

Docker



Imágenes

Una imagen es una forma empaquetada de un programa para poderse ejecutar en un ambiente aislado, teniendo su propio ambiente de ejecución, dependencias y su código fuente.

Descargar imagen de Docker Hub

```
docker image pull <name>
```

Listar imágenes descargadas

```
docker image ls
```

Eliminar una imagen

```
docker image rm <id>
```

Eliminar todas las imágenes

```
docker image prune -a
```

Crear alias para imagen existente

```
docker tag miimagen:1.0 miusuario/miimagen:1.0
```

Inspeccionar una imagen

```
docker inspect <id>
```

Contenedores

Un contenedor es una instancia de una imagen, se encarga de ejecutar la imagen en un ambiente aislado de nuestra computadora, como una máquina virtual pero más ligera.

Crear y ejecutar un contenedor

```
docker container run <image-name>
```

Detener la ejecución de contenedor

```
docker container stop <id>
```

Ejecutar contenedor ya existente

```
docker container start <id>
```

Mostrar todos los contenedores

```
docker container ls -a
```

Observar los logs de un contenedor

```
docker container logs -f <id>
```

Ejecutar comandos dentro del contenedor

```
docker container exec <id> <command>
```

Eliminar un contenedor

```
docker container rm -f <id>
```

Eliminar todos los contenedores

```
docker container prune -f
```

Parámetros usados:

-f	Forzar el comando (ej. al eliminar con)
-d	Desenlazado (no seguir logs)
-p	Asignación de puerto contenedor/local
-e	Variables de entorno necesarias

-a	Forzar a listar todo
-v	Asignar un volumen
-it	Entrar a la terminal del contenedor
--name	Nombrar el contenedor

Volumenes

Un volumen es una forma en que un contenedor puede almacenar información de forma persistente sin que se pierda al eliminar el contenedor y puede ser compartida entre varios.

Crear un volumen

```
docker volume create <name>
```

Listar volúmenes creados

```
docker volume ls
```

Eliminar un volumen

```
docker volume rm <id>
```

Eliminar todos los volúmenes.

```
docker volume prune
```

Existen 3 tipos de volúmenes:

- Simple volumes: Administrados por Docker, son la forma mas recomendada de persistir datos, se pueden compartir fácilmente y solo necesitas darle un nombre.
- Bind mount volumes: Enlazados del contenedor a la computadora host (la propia), todo archivo que el contenedor o el host altere, se ve reflejado el cambio en ambos.
- Tmpfs: Se guardan en memoria RAM y se perderán al reiniciar el contenedor

Redes

Cuando se tienen más de un contenedor y se busca una comunicación entre ellos, al estar en un entorno aislado no es posible la comunicación, es necesario crear una red en Docker para ello. Docker se encarga de asignar IP y DNS a cada contenedor conectado.

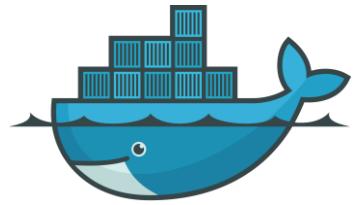
Crear una red

```
docker network create <name>
```

Agregar un contenedor a la red

```
docker network connect <net> <id-cont>
```

Docker



Desconectar un contenedor de la red

```
docker network disconnect <net> <id-cont>
```

Eliminar una red

```
docker network rm <net>
```

Eliminar todas las redes

```
docker network prune
```

Ver detalles de la red

```
docker network inspect <net>
```

(--help para obtener ayuda de un comando)

Terminal Interactiva

Puedes acceder a la terminal interactiva dentro del contenedor y navegar dentro de él.

```
docker container exec -it <id> /bin/bash
```

Dentro del contenedor tendrás los comandos de Linux disponibles para hacer fiesta.

Comandos de LINUX

- ls: Listar el directorio.
- cd: Movernos entre directorios.
- cat: Ver el contenido de un archivo
- vi: Editar archivo.
- i: empezar a editar.
- Esc+wq!: Guardar y salir
- Exit: salir de la terminal

Docker Compose

Es un archivo que se usa para no tener que repetir tanto todos los “container run”. Así solo se escribe una vez y se ejecuta con un solo comando.

Puede manejar variables de entorno en un archivo .env

Compose crea una red por defecto entre los servicios con DNS de nombre de contenedor a cada uno.

Ejecutar Docker Compose (en la ruta donde este docker-compose-yaml)

```
docker compose up
```

Eliminar y bajar todos los contenedores

```
docker compose down
```

Ejemplo de docker-compose.yaml

```
services: #? SERVICIOS / CONTENEDORES
  db: #? NOMBRE DEL SERVICIO
    container_name: 'mongo' ## nombre contenedor
    image: mongo:4.0 ## imagen
    volumes: ## volumen a usar
      - pokeball:/data/db
    environment:
      - MONGO_INITDB_ROOT_USERNAME=${MUSER}
      - MONGO_INITDB_ROOT_PASSWORD=${MPASS}
    restart: always
    command: ['--auth'] ## comando que ejecuta en shell
    ports:
      - "27017:27017"
  express-db:
    container_name: express-db-container
    depends_on: - db ## Esperar a otro contenedor
    image: mongo-express:1.0.0-alpha.4
    environment: ## env variables
      - ME_CONFIG_MONGODB_ADMINUSERNAME=${MUSER}
      - ME_CONFIG_MONGODB_ADMINPASSWORD=${MPASS}
      - ME_CONFIG_MONGODB_SERVER=${MDB}

    ports: ## Puertos enlazados
      - "8080:8081"
    restart: always ## Reinicia si hay error
  volumes: ## Declaracion de volumenes
    pokeball: ## Nombre del volumen
      external: true ## Si el volumen ya existe
```

Dockerfile

Dockerfile es el archivo que nos permite crear imágenes propias de nuestro proyecto.

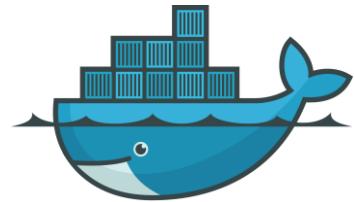
FROM: Instrucción inicial de un Dockerfile, donde marcamos la imagen base para crear nuestra propia imagen (node, python...)

```
FROM <image:tag>
```

COPY / ADD: Copea nuestros archivos locales a la imagen destino (origen / destino) (. coppear todo).

```
COPY app.js /app
```

Docker



WORKDIR: Navegar a una carpeta dentro de la imagen.

```
WORKDIR /app
```

RUN: Ejecutar comando en la consola de la imagen.

```
RUN npm install
```

CMD: Comandos que se ejecutaran cuando la imagen se corra en un contenedor.

```
CMD ["node", "app.js"]
```

(Pueden ejecutarse test con RUN, al fallar la imagen no se construirá)

EXPOSE: Expone el puerto del contenedor cuando se ejecute la aplicación dentro del mismo, para poderse mapear en el run container con -p

```
EXPOSE 3300
```

ENV: Crea variables de entorno en el contenedor (variable de sistema) que se pueden sobrescribir en el run container, por lo que declararla en el dockerfile solo es para colocar valor por defecto.

```
ENV variable valor
```

Construir imagen

Es un proceso más organizado para construir imágenes, donde utilizamos varias imágenes para dividir etapas en el proceso y crear con ellas una imagen final. Cada etapa tendrá un alias “as” y lo usaremos de referencia cuando necesitemos algo de cierta etapa en una nueva.

Ejemplo de Dockerfile:

```
# Primera etapa "Instalacion de dependencias dev"
FROM node:19.2-alpine3.16 as test-deps
WORKDIR /app
COPY package.json ./
RUN npm install

# Segunda etapa "Ejecucion de los tests"
FROM node:19.2-alpine3.16 as tests
WORKDIR /app
COPY --from=test-deps /app/node_modules /node_modules
COPY . .
RUN npm run test
```

```
# Tercera etapa "Dependencias de produccion"
```

```
FROM node:19.2-alpine3.16 as prod-deps
```

```
WORKDIR /app
```

```
COPY package.json .
```

```
RUN npm install -prod
```

```
# Cuarta etapa "Runner de la app en prod"
```

```
FROM node:19.2-alpine3.16 as prod-runner
```

```
WORKDIR /app
```

```
COPY --from=prod-deps /app/node_modules /node_modules
```

```
COPY . .
```

```
CMD ["node", "app.js"]
```

Imagen con Docker Compose

Para usar una imagen al correr un contenedor, pero quieres crearla en el momento justo en donde se ejecuta el docker compose, puede usar build en lugar de image, ahí puedes seleccionar el dockerfile que construye tu imagen (útil para desarrollo y ver los cambios rápidamente sin construir la imagen a cada rato).

```
chronotrack:
```

```
  build:
```

```
    context: './'
```

```
    dockerfile: 'Dockerfile'
```

```
    image: 'chronotrack-backend'
```

```
Construir imagen con docker compose
```

```
docker compose build
```

```
Construir servicio específico del compose
```

```
docker compose build <servicio>
```

Builders

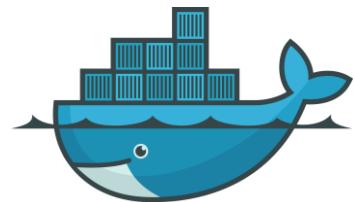
Construir imagen de Docker para distintas arquitecturas

Un builder en Docker es el encargado de crear y construir tu imagen a travez de un Dockerfile

Docker usa un motor llamado **BuildKit** para hacer esto de forma más rápida y eficiente

Cuando haces docker build, estás usando un builder por defecto, pero puedes usar builders más avanzados para construir imágenes para distintas arquitecturas.

Docker



BuildX

Buildx es una herramienta avanzada de Docker que te da superpoderes para construir imágenes. Con ella puedes:

- Construir imágenes para múltiples arquitecturas.
- Usar caché más inteligente para acelerar builds.
- Hacer builds remotos (por ejemplo, en un servidor o en la nube).
- Subir directamente al Docker Hub desde el build

Crear tu builder personalizado

```
docker buildx create --name <builder> --use
```

Ver builders disponibles

```
docker buildx ls
```

Construir una imagen multiplataforma

```
docker buildx build --platform <platforms> -t <tagname> --push <where is Dockerfile>
```

Ejemplo

```
docker buildx build --platform linux/amd64,linux/arm64 -t usuario/miImagen:latest --push .
```

(--push: Este parámetro sube directamente la imagen construida al **Docker Hub** u otro registro, sin necesidad de hacer docker push manualmente después.)

--load: Si solo será local)

Exportar imagen .tar

```
docker buildx build -t <image/tagname> --output type=docker,dest=<directory/name.tar> <directory Dockerfile>
```

```
docker buildx build -t miapp:latest --output type=docker,dest=miapp.tar .
```

Docker Hub

Al igual que github, Docker cuenta con repositorios públicos y privados para subir tus imágenes, con versión gratuita y de paga, para poder subir tus imágenes es necesario ingresar tus credenciales de docker hub.

Loggearse en DockerHub (necesitas crear tu cuenta desde el navegador en el sitio)

```
docker login
```

Etiquetar imagen (para luego subirla a DockerHub)

```
docker tag miimagen:1.0 miusuario/miimagen:1.0
```

- *miimagen:1.0*: nombre local que tenías
- *miusuario/miimagen:1.0*: así debe llamarse en DockerHub (incluye tu usuario)

(Puedes usar :latest si quieres tener la versión "por defecto".)

Subir tu imagen a DockerHub

```
docker push miusuario/miimagen:1.0
```

(Cuesta varo una suscripción para subir tus imágenes :c)

CI/CD (Continuos integrations and Continuos Deployments)

GitHub Actions (Pipelines)

Un pipeline es una serie de pasos automatizados que se ejecutaran cada vez que sucede algo en tu repositorio de GitHub, por ejemplo, si hacemos un push para subir nuestro código al repositorio de git, podemos hacer automáticamente, la creación de la imagen de Dockerfile, y subirla a nuestro registro

The screenshot shows a GitHub repository interface. At the top, there's a navigation bar with 'Code', 'Issues', 'Pull requests', 'Actions' (which is circled in red), 'Projects', and 'Security'. Below the navigation bar, the repository name 'ReservameKbron_Frontend' is shown, along with a 'Private' link. Under the repository name, there's a section titled 'Get started with GitHub Actions' with a sub-section 'Categories' containing 'Deployment', 'Continuous integration', 'Automation', and 'Pages'. To the right, there's a search bar with 'docker image' and a list of 'Found 3 workflows': 'Jekyll using Docker image' (By GitHub Actions), 'Docker image' (By GitHub Actions), and 'Publish D' (By GitHub). Each workflow has a 'Configure' button.

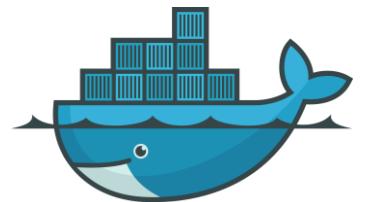
Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started. Skip this and set up a workflow yourself →

This part of the screenshot shows the search results for 'docker image' in the GitHub Actions search bar. It lists three workflows: 'Jekyll using Docker image' (By GitHub Actions), 'Docker image' (By GitHub Actions), and 'Publish D' (By GitHub). Each workflow has a 'Configure' button and a small preview icon.

Esto nos generara un YAML (.github/workflows/<archivo.yaml>) con los pasos a seguir (pipelines) que ejecutara ciertos procesos cada que un evento de git suceda.

Docker



Ejemplo workflow.yaml

```
## Nombre del workflow
name: Build & Push Docker image

on: ## Bajo que evento se ejecutara el workflow
  push: ## Al hacer push en rama Main
    branches: [ main ]
jobs: ## Trabajos a realizar
  build:
    runs-on: ubuntu-latest
    steps:
      ## Clona tu repositorio dentro del entorno del runner.
      ## Sin esto, git no tendra acceso al codigo
      - name: Checkout código ## Titulo del paso
        uses: actions/checkout@v4

      ## Login al DockerHub para subir la imagen
      - name: Login to DockerHub
        ## Variables definidas en el paso
        with:
          DOCKER_USER: ${{ secrets.DOCKER_USERNAME }}
          DOCKER_PASS: ${{ secrets.DOCKER_PASSWORD }}

      ## Comando a ejecutar en el paso
      run: docker login -u $DOCKER_USER -P $DOCKER_PASS

      ## Ejecucion de construccion de la imagen
      - name: Build Docker image
        run: docker build -t <usuario>/<img>:<tag> .

      ## Subir la imagen a nuestra repo de DockerHub
      - name: Push Docker image
        run: docker push <usuario>/<img>:<tag>
```

Estos workflows se ejecutan en un servidor de Github, no se ejecutan en local, si se usan estos pipelines por mucho tiempo, probablemente inicie el proceso de facturación y cobro de github por el uso de computo

También se puede colocar las versiones <tag> automático con otras acciones de github para sacar la versión de los mensajes de los commits,

Cluster: Conjunto de computadoras que trabajan juntas como si fueran una sola.

Varias máquinas (físicas o virtuales) unidas en

equipo para compartir trabajo, aumentar potencia y asegurar que todo siga funcionando, aunque una falle.

Kubernetes (K8s)



Kubernetes es un orquestador de contenedores, originalmente creado por Google, es una plataforma que automatiza el despliegue, escalado y manejo de contenedores.

Kubernetes:

- Crea, destruye y reinicia contenedores automáticamente
- Balancea el tráfico entre replicas
- Guarda datos en volúmenes persistentes
- Maneja .env y secrets de forma segura
- Te permite escalar con un comando (kubectl scale)
- Te da actualizaciones sin tumbar todo (rolling updates)
- Automatiza el deploy de tus servicios

Conceptos:

POD: Es el contenedor mínimo que K8s maneja, puede tener 1 o varios contenedores dentro (generalmente 1), es una capa abstracta que cubre nuestros contenedores para remplazarlos fácilmente.

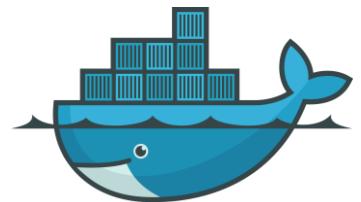
Cada contenedor dentro de los PODs, tienen una IP única asignada, cada que se destruye un POD, las ip asignadas cambian.

Service: Tienen una ip única asignada y este sabe la IP de los contenedores (aunque cambien), para la comunicación entre los mismos (Internos y externos)

Ingress: Maneja rutas, TLS, dominios. “/api va al backend”, “/web va al frontend”, etc. Es la puerta de entrada HTTP a tu cluster en K8s (como un router inteligente que dirige el tráfico dependiendo el endpoint).

ConfigMap: Son archivos de configuración que no son secretos, como variables, settings, etc.

Docker



Secret: Igual que ConfigMap, pero para contraseñas, tokens, llaves privadas, etc. Se guarda cifrado o encriptados y ya son secretas.

Volumen / PVC: Para guardar datos que no se borren si el pod (grupos de contenedores) muere. Por ejemplo, la base de datos. Exactamente el concepto de volúmenes ya vistos en Docker.

Deployment: Aquí es donde escalamos vertical o horizontal (aumento de poder o de máquinas), cuantos PODs quieres y como reiniciarlos si mueren.

StatefulSet: Similar a los deployments pero para desplegar bases de datos, servicios que necesitan una identidad única.

Minikube

[minikube start | minikube](#)

Minikube es una herramienta que te permite crear un clúster de Kubernetes en tu propia computadora, sin necesidad de servidores externos ni la nube.

Es como tener una versión mini de Kubernetes para hacer pruebas, aprender o desarrollar localmente.

Ver version de minikube

`minikube version`

Crear cluster

`minikube start`