

# Docker Container Fundamental & Basic Docker Orchestration Part 1

Detail Materi

Indikator :  
Memahami konsep dasar Containerisasi  
menggunakan Docker, dapat mempraktekan  
perintah dasar docker dan memahami  
arsitektur dari sistem Container tersebut



Konsep Dasar Container  
dengan Docker



Instalasi Docker MultiPlatform



Perintah Dasar Docker



Container Architecture dan  
Perbandingan dengan VM



Management dan Monitoring  
Container

## **Modul Sekolah DevOps Cilsy**

Hak Cipta © 2020 **PT. Cilsy Fiolution Indonesia**

Hak Cipta dilindungi Undang-Undang. Dilarang memperbanyak atau memindahkan sebagian atau seluruh isi buku ini dalam bentuk apapun, baik secara elektronik maupun mekanis, termasuk mecropy, merekam atau dengan sistem penyimpanan lainnya, tanpa izin tertulis dari Penulis dan Penerbit.

Penulis : Estu Fardani, Adi Saputra, Irfan Herfiandana & Tresna Widiyaman  
Editor: Taufik Maulana, Iqbal Ilman F, Muhammad Fakhri A., Rizal Rahman & Tresna Widiyaman

**Revisi Batch 9**

Penerbit : **PT. Cilsy Fiolution Indonesia**

Web Site : <https://cilsyfiolution.com> , <https://devops.cilsy.id>

### **Sanksi Pelanggaran Pasal 113 Undang-undang Nomor 28 Tahun 2014 tentang Hak Cipta**

1. Setiap orang yang dengan tanpa hak melakukan pelanggaran hak ekonomi sebagaimana dimaksud dalam pasal 9 ayat (1) huruf i untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 1 (satu) tahun dan atau pidana denda paling banyak Rp100.000.000,00 (seratus juta rupiah).
2. Setiap orang yang dengan tanpa hak dan atau tanpa izin pencipta atau pemegang hak cipta melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam pasal 9 ayat (1) huruf c, huruf d, huruf f, dan atau huruf h, untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 3 (tiga) tahun dan atau pidana denda paling banyak Rp500.000.000,00 (lima ratus juta rupiah)
3. Setiap orang yang dengan tanpa hak dan atau tanpa izin pencipta atau pemegang hak melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam pasal 9 ayat (1) huruf a, huruf b, huruf e, dan atau huruf g, untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 4 (empat) tahun dan atau pidana denda paling banyak Rp1.000.000.000,00 (satu miliar rupiah)
4. Setiap orang yang memenuhi unsur sebagaimana dimaksud pada ayat (3) yang dilakukan dalam bentuk pembajakan, dipidana dengan pidana penjara paling lama 10 (sepuluh) tahun dan atau pidana denda paling banyak Rp4.000.000.000,00 (empat miliar rupiah)

## Daftar Isi

Cover.....	1
8. <i>Docker Container Fundamental &amp; Basic Docker Orchestration di Server</i>	
<i>Production Part 1</i> .....	5
Learning Outcomes.....	5
Outline Materi.....	5
8.1. Docker.....	6
8.1.1. Sejarah dan Konsep Docker.....	6
8.1.2. Perbedaan Docker dan Virtualisasi.....	8
8.1.2.1. Struktur Virtualisasi.....	8
8.1.2.2. Stuktur Docker.....	9
8.1.3. Komponen Docker.....	10
8.1.4. Masalah utama yang dipecahkan.....	12
8.1.5. Contoh Mesin Kontainerisasi Lainnya.....	15
8.1.5.1. PODMAN.....	15
8.1.5.2. Buildah.....	16
8.1.5.3. Cri-O.....	17
8.2. Docker Installation.....	18
8.2.1. Docker Edition.....	18
8.2.1.1. CE vs EE.....	18
8.2.1.2. Edge vs Stable.....	19
8.2.1.3. Penulisan Versi.....	19
8.2.2. Langkah instalasi Docker.....	20
8.2.2.1. Linux.....	20
8.2.2.2. Pengecekan hasil instalasi.....	21
8.2.2.3. Instalasi Software tambahan.....	21
8.2.3. Exercise.....	22
8.3. Perintah-Perintah Dasar Docker.....	23
8.3.1. Docker <i>Version</i> .....	23



8.3.2. Docker Info.....	23
8.3.3. Docker <i>Help</i> .....	24
8.3.4. Docker Command Structure.....	25
8.4. Arsitektur Docker.....	26
8.4.1. Client.....	27
8.4.2. <i>Host</i> dan <i>Daemon</i> .....	27
8.4.3. Image.....	27
8.4.4. Container.....	28
8.4.5. Registry.....	28
8.4.6. Alur penggunaan Docker secara umum.....	28
8.4.7. Komponen lainnya.....	29
8.5. Manajemen <i>Container</i> .....	29
8.5.1. Menjalankan <i>Container</i> Baru.....	29
8.5.2. Melihat <i>Container</i> yang Berjalan dan Menghentikannya.....	31
8.5.3. Menjalankan <i>Container</i> yang sudah dihentikan.....	32
8.5.4. Docker <i>Logs</i> dan <i>Top</i> .....	33
8.5.5. Menghapus <i>Container</i> .....	33
8.5.6. Exercise.....	34
8.6. Menggali Lebih dalam yang terjadi saat <i>Container</i> dijalankan.....	34
8.7. Container vs VM.....	36
8.8. Monitoring Container.....	38
8.8.1. <i>Top</i> .....	38
8.8.2. <i>Inspect</i> .....	39
8.8.3. <i>Stats</i> .....	40
8.9. Masuk ke dalam <i>Container</i> .....	40
8.9.1. Menjalankan Container dengan mode Interaktif.....	40
8.9.2. Docker Exec.....	41
8.10. Exercise.....	42
8.11. Summary.....	43



## 8.

# ***Docker Container Fundamental & Basic Docker Orchestration di Server Production Part 1***

## ***Learning Outcomes***

Setelah selesai mempelajari bab ini, peserta mampu :

1. Memahami Konsep Dasar Containerisasi menggunakan Docker
2. Dapat Menginstall dan menggunakan Docker di Berbagai *Platform*
3. Memahami dan Mengaplikasikan Perintah Dasar Docker
4. Memahami Arsitektur *Container* dan Perbedaannya dengan VM
5. Melakukan *Management* dan *Monitoring Container*

## **Outline Materi**

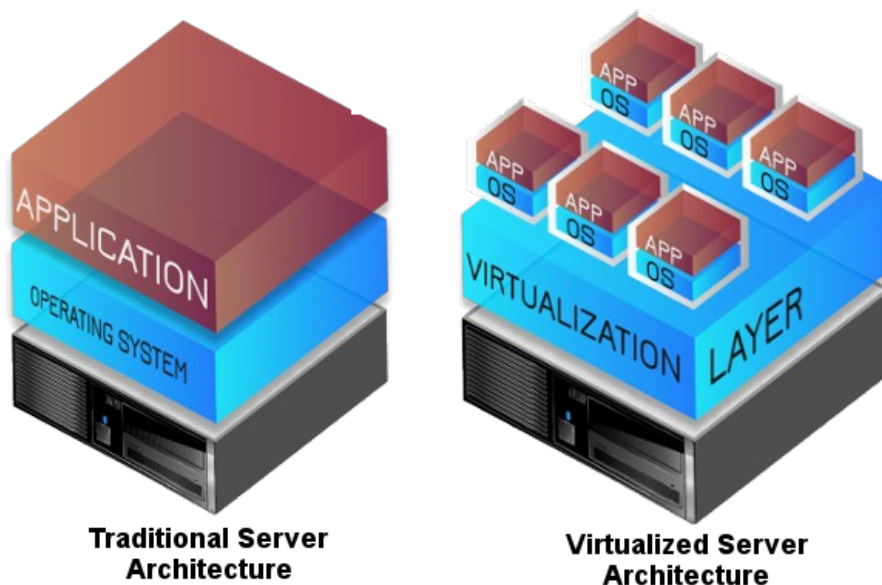
1. Konsep dasar Docker dan *Container*
2. Docker *Installation*
3. Perintah Dasar Docker
4. Arsitektur Docker
5. Management *Container*
6. *Container* vs *Virtual Machine*
7. *Monitoring Container*



## 8.1. Docker

### 8.1.1. Sejarah dan Konsep Docker

Pada dasarnya sebelum ada Docker, teknologi yang sudah cukup sering dipakai adalah teknologi Virtualisasi. Yaitu teknologi untuk dapat memiliki banyak Server Virtual didalam sebuah server fisik. Sistem Virtualisasi ini sangat digandrungi, karena dapat memaksimalkan efisiensi secara budget dan pengelolaan. Bisa bayangkan kita cukup membeli 1 buah Server, tapi didalamnya bisa kita bangun lagi 5 buah server virtual. Sangat cost-effective.



*Arsitektur tradisional vs Virtualisasi*

Sayangnya **Virtualisasi** memiliki 1 kekurangan besar, yaitu untuk menggunakan sebuah layanan/service kita tetap harus menginstall terlebih dahulu Sistem Operasi di atasnya, walaupun layanan/service tersebut sangat kecil kebutuhan resourcenya. Bisa kita lihat pada gambar diatas, untuk setiap App, tetap perlu OS dibawahnya.

Perlunya ada sistem operasi ini di masing-masing service ini juga menyebabkan pengelolaan menjadi kompleks dan lambat. Belum lagi beban yang semakin berat. Bisa bayangkan jika kita memiliki 5 server virtual, masing-



masing memakan resource 2GB RAM, maka server kita sudah habis *resource* sebesar 10GB RAM.

Hal ini yang menyebabkan Virtualisasi menjadi kurang relevan di era serba cepat, era aplikasi seperti saat ini. Dibutuhkan sebuah teknologi mirip seperti Virtualisasi namun yang tidak memakan *resource* sebesar itu dan tidak selambat itu dalam pengelolaannya. Teknologi mirip Virtualisasi yang ringan dan cepat. Inilah yang disebut sebagai **Teknologi Container**.

Pada mulanya, teknologi *Container* ini muncul dari seseorang bernama Solomon Hykes yang memulai project Container bernama **Docker** di Prancis sebagai proyek internal di dotCloud, sebuah perusahaan *platform-as-a-service*. Docker dirilis sebagai proyek aplikasi *open source* pada bulan Maret 2013. Dengan sangat cepat Docker menjadi terkenal hingga pada tahun 2015 saja Docker sudah terdaftar memiliki lebih dari 6000 pengguna. Oleh karena itulah bermunculan standar – standar untuk teknologi Docker container, seperti OCI dan CNCF.

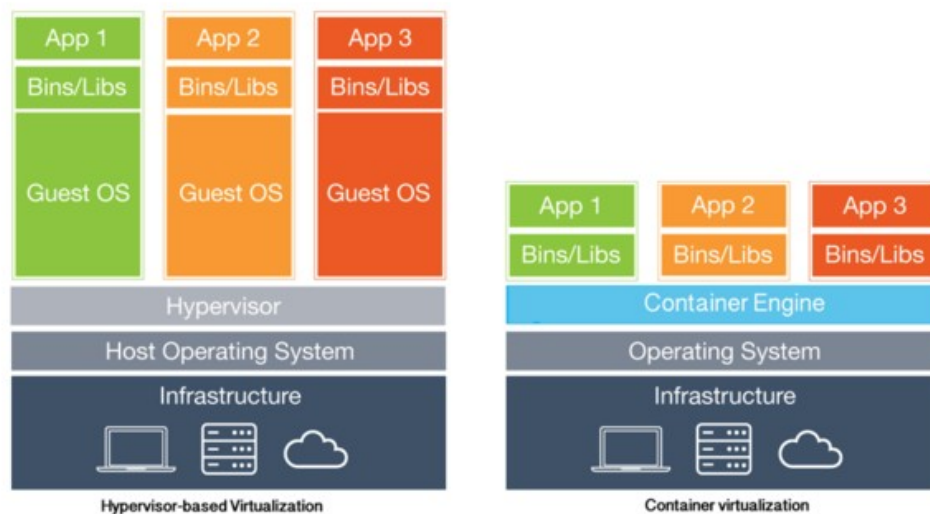


Docker dengan teknologi Containernya muncul dan membuat gebrakan di tengah-tengah pengguna Virtualisasi biasa. Kini Docker berhasil menjadi salah satu teknologi yang banyak dipakai di dalam industri startup. Alasannya mudah, karena dengan docker kita tidak perlu menginstall Sistem Operasi tertentu apabila kita hanya ingin menggunakan salah satu servicenya. Sebuah kelebihan yang tidak ada pada sistem Virtualisasi biasa.

Dengan demikian kita dapat menghemat sumber daya mesin kita untuk dapat melakukan job lainnya juga.

### 8.1.2. Perbedaan Docker dan Virtualisasi

Baik **Teknologi Docker Container** dan **teknologi Virtualisasi** memiliki beberapa perbedaan dalam arsitekturnya, antara lain adalah sebagai berikut.



*Ilustrasi perbedaan antara virtualisasi dengan container*

Gambar diatas merupakan analogi struktur Virtualisasi (Kiri) dan Docker *Container* (Kanan). Nah dari gambar diatas dapat kita lihat perbedaannya.

#### 8.1.2.1. Struktur Virtualisasi

Pada struktur VM dibutuhkan Hypervisor (*Virtualization Software*) sebagai penghubung antara mesin kita (*Host OS*) dan OS Virtualisasi (*Guest OS*). Dengan Demikian kita dapat menginstall OS apapun dengan versi kernel berapapun di dalam *Guest OS*, karena *Guest OS* dan *Host OS* bisa dikatakan terpisah meskipun secara logical *Guest OS* terletak di dalam *Host OS*. Misalnya saja kita dapat menginstall Windows 7 di dalam *Guest OS* meskipun *Host OS* kita menggunakan Ubuntu 16.04.





### 8.1.2.2. Struktur Docker

Pada Struktur Docker dapat dilihat bahwa Container berjalan langsung diatas Mesin Kita (*Host OS*) tanpa adanya penghubung (*Hypervisor*) dengan demikian tidak dimungkinkan untuk menginstall OS maupun service di dalam Container dengan versi kernel yang berbeda. Jadi apabila kita menginstall OS atau service pada container pastilah OS atau *service* yang memiliki versi kernel dan *library* yang sama dengan *Host OS*.

Misalnya kita dapat menginstall Centos pada container meskipun *Host OS* kita adalah ubuntu dengan catatan Centos yang kita install menjalankan versi kernel yang sama dengan *Host OS*.

Masing-masing tentunya ada plus dan minusnya. Bila kita menggunakan hypervisor:

- Sistem operasi host dan guest tidak ada hubungan sama sekali. Jadi kita bisa menginstal sistem operasi apapun sebagai guest.
- *Overhead* lebih tinggi, karena adanya hardware virtual. Konsekuensinya, *performance* menjadi lebih terbatas. Penggunaan resource juga tidak bisa terlalu dioptimalkan, karena begitu sudah dibooking oleh salah satu guest tidak bisa digunakan *guest* lain, walaupun *guest* pertama sedang sibuk.
- Lebih mudah mengatur keamanan, karena hubungan antara *host* dan *guest* sangat sedikit.

Bila kita menggunakan *container*:

- Sistem operasi *guest* berbagi kernel dan aplikasi utama dengan *host*. Dengan demikian kita tidak bisa menginstal sistem operasi yang berbeda dengan *host*, seperti Windows di host Linux.
- Penggunaan *resource hardware* bisa lebih dioptimalkan. Karena *host* mengetahui secara detail kondisi masing-masing guest, maka dia bisa

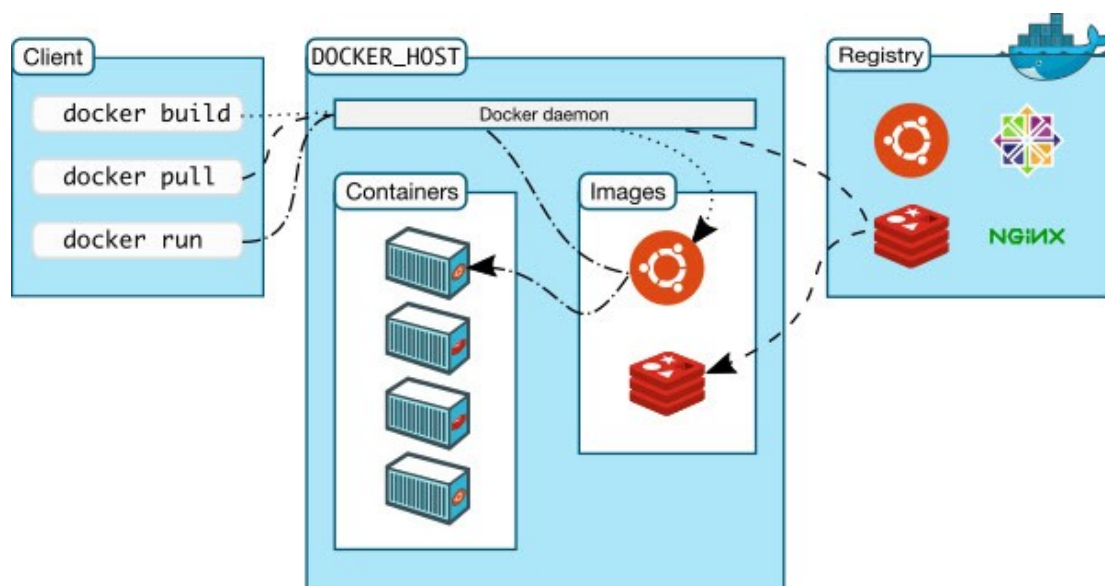


mengalokasikan *resource* yang tidak terpakai kepada *guest* yang sedang sibuk.

- Dengan Docker, kita bisa membuat macam-macam *environment* yang mendukung 1 aplikasi.

### 8.1.3. Komponen Docker

Selain memiliki kelebihan, docker itu sendiri dapat berjalan karena beberapa komponen yang ada didalamnya, berikut merupakan beberapa komponen yang ada dalam docker.



Arsitektur komponen Docker

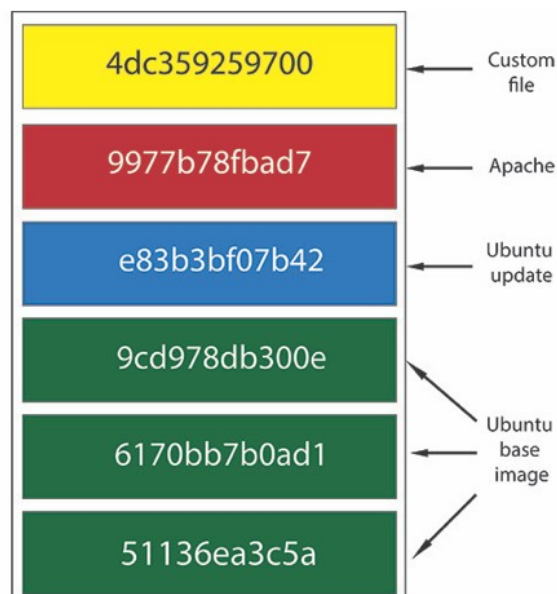
Untuk memahami diagram arsitektur tersebut, kita perlu memahami beberapa istilah yang digunakan di dunia Docker.

- **Docker Container** : adalah virtual machine atau guest operating system. Aplikasi kita berjalan di dalam docker container dan merasa bahwa dia berjalan di dalam sistem operasi biasa. Docker container berjalan di atas docker engine.
- **Docker Client** : adalah seperangkat perintah *command line* untuk mengoperasikan docker container, misalnya membuat *container*, *start/stop container*, menghapus (*destroy*), dan sebagainya. Docker client



hanya bertugas mengirim perintah saja. Pekerjaan sesungguhnya dilakukan oleh docker daemon.

- **Docker Daemon** : adalah aplikasi yang berjalan di host machine. Docker server berjalan di *background* (sebagai *daemon*) dan menunggu perintah dari docker *client*. Begitu mendapatkan perintah, docker server bekerja membuat *container*, menjalankan/mematikan container, dan sebagainya.
- **Docker Image** : adalah template yang digunakan untuk membuat *container*. Contohnya, ada *image* Ubuntu, CentOS, dan sebagainya. Kita juga bisa membuat image baru dari image yang lama. Misalnya kita buat image Ubuntu yang sudah terinstal Java dan MySQL. Docker image memiliki komposisi dari beberapa layer. Tiap layer bisa mewakili *command* atau perubahan tertentu.









*Ilustrasi Docker Image*









- **Docker Registry** : adalah tempat kumpulan *image-image* (*repository*) yang dapat digunakan dan diakses oleh semua orang karena tersimpan di *cloud*/di server yang sudah disediakan oleh pengembang Docker.

### 8.1.4. Masalah utama yang dipecahkan

Masalah utama yang dipecahkan oleh Docker dengan teknologi Kontainernya adalah soal kompleksitas. Kita coba lihat matriks dibawah :

**The Matrix of Hell**

	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers

*Gambar Matrix of Hell*

Diatas disebut sebagai *Matrix of Hell*. Yaitu kondisi yang terjadi ketika setiap komponen aplikasi yang dibangun harus disesuaikan dengan masing-masing infrastruktur, envirointment, dan *device* agar dapat berjalan dengan lancar.

Jaman sekarang, untuk sebuah aplikasi sederhana saja itu biasanya pasti memiliki beberapa komponen yang dipisahkan. Misalnya seperti berikut :

1. *Frontend*
2. *Backend*
3. *Database*
4. *Queue*
5. dll



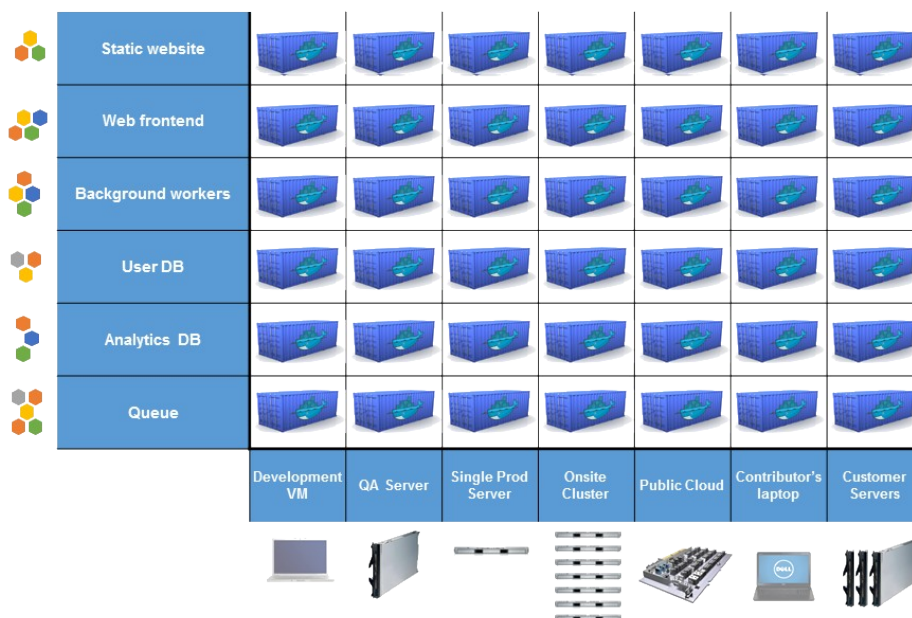
Masing-masing komponen tersebut tentunya agar bisa running harus memiliki konfigurasi OS tertentu, dependensi tertentu, topologi ip tertentu, dsb.

Bisa Anda bayangkan kompleksitas yang akan terjadi?

Di server datacenter kita harus konfigurasi sendiri VM nya, OS nya, konfigurasi tambahan dllnya. Lalu kalau mau migrasi ke Public Cloud Amazon Web Services juga harus di atur ulang VM, OS, dependensinya, ip nya dll. Dst. Lalu bagaimana juga agar aplikasi tersebut bisa jalan di komputer si Developer? Tentu harus setup ulang segala macamnya.

Docker mengatasi masalah kompleksitas ini dengan sangat baik. Docker memiliki konsep untuk “mempacking” seluruh kebutuhan OS, dependensi, konfigurasi dll nya. Ini menjadi sebuah bentuk Container untuk bisa “dibuka dan dijalankan” di manapun kita mau.

Baik itu di Windows, Mac, Linux, baik itu di Publik *Cloud*, di Datacenter, di Komputer, laptop, selama menjalankan Docker maka Container yang sudah “di-*packing*” tersebut bisa “dibuka dan dijalankan” dengan lancar.

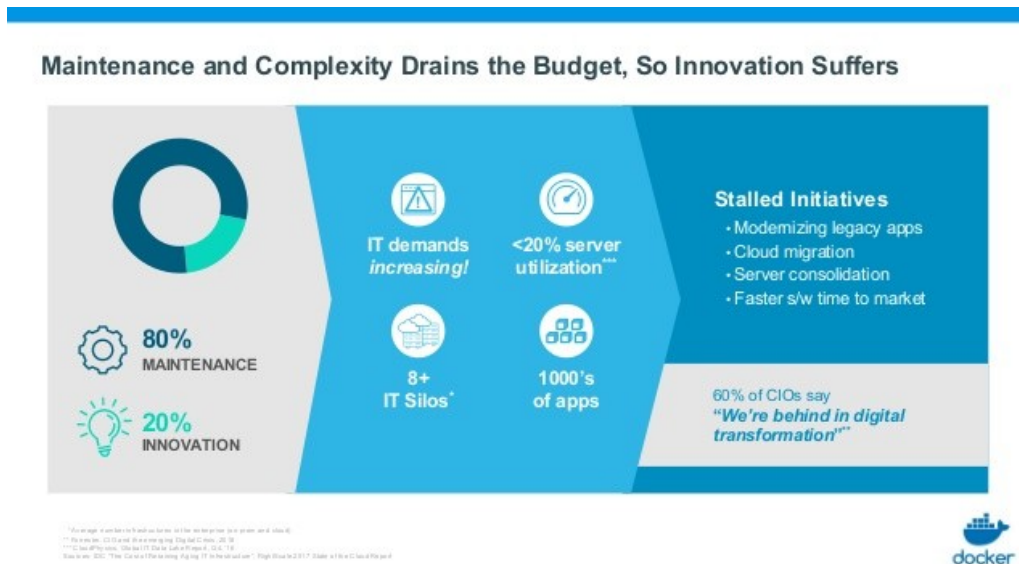


*Ilustrasi Container*

Sebuah fakta menarik lainnya adalah, dari hasil penelitian ternyata kegiatan orang-orang infrastruktur (*sysops*) selama ini sebanyak 80%nya hanya



dihabiskan untuk *maintenance*. Mulai dari melakukan *monitoring*, backup data, *troubleshooting*, migrasi, *setup* ini itu, monitoring lagi, troubleshooting lagi, dst. Seluruh kegiatan harian yang kompleks ini menyebabkan orang infrastruktur tidak memiliki waktu yang cukup untuk melakukan inovasi.



*Data hasil penelitian tentang sysops*

Dengan Docker, seluruh kompleksitas pekerjaan harian maintenance tersebut juga bisa hilang secara signifikan. Bayangkan saja, jika kita tidak menggunakan teknologi *Container* dari Docker maka berapa lama waktu yang dibutuhkan untuk bisa melakukan *deployment* puluhan server di banyak infrastruktur? Bisa sehari-hari dengan berbagai kompleksitas dan kemungkinan *error* yang terjadi.

Harus *setup* VM, setup OS, konfigurasi ini itu, setup ini itu. Sedangkan apabila kita menggunakan *Container*, kita cukup lakukan deployment menggunakan beberapa *script*, maka dalam hitungan menit seluruh server sudah siap tersedia.

**Docker is all about speed.**  
**Develop Faster.**  
**Build Faster.**  
**Test Faster.**  
**Deploy Faster.**  
**Update Faster.**  
**Recover Faster.**



*Moto Docker*

Docker menghilangkan kompleksitas untuk mencapai kecepatan. Kecepatan dalam seluruh aspek pada development aplikasi yang menyebabkan bisnis bisa tumbuh dengan lebih cepat yang tentunya menaikkan profit bisnis itu sendiri.

### 8.1.5. Contoh Mesin Kontainerisasi Lainnya

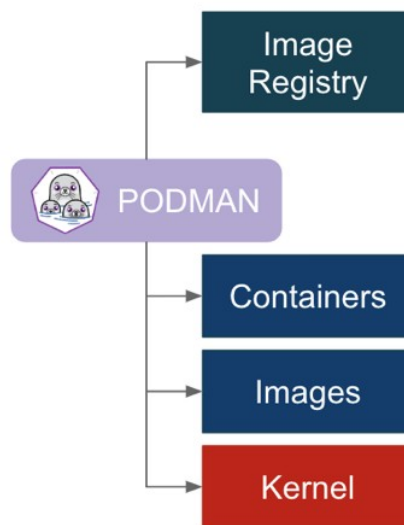
Perlu diketahui bahwa sesungguhnya mesin kontainerisasi yang digunakan di industri saat ini **tidak hanya Docker**, bahkan, ada yang performanya lebih baik dibandingkan docker. Namun, untuk belajar konsep dasar dari semua mesin kontainerisasi itu, kita dapat mempelajari Docker karena memang Docker merupakan *platform* yang umum digunakan. Berikut adalah beberapa contoh mesin kontainerisasi yang ada di Industri:

#### 8.1.5.1. PODMAN



Semenjak rilis *RHEL (Red Hat Enterprise Linux)* 7.6 dan 8.0, podman diadopsi oleh RHEL dan menggantikan Docker. Podman adalah *daemonless container engine* yang digunakan untuk menjalankan maupun memanajemen ekosistem container seperti pods, containers, container images, dan container volumes menggunakan libpod library. Apabila kita pernah menggunakan Docker sebelumnya, kita tidak perlu belajar cara menggunakan Podman karena hampir semua perintah di Docker bisa dijalankan di Podman.





Img Source:  
<https://www.i-3.co.id/2019/10/16/3-tools-alternatif-pengganti-docker/>

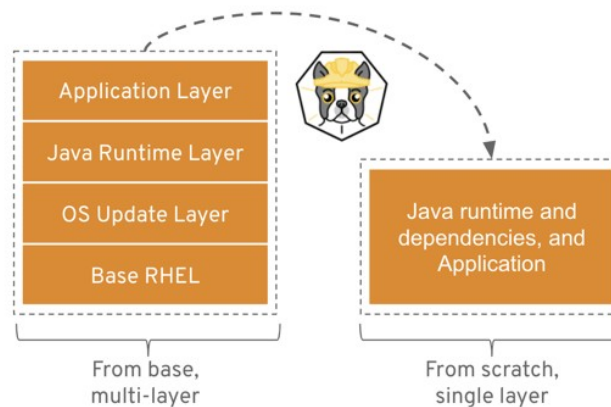
#### 8.1.5.2. Buildah



Buildah memungkinkan Anda untuk membangun dan memodifikasi container tanpa menggunakan *daemon* ataupun *docker*. Buildah akan tetap memakai *workflow dockerfile* yang sudah ada, tetapi memberikan para developer kontrol perihal *image*, *layers*, konten, dan *commits*. Buildah juga mengecilkan ukuran container image dengan menggunakan tools dari container host.







Img Src: <https://www.i-3.co.id/2019/10/16/3-tools-alternatif-pengganti-docker/>

### 8.1.5.3. Cri-O



*Container Runtime Interface - Orchestrator* (CRI-O) adalah *runtime* yang ringan dan telah dioptimalkan untuk kluster Kubernetes. Dikembangkan oleh Red Hat dan kemudian diserahkan kepada subset dari *Special Interest Group* (SIG) yang diberi kewenangan perihal pengembangan Kubernetes.

CRI-O memberikan alternatif untuk *relying* pada *Docker runtime* yang kompatibel dengan *container images* dengan format *Open Container Initiative* (OCI).



## 8.2. Docker Installation

### 8.2.1. Docker Edition

#### 8.2.1.1. CE vs EE

Docker memiliki 2 bersi yaitu *Community Edition* (CE) dan *Enterprise Edition* (EE). Versi CE ini adalah versi gratisan dan versi EE adalah versi berbayar. Dimana tentunya pada versi EE memiliki beberapa fitur-fitur unggulan dan support tim ahli yang tidak ada di versi CE.

Versi EE lebih cocok digunakan di perusahaan-perusahaan yang memang sudah memiliki dana dan benar-benar digunakan dalam skala Production yang besar. Sedangkan versi CE lebih cocok digunakan untuk riset atau digunakan skala production oleh perusahaan-perusahaan yang masih belum mempunyai cukup dana. Secara overall dalam versi CE Anda tetap bisa menggunakan Docker tanpa kurang suatu apapun. Namun tentunya Anda yang harus melakukannya semua sendiri.

Berikut adalah sedikit tabel perbandingan dari CE dan EE :

Daftar Fitur	<i>Enterprise Edition</i>	<i>Community Edition</i>
Harga	Berbayar	Gratis
Tim Support	Ada, resmi dari Docker.inc	Tidak ada
Rilis	<i>Stable</i> rilis saja	<i>Edge</i> dan <i>Stable</i>
Fitur tambahan	1. Partner resmi tersertifikasi 2. Fitur kelas enterprise lain seperti : - Layer 7 Proxy - Load balancer - dll	-



### 8.2.1.2. Edge vs Stable

Docker juga dalam setiap rilisnya dibagi menjadi 2 tipe. Yaitu *Edge* dan *Stable*. Dimana versi *Edge* dikhususkan untuk versi “coba-coba” sedangkan *Stable* adalah versi siap pakai untuk real implementasi.

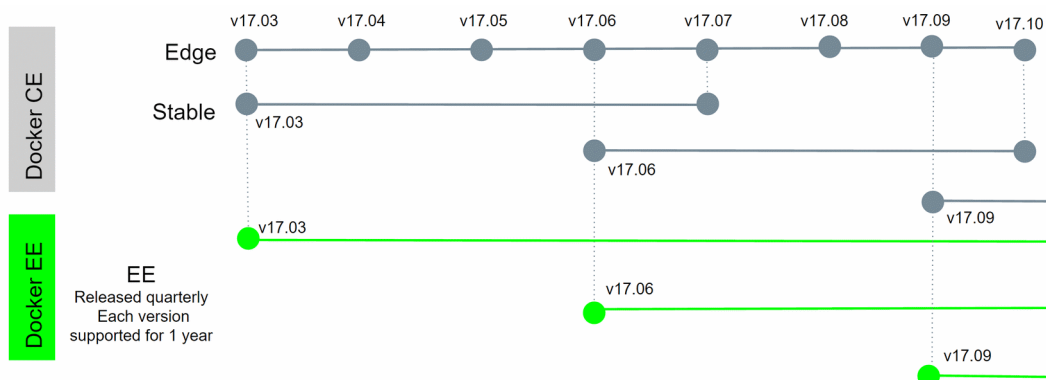
Secara umum karakteristik kedua versi ini adalah :

#### Versi *Edge*

1. Rilis tiap bulan
2. Disupport selama 1 bulan
3. Selalu dapat fitur terbaru
4. Cenderung banyak bug.

#### Versi *Stable*

1. Rilis setiap 4 bulan sekali.
2. Di support selama 4 bulan
3. Tidak dapat semua fitur-fitur terbaru.
4. Cenderung lebih stabil



Data version stable dan Edge

### 8.2.1.3. Penulisan Versi

Mulai tahun 2017, penulisan versi dari Docker menggunakan format YY.MM (tahun.bulan) mengikuti penulisan versi mirip salah satu distro Linux yaitu



Ubuntu. Jadi misalnya ada Docker versi 18.04 maka artinya Docker tersebut dirilis tahun 2018 bulan 4.

Namun jika Anda pernah melihat versi seperti 1.12 maupun 1.13 itu adalah versi terdahulu sebelum tahun 2017.

## 8.2.2. Langkah instalasi Docker

### 8.2.2.1. Linux

Untuk instalasi di Linux paling mudah menggunakan script berikut yang bisa dapat dari <https://get.docker.com> :

```
$ curl -fsSL get.docker.com -o get-docker.sh
```

Setelah itu jalankan scriptnya dan proses instalasi Docker akan berlangsung secara otomatis :

```
$ chmod +x get-docker.sh
```

```
$ sh get-docker.sh
```

Baik Linux distro berbasis RHEL (Centos, Fedora) maupun Debian (Ubuntu, Linux Mint), dapat menggunakan perintah tersebut.

Ada sedikit perbedaan instalasi Docker di Linux, yaitu di Linux kita belum terinstall komponen Docker secara lengkap. Yang belum terinstall adalah Docker *Compose* dan Docker *Machine*.

Instalasi Docker Compose dapat dilakukan dengan mengeksekusi script berikut :

```
$ sudo curl -L
"https://github.com/docker/compose/releases/download/1.28.5/docker-compose-$(
uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
$ docker-compose --version
```

Sedangkan untuk Docker Machine bisa eksekusi perintah berikut :

```
$ base=https://github.com/docker/machine/releases/download/v0.16.2
$ curl -L $base/docker-machine-$(uname -s)-$(uname -m) >/tmp/docker-machine
$ sudo install /tmp/docker-machine /usr/local/bin/docker-machine
```



#### 8.2.2.2. Pengecekan hasil instalasi

Untuk mengecek seluruh hasil instalasi dapat mengetikkan perintah berikut di terminal/cmd :

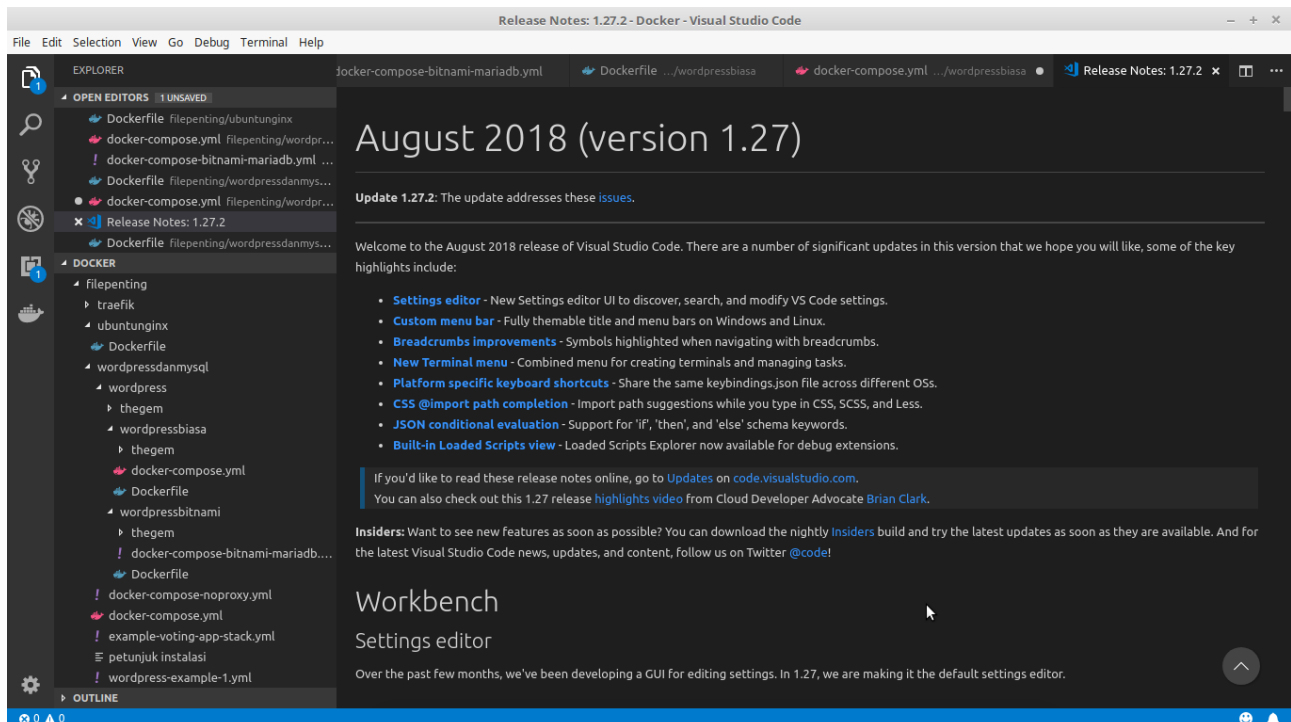
```
$ docker version
$ docker-compose --version
$ docker-machine --version
```

#### 8.2.2.3. Instalasi Software tambahan

Salah satu software tambahan yang direkomendasikan untuk diinstall adalah teks editor Visual Studio Code. Karena nantinya kita akan cukup sering untuk melakukan berbagai unsur “ngoding” selama menggunakan Docker. Oleh karena itu kita membutuhkan software teks editor yang cukup nyaman untuk digunakan ngoding.

- **Instalasi di Linux**

```
$ sudo dpkg -i namafilevisualstudioyangdiunduh.deb (jika menggunakan Ubuntu)
$ sudo rpm -ivh namafilevisualstudioyangdiunduh.deb (jika menggunakan
Centos/Fedora)
```



### 8.2.3. Exercise

#### Teori

1. Kenapa perlu mempelajari Docker?
2. Ceritakan apa yang akan Anda coba terapkan dengan Docker untuk kebutuhan Anda?

## 8.3. Perintah-Perintah Dasar Docker

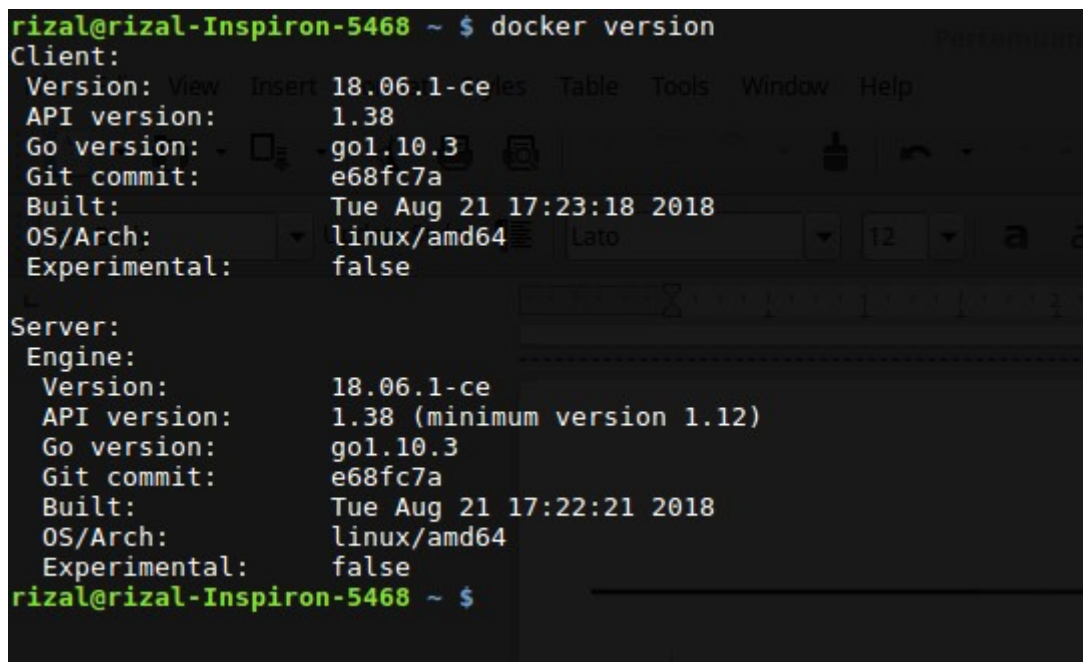
Pada bagian ini kita akan mempelajari beberapa perintah-perintah paling dasar dari Docker sebagai prinsip dasar agar kita menggunakan berbagai command Docker kedepannya.

### 8.3.1. Docker Version

Docker *version* merupakan perintah untuk mengecek versi dari Docker yang sudah terinstall. Selain itu juga untuk mengecek apakah Docker *Client* sudah dapat berkomunikasi dengan Docker Server.

Perintahnya :

```
$ docker version
```



```
rizal@rizal-Inspiron-5468 ~ $ docker version
Client:
Version: 18.06.1-ce
API version: 1.38
Go version: go1.10.3
Git commit: e68fc7a
Built: Tue Aug 21 17:23:18 2018
OS/Arch: linux/amd64
Experimental: false

Server:
Engine:
Version: 18.06.1-ce
API version: 1.38 (minimum version 1.12)
Go version: go1.10.3
Git commit: e68fc7a
Built: Tue Aug 21 17:22:21 2018
OS/Arch: linux/amd64
Experimental: false
rizal@rizal-Inspiron-5468 ~ $
```

### 8.3.2. Docker Info

Docker info merupakan command untuk mendapatkan informasi sistem Docker secara keseluruhan. Kita bisa mendapatkan info-info seperti :

1. Ada berapa *container* yang berjalan, yang di *pause*, yang dihentikan



2. Ada berapa *image*
3. Versi server
4. *Plugin, Driver, IP*
5. dll

```
rizal@rizal-Inspiron-5468 ~ $ docker info
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 4
Server Version: 18.06.1-ce
Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: bridge host macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog
Swarm: active
  NodeID: yjo4j374704b9i24jpxho352k
  Is Manager: true
  ClusterID: q9doau30y06njd1uif2gb14t
  Managers: 1
  Nodes: 1
  Orchestration:
    Task History Retention Limit: 5
  Experimental: false
  OS/Arch: linux/amd64
  Experimental: false
  rizal@rizal-Inspiron-5468 ~ $
```

Biasanya kita menggunakan command ini ketika ingin secara sekilas melihat sistem Docker yang sedang berjalan di Server.

### 8.3.3. Docker Help

Kita tidak perlu menghafal seluruh command yang ada pada Docker karena Docker sudah menyediakan seluruh petunjuk dan bantuan pada setiap commandnya. Kita cukup membaca penjelasan dari setiap command tersebut untuk mengerti seperti apa penggunaan dan formatnya.

Perintah untuk mendapatkan bantuan dari perintah-perintah Docker adalah :

```
$ docker --help
```

atau :

```
$ docker <command> --help
```

Disana akan tampil seluruh command beserta masing-masing penjelasannya.





Selain itu jangan lupa peran dari website Docker Docs. Seluruh penjelasan, petunjuk, konsep, dan contoh-contoh penggunaan command sudah dituliskan dengan sangat rapi oleh para pengembang Docker disana. Saat kita melakukan googling pun biasanya ujung-ujungnya diarahkan ke website tersebut. Jadi jika kita butuh bantuan, sering-sering kunjungi website ini.

### 8.3.4. Docker Command Structure

Penulisan format *command* Docker ini merupakan acuan setiap kita ingin melakukan *command* apapun kedepannya. Struktur ini berlaku universal sehingga walaupun *command*-nya berbeda-beda, biasanya strukturnya sama.

Ada 2 buah struktur command pada Docker yaitu :

#### Struktur Lama

```
docker <command> (options)
```

Contoh :

```
$ docker run -p 80:80 mysql
$ docker rm containerA
$ docker inspect
```

#### Struktur Baru

```
docker <command> <sub-command> (options)
```

Contoh :

```
$ docker container run -p 80:80 mysql
$ docker container rm containerA
$ docker container inspect
```

Pada struktur baru, setiap *command* menjadi lebih spesifik dan rapi secara tujuannya. Misal diatas kita jadi lebih tahu bahwa sesuatu yang di “run” adalah *container*. Berbeda dengan yang format penulisan lama bahwa hanya ada command “run” tapi tidak cukup jelas bahwa yang di run itu apa (bagi orang awam).

Contoh lain misal kita dapat menggunakan perintah berikut :

```
$ docker container rm testing
$ docker image rm testing
```

2 *command* diatas juga menunjukkan secara jelas bahwa yang ingin kita hapus adalah sebuah container dan sebuah image masing-masing bernama *testing*. Bandingkan dengan perintah hapus pada format lama seperti berikut :

```
$ docker rm testing
```

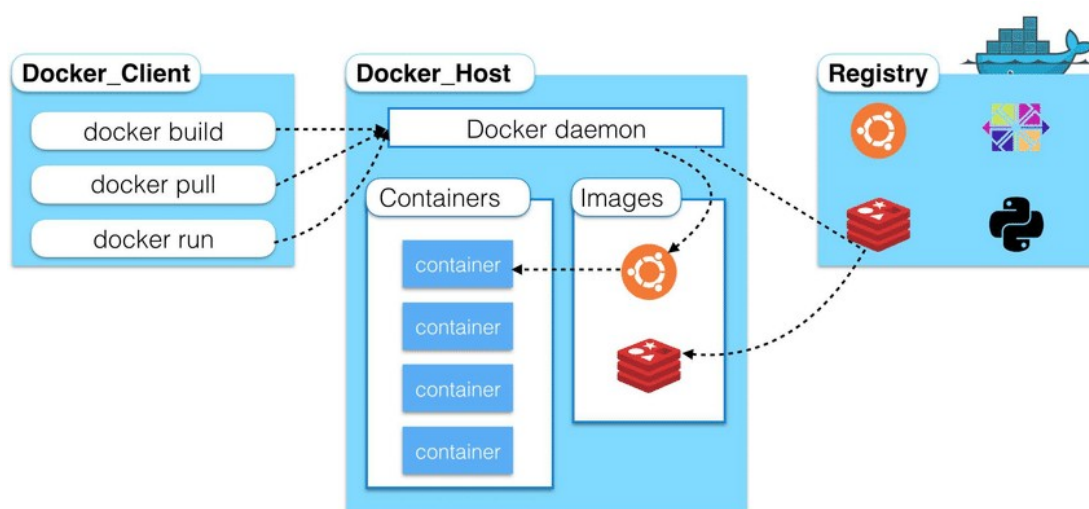
Secara kasat mata kita akan bingung bahwa "*testing*" yang dihapus diatas adalah container atau image.

Walaupun struktur penulisan yang lama masih dapat digunakan, sebaiknya kita tetap lebih mengutamakan menggunakan penggunaan struktur command yang baru.

## 8.4. Arsitektur Docker

Pada bagian ini kita akan mempelajari secara ringkas komponen apa saja yang terdapat didalam Docker dan seperti apa alur proses dari komponen-komponen tersebut. Setiap komputer/server yang sudah kita *install* Docker, maka pasti memiliki komponen-komponen ini dan juga memiliki alur proses yang sama.

Berikut adalah gambar dari arsitektur Docker yang akan kita bahas masing-masing berikutnya :



Arsitektur Docker



### 8.4.1. Client

**Docker Client** adalah seluruh *Command* Docker yang kita jalankan di layar terminal. Command ini nantinya akan masuk ke Docker *Host* untuk kemudian dieksekusi sesuai fungsinya.

### 8.4.2. Host dan Daemon

**Docker Host** merupakan mesin (komputer,server) yang terinstall Docker didalamnya. Didalam Docker Host terdapat Docker *Daemon* yang merupakan otak utama untuk memproses setiap permintaan (command) dari Docker *Client*. **Docker Daemon** yang akan mengatur bahwa misalnya *command* “docker image push” akan melakukan upload sebuah image ke *Registry*, atau *command* “docker container ls” akan menampilkan seluruh daftar container yang sedang berjalan di sistem.

Docker Host dan Docker Client sudah terinstall di mesin kita masing-masing saat kita melakukan instalasi Docker sebelumnya.

### 8.4.3. Image

**Image** adalah aplikasi yang ingin kita jalankan. Image berisi *binary*, *library* dan *source code* dari aplikasi yang ingin kita jalankan tersebut. Pada prakteknya Image ini akan sering kita isi komponen berikut :

1. Layanan (nginx,mysql)
2. Layanan + Aplikasi (nodejs + web apps berbasis nodejs)
3. OS (ubuntu,centos,alpine)
4. OS + Layanan (ubuntu+nginx, ubuntu+nodejs)
5. OS + Layanan + Aplikasi + Konfigurasi tambahan (ubuntu + nginx + wordpress + file website perusahaan + konfigurasi php.ini yang sudah dimodifikasi sesuai keinginan)



Semua tergantung dari kebutuhan dan kasus kita masing-masing. Namun konsep utamanya adalah sebisa mungkin kita membuat image yang sekecil dan seefisien mungkin. Misalnya kita tidak perlu mengisi image kita dengan OS Ubuntu apabila kita hanya ingin menjalankan aplikasi webserver berbasis XAMPP, cukup gunakan image berisi layanan XAMPP saja.

Image ini akan disimpan di Registry, baik itu Registry lokal maupun Registry publik seperti Docker Hub.

#### 8.4.4. Container

**Container** adalah instance yang menjalankan image dalam bentuk sebuah proses. Kuncinya adalah ini. Container hanya akan menjadi sebuah proses, bukan sebuah server *Virtual Machine* tersendiri. Sehingga sifatnya seperti menjalankan aplikasi biasa. Seperti layaknya menjalankan Firefox di OS Windows.

Kita dapat menjalankan banyak *Container* untuk 1 buah image yang sama.

#### 8.4.5. Registry

**Registry** merupakan tempat penyimpanan *Image* yang dapat berupa penyimpanan Publik maupun lokal. Siapapun bisa membuat Registry ini termasuk kita sendiri. Namun biasanya sudah sangat cukup menggunakan Registry publik seperti Docker Hub untuk memenuhi kebutuhan kita dalam menggunakan Docker sehari-hari.

Untuk pembelajaran kedepan, kalian wajib untuk mendaftar di Registry publik Docker Hub terlebih dahulu. Daftar melalui link berikut : <https://hub.docker.com>

#### 8.4.6. Alur penggunaan Docker secara umum

Secara umum berikut adalah alur cara kerja Docker :

1. Kita sebagai administrator mengeksekusi sebuah Command, dimana command ini akan disampaikan ke Docker *Host*.

2. Permintaan Docker *Client* (*command*) akan diperoses oleh Docker *Host* sesuai fungsinya masing-masing (melalui jasa Docker *Daemon*).
3. Kita dapat mengambil *image* dari *Registry* maupun menyimpannya kembali ke *Registry*.
4. Setiap Container akan menjalankan image yang sudah diambil tadi.
5. *User* (atau *Container* lainnya) dapat menikmati layanan dari *Container* yang dijalankan tsb. Misalnya *container* yang berisi Wordpress maka dapat benar-benar kita akses web wordpressnya, atau container yang berisi Database maka benar-benar bisa kita isi *database* di dalamnya, dll.

### 8.4.7. Komponen lainnya

Sebenarnya ada komponen-komponen lainnya dari Docker seperti Docker *Machine*, Docker *Swarm*, namun itu akan kita pelajari di materi-materi berikutnya agar tidak terlalu banyak informasi yang perlu dicerna.

## 8.5. Manajemen *Container*

Container merupakan komponen utama dari Docker. Kita tidak dapat mengotak-atik berbagai komponen lainnya seperti Image dan Registry selama kita belum paham terkait Container. Sehingga di bagian ini kita akan coba memahami cara kerja *Container* dengan langsung coba mempraktekkan beberapa command-commandnya.

### 8.5.1. Menjalankan *Container* Baru

Untuk menjalankan sebuah container baru kita dapat menggunakan format command berikut :

```
$ docker container run <options> image
```

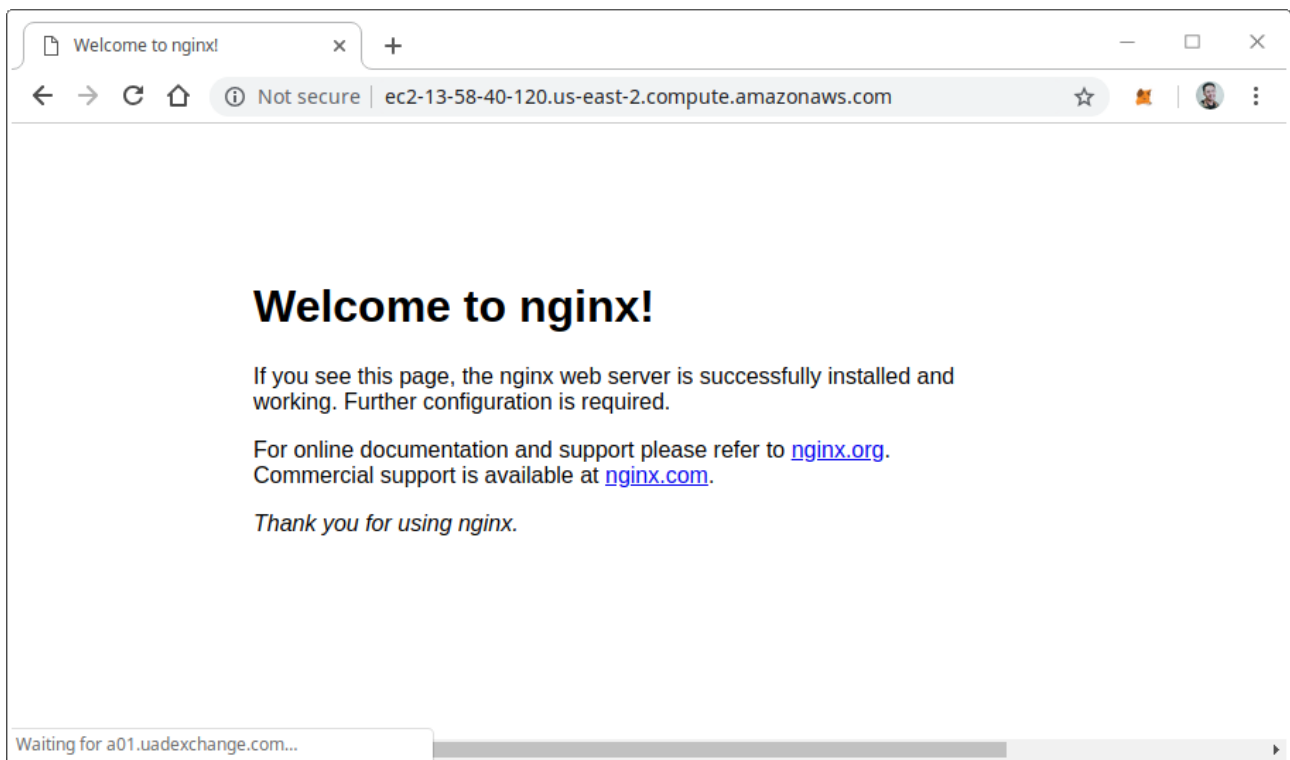
Contoh kita akan membuat sebuah Container yang berisi webserver Nginx. Maka kita bisa coba eksekusi command berikut :

```
$ docker container run -p 80:80 --name nginxhost nginx
```



```
rizal@rizal-Inspiron-5468 ~ $ docker container run -p 80:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
802b080ed6f79: Already exists
e9d0e0ea082b: Pull complete
d8b7092b9221: Pull complete
Digest: sha256:24a0c4b4a4c0eb97a1aabb8e29f18e917d05abfe1b7a7c07857230879ce7d3d3
Status: Downloaded newer image for nginx:latest
172.17.0.1 - - [25/Sep/2018:16:37:45 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36" "-"
172.17.0.1 - - [25/Sep/2018:16:37:46 +0000] "GET /favicon.ico HTTP/1.1" 404 571 "http://localhost/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36" "-"
2018/09/25 16:37:46 [error] 7#7: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "localhost", referer: "http://localhost/"
Commercial support is available at nginx.com.
Thank you for using nginx.
```

Sekarang coba buka browser dan arahkan ke alamat server yang kita miliki.



Webserver NGINX sudah dapat kita akses. Artinya Container sudah berhasil berjalan dengan baik.

Sebenarnya apa proses yang terjadi ketika kita menjalankan perintah tersebut? Berikut kira-kira penjelasannya :

1. Pertama-tama Docker akan mencari image bernama nginx di *registry* local terlebih dahulu.



2. Karena *image* nginx belum ada dilokal, maka Docker akan mengambil *image* nginx dari *registry* publik yaitu Docker Hub. *Image* ini kemudian akan disimpan di lokal.
3. Sebuah *container* baru bernama *nginxhost* akan dijalankan menggunakan *image* nginx tersebut.
4. Perintah *-p* (*publish*) membuat *port* 80 pada *host* (komputer/mesin yang menjalankan Docker) akan di buka.
5. Seluruh traffic menuju *port* 80 dari *host* akan diarahkan ke *port* 80 pada *container* *nginxhost*.

Sebagai catatan tidak boleh ada 2 *container* atau lebih yang menjalankan *port* yang sama. Jadi misalnya kita menjalankan 2 buah *container* berisi *image* NGINX, maka kita bisa jalankan di 2 *port* yang berbeda misalnya di 8080 dan 8081. Lalu bagaimana kalau kita ingin tetap membutuhkan menjalankan banyak *container* dengan *port* yang sama? Solusinya adalah *load balancer*.

Kita juga biasanya tidak menginginkan *Container* berjalan dalam terminal yang aktif. Karena jika kita jalankan *Container* dalam terminal aktif, ketika layar terminal ini kita tutup atau kita tekan CTRL + C maka *container* tersebut akan berhenti. Oleh karena itu kita bisa jalankan *Container* dengan opsi *-d* (*detach mode*) :

```
$ docker container run -d -p 80:80 --name nginxhost nginx
```

Lalu bagaimana kita bisa melihat *container-container* yang sedang berjalan dalam mode *detach*? Kita akan bahas pada bagian selanjutnya.

### 8.5.2. Melihat *Container* yang Berjalan dan Menghentikannya

Perintah untuk melihat *container* yang sedang berjalan adalah :

```
$ docker container ls
```



Disana akan terlihat *container-container* yang saat ini sedang berjalan di host tersebut. Lalu bagaimana kita menghentikan *container* yang sedang berjalan? Kita bisa gunakan perintah berikut :

```
$ docker container stop <container name>
```

Contoh :

```
$ docker container stop nginxhost
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
bfa19c9b68df   nginx    "/bin/bash"             12 minutes ago  Up 7 minutes    0.0.0.0:80->80/tcp  nginxhost
rizal@rizal-Inspiron-5468 ~ $ docker container stop nginxhost
nginxhost
rizal@rizal-Inspiron-5468 ~ $ docker container ls
```

Coba kita lihat kembali daftar *container* yang sedang berjalan menggunakan perintah :

```
$ docker container ls
```

Disana kita tidak akan dapat melihat *container-container* yang sudah pernah dihentikan. Terkadang kita tetap butuh informasi *container* yang sudah dihentikan itu apa saja agar dapat kita jalankan kembali di kemudian hari. Maka untuk melihat daftar *container* yang sudah berhenti kita bisa gunakan opsi *-a* seperti berikut :

```
$ docker container ls -a
```

### 8.5.3. Menjalankan *Container* yang sudah dihentikan

Kita dapat menjalankan kembali *Container* yang sudah berhenti menggunakan perintah :

```
$ docker container start <container name>
```

Contoh :

```
$ docker container start nginxhost
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
bfa19c9b68df   nginx    "/bin/bash"             15 minutes ago  Up 3 seconds    0.0.0.0:80->80/tcp  nginxhost
rizal@rizal-Inspiron-5468 ~ $ docker container start nginxhost
nginxhost
rizal@rizal-Inspiron-5468 ~ $ docker container ls
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS          NAMES
bfa19c9b68df   nginx    "/bin/bash"             15 minutes ago  Up 3 seconds    0.0.0.0:80->80/tcp  nginxhost
```

Perbedaan utama dari *command* *docker container run* dengan *docker container start* adalah, *command* *docker container run* selalu membuat *container* baru.





Sedangkan docker container *start* hanya menjalankan container yang sudah ada, namun sedang berhenti.

### 8.5.4. Docker *Logs* dan *Top*

Jika Anda terbiasa memantau informasi-informasi penting Server dari lognya, maka Anda dapat gunakan perintah berikut :

```
$ docker container logs <container name>
```

Contoh :

```
$ docker container logs nginxhost
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container logs nginxhost
172.17.0.1 - - [25/Sep/2018:17:12:42 +0000] "GET / HTTP/1.1" 304 0 "-" Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36 "-"
172.17.0.1 - - [25/Sep/2018:17:12:43 +0000] "GET / HTTP/1.1" 304 0 "-" Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36 "-"
172.17.0.1 - - [25/Sep/2018:17:12:43 +0000] "GET / HTTP/1.1" 304 0 "-" Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36 "-"
172.17.0.1 - - [25/Sep/2018:17:12:44 +0000] "GET / HTTP/1.1" 304 0 "-" Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36 "-"
172.17.0.1 - - [25/Sep/2018:17:12:44 +0000] "GET / HTTP/1.1" 304 0 "-" Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.92 Safari/537.36 "-"
```

Disana akan tampil informasi-informasi log standar tergantung layanan yang digunakan. Misalnya layanan nginx biasanya yang ditampilkan adalah log dari /var/nginx/access.log dan /var/nginx/error.log.

Kita juga dapat meliha proses apa saja yang dijalankan oleh suatu *Container* menggunakan perintah "*top*". Ini biasa kita gunakan untuk menganalisa container mana yang misalnya memakan RAM paling besar karena terlalu banyak menjalankan proses. Perintah *top* ini sangat mirip dengan perintah *top* Linux pada umumnya :

```
$ docker container top <container name>
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container top nginxhost
UID          PID          PPID         Comm           C support is available (TIME)  .com      TTY          TIME          CMD
root         25931        25909        nginx          0                               00:12       ?            00:00:00      nginx: master process nginx
-g daemon off;
systemd+     25995        25931        nginx          0                               00:12       ?            00:00:00      nginx: worker process
```

### 8.5.5. Menghapus *Container*

Untuk menghapus *container* kita dapat menggunakan perintah berikut :

```
$ docker container rm <container name>
```

Contoh :

```
$ docker container rm nginxhost
```



Namun perintah hapus ini tidak akan bisa digunakan pada *Container* yang sedang dalam keadaan berjalan. Kita dapat menggunakan opsi `-f` untuk memaksa menghapus *container* yang sedang berjalan :

```
$ docker container rm -f <container name>
```

Contoh :

```
$ docker container rm -f nginxhost
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container ls
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
bfa19c9b68df   nginx    "/bin/bash"   29 minutes ago   Up 14 minutes   0.0.0.0:80->80/tcp   nginxhost
rizal@rizal-Inspiron-5468 ~ $ docker container rm -f nginxhost
nginxhost
```

### 8.5.6. Exercise

#### Teori

1. Menurut Anda seperti apa proses yang terjadi jika kita menjalankan *Container* menggunakan *images* yang sudah ada di local ?

#### Praktek

1. Coba jalankan 3 buah *Container* menggunakan image yang sama yaitu : **mysql**. Tapi coba jalankan tanpa menggunakan opsi `--name`.
2. Tampilkan semua container yang sedang berjalan tersebut. Apa keanehan yang Anda temukan? Apa kesimpulan Anda?

## 8.6. Menggali Lebih dalam yang terjadi saat *Container* dijalankan

Pada materi sebelumnya kita sudah sempat menyinggung terkait apa sebenarnya yang terjadi ketika perintah Docker run dijalankan. Disini kita akan coba ulas sedikit lebih dalam terkait hal tersebut.



Saat perintah Docker run di eksekusi, sebenarnya cukup banyak hal yang terjadi di balik layar. Bukan hanya semata-mata jalankan Container, lalu selesai.

Contohnya kita coba jalankan sebuah container dengan perintah berikut :

```
$ docker container run -d -p 80:80 --name httpdhost httpd
```

Maka yang terjadi adalah :

1. Pertama-tama Docker akan mencari image bernama httpd di *registry* local terlebih dahulu.
2. Karena image httpd belum ada di lokal, maka Docker akan mengambil *image* httpd dari *registry* publik yaitu Docker Hub. Jika kita tidak menentukan versi spesifik dari image yang ingin kita ambil, maka secara *default* Docker akan mengambil *image* dengan tag "*latest*". Kita dapat tentukan versi tertentu misalnya httpd:1.11.1.
3. Sebuah *container* baru bernama httpdhost akan dijalankan menggunakan *image* httpd tersebut.
4. *Virtual IP* dan *private network* akan dipasang pada *Container* tersebut.
5. Perintah -p (*publish*) membuat *port* 8080 pada *host* akan di buka.
6. Seluruh *traffic* menuju *port* 8080 dari host akan diarahkan ke port 80 pada *container* httpdhost.
7. Container akan dijalankan menggunakan *command* utama yang sudah di tentukan pada Dockerfile (ini akan kita bahas pada materi berikutnya)

Dari banyaknya proses ini, kita dapat memodifikasi perintah docker *run* berdasarkan kebutuhan :

```
$ docker container run -d -p 8080:80 --name httpdhost httpd:2.4.35 httpd -e debug
```

Misalnya pada perintah diatas kita mengubah port host yang awalnya 80 menjadi 8080, mengubah versi *image* httpd menjadi 2.4.35 dan menggunakan



*Command custom* berupa “`httpd -e debug`” dimana *command* ini merupakan *command* untuk menjalankan webserver `httpd` dengan *mode log debug*.

Berikut adalah hasilnya, dimana *log* dari *container* ini yang biasanya hanya sedikit, menjadi sangat lengkap karena kita menjalankannya dengan *mode debug* :

```
rizal@rizal-Inspiron-5468 ~$ docker container run -d -p 8080:80 --name httpd httpd -e debug
d8fabd2c13e523fbc85bcc555a0fbab310bc39784bc7ce14e84297b32f86b57e
rizal@rizal-Inspiron-5468 ~$ docker container logs httpd
[Wed Sep 26 07:01:13.170810 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module mpm_event_module from /usr/local/apache2/modules/mod_mpm_event.so
[Wed Sep 26 07:01:13.198857 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authn_file_module from /usr/local/apache2/modules/mod_authn_file.so
[Wed Sep 26 07:01:13.199162 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authn_core_module from /usr/local/apache2/modules/mod_authn_core.so
[Wed Sep 26 07:01:13.199494 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authz_host_module from /usr/local/apache2/modules/mod_authz_host.so
[Wed Sep 26 07:01:13.199609 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authz_groupfile_module from /usr/local/apache2/modules/mod_authz_groupfile.so
[Wed Sep 26 07:01:13.199774 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authz_user_module from /usr/local/apache2/modules/mod_authz_user.so
[Wed Sep 26 07:01:13.199957 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module authz_core_module from /usr/local/apache2/modules/mod_authz_core.so
[Wed Sep 26 07:01:13.200155 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module access_compat_module from /usr/local/apache2/modules/mod_access_compat.so
[Wed Sep 26 07:01:13.200434 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module auth_basic_module from /usr/local/apache2/modules/mod_auth_basic.so
[Wed Sep 26 07:01:13.200715 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module reqtimeout_module from /usr/local/apache2/modules/mod_reqtimeout.so
[Wed Sep 26 07:01:13.200948 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module filter_module from /usr/local/apache2/modules/mod_filter.so
[Wed Sep 26 07:01:13.201162 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module mime_module from /usr/local/apache2/modules/mod_mime.so
[Wed Sep 26 07:01:13.201360 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module log_config_module from /usr/local/apache2/modules/mod_log_config.so
[Wed Sep 26 07:01:13.201532 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module env_module from /usr/local/apache2/modules/mod_env.so
[Wed Sep 26 07:01:13.201697 2018] [so:debug] [pid 1:tid 140711875110784] mod_so.c(266): AH01575: loaded module headers_module from /usr/local/apache2/modules/mod_headers.so
```

Kita juga bisa melakukan berbagai variasi dan perubahan lain selama masih memenuhi standar struktur *command* dari *docker container run* yaitu :

```
$ docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

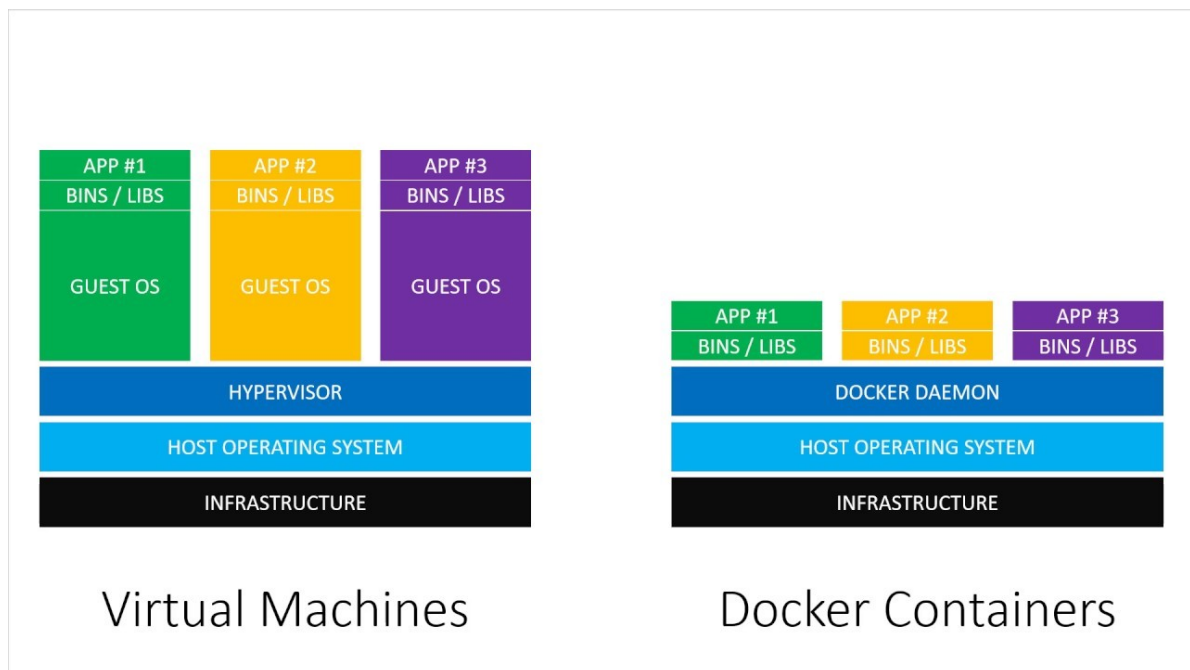
Misal kita bisa menghubungkan container kita dengan *private* network bernama *net-kuasai*, atau menentukan agar IP dari container ini adalah *172.16.0.100*, dll. Itu semua bisa dilakukan.

Disini mungkin Anda masih cukup bingung, namun yang pasti semua proses diatas akan kita kupas satu persatu selama berjalannya kelas. Sehingga di akhir, semuanya baru menjadi masuk akal bagi Anda.

## 8.7. Container vs VM

Seringkali materi yang membahas terkait *Container* selalu membandingkan *Container* dengan VM seperti ini :





*Ilustrasi perbandingan VM dengan Docker*

Ada yang menyebut *Container* adalah mini VM juga. Padahal bahkan *Container* bisa dibilang tidak bisa sama sekali dibandingkan dengan VM. Karena Container hanya menjalankan sebuah proses pada *Host*, layaknya kita menjalankan sebuah aplikasi Firefox di OS Windows.

Untuk lebih mudah memahaminya kita coba jalankan sebuah container menggunakan image mongodb :

```
$ docker container run -d --name mongo mongo
```

Kemudian kita coba lihat proses apa saja yang muncul dari container ini :

```
$ docker container top mongo
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container top mongo
UID          PID         PPID        C          STIME       TTY         TIME        CMD
999          10047      10027       1          12:17       ?           00:00:01    mongod --bind_ip_all
```

Terlihat ada sebuah proses dengan PID 10047 yang menjalankan command mongod. Di Linux, setiap proses akan memiliki ID unik tersendiri bernama PID.

Nah sekarang kita buktikan bahwa sebenarnya proses ini hanyalah proses yang dijalankan di host layaknya aplikasi biasa. Kita dapat menggunakan perintah :

```
$ ps aux
```



```

root      8942  0.0  0.0   0   0 ?        S    12:04   0:00 [kworker/u8:1]
rizal     9034 40.8  9.3 1096872 368044 ?        Sl   12:05   6:11 /opt/google/chrome/chrome --type=renderer --field-trial-handle=12028001816946747308,2825656927067075462
rizal     9045  0.0  2.3 714796 90800 ?        Sl   12:05   0:00 /opt/google/chrome/chrome --type=renderer --field-trial-handle=12028001816946747308,2825656927067075462
root      9140  0.0  0.0   0   0 ?        S    12:06   0:00 [kworker/0:1]
root      9504  0.0  0.0   0   0 ?        S    12:09   0:00 [kworker/2:0]
rizal     9573  0.0  1.1 656108 46160 ?        Sl   12:10   0:00 /opt/google/chrome/chrome --type=renderer --field-trial-handle=12028001816946747308,2825656927067075462
root      9610  0.0  0.0   0   0 ?        S    12:11   0:00 [kworker/u8:0]
root      9657  0.0  0.0   0   0 ?        S    12:13   0:00 [kworker/1:2]
root      9735  0.0  0.0   0   0 ?        S    12:14   0:00 [kworker/2:2]
root      9737  0.0  0.0   0   0 ?        S    12:14   0:00 [kworker/u8:3]
root      9774  0.0  0.0   0   0 ?        S    12:15   0:00 [kworker/3:0]
root      9804  0.0  0.0   0   0 ?        S    12:15   0:00 [kworker/0:0]
root     10027  0.0  0.1 7496 4868 ?        Sl   12:17   0:00 docker-containerd-shim -namespace moby -workdir /var/lib/docker/containerd/daemon/io.containerd.runtime
999      10047  1.0  1.9 1090976 77360 ?        Ssl  12:17   0:01 mongod --bind_ip_all
root     10204  0.0  0.0   0   0 ?        S    12:18   0:00 [kworker/1:0]
root     10277  0.0  0.0   0   0 ?        S    12:19   0:00 [kworker/2:1]
rizal    10330 18.9  3.8 1016000 149904 ?        Sl   12:20   0:07 /usr/bin/perl /usr/bin/shutter
root     10335  0.0  0.0   0   0 ?        S    12:20   0:00 [kworker/3:2]
rizal    10378  0.0  0.0 37368 3232 pts/0    R+   12:20   0:00 ps aux
rizal@rizal-Inspiron-5468 ~ $

```

Disana terlihat ada proses mongodb dari sekian banyak proses yang berjalan di host tersebut. Bahkan jika Anda teliti lebih dalam, pada gambar diatas juga ada proses yang menjalankan Google Chrome. Artinya disini level mongodb dan Google Chrome itu setara, sama-sama hanya sebuah proses.

Anda bisa coba hentikan container Mongodb tersebut dan bandingkan bahwa proses mongodb sudah menghilang dari host.

Inilah yang menyebabkan kenapa Container begitu ringan, jauh dibandingkan dengan VM. Jika kita menjalankan mongodb di dalam VM maka kita perlu menjalankan OS yang lengkap beserta seluruh proses-proses yang lengkap pula di dalamnya. Sedangkan jika pada Container, hanya cukup menjalankan 1 proses mongodb itu saja, tidak ada proses-proses lainnya.

## 8.8. Monitoring Container

Ketika sebuah *Container* sudah dijalankan, tentunya kita ingin tahu sebenarnya apa yang sedang terjadi pada *Container* tersebut saat ini. Misalnya kita ingin melihat proses apa saja yang dijalankan, bagaimana statistik penggunaan RAM dan CPU dari Container tersebut, dsb. Disini kita akan coba mempelajari beberapa perintah lain selain Docker *Logs* dan Docker *Top* yang sudah kita pelajari di pertemuan sebelumnya.

### 8.8.1. Top

Ini merupakan perintah untuk menampilkan list proses apa saja yang dijalankan oleh sebuah *Container*. Misalnya Anda dapat mengetahui di *Container* NGINX, ada berapa proses nginx *workers* yang sedang berjalan,





apakah nginxnya sedang mati atau tidak, dsb. Jika Anda memang sudah cukup sering mengelola server, pasti sudah tidak asing dengan perintah top ini.

Sebelumnya kita coba jalankan 2 buah container terlebih dahulu :

```
$ docker container run -d -p 80:80 --name nginx nginx
$ docker container run -d --name mongo mongo
```

Setelah itu coba lihat masing-masing proses yang dijalankan oleh masing-masing *container* tersebut :

```
$ docker container top nginx
$ docker container top mongo
```

```
rizal@rizal-Inspiron-5468 ~ $ docker container top nginx
rtdm nginx
UID PID PPID C TIME TTY TIME CMD
root 12388 12365 0 12:51 ? 00:00:00 nginx: master process nginx
-g daemon off;
systemd+ 12453 12388 0 12:51 ? 00:00:00 nginx: worker process
rizal@rizal-Inspiron-5468 ~ $ docker container top mongo
UID PID PPID C TIME TTY TIME CMD
999 14342 14317 0 13:09 ? 00:00:12 mongod --bind_ip_all
```

### 8.8.2. Inspect

*Command Inspect* akan menampilkan informasi yang sangat lengkap terkait konfigurasi bagaimana Container ini dijalankan. Kita coba lihat dengan perintah berikut :

```
$ docker container inspect <nama container>
```

Contoh :

```
$ docker container inspect nginx
```

Disana anda akan mendapatkan informasi yang begitu banyak seperti berapa virtualisasi ip dari container ini, hostnamenya apa, dll.

```
ubuntu@ip-172-31-34-193:~$ sudo docker container inspect nginx
[
  {
    "Id": "90e750cc05fa7b61b6d8fd626ba948affd60e91d8f8fe35ed12c75344d5d3fdf",
    "Created": "2019-03-12T10:29:14.247002766Z",
    "Path": "nginx",
    "Args": [
      "-g",
      "daemon off;"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 28457,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2019-03-12T10:29:14.738710206Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    }
  }
]
```



Seluruh informasi ini istilahnya adalah *metadata*. Disini kita mungkin tidak akan terlalu mengerti apa maksud semua ini, namun semuanya akan terasa semakin masuk akal selama berjalannya kelas ini kedepannya. Ketika kita sudah mulai mengotak-atik terkait *Volume*, *restart policy*, *Driver*, dll.

Disini Anda cukup memahami bahwa metadata ini digunakan sebagai petunjuk bagaimana suatu *Container* dijalankan.

### 8.8.3. Stats

*Command* ini memberikan kita informasi sederhana terkait statistik penggunaan dari setiap *container* yang berjalan. Seperti berapa penggunaan RAM, CPU, dan Network dari setiap *Container*. Memang bukan monitoring yang super lengkap dan *fancy*, namun cukup untuk memberikan kita gambaran ringkas terkait penggunaan *resource container* kita.

```
$ docker container stats
```

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
18fb2af47e0e	nginx	0.00%	2.535MiB / 3.758GiB	0.07%	38.1kB / 0B	18.3MB / 0B	2
707e047d145a	mongo	0.66%	39.98MiB / 3.758GiB	1.04%	9.06kB / 0B	106MB / 238kB	26

Misalnya pada gambar diatas kita dapat melihat bahwa container mongo memakan RAM 39MB saat container baru dijalankan.

## 8.9. Masuk ke dalam *Container*

Jika anda sudah terbiasa mengelola server, maka anda pasti sering menggunakan fasilitas *remote server* seperti SSH untuk bisa masuk ke dalam server. Dengan *Container*, anda bahkan tidak perlu melakukan itu. Kita bisa masuk langsung ke dalam *Shell Container* memanfaatkan mode interaktif.

### 8.9.1. Menjalankan Container dengan mode Interaktif

Perintahnya adalah sebagai berikut :

```
$ docker container run -it nginx bash
```

Diatas adalah contohnya kita menjalankan container menggunakan image nginx secara mode interaktif dengan *shell Bash*. Hasilnya Anda akan mendapatkan shell seperti berikut :





```
root@3642bcc47543:/#
```

Jika sudah didalam *shell container* seperti ini, maka semua command yang kita lakukan akan terjadi di dalam *container* tersebut, bukan pada Host.

Contohnya disini kita bisa mengeksekusi perintah **cat /etc/nginx/nginx.conf** untuk melihat isi dari konfigurasi NGINX kita :

```
root@3642bcc47543:/# cat /etc/nginx/nginx.conf
```

```
root@e35b7dalf567:/# cat /etc/nginx/nginx.conf
user nginx;
worker_processes 1;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';
```

Bahkan kita dapat melakukan instalasi paket tambahan seperti nano :

```
root@3642bcc47543:/# apt-get update && apt-get install nano
```

```
root@3642bcc47543:/# nano /etc/nginx/nginx.conf
```

Jika sudah selesai, Anda dapat mengetikkan exit untuk keluar dari *Shell*. Sayangnya jika kita sudah keluar dari *Shell* seperti ini, maka *container* otomatis akan dihentikan.

### 8.9.2. Docker Exec

Lalu bagaimana agar *Container* tetap berjalan dan kita tetap bisa masuk ke dalam Shell ? Pertama-tama kita perlu untuk menjalankan *Container* seperti biasa dengan mode detach.

```
$ docker container run -d -p 80:80 --name nginx nginx
```

Container nginx ini sama sekali tidak memiliki mode interaktif bukan? Namun kita tetap bisa masuk ke dalamnya menggunakan command Exec. Berikut adalah contohnya :



```
$ docker container exec -it nginx bash
```

Maka kita akan masuk ke dalam *shell mode* interaktif, sama seperti yang kita bahas sebelumnya. Saat kita exit dari shell pun *container* tetap akan berjalan, karena Docker Exec ini sifatnya hanya sebagai proses tambahan bukan proses utama dari Container-nya itu sendiri.

```
rizal@rizal-Inspiron-5468 ~ $ docker container run -d -p 80:80 --name nginx nginx
18fb2af47ebe742e4ac05569d6efb5ae81fc11e7b42db57426c24c17563953bd
rizal@rizal-Inspiron-5468 ~ $ docker container exec -it nginx bash
root@18fb2af47ebe:/# exit
exit
rizal@rizal-Inspiron-5468 ~ $ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
18fb2af47ebe	nginx	"nginx -g 'daemon of..."	2 minutes ago	Up 2 minutes	0.0.0.0:80->80/tcp	nginx

## 8.10. Exercise

### Teori

1. Menurut Anda apakah mungkin VM bisa lebih ringan dari Container jika isi aplikasinya sama? Misalnya sama-sama OS Ubuntu atau sama-sama Wordpress. Jika mungkin jelaskan alasannya, jika tidak jelaskan alasannya.
2. Menurut Anda bisakah kita masuk ke shell Container menggunakan SSH? Jika bisa jelaskan step by step caranya, jika tidak bisa jelaskan alasannya.

### Praktek

1. Coba jalankan dua buah Container yaitu Ubuntu 14.04 dan Centos 7 dengan mode detach.
2. Dapatkan informasi sebanyak mungkin terkait kedua container ini dan bandingkan secara kasat mata *Container* mana yang menurut Anda paling ringan dijalankan.
3. Masuklah ke dalam *shell container* keduanya dan install aplikasi *curl*. Cek perbedaan versi kedua aplikasi tersebut. Disini Anda belajar bagaimana melakukan manajemen paket sederhana untuk distro Linux yang berbeda. Seperti biasa, googling adalah teman Anda.



4. Keluar dari container dan hapus container tersebut. Presentasikan hasil yang Anda dapat.

## 8.11. Summary

1. Masalah utama yang dipecahkan oleh Docker adalah menghilangkan kompleksitas dalam infrastruktur. Dengan teknologi *Container* pada Docker, seluruh proses development aplikasi menjadi jauh lebih cepat. Efek domino pun terasa, mulai dari menurunnya biaya infrastruktur, inovasi bisnis meningkat, produktifitas meningkat, hingga peningkatan efisiensi penggunaan server.
2. Gunakan Docker EE hanya ketika perusahaan Anda memiliki cukup dana dan benar-benar akan implementasi Docker skala *production* besar-besaran.
3. Sebaiknya gunakan Docker versi *Edge* jika ingin testing-testing, dan gunakan versi *Stable* jika ingin menggunakan Docker di server *Production*.
4. Instalasi Docker caranya cukup mudah dan dapat dilakukan di *platform* Windows, Mac, maupun Linux.
5. Kita perlu memahami perintah-perintah dasar Docker sebagai fundamental melakukan berbagai perintah Docker lainnya.
6. *Image* berisi aplikasi yang ingin kita jalankan dan *Container* adalah *instance* yang menjalankan *image* sedangkan Registry adalah tempat menyimpan *image*.
7. *Container* adalah komponen terpenting dari Docker yang harus dipahami terlebih dahulu sehingga kita juga perlu menguasai manajemen *Container*.



8. *Container* bahkan tidak bisa dibandingkan dengan VM. Karena *Container* hanyalah berupa proses untuk masuk ke dalam *Container*