

Integrating Source Code with Version Control

Describing Version Control

1. Version Control is the practice of managing the changes that happen in a set of files or documents over time.
2. In software engineering, Version Control Systems (VCS) are applications that store, control, and organize the changes that happen in the source code of an application.
3. With version control systems, teams can organize and coordinate their work more reliably.
4. When several developers are working on the same project simultaneously, they must manage changes in a way that avoids overriding others' work, and solve potential conflicts that arise.
5. Version control systems serve as a *centralized, shared, or distributed storage for teams to save their progress*.
6. To coordinate work, teams store their code base in repositories. A repository is a common location within a VCS where teams store their work.
7. Source code repositories store the project files, as well as the history of all changes made to those files and their related metadata.
8. Teams can decide to use version control to store and manage the configuration of their platform infrastructure. This practice is known as **Infrastructure as Code (IaC)** and is a critical part of building a DevOps culture.
9. The most popular version control implementation is **Git**. Other popular version control systems are **Mercurial, SVN, and CVS**.

Software Version Control

A Version Control System (VCS) enables you to efficiently manage and collaborate on code changes with others. Version control systems provide many benefits, including:

- The ability to review and restore old versions of files.
- The ability to compare two versions of the same file to identify changes.
- A record or log of who made changes at a particular time.
- Mechanisms for multiple users to collaboratively modify files, resolve conflicting changes, and merge the changes together.

Defining the Role of Version Control in Continuous Integration

1. Version Control is the foundation for *Continuous Integration (CI)*. Because Continuous Integration is about integrating changes to files across a team frequently, you need a place to store the changes.

2. By using a VCS, you use *repositories* as the common location where you store your work and integrate changes.
3. Developers use **branches** as a way to develop their work without interfering with other team members' work.
4. In the context of Version Control, *a branch is a specific stream of work in a repository*.
5. Repositories usually have a **principal or main branch**, as well as a series of **secondary feature branches** where developers carry out their work.
6. After developers finish their work in a feature branch and the code is reviewed by another team member, they integrate the changes of the feature branch into the main branch. This basic workflow is the foundation for continuous integration.

Introducing Git

1. Git is a version control system (VCS) originally created by *Linus Torvalds* in 2005.
2. His goal was to develop a version control system (VCS) for maintaining the Linux kernel.
3. Git is a [free and open source](#) distributed version control system.
4. **Git is a source code management (SCM) tool.**
5. Projects are tracked in Git repositories. Within that repository, you run Git commands to create a snapshot of your project at a specific point in time. In turn, these snapshots are organized into branches.
6. The snapshots and branches are typically uploaded ("*pushed*") to a separate server ("*remote*").
7. There are many free *code repository platforms*, including:
 - GitHub
 - GitLab
 - BitBucket
 - SourceForge

Distributed vs Centralized VCS

1. In a **centralized** VCS, such as **Subversion**, you must upload file changes to a central server.
2. Although a centralized model is easier to understand, it carries limitations in workflow flexibility.
3. In a **decentralized/distributed** VCS, such as **git**, every copy of the repository is a complete or "**deep**" copy.
4. In a decentralized system, the designation of "primary" is arbitrary.

Note : Although Git repositories do not require a central server or copy, it is common for software projects to designate a primary copy. This is often referred to as the "**upstream**" repository.

Benefits of a decentralized VCS

- Works offline
- Create local backups
- Flexible workflows

- Ad hoc code sharing
- More granular permissions

Drawbacks of a decentralized VCS

- Steeper learning curve
- Longer onboarding times
- Increased workflow complexity

What is a commit?

1. As you make changes to files within your repository, you can persist those changes in a commit.
2. Git also tracks commit order by storing a reference to a parent commit within each commit.
3. You can imagine *each commit as a new layer of changes on top of the last*. This approach allows you to more easily rollback and track changes or catch up to new changes in files.

Contents of a Commit

Name	Purpose
Author	Name and email address of the person who committed the changes.
Time stamp	The date and time of commit creation.
Message	A brief comment describing changes made.
Hash	A generated unique identifier (UID) for the commit.

Managing Git Repositories

1. The primary interface for Git is the provided command-line interface (CLI).
2. This CLI includes various "subcommands", such as *add*, *init*, and *commit*.
3. Git provides a man page for each of these subcommands. For example, you can view the man page for the add subcommand by running: **git help add**

git --help

Obtaining a Git Repository

1. To run most Git operations, you must run the command within a Git repository.
2. One option is to initialize a repository in your local system by using *git init*. This subcommand will create a *.git directory*. Remember, this subdirectory contains all of the information about your repository.
3. Another option is to clone an existing repository by using *git clone*. For example, you can clone from GitHub, which is a popular code sharing website.

Staging Changes

1. In order to create a *commit*, you must first stage changes.
2. Any "*unstaged*" changes will not be included in the commit.
3. Add a file's changes to the stage with the ***git add*** subcommand and specify the name of the file.
4. To remove changes from the stage, use ***git reset***.

Creating a Commit

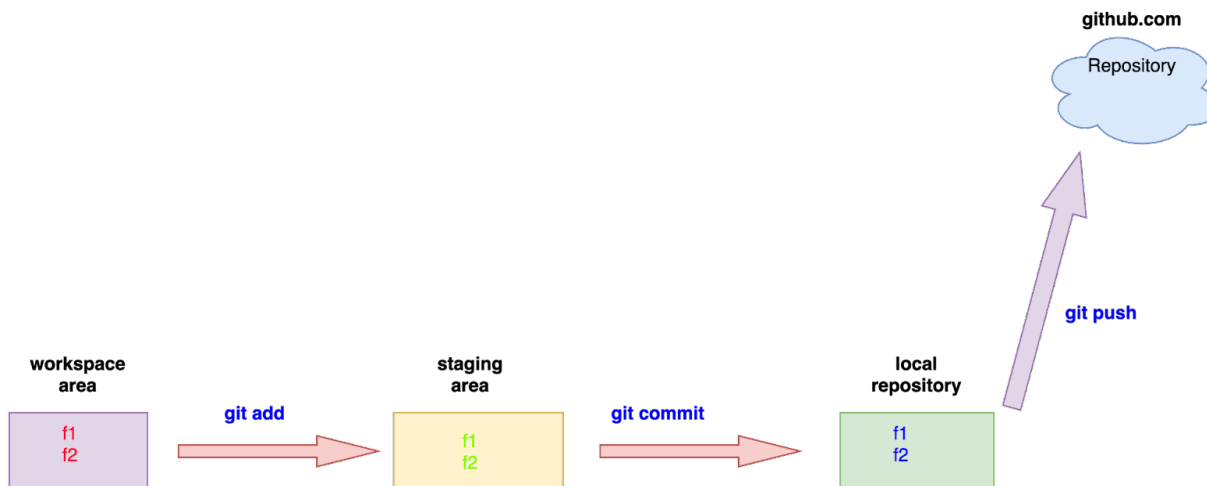
1. Once you have staged changes, you can use ***git commit*** to create a commit.
2. With your changes committed, you are free to continue making changes knowing that you can easily restore to earlier versions.

As you use Git, you adopt the following workflow:

- Make changes to your project files
- Stage changes with the *git add* command
- Commit the changes with the *git commit* command

In GIT, there are 3 phases -

- a. Workspace Area
- b. Staging / Indexing Area
- c. Local Repository



Building Applications with Git

1. Configure your git identity at the user level in your system
git config --global user.name "Yourname"
git config --global user.email "Your MailId"

Note : we can find the the these info inside **~/.gitconfig** file

`git config --list` ⇒ Review the user identity settings

2. Create a directory called git-demo and then initialize a Git repository

```
mkdir git-demo
```

```
cd git-demo
```

```
git init
```

OR

We can clone a remote Git repository from github.com

```
git clone <URL_Of_The_Remote_Repository>
```

3. Create an application, add it to the repository and verify the staged changes.

```
echo "Hello World" > hello.php
```

```
cat hello.php
```

```
touch f1 f2
```

`git status` ⇒ The file will be appeared in **red color** which means it is in the workspace area

Staging Changes ⇒ `git add` ⇒ Creating a Commit ⇒ `git commit`

`git add .` ⇒ will stage will untracked files

`git add hello.php` ⇒ will stage a specific file

`git status` ⇒ The file will be appeared in green color which means it is in the staging area

`git commit` ⇒ will record changes to the repository

```
git commit -m "Initial commit for hello.php file"
```

```
git status
```

`git log` ⇒ It shows all the commits of the repository in chronological order

`git show` ⇒ This command is used to view the latest commit and the changes made in the repository files.

`git show <commit_id>` ⇒ to display info about a specific commit

Commit ID ⇒ It is a SHA(Secure Hash Algorithm) which is randomly generated. It is a 40 character string(SHA Code). We can use 7 characters from the commit id with `git show` command.

Note : From August 13, 2021, GitHub is no longer accepting account passwords when authenticating Git operations. You need to add a **PAT (Personal Access Token)** instead, and you can follow the below method to add a PAT on your system.

Create Personal Access Token on GitHub

From your GitHub account, go to **Settings => Developer Settings => Personal Access Token => Generate New Token** (Give your password) => **Fillup the form** => click **Generate token** => **Copy the generated Token**, it will be something like *ghp_sFhFsSHhTzMDreGRLjmks4Tzuzgthdvfsrta*

git push origin main => It will ask for password [paste the PAT here]

Once GIT is configured, we can begin using it to access GitHub. Example:

```
$ git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
```

```
$ Username for 'https://github.com' : username
```

```
$ Password for 'https://github.com' : give your personal access token here
```

Now cache the given record in your computer to remember the token:

```
$ git config --global credential.helper cache
```

If needed, anytime you can delete the cache record by:

```
$ git config --global --unset credential.helper
```

```
$ git config --system --unset credential.helper
```

1. Push the changes to the Remote Repository from the Local Repository git push origin master
 - Origin is the local abbreviated name for the Remote Repository
 - main is the branch name of the Local Repository
2. Make some changes in the source code and stage and commit the changes

```
git add .
git status
git commit -m "New functionality has been added to the code"
git status
git push origin main
```

Creating Git Branches

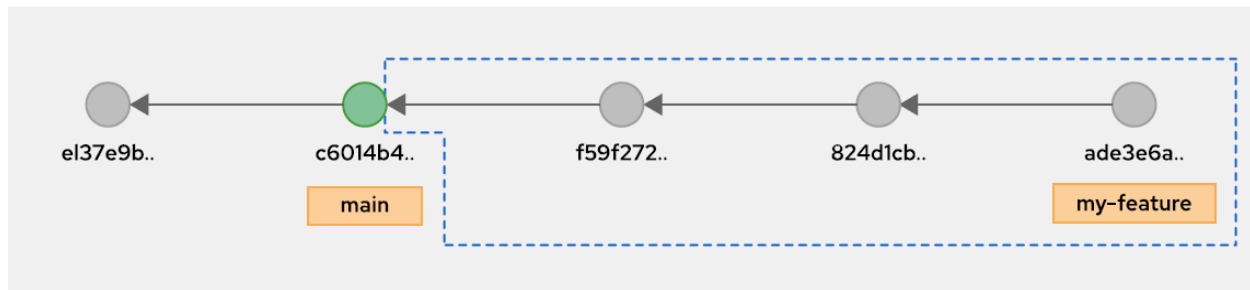
1. The nature of software development is that the code of applications grows and evolves complexity over time.
2. What was once a small application is now a large project with multiple developers collaborating in the same repository simultaneously.
3. A project might have different features and fixes in development at the same time. Those streams of work must happen in parallel to optimize the time to market of your project.
4. With Git, you can organize those streams into collections of code changes named **branches**.
5. Once a stream of work is ready, its branch is *merged back into the main branch*.

Establishing a Main Branch

1. As you work on a repository, it is best to establish a primary or main branch.
2. At any given point, this branch is considered the most recent version of the repository.
3. Creating a new Git repository via GitHub creates a default branch named **main**, whereas using the *git init* command produces a **master** branch.
4. You can change the default branch for existing repositories or configure Git to use a custom default branch name.

Defining Branches

1. Conceptually, a branch houses a collection of commits that implement changes not yet ready for the main branch.
2. Contributors should create commits on a new branch while they work on their changes.



Commits in a branch not yet ready for the main branch

In the preceding example, the my-feature branch incorporates the commits with hashes ade3e6a, 824d1cb, and f59f272. You can quickly compare branches by using git log.

3. Git tracks your current branch by maintaining a special pointer named **HEAD**. For example, when you run *git checkout main*, git attaches the HEAD pointer to the main branch.

How Git Stores Branches

Git does not store the contents individually for each branch. Instead, it stores a key-value pair with the name of the branch and a commit hash. For example, in the previous image, Git simply stores the pair: **my-feature: ade3e6a**.

Git Branch Commands

1. To list all branches ⇒ ***git branch***
2. To create a new branch ⇒ ***git branch development***. Note that the git branch command only creates the branch, it does not switch you to that branch.
3. To delete a branch ⇒ ***git branch -d development***
4. Navigating / Switching branches ⇒ ***git checkout development***
5. A common shortcut to create a branch and check it out in one go is to use the **-b** flag, for example:
git checkout -b testing
6. To rename a branch ⇒ ***git branch -M test***
7. To get help on a particular git command ⇒ ***git add --help***

Merging

1. Once a branch is considered done, then its changes are ready to be incorporated back into the main branch. This is called ***merging***.
2. The Git command for merging is ***git merge***. It accepts a single argument for the branch to be merged from.
3. Implicitly, your currently checked out branch is the branch that is merged into.

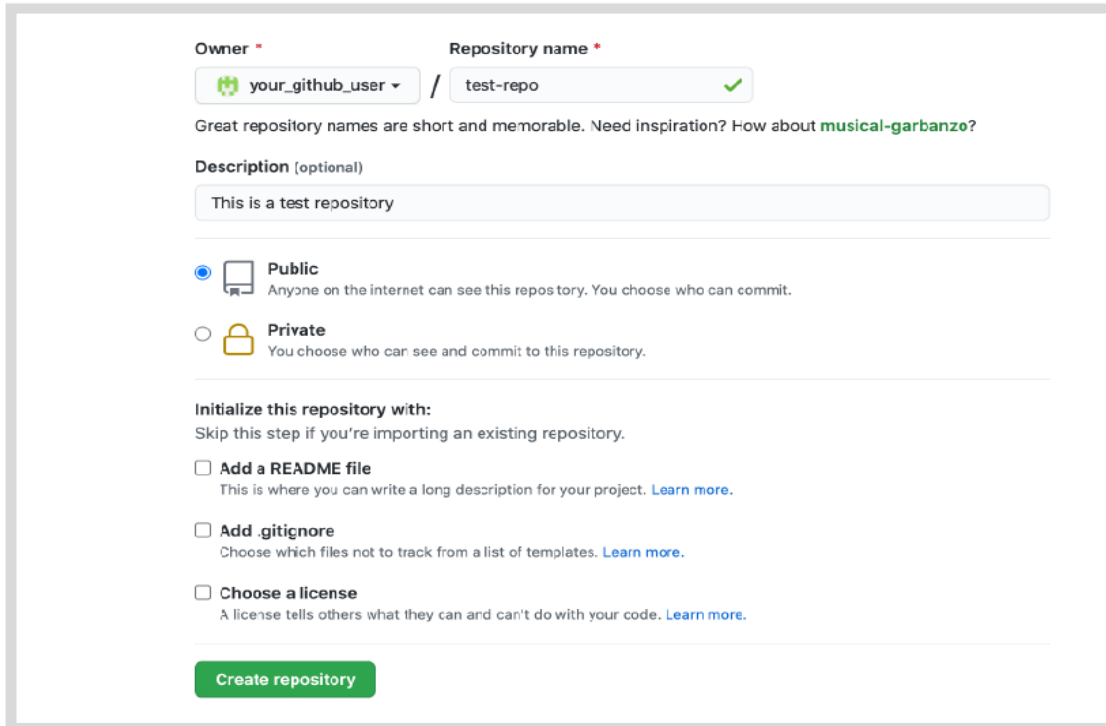
Managing Remote Repositories

1. Git is a distributed and decentralized version control system.
2. With Git, you can have multiple copies of the same repository, without a central server that hosts a primary copy.
3. Although Git does not force you to use a principal repository copy, it does allow you to do so. In fact, it is very common for teams to work with a primary repository, which centralizes the development and integrates the changes made by different contributors.
4. Because the primary repository copy is often remote, this copy is referred to as the remote repository, or simply as the **remote**.
5. From the perspective of a local repository, a remote is simply a reference to another repository. You can add multiple remotes to a local repository to push changes to different remote repositories.
6. Although a remote often points to an external repository it does not necessarily
7. have to do so. You can add a remote that points to another local repository in your workstation.

Working with Remotes

After creating the remote repository, you can add the remote repository as a new remote to the local repository.

Creating a New GitHub Repository

The image shows the GitHub 'Create new repository' form. At the top, there are two input fields: 'Owner' with a dropdown menu showing 'your_github_user' and a green checkmark, and 'Repository name' with the text 'test-repo' and a green checkmark. Below these fields is a line of text: 'Great repository names are short and memorable. Need inspiration? How about musical-garbanzo?'. Underneath is a 'Description (optional)' text area containing the text 'This is a test repository'. Below the description are two radio button options: 'Public' (selected) with a subtext 'Anyone on the internet can see this repository. You choose who can commit.', and 'Private' with a subtext 'You choose who can see and commit to this repository.'. Below these are three checkboxes under the heading 'Initialize this repository with:'. The first checkbox is 'Add a README file' with subtext 'This is where you can write a long description for your project. Learn more.'. The second checkbox is 'Add .gitignore' with subtext 'Choose which files not to track from a list of templates. Learn more.'. The third checkbox is 'Choose a license' with subtext 'A license tells others what they can and can't do with your code. Learn more.'. At the bottom is a green button labeled 'Create repository'.

GitHub new repository form



Note

It is good practice to add a description, as well as the following files to your GitHub repositories:

- A *README* file, which describes and explains your project.
- A *.gitignore* file, which indicates to Git what files or folders to ignore in a project.
- A *LICENSE* file, which defines the current license of your project.

After you have created the remote repository in GitHub, you can start adding code. To start working in the new repository from your local workstation, you generally have two options:

- Clone the remote repository as a new local repository in your workstation.
- Add the remote repository as a remote to a local repository in your workstation.

Cloning a Repository

1. When you clone a remote repository, Git downloads a copy of the remote repository and creates a new repository in your local machine. This newly created repository is the local repository.
2. To clone a repository, use the git clone command, specifying the URL of the repository that you want to clone.

```
[user@host ~]$ git clone https://github.com/YOUR_GITHUB_USER/your-repo.git
Cloning into 'your-repo'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 12 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (12/12), done.
```

Remote Branches

When working with remote repositories, you can have both remote and local branches. If you work on a local repository connected with a remote repository, then you will have the following branch categories:

- **Local branches:** Branches that exist in your local repository. You normally organize, carry out your work, and commit changes by using local branches. For example, if you use the git branch command to create a new branch called my-feature, then my-feature is a local branch.
- **Remote branches:** Branches that exist in a remote repository. For example, all the branches in a GitHub repository are considered remote branches. Teams use remote branches for coordination, review and collaboration.
- **Remote-tracking branches:** Local branches that reference remote branches. Remote Tracking branches allow your local repository to push to the remote and fetch from the remote. Git names these branches by using the following convention: remote_name/branch_name. To download changes from the remote, you first fetch the remote changes into a remote-tracking branch and then merge it into the local branch.

You can also pull changes to update the local branch. To upload changes to the remote, you must push the changes from your local branch to a remote-tracking branch for Git to update the remote.

Adding a Remote to a Local Repository

1. As a part of the clone process, Git configures the local repository with a remote called **origin**, which points to the remote repository.
2. After cloning the repository, you can navigate to the newly created local repository folder, and check that Git has configured a remote that points to the remote repository. Use the **git remote -v** command for this.

```
[user@host ~]$ cd your-repo
[user@host your-repo]$ git remote -v
origin https://github.com/your_github_user/your-repo.git (fetch)
origin https://github.com/your_github_user/your-repo.git (push)
```

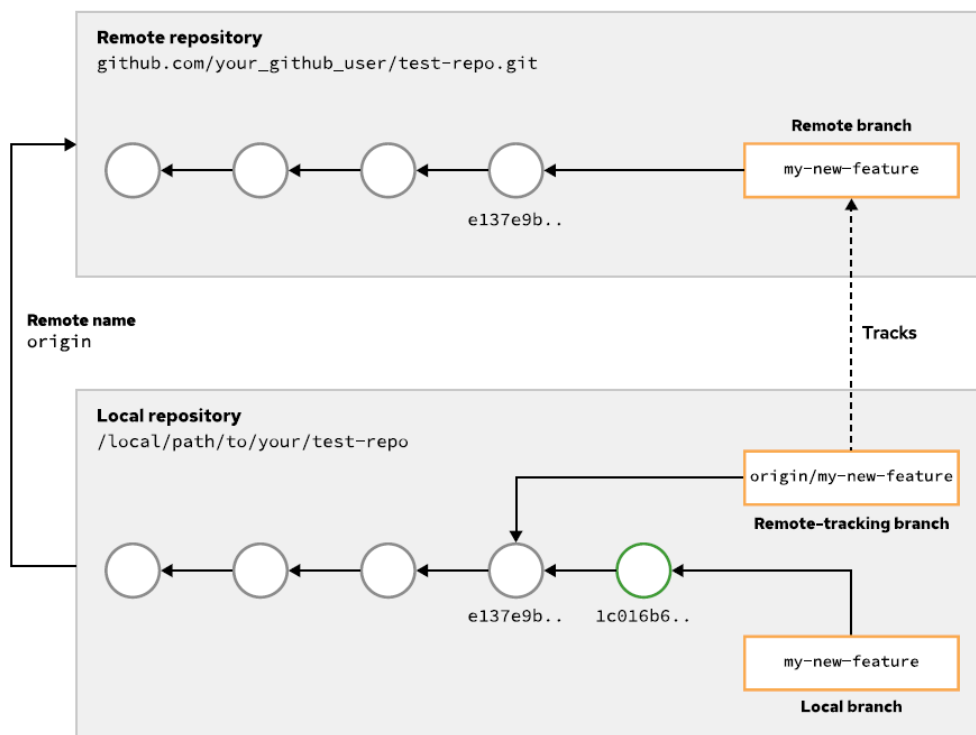
- When you add a new remote to an existing local repository, you must identify the remote by choosing a remote name, as well as specify the URL of the remote repository.

To add a new remote, use the `git remote add` command, followed by a name and the URL of the remote repository.

```
[user@host your-local-repo]$ git remote add \
> my-remote https://github.com/YOUR_GITHUB_USER/your-repo.git
[user@host your-local-repo]$
```

Pushing to the Remote

When you create new commits in your local branch, the remote-tracking branch and the remote branch become outdated with respect to the local branch, as shown in the following figure:



The local `my-new-feature` branch is ahead of the `origin/my-new-feature` branch and the remote `my-new-feature` branch

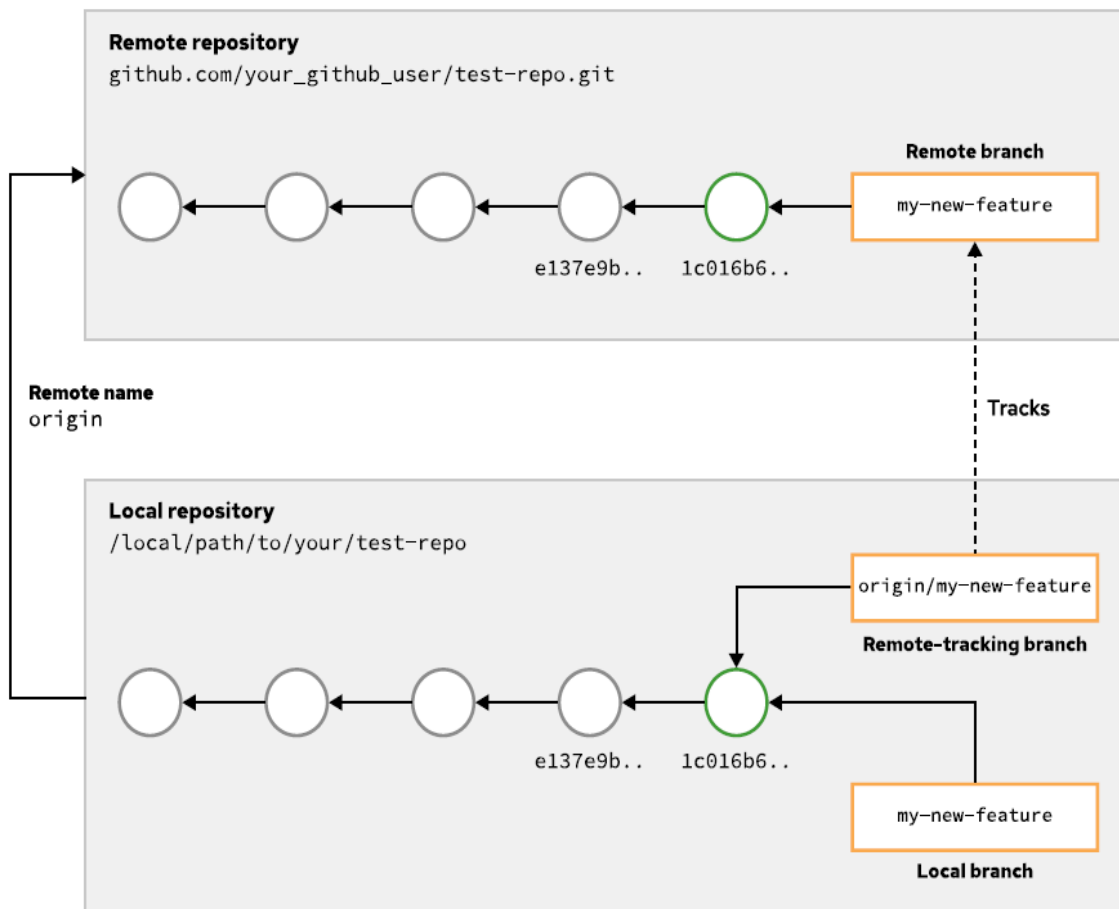
The figure shows how the new 1c016b6.. commit is in the local branch only. To update the remote branch with your new commit, you must push your changes to the remote.

```
[user@host your-local-repo]$ git status
On branch my-new-feature 1
...output omitted...
[user@host your-local-repo]$ git push origin my-new-feature 2
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
Total 3 (delta 0), reused 1 (delta 0)
To https://github.com/your_github_user/test-repo.git
e137e9b..1c016b6 my-new-feature -> my-new-feature
```

The git push command pushes the currently selected **my-new-feature** local branch to the remote my-new-feature branch in GitHub and consequently updates the remote-tracking origin/my-new-feature branch.

1. Check that the currently selected local branch is my-new-feature.
2. Push the local my-new-feature branch to the GitHub my-new-feature branch and update the remote-tracking origin/my-new-feature branch.

After the push, the local, remote, and remote-tracking branches all point to the same commit. At this point, the remote branch is synchronized with the local branch.



The remote branch is up to date with the local branch

Lab :

```
def say_hello(name):
    print(f"Hello, {name}!")
say_hello("world")
```

Pull Requests

When you finish your work in a branch, normally you would want to propose your work for integration. To propose changes to the main development branch in the GitHub remote, you can either just merge and push the changes, or instead use pull requests.

If you just merge your work into the main branch and push it to the remote, then your team does not have a chance to review your changes. In contrast, if you use a pull request, then your code goes through manual and automatic validations, which improve the quality of your work.

Pull requests are a powerful web-based feature

Pull requests help achieve the following:

⇒ Compare branches

- ⇒ Review proposed changes
- ⇒ Add comments
- ⇒ Reject or accept changes

Stash Memory ⇒ is a temporary memory provided by git

To define the stash memory ⇒ **git stash**

To list the stash memory ⇒ **git stash list** ⇒ `stash@{0}`

To bring back the files from stash memory into the staging area ⇒ **git stash pop**

To copy the files from stash memory into the staging area ⇒ **git stash apply stash{0}**

To list the contents of the stash memory referred by `stash@{0}` ⇒ **git stash show -p stash@{0}**

To undefine the stash memory at `stash@{0}` ⇒ **git stash drop stash{0}**

Defining Development Workflows

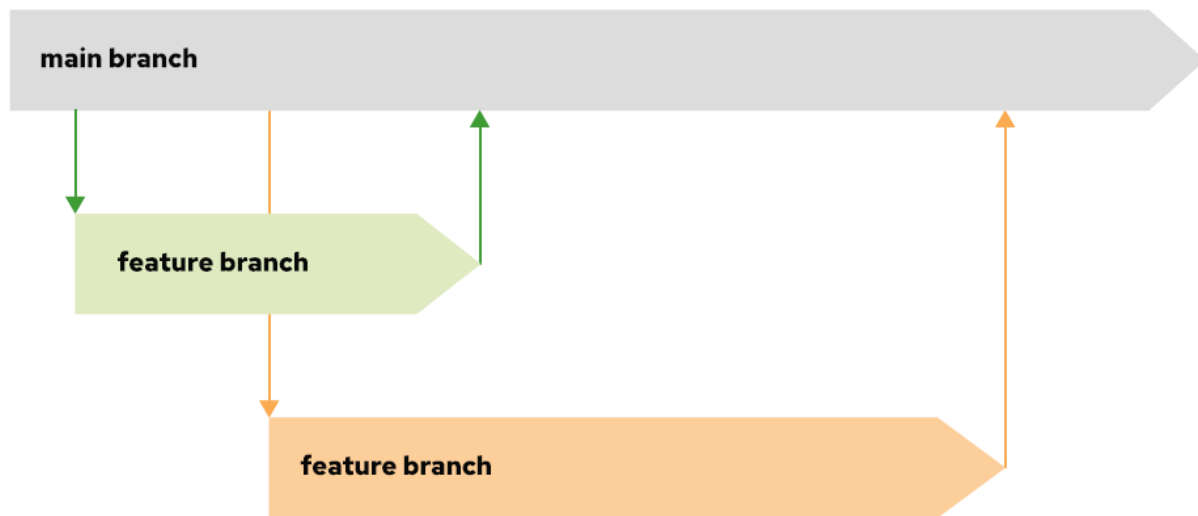
- ❖ Developers love finding reusable solutions to problems.
- ❖ We can follow one of the many workflows available to guide the development process.
- ❖ A workflow consists of a defined set of rules and processes, which guide your development process.
- ❖ Most of the popular development workflows share the same core idea. They want to facilitate the collaboration between the developers involved in a project. In some cases, this collaboration includes the use of branches, pull requests, and forks.
- ❖ A fork is an independent copy of a repository, including branches. It can be used to contribute to projects or to start an independent line of development.
- ❖ A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Trunk-based Development

1. In this workflow, developers avoid the creation of long-lived branches. Instead, they work directly on a shared branch named `main`, or use short-lived branches to integrate the changes.
2. When developers finish a feature or a fix, they merge all changes to the `main` branch.

Feature Branching

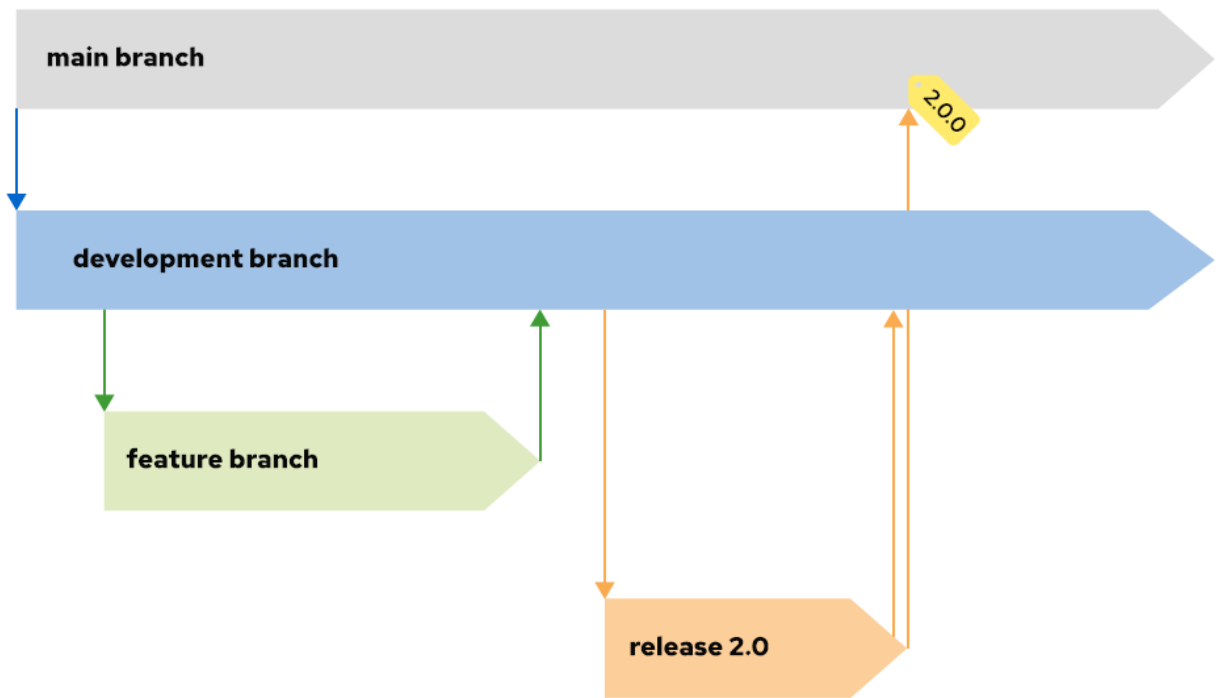
1. In this workflow, developers work in dedicated branches, and the `main` branch only contains the stable code.
2. Developers create a new long-lived branch from the `main` branch every time they start working on a new feature. The developer merges the feature branch into the `main` branch as soon as they finish the work.



Branching structure in Feature Branching

Git Flow

- Git Flow is a well-known development workflow created in 2010 and inspired by the Feature Branching workflow.
- Usually, this workflow has a central repository, which acts as the single source of truth. The central repository holds the following principal branches:
 - a. main: this branch contains all the production-ready code.
 - b. develop: this branch contains all the changes for the next release.
- A feature branch uses the develop branch as the starting point. When the work is done in a feature branch, you merge back this branch with the develop branch.



Branching structure in Git Flow

- A release branch uses the develop branch as the starting point.
- We can only add minor bug fixes, and small meta-data changes to this branch. When we finish all the work in a release branch, we must merge the branch in the development and main branches. After the merge, you must add a tag with a release number.

Releasing Code

1. As a software developer, one of your main goals is to deliver **functional versions** of your application to your users. Those functional versions are called **releases**.
2. The releasing process involves multiple steps, such as planning the release, running quality checks and generating documentation.
3. *Naming Releases* ⇒ One of the most important steps involved in the release of a new version of your application is appropriately versioning and naming the release.
4. Naming a release is assigning a *unique name* to a specific point in the commit history of your project.
5. There is no industry standard for how to uniquely name a release, but there are multiple version naming conventions you can follow. One of the most used naming conventions is **Semantic Versioning**.
6. *Semantic Versioning* ⇒
 - a. The semantic version specification is a set of rules that indicate how version numbers are assigned and increased.

- b. This specification names releases with a series of 3 sets of numbers in the format *MAJOR.MINOR.PATCH*.
- c. The MAJOR version increases when you make incompatible API changes.
- d. The MINOR version increases when you add features that are backwards compatible.
- e. The PATCH version increases when you add bug fixes that are backwards compatible.
- f. A MAJOR version of zero implies the project is in early stages.

Creating Releases with Git

- Tags represent a particular commit / release
- We can use Git tags to uniquely name the releases or versions.
- In Git, a tag is a *static pointer to a specific moment in the commit history*, such as releases or versions.
- Git supports 2 types of tags ⇒ **annotated** and **lightweight**. The main difference between the 2 types is the amount of metadata they store.

In the lightweight tag, Git stores the name and the pointer to the commit. The ***git tag*** command creates a lightweight tag.

Git stores annotated tags as full objects which include metadata such as the date, the details of the user who created the tag and a comment. The ***git tag*** command with the **-a** option creates an annotated tag.

```
git tag -a 1.2.3
```

When we have our code with a tag assigned, we can distribute it to our users.

- To list all tags ⇒ `git tag`
- To create a tag ⇒ `git tag 1.2.3`
- To delete a tag ⇒ `git tag -d 1.2.3`
- After you have your code with a tag assigned, you can distribute it to your users.
- *When we delete a tag from a local workstation, it does not delete the tag from the GitHub repository.* If we want to delete the tags from the GitHub repo also, we need to give the following command :

```
git push origin :refs/tags/<tag_name>
```

Lab : Releasing code

In this exercise you will learn how to use tags, releases, and forks.

To set up the scenario necessary to complete this exercise, first you must create a new GitHub organization with your GitHub account. Next, you will create a repository in the organization by forking the source code we provide at <https://github.com/RedHatTraining/DO400-apps-external>. The repository forked in your organization will simulate a third-party project in which to contribute.

With the organization set up as a hypothetical third-party project, you will practice a common contribution workflow, forking the third-party project into your username account and creating pull requests back to the third-party project to propose code changes.

1. Create a new GitHub organization with your GitHub account.
2. Create a repository in the organization by forking the source code provided at <https://github.com/RedHatTraining/DO400-apps-external> [Upstream Repository]
3. The repository forked in your organization will simulate a third-party project in which to contribute.
4. Fork the upstream repository to start contributing to the upstream project.
At this point, you have set up the scenario to contribute to a third-party project.
You will treat the repository you have just created as the **upstream repository**, meaning that you will consider this repository as the third-party project you send your contributions to.
5. Fork the upstream repository to start contributing to the upstream project. Fork the YOUR_GITHUB_USER-do400-org/DO400-apps-external repository to your username account and clone the forked repository located at https://github.com/YOUR_GITHUB_USER/DO400-apps-external.
6. Clone the repository using git clone command
https://github.com/YOUR_GITHUB_ACCOUNT/DO400-apps-external.git
7. Create a tag. Next, create a release by using the created tag.
cd DO400-apps-external
git tag 1.0.0
git tag
git push origin --tags [to push the tag to your username fork in GitHub]

=====

What is a GitLab Server?

Gitlab CE or Community Edition is an open-source application used to host your Git repositories. It offers you the advantage of keeping the data on your server for your team and your clients.

Differences between GitHub and GitLab

Reference : <https://about.gitlab.com/devops-tools/github-vs-gitlab/>

The core difference is **GitLab has Continuous Integration/Continuous Delivery (CI/CD) and DevOps workflows built-in**. GitHub lets you work with the CI/CD tools of your choice, but you'll need to integrate them yourself. Typically, GitHub users work with a third-party CI program such as Jenkins, CircleCI, or TravisCI.

Lab : Setup a GitLab Server [Reference : <https://about.gitlab.com/install/>]

1. GitLab is open source and free software
2. Unlike GitHub, it can be installed on a local server or cloud based vm/instance
3. GitHub : Public Repository is free but Private Repository is paid
GitLab : Private Repository is free
4. GitLab comes in 2 editions -
 - a. Enterprise Edition (EE) ⇒ Paid version
 - b. Community Edition (CE) ⇒ Free version
5. Components inside GitLab ⇒
 - a. git
 - b. nginx
 - c. PostgreSQL ⇒ Relational Database Management System
 - d. Chef ⇒ Configuration Management Tool
 - e. Redis ⇒ In-Memory database
5. Official URL ⇒ <https://about.gitlab.com/install/>

=====

What is Maven?

Maven is a build tool.

Apache Maven is a software *project management* and comprehension tool. Based on the concept of a *project object model (POM)*, Maven can manage a project's build, reporting and documentation from a central piece of information.

Apache Maven *is a build tool mainly for Java applications* to help the developer at the whole process of a software project.

It is also a *Project management tool*. Using Maven, we will generate a folder structure.

By using maven, we will generate a **WAR** file which will be deployed in different environments(dev,test,uat,prod).

Source Code ⇒ Github Repository ⇒ Build Engineer will pull the code ⇒ Build the code ⇒ WAR File ⇒ Deployed in dev/test/prod Environment

Main purpose of Maven is to build and deploy the code.

What Maven does?

Compilation of Source Code (hello.java)

```
javac hello.java
```

```
hello.class ....BYTECODE
```

```
java hello
```

Running Tests (unit tests and functional tests)

Packaging the results into JAR's, WAR's, RPM's etc..

Upload the packages to remote repo's (Nexus,Artifactory)

Important Concepts

1. Maven Build Lifecycle

Maven defines and follows conventions. Right from the project structure to building steps, Maven provides conventions to follow. If we follow those conventions, with minimal configuration we can easily get the build job done.

For example, the default lifecycle comprises of the following phases ::

- **validate** - validate the project is correct and all necessary information is available
- **compile** - compile the source code of the project
- **test** - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** - take the compiled code and package it in its distributable format, such as a JAR/WAR.
- **verify** - run any checks on results of integration tests to ensure quality criteria are met
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

For the above command, starting from the first phase, all the phases are executed sequentially till the 'install' phase.

A goal represents a specific task which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.

The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The clean and package arguments are build phases while the dependency:copy-dependencies is a goal.

mvn clean dependency:copy-dependencies package

Here the clean phase will be executed first, followed by the dependency:copy-dependencies goal, and finally the package phase will be executed.

2. Maven Repository

Repository is where the build *artifacts* are stored. *Build artifacts* means, the dependent files (Ex: dependent jar files) and the build outcome (the package we build out of a project).

There are 3 types of repositories, **local, central and remote**.

Local maven repository (**.m2**) is in the user's system. It stores the copy of the dependent files that we use in our project as dependencies.

Remote maven repository is set up by a third party(nexus/jfrog) to provide access and distribute dependent files.

Ex: repo.maven.apache.org from the internet.

Central Repository - Maintained and managed by maven.apache.org

3. POM Example

POM stands for Project Object Model. It is a fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml. And has all the configuration settings for the project build.

Generally we define the project dependencies (Ex: dependent jar files for a project), maven plugins to execute and project description / version etc.

The pom.xml file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project in just the way you want.

Simplest pom.xml should have 4 important information -

- a. modelVersion- This element indicates what version of the object model this POM is using
- b. groupId – This element indicates the unique identifier of the organization or group that created the project. For example org.apache.maven.plugins is the designated groupId for all Maven plugins.
- c. artifactId – This element indicates the unique base name of the primary artifact being generated by this project. A typical artifact produced by Maven would have the form
<artifactId>-<version>.<extension> (for example, myapp-1.0.jar).
- d. version – This element indicates the version of the artifact generated by the project. 1.0-snapshot means the project is in the development stage. We will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development.

Here's an example:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

4. Maven Dependencies

There is an element available for declaring dependencies in project pom.xml This is used to define the dependencies that will be used by the project. Maven will look for these dependencies when executing in the local maven repository. If not found, then Maven will download those dependencies from the remote repository and store it in the local maven repository.

5. Maven Plugins

All the execution in Maven is done by plugins. A plugin is mapped to a phase and executed as part of it. A phase is mapped to multiple goals. Those goals are executed by a plugin. We can directly invoke a specific goal while executing Maven execution. A plugin configuration can be modified using the plugin declaration.

An example for the Maven plugin is 'compiler', it compiles the java source code. This compiler plugin has two goals: compiler:compile and compiler:testCompile.

Maven Project Structure

Maven uses a convention for project folder structure. If we follow that, we need not describe in our configuration setting, what is located where. Maven knows where to pick the source files, test cases etc. Following is a snapshot from a Maven project and it shows the project structure.

What is Archetype?

In short, Archetype is a Maven project templating toolkit. *An archetype is a very simple artifact that contains the project prototype you wish to create.*

Using an Archetype

To create a new project based on an Archetype, you need to call *mvn archetype:generate goal*, as :

mvn archetype:generate

Lab : Install Apache Maven on CentOS 7

1. Install OpenJDK8 on CentOS7

```
# yum install -y java-1.8.0-openjdk-devel
```

```
# java -version
```

2. Install Apache Maven on CentOS7

```
# cd /usr/local/src
```

```
# wget
http://www.apache.org/dist/maven/maven-3/3.5.4/binaries/apache-maven-
3.5.4-bin.tar.gz
```

3. Extract the downloaded archive file, and rename it using following commands.

```
# tar -xvf apache-maven-3.5.4-bin.tar.gz
# mv apache-maven-3.5.4/ apache-maven/
```

4. Configure Apache Maven Environment

Now we need to configure the environment variables to pre-compiled Apache Maven files on our system by creating a configuration file '**maven.sh**' in the '**/etc/profile.d**' directory.

```
# cd /etc/profile.d/
# vim maven.sh
```

Add the following configuration in the '**maven.sh**' configuration file.

```
# Apache Maven Environment Variables
# MAVEN_HOME for Maven 1 - M2_HOME for Maven 2 & 3
export M2_HOME=/usr/local/src/apache-maven
export PATH=${M2_HOME}/bin:${PATH}
# chmod +x maven.sh
# source /etc/profile.d/maven.sh
```

5. Check Apache Maven version

```
mvn --version
```

Clone and Build the gameoflife project

1. Clone the project from github repository
2. cd gameoflife
3. mvn clean install package ⇒ This command should produce a deployable WAR file inside gameoflife/gameoflife-web/target folder as *gameoflife.war*

Tomcat

Reference : <https://tomcat.apache.org/>

- **Apache Tomcat** (called "Tomcat" for short) is a [free and open-source](#) implementation of the [Java Servlet](#), [JavaServer Pages](#), [Java Expression Language](#) and [WebSocket](#) technologies.
- Tomcat provides a "pure Java" [HTTP web server](#) environment in which [Java](#) code can run.
- Tomcat is developed and maintained by an open community of developers under the auspices of the [Apache Software Foundation](#), released under the [Apache License 2.0](#) license.
- It powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations
- Default port of tomcat is **8080**
- Java is the prerequisite for installing tomcat

Components of Tomcat

1. **Catalina** : It is the Servlet Container of Tomcat.
2. **Coyote** : Coyote acts as a connector and supports HTTP 1.1
3. **Jasper** : It is the Tomcat's JSP Engine.
4. **Cluster** : A component for load balancing to manage large applications.
5. **High availability** : A Tomcat component to schedule system upgrades and changes without affecting the live environment.
6. **Web Application** : Manage Sessions, Support deployment across different environments.

Install and Configure Apache Tomcat 9 in CentOS 8/7

Download from <https://tomcat.apache.org/>

```
cd /usr/local/src
```

```
wget https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.64/bin/apache-tomcat-9.0.64.tar.gz
```

```
tar xvf apache-tomcat-9.0.64.tar.gz
```

```
mv apache-tomcat-9.0.64 tomcat9
```

Further Configuration to access Tomcat Web Interface

```
cd /usr/local/src/tomcat9/conf
```

```
vim tomcat-users.xml
```

```
<role rolename="manager-gui"/>
```

```
<user username="admin" password="password" roles="manager-gui"/>
```

By default the Host Manager is only accessible from a browser running on the same machine as Tomcat. If you wish to modify this restriction, you'll need to edit the Host Manager's context.xml file

```
/usr/local/src/tomcat9/webapps/host-manager/META-INF
```

```
cp context.xml context.xml.bak
```

```
vim context.xml
```

```
<Context antiResourceLocking="false" privileged="true" >
```

```
<CookieProcessor className="org.apache.tomcat.util.http.Rfc6265CookieProcessor"
```

```
    sameSiteCookies="strict" />
```

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
```

```
    allow=".*" />
```

```
<Manager
```

```
    sessionAttributeValueClassNameFilter="java\\.lang\\.(?:Boolean|Integer|Long|Number|String)|org\\.apache\\.catalina\\.filters\\.CsrfPreventionFilter|LruCache(?:\\$1)?|java\\.util\\.(?:Linked)?HashMap"/>
```

```
</Context>
```

Stop and Start the Tomcat Server

```
/usr/local/src/tomcat9/bin/shutdown.sh
```

```
/usr/local/src/tomcat9/bin/startup.sh
```

```
=====
```

Managing Containers

Software applications typically depend on other libraries, configuration files, or services provided by their runtime environment. Traditionally, the runtime environment for a software application is installed in an operating system running on a physical host or virtual machine. Any application dependencies are installed along with that operating system on the host.

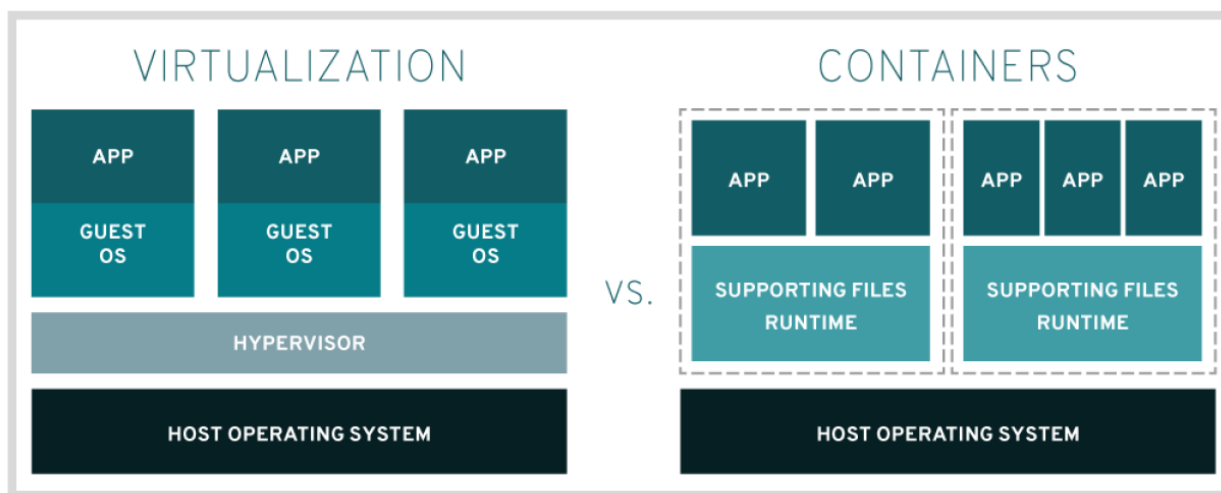
In Red Hat Enterprise Linux, packaging systems like RPM are used to help manage application dependencies. When you install the httpd package, the RPM system ensures that the correct libraries and other dependencies for that package are also installed.

The major drawback to traditionally deployed software applications is that these dependencies are entangled with the runtime environment. An application might require versions of supporting software that are older or newer than the software provided with the operating system. Similarly, two applications on the same system might require different versions of the same software that are incompatible with each other.

One way to resolve these conflicts is to package and deploy the application as a container. A container is a set of one or more processes that are isolated from the rest of the system.

Comparing Containers to Virtual Machines

Both technologies isolate their application libraries and runtime resources from the host operating system or hypervisor and vice versa.



Containers and Virtual Machines are different in the way they interact with hardware and the underlying operating system.

Virtualization:

- Enables multiple operating systems to run simultaneously on a single hardware platform.

- Uses a hypervisor to divide hardware into multiple virtual hardware systems, allowing multiple operating systems to run side by side.
- Requires a complete operating system environment to support the application.

Compare and contrast this with containers, which:

- Run directly on the operating system, sharing hardware and OS resources across all containers on the system. This enables applications to stay lightweight and run swiftly in parallel.
- Share the same operating system kernel, isolate the containerized application processes from the rest of the system, and use any software compatible with that kernel.
- Require far fewer hardware resources than virtual machines, which also makes them quick to start and stop and reduce storage requirements.

Exploring the Implementation of Containers

Red Hat Enterprise Linux implements containers using core technologies such as:

- *Control Groups (cgroups)* for resource management.
- *Namespaces* for process isolation.
- *SELinux and Seccomp (Secure Computing mode)* to enforce security boundaries.

Running Containers from Container Images

- Containers are run from container images.
- Container images serve as blueprints for creating containers.
- Container images package an application together with all its dependencies, such as:
 - System libraries
 - Programming language runtimes
 - Programming language libraries
 - Configuration settings
 - Static data files
- Container images are unchangeable, or *immutable*, files that include all the required code and dependencies to run a container.

Container images are built according to specifications, such as the *Open Container Initiative (OCI)* image format specification. These specifications define the format for container images, as well as the metadata about the container host operating systems and hardware architectures that the image supports.

Designing Container-based Architectures

You could install a complex software application made up of multiple services in a single container. For example, you might have a web server that needs to use a database and a messaging system. But using one container for multiple services is hard to manage.

A better design runs each component, the web server, the database, and the messaging system, in separate containers. This way, updates and maintenance to individual application components do not affect other components or the application stack.

Managing Containers with Podman

Red Hat Enterprise Linux provides a set of container tools that you can use to do this, including:

- **podman**, which directly manages containers and container images.
- **skopeo**, which you can use to inspect, copy, delete, and sign images.
- **buildah**, which you can use to create new container images.

These tools are compatible with the *Open Container Initiative (OCI)*. They can be used to manage any Linux containers created by OCI-compatible container engines, such as Docker

Running Rootless Containers

On the container host, you can run containers as the root user or as a regular, unprivileged user. Containers run by non-privileged users are called **rootless containers**.

Rootless containers are more secure, but have some restrictions. For example, rootless containers cannot publish their network services through the container host's privileged ports (those below port 1024).

You can run containers directly as root, if necessary, but this somewhat weakens the security of the system if a bug allows an attacker to compromise the container.

Container Images and Registries

- A container registry is a repository for storing and retrieving container images.
- Container images are uploaded, or pushed, to a container registry by a developer.
- We download, or pull, those container images from the registry to a local system so that you can use them to run containers.
- You might use a public registry containing third-party images, or you might use a private registry controlled by your organization.
- The source of your container images matters. Just like any other software package, you must know whether you can trust the code in the container image.

- Different registries have different policies about whether and how they provide, evaluate, and test container images submitted to them.
- Red Hat distributes certified container images through two main container registries that you can access with your Red Hat login credentials.
 - registry.redhat.io** for containers based on official Red Hat products.
 - registry.connect.redhat.com** for containers based on third-party products.
- Red Hat is gradually phasing out an older registry, **registry.access.redhat.com**.

To delete all images ⇒ `docker rmi $(docker images -q -a)`

Container Image Naming Conventions

Container images are named based on the following fully qualified image name syntax:

registry_name/user_name/image_name:tag

- The *registry_name* is the name of the registry storing the image. It is usually the fully qualified domain name of the registry.
- The *user_name* represents the user or organization to which the image belongs.
- The *image_name* must be unique in the user namespace.
- The tag identifies the image version. If the image name includes no image tag, then the latest is assumed.

Registry Configuration File : `/etc/containers/registries.conf`

Running Containers

- To run a container on your local system, you must first pull a container image.
- The `podman pull` command pulls the image you specify from the registry and saves it locally:


```
docker pull ubuntu OR docker pull docker.io/ubuntu
```
- Podman stores images locally and you can list them using the `podman images` command:


```
docker images
```
- To run a container from this image, use the `podman run` command.


```
docker run -it --name=myubuntu <image_name> /bin/bash
cat /etc/os-release
exit
```

To execute a command inside a container ⇒ `docker exec -t <container_name>`

`uname -r`

To stop a container ⇒ `docker stop <container_name>`

To remove a container ⇒ `docker rm <container_name>`

To remove a container without stopping it ⇒ `docker rm <container_name> - - force`

Analyzing Container Isolation

- Containers provide run time isolation of resources.
- Containers utilize Linux namespaces to provide separate, isolated environments for resources, such as processes, network communications, and volumes.
- Processes running within a container are isolated from all other processes on the host machine.

View the processes running inside the container:

```
podman run -it registry.redhat.io/ubi7/ubi /bin/bash
```

```
ps aux
```

Note that the user name and ID inside the container is different from the user name and ID on the host machine:

Create a Repository

Create a custom container image from a running container

Push the custom image to container registry

Without exiting the container, to exit ⇒ **<Ctrl>+p+q**

Pull a base image

Start a container

Perform customizations inside the container

Create a new container image using docker commit command

Assign Tag to the newly created image

Push the image to Docker Hub

docker commit ⇒ Creates a new image from a container's changes

docker tag ⇒ Creates a tag TARGET_IMAGE that refers to SOURCE_IMAGE

docker push ⇒ Pushes an image or a repository to a registry

Create a container image from Dockerfile

docker build ⇒ Build an image from a Dockerfile

Lab 1:

1. Pull the image `docker.io/centos:7`.
2. Start the container with an entry point `/bin/bash`.
3. Install the **git** package inside the container.
4. Create a user called `devops`.
5. Create a few files and directories.
6. Clone the **github** repository from your github account.
7. Stop the container.
8. Create a container image from the container using `docker commit` command
9. Assign a tag to the image
10. Push the image to your own repository within `docker.io`

Lab 2: Create a Basic Apache Container image using Dockerfile

1. Create a Dockerfile to build a custom container image based on the following specifications :
 - a. Use the parent image as `docker.io/centos:7`
 - b. Give label and maintainer instruction.
 - c. Install `httpd` package.
 - d. Expose port 80 for the container.
 - e. Configure entrypoint so that the `httpd` process gets started automatically.
 - f. Create a custom `index.html` page
2. Build an image named as `apache` using the Dockerfile.
3. Inspect the image for author, label, exposed port and entry point entries.

Mapping Container Host Ports to the Container

- When you map a network port on the container host to a port in the container, network traffic sent to the host network port is received by the container.
- For example, we could map port 8000 on the container host to port 8080 on the container. The container might be running an `httpd` process that is listening on port 8080. Therefore, traffic sent to the container host port 8000 would be received by the web server running in the container.
- Set up a port mapping with `podman run` by using the `-p` option. It takes two colon-separated port numbers, the port on the container host, followed by the port in the container

`docker run -d -p 8000:8080 registry.redhat.io/rhel8/httpd-24`

The `-p 8000:8080` option maps port 8000 on the container host to port 8080 in the container.

We can use the **`podman port -a`** option to list all port mappings in use.

You must also make sure that the firewall on your container host allows external clients to connect to its mapped port.

Attaching Persistent Storage to a Container

- Storage in the container is ephemeral, meaning that its contents are lost after you remove the container.
- To support containerized applications with this requirement, you must provide the container with persistent storage.

Providing Persistent Storage from the Container Host

- One easy way to provide a container with persistent storage is to use a directory on the container host to store the data.
- Docker / Podman can mount a host directory inside a running container.
- The containerized application sees these host directories as part of the container storage, much like regular applications see a remote network volume as part of the host file system.
- For example, a database container can use a host directory to store database files.

Preparing the Host Directory

When you prepare a host directory, you must configure it so that the processes inside the container can access it. Directory configuration involves:

- Configuring the ownership and permissions of the directory.
- Setting the appropriate SELinux context (**container_file_t**)

Mounting a Volume

After creating and configuring the host directory, the next step is to mount this directory to a container. To mount a host directory to a container, add the **--volume (or -v)** option to the podman run command, specifying the host directory path and the container storage path, separated by a colon:

```
--volume host_dir:container_dir:Z
```

Building a Kubernetes 1.22 Cluster with kubeadm

A. Install Packages

1. Log into the Control Plane Node (**Note: The following steps must be performed on all three nodes.**).
2. Create configuration file for containerd:


```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF
```
3. Load modules:

```
sudo modprobe overlay
sudo modprobe br_netfilter
```

4. Set system configurations for Kubernetes networking:

```
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

5. Apply new settings:

```
sudo sysctl --system
```

6. Install containerd:

```
sudo apt-get update && sudo apt-get install -y containerd
```

7. Create default configuration file for containerd:

```
sudo mkdir -p /etc/containerd
```

8. Generate default containerd configuration and save to the newly created default file:

```
sudo containerd config default | sudo tee /etc/containerd/config.toml
```

9. Restart containerd to ensure new configuration file usage:

```
sudo systemctl restart containerd
```

10. Verify that containerd is running.

```
sudo systemctl status containerd
```

11. Disable swap:

```
sudo swapoff -a
```

12. Disable swap on startup in /etc/fstab:

```
sudo sed -i ' / swap / s/^\(.*\)$/#1/g' /etc/fstab
```

13. Install dependency packages:

```
sudo apt-get update && sudo apt-get install -y apt-transport-https curl
```

14. Download and add GPG key:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

15. Add Kubernetes to repository list:

```
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

16. Update package listings:

```
sudo apt-get update
```

17. Install Kubernetes packages (Note: If you get a dpkg lock message, just wait a minute or two before trying the command again):

```
sudo apt-get install -y kubelet=1.22.0-00 kubeadm=1.22.0-00 kubectl=1.22.0-00
```

18. Turn off automatic updates:

```
sudo apt-mark hold kubelet kubeadm kubectl
```

19. Log into both Worker Nodes to perform previous steps.

B. Initialize the Cluster

1. Initialize the Kubernetes cluster on the control plane node using kubeadm (**Note: This is only performed on the Control Plane Node**):

```
sudo kubeadm init --pod-network-cidr 192.168.0.0/16 --kubernetes-version 1.22.0
```

2. Set kubectl access:

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

=====

To create a regular user called devops

```
useradd -d /home/devops -m -s /bin/bash devops
```

```
passwd devops [ Assign Password for the devops user ]
```

```
id devops
```

Give sudo access to devops user

```
visudo
```

Switch to devops user and execute the following commands -

```
su - devops
```

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

=====

3. Test access to cluster:

```
kubectl get nodes
```

C. Install the Calico Network Add-on

1. **On the Control Plane Node, install Calico Networking:**

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

2. Check status of the control plane node:

```
kubectl get nodes
```

D. Join the Worker Nodes to the Cluster

1. In the **Control Plane Node**, create the token and copy the kubeadm join command (*NOTE: The join command can also be found in the output from kubeadm init command*):
kubeadm token create --print-join-command
2. In both **Worker Nodes**, paste the kubeadm join command to join the cluster. Use sudo to run it as root:
sudo kubeadm join ...
3. In the Control Plane Node, view cluster status (Note: You may have to wait a few moments to allow all nodes to become ready):
kubectl get nodes

Pods - Running containers in Kubernetes

- Pods are the smallest deployable units of computing that we can create and manage in Kubernetes.
- A Pod is a group of one or more containers, with shared storage and network resources.
- Pods have a single IP address that is applied to every container within the pod.
- Pods can be of 2 types - Managed Pods and Unmanaged Pods
- Remember, you CANNOT edit specifications of an existing POD other than the below -
⇒ spec.containers[*].image
⇒ spec.initContainers[*].image
⇒ spec.activeDeadlineSeconds
⇒ spec.tolerations

Labels & Annotations :

1. Both of them are basically Metadata information.
2. Labels can be used for identifying the resource but annotations provide just the metadata info of that resource.
3. Annotations are NOT used for identification purpose
4. Labels are key/value pairs that are attached to objects, such as pods.

How to host a Particular Pod on a specific Node?

1. We can constrain a Pod so that it can only run on a particular set of Node(s).
2. `nodeSelector` is the simplest recommended form of node selection constraint.
3. As cluster admin, set the labels for the worker nodes
kubectl edit node <node_name>
4. `nodeSelector` is a field of PodSpec.

Namespace

1. Namespace allows users to logically group multiple resources.
2. To display the available namespaces -
kubectl get ns

Kubernetes starts with four initial namespaces:

- `default` The default namespace for objects with no other namespace.
- `kube-system` The namespace for objects created by the Kubernetes system.
- `kube-public` This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
- `kube-node-lease` This namespace for the lease objects associated with each node which improves the performance of the node heartbeats as the cluster scales.

Replication Controller

Replication Controller (RC) \Rightarrow A *ReplicationController* ensures that a specified number of pod replicas are running at any point of time.

Pods can be of 2 types -

- Managed* - Pods are managed by RC.
- Unmanaged* - Pods are NOT managed by RC.

Replication Controller \Rightarrow *Replica Count* \Rightarrow *Label Selector* \Rightarrow *Pod Template* \Rightarrow Pods

Replica Set

ReplicaSet \Rightarrow Is the next-generation of Replication Controller

I want to define 3 Labels in RC \Rightarrow

In Replication Controller

selector:

env: dev

env: test

env: prod

In ReplicaSet

selector:

env: * \Rightarrow means all env/test/prod

This is why we say that ReplicaSet is more **expressive**.

kubect! explain replica set

kubect! explain replicaset.apiVersion

DaemonSet \Rightarrow

- \triangleright A *DemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them.
- \triangleright As nodes are removed from the cluster, those Pods are garbage collected.
- \triangleright Deleting a *DemonSet* will clean up the Pods it created.

Some typical uses of a *DemonSet* are:

- running a cluster storage daemon on every node
- running a logs collection daemon on every node
- running a node monitoring daemon on every node

Kubernetes Service

1. A kubernetes service is a resource to make **a single, constant point of entry** to a group of pods providing the same service.
2. Each service has an IP address and port that never changes while the service exists.
3. An easiest way to create a service is to use \Rightarrow `kubectl expose` command
4. Services are agents which connect Pods together or provide access outside of the cluster.
5. The kube-proxy agent watches the Kubernetes API for new services and endpoints being created on each node.
6. Services provide *automatic load-balancing*, matching a label query
7. Unique IP addresses are assigned and configured via the etcd database, so that Services implement iptables to route traffic.
8. Labels are used to determine which Pods should receive traffic from a service.

Service Types

- a. ClusterIP \Rightarrow It is the default service type and only provides access internally within the kubernetes cluster. The range of ClusterIP used is defined via an API server startup option.
- b. NodePort \Rightarrow This is great for debugging or when a static IP address is necessary. The NodePort range is defined in the cluster configuration. By default, the range of NodePorts is **30000-32768**.
- c. LoadBalancer \Rightarrow Was created to pass requests to a cloud provider like GKE / AKS / EKS
- d. ExternalName \Rightarrow Is a newer service. It has no selectors nor does it define ports or endpoints. The redirection happens at the DNS level.

Kubernetes Deployments :: A Deployment is responsible for creating and updating instances of your application.

1. Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it.
2. To do so, you create a Kubernetes Deployment configuration.
3. The Deployment instructs Kubernetes how to create and update instances of your application.
4. Once you've created a Deployment, the Kubernetes control plane schedules the application instances included in that Deployment to run on individual Nodes in the cluster.
5. Once the application instances are created, a Kubernetes *Deployment Controller* continuously monitors those instances.
6. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. **This provides a self-healing mechanism to address machine failure or maintenance.**
7. You can create and manage a Deployment by using the Kubernetes command line interface, `kubectl`. `Kubectl` uses the Kubernetes API to interact with the cluster.
8. When you create a Deployment, you'll need to specify the **container image** for your application and **the number of replicas** that you want to run.
9. You can change that information later by updating your Deployment

Creating a Deployment

The following is an example of a Deployment. It creates a ReplicaSet to bring up three `nginx` Pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Jenkins

Jenkins Terms & Definitions ⇒ <https://www.jenkins.io/doc/book/glossary/>

Reference : Install Jenkins ⇒ <https://linuxize.com/post/how-to-install-jenkins-on-ubuntu-18-04/>

Jenkins Installation

Introduction

In this hands-on lab, we will install Java and Jenkins. Once this is done and the Jenkins instance is up and running, take some time to look around the interface and ensure you are familiar with all the menu items and other parts of the interface.

Solution

Log in to the server using the credentials provided:

```
ssh cloud_user@<PUBLIC_IP_ADDRESS>
```

Install java-1.8.0-openjdk-devel

1. Install Java:

```
sudo yum install -y java-1.8.0-openjdk-devel  
sudo apt install openjdk-11-jdk [ for Ubuntu ]
```

Install the Repo and Key, and Then Install Jenkins

1. Install wget:

```
sudo yum install -y wget
```

2. Download the repo:

```
sudo wget -O /etc/yum.repos.d/jenkins.repo  
https://pkg.jenkins.io/redhat/jenkins.repo
```

3. Import the required key:

```
sudo rpm --import https://pkg.jenkins.io/redhat/jenkins.io.key
```

4. Install Jenkins:

```
sudo yum install -y jenkins
```

5. Enable Jenkins:

```
sudo systemctl enable jenkins
```

6. Start Jenkins:

```
sudo systemctl start jenkins
```

7. In a new browser tab, navigate to `http://<PUBLIC_IP_ADDRESS>:8080`, replacing `<PUBLIC_IP_ADDRESS>` with the IP address of the cloud server provided on the lab page.

8. We'll be taken to an *Unlock Jenkins* page telling us we need to locate the password. In the terminal, run:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

9. Copy the result of the command, as this is the password we need.

10. Paste the password into the *Administrator password* field on the Jenkins browser page.

11. Click **Continue**.

12. Click **Install suggested plugins**. Note: If the install of any plugins fails, just wait a moment and click retry.

13. On the new user creation page, set the following values:

- Username: **student**
- Password: **student**
- Full name: **AEM Student**
- E-mail address: **student@gmail.com**

14. Click **Save and Continue**.
15. Click **Save and Finish**.
16. Click **Start using Jenkins**.
17. Click **Manage Jenkins** in the left-hand
18. menu, and then look around a bit to get familiar with the items in that area.

Jenkins Pipelines

Describing Pipelines

The DevOps principles rely strongly on automation. To achieve reliability, cohesion, and scalability, the DevOps approach makes use of automation tools to implement pipelines.

*A **pipeline** is a series of connected steps executed in sequence to accomplish a task.* Usually, when one of the steps fails, the next steps in the pipeline do not run. In software development the main goal of a pipeline is to deliver a new version of software. Jenkins is a popular open source tool that helps automate building, testing, and deploying applications.

Subsets of steps that share a common objective constitute a pipeline *stage*. Common pipeline stages in software development include the following:

Project : A logical way to describe a workflow that Jenkins should perform, such as building an application. This description includes the definition and configuration of the automated tasks.

Job is a deprecated term and is synonymous with Project.

Checkout : Gets the application code from a code repository.

Build : Combines the application source with its dependencies into an executable artifact.

Step : A single task to perform inside a project.

Stage : A subset of steps with a common objective.

Workspace : The working directory where Jenkins executes projects. In this folder, project executions store data that is either temporary, or reused between multiple project executions of the same project.

Node : A machine, or a container capable of executing projects.

Jenkinsfile : A Jenkinsfile is a file including a set of instructions for automating project tasks.

This file is usually checked in the source control system.

Build Statuses : Aborted // Failed // Successful // Stable // Unstable

Test : Runs automated tests to validate that the application matches the expected requirements.

Release : Delivers the application artifacts to a repository.

Validation and Compliance : Validates the quality and security of the application artifacts.

Deploy : Deploys the application artifacts to an environment.

Writing a Declarative Pipeline : The Jenkins Pipeline is a suite of plug-ins that extend the Jenkins core functionalities. This suite offers a way to define CI/CD pipelines into Jenkins. With the Pipeline suite enabled, you can create projects with the pipeline project type.

Declarative pipelines must start with a pipeline block, and enclose the pipeline definition within this block. The following example illustrates the use of the pipeline block.

```
pipeline {  
    agent ...output omitted...  
    stages {  
        ...output omitted...  
    }  
    post {  
        ...output omitted...  
    }  
}
```

Declarative Pipeline Sections

Agent : The agent section defines conditions to select the machine or container that will execute the pipeline. Keywords allowed for agent are - none, any, node, container

Stages : The stages section defines one or more stages to execute in the pipeline.

Steps : The steps section specifies a list of tasks to execute.

Post : The post section defines additional tasks to execute upon the completion of a pipeline, or a stage.

```

pipeline {
  agent any ❶
  stages { ❷
    stage('Hello') {
      steps { ❸
        echo 'Hello world!'
      }
    }
  }
}

```

- ❶ Specifies that the pipeline should run on any available agent.
- ❷ Defines the list of stages. This pipeline only has one stage, named *Hello*.
- ❸ Defines the list of steps to execute in the *Hello* stage. The *Hello* stage only has the echo step.

Declarative Pipeline Steps

In Jenkins, tasks are functions defined by the plug-ins. You can integrate those functions in your

pipeline scripts. Some of the most *used steps* are the following:

echo : Sends a message to the standard output. It requires one mandatory string parameter with the message to output.

git : Clones a specified repository. It requires a string parameter with the repository. The optional

string branch parameter specifies a branch name. The string url parameter specifies the git repository.

sh : Executes a shell script. It requires a mandatory string parameter with the script or command to execute.

Lab : Building a Basic Declarative Pipeline

Scripted Pipeline

Automation with Ansible

Control Node

Managed Host1 : Workstation

Managed Host2 : Jenkins Slave

Lab Setup

Install and Configure Ansible on RHEL 8 using the steps below.

controlnode ⇒ centos

managedhost1 ⇒ jenkinslave [ubuntu]

managedhost2 ⇒ workstation [centos]

Step 1: Install Python on RHEL 8

Install and Set your default Python on RHEL 8

python3 --version

Step 2: Install EPEL repo on RHEL 8

Add EPEL repository to your RHEL 8 system.

dnf install <https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm>

Install EPEL repo on CentOS 7

yum install epel-release

yum install ansible

Step 3: Update the system and install Ansible on RHEL 8 from it.

dnf update

dnf install ansible

ansible --version

Step 4 : Create users and setup Password Less SSH authentication

On Control Node : Create a user called devops

useradd devops

passwd devops

Setup password-less SSH authentication between control node and managed hosts

- Login as devops user and generate a key pair (Private key and Public key)
su - devops
ssh-keygen
.ssh/id_rsa ⇒ Private Key
.ssh/id_rsa.pub ⇒ Public Key
- Copy the public key file from the control node to all managed hosts
ssh-copy-id username@Node_IP
- Test the connectivity
ssh user@node_IP [should login without any password]

Step 5: Create user on Managed Host, give sudo access and configure SSH server for Password Based Authentication.

On Managed Host : Create a user called ansible

```
useradd ansible  
passwd ansible
```

If the target system is *Ubuntu*, then perform the following commands -

```
useradd -h /home/ansible -m -s /bin/bash ansible  
passwd ansible
```

Configure sudo access for ansible and devops user

```
cd /etc/sudoers.d  
vim devops  
devopsALL=(ALL) NOPASSWD: ALL
```

SSH server configuration

```
vim /etc/ssh/sshd_config  
PasswordAuthentication yes  
systemctl restart sshd
```

Overview

Automation is one of the most critical areas of improvement in most organizations. Today, most companies are in the process of re-inventing themselves in one way or another to add software development capabilities and as such, take full advantage of the digitization of everything. Software development release cycles are changing in order to release faster. Continuous delivery, where every change is potentially its own release is becoming the new standard. Infrastructure is following suit, after all, continuous delivery is not about just software changes but all changes and infrastructure plays a

key role. For any of this to work of course, 100% automation is required. To achieve that goal, an automation language that is easy and applicable to development and operations is needed. Ansible is that language and if you are not on-board yet, now is your chance not to miss the train because it is leaving the station. Ansible is easy, Ansible is powerful and Ansible is flexible.

Infrastructure as Code

1. A good automation system allows you to implement Infrastructure as Code practices.
2. IAC means that you can use a machine-readable automation language to define and describe the state you want your IT infrastructure to be in.
3. If the automation language is represented as simple text files, it can easily be managed in a version control system like software code.
4. The advantage of this is that every change can be checked into the version control system.
5. If you want to revert to an earlier known-good configuration, you simply can check out that version of the code and apply it to your infrastructure.
6. This builds a foundation to help you follow best practices in DevOps.
7. Developers can define their desired configuration in the automation language. Operators can review those changes more easily to provide feedback, and use that automation to reproducibly ensure that systems are in the state expected by the developers.

WHAT IS ANSIBLE?

1. Ansible is an open source automation platform.
2. It is a simple automation language
3. It is also an automation engine that runs Ansible Playbooks.
4. It is a deployer tool
5. It is an orchestration tool
6. It is a configuration management tool
7. Architecture is agentless
8. It uses push based mechanism
9. Ansible is idempotent

Ansible Is Simple

1. Ansible Playbooks provide human-readable automation.
2. No special coding skills are required to write Playbooks

Ansible Is Powerful

1. You can use Ansible to deploy applications, for configuration management, for workflow automation, and for network automation.
2. Ansible can be used to orchestrate the entire application life cycle.

Ansible Is Agentless

1. Ansible is built around an agentless architecture. Typically, Ansible connects to the target hosts it manages using OpenSSH or WinRM and runs tasks, by pushing out small programs called Ansible modules to those hosts.
2. Any modules that are pushed are removed when Ansible is finished with its tasks.
3. We can start using Ansible almost immediately because no special agents need to be deployed to the managed hosts.

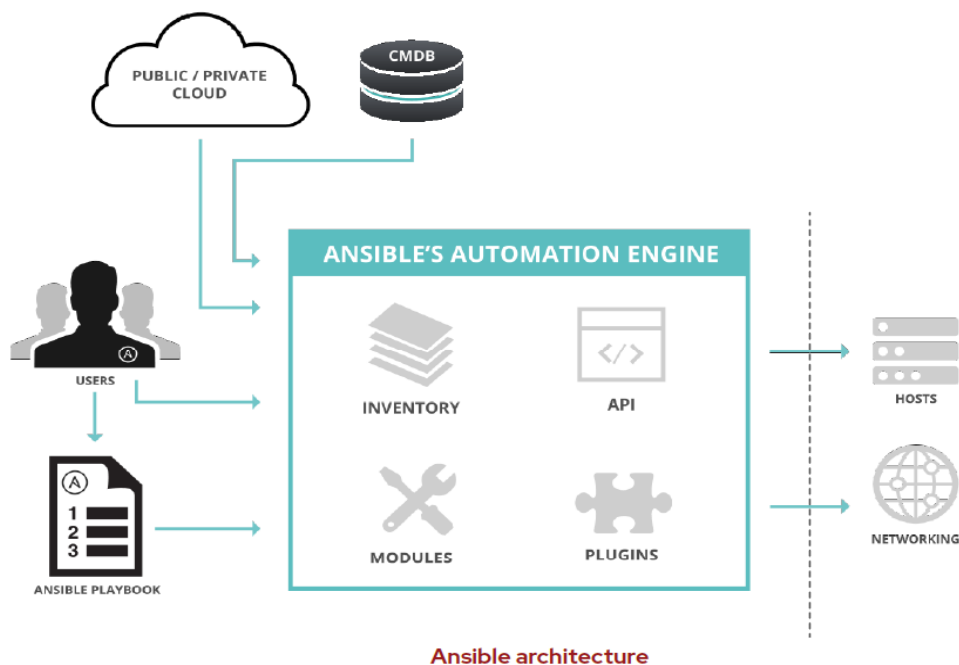
Ansible has a number of important strengths:

- Cross platform support: Ansible provides agentless support for Linux, Windows, UNIX, and network devices, in physical, virtual, cloud, and container environments.
- Human-readable automation: Ansible Playbooks, written as YAML text files, are easy to read and help ensure that everyone understands what they will do.
- Perfect description of applications: Every change can be made by Ansible Playbooks, and every aspect of your application environment can be described and documented.
- Easy to manage in version control: Ansible Playbooks and projects are plain text. They can be treated like source code and placed in your existing version control system.
- Support for dynamic inventories: The list of machines that Ansible manages can be dynamically updated from external sources in order to capture the correct, current list of all managed servers all the time, regardless of infrastructure or location.
- Orchestration : that integrates easily with other systems: HP SA, Puppet, Jenkins, Red Hat Satellite, and other systems that exist in your environment can be leveraged and integrated into your Ansible workflow.

ANSIBLE CONCEPTS AND ARCHITECTURE

- There are two types of machines in the Ansible architecture: **control nodes** and **managed hosts**.
- Ansible is installed and run from a control node, and this machine also has copies of your Ansible project files.
- Managed hosts are listed in an inventory, which also organizes those systems into groups for easier collective management.
- The inventory can be defined in a static text file, or dynamically determined by scripts that get information from external sources.
- **Playbook** : is a collection of plays(tasks). One playbook may consist of a single play or multiple plays. Playbooks are written using YAML (Yet Another Markup Language)
- A play performs a series of tasks on the hosts, in the order specified by the play. A play may have a single task or multiple tasks.
- Each task runs a module, a small piece of code (written in Python, PowerShell, or some other language), with specific arguments.
- Ansible ships with hundreds of useful modules that can perform a wide variety of automation tasks.

- Tasks, plays, and playbooks are designed to be **idempotent**. This means that you can safely run a playbook on the same hosts multiple times.
- When your systems are in the correct state, the playbook makes no changes when you run it.
- Ansible also uses **plug-ins**. Plug-ins are code that you can add to Ansible to extend it and adapt it to new uses and platforms.
- Red Hat **Ansible Tower** is an enterprise framework to help you control, secure, and manage your Ansible automation at scale. It provides a web-based user interface (**web UI**) and a RESTful API. It is not a core part of Ansible, but a separate product that helps you use Ansible more effectively with a team or at a large scale.



SUMMARY

What we have learned:

- Automation is a key tool to mitigate human error and quickly ensure that your IT infrastructure is in a consistent, correct state.
- Ansible is an open source automation platform that can adapt to many different workflows and environments.
- Ansible can be used to manage many different types of systems, including servers running Linux, Microsoft Windows, or UNIX, and network devices.
- Ansible Playbooks are human-readable text files that describe the desired state of an IT infrastructure.
- Ansible is built around an agentless architecture in which Ansible is installed on a control node and clients do not need any special agent software.

- Ansible connects to managed hosts using standard network protocols such as SSH, and runs code or commands on the managed hosts to ensure that they are in the state specified by Ansible.

Managing Ansible Configuration Files

Ansible chooses its configuration file from one of several locations on the control node -

1. `/etc/ansible/ansible.cfg` ⇒ the base configuration file provided by ansible package
2. `~/.ansible.cfg`
3. `./ansible.cfg`
4. Using `ANSIBLE_CONFIG` environment variable

Configuration File Precedence ⇒

`ANSIBLE_CONFIG` ⇒ `ansible.cfg` in current working directory ⇒ `ansible.cfg` in home directory ⇒ the default `/etc/ansible/ansible.cfg`

Managing Settings in the Ansible Configuration File ⇒

The ansible configuration file consists of several sections, with each section containing settings defined as key-value pairs.

- `[defaults]` ⇒ sets the defaults for Ansible operation
- `[privilege_escalation]` ⇒ configures how Ansible performs privilege escalation on managed hosts

ansible --help

Privilege Escalation Options:

control how and which user you become as on target hosts

`--become-method BECOME_METHOD`

privilege escalation method to use (default=sudo), use

``ansible-doc -t become -l`` to list valid choices.

`--become-user BECOME_USER`

run operations as this user (default=root)

`-K, --ask-become-pass`

ask for privilege escalation password

`-b, --become` run operations with become (does not imply password prompting)

DIRECTIVE	DESCRIPTION
inventory	Specifies the path to the inventory file.
remote_user	The name of the user to log in as on the managed hosts. If not specified, the current user's name is used.
ask_pass	Whether or not to prompt for an SSH password. Can be false if using SSH public key authentication.
become	Whether to automatically switch user on the managed host (typically to root) after connecting. This can also be specified by a play.
become_method	How to switch user (typically sudo , which is the default, but su is an option).
become_user	The user to switch to on the managed host (typically root , which is the default).
become_ask_pass	Whether to prompt for a password for your become_method . Defaults to false .

Configuring Connections ⇒

- Ansible needs to know how to communicate with its Managed Hosts.
- By default, Ansible connects to managed hosts using SSH protocol.

Non-SSH Connections ⇒

- The protocol used by Ansible to connect to Managed hosts is set by default to **smart**, which determines the most efficient way to use SSH.
- If we do not have *localhost* in the inventory file, Ansible sets up an *implicit localhost* entry to allow us to run ad hoc commands and playbooks that target *localhost*.

```
[ansible@control rh294]$ ansible localhost --list-hosts
```

```
hosts (1):
```

```
localhost
```

Here, instead of using the **smart SSH** connection type, Ansible connects to the localhost using the special **local connection type** by default.

- The **local** connection type ignores the **remote_user** parameter setting and runs commands directly on the **local** system.

Configuring File Comments ⇒

There are 2 comment characters allowed by Ansible configuration files ⇒ The hash(#) and the semicolon(;))

Running AD HOC Command ⇒

Syntax :

```
ansible host-pattern -m module -a module_arguments
```

Examples :

1. To ping all hosts ⇒ `ansible all -m ping`
2. To display the content of `/etc/redhat-release` file ⇒ `ansible all -m command -a 'cat /etc/redhat-release'`
3. To create a user named bob on the host(s) within the host group called web

```
ansible web -m user -a 'name=bob state=present'
```

```
ansible web -m command -a 'id bob'
```

```
ansible-doc module_name ⇒ will display module arguments
```

```
ansible-doc -l ⇒ lists all modules installed on a system
```

Note : If we do not specify a particular module name in the ansible command, ansible will by default use the **command** module as configured in `/etc/ansible/ansible.cfg` file

Difference between command and shell module ⇒

1. If you want to run a command through the shell (say you are using ``<'`, ``>'`, ``|'`, etc), you actually want the `[shell]` module instead.
2. If you try to execute the built-in bash command **set** with command and shell module, it only succeeds with the **shell** module.

```
ansible localhost -m command -a set ⇒ fails
```

```
ansible localhost -m shell -a set ⇒ succeeds
```

SUMMARY

In this chapter, you learned:

- Any system upon which Ansible is installed and which has access to the required configuration files and playbooks to manage remote systems (**managed hosts**) is called a **control node**.
- Managed hosts are defined in the inventory. Host patterns are used to reference managed hosts defined in an inventory.
- Inventories can be static files or dynamically generated by a program from an external source, such as a directory service or cloud management system.
- Ansible looks for its configuration file in a number of places in order of precedence. The first configuration file found is used; all others are ignored.
- The ansible command is used to perform ad hoc commands on managed hosts.
- Ad hoc commands determine the operation to perform through the use of modules and their arguments, and can make use of Ansible's privilege escalation features.

Implementing Playbooks

- Ad hoc commands can run a single, simple task against a set of targeted hosts as a one-time command.
- The real power of Ansible, however, is in learning how to use playbooks to run multiple, complex tasks against a set of targeted hosts in an easily repeatable manner.
- A play is an *ordered* set of tasks run against hosts selected from your inventory.
- A playbook is a text file containing a list of one or more plays to run in a specific order.
- A playbook is a text file written in **YAML** format, and is normally saved with the extension **.yml**
- The playbook uses *indentation* with space characters to indicate the structure of its data.
- YAML does not place strict requirements on how many spaces are used for the indentation, but there are two basic rules.
 - a. Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.
 - b. Items that are children of another item must be indented more than their parents.
- A playbook begins with a line consisting of three dashes (---) as a start of document marker.
- It may end with three dots (...) as an end of document marker.
- An **item** in a YAML **list** starts with a single dash followed by a space.
- The play itself is a collection of key-value pairs. Keys in the same play should have the same indentation.
- The following example shows a YAML snippet with **three keys**. The first two keys have simple values. The third has a list of *three items as a value*.

name: just an example

hosts: host1

tasks:

- *first*
- *second*
- *third*

The original example play has three keys, name, hosts, and tasks, because these keys all have the same indentation.

- *The first line of the example play starts with a dash and a space* (indicating the play is the first item of a list), and then the first key, the name attribute.
- The name key associates an arbitrary string with the play as a label. This identifies what the play is for.
- The name key is *optional*, but is recommended because it helps to document your playbook. This is especially useful when a playbook contains multiple plays.
- The second key in the play is a hosts attribute, which specifies the hosts against which the play's tasks are run.
- hosts attribute takes a host pattern as a value, such as the names of managed hosts or groups in the inventory.

- The last key in the play is the tasks attribute, whose value specifies a list of tasks to run for this play.

Example : Play with a single task

tasks:

- name: alex exists with UID 5000

user:

name: alex

uid: 4000

state: present

Example : Play with multiple tasks

tasks:

- name: web server is enabled

service:

name: httpd

enabled: true

- name: NTP server is enabled

service:

name: chronyd

enabled: true

Running Playbooks

The ansible-playbook command is used to run playbooks.

ansible-playbook example.yml

- Note that the value of the name key for each play and task is displayed when the playbook is run.
- The *Gathering Facts task is a special task* that the setup module usually runs automatically at the start of a play.

Syntax Verification

The ansible-playbook command offers a --syntax-check option that you can use to verify the syntax of a playbook.

ansible-playbook --syntax-check example.yml

Executing a Dry Run

You can use the -C option to perform a dry run of the playbook execution. This causes Ansible to report what changes would have occurred if the playbook were executed, but does not make any actual changes to managed hosts.

ansible-playbook -C webserver.yml

Implementing Multiple Plays

- A playbook is a YAML file containing a list of one or more plays.
- This can be very useful when orchestrating a complex deployment which may involve different tasks on different hosts.
- You can write a playbook that runs one play against one set of hosts, and when that finishes runs another play against another set of hosts.
- *Each play in the playbook is written as a top-level list item in the playbook.*

Example

```
---
# This is a simple playbook with two plays
- name: first play
  hosts: web
  tasks:
    - name: first task
      yum:
        name: httpd
        status: present
- name: second play
  hosts: db
  tasks:
    - name: first task
      service:
        name: mariadb
        enabled: true
...
```

Remote Users and Privilege Escalation in Plays

- Plays can use different remote users or privilege escalation settings for a play than what is specified by the defaults in the configuration file.
- The following example demonstrates the use of these keywords in a play:

```
- name: /etc/hosts is up to date
  hosts: datacenter-west
  remote_user: remote
  become: yes
  tasks:
```

```
- name: server.example.com in /etc/hosts
  lineinfile:
    path: /etc/hosts
    line: '192.168.0.150 server.example.com server'
    state: present
```

PLAYBOOK SYNTAX VARIATIONS

- **YAML Comments** : Comments can also be used to aid readability. In YAML, everything to the right of the number or hash symbol (#) is a comment.
eg. # This is a YAML comment
- **YAML Strings** : Strings in YAML do not normally need to be put in quotation marks even if there are spaces contained in the string.
eg. this is a string
 'this is another string'
 "this is yet another string"

- **YAML Dictionaries** : collections of key-value pairs written as an indented block, as follows:
 name: svcrole
 svcservice: httpd
 svcport: 80

Dictionaries can also be written in an inline block format enclosed in curly braces, as follows:

```
{name: svcrole, svcservice: httpd, svcport: 80}
```

- **YAML Lists** : lists written with the normal single-dash syntax:
 hosts:
 - *servera*
 - *serverb*
 - *serverc*

Lists also have an inline format enclosed in square braces, as follows:

```
hosts: [servera, serverb, serverc]
```

Managing Variables and Facts

Ansible Variables

- Ansible supports variables that can be used to store values.
- Variables provide a convenient way to manage dynamic values for a given environment in your ansible project

Naming Variables

- Variable names must start with a letter, and they can only contain letters, numbers, and underscores.

Examples of Invalid and Valid Ansible Variable Names

INVALID VARIABLE NAMES	VALID VARIABLE NAMES
web server	web_server
remote.file	remote_file
1st file	file_1 file1
remoteserver\$1	remote_server_1 remote_server1

Defining Variables

- Variables can be defined in a variety of places in an Ansible project.
- It can be simplified to *three basic scope levels*:
 - Global scope: Variables set from the command line or Ansible configuration.
 - Play scope: Variables set in the play.
 - Host scope: Variables set on host groups and individual hosts by the inventory.

Using Variables in Playbooks

- After variables have been declared, administrators can use the variables in tasks.
- Variables are referenced by placing the variable name in double curly braces ({{ }}). Ansible substitutes the variable with its value when the task is executed.

```
vars:
  user: joe

tasks:
  # This line will read: Creates the user joe
  - name: Creates the user {{ user }}
    user:
      # This line will create the user named Joe
      name: "{{ user }}"
```

It is possible to define playbook variables in external files. In this case, instead of using **vars** block, the **vars_files** directive may be used.

- hosts: all
- vars_files:
 - vars/users.yml

Host Variables and Group Variables

- Inventory variables that apply directly to hosts fall into two broad categories:
 - host variables* ⇒ apply to a specific host
 - group variables* ⇒ apply to all hosts in a host group
- One way to define host variables and group variables is to do it directly in the inventory file. This is an older approach and not preferred, but you may still encounter it.
 1. Defining the `ansible_user` host variable for `demo.example.com`:

```
[servers]
demo.example.com  ansible_user=joe
```

2. Defining the user group variable for the servers host group.

```
[servers]
demo1.example.com
demo2.example.com

[servers:vars]
user=joe
```

3. Defining the user group variable for the servers group, which consists of two host groups each with two servers.

```
[servers1]
demo1.example.com
demo2.example.com

[servers2]
demo3.example.com
demo4.example.com

[servers:children]
servers1
servers2

[servers:vars]
user=joe
```

Managing Secrets

- Ansible may need access to *sensitive data* such as passwords or API keys in order to configure managed hosts.
- Normally, this information might be stored as plain text in inventory variables or other Ansible files.
- In that case, however, any user with access to the Ansible files or a version control system which stores the Ansible files would have access to this sensitive data. This poses an obvious security risk.
- **Ansible Vault**, which is included with Ansible, can be used to *encrypt and decrypt* any structured data file used by Ansible.
- To use Ansible Vault, a command-line tool named **ansible-vault** is used to create, edit, encrypt, decrypt, and view files.

Creating an encrypted File

```
[student@demo ~]$ ansible-vault create secret.yml
New Vault password: redhat
Confirm New Vault password: redhat
```

Instead of entering the vault password through standard input, you can use a vault password file to store the vault password.

```
[student@demo ~]$ ansible-vault create --vault-password-file=vault-pass secret.yml
```

The cipher used to protect files is AES256 in recent versions of Ansible, but files encrypted with older versions may still use 128-bit AES.

Viewing an Encrypted File

```
[student@demo ~]$ ansible-vault view secret1.yml
Vault password: secret
less 458 (POSIX regular expressions)
```

Editing an existing Encrypted File

```
[student@demo ~]$ ansible-vault edit secret.yml
Vault password: redhat
```

Encrypting an existing File

```
[student@demo ~]$ ansible-vault encrypt secret1.yml secret2.yml
New Vault password: redhat
Confirm New Vault password: redhat
Encryption successful
```

Decrypting an existing Encrypted File

```
[student@demo ~]$ ansible-vault decrypt secret1.yml --output=secret1-decrypted.yml
Vault password: redhat
Decryption successful
```

Changing the Password of an Encrypted File

```
[student@demo ~]$ ansible-vault rekey secret.yml
Vault password: redhat
New Vault password: RedHat
Confirm New Vault password: RedHat
Rekey successful
```

Playbooks and Ansible Vault

To run a playbook that accesses files encrypted with Ansible Vault, you need to provide the encryption password to the `ansible-playbook` command. If you do not provide the password, the playbook returns an error:

```
[student@demo ~]$ ansible-playbook site.yml
ERROR: A vault password must be specified to decrypt vars/api_key.yml
```

Managing Facts

- Ansible facts are variables that are automatically discovered by Ansible on a managed host.
- Facts contain host-specific information that can be used just like regular variables in plays, conditionals, loops, or any other statement that depends on a value collected from a managed host.
- Some of the facts gathered for a managed host might include:
 - . The host name
 - The kernel version
 - The network interfaces
 - The IP addresses
 - The version of the operating system
 - Various environment variables
 - The number of CPUs
 - The available or free memory
 - The available disk space
- Facts are a convenient way to retrieve the state of a managed host and to determine what action to take based on that state.
- Normally, every play runs the *setup module* automatically before the first task in order to gather facts.
- One way to see what facts are gathered for your managed hosts is to run a short playbook that gathers facts and uses the *debug module* to print the value of the *ansible_facts* variable.

```
- name: Fact dump
  hosts: all
  tasks:
    - name: Print all facts
      debug:
        var: ansible_facts
```

- The playbook displays the content of the **ansible_facts** variable in JSON format as a hash/dictionary of variables.

Examples of Ansible Facts

FACT	VARIABLE
Short host name	<code>ansible_facts['hostname']</code>
Fully qualified domain name	<code>ansible_facts['fqdn']</code>
Main IPv4 address (based on routing)	<code>ansible_facts['default_ipv4']['address']</code>
List of the names of all network interfaces	<code>ansible_facts['interfaces']</code>
Size of the /dev/vda1 disk partition	<code>ansible_facts['devices']['vda']['partitions']['vda1']['size']</code>
List of DNS servers	<code>ansible_facts['dns']['nameservers']</code>
Version of the currently running kernel	<code>ansible_facts['kernel']</code>



NOTE

Remember that when a variable's value is a hash/dictionary, there are two syntaxes that can be used to retrieve the value. To take two examples from the preceding table:

- `ansible_facts['default_ipv4']['address']` can also be written `ansible_facts.default_ipv4.address`
- `ansible_facts['dns']['nameservers']` can also be written `ansible_facts.dns.nameservers`

When a fact is used in a playbook, Ansible dynamically substitutes the variable name for the fact with the corresponding value:

```

---
- hosts: all
  tasks:
    - name: Prints various Ansible facts
      debug:
        msg: >
          The default IPV4 address of {{ ansible_facts.fqdn }}
          is {{ ansible_facts.default_ipv4.address }}

```

Comparison of Selected Ansible Fact Names

ANSIBLE_FACTS FORM	OLD FACT VARIABLE FORM
ansible_facts['hostname']	ansible_hostname
ansible_facts['fqdn']	ansible_fqdn
ansible_facts['default_ipv4'] ['address']	ansible_default_ipv4['address']
ansible_facts['interfaces']	ansible_interfaces
ansible_facts['devices']['vda'] ['partitions']['vda1']['size']	ansible_devices['vda'] ['partitions']['vda1']['size']
ansible_facts['dns'] ['nameservers']	ansible_dns['nameservers']
ansible_facts['kernel']	ansible_kernel

Turning off facts gathering

To disable fact gathering for a play, set the `gather_facts` keyword to **no**:

```

---
- name: This play gathers no facts automatically
  hosts: large_farm
  gather_facts: no

```

Implementing Task Control : Loops and Conditional Tasks

- Ansible supports iterating a task over a set of items using the **loop** keyword.
- A simple loop iterates a task over a list of items.
- The loop keyword is added to the task, and takes as a value the *list of items* over which the task should be iterated.
- The *loop variable item* holds the value used during each iteration.

```
vars:
  mail_services:
    - postfix
    - dovecot

tasks:
  - name: Postfix and Dovecot are running
    service:
      name: "{{ item }}"
      state: started
    loop: "{{ mail_services }}"
```

Loops over a List of Hashes or Dictionaries

- The loop list does not need to be a list of simple values. In the following example, *each item in the list is actually a hash or a dictionary*.
- Each hash or dictionary in the example has two keys, *name* and *groups*, and the value of each key in the current item loop variable can be retrieved with the *item.name* and *item.groups* variables, respectively.


```

- name: Users exist and are in the correct groups
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - name: jane
      groups: wheel
    - name: joe
      groups: root

```

Running Tasks Conditionally

- Ansible can use conditionals to execute tasks or plays when certain conditions are met.
- The **when** statement is used to run a task conditionally.
- It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.
- One of the simplest conditions that can be tested is whether a Boolean variable is true or false.
- The when statement in the following example causes the task to run only if *run_my_task* is true:

```

---
- name: Simple Boolean Task Demo
  hosts: all
  vars:
    run_my_task: true

  tasks:
    - name: httpd package is installed
      yum:
        name: httpd
        when: run_my_task

```

The following example tests whether the *my_service* variable has

a value. If it does, the value of my_service is used as the name of the package to install. If the my_service variable is not defined, then the task is skipped without an error.

```
---
- name: Test Variable is Defined Demo
  hosts: all
  vars:
    my_service: httpd

  tasks:
    - name: "{{ my_service }}" package is installed
      yum:
        name: "{{ my_service }}"
        when: my_service is defined
```

Example Conditionals

OPERATION	EXAMPLE
Equal (value is a string)	<code>ansible_machine == "x86_64"</code>
Equal (value is numeric)	<code>max_memory == 512</code>
Less than	<code>min_memory < 128</code>
Greater than	<code>min_memory > 256</code>
Less than or equal to	<code>min_memory <= 256</code>
Greater than or equal to	<code>min_memory >= 512</code>
Not equal to	<code>min_memory != 512</code>
Variable exists	<code>min_memory is defined</code>
Variable does not exist	<code>min_memory is not defined</code>

```

---
- name: Demonstrate the "in" keyword
  hosts: all
  gather_facts: yes
  vars:
    supported_distros:
      - RedHat
      - Fedora
  tasks:
    - name: Install httpd using yum, where supported
      yum:
        name: http
        state: present
      when: ansible_distribution in supported_distros

```

Implementing Handlers

1. Ansible modules are designed to be idempotent. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host unless they need to make a change to get the managed host to the desired state.
2. However, sometimes when a task does make a change to the system, a further task may need to be run. For example, a change to a service's configuration file may then require that the service be reloaded so that the changed configuration takes effect.
3. *Handlers are tasks that respond to a notification triggered by other tasks.* Tasks only notify their handlers when the task changes something on a managed host.
4. *Each handler has a globally unique name and is triggered at the end of a block of tasks in a playbook.*
5. If no task notifies the handler by name then the handler will not run. If one or more tasks notify the handler, the handler will run exactly once after all other tasks in the play have completed.
6. Because handlers are tasks, administrators can use the same modules in handlers that they would use for any other task.
7. Normally, handlers are used to reboot hosts and restart services.
8. Handlers can be considered as inactive tasks that only get triggered when explicitly invoked using a ***notify*** statement.

9. A task may call more than one handler in its notify section. Ansible treats the notify statement as an array and iterates over the handler names

```
tasks:
  - name: copy demo.example.conf configuration template❶
    template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:❷
      - restart apache❸

handlers:❹
  - name: restart apache❺
    service:❻
      name: httpd
      state: restarted
```

- ❶ The task that notifies the handler.
- ❷ The **notify** statement indicates the task needs to trigger a handler.
- ❸ The name of the handler to run.
- ❹ The **handlers** keyword indicates the start of the list of handler tasks.

Simplifying Playbooks with Roles

=====

- Ansible roles provide a way for you to make it easier to reuse Ansible code generically.
- You can package, in a standardized directory structure, all the tasks, variables, files, templates, and other resources needed to provision infrastructure or deploy applications.
- Copy that role from project to project simply by copying the directory. You can then simply call that role from a play to execute it.
- Ansible roles have the following benefits:
 - Roles group content, allowing easy sharing of code with others.
 - Roles can be written that define the essential elements of a system type: web server, database server, Git repository, or other purpose.
 - Roles make larger projects more manageable.
 - Roles can be developed in parallel by different administrators.

- In addition to writing, using, reusing, and sharing your own roles, you can get roles from other sources. Some roles are included as part of Red Hat Enterprise Linux, in the **rhel-system-roles** package.
- You can also get numerous community-supported roles from the Ansible Galaxy website. Reference : <https://galaxy.ansible.com>

Examining the Ansible Role Structure

- An Ansible role is defined by a standardized structure of subdirectories and files.
- The top-level directory defines the name of the role itself.

Ansible role subdirectories

SUBDIRECTORY	FUNCTION
defaults	The main.yml file in this directory contains the default values of role variables that can be overwritten when the role is used. These variables have low precedence and are intended to be changed and customized in plays.
files	This directory contains static files that are referenced by role tasks.
handlers	The main.yml file in this directory contains the role's handler definitions.
meta	The main.yml file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
tasks	The main.yml file in this directory contains the role's task definitions.
templates	This directory contains Jinja2 templates that are referenced by role tasks.
tests	This directory can contain an inventory and test.yml playbook that can be used to test the role.
vars	The main.yml file in this directory defines the role's variable values. Often these variables are used for internal purposes within the role. These variables have high precedence, and are not intended to be changed when used in a playbook.

Not every role will have all of these directories.

How to use Ansible Roles in Playbook

Using roles in a playbook is straightforward. The following example shows one way to call Ansible roles.

```
---
- hosts: remote.example.com
  roles:
    - role1
    - role2
```

Creating Roles

Creating and using role is a 3-step process -

1. Create a role directory structure
2. Define the role content
3. Use the role in the playbook

ansible-galaxy ⇒ performs various Role and Collection related operations

ansible-galaxy - -help

ansible-galaxy role - -help

man ansible-galaxy

Creating a Role Directory Structure

- By default, Ansible looks for roles in a subdirectory called **roles** in the directory containing your Ansible Playbook.
- If Ansible cannot find the role there, it looks at the directories specified by the Ansible configuration setting **roles_path**.
- We can create all subdirectories and files needed for a new role using standard linux command **mkdir**.
- The **ansible-galaxy** is a command-line tool which can be used to create role directory structure. **man ansible-galaxy // ansible-galaxy role --help**

To create ansible role directory structure ⇒ **ansible-galaxy role init apache**