Code.Hub

The first Hub for Developers

Introduction to Docker

1. Install Docker
2. The Basics of How Docker Works
3. Understand Linux Containers
4. Manage Images, Containers, Networks & Volumes
5. Build Images & the Use of Dockerfile
6. Tag Images & Push them to the Docker Hub Registry

About the Lecture

Code.Hub

1. Admin Access to Install Docker
2. Linux Basic Knowledge
3. Web & Database Knowledge
4. Docker Hub Account

Prerequisites

Code.Hub

Introduction to Docker

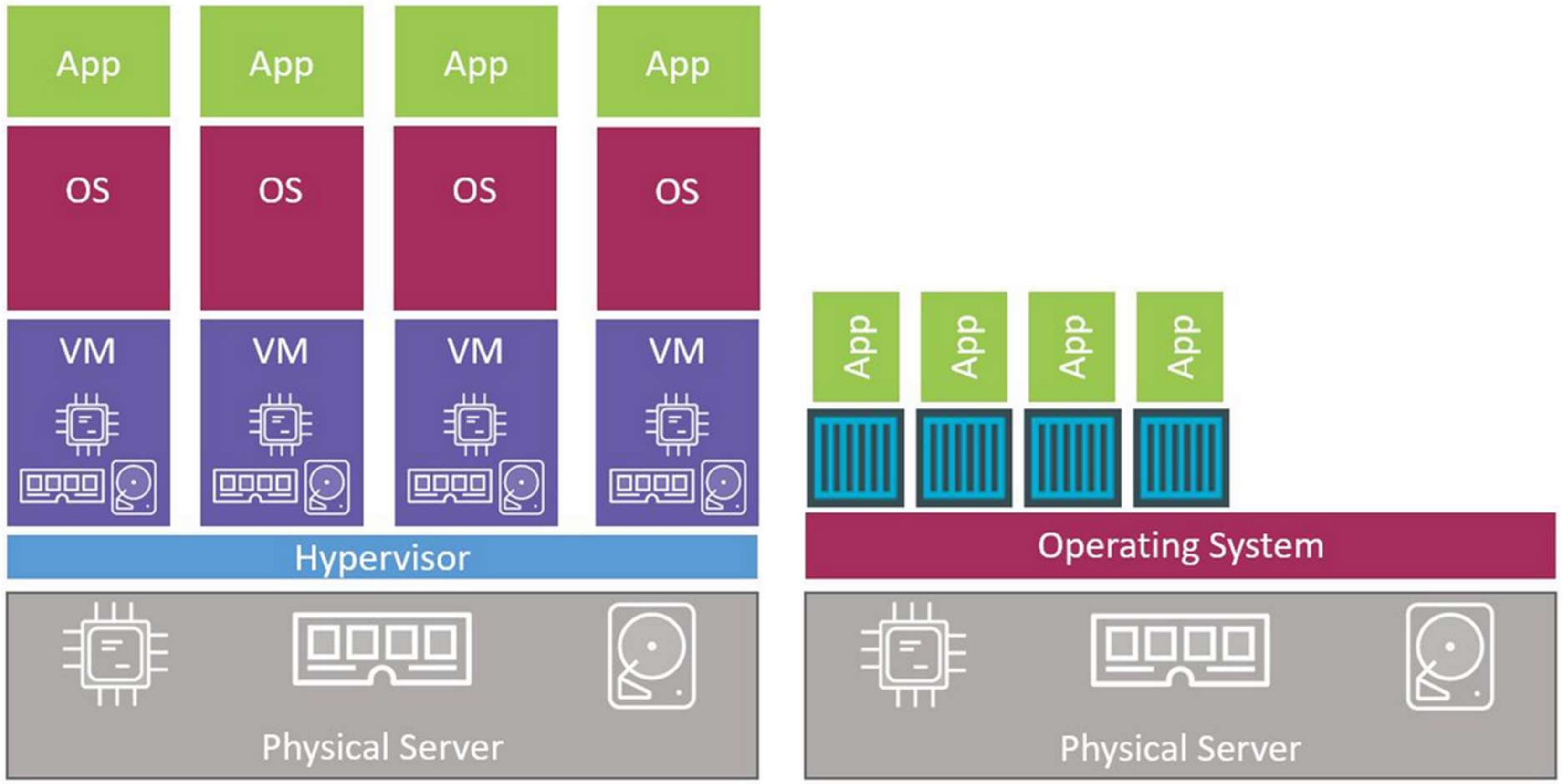
Code.Hub

What do you think Docker is?

Docker Concepts

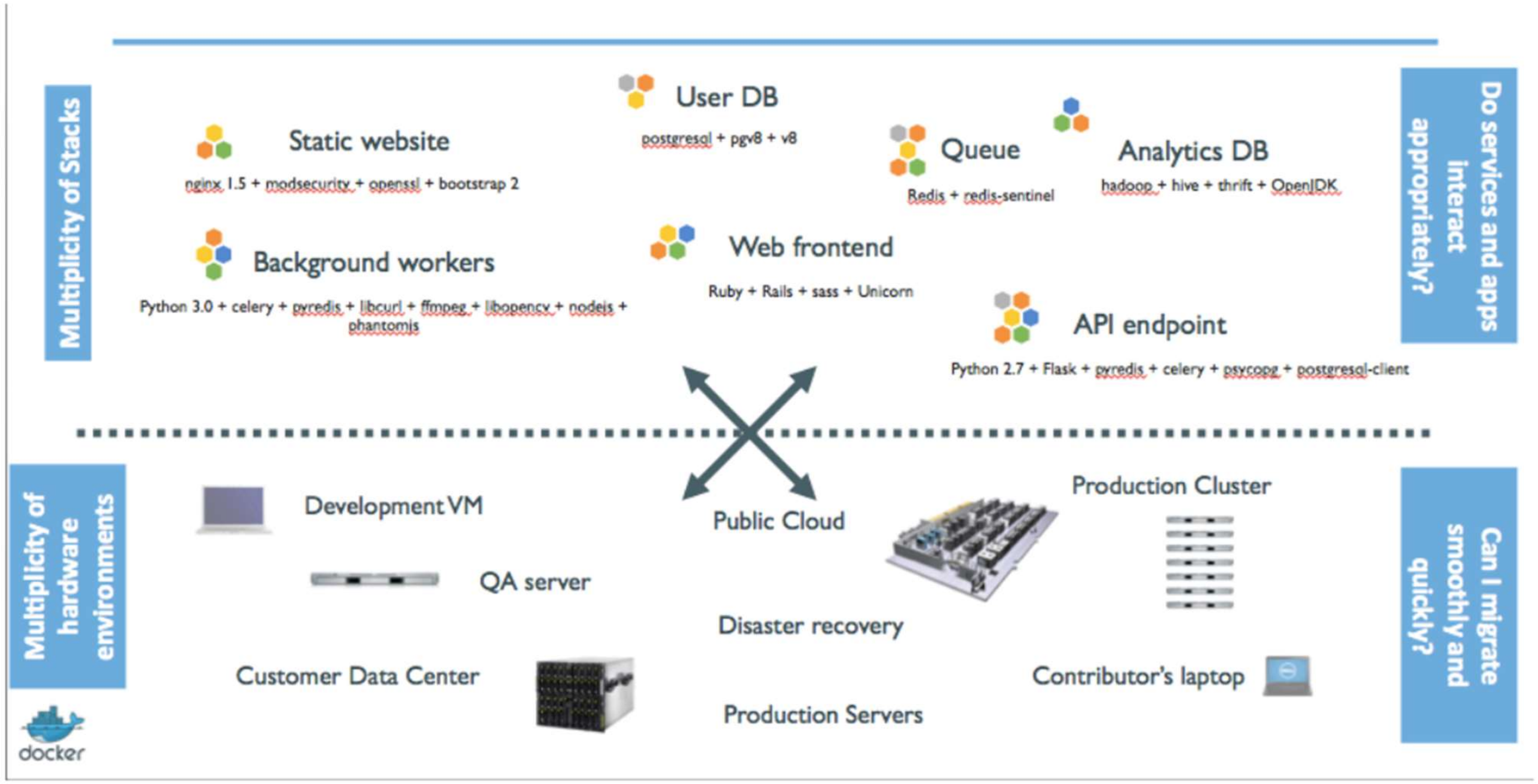
- Docker is a platform to develop, deploy and run applications with containers.
- The use of containers to deploy applications is called **containerization**.
  - Containers are not new, their use for **easily deploying applications** is.
- A container is created by running a Docker **image**.
- An image is an **executable package** that includes everything needed to run an application.
  - This includes: the code, a runtime, libraries, environment variables, configuration files etc.

Virtual Machines VS Docker Containers



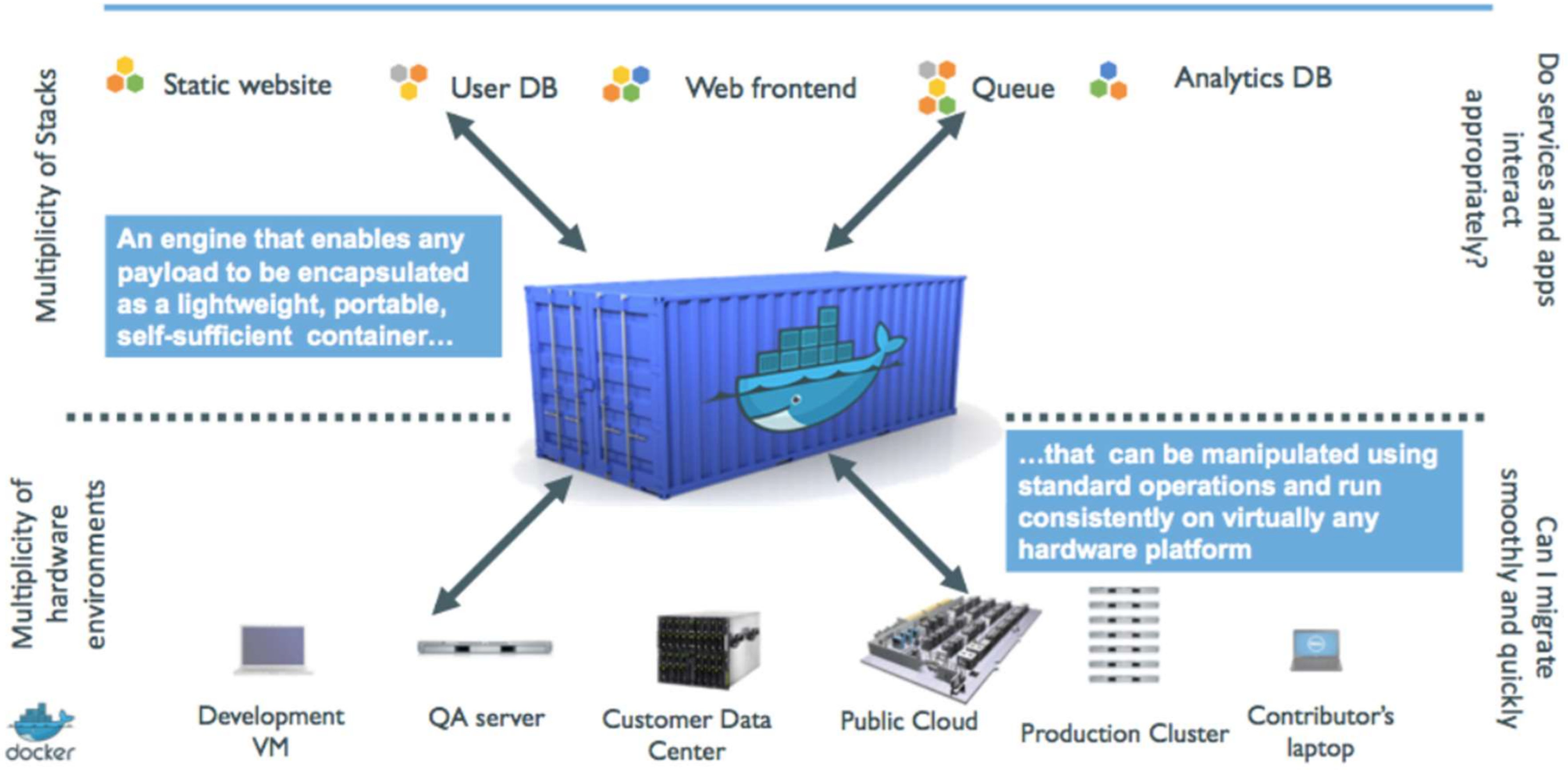
Code.Hub

The Deployment Problem


Multiplicity of Stacks
Static website
nginx 1.5 + modsecurity + openssl + bootstrap 2
User DB
postgresql + pgv8 + v8
Queue
Redis + redis-sentinel
Analytics DB
hadoop + hive + thrift + OpenJDK
Background workers
Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs
Web frontend
Ruby + Rails + sass + Unicorn
API endpoint
Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client
Do services and apps interact appropriately?
Multiplicity of hardware environments
Development VM
QA server
Public Cloud
Production Cluster
Disaster recovery
Customer Data Center
Contributor's laptop
Production Servers
Can I migrate smoothly and quickly?
docker

Code.Hub

The Deployment Solution


Multiplicity of Stacks
Static website
User DB
Web frontend
Queue
Analytics DB
Do services and apps interact appropriately?
An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…
…that can be manipulated using standard operations and run consistently on virtually any hardware platform
Multiplicity of hardware environments
Can I migrate smoothly and quickly
docker
Development VM
QA server
Customer Data Center
Public Cloud
Production Cluster
Contributor's laptop

Docker Engine Types

- Two types of Docker containers: **Linux Containers** & **Windows Containers**.
- Running a container shares the kernel of the host machine that it is running on.
  - Linux containers require a Linux host & Windows containers require a Windows host.
- Windows containers are very new. Microsoft announced native support for Windows containers in late **2016**.
- Can it work on MAC?

Installation of Docker

Code.Hub

Docker Installation

- Linux installation:

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

- Windows & Mac installation:

https://docs.docker.com/get-docker/

- Play with Docker:

https://labs.play-with-docker.com/

Docker Requirements

- VT-X enabled
  - Virtualization technology that should be enabled from the BIOS (if not enabled already)

- Minimum Hardware
  - CPU: i3
  - Ram: 4Gb
  - HD: 20Gb free space

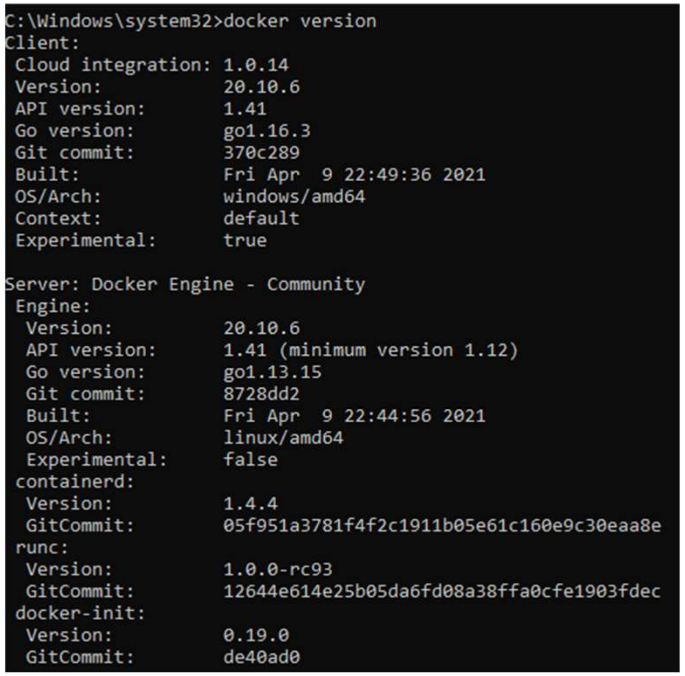
- Recommended Hardware
  - CPU: i5
  - Ram: 8Gb
  - HD: 20Gb free space

CLI/System Verification & First Docker Commands

Code.Hub

Verify Docker Installation

- To verify Docker installation and to display all version information, use the `docker version` command.



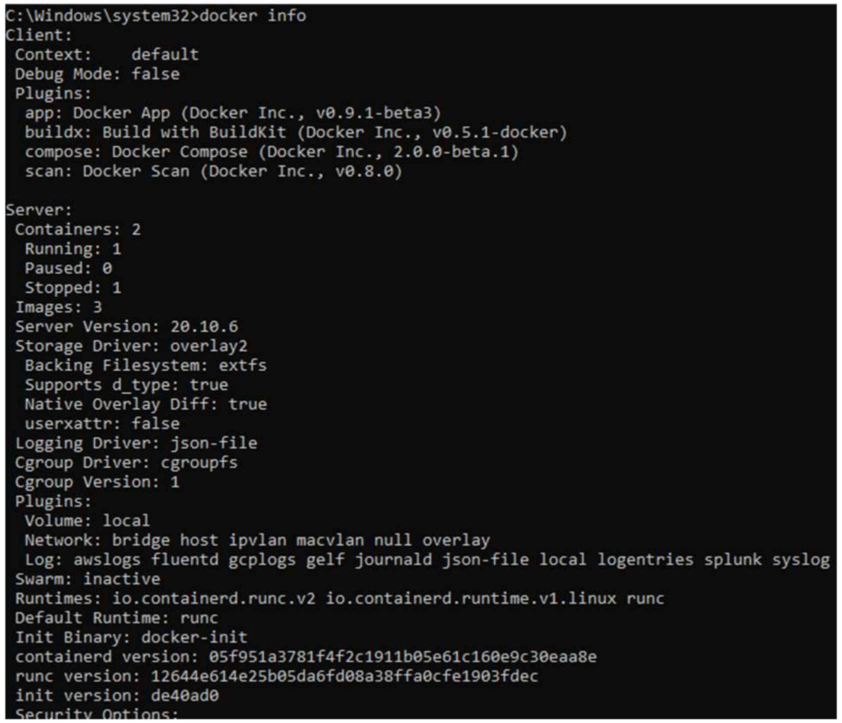
```
C:\Windows\system32>docker version
Client:
 Cloud integration: 1.0.14
 Version:           20.10.6
 API version:       1.41
 Go version:        go1.16.3
 Git commit:        370c289
 Built:             Fri Apr  9 22:49:36 2021
 OS/Arch:           windows/amd64
 Context:           default
 Experimental:      true

Server: Docker Engine - Community
 Engine:
  Version:          20.10.6
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.13.15
  Git commit:       8728dd2
  Built:            Fri Apr  9 22:44:56 2021
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.4.4
  GitCommit:        05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc:
  Version:          1.0.0-rc93
  GitCommit:        12644e614e25b05da6fd08a38ffa0cfe1903fdec
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

docker version - Notes

- By using the command, we can see that there is a **client** and a **server** version.
- Ideally, the server and the client versions should be the same.
- The Docker server can also be referred as **Docker Engine** or **Docker Daemon**.
- Getting information from the server validates the Docker installation as we can properly communicate with it.

docker info

- Use the `docker info` command to display system-wide information.



```
C:\Windows\system32>docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  app: Docker App (Docker Inc., v0.9.1-beta3)
  buildx: Build with BuildKit (Docker Inc., v0.5.1-docker)
  compose: Docker Compose (Docker Inc., 2.0.0-beta.1)
  scan: Docker Scan (Docker Inc., v0.8.0)

Server:
 Containers: 2
  Running: 1
  Paused: 0
  Stopped: 1
 Images: 3
 Server Version: 20.10.6
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: 05f951a3781f4f2c1911b05e61c160e9c30eaa8e
 runc version: 12644e614e25b05da6fd08a38ffa0cfe1903fdec
 init version: de40ad0
 Security Options:
```

docker info - Notes

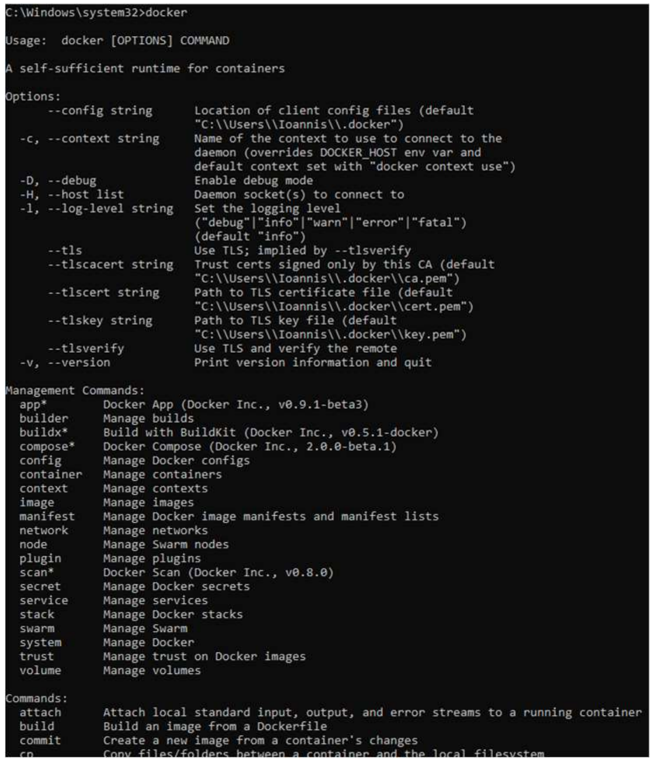
- The command shows us some useful information:
  - Number of containers (Running, Paused & Stopped).
  - Number of images stored.
  - The Swarm state (active/inactive).

Docker Swarm, is a native container orchestration and clustering tool provided by Docker. Docker Swarm allows you to create and manage a cluster of Docker nodes, which can be used to deploy and manage containerized applications at scale. When you refer to the "Swarm state" as "active/inactive," you are likely talking about the status of a Docker Swarm.

Complete list of Docker commands

- To get the complete list of Docker commands, simply use the `docker` command.

- The docker command has 3 main sections:
  - Options
  - Management Commands
  - Commands



```
C:\Windows\system32>docker

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default
                           "C:\\Users\\Ioannis\\.docker")
  -c, --context string     Name of the context to use to connect to the
                           daemon (overrides DOCKER_HOST env var and
                           default context set with "docker context use")
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level
                           ("debug"|"info"|"warn"|"error"|"fatal")
                           (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default
                           "C:\\Users\\Ioannis\\.docker\\ca.pem")
      --tlscert string     Path to TLS certificate file (default
                           "C:\\Users\\Ioannis\\.docker\\cert.pem")
      --tlskey string      Path to TLS key file (default
                           "C:\\Users\\Ioannis\\.docker\\key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit

Management Commands:
  app*        Docker App (Docker Inc., v0.9.1-beta3)
  builder     Manage builds
  buildx*     Build with BuildKit (Docker Inc., v0.5.1-docker)
  compose*    Docker Compose (Docker Inc., 2.0.0-beta.1)
  config      Manage Docker configs
  container   Manage containers
  context     Manage contexts
  image       Manage images
  manifest    Manage Docker image manifests and manifest lists
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  scan*       Docker Scan (Docker Inc., v0.8.0)
  secret      Manage Docker secrets
  service     Manage services
  stack       Manage Docker stacks
  swarm       Manage Swarm
  system      Manage Docker
  trust       Manage trust on Docker images
  volume      Manage volumes

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
```

Docker Command Format

- New format:

```
docker <mng-command> <command> [options]
```

- Old format (still in use):

```
docker <command> [options]
```

Docker Command Format - Examples

- New format:

```
docker container run
Docker container ps
```

- Old format (still in use):

```
docker run
docker ps
```

Note: Docker is focused on backwards compatibility. Therefore, it is safe to say that the `docker run` command will work forever; but new commands will use the new format.

Docker Commands Summary

```
docker version
docker info
docker <Enter> => CLI documentation
docker <mng-command> <command>
docker container run
docker run
```

Exercises/Questions

- Verify the “Swarm” status.
- Explain the difference between `docker info` and `docker system info`.

Basic Docker Terminology

Code.Hub

Containers

- A container is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings(e.g. configuration/Env Variables).

- Containers are isolated from the host system and other containers, making them portable and consistent across different environments.

Images

- An image is a **read-only template** or blueprint **used to create containers**. It includes the application code, runtime, system tools, libraries, and environment variables.

- Images are **immutable**, meaning they cannot be modified once created. To make changes, you create a new image based on an existing one.

Layers

- Images are composed of multiple layers. Each layer represents a set of file system changes. **Layers can be shared among multiple images**, which helps save disk space and speeds up image creation.

- When you make changes to an image (e.g., installing software or modifying code), **Docker only needs to create a new layer for those changes**. Existing layers are reused.

- Layering is a key feature of Docker's efficiency and **helps reduce the size of image updates.**

Let’s put it all together

Images and Containers Relationship:

- An image is a blueprint for creating containers. You can think of it as a snapshot of a container's file system at a specific point in time.
- When you run an image, Docker creates a container from it. **The container is an instance of the image, and it is a runnable environment that can execute your application.**

Layers and Images Relationship:

- Each image is composed of multiple layers. These layers are stacked on top of each other, forming a hierarchy.
- The layers are read-only and can be shared between different images. F**or example, if you have two images that both require the same base operating system, they can share the common base layer**, reducing storage space and download time.
- When you create a new image with changes, **Docker only needs to add new layers for those changes**. This means that the existing layers are not duplicated, which saves space and time.

Containers and Layers Relationship:

- Containers are instances of images. When you run a container, it is created from an image, and it can read and write data to a writable layer on top of the image's read-only layers.
- This writable layer is separate for each container, providing isolation. Any changes made to the container are stored in this writable layer, leaving the underlying image layers unchanged.

Docker and Microservices

Code.Hub

Containers and Microservices

- Containers encapsulate applications and their dependencies.
- Benefits: consistency, portability, and isolation.
- Containers are lightweight, which makes them suitable for microservices.

Live Demo: Running a simple container

Step 1: Pull the "Hello World" Image

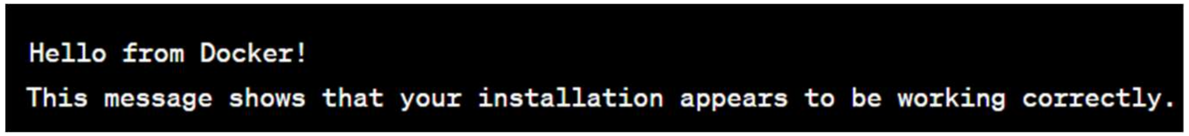
- Open your terminal or command prompt.
- To pull the "Hello World" image, use the following Docker command:

docker pull hello-world

Step 2: Run the "Hello World" Container

docker run hello-world

You should see the following output, indicating that the container ran successfully

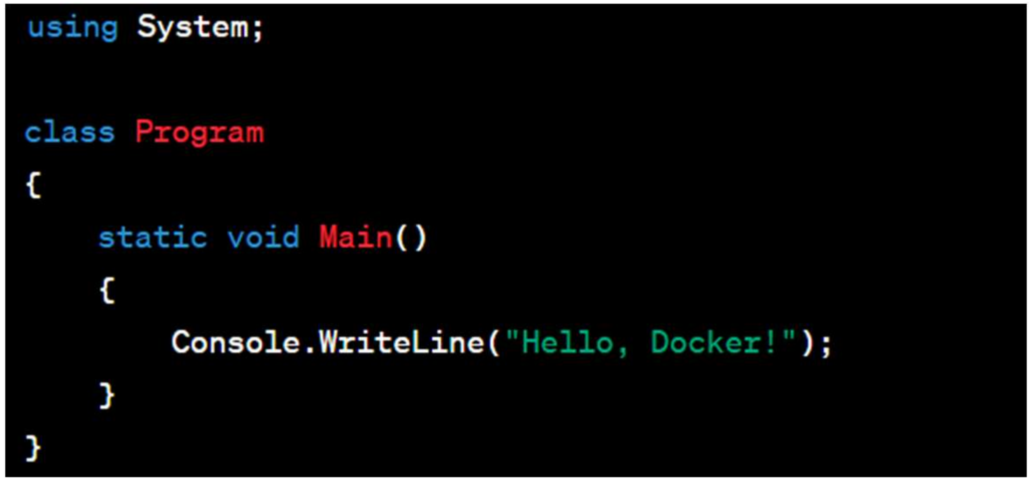


```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Code.Hub

Creating a simple Dockerfile

Step 1: Create Your C# Application - We really need something simple like



```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, Docker!");
    }
}
```

Step 2: Create a Dockerfile

Next, create a Dockerfile **in the same directory** as your C# application.

Creating a simple Dockerfile

```
# Use the official .NET Core runtime as a parent image
FROM mcr.microsoft.com/dotnet/runtime:6.0

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Run the application when the container launches
CMD ["dotnet", "Program.dll"]
```

In this Dockerfile we are doing the following:

- FROM mcr.microsoft.com/dotnet/runtime:6.0 specifies the base image, which is an official .NET Core 6.0 runtime image.
- WORKDIR /app sets the working directory inside the container to "/app."
- COPY . /app copies the contents of your local directory to the "/app" directory in the container.
- CMD ["dotnet", "Program.dll"] defines the command to run when the container starts. It uses dotnet to execute the compiled C# application (Program.dll).

Using our Dockerfile

Step 3: Build the Docker Image

Now, open your terminal, navigate to the directory containing your Dockerfile and C# application, and run the following command to build the Docker image:

```
docker build -t my-hellodocker-app .
```

- `-t my-hellodocker-app` assigns a tag ("my-hellodocker-app") to the image.
- **Don't forget the period at the end, which specifies the build context (the current directory)**

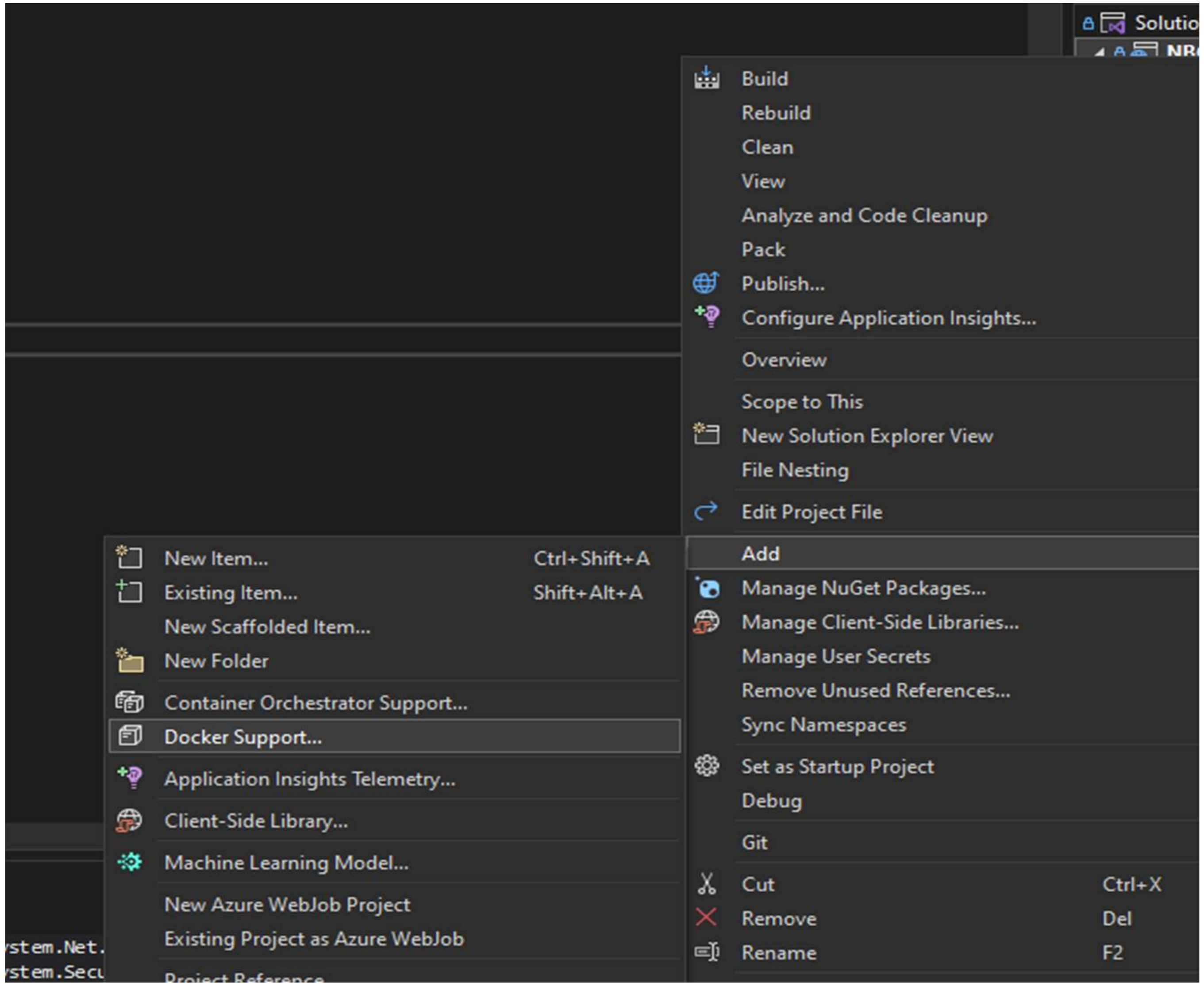
Step 4: Run a Container from the Image

```
docker run my-hellodocker-app

or…. docker run my-hellodocker-app:latest dotnet /app/bin/Debug/net6.0/HelloDocker.dll
```

Live Demo 2 - Containerizing our simple API

Step 1: Add Docker Support

This will create a dockerfile for us ...but not just that



Code.Hub

Live Demo 2 -
Containerizing our simple app

Let’s see what this docker has in place for us

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
```

- The FROM command specifies the base image to use. In this case, it uses the official .NET 6.0 ASP.NET base image from Microsoft's container registry (mcr.microsoft.com/dotnet/aspnet:6.0).
- It creates a new build stage named "base" using the AS keyword.
- WORKDIR /app sets the working directory inside the container to /app.
- EXPOSE 80 informs Docker that the container will listen on port 80. This is a declaration for documentation purposes and does not actually map any ports.

Live Demo 2 - Containerizing our simple app

```
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["NBGCustomerService/NBGCustomerService.csproj", "NBGCustomerService/"]
RUN dotnet restore "NBGCustomerService/NBGCustomerService.csproj"
COPY . .
WORKDIR "/src/NBGCustomerService"
RUN dotnet build "NBGCustomerService.csproj" -c Release -o /app/build
```

- Another build stage is created named "build," using the .NET 6.0 SDK base image. This stage is used for building the application.
- WORKDIR /src sets the working directory for this stage to /src.
- COPY is used to copy the project file (NBGCustomerService/NBGCustomerService.csproj) and its contents to the container's /src directory.
- dotnet restore is run to restore the project's dependencies.
- COPY . . copies the application source code to the container.
- WORKDIR "/src/NBGCustomerService" changes the working directory to the application folder.
- dotnet build is used to build the application in Release configuration and output the build to /app/build.

Live Demo 2 -
Containerizing our simple app

```
FROM build AS publish
RUN dotnet publish "NBGCustomerService.csproj" -c Release -o /app/publish /p:UseAppHost=false
```

- This stage is derived from the "build" stage, inheriting its context.
- dotnet publish is used to publish the application, creating a self-contained output in the /app/publish directory. The /p:UseAppHost=false option indicates not to use an app host when publishing.

Live Demo 2 -
Containerizing our simple app

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "NBGCustomerService.dll"]
```

- This stage is derived from the "base" stage, inheriting its context.
- WORKDIR /app sets the working directory to /app within the final image.
- COPY --from=publish /app/publish . copies the published application from the "publish" stage to the current working directory.
- ENTRYPOINT specifies the command to run when a container based on this image is started. In this case, it runs the .NET application by executing dotnet NBGCustomerService.dll.

Code.Hub

Useful Docker Commands for Containers

Code.Hub

Running a Container

- First Pull the relevant image from Docker Hub (e.g hello-world)
  - **docker pull hello-world**
- Run a container based on the pulled image
  - **docker run -it --name my-hw hello-world**

-it: This flag creates an interactive session and attaches your terminal to the container.
--name: Assigns a name to your container.

Managing a Container

- **To view the running containers**
  docker ps
- **To view all the existing containers**
  docker ps -a
- **To stop a running container**
  docker stop my-hw
- **To start a previously stopped container**
  docker start my-hw

Removing a Container

- **To stop and remove a container**

```
docker rm -f my-hw
```

- **To remove a previously stopped container**

```
docker rm my-hw
```

Managing Container Resources

- **Limit CPU and Mem Resources**
  ```
  docker run -it --name my-hw --cpus 2 --memory 512m ubuntu
  ```

- **To remove a previously stopped container**
  ```
  docker stats my-hw
  ```

Multi-Stage Dockerfiles

Code.Hub

Multi-stage builds in Docker

Multi-stage builds in Docker:

- Technique used to minimize the size of the final Docker image
- Separating the build environment from the runtime environment.
- Useful when building applications, as it allows you to create a lean production image without including unnecessary development tools and dependencies.

Multi-stage builds in Docker - Example

```
# Build Stage
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /app

# Copy and restore as distinct layers
COPY *.csproj .
RUN dotnet restore

# Copy the remaining source code
COPY . .
# Build the application
RUN dotnet publish -c Release -o out


# Runtime Stage
FROM mcr.microsoft.com/dotnet/runtime:6.0 AS runtime
WORKDIR /app

# Copy the published application from the build stage
COPY --from=build /app/out .

# Specify the entry point for the runtime image
ENTRYPOINT ["dotnet", "NBGCustomerService.dll"]
```

What's going on here?

- The build stage uses the .NET SDK image to compile and publish the application.
- The runtime stage uses the .NET runtime image, which is much smaller and contains only the runtime components necessary to run the application.
- The --from=build flag in the COPY command copies the published application from the build stage to the runtime stage.
- The final image only includes the runtime components and the built application, resulting in a smaller and more efficient Docker image.

Multi-stage builds Explained

1. **Build Stage:** In the first stage of the Dockerfile, you create a temporary container called a "build stage" where you **include all the tools, libraries, and dependencies required for building your application.** This typically includes compilers, build tools, and development dependencies.
2. **Copy Application Code:** You copy your application source code into the build stage container.
3. **Build Application:** You build your application within the build stage container using the necessary tools and dependencies.
4. **Runtime Stage:** In the second stage of the Dockerfile, you create another container called the "runtime stage." This container includes only the runtime components necessary to run your application, such as the .NET Core runtime and minimal dependencies. It doesn't include any of the build tools.
5. **Copy Built Artifacts:** You copy the built artifacts (e.g., executable files) from the build stage container into the runtime stage container.
6. **Final Image:** The final image is created from the runtime stage, resulting in a lightweight image that contains only what is needed to run your application, without the overhead of build dependencies.

Thank you!

Code.Hub