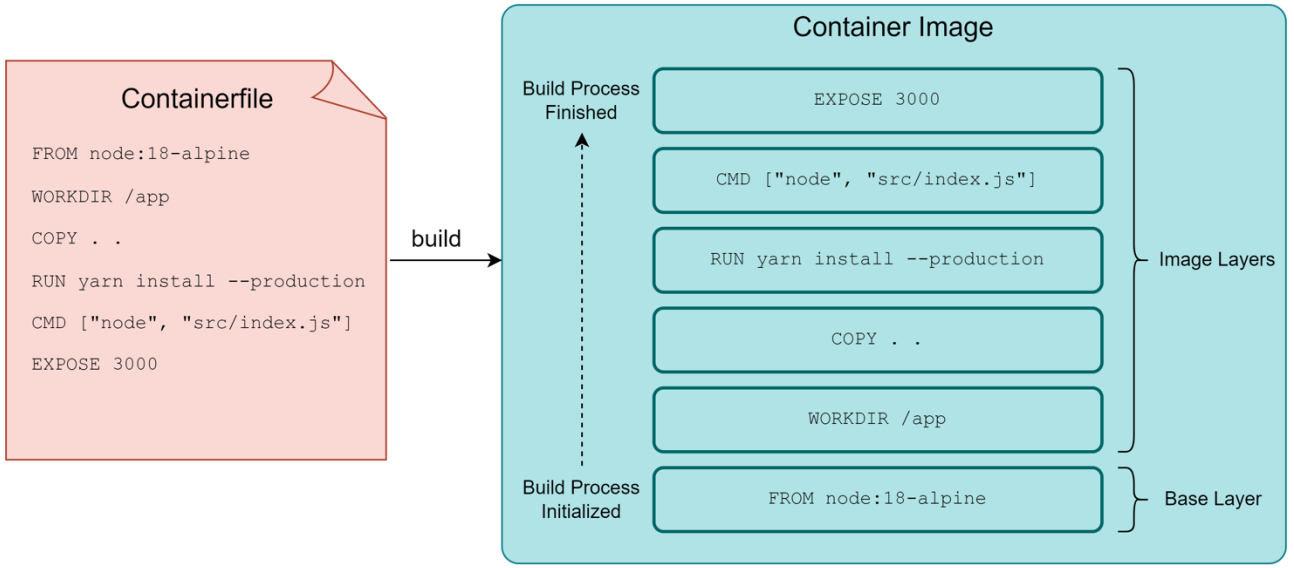
MÓDULO 3

Imágenes de Contenedores Personalizadas

REUTILIZACIÓN DE CONTAINERFILES EXISTENTES

- Los archivos Containerfile son otra opción para crear imágenes de contenedor, de manera automatizada, abordando las limitaciones que tenemos con la línea de comandos.
- Los archivos Containerfiles son fáciles de compartir, controlar, reutilizar y ampliar.
- Los Containerfile también facilitan la extensión de una imagen, llamada imagen secundaria, desde otra imagen, llamada imagen principal.
- Una imagen secundaria incorpora todo en la imagen principal y todos los cambios y adiciones realizados para crearla.



¿QUÉ ES UN ARCHIVO CONTAINERFILE?

- Containerfile es un archivo de texto que contiene instrucciones sobre cómo crear una imagen .
- Estos comandos se conocen como **directivas** .
- A Containerfile es un mecanismo que utilizamos para crear una imágenes personalizada según nuestros requisitos.
- El formato de a Containerfile es el siguiente:

```
# This is a comment
DIRECTIVE argument
```

CREACIÓN DE IMÁGENES DE CONTENEDOR PERSONALIZADAS CON CONTAINERFILES

- Un Containerfile es un mecanismo para automatizar la creación de imágenes de contenedor.
- Crear una imagen a partir de un Containerfile es un proceso de tres pasos:
  - Crea un directorio de trabajo.
  - Escribe el Containerfile.
  - Crea la imagen con Podman.

TRABAJAR CON RED HAT SOFTWARE COLLECTIONS LIBRARY (RHSCL)

- Al construir imágenes, cada instrucción del Containerfile es procesado generalmente como una nueva capa.

- A través de una buena planificación y orden de instrucciones dentro del archivo Docker en uso, las capas inferiores se pueden compartir entre muchas imágenes finales diferentes.

- Esto puede significar un ahorro significativo de espacio en disco para el almacenamiento imágenes de hosts Docker.

File: Dockerfile

```
FROM ubuntu:15.04
MAINTAINER Bryan Croft <bcroft@example.com>
RUN apt-get update && \
    apt-get -y install python
COPY requirements.txt /tmp
RUN pip install /tmp/requirements.txt
COPY . /code
EXPOSE 80
CMD ["python", "/code/app.py"]
```

ESPECIFICACIÓN DE CONTAINERFILE

- Un Containerfile es un archivo de texto que debe existir en el directorio de trabajo.
- Este archivo contiene las directivas o instrucciones necesarias para crear la imagen.
- La sintaxis básica de un Containerfile es la siguiente:

```
# Comment
INSTRUCTION arguments
```

- La primera instrucción sin comentarios debe ser una instrucción FROM para especificar la imagen base.

DIRECTIVAS : FROM

- A Containerfile suele comenzar con la FROM directiva.
- Esto se utiliza para especificar la imagen principal de nuestra imagen Docker personalizada. La imagen principal es el punto de partida de nuestra imagen Docker personalizada.
- Toda la personalización que hagamos se aplicará encima de la imagen principal.

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

```
FROM ubuntu:20.04
```

DIRECTIVAS : LABEL

- LABEL es un par clave-valor que se puede utilizar para agregar metadatos a una imagen de Docker.
- Estas etiquetas se pueden utilizar para organizar las imágenes de Docker correctamente.

```
LABEL maintainer=sathsara@mydomain.com
LABEL version=1.0
LABEL environment=dev
```

DIRECTIVAS: CMD & ENTRYPOINT

- La instrucción **CMD** se usa principalmente para definir el comando predeterminado cuando se ejecuta un contenedor sin especificar un comando.
- Podman puede anular la instrucción CMD al iniciar un contenedor. Si está presente, todos los parámetros para el comando ***podman run*** después del nombre de la imagen forman la instrucción CMD.

```
ENTRYPOINT ["/bin/date", "+%H:%M"]
```

- El **ENTRYPOINT** define tanto el comando a ejecutar como los parámetros. Por tanto, no se puede utilizar la instrucción CMD. El siguiente ejemplo proporciona la misma funcionalidad, con el beneficio adicional de que la instrucción CMD se puede sobrescribir cuando se inicia un contenedor:

```
ENTRYPOINT ["/bin/date"]
CMD ["+%H:%M"]
```

- En ambos casos, cuando un contenedor se inicia sin proporcionar un parámetro, se muestra la hora ac

```
[student@workstation ~]$ sudo podman run -it do180/rhel
11:41
```

DIRECTIVAS : ADD & COPY

- La instrucción **COPY** es la forma preferida de obtener archivos del sistema local en una imagen. Hay dos formas básicas:
  - Formulario de texto simple: ***COPY <src> ... <dest>***
  - Forma de matriz JSON: ***COPY ["<src>", ... "<dest>"])***
- Permite copiar una ruta o nombre de archivo que contenga espacios en blanco.
- Los archivos de origen deben estar dentro del directorio de compilación.
- Hay una superposición entre las instrucciones **COPY** y **ADD**.
- En general, se debe usar COPY, si es posible, reservando ADD para casos donde se necesite su característica especial.

File: Dockerfile

```
FROM scratch
ADD rootfs.tar.xz /
ADD http://example.com/proj1/app1 /code
ADD http://example.com/proj1/data1.gz /data
```

WORKDIR & ENV

- La instrucción **ENV** permite definir variables de entorno (y exportado) dentro de contenedores basados en esta imagen.
- Cuando se lanzan contenedores, se pueden configurar variables adicionales o variables definidas por ENV .
- ENV puede establecer múltiples variables en la misma línea (lo que resulta en una sola capa) cuando se usa con la tecla ENV = valor.
- La instrucción **WORKDIR** ofrece una manera de establecer el directorio de trabajo para instrucciones

File: Dockerfile

```
ENV DATASET=/data1
WORKDIR $DATASET/
RUN cp * /var/lib/data/appl
```

DIRECTIVAS : RUN

- RUN se utiliza para ejecutar comandos durante el tiempo de creación de la imagen.
- Esto creará una nueva capa encima de la capa existente, ejecutará el comando especificado y enviará los resultados a la capa recién creada.
- RUN se puede utilizar para instalar los paquetes necesarios, actualizar los paquetes, crear usuarios y grupos, etc.

```
RUN apt-get update
RUN apt-get install nginx -y
```

- Puede agregar varios comandos de shell a una sola RUN Separándolos con el && símbolo.

```
RUN apt-get update && apt-get install nginx -y
```

DIRECTIVAS : HEALTCHECK

- Los controles de estado se utilizan en Docker para comprobar si los contenedores se están ejecutando correctamente.

- Podemos utilizar la siguiente directiva para garantizar que el contenedor pueda recibir tráfico en el http://localhost/ punto final:

```
HEALTHCHECK CMD curl -f http://localhost/ || exit 1
```

EJEMPLO CONTAINERFILE

```
# This is a comment line (1)
FROM ubi7/ubi:7.7 (2)
LABEL description="This is a custom httpd container image" (3)
MAINTAINER John Doe <jdoe@xyz.com> (4)
RUN yum install -y httpd (5)
EXPOSE 80 (6)
ENV LogLevel "info" (7)
ADD http://someserver.com/filename.pdf /var/www/html (8)
COPY ./src/ /var/www/html/ (9)
USER apache (10)
ENTRYPOINT ["/usr/sbin/httpd"] (11)
CMD ["-D", "FOREGROUND"] (12)
```

1. Lines that begin with a hash, or pound, sign (#) are comments.
2. The **FROM** instruction declares that the new container image extends **ubi7/ubi:7.7** container base image. Dockerfiles can use any other container image as a base image, not only images from operating system distributions. Red Hat provides a set of container images that are certified and tested and highly recommends using these container images as a base.
3. The **LABEL** is responsible for adding generic metadata to an image. A **LABEL** is a simple key-value pair.
4. **MAINTAINER** indicates the **Author** field of the generated container image's metadata. You can use the **podman inspect** command to view image metadata.
5. **RUN** executes commands in a new layer on top of the current image. The shell that is used to execute commands is **/bin/sh**.
6. **EXPOSE** indicates that the container listens on the specified network port at runtime. The **EXPOSE** instruction defines metadata only; it does not make ports accessible from the host. The **-p** option in the **podman run** command exposes container ports from the host.
7. **ENV** is responsible for defining environment variables that are available in the container. You can declare multiple **ENV** instructions within the **Dockerfile**. You can use the **env** command inside the container to view each of the environment variables.
8. **ADD** instruction copies files or folders from a local or remote source and adds them to the container's file system. If used to copy local files, those must be in the working directory. **ADD** instruction unpacks local **.tar** files to the destination image directory.
9. **COPY** copies files from the working directory and adds them to the container's file system. It is not possible to copy a remote file using its URL with this Dockerfile instruction.
10. **USER** specifies the username or the UID to use when running the container image for the **RUN**, **CMD**, and **ENTRYPOINT** instructions. It is a good practice to define a different user other than **root** for security reasons.
11. **ENTRYPOINT** specifies the default command to execute when the image runs in a container. If omitted, the default **ENTRYPOINT** is **/bin/sh -c**.
12. **CMD** provides the default arguments for the **ENTRYPOINT** instruction. If the default **ENTRYPOINT** applies (**/bin/sh -c**), then **CMD** forms an executable command and parameters that run at container start.

CONSTRUCIÓN DE IMÁGENES CON PODMAN

- El comando **podman build** procesa el Containerfile y crea una nueva imagen basada en las instrucciones que contiene.

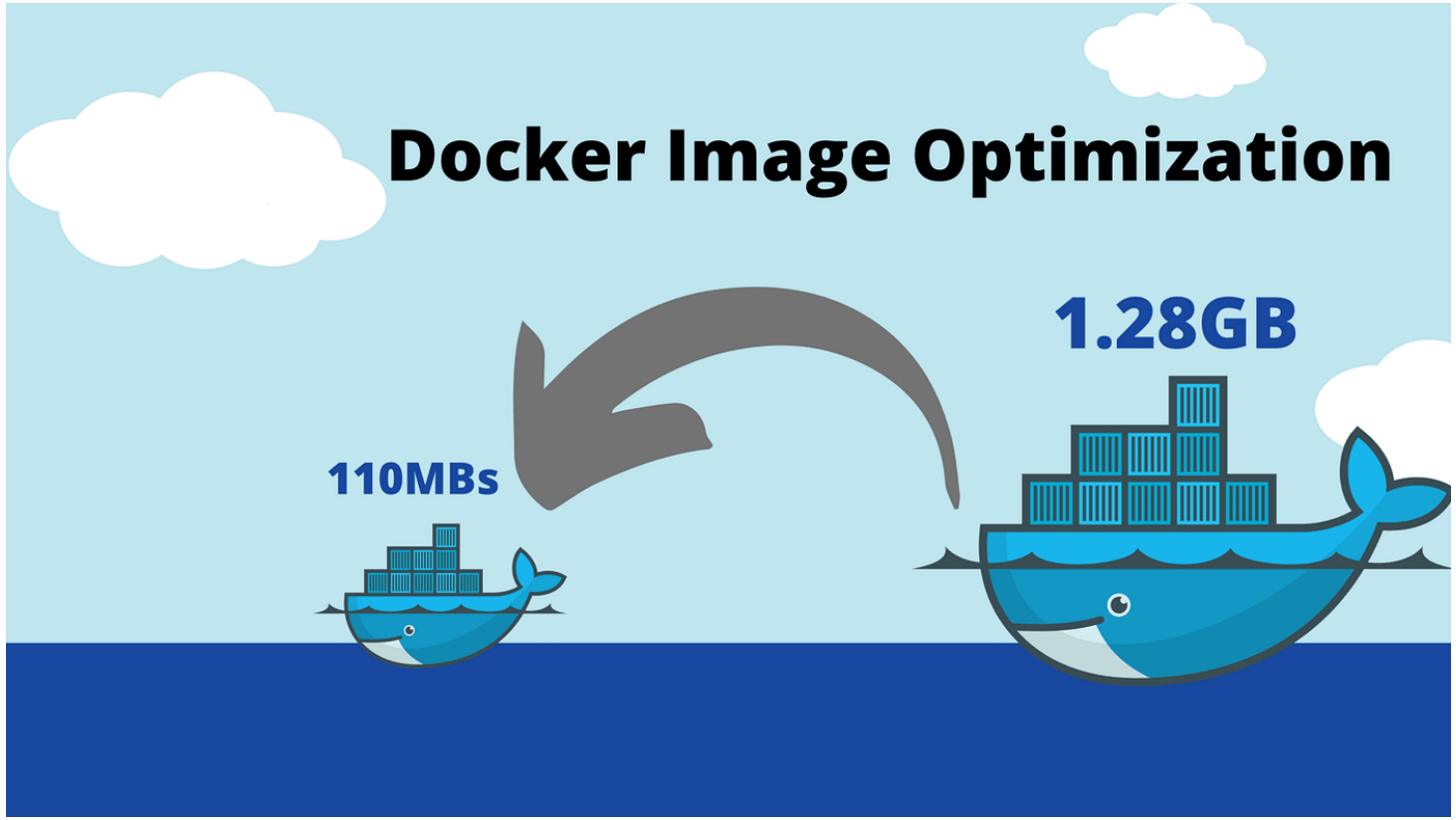
- La sintaxis de este comando es la siguiente:

```
$ podman build -t NAME:TAG DIR
```

CONSTRUCCIONES NORMALES DE DOCKER

CONSTRUCCIONES NORMALES DE DOCKER

- Con Docker, podemos utilizarlo Dockerfiles para crear imágenes de Docker personalizadas.
- Es fundamental tener imágenes de Docker de tamaño mínimo al ejecutarlas en entornos de producción.



EJEMPLO DE PROBLEMAS DE DOCKERFILE

1- Consideremos un ejemplo en el que creamos una aplicación Golang sencilla.
2- Vamos a implementar una hello worldaplicación escrita en Golang utilizando lo siguiente Dockerfile:

```
# Start from latest golang parent image
FROM golang:latest
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

EJEMPLO DE PROBLEMAS DE DOCKERFILE

3- El siguiente es el contenido del helloworld.goarchivo. Es un archivo simple que imprimirá el texto "Hello World"cuando se ejecute:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello World")
}
```

4- Una vez que Dockerfileesté listo, podemos crear la imagen de Docker mediante el docker image buildcomando. Esta imagen se etiquetará como helloworld:v1:

```
$ docker image build -t helloworld:v1 .
```

5- La imagen creada con el docker image ls comando.

```
REPOSITORY   TAG   IMAGE ID       CREATED          SIZE
helloworld   v1    23874f841e3e   10 seconds ago   805MB
```

PATRÓN BUILDER

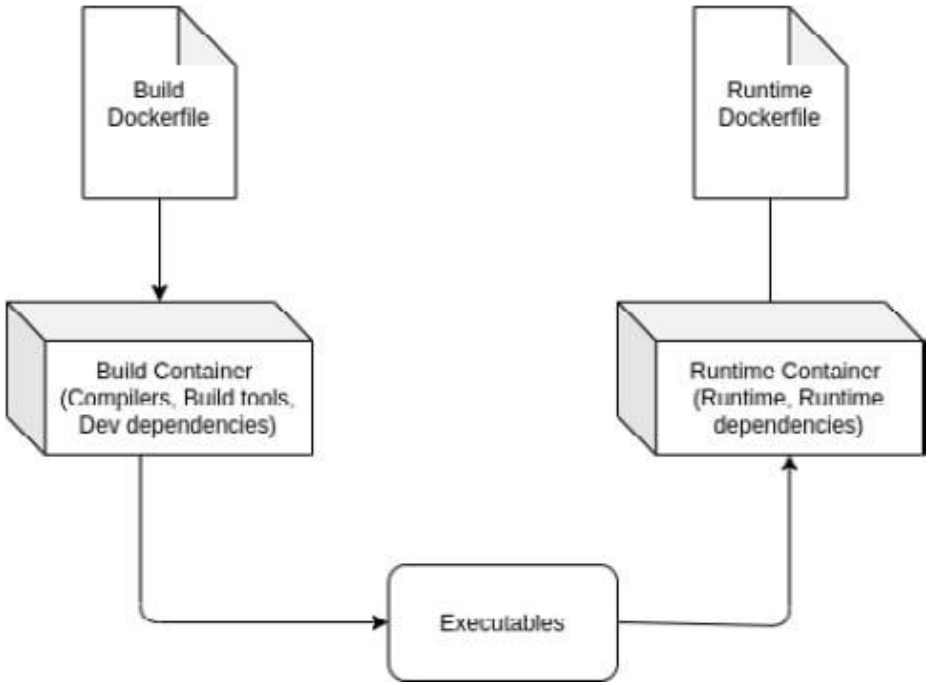
¿QUÉ ES EL PATRON BUILDER?

- El **patrón de compilación** es un método que se utiliza para crear imágenes de Docker de tamaño óptimo.

- Todo el proceso de construcción de la imagen utilizando el patrón constructor consta de los siguientes pasos:

1.- Crea la Buildimagen de Docker.

2.- Crea un contenedor a partir de la Buildimagen de Docker.

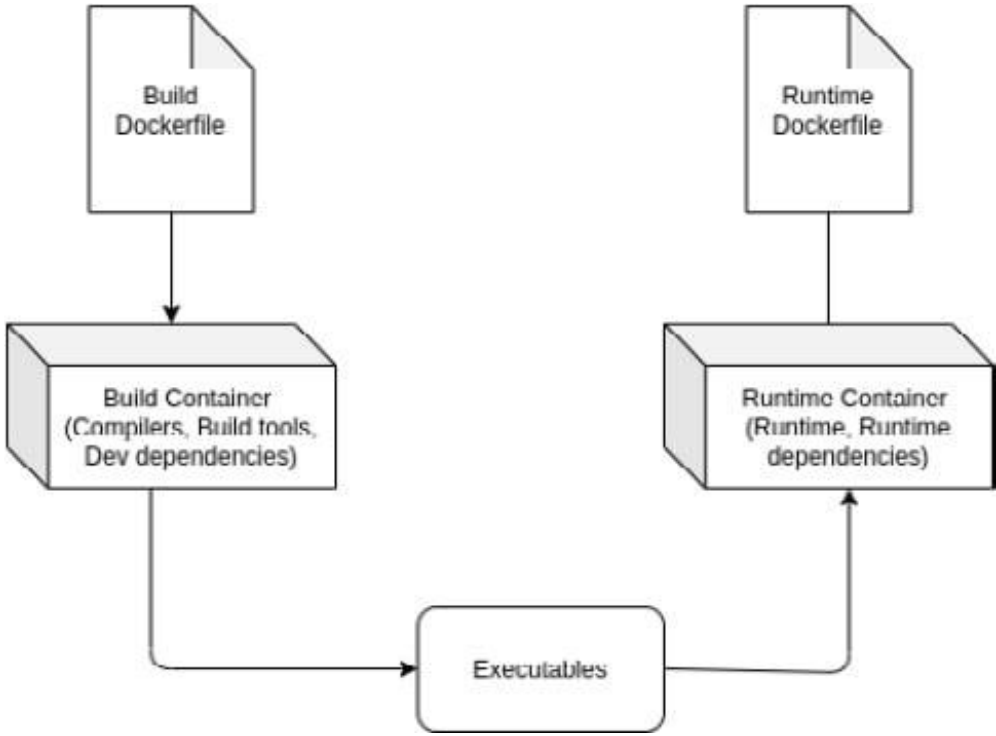


CREACIÓN DEL PRIMER DOCKERFILE

- Esto es lo mismo Dockerfile que observamos con las compilaciones normales de Docker.

- Esto se utilizó para crear el helloworld ejecutable a partir del helloworld.go archivo de origen.

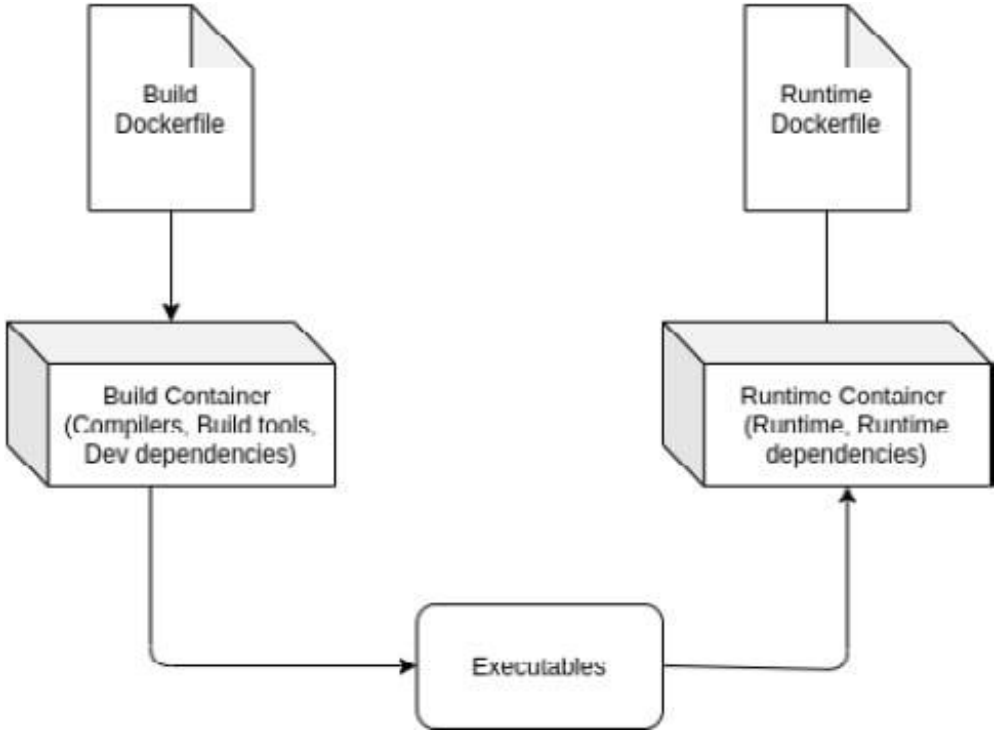
```
# Start from latest golang parent image
FROM golang:latest
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```



CREACIÓN DEL SEGUNDO DOCKERFILE

- El siguiente es el segundo Dockerfile utilizado para construir el Runtimecontenedor Docker:

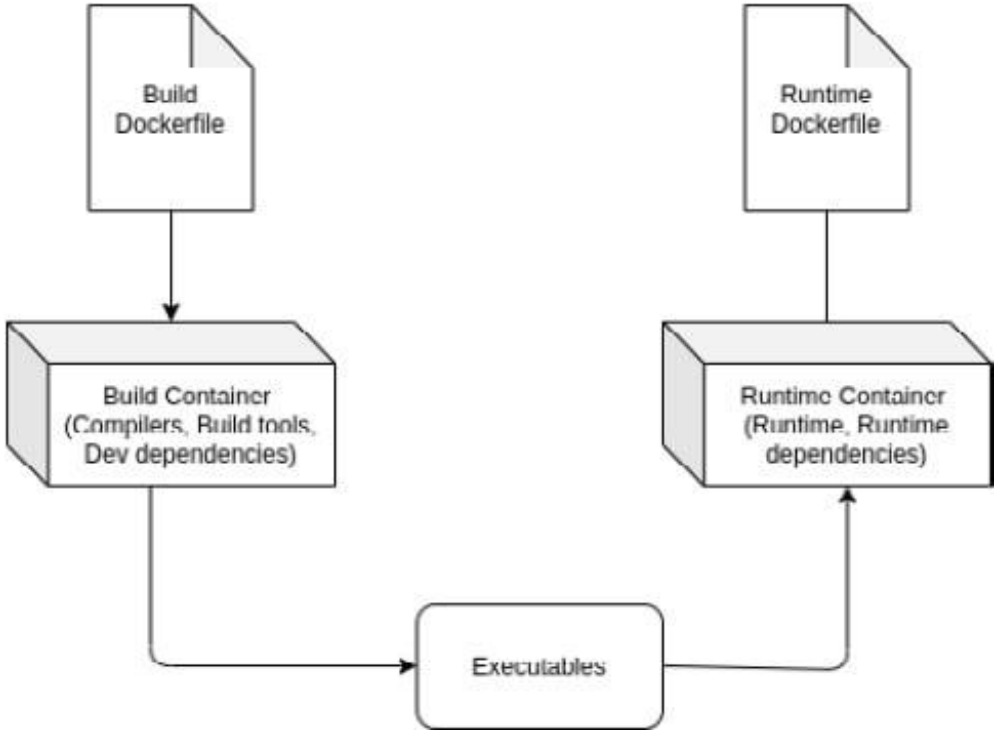
```
# Start from latest alpine parent image
FROM alpine:latest
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```



CREACIÓN DEL SCRIPT

- Considerar el siguiente script de shell utilizado para copiar los artefactos de compilación entre contenedores Docker:

```
#!/bin/sh
# Build the builder Docker image
docker image build -t helloworld-build -f Dockerfile.build .
# Create container from the build Docker image
docker container create --name helloworld-build-container   helloworld-build
# Copy build artifacts from build container to the local filesystem
docker container cp helloworld-build-container:/myapp/helloworld .
# Build the runtime Docker image
docker image build -t helloworld .
# Remove the build Docker container
docker container rm -f helloworld-build-container
# Remove the copied artifact
rm helloworld
```



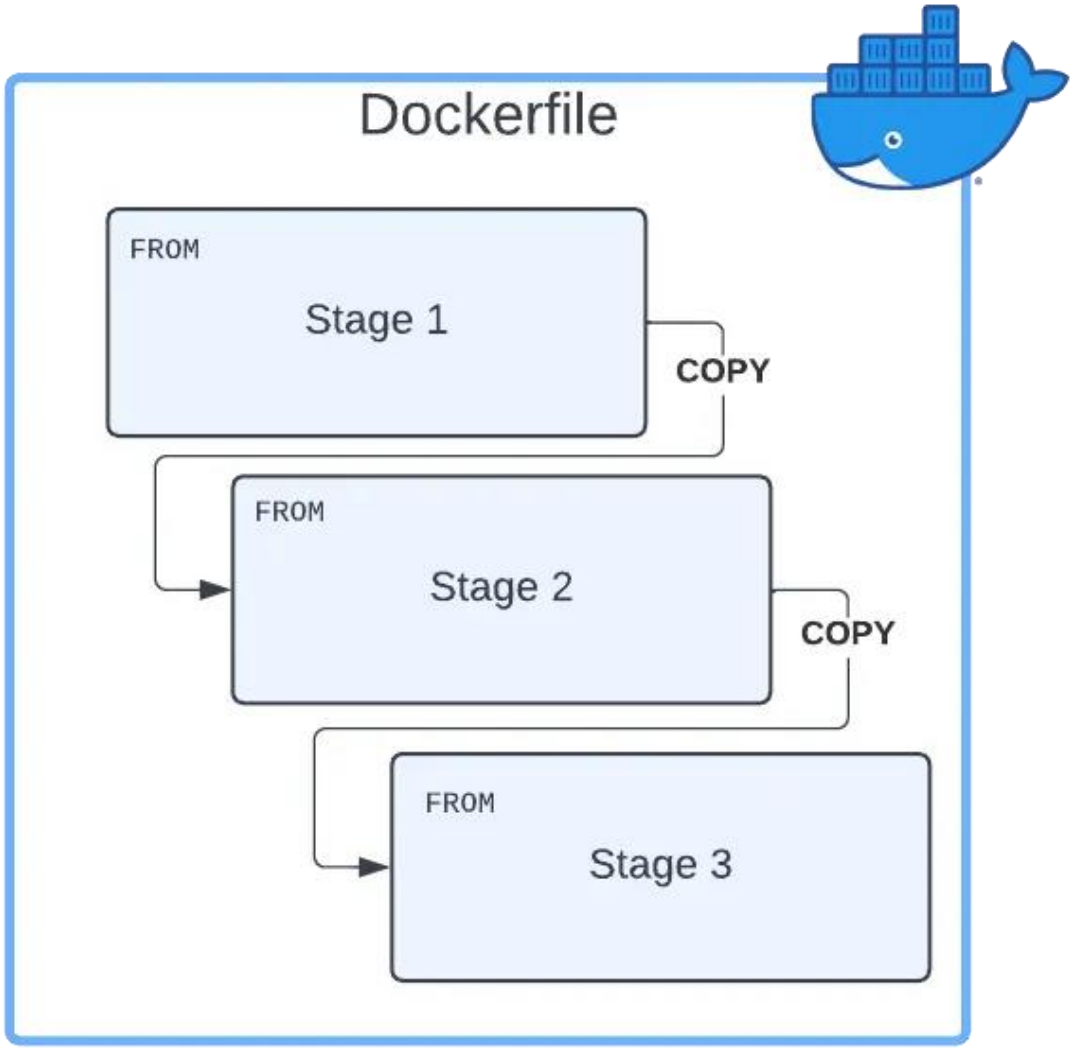
- Una vez que ejecutamos el script de shell, deberíamos poder ver dos imágenes de Docker:

```
REPOSITORY          TAG       IMAGE ID       CREATED         SIZE
helloworld          latest    faff247e2b35   3 hours ago     7.6MB
helloworld-build    latest    f8c10c5bd28d   3 hours ago     805MB
```

DOCKERFILE MULTI-STAGE

MULTI-STAGE

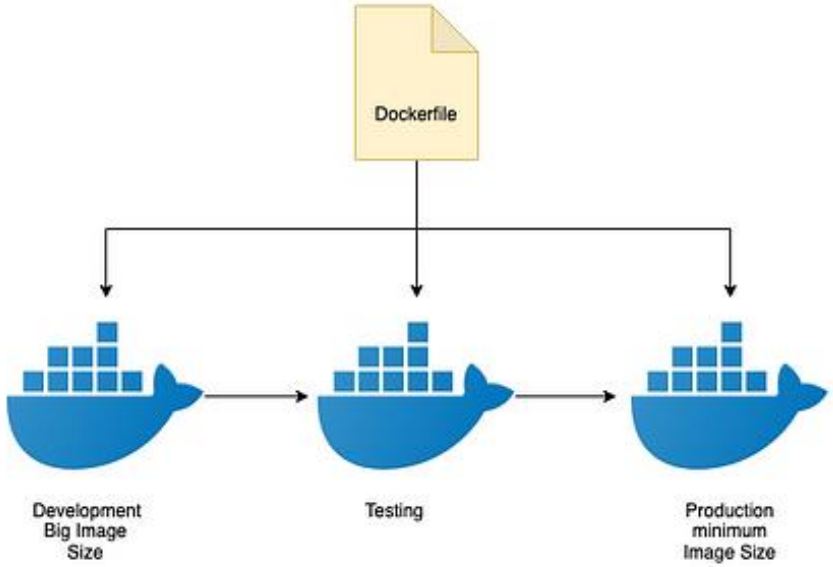
- Un enfoque para mantener pequeñas las imágenes de Docker es usar compilaciones en varias etapas.
- Una compilación en varias etapas le permite usar múltiples imágenes para compilar un producto final.
- En una compilación en varias etapas, tiene un solo Dockerfile, pero puede definir múltiples imágenes dentro de él para ayudar a compilar la imagen final



CONSTRUIR EN MULTI-STAGE (MULTI-ETAPA)

- Con compilaciones de varias etapas, se utilizan varias FROM en el Dockerfile.
- Cada FROM puede utilizar una base diferente y cada una de ellas inicia una nueva etapa de la compilación
- Puede copiar artefactos de forma selectiva de una etapa a otra y dejar todo lo que no desea en la imagen final.
- En la instrucción anterior

```
1  # Stage - Development
2  FROM Dev-Image as dev
3  . . .          stage 0
4  . . .
5
6  # Stage - Testing
7  FROM Test-Image as test
8  COPY --from=dev /src/app .
9  . . .          stage 1
10 . . .
11
12 # Stage - Production
13 FROM Minimum-Image as prod
14 COPY --from=dev /src/app .
15 . . .          stage 2
16 . . .
```



DORKEFILE MULTI-STAGE

- La estructura de un sistema multietapa Dockerfile:

```
# Start from latest golang parent image
FROM golang:latest
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Start from latest alpine parent image
FROM alpine:latest
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY --from=0 /myapp/helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

CREACIÓN IMAGEN MULTI-STAGE

1.- Creamos la imagen de Docker y se etiqueta como multi-stage:v1

```
docker image build -t multi-stage:v1 .
```

2.- Enumeramos las imágenes de Docker disponibles:

```
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
multi-stage   latest    75e1f4bcabd0   7 seconds ago    7.6MB
```

CREACIÓN DEL PRIMER DOCKERFILE

- Se nombra la primera etapa builder y la segunda etapa runtime

```
FROM golang:latest AS builder
FROM alpine:latest AS runtime
```

- Se Copia los artefactos en la segunda etapa

```
COPY --from=builder /myapp/helloworld .
```

- Puedes usar el --target indicador con el docker build comando para especificar una etapa intermedia como la etapa

```
docker image build --target builder -t multi-stage-dev:v1 .
```

```
# Start from latest golang parent image
FROM golang:latest AS builder
# Set the working directory
WORKDIR /myapp
# Copy source file from current directory to container
COPY helloworld.go .
# Build the application
RUN go build -o helloworld .
# Start from latest alpine parent image
FROM alpine:latest AS runtime
# Set the working directory
WORKDIR /myapp
# Copy helloworld app from current directory to container
COPY --from=builder /myapp/helloworld .
# Run the application
ENTRYPOINT ["./helloworld"]
```

¿CÓMO SE DEFINE UN DOCKERFILE DE VARIAS ETAPAS?

- En primer lugar, es necesario tener un Dockerfile con más de un FROM

```
FROM eclipse-temurin:11-jre-alpine AS builder
```

- Un Dockerfile de varias etapas será algo como esto:

```
FROM eclipse-temurin:11-jre-alpine AS builder
LABEL stage=builder
COPY . /
RUN apk add  --no-cache unzip zip && zip -qq -d /resources/bwce-runtime/bwce-runtime-2.7.2.zip "tibco.home/tibcojre64/*"
RUN unzip -qq /resources/bwce-runtime/bwce*.zip -d /tmp && rm -rf /resources/bwce-runtime/bwce*.zip 2> /dev/null

FROM  eclipse-temurin:11-jre-alpine
RUN addgroup -S bwcegroup && adduser -S bwce -G bwcegroup
```

COPIAR RECURSOS DE UNA ETAPA A OTRA

- **COPY** es el mismo comando que utilizas para mover datos desde tu almacenamiento local a la imagen del contenedor.
- Una forma de diferenciar que esta vez no lo estás copiando desde tu almacenamiento local sino desde otra etapa, es con el argumento **--from**.

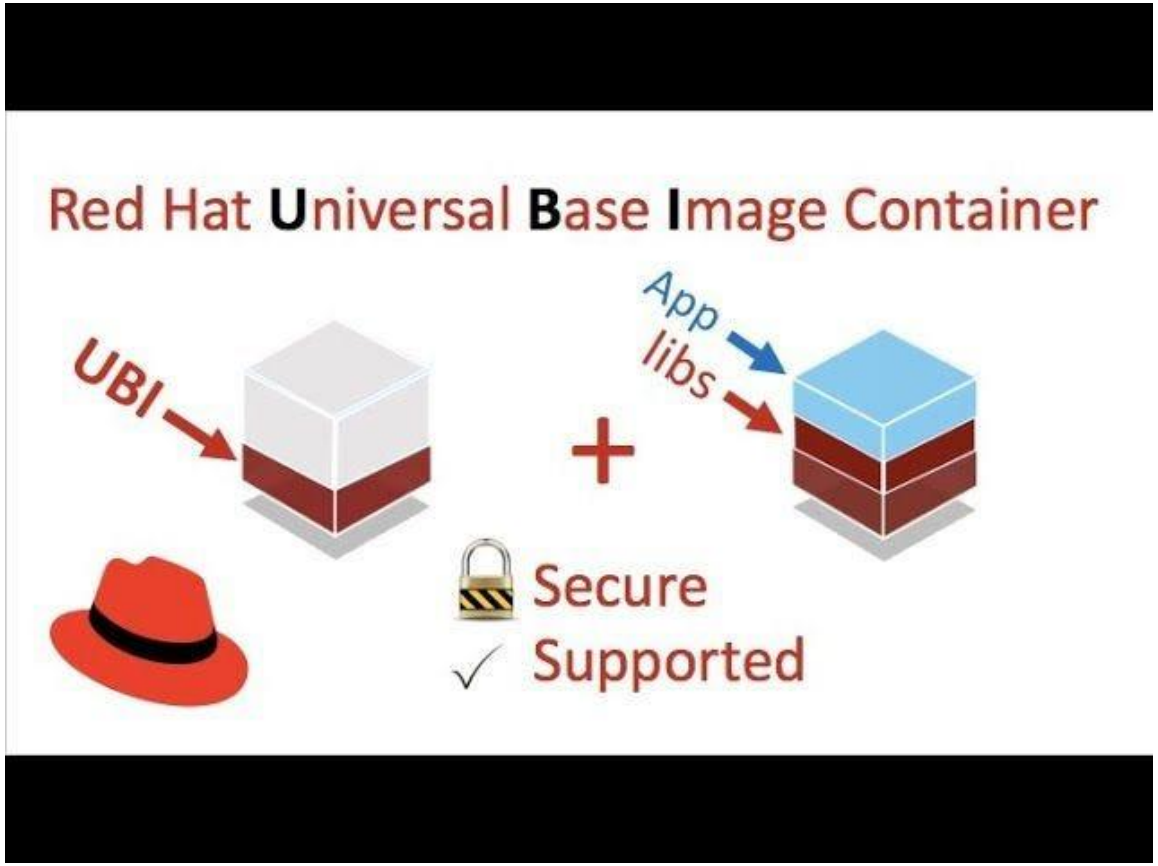
```
COPY --from=builder /resources/ /resources/
```

IMÁGENES DE CONTENEDOR PARA RED HAT OPENSHIFT

USO DE RED HAT UNIVERSAL BASE IMAGES

- Al definir imágenes de contenedor personalizadas, Red Hat recomienda usar la imagen base universal (UBI) de Red Hat como imagen de contenedor base para sus aplicaciones.

- Las imágenes UBI son imágenes certificadas, probadas y mantenidas con regularidad que Red Hat proporciona sin costo alguno



TIPOS DE IMÁGENES UBI

Red Hat proporciona cuatro tipos de imágenes UBI, diseñadas para cubrir la mayoría de los casos de uso:

Tipo de imagen	Nombre de imagen	Descripción
Estándar	`ubi`	Para la mayoría de las aplicaciones y casos de uso.
Init	`ubi-init`	Para contenedores que ejecutan varios servicios `systemd`.
Mínima	`ubi-minimal`	Imagen más pequeña para aplicaciones que gestionan sus propias dependencias y dependen de menos componentes del sistema operativo.
Micro	`ubi-micro`	La imagen más pequeña para casos de uso optimizados de espacio de memoria. Para aplicaciones que casi no usan componentes del sistema operativo.

IMÁGENES UBI DE TIEMPO DE EJECUCIÓN PARA DESARROLLADORES

- Además de las cuatro imágenes UBI principales, Red Hat proporciona imágenes UBI específicas para tiempos de ejecución populares.

- Para cada tiempo de ejecución, Red Hat proporciona imágenes para cada versión principal admitida del tiempo de ejecución.

- OpenJDK
- Node.js
- Python
- PHP
- .NET
- Go
- Ruby

OPTIMIZAR CONTAINERFILES PARA OPENSHIFT

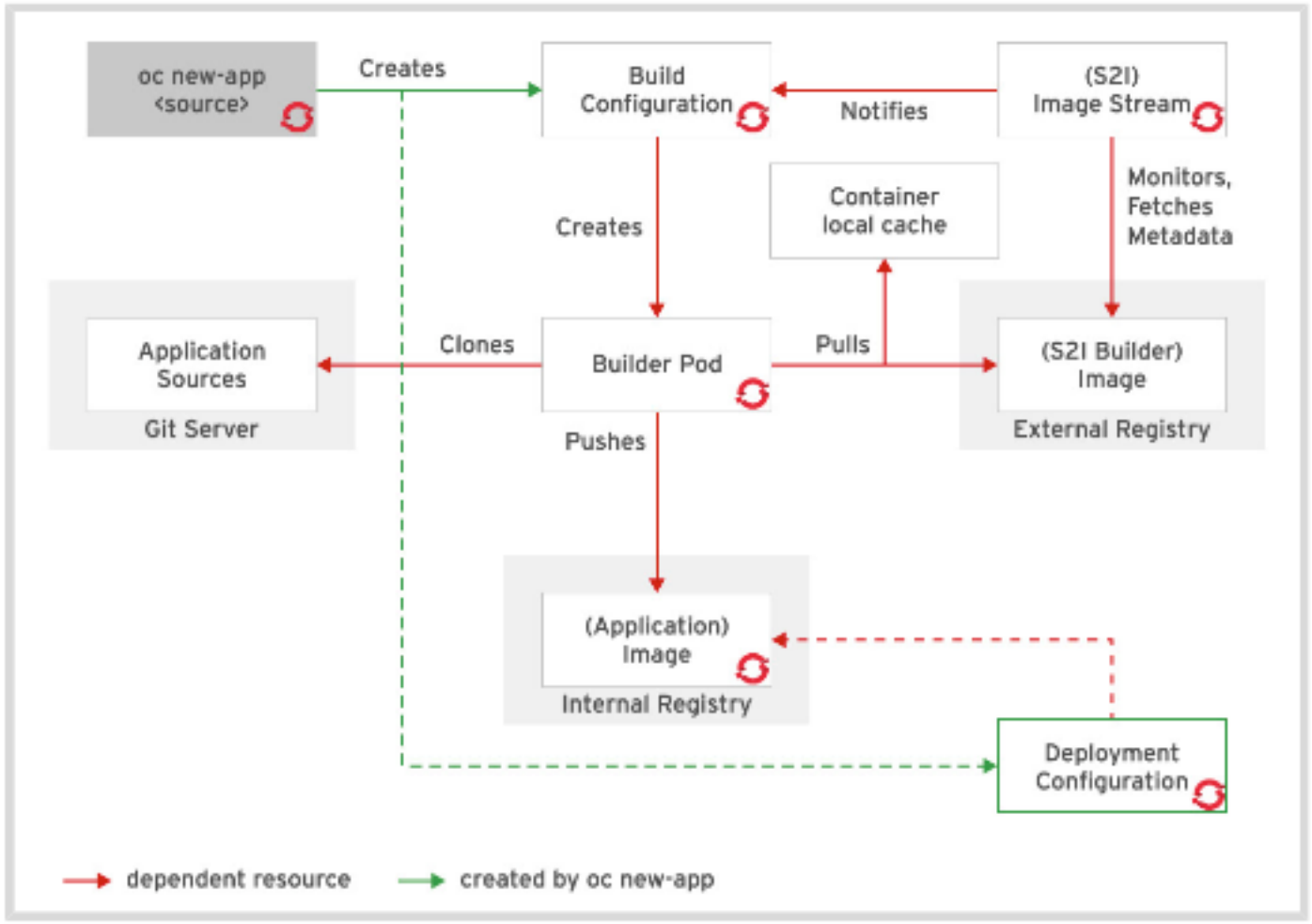
- Comience usando una imagen UBI en la instrucción FROM de su Containerfile.
- Las imágenes UBI están disponibles en el registro de contenedores público registry.access.redhat.com.
- El formato del nombre de la imagen es el siguiente:

```
registry.access.redhat.com/NAMESPACE/NAME[:TAG]
```

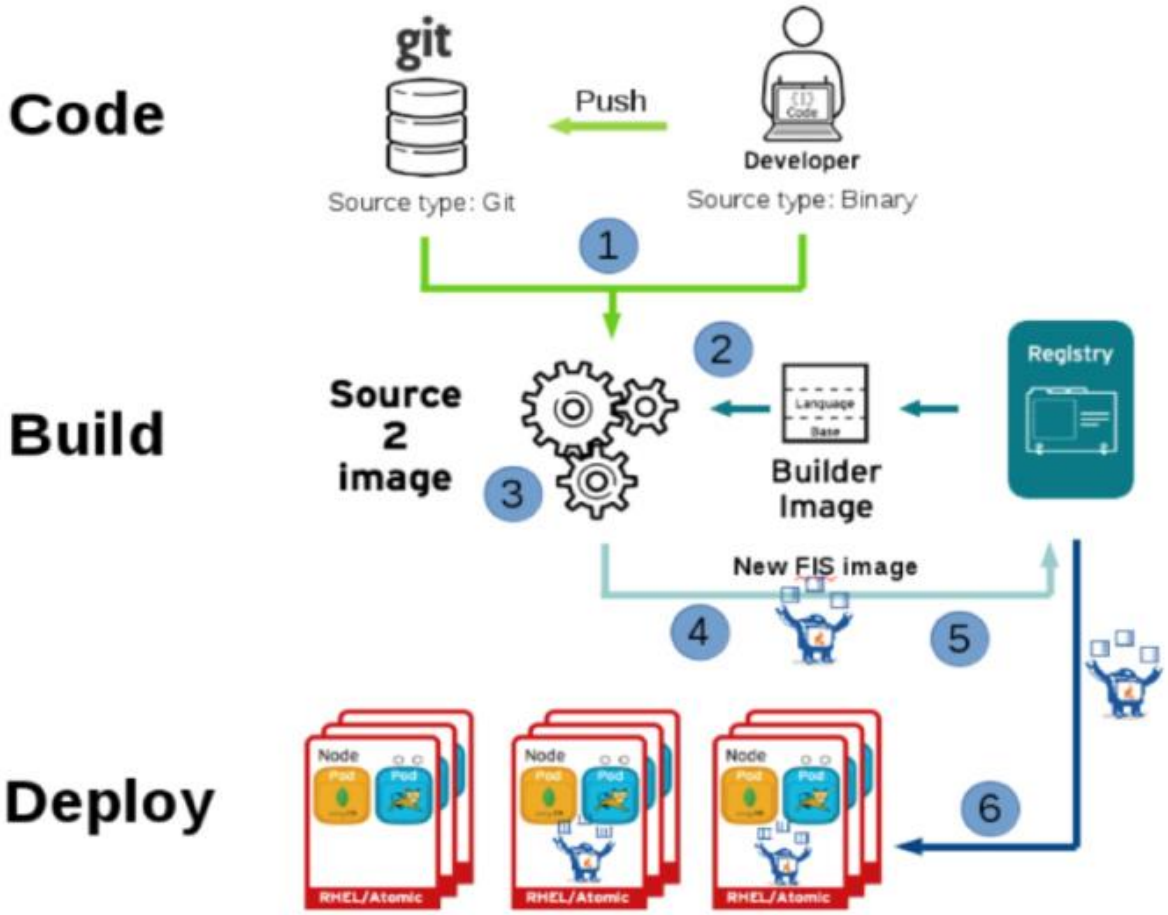
- La parte NAMESPACE especifica la versión de UBI que usa, como ubi8 o ubi9

```
FROM registry.access.redhat.com/ubi9/nodejs-18-minimal:1-56
```

CREACIÓN DE APLICACIONES CON SOURCE-TO-IMAGE



¿CÓMO PUEDE AYUDARTE EL SOURCE TO IMAGE (S2I)



LAB 3
CREACIÓN DE IMÁGENES PERSONALIZADAS