Docker y Kubernetes
Con ejemplos en .Net

Autor:

Kevin Cárdenas.

2023

Índice

1. Docker

Una de las ventajas principales de Docker es su portabilidad. Los contenedores de Docker se pueden ejecutar en cualquier entorno donde Docker esté instalado, lo que facilita la creación y despliegue de aplicaciones de manera consistente en diferentes sistemas.

¿Qué es Docker?

Docker nos permite integrar todos los componentes de nuestra aplicación en un contenedor con las dependencias y librerías necesarias para que funcione en cualquier entorno. Una de las ventajas para los desarrolladores es la posibilidad de ejecutar diferentes aplicaciones sin que interfieran unas con otras ya que cada aplicación está dentro de su propio contenedor. Los contenedores Docker no tienen su propio sistema operativo sino que utilizan el del host. Arrancar un contenedor sólo lleva unos segundos.

Dockerfile

Un Dockerfile es un documento de texto que contiene una secuencia de comandos o instrucciones que se ejecutan para construir una imagen Docker. La estructura del Dockerfile típicamente comienza con el uso de una imagen base, haciendo algunas configuraciones a la imagen, y estableciendo los comandos necesarios para empaquetar o ejecutar la aplicación. Para construir una imagen utilizando un Dockerfile, utilizamos el comando `docker build` y especificamos el nombre y la etiqueta del Dockerfile.

¿Qué es una Imagen?

Una imagen de Docker es una plantilla de solo lectura que contiene todo lo necesario para ejecutar una aplicación, incluyendo el código, las bibliotecas, las dependencias y las variables de entorno. Las imágenes se utilizan como base para crear contenedores.

Contenedor

Un contenedor de Docker es una instancia en ejecución de una imagen. Es un entorno aislado y ligero que contiene todo lo necesario para que una aplicación se ejecute de manera independiente. Los contenedores son portátiles y se pueden ejecutar en cualquier entorno que tenga Docker instalado.

Volumen

Un volumen en Docker es un mecanismo para persistir y compartir datos entre contenedores y el host. Los volúmenes son directorios montados en el sistema de archivos del host, pero son administrados por Docker y pueden ser utilizados por uno o varios contenedores.

Comandos de Docker

Veamos algunos comandos comunes de Docker:

- `docker pull <image>`: Descarga una imagen desde un registro de Docker.
- `docker build -t <image:tag> <path>`: Construye una imagen a partir de un Dockerfile.
- `docker run -d --name <container> <image:tag>`: Crea y ejecuta un contenedor a partir de una imagen.
- `docker stop <container>`: Detiene un contenedor en ejecución.
- `docker start <container>`: Inicia un contenedor detenido previamente.
- `docker restart <container>`: Reinicia un contenedor.
- `docker rm <container>`: Elimina un contenedor.
- `docker rmi <image:tag>`: Elimina una imagen.
- `docker images`: Muestra las imágenes disponibles en el sistema.
- `docker exec -it <container> <command>`: Ejecuta un comando en un contenedor en ejecución.

Comando	Descripción
docker ps	Muestra una lista de los contenedores en ejecución.
docker ps -a	Muestra todos los contenedores en el sistema, incluso los que no están en ejecucuón.
docker logs «container_name»	Muestra los registros de un contenedor específico.
docker inspect «container_name»	Muestra información detallada sobre un contenedor.
docker network	Permite administrar las redes de Docker.
docker volume	Permite administrar los volúmenes de Docker.
docker-compose up	Levanta todos los servicios definidos en un archivo docker-compose.yml.
docker-compose down	Detiene y elimina todos los servicios definidos en un archivo docker-compose.yml.

SQL SERVER

SQL Server es un sistema de gestión de bases de datos relacional desarrollado por Microsoft. Permite almacenar, consultar y administrar grandes volúmenes de datos de manera eficiente. Para conectar, ver y administrar una instancia de SQL Server, podemos utilizar Azure Data Studio, una herramienta multiplataforma y gratuita proporcionada por Microsoft. Para ejecutar una instancia de SQL Server en un contenedor Docker, podemos utilizar el siguiente comando:

```
docker run -e "ACCEPT_EULA=Y" -e "MSSQL_SA_PASSWORD=@udemy123" -p 1433:1433 -d mcr.microsoft.com/mssql/server:2022-latest
```

Donde:

- `-e "ACCEPT_EULA=Y"`: Establece la variable de entorno `ACCEPT_EULA` en `Y` para aceptar el acuerdo de licencia de SQL Server.
- `-e "MSSQL_SA_PASSWORD=@udemy123"`: Establece la variable de entorno `MSSQL_SA_PASSWORD` con la contraseña que se utilizará para el usuario SA (administrador de SQL Server).
- `-p 1433:1433`: Mapea el puerto `1433` del contenedor al puerto `1433` del host, lo que permite la comunicación con SQL Server a través de ese puerto.
- `-d`: Ejecuta el contenedor en segundo plano (modo `detached`).
- `mcr.microsoft.com/mssql/server:2022-latest`: Especifica la imagen de SQL Server utilizada para crear el contenedor. En este caso, se utiliza la imagen `mcr.microsoft.com/mssql/server:2022-latest`, que corresponde a la última versión disponible en el momento de escribir esto (2022).

Una vez que el contenedor esté en ejecución, podremos utilizar Azure Data Studio para conectarnos a la instancia de SQL Server, ya sea en el mismo equipo o en un equipo remoto. Azure Data Studio nos permite realizar tareas como crear y administrar bases de datos, ejecutar consultas, administrar usuarios y permisos, y muchas otras actividades relacionadas con SQL Server.

MONGO DB

MongoDB es una base de datos NoSQL de alto rendimiento y orientada a documentos. Para utilizar MongoDB, se puede utilizar la imagen de Docker proporcionada por MongoDB llamada mongo:latest. Una imagen de Docker es un paquete ligero y autónomo que contiene todo lo necesario para ejecutar una pieza de software, incluido el sistema operativo, las bibliotecas y las dependencias.

La imagen mongo:latest se puede utilizar para ejecutar un contenedor de MongoDB de la siguiente manera:

```
docker run -d -e MONGO_INITDB_ROOT_USERNAME=admin -e MONGO_INITDB_ROOT_PASSWORD=@udemy123 -p 27017:27017 mongo:latest
```

En este comando, estamos ejecutando un contenedor en segundo plano (-d) utilizando la imagen mongo:latest. También hemos especificado las variables de entorno `-e MONGO_INITDB_ROOT_USERNAME y MONGO_INITDB_ROOT_PASSWORD` para establecer el nombre de usuario y la contraseña del usuario root de MongoDB. Además, hemos asignado el puerto del contenedor 27017 al puerto del host 27017 mediante la opción -p. Esto permite que los clientes se conecten al contenedor MongoDB desde el host utilizando el puerto 27017. Una vez que el contenedor se está ejecutando, se puede utilizar una herramienta de administración de MongoDB, como Studio 3T, para conectarse, ver y administrar la base de datos. Studio 3T es una herramienta popular y fácil de usar que proporciona una interfaz gráfica para interactuar con MongoDB y realizar tareas de administración de bases de datos de manera eficiente.

POSTGRESQL

PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto y potente. Para utilizar PostgreSQL, podemos aprovechar la imagen de Docker proporcionada por la comunidad de PostgreSQL llamada postgres:latest. Una imagen de Docker es un paquete autónomo que incluye todo lo necesario para ejecutar una aplicación, incluido el sistema operativo, las bibliotecas y las dependencias. La imagen postgres:latest nos permite ejecutar un contenedor de PostgreSQL de manera sencilla. Podemos utilizar el siguiente comando de Docker para ejecutar un contenedor de PostgreSQL:

```
docker run -d -e POSTGRES_USER=admin -e POSTGRES_PASSWORD=password -p 5432:5432 postgres:latest
```

En este comando, estamos ejecutando un contenedor en segundo plano (-d) utilizando la imagen postgres:latest. También hemos especificado las variables de entorno `-e POSTGRES_USER y POSTGRES_PASSWORD` para establecer el nombre de usuario y la contraseña del usuario administrador de PostgreSQL. Además, hemos asignado el puerto del contenedor 5432 al puerto del host 5432 mediante la opción -p. Esto permite que las aplicaciones y los clientes se conecten al servidor PostgreSQL dentro del contenedor desde el host utilizando el puerto 5432. Una vez que el contenedor se esté ejecutando, podemos utilizar herramientas de administración de bases de datos como pgAdmin o DBeaver para conectarnos, ver y administrar la base de datos PostgreSQL. Estas herramientas proporcionan interfaces gráficas intuitivas que facilitan la administración de la base de datos y la ejecución de consultas. En resumen, utilizando la imagen de Docker postgres:latest, podemos ejecutar un contenedor de PostgreSQL y utilizar herramientas como pgAdmin o DBeaver para administrar y trabajar con la base de datos PostgreSQL de manera eficiente y conveniente.

Conectarse desde una aplicación .Net a Un contenedor postgres

Despues de tener un contenedor de una base de dartos corriendo y mapeado en un puerto puedes conectarte a él desde una aplicación como si tuvieras instalado el servidor de base de datos en cuestion. Esta es una gran herramienta que nos otorga Docker. Veamos un ejemplo.

Para conectarse a un contenedor de PostgreSQL desde una aplicación .NET, sigue los siguientes pasos:

1. Abre un contenedor de PostgreSQL utilizando el siguiente comando:

```
docker run -d -p 5432:5432 -e POSTGRES_USER=<username> -e POSTGRES_PASSWORD=<password> postgres
```

2. Debemos asegurarnos de tener instalado el paquete NuGet Npgsql en el proyecto .NET. Este paquete proporciona una biblioteca para interactuar con PostgreSQL desde aplicaciones .NET.

3. En el código de la aplicación, debemos importar el espacio de nombres Npgsql y crea una cadena de conexión para establecer la conexión con el contenedor de PostgreSQL. La cadena de conexión debe incluir la dirección IP o el nombre de host del contenedor (en este caso, localhost), el puerto mapeado (por ejemplo, 5432), el nombre de la base de datos y las credenciales de autenticación. Por último Establece la conexión con PostgreSQL utilizando la cadena de conexión.

Veamos un ejemplo de código para establecer la conexión:

```
using Npgsql;

string connectionString = "Host=localhost;Port=5432;Database=<database_name>;Username=<username>;Password=<password>";

using (NpgsqlConnection connection = new NpgsqlConnection(connectionString))
{
    connection.Open();

    // Realizar operaciones en la base de datos

    connection.Close();
}
```

1.1. Dockerfile

Un Dockerfile es un archivo de texto que contiene instrucciones para construir una imagen de Docker de manera automatizada. El Dockerfile define qué software se incluirá en la imagen, cómo se configurará y cómo se ejecutará la aplicación dentro del contenedor.

Comandos de Dockerfile

Veamos un ejemplo del Dockerfile para una aplicación de .NET:

```
# Build Stage
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /app
COPY . ./

RUN dotnet restore
RUN dotnet publish -c release -o /app

# Runtime Stage
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build /app .

EXPOSE 80

ENTRYPOINT ["dotnet", "app.dll"]
```

Ahora para crear la imagen y constuir un contenedor ejecutamos:

```
# Construir la imagen
docker build -t myapp .
```

```
# Ejecutar un contenedor a partir de la imagen
docker run -d -p 3000:3000 myapp
```

El primer comando (`docker build`) se utiliza para construir la imagen utilizando el Dockerfile presente en el directorio actual (`.`), y se le asigna el nombre `myapp`. El segundo comando (`docker run`) crea y ejecuta un contenedor a partir de la imagen `myapp` , y realiza el mapeo del puerto 3000 del contenedor al puerto 3000 del host (`-p 3000:3000`). Esto permite acceder a la aplicación desde el navegador utilizando `localhost:3000`. Estos te permitirá construir y ejecutar el contenedor Docker para tu aplicación Node.js de forma efectiva. Veamos algunos comandos de Docker file.

Comando	**Descripción**
FROM	Especifica la imagen base a partir de la cual se construirá la nueva imagen. Es Obligatorio, pues define la imagen base
RUN	Ejecuta un comando en el sistema de archivos de la imagen durante el proceso de construcción.
COPY	Copia archivos o directorios desde el sistema de archivos local al sistema de archivos de la imagen.
ADD	Copia archivos o directorios desde el sistema de archivos local a la imagen, y también admite la descarga y extracción de archivos remotos.
WORKDIR	Establece el directorio de trabajo para cualquier comando **RUN**, **CMD**, **ENTRYPOINT**, **COPY** o **ADD** que se ejecute en la imagen.
ENV	Establece variables de entorno en la imagen.
EXPOSE	Indica los puertos en los que el contenedor escuchará en tiempo de ejecución.
CMD	Proporciona comandos predeterminados para ejecutar cuando se inicie un contenedor a partir de la imagen.
ENTRYPOINT	Especifica un comando que se ejecutará siempre cuando se inicie un contenedor a partir de la imagen, incluso si se proporcionan argumentos adicionales.
LABEL	Agrega metadatos a la imagen en forma de pares clave-valor.
ENV	Establece variables de entorno en la imagen.
CMD	Proporciona comandos predeterminados para ejecutar cuando se inicie un contenedor a partir de la imagen.
VOLUME	Crea un punto de montaje para compartir datos entre el sistema host y el contenedor.

Veamos un ejemplo más completo:

1.1.1. Ejemplo en .NET

A continuación se presenta un ejemplo de una aplicación .NET que imprime un contador en la consola.

Archivo program.cs

El archivo `program.cs` contiene el código de la aplicación.

```
using System;
using System.Threading.Tasks;

namespace DotNetExample
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) : -1;

            while (max == -1 || counter < max)
            {
                Console.WriteLine($"Counter: {++counter}");
                await Task.Delay(TimeSpan.FromMilliseconds(1000));
            }
        }
    }
}
```

Compilación y ejecución

Para compilar y ejecutar la aplicación, sigue estos pasos:

```
dotnet new console -o DotNetExample
cd DotNetExample
dotnet run
```

El resultado debería ser la impresión del contador en la consola:

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
...
```

Publicación de la aplicación

Antes de agregar la aplicación .NET a la imagen de Docker, es recomendable publicarla para ejecutar la versión publicada en el contenedor. Para publicar la aplicación, ejecuta el siguiente comando:

```
dotnet publish -c Release -o publish
```

Esto generará una carpeta `publish` con los archivos de la aplicación publicada.

Creación del archivo Dockerfile

El archivo Dockerfile se utiliza para crear una imagen de contenedor que incluya la aplicación .NET. A continuación se muestra un ejemplo de Dockerfile:

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build-env
WORKDIR /app

COPY . ./
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/runtime:6.0
WORKDIR /app
COPY --from=build-env /app/out .

ENTRYPOINT ["dotnet", "DotNetExample.dll"]
```

Este Dockerfile utiliza una imagen de SDK de .NET para compilar y publicar la aplicación, y luego utiliza una imagen de tiempo de ejecución para ejecutar la aplicación.

Construcción de la imagen Docker

Para construir la imagen Docker que contiene la aplicación, ejecuta el siguiente comando:

```
docker build -t dotnet-example .
```

Esto construirá la imagen y la etiquetará como `dotnet-example`.

Se puede usar la aplicación Docker DesKtop para visualizar y administrar con facilidad las imagenes o contenedores que estemos usando.

Ejecución del contenedor Docker

Una vez construida la imagen, puedes ejecutar un contenedor Docker con la aplicación utilizando el siguiente comando:

```
docker run dotnet-example
```

Esto ejecutará la aplicación dentro del contenedor y mostrará la salida en la consola.

1.2. Creación y Ejecución de una Aplicación con una Red

Ya vimos cómo ejecutar una aplicación conectrandose a un contenedor de una base de datos. Es posible crear y ejecutar la aplicación y la base de datos PostgreSQL en contenedores individuales, conectándolos mediante una red personalizada en Docker.

Primero, se debe crear una red utilizando el siguiente comando:

```
docker network create mi-red
```

Este comando crea una red de Docker con el nombre “mi-red”, que se utilizará para conectar los contenedores de la aplicación y la base de datos.

A continuación, se debe ejecutar el contenedor de la base de datos PostgreSQL utilizando el siguiente comando:

```
docker run -d --name mi-postgres --network mi-red -p 5433:5432 -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password -e
    POSTGRES_DB=Curso postgres
```

En este comando, se ejecuta un contenedor a partir de la imagen de PostgreSQL, se le asigna el nombre “mi-postgres” y se conecta a la red “mi-red”. Se realiza un mapeo de puertos para que el puerto 5432 del contenedor se exponga en el puerto 5433 del host. Además, se configuran variables de entorno para el usuario, la contraseña y la base de datos de PostgreSQL.

Una vez que el contenedor de la base de datos está en ejecución, se puede construir y ejecutar el contenedor de la aplicación .NET utilizando el siguiente comando:

```
docker build -t mi-aplicacion .
docker run -d --name mi-app --network mi-red -p 7263:80 -e Pgsql="Host=mi-postgres;Port=5432;Database=Curso;User Id=postgres;
    Password=password;" mi-aplicacion
```

En estos comandos, se construye la imagen de la aplicación .NET utilizando el Dockerfile presente en el directorio actual y se le asigna el nombre “mi-aplicacion”. Luego, se ejecuta un contenedor a partir de esta imagen, se le asigna el nombre “mi-app” y se conecta a la red “mi-red”. Se realiza un mapeo de puertos para que el puerto 80 del contenedor se exponga en el puerto 7263 del host. Además, se configura la variable de entorno “Pgsql” con la cadena de conexión necesaria para conectarse al servicio de PostgreSQL.

Con estos pasos, la aplicación .NET y la base de datos PostgreSQL se ejecutan en contenedores separados, pero están conectados en la misma red personalizada “mi-red”, lo que permite la comunicación entre ellos.

Si deseas detener y eliminar los contenedores creados, puedes utilizar los siguientes comandos:

```
docker stop mi-postgres
docker stop mi-app
docker rm mi-postgres
docker rm mi-app
```

Estos comandos detienen y eliminan los contenedores, liberando los recursos utilizados.

La creación y ejecución de la aplicación y la base de datos en contenedores individuales con una red personalizada es un enfoque útil cuando se desea mayor control y flexibilidad en la configuración y comunicación entre los servicios. Sin embargo, para gestionar y orquestar aplicaciones más complejas y con múltiples servicios, se recomienda utilizar Docker Compose. A continuación, veremos cómo hacerlo.

1.3. Docker Compose

Docker ofrece una potente funcionalidad de redes que permite la comunicación entre contenedores y con el host. Cada contenedor puede tener su propia interfaz de red y dirección IP dentro de una red virtual. Esto facilita la comunicación entre contenedores y permite aislar los servicios.

Una red en Docker es una colección de contenedores conectados a la misma red virtual. Los contenedores pueden comunicarse entre sí utilizando nombres de host o direcciones IP. Docker proporciona diferentes tipos de redes, como la red predeterminada, redes puente y redes superpuestas, para adaptarse a diferentes escenarios y requisitos de comunicación.

Docker Compose es una herramienta que simplifica aún más el manejo de aplicaciones multi-contenedor en Docker. Permite definir y gestionar servicios relacionados en un archivo YAML, lo que facilita la creación y el despliegue de aplicaciones que dependen de múltiples servicios. Con Docker Compose, puedes especificar la configuración de cada servicio, incluyendo imágenes base, variables de entorno, puertos expuestos y redes en las que deben conectarse.

A continuación, se presenta un ejemplo completo utilizando Docker Compose para orquestar una aplicación que consta de un servicio de base de datos PostgreSQL y una aplicación web en .NET:

```
version: '3.8'

services:
#Servicio de base de datos PostgreSQL
    mi-postgres:
        image: postgres
        ports:
         - '5433:5432' # Mapeo del puerto del host al puerto del contenedor
        environment:
            - POSTGRES_USER=postgres
            - POSTGRES_PASSWORD=password
            - POSTGRES_DB=Curso
        volumes:
            - postgres-data:/var/lib/postgresql/data

#Servicio de la aplicacion .NET
    mi-aplicacion:
        build:
            context: .
            dockerfile: Dockerfile
        ports:
            - '7263:80' # Mapeo del puerto del host al puerto del contenedor
        depends_on:
            - mi-postgres # Dependencia del servicio mi-postgres
        environment:
            - Pgsql=Host=mi-postgres;Port=5432;Database=Curso;User Id=postgres;Password=password;

volumes:
    postgres-data: # Volumen para almacenar datos persistentes de PostgreSQL
```

A continuación, se muestra el Dockerfile utilizado en el ejemplo para construir la imagen de la aplicación .NET:

```
Build Stage
```

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /source
COPY . .

RUN dotnet restore
RUN dotnet publish -c release -o /app

Runtime Stage
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build /app .

EXPOSE 80

ENTRYPOINT ["dotnet", "EvsD.dll"]
```

En el ejemplo, se definen dos servicios: `mi-postgres` y `mi-aplicacion`. El servicio `mi-postgres` utiliza la imagen base de PostgreSQL y expone el puerto 5432 del contenedor al puerto 5433 del host. Se especifican las variables de entorno para configurar el usuario, la contraseña y la base de datos de PostgreSQL. Además, se utiliza un volumen para almacenar los datos persistentes de PostgreSQL.

El servicio `mi-aplicacion` construye la imagen de la aplicación .NET utilizando el Dockerfile proporcionado. Se expone el puerto 80 del contenedor al puerto 7263 del host y se especifica la dependencia del servicio `mi-postgres`. También se define la variable de entorno `Pgsql` con la cadena de conexión para conectarse al servicio de PostgreSQL.

Con Docker Compose, puedes ejecutar el siguiente comando para construir y ejecutar la aplicación:

```
docker-compose up -d
```

Este comando construye las imágenes necesarias, crea y ejecuta los contenedores de acuerdo con la configuración definida en el archivo Docker Compose.

La utilización de volúmenes en Docker es fundamental para garantizar la persistencia y disponibilidad de los datos de tus aplicaciones.

2. Kubernetes

Kubernetes es una plataforma portátil, extensible y de código abierto para administrar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Tiene un ecosistema grande y de rápido crecimiento. Los servicios, el soporte y las herramientas de Kubernetes están ampliamente disponibles.

2.1. Arquitectura de Clúster

En Kubernetes, la arquitectura de clúster se compone de varios elementos que trabajan juntos para administrar las aplicaciones y los servicios. A continuación, exploraremos los componentes clave de la arquitectura de clúster de Kubernetes:

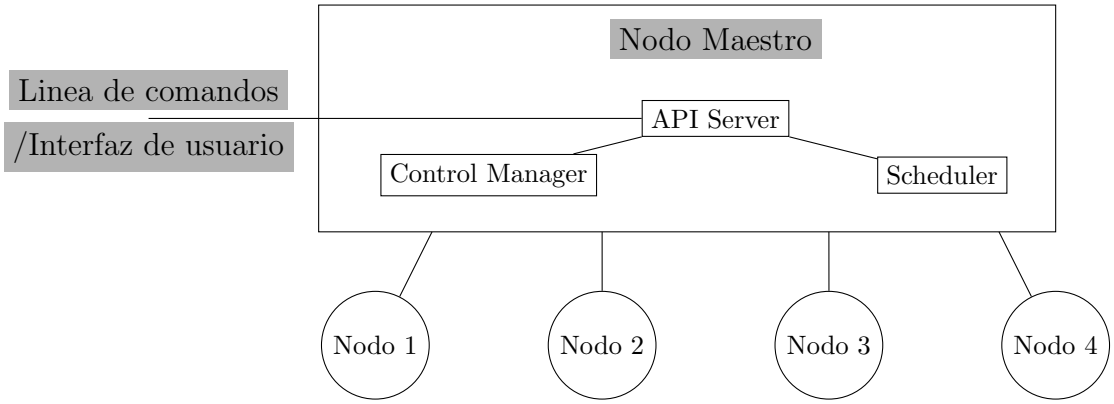

Figura 1: Arquitectura de un clúster de Kubernetes

Nodos

Un nodo en Kubernetes es una máquina física o virtual donde se ejecutan los contenedores. Cada nodo tiene un agente llamado *kubelet* que se comunica con el nodo maestro y gestiona los contenedores en el nodo. Los nodos también pueden tener otros componentes, como el *kube-proxy* para la configuración de redes.

Comunicación entre Nodos

Los nodos en un clúster de Kubernetes se comunican entre sí a través de una red. Utilizan el protocolo *kubelet API* para reportar su estado al nodo maestro y recibir instrucciones. La comunicación entre nodos es esencial para la coordinación y el equilibrio de carga en el clúster.

2.1.1. Plano de Control

El plano de control de Kubernetes es el conjunto de componentes que supervisan y gestionan el clúster en su conjunto. Estos componentes incluyen el *nodo maestro* y otros servicios centrales como el *API Server*, el *Control Manager* y el *Scheduler*.

El *API Server* proporciona una interfaz para que los usuarios y los componentes internos interactúen con el clúster. El *Control Manager* es responsable de mantener el estado deseado del clúster y realizar tareas de administración, como la gestión

de réplicas y el control de acceso. El *Scheduler* se encarga de asignar los pods a los nodos disponibles según las políticas de programación definidas.

2.1.2. Controladores

Los controladores en Kubernetes son componentes que se ejecutan en el plano de control y se encargan de gestionar aspectos específicos del clúster. Por ejemplo, el controlador de replicación se encarga de mantener el número deseado de réplicas de un pod, mientras que el controlador de servicios gestiona la conectividad de red y la carga equilibrada hacia los pods.

Estos controladores supervisan constantemente el estado del clúster y toman acciones para garantizar que las aplicaciones se ejecuten de acuerdo con las políticas definidas.

2.1.3. Arrendamientos

Los arrendamientos en Kubernetes son mecanismos utilizados para la coordinación entre los componentes del clúster. Permiten que los componentes reclamen y mantengan derechos temporales sobre un recurso, como un IP o un nombre de servicio. Los arrendamientos se utilizan para evitar conflictos y asegurar una operación coherente en el clúster.

En la arquitectura de clúster de Kubernetes, la interacción entre los nodos, el plano de control, los controladores y los arrendamientos es fundamental para garantizar la correcta administración y operación de las aplicaciones en el clúster.