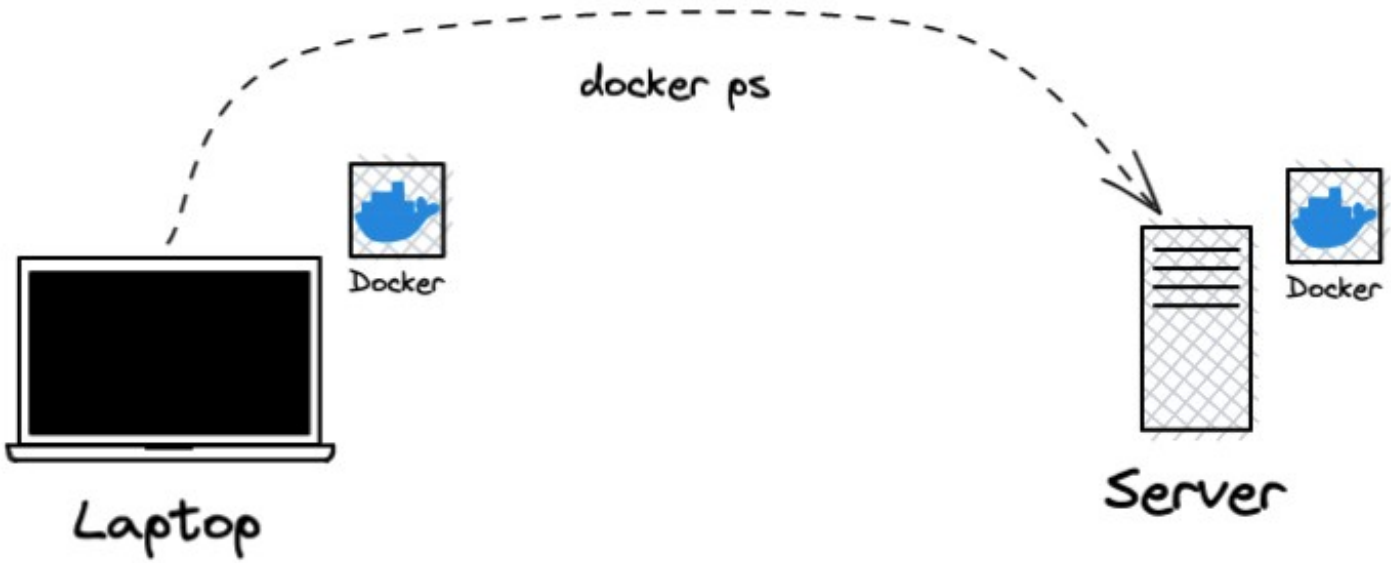
Docker Build

Índice

Docker Context

Docker Context

Herramienta que me permite manejar múltiples entornos de Docker.



Consideraciones:

1. Los volúmenes deben ser con ruta absoluta en el servidor.
2. El docker context del server es importante.

Docker Context

Definición de contextos para el curso Docker25:

Contexto: default

- Configuración Docker del laboratorio

Contexto: rootless

- Rootless

Contexto: cap

- Capabilities

Contexto: inseguro

- Docker inseguro

Contexto: rmap

- Isolate containers con user namespaces

Contexto: n

- ...

Docker Context

Forma de trabajar para probar diferentes configuraciones de dockerd:

Probando elevación de privilegios (lab3):

1. docker context use inseguro => Realizar test
2. docker context use default => Realizar test
3. docker context use rootless => Realizar test
4. Extender los laboratorios => ¿Contexto Windows?

Problemas:

- Dificultad para crear instancias.
- AWS no es un requisito para el curso.
- Configuraciones de los laboratorios se solapan. Rootless y namespace-map.

Docker Context

Solución: 6 contextos definidos

Usuario inti:

- **Contexto default:** *Configuración Docker del laboratorio*
- **Contexto inseguro:** *docker25-inti-inseguro.jpaniorte.com*
- **Contexto rootless:** *docker25-inti-rootless.jpaniorte.com*
- **Contexto rmap:** *docker25-inti-rmap.jpaniorte.com*
- **Contexto cap:** *docker25-inti-cap.jpaniorte.com*
- **Contexto remoto1:** *docker25-inti.jpaniorte.com*

Forma rápida de crear contexto (sin moficar .ssh/config):

```
docker context create  inseguro --docker \
    "host=ssh://ubuntu@docker25-inti-inseguro.jpaniorte.com” \
    “key=/ruta/clave/privada”
```

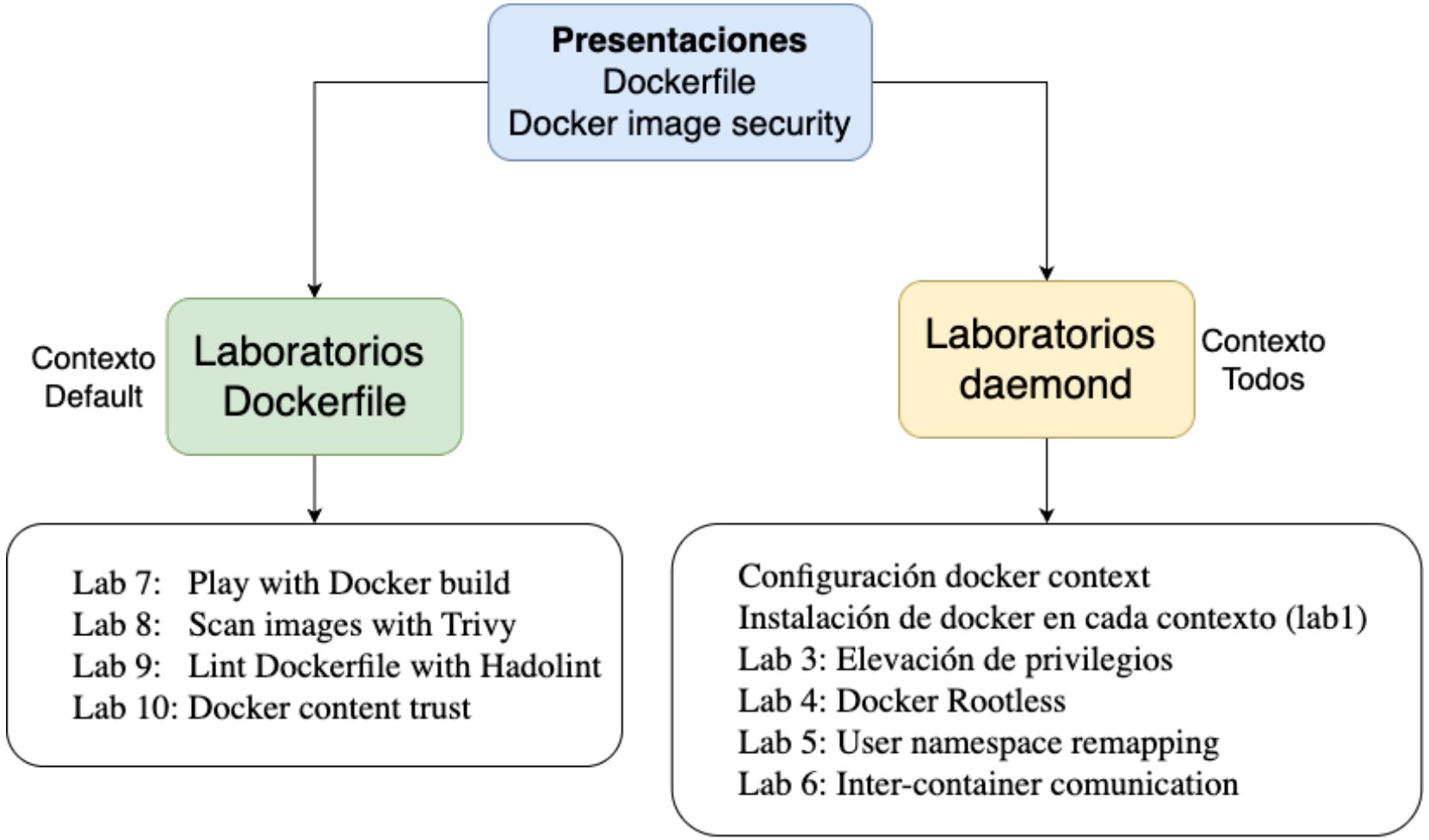
Docker Context

Recomendaciones

1. Usar Ubuntu.
2. Configuración del dockerd según indique el laboratorio:
    - SSH a la máquina
    - Configuración del daemond
    - Cierro la terminal

3. Pruebas y tests:
    - Docker context use <contexto>
    - Prueba
    - Docker context use <otro_contexto>
    - Prueba

Hoja de ruta



Recordatorio:
Si tienes usuario de github, puedes levantar las máquinas de los contextos a demanda.

1. Docker Build

1. Docker Build: Arquitectura

Docker Build => arquitectura cliente-servidor:

- Buildx: CLI para ejecutar y gestionar builds.
- BuildKit: servidor, (o compilador), que gestiona la ejecución de la build.

Docker Build Options:

- Fichero Dockerfile ←
- Argumentos ←
- Opciones de exportación
- Opciones de Caché

Docker Build Resources:

- Filesystem del contexto ←
- Build Secrets ←
- SSH Sockets
- Registry Auth Tokens

2. Dockerfile

2. Dockerfile

```
1 FROM ubuntu:24.04
2 RUN apt-get update &&\
      apt-get -y install <package>

3 COPY script.sh /usr/bin/do_action
4 RUN  chmod +x /usr/bin/do_action
5 CMD ["do_action"]
```

- Cada línea del fichero representa una instrucción.
- Cada instrucción comienza con una etiqueta. (RUN, COPY, FROM, ..)
- Se ejecutan de forma secuencial .

2. Dockerfile

```
FROM python:3.11-alpine3.21

WORKDIR /app/scripts
COPY requeriments.txt /app
RUN pip i –r /app/requirements.txt
COPY ./scripts /app/scripts
ENTRYPOINT python
```

lifecycle

docker run -it -v $(pwd):/app python:3.11-alpine3.21 sh

docker build –t py_scripts:1.0 .

docker run py_scripts:1.0 do_action.py

2. Dockerfile
Jerarquía y Cache

⇒ Optimizar el proceso de construcción de una imagen.

id: 123abc123 `FROM golang:1.20-alpine` =>imagen base

id: 223abc123 `WORKDIR /src` => directorio de trabajo

id: 323abc123 `COPY go.mod go.sum .` =>app sources

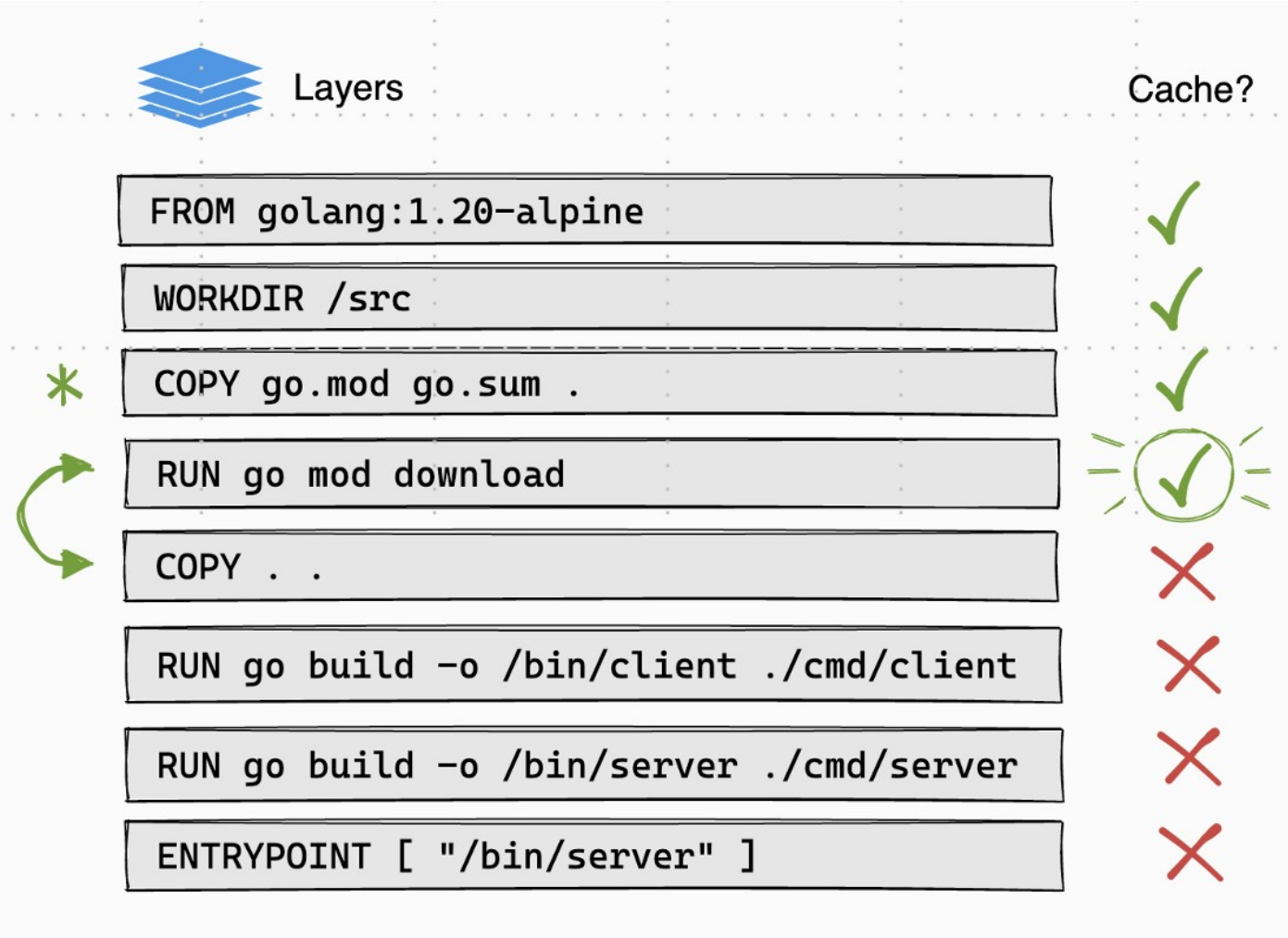
id: 423abc123 `RUN go mod download` =>instalar dependencias

En una segunda construcción de este fichero Dockerfile,

1. Docker intentará reutilizar cada una de estas capas desde su cache
2. Si una de las capas cambia, el resto de las capas de deben reconstruir

¿Es óptimo el fichero Dockerfile anterior?

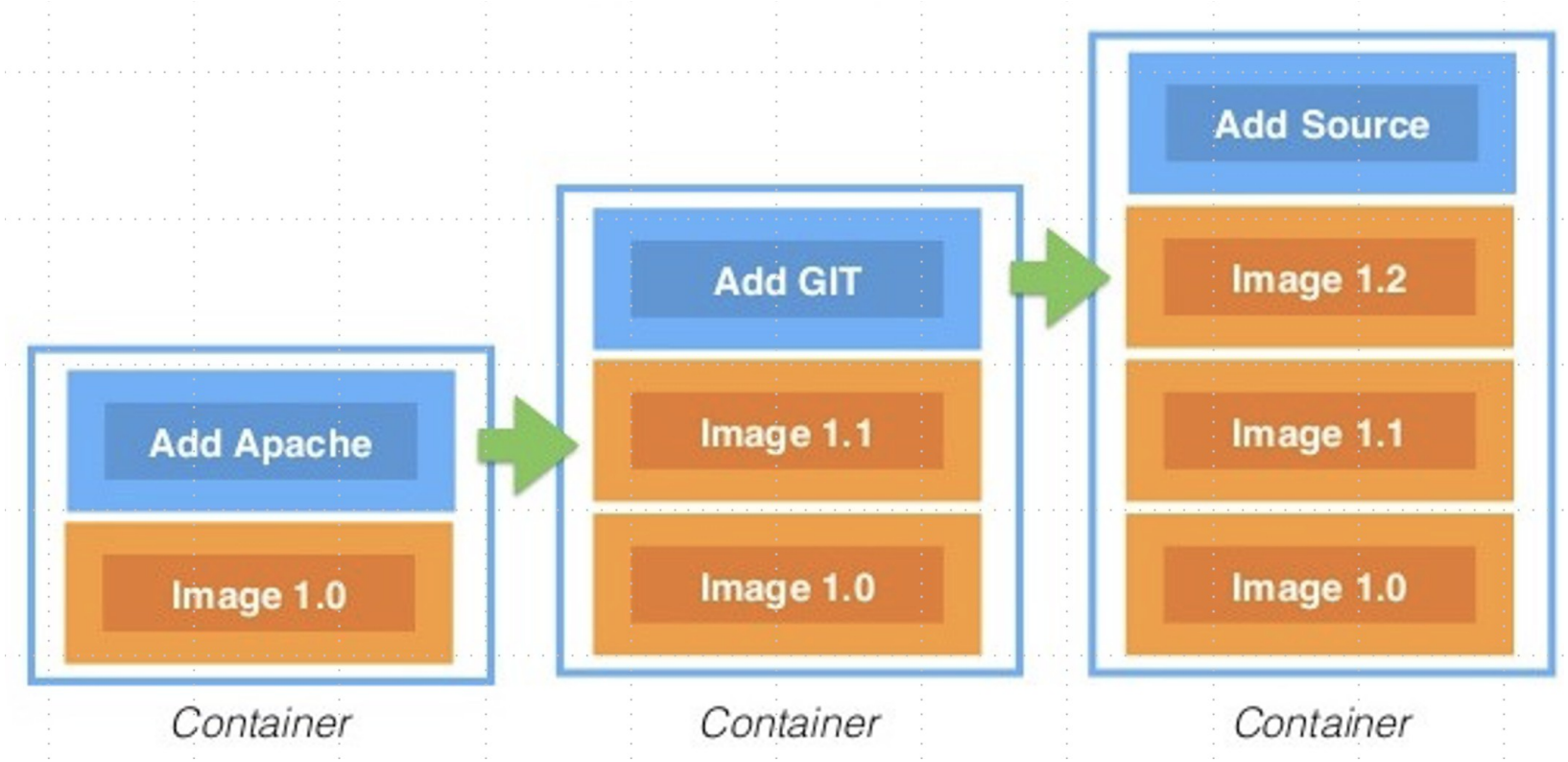
2. Dockerfile
Jerarquia y Cache



Cuando se ejecuta docker build, el constructor intenta reutilizar las capas de compilaciones anteriores. Si una capa no se modifica, esta se obtiene de cache.

Si una capa cambia desde la última construcción, esa capa y todas las siguientes deben reconstruirse.

2. Dockerfile
Union File System



Capas inmutables: no se puede eliminar o modificar

2. Dockerfile

Union File System

⇒ Analizamos el siguiente Dockerifle

id: 123abc123 `FROM busybox`

id: 223abc123 `RUN touch a` => creo un fichero

id: 323abc123 `RUN rm a` => elimino el fichero

¿El fichero a consume espacio en disco?

2. Dockerfile

Union File System

⇒ Analizamos el siguiente Dockerifle

id: 123abc123 `FROM busybox`

id: 223abc123 `RUN touch a && rm a`

¿El fichero a consume espacio en disco?

3. Etiquetas

3. Etiquetas

- Imagen base: FROM
- Metadatos: LABEL
- Instrucciones de construcción:
  RUN, COPY, ADD, WORKDIR
- Configuración:
  USER, EXPOSE, ENV
- Instrucciones de arranque:
  CMD, ENTRYPOINT

Documentación: https://docs.docker.com/reference/dockerfile/

3. Etiquetas: FROM

- Es obligatorio y debe ser la primera etiqueta.

FROM imagen[:tag] [AS alias]

  - Tag: por defecto, latest
  - Alias: Útil cuando se trabaja con múltiples etapas de construcción. Ejemplo.

- Seleccionar una imagen base adecuada para tu aplicación puede simplificar mucho el desarrollo.
- Se recomienda utilizar imágenes base oficiales y de confianza, así como mantenerlas actualizadas para evitar vulnerabilidades de seguridad.

3. Etiquetas: LABEL

- Se utiliza para añadir metadatos que documenten y faciliten el mantenimiento de la imagen.

```
LABEL clave=valor clave=valor ...
```

- Aunque puedes definir cualquier etiqueta que desee, Docker recomienda: mantainer, description, versión y vendor (proveedor).
- Se pueden usar variables de entorno:

```
LABEL build_date=$BUILD_DATE
```

- Los metadatos son visibles con el comando:

```
docker inspect <image>
```

3. Etiquetas: COPY, ADD, RUN y WORKDIR

- **COPY**: Para copiar ficheros desde mi equipo a la imagen. Esos ficheros deben estar en el mismo contexto (carpeta o repositorio).
- **ADD**: Es similar a COPY pero tiene funcionalidades adicionales:
  - permite especificar una URL como fuente
  - descomprimir automáticamente los archivos comprimidos

ADD/COPY [--chown=<usuario>:<grupo>] <fuente> <destino>

- **RUN**: Ejecuta una orden creando una nueva capa.

RUN orden

  - Importante: Durante el proceso de construcción no puede haber interacción con el usuario (apt install -y).
- **WORKDIR**: Establece el directorio de trabajo dentro de la imagen -> todas las instrucciones se ejecutarán en ese directorio.

4. Etiquetas: USER, EXPOSE, ENV

- **USER**: Para especificar (por nombre o UID/GID) el usuario de trabajo para todas las órdenes posteriores.
- **EXPOSE**: No publica realmente los puertos. Nos da información acerca de qué puertos deberá tener abiertos el contenedor.
- **ENV**: Para establecer variables de entorno dentro del contenedor.
  - Se pueden sobrescribir en tiempo de ejecución.

5. Etiquetas: ENTRYPOINT, CMD

Nos permite definir qué comando se ejecuta cuando un contenedor se inicia.

- **CMD**: CMD [”executable”, ”param1", "param2"]
  - Proporciona valores por defecto para la ejecución del contenedor.
  - Se puede sobrescribir cuando haces docker run.

- **ENTRYPOINT**: ENTRYPOINT ["executable", "param1", "param2"]
  - Define el comando que siempre se va a ejecutar cuando arranca el contenedor.
  - Ideal cuando quieres que el contendor siempre ejecute algo concreto (por ejemplo, un script).

5. Etiquetas: ENTRYPOINT, CMD

```
FROM ubuntu:24.04
ENTRYPOINT ["echo"]
CMD [”Hola mundo"]
```

docker build example1 .

1. docker run example1?
2. docker run example1 “Jose”?

5. Etiquetas: ENTRYPOINT, CMD

```
FROM python
WORKDIR /app
COPY script.py /app/script.py
ENTRYPOINT [”python”, “script.py”]
CMD [”arg1"]
```

docker build example2 .

1. docker run example2?
2. docker run example2 arg_modificado

4. Build Context

4. Build context

```
docker build –t mi-imagen .
```

Docker build usa el argumento final “.” como ruta del build context.

1. Comprime todo ese directorio en un tarball.
2. Envía ese tarball al Docker daemon.
3. El docker daemon solo puede acceder a archivos dentro del build context.

4. Build context

Dos formas de controlar el build context:

1. .dockerignore: indicar qué no incluir en el contexto

```
# .dockerignore
*.log
*.env
node_modules/
.git/
```

2. En vez de usar . usar un subidrectorio ./src

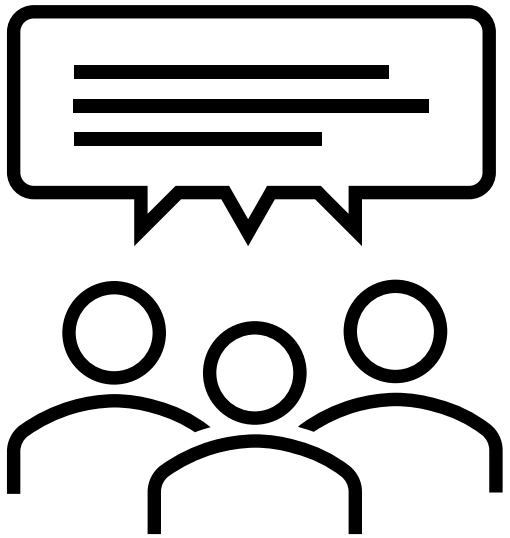
```
docker build -t mi-imagen ./src
```

4. Build context

Estructura de directorios de un proyecto Docker:

```
mi-proyecto/
├── Dockerfile
├── .dockerignore
├── src/
│   └── app.py
└── credenciales.env
```

docker build -t mi-imagen ./src



¿Preguntas?

Image Security

Buenas prácticas

1. Utilizar imágenes mínimas y de confianza

- Utilizar imágenes mínimas (como las variantes basadas en Alpine) siempre que sea posible: Estas deberían contener menos paquetes, lo que reduce su superficie de ataque.

- Puedes encontrar imágenes de confianza filtrando mediante las opciones “*Docker Official Image*” y “*Verified Publisher*” en Docker Hub.

2. Reconstruir regularmente las imágenes.

- Reconstruye regularmente tus imágenes para asegurarte de que incluyen paquetes y dependencias actualizados.

- Las imágenes construidas son inmutables, por lo que las correcciones de errores de paquetes y los parches de seguridad publicados después de su construcción no llegarán a sus contenedores en ejecución.

- Puedes automatizar el proceso de construcción de contenedores utilizando una herramienta como Watchtower.

3. Utilizar escáneres de vulnerabilidad de imágenes

- Existen herramientas capaces de identificar qué paquetes está utilizando la imagen, si contienen alguna vulnerabilidad y cómo solucionar el problema.

- Es una buena idea incluir estas herramientas en la CI/CD.

4. Docker content trust

- Docker Content Trust es un mecanismo para firmar y verificar imágenes.

- Los creadores de la imagen pueden firmarla para demostrar que son de su autoría; los consumidores que extraen imágenes pueden verificar la confianza comparando la firma pública de la imagen.

- Con este mecanismo, antes de iniciar un contenedor, puedes comprobar que un atacante no ha subido contenido malicioso o interceptado la descarga de la imagen.

5. Lint Dockerfiles

- Los comprobadores de código como Hadolint comprueban las instrucciones de tu Dockerfile y señalan cualquier problema que contravenga las mejores prácticas.

- Corregir los problemas detectados antes de la construcción de la imagen ayudará a garantizar que tus imágenes sean seguras y fiables. Este es otro proceso que puedes incorporar a los procesos de CI.