Week 3 - Package Management (Docker) Using Real-Time Scenarios & Understanding SonarQube

- Conceptual Concepts of Dockers
- What is Virtualization before deep dive into the Containerization
- O.S level virtualization
- Docker vs Virtual Machine
- What is Docker and its History
- Docker Architecture
- Advantages and limitations of Docker
- Components of Docker (Docker Daemon, Docker Client, Docker Host)
- Docker Image
- Docker lifecycle
- Docker Image TAR and Unarchive, Docker container states, Docker Networking
- (Create and Manage), Dockerfile and CD flow
- CD Tools with Docker (Integrating CD tools like Jnekins and Github action using projects)
- Docker Networking
- Docker Security Introduction
- Docker volume

SonarQube, Quality Gates, and Profiles:

- Understanding SonarQube's Role in Code Quality Assessment
- Implementing Quality Gates to Ensure Code Quality Standards
- Configuring and Managing SonarQube Profiles for Code Analysis

---

Virtualization

Virtualization is a technology that allows you to create multiple simulated environments or dedicated resources from a single physical hardware system. It involves the use of software to create an abstraction layer over the physical hardware, enabling the creation of virtual machines (VMs). These VMs operate as if they were individual physical computers, with their own operating systems and applications. Virtualization gives birth to cloud computing

Types of Virtualization

- Hardware Virtualization: Creates virtual versions of computers and operating systems.

- Network Virtualization: Combines hardware and software network resources and network functionality into a single, software-based administrative entity.

- Storage Virtualization: Pools physical storage from multiple network storage devices into what appears to be a single storage device managed from a central console.

Benefits of Virtualization

- Resource Efficiency: Increases the utilization of physical resources by sharing them among multiple virtual environments.

- Scalability: Easily scale resources up or down according to demand.

- Isolation: Each VM is isolated from others, enhancing security and stability.

- Cost Savings: Reduces the need for physical hardware, leading to lower capital and operational costs.

Containerization

Containerization is a technology that packages an application and its dependencies into a standardized unit called a container. Containers are lightweight, portable, and ensure that applications run consistently across different environments. Container Images are like the recipe for your application, Once this recipe has been built/created, Then we can launch millions of containers from this recipe within a second. That is the power of Containerization Technology.

Docker

Docker is a tool designed to create, deploy, and run applications using containers. Containers package an application with its dependencies, allowing it to run consistently across various environments.

The Docker platform

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host. Containers are lightweight and contain

everything needed to run the application, so you don't need to rely on what's installed on the host. *You can share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.*

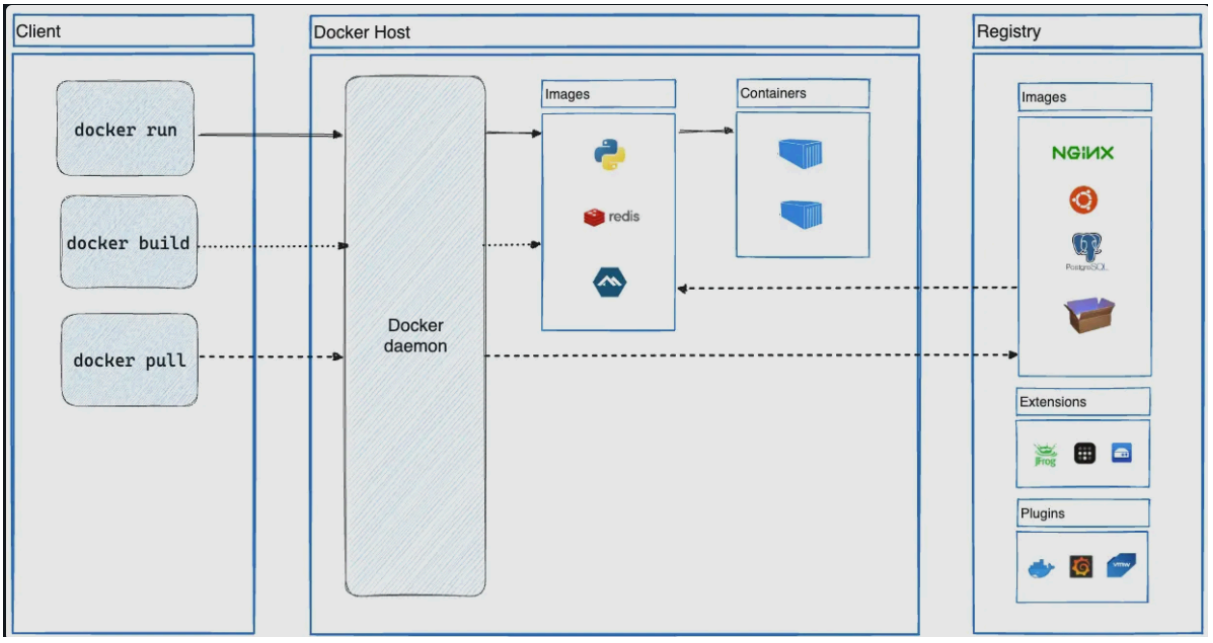
Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers.
- The container becomes the unit for distributing and testing your application.
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API.

Docker Architecture Diagram:



Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker registries:

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

Docker objects:

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Docker Images

Docker Image is a template for creating Docker containers. It contains all dependencies and libraries needed to run an application. We can create a Docker image by using Dockerfile or by using the docker commit command. It is like a Package that contains all the dependencies and libraries required to run the application.

Docker Container

A Docker Container is a running instance of a Docker image. We can create a Docker container from a Docker image. It is similar to a virtual machine but more lightweight as it won't create the guest operating system from scratch.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

The underlying technology Docker uses:

Docker is written in the Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. Docker uses a technology called ***namespaces*** to provide the isolated workspace called the container. When you run a container, Docker creates a set of namespaces for that container.

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Commands

- *docker version*: Shows Docker version.
- ***docker info***: Displays Docker information.
- ***docker images***: Lists all Docker images.
- ***docker ps***: Shows running Docker containers.
- ***docker ps -a***: Lists all Docker containers.
- ***docker pull <image-name>***: Pulls a Docker image from the Docker Hub.
- ***docker run <image-name>***: Runs a Docker image, creating a Docker container.
- ***docker run -it <image-name>***: Runs a Docker image, creating a Docker container with an open interactive terminal.
- ***docker run -it -d <image-name>***: Create a Docker container but run it in background(detach mode).
- ***docker run -it -d -p 8080:80 <image-name>***: Runs a Docker image, creating a background Docker container, mapping port 8080 on the host to port 80 on the container.
- ***docker attach <container-id>***: Attaches the terminal to a Docker container.
- ***docker exec -it <container-id> bash***: Opens a terminal in a Docker container.
- ***docker stop <container-id>***: Stops a Docker container.
- ***docker start <container-id>***: Starts a Docker container.

- ***docker rm <container-id>:*** Deletes a Docker container.
- ***docker rmi <image-id>:*** Deletes a Docker image.
- ***docker commit <container-id> <image-name>***: Creates a Docker image from a Docker container.
- ***docker rm -f <container-id>***: Deletes a Docker container forcefully.
- ***docker rm -f $(docker ps -a -q)***: Deletes all Docker containers forcefully.
- ***docker rmi -f <image-id>***: Deletes a Docker image forcefully.
- ***netstat -tnlp:*** Displays all ports running on the host machine.
- ***ctrl+p+q***: Detaches the terminal from the Docker container without stopping it.

Docker run command

The following command runs an centos container, attaches interactively to your local command-line session, and runs /bin/bash.

#docker run -it centos:7
When you run this command, the following happens (assuming you are using the default registry configuration):

If you don't have the centos:7 image locally, Docker pulls it from your configured registry, as though you had run *docker pull centos:7* manually.

Docker creates a new container, as though you had run a docker container create command manually.

Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.

Docker creates a network interface to connect the container to the default network, since you didn't specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.

Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while Docker logs the output to your terminal.

When you run exit to terminate the /bin/bash command, the container stops but isn't removed. You can start it again or remove it.

Dockerfile Instructions

- ***FROM*** - This instruction is used to specify the base image.
- ***RUN*** - This instruction is used to run commands at the time of building the image.
- ***LABEL*** - This instruction is used to add metadata to the image. You can specify any key-value pair as metadata such as maintainer, description, version, etc.
- ***COPY*** - This instruction is used to copy files from the local machine to the docker image.
- ***ENV*** - This instruction is used to set environment variables inside of your image.
- ***WORKDIR*** - This instruction is used to set the working directory for the instructions that follow it.
- ***CMD*** - This instruction is used to specify the command that needs to be executed when a container is created from the image.
- ***ENTRYPOINT*** - This instruction is used to specify the command that needs to be executed when a container is created from the image. You can specify any command that you would normally run on a Linux machine. The difference between *CMD* and *ENTRYPOINT* is that CMD can be overridden by passing arguments to the docker run command. Whereas ENTRYPOINT cannot be overridden by passing arguments to the docker run command, even ENTRYPOINT will use those arguments in the base command and then execute it.

Docker Container States

Docker containers can be in several states throughout their lifecycle. Here are the main container states:

- Created: This state occurs when a container is created using the docker create command but has not yet started. In this state, the container exists but is not running.

- Running: When a container is started using the docker start command, it transitions into the running state. In this state, the container's processes are actively running, and it is executing the commands defined in its Docker image.

- Paused: Containers in the running state can be paused using the docker pause command. Pausing a container suspends its processes, effectively freezing its state. This can be useful for temporarily halting container execution without stopping it entirely.

- Restarting: When a container is restarted either manually using the docker restart command or automatically due to a failure or restart policy, it enters the

restarting state. In this state, Docker is stopping the container's processes and preparing to start them again.

- Exited: Containers transition to the exited state when their main process completes execution and they stop naturally. This could happen because the container's task is finished, or there was an error during execution. Containers in the exited state retain their filesystem and configuration but are not actively running.

- Dead: The dead state indicates that Docker has stopped monitoring the container. This could happen if the container process crashes or if Docker itself encounters an error while managing the container.

- Paused: As mentioned earlier, containers can also be explicitly paused using the docker pause command. In this state, the container's processes are suspended, preserving their state until the container is unpaused.

- Stopped: When a container is manually stopped using the docker stop command, it transitions into the stopped state. Unlike the exited state, which indicates a natural termination of the container's processes, containers in the stopped state are explicitly halted by the user.

Docker Hub

Docker Hub is a container registry built for developers and open-source contributors to find, use, and share their container images. With DockerHub, developers can host public repos that can be used for free, or private repos for teams and enterprises.

To push an image to Docker Hub, you can run the following commands:

- *docker login*
- *docker tag <image_name> <docker_hub_username>/<image_name>*
- *docker push <docker_hub_username>/<image_name>*

To pull an image from Docker Hub, you can run the following command:

- *docker pull <docker_hub_username>/<image_name>*

To Push image into any other registry, you can run the following commands:

- *docker login <registry_url>*
- *docker tag <image_name> <registry_url>/<image_name>*
- *docker push <registry_url>/<image_name>*

Docker Image TAR And Unarchive

These commands allow you to tar and untar Docker images, making it easier to distribute and share images across different Docker environments.

- To tar a Docker image, you can use the docker save command. This command exports an image to a tarball file.

  #docker save -o myimg.tar <image-name>

- To untar a Docker image from a tarball file, you can use the docker load command. This command imports an image from a tarball file.

  #docker load -i myimg.tar

Docker Security Introduction

Docker has revolutionized the way applications are packaged, distributed, and deployed, but ensuring the security of Docker environments is Important

- Container Isolation: Docker containers provide a level of isolation, allowing applications to run in their own environments without interfering with each other or the host system. However, it's essential to understand that this isolation is not absolute, and vulnerabilities in the container runtime or kernel can potentially be exploited.

- Immutable Infrastructure: Docker promotes the concept of immutable infrastructure, where containers are treated as disposable and are rebuilt from scratch whenever changes are required. This approach reduces the attack surface by minimizing the time a potentially vulnerable container is running and by ensuring that containers are always deployed from a known, secure base image.

- Image Security: Docker images serve as the foundation for containers, so ensuring the security of these images is crucial. Best practices include regularly scanning images for vulnerabilities using tools like Clair, Trivy, or Anchore, and only using images from trusted sources.

- Container Configuration: Securely configuring Docker containers is essential for minimizing security risks. This includes practices such as running containers with the least privilege necessary, avoiding running containers as root whenever possible, and restricting network access to only what is required.

- Container Orchestration: Docker is often used in conjunction with container orchestration platforms like Kubernetes or Docker Swarm. These platforms provide additional security features such as network policies, pod security policies, and role-based access control (RBAC) to help secure containerized workloads at scale.

- Monitoring and Logging: Implementing robust monitoring and logging solutions is crucial for detecting and responding to security incidents in Docker environments. Tools like Docker Security Scanning, Sysdig, or Prometheus can help monitor container activity and collect security-related metrics and logs.

- Patch Management: Keeping Docker hosts and containers up-to-date with the latest security patches is essential for mitigating known vulnerabilities. Regularly updating Docker Engine, the host operating system, and container images can help reduce the risk of exploitation.

Docker-Compose:

Docker Compose is a tool for defining and running multi-container applications. Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

Compose works in all environments; production, staging, development, testing, as well as CI workflows. We would be mostly using it for testing environments. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

Install Docker Compose in amazon linux.

- Download the docker-comose - *sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose*

- Make docker-compose executable - *sudo chmod +x /usr/local/bin/docker-compose*

- *docker-compose --version*

SonarQube

SonarQube is an open-source platform designed to continuously inspect code quality, providing insights into its health and security. Originally developed by SonarSource, it has become a popular tool in the software development community. SonarQube operates on the principle of static code analysis, examining source code without executing it. Here are some key aspects of SonarQube:

- Code Quality Analysis: SonarQube assesses code quality based on a set of predefined rules and best practices. It analyzes various aspects such as code complexity, duplication, coding standards adherence, potential bugs, and security vulnerabilities.

- Language Support: SonarQube supports multiple programming languages including Java, JavaScript, C#, Python, C/C++, TypeScript, PHP, and more. This broad language support makes it applicable to a wide range of development projects.

- Integration: SonarQube seamlessly integrates with popular development tools and Continuous Integration (CI) platforms such as Jenkins, Azure DevOps, GitHub Actions, and GitLab CI/CD. This integration enables developers to automate code analysis as part of their build pipelines, ensuring that code quality checks are performed consistently.

- Dashboard and Reports: SonarQube provides a centralized dashboard where developers and project managers can monitor code quality metrics and trends over time. It generates detailed reports highlighting issues identified during code analysis, along with recommendations for improvement.

- Security Vulnerability Detection: In addition to code quality, SonarQube also helps identify security vulnerabilities within the codebase. It scans for common

security weaknesses such as SQL injection, Cross-Site Scripting (XSS), and insecure authentication mechanisms.

- Customizable Rules: While SonarQube comes with a default set of rules, organizations can customize these rulesets to align with their specific coding standards and policies. This flexibility allows teams to tailor code analysis to their unique requirements.

SonarQube Architecture -

- Web Server: SonarQube's web server component serves as the entry point for users to interact with the platform via a web browser. It handles user authentication, and session management, and serves the SonarQube user interface.
  The web server also communicates with other components to fetch analysis results, project configurations, and other necessary data.

- Database: SonarQube relies on a relational database to store various types of data, including:
  - Configuration settings
  - Analysis results
  - User accounts and permissions
  - Historical data for trend analysis
  - Supported databases include PostgreSQL, Microsoft SQL Server, Oracle, and MySQL.

- Scanner: The SonarQube Scanner is a command-line tool or plugin used to analyze source code and send the results to the SonarQube server for processing. It can be integrated into Continuous Integration (CI) pipelines to automate code analysis as part of the build process.The scanner collects code metrics, identifies code issues, and sends this data to the SonarQube server for further processing.

- Analysis Engine: The analysis engine is responsible for processing code analysis data received from the scanner and applying predefined rules to detect code issues and quality violations. It leverages language-specific analyzers to parse source code, extract relevant information, and perform static code analysis.

The analysis engine calculates various code metrics and identifies code smells, security vulnerabilities, and other issues.

- Rules Engine: SonarQube's rules engine contains a set of predefined coding rules and best practices across various programming languages. These rules cover a wide range of aspects including code complexity, code duplication, coding standards adherence, security vulnerabilities, and potential bugs. Organizations can customize the rulesets to enforce their specific coding standards and policies.

- User Interface (UI): The SonarQube UI provides a user-friendly interface for developers, project managers, and other stakeholders to view analysis results, monitor code quality metrics, and manage projects. It includes dashboards, project overviews, detailed issue reports, and trend analysis charts. The UI is accessed through a web browser and interacts with the SonarQube server to retrieve and display relevant data.

- Extensions: SonarQube supports extensions, plugins, and integrations to extend its functionality and integrate with other tools and services. These extensions can provide additional code analysis rules, integrate with third-party issue trackers, source code repositories, and CI/CD platforms, or add custom reporting capabilities.

Let’s Install SonarQube Server on top of docker with the help of docker-compose.

Use the below docker-compose.yml file

```
version: "3.8"
name: mysonarqube
services:
  sonarqube:
    container_name: sonarqube
    image: sonarqube
    depends_on:
      - sonarqube-database
    environment:
      - SONARQUBE_JDBC_USERNAME=sonarqube
      - SONARQUBE_JDBC_PASSWORD=sonarpass
      -
SONARQUBE_JDBC_URL=jdbc:postgresql://sonarqube-database:5432/sonarqube
    volumes:
```

```
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_bundled-plugins:/opt/sonarqube/lib/bundled-plugins
    ports:
      - 9000:9000

  sonarqube-database:
    container_name: sonarqube-database
    image: postgres:12
    environment:
      - POSTGRES_DB=sonarqube
      - POSTGRES_USER=sonarqube
      - POSTGRES_PASSWORD=sonarpass
    volumes:
      - sonarqube_database:/var/lib/postgresql
      - sonarqube_database_data:/var/lib/postgresql/data
    ports:
      -   5432:5432


volumes:
  sonarqube_database_data:
  sonarqube_bundled-plugins:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_database:
  sonarqube_extensions:
```

Here we have created a multi-container sonarqube server with the database, We have also Defined volumes for persistent storage used by SonarQube and PostgreSQL containers. We have also mapped the container's port 9000 to host port 9000 to access SonarQube's web interface.

- Run the command - #docker-compose up -d
- We can access the SonarQube server using the following URL - http://IP:9000
- We can use the default username and password to log in to the SonarQube server.
    - username: admin
    - password: admin

Docker Networking

Docker networking refers to the capability within Docker containers to communicate with each other and with the outside world. It provides a way for containers to connect, share data, and access resources while remaining isolated from each other.

- Bridge Network: By default, Docker creates a bridge network called bridge for communication between containers on the same host. Containers connected to this network can communicate with each other using IP addresses assigned by Docker.

- Container Ports: Docker containers can expose specific ports, allowing external processes to communicate with services running inside the container. These ports can be mapped to ports on the host system, enabling external access.

- Host Networking: Docker containers can also use the host's network stack directly, bypassing Docker's network isolation. This can be useful for high-performance scenarios but may sacrifice some degree of isolation.

- Custom Networks: Docker allows users to create custom networks to facilitate communication between specific groups of containers. Custom networks can provide better isolation and organization than the default bridge network.

- Overlay Networks: Docker Swarm, Docker's native clustering and orchestration tool, supports overlay networks. These networks enable communication between containers across multiple Docker hosts in a cluster, facilitating distributed applications.

Benefits of Docker Networking:

- Isolation: Docker networking provides isolation between containers, preventing interference and conflicts between different services running on the same host.

- Flexibility: Docker's networking features offer flexibility in configuring communication between containers, allowing for various deployment scenarios and architectures.

- Scalability: With Docker Swarm and overlay networks, Docker enables the creation of scalable and distributed applications that span multiple hosts while maintaining seamless communication between containers.

- Simplicity: Docker's networking model simplifies the process of setting up and managing network connectivity for containers, abstracting away much of the complexity of traditional networking configurations.

Create a custom docker network - *#docker network create --driver bridge --subnet=172.18.0.0/16 my_gfg_network*

Now while launching the container, we can use “--network” to specify the network in which you want to launch the container

Docker Volume:

Docker volumes provide a way to persist data generated and used by Docker containers. They allow containers to store and share data independently of the container's lifecycle, enabling data to persist even if the container is stopped or removed. Here's a brief overview of Docker volumes:

- Persistent Storage: Docker volumes provide persistent storage for containers. Unlike data stored within a container's writable layer, which is ephemeral and deleted when the container is removed, data stored in volumes persists beyond the lifecycle of the container.

- Flexible Mounting: Volumes can be mounted into one or multiple containers simultaneously. This allows multiple containers to share and access the same data, facilitating collaboration and data sharing between containers.

- Data Sharing: Volumes can be used to share data between a container and the host system or between multiple containers. This makes it easy to exchange files and resources between containers and external systems.

- Types of Volumes: Docker supports various types of volumes, including named volumes, host-mounted volumes, and anonymous volumes. Named volumes provide a way to manage and reference volumes using user-defined names, while host-mounted volumes allow containers to access directories on the host filesystem. Anonymous volumes are created automatically by Docker and are typically used for temporary or disposable data.

- Creating a Docker Volume : *#docker volume create <volume_name>*
- Listing Docker Volumes: *#docker volume ls*
- Inspecting a Docker Volume: *#docker volume inspect my_volume*
- Using a Docker Volume in a Container

To use a Docker volume with a container, you use the -v or --mount flag when running the container. The -v flag has the syntax <volume_name>:<path_in_container>. For example:
#docker run -d --name my_container -v my_volume:/app/data nginx