DEVOPS PRINCIPLES AND PRACTICES FOR QA PROFESSIONALS

Date: Oct-2023
By: Quy (Christian) P. TRAN
Duration: 15 (minutes)

TABLE OF CONTENT

TABLE OF CONTENT

DevOps for QA: Introduction

- Welcome and introduction to the topic of DevOps for QA professionals.
- Briefly explain the importance of DevOps in software development and testing.

What is DevOps?

- Definition of DevOps as the combination of development (Dev) and operations (Ops) teams.
- Explain how it improves collaboration, efficiency, and quality in software development.

Benefits of DevOps for QA

- Faster software development and deployment cycles.
- Increased collaboration and communication between teams.
- Continuous integration and delivery to ensure constant feedback and improvement.
- Improved software quality and customer satisfaction.

Continuous Integration and Continuous Deployment (CI/CD) in DevOps

- Explain the significance of CI and CD in DevOps.
- CI ensures regular integration and testing of code changes.
- CD automates the deployment process for faster software updates.
- Mention popular tools like Jenkins, GitLab CI/CD.
- Showcase real-world examples of successful CI/CD implementations.

DevOps Tools and Technologies

Introduction to popular DevOps tools and technologies:

1. Version control: Git for code collaboration and management.
2. Continuous Integration/Continuous Delivery (CI/CD) platforms: Jenkins, GitLab CI/CD, etc.
3. Automation and scripting: Bash, Python, etc.
4. Infrastructure provisioning and management: Docker, Kubernetes, etc.

Continuous Testing in DevOps

- Explanation of the importance of integrating testing throughout the development lifecycle.
- Introduction to concepts like shift-left testing, test automation, and continuous feedback.

Testing in Containerized Environments

- Overview of containerization technologies like Docker.
- Discuss how testing in containerized environments can improve portability and scalability.

Testing in Cloud Environments (AWS)

- Understanding the nuances of testing in cloud platforms like AWS. (deploying the tests, monitoring, and reporting)

- Highlight the importance of testing for scalability, performance, and security in cloud-based applications.

Performance Testing in DevOps

- Overview of performance testing in DevOps.
- Introduction to tools like JMeter, or Locust for conducting stress, load, and performance tests.

Required Skills and Knowledge

- List of essential skills and knowledge for QA professionals in a DevOps environment.
  - Proficiency in version control systems (Git).
  - Familiarity with Linux/Unix operating systems.
  - Scripting skills (Bash, Python).
  - Understanding of agile methodologies.
  - Strong communication and collaboration abilities.

Sessions and Integration Topics

- Docker: Exploring containerization for efficient deployment and testing. (1 session)
- Testing framework integration in containers and cloud environments.
- GitHub Actions & GitLab CI: Implementing CI/CD processes. (1 + 1 sessions)
- AWS: Understanding cloud services for scalable and reliable testing environments. (1 session)
- Kubernetes: Orchestration and management of containerized applications. (1 session)

* Note:

- Feel free to customize the list of sessions based on your specific needs or add any other relevant topics as desired.
- The number of sessions can be adjusted based on the specific requirements and depth of coverage desired for each topic.
- 1 Session = 1 Hour

References

- Shift-left testing: https://viblo.asia/p/shift-left-testing-bi-quyet-cho-phan-mem-thanh-cong-oOVlY14zl8W
- BrowserStack: https://www.browserstack.com/guide/role-of-qa-in-devops
- ChatGPT

- Docker documentation: https://docs.docker.com/
- Amazon Web Services (AWS) documentation: https://aws.amazon.com/documentation/
- Kubernetes documentation: https://kubernetes.io/docs/home/
- GitHub Actions documentation: https://docs.github.com/en/actions
- GitLab CI/CD documentation: https://docs.gitlab.com/ee/ci/

Questions and Discussion

INTRODUCTION TO DOCKER FOR SOFTWARE DEVELOPMENT AND DEPLOYMENT

Date: Oct-2023
By: Quy (Christian) P. TRAN
Duration: 60 (minutes)

TABLE OF CONTENT

TABLE OF CONTENT

Introduction

- Briefly introduce the training topic: Docker for software development and deployment.
- Highlight the benefits of using Docker for application portability, scalability, and reproducibility.

What is Docker?

- Define Docker as an open-source platform for containerization.
- Explain the concept of containers and their advantages compared to traditional virtual machines.

Docker vs. Virtual Machines (VMs)

- Docker: Lightweight containers, shared host OS, efficient resource usage, faster startup.
- VMs: Complete virtualization, separate guest OS, heavier resource requirements, slower startup.
- Docker provides portability and scalability across different environments.
- Docker offers better performance and resource utilization compared to VMs.
- Docker's containerization simplifies deployment and enhances efficiency.

Installation

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

Key Docker Components

- Introduce the main components of Docker: Docker Engine, Images, and Containers.
- Explain the role of Docker Engine in running and managing containers.

Docker Images

- Docker Image: Lightweight, self-contained package with everything needed to run software.
- Similar to a blueprint, it provides a consistent and reproducible environment.
- Contains base OS, application code, dependencies, and configuration files.
- Version-controlled and built using Dockerfiles.
- Enables portability, consistency, and ease of deployment for Docker Containers.

Docker Containers

- Docker Containers: Running instances of Docker Images.
- Containers provide isolated environments for running applications.
- Lightweight, fast to start, and efficient in resource usage.
- Containers ensure application portability and consistency.
- Simplify software development and deployment.
- Easily managed, scaled, and orchestrated using tools like Kubernetes.

Docker Operations

- Pull: Download Docker Images from a registry.
- Build: Create a Docker Image from a Dockerfile.
- Run: Start a Docker Container from an Image.
- Push: Upload Docker Images to a registry.
- Exec: Run commands inside a running container.
- Stop: Gracefully stop a running container.
- Remove: Delete unused containers or images.

Docker Pull

- Docker Pull: Download Docker Images from a registry.
- Command: docker pull <image-name>:<tag>
- Function: Fetches pre-built Docker Images from a registry.
- Example: docker pull ubuntu:latest

Docker Build

- Docker Build: Create a Docker Image from a Dockerfile.
- Command: docker build <flags> -t <image-name>:<tag> <path-to-Dockerfile>
- Function: Builds a Docker Image based on the instructions in a Dockerfile.
- Example: docker build -f Dockerfile.prod -t myapp:latest .

Flags:

  - -f <path/to/Dockerfile>: Specifies the path to the Dockerfile.
  - -t <image-name>:<tag>: Tags the built image with a name and tag.
  - --no-cache: Builds the image without using the cache.
  - --build-arg <key=value>: Allows passing build-time variables to the Dockerfile.
  - --pull: Forces a fresh pull of the base image during the build process.
  - --target <stage>: Specifies a specific build stage to build.
-

Docker Run

- Docker Run: Start a Docker Container from an Image.
- Command: docker run <flags> <image-name>:<tag>
- Function: Creates and starts a container from a Docker Image.
- Example: docker run -d -p 8080:80 --name mycontainer myapp:latest
- Flags:
  - -d: Runs the container in the background (detached mode).
  - -p 8080:80: Maps port 80 in the container to port 8080 on the host system.
  - --name mycontainer: Assigns a custom name to the container.
  - --rm: Automatically removes the container when it stops running.
  - -e <env-variable=value>: Sets environment variables within the container.
  - -v <host-path>:<container-path>: Mounts a volume from the host to the container.
  - --network <network-name>: Connects the container to a specific network.
-

Docker Exec

- Docker Exec: Run commands inside a running container.
- Command: docker exec <flags> <container-id/name> <command>
- Function: Executes a command within a running container.
- Example: docker exec -it mycontainer bash

Flags:

  - -i: Attach stdin for interactive input.
  - -t: Allocate a pseudo-TTY for the command.
  - -d: Detach from the container's console without killing the command.
  - -e <env-variable=value>: Sets environment variables within the container.
  - --user <username>: Specifies the username or UID for the command.
  - --workdir <directory>: Sets the working directory for the command.
-

Docker Push

- Docker Push
- Docker Push: Upload Docker Images to a registry.
- Command: docker push <registry>/<image-name>:<tag>
- Function: Uploads a Docker Image to a registry for sharing and distribution.
- Example: docker push myregistry/myapp:latest

Docker Remove Container

- Docker Remove Container: Delete a container from your system.
- Command: docker rm <container-id/name>
- Function: Removes a specific container from your system.
- Example: docker rm mycontainer
- Additional Options:
  - docker rm -f <container-id/name>: Forcefully removes a running container.
  - docker rm -v <container-id/name>: Removes a container and its associated volumes.
-

Docker Remove Image

- Docker Remove Image: Delete an image from your system.
- Command: docker rmi <image-id/name>
- Function: Removes a specific image from your system.
- Example: docker rmi myimage:latest
- Additional Options:
  - docker rmi -f <image-id/name>: Forcefully removes an image, even if it's being used by containers.
  - docker rmi $(docker images -a -q): Removes all images on your system.
-

Docker Networking (1)

- Docker Networking: Connecting Docker containers and enabling communication.
- Default Networking:
  - Docker creates a default bridge network for containers to communicate.
  - Each container gets its own IP address and can access other containers using that IP.
- Docker Network Commands:
  - docker network create: Creates a new user-defined network.
  - docker network connect: Connects a container to a network.
  - docker network disconnect: Disconnects a container from a network.
-

Docker Networking (2)

- Types of Networking:
  - Bridge Network: Default network allowing containers on the same host to communicate.
  - Host Network: Container uses the host's network stack directly.
  - Overlay Network: Connects containers across different Docker hosts.
  - Macvlan Network: Assigns a MAC address to each container, making it appear as a physical device on the network.

Docker Networking (3)

- Types of Networking:

Driver/ Features	Bridge	User defined bridge	Host	Overlay	Macvlan/ipvlan
Connectivity	Same host	Same host	Same host	Multi-host	Multi-host
Service Discovery and DNS	Using “links”. DNS using /etc/hosts	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine	Done using DNS server in Docker engine
External connectivity	NAT	NAT	Use Host gateway	No external connectivity	Uses underlay gateway
Namespace	Separate	Separate	Same as host	Separate	Separate
Swarm mode [1]	No support yet	No support yet	No support yet	Supported	No support yet
Encapsulation	No double encap	No double encap	No double encap	Double encap using Vxlan	No double encap
Application	North, South external access	North, South external access	Need full networking control, isolation not needed	Container connectivity across hosts	Containers needing direct underlay networking

Docker Volumes

- Docker Volumes: Persisting and sharing data.
- Types: Bind Mounts, Named Volumes, tmpfs Mounts.
- Commands: docker volume create, docker volume ls, docker volume inspect.
- Examples:
  - Bind Mount: docker run -v /host/path:/container/path image:tag
  - Named Volume: docker run -v volname:/container/path image:tag
  - tmpfs Mount: docker run -v /container/path --tmpfs /container/path image:tag
- Benefits: Data persistence, inter-container data sharing, easy backup/restore, improved performance.
-

Docker Compose

- Docker Compose: Define and manage multi-container Docker applications.
- Compose File (docker-compose.yml):
  - YAML file format to define services, networks, and volumes.
  - Describes relationships between containers and their configurations.
- Key Concepts:
  - Services: Containers defined in the Compose file.
  - Networks: Networks to connect services together.
  - Volumes: Persistent data storage for services.
- Docker Compose Commands:
  - docker-compose up: Create and start all services in the Compose file.
  - docker-compose down: Stop and remove all services defined in the Compose file.
  - docker-compose pause/resume: Pause/resume running containers.
  - docker-compose build: Build or rebuild services defined in the Compose file.
-

Docker Compose (2)

```
version: '3'
services:
  web:
    build: .
    ports:
      - "8080:80"
    volumes:
      - ./app:/app
  db:
    image: mysql:latest
    environment:
      - MYSQL_ROOT_PASSWORD=root
```

- Benefits: Simplifies managing complex multi-container applications, easy to version and share configurations, enables running applications with a single command.

Best Practices for Docker

- Docs:
  - https://docs.docker.com/develop/dev-best-practices/#how-to-keep-your-images-small
  - https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
  - https://docs.docker.com/develop/security-best-practices/

Hands-On Exercise -

Containerizing Selenium and Postman for Testing

Objective: Containerize Selenium for web automation testing and use Postman/Newman for API testing in Docker.

1. Containerizing Selenium:
    - Build a Docker image with Selenium dependencies and browser driver.
    - Run the Docker container for web automation testing.
    - Verify successful execution of Selenium tests.
2. Using Postman/Newman:
    - Build a Docker image with Postman and Newman dependencies.
    - Run the Docker container for API testing.
    - Verify successful execution of API tests.

Source code: https://github.com/tranphuquy19/QA-DevOps-training

Hands-On Exercise -

Containerizing Selenium and Postman for Testing

Objective: Containerize Selenium for web automation testing and use Postman/Newman for API testing in Docker.

1. Containerizing Selenium:
    - Build a Docker image with Selenium dependencies and browser driver.
    - Run the Docker container for web automation testing.
    - Verify successful execution of Selenium tests.
2. Using Postman/Newman:
    - Build a Docker image with Postman and Newman dependencies.
    - Run the Docker container for API testing.
    - Verify successful execution of API tests.

Source code: https://github.com/tranphuquy19/QA-DevOps-training

Summary and Additional Resources

Additional Resources:

1. Docker Documentation: Official documentation from Docker for detailed information and guides on Docker best practices.
    - Link: docker.com/get-started
2. Docker Best Practices Guide: A comprehensive guide on Docker best practices, including container security, performance optimization, and image management.
    - Link: docs.docker.com/develop
3. Selenium: Official documentation for Selenium WebDriver to understand how to automate web browser testing.
    - Link: selenium.dev/documentation
4. Postman and Newman Documentation: Official documentation for Postman and Newman tools for API testing and automation.
    - Link: postman.com/docs