Docker 

#docker

**Table of Contents**

About 1 Chapter 1: Getting started with Docker 2 Remarks 2 Versions 2 Examples 2 Installing Docker on Mac OS X 3 Installing Docker on Windows 4 Installing docker on Ubuntu Linux 5 Installing Docker on Ubuntu 9 Create a docker container in Google Cloud 11 Install Docker on Ubuntu 12 Installating Docker-ce OR Docker-ee on CentOS 16 Docker-ce Installation 16 -Docker-ee (Enterprise Edition) Installation 17 Chapter 2: Building images 19 Parameters 19 Examples 19 Building an image from a Dockerfile 19 A simple Dockerfile 20 Difference between ENTRYPOINT and CMD 20 Exposing a Port in the Dockerfile 21 Example: 21 ENTRYPOINT and CMD seen as verb and parameter 22 Pushing and Pulling an Image to Docker Hub or another Registry 22 Building using a proxy 23 Chapter 3: Checkpoint and Restore Containers 24 Examples 24 Compile docker with checkpoint and restore enabled (ubuntu) 24 Checkpoint and Restore a Container 25 Chapter 4: Concept of Docker Volumes 27

Remarks 27 Examples 27 A) Launch a container with a volume 27 B) Now press [cont +P+Q] to move out from container without terminating the container chec 27 C) Run 'docker inspect' to check out more info about the volume 27 D) You can attach a running containers volume to another containers 27 E) You can also mount you base directory inside container 28 Chapter 5: Connecting Containers 29 Parameters 29 Remarks 29 Examples 29 Docker network 29 Docker-compose 29 Container Linking 30 Chapter 6: Creating a service with persistence 31 Syntax 31 Parameters 31 Remarks 31 Examples 31 Persistence with named volumes 31 Backup a named volume content 32 Chapter 7: Data Volumes and Data Containers 33 Examples 33 Data-Only Containers 33 Creating a data volume 33 Chapter 8: Debugging a container 35 Syntax 35 Examples 35 Entering in a running container 35 Monitoring resource usage 35 Monitoring processes in a container 36 Attach to a running container 36

Printing the logs 37 Docker container process debugging 38 Chapter 9: Docker Data Volumes 39 Introduction 39 Syntax 39 Examples 39 Mounting a directory from the local host into a container 39 Creating a named volume 39 Chapter 10: Docker Engine API 41 Introduction 41 Examples 41 Enable Remote access to Docker API on Linux 41 Enable Remote access to Docker API on Linux running systemd 41 Enable Remote Access with TLS on Systemd 42 Image pulling with progress bars, written in Go 42 Making a cURL request with passing some complex structure 45 Chapter 11: Docker events 46 Examples 46 Launch a container and be notified of related events 46 Chapter 12: Docker in Docker 47 Examples 47 Jenkins CI Container using Docker 47 Chapter 13: docker inspect getting various fields for key:value and elements of list 48 Examples 48 various docker inspect examples 48 Chapter 14: Docker Machine 51 Introduction 51 Remarks 51 Examples 51 Get current Docker Machine environment info 51 SSH into a docker machine 51 Create a Docker machine 51

List docker machines 52 Upgrade a Docker Machine 53 Get the IP address of a docker machine 53

Chapter 15: Docker --net modes (bridge, hots, mapped container and none). 54 Introduction 54 Examples 54

Bridge Mode, Host Mode and Mapped Container Mode 54 Chapter 16: Docker network 56 Examples 56 How to find the Container's host ip 56 Creating a Docker network 56 Listing Networks 56 Add container to network 56 Detach container from network 57 Remove a Docker network 57 Inspect a Docker network 57 Chapter 17: Docker private/secure registry with API v2 59 Introduction 59 Parameters 59 Remarks 60 Examples 60 Generating certificates 60 Run the registry with self-signed certificate 60 Pull or push from a docker client 61 Chapter 18: Docker Registry 62 Examples 62 Running the registry 62 Configure the registry with AWS S3 storage backend 62 Chapter 19: Docker stats all running containers 63 Examples 63 Docker stats all running containers 63 Chapter 20: Docker swarm mode 64

Introduction 64 Syntax 64 Remarks 64

Swarm Mode CLI Commands 64 Examples 65 Create a swarm on Linux using docker-machine and VirtualBox 65 Find out worker and manager join token 66 Hello world application 66 Node Availablility 68 Promote or Demote Swarm Nodes 68 Leaving the Swarm 68 Chapter 21: Dockerfile contents ordering 70 Remarks 70 Examples 70 Simple Dockerfile 70 Chapter 22: Dockerfiles 72 Introduction 72 Remarks 72 Examples 72 HelloWorld Dockerfile 72 Copying files 73 Exposing a port 73 Dockerfiles best pratices 73 USER Instruction 74 WORKDIR Instruction 74 VOLUME Instruction 75 COPY Instruction 75 The ENV and ARG Instruction 76 ENV 76 ARG 77 EXPOSE Instruction 77 LABEL Instruction 78

CMD Instruction 79 MAINTAINER Instruction 80 FROM Instruction 80 RUN Instruction 81 ONBUILD Instruction 82 STOPSIGNAL Instruction 83 HEALTHCHECK Instruction 83 SHELL Instruction 84 Installing Debian/Ubuntu packages 86

Chapter 23: How to debug when docker build fails 88 Introduction 88 Examples 88

basic example 88

Chapter 24: How to Setup Three Node Mongo Replica using Docker Image and Provisioned using 89

Introduction 89 Examples 89 Build Step 89 Chapter 25: Inspecting a running container 93 Syntax 93 Examples 93 Get container information 93 Get specific information from a container 93 Inspect an image 95 Printing specific informations 96 Debugging the container logs using docker inspect 97 Examining stdout/stderr of a running container 97 Chapter 26: Iptables with Docker 98 Introduction 98 Syntax 98 Parameters 98 Remarks 98

The problem 98 The solution 99 Examples 100 Limit access on Docker containers to a set of IPs 100 Configure restriction access when Docker daemon starts 101 Some custom iptables rules 101 Chapter 27: Logging 102 Examples 102 Configuring a log driver in systemd service 102 Overview 102 Chapter 28: Managing containers 103 Syntax 103 Remarks 103 Examples 103 Listing containers 103 Referencing containers 104 Starting and stopping containers 104 List containers with custom format 105 Finding a specific container 105 Find container IP 105 Restarting docker container 105 Remove, delete and cleanup containers 105 Run command on an already existing docker container 106 Container logs 107 Connect to an instance running as daemon 107 Copying file from/to containers 107 Remove, delete and cleanup docker volumes 108 Export and import Docker container filesystems 108 Chapter 29: Managing images 110 Syntax 110 Examples 110 Fetching an image from Docker Hub 110

Listing locally downloaded images 110 Referencing images 110 Removing Images 111 Search the Docker Hub for images 112 Inspecting images 112 Tagging images 113 Saving and loading Docker images 113

Chapter 30: Multiple processes in one container instance 114 Remarks 114 Examples 114

Dockerfile + supervisord.conf 114 Chapter 31: passing secret data to a running container 116 Examples 116 ways to pass secrets in a container 116 Chapter 32: Restricting container network access 117 Remarks 117 Examples 117 Block access to LAN and out 117 Block access to other containers 117 Block access from containers to the local host running docker daemon 117 Block access from containers to the local host running docker daemon (custom network) 118 Chapter 33: run consul in docker 1.12 swarm 119 Examples 119 Run consul in a docker 1.12 swarm 119 Chapter 34: Running containers 120 Syntax 120 Examples 120 Running a container 120 Running a different command in the container 120 Automatically delete a container after running it 120 Specifying a name 121 Binding a container port to the host 121

Container restart policy (starting a container at boot) 121 Run a container in background 122 Assign a volume to a container 122 Setting environment variables 123 Specifying a hostname 124 Run a container interactively 124 Running container with memory/swap limits 124 Getting a shell into a running (detached) container 124

Log into a running container 124 Log into a running container with a specific user 124 Log into a running container as root 125 Log into a image 125 Log into a intermediate image (debug) 125

Passing stdin to the container 126 Detaching from a container 126 Overriding image entrypoint directive 126 Add host entry to container 126 Prevent container from stopping when no commands are running 127 Stopping a container 127 Execute another command on a running container 127 Running GUI apps in a Linux container 127

Chapter 35: Running services 130 Examples 130 Creating a more advanced service 130 Creating a simple service 130 Removing a service 130 Scaling a service 130 Chapter 36: Running Simple Node.js Application 131 Examples 131 Running a Basic Node.js application inside a Container 131 Build your image 132

Running the image 133 Chapter 37: security 135 Introduction 135 Examples 135 How to find from which image our image comes from 135 Credits 136

**About**

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: docker

It is an unofficial and free Docker ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Docker.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

https://riptutorial.com/ 1

**Chapter 1: Getting started with Docker Remarks**

Docker is an open-source project that automates the deployment of applications inside software containers. These application containers are similar to lightweight virtual machines, as they can be run in isolation to each other and the running host.

Docker requires features present in recent linux kernels to function properly, therefore on Mac OSX and Windows host a virtual machine running linux is required for docker to operate properly. Currently the main method of installing and setting up this virtual machine is via Docker Toolbox that is using VirtualBox internally, but there are plans to integrate this functionality into docker itself, using the native virtualisation features of the operating system. On Linux systems docker run natively on the host itself.

**Versions**

| **Version** | **Release Date** |
| --- | --- |
| 17.05.0 | 2017-05-04 |
| 17.04.0 | 2017-04-05 |
| 17.03.0 | 2017-03-01 |
| 1.13.1 | 2016-02-08 |
| 1.12.0 | 2016-07-28 |
| 1.11.2 | 2016-04-13 |
| 1.10.3 | 2016-02-04 |
| 1.9.1 | 2015-11-03 |
| 1.8.3 | 2015-08-11 |
| 1.7.1 | 2015-06-16 |
| 1.6.2 | 2015-04-07 |
| 1.5.0 | 2015-02-10 |

**Examples**

https://riptutorial.com/ 2

**Installing Docker on Mac OS X**

**Requirements:** OS X 10.8 “Mountain Lion” or newer required to run Docker.

While the docker binary can run natively on Mac OS X, to build and host containers you need to run a Linux virtual machine on the box.

1.12.0

Since version 1.12 you don't need to have a separate VM to be installed, as Docker can use the native Hypervisor.framework functionality of OSX to start up a small Linux machine to act as backend.

To install docker follow the following steps:

1. Go to Docker for Mac

2. Download and run the installer.

3.

Continue through installer with default options and enter your account credentials when requested.

Check here for more information on the installation.

1.11.2

Until version 1.11 the best way to run this Linux VM is to install Docker Toolbox, that installs Docker, VirtualBox and the Linux guest machine.

To install docker toolbox follow the following steps:

1. Go to Docker Toolbox

2. Click the link for Mac and run the installer.

3.

Continue through installer with default options and enter your account credentials when requested.

This will install the Docker binaries in /usr/local/bin and update any existing Virtual Box installation. Check here for more information on the installation.

**To Verify Installation:**

1.12.0

1.

Start Docker.app from the Applications folder, and make sure it is running. Next open up Terminal.

1.11.2

1.

Open the Docker Quickstart Terminal, which will open a terminal and prepare it for use for Docker commands.

2. Once the terminal is open type

https://riptutorial.com/ 3

$ docker run hello-world

3.

If all is well then this should print a welcome message verifying that the installation was successful.

**Installing Docker on Windows**

**Requirements:** 64-bit version of Windows 7 or higher on a machine which supports Hardware Virtualization Technology, and it is enabled.

While the docker binary can run natively on Windows, to build and host containers you need to run a Linux virtual machine on the box.

1.12.0

Since version 1.12 you don't need to have a separate VM to be installed, as Docker can use the native Hyper-V functionality of Windows to start up a small Linux machine to act as backend.

To install docker follow the following steps:

1. Go to Docker for Windows

2. Download and run the installer.

3.

Continue through installer with default options and enter your account credentials when requested.

Check here for more information on the installation.

1.11.2

Until version 1.11 the best way to run this Linux VM is to install Docker Toolbox, that installs Docker, VirtualBox and the Linux guest machine.

To install docker toolbox follow the following steps:

1. Go to Docker Toolbox

2. Click the link for Windows and run the installer.

3.

Continue through installer with default options and enter your account credentials when requested.

This will install the Docker binaries in Program Files and update any existing Virtual Box installation. Check here for more information on the installation.

**To Verify Installation:**

1.12.0

1.

Start Docker from the Start menu if it hasn't been started yet, and make sure it is running. Next upen up any terminal (either cmd or PowerShell)

1.11.2

https://riptutorial.com/ 4

1.

On your Desktop, find the Docker Toolbox icon. Click the icon to launch a Docker Toolbox terminal.

2.

Once the terminal is open type

docker run hello-world

3.

If all is well then this should print a welcome message verifying that the installation was successful.

**Installing docker on Ubuntu Linux**

Docker is supported on the following 64-bit versions of Ubuntu Linux:

• Ubuntu Xenial 16.04 (LTS)

• Ubuntu Wily 15.10

• Ubuntu Trusty 14.04 (LTS)

• Ubuntu Precise 12.04 (LTS)

A couple of notes:

The following instructions involve installation using **Docker** packages only, and this ensures obtaining the latest official release of **Docker**. If you need to install only using Ubuntu-managed packages, consult the Ubuntu documentation (Not recommended otherwise for obvious reasons).

Ubuntu Utopic 14.10 and 15.04 exist in Docker’s APT repository but are no longer officially supported due to known security issues.

**Prerequisites**

• Docker only works on a 64-bit installation of Linux.

•

Docker requires Linux kernel version 3.10 or higher (Except for Ubuntu Precise 12.04, which requires version 3.13 or higher). Kernels older than 3.10 lack some of the features required to run Docker containers and contain known bugs which cause data loss and frequently panic under certain conditions. Check current kernel version with the command uname -r. Check this post if you need to update your Ubuntu Precise (12.04 LTS) kernel by scrolling further down. Refer to this WikiHow post to obtain the latest version for other Ubuntu installations.

**Update APT sources**

This needs to be done so as to access packages from Docker repository.

1. Log into your machine as a user with sudo or root privileges. 2. Open a terminal window.

3.

Update package information, ensure that APT works with the https method, and that CA certificates are installed.

https://riptutorial.com/ 5

$ sudo apt-get update

$ sudo apt-get install \ apt-transport-https \ ca-certificates \

curl \

software-properties-common

4.

Add Docker’s official GPG key:

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

Verify that the key fingerprint is **9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88** .

$ sudo apt-key fingerprint 0EBFCD88

pub 4096R/0EBFCD88 2017-02-22

Key fingerprint = 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88 uid Docker Release (CE deb) <docker@docker.com> sub 4096R/F273FCD8 2017-02-22

5.

Find the entry in the table below which corresponds to your Ubuntu version. This determines where APT will search for Docker packages. When possible, run a long-term support (LTS) edition of Ubuntu.

| **Ubuntu Version** | **Repository** |
| --- | --- |
| Precise 12.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-precise main |
| Trusty 14.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-trusty main |
| Wily 15.10 | deb https://apt.dockerproject.org/repo ubuntu-wily main |
| Xenial 16.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-xenial main |

**Note:** Docker does not provide packages for all architectures. Binary artifacts are built nightly, and you can download them from https://master.dockerproject.org. To install docker on a multi-architecture system, add an [arch=...] clause to the entry. Refer to Debian Multiarch wiki for details.

6.

Run the following command, substituting the entry for your operating system for the placeholder <REPO>.

$ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Update the APT package index by executing sudo apt-get update.

8. Verify that APT is pulling from the right repository.

When you run the following command, an entry is returned for each version of Docker that is https://riptutorial.com/ 6

available for you to install. Each entry should have the URL https://apt.dockerproject.org/repo/. The version currently installed is marked with \*\*\*.See the below example's output.

$ apt-cache policy docker-engine

docker-engine:

Installed: 1.12.2-0~trusty

Candidate: 1.12.2-0~trusty

Version table:

\*\*\* 1.12.2-0~trusty 0

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages 100 /var/lib/dpkg/status

1.12.1-0~trusty 0

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages 1.12.0-0~trusty 0

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages

From now on when you run apt-get upgrade, APT pulls from the new repository. **Prerequisites by Ubuntu Version**

For Ubuntu Trusty (14.04) , Wily (15.10) , and Xenial (16.04) , install the linux-image-extra-\* kernel packages, which allows you use the aufs storage driver.

To install the linux-image-extra-\* packages:

1. Open a terminal on your Ubuntu host.

2. Update your package manager with the command sudo apt-get update.

3.

Install the recommended packages.

$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual

4. Proceed to Docker installation

For Ubuntu Precise (12.04 LTS), Docker requires the 3.13 kernel version. If your kernel version is older than 3.13, you must upgrade it. Refer to this table to see which packages are required for your environment:

| **Package** | **Description** |
| --- | --- |
| linux-image  generic-lts  trusty | Generic Linux kernel image. This kernel has AUFS built in. This is required to run Docker. |
| linux-headers generic-lts  trusty | Allows packages such as ZFS and VirtualBox guest additions which depend on them. If you didn’t install the headers for your existing kernel, then you can skip these headers for the trusty kernel. If you’re unsure, you should include this package for safety. |
| xserver-xorg  lts-trusty | Optional in non-graphical environments without Unity/Xorg. **Required** when running Docker on machine with a graphical environment. |

https://riptutorial.com/ 7

| **Package** | **Description** |
| --- | --- |
| ligbl1-mesa  glx-lts-trusty | To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack. Refer to note 5 under each version. |

To upgrade your kernel and install the additional packages, do the following: 1. Open a terminal on your Ubuntu host.

2. Update your package manager with the command sudo apt-get update.

3.

Install both the required and optional packages. $ sudo apt-get install linux-image-generic-lts-trusty

4. Repeat this step for other packages you need to install.

5. Reboot your host to use the updated kernel using the command sudo reboot. 6. After reboot, go ahead and install Docker.

**Install the latest version**

Make sure you satisfy the prerequisites, only then follow the below steps.

**Note:** For production systems, it is recommended that you install a specific version so that you do not accidentally update Docker. You should plan upgrades for production systems carefully.

1. Log into your Ubuntu installation as a user with sudo privileges. (Possibly running sudo -su). 2. Update your APT package index by running sudo apt-get update.

3. Install Docker Community Edition with the command sudo apt-get install docker-ce. 4. Start the docker daemon with the command sudo service docker start.

5.

Verify that docker is installed correctly by running the hello-world image. $ sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

**Manage Docker as a non-root user**

If you don’t want to use sudo when you use the docker command, create a Unix group called docker and add users to it. When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

https://riptutorial.com/ 8

To create the docker group and add your user:

1. Log into Ubuntu as a user with sudo privileges.

2. Create the docker group with the command sudo groupadd docker.

3.

Add your user to the docker group. $ sudo usermod -aG docker $USER

4. Log out and log back in so that your group membership is re-evaluated.

5.

Verify that you can docker commands without sudo permission. $ docker run hello-world

If this fails, you will see an error:

Cannot connect to the Docker daemon. Is 'docker daemon' running on this host? Check whether the DOCKER\_HOST environment variable is set for your shell.

$ env | grep DOCKER\_HOST

If it is set, the above command will return a result. If so, unset it.

$ unset DOCKER\_HOST

You may need to edit your environment in files such as ~/.bashrc or ~/.profile to prevent the DOCKER\_HOST variable from being set erroneously.

**Installing Docker on Ubuntu**

**Requirements:** Docker can be installed on any Linux with a kernel of at least version 3.10. Docker is supported on the following 64-bit versions of Ubuntu Linux:

• Ubuntu Xenial 16.04 (LTS)

• Ubuntu Wily 15.10

• Ubuntu Trusty 14.04 (LTS)

• Ubuntu Precise 12.04 (LTS)

**Easy Installation**

**Note: Installing Docker from the default Ubuntu repository will install an old version of Docker.**

To install the latest version of Docker using the Docker repository, use curl to grab and run the installation script provided by Docker:

https://riptutorial.com/ 9

$ curl -sSL https://get.docker.com/ | sh

Alternatively, wget can be used to install Docker:

$ wget -qO- https://get.docker.com/ | sh

Docker will now be installed.

**Manual Installation**

If, however, running the installation script is not an option, the following instructions can be used to manually install the latest version of Docker from the official repository.

$ sudo apt-get update

$ sudo apt-get install apt-transport-https ca-certificates

Add the GPG key:

$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \

--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

Next, open the /etc/apt/sources.list.d/docker.list file in your favorite editor. If the file doesn’t exist, create it. Remove any existing entries. Then, depending on your version, add the following line:

•

Ubuntu Precise 12.04 (LTS):

deb https://apt.dockerproject.org/repo ubuntu-precise main •

Ubuntu Trusty 14.04 (LTS)

deb https://apt.dockerproject.org/repo ubuntu-trusty main •

Ubuntu Wily 15.10

deb https://apt.dockerproject.org/repo ubuntu-wily main •

Ubuntu Xenial 16.04 (LTS)

deb https://apt.dockerproject.org/repo ubuntu-xenial main

Save the file and exit, then update your package index, uninstall any installed versions of Docker, and verify apt is pulling from the correct repo:

$ sudo apt-get update

$ sudo apt-get purge lxc-docker

$ sudo apt-cache policy docker-engine

Depending on your version of Ubuntu, some prerequisites may be required: • Ubuntu Xenial 16.04 (LTS), Ubuntu Wily 15.10, Ubuntu Trusty 14.04 (LTS)

https://riptutorial.com/ 10

sudo apt-get update && sudo apt-get install linux-image-extra-$(uname -r) •

Ubuntu Precise 12.04 (LTS)

This version of Ubuntu requires kernel version 3.13. You may need to install additional packages depending on your environment:

linux-image-generic-lts-trusty

Generic Linux kernel image. This kernel has AUFS built in. This is required to run Docker. linux-headers-generic-lts-trusty

Allows packages such as ZFS and VirtualBox guest additions which depend on them. If you didn’t install the headers for your existing kernel, then you can skip these headers for the trusty kernel. If you’re unsure, you should include this package for safety.

xserver-xorg-lts-trusty

libgl1-mesa-glx-lts-trusty

These two packages are optional in non-graphical environments without Unity/Xorg. Required when running Docker on machine with a graphical environment.

To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack — refer to note 5 under each version.

Install the required packages then reboot the host:

$ sudo apt-get install linux-image-generic-lts-trusty

$ sudo reboot

Finally, update the apt package index and install Docker:

$ sudo apt-get update

$ sudo apt-get install docker-engine

Start the daemon:

$ sudo service docker start

Now verify that docker is running properly by starting up a test image:

$ sudo docker run hello-world

This command should print a welcome message verifying that the installation was successful. **Create a docker container in Google Cloud**

You can use docker, without using docker daemon (engine), by using cloud providers. In this https://riptutorial.com/ 11

example, you should have a gcloud (Google Cloud util), that connected to your account

docker-machine create --driver google --google-project `your-project-name` google-machine-type f1-large fm02

This example will create a new instance, in your Google Cloud console. Using machine time f1- large

**Install Docker on Ubuntu**

Docker is supported on the following 64-bit versions of Ubuntu Linux:

• Ubuntu Xenial 16.04 (LTS)

• Ubuntu Wily 15.10

• Ubuntu Trusty 14.04 (LTS)

• Ubuntu Precise 12.04 (LTS)

A couple of notes:

The following instructions involve installation using **Docker** packages only, and this ensures obtaining the latest official release of **Docker**. If you need to install only using Ubuntu-managed packages, consult the Ubuntu documentation (Not recommended otherwise for obvious reasons).

Ubuntu Utopic 14.10 and 15.04 exist in Docker’s APT repository but are no longer officially supported due to known security issues.

**Prerequisites**

• Docker only works on a 64-bit installation of Linux.

•

Docker requires Linux kernel version 3.10 or higher (Except for Ubuntu Precise 12.04, which requires version 3.13 or higher). Kernels older than 3.10 lack some of the features required to run Docker containers and contain known bugs which cause data loss and frequently panic under certain conditions. Check current kernel version with the command uname -r. Check this post if you need to update your Ubuntu Precise (12.04 LTS) kernel by scrolling further down. Refer to this WikiHow post to obtain the latest version for other Ubuntu installations.

**Update APT sources**

This needs to be done so as to access packages from Docker repository.

1. Log into your machine as a user with sudo or root privileges. 2. Open a terminal window.

3.

Update package information, ensure that APT works with the https method, and that CA certificates are installed.

$ sudo apt-get update

$ sudo apt-get install apt-transport-https ca-certificates

https://riptutorial.com/ 12

4.

Add the new GPG key. This commands downloads the key with the ID

58118E89F3A912897C070ADBF76221572C52609D from the keyserver hkp://ha.pool.sks keyservers.net:80 and adds it to the adv keychain. For more information, see the output of man apt-key.

$ sudo apt-key adv \

--keyserver hkp://ha.pool.sks-keyservers.net:80 \

--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

5.

Find the entry in the table below which corresponds to your Ubuntu version. This determines where APT will search for Docker packages. When possible, run a long-term support (LTS) edition of Ubuntu.

| **Ubuntu Version** | **Repository** |
| --- | --- |
| Precise 12.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-precise main |
| Trusty 14.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-trusty main |
| Wily 15.10 | deb https://apt.dockerproject.org/repo ubuntu-wily main |
| Xenial 16.04 (LTS) | deb https://apt.dockerproject.org/repo ubuntu-xenial main |

**Note:** Docker does not provide packages for all architectures. Binary artifacts are built nightly, and you can download them from https://master.dockerproject.org. To install docker on a multi-architecture system, add an [arch=...] clause to the entry. Refer to Debian Multiarch wiki for details.

6.

Run the following command, substituting the entry for your operating system for the placeholder <REPO>.

$ echo "" | sudo tee /etc/apt/sources.list.d/docker.list

7. Update the APT package index by executing sudo apt-get update.

8. Verify that APT is pulling from the right repository.

When you run the following command, an entry is returned for each version of Docker that is available for you to install. Each entry should have the URL https://apt.dockerproject.org/repo/. The version currently installed is marked with \*\*\*.See the below example's output.

$ apt-cache policy docker-engine

docker-engine:

Installed: 1.12.2-0~trusty

Candidate: 1.12.2-0~trusty

Version table:

\*\*\* 1.12.2-0~trusty 0

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages 100 /var/lib/dpkg/status

1.12.1-0~trusty 0

https://riptutorial.com/ 13

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages 1.12.0-0~trusty 0

500 https://apt.dockerproject.org/repo/ ubuntu-trusty/main amd64 Packages

From now on when you run apt-get upgrade, APT pulls from the new repository. **Prerequisites by Ubuntu Version**

For Ubuntu Trusty (14.04) , Wily (15.10) , and Xenial (16.04) , install the linux-image-extra-\* kernel packages, which allows you use the aufs storage driver.

To install the linux-image-extra-\* packages:

1. Open a terminal on your Ubuntu host.

2. Update your package manager with the command sudo apt-get update.

3.

Install the recommended packages.

$ sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual

4. Proceed to Docker installation

For Ubuntu Precise (12.04 LTS), Docker requires the 3.13 kernel version. If your kernel version is older than 3.13, you must upgrade it. Refer to this table to see which packages are required for your environment:

| **Package** | **Description** |
| --- | --- |
| linux-image  generic-lts  trusty | Generic Linux kernel image. This kernel has AUFS built in. This is required to run Docker. |
| linux-headers generic-lts  trusty | Allows packages such as ZFS and VirtualBox guest additions which depend on them. If you didn’t install the headers for your existing kernel, then you can skip these headers for the trusty kernel. If you’re unsure, you should include this package for safety. |
| xserver-xorg  lts-trusty | Optional in non-graphical environments without Unity/Xorg. **Required** when running Docker on machine with a graphical environment. |
| ligbl1-mesa  glx-lts-trusty | To learn more about the reasons for these packages, read the installation instructions for backported kernels, specifically the LTS Enablement Stack. Refer to note 5 under each version. |

To upgrade your kernel and install the additional packages, do the following: 1. Open a terminal on your Ubuntu host.

2. Update your package manager with the command sudo apt-get update.

https://riptutorial.com/ 14

3.

Install both the required and optional packages. $ sudo apt-get install linux-image-generic-lts-trusty

4. Repeat this step for other packages you need to install.

5. Reboot your host to use the updated kernel using the command sudo reboot. 6. After reboot, go ahead and install Docker.

**Install the latest version**

Make sure you satisfy the prerequisites, only then follow the below steps.

**Note:** For production systems, it is recommended that you install a specific version so that you do not accidentally update Docker. You should plan upgrades for production systems carefully.

1. Log into your Ubuntu installation as a user with sudo privileges. (Possibly running sudo -su). 2. Update your APT package index by running sudo apt-get update.

3. Install Docker with the command sudo apt-get install docker-engine.

4. Start the docker daemon with the command sudo service docker start.

5.

Verify that docker is installed correctly by running the hello-world image. $ sudo docker run hello-world

This command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits.

**Manage Docker as a non-root user**

If you don’t want to use sudo when you use the docker command, create a Unix group called docker and add users to it. When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

To create the docker group and add your user:

1. Log into Ubuntu as a user with sudo privileges.

2. Create the docker group with the command sudo groupadd docker.

3.

Add your user to the docker group. $ sudo usermod -aG docker $USER

4. Log out and log back in so that your group membership is re-evaluated.

https://riptutorial.com/ 15

5.

Verify that you can docker commands without sudo permission. $ docker run hello-world

If this fails, you will see an error:

Cannot connect to the Docker daemon. Is 'docker daemon' running on this host? Check whether the DOCKER\_HOST environment variable is set for your shell.

$ env | grep DOCKER\_HOST

If it is set, the above command will return a result. If so, unset it.

$ unset DOCKER\_HOST

You may need to edit your environment in files such as ~/.bashrc or ~/.profile to prevent the DOCKER\_HOST variable from being set erroneously.

**Installating Docker-ce OR Docker-ee on CentOS**

Docker has announced following editions:

-Docker-ee (Enterprise Edition) along with Docker-ce(Community Edition) and Docker (Commercial Support)

This document will help you with installation steps of Docker-ee and Docker-ce edition in CentOS **Docker-ce Installation**

Following are steps to install docker-ce edition

1.

Install yum-utils, which provides yum-config-manager utility:

$ sudo yum install -y yum-utils

2.

Use the following command to set up the stable repository:

$ sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

3.

Optional: Enable the edge repository. This repository is included in the docker.repo file above but is disabled by default. You can enable it alongside the stable repository.

$ sudo yum-config-manager --enable docker-ce-edge

https://riptutorial.com/ 16

•

You can disable the edge repository by running the yum-config-manager command with the -- disable flag. To re-enable it, use the --enable flag. The following command disables the edge repository.

$ sudo yum-config-manager --disable docker-ce-edge

4.

Update the yum package index.

$ sudo yum makecache fast

5.

Install the docker-ce using following command:

$ sudo yum install docker-ce-17.03.0.ce

6.

Confirm the Docker-ce fingerprint

060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35

If you want to install some other version of docker-ce you can use following command: $ sudo yum install docker-ce-VERSION

Specify the VERSION number

7.

If everything went well the docker-ce is now installed in your system, use following command to start:

$ sudo systemctl start docker

8.

Test your docker installation:

$ sudo docker run hello-world

you should get following message:

Hello from Docker!

This message shows that your installation appears to be working correctly.

**-Docker-ee (Enterprise Edition) Installation**

For Enterprise Edition (EE) it would be required to signup, to get your <DOCKER-EE-URL>.

1.

To signup go to https://cloud.docker.com/. Enter your details and confirm your email id. After confirmation you would be given a <DOCKER-EE-URL>, which you can see in your dashboard after clicking on setup.

2. Remove any existing Docker repositories from /etc/yum.repos.d/

https://riptutorial.com/ 17

3.

Store your Docker EE repository URL in a yum variable in /etc/yum/vars/. Replace <DOCKER-EE-URL> with the URL you noted down in the first step.

$ sudo sh -c 'echo "<DOCKER-EE-URL>" > /etc/yum/vars/dockerurl'

4.

Install yum-utils, which provides the yum-config-manager utility:

$ sudo yum install -y yum-utils

5.

Use the following command to add the stable repository:

$ sudo yum-config-manager \

--add-repo \

<DOCKER-EE-URL>/docker-ee.repo

6.

Update the yum package index.

$ sudo yum makecache fast

7.

Install docker-ee

sudo yum install docker-ee

8.

You can start the docker-ee using following command:

$ sudo systemctl start docker

Read Getting started with Docker online: https://riptutorial.com/docker/topic/658/getting-started with-docker

https://riptutorial.com/ 18

**Chapter 2: Building images**

**Parameters**

| **Parameter** | **Details** |
| --- | --- |
| --pull | Ensures that the base image (FROM) is up-to-date before building the rest of the Dockerfile. |

**Examples**

**Building an image from a Dockerfile**

Once you have a Dockerfile, you can build an image from it using docker build. The basic form of this command is:

docker build -t *image-name path*

If your Dockerfile isn't named Dockerfile, you can use the -f flag to give the name of the Dockerfile to build.

docker build -t image-name -f Dockerfile2 .

For example, to build an image named dockerbuild-example:1.0.0 from a Dockerfile in the current working directory:

$ ls

Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .

See the docker build usage documentation for more options and settings.

A common mistake is creating a Dockerfile in the user home directory (~). This is a bad idea because during docker build -t mytag . this message will appear for a long time:

Uploading context

The cause is the docker daemon trying to copy all the user's files (both the home directory and it's subdirectories). Avoid this by always specifying a directory for the Dockerfile.

Adding a .dockerignore file to the build directory is a good practice. Its syntax is similar to .gitignore files and will make sure only wanted files and directories are uploaded as the context of the build.

https://riptutorial.com/ 19

**A simple Dockerfile**

FROM node:5

The FROM directive specifies an image to start from. Any valid image reference may be used. WORKDIR /usr/src/app

The WORKDIR directive sets the current working directory inside the container, equivalent to running cd inside the container. (Note: RUN cd will not change the current working directory.)

RUN npm install cowsay knock-knock-jokes

RUN executes the given command inside the container.

COPY cowsay-knockknock.js ./

COPY copies the file or directory specified in the first argument from the build context (the *path* passed to docker build *path*) to the location in the container specified by the second argument.

CMD node cowsay-knockknock.js

CMD specifies a command to execute when the image is run and no command is given. It can be overridden by passing a command to docker run.

There are many other instructions and options; see the Dockerfile reference for a complete list. **Difference between ENTRYPOINT and CMD**

There are two Dockerfile directives to specify what command to run by default in built images. If you only specify CMD then docker will run that command using the default ENTRYPOINT, which is /bin/sh -c. You can override either or both the entrypoint and/or the command when you start up the built image. If you specify both, then the ENTRYPOINT specifies the executable of your container process, and CMD will be supplied as the parameters of that executable.

For example if your Dockerfile contains

FROM ubuntu:16.04

CMD ["/bin/date"]

Then you are using the default ENTRYPOINT directive of /bin/sh -c, and running /bin/date with that default entrypoint. The command of your container process will be /bin/sh -c /bin/date. Once you run this image then it will by default print out the current date

$ docker build -t test .

$ docker run test

Tue Jul 19 10:37:43 UTC 2016

https://riptutorial.com/ 20

You can override CMD on the command line, in which case it will run the command you have specified.

$ docker run test /bin/hostname

bf0274ec8820

If you specify an ENTRYPOINT directive, Docker will use that executable, and the CMD directive specifies the default parameter(s) of the command. So if your Dockerfile contains:

FROM ubuntu:16.04

ENTRYPOINT ["/bin/echo"]

CMD ["Hello"]

Then running it will produce

$ docker build -t test .

$ docker run test

Hello

You can provide different parameters if you want to, but they will all run /bin/echo

$ docker run test Hi

Hi

If you want to override the entrypoint listed in your Dockerfile (i.e. if you wish to run a different command than echo in this container), then you need to specify the --entrypoint parameter on the command line:

$ docker run --entrypoint=/bin/hostname test

b2c70e74df18

Generally you use the ENTRYPOINT directive to point to your main application you want to run, and CMD to the default parameters.

**Exposing a Port in the Dockerfile**

EXPOSE <port> [<port>...]

From Docker's documentation:

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. EXPOSE does not make the ports of the container accessible to the host. To do that, you must use either the -p flag to publish a range of ports or the -P flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

**Example:**

https://riptutorial.com/ 21

Inside your Dockerfile:

EXPOSE 8765

To access this port from the host machine, include this argument in your docker run command: -p 8765:8765

**ENTRYPOINT and CMD seen as verb and parameter**

Suppose you have a Dockerfile ending with

ENTRYPOINT [ "nethogs"] CMD ["wlan0"]

if you build this image with a

docker built -t inspector .

launch the image built with such a Dockerfile with a command such as

docker run -it --net=host --rm inspector

,nethogs will monitor the interface named wlan0

Now if you want to monitor the interface eth0 (or wlan1, or ra1...), you will do something like docker run -it --net=host --rm inspector eth0

or

docker run -it --net=host --rm inspector wlan1

**Pushing and Pulling an Image to Docker Hub or another Registry**

Locally created images can be pushed to Docker Hub or any other docker repo host, known as a registry. Use docker login to sign in to an existing docker hub account.

docker login

Login with your Docker ID to push and pull images from Docker Hub.

If you don't have a Docker ID, head over to https://hub.docker.com to create one.

Username: cjsimon

Password:

Login Succeeded

A different docker registry can be used by specifying a server name. This also works for private or self-hosted registries. Further, using an external credentials store for safety is possible.

docker login quay.io

You can then tag and push images to the registry that you are logged in to. Your repository must https://riptutorial.com/ 22

be specified as server/username/reponame:tag. Omitting the server currently defaults to Docker Hub. (The default registry cannot be changed to another provider, and there are no plans to implement this feature.)

docker tag mynginx quay.io/cjsimon/mynginx:latest

Different tags can be used to represent different versions, or branches, of the same image. An image with multiple different tags will display each tag in the same repo.

Use docker images to see a list of installed images installed on your local machine, including your newly tagged image. Then use push to upload it to the registry and pull to download the image.

docker push quay.io/cjsimon/mynginx:latest

All tags of an images can be pulled by specifying the -a option

docker pull quay.io/cjsimon/mynginx:latest

**Building using a proxy**

Often when building a Docker image, the Dockerfile contains instructions that runs programs to fetch resources from the Internet (wget for example to pull a program binary build on GitHub for example).

It is possible to instruct Docker to pass set set environment variables so that such programs perform those fetches through a proxy:

$ docker build --build-arg http\_proxy=http://myproxy.example.com:3128 \

--build-arg https\_proxy=http://myproxy.example.com:3128 \

--build-arg no\_proxy=internal.example.com \

-t test .

build-arg are environment variables which are available at build time only.

Read Building images online: https://riptutorial.com/docker/topic/713/building-images https://riptutorial.com/ 23

**Chapter 3: Checkpoint and Restore Containers**

**Examples**

**Compile docker with checkpoint and restore enabled (ubuntu)**

In order to compile docker its recommended you have at least **2 GB RAM**. Even with that it fails sometimes so its better to go for **4GB** instead.

1.

make sure git and make is installed

sudo apt-get install make git-core -y

2.

install a new kernel (at least 4.2)

sudo apt-get install linux-generic-lts-xenial

3.

reboot machine to have the new kernel active

sudo reboot

4.

compile criu which is needed in order to run docker checkpoint

sudo apt-get install libprotobuf-dev libprotobuf-c0-dev protobuf-c-compiler protobuf compiler python-protobuf libnl-3-dev libcap-dev -y

wget http://download.openvz.org/criu/criu-2.4.tar.bz2 -O - | tar -xj

cd criu-2.4

make

make install-lib

make install-criu

5.

check if every requirement is fulfilled to run criu

sudo criu check

6.

compile experimental docker ( we need docker to compile docker)

cd ~

wget -qO- https://get.docker.com/ | sh

sudo usermod -aG docker $(whoami)

•

**At this point we have to logoff and login again to have a docker daemon. After relog continue with compile step**

https://riptutorial.com/ 24

git clone https://github.com/boucher/docker

cd docker

git checkout docker-checkpoint-restore

make #that will take some time - drink a coffee

DOCKER\_EXPERIMENTAL=1 make binary

7.

We now have a compiled docker. Lets move the binaries. Make sure to replace <version> with the version installed

sudo service docker stop

sudo cp $(which docker) $(which docker)\_ ; sudo cp ./bundles/latest/binary-client/docker- <version>-dev $(which docker)

sudo cp $(which docker-containerd) $(which docker-containerd)\_ ; sudo cp ./bundles/latest/binary-daemon/docker-containerd $(which docker-containerd) sudo cp $(which docker-containerd-ctr) $(which docker-containerd-ctr)\_ ; sudo cp ./bundles/latest/binary-daemon/docker-containerd-ctr $(which docker-containerd-ctr) sudo cp $(which docker-containerd-shim) $(which docker-containerd-shim)\_ ; sudo cp ./bundles/latest/binary-daemon/docker-containerd-shim $(which docker-containerd-shim) sudo cp $(which dockerd) $(which dockerd)\_ ; sudo cp ./bundles/latest/binary daemon/dockerd $(which dockerd)

sudo cp $(which docker-runc) $(which docker-runc)\_ ; sudo cp ./bundles/latest/binary daemon/docker-runc $(which docker-runc)

sudo service docker start

Dont worry - we backed up the old binaries. They are still there but with an underscore added to its names (docker\_).

Congratulation you now have an experimental docker with the ability to checkpoint a container and restore it.

**Please note that experimental features are NOT ready for production**

**Checkpoint and Restore a Container**

# create docker container

export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while true; do echo $i; i=$(expr $i + 1); sleep 1; done')

# container is started and prints a number every second

# display the output with

docker logs $cid

# checkpoint the container

docker checkpoint create $cid checkpointname

# container is not running anymore

docker np

# lets pass some time to make sure

# resume container

docker start $cid --checkpoint=checkpointname

# print logs again

docker logs $cid

https://riptutorial.com/ 25

Read Checkpoint and Restore Containers online:

https://riptutorial.com/docker/topic/5291/checkpoint-and-restore-containers https://riptutorial.com/ 26

**Chapter 4: Concept of Docker Volumes Remarks**

People new to Docker often don't realize that Docker filesystems are temporary by default. If you start up a Docker image you'll get a container that on the surface behaves much like a virtual machine. You can create, modify, and delete files. However, unlike a virtual machine, if you stop the container and start it up again, all your changes will be lost -- any files you previously deleted will now be back, and any new files or edits you made won't be present.

Volumes in docker containers allow for persistent data, and for sharing host-machine data inside a container.

**Examples**

**A) Launch a container with a volume**

[root@localhost ~]# docker run -it -v /data --name=vol3 8251da35e7a7 /bin/bash root@d87bf9607836:/# cd /data/

root@d87bf9607836:/data# touch abc{1..10}

root@d87bf9607836:/data# ls

abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

**B) Now press [cont +P+Q] to move out from container without terminating the container checking for container that is running**

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES d87bf9607836 8251da35e7a7 "/bin/bash" About a minute ago Up 31 seconds vol3 [root@localhost ~]#

**C) Run 'docker inspect' to check out more info about the volume**

[root@localhost ~]# docker inspect d87bf9607836

"Mounts": [ { "Name":

"cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c", "Source": "/var/lib/docker/volumes/cdf78fbf79a7c9363948e133abe4c572734cd788c95d36edea0448094ec9121c/\_d"Destination": "/data", "Driver": "local", "Mode": "", "RW": true

**D) You can attach a running containers volume to another containers** [root@localhost ~]# docker run -it --volumes-from vol3 8251da35e7a7 /bin/bash

https://riptutorial.com/ 27

root@ef2f5cc545be:/# ls

bin boot data dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var root@ef2f5cc545be:/# ls /data abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9 **E) You can also mount you base directory inside container**

[root@localhost ~]# docker run -it -v /etc:/etc1 8251da35e7a7 /bin/bash

Here: /etc is host machine directory and /etc1 is the target inside container

Read Concept of Docker Volumes online: https://riptutorial.com/docker/topic/5908/concept-of docker-volumes

https://riptutorial.com/ 28

**Chapter 5: Connecting Containers**

**Parameters**

| **Parameter** | **Details** |
| --- | --- |
| tty:true | In docker-compose.yml, the tty: true flag keeps the container's sh command running waiting for input. |

**Remarks**

The host and bridge network drivers are able to connect containers on a single docker host. To allow containers to communicate beyond one machine, create an overlay network. Steps to create the network depend on how your docker hosts are managed.

• Swarm Mode: docker network create --driver overlay

• docker/swarm: requires an external key-value store

**Examples**

**Docker network**

Containers in the same docker network have access to exposed ports.

docker network create sample

docker run --net sample --name keys consul agent -server -client=0.0.0.0 -bootstrap

Consul's Dockerfile exposes 8500, 8600, and several more ports. To demonstrate, run another container in the same network:

docker run --net sample -ti alpine sh

/ # wget -qO- keys:8500/v1/catalog/nodes

Here the consul container is resolved from keys, the name given in the first command. Docker provides dns resolution on this network, to find containers by their --name.

**Docker-compose**

Networks can be specified in a compose file (v2). By default all the containers are in a shared network.

Start with this file: example/docker-compose.yml:

version: '2'

https://riptutorial.com/ 29

services:

keys:

image: consul

command: agent -server -client=0.0.0.0 -bootstrap

test:

image: alpine

tty: true

command: sh

Starting this stack with docker-compose up -d will create a network named after the parent directory, in this case example\_default. Check with docker network ls

> docker network ls

NETWORK ID NAME DRIVER SCOPE

719eafa8690b example\_default bridge local

Connect to the alpine container to verify the containers can resolve and communicate:

> docker exec -ti example\_test\_1 sh

/ # nslookup keys

...

/ # wget -qO- keys:8500/v1/kv/?recurse

...

A compose file can have a networks: top level section to specify the network name, driver, and other options from the docker network command.

**Container Linking**

The docker --link argument, and link: sections docker-compose make aliases to other containers.

docker network create sample

docker run -d --net sample --name redis redis

With link either the original name or the mapping will resolve the redis container.

> docker run --net sample --link redis:cache -ti python:alpine sh -c "pip install redis && python"

>>> import redis

>>> r = redis.StrictRedis(host='cache')

>>> r.set('key', 'value')

True

Before docker 1.10.0 container linking also setup network connectivity - behavior now provided by docker network. Links in later versions only provide legacy effect on the default bridge network.

Read Connecting Containers online: https://riptutorial.com/docker/topic/6528/connecting containers

https://riptutorial.com/ 30

**Chapter 6: Creating a service with persistence**

**Syntax**

• docker volume create --name <volume\_name> # Creates a volume called <volume\_name>

•

docker run -v <volume\_name>:<mount\_point> -d crramirez/limesurvey:latest # Mount the <volume\_name> volume in <mount\_point> directory in the container

**Parameters**

| **Parameter** | **Details** |
| --- | --- |
| --name <volume\_name> | Specify the volume name to be created |
| -v  <volume\_name>:<mount\_point> | Specify where the named volume will be mounted in the container |

**Remarks**

Persistence is created in docker containers using volumes. Docker have many ways to deal with volumes. Named volumes are very convenient by:

• They persist even when the container is removed using the -v option.

• The only way to delete a named volume is doing an explicit call to docker volume rm

•

The named volumes can be shared among container without linking or --volumes-from option.

• They don't have permission issues that host mounted volumes have.

• They can be manipulated using docker volume command.

**Examples**

**Persistence with named volumes**

Persistence is created in docker containers using volumes. Let's create a Limesurvey container and persist the database, uploaded content and configuration in a named volume:

docker volume create --name mysql

docker volume create --name upload

docker run -d --name limesurvey -v mysql:/var/lib/mysql -v upload:/app/upload -p 80:80 crramirez/limesurvey:latest

https://riptutorial.com/ 31

**Backup a named volume content**

We need to create a container to mount the volume. Then archive it and download the archive to our host.

Let's create first a data volume with some data:

docker volume create --name=data

echo "Hello World" | docker run -i --rm=true -v data:/data ubuntu:trusty tee /data/hello.txt Let's backup the data:

docker run -d --name backup -v data:/data ubuntu:trusty tar -czvf /tmp/data.tgz /data docker cp backup:/tmp/data.tgz data.tgz

docker rm -fv backup

Let's test:

tar -xzvf data.tgz

cat data/hello.txt

Read Creating a service with persistence online: https://riptutorial.com/docker/topic/7429/creating a-service-with-persistence

https://riptutorial.com/ 32

**Chapter 7: Data Volumes and Data**

**Containers**

**Examples**

**Data-Only Containers**

**Data-only containers are obsolete and are now considered an anti-pattern!**

In the days of yore, before Docker's volume subcommand, and before it was possible to create named volumes, Docker deleted volumes when there were no more references to them in any containers. Data-only containers are obsolete because Docker now provides the ability to create named volumes, as well as much more utility via the various docker volume subcommand. Data only containers are now considered an anti-pattern for this reason.

Many resources on the web from the last couple of years mention using a pattern called a "data only container", which is simply a Docker container that exists only to keep a reference to a data volume around.

Remember that in this context, a "data volume" is a Docker volume which is not mounted from the host. To clarify, a "data volume" is a volume which is created either with the VOLUME Dockerfile directive, or using the -v switch on the command line in a docker run command, specifically with the format -v /path/on/container. Therefore a "data-only container" is a container whose only purpose is to have a data volume attached, which is used by the --volumes-from flag in a docker run command. For example:

docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true

When the above command is run, a "data-only container" is created. It is simply an empty container which has a data volume attached. It was then possible to use this volume in another container like so:

docker run -d --name="mysql" --volumes-from="mysql-data" mysql

The mysql container now has the same volume in it that is also in mysql-data.

Because Docker now provides the volume subcommand and named volumes, this pattern is now obsolete and not recommended.

To get started with the volume subcommand and named volumes see Creating a named volume **Creating a data volume**

docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql

https://riptutorial.com/ 33

This command creates a new container from the mysql image. It also creates a new data volume, which it then mounts in the container at /var/lib/mysql. This volume helps any data inside of it persist beyond the lifetime of the container. That is to say, when a container is removed, its filesystem changes are also removed. If a database was storing data in the container, and the container is removed, all of that data is also removed. Volumes will persist a particular location even beyond when its container is removed.

It is possible to use the same volume in multiple containers with the --volumes-from command line option:

docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql

The mysql-2 container now has the data volume from mysql-1 attached to it, also using the path /var/lib/mysql.

Read Data Volumes and Data Containers online: https://riptutorial.com/docker/topic/3224/data volumes-and-data-containers

https://riptutorial.com/ 34

**Chapter 8: Debugging a container**

**Syntax**

• docker stats [OPTIONS] [CONTAINER...]

• docker logs [OPTIONS] CONTAINER

• docker top [OPTIONS] CONTAINER [ps OPTIONS]

**Examples**

**Entering in a running container**

To execute operations in a container, use the docker exec command. Sometimes this is called "entering the container" as all commands are executed inside the container.

docker exec -it container\_id bash

or

docker exec -it container\_id /bin/sh

And now you have a shell in your running container. For example, list files in a directory and then leave the container:

docker exec container\_id ls -la

You can use the -u flag to enter the container with a specific user, e.g. uid=1013, gid=1023. docker exec -it -u 1013:1023 container\_id ls -la

The uid and gid does not have to exist in the container but the command can result in errors.If you want to launch a container and immediately enter inside in order to check something, you can do

docker run...; docker exec -it $(docker ps -lq) bash

the command docker ps -lq outputs only the id of the last (the l in -lq) container started. (this supposes you have bash as interpreter available in your container, you may have sh or zsh or any other)

**Monitoring resource usage**

Inspecting system resource usage is an efficient way to find misbehaving applications. This example is an equivalent of the traditional top command for containers:

docker stats

https://riptutorial.com/ 35

To follow the stats of specific containers, list them on the command line:

docker stats 7786807d8084 7786807d8085

Docker stats displays the following information:

CONTAINER CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O 7786807d8084 0.65% 1.33 GB / 3.95 GB 33.67% 142.2 MB / 57.79 MB 46.32 MB / 0 B

By default docker stats displays the id of the containers, and this is not very helpful, if your prefer to display the names of the container, just do

docker stats $(docker ps --format '{{.Names}}')

**Monitoring processes in a container**

Inspecting system resource usage is an efficient way to narrow down a problem on a live running application. This example is an equivalent of the traditional ps command for containers.

docker top 7786807d8084

To filter of format the output, add ps options on the command line:

docker top 7786807d8084 faux

Or, to get the list of processes running as root, which is a potentially harmful practice: docker top 7786807d8084 -u root

The docker top command proves especially useful when troubleshooting minimalistic containers without a shell or the ps command.

**Attach to a running container**

'Attaching to a container' is the act of starting a terminal session within the context that the container (and any programs therein) is running. This is primarily used for debugging purposes, but may also be needed if specific data needs to be passed to programs running within the container.

The attach command is utilized to do this. It has this syntax:

docker attach <container>

<container> can be either the container id or the container name. For instance: docker attach c8a9cf1a1fa8

https://riptutorial.com/ 36

Or:

docker attach graceful\_hopper

You may need to sudo the above commands, depending on your user and how docker is set up. Note: Attach only allows a single shell session to be attached to a container at a time.

Warning: all keyboard input will be forwarded to the container. Hitting Ctrl-c will kill your container.

To detach from an attached container, successively hit Ctrl-p then Ctrl-q

To attach multiple shell sessions to a container, or simply as an alternative, you can use exec. Using the container id:

docker exec -i -t c8a9cf1a1fa8 /bin/bash

Using the container's name:

docker exec -i -t graceful\_hopper /bin/bash

exec will run a program within a container, in this case /bin/bash (a shell, presumably one the container has). -i indicates an interactive session, while -t allocates a pseudo-TTY.

Note: Unlike attach, hitting Ctrl-c will only terminate the exec'd command when running interactively.

**Printing the logs**

Following the logs is the less intrusive way to debug a live running application. This example reproduces the behavior of the traditional tail -f some-application.log on container 7786807d8084.

docker logs --follow --tail 10 7786807d8084

This command basically shows the standard output of the container process (the process with pid 1).

If your logs do not natively include timestamping, you may add the --timestamps flag. It is possible to look at the logs of a stopped container, either

• start the failing container with docker run ... ; docker logs $(docker ps -lq) • find the container id or name with

docker ps -a

and then

https://riptutorial.com/ 37

docker logs container-id or

docker logs containername

as it is possible to look at the logs of a stopped container

**Docker container process debugging**

Docker is just a fancy way to run a process, not a virtual machine. Therefore, debugging a process "in a container" is also possible "on the host" by simply examining the running container process as a user with the appropriate permissions to inspect those processes on the host (e.g. root). For example, it's possible to list every "container process" on the host by running a simple ps as root:

sudo ps aux

Any currently running Docker containers will be listed in the output.

This can be useful during application development for debugging a process running in a container. As a user with appropriate permissions, typical debugging utilities can be used on the container process, such as strace, ltrace, gdb, etc.

Read Debugging a container online: https://riptutorial.com/docker/topic/1333/debugging-a container

https://riptutorial.com/ 38

**Chapter 9: Docker Data Volumes**

**Introduction**

Docker data volumes provide a way to persist data independent of a container's life cycle. Volumes present a number of helpful features such as:

Mounting a host directory within the container, sharing data in-between containers using the filesystem and preserving data if a container gets deleted

**Syntax**

• docker volume [OPTIONS] [COMMAND]

**Examples**

**Mounting a directory from the local host into a container**

It is possible to mount a host directory to a specific path in your container using the -v or --volume command line option. The following example will mount /etc on the host to /mnt/etc in the container:

(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd

(on windows) docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd

The default access to the volume inside the container is read-write. To mount a volume read-only inside of a container, use the suffix :ro:

docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd

**Creating a named volume**

docker volume create --name="myAwesomeApp"

Using a named volume makes managing volumes much more human-readable. It is possible to create a named volume using the command specified above, but it's also possible to create a named volume inside of a docker run command using the -v or --volume command line option:

docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3

Note that creating a named volume in this form is similar to mounting a host file/directory as a volume, except that instead of a valid path, the volume name is specified. Once created, named volumes can be shared with other containers:

https://riptutorial.com/ 39

docker run -d --name="myApp-2" --volumes-from "myApp-1" myApp:1.5.3

After running the above command, a new container has been created with the name myApp-2 from the myApp:1.5.3 image, which is sharing the myAwesomeApp named volume with myApp-1. The myAwesomeApp named volume is mounted at /data/app in the myApp-2 container, just as it is mounted at /data/app in the myApp-1 container.

Read Docker Data Volumes online: https://riptutorial.com/docker/topic/1318/docker-data-volumes https://riptutorial.com/ 40

**Chapter 10: Docker Engine API**

**Introduction**

An API that allows you to control every aspect of Docker from within your own applications, build tools to manage and monitor applications running on Docker, and even use it to build apps on Docker itself.

**Examples**

**Enable Remote access to Docker API on Linux**

Edit /etc/init/docker.conf and update the DOCKER\_OPTS variable to the following: DOCKER\_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'

Restart Docker deamon

service docker restart

Verify if Remote API is working

curl -X GET http://localhost:4243/images/json

**Enable Remote access to Docker API on Linux running systemd**

Linux running systemd, like Ubuntu 16.04, adding -H tcp://0.0.0.0:2375 to /etc/default/docker does not have the effect it used to.

Instead, create a file called /etc/systemd/system/docker-tcp.socket to make docker available on a TCP socket on port 4243:

[Unit]

Description=Docker Socket for the API

[Socket]

ListenStream=4243

Service=docker.service

[Install]

WantedBy=sockets.target

Then enable the new socket:

systemctl enable docker-tcp.socket

systemctl enable docker.socket

systemctl stop docker

systemctl start docker-tcp.socket

systemctl start docker

https://riptutorial.com/ 41

Now, verify if Remote API is working:

curl -X GET http://localhost:4243/images/json

**Enable Remote Access with TLS on Systemd**

Copy the package installer unit file to /etc where changes will not be overwritten on an upgrade: cp /lib/systemd/system/docker.service /etc/systemd/system/docker.service

Update /etc/systemd/system/docker.service with your options on ExecStart:

ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 \

--tlsverify --tlscacert=/etc/docker/certs/ca.pem \

--tlskey=/etc/docker/certs/key.pem \

--tlscert=/etc/docker/certs/cert.pem

Note that dockerd is the 1.12 daemon name, prior it was docker daemon. Also note that 2376 is dockers standard TLS port, 2375 is the standard unencrypted port. See this page for steps to create your own TLS self signed CA, cert, and key.

After making changes to the systemd unit files, run the following to reload the systemd config: systemctl daemon-reload

And then run the following to restart docker:

systemctl restart docker

It's a bad idea to skip TLS encryption when exposing the Docker port since anyone with network access to this port effectively has full root access on the host.

**Image pulling with progress bars, written in Go**

Here is an example of image pulling using Go and Docker Engine API and the same progress bars as the ones shown when you run docker pull your\_image\_name in the CLI. For the purposes of the progress bars are used some ANSI codes.

package yourpackage

import (

"context"

"encoding/json"

"fmt"

"io"

"strings"

"github.com/docker/docker/api/types"

"github.com/docker/docker/client"

)

https://riptutorial.com/ 42

// Struct representing events returned from image pulling

type pullEvent struct {

ID string `json:"id"`

Status string `json:"status"`

Error string `json:"error,omitempty"`

Progress string `json:"progress,omitempty"`

ProgressDetail struct {

Current int `json:"current"`

Total int `json:"total"`

} `json:"progressDetail"`

}

// Actual image pulling function

func PullImage(dockerImageName string) bool {

client, err := client.NewEnvClient()

if err != nil {

panic(err)

}

resp, err := client.ImagePull(context.Background(), dockerImageName,

types.ImagePullOptions{})

if err != nil {

panic(err)

}

cursor := Cursor{}

layers := make([]string, 0)

oldIndex := len(layers)

var event \*pullEvent

decoder := json.NewDecoder(resp)

fmt.Printf("\n")

cursor.hide()

for {

if err := decoder.Decode(&event); err != nil {

if err == io.EOF {

break

}

panic(err)

}

imageID := event.ID

// Check if the line is one of the final two ones

if strings.HasPrefix(event.Status, "Digest:") || strings.HasPrefix(event.Status, "Status:") {

fmt.Printf("%s\n", event.Status)

continue

}

// Check if ID has already passed once

index := 0

for i, v := range layers {

if v == imageID {

index = i + 1

https://riptutorial.com/ 43

break

}

}

// Move the cursor

if index > 0 {

diff := index - oldIndex

if diff > 1 {

down := diff - 1

cursor.moveDown(down)

} else if diff < 1 {

up := diff\*(-1) + 1

cursor.moveUp(up)

}

oldIndex = index

} else {

layers = append(layers, event.ID)

diff := len(layers) - oldIndex

if diff > 1 {

cursor.moveDown(diff) // Return to the last row

}

oldIndex = len(layers)

}

cursor.clearLine()

if event.Status == "Pull complete" {

fmt.Printf("%s: %s\n", event.ID, event.Status)

} else {

fmt.Printf("%s: %s %s\n", event.ID, event.Status, event.Progress) }

}

cursor.show()

if strings.Contains(event.Status, fmt.Sprintf("Downloaded newer image for %s", dockerImageName)) {

return true

}

return false

}

For better readability, cursor actions with the ANSI codes are moved to a separate structure, which looks like this:

package yourpackage

import "fmt"

// Cursor structure that implements some methods

// for manipulating command line's cursor

type Cursor struct{}

func (cursor \*Cursor) hide() {

https://riptutorial.com/ 44

fmt.Printf("\033[?25l")

}

func (cursor \*Cursor) show() {

fmt.Printf("\033[?25h")

}

func (cursor \*Cursor) moveUp(rows int) {

fmt.Printf("\033[%dF", rows)

}

func (cursor \*Cursor) moveDown(rows int) {

fmt.Printf("\033[%dE", rows)

}

func (cursor \*Cursor) clearLine() {

fmt.Printf("\033[2K")

}

After that in your main package you can call the PullImage function passing the image name you want to pull. Of course, before calling it, you have to be logged into the Docker registry, where the image is.

**Making a cURL request with passing some complex structure**

When using cURL for some queries to the Docker API, it might be a bit tricky to pass some complex structures. Let's say, getting a list of images allows using filters as a query parameter, which have to be a JSON representation of map[string][]string (about the maps in Go you can find more here). Here is how to achieve this:

curl --unix-socket /var/run/docker.sock \

-XGET "http:/v1.29/images/json" \

-G \

--data-urlencode 'filters={"reference":{"yourpreciousregistry.com/path/to/image": true}, "dangling":{"true": true}}'

Here the -G flag is used to specify that the data in the --data-urlencode parameter will be used in an HTTP GET request instead of the POST request that otherwise would be used. The data will be appended to the URL with a ? separator.

Read Docker Engine API online: https://riptutorial.com/docker/topic/3935/docker-engine-api https://riptutorial.com/ 45

**Chapter 11: Docker events**

**Examples**

**Launch a container and be notified of related events**

The documentation for docker events provides details, but when debugging it may be useful to launch a container and be notified immediately of any related event:

docker run... & docker events --filter 'container=$(docker ps -lq)'

In docker ps -lq, the l stands for last, and the q for quiet. This removes the id of the last container launched, and creates a notification immediately if the container dies or has another event occur.

Read Docker events online: https://riptutorial.com/docker/topic/6200/docker-events https://riptutorial.com/ 46

**Chapter 12: Docker in Docker**

**Examples**

**Jenkins CI Container using Docker**

This chapter describes how to set up a Docker Container with Jenkins inside, which is capable of sending Docker commands to the Docker installation (the Docker Daemon) of the Host. Effectively using Docker in Docker. To achieve this, we have to build a custom Docker Image which is based on an arbitrary version of the official Jenkins Docker Image. The Dockerfile (The Instruction how to build the Image) looks like this:

FROM jenkins

USER root

RUN cd /usr/local/bin && \

curl https://master.dockerproject.org/linux/amd64/docker > docker && \

chmod +x docker && \

groupadd -g 999 docker && \

usermod -a -G docker jenkins

USER Jenkins

This Dockerfile builds an Image which contains the Docker client binaries this client is used to communicate with a Docker Daemon. In this case the Docker Daemon of the Host. The RUN statement in this file also creates an UNIX usergroup with the UID 999 and adds the user Jenkins to it. Why exactly this is necessary is described in the further chapter. With this Image we can run a Jenkins server which can use Docker commands, but if we just run this Image the Docker client we installed inside the image cannot communicate with the Docker Daemon of the Host. These two components do communicate via a UNIX Socket /var/run/docker.sock. On Unix this is a file like everything else, so we can easily mount it inside the Jenkins Container. This is done with the command docker run -v /var/run/docker.sock:/var/run/docker.sock --name jenkins MY\_CUSTOM\_IMAGE\_NAME. But this mounted file is owned by docker:rootand because of this does the Dockerfile create this group with a well know UID and adds the Jenkins user to it. Now is the Jenkins Container really capable of running and using Docker. In production the run command should also contain -v jenkins\_home:/var/jenkins\_home to backup the Jenkins\_home directory and of course a port-mapping to access the server over network.

Read Docker in Docker online: https://riptutorial.com/docker/topic/8012/docker-in-docker https://riptutorial.com/ 47

**Chapter 13: docker inspect getting various fields for key:value and elements of list**

**Examples**

**various docker inspect examples**

I find that the examples in the docker inspect documentation seem magic, but do not explain much.

Docker inspect is important because it is the clean way to extract information from a running container docker inspect -f ... container\_id

(or all running container)

docker inspect -f ... $(docker ps -q)

avoiding some unreliable

docker command | grep or awk | tr or cut

When you launch a docker inspect you can get the values from the "top-level" easily, with a basic syntax like, for a container running htop (from https://hub.docker.com/r/jess/htop/) with a pid ae1

docker inspect -f '{{.Created}}' ae1

can show

2016-07-14T17:44:14.159094456Z

or

docker inspect -f '{{.Path}}' ae1

can show

htop

Now if I extract a part of my docker inspect

I see

"State": { "Status": "running", "Running": true, "Paused": false, "Restarting": false, "OOMKilled": false, "Dead": false, "Pid": 4525, "ExitCode": 0, "Error": "", "StartedAt": "2016- 07-14T17:44:14.406286293Z", "FinishedAt": "0001-01-01T00:00:00Z" So I get a dictionary, as it has { ...} and a lot of key:values

So the command

docker inspect -f '{{.State}}' ae1

https://riptutorial.com/ 48

will return a list, such as

{running true false false false false 4525 0 2016-07-14T17:44:14.406286293Z 0001-01- 01T00:00:00Z}

I can get the value of State.Pid easily

docker inspect -f '{{ .State.Pid }}' ae1

I get

4525

Sometimes docker inspect gives a list as it begins with [ and ends with ]

another example, with another container

docker inspect -f ‘{{ .Config.Env }}’ 7a7

gives

[DISPLAY=:0 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin LANG=fr\_FR.UTF-8 LANGUAGE=fr\_FR:en LC\_ALL=fr\_FR.UTF-8 DEBIAN\_FRONTEND=noninteractive HOME=/home/gg WINEARCH=win32 WINEPREFIX=/home/gg/.wine\_captvty]

In order to get the first element of the list, we add index before the required field and 0 (as first element) after, so

docker inspect -f ‘{{ index ( .Config.Env) 0 }}’ 7a7

gives

DISPLAY=:0

We get the next element with 1 instead of 0 using the same syntax

docker inspect -f ‘{{ index ( .Config.Env) 1 }}’ 7a7

gives

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

We can get the number of elements of this list

docker inspect -f ‘{{ len .Config.Env }}’ 7a7

gives

9

and we can get the last element of the list, the syntax is not easy

docker inspect -f “{{ index .Config.Cmd $[$(docker inspect –format ‘{{ len .Config.Cmd }}’ $CID)-1]}}” 7a7

Read docker inspect getting various fields for key:value and elements of list online: https://riptutorial.com/docker/topic/6470/docker-inspect-getting-various-fields-for-key-value-and

https://riptutorial.com/ 49

elements-of-list

https://riptutorial.com/ 50

**Chapter 14: Docker Machine**

**Introduction**

Remote management of multiple docker engine hosts.

**Remarks**

docker-machine manages remote hosts running Docker.

The docker-machine command line tool manages the full machine's life cycle using provider specific drivers. It can be used to select an "active" machine. Once selected, an active machine can be used as if it was the local Docker Engine.

**Examples**

**Get current Docker Machine environment info**

All these are shell commands.

docker-machine env to get the current default docker-machine configuration

eval $(docker-machine env) to get the current docker-machine configuration and set the current shell environment up to use this docker-machine with .

If your shell is set up to use a proxy, you can specify the --no-proxy option in order to bypass the proxy when connecting to your docker-machine: eval $(docker-machine env --no-proxy)

If you have multiple docker-machines, you can specify the machine name as argument: eval $(docker-machine env --no-proxy machinename)

**SSH into a docker machine**

All these are shell commands

• If you need to log onto a running docker-machine directly, you can do that: docker-machine ssh to ssh into the default docker-machine

docker-machine ssh machinename to ssh into a non-default docker-machine

•

If you just want to run a single command, you can do so. To run uptime on the default docker machine to see how long it's been running for, run docker-machine ssh default uptime

**Create a Docker machine**

https://riptutorial.com/ 51

Using docker-machine is the best method to install Docker on a machine. It will automatically apply the best security settings available, including generating a unique pair of SSL certificates for mutual authentication and SSH keys.

To create a local machine using Virtualbox:

docker-machine create --driver virtualbox docker-host-1

To install Docker on an existing machine, use the generic driver:

docker-machine -D create -d generic --generic-ip-address 1.2.3.4 docker-host-2

The --driver option tells docker how to create the machine. For a list of supported drivers, see: • officially supported

• third party

**List docker machines**

Listing docker-machines will return the state, address and version of Docker of each docker machines.

docker-machine ls

Will print something like:

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

docker-machine-1 - ovh Running tcp://1.2.3.4:2376 v1.11.2 docker-machine-2 - generic Running tcp://1.2.3.5:2376 v1.11.2

To list running machines:

docker-machine ls --filter state=running

To list error machines:

docker-machine ls --filter state=

To list machines who's name starts with 'side-project-', use Golang filter:

docker-machine ls --filter name="^side-project-"

To get only the list of machine's URLs:

docker-machine ls --format '{{ .URL }}'

https://riptutorial.com/ 52

See https://docs.docker.com/machine/reference/ls/ for the full command reference. **Upgrade a Docker Machine**

Upgrading a docker machine implies a downtime and may require planing. To upgrade a docker machine, run:

docker-machine upgrade docker-machine-name

This command does not have options

**Get the IP address of a docker machine**

To get the IP address of a docker machine, you can do that with this command : docker-machine ip machine-name

Read Docker Machine online: https://riptutorial.com/docker/topic/1349/docker-machine https://riptutorial.com/ 53

**Chapter 15: Docker --net modes (bridge, hots, mapped container and none).**

**Introduction**

Getting Started

**Bridge Mode** It's a default and attached to docker0 bridge. Put container on a completely separate network namespace.

**Host Mode** When container is just a process running in a host, we'll attach the container to the host NIC.

**Mapped Container Mode** This mode essentially maps a new container into an existing containers network stack. It's also called 'container in container mode'.

**None** It tells docker put the container in its own network stack without configuration **Examples**

**Bridge Mode, Host Mode and Mapped Container Mode**

**Bridge Mode**

$ docker run –d –-name my\_app -p 10000:80 image\_name

Note that we did not have to specify **--net=bridge** because this is the default working mode for docker. This allows to run multiple containers to run on same host without any assignment of dynamic port. So **BRIDGE** mode avoids the port clashing and it's safe as each container is running its own private network namespace.

**Host Mode**

$ docker run –d –-name my\_app –net=host image\_name

As it uses the host network namespace, no need of special configuraion but may leads to security issue.

**Mapped Container Mode**

This mode essentially maps a new container into an existing containers network stack. This implies that network resources such as IP address and port mappings of the first container will be shared by the second container. This is also called as 'container in container' mode. Suppose you have two contaienrs as web\_container\_1 and web\_container\_2 and we'll run web\_container\_2 in mapped container mode. Let's first download web\_container\_1 and runs it into detached mode

https://riptutorial.com/ 54

with following command,

$ docker run -d --name web1 -p 80:80 USERNAME/web\_container\_1

Once it’s downloaded let’s take a look and make sure its running. Here we just mapped a port into a container that's running in the default bridge mode. Now, let’s run a second container in mapped container mode. We’ll do that with this command.

$ docker run -d --name web2 --net=container:web1 USERNAME/web\_container\_2

Now, if you simply get the interface information on both the contaienrs, you will get the same network config. This actually include the HOST mode that maps with exact info of the host. The first contaienr ran in default bridge mode and second container is running in mapped container mode. We can obtain very similar results by starting the first container in host mode and the second container in mapped container mode.

Read Docker --net modes (bridge, hots, mapped container and none). online: https://riptutorial.com/docker/topic/9643/docker---net-modes--bridge--hots--mapped-container-and none--

https://riptutorial.com/ 55

**Chapter 16: Docker network**

**Examples**

**How to find the Container's host ip**

You need to find out the IP address of the container running in the host so you can, for example, connect to the web server running in it.

docker-machine is what is used on MacOSX and Windows.

Firstly, list your machines:

$ docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM default \* virtualbox Running tcp://192.168.99.100:2376

Then select one of the machines (the default one is called default) and:

$ docker-machine ip default

192.168.99.100

**Creating a Docker network**

docker network create app-backend

This command will create a simple bridged network called appBackend. No containers are attached to this network by default.

**Listing Networks**

docker network ls

This command lists all networks that have been created on the local Docker host. It includes the default bridge bridge network, the host host network, and the null null network. All containers by default are attached to the default bridge bridge network.

**Add container to network**

docker network connect app-backend myAwesomeApp-1

This command attaches the myAwesomeApp-1 container to the app-backend network. When you add a container to a user-defined network, the embedded DNS resolver (which is not a full-featured DNS

https://riptutorial.com/ 56

server, and is not exportable) allows each container on the network to resolve each other container on the same network. This simple DNS resolver is not available on the default bridge bridge network.

**Detach container from network**

docker network disconnect app-backend myAwesomeApp-1

This command detaches the myAwesomeApp-1 container from the app-backend network. The container will no longer be able to communicate with other containers on the network it has been disconnected from, nor use the embedded DNS resolver to look up other containers on the network it has been detached from.

**Remove a Docker network**

docker network rm app-backend

This command removes the user-defined app-backend network from the Docker host. All containers on the network not otherwise connected via another network will lose communication with other containers. It is not possible to remove the default bridge bridge network, the host host network, or the null null network.

**Inspect a Docker network**

docker network inspect app-backend

This command will output details about the app-backend network.

The of the output of this command should look similar to:

[

{

"Name": "foo",

"Id": "a0349d78c8fd7c16f5940bdbaf1adec8d8399b8309b2e8a969bd4e3226a6fc58", "Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1/16"

}

]

},

"Internal": false,

"Containers": {},

"Options": {},

"Labels": {}

https://riptutorial.com/ 57

}

]

Read Docker network online: https://riptutorial.com/docker/topic/3221/docker-network https://riptutorial.com/ 58

**Chapter 17: Docker private/secure registry with API v2**

**Introduction**

A private and secure docker registry instead of a Docker Hub. Basic docker skills are required. **Parameters**

| **Command** | **Explanation** |
| --- | --- |
| sudo docker run -p 5000:5000 | Start a docker container and bind the port 5000 from container to the port 5000 of the physical machine. |
| --name registry | Container name (use to make “docker ps” readability better). |
| -v 'pwd'/certs:/certs | Bind CURRENT\_DIR/certs of the physical machine on /certs of the container (like a “shared folder”). |
| -e  REGISTRY\_HTTP\_TLS\_CERTIFICATE=/certs/server.crt | We specify that the registry  should use /certs/server.crt file to start. (env variable) |
| -e REGISTRY\_HTTP\_TLS\_KEY=/certs/server.key | Same for the RSA key  (server.key). |
| -v /root/images:/var/lib/registry/ | If you want to save all your  registry images you should do this on the physical machine. Here we save all images on /root/images on the physical machine. If you do this then you can stop and restart the registry without losing any images. |
| registry:2 | We specify that we would like to pull the registry image from  docker hub (or locally), and we add « 2 » because we want install the version 2 of registry. |

https://riptutorial.com/ 59

**Remarks**

**How to install a docker-engine (called client on this tutorial)**

**How to generate SSL self-signed certificate**

**Examples**

**Generating certificates**

**Generate a RSA private key:** openssl genrsa -des3 -out server.key 4096

Openssl should ask for a pass phrase at this step. Notice that we’ll use only certificate for communication and authentication, without pass phrase. Just use 123456 for example.

**Generate the Certificate Signing Request:** openssl req -new -key server.key -out server.csr

This step is important because you’ll be asked for some information about certificates. The most important information is “Common Name” that is the domain name, which be used for communication between private docker registry and all other machine. Example : mydomain.com

**Remove pass phrase from RSA private key:** cp server.key server.key.org && openssl rsa -in server.key.org -out server.key

Like I said we’ll focus on certificate without pass phrase. So be careful with all your key's files (.key,.csr,.crt) and keep them on a secure place.

**Generate the self-signed certificate:** openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

You have now two essential files, server.key and server.crt, that are necessary for the private registry authentication.

**Run the registry with self-signed certificate**

To run the private registry (securely) you have to generate a self-signed certificate, you can refer to previous example to generate it.

For my example I put server.key and server.crt into /root/certs

Before run docker command you should be placed (use cd) into the directory that contains certs folder. If you're not and you try to run the command you'll receive an error like

level=fatal msg="open /certs/server.crt: no such file or directory"

When you are (cd /root in my example), you can basically start the secure/private registry using : sudo docker run -p 5000:5000 --restart=always --name registry -v `pwd`/certs:/certs -e REGISTRY\_HTTP\_TLS\_CERTIFICATE=/certs/server.crt -e REGISTRY\_HTTP\_TLS\_KEY=/certs/server.key -v

/root/Documents:/var/lib/registry/ registry:2

Explanations about the command is available on Parameters part.

https://riptutorial.com/ 60

**Pull or push from a docker client**

When you get a working registry running you can pull or push images on it. For that you need the server.crt file into a special folder on your docker client. The certificate allows you to authenticate with the registry, and then encrypt communication.

Copy server.crt from registry machine into /etc/docker/certs.d/mydomain.com:5000/ on your client machine. And then rename it to ca-certificates.crt : mv

/etc/docker/certs.d/mydomain.com:5000/server.crt /etc/docker/certs.d/mydomain.com:5000/ca certificates.crt

At this point you can pull or push images from your private registry :

PULL : docker pull mydomain.com:5000/nginx or

PUSH :

1. Get an official image from hub.docker.com : docker pull nginx

2.

Tag this image before pushing into private registry : docker tag IMAGE\_ID mydomain.com:5000/nginx (use docker images to get the IMAGE\_ID)

3. Push the image to the registry : docker push mydomain.com:5000/nginx

Read Docker private/secure registry with API v2 online:

https://riptutorial.com/docker/topic/8707/docker-private-secure-registry-with-api-v2 https://riptutorial.com/ 61

**Chapter 18: Docker Registry**

**Examples**

**Running the registry**

**Do not use registry:latest!** This image points to the old v1 registry. That Python project is no longer being developed. The new v2 registry is written in Go and is actively maintained. When people refer to a "private registry" they are referring to the v2 registry, not the v1 registry!

docker run -d -p 5000:5000 --name="registry" registry:2

The above command runs the newest version of the registry, which can be found in the Docker Distribution project.

For more examples of image management features, such as tagging, pulling, or pushing, see the section on managing images.

**Configure the registry with AWS S3 storage backend**

Configuring a private registry to use an AWS S3 backend is easy. The registry can do this automatically with the right configuration. Here is an example of what should be in your config.yml file:

storage:

s3:

accesskey: AKAAAAAACCCCCCCBBBDA

secretkey: rn9rjnNuX44iK+26qpM4cDEoOnonbBW98FYaiDtS

region: us-east-1

bucket: registry.example.com

encrypt: false

secure: true

v4auth: true

chunksize: 5242880

rootdirectory: /registry

The accesskey and secretkey fields are IAM credentials with specific S3 permissions (see the documentation for more information). It can just as easily use credentials with the AmazonS3FullAccess policy attached. The region is the region of your S3 bucket. The bucket is the bucket name. You may elect to store your images encrypted with encrypt. The secure field is to indicate the use of HTTPS. You should generally set v4auth to true, even though its default value is false. The chunksize field allows you to abide by the S3 API requirement that chunked uploads are at least five megabytes in size. Finally, rootdirectory specifies a directory underneath your S3 bucket to use.

There are other storage backends that can be configured just as easily.

Read Docker Registry online: https://riptutorial.com/docker/topic/4173/docker-registry https://riptutorial.com/ 62

**Chapter 19: Docker stats all running containers**

**Examples**

**Docker stats all running containers**

sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))

Shows live CPU usage of all running containers.

Read Docker stats all running containers online: https://riptutorial.com/docker/topic/5863/docker stats-all-running-containers

https://riptutorial.com/ 63

**Chapter 20: Docker swarm mode**

**Introduction**

A swarm is a number of Docker Engines (or nodes) that deploy services collectively. Swarm is used to distribute processing across many physical, virtual or cloud machines.

**Syntax**

• Initialize a swarm: docker swarm init [OPTIONS]

• Join a swarm as a node and/or manager: docker swarm join [OPTIONS] HOST:PORT • Create a new service: docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

•

Display detailed information on one or more services: docker service inspect [OPTIONS] SERVICE [SERVICE...]

• List services: docker service ls [OPTIONS]

• Remove one or more services: docker service rm SERVICE [SERVICE...]

•

Scale one or multiple replicated services: docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]

• List the tasks of one or more services: docker service ps [OPTIONS] SERVICE [SERVICE...] • Update a service: docker service update [OPTIONS] SERVICE

**Remarks**

Swarm mode implements the following features:

• Cluster management integrated with Docker Engine

• Decentralized design

• Declarative service model

• Scaling

• Desired state reconciliation

• Multi-host networking

• Service discovery

• Load balancing

• Secure design by default

• Rolling updates

For more official Docker documentation regarding Swarm visit: Swarm mode overview https://riptutorial.com/ 64

**Swarm Mode CLI Commands**

Click on commands description for documentation

Initialize a swarm

docker swarm init [OPTIONS]

Join a swarm as a node and/or manager

docker swarm join [OPTIONS] HOST:PORT

Create a new service

docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

Display detailed information on one or more services

docker service inspect [OPTIONS] SERVICE [SERVICE...]

List services

docker service ls [OPTIONS]

Remove one or more services

docker service rm SERVICE [SERVICE...]

Scale one or multiple replicated services

docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS...]

List the tasks of one or more services

docker service ps [OPTIONS] SERVICE [SERVICE...]

Update a service

docker service update [OPTIONS] SERVICE

**Examples**

**Create a swarm on Linux using docker-machine and VirtualBox** https://riptutorial.com/ 65

# Create the nodes

# In a real world scenario we would use at least 3 managers to cover the fail of one manager. docker-machine create -d virtualbox manager

docker-machine create -d virtualbox worker1

# Create the swarm

# It is possible to define a port for the \*advertise-addr\* and \*listen-addr\*, if none is defined the default port 2377 will be used.

docker-machine ssh manager \

docker swarm init \

--advertise-addr $(docker-machine ip manager)

--listen-addr $(docker-machine ip manager)

# Extract the Tokens for joining the Swarm

# There are 2 different Tokens for joining the swarm.

MANAGER\_TOKEN=$(docker-machine ssh manager docker swarm join-token manager --quiet) WORKER\_TOKEN=$(docker-machine ssh manager docker swarm join-token worker --quiet)

# Join a worker node with the worker token

docker-machine ssh worker1 \

docker swarm join \

--token $WORKER\_TOKEN \

--listen-addr $(docker-machine ip worker1) \

$(docker-machine ip manager):2377

**Find out worker and manager join token**

When automating the provisioning of new nodes to a swarm, you need to know what the right join token is for the swarm as well as the advertised address of the manager. You can find this out by running the following commands on any of the existing manager nodes:

# grab the ipaddress:port of the manager (second last line minus the whitespace) export MANAGER\_ADDRESS=$(docker swarm join-token worker | tail -n 2 | tr -d '[[:space:]]')

# grab the manager and worker token

export MANAGER\_TOKEN=$(docker swarm join-token manager -q)

export WORKER\_TOKEN=$(docker swarm join-token worker -q)

The -q option outputs only the token. Without this option you get the full command for registering to a swarm.

Then on newly provisioned nodes, you can join the swarm using.

docker swarm join --token $WORKER\_TOKEN $MANAGER\_ADDRESS

**Hello world application**

Usually you'd want to create a stack of services to form a replicated and orchestrated application. A typical modern web application consists of a database, api, frontend and reverse proxy.

https://riptutorial.com/ 66

**Persistence**

Database needs persistence, so we need some filesystem which is shared across all the nodes in a swarm. It can be NAS, NFS server, GFS2 or anything else. Setting it up is out of scope here. Currently Docker doesn't contain and doesn't manage persistence in a swarm. This example assumes that there's /nfs/ shared location mounted across all nodes.

**Network**

To be able to communicate with each other, services in a swarm need to be on the same network. Choose an IP range (here 10.0.9.0/24) and network name (hello-network) and run a command:

docker network create \

--driver overlay \

--subnet 10.0.9.0/24 \

--opt encrypted \

hello-network

**Database**

The first service we need is a database. Let's use postgresql as an example. Create a folder for a database in nfs/postgres and run this:

docker service create --replicas 1 --name hello-db \

--network hello-network -e PGDATA=/var/lib/postgresql/data \

--mount type=bind,src=/nfs/postgres,dst=/var/lib/postgresql/data \ kiasaki/alpine-postgres:9.5

Notice that we've used --network hello-network and --mount options.

**API**

Creating API is out of scope of this example, so let's pretend you have an API image under username/hello-api.

docker service create --replicas 1 --name hello-api \

--network hello-network \

-e NODE\_ENV=production -e PORT=80 -e POSTGRESQL\_HOST=hello-db \

username/hello-api

Notice that we passed a name of our database service. Docker swarm has an embedded round robin DNS server, so API will be able to connect to database by using its DNS name.

**Reverse proxy**

Let's create nginx service to serve our API to an outer world. Create nginx config files in a shared location and run this:

docker service create --replicas 1 --name hello-load-balancer \

--network hello-network \

https://riptutorial.com/ 67

--mount type=bind,src=/nfs/nginx/nginx.conf,dst=/etc/nginx/nginx.conf \ -p 80:80 \

nginx:1.10-alpine

Notice that we've used -p option to publish a port. This port would be available to any node in a swarm.

**Node Availablility**

Swarm Mode Node Availability:

• Active means that the scheduler can assign tasks to a node.

•

Pause means the scheduler doesn’t assign new tasks to the node, but existing tasks remain running.

•

Drain means the scheduler doesn’t assign new tasks to the node. The scheduler shuts down any existing tasks and schedules them on an available node.

To change Mode Availability:

#Following commands can be used on swarm manager(s)

docker node update --availability drain node-1

#to verify:

docker node ls

**Promote or Demote Swarm Nodes**

To promote a node or set of nodes, run docker node promote from a manager node:

docker node promote node-3 node-2

Node node-3 promoted to a manager in the swarm.

Node node-2 promoted to a manager in the swarm.

To demote a node or set of nodes, run docker node demote from a manager node:

docker node demote node-3 node-2

Manager node-3 demoted in the swarm.

Manager node-2 demoted in the swarm.

**Leaving the Swarm**

Worker Node:

#Run the following on the worker node to leave the swarm.

docker swarm leave

Node left the swarm.

If the node has the Manager role, you will get a warning about maintaining the quorum of https://riptutorial.com/ 68

Managers. You can use --force to leave on the manager node:

#Manager Node

docker swarm leave --force

Node left the swarm.

Nodes that left the Swarm will still show up in docker node ls output.

To remove nodes from the list:

docker node rm node-2

node-2

Read Docker swarm mode online: https://riptutorial.com/docker/topic/749/docker-swarm-mode https://riptutorial.com/ 69