Index

33. Git flow
34. Git cheat sheet
35. Git init
36. Git add
37. Git commit
38. Git clone
39. Git stash
40. Git ignore
41. Git fork
42. Git repository
43. Git index
44. Git head
45. Git origin master
46. Git remote
47. Git tags
48. Upstream and downstream
49. Git checkout
50. Git revert
51. Git reset
52. Git rm
53. Git cherry-pick
54. Git log
55. Git diff
56. Git status
57. Git branch
58. Merge and merge conflict
59. Git rebase
60. Git squash
61. Git fetch
62. Git pull
63. Git push

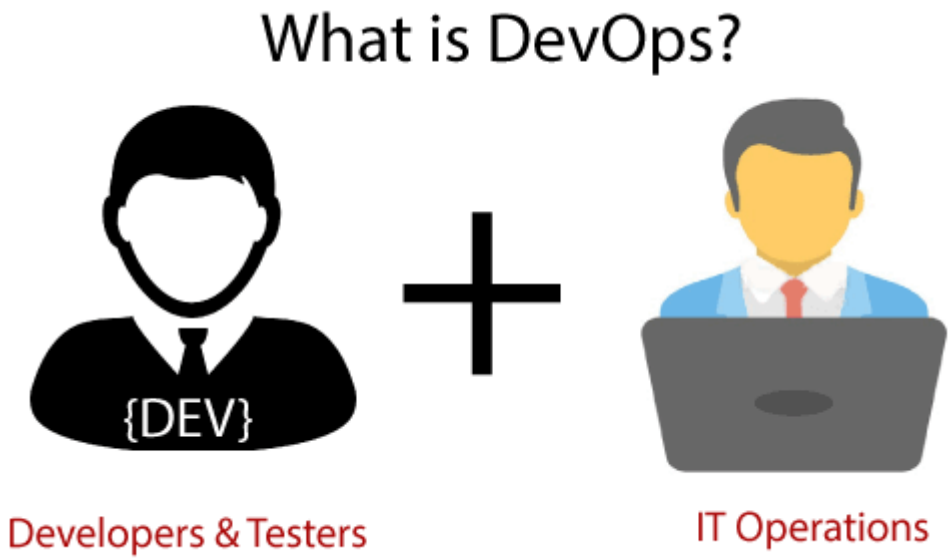
DevOps Tutorial

The DevOps is the combination of two words, one is **Development** and other is **Operations**. It is a culture to promote the development and operation process collectively.

The DevOps tutorial will help you to learn DevOps basics and provide depth knowledge of various DevOps tools such as **Git, Ansible, Docker, Puppet, Jenkins, Chef, Nagios**, and **Kubernetes**.

What is DevOps?

The DevOps is a combination of two words, one is software Development, and second is Operations. This allows a single team to handle the entire application lifecycle, from development to **testing, deployment**, and **operations**. DevOps helps you to reduce the disconnection between software developers, quality assurance (QA) engineers, and system administrators.



DevOps promotes collaboration between Development and Operations team to deploy code to production faster in an automated & repeatable way.

DevOps helps to increase organization speed to deliver applications and services. It also allows organizations to serve their customers better and compete more strongly in the market.

DevOps can also be defined as a sequence of development and IT operations with better communication and collaboration.

DevOps has become one of the most valuable business disciplines for enterprises or organizations. With the help of DevOps, **quality**, and **speed** of the application delivery has improved to a great extent.

DevOps is nothing but a practice or methodology of making "**Developers**" and "**Operations**" folks work together. DevOps represents a change in the IT culture with a complete focus on rapid IT service delivery through the adoption of agile practices in the context of a system-oriented approach.

DevOps is all about the integration of the operations and development process. Organizations that have adopted DevOps noticed a 22% improvement in software quality and a 17% improvement in application deployment frequency and achieve a 22% hike in customer satisfaction. 19% of revenue hikes as a result of the successful DevOps implementation.

Why DevOps?

Before going further, we need to understand why we need the DevOps over the other methods.

- The operation and development team worked in complete isolation.
- After the design-build, the testing and deployment are performed respectively. That's why they consumed more time than actual build cycles.
- Without the use of DevOps, the team members are spending a large amount of time on designing, testing, and deploying instead of building the project.
- Manual code deployment leads to human errors in production.
- Coding and operation teams have their separate timelines and are not in synch, causing further delays.

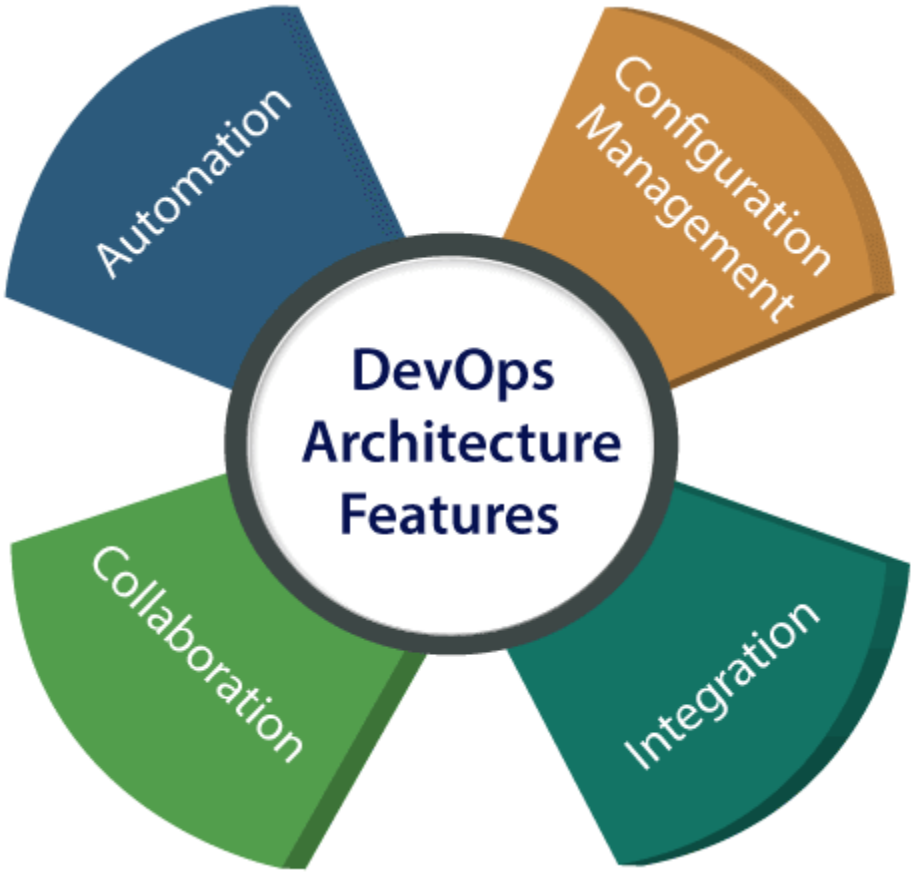
DevOps History

- In 2009, the first conference named **DevOpsdays** was held in Ghent Belgium. Belgian consultant and Patrick Debois founded the conference.
- In 2012, the state of DevOps report was launched and conceived by Alanna Brown at Puppet.
- In 2014, the annual State of DevOps report was published by Nicole Forsgren, Jez Humble, Gene Kim, and others. They found DevOps adoption was accelerating in 2014 also.
- In 2015, Nicole Forsgren, Gene Kim, and Jez Humble founded DORA (DevOps Research and Assignment).

- In 2017, Nicole Forsgren, Gene Kim, and Jez Humble published "Accelerate: Building and Scaling High Performing Technology Organizations".

DevOps Architecture Features

Here are some key features of DevOps architecture, such as:



1) Automation

Automation can reduce time consumption, especially during the testing and deployment phase. The productivity increases, and releases are made quicker by automation. This will lead in catching bugs quickly so that it can be fixed easily. For contiguous delivery, each code is defined through automated tests, cloud-based services, and builds. This promotes production using automated deploys.

2) Collaboration

The Development and Operations team collaborates as a DevOps team, which improves the cultural model as the teams become more productive with their productivity, which strengthens accountability and ownership. The teams share their responsibilities and work closely in sync, which in turn makes the deployment to production faster.

3) Integration

Applications need to be integrated with other components in the environment. The integration phase is where the existing code is combined with new functionality and then tested. Continuous integration and testing enable continuous development. The frequency in the releases and micro-services leads to significant operational challenges. To overcome such problems, continuous integration and delivery are implemented to deliver in a **quicker, safer**, and **reliable manner**.

4) Configuration management

It ensures the application to interact with only those resources that are concerned with the environment in which it runs. The configuration files are not created where the external configuration to the application is separated from the source code. The configuration file can be written during deployment, or they can be loaded at the run time, depending on the environment in which it is running.

DevOps Advantages and Disadvantages

Here are some advantages and disadvantages that DevOps can have for business, such as:

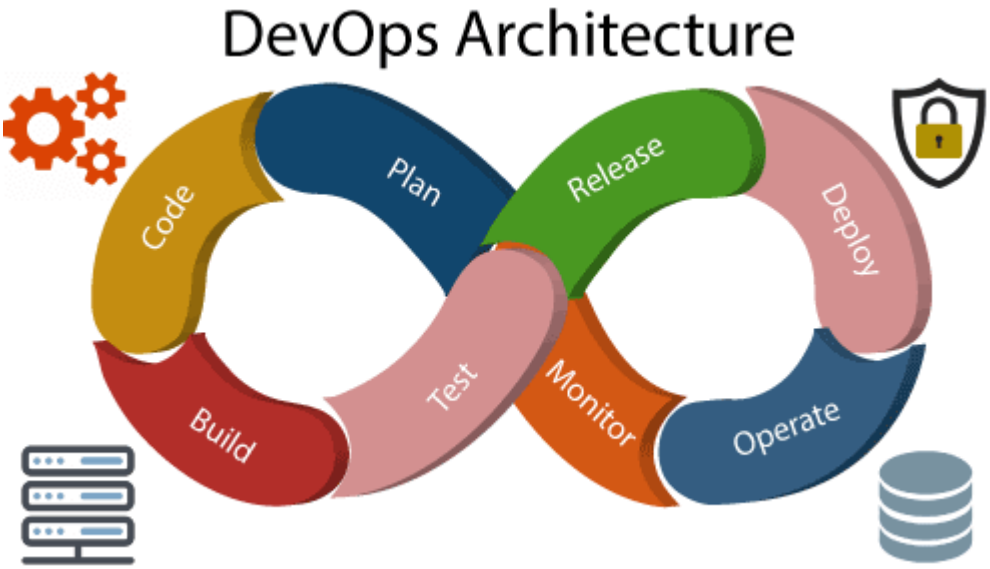
Advantages

- DevOps is an excellent approach for quick development and deployment of applications.
- It responds faster to the market changes to improve business growth.
- DevOps escalate business profit by decreasing software delivery time and transportation costs.
- DevOps clears the descriptive process, which gives clarity on product development and delivery.
- It improves customer experience and satisfaction.

- DevOps simplifies collaboration and places all tools in the cloud for customers to access.
- DevOps means collective responsibility, which leads to better team engagement and productivity.

Disadvantages

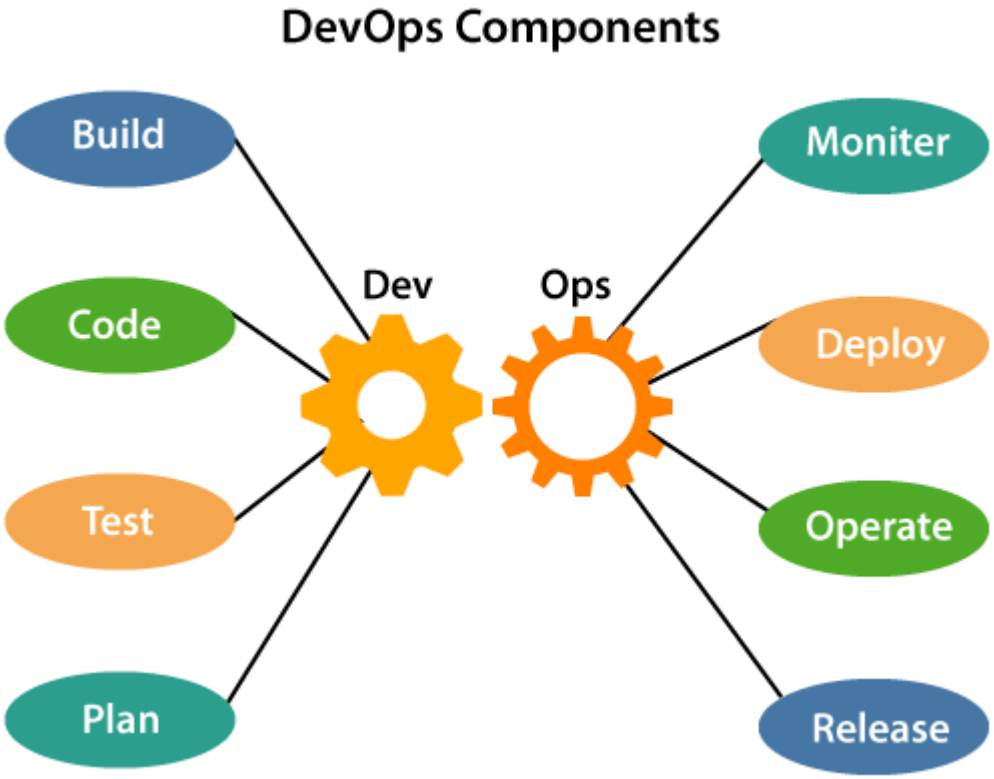
- DevOps professional or expert's developers are less available.
- Developing with DevOps is so expensive.
- Adopting new DevOps technology into the industries is hard to manage in short time.
- Lack of DevOps knowledge can be a problem in the continuous integration of automation projects.
- # DevOps Architecture



- Development and operations both play essential roles in order to deliver applications. The deployment comprises analyzing the **requirements, designing, developing**, and **testing** of the software components or frameworks.

- The operation consists of the administrative processes, services, and support for the software. When both the development and operations are combined with collaborating, then the DevOps architecture is the solution to fix the gap between deployment and operation terms; therefore, delivery can be faster.

- DevOps architecture is used for the applications hosted on the cloud platform and large distributed applications. Agile Development is used in the DevOps architecture so that integration and delivery can be contiguous. When the development and operations team works separately from each other, then it is time-consuming to **design, test**, and **deploy**. And if the terms are not in sync with each other, then it may cause a delay in the delivery. So DevOps enables the teams to change their shortcomings and increases productivity.
- Below are the various components that are used in the DevOps architecture:



- ## 1) Build
- Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.
- ## 2) Code
- Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, getting notified about the reason behind the difference in the actual and the expected output, and if necessary reverting to the original code developed. The code can be appropriately arranged in **files, folders**, etc. And they can be reused.
- ## 3) Test

- The application will be ready for production after testing. In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.

4) Plan

- DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.

5) Monitor

- Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with services where the log data may get monitored through many third-party tools such as **Splunk**.

6) Deploy

- Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.

7) Operate

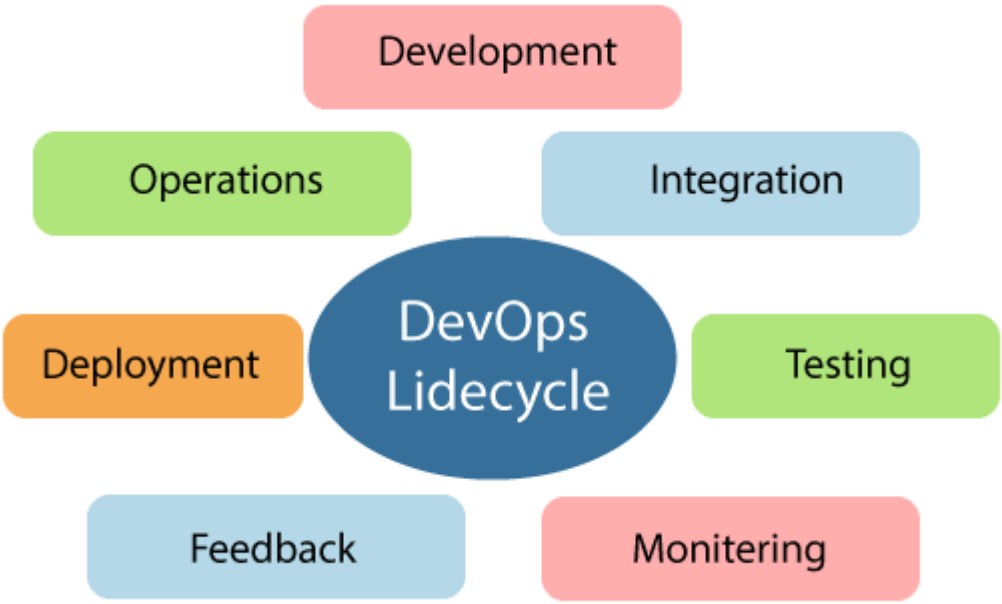
- DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.

8) Release

- Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering. Many processes involved in release management commonly used to do the deployment in the production environment manually to lessen the impact on the customers.

DevOps Lifecycle

- DevOps defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.



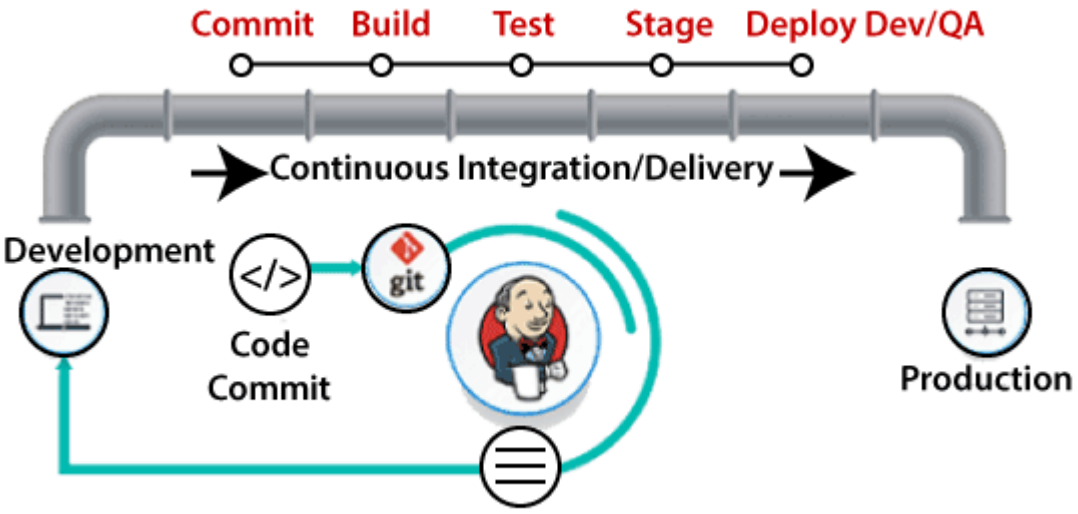
- Learning DevOps is not complete without understanding the DevOps lifecycle phases. The DevOps lifecycle includes seven phases as given below:

1) Continuous Development

- This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

2) Continuous Integration

- This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.
- The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.



- Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.

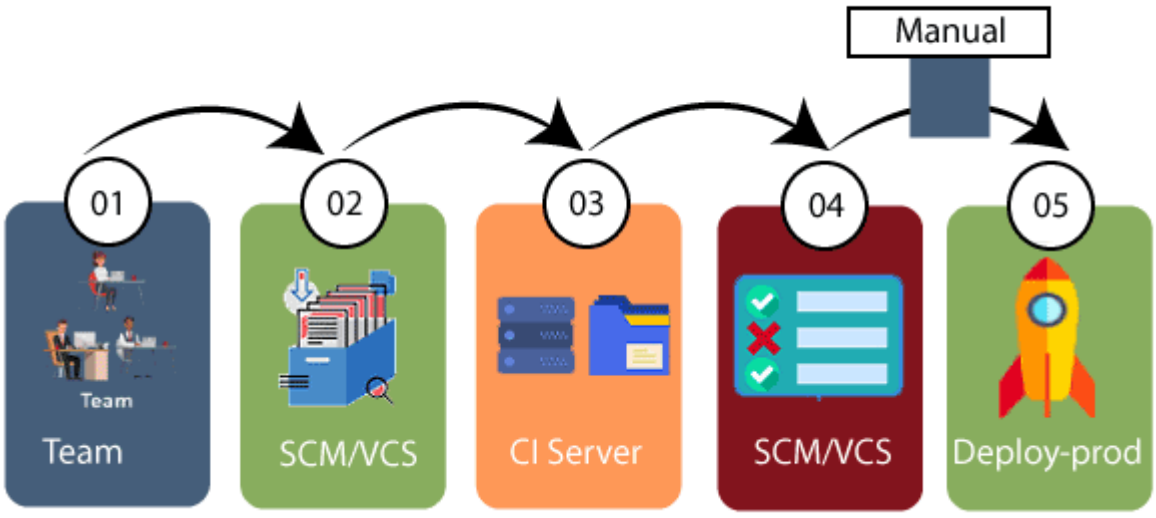
3) Continuous Testing

- This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.

- **Selenium** does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.
- Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can

schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

- ## 4) Continuous Monitoring
- Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.
- It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.
- ## 5) Continuous Feedback
- The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.
- The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version. It kills the efficiency that may be possible with the app and reduce the number of interested customers.
- ## 6) Continuous Deployment
- In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



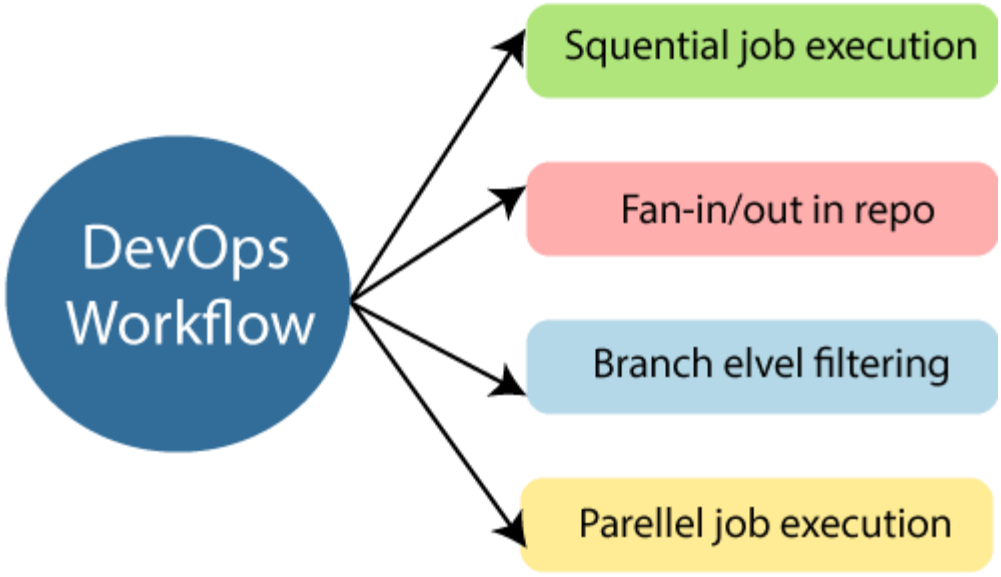
- The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.
- Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.
- Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

7) Continuous Operations

- All DevOps operations are based on the continuity with complete automation of the release process and allow the organization to accelerate the overall time to market continuingly.
- It is clear from the discussion that continuity is the critical factor in the DevOps in removing steps that often distract the development, take it longer to detect issues and produce a better version of the product after several months. With DevOps, we can make any software product more efficient and increase the overall count of interested customers in your product.

DevOps Workflow

DevOps workflow provides a visual overview of the sequence in which input is provided. Also, it tells about which one action is performed, and output is generated for an operations process.



DevOps workflow allows the ability to separate and arrange the jobs which are top requested by the users. Also, it gives the ability to mirror their ideal process in the configuration jobs.

DevOps Principles

The main principles of DevOps are Continuous delivery, automation, and fast reaction to the feedback.

1. **End to End Responsibility:** DevOps team need to provide performance support until they become the end of life. It enhances the responsibility and the quality of the products engineered.
2. **Continuous Improvement:** DevOps culture focuses on continuous improvement to minimize waste. It continuously speeds up the growth of products or services offered.
3. **Automate Everything:** Automation is an essential principle of the DevOps process. This is for software development and also for the entire infrastructure landscape.
4. **Custom Centric Action:** DevOps team must take customer-centric for that they should continuously invest in products and services.

5. **Monitor and test everything:** The DevOps team needs to have robust monitoring and testing procedures.
6. **Work as one team:** In the DevOps culture role of the designers, developers, and testers are already defined. All they needed to do is work as one team with complete collaboration.

These principles are achieved through several DevOps practices, which include frequent deployments, QA automation, continuous delivery, validating ideas as early as possible, and in-team collaboration.

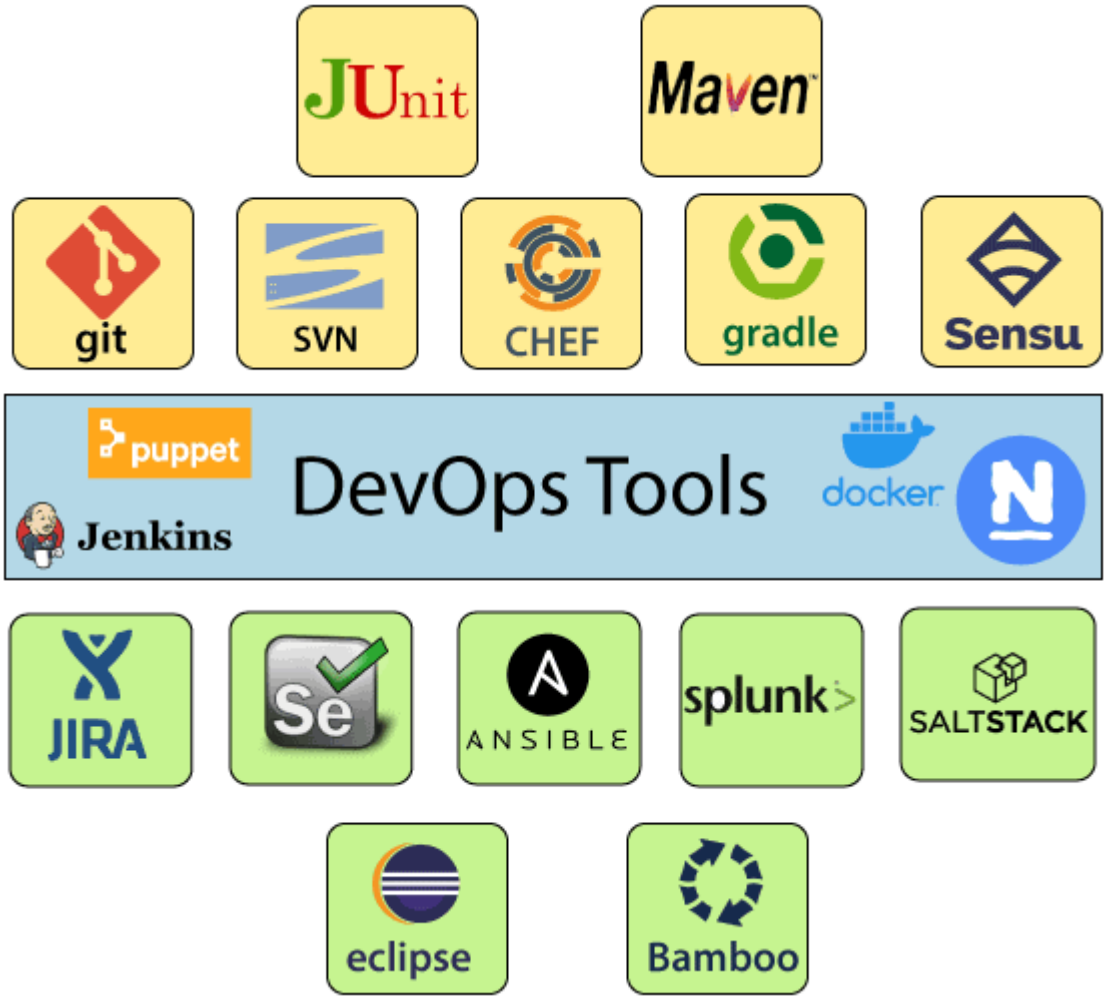
DevOps Practices

Some identified DevOps practices are:

- Self-service configuration
- Continuous build
- Continuous integration
- Continuous delivery
- Incremental testing
- Automated provisioning
- Automated release management

DevOps Tools

Here are some most popular DevOps tools with brief explanation shown in the below image, such as:



1) Puppet

Puppet is the most widely used DevOps tool. It allows the delivery and release of the technology changes quickly and frequently. It has features of versioning, automated testing, and continuous delivery. It enables to manage entire infrastructure as code without expanding the size of the team.

Features

- Real-time context-aware reporting.
- Model and manage the entire environment.
- Defined and continually enforce infrastructure.
- Desired state conflict detection and remediation.
- It inspects and reports on packages running across the infrastructure.

- It eliminates manual work for the software delivery process.
- It helps the developer to deliver great software quickly.

2) Ansible

Ansible is a leading DevOps tool. Ansible is an open-source IT engine that automates application deployment, cloud provisioning, intra service orchestration, and other IT tools. It makes it easier for DevOps teams to scale automation and speed up productivity.

Ansible is easy to deploy because it does not use any **agents** or **custom security** infrastructure on the client-side, and by pushing modules to the clients. These modules are executed locally on the client-side, and the output is pushed back to the Ansible server.

Features

- It is easy to use to open source deploy applications.
- It helps in avoiding complexity in the software development process.
- It eliminates repetitive tasks.
- It manages complex deployments and speeds up the development process.

3) Docker

Docker is a high-end DevOps tool that allows building, ship, and run distributed applications on multiple systems. It also helps to assemble the apps quickly from the components, and it is typically suitable for container management.

Features

- It configures the system more comfortable and faster.
- It increases productivity.
- It provides containers that are used to run the application in an isolated environment.
- It routes the incoming request for published ports on available nodes to an active container. This feature enables the connection even if there is no task running on the node.
- It allows saving secrets into the swarm itself.

4) Nagios

Nagios is one of the more useful tools for DevOps. It can determine the errors and rectify them with the help of network, infrastructure, server, and log monitoring systems.

Features

- It provides complete monitoring of desktop and server operating systems.
- The network analyzer helps to identify bottlenecks and optimize bandwidth utilization.
- It helps to monitor components such as services, application, OS, and network protocol.
- It also provides to complete monitoring of Java Management Extensions.

5) CHEF

A chef is a useful tool for achieving scale, speed, and consistency. The chef is a cloud-based system and open source technology. This technology uses Ruby encoding to develop essential building blocks such as recipes and cookbooks. The chef is used in infrastructure automation and helps in reducing manual and repetitive tasks for infrastructure management.

Chef has got its convention for different building blocks, which are required to manage and automate infrastructure.

Features

- It maintains high availability.
- It can manage multiple cloud environments.
- It uses popular Ruby language to create a domain-specific language.
- The chef does not make any assumptions about the current status of the node. It uses its mechanism to get the current state of the machine.

6) Jenkins

Jenkins is a DevOps tool for monitoring the execution of repeated tasks. Jenkins is a software that allows continuous integration. Jenkins will be installed on a server where the

central build will take place. It helps to integrate project changes more efficiently by finding the issues quickly.

Features

- Jenkins increases the scale of automation.
- It can easily set up and configure via a web interface.
- It can distribute the tasks across multiple machines, thereby increasing concurrency.
- It supports continuous integration and continuous delivery.
- It offers 400 plugins to support the building and testing any project virtually.
- It requires little maintenance and has a built-in GUI tool for easy updates.

7) Git

Git is an open-source distributed version control system that is freely available for everyone. It is designed to handle minor to major projects with speed and efficiency. It is developed to co-ordinate the work among programmers. The version control allows you to track and work together with your team members at the same workspace. It is used as a critical distributed version-control for the DevOps tool.

Features

- It is a free open source tool.
- It allows distributed development.
- It supports the pull request.
- It enables a faster release cycle.
- Git is very scalable.
- It is very secure and completes the tasks very fast.

8) SALTSTACK

Stackify is a lightweight DevOps tool. It shows real-time error queries, logs, and more directly into the workstation. SALTSTACK is an ideal solution for intelligent orchestration for the software-defined data center.

Features

- It eliminates messy configuration or data changes.
- It can trace detail of all the types of the web request.
- It allows us to find and fix the bugs before production.
- It provides secure access and configures image caches.
- It secures multi-tenancy with granular role-based access control.
- Flexible image management with a private registry to store and manage images.

9) Splunk

Splunk is a tool to make machine data usable, accessible, and valuable to everyone. It delivers operational intelligence to DevOps teams. It helps companies to be more secure, productive, and competitive.

Features

- It has the next-generation monitoring and analytics solution.
- It delivers a single, unified view of different IT services.
- Extend the Splunk platform with purpose-built solutions for security.
- Data drive analytics with actionable insight.

10) Selenium

Selenium is a portable software testing framework for web applications. It provides an easy interface for developing automated tests.

Features

- It is a free open source tool.
- It supports multiplatform for testing, such as Android and ios.
- It is easy to build a keyword-driven framework for a WebDriver.
- It creates robust browser-based regression automation suites and tests.

DevOps Automation

Automation is the crucial need for DevOps practices, and automate everything is the fundamental principle of DevOps. Automation kick starts from the code generation on

the developers machine, until the code is pushed to the code and after that to monitor the application and system in the production.

Automating infrastructure set up and configurations, and software deployment is the key highlight of DevOps practice. DevOps practice id is dependent on automation to make deliveries over a few hours and make frequent deliveries across platforms.

Automation in DevOps boosts speed, consistency, higher accuracy, reliability, and increases the number of deliveries. Automation in DevOps encapsulates everything right from the building, deploying, and monitoring.

DevOps Automation Tools

In large DevOps team that maintain extensive massive IT infrastructure can be classified into six categories, such as:

- Infrastructure Automation
- Configuration Management
- Deployment Automation
- Performance Management
- Log management
- Monitoring

Below are few tools in each of these categories let see in brief, such as:

Infrastructure Automation

Amazon Web Services (AWS): Being a cloud service, you don't need to be physically present in the data center, they are easy to scale on-demand, and there are no up-front hardware costs. It can be configured to provide more servers based on traffic automatically.

Configuration Management

Chef: Chef is a handy DevOps tool for achieving speed, scale, and consistency. It can be used to ease out of complex tasks and perform configuration management. With the help of this tool, the DevOps team can avoid making changes across ten thousand servers.

Rather, they need to make changes in one place, which is automatically reflected in other servers.

Deployment Automation

Jenkins: It facilitates continuous integration and testing. It helps to integrate project changes more efficiently by quickly finding issues as soon as built is deployed.

Performance Management

App Dynamic: It offers real-time performance monitoring. The data collected by this tool help developers to debug when issues occur.

Log Management

Splunk: This DevOps tool solves issues such as storing, aggregating, and analyzing all logs in one place.

Monitoring

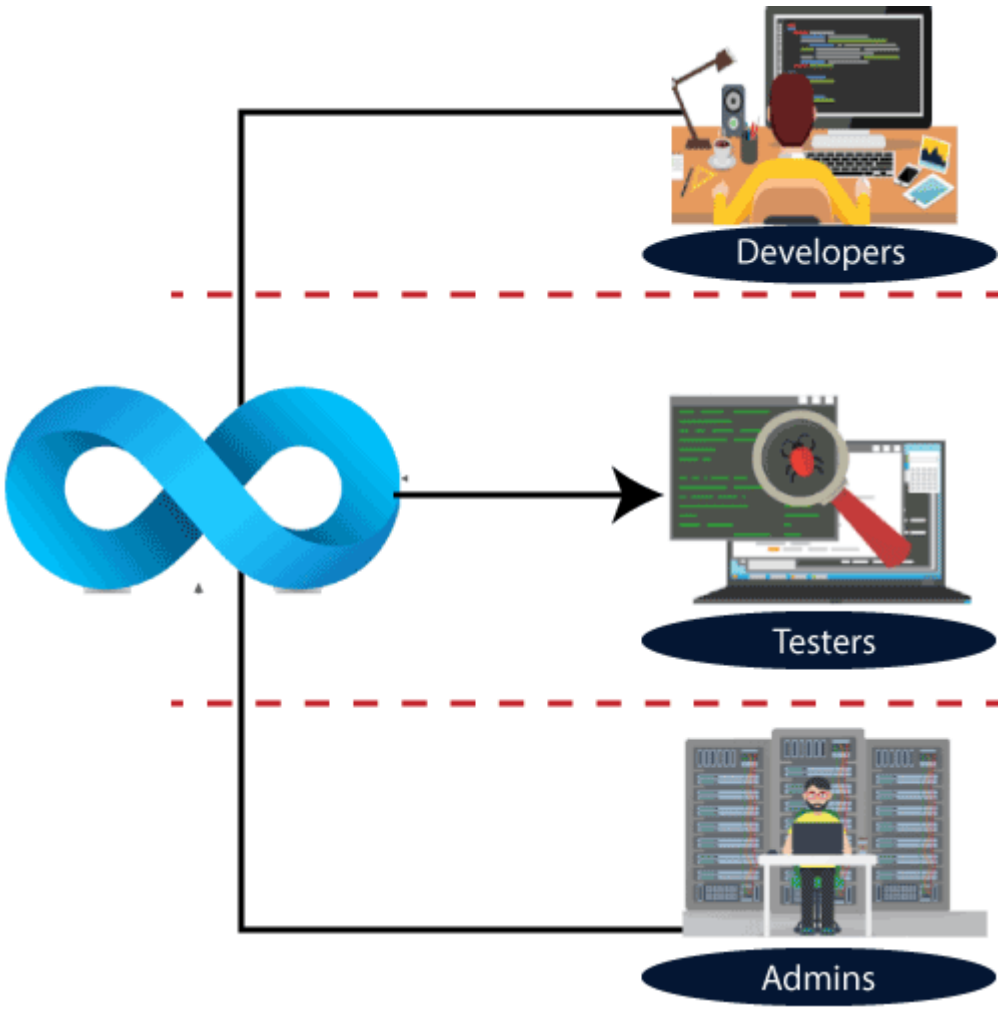
Nagios: It notified people when infrastructure and related service go down. Nagios is a tool for this purpose, which helps the DevOps team to find and correct problems.

DevOps Engineers

DevOps Engineer is an IT professional who works with system operators, software developers, and other production IT staff to administer code releases.

DevOps engineer understands the software development lifecycle and various automation tools for developing digital pipelines.

DevOps have hard as well as soft skills to communicate and collaborate with development, testing, and operations teams.



DevOps engineers need to code occasionally from scratch, and they must have the basics of software development languages.

The DevOps engineer will work with development team staff to tackle the coding and scripting needed to connect elements of code, like libraries or software development kits.

A bachelor's degree in computer science or related fields is generally required for DevOps engineers. Many companies prefer those who have a master's degree and at least three to five years of work experience in this field. HTTP, HTML, CSS, SSL, XML, Linux, Java, Amazon Web Services (AWS), NoSQL technologies, DNS, and web app development.

DevOps Engineer Roles and Responsibilities

DevOps engineers work full time. They are responsible for the production and continuing maintenance of a software application platform.

Below are some roles, responsibilities, and skills which are expected from DevOps engineers, such as:

- Manage projects effectively through an open standard based platform.
- Increases project visibility through traceability.
- Improve quality and reduce the development cost with collaboration.
- DevOps should have the soft skill of problem solver and a quick learner.
- Analyze, design, and evaluate automation scripts and systems.
- Able to perform system troubleshooting and problem-solving across the platform and application domains.
- Ensuring the critical resolution of system issues by using the best cloud security solution services.

DevOps Engineers Salary

The DevOps Engineers salary estimates are based on two reports of salaries, wages, bonuses, and hourly pay.

Here is a list of DevOps engineers salary according to the most recent DevOps engineer salary report, such as:

Salary	Area	Experiences	Company
₹4,20,000	A DevOps engineer in the **Bengaluru** area reported making **₹4,20,000** per year.	0-year experience	Private
₹3,00,000	A DevOps engineer in the **Bengaluru** area reported making **₹3,00,000** per year.	1-2 year experience	Public
₹5,00,000	A DevOps engineer in the **Hyderabad** area reported making **₹5,00,000** per year.	3-4 year experiences	Public
₹3,00,000	A DevOps engineer in the **Hyderabad** area reported making **₹3,00,000** per year.	3-4 year experience	Private
₹6,00,000	An Azure DevOps engineer in the **Chennai** area reported making **₹6,00,000** per year.	1-2 year experience	Public
₹4,80,000	A DevOps engineer in the **Pune** area reported making **₹4,80,000** per year.	3-4 year experience	Public
₹11,06,561	A DevOps engineer in the **New Delhi** area reported making **₹11,06,561** per year.	3-4 year experience	Private

DevOps Pipeline

A pipeline in software engineering team is a set of automated processes which allows DevOps professionals and developer to reliably and efficiently compile, build, and deploy their code to their production compute platforms.

The most common components of a pipeline in DevOps are build automation or continuous integration, test automation, and deployment automation.

A pipeline consists of a set of tools which are classified into the following categories such as:

- Source control
- Build tools
- Containerization
- Configuration management
- Monitoring

Continuous Integration Pipeline

Continuous integration (CI) is a practice in which developers can check their code into a version-controlled repository several times per day. Automated build pipelines are triggered by these checks which allows fast and easy to locate error detection.

Some significant benefits of CI are:

- Small changes are easy to integrate into large codebases.
- More comfortable for other team members to see what you have been working.
- Fewer integration issues allowing rapid code delivery.
- Bugs are identified early, making them easier to fix, resulting in less debugging work.

Continuous Delivery Pipeline

Continuous delivery (CD) is the process that allows operation engineers and developers to deliver bug fixes, features, and configuration change into production reliably, quickly, and sustainably. Continuous delivery offers the benefits of code delivery pipelines, which are carried out that can be performed on demand.

Some significant benefits of the CD are:

- Faster bug fixes and features delivery.

- CD allows the team to work on features and bug fixes in small batches, which means user feedback received much quicker. It reduces the overall time and cost of the project.

DevOps Methodology

We have a demonstrated methodology that takes an approach to cloud adoption. It accounts for all the factors required for successful approval such as people, process, and technology, resulting in a focus on the following critical consideration:

- **The Teams:** Mission or project and cloud management.
- **Connectivity:** Public, on-premise, and hybrid cloud network access.
- **Automation:** Infrastructure as code, scripting the orchestration and deployment of resources.
- **On-boarding Process:** How the project gets started in the cloud.
- **Project Environment:** TEST, DEV, PROD (identical deployment, testing, and production).
- **Shared Services:** Common capabilities provided by the enterprise.
- **Naming Conventions:** Vital aspect to track resource utilization and billing.
- **Defining Standards Role across the Teams:** Permissions to access resources by job function.

Azure DevOps

Azure DevOps is also known as Microsoft visual studio team services (VSTS). It is a set of collaborative development tools built for the cloud. VSTS was commonly used as a standalone term, and Azure DevOps is a platform which is made up of a few different products, such as:

- Azure Test Plans
- Azure Boards
- Azure Repos

- Azure Pipeline
- Azure Artifacts

Azure DevOps is everything that needs to turn an idea into a working piece software. You can plan a project with azure tools.

The azure pipeline is the CI component of azure DevOps. The azure pipeline is Microsoft's cloud-native continuous integration server, which allows teams to continuously build, test, and deploy all from the cloud. An azure pipeline can connect to any number of source code repositories such as Azure Repos, GitHub, Tests, to grab code and artifacts for application delivery.

Azure DevOps Server

Azure DevOps Server is a Microsoft product that provides version control, requirements management, reporting, lab management, project management, testing, automated builds, and release management capabilities. It covers the entire application of lifecycle and enables DevOps capabilities.

Azure DevOps can be used as a back-end to the numerous integrated development environments, but it is modified for Microsoft visual studio and eclipse on all platforms.

Azure DevOps Services

Microsoft announced the release of the software as a service offering of visual studio on the Microsoft Azure platform at the time Microsoft called it a visual studio online.

Microsoft offers visual studio, basic, and stakeholder subscriber access levels for the Azure DevOps services. The basic plan is free of cost for up to five users. Users with a visual studio subscription can be added to a project with no additional charge.

AWS DevOps

AWS is the best cloud service provider, and DevOps is the implementation of the software development lifecycle.

Here are some reasons which make AWS DevOps a highly popular combination, such as:

- AWS CloudFormation
- AWS EC2
- AWS CloudWatch
- AWS CodePipeline

Let's see all of these by one in brief such as:

AWS CloudFormation

DevOps team is required to create and release cloud instances and services more frequently in comparison to development teams. Templates of AWS resources such as EC2 instances, ECS containers, and S3 storage buckets let you set up the entire stack without having to bring everything together.

AWS EC2

You can run containers inside EC2 instances. Hence you can leverage the AWS security and management features.

AWS CloudWatch

This monitoring tool tracks every resource that AWS has to offer. It makes it easy to use third-party tools for monitoring such as sumo logic etc.

AWS CodePipeline

Code Pipeline is an essential feature from AWS, which highly simplifies the way you manage your CI/CD toolset. It integrates with tools such as **Jenkins, GitHub**, and CodeDeploy that enable you to visually control the flow of app updates from build to production.

DevOps Training Certification

DevOps training certification helps anyone to make a career as a DevOps engineer. DevOps certifications are available from Red Hat, Amazon web services, DevOps institution, and Microsoft academy.

Let's see all of these certifications one by one in brief such as:

Red Hat Certification

Red Hat offers a different level of certifications for DevOps professional as follows:

- Red Hat certificate of expertise in the Ansible automation.
- Red Hat certificate of expertise in Platform-as-a-service.
- Red Hat certificate of expertise in Container Administrator.
- Red Hat certificate of expertise in Configuration Management.
- Red Hat certificate of expertise in the Containerized Application Development.

Amazon Web Service Certification

This certificate tests you on how to use the most common DevOps patterns to develop, deploy, and maintain applications on AWS. It also evaluates you on the core principle of the DevOps methodology.

Amazon web service certificate has two requisites. First, the certification fee is $300, and the second time duration is 170 minutes.

DevOps Institution

The DevOps institution is a global learning community around emerging DevOps practices. This organization is setting the quality standards for DevOps competency-based qualifications.

Some offered certification courses are:

- DevOps Leader
- DevOps Test Engineer
- DevOps Foundation Certified
- DevOps Foundation
- Certified Agile Process Owner
- Certified Agile Service Manager
- Continuous Delivery Architecture

- DevSecOps Engineer

evOps vs Agile

DevOps and Agile are the two software development methodologies with similar aims, getting the end-product as quickly and efficiently as possible. While many organizations are hoping to employ these practices, there is often some confusion between both methodologies.

What does each methodology enclose? Where do they overlap? Can they work together, or should we choose one over the other?

Before move further, take a glance at DevOps and Agile.

What is DevOps?

The DevOps is a combination of two words, one is software Development, and second is Operations. This allows a single team to handle the entire application lifecycle, from development to **testing, deployment**, and **operations**. DevOps helps you to reduce the disconnection between software developers, quality assurance (QA) engineers, and system administrators.

DevOps promotes collaboration between Development and Operations team to deploy code to production faster in an automated & repeatable way.

DevOps helps to increase organization speed to deliver applications and services. It also allows organizations to serve their customers better and compete more strongly in the market.

DevOps can also be defined as a sequence of development and IT operations with better communication and collaboration.

DevOps has become one of the most valuable business disciplines for enterprises or organizations. With the help of DevOps, **quality**, and **speed** of the application delivery has improved to a great extent.

DevOps is nothing but a practice or methodology of making "**Developers**" and "**Operations**" folks work together. DevOps represents a change in the IT culture with a complete focus on rapid IT service delivery through the adoption of agile practices in the context of a system-oriented approach.

What is Agile?

The Agile involves continuous iteration of development and testing in the **SDLC** process. Both development and testing activities are concurrent, unlike the waterfall model. This software development method emphasizes on incremental, iterative, and evolutionary development.

It breaks the product into small pieces and integrates them for final testing. It can be implemented in many ways, such as **Kanban, XP, Scrum**, etc.

The Agile software development focus on the four core values, such as:

- Working software over comprehensive documentation.
- Responded to change over following a plan.
- Customer collaboration over contract negotiation.
- Individual and team interaction over the process and tools.

Below are some essential differences between the DevOps and Agile:

Parameter	DevOps	Agile
Definition	DevOps is a practice of bringing development and operation teams together.	Agile refers to the continuous iterative approach, which focuses on collaboration, customer feedback, small, and rapid releases.
Purpose	DevOps purpose is to manage end to end engineering processes.	The agile purpose is to manage complex projects.
Task	It focuses on constant testing and delivery.	It focuses on constant changes.
Team size	It has a large team size as it involves all the stack holders.	It has a small team size. As smaller is the team, the fewer people work on it so that they can move faster.
Team skillset	The DevOps divides and spreads the skill set between development and the operation team.	The Agile development emphasizes training all team members to have a wide variety of similar and equal skills.
Implementation	DevOps is focused on collaboration, so it does not have any commonly accepted framework.	Agile can implement within a range of tactical frameworks such as **safe, scrum**, and **sprint**.
Duration	The ideal goal is to deliver the code to production daily or every few hours.	Agile development is managed in units of sprints. So this time is much less than a month for each sprint.
Target areas	End to End business solution and fast delivery.	Software development.
Feedback	Feedback comes from the internal team.	In Agile, feedback is coming from the customer.
Shift left principle	It supports both variations left and right.	It supports only shift left.
Focus	DevOps focuses on operational and business readiness.	Agile focuses on functional and non-functional readiness.
Importance	In DevOps, developing, testing, and implementation all are equally important.	Developing software is inherent to Agile.
Quality	DevOps contributes to creating better quality with automation and early bug removal. Developers need to follow Coding and best Architectural practices to maintain quality standards.	The Agile produces better applications suites with the desired requirements. It can quickly adapt according to the changes made on time during the project life.
Tools	**Puppet, Chef, AWS, Ansible**, and team City OpenStack are popular DevOps tools.	**Bugzilla, Kanboard, JIRA** are some popular Agile tools.
Automation	Automation is the primary goal of DevOps. It works on the principle of maximizing efficiency when deploying software.	Agile does not emphasize on the automation.
Communication	DevOps communication involves specs and design documents. It is essential for the operational team to fully understand the software release and its network implications for the enough running the deployment process.	Scrum is the most common method of implementing Agile software development. Scrum meeting is carried out daily.
Documentation	In the DevOps, the process documentation is foremost because it will send the software to an operational team for deployment. Automation minimizes the impact of insufficient documentation. However, in the development of sophisticated software, it's difficult to transfer all the knowledge required.	The agile method gives priority to the working system over complete documentation. It is ideal when you are flexible and responsive. However, it can harm when you are trying to turn things over to another team for deployment.

Bash for DevOps

In the early days of computing, the computer that processed data or performed operations was separate from the tool that gave it the instructions to do the processing.

On the one hand, a terminal was used to send commands to the computer. On the other hand, we have a computer, which is hardware that processes the commands.

Today, some computers can provide commands AND perform computation via Graphical User Interface (GUI). However, accessing the command line or terminal can often be more efficient than using GUIs for certain tasks.

We can send commands via the terminal to programmatically accomplish these tasks. For example, working with files in the terminal is faster and more efficient than working with files in a graphical environment like windows explorer. We can also use the terminal to launch and execute open, reproducible tasks such as Jupyter Notebook, Python, and GIT.

Before knowing about Bash, we need to learn what a **shell** is. Shell is the primary program computers use to receive commanding code. These commands can be entered and executed via the terminal, which allows us to control the computer by typing commands with the keyboard, Instead of using buttons or dropdown menus in the GUI with the mouse or keyboard.

Bash

Bash is also known as the **"Bourne Again Shell."** It is the implementation of the shell which allows us to perform many tasks efficiently. We can quickly use Bash to perform operations on multiple files via the command line. We can also write and execute scripts in Bash as in Python, which can be executed across different operating systems. Using Bash in a terminal is a powerful way of interacting with computers, GUIs, and command lines. Bash is complementary; by knowing both, we can greatly expand the range of tasks we can accomplish with our computer.

With Bash commands, we can perform many tasks efficiently and automate and replicate the workforce across operating systems like Linux, Windows, etc.

The common tasks we can run on the command line include the following:

1. Checking and working on the current directory.
2. Changing the directory.
3. Making a new directory.
4. Extracting files.

5. Finding files on our computer.

Features of Bash

Working with terminal Bash provides us to:

1. Easily navigate our computer to access and manage files and folders. That is, we can easily navigate computer directories. We can work with many files and directories quickly and efficiently at once.
2. We can also run programs that provide more functionality at the command line, like GIT.
3. We can also launch programs from a specific directory on our computer, like a Jupyter Notebook.
4. Finally, with the help of Bash, we use repeatable commands for these tasks across many different operating like Windows, Mac, and Linux.

Before DevOps

It was a waterfall model and a traditional approach to building solutions. It is called waterfall because we bring out all the individual requirements and individual sections of a project and cascade off each other.

What is DevOps

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

Benefits of DevOps

Let us discuss some of the benefits of DevOps. They are:

1. Continuous delivery of software. It allows us to continuously release new features with the security and understanding that the software is high quality.
2. It allows the teams working on the software delivery within our organization to collaborate more effectively.
3. The deployment process moves from an event with a lot of stress and contingency plans to a much easier deployment.

4. The efficiency within the actual code that we are writing is the ability to scale up using the different tools available allows us to bring in and scale up and reduce the teams we have running the software as needed.
5. Errors can be fixed much earlier and quickly and caught before anything gets pushed out to the production environment.
6. We were looking to improve the security of the actual releases. So the actual concept of security is the center of all the work.
7. Finally, what allows us to reduce the number of errors is that there is much less manual intervention. There is greater reliance on scripted environments that we can test and validate for their security, reliability, and efficiency.

Brian Fox created the Unix shell and command language Bash for the GNU Project as a free software substitute for the Bourne shell. It was first made accessible in 1989, and since then, most Linux distributions, **Apple's macOS Mojave,** and earlier versions have adopted it as their default login shell. A variant is likewise accessible for **Windows 10** and the default client shell in **Solaris 11.**

Slam is an order processor that commonly runs in a text window where the client types orders that cause activities. Slam can likewise peruse and execute orders from a shell script document. Like all Unix shells, it upholds filename globbing (trump card coordinating), funneling, here reports, order replacement, fact7ors, and control structures for condition-testing and cycle. The watchwords, linguistic structure, powerfully perused factors, and other fundamental language highlights are completely replicated from sh. Different highlights, e.g., history, are duplicated from csh and ksh. Slam is a POSIX-consistent shell, yet with various expansions.

The shell, which is a play on the name of the Bourne shell it replaces and the idea of being **"born again,"** is an acronym for Bourne Again Shell.

A safety opening in Slam dating from variant 1.03 (August 1989), named Shellshock, was found toward the beginning of September 2014 and immediately prompted a scope of assaults across the Web. Patches to fix the bugs were made accessible not long after the bugs were distinguished.

Bash Scripting in DevOps

So every Management in development should master a scripting language as their first skill to help them handle servers, software, and hardware. They are fundamental to the growth of learning and are quite potent. There is still usage of Bash, yes. If you look at a normal Linux or UNIX system, you'll observe that it comes with several shell scripts.

Most contemporary Linux distributions still make use of Bash. It is the default shell for most system initialization, including the system V init scripts. It is a scripting language for the shell; you must know it to maintain a Linux server. Making GUI apps is hard, but it's worthwhile.

I have used the scripts for the specified objectives, amongst many others.

You may write a script to initialize anything as the system boots. Hence, manual labor is not required.

1. To enable/disable certain features, you may create a script that installs each need individually and builds the code based on user input.
2. We are combining the killing or starting from several programs.
3. Identify some patterns in a big database of files by observing it.
4. As a result, the list of things to automate continues.
5. Incredible uses

The bootup scripts (/etc/init.d)

1. For automating many computer maintenance tasks, such as user account creation, etc., technical staff.
2. Tools for downloading software packages, More information
3. Startup scripts for programs, particularly for unattended programs (e.g., started from cron or at)
4. Every user requiring automation

Why would someone use Bash?

Like other CLIs, Bash is used by computer programs that demand accuracy while handling files and data, especially when there are many. Data has to be searched, sorted, processed, or in some other way. Among the most typical uses for Bash are:

1. **System admins** use Bash to manage systems methodically and consistently. System administrators use Bash to get into systems and examine system configurations and internet connectivity to debug systems that are not performing as planned or expected. Moreover, system administrators use Bash scripts to maintain and set up systems, monitor operating systems, and distribute software patches and updates.
2. Bash is used for many development tasks by **software professionals**. Bash is used for many development activities by software professionals.
3. Using Bash, automating **software development** chores like code compilation, source code debugging, change Management, and software testing is possible.
4. To test, configure, and improve network performance on business networks, **network engineers** utilize Bash.
5. Bash is a programming language that **computer scientists** use to administer research systems and conduct research on them.
6. To communicate with their computers, run applications, and manage them, **power users and hobbyists** alike utilize Bash.

Bash may be used to build shell scripts in addition to being often used interactively. A Bash script can be utilized to automate almost any computer process, and Bash scripts can be run immediately or on a routine basis.

How, in all actuality, does Bash work?

From the outset, Slam gives off an impression of being a straightforward order/reaction framework, where clients enter orders, and Slam returns the outcomes after those orders are run. In any case, Slam is likewise a programming stage, and clients are empowered to compose programs that acknowledge information and produce yield utilizing shell orders in shell scripts.

One of the essential slam orders, ls, does a certain something: list catalog contents. Without anyone else, this order records the names of documents and subdirectories in the ongoing working registry.

The ls order has various boundaries that alter how the outcomes are shown. A few frequently utilized boundaries utilized with the ls order include:

-l

Utilize a more drawn-out, nitty gritty posting configuration to incorporate record consents, document proprietor, gathering, size, and date/season of creation.

-a

List all records and subdirectories, even those customarily expected to be covered.

-s

Show the size of each document.

-h

Show record and subdirectory sizes in comprehensible arrangement utilizing K, M, G, etc., to demonstrate kilobytes, megabytes, and gigabytes.

-R

It tells us the Recursive posting of all documents and subdirectories under the ongoing working catalog.

By using the output with one command as the input for another, Bash makes it possible to combine commands. Even with the -R argument to specify that the listing ought to be recursive, for example, one might use this command to list every file on a file system:

1. user@hostname:/$ 1s -1ashR

The above command returns too many records for humans to understand, especially from the system root directory easily.

The grep command only returns files and subdirectories with filenames that contain the specified text pattern when the pipe symbol (vertical bar, or "|") is used to funnel output from the directory listing into it. This order:

This program may be used to find a specific file since it only returns files with the text:

1. user@hostname:/$ 1s -1ashR |grep 'filename.txt'

By using the bash command line, it is considerably simpler to perform the following interactively:

managing files and directories; monitoring network configuration;

We are modifying a configuration file (or any other text file) and comparing two files.

Examples

Some examples of bash commands are:

- Basic commands are often executed either alone or in conjunction with arguments and variables. For instance, the ls command accepts variables for the directories or files listed in addition to arguments.
- Pipes connect the output of one or more commands to the input of other commands.
- Lists are used to allow users to execute several instructions sequentially.
- Compound commands facilitate script writing and include conditional structures and loops (for repeating a command a certain amount of times).
- One unique bash feature that isn't always accessible with other CLIs is command-line editing. By hitting the up arrow key, Bash's command history may be retrieved. So, it makes it simpler to execute a command again exactly. These previous commands can also be changed at the command line by copying, pasting, deleting, or changing a previous command using special keys.
- The usage of Bash requires some initial understanding for new users because it is one of the fundamental tools for current system and network management. A system administrator from 1992 who traveled through time and learned Bash could resume working on a modern Linux system immediately.

What is Bash Mechanization?

The Bash shell is a strong Linux shell that permits top-to-bottom robotization of tedious undertakings. Not exclusively is the Bash Linux shell a superb decision for DevOps. Still, data sets and test designs are the same; consistently, clients can profit from gradually scholarly, steadily expanding Bash abilities. Bash is likewise a prearranging and coding language that develops on you. We have been effectively coding in Bash beginning around 2012 and have utilized it significantly longer than that.

Bash additionally fits a wide range of use spaces and use cases. For instance, you can undoubtedly use it for Huge Information taking care, and shockingly it appears to loan

itself incredibly well to this undertaking because of the horde of text-handling instruments accessible inside it or accessible as simple to introduce bundles. It is also extremely appropriate for reinforcement and information base planning and support, dealing with enormous record stockpiling arrangements, computerizing web servers, and more.

At whatever point the following issue introduces itself, a little exploration in a web crawler, or the different Stackoverflow sites, will rapidly yield an answer for the issue, yet a possible chance to develop and learn. So, it is very much like insight to an individual learning the manager vi where similar holds; at whatever point an issue introduces itself, the arrangement is close by.

This smaller-than-usual series comprises three pieces, which is first; in it, we will check out Bash robotization and prearranging essentials.

Terraform Destroy Command

Terraform is an open-source infrastructure as code (IaC) tool. It enables users to manage infrastructure resources declaratively. It allows the user to define, provision, and operate cloud resources across multiple providers using a single configuration file. One of the most important commands in Terraform is "terraform destroy." This command is used to delete all the resources created using the Terraform configuration. In this article, we are going to learn briefly the Terraform destroy command, how it works, and some examples to help you understand its usage.

What is Terraform Destroy Command?

Terraform destroy command is used to remove all the resources that were created using the Terraform configuration. It is the opposite of the "terraform apply" command. The terraform apply command is used to create resources. The destroy command removes all the resources specified in the configuration files and will prompt the user for confirmation before deleting the resources.

When to Use Terraform Destroy Command?

There is some scenario where we can use terraform destroy command. These are as follows.

1. When the user wants to delete specific resources that are no longer required or have been replaced by new ones, we use destroy command.
2. When the user wants to destroy the entire infrastructure stack, including the networking, computing, and storage resources, then we use destroy command.
3. When the user wants to delete all the resources created using Terraform, we use destroy command to clean up the environment or start over with a new configuration.

How does Terraform Destroy Command Work?

There are some ways by which we can implement the terraform destroy command. These ways are as follows.

1. First, the terraform reads the Terraform configuration files to determine the resources that need to be deleted.
2. Then it prompts the user for confirmation before deleting the resources.
3. After the user's confirmation, it deletes the resources in the reverse order of their creation so that resources depending on other resources are deleted last.
4. After deleting the resources, It updates the Terraform state file to remove the deleted resources from the state.

Example of Terraform Destroy Command

Let's take an example to understand briefly the terraform destroy command.

1. First, we have to create a new directory, and then we have to navigate to that directory. This can be done by the below commands.

1. mkdir my-terraform && cd my-terraform

2. Then we have to create a new terraform configuration file and name it "main.tf". Then we have to write the below code in the "main.tf" file.

```
1. provider "aws" {
2.
3. region = "us-east-1"
```

```
4. }
5.
6. resource "aws_instance" example" {
7.
8. ami = "ami-0c55b159cbfafe1f0"
9.
10. instance_type = "t2.micro"
11. }
```

This code creates an AWS EC2 instance with the help of specified AMI and instance type.

3. Then we have to initialize the working directory with the help of the below commands.

1. terraform init

4. Then we have to create the resources with the help of the below commands.

1. terraform apply

5. Then we have to verify the instances that were created previously. That can be done by the below commands.

1. terraform show

6. Then we have to delete the resources with the help of the below commands.

1. terraform destroy

Then a confirmation is prompted for the user before deleting the resource. When the user approves it at that moment, the resources are successfully destroyed.

Advantages of Terraforming Destroy Command

There are some advantages of terraforming destroy command. These are as below.

1. **Easy cleanup:** The Terraform destroy command makes it easy to clean up the resources that were created using Terraform. This helps to avoid cluttering your infrastructure with unused resources.
2. **Efficiency:** The Terraform destroy command removes all the resources in one go without leaving any errors. This saves time and ensures that resources are removed efficiently.
3. **Cost savings:** Removing unused resources can save you money on cloud provider bills. The Terraform destroy command helps you to identify and remove these resources.
4. **Consistency:** The Terraform destroy command helps to maintain consistency in your infrastructure by removing all resources that were created by Terraform.

Disadvantages of Terraforming Destroy Command

There are some disadvantages of terraforming destroy command. These are as below.

1. **Accidental destruction:** The Terraform destroy command can lead to the accidental destruction of resources if used incorrectly. It'sIt's important to use this command with caution and double-check the resources that will be destroyed.
2. **Data loss:** If the user is not careful, then the user can lose data by accidentally destroying resources that contain important data. It'sIt's important to make backups of data before using the Terraform destroy command.
3. **Time-consuming:** Destroying resources can take a long time, depending on the number and complexity of the resources. This can be an inconvenience when the user needs to quickly clean up resources.
4. **Dependency issues:** The Terraform destroy command can have dependency issues, where a resource cannot be destroyed because it'sit's dependent on another resource. This can cause problems with cleanup and may require additional work to resolve.

Terraform For Loop

Terraform is an infrastructure-as-code (IaC) tool. It is used to create and manage infrastructure resources in a declarative way. It enables the user to define the

infrastructure as a code and automates creating and managing infrastructure resources. One of the most useful features of Terraform is the ability to use loops in the code to create and manage multiple resources at once. In this article, we are going to explore Terraform loops, including their syntax, examples, advantages, and disadvantages.

Terraform Loops

Terraform loops allow the user to iterate over a set of values and create multiple resources with the same configuration. There are two types of loops supported by Terraform: for_each and count.

1. For_Each Loop

The for_each loop creates multiple resources with the same configuration, each with a unique name or ID. It works by iterating over a map or set of strings, and for each item in the map or set; it creates a new resource.

Syntax

```
resource "resource_type" "resource_name" {
  for_each = {key1 = value1, key2 = value2, ...}
  # Resource configuration here
}
```

Example

```
variable "regions" {
  type = list(string)
  default = ["us-west-1", "us-west-2", "us-east-1"]
}

resource "aws_instance" "ec2" {
  for_each = toset(var.regions)
  ami = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id = "subnet-123456"
```

```
  availability_zone = "${each.value}a"
}

output "instance_public_ips" {
  value = {
    for instance_id, instance in aws_instance.ec2 :
      instance.id => instance.public_ip
  }
}
```

Output:

```
$ terraform apply

...

Outputs:

instance_public_ips = {
  "i-0a1b2c3d4e5f6g7h8" = "54.111.222.333"
  "i-9j8h7g6f5e4d3c2b1" = "52.444.555.666"
  "i-a1b2c3d4e5f6g7h8" = "52.777.888.999"
}
```

Explanation

In the above example, we use a for_each loop to create an EC2 instance in each region specified in the region's variable. The for_each loop is applied to the aws_instance resource and creates an instance for each element in the regions list.

The output block uses a for expression to create a mapping of instance IDs to public IP addresses. This output is useful for debugging or verifying the instances that were created.

2. Count Loop

The count loop is used to create a fixed number of resources with the same configuration. It defines a numeric count value and creates a new resource for each count value.

Syntax:

```
resource "resource_type" "resource_name" {
  count = number_of_resources
  # Resource configuration here
}
```

Example

```
variable "count" {
  default = 3
}

resource "aws_instance" "example" {

count = var.count
  ami = "ami-0c55b159cbfafe1f0"

instance_type = "t2.micro"
}

output "public_ips" {
  value = [
    for instance in aws_instance.example : instance.public_ip
  ]
}

output "print_public_ips" {
  value = [
    for instance in aws_instance.example : "Instance ${instance.id} public IP: ${instance.public_ip}"
  ]
}
```

Output:

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

public_ips = [
  "54.152.137.33",
  "54.90.163.105",
  "54.236.204.234",
]
print_public_ips = [
  "Instance 0 public IP: 54.152.137.33",
  "Instance 1 public IP: 54.90.163.105",
  "Instance 2 public IP: 54.236.204.234",
]
```

Explanation

- In the above example, we take a variable called "count" with a default value of 3. Then we create multiple instances of an EC2 instance using the count loop with the specified AMI and instance type.
- Next, we define two outputs with the help of the output block. The first output, "public_ips," outputs an array of the public IP addresses of the created instances. We use a for loop with the "aws_instance.example" resource to iterate over each instance and extract its public IP.
- The second output, "print_public_ips," outputs an array of strings containing each instance's ID and public IP address. We use a similar for loop to iterate over each instance and create a string that combines the instance ID and public IP address.

Advantages of Terraform Loops

There is some benefit to using the terraform loop. These are as follows.

1. **Increased efficiency:** Terraform loops allow users to create multiple resources in a single code block with the same configuration. This can save time and increase efficiency when managing large numbers of resources.

2. **Improved readability:** Terraform loops can make the code more readable and easier to understand by reducing the amount of repetitive code needed to create multiple resources.
3. **Simplified resource management:** Terraform loops can simplify the process of managing resources by allowing the user to group related resources together in a single block of code.

Disadvantages of Terraform Loops

There are also some disadvantages to using the terraform loop. These are as follows.

1. **Complexity:** Terraform loops can add complexity to the code. Suppose the user is unfamiliar with the syntax and logic required to use them effectively.
2. **Limited functionality:** While Terraform loops can be powerful, they are limited in their ability to handle complex use cases. In some cases, the user may need to use a combination of loops and other Terraform features to achieve the desired outcome.
3. **Lack of flexibility:** Terraform loops are not always flexible enough to handle dynamic changes to your infrastructure. For example, if the user needs to add or remove resources from a loop, you may need to modify the loop's configuration, re-run the Terraform plan, and apply commands.

Terraform Format

Terraform is an open-source infrastructure-as-code tool. It allows the developer to define and manage their infrastructure declaratively. This means that the developer can describe their infrastructure with the help of a high-level language, and Terraform will create, modify, and delete the necessary resources to achieve the desired state.

One of the most important aspects of using Terraform is understanding its format. In this article, we are going to explore the Terraform format, including its syntax, examples, and best practices.

Terraform Syntax

Terraform uses its own domain-specific language (DSL) to describe infrastructure. The language is based on HashiCorp Configuration Language (HCL), designed to be both human-readable and machine-friendly.

The syntax of Terraform files consists of blocks, arguments, and values. Blocks are also called resources or modules, arguments are also known for the properties of those resources or modules, and values are the actual values of those properties.

For example, with the help of the below command, we can create a simple Terraform file that creates an AWS EC2 instance:

```
provider "aws" {

region = "us-west-2"
}

resource "aws_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance_type = "t2.micro"
}
```

In the above file, the provider block defines the AWS provider and region to use, while the resource block defines the EC2 instance to create. The AMI and instance_type arguments define the instance's properties, and the values for those properties are provided on the right-hand side of the equals sign.

Terraform Examples

Terraform can manage a wide range of infrastructure resources, including compute instances, networking components, databases, and more.

Below are examples of terraforming.

1. Creating an Azure virtual machine:

Program

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name
 = "example-resource-group"
  location = "westus2"
}

resource "azurerm_virtual_network" "example" {
  name = "example-vnet"
  address_space = ["10.0.0.0/16"]
  location = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
}

resource "azurerm_subnet" "example" {
  name = "example-subnet"
  resource_group_name = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
  address_prefixes = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "example" {
  name = "example-nic"
  location = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  ip_configuration {
    name = "example-ipconfig"
    subnet_id = azurerm_subnet.example.id
    private_ip_address_allocation = "Dynamic"
  }
```

```
}

resource "azurerm_virtual_machine" "example" {
  name = "example-vm"
  location = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  network_interface_ids = [azurerm_network_interface.example.id]
  vm_size = "Standard_DS1_v2"

  storage_image_reference {
    publisher = "Canonical"
    offer
= "UbuntuServer"
    sku
= "18.04-LTS"
    version
= "latest"
  }

  storage_os_disk {
    name = "example-osdisk"
    caching = "ReadWrite"
    create_option = "FromImage"
  }
}

output "vm_ip_address" {
  value = azurerm_network_interface.example.private_ip_address
}
```

Output:

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

vm_ip_address = "10.0.1.4"
```

Explanation

The above program creates a virtual machine in Azure with the help of the following resources:

- A resource group
- A virtual network with a subnet
- A network interface with an IP configuration
- A virtual machine with reference to the network interface and an OS disk created from the specified image.

The output block at the end of the code specifies the private IP address of the network interface, which can be printed to the console using the terraform output command after running terraform apply.

Terraform Best Practices

When the programmer is working with Terraform, it is important to follow best practices to ensure the infrastructure is secure, scalable, and maintainable. Here are a few best practices to keep in mind:

1. The programmer must Use modules to modularize your code and promote reuse.
2. The programmer must Use variables and parameterized modules to make the code more flexible.
3. The programmer must Use version control to manage the Terraform code and collaborate with others.
4. The programmer must Use Terraform Cloud or Enterprise to manage the state and execute Terraform runs.

5. The programmer must Use Terraform's built-in functionality to manage secrets and sensitive data, such as the sensitive argument or the terraform.tfvars file.

Advantages of Terraform

There are some advantages of using Terraform. These are as follows.

1. **Declarative language:** It uses declarative language to define infrastructure, which makes it easier to understand and maintain.
2. **Multi-cloud support:** It supports multiple cloud providers, allowing the programmer to manage infrastructure across different clouds with the same tool.
3. **Modular design:** it allows the programmer to create reusable modules that can be used across different projects, making it easier to standardize infrastructure.
4. **Version control:** the code can be versioned with popular tools like Git, allowing the programmer to track changes and collaborate with other team members.
5. **Infrastructure as code:** it allows the programmer to treat infrastructure as code, which means the programmer can version, test, and deploy infrastructure changes just like the programmer does with application code.

Disadvantages of Terraform

There are also some disadvantages of using the terraform. These are as follows.

1. **Learning curve:** It has a steep learning curve, especially if the programmer is new to infrastructure such as code or declarative languages.
2. **Limited functionality:** While Terraform can manage many different types of resources, some advanced features are still unavailable.
3. **Dependencies:** Terraform resources can have dependencies on other resources, which can make it difficult to manage changes or deletions.
4. **State management:** Terraform requires managing state files, which can be challenging when working with a team or in a distributed environment.
5. **Complexity:** Terraform can become complex when managing large or complex infrastructures, which can make it difficult to debug or troubleshoot issues.

Terraform Output Command

Terraform is a widely used infrastructure-as-code (IaC) tool .it allows the programmer to define, provision, and manage cloud resources in a declarative manner. One of the powerful features of Terraform is the "output" command, which helps the programmer extract and share values from the Terraform configurations. In this article, we are going to learn about the Terraform output command, including its purpose, syntax, usage, and examples, to help you understand how to leverage this powerful tool in the IaC workflows.

Purpose of Terraform Output Command

The main purpose of Terraform output command is to expose values from the Terraform configurations. These output values can be dynamically generated during provisioning or retrieved from existing resources. Then these Outputs can be used to capture information such as resource IDs, IP addresses, DNS names, or any other relevant data that the programmer may need to use in subsequent Terraform configurations, scripts, or tools. Those Outputs provide a way to make the outputs of the infrastructure deployments accessible and usable outside of Terraform, allowing for greater flexibility and integration with other parts of the infrastructure ecosystem.

Syntax of Terraform Output Command

The terraform output command can be written with the help of the below command.

```
output "<name>" {
  value = <expression>
}
```

In the above syntax:

- **<name>:-**It is a unique name for the output, which can consist of letters, numbers, underscores, and hyphens.
- **<expression>:-** It is the value or expression that the programmer wants to expose as the output. This can be a reference to a Terraform resource or data source, a computed value, or a combination of multiple values using interpolation syntax.

Usage of Terraform Output Command

When the programmer wants to utilize the functionality of Terraform output command, then the programmer just has to include the code in the Terraform configuration file (.tf) within a module or root module block. After applying the Terraform configuration using the terraform apply command, the programmer can view the values of the outputs using the terraform output command followed by the <name> of the output.

Examples of Terraform Output Command

Let's understand the output command with the help of some examples.

Program 1 (Extracting VPC ID)

Let's consider the scenario in which the programmer deals with an Amazon Web Services (AWS) VPC with Terraform, and the programmer want to extract the VPC ID for further use in the infrastructure. Here's an example of how the programmer can use the Terraform output command to perform the operation:

Code

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name = "example-vpc"
  }
}

output "vpc_id" {
  value = aws_vpc.example.id
}
```

Output:

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

vpc_id = vpc-0123456789abcdef
```

Explanation

The above program defines an AWS VPC resource named "example" with a given CIDR block and tags. We then define an output called "vpc_id" that references the "id" attribute of the "aws_vpc.example" resource. This makes the VPC ID available as an output that can be accessed using the "vpc_id" name.

Program 2 (Composing Output Values)

Code

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name = "example-vpc"
  }
}

resource "aws_subnet" "example_subnet" {
count = 2

cidr_block = "10.0.${100 + count.index.index}.0/24"
vpc_id     = aws_vpc.example.id

tags = {
  Name = "example-subnet-${count.index.index + 1}"
}
}
```

```
19.
20. output "subnet_ids" {
21.   value = aws_subnet.example_subnet.*.id
22. }
23.
24. output "subnet_cidr_blocks" {
25.   value = aws_subnet.example_subnet.*.cidr_block
26. }
```

Output:

```
Outputs:

subnet_cidr_blocks = [
  "10.0.101.0/24",
  "10.0.102.0/24",
]
subnet_ids = [
  "subnet-1234567890",
  "subnet-0987654321",
]
```

Explanation

In the above code, we define an AWS VPC with a given CIDR block and tags and two AWS subnets with dynamically generated CIDR blocks and tags. We then define two outputs: "subnet_ids" and "subnet_cidr_blocks." The "subnet_ids" output uses the aws_subnet.example_subnet.*.id reference to capture the IDs of all the subnets created by the "example_subnet" resource. The "subnet_cidr_blocks" output uses the aws_subnet.example_subnet.*.cidr_block reference to capture the CIDR blocks of all the subnets. These outputs can be accessed and used in subsequent Terraform configurations or scripts.

Benefits of Using Terraform Output Command

There is also some benefit of using the terraform output command. These are as follows.

1. **Reusability:** The Outputs command allows the programmer to capture and reuse values from the Terraform configurations in subsequent configurations or scripts. This provides a feature to reuse and reduces code duplication, making the IaC code more modular and maintainable.
2. **Flexibility:** The Output command provide a way to make the outputs of the infrastructure deployments accessible and usable outside of Terraform. This command also allows for greater flexibility in integrating with other parts of the infrastructure ecosystem, such as provisioning tools, configuration management systems, or monitoring and logging solutions.
3. **Dynamicity:** The Output command is also evaluated during the Terraform apply command, which means that they can capture dynamically generated values or computed values based on the current state of your infrastructure. This makes outputs a powerful tool for extracting and managing dynamic information in your IaC workflows.
4. **Clarity:** The Output command also allows the programmer to expose specific values from the Terraform configurations and clarify what information is intended to be used outside of Terraform. This enhances the readability and maintainability of your IaC code, as it provides a clear separation of concerns between the inputs and outputs of the infrastructure.
5. **Debugging:** The Output command can also be used for debugging and troubleshooting. The programmer can use the Terraform output command to view the values of outputs after applying the Terraform configuration, helping the programmer to verify that the expected values are being generated and used correctly in your infrastructure.

Best Practice

There are some important points that the programmer should keep in mind during the use of terraform commands.

1. We must ensure the output names are clear, descriptive, and reflect the output value. This will help make the IaC code more readable and maintainable.
2. The Interpolation syntax allows the programmer to combine values from multiple resources or data sources, generating more complex outputs. This can help the

programmer to capture and reuse more specific and dynamic information from the infrastructure.

3. Sensitive outputs are used when using the terraform output command. It helps the programmer protect secret or private information from being exposed. Also, we have to ensure to mark sensitive outputs appropriately using the sensitive = true argument.
4. The documentation and comments in the IaC code need to explain what each output value represents, how it is generated, and how it can be used. This will help others who work with the code understand its purpose and usage.

Terraform Output

Terraform is a popular Infrastructure as Code (IaC) tool. It enables users to automate cloud resource provisioning, configuration, and management across multiple cloud platforms. It is also an open-source tool that supports various cloud providers such as AWS, Google Cloud, Azure, and many more. One of the essential features of Terraform is the ability to output the results of the infrastructure creation process.

In this article, we are going to learn Terraform output with examples and its advantages and disadvantages.

Terraform Output

Terraform output is a command that is used to display the values of resources created during the Terraform provisioning process. The command is used to retrieve the attributes of a resource and returns the values that can be accessed and used by other tools or scripts. It is also a valuable feature as it enables automating different processes or scripts that require access to the created resources.

The output command in Terraform is a simple command that can be executed from the command line interface (CLI). The output values are stored in a state file, and the state file can be accessed and used by other tools or scripts.

Example

```
provider "aws" {
  region = "us-west-2"
}
```

```
4.
5.  resource "aws_instance" "example" {
6.    ami           = "ami-0c55b159cbfafe1f0"
7.    instance_type = "t2.micro"
8.
9.    tags = {
10.     Name = "example-instance"
11.   }
12. }
13.
14. output "instance_id" {
15.   value = aws_instance.example.id
16. }
17.
18. output "public_ip" {
19.   value = aws_instance.example.public_ip
20. }
```

Output:

```
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

instance_id = i-1234567890abcdef0
```

Explanation

The above Terraform code creates an AWS EC2 instance in the us-west-2 region using an ami-0c55b159cbfafe1f0 Amazon Machine Image (AMI) and a t2.micro instance type. It also shows an output variable named "instance_id" and that output variable retrieves the ID of the created EC2 instance. When Terraform is run with Terraform applied, it creates the EC2 instance and displays the output variable "instance_id" with the value of the created EC2 instance ID.

Advantages

There are several advantages of terraform output. These are as follows.

1. **Automation:** It enables the automation of other processes or scripts that require access to the created resources. The output values can be used as inputs for additional tools or scripts, thereby simplifying the process of managing cloud infrastructure.
2. **Consistency:** It also ensures that the infrastructure is consistent across all environments. The output values are stored in the state file, which is versioned and stored in a central location. This ensures that the infrastructure is consistent across all environments, including development, staging, and production.
3. **Reusability:** It also makes it easy to reuse infrastructure resources. The output values can be used as inputs for other Terraform modules, making it easy to reuse infrastructure resources across different projects.

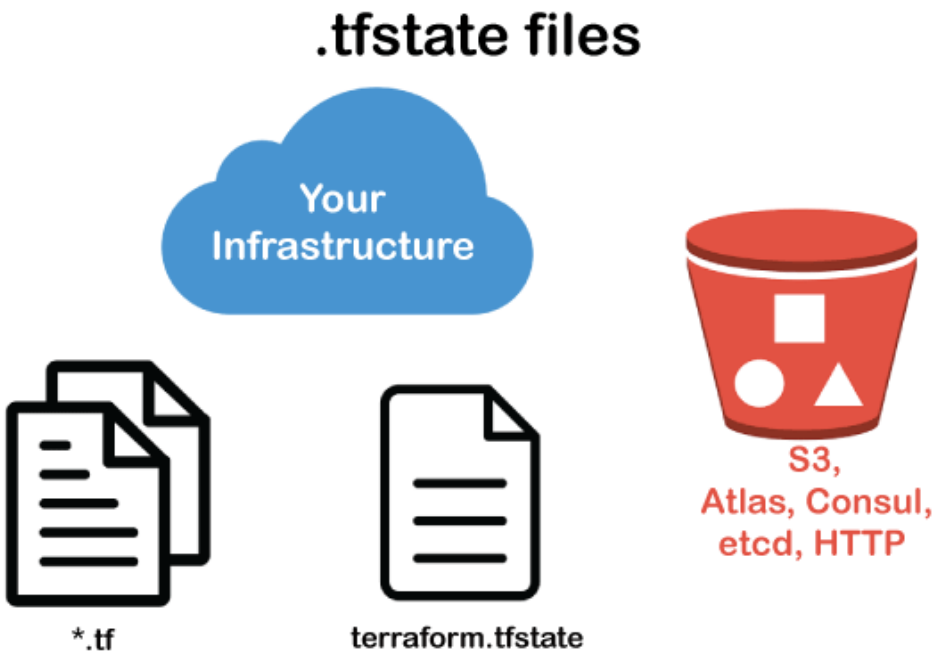
Disadvantages:

Terraform output has some disadvantages. These are listed below:

1. **Security:** If we do not properly secure the information, then there may be a chance of exposing sensitive information if not properly secured. The output values can include sensitive information such as access keys, passwords, and secrets. Ensuring the state file contains the output values is essential to prevent unauthorized access.
2. **Complexity:** Terraform output can add complexity to the Terraform configuration file. The output variables must be defined correctly, and the syntax must be correct. This can make the Terraform configuration file more complex and challenging to manage.

Terraform tfstate

When we are using Terraform to manage infrastructure as code, tfstate is an important thing that cannot be ignored. Tfstate files contain critical information about the infrastructure and its state. The management of tfstate is very important for effective infrastructure management. In this article, we are going to learn what Terraform tfstate is, why it is important, and how to manage it effectively.



What is Terraform tfstate?

Terraform tfstate is a JSON file that contains information about the current state of infrastructure that Terraform manages. It also contains the details of the resources that has created, modified, or destroyed by terraform in a particular environment. Tfstate files also include metadata, and that metadata describes the resources' dependencies and relationships, which Terraform uses to manage infrastructure changes effectively.

Why is Terraform tfstate Important?

Tfstate files are very much essential for several reasons. Firstly, they help Terraform to track the current state of the infrastructure and identify changes that have occurred since the last deployment. This enables Terraform to determine which resources need to be created, updated, or destroyed to bring the infrastructure to the desired state.

Secondly, tfstate files help Terraform to manage infrastructure changes safely and effectively. Terraform uses the information in the tfstate file to calculate the changes required to bring the infrastructure to the desired state. Terraform then applies these changes in the correct order, considering any dependencies or relationships between resources.

Finally, tfstate files are essential for collaboration between multiple team members or across different environments. Tfstate files provide a consistent view of the infrastructure state, ensuring that all team members are working with the same information. They also enable team members to work on different parts of the infrastructure simultaneously without conflicts.

Managing Terraform tfstate:

The Terraform tfstate can be managed effectively to ensure infrastructure changes are handled safely and efficiently. There are some best practices for managing Terraform tfstate.

1. **Use a remote backend:** One of the best ways to manage tfstate is to use a remote backend. A remote backend stores the tfstate file in a remote location, such as AWS S3 or Azure Blob Storage. This helps multiple team members to access the tfstate file from different locations, making collaboration easier. It also ensures that the tfstate file is stored securely and can be easily recovered if necessary.
2. **Version control of the tfstate file:** It is essential to use version control in the tfstate file to track changes over time. This enables team members to see who made changes, when they were made and why they were made. It also allows the team to revert to previous versions of the tfstate file if necessary.
3. **Lock tfstate files:** Tfstate files should be locked to prevent multiple team members from making changes simultaneously. Locking the tfstate file helps that only one team member can make changes at a time.
4. **Use remote state data sources:** Using remote state data sources enables the programmer to reference resources created by other Terraform configurations. This is particularly useful while managing complex infrastructure. It also allows the programmer to reference resources across different configurations and environments.
5. **Use state backups:** Regularly backing up the tfstate file ensures that it can be easily recovered if necessary. Backups should be stored in a secure location and regularly tested to ensure they can be restored if required

Advantages of Terraform tfstate

There are some advantages of using terraform tfstate. These are as follows.

1. **Accurate Tracking of Infrastructure State:** The tfstate file can track the current state of the infrastructure accurately. Terraform uses this information to identify the changes required to bring the infrastructure to the desired state, making it easier to manage infrastructure changes safely and effectively.
2. **Safe Infrastructure Changes:** Tfstate files can enable Terraform to manage infrastructure changes safely and effectively. Terraform uses the information in the tfstate file to calculate the required modifications and apply them in the correct order, ensuring that dependencies and relationships between resources are considered.
3. **Collaboration:** Tfstate files also can enable team members to work on different parts of the infrastructure simultaneously without conflicts. They provide a consistent view of the infrastructure state, ensuring that all team members are working with the same information.
4. **Disaster Recovery:** Backing up the tfstate file regularly ensures that it can be easily recovered if necessary, minimizing downtime in the event of a disaster.

Disadvantages of Terraform tfstate

There are also some disadvantages of using terraform tfstate. These disadvantages are as follows,

1. **Complexity:** When the developer wants to manage large and complex infrastructure, at that time, it becomes more complex and large in size. So Managing the tfstate files effectively requires significant knowledge and experience with Terraform.
2. **Security:** It also contains sensitive information about the infrastructure, making it a potential security risk. So It is very much essential to store tfstate files securely and manage access control to ensure that only authorized personnel can access them.
3. **Version Control:** Version controlling the tfstate file can be more challenging, especially when managing multiple environments and configurations. So It is very much essential to implement robust version control processes to ensure that changes are tracked accurately and effectively.

4. **Locking:** The programmer needs to Lock the tfstate file so that it can result in conflicts if team members need to make changes simultaneously. It is essential to implement effective locking processes to minimize the risk of conflicts

Git Tutorial

Git tutorial provides basic and advanced concepts of Git and GitHub. Our Git tutorial is designed for beginners and professionals.

Git is a modern and widely used **distributed version control** system in the world. It is developed to manage projects with high speed and efficiency. The version control system allows us to monitor and work together with our team members at the same workspace.

This tutorial will help you to understand the distributed version control system Git via the command line as well as with GitHub. The examples in this tutorial are performed on **Windows**, but we can also perform same operations on other operating systems like **Linux (Ubuntu)** and **MacOS**.

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**.

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**.

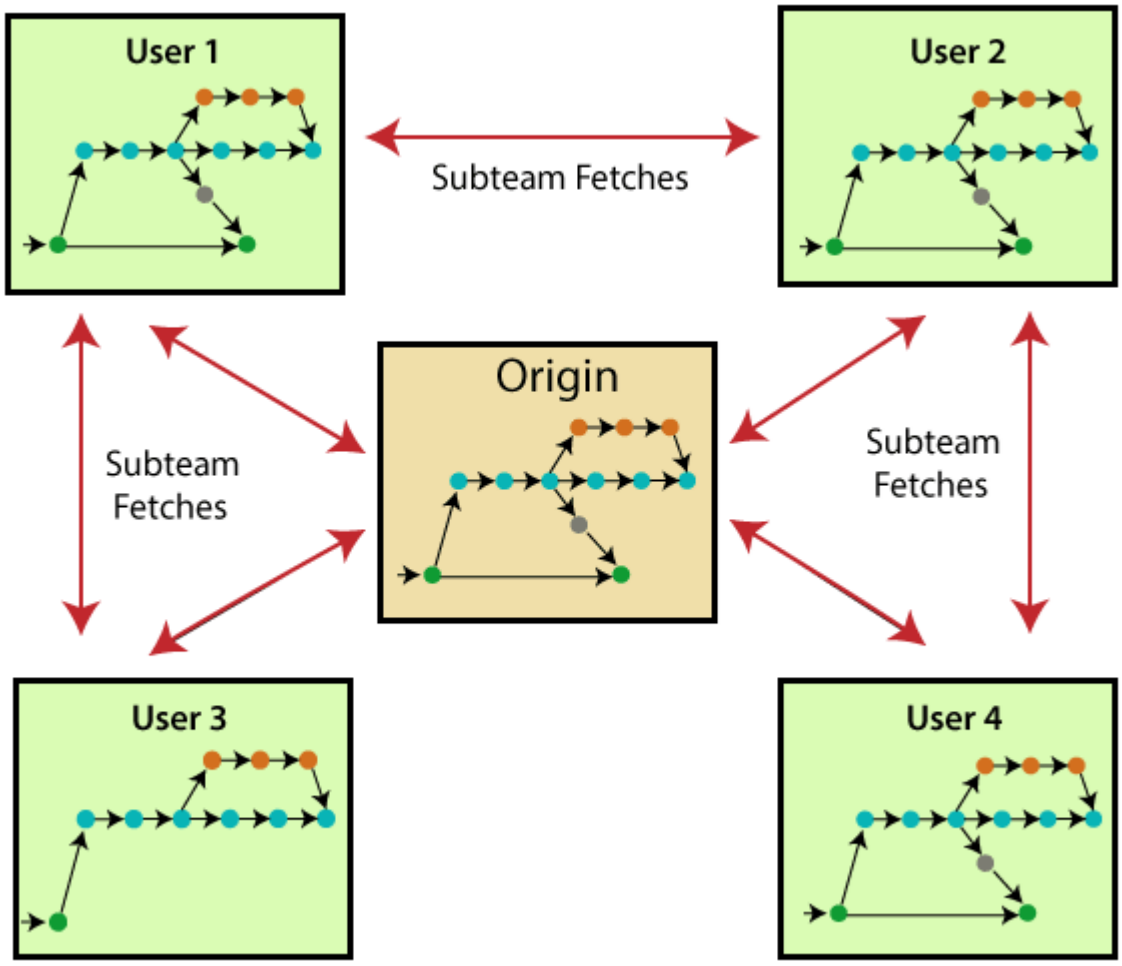
Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Features of Git

Some remarkable features of Git are as follows:

- **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.



- **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.

- **Supports non-linear development**

  Git supports **seamless branching and merging**, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

- **Branching and Merging**

  Branching and merging are the **great feature**s 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:

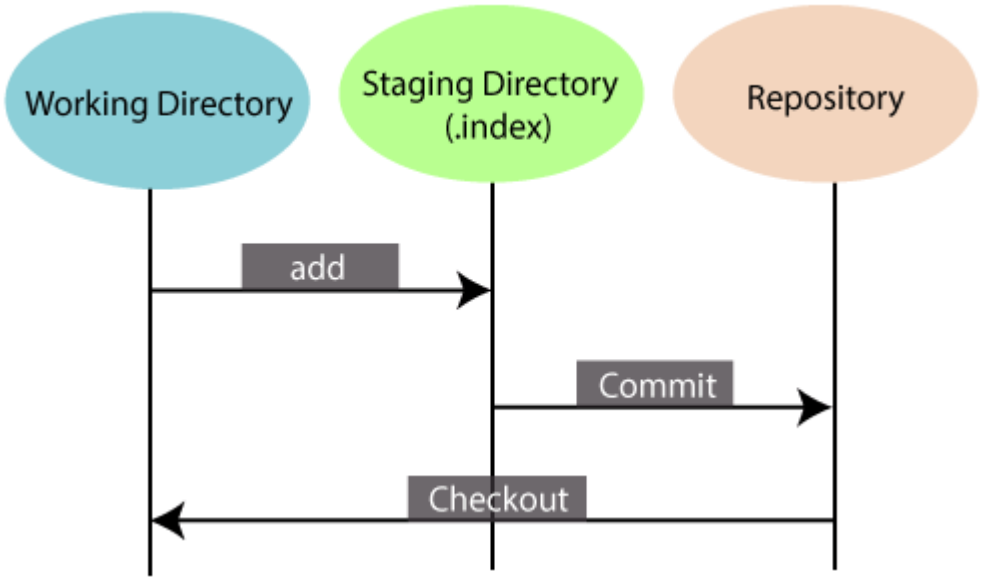
  - We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
  - We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
  - We can create a **demo branch** for the experiment and 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.

- **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.

- **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.



  Another feature of Git that makes it apart from other SCM tools is that **it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.**

- **Maintain the clean history**

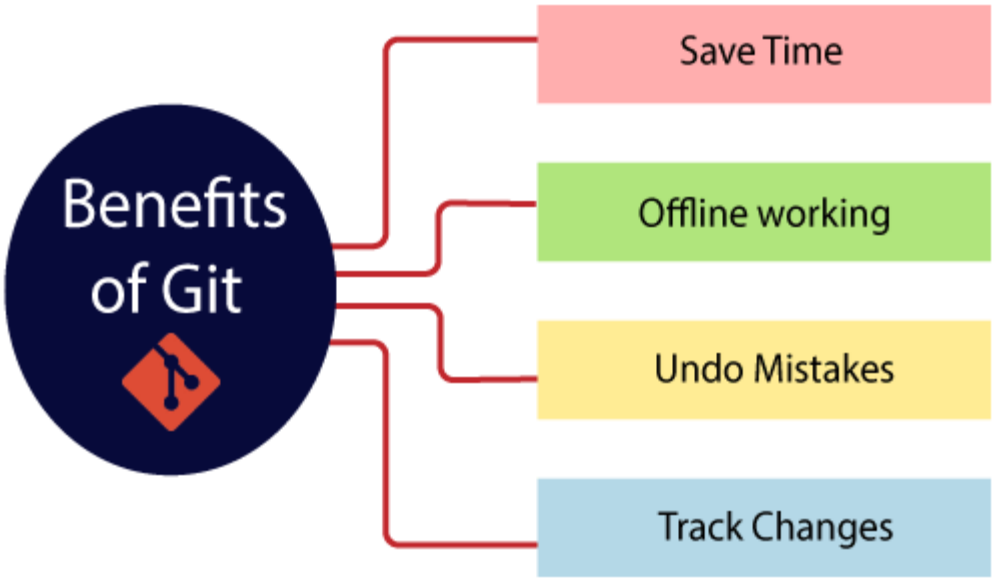
  Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

Benefits of Git

A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes

from the repository and continue to work with the updates that you added to the project files.

Some **significant benefits** of using Git are as follows:

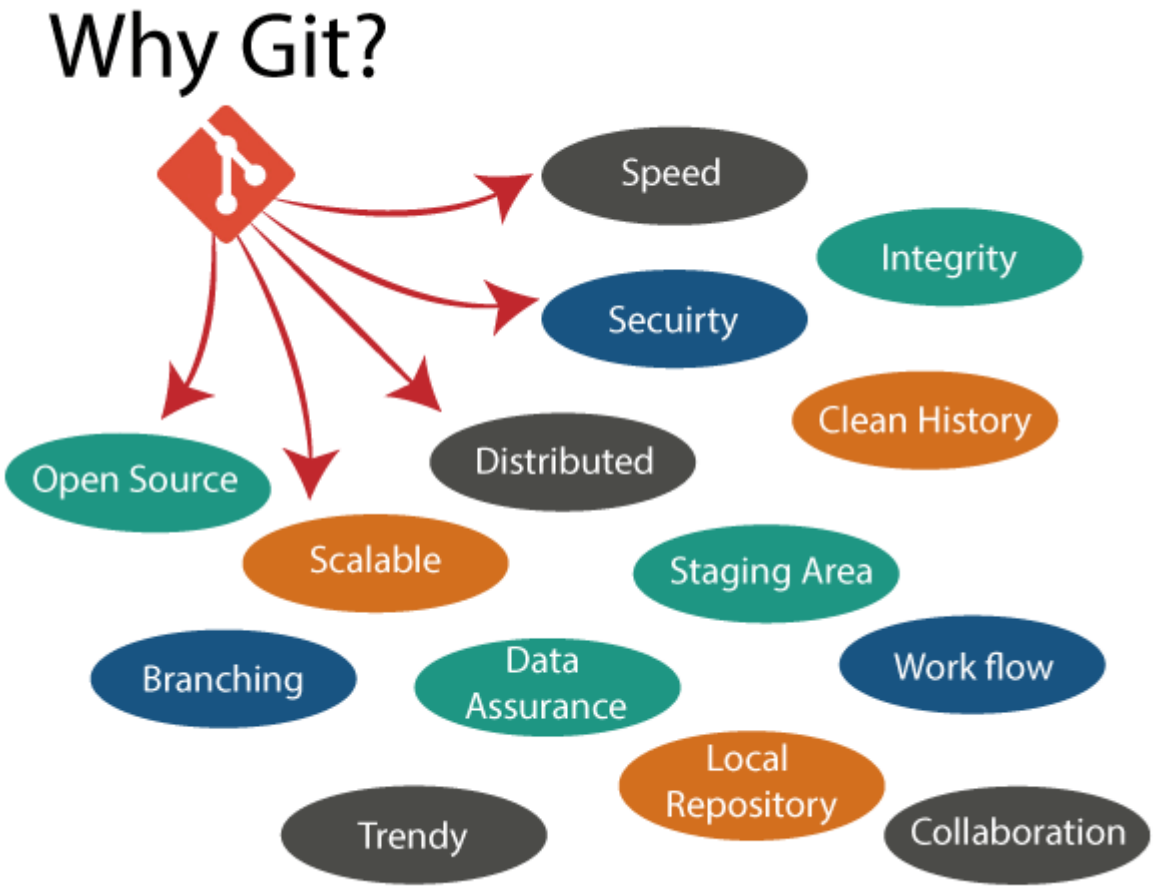


- **Saves Time**
  Git is lightning fast technology. Each command takes only a few seconds to execute so we can save a lot of time as compared to login to a GitHub account and find out its features.
- **Offline Working**
  One of the most important benefits of Git is that it supports **offline working**. If we are facing internet connectivity issues, it will not affect our work. In Git, we can do almost everything locally. Comparatively, other CVS like SVN is limited and prefer the connection with the central repository.
- **Undo Mistakes**
  One additional benefit of Git is we can **Undo** mistakes. Sometimes the undo can be a savior option for us. Git provides the undo option for almost everything.
- **Track the Changes**
  Git facilitates with some exciting features such as **Diff, Log,** and **Status**, which

allows us to track changes so we can **check the status, compare** our files or branches.

Why Git?

We have discussed many **features** and **benefits** of Git that demonstrate the undoubtedly Git as the **leading version control system**. Now, we will discuss some other points about why should we choose Git.



- **Git** **Integrity**
  Git is **developed to ensure** the **security** and **integrity** of content being version controlled. It uses checksum during transit or tampering with the file system to confirm that information is not lost. Internally it creates a checksum value from the contents of the file and then verifies it when transmitting or storing data.

- **Trendy Version Control System**
  Git is the **most widely used version control system**. It has **maximum projects** among all the version control systems. Due to its **amazing workflow** and features, it is a preferred choice of developers.
- **Everything is Local**
  Almost All operations of Git can be performed locally; this is a significant reason for the use of Git. We will not have to ensure internet connectivity.
- **Collaborate to Public Projects**
  There are many public projects available on the GitHub. We can collaborate on those projects and show our creativity to the world. Many developers are collaborating on public projects. The collaboration allows us to stand with experienced developers and learn a lot from them; thus, it takes our programming skills to the next level.
- **Impress Recruiters**
  We can impress recruiters by mentioning the Git and GitHub on our resume. Send your GitHub profile link to the HR of the organization you want to join. Show your skills and influence them through your work. It increases the chances of getting hired.

What is GitHub?

GitHub is a Git repository hosting service. GitHub also facilitates with many of its features, such as access control and collaboration. It provides a Web-based graphical interface.

GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.

It offers both **distributed version control and source code management (SCM)** functionality of Git. It also facilitates with some collaboration features such as bug tracking, feature requests, task management for every project.

Features of GitHub

GitHub is a place where programmers and designers work together. They collaborate, contribute, and fix bugs together. It hosts plenty of open source projects and codes of various programming languages.

Some of its significant features are as follows.

- Collaboration
- Integrated issue and bug tracking
- Graphical representation of branches
- Git repositories hosting
- Project management
- Team management
- Code hosting
- Track and assign tasks
- Conversations
- Wikisc

Benefits of GitHub

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.

The key benefits of GitHub are as follows.

- It is easy to contribute to open source projects via GitHub.
- It helps to create an excellent document.
- You can attract recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.
- You can track changes in your code across versions.

Difference between git and gitHub

Programming language wordings are very intuitive these days. By hearing the name of a particular language, we start imagining what all it will be.

Java and Javascript are very similar to the names ham and hamster, the logo of python is intertwined with the image of snakes.

So, someone looking at git and github would find any apparent connection between them. Let us see git and github in detail with the differences between them.

Git

There are many words to define git, but it is an open-source distributed version control system in simpler words.

Let us break each component in the definition and understand it.

- **Open-source -** A type of computer software released under a specific license. The users are given permissions to use the code, modify the code, give suggestions, clone the code to add new functionality. In other words, if the software is open-source, it is developed collaboratively in a public manner. The open-source softwares is cheaper, more flexible, and lasts longer than an authority or a company. The products in the source code include code, documents, formats for the users to understand and contribute to it. Using open-source a project can be expanded to update or revise the current features. Unix and Linux are examples of open-source softwares.
- **Control system -** The work of a control system is to track the content. In other words, git is used to storing the content to provide the services and features to the user.
- **Version Control system -** Just like an app has different updates due to bugs and additional feature addition, version changes, git also supports this feature. Many developers can add their code in parallel. So the version control system easily manages all the updates that are done previously.
  Git provides the feature of branching in which the updated code can be done, and then it can be merged with the main branch to make it available to the users. It not

only makes everything organized but keeps synchronization among the developers to avoid any mishap. Other examples of version control systems are Helix core, Microsoft TFS, etc.

- **Distributed version control system -** Here distributed version control system means if a developer contributes to open source, the code will also be available in his remote repository. The developer changes his local repository and then creates a pull request to merge his changes in the central repository. Hence, the word distributed means the code is stored in the central server and stored in every developer's remote system.

Why is git needed?

When a team works on real-life projects, git helps ensure no code conflicts between the developers. Furthermore, the project requirements change often. So a git manages all the versions. If needed, we can also go back to the original code. The concept of branching allows several projects to run in the same codebase.

GitHub

By the name, we can visualize that it is a Hub, projects, communities, etc. GitHub is a Git repository hosting service that provides a web-based graphical interface. It is the largest community in the world. Whenever a project is open-source, that particular repository gains exposure to the public and invites several people to contribute.

The source code of several projects is available on github which developers can use in any means.

Using github, many developers can work on a single project remotely because it facilitates collaboration.

Features of gitHub

- Using github the project managers can collaborate, review and guide the developers regarding any changes. This makes project management easy.
- The github repositories can be made public or private. Thus allowing safety to an organization in case of a project.
- GitHub has a feature of pull requests and issues in which all the developers can stay on the same page and organize.
- All the codes and their documentation are in one place in the same repository. Hence it makes easy code hosting.
- There are some special tools that github uses to identify the vulnerabilities in the code which other softwares do not have. Hence there is safety among the developers from code start till launch.
- Github is available for mobile and desktops. The UI is so user-friendly that it becomes straightforward to get comfortable with and use it.

Git vs SVN

- Apache Subversion or **SVN is one of the most popular centralized version control systems**. Now, SVN's popularity is on the decrease, but there are still millions of projects stored in it. It can continue to be actively maintained by an open-source community. In SVN, you can check out a single version of the repository. It stores data in a central server. The drawback of the SVN is, it has the entire history on a local repository which limits you. You can only do commits, diffs, logs, branches, merges, file annotations, etc.

- While, **Git is a popular distributed version control system**, which means that you can clone your repository. Thus you can get a complete copy of your entire history of that project. This means you can access all your commits.

- **Git has more advantages than SVN**. It is much better for those developers who are not always connected to the master repository. Also, it is much faster than SVN.
- To better understand the differences between Git and Subversion. Let's have a look at following significance points.

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.

Git vs Mercurial

Mercurial and Git both are two quite similar and most popular distributed version control systems. Their strengths and weaknesses make them ideal for different use cases. Both tools use a directed acyclic graph to store history.

Mercurial is a distributed source control management tool. It is free and open-source. It can handle projects of any size and offers an easy and intuitive interface.

Today, Git has more than 31 million users and is owned by Microsoft. Since the last decade, the Git has become the standard for most development projects.

Mercurial still has a handful tool of large development organizations. Some software development giants like Facebook, Mozilla, and World Wide Web Consortium are using it. But it only has approx 2 % of the VCS market share. Comparatively, Git has covered more than 80% market share.

Both version control systems, i.e., Mercurial and Git are distributed version control systems (DVCS).

To better understand the similarities and differences between Git and Mercurial, let's have a look at the following points.

Git	Mercurial
Git is a little bit of complex than Mercurial.	Mercurial is simpler than Git.
No VCS are entirely secured, but Git offers many functions to enhance safety.	Mercurial may be safer for fresher. It has more security features.
Git has a powerful and effective branching model. Branching in Git is better than Branching in Mercurial.	Branching in Mercurial doesn't refer the same meaning as in Git.
Git supports the staging area, which is known as the index file.	There is no index or staging area before the commit in Mercurial.
The most significant benefit with Git is that it has become an industry-standard, which means more developers are familiar with it.	Mercurial's significant benefit is that it's easy to learn and use, which is useful for less-technical content contributors.
Git needs periodic maintenance for repositories.	It does not require any maintenance.
It holds Linux heritage.	It is python based.
Git is slightly slower than Mercurial.	It is faster than Git.
Git supports the unlimited number of parents.	Mercurial allows only two parents.

Git Version Control System

A version control system is a software that tracks changes to a file or set of files over time so that you can recall specific versions later. It also allows you to work together with other programmers.

The version control system is a collection of software tools that help a team to manage changes in a source code. It uses a special kind of database to keep track of every modification to the code.

Developers can compare earlier versions of the code with an older version to fix the mistakes.

Benefits of the Version Control System

The Version Control System is very helpful and beneficial in software development; developing software without using version control is unsafe. It provides backups for uncertainty. Version control systems offer a speedy interface to developers. It also allows

software teams to preserve efficiency and agility according to the team scales to include more developers.

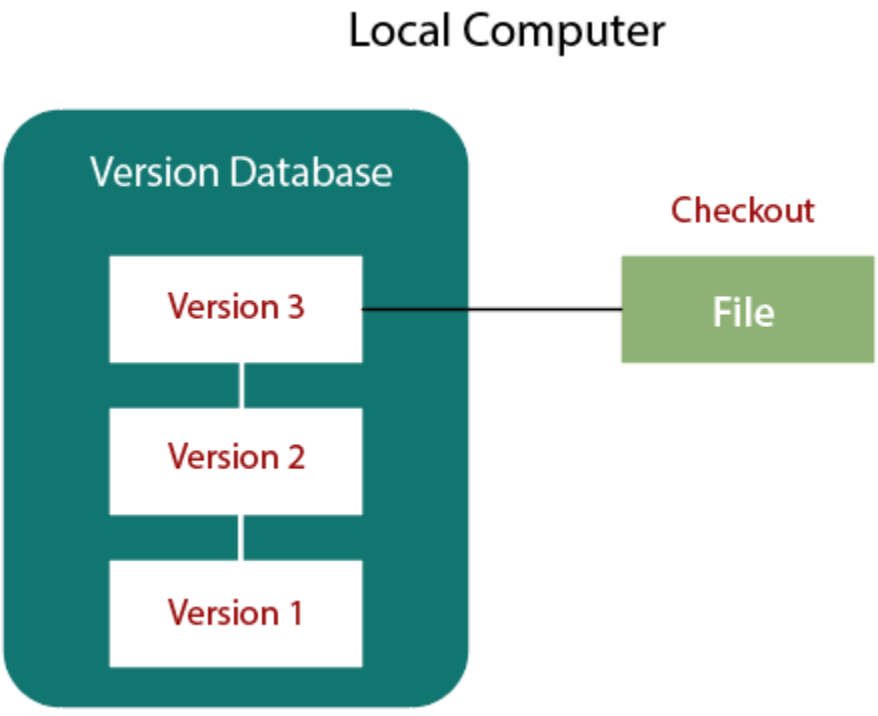
Some key benefits of having a version control system are as follows.

- Complete change history of the file
- Simultaneously working
- Branching and merging
- Traceability

Types of Version Control System

- Localized version Control System
- Centralized version control systems
- Distributed version control systems

Localized Version Control Systems



The localized version control method is a common approach because of its simplicity. But this approach leads to a higher chance of error. In this approach, you may forget which

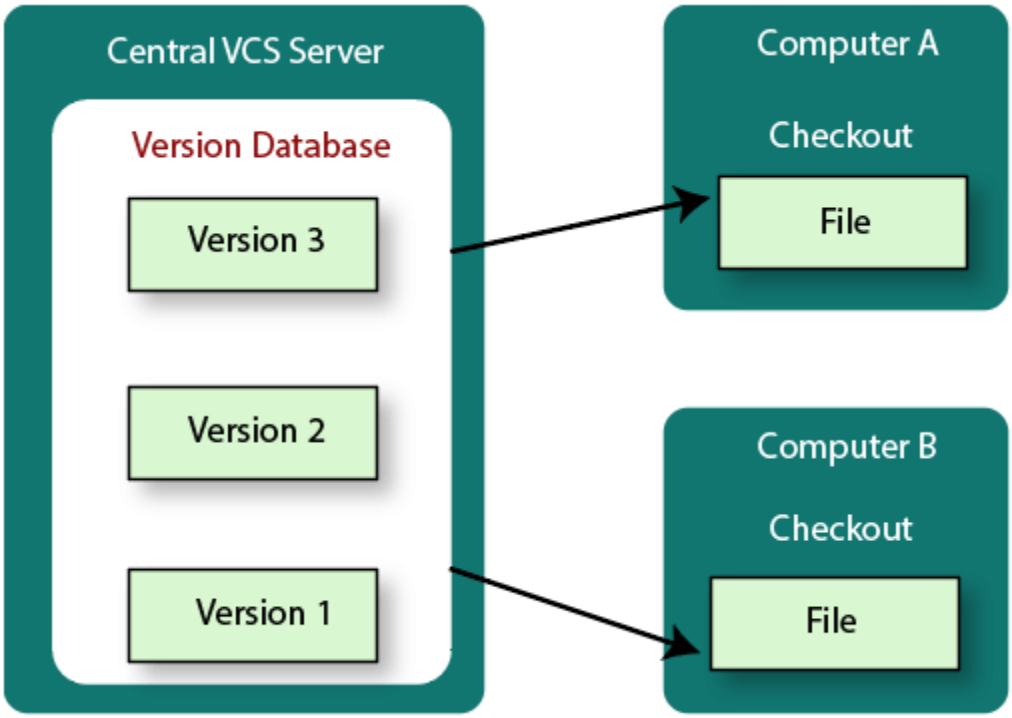
directory you're in and accidentally write to the wrong file or copy over files you don't want to.

To deal with this issue, programmers developed local VCSs that had a simple database. Such databases kept all the changes to files under revision control. A local version control system keeps local copies of the files.

The major drawback of Local VCS is that it has a single point of failure.

Centralized Version Control System

The developers needed to collaborate with other developers on other systems. The localized version control system failed in this case. To deal with this problem, Centralized Version Control Systems were developed.



These systems have a single server that contains the versioned files, and some clients to check out files from a central place.

Centralized version control systems have many benefits, especially over local VCSs.

- Everyone on the system has information about the work what others are doing on the project.

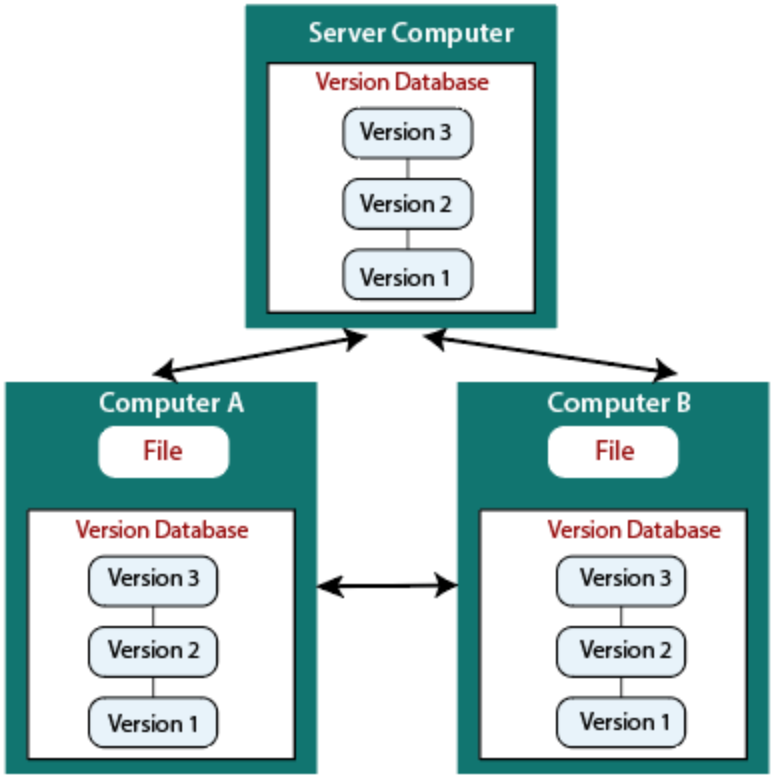
- Administrators have control over other developers.
- It is easier to deal with a centralized version control system than a localized version control system.
- A local version control system facilitates with a server software component which stores and manages the different versions of the files.

It also has the same drawback as in local version control system that it also has a single point of failure.

Distributed Version Control System

Centralized Version Control System uses a central server to store all the database and team collaboration. But due to single point failure, which means the failure of the central server, developers do not prefer it. Next, the Distributed Version Control System is developed.

In a Distributed Version Control System (such as Git, Mercurial, Bazaar or Darcs), the user has a local copy of a repository. So, the clients don't just check out the latest snapshot of the files even they can fully mirror the repository. The local repository contains all the files and metadata present in the main repository.



DVCS allows automatic management branching and merging. It speeds up of most operations except pushing and pulling. DVCS enhances the ability to work offline and does not rely on a single location for backups. If any server stops and other systems were collaborating via it, then any of the client repositories could be restored by that server. Every checkout is a full backup of all the data.

These systems do not necessarily depend on a central server to store all the versions of a project file.

Difference between Centralized Version Control System and Distributed Version Control System

Centralized Version Control Systems are systems that use **client/server** architecture. In a centralized Version Control System, one or more client systems are directly connected to a central server. Contrarily the Distributed Version Control Systems are systems that use **peer-to-peer** architecture.

There are many benefits and drawbacks of using both the version control systems. Let's have a look at some significant differences between Centralized and Distributed version control system.

Centralized Version Control System	Distributed Version Control System
In CVCS, The repository is placed at one place and delivers information to many clients.	In DVCS, Every user has a local copy of the repository in place of the central repository on the server-side.
It is based on the client-server approach.	It is based on the client-server approach.
It is the most straightforward system based on the concept of the central repository.	It is flexible and has emerged with the concept that everyone has their repository.
In CVCS, the server provides the latest code to all the clients across the globe.	In DVCS, every user can check out the snapshot of the code, and they can fully mirror the central repository.
CVCS is easy to administrate and has additional control over users and access by its server from one place.	DVCS is fast comparing to CVCS as you don't have to interact with the central server for every command.
The popular tools of CVCS are **SVN** (Subversion) and **CVS**.	The popular tools of DVCS are **Git** and **Mercurial**.
CVCS is easy to understand for beginners.	DVCS has some complex process for beginners.
If the server fails, No system can access data from another system.	if any server fails and other systems were collaborating via it, that server can restore any of the client repositories

How to Install Git on Windows

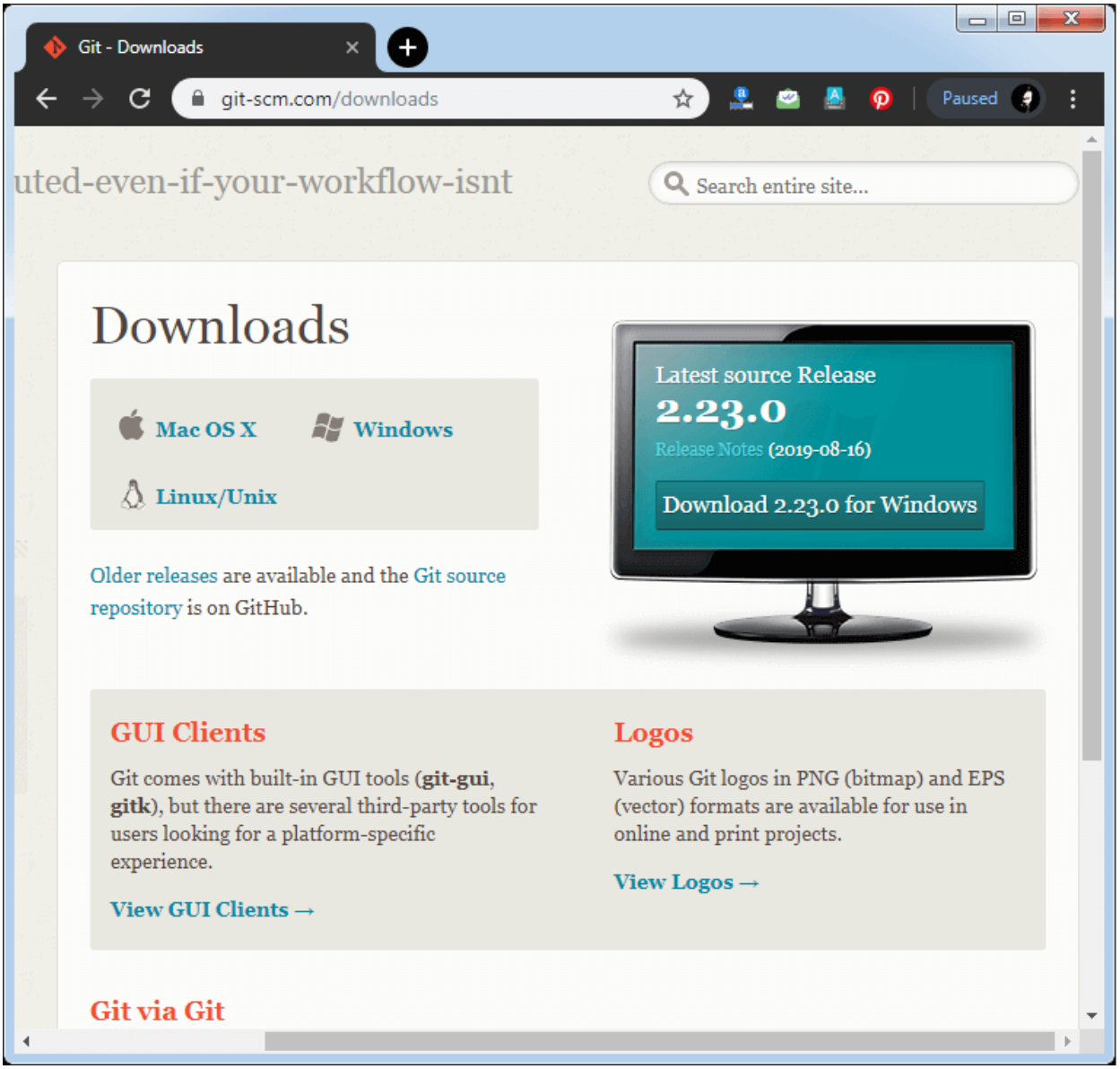
To use Git, you have to install it on your computer. Even if you have already installed Git, it's probably a good idea to upgrade it to the latest version. You can either install it as a package or via another installer or download it from its official site.

Now the question arises that how to download the Git installer package. Below is the stepwise installation process that helps you to download and install the Git.

How to download Git?

Step1

To download the Git installer, visit the Git's official site and go to download page. The link for the download page is https://git-scm.com/downloads. The page looks like as



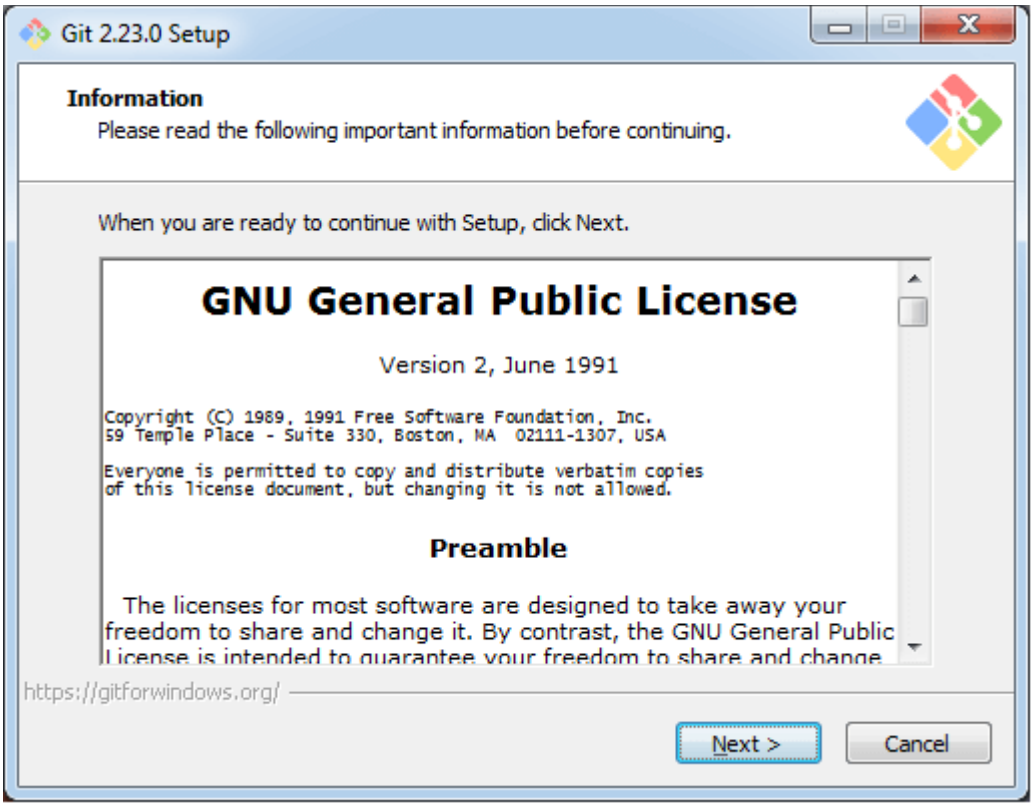
Click on the package given on the page as **download 2.23.0 for windows**. The download will start after selecting the package.

Now, the Git installer package has been downloaded.

Install Git

Step2

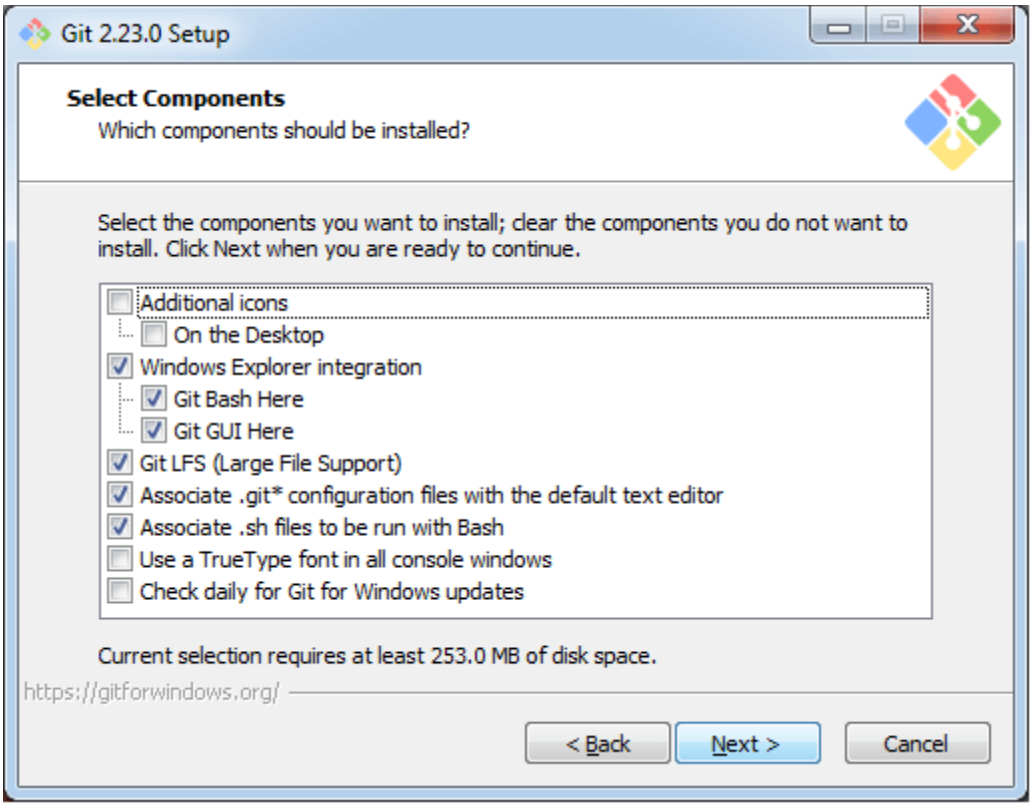
Click on the downloaded installer file and select **yes** to continue. After the selecting **yes** the installation begins, and the screen will look like as



Click on **next** to continue.

Step3

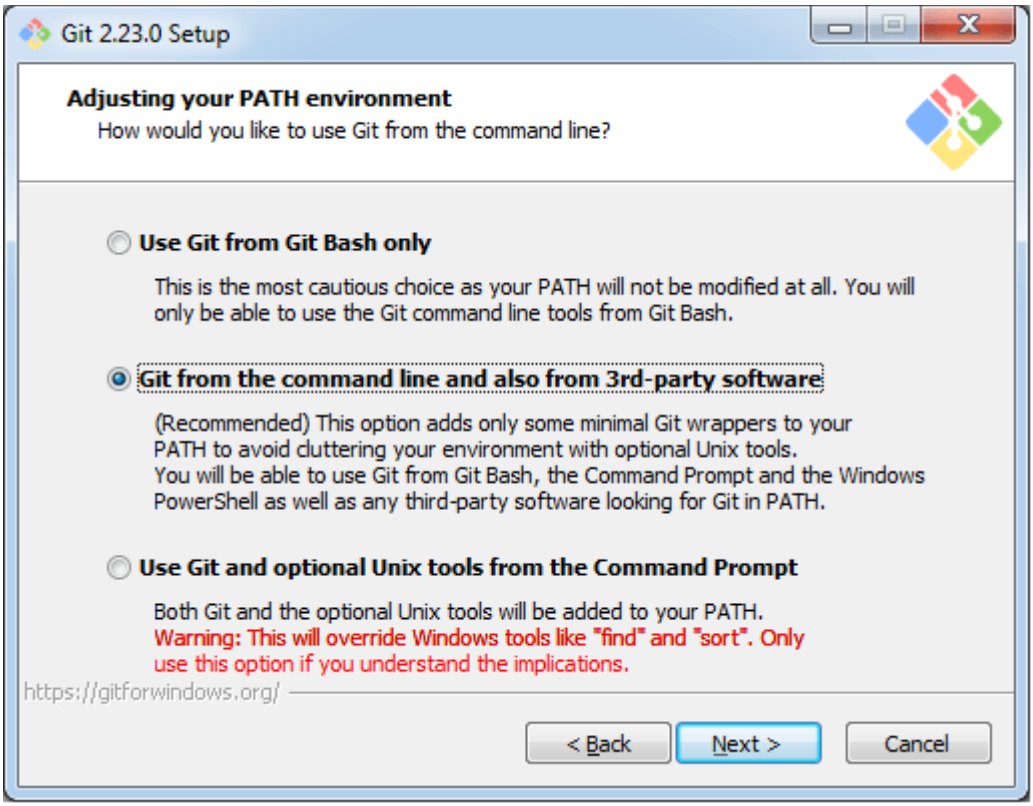
Default components are automatically selected in this step. You can also choose your required part.



Click next to continue.

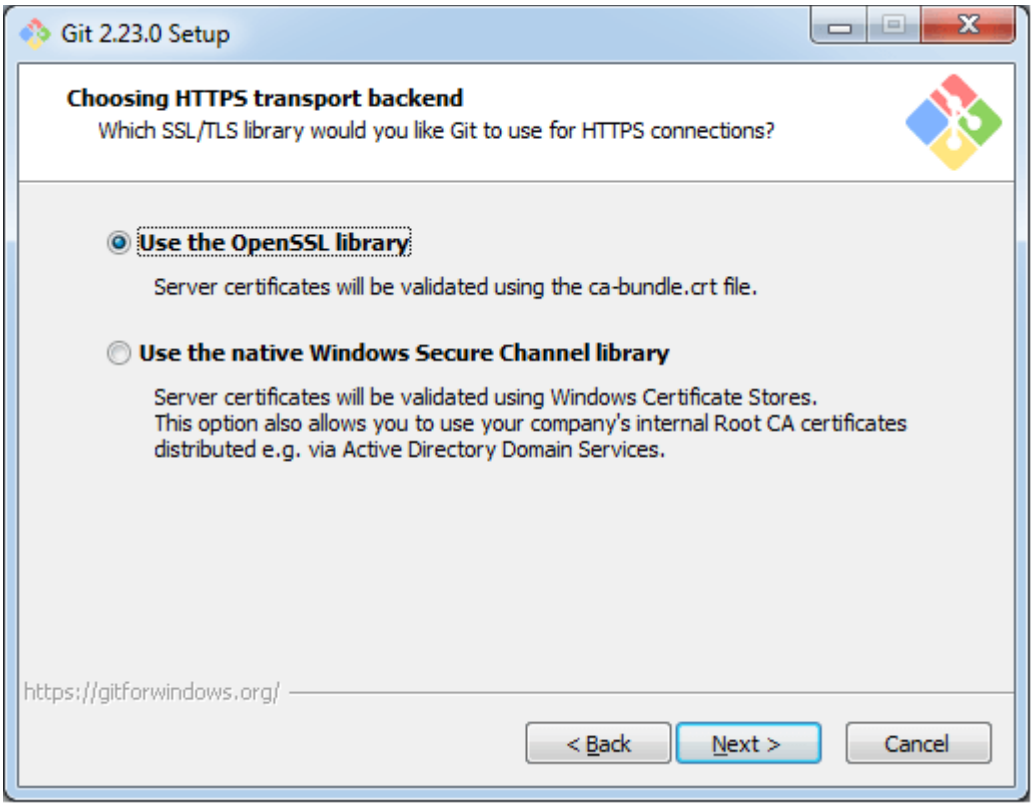
Step4

The default Git command-line options are selected automatically. You can choose your preferred choice. Click **next** to continue.



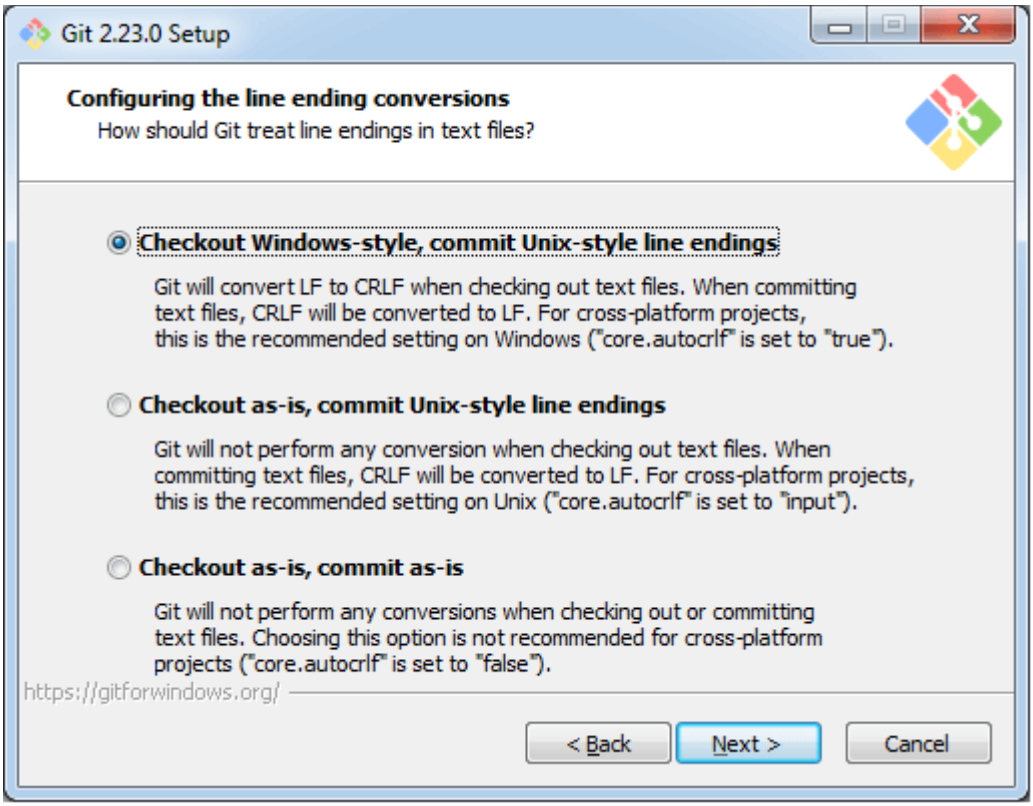
Step5

The default transport backend options are selected in this step. Click **next** to continue.



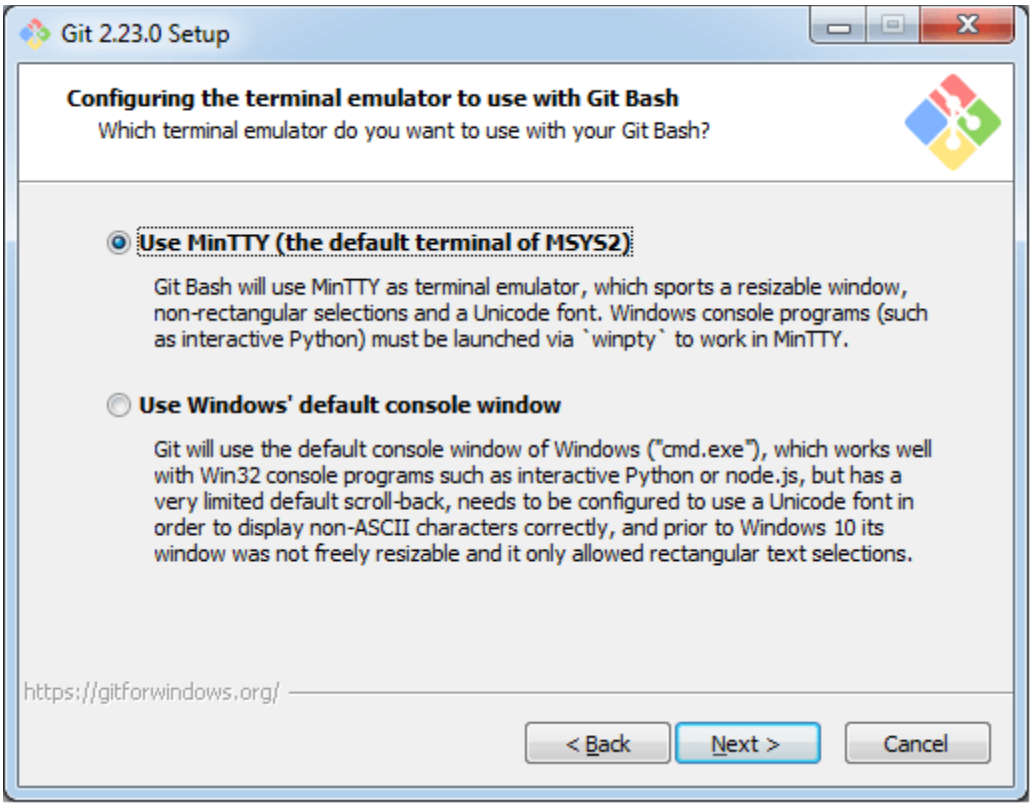
Step6

Select your required line ending option and click next to continue.



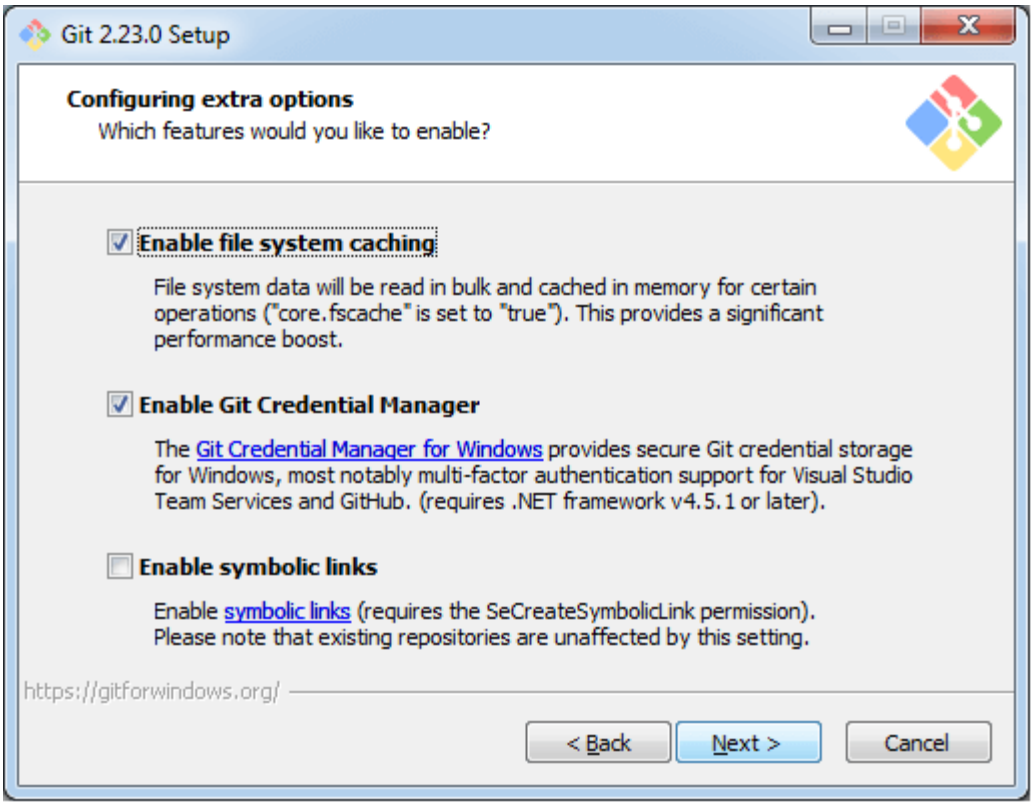
Step7

Select preferred terminal emulator clicks on the **next** to continue.



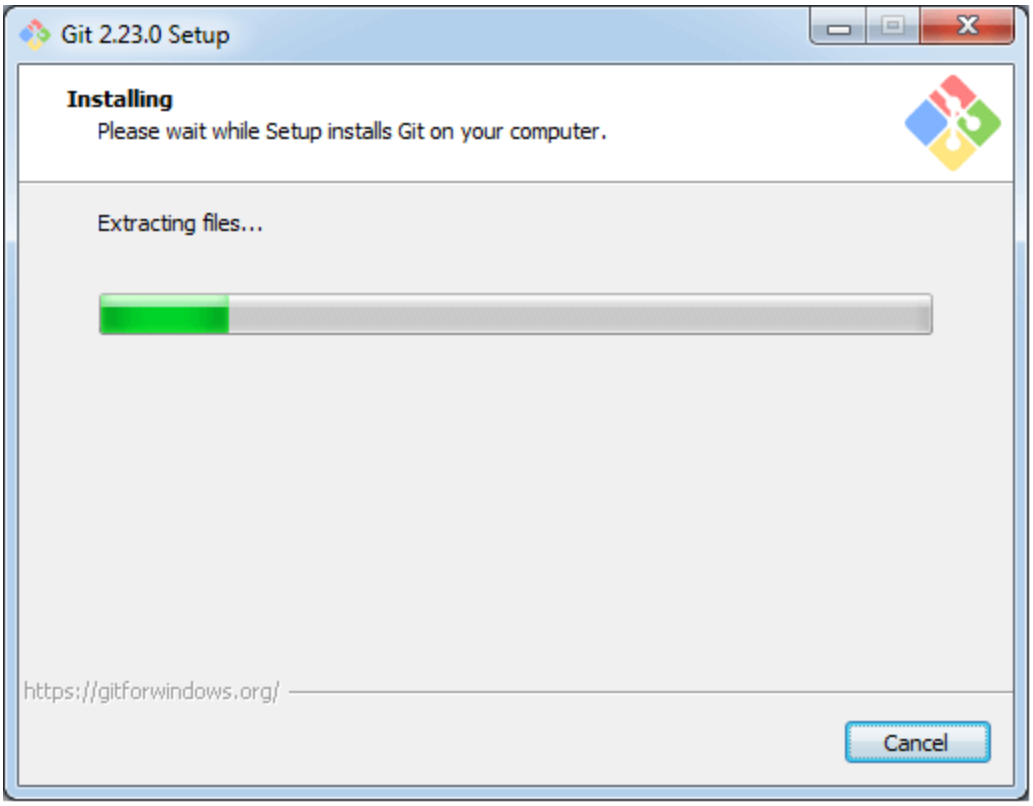
Step8

This is the last step that provides some extra features like system caching, credential management and symbolic link. Select the required features and click on the **next** option.



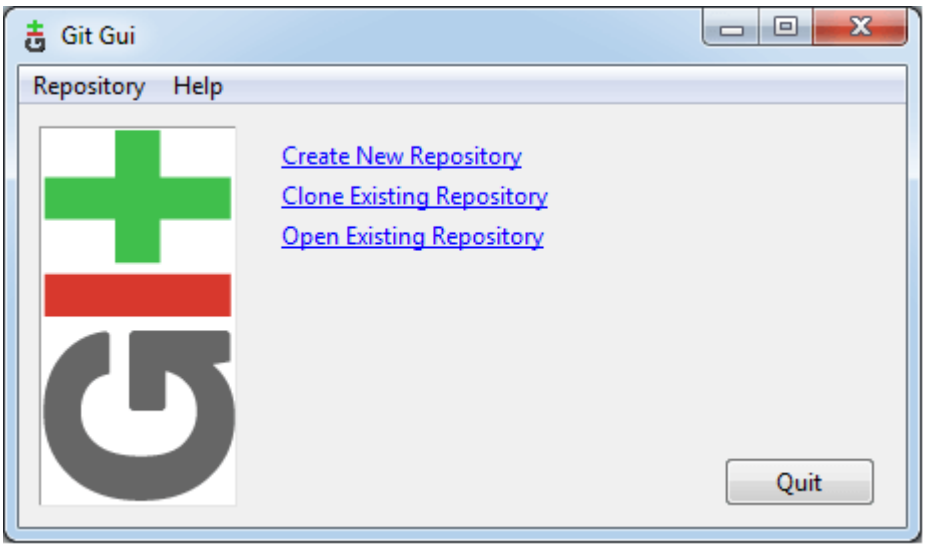
Step9

The files are being extracted in this step.



Therefore, The Git installation is completed. Now you can access the **Git Gui** and **Git Bash**.

The **Git Gui** looks like as

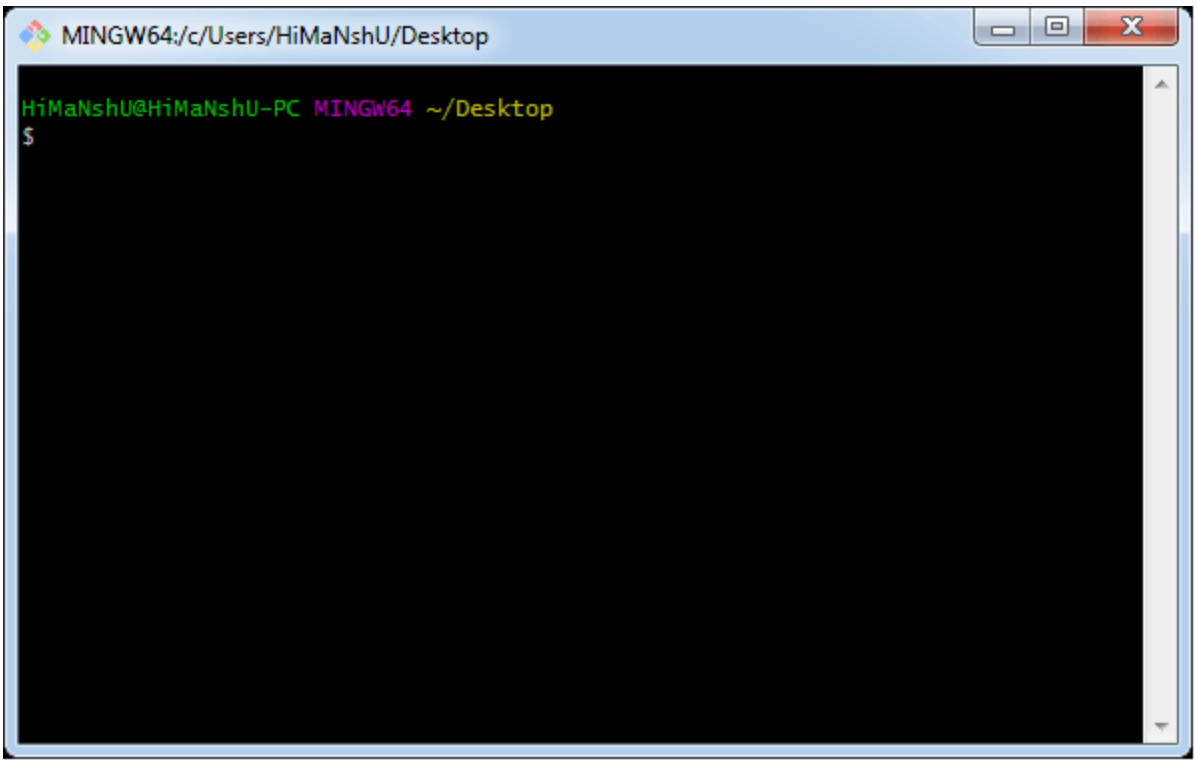


It facilitates with three features.

- Create New Repository

- Clone Existing Repository
- Open Existing Repository

The **Git Bash** looks like as



Install Git on Ubuntu

Git is an open-source distributed version control system that is available for everyone at zero cost. It is designed to handle minor to major projects with speed and efficiency. It is developed to co-ordinate the work among programmers. The version control allows you to track and work together with your team members at the same workspace.

Git is the most common source code management (SCM) and covers more users than earlier VCS systems like SVN. Let's understand how to install Git on your Ubuntu server.

Introduction to Git

Git focuses on data integrity, speed, and non-linear, distributed workflow support. Originally, git was started in 2005 by Linus Torvalds for the Linux kernel development, with other developers of the kernel contributing to its starting development. Junio Hamano has been the main maintainer since 2005.

Unlike almost every client-server system, and with almost every distributed version control system, all Git directories on all computers are a completely developed repository with full version-tracking and history abilities, free from a central server or network access. Git is an open-source and free software distributed upon the GPL-2.0-only license.

Brief History of Git

In April 2005, Git development started after several kernel developers gave up using BitKeeper, an SCM (source control management) system they had been utilizing to manage the project.

Linus Torvalds wished for a distributed system that could be used like BitKeeper, but the available open-source systems don't meet his requirements. Torvalds specified an instance of a source-control management system requiring thirty seconds to use a patch and update every related metadata and esteemed that it wouldn't scale to the requirements of the development of the Linux kernel, in which synchronizing with associate maintainers could need 250 such operations at once. He cited that patching shouldn't take 3+ seconds with his design principle and included three more purposes:

- Add robust safeguards opposed to corruption, either malicious or accidental.
- Support a BitKeeper-like, distributed workflow.
- Take CVS (Concurrent Versions System) as an instance of what not to do; make the same opposite decision if not sure.

Design of Git

The design of Git was inspired by Monotone and BitKeeper. Originally Git was developed as a low-level engine for the version control system, where others can specify front ends like StGIT or Cogito.

Characteristics

The design of Git is the synthesis of the experience of Torvalds with Linux in managing a bigger distributed development project with his file-system performance knowledge gained from a similar project and the requirement to generate an active system. These conditions led to the below implementation options:

- **Non-linear development support:** Git supports fast merging and branching and contains special tools for navigating and visualizing a non-

linear development history. A basic thought is that modification will be combined more frequently than it's written in Git because it's passed across several reviewers. In Git, the branches are lightweight, and a branch is just a reference to a single commit. The structure of a full branch can be made using its parental commits.

- **Distributed development:** Like Monotone, Bazaar, Mercurial, BitKeeper, and Darcs, Git provides all developers a copy of the complete development history, and modifications are copied from such repositories to others.
- **Compatibility with older protocols and systems:** Repositories can be released by a Git protocol, FTP, HTTP, or HTTPS on either a Secure Shell or plain socket.
- **Efficient handling of bigger projects:** Torvalds has defined Git as being very scalable and fast, and performance tests implemented by Mozilla represented that it's an order of magnitude rapid differentiating bigger repositories than GNU Bazaar and Mercurial.
- **History cryptographic authentication:** The history of Git is stored in a form that the ID of a specific version relies on the full development history causing that commit.
- **Toolkit-based structure:** Git was developed as a collection of programs specified in C and many shell scripts that offer wrappers across these programs. However, most of these scripts have been since re-specified in C for portability and speed.
- **Pluggable strategies:** Git contains a well-defined structure of a lacking merge, and it contains two or more algorithms to complete it as an element of its toolkit structure.

Data structures

The primitives of Git are not a source-code management system inherently. Git has integrated the full set of aspects expected of a classic SCM, with aspects mostly being made as required, then refined and increased over time from this starting design approach.

Git includes two different data structures. The first data structure is a mutable index (also known as cache or stage) that caches details about the active directory and the upcoming

revision to be devoted. The second data structure is an append-only immutable object database.

The immutable database includes five object types:

- Blob
- Tree
- Commit
- Tag
- Packfile

Git additionally stores labels known as refs (or references) to represent the location of several commits. They are:

- Heads (branches)
- HEAD
- Tags

Git Installation

I have done this installation on Ubuntu 16.04 LTS. But the given commands should also work with the other versions.

Below are the steps to install the Git on Ubuntu server:

Step1: Start the General OS and Package update

First of all, we should start the general OS and package updates. To do so, run the below command:

1. $ apt-get update

Now we have started the general OS and package updates. After this, we will run the general updates on the server so that we can get started with installing Git. To do so, run the following commands:

Step2: Install Git

To install Git, run the below command:

1. $ apt-get install git-core

The above command will install the Git on your system, but it may ask you to confirm the download and installation.

Step3: Confirm Git the installation

To confirm the installation, press '**y**' key on the editor. Now, Git is installed and ready to use.

When the central installation done, first check to ensure the executable file is set up and accessible. The best way to do this is the git version command. It will be run as:

1. $ git --version

Output:

```
git version 2.24.0
```

Step4: Configure the Git for the First use

Now you can start using Git on your system. You can explore many features of the version control system. To go with Git, you have to configure the initial user access process. It can be done with the git config command.

Suppose I want to register a user whose user name is "javaTpoint" and email address is "Javatpoint@xyz", then it will be done as follows:

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

Now, you have successfully registered a user for the version control system.

Install Git on Mac

There are multiple ways to install Git on mac. It comes inbuilt with Xcode or its other command-line tools. To start the Git, open terminal and enter the below command:

1. $ git --version

The above command will display the installed version of Git.

Output:

```
git version 2.24.0 (Apple Git-66)
```

If you do not have installed it already, then it will ask you to install it.

Apple provides support for Git, but it lags by several major versions. We may install a newer version of Git using one of the following methods:

Git Installer for Mac

This process is the simplest way to download the latest version of Git. Visit the official page of git downloads. Choose the download option for **Mac OS X**.

The installer file will download to your system. Follow the prompts, choose the required installer option. After the installation process completed, verify the installation was successful by running the below command on the terminal:

1. $ git --version

The above command will display the installed version of Git. Consider the below output.

Output:

```
git version 2.24.0 (Apple Git-66)
```

Now, we have successfully installed the latest version on our mac OS. It's time to configure the version control system for the first use.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

To go in-depth with the git config command, visit Here.

Installation via MacPorts

Sometimes MacPorts also referred to DarwinPorts. It makes the straightforward installation of software on the Mac OS and Darwin operating systems. If we have installed MacPorts for managing packages on OS X, follow the below steps to install Git.

Step1: Update MacPorts

To update MacPorts, run the below command:

1. $ sudo port selfupdate

Step2: Search for the latest Ports

To search for the most recent available Git ports and variants, run the below command:

1. $ port search git

2. $ port variants git

The above command will search for the latest available port and options and will install it.

Step3: Install Git

To install Git, run the below command:

1. $ sudo port install git

We can also install some extra tools with Git. These tools may assist Git in different manners. To Install Git with bash-completion, svn, and the docs, run the below command:

1. $ sudo port install git +svn +doc +bash_completion +gitweb

Now, we have successfully installed Git with the help of MacPorts on our system.

Step4: Configure Git

The next step for the first use is git configuration.

We will configure the Git username and email address as same as given above.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

Install Git via Homebrew

Homebrew is used to make the software installation straight forward. If we have installed Homebrew for managing packages on OS X, follow the below steps to go with Git:

Step1: install Git

Open the terminal and run the below command to install Git using Homebrew:

1. $ brew install git

The above command will install the Git on our machine. The next step is to verify the installation.

Step2: Verify the installation

It is essential to ensure that whether the installation process has been succeeded or not.

To verify whether the installation has been successful or not, run the below command:

1. $ git --version

The above command will display the version that has been installed on your system. Consider the below output:

```
git version 2.24.0
```

Step3: Configure Git

We will configure the Git username and email address same as given above.

To register a username, run the below command:

1. $ git config --global user.name "javaTpoint"

To register an email address for the given author, run the below command:

1. $ git config --global user.email "javatpoint@xyz"

Git Environment Setup

The environment of any tool consists of elements that support execution with software, hardware, and network configured. It includes operating system settings, hardware configuration, software configuration, test terminals, and other support to perform the operations. It is an essential aspect of any software.

It will help you to understand how to set up Git for first use on various platforms so you can read and write code in no time.

The Git config command

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.

Setting username

The username is used by the Git for each commit.

1. $ git config --global user.name "Himanshu Dubey"

Setting email id

The Git uses this email id for each commit.

1. $ git config --global user.email "himanshudubey481@gmail.com"

There are many other configuration options that the user can set.

Setting editor

You can set the default text editor when Git needs you to type in a message. If you have not selected any of the editors, Git will use your default system's editor.

To select a different text editor, such as Vim,

1. $ git config --global core.editor Vim

Checking Your Settings

You can check your configuration settings; you can use the **git config --list** command to list all the settings that Git can find at that point.

1. $ git config -list

This command will list all your settings. See the below command line output.

Output

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop
$ git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
rebase.autosquash=true
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
http.sslbackend=openssl
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge --skip -- %f
filter.lfs.process=git-lfs filter-process --skip
filter.lfs.required=true
credential.helper=manager
gui.recentrepo=C:/Git
user.email=dav.himanshudubey481@gmail.com
user.name=Himanshu Dubey
```

Colored output

You can customize your Git output to view a personalized color theme. The **git config** can be used to set these color themes.

Color.ui

1. $ Git config -global color.ui true

The default value of color.ui is set as auto, which will apply colors to the immediate terminal output stream. You can set the color value as true, false, auto, and always.

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

--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.

--system

The system-level configuration is applied across an entire system. The entire system means all users on an operating system and all repositories. The system-level configuration file stores in a **gitconfig** file off the system directory. **$(prefix)/etc/gitconfig** on UNIX systems and **C:\ProgramData\Git\config** on Windows.

The order of priority of the Git config is local, global, and system, respectively. It means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

Git Tools

To explore the robust functionality of Git, we need some tools. Git comes with some of its tools like Git Bash, Git GUI to provide the interface between machine and user. It supports inbuilt as well as third-party tools.

Git comes with built-in GUI tools like **git bash**, **git-gui**, and **gitk** for committing and browsing. It also supports several third-party tools for users looking for platform-specific experience.

Git Package Tools

Git provides powerful functionality to explore it. We need many tools such as commands, command line, Git GUI. Let's understand some essential package tools.

GitBash

Git Bash is an application for the Windows environment. It is used as Git command line for windows. Git Bash provides an emulation layer for a Git command-line experience. Bash is an abbreviation of **Bourne Again Shell**. Git package installer contains Bash, bash utilities, and Git on a Windows operating system.

Bash is a standard default shell on Linux and macOS. A shell is a terminal application which is used to create an interface with an operating system through commands.

By default, Git Windows package contains the Git Bash tool. We can access it by right-click on a folder in Windows Explorer.

Git Bash Commands

Git Bash comes with some additional commands that are stored in the **/usr/bin** directory of the Git Bash emulation. Git Bash can provide a robust shell experience on Windows. Git Bash comes with some essential shell commands like **Ssh**, **scp**, **cat**, **find**.

Git Bash also includes the full set of Git core commands like **git clone, git commit, git checkout, git push**, and more.

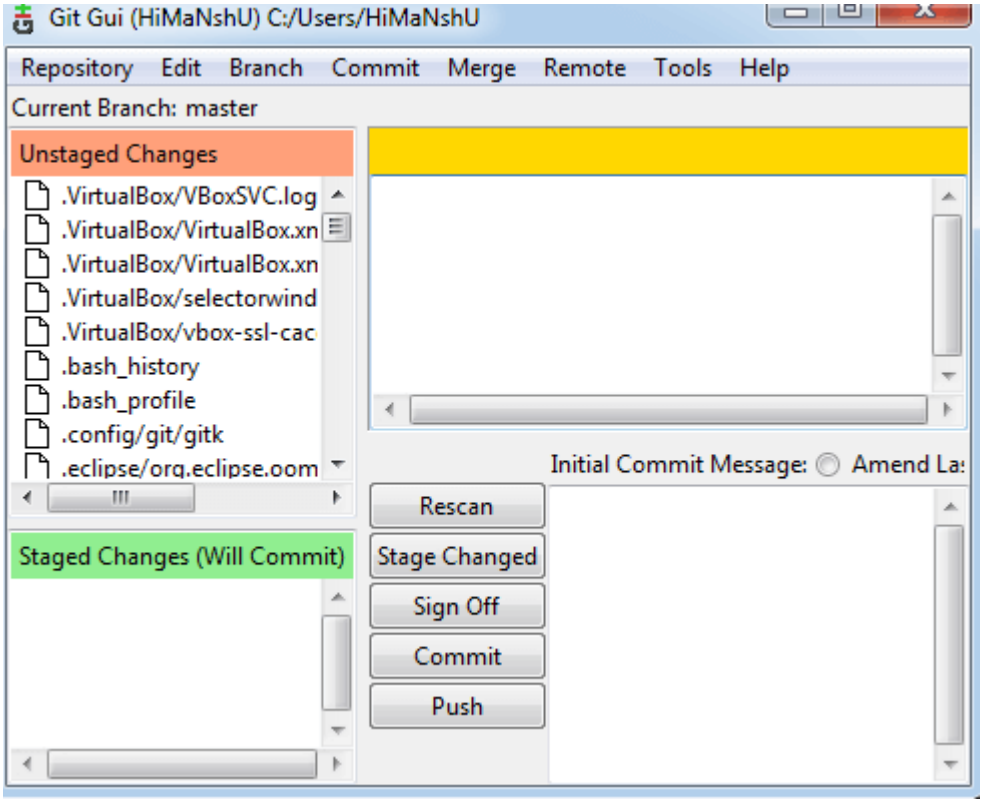
Git GUI

Git GUI is a powerful alternative to Git BASH. It offers a graphical version of the Git command line function, as well as comprehensive visual diff tools. We can access it by simply right click on a folder or location in windows explorer. Also, we can access it through the command line by typing below command.

1. $ git gui

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git gui
```

A pop-up window will open as Git gui tool. The Git GUI's interface looks like as:



Git facilitates with some built-in GUI tools for committing (git-gui) and browsing (gitk), but there are many third-party tools for users looking for platform-specific experience.

Gitk

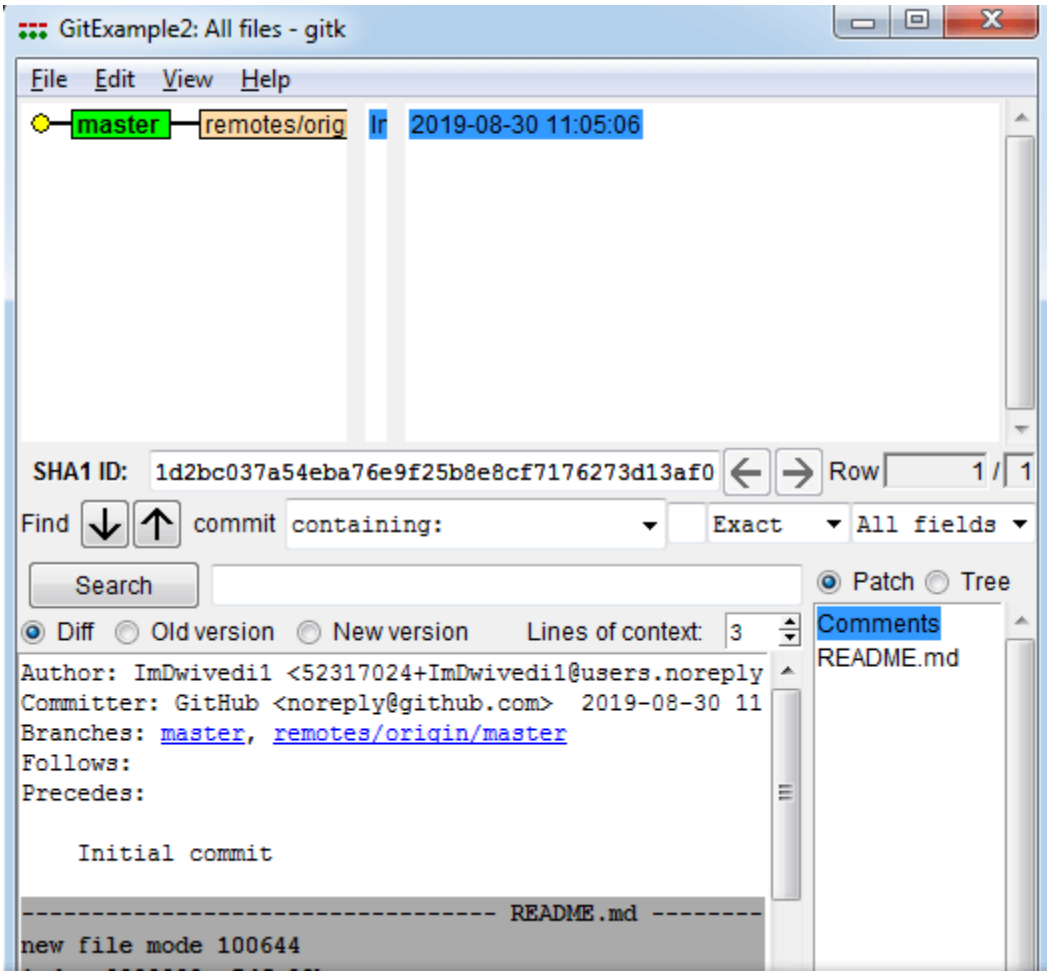
gitk is a graphical history viewer tool. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.

Gitk can invoke from the command-line. Just change directory into a Git repository, and type:

1. $ gitk [git log options]

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ cd GitExample2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ gitk
```

This command invokes the gitk graphical interface and displays the project history. The Gitk interface looks like this:



Gitk supports several command-line options, most of which are passed through to the underlying git log action.

Git Third-Party Tools

Many third-party tools are available in the market to enhance the functionality of Git and provide an improved user interface. These tools are available for distinct platforms like Windows, Mac, Linux, Android, iOS.

A list of popular third party Git tools are as follows:

Tools	Platforms					Price	License Type
	Windows	Mac	Linux	Android	iOS		
SourceTree	Yes	Yes	No	No	No	Free	Proprietary
GitHub Desktop	Yes	Yes	No	No	No	Free	MIT
TortoiseGit	Yes	No	No	No	No	Free	GNU GPL
Git Extensions	Yes	Yes	Yes	No	No	Free	GNU GPL
GitKraken	Yes	Yes	Yes	No	No	Free/$29/$49	Proprietary
SmartGit	Yes	Yes	Yes	No	No	$79/user/free for non-commercial use	Proprietary
Tower	Yes	Yes	No	No	No	$79/user (30 days free trial)	Proprietary
Git Up	No	Yes	No	No	No	Free	GNU GPL
GitEye	Yes	Yes	Yes	No	No	Free	Proprietary
gitg	Yes	No	Yes	No	No	Free	GNUGPL
Git2Go	No	No	No	No	Yes	Free with in-app purchases	Proprietary
GitDrive	No	No	No	No	Yes	Free with in-app purchases	Proprietary
GitFinder	No	Yes	No	No	No	$24.95	Proprietary
SnailGit	No	Yes	No	No	No	&9.99/Lite version	Proprietary
Pocket Git	No	No	No	Yes	No	1.99€	Proprietary
Sublime Merge	Yes	Yes	Yes	No	No	$99/user, $75 annual business sub, free eval	Proprietary

Git Terminology

Git is a tool that covered vast terminology and jargon, which can often be difficult for new users, or those who know Git basics but want to become Git masters. So, we need a little explanation of the terminology behind the tools. Let's have a look at the commonly used terms.

Some commonly used terms are:

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.

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.

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.

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.

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.

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.

1. Light-weighted tag
2. Annotated tag

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 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 Flow

GitFlow is a **branching model** for Git, developed by **Vincent Driessen**. It is very well organized to collaborate and scale the development team. Git flow is a collection of Git commands. It accomplishes many repository operations with just single commands.

Git Squash

In Git, the term squash is used to squash previous commits into one. Git squash is an excellent technique to group-specific changes before forwarding them to others. You can merge several commits into a single commit with the powerful interactive rebase command.

Git Rm

In Git, the term rm stands for **remove**. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.

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.

12 Git Commands

There are many different ways to use Git. Git supports many command-line tools and graphical user interfaces. The Git command line is the only place where you can run all the Git commands.

The following set of commands will help you understand how to use Git via the command line.

Basic Git Commands

Here is a list of most essential Git commands that are used daily.

1. Git Config command
2. Git init command
3. Git clone command
4. Git add command
5. Git commit command
6. Git status command
7. Git push Command
8. Git pull command
9. Git Branch Command
10. Git Merge Command
11. Git log command
12. Git remote command

Let's understand each command in detail.

1) 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

1. $ git config --global user.name "ImDwivedi1"
2. $ git config --global user.email "Himanshudubey481@gmail.com"

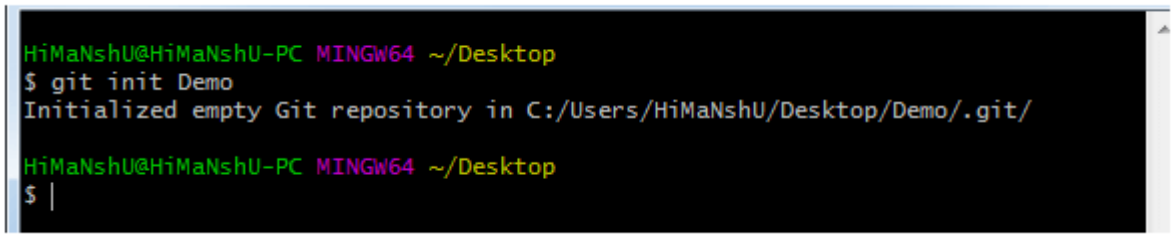
2) Git Init command

This command is used to create a local repository.

Syntax

1. $ git init Demo

The init command will initialize an empty repository. See the below screenshot.

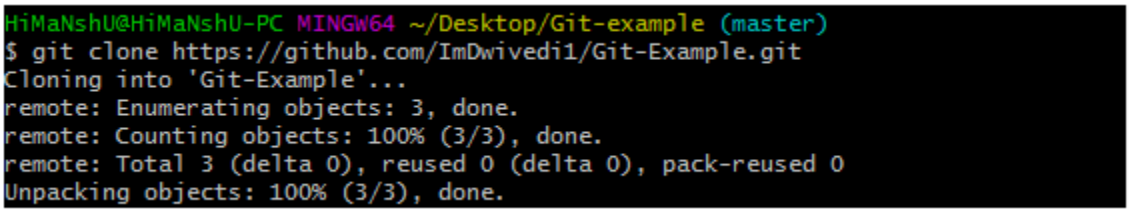


3) Git clone command

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

1. $ git clone URL



4) Git add command

This command is used to add one or more files to staging (Index) area.

Syntax

To add one file

1. $ git add Filename

To add more than one file

1. $ git add*

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git add README.md
```

5) 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

1. $ 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

1. $ git commit -a

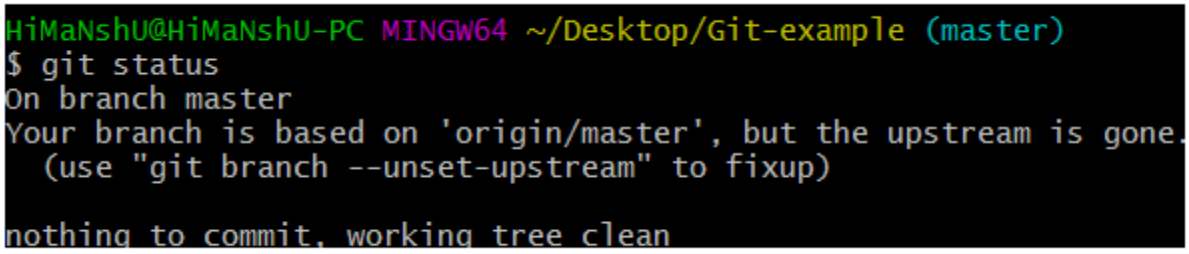
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git commit -a -m "Adding the key of c"
[master (root-commit) 758797a] Adding the key of c
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

6) 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

1. $ git status



7) 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.

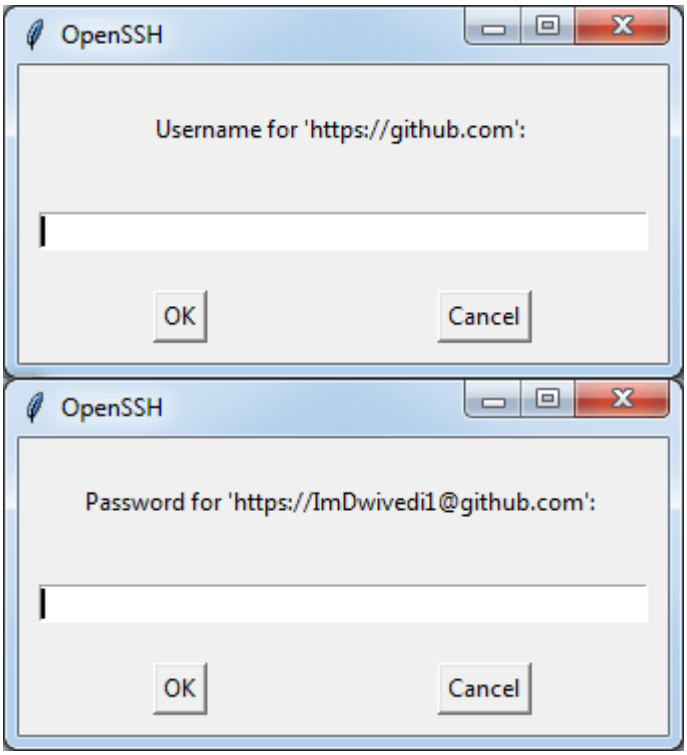
Git push origin master

This command sends the changes made on the master branch, to your remote repository.

Syntax

1. $ git push [variable name] master

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (mast
er)
$ git push origin master
```




```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git push origin master
Everything up-to-date

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$
```

Git push -all

This command pushes all the branches to the server repository.

Syntax

1. $ git push --all

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git push --all
Everything up-to-date

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$
```

8) 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

1. $ git pull URL

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git pull https://github.com/ImDwivedi1/Git-Example
warning: no common commits
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            HEAD       -> FETCH_HEAD
fatal: refusing to merge unrelated histories

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$
```

9) Git Branch Command

This command lists all the branches available in the repository.

Syntax

1. $ git branch

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git branch
* master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
```

10) Git Merge Command

This command is used to merge the specified branch?s history into the current branch.

Syntax

1. $ git merge BranchName

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git merge master
Already up to date.
```

11) Git log Command

This command is used to check the commit history.

Syntax

1. $ git log

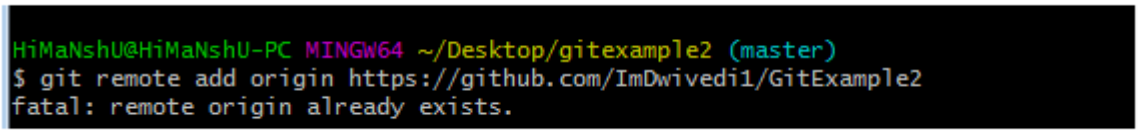
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git log
commit 1d2bc037a54eba76e9f25b8e8cf7176273d13af0 (HEAD -> master, origin/master,
origin/HEAD)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Fri Aug 30 11:05:06 2019 +0530

    Initial commit
```

By default, if no argument passed, Git log shows the most recent commits first. We can limit the number of log entries displayed by passing a number as an option, such as -3 to show only the last three entries.

1. $ git log -3

12) Git remote Command

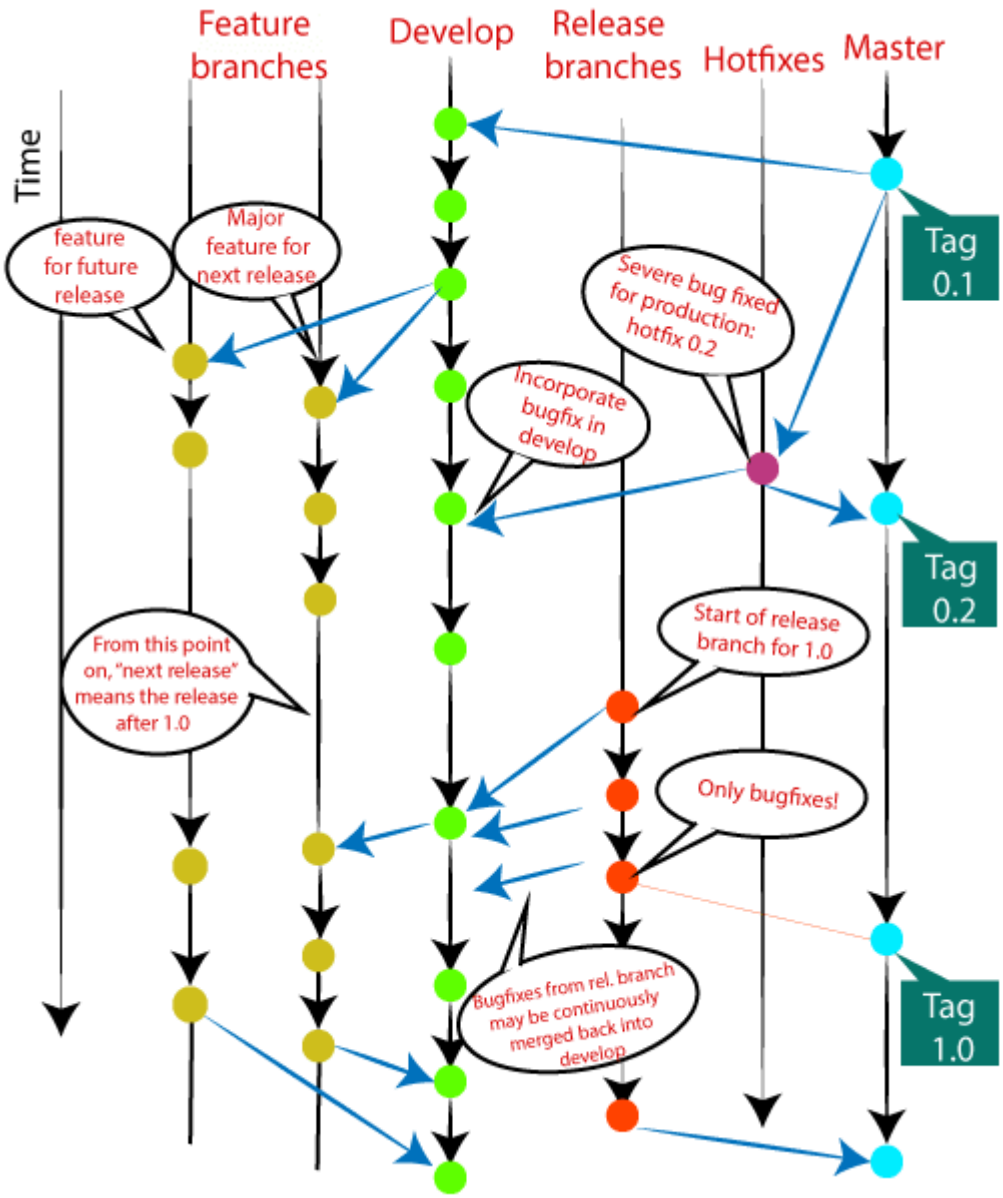
Git Remote command is used to connect your local repository to the remote server. This command allows you to create, view, and delete connections to other repositories. These connections are more like bookmarks rather than direct links into other repositories. This command doesn't provide real-time access to repositories.



Git Flow / Git Branching Model

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.



There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:

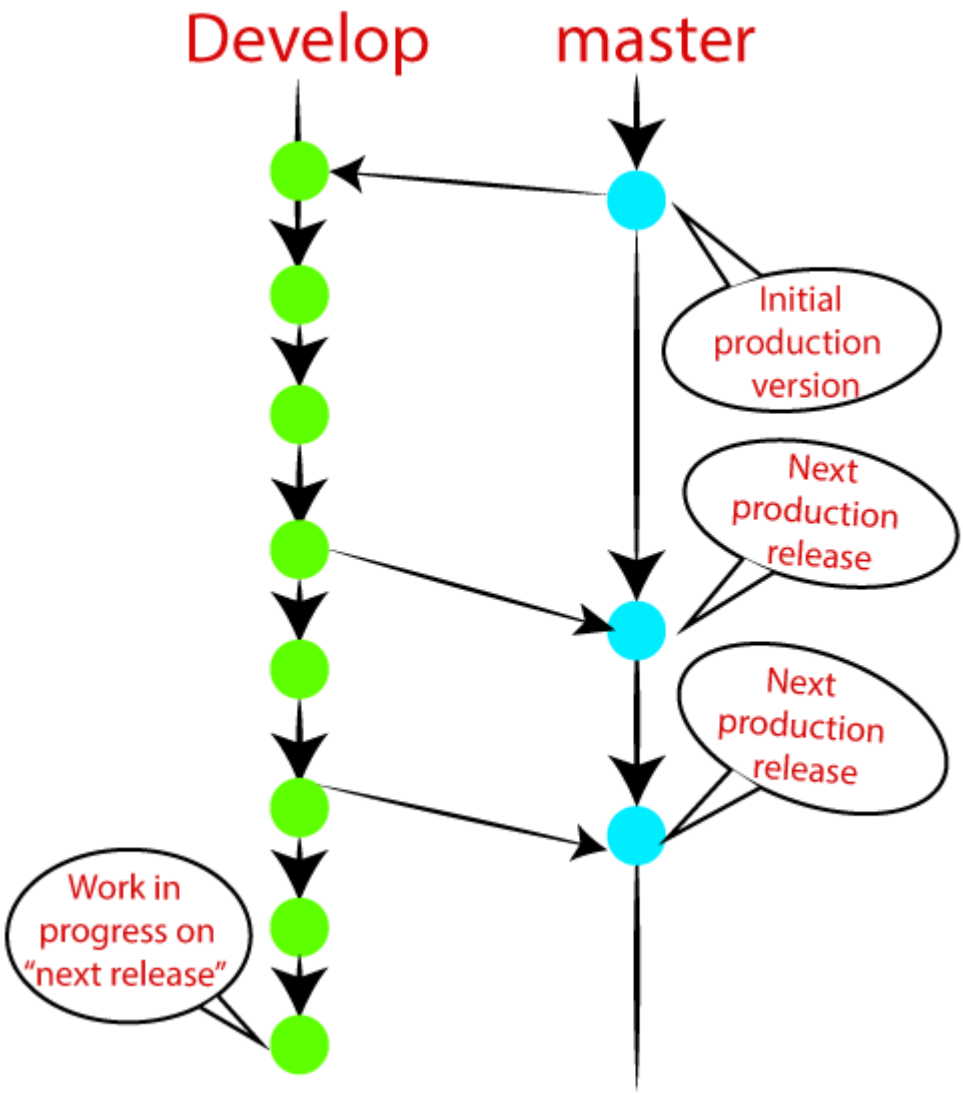
- **Master**
- **Develop**
- **Hotfixes**
- **Release branches**
- **Feature branches**

Every branch has its meaning and standard. Let's understand each branch and its usage.

The Main Branches

Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

- **master**
- **develop**



Master Branch

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

Develop Branch

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "**integration branch**."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

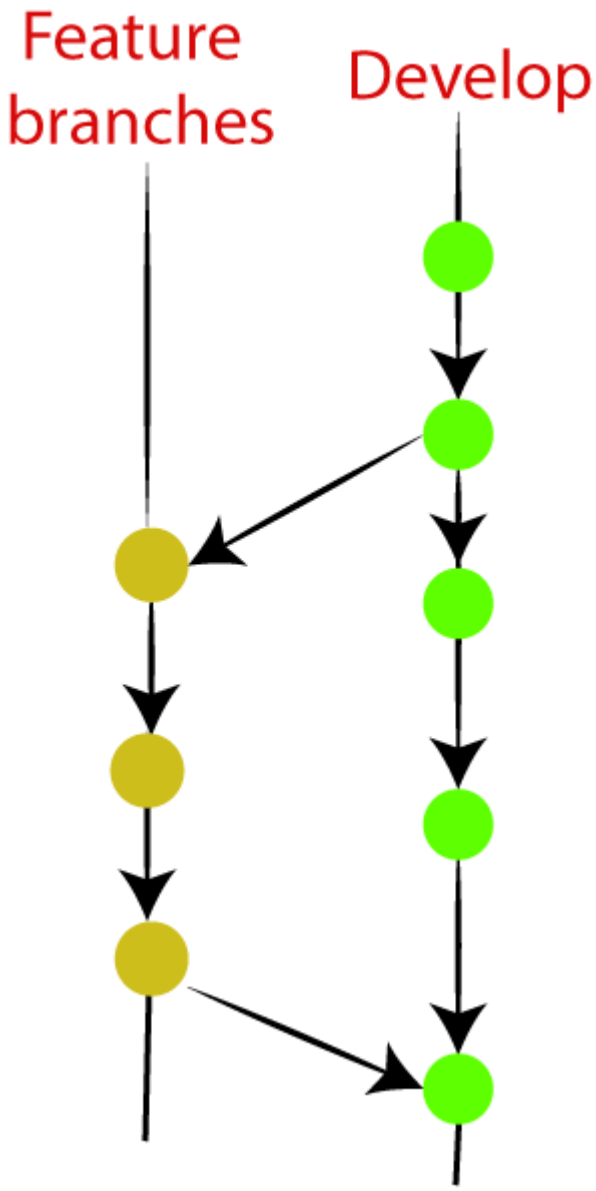
The different types of supportive branches, we may use are as follows:

- **Feature branches**
- **Release branches**
- **Hotfix branches**

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

Feature Branches

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.



To learn how to create a Feature Branch **Visit Here**.

Release Branches

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

When all the targeted features are created, then it can be merged with the develop branch. Some usual standard of the release branch are as follows:

- Generally, senior developers will create a release branch.
- The release branch will contain the predetermined amount of the feature branch.
- The release branch should be deployed to a staging server for testing.
- Any bugs that need to be improved must be addressed at the release branch.
- The release branch must have to be merged back into developing as well as the master branch.
- After merging, the release branch with the develop branch must be tagged with a version number.

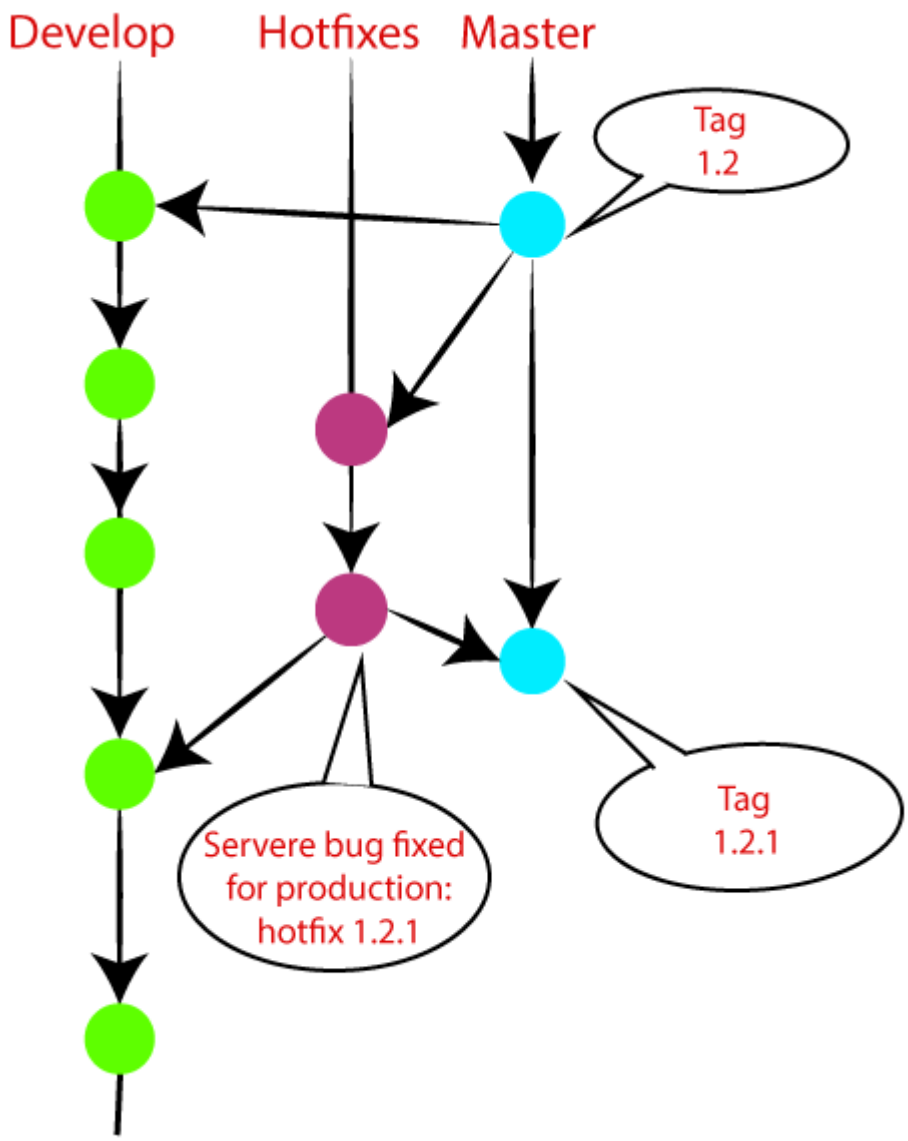
To create a release branch, visit **Git Branching**.

To tag branch after merging the release branch, Visit **Git tag**.

Hotfix Branches

Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.



Git Cheat Sheet

1. Git configuration

- **Git config**
  Get and set configuration variables that control all facets of how Git looks and operates.
  Set the name:
  $ git config --global user.name "User name"
  Set the email:

$ git config --global user.email "himanshudubey481@gmail.com"

Set the default editor:

$ git config --global core.editor Vim

Check the setting:

$ git config -list

- **Git alias**

  Set up an alias for each command:

  $ git config --global alias.co checkout

  $ git config --global alias.br branch

  $ git config --global alias.ci commit

  $ git config --global alias.st status

2. Starting a project

- **Git init**

  Create a local repository:

  $ git init

- **Git clone**

  Make a local copy of the server repository.

  $ git clone

3. Local changes

- **Git add**

  Add a file to staging (Index) area:

  $ git add Filename

  Add all files of a repo to staging (Index) area:

  $ git add*

- **Git commit**

  Record or snapshots the file permanently in the version history **with a message**.

  $ git commit -m " Commit Message"

4. Track changes

- **Git diff**
  Track the changes that have not been staged: $ git diff
  Track the changes that have staged but not committed: $ git diff --staged
  Track the changes after committing a file: $ git diff HEAD
  Track the changes between two commits: $ git diff Git Diff Branches: $ git diff < branch 2>
- **Git status**
  Display the state of the working directory and the staging area. $ git status
- **Git show Shows objects:**
  $ git show

5. Commit History

- **Git log**
  Display the most recent commits and the status of the head: $ git log
  Display the output as one commit per line: $ git log -oneline
  Displays the files that have been modified: $ git log -stat
  Display the modified files with location: $ git log -p
- **Git blame**
  Display the modification on each line of a file: $ git blame <file name>

6. Ignoring files

- **.gitignore**
  Specify intentionally untracked files that Git should ignore. Create .gitignore:

$ touch .gitignore List the ignored files:
$ git ls-files -i --exclude-standard

7. Branching

- **Git branch Create branch:**
  $ git branch List Branch:
  $ git branch --list Delete a Branch:
  $ git branch -d Delete a remote Branch:
  $ git push origin -delete Rename Branch:
  $ git branch -m
- **Git checkout**
  Switch between branches in a repository.
  Switch to a particular branch:
  $ git checkout
  Create a new branch and switch to it:
  $ git checkout -b Checkout a Remote branch:
  $ git checkout
- **Git stash**
  Switch branches without committing the current branch. Stash current work:
  $ git stash
  Saving stashes with a message:
  $ git stash save ""
  Check the stored stashes:
  $ git stash list
  Re-apply the changes that you just stashed:
  $ git stash apply
  Track the stashes and their changes:
  $ git stash show
  Re-apply the previous commits:
  $ git stash pop
  Delete a most recent stash from the queue:
  $ git stash drop
  Delete all the available stashes at once:

$ git stash clear
Stash work on a separate branch:
$ git stash branch

- **Git cherry pic**
  Apply the changes introduced by some existing commit:
  $ git cherry-pick

8. Merging

- **Git merge**
  Merge the branches:
  $ git merge
  Merge the specified commit to currently active branch:
  $ git merge
- **Git rebase**
  Apply a sequence of commits from distinct branches into a final commit.
  $ git rebase
  Continue the rebasing process:
  $ git rebase -continue Abort the rebasing process:
  $ git rebase --skip
- **Git interactive rebase**
  Allow various operations like edit, rewrite, reorder, and more on existing commits.
  $ git rebase -i

9. Remote

- **Git remote**
  Check the configuration of the remote server:
  $ git remote -v
  Add a remote for the repository:
  $ git remote add Fetch the data from the remote server:
  $ git fetch
  Remove a remote connection from the repository:
  $ git remote rm
  Rename remote server:

$ git remote rename
Show additional information about a particular remote:
$ git remote show
Change remote:
$ git remote set-url

- **Git origin master**
  Push data to the remote server:
  $ git push origin master Pull data from remote server:
  $ git pull origin master

10. Pushing Updates

- **Git push**
  Transfer the commits from your local repository to a remote server. Push data to the remote server:
  $ git push origin master Force push data:
  $ git push -f
  Delete a remote branch by push command:
  $ git push origin -delete edited

11. Pulling updates

- **Git pull**
  Pull the data from the server:
  $ git pull origin master
  Pull a remote branch:
  $ git pull
- **Git fetch**
  Download branches and tags from one or more repositories. Fetch the remote repository:
  $ git fetch< repository Url> Fetch a specific branch:
  $ git fetch
  Fetch all the branches simultaneously:
  $ git fetch -all

Synchronize the local repository:
$ git fetch origin

12. Undo changes

- **Git revert**
  Undo the changes:
  $ git revert
  Revert a particular commit:
  $ git revert
- **Git reset**
  Reset the changes:
  $ git reset -hard
  $ git reset -soft:
  $ git reset --mixed

13. Removing files

- **Git rm**
  Remove the files from the working tree and from the index:
  $ git rm <file Name>
  Remove files from the Git But keep the files in your local repository:
  $ git rm --cached

Git Init

The git init command is the first command that you will run on Git. The git init command is used to create a new blank repository. It is used to make an existing project as a Git project. Several Git commands run inside the repository, but init command can be run outside of the repository.

The git init command creates a .git subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata. These metadata can be categorized into objects, refs, and temp files. It also initializes a HEAD pointer for the master branch of the repository.

Creating the first repository

Git version control system allows you to share projects among developers. For learning Git, it is essential to understand that how can we create a project on Git. A repository is a directory that contains all the project-related data. There can also be more than one project on a single repository.

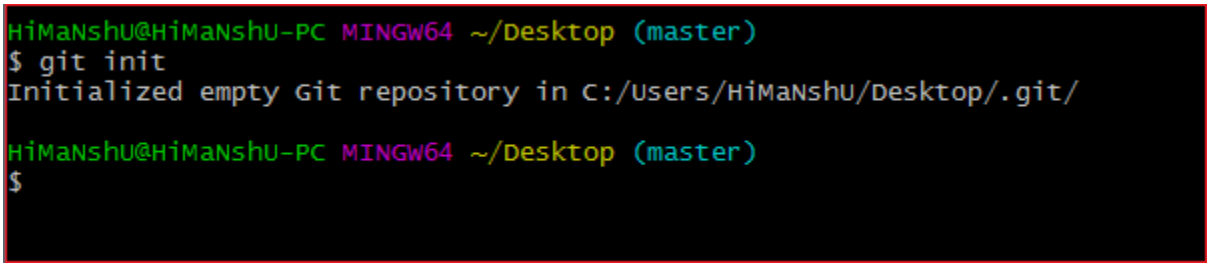
We can create a repository for blank and existing projects. Let's understand how to create a repository.

Create a Repository for a Blank (New) Project:

To create a blank repository, open command line on your desired directory and run the init command as follows:

1. $ git init

The above command will create an empty .git repository. Suppose we want to make a git repository on our desktop. To do so, open Git Bash on the desktop and run the above command. Consider the below output:



The above command will initialize a **.git** repository on the desktop. Now we can create and add files on this repository for version control.

To create a file, run the cat or touch command as follows:

1. $ touch <file Name>

To add files to the repository, run the git add command as follows:

1. $ git add <file name>

Learn more about git add command visit Git Add.

Create a Repository for an existing project

If you want to share your project on a version control system and control it with Git, then, browse your project's directory and start the git command line (Git Bash for Windows) here. To initialize a new repository, run the below command:

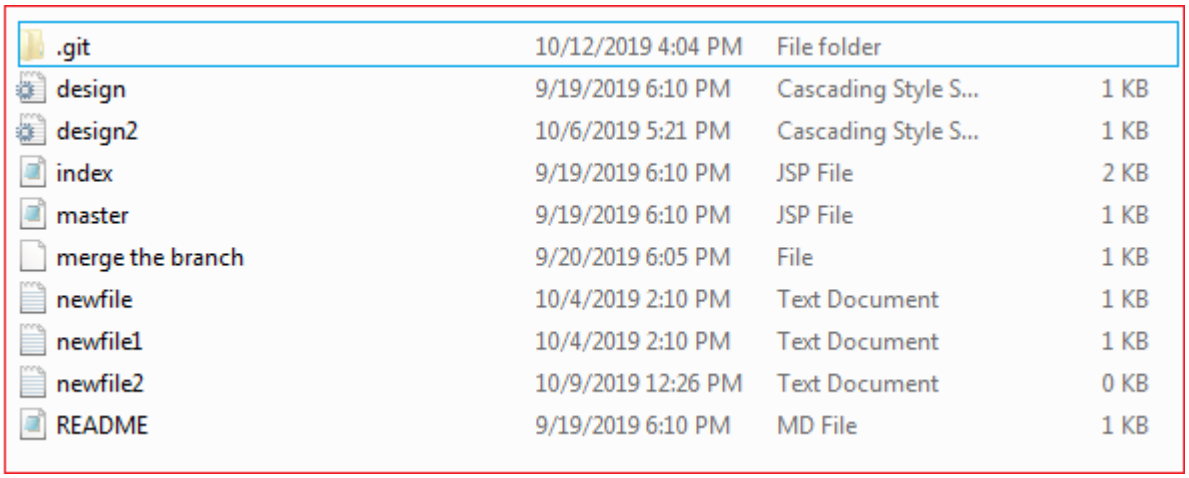
Syntax:

1. $ git init

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project
$ git init
Initialized empty Git repository in C:/My Project/.git/

HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project (master)
$
```

The above command will create a new subdirectory named **.git** that holds all necessary repository files. The .git subdirectory can be understood as a Git repository skeleton. Consider the below image:

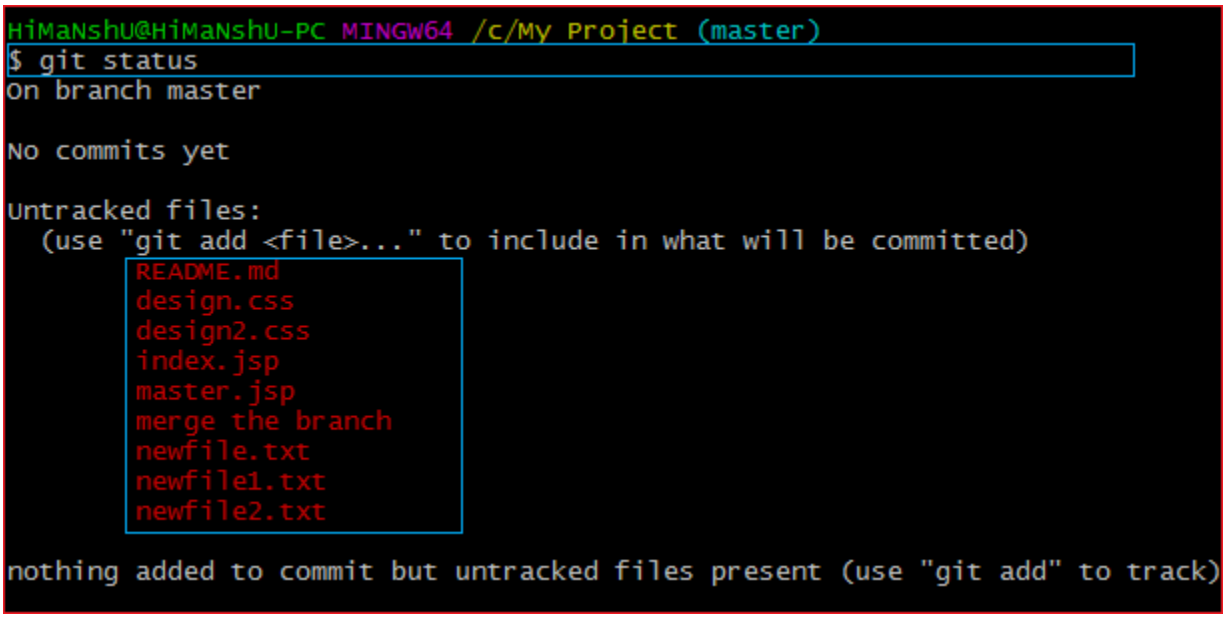


An empty repository .git is added to my existing project. If we want to start version-controlling for existing files, we have to track these files with git add command, followed by a commit.

We can list all the untracked files by git status command.

1. $ git status

Consider the below output:



In the above output, the list of all untracked files is displayed by the git status command. To learn more about status command, visit **Git Status**.

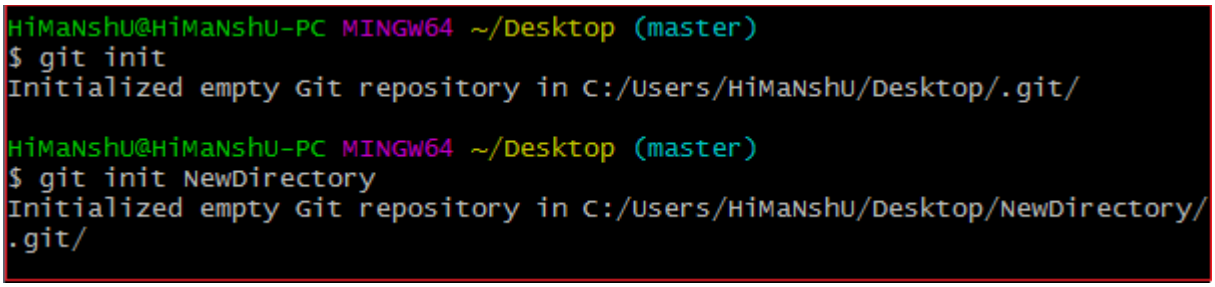
We can track all the untracked files by Git Add command.

Create a Repository and Directory Together

The git init command allows us to create a new blank repository and a directory together. The empty repository .git is created under the directory. Suppose I want to create a blank repository with a project name, then we can do so by the git init command. Consider the below command:

1. $ git init NewDirectory

The above command will create an empty .git repository under a directory named **NewDirectory**. Consider the below output:



In the above output, the directory and the repository both are created.

Hence we can create a repository using git init command. Two other commands are handy to start with git. They are **Git Add**, and **Git commit**.

Also, see various operations on the repository, see **Git Repository**.

Git Add

The git add command is used to add file contents to the Index (Staging Area).This command updates the current content of the working tree to the staging area. It also prepares the staged content for the next commit. Every time we add or update any file in our project, it is required to forward updates to the staging area.

The git add command is a core part of Git technology. It typically adds one file at a time, but there some options are available that can add more than one file at once.

The "index" contains a snapshot of the working tree data. This snapshot will be forwarded for the next commit.

The git add command can be run many times before making a commit. These all add operations can be put under one commit. The add command adds the files that are specified on command line.

The git add command does not add the .gitignore file by default. In fact, we can ignore the files by this command.

Let's understand how to add files on Git?

Git add files

Git add command is a straight forward command. It adds files to the staging area. We can add single or multiple files at once in the staging area. It will be run as:

1. $ git add <File name>

The above command is added to the git staging area, but yet it cannot be shared on the version control system. A commit operation is needed to share it. Let's understand the below scenario.

We have created a file for our newly created repository in **NewDirectory**. To create a file, use the touch command as follows:

1. $ touch newfile.txt

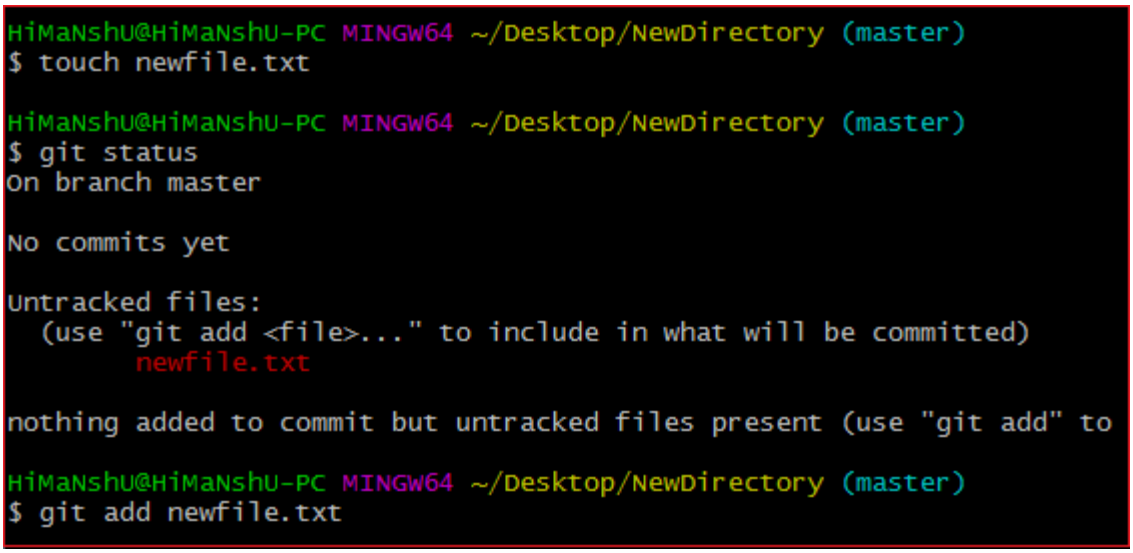
And check the status whether it is untracked or not by git status command as follows:

1. $ git status

The above command will display the untracked files from the repository. These files can be added to our repository. As we know we have created a newfile.txt, so to add this file, run the below command:

1. $ git add newfile.txt

Consider the below output:



From the above output, we can see **newfile.txt** has been added to our repository. Now, we have to commit it to share on Git.

Git Add All

We can add more than one files in Git, but we have to run the add command repeatedly. Git facilitates us with a unique option of the add command by which we can add all the available files at once. To add all the files from the repository, run the add command with **-A** option. We can use '.' Instead of **-A** option. This command will stage all the files at a time. It will run as follows:

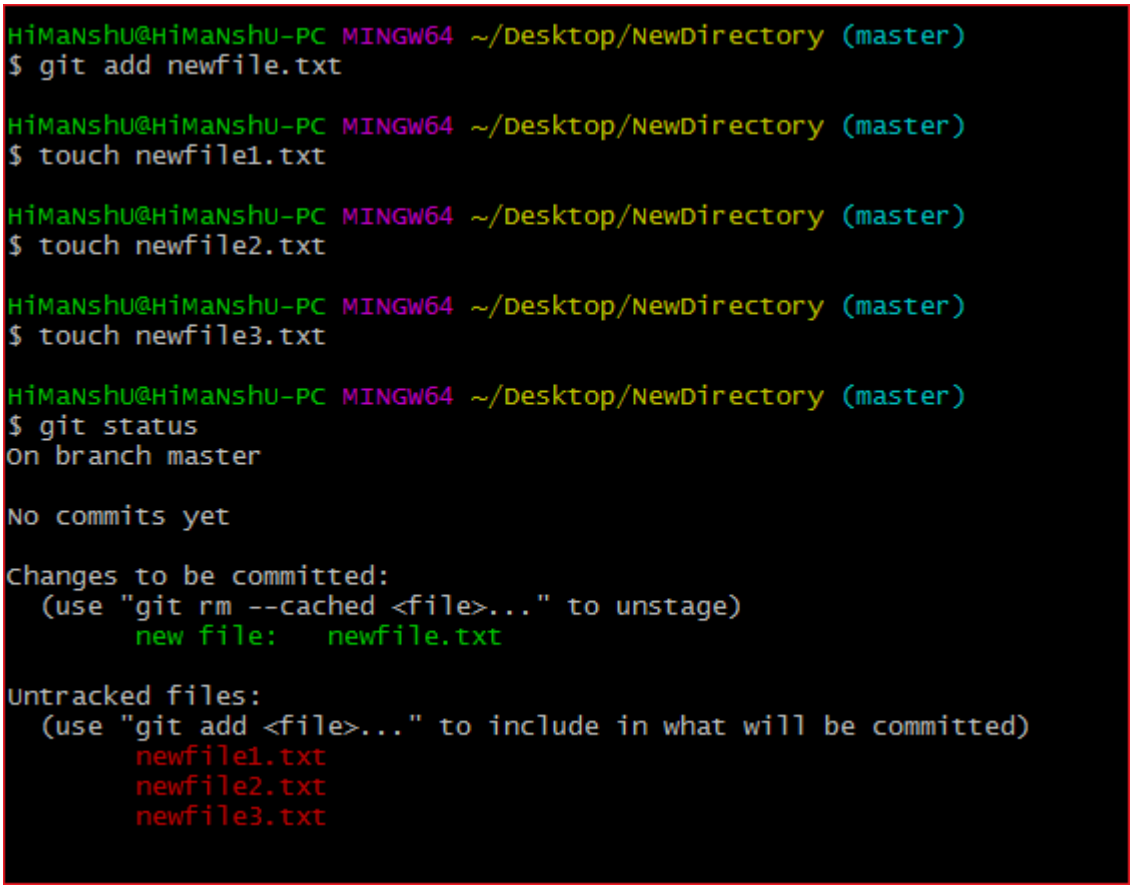
1. $ git add -A

Or

1. $ git add .

The above command will add all the files available in the repository. Consider the below scenario:

We can either create four new files, or we can copy it, and then we add all these files at once. Consider the below output:



In the above output, all the files are displaying as untracked files by Git. To track all of these files at once, run the below command:

1. $ git add -A

The above command will add all the files to the staging area. Remember, the **-A** option is case sensitive. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git add -A

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   newfile.txt
        new file:   newfile1.txt
        new file:   newfile2.txt
        new file:   newfile3.txt
```

In the above output, all the files have been added. The status of all files is displaying as staged.

Removing Files from the Staging Area

The git add command is also used to remove files from the staging area. If we delete a file from the repository, then it is available to our repository as an untracked file. The add command is used to remove it from the staging area. It sounds strange, but Git can do it. Consider the below scenario:

We have deleted the **newfile3.txt** from the repository. The status of the repository after deleting the file is as follows:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   newfile.txt
        new file:   newfile1.txt
        new file:   newfile2.txt
        new file:   newfile3.txt

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
)
        deleted:    newfile3.txt
```

Git Commit

It is used to record the changes in the repository. It is the next command after the git add. Every commit contains the index data and the commit message. Every commit forms a parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.

The staging and committing are co-related to each other. Staging allows us to continue in making changes to the repository, and when we want to share these changes to the version control system, committing allows us to record these changes.

Commits are the snapshots of the project. Every commit is recorded in the master branch of the repository. We can recall the commits or revert it to the older version. Two different commits will never overwrite because each commit has its own commit-id. This commit-id is a cryptographic number created by **SHA (Secure Hash Algorithm)** algorithm.

Let's see the different kinds of commits.

The git commit command

The commit command will commit the changes and generate a commit-id. The commit command without any argument will open the default text editor and ask for the commit message. We can specify our commit message in this text editor. It will run as follows:

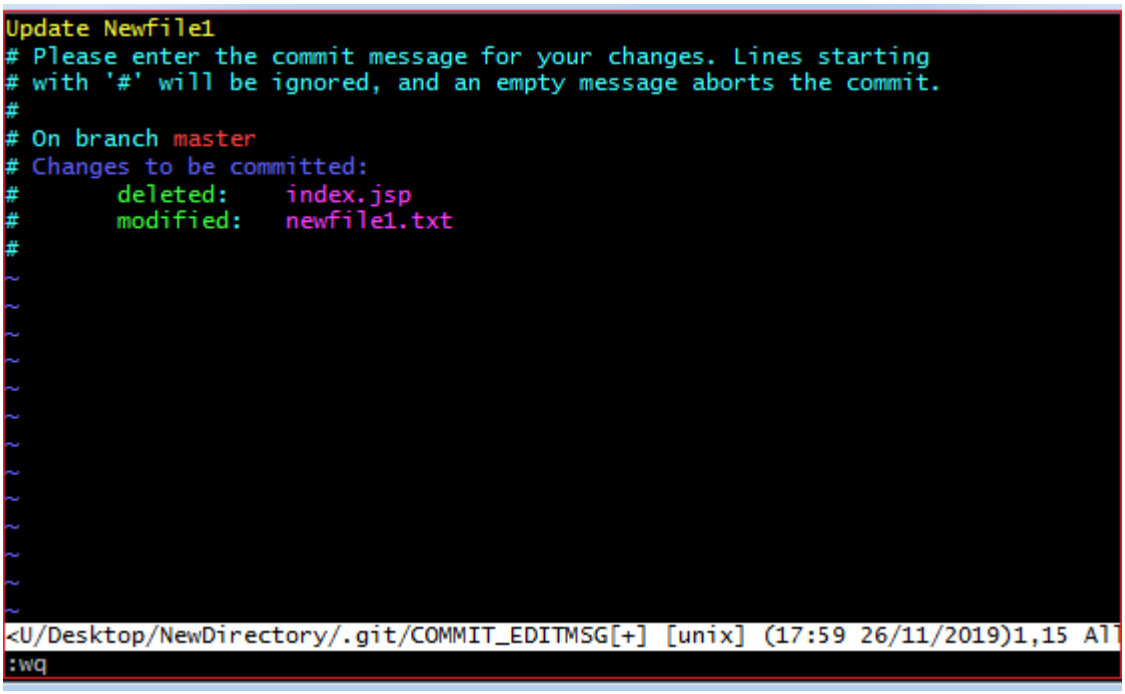
1. $ git commit

The above command will prompt a default editor and ask for a commit message. We have made a change to **newfile1.txt** and want it to commit it. It can be done as follows:

Consider the below output:

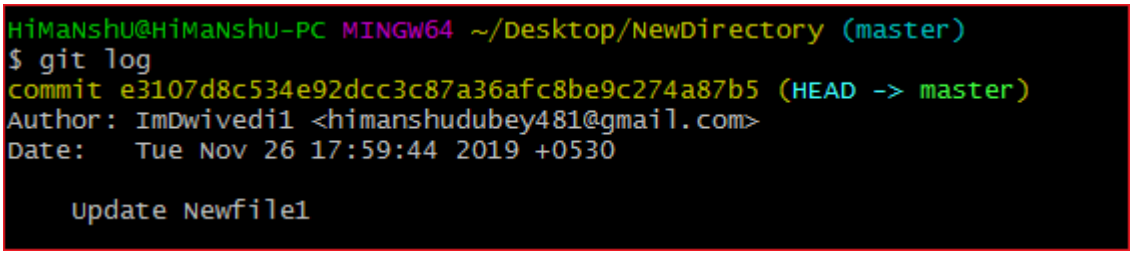
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git commit
[master e3107d8] Update Newfile1
 2 files changed, 1 insertion(+)
 delete mode 100644 index.jsp
```

As we run the command, it will prompt a default text editor and ask for a commit message. The text editor will look like as follows:



Press the **Esc** key and after that '**I**' for insert mode. Type a commit message whatever you want. Press **Esc** after that '**:wq**' to save and exit from the editor. Hence, we have successfully made a commit.

We can check the commit by git log command. Consider the below output:



We can see in the above output that log option is displaying commit-id, author detail, date and time, and the commit message.

To know more about the log option, visit Git Log.

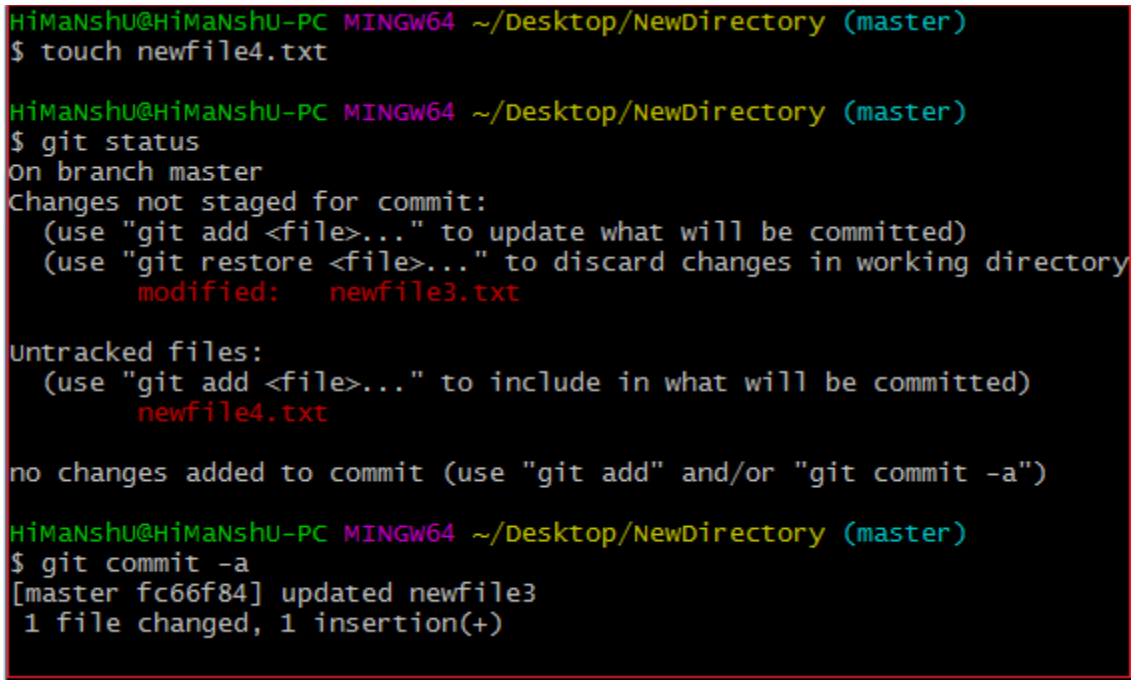
Git commit -a

The commit command also provides **-a** option to specify some commits. It is used to commit the snapshots of all changes. This option only consider already added files in Git. It will not commit the newly created files. Consider below scenario:

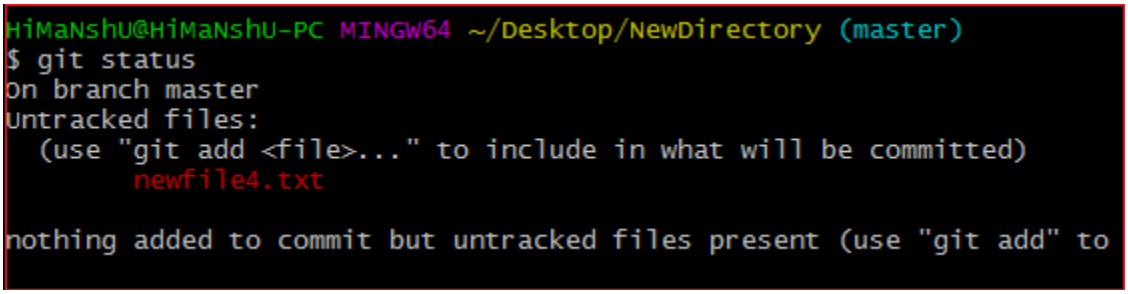
We have made some updates to our already staged file newfile3 and create a file newfile4.txt. Check the status of the repository and run the commit command as follows:

1. $ git commit -a

Consider the output:



The above command will prompt our default text editor and ask for the commit message. Type a commit message, and then save and exit from the editor. This process will only commit the already added files. It will not commit the files that have not been staged. Consider the below output:



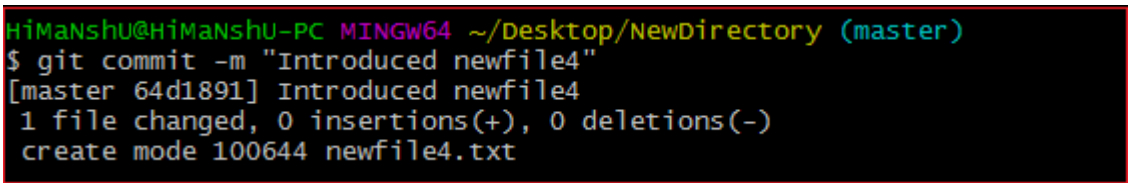
As we can see in the above output, the newfile4.txt has not been committed.

Git commit -m

The -m option of commit command lets you to write the commit message on the command line. This command will not prompt the text editor. It will run as follows:

1. $ git commit -m "Commit message."

The above command will make a commit with the given commit message. Consider the below output:



In the above output, a **newfile4.txt** is committed to our repository with a commit message.

We can also use the **-am** option for already staged files. This command will immediately make a commit for already staged files with a commit message. It will run as follows:

1. $ git commit -am "Commit message."

Git Commit Amend (Change commit message)

The amend option lets us to edit the last commit. If accidentally, we have committed a wrong commit message, then this feature is a savage option for us. It will run as follows:

1. $ git commit -amend

The above command will prompt the default text editor and allow us to edit the commit message.

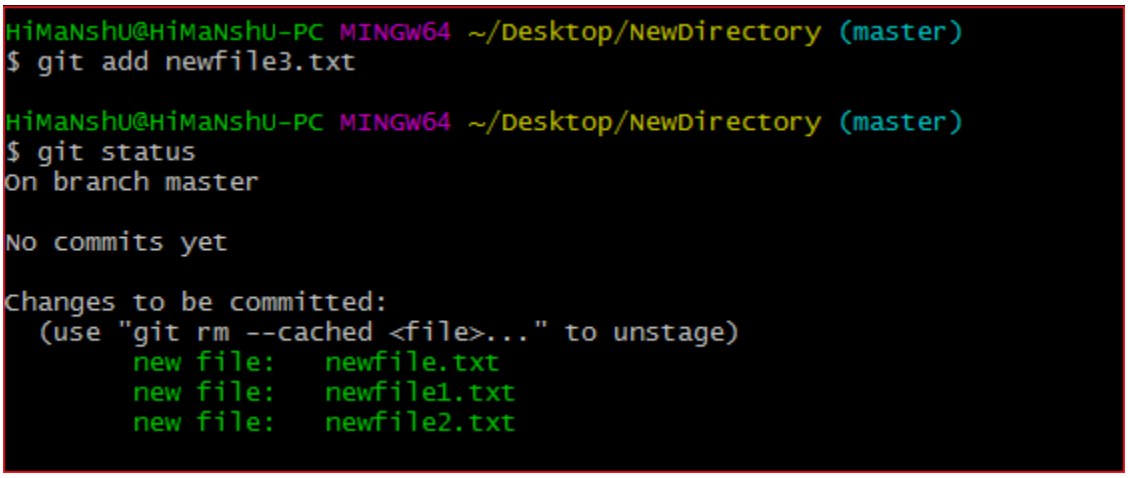
We may need some other essential operations related to commit like revert commit, undo a commit, and more, but these operations are not a part of the commit command. We can do it with other commands. Some essential operations are as follows:

- Git undo commit: Visit **Git Reset**
- Git revert commit: Visit **Git Revert**
- git remove commit: Visit **Git Rm**

As we can see from the above output, the deleted file is still available in the staging area. To remove it from the index, run the below command as follows:

1. $ git add newfile3.txt

Consider the below output:



From the above output, we can see that the file is removed from the staging area.

Add all New and Updated Files Only:

Git allows us to stage only updated and newly created files at once. We will use the ignore removal option to do so. It will be used as follows:

1. $ git add --ignore-removal .

Add all Modified and Deleted Files

Git add facilitates us with a variety of options. There is another option that is available in Git, which allows us to stage only the modified and deleted files. It will not stage the newly created file. To stage all modified and deleted files only, run the below command:

1. $ git add -u

Add Files by Wildcard

Git allows us to add all the same pattern files at once. It is another way to add multiple files together. Suppose I want to add all java files or text files, then we can use pattern .java or .txt. To do so, we will run the command as follows:

1. $ git add *.java

The above command will stage all the Java files. The same pattern will be applied for the text files.

The next step after adding files is committing to share it on Git.

Git Undo Add

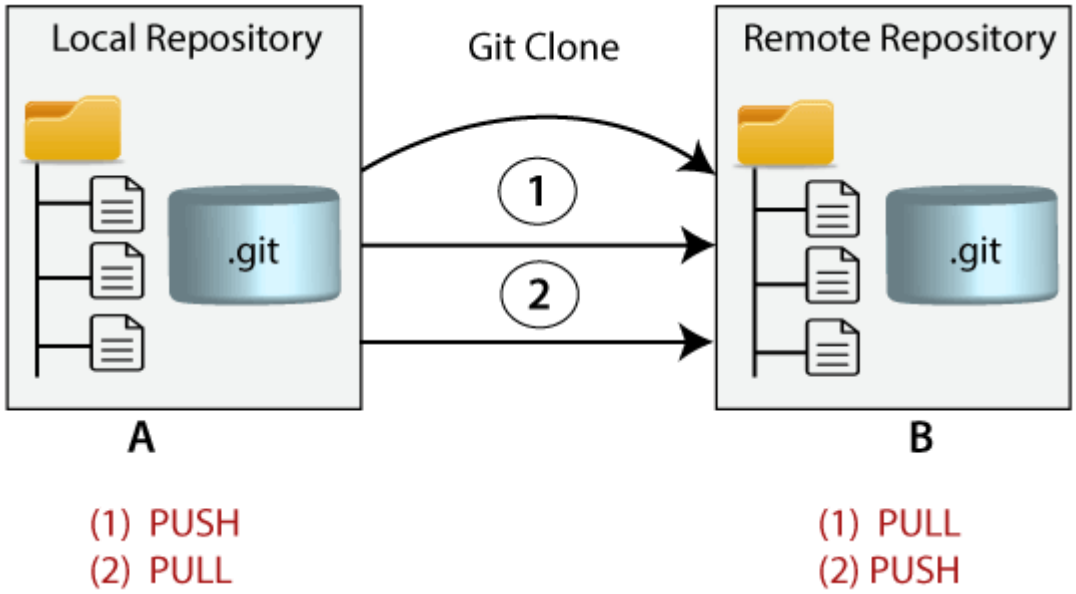
We can undo a git add operation. However, it is not a part of git add command, but we can do it through git reset command.

To undo an add operation, run the below command:

1. $ git reset <filename>

Git Clone

In Git, cloning is the act of making a copy of any target repository. The target repository can be remote or local. You can clone your repository from the remote repository to create a local copy on your system. Also, you can sync between the two locations.



Git Clone Command

The **git clone** is a command-line utility which is used to make a local copy of a remote repository. It accesses the repository through a remote URL.

Usually, the original repository is located on a remote server, often from a Git service like GitHub, Bitbucket, or GitLab. The remote repository URL is referred to the **origin**.

Syntax:

1. $ git clone **<repository** URL**>**

Git Clone Repository

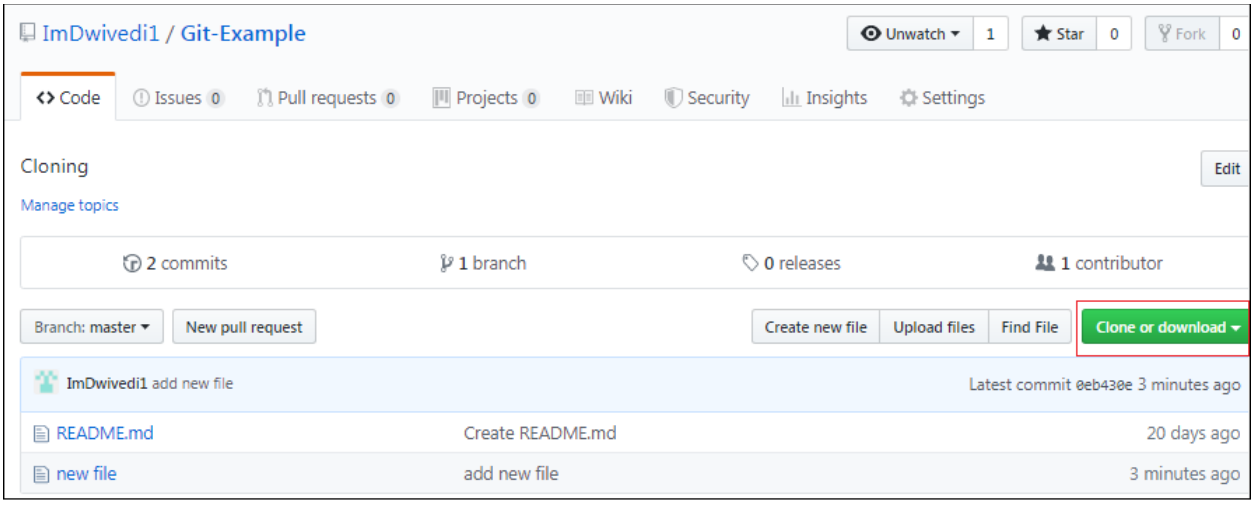
Suppose, you want to clone a repository from GitHub, or have an existing repository owned by any other user you would like to contribute. Steps to clone a repository are as follows:

Step 1:

Open GitHub and navigate to the main page of the repository.

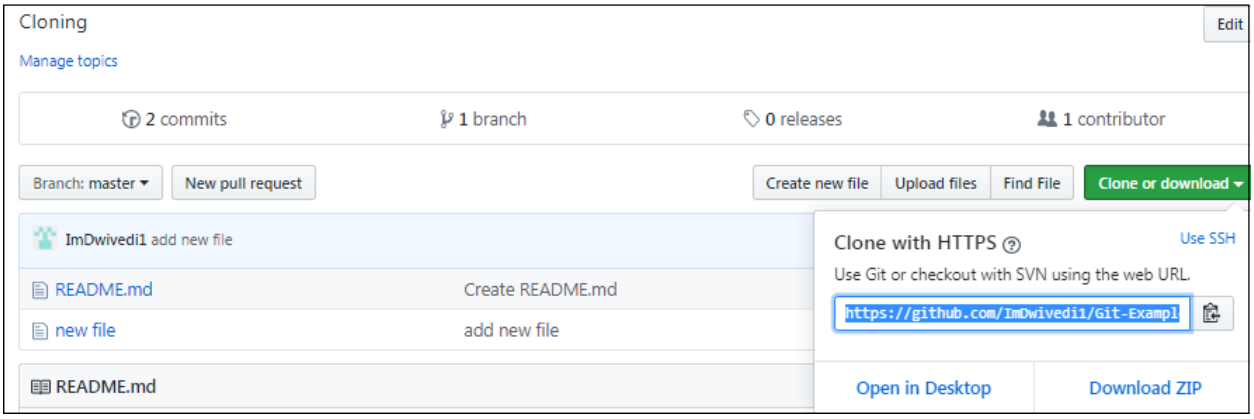
Step 2:

Under the repository name, click on **Clone or download**.



Step 3:

Select the **Clone with HTTPs section** and **copy the clone URL** for the repository. For the empty repository, you can copy the repository page URL from your browser and skip to next step.



Step 4:

Open Git Bash and change the current working directory to your desired location where you want to create the local copy of the repository.

Step 5:

Use the git clone command with repository URL to make a copy of the remote repository. See the below command:

1. $ git clone https://github.com/ImDwivedi1/Git-Example.git

Now, Press Enter. Hence, your local cloned repository will be created. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ cd "new folder"

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder (master)
$
```

Cloning a Repository into a Specific Local Folder

Git allows cloning the repository into a specific directory without switching to that particular directory. You can specify that directory as the next command-line argument in git clone command. See the below command:

1. $ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git clone https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"
Cloning into 'new folder(2)'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

The given command does the same thing as the previous one, but the target directory is switched to the specified directory.

Git has another transfer protocol called SSH protocol. The above example uses the git:// protocol, but you can also use http(s):// or user@server:/path.git, which uses the SSH transfer protocol.

Git Clone Branch

Git allows making a copy of only a particular branch from a repository. You can make a directory for the individual branch by using the git clone command. To make a clone branch, you need to specify the branch name with -b command. Below is the syntax of the command to clone the specific git branch:

Syntax:

1. $ git clone -b **<Branch** name**><Repository** URL**>**

See the below command:

1. $ git clone -b master https://github.com/ImDwivedi1/Git-Example.git "new folder(2)"

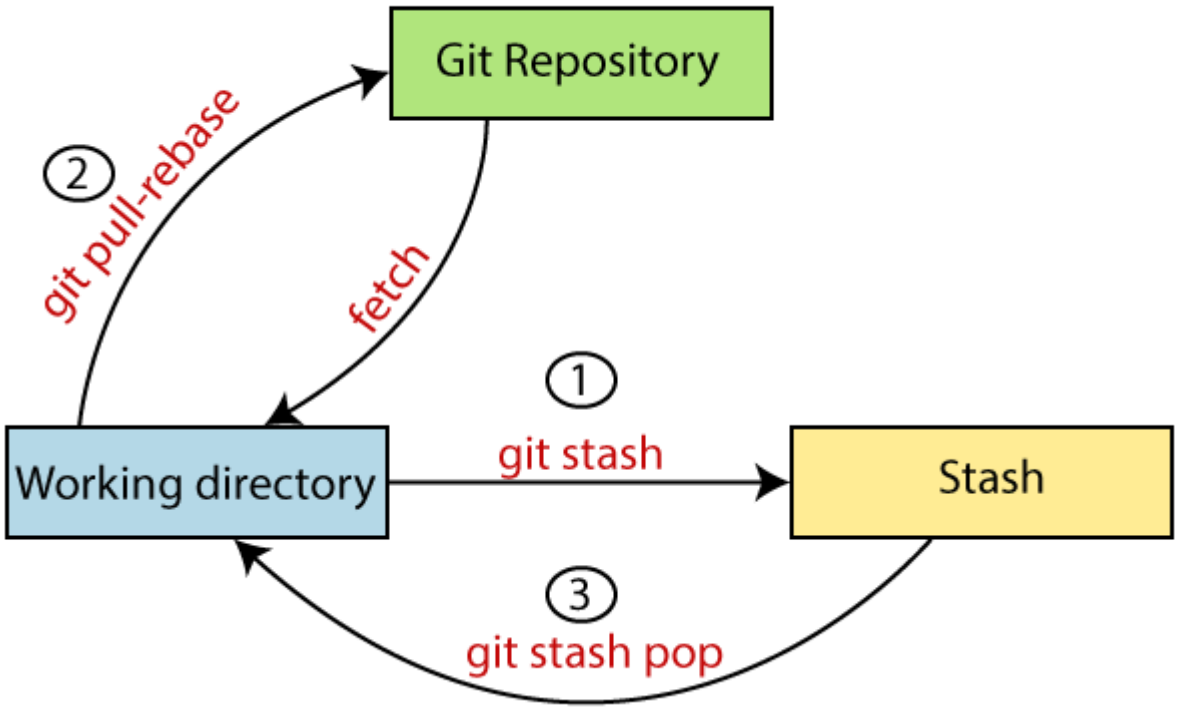
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$ git clone -b master https://github.com/ImDwivedi1/Git-Example.git
Cloning into 'Git-Example'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/new folder(2) (master)
$
```

In the given output, only the master branch is cloned from the principal repository Git-Example

Git Stash

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 branches without committing the current branch.

The below figure demonstrates the properties and role of stashing concerning repository and working directory.



Generally, the stash's meaning is "**store something safely in a hidden place**." The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

Stashing takes the messy state of your working directory, and temporarily save it for further use. Many options are available with 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**

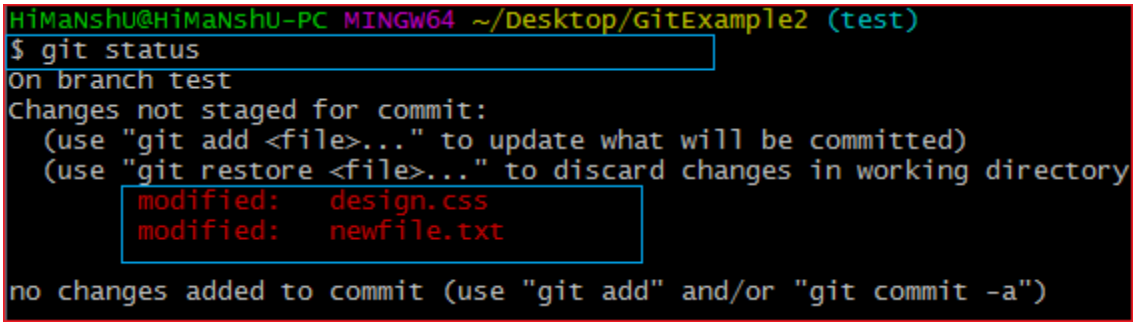
Stashing Work

Let's understand it with a real-time scenario. I have made changes to my project GitExample2 in two files from two distinct branches. I am in a messy state, and I have not entirely edited any file yet. So I want to save it temporarily for future use. We can stash it to save as its current status. To stash, let's have a look at the repository's current status. To check the current status of the repository, run the git status command. The git status command is used as:

Syntax:

1. $ git status

Output:

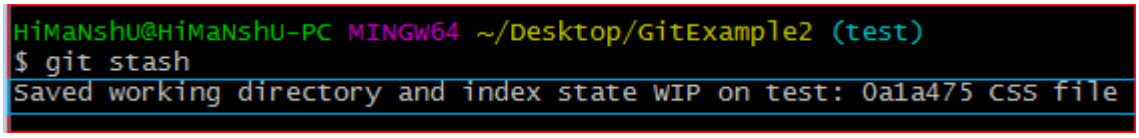


From the above output, you can see the status that there are two untracked file **design.css** and **newfile.txt** available in the repository. To save it temporarily, we can use the git stash command. The git stash command is used as:

Syntax:

1. $ git stash

Output:



In the given output, the work is saved with git stash command. We can check the status of the repository.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git status
On branch test
nothing to commit, working tree clean
```

As you can see, my work is just stashed in its current position. Now, the directory is cleaned. At this point, you can switch between branches and work on them.

Git Stash Save (Saving Stashes with the message):

In Git, the changes can be stashed with a message. To stash a change with a message, run the below command:

Syntax:

1. $ git stash save "**<Stashing** Message**>**"

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git stash save "Edited both files"
Saved working directory and index state On test: Edited both files
```

The above stash will be saved with a message

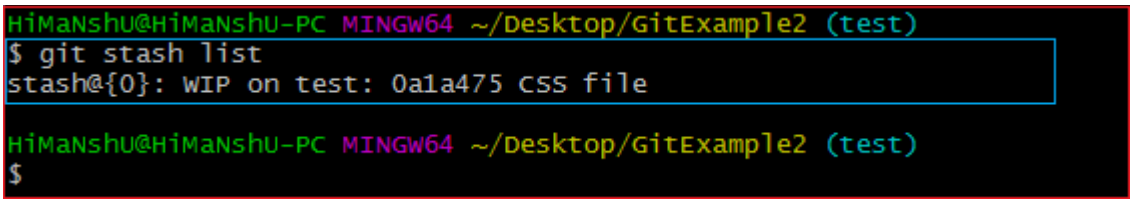
Git Stash List (Check the Stored Stashes)

To check the stored stashes, run the below command:

Syntax:

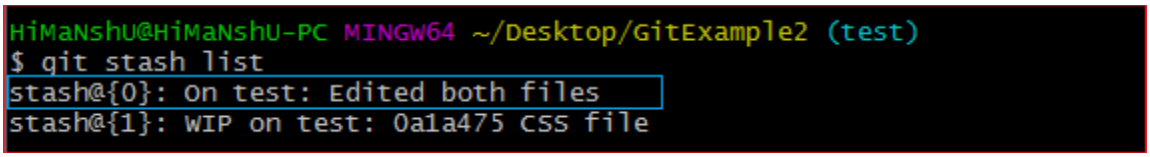
1. $ git stash list

Output:



In the above case, I have made one stash, which is displayed as "**stash@{0}: WIP on the test: 0a1a475 CSS file**".

If we have more than one stash, then It will display all the stashes respectively with different stash id. Consider the below output:



It will show all the stashes with indexing as **stash@{0}: stash@{1}:** and so on.

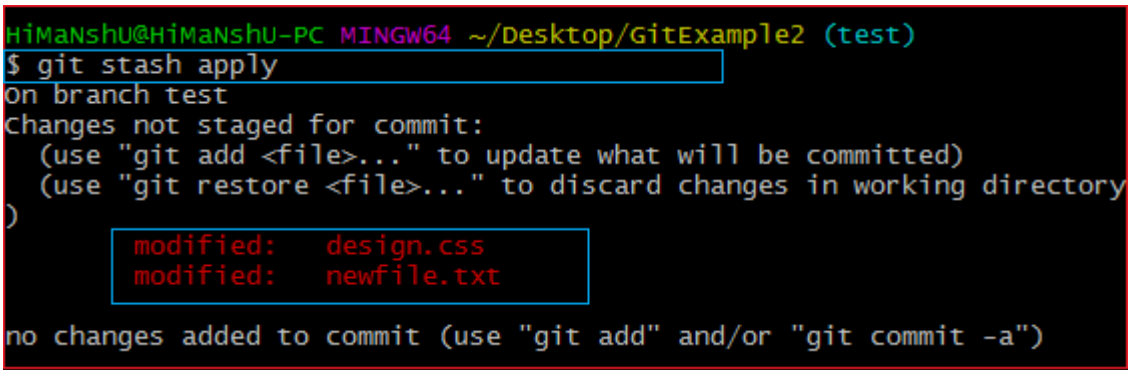
Git Stash Apply

You can re-apply the changes that you just stashed by using the git stash command. To apply the commit, use the git stash command, followed by the apply option. It is used as:

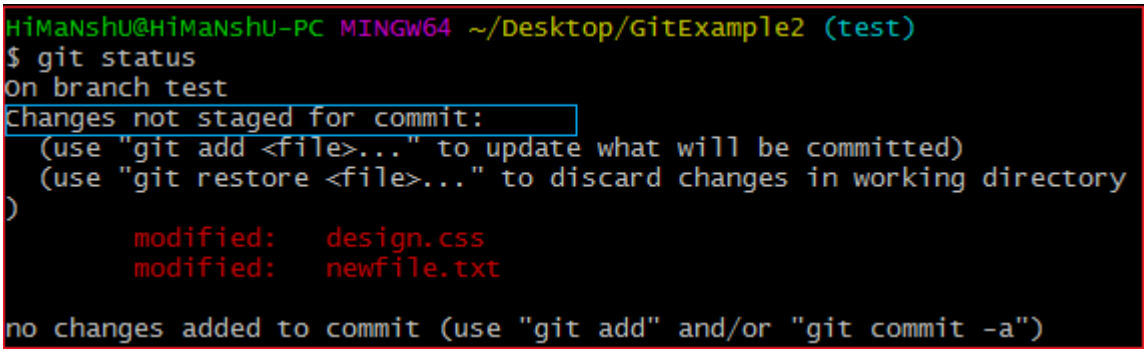
Syntax:

1. $ git stash apply

Output:



The above output restores the last stash. Now, if you will check the status of the repository, it will show the changes that are made on the file. Consider the below **output:**



From the above output, you can see that the repository is restored to its previous state before stash. It is showing output as "**Changes not staged for commit**."

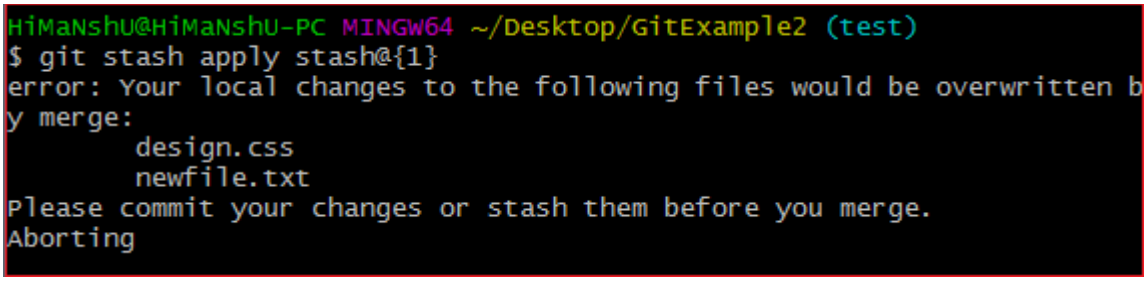
In case of more than one stash, you can use "git stash apply" command followed by stash index id to apply the particular commit. It is used as:

Syntax:

1. $ git stash apply **<stash** id**>**

Consider the below output:

Output:



If we don't specify a stash, Git takes the most recent stash and tries to apply it.

Git Stash Changes

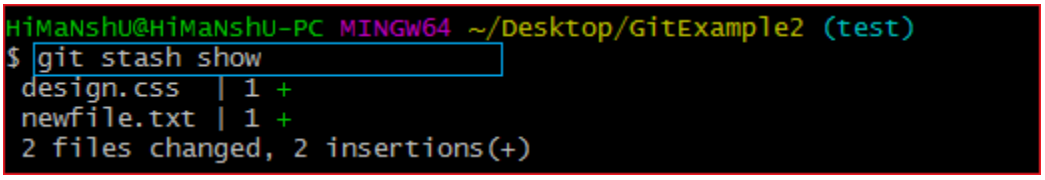
We can track the stashes and their changes. To see the changes in the file before stash and after stash operation, run the below command:

Syntax:

1. $ git stash show

The above command will show the file that is stashed and changes made on them. Consider the below output:

Output:



The above output illustrates that there are two files that are stashed, and two insertions performed on them.

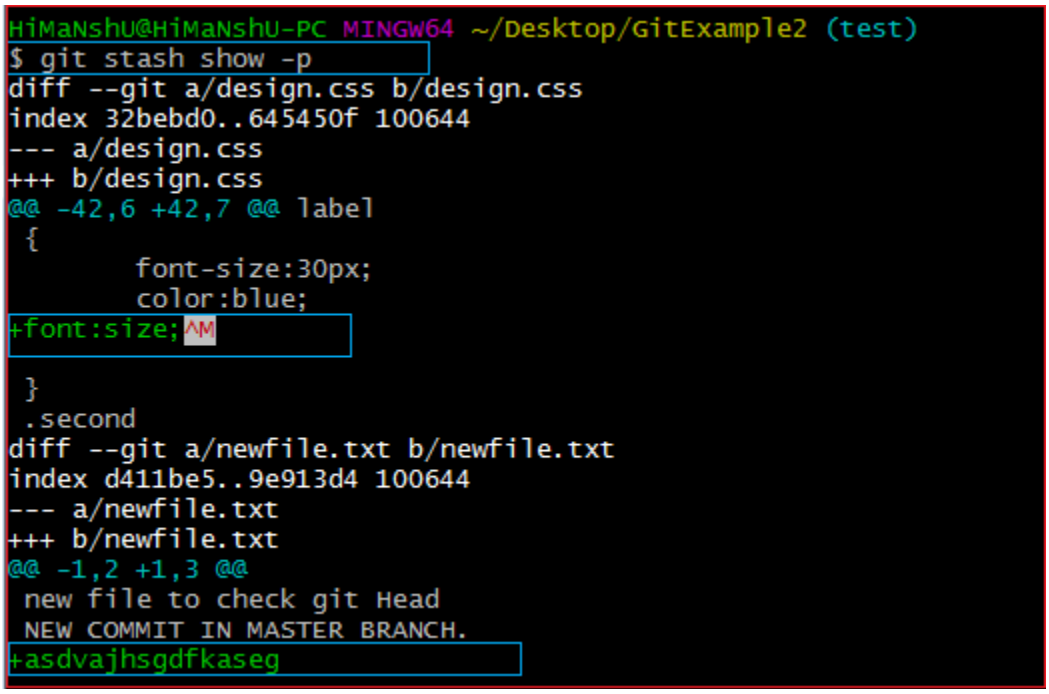
We can exactly track what changes are made on the file. To display the changed content of the file, perform the below command:

Syntax:

1. $ git stash show -p

Here, -p stands for the partial stash. The given command will show the edited files and content, consider the below output:

Output:



The above output is showing the file name with changed content. It acts the same as git diff command. The **git diff** command will also show the exact output.

Git Stash Pop (Reapplying Stashed Changes)

Git allows the user to re-apply the previous commits by using git stash pop command. The popping option removes the changes from stash and applies them to your working file.

The git stash pop command is quite similar to git stash apply. The main difference between both of these commands is stash pop command that deletes the stash from the stack after it is applied.

Syntax:

1. $ git stash pop

The above command will re-apply the previous commits to the repository. Consider the below output.

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash pop
On branch master
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:   newfile.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (55eb409b4135a9d378a6bd2e27940f405164573a)
```

Git Stash Drop (Unstash)

The **git stash drop** command is used to delete a stash from the queue. Generally, it deletes the most recent stash. Caution should be taken before using stash drop command, as it is difficult to undo if once applied.

The only way to revert it is if you do not close the terminal after deleting the stash. The stash drop command will be used as:

Syntax:

1. $ git stash drop

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash list
stash@{0}: WIP on master: 56afce0 Added an empty newfile2
stash@{1}: WIP on master: 56afce0 Added an empty newfile2
stash@{2}: WIP on test: 0a1a475 CSS file

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash drop
Dropped refs/stash@{0} (a9dc6ba6847ebcd2a69c16693359b516e6e8c3d9)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash list
stash@{0}: WIP on master: 56afce0 Added an empty newfile2
stash@{1}: WIP on test: 0a1a475 CSS file
```

In the above output, the most recent stash **(stash@{0})** has been dropped from given three stashes. The stash list command lists all the available stashes in the queue.

We can also delete a particular stash from the queue. To delete a particular stash from the available stashes, pass the stash id in stash drop command. It will be processed as:

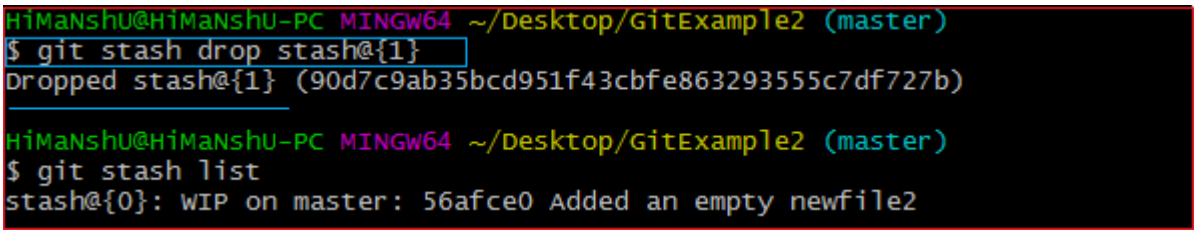
Syntax:

1. $ git stash drop **<stash** id**>**

Assume that I have two stashes available in my queue, and I don't want to drop my most recent stash, but I want to delete the older one. Then, it will be operated as:

1. $ git stash drop stash@{1}

Consider the below output:



In the above output, the commit **stash@{1}** has been deleted from the queue.

Git Stash Clear

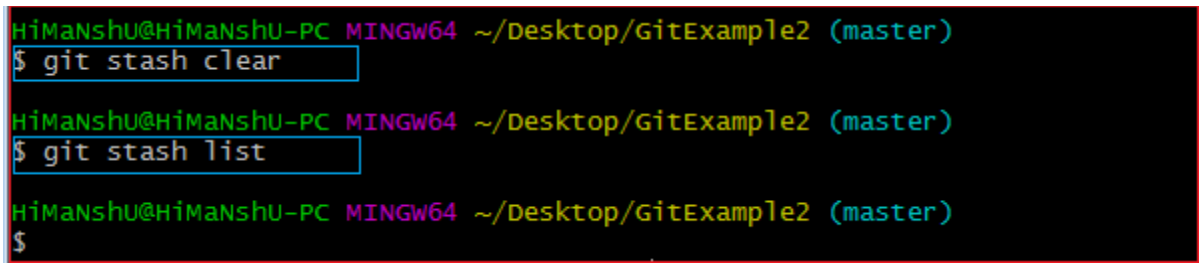
The **git stash clear** command allows deleting all the available stashes at once. To delete all the available stashes, operate below command:

Syntax:

1. $ git stash clear

it will delete all the stashes that exist in the repository.

Output:



All the stashes are deleted in the above output. The git stash list command is blank because there are no stashes available in the repository.

Git Stash Branch

If you stashed some work on a particular branch and continue working on that branch. Then, it may create a conflict during merging. So, it is good to stash work on a separate branch.

The git stash branch command allows the user to stash work on a separate branch to avoid conflicts. The syntax for this branch is as follows:

Syntax:

1. $ git stash branch **<Branch** Name**>**

The above command will create a new branch and transfer the stashed work on that. Consider the below output:

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash save "Demo for stash Branch"
Saved working directory and index state On master: Demo for stash Branch

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git stash branch testing
Switched to a new branch 'testing'
On branch testing
```

In the above output, the stashed work is transferred to a newly created branch testing. It will avoid the merge conflict on the master branch.

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 you don't want to send the files to Git service like GitHub. We can specify files in Git to 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.

Generally, the Ignored files are artifacts and machine-generated files. These files can be derived from your repository source or should otherwise not be committed. Some commonly ignored files are as follows:

- dependency caches
- compiled code
- build output directories, like /bin, /out, or /target
- runtime file generated, like .log, .lock, or .tmp
- Hidden system files, like Thumbs.db or.DS_Store
- Personal IDE config files, such as .idea/workspace.xml

Git Ignore Files

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.

There is no explicit git ignore command; instead, the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. The .gitignore files hold patterns that are matched against file names in your repository to determine whether or not they should be ignored.

How to Ignore Files Manually

There is no command in Git to ignore files; alternatively, there are several ways to specify the ignore files in git. One of the most common ways is the **.gitignore** file. Let's understand it with an example.

The .gitignore file:

Rules for ignoring file is defined in the .gitignore file. The .gitignore file is a file that contains all the formats and files of the ignored file. We can create multiple ignore files in a different directory. Let's understand how it works with an example:

Step1: Create a file named .gitignore if you do not have it already in your directory. To create a file, use the command touch or cat. It will use as follows:

1. $ touch .gitignore

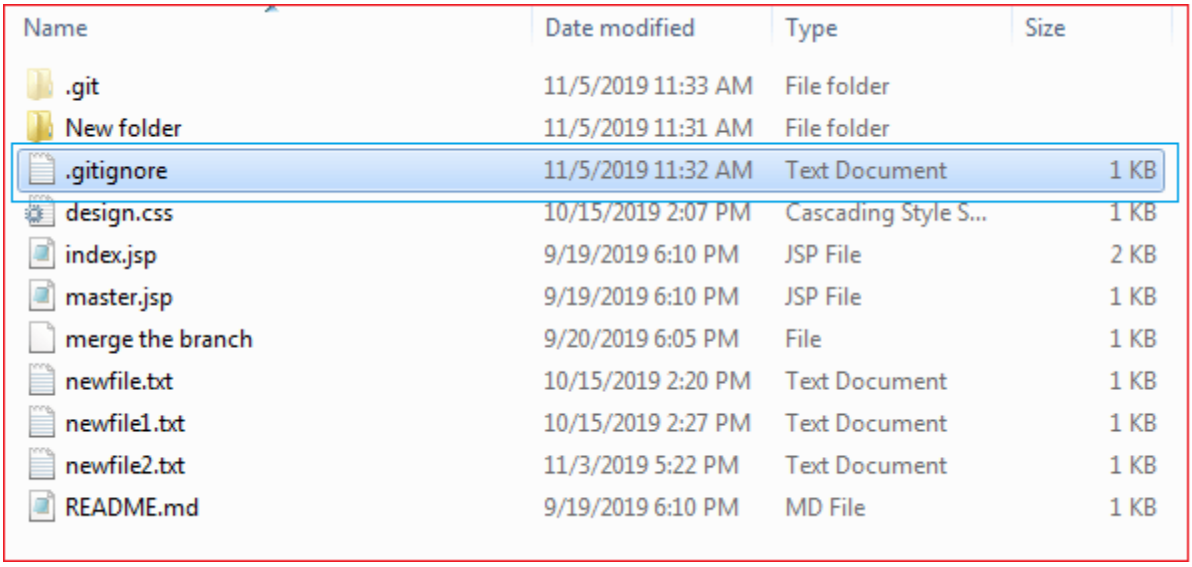
Or

1. $ cat .gitignore

The above command will create a .gitignore file on your directory. Remember, you are working on your desired directory. Consider the below command:

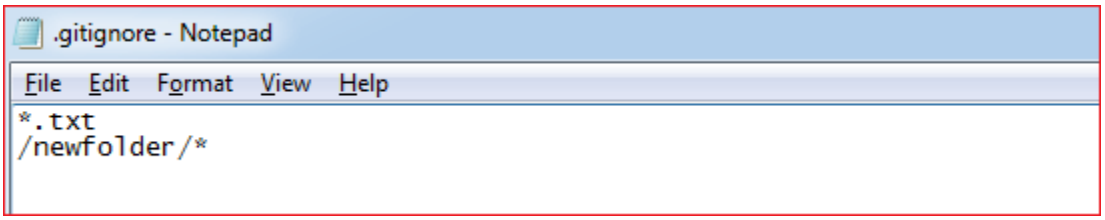
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ touch .gitignore
```

The above command will create a file named .gitignored. We can track it on the repository. Consider the below image:



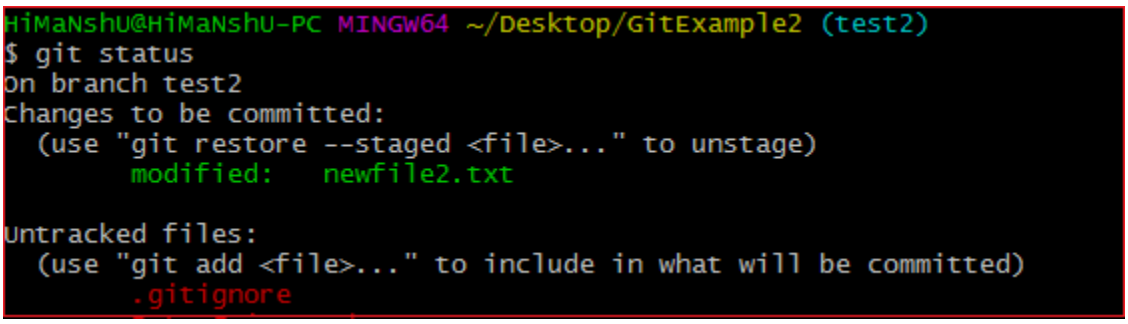
As you can see from the above image, a .gitignore file has been created for my repository.

Step2: Now, add the files and directories to the **.gitignore** file that you want to ignore. To add the files and directory to the .git ignore the file, open the file and type the file name, directory name, and pattern to ignore files and directories. Consider the below image:



In the above file, I have given one format and a directory to ignore. The above format ***.txt** will ignore all the text files from the repository, and **/newfolder/*** will ignore the newfolder and its sub-content. We can also give only the name of any file to ignore.

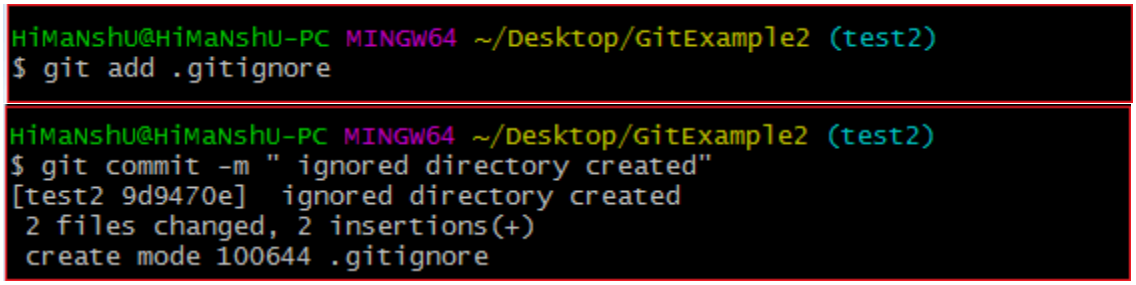
Step3: Now, to share it on Git, we have to commit it. The .gitignore file is still now in staging area; we can track it by git status command. Consider the below output:



Now to stage it, we have to commit it. To commit it, run the below command:

1. $ git add .gitignore
2. $ git commit -m "ignored directory created."

The above command will share the file .gitignore on Git. Consider the below output.



Now, we have ignored a pattern file and a directory in Git.

Rules for putting the pattern in .gitignore file:

The rules for the patterns that can be put in the .gitignore file are as follows:

- Git ignores the Blank lines or lines starting with #.
- Only the Standard glob patterns work and will be applied recursively throughout the entire working tree.
- The patterns can be started with a forward slash (/) to avoid recursively.
- The patterns can be ended with a forward slash (/) to specify a directory.
- The patterns can be negated by starting it with an exclamation point (!).

Global .gitignore:.

As we know that we can create multiple .gitignore files for a project. But Git also allows us to create a universal .gitignore file that can be used for the whole project. This file is known as a global .gitignore file. To create a global .gitignore, run the below command on terminal:

1. $ git config --global core.excludesfile ~/.gitignore_global

The above command will create a global .gitignore file for the repository.

How to List the Ignored Files?

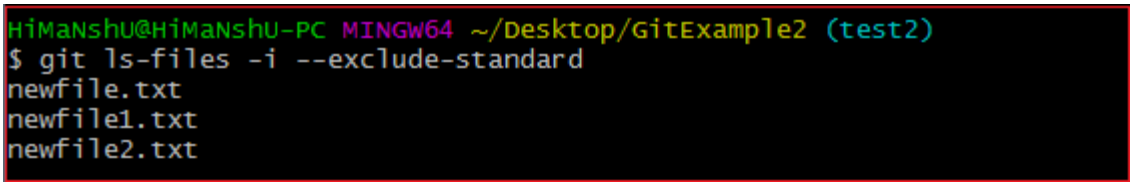
In Git, We can list the ignored files. There are various commands to list the ignored files, but the most common way to list the file is the **ls command**. To list the ignored file, run the ls command as follows:

1. $ git ls-files -i --exclude-standard

Or

1. $ git ls-files --ignore --exclude-standard

The above command will list all available ignored files from the repository. In the given command, **-I** option stands for ignore and **--exclude-standard** is specifying the exclude pattern. Consider the below output:



From the above output, we can see that the **ls** command is listing the available ignored files from the repository.

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.

Forking is not a Git function; it is a feature of Git service like GitHub.

When to Use Git 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.

Let's understand how to fork a repository on GitHub?

How to Fork a Repository?

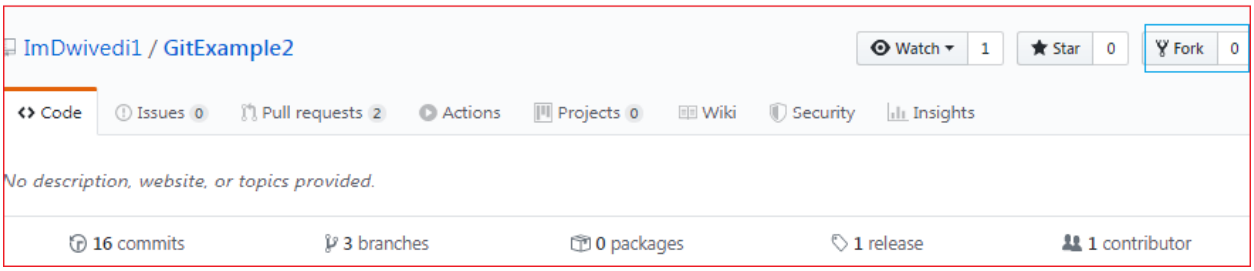
The forking and branching are excellent ways to contribute to an open-source project. These two features of Git allows the enhanced collaboration on the projects.

Forking is a safe way to contribute. It allows us to make a rough copy of the project. We can freely experiment on the project. After the final version of the project, we can create a pull request for merging.

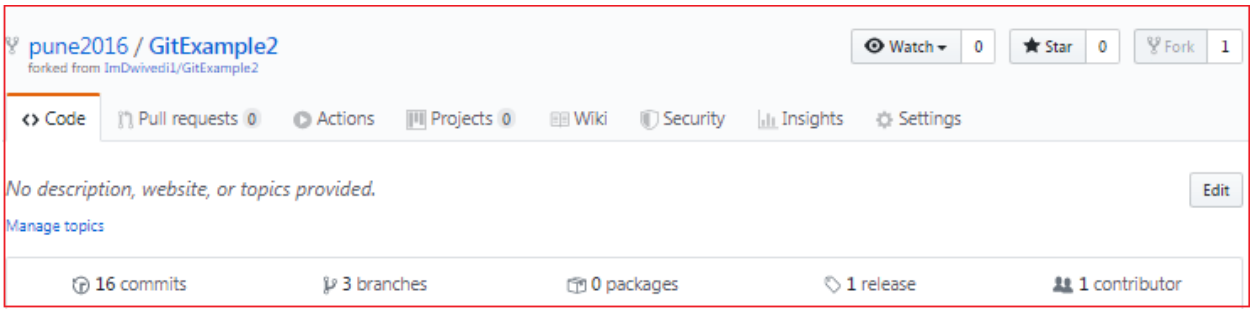
It is a straight-forward process. Steps for forking the repository are as follows:

- Login to the GitHub account.
- Find the GitHub repository which you want to fork.
- Click the Fork button on the upper right side of the repository's page.

We can't fork our own repository. Only shared repositories can be fork. If someone wants to fork the repository, then he must log in with his account. Let's understand the below scenario in which a user pune2016 wants to contribute to our project **GitExample2**. When he searches or put the address of our repository, our repository will look like as follows:



The above image shows the user interface of my repository from other contributors. We can see the fork option at the top right corner of the repository page. By clicking on that, the forking process will start. It will take a while to make a copy of the project for other users. After the forking completed, a copy of the repository will be copied to your GitHub account. It will not affect the original repository. We can freely make changes and then create a pull request for the main project. The owner of the project will see your suggestion and decide whether he wants to merge the changes or not. The fork copy will look like as follows:



As you can see, the forked repository looks like **pune2016/GitExample2**. At the bottom of the repository name, we can see a description of the repository. At the top right corner, the option fork is increased by 1 number.

Hence one can fork the repository from GitHub.

Fork vs. Clone

Sometimes people considered the fork as clone command because of their property. Both commands are used to create another copy of the repository. But the significant difference is that the 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.

A pull request can merge the changes made on the fork repository. We can create a pull request to propose changes to the project. Comparatively, changes made on the cloned repository can be merged by pushing. We can push the changes to our remote repository.

Git Repository

In Git, the 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 files as well as the history of changes made to those files. Repository in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

Getting a Git Repository

There are two ways to obtain a repository. They are as follows:

- Create a local repository and make it as Git repository.
- Clone a remote repository (already exists on a server).

In either case, you can start working on a Git repository.

Initializing a Repository

If you want to share your project on a version control system and control it with Git. Then, browse your project's directory and start the git command line (Git Bash for Windows) here. To initialize a new repository, run the below command:

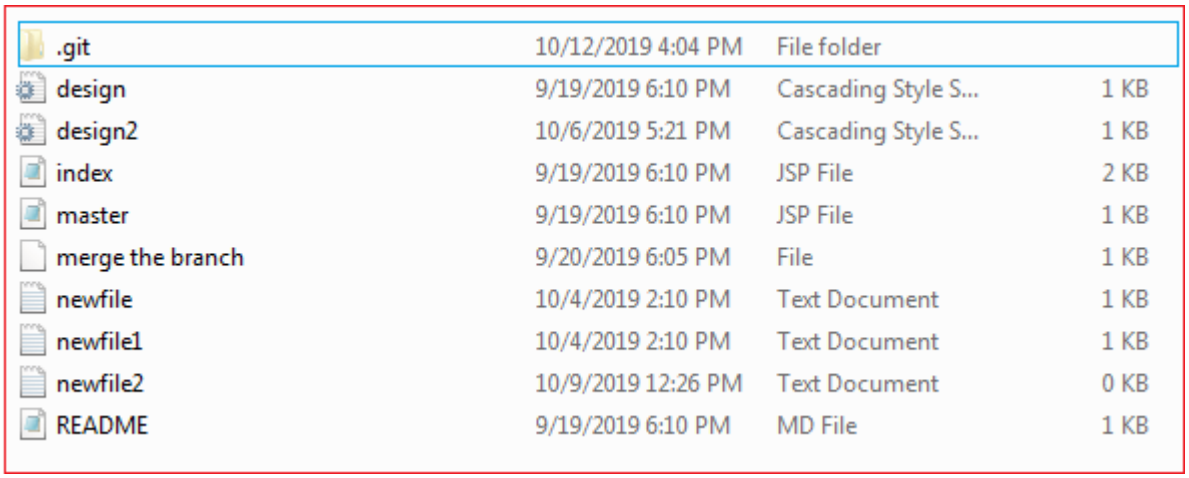
Syntax:

1. $ git init

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project
$ git init
Initialized empty Git repository in C:/My Project/.git/

HiMaNshU@HiMaNshU-PC MINGW64 /c/My Project (master)
$
```

The above command will create a new subdirectory named .git that holds all necessary repository files. The **.git** subdirectory can be understood as a Git repository skeleton. Consider the below image:



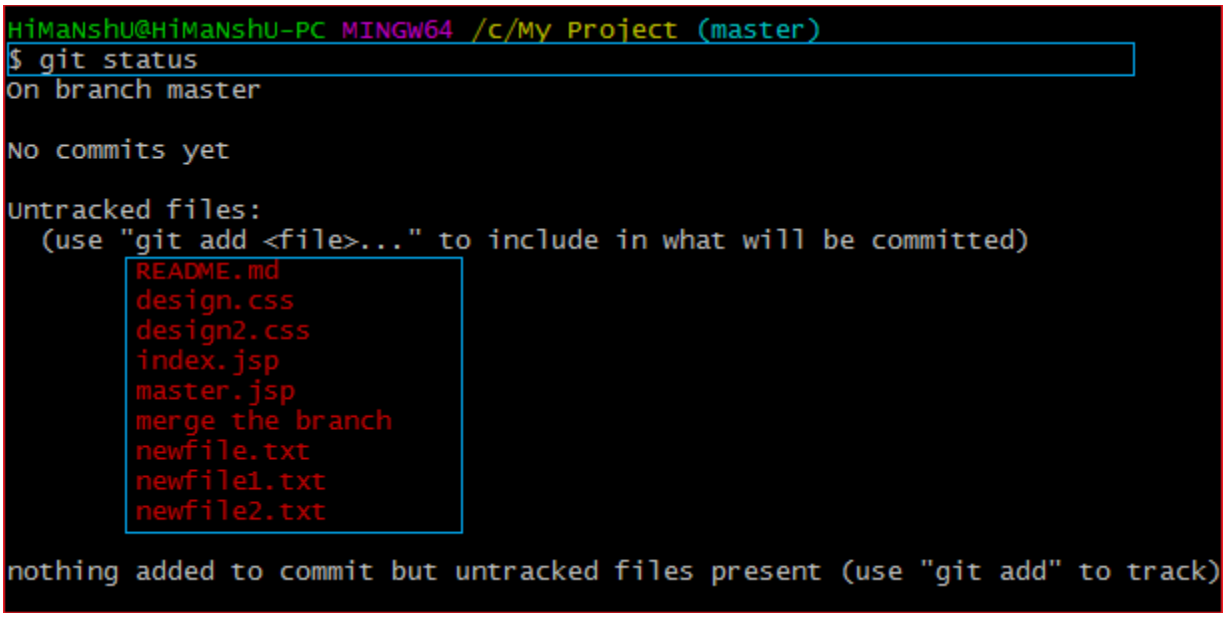
.git	10/12/2019 4:04 PM	File folder	
design	9/19/2019 6:10 PM	Cascading Style S...	1 KB
design2	10/6/2019 5:21 PM	Cascading Style S...	1 KB
index	9/19/2019 6:10 PM	JSP File	2 KB
master	9/19/2019 6:10 PM	JSP File	1 KB
merge the branch	9/20/2019 6:05 PM	File	1 KB
newfile	10/4/2019 2:10 PM	Text Document	1 KB
newfile1	10/4/2019 2:10 PM	Text Document	1 KB
newfile2	10/9/2019 12:26 PM	Text Document	0 KB
README	9/19/2019 6:10 PM	MD File	1 KB

An empty repository .git is added to my existing project. If we want to start version-controlling for existing files, we should track these files with git add command, followed by a commit.

We can list all the untracked files by git status command.

1. $ git status

Consider the below output:



In the above output, the list of all untracked files is displayed by the git status command. To share these files on the version control system, we have to track it with git add command followed by a commit. To track the files, operate git add command as follows:

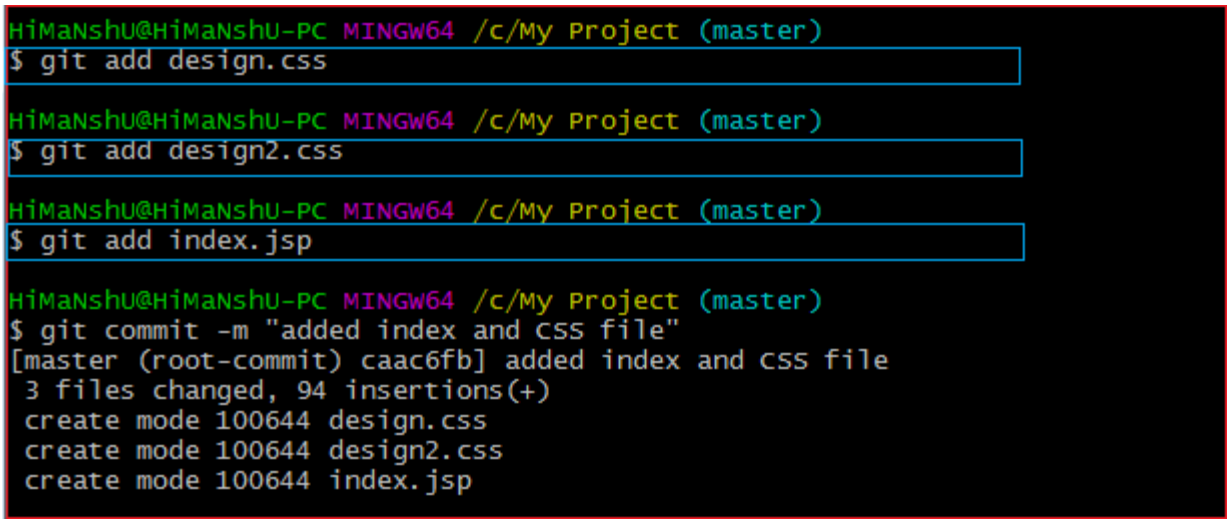
Syntax:

1. $ git add **<filename>**

To commit a file, perform the git commit command as follows:

1. $ git commit -m "Commit message."

Output:



In the above output, I have added three of my existing files by git add command and commit it for sharing.

We can also create new files. To share the new file, follow the same procedure as described above; add and commit it for sharing. Now, you have a repository to share.

Cloning an Existing Repository

We can clone an existing repository. Suppose we have a repository on a version control system like subversion, GitHub, or any other remote server, and we want to share it with someone to contribute. The git clone command will make a copy for any user to contribute.

We can get nearly all data from server with git clone command. It can be done as:

Syntax:

1. $ git clone **<Repository** URL**>**

Suppose one of my friends has a repository on my GitHub account, and I want to contribute to it. So the first thing I will do, make a copy of this project to my local system for a better work interface. The essential element needed for cloning the repository URL. I have a repository URL "https://github.com/ImDwivedi1/Git-Example". To clone this repository, operate the clone command as:

1. $ git clone https://github.com/ImDwivedi1/Git-Example

Consider the below output:

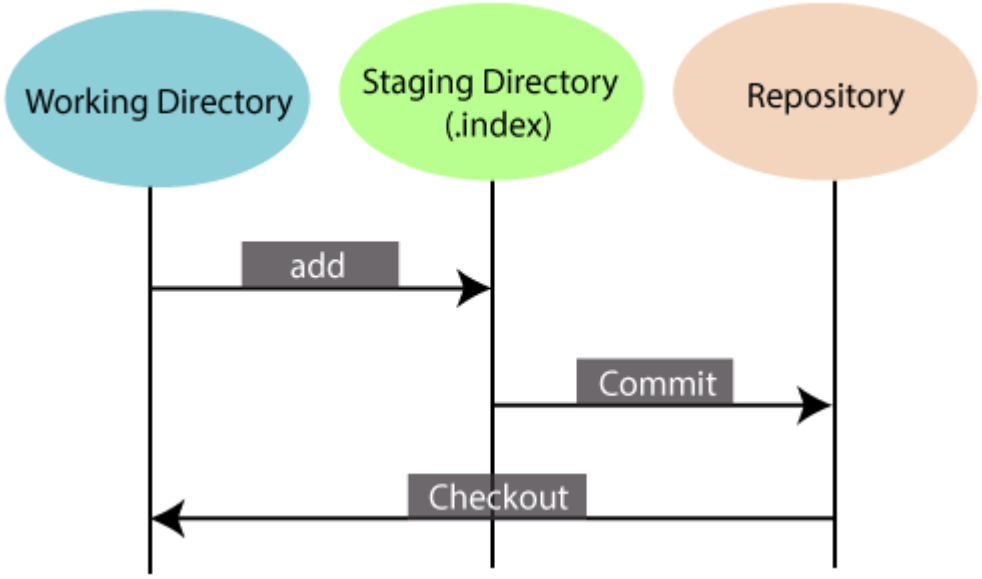
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo (master)
$ git clone https://github.com/ImDwivedi1/Git-Example
Cloning into 'Git-Example'...
remote: Enumerating objects: 23, done.
remote: Counting objects: 100% (23/23), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 23 (delta 5), reused 6 (delta 1), pack-reused 0
Unpacking objects: 100% (23/23), done.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo (master)
$
```

In the above output, the repository Git-Example has been cloned. Now this repository is available on your local storage. You can commit it and contribute to the project by pushing it on a remote server.

A single repository can be cloned any number of times. So we can clone a repository on various locations and various systems.

Git Index

The Git index is a staging area between the working directory and repository. It is used to build up a set of changes that you want to commit together. To better understand the Git index, then first understand the working directory and repository.



There are three places in Git where file changes can reside, and these are working directory, staging area, and the repository. To better understand the Git index first, let's take a quick view of these places.

Working directory:

When you worked on your project and made some changes, you are dealing with your project's working directory. This project directory is available on your computer's filesystem. All the changes you make will remain in the working directory until you add them to the staging area.

Staging area:

The staging area can be described as a preview of your next commit. When you create a git commit, Git takes changes that are in the staging area and make them as a new commit. You are allowed to add and remove changes from the staging area. The staging area can be considered as a real area 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.

Repository:

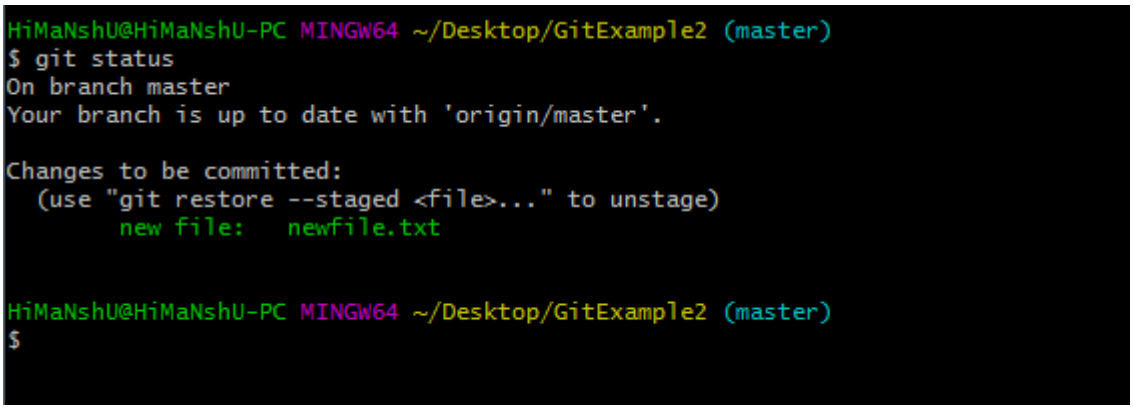
In Git, Repository is like a data structure used by GIt to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repositories in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

You can check what is in the index by the **git status command**. The git status command allows you to see which files are staged, modified but not yet staged, and completely untracked. Staged files mean, it is currently in the index. See the below example.

Syntax:

1. $ git status

Output:



In the given output, the status command shows the index.

As we mentioned earlier index is a file, not a directory, So Git is not storing objects into it. Instead, it stores information about each file in our repository. This information could be:

- **mtime:** It is the time of the last update.
- **file:** It is the name of the file.
- **Wdir:** The version of the file in the working directory.
- **Stage:** The version of the file in the index.
- **Repo:** The version of the file in the repository.

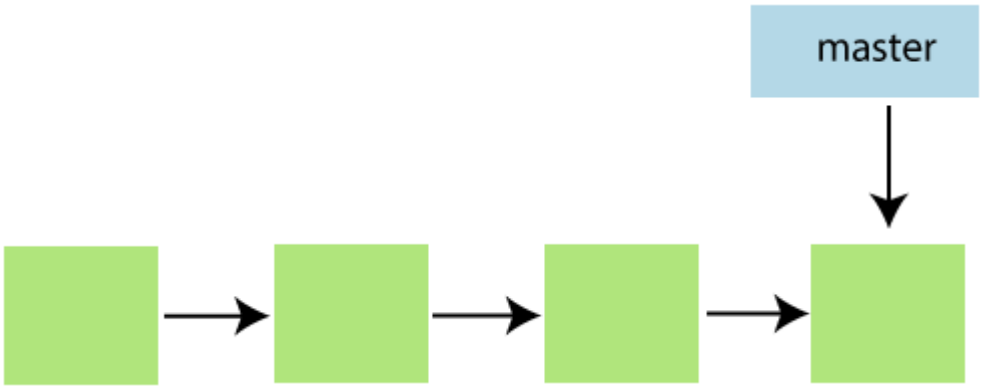
And finally, Git creates your working directory to match the content of the commit that HEAD is pointing.

Git Origin Master

The term "git origin master" is used in the context of a remote repository. It is used to deal with the remote repository. The term origin comes from where repository original situated and master stands for the main branch. Let's understand both of these terms in detail.

Git 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.

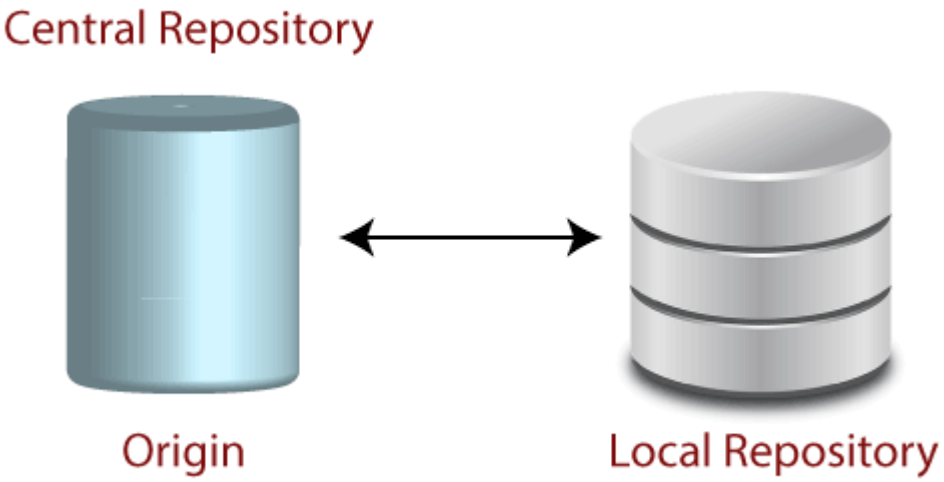


In most cases, the master is referred to as the main branch. Master branch is considered as the final view of the repo. Your local repository has its master branch that always up to date with the master of a remote repository.

Do not mess with the master. If you edited the master branch of a group project, your changes will affect everyone else and very quickly there will be merge conflicts.

Git Origin

In Git, The term origin is referred to the remote repository where you want to publish your commits. The default remote repository is called **origin**, although you can work with several remotes having a different name at the same time. It is said as an alias of the system.



The origin is a short name for the remote repository that a project was initially being cloned. It is used in place of the original repository URL. Thus, it makes referencing much easier.

Origin is just a standard convention. Although it is significant to leave this convention untouched, you could ideally rename it without losing any functionality.

In the following example, the URL parameter acts as an origin to the "clone" command for the cloned local repository:

1. $ git clone https://github.com/ImDwivedi1/Git-Example

Some commands in which the term origin and master are widely used are as follows:

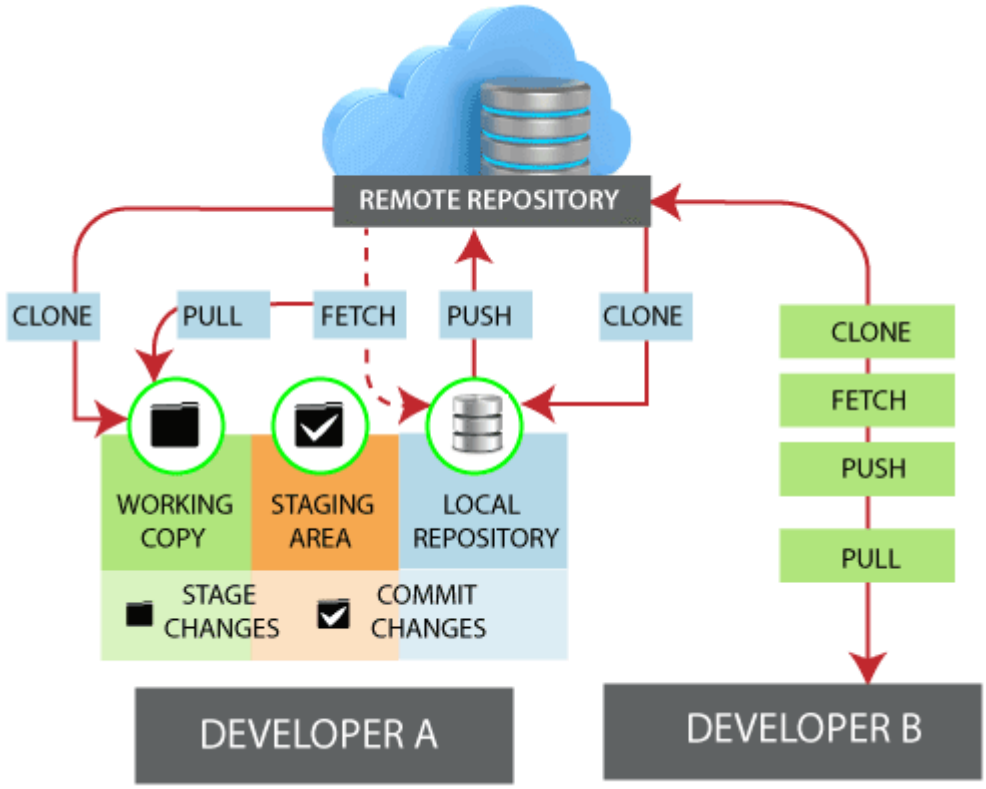
- Git push origin master
- Git pull origin master

Git has two types of branches called local and remote. To use git pull and git push, you have to tell your local branch that on which branch is going to operate. So, the term origin master is used to deal with a remote repository and master branch. The term **push origin master** is used to push the changes to the remote repository. The term **pull origin master** is used to access the repository from remote to local.

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.

The developers can perform many operations with the remote server. These operations can be a clone, fetch, push, pull, and more. Consider the below image:



Check your Remote

To check the configuration of the remote server, run the **git remote** command. The git remote command allows accessing the connection between remote and local. If you want to see the original existence of your cloned repository, use the git remote command. It can be used as:

Syntax:

1. $ git remote

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote
origin
```

The given command is providing the remote name as **the origin**. Origin is the default name for the remote server, which is given by Git.

Git remote -v:

Git remote supports a specific option -v to show the URLs that Git has stored as a short name. These short names are used during the reading and write operation. Here, -v stands for **verbose**. We can use **--verbose** in place of -v. It is used as:

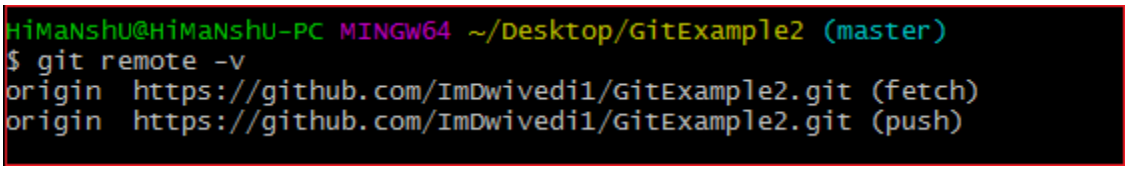
Syntax:

1. $ git remote -v

Or

1. $ git remote --verbose

Output:



The above output is providing available remote connections. If a repository contains more than one remote connection, this command will list them all.

Git Remote Add

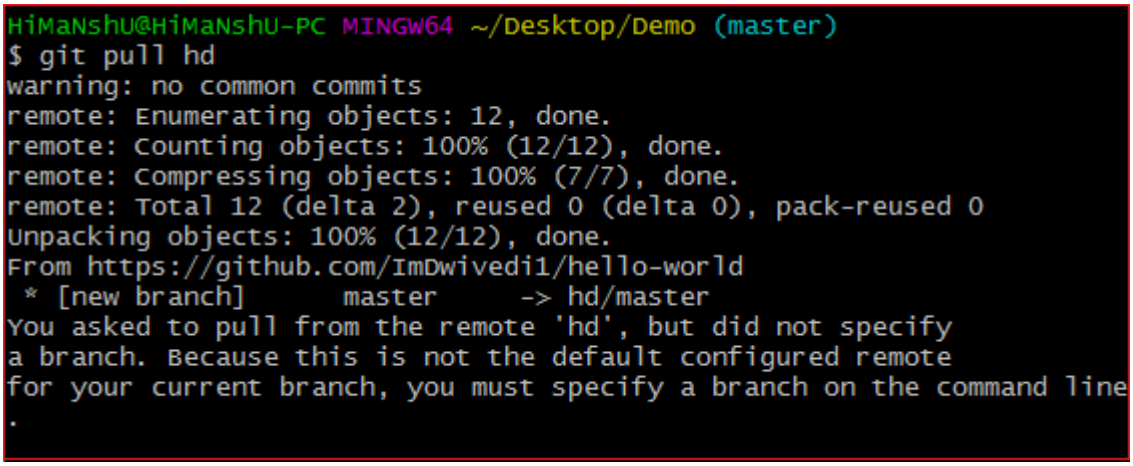
When we fetch a repository implicitly, git adds a remote for the repository. Also, we can explicitly add a remote for a repository. We can add a remote as a shot nickname or short name. To add remote as a short name, follow the below command:

Syntax:

1. $ git remote add <**short** name><remote URL>

Output:

In the above output, I have added a remote repository with an existing repository as a short name "**hd**". Now, you can use "**hd**" on the command line in place of the whole URL. For example, you want to pull the repository, consider below output:



I have pulled a repository using its short name instead of its remote URL. Now, the repository master branch can be accessed through a short name.

Fetching and Pulling Remote Branch

You can fetch and pull data from the remote repository. The fetch and pull command goes out to that remote server, and fetch all the data from that remote project that you don't have yet. These commands let us fetch the references to all the branches from that remote.

To fetch the data from your remote projects, run the below command:

1. $ git fetch <remote>

To clone the remote repository from your remote projects, run the below command:

1. $ git clone<remote>

When we clone a repository, the remote repository is added by a default name "**origin**." So, mostly, the command is used as git fetch origin.

The git fetch origin fetches the updates that have been made to the remote server since you cloned it. The git fetch command only downloads the data to the local repository; it doesn't merge or modify the data until you don't operate. You have to merge it manually into your repository when you want.

To pull the repository, run the below command:

1. $ git pull <remote>

The git pull command automatically fetches and then merges the remote data into your current branch. Pulling is an easier and comfortable workflow than fetching. Because the git clone command sets up your local master branch to track the remote master branch on the server you cloned.

Pushing to Remote Branch

If you want to share your project, you have to push it upstream. The git push command is used to share a project or send updates to the remote server. It is used as:

1. $ git push <remote><branch>

To update the main branch of the project, use the below command:

1. $ git push origin master

It is a special command-line utility that specifies the remote branch and directory. When you have multiple branches on a remote server, then this command assists you to specify your main branch and repository.

Generally, the term **origin** stands for the remote repository, and master is considered as the main branch. So, the entire statement "**git push origin master**" pushed the local content on the master branch of the remote location.

Git Remove Remote

You can remove a remote connection from a repository. To remove a connection, perform the git remote command with **remove** or **rm** option. It can be done as:

Syntax:

1. $ git remote rm <destination>

Or

1. $ git remote remove <destination>

Consider the below example:

Suppose you are connected with a default remote server "**origin**." To check the remote verbosely, perform the below command:

1. $ git remote -v

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v
origin  https://github.com/ImDwivedi1/GitExample2.git (fetch)
origin  https://github.com/ImDwivedi1/GitExample2.git (push)
```

The above output will list the available remote server. Now, perform the remove operation as mentioned above. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote rm origin

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git remote -v

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

In the above output, I have removed remote server "origin" from my repository.

Git Remote Rename

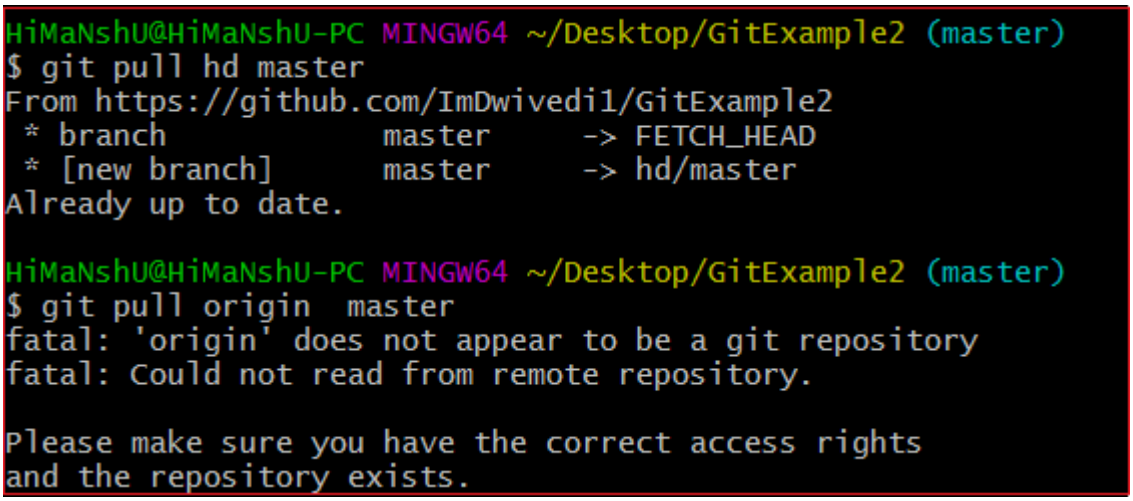
Git allows renaming the remote server name so that you can use a short name in place of the remote server name. Below command is used to rename the remote server:

Syntax:

1. $ git remote rename <old name><**new** name>

Output:

In the above output, I have renamed my default server name origin to hd. Now, I can operate using this name in place of origin. Consider the below output:



In the above output, I have pulled the remote repository using the server name hd. But, when I am using the old server name, it is throwing an error with the message "**'origin' does not appear to be a git repository**." It means Git is not identifying the old name, so all the operations will be performed by a new name.

Git Show Remote

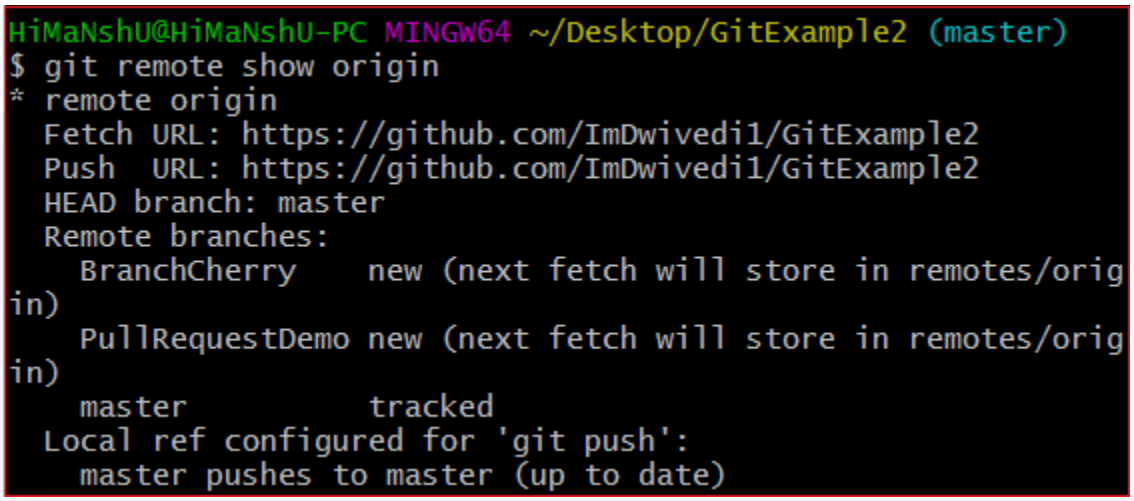
To see additional information about a particular remote, use the git remote command along with show sub-command. It is used as:

Syntax:

1. $ git remote show <remote>

It will result in information about the remote server. It contains a list of branches related to the remote and also the endpoints attached for fetching and pushing.

Output:



The above output is listing the URLs for the remote repository as well as the tracking branch information. This information will be helpful in various cases.

Git Change Remote (Changing a Remote's URL)

We can change the URL of a remote repository. The git remote set command is used to change the URL of the repository. It changes an existing remote repository URL.

Git Remote Set:

We can change the remote URL simply by using the git remote set command. Suppose we want to make a unique name for our project to specify it. Git allows us to do so. It is a simple process. To change the remote URL, use the below command:

1. $ git remote set-url <remote name><newURL>

The **remote set-url** command takes two types of arguments. The first one is <remote name >, it is your current server name for the repository. The second argument is <newURL>, it is your new URL name for the repository. The <new URL> should be in below format: **https://github.com/URLChanged**

Consider the below image:

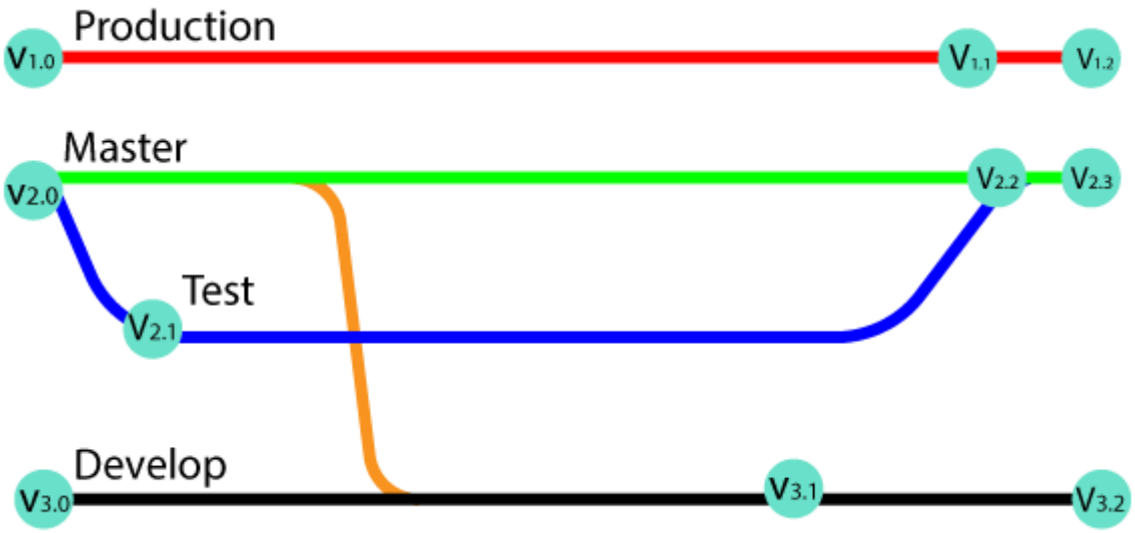
In the above output, I have changed my existing repository URL as **https://github.com/URLChanged** from **https://github.com/ImDwivedi1/GitExample2**. It can be understood by my URL name that I have changed this. To check the latest URL, perform the below command:

1. $ git remote -v

Git Tags

Tags make a point as a specific point in Git history. Tags are used to mark a commit stage as relevant. We can tag a commit for future reference. Primarily, it is used to mark a project's initial point like v1.1.

Tags are much like branches, and they do not change once initiated. We can have any number of tags on a branch or different branches. The below figure demonstrates the tags on various branches.



In the above image, there are many versions of a branch. All these versions are tags in the repository.

There are two types of tags.

- Annotated tag
- Light-weighted tag

Both of these tags are similar, but they are different in case of the amount of Metadata stores.

When to create a Tag:

- When you want to create a release point for a stable version of your code.
- When you want to create a historical point that you can refer to reuse in the future.

Git Create tag

To create a tag first, checkout to the branch where you want to create a tag. To check out the branch, run the below command:

1. $ git checkout **<Branch** name**>**

Now, you are on your desired branch, say, master. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (testing)
$ git checkout master
Switched to branch 'master'
```

You can create a tag by using the git tag command. Create a tag with some name say **v1.0, v1.1,** or any other name whatever you want. To create a tag, run the command as follows:

Syntax:

1. $ git tag **<tag** name**>**

The above command will mark the current status of the project. Consider the below example:

1. $ git tag projectv1.0

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag projectv1.0
```

The above command will create a mark point on the master branch as **projectv1.0**.

Git List Tag

We can list the available tags in our repository. There are three options that are available to list the tags in the repository. They are as follows:

- git tag
- git show
- git tag -l ".*"

The "git tag":

It is the most generally used option to list all the available tags from the repository. It is used as:

1. $ git tag

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag
projectv1.0
projectv1.1
```

As we can see from the above output, the git tag command is listing the available tags from the repository.

The git tag show <tagname>:

It's a specific command used to display the details of a particular tag. It is used as:

Syntax:

1. $ git tag show **<tagname>**

The above command will display the tag description, consider the below command:

1. $ git tag show projectv1.0

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git show projectv1.0
commit 56afce0ea387ab840819686ec9682bb07d72add6 (HEAD -> master, tag:
projectv1.1, tag: projectv1.0, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29
```

In the above output, the git show tag is displaying the description of tag **projectv1.0**, such as author name and date.

The git tag -l ".*":

It is also a specific command-line tool. It displays the available tags using wild card pattern. Suppose we have ten tags as v1.0, v1.1, v1.2 up to v1.10. Then, we can list all v pattern using tag pattern v. it is used as:

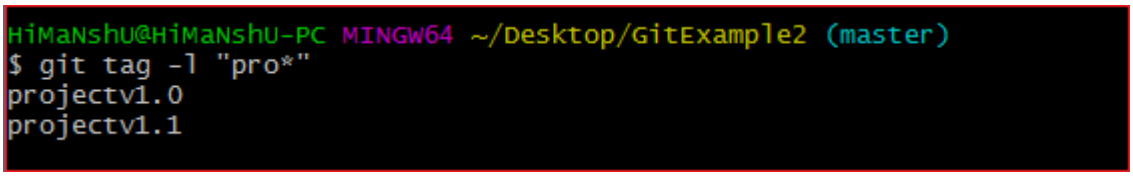
Syntax:

1. $ git tag -l "**<pattern>**.*"

The above command will display all the tags that contain wild card characters. Consider the below command:

1. $ git tag -l "pro*"

Output:



The above command is displaying a list of the tags that started with a word **pro**.

Types of Git tags

There are two types of tags in git. They are as:

- Annotated tag
- Light-weighted tag

Let's understand both of these tags in detail.

Annotated Tags

Annotated tags are tags that store extra Metadata like developer name, email, date, and more. They are stored as a bundle of objects in the Git database.

If you are pointing and saving a final version of any project, then it is recommended to create an annotated tag. But if you want to make a temporary mark point or don't want to share information, then you can create a light-weight tag. The data provided in annotated tags are essential for a public release of the project. There are more options available to annotate, like you can add a message for annotation of the project.

To create an annotated tag, run the below command:

Syntax:

1. $ git tag **<tag** name**>** -m "**< Tag** message**>**

The above command will create a tag with a message. Annotated tags contain some additional information like author name and other project related information. Consider the below image:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag projectv1.1 -m "It's a relaese point for projectv1.1"
```

The above command will create an annotated tag **projectv1.1** in the master branch of my project's repository.

When we display an annotated tag, it will show more information about tag. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git show projectv1.1
tag projectv1.1
Tagger: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 16 11:53:23 2019 +0530

It's a relaese point for projectv1.1

commit 56afce0ea387ab840819686ec9682bb07d72add6 (HEAD -> master, tag:
projectv1.1, tag: projectv1.0, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29
```

Light-Weighted Tag:

Git supports one more type of tag; it is called as Light-weighted tag. The motive of both tags is the same as marking a point in the repository. Usually, it is a commit stored in a file. It does not store unnecessary information to keep it light-weight. No command-line option such as **-a,-s** or **-m** are supplied in light-weighted tag, pass a tag name.

Syntax:

1. $ git tag **<tag** name**>**

The above command will create a light-weight tag. Consider the below example:

1. $ git tag projectv1.0

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag projectv1.0
```

The given output will create a light-weight tag named **projectv1.0**.

It will display a reduced output than an annotated tag. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git show projectv1.0
commit 56afce0ea387ab840819686ec9682bb07d72add6 (HEAD -> master, tag:
projectv1.1, tag: projectv1.0, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29
```

Git Push Tag

We can push tags to a remote server project. It will help other team members to know where to pick an update. It will show as **release point** on a remote server account. The git push command facilitates with some specific options to push tags. They are as follows:

- Git push origin <tagname>
- Git push origin -tags/ Git push --tags

The git push origin :

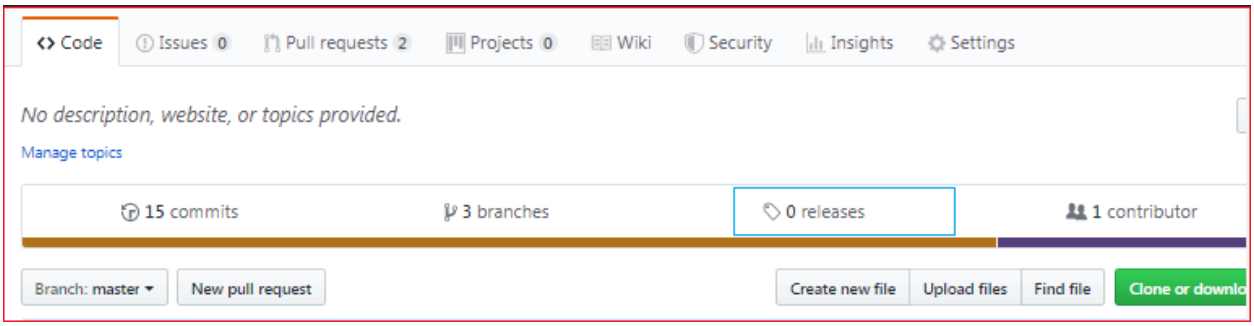
We can push any particular tag by using the git push command. It is used as follows:

Syntax:

1. $ git push origin **<tagname>**

The above command will push the specified tag name as a release point. Consider the below example:

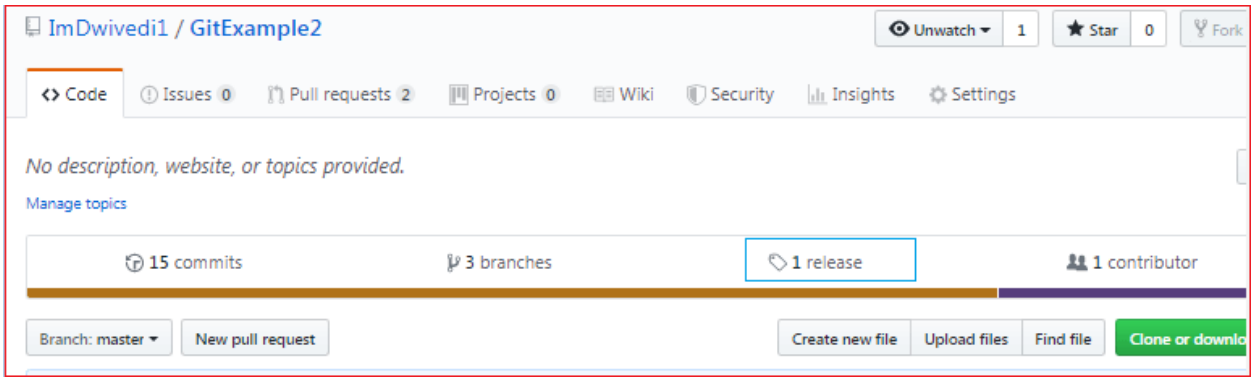
I have created some tags in my local repository, and I want to push it on my GitHub account. Then, I have to operate the above command. Consider the below image; it is my remote repository current status.



The above image is showing the release point as **0 releases**. Now, perform the above command. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin projectv1.0
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/ImDwivedi1/GitExample2
 * [new tag]         projectv1.0 -> projectv1.0
```

I have pushed my projectv1.0 tag to the remote repository. It will change the repository's current status. Consider the below image:



By refreshing the repository, it is showing release point as **1 release**. We can see this release by clicking on it. It will show as:

We can download it as a zip and tar file.

The git push origin --tag/ git push --tags:

The given command will push all the available tags at once. It will create as much release point as the number of tags available in the repository. It is used as follows:

Syntax:

1. $ git push origin --tags

Or

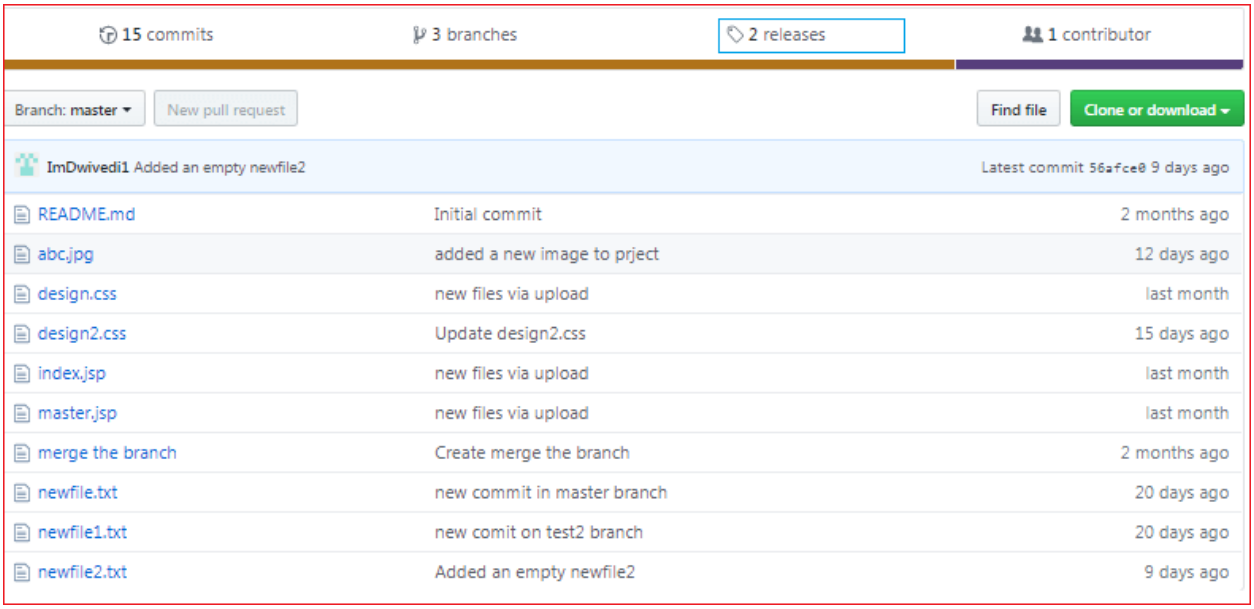
1. $ git push --tags

The above command will push all the available tags from the local repository to the remote repository. Consider the below output:

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 186 bytes | 62.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/ImDwivedi1/GitExample2
 * [new tag]         projectv1.1 -> projectv1.1
```

Tags have been pushed to remote server origin; thus, the release point is also updated. Consider the below snapshot of the repository:



The release point is updated in the above output according to tags. You can see that releases updated as **2 releases**.

Git Delete Tag

Git allows deleting a tag from the repository at any moment. To delete a tag, run the below command:

Syntax:

1. $git tag --d **<tagname>**

Or

1. $ git tag --delete **<tagname>**

The above command will delete a particular tag from the local repository. Suppose I want to delete my tag **projectv1.0** then the process will be as:

1. $ git tag --d projectv1.0

Consider below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag --d projectv1.0
Deleted tag 'projectv1.0' (was 56afce0)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git tag
projectv1.1
```

The tag projectv1.0 has been deleted from the repository.

Delete a Remote Tag:

We can also delete a tag from the remote server. To delete a tag from the remote server, run the below command:

Syntax:

1. $ git push origin -d **<tagname>**

Or

1. $ git push origin --delete**<tag** name**>**

The above command will delete the specified tag from the remote server. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin -d projectv1.0
To https://github.com/ImDwivedi1/GitExample2
 - [deleted]         projectv1.0
```

The **projectv1.0** tag has been deleted from the remote server origin.

Delete Multiple Tags:

We can delete more than one tag just from a single command. To delete more than one tag simultaneously, run the below command:

Syntax:

1. $ git tag -d **<tag1> <tag2>**

Output:

The above command will delete both the tags from the local repository.

We can also delete multiple tags from the remote server. To delete tags from the server origin, run the below command:

1. $ git push origin -d **<tag1> <tag2>**

The above command will delete both tags from the server.

Git Checkout Tags

There is no actual concept of check out the tags in git. However, we can do it by creating a new branch from a tag. To check out a tag, run the below command:

Syntax:

1. $ git checkout -b **< new** branch name**> <tag** name**>**

The above command will create a new branch with the same state of the repository as it is in the tag. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout -b new_branchV1.1 projectv1.1
Switched to a new branch 'new_branchV1.1'
M       newfile2.txt
```

The above command will create a new branch and transfer the status of the repository to **new_branchv1.1** as it is on tag projectv1.1.

Create a tag from an older commit:

If you want to go back to your history and want to create a tag on that point. Git allows you to do so. To create a tag from an older commit, run the below command:

1. **< git** tag **<tagname> < reference** of commit**>**

In the above command, it is not required to give all of the 40 digit number; you can give a part of it.

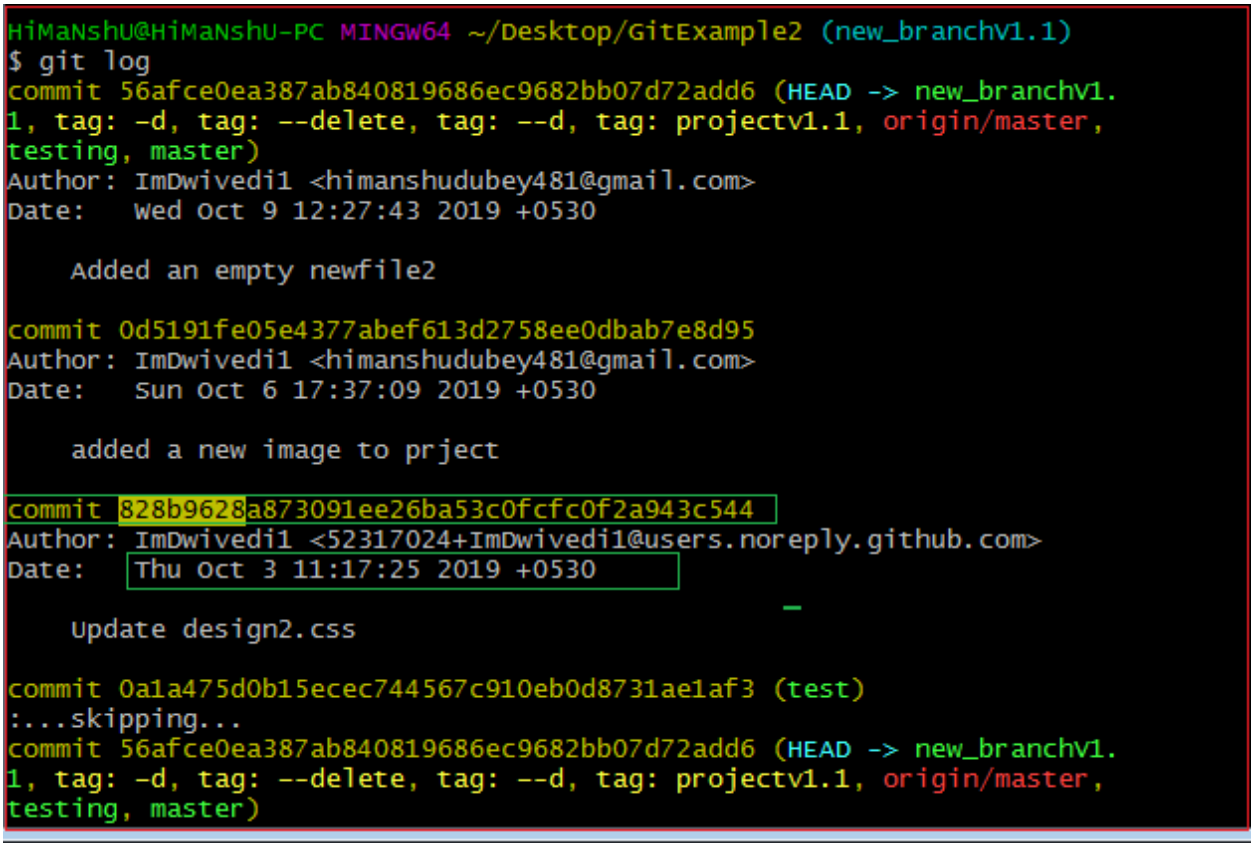
Suppose I want to create a tag for my older commit, then the process will be as follows:

Check the older commits:

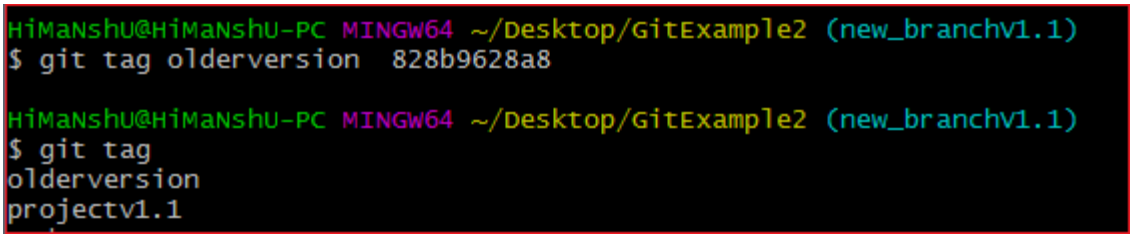
To check the older commit, run the git status command. It will operate as follows:

1. $ git status

Consider the below output:



The above output is showing the older commits. Suppose I want to create a tag for my commit, starting with **828b9628**. Copy the particular reference of the commit. And pass it as an argument in the above command. Consider the below output:



In the above output, an earlier version of the repository is tagged as an **olderversion**.

Upstream and Downstream

The term upstream and downstream refers to the repository. Generally, upstream is from where you clone the repository, and downstream is any project that integrates your work with other works. However, these terms are not restricted to Git repositories.

There are two different contexts in Git for upstream/downstream, which are remotes and time/history. In the reference of remote upstream/downstream, the downstream repo will be pulled from the upstream repository. Data will flow downstream naturally.

In the reference of time/history, it can be unclear, because upstream in time means downstream in history, and vice-versa. So it is better if we use the parent/child terms in place of upstream/downstream in case of time/history.

Git set-upstream

The git set-upstream allows you to set the default remote branch for your current local branch. By default, every pull command sets the master as your default remote branch.

Sometimes we are trying to push some changes to the remote server, but it will show the error like "**error: failed to push some refs to 'https :< remote repository Address>**." There may be the reason that you have not set your remote branch. We can set the remote branch for the local branch. We will implement the following process to set the remote server:

To check the remote server, use the below command:

1. $ git remote -v

It will result as follows:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git remote -v
origin  https://github.com/ImDwivedi1/GitExample2 (fetch)
origin  https://github.com/ImDwivedi1/GitExample2 (push)
```

The above output is displaying the remote server name. To better understand remote server, Click here. Now, check the available branches, run the below command:

1. $ git branch -a

It will result as follows:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git branch -a
* master
  new_branchV1.1
  test
  test2
  testing
  remotes/origin/BranchCherry
  remotes/origin/PullRequestDemo
  remotes/origin/master
```

The above command will list the branches on the local and remote repository. To learn more about branches, click here. Now push the changes to remote server and set the particular branch as default remote branch for the local repository. To push the changes and set the remote branch as default, run the below command:

1. $ git push --set-upstream origin master

The above command will set the master branch as the default remote branch. To better understand the origin master click here.

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git push --set-upstream origin master
Everything up-to-date
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

In the given output, everything is up to date with the remote branch.

We can also set the default remote branch by using the git branch command. To do so, run the below command:

1. $ git branch --set-upstream-to origin master

To display default remote branches, run the below command:

1. $ git branch -vv

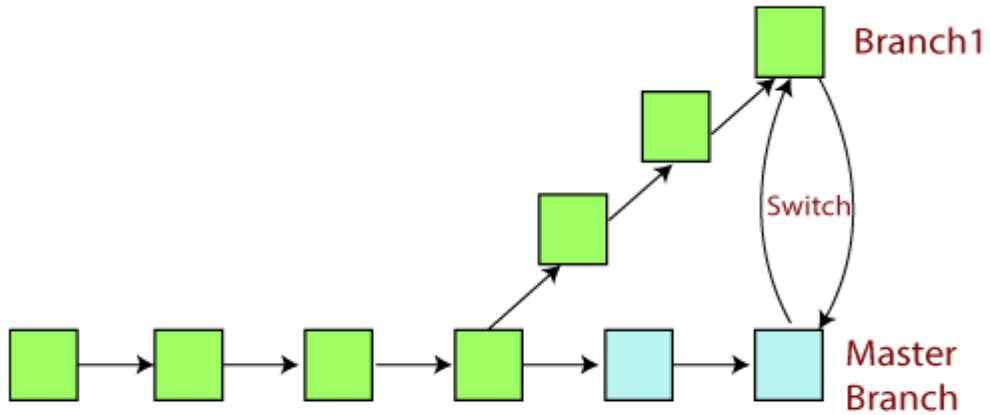
Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master|REBASE-i)
$ git branch -vv
* master          a3a4f45 [origin/master] Removed last lione from the repository
  new_branchV1.1 1778984 Revert "new file2 is edited"
  test           0a1a475 CSS file
  test2          5f8aab1 added Changes
  testing        56afce0 Added an empty newfile2
```

The above output is displaying the branches available on the repository. We can see that the default remote branch is specified by highlighted letters.

Git 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. Be careful with your staged files and commits when switching between branches.



Git Checkout

The git checkout command operates upon three different entities which are files, commits, and branches. Sometimes this command can be dangerous because there is no undo option available on this command.

It checks the branches and updates the files in the working directory to match the version already available in that branch, and it forwards the updates to Git to save all new commit in that branch.

Operations on Git Checkout

We can perform many operations by git checkout command like the switch to a specific branch, create a new branch, checkout a remote branch, and more. The **git branch** and **git checkout** commands can be integrated.

Checkout Branch

You can demonstrate how to view a list of available branches by executing the git branch command and switch to a specified branch.

To demonstrate available branches in repository, use the below command:

1. $ git branch

Now, you have the list of available branches. To switch between branches, use the below command.

Syntax:

1. $ git checkout <branchname>

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
  TestBranch
* master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout testbranch
Switched to branch 'testbranch'
```

As you can see in the given output that master branch has switched to TestBranch.

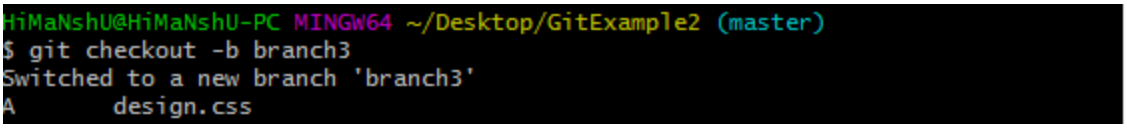
Create and Switch Branch

The git checkout commands let you create and switch to a new branch. You can not only create a new branch but also switch it simultaneously by a single command. The git checkout -b option is a convenience flag that performs run git branch <new-branch>operation before running git checkout <new-branch>.

Syntax:

1. $ git checkout -b <branchname>

Output:



As you can see in the given output, branch3 is created and switched from the master branch.

Checkout Remote Branch

Git allows you to check out a remote branch by git checkout command. It is a way for a programmer to access the work of a colleague or collaborator for review and collaboration. Each remote repository contains its own set of branches. So, to check out a remote branch, you have first to fetch the contents of the branch.

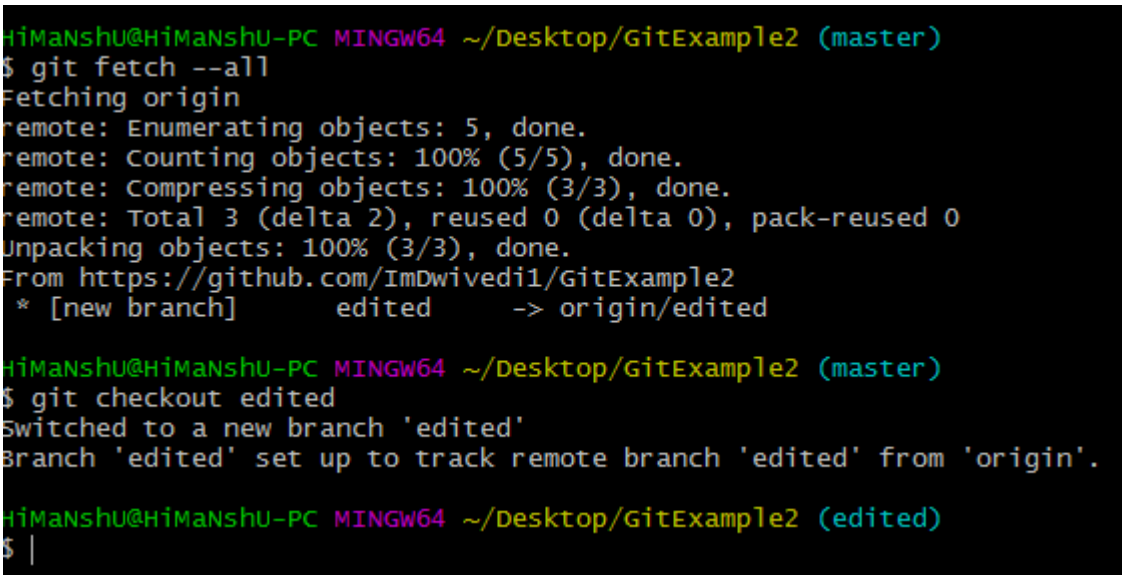
1. $ git fetch --all

In the latest versions of Git, you can check out the remote branch like a local branch.

Syntax:

1. $ git checkout <remotebranch>

Output:



In the above output, first, the fetch command is executed to fetch the remote data; after that, the checkout command is executed to check out a remote branch.

Edited is my remote branch. Here, we have switched to edited branch from master branch by git command line.

The earlier versions of Git require the creation of a new branch based on the remote. In earlier versions, below command is used to check out the remote branch.

1. $ git checkout <remotebranch> origin/<remotebranch>

Git Revert

In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a commit.

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Moreover, we can say that git revert records some new changes that are just opposite to previously made commits. To undo the changes, run the below command:

Syntax:

1. $ git revert

Git Revert Options:

Git revert allows some additional operations like editing, no editing, cleanup, and more. Let's understand these options briefly:

< commit>: The commit option is used to revert a commit. To revert a commit, we need the commit reference id. The git log command can access it.

1. $ git revert **<commit-ish>**

<--edit>: It is used to edit the commit message before reverting the commit. It is a default option in git revert command.

1. $ git revert -e **<commit-ish>**

-m parent-number /--mainline parent-number: it is used to revert the merging. Generally, we cannot revert a merge because we do not know which side of the merge should be considered as the mainline. We can specify the parent number and allows revert to reverse the change relative to the specified parent.

-n/--no edit: This option will not open a text editor. It will directly revert the last commit.

1. $ git revert -n **<commit-ish>**

--cleanup=<mode>: The cleanup option determines how to strip spaces and comments from the message.

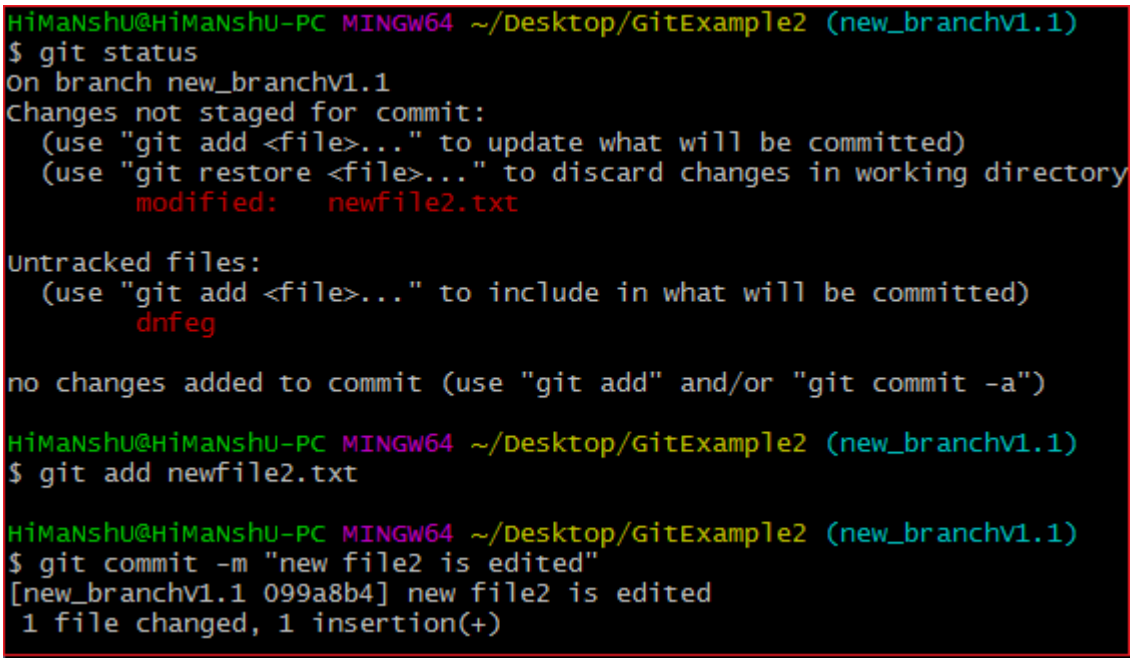
-n/--no-commit: Generally, the revert command commits by default. The no-commit option will not automatically commit. In addition, if this option is used, your index does not have to match the HEAD commit.

The no-commit option is beneficial for reverting more than one commits effect to your index in a row.

Let's understand how to revert the previous commits.

Git Revert to Previous Commit

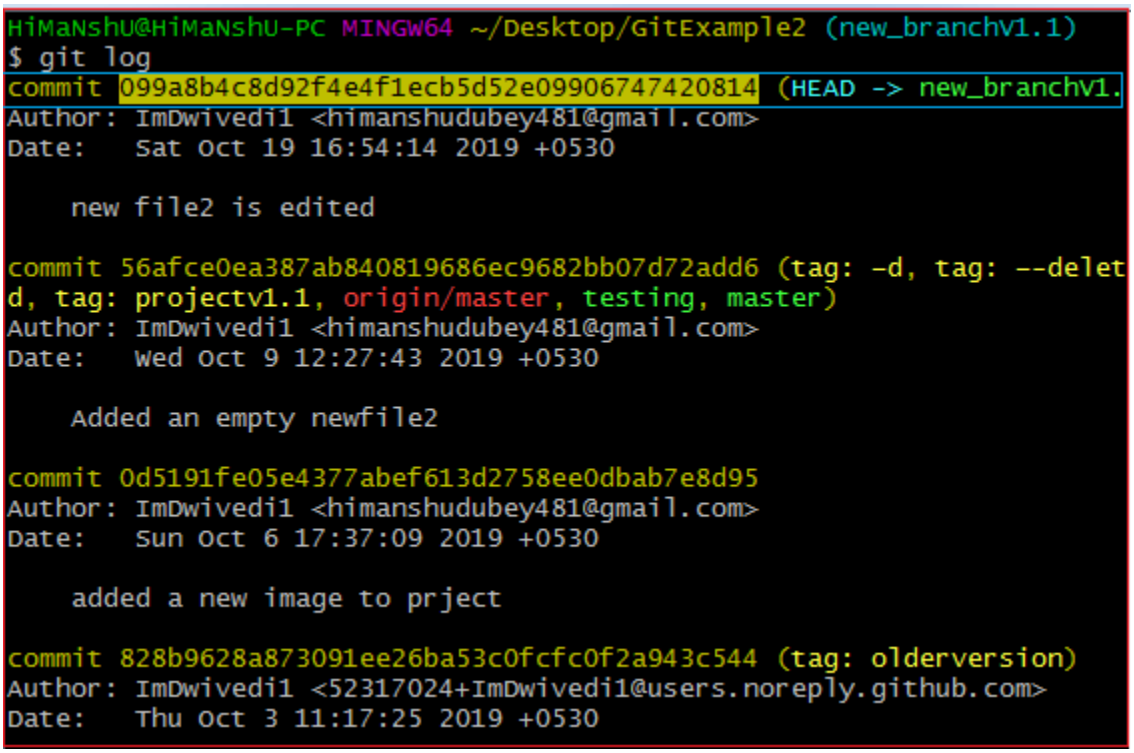
Suppose you have made a change to a file say **newfile2.txt** of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now, you want to undo the changes you can do so. Git allows you to correct your mistakes. Consider the below image:



As you can see from the above output that I have made changes in newfile2.txt. We can undo it by git revert command. To undo the changes, we will need the commit-ish. To check the commit-ish, run the below command:

1. $ git log

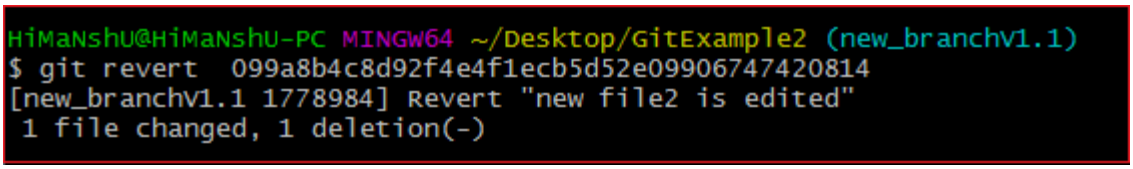
Consider the below output:



In the above output, I have copied the most recent commit-ish to revert. Now, I will perform the revert operation on this commit. It will operate as:

1. $ git revert 099a8b4c8d92f4e4f1ecb5d52e09906747420814

The above command will revert my last commit. Consider the below output:



As you can see from the above output, the changes made on the repository have been reverted.

Git Revert Merge

In Git, merging is also a commit that has at least two parents. It connects branches and code to create a complete project.

A merge in Git is a commit that has at least two parents. It brings together multiple lines of development. In a work-flow where features are developed in branches and then merged into a mainline, the merge commits would typically have two parents.

How to Revert a Merge

Usually, reverting a merge considered a complicated process. It can be complex if not done correctly. We are going to undo a merge operation with the help of git revert command. Although some other commands like git reset can do it. Let's understand how to revert a merge. Consider the below example.

I have made some changes to my file **design2.css** on the test and merge it with **test2**. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge test
Updating f1ddc7c..0a1a475
Fast-forward
 design2.css | 6 ++++++
 1 file changed, 6 insertions(+)
 create mode 100644 design2.css
```

To revert a merge, we have to get its reference number. To check commit history, run the below command:

1. $ git log

The above command will display the commit history. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git log
commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (HEAD -> test2, test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530

    new comit on test2 branch

commit 7fe5e7a8a733e55596db3f49d85e66245f88ddc0
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:33:18 2019 +0530

     new commit in master branch

commit dfb53648625baf81ab66f70944b2dc8c0fc9ef64
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Sep 27 18:18:53 2019 +0530

    commit2

commit 4fddabb36bf1690bec8a3b6605573ca9cffe00f4
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Sep 27 18:18:14 2019 +0530

    commit1

commit a3644e15993d30c7834b700077d4503c9afb6abd
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 14:40:35 2019 +0530
```

From the above output, copy your merging commit that you to want to revert and run the below command:

1. $ git revert **<commit** reference**>** -m 1

The above command will revert the merging operation. Here, -m 1 is used for the first parent as the mainline. Merge commit has multiple parents. The revert needs additional information to decide which parent of the merge shall be considered as the mainline. In such cases, the parameter -m is used. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (new_branchV1.1)
$ git revert  099a8b4c8d92f4e4f1ecb5d52e09906747420814
[new_branchV1.1 1778984] Revert "new file2 is edited"
 1 file changed, 1 deletion(-)
```

From the above output, we can see that the previous merge has been reverted.

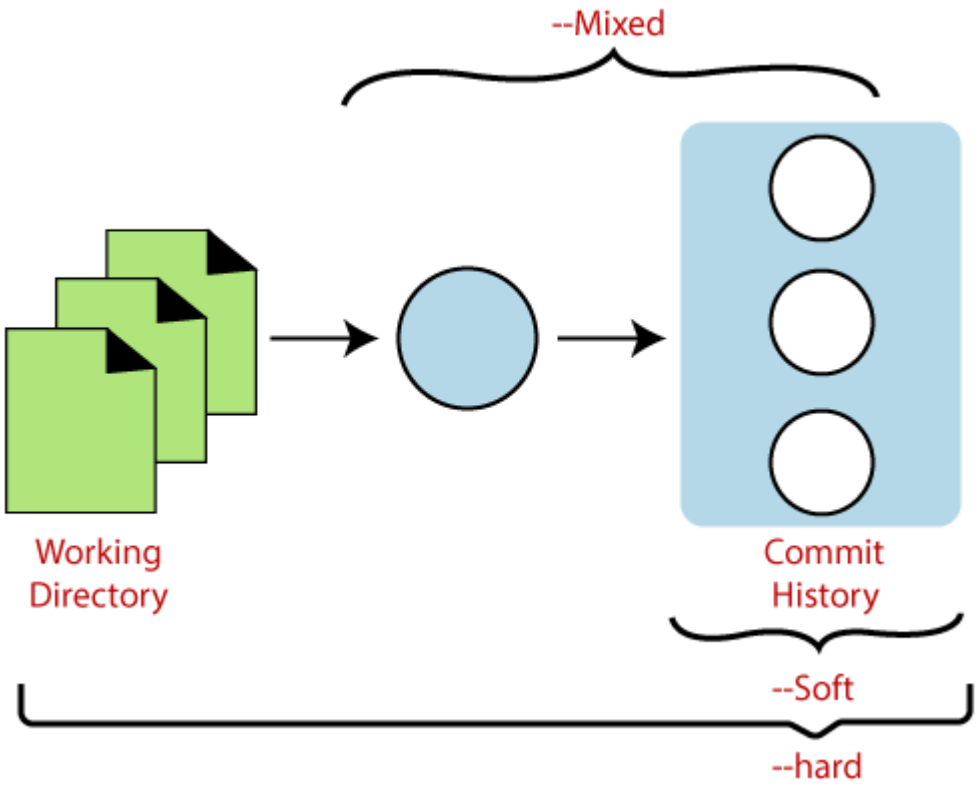
Git Reset

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**

If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state. It is a sophisticated and versatile tool for undoing changes. It acts as a **time machine for Git**. You can jump up and forth between the various commits. Each of these reset variations affects specific trees that git uses to handle your file in its content.

Additionally, git reset can operate on whole commits objects or at an individual file level. Each of these reset variations affects specific trees that git uses to handle your file and its contents.



Git uses an index (staging area), HEAD, and working directory for creating and reverting commits. If you have no idea about what is Head, trees, index, then do visit here Git Index and Git Head.

The working directory lets you change the file, and you can stage into the index. The staging area enables you to select what you want to put into your next commit. A commit object is a cryptographically hashed version of the content. It has some Metadata and points which are used to switch on the previous commits.

Let's understand the different uses of the git reset command.

Git Reset Hard

It will first move the Head and update the index with the contents of the commits. It is the most direct, unsafe, and frequently used option. The --hard option changes the Commit History, and ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory need to reset to match that of the specified commit. Any previously pending commits to the Staging Index and the Working Directory gets reset to match Commit Tree. It means any awaiting work will be lost.

Let's understand the --hard option with an example. Suppose I have added a new file to my existing repository. To add a new file to the repository, run the below command:

1. $ git add <file name>

To check the status of the repository, run the below command:

1. $ git status

To check the status of the Head and previous commits, run the below command:

1. $ git log

Consider the below image:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add newfile2.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newfile2.txt


HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git log
commit 34c25eb6f855476cb7999e3bec3eeb8e57727162 (HEAD -> test2)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Mon Oct 21 11:35:16 2019 +0530

    Revert "CSS file "

    This reverts commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3.

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.
```

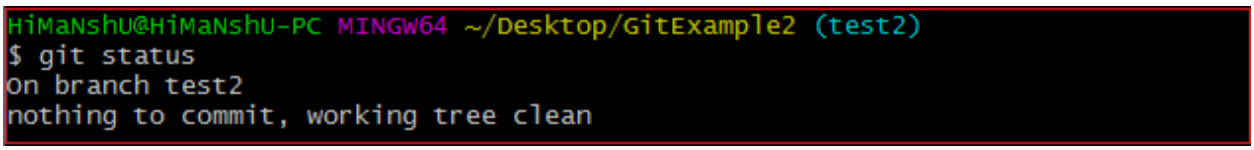
In the above output, I have added a file named **newfile2.txt**. I have checked the status of the repository. We can see that the current head position yet not changed because I have not committed the changes. Now, I am going to perform the **reset --hard** option. The git reset hard command will be performed as:

1. $ git reset --hard

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git reset --hard
HEAD is now at 34c25eb Revert "CSS file "
```

As you can see in the above output, the -hard option is operated on the available repository. This option will reset the changes and match the position of the Head before the last changes. It will remove the available changes from the staging area. Consider the below output:



The above output is displaying the status of the repository after the hard reset. We can see there is nothing to commit in my repository because all the changes removed by the reset hard option to match the status of the current Head with the previous one. So the file **newfile2.txt** has been removed from the repository.

There is a safer way to reset the changes with the help of git stash.

Generally, the reset hard mode performs below operations:

- It will move the HEAD pointer.
- It will update the staging Area with the content that the HEAD is pointing.
- It will update the working directory to match the Staging Area.

Git Reset Mixed

A mixed option is a default option of the git reset command. If we would not pass any argument, then the git reset command considered as **--mixed** as default option. A mixed option updates the ref pointers. The staging area also reset to the state of a specified commit. The undone changes transferred to the working directory. Let's understand it with an example.

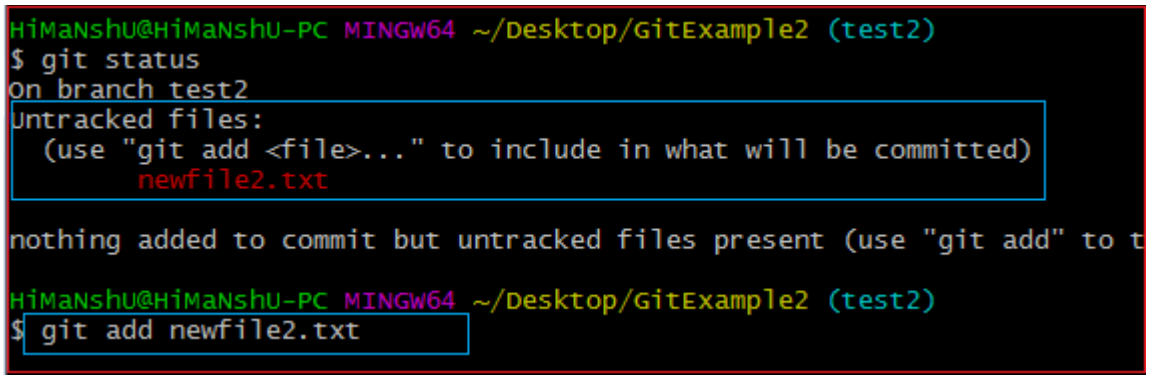
Let's create a new file say **newfile2.txt**. Check the status of the repository. To check the status of the repository, run the below command:

1. $ git status

It will display the untracked file from the staging area. Add it to the index. To add a file into stage index, run the git add command as:

1. $ git add <filename>

The above command will add the file to the staging index. Consider the below output:



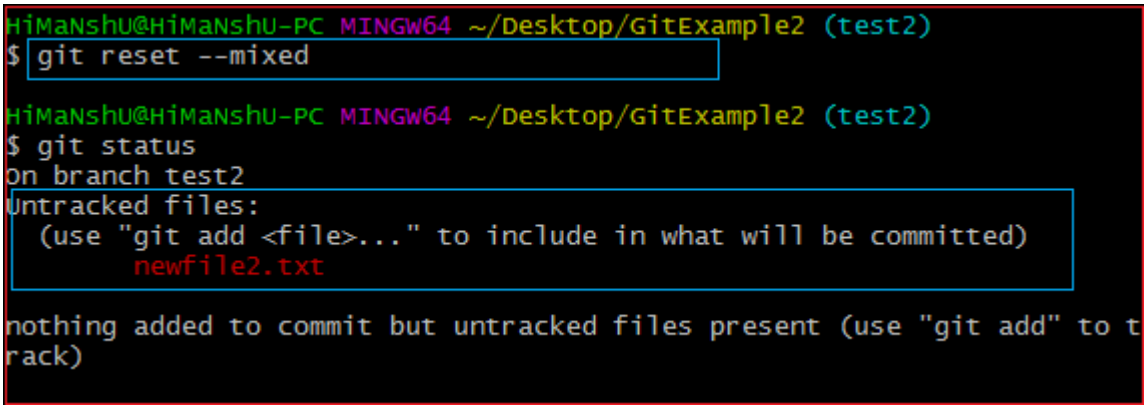
In the above output, I have added a **newfile2.txt** to my local repository. Now, we will perform the reset mixed command on this repository. It will operate as:

1. $ git reset --mixed

Or we can use only git reset command instead of this command.

1. $ git reset

The above command will reset the status of the Head, and it will not delete any data from the staging area to match the position of the Head. Consider the below output:



From the above output, we can see that we have reset the position of the Head by performing the git reset -mixed command. Also, we have checked the status of the repository. As we can see that the status of the repository has not been changed by this command. So it is clear that the mixed-mode does not clear any data from the staging area.

Generally, the reset mixed mode performs the below operations:

- It will move the HEAD pointer

- It will update the Staging Area with the content that the HEAD is pointing to.

It will not update the working directory as git hard mode does. It will only reset the index but not the working tree, then it generates the report of the files which have not been updated.

If -N is specified on the command line, then the statements will be considered as intent-to-add by Git.

Git Reset Head (Git Reset Soft)

The soft option does not touch the index file or working tree at all, but it resets the Head as all options do. When the soft mode runs, the refs pointers updated, and the resets stop there. It will act as git amend command. It is not an authoritative command. Sometimes developers considered it as a waste of time.

Generally, it is used to change the position of the Head. Let's understand how it will change the position of the Head. It will use as:

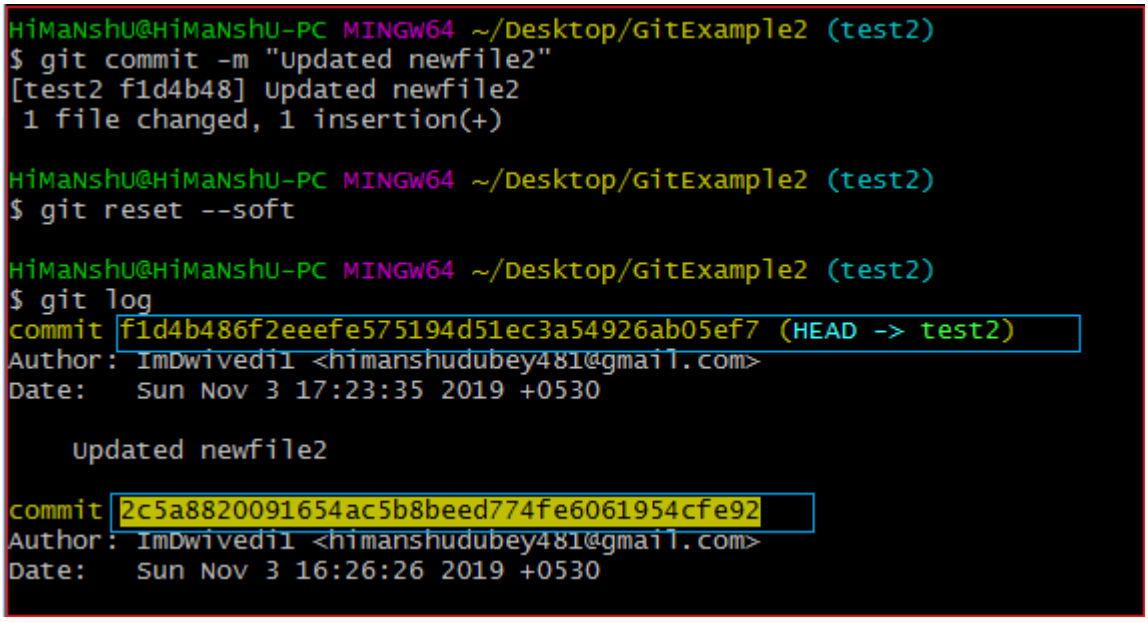
1. $ git reset--soft <commit-sha>

The above command will move the HEAD to the particular commit. Let's understand it with an example.

I have made changes in my file newfile2.txt and commit it. So, the current position of Head is shifted on the latest commit. To check the status of Head, run the below command:

1. $ git log

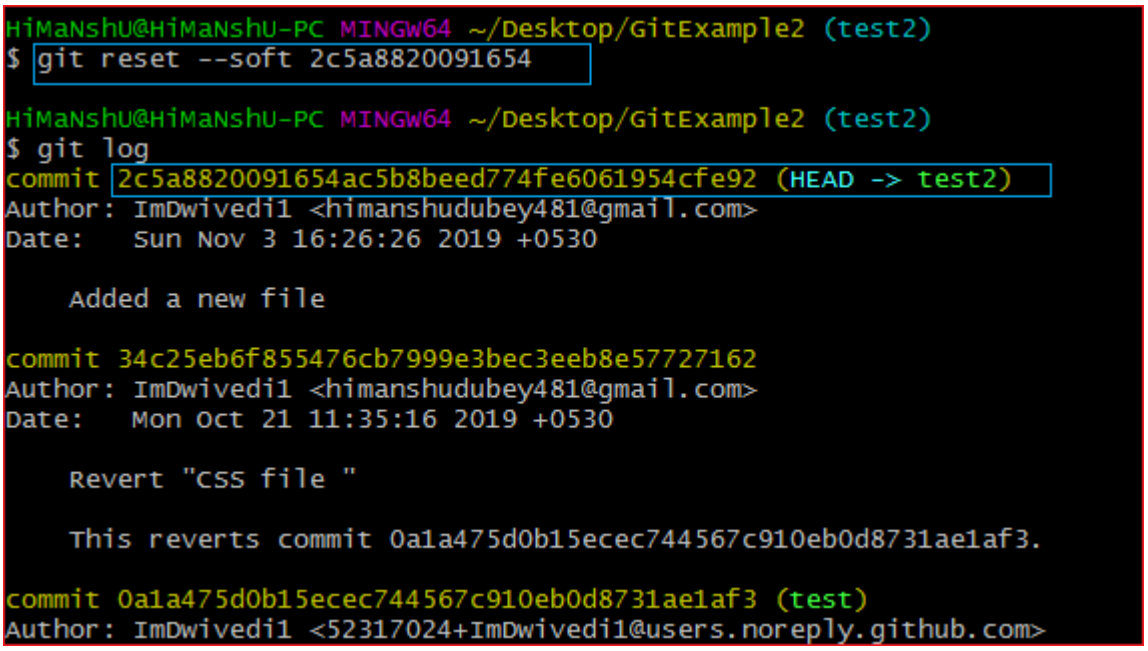
Consider the below output:



From the above output, you can see that the current position of the HEAD is on f1d4b486f2eeefe575194d51ec3a54926ab05ef7 commit. But, I want to switch it on my older commit 2c5a8820091654ac5b8beed774fe6061954cfe92. Since the commit-sha number is a unique number that is provided by sha algorithm. To switch the HEAD, run the below command:

1. $ git reset --soft 2c5a8820091654

The above command will shift my HEAD to a particular commit. Consider the below output:



As you can see from the above output, the HEAD has been shifted to a particular commit by git reset --soft mode.

Git Reset to Commit

Sometimes we need to reset a particular commit; Git allows us to do so. We can reset to a particular commit. To reset it, git reset command can be used with any option supported by reset command. It will take the default behavior of a particular command and reset the given commit. The syntax for resetting commit is given below:

1. $ git reset <option> <commit-sha>

These options can be

- --soft
- --mixed
- --Hard

Git Rm

In Git, the term rm stands for remove. It is used to remove individual files or a collection of files. The key function of git rm is to remove tracked files from the Git index.

Additionally, it can be used to remove files from both the working directory and staging index.

The files being removed must be ideal for the branch to remove. No updates to their contents can be staged in the index. Otherwise, the removing process can be complex, and sometimes it will not happen. But it can be done forcefully by **-f** option.

Let's understand it with an example.

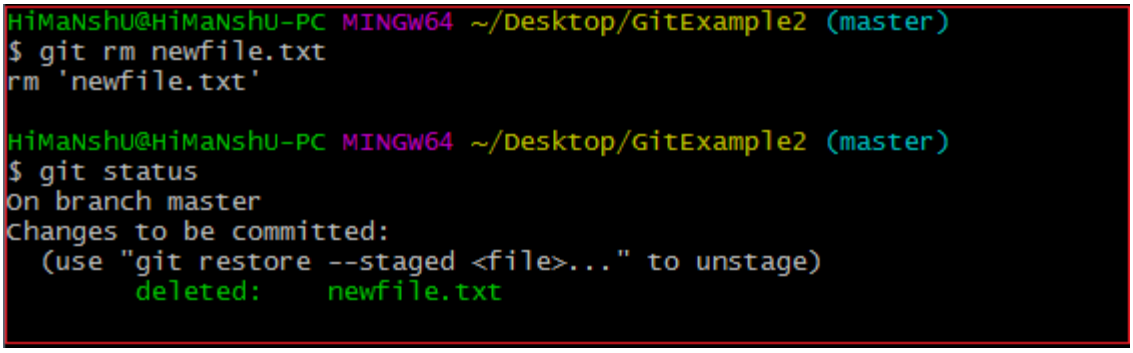
The git rm command

The git rm command is used to remove the files from the working tree and the index.

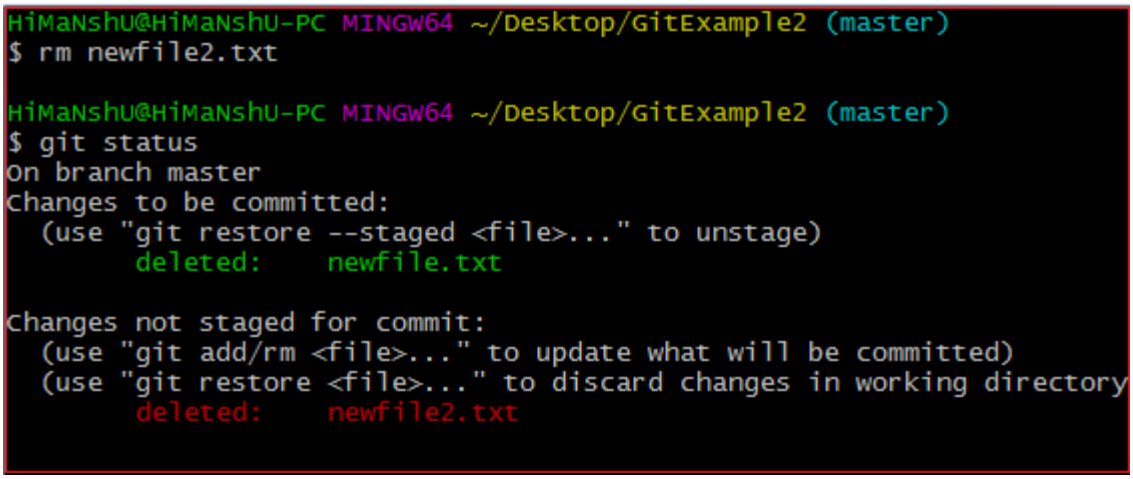
If we want to remove the file from our repository. Then it can be done by the git rm command. Let's take a file say newfile.txt to test the rm command. The git rm command will be operated as:

1. $ git rm <file Name>

The above command will remove the file from the Git and repository. The git rm command removes the file not only from the repository but also from the staging area. If we check the status of the repository, then it will show as deleted. Consider the below output:



In the above output, the file **newfile.txt** has been removed from the version control system. So the repository and the status are shown as deleted. If we use only **the rm command**, then it will not permanently delete the file from the Git. It can be tracked in the staging area. Consider the below output:



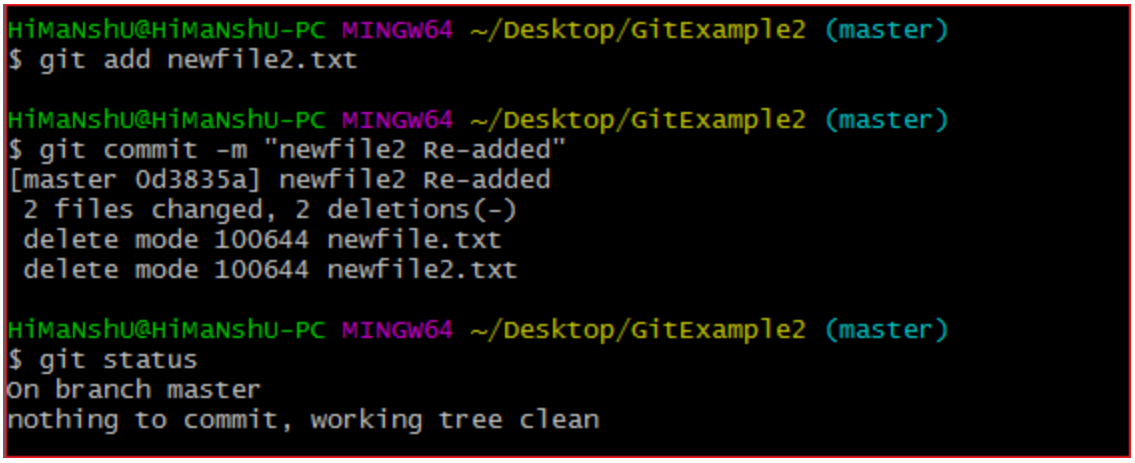
In the above output, the file newfile2.txt has been deleted. But when we check the status of the repository, we can track the file in the staging area. It means the newfile2 yet not deleted from the staging area, and it is also available in the repository. We can get it back on the version control system by committing it. To commit the file, first, add it to the index and then commit it. To add this file in the index, run the below command:

1. $ git add newfile2.txt

The above command will add the file to the index. To commit it, run the below command:

1. $ git commit -m "commit message."

It will commit the file and make it available to the version control system. Consider the below output:



In the above output, we are retrieving the file from the staging area to our directory. The newfile2.txt is re-added to our repository.

Git Rm Cached

Sometimes you want to remove files from the Git but keep the files in your local repository. In other words, you do not want to share your file on Git. Git allows you to do so. The cached option is used in this case. It specifies that the removal operation will only act on the staging index, not on the repository. The git rm command with cached option will be uses as:

1. $ git rm --cached <file name>

The above command will remove a file from the version control system. The deleted file will remain in the repository. Somehow this command will act as rm command. Let's understand it with an example.

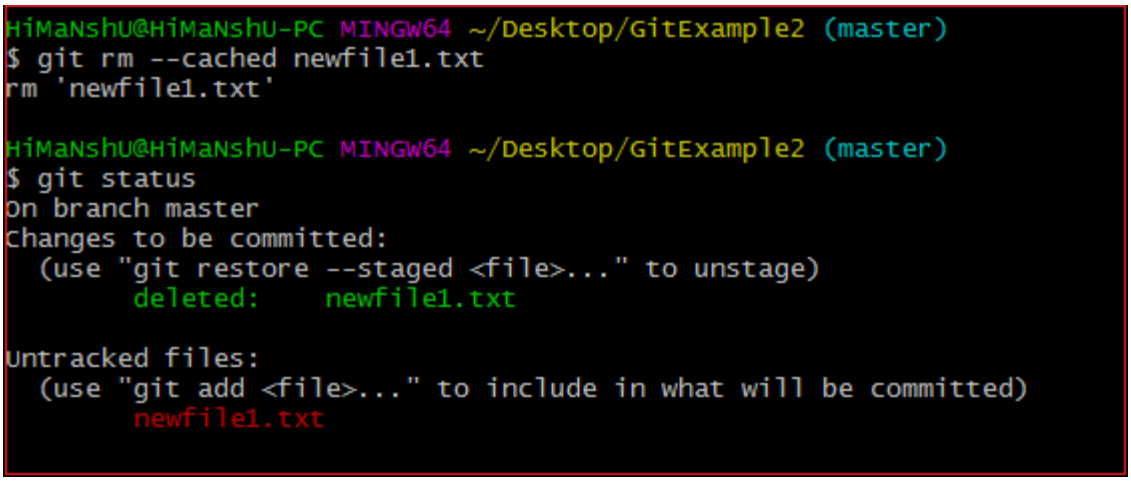
Suppose we want to remove a file from Git, take **newfile1.txt** for operation to remove this file, use the below command:

1. $ git rm --cached newfile1.txt

The above command will delete the file from the version control system, but still, it can be tracked in the repository. It also can be re-added on the version control system. To check the file status, use the status command as:

1. $ git status

Consider the below output:



As we can see from the above output, the newfile1.txt file is deleted from the version control system, but it can be tracked in the repository. This file is available on the version control system as an untracked file. We can track it by committing it.

Undo the Git Rm Command

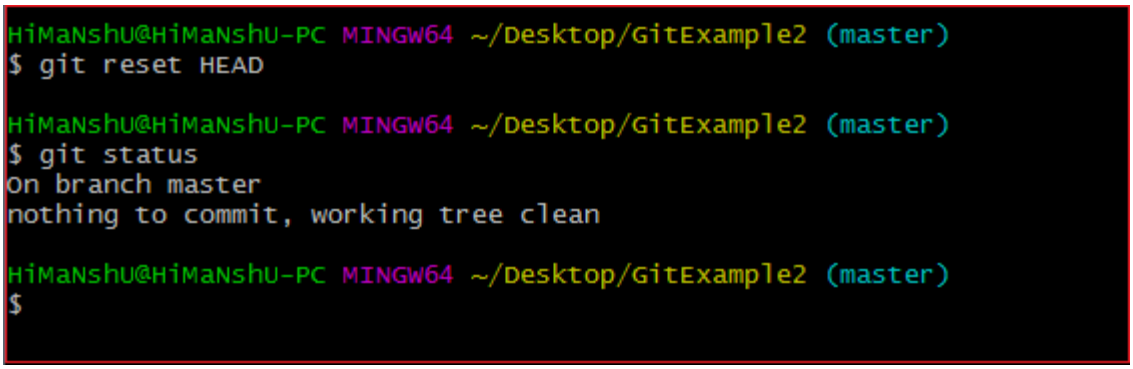
Execution of git rm command is not permanent; it can be reverted after execution. These changes cannot be persisted until a new commit is made on the repository. We can undo the git rm command. There are several ways to do so. The most usual and straight-forward way is git reset command. The git reset command will be used as follows:

1. $ git reset HEAD

Or we can also use:

1. $ git reset --hard

The above command will reset the position of the head. So that it will get the position of its just previous point. Consider the below output:



From the above output, we can see that the file has been successfully reset to its previous position.

There is another way to undo the git rm command. We can also do it by git checkout command. A checkout has the same effect and restores the latest version of a file from HEAD. It will be used as follows:

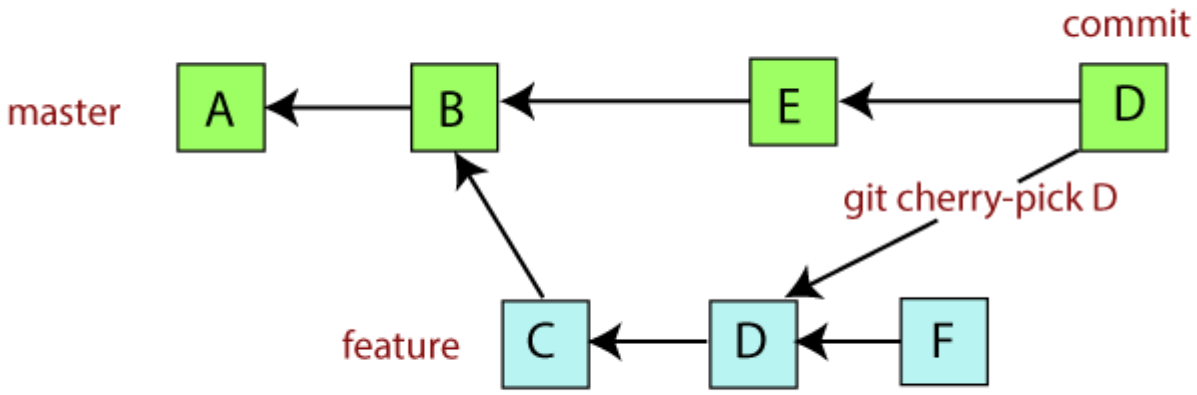
1. $ git checkout.

Limits of Git Rm command

The git rm is operated only on the current branch. The removing process is only applied to the working directory and staging index trees. It is not persisted in the repository history until a new commit is created.

Git Cherry-pick

Cherry-picking in Git stands for applying 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 apply it on another branch.



The main motive of a cherry-pick is to apply the changes introduced by some existing commit. A cherry-pick looks at a previous commit in the repository history and update the changes that were part of that last commit to the current working tree. The definition is straight forward, yet it is more complicated when someone tries to cherry-pick a commit, or even cherry-pick from another branch.

Cherry-pick is a useful tool, but always it is not a good option. It can cause duplicate commits and some other scenarios where other merges are preferred instead of cherry-picking. It is a useful tool for a few situations. It is in contrast with different ways such as **merge** and **rebase** command. Merge and rebase can usually apply many commits in another branch.

Why Cherry-Pick

Suppose you are working with a team of developers on a medium to large-sized project. Some changes proposed by another team member and you want to apply some of them to your main project, not all. Since managing the changes between several Git branches can become a complex task, and you don't want to merge a whole branch into another branch. You only need to pick one or two specific commits. To pick some changes into your main project branch from other branches is called cherry-picking.

Some scenarios in which you can cherry-pick:

Scenerio1: Accidently make a commit in a wrong branch.

Git cherry-pick is helpful to apply the changes that are accidentally made in the wrong branch. Suppose I want to make a commit in the master branch, but by mistake, we made it in any other branch. See the below commit.

```
omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$ git add file1.txt

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$ git commit -m " new changes proposes for master branch"
[newbranch 4f7cc74]  new changes proposes for master branch
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$
```

In the above example, I want to make a commit for the master branch, but accidentally I made it in the new branch. To make all the changes of the new branch into the master branch, we will use the git pull, but for this particular commit, we will use git cherry-pick command. See the below output:

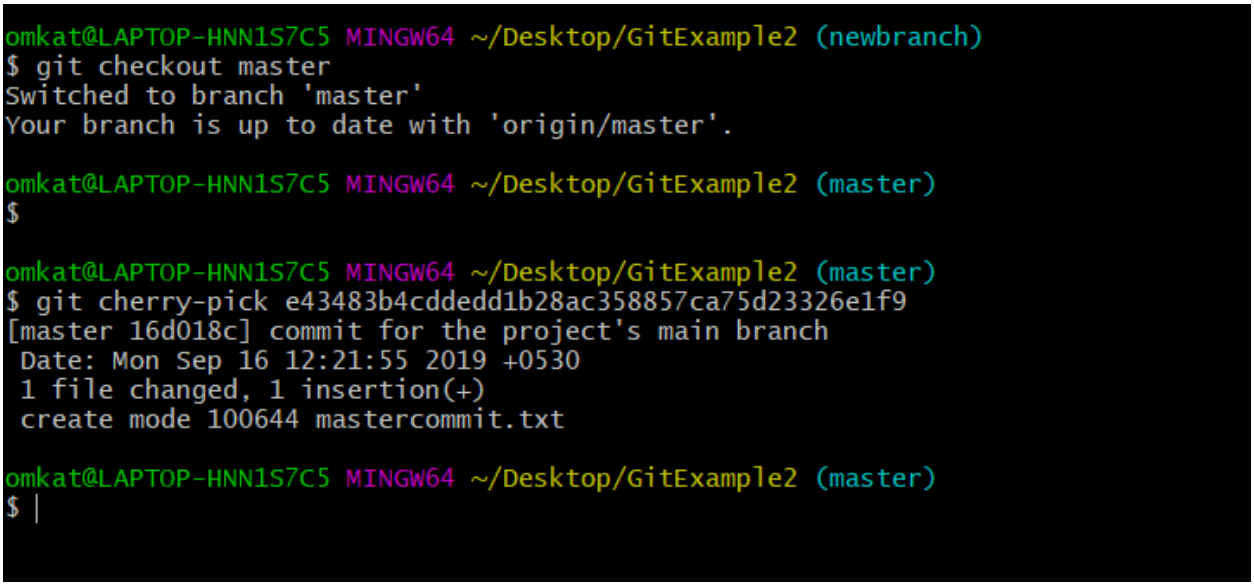
```
omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (newbranch)
$ git log
commit e43483b4cddedd1b28ac358857ca75d23326e1f9 (HEAD -> newbranch)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Mon Sep 16 12:21:55 2019 +0530

    commit for the project's main branch

commit 4f7cc7458f79f8c21fa3563dbecee03381dc9645
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Mon Sep 16 12:02:23 2019 +0530

     new changes proposes for master branch

commit 4a6693a71151323623c04dd75cb0d44c1c4dbadf (origin/master, origin/HEAD)
Merge: 30193f3 78c5fbd
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Mon Sep 9 15:24:13 2019 +0530

    Merge pull request #1 from ImDwivedi1/branch2

    Create merge the branch
```

In the given output, I have used the git log command to check the commit history. Copy the particular commit id that you want to make on the master branch. Now switch to master branch and cherry-pick it there. See the below output:

Syntax:

1. $ git cherry-pick **<commit** id**>**

Output:



From the given output, you can see that I have pasted the commit id with git cherry-pick command and made that commit into my master branch. You can check it by git log command.

Scenario2: Made the changes proposes by another team member.

Another use of cherry-picking is to make the changes proposed by another team member. Suppose one of my team members made any changes in the main project and suggests it for the main project. You can cheery-pick it after review.

Usage of cherry-pick

- It is a handy tool for **team collaboration**.
- It is necessary in case of **bug fixing** because bugs are fixed and tested in the development branch with their commits.
- It is mostly used in **undoing changes and restoring lost commits**.
- You can **avoid useless conflicts** by using git cherry-pick instead of other options.

- It is a useful tool when a full branch merge is not possible due to incompatible versions in the various branches.
- The git cherry-pick is used to access the changes introduced to a sub-branch, without changing the branch.

Git log

The advantage of a version control system is that it records changes. These records allow us to retrieve the data like commits, figuring out bugs, updates. But, all of this history will be useless if we cannot navigate it. At this point, we need the git log command.

Git log is a utility tool to review and read a history of everything that happens to a repository. Multiple options can be used with a git log to make history more specific.

Generally, the git log is a record of commits. A git log contains the following data:

- **A commit hash**, which is a 40 character checksum data generated by SHA (Secure Hash Algorithm) algorithm. It is a unique number.
- **Commit Author metadata**: The information of authors such as author name and email.
- **Commit Date metadata**: It's a date timestamp for the time of the commit.
- **Commit title/message**: It is the overview of the commit given in the commit message.

How to Exit the git log Command?

There may be a situation that occurs, you run the git log command, and you stuck there. You want to type or back to bash, but you can't. When you click the **Enter** key, it will navigate you to the older command until the end flag.

The solution to this problem is to **press** the **q (Q for quit)**. It will quit you from the situation and back you to the command line. Now, you can perform any of the commands.

Basic Git log

Git log command is one of the most usual commands of git. It is the most useful command for Git. Every time you need to check the history, you have to use the git log command. The basic git log command will display the most recent commits and the status of the head. It will use as:

1. $ git log

The above command will display the last commits. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530
```

The above command is listing all the recent commits. Each commit contains some unique sha-id, which is generated by the SHA algorithm. It also includes the date, time, author, and some additional details.

We can perform some action like scrolling, jumping, move, and quit on the command line. To scroll on the command line press k for moving up, j for moving down, the spacebar for scrolling down by a full page to scroll up by a page and q to quit from the command line.

<

Git Log Oneline

The oneline option is used to display the output as one commit per line. It also shows the output in brief like the first seven characters of the commit SHA and the commit message.

It will be used as follows:

1. $ git log --oneline

So, usually we can say that the --oneline flag causes git log to display:

- one commit per line
- the first seven characters of the SHA
- the commit message

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --oneline
0d3835a (HEAD -> master) newfile2 Re-added
56afce0 (tag: -d, tag: --delete, tag: --d, tag: projectv1.1, origin/master, test
ing) Added an empty newfile2
0d5191f added a new image to prject
828b962 (tag: olderversion) Update design2.css
0a1a475 (test) css file
f1ddc7c new comit on test2 branch
7fe5e7a  new commit in master branch
dfb5364 commit2
4fddabb commit1
a3644e1 edit newfile1
d2bb07d edited newfile1.txt
2852e02 newfile1 added
4a6693a Merge pull request #1 from ImDwivedi1/branch2
30193f3 new files via upload
78c5fbd Create merge the branch
1d2bc03 Initial commit
```

As we can see more precisely from the above output, every commit is given only in one line with a seven-digit sha number and commit message.

Git Log Stat

The log command displays the files that have been modified. It also shows the number of lines and a summary line of the total records that have been updated.

Generally, we can say that the stat option is used to display

- the modified files,
- The number of lines that have been added or removed
- A summary line of the total number of records changed
- The lines that have been added or removed.

It will be used as follows:

1. $ git log --stat

The above command will display the files that have been modified. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --stat
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

 newfile.txt  | 2 --
 newfile2.txt | 0
 2 files changed, 2 deletions(-)

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --d, ta
g: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

 newfile2.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

 abc.jpg | Bin 0 -> 777835 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css
```

From the above output, we can see that all listed commits are modifications in the repository.

Git log P or Patch

The git log patch command displays the files that have been modified. It also shows the location of the added, removed, and updated lines.

It will be used as:

1. $ git log --patch

Or

1. $ git log -p

Generally, we can say that the --patch flag is used to display:

- Modified files
- The location of the lines that you added or removed
- Specific changes that have been made.

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --patch
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

diff --git a/newfile.txt b/newfile.txt
deleted file mode 100644
index d411be5..0000000
--- a/newfile.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-new file to check git Head
-NEW COMMIT IN MASTER BRANCH.
diff --git a/newfile2.txt b/newfile2.txt
deleted file mode 100644
index e69de29..0000000

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530
:
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
```

The above output is displaying the modified files with the location of lines that have been added or removed.

Git Log Graph

Git log command allows viewing your git log as a graph. To list the commits in the form of a graph, run the git log command with --graph option. It will run as follows:

1. $ git log --graph

To make the output more specific, you can combine this command with --oneline option. It will operate as follows:

1. $ git log --graph --oneline

Filtering the Commit History

We can filter the output according to our needs. It's a unique feature of Git. We can apply many filters like amount, date, author, and more on output. Each filter has its specifications. They can be used for implementing some navigation operations on output.

Let's understand each of these filters in detail.

By Amount:

We can limit the number of output commit by using git log command. It is the most specific command. This command will remove the complexity if you are interested in fewer commits.

To limit the git log's output, including the -<n> option. If we want only the last three commit, then we can pass the argument -3 in the git log command. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log -3
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject
```

As we can see from the above output, we can limit the output of git log.

By Date and Time:

We can filter the output by date and time. We have to pass **--after** or **-before** argument to specify the date. These both argument accept a variety of date formats. It will run as follows:

1. $ git log --after="yy-mm-dd"

The above command will display all the commits made after the given date. Consider the below output:

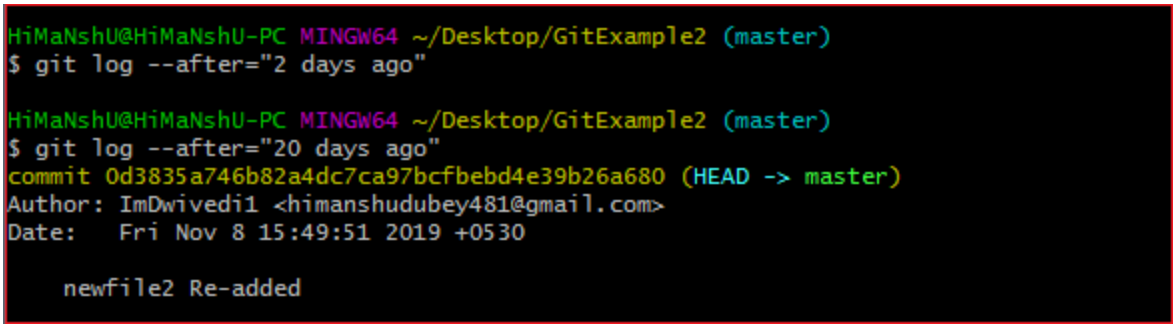
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --after="2019-11-01"
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added
```

The above command is listing all the commits after "2019-11-01".

We can also pass the applicable reference statement like "yesterday," "1 week ago", "21 days ago," and more. It will run as:

1. git log --after="21 days ago"

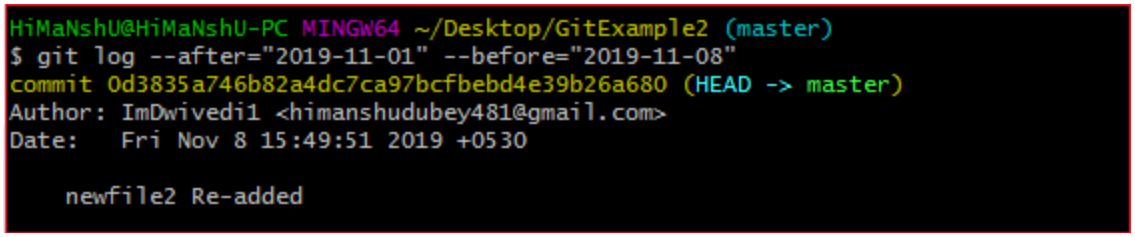
The above command will display the commits which have been made 21 days ago. Consider the below output:



We can also track the commits between two dates. To track the commits that were created between two dates, pass a statement reference **--before** and **--after** the date. Suppose, we want to track the commits between "2019-11-01" and "2019-11-08". We will run the command as follows:

1. $ git log --after="2019-11-01" --before="2019-11-08 "

The above command will display the commits made between the dates. Consider the below output:



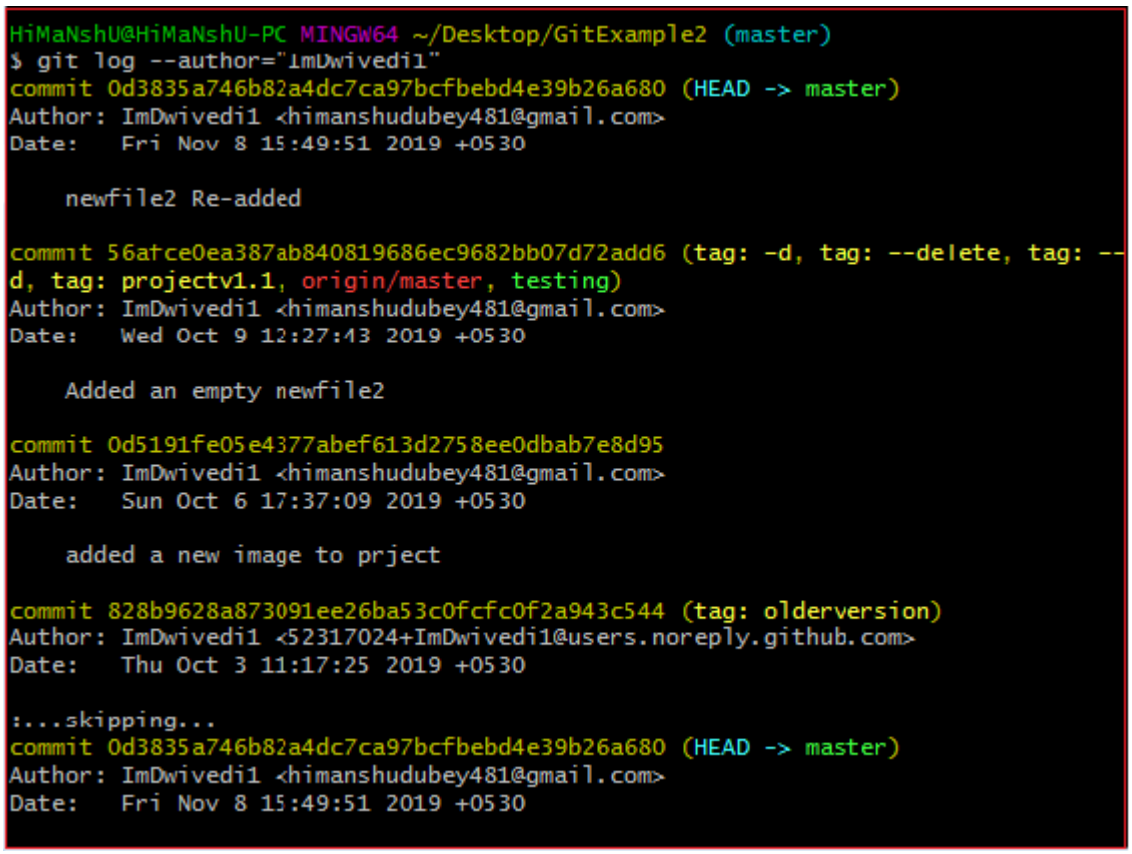
The above output is displaying the commits between the given period. We can use --since and --until instead of --after and --before. Because they are synonyms, respectively.

By Author:

We can filter the commits by a particular user. Suppose, we want to list the commits only made by a particular team member. We can use -author flag to filter the commits by author name. This command takes a regular expression and returns the list of commits made by authors that match that pattern. You can use the exact name instead of the pattern. This command will run as follows:

1. $ git log --author="Author name"

The above command will display all the commits made by the given author. Consider the below output:



From the above output, we can see that all the commits by the author **ImDwivedi1** are listed.

We can use a string instead of a regular expression or exact name. Consider the below statement:

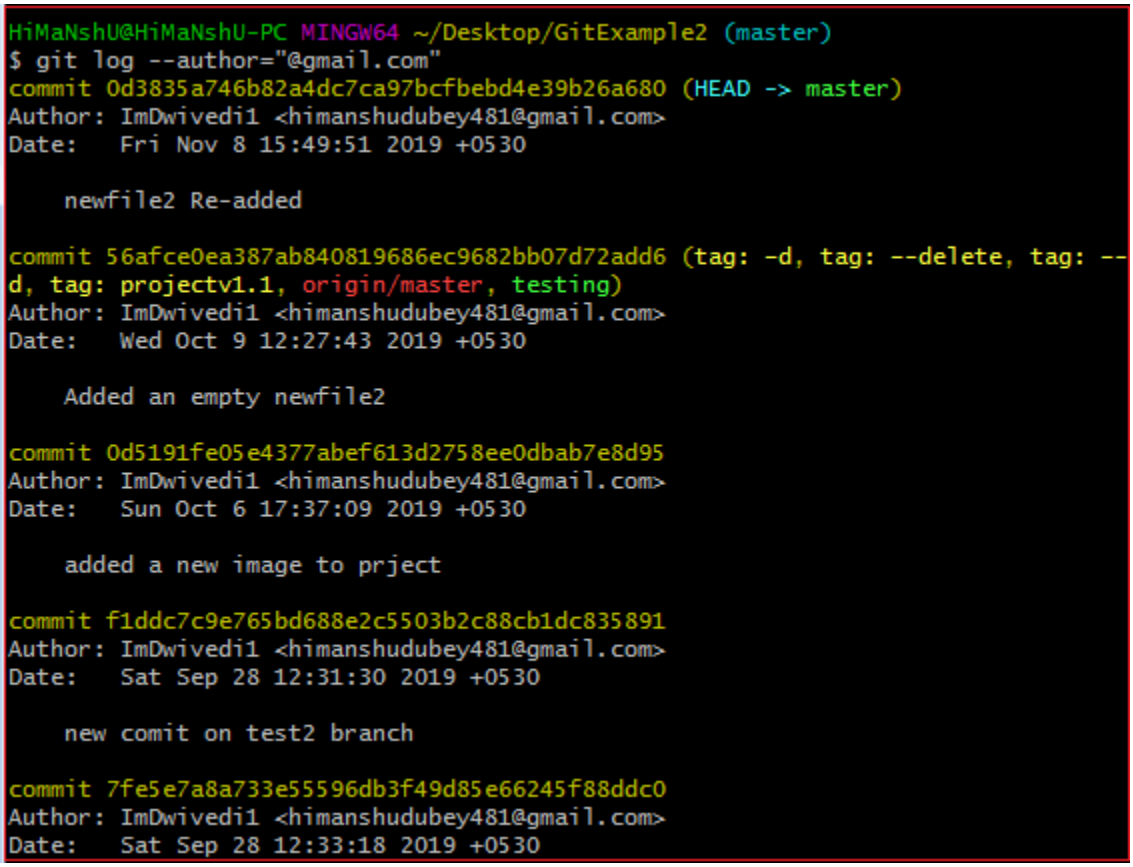
1. $ git log --author="Stephen"

The above statement will display all commits whose author includes the name, Stephen. The author's name doesn't need to be an exact match; it just has the specified phrase.

As we know, the author's email is also involved with the author's name, so that we can use the author's email as the pattern or exact search. Suppose, we want to track the commits by the authors whose email service is google. To do so, we can use wild cards as "@gmail.com." Consider the below statement:

1. $ git log -author="@gmail.com"

The above command will display the commits by authors as given in the pattern. Consider the below output:



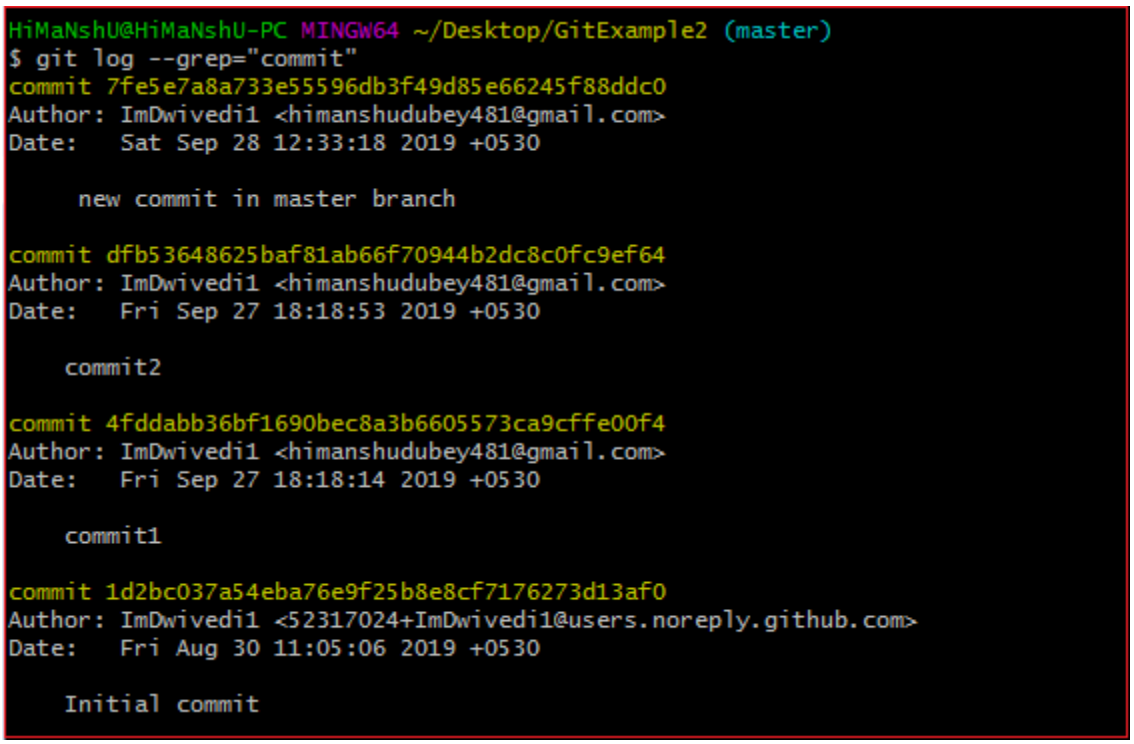
By Commit message:

To filter the commits by the commit message. We can use the grep option, and it will work as the author option.

It will run as follows:

1. $ git log --grep=" Commit message."

We can use the short form of commit message instead of a complete message. Consider the below output.



The above output is displaying all the commits that contain the word commit in its commit message.

There are many other filtering options available like we can filter by file name, content, and more.

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.

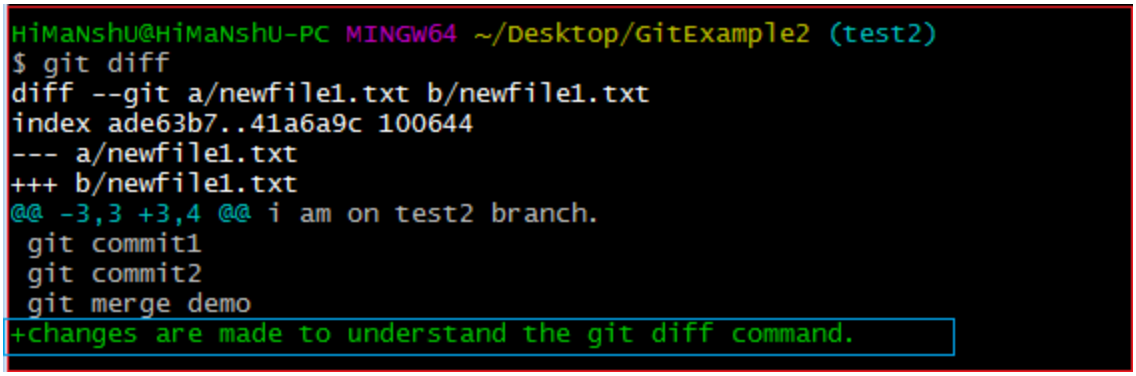
However, we can also track the changes with the help of git log command with option -p. The git log command will also work as a git diff command.

Let's understand different scenarios where we can utilize the git diff command.

Scenerio1: Track the changes that have not been staged.

The usual use of git diff command that we can track the changes that have not been staged.

Suppose we have edited the newfile1.txt file. Now, we want to track what changes are not staged yet. Then we can do so from the git diff command. Consider the below output:



From the above output, we can see that the changes made on newfile1.txt are displayed by git diff command. As we have edited it as "changes are made to understand the git diff command." So, the output is displaying the changes with its content. The highlighted section of the above output is the changes in the updated file. Now, we can decide whether we want to stage this file like this or not by previewing the changes.

Scenerio2: Track the changes that have staged but not committed:

The git diff command allows us to track the changes that are staged but not committed. We can track the changes in the staging area. To check the already staged changes, use the --staged option along with git diff command.

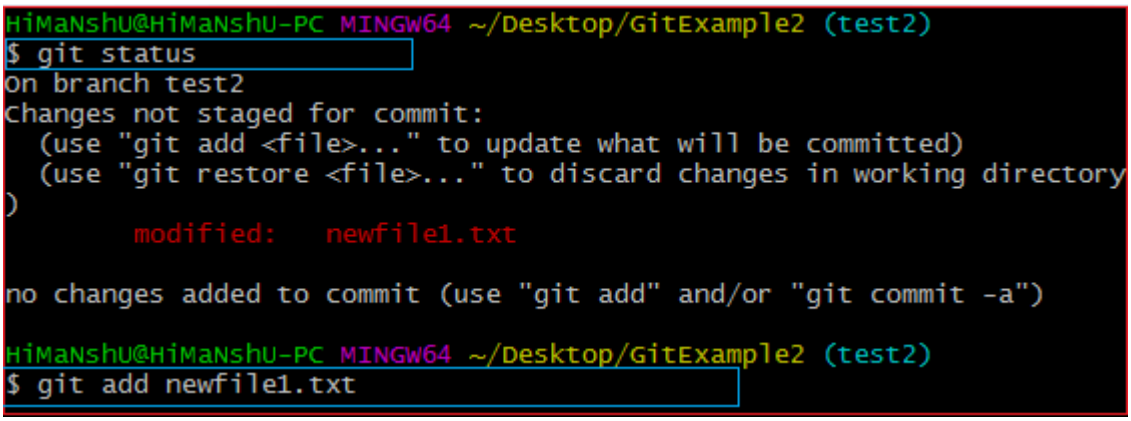
To check the untracked file, run the git status command as:

1. $ git status

The above command will display the untracked file from the repository. Now, we will add it to the staging area. To add the file in the staging area, run the git add command as:

1. $ git add < file name>

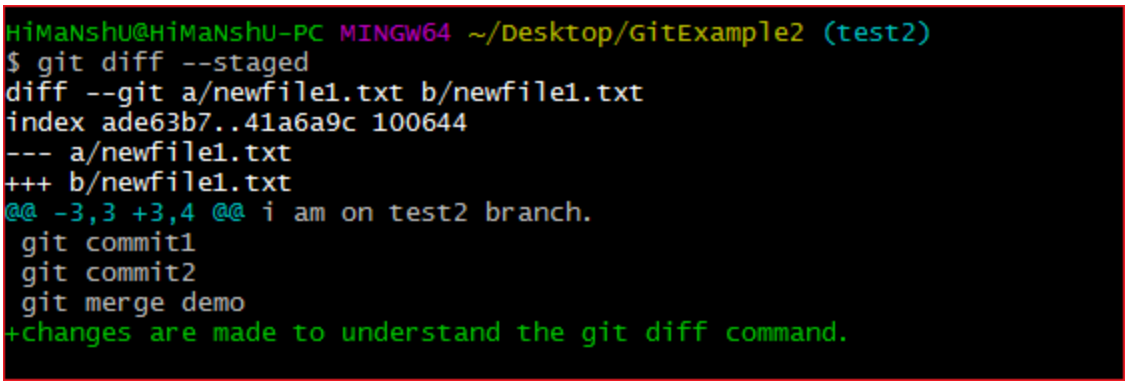
The above command will add the file in the staging area. Consider the below output:



Now, the file is added to the staging area, but it is not committed yet. So, we can track the changes in the staging area also. To check the staged changes, run the git diff command along with **--staged** option. It will be used as:

1. $ git diff --staged

The above command will display the changes of already staged files. Consider the below output:



The given output is displaying the changes of newfile1.txt, which is already staged.

Scenerio3: Track the changes after committing a file:

Git, let us track the changes after committing a file. Suppose we have committed a file for the repository and made some additional changes after the commit. So we can track the file on this stage also.

In the below output, we have committed the changes that we made on our newfile1.txt. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m "newfile1 is updated"
[test2 e553fc0] newfile1 is updated
 1 file changed, 1 insertion(+)
```

Now, we have changed the newfile.txt file again as "Changes are made after committing the file." To track the changes of this file, run the git diff command with **HEAD** argument. It will run as follows:

1. $ git diff HEAD

The above command will display the changes in the terminal. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff HEAD
diff --git a/newfile1.txt b/newfile1.txt
index 41a6a9c..e14624d 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -4,3 +4,4 @@ git commit1
 git commit2
 git merge demo
 changes are made to understand the git diff command.
+changes are made after committing the file.
```

The above command is displaying the updates of the file newfile1.txt on the highlighted section.

Scenario4: Track the changes between two commits:

We can track the changes between two different commits. Git allows us to track changes between two commits, whether it is the latest commit or the old commit. But the required thing for this is that we must have a list of commits so that we can compare. The usual command to list the commits in the git log command. To display the recent commits, we can run the command as:

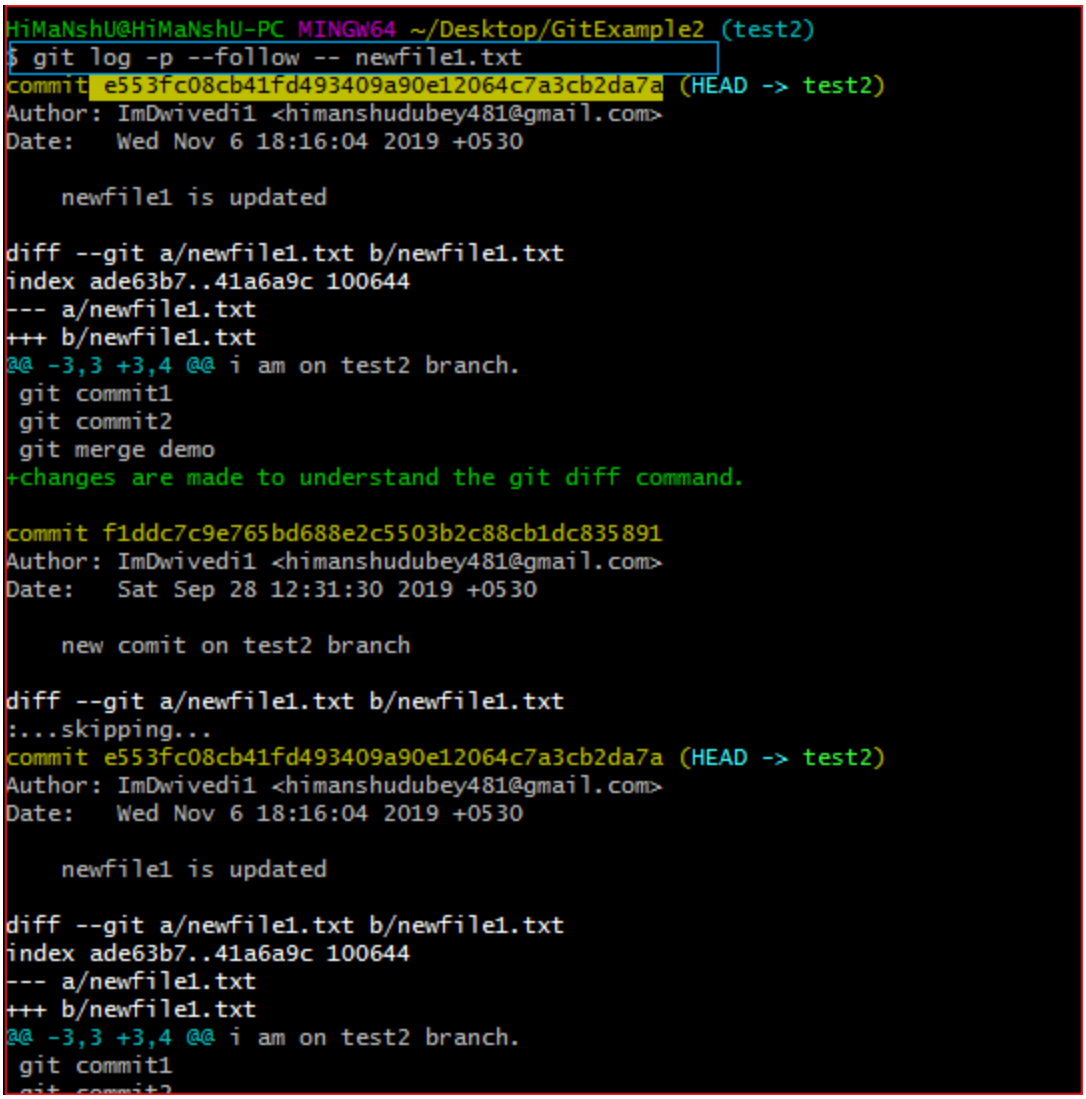
1. $ git log

The above command will list the recent commits.

Suppose, we want to track changes of a specified from an earlier commit. To do so, we must need the commits of that specified file. To display the commits of any specified, run the git log command as:

1. $ git log -p --follow -- filename

The above command will display all the commits of a specified file. Consider the below output:



The above output is displaying all the commits of newfile1.txt. Suppose we want to track the changes between commits **e553fc08cb** and **f1ddc7c9e7**. The git diff command lets track the changes between two commits. It will be commanded as:

1. $ git diff <commit1-sha> <commit2-sha>

The above command will display the changes between two commits. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff e553fc08cb f1ddc7c9e7
diff --git a/.gitignore b/.gitignore
deleted file mode 100644
index 8527c9c..0000000
--- a/.gitignore
+++ /dev/null
@@ -1,2 +0,0 @@
-*.txt
-/newfolder/*
\ No newline at end of file
diff --git a/newfile1.txt b/newfile1.txt
index 41a6a9c..ade63b7 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -3,4 +3,3 @@ i am on test2 branch.
 git commit1
 git commit2
 git merge demo
-changes are made to understand the git diff command.
diff --git a/newfile2.txt b/newfile2.txt
deleted file mode 100644
index 1b46874..0000000
--- a/newfile2.txt
+++ /dev/null
@@ -1 +0,0 @@
-it sdgsfuwytruhdsmzbxJDBWMFBYUG
\ No newline at end of file
```

The above output is displaying all the changes made on **newfile1.txt** from commit **e553fc08cb** (most recent) to commit **f1ddc7c9e7** (previous).

Git Diff Branches

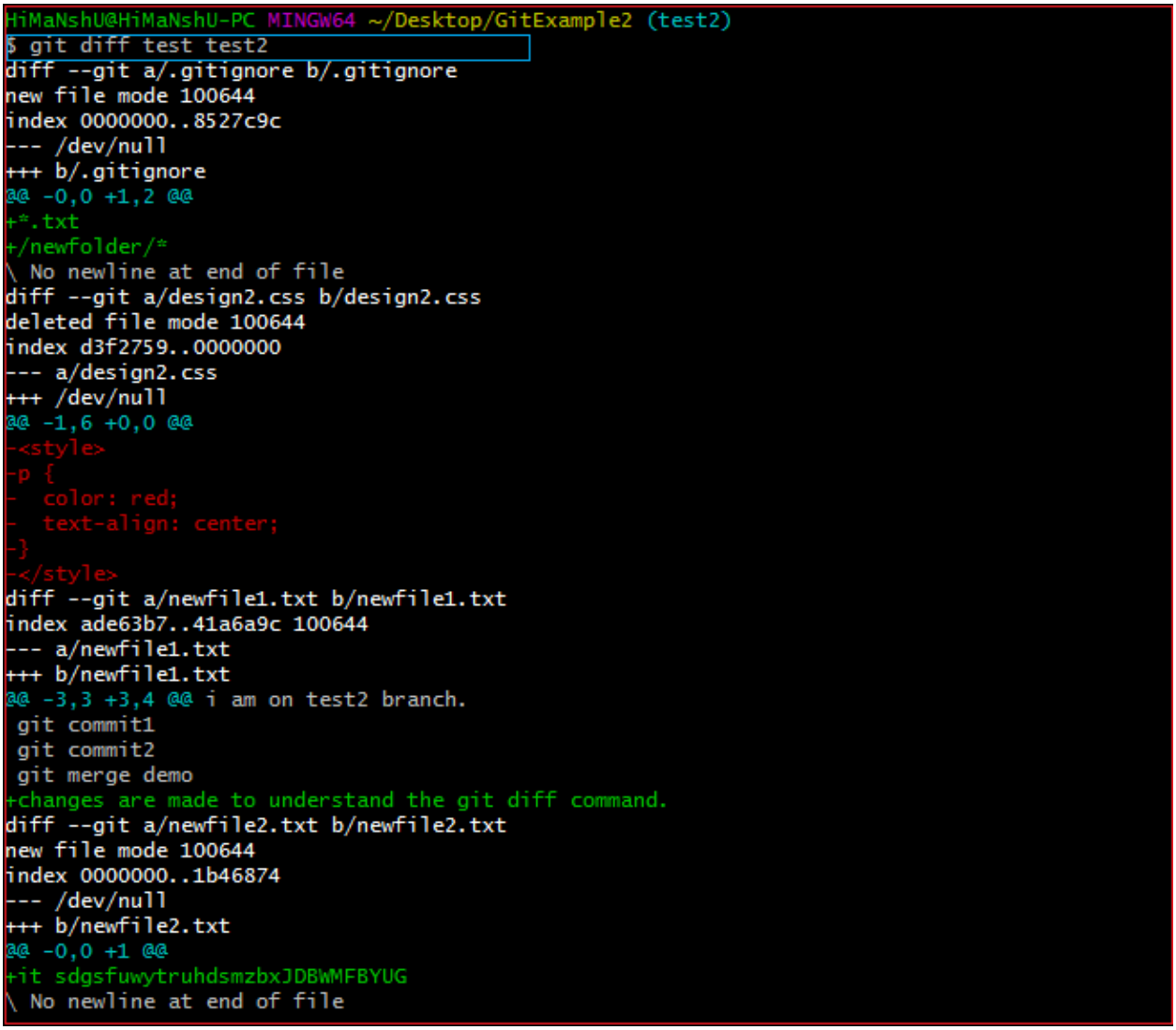
Git allows comparing the branches. If you are a master in branching, then you can understand the importance of analyzing the branches before merging. Many conflicts can arise if you merge the branch without comparing it. So to avoid these conflicts, Git allows many handy commands to preview, compare, and edit the changes.

We can track changes of the branch with the git status command, but few more commands can explain it in detail. The git diff command is a widely used tool to track the changes.

The git diff command allows us to compare different versions of branches and repository. To get the difference between branches, run the git diff command as follows:

1. $ git diff <branch 1> < branch 2>

The above command will display the differences between branch 1 and branch 2. So that you can decide whether you want to merge the branch or not. Consider the below output:



The above output is displaying the differences between my repository branches **test** and **test2**. The git diff command is giving a preview of both branches. So, it will be helpful to perform any operation on branches.

Git Status

The git status command is used to display the state of the repository and staging area. It allows us to see the tracked, untracked files and changes. This command will not show any commit records or information.

Mostly, it is used to display the state between **Git Add** and **Git commit** command. We can check whether the changes and files are tracked or not.

Let's understand the different states of status command.

Status when Working Tree is cleaned

Before starting with git status command, let's see how the git status looks like when there are no changes made. To check the status, open the git bash, and run the status command on your desired directory. It will run as follows:

1. $ git status

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Since there is nothing to track or untrack in the working tree, so the output is showing as the **working tree is clean**.

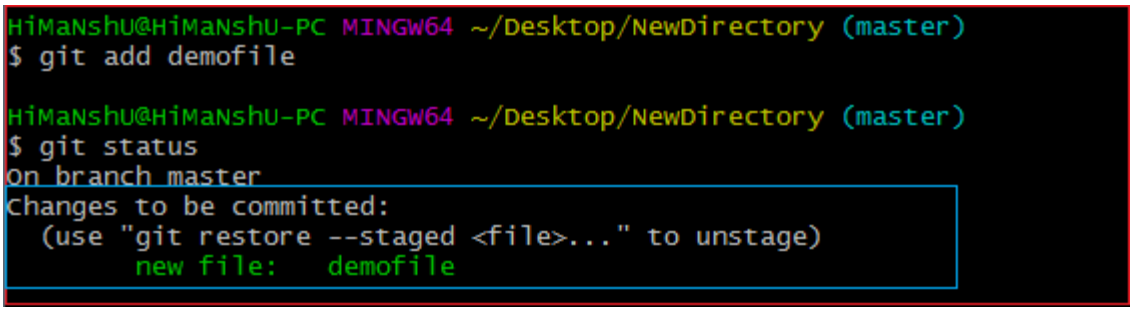
Status when a new file is created

When we create a file in the repository, the state of the repository changes. Let's create a file using the **touch** command. Now, check the status using the status command. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ touch demofile

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        demofile

nothing added to commit but untracked files present (use "git add" to
track)
```

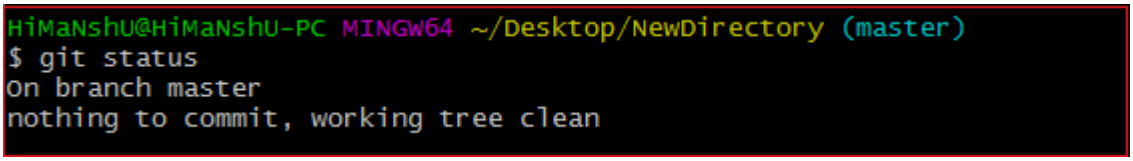
As we can see from the above output, the status is showing as "**nothing added to commit but untracked files present (use "git add" to track)**". The status command also displays the suggestions. As in the above output, it is suggesting to use the add command to track the file.

Let's track the file and will see the status after adding a file to the repository. To track the file, run the add command. Consider the below output:



From the above output, we can see that the status after staging the file is showing as "**changes to be committed**".

Before committing blindly, we can check the status. This command will help us to avoid the changes that we don't want to commit. Let's commit it and then check the status. Consider the below output:



We can see that the current status after committing the file is clean as it was before.

Status when an existing file is modified

Let's check the status when an existing file is modified. To modify file, run the **echo** command as follows:

1. $ echo "Text"> Filename

The above command will add the text to the specified file, now check the status of the repository. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ echo "Add some text "> newfile3,txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        newfile3,txt

nothing added to commit but untracked files present (use "git add" to
track)
```

We can see that the updated file is displayed as untracked files. It is shown in red color because it is not staged yet. When it will stage, its color will change to Green. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newfile3,txt
```

Status when a file is deleted

Let's check the status when a file is deleted from the repository. To delete a file from the repository, run the rm command as follows:

1. $ git rm < File Name>

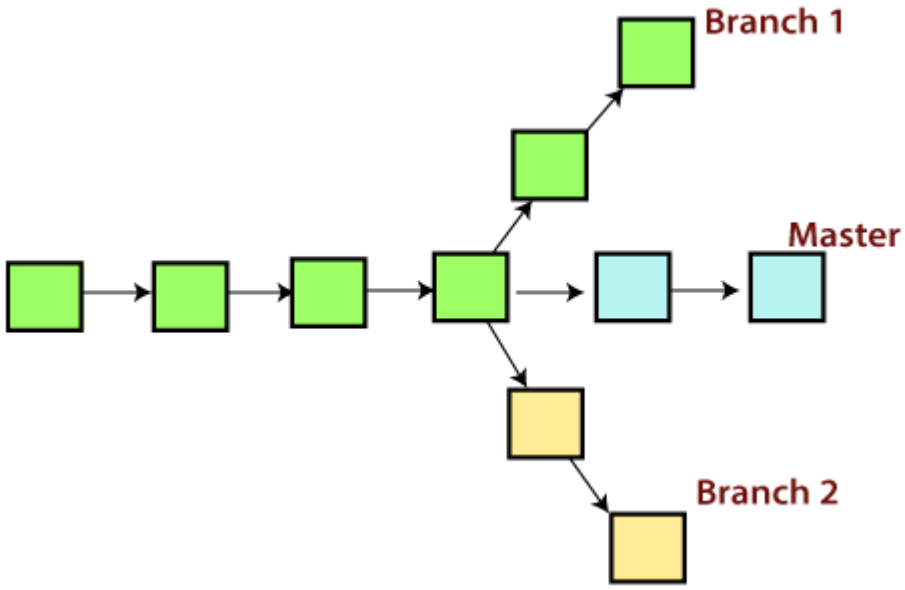
The above command will delete the specified file from the repository. Now, check the status of the repository. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git rm newfile3.txt
rm 'newfile3.txt'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/NewDirectory (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    newfile3.txt
```

The current status of the repository has been updated as deleted.

Git Branch

A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems. A Git project can have more than one branch. These branches are a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes. So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



Git Master Branch

The master branch is a default branch in Git. It is instantiated when first commit made on the project. When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.

Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

Operations on Branches

We can perform various operations on Git branches. The **git branch command** allows you to **create**, **list**, **rename** and **delete** branches. Many operations on branches are

applied by git checkout and git merge command. So, the git branch is tightly integrated with the **git checkout** and **git merge commands**.

The Operations that can be performed on a branch:

Create Branch

You can create a new branch with the help of the **git branch** command. This command will be used as:

Syntax:

1. $ git branch **<branch** name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch B1
```

This command will create the **branch B1** locally in Git directory.

List Branch

You can List all of the available branches in your repository by using the following command.

Either we can use **git branch - list** or **git branch** command to list the available branches in the repository.

Syntax:

1. $ git branch --list

or

1. $ git branch

Output:

Here, both commands are listing the available branches in the repository. The symbol * is representing currently active branch.

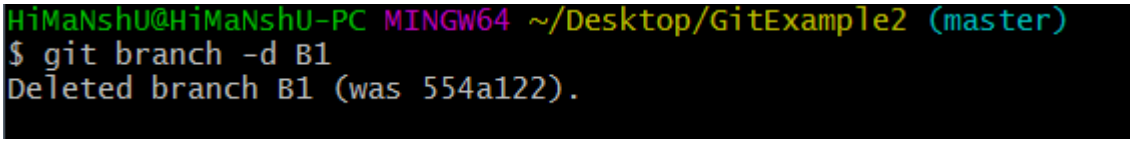
Delete Branch

You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

Syntax:

1. $ git branch -d**<branch** name**>**

Output:



This command will delete the existing branch B1 from the repository.

The **git branch d** command can be used in two formats. Another format of this command is **git branch D**. The '**git branch D**' command is used to delete the specified branch.

1. $ git branch -D **<branch** name**>**

Delete a Remote Branch

You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

Syntax:

1. $ git push origin -delete **<branch** name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin --delete branch2
To https://github.com/ImDwivedi1/GitExample2
 - [deleted]         branch2

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see in the above output, the remote branch named **branch2** from my GitHub account is deleted.

Switch Branch

Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:

1. $ git checkout**<branch** name**>**

Switch from master Branch

You can switch from master to any other branch available on your repository without making any commit.

Syntax:

1. $ git checkout **<branch** name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout branch4
Switched to branch 'branch4'
```

As you can see in the output, branches are switched from **master** to **branch4** without making any commit.

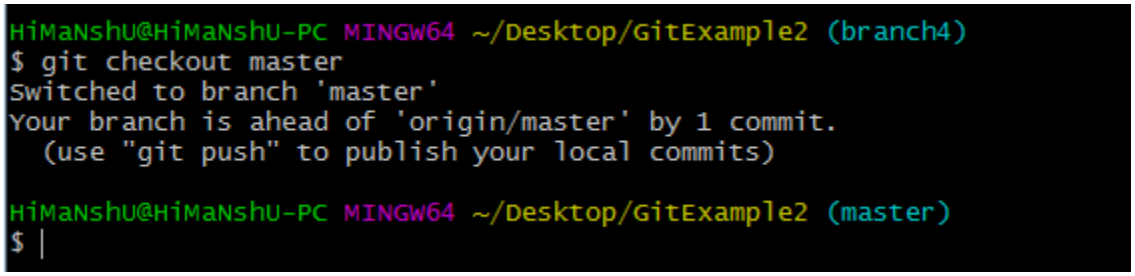
Switch to master branch

You can switch to the master branch from any other branch with the help of below command.

Syntax:

1. $ git branch -m master

Output:



As you can see in the above output, branches are switched from **branch1 to master** without making any commit.

Rename Branch

We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

Syntax:

1. $ git branch -m **<old** branch name**><new** branch name**>**

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch -m branch4 renamedB1

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
* master
  renamedB1

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see in the above output, **branch4** renamed as **renamedB1**.

Merge Branch

Git allows you to merge the other branch with the currently active branch. You can merge two branches with the help of **git merge** command. Below command is used to merge the branches:

Syntax:

1. $ git merge **<branch** name**>**

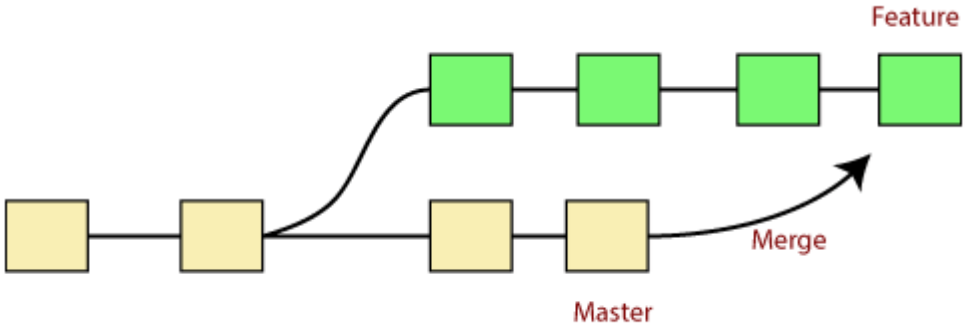
Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge renamedB1
Already up to date.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

From the above output, you can see that **the master** branch **merged** with **renamedB1**. Since I have made no-commit before merging, so the output is showing as already up to date.

Git Merge and Merge Conflict

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



It is used to maintain distinct lines of development; at some stage, you want to merge the changes in one branch. It is essential to understand how merging works in Git.

In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

The "git merge" command

The git merge command is used to merge the branches.

The syntax for the git merge command is as:

1. $ git merge **<query>**

It can be used in various context. Some are as follows:

Scenario1: To merge the specified commit to currently active branch:

Use the below command to merge the specified commit to currently active branch.

1. $ git merge **<commit>**

The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git add newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git commit -m "edited newfile1.txt"
[test d2bb07d] edited newfile1.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit d2bb07dc9352e194b13075dcfd28e4de802c070b (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 11:27:44 2019 +0530

    edited newfile1.txt

commit 2852e020909dfe705707695fd6d715cd723f9540 (test2, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added
```

Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout test2
Switched to branch 'test2'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge  d2bb07dc9352e194b13075dcfd28e4de802c070b
Updating 2852e02..d2bb07d
Fast-forward
 newfile1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$
```

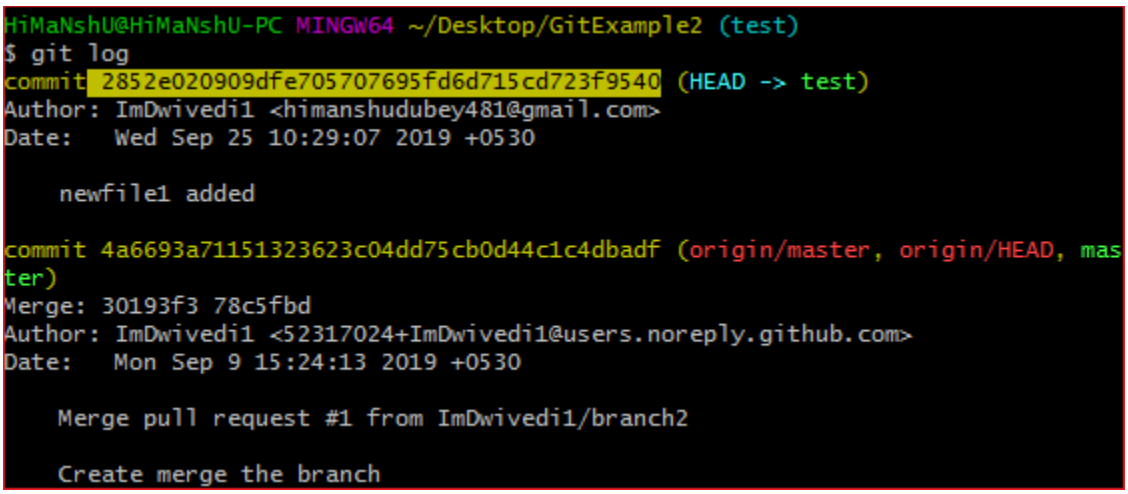
In the above output, we have merged the previous commit in the active branch test2.

Scenario2: To merge commits into the master branch:

To merge a specified commit into master, first discover its commit id. Use the log command to find the particular commit id.

1. $git log

See the below output:



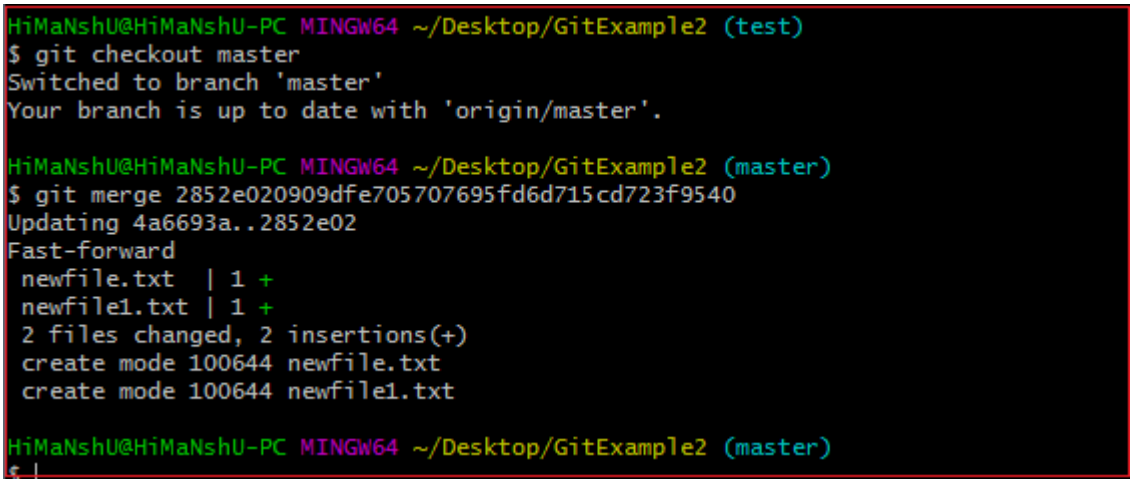
To merge the commits into the master branch, switch over to the master branch.

1. $ git checkout master

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name. The syntax for this is as follows:

1. $ git merge master

See the below output:



As shown in the above output, the commit for the commit id ***2852e020909dfe705707695fd6d715cd723f9540*** has merged into the master

branch. Two files have changed in master branch. However, we have made this commit in the **test** branch. So, it is possible to merge any commit in any of the branches.

Open new files, and you will notice that the new line that we have committed to the test branch is now copied on the master branch.

Scenario 3: Git merge branch.

Git allows merging the whole branch in another branch. Suppose you have made many changes on a branch and want to merge all of that at a time. Git allows you to do so. See the below example:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add newfile1.txt
```

In the given output, I have made changes in newfile1 on the test branch. Now, I have committed this change in the test branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m "edit newfile1"
[test2 a3644e1] edit newfile1
 1 file changed, 1 insertion(+)
```

Now, switch to the desired branch you want to merge. In the given example, I have switched to the master branch. Perform the below command to merge the whole branch in the active branch.

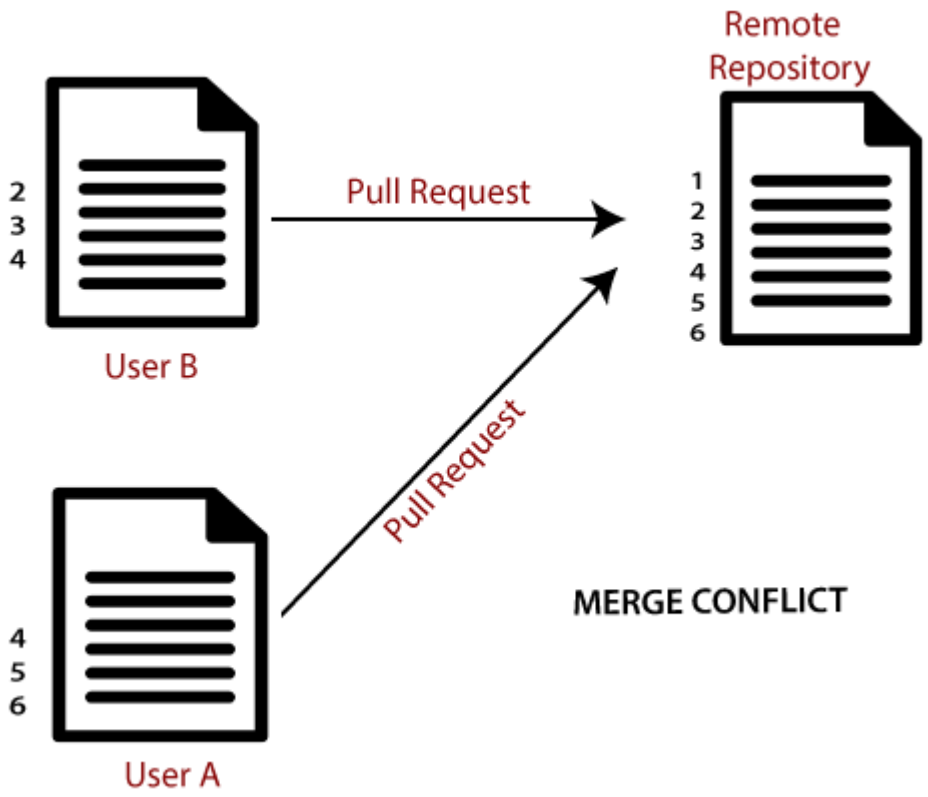
1. $ git merge **<branchname>**

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge test2
Updating 2852e02..a3644e1
Fast-forward
 newfile1.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

As you can see from the given output, the whole commits of branch test2 have merged to branch master.

Git Merge Conflict

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



Let's understand it by an example.

Suppose my remote repository has cloned by two of my team member **user1** and **user2**. The user1 made changes as below in my projects index file.

.bash_profile | index.html | index.html

```
1  <head>
2  <body>
3  <title> This is a Git example</Title>
4  <h1> Git is a version control</h1>
5  </head>
6  </body>
```

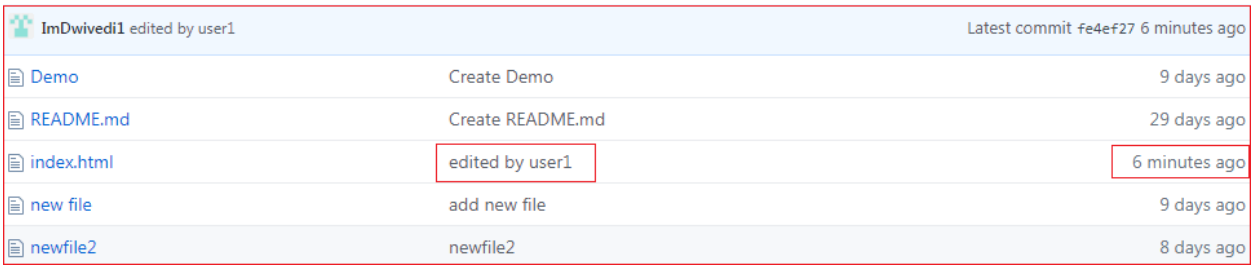
Update it in the local repository with the help of git add command.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git add index.html
```

Now commit the changes and update it with the remote repository. See the below output:

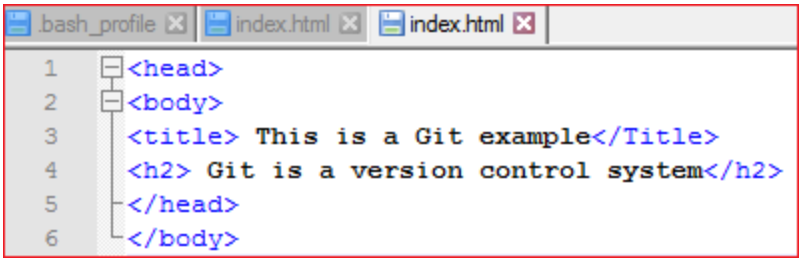
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git commit -m "edited by user1"
[master fe4ef27] edited by user1
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
$ git push origin master
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 345.00 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/ImDwivedi1/Git-Example
   039c01b..fe4ef27  master -> master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/user1repo (master)
```

Now, my remote repository will look like this:

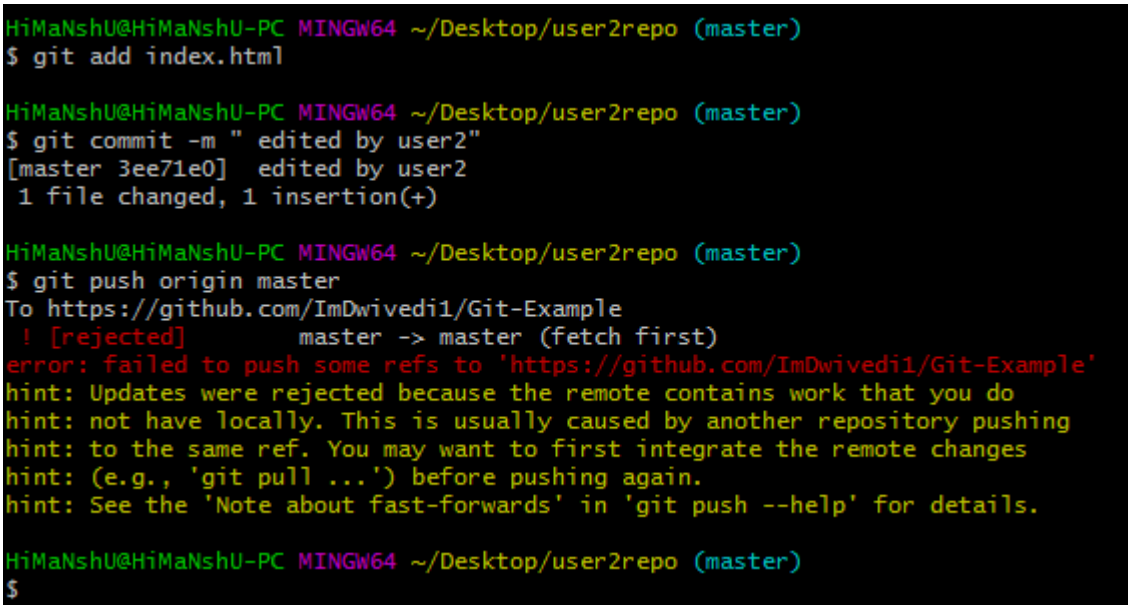


It will show the status of the file like edited by whom and when.

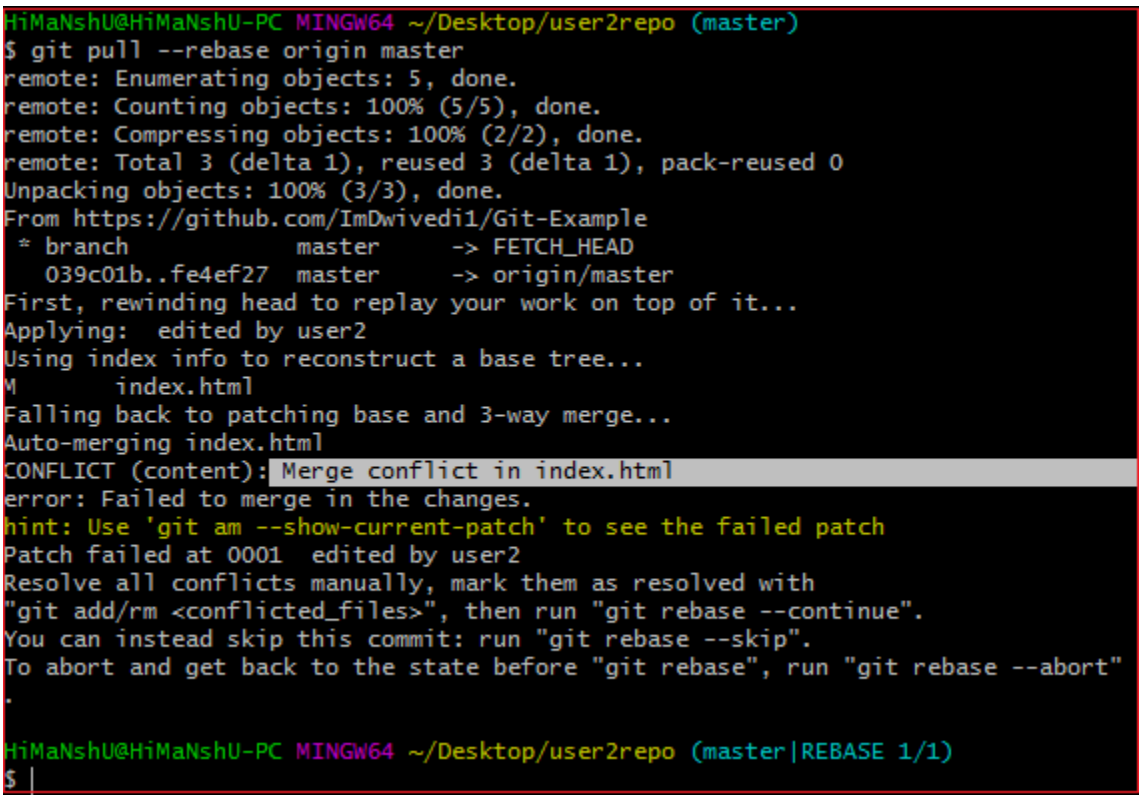
Now, at the same time, **user2** also update the index file as follows.



User2 has added and committed the changes in the local repository. But when he tries to push it to remote server, it will throw errors. See the below output:



In the above output, the server knows that the file is already updated and not merged with other branches. So, the push request was rejected by the remote server. It will throw an error message like **[rejected] failed to push some refs to <remote URL>**. It will suggest you to pull the repository first before the push. See the below command:



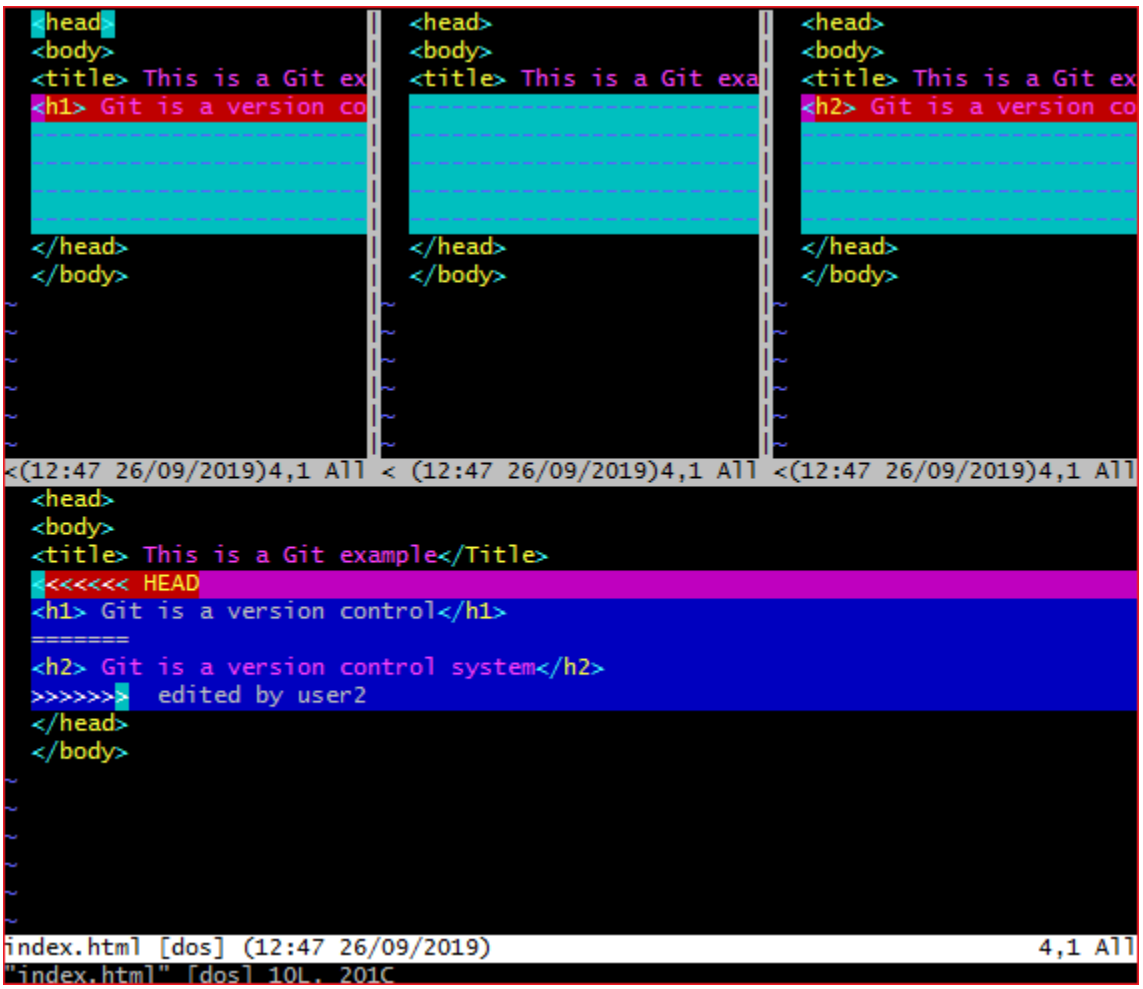
In the given output, git rebase command is used to pull the repository from the remote URL. Here, it will show the error message like **merge conflict in <filename>**.

Resolve Conflict:

To resolve the conflict, it is necessary to know whether the conflict occurs and why it occurs. Git merge tool command is used to resolve the conflict. The merge command is used as follows:

1. $ git mergetool

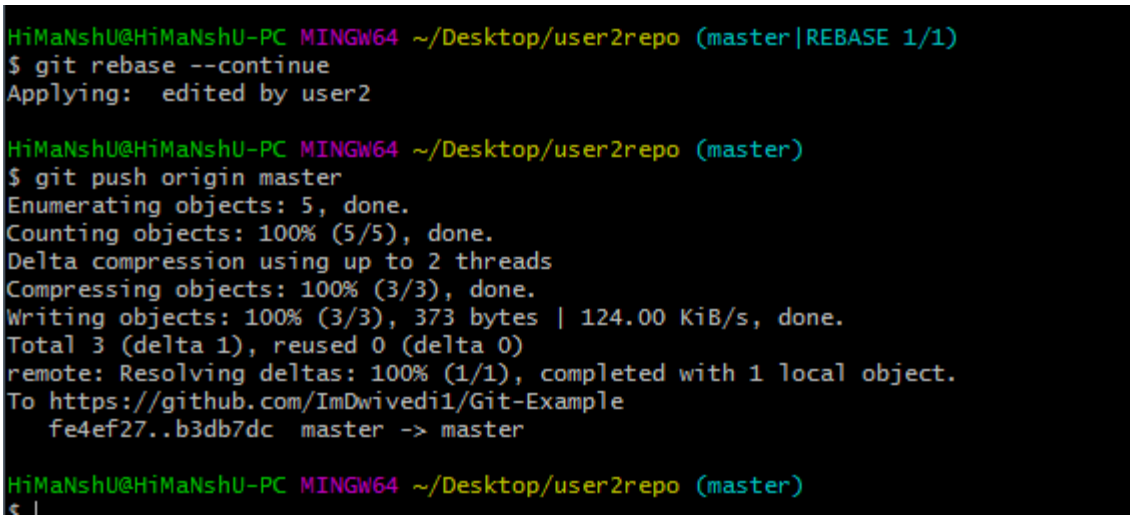
In my repository, it will result in:



The above output shows the status of the conflicted file. To resolve the conflict, enter in the insert mode by merely pressing **I key** and make changes as you want. Press the **Esc key**, to come out from insert mode. Type the: **w!** at the bottom of the editor to save and exit the changes. To accept the changes, use the rebase command. It will be used as follows:

1. $ git rebase --continue

Hence, the conflict has resolved. See the below output:



In the above output, the conflict has resolved, and the local repository is synchronized with a remote repository.

To see that which is the first edited text of the merge conflict in your file, search the file attached with conflict marker **<<<<<<<**. You can see the changes from the **HEAD** or base branch after the line **<<<<<<< HEAD** in your text editor. Next, you can see the divider like **=======**. It divides your changes from the changes in the other branch, **followed by >>>>>>> BRANCH-NAME**. In the above example, user1 wrote "<h1> Git is a version control</h1>" in the base or HEAD branch and user2 wrote "<h2> Git is a version control</h2>".

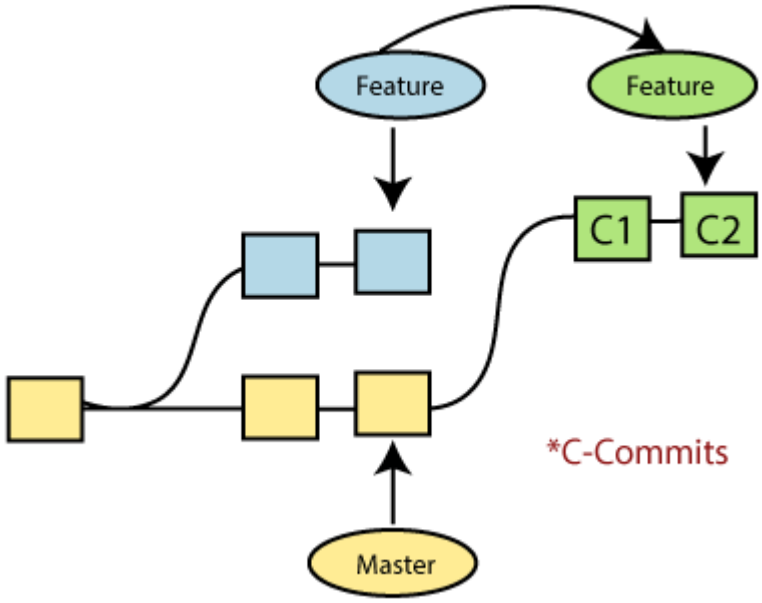
Decide whether you want to keep only your branch's changes or the other branch's changes, or create a new change. Delete the conflict markers **<<<<<<<, =======, >>>>>>>** and create final changes you want to merge.

Git Rebase

Rebasing is a process to reapply commits on top of another base trip. It is used to apply a sequence of commits from distinct branches into a final commit. It is an alternative of git merge command. It is a linear process of merging.

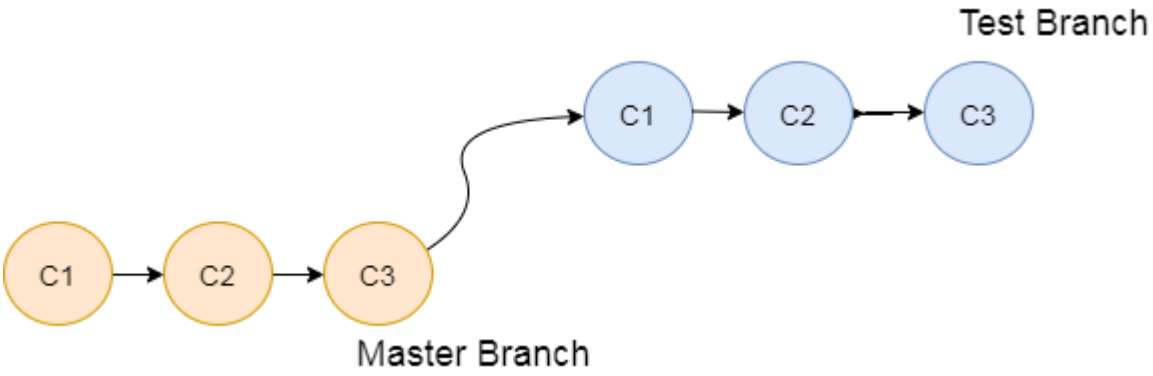
In Git, the term rebase is referred to as the process of moving or combining a sequence of commits to a new base commit. Rebasing is very beneficial and it visualized the process in the environment of a feature branching workflow.

It is good to rebase your branch before merging it.



Generally, it is an alternative of git merge command. Merge is always a forward changing record. Comparatively, rebase is a compelling history rewriting tool in git. It merges the different commits one by one.

Suppose you have made three commits in your master branch and three in your other branch named test. If you merge this, then it will merge all commits in a time. But if you rebase it, then it will be merged in a linear manner. Consider the below image:



The above image describes how git rebase works. The three commits of the master branch are merged linearly with the commits of the test branch.

Merging is the most straightforward way to integrate the branches. It performs a three-way merge between the two latest branch commits.

How to Rebase

When you made some commits on a feature branch (test branch) and some in the master branch. You can rebase any of these branches. Use the git log command to track the changes (commit history). Checkout to the desired branch you want to rebase. Now perform the rebase command as follows:

Syntax:

1. $git rebase **<branch** name**>**

If there are some conflicts in the branch, resolve them, and perform below commands to continue changes:

1. $ git status

It is used to check the status,

1. $git rebase --continue

The above command is used to continue with the changes you made. If you want to skip the change, you can skip as follows:

1. $ git rebase --skip

When the rebasing is completed. Push the repository to the origin. Consider the below example to understand the git merge command.

Suppose that you have a branch say **test2** on which you are working. You are now on the test2 branch and made some changes in the project's file **newfile1.txt**.

Add this file to repository:

1. $ git add newfile1.txt

Now, commit the changes. Use the below command:

1. $ git commit -m "new commit for test2 branch."

The output will look like:

```
[test2 a835504] new commitfor test2 branch
 1 file changed, 1 insertion(+)
```

Switch the branch to master:

1. $ git checkout master

Output:

```
Switched to branch 'master.'
Your branch is up to date with 'origin/master.'
```

Now you are on the master branch. I have added the changes to my file, says **newfile.txt**. The below command is used to add the file in the repository.

1. $ git add newfile.txt

Now commit the file for changes:

1. $ git commit -m " new commit made on the master branch."

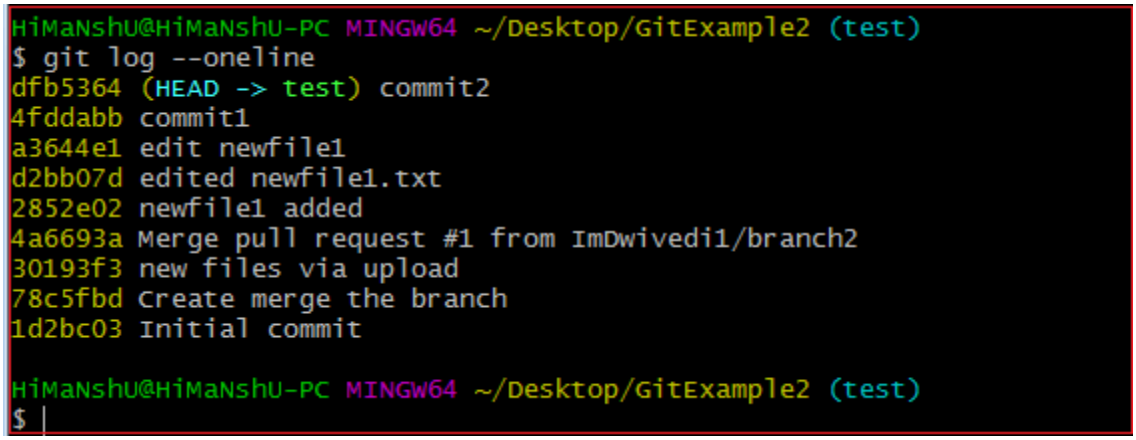
Output:

```
[master 7fe5e7a]  new commit made on master
 1 file changed, 1 insertion(+)
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
```

To check the log history, perform the below command.

1. $ git log --oneline

Output:



As we can see in the log history, there is a new commit in the master branch. If I want to rebase my test2 branch, what should I do? See the below rebase branch scenario:

Rebase Branch

If we have many commits from distinct branches and want to merge it in one. To do so, we have two choices either we can merge it or rebase it. It is good to rebase your branch.

From the above example, we have committed to the master branch and want to rebase it on the test2 branch. Let's see the below commands:

1. $ git checkout test2

This command will switch you on the test2 branch from the master.

Output:

```
Switched to branch 'test2.'
```

Now you are on the test2 branch. Hence, you can rebase the test2 branch with the master branch. See the below command:

1. $ git rebase master

This command will rebase the test2 branch and will show as **Applying: new commit on test2 branch**. Consider the below output:

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git rebase master
First, rewinding head to replay your work on top of it...
Fast-forwarded test to master.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$
```

Git Interactive Rebase

Git facilitates with Interactive Rebase; it is a potent tool that allows various operations like **edit, rewrite, reorder,** and more on existing commits. Interactive Rebase can only be operated on the currently checked out branch. Therefore, set your local HEAD branch at the sidebar.

Git interactive rebase can be invoked with rebase command, just type **-i** along with rebase command. Here '**i**' stands for interactive. Syntax of this command is given below:

Syntax:

1. $ git rebase -i

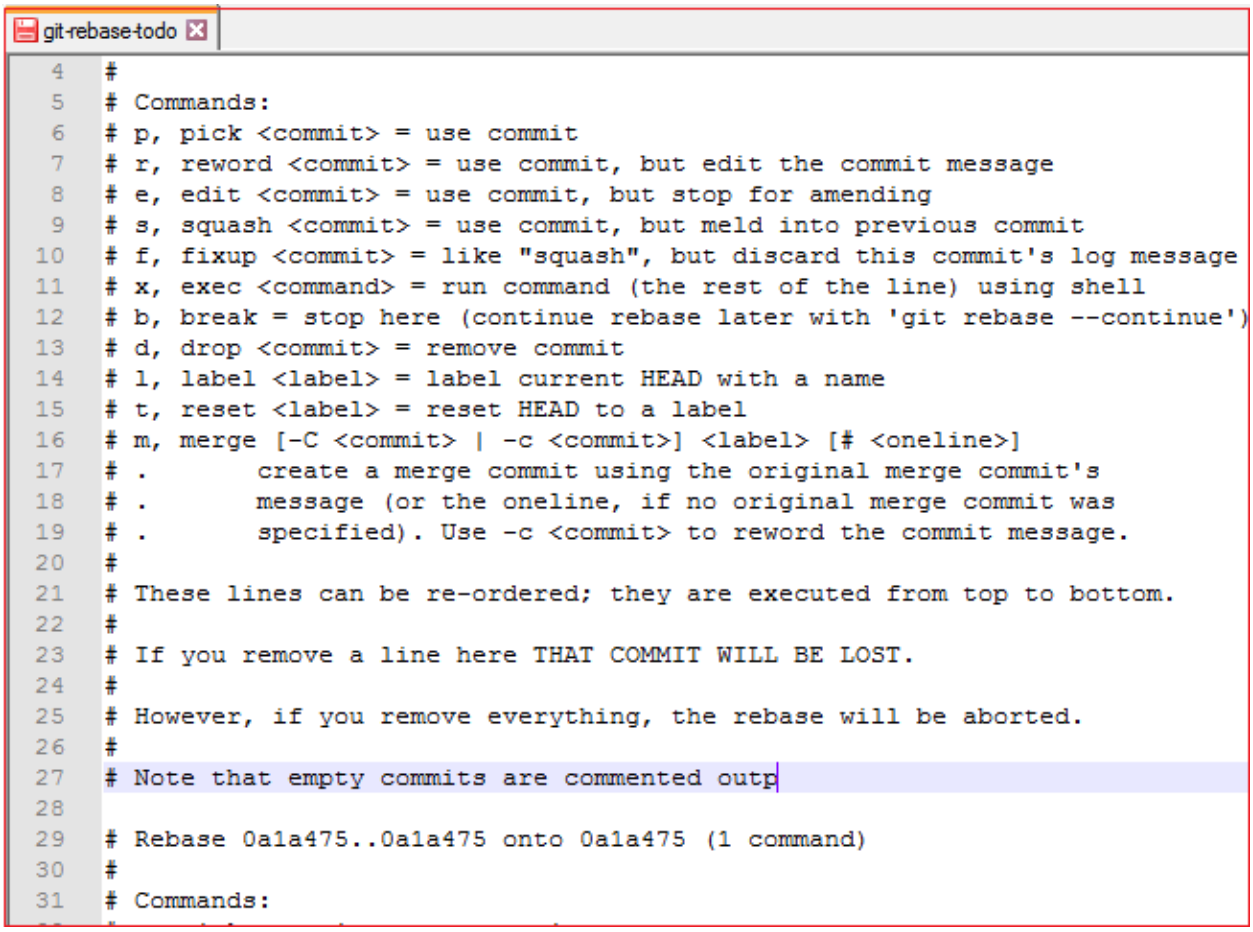
It will list all the available interactive options.

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git rebase -i
hint: Waiting for your editor to close the file...
```

After the given output, it will open an editor with available options. Consider the below output:

Output:



When we perform the git interactive rebase command, it will open your default text editor with the above output.

The options it contains are listed below:

- Pick
- Reword
- Edit
- Squash
- Fixup
- Exec
- Break
- Drop
- Label

- Reset
- Merge

The above options perform their specific tasks with git-rebase. Let's understand each of these options in brief.

Pick (-p):

Pick stands here that the commit is included. Order of the commits depends upon the order of the pick commands during rebase. If you do not want to add a commit, you have to delete the entire line.

Reword (-r):

The reword is quite similar to pick command. The reword option paused the rebase process and provides a chance to alter the commit message. It does not affect any changes made by the commit.

Edit (-e):

The edit option allows for amending the commit. The amending means, commits can be added or changed entirely. We can also make additional commits before rebase continue command. It allows us to split a large commit into the smaller commit; moreover, we can remove erroneous changes made in a commit.

Squash (-s):

The squash option allows you to combine two or more commits into a single commit. It also allows us to write a new commit message for describing the changes.

Fixup (-f):

It is quite similar to the squash command. It discarded the message of the commit to be merged. The older commit message is used to describe both changes.

Exec (-x):

The exec option allows you to run arbitrary shell commands against a commit.

Break (-b):

The break option stops the rebasing at just position. It will continue rebasing later with '**git rebase --continue**' command.

Drop (-d):

The drop option is used to remove the commit.

Label (-l):

The label option is used to mark the current head position with a name.

Reset (-t):

The reset option is used to reset head to a label.

GitMerge vs. Rebase

It is a most common puzzling question for the git user's that when to use merge command and when to use rebase. Both commands are similar, and both are used to merge the commits made by the different branches of a repository.

Rebasing is not recommended in a shared branch because the rebasing process will create inconsistent repositories. For individuals, rebasing can be more useful than merging. If you want to see the complete history, you should use the merge. Merge tracks the entire history of commits, while rebase rewrites a new one.

Git rebase commands said as an alternative of git merge. However, they have some key differences:

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 Squash

In Git, the term squash is used to squash the previous commits into one. It is not a command; instead, it is a keyword. The squash is an excellent technique for group-specific changes before forwarding them to others. You can merge several commits into a single commit with the compelling interactive rebase command.

If you are a Git user, then you must have realized the importance of squashing a commit. Especially if you are an open-source contributor, then many times, you have to create a PR (pull request) with squashed commit. You can also squash commits if you have already created a PR.

Let's understand how to squash commits?

Git Squash Commits

Being a responsible contributor to Git, it is necessary to make the collaboration process efficient and meaningful. Git allows some powerful collaboration tools in different ways. Git squash is one of the powerful tools that facilitate efficient and less painful collaboration.

The squash is not any command; instead, it's one of many options available to you under git interactive rebases. The squash allows us to rewrite history. Suppose we have made

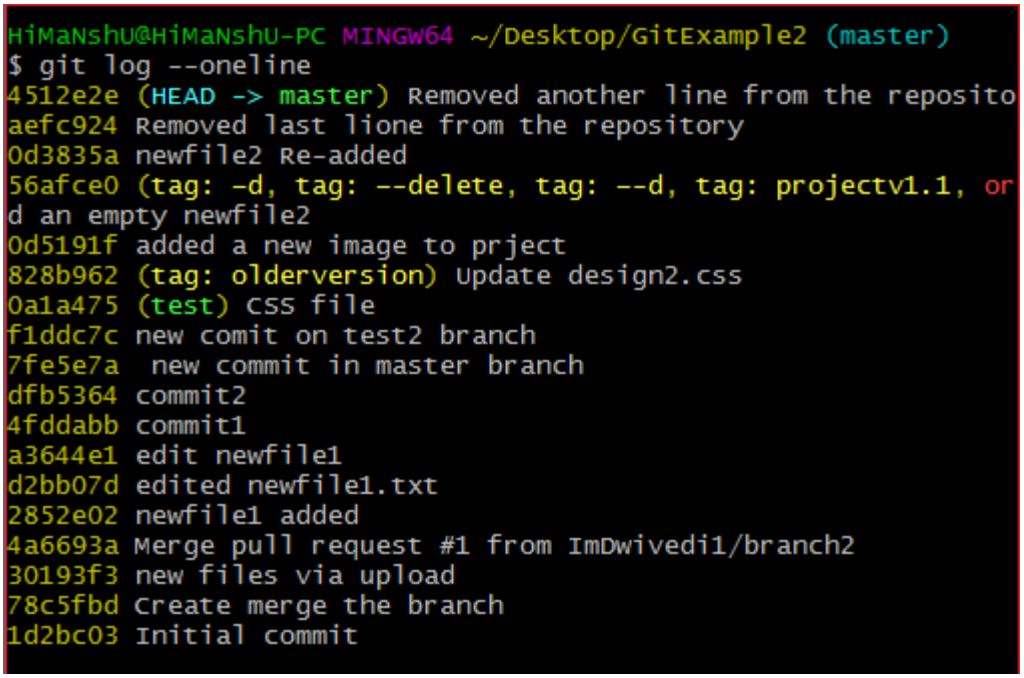
many commits during the project work, squashing all the commits into a large commit is the right choice than pushing. Let's understand how to squash two commits.

Step1: Check the commit history

To check the commit history, run the below command:

1. $ git log --oneline

The given command will display the history in one line. We can track the history and choose the commits we want to squash. Consider the below output:

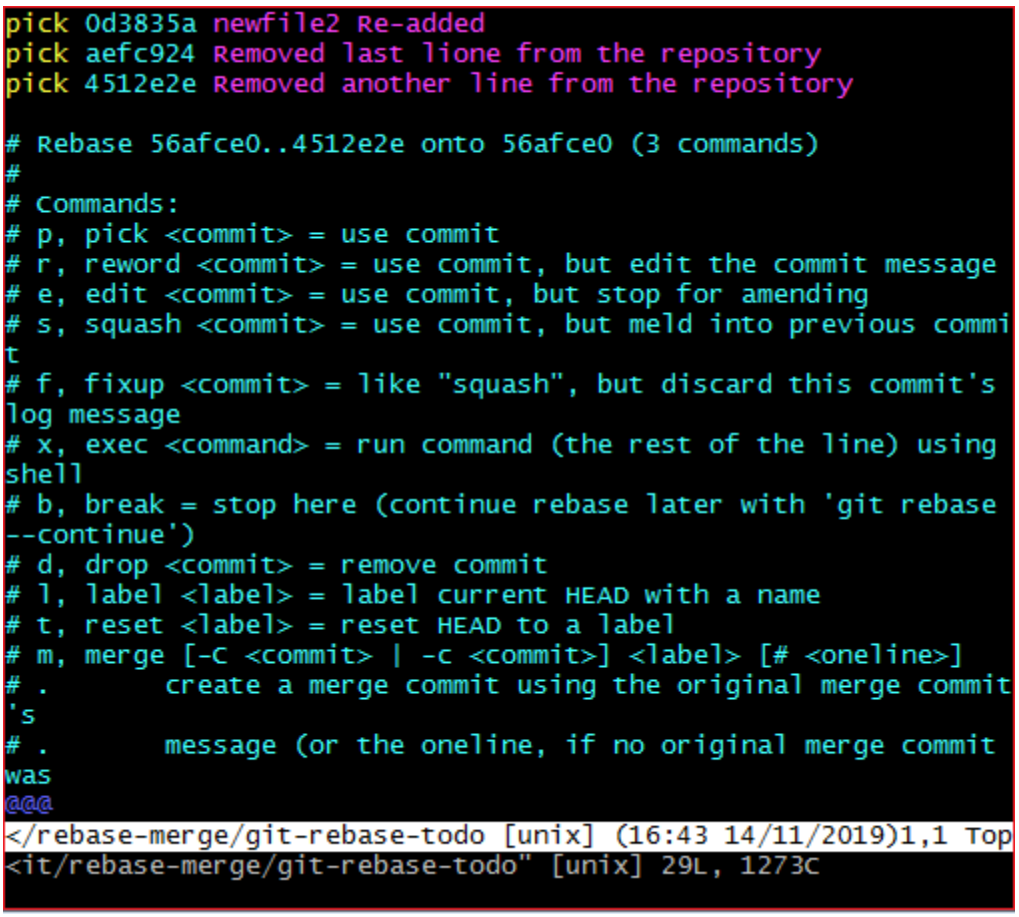


Step 2: Choose the commits to squash.

Suppose we want to squash the last commits. To squash commits, run the below command:

1. $ git rebase -i HEAD ~3

The above command will open your default text editor and will squash the last three commits. The editor will open as follows:

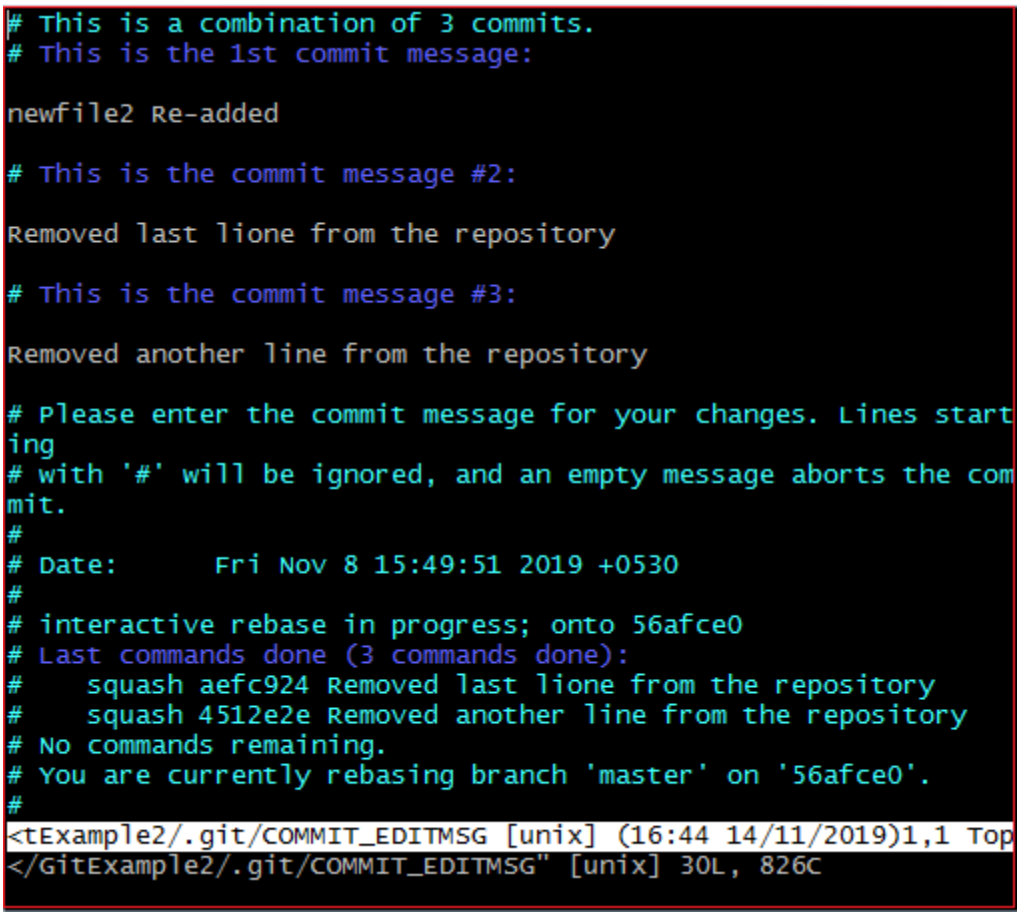


From the above image, we can see previous commits shown at the top of the editor. If we want to merge them into a single commit, then we have to **replace** the word **pick** with **the squash** on the top of the editor. To write on the editor, press '**i**' button to enter in **insert mode**. After editing the document, press the **:wq** to save and exit from the editor.

Step 3: update the commits

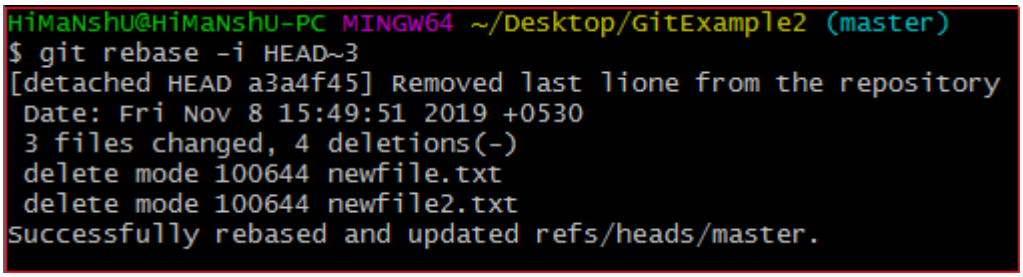
On pressing **enter** key, a new window of the text editor will be opened to confirm the commit. We can edit the commit message on this screen.

I am editing my first commit message because it will be a combination of all three commits. Consider the below image:

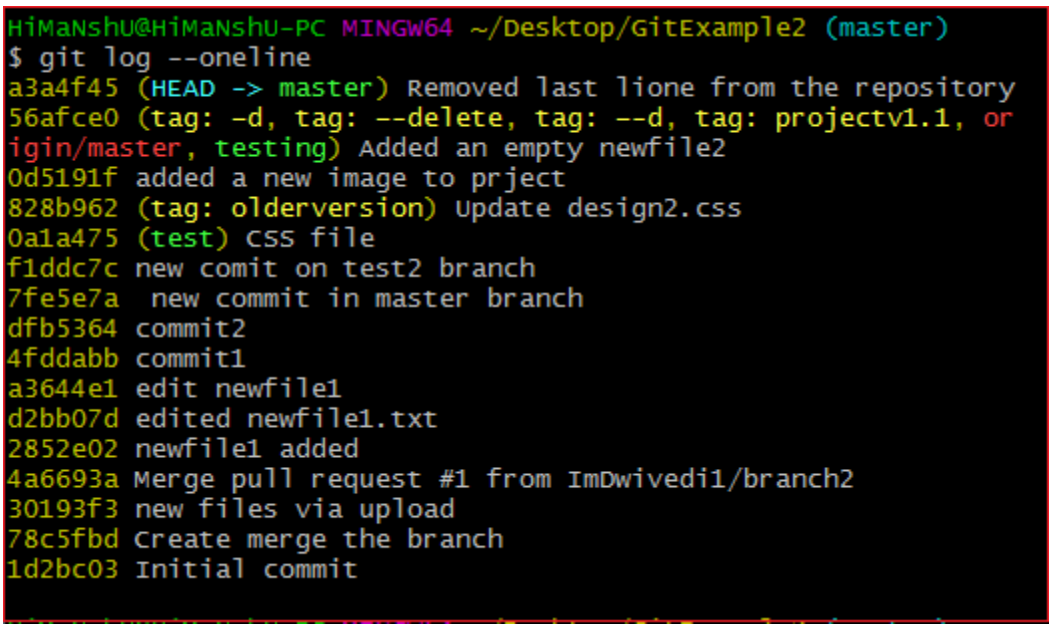


The above image is the editor screen to confirm the merging of commits. Here we can update the commit messages. To edit on this editor, press the '**i**' button for insert mode and edit the desired text. Press the **:wq** keys, to save and exit from the editor.

When we exit the editor, it will show the description of updates. Consider the below output:



The above output is listing the description of changes that have been made on the repository. Now, the commits have been squashed. Check the commit history for confirmation with the help of the git log. Consider the below output:



Step 4: Push the squashed commit

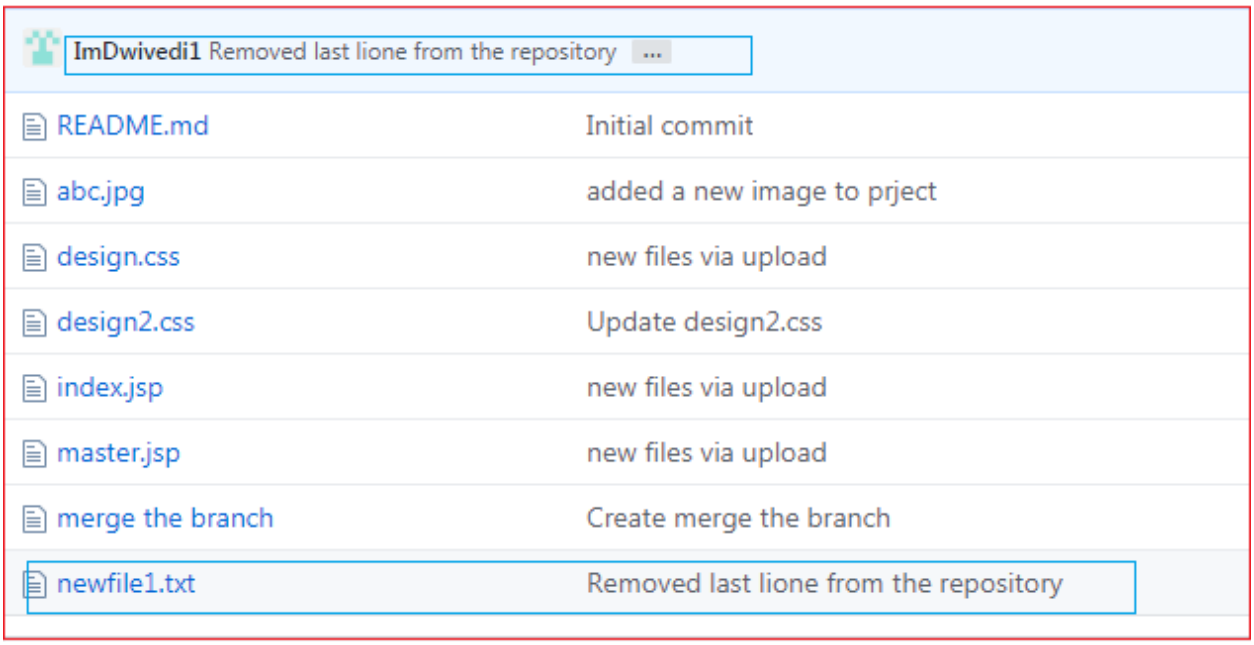
Now, we can push this squashed commit on the remote server. To push this squashed commit, run the below command:

1. $ git push origin master

Or

1. $ git push -f origin master

The above command will push the changes on the remote server. We can check this commit on our remote repository. Consider the below image:



As you can see from the above image. A new commit has been added to my remote repository.

Drawbacks of Squashing

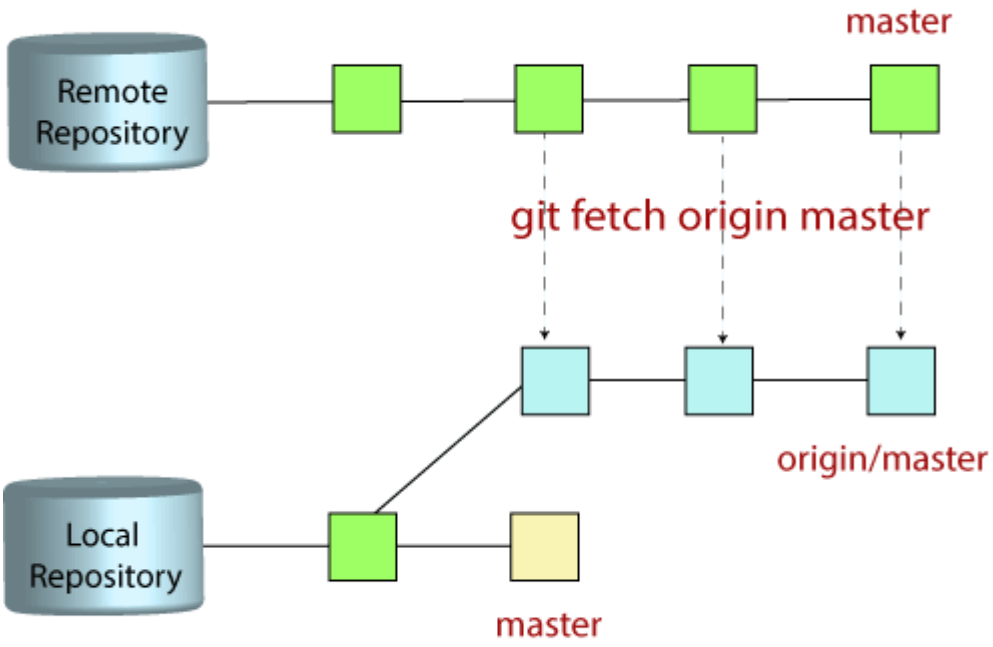
There are no significant drawbacks of squashing. But we can consider some facts that may affect the project. These facts are as follows:

The squashing commits, and rebasing changes the history of the repository. If any contributor does not pay attention to the updated history, then it may create conflict. I suggest a clean history because it is more valuable than another one. Although we can check the original history in the ref log.

There is another drawback, we may lose granularity because of squashing. Try to make minimum squashes while working with Git. So, if you are new on Git, then try to stay away from squash.

Git Fetch

Git "fetch" Downloads commits, objects and refs from another repository. It fetches branches and tags from one or more repositories. It holds repositories along with the objects that are necessary to complete their histories to keep updated remote-tracking branches.



The "git fetch"command

The "**git fetch**" **command** is used to pull the updates from remote-tracking branches. Additionally, we can get the updates that have been pushed to our remote branches to our local machines. As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.

How to fetch Git Repository

We can use fetch command with many arguments for a particular data fetch. See the below scenarios to understand the uses of fetch command.

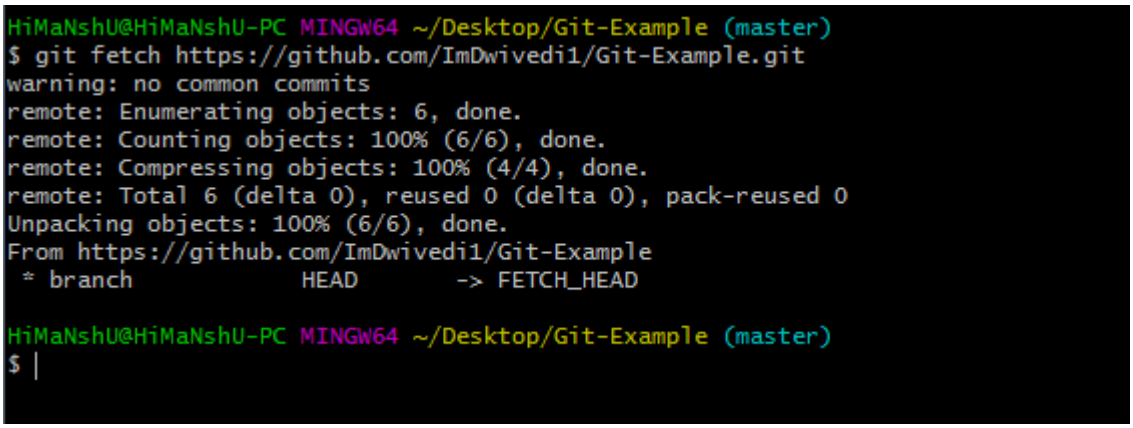
Scenario 1: To fetch the remote repository:

We can fetch the complete repository with the help of fetch command from a repository URL like a pull command does. See the below output:

Syntax:

1. $ git fetch< repository Url>

Output:



In the above output, the complete repository has fetched from a remote URL.

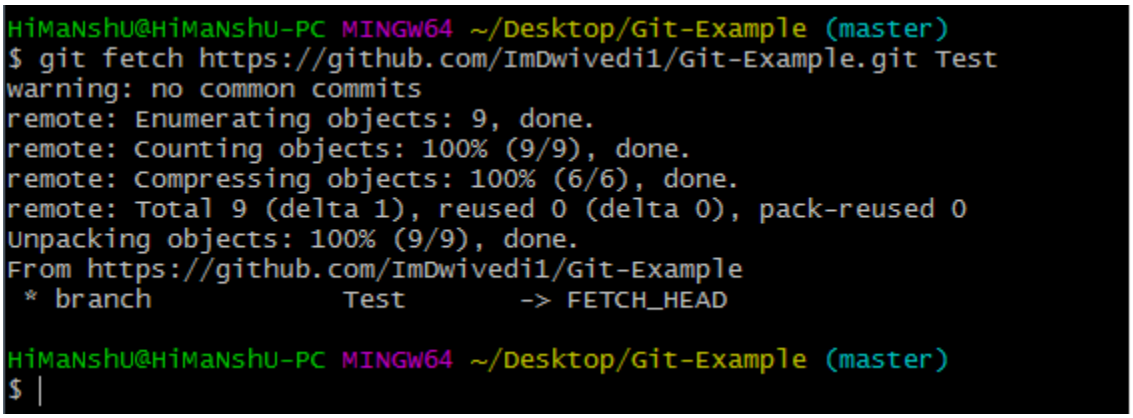
Scenario 2: To fetch a specific branch:

We can fetch a specific branch from a repository. It will only access the element from a specific branch. See the below output:

Syntax:

1. $ git fetch <branch URL><branch name>

Output:



In the given output, the specific branch **test** has fetched from a remote URL.

Scenario 3: To fetch all the branches simultaneously:

The git fetch command allows to fetch all branches simultaneously from a remote repository. See the below example:

Syntax:

1. $ git fetch -all

Output:

In the above output, all the branches have fetched from the repository Git-Example.

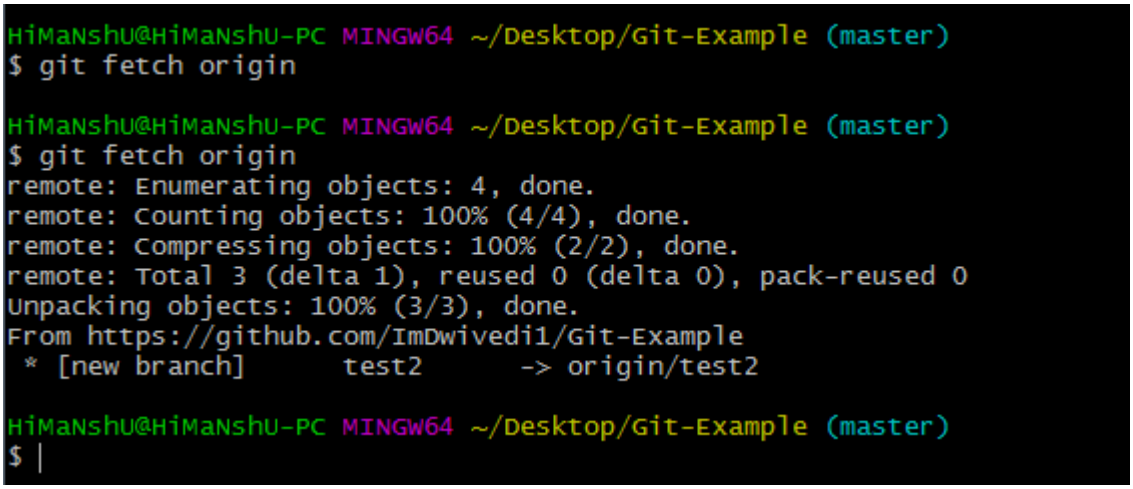
Scenario 4: To synchronize the local repository:

Suppose, your team member has added some new features to your remote repository. So, to add these updates to your local repository, use the git fetch command. It is used as follows.

Syntax:

1. $ git fetch origin

Output:



In the above output, new features of the remote repository have updated to my local system. In this output, the branch **test2** and its objects are added to the local repository.

The git fetch can fetch from either a single named repository or URL or from several repositories at once. It can be considered as the safe version of the git pull commands.

The git fetch downloads the remote content but not update your local repo's working state. When no remote server is specified, by default, it will fetch the origin remote.

Differences between git fetch and git pull

To understand the differences between fetch and pull, let's know the similarities between both of these commands. Both commands are used to download the data from a remote repository. But both of these commands work differently. Like when you do a git pull, it gets all the changes from the remote or central repository and makes it available to your corresponding branch in your local repository. When you do a git fetch, it fetches all the changes from the remote repository and stores it in a separate branch in your local repository. You can reflect those changes in your corresponding branches by merging.

So basically,

1. git pull = git fetch + git merge

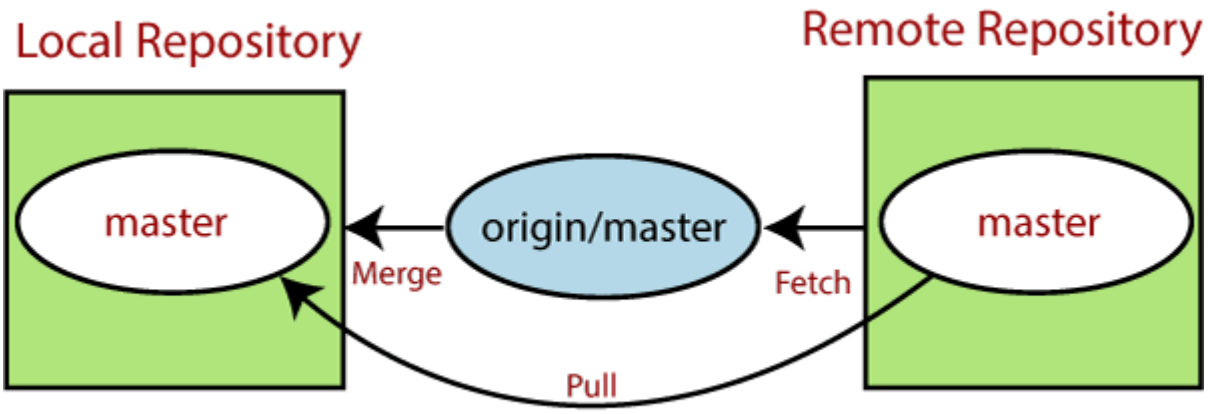
Git Fetch vs. Pull

Some of the key differences between both of these commands are as follows:

git fetch	git pull
Fetch downloads only new data from a remote repository.	Pull is used to update your current HEAD branch with the latest changes from the remote server.
Fetch is used to get a new view of all the things that happened in a remote repository.	Pull downloads new data and directly integrates it into your current working copy files.
Fetch never manipulates or spoils data.	Pull downloads the data and integrates it with the current working file.
It protects your code from merge conflict.	In git pull, there are more chances to create the **merge conflict**.
It is better to use git fetch command with git merge command on a pulled repository.	It is not an excellent choice to use git pull if you already pulled any repository.

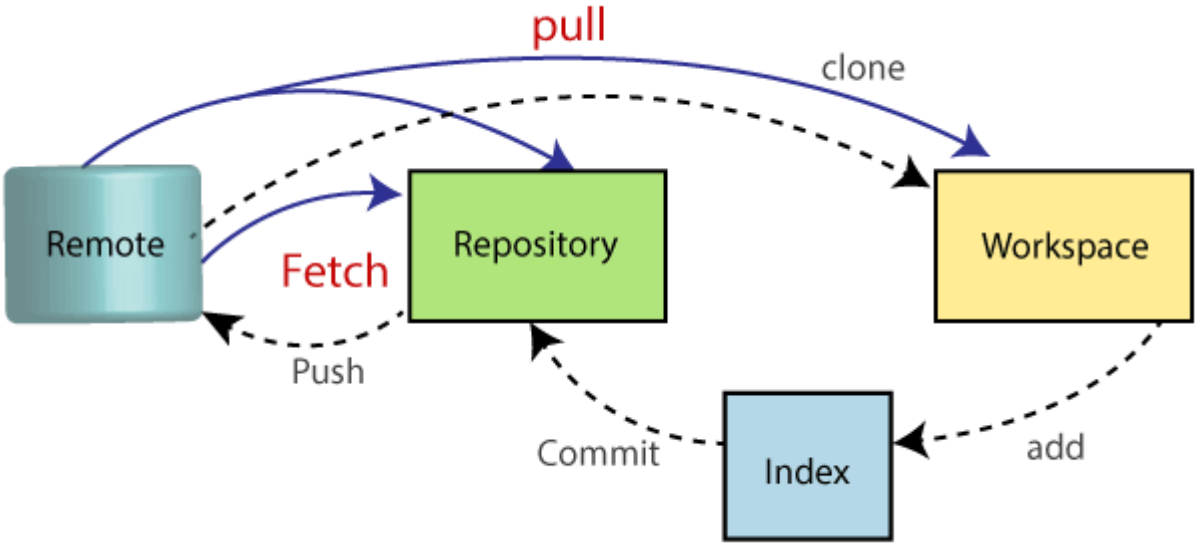
Git Pull / Pull Request

The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repository.



Pull request is 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.

The below figure demonstrates how pull acts between different locations and how it is similar or dissimilar to other related commands.



The "git pull" command

The pull command is used to access the changes (commits)from a remote repository to the local repository. It updates the local branches with the remote-tracking branches. Remote tracking branches are branches that have been set up to push and pull from the remote repository. Generally, it is a collection of the fetch and merges command. First, it fetches the changes from remote and combined them with the local repository.

The syntax of the git pull command is given below:

Syntax:

1. $ git pull **<option>** [**<repository** URL**><refspec>**...]

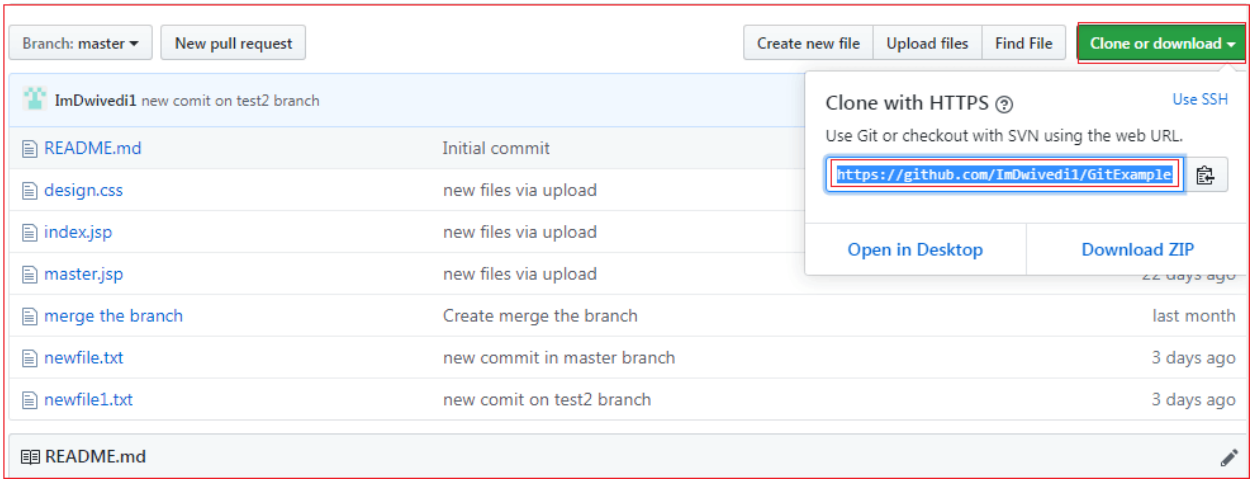
In which:

<option>: Options are the commands; these commands are used as an additional option in a particular command. Options can be **-q** (quiet), **-v** (verbose), **-e**(edit) and more.

<repository URL>: Repository URL is your remote repository's URL where you have stored your original repositories like GitHub or any other git service. This URL looks like:

1. https://github.com/ImDwivedi1/GitExample2.git

To access this URL, go to your account on GitHub and select the repository you want to clone. After that, click on the **clone** or **download** option from the repository menu. A new pop up window will open, select **clone with https option** from available options. See the below screenshot:



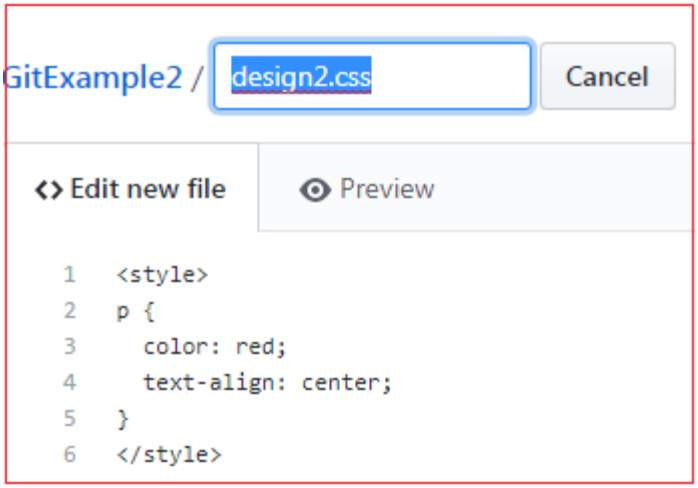
Copy the highlighted URL. This URL is used to Clone the repository.

<Refspec>: A ref is referred to commit, for example, head (branches), tags, and remote branches. You can check head, tags, and remote repository in **.git/ref** directory on your local repository. **Refspec** specifies and updates the refs.

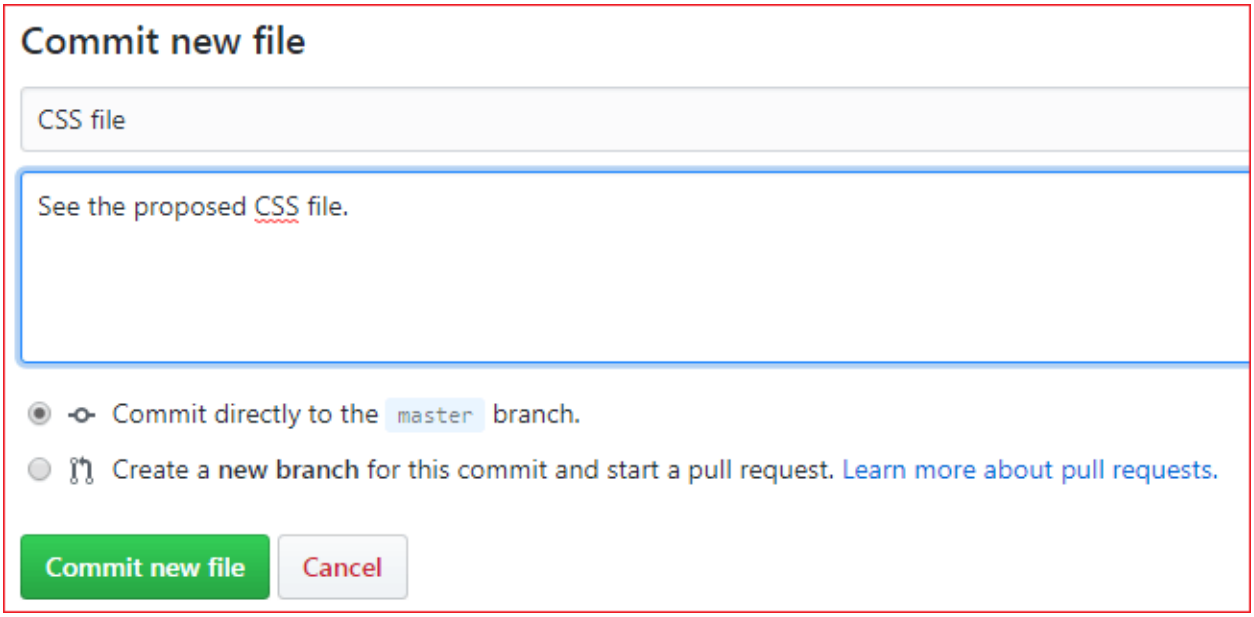
How to use pull:

It is essential to understand how it works and how to use it. Let's take an example to understand how it works and how to use it. Suppose I have added a new file say **design2.css** in my remote repository of project GitExample2.

To create the file first, go to create a file option given on repository sub-functions. After that, select the file name and edit the file as you want. Consider the below image.



Go to the bottom of the page, select a commit message and description of the file. Select whether you want to create a new branch or commit it directly in the master branch. Consider the below image:



Now, we have successfully committed the changes.

To pull these changes in your local repository, perform the git pull operation on your cloned repository. There are many specific options available for pull command. Let's have a look at some of its usage.

Default git pull:

We can pull a remote repository by just using the git pull command. It's a default option. Syntax of git pull is given below:

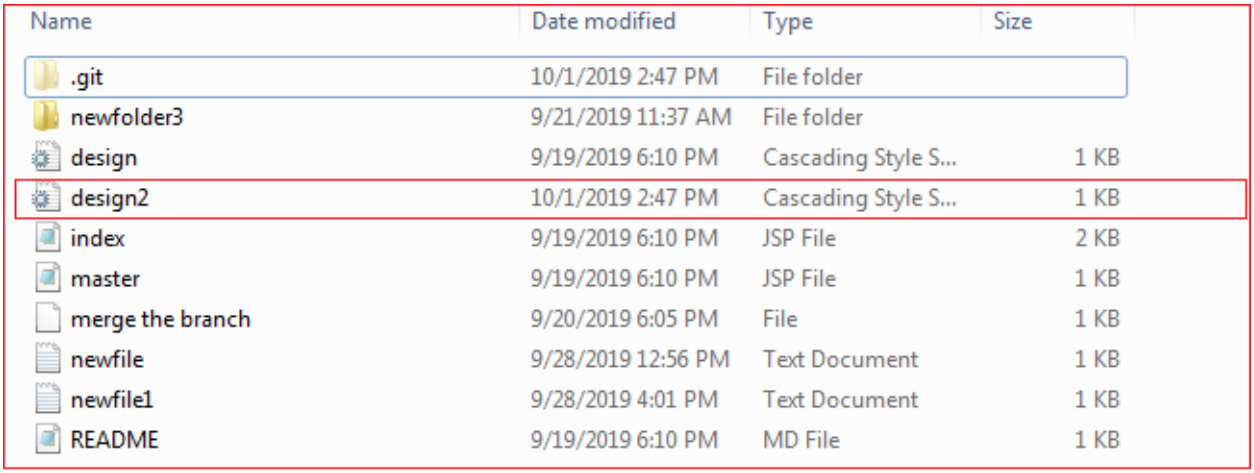
Syntax:

1. $ git pull

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/GitExample2
   f1ddc7c..0a1a475  master     -> origin/master
Updating f1ddc7c..0a1a475
Fast-forward
 design2.css | 6 ++++++
 1 file changed, 6 insertions(+)
 create mode 100644 design2.css

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

In the given output, the newly updated objects of the repository are fetched through the git pull command. It is the default version of the git pull command. It will update the newly created file **design2.css** file and related object in the local repository. See the below image.

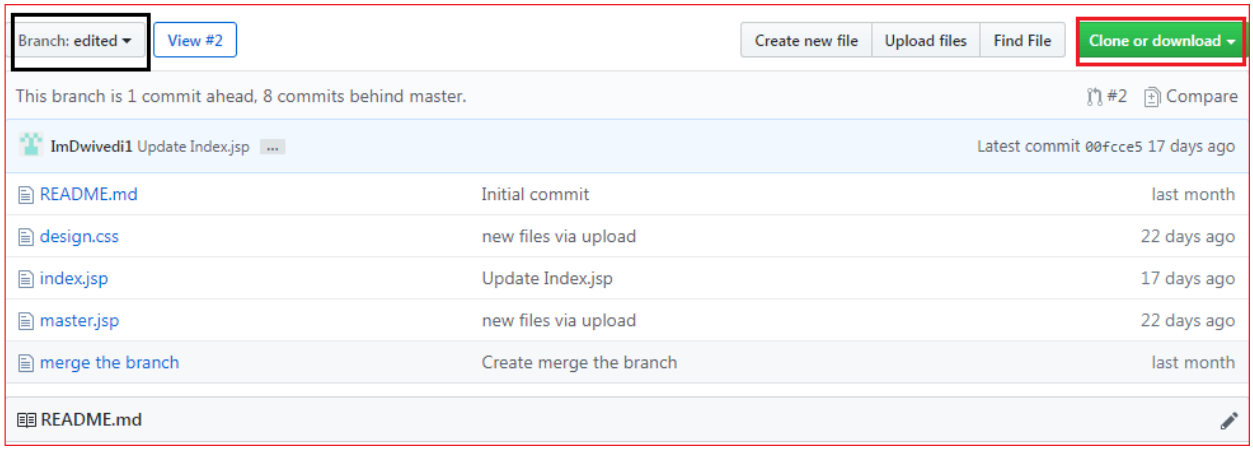


Name	Date modified	Type	Size
.git	10/1/2019 2:47 PM	File folder	
newfolder3	9/21/2019 11:37 AM	File folder	
design	9/19/2019 6:10 PM	Cascading Style S...	1 KB
design2	10/1/2019 2:47 PM	Cascading Style S...	1 KB
index	9/19/2019 6:10 PM	JSP File	2 KB
master	9/19/2019 6:10 PM	JSP File	1 KB
merge the branch	9/20/2019 6:05 PM	File	1 KB
newfile	9/28/2019 12:56 PM	Text Document	1 KB
newfile1	9/28/2019 4:01 PM	Text Document	1 KB
README	9/19/2019 6:10 PM	MD File	1 KB

As you can see in the above output, the design2.css file is added to the local repository. The git pull command is equivalent to **git fetch origin head** and **git merge head**. The head is referred to as the ref of the current branch.

Git Pull Remote Branch

Git allows fetching a particular branch. Fetching a remote branch is a similar process, as mentioned above, in **git pull command**. The only difference is we have to copy the URL of the particular branch we want to pull. To do so, we will select a specific branch. See the below image:

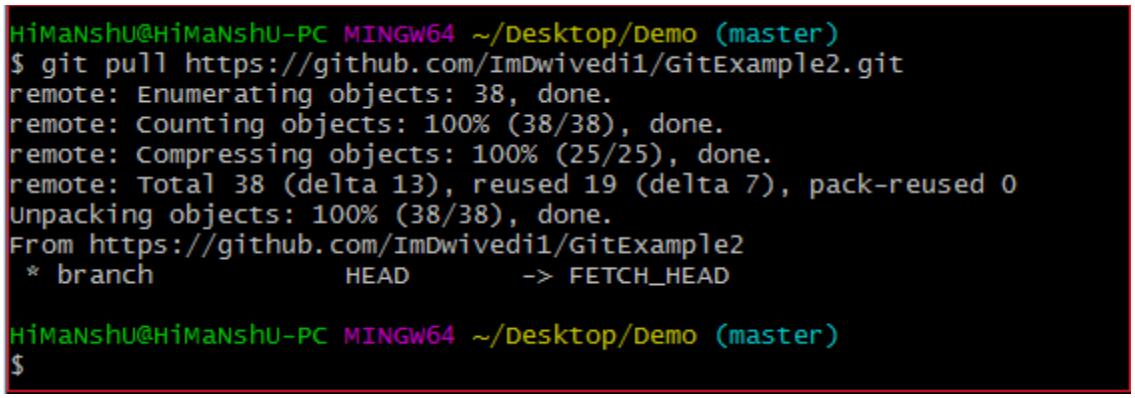


In the above screenshot, I have chosen my branch named **edited** to copy the URL of the edited branch. Now, I am going to pull the data from the edited branch. Below command is used to pull a remote branch:

Syntax:

1. $ git pull **<remote** branch URL**>**

Output:



In the above output, the remote branch **edited** has copied.

Git Force Pull

Git force pull allows for pulling your repository at any cost. Suppose the below scenario:

If you have updated any file locally and other team members updated it on the remote. So, when will you fetch the repository, it may create a conflict.

We can say **force pull** is used for overwriting the files. If we want to discard all the changes in the local repository, then we can overwrite it by influentially pulling it. Consider the below process to force pull a repository:

Step1: Use the git fetch command to download the latest updates from the remote without merging or rebasing.

1. $ git fetch -all

Step2: Use the git reset command to reset the master branch with updates that you fetched from remote. The hard option is used to forcefully change all the files in the local repository with a remote repository.

1. $ git reset -hard **<remote>**/**<branch_name>**
2. $ git reset-hard master

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$ git fetch --all
Fetching origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/GitExample2
   0a1a475..828b962  master     -> origin/master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$ git reset --hard master
HEAD is now at 0a1a475 CSS file
```

In the above output, I have updated my design2.css file and forcefully pull it into the repository.

Git Pull Origin Master

There is another way to pull the repository. We can pull the repository by using the **git pull** command. The syntax is given below:

1. $ git pull **<options><remote>/<branchname>**
2. $ git pull origin master

In the above syntax, the term **origin** stands for the repository location where the remote repository situated. **Master** is considered as the main branch of the project.

Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$ git pull origin master
From https://github.com/ImDwivedi1/GitExample2
 * branch            master     -> FETCH_HEAD
Updating 0a1a475..828b962
Fast-forward
 design2.css | 1 +
 1 file changed, 1 insertion(+)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Demo/GitExample2 (master)
$
```

It will overwrite the existing data of the local repository with a remote repository.

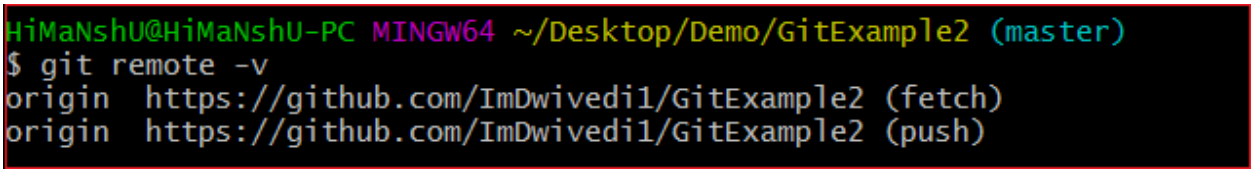
You can check the remote location of your repository. To check the remote location of the repository, use the below command:

1. $ git remote -v

The given command will result in a remote location like this:

1. origin https://github.com/ImDwivedi1/GitExample2 (fetch)
2. origin https://github.com/ImDwivedi1/GitExample2 (push)

The output displays fetch and push both locations. Consider the below image:



Git Pull Request

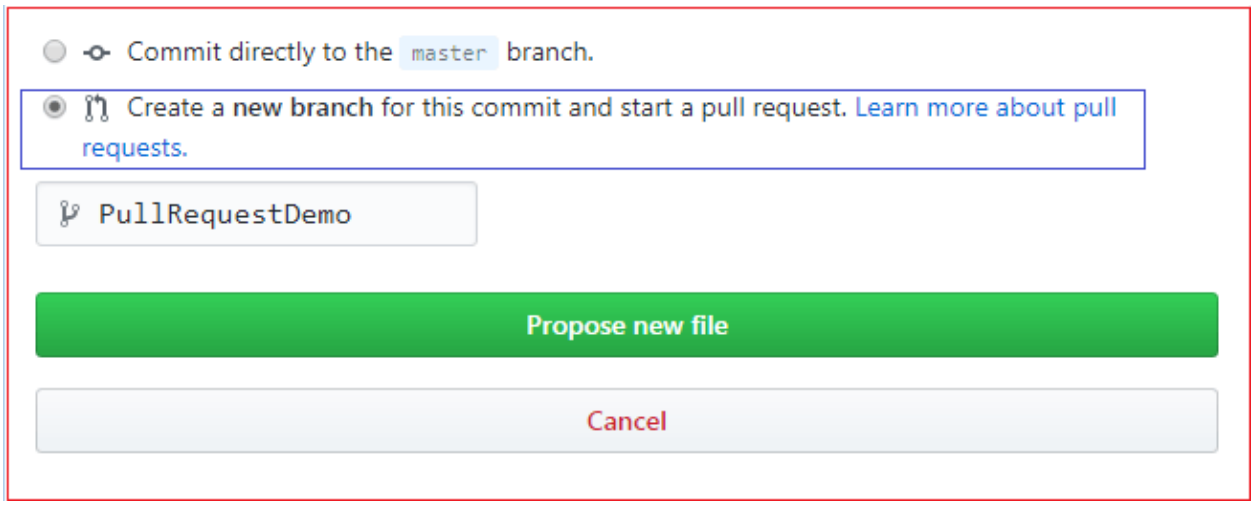
Pull request allows you to announce a change made by you in the branch. Once a pull request is opened, you are allowed to converse and review the changes made by others. It allows reviewing commits before merging into the main branch.

Pull request is created when you committed a change in the GitHub project, and you want it to be reviewed by other members. You can commit the changes into a new branch or an existing branch.

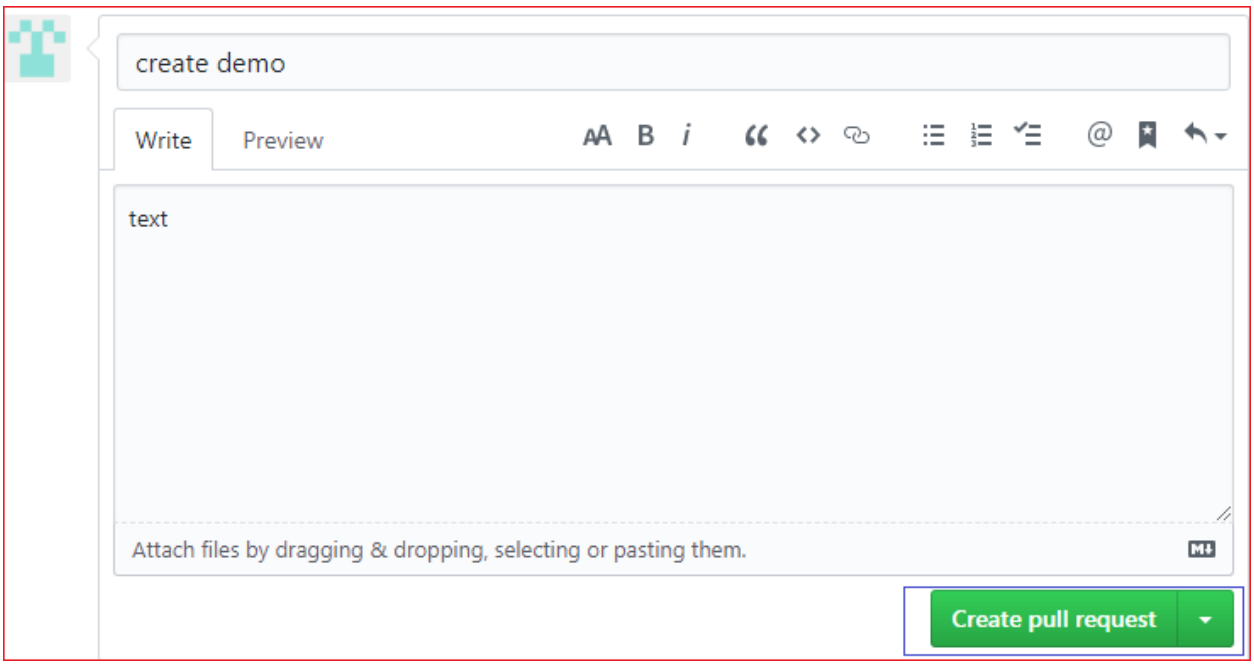
Once you've created a pull request, you can push commits from your branch to add them to your existing pull request.

How to Create a Pull Request

To create a pull request, you need to create a file and commit it as a new branch. As we mentioned earlier in this topic, how to commit a file to use git pull. Select the option "**create a new branch for this commit and start a pull request**" from the bottom of the page. Give the name of the new branch. Select the option to **propose a new file** at the bottom of the page. Consider the below image.



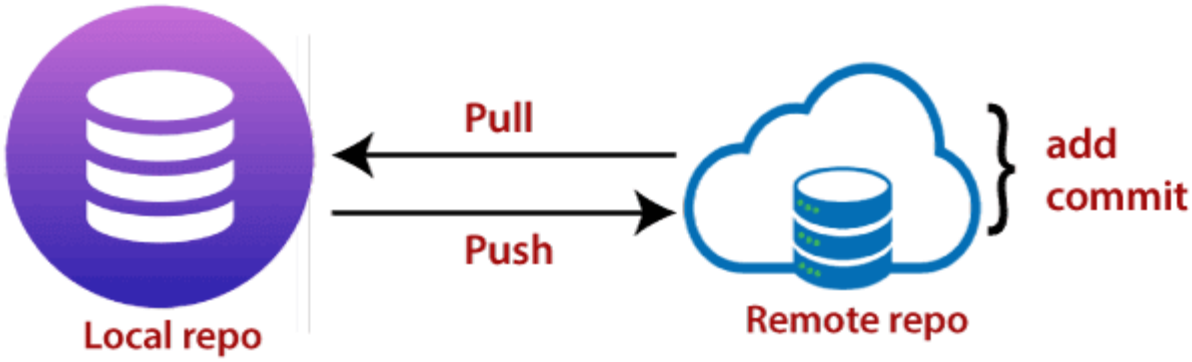
In the above image, I have selected the required option and named the file as **PullRequestDemo**. Select the option to propose a new file. It will open a new page. Select the option **create pull request**. Consider the below image:



Now, the pull request is created by you. People can see this request. They can merge this request with the other branches by selecting a merged pull request.

Git 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.



Moreover, we can say the push updates the remote refs with local refs. Every time you push into the repository, it is updated with some interesting changes that you made. If we do not specify the location of a repository, then it will push to default location at **origin master**.

The "git push" command is used to push into the repository. The push command can be considered as a tool to transfer commits between local and remote repositories. The basic syntax is given below:

1. $ git push <option> [<Remote URL><branch name><refspec>...]

Push command supports many additional options. Some options are as follows under push tags.

Git Push Tags

<repository>: The repository is the destination of a push operation. It can be either a URL or the name of a remote repository.

<refspec>: It specifies the destination ref to update source object.

--all: The word "all" stands for all branches. It pushes all branches.

--prune: It removes the remote branches that do not have a local counterpart. Means, if you have a remote branch say demo, if this branch does not exist locally, then it will be removed.

--mirror: It is used to mirror the repository to the remote. Updated or Newly created local refs will be pushed to the remote end. It can be force updated on the remote end. The deleted refs will be removed from the remote end.

--dry-run: Dry run tests the commands. It does all this except originally update the repository.

--tags: It pushes all local tags.

--delete: It deletes the specified branch.

-u: It creates an upstream tracking connection. It is very useful if you are going to push the branch for the first time.

Git Push Origin Master

Git push origin master is a special command-line utility that specifies the remote branch and directory. When you have multiple branches and directory, then this command assists you in determining your main branch and repository.

Generally, the term **origin stands** for the remote repository, and master is considered as the main branch. So, the entire statement "**git push origin master**" pushed the local content on the master branch of the remote location.

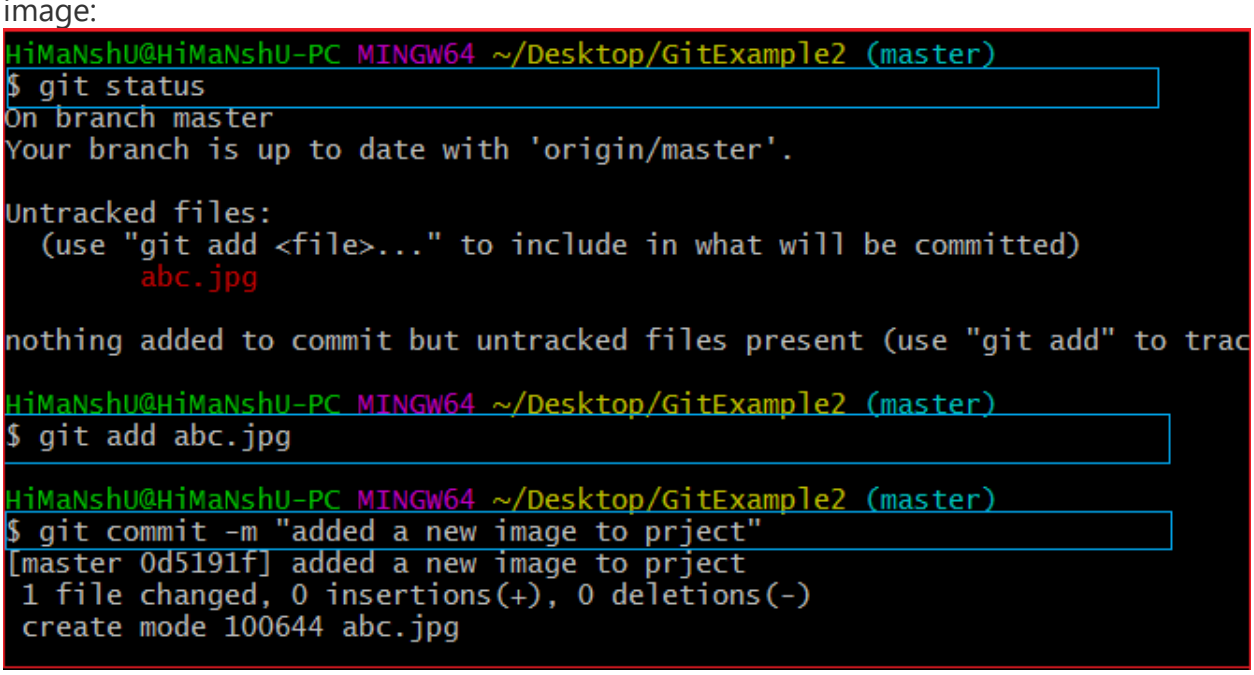
Syntax:

1. $ git push origin master

Let's understand this statement with an example.

Let's make a new commit to my existing repository, say **GitExample2**. I have added an image to my local repository named **abc.jpg** and committed the changes. Consider the below

image:



In the above output, I have attached a picture to my local repository. The git status command is used to check the status of the repository. The git status command will be performed as follows:

1. $ git status

It shows the status of the untracked image **abc.jpg**. Now, add the image and commit the changes as:

1. $ git add abc.jpg
2. $git commit -m "added a new image to project."

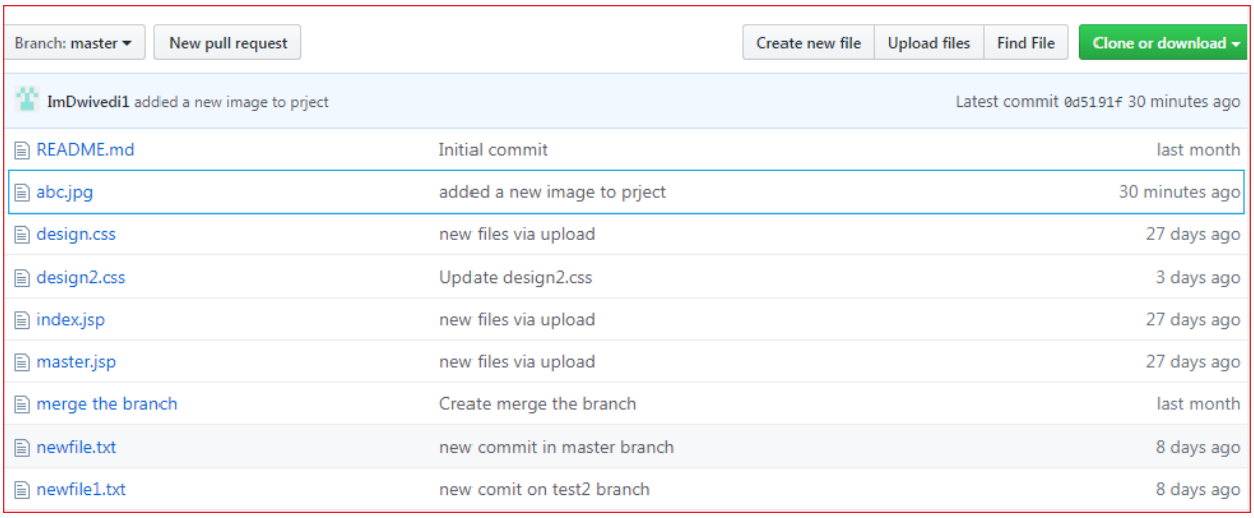
The image is wholly tracked in the local repository. Now, we can push it to origin master as:

1. $ git push origin master

Output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 757.29 KiB | 6.95 MiB/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/ImDwivedi1/GitExample2.git
   828b962..0d5191f  master -> master
```

The file **abc.jpg** is successfully pushed to the origin master. We can track it on the remote location. I have pushed these changes to my GitHub account. I can track it there in my repository. Consider the below image:



In the above output, the pushed file abc.jpg is uploaded on my GitHub account's master branch repository.

Git Force Push

The git force push allows you to push local repository to remote without dealing with conflicts. It is used as follows:

1. $ git push <remote><branch> -f

Or

1. $ git push <remote><branch> -force

The -f version is used as an abbreviation of force. The remote can be any remote location like GitHub, Subversion, or any other git service, and the branch is a particular branch name. For example, we can use git push origin master -f.

We can also omit the branch in this command. The command will be executed as:

1. $git push <remote> -f

We can omit both the remote and branch. When the remote and the branch both are omitted, the default behavior is determined by **push.default** setting of git config. The command will be executed as:

1. $ git push -f

How to Safe Force Push Repository:

There are several consequences of force pushing a repository like it may replace the work you want to keep. Force pushing with a lease option is capable of making fail to push if there are new commits on the remote that you didn't expect. If we say in terms of git, then we can say it will make it fail if remote contains untracked commit. It can be executed as:

1. $git push <remote><branch> --force-with-lease

Git push -v/--verbose

The -v stands for verbosely. It runs command verbosely. It pushed the repository and gave a detailed explanation about objects. Suppose we have added a **newfile2.txt** in our local repository and commit it. Now, when we push it on remote, it will give more description than the default git push. Syntax of push verbosely is given below:

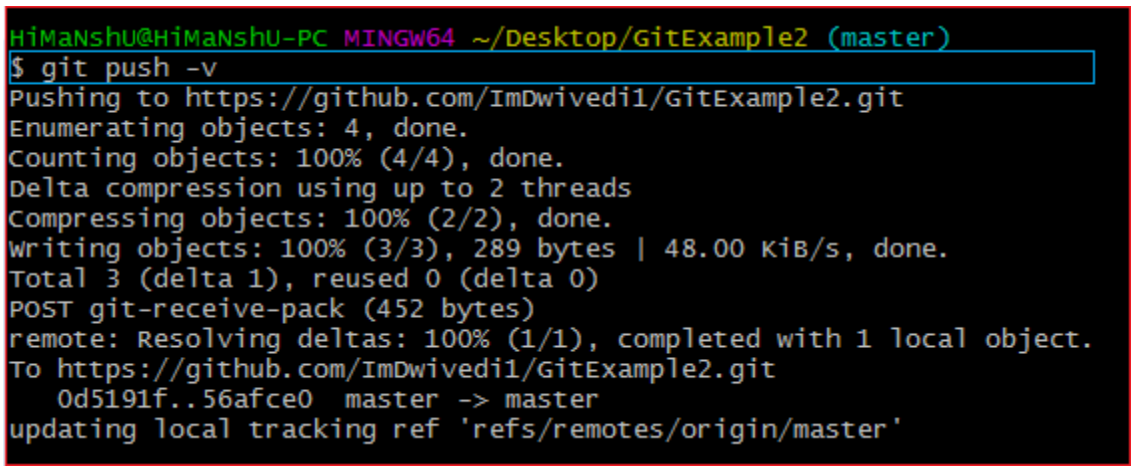
Syntax:

1. $ git push -v

Or

1. $ git push --verbose

Consider the below output:



If we compare the above output with the default git option, we can see that git verbose gives descriptive output.

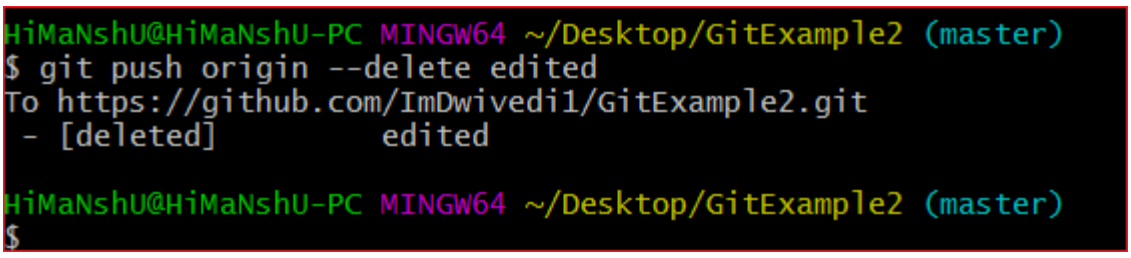
Delete a Remote Branch

We can delete a remote branch using git push. It allows removing a remote branch from the command line. To delete a remote branch, perform below command:

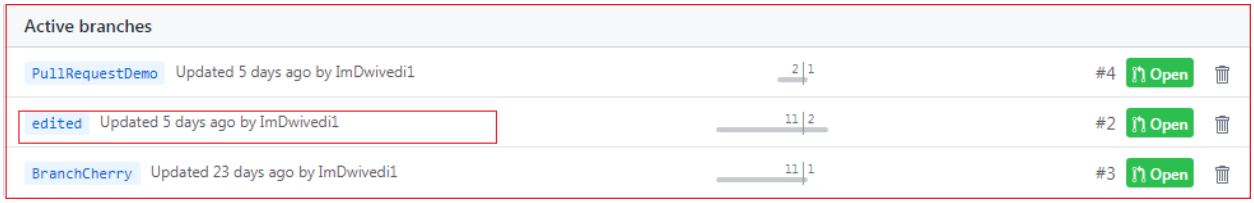
Syntax:

1. $ git push origin -delete edited

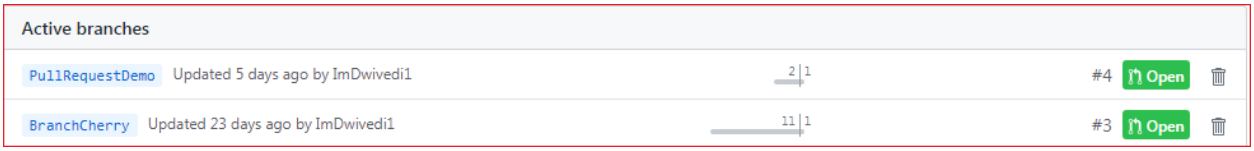
Output:



In the above output, the git push origin command is used with -delete option to delete a remote branch. I have deleted my remote branch **edited** from the repository. Consider the below image:



It is a list of active branches of my remote repository before the operating command.



The above image displays the list of active branches after deleting command. Here, you can see that the branch **edited** has removed from the repository.