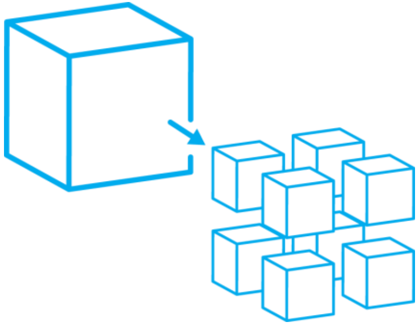
MICROSERVICE ARCHITECTURE WITH PYTHON & DOCKER



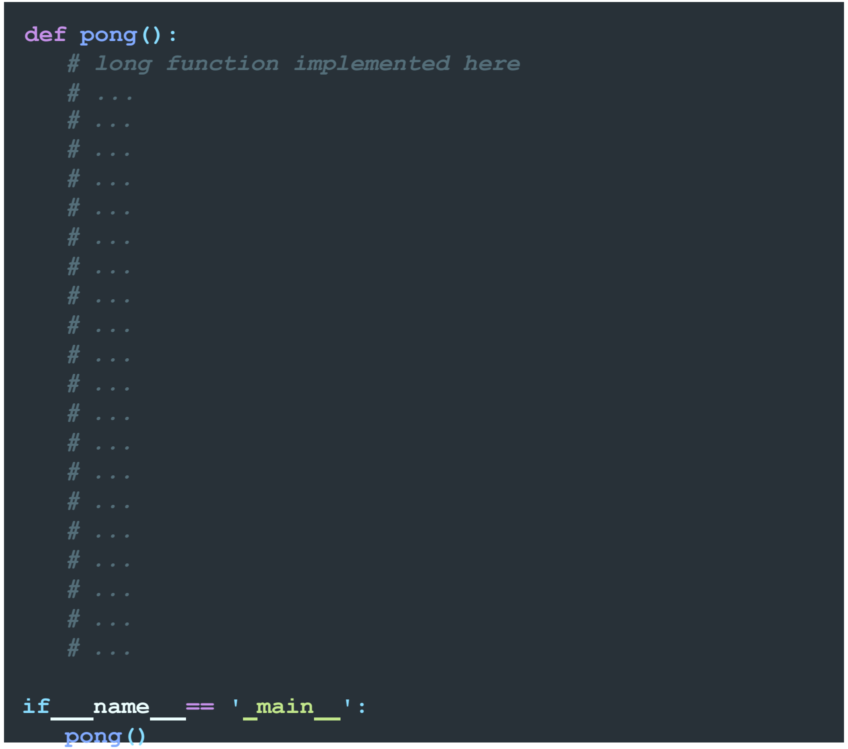
Meet Muhammad Zunair

- Technical Evangelist @ Systems Limited
  - Python and docker Lover
  - Focused on Cloud Computing and new development technologies.
  - Worked on Containerization of EAS
- Speaker
  - HEC & Microsoft Academic Initiative
  - Global Azure Bootcamp
  - Microtechx

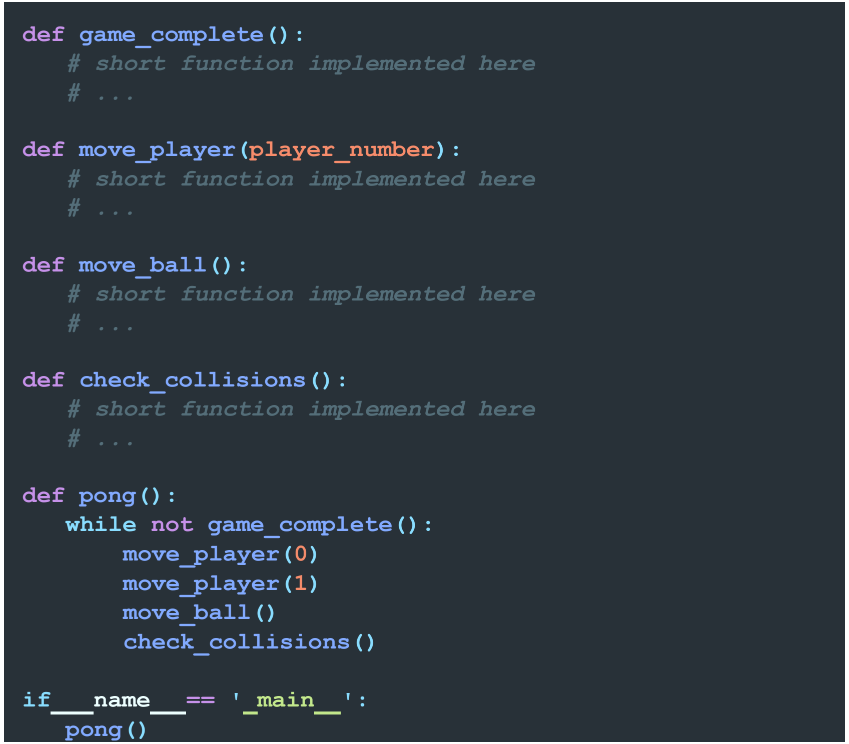
LinkedIn: https://www.linkedin.com/in/muhammadzunair/

GitHub: https://www.github.com/zunair-ch

Long Functions vs Short Functions

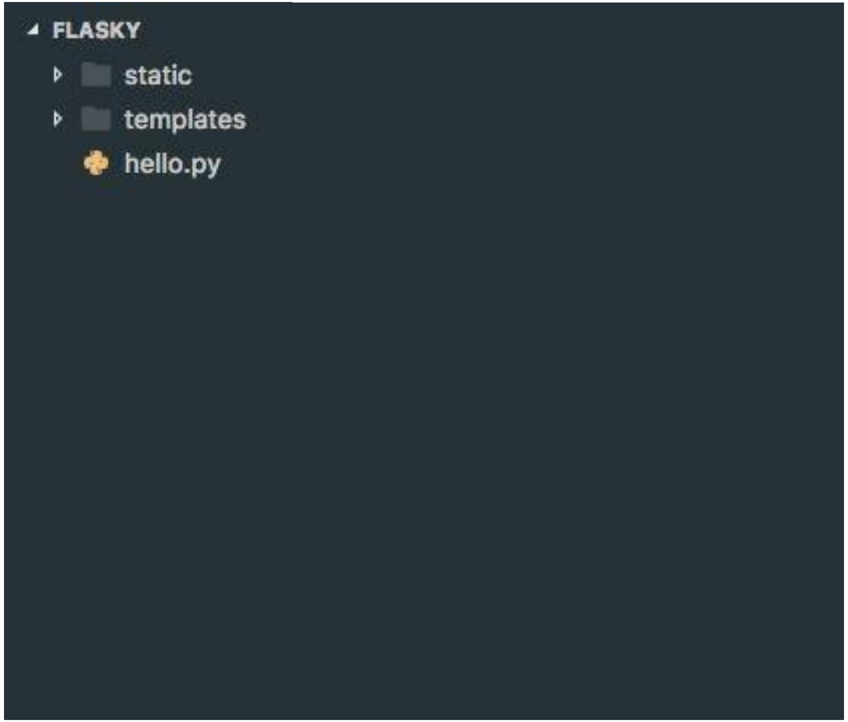


```
def pong():
    # long function implemented here
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...
    # ...

if __name__ == '_main__':
    pong()
```

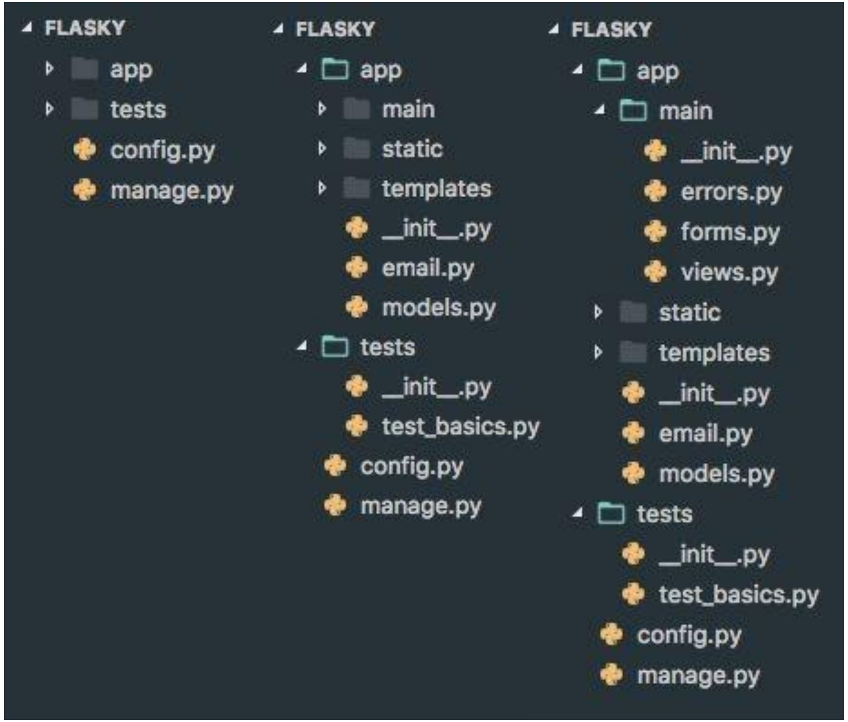


```
def game_complete():
    # short function implemented here
    # ...

def move_player(player_number):
    # short function implemented here
    # ...

def move_ball():
    # short function implemented here
    # ...

def check_collisions():
    # short function implemented here
    # ...

def pong():
    while not game_complete():
        move_player(0)
        move_player(1)
        move_ball()
        check_collisions()

if __name__ == '_main__':
    pong()
```

Long Modules vs Short Modules



```
▾ FLASKY
    ▸ static
    ▸ templates
      hello.py
```

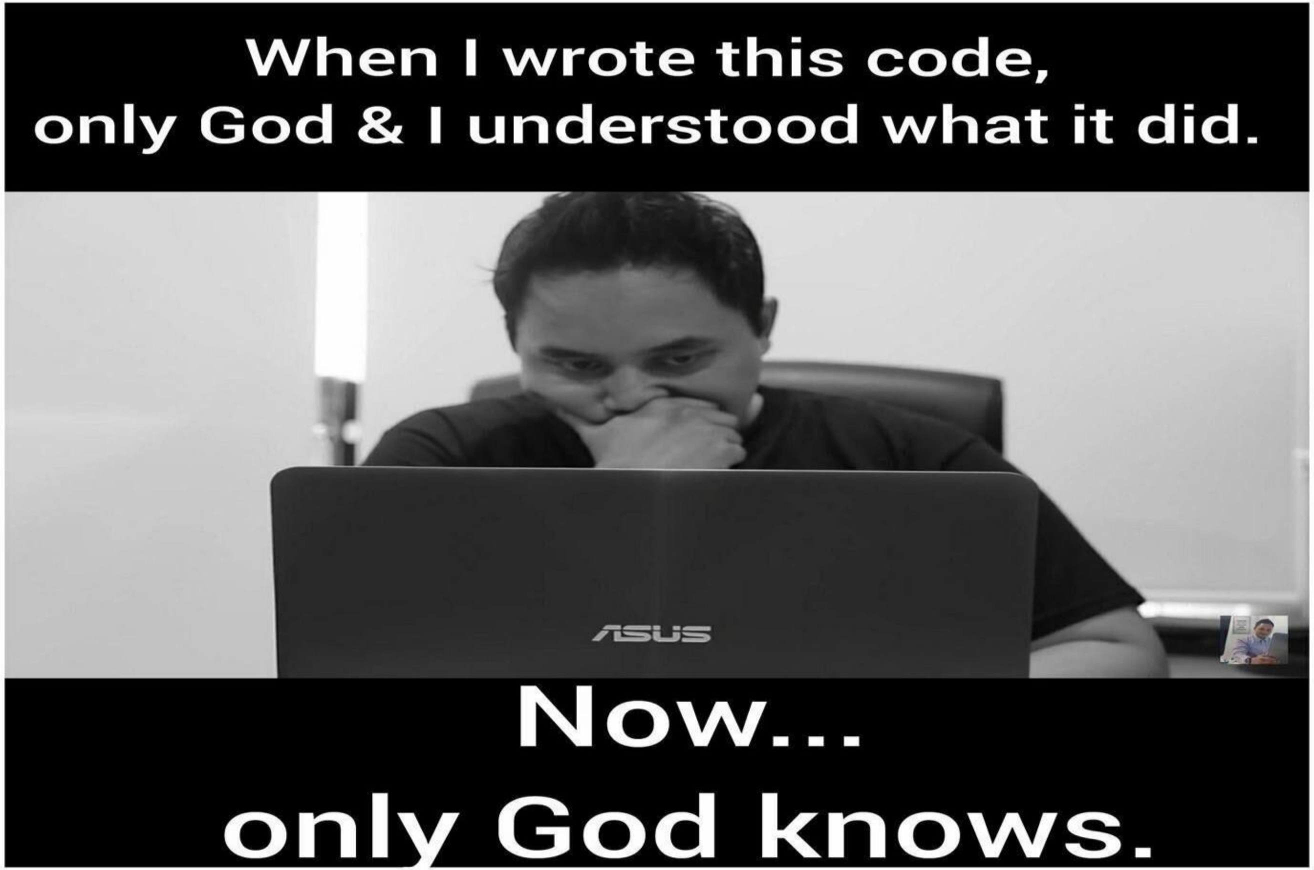


```
▾ FLASKY
    ▸ app
    ▸ tests
      config.py
      manage.py
```

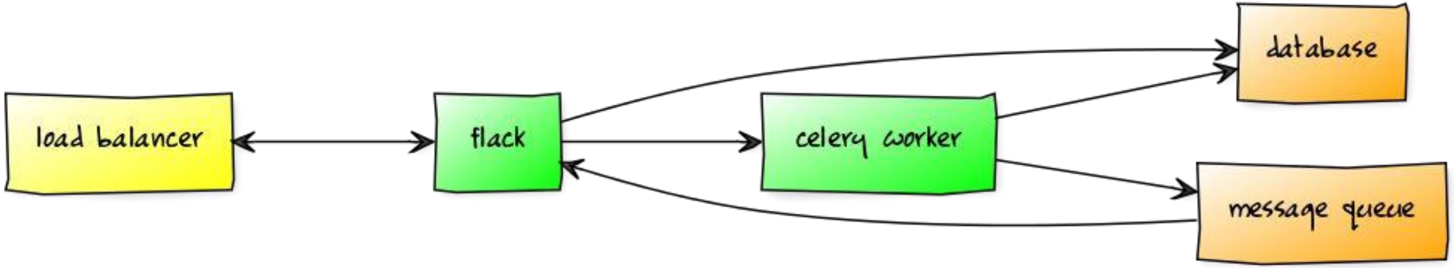
```
▾ FLASKY
    ▾ app
        ▸ main
        ▸ static
        ▸ templates
          __init__.py
          email.py
          models.py
    ▾ tests
          __init__.py
          test_basics.py
      config.py
      manage.py
```

```
▾ FLASKY
    ▾ app
        ▾ main
              __init__.py
              errors.py
              forms.py
              views.py
        ▸ static
        ▸ templates
          __init__.py
          email.py
          models.py
    ▾ tests
          __init__.py
          test_basics.py
      config.py
      manage.py
```

Why it’s important?



A Typical Monolithic Python Web Application



Find this app at https://github.com/zunair-ch/flack

The Problems with Monoliths

- Codebase becomes harder to maintain and test as it grows larger
- Coupling between modules causes random bugs when changes are made
- Steep learning curve for new team members
- Deployments and upgrades require downtime
- If the service crashes, your entire site goes down
- Inefficient **scaling**
- Difficult to incorporate to new technologies

Traditional Solution

- scale the application by running multiple instances of the monolith

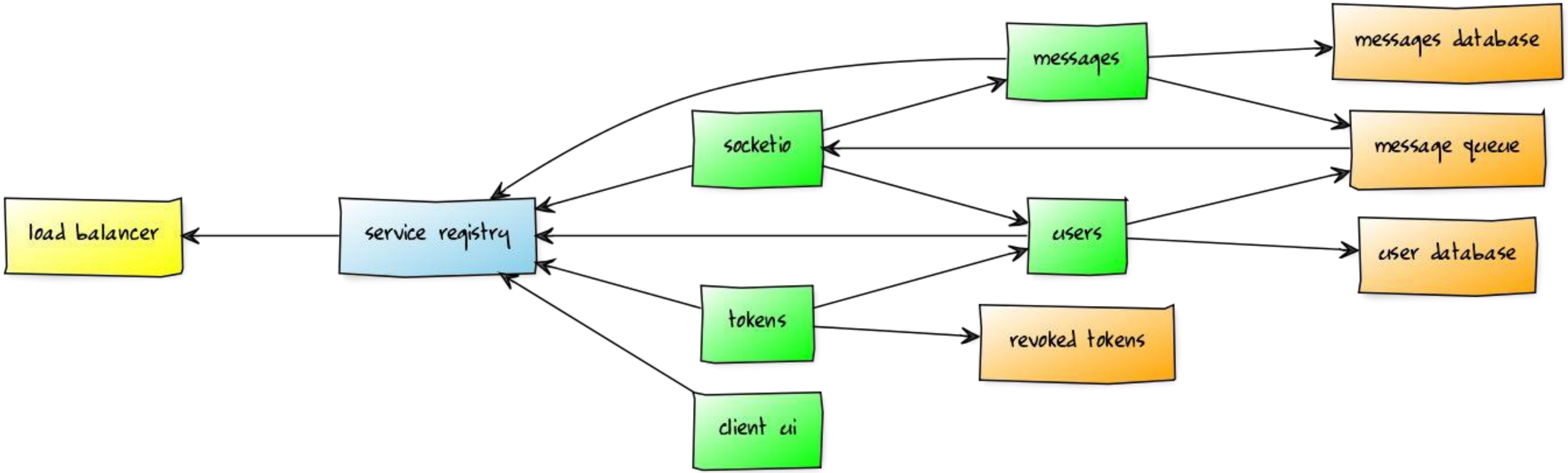
Netflix and Amazon address these problems with a solution called

Microservices

What are Microservices?

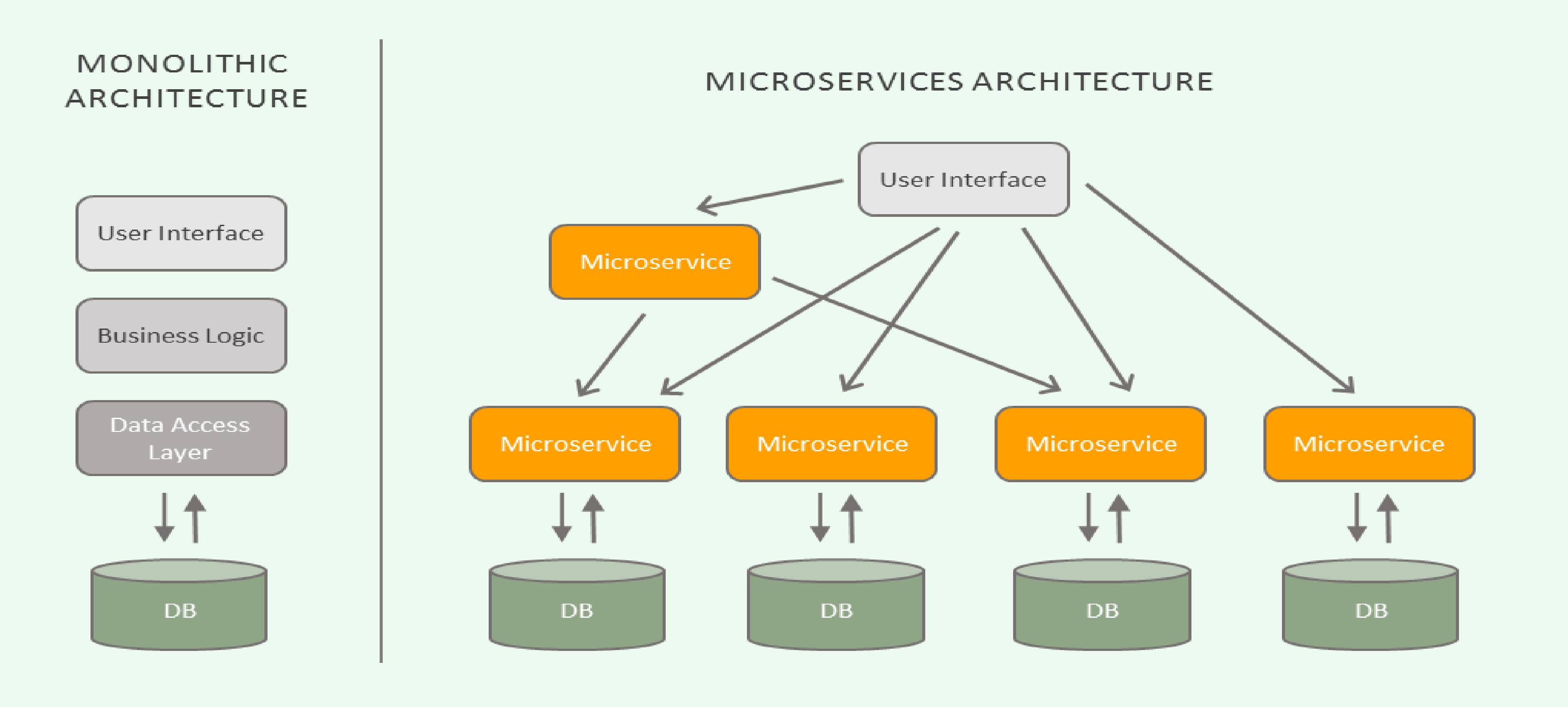
Microservice architecture is *an approach to develop a single application as a suite of small services.*

A Microservices Example



This app is also real! See https://github.com/zunair-ch/microflack_admin

Monolithic vs Microservices Architecture



Benefits of Microservices

- Code complexity greatly reduced
- Service separation promotes decoupled designs that have less bugs
- There is a lot less to learn to become productive
- Deployments don’t require downtime
- If a microservice crashes, the rest of the system keeps going
- Each microservice can be scaled individually according to its needs
- Services can use different tech stacks

Disadvantages of Microservices

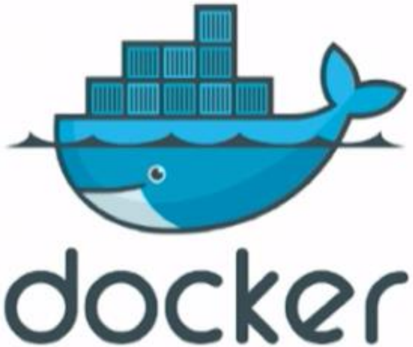
- The complexity moves from the code to the interactions between services
- Complex database joins must be implemented by the application
- Deployments have a lot of moving pieces
- Lower performance when a request “pinballs” through multiple microservices

Refactoring a Monolith into Microservices

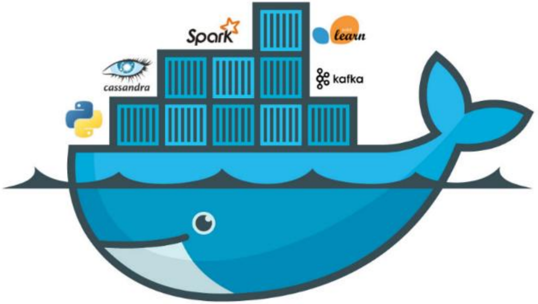
- Strategy #1: Microservices only going forward
- Strategy #2: Break pieces of functionality into microservices over time
- Strategy #3: Refactor the entire monolith into microservices
- In all cases, a base microservices platform needs to be put in place before refactoring work begins

What is Docker?

Docker is an open platform that helps companies build, ship and run their applications, anywhere.



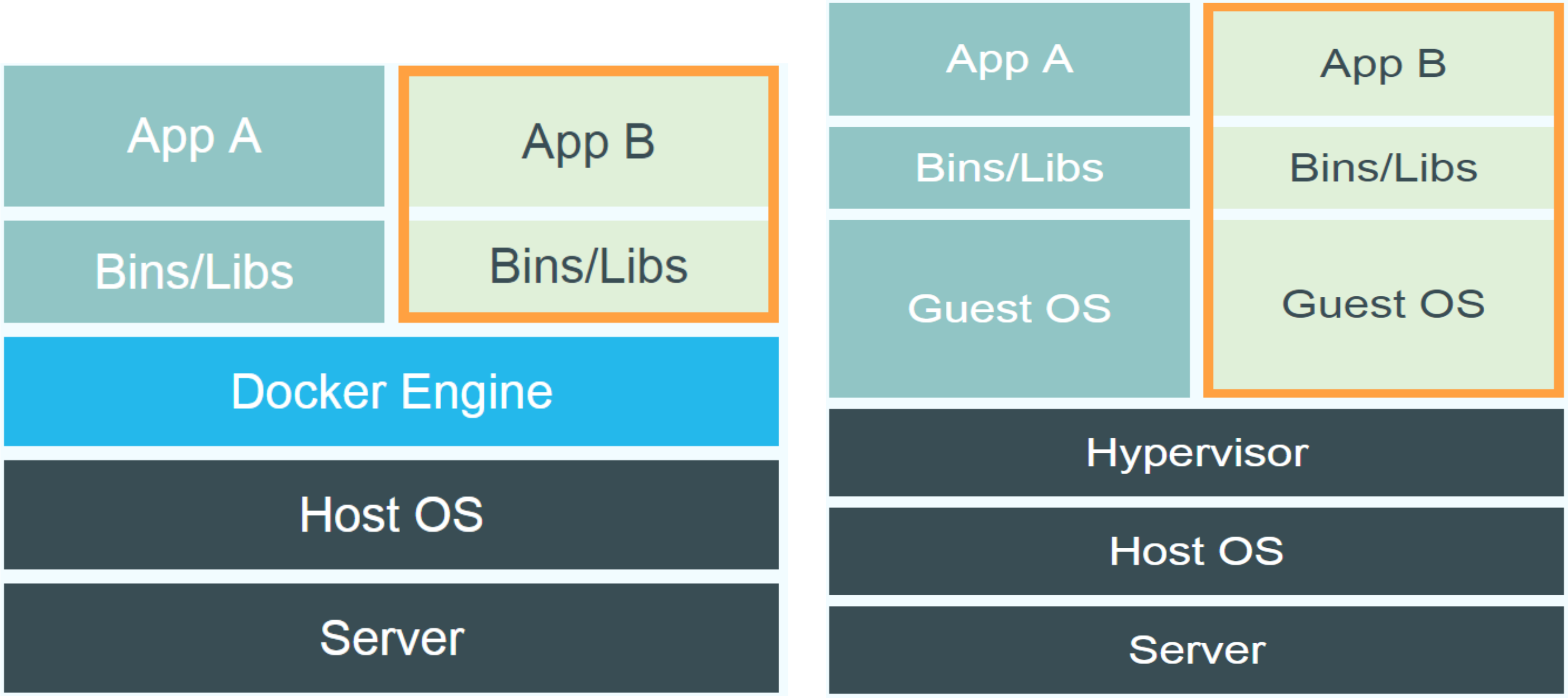
What is Docker?



A software technology company providing operating-system-level virtualization also known as **Containers**.

Promoted by company Docker Inc.

Docker Containers vs VM



App A

App B

Bins/Libs

Bins/Libs

Docker Engine

Host OS

Server

App A

App B

Bins/Libs

Bins/Libs

Guest OS

Guest OS

Hypervisor

Host OS

Server

VM vs Containers

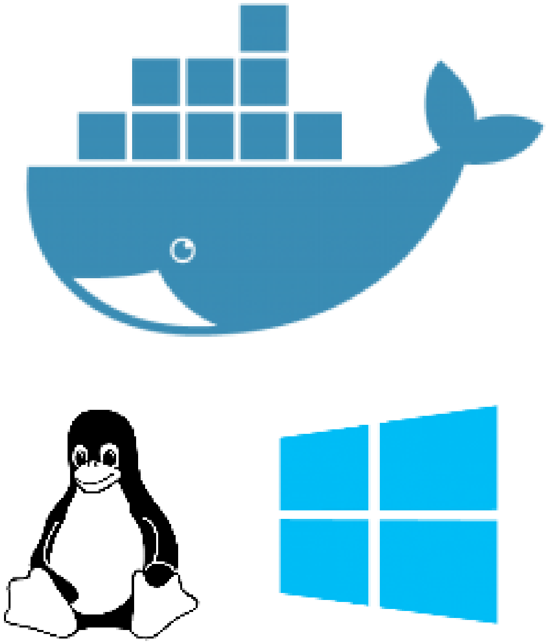
WORKED FINE IN DEV
OPS PROBLEM NOW
memegenerator.net

Docker Containers

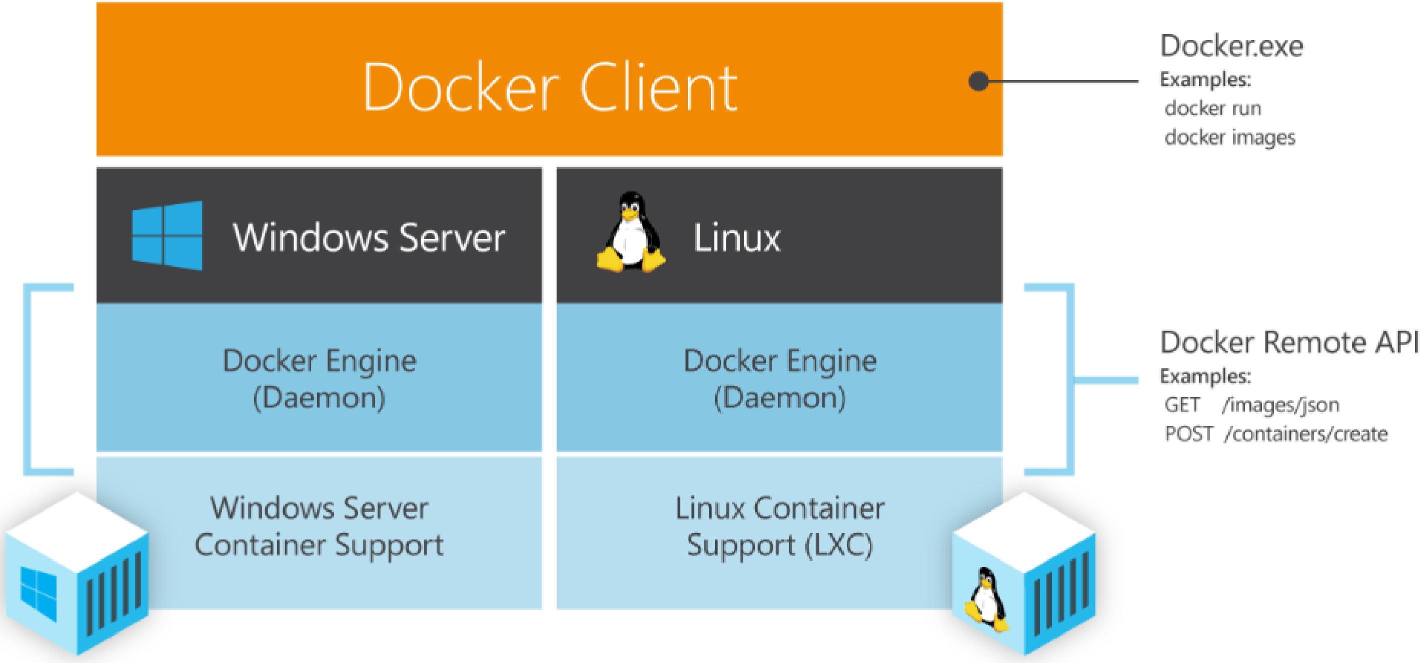
- Docker improves the deployment of applications with portable, self-sufficient containers, Linux or Windows, that can run on any cloud or on-premises.

No more:
"It works in my dev machine!... Why not in production?"

Now it is:
"If it works in Docker, it works in production"



Docker Engine for Linux and Windows



Docker Client

Windows Server

Linux

Docker Engine
(Daemon)

Docker Engine
(Daemon)

Windows Server
Container Support

Linux Container
Support (LXC)

Docker.exe
Examples:
docker run
docker images

Docker Remote API
Examples:
GET /images/json
POST /containers/create

Demo

The Microservices Platform

Load Balancer

- All microservices are accessed through the load balancer
- While microservices come and go, the load balancer is the “switchboard”
- Enables horizontal scaling of services
- Enables very cool tricks
  - Rolling upgrades
  - A/B testing
  - Green/Blue deployments
  - Etc.

Service Registry

- Datastore that keeps a list of running services
- Must be redundant, highly available, and fast
- Services make themselves known to the registry when they start
- They are removed (and possibly replaced) when they end or crash
- The load balancer is dynamically reconfigured when the registry changes

Containers

- Make services portable across host platforms
- Provide an additional layer of isolation over processes
- Allow each service to use its own dependencies
- Simplify managing of network ports

Storage

- Service registry, databases, message queues, etc. are stateful services
- It is important to make these services robust to prevent data loss
- Most storage solutions have clustering or mirroring options
  - MySQL → Galera, Aurora (AWS)
  - RabbitMQ → Native clustering and mirroring
  - Etc.

Application Microservices

- The microservices that you write are (ideally) stateless
- They can start and stop at any time, without data loss
- Horizontally scalable for free
- Python microservices can be written as simple web APIs using any framework
- Or you can use other mechanisms such as RPC to receive requests

Lifecycle of a Microservice

- On startup, the microservice registers with the service registry
- The load balancer detects the change in the registry and updates itself to include the new microservice
- The new service starts receiving traffic from the load balancer
- If more than one instance of the service exist, the traffic is split among them
- The service sends “keep-alive” signals, or responds to periodic health checks
- When the service is stopped, or stops sending keep-alives, or fails a health check, it is removed from the registry, and in turn from the load balancer

Service-to-Service Communication

- Outside clients connect over HTTP/REST (or maybe WebSocket)
- The service receiving the client request may need to invoke other services
- Services communicate with each other in a variety of ways
  - HTTP/REST
  - Job or message queues
  - RPC mechanisms
- Payloads exchanged between services should use well known formats
  - Pickle is not a good idea
  - JSON, msgpack, protobufs are all good

Try It Yourself!

Deploying MicroFlack to your Laptop

- Requirements
  - 4GB RAM (8GB recommended)
  - Vagrant
  - VirtualBox
  - Everything is installed in an Ubuntu 16.04 VM (Windows, Mac, Linux laptops are all OK!)
- Deployment commands:

```
git clone https://github.com/zunair-ch/microflack_admin  cd
microflack_admin
vagrant up                      # to create the VM or restart it after shutdown
vagrant ssh                     # to open a shell session on the VM
vagrant halt                    # to shutdown the VM (without destroying it)
vagrant snapshot save clean     # to save a snapshot with name “clean”
vagrant snapshot restore clean --no-provision  # to restore the snapshot
vagrant destroy                 # to delete the VM
```

☺ Thank You!