

ACTIVIDAD 13 Contenedores Docker



**UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS**

KEVIN NICOLÁS SIERRA GONZÁLEZ 20182020151

LUIS MIGUEL POLO 20182020158

YEISON ALEXANDER FARFAN PERALTA 20201020138

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

INGENIERÍA DE SISTEMAS

TELEINFORMATICA I

ANDRES ALEXANDER RODRIGUEZ FONSECA

2024-III

Objetivo

- Implementar un sistema de contenedores para despliegue de servicios.

Procedimiento

Previamente se descarga e instala docker al igual que en el docker client se descarga una imagen por defecto, verificamos su existencia.

```
PS C:\Windows\system32> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker/welcome-to-docker  latest             eedaff45e3c7       12 months ago      29.5MB
PS C:\Windows\system32> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
c0ab3ecc8ef6       docker/welcome-to-docker:latest  "/docker-entrypoint..."  2 minutes ago      Up 2 minutes       80/tcp             upbea
```

Comenzamos con la instalación de Apache HTTP Server.

```
PS C:\Windows\system32> docker run -dit --name mayapache -p8080:80 httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
0062038102c9: Download complete
3ed0d9182dde: Download complete
2d429b9e73a6: Download complete
334a67c7f78b: Download complete
d675ed392a91: Download complete
4f4fb700ef54: Download complete
Digest: sha256:6bdbdf5ac16ac3d6ef543a693fd5dfafae2428b4b0cdc52a480166603a069136
```

Verificamos de nuevo la existencia de la imagen en docker y comprobamos que efectivamente se creó la imagen “httpd”.

```
4f4fb700ef54: Download complete
Digest: sha256:6bdbdf5ac16ac3d6ef543a693fd5dfafae2428b4b0cdc52a480166603a069136
Status: Downloaded newer image for httpd:latest
ef4fb8eaafc909cffc400f6b29cbcf0f2850a78f59c54246233cbc060c6678fe
PS C:\Windows\system32> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
httpd                latest             6bdbdf5ac16a       4 months ago       221MB
docker/welcome-to-docker  latest             eedaff45e3c7       12 months ago      29.5MB
```

En el Docker client se puede comprobar que existe el servidor y la imagen por defecto.

Search

Only show running containers

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	<div><div></div>upbeat_payne</div>	c0ab3ecc8ef6	docker/welcome		0%	41 minutes ago	<div><div></div><div>:</div><div></div></div>
<input type="checkbox"/>	<div><div></div>mayapache</div>	ef4fb8eaafc9	httpd	8080:80	0.01%	34 minutes ago	<div><div></div><div>:</div><div></div></div>

Ingresamos al contenedor de httpd, el SO de este contenedor es Linux.

```
PS C:\Windows\system32> Docker exec -it mayapache bash
root@ef4fb8eaafc9:/usr/local/apache2# pwd
/usr/local/apache2
root@ef4fb8eaafc9:/usr/local/apache2# ls
bin  build  cgi-bin  conf  error  htdocs  icons  include  logs  modules
root@ef4fb8eaafc9:/usr/local/apache2# cd htdocs
```










Por lo tanto, nos dirigimos al directorio donde se guardan los archivos html públicos.

```
root@ef4fb8eaafc9:/usr/local/apache2/htdocs# cat index.html
<html><body><h1>It works!</h1></body></html>
```

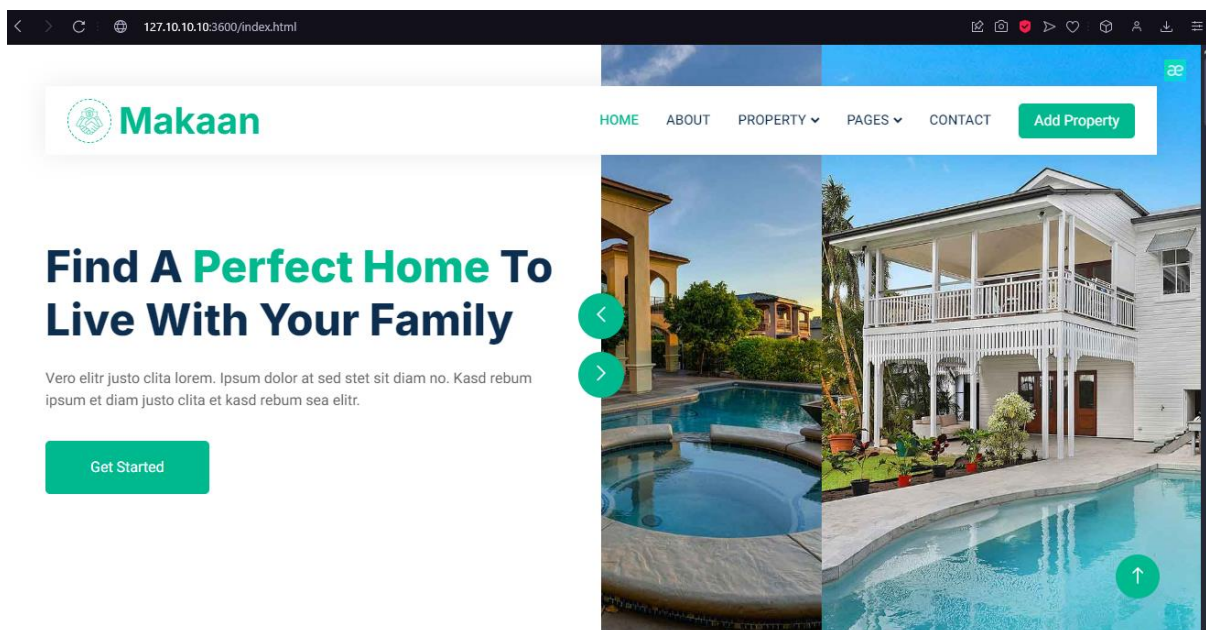
Con anterioridad descargamos el template html css que aparece en el ejemplo de la guía que ubicamos en la carpeta real-estate-html-template y creamos una instancia.

```
PS C:\Windows\system32> cd real-estate-html-template
PS C:\Windows\system32\real-estate-html-template> docker run --name myweb -dit -p 3600:80 -v ${PWD}:/usr/local/apache2/
htdocs/ httpd
ee9e513c4f328a6b32618d11bd6822b124389f185afc5029f881120dcac6efb0
PS C:\Windows\system32\real-estate-html-template>
```

Esta instancia se encuentra activa y funcionando con el nombre de myweb.

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	mayapache	ef4fb8eaafc9	httpd	8080:80	0%	3 minutes ago	  
<input type="checkbox"/>	upbeat_payne	c0ab3ecc8ef6	docker/welcome		0%	3 minutes ago	  
<input type="checkbox"/>	myweb	ee9e513c4f32	httpd	3600:80	0%	18 seconds ago	  

Corroboramos que en el localhost se encuentra corriendo dicho template.



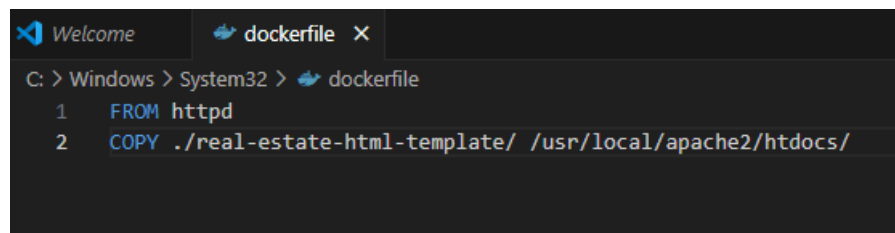
Para la creación del dockerfile debemos detener todas las imagenes activas

```

PS C:\windows\system32> docker stop $(docker ps -aq)
ee9e513c4f32
ef4fb8eaaafc9
c0ab3ecc8ef6
PS C:\windows\system32> docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
PS C:\windows\system32> docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ee9e513c4f32   httpd     "httpd-foreground"   7 minutes ago   Exited (0) 3 minutes ago
ef4fb8eaaafc9   httpd     "httpd-foreground"   About an hour ago   Exited (0) 3 minutes ago
c0ab3ecc8ef6   docker/welcome-to-docker:latest   "/docker-entrypoint..."   About an hour ago   Exited (0) 3 minutes ago
upbeat_payne
PS C:\windows\system32> docker rm $(docker ps -aq)
ee9e513c4f32
ef4fb8eaaafc9
c0ab3ecc8ef6

```

Nos dirigimos al entorno de desarrollo visual estudio code y desde la posición por fuera de la carpeta donde se encuentra la carpeta template creamos el dockerfile con las siguientes instrucciones:



```

C: > Windows > System32 > dockerfile
1 FROM httpd
2 COPY ./real-estate-html-template/ /usr/local/apache2/htdocs/

```

Desde el powershell hacemos uso del dockerfile.

```

PS C:\> cd windows
PS C:\windows> cd system32
PS C:\windows\system32> docker build -t docker-apache .
[+] Building 2.1s (7/7) FINISHED                                docker:desktop-linux
=> [internal] load build definition from dockerfile              0.1s
=> => transferring dockerfile: 110B                             0.0s
=> [internal] load metadata for docker.io/library/httpd:latest  0.0s
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build context                                0.7s
=> => transferring context: 1.58MB                               0.7s
=> [1/2] FROM docker.io/library/httpd:latest@sha256:6bdbdf5ac16ac3d6ef543a693fd5dfafae2428b4b0cdc52a480166603a06 1.4s
=> => resolve docker.io/library/httpd:latest@sha256:6bdbdf5ac16ac3d6ef543a693fd5dfafae2428b4b0cdc52a480166603a06 1.3s
=> [2/2] COPY ./real-estate-html-template/ /usr/local/apache2/htdocs/ 0.1s
=> exporting to image                                           0.4s
=> => exporting layers                                           0.2s
=> => exporting manifest sha256:fbcf4e49e5fa4acd1e13ca57b8c138d2dd073ea9c184711cdecea849ad2cb094 0.0s
=> => exporting config sha256:6c2c66a318f459e4019634dfc07abea9d006343f3a4fbfdec290f5c066e337f8 0.0s
=> => exporting attestation manifest sha256:384c8ce053f816fd106adbb13275bbf31c51991240f77a7a25387e3205bfee7f 0.0s
=> => exporting manifest list sha256:5e71d15f734209315f940eed2b7a27e3d61dff7c05eb96dad42b7ea367c639b 0.0s
=> => naming to docker.io/library/docker-apache:latest         0.0s
=> => unpacking to docker.io/library/docker-apache:latest      0.1s
PS C:\windows\system32>

```

Confirmamos la creación de una nueva imagen.

```

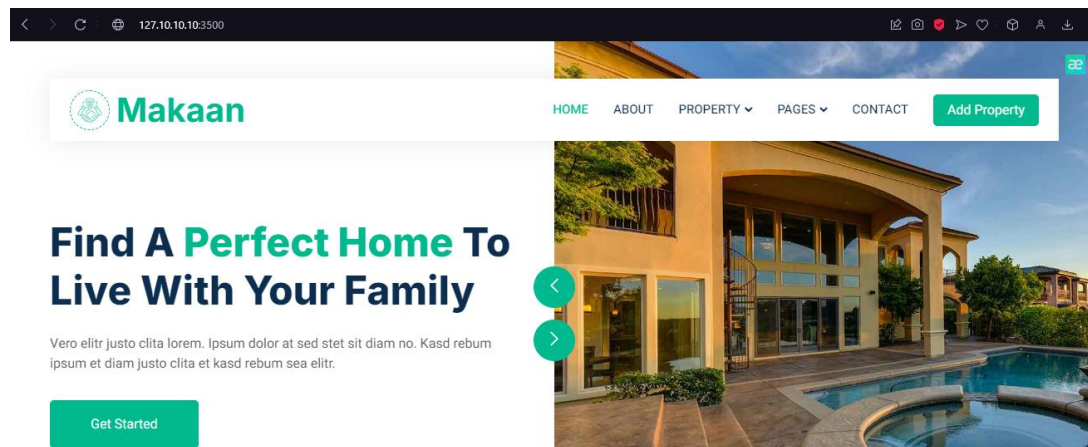
PS C:\windows\system32> docker images
REPOSITORY      TAG         IMAGE ID      CREATED        SIZE
docker-apache   latest     5e71d15f7342 About a minute ago 224MB
httpd           latest     6bdbdf5ac16a 4 months ago  221MB
docker/welcome-to-docker latest     eedaff45e3c7 12 months ago  29.5MB

```

Creamos un contenedor para la imagen.

```
PS C:\windows\system32> docker run -p 3500:80 --name mayapach2 -d docker-apache
423eb78dfbb3a7bfee249ba21de07b067dc7e05e63b9468ec01065ff8ea73c71
```

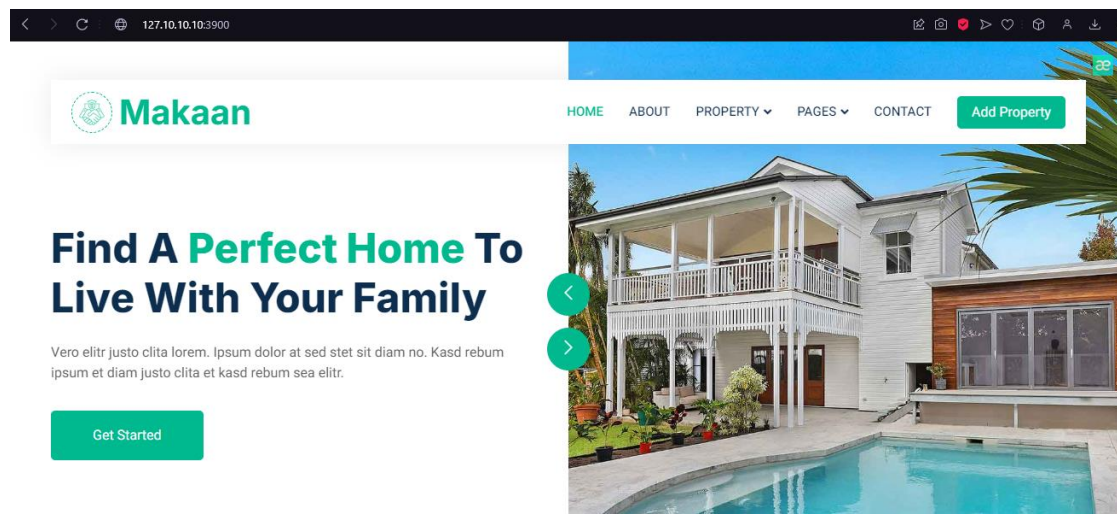
Confirmamos desde el localhost que al acceder al puerto correspondiente también envía la respuesta esperada.



Desde le powershell creamos más contenedores con la misma imagen.

```
PS C:\windows\system32> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
423eb78dfbb3   docker-apache  "httpd-foreground"      3 minutes ago Up 3 minutes  0.0.0.0:3500->80/tcp               mayapach2
e6325b5d04c9   docker-apache  "httpd-foreground"      10 minutes ago Up 10 minutes  0.0.0.0:3900->80/tcp               myapach2
```

Accedemos desde el localhost con la última ruta para confirmar que funciona.



Desde el docker desktop se muestra lo siguiente:

<div> <input type="text" value="Search"/> <div> <div></div> <div>Only show running containers</div> </div> </div>							
<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	myapach2	e6325b5d04c9	docker-apache	3900:80	0.01%	12 minutes ago	<div> <div></div> <div></div> <div></div> </div>
<input type="checkbox"/>	mayapach2	423eb78dfbb3	docker-apache	3500:80	0%	6 minutes ago	<div> <div></div> <div></div> <div></div> </div>

Preguntas

¿Cómo puedes asegurar un contenedor Docker en un entorno de producción, y qué prácticas se deben implementar para minimizar la superficie de ataque en el contenedor y en la imagen de Docker?

Rta: Asegurar un contenedor Docker en un entorno de producción implica adoptar prácticas que fortalezcan tanto el contenedor como la imagen de Docker utilizada. En primer lugar, es crucial minimizar la superficie de ataque mediante la construcción de imágenes ligeras y seguras. Esto se logra utilizando imágenes base oficiales o verificadas, eliminando dependencias innecesarias y evitando incluir herramientas o bibliotecas que no sean indispensables para la funcionalidad del contenedor. Además, se recomienda escanear las imágenes en busca de vulnerabilidades utilizando herramientas como Trivy o Docker Security Scanning antes de implementarlas en producción. Mantener las imágenes actualizadas con los últimos parches de seguridad también es esencial.

Por otro lado, es fundamental configurar adecuadamente el entorno del contenedor. Esto incluye ejecutar los contenedores con usuarios no privilegiados para reducir el impacto de posibles ataques. Evitar el uso de permisos excesivos, como el acceso al host a través de opciones como `--privileged`, es clave para limitar el alcance de los contenedores. También se deben definir políticas estrictas de red utilizando herramientas como Docker Network o CNI, asegurándose de que los contenedores solo tengan acceso a los recursos necesarios.

Asimismo, implementar controles adicionales de seguridad, como el uso de perfiles de AppArmor o SELinux, puede restringir las operaciones que un contenedor puede realizar en el sistema host. Adicionalmente, los sistemas de monitoreo y registro, como Docker Audit Logs o herramientas de observabilidad como Prometheus y Grafana, son cruciales para detectar y responder rápidamente a actividades sospechosas.

Finalmente, se recomienda adoptar prácticas DevSecOps, incorporando la seguridad desde el inicio del ciclo de desarrollo, para garantizar que tanto las imágenes como las configuraciones cumplan con los estándares de seguridad desde su concepción.

Explica cómo podrías configurar Docker para usar almacenamiento persistente en un clúster Kubernetes, incluyendo detalles sobre cómo gestionar el almacenamiento dinámico mediante Persistent Volumes y Persistent Volume Claims.

Rta: Configurar Docker para usar almacenamiento persistente en un clúster de Kubernetes consiste en aprovechar los mecanismos de Persistent Volumes (PV) y Persistent Volume Claims (PVC), los cuales permiten gestionar el almacenamiento de

manera dinámica y separada de los contenedores. En Kubernetes, un Persistent Volume es un recurso de almacenamiento independiente del ciclo de vida de un pod, mientras que un Persistent Volume Claim funciona como una solicitud que los pods utilizan para acceder a ese almacenamiento.

El primer paso para implementar esta configuración es asegurarse de que el clúster de Kubernetes esté conectado a un proveedor de almacenamiento compatible, como NFS, Ceph, AWS EBS o un sistema de almacenamiento CSI (Container Storage Interface). Una vez integrado el proveedor, se debe configurar una StorageClass, que define las características del almacenamiento, como el tipo de volumen, la política de aprovisionamiento y los parámetros del backend. Por ejemplo, una StorageClass puede estar configurada para crear volúmenes dinámicamente en un sistema de almacenamiento en la nube, como Amazon EBS o Google Persistent Disk.

Con la StorageClass configurada, Kubernetes puede aprovisionar Persistent Volumes de manera dinámica en respuesta a un Persistent Volume Claim. Un PVC es un recurso que se crea en el clúster y especifica las necesidades de almacenamiento de un pod, como la capacidad y el tipo de acceso (lectura/escritura compartida o exclusiva). Cuando un pod requiere almacenamiento persistente, se indica el PVC en su manifiesto, y Kubernetes verifica la StorageClass asociada para crear un volumen que cumpla con los requisitos, o asignar uno existente.

Por último, es esencial asegurar que el almacenamiento esté correctamente montado y accesible para las aplicaciones dentro de los pods. Esto se consigue configurando los volúmenes en los manifiestos de los pods y asegurándose de que el PVC esté bien referenciado. Además, es necesario implementar prácticas de respaldo y recuperación para asegurar la persistencia de los datos ante posibles fallos. Al combinar StorageClass, PV y PVC, se optimiza la gestión del almacenamiento persistente en entornos Docker y Kubernetes.

Detalla el proceso y las consideraciones necesarias para migrar una aplicación monolítica a contenedores Docker, incluyendo la descomposición de componentes, manejo de dependencias y estrategias para la gestión del estado.

Rta: Migrar una aplicación monolítica a contenedores Docker requiere un enfoque planificado y gradual para garantizar que la aplicación pueda beneficiarse del entorno de contenedores sin comprometer su funcionalidad o estabilidad. El primer paso es analizar la arquitectura del monolito para identificar sus componentes principales, como módulos funcionales, dependencias y puntos de integración con sistemas externos. Aunque en un inicio se puede migrar todo el monolito como un único contenedor para simplificar la transición, lo ideal es planificar la descomposición del

monolito en servicios más pequeños y autónomos que puedan ejecutarse en contenedores separados.

Durante la descomposición, se deben identificar componentes que puedan ser desacoplados, como servicios de autenticación, manejo de datos o APIs. Cada uno de estos componentes puede ser transformado en un contenedor independiente. Este proceso requiere mapear cuidadosamente las dependencias internas y externas de la aplicación. Es fundamental garantizar que estas dependencias estén contenidas o sean accesibles a través de redes definidas, evitando configuraciones rígidas que dependan de la infraestructura original. Por ejemplo, bases de datos, servidores de caché o servicios externos deben estar claramente definidos en variables de entorno o archivos de configuración.

Otro aspecto crítico es el manejo del estado. Las aplicaciones monolíticas suelen gestionar estados localmente en el sistema de archivos o en memoria, lo que no es adecuado para contenedores, ya que estos son efímeros por naturaleza. Para resolver esto, se deben externalizar los datos persistentes utilizando sistemas como bases de datos, almacenamiento en la nube o volúmenes de Docker. Además, si la aplicación incluye sesiones de usuario o caché, es necesario implementar soluciones distribuidas, como Redis o Memcached, para garantizar la disponibilidad y consistencia del estado.

Finalmente, es importante establecer un proceso de automatización para la construcción, prueba y despliegue de los contenedores, integrando herramientas como Docker Compose para entornos locales y Kubernetes para la gestión en producción. Implementar un pipeline CI/CD asegura que los cambios puedan ser probados y desplegados de manera consistente. La observabilidad también es clave: integrar sistemas de monitoreo y logging desde el principio permitirá detectar problemas de rendimiento o errores durante y después de la migración. Este enfoque estructurado ayuda a garantizar una transición exitosa de una aplicación monolítica a un entorno basado en contenedores Docker.

¿Cómo se pueden implementar y gestionar redes seguras en Docker, específicamente para contenedores que deben comunicarse entre múltiples hosts Docker en diferentes redes? Incluye una explicación sobre el uso de Docker Swarm o Kubernetes para la gestión de redes.

Rta: Implementar y gestionar redes seguras en Docker para contenedores que deben comunicarse entre múltiples hosts requiere el uso de soluciones de red como Docker Swarm o Kubernetes, que permiten configurar redes superpuestas (overlay networks)

para garantizar conectividad segura y eficiente. Estas redes superpuestas facilitan la comunicación entre contenedores en diferentes hosts al abstraer la infraestructura subyacente y proporcionar un canal virtualizado protegido.

En Docker Swarm, las redes overlay se crean automáticamente cuando se despliega un servicio. Estas redes utilizan VXLAN para encapsular el tráfico, permitiendo que los contenedores en diferentes nodos del clúster se comuniquen de forma segura. Además, Docker Swarm incluye cifrado integrado de redes superpuestas, que se puede habilitar para proteger los datos en tránsito entre los nodos. La configuración es sencilla y se define en el archivo de despliegue (`docker-compose.yml`) especificando la red para los servicios.

En Kubernetes, la gestión de redes seguras se logra mediante un plugin CNI (Container Network Interface), como Calico, Weave Net o Flannel, que proporciona conectividad entre los pods en diferentes nodos. Kubernetes abstrae las redes mediante un modelo plano, permitiendo que todos los pods puedan comunicarse sin configuraciones adicionales. Sin embargo, para reforzar la seguridad, se implementan Network Policies, que controlan el tráfico de entrada y salida basado en reglas específicas. Esto asegura que solo los contenedores autorizados puedan comunicarse.

Finalmente, tanto en Docker Swarm como en Kubernetes, es crucial monitorear el tráfico y mantener actualizados los sistemas para protegerse contra vulnerabilidades. Además, el uso de redes cifradas y la segmentación del tráfico según las necesidades de las aplicaciones fortalecen la seguridad de la comunicación entre múltiples hosts.

Describa un flujo de trabajo de CI/CD utilizando Docker, Jenkins y GitHub, incluyendo cómo se pueden construir imágenes Docker automáticamente en respuesta a cambios de código en un repositorio de GitHub y cómo estas imágenes pueden ser desplegadas automáticamente a un entorno de prueba o producción.

Rta: Un flujo de trabajo de CI/CD con Docker, Jenkins y GitHub comienza con la integración del repositorio de código en GitHub con Jenkins. Cada vez que se realiza un cambio en el código, como un commit o un pull request, GitHub activa un webhook que notifica a Jenkins para iniciar el pipeline configurado.

En el pipeline de Jenkins, el primer paso es clonar el repositorio desde GitHub y ejecutar pruebas automatizadas para garantizar la calidad del código. Si las pruebas son exitosas, Jenkins utiliza un archivo `Dockerfile` presente en el repositorio para construir una nueva imagen Docker. Esta imagen es etiquetada con un identificador único, como el número de versión o el hash del commit, y se almacena en un registro de imágenes Docker, como Docker Hub o un registro privado.

Después de construir y almacenar la imagen, el pipeline puede incluir un paso para desplegarla automáticamente en un entorno de prueba. Esto se logra mediante comandos de Docker Compose o herramientas de orquestación como Kubernetes, que extraen la nueva imagen desde el registro y actualizan los contenedores en el entorno correspondiente. Una vez validado el funcionamiento en pruebas, se puede configurar un paso adicional para desplegar la imagen en producción, siguiendo un enfoque de despliegue controlado como blue-green o canary deployments.

Este flujo automatiza todo el ciclo desde los cambios en el código hasta el despliegue en producción, asegurando rapidez y consistencia. Además, se puede ampliar con herramientas de monitoreo y alertas para supervisar la salud de las aplicaciones después del despliegue.

Analiza las diferencias entre CMD y ENTRYPOINT en un Dockerfile, y proporciona ejemplos de cuándo sería más apropiado usar uno sobre el otro. Además, explica cómo estos comandos interactúan con los argumentos pasados al iniciar un contenedor.

Rta: En un Dockerfile, CMD y ENTRYPOINT se utilizan para definir el proceso que ejecutará el contenedor al iniciarse, pero tienen diferencias clave en su comportamiento y casos de uso.

CMD especifica un comando por defecto que se ejecuta si no se proporcionan argumentos al iniciar el contenedor. Puede ser sobrescrito directamente pasando comandos al ejecutar `docker run`. Por ejemplo, en un Dockerfile con `CMD ["echo", "Hello World"]`, si se inicia el contenedor sin argumentos, se mostrará *Hello World*. Sin embargo, si se ejecuta `docker run imagen ls`, el contenedor ejecutará `ls` en lugar del comando definido en CMD. CMD es ideal para comandos genéricos que podrían necesitar ser fácilmente sobrescritos.

ENTRYPOINT, en cambio, establece el comando principal que siempre se ejecutará, aunque se pasen argumentos adicionales al iniciar el contenedor. Por ejemplo, con `ENTRYPOINT ["echo"]`, al ejecutar `docker run imagen Hello`, el contenedor ejecutará `echo Hello`. Esto hace que ENTRYPOINT sea más adecuado para aplicaciones específicas que siempre requieren un comando base, como un servidor o una herramienta CLI.

Ambos comandos pueden combinarse: ENTRYPOINT define el comando base y CMD proporciona los argumentos por defecto. Por ejemplo, con `ENTRYPOINT ["curl"]` y `CMD ["https://example.com"]`, al ejecutar el contenedor sin argumentos, se ejecutará `curl https://example.com`. Si se pasan argumentos diferentes, estos

sobrescribirán el CMD pero no el ENTRYPOINT, permitiendo flexibilidad en el comportamiento del contenedor.

Explora las implicaciones de seguridad de ejecutar aplicaciones en contenedores Docker como usuario root dentro del contenedor, y cómo el uso de User Namespaces en Docker puede ayudar a mitigar los riesgos asociados.

Rta: Ejecutar aplicaciones en contenedores Docker como usuario root dentro del contenedor es una práctica riesgosa porque, aunque el contenedor esté aislado, un atacante que comprometa el contenedor podría escalar privilegios al host si encuentra vulnerabilidades en el sistema. Además, cualquier proceso que escape del aislamiento del contenedor tendría permisos de superusuario en el sistema host, aumentando significativamente el impacto potencial de un ataque.

Para mitigar estos riesgos, se recomienda evitar ejecutar contenedores como root y utilizar User Namespaces en Docker. Los User Namespaces permiten mapear los usuarios dentro del contenedor a usuarios no privilegiados en el sistema host. Por ejemplo, un proceso que parece ejecutarse como root dentro del contenedor puede estar asociado con un usuario de bajo privilegio en el host, reduciendo el alcance de cualquier ataque. Esto añade una capa de protección incluso si un contenedor se ve comprometido.

Configurar User Namespaces implica habilitar la opción en el demonio de Docker y definir los rangos de UID y GID que se usarán para el mapeo. Sin embargo, es importante ajustar correctamente los permisos y configuraciones de volúmenes compartidos, ya que los volúmenes montados deben ser accesibles por los UIDs mapeados para evitar problemas de acceso.

Adicionalmente, combinar User Namespaces con otras prácticas como el uso de usuarios no root en las imágenes Docker (USER en el Dockerfile) y la implementación de políticas de seguridad, como AppArmor o SELinux, mejora significativamente la seguridad de los contenedores. Estas estrategias reducen la posibilidad de comprometer tanto el contenedor como el host subyacente.

¿Cómo se pueden utilizar las capacidades de autoescala de Docker en un entorno de producción, específicamente en relación con Docker Swarm o Kubernetes, para manejar variaciones en la carga de trabajo? Incluye una discusión sobre las métricas y los parámetros que se deberían monitorizar y utilizar para tomar decisiones de escalado.

Rta: Las capacidades de autoescala en Docker, a través de plataformas como Docker Swarm o Kubernetes, permiten ajustar dinámicamente la cantidad de réplicas de servicios o pods para manejar variaciones en la carga de trabajo, optimizando el uso de recursos en un entorno de producción.

En Docker Swarm, el escalado manual de servicios puede configurarse fácilmente, pero para un autoescalado más avanzado, es necesario integrar herramientas externas como Docker Autoscaler o scripts personalizados. Estas herramientas utilizan métricas como la utilización de CPU, memoria o la cantidad de solicitudes activas para decidir cuándo añadir o eliminar réplicas de un servicio. Aunque Docker Swarm no tiene autoescalado nativo, su simplicidad lo hace adecuado para entornos menos complejos.

En Kubernetes, el autoescalado es más robusto gracias al Horizontal Pod Autoscaler (HPA), que ajusta automáticamente el número de pods según métricas como la utilización de CPU, memoria o métricas personalizadas expuestas por las aplicaciones mediante Prometheus o métricas de la API de Kubernetes. Además, el Cluster Autoscaler puede escalar los nodos del clúster para garantizar que haya suficiente capacidad para alojar los pods necesarios.

Las métricas clave que deben monitorizarse incluyen el uso de CPU, la memoria disponible, las tasas de solicitudes entrantes y el tiempo de respuesta de las aplicaciones. Estas métricas ayudan a tomar decisiones sobre cuándo escalar horizontalmente (añadir más réplicas) o verticalmente (aumentar los recursos de los nodos). Es fundamental configurar umbrales adecuados para evitar escalados innecesarios y garantizar que la infraestructura responda de manera eficiente a las fluctuaciones de la carga. Además, integrar herramientas de monitoreo como Grafana y Prometheus facilita la visualización y el análisis de estas métricas en tiempo real.

Conclusiones

Con la implementación de un sistema de contenedores por medio de la herramienta Docker permitió el despliegue de servicios como Apache HTTP Server utilizado como servidor dentro de un contenedor Docker, para esto creándose una imagen con httpd, esto con el fin de validarse diferentes end-points que pueda utilizar una aplicación, es decir, facilita la configuración y desarrollo de aplicaciones ahorrando recursos del sistema, ya que sólo se utiliza las herramientas necesarias y sin comprometer la integridad del sistema operativo donde se está probando, es decir, en entornos aislados, ya que se ejecutan en espacios independientes, esta otra tecnología permite

tener otras alternativas para trabajar con softwares y aplicaciones y no depender sólo de la virtualización.

Bibliografía

- «*What is Docker?*» (2024, 10 septiembre). Docker Documentation.
<https://docs.docker.com/get-started/docker-overview/>
- «*What is Docker Compose?*» (2024, 19 septiembre). Docker Documentation.
<https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-docker-compose/>
- *Persistent volumes*. (2024, 18 octubre). Kubernetes.
<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>
- “Overlay network driver”. Docker Documentation.
<https://docs.docker.com/engine/network/drivers/overlay/>
- «*Swarm mode key concepts*». (2024, 10 septiembre). Docker Documentation.
<https://docs.docker.com/engine/swarm/key-concepts/>
- “Jenkins User Documentation”. Jenkins User Documentation.
<https://www.jenkins.io/doc/>