Contents

1 Continuous Integration & Continuous Deployment

1.1 Introduction

CI/CD stands for Continuous Integration and Continuous Deployment (or Continuous Delivery), which is a set of practices and tools used by software development teams to automate and streamline the process of building, testing, and deploying software.

Continuous Integration is the practice of automatically building and testing code changes as soon as they are committed to a shared repository. This is usually done by integrating code changes into a shared codebase multiple times a day, allowing teams to catch and fix errors quickly. This can help reduce the amount of time and effort required to find and fix errors later in the development process.

Continuous Deployment (or Continuous Delivery) is the practice of automatically deploying code changes to production environments after they have been built and tested. This is usually done using automation tools to ensure that code changes are deployed quickly and consistently, while minimizing the risk of errors or downtime.

By implementing CI/CD practices, development teams can improve their software quality, reduce time-to-market, and increase their ability to respond quickly to changing requirements or customer needs. It allows them to deliver software more frequently and reliably, while also minimizing the risks and costs associated with manual processes.

1.1.1 Continuous Integration

Continuous Integration (CI) is a software development practice that involves merging code changes from multiple developers into a shared code repository on a regular basis. The goal of CI is to catch integration errors and bugs early in the development process, before they can cause larger issues down the line.

In a typical CI setup, code changes are committed to a shared repository several times a day, rather than waiting for a large batch of changes to be completed. This triggers an automated build process, which compiles the code and runs a suite of automated tests to ensure that the changes work as expected and do not introduce new bugs.

If the tests pass, the changes are merged into the shared repository and the new code becomes part of the larger codebase. If the tests fail, the developers are notified of the failure and can work to fix the issue before it becomes a larger problem.

CI helps ensure that the codebase remains stable and functional throughout the development process, while also helping to catch and resolve issues early on. It also encourages developers to write better code, as they can see the results of their changes immediately and work to resolve issues before they become larger problems.

1.1.2 Continuous Deployment

Continuous Deployment (CD) is a software development practice that involves automatically deploying software changes to production as soon as they are validated by the automated testing and release pipeline. The goal of CD is to reduce the time it takes to deliver software changes to users, and to increase the frequency and reliability of software releases.

In a typical CD setup, code changes are automatically built, tested, and packaged into a release artifact, which is then deployed to a staging or production environment. The deployment process is typically automated, using tools such as configuration management and deployment automation tools like Ansible, Terraform or Kubernetes.

Once the code changes are deployed to the production environment, they are monitored to ensure that they are functioning as expected. If any issues arise, they can be quickly identified and resolved using tools such as monitoring and logging systems.

CD helps to reduce the time it takes to deliver software changes to users and increase the reliability of the software release process. It allows developers to deliver new features and fixes more quickly, and to respond more rapidly to changes in user needs and requirements. However, it requires a strong focus on automated testing and quality control to ensure that the software changes are properly validated before they are deployed to production, and that any issues are quickly identified and resolved.

1.1.3 CD/CI lexicon

There are several terms related to CI/CD (Continuous Integration/Continuous Deployment) that are commonly used in software development. Some of the most common terms include:

- Build: The process of compiling source code and linking it to create an executable program or library.
- Test: The process of verifying that software functions as expected and meets requirements. This may include unit tests, integration tests, performance tests, and other types of tests.
- Deployment: The process of installing software on a production server or other environment.
- Pipeline: A series of automated steps that code changes go through, including building, testing, and deploying.
- Artifact: A package that contains the compiled code, libraries, and other files needed to deploy a software application.

- Version Control: The process of managing changes to software code over time, including storing and tracking changes, and making it easier to collaborate on code changes.
- Continuous Integration: The practice of continuously integrating code changes into a shared code repository, and running automated tests to ensure that the changes do not break existing functionality.
- Continuous Deployment: The practice of automatically deploying code changes to production environments after they have been built and tested.
- Infrastructure as Code (IaC): The practice of managing infrastructure, such as servers and networks, using code and automation tools.
- DevOps: The culture and practices that focus on collaboration between software development and IT operations teams, with a focus on automating processes and improving communication and feedback between teams.

1.1.3.0.1 Pipeline In software development, a pipeline (or a deployment pipeline) is a set of automated steps that code changes go through to be built, tested, and deployed. The pipeline is a series of stages that the code changes must pass through before they can be deployed to a production environment.

The pipeline typically begins with a build stage, where the code changes are compiled into an executable program or library. The next stage is usually testing, where automated tests are run to ensure that the code changes function correctly and do not break existing functionality. Depending on the complexity of the software, the testing stage can include various types of tests such as unit, integration, regression, and performance testing.

Once the code changes pass the testing stage, the next stage in the pipeline is usually deployment, where the changes are automatically deployed to a staging or production environment. This is typically done using automated deployment tools and configuration management systems to ensure consistency and reliability in the deployment process.

The pipeline can be configured to include additional stages such as security scanning and code quality checks. The ultimate goal of a pipeline is to automate the software release process and minimize the time it takes to deliver new features or bug fixes to end-users while ensuring quality and reliability.

1.1.4 Dev should know

As a software developer, there are several important things to know about CI/CD (Continuous Integration/Continuous Deployment) to ensure that you can effectively contribute to the software development process:

- Code frequently and commit changes often: Continuous Integration requires developers to commit changes to the shared repository frequently, and ensure that the code compiles and passes automated tests. Therefore, developers should be able to code frequently and ensure that their changes don't break the existing functionality.
- Write automated tests: Automated testing is a critical part of the CI/CD process, so developers should be proficient in writing unit tests, integration tests, performance tests, and other types of automated tests to validate their changes.
- Collaborate with other developers: As part of the CI/CD process, developers need to collaborate with their colleagues to ensure that their code changes do not conflict with other changes.

This includes communication around code changes, version control, and other aspects of the software development process.
- Understand the deployment process: Developers should have a good understanding of the deployment process and the tools used in the process, such as build tools, configuration management, and deployment automation tools.
- Be aware of security and compliance requirements: As part of the CD process, it's important to be aware of any security and compliance requirements that need to be met. Developers should be familiar with security best practices and incorporate them into their code changes.
- Embrace DevOps culture: CI/CD is part of a broader DevOps culture that emphasizes collaboration, automation, and continuous improvement. Developers should be open to feedback, embrace automation and continuous learning, and work closely with other teams involved in the development and deployment process.

By understanding these key aspects of CI/CD and incorporating them into their work, developers can contribute to a successful software development process that delivers high-quality software to end-users quickly and reliably.

1.1.5 Dev must know

As a developer, there are a few important things to know about CI/CD, even if you are not directly involved in building or managing the pipeline:

1. Know how to use version control: CI/CD relies on version control, so it's important to be proficient in using version control tools such as Git. You should know how to create and switch between branches, commit changes, and resolve conflicts.
2. Understand the build and deploy process: While you may not be responsible for building or deploying the code changes, it's still helpful to understand the basic process. You should know how to trigger a build and deploy process, how to monitor the progress, and how to access logs and other information.
3. Write high-quality code: CI/CD depends on automated tests to ensure code quality, so it's important to write high-quality code that can pass these tests. You should follow best practices for writing clean, maintainable, and testable code.
4. Know how to troubleshoot issues: If there are issues with the build or deployment process, you should know how to troubleshoot them. This includes understanding error messages, logs, and other diagnostic information.
5. Understand the importance of automation: CI/CD relies on automation to ensure that the process is efficient and consistent. You should understand the importance of automation and be willing to learn new tools and technologies to improve the process.

By having a basic understanding of CI/CD and the tools and processes involved, you can contribute to a successful software development process and ensure that your code changes are delivered quickly and reliably to end-users.

1.1.5.1 Automated tests There are several different types of tests that are commonly used in software development. Here are some of the most important types:

1. Unit Tests: Unit tests are tests that focus on individual units of code, such as functions, classes, or methods. They are typically automated tests that are run frequently during development to catch issues early.

2. Integration Tests: Integration tests are tests that check how different units of code work together. They are used to ensure that the different parts of an application are communicating and functioning correctly.
3. System Tests: System tests are tests that check the entire system or application, rather than individual units. They are used to ensure that the application functions correctly from end-to-end.
4. Acceptance Tests: Acceptance tests are tests that check that the application meets the requirements and expectations of the end-user. They are used to ensure that the application is useful and valuable to the people who will be using it.
5. Regression Tests: Regression tests are tests that are used to ensure that code changes do not cause previously working features to break. They are used to catch issues that may be introduced as a result of changes to the code.
6. Performance Tests: Performance tests are tests that check the speed and scalability of an application. They are used to ensure that the application can handle the expected load and user traffic.
7. Security Tests: Security tests are tests that check the security of the application, looking for vulnerabilities that could be exploited by hackers or malicious actors.

By using a combination of these different types of tests, developers can ensure that their software is robust, reliable, and secure.

difference between unit tests and integration tests.

Unit tests are tests that focus on individual units of code, such as functions, classes, or methods. They are typically automated tests that are designed to catch issues early in the development process. The purpose of unit testing is to ensure that each unit of code performs as intended, and that changes to the code do not introduce new errors.

In a unit test, a developer writes a test case that exercises a specific function or method, and checks that it produces the expected output for a given input. For example, a developer might write a unit test that checks that a function that calculates the area of a rectangle returns the correct value for a given set of inputs.

Integration tests, on the other hand, are tests that check how different units of code work together. They are used to ensure that the different parts of an application are communicating and functioning correctly. The purpose of integration testing is to identify issues that can arise when different units of code are combined, and to ensure that the application as a whole works correctly.

In an integration test, a developer writes a test case that exercises multiple units of code together, and checks that they produce the expected output for a given input. For example, a developer might write an integration test that checks that a user can successfully log in to an application, by exercising the login form, the authentication service, and the user database.

To summarize, unit tests are used to test individual units of code in isolation, while integration tests are used to test how different units of code work together in the context of the application as a whole. Both types of tests are important for ensuring that software is reliable, robust, and works as intended.

1.2 Tools

There are many tools available for implementing continuous integration (CI). Here are some of the most popular CI tools:

Jenkins: Jenkins is an open-source automation server that can be used for building, testing, and deploying software. It is one of the most popular CI tools and has a large user community. Travis CI: Travis CI is a cloud-based CI tool that supports a wide range of programming languages and frameworks. It integrates easily with GitHub and can be used for building, testing, and deploying code. GitLab CI/CD: GitLab is a code hosting platform that includes built-in CI/CD functionality. It can be used to automate build, test, and deployment processes for code hosted on GitLab.

1.2.1 GitLab CI/CD

GitLab CI is a continuous integration and continuous delivery (CI/CD) platform that is built into the GitLab source code management system. It provides a way to automate the build, test, and deployment of code changes, and is designed to be simple to use and flexible enough to support a wide range of workflows and use cases.

Here are some key features and concepts of GitLab CI:

- GitLab Runner: The GitLab Runner is a lightweight agent that runs on your infrastructure and executes jobs defined in your GitLab CI configuration. The runner can be installed on Linux, macOS, or Windows, and can be configured to run in various modes, including shell, Docker, and Kubernetes.
- Pipelines: A pipeline is a series of stages that define the tasks and jobs that will be executed to build, test, and deploy your code. Pipelines are defined using a YAML file, and can be configured to run automatically when changes are pushed to your GitLab repository.
- Jobs: A job is a specific task that is executed as part of a pipeline. Jobs can be configured to run in parallel, and can be defined to run on specific runners or tags.
- Artifacts: Artifacts are files generated by a job, such as compiled code, test results, or build logs. Artifacts can be archived and downloaded from GitLab after a job has completed, allowing developers to inspect and analyze the output of the job.
- Environments: An environment is a specific deployment target, such as a staging or production server. Environments can be defined in GitLab CI, and jobs can be configured to deploy code changes to specific environments.
- Integration: GitLab CI integrates with a wide range of other tools and services, such as Jira, Kubernetes, and AWS. This allows you to create a complete CI/CD pipeline that includes source code management, build tools, testing frameworks, and deployment tools.

Overall, GitLab CI provides a powerful and flexible platform for automating the build, test, and deployment processes of software development, and its integration with GitLab makes it a popular choice for organizations that use GitLab for source code management.

1.2.1.1 Gitlab runner A GitLab Runner is a lightweight agent that runs jobs defined in your GitLab CI/CD pipeline. It can be installed on your infrastructure, including Linux, macOS, and Windows, and can be configured to run jobs in a variety of environments, including shell, Docker, and Kubernetes.

The runner listens for jobs that are assigned to it by the GitLab server and runs them on the

configured environment. The runner is responsible for pulling the code from the GitLab repository, executing the build and test steps, and sending the results back to the GitLab server.

GitLab runners can be registered with a specific GitLab instance or can be shared across multiple instances. They can be set up to run jobs in parallel, which can help to speed up the build and test process.

GitLab Runner is an open-source project and is available for free. It provides a flexible and scalable way to execute your CI/CD pipeline and can be used with a wide range of tools and technologies to build and deploy your code.

1.2.1.2 Gitlab jobs In the context of GitLab CI/CD, a job is a specific task that is defined in your pipeline. It represents a single unit of work that needs to be performed, such as building your application, running tests, or deploying your code.

A job is defined by a set of instructions that specify how it should be executed, such as what script to run or what Docker image to use. Jobs can be configured to run in parallel or sequentially, and can be dependent on other jobs in the pipeline.

Each job runs in a separate environment, such as a Docker container, and has access to its own set of variables and resources. Jobs can also be configured to run on a specific runner or with a specific tag.

The output of each job is recorded and can be accessed from the GitLab UI or through the GitLab API. This allows you to monitor the progress of your pipeline and troubleshoot any issues that may arise.

In summary, a job is a key concept in GitLab CI/CD that allows you to define the specific tasks that need to be executed as part of your pipeline, and provides a way to automate your build, test, and deployment processes.

1.2.1.3 Basic setup To set up a GitLab CI/CD pipeline, you will need to define the jobs that you want to run as part of your pipeline and specify how they should be executed. Here are the basic steps for setting up a simple GitLab CI/CD pipeline:

Create a .gitlab-ci.yml file in the root of your repository. This file contains the configuration for your pipeline, including the jobs and the stages they belong to.

Define the stages for your pipeline. Stages represent the phases of your pipeline, such as build, test, and deploy. You can define as many stages as you need.

```yaml
stages:
  - build
  - test
  - deploy
```

- Define the jobs for each stage. Jobs are the specific tasks that need to be executed as part of each stage. You can define as many jobs as you need, and each job can have its own set of instructions.

```yaml
build:
  stage: build
```

```
  script:
    - echo "Building the application"

test:
  stage: test
  script:
    - echo "Running tests"

deploy:
  stage: deploy
  script:
    - echo "Deploying the application"
```

Configure any additional settings for your pipeline, such as runners, variables, and artifacts. For example, you can specify the runner that should be used to execute your pipeline, define environment variables that should be available to your jobs, and specify artifacts that should be produced by your pipeline.

```
variables:
  DATABASE_URL: "postgresql://user:password@localhost/database"

artifacts:
  paths:
    - build/
```

Commit and push your .gitlab-ci.yml file to your repository. This will trigger your pipeline to run automatically.

Monitor the progress of your pipeline in the GitLab UI or through the GitLab API. You can view the status of each job, check the logs for any errors, and access any artifacts that were produced by your pipeline.

1.2.1.4 Docker - k8s integration GitLab integrates with Docker in several ways. Here are some of the key features:

1. Docker Build: GitLab CI/CD can use the Dockerfile in your repository to build a Docker image of your application, which can then be pushed to a container registry.
2. Container Registry: GitLab provides a built-in container registry that allows you to store and manage your Docker images. You can push your Docker images to the GitLab container registry as part of your CI/CD pipeline.
3. Docker Compose: GitLab CI/CD can also use Docker Compose to orchestrate the deployment of your Docker containers. This allows you to define a multi-container environment in a single YAML file, and then use GitLab CI/CD to automatically deploy it.
4. Kubernetes: GitLab also integrates with Kubernetes, an open-source container orchestration platform. You can use GitLab to manage your Kubernetes clusters and deploy your applications to them. GitLab can even automatically create and manage Kubernetes namespaces for you.

1.2.1.5 Configuration file Here is an example .gitlab-ci.yml configuration that will run tests and build and publish a Docker image when a commit is made to the master branch:

```yaml
stages:
  - test
  - build
  - deploy

variables:
  DOCKER_REGISTRY: registry.example.com
  DOCKER_IMAGE_TAG: latest
  DOCKER_IMAGE_NAME: my-app

test:
  stage: test
  script:
    - echo "Running tests"

build:
  stage: build
  script:
    - echo "Building Docker image"
    - docker build -t $DOCKER_REGISTRY/$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG .
    - echo "Pushing Docker image to registry"
    - docker push $DOCKER_REGISTRY/$DOCKER_IMAGE_NAME:$DOCKER_IMAGE_TAG
  only:
    - master

deploy:
  stage: deploy
  script:
    - echo "Deploying to production"
  environment:
    name: production
    url: https://my-app.example.com
  only:
    - master
```

This configuration defines three stages: test, build, and deploy. The variables section defines variables that will be used throughout the pipeline.

In the test stage, tests are run. In the build stage, the Docker image is built and pushed to the registry defined in the DOCKER_REGISTRY variable. This stage only runs when a commit is made to the master branch.

In the deploy stage, the application is deployed to a production environment, with the environment name and URL specified. This stage also only runs when a commit is made to the master branch.

Note that in order to use this configuration, you will need to have a Dockerfile in the root of your repository that defines how to build the Docker image for your application. You will also need to replace the values for the DOCKER_REGISTRY, DOCKER_IMAGE_TAG, and DOCKER_IMAGE_NAME variables with the appropriate values for your environment.

1.2.1.6 Advanced configuration or using more advance features: in GitLab CI/CD, rules allow you to define conditions that control when a job is created and whether it is allowed to run. You can use rules to specify when a job runs based on the changes made to the code, the branch or tag that triggered the pipeline, the environment variables, and other factors.

The if statement is a common condition that can be used to specify when a job should run. It allows you to run a job only if a certain condition is met. For example, you can use the if statement to run a job only when a particular environment variable is set or only when the pipeline is triggered by a certain branch.

The on_success keyword is used to specify actions that should be taken if the job succeeds. This keyword can be used to trigger additional jobs, notify users or external systems of the job's success, or perform other actions.

Here is an example of using rules, if, and on_success in a .gitlab-ci.yml file:

```
build:
  script:
    - make build
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: always
    - when: manual
  tags:
    - docker
  artifacts:
    paths:
      - bin/

deploy:
  script:
    - make deploy
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      on_success:
        - notify_slack
  tags:
    - docker

notify_slack:
  script:
    - slack-notification.sh success
  when: on_success
```

In this example, the build job is run automatically when a commit is made to the main branch. The deploy job is also run when a commit is made to the main branch, but it requires manual intervention to start. The notify_slack job is only run when the deploy job succeeds and sends a Slack notification.

1.2.1.7 GitLab CI/CD variables Here are some of the most common parameters that can be used in the .gitlab-ci.yml file:

1. image: specifies the Docker image to use as the build environment for the job
2. script: specifies the commands to run as part of the job
3. stage: specifies the stage in which the job should be run (e.g. "build", "test", "deploy")
4. artifacts: specifies the files or directories to be saved as artifacts and made available to subsequent jobs
5. cache: specifies the files or directories to be cached between pipeline runs to speed up the build process
6. environment: specifies the environment variables to be set for the job
7. when: specifies when the job should be run, based on the status of the previous job(s)
8. dependencies: specifies the names of the jobs that must complete successfully before the current job can run
9. rules: specifies the conditions under which the job should be run, b

1.2.1.8 GitLab CI/CD regular expression There are many useful regular expressions that you can use in a GitLab CI/CD configuration file. Here are some examples:

`if: $CI_COMMIT_BRANCH =~ /^release-\\d+\\.\\d+$/`: This regular expression matches a branch name that starts with "release-" followed by one or more digits, a dot, and one or more digits, such as "release-1.0" or "release-2.3". `if: $CI_COMMIT_TAG =~ /^v\\d+\\.\\d+\\.\\d+$/`: This regular expression matches a tag name that starts with "v" followed by three sets of one or more digits separated by dots, such as "v1.2.3" or "v2.0.0" `if: $CI_COMMIT_MESSAGE =~ /\\[skip ci\\]/`: This regular expression matches a commit message that contains the phrase "[skip ci]". This is often used to skip running the pipeline for commits that don't require it, such as changes to documentation or comments.

1.2.1.9 Arg `when` The when parameter in GitLab CI allows you to specify when a job should run. Here are the different options that you can use with the when parameter:

1. on_success: runs the job only when the previous job in the pipeline completes successfully.
2. on_failure: runs the job only when the previous job in the pipeline fails.
3. always: runs the job regardless of the status of the previous job.
4. manual: runs the job only when it is manually triggered through the GitLab UI or API.
5. delayed: runs the job after a delay, specified using the start_in parameter.
6. never: prevents the job from running, even if it is explicitly specified as a dependency of another job.

1.2.1.10 Arg `stage` In GitLab CI, a stage is a named phase in your pipeline. Each job in your pipeline can be assigned to a stage, and the stages run in the order they are defined in your .gitlab-ci.yml file.

By default, GitLab CI provides four stages:

build: for building your code test: for running tests on your code deploy: for deploying your code review: for reviewing changes made in a merge request You can use these stages as a starting point for your pipeline, or you can define your own custom stages to suit your workflow. For example, you

might define a lint stage for running code quality checks, or a package stage for packaging your code for deployment.

1.2.1.11 Arg `script`

In GitLab CI, the script keyword is used to define the commands that should be run for a given job. The script section is where you define the actual steps that your job should perform, such as compiling your code, running tests, or building a Docker image.

The script section should be a list of shell commands that will be executed by the runner. These commands can be as simple or as complex as you need them to be, and can include variables, conditionals, and other shell constructs.

Note that GitLab CI runs each command in its own shell, so if you need to set environment variables or change the working directory for a command, you'll need to do so explicitly within that command. For example:

```
script:
  - export MY_VAR=123
  - cd myproject && make build
```

In this example, the first command exports the MY_VAR environment variable, and the second command changes the working directory to myproject before running the make build command.

1.2.1.12 Arg `artifacts`

In GitLab CI/CD, artifacts refer to the files generated by a build job that are passed to downstream jobs or can be downloaded for debugging purposes. These files can include build outputs, test results, logs, and any other files generated during the build process.

Artifacts are uploaded to the GitLab server after a job completes and can be accessed through the web interface or downloaded using the GitLab API or the GitLab CLI. Artifacts can also be downloaded using the "Download" button in the GitLab web interface.

Artifacts can be defined in the .gitlab-ci.yml file using the artifacts keyword. You can specify which files to include as artifacts and also specify any additional settings, such as expiration time, file type, and file size limits.

Using artifacts in GitLab CI/CD can help you debug your builds and provide transparency into the build process for your team. They can also be useful for keeping a record of build outputs and test results for future reference.

1.2.1.13 Gitlab interview

1. What is continuous integration and continuous deployment?
2. What is a pipeline, and what are the stages in a pipeline?
3. How can you define a pipeline in GitLab CI using the .gitlab-ci.yml configuration file?
4. What are jobs, and how are they defined in the configuration file?
5. What is a runner, and how does it execute jobs?
6. What are some of the built-in GitLab CI variables?
7. What are the common types of tests, and how do you define them in GitLab CI?
8. How can you publish artifacts, such as Docker images or compiled binaries, to a registry?
9. How can you define triggers for a pipeline, such as manual or scheduled triggers?
10. How can you define rules to control when jobs are executed, based on conditions such as the branch or tag name?

11. You should also be familiar with some of the common tools and technologies that are commonly used in conjunction with GitLab CI, such as Docker, Kubernetes, Ansible, and Terraform.

is also highly scalable, with the ability to manage tens of thousands of servers simultaneously. It's commonly used by DevOps teams and system administrators to automate repetitive tasks and streamline infrastructure management.

1.2.2 Jenkins

Jenkins is an open-source automation server that is used for implementing continuous integration and continuous delivery (CI/CD) pipelines. Here's a high-level overview of how Jenkins works:

1. Configuration: First, you need to install Jenkins and configure it to work with your source code management system, build tools, and testing frameworks. This is done through a web-based interface that allows you to install plugins and set up build jobs.
2. Build Jobs: A build job is a set of instructions that Jenkins will follow to build, test, and deploy your code. Build jobs can be configured to run on a schedule or triggered by changes to your source code repository.
3. Build Process: When a build job is triggered, Jenkins will check out the source code from your repository and use the build tools and testing frameworks that you have configured to build and test the code.
4. Notifications: Once the build and testing are complete, Jenkins will generate notifications to alert developers if the build succeeded or failed. If the build fails, Jenkins will send an email notification to the developers responsible for the code changes that caused the build failure.
5. Integrations: Jenkins can integrate with a wide range of other tools and services, such as GitHub, Jira, and Slack. This allows you to create a complete CI/CD pipeline that includes source code management, build tools, testing frameworks, and deployment tools.
6. Plug-ins: Jenkins is extensible and has a large ecosystem of plugins that can be used to extend its functionality. There are plugins available for everything from additional build tools to advanced reporting and analysis tools.

Jenkins is primarily used through a web-based user interface, but it also supports automation through its REST API and command-line interface (CLI).

The web-based interface is the primary way to configure Jenkins, create and manage jobs, and view build results. It provides a rich and interactive user experience, with features such as job configuration wizards, real-time build status updates, and visualizations of build logs and test results.

However, the Jenkins CLI and REST API can be used to automate common tasks, such as creating jobs, triggering builds, and retrieving build results. This can be useful for integrating Jenkins with other tools and services in your development workflow, or for automating routine administrative tasks.

Overall, the web-based interface is the most common way to use Jenkins, but the CLI and REST API can provide additional flexibility and automation capabilities for advanced use cases.

1.3 Ansible

1.3.1 Introduction

Ansible is an open-source automation tool that allows you to automate the configuration, deployment, and management of IT infrastructure. It provides a simple and powerful way to automate complex tasks, such as application deployment, cloud provisioning, and network management.

Ansible is agentless, meaning it doesn't require any additional software or agents to be installed on remote systems. Instead, Ansible communicates with remote systems over SSH or WinRM, using a set of modules that provide the necessary functionality.

Ansible uses a declarative language, YAML, to define playbooks, which are sets of instructions that define the desired state of a system. Playbooks are used to define the steps necessary to configure or deploy a system, and they can be reused across multiple systems, making it easy to manage large infrastructures.

Here are some common use cases for Ansible:

1. Configuration management: Ansible can be used to ensure that the configuration of all hosts in an infrastructure is consistent and up to date. It can be used to manage files, packages, services, and users across multiple hosts.
2. Application deployment: Ansible can automate the deployment of applications to multiple hosts, ensuring that they are installed and configured correctly.
3. Cloud provisioning: Ansible can be used to automate the creation and configuration of virtual machines, containers, and other cloud resources on platforms such as AWS, Azure, and Google Cloud Platform.
4. Continuous integration and delivery (CI/CD): Ansible can be used to automate the testing, building, and deployment of applications as part of a CI/CD pipeline.
5. Network automation: Ansible can be used to automate the configuration and management of network devices, such as routers, switches, and firewalls.
6. Security and compliance: Ansible can be used to automate security-related tasks, such as applying patches, configuring firewalls, and ensuring compliance with security policies.

Overall, Ansible is a versatile tool that can be used in a wide range of IT environments and scenarios, helping to reduce manual effort, increase efficiency, and improve consistency and reliability.

1.3.1.1 Ansible child example Imagine you have a toy box with lots of different toys, and you want to organize them in a specific way. You could do it by hand, which would take a long time and be very tiring, or you could use a robot to do it for you. The robot would know exactly how to organize the toys and could do it quickly and easily.

Ansible is like that robot, but for organizing computer systems instead of toys. It helps you to organize and manage lots of different computers, servers, and other devices, so that you don't have to do it all by hand.

For example, if you have a lot of different computers that you need to update with the latest software, you could use Ansible to do it automatically. You would tell Ansible which computers to update, and it would go and update them all for you, without you having to do it manually on each one.

Similarly, if you need to install a new application on a lot of different computers, Ansible can help you do it quickly and easily, without having to manually install it on each computer.

In summary, Ansible is a tool that helps you to manage and organize lots of different computers and devices, so that you can save time and effort and get things done more quickly and easily.

1.3.1.2 why ansible If you have an application that needs to be installed, configured, and managed on multiple servers or machines, Ansible can be a great tool to help you automate those tasks.

For example, let's say you have a web application that you want to deploy to a production environment, which consists of multiple servers. Without Ansible, you would need to manually install and configure the application on each server, which would be time-consuming and error-prone.

With Ansible, you can write a playbook that automates the deployment process, including installing and configuring the necessary software, copying files to the servers, and starting the application. Once you have written the playbook, you can run it on all the servers at once, or one server at a time, depending on your needs. This saves time, reduces errors, and ensures that the application is consistently deployed across all servers.

Ansible can also help you manage the application after deployment, such as monitoring its performance, updating its configuration, and scaling it up or down as needed. By automating these tasks with Ansible, you can focus on developing your application and improving its features, rather than spending time on repetitive manual tasks.

Overall, Ansible can be a valuable tool for any developer or system administrator who needs to manage applications or infrastructure across multiple servers, and wants to do it efficiently, reliably, and at scale.

1.3.2 Common terms

- Playbook: A YAML file that defines a set of tasks to be executed on a group of hosts.
- Task: A single unit of work to be performed on a host, such as installing a package, copying a file, or executing a command.
- Role: A collection of tasks and files that can be reused across multiple playbooks.
- Inventory: A file that lists the hosts to be managed by Ansible and their properties, such as IP address, hostname, and group membership.
- Module: A unit of code that performs a specific task, such as managing files, packages, users, or services.
- Fact: A piece of information about a host, such as its hostname, IP address, or operating system, that Ansible gathers automatically.
- Handler: A task that is triggered by another task, such as restarting a service after it has been configured.
- Vault: A secure storage mechanism for storing sensitive data, such as passwords or API keys, in encrypted form.
- Ad hoc command: A one-line command that performs a single task on one or more hosts, such as pinging a host or checking the status of a service.
- Play: A set of tasks that are executed on a single host, as opposed to a playbook, which is executed on a group of hosts.

1.3.3 Playbook

In Ansible, a playbook is a file containing a set of instructions that define a set of tasks to be executed on one or more remote hosts. Playbooks are written in YAML format and can include multiple plays, each of which consists of one or more tasks.

Each task in a playbook is a single action, such as installing a package, copying a file, or starting a service. Tasks are executed in the order they appear in the playbook, and Ansible will report on the success or failure of each task.

Playbooks are a core component of Ansible and provide a way to automate complex tasks, such as deploying an application or configuring a group of servers. Playbooks can be run against one or many hosts, depending on your requirements, and can be used to perform a wide range of tasks, including software installation, configuration, and management.

Playbooks can also be combined with variables and templates, which allow you to create dynamic configurations that can be customized based on the needs of the target hosts. Overall, playbooks provide a flexible and powerful way to automate tasks across multiple hosts and can help to streamline your IT operations and reduce the risk of errors or inconsistencies.

Here is a simpler example:

Suppose you have two servers named "webserver1" and "webserver2", and you want to install the Apache web server on both of them. Here is what your playbook might look like:

```yaml
- name: Install Apache on webservers
  hosts:
    - webserver1
    - webserver2
  become: true
  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: latest
```

This playbook has a single play that targets the two hosts specified in the hosts section. We then use the become keyword to become a privileged user on the target hosts (i.e., a user with sudo or root permissions).

Finally, we have a single task that uses the yum module to install the latest version of the Apache web server package. When we run this playbook using the ansible-playbook command, Ansible will execute this task on both servers and install the Apache web server.

I hope this example helps to clarify what a playbook is and how it can be used to automate tasks on remote servers. Let me know if you have any further questions or if there's anything specific you'd like me to explain in more detail.

In an Ansible playbook, the yum module can be used to manage packages on remote hosts. The yum module allows you to install, update, and remove packages, as well as manage repositories and perform other package management tasks.

In an Ansible playbook, you can use the httpd package to install the Apache HTTP Server on a Linux system, and then use the service module to start, stop, or restart the server.

1.3.4 Server connections

o connect to "webserver1" and "webserver2" using Ansible, you need to specify the IP addresses or hostnames of these servers in your inventory file. The inventory file is a text file that lists the servers that Ansible should manage, along with any groupings and variables that are associated with them.

Here's an example of how you can define the inventory file for your servers:

```
[webservers]
webserver1 ansible_host=192.168.1.100
webserver2 ansible_host=192.168.1.101
```

In this example, we define a group called "webservers" and list the two servers "webserver1" and "webserver2" under this group. We also use the ansible_host variable to specify the IP addresses of these servers.

1.3.5 Task

In Ansible, a task is the basic unit of work that you want to perform on a remote system. It is defined within a playbook and represents a single action that needs to be executed on a remote host, such as installing a package, starting a service, or copying a file.

A task is defined using YAML syntax and includes several key-value pairs that specify what action needs to be taken, on which host(s), and with what parameters. Here's an example of a simple Ansible task that installs the apache2 package on a Debian-based system:

```
- name: Install Apache
  apt:
    name: apache2
    state: present
```

In this example, the task is named "Install Apache package" and uses the apt module to install the apache2 package. The name parameter specifies the name of the package to install, and the state parameter specifies that the package should be present on the system.

Tasks can be grouped together into a playbook, which is a collection of tasks that are executed in order on one or more remote hosts. By defining tasks in a playbook, you can automate complex workflows and manage your infrastructure more efficiently.