## 

## rkt (pronounced "rock it")

is a container runtime for applications made by CoreOS and is designed for composability, speed, and security.

rkt is a standalone tool, compared to Docker's

CoreOS rkt is a secure, modular execution engine for app containers. It was designed to reflect the lessons of running container cluster infrastructure at scale and is released under the Apache license. Written in Go, rkt implements container isolation through a flexible and interchangeable set of “stages,” providing multiple execution regimes for a container image. At rkt’s core is a command line utility that does not invoke a long-running daemon process, making app container lifecycle management simpler and allowing loosely-coupled integrations with service management and orchestration systems like systemd and Kubernetes.

## rkt Features

rkt boasts the following features:

* **Modularity** – rkt is architected in stages (image fetching, cgroup and networking setup, and execution) that can have different implementations, providing separation of privileges as well as concerns.
* **Composability** – rkt is not a daemon, is not the parent process of all your containers (and therefore can be updated without affecting running containers), and is composable with other tools. Natively runs appc images built with acbuild.
* **Security** – It has Intel Clear Container, SELinux and TPM support, as well as image signature validation.
* It can run Docker images.

1. First prepare the environment:  
    Install Ubuntu 16  
    Debian 8  
    prepare the network environment for the servers
2. After preparing the physical server environment   
    Install and prepare **rkt** >> apt-get install git

>> git clone <https://github.com/coreos/rkt>

>> wget <https://raw.githubusercontent.com/coreos/rkt/master/scripts/install-rkt.sh>

>> chmod +x install-rkt.sh

>> sudo ./install-rkt.sh

1. Install GO and acbuild packages

Step 1 — Installing Go  
 >>cd ~

>>curl -O [https://storage.googleapis.com/golang/go1.6.linux- amd64.tar.gz](https://storage.googleapis.com/golang/go1.6.linux-%20%20%20%20%20%20%20amd64.tar.gz)

>>tar xvf go1.6.linux-amd64.tar.gz

>>sudo chown -R root:root ./go

>> sudo mv go /usr/local

Step 2 — Setting Go Paths

>>sudo nano ~/.profile

At the end of the file, add this line:

export GOPATH=$HOME/work

export PATH=$PATH:/usr/local/go/bin:$GOPATH/bin

Save and exit then

>>source ~/.profile

Step 3 — Testing Your Install

>>mkdir $HOME/work

>>mkdir -p work/src/github.com/user/hello

>>nano ~/work/src/github.com/user/hello/hello.go

Inside your editor, paste the code below, which uses the main Go packages, imports the formatted IO content component, and sets a new function to print "Hello, World" when run.

package main

import "fmt"

func main() {

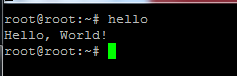
fmt.Printf("hello, world\n")

}

>>go install github.com/user/hello

With the file compiled, you can run it by simply executing the command:

>>hello



1. Prepare the system to build the containers

>>cd ~

>>git clone https://github.com/containers/build acbuild

>>cd acbuild

>>./build

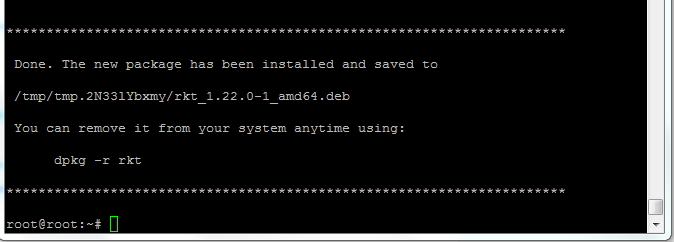
>>vi ~/.bashrc  
and put the following lines at the end of the file:

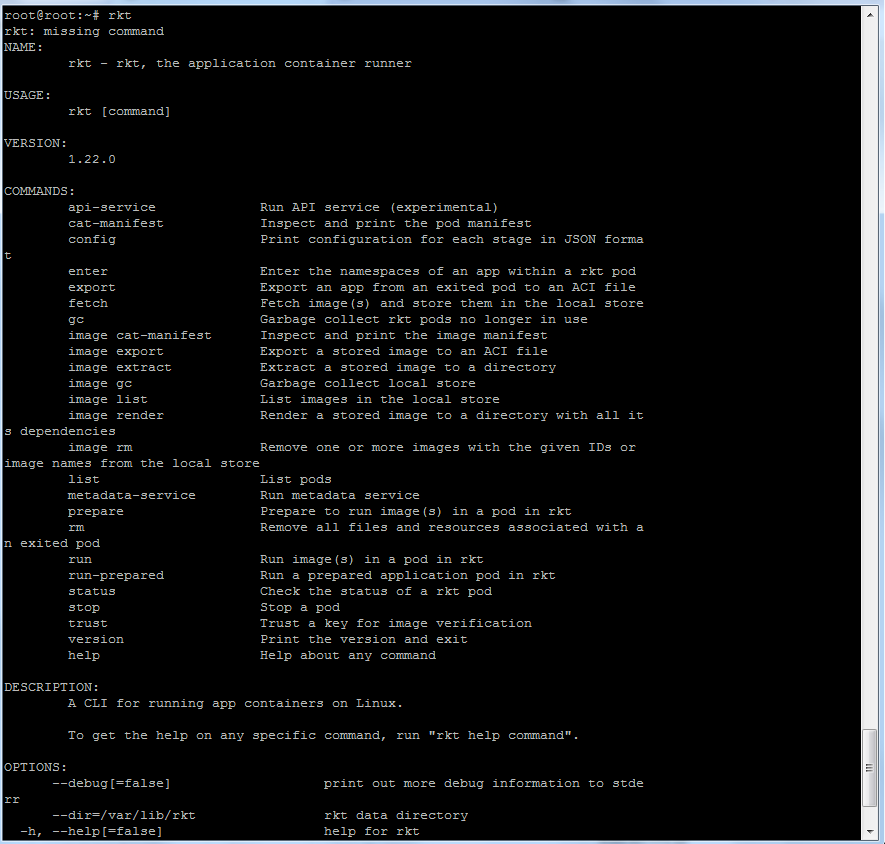
export ACBUILD\_BIN\_DIR=~/acbuild/bin

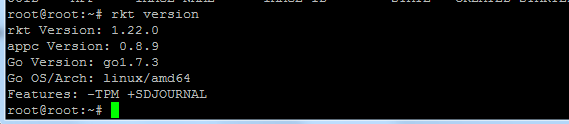
export PATH=$PATH:$ACBUILD\_BIN\_DIR

Then

>>source ~/.bashrc









Create the image

To create the image, we can use acbuild, which can be downloaded via one of the releases in the containers/build repository.

The following commands will create an ACI containing our application and important metadata.

>>acbuild begin

>>acbuild set-name example.com/hello

>>acbuild copy hello /bin/hello

>>acbuild set-exec /bin/hello

>>acbuild port add www tcp 5000

>>acbuild label add version 0.0.1

>>acbuild label add arch amd64

>>acbuild label add os linux

>>acbuild annotation add authors "Carly Container <carly@example.com>"

>>acbuild write hello-0.0.1-linux-amd64.aci

>>acbuild end

Run

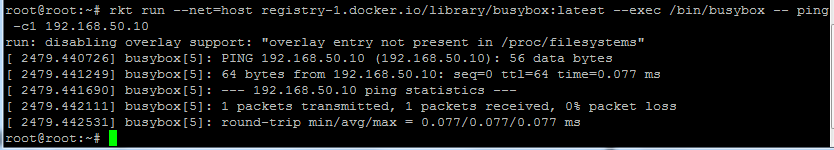
Launch a local application image

>>rkt --insecure-options=image run hello-0.0.1-linux-amd64.aci

>>rkt image cat-manifest coreos.com/rkt/stage1-coreos:1.22.0

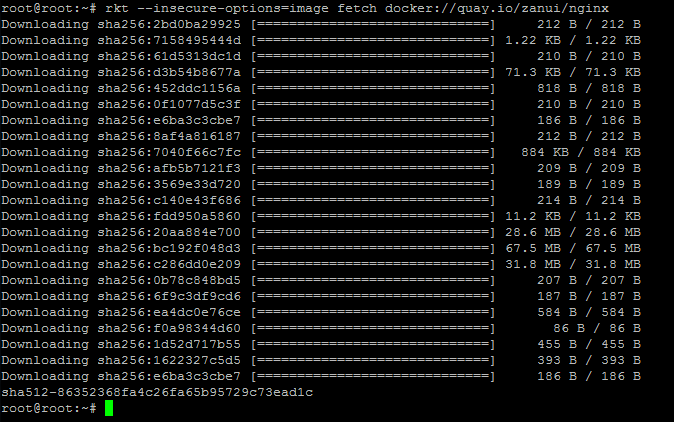
Ping from the image to the physical server

>>rkt run --net=host registry-1.docker.io/library/busybox:latest --exec /bin/busybox -- ping -c1 192.168.50.10

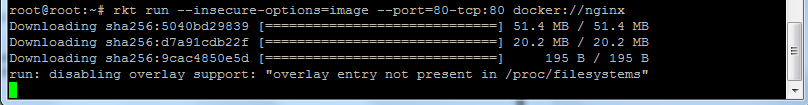


## Running Docker images with rkt Fetch from a Docker registry

If you want to run an existing Docker image, you can fetch from a Docker registry. rkt will download and convert the image to ACI.eg.  
 >>rkt --insecure-options=image fetch docker://quay.io/zanui/nginx



>>rkt run --insecure-options=image --port=80-tcp:80 docker://nginx



**Networking**

**Host mode**

When --net=host is passed the pod's apps will inherit the network namespace of the process that is invoking rkt.

If rkt is directly called from the host the apps within the pod will share the network stack and the interfaces with the host machine. This means that

every network service that runs in the pod has the same connectivity as if it was started on the host directly.

**Contained mode**

If anything other than host is passed to --net=, the pod will live in a separate network namespace with the help of [CNI](https://github.com/appc/cni) and its plugin system. The network setup for the pod's network namespace depends on the available CNI configuration files that are shipped with rkt and also configured by the user.

### Builtin networks

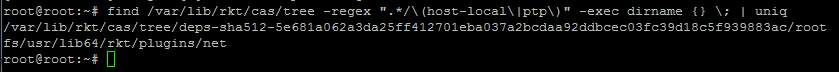
rkt ships with two built-in networks, named default*and*default-restricted.

Default: It consists of a loopback device and a veth device. The veth pair creates a point-to-point link between the pod and the host. rkt will allocate an IPv4 address out of 172.16.28.0/24 for the pod's veth interface. It will additionally set the default route in the pod namespace. Finally, it will enable IP masquerading on the host to NAT the egress traffic.

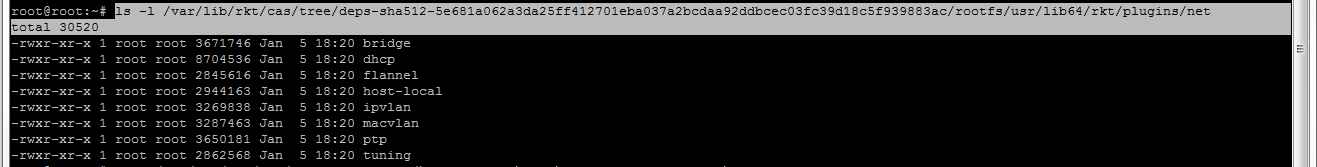
Default-restricted: does not set up the default route and IP masquerading. It only allows communication with the host via the veth interface and thus en

ables the pod to communicate with the metadata service which runs on the host.

>>find /var/lib/rkt/cas/tree -regex ".\*/\(host-local\|ptp\)" -exec dirname {} \; | uniq



>>ls -l /var/lib/rkt/cas/tree/deps-sha512-5e681a062a3da25ff412701eba037a2bcdaa92ddbcec03fc39d18c5f939883ac/rootfs/usr/lib64/rkt/plugins/net



**References**

<https://www.digitalocean.com/community/tutorials/how-to-install-go-1-6-on-ubuntu-16-04>  
<https://sreeninet.wordpress.com/2015/02/02/containers-docker-lxc-and-rocke>

<https://dzone.com/articles/getting-started-with-rkt-via-codeship>  
<https://coreos.com/blog/announcing-rkt-0.5/>

<https://coreos.com/rkt/docs/latest/trying-out-rkt.html>

<https://coreos.com/blog/getting-started-with-rkt-1-0.html>

<https://coreos.com/rkt/docs/latest/subcommands/prepare.html>  
<https://github.com/containers/build>

<http://opensourcebridge.org/proposals/1876>

<https://coreos.com/blog/rkt-and-kubernetes.html>

<https://coreos.com/rkt/docs/latest/networking/overview.html>