**What is Version Control and Git?**

**Version Control**

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting.

Benefits

1. A complete long-term change history of every file
2. Branching and merging.
3. Traceability.

**Git history**

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds for development of the Linux Kernel.

Git focuses on the file content itself.

Git is not GitHub. Git is the version control software, and GitHub is a git repository hosting service which offers all the source code management provided in git. GitHub is where you upload your git repository.

**Usage of Git**

Real life projects generally have multiple developers working in parallel. So, a version control system like Git is needed to ensure there are no code conflicts between the developers.

Additionally, the requirements in such projects change often. So, a version control system allows developers to revert and go back to an older version of the code.

Finally, sometimes several projects which are being run in parallel involve the same codebase. In such a case, the concept of branching in Git is very important.

**Git setup and configuration**

1. Download git from [https://gitforwindows.org](https://gitforwindows.org/)
2. Open Git Bash
3. Check git version:

git --version

1. Configure your Git username and email:

git config --global user.name "Javid Salmanov"

git config --global user.email "javid.salmanov@accenture.com"

**Git basics**

Creating Git repositories

A Git repository is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

To create a new repo, you'll use the git init command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new master branch.

1. Create a folder in a Desktop for your project with a name “bootcamp” (mkdir ~/Desktop/bootcamp)
2. Open Git Bash
3. Change directory to “bootcamp”
4. Run a command git init

Adding files to a Git repository

1. Change directory to “bootcamp” directory
2. Create a file called devops.txt in the project folder
3. Add the following text into it:

“DevOps is Hero!”

(Here we will be demoing with just plain text instead of actual code, our focus is on Git and not on any specific programming language 😊)

1. Run git status
2. Run the command git add (or git add devops.txt)

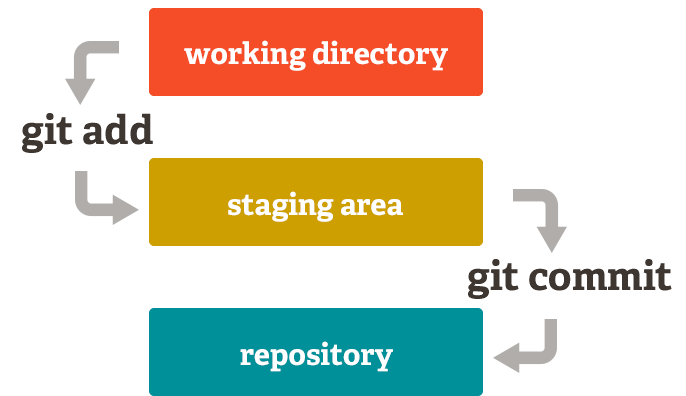
The git add command adds a change in the working directory to the staging area.

git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

1. Run the following command to add your change to repository:

git commit -m "Add devops.txt"

Internal structure of a Git repository



First, you edit your files in the working directory. When you’re ready to save a copy of the current state of the project, you stage changes with git add. After you’re happy with the staged snapshot, you commit it to the project history with git commit.

Remote repositories

When time comes to **sharing data** with your teammates, a remote repo comes into play. Think of it like a "file server" that you use to exchange data with your colleagues.

Location

Local repositories reside on the computers of team members. In contrast, remote repositories are hosted on a server that is accessible for all team members - most likely on the internet or on a local network.

Features

Technically, a remote repository doesn't differ from a local one: it contains branches, commits just like a local repository. However, a local repository has a working copy associated with it: a directory where some version of your project's files is checked out for you to work with.  
A remote repository doesn't have such a working directory: it only consists of the bare ".git" repository folder.

Usage

It's important to stress that the actual *work* on your project happens *only* in your local repository: all modifications must be made & committed locally.  
Then, those changes *can* be uploaded to a remote repository in order to share them with your team. Remote repositories are only thought as a means for sharing and exchanging code between developers - not for working on files.

Create:

git remote add origin https://github.com/JavidSalmanov/bootcamp.git

And push:

git push -u origin master

**Git history**

Accessing different versions of the files

You can get all git history by :

git log (--oneline --decorate –graph)

To particular version of a particular file:

git show COMMITSHA:path/to/file

To be able to get all files particular version :

git checkout COMMITSHA

Temporary saving changes with git stash

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

To add files stash run :

git stash

To provide a bit more context, it's good practice to annotate your stashes with a description, using git stash save "message":

git stash save "Add new line to devops.txt file"

You can view a summary of a stash with:

git stash show -p

To return all files from stash:

git stash pop

By default, running git stash will stash:

* changes that have been added to your index (staged changes)
* changes made to files that are currently tracked by Git (unstaged changes)

But it will **not** stash:

* new files in your working copy that have not yet been staged
* files that have been ignored

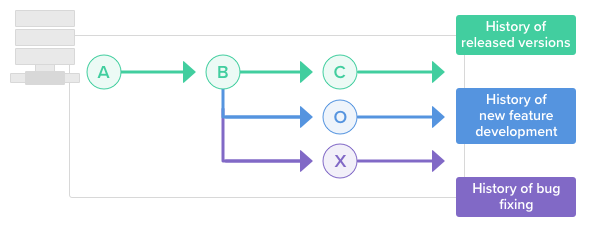
**Branches**

Local and Remote branches

Local

In a collaborative environment, it is common for several developers to share and work on the same source code. While some developers will be fixing bugs, others will be implementing new features, etc. With so much going on, there needs to be a system in place for managing different versions of the same code base. Branching allows each developer to branch out from the original code base and isolate their work from others. It also helps Git to easily merge versions later.

A Git branch is essentially an independent line of development. You can take advantage of branching when working on new features or bug fixes because it isolates your work from that of other team members.



Creating a new branch does not change the repository; it simply points out the commit.

To create new branch:

git branch dev1

or

git checkout -b dev1

git checkout -b dev1

To switch branch:

git checkout dev1

Remote

When you clone a repository from a remote server, Git automatically remembers this connection for you. It saves it as a remote called "origin" by default.

In other cases where you started with a fresh local repository, no remote connections are saved. In that situation, we need to connect our local repository to a new remote before we can try some remote interactions:

git remote add origin https://github.com/JavidSalmanov/bootcamp.git

To check the remote repo:

git remote -v

To see the all branches:

git branch -va

Once a remote record has been configured using the git remote command, the remote name can be passed as an argument to other Git commands to communicate with the remote repo. Both git fetch, and git pull can be used to read from a remote repository.

Fetch

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Git isolates fetched content as a from existing local content, it has absolutely no effect on your local development work.

git fetch

git diff --stat --color master origin

Pull

git pull does a git fetch followed by a git merge

Push

git push is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.

git push origin dev1

Pull request

The general process of pull request is as follows:

1. A developer creates the feature in a dedicated branch in their local repo.
2. The developer pushes the branch to a public repository.
3. The developer files a pull request.
4. The rest of the team reviews the code, discusses it, and alters it.
5. The project maintainer merges the feature into the official repository and closes the pull request.

It’s also possible to file a pull request for a feature that is incomplete. For example, if a developer is having trouble implementing a requirement, they can file a pull request containing their work-in-progress. Other developers can then provide suggestions inside of the pull request, or even fix the problem themselves with additional commits.

**Merging changes**

Merging changes with the merge command

git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches.

Execute git status to ensure that HEAD is pointing to the correct merge-receiving branch. If needed, execute git checkout <receiving> to switch to the receiving branch. In our case we will execute git checkout master.

Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes. Execute git fetch to pull the latest remote commits. Once the fetch is completed ensure the master branch has the latest updates by executing git pull.

Once the previously discussed "preparing to merge" steps have been taken a merge can be initiated by executing git merge <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch.

git checkout master

git merge dev1

Fast forward

If Master has not diverged, instead of creating a new commit, git will just point master to the latest commit of the feature branch. This is a “fast forward.” There won't be any "merge commit" in fast-forwarding merge.

Rebase

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

The major benefit of rebasing is that you get a much cleaner project history.

git checkout master

git rebase dev1

**Change analysis**

Differences between two commits

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more.

To get all changes with content:

git diff SHA SHA

To get only changed file list:

git diff SHA --name-only

**Best practices**

*Commit often*

Commits are cheap and easy to make. They should be made frequently to capture updates to a code base. Each commit is a snapshot that the codebase can be reverted to if needed. Frequent commits give many opportunities to revert or undo work. A group of commits can be combined into a single commit using a rebase to clarify the development log.

*Ensure you're working from latest version*

SCM enables rapid updates from multiple developers. It’s easy to have a local copy of the codebase fall behind the global copy. Make sure to git pull or fetchthe latest code before making updates. This will help avoid conflicts at merge time.

*Make detailed notes*

Each commit has a corresponding log entry. At the time of commit creation, this log entry is populated with a message. It is important to leave descriptive explanatory commit log messages. These commit log messages should explain the “why” and “what” that encompass the commits content. These log messages become the canonical history of the project’s development and leave a trail for future contributors to review.

*Review changes before committing*

SCM’s offer a ‘staging area’. The staging area can be used to collect a group of edits before writing them to a commit. The staging area can be used to manage and review changes before creating the commit snapshot. Utilizing the staging area in this manner provides a buffer area to help refine the contents of the commit.

*Use Branches*

Branching is a powerful SCM mechanism that allows developers to create a separate line of development. Branches should be used frequently as they are quick and inexpensive. Branches enable multiple developers to work in parallel on separate lines of development. These lines of development are generally different product features. When development is complete on a branch it is then merged into the master line of development.

*Agree on a Workflow*

By default, SCMs offer very free form methods of contribution. It is important that teams establish shared patterns of collaboration. SCM workflows establish patterns and processes for merging branches. If a team doesn't agree on a shared workflow it can lead to inefficient communication overhead when it comes time to merge branches.