

Introduction to Devops,GIT and SVN Configuration Management

Parameswari Ettiappan

A black and white photograph of a woman with curly hair, wearing a light-colored shirt, sitting at a desk and working on a laptop. She is looking down at the screen. The background is plain.

High performance. Delivered.



Goals

- Devops
- CI/CD Pipeline
- GIT
- SVN

Software Requirements



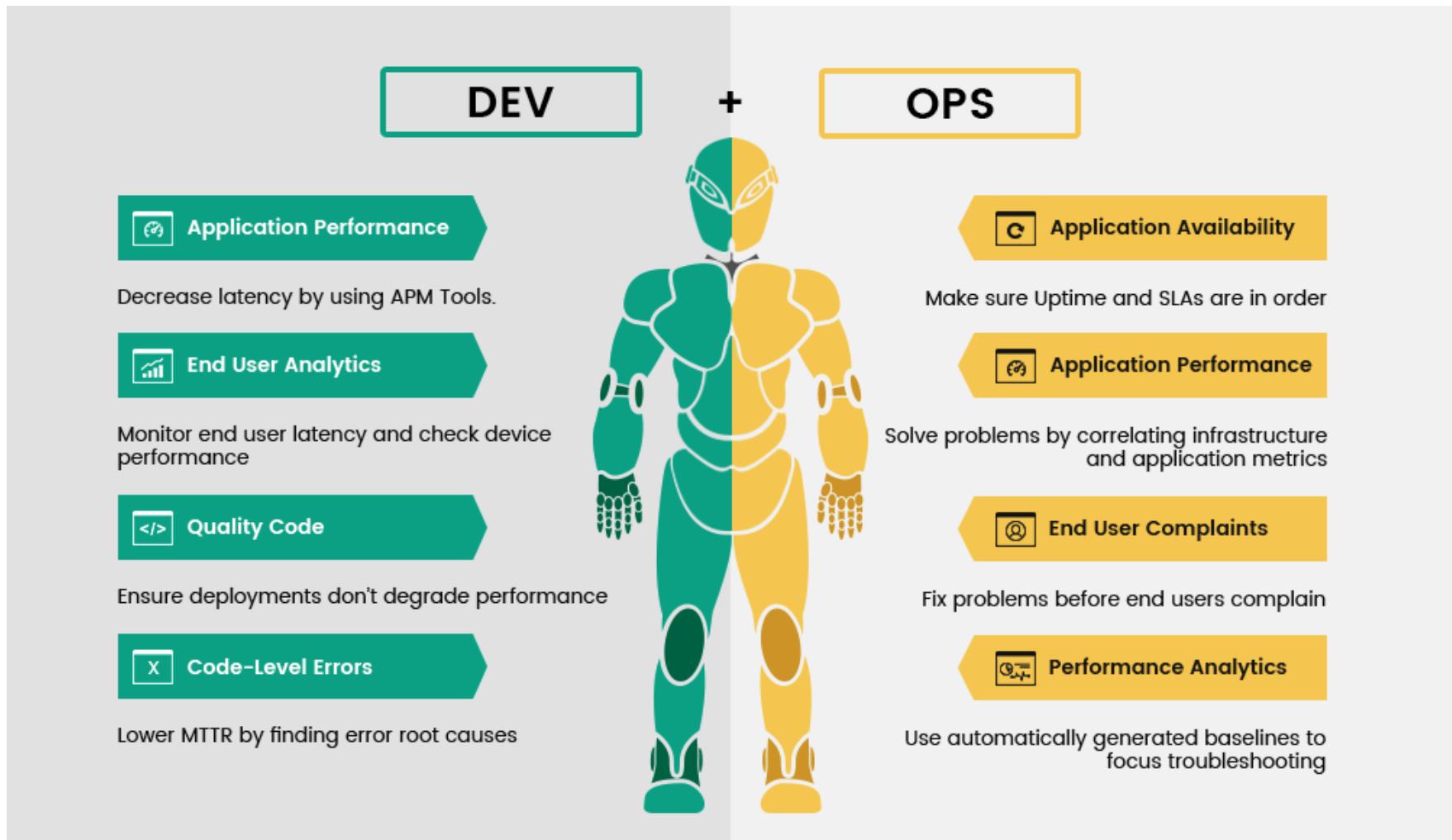
-
- Minimum java 8
 - github/gitlab
 - SVN



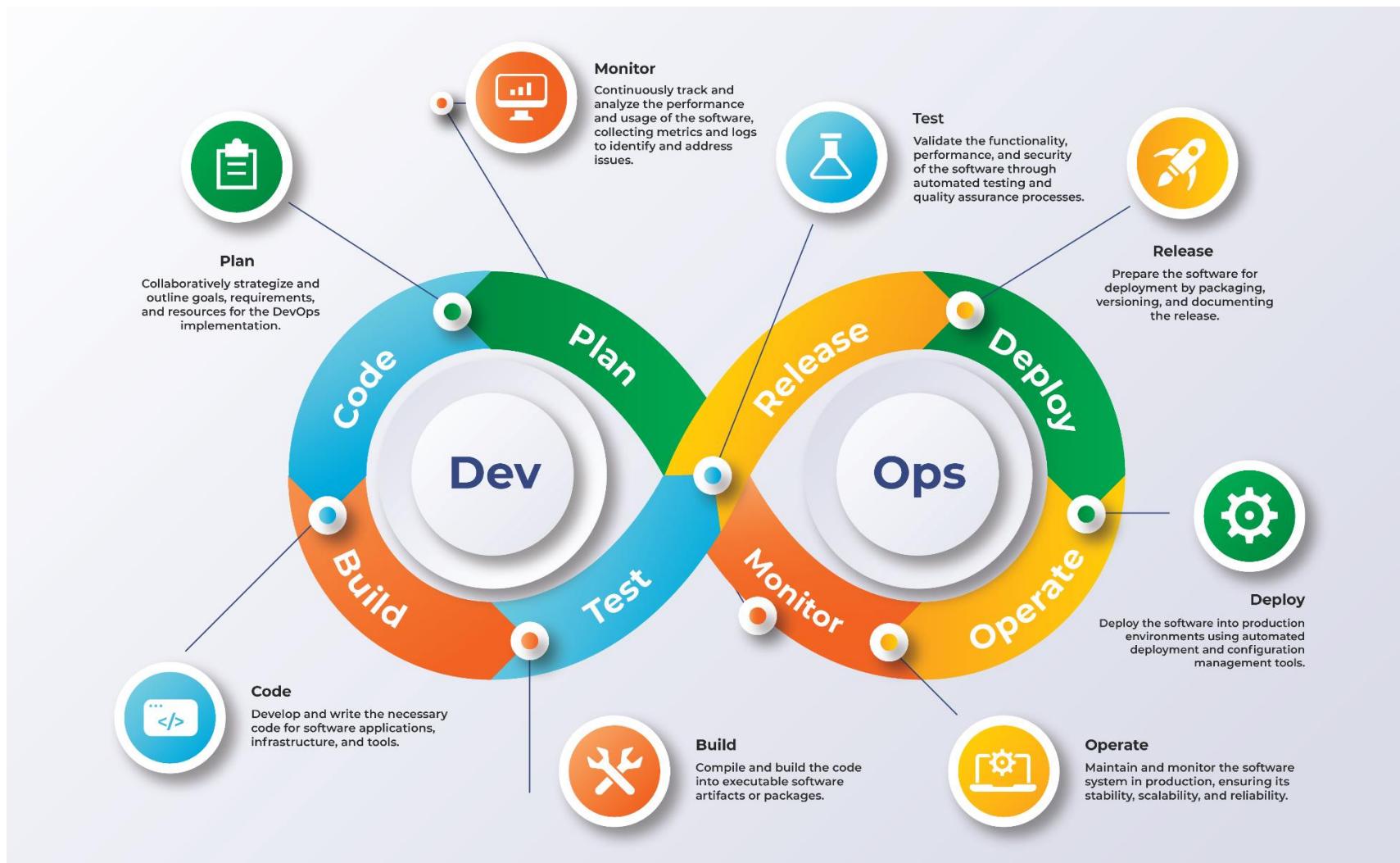
What is Devops

- DevOps is a collaboration between development and operation teams, which enables continuous delivery of applications and services to our end users.

What is Devops

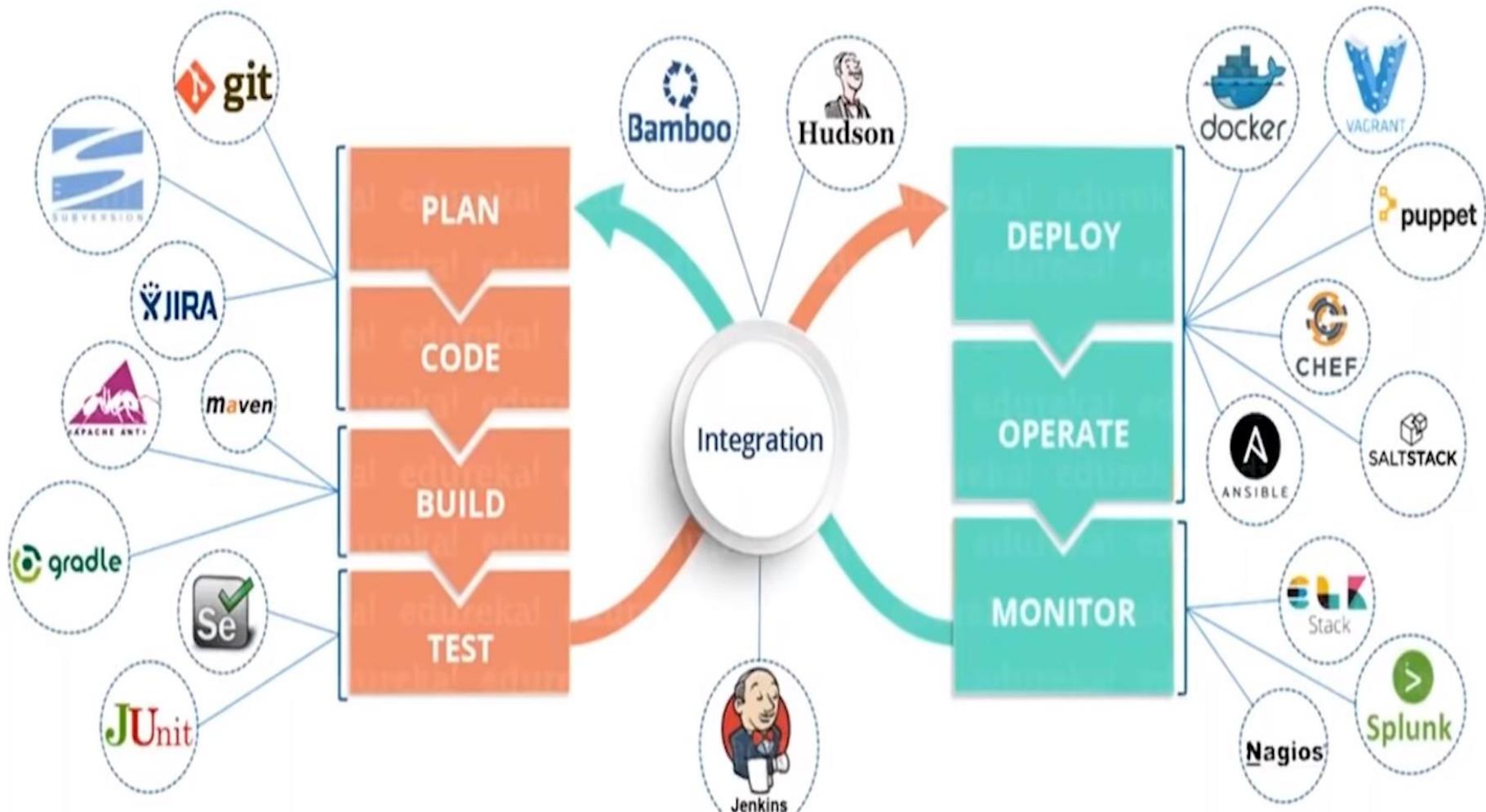


Why Devops



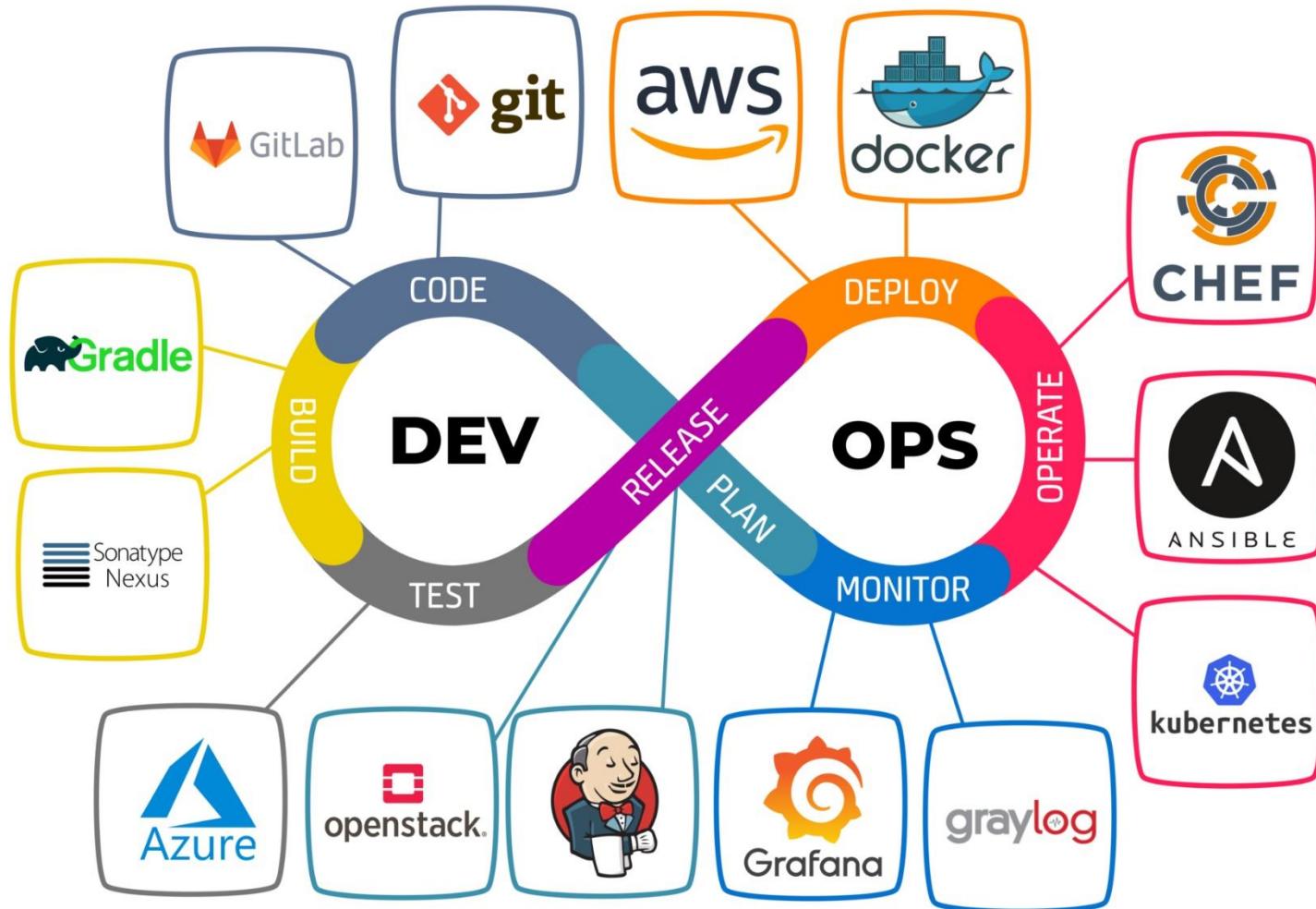


Devops Tools



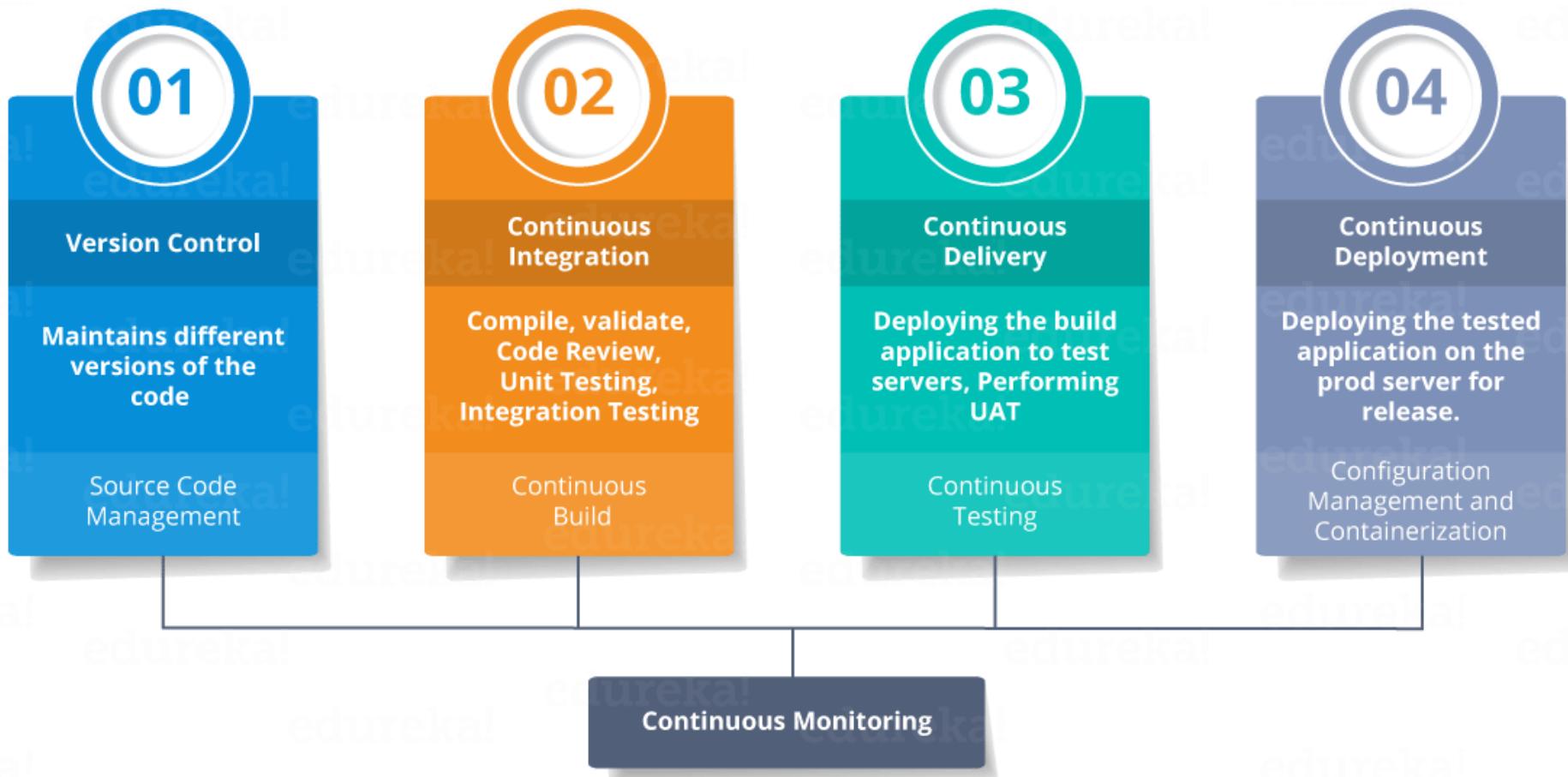


Devops Tools





CI/CD Pipeline

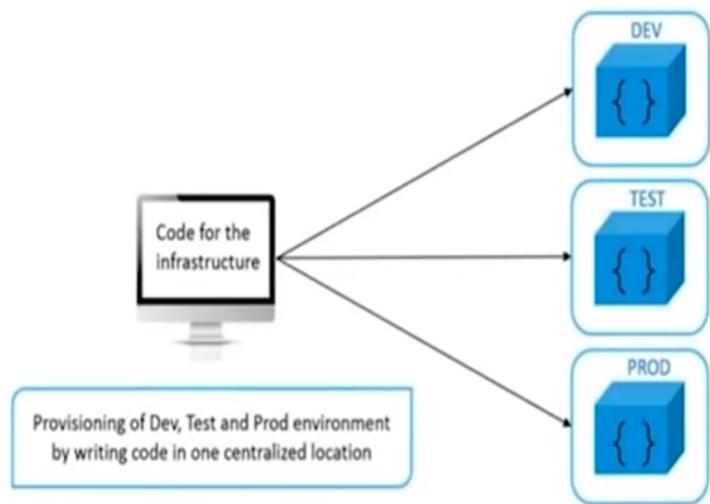


Configuration Management

- Configuration Management is the process of systematically handling changes to a system in a way that maintains its integrity over time.
- It ensures that systems are consistent, accurate, and functioning as intended, especially as they evolve with new features, updates, or maintenance.
- This practice is critical in software development, IT infrastructure, and other technical fields where complex systems are deployed and maintained.

Configuration Management

- Configuration Management is the practice of handling changes systematically so that a system maintains its integrity over time
-
-
-
-
- Configuration Management (CM) ensures that the current design and build state of the system is known, good & trusted
-
-
- It doesn't rely on the tacit knowledge of the development team



Key Components of Configuration Management

- **Identification:** Define and identify all the configuration items (CIs) in a system, including hardware, software, documentation, and processes.
This provides a baseline of what is being managed
- **Change Management:** Establish a controlled process to handle changes, ensuring that any modifications are deliberate, documented, and approved.
This minimizes unintended impacts and maintains system stability.

Key Components of Configuration Management

- **Configuration Control:** Monitor and manage changes systematically.
This ensures that only approved changes are implemented, and any unauthorized or unintended changes are flagged.
- **Configuration Status Accounting:** Maintain records of configuration items and their status, tracking changes over time.
This creates a historical record for auditing, troubleshooting, and compliance purposes.

Key Components of Configuration Management

- **Configuration Audits:** Periodically review the system's configuration to ensure that it aligns with the intended design and operational requirements.
- This helps identify any discrepancies or areas for improvement.



Tools Used in Configuration Management

- **Version Control Systems (VCS):** Tools like Git or Subversion (SVN) track changes to code and configuration files, enabling teams to manage different versions of software components.
- **Configuration Management Tools:** Tools like Ansible, Puppet, Chef, and SaltStack automate the process of deploying and maintaining consistent configurations across servers and environments.
- **Infrastructure as Code (IaC):** Tools like Terraform or CloudFormation help automate the management and provisioning of infrastructure in a repeatable and controlled manner.

Benefits of Configuration Management

- **Consistency:** Ensures systems are consistently configured across environments, reducing configuration drift.
- **Efficiency:** Automates repetitive tasks, such as provisioning servers or deploying software.
- **Control and Security:** Minimizes unauthorized changes, ensuring the integrity and security of the system.
- **Faster Troubleshooting:** Historical records of configuration changes help in identifying the root cause of issues faster.
- **Scalability:** Supports the rapid scaling of systems by providing a reliable and repeatable process for deploying configurations.

Why Version Control System?

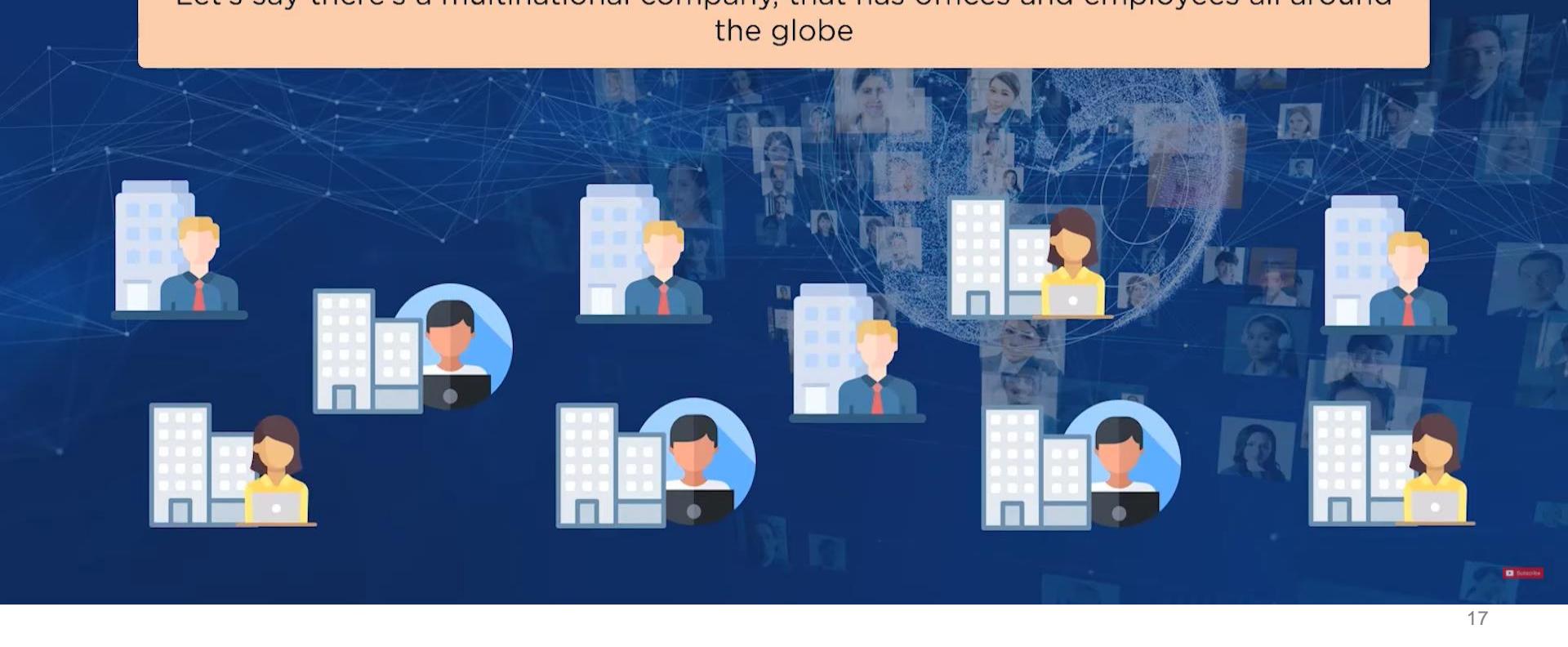
Suggested: Git Tutorial Videos [2022 Updated]



Info

Use Case

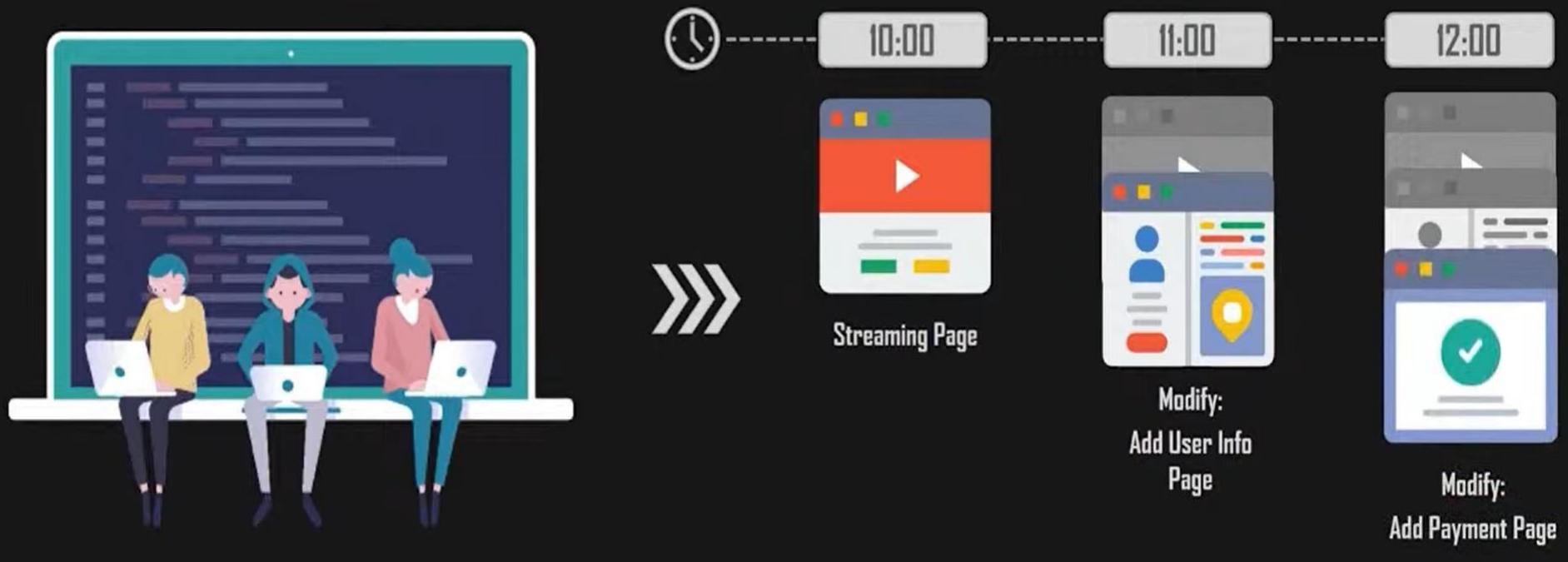
Let's say there's a multinational company, that has offices and employees all around the globe





Why Version Control System?

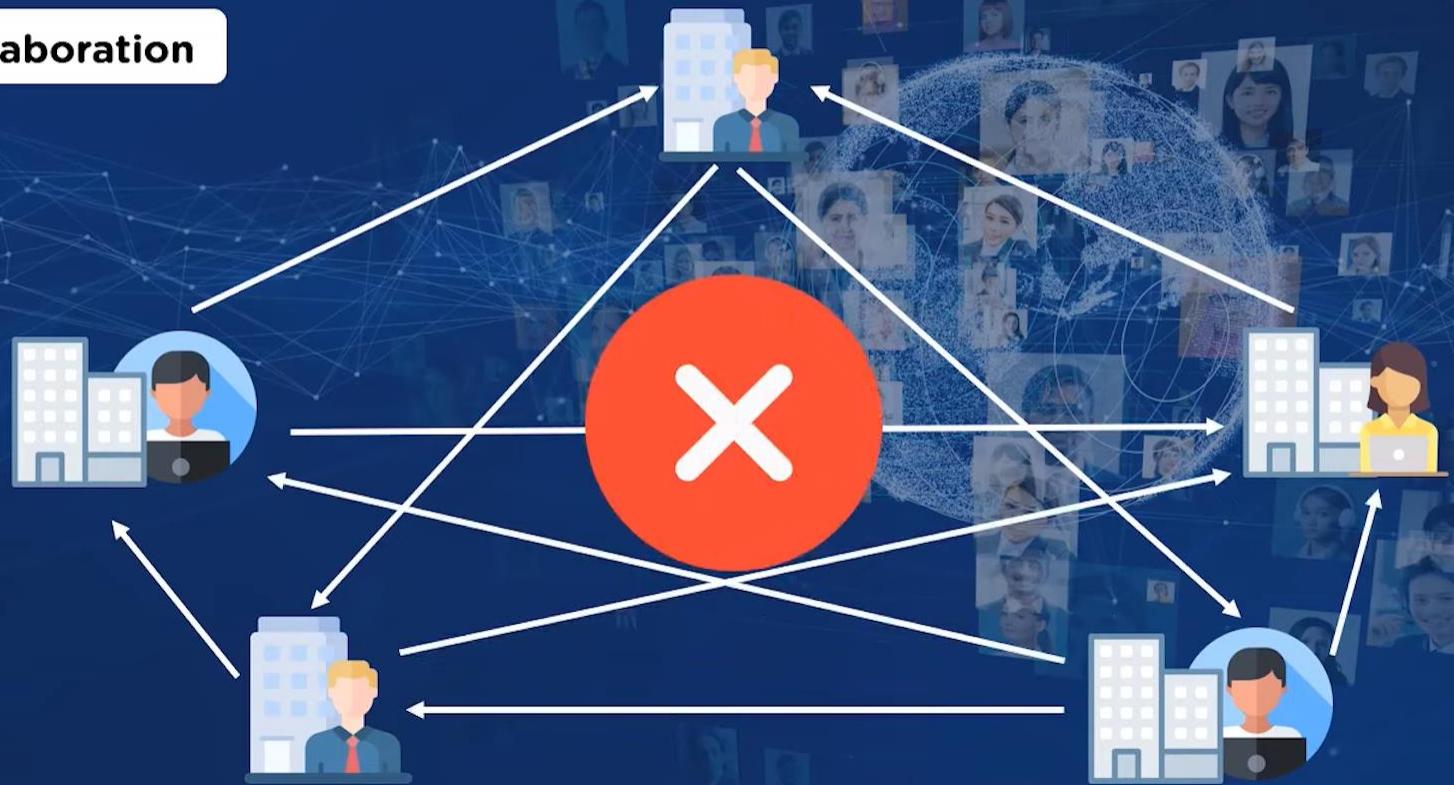
The current state of your project is saved like a snapshot with each modification.



Why Version Control System?

Problems that may arise:

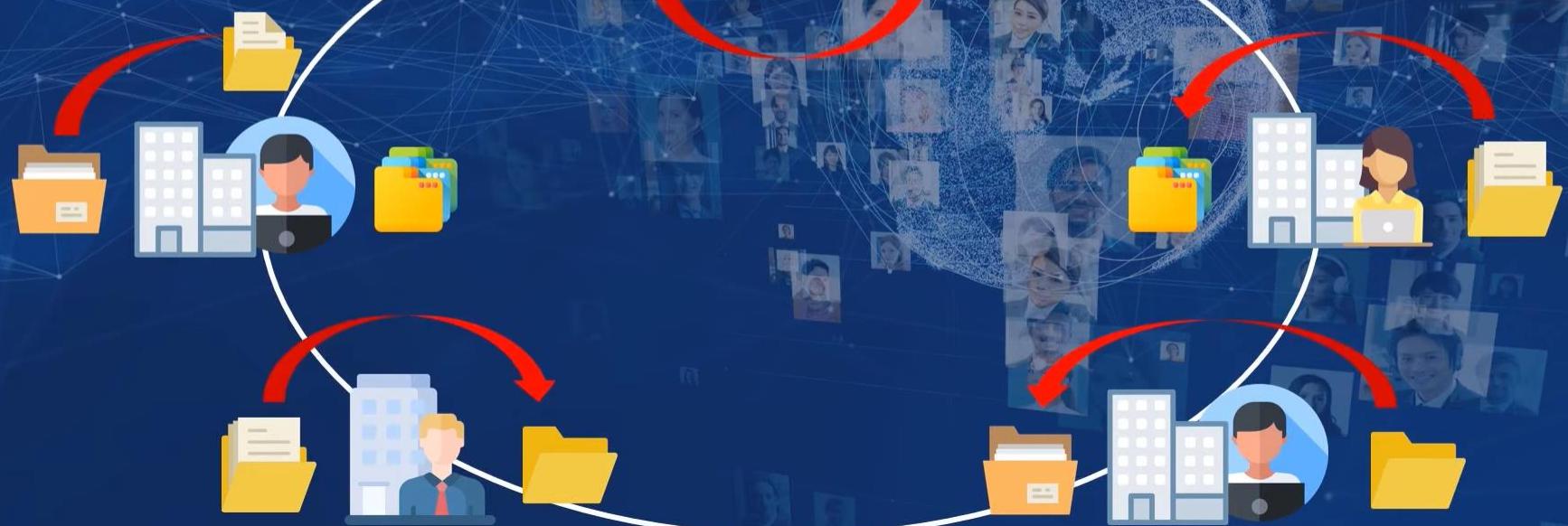
Collaboration



Why Version Control System?

Problems that may arise:

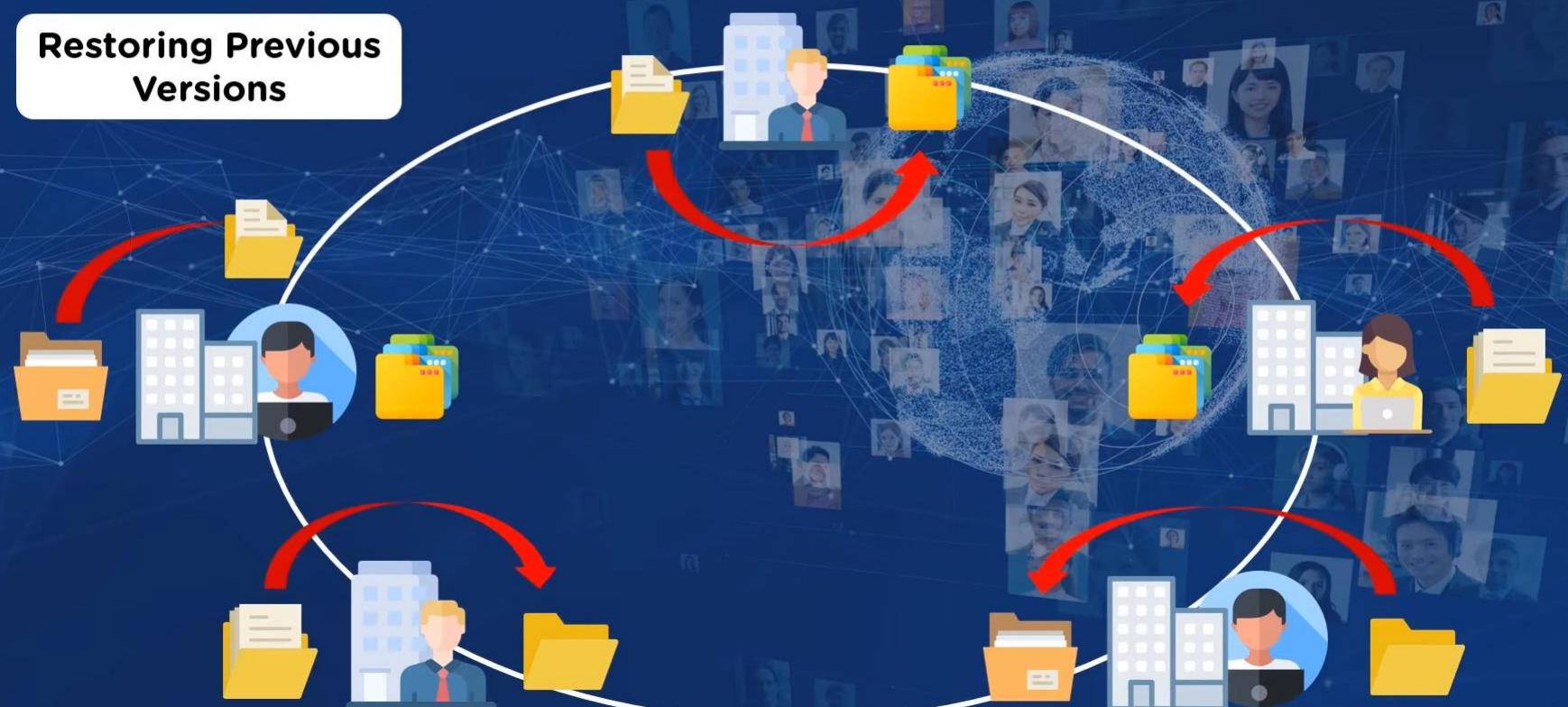
Restoring Previous Versions



Why Version Control System?

Problems that may arise:

Restoring Previous Versions



Why Version Control System?

Problems that may arise:

Figuring out
what happened



Why Version Control System?

Problems that may arise:

Backup



Why Version Control System?

What could solve all these problems?

All these problems can be solved with the help of a

“Version Control System”





Why Version Control System?

WHY VERSION CONTROL?

Collaboration 	Manage Versions 	Rollbacks 	Reduce Downtime 	Analyse Project
Shared workspace & real-time updates	All versions of code are preserved	Easy rollback from current version	Reverse faulty update & save time	Analyze and compare versions



What is Version Control System?

- A version control system allows users to keep track of the changes in software development projects and enable them to collaborate on those projects.
- Using it, the developers can work together on code and separate their tasks through branches.
- There can be several branches in a version control system, according to the number of collaborators.
- The branches maintain individuality as the code changes remain in a specified branch(s).



What is Version Control System?

- Developers can combine the code changes when required.
- Further, they can view the history of changes, go back to the previous version(s) and use/manage code in the desired fashion.



What is Repository



What is a Repository?

A **repository**(or a repo) is a **directory** or storage space where your projects can live. It can be local to a folder on your computer, or it can be a storage space on another online host (such as Github). You can keep code files, text files, image files, you name it, inside a repository.

What is Version Control System?

What is Version Control?

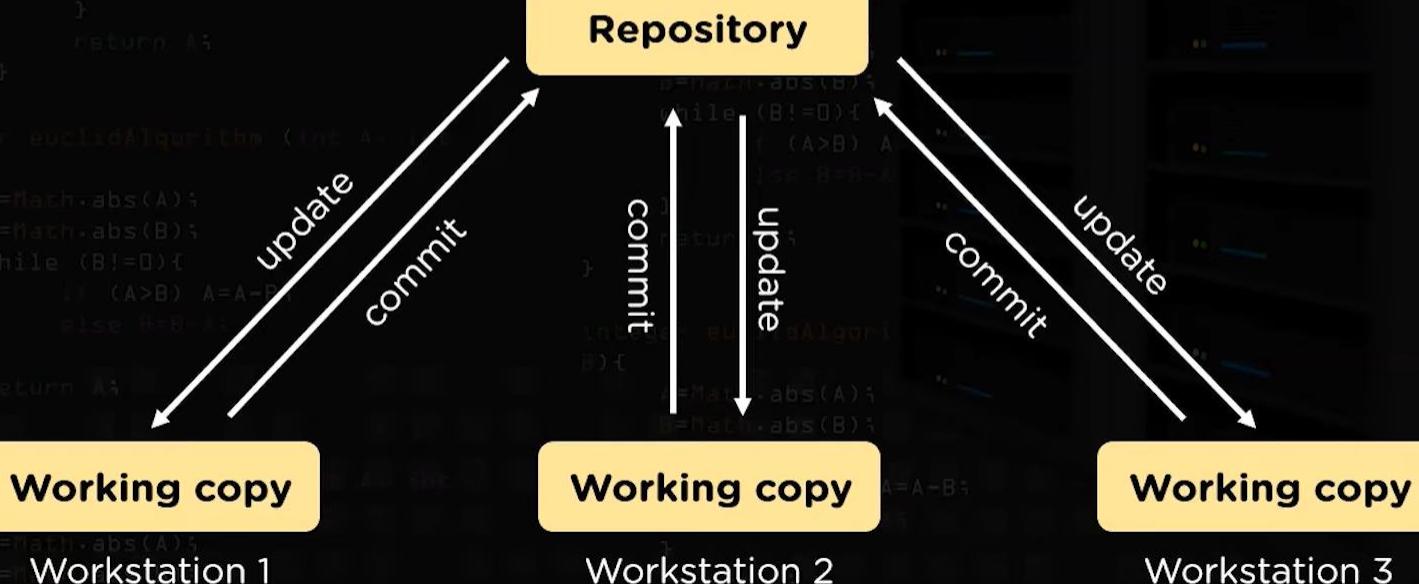
- A **Version Control** System records all the changes made to a file or set of files, so a specific version may be called later if needed
- The system makes sure that all the team members are working on the latest version of the file





What is Version Control System?

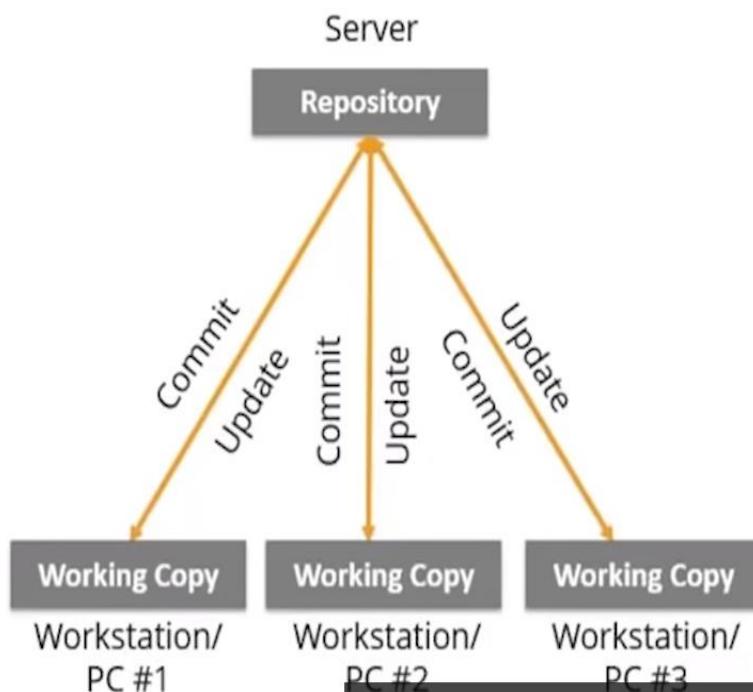
What is Version Control?



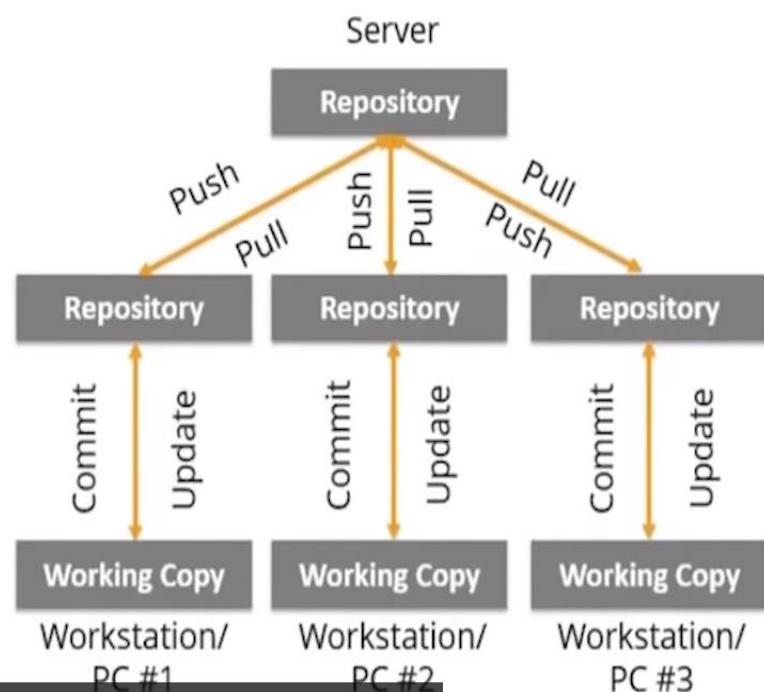
Source Code Management

The management of changes to documents, computer programs, large websites and other collection of information

Centralized Version Control System



Distributed Version Control System





Different Version Control Systems in the market

Best Version Control Systems



GitHub



Beanstalk



GitLab



AWS CodeCommit



Perforce



Different Version Control Systems in the market

Best Version Control Systems



Apache Subversion



**Team Foundation
Server**



Mercurial



Bitbucket



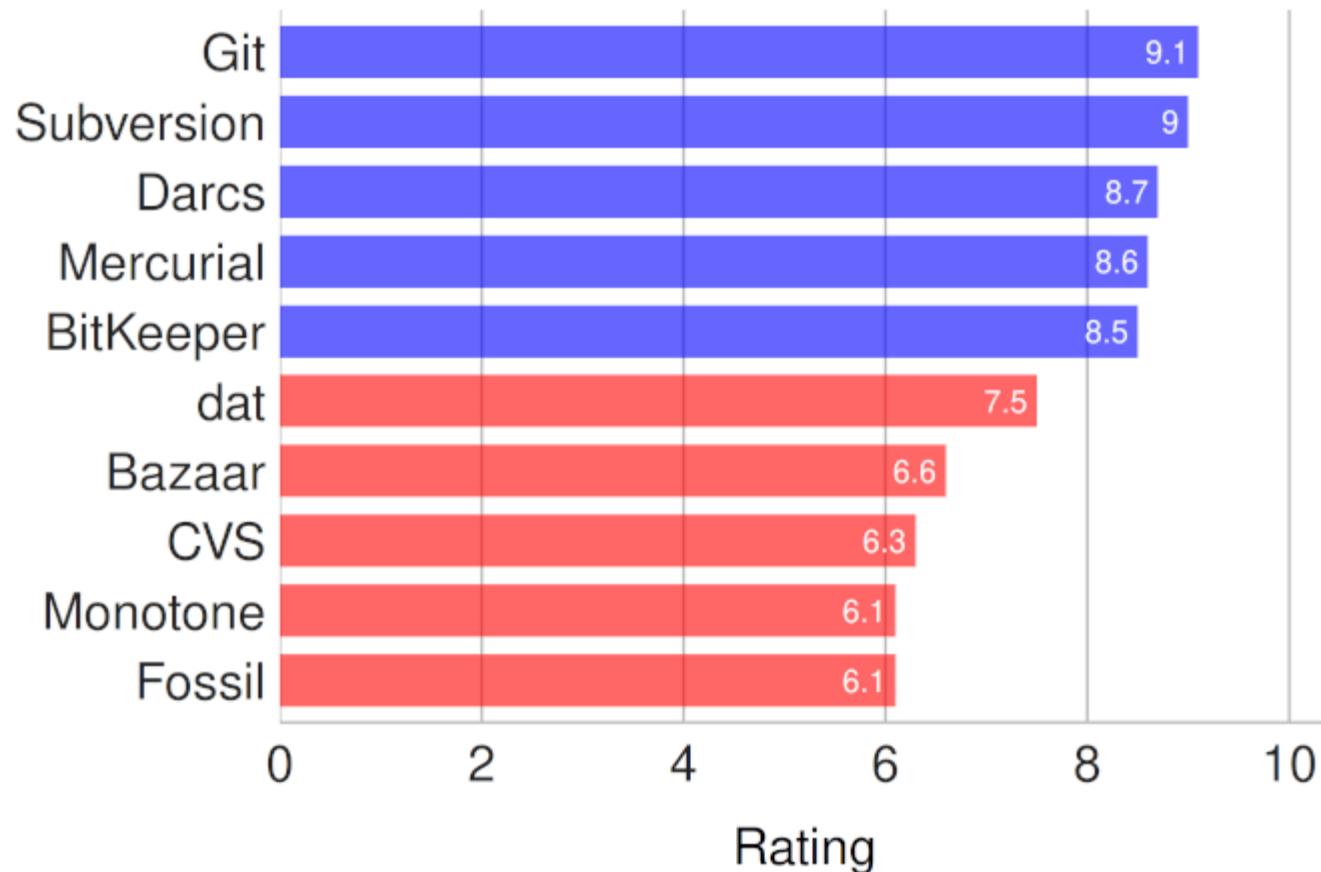
Concurrent Version Control



Different Version Control Systems in the market

Open Source Version Control

■ Recommended ■ Good





What is Git?

- Git is a distributed version control system (VCS) that helps developers manage and track changes in their source code during software development.
- It enables multiple developers to work on the same project simultaneously without overwriting each other's changes and keeps a complete history of modifications.
- Git is one of the most widely used version control systems in modern software development.
- Particularly because of its speed, reliability, and ability to handle both small and large projects.



What is Git?

- Git is an open-source distributed version control system.
- It is designed to handle minor to major projects with high speed and efficiency.
- It is developed to co-ordinate the work among the developers.
- The version control allows us to track and work together with our team members at the same workspace.
- Git is foundation of many services like GitHub and GitLab, but we can use Git without using any other Git services. Git can be used privately and publicly.



What is Git?

- Git was created by Linus Torvalds in 2005 to develop Linux Kernel.
- It is also used as an important distributed version-control tool for the DevOps.
- Git is easy to learn and has fast performance.
- It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.



Key Concepts in Git

- **Version Control System (VCS):**
 - Git falls under this category.
 - A VCS records changes to files over time, allowing us to revert to earlier versions, collaborate with others, and track progress in a project.
 - Git is a distributed VCS, meaning every developer has a full copy of the repository (a project's files and its complete history) on their local machine.



Key Concepts in Git

- **Repository:**
 - A Git repository is where all our project files and their revision history are stored.
 - There are two types of repositories:
 - Local repository:
 - Stored on our local machine.
 - Remote repository:
 - Stored on a server (e.g., GitHub, GitLab, Bitbucket) and can be shared among collaborators.



Key Concepts in Git

- Commit :
 - A commit is a snapshot of your project at a specific point in time.
 - Every time we make changes to files and want to save those changes, we create a commit.
 - Each commit has a unique identifier called a hash.



Key Concepts in Git

- **Branch :**
 - Branching allows you to diverge from the main line of development and work on changes in isolation.
 - The master or main branch is often the default, but developers can create separate branches for new features, bug fixes, or experimentation.
- **Merge:**
 - Merging is the process of integrating changes from one branch into another.
 - When a feature is completed on a branch, it is typically merged back into the main branch.



Branch vs Repository

Branch	Repository
A line of work happening in isolated way	It has complete project with all branches and history of all the files
Isolated branch changes will not be visible to other branches	The entire project and all branches and all commit visible
Lightweight Pointer to commits to that specific branch	Larger contains all commits irrespective of branches
Used for parallel development or testing or qa	Used to see all the changes happening on all the branches
Can merge branches together to integrate changes	Can push or pull repositories to from remote resources
Exists within the repository	Can exists has local or remote repository



Key Concepts in Git

- **Pull and Push:**
 - Push: Sending your local changes (commits) to the remote repository.
 - Pull: Fetching changes from the remote repository to update your local repository.
- **Clone:**
 - Cloning is creating a local copy of a remote Git repository, allowing you to start working on a project from your local machine.



Key Concepts in Git

- **Staging Area (Index):**
 - Git has a staging area, where changes to files are "staged" before being committed.
 - This allows you to selectively commit parts of the changes made to your project.
- **Checkout:**
 - Checking out means switching between different branches or commits in your repository.
 - This allows you to work on different parts of the project or inspect previous versions.



Key Concepts in Git

- **Conflict:**
 - When two people make changes to the same file (or even the same line in a file), Git may not know which changes to keep.
 - This situation is called a merge conflict, and it must be resolved manually by the developer.



What is Git?

- Projects and companies using Git include
 - Google
 - Facebook
 - Microsoft
 - Twitter
 - LinkedIn
 - Netflix
 - O'Reilly
 - PostgreSQL
 - Android
 - Linux
 - Eclipse



GIT Eco System

- We can break down the Git-based offerings into a few categories:
 - core Git
 - Git-hosting sites
 - self-hosting packages
 - ease-of-use packages
 - plugins
 - Tools that incorporate Git
 - Git libraries



Core Git

- In the core Git category, we have the basic Git executables, configuration files, and repository management tooling.
- We can install and use through the command line interface.
- In addition to the basic pieces, the distributions usually include some supporting tools such as a simple GUI (`git gui`), a history visualization tool (`gitk`).
- In some cases, an alternate interface such as a Bash shell that runs on Windows.
- The distribution for Windows is now called Git for Windows.
- Similarly, there is a ported version of Git for OS/X.
- This version can be installed directly from the git-scm.com site, or via the Homebrew package manager or built via the Mac Ports application.



GIT Installations

Debian/Ubuntu

- \$ apt-get install git
- Fedora (up to 21)
 - \$ yum install git
- Fedora (22 and beyond)
 - \$ dnf install git
- FreeBSD
 - \$ cd/usr/ports/devel/git
 - \$ make install



GIT Installations

- Gentoo
 - \$ emerge --ask --verbose dev-vcs/git
- OpenBSD
 - \$ pkg_add git
- Solaris 11 Express
 - \$ pkg install developer/versioning/git.



GIT Hosting Sites

- Git-hosting sites are websites that provide hosting services for Git repositories, both for personal and shared projects.
- Customers may be individuals, open-source collaborators, or businesses.
- Many open-source projects have their Git repositories hosted on these sites.



GIT Hosting Sites – GitHub Website

- <https://github.com>
 - GitHub is the most popular Git hosting platform, widely used for open-source projects as well as private repositories.
 - It provides a web-based interface for Git repositories and offers a suite of collaboration tools, making it ideal for developers and teams to manage and share code.



GIT Hub Key Features

- **Pull Requests:** GitHub popularized pull requests, which allow developers to propose changes, review code, and merge updates collaboratively.
- **GitHub Actions (CI/CD):** Built-in CI/CD tool for automating workflows, testing, and deployment.
- **GitHub Pages:** Free hosting for static websites directly from GitHub repositories.
- **GitHub Marketplace:** Offers integrations and tools for enhanced development workflows.
- **GitHub Discussions:** For community discussions and support within projects.
- **Community and OpenSource Projects:** GitHub has a massive open-source community, hosting millions of projects that can be forked and contributed to.



GIT Hub Key Features

- **Usage:**
 - GitHub is used by millions of developers and companies worldwide, including large open-source projects like Linux, React, and Kubernetes.
 - It's ideal for both personal and professional projects, especially if we're looking to make our work visible to the open-source community.
- **Free vs Paid:**
 - **Free:** Offers unlimited public repositories and private repositories with limited collaborators.
 - **Paid:** Includes advanced features like additional storage, team permissions, and enhanced support.



GIT Self-Hosting GitLab

- Website: <https://gitlab.com>
- GitLab is another widely used Git hosting service, particularly popular for its DevOps and CI/CD capabilities.
- Unlike GitHub, GitLab offers a comprehensive platform that covers the entire software development lifecycle (SDLC), from version control to CI/CD, deployment, and monitoring.



GitLab key features

- Integrated CI/CD Pipelines:
 - GitLab's CI/CD is built-in and fully integrated, allowing you to automate testing, building, and deployment from the repository itself.
- Issue Tracking and Project Management:
 - Advanced tools for tracking bugs, features, and managing agile workflows (e.g., milestones, kanban boards).
- Merge Requests:
 - Like GitHub's pull requests, allowing teams to collaborate on code before merging.



GitLab key features

- Container Registry:
 - Built-in container registry for storing Docker images.
- Security and Compliance:
 - Built-in security scans and monitoring for vulnerabilities.
- Self-hosted Option:
 - GitLab provides an open-source version that you can host on your own servers, which is useful for enterprises with specific security or compliance needs.



GitLab key features

- Usage:
 - GitLab is widely used by enterprises, startups, and DevOps teams that need robust CI/CD, security, and deployment features in one platform.
 - It is ideal for teams looking for full lifecycle management without needing separate tools for continuous integration, monitoring, and security.
- Free vs Paid:
 - Free: Provides public and private repositories with basic CI/CD and project management features.
 - Paid: Includes premium features like more powerful CI/CD pipelines, project management tools, and advanced security scans.



Git Hosting Sites – Bit Bucket

- Website:
- <https://bitbucket.org>
 - Bitbucket is a Git hosting service from Atlassian, designed for teams and enterprises, especially those already using other Atlassian tools like Jira and Confluence.
 - It offers deep integration with Jira for tracking issues and features alongside code repositories.



Bit Bucket Key Features

- Jira Integration:
 - Tight integration with Jira, making it easy to track issues and commits side by side.
- Bitbucket Pipelines:
 - Built-in CI/CD for automating testing and deployments.
- Branch Permissions:
 - Set branch permissions and enforce code review policies to ensure quality control.
- Pull Requests:
 - For reviewing and merging code with in-line comments and feedback.
- Free for Small Teams:
 - Offers unlimited private repositories for teams of up to 5 users.



Bit Bucket Key Features

- Usage:
 - Bitbucket is a great option for teams that are already using Atlassian products, as it integrates well with the entire Atlassian ecosystem.
 - It's often preferred by professional teams and enterprises needing more control over their workflow, especially those using Jira for project management.
- Free vs Paid:
 - Free: Free plan for teams with up to 5 users and unlimited private repositories.
 - Paid: Provides additional user capacity, advanced permissions, and integration options.



GIT Hosting Sites – Source Forge

- Website:
- <https://sourceforge.net>
 - Source Forge was one of the earliest Git hosting platforms and is focused on open-source projects.
 - Though its popularity has waned in favor of GitHub, it remains a solid platform for hosting and managing open-source projects.



GIT Hosting Sites – Source Forge

- Key Features:
- Open-Source Hosting:
 - Specializes in providing hosting for open-source projects.
- Built-in Tools:
 - Issue tracking, discussion forums, project wikis, and download management.
 - Repository
- Support:
 - Supports Git, Subversion (SVN), and Mercurial repositories.
 - Statistics and Analytics: Provides detailed download statistics for project releases.



Ease-of-Use Packages

- In the context of Git and software development, Ease-of-Use Packages typically refer to
 - tools, frameworks, or services that simplify the process of managing repositories, writing code, or automating workflows.
- These packages and tools often wrap
 - complex processes into more intuitive commands or interfaces, making them easier to use, especially for beginners or teams looking to streamline their development process.



Ease-of-Use Packages

- Git GUI Clients:
 - Graphical interfaces for Git can make working with Git easier by providing a visual representation of repositories, branches, and commits, as well as simplifying Git commands.
- GitKraken:
 - A popular Git GUI with an intuitive interface for managing repositories, branches, commits, and pull requests. It's great for both new and experienced developers.
- Sourcetree:
 - A free Git client for Windows and macOS developed by Atlassian.
 - It provides a visual interface for Git repositories and supports Git, Mercurial, and other version control systems.



Ease-of-Use Packages

- GitHub Desktop:
 - A simple Git client for managing repositories hosted on GitHub.
 - It integrates well with GitHub workflows and is a good choice for beginners.
- Tower:
 - A commercial Git GUI client that is feature-rich and focuses on improving productivity for developers.
 - It supports Git and SVN.



Ease-of-Use Packages

- Integrated Development Environments (IDEs):
- Visual Studio Code (VS Code):
 - A lightweight but powerful text editor from Microsoft with built-in Git integration.
 - It provides an intuitive Git sidebar for managing branches, commits, and conflicts.
- JetBrains IntelliJ IDEA / PyCharm:
 - These JetBrains IDEs come with seamless Git integration, allowing developers to handle version control directly within the editor.
 - It provides Git GUI-like features for branching, diffing, and merging.
- Atom:
 - A text editor developed by GitHub, Atom has built-in Git and GitHub integration, making version control a seamless part of the development workflow.



Ease-of-Use Packages

- Package Managers:
 - Package managers simplify the process of installing and managing software dependencies.
 - Many of these tools are used to handle dependencies in specific programming languages or frameworks.
- Homebrew (for macOS and Linux):
 - A package manager for installing software on macOS and Linux.
 - It simplifies the installation of tools like Git, programming languages, and other utilities.



Ease-of-Use Packages

- Chocolatey (for Windows):
 - A Windows package manager that simplifies the installation of development tools like Git, Docker, and many other utilities.
- npm (Node.js Package Manager):
 - Simplifies installing JavaScript libraries and packages for web development.
- pip (Python Package Manager):
 - Manages the installation of Python libraries and dependencies for Python development projects.



Ease-of-Use Packages

- DevOps and CI/CD Platforms:
 - Continuous Integration (CI) and Continuous Deployment (CD) platforms automate workflows for building, testing, and deploying code.
 - These platforms provide easy-to-use interfaces to set up automated pipelines.
- GitHub Actions:
 - Integrated directly into GitHub, GitHub Actions allows developers to automate workflows like running tests, building projects, and deploying applications.
 - It provides pre-built templates for common tasks, simplifying the CI/CD process.
- GitLab CI/CD:
 - Built into GitLab, it offers CI/CD pipelines with an easy-to-use configuration file.
 - It supports auto-deployments and automatic code testing.



Ease-of-Use Packages

- CircleCI:
 - A cloud-based CI/CD tool that integrates with GitHub and Bitbucket.
 - It provides intuitive configuration and automation for testing and deploying code changes.
- Travis CI:
 - A popular CI tool that integrates well with GitHub. It simplifies the process of setting up automated builds and tests for code.

Ease-of-Use Packages

- Automation and Scripting Tools:
 - These tools help in automating repetitive tasks, especially in large-scale projects, making the overall workflow smoother and faster.
 - Ansible:
 - An open-source automation tool that makes it easy to automate provisioning, configuration management, and deployment.
 - It's popular for Infrastructure as Code (IaC) practices.
 - Terraform:
 - A tool that automates the management of infrastructure using declarative configuration files.
 - It simplifies the process of managing cloud infrastructure across providers like AWS, Azure, and Google Cloud.



Ease-of-Use Packages

- Automation and Scripting Tools:
 - Docker:
 - Simplifies the process of packaging and deploying applications.
 - By containerizing applications, developers can create consistent environments that are easy to deploy across machines and platforms.
 - Prettier:
 - A code formatter that automates the process of keeping code style consistent across a project.
 - Prettier integrates with Git to automatically format code before commits.



Plugins

- Git Plugins are tools or extensions that enhance the functionality of Git by adding features, improving workflows, and simplifying tasks.
- These plugins integrate directly with Git or Git platforms (like GitHub, GitLab) and development environments (like VS Code, IntelliJ) to improve the developer experience.
- Below are some commonly used Git plugins categorized by their functionality.



Git Workflow Enhancement Plugins

- Git Flow:
 - Purpose:
 - Provides an opinionated branching model for Git that simplifies feature branching, releases, and hotfixes.
 - Features:
 - Automatically handles the creation of feature branches, release branches, and hotfixes.
 - Uses predefined branch names like develop, feature/*, release/*, and hotfix/*.
 - Commands: git flow init, git flow feature start, git flow release finish, etc.



Git Workflow Enhancement Plugins

- Hub:
- Purpose:
 - A Git wrapper that makes it easier to interact with GitHub.
- Features:
 - Simplifies common GitHub actions like creating pull requests or cloning repos.
 - Allows you to git browse, git compare, and git create without leaving your terminal.



Git Workflow Enhancement Plugins

- git-extras:
- Purpose:
 - Adds useful Git utilities that are not available by default.
- Features:
 - Adds commands like git ignore (for generating .gitignore files), git summary (to show contributions summary), and git release.
 - Simplifies complex Git workflows by adding high-level commands for day-to-day usage.



Tools That Incorporate Git

- Over the past few years, tooling has emerged that directly incorporates and uses Git as part of its model.
- One example is Gerrit, a tool designed primarily to do code reviews on changes targeted for Git remote repositories.
- At its core, Gerrit manages Git repositories and inserts itself into the Git workflow.
- It wraps Git repositories in a project structure with access controls, a code review workflow and tooling, and the ability to configure other validations and checks on the code.



GIT Libraries

- Git Libraries are libraries or APIs that allow developers to interact with Git repositories programmatically.
- These libraries provide tools for accessing Git functionality (such as commits, branches, merges, diffs, etc.) from within applications, scripts, or custom workflows.
- They are used to build tools, integrate Git functionality into applications, and automate version control operations.



GIT Libraries

- LibGit2 (C Library)
 - Purpose:
 - A low-level Git library written in C, which is widely used as the backbone of Git integrations in various applications and platforms.
 - Features:
 - Provides a comprehensive API to work with Git repositories: create, manage branches, commits, diffs, merges, etc.
 - Can be used to create custom Git tools, IDE integrations, and version control workflows.
 - Used by projects like GitHub Desktop, Microsoft Visual Studio, and many others.
 - Bindings:
 - LibGit2 has language bindings for several programming languages, making it accessible across different platforms.
 - Use Case: Great for building Git integrations in low-level applications or tools that require direct Git access.

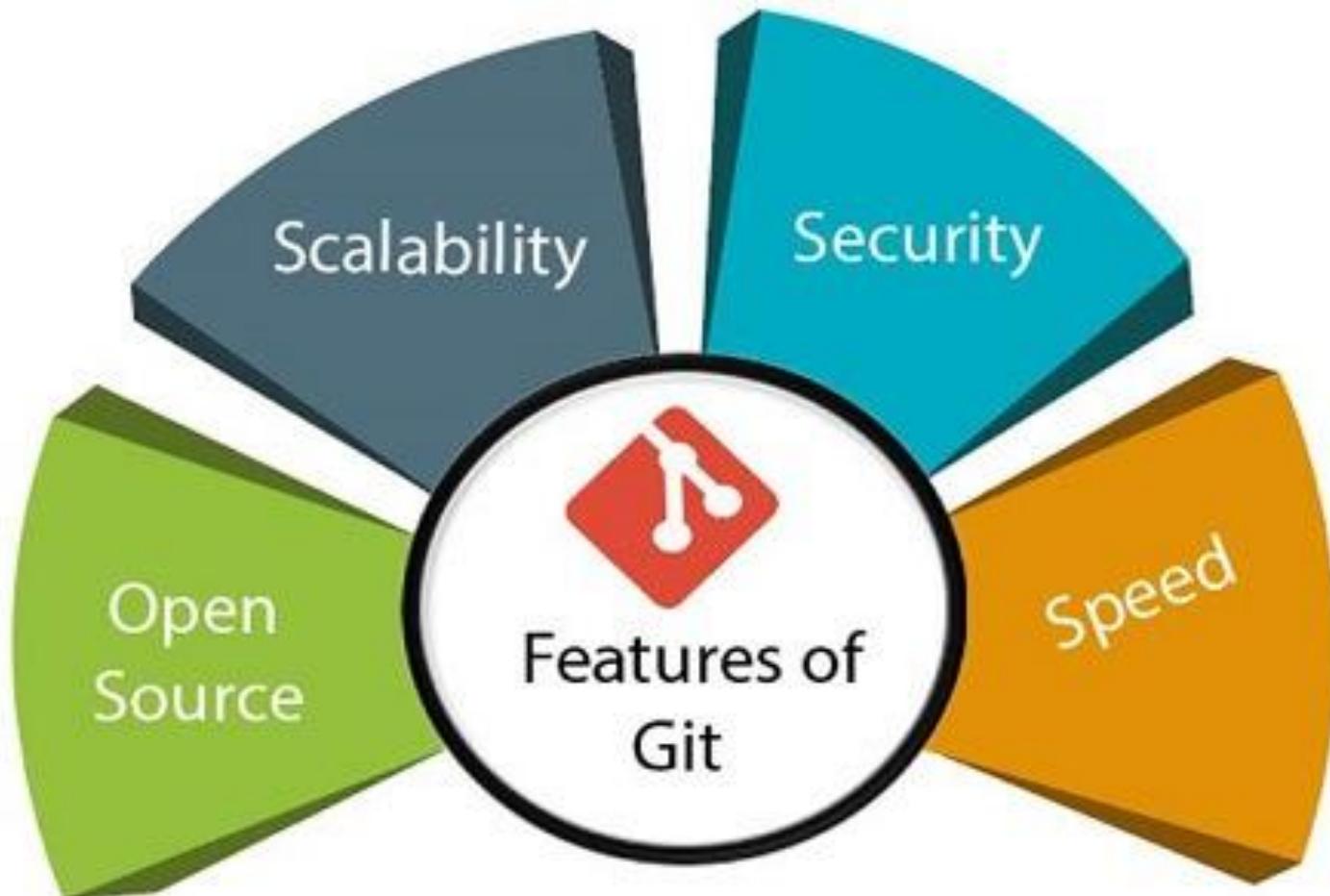


GIT Libraries

- JGit (Java Library)
- Dulwich (Python Library)
- GitPython (Python Library)
- Rugged (Ruby Library)
- SharpGit (C# Library)
- go-git (Go Library)
- nodegit (Node.js Library)
- PyGit2 (Python Library)
- NGit (Java Library)

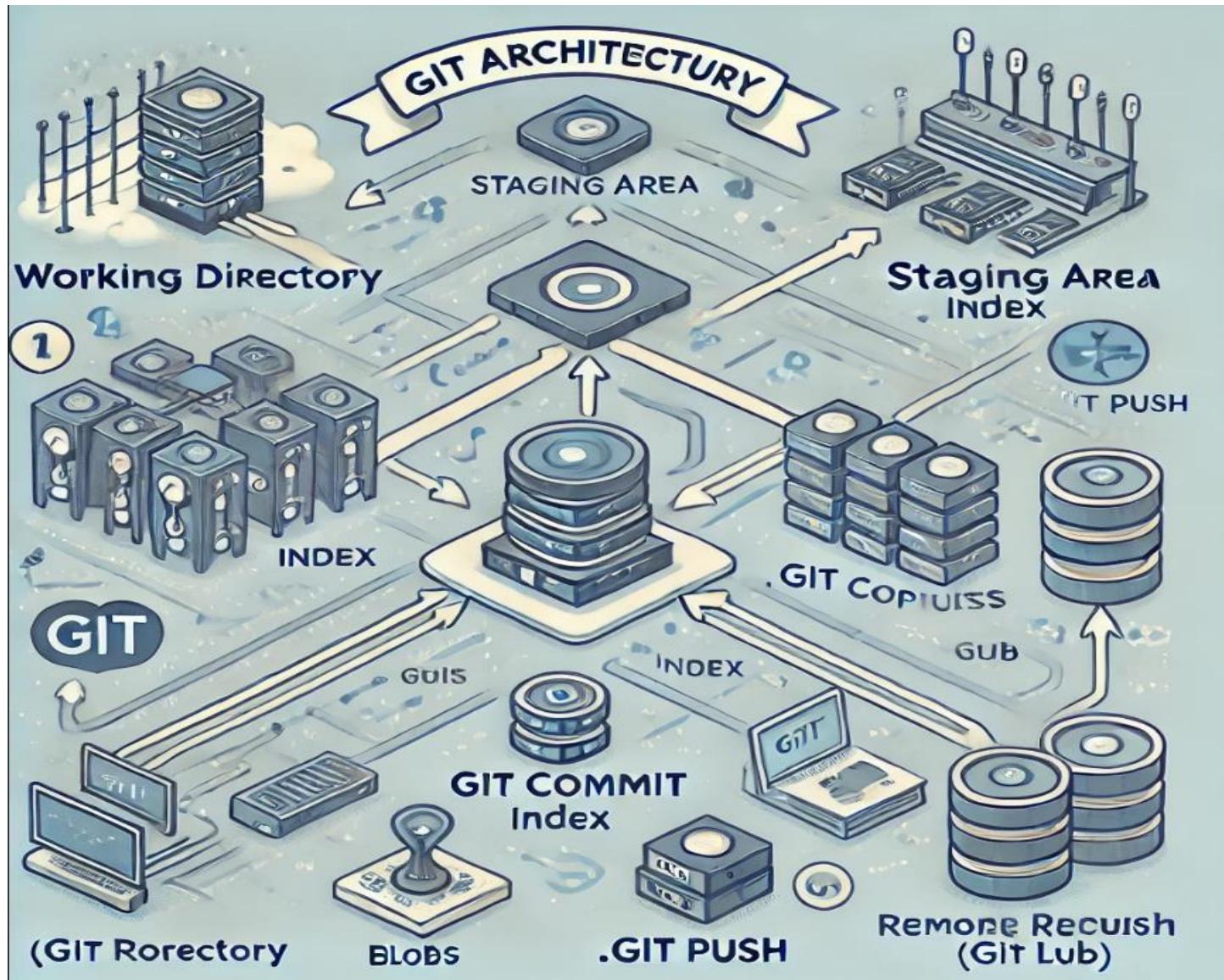


Features of Git





GIT Internal Architecture

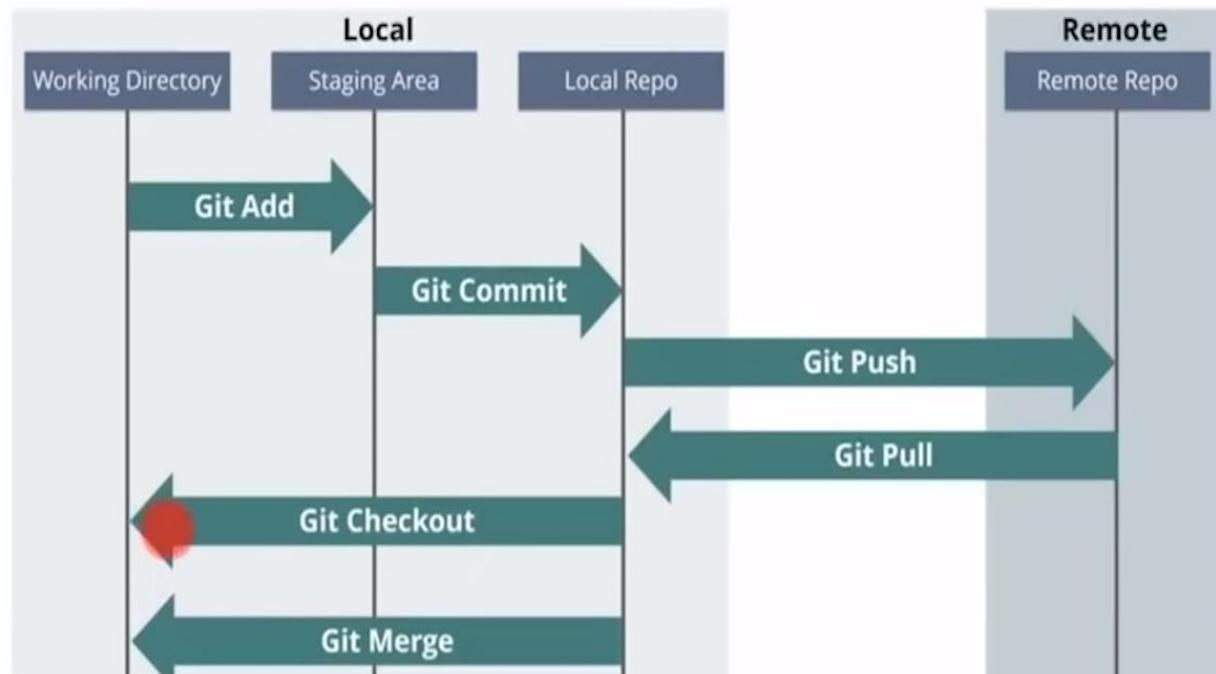




GIT Internal Architecture



Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software





GIT Internal Architecture

- Working Directory
 - Description:
 - This is the local directory on your machine where you edit files.
 - It represents the current state of the project.
 - Function:
 - Users modify, add, and delete files in the working directory.
 - These changes are not yet tracked by Git until they are staged.



GIT Internal Architecture

- Staging Area (Index)
 - Description:
 - The staging area is an intermediate space where changes are placed before being committed. It holds information about what changes will go into the next commit.
 - Function:
 - Changes are "staged" using the git add command.
 - We can choose specific files or changes to stage, allowing flexibility in what will be committed.
 - This area prepares the working directory for committing to the local repository.



GIT Internal Architecture

- Local Repository
 - Description:
 - The local repository stores all project history and metadata. It resides inside the .git directory in your project folder.
 - Function:
 - When you commit changes, they are saved into the local repository.
 - It tracks versions of the project, including every file and change ever made.
 - Key contents of the .git directory:
 - Objects: Stores all data in Git (blobs, trees, commits).
 - References: Contains branches, tags, and pointers to commits.
 - HEAD: Points to the current branch or commit being worked on.



GIT Internal Architecture

- Commit Objects
 - Description:
 - A commit is a snapshot of the project at a specific point in time, representing the state of the project.
 - Structure:
 - A commit object contains a pointer to the root tree object (which represents the file hierarchy) and references the parent commit.
 - Each commit has a unique SHA-1 hash, author information, and a commit message.
 - Function:
 - Commits record the project's history, enabling tracking, reverting, or branching from any point in the past.



GIT Internal Architecture

- Remote Repository
 - Description:
 - The remote repository is hosted on an external server (e.g., GitHub, GitLab) and is used to share your project with others.
 - Function:
 - It allows teams to collaborate by syncing changes between local repositories and the remote repository.
 - Commands like git push and git pull are used to send changes to and receive updates from the remote repository.



GIT Internal Architecture

- Branches
 - Description:
 - Branches are pointers to specific commits. They allow developers to work on different features or fixes in isolation.
 - Function:
 - The default branch is usually called main or master.
 - New branches can be created for features, allowing for independent work without disrupting the main codebase.
 - Branches can be merged back into the main branch once the feature or fix is ready.



Git config

- The git config command is used in Git to set and get configuration variables that control aspects of how Git behaves.
- These configuration settings can be at different levels, including local (specific to a repository), global (specific to a user), and system-wide (for all users on the system).



Git config

- Configuration Levels:
 - Local (--local):
 - Settings that apply only to the specific Git repository. Stored in `.git/config`.
 - Command: `git config --local <key> <value>`
 - Global (--global):
 - Settings that apply to all repositories for the current user.
 - Stored in `~/.gitconfig`.
 - Command: `git config --global <key> <value>`
 - System (--system):
 - Settings that apply system-wide, across all users and repositories.
 - Stored in `/etc/gitconfig`.
 - Command: `git config --system <key> <value>`



Common Uses

1. Setting User Information:

- Set your name:

```
bash
```

Copy code

```
git config --global user.name "Your Name"
```

- Set your email:

```
bash
```

Copy code

```
git config --global user.email "you@example.com"
```

2. Checking Configuration:

- To see all Git configurations:

```
bash
```

Copy code

```
git config --list
```



Common Uses

```
3 Dir(s) 43,027,730,432 bytes free
```

```
E:\coggitsep2024>git config --list
diff.astextplain.textconv=astextplain
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=parameswaribala@gmail.com
user.name=Parameswari Bala
difftool.p4merge.cmd=''
mergetool.p4merge.cmd=''
mergetool.p4merge.trustexitcode=true
mergetool.p4merge.path=G:\software\A08\file\p4vinst.exe
safe.directory=E:/reactbatch1ma2024training
safe.directory=F:/Local disk/NovacTraining
safe.directory=F:/Local disk/NovacTraining
safe.directory=E:/reactbatch2mar2024training
```



Common Uses

Editor

config command

Atom	<pre>~ git config --global core.editor "atom --wait"~</pre>
emacs	<pre>~ git config --global core.editor "emacs"~</pre>
nano	<pre>~ git config --global core.editor "nano -w"~</pre>
vim	<pre>~ git config --global core.editor "vim"~</pre>
Sublime Text (Mac)	<pre>~ git config --global core.editor "subl -n -w"~</pre>
Sublime Text (Win, 32-bit install)	<pre>~ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublimerедактор.exe' -w"~</pre>
Sublime Text (Win, 64-bit install)	<pre>~ git config --global core.editor "'c:/program files/sublime text 3/sublimerедактор.exe' -w"~</pre>
Textmate	<pre>~ git config --global core.editor "mate -w"~</pre>



Common Uses

3. Changing Editor:

- Set a default editor (e.g., to use VS Code):

bash

Copy code

```
git config --global core.editor "code --wait"
```

4. Changing Default Branch Name:

- Change the default branch name (e.g., from `master` to `main`):

bash

Copy code

```
git config --global init.defaultBranch main
```

5. Setting Merge and Diff Tools:

- Set a default tool for resolving merge conflicts (e.g., `vimdiff`):

bash

Copy code

```
git config --global merge.tool vimdiff
```



Git Config Password

- Git itself doesn't store passwords directly.
- But it can be configured to handle authentication through various methods like credential helpers, which can cache or store your credentials (username and password or access token).
- Since most services like GitHub, GitLab, and Bitbucket have moved away from using passwords in favor of Personal Access Tokens (PAT).
- We will generally use a PAT instead of a password for authentication.



Git Config Password

- Git itself doesn't store passwords directly.
- But it can be configured to handle authentication through various methods like credential helpers, which can cache or store your credentials (username and password or access token).
- Since most services like GitHub, GitLab, and Bitbucket have moved away from using passwords in favor of Personal Access Tokens (PAT).
- We will generally use a PAT instead of a password for authentication.



Git Config Password

1. Using Git Credential Helper to Store Credentials

To store your username and password (or token) permanently, you can use Git's credential helper.

Store credentials permanently:

bash

Copy code

```
git config --global credential.helper store
```

This will save your credentials in a plain-text file (`~/.git-credentials` on Linux/Mac or

`%USERPROFILE%\git-credentials` on Windows).

1. When you push to the repository for the first time, Git will prompt you for your username and password (or token).
2. After entering your credentials, they will be saved in the `.git-credentials` file, and you won't be prompted again.



Git Config Password

2. Using a Personal Access Token (PAT) for GitHub, GitLab, etc.

If you're using GitHub, GitLab, or Bitbucket, you'll need to generate a Personal Access Token (PAT) from your account and use it in place of a password. Here's how to use it:

- 1. Generate a PAT:** Go to your Git service provider (e.g., GitHub > Settings > Developer Settings > Personal Access Tokens) and generate a token with the necessary scopes.
- 2. When prompted for a password:** Use the generated token as your "password" when Git asks for credentials during a push, pull, or clone operation.

You can then store it permanently using the credential helper:

bash

Copy code

```
git config --global credential.helper store
```

The token will be stored in plain text as shown earlier in the `.git-credentials` file.



Git Config Password

3. Using Git Credential Manager for Secure Storage (Recommended)

To securely manage credentials, you can use Git Credential Manager (GCM), which integrates with platform-native credential stores like the Windows Credential Manager, macOS Keychain, or Linux Secret Service.

Install Git Credential Manager (GCM):

If you're using a recent version of Git, `manager-core` is usually already installed. To set it up:

bash

Copy code

```
git config --global credential.helper manager-core
```

Use the credential manager:

When you first push or pull, Git Credential Manager will prompt you for your username and password or token, and it will securely store them using the native credential storage of your operating system.



Git Config Password

4. Remove Cached Credentials

If you ever need to remove saved credentials, you can unset the credential helper:

bash

Copy code

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

Or delete the `.git-credentials` file where the username and password/token were stored.



Git Config Password

3. If Using `manager-core` or `manager` Credential Helper (Stored in Secure Store)

When using `git credential.helper manager-core` or `manager`, credentials (including PATs) are stored in platform-native credential storage like:

- Windows: Windows Credential Manager
- macOS: macOS Keychain
- Linux: Secret Service or Gnome Keyring

To remove the stored credentials:

- On Windows:
 1. Open the **Credential Manager** (Search for "Credential Manager" in the Control Panel).
 2. Navigate to **Windows Credentials**.
 3. Find the entry for your Git credentials (look for entries associated with `git:` or `github.com`), and then click **Remove**.



Git Config Password



Web Credentials

[Back up Credentials](#) [Restore Credentials](#)

Windows Credentials

[TERMSRV/caitlabs.net](#)

[Add a Windows credential](#)

Modified: 05-09-2024 ▾

Certificate-Based Credentials

No certificates.

[Add a certificate-based credential](#)

Generic Credentials

[MongoDB Compass/Connections/289a1cb8-2004-4b2...](#)

Modified: 08-07-2024 ▾

[pgAdmin4](#)

Modified: 16-07-2024 ▾

[pgAdmin4-desktop-user-pgadmin4@pgadmin.org@p...](#)

Modified: 16-07-2024 ▾

[pgAdmin4-PostgreSQL 14-1@pgAdmin4](#)

Modified: 24-05-2024 ▾

[teamsly/teams](#)

Modified: 26-12-2023 ▾

[teamsKey/teams](#)

Modified: 26-12-2023 ▾

[git:https://bitbucket.org](#)

Modified: 15-03-2024 ▾

[git:https://github.com](#)

Modified: 17-01-2024 ▾

Internet or network address: [git:https://github.com](#)

User name: eswaribala

Password: *****

Persistence: Local computer

[Edit](#) [Remove](#)



Git Config Password

```
Microsoft Windows [Version 10.0.22631.4249]  
(c) Microsoft Corporation. All rights reserved.
```

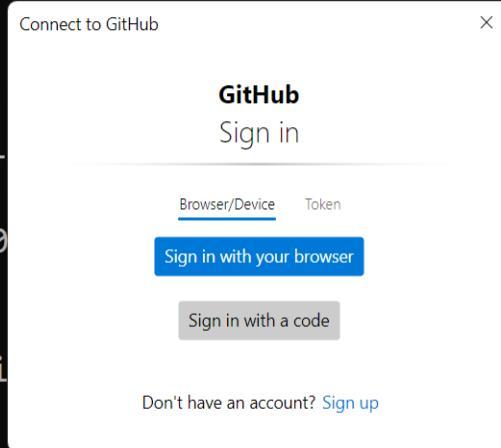
```
C:\Windows\System32>cd E:\coggitsep2024\user1
```

```
C:\Windows\System32>e:
```

```
E:\coggitsep2024\user1>git add .
```

```
E:\coggitsep2024\user1>git commit -  
[master 9b32756] updated  
 1 file changed, 0 insertions(+), 0  
 create mode 100644 test5.txt
```

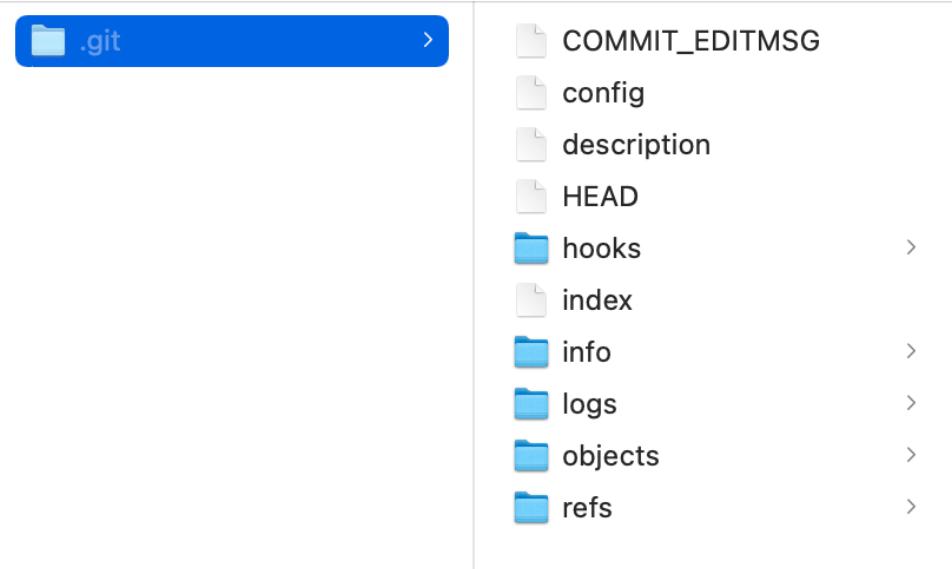
```
E:\coggitsep2024\user1>git push ori
```





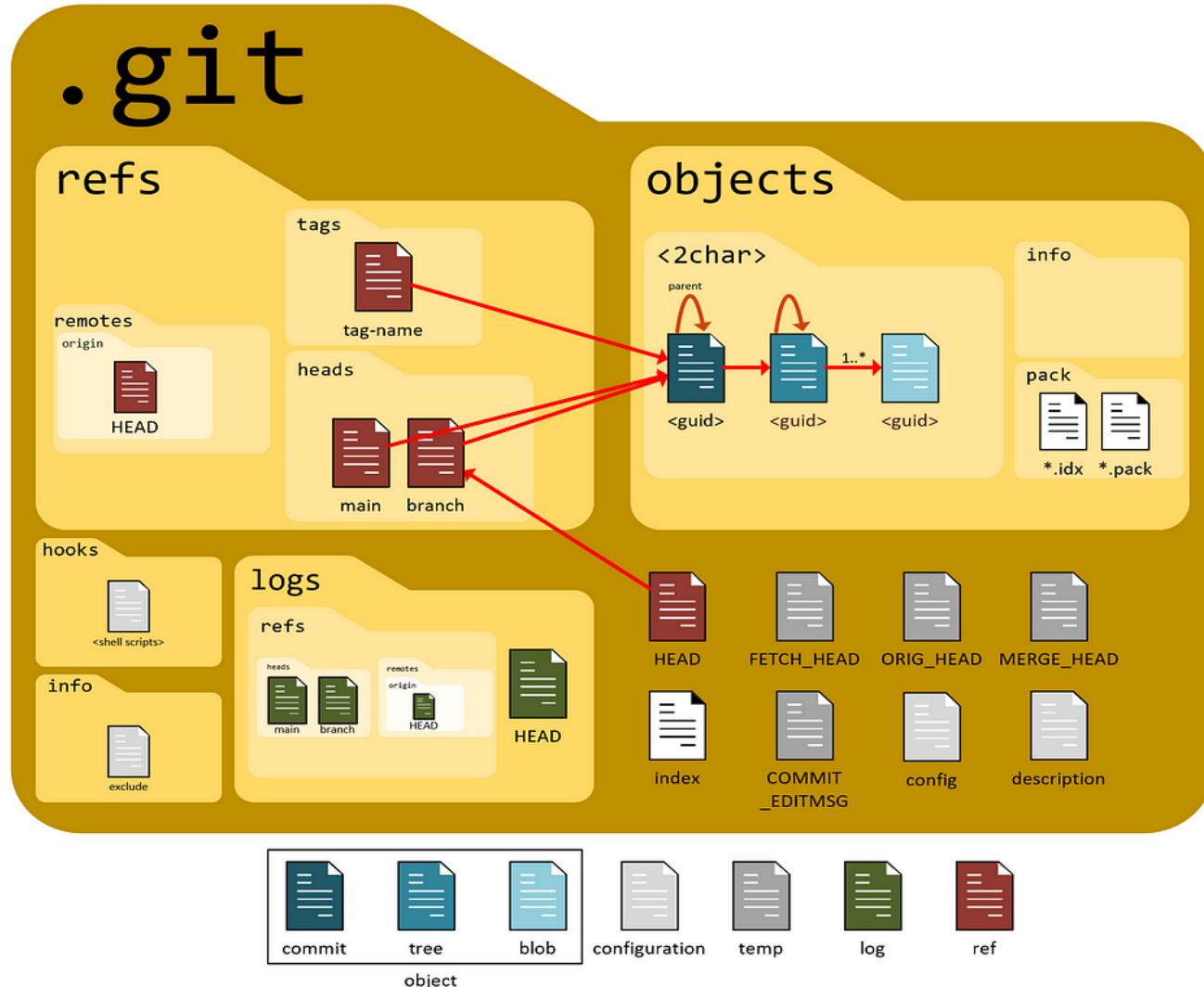
.git File

- The .git directory is a hidden folder that Git creates in the root of a project when you initialize a new repository (**git init**).
- It contains all the necessary metadata, history, and configuration for the Git version control system to manage your project.





.git File





.git File

- hooks:
 - Git hooks are the scripts that are executed before or after the commit, push, or receive commands.
 - Git hooks are a built-in feature that comes with the standard Git creation, it doesn't need to be downloaded or installed separately.
- Local Hooks:
 - pre-commit: The pre-commit hook is run first, before you even type in a commit message.
 - It could be used to make sure tests run, or to examine whatever you need to inspect in the code



.git File

- **prepare-commit-msg:**
 - The prepare-commit-msg hook is run before the commit message editor is fired up but after the default message is created.
 - It's good for commits where the default message is auto-generated.
- **commit-msg:**
 - The commit-msg hook is much like the prepare-commit-msg hook, but it's called after the user enters a commit message.
 - This is an appropriate place to warn developers that their message doesn't adhere to your team's standards.
- **post-commit:**
 - After the entire commit process is completed, the post-commit hook runs.
 - Generally, this script is used for notification or something similar.



.git File

Administrator: Command Prompt

Volume Serial Number is 3AFE-484F

Directory of E:\coggitsep2024\user1\coggitsep2024\.git\hooks

```
30-09-2024  11:54    <DIR>          .
30-09-2024  11:54            478  applypatch-msg.sample
30-09-2024  11:54            896  commit-msg.sample
30-09-2024  11:54          4,726  fsmonitor-watchman.sample
30-09-2024  11:54            189  post-update.sample
30-09-2024  11:54            424  pre-applypatch.sample
30-09-2024  11:54          1,643  pre-commit.sample
30-09-2024  11:54            416  pre-merge-commit.sample
30-09-2024  11:54            1,374  pre-push.sample
30-09-2024  11:54            4,898  pre-rebase.sample
30-09-2024  11:54            544  pre-receive.sample
30-09-2024  11:54          1,492  prepare-commit-msg.sample
30-09-2024  11:54            2,783  push-to-checkout.sample
30-09-2024  11:54          2,308  sendemail-validate.sample
30-09-2024  11:54            3,650  update.sample
14 File(s)           25,821 bytes
1 Dir(s)   43,027,730,432 bytes free
```

E:\coggitsep2024\user1\coggitsep2024\.git\hooks>



.git File

- Server-Side Hooks:
 - pre-receive: The pre-receive hook is executed every time somebody uses git push to push commits to the repository.
 - It should always reside in the remote repository that is the destination of the push, not in the originating repository.
 - If we don't like who is doing the pushing, how the commit message is formatted, or the changes contained in the commit, you can simply reject it.



.git File

- update:
 - The update script is very similar to the pre-receive script, except that it's run once for each branch the pusher is trying to update.
 - If the pusher is trying to push to multiple branches, pre-receive runs only once, whereas update runs once per branch they're pushing to.
- post-receive:
 - The post-receive hook gets called after a successful push operation, making it a good place to perform notifications.



Source Tree Git commit options

The screenshot shows the SourceTree application interface. On the left is a sidebar with various project management sections: WORKSPACE, File status, History, Search, BRANCHES (with master and redesign-navbar selected), TAGS, REMOTES (with git-training selected), STASHES, SUBMODULES, and SUBTREES. The main area displays a diff view for the file `index.html`. A specific hunk of code is highlighted, showing changes made to the `<header>` section. The commit message at the bottom reads: "Change Who I Am section to About". A context menu is open on the right, with the "Amend last commit" option highlighted. A red wavy line points from the "Amend last commit" option to the commit message input field. At the bottom, there are buttons for "Push changes immediately to git-training/redesign-navbar", "Cancel", and "Commit".

Commit Pull Push Fetch Branch Merge Stash

Pending files, sorted by path: index.html

File status

History Search

BRANCHES

master

redesign-navbar

TAGS

REMOTES

git-training

master

redesign-navbar

STASHES

SUBMODULES

SUBTREES

Lucas Bassetti <lucasbr.dafonseca@gmail.com>

Change Who I Am section to About

Amend last commit

Bypass commit hooks

Sign commit

Sign off

Create pull request

Push changes immediately to git-training/redesign-navbar

Cancel Commit



Understanding Git commit hooks

- Git hooks are in the hook's directory inside the .git directory of your repository.
- Commit hooks, specifically, include pre-commit, prepare-commit-msg, commit-msg, and post-commit hooks.
- pre-commit: Runs before you even type in a commit message. It's used to inspect the snapshot that's about to be committed.
- commit-msg: Invoked after the commit message is provided but before the commit is completed, to inspect and modify the commit message.
- post-commit: Executes after the commit is finalized.



Why skip commit hooks?

- Speed up small changes: Sometimes, especially when making trivial changes, running commit hooks can be seen as an unnecessary delay.
- Bulk changes: When committing many files, perhaps during a bulk refactor or formatting operation, hooks might repetitively perform unnecessary checks.
- Testing and debugging: At times, you might need to test the behavior of your repository without the interference of hooks.
- Overriding faulty or outdated hooks: In some cases, hooks might be out of date or malfunctioning, and you need to bypass them temporarily until they can be updated or fixed.



Why skip commit hooks?

Example 1: Committing a trivial change

Suppose you've updated a README file or made a small comment change in your code. You know it doesn't need to go through rigorous checks, and you want to commit it quickly:

TERMINAL

```
git add README.md  
git commit -m "Update README with new contact info" --no-verify
```

Example 2: Committing after a bulk formatting operation

Let's say you've just reformatted your entire project using a code formatter. Running code quality checks on all files would be redundant since the formatter already aligns the code to your standards:

TERMINAL

```
git add .  
git commit -m "Reformat codebase using XYZ formatter" --no-verify
```



Git Commit Sign Off

- Commit signoffs enable users to affirm that a commit complies with the rules and licensing governing a repository.
- We can enable compulsory commit signoffs on individual repositories for users committing through GitHub.com's web interface, making signing off on a commit a seamless part of the commit process.
- Once compulsory commit signoffs are enabled for a repository, every commit made to that repository through GitHub.com's web interface will automatically be signed off on by the commit author.



Git Commit Sign Off

Creating your signoff

Git has a `-s | --signoff` command-line option to append this automatically to your commit message:

```
git commit --signoff --message 'This is my commit message'
```



```
git commit -s -m "This is my commit message"
```



This will use your default git configuration which is found in `.git/config` and usually, it is the `username systemaddress` of the machine which you are using.

To change this, you can use the following commands (Note these only change the current `repo` settings, you will need to add `--global` for these commands to change the installation default).

Your name:

```
git config user.name "FIRST_NAME LAST_NAME"
```



Your email:

```
git config user.email "MY_NAME@example.com"
```





.git File

- Info:
 - It contains an exclude file.
 - This file is used to extract certain patterns from the code that we don't want Git to read or execute.
 - This file is local and private to us.
 - Other developers cloning the project cannot see this file.
 - If there is something that should be ignored by all developers in the project, it should be written to the **.gitignore file**.



.git File

- Objects:
 - It is the most important part of the .git file structure.
 - Every action we take, and every object or files we create are saved here in the object folder.
 - Git saves the objects we created with hash values.
 - Folders are also created according to this.
 - When a git folder is created, the object folder is empty.
 - When the commit operation is complete, the object folders are created.
 - So, why use a custom hash instead of an object name?
Because hashing method is generally used for security purposes.
 - But it is not used for this purpose in Git.
 - The main reason Git uses this method is to avoid conflicts.
 - This way, it can provide millions of files with the same content without conflicts.



.git File

- Refs(Reference):
- Refs or references are the pointers to commits.
- Let's go to the refs file in .git folder. There are three basic folders here.
 - 1. Heads(Branches): This file contains the hash values of the branches.
 - 2. Remotes: if we are working in a remote repo, Git stores the Hash values of the last reference(s) we pushed to that branch for each branch.
 - 3. Tags



.git File

```
E:\coggitsep2024\user1\coggitsep2024\.git>cd objects
```

```
E:\coggitsep2024\user1\coggitsep2024\.git\objects>cd ..
```

```
E:\coggitsep2024\user1\coggitsep2024\.git>cd refs
```

```
E:\coggitsep2024\user1\coggitsep2024\.git\refs>dir
```

```
Volume in drive E is New Volume
```

```
Volume Serial Number is 3AFE-484F
```

```
Directory of E:\coggitsep2024\user1\coggitsep2024\.git\refs
```

```
30-09-2024  11:54    <DIR>          .
30-09-2024  11:54    <DIR>          heads
30-09-2024  11:54    <DIR>          remotes
30-09-2024  11:54    <DIR>          tags
                  0 File(s)            0 bytes
                  4 Dir(s)  43,027,730,432 bytes free
```

```
E:\coggitsep2024\user1\coggitsep2024\.git\refs>
```



.git File

- A branch is simply the movable pointer to one of these commits.
- The default branch name in Git is called master.
- As we start making commits, we're given a master branch that points to the last commit you made.
- Every time we commit, the master branch pointer moves forward automatically.
- It's the head that does this.
- Head is a pointer that points to the name of the current branch.
- It moves automatically when you checkout a new branch or make a commit to a branch.



.git File

- HEAD: We can think of the HEAD as the current branch. it's probably write refs/heads/master.
- FETCH_HEAD: records the branch which we fetched from a remote repository with your last git fetch invocation
- ORIG_HEAD: created by commands that move your HEAD in a drastic way, to record the position of the HEAD before their operation, so that we can easily change the tip of the branch back to the state before you ran them
- MERGE_HEAD: Records the commit(s) which you are merging into your branch when you run git merge



.git File

- config: Used to set Git configuration values on a global or local project level.
- index: Git index is a staging area between the working directory and repository.
- description: Git generates a file named description which contains the name of the repository as set by the user.
- Default value of the file description: “Unnamed repository”.
- Edit this file “description” to name the repository.
- It is used by GitWeb and is not considered by GitHub or GitLab.



GIT Data Storage Architecture

- Git's data storage architecture is a sophisticated system that stores all the information necessary to manage and track changes in a project's files.
- Data Storage Break Down
 - Content-Addressable Storage (CAS)
 - Objects in Git
- File System Layout
- Commit Graph (DAG)



GIT Data Storage Architecture

- Delta Storage and Compression
- Efficient Data Storage
- Garbage Collection



Content Addressable Graph

- Git's core storage model is content-addressable storage.
- Each piece of data (whether a file, directory, commit, etc.) is stored as an object and referenced by a SHA-1 hash of its content.
- This means that if two files (or objects) have the same content, they will share the same hash and will not be stored twice, reducing redundancy.



Objects in GIT

- Git uses four main types of objects in its storage architecture, all stored in the .git/objects directory.
- Each object is referenced by a 40-character SHA-1 hash.
- a) Blob (Binary Large Object)
 - Purpose: Stores the actual file content.
 - Details: A blob is the simplest object type.
 - It stores the contents of a file but does not store the filename or any other metadata.
 - It's simply a snapshot of the file's contents at a certain point in time.



Objects in GIT

- b) Tree
 - Purpose:
 - Represents a directory and its contents.Detailed:
 - A tree object stores metadata about files and directories.
 - It contains pointers (SHA-1 hashes) to blob objects (files) and other tree objects (subdirectories).
 - It also stores file names and file modes (permissions).



Objects in GIT

- c) Commit
 - Purpose:
 - Represents a snapshot of the project at a particular point in time.
 - Details:
 - A commit object stores information about the state of the repository at a given time.
 - It contains metadata such as the author, commit message, a reference to the tree object representing the directory at that point, and pointers to any parent commits (for previous states).



Objects in GIT

- d) Tag
 - Purpose:
 - A reference to a commit (often used to mark releases).
 - Details:
 - Tags can point to specific commits.
 - They are commonly used to mark significant points in the project, such as version releases.
 - Tags can either be lightweight (just a reference to a commit) or annotated (containing extra metadata).



File System Layout

- When you initialize a Git repository, the system creates a `.git` directory in the root of the project.
- Here is an overview of key directories and files used for storage:
 - `.git/objects/`: Stores the actual Git objects (blobs, trees, commits, and tags).
 - Each object is stored as a compressed file under a subdirectory named by the first two characters of its SHA-1 hash.
 - `.git/refs/`: Stores references to commits, including branches (`.git/refs/heads/`), tags (`.git/refs/tags/`), and other refs (e.g., remotes).
 - `.git/index`: Represents the staging area where changes to files are prepared before committing.
 - It's an important structure that sits between the working directory and the object store.
 - `.git/HEAD`: A special file that stores the reference to the current branch.
 - It points to one of the refs, usually a branch under `.git/refs/heads/`.



Commit Graph (DAG)

- Git's commit history is stored in a Directed Acyclic Graph (DAG) structure.
- Each commit object points to a tree object (which represents the state of the directory) and possibly one or more parent commit objects.
- This forms a chain of commits, tracking the entire history of the project in a non-linear fashion (due to branching and merging).
 - Branches are pointers to specific commits in the DAG.
 - Merges are commits with more than one parent, representing the merging of two branches.



Delta Storage and Compression

- Git efficiently stores file history using delta compression and pack files:
- a) Loose Objects When a new object (blob, tree, commit, or tag) is created, it is initially stored as a loose object—a single compressed file under the .git/objects/ directory.



Delta Storage and Compression

- b) Pack Files
 - As the repository grows, Git uses a mechanism to store objects more efficiently.
 - Git collects many objects into pack files by delta-compressing similar objects (e.g., different versions of the same file).
 - This process happens automatically during operations like `git gc` (garbage collection) or `git repack`.
 - Delta compression reduces storage requirements by storing only the differences between successive versions of a file.



Efficient Data Storage

- Git is designed for efficiency in storing data, particularly for large projects.
- It uses the following strategies:
 - Deduplication: Since objects are identified by the hash of their content, identical objects are only stored once.
 - If a file is unchanged between commits, it is not stored again.
 - Delta Encoding:
 - Git uses delta compression to store changes between files rather than storing entire files for every commit, significantly reducing the size of history.
 - Compression: All objects in Git are compressed, whether they are stored as loose objects or packed into pack files.



Garbage Collection

- Over time, Git may accumulate objects that are no longer needed (e.g., from aborted commits, deleted branches, etc.).
- Git's garbage collection mechanism (`git gc`) cleans up these unreferenced objects to save space.
- It also packs loose objects into more efficient pack files.

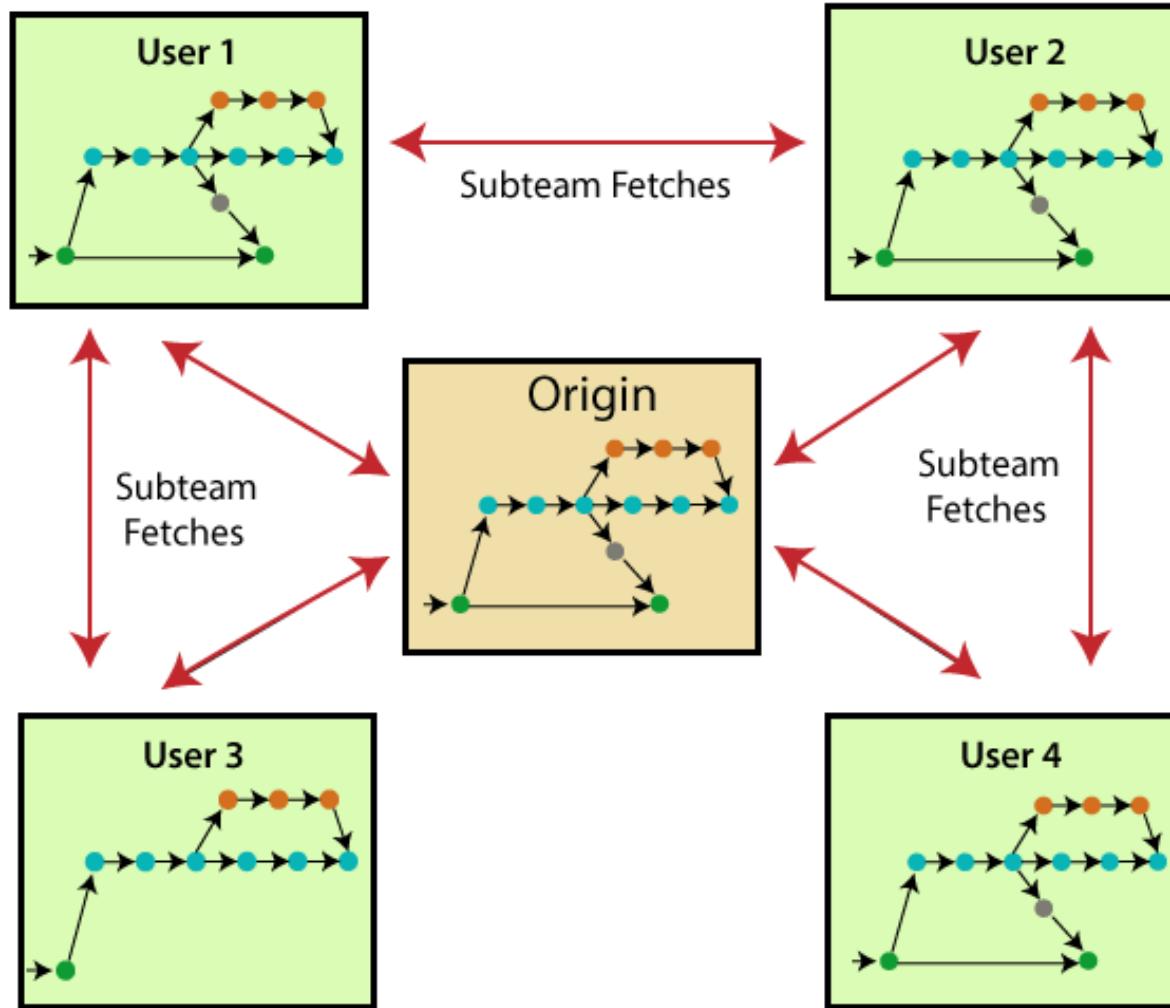
Summary of Key Features in Git's Data Storage Architecture:



- Object-Oriented: All data in Git is stored as objects (blob, tree, commit, tag).
- SHA-1 Hashing: Git uses SHA-1 hashes to identify all objects, ensuring immutability and integrity.
- Efficient Storage: Git avoids duplication by reusing objects when content is identical and employs delta compression to store changes efficiently.
- Pack Files: Objects are packed and compressed for better storage performance.
- DAG for Commits: The commit history is stored as a Directed Acyclic Graph, enabling branching and merging.



Features of Git





Features of Git

- **Open Source**

- Git is an open-source tool. It is released under the GPL (General Public License) license.

- **Scalable**

- Git is scalable, which means when the number of users increases, the Git can easily handle such situations.

- **Distributed**

- One of Git's great features is that it is distributed.
 - Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository.
 - Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project.
 - We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.



Features of Git

- **Security**

- Git is secure. It uses the SHA1 (Secure Hash Function) to name and identify objects within its repository.
- Files and commits are checked and retrieved by its checksum at the time of checkout.
- It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit.
- Once it is published, one cannot make changes to its old version.

- **Speed**

- Git is very fast, so it can complete all the tasks in a while.
- Most of the git operations are done on the local repository, so it provides a huge speed.
- Also, a centralized version control system continually communicates with a server somewhere.
- Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs.
- Fetching version history from a locally stored repository is much faster than fetching it from the remote server.
- The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages.
- Git was developed to work on the Linux kernel; therefore, it is capable enough to handle large repositories effectively. From the beginning, speed and performance have been Git's primary goals.



Features of Git

- **Branching and Merging**

- Branching and merging are the great features of Git, which makes it different from the other SCM tools.
- Git allows the creation of multiple branches without affecting each other.
- We can perform tasks like creation, deletion, and merging on branches, and these tasks take a few seconds only.

Below are some features that can be achieved by branching:

- We can create a separate branch for a new module of the project, commit and delete it whenever we want.
- We can have a production branch, which always has what goes into production and can be merged for testing in the test branch.
- We can create a demo branch for the experiment and check if it is working. We can also remove it if needed.
- The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.



Features of Git

- **Data Assurance**
 - The Git data model ensures the cryptographic integrity of every unit of our project.
 - It provides a unique commit ID to every commit through a SHA algorithm.
 - We can retrieve and update the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default.
- **Maintain the clean history**
 - Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

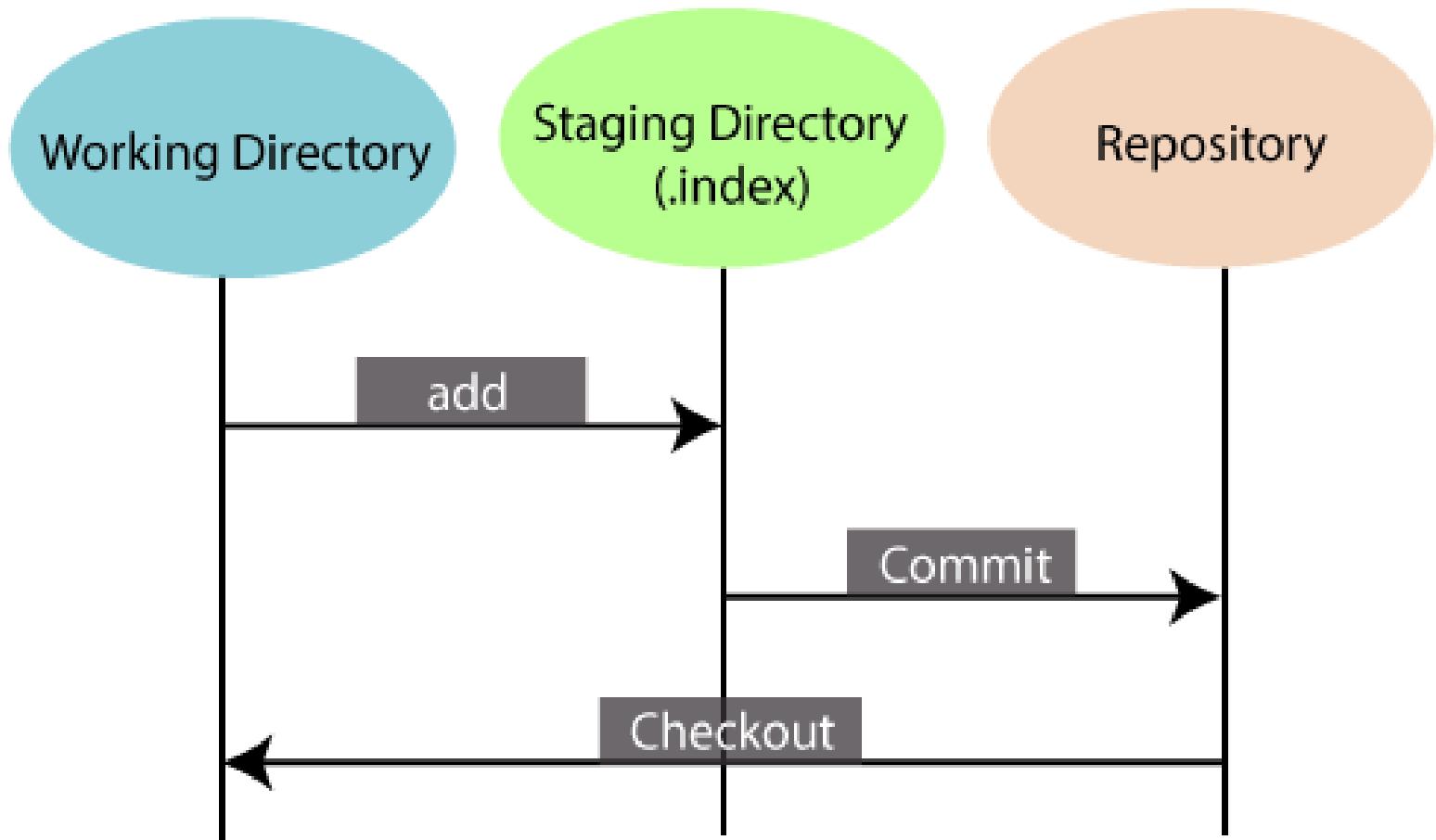


Features of Git

- **Staging Area**
 - The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion.
 - When you make a commit, Git takes changes that are in the staging area and make them as a new commit.
 - We are allowed to add and remove changes from the staging area.
 - The staging area can be considered as a place where Git stores the changes.
 - Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

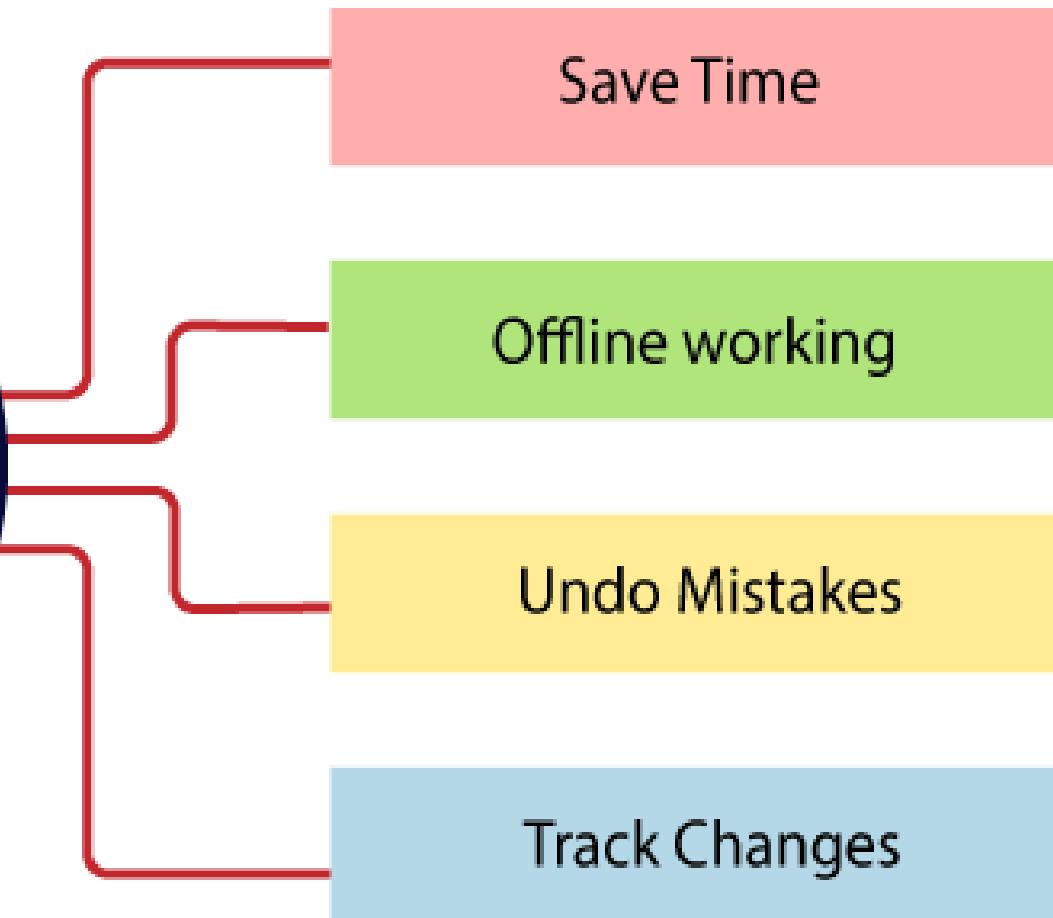


Features of Git



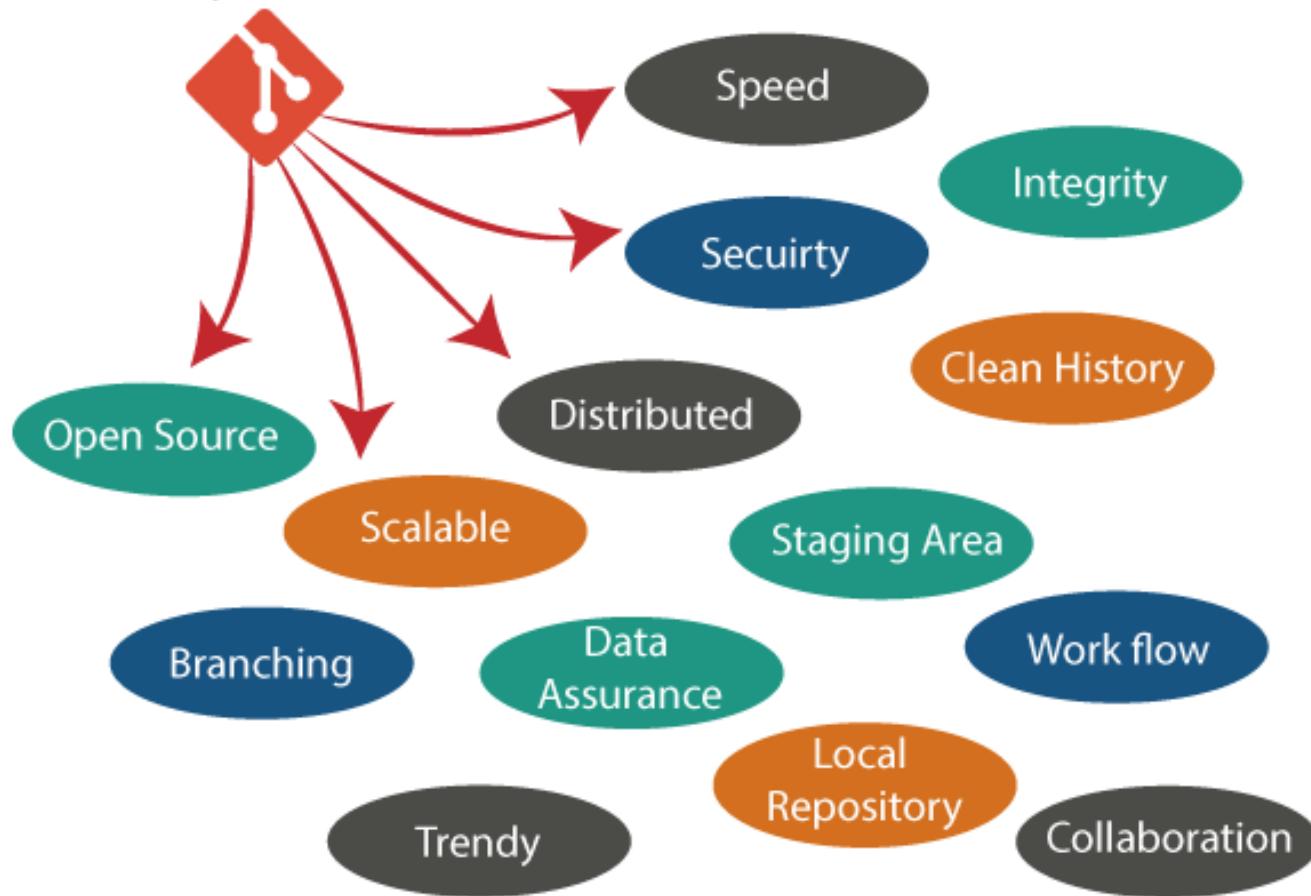


Benefits of Git



Why Git

Why Git?

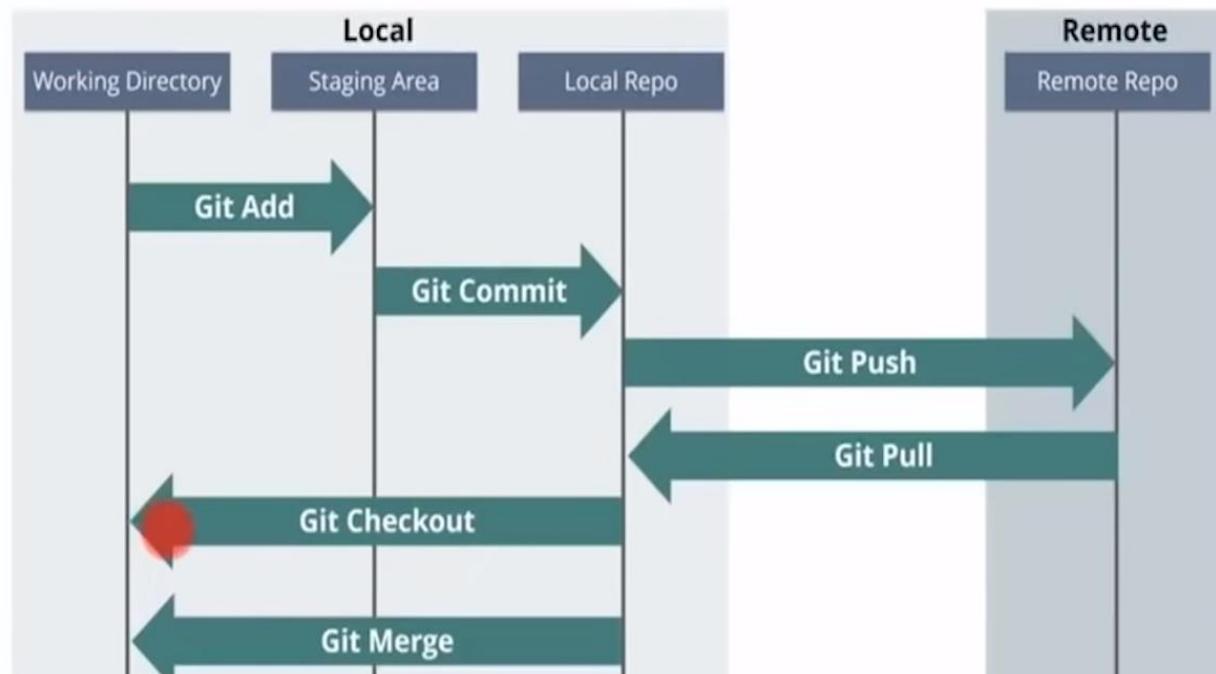


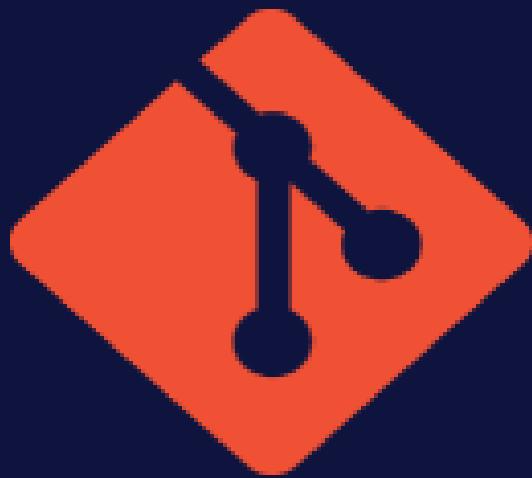


Source Code Management



Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software





Git

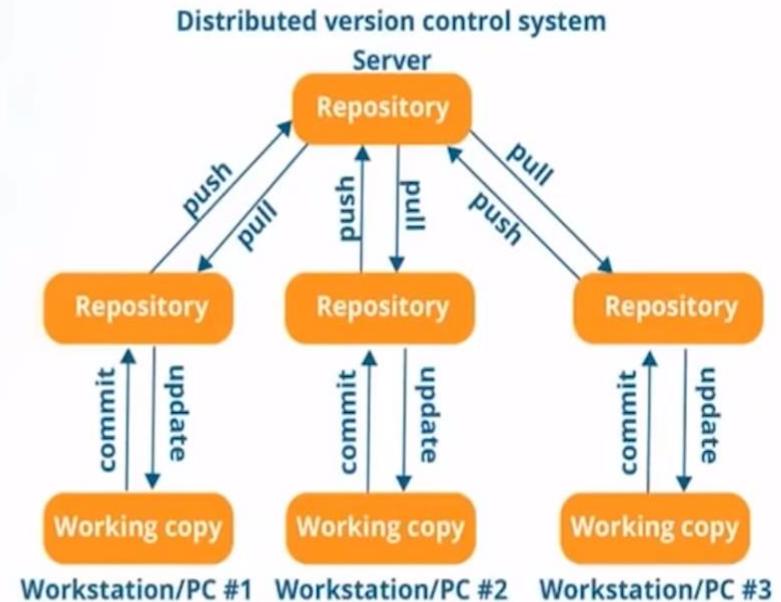
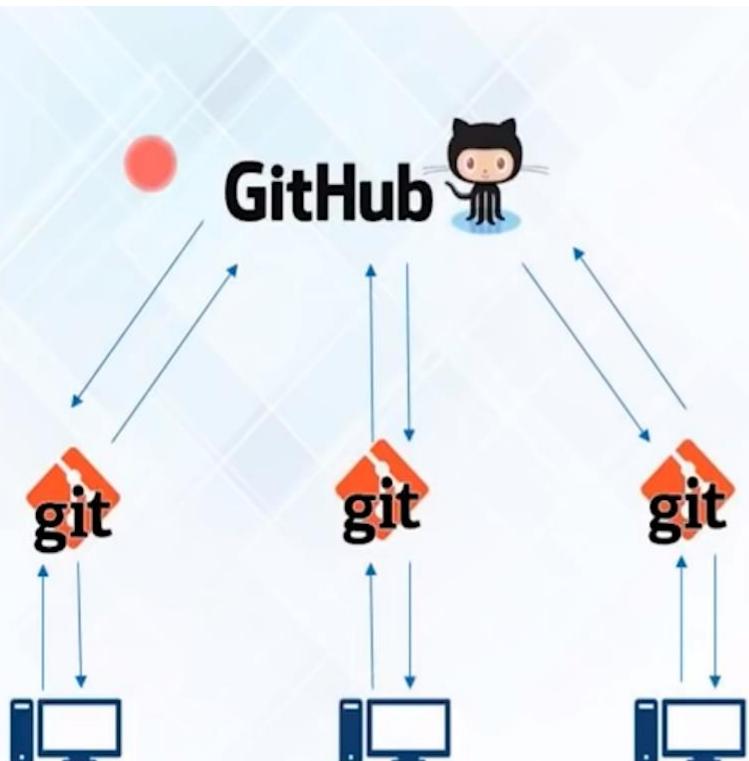
VS



GitHub



Git vs Github





Git vs Github

Git	GitHub
Git is a distributed version control tool that can manage a programmer's source code history.	GitHub is a cloud-based tool developed around the Git tool.
A developer installs Git tool locally.	GitHub is an online service to store code and push from the computer running the Git tool.
Git focused on version control and code sharing.	GitHub focused on centralized source code hosting.
It is a command-line tool.	It is administered through the web.
It facilitates with a desktop interface called Git Gui.	It also facilitates with a desktop interface called GitHub Gui.
Git does not provide any user management feature.	GitHub has a built-in user management feature.
It has minimal tool configuration feature.	It has a market place for tool configuration.



How to Install Git on Windows

The screenshot shows a web browser window with the title "Git - Downloads" and the URL "git-scm.com/downloads". The page content includes:

- A sidebar with links for "Mac OS X", "Windows", and "Linux/Unix".
- A message: "Older releases are available and the Git source repository is on GitHub."
- A large monitor icon displaying the "Latest source Release" for "2.23.0" (Release Notes from 2019-08-16) with a "Download 2.23.0 for Windows" button.
- A "GUI Clients" section with text about built-in tools like `git-gui` and `gitk`, and a link to "View GUI Clients →".
- A "Logos" section with text about various Git logos available in PNG and EPS formats, and a link to "View Logos →".
- A footer section with a link to "Git via Git".



How to Install Git on Windows

Git 2.23.0 Setup

Information

Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change

<https://gitforwindows.org/>

Next > Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Select Components

Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- Additional icons
 - On the Desktop
- Windows Explorer integration
 - Git Bash Here
 - Git GUI Here
- Git LFS (Large File Support)
- Associate .git* configuration files with the default text editor
- Associate .sh files to be run with Bash
- Use a TrueType font in all console windows
- Check daily for Git for Windows updates

Current selection requires at least 253.0 MB of disk space.
<https://gitforwindows.org/>

< Back **Next >** Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Adjusting your PATH environment

How would you like to use Git from the command line?

Use Git from Git Bash only

This is the most cautious choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

Git from the command line and also from 3rd-party software

(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools.
You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.

Use Git and optional Unix tools from the Command Prompt

Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

<https://gitforwindows.org/>

< Back Next > Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Choosing HTTPS transport backend

Which SSL/TLS library would you like Git to use for HTTPS connections?

Use the OpenSSL library

Server certificates will be validated using the ca-bundle.crt file.

Use the native Windows Secure Channel library

Server certificates will be validated using Windows Certificate Stores.
This option also allows you to use your company's internal Root CA certificates distributed e.g. via Active Directory Domain Services.

<https://gitforwindows.org/>

< Back Next > Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Configuring the line ending conversions
How should Git treat line endings in text files?

Checkout Windows-style, commit Unix-style line endings

Git will convert LF to CRLF when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Windows ("core.autocrlf" is set to "true").

Checkout as-is, commit Unix-style line endings

Git will not perform any conversion when checking out text files. When committing text files, CRLF will be converted to LF. For cross-platform projects, this is the recommended setting on Unix ("core.autocrlf" is set to "input").

Checkout as-is, commit as-is

Git will not perform any conversions when checking out or committing text files. Choosing this option is not recommended for cross-platform projects ("core.autocrlf" is set to "false").

<https://gitforwindows.org/>

< Back **Next >** Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Configuring the terminal emulator to use with Git Bash

Which terminal emulator do you want to use with your Git Bash?

Use MinTTY (the default terminal of MSYS2)

Git Bash will use MinTTY as terminal emulator, which sports a resizable window, non-rectangular selections and a Unicode font. Windows console programs (such as interactive Python) must be launched via `winpty` to work in MinTTY.

Use Windows' default console window

Git will use the default console window of Windows ("cmd.exe"), which works well with Win32 console programs such as interactive Python or node.js, but has a very limited default scroll-back, needs to be configured to use a Unicode font in order to display non-ASCII characters correctly, and prior to Windows 10 its window was not freely resizable and it only allowed rectangular text selections.

<https://gitforwindows.org/>

< Back **Next >** Cancel



How to Install Git on Windows

Git 2.23.0 Setup

Configuring extra options
Which features would you like to enable?

Enable file system caching
File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

Enable Git Credential Manager
The [Git Credential Manager for Windows](#) provides secure Git credential storage for Windows, most notably multi-factor authentication support for Visual Studio Team Services and GitHub. (requires .NET framework v4.5.1 or later).

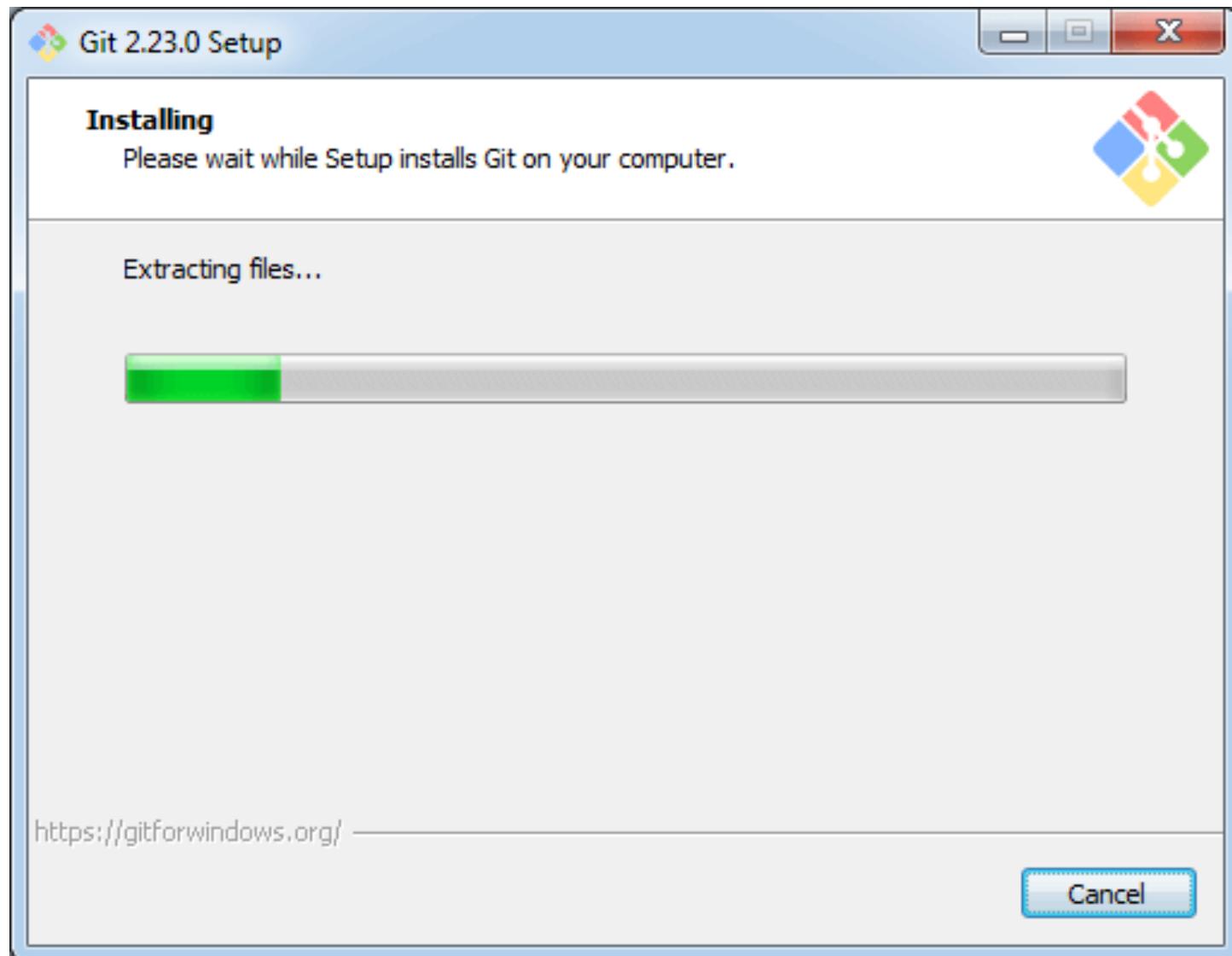
Enable symbolic links
Enable [symbolic links](#) (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

<https://gitforwindows.org/>

< Back [Next >](#) Cancel

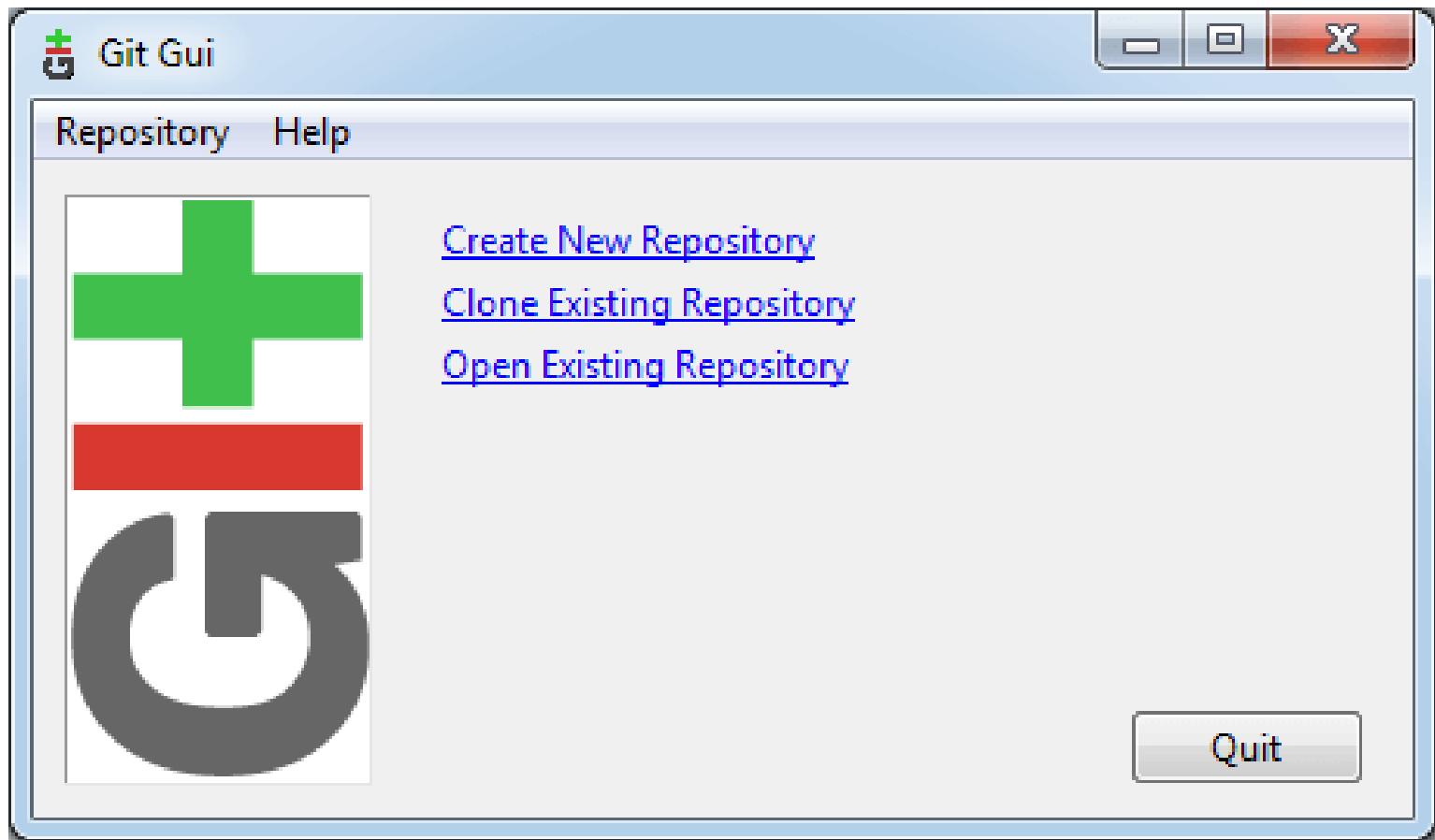


How to Install Git on Windows



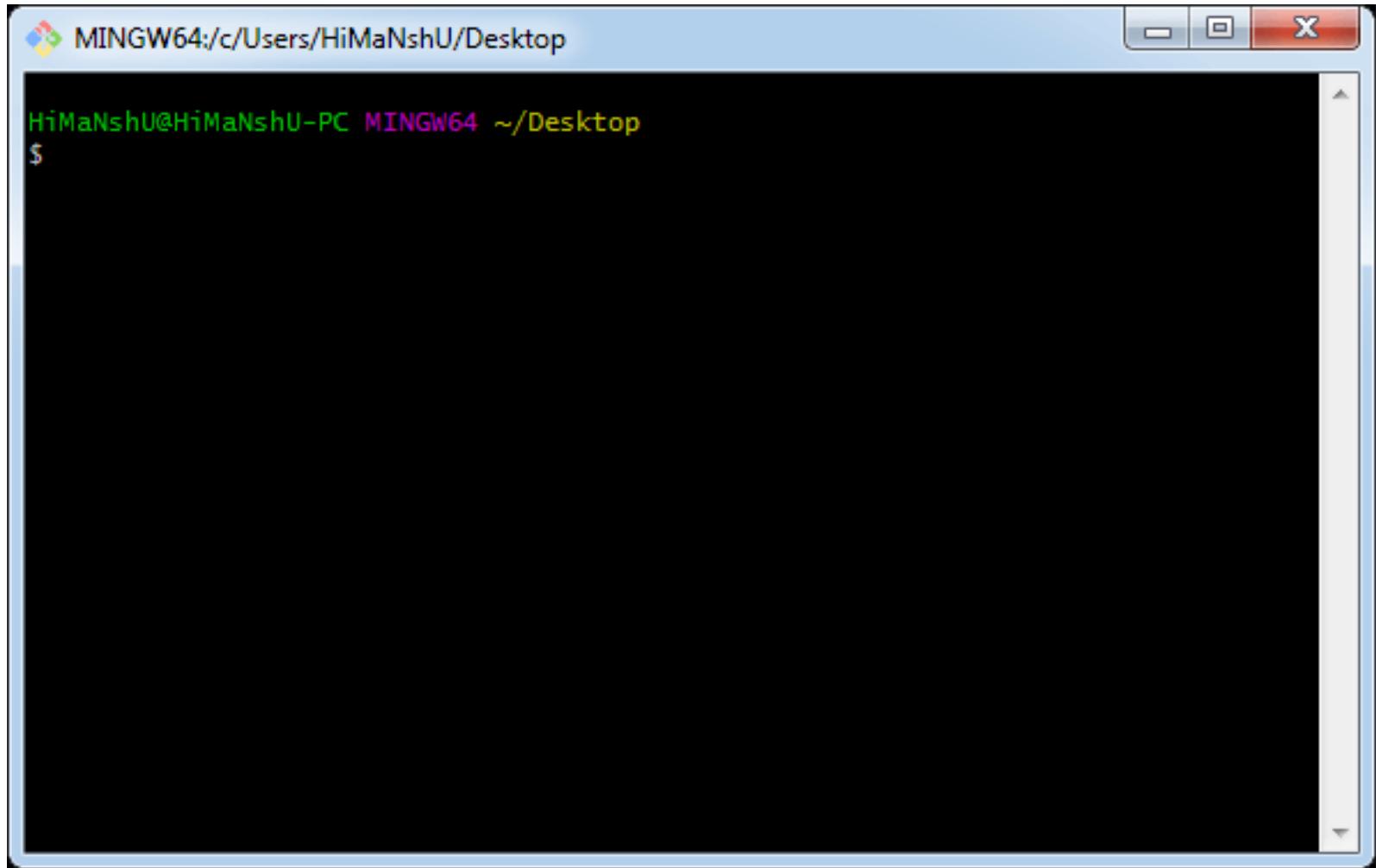


How to Install Git on Windows





How to Install Git on Windows

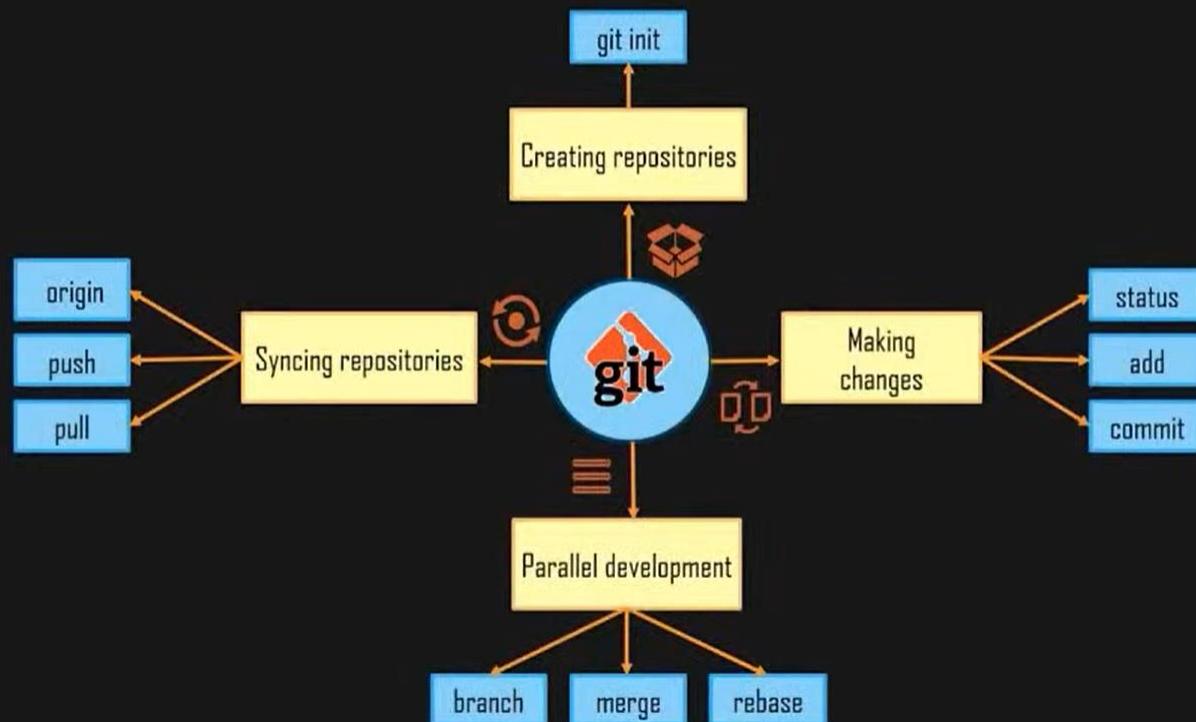
A screenshot of a Windows terminal window titled "MINGW64:/c/Users/HiMaNshU/Desktop". The title bar includes the MinGW logo, the path, and standard window controls. The main area is a black terminal window showing a command prompt: "HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop" followed by a "\$" sign. The window has a light blue border and scroll bars on the right side.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop$
```



GIT Commands

GIT OPERATIONS & COMMANDS





GIT Commands

GIT OPERATIONS

CREATE REPO

SYNC REPO

MAKE CHANGES

PARALLEL DEVELOPMENT

BRANCH

MERGE

REBASE

git init



The **git init** command creates a new **Git** repository.

git clone



When you **clone** a repository, you create a copy of the original repository on your **Local machine**.

git fork

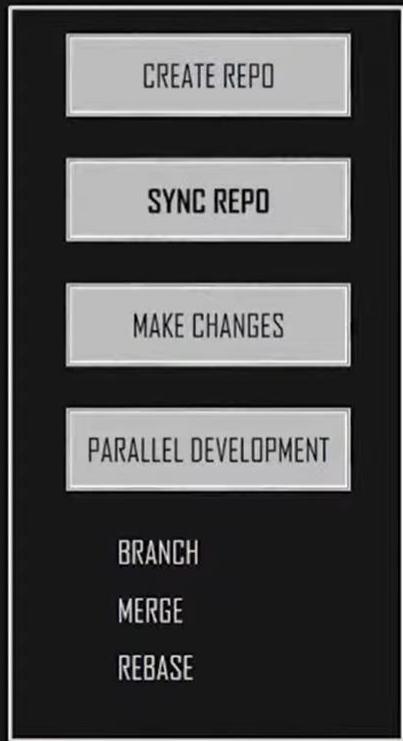


When you **fork** a repository, you create a copy of the original repository on your **GitHub account**.



GIT Commands

GIT OPERATIONS



git origin

git remote add origin
<repo_link>

Lets you **add** a remote repository.

git pull

git pull origin master

Lets you **copy** all the files from the master branch of **remote repository to your local repository**.

git push

git push origin master

Lets you push your **local changes into central repository**



Git Environment Setup

- Git supports a command called `git config` that lets you get and set configuration variables that control all facets of how Git looks and operates.
- It is used to set Git configuration values on a global or local project level.
- Setting `user.name` and `user.email` are the necessary configuration options as your name and email will show up in your commit messages.



Git Environment Setup

- \$ git config --global user.name "Hemanth"
- \$ git config --global user.email hemanth@gmail.com
- git config --global core.editor Vim
- git config - -list
- Git config -global color.ui true



Git Environment Setup

```
# git config --global user.name "Mike McQuaid"  
# git config --global user.email mike@mikemcquaid.com  
# git config --global user.email  
  
mike@mikemcquaid.com
```

1 A run command

2 No output

3 Command output



Git configuration levels

- The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.
 - local
 - global
 - system



.git subdirectory

- Under the new Git repository directory, a .git subdirectory at /Users/mike/GitIn-
- PracticeRedux/.git/ (for example) is created with various files and directories under it.
- WHY IS THE .GIT DIRECTORY NOT VISIBLE?
- On some operating systems, directories starting with a period (.) such as .git are hidden by default.
- They can still be accessed in the console using their full path (such as /Users/mike/GitInPracticeRedux/.git/) but won't show up in file listings in file browsers or by running a command like ls /Users/mike/GitInPracticeRedux/.



.git subdirectory

```
# cd /Users/mike/ && find GitInPracticeRedux
```

GitInPracticeRedux/.git/config ↪ ① Local configuration
GitInPracticeRedux/.git/description ↪ ② Description file
GitInPracticeRedux/.git/HEAD ↪ ③ HEAD pointer
GitInPracticeRedux/.git/hooks/applypatch-msg.sample ↪ ④ Event hooks
GitInPracticeRedux/.git/hooks/commit-msg.sample
GitInPracticeRedux/.git/hooks/post-update.sample
GitInPracticeRedux/.git/hooks/pre-applypatch.sample
GitInPracticeRedux/.git/hooks/pre-commit.sample
GitInPracticeRedux/.git/hooks/pre-push.sample
GitInPracticeRedux/.git/hooks/pre-rebase.sample
GitInPracticeRedux/.git/hooks/prepare-commit-msg.sample
GitInPracticeRedux/.git/hooks/update.sample
GitInPracticeRedux/.git/info/exclude ↪ ⑤ Excluded files
GitInPracticeRedux/.git/objects/info ↪ ⑥ Object information
GitInPracticeRedux/.git/objects/pack ↪ ⑦ Pack files
GitInPracticeRedux/.git/refs/heads ↪ ⑧ Branch pointers
GitInPracticeRedux/.git/refs/tags ↪ ⑨ Tag pointers



Git configuration levels

- --local
- It is the default level in Git. Git config will write to a local level if no configuration option is given. Local configuration values are stored in .git/config directory as a file.
- --global
- The global level configuration is user-specific configuration. User-specific means, it is applied to an individual operating system user. Global configuration values are stored in a user's home directory. ~/.gitconfig on UNIX systems and C:\Users\\.gitconfig on windows as a file format.



Git Terminology

- **Branch**
 - A branch is a version of the repository that diverges from the main working project. It is an essential feature available in most modern version control systems. A Git project can have more than one branch. We can perform many operations on Git branch-like rename, list, delete, etc.
- **Checkout**
 - In Git, the term checkout is used for the act of switching between different versions of a target entity. The git checkout command is used to switch between branches in a repository.
- **Cherry-Picking**
 - Cherry-picking in Git is meant to apply some commit from one branch into another branch. In case you made a mistake and committed a change into the wrong branch, but do not want to merge the whole branch. You can revert the commit and cherry-pick it on another branch.



Git Terminology

- **Clone**
 - The git clone is a Git command-line utility. It is used to make a copy of the target repository or clone it. If I want a local copy of my repository from GitHub, this tool allows creating a local copy of that repository on your local directory from the repository URL.
- **Fetch**
 - It is used to fetch branches and tags from one or more other repositories, along with the objects necessary to complete their histories. It updates the remote-tracking branches.
- **HEAD**
 - HEAD is the representation of the last commit in the current checkout branch. We can think of the head like a current branch. When you switch branches with git checkout, the HEAD revision changes and points the new branch.



Git Terminology

- **Index**
 - The Git index is a staging area between the working directory and repository. It is used as the index to build up a set of changes that you want to commit together.
- **Master**
 - Master is a naming convention for Git branch. It's a default branch of Git. After cloning a project from a remote server, the resulting local repository contains only a single local branch. This branch is called a "master" branch. It means that "master" is a repository's "default" branch.
- **Merge**
 - Merging is a process to put a forked history back together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch.



Git Terminology

- **Origin**
 - In Git, "origin" is a reference to the remote repository from a project was initially cloned. More precisely, it is used instead of that original repository URL to make referencing much easier.
- **Pull/Pull Request**
 - The term Pull is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory. The git pull command is used to make a Git pull.
 - Pull requests are a process for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their remote server account. Pull request announces all the team members that they need to review the code and merge it into the master branch.



Git Terminology

- **Push**
 - The push term refers to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository. Pushing is capable of overwriting changes; caution should be taken when pushing.
- **Rebase**
 - In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and visualized the process in the environment of a feature branching workflow.
 - From a content perception, rebasing is a technique of changing the base of your branch from one commit to another.
- **Remote**
 - In Git, the term remote is concerned with the remote repository. It is a shared repository that all team members use to exchange their changes. A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion and more.
 - In case of a local repository, a remote typically does not provide a file tree of the project's current state, as an alternative it only consists of the .git versioning data.



Git Terminology

- **Repository**
 - In Git, Repository is like a data structure used by VCS to store metadata for a set of files and directories. It contains the collection of the file as well as the history of changes made to those files. Repositories in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.
- **Stashing**
 - Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The git stash command enables you to switch branch without committing the current branch.
- **Tag**
 - Tags make a point as a specific point in Git history. It is used to mark a commit stage as important. We can tag a commit for future reference. Primarily, it is used to mark a projects initial point like v1.1. There are two types of tags.
- **Light-weighted tag**
- **Annotated tag**



Git Terminology

- **Upstream And Downstream**

- The term upstream and downstream is a reference of the repository. Generally, upstream is where you cloned the repository from (the origin) and downstream is any project that integrates your work with other works. However, these terms are not restricted to Git repositories.

- **Git Revert**

- In Git, the term revert is used to revert some commit. To revert a commit, git revert command is used. It is an undo type command. However, it is not a traditional undo alternative.

- **Git Reset**

- In Git, the term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.

- **Soft**
- **Mixed**
- **Hard**



Git Terminology

- **Git Ignore**
 - In Git, the term ignore used to specify intentionally untracked files that Git should ignore. It doesn't affect the Files that already tracked by Git.
- **Git Diff**
 - Git diff is a command-line utility. It's a multiuse Git command. When it is executed, it runs a diff function on Git data sources. These data sources can be files, branches, commits, and more. It is used to show changes between commits, commit, and working tree, etc.
- **Git Cheat Sheet**
 - A Git cheat sheet is a summary of Git quick references. It contains basic Git commands with quick installation. A cheat sheet or crib sheet is a brief set of notes used for quick reference. Cheat sheets are so named because the people may use it without no prior knowledge.



Git command line

- Git Config command
- Git init command
- Git clone command
- Git add command
- Git commit command
- Git status command
- Git push Command
- Git pull command
- Git Branch Command
- Git Merge Command
- Git log command
- Git remote command



Git command line

COMMAND	PURPOSE
add	Add file contents to the index.
bisect	Find by binary search the change that introduced a bug.
branch	List, create, or delete branches.
checkout	Switch branches or restore working tree files.
cherry	Find commits yet to be applied to upstream (branch on the remote).
cherry-pick	Apply the changes introduced by some existing commits.
clone	Clone a repository into a new directory.
commit	Record changes to the repository.
config	Get and set repository or global options.
diff	Show changes between commits, commits and working tree, and so on.
fetch	Download objects and refs from another repository.
grep	Print lines matching a pattern.
help	Display help information.
log	Show commit logs.
merge	Join two or more development histories together.
mv	Move or rename a file, directory, or symlink.
pull	Fetch from, or integrate with, another repository or a local branch.
push	Update remote refs along with associated objects.
rebase	Forward-port local commits to the updated upstream head.



Git command line

rerere	Reuse recorded resolution for merged conflicts.
reset	Reset current HEAD to the specified state.
revert	Revert some existing commits.
rm	Remove files from the working tree and from the index.
show	Show various types of objects.
status	Show the working tree status.
submodule	Initialize, update, or inspect submodules.
subtree	Merge subtrees and split repositories into subtrees.
tag	Create, list, delete, or verify a tagged object.
worktree	Manage multiple working trees.

Git commands





Git command line

- **Git config command**
- This command configures the user. The Git config command is the first and necessary command used on the Git command line. This command sets the author name and email address to be used with your commits. Git config is also used in other scenarios.
- Syntax
 - **\$ git config --global user.name "ImHemanth"**
 - **\$ git config --global user.email "Hemanth@gmail.com"**
- **Git Init command**
- This command is used to create a local repository.
- Syntax
 - **\$ git init Demo**



Git command line

- **Git clone command**
- This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows creating a local copy of that repository on your local directory from the repository URL.
- Syntax
- **\$ git clone URL**



Git command line

- **Git add command**
- This command is used to add one or more files to staging (Index) area.
- Syntax
- To add one file
- **\$ git add Filename**
- To add more than one file
- **\$ git add***



Git command line

- **Git commit command**
- Commit command is used in two scenarios. They are as follows.
- **Git commit -m**
- This command changes the head. It records or snapshots the file permanently in the version history with a message.
- Syntax
- **\$ git commit -m " Commit Message"**
- **Git commit -a**
- This command commits any files added in the repository with git add and also commits any files you've changed since then.
- Syntax
- **\$ git commit -a**



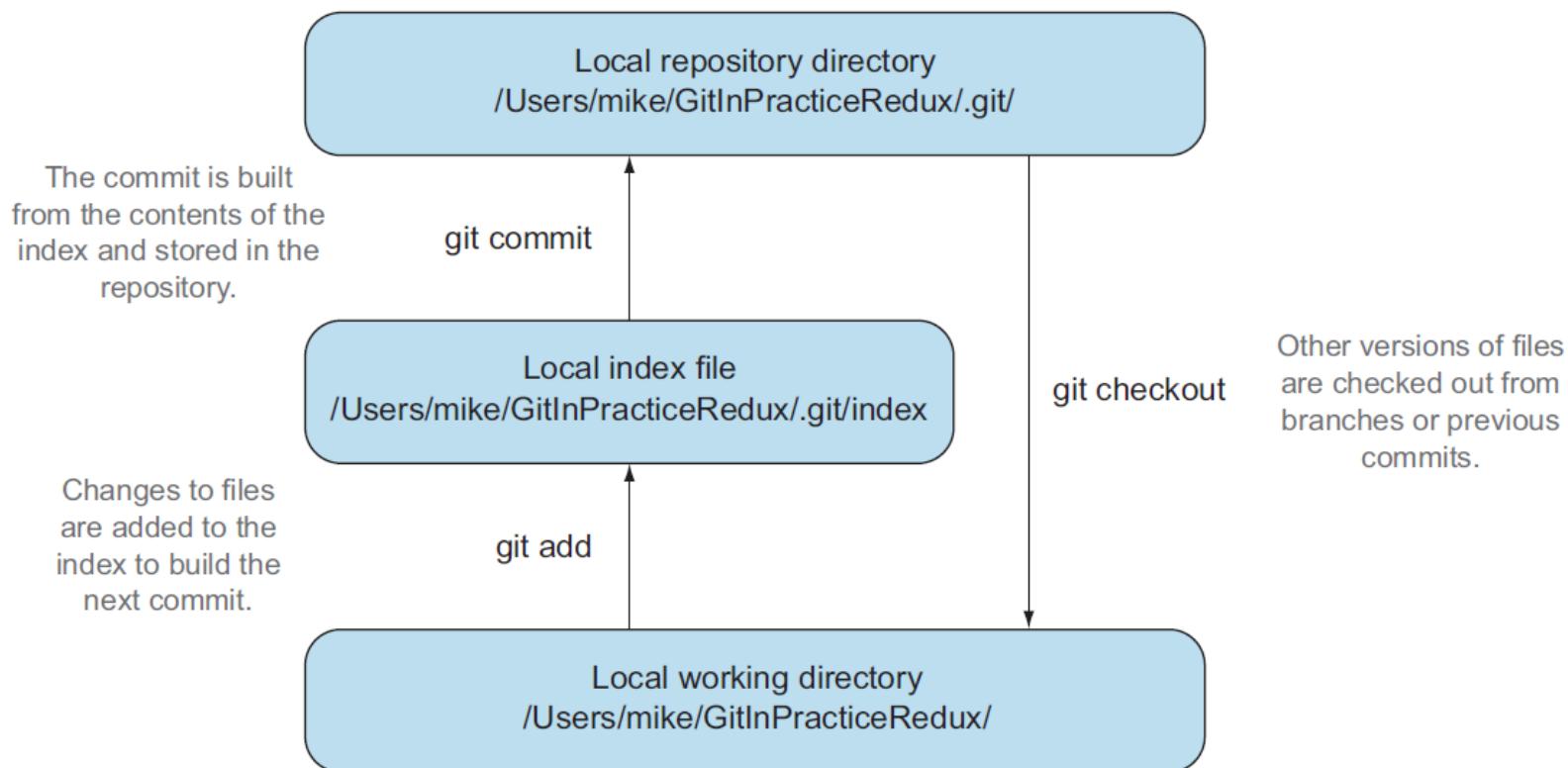
Git command line

- **Git status command**
- The status command is used to display the state of the working directory and the staging area.
- It allows you to see which changes have been staged, which haven't, and which files aren't being tracked by Git.
- It does not show you any information about the committed project history.
- For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.
- Syntax
- **\$ git status**



Creating a new commit: git add, git commit

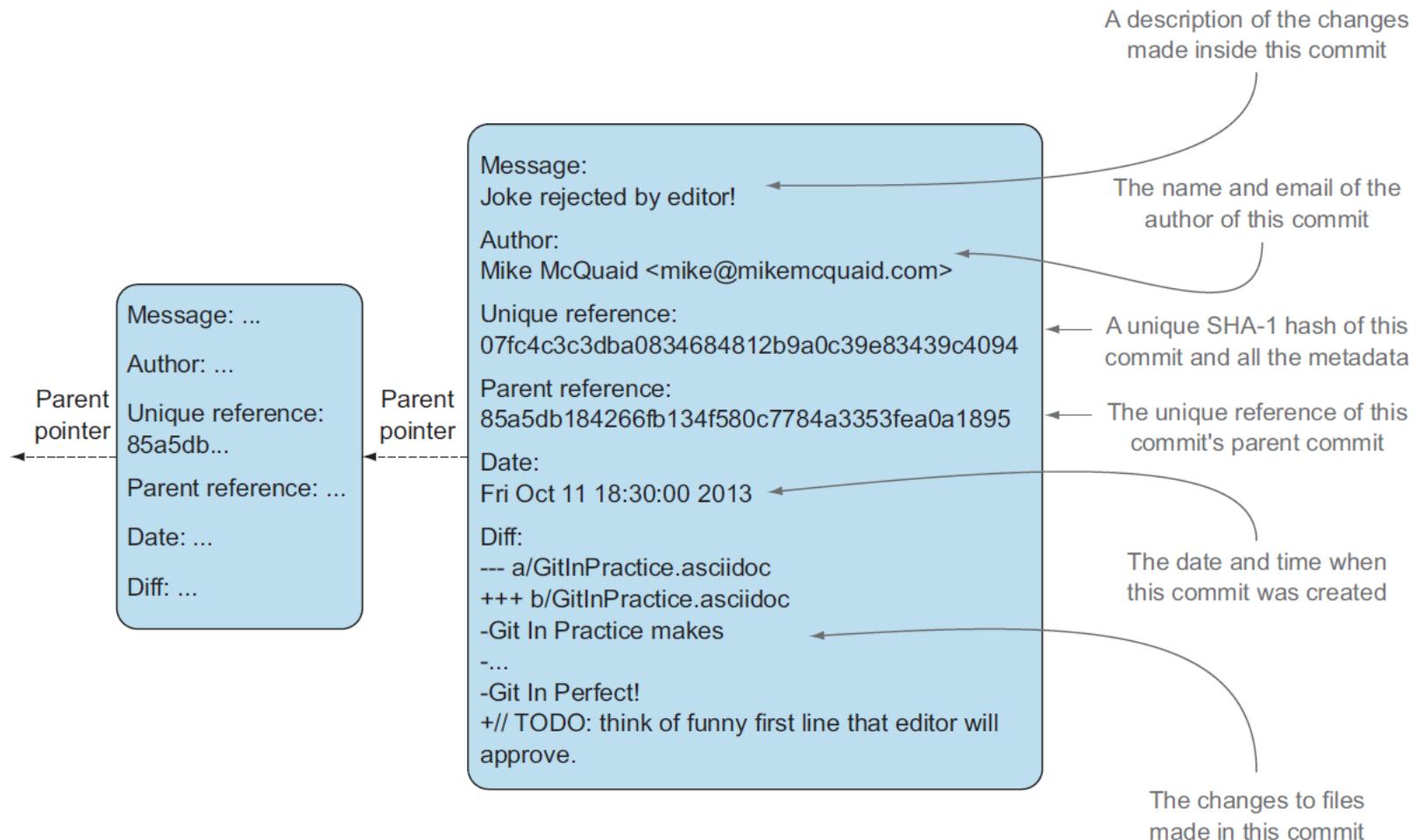
The repository stores the complete history of all the commits that have been previously made.



The working directory contains the current state of all the files that can be changed and versioned by Git.



Committing changes to files: git commit





Object Store

- Git is a version control system built on top of an object store.
- Git creates and stores a collection of objects when you commit.
- The object store is stored inside the Git repository.
- We can see the main Git objects we're concerned with commits, blobs, and trees.
- The file-contents reference is a reference to a tree object.
- A tree object stores a reference to all the blob objects at a particular point in time and other tree objects if there are any subdirectories.
- A blob object stores the contents of a particular version of a particular single file in the Git repository.

Object Store

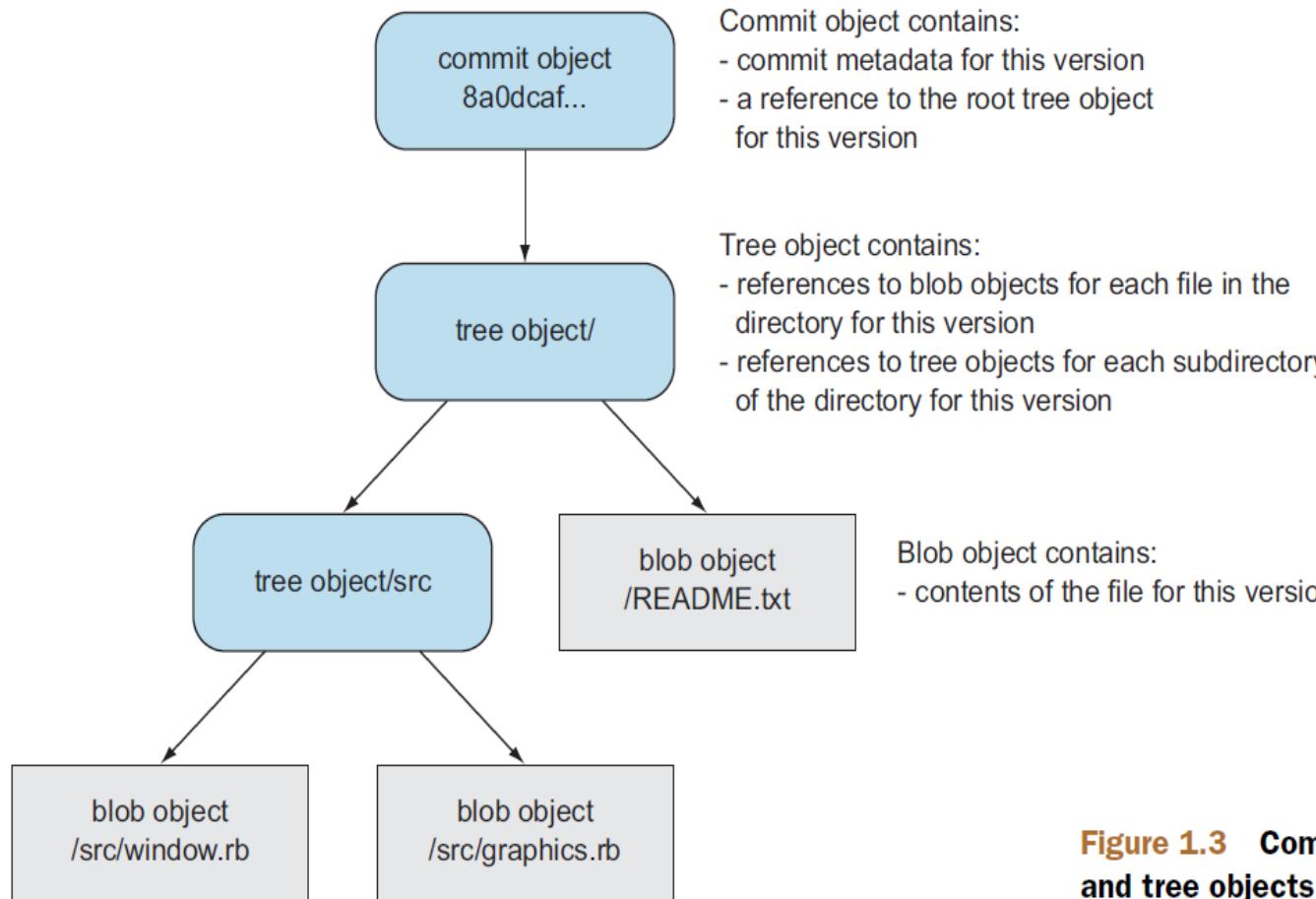
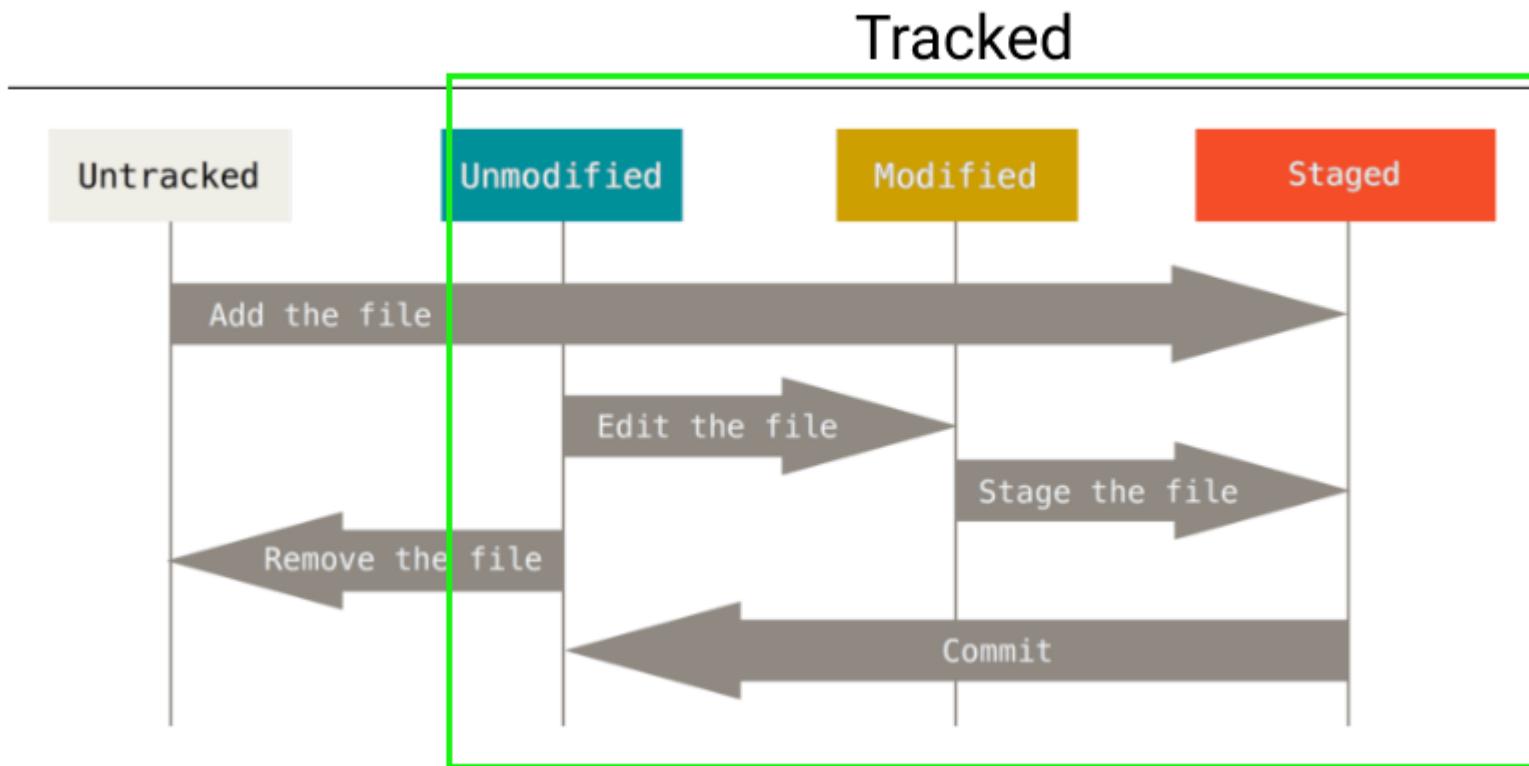


Figure 1.3 Commit, blob, and tree objects



Git Recording Changes





Smart Git

gitangular - [G:\Local disk\git\gittraining\gitangular] - SmartGit 21.2.4 For non-commercial use only (single user)

Repository Edit View Remote Local Branch Query Tools Review Window Help

Pull V Sync V Push V Stage Index Editor Unstage Discard Save Stash V Apply Stash Blame Investigate

Repositories + Graph C:\Users\Balasubramaniam D:\ F:\ gitangular (master)

Working Tree/Index (1 changed)

- master|origin/master Customer.cs merged
- dev2|origin/dev2 Customer.cs line no 8 modified by dev2
- Customer.cs merged
- Customer.cs line no 8 modified
- Customer.cs updated by developer2
- Customer.cs modified by developer1
- dev1|origin/dev1 Customer.cs added
- Merge branch 'master' of https://github.com/eswaribala/gitangularsep2022
- Initial commit
- Readme.txt committed

Yesterday 11:42 PM
Yesterday 11:15 PM
Yesterday 11:39 PM
Yesterday 11:08 PM
Yesterday 11:05 PM
Yesterday 10:13 PM
Yesterday 09:58 PM
Yesterday 09:34 PM
Yesterday 09:24 PM
Yesterday 09:16 PM
Yesterday 09:14 PM

Amend last commit Commit

Files + File Filter * | Comments

Name State Relative Directory Renamed P

Customer.cs.orig Untrac...

Changes of Customer.cs.orig (Untracked) - Index vs. Working Tree

```
1 Class Customer{  
2     public String CustomerId{get;set;}  
3     # developer 1  
4     # developer 2  
5     public String CustomerName{get;set;}  
6     public String Email{get;set;}  
7     <<<<< HEAD  
8     public String DOB {get;set;}  
9     =====  
10    public String Gender{get;set;}  
11    >>>>> dev2  
12 }  
13  
14  
15  
16 }
```

Unified Side by Side Compact

Ready



Git command line

- **Git status command**
- The status command is used to display the state of the working directory and the staging area.
- It allows you to see which changes have been staged, which haven't, and which files aren't being tracked by Git.
- It does not show you any information about the committed project history.
- For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.
- Syntax
- **\$ git status**



Git Recording Changes

```
Microsoft Windows [Version 10.0.22000.856]
(c) Microsoft Corporation. All rights reserved.
```

```
F:\sapientws2022>git status
On branch projects
nothing to commit, working tree clean
```

```
F:\sapientws2022>git status
On branch projects
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java

no changes added to commit (use "git add" and/or "git commit -a")
```

```
F:\sapientws2022>
```



Git Undoing Changes (Reset)

- The basic form of the reset command is as follows:
- `git reset [-q] [<tree-ish>] [--] <paths>...`
- `git reset (--patch | -p) [<tree-ish>] [--] [<paths>...]`
- `git reset [--soft | --mixed [-N] | --hard | --merge | --keep] [-q] [<commit>]`



Git Undoing Changes

```
F:\sapientws2022>git add .
```

```
F:\sapientws2022>git commit -m "updated"  
[projects d726761] updated  
 1 file changed, 2 insertions(+)
```

```
F:\sapientws2022>git statu  
git: 'statu' is not a git command. See 'git --help'.
```

```
The most similar commands are  
  status  
  stage  
  stash
```

```
F:\sapientws2022>git status  
On branch projects  
nothing to commit, working tree clean
```

```
F:\sapientws2022>git reset HEAD README.md
```

```
F:\sapientws2022>git status  
On branch projects  
nothing to commit, working tree clean
```

```
F:\sapientws2022>git reset HEAD~  
Unstaged changes after reset:  
M      cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
```

```
F:\sapientws2022>_
```



Git Undoing Changes

```
F:\sapientws2022>git status
```

```
On branch projects  
nothing to commit, working tree clean
```

```
F:\sapientws2022>git reset HEAD~
```

```
Unstaged changes after reset:  
M      cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
```

```
F:\sapientws2022>git status
```

```
On branch projects
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working directory)
```

```
modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
F:\sapientws2022>
```



Git command line

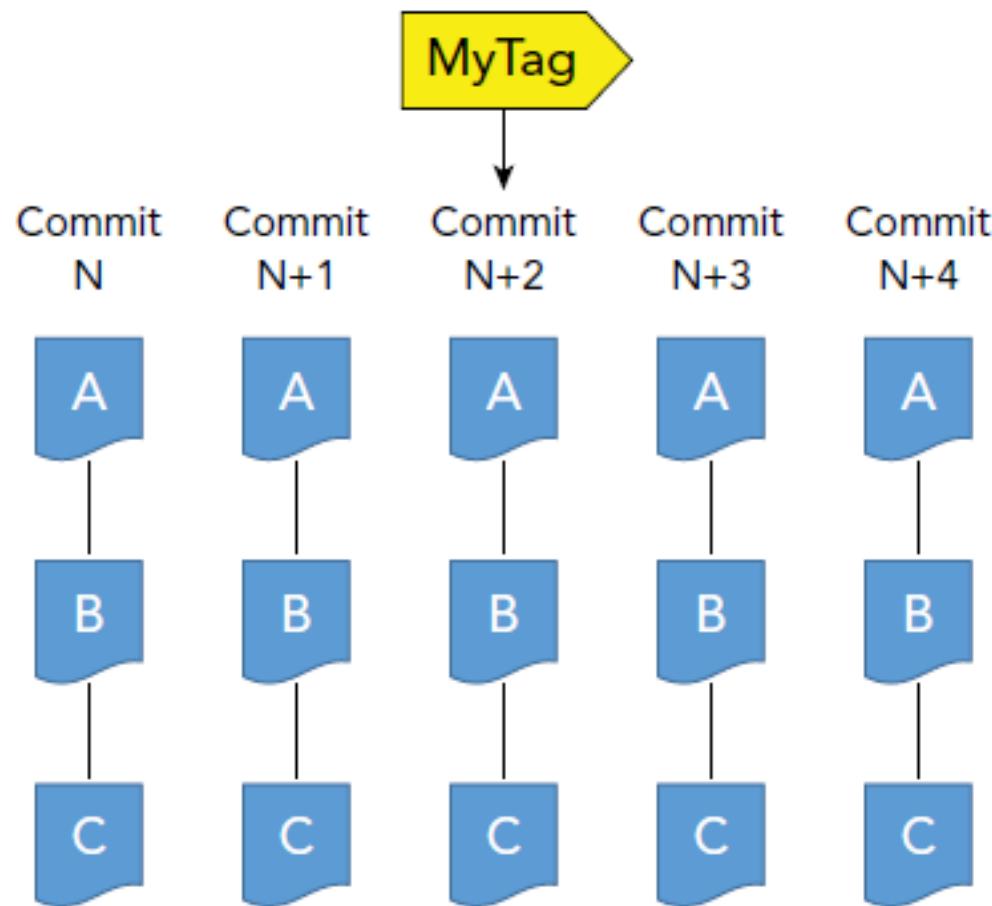
- **Git push Command**
 - It is used to upload local repository content to a remote repository.
 - Pushing is an act of transfer commits from your local repository to a remote repo.
 - It's the complement to git fetch, but whereas fetching imports commits to local branches on comparatively pushing exports commits to remote branches.
 - Remote branches are configured by using the git remote command. Pushing is capable of overwriting changes, and caution should be taken when pushing.
 - Git push command can be used as follows.
 - `Git push origin master`
 - This command sends the changes made on the master branch, to your remote repository.
- Syntax
- **\$ `git push [variable name] master`**



Tagging

- Like most VCSs, Git can tag specific points in a repository's history as being important.
- Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).
- Listing Your Tags
 - Listing the existing tags in Git is straightforward.
 - Just type `git tag` (with optional `-l` or `--list`):
 - `$ git tag`
 - `v1.0`
 - `v2.0`

Tagging





Create Tag

- Git supports two types of tags: lightweight and annotated.
- A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit.
- Annotated tags, however, are stored as full objects in the Git database.
- They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
- It's generally recommended that you create annotated tags so you can have all this information.
- But if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too



Annotated Tags

- Creating an annotated tag in Git is simple.
- The easiest way is to specify -a when you run the tag command:
- `$ git tag -a v1.4 -m "my version 1.4"`
- `$ git tag`
- `v0.1`
- `v1.3`
- `v1.4`



Lightweight Tags

- Another way to tag commits is with a lightweight tag.
- This is basically the commit checksum stored in a file — no other information is kept.
- To create a lightweight tag, don't supply any of the -a, -s, or -m options, just provide a tag name:
- `$ git tag v1.4-lw`
- `$ git tag`



Tags

```
F:\sapientws2022>git tag  
F:\sapientws2022>git tag v1.4-lw  
F:\sapientws2022>git tag  
v1.4-lw  
F:\sapientws2022>git show v1.4-lw  
commit 56579d6f1ede25916aa58c247b3c6554d192fbe8 (HEAD -> projects, tag: v1.4-lw, origin/projects)  
Author: eswaribala <parameswaribala@gmail.com>  
Date:   Tue Sep 6 22:15:58 2022 +0530  
  
        updated  
  
diff --git a/globalinsurancereactjs/globalinsuranceapi/default.log b/globalinsurancereactjs/globalinsuranceapi/default.log  
index 25785d0..d7f6fdb 100644  
--- a/globalinsurancereactjs/globalinsuranceapi/default.log  
+++ b/globalinsurancereactjs/globalinsuranceapi/default.log  
@@ -12,3 +12,5 @@  
 2022.09.06, 07:16:20.0165 UTC -> Button was clicked.  
 2022.09.06, 07:16:20.0377 UTC -> Button was clicked.  
 2022.09.06, 07:16:20.0583 UTC -> Button was clicked.  
+2022.09.06, 16:38:50.0310 UTC -> undefined  
+2022.09.06, 16:40:54.0453 UTC -> undefined  
diff --git a/globalinsurancereactjs/globalinsuranceapi/my-log2021.log b/globalinsurancereactjs/globalinsuranceapi/my-log2021.log  
index aa321de..718e617 100644  
--- a/globalinsurancereactjs/globalinsuranceapi/my-log2021.log  
+++ b/globalinsurancereactjs/globalinsuranceapi/my-log2021.log  
@@ -11,3 +11,5 @@  
 2022.09.06, 07:16:20.0165 UTC -> Button was clicked.  
 2022.09.06, 07:16:20.0378 UTC -> Button was clicked.  
 2022.09.06, 07:16:20.0583 UTC -> Button was clicked.
```



Logs

- git log --oneline --name-only
- git log --name-only -- web/src/main/webapp
- git log –oneline



Logs

```
F:\sapientws2022> git log --pretty=oneline
56579d6f1ede25916aa58c247b3c6554d192fbe8 (HEAD -> projects, tag: v1.4-lw, origin/projects) updated
d81acf51dd86edcf09fe1b4c63e596b0050e1291 updated
5d8c9a240daff181b3ee520fdf03c99648c46e6d updated
cf91ed53f5663c0c33cf025abbc4955cbee62d93 updated
e1ef1bb015483c55e30c1ec4e4f18e6eaf63b454 updated
d3c9212867d4eb49cce120587288ace7bddac119 updated
8b7ba3333a8fe0768bdccb52448316975122a33d updated
b1f7bfa6dbd7af8930e397492ea567ad63731adb updated
cff1727a5c44bda995a4a8cdc75fca0545115445 updated
a802ebc2e6d5734d63d76b8fecbc29279cb7ec27 updated
bdea930f4a517961fde7fdbb0a4c3873f7f25728 updated
f08511ced50aa7d96f89056245c9bfe4d8f2d745 Merge branch 'projects' of https://github.com/eswaribala/sapientjun2022 into projects
defa4ed45488717c1465489ebb68d4e3dbe6f7dc updated
ad6e215ad2d95534acf30dc18a84660e09eb8c20 Update application.properties
4401ebc904b96b7e5a114da0f8e3a601cceeeba3 Update application.properties
23e5cb8b07e56e3d6d601ccdf8f266bb14d33c859 updated
5ccba52e7767c6dca04e88ff154a8993fbc55643 updated
ec73221eb527f937d0310a200999db4f24c7cd0c updated
da9d6eefdc5d56d98c4b15220c2732b287c29c72 updated
4f1f685d9136d00601ffbd9ea04fad69a3cbebf0 updated
6368cea8c9388d9e481aa8265cfb05f52be4e23e updated
9deadadf98f184eb76959f68add6e10a031820d9 updated
269c24aedc21955d7d5b68bbb5ae6410de4b39f4 updated
6660cb191ed6e2aae07c24f12d9fc8973806c86f updated
ff201c92b81a5032fcf34f8937332c17aa6ef29c updated
98bae47847d036942b1874c824ce94be08f3a7c9 updated
9a48ab06183dd0bc5715fbce89c395d2de644fdd updated
905a1b5f1ae2ad85ee31348047625005d38af5f9 updated
b8ada114cb58fb6a0283d0f07754b92293c9cbb1 updated
67550db466364c1e6b4feb3d08b0204e71ab9fc6 updated
374d49646b8501b2c57ee218ba6e7d1fd6f7cced updated
4054184fdcba2db917d4636967e16a369fa241ea updated
```



Forgot to tag – Tag later

```
e16b6ae9a19b6970038864bb4bc91510ac306511 updated
```

```
F:\sapientws2022>git tag v1.2 ad6e215ad2d95534acf30dc18a84660e09eb8c20
```

```
F:\sapientws2022>git tag  
v1.2  
v1.4-lw
```

```
F:\sapientws2022>
```



Delete Tag

- `git tag -d v1.4-lw`



Git Fork

- A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project.
- Great use of using forks to propose changes for bug fixes. To resolve an issue for a bug that you found, you can:
 - Fork the repository.
 - Make the fix.
 - Forward a pull request to the project owner.



Git Fork

- A fork is a rough copy of a repository. Forking a repository allows you to freely test and debug with changes without affecting the original project.
- One of the excessive use of forking is to propose changes for bug fixing.
- To resolve an issue for a bug that you found, you can:
- Fork the repository.
- Make the fix.
- Forward a pull request to the project owner.



When to use Fork

- Generally, forking a repository allows us to experiment on the project without affecting the original project.
Following are the reasons for forking the repository:
 - Propose changes to someone else's project.
 - Use an existing project as a starting point.



How to Fork a Repository?

- The forking and branching are excellent ways to contribute to an open-source project.
- These two features of Git allows the enhanced collaboration on the projects.
- It is a straight-forward process. Steps for forking the repository are as follows:
 - Login to the GitHub account.
 - Navigate to the GitHub repository which you want to fork.
 - Click the Fork button on the upper right side of the repository's page.



Fork vs. Clone

- Fork is used to create a server-side copy and clone is used to create a local copy of the repository.
- There is no particular command for forking the repository; instead, it is a service provided by third-party Git service like GitHub.
- Comparatively, git clone is a command-line utility that is used to create a local copy of the project.
- Generally, people working on the same project clone the repository and the external contributors fork the repository.



Git command line

- **Git pull command**
- Pull command is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory.
- **Syntax**
- **\$ git pull URL**



Git command line

- **Git Branch Command**
 - This command lists all the branches available in the repository.
- Syntax
- **\$ git branch**
- **Git Merge Command**
 - This command is used to merge the specified branch's history into the current branch.
- Syntax
- **\$ git merge BranchName**



Git command line

- **Git log Command**
 - This command is used to check the commit history.
- Syntax
- **\$ git log**
- **Cheatsheet**



GIT Commands

Activities Terminal ▾

linuxvmimage

File Edit View Search Terminal Help

```
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 51f3604a22a6d636a03a5720fa53c5bdd520f243 (HEAD -> master, origin/master, devopslab)
Author: eswaribala <parameeswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500
```

initial commit

```
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch master
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$
```



GIT Commands

```
Activities Terminal ▾ Sun 10:01
linuxvmimages@ubuntu1804: ~/devopslab
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 51f3604a22a6d636a03a5720fa53c5bdd520f243 (HEAD -> master, origin/master, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch master
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java
linuxvmimages@ubuntu1804:~/devopslab$ cp employee.java employeev1.java
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java  employeev1.java
linuxvmimages@ubuntu1804:~/devopslab$ git add .
linuxvmimages@ubuntu1804:~/devopslab$ git commit -m "updated"
[master b60c874] updated
 1 file changed, 9 insertions(+)
 create mode 100644 employeev1.java
linuxvmimages@ubuntu1804:~/devopslab$ git push origin master
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 237 bytes | 237.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/eswaribala/skilldevops2022.git
 51f3604..b60c874  master -> master
linuxvmimages@ubuntu1804:~/devopslab$
```

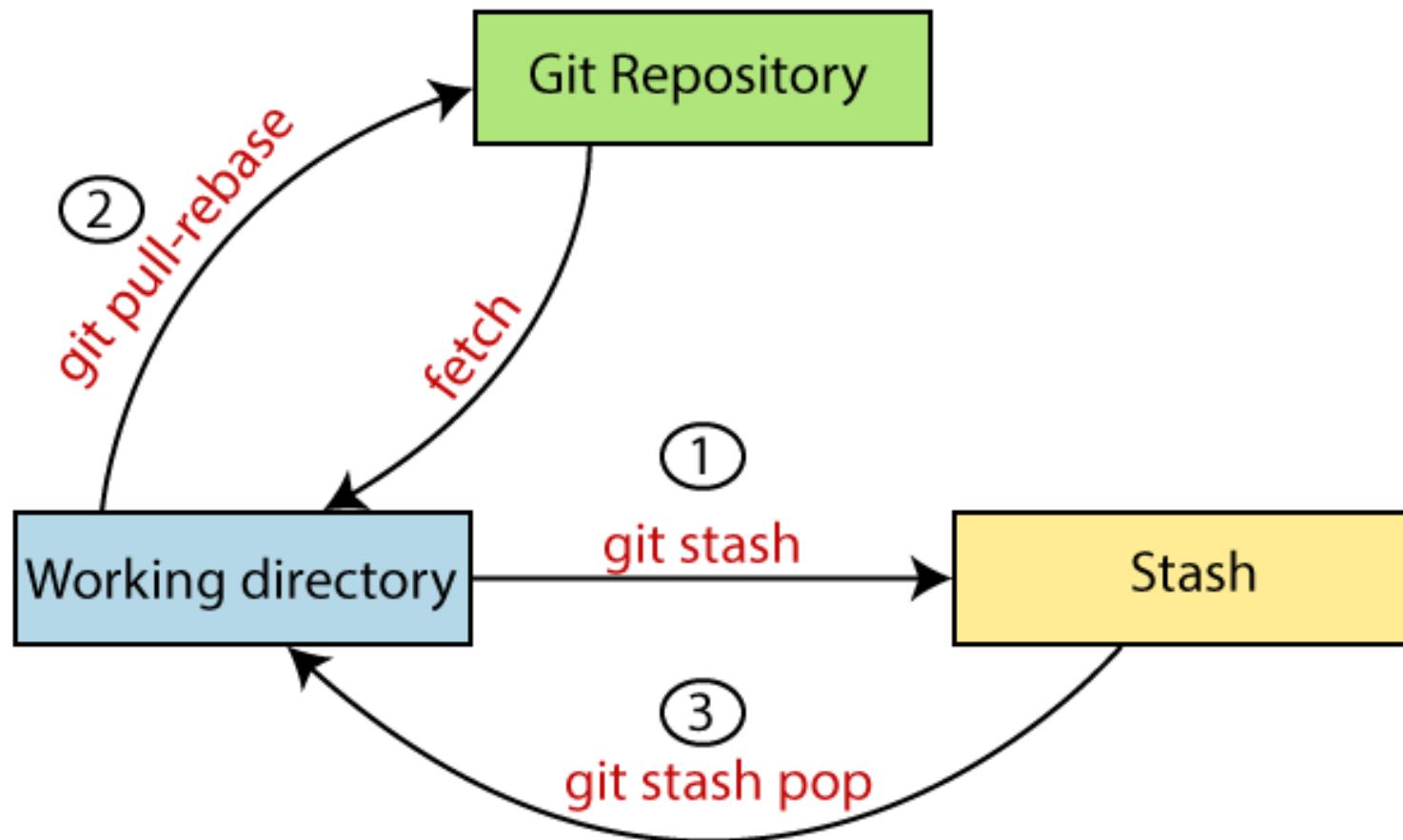


Git Stash

- Sometimes we want to switch the branches, but we are working on an incomplete part of our current project.
- We don't want to make a commit of half-done work. Git stashing allows you to do so.
- The git stash command enables us to switch branches without committing the current branch.



Git Stash





Git Stash

- Some useful options are given below:
- Git stash
- Git stash save
- Git stash list
- Git stash apply
- Git stash changes
- Git stash pop
- Git stash drop
- Git stash clear
- Git stash branch



Git Stash

```
Microsoft Windows [Version 10.0.22000.856]
(c) Microsoft Corporation. All rights reserved.
```

```
F:\sapientws2022>git status
Refresh index: 100% (3442/3442), done.
On branch projects
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
    modified:   customer/pom.xml

no changes added to commit (use "git add" and/or "git commit -a")

F:\sapientws2022>■
```



Git Stash

```
Microsoft Windows [Version 10.0.22000.856]
(c) Microsoft Corporation. All rights reserved.
```

```
F:\sapientws2022>git status
Refresh index: 100% (3442/3442), done.
On branch projects
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
    modified:   customer/pom.xml

no changes added to commit (use "git add" and/or "git commit -a")
```

```
F:\sapientws2022>git stash
Saved working directory and index state WIP on projects: 56579d6 updated
```

```
F:\sapientws2022>git status
On branch projects
nothing to commit, working tree clean
```

```
F:\sapientws2022>
```



Git Ignore

- In Git, the term "ignore" is used to specify intentionally untracked files that Git should ignore.
- It doesn't affect the Files that already tracked by Git.
- Sometimes we don't want to send the files to Git service like GitHub.
- We can specify files in Git to ignore.



Git Ignore

- The file system of Git is classified into three categories:
- Tracked:
 - Tracked files are such files that are previously staged or committed.
- Untracked:
 - Untracked files are such files that are not previously staged or committed.
- Ignored:
 - Ignored files are such files that are explicitly ignored by git. We have to tell git to ignore such files.

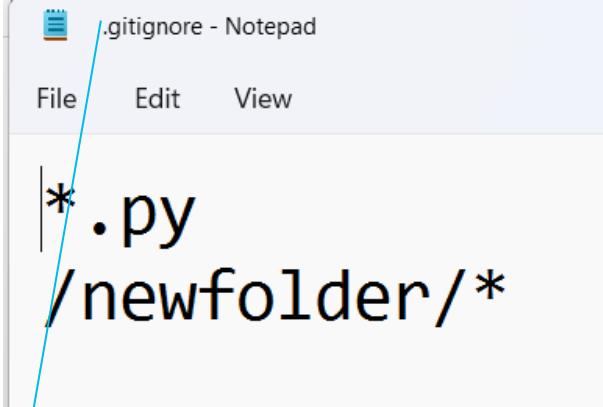


Git Ignore

- Git ignore files is a file that can be any file or a folder that contains all the files that we want to ignore.
- The developers ignore files that are not necessary to execute the project.
- Git itself creates many system-generated ignored files.
- Usually, these files are hidden files.
- There are several ways to specify the ignore files.
- The ignored files can be tracked on a `.gitignore` file that is placed on the root folder of the repository.
- No explicit command is used to ignore the file.



How to Ignore Files Manually



A screenshot of a Notepad window titled ".gitignore - Notepad". The window shows the following content:

```
*.*py
/newfolder/*
```

```
F:\sapientws2022>git status
On branch projects
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
    modified:   customer/pom.xml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore

no changes added to commit (use "git add" and/or "git commit -a")
```



How to Ignore Files Manually

```
F:\sapientws2022>git add .gitignore
```

```
F:\sapientws2022> git commit -m "ignored directory created."  
[projects 61a31c2] ignored directory created.  
1 file changed, 2 insertions(+)  
create mode 100644 .gitignore
```

git ls-files --ignore --exclude-standard



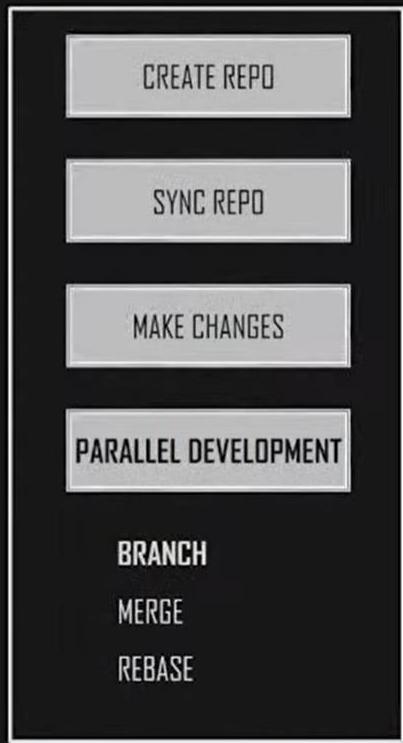
How to Ignore Files Manually

```
F:\sapientws2022>git status --ignored
On branch projects
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   cucumberdemo/src/test/java/com/amex/cucumberdemo/RunCucumberTest.java
    modified:   customer/pom.xml

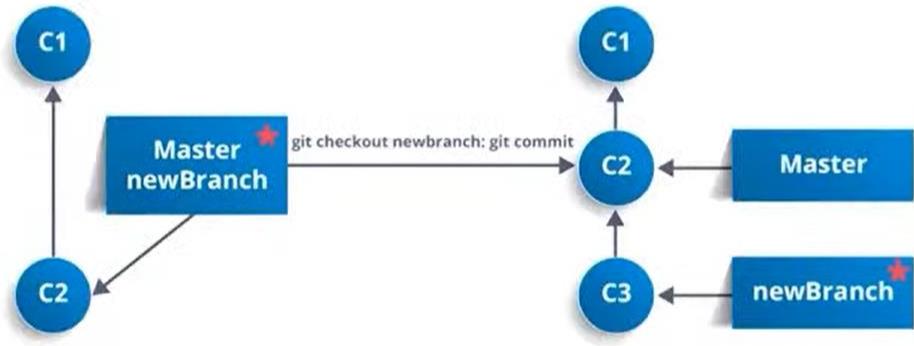
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
    HDFSSample/.idea/workspace.xml
    ReadWriteHDFSExample/ReadWriteHDFSExample/.idea/workspace.xml
    claimkafkaconsumerapi/.idea/
    claimkafkaconsumerapi/HELP.md
    claimkafkaconsumerapi/claimkafkaconsumerapi.iml
    claimkafkaconsumerapi/target/
    cucumberdemo/.idea/workspace.xml
    cucumberdemo/target/
    customer/.classpath
    customer/.project
    customer/.settings/
    customer/HELP.md
    customer/target/
    elkmisdemo/.classpath
    elkmisdemo/.project
    elkmisdemo/.settings/
    elkmisdemo/HELP.md
    elkmisdemo/target/
    gloalinsurancespringmvcapp/.idea/compiler.xml
    gloalinsurancespringmvcapp/.idea/jarRepositories.xml
    gloalinsurancespringmvcapp/.idea/libraries/
```

Parallel Development

GIT OPERATIONS



Branching is an integral part of any Version Control System. Unlike other VCS, Git **does not** create a copy of existing files for new branch. It points to snapshot of the changes you have made in the system



```
git branch <branch_name>
```



Parallel Development

Activities Terminal ▾ linuxvmimag

File Edit View Search Terminal Help

```
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch master
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ git checkout devopslab
Switched to branch 'devopslab'
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch devopslab
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ git branch
* devopslab
  master
linuxvmimages@ubuntu1804:~/devopslab$
```



Parallel Development

Activities Terminal ➔ linuxvm

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git branch
* master
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch master
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ git checkout devopslab
Switched to branch 'devopslab'
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch devopslab
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ git branch
* devopslab
  master
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 51f3604a22a6d636a03a5720fa53c5bdd520f243 (HEAD -> devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java
linuxvmimages@ubuntu1804:~/devopslab$ cp employee.java employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git add .
linuxvmimages@ubuntu1804:~/devopslab$ git commit -m "updated"
[devopslab 81c97e8] updated
 1 file changed, 9 insertions(+)
 create mode 100644 employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git push origin devopslab
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 239 bytes | 239.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'devopslab' on GitHub by visiting:
remote:     https://github.com/eswaribala/skilldevops2022/pull/new/devopslab
remote:
To https://github.com/eswaribala/skilldevops2022.git
 * [new branch]      devopslab -> devopslab
linuxvmimages@ubuntu1804:~/devopslab$
```



Parallel Development

Activities Terminal ▾

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeenv1.java
linuxvmimages@ubuntu1804:~/devopslab$ git checkout devopslab
Switched to branch 'devopslab'
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeenv2.java
linuxvmimages@ubuntu1804:~/devopslab$
```



Parallel Development

```
Activities Terminal ▾
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeenv1.java
linuxvmimages@ubuntu1804:~/devopslab$ git checkout devopslab
Switched to branch 'devopslab'
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeenv2.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
* devopslab
  master
linuxvmimages@ubuntu1804:~/devopslab$ git checkout master
Switched to branch 'master'
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeenv1.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
linuxvmimages@ubuntu1804:~/devopslab$ git merge devopslab
Merge made by the 'recursive' strategy.
 employeenv2.java | 9 ++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 employeenv2.java
linuxvmimages@ubuntu1804:~/devopslab$ █
```



Parallel Development

Activities Terminal ▾ Sun 10:46
linuxvmimages@ubuntu1804: ~/devopslab\$

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
* devopslab
  Master
linuxvmimages@ubuntu1804:~/devopslab$ git checkout master
Switched to branch 'master'
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeev1.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* Master
linuxvmimages@ubuntu1804:~/devopslab$ git merge devopslab
Merge made by the 'recursive' strategy.
 employeev2.java | 9 ++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch master
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit d1d130502208923358fd799f0744d0decdf9f8108 (HEAD -> master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a (origin/master)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 09:58:20 2022 -0500

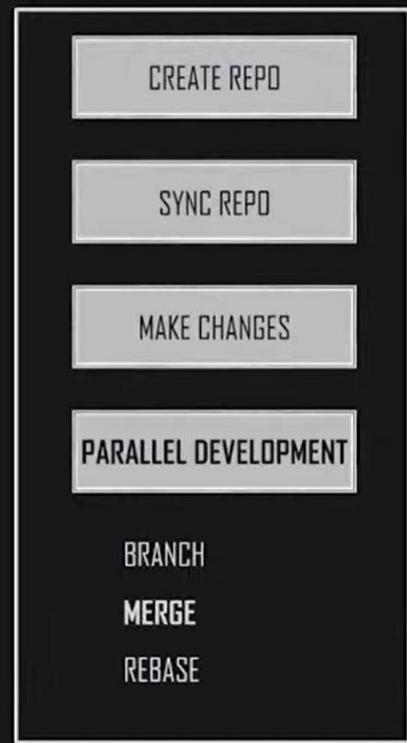
    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

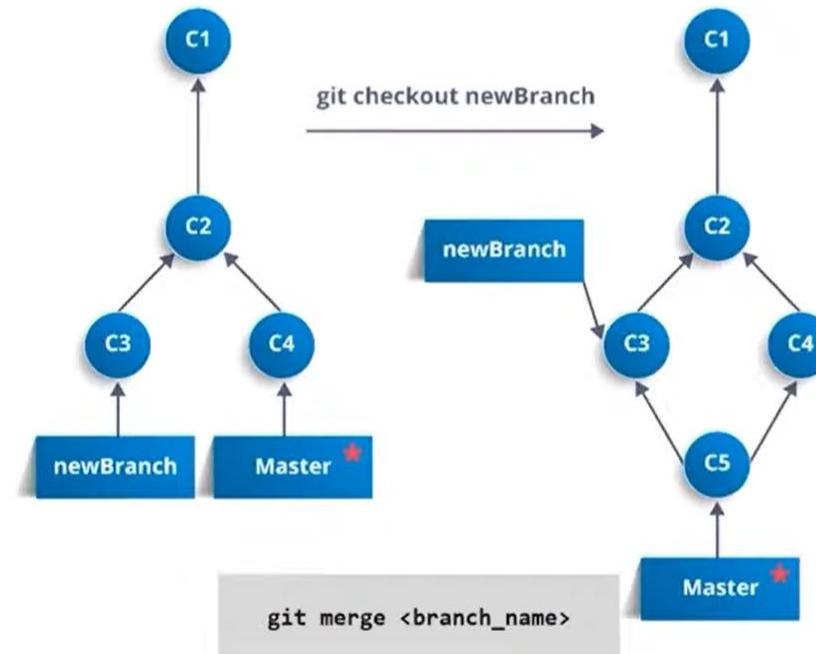
    initial commit
linuxvmimages@ubuntu1804:~/devopslab$
```

Parallel Development

GIT OPERATIONS



Merge integrates the changes made in different branches to one single branch





Parallel Development

Activities Terminal ▾



```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeev1.java
employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
linuxvmimages@ubuntu1804:~/devopslab$ git ls-files
employee.java
employeev1.java
employeev2.java
linuxvmimages@ubuntu1804:~/devopslab$ git push origin master
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 285 bytes | 285.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To https://github.com/eswaribala/skilldevops2022.git
  b60c874..d1d1305  master -> master
linuxvmimages@ubuntu1804:~/devopslab$
```



Parallel Development

- Delete Branch
- git branch -d test
- Rename Branch
- git branch -m test testing

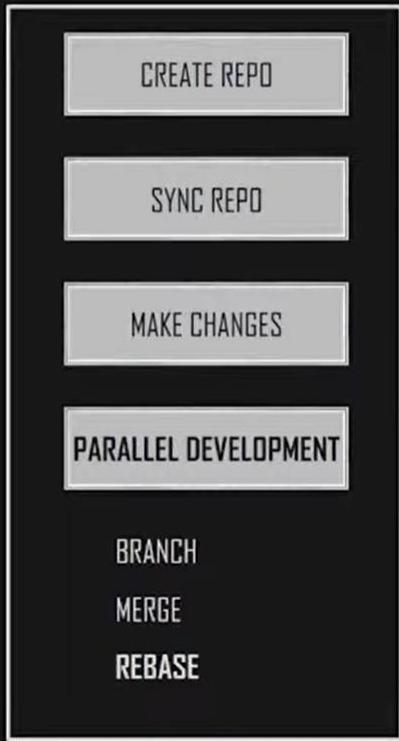


Rebase

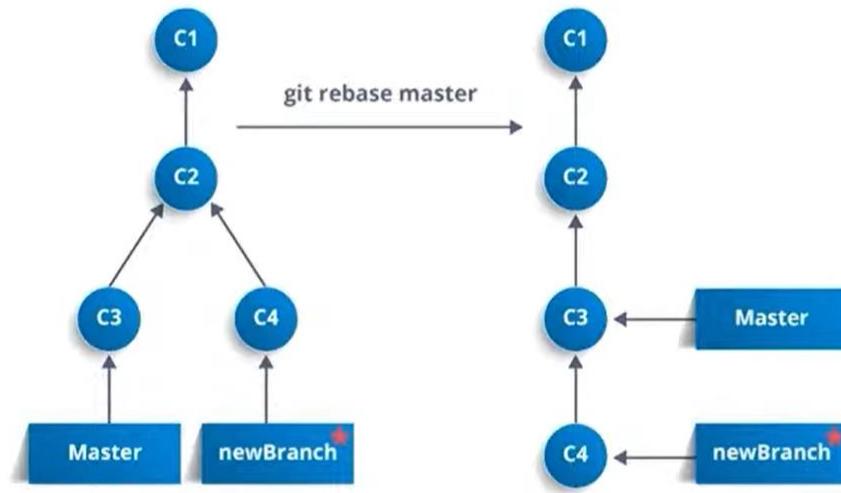
- A rebase – We can combine a commit or series of commits to a new commit.
- It is like merging that moves changes from one branch to another.
- Rebasing allows you to rewrite the history of a Git repository.
- When you run a rebase operation, it merges the entire history of two branches into one.
- This will create brand new commits for each commit in another branch inside the branch you are rebasing.
- Rebasing is just like changing the base of a branch from one commit to another.
- This will change your repository's history to make it look like you created a branch from another commit.

Rebase

GIT OPERATIONS

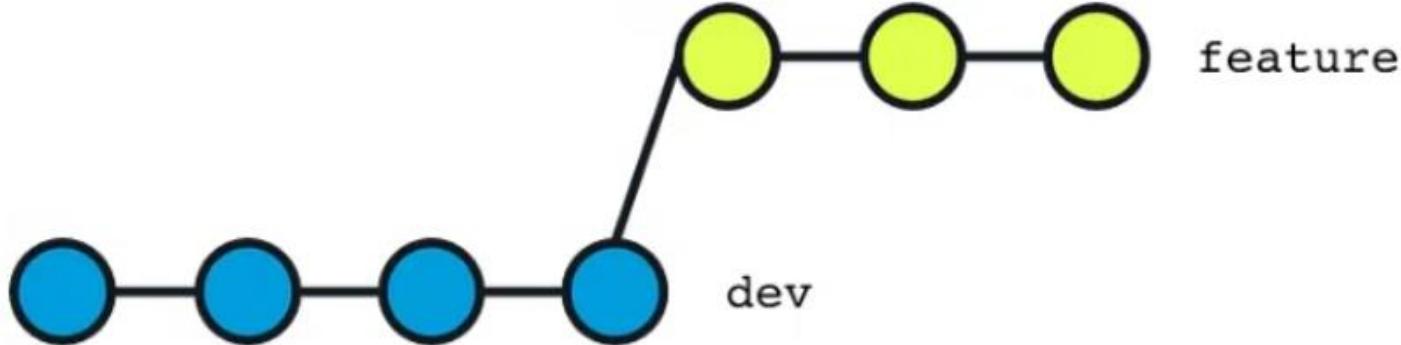
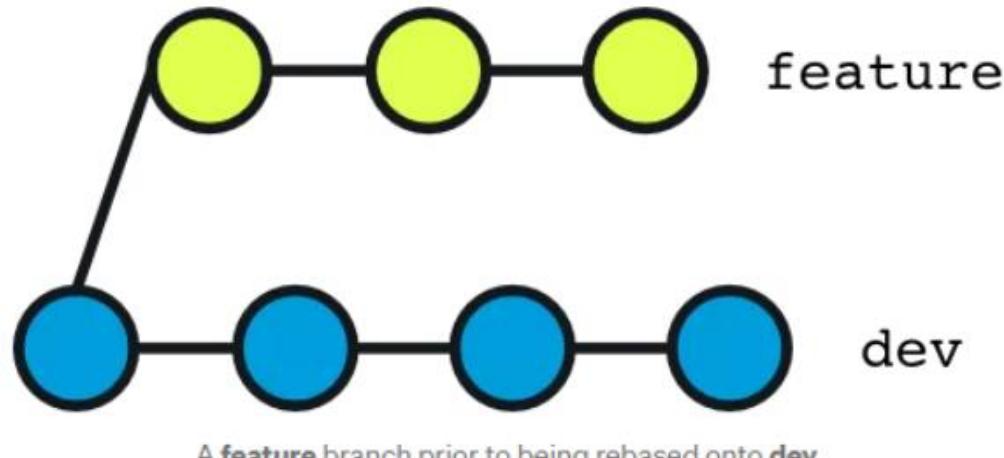


Rebase is used when changes made in one branch needs to be reflected in another branch



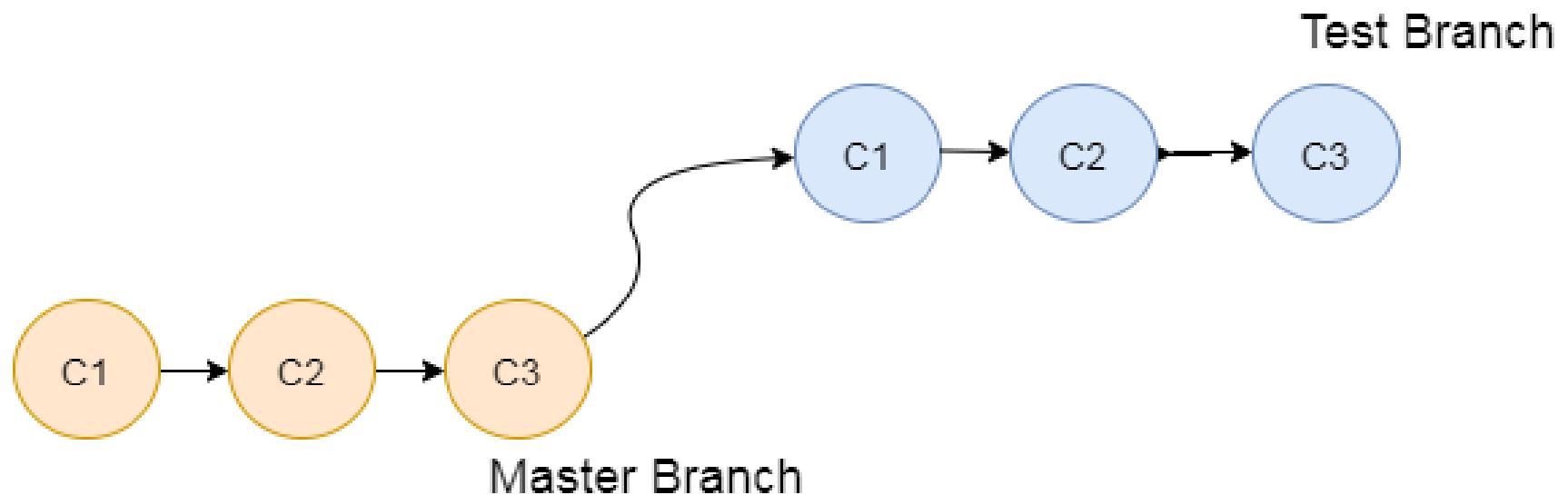
git rebase master

Rebase





Rebase





Rebase

```
/o-----o---o---o----- branch  
--o-o--A--o---o---o---o---o-o-o--- master
```

When you rebase you can move it like this:

```
/o-----o---o---o----- branch  
--o-o--A--o---o---o---o---o-o-o--- master
```



Rebase

Activities Terminal ▾ Sun 11:44
linuxvmimages@ubuntu1804:

```
File Edit View Search Terminal Help
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ git branch rebasebranch
linuxvmimages@ubuntu1804:~/devopslab$ git checkout rebasebranch
Switched to branch 'rebasebranch'
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit d1d130502208923358fd799f0744d0decdf9f8108 (HEAD -> rebasebranch, origin/master, master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date: Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ git status
On branch rebasebranch
nothing to commit, working tree clean
linuxvmimages@ubuntu1804:~/devopslab$ █
```



Rebase

Activities Terminal ▾ Sun 11:48
linuxvmimages@ubuntu1804: ~/devopslab

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ sudo nano testrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git add .
linuxvmimages@ubuntu1804:~/devopslab$ git commit -m "Updated rebase branch"
[rebasebranch 4750d5a] Updated rebase branch
 1 file changed, 1 insertion(+)
 create mode 100644 testrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 4750d5a1f44f486b8a56a210623c9ce4b2f985dd (HEAD -> rebasebranch)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 11:48:12 2022 -0500

    Updated rebase branch

commit d1d130502208923358fd799f0744d0dec9f8108 (origin/master, master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ █
```



Rebase

Activities Terminal ▾

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ sudo nano testrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git add .
linuxvmimages@ubuntu1804:~/devopslab$ git commit -m "Updated rebase branch"
[rebasebranch 4750d5a] Updated rebase branch
 1 file changed, 1 insertion(+)
 create mode 100644 testrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 4750d5a1f44f486b8a56a210623c9ce4b2f985dd (HEAD -> rebasebranch)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 11:48:12 2022 -0500

    Updated rebase branch

commit d1d130502208923358fd799f0744d0dec9f8108 (origin/master, master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$ git log --oneline
4750d5a (HEAD -> rebasebranch) Updated rebase branch
d1d1305 (origin/master, master) Merge branch 'devopslab'
81c97e8 (origin/devopslab, devopslab) updated
b60c874 updated
51f3604 initial commit
linuxvmimages@ubuntu1804:~/devopslab$ █
```



Rebase

Activities Terminal ▾ Sun 11:55
linuxvmimages@ubuntu1804: ~/devopslab

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git checkout master
Switched to branch 'master'
linuxvmimages@ubuntu1804:~/devopslab$ sudo nano masterrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git add .
linuxvmimages@ubuntu1804:~/devopslab$ git commit -m "master updated with masterrebase.txt"
> "
[master 20350b2] master updated with masterrebase.txt
 1 file changed, 1 insertion(+)
  create mode 100644 masterrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git logs
git: 'logs' is not a git command. See 'git --help'.

The most similar command is
  log
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 20350b2da64ac90a13d17b7a8023d9ec1997dd6e (HEAD -> master)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 11:54:49 2022 -0500

    master updated with masterrebase.txt

commit d1d130502208923358fd799f0744d0dec9f8108 (origin/master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$
```



Rebase

Activities Terminal ▾ Sun 11:56
linuxvmimages@ubuntu1804: ~/devopslab

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git checkout rebasebranch
Switched to branch 'rebasebranch'
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
* rebasebranch
linuxvmimages@ubuntu1804:~/devopslab$ git log
commit 4750d5a1f44f486b8a56a210623c9ce4b2f985dd (HEAD -> rebasebranch)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 11:48:12 2022 -0500

    Updated rebase branch

commit d1d130502208923358fd799f0744d0dec9f8108 (origin/master)
Merge: b60c874 81c97e8
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:40:31 2022 -0500

    Merge branch 'devopslab'

commit 81c97e86f9b264eb7b1e5187b3d5150cf69ce737 (origin/devopslab, devopslab)
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 10:21:34 2022 -0500

    updated

commit b60c874fa8d7c4801040d299e44e58ee5bc5de2a
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 09:58:20 2022 -0500

    updated

commit 51f3604a22a6d636a03a5720fa53c5bdd520f243
Author: eswaribala <parameswaribala@gmail.com>
Date:   Sun Feb 20 02:14:56 2022 -0500

    initial commit
linuxvmimages@ubuntu1804:~/devopslab$
```



Rebase

Activities Terminal ▾ Sun 12:00

linuxvmimages@ubuntu1804: ~/devopslab

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~/devopslab$ git log --oneline rebasebranch
49c6af1 (HEAD -> rebasebranch) Updated rebase branch
20350b2 (master) master updated with masterrebase.txt
d1d1305 (origin/master) Merge branch 'devopslab'
81c97e8 (origin/devopslab, devopslab) updated
b60c874 updated
51f3604 initial commit
linuxvmimages@ubuntu1804:~/devopslab$ git rebase master
Current branch rebasebranch is up to date.
linuxvmimages@ubuntu1804:~/devopslab$ █
```

The screenshot shows a Linux desktop environment with a dark theme. On the left is a vertical dock containing icons for various applications: a browser (Firefox), email (Thunderbird), file manager (Nautilus), system settings (Gnome Control Center), and others. The main window is a terminal titled "Terminal". The title bar also shows "Activities" and the current time "Sun 12:00". The terminal window has a dark background and displays a command-line session. The user runs "git log --oneline rebasebranch" which shows a history of commits on the "rebasebranch". Then, the user runs "git rebase master" and checks the status with "git log" again, confirming that the branch is now up-to-date with the "master" branch. The bottom right corner of the terminal window contains a small terminal icon.



Git Rebase vs. Git Merge

Git Merge	Git Rebase
Merging creates a final commit at merging.	Git rebase does not create any commit at rebasing.
It merges all commits as a single commit.	It creates a linear track of commits.
It creates a graphical history that might be a bit complex to understand.	It creates a linear history that can be easily understood.
It is safe to merge two branches.	Git "rebase" deals with the severe operation.
Merging can be performed on both public and private branches.	It is the wrong choice to use rebasing on public branches.
Merging integrates the content of the feature branch with the master branch. So, the master branch is changed, and feature branch history remains consistence.	Rebasing of the master branch may affect the feature branch.
Merging preserves history.	Rebasing rewrites history.
Git merge presents all conflicts at once.	Git rebase presents conflicts one by one.



Git Rebase vs. Git Merge

- Both git rebase and git merge have their ideal use cases.
- The git merge command is best used if you want to merge a branch into another branch.
- The merge command creates a merge commit which ties together the histories of the projects.



Git Rebase vs. Git Merge

- The rebase command is best used if you need to rewrite the history of a project.
- Rebase will create new commits for each commit in the master branch.
- This will ensure that the final version of your repository contains all the history from both branches.
- You should use the git merge command if you are making changes to a public repository.
- This is because the git merge command does not rewrite history.
- It maintains a record that is forward-looking.



GIT Merge Conflicts

Activities Terminal ▾ Mon 11:22

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~$ ls
Desktop devopslabday11 Downloads Pictures Templates
devopslab Documents examples.desktop Public Videos
devopslabday1 dotnetinvs Music snap
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java employeev2.java testrebase.txt
employeev1.java masterrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ git init user1
Initialized empty Git repository in /home/linuxvmimages/devopslab/user1/.git/
linuxvmimages@ubuntu1804:~/devopslab$ git init user2
Initialized empty Git repository in /home/linuxvmimages/devopslab/user2/.git/
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java employeev1.java employeev2.java masterrebase.txt testrebase.txt user1 user2
linuxvmimages@ubuntu1804:~/devopslab$ cd user1
linuxvmimages@ubuntu1804:~/devopslab/user1$ ls
linuxvmimages@ubuntu1804:~/devopslab/user1$ git pull https://github.com/eswaribala/skilldevops2022.git
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 3), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
From https://github.com/eswaribala/skilldevops2022
 * branch HEAD      -> FETCH_HEAD
linuxvmimages@ubuntu1804:~/devopslab/user1$ ls
employee.java employeev1.java employeev2.java
linuxvmimages@ubuntu1804:~/devopslab/user1$ sudo nano employee.java
linuxvmimages@ubuntu1804:~/devopslab/user1$ git add .
linuxvmimages@ubuntu1804:~/devopslab/user1$ git commit -m "Employee updated"
[master cc90ef4] Employee updated
1 file changed, 2 insertions(+)
linuxvmimages@ubuntu1804:~/devopslab/user1$ git push origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
linuxvmimages@ubuntu1804:~/devopslab/user1$ git push --force origin master
fatal: 'origin' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
linuxvmimages@ubuntu1804:~/devopslab/user1$ git remote add origin https://github.com/eswaribala/skilldevops2022.git
linuxvmimages@ubuntu1804:~/devopslab/user1$ git push --force origin master
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
Counting objects: 3, done.
```

English (India)
English (India)

To switch input methods, press Windows key + space.



GIT Merge Conflicts

Ubuntu_18.04.5_VM_LinuxVMImages - VMware Workstation 16 Player (Non-commercial use only)

Activities Terminal Mon 11:35

linuxvmmimages@ubuntu1804: ~/devopslab/user1

```
File Edit View Search Terminal Help
linuxvmmimages@ubuntu1804:~/devopslab/user1$ git push -f origin master
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 329 bytes | 329.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/eswaribala/skilldevops2022.git
  cc90ef4..f958372 master -> master
linuxvmmimages@ubuntu1804:~/devopslab/user2$ sudo nano employee.java
linuxvmmimages@ubuntu1804:~/devopslab/user2$ git add .
linuxvmmimages@ubuntu1804:~/devopslab/user2$ git commit -m "Employee updated by user2 now"
[master f2cc27c] Employee updated by user2 now
 1 file changed, 1 insertion(+), 1 deletion(-)
linuxvmmimages@ubuntu1804:~/devopslab/user2$ cd ..
linuxvmmimages@ubuntu1804:~/devopslab$ cd user1
linuxvmmimages@ubuntu1804:~/devopslab/user1$ ls
employee.java employee1.java employee2.java
linuxvmmimages@ubuntu1804:~/devopslab/user1$ sudo nano employee.java
linuxvmmimages@ubuntu1804:~/devopslab/user1$ git add .
linuxvmmimages@ubuntu1804:~/devopslab/user1$ git commit -m "Employee updated by user1 now"
[master 1d8c6ac] Employee updated by user1 now
 1 file changed, 1 insertion(+), 1 deletion(-)
linuxvmmimages@ubuntu1804:~/devopslab/user1$ git push origin master
Username for 'https://github.com': eswaribala
Password for 'https://eswaribala@github.com':
To https://github.com/eswaribala/skilldevops2022.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'https://github.com/eswaribala/skilldevops2022.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
linuxvmmimages@ubuntu1804:~/devopslab/user1$ git pull https://github.com/eswaribala/skilldevops2022.git
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/eswaribala/skilldevops2022
 * branch      HEAD    -> FETCH_HEAD
Auto-merging employee.java
CONFLICT (content): Merge conflict in employee.java
Automatic merge failed; fix conflicts and then commit the result.
```

Player Activities Terminal Mon 11:35

linuxvmmimages@ubuntu1804:~/devopslab/user1

22:05 21/02/2022 19



GIT Merge Conflicts

Ubuntu_18.04.5_VM_LinuxVMImages - VMware Workstation 16 Player (Non-commercial use only)

Player Terminal Mon 12:05 linuxvmimages@ubuntu1804: ~/devopslab/user1

Activities

File Edit View Search Terminal Help

```
package com.skillwise.models
@Data
#modified to create conflict by user1

class Customer{
    private long customerId;
    private String customerName;
}
```

```
package com.skillwise.models
@Data
#modified to create conflict

class Customer{
    private long customerId;
    private String customerName;
}
```

```
package com.skillwise.models
@Data
#modified to create conflict by user2

class Customer{
    private long customerId;
    Private String customerName;
}
```

./employee_LOCAL_91392.java 1,1 All ./employee_BASE_91392.java 1,1 All ./employee_REMOTE_91392.java 1,1 All

```
package com.skillwise.models
@Data
<<<<< HEAD
#modified to create conflict by user1
=====
#modified to create conflict by user2
>>>> f958372cfcc63b0fb40d3f8d7fac03d2cf39b8178
class Customer{

    private long customerId;
    private String customerName;

}
~
~
~
~
~
```

employee.java

English (India)
English (India)

To switch input methods, press Windows key + space.

1,1 All

22:35 21/02/2022 19

Windows Start button, Taskbar icons (Calculator, File Explorer, Mail, Microsoft Edge, Google Chrome, File Explorer, Mail, Microsoft Edge, Microsoft Word, Microsoft Excel, Microsoft Powerpoint, Microsoft Teams, Microsoft Word, Microsoft Excel, Microsoft Word), Network icon, Battery icon, Volume icon, Date and time (22:35, 21/02/2022, 19).

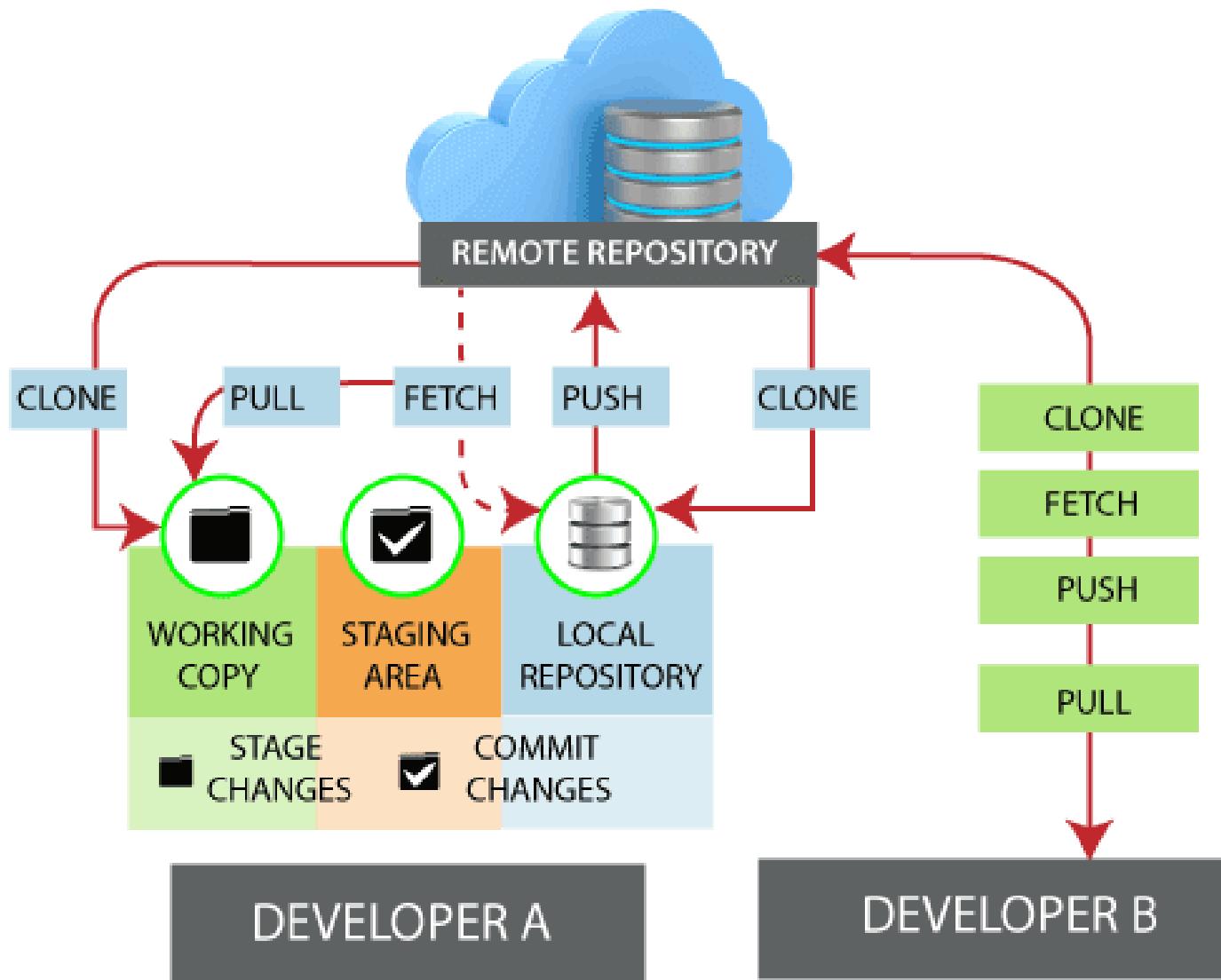


Git Remote

- In Git, the term remote is concerned with the remote repository.
- It is a shared repository that all team members use to exchange their changes.
- A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion, and more.
- In the case of a local repository, a remote typically does not provide a file tree of the project's current state; as an alternative, it only consists of the .git versioning data.



Git Remote





Git Remote

```
F:\sapientws2022> git remote  
origin
```

```
F:\sapientws2022> git remote --verbose  
origin https://github.com/eswaribala/sapientjun2022.git (fetch)  
origin https://github.com/eswaribala/sapientjun2022.git (push)
```

```
F:\sapientws2022>git remote show origin  
* remote origin  
  Fetch URL: https://github.com/eswaribala/sapientjun2022.git  
  Push URL: https://github.com/eswaribala/sapientjun2022.git  
  HEAD branch: master  
  Remote branches:  
    master tracked  
    projects tracked  
  Local refs configured for 'git push':  
    master pushes to master (local out of date)  
    projects pushes to projects (fast-forwardable)
```

```
F:\sapientws2022>
```



GIT Additional Commands

A few bonus commands you should keep in mind :

Archive your Repository

```
git archive master --format=zip -  
output=../name_of_file.zip
```





GIT Additional Commands

A few bonus commands you should keep in mind :

Archive your Repository

Bundle your Repository

```
git bundle create ../repo.bundle master
```



GIT Additional Commands

A few bonus commands you should keep in mind :

Archive your Repository

Bundle your Repository

Stash Uncommitted Changes

```
git stash  
git stash apply
```



THE NEED FOR GITHUB



- Developers need a web/cloud based code hosting platform
- Useful for version control
- Enables effective collaboration
- Download projects and files in one go
- Easy evaluation of each other's work

THE NEED FOR GITHUB



But what makes GitHub so popular?



Immensely powerful community



The largest shared repository



Easy version control



Secure cloud storage



Git Diff

- Git diff is a command-line utility. It's a multiuse Git command.
- When it is executed, it runs a diff function on Git data sources.
- These data sources can be files, branches, commits, and more.
- It is used to show changes between commits, commit, and working tree, etc.
- It compares the different versions of data sources.
- The version control system stands for working with a modified version of files.
- So, the diff command is a useful tool for working with Git.



Git Diff

Activities Terminal ▾

Mon 02:22

linuxvmimages@ubuntu1804: ~/devopslab

```
File Edit View Search Terminal Help
linuxvmimages@ubuntu1804:~$ ls
Desktop  Documents  examples.desktop  Pictures  snap      Videos
devopslab  Downloads  Music          Public    Templates
linuxvmimages@ubuntu1804:~$ cd devopslab
linuxvmimages@ubuntu1804:~/devopslab$ ls
employee.java  employeev1.java  employeev2.java  masterrebase.txt  testrebase.txt
linuxvmimages@ubuntu1804:~/devopslab$ sudo nano employee.java
linuxvmimages@ubuntu1804:~/devopslab$ git branch
  devopslab
* master
* rebasebranch
linuxvmimages@ubuntu1804:~/devopslab$ git diff
diff --git a/employee.java b/employee.java
index 4aae27a..66ff9a8 100644
--- a/employee.java
+++ b/employee.java
@@ -1,9 +1,12 @@
 package com.skillwise.models

+
@Data
class Customer{

    private long customerId;
    private String customerName;

    + private LocalDate dob;
    +
}
linuxvmimages@ubuntu1804:~/devopslab$
```



Git Diff

```
devopslab
master
* rebasebranch
linuxvmimages@ubuntu1804:~/devopslab$ git diff
diff --git a/employee.java b/employee.java
index 4aae27a..66ff9a8 100644
--- a/employee.java
+++ b/employee.java
@@ -1,9 +1,12 @@
 package com.skillwise.models

+
@Data
class Customer{

    private long customerId;
    private String customerName;
-
+ private LocalDate dob;
+
}

linuxvmimages@ubuntu1804:~/devopslab$ git diff --staged
linuxvmimages@ubuntu1804:~/devopslab$ git diff master devopslab
diff --git a/employeev1.java b/employeev1.java
deleted file mode 100644
index 4aae27a..00000000
--- a/employeev1.java
+++ /dev/null
@@ -1,9 +0,0 @@
-package com.skillwise.models
-
-@Data
-class Customer{
-
-    private long customerId;
-    private String customerName;
-
-}
diff --git a/masterrebase.txt b/masterrebase.txt
deleted file mode 100644
index 2ada78e..00000000
--- a/masterrebase.txt
+++ /dev/null
@@ -1 +0,0 @@
-log "Master rebase test done..."
linuxvmimages@ubuntu1804:~/devopslab$ ^C
linuxvmimages@ubuntu1804:~/devopslab$ █
```



GIT Rename Branch

- Start by switching to the local branch which you want to rename:
 - git checkout <old_name>
 - Rename the local branch by typing:
 - git branch -m <new_name>
- At this point, you have renamed the local branch.
- If you've already pushed the <old_name> branch to the remote repository , perform the next steps to rename the remote branch.
 - Push the <new_name> local branch and reset the upstream branch:
 - git push origin -u <new_name>
 - Delete the <old_name> remote branch:
 - git push origin --delete <old_name>



Branch and Tag

- In git, you can use tags to keep track of reference points during the development phase.
- So instead of people or accounts, you tag your code.
- This way you can reference it easily. See, not that different.
- To create a new tag execute we use:
- `git tag <tagname>`



Branch and Tag

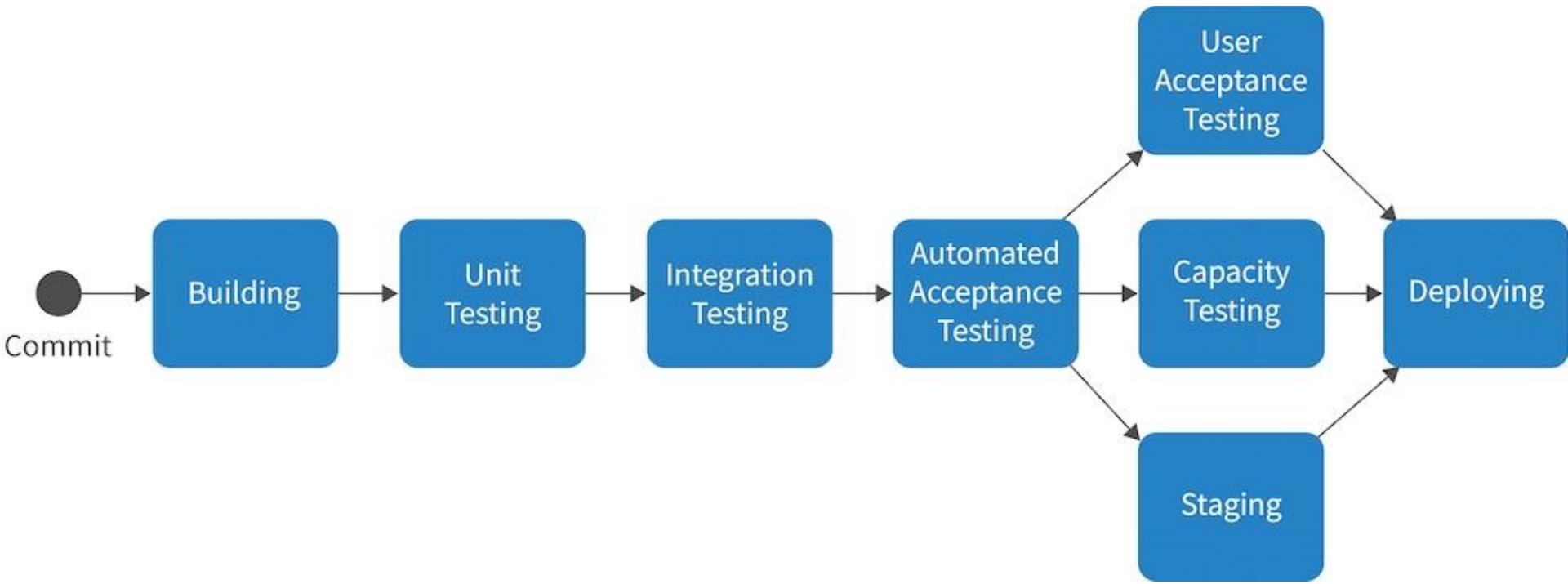
- Tag old commit
- git tag -a v1.2
15027957951b64cf874c3557a0f3547bd83b3ff6
- Retag
- git tag -a -f v1.4
15027957951b64cf874c3557a0f3547bd83b3ff6



Branch and Tag

- To checkout a specific git tag meaning to go to that specific code in that point of development we use:
- `$ git checkout tags/<tag> -b <branch>`

Continuous Integration

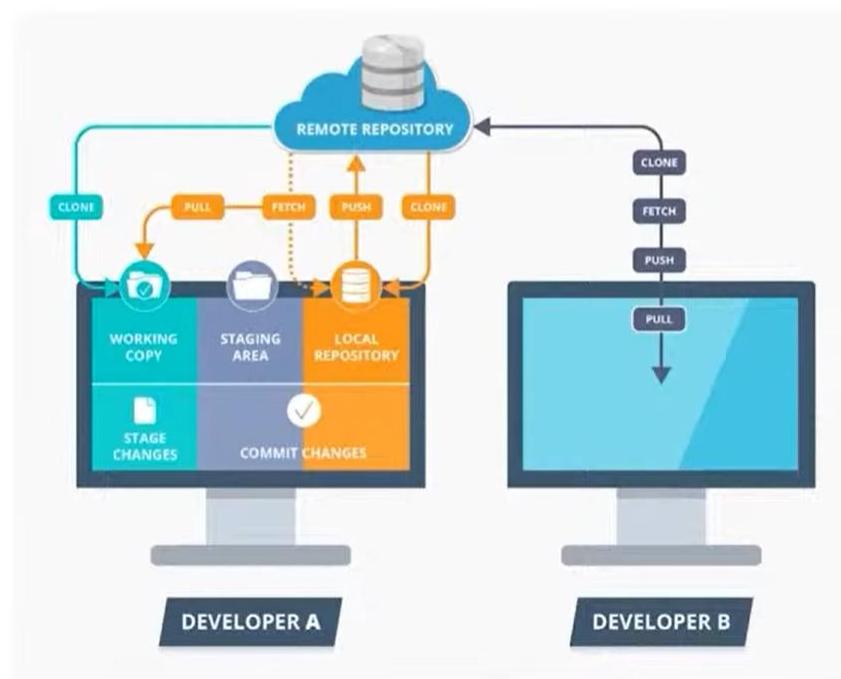




GIT Workflow

BUT HOW
DOES IT
WORK
?

Following is the basic workflow of Git:

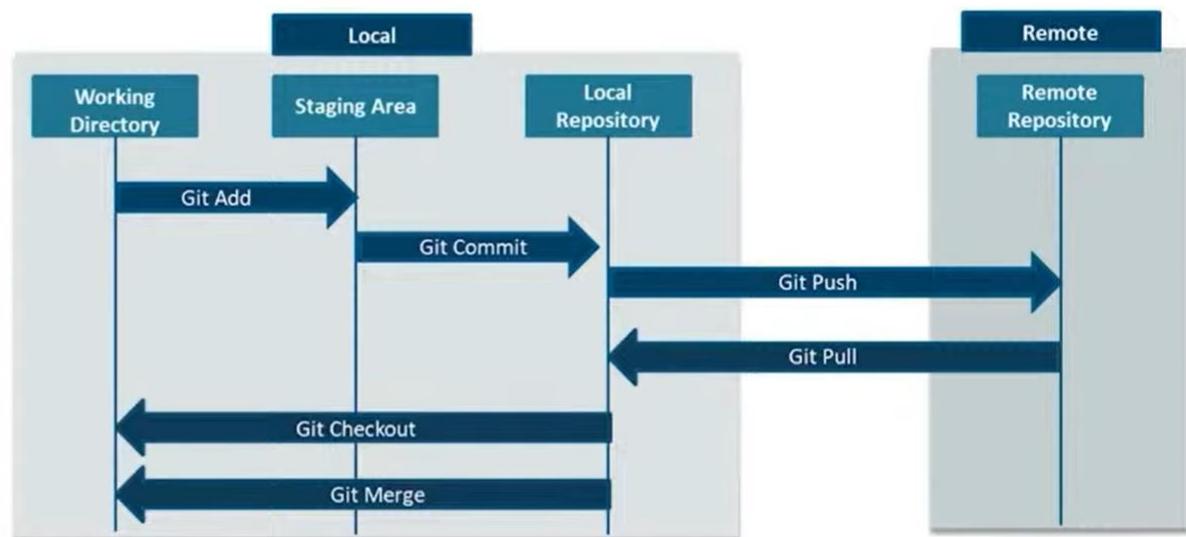




GIT Workflow

BUT HOW
DOES IT
WORK
?

Following is the basic workflow of Git:





Git hub Actions

- GitHub Actions is an event-driven automation platform that allows you to run a series of commands after a specified event has occurred.
- For example, when a commit is made to your staging branch, and we want to build, test and then deploy the changes to our staging environment.
- With Actions, we can automate tasks within our development lifecycle, all within GitHub.
- A common use case for Actions is automated continuous integration and deployment, and we may ask if we need to know yet another CI/CD tool, or which is better.
- GitHub Actions is much more than a CI/CD tool.



Git hub Actions Key Concepts

- The root concept in GitHub Actions is a workflow, or a series of automated procedures.
- In practice, a workflow is like a pipeline and allows developers to configure a series of stages that can be executed each time a specific event is triggered.
- Every repository can have any number of workflows.
- GitHub Actions is a CI/CD tool that is incorporated directly into every GitHub repository, utilizing text-based configuration files that reside directly within the repository.



Git hub Actions Key Concepts

COMPONENT	DESCRIPTION
Job	<p><i>A set of steps that are executed on the same runner</i></p> <p>By default, if a workflow has more than one job, the jobs are executed in parallel, but jobs can be configured to run in series by declaring that one job depends on another. If job <i>B</i> depends on job <i>A</i>, job <i>B</i> will only execute if job <i>A</i> completes successfully.</p>
Step	<p><i>A task that is composed of one or more shell commands or actions</i></p> <p>All steps from a job are executed on the same runner, and therefore, can share data with one another.</p>
Action	<p><i>A prepackaged set of procedures that can be executed within a step</i></p> <p>There are numerous actions already available through the GitHub community that perform common tasks, such as checking out code or uploading artifacts.</p>
Event	<p><i>A stimulus that triggers the execution of a workflow</i></p> <p>One of the most common events is a user checking in code to a repository.</p>
Runner	<p><i>A server that executes jobs on a specific Operating System (OS) or platform</i></p> <p>Runners can either be hosted by GitHub or on standalone servers.</p>

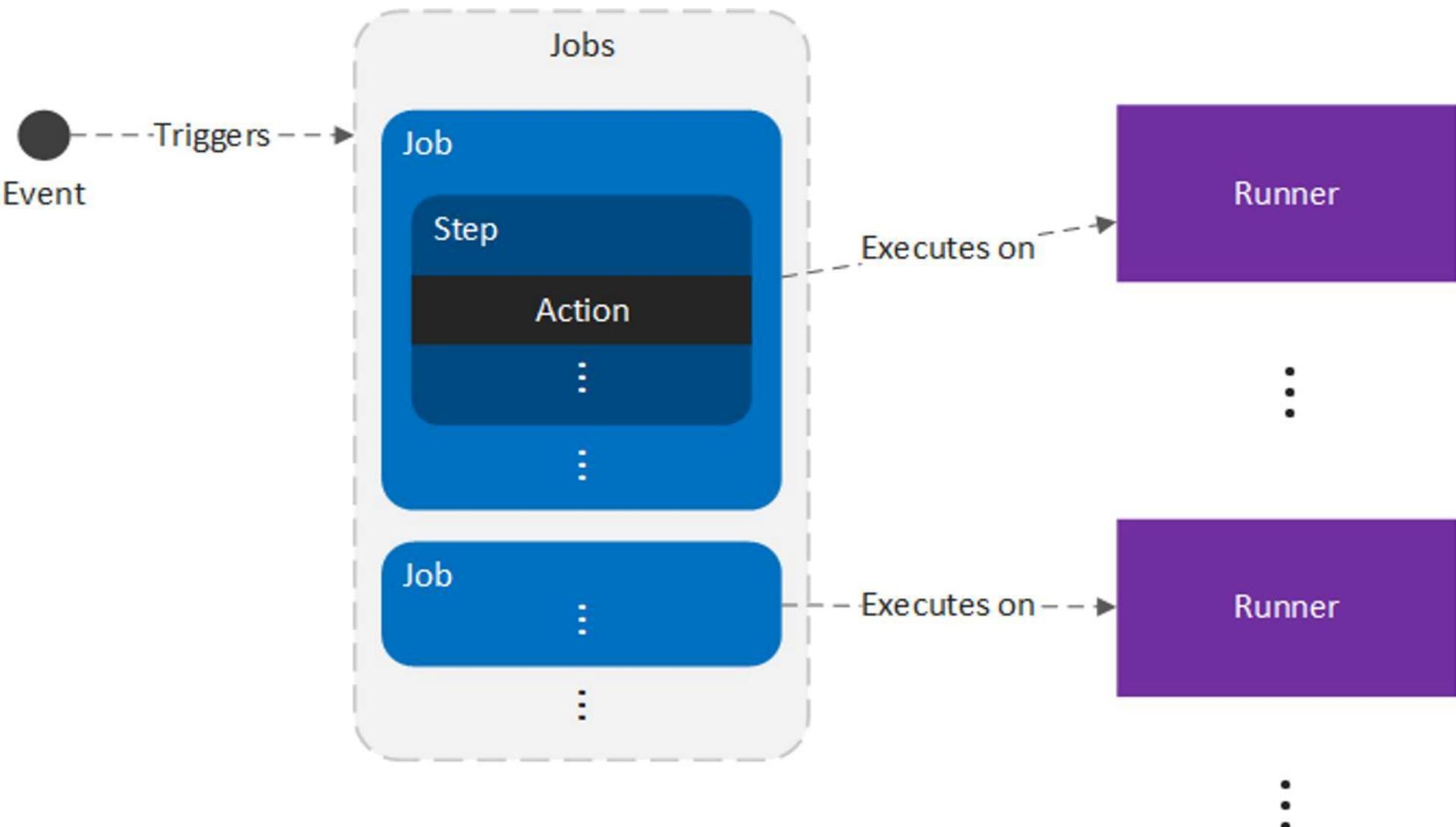


Git hub Actions Key Concepts





Git hub Actions Key Concepts





Git hub Actions Key Concepts

- In practice, workflows are more generalized than a CD pipeline, but they are closely related:
- Workflows = pipelines
- Jobs = stages
- Steps = the series of procedures that make up a stage



Git Workflow

- GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows us to automate your build, test, and deployment pipeline.
- We can create workflows that build and test every pull request to your repository or deploy merged pull requests to production.
- GitHub Actions goes beyond just DevOps and lets us run workflows when other events happen in your repository.
- For example, we can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in our repository.

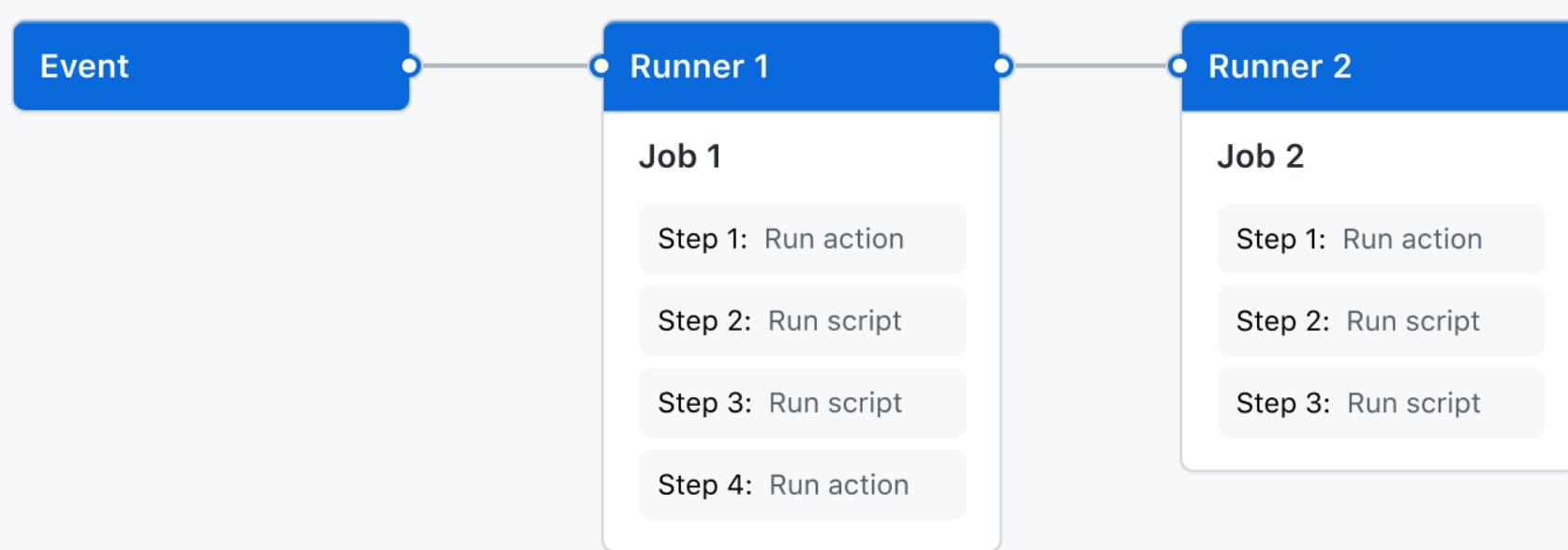


The components of GitHub Actions

- We can configure a GitHub Actions workflow to be triggered when an event occurs in our repository, such as a pull request being opened, or an issue being created.
- Our workflow contains one or more jobs which can run in sequential order or in parallel.
- Each job will run inside its own virtual machine runner, or inside a container, and has one or more steps that either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.



Git hub Actions





Types of Actions

- We can build Docker container and JavaScript actions.
- Actions require a metadata file to define the inputs, outputs and main entry point for the action.
- The metadata filename must be either `action.yml` or `action.yaml`

Type	Operating system
Docker container	Linux
JavaScript	Linux, macOS, Windows
Composite Actions	Linux, macOS, Windows



Docker Container Actions

- Docker containers package the environment with the GitHub Actions code.
- This creates a more consistent and reliable unit of work because the consumer of the action does not need to worry about the tools or dependencies.



Java script Actions

- JavaScript actions can run directly on a runner machine and separate the action code from the environment used to run the code.
- Using a JavaScript action simplifies the action code and executes faster than a Docker container action.
- To ensure our JavaScript actions are compatible with all GitHub-hosted runners (Ubuntu, Windows, and macOS), the packaged JavaScript code we write should be pure JavaScript and not rely on other binaries.
- JavaScript actions run directly on the runner and use binaries that already exist in the runner image.



Fork the project – Action and Work Flow Demo

- <https://github.com/albanoj2/dzone-github-actions-refcard-example.git>



Fork the project – Action and Work Flow Demo

Screenshot of a browser showing a GitHub Actions workflow run.

The URL in the address bar is github.com/eswaribala/rpsboamsmay2022projects/runs/8302677743?check_suite_focus=true.

The GitHub repository is [eswaribala / rpsboamsmay2022projects](#) (Public).

The workflow is named **Create github-actions.yaml GitHub Actions Demo #1**.

The workflow status is **succeeded 24 seconds ago in 4s**.

The workflow steps are:

- > ✓ Set up job 1s
- > ✓ Run echo "🎉 The job was automatically triggered by a push event." 0s
- > ✓ Run echo "🐧 This job is now running on a Linux server hosted by GitHub!" 0s
- > ✓ Run echo "🌐 The name of your branch is refs/heads/master and your repository is eswaribala/rpsboamsmay2022projects." 0s
- > ✓ Check out repository code 1s
- > ✓ Run echo "💡 The eswaribala/rpsboamsmay2022projects repository has been cloned to the runner." 0s
- > ✓ Run echo "💻 The workflow is now ready to test your code on the runner." 0s
- > ✓ List files in the repository 0s

The GitHub Actions tab is selected in the navigation bar.

The browser taskbar shows various open tabs and system icons.



Fork the project – Action and Work Flow Demo

GitHub Actions · albanoj2/dzone-github-actions-refcard-example/actions

github.com/albanoj2/dzone-github-actions-refcard-example/actions

Insert title here Empire New Tab How to use Asserti... Browser Automatio... node.js - How can I... Freelancer-dev-810... Courses New Tab Airtel 4G Hotspot nt8F83 Tryit Editor v3.6

dzone-github-actions-example

All workflows

Showing runs from all workflows

Filter workflow runs

Event	Status	Branch	Actor
Trigger rerun	Success	main	albanoj2
Changed target directory.	Success	main	albanoj2
Added debug.	Failure	main	albanoj2
Added download of JAR file.	Failure	main	albanoj2
Added missing checkout step in deployment.	Failure	main	albanoj2
Added deployment GitHub Action.	Failure	main	albanoj2

13 workflow runs

Event Status Branch Actor

Trigger rerun Success main albanoj2 9 months ago 3m 0s ...
Changed target directory. Success main albanoj2 10 months ago 2m 32s ...
Added debug. Failure main albanoj2 10 months ago 1m 25s ...
Added download of JAR file. Failure main albanoj2 10 months ago 1m 44s ...
Added missing checkout step in deployment. Failure main albanoj2 10 months ago 1m 52s ...
Added deployment GitHub Action. Failure main albanoj2 10 months ago 1m 32s ...

96°F Partly sunny

15:02 12/09/2022

SVN Repository

- SVN repository is a collection of files and directories.
- These files and directories are bundled together in a particular database.
- SVN also records the complete history of all the modifications that have ever been made to these files.
- Generally, the SVN repository can be considered as a folder or directory on our computer.
- These repositories may contain a collection of different or similar types of files.
- An SVN repository typically stores all the files and directories of a single project or maybe a collection of the interrelated projects.

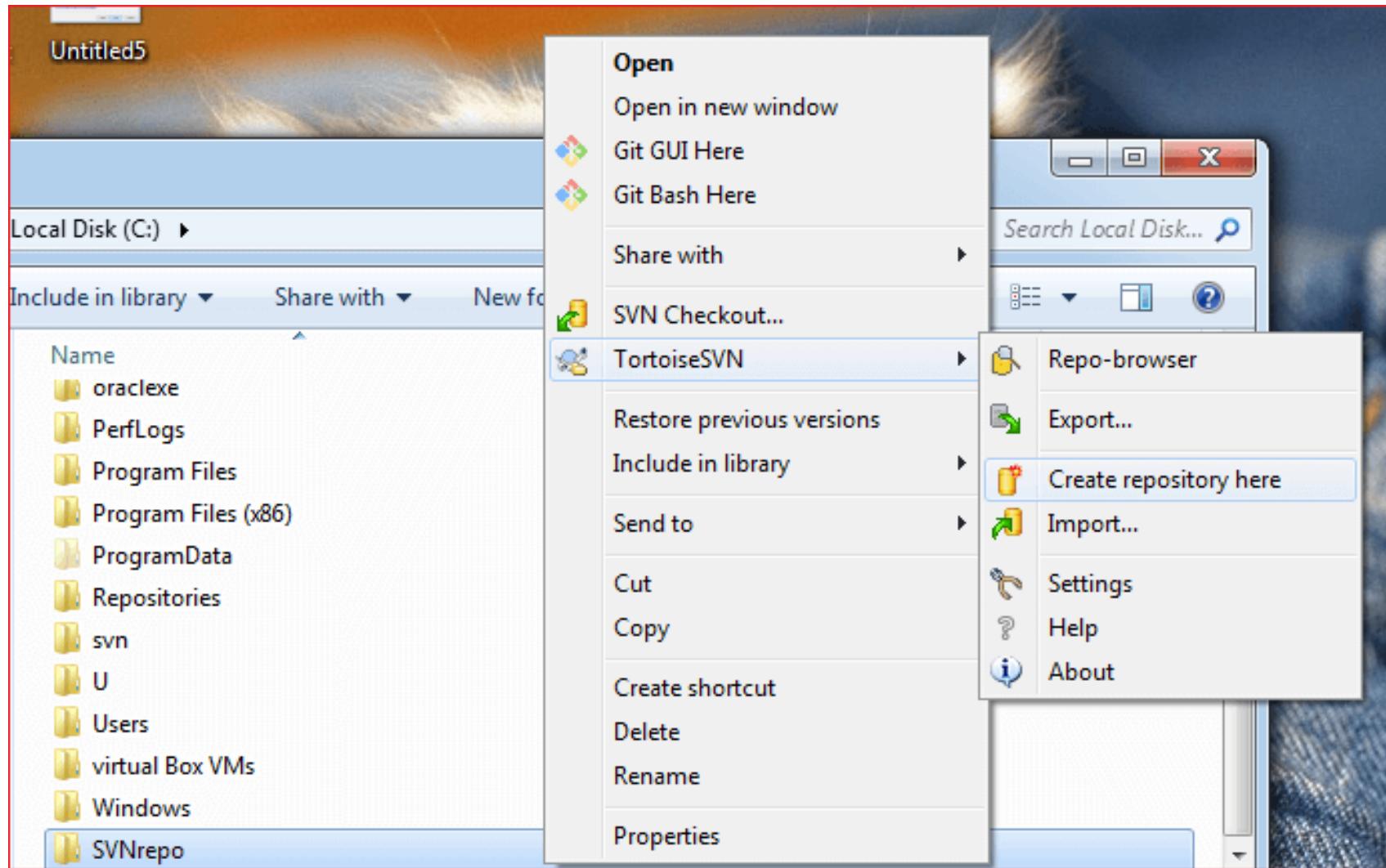


Create a repository using the command line

- To create an SVN repository using the command-line client, follow the below steps:
- Step1: Create an empty folder with the name svn (e.g., C:\svn\). However, we can create it with any name. It is used as the root of all our repositories.
- Step2: Create another folder newrepo inside C:\svn\.
- Step3: Open the command prompt, change directory to D:\svn\newrepo and type the below command:
- `svnadmin create --fs-type fsfs newrepo`

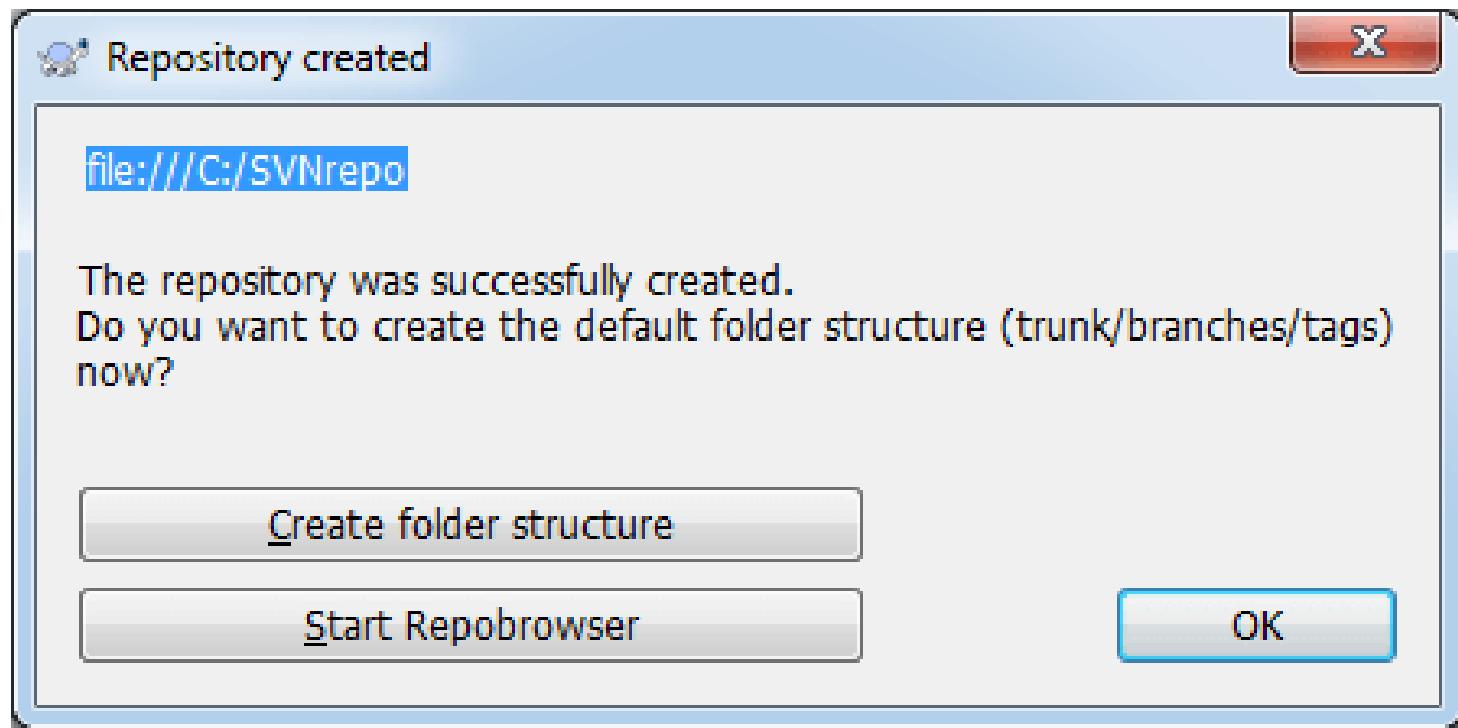


Create a Repository using TortoiseSVN



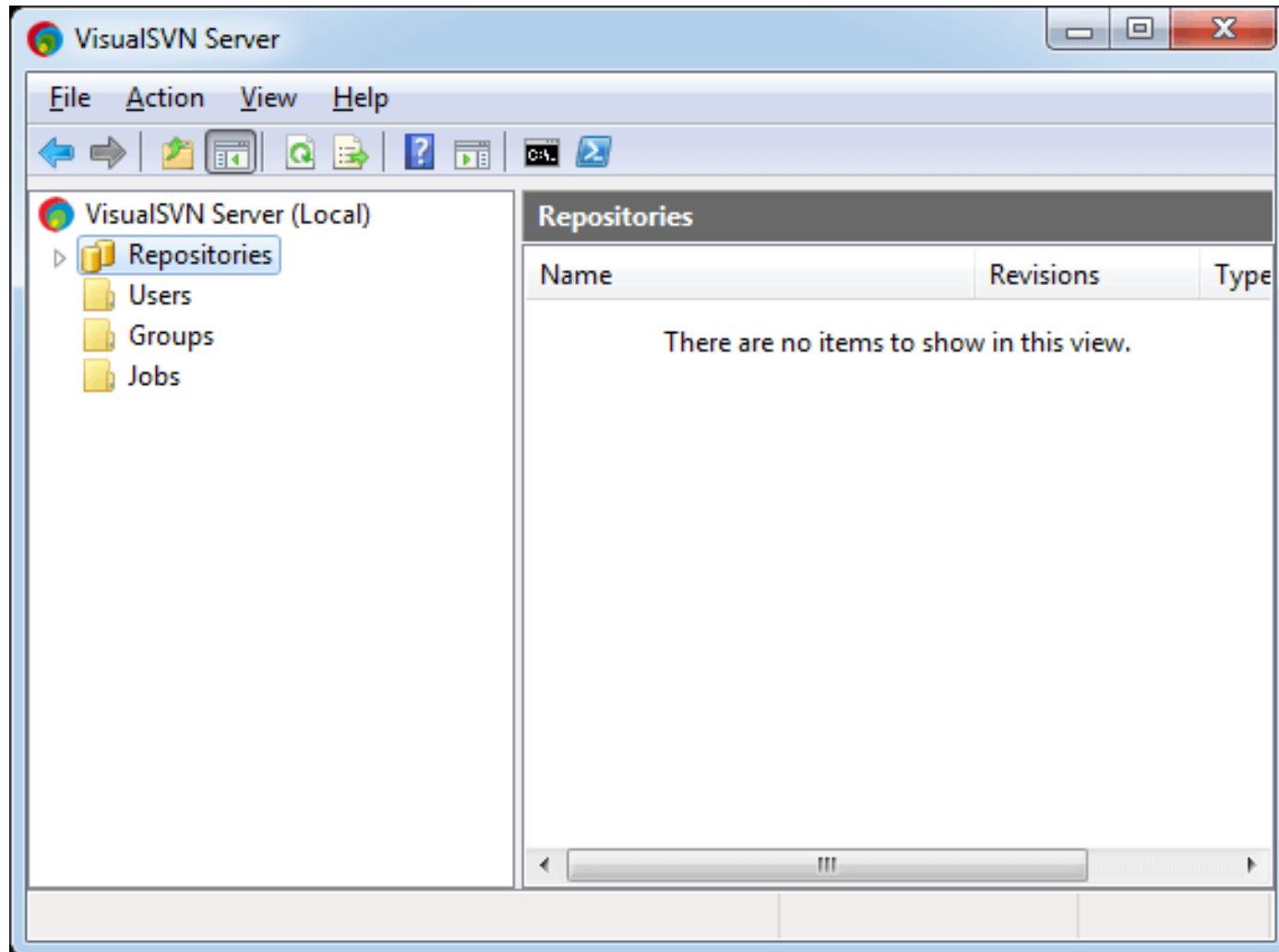


Create a Repository using TortoiseSVN





Create a Repository using TortoiseSVN

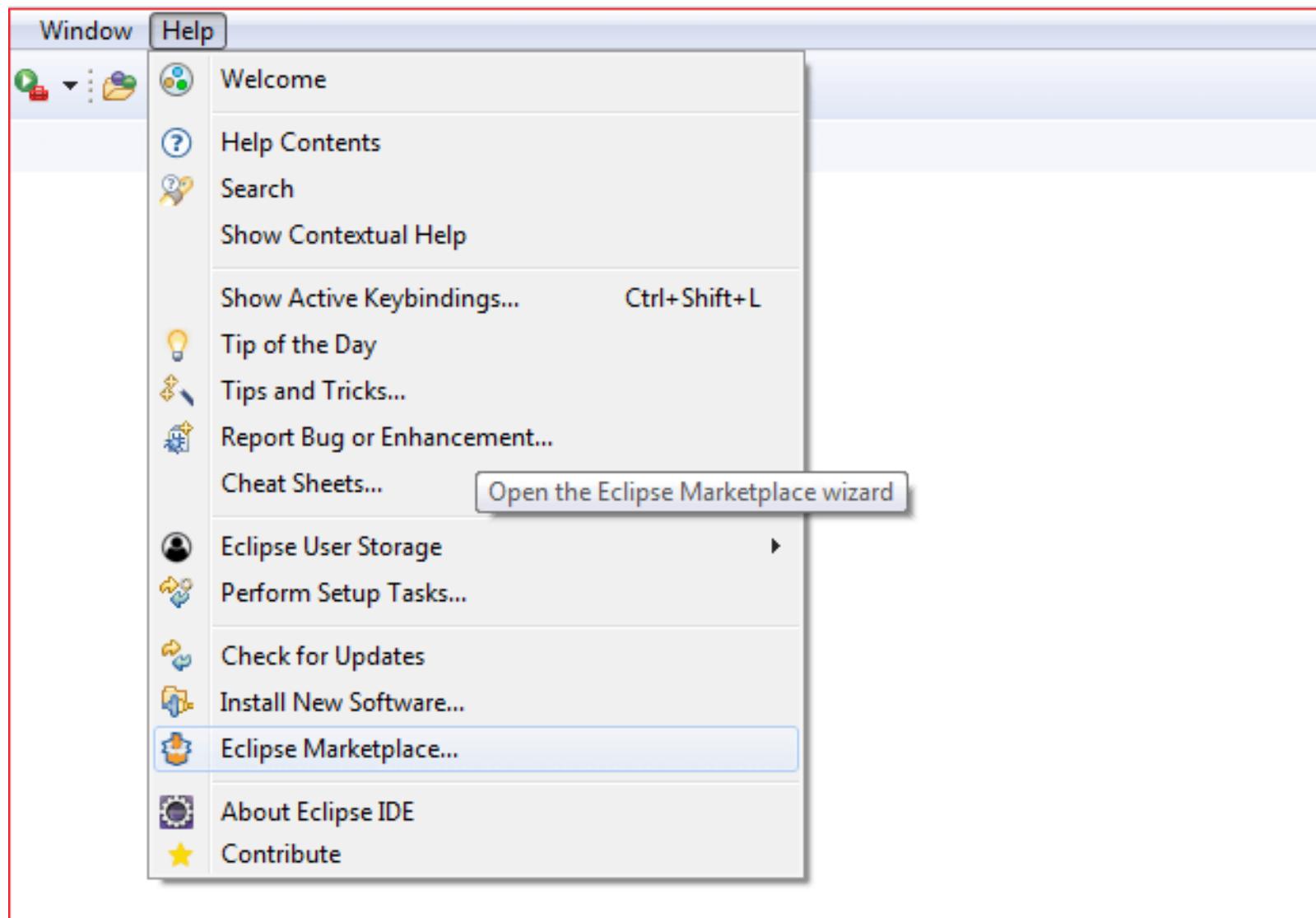




SVN Commands

- SVN Checkout Command
- SVN Add Command
- SVN Delete Command
- SVN Commit Command
- SVN Diff Command
- SVN Status Command
- SVN Log Command
- SVN Move Command
- SVN Rename Command
- SVN List Command
- SVN Update Command
- SVN Info Command
- SVN Merge Command

SVN for Eclipse





SVN for Eclipse

Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

A blue and orange icon depicting a document with a gear and a downward arrow.

Search Recent Popular Favorites Installed  Giving IoT an Edge

Find: Subclipse  All Markets  All Categories 

Subclipse 4.3.0

An Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE. Developed and maintained by Subversion core committers, Subclipse is... [more info](#)

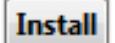
by [Subclipse Project](#), EPL
[svn](#) [subversion](#) [team provider](#) [mylyn](#) [alm](#) ...

 1895  Installs: 2.43M (24,965 last month) 

Teamscale Integration for Eclipse 5.6.0

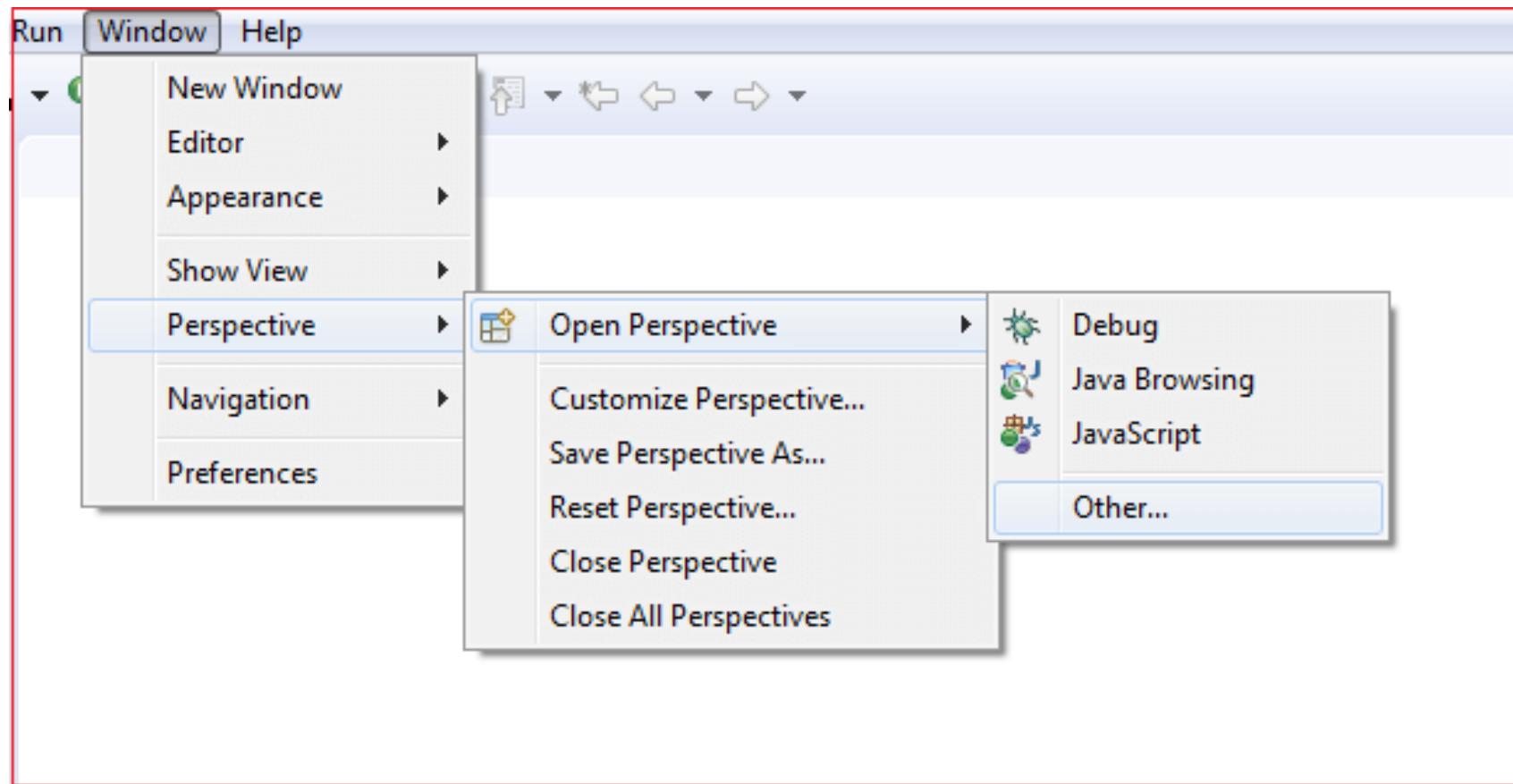
The Teamscale Integration for Eclipse allows for seamless browsing of quality defects found by the Teamscale software-quality analysis server. Teamscale analyzes... [more info](#)

by [CQSE GmbH](#), Apache 2.0
[teamscale](#) [clone detection](#) [Software Quality](#)

 4  Installs: 587 (18 last month) 

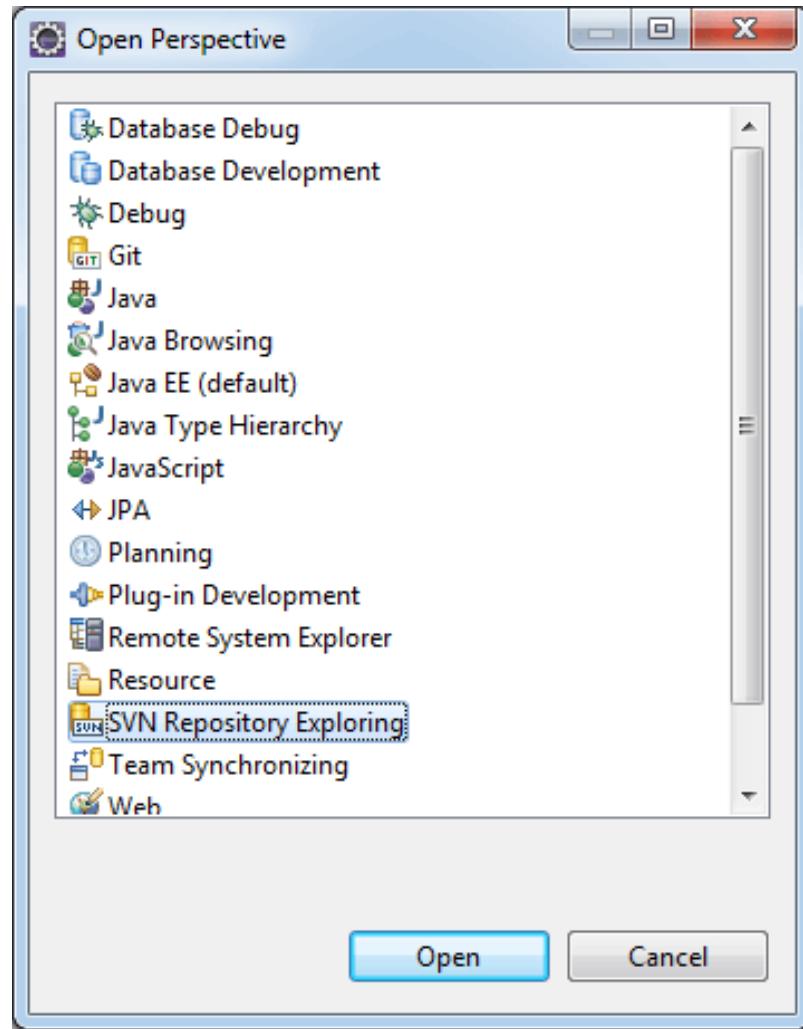


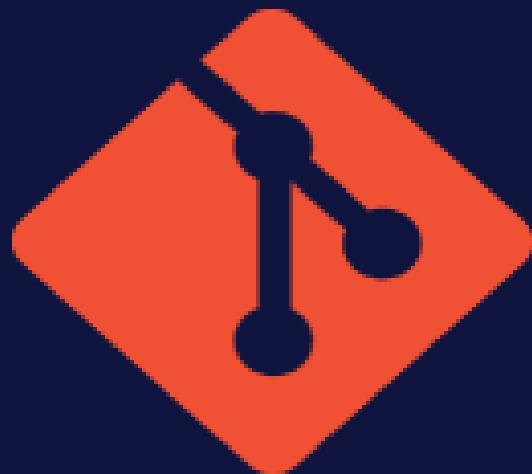
SVN for Eclipse





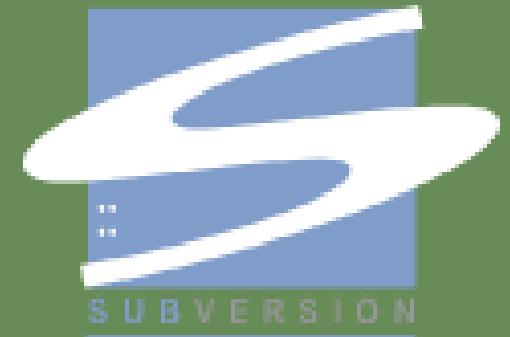
SVN for Eclipse





Git

VS



SVN



Git vs SVN

Git	SVN
It's a distributed version control system.	It's a Centralized version control system
Git is an SCM (source code management).	SVN is revision control.
Git has a cloned repository.	SVN does not have a cloned repository.
The Git branches are familiar to work. The Git system helps in merging the files quickly and also assist in finding the unmerged ones.	The SVN branches are a folder which exists in the repository. Some special commands are required For merging the branches.
Git does not have a Global revision number.	SVN has a Global revision number.
Git has cryptographically hashed contents that protect the contents from repository corruption taking place due to network issues or disk failures.	SVN does not have any cryptographically hashed contents.
Git stored content as metadata.	SVN stores content as files.
Git has more content protection than SVN.	SVN's content is less secure than Git.
Linus Torvalds developed git for Linux kernel.	CollabNet, Inc developed SVN.
Git is distributed under GNU (General public license).	SVN is distributed under the open-source license.

Ansible

- Ansible is an open-source platform used for automation and for various operations such as configuration management, application deployment, task automation, and IT orchestration.
- Ansible is easy to set up, and it is efficient, reliable, and powerful.
- It runs on Linux, Mac, or BSD.
- Apart from the free version, it has an enterprise edition called ‘Ansible Tower.’

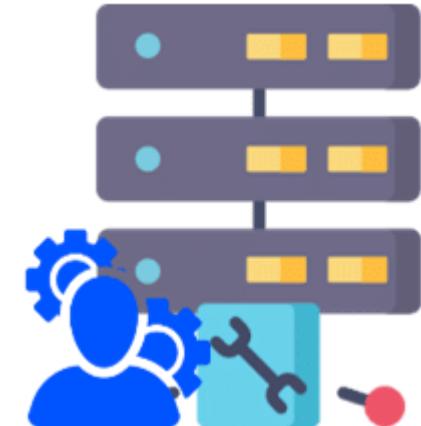


Ansible History

- Michael DeHaan developed Ansible, and the Ansible project began in February 2012.
- The creator of Cobbler and Func is also the controller of the Fedora Unified network.
- RedHat acquired the Ansible tool in 2015.
- Ansible is included as part of the Fedora distribution of the Linux.
- Ansible is also available for RedHat Enterprise Linux, Debian, CentOS, Oracle Linux, and Scientific Linux via Extra Packages for Enterprise Linux (EPEL) and Ubuntu as well as for other operating systems.

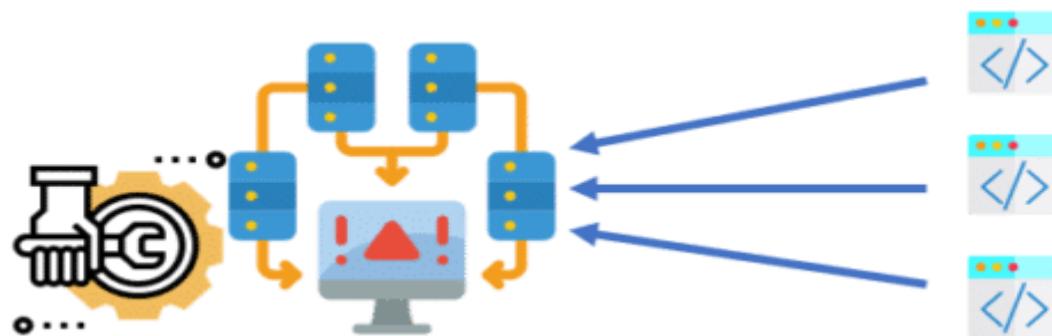
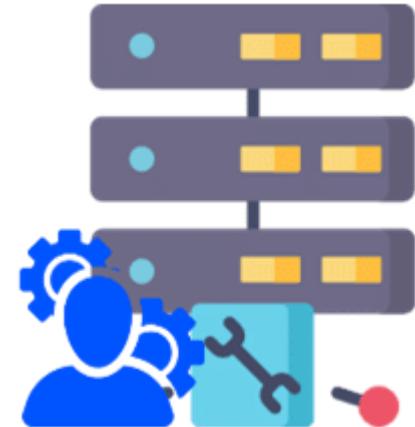
Why do we need Ansible?

- Let's consider a case where you as a System Admin is responsible for handling a company's infrastructure.
- Suppose, there are nine servers, out of which five are acting as web servers and the rest four as database servers.
- You want to install Tomcat on web servers and MySQL on database servers.
- In the traditional method, you will have to manage the servers manually, install the required software, and change the configurations, along with administering the services on each server individually.



Why do we need Ansible?

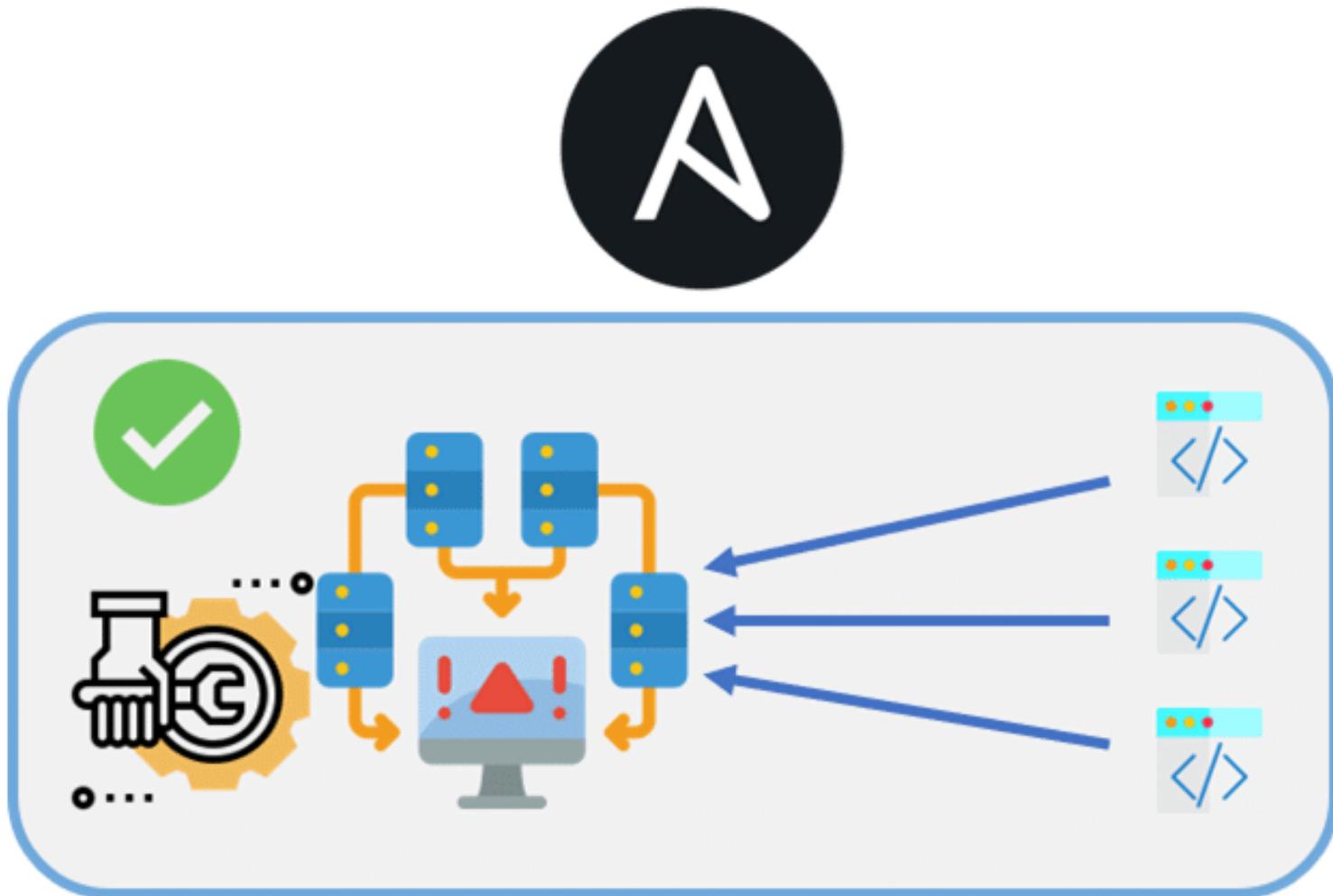
- When there are only a few services and configuration requirements, it will be easy for you to handle these amounts of servers; even if there is a minimal increment in the services, you can still handle the situation by provisioning a few more servers to maintain the infrastructure.



Why do we need Ansible?

- As the number of services starts to increase, the same task must be repeated multiple times.
- As sysadmins provision more servers, it gets more difficult for them to set up, update, and maintain all these servers manually.
- In most cases, sysadmins won't be able to set up each server identically.
- Also, such processes would end up hampering the velocity of the work for developers as the development team is agile and is always releasing software frequently.
- On the other hand, the sysadmins would have to spend extra time on system configurations and managing the infrastructure.

Why do we need Ansible?

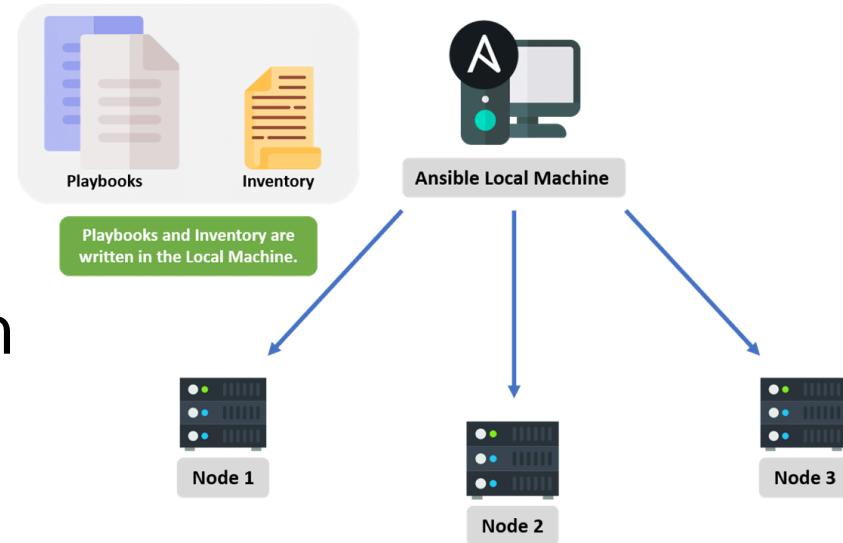


Important Terms in Ansible

- Controller Machine: This is where Ansible gets installed. The controller machine helps in enabling provisioning on servers we manage.
- Inventory: This is basically an initializing file that contains information about the servers that we are managing.
- Playbook: It is an organized unit of scripts defining an automated work for the configuration management of our server.
- Task: A task block defines a single procedure to be executed on the server like installing packages.

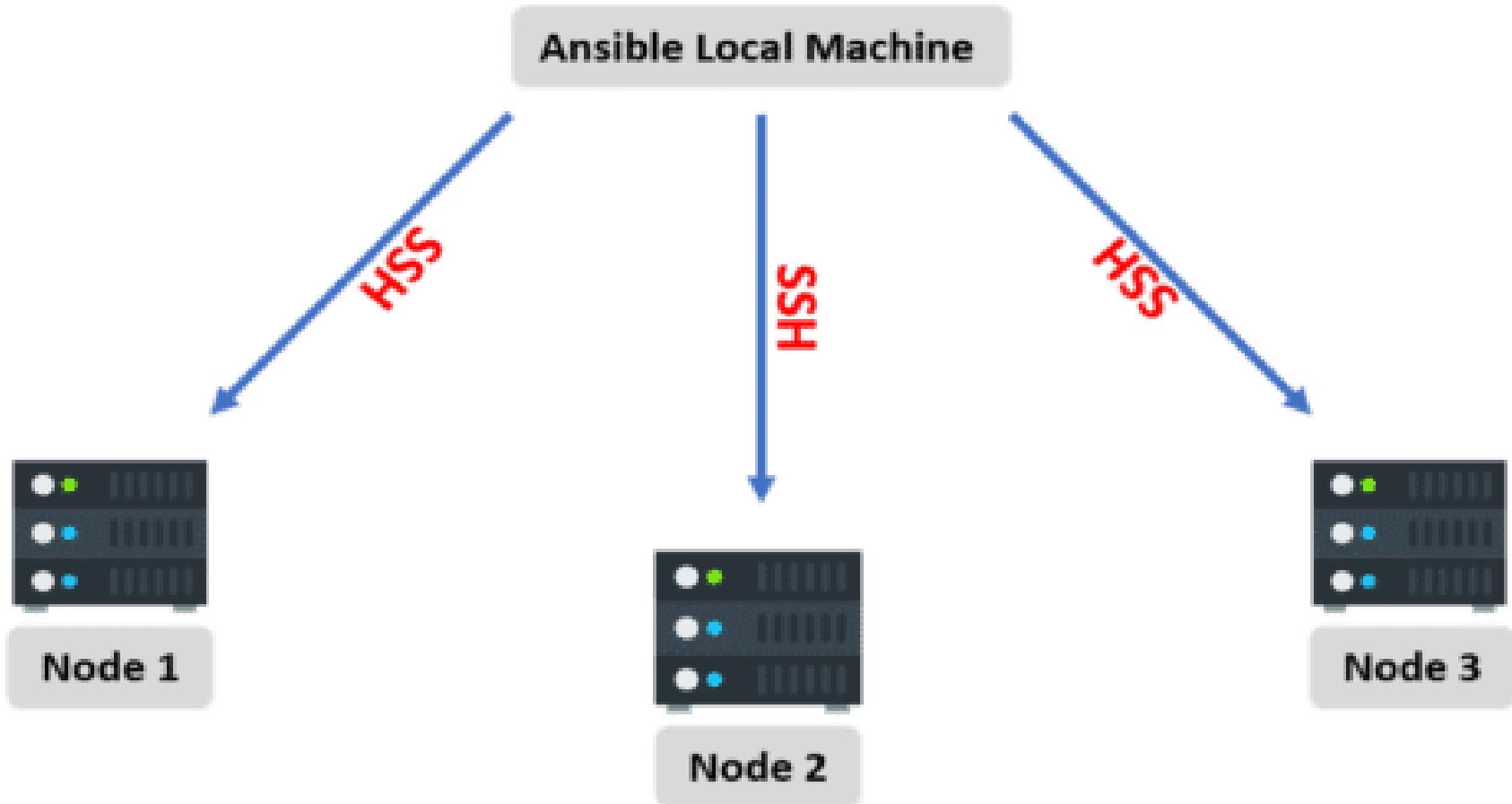
Ansible Workflow

- When services increase, sysadmins will provision more servers to do configuration management.
- They need not do it manually anymore but install Ansible on the master node where they need to write the code into the Ansible playbook to describe the setup, installation process, and the configuration required for these servers.



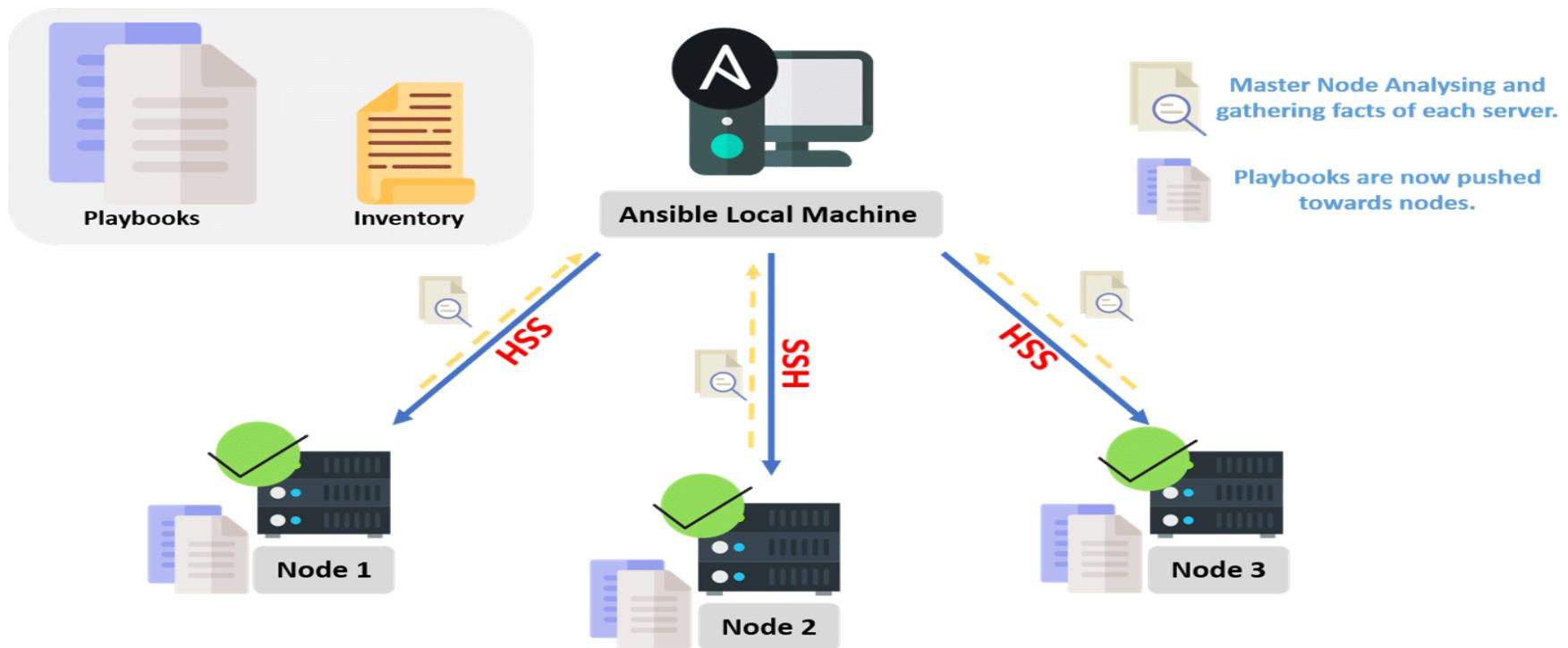
Ansible Workflow

The local machine connects to these servers (nodes) through an inventory using secured SSH connections.

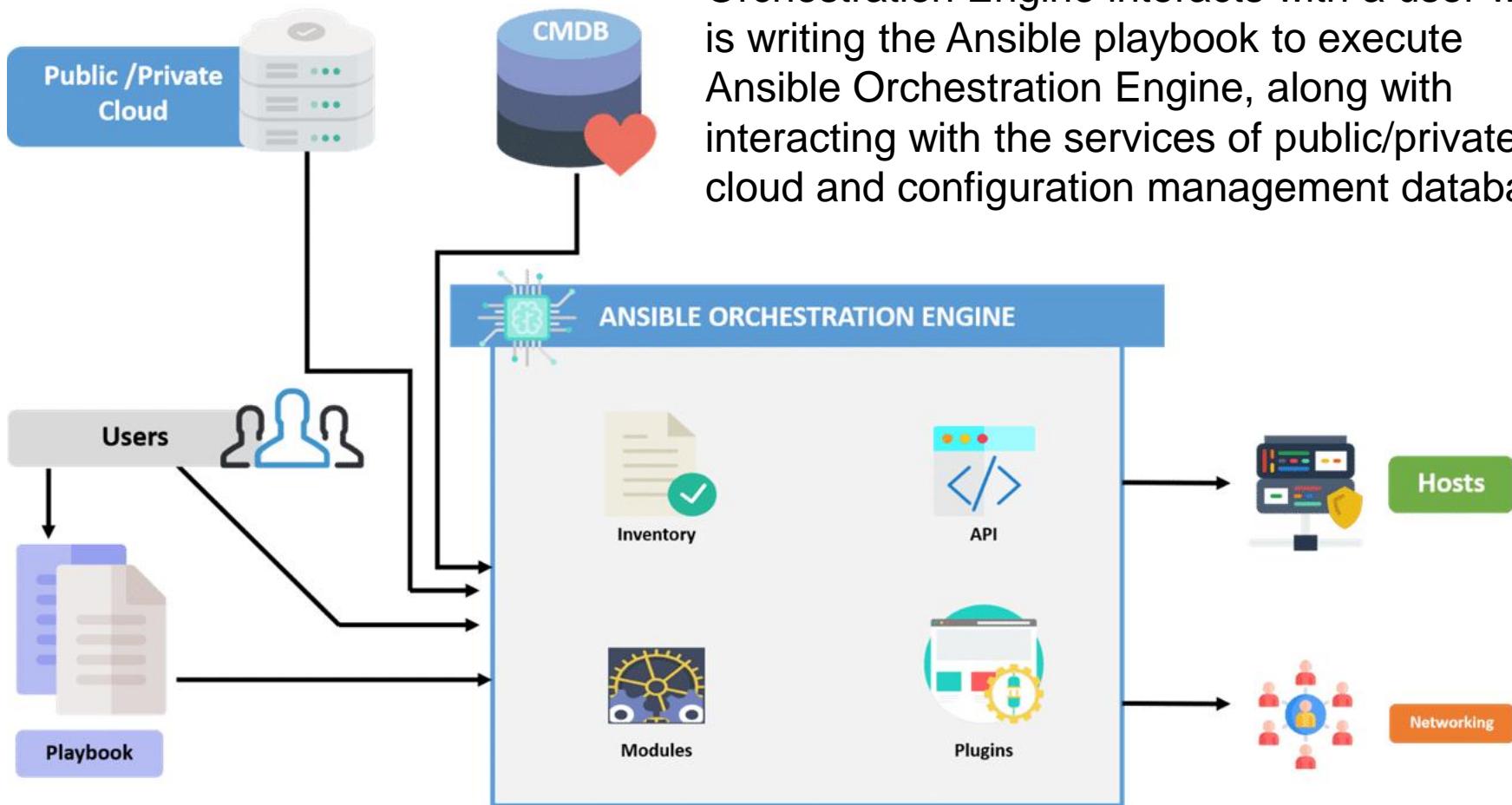


Ansible Workflow

Once these nodes are connected to the master server, then the node servers are analyzed and the playbook codes are pushed toward each of the servers so that these playbooks can configure the servers remotely, which leads to a consistent environment.



Ansible Architecture



From the diagram below it is clear that Ansible Orchestration Engine interacts with a user who is writing the Ansible playbook to execute Ansible Orchestration Engine, along with interacting with the services of public/private cloud and configuration management database.

Ansible architecture - Modules

- Modules
 - Ansible connects the nodes and pushes out small ‘Ansible Modules.’ Modules are executed by Ansible and then get removed when finished.
 - These modules can reside on any machine; no servers or daemons or databases are required here.
 - We can work with the text editor of our choice or a terminal and a version control system to keep track of the changes made in our content.

Ansible Architecture – Plugins and Infrastructure

- Plugins
 - A plugin is a piece of code that expands the core functionality of Ansible.
 - There are plenty of handy plugins, and we can write our own plugins as well.
- Inventory
 - An inventory is a list of hosts/nodes, having IP addresses, servers, databases, etc., which need to be managed.

Ansible Architecture – Playbooks and API

- Playbooks
 - We write our code in a playbook. It is simple and written in the YAML format and basically describes tasks that are supposed to be executed through Ansible.
 - We can launch tasks synchronously or asynchronously with playbooks.
- APIs
 - Ansible APIs work as transport for cloud services, either public or private.

Ansible Architecture – Hosts and Networking

- Hosts
 - Hosts are basically the node systems in the Ansible architecture getting automated by Ansible only.
 - They can be any type of machine such as Windows, Red Hat, Linux, etc.
- Networking
 - We can use Ansible to automate different networks. It uses the easy, simple, yet powerful, agentless automation framework for IT operations and development.
 - It uses a type of data model (playbook or role), which is separated from the Ansible Automation Engine that spans across different network hardware quite easily.

Ansible Architecture – Hosts and Networking

- CMDB
 - CMDB is a type of repository that acts as a data warehouse for the IT installations.
- Cloud
 - A network of remote servers on which we can store, manage, and process our data.
 - These servers are hosted on the Internet. For storing the data remotely rather than on local servers, we would just launch our resources and instances on the cloud and connect them to our servers, and we would have the wisdom of operating our task remotely.

Benefits of Using Ansible

- Agentless: connection can be SSHed and it uses Python, it can be configured with Ansible; no agent/software or additional firewall ports are required to install on our client or host systems for automation.
- Simple: Ansible uses a very simple syntax written in YAML known as playbooks—YAML (Yet Another Markup Language) is a human-readable data serialization language.
- No need of special coding skills to code and understand playbooks.
- It is very easy to install and execute tasks in order.



Benefits of Using Ansible

- Efficient: Not requiring any extra software on our servers means that there is more space for our resources.
- Powerful and flexible: Having powerful features gives us the capability to model even complex IT workflows in lesser time, along with managing infrastructure, networks, operating systems, and services that are already in use.

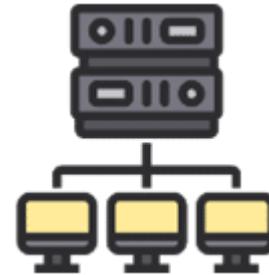
Operations by Ansible



- Configuration Management
- Ansible provides stability in the performance of our product by recording and updating stats in detail.
- This describes all information about the hardware and software of an enterprise/organization, such as versions and updates applied to the installed software packages, along with the locations and network addresses of hardware devices.
- For example, suppose, Company A wants to install the new version of Nginx on all of its server machines.
- It won't be feasible for the company to update each and every machine manually.
- Hence, it just installs Nginx on one of its machines and deploys it across the rest of the machines using Ansible playbooks.

Operations by Ansible

- Resource/Server Provisioning



- Whether we are booting/starting servers/virtual machines or creating cloud instances from various templates, Ansible is always there to help us with the smooth running of the process.
- Ansible makes sure to provide the required packages installed on our application.

Operations by Ansible

- Application Deployment
- With Ansible, we can define our application and manage its deployment.
- Instead of performing the deployment steps one by one, we just need to install Ansible on our machine and it will do the same tasks for us, even in lesser time.
- Provided those tasks are listed in our Ansible playbook, Ansible executes them in order.



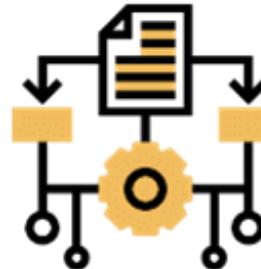
Operations by Ansible

- Security and Compliance
- With Ansible, we can easily configure our security details.
- We do it once in a control machine and the same security details will be spread across all the other nodes.



Operations by Ansible

- Orchestration
- We can also perform the orchestration of our application using Ansible.
- Ansible provides orchestration by aligning the business requests with the application, data, and infrastructure.
- It creates an application-aligned infrastructure that can be scaled up or down based on the needs.



Ansible Setup and Commands



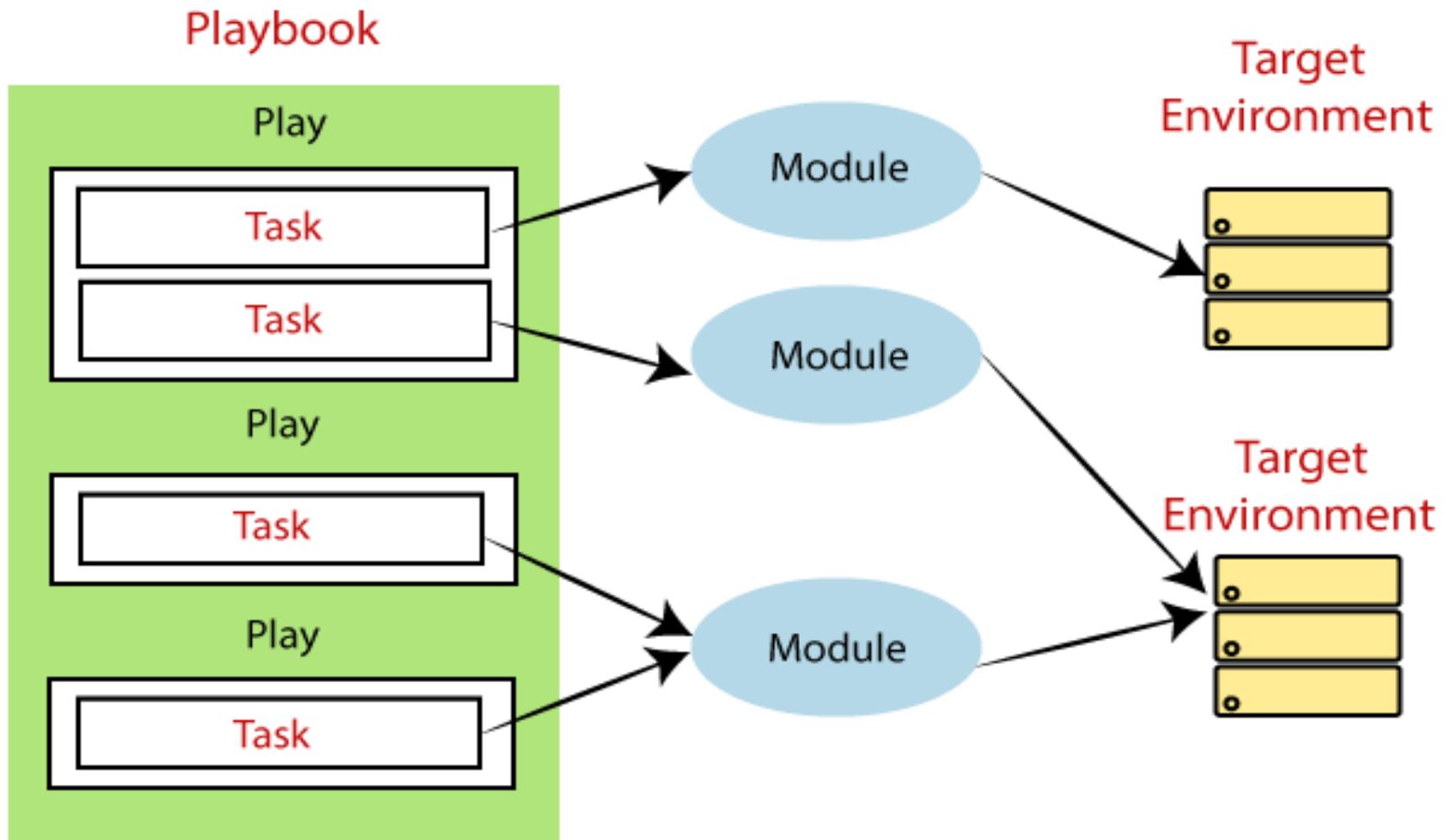
Ansible Playbook

- Playbooks are the files where the Ansible code is written. Playbooks are written in YAML format.
- YAML means "Yet Another Markup Language," so there is not much syntax needed.
- Playbooks are one of the core features of Ansible and tell Ansible what to execute, and it is used in complex scenarios.
- They offer increased flexibility.

Ansible Playbook

- Playbooks contain the steps which the user wants to execute on a particular machine.
- Playbooks are run sequentially.
- Playbooks are the building blocks for all the use cases of Ansible.
- Ansible playbooks tend to be more configuration language than a programming language.
- Through a playbook, you can designate specific roles to some of the hosts and other roles to other hosts

Playbook Structure





Playbook

- The function of the play is to map a set of instructions which is defined against a particular host.
- There are different YAML editors, but prefer to use a simple editor such as notepad++.
- First, open the notepad++ and copy-paste the below YAML and change the language to YAML (Language → YAML).
- A YAML starts with --- (3 hyphens) always.

Ansible Galaxy

- Ansible Galaxy is a galaxy website where users can share roles and to a command-line tool for installing, creating, and managing roles.
- Ansible Galaxy gives greater visibility to one of Ansible's most exciting features, such as application installation or reusable roles for server configuration.
- Lots of people share roles in the Ansible Galaxy.
- Ansible roles consist of many playbooks, which is a way to group multiple tasks into one container to do the automation in a very effective manner with clean, directory structures.



Adhoc Commands

- An Ansible ad hoc command uses the /usr/bin/ansible command-line tool to automate a single task on one or more managed nodes.
- ad hoc commands are quick and easy, but they are not reusable.
- ad hoc commands demonstrate the simplicity and power of Ansible.



Ansible Special Variables

- https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html



Ansible Vault

Ubuntu_18.04.5_VM_LinuxVMImages - VMware Workstation 16 Player (Non-commercial use only)

Player ▾ | || ▾ □ □ ▾

Activities Terminal Fri 00:55

File Edit View Search Terminal Help

```
linuxvmimages@ubuntu1804:~/Downloads$ sudo ansible-vault encrypt test.yml
New Vault password:
Confirm New Vault password:
Encryption successful
linuxvmimages@ubuntu1804:~/Downloads$ sudo ansible-vault view test.yml
Vault password:
-----
-host: all
linuxvmimages@ubuntu1804:~/Downloads$
```

The screenshot shows a terminal window on an Ubuntu 18.04.5 VM. The user runs `sudo ansible-vault encrypt test.yml` to encrypt a file named `test.yml`. They are prompted for a new vault password, which they enter twice. The output shows the encryption was successful. Then, they run `sudo ansible-vault view test.yml` to decrypt the file, but are prompted for the vault password again, showing that the file is now encrypted.

Questions



Module Summary

- In this module we discussed
 - Overview of Maven
 - Maven archetypes
 - Maven life cycle phases
 - The pom.xml file
 - Creation of Java projects using Maven
 - Creation of war files

