VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

PHAM HOANG SON – 521H0476
NGUYEN DINH VIET HOANG - 522H0120

MIDTERM REPORT
WEB APPLICATION DEVELOPMENT USING NODEJS

HO CHI MINH CITY, YEAR 2024

VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

PHAM HOANG SON – 521H0476
NGUYEN DINH VIET HOANG - 522H0120

MIDTERM REPORT
WEB APPLICATION DEVELOPMENT USING NODEJS

Advised by

MsC. MAI VAN MANH

HO CHI MINH CITY, YEAR 2024

ACKNOWLEDGEMENT

We sincerely thank MsC. Mai Van Manh for teaching us the Web Application Development Using NodeJS course with great enthusiasm. We want to express our deep appreciation for the dedication and professional knowledge that you shared with us. Through your classes, we gained a better understanding of the fundamental aspects of the Web Application Development Using NodeJS, thanks to your detailed explanations and practical applications. You helped us grasp the knowledge and apply it effectively. Finally, we extend our heartfelt gratitude to MsC. Mai Van Manh for your commitment and invaluable support throughout our learning journey in this course. The skills and knowledge we acquired will continue to impact our future development. We sincerely thank you and wish your health, success, and happiness.

Ho Chi Minh City, October 21, 2024

Authors:

Pham Hoang Son

Nguyen Dinh Viet Hoang

DECLARATION OF AUTHORSHIP

We hereby declare that this thesis was carried out by ourselves under the guidance and supervision of MsC. Mai Van Manh; and that the work and the results contained in it are original and have not been submitted anywhere for any previous purposes. The data and figures presented in this thesis are for analysis, comments, and evaluations from various resources by our own work and have been duly acknowledged in the reference part.

In addition, other comments, reviews and data used by other authors, and organizations have been acknowledged, and explicitly cited.

We will take full responsibility for any fraud detected in my thesis. Ton Duc Thang University is unrelated to any copyright infringement caused on my work (if any).

Ho Chi Minh City, October 21, 2024

Authors:

Pham Hoang Son

Nguyen Dinh Viet Hoang

ABSTRACT

TABLE OF CONTENTS

sdf

LIST OF FIGURES

LIST OF TABLES

sdfgsdf

LIST OF ABBREVIATIONS

dsdfg

CHAPTER 1: INTRODUCTION TO DOCKER

1.1 Docker definition and its importance

1.1.1 What is docker?

Docker is an open-source platform that helps build, deploy, and manage applications in containers. Containers are self-contained units that include everything needed to run an application, like the source code, libraries, and the operating system environment. By isolating applications from the underlying infrastructure, Docker makes deployment consistent and straightforward across different environments, such as development, testing, and production.

1.1.2 Why docker is important?

Docker is essential in modern web development because it ensures consistency across systems, from local development to production. It addresses the common issue of "it works on my machine but not on yours" by creating a uniform environment for running applications. Containers are also more resource-efficient than virtual machines, which makes it easier to scale applications as needed. Docker improves performance, reduces hardware costs through better resource management, and integrates with CI/CD pipelines to automate and speed up deployments. This allows applications and their dependencies to move smoothly across various environments, including local machines, on-premise servers, and cloud infrastructure.

1.2 Key Concepts

1.2.1 Containers

Containers are packages that include an application and everything it needs to run, such as libraries and dependencies. They provide an isolated environment while sharing the host's kernel, which helps optimize resource usage. Containers are lightweight, fast, and reusable. They start quickly and use fewer resources compared to virtual machines because they don't need a full operating system.

1.2.2 Images

Docker images are unchangeable files that contain the source code and the environment needed to run containers. An image is like a snapshot of an application at a certain point and is used to create containers. Images can have multiple layers based on changes made when they were created (using a Dockerfile). They can be stored and shared on registries like Docker Hub.

1.2.3 Dockerfile

A Dockerfile is a text file with instructions that Docker uses to build an image. It automates the image creation process and specifies the environment required for an application.

Example of a Dockerfile:

```
FROM node:14
WORKDIR /app
COPY . .
RUN npm install
CMD ["npm", "start"]
```

This Dockerfile sets up a Node.js environment, copies the application code, installs dependencies, and starts the application with npm start.

1.2.4 Docker Compose

Docker Compose is a tool for managing multiple containers in a complex application. It lets you define containers and their interactions in a file called docker-compose.yml. With Docker Compose, you can start or stop all containers for an application with one command.

Example of Docker Compose:

File yaml:

```
version: '3'

services:

  web:

    image: nginx

    ports:

      - "80:80"

  database:

    image: postgres

    environment:

      POSTGRES_PASSWORD: example
```

In this example, Docker Compose sets up two services: one container running the NGINX web server and another running a PostgreSQL database.

1.2.5 Container Orchestration

Container orchestration is the automation of deploying, managing, scaling, and coordinating containers in a distributed system. The most popular tool for this is Kubernetes. Kubernetes helps manage containers across multiple servers, ensuring that applications run smoothly. It automatically distributes resources, balances the load, and restarts any failed containers. This makes it easier to scale applications and ensures they are always available.

CHAPTER 2: THEORETICAL SURVEY

2.1 Docker architecture and its underlying technology

2.1.1 Docker architecture

Docker has a client-server architecture. The Docker server (or daemon) builds, runs, and distributes Docker containers. The Docker client and server can be on the same machine or different machines, and they communicate via a REST API using UNIX sockets or network interfaces.

2.1.1.1 Docker daemon

The main part of Docker is the daemon (called dockerd), which listens for API requests and manages Docker objects. It can also communicate with daemons on other machines.

2.1.1.2 Docker client

The Docker client (simply docker) is how users interact with Docker. When you type a command like docker run imageABC, you're using the CLI to send requests to the daemon. The Docker client can communicate with multiple Docker daemons.

2.1.1.3 Docker registry

A Docker registry is a place where images are stored. The most well-known registry is Docker Hub, but you can also create your own registry.

2.1.1.4 Docker object

These objects are the objects that you often encounter when using Docker, including:

1. Images
    - An image is a read-only template used to run a container.
    - Images can be based on other images. For example, to create an Nginx image, it needs to be based on an Ubuntu image since Nginx runs on Ubuntu.
    - You can create your own images or download existing ones from the Docker registry.
    - Later, I'll explain how to build your own image using a Dockerfile and discuss image layers.
2. Container
    - Containers are created and run from specific images. You can create, start, stop, move, and delete containers.
    - You can connect containers to each other, attach storage, or recreate an image from a container's current state.
    - Containers keep their resources separate from the host and other containers.

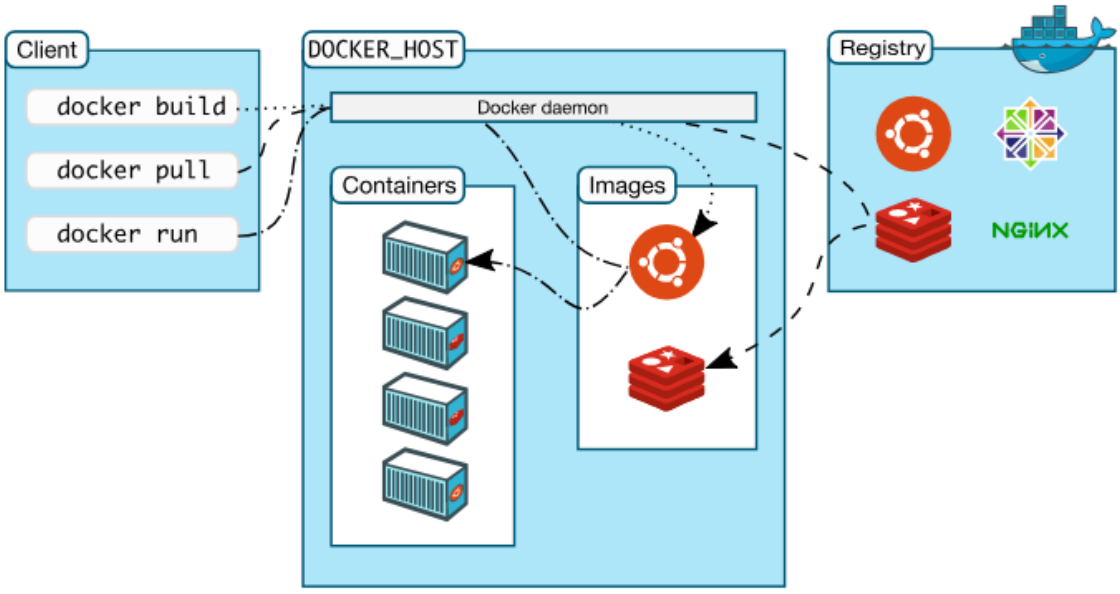

Figure 2.1.1: Example of a process working with Docker.

Source: congdonglinux.

I will take an example of the process to run a container:

1. To run a blank CentOS container, you use the command:

```
docker run -itd centos
```

2. The Docker daemon checks if it has the CentOS image. If it does, it will run the container. If not, it will download the latest CentOS image from the Registry and then run the container. You can check if the container is running with the command:

```
docker ps
```

2.1.2 Docker platform technology

2.1.2.1 Namespaces

Namespaces have been part of the Linux kernel since around 2002, with more features added over time. In 2013, real container functionality was introduced, making namespaces popular and useful. They allow the creation of isolated environments, so each container only sees what it has access to within its namespace. When you start a container, Docker creates unique namespaces for it.

Docker's namespaces help separate and manage resources effectively. They are crucial for isolating processes from one another. By isolating different services on a server, you can reduce the impact of changes and security issues, making the system more stable and secure.

1. Types of Namespaces
    - Process ID (PID) Namespace: This gives each process a unique ID that isn’t shared with others. The first process in a new namespace is assigned PID 1, and child processes follow in order.
    - Mount Namespace: Each mount namespace has its own set of mount points. This means you can mount and unmount filesystems without affecting the main filesystem on the host.
    - User Namespace: Each user namespace has its own user and group IDs. A process can have root privileges in its user namespace but not in others.
    - UNIX Time-Sharing (UTS) Namespace: Processes can have different host and domain names even if they are on the same system.

2. Understanding General Process-Sharing Resources
    - In a general computing environment, multiple processes share resources, which can lead to conflicts and management issues.
    - When processes run on the same machine without isolation, they may interfere with each other, causing resource congestion and security problems.
    - Since processes share the same namespace, they lack independence. This can lead to conflicts, like two processes trying to access the same memory or file, resulting in unstable behavior.
3. How Do Docker Namespaces Work?
    1) To enable User Namespaces in Docker, add the user with the command “sudo adduser dockremapper”.
    2) Then restart Docker using the command “sudo /etc/init.d/docker”.
    3) Check access: After configuration, new Docker containers cannot access the host's “/etc/” directory.
    4) Check the file structure of the newly set up system by cd to the correct directory. To disable user namespaces for a container, use the “--userns=host parameter”.
    5) Run the “docker run” command to initialize a new bash.
    6) Enter the “nsenter” command to access the container by specifying different namespaces.
    7) Finally, use “nsenter -t <target IP> -a” to access the entire namespace of the container, similar to the “docker attach <name>” command.

2.1.2.2 cgroups

1. Understanding cgroups

Cgroups are a feature of the Linux kernel that allows for the management and partitioning of system resources by controlling resources for a group of processes. Administrators can use cgroups to allocate resources, set limits, and prioritize processes. Docker uses cgroups to limit resources for containers.

There are many types of cgroups such as CPU cgroup, memory cgroup, block I/O cgroup, and device cgroup. Although cgroups are not specifically designed for security purposes, they are important for controlling and monitoring resource usage by processes.

Namespaces and cgroups differ in purpose: namespaces create separate environments to isolate processes, while cgroups manage resource allocation and limitation. Often, both are used together to isolate and manage resources.

2. How to Use cgroups to Control Docker Container Resources?

First, to check if your system is using cgroup v1 or v2, look for the file /sys/fs/cgroup/cgroup.controllers. If it exists, you're using cgroup v2. Otherwise, it's cgroup v1.

To see what cgroups are being used, run an Nginx container:

```
docker run -d nginx
```

Then, based on the cgroup version and driver, you can check the memory usage at the following locations:

- cgroup v1, cgroupfs driver:

```
/sys/fs/cgroup/memory/docker/<container_id>/
```

- cgroup v1, systemd driver:

```
/sys/fs/cgroup/memory/system.slice/docker-<container_id>.scope/
```

- cgroup v2, cgroupfs driver:

```
/sys/fs/cgroup/docker/<container_id>/
```

- cgroup v2, systemd driver:

```
/sys/fs/cgroup/system.slice/docker-<container_id>.scope/
```

If you want to limit the CPU usage of the container, you can do so by recreating it with the --cpus option:

```
docker run --cpus 0.5 -d nginx
```

This limits the container to use only half of one CPU.

To limit memory usage, use the --memory option. For example, to limit a container to 256 MB of memory:

```
docker run -d --name new-container --memory=256M ubuntu sleep infinity
```

You can check if the memory limits were applied by using docker stats:

```
docker stats new-container
```

This shows real-time statistics, including memory usage and CPU percentage. You can also format the output to make it more readable:

```
docker stats new-container --no-stream
--format "  {{ json . }}" | python3 -m json
.tool
```

This command will show detailed info about memory limits and CPU usage. For example, a memory limit of 256 MiB will be displayed in the "MemUsage" field.

2.1.2.3 Union file systems

1. What is a Union File System?

UnionFS is a filesystem used in Linux, FreeBSD, and NetBSD that merges different file systems into one. It lets you combine multiple file systems (called branches) so their files and directories appear as a single, unified file system. If the same file or directory exists in different branches, the one from the top layer will be used. This allows you to overlay file systems, with changes appearing in the top layer, while keeping the original files in the lower layers untouched.

2. So how does it work?

An overlay filesystem (OverlayFS) works like using an overhead projector with transparent sheets. The base sheet is like the original file system, and the top sheet is where you make notes (or changes). You can keep both layers separate, so the original file (the base) stays untouched while you can see and use the changes (the notes). This allows you to overlay files or directories without altering the original data.

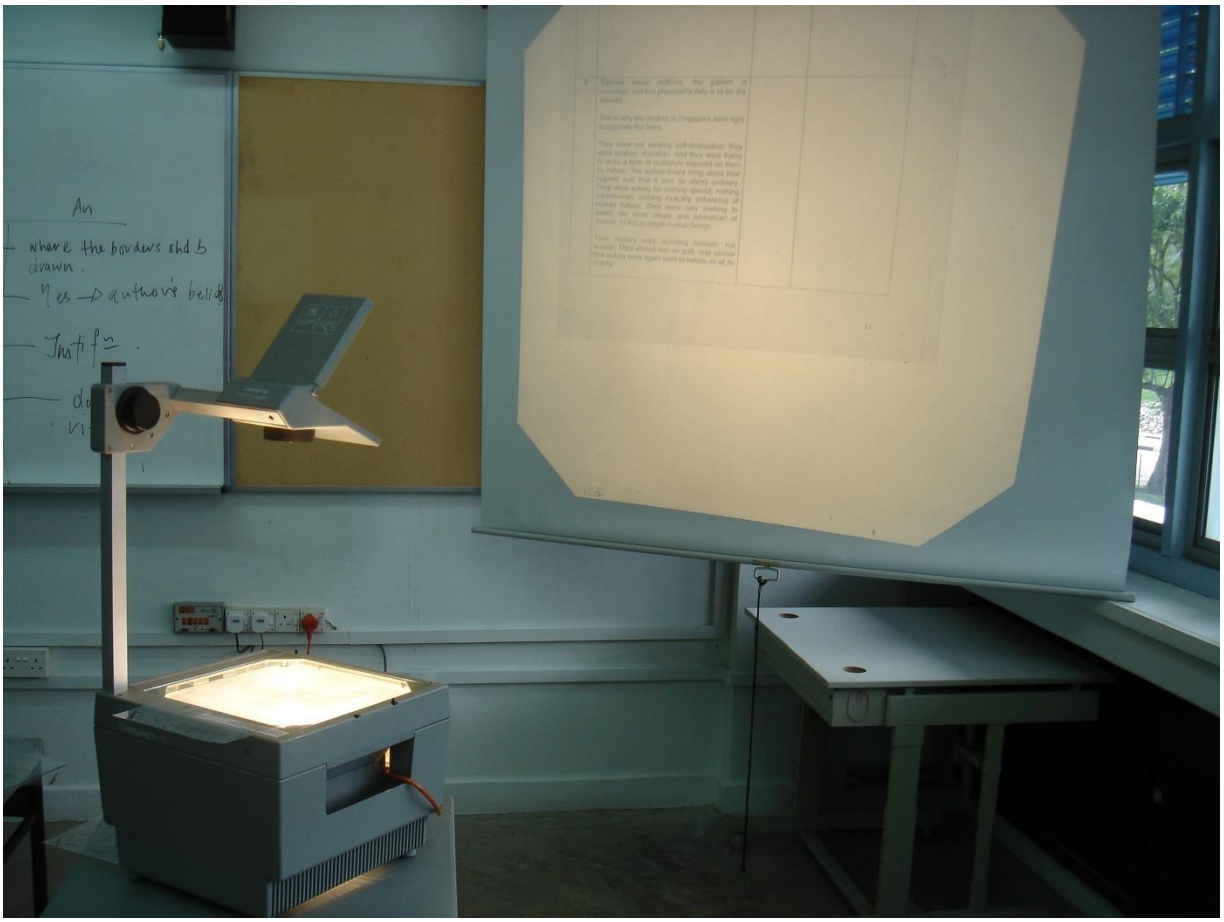

Figure 1.1.2.3: How does Unions work?

3. So how does it work? (From a technical perspective)
    a) Layers Involved
        - Base Layer (Read Only): This is where the base files are stored, and it's read-only. In Docker, this would be like the base image.
        - Overlay Layer (Main User View): This is where the user interacts. Initially, it shows the base layer and allows the user to make changes, which are handled by the Diff Layer.
        - Diff Layer: Any changes the user makes are stored here.

b) Copy-on-Write

If you change a file already in the Base Layer, OverlayFS copies it to the Diff Layer and makes the changes there. This process is called "copy-on-write" and is essential for OverlayFS to function properly.

4. How to set up an overlay(fs)

Here's a simplified explanation of how to set up an OverlayFS:

1) Create directories

```
mkdir base diff overlay workdir
```

Add files to the base layer:

```
echo "this is my base layer" > base/file1
```

2) Mount the overlay filesystem:

```
sudo mount -t overlay -o lowerdir=base,upperdir=diff,workdir=workdir overlay overlay
```

Now, the overlay directory shows the combined view of the base and diff layers. You can add files in the overlay directory, and they'll appear in the diff directory.

3) Unmount when done:

```
sudo umount overlay
```

5. So how does this relate back to Docker Images?

Relation to Docker: Docker uses the same concept with its overlay2 storage driver, stacking multiple layers (like base and diff) to form Docker images. Each container runs with these layers, and any changes are stored in the diff layer, just like OverlayFS.

2.2 How Docker Simplifies Node.js Application Development and Deployment

2.2.1 Discuss how Docker simplifies development

Before Docker was publicly introduced at PyCon in 2013, developers used many different tools to manage software deployments. This included virtual machines, configuration management tools, package management systems, and other complicated libraries. Managing all these tools required special knowledge and setup, which made software deployments complex and costly. Docker simplified this process by providing a more efficient way to deploy applications.

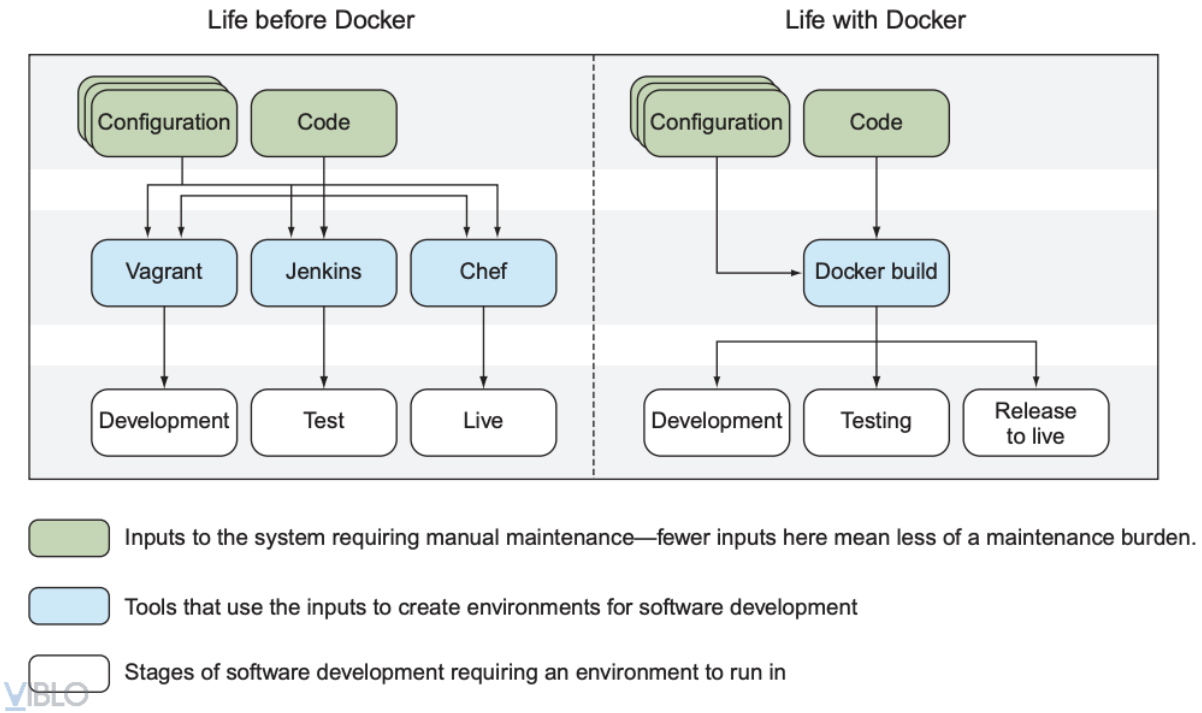

Figure 2.2.1.a: Software development and deployment process before and after using Docker

Docker Makes Deployment Simpler and More Efficient

Docker simplifies the process of deploying applications by using containers. Containers package an application and all its dependencies together, making it easier to manage development and deployment environments. This means developers can build, ship, and run applications anywhere quickly and consistently.

Real-Life Example: Containerization in Shipping

Imagine a busy port with many ships carrying goods. In the past, dock workers had to manually load different items onto each ship, which was time-consuming and costly. This required a lot of workers and effort, and every ship had to stop while the loading took place.

Now, instead of loading individual items directly, workers use containers to hold the goods. One dock worker, or "docker," can then operate machines to move these containers onto the ship. This method saves time and money, while also ensuring that the goods are safely and consistently loaded.

Just like how it doesn’t matter what shape or size the ship is, containers make it irrelevant what the goods inside look like. Similarly, Docker allows developers to focus on their applications without worrying about the underlying environment.

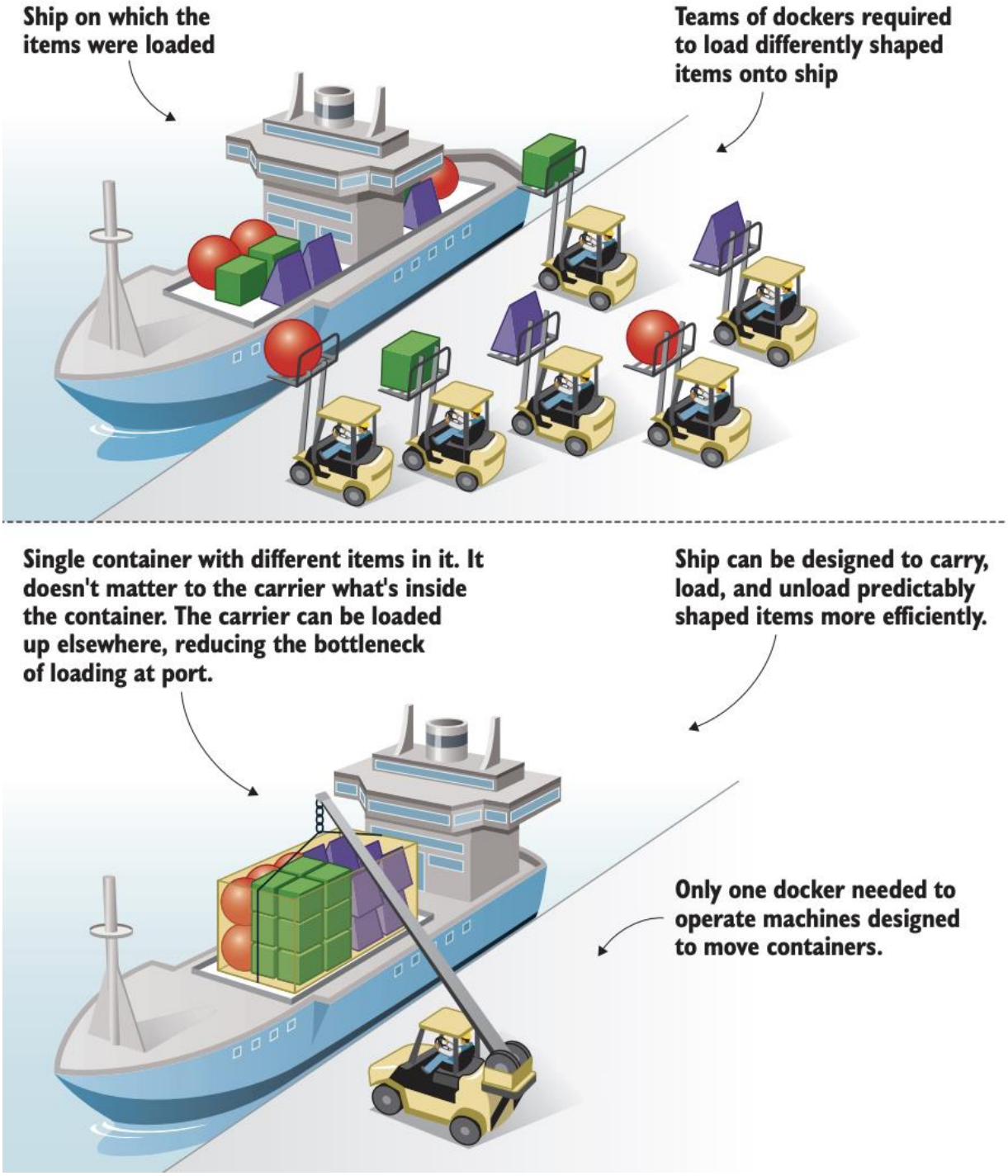

Figure 2.2.1.b: A real-life example

2.2.2 Discuss how Docker simplifies Node.js application development and deployment

Docker brings many benefits to Node.js application development and deployment by simplifying the environment and increasing flexibility.

2.2.2.1 Create a consistent development environment

Problem: Node.js versions can differ between development machines, servers, and CI/CD environments, leading to errors.

Docker Solution: Docker creates containers that ensure Node.js applications run in the same environment everywhere. This means the entire team can use the same Node.js version and dependencies by running the same container.

2.2.2.2 Simplify the configuration process

Problem: Configuring a Node.js application can be complex due to system files, environment variables, and dependencies.

Docker Solution: Dockerfile and Docker Compose allow you to define all configurations and services clearly. Team members can start everything they need with just one command, instead of installing each component manually.

2.2.2.3 Easy CI/CD Integration

Problem: Setting up a CI/CD pipeline for deploying Node.js applications can be tricky due to environment differences.

Docker Solution: Docker easily integrates with CI/CD tools like Jenkins and GitLab CI, allowing you to build and test containers in the pipeline before deploying them. This reduces errors related to different environments.

2.2.2.4 Increased Portability and Scalability

Problem: Moving a Node.js application from development to production can cause errors due to environment differences.

Docker Solution: Docker containers can run on any system that supports Docker, making it easy to deploy applications on cloud platforms like AWS or Azure. Containers can also be easily replicated and scaled when needed.

2.2.2.5 Improve Deployment Speed

Problem: Traditional Node.js deployments can be slow, especially when reconfiguring environments or reinstalling dependencies.

Docker Solution: Docker caches each layer during the image build process, which speeds up future builds. Only the layers that changed need to be rebuilt, making deployments faster.

2.2.2.6 Isolate application components

Problem: Node.js applications often require multiple services (like databases), making management complex and prone to conflicts.

Docker Solution: Each service can run in its own container, ensuring they are isolated from each other. Docker Compose helps manage starting, stopping, and connecting these services easily.

2.3 Covering the benefits and challenges of containerization in web development

2.3.1 Benefits of containerization in web development

2.3.1.1 Platform Independence: "Build it once, run it anywhere"

Containers are portable, meaning you can build an application once and run it anywhere, like on local desktops, physical or virtual servers, and in the cloud. This makes development faster and easier to move between different environments.

2.3.1.2 Resource efficiency and density

Containers use fewer resources than virtual machines because they don’t need their own operating system. While virtual machines can be several gigabytes, containers are usually only tens of megabytes. This means you can run more containers on a single server, reducing costs.

2.3.1.3 Efficient isolation and resource sharing

Containers run on the same host but don’t interact with each other. If one application crashes, others continue to run without issues. This isolation also improves security because if one application is compromised, it doesn’t affect others.

2.3.1.4 Speed: Start, instantiate, clone, or destroy containers in seconds

Containers start in less than a second since they don’t need to boot an operating system. This allows for quick creation, cloning, or destruction of containers, speeding up the development process and time to market.

2.3.1.5 Scaling Up

Containers allow for easy horizontal scaling, meaning you can add more identical containers as needed. This helps save resources and increase return on investment, which many companies, like Google and Netflix, have been using for years.

2.3.1.6 Simple to operate

Unlike virtual machines that need their own operating systems, containers run application processes separately from the host OS. This simplifies server management and makes applying updates and security patches faster.

2.3.1.7 Improve developer productivity and development pipeline

Containerization leads to a more efficient development process. Containers ensure consistent application performance, making testing and debugging easier. Developers can quickly modify configurations, create new containers, and remove old ones without downtime. Tools like Docker also allow for version control and easy sharing among team members.

2.3.1.8 Container coordination

Container technology is gaining popularity, especially among startups and small to medium-sized businesses, as it offers many advantages in managing applications.

2.3.2 The Challenge of Containerization in Web Development

Complexity: Managing multiple containers and orchestrating them can become quite complex, especially as the number of containers grows.

Storage: Persistent storage solutions that work well with containers can be tricky to implement and manage.

Networking: Ensuring reliable and secure communication between containers often requires additional configuration.

Performance Overhead: Although lightweight, containers do introduce some overhead compared to running applications directly on the host OS.

2.3.2.1 Complexity

So, when we talk about managing a large number of containers, think of it like trying to juggle more and more balls simultaneously. As the number of containers increases, so do the challenges:

1. Orchestration: Ensuring that containers start, stop, and scale correctly requires advanced orchestration tools like Kubernetes, which add layers of complexity.
2. Networking: Maintaining stable, secure communication between a growing number of containers can become increasingly intricate.
3. Monitoring: Keeping track of the health and performance of many containers demands robust monitoring and logging systems.
4. Configuration Management: Each container might need different configurations, and keeping these organized can be a headache.

Balancing all these elements without losing control can be a real challenge, requiring both good tools and skilled engineers. Complexity scales up with the number of containers, making efficient management crucial. How does that resonate with your experience?

2.3.2.2 Storage

When it comes to persistent storage in containerized environments, the challenges can stack up:

1. Data Persistence: By design, containers are stateless and ephemeral, meaning their data doesn't persist after they're destroyed. Persistent storage solutions are needed to ensure data is saved and accessible.
2. Volume Management: Managing storage volumes that can be attached and detached from containers without causing data loss or corruption requires careful planning and orchestration.
3. Storage Drivers: Different storage drivers offer various features, and selecting the right one can be crucial but tricky.
4. Backup and Recovery: Ensuring reliable backup and recovery processes for container storage can be more complex than traditional systems.
5. Performance: Maintaining high performance while using shared storage across multiple containers can be challenging, especially with I/O intensive applications.

Navigating these issues calls for a robust understanding of both storage technologies and container management best practices. Getting it right means containers can effectively handle data without hiccups.

2.3.2.3 Networking

Network stability and security become intricate as the number of containers grows:

1. Dynamic IP Addresses: Containers often use dynamic IPs, making it hard to keep track of their locations.
2. Service Discovery: Ensuring containers can locate each other requires robust service discovery mechanisms.
3. Network Policies: Managing and enforcing network policies to control traffic flow between containers can get complex.
4. Load Balancing: Distributing network traffic evenly across containers without overloading any single one is crucial.
5. Security: Ensuring data security and preventing unauthorized access between containers demands strong network isolation.

Keeping these connections smooth and secure requires sophisticated tools and strategies to handle the growing complexity. It’s a delicate balancing act.

2.3.2.4 Performance Overhead

Digging into the performance overhead issue of containers:

1. Resource Allocation: Containers share the host OS kernel, but each container still requires its own set of resources (CPU, memory, etc.), adding some overhead.
2. Abstraction Layers: Containers add layers of abstraction between the application and the hardware, which can introduce slight performance hits compared to bare metal.
3. Storage I/O: Shared storage solutions for containers might not match the performance of direct-attached storage, leading to latency.
4. Network Overhead: Networking between containers adds a bit of overhead compared to direct communication on the host OS.

So, while containers are efficient, there's an inevitable trade-off in performance compared to running applications directly on the host OS. Balancing this requires understanding your workload and optimizing accordingly.

CHAPTER 3: PROJECT OVERVIEW

sdfsdf

REFERENCES

1. What is Docker? | Docker Docs
2. Docker for Web Developers: Starting with the Basics | Docker
3. What is a Container? | Docker
4. Dockerfile reference | Docker Docs
5. Docker Compose | Docker Docs
6. What is container orchestration?
7. Tổng hợp kiến thức về Docker: Khám phá Docker (phần 1)
8. 7 lợi ích hàng đầu của việc sử dụng container

sdfa