Google's Distroless images, contain fewer utilities and tools so have a reduced attack surface. Compiled languages such as Go can use scratch images that contain only the application binary.

Trivy and Clair are Static Analysis tools used to identify container vulnerabilities, including application and dependency vulnerabilities.

Code

Code security is a large and complex area, and is covered within Module 10.

Other/best practice

Kubernetes is a constantly evolving platform, with regular patching of core components and changes to the API, which should be applied as early as possible. Most cloud providers provide updates from upstream Kubernetes within short time frames, particularly for security vulnerabilities.

There are various extensions, built on top of Kubernetes, that can provide additional security features. For example, service meshes (e.g. Istio) allow precise control over networks. If an application handles sensitive data or has strict security requirements, these extensions may be worth investing in.

Further Reading

- https://kubernetes.io/docs/concepts/security/overview/
- https://www.whitesourcesoftware.com/resources/blog/kubernetes-security
- https://cloudian.com/guides/kubernetes-storage/kubernetes-security-the-complete-guide/
- https://snyk.io/blog/10-docker-image-security-best-practices
- https://docs.microsoft.com/en-us/azure/key-vault/general/key-vault-integrate-kubernetes
- https://learn.hashicorp.com/tutorials/vault/kubernetes-sidecar