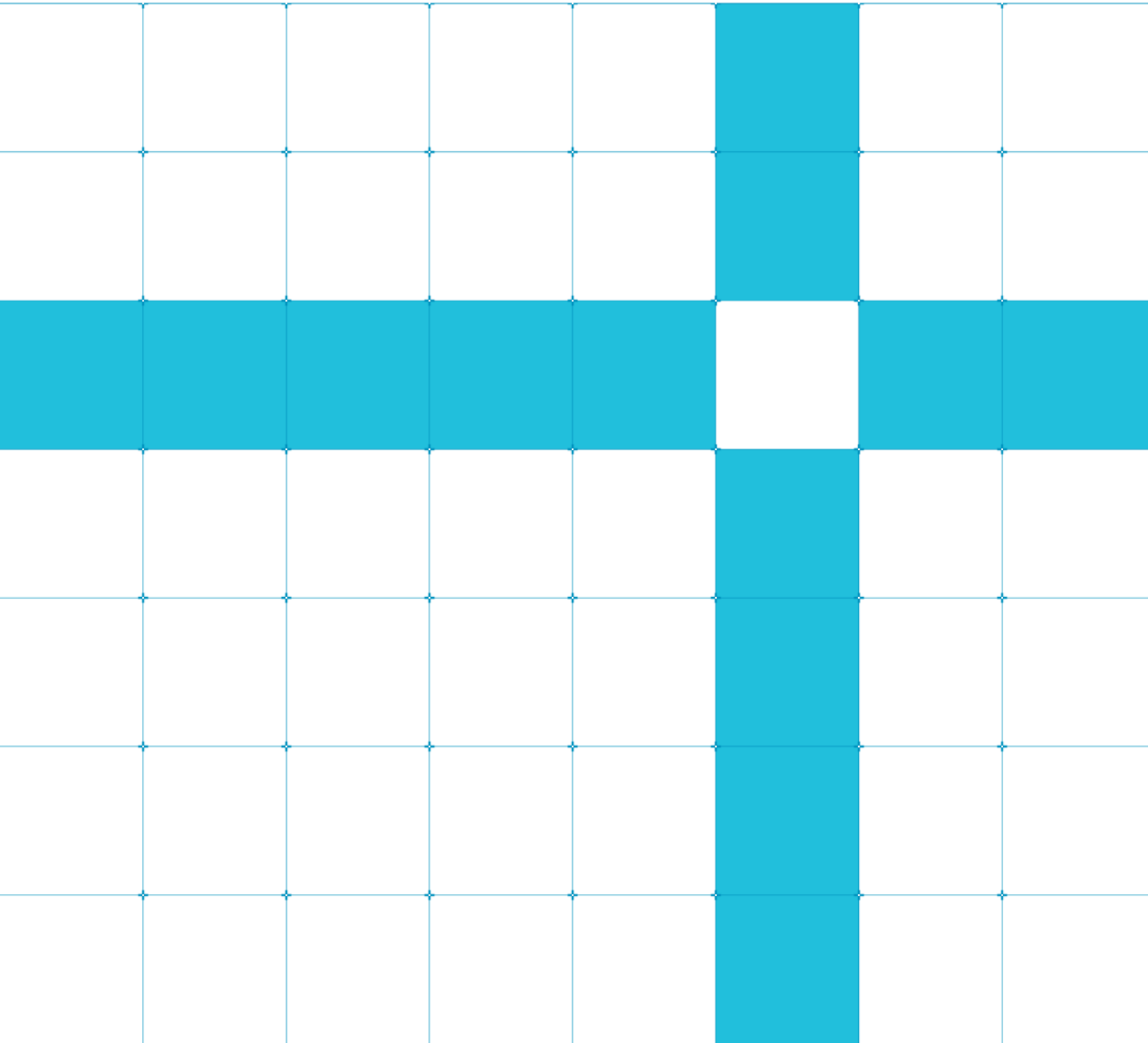
arm

Getting started with Docker

Version 1.0



Common Tasks: Getting started with Docker

Release Information

Document History

Version	Date	Confidentiality	Change
1.0	01 Mar 2020	Non-confidential	First release.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

1 Overview

This guide describes how you can use Docker to simplify multi-architecture application deployment on both embedded devices and servers.

Scaling up software environments quickly can be a difficult and time-consuming task. In this guide, we will show you how Docker abstracts operating systems and hardware details, allowing you to develop more efficiently because you can focus on applications. Benefits include:

- Improved productivity
- Increased infrastructure utilization
- Ability to run both legacy environments and new architectures

We will also show you how using Arm and Docker allows you to:

- Explore a diversity of compute choices
- Simplify application development for Arm hardware
- Work in the cloud and deploy at the edge

1.1. Before you begin

This guide assumes that you are familiar with container concepts. If you are not familiar with container concepts, you can learn more at **Docker Get Started.**

You will also need the following:

- Access to GitHub, so that you can find and use a hello world PHP example for Docker.
- An AWS account, to launch an A1 instance in **Elastic Compute Cloud** (EC2). There will be a small charge to use the option that is required in this guide. See **Related information** to learn more.
- A Raspberry Pi 3 or Raspberry Pi 4 running **Raspbian**, if you want to follow and run the example in this guide on an embedded device.

2 Multi-architecture containers?

To facilitate multi-platform portability, Docker and Arm have built multi-architecture container images with transparent support for the Arm architecture.

These multi-architecture containers abstract away underlying hardware details, allowing you to develop consistently reproducible environments for the Arm architecture. You can use this functionality across various platforms like Windows or Mac laptops, servers in the cloud, and embedded or IoT devices. Another benefit of this functionality is that you can try the Arm architecture with almost no change to the development process.

Multi-architecture containers provide native execution on Arm servers in the cloud and on embedded devices. The same containers can be run and validated on the desktop using instruction translation. For example, you can develop an application with Docker on a desktop machine and then move the application to a server, cloud machine, or embedded device.

In this guide, we will describe how to use multi-architecture containers at each step of the development process. At the end of the guide, you will be able to:

- Install Docker on a local desktop machine
- Create an application in a Docker image
- Push the Docker image to a registry like **Docker Hub**
- Install Docker on a cloud machine like an **AWS A1 instance**
- Run the application in the cloud

Both Windows and Mac are supported.

We will also show how to run the same image is run on a Raspberry Pi.

2.1. Local development

Even though much software deployment happens in the cloud or on an embedded system, developers often prefer to run locally to get started and to fix simple issues. One of the benefits of containers is they can be easily migrated to different types of machines. Docker facilitates these migrations and enables applications to run on the Arm architecture.

3 Install Docker

In this section of the guide, we will install Docker Desktop for Windows or Mac. To install Docker Desktop, follow these steps:

1. Download the **package**.
2. Install Docker Desktop either by dragging the whale icon into your Applications folder on Mac or clicking though the installer on Windows.

Note: Docker Desktop for Windows requires Microsoft Hyper-V to run. The Docker Desktop for Windows installer enables Hyper-V for you. Restart your machine if you are prompted to enact this change.

Commands starting with > are for PowerShell on Windows but are identical on Mac. Commands starting with $ are for bash on Linux. Do not enter the > or the $ on any operating system.

3. Open Windows PowerShell, or Windows Command Prompt, and run hello-world to confirm that Docker is set up correctly, if it is, it will look like the following:

```
> docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/
```

4 Multi-architecture images

In this section of the guide, we explain multi-architecture containers. Multi-architecture containers support execution as an Arm image or as an x86 image.

For example, to run a Python container with the x86_64 architecture on a Windows PC, you can execute the command that is shown in the following code block:

```
> docker run python:2 python -c "import platform; print 'Python running on arch: %s' %platform.machine()"
Python running on arch: x86_64
```

The following code shows another example using Ubuntu Linux and printing the architecture using uname:

```
> docker run ubuntu uname -a
Linux 436761beed66 4.9.125-linuxkit #1 SMP Fri Sep 7 08:20:28 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
```

You might want to deploy your application on an Arm server on AWS, or on an Arm embedded board. Developing an application using Docker on an x86 machine may introduce incompatibilities when running the same software on an Arm machine. Docker Desktop allows you to create and test Arm images from your Windows desktop.

The best way to create images for Arm is to use the new `buildx` command which is included in Docker Desktop. The command usage is shown in this code:

```
> docker buildx

Usage:  docker buildx COMMAND

Build with BuildKit

Management Commands:
  imagetools  Commands to work on images in registry

Commands:
  bake        Build from a file
  build       Start a build
  create      Create a new builder instance
  inspect     Inspect current builder instance
  ls          List builder instances
  rm          Remove a builder instance
  stop        Stop builder instance
  use         Set the current builder instance

Run 'docker buildx COMMAND --help' for more information on a command.
```

Let's use a simple Dockerfile to see how the same Dockerfile supports multiple architectures with no modifications. Use a text editor to create a two-line Dockerfile with the contents that are shown in the following code:

```
FROM alpine
RUN apk --no-cache add curl
```

The --no-cache argument is a useful trick to not cache the index and keep containers small.

Now we look at how to build images for different platforms using `buildx`. Follow these steps:

1. Create a new builder instance and use it, then build three images which are stored locally, this can be seen in the following code:

```
> docker buildx create --name mybuilder
> docker buildx use mybuilder
> docker buildx build --platform linux/amd64 -t alpine-amd64 --load .
> docker buildx build --platform linux/arm64 -t alpine-arm64 --load .
> docker buildx build --platform linux/arm/v7 -t alpine-arm32 --load .
```

2. Run each image, including the Arm images, on the local desktop, as you can see in the following code:

```
> docker run alpine-amd64 uname -a
Linux 4bc3bd4b8ff0 4.9.125-linuxkit #1 SMP Fri Sep 7 08:20:28 UTC 2018 x86_64 Linux

> docker run alpine-arm64 uname -a
Linux 404631ac3379 4.9.125-linuxkit #1 SMP Fri Sep 7 08:20:28 UTC 2018 aarch64 Linux

> docker run alpine-arm32 uname -a
Linux 5a869d794098 4.9.125-linuxkit #1 SMP Fri Sep 7 08:20:28 UTC 2018 armv7l Linux
```

Now the images for each architecture are on this local machine. However, the goal is to save a single image that works on all platforms. This can be done with a single `buildx` command which pushes directly to a repository like Docker Hub. Using a repository is an easy way to migrate images to other machines, because the images can be pulled directly from the repository using any machine.

3. Create a Docker Hub account if you do not have one. You can go to **hub.docker.com** and click **Sign Up**.

Once pushed, `buildx` can be used to inspect the image and see that it supports three platforms, using the following code:

```
> docker buildx build  --platform linux/amd64,linux/arm64,linux/arm/v7 -t
jasonrandrews/alpine-test --push .


> docker buildx imagetools inspect jasonrandrews/alpine-test
Name:      docker.io/jasonrandrews/alpine-test:latest
MediaType: application/vnd.docker.distribution.manifest.list.v2+json
Digest:    sha256:6f36d248c3b139dc998cd7129a768cf068dd6371ecd6f027ce43c49b6bf80aa4


Manifests:
  Name:      docker.io/jasonrandrews/alpine-
test:latest@sha256:aaf426c683e2b1369fdba62e6c420980402fc3706c59ec59eacf0d1ab419e719
```

```
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/amd64


  Name:      docker.io/jasonrandrews/alpine-
test:latest@sha256:5966f7b12b7c7ba3146102cf3079e57cccaf1de7228651661276dd1606d84108
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm64


  Name:      docker.io/jasonrandrews/alpine-
test:latest@sha256:12e131c1e16083f5d8a8dcaf8b52b10e78612cea439e98b1cd32fe9a60a3cefa
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm/v7
```

4. Check your repositories by going to hub.docker.com and signing in with the same account you pushed to.

If the new image is run with just the name, it will automatically run the native version. You can use the image name from the inspect command to specify the Arm image to run on Docker Desktop, as you can see in this code:

```
> docker run jasonrandrews/alpine-test uname -m
x86_64


> docker run jasonrandrews/alpine-
test:latest@sha256:5966f7b12b7c7ba3146102cf3079e57cccaf1de7228651661276dd1606d84108
uname -m
aarch64


> docker run jasonrandrews/alpine-
test:latest@sha256:12e131c1e16083f5d8a8dcaf8b52b10e78612cea439e98b1cd32fe9a60a3cefa
uname -m
armv7l
```

Docker abstracts the underlying operating system and the underlying hardware architecture. This helps to avoid architecture-specific bugs during development. We have shown how to create a single repository with support for three architectures using the `buildx` command.

To create multi-architecture support for more than one hundred official images, follow these steps:

1. Build the application using Docker. A Dockerfile is provided and you can see the command to build the image in the following code:

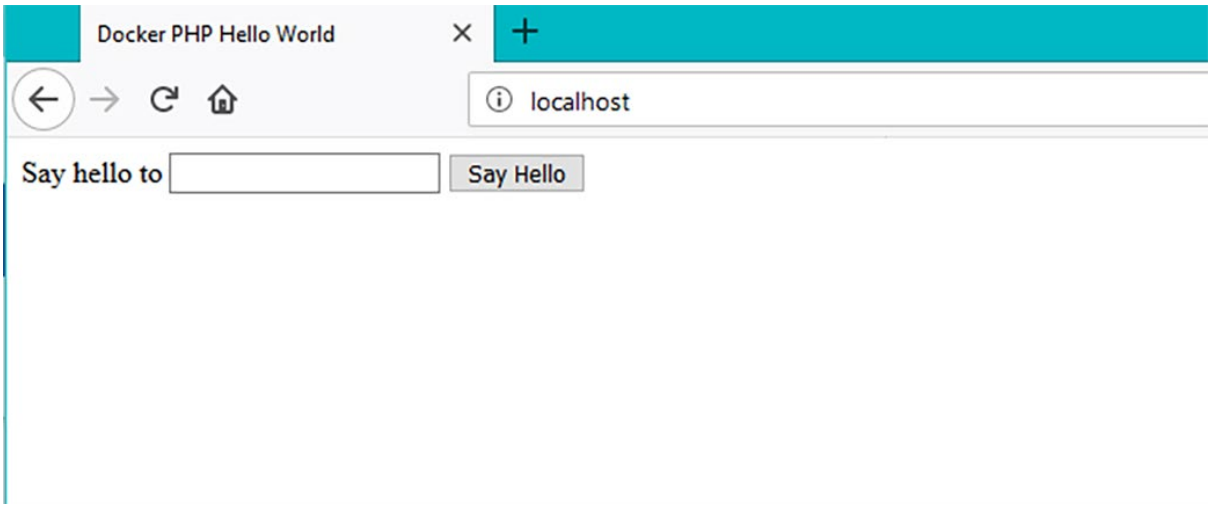
```
> docker build -t php-example docker-php-hello-world
Sending build context to Docker daemon   76.8kB
Step 1/2 : FROM php:5.6-apache
5.6-apache: Pulling from library/php
5e6ec7f28fb7: Pull complete
cf165947b5b7: Pull complete
7bd37682846d: Pull complete
99daf8e838e1: Pull complete
ae320713efba: Pull complete
```

```
ebcb99c48d8c: Pull complete
9867e71b4ab6: Pull complete
936eb418164a: Pull complete
bc298e7adaf7: Pull complete
ccd61b587bcd: Pull complete
b2d4b347f67c: Pull complete
56e9dde34152: Pull complete
9ad99b17eb78: Pull complete
Digest: sha256:0a40fd273961b99d8afe69a61a68c73c04bc0caa9de384d3b2dd9e7986eec86d
Status: Downloaded newer image for php:5.6-apache
 ---> 24c791995c1e
Step 2/2 : COPY public/ /var/www/html/
 ---> 00a704f44d8a
Successfully built 00a704f44d8a
Successfully tagged php-example:latest
```

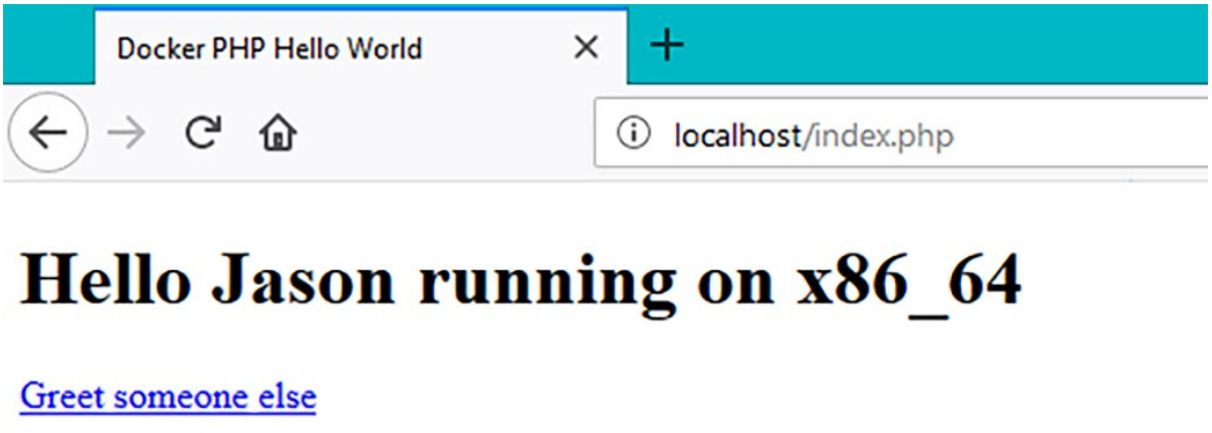
2. Run the image with Docker using the following code:

```
> docker run -it --rm -p 80:80 php-example
AH00558: apache2: Could not reliably determine the server's fully qualified domain
name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this
message
AH00558: apache2: Could not reliably determine the server's fully qualified domain
name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this
message
[Mon Apr 22 14:34:24.853585 2019] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4.25
(Debian) PHP/5.6.40 configured -- resuming normal operations
[Mon Apr 22 14:34:24.853661 2019] [core:notice] [pid 1] AH00094: Command line: 'apache2
-D FOREGROUND'
172.17.0.1 - - [22/Apr/2019:14:34:34 +0000] "GET / HTTP/1.1" 200 520 "-" "Mozilla/5.0
(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/73.0.3683.103 Safari/537.36"
```

3. Open a browser and connect to the port 80 on the local machine. You will see what is displayed in the following screenshot:



4. Enter a name in the box and confirm that the hello world application is working. The application also prints the output of uname -m so that we can confirm the platform. You will see something like the following screenshot:



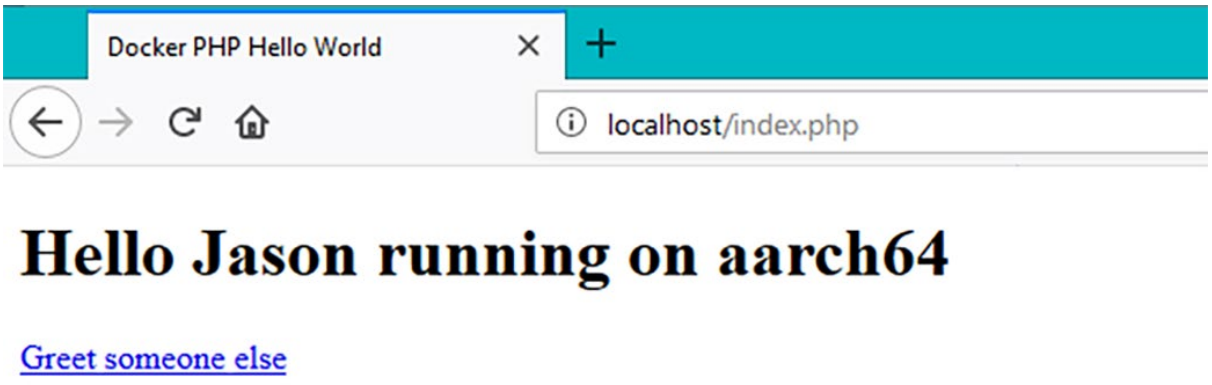
Now, let's prepare and run the same application for multiple architectures, including Arm:

1. Follow the preceding steps 1-5.
2. Use the new `buildx` flow to create all the images and push them to Docker Hub with a single command.
3. Run the native image by specifying the image name.
4. Test the Arm images by specifying the full name that is provided by the `buildx inspect` command, this should look like the following code:

```
> docker buildx build  --platform linux/amd64,linux/arm64,linux/arm/v7 -t
jasonrandrews/php-example --push docker-php-hello-world
```

```
> docker run -it --rm -p 80:80 jasonrandrews/php-example


> docker buildx imagetools inspect jasonrandrews/php-example
docker buildx imagetools inspect jasonrandrews/php-example
Name:      docker.io/jasonrandrews/php-example:latest
MediaType: application/vnd.docker.distribution.manifest.list.v2+json
Digest:    sha256:82242b9bdacb51c6201c3f4943f2139ba77c1b1233d0e18862c12c1649d1c1be

Manifests:
  Name:      docker.io/jasonrandrews/php-
example:latest@sha256:6e4f5a17b2fd4803a43d4586e2f928c04174c42d16d2b07a7444344d77c7458f
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/amd64

  Name:      docker.io/jasonrandrews/php-
example:latest@sha256:4b47c9e79d744f039e893324fcadcabdcd7f33134740abf908115e7a9226de95
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm64

  Name:      docker.io/jasonrandrews/php-
example:latest@sha256:27cf14ff1a5f9079e9b58262c763d1b75c8760e4934d488006bd480163843344
  MediaType: application/vnd.docker.distribution.manifest.v2+json
  Platform:  linux/arm/v7

> docker run -it --rm -p 80:80 docker.io/jasonrandrews/php-
example:latest@sha256:4b47c9e79d744f039e893324fcadcabdcd7f33134740abf908115e7a9226de95
```

Images have been created for three platforms. They are amd64, arm64, and amr32. If the arm64 image is run the output below now shows AArch64 from uname -m. This is an image created for the AArch64 architecture which is running on a Windows or MacOS machine but thinks it is an Arm machine:



We now have a single command to create the Docker image with multi-architecture support for the hello world PHP application for amd64, arm64, and arm32, and to store the image in Docker Hub.

With the application now in an accessible repository, the next step is to pull and run the images on other machines, like an Arm-based cloud server or an embedded device. We will look at this in **Setting up an Arm server**.

5 Setting up an Arm server

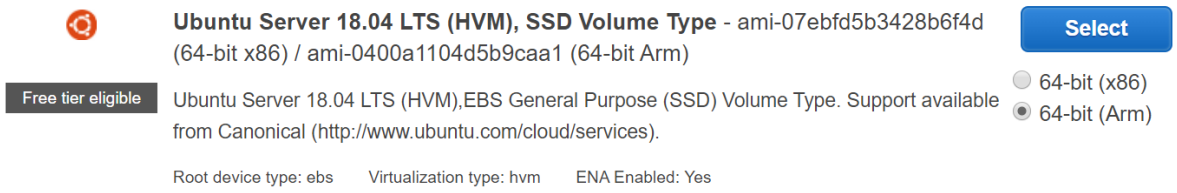
One way to execute the example application on an Arm server is to use **Amazon Web Services** (AWS). The **A1 instance** type features 64-bit Arm Neoverse cores, powered by custom silicon that is designed by AWS.

You need an AWS account to launch an A1 instance in **Elastic Compute Cloud** (EC2). Here are the steps to launch an A1 instance on AWS:

1. Create an AWS account if you do not already have one.
2. Navigate to the EC2 dashboard and select **Launch Instance**.
3. Select the **Amazon Machine Image (AMI)** for the cloud instance.

The AMI includes the operating system and hardware architecture.

4. Select the **Ubuntu Server 18.04 OS**, and click the **64-bit (Arm)** button to the right of the image name as seen in the following screenshot:



5. Click **Select** and chose any A1 instance type. For this example, the **a1.medium** option with one CPU and 2GB RAM is enough. No additional configuration parameters are needed.

When creating the instance, ensure that the security group is set up to allow port 22 for ssh access and port 80 for the PHP application.

6. Click **Review and Launch** and then **Launch** to create your instance. You will be prompted to select a private key pair to connect to this instance.
7. Select **Create a new key pair**, type in a key pair name, for example, A1-PHP-example and select **Download Key Pair**. Keep this A1-PHP-example.pem file safe and accessible. You will need this file to connect to this instance remotely. More information about how to use the key pair from different operating systems is provided in the **AWS documentation.**

This option is not eligible for free-tier AWS usage, but the charges to just try it are minimal, see **Related information**.

8. Navigate back to the **EC2 dashboard > Instances** tab to view the created machine instance. Highlighting the instance displays relevant data on the bottom of the screen, for example the description, status, and tags of the instance. Under the **Description** tab, there is a **Public DNS (IPv4)** section representing this IP address of this instance. You can also use the IP address under the **IPv4 Public IP**.
9. Copy the IP address and use it with the key pair file to ssh into the machine. There are different ways to connect. You can use the AWS documentation for more information. The username to ssh into the machine is **ubuntu**. A simple command-line connection would be what you can see in the following code:

```
> ssh -i A1-PHP-example.pem ubuntu@A1InstanceIP
```

Running this command will log in to the created instance with the ubuntu username.

With an instance created and connected, the next steps are to install Docker and run the containerized application in the cloud. We will explore this in **Docker installation**.

6 Docker installation

Follow these steps to install Docker on supported Linux systems.

```
Execute this terminal command:
$ sudo apt-get update
$ sudo apt-get upgrade
$ curl -fsSL get.docker.com -o get-docker.sh && sh get-docker.sh
```

1. Add the ubuntu user, which is what you are currently, to the Docker group. This will allow you to avoid needing super user (sudo) to run the Docker command, as you can see in the following code:

```
$ sudo usermod -aG docker $USER
```

2. Log out and back in again for the change to take effect.
3. Test the install with a quick hello-world run, as you can see here:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

7 Run Docker on an Arm server

In **Docker installation**, we explained how to install Docker on an Arm server. Now let's look at how to run the containerized application on the AWS Arm instance.

There are several ways to get a Docker image onto this cloud instance, from rebuilding it locally from a Dockerfile to pulling a pre-compiled image from Docker Hub. In **Multi-architecture images,** we pushed the PHP example images to Docker Hub so that we can run them immediately.

Follow these steps to run the PHP Docker image on an Arm server:

1. Run the following command in the terminal in the AWS instance:

```
$ docker run --rm -it -p 80:80 jasonrandrews/php-example
```

Using the `docker run` command will automatically fetch the image from Docker Hub, if the image is not found locally. Running this command pulls the application from Docker Hub and run it on this AWS A1 instance.

2. Open a web browser and paste the IP address of the A1 instance into the browser to view the PHP example application. The same PHP example application appears with the uname -m showing AArch64.

The Docker image was developed using multi-architecture containers. This means that the behavior during development and deployment will be the same, even though the application was developed on a different hardware architecture than Arm.

In **Execute on an embedded device**, we will look at how to use the PHP example application on an embedded device.

8 Execute on an embedded device

Although the example in this guide focuses on a cloud-based application, the same flow applies to at-scale embedded use-cases as well.

The benefits of abstracting hardware differences during development and using the Arm architecture also apply in the embedded space. Docker makes software deployment for IoT and embedded systems easier because you do not need to flash individual boards, install necessary dependencies, or build scripts to copy files to boards. Downloading and running a Docker image ensures environment consistency and simplifies deployment across embedded or IoT devices.

Let's run the same PHP example application on a **Raspberry Pi 3** or a Raspberry Pi 4. Some boards may require some minor Linux kernel modifications to ensure that the Docker engine can run properly. The Raspberry Pi running **Raspbian** will run Docker with no configuration changes. Follow **installation instructions** to install Raspbian and boot the Raspberry Pi from an SD card.

The instructions to run the PHP example application are the same as the AWS A1 instance. The only difference is that Raspbian runs only 32-bit images.

Follow these steps:

1. Enable the **ssh server** and connect to the board.
2. Run the same commands that you ran on the A1 instance, as you can see in the following code:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ curl -fsSL get.docker.com -o get-docker.sh && sh get-docker.sh
$ sudo usermod -aG docker $USER
```

3. Add the pi user to the docker group to avoid needing sudo to run the docker command.
4. Log out and back in again.
5. Test the install with a quick hello-world run, as you can see here:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1eda109e4da: Already exists
Digest: sha256:92695bc579f31df7a63da6922075d0666e565ceccad16b59c3374d2cf4e8e50e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm32v7)
 3. The Docker daemon created a new container from that image which runs the
```

```
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

6. Run the PHP example application. The example application is the same as it was with Docker Desktop on Windows or Mac, on the AWS A1 Arm server, and on the Raspberry Pi.

```
$ docker run --rm -it -p 80:80 jasonrandrews/php-example
```

7. Open a browser on the Raspberry Pi, or on another computer on the network, and enter in the IP address of the Pi. You will see the same PHP example application appear, but this time the response shows the uname as armv7l. The same command runs the PHP example application on the Windows, or Mac, desktop, the Arm server in AWS, and the Raspberry Pi.

9 Related information

Here are some resources related to material in this guide:

- AWS
  - **AWS A1 instance**
  - **AWS documentation**
  - **AWS pricing information**
- **Docker Hub**
  - **Docker get started**
  - **Official Images on Docker Hub**
  - **Docker Desktop**
- **Raspberry Pi**
  - **Raspbian**

10 Next steps

This guide has shown the benefits for software developers of using Docker Desktop for software development for Arm. Use multi-architecture containers for development leads to faster time to market and lower deployment costs. Almost any software project in almost any industry can leverage these benefits.