

Introduction to Docker and Containers

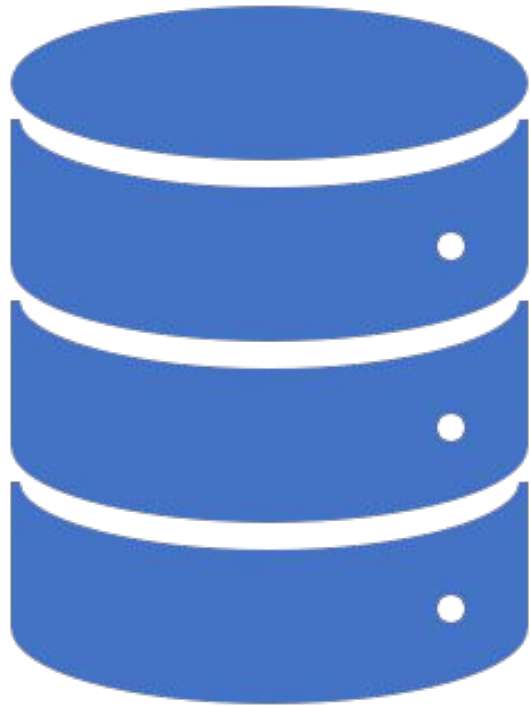
JJTECH 2023



Topics

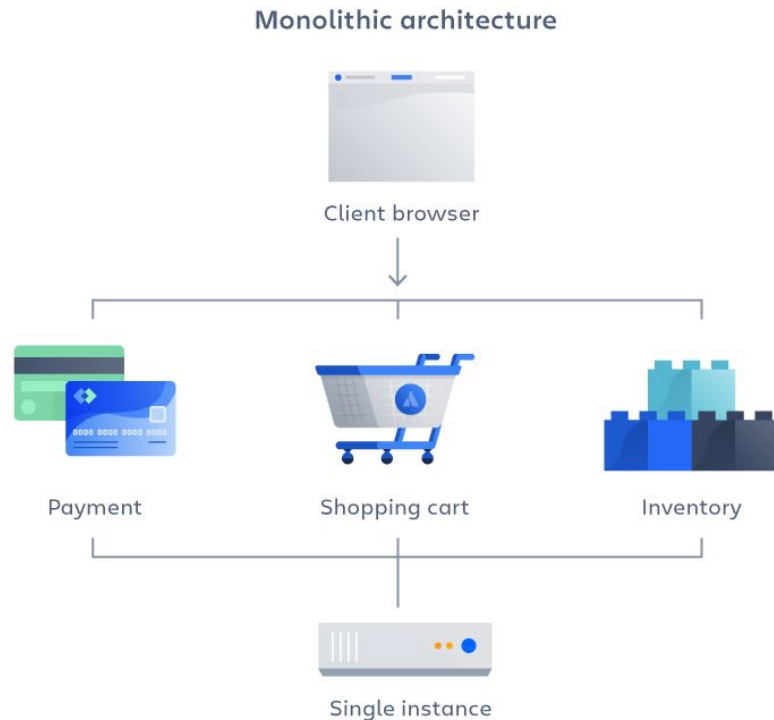
- Monolithic and Microservices Architecture
- Containerization
- Docker
- Dockerfile
- Docker Compose
- Docker Registry
- Docker volume
- Docker Network





Monolithic vs Microservices Architecture

Monolithic Architecture



- Monolithic architecture is like a big container, where all the software components of an app are created and tightly coupled
- All functionalities of a project exist in a single codebase,
- Changes to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface.

Advantages and Disadvantages of Monolithic Architecture

Advantages:

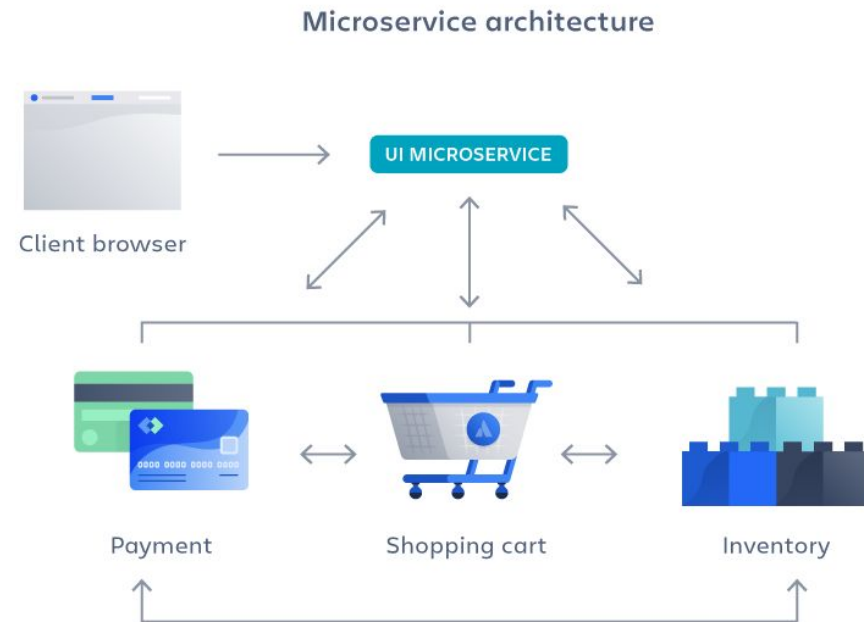
- **Easy deployment** – One executable file or directory makes deployment easier.
 - **Development** – When an application is built with one code base, it is easier to develop. Developers need not learn different applications; they can keep their focus on one application.
 - **Performance** – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
 - **Simplified testing** – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.
- Easy debugging** – With all code located in one place, it's easier to follow a request and find an issue.

Disadvantages:

- **Slower development speed** – A large, monolithic application makes development more complex and slower.
- **Scalability** – You can't scale individual components.
- **Reliability** – It is not very reliable, as a single bug in any module can bring down the entire monolithic application.
- **Barrier to technology adoption** – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.
- **Deployment** – A small change to a monolithic application requires the redeployment of the entire monolith.

Microservices Architecture

- It is an architectural method that relies on a series of independently deployable services
- These services have their own business logic and database with a specific goal
- Updating, testing, deployment, and scaling occur within each service
- Microservices decouple major business, domain-specific concerns into separate, independent code bases and deployed as containers



Advantages and Disadvantages of Microservices Architecture

Advantages:

- Agility** – Microservices are self-contained and, hence, deployed independently. Their start-up and deployment times are relatively less.
- Flexible scaling** – If a particular microservice is facing a large load because of the users using that functionality in excess, then we need to scale out that microservice only. Hence, the microservices architecture supports horizontal scaling.
- Highly maintainable and testable** – Teams can experiment with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time-to-market for new features. It is easy to manage as it is relatively smaller.
- Independently deployable** – Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- Technology flexibility** – Each microservice can use different technology based on the business requirements.
- High reliability** – If a particular microservice goes down due to some bug, then it doesn't affect other microservices and the whole system remains intact and continues providing other functionalities to the users.
- Adaptability** – It is very easy for a new developer to onboard the project as he/she needs to understand only a particular microservice providing the functionality he will be working on and not the whole system.

Disadvantages:

- Development complexity** – Microservices add more complexity compared to a monolith architecture, since there are more services in more places created by multiple teams. If development complexity isn't properly managed, it results in slower development speed and poor operational performance.
- Exponential infrastructure costs** – Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.
- Added organizational overhead** – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.
- Debugging challenges** – Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
- Lack of standardization** – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.
- Lack of clear ownership** – As more services are introduced, so are the number of teams running those services. Over time it becomes difficult to know the available services a team can leverage and who to contact for support.
- Security concerns** – Microservices are less secure relative to monolithic applications due to the inter-services communication over the network.

Containerization

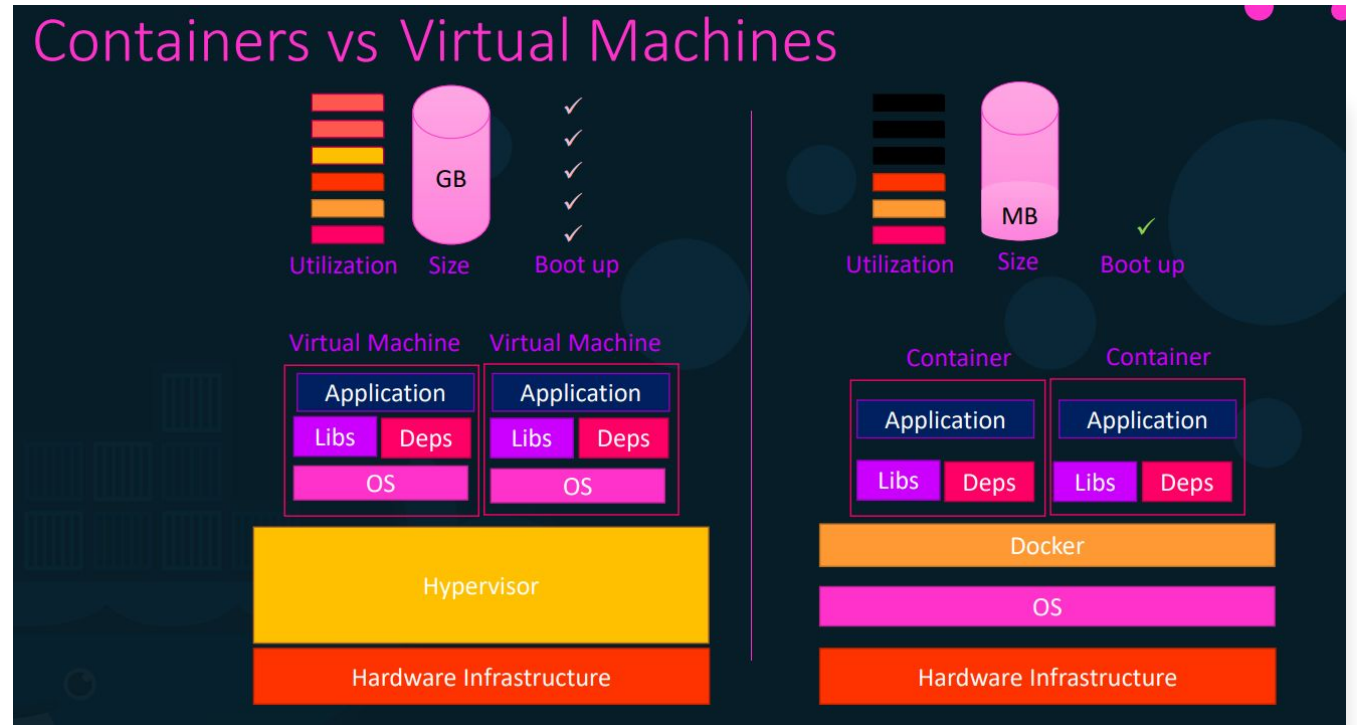


A **container** is a portable computing environment. It contains everything an application needs to run, from binaries to dependencies to configuration files.



Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system.

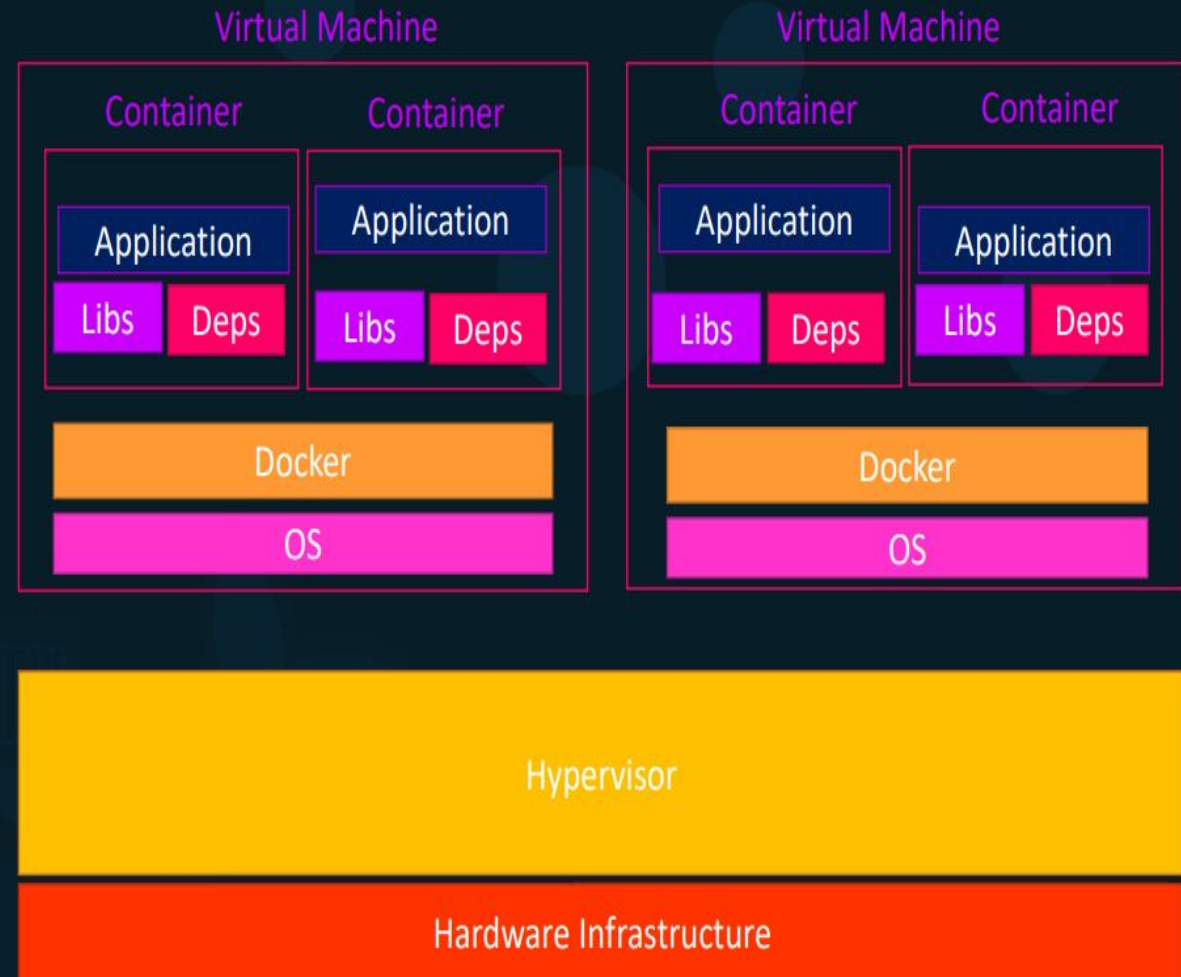
Containers vs Virtual Machines



VMS	CONTAINERS
Heavyweight.	Lightweight.
Each VM runs in its own OS.	All containers share the host OS.
Hardware-level virtualization.	OS virtualization.
Startup time in minutes.	Startup time in milliseconds.
Allocates required memory.	Requires less memory space.
Fully isolated and hence more secure.	Process-level isolation, possibly less secure.

Containers vs Virtual Machines

Containers & Virtual Machines

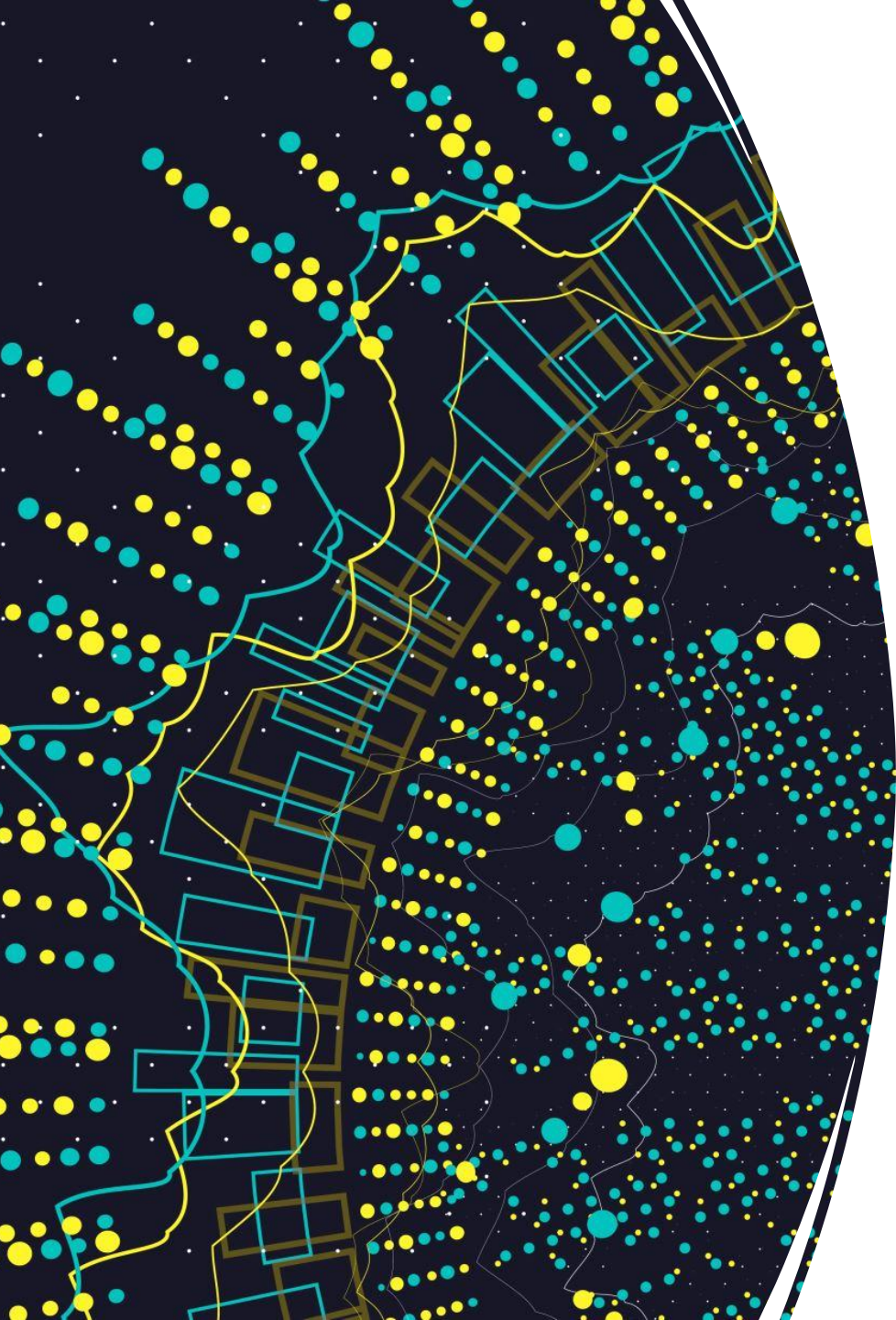


What is Docker

<https://docs.docker.com/desktop/>



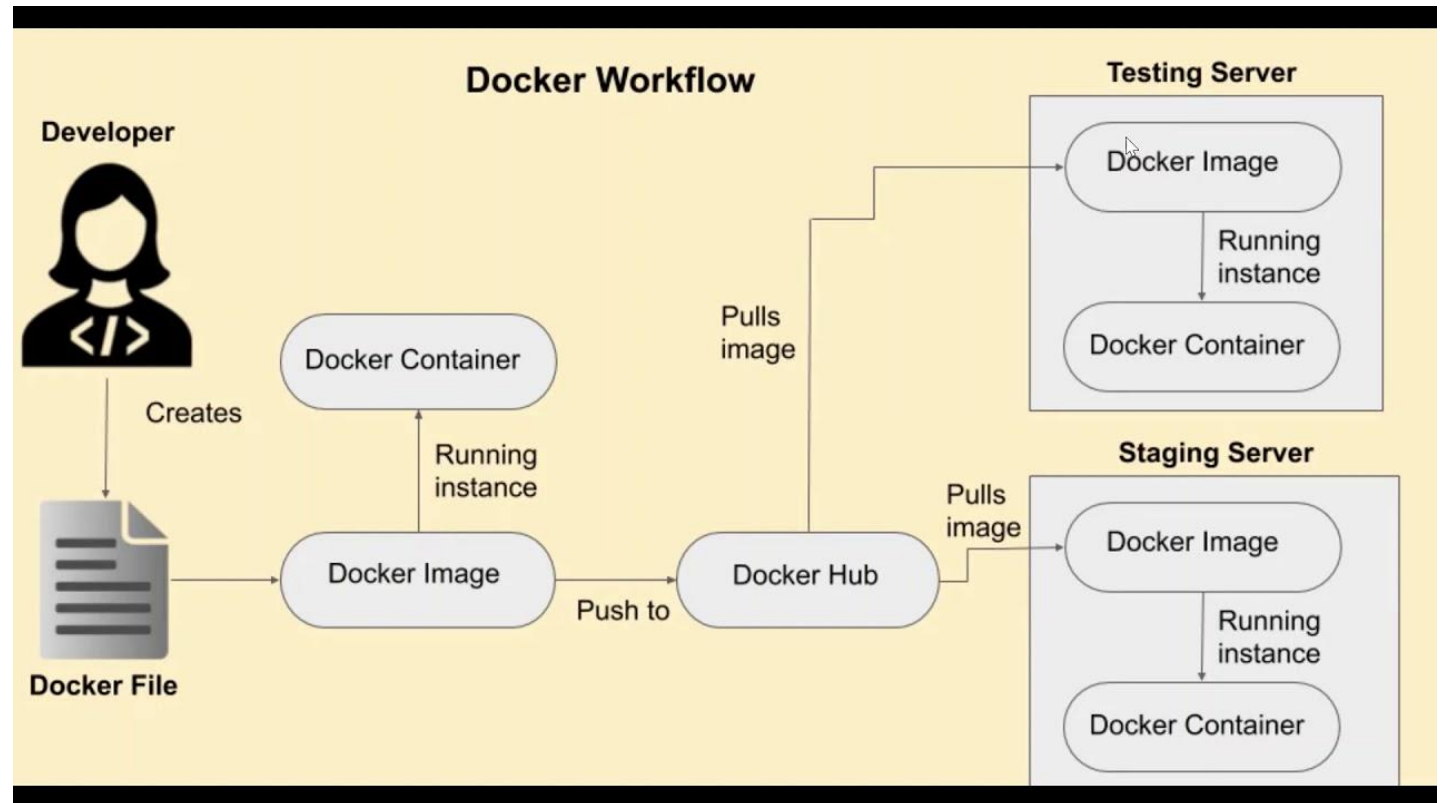
docker



What is Docker

- Docker is an open-source platform for creating, deploying, and running applications using containers.
- It allows developers to wrap up an application, or part of an application, into a container, which can be deployed on any system with the Docker runtime installed

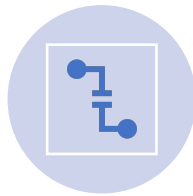
Docker Workflow



Benefits of Docker



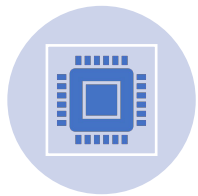
Portability: Docker containers can run on any machine that supports Docker, regardless of the underlying operating system or hardware, making it easy to move applications between environments.



Isolation: Docker containers provide a high level of isolation between applications and their dependencies, which helps to avoid conflicts and ensures that each application runs in a consistent and reproducible environment.



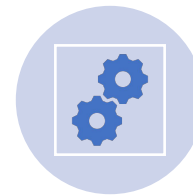
Efficiency: Docker containers are lightweight and share the underlying host operating system, which means that they require less resources than traditional virtual machines, making them more efficient.



Scalability: Docker containers can be easily scaled up or down depending on the application workload, which enables organizations to quickly respond to changing business requirements.



Consistency: Docker images provide a consistent and repeatable way to package and deploy applications, which helps to ensure that the application runs the same way in any environment.



DevOps integration: Docker integrates well with modern DevOps toolchains and provides a standardized way to package and deploy applications, making it easier to automate the deployment process.

Docker Security

- **Use the Latest Version of Docker:** Make sure to always use the latest version of Docker to ensure you are taking advantage of the latest security features and bug fixes.
- **Implement Security Policies and Procedures:** Establish security policies and procedures for developing with Docker. These policies should include topics such as user authentication, access control, and patching.
- **Use Signed Images:** Whenever possible, use signed Docker images from an official source. Signed images are cryptographically verified and provide an extra layer of security.
- **Implement the Least Privilege Principle:** When launching Docker containers, make sure to use the principle of least privilege and limit user access to only the necessary resources.
- **Monitor Network Connections:** Monitor network connections to and from Docker containers to ensure only authorized access is allowed.
- **Scan Images:** Scan Docker images for vulnerabilities and malicious code before deploying them.
- **Monitor Container Logs:** Monitor container logs on a regular basis to identify unusual or suspicious activity.
- **Harden the Host System:** Harden the underlying host system to prevent unauthorized access and protect it from malicious attacks.
- **Implement Security Automation:** Utilize security automation to scan images and containers for vulnerabilities and malicious code. Automation can also be used to detect suspicious network traffic and unauthorized access.



Docker Image

- A Docker image is a packaged and self-contained software environment that includes all the dependencies, configurations, and libraries needed to run an application or service.
 - Docker images are built using a Dockerfile, which contains a set of instructions that specify how to create the image.
 - Docker images are designed to be portable and lightweight, making it easy to run an application or service across different environments and platforms.
 - Docker images can be easily shared and deployed, and can be run as Docker containers, which are isolated and portable execution environments.
 - Docker images can be stored and managed in a Docker registry, which is a central repository for storing and distributing Docker images. Popular Docker registries include Docker Hub, Google Container Registry, and Amazon Elastic Container Registry.
-

Dockerfile

A Dockerfile is a text file that contains a set of instructions used to build a Docker image.

Dockerfile

The instructions in a Dockerfile specify:

- how to build a Docker image, including the base image to use, the files and directories to copy into the image,
- the commands to run to set up the environment and configure the application.
- Each instruction in the Dockerfile creates a new layer in the image, which can be cached and reused to speed up the build process.

Dockerfile Instructions

- FROM:** specifies the base image to use for the Docker image
- RUN:** runs a command inside the Docker image during the build process
- COPY:** copies files and directories from the host system into the Docker image
- EXPOSE:** exposes a port for the Docker container
- ENV:** sets environment variables in the Docker image.
- ENTRYPOINT:** specifies the command to run when the Docker container starts.
- CMD:** provides default arguments for the ENTRYPOINT command.
- ARG:** defines build-time variables that can be passed to the Docker build command using the --build-arg option.
- WORKDIR:** sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it.
- USER:** sets the user account to use when running commands in the Docker image.
- VOLUME:** creates a mount point for a volume in the Docker container.
- ADD:** copies files or directories from the host system into the Docker image. Can also retrieve and extract files from remote URLs or archives.
- LABEL:** adds metadata to the Docker image in the form of key-value pairs.
- MAINTAINER:** sets the name and email address of the image maintainer.
- ONBUILD:** specifies instructions to run when the Docker image is used as the base image for another Dockerfile.
- HEALTHCHECK:** defines a command to run to check the health of the container.

Sample Dockerfile

```
FROM ubuntu
LABEL JJTECH
RUN apt update
RUN apt full-upgrade -y && apt install python-pip -y
RUN pip2 install flask
WORKDIR /opt
COPY app.py /opt/app.py
ENTRYPOINT FLASK_APP=/opt/app.py flask run --host=0.0.0.0 --port=8080
```


Docker Commands

Practice Time !!!

- Docker CLI commands
- Dockerfile
- Scan images with Snyk



Docker Registry

- A Docker registry is a storage and content delivery system that holds named Docker images, available in different tagged versions.
- It enables Docker users to store and distribute Docker images.
- Docker images can be pushed to and pulled from a registry
- The most popular public Docker registry is Docker Hub, but private registries can also be set up for organizations that need to manage their own images internally.

Docker Registry Flow

- Create a Dockerfile: The first step is to create a Dockerfile that specifies the instructions to build the image.
- Build the Docker image: Once the Dockerfile is created, the next step is to build the Docker image using the **docker build** command.
- Tag the Docker image: After building the Docker image, the next step is to tag it with a unique identifier. The tag is a label that is attached to the image, which helps to identify the image and differentiate it from other images.
- Push the Docker image to a registry: Finally, the Docker image can be pushed to a Docker registry using the **docker push** command. The registry is a centralized location where Docker images can be stored and managed. Common Docker registries include Docker Hub, Amazon ECR, and Google Container Registry.

Docker Networks

In Docker, a network is a communication pathway between containers that enables them to communicate with each other and with other resources outside of the container environment.



Docker Networks

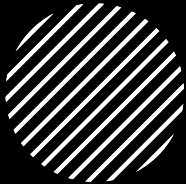
Types

1. Bridge network: This is the default network in Docker, which allows containers to communicate with each other and with resources outside of the container environment. However, it is limited in terms of scalability and does not provide name resolution.
2. Host network: This allows the container to use the network stack of the host, which means that the container shares the same IP address as the host and can access resources on the host network.
3. Overlay network: This is a distributed network that spans multiple hosts and enables containers to communicate with each other across different hosts.
4. None network: This disables networking for the container, which is useful in scenarios where network access is not required.





Docker Volumes



- In Docker, a volume is a way to persist data generated by a container, even after the container has stopped or been deleted.
- Docker volumes provide a convenient and reliable way to store data outside of the container file system and can be used to share data between containers or to persist data between container restarts.
- Volumes can be used to store a wide variety of data, such as application logs, databases, and configuration files. They provide a convenient and reliable way to store data in a containerized environment and are an essential tool for building scalable and reliable Docker applications.

Docker Volume Types



Host-mounted volumes: These are volumes that are mapped to a directory on the host file system, which enables the data to be persisted even after the container is deleted.



Docker-managed volumes: These are volumes that are managed by Docker and are stored in a directory on the host file system, which is managed by Docker.



Named volumes: These are volumes that are given a specific name when they are created and can be used to share data between containers or to persist data between container restarts.

Docker Compose

- Docker Compose is a tool that allows you to define and run multi-container Docker applications.
- It allows you to define the services that make up your application, how they are connected, and how they should be run.
- With Docker Compose, you can define your application's infrastructure as code, making it easy to maintain, scale, and deploy.
- Docker Compose uses YAML files `(docker-compose.yaml)` to define the services, networks, and volumes that make up your application.
- To run your applications with docker compose, use the command `docker-compose up` to deploy the apps or `docker-compose down` to stop the apps

Docker Compose Command Options

sub-command	does
up	Builds, re(creates), starts and attaches to containers for a service.
down	Stops containers and removes containers, networks, volumes, and images created by up .
top	Displays the running processes.
start	Starts existing containers for a service.
stop	Stops running containers without removing them. They can be started again with start .
pause	Pauses running containers of a service. They can be unpaused with unpause .
build	Builds services and tags them as (by default) <i>project_service</i> .
logs	Displays log output from services.
scale	Sets the number of containers to run for a service. Deprecated! use --scale flag of the up command instead.
images	Lists images used by the created containers.
kill	Forces running containers to stop by sending a <i>SIGKILL</i> signal.
rm	Removes stopped service containers.

```

simple-app2 > ! docker > {} services > {} web > [ ] ports
1  version: "3.9" # the version of the Docker Compose file syntax to use
2
3  services: # defines the services to run
4    web: # the name of the service
5      build: # build the image using the Dockerfile in the current directory
6        context: . # the build context (the current directory)
7        dockerfile: Dockerfile # the name of the Dockerfile
8        args: # build arguments
9          ARG_NAME: "value" # a build argument with the name "ARG_NAME" and value "value"
10       image: example-image:latest # use an existing image instead of building one
11       container_name: container-name # the name of the container
12       command: command # override the default command
13       entrypoint: entrypoint # override the default entrypoint
14       environment: # environment variables
15         VAR_NAME: "value" # an environment variable with the name "VAR_NAME" and value "value"
16       env_file: # environment variables from a file
17         - env_file_name # the name of the file
18       ports: # port mapping
19         - "80:80" # map port 80 on the host to port 80 in the container
20       volumes: # mount volumes
21         - /path/on/host:/path/in/container # mount /path/on/host as /path/in/container in the container
22       networks: # attach to networks
23         - network-name # the name of the network
24       depends_on: # depend on other services
25         - service-name # the name of the service to depend on
26       restart: "no" # restart policy ("no", "always", or "on-failure")
27       mem_limit: 512m # memory limit
28       cpu_shares: 1024 # CPU shares
29       healthcheck: # healthcheck
30         test: ["CMD", "curl", "-f", "http://localhost"] # the command to run the healthcheck
31         interval: 30s # the interval between healthchecks
32         timeout: 10s # the timeout for each healthcheck
33         retries: 3 # the number of retries before considering the container unhealthy

```

Docker Compose File

Note: Not all of these arguments are required or applicable for every service, and there may be additional arguments available depending on your version of Docker Compose



Thank you