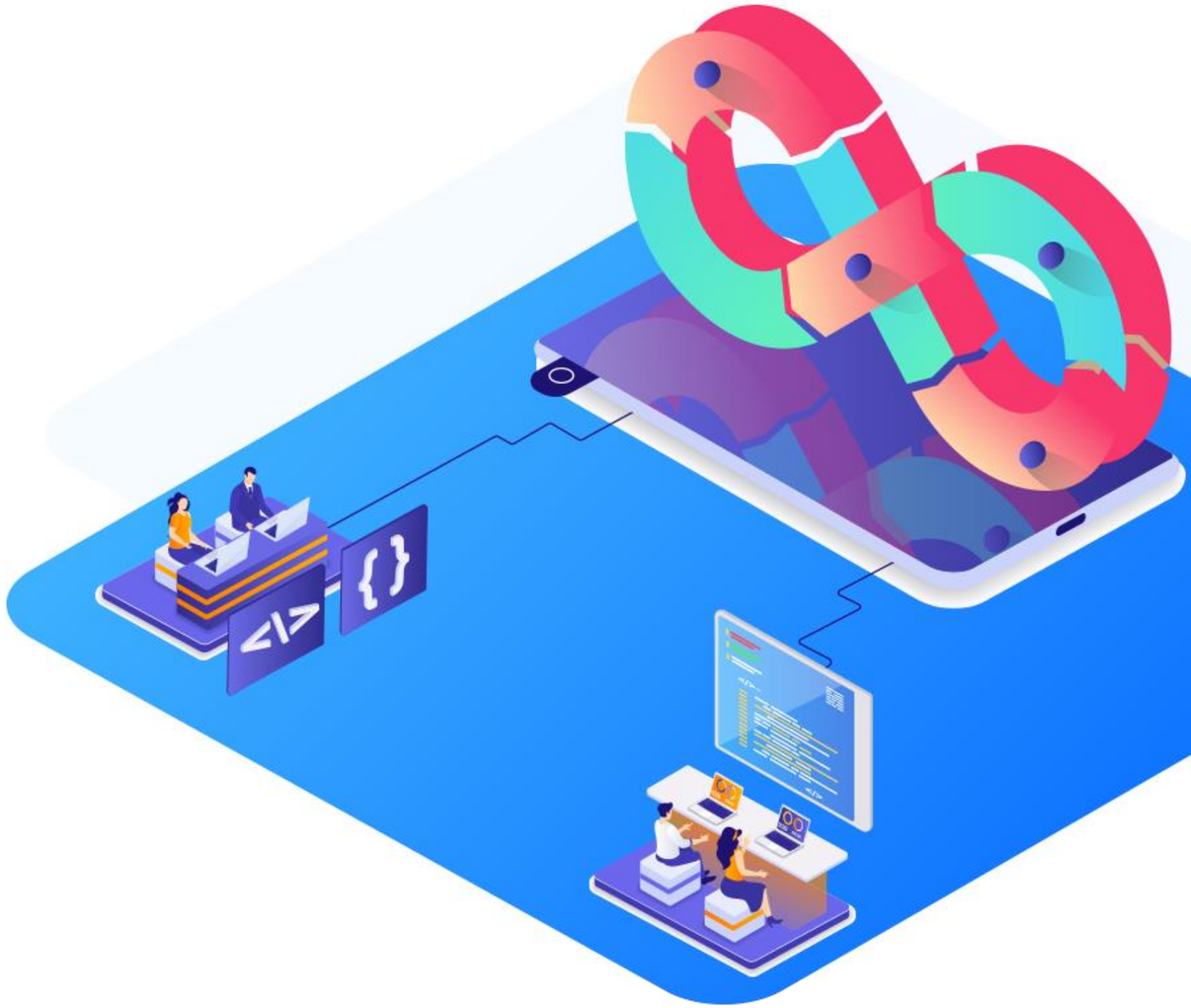
Containerization with Docker



Microservices Architecture in Docker

Learning Objectives

By the end of this lesson, you will be able to:

- Deploy a microservice using Docker to ensure proper network configuration and resource allocation
- Configure microservices network using Docker Compose for establishing secure communication between services
- Implement Docker health checks to monitor and identify performance bottlenecks within the microservices architecture
- Deploy application stack on a Swarm cluster using microservices to demonstrate distributed deployment

Introduction to Microservices

What Are Microservices?

Microservices in Docker refer to a software architecture approach where applications are built as a collection of independently deployable services, each running in its container.

They offer a scalable, resilient, and efficient approach to building and deploying modern applications, promoting agility and innovation in software development.

Microservices: Benefits

Using microservices helps users achieve:

Improved scalability

Simplified deployment

Enhanced fault isolation

Improved productivity

Better resiliency

Microservices: Benefits

Improved scalability

Microservices enable horizontal scalability by allowing independent scaling of individual components, enhancing resource utilization.

Simplified deployment

Independent deployment of services facilitates continuous delivery, enabling faster and more frequent releases.

Enhanced fault isolation

Isolating services minimizes the impact of failures, ensuring that issues in one service do not affect the entire system.

Microservices: Benefits

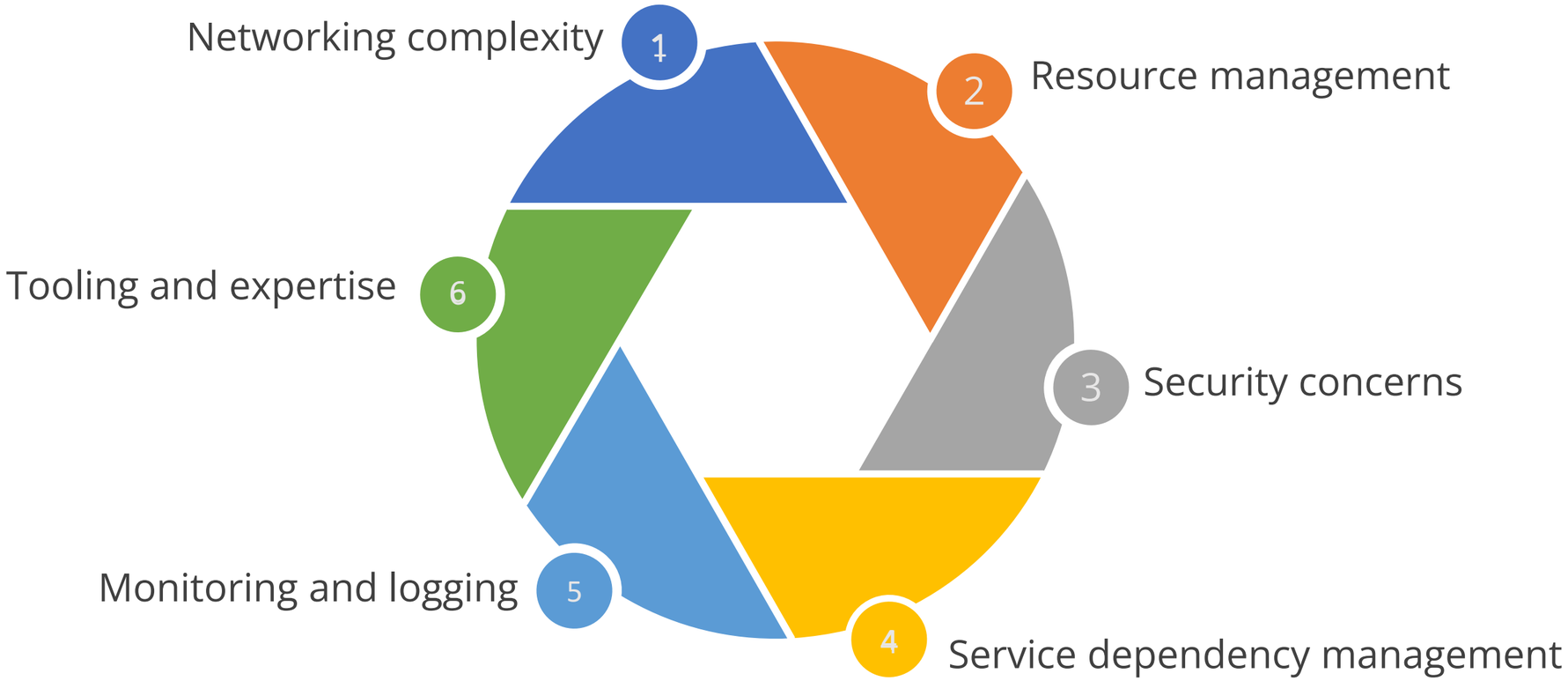
Improved productivity

Microservices foster team autonomy and faster development cycles, leading to increased productivity.

Better resiliency

With the ability to patch, update, and restart individual components, microservices enhance the resilience of the overall system.

Challenges in Deploying Microservices Architecture



Use Cases of Microservices in Docker

E-commerce

Use case: Microservices in Docker allow the platform to scale individual components like the payment gateway, inventory management, or user authentication independently, ensuring seamless performance during peak times.

Scenario: An online shopping site must handle fluctuating traffic, particularly during sales or holidays.

Banking

Use case: Microservices in Docker enable the bank to break down its application into smaller, more manageable services like account management, transaction processing, and fraud detection. Each service can be updated, tested, and deployed independently, reducing the risk of downtime.

Scenario: A bank wants to modernize its monolithic application to improve agility and reduce downtime during updates.

Use Cases of Microservices in Docker

Healthcare

Use case: Microservices in Docker allow for the isolation of sensitive data and functionalities into separate services, making it easier to maintain compliance and manage complex workflows across different departments.

Scenario: A healthcare provider needs to manage patient records, appointment scheduling, billing, and more, all while complying with strict data privacy regulations.

Social media

Use case: Microservices in Docker can be used to separate these features into distinct services, each capable of scaling independently. This architecture ensures that the platform remains responsive even as the user base grows.

Scenario: A social media platform needs to handle real-time interactions like messaging, notifications, and newsfeed updates for millions of users.

Quick Check

Your team is building an e-commerce application using a microservices architecture deployed in Docker. The team is discussing the deployment strategy and potential challenges. Which of the following scenarios best illustrates a common challenge you might face when deploying a microservices architecture?

A. Difficulty in scaling the application to handle increased traffic during sales events

B. Ensuring that a failure in the payment service does not impact the inventory management service

C. Managing the complexity of inter-service communication and monitoring across multiple microservices

D. Rolling out updates to the entire application without affecting any running services

Microservices with Containers

Microservices Architecture Patterns

Microservices architecture patterns in Docker leverage the containerization technology to design scalable and modular applications, including:

Service registry patterns

Observability patterns

API gateway patterns

Microservices Architecture Patterns

Service registry patterns

Docker containers can register their services dynamically in a service registry, enabling service discovery and communication among microservices.

Observability patterns

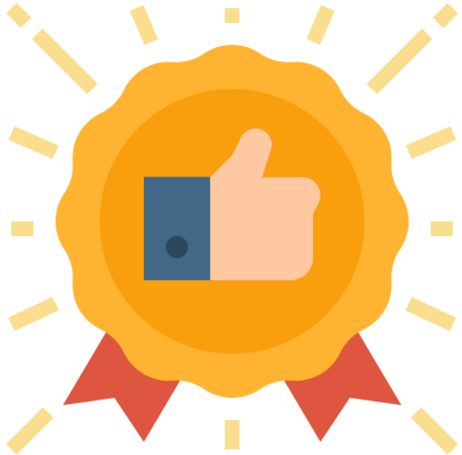
Docker containers enable the implementation of observability patterns such as logging, monitoring, and tracing to gain insights into microservices' behavior and performance.

API gateway patterns

Dockerized microservices, when paired with an API gateway container, offer a centralized client entry point and support essential features such as authentication and rate limiting.

Microservices Architecture Patterns: Best Practices

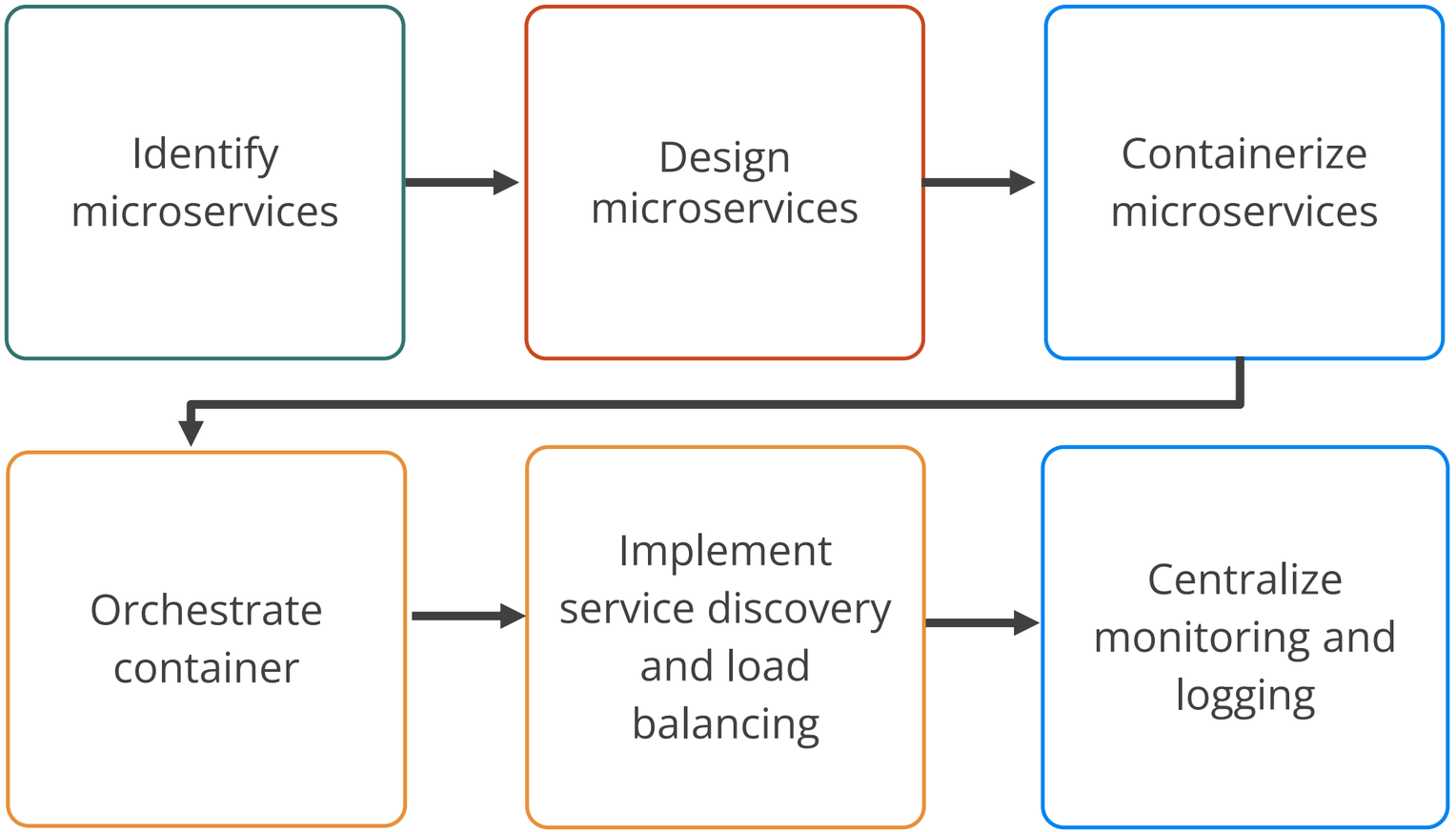
Here are a few best practices for working with microservices architecture patterns:



- Use the principle of one microservice with one API
- Write detailed testing parameters for each microservice
- Automate all aspects of the microservice environment

Designing Microservices with Docker

The following are the steps to design microservices with Docker:



Dockerfile for NodeJS Application: Example

Scenario: You are tasked to create a microservice using Dockerfile for a NodeJS application containing order and user services.

Example:

```
# Use a base image
FROM node:14

# Set the working directory
WORKDIR /app

# Copy the package.json and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port and start the application
EXPOSE 3000
CMD ["npm", "start"]
```

Orchestrate Dockerfile for NodeJS Application: Example

Here is a Docker compose file to orchestrate the Dockerfile made for NodeJS application:

Example:

```
version: '3'
services:
  user-service:
    build: ./user-service
    ports:
      - "5000:5000"
  order-service:
    build: ./order-service
    ports:
      - "5001:5001"
```

Assisted Practice

Creating a Simple Microservices Architecture Design **Duration: 10 Min.**

Problem statement:

You are tasked with creating a simple microservices architecture using Docker Compose to demonstrate the interaction between a server and client in a containerized environment.

Outcome:

By completing this demo, you will successfully implement a microservices architecture that facilitates interaction between a server and client using Docker Compose, showcasing effective container communication.

Note: Refer to the demo document for detailed steps:
01_Creating_a_Simple_Microservices_Architecture_Design

Assisted Practice: Guidelines

Steps to be followed:

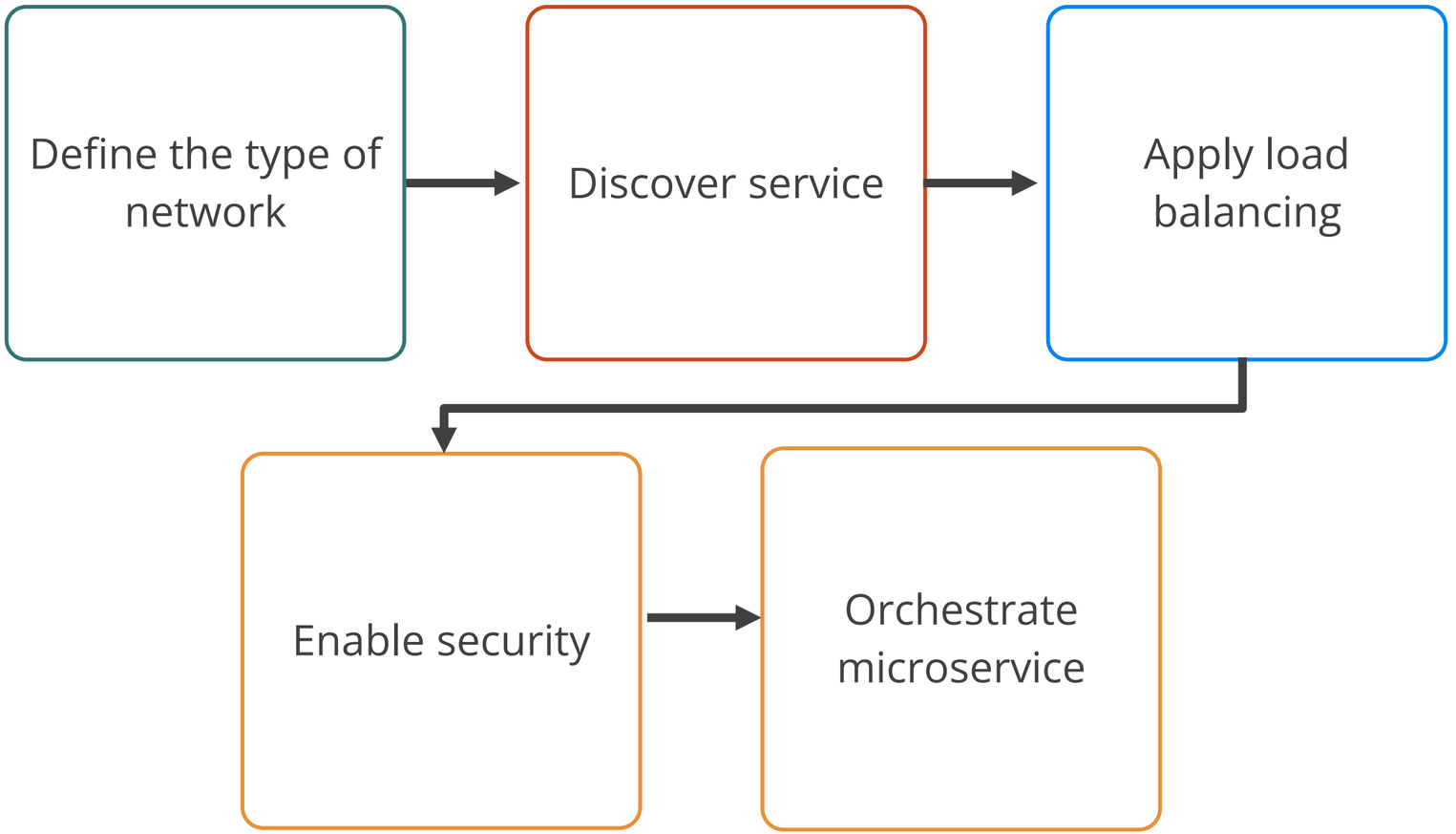
1. Create the server
2. Create the client
3. Create a Docker compose file
4. Install and run Docker Compose

ASSISTED PRACTICE

Docker Networking for Microservices

Networking in Microservices

Networking in a microservices architecture within Docker enables communication between different services. Here are the steps to be followed to configure the network between multiple microservices:



Compose File for NodeJS Application: Example

Scenario: You are tasked to create a Docker compose file to orchestrate microservice-based NodeJS application containing frontend and backend.

Example:

```
version: '3'
services:
  frontend:
    image: frontend-image
    networks:
      - frontend-network
      - backend-network

  backend:
    image: backend-image
    networks:
      - backend-network
      - db-network

  db:
    image: db-image
    networks:
      - db-network

networks:
  frontend-network:
  backend-network:
  db-network:
```

Microservices Using Swarm

Deploying a microservice using Docker Swarm involves creating a Swarm cluster, defining services, and deploying them across the cluster.

Here are the steps to implement microservice through Docker Swarm:

1. Initialize Docker Swarm
2. Join Worker nodes (optional)
3. Create Docker compose file to define services
4. Deploy the stack
5. Scale microservices

Microservices Using Swarm

Initialize Docker Swarm

Run **docker swarm init** to initialize Swarm. In case of failure, it is always recommended to reinstall or reinitialize Swarm.

Join worker nodes

Use **docker swarm join** on worker nodes to add them to the Swarm cluster

Create Docker compose file

Write a **docker-compose.yml** file to specify microservices, networks, and volumes associated with the application

Microservices Using Swarm

Deploy the stack

Deploy the defined services across the Swarm cluster using **docker stack deploy -c <compose-file> <stack-name>**

Scale microservices

Adjust the number of service replicas with **docker service scale <service-name>=<replica-count>** to manage load

Compose File for NodeJS Application with Swarm: Example

Scenario: You are tasked to create a Docker compose file to orchestrate a microservice-based NodeJS application containing frontend and backend with Swarm.

Example:

```
version: '3'
services:
  frontend:
    image: frontend-image
    ports:
      - "8080:80"
    networks:
      - frontend-network
      - backend-network
    deploy:
      replicas: 3
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure

  backend:
    image: backend-image
    networks:
      - backend-network
      - db-network
    deploy:
      replicas: 2
      restart_policy:
        condition: on-failur
```

Compose File for NodeJS Application with Swarm: Example

Scenario: You are tasked to create a Docker compose file to orchestrate a microservice-based NodeJS application containing frontend and backend with Swarm.

Example:

```
  db:
    image: db-image
    volumes:
      - db-data:/var/lib/db
    networks:
      - db-network
    deploy:
      replicas: 1

networks:
  frontend-network:
  backend-network:
  db-network:

volumes:
  db-data:
```

Scaling Microservices Using Swarm

Scaling microservices in Docker using Swarm involves increasing or decreasing the number of container instances running a particular service to manage load, improve availability, and ensure fault tolerance.

There are two ways to scale microservice:

- Through compose file
- Through **scale** command

Example:

```
//Scaling through Docker Compose file

version: '3'
services:
  frontend:
    image: frontend-image
    deploy:
      replicas: 3  # This line specifies that 3 replicas of the
frontend service should be created.
  backend:
    image: backend-image
    deploy:
      replicas: 2  # The backend service will have 2 replicas.
```

Scaling Microservices Using Swarm

Here is a command to scale microservice through **scale** command:

Command:

```
docker service scale my_stack_frontend=5
```

Assisted Practice

Configuring Docker Networking for a Microservices Environment **Duration: 10 Min.**

Problem statement:

You are tasked with configuring Docker networking to facilitate seamless communication and scalability among microservices in a Docker container environment.

Outcome:

By completing this demo, you will successfully configure Docker networking, demonstrating effective communication and scalability among microservices within Docker containers.

Note: Refer to the demo document for detailed steps:
02_Configuring_Docker_Networking_for_a_Microservices_Environment

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Set up a simple HTTP server with Docker
2. Set up a client script with Docker
3. Configure Docker networking for microservices

ASSISTED PRACTICE

Assisted Practice

Scaling Microservices with Docker Swarm **Duration: 10 Min.**

Problem statement:

You are tasked with scaling microservices using Docker Swarm, which involves initializing the swarm, defining services in Docker Compose, deploying them using Docker Stack, and managing their scalability and updates.

Outcome:

By completing this demo, you will successfully scale and manage microservices using Docker Swarm, ensuring high availability and streamlined updates across your services architecture.

Note: Refer to the demo document for detailed steps:
03_Scaling_Microservices_with_Docker_Swarm

Assisted Practice: Guidelines

Steps to be followed:

1. Initialize Docker Swarm
2. Define microservices in Docker Compose
3. Deploy microservices using the Docker Stack
4. Monitor and manage your microservices

ASSISTED PRACTICE

Quick Check

You are managing a microservices-based application deployed using Docker Swarm. During high traffic, one of the worker nodes becomes unresponsive. What is the most appropriate first step to maintain the system's functionality?

A. Use the command Docker Swarm leave on the unresponsive node to remove it from the Swarm

B. Use Docker node ls to check the status of all nodes in the Swarm and confirm the issue

C. Reinstall Docker Swarm immediately on the unresponsive node to try to bring it back online

D. Restart the services running on the unresponsive node manually to recover the applications

Monitoring Microservices

Monitoring Microservices

Monitoring microservices helps maintain high availability and quickly diagnose issues in the Docker-based microservices architecture. It can be done through **Docker Health Checks.**

Docker Health Check can be added in the compose file along with the microservice definition.

Importance of Monitoring

Here are some of the major outcomes that are achieved by monitoring microservices:

High availability: Monitoring proactively detects issues and maintains service uptime, ensuring microservices remain accessible and scale when the workload is high.

Performance optimization: Monitoring tracks resource usage (CPU, memory) and service performance metrics, helping optimize resource allocation.

Security and compliance: Continuous monitoring detects anomalies or unauthorized access, bolstering system security.

Key Aspects of Docker Health Check

Health Command

- It is a command that Docker runs inside the container at specified intervals to check the container's health.

- **HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \ CMD curl -f http://localhost:3000/health || exit 1**

Health Status

- **healthy:** The command ran successfully, indicating the container is operating as expected.
- **unhealthy:** The command failed, indicating a potential issue with the container.
- **starting:** The container is in the process of starting up, and the health check has not completed yet.

Adding Health Check in Dockerfile

Here is an example of a Dockerfile with **Health Check** added at the end of the execution:

Example:

```
FROM node:14

# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install

# Copy the rest of the application
COPY . .

# Expose the port the app runs on
EXPOSE 3000

# Command to run the application
CMD ["node", "app.js"]

# Health check to see if the app is responding
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --
retries=3 \
  CMD curl -f http://localhost:3000/health || exit 1
```

Adding Health Check in Docker Compose

Here is an example of a Docker compose file with **Health Check** added at the end of the execution:

Example:

```
version: '3.8'

services:
  web:
    image: my-node-app
    ports:
      - "3000:3000"
    healthcheck:
      test: ["CMD", "curl", "-f",
"http://localhost:3000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 5s
```

Assisted Practice

Applying Health Checks and Monitoring to Microservices **Duration: 10 Min.**

Problem statement:

You are tasked with implementing health checks and monitoring for microservices using Docker, Docker Compose, and Prometheus, enhancing the reliability and observability of the services.

Outcome:

By completing this demo, you will successfully apply health checks and set up monitoring for microservices, demonstrating improved service reliability and observability through Docker and Prometheus.

Note: Refer to the demo document for detailed steps:
04_Applying Health Checks and Monitoring to Microservices

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create microservices
2. Create a requirements file for dependencies
3. Create a Dockerfile for each microservice
4. Create a Docker compose file and run the setup

ASSISTED PRACTICE

Securing Microservices

Microservice Security

Securing microservices in Docker requires a multi-layered approach that includes network isolation, encrypted communication, robust authentication, and vigilant monitoring.

A microservice can be secured using network isolation, TLS encryption, and secure image and secrets management.

Securing Microservice Using Compose

Scenario: You have a Node.js microservice that interacts with a MySQL database. You aim to securely deploy this setup in Docker, focusing on encrypting communication, securing Docker images, and effectively managing sensitive information.

Here is an example showing network isolation, secrets management using Docker compose:

Example:

```
version: '3.8'

services:
  app:
    image: my-secure-node-app:latest
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "443:443"
    networks:
      - app-network
    environment:
      - NODE_ENV=production
      - DATABASE_URL=mysql://db:3306/mydatabase
    depends_on:
      - db
    secrets:
      - db_password
    command: sh -c "node server.js"
```

Securing Microservice Using Compose

Example:

```
  db:
    image: mysql:8.0
    networks:
      - db-network
    environment:
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db_password
      - MYSQL_DATABASE=mydatabase
    secrets:
      - db_password
    volumes:
      - db_data:/var/lib/mysql

networks:
  app-network:
  db-network:

volumes:
  db_data:

secrets:
  db_password:
    file: ./secrets/db_password.txt
```

Secure Image Management

In this method, the Docker image is scanned for vulnerabilities before building and deployment. Here is a command to check the same:

Example:

```
docker build -t my-secure-node-app:latest .
docker scan my-secure-node-app:latest
```

Assisted Practice

Securing Microservices Using Docker **Duration: 10 Min.**

Problem statement:

You are tasked with demonstrating the secure setup and management of microservices using Docker, focusing on implementing best practices for secrets management, resource limitations, and security configurations.

Outcome:

By completing this demo, you will successfully establish and secure microservices using Docker, showcasing enhanced security practices and efficient deployment in a containerized environment.

Note: Refer to the demo document for detailed steps:
05_Securing_Microservices_Using_Docker

Assisted Practice: Guidelines

Steps to be followed:

1. Create microservices
2. Create a requirements file for dependencies
3. Create a Dockerfile for each microservice
4. Create a Docker compose file and run the setup

ASSISTED PRACTICE

Quick Check

You are tasked with deploying a critical microservice using Docker. Which of the following practices should you implement to ensure the security of your Docker images?

A. Use any available base image to save time

B. Regularly scan images for vulnerabilities before building the images

C. Disable image signing to simplify the deployment process

D. Include all possible dependencies to avoid future updates

Key Takeaways

- Scaling microservices in Docker using Swarm involves increasing or decreasing the number of container instances running a particular service to manage load, improve availability, and ensure fault tolerance.
- Microservices architecture patterns in Docker leverage containerization technology to design scalable and modular applications.
- Docker health check command runs inside the container at specified intervals to check the container's health.
- Monitoring tracks resource usage (CPU, memory) and service performance metrics, helping optimize resource allocation.
- In secure image scanning, the Docker image is scanned for vulnerabilities before building and deployment.

Implementing Microservice Architecture with Docker

Duration: 25 min.

Project agenda: To develop and deploy a basic microservices architecture using Docker, which involves creating a web server and a client service. The focus is on isolating each component within its own Docker container to streamline development, testing, and deployment processes.

Description: As a DevOps engineer in a fast-paced startup, your current project focuses on establishing a streamlined microservices architecture for a specific application that serves static content via a web server and a corresponding client, both encapsulated within Docker containers. This architecture supports rapid development, deployment, and scaling and enhances system robustness by isolating each service component. By leveraging Docker and Docker Compose, your task is to ensure that all environments, from development to production, are consistent and that changes can be deployed swiftly and reliably. This setup aims to provide a foundational model for future expansions and service integrations.

LESSON-END PROJECT

Implementing Microservice Architecture with Docker

Duration: 25 min.

Perform the following:

1. Set up project directories and server
2. Configure the client and server Dockerfiles
3. Create and configure Docker Compose
4. Build and deploy with Docker Compose

Expected deliverables: A microservices architecture using Docker, demonstrating the ability to manage and run separate client and server containers with Docker Compose for efficient, scalable service deployment.

LESSON-END PROJECT

Thank You