Atgen

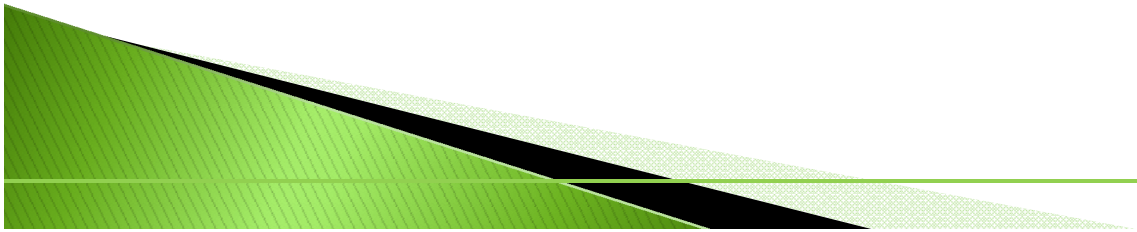
Docker & Kubernetes

Agenda

- Introduction
- Docker Components
- Classroom Environment
- Containers
- Docker – Images
- Docker – Building Images
- Deep Dive – Images
- Deep Dive – Containers
- Container Network Model
- Docker Volumes
- Kubernetes
- Deep Dive – Kubernetes
- Managing – Kubernetes

Session: 1

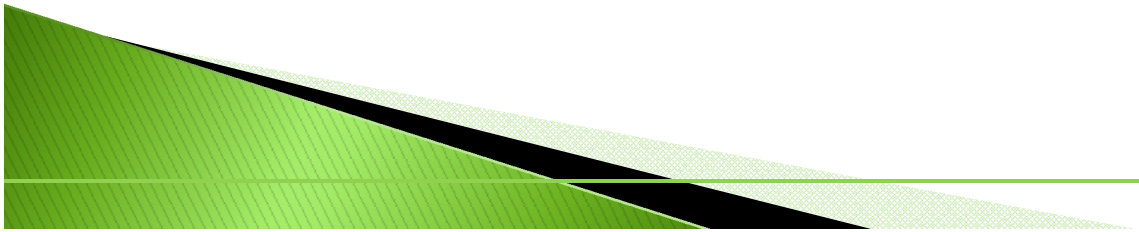
Introduction



Atgen

History of Containers

- Containers have been around for a very long time indeed.
  - IBM VM/370 (1972)
  - FreeBSD jails (1999)
  - Linux VServers (2001)
  - Solaris Containers (2004)



Atgen

Origins of Docker Project

- ‘dotCloud’ was operating a PaaS, using a custom container engine.
- This engine was based on ‘OpenVZ’ (and later, LXC) and AUFS.
- It started (circa 2008) as a single Python script.
- By 2012, the engine had multiple (~10) Python components. (and ~100 other micro-services!)
- End of 2012, ‘dotCloud’ refractors this container engine.
- The codename for this project is "Docker."

First Public Release

- March 2013, PyCon, Santa Clara:

  "Docker" is shown to a public audience for the first time.
- It is released with an open source license.
- Very positive reactions and feedback!
- The ‘dotCloud’ team progressively shifts to Docker development.
- The same year, ‘dotCloud’ changes name to Docker.
- In 2014, the PaaS activity is sold.

First Users of Docker(2013-14)

- PAAS builders (Flynn, Dokku, Tsuru, Deis…)
- PAAS users (those big enough to justify building their own)
- CI platforms
- developers, developers, developers, developers

Becomes industry standard(15-16)

- Docker reaches the symbolic 1.0 milestone.
- Existing systems like Mesos and Cloud Foundry add Docker support.
- Standards like OCI, CNCF appear.
- Other container engines are developed.

Docker becomes a platform

- The initial container engine is now known as "Docker Engine“.
- Other tools are added:
  - Docker Compose (formerly "Fig")
  - Docker Machine
  - Docker Swarm
  - Kitematic
  - Docker Cloud (formerly "Tutum")
  - Docker Datacenter
- Docker Inc. launches commercial offers.

About Docker Inc.

- Docker Inc. used to be ‘dotCloud’ Inc.
- ‘dotCloud’ Inc. used to be a French company.
- Docker Inc. is the primary sponsor and contributor to the Docker Project:
  - Hires maintainers and contributors.
  - Provides infrastructure for the project.
  - Runs the Docker Hub.
- HQ in San Francisco.
- Backed by more than 100M in venture capital.

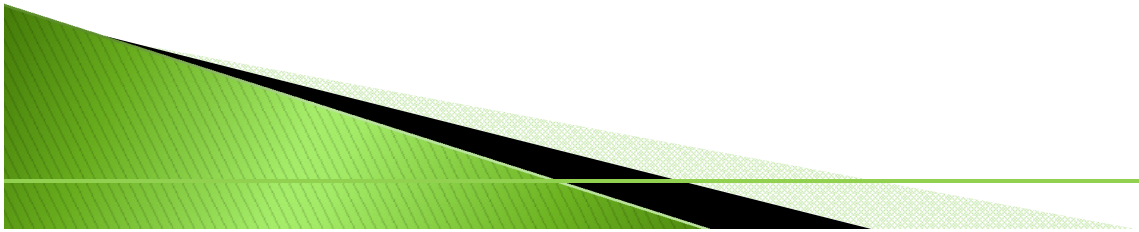
Why containers(non-technical)

- The software industry has changed.
- Before:
  - monolithic applications
  - long development cycles
  - single environment
  - slowly scaling up
- Now:
  - decoupled services
  - fast, iterative improvements
  - multiple environments
  - quickly scaling out

Atgen

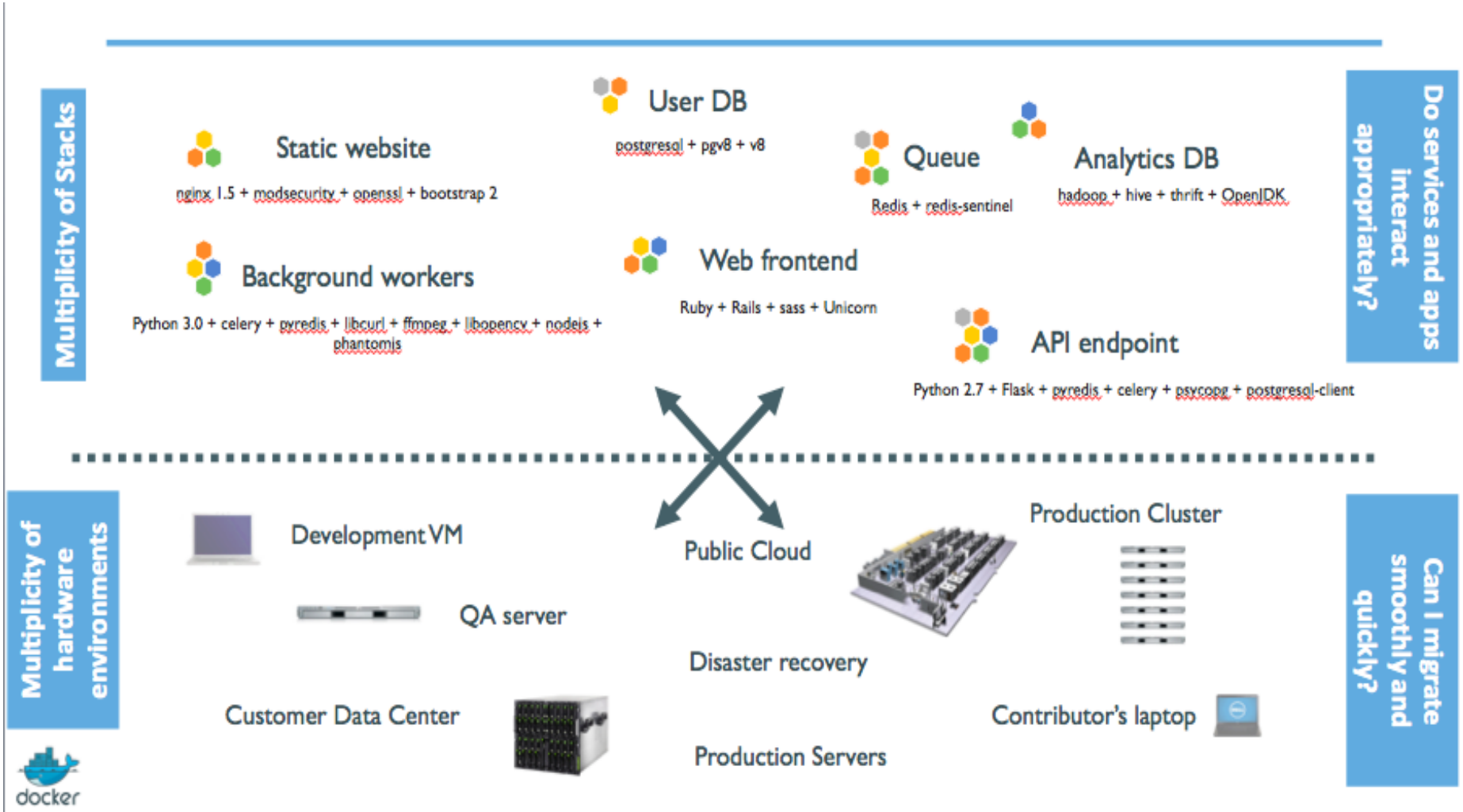
Deployment becomes very complex

- Many different stacks:
  - Languages
  - Frameworks
  - Databases
- Many different targets:
  - individual development environments
  - pre-production, QA, staging...
  - production: on physical, cloud, hybrid



Atgen

Deployment Problem



Atgen

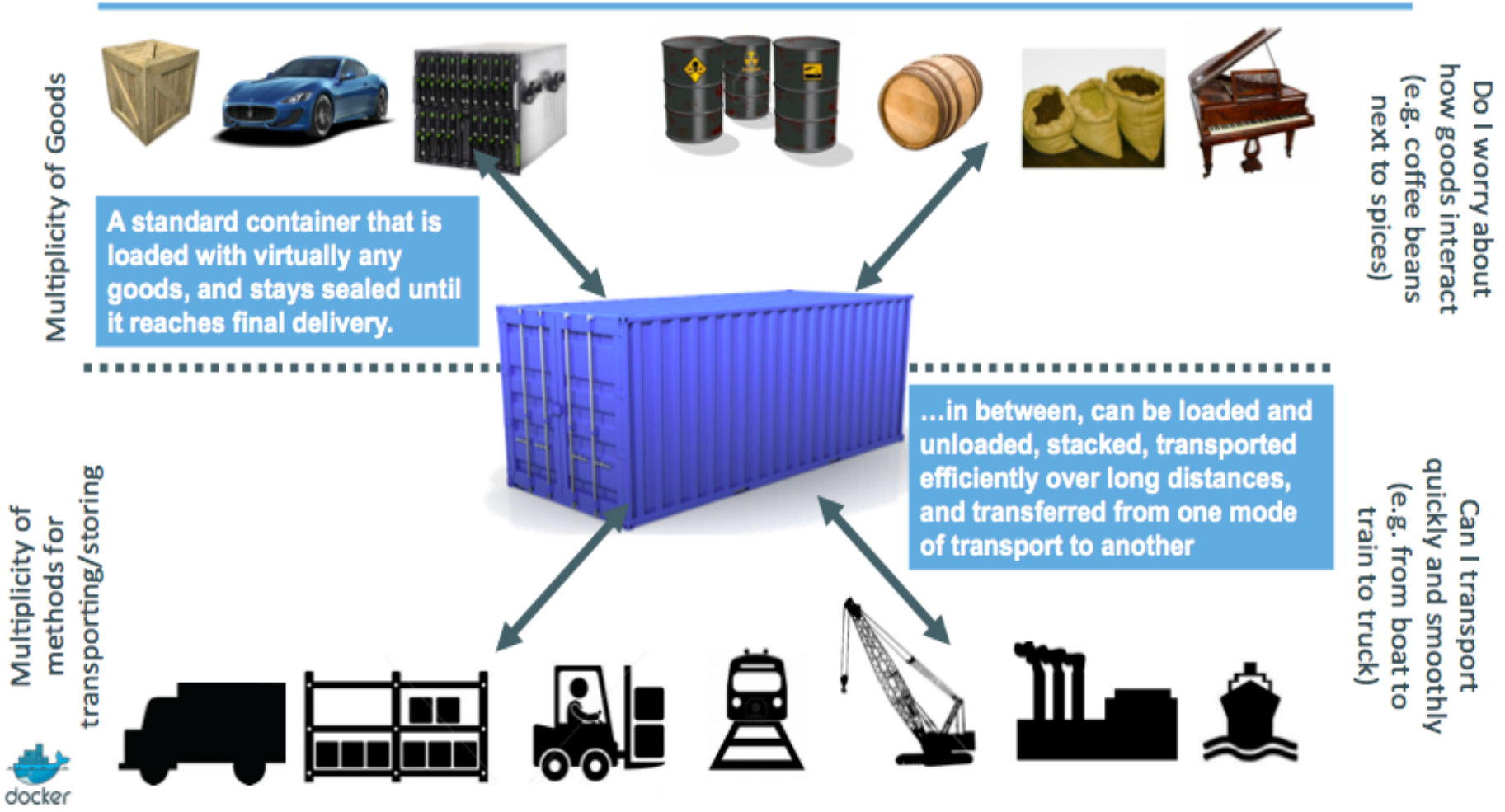
Matrix Checks

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

docker

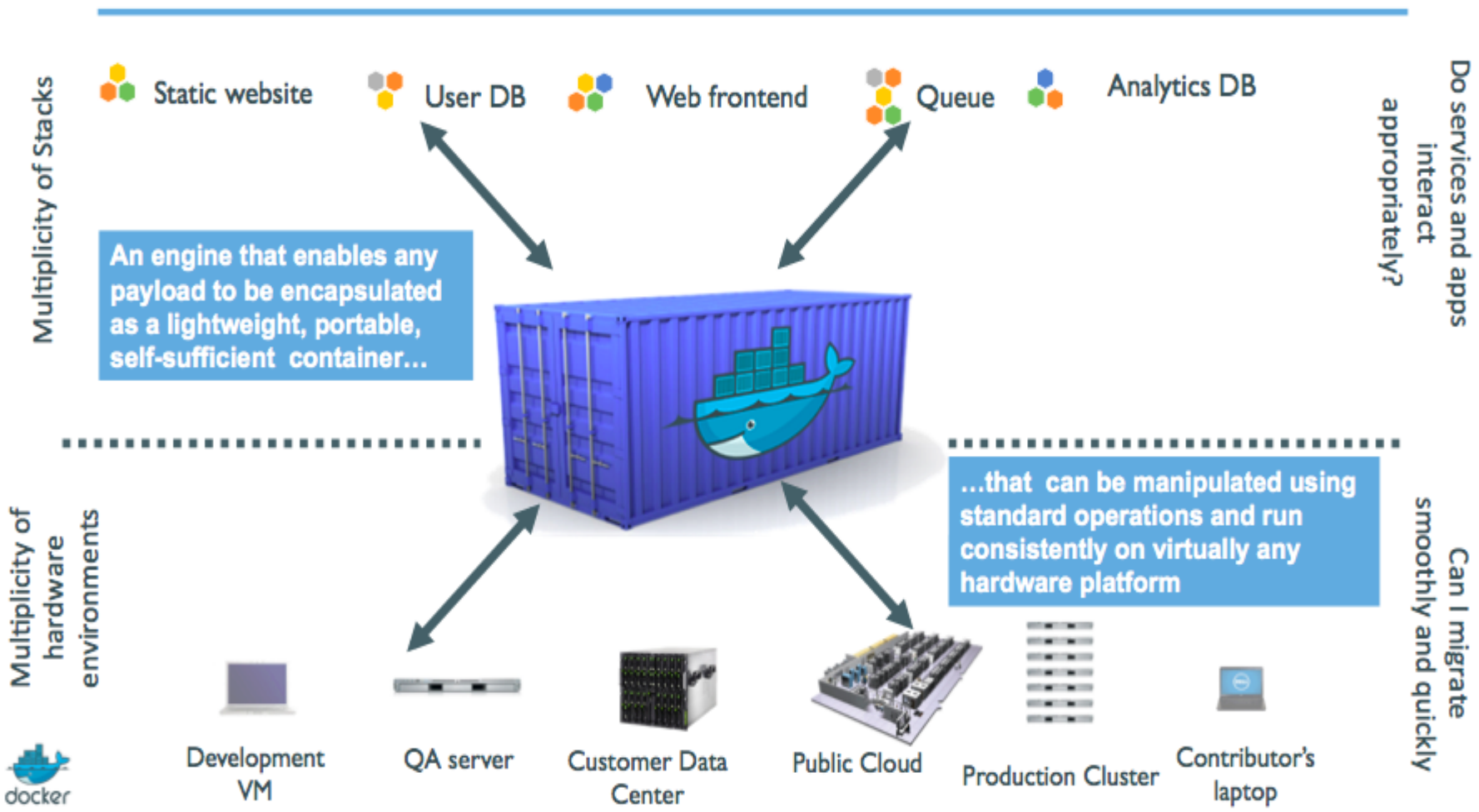
Atgen

Intermodal Shipping Containers



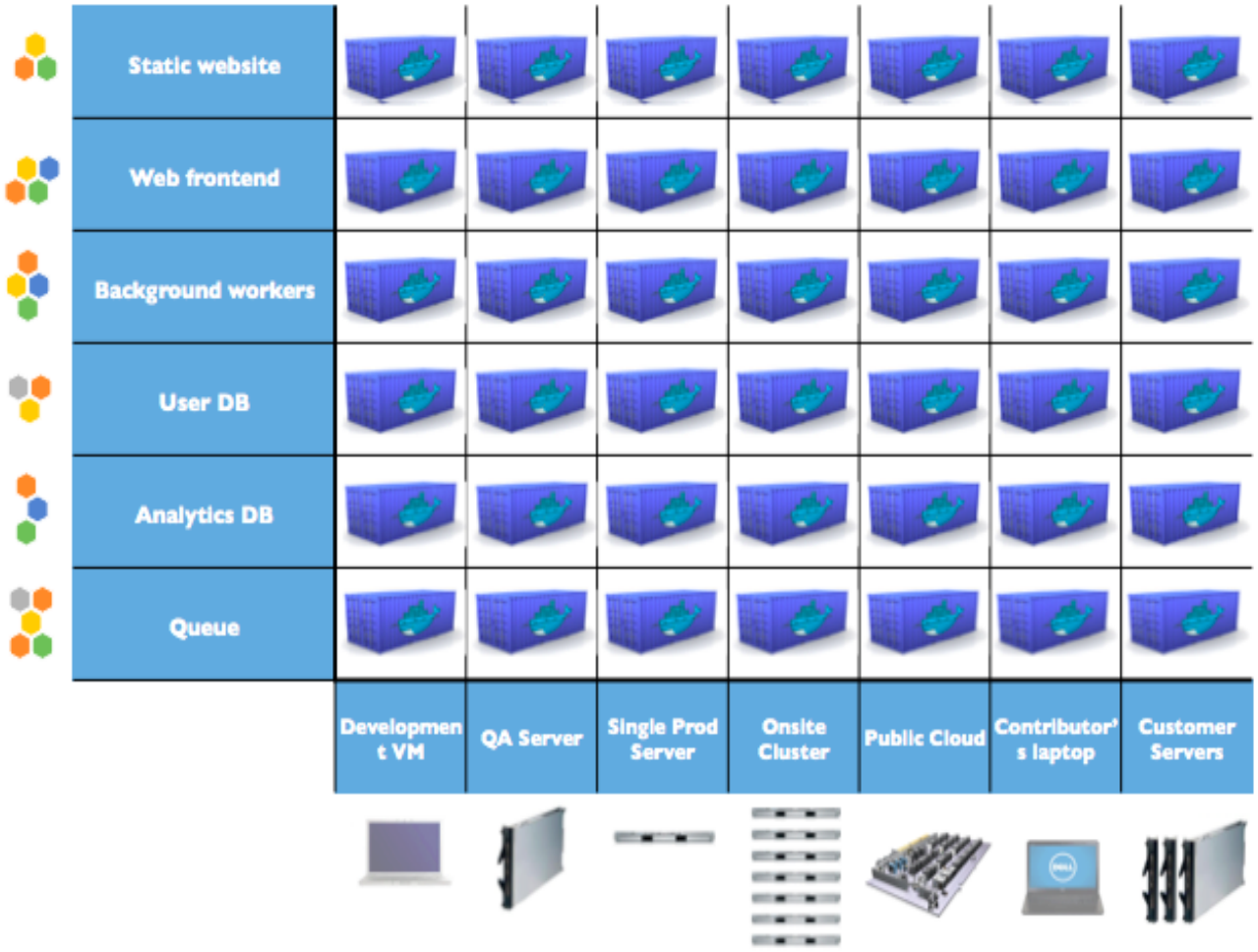
Atgen

Shipping Container for Applications



Atgen

Eliminate the Matrix



docker

Atgen

Results

- Dev-to-prod reduced time.
- Continuous integration job time reduced by more than 60%.
- Better application management.

Why containers(technical)

- Escape dependency hell
  - Write installation instructions into an "INSTALL.txt" file
  - Using this file, write an "install.sh" script that works for you
  - Turn this file into a "Dockerfile", test it on your machine
  - If the Dockerfile builds on your machine, it will build anywhere
  - Rejoice as you escape dependency hell and "works on my machine"

Never again "worked in dev – ops problem now!"

Why containers(technical)

- Implement reliable CI easily
  - Build test environment with a Dockerfile or Compose file
  - For each test run, stage up a new container or stack
  - Each run is now in a clean environment
  - No pollution from previous tests

Way faster and cheaper than creating VMs each time!

Why containers(technical)

- Use container images as build artifacts
  - Build your app from Dockerfiles
  - Store the resulting images in a registry
  - Keep them forever (or as long as necessary)
  - Test those images in QA, CI, integration...
  - Run the same images in production
  - Something goes wrong? Rollback to previous image
  - Investigating old regression? Old image has your back!

Images contain all the libraries, dependencies, etc. needed to run the app.

Atgen

Formats & APIs

- Before Docker:
  - No standardized exchange format.
  - Containers are hard to use for developers.
  - As a result, they are hidden from the end users.
  - No re-usable components, APIs, tools.

- After Docker:
  - Standardize the container format, because containers were not portable.
  - Make containers easy to use for developers.
  - Emphasis on re-usable components, APIs, ecosystem of standard tools.
  - Improvement over ad-hoc, in-house, specific tools.

Shipping

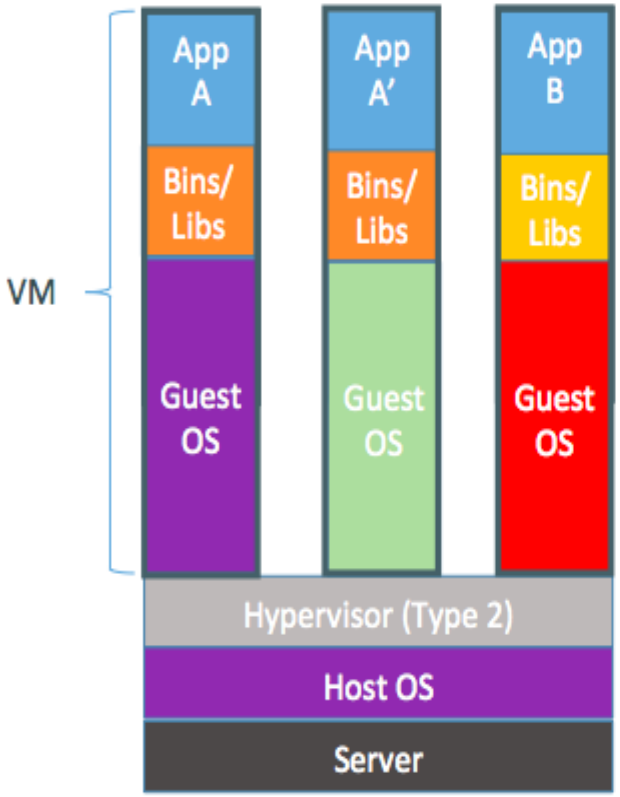
- Before Docker:
  - Ship packages: deb, rpm, gem, jar, homebrew...
  - Dependency hell.
  - Works on my machine?
  - Base deployment often done from scratch (debootstrap...) and unreliable.
- After Docker:
  - Ship container images with all their dependencies.
  - Images are bigger, but they are broken down into layers.
  - Only ship layers that have changed.
  - Save disk, network, memory usage.

DevOps

- Before Docker:
  - Drop a tarball (or a commit hash) with instructions.
  - Dev environment very different from production.
  - Ops don't always have a dev environment.
  - Ops have to sort out differences and make it work ...
  - Shipping code causes frictions and delays.

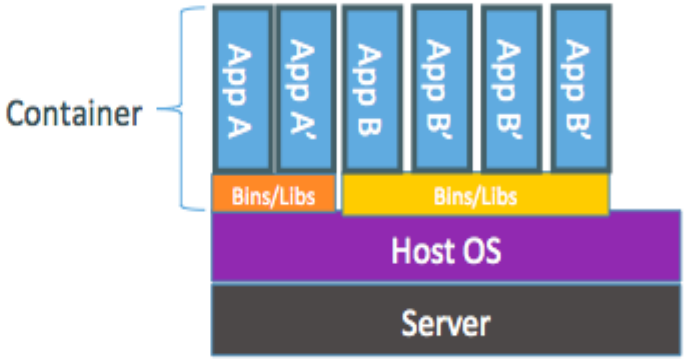
- After Docker:
  - Drop a container image or a Compose file.
  - Ops can always run that container image.
  - Ops can always run that Compose file.
  - Ops still have to adapt to prod environment, but at least they have a reference point.
  - Ops have tools allowing to use the same image in dev and prod.
  - Devs can be empowered to make releases themselves more easily.

Containers = cheaper than VMs



Containers are isolated,
but share OS kernel and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart



docker

Atgen

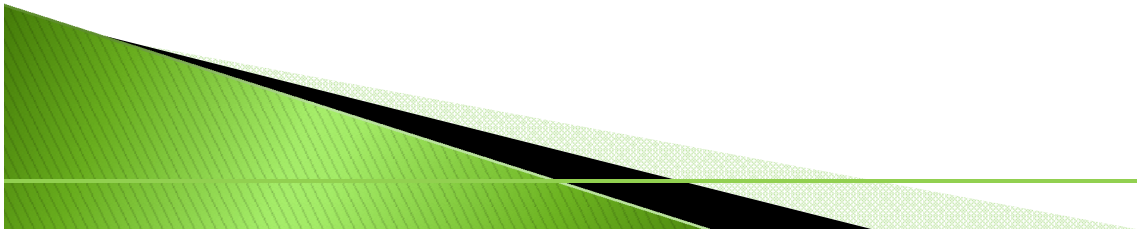
Containers = easier than VMs

Instantly Live
When you create or import an app with Heroku, it's already live on the web. No configuration or deployment necessary.

Create and Edit Online
Edit all of your code and data right in your browser. You can access it anywhere and there's nothing to install.

Share and Collaborate
Make your app public to the web or private to users you specify. Add collaborators who can edit the app with you.

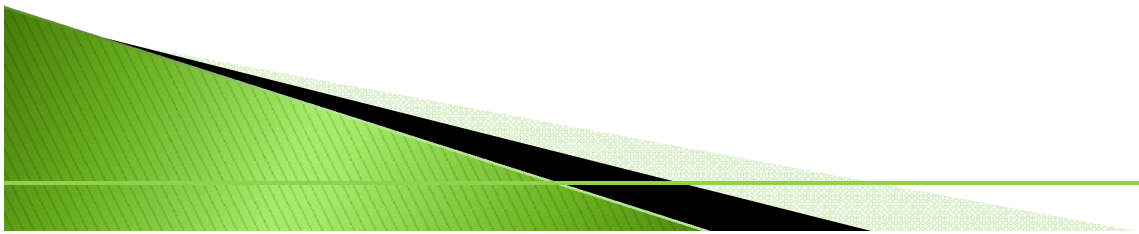
Import & Export
Easily import and export your app's code, schema, and data at any time with just one click.



Atgen

Session: 2

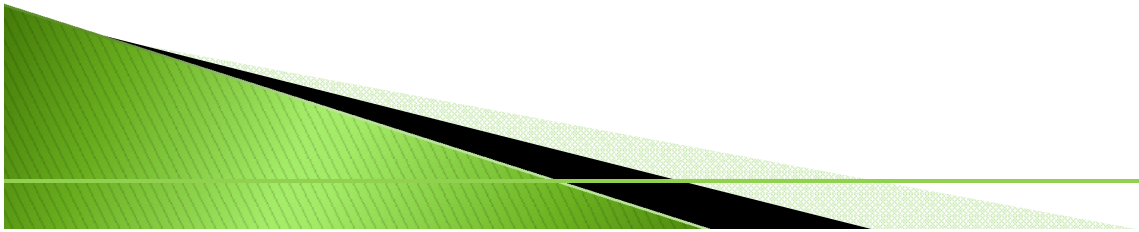
Docker Components



Atgen

Docker Overview

- Docker is an open platform for developing, shipping, and running applications.
- Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.
- With Docker, you can manage your infrastructure in the same ways you manage your applications.
- By using Docker's methodologies for shipping, testing, and deploying, you can reduce time of customer delivery.



Atgen

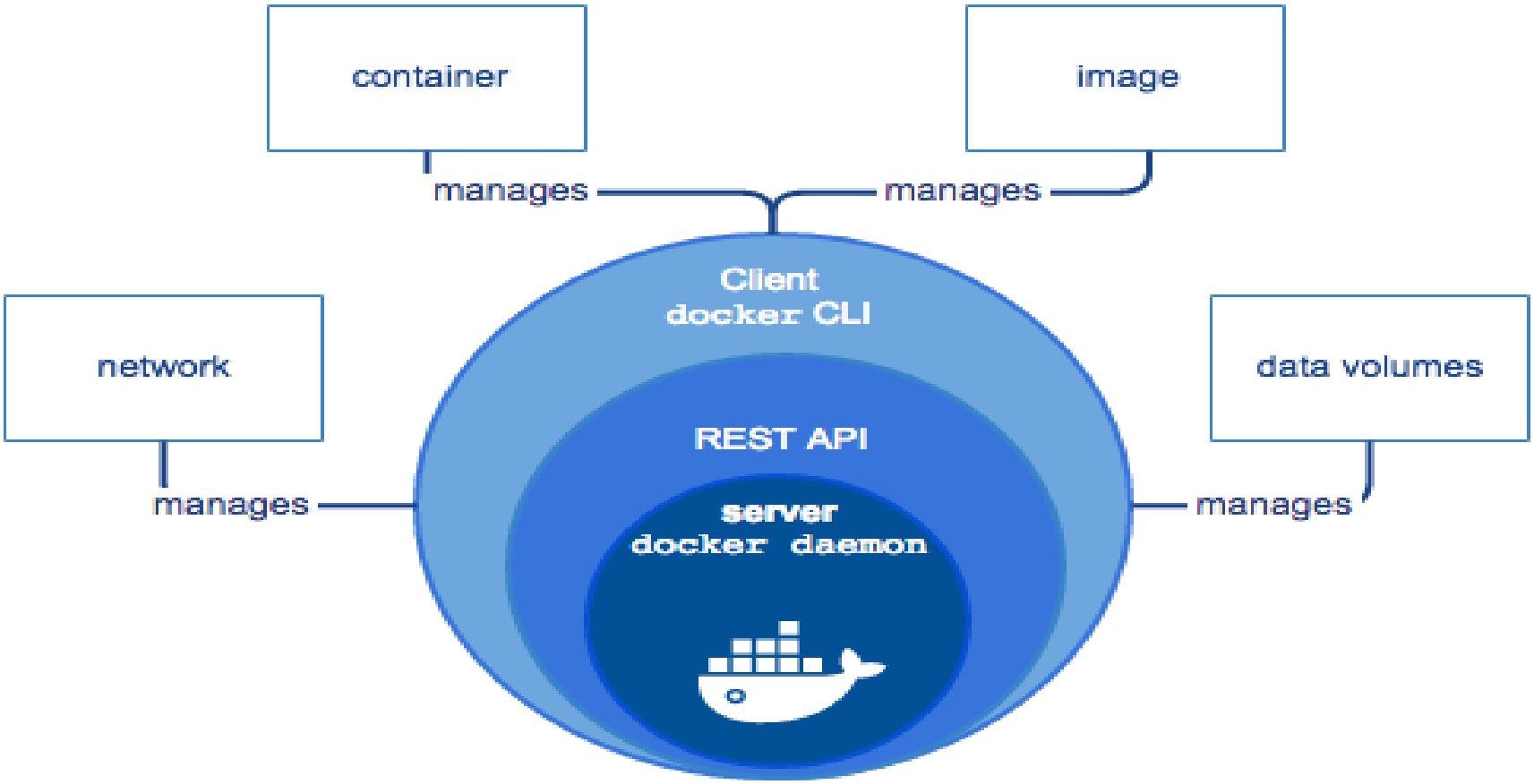
Docker Platform

- Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host.
- Because of the lightweight nature of containers, which run without the extra load of a hypervisor, you can run more containers on a given hardware combination than if you were using virtual machines.

Docker Platform

- Docker provides tooling and a platform to manage the lifecycle of your containers:
  - Encapsulate your applications (and supporting components) into Docker containers
  - Distribute and ship those containers to your teams for further development and testing
  - Deploy those applications to your production environment, whether it is in a local data center or the Cloud

Docker Engine

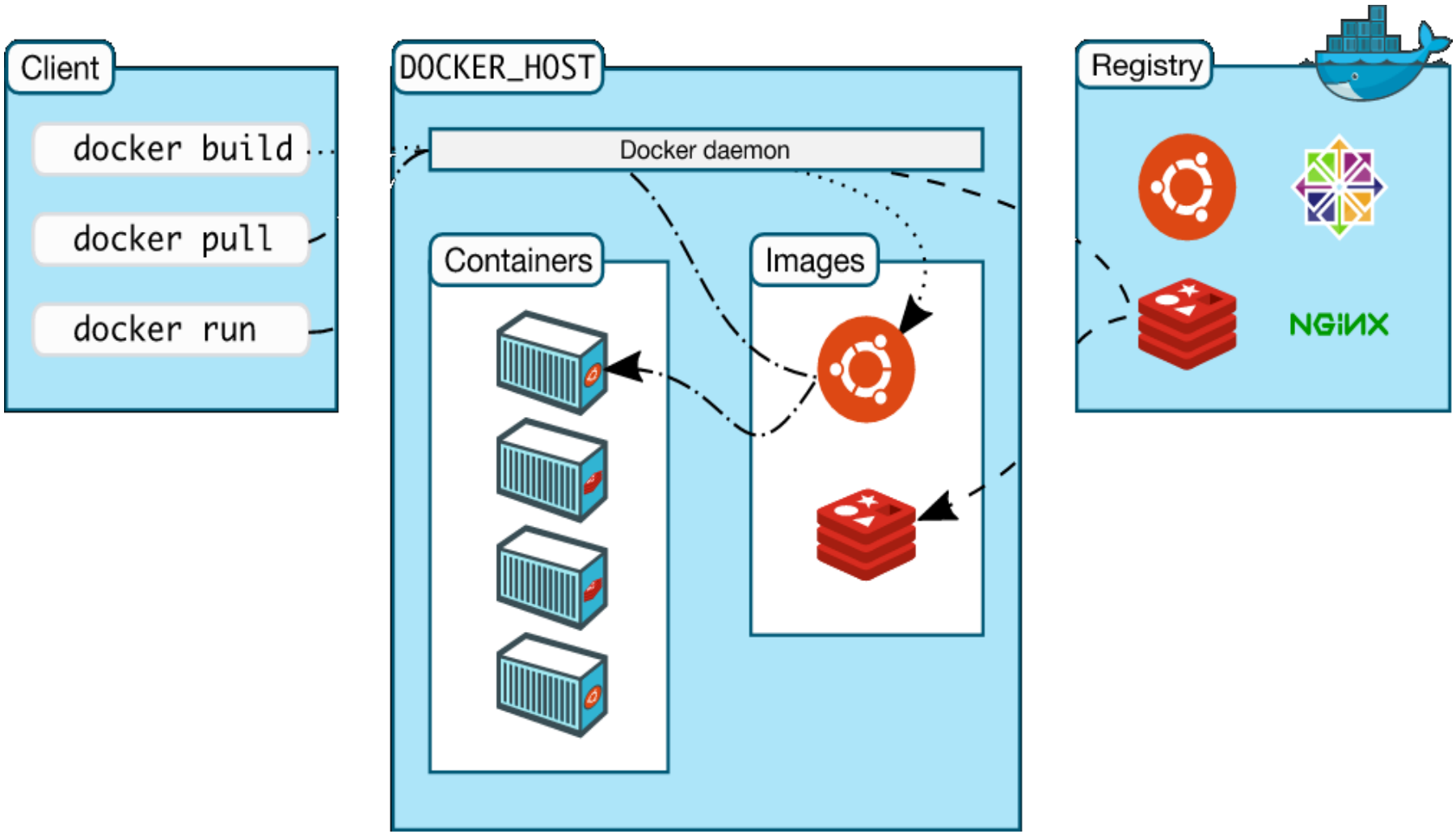


Atgen

Docker Engine

- Docker Engine is a client-server application with these major components:
  - A server which is a type of long-running program called a daemon process.
  - A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
  - A command line interface (CLI) client.
- The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands.
- The daemon creates and manages Docker objects, such as images, containers, networks, and data volumes.

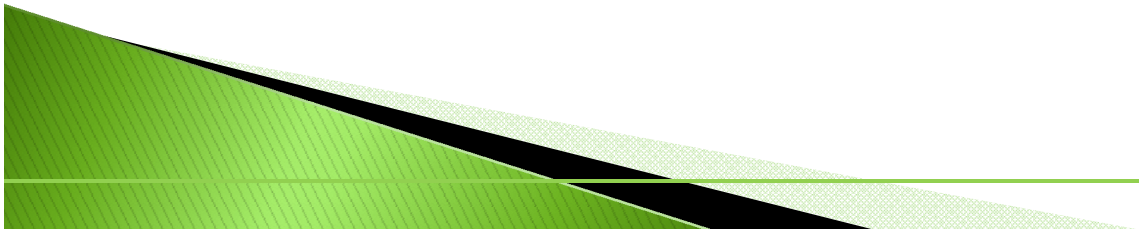
Docker Architecture



Atgen

Docker Architecture

- Docker uses a client-server architecture.
- The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

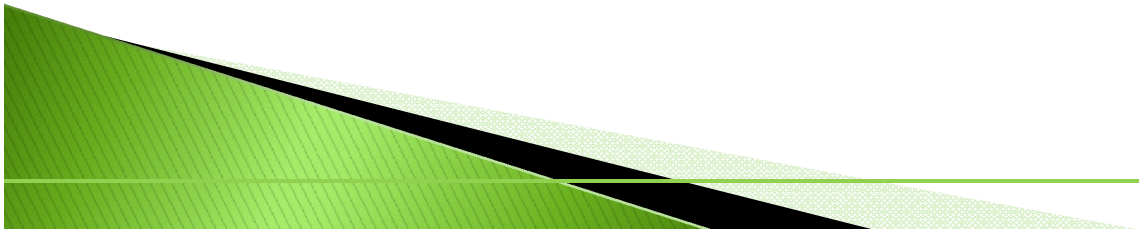


Atgen

Docker Architecture

- The Docker Daemon
  - The Docker daemon runs on a host machine. The user uses the Docker client to interact with the daemon.

- The Docker Client
  - The Docker client, in the form of the docker binary, is the primary user interface to Docker.
  - It accepts commands and configuration flags from the user and communicates with a Docker daemon.



Docker Images

- A Docker image is a read-only template with instructions for creating a Docker container.
- For example, an image might contain an Ubuntu operating system with Apache web server and your web application installed. You can build or update images from scratch or download and use images created by others.
- A docker image is described in text file called a Dockerfile, which has a simple, well-defined syntax.
- Docker images are the build component of Docker.

Docker Containers

- A Docker container is a running instance of a Docker image.
- You can run, start, stop, move, or delete a container using Docker API or CLI commands.
- When you run a container, you can provide configuration metadata such as networking information or environment variables.
- Each container is an isolated and secure application platform, but can be given access to resources running in a different host or container, as well as persistent storage or databases.
- Docker containers are the run component of Docker.

Docker Registries

- A docker registry is a library of images.
- A registry can be public or private, and can be on the same server as the Docker daemon or Docker client, or on a totally separate server.
- Docker registries are the distribution component of Docker.
- “Docker Hub” is known as global registry.

Docker Services

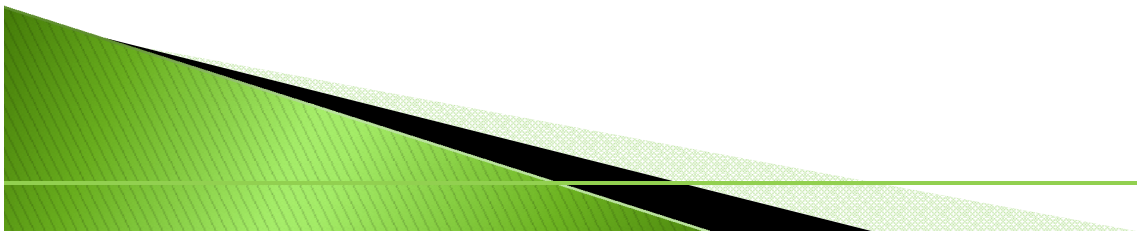
- A Docker service allows a swarm of Docker nodes to work together, running a defined number of instances of a replica task, which is itself a Docker image.
- You can specify the number of concurrent replica tasks to run, and the swarm manager ensures that the load is spread evenly across the worker nodes.
- To the consumer, the Docker service appears to be a single application.
- Docker services are the scalability component of Docker.

Docker Features

- Lightweight
  - Containers running on a single machine all share the same operating system kernel so they start instantly and make more efficient use of RAM. Images are constructed from layered filesystems so they can share common files, making disk usage and image downloads much more efficient.
- Open
  - Docker containers are based on open standards allowing containers to run on all major Linux distributions and Microsoft operating systems with support for every infrastructure.
- Secure
  - Containers isolate applications from each other and the underlying infrastructure while providing an added layer of protection for the application.

Session: 3

Classroom Environment



Atgen

Docker Engine Install Demo

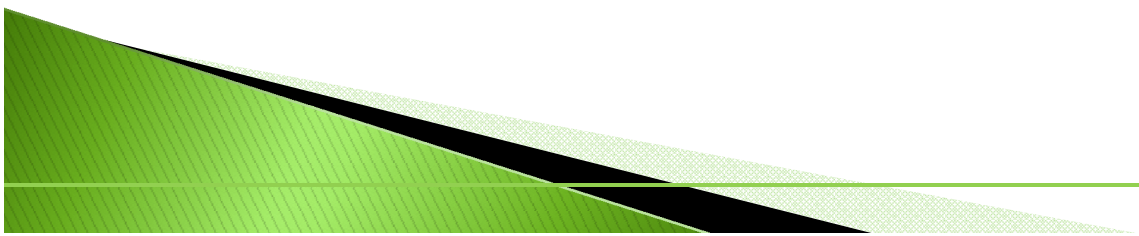
- Docker Engine/Client would be installed on Training Environment as demo LAB.

Manage Docker as a non-root user

- The docker daemon binds to a Unix socket instead of a TCP port.
- By default that Unix socket is owned by the user ”root” and other users can only access it using sudo.
- The docker daemon always runs as the root user.
- If you don’t want to use sudo when you use the docker command, add users to Unix group called ”docker”.
- When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

Session: 4

Containers



Atgen

How Container works

- A container uses the host machine's Linux kernel, and consists of any extra files you add when the image is created, along with metadata associated with the container at creation or when the container is started.
- Each container is built from an image.
- The image defines the container's contents, which process to run when the container is launched, and a variety of other configuration details.
- The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image (using a UnionFS) in which your application runs.

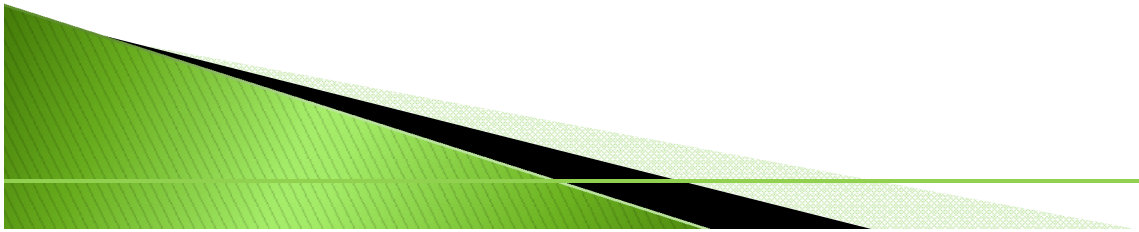
How Container works

- When you use the ”docker run” CLI command, the Docker Engine client instructs the Docker daemon to run a container.
- This example tells the Docker daemon to run a container using the ubuntu Docker image, to remain in the foreground in interactive mode (-i), and to run the /bin/bash command.

```
docker run -i -t ubuntu /bin/bash
```

How Container works

- When you run this command, Docker Engine does the following:
  - Pulls the ubuntu image
  - Creates a new container
  - Allocates a filesystem and mounts a read-write layer
  - Allocates a network / bridge interface
  - Sets up an IP address
  - Executes a process that you specify
  - Captures and provides application output



Atgen

Hello World – Container

- In your Docker environment, just run the following command:

```
docker run busybox echo hello world
```

- We used one of the smallest, simplest images available: busybox.
- We ran a single process and echo'ed hello world.

More Useful – Container

- In your Docker environment, just run the following command:

```
docker run -it ubuntu
```

- This is a brand new container.
- It runs a bare-bones, no-frills ubuntu system.
- -i tells Docker to connect us to the container's stdin.
- -t tells Docker that we want a pseudo-terminal.

More Useful – Container

- Do something in our container:

```
figlet hello
```

- Let's check how many packages are installed:

```
dpkg -l | wc -l
```

- Install a package in our container

```
apt-get update
apt-get install figlet
```

More Useful – Container

- Exiting our container:

```
exit
```

- Our container is now in a *stopped state.*
- It still exists on disk, but all compute resources have been freed up.

Starting Another – Container

- In your Docker environment, just run the following command:

```
docker run -it ubuntu
figlet hello
```

- We started a *brand new container.*
- The basic Ubuntu image was used, and figlet is not here.

A non interactive – Container

- In your Docker environment, just run the following command:

```
docker run jpetazzo/clock
```

- This container just displays the time every second.
- This container will run forever.
- To stop it, press ^C.
- Docker has automatically downloaded the image jpetazzo/clock.

Run in background – Container

- Containers can be started in the background, with the -d flag (daemon mode):

```
docker run -d jpetazzo/clock
```

- We don't see the output of the container.
- But don't worry: Docker collects that output and logs it!
- Docker gives us the ID of the container.

List Running Containers

- With docker ps, just like the UNIX ps command, lists running processes.

```
docker ps
```

- The (truncated) ID of our container.
- The image used to start the container.
- That our container has been running (Up) for a couple of minutes.
- Now, start multiple containers and use “docker ps” to list them.

List Running Containers - Flags

- To see only the last container that was started:

```
docker ps -l
```

- To see only the ID of containers:

```
docker ps -q
```

- We can combine the flags to see ID of last container started as well.

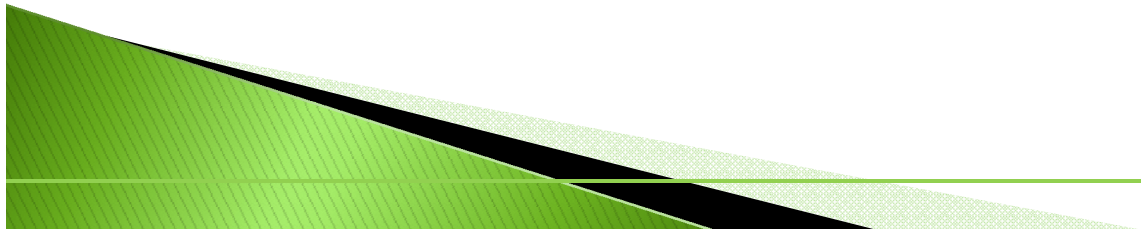
Logs of Container

- Logs of container can be seen using:

```
docker logs 068 [[where 068 is prefix of ID]]
```

- We specified a *prefix of the full container ID.*
- You can, of course, specify the full ID.
- The logs command will output the *entire logs of the container.*
- To avoid being spammed with pages of output, we can use the –tail option:

```
docker logs --tail 3 068
```

Atgen

Logs of Container

- We can also follow the logs real time using:

```
docker logs --tail 1 --follow 068
```

- This will display the last line in the log file.
- Then, it will continue to display the logs in real time.
- Use ^C to exit.

Stop our Container

- There are two ways we can terminate our detached container.
  - Killing it using the docker “kill” command.
  - Stopping it using the docker “stop” command.
- The first one stops the container immediately, by using the KILL signal.
- The second one is more graceful. It sends a TERM signal, and after 10 seconds, if the container has not stopped, it sends KILL.

Stop our Container

- Let's stop one of our containers:

```
docker stop 47d6
```

- This might take 10 seconds.
- Docker sends the TERM signal;
- If, the container doesn't react to this signal, 10 seconds later, since the container is still running, Docker sends the KILL signal; this terminates the container.