Multi-platform builds

A multi-platform build refers to a single build invocation that targets multiple different operating system or CPU architecture combinations. When building images, this lets you create a single image that can run on multiple platforms, such as `linux/amd64`, `linux/arm64`, and `windows/amd64`.

Why multi-platform builds?

Docker solves the "it works on my machine" problem by packaging applications and their dependencies into containers. This makes it easy to run the same application on different environments, such as development, testing, and production.

But containerization by itself only solves part of the problem. Containers share the host kernel, which means that the code that's running inside the container must be compatible with the host's architecture. This is why you can't run a `linux/amd64` container on an arm64 host (without using emulation), or a Windows container on a Linux host.

Multi-platform builds solve this problem by packaging multiple variants of the same application into a single image. This enables you to run the same image on different types of hardware, such as development machines running x86-64 or ARM-based Amazon EC2 instances in the cloud, without the need for emulation.

Difference between single-platform and multi-platform images

Multi-platform images have a different structure than single-platform images. Single-platform images contain a single manifest that points to a single configuration and a single set of layers. Multi-platform images contain a manifest list, pointing to multiple manifests, each of which points to a different configuration and set of layers.

When you push a multi-platform image to a registry, the registry stores the manifest list and all the individual manifests. When you pull the image, the registry returns the manifest list, and Docker automatically selects the correct variant based on the host's architecture. For example, if you run a multi-platform image on an ARM-based Raspberry Pi, Docker selects the `linux/arm64` variant. If you run the same image on an x86-64 laptop, Docker selects the `linux/amd64` variant (if you're using Linux containers).

Prerequisites

To build multi-platform images, you first need to make sure that your Docker environment is set up to support it. There are two ways you can do that:

- You can switch from the "classic" image store to the containerd image store.
- You can create and use a custom builder.

The "classic" image store of the Docker Engine does not support multi-platform images. Switching to the containerd image store ensures that your Docker Engine can push, pull, and build multi-platform images.

Creating a custom builder that uses a driver with multi-platform support, such as the `docker-container` driver, will let you build multi-platform images without switching to a different image store. However, you still won't be able to load the multi-platform images you build into your Docker Engine image store. But you can push them to a container registry directly with `docker build --push`.

containerd image store Custom builder

The steps for enabling the containerd image store depends on whether you're using Docker Desktop or Docker Engine standalone:

- If you're using Docker Desktop, enable the containerd image store in the Docker Desktop settings.
- If you're using Docker Engine standalone, enable the containerd image store using the daemon configuration file.

If you're using Docker Engine standalone and you need to build multi-platform images using emulation, you also need to install QEMU, see Install QEMU manually.

Build multi-platform images

When triggering a build, use the `--platform` flag to define the target platforms for the build output, such as `linux/amd64` and `linux/arm64`:

```
$ docker buildx build --platform linux/amd64,linux/arm64 .
```

Strategies

You can build multi-platform images using three different strategies, depending on your use case:

1. Using emulation, via QEMU
2. Use a builder with multiple native nodes

3. Use cross-compilation with multi-stage builds

QEMU

Building multi-platform images under emulation with QEMU is the easiest way to get started if your builder already supports it. Using emulation requires no changes to your Dockerfile, and BuildKit automatically detects the architectures that are available for emulation.

Note

Emulation with QEMU can be much slower than native builds, especially for compute-heavy tasks like compilation and compression or decompression.

Use multiple native nodes or cross-compilation instead, if possible.

Docker Desktop supports running and building multi-platform images under emulation by default. No configuration is necessary as the builder uses the QEMU that's bundled within the Docker Desktop VM.

Install QEMU manually

If you're using a builder outside of Docker Desktop, such as if you're using Docker Engine on Linux, or a custom remote builder, you need to install QEMU and register the executable types on the host OS. The prerequisites for installing QEMU are:

- Linux kernel version 4.8 or later
- `binfmt-support` version 2.1.7 or later
- The QEMU binaries must be statically compiled and registered with the `fix_binary` flag

Use the `tonistiigi/binfmt` image to install QEMU and register the executable types on the host with a single command:

```
$ docker run --privileged --rm tonistiigi/binfmt --install all
```

This installs the QEMU binaries and registers them with `binfmt_misc`, enabling QEMU to execute non-native file formats for emulation.

Once QEMU is installed and the executable types are registered on the host OS, they work transparently inside containers. You can verify your registration by checking if `F` is among the flags in `/proc/sys/fs/binfmt_misc/qemu-*`.

Multiple native nodes

Using multiple native nodes provide better support for more complicated cases that QEMU can't handle, and also provides better performance.

You can add additional nodes to a builder using the `--append` flag.
The following command creates a multi-node builder from Docker contexts named `node-amd64` and `node-arm64`. This example assumes that you've already added those contexts.

```
$ docker buildx create --use --name mybuild node-amd64
mybuild
$ docker buildx create --append --name mybuild node-arm64
$ docker buildx build --platform linux/amd64,linux/arm64 .
```

While this approach has advantages over emulation, managing multi-node builders introduces some overhead of setting up and managing builder clusters. Alternatively, you can use Docker Build Cloud, a service that provides managed multi-node builders on Docker's infrastructure. With Docker Build Cloud, you get native multi-platform ARM and X86 builders without the burden of maintaining them. Using cloud builders also provides additional benefits, such as a shared build cache.

After signing up for Docker Build Cloud, add the builder to your local environment and start building.

```
$ docker buildx create --driver cloud ORG/BUILDER_NAME
cloud-ORG-BUILDER_NAME
$ docker build \
  --builder cloud-ORG-BUILDER_NAME \
  --platform linux/amd64,linux/arm64,linux/arm/v7 \
  --tag IMAGE_NAME \
  --push .
```

For more information, see Docker Build Cloud.

Cross-compilation

Depending on your project, if the programming language you use has good support for cross-compilation, you can leverage multi-stage builds to build binaries for target platforms from the native architecture of the builder. Special build arguments, such as `BUILDPLATFORM` and `TARGETPLATFORM`, are automatically available for use in your Dockerfile.
In the following example, the `FROM` instruction is pinned to the native platform of the builder (using the `--platform=$BUILDPLATFORM` option) to prevent emulation from kicking in. Then the pre-defined `$BUILDPLATFORM` and `$TARGETPLATFORM` build arguments are interpolated in a `RUN` instruction. In this case, the values are just printed to stdout with `echo`, but this illustrates how you would pass them to the compiler for cross-compilation.

```
# syntax=docker/dockerfile:1
FROM --platform=$BUILDPLATFORM golang:alpine AS build
ARG TARGETPLATFORM
ARG BUILDPLATFORM
RUN echo "I am running on $BUILDPLATFORM, building for $TARGETPLATFORM"
> /log
FROM alpine
COPY --from=build /log /log
```

Examples

Here are some examples of multi-platform builds:

- Simple multi-platform build using emulation
- Multi-platform Neovim build using Docker Build Cloud
- Cross-compiling a Go application

Simple multi-platform build using emulation

This example demonstrates how to build a simple multi-platform image using emulation with QEMU. The image contains a single file that prints the architecture of the container.

Prerequisites:

- Docker Desktop, or Docker Engine with QEMU installed
- containerd image store enabled

Steps:

1. Create an empty directory and navigate to it:

2.
   ```
   $ mkdir multi-platform
   ```

3.
   ```
   $ cd multi-platform
   ```
4. Create a simple Dockerfile that prints the architecture of the container:

5.
   ```
   # syntax=docker/dockerfile:1
   ```

6.
   ```
   FROM alpine
   RUN uname -m > /arch
   ```
7. Build the image for `linux/amd64` and `linux/arm64`:

8.
   ```
   $ docker build --platform linux/amd64,linux/arm64 -t multi-
   platform .
   ```
9. Run the image and print the architecture:

10.
    ```
    $ docker run --rm multi-platform cat /arch
    ```
    - If you're running on an x86-64 machine, you should see `x86_64`.
    - If you're running on an ARM machine, you should see `aarch64`.

Multi-platform Neovim build using Docker Build Cloud

This example demonstrates how run a multi-platform build using Docker Build Cloud to compile and export Neovim binaries for the `linux/amd64` and `linux/arm64` platforms.

Docker Build Cloud provides managed multi-node builders that support native multi-platform builds without the need for emulation, making it much faster to do CPU-intensive tasks like compilation.

Prerequisites:

- You've signed up for Docker Build Cloud and created a builder

Steps:

1. Create an empty directory and navigate to it:

```
$ mkdir docker-build-neovim
$ cd docker-build-neovim
```

4. Create a Dockerfile that builds Neovim.

```
# syntax=docker/dockerfile:1
FROM debian:bookworm AS build
WORKDIR /work
RUN --mount=type=cache,target=/var/cache/apt,sharing=locked \
    --mount=type=cache,target=/var/lib/apt,sharing=locked \
     apt-get update && apt-get install -y \
     build-essential \
     cmake \
     curl \
     gettext \
     ninja-build \
     unzip
ADD https://github.com/neovim/neovim.git#stable .
RUN make CMAKE_BUILD_TYPE=RelWithDebInfo

FROM scratch
COPY --from=build /work/build/bin/nvim /
```

21. Build the image for `linux/amd64` and `linux/arm64` using Docker Build Cloud:

```
$ docker build \
    --builder <cloud-builder> \
    --platform linux/amd64,linux/arm64 \
    --output ./bin .
```

This command builds the image using the cloud builder and exports the binaries to the `bin` directory.

26. Verify that the binaries are built for both platforms. You should see the `nvim` binary for both `linux/amd64` and `linux/arm64`.

```
$ tree ./bin
./bin
```

```
├── linux_amd64
│       └── nvim
└── linux_arm64
        └── nvim

3 directories, 2 files
```

Cross-compiling a Go application

This example demonstrates how to cross-compile a Go application for multiple platforms using multi-stage builds. The application is a simple HTTP server that listens on port 8080 and returns the architecture of the container. This example uses Go, but the same principles apply to other programming languages that support cross-compilation.

Cross-compilation with Docker builds works by leveraging a series of pre-defined (in BuildKit) build arguments that give you information about platforms of the builder and the build targets. You can use these pre-defined arguments to pass the platform information to the compiler.

In Go, you can use the `GOOS` and `GOARCH` environment variables to specify the target platform to build for.

Prerequisites:

- Docker Desktop or Docker Engine

Steps:

1. Create an empty directory and navigate to it:

```
$ mkdir go-server
$ cd go-server
```

4. Create a base Dockerfile that builds the Go application:

```
# syntax=docker/dockerfile:1
FROM golang:alpine AS build
WORKDIR /app
ADD https://github.com/dvdksn/buildme.git#eb6279e0ad8a10003718656c6867539bd9426ad8 .
RUN go build -o server .

FROM alpine
COPY --from=build /app/server /server
ENTRYPOINT ["/server"]
```

This Dockerfile can't build multi-platform with cross-compilation yet. If you were to try to build this Dockerfile with `docker build`, the builder would attempt to use emulation to build the image for the specified platforms.

13. To add cross-compilation support, update the Dockerfile to use the pre-defined `BUILDPLATFORM` and `TARGETPLATFORM` build arguments. These arguments are automatically available in the Dockerfile when you use the `--platform` flag with `docker build`.

    - Pin the `golang` image to the platform of the builder using the `--platform=$BUILDPLATFORM` option.
    - Add `ARG` instructions for the Go compilation stages to make the `TARGETOS` and `TARGETARCH` build arguments available to the commands in this stage.
    - Set the `GOOS` and `GOARCH` environment variables to the values of `TARGETOS` and `TARGETARCH`. The Go compiler uses these variables to do cross-compilation.

Updated Dockerfile Old Dockerfile Diff

```
# syntax=docker/dockerfile:1
FROM --platform=$BUILDPLATFORM golang:alpine AS build
ARG TARGETOS
ARG TARGETARCH
WORKDIR /app
ADD
https://github.com/dvdksn/buildme.git#eb6279e0ad8a10003718656c686
7539bd9426ad8 .
RUN GOOS=${TARGETOS} GOARCH=${TARGETARCH} go build -o server .

FROM alpine
COPY --from=build /app/server /server
ENTRYPOINT ["/server"]
```

14. Build the image for `linux/amd64` and `linux/arm64`:

```
15.  $ docker build --platform linux/amd64,linux/arm64 -t go-server .
```

This example has shown how to cross-compile a Go application for multiple platforms with Docker builds. The specific steps on how to do cross-compilation may vary depending on the programming language you're using. Consult the documentation for your programming language to learn more about cross-compiling for different platforms.

Tip

You may also want to consider checking out xx - Dockerfile cross-compilation helpers. `xx` is a Docker image containing utility scripts that make cross-compiling with Docker builds easier.