SOFTWARE ENGINEERING

UNIT – 4

CHAPTER – 1

SOFTWARE TESTING STRATEGIES AND ART OF DEBUGGING

I. INTRODUCTION TO TESTING

Software testing is a critical phase in the software development process aimed at ensuring the quality and reliability of the software.

Testing helps identify defects, vulnerabilities, and issues that could affect the software's performance and functionality. Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

II. A STRATEGIC APPROACH TO SOFTWARE TESTING:

1. A strategic approach to software testing involves planning, organizing, and executing tests systematically.
2. It helps in maximizing test coverage, reducing risks, and optimizing resources.
3. Testing is a set of activities that can be planned in advance and conducted systematically

Key Elements of a Testing Strategy

1. **Testing Objectives**: Clearly define what the testing process aims to achieve. This could include functionality validation, performance testing, security testing, etc.
2. **Test Scope**: Identify the components or modules of the software to be tested.
3. **Test Environment**: Establish the hardware, software, and data configurations required for testing.
4. **Testing Methods**: Determine the types of testing methods to be used, such as manual testing, automated testing, or a combination.
5. **Test Schedule**: Create a timeline for testing activities, including milestones and deadlines.
6. **Resource Allocation**: Allocate human and technical resources for testing, including testers,

testing tools, and equipment.

7. **Risk Assessment**: Identify potential risks in the testing process and develop mitigation strategies.
8. **Test Documentation**: Maintain comprehensive test documentation, including test plans, test cases, and test reports.

Testing vs Debugging:

Testing and debugging are two essential phases in the software development life cycle with distinct purposes.

Testing involves the systematic execution of a program to ensure it behaves as expected, verifying that it meets specified requirements and quality standards. Testers or quality assurance professionals are responsible for designing and executing tests throughout the development process.

Debugging occurs after testing and is the process of identifying and rectifying defects or errors discovered during testing. Developers take charge of debugging, using various tools and techniques to trace and correct issues, aiming to enhance the overall quality of the software.

While testing aims to validate the software's functionality and adherence to requirements, debugging focuses on locating and fixing specific issues to ensure a robust and error-free codebase. Both testing and debugging are integral for delivering reliable and high-quality software products.

Verification and Validation:

Verification and validation are two distinct processes in the quality assurance and testing of software. Verification involves the assessment of work products throughout the development life cycle to ensure that they meet specified requirements. This process focuses on determining whether the software is being built correctly by checking the adherence to design specifications and standards. Verification activities include reviews, inspections, and walkthroughs. On the other hand, validation is concerned with evaluating the end product to ensure it meets the customer's expectations and satisfies its intended purpose. It verifies whether the software fulfills the user's needs and requirements. Validation activities include dynamic testing, such as system testing and acceptance testing. In essence, verification checks that the software is built right, while validation confirms that the right software is being built, collectively ensuring the development of a high-quality and reliable product.

Verification refers to set of activities that ensure that software correctly implements a specific function.

Example: *Verification: Are we building the product, right?*

Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements

Example: *Validation: Are we building the right product?*

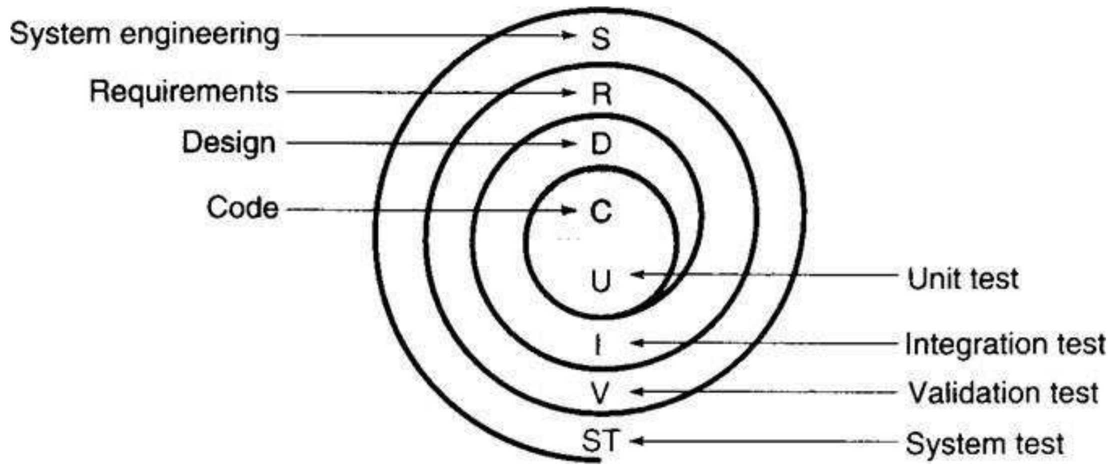
III. SOFTWARE TESTING STRATEGY FOR CONVENTIONAL SOFTWARE:

Software testing strategy for conventional software involves a systematic approach to ensure the product's quality, functionality, and adherence to requirements. The process begins with a thorough analysis of software requirements, followed by the development of a comprehensive test plan that outlines testing objectives, scope, and resources. Test design includes the creation of detailed test cases covering various scenarios. The establishment of a reliable test environment mirroring the production setup is crucial.

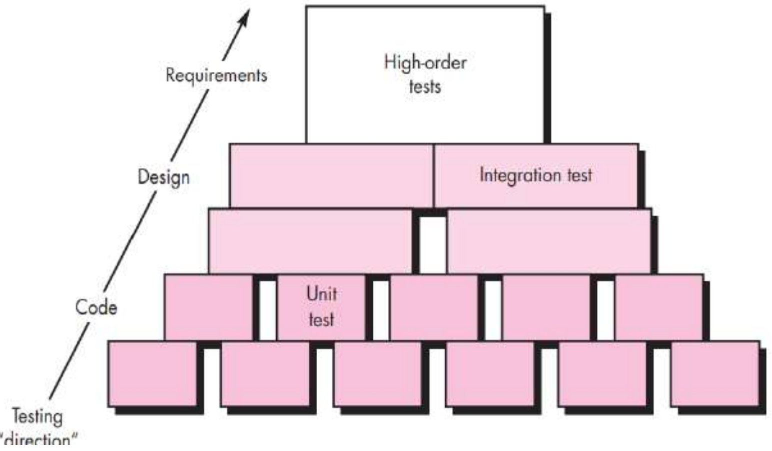
During test execution, comprehensive testing, including functional, performance, and security aspects, is conducted. Defects are tracked and managed, and regression testing is performed to ensure that new changes do not introduce new issues. User acceptance testing involving collaboration with end-users is pivotal for validating the software against business requirements.

Documentation of test plans, cases, and results is maintained, and a test closure report is generated summarizing the testing process. Continuous improvement is emphasized by gathering feedback and refining testing processes. Overall, this strategy ensures a systematic and thorough validation of conventional software, promoting quality and reliability.

A strategy for software testing may be viewed in the context of the spiral as shown below



- Unit testing begins at the vortex of the spiral and concentrates on unit [i.e., components] of the software as implemented in the source code
- Taking another turn by moving along the spiral to integrate testing which focus on design and the construction of software architecture
- Next turn we encounter validation testing which validate requirements established as part of software requirements analysis against software that has been constructed
- Finally, we arrive at system testing, where the software and other system elements are tested as whole.



a. Unit Testing:

Unit testing is a software testing technique where individual components or units of a software application are tested in isolation. The primary purpose of unit testing is to validate that each unit of the software performs as designed. A "unit" in this context refers to the smallest testable part of an application, typically a single function, method, or procedure.

Unit test Considerations:

In unit testing, it's crucial to consider various aspects, including testing interfaces, local data structures, and boundary conditions. Let's explore each of these considerations:

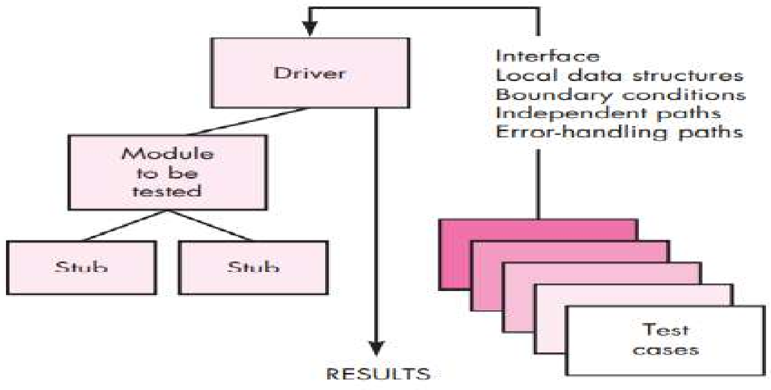
1) Testing Interfaces:
   a) Method Signatures: Verify that the public methods or functions within the unit adhere to their specified interfaces, including parameter types, return types, and exception handling.
2) Collaborating Components: If the unit interacts with other components or services, use mocking or stubbing to isolate the unit and focus on testing its behavior independently.
3) Local Data Structures:
   a) Initialization: Test the unit's behavior when initializing or creating local data structures, ensuring that they are set up correctly.
   b) State Changes: Evaluate how the unit modifies and interacts with local data structures during its execution. Verify that the expected state changes occur.
4) Boundary Conditions:
   a) Input Boundary Conditions: Test the unit with minimum and maximum valid input values to ensure it handles boundary conditions appropriately.
   b) Output Boundary Conditions: Verify that the unit produces correct results when approaching or reaching limits, such as the maximum value of an integer or the end of an array.
5) Error Handling: Test how the unit behaves when invalid or unexpected inputs are provided. Ensure proper error handling and validation.
6) Exception Handling: Exception Scenarios: Test the unit under conditions that may lead to exceptions. Ensure that the unit handles exceptions gracefully and provides meaningful error messages when necessary.

a) Error Paths: Verify that the unit correctly follows error paths in the code, triggering and handling exceptions as expected.

b) By thoroughly considering these aspects in unit testing, developers can create tests that cover a wide range of scenarios, leading to more robust and reliable software components. Additionally, these considerations help identify and address issues related to interfaces, data structures, and boundary conditions early in the development process.

Unit Test Procedures:

- Unit testing is normally considered as an adjunct to the coding step.
- Design of unit test can be performed before code begin or after code generation
- Each test case should be coupled with a set of expected results.
- Driver and/or Stub software must often be developed for each unit test.



In unit testing, the terms "driver" and "stub" refer to specific testing components that assist in the testing process. These components are used to isolate and control the behavior of the unit under test, ensuring that it can be tested in a controlled environment. Let's explore the roles of drivers and stubs in unit testing:

Driver:

A driver is a testing component that is responsible for invoking the unit under test and driving the execution of the test cases. It is used when testing a component that depends on or interacts with other components or services.

The driver simulates the higher-level components or systems that the unit under test interacts with, ensuring that the unit receives the necessary inputs and can be tested in isolation.

Drivers are temporary components created specifically for testing purposes and are not part of the production code.

Stub:

A stub is a testing component that replaces a dependent component or module in the system. It is used when testing a component that relies on the functionality of another component, and the real implementation of that component is not available or is impractical to use in the testing environment.

Stubs provide a simplified or predefined response to the calls made by the unit under test, allowing the testing of the unit in isolation. Stubs are used to emulate the behavior of the real component and ensure that the unit under test can be tested independently of its dependencies.

b. Integration Testing:

Integration testing is a phase in the software testing process where individual software components or modules are combined and tested as a group. The primary goal of integration testing is to ensure that the integrated components work together as expected, collectively providing the desired functionality of a larger system or subsystem. Key characteristics of integration testing include:

Combining Components: In integration testing, individual software components, which have been unit-tested and verified in isolation, are combined and tested together. This may involve testing the interaction between modules, classes, services, or any other building blocks of the software.

Identifying Interface Issues: Integration testing focuses on identifying issues related to the interfaces and interactions between integrated components. This includes testing how data is passed between modules, how different components communicate, and how they handle shared resources.

Integration Points: Integration points refer to the areas where components connect or interact with each other. These points are critical for testing to ensure that data and control flow smoothly between the integrated components.

Stubs and Drivers: During integration testing, stubs and drivers may be used to simulate the behavior of components that are not yet integrated or are external to the system. Stubs are used to emulate lower-level components, while drivers simulate higher-level components.

Functional and Non-functional Testing: Integration testing includes both functional and non-functional testing. Functional testing verifies that the integrated components perform the intended functions, while non-functional testing assesses aspects like performance, scalability, and reliability in the integrated environment.

Incremental Approach: Integration testing is often performed incrementally, starting with the integration of smaller components and gradually incorporating larger portions of the system. This allows for the early detection and resolution of integration issues.

Regression Testing: As new components are integrated, regression testing is conducted to ensure that existing functionalities are not negatively impacted. This helps maintain the stability of the system as it evolves.

Incremental Integration:

Incremental integration is an approach to software development and testing where components or modules are gradually added to the system and tested incrementally, one at a time. The incremental integration process allows for the systematic integration of individual units or modules, making it easier to identify and address integration issues early in the development lifecycle.

Different types of incremental integration are available

(a) Top-Down Incremental Integration:
(b) Bottom-Up Incremental Integration:
(c) Functional Incremental Integration:
(d) Non-functional Incremental Integration:
(e) Big Bang Incremental Integration:
(f) Continuous Integration:
(g) Feature-Based Incremental Integration:
(h) Concurrency Incremental Integration:

(i) Mixed Incremental Integration:

(j) Regression Incremental Integration:

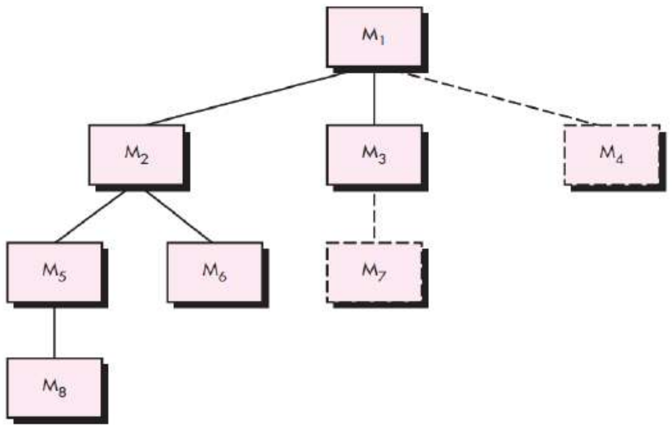
Commonly used are

a. Top-Down Integration
b. Bottom-Up Integration

(1) <u>Top – Down Integration</u>:

Top-down integration is a software development and testing approach where the system is built and tested from the top (high-level modules or components) to the bottom (low-level modules or components). In other words, the integration process starts with the higher-level functionalities and gradually incorporates the lower-level ones. The steps involved are:

- <u>Start with the Main Module:</u> Begin the integration process by focusing on the main or top-level module of the system. This module represents the overall functionality of the software.
- <u>Stub Implementation:</u> For any lower-level modules or components that the main module depends on but have not yet been implemented, placeholder modules called "stubs" are created. These stubs simulate the behavior of the lower-level modules.
- <u>Progressively Integrate:</u> Gradually integrate the lower-level modules one at a time. For each lower-level module, replace the corresponding stub with the actual implementation.
- <u>Test After Each Integration:</u> After integrating each lower-level module, conduct testing to ensure that the integrated components work together as expected. This involves both functional and non-functional testing.
- <u>Repeat the Process:</u> Continue this process iteratively, moving down the hierarchy of modules until all modules are integrated and the entire system is built.
- The subordinate module to main control module is integrated either in a
  - Depth first (Or)
  - Breadth first manner



- **<u>Depth First Integration:</u>**

In depth-first integration, the integration process starts with the top-level module, and instead of integrating and testing one lower-level module at a time, it goes deeper into the hierarchy before returning to the next sibling module. This approach prioritizes the exploration of a specific path or branch of the module hierarchy, hence the term "depth-first."

For example, selecting the left-hand path components M1, M2, M5 would be integrated first and next M8. Then the central and right-hand control paths are built

- **<u>Breadth First integration:</u>**

Breadth-first integration is another software integration testing strategy, and it contrasts with depth-first integration. In breadth-first integration, the integration process starts with the top-level module, but instead of going deep into a specific branch of the hierarchy, it integrates and tests all immediate submodules (siblings) at the same level before moving to the next level. This approach explores the integration of components horizontally across the hierarchy, hence the term "breadth-first."

For example, components M2, M3 and M4 would be Integrated first and next M5, M6 and so on.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced

Advantages:

1. Early Identification of System Architecture: The top-down approach allows for early identification and definition of the system architecture and high-level design.
2. Early Detection of Interface Issues: Interface specifications are established early in the process, facilitating early detection and resolution of interface issues.
3. Early Testing of Critical Functionalities: Critical functionalities are implemented and tested early in the development process.
4. Simplified Implementation of Stub Modules: Stub modules (placeholders for lower-level components) are simpler to implement.

Disadvantages:

1. Delayed Detection of Low-Level Issues: Low-level issues may not be detected until later in the development process.
2. Dependency on Stub Modules: The approach relies on the availability and accuracy of stub modules.
3. Postponed Testing of Detailed Functionalities: Testing of detailed functionalities is postponed until lower levels are integrated.

(2) Increased Complexity in Debugging: Debugging can be more complex due to the need to trace issues from higher to lower levels.

2. **Bottom – Up Integration:**

Bottom-up integration is a software development and testing approach where the system is built and tested from the bottom (low-level modules or components) to the top (high-level modules or components). In other words, the integration process starts with the lower-level functionalities and gradually incorporates the higher-level ones.

Steps followed in the bottom-up integration process:

Start with Low-Level Modules: Begin the integration process by focusing on the lowest-level modules or components of the system. These are typically individual functions or units **called Clusters**.

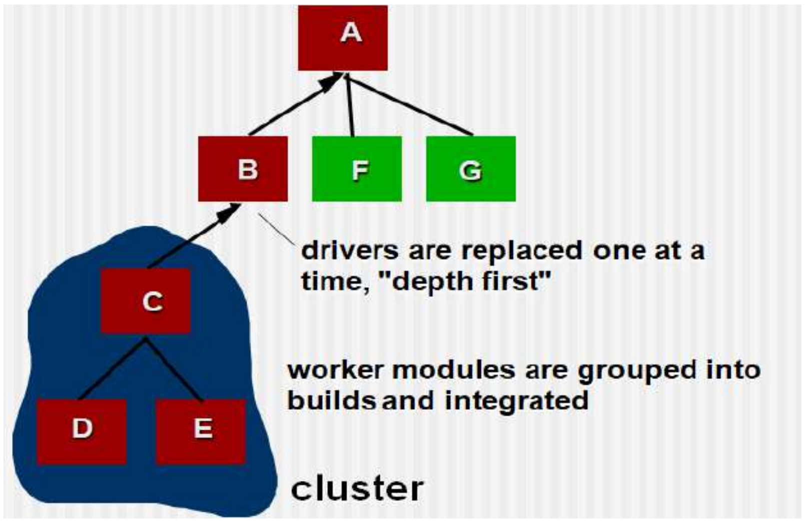
Integrate and Test Low-Level Components: Integrate the low-level modules one by one, and immediately conduct testing on each integrated component to ensure its functionality and correctness.

Build Higher-Level Components: As the lower-level components are integrated and tested, higher-level components called drivers (Drivers are the high-level modules or functionalities that depend on the successful integration of lower-level components) that depend on them are built. This process continues, gradually building up to the top-level modules.

Continue Integration Upwards: Repeat the integration and testing process, moving from the lower levels to higher levels. Each integrated module is tested immediately, and the process continues until the entire system is constructed.

Test Top-Level System: Once all lower-level modules are integrated and tested, the top-level module or the entire system is tested to ensure that all components work together as intended.

The connection between clusters and drivers lies in the hierarchical integration process. Each higher-level driver depends on the successful integration of lower-level clusters. This approach allows for incremental development and testing, starting with smaller units (clusters) and gradually building up to the complete and fully integrated system represented by the top-level driver.


A
B
F
G
C
D
E
drivers are replaced one at a time, "depth first"
worker modules are grouped into builds and integrated
cluster

IV. WHITE BOX TESTING AND BLACK BOX TESTING:

A. <u>White box Testing:</u>

White-box testing, also known as clear-box testing, glass-box testing, or structural testing, is a software testing method that involves examining the internal structure and implementation details of a software application. The goal of white-box testing is to ensure that all components and aspects of the software are functioning correctly according to the specified requirements. Developers do white box testing.

Basis Path Testing:

Basis Path Testing is a white-box testing technique that is based on the control flow structure of the program. It involves identifying a set of linearly independent paths through the program, known as basis paths, and designing test cases to execute these paths. The goal is to ensure that each basis path is exercised at least once during testing.

Steps in Basis Path Testing:

1.Determine the Control Flow Graph (CFG): Construct a Control Flow Graph, which represents the control flow structure of the program. Nodes in the graph represent individual statements or code blocks, and edges represent the flow of control between them.

2.Identify Cyclomatic Complexity (V(G)): Calculate the Cyclomatic Complexity (V(G)) of the program using the formula $V(G) = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components.

3.Identify Basis Set: Identify a set of basis paths, which are linearly independent paths that cover all possible control flow scenarios. These paths collectively represent the basis set for testing.

4.Design Test Cases: Design test cases to execute each basis path. Each test case should follow a specific basis path through the program.

Flow graph notation:

Flow graph notation is used to represent the control flow graph of a program visually. In this notation:

Nodes: Represent statements, conditions, or code blocks.

Edges: Represent the flow of control between nodes.

Cyclomatic Complexity:

Cyclomatic Complexity is a quantitative measure of the complexity of a program's control flow structure. It is calculated using the formula:

$V(G)=E-N+2P$

where:

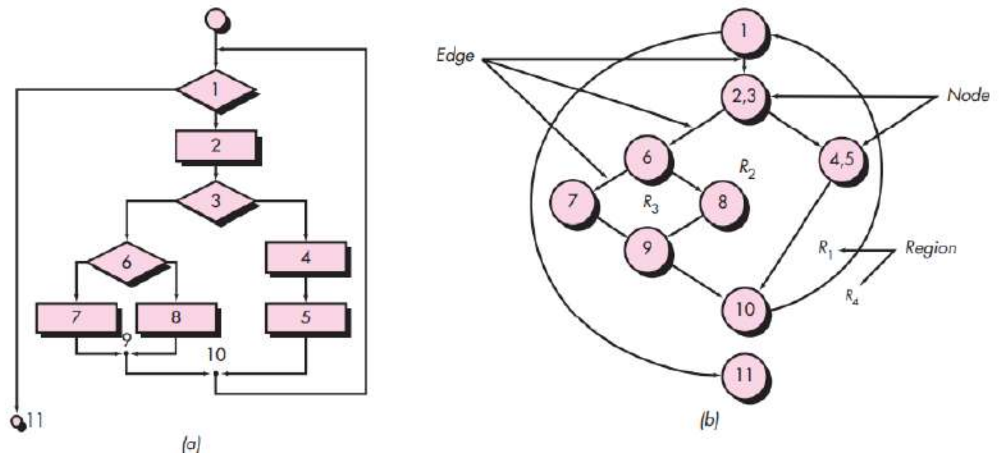
V(G) is the Cyclomatic Complexity.

E is the number of edges in the control flow graph.

N is the number of nodes in the control flow graph.

P is the number of connected components (regions) in the graph.

Cyclomatic Complexity provides insights into the number of test cases required to achieve thorough coverage in basis path testing. The higher the Cyclomatic Complexity, the more test cases may be needed to achieve comprehensive testing coverage.



An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Set of independent paths for the above flow graph

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Cyclomatic complexity calculation for the above flow graph

The flow graph has four regions.

V(G) 11 edges - 9 nodes - 2 = 4

V(G) 3 predicate nodes 1 = 4.

Hence cyclomatic complexity is 4

Finally, prepare test cases that will force execution of each path in the basis set. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Condition Testing: Condition testing, also known as decision testing or branch testing, focuses on testing the different possible outcomes of conditions or decision points within the code. The goal is to ensure that both true and false outcomes of conditions are exercised, helping identify potential logic errors.

Loop Testing: Loop testing focuses on testing the different aspects of loops within the code. This includes testing the loop's entry, exit, and iteration conditions, as well as boundary conditions related to the number of iterations.

B. Black – Box Testing:

Black-box testing is a software testing method that focuses on evaluating the functionality of a software application without examining its internal code or implementation details. In other words, the tester is concerned with the inputs to the software, the expected outputs, and how the software behaves without having knowledge of its internal workings.

- Black box testing attempts to find errors in the following categories

    (1) Incorrect (Or) Missing Functions

(2) Interface errors

(3) Errors in data structures (Or) external data base access

(4) Behaviors (Or) Performance errors

(5) Initialization and termination errors

Equivalence Partitioning:

Equivalence Partitioning is a black-box testing technique that divides the input domain of a software application into classes or groups of data. The goal is to reduce the number of test cases while ensuring that each class is tested at least once. The idea is that if a system works correctly for one input within a partition, it should work correctly for all inputs within that partition.

Steps in Equivalence Partitioning:

Identify Input Classes: Divide the input data into different classes or partitions. Classes should be chosen so that each input within a class is expected to exhibit similar behavior.

Select Representative Test Cases: Choose one or a few representative test cases from each partition. These test cases are used to verify that the software behaves correctly for inputs within that class.

Execute Test Cases: Execute the selected test cases to validate the behavior of the software within each equivalence class.

Verify Correct Behavior: Verify that the software behaves as expected for each equivalence class. If one test case within a class passes, it is assumed that all other inputs in the same class will also pass.

Example:

If a software application accepts positive integers, negative integers, and zero as input, the input domain can be divided into three equivalence classes: positive, negative, and zero. Test cases would then be selected to represent each class.

Boundary Value Analysis:

Boundary Value Analysis is a black-box testing technique that focuses on testing values at the boundaries or edges of input domains. The idea is that errors often occur at the extremes of input ranges, so testing at these boundaries can help uncover potential issues.

Steps in Boundary Value Analysis:

Identify Input Boundaries: Identify the boundaries of the input data. This includes the minimum and maximum values, as well as any values at which the software behavior might change.

Select Test Cases at Boundaries: Choose test cases that represent values at the boundaries and immediately outside the boundaries. For example, if a range is 1 to 100, test cases might include values like 0, 1, 100, and 101.

Execute Test Cases: Execute the selected test cases to verify the behavior of the software at and around the boundaries.

Verify Correct Handling: Verify that the software correctly handles values at the boundaries. This includes testing for correct validation, error handling, and boundary conditions.

Example: If a software application accepts values from 1 to 100, test cases would include values like 1, 100, 0, and 101 to ensure correct behavior at the lower and upper boundaries.

Compatibility testing:

Compatibility Testing is a black-box testing technique that assesses the compatibility of a software application across different environments, platforms, browsers, and devices. The goal is to ensure that the software functions correctly and provides a consistent user experience across a variety of configurations.

Steps in Compatibility Testing:

<u>Identify Target Environments:</u> Identify the target environments, such as different operating systems, browsers, devices, or network configurations.

<u>Design Test Cases:</u> Design test cases that cover a range of configurations, including different browsers, operating systems, screen resolutions, and hardware specifications.

<u>Execute Test Cases:</u> Execute the test cases on each target environment to assess the compatibility of the software.

Verify Consistent Behavior: Verify that the software behaves consistently across different environments and configurations. This includes checking for functionality, performance, and user interface consistency.

Example: If a web application is targeted for use on multiple browsers (Chrome, Firefox, Safari, etc.) and operating systems (Windows, macOS, Linux), compatibility testing would involve executing test cases on each combination to ensure consistent functionality and appearance.

I. **Difference between White-box and Black-box Testing**

Black Box Testing	**White Box Testing**
It is a testing method without having knowledge about the actual code or internal structure of the application.	It is a testing method having knowledge about the actual code and internal structure of the application.
This is a higher level testing such as functional testing.	This type of testing is performed at a lower level of testing such as Unit Testing, Integration Testing.
It concentrates on the functionality of the system under test.	It concentrates on the actual code – program and its syntax's.
Black box testing requires Requirement specification to test.	White Box testing requires Design documents with data flow diagrams, flowcharts etc.
Black box testing is done by the testers.	White box testing is done by Developers or testers with programming knowledge.

V. Validation Testing:

Validation testing is a type of software testing that focuses on evaluating a system or component during or at the end of the development process to determine whether it satisfies the specified requirements. The primary goal of validation testing is to ensure that the software meets the intended use and purpose and that it aligns with the user's expectations and needs.

Key characteristics of validation testing include:

1. User-Centric Focus: Validation testing is centered around the perspective of end-users and their needs. It aims to verify that the software meets user expectations and requirements.
2. Functional and Non-functional Aspects: Both functional and non-functional aspects of the software are validated. This includes ensuring that the software's features work as intended (functional validation) and that it meets performance, usability, security, and other non-functional requirements.
3. End-to-End Testing: Validation testing often involves end-to-end testing, where the entire system or a significant part of it is tested in an environment that closely simulates the production environment.
4. Requirement Compliance: The primary focus is on validating whether the software complies with the specified requirements, which may include both business and technical requirements.
5. Dynamic Testing: Validation testing is typically dynamic, involving the execution of the software to observe its behavior and verify its correctness.

Types of Validation Testing:

Various types of testing fall under the umbrella of validation testing, including:

1. System Testing: Testing the entire system as a whole.
2. Acceptance Testing: Verifying that the system meets user acceptance criteria.
3. Alpha and Beta Testing: Real-world testing in controlled and real environments, respectively.
4. Regression Testing: Ensuring that new changes do not adversely affect existing functionalities.

Validation vs. Verification:

Validation is often contrasted with verification. While verification focuses on checking that the software is built according to its specifications, validation is concerned with evaluating whether the software meets the user's needs and expectations.

Validation Testing Process:

1. Define Validation Criteria: Clearly define the criteria for successful validation. This includes user requirements, business objectives, and other relevant criteria.

2. Design Test Cases: Design test cases based on the defined validation criteria. Test cases should cover various scenarios, user interactions, and system behaviors.
3. Execute Test Cases: Execute the designed test cases in an environment that simulates the production environment. This may involve functional, performance, security, and usability testing.
4. Verify Compliance: Verify whether the software complies with the specified requirements and meets user expectations.
5. Report Defects and Issues: Document and report any defects or issues identified during the validation process.
6. Iterative Validation: If issues are identified, the development team may address them, and the validation process may be repeated iteratively until the software meets the required standards.

Validation testing is a crucial step in the software development life cycle, providing assurance that the developed software delivers the expected value to users and stakeholders. It helps confirm that the software is fit for its intended purpose and contributes to the overall quality of the product.

VI. Alpha and Beta testing:

Alpha testing and beta testing are two stages in the software development life cycle, where a software product undergoes testing before its final release. These testing phases help identify and address issues and bugs to ensure a higher quality and more stable product.

Alpha Testing:

<u>Definition:</u> Alpha testing is the initial phase of software testing, conducted by the internal development team. It is performed in a controlled environment, usually within the development organization.

<u>Purpose:</u> The primary goal of alpha testing is to identify and fix issues within the software before it is released to a larger audience. It helps ensure that the basic functionality of the software works as intended.

<u>Testers:</u> Alpha testing is typically conducted by in-house developers or a dedicated testing team within the organization.

Environment: It is conducted in a development environment, and the testing team has access to the source code to identify and fix any defects found.

Beta Testing:

Definition: Beta testing is the second phase of software testing, conducted by a selected group of external users who are not part of the development team. It takes place in a more realistic environment.

Purpose: Beta testing aims to collect feedback from real users and uncover potential issues in different usage scenarios. It helps validate the software's usability, compatibility, and overall performance.

Testers: Beta testers are external users who represent the target audience. They use the software in their own environments and provide feedback on their experiences.

Environment: Beta testing is done in a more diverse and real-world setting, allowing for the identification of issues that may not have been apparent during alpha testing.

Alpha testing is an internal testing phase performed by the development team to catch basic issues, while beta testing involves external users testing the software in a real-world environment to provide valuable feedback and identify issues that may not have been discovered during the alpha phase. Both testing phases are crucial for delivering a robust and high-quality software product to end-users.

VII. System Testing:

System testing is a level of testing that evaluates the complete and integrated software product. The purpose of system testing is to ensure that the software behaves as expected in its intended environment and meets specified requirements. It is performed after the completion of unit testing, integration testing, and sometimes after acceptance testing.

The types of system testing:

1. Functional Testing: Functional testing verifies that the software functions according to the specified requirements. It involves testing the system's functionality to ensure that it performs as intended.

Examples: Input validation, output verification, data manipulation, and functional correctness testing.

2. Performance Testing: Performance testing assesses how well the system performs under various conditions, such as load, stress, and scalability. It helps identify bottlenecks and ensures that the software meets performance expectations.

   Examples: Load testing, stress testing, scalability testing, and endurance testing.
3. Usability Testing: Usability testing evaluates the user-friendliness of the software. It assesses how easily users can interact with the system, understand its features, and accomplish their tasks.

   Examples: User interface testing, navigation testing, and overall user experience evaluation.
4. Security Testing: Security testing assesses the software's ability to protect data, maintain confidentiality, and prevent unauthorized access. It aims to identify vulnerabilities and ensure that the system is secure.

   Examples: Authentication testing, authorization testing, encryption testing, and penetration testing.
5. Compatibility Testing: Compatibility testing checks the software's compatibility with different operating systems, browsers, databases, and hardware configurations. It ensures that the software can function seamlessly in various environments.

   Examples: Browser compatibility testing, operating system compatibility testing, and database compatibility testing.
6. Reliability and Availability Testing: Reliability testing evaluates the system's ability to perform consistently and reliably under various conditions. Availability testing checks the system's availability and uptime.

   Examples: Reliability testing involves assessing the system's stability, fault tolerance, and error recovery mechanisms.
7. Installation and Deployment Testing: Installation testing verifies the proper installation and configuration of the software. Deployment testing ensures that the software is deployed successfully in the target environment.

   Examples: Installation process testing, configuration testing, and rollback testing.

8. Regression Testing: Regression testing ensures that new changes or enhancements do not negatively impact existing functionality. It involves retesting the entire system or specific components after modifications.

   Examples: Test case reuse, automated regression testing, and selective regression testing.

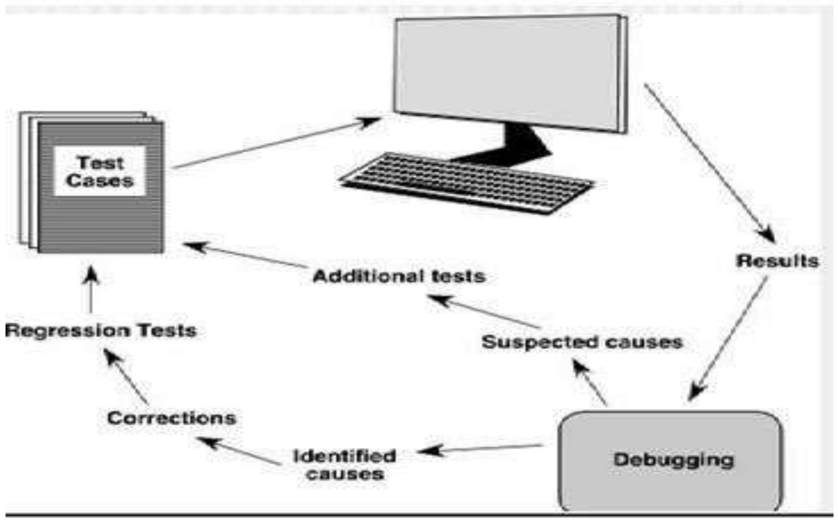
System testing is a critical phase in the software development life cycle, and each type of testing mentioned above plays a specific role in ensuring the overall quality and reliability of the software system.

VIII. The Art of Debugging:

Debugging is an essential skill for software developers, and it's often considered an art because it requires a combination of technical expertise, problem-solving skills, and creativity. The "art of debugging" refers to the skill and practice of identifying, isolating, and fixing bugs or issues within a software system. Debugging is a crucial aspect of the software development process, and it often requires a combination of technical proficiency, problem-solving ability, and creative thinking.

The Debugging Process

The debugging process in software engineering involves systematically identifying, isolating, and fixing bugs or issues within a software system. The process can vary based on the nature of the problem and the tools available.



The steps involved in the debugging process:

Identify the Problem: Begin by identifying the symptoms or issues reported by users or discovered during testing. Understand the specific behavior that deviates from the expected functionality.

Reproduce the Issue: Replicate the problem in a controlled environment. Ensure that you can consistently reproduce the issue, which is essential for effective debugging.

Understand the Code: Analyze the relevant sections of the code. Understand the logic, algorithms, and data flow to identify potential areas where the problem may originate.

Use Logging and Debugging Tools: Insert logging statements or use debugging tools provided by your development environment or IDE. Tools such as breakpoints, variable inspection, and step-through execution are valuable for understanding the program's behavior.

Review Recent Changes: If the issue is related to recent code changes, review the modifications. Version control systems, like Git, can help identify when the problem was introduced.

Isolate the Problem: Narrow down the scope of the issue. Identify the specific part of the code or the conditions that trigger the problem. This step helps in focusing on the relevant sections during debugging.

Check Data and Variables: Inspect the values of variables and data structures at critical points in the code. Verify that data is being processed correctly and that variables have the expected values.

Consult Documentation and Resources: Refer to documentation for libraries, frameworks, and language features. Online forums and communities can provide insights into common issues and their solutions.

Experiment and Hypothesize: Experiment with changes to the code to test hypotheses about the root cause. Make small modifications and observe their impact on the behavior. This iterative process can lead to identifying the bug.

Fix the Issue: Once the bug is identified, implement the necessary code changes to fix the issue. Test the fix thoroughly to ensure that it resolves the problem without introducing new issues.

Regression Testing: After fixing the bug, perform regression testing to ensure that the change does not negatively impact other parts of the system. Re-run relevant test cases to confirm the overall stability of the software.

Document the Fix: Document the bug, steps taken to reproduce it, and the fix implemented. This documentation is valuable for knowledge sharing within the development team and for future reference.

Learn from the Experience: Reflect on the debugging process and learn from the experience. Consider how the issue could have been prevented and apply this knowledge to future development efforts.

Debugging Strategies:

In general, three debugging strategies have been proposed

(1) Brute Force
(2) Back Tracking
(3) Cause Elimination

While brute force, backtracking, and cause elimination are commonly associated with algorithmic problem-solving, they can also be applied as debugging strategies in software development.

Brute Force:

In debugging, the brute force method involves systematically testing various inputs, configurations, or scenarios to identify the cause of a bug. This approach is typically used when the nature of the bug is not well understood, and more sophisticated debugging techniques are not immediately applicable.

Example Scenario: Imagine you are a developer working on a software application, and users have reported that the application occasionally crashes when a specific feature is used. You have little information about the conditions leading to the crash, and the bug is challenging to reproduce consistently.

Identify the Problematic Feature: Begin by identifying the feature or functionality where users have reported issues. In this case, let's say it's a file upload feature.

Explore Different Scenarios: Systematically explore different scenarios related to the file upload feature. This might involve trying various file formats, sizes, and naming conventions. For example, try uploading different types of files, large files, and files with special characters in their names.

Test Edge Cases: Test extreme or edge cases. For instance, try uploading an empty file, a file with the maximum allowed size, or a file with a name that exceeds the character limit.

Vary User Interactions: Consider different user interactions with the application during file upload. For example, try interrupting the upload process, navigating away from the page, or performing other actions while a file is being uploaded.

Use Different Environments: Test the file upload feature in various environments (e.g., different operating systems, browsers, or network conditions) to see if the bug is environment-specific.

Monitor Logs and Outputs: During each test, monitor logs, error messages, and any other relevant outputs. Look for patterns or error messages that may provide clues about the source of the crash.

Iterative Testing: Iterate through different scenarios and gradually narrow down the conditions under which the crash occurs. Make small changes to the test scenarios and observe the impact on the application's behavior.

Isolate the Issue: As you systematically test different scenarios, you may discover a specific condition or input that consistently triggers the crash. Once identified, focus on isolating that particular condition to understand its impact on the application.

Debugging Tools: Use debugging tools, logging statements, or other instrumentation to gain insights into the application's internal state during the tests.

Refine the Search: Refine your approach based on the information gathered during testing. If a specific aspect seems to be related to the crash, focus on exploring that area in more detail.

Document Findings: Document your findings, including the conditions under which the crash occurs and any patterns you observe. This documentation can be valuable when seeking assistance from colleagues or reporting the issue to a bug tracking system.

.

Back Tracking:

Backtracking in debugging refers to the systematic reversal of steps or actions in order to identify the cause of a bug or unexpected behavior. It's a methodical approach to explore the execution path of a program, typically used when the source of an issue is unclear or when a bug is discovered after a series of changes.

Example Scenario:

Suppose you are working on a web application, and users have reported an issue where, under certain conditions, submitting a form results in an error. The bug seems to be intermittent, making it challenging to identify the exact steps leading to the problem.

Identify Recent Changes: Start by identifying recent changes in the codebase that might be related to the form submission process. This could include changes to the form's HTML, JavaScript, or the server-side processing logic.

Review Commits in Version Control: Use version control (e.g., Git) to review recent commits related to the form or the affected components. Look for changes that might have introduced the bug.

Revert Changes: Once you've identified a set of changes that could be related to the issue, use version control to selectively revert those changes. This effectively takes the codebase back to a previous state.

Test to Reproduce the Bug: After reverting changes, test the application to see if the bug can still be reproduced. This helps you determine whether the issue is indeed related to the recent changes you reverted.

Gradual Reintroduction: If the bug persists after reverting changes, gradually reintroduce portions of the changes. Test the application after each reintroduction to identify the specific change or set of changes that triggers the bug.

Isolate the Problematic Code: Continue the process of gradual reintroduction until you isolate the specific code or configuration that leads to the error. This is the point where you've effectively backtracked to identify the root cause.

Inspect Variables and State: As you identify the problematic code, use debugging tools to inspect the state of variables, check the flow of execution, and gather more information about why the error occurs.

Apply Fix or Enhancement: Once you've identified and understood the root cause, apply the necessary fix or enhancement to address the issue.

Test the Fixed Code: After making changes, thoroughly test the application to ensure that the bug is indeed fixed and that the changes do not introduce new issues.

Document and Communicate: Document the debugging process, including the steps taken to backtrack and resolve the bug. Communicate the findings and solution to team members, and consider updating documentation to prevent similar issues in the future.

Cause Elimination:

Cause elimination, also known as fault isolation or root cause analysis, is a debugging technique that involves systematically narrowing down the potential causes of a bug or unexpected behavior by eliminating factors that are unlikely to be the root cause. It's a methodical approach to identify the specific conditions, code, or configurations that contribute to the problem.

Example Scenario:

Suppose you are working on a software application, and users have reported an issue where, intermittently, the application crashes when a specific operation is performed. The goal is to identify and fix the root cause of the crash.

Reproduce the Issue: Begin by attempting to reproduce the issue in a controlled environment. This involves executing the operation or sequence of actions that users reported led to the crash.

Review Recent Changes: Examine recent changes in the codebase, including new features, bug fixes, or updates. Pay special attention to changes related to the area of the code where the crash occurs.

Temporary Code Removal: Temporarily comment out or disable sections of code that are less likely to be the cause of the issue. This could include recently added features or modifications that are unrelated to the problematic area.

Test to Isolate the Issue: After making temporary changes, test the application to see if the crash persists. If the crash is still present, continue the elimination process by commenting out additional sections of code.

Gradual Restoration: If the crash disappears after commenting out specific code, gradually restore sections of the code while testing at each step. This helps identify the specific portion of code contributing to the crash.

Inspect Logs and Outputs: Review error logs, console outputs, or any other relevant diagnostic information. Look for error messages, stack traces, or patterns that can provide insights into the nature of the crash.

Check Dependencies: Examine external dependencies, libraries, or services that the application relies on. Ensure that these external components are compatible with the application's current version.

Environment Considerations: Investigate whether the crash is influenced by specific environmental factors, such as operating system versions, browser types, or network conditions. Consider testing the application in different environments.

Verify Input Data: Ensure that the input data provided to the application is valid and within expected bounds. Unexpected or malicious input can sometimes lead to crashes.

Collaborate and Seek Input: Collaborate with team members or seek input from others who may have insights into the code or the reported issue. Fresh perspectives can be valuable in the cause elimination process.

Apply a Fix: Once the root cause is identified, apply the necessary fix or code modification to address the issue.

Thorough Testing: After applying the fix, conduct thorough testing to ensure that the issue is resolved and that the fix does not introduce new problems.

Document Findings: Document the debugging process, including the steps taken to eliminate potential causes and the final resolution. This documentation can be valuable for future reference and knowledge sharing within the team.

SOFTWARE ENGINEERING

UNIT – 4

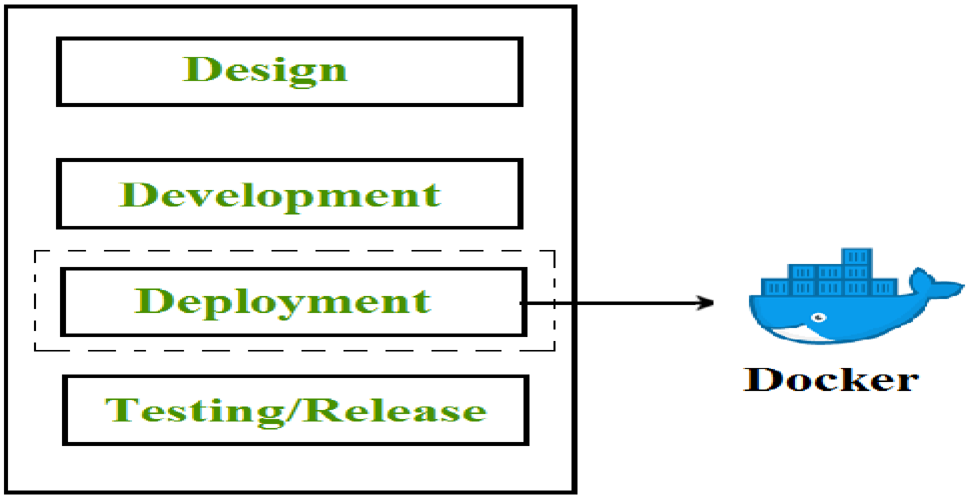
CHAPTER – 2

CONTAINERIZATION USING DOCKER AND DOCKER COMPOSE

1. INTRODUCTION TO DOCKER

Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, portable, and self-sufficient units that can run applications and their dependencies in isolated environments. Docker provides a set of tools and a platform for managing containers, making it easier to deploy and scale applications consistently across different environments.

Docker wraps up applications and all their gadgets (code, libraries, and dependencies) into neat containers, making them consistent and easy to move around. Team using DevOps can confidently deploy these containers anywhere – whether it's in development, testing, or live production. Docker helps avoid the classic "it works on my machine" problem, making sure that what the developers create is exactly what gets tested and deployed. It's like having a trustworthy sidekick that ensures applications run smoothly across different environments, making the whole development and deployment process much smoother for the entire DevOps team.



Difference between Docker and Virtual Machines

Docker:

It's like having individual lunchboxes for each dish at a potluck. Docker uses containers, which are lightweight, standalone units holding everything an application needs to run (code, libraries, etc.). Containers are quick to start, share the same OS kernel, and are highly portable, making applications consistent across different environments.

Virtual Machine (VM):

It's like having separate lunchrooms for each dish at a potluck. VMs are like mini-computers within your computer, each with its own full operating system and set of resources. VMs provide strong isolation but can be heavier, as they include a full OS. They might take longer to start and consume more resources.

Key Difference:

Docker uses lightweight containers for efficiency, while virtual machines provide stronger isolation by running multiple operating systems on a single physical machine.

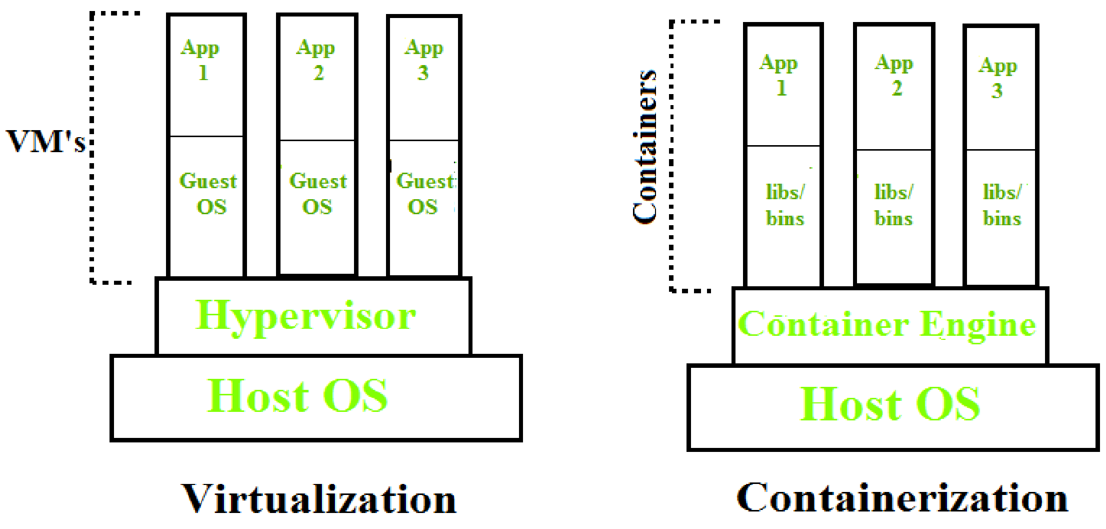
Difference between Docker Containers and Virtual Machines

Docker Containers:

It's like having a compact, self-contained lunchbox with your sandwich, juice box, and cookies (your application and its dependencies). Containers share the same cafeteria table (operating system kernel) but have their own space settings (isolated environment), making them lightweight and quick to start. They're like lunchboxes that don't need a whole new lunchroom to enjoy your meal.

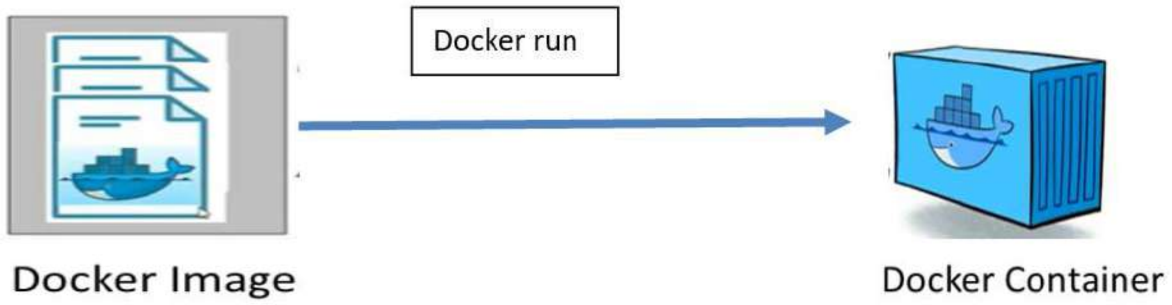
Virtual Machines (VMs):

It's like having a full, separate room (virtual machine) for each person (application). VMs are like having individual dining rooms, each with its own table, chairs, and kitchen (operating system). This makes them a bit heavier and slower to start because you're essentially running multiple mini-computers within your computer.



<u>Key components of Docker include:</u>

1. <u>Docker Engine:</u> This is the core of Docker and is responsible for creating and running containers. It consists of a server, a REST API, and a command-line interface.
2. <u>Docker Image:</u> An image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Images are used to create containers.
3. <u>Container:</u> A container is an instance of a Docker image. It encapsulates an application and its dependencies, ensuring that it runs consistently across different environments.



4. <u>Dockerfile:</u> A Dockerfile is a script that contains instructions for building a Docker image. It specifies the base image, adds application code, and defines runtime configurations.
5. <u>Docker Compose:</u> Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure the application's services, networks, and volumes.

Docker Architecture

Docker's architecture is designed to be modular and flexible. The key components in Docker's architecture:

Docker Daemon: The Docker Daemon is a background process that manages Docker containers on a host system. It listens for Docker API requests and manages Docker objects, such as images, containers, networks, and volumes.

Docker Client: The Docker Client is a command-line tool or graphical user interface that allows users to interact with the Docker Daemon. Users issue commands to the Docker Daemon using the Docker Client to build, run, and manage containers.

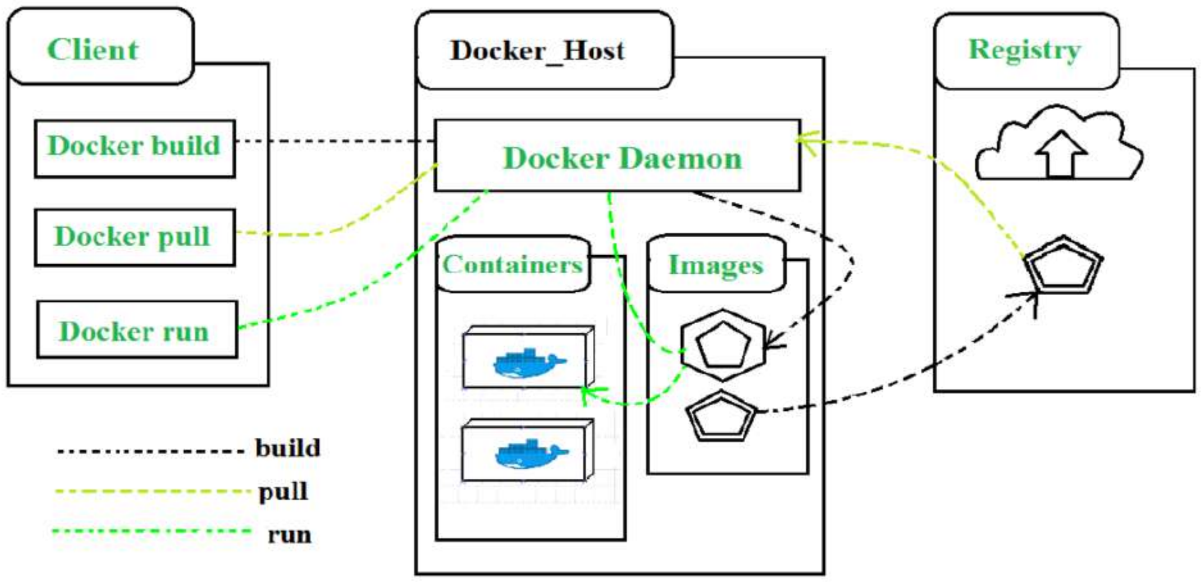
Docker Images: Docker Images are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Images serve as templates for creating containers. They are stored in a registry and can be shared and reused.

Docker Containers: Containers are instances of Docker Images, running as isolated processes on the host system. Containers encapsulate an application and its dependencies, ensuring consistency across different environments. They are portable and can run on any system that supports Docker.

Docker Registry: A Docker Registry is a storage and distribution system for Docker Images. It allows users to share and distribute Docker Images. Docker Hub is a public registry, but organizations can also set up private registries for internal use. With the help of **docker run** or **docker pull** commands, we can pull the required images from our configured registry. Images are pushed into configured registry with the help of the **docker push** command.

Docker Compose: Docker Compose is a tool for defining and running multi-container Docker applications. It uses a YAML file to configure application services, networks, and volumes, allowing users to define complex applications and their interdependencies.

Docker Swarm (Optional): Docker Swarm is Docker's native clustering and orchestration solution. It enables the creation and management of a swarm of Docker nodes, allowing users to deploy and scale applications across a cluster of machines.



Workflow:

1. Developers use the Docker Client to build and push Docker Images to a registry.
2. Operations teams use the Docker Daemon to pull these images and run containers on host systems.
3. Docker Compose can be used to define and manage multi-container applications.

2. CONTAINERIZATION USING DOCKER

Containerization in DevOps refers to the practice of packaging, distributing, and managing applications and their dependencies in lightweight, standalone containers. Containers encapsulate the application code, runtime, libraries, and system tools needed to run the software, ensuring consistency across different environments.

Key components and concepts associated with containerization include:

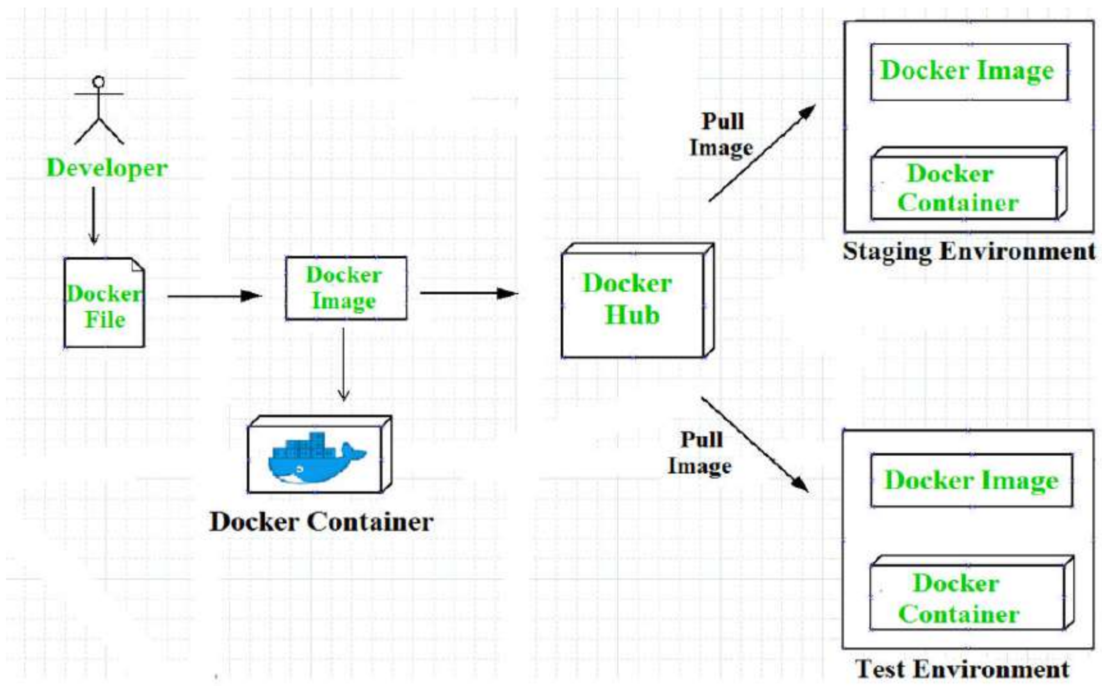
a. Container Image: A lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Container images are immutable and can be easily distributed.
b. Docker: Docker is a popular platform for containerization. It provides tools and a runtime environment to create, deploy, and manage containers. Docker images are widely used and can run on various operating systems and cloud platforms.
c. Container Orchestration: Container orchestration tools help manage the deployment, scaling, and operation of containerized applications. Kubernetes is a widely used container orchestration platform that automates the deployment, scaling, and management of containerized applications.
d. Portability: Containers ensure that applications run consistently across different environments, from development to testing to production. This portability simplifies the deployment process and reduces the likelihood of issues related to differences in underlying infrastructure.
e. Isolation: Containers provide a level of isolation between applications, ensuring that each containerized application runs independently without interfering with others on the same host. This isolation improves security and resource utilization.
f. Efficiency: Containers share the host operating system's kernel and resources, making them lightweight and efficient. They start quickly and use fewer resources compared to traditional virtual machines.

Containerization in Docker is a lightweight, portable, and consistent way to package, distribute, and run applications and their dependencies. It involves encapsulating an application and its runtime environment into a container, which is a standalone and executable software unit. Containers provide isolation for applications, making them independent of the underlying system and ensuring consistent behavior across different environments.

3. DOCKER IMAGE

There are two ways for containerization or creating images.

- Containerization of image from docker hub
- Containerization of custom image pushed into docker hub



a. <u>Containerization of image from docker hub</u>

The docker pull command helps to pull image from the docker hub

$ docker pull <image name>

1. pull docker image hello-world locally
2. First, check in the cache, if not present, get from docker hub

Once image is pulled, we can check the image by

$ docker images

<u>To create the container</u>

$ docker run <image name>

1. Then create a container for it and run

(or)

The docker run command helps to pull, create, and run the container

$ docker run <image name>

1. pull docker image locally
2. First, check in the cache, if not present, get from docker hub.
3. Then create a container for it and run

Example

$ docker run hello-world

It executes a docker image hello-world locally

b. Containerization of custom image pushed into docker hub

Dockerfile

The Dockerfile uses DSL (Domain Specific Language) and a Dockerfile is a script used to create a Docker image. It contains a set of instructions that are executed in order to build a Docker container, a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, runtime libraries, and system tools. Dockerfile is the source code of the image

Steps To Create a Dockerfile

1. Choose a Base Image: Select a base image from Docker Hub or another registry.

Example: **FROM python:3.9**

2. Set the Working Directory: Define a working directory inside the container.

Example: **WORKDIR /app**

3. Copy Application Files: Copy your application code and files into the container.

Example: **COPY. /app**

4. Expose Ports (optional): If your application listens on a specific port, expose it.

Example: **EXPOSE 80**

5. Define Default Command: Specify the default command to run when the container starts.

Example: **CMD ["python", "app.py"]**

6. Build the Docker Image: Use the docker build command to build the Docker image.

Example: **docker build -t your-image-name:tag .**

Replace your-image-name and tag with your desired image name and version tag.

7. Run a Container (optional): Use the docker run command to create and start a container based on the built image.

Example: **docker run -p 8080:80 your-image-name:tag**

Replace 8080 with the host machine's port and 80 with the exposed container port.

Dockerfile Keywords

Example:

```
# Use an official base image (in this case, the latest version of Alpine Linux)
FROM alpine:latest
# Set the working directory inside the container
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY. /app
# Install any needed packages specified in requirements.txt
RUN apk add --no-cache python3 && \
    pip3 install --upgrade pip && \
    pip3 install -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python3", "app.py"]
```

This is a simple text file, which uses predefined keywords for creating customized docker images.

Key words used in dockerfile (case sensitive)

FROM: Specifies the base image to use, it's using the latest version of Alpine Linux.

WORKDIR: Sets the working directory inside the container. All subsequent instructions will be executed from this directory.

COPY: Copies files or directories from the host machine into the container. In this example, it copies the contents of the current directory into the /app directory inside the container.

RUN: Executes a command during the image build. In this case, it installs Python 3 and any required dependencies specified in a requirements.txt file.

EXPOSE: Informs Docker that the container will listen on the specified network ports at runtime. It does not actually publish the ports.

ENV: Sets environment variables. In this example, it sets an environment variable named NAME with the value "World".

CMD: Specifies the command to run when the container starts. In this case, it runs the Python script app.py.

Creating customized docker images by using docker file.

<u>Example: 1</u>

a) Create the customized nginx image with Dockerfile

```
$vim Dockerfile

        FROM nginx
        MAINTAINER archana

:wq
```

To build an image from the dockerfile

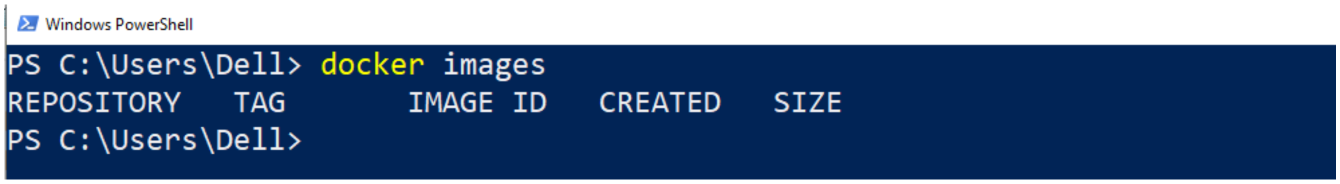
```
        $docker build -t  mynginx .
```

t - stands for tag,
. (dot) - stands for current working dir
mynginx is the new image name)

To see the image

```
        $docker images
```

Step 1 First check for images, we see there is no images

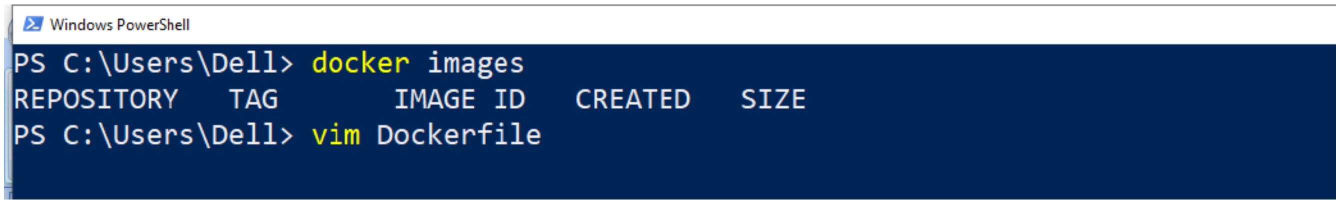

Windows PowerShell

```
PS C:\Users\Dell> docker images
REPOSITORY    TAG       IMAGE ID    CREATED    SIZE
PS C:\Users\Dell>
```

Step 2: First create Dockerfile

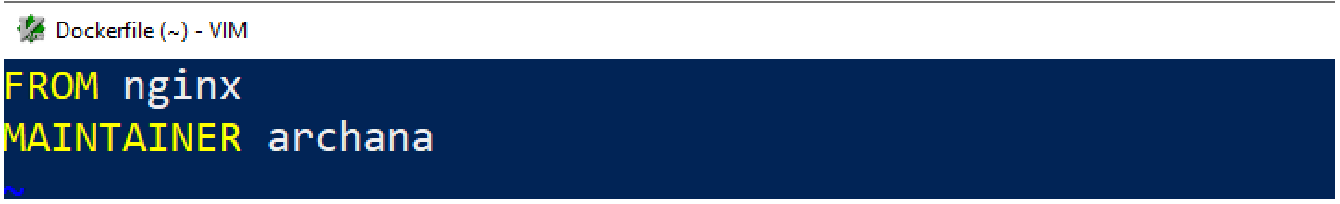
By using nano/vim/vi/otherwise Create a file named Dockerfile and navigate to that path

Here I am creating the file by vim


Windows PowerShell

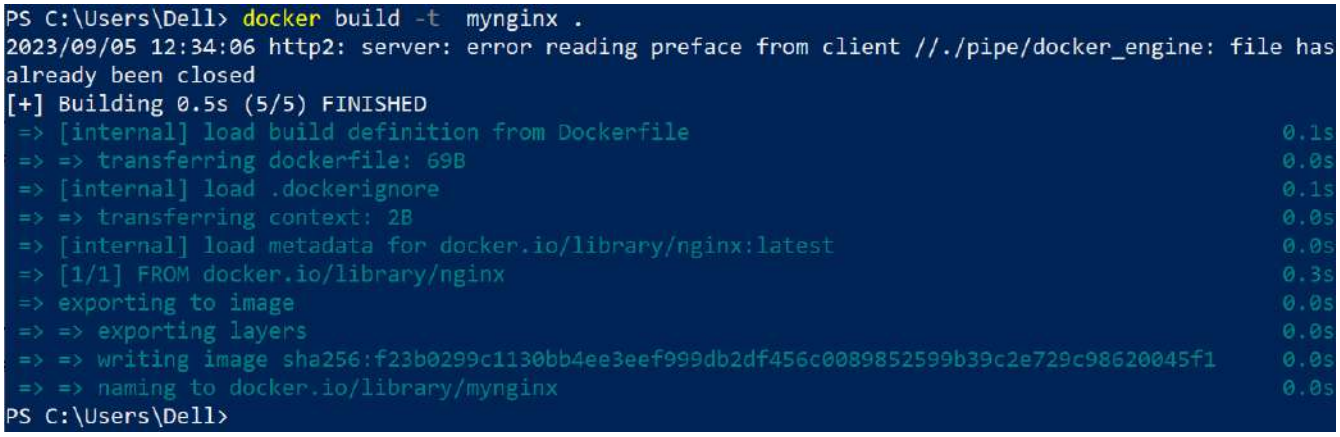
```
PS C:\Users\Dell> docker images
REPOSITORY    TAG       IMAGE ID    CREATED    SIZE
PS C:\Users\Dell> vim Dockerfile
```

Step 3: Start writing the command that is required to build the nginx image


Dockerfile (~) - VIM

```
FROM nginx
MAINTAINER archana
~
```

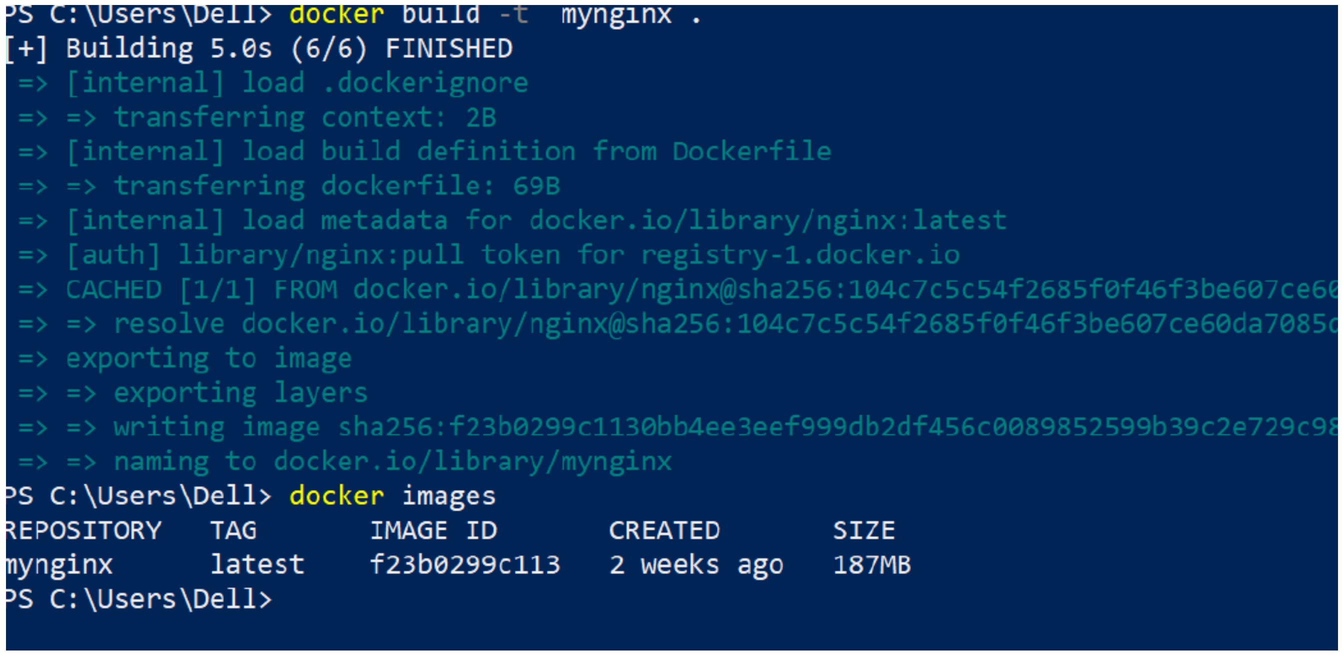
Step 4: Build Dockerfile to create image



```
PS C:\Users\Dell> docker build -t  mynginx .
2023/09/05 12:34:06 http2: server: error reading preface from client //./pipe/docker_engine: file has
already been closed
[+] Building 0.5s (5/5) FINISHED
 => [internal] load build definition from Dockerfile                                              0.1s
 => => transferring dockerfile: 69B                                                               0.0s
 => [internal] load .dockerignore                                                                 0.1s
 => => transferring context: 2B                                                                   0.0s
 => [internal] load metadata for docker.io/library/nginx:latest                                   0.0s
 => [1/1] FROM docker.io/library/nginx                                                            0.3s
 => exporting to image                                                                            0.0s
 => => exporting layers                                                                           0.0s
 => => writing image sha256:f23b0299c1130bb4ee3eef999db2df456c0089852599b39c2e729c98620045f1      0.0s
 => => naming to docker.io/library/mynginx                                                        0.0s
PS C:\Users\Dell>
```

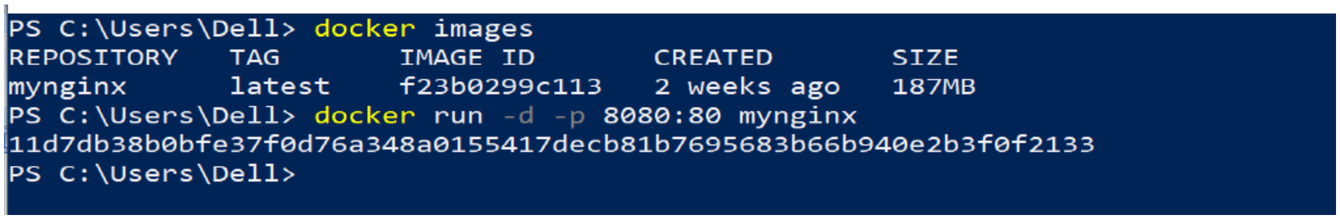
Step 5: Now let's check whether image is created or not by

$ docker images



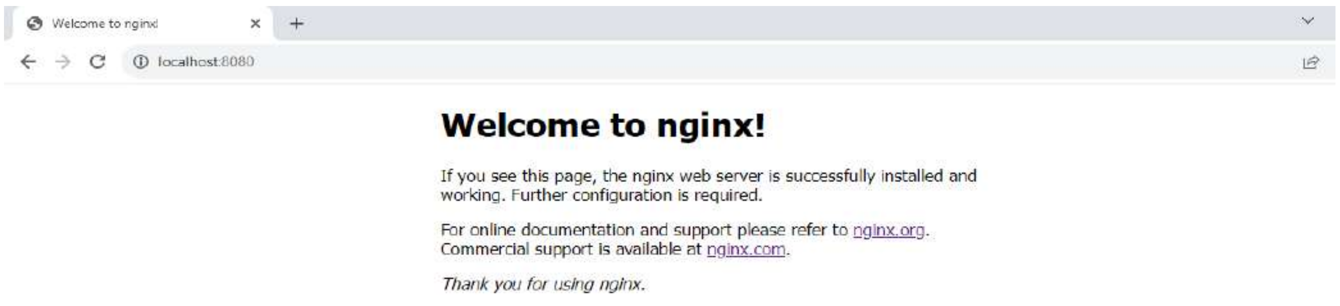
```
PS C:\Users\Dell> docker build -t  mynginx .
[+] Building 5.0s (6/6) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 69B
 => [internal] load metadata for docker.io/library/nginx:latest
 => [auth] library/nginx:pull token for registry-1.docker.io
 => CACHED [1/1] FROM docker.io/library/nginx@sha256:104c7c5c54f2685f0f46f3be607ce60
 => => resolve docker.io/library/nginx@sha256:104c7c5c54f2685f0f46f3be607ce60da7085c
 => exporting to image
 => => exporting layers
 => => writing image sha256:f23b0299c1130bb4ee3eef999db2df456c0089852599b39c2e729c98
 => => naming to docker.io/library/mynginx
PS C:\Users\Dell> docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
mynginx      latest    f23b0299c113   2 weeks ago   187MB
PS C:\Users\Dell>
```

Step 6: Run the image with image ID /image name to create a container



```
PS C:\Users\Dell> docker images
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
mynginx      latest    f23b0299c113   2 weeks ago   187MB
PS C:\Users\Dell> docker run -d -p 8080:80 mynginx
11d7db38b0bfe37f0d76a348a0155417decb81b7695683b66b940e2b3f0f2133
PS C:\Users\Dell>
```

Step7: Now check in browser with port number 8080



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Example: 2

b) Create image for installing git in ubuntu
$vim Dockerfile

```
FROM ubuntu
MAINTAINER <optional>
RUN apt-get update
RUN apt-get install -y git

:wq
```

Note: CMD -- will run when container starts.
RUN -- will executed when image is created.

```
$ docker build -t  myubuntu .
```

Let's see the images list and space consumed by our image

```
$ docker images

$ docker run -it   myubuntu
$ git  --version
# exit
```

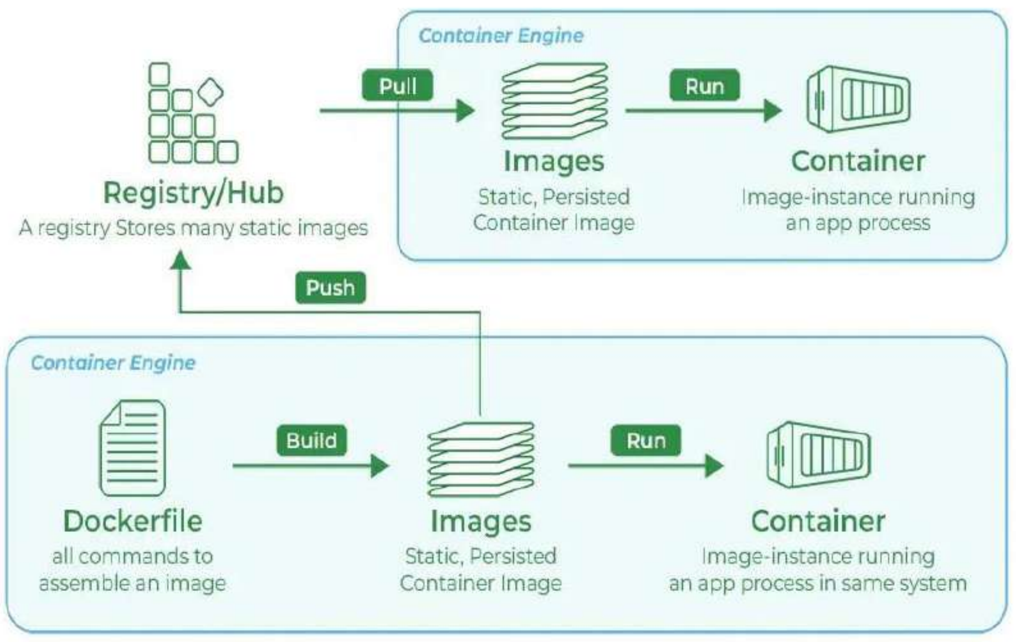
4. DOCKER HUB AND DOCKER COMPOSE

What is Docker Hub?

Docker Hub is a cloud-based registry service provided by Docker, the company behind the popular containerization platform. In the context of DevOps, Docker Hub serves as a centralized repository for Docker images, allowing developers to store, share, and manage their container images. It plays a crucial role in the containerization workflow, providing a platform for collaboration and distribution of containerized applications.

Key features of Docker Hub in DevOps include:

a. Image Repository: Docker Hub is a repository for Docker images, which are used to package applications and their dependencies into containers. Developers can push their Docker images to Docker Hub, making them accessible to other team members or the broader community.
b. Public and Private Repositories: Docker Hub supports both public and private repositories. Public repositories are openly accessible, allowing anyone to pull and use the images. Private repositories require authentication, providing a secure way for teams to store and share proprietary or sensitive container images.
c. Collaboration: Docker Hub facilitates collaboration by allowing developers to share their Docker images with others. This is especially useful in open-source projects or when working with distributed teams. Team members can easily pull and use shared images to ensure consistency in development and deployment environments.
d. Versioning: Docker Hub supports versioning of images, allowing developers to tag images with version numbers or other identifiers. This helps in tracking changes to images over time and ensures that specific versions of applications can be deployed reliably.
e. Automated Builds: Docker Hub provides a feature called "Automated Builds" that allows developers to link their Docker Hub repositories with source code repositories (e.g., on GitHub). When changes are made to the source code, Docker Hub can automatically build and update the corresponding Docker image.
f. Webhooks: Docker Hub supports webhooks, which are HTTP callbacks triggered by events such as image pushes. Webhooks enable automation and integration with other DevOps tools and systems, allowing for streamlined workflows.
g. Integration with CI/CD: Docker Hub integrates seamlessly with continuous integration and continuous deployment (CI/CD) pipelines. Developers can use Docker images stored in Docker Hub as part of their CI/CD processes, ensuring consistency between development, testing, and production environments.


Container Engine
Pull
Run
Images
Static, Persisted Container Image
Container
Image-instance running an app process
Registry/Hub
A registry Stores many static images
Push
Container Engine
Dockerfile
all commands to assemble an image
Build
Images
Static, Persisted Container Image
Run
Container
Image-instance running an app process in same system

1. **Advantages of Docker Hub**

a. Image Storage and Sharing: Docker Hub provides a centralized repository for storing and sharing Docker images.

b. Versioning and Tagging: Docker Hub supports versioning and tagging of images. This allows developers to track different versions of their images and easily reference specific versions when deploying containers.

c. Public and Private Repositories: Docker Hub offers both public and private repositories. Public repositories are accessible to anyone, Private repositories require authentication, providing a secure space.

d. Automated Builds: Docker Hub supports automated builds, which means that you can link your GitHub or Bitbucket repository to Docker Hub and automatically build and update your Docker images whenever changes are made to the source code.

e. Webhooks and Integration: Docker Hub supports webhooks, allowing you to trigger events or notifications when certain actions occur, such as a new image being pushed or an automated build completing.

f. Official Images: Docker Hub hosts a collection of official images maintained by Docker, including popular operating systems, databases, and other software components.

g. Integration with Docker CLI: Docker Hub seamlessly integrates with the Docker Command-Line Interface (CLI), making it straightforward for developers to push and pull images using familiar commands.

h. Scalability: Docker Hub can scale to accommodate the growing needs of developers and organizations.

2. Pushing image to Docker Hub

To get started with Docker Hub and push an image, we use the below two commands:

i. **push command**

ii. **tag command**

Example:

Step 1: Go to the browser and search *hub.docker.com.*

Step 2: Sign up on the Docker Hub if you do not have a docker hub account and login into Docker Hub.

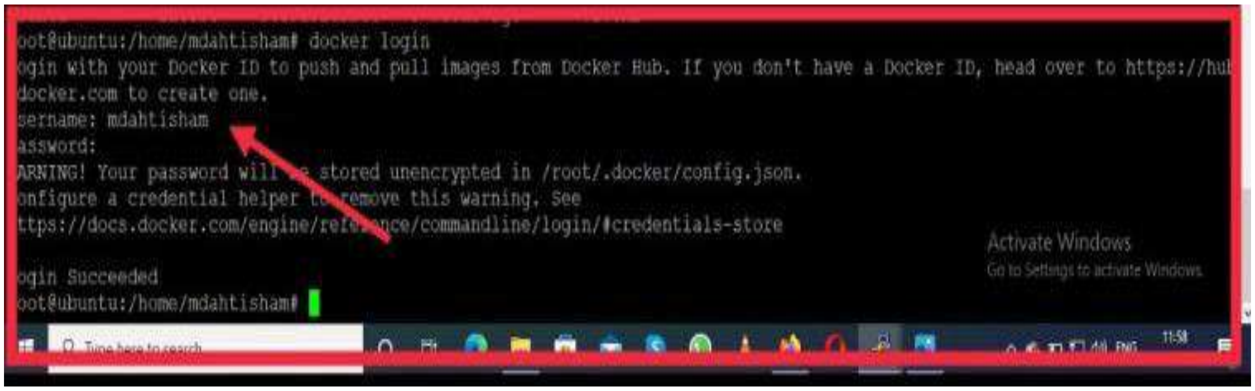
Step 3: Go back to the docker terminal and execute the below command:

```
$ docker login
```

Step 4: Then give your credentials same as your docker hub username or password.

- username
- password

Step 5: After that hit the Enter key, you will see login success on your screen.



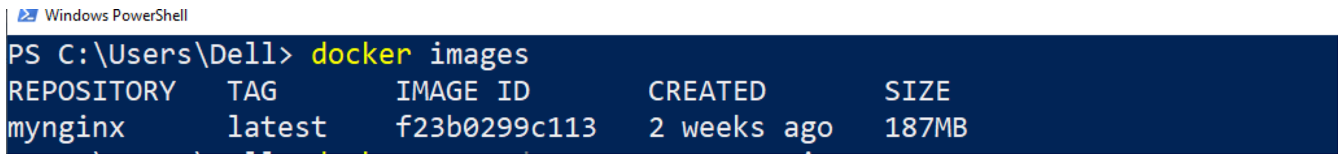
```
oot@ubuntu:/home/mdahtisham# docker login
ogin with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub
docker.com to create one.
sername: mdahtisham
assword:
ARNING! Your password will be stored unencrypted in /root/.docker/config.json.
onfigure a credential helper to remove this warning. See
ttps://docs.docker.com/engine/reference/commandline/login/#credentials-store

ogin Succeeded
oot@ubuntu:/home/mdahtisham#
```

Step 6: Now select the image that you want push into the docker hub using the following command.

$ docker images

The above command will list all the images on your system.

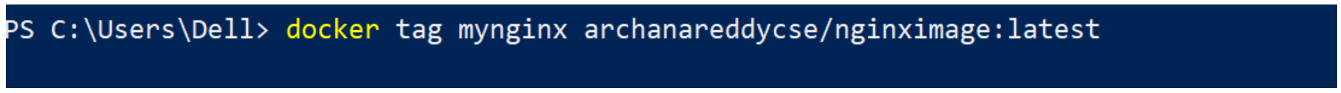


Windows PowerShell

```
PS C:\Users\Dell> docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
mynginx       latest    f23b0299c113   2 weeks ago    187MB
```

Step 7: Then type the tag, image name, docker hub username, and give the new name of the image as you want it to appear on the docker hub using the below command:

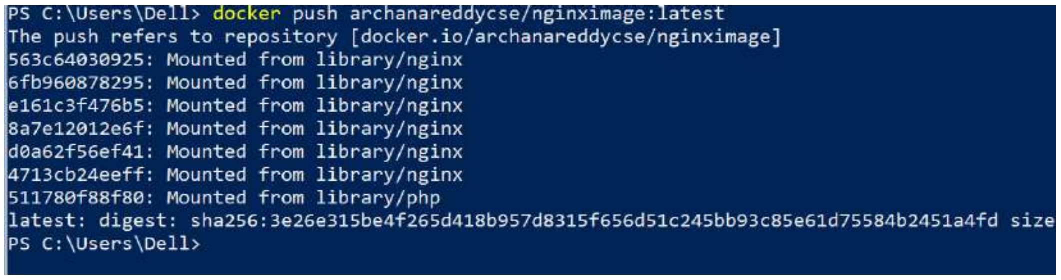
$ docker tag <image name><docker hub username>/<name that appear in docker hub>



```
PS C:\Users\Dell> docker tag mynginx archanareddycse/nginximage:latest
```

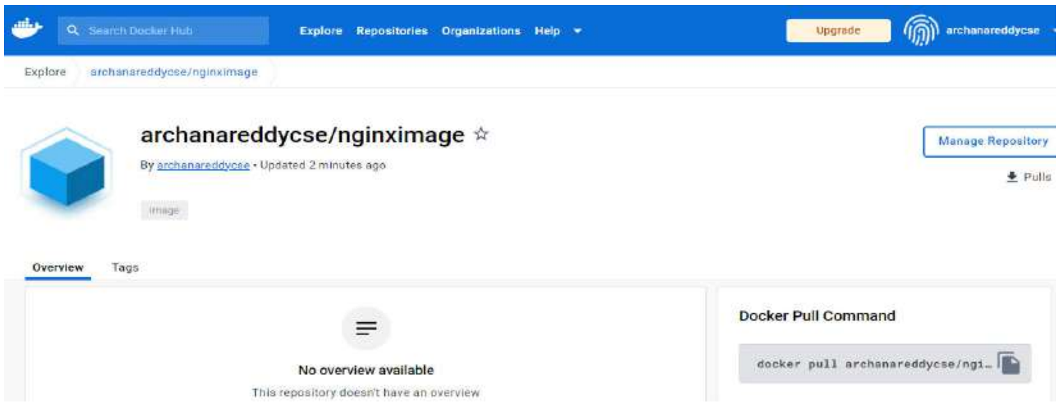
Step 8: Now push your image using the below command:

$ docker push <docker hub username>/<name that appear in docker hub>



```
PS C:\Users\Dell> docker push archanareddycse/nginximage:latest
The push refers to repository [docker.io/archanareddycse/nginximage]
563c64030925: Mounted from library/nginx
6fb960878295: Mounted from library/nginx
e161c3f476b5: Mounted from library/nginx
8a7e12012e6f: Mounted from library/nginx
d0a62f56ef41: Mounted from library/nginx
4713cb24eeff: Mounted from library/nginx
511780f88f80: Mounted from library/php
latest: digest: sha256:3e26e315be4f265d418b957d8315f656d51c245bb93c85e61d75584b2451a4fd size
PS C:\Users\Dell>
```

Note: Below you can see the Docker Image successfully pushed on the docker hub:


Search Docker Hub
Explore Repositories Organizations Help
Upgrade
archanareddycse
Explore / archanareddycse/nginximage

archanareddycse/nginximage ☆

By archanareddycse • Updated 2 minutes ago

image

Manage Repository
Pulls
Overview Tags

No overview available

This repository doesn't have an overview

Docker Pull Command

`docker pull archanareddycse/ngi…`

5. DOCKER CLI COMMANDS

1) TO CHECK DOCKER VERSION

```
$ docker –version
```

2) TO RUN an image example, HELLO-WORLD IMAGE

```
$ docker run <image name>
```

3) TO CHECK list of IMAGES

```
$ docker images
```

4) To pull an image (e.g. ubuntu) from docker hub

```
$ docker pull ubuntu
```

5) To run the image (e.g. ubuntu) as docker container

```
$ docker run -it -d ubuntu
```

6) To show list of all running and stopped containers

```
$ docker ps –a
```

To show list of only running containers

```
$ docker ps
```

7) To accessing/executing a running container (ubuntu)

```
$ docker exec -it <container id > bash
```

8) Give echo message in the running container

```
root@<container id>:/# echo hello
```

9) Exit from the running container

```
root@<container id>:/# exit
PS C:\Users>
```

10) To stop the running container

```
$ docker stop <containerID>
```

11) How to push images from local rep to docker hub rep

Step1: create DockerHub account

Step2:login into DockerHub

```
PS C:\Users >docker login
Username:<<>> (here give user name)
Password:<<>> (here give user password)
Login Succeeded
```

12) To restart image

```
$ docker start <containerID>
```

used to restart the container if it has stopped due to application crash or if error occurred

13) Working with container in interactive mode

```
$ docker container exec -it <containerID>
```

14) The containers having the same port can be bind to a different port on a host machine/ laptop. Example for ngnix

Use –p port number to bind the nginx to port 80

```
$ docker run –d – p 8080:80 nginx
```

15) Naming the container nginx to new name mynginx

```
$ docker run --name mynginx –d –p 8080:80 nginx
```

16) Running container in interactive mode

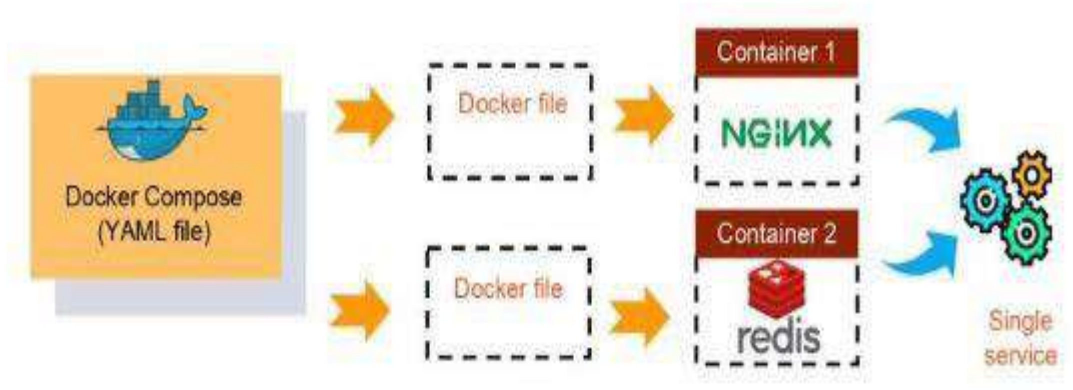
```
$ docker exec -it <container id> /bin/bash
```

Runs a container in interactive mode. Helps to see all directories, we can move to the home directory, see the virtual file system inside the container, navigate to different directories, and can see environment variables.

6. DOCKER COMPOSE

Docker Compose is a tool for defining and managing multi-container Docker applications. It allows you to define a multi-container environment in a single file, called docker-compose.yml, and then use that file to spin up the entire application stack with a single command. Docker Compose simplifies the process of managing complex applications with multiple services, dependencies, and configurations.

For example: If you have an application that requires an NGINX server and Redis database, you can create a Docker Compose file that can run both the containers as a service without the need to start each one separately.



<u>Key concepts and features of Docker Compose:</u>

<u>docker-compose.yml file:</u> This YAML file is used to define the configuration of your application's services, networks, and volumes. It includes information such as the base Docker image for each service, environment variables, ports to expose, volumes to mount, and dependencies between services.

Services: Each containerized component of your application (e.g., a web server, database, cache) is referred to as a "service" in Docker Compose. Services are defined in the docker-compose.yml file and can be built from a specified Dockerfile or pulled from a Docker image.

Networking: Docker Compose automatically creates a default network for your application, allowing services to communicate with each other using service names as hostnames. You can define custom networks in the docker-compose.yml file to control communication between services and manage container isolation.

Build and Up: The docker-compose build command builds the images for your services based on the configurations in the docker-compose.yml file. The docker-compose up command starts the containers based on the defined configuration.

Scaling: Docker Compose allows you to scale your services easily. For example, you can specify the number of replicas for a service, and Docker Compose will create the specified number of containers.

Dependencies: You can define dependencies between services in the docker-compose.yml file. This ensures that services start in the correct order.

Example of a docker-compose.yml file:

```
version: '3'

services:

  web:

    image: nginx:latest

    ports:

      - "8080:80"

  db:
```

```
  image: postgres:latest

  environment:

    POSTGRES_DB: mydatabase
```

In this example, there are two services: web and db. The web service uses the Nginx image and maps port 8080 on the host to port 80 on the container. The db service uses the latest PostgreSQL image and sets the POSTGRES_DB environment variable.

To start these services, you can run docker-compose up in the same directory as the docker-compose.yml file.

Basic Commands in Docker Compose

Docker Compose provides a set of commands that you can use to manage your multi-container applications defined in a docker-compose.yml file. Here are some basic commands:

a. docker-compose up: Start the containers based on the configurations specified in the docker-compose.yml file. If the images are not built, it builds them first.

```
docker-compose up
```

b. To run in detached mode:

```
docker-compose up -d
```

c. docker-compose down: Stop and remove the containers, networks, and volumes defined in your docker-compose.yml file.

```
docker-compose down
```

d. To also remove volumes:

docker-compose down -v

e. docker-compose ps: List the containers that are part of your Docker Compose application along with their status.

docker-compose ps

f. docker-compose logs: View the logs generated by the services.

docker-compose logs -f

g. To follow the logs in real-time:

docker-compose logs -f

h. docker-compose exec: Run commands in a running container.

docker-compose exec <service-name> <command>
Example:
docker-compose exec web bash

i. docker-compose build: Build or rebuild services. This is useful if you've made changes to your Dockerfiles or other build-related configurations.

docker-compose build

j. To build a specific service:

docker-compose build <service-name>

k. docker-compose pull: Pull images defined in your docker-compose.yml file without starting the services.

docker-compose pull

l. docker-compose scale: Scale services to a specified number of containers.

docker-compose scale <service-name>=<number-of-containers>

Example:

docker-compose scale web=3

m. docker-compose config: Validate and view the composed Dockerfile.

docker-compose config

7. MULTICONTAINER USING DOCKER COMPOSE:

Create a simple example of using Docker Compose to set up WordPress with MySQL. This example will consist of two services: one for WordPress and one for MySQL.

Step 1: Create a Project Directory

Create a new directory for your Docker Compose project. For example, you can create a directory called wordpress-docker:

```
mkdir wordpress-docker

cd wordpress-docker
```

Step 2: Create a Docker Compose YAML File

Create a file named docker-compose.yml in your project directory. You can use a text editor like Notepad or Visual Studio Code to create and edit this file.

```
version: '3'

services:

  wordpress:

    image: wordpress:latest

    ports:

      - "8080:80"

    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_USER: root
      WORDPRESS_DB_PASSWORD: example
      WORDPRESS_DB_NAME: wordpress
    volumes:
      - wordpress_data:/var/www/html
  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example
      MYSQL_DATABASE: wordpress
    volumes:
      - mysql_data:/var/lib/mysql
volumes:
  wordpress_data:
  mysql_data:
```

Explanation of the above docker-compose.yml file:

Version:

version: '3': This line specifies the version of the Docker Compose file format being used.

Services: The services section defines the different parts of your application.

WordPress Service:

Image: wordpress:latest - This service uses the latest version of the WordPress image, which is like a pre-packaged set of files and configurations needed to run WordPress.

Ports: It makes WordPress accessible from your computer's browser through port 8080.

Environment: These are settings WordPress needs to connect to its companion MySQL database (like the database host, user, password, and name).

Volumes: This is where WordPress stores its data, such as posts and settings. The wordpress_data volume is like a shared folder between your computer and the WordPress container.

MySQL Service:

Image: mysql:5.7 - This service uses version 5.7 of the MySQL image.

Environment: These are settings needed to set up the MySQL database, including the root password and the name of the WordPress database.

Volumes: Similar to WordPress, this is where MySQL stores its data. The mysql_data volume is like a shared folder.

Volumes: The volumes section defines named volumes, which are like special folders used by the services to store data. These volumes persist even if the containers are stopped or removed.

This docker-compose.yml file sets up two things:

WordPress Service: The part of your application that shows your website. It needs a database to store your site's information.

MySQL Service: The database part of your application where WordPress stores things like posts, comments, and settings.

The file tells Docker how to connect these two services and where to store their data. When you run this setup with docker-compose up, it's like starting your website and its database. You can then access your WordPress site in a web browser at http://localhost:8080.

Step 3: Start the Services

Open a command prompt or PowerShell in the project directory and run the following command to start the services:

```
docker-compose up -d
```

This command will download the required images (if not already downloaded) and start the WordPress and MySQL containers in the background.

Step 4: Access WordPress

Open a web browser and navigate to http://localhost:8080. You should see the WordPress setup page. Complete the WordPress installation by providing the necessary information.

Step 5: Stop and Remove Containers

When you're done working with the containers, you can stop and remove them using the following command:

```
docker-compose down
```

This will stop and remove the WordPress and MySQL containers. The data volumes (wordpress_data and mysql_data) will persist, so your WordPress installation will be preserved.

SOFTWARE ENGINEERING

UNIT – 4

CHAPTER – 3

INTRODUCTION TO ORCHESTRATION USING KUBERNETES

I. UNDERSTANDING CONTAINER ORCHESTRATION

Definition of Orchestration

Orchestration in DevOps refers to the coordination and automation of different tasks and processes involved in the development, deployment, and management of software applications. It involves organizing and controlling various elements of the software delivery lifecycle, such as provisioning infrastructure, configuring servers, deploying code, and managing dependencies.

Think of orchestration as the conductor in an orchestra. The conductor ensures that each instrument plays its part at the right time and in harmony with others, creating a well-organized and synchronized performance. Similarly, in DevOps, orchestration tools help coordinate and automate the different components of software development and delivery, ensuring that everything works together seamlessly.

An orchestration tool automates and organizes various processes, ensuring they are executed in the right order and efficiently. It helps manage complex workflows, deploy applications, allocate resources, and handle other tasks, making it easier for systems to run smoothly and consistently. Essentially, it's the behind-the-scenes coordinator that ensures all the parts of a system or process collaborate effectively.

Definition of Container Orchestration

Container orchestration is like managing a group of containers (small, lightweight, standalone software packages that include everything needed to run a piece of software) to work together smoothly. It helps automate and streamline the deployment, scaling, and management of these containers, making it easier for developers to build, ship, and run applications consistently across different environments.

Container orchestration tools, like Kubernetes, Docker Swarm, or Amazon ECS, handle tasks such as deploying containers, managing their lifecycle, balancing their workload, and making sure they can talk to each other correctly. This makes it easier for developers to build and run applications consistently, no matter where they are deployed.

II. KUBERNETES CORE CONCEPT

Definition of Kubernetes

Kubernetes is like a smart manager for your software applications. Imagine you have a bunch of workers (containers) doing different tasks for your application. Kubernetes is the boss that takes care of where these workers should work, how many of them should be working, and makes sure they're all doing their jobs correctly.

It helps in deploying your application, scaling it up or down based on the workload, and managing any problems that might come up, like if a worker stops working, Kubernetes can replace it. So, instead of manually handling each part of your application, you let Kubernetes handle the management, making your life as a developer or IT person much easier.

Definition: “Kubernetes is an open-source container orchestration platform. It automates the deployment, scaling, and management of containerized applications. It allows you to run your applications on a cluster of machines, abstracting away the underlying infrastructure complexities and providing a unified way to manage and deploy your software.”

kubernetes

"K8s" is an abbreviation for Kubernetes. The "8" in "K8s" represents the eight letters between "K" and "s" in the word "Kubernetes."

Explanation of Definition:

1. Containerized Applications: Applications are packaged into containers, which are self-contained units that include all the necessary software and dependencies to run the application.
2. Orchestration: Kubernetes acts like a manager for these containers. It helps in deploying containers to a cluster of machines, scaling them up or down based on demand, and managing their lifecycle.
3. Cluster: Kubernetes operates on a cluster of machines (physical or virtual), turning them into a unified computing resource. Each machine in the cluster is referred to as a node.
4. Automation: Kubernetes automates many tasks involved in deploying and managing applications. It ensures that containers are running, replaces failed containers, distributes the load between containers, and more.
5. Declarative Configuration: Instead of telling Kubernetes how to do each task, you declare the desired state of your application, and Kubernetes works to make the actual state match the desired state.

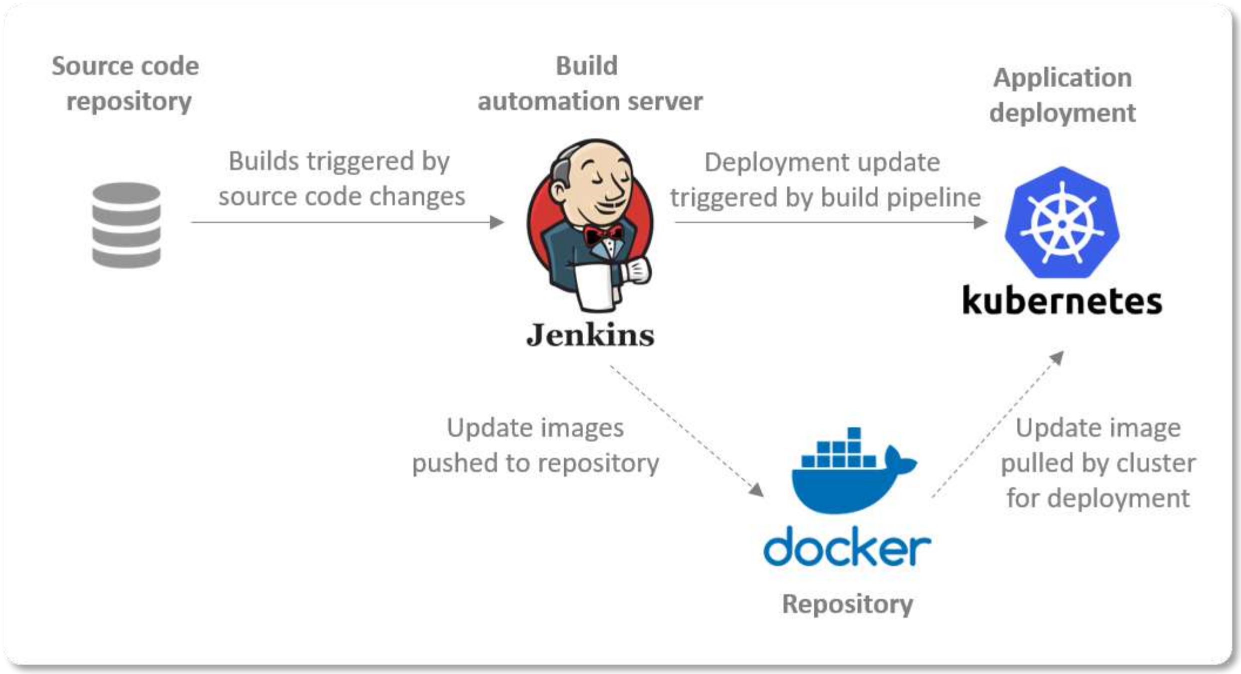
Kubernetes has become a popular tool in the world of containerized applications because it provides a standardized way to manage and scale applications, making it easier for developers to deploy and operate their software consistently across various environments.

Role of Kubernetes in DevOps

In DevOps, Kubernetes is often used to streamline the deployment and management of software applications. Here's where Kubernetes fits into DevOps:

1. Deployment Automation: Kubernetes automates the deployment process of applications. Instead of manually installing and configuring software on servers, developers can define how their applications should run in Kubernetes, and the platform takes care of deploying and managing them.
2. Scaling Applications: DevOps teams use Kubernetes to scale applications easily. As the demand for an application increase, Kubernetes can automatically add more instances (containers) to handle the load, and when the demand decreases, it can scale down to save resources.

3. Resource Optimization: Kubernetes helps optimize the use of computing resources. It efficiently distributes containers across the available infrastructure, ensuring that applications run smoothly and that resources are utilized effectively.
4. Consistent Environments: Kubernetes provides a consistent environment for applications to run, whether they are deployed on a developer's laptop, a testing server, or a production environment. This consistency reduces the chances of errors caused by differences in environments.
5. Fault Tolerance: Kubernetes improves the resilience of applications. If a container or a server fails, Kubernetes can automatically replace or reschedule the affected containers to maintain the desired state of the application.
6. Rolling Updates: When there's a need to update an application, Kubernetes supports rolling updates, allowing new versions of the application to be deployed gradually without causing downtime.

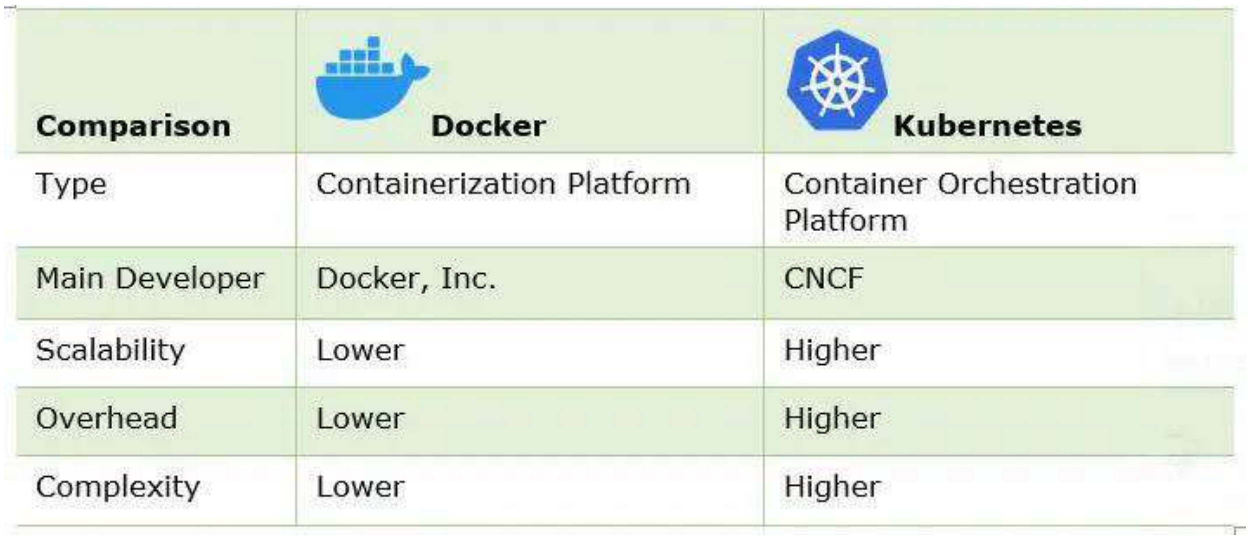

Source code repository
Build automation server
Application deployment
Builds triggered by source code changes
Jenkins
Deployment update triggered by build pipeline
kubernetes
Update images pushed to repository
Update image pulled by cluster for deployment
docker
Repository

Difference between Kubernetes and Docker

Docker: Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Docker helps create and manage these containers, making it easy to package and distribute applications.

Kubernetes: Kubernetes is a container orchestration platform. It helps manage and coordinate multiple containers that make up an application, automating tasks like deployment, scaling, and load balancing. While Docker is focused on creating and running individual containers, Kubernetes is focused on orchestrating and managing the lifecycles of those containers in a cluster of machines.

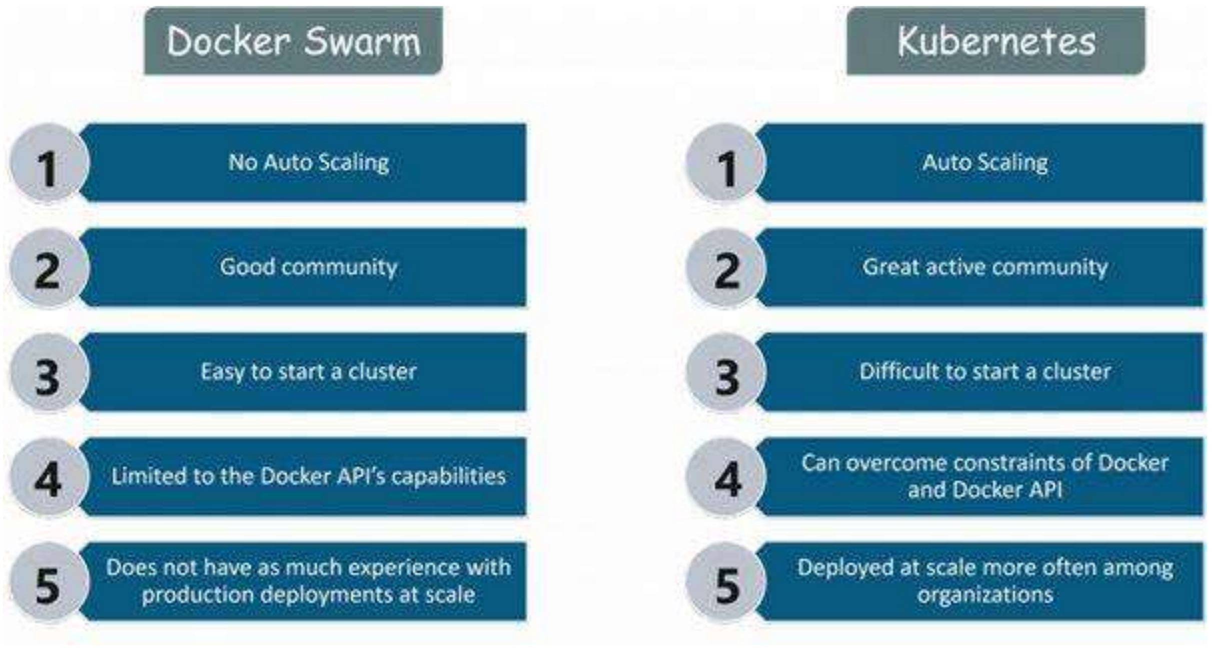
Docker is like a tool to build and run containers, while Kubernetes is like a manager that oversees and organizes how those containers work together in a larger system. They can be used together, with Docker providing the containers and Kubernetes orchestrating their deployment and operation.



Comparison	**Docker**	**Kubernetes**
Type	Containerization Platform	Container Orchestration Platform
Main Developer	Docker, Inc.	CNCF
Scalability	Lower	Higher
Overhead	Lower	Higher
Complexity	Lower	Higher

Difference between Kubernetes and Docker Swarm

Kubernetes and Docker Swarm are both container orchestration platforms, but they have differences in their architectures and features. Think of Kubernetes as a sophisticated conductor orchestrating a large symphony, while Docker Swarm is like a conductor leading a smaller chamber orchestra. Kubernetes is more comprehensive and scalable, suitable for complex, large-scale applications. It offers advanced features like automatic load balancing, rolling updates, and a vast ecosystem of tools. On the other hand, Docker Swarm is simpler and more user-friendly, making it easier for smaller projects or those getting started with containers. For example, if you have a handful of containers representing different parts of a website, Docker Swarm might be a straightforward choice. However, if you're managing a complex microservices architecture with dozens or hundreds of containers with diverse requirements, Kubernetes provides a more robust solution.



	Docker Swarm	Kubernetes
1	No Auto Scaling	Auto Scaling
2	Good community	Great active community
3	Easy to start a cluster	Difficult to start a cluster
4	Limited to the Docker API's capabilities	Can overcome constraints of Docker and Docker API
5	Does not have as much experience with production deployments at scale	Deployed at scale more often among organizations

<u>Features of Kubernetes</u>

The various features are:

a) <u>Container Orchestration:</u> Manages the deployment, scaling, and operation of application containers, making it easy to run and scale applications in various environments.

b) <u>Automated Load Balancing:</u> Distributes network traffic across multiple containers to ensure even utilization and prevent any one container from being overloaded.

c) <u>Scaling:</u> Allows automatic scaling of the number of containers based on demand, ensuring applications have enough resources during periods of high traffic.

d) <u>Self-healing:</u> Monitors the health of containers and automatically replaces or restarts failed containers, maintaining the desired state of the application.

e) <u>Rolling Updates:</u> Enables seamless updates of applications with zero downtime, allowing for continuous delivery and integration.

f) <u>Declarative Configuration:</u> Defines the desired state of an application and Kubernetes works to ensure the actual state matches it, simplifying configuration and reducing manual intervention.

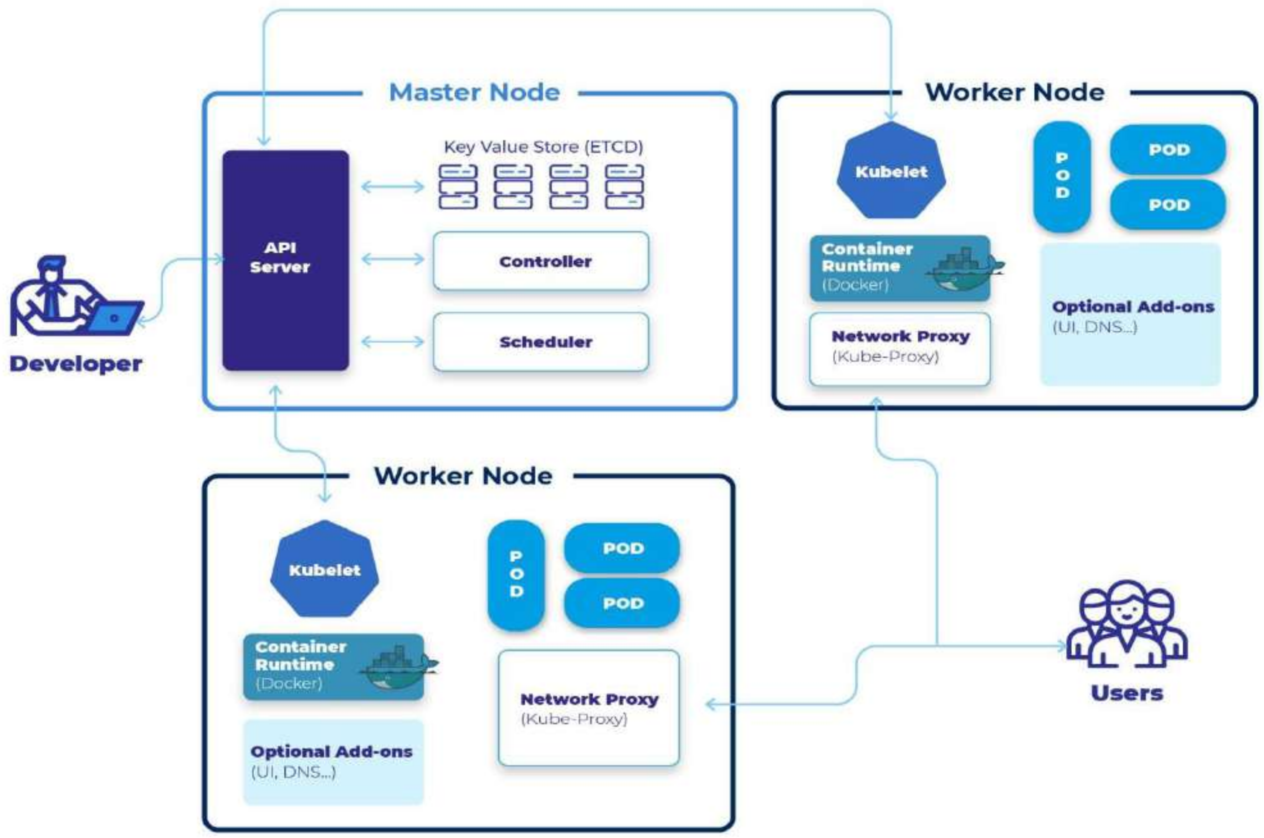
g) <u>Service Discovery and Load Balancing:</u> Automatically discovers and manages the network endpoints of services, facilitating communication between containers.

h) Storage Orchestration: Manages storage for containers, allowing them to persist data and be dynamically provisioned based on application needs.

i) Secrets and Configuration Management: Safely manages sensitive information and configuration parameters, preventing exposure of sensitive data.

j) Multi-Cloud and Hybrid Environments: Provides flexibility by allowing deployment across different cloud providers or on-premises infrastructure, supporting hybrid cloud setups.

Kubernetes Architecture


Master Node
Key Value Store (ETCD)
API Server
Controller
Scheduler
Developer
Worker Node
Kubelet
POD
POD
POD
Container Runtime (Docker)
Network Proxy (Kube-Proxy)
Optional Add-ons (UI, DNS...)
Worker Node
Kubelet
POD
POD
POD
Container Runtime (Docker)
Network Proxy (Kube-Proxy)
Optional Add-ons (UI, DNS...)
Users

Master Node: Think of the master node as the brain of the operation. It makes decisions about the overall state of your applications. It manages the cluster, schedules applications, and ensures that the desired state (what you want your apps to look like) matches the actual state (how they are currently running).

Worker Nodes: These are like the hands and feet that do the actual work. Worker nodes host the containers that make up your applications. They communicate with the master node and execute the tasks assigned to them.

Pods: Pods are the smallest deployable units in Kubernetes. You can think of them as the basic building blocks for your applications. Each pod can contain one or more containers that work together.

Controller: Controllers are like supervisors that manage the desired state of your applications. They continuously watch over your pods, ensuring the right number of them are running and replacing any that fail.

Service: Services help your pods communicate with each other. You can think of a service as a stable network endpoint that your applications can use to find and talk to each other.

Kubelet: Kubelet is like a worker on each node. It takes care of making sure that the containers in a pod are running and healthy. It communicates with the master node to report the status of its containers.

Kube Proxy: Kube Proxy helps with network communication. It makes services accessible to the external world and manages the network routing to the pods.

The master node is the boss making decisions, the worker nodes are the doers running your applications, pods are your applications' building blocks, controllers keep things in check, services help with communication, and kubelet/kube proxy are the workers ensuring everything runs smoothly. Together, they make up the Kubernetes architecture, coordinating the deployment and management of your containerized applications.

MiniKube

Minikube is a lightweight, easy-to-use way to run Kubernetes on your own machine for learning, development, and testing purposes.

Minikube is like a small, personal playground for Kubernetes. It is a miniature version of the Kubernetes system that you can run on your own computer. It allows you to experiment, learn, and develop with Kubernetes without needing a big, complex setup.

So, instead of dealing with a full-blown Kubernetes cluster that spans multiple machines, Minikube lets you have a tiny, single-node cluster right on your laptop. It's a great tool for developers who want to test their applications in a Kubernetes environment locally before deploying them to a larger-scale Kubernetes cluster.

Kubernetes is like a powerful system that manages and orchestrates containers for deploying and running applications, while Minikube is a tool that helps you run a small, simplified version of Kubernetes on your own computer.

III. DEPLOY PODS

A pod in Kubernetes is like a tiny, self-contained unit that holds one or more containers. Think of it as the smallest, deployable building block in Kubernetes. A Kubernetes pod hosts containers that work together on a specific task or service within your application.

Containers inside a pod share the same network space and can easily communicate with each other. This makes it convenient for them to collaborate on tasks, share data, or provide different functionalities needed by your application. So, if you have a group of containers that need to work closely together, you put them in the same pod, ensuring they can talk to each other easily.

What happens in Minikube?

Minikube is a tool that allows you to run a single-node Kubernetes cluster on your local machine.

While Docker is excellent for containerization and local development, Kubernetes complements it by providing advanced orchestration and management capabilities, making it well-suited for complex, scalable, and production-grade containerized applications.

Minikube is like a magic box that lets you create a tiny, personal version of a whole city for your computer programs. This city is called Kubernetes, and it's a powerful system that helps organize and run your applications.

When you start Minikube, it's like opening that magic box. Minikube sets up a small, virtual city on your computer, complete with its own little streets (nodes) and houses (containers). These containers are where your applications live.

So, when you deploy pods (which are groups of containers) using Minikube, it's like placing houses in your miniature city. You tell Minikube what kind of houses (containers) you want, and it takes care of putting them in the right places, making sure they have what they need to run, and even connecting them to each other so they can talk.

When you're done playing with your miniature city, you can close Minikube, and it neatly packs up the little streets and houses until you're ready to open it again. It's a convenient way to learn and experiment with Kubernetes without needing a full-sized city.

What is deploying pods?

Deploying pods in Minikube means creating and running instances of your containerized applications within a local Kubernetes environment provided by Minikube.

Key concepts:

a) Pods: In Kubernetes, a pod is the smallest deployable unit and can contain one or more containers. Containers within a pod share the same network namespace, allowing them to easily communicate with each other. Pods are the basic building blocks of applications in Kubernetes.

b) Minikube: Minikube is a tool that enables you to run a single-node Kubernetes cluster on your local machine. It's a lightweight and convenient way to experiment with Kubernetes without the need for a full-scale, multi-node cluster.

c) Deploying Pods in Minikube: When you deploy pods in Minikube, you are essentially instructing Minikube to create and manage instances of your applications. This involves specifying the configuration for your pods (such as the container image, environment variables, and other settings) and using kubectl (the Kubernetes command-line tool) to apply this configuration to the Minikube cluster.

Example:

```
# Create a pod with a simple container

kubectl create deployment mypod --image=nginx:latest
```

This command tells Minikube to create a deployment named mypod using the Nginx container image. Minikube then takes care of scheduling the pod, pulling the necessary container image, and ensuring that the pod is running as per the specified configuration.

Deploying pods in Minikube involves using Kubernetes concepts (like pods and deployments) within the context of a local, single-node Kubernetes cluster provided by Minikube. It allows you to test and develop your containerized applications in an environment that simulates a Kubernetes cluster.

Process of MiniKube

Terms to understand in the process of Minikube: Image, Container, Pod, Node and Cluster

a. Image: An image is like a packaged box containing everything needed to run a specific application. It includes the application code, libraries, and other dependencies. In Minikube, you use container images to define what should run inside your containers. These images are often stored in a registry, like Docker Hub, and are pulled by Minikube when needed.

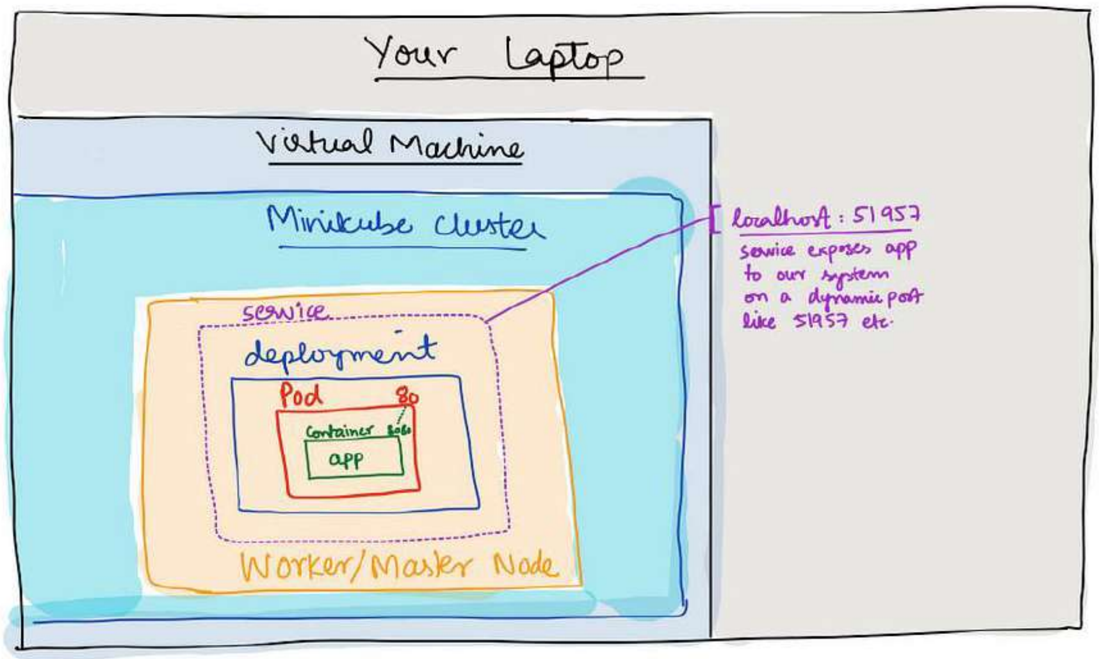
b. Container: A container is like an executable instance of an image. It's a running process that encapsulates the application and its dependencies in an isolated environment. Containers in Minikube are created based on the images you specify. They provide a consistent and isolated environment for your application to run.

c. Pod: A pod is like a small, self-contained unit that can host one or more containers. Containers within a pod share the same network and storage, making it easy for them to communicate and work together. In Minikube, you deploy pods to run your applications. Each pod encapsulates one or more containers, allowing them to collaborate closely.

d. Node: A node is like a worker machine in a cluster. It's a physical or virtual machine that runs your containers and manages their lifecycle. Minikube runs a single-node Kubernetes cluster on your local machine. In this context, the node is your local machine, and it hosts the pods and containers you deploy with Minikube.

e. Kubernetes Cluster: a Kubernetes cluster in Minikube is like a small, self-contained universe where you can run and manage your containerized applications. It's a way to simulate a tiny version of a larger Kubernetes setup on your own computer.

In Minikube, you use container images to define what runs inside containers. Containers are instances of these images, and pods are groups of containers that work together. The node, in Minikube's case, is your local machine hosting the Kubernetes cluster. So, images are like packaged applications, containers are running instances of those applications, pods are units that group containers, and nodes are the machines where these pods and containers run in Minikube.



1. Imagine you have a special place (Minikube cluster) on your computer where you can play with your apps using a tool called Kubernetes.
2. You've put your app inside a small team (pod) that has one worker (container) running your app's code.
3. To let the outside world interact with your app, you've set up a system (Service object) that gives your app a unique address (external IP).
4. Your app's worker is listening on a particular door (port 8080), and within the team (pod), that door is connected to another door (port 80).
5. There's someone (Kubernetes) keeping an eye on your app. If many people want to use your app, this someone can get more workers (scale up) to handle the workload.
6. If, by chance, the worker gets tired and stops working (crashes), this someone can quickly get a new worker to replace the tired one.

You've set up your app to work in a special play area (Minikube) where it can talk to the world, someone is watching over it to make sure it behaves well, and if needed, more workers can join in to help out. It's like a small, controlled environment to test and develop your apps with a safety net in case anything goes wrong.

IV. CREATE DEPLOYMENTS TO MANAGE PODS

Minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes. All you need is Docker (or similarly compatible) container or a Virtual Machine environment, and Kubernetes is a single command away: minikube start

We need

- 2 CPUs or more
- 2GB of free memory
- 20GB of free disk space
- Internet connection
- Container or virtual machine manager, such as: Docker, QEMU, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, or VMware Fusion/Workstation

Installation

- To install the latest minikube stable release on x86-64 Windows using .exe download:
- Download and run the installer for the latest release.
- Run the installer to complete the installation process
- Set the path for **minikube** in the Environment variables
- Start your cluster by running the following command in PowerShell with administrator access (but not logged in as root), run:

```
> minikube start [Docker service should be running]
```

- Next command is used to retrieve the IP address of the Minikube cluster. This IP address is the external IP through which you can access services running within your Minikube cluster.

```
> minikube ip
```

Example:

```
$ minikube ip
192.168.49.2
```

This IP address is useful when you need to access services or applications running inside your Minikube cluster from your host machine. The IP address may vary depending on your Minikube setup and the underlying hypervisor (such as VirtualBox, Hyper-V, or Docker).

- Next is the kubectl get po -A command in Minikube is used to retrieve information about pods across all namespaces in your Kubernetes cluster running within Minikube. Here's a breakdown of the command:

kubectl: The command-line tool for interacting with Kubernetes clusters.

get: Retrieves one or more resources.

po: Short for "pods," specifies the type of resource to retrieve.

-A: Specifies that the command should operate on all namespaces.

Here's how you can use this command:

Open a terminal (Command Prompt, PowerShell, or any terminal emulator).

Run the following command:

```
kubectl get po -A
```

This command fetches information about all pods (po) in all namespaces (-A) within your Minikube cluster. The output will include details such as pod names, namespaces, status, and more.

The Minikube typically creates a default namespace called default where your pods are deployed if you haven't specified a different namespace for your resources. The -A flag ensures that you get information from all namespaces, including the default one.

Steps to Create Deployments:

Creating deployments to manage pods in Minikube involves defining a deployment configuration and applying it using **kubectl**, the Kubernetes command-line tool. Below are detailed steps:

Step 1: Start Minikube

Ensure that Minikube is running. If not, start it using:

```
minikube start
```

Step 2: Create a Deployment Configuration

Create a file named my-deployment.yaml with the following content. This example uses an Nginx container:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
```

```
image: nginx:latest

ports:

- containerPort: 80
```

Explanation of the YAML code:

a. apiVersion: apps/v1: Think of it as a way to tell Kubernetes which "version" of its tools you want to use. It's like saying, "Hey, Kubernetes, I'm using the 'apps' tools, version 1."
b. kind: Deployment: This is like telling Kubernetes, "I want you to manage my application using a tool called 'Deployment.' Think of it as a supervisor for running multiple copies of my application."
c. metadata: Just some basic information about your deployment. In this case, you named it my-deployment.
d. spec: This is where you describe how you want your application to run.

replicas: 3: You're saying, "I want three identical copies (replicas) of my application running at the same time."

selector: You're specifying how Kubernetes should identify which parts of your application it should manage.

matchLabels: You're telling Kubernetes, "Look for things labeled with 'app: my-app' and manage those."

template: Now you're describing the template for each copy (pod) of your application.

metadata: Again, just some labels for your pods. You're saying, "Label each pod with 'app: my-app.'"

spec: Describes what's inside each pod.

containers: This is where you say, "Each pod contains one thing, and that thing is a container."

name: my-container: The container is called 'my-container.' It's like giving a name to a box that holds your application.

image: nginx:latest: You're specifying, "Inside this box (container), I want to run the latest version of Nginx, a popular web server."

ports: Finally, you're saying, "Nginx inside my container will listen for requests on port 80."

So, this YAML code is like telling Kubernetes: "Hey, I've got an application, and I want you to make sure there are three copies of it running all the time. Each copy is like a box containing the latest Nginx web server, and they should be labeled with 'app: my-app.'"

This configuration specifies a deployment named my-deployment with three replicas, each running a Nginx container.

Step 3: Apply the Deployment Configuration

Apply the deployment configuration to create the deployment and pods:

```
kubectl apply -f my-deployment.yaml
```

The command kubectl apply -f my-deployment.yaml is like telling Kubernetes, "Hey, I've got a plan for how I want my application to run. Please make it happen."

kubectl: Think of it as your tool to talk to the Kubernetes system. It's like a magic wand that lets you command your Kubernetes cluster.

apply: This is the action you want to perform. You're saying, "I want to apply a set of instructions to the cluster."

-f my-deployment.yaml: Here you're pointing to a file (my-deployment.yaml) that contains your instructions in a special format called YAML. It's like handing over a document that says, "This is how I want things to be."

So, when you run kubectl apply -f my-deployment.yaml, you're giving Kubernetes the instructions on how to set up your application. In this case, the instructions are to create a deployment named my-deployment with three copies of your application, each running the latest version of Nginx. Kubernetes takes care of making sure this plan becomes a reality in your cluster.

Step 4: Verify Deployment and Pods

Check the status of the deployment:

```
kubectl get deployments
```

You should see your deployment (my-deployment) and the desired number of replicas.

The command kubectl get deployments is like asking Kubernetes, "Tell me about the teams of my applications."

kubectl: Think of it as your tool to communicate with Kubernetes. It's like talking to the manager of your application teams.

get: This is the action you want to perform. You're saying, "I want to get information."

deployments: You're specifying what kind of information you're interested in. In this case, it's information about the teams managing your applications, known as Deployments in Kubernetes.

So, when you run kubectl get deployments, you're asking Kubernetes to show you a list of all the application teams (Deployments) in your cluster. The information will include details like the team names, how many members (replicas) are in each team, and the current status of each team. It's a quick way to check on the health and status of your deployed applications in the Kubernetes cluster.

Check the status of the pods:

```
kubectl get pods
```

You should see the pods created by the deployment.

The command kubectl get pods is like asking Kubernetes, "Tell me about my workers."

kubectl: Your tool for talking to Kubernetes. It's like your communicator with the manager of your worker bees (pods).

get: This is the action you want to perform. You're saying, "I want to get information."

pods: You're specifying what kind of information you're interested in. In this case, it's information about your worker bees, which are called pods in Kubernetes.

So, when you run kubectl get pods, you're asking Kubernetes to show you a list of all your worker pods. The information will include details like the names of the pods, whether they are working well, and other status information. It's a quick way to check on the health and status of the individual components (pods) of your applications in the Kubernetes cluster.

Step 5: Expose the Deployment (Optional)

If you want to access your deployment externally, you can expose it as a service:

```
kubectl expose deployment my-deployment --type=NodePort --port=80
```

This command creates a service that maps to the pods created by your deployment.

The command kubectl expose deployment my-deployment --type=NodePort --port=80 is like telling Kubernetes, "Hey, I want to make my application accessible to the outside world.

kubectl: Your tool to talk to Kubernetes. It's like your way of giving instructions to the manager of your application.

expose: This is the action. You're saying, "I want to expose my application so people can access it."

deployment my-deployment: You're specifying which application you're talking about. In this case, it's the one named my-deployment.

--type=NodePort: This is how you want to expose it. You're saying, "Make it available on a specific port on each of the worker machines in the cluster." It's like opening a door on each worker machine for people to enter.

--port=80: This is the door number. You're specifying that you want people to come in through door number 80.

So, when you run kubectl expose deployment my-deployment --type=NodePort --port=80, you're instructing Kubernetes to set up a way for people (or other services) to access your application from outside the Kubernetes cluster. It's like making your application visible and reachable on a specific door (port) on each machine in the cluster.

Step 6: Access the Deployed Application

If you exposed the deployment, you can access the application using Minikube's service command:

```
minikube service my-deployment
```

This will open your default web browser to the exposed service.

The command minikube service my-deployment is like saying, "Hey Minikube, show me my app!" Let's break it down:

minikube: Your tool for managing a small, local Kubernetes cluster on your computer.

service: This is the action you want to perform. You're saying, "I want to expose my app so I can see it."

my-deployment: You're specifying which app you're interested in. In this case, it's the one named my-deployment.

When you run minikube service my-deployment, Minikube does two things:

Opens a Web Browser: It automatically opens your web browser to show your app.

Temporary URL: Minikube creates a special, temporary web address (URL) that allows you to view your app. This is often done using a NodePort, which means your app is accessible on a specific port on your local machine.

So, this command is a quick way to see and interact with your app running in the Minikube cluster. It's like opening a window to check how your application looks and behaves.

Step 7: Cleanup (Optional)

If you want to clean up the resources, you can delete the deployment and service:

```
kubectl delete deployment my-deployment

kubectl delete service my-deployment
```

the commands kubectl delete deployment my-deployment and kubectl delete service my-deployment

Delete Deployment:

kubectl delete deployment my-deployment

Its like dismantling a team of workers (pods) that handle your application. This command tells Kubernetes to take down the entire team named my-deployment. If you previously deployed your application using my-deployment, running this command removes all the worker pods associated with it.

Delete Service:

```
kubectl delete service my-deployment
```

Think of this as closing the door through which people access your application. The command tells Kubernetes to shut down the entrance (service) named my-deployment. If you exposed your application to the outside world using a service named my-deployment, running this command closes that access point.

These commands are like telling Kubernetes to disband the team (pods) and close the door (service) associated with your application named my-deployment. It's a way to clean up and remove the resources you previously set up in the Kubernetes cluster.

These steps cover the creation of a deployment in Minikube, managing pods with replicas, and optionally exposing the deployment as a service for external access. Adjust the configuration based on your specific application needs.

SOFTWARE ENGINEERING

UNIT – 4

CHAPTER – 4

CONTINUOUS MONITORING USING NAGIOS

I. UNDERSTANDING CONTINUOUS MONITORING

Continuous monitoring in DevOps refers to the practice of keeping a close and constant eye on the various aspects of software development and operations throughout the entire development lifecycle. It involves continuously gathering and analyzing data to ensure that the software is performing well, meeting security standards, and delivering value to users.

It involves:

Constant Observation: Continuous monitoring means consistently observing and collecting data about the software development and operation processes. This includes tracking code changes, system performance, and user interactions.

Automation: Automation plays a key role in continuous monitoring. Automated tools are used to collect data, run tests, and perform various checks on the software. This helps in quickly identifying issues and trends without manual intervention.

Feedback Loop: Continuous monitoring creates a feedback loop, providing information to development and operations teams in real-time or near-real-time. This allows teams to react promptly to any issues, make improvements, and iterate on the software more rapidly.

Security Monitoring: It also involves monitoring for security vulnerabilities and threats. This ensures that the software is protected against potential security risks, and any vulnerabilities are identified and addressed promptly.

Performance Monitoring: Continuous monitoring includes keeping an eye on the performance of the software. This involves tracking response times, resource usage, and other performance metrics to ensure that the application is running efficiently.

User Experience Monitoring: Monitoring user interactions and feedback helps in understanding how users are experiencing the software. This information can be valuable for making user-centric improvements.

Compliance Monitoring: Ensuring that the software complies with relevant standards, regulations, and best practices is another aspect of continuous monitoring. This is crucial for industries with strict compliance requirements.

Continuous monitoring in DevOps is about staying vigilant throughout the entire software development lifecycle, using automated tools to collect and analyze data, and ensuring that the software is secure, performs well, and meets the needs of users. It helps teams to identify and address issues early, promoting a more efficient and reliable development process.

II. NEED OF CONTINUOUS MONITORING

Continuous monitoring in DevOps is needed for three main reasons:

1. **Early Problem Detection:** To catch issues before they become big problems. By constantly checking how the software is doing in terms of performance, security, and other aspects.

   Example: Imagine finding and fixing a small bug before it causes a major system failure.

2. **Quick Response to Changes:** To adapt swiftly to changes in the software or its environment. By keeping a close eye on the software's behavior and user feedback in real-time.

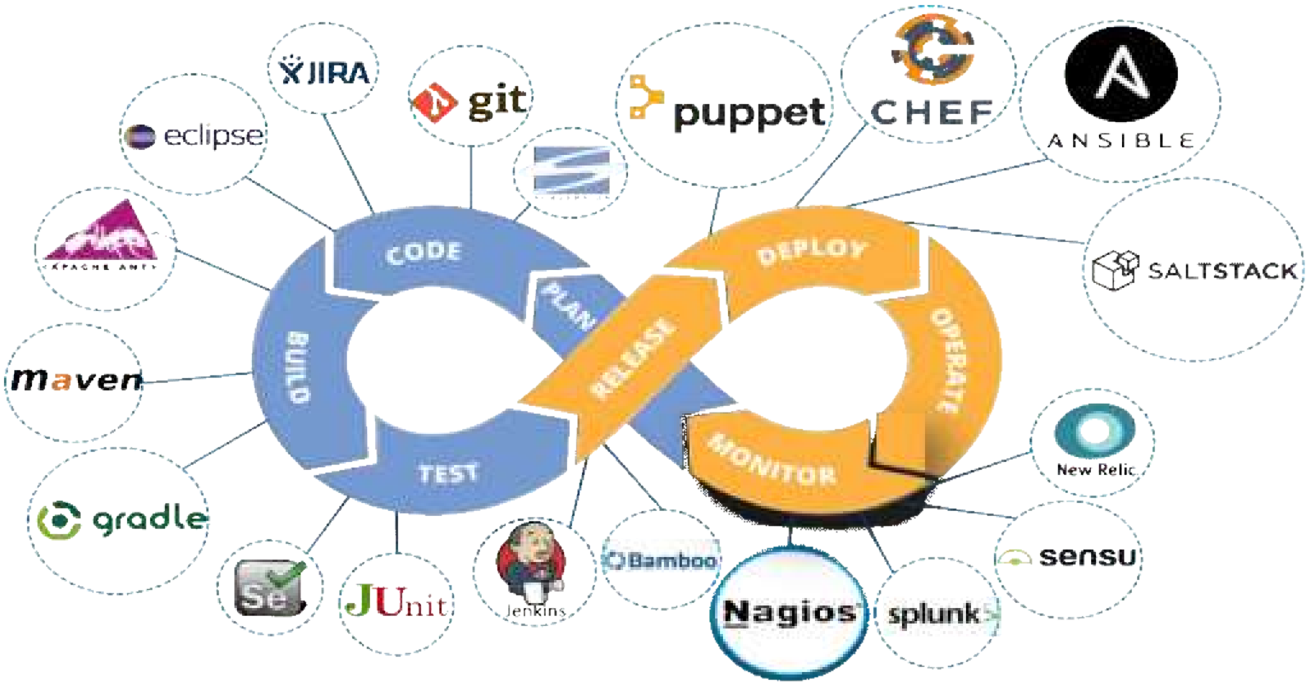
   Example: If a new feature causes unexpected errors, continuous monitoring helps detect and address the issues promptly.

3. **Continuous Improvement:** To make the software better over time. By gathering data on how the software is used and performing, and using that information to make informed improvements.

   Example: Learning from user interactions to enhance the user experience or optimizing code based on performance metrics.

Continuous monitoring helps DevOps teams to find and fix problems early, respond quickly to changes, and continuously make the software better. It's like having a watchful eye on the software, ensuring that it stays healthy, performs well, and evolves to meet user needs effectively.

III. VARIOUS CONTINUOUS MONITORING TOOLS IN DEVOPS



Continuous monitoring tools in DevOps are like superheroes that keep a watchful eye on the health and performance of your software. The various tools are:

1. Log Management Tools: Imagine your software keeping a diary. Log management tools read this diary and point out anything unusual or problematic.

Example Tool: ELK Stack (Elasticsearch, Logstash, Kibana).

2. Performance Monitoring Tools: These tools act like fitness trackers for your software, measuring how fast and efficiently it's running.

Example Tool: New Relic.

3. Security Monitoring Tools: Security tools are like guardians, watching out for any suspicious activity or potential threats to your software.

Example Tool: Security Information and Event Management (SIEM) tools like Splunk.

4. Infrastructure Monitoring Tools: Think of them as caretakers for the servers and systems supporting your software, ensuring everything is running smoothly.

Example Tool: Nagios.

5. Continuous Integration/Continuous Deployment (CI/CD) Tools: These tools ensure that new changes to your software are smoothly and automatically tested, built, and deployed.

Example Tool: Jenkins.

6. User Experience Monitoring Tools: These tools act like undercover agents, observing how users interact with your software to ensure a smooth experience.

Example Tool: UserZoom.

7. Application Performance Monitoring (APM) Tools: APM tools are like detectives, investigating and pinpointing issues within your application's code.

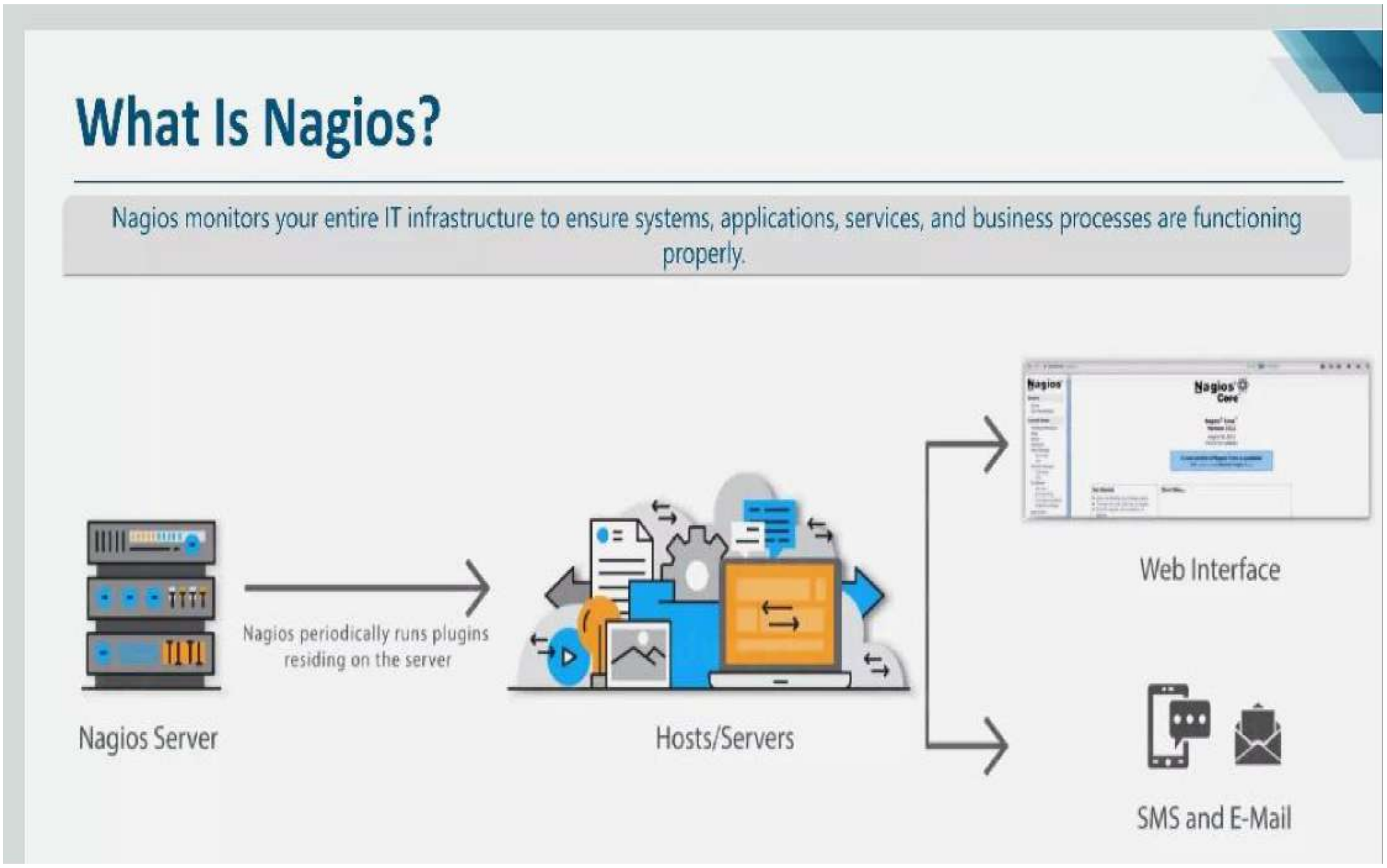
Example Tool: AppDynamics.

8. Container Orchestration Tools: Imagine a conductor directing an orchestra; these tools manage and organize containers (like Docker) running your software.

Example Tool: Kubernetes.

These tools work together to provide a comprehensive view of your software's health, security, and performance. They help DevOps teams keep everything in check and ensure that the software runs smoothly, adapts to changes seamlessly, and continuously improves over time.

IV. INTRODUCTION TO CONTINUOUS MONITORING USING NAGIOS



What Is Nagios?

Nagios monitors your entire IT infrastructure to ensure systems, applications, services, and business processes are functioning properly.

Nagios is a monitoring tool that keeps an eye on various aspects of your IT infrastructure, such as servers, networks, and services. If anything goes wrong or starts acting out of the ordinary, Nagios alerts to let you know.

Watching Everything: Nagios is set up to watch over your computers, servers, and the different programs and services running on them.

Checking Regularly: It regularly checks in with these systems, asking, "Hey, are you doing, okay?"

Alerting When Needed: If it notices something unusual, like a server not responding or a service acting up, Nagios sends out alerts – typically emails or messages – to the responsible parties, so they can address the issue.

Customizable: You can teach Nagios what to watch and how to respond. It's like training a watchful pet to recognize and alert you to specific problems.

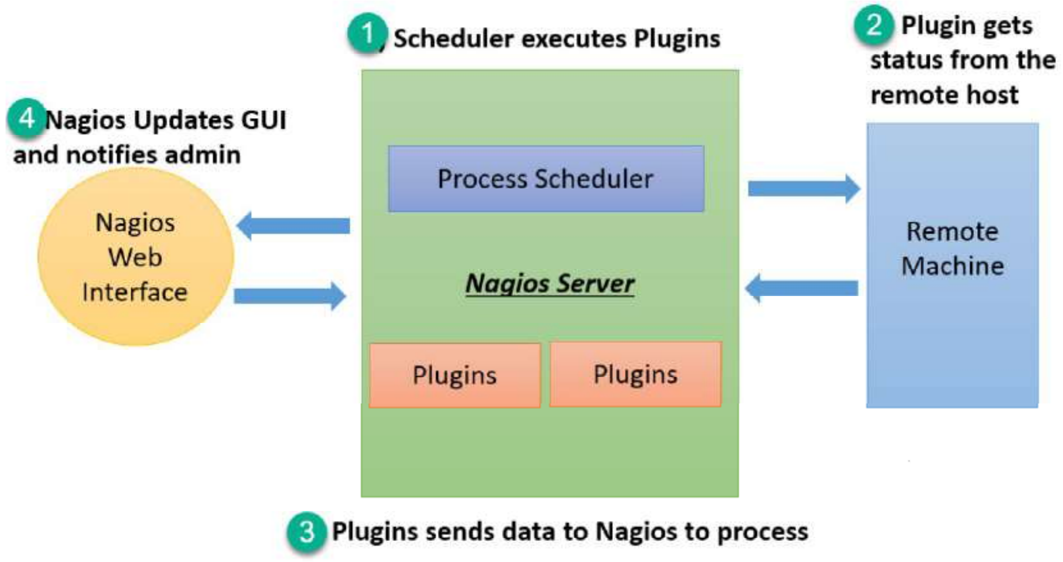
Keeping Things Smooth: The goal is to help you keep your IT systems running smoothly by catching and addressing issues before they become big problems.

Nagios is available as both a free, open-source version and a paid, commercial version.

Nagios Core (Free and Open Source): Nagios Core is the open-source version that is freely available for download and use. It provides basic monitoring functionality and can be extended with additional plugins to monitor various aspects of your IT infrastructure.

Nagios XI (Commercial): Nagios XI is the commercial version of Nagios that comes with additional features, a more user-friendly interface, and professional support. It is a paid product and is suitable for organizations that require advanced monitoring capabilities and official technical support.

V. NAGIOS ARCHITECTURE


1 Scheduler executes Plugins
2 Plugin gets status from the remote host
4 Nagios Updates GUI and notifies admin
Process Scheduler
Nagios Web Interface
Nagios Server
Remote Machine
Plugins
Plugins
3 Plugins sends data to Nagios to process

Nagios has a three-tier architecture, and it works like a team of specialized agents collaborating to monitor and report on the health of your IT systems.

Nagios Core (Central Server): The "boss" or the central coordinator. Nagios Core is the brain of the operation. It makes decisions, schedules checks, and receives reports from other components. This is typically installed on a central server that manages the entire monitoring process.

Plugins (Workers): Specialized workers in the field. These are like specialized agents that perform specific checks on servers, services, or devices. They report back to the Nagios Core with their findings. Plugins run on the systems being monitored. They check things like server load, disk space, network status, etc.

Web Interface (Communication Hub): The communicator and notifier. This is the friendly face of Nagios that you interact with. It provides a web-based interface for configuration, reporting, and acknowledging alerts. Installed on the same server as Nagios Core or on a different server, accessible through a web browser.

How It Works:

Nagios Core tells the plugins (workers) what to check and when.

a. Plugins perform the checks on the monitored systems and report back to Nagios Core.
b. Nagios Core compiles this information and decides if there are any issues.
c. If problems are detected, alerts are sent out through the web interface or other notification methods.
d. Nagios Core is the central command, plugins are the field agents doing the checks, and the web interface is the communication hub that lets you interact with the monitoring system. Together, they form a team that keeps a vigilant eye on your IT infrastructure and reports back to you when anything needs attention.

VI. INSTALL NAGIOS TOOL FROM DOCKER HUB

To install the Jason Rivers Nagios Docker container on Windows using PowerShell, you'll need to have Docker Desktop for Windows installed. Ensure that you have Docker Desktop running and that you are using PowerShell as your command line interface.

Jason Rivers's Docker image for Nagios is like a ready-made, easy-to-use package that allows you to set up Nagios for monitoring without the usual hassle. You save time and effort because the image comes pre-configured. Setting up Nagios becomes a simple task, especially if you're familiar with Docker. Using the same image ensures consistency across different deployments.

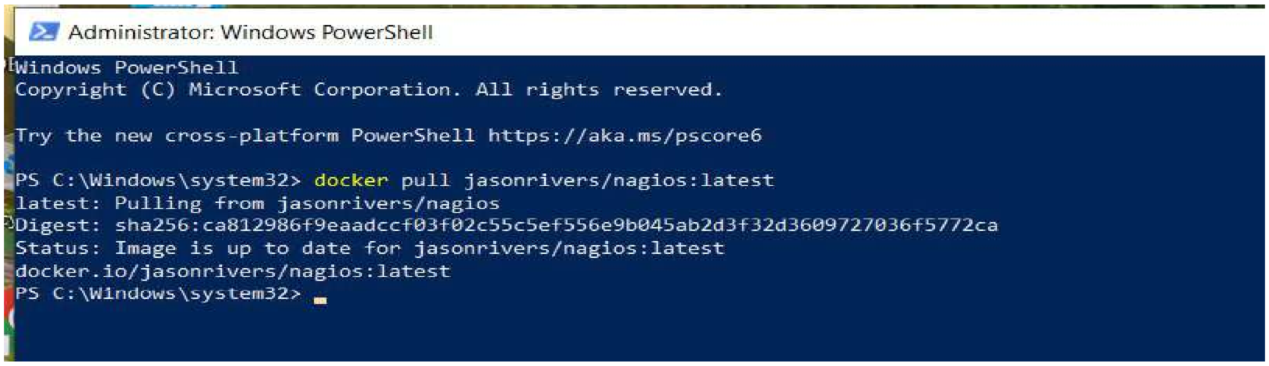
You run a single command in Docker to start Nagios using Jason Rivers's image. Once it's up, you can access Nagios through a web interface and start monitoring your systems.

Open PowerShell: Open PowerShell as an administrator. Right-click on the PowerShell icon and select "Run as administrator."

Pull the Nagios Container from Docker Hub: Run the following command to pull the Nagios container from Jason Rivers' Docker Hub repository:

```
docker pull jasonrivers/nagios:latest
```

This command downloads the latest version of the Nagios container image.



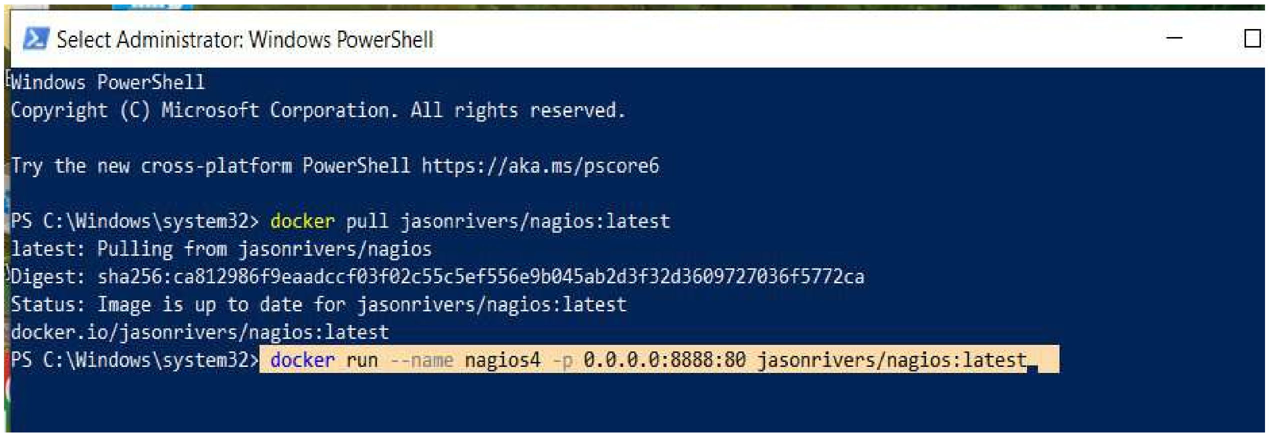
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> docker pull jasonrivers/nagios:latest
latest: Pulling from jasonrivers/nagios
Digest: sha256:ca812986f9eaadccf03f02c55c5ef556e9b045ab2d3f32d3609727036f5772ca
Status: Image is up to date for jasonrivers/nagios:latest
docker.io/jasonrivers/nagios:latest
PS C:\Windows\system32>
```

Run the Docker Command: Copy and paste the following command into the PowerShell terminal:

```
docker run -d --name nagiosdemo -p 8888:80 jasonrivers/nagios:latest
```



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> docker pull jasonrivers/nagios:latest
latest: Pulling from jasonrivers/nagios
Digest: sha256:ca812986f9eaadccf03f02c55c5ef556e9b045ab2d3f32d3609727036f5772ca
Status: Image is up to date for jasonrivers/nagios:latest
docker.io/jasonrivers/nagios:latest
PS C:\Windows\system32> docker run --name nagios4 -p 0.0.0.0:8888:80 jasonrivers/nagios:latest
```

This command tells Docker to run a detached container named nagiosdemo from the jasonrivers/nagios image, mapping port 8888 on your host machine to port 80 in the container.

Wait for the Download: Docker will download the Nagios image from Docker Hub if it's not already available on your machine. This might take some time depending on your internet connection.

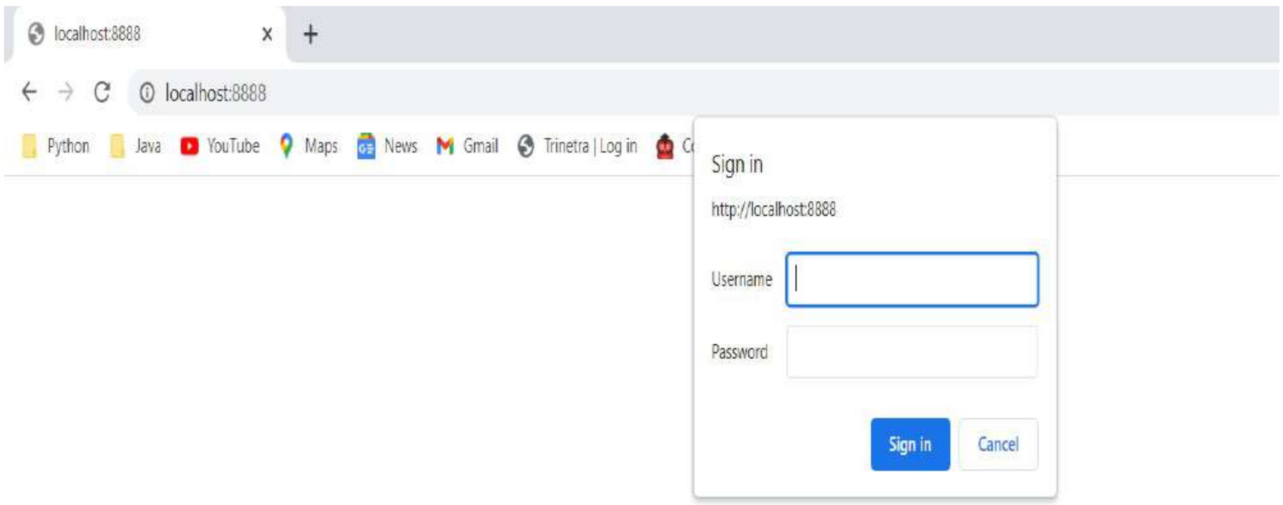
Access Nagios Web Interface: Once the container is running, you can access the Nagios web interface by opening a web browser and navigating to

```
http://localhost:8888/nagios.
```

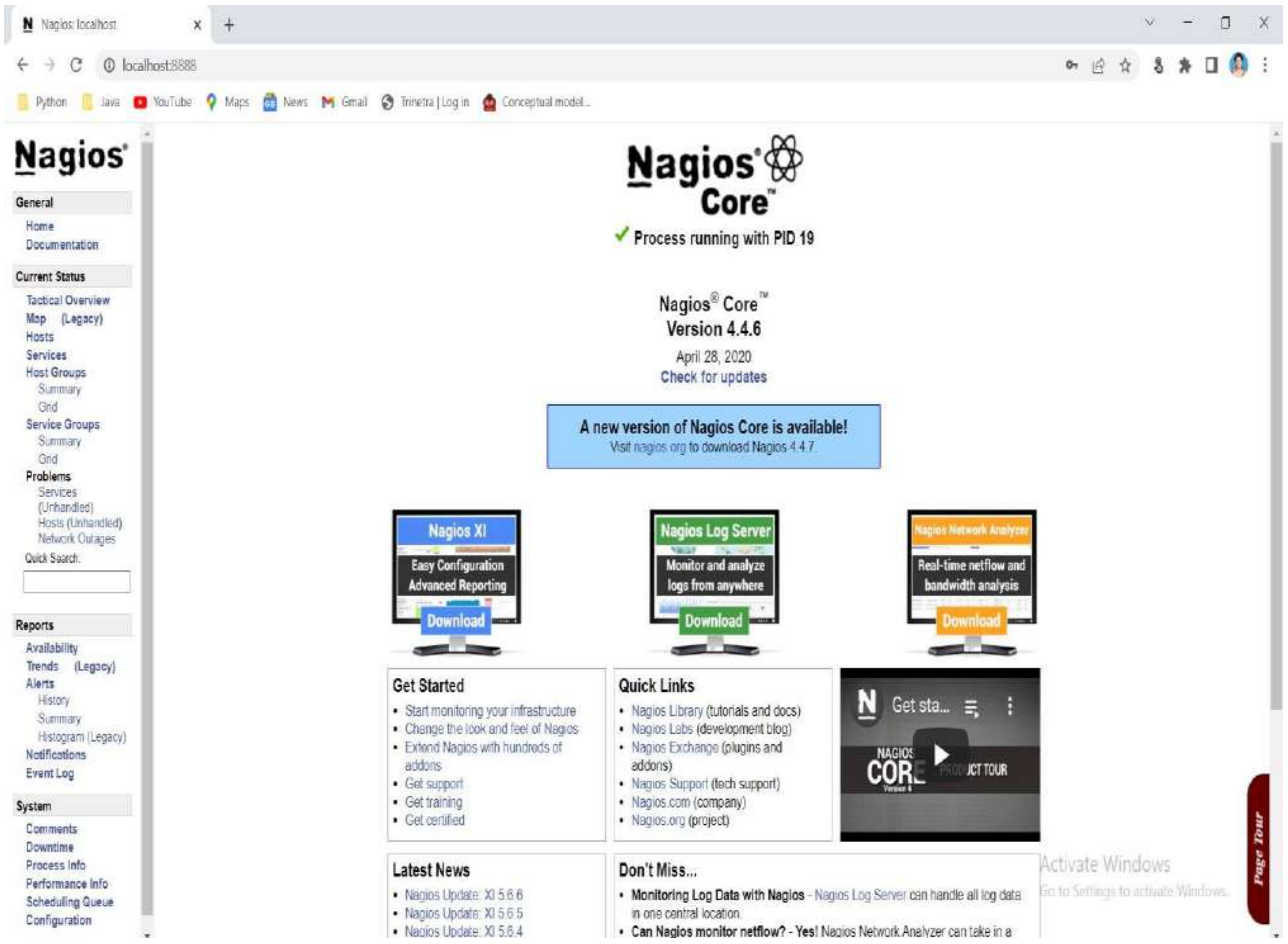
The default login credentials are:

```
Username: nagiosadmin

Password: nagios
```



Explore Nagios: You should now be able to explore the Nagios web interface and start configuring it for your monitoring needs.



✓ Process running with PID 19

Nagios® Core™
Version 4.4.6
April 28, 2020
Check for updates

A new version of Nagios Core is available!
Visit nagios.org to download Nagios 4.4.7.

Get Started
- Start monitoring your infrastructure
- Change the look and feel of Nagios
- Extend Nagios with hundreds of addons
- Get support
- Get training
- Get certified

Quick Links
- Nagios Library (tutorials and docs)
- Nagios Labs (development blog)
- Nagios Exchange (plugins and addons)
- Nagios Support (tech support)
- Nagios.com (company)
- Nagios.org (project)

Latest News
- Nagios Update: XI 5.6.6
- Nagios Update: XI 5.6.5
- Nagios Update: XI 5.6.4

Don't Miss...
- Monitoring Log Data with Nagios - Nagios Log Server can handle all log data in one central location.
- Can Nagios monitor netflow? - Yes! Nagios Network Analyzer can take in a

The Nagios Core dashboard is the central hub where you can monitor the health and status of various devices, services, and systems in your network. The main components on the dashboard:

a. **Host Status:** This section shows a list of all the devices (hosts) you are monitoring, such as servers, routers, or computers. Each host is usually represented by a specific icon or color that indicates its status (e.g., up, down, unreachable).
b. **Service Status:** Services are specific functionalities or applications running on the monitored hosts (e.g., web server, email service). This section displays the status of each service, indicating whether they are working correctly or experiencing issues.
c. **Status Summary:** A quick summary that provides an overview of the total number of hosts and services being monitored, as well as their current status (e.g., how many are up, down, etc.).
d. **Performance Data:** Some versions of Nagios include performance data, which gives you insights into resource usage, response times, and other metrics related to the monitored hosts and services.

e. **Notifications:** Nagios can send notifications when it detects a problem with a host or service. This section may show recent notifications, helping you stay informed about issues in your network.
f. **Event Log:** A log that provides a history of events and changes in the monitoring system. This can be useful for troubleshooting and understanding what has happened over time.
g. **Configuration:** A section that allows you to manage and configure the monitoring settings. You can add new hosts, services, or modify existing configurations through this interface.
h. **Quick Links**: Convenient links or buttons that provide shortcuts to important features or sections within Nagios Core.

The Nagios Core dashboard is like a control center where you can keep an eye on the health of your network, receive alerts about potential issues, and manage your monitoring configurations. It's a powerful tool for system administrators to maintain the reliability and performance of their IT infrastructure.

If you want to stop the Nagios container, you can use the following command:

```
docker stop nagiosdemo
```

To remove the stopped container, you can use:

```
docker rm nagiosdemo
```

Note that the provided command assumes that there are no other services running on port 8888 on your host machine. If port 8888 is already in use, you can modify the command to use a different host port, such as docker run -d --name nagiosdemo -p 8080:80 jasonrivers/nagios:latest. Adjust the command based on your specific requirements and port availability.