

## Integrating Source Code with Version Control

### Describing Version Control

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

### Software Version Control

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

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

## Defining the Role of Version Control in Continuous Integration

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

## Introducing Git

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

## Distributed vs Centralized VCS

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

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

### **Benefits of a decentralized VCS**

- Works offline
- Create local backups
- Flexible workflows
- Ad hoc code sharing
- More granular permissions

### **Drawbacks of a decentralized VCS**

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

### **What is a commit?**

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

#### **Contents of a Commit**

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

### **Managing Git Repositories**

1. The primary interface for Git is the provided command-line interface (CLI).
2. This CLI includes various "subcommands", such as *add*, *init*, and *commit*.

3. Git provides a man page for each of these subcommands. For example, you can view the man page for the add subcommand by running: **git help add**  
**git --help**

### **Obtaining a Git Repository**

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

### **Staging Changes**

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

### **Creating a Commit**

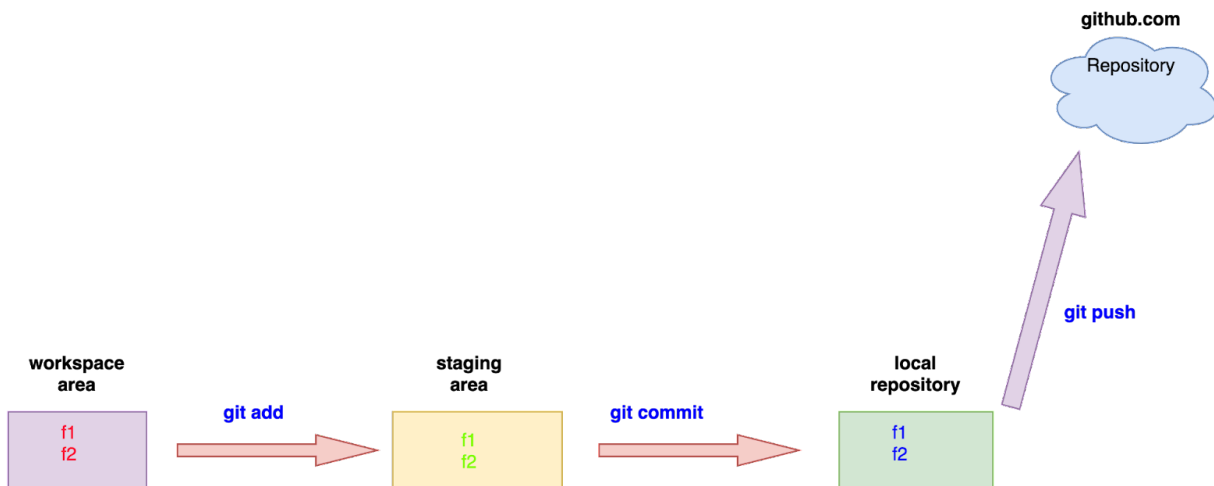
1. Once you have staged changes, you can use **git commit** to create a commit.
2. With your changes committed, you are free to continue making changes knowing that you can easily restore to earlier versions.

### **As you use Git, you adopt the following workflow:**

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

### **In GIT, there are 3 phases -**

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



## Building Applications with Git

1. Configure your git identity at the user level in your system

```
git config --global user.name "Yourname"
```

```
git config --global user.email "Your MailId"
```

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

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

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

```
mkdir git-demo
```

```
cd git-demo
```

```
git init
```

OR

We can clone a remote Git repository from github.com

```
git clone <URL_Of_The_Remote_Repository>
```

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

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

```
cat hello.php
```

```
touch f1 f2
```

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

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

`git add .` ⇒ will stage all untracked files

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

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

**`git commit`** ⇒ will record changes to the repository

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

`git status`

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

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

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

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

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

### Create Personal Access Token on GitHub

From your GitHub account, go to **Settings** ⇒ **Developer Settings** ⇒ **Personal Access Token** ⇒ **Generate New Token** (Give your password) ⇒ **Fillup the form** ⇒ click **Generate token** ⇒ **Copy the generated Token**, it will be something like  
*ghp\_sFhFsSHhTzMDreGRLjmks4Tzuzgthdvfsrta*

`git push origin main` ⇒ It will ask for password [ paste the PAT here ]

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

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

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

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

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

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

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

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

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

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

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

### **Creating Git Branches**

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

### **Establishing a Main Branch**

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

## Defining Branches

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



Commits in a branch not yet ready for the main branch

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

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

## How Git Stores Branches

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

## Git Branch Commands

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

## Merging

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



3. Implicitly, your currently checked out branch is the branch that is merged into.

**Merge Types :**

1. Fast Forward Merge ⇒ Does not create a Commit ID. It uses previous latest ID of a particular branch to do a merge
2. Recursive Merge ⇒ Creates a new commit ID.

**Merge Conflict :** A merge conflict happens when 2 developers [ 2 branches ] modify the same region of a file and are subsequently merged.

There are 3 ways to resolve the merge conflict –

1. Merge
2. Manual
3. Automatic

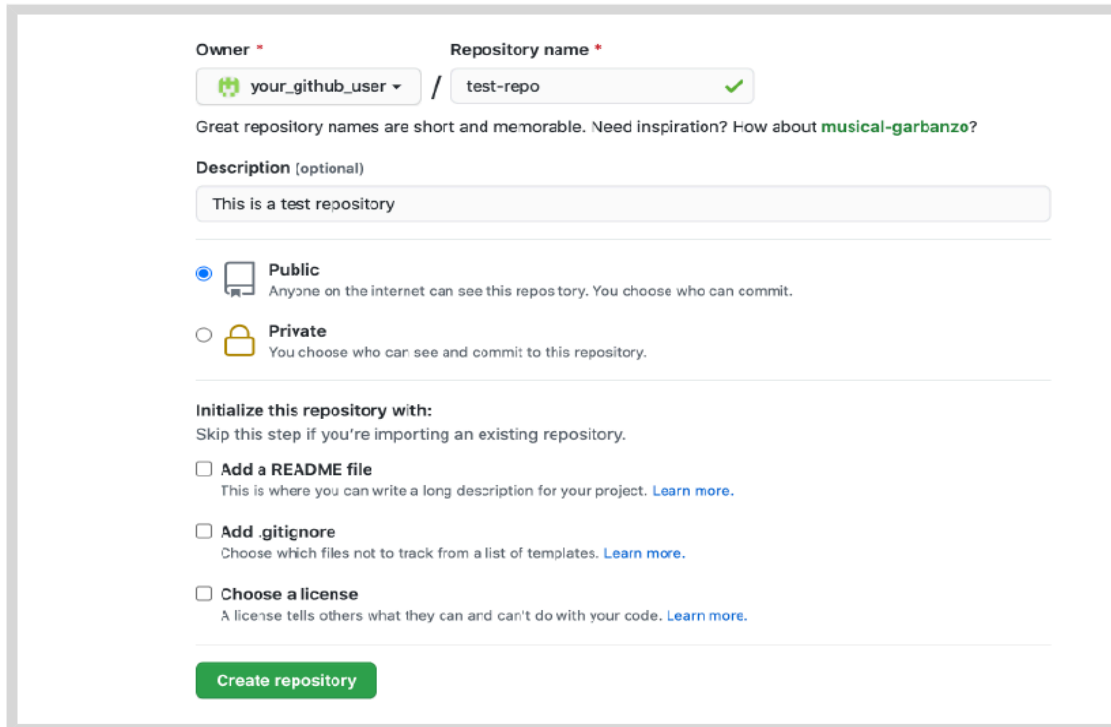
**Managing Remote Repositories**

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

**Working with Remotes**

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

## Creating a New GitHub Repository



The screenshot shows the GitHub 'Create new repository' form. At the top, there are two input fields: 'Owner' with a dropdown menu showing 'your\_github\_user' and 'Repository name' with a text input 'test-repo' and a green checkmark. Below these is a hint: 'Great repository names are short and memorable. Need inspiration? How about **musical-garbanzo**?'. The 'Description (optional)' field contains the text 'This is a test repository'. Under the 'Visibility' section, the 'Public' option is selected with a radio button, and the 'Private' option is unselected. Below this, the 'Initialize this repository with:' section has three checkboxes: 'Add a README file', 'Add .gitignore', and 'Choose a license', all of which are currently unchecked. At the bottom is a green 'Create repository' button.

**GitHub new repository form**

### **Note**

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

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

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

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

**.gitignore** ⇒ lists files that we want git to ignore in the working directory.

## **Cloning a Repository**

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

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

## **Remote Branches**

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

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

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

## **Adding a Remote to a Local Repository**

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

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

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

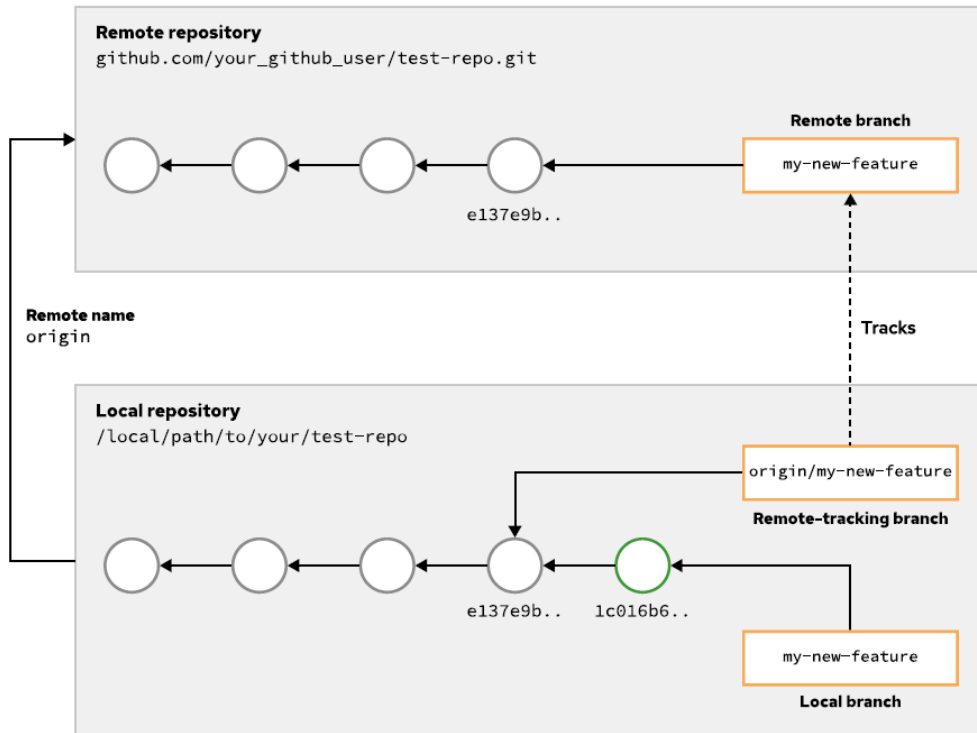
To add a new remote, use the `git remote add` command, followed by a name and the URL of the remote repository.

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

### **Pushing to the Remote**

When you create new commits in your local branch, the remote-tracking branch and the remote

branch become outdated with respect to the local branch, as shown in the following figure:



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

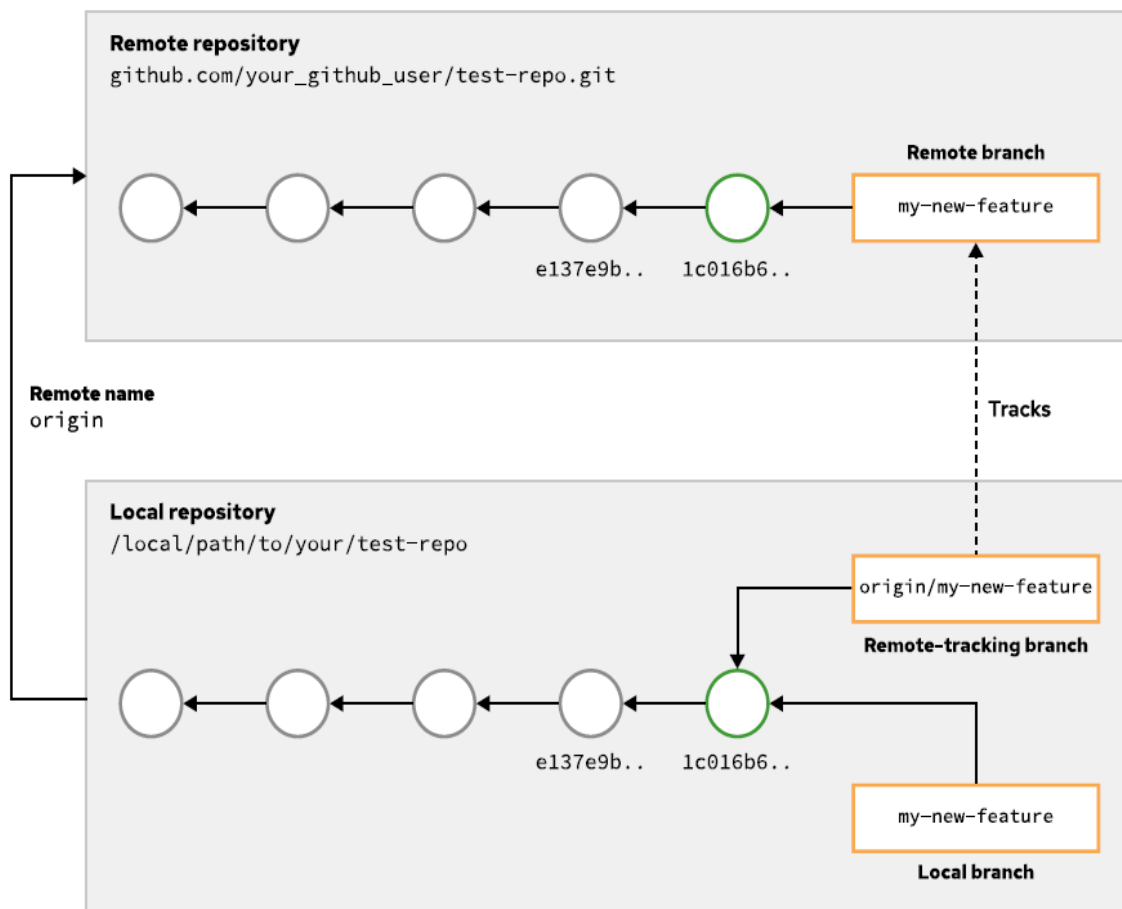
The figure shows how the new `1c016b6..` commit is in the local branch only. To update the remote branch with your new commit, you must push your changes to the remote.

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

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

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

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



**The remote branch is up to date with the local branch**

Lab :

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

## Pull Requests

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

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

Pull requests are a powerful web-based feature

Pull requests help achieve the following:

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

**Stash Memory** ⇒ is a temporary memory provided by git

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

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

To bring back the files from stash memory into the staging area ⇒ ***git stash pop*** [ It also deallocates the stash memory ]

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

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

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

## Defining Development Workflows

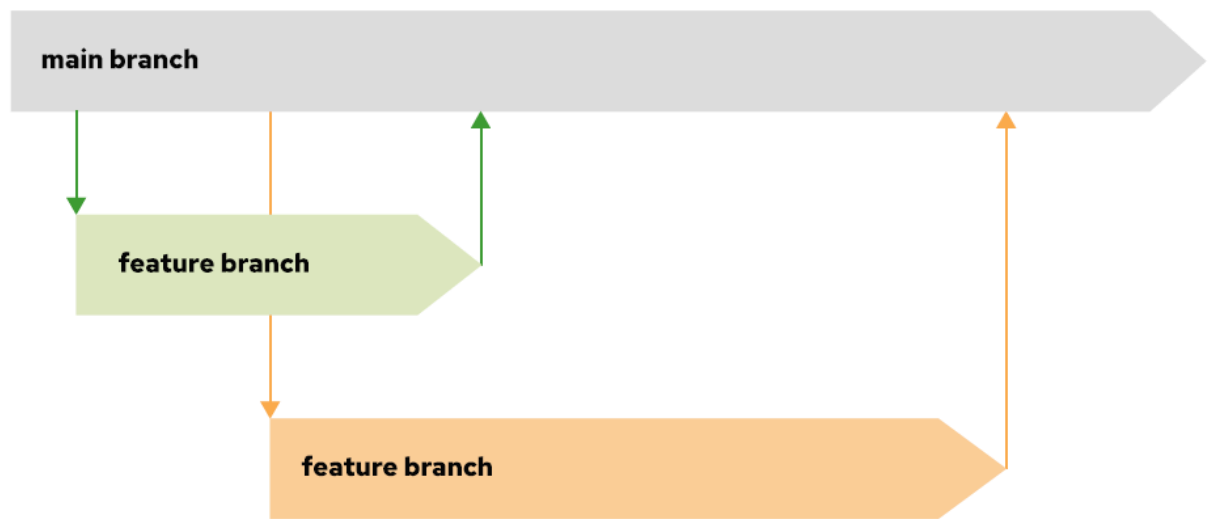
- ❖ Developers love finding reusable solutions to problems.
- ❖ We can follow one of the many workflows available to guide the development process.
- ❖ A workflow consists of a defined set of rules and processes, which guide your development process.
- ❖ Most of the popular development workflows share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

## Trunk-based Development

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

### Feature Branching

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

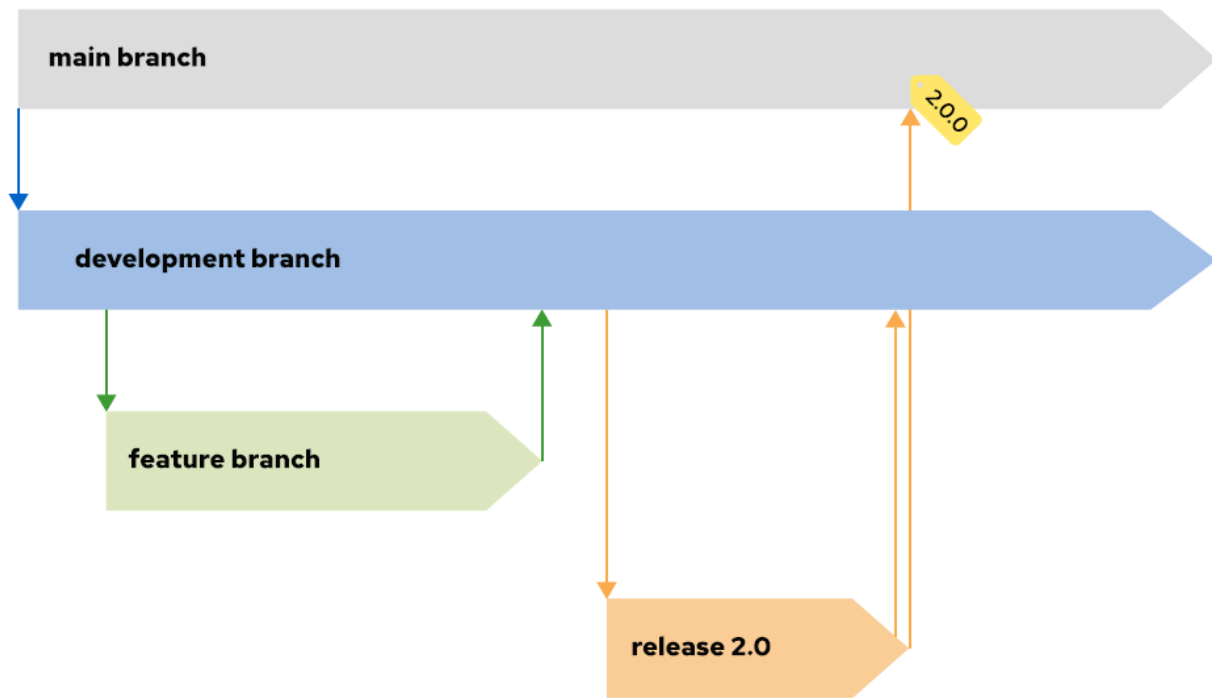


**Branching structure in Feature Branching**

### Git Flow

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





**Branching structure in Git Flow**

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

### Releasing Code

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

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

### Creating Releases with Git

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

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

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

```
git tag -a v1.0
```

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

- To list all tags ⇒ `git tag`
- To create a tag ⇒ `git tag v1.0`
- To delete a tag ⇒ `git tag -d v1.0`
- To push a tag ⇒ `git push origin v1.0`
- After you have your code with a tag assigned, you can distribute it to your users.
- *When we delete a tag from a local workstation, it does not delete the tag from the GitHub repository.* If we want to delete the tags from the GitHub repo also, we need to give the following command :

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

### Lab : Releasing code

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

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

With the organization set up as a hypothetical third-party project, you will practice a

common contribution workflow, forking the third-party project into your username account and creating pull requests back to the third-party project to propose code changes.

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

=====

### What is a GitLab Server?

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

### Differences between GitHub and GitLab

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

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

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

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