Agenda: Azure Repos and Git

- Version control using Git
- What is Git, Azure Repos and GitHub
- Install Git Locally
- Getting Started with Git Commands
- Updating to DevOps Repository
- Working with Branches
- Merging Branches
- Creating and Committing a Pull Request
- Add a rule to Require a Review
- Squash Merging during Pull Request.
- Working with Merge Conflicts
- Cherry-Picking and Rebase
- Undo Changes using Reset and Revert
- Ignoring files using gitignore
- Managing Git Branches in Azure Repos
- Branch Policies and Branch Permissions
- Branches in Folders
- Working the GitHub Repositories
- Branching Workflow Types
  - Feature Branching
  - Gitflow Branching
  - Forking Workflow
- Summary of Git Commands

Version Control using Git

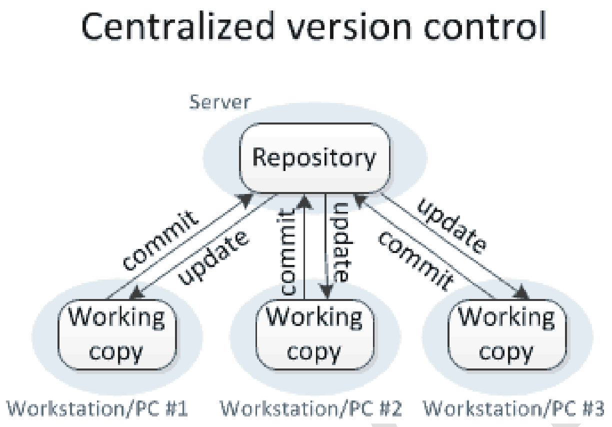
DevOps is a revolutionary way to release **software quickly and efficiently** while maintaining a high level of security. Source control (version control) is a critical part of DevOps

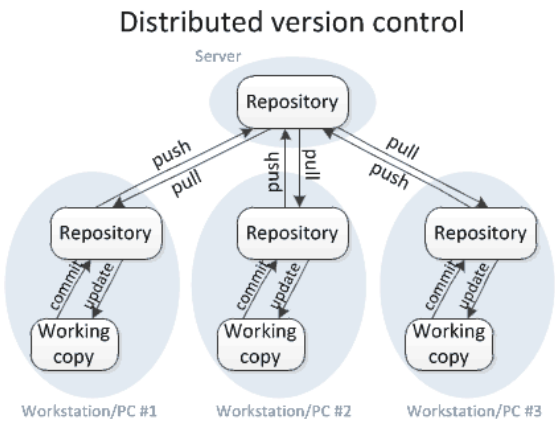
- Version control systems are software that help you track changes you make in your code over time. As you edit your code, you tell the version control system to take a **snapshot** of your files. The version control system saves that snapshot permanently so you can recall it later if you need it.
- Use version control to save your work and coordinate code **changes across your team**.

- Even if you're just a single developer, version control helps you stay organized as you fix bugs and develop new features. Version control keeps a history of your development so that you can review and even rollback to any version of your code with ease.

Benefits of Source Control

- Create workflows
- Work with versions
- Collaboration between developers
- Maintains history of changes
- Automate tasks





Centralized Version Control:

- There is a single central copy of your project and programmers commit their changes to this central copy.
- Common centralized version control systems are TFVC, CVS, Subversion (or SVN) and Perforce.

Strengths	Best Used for
• Easily scales for very large codebases • Granular permission control • Permits monitoring of usage • Allows exclusive file locking	• Large integrated codebases • Audit and access control down to the file level • Hard to merge file types

Distributed Version Control:

- Every developer clones a copy of a repository and has the full history of the project
- Common distributed source control systems are Mercurial, Git and Bazaar.

Strengths:	Best Used for

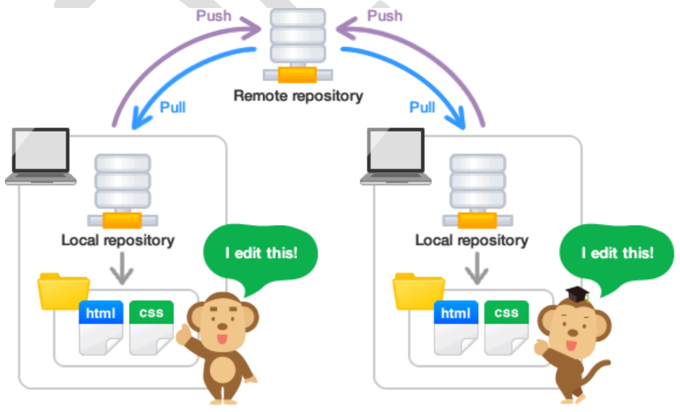
• Cross platform support • An open source friendly code review model via pull requests • Complete offline support • Portable history • An enthusiastic growing user based.	• Small and modular codebases • Evolving through open source • Highly distributed teams • Teams working across platforms

What is Git

- Git is the most commonly used **version control system**, a tool to manage your source code history. It is quickly becoming the standard for version control.
- Git is a **distributed** version control system, meaning that your **local copy** of code is a complete **version control repository**. These fully functional local repositories make it is easy to work offline or remotely. You commit your work locally, and then sync your copy of the repository with the copy on the server.
- You can use the clients and tools of your choice, such as Git for Windows, Mac, partners' Git services, and tools such as Visual Studio and Visual Studio Code.

Git Repository

- A Git repository, or repo, is a folder that you've told Git to help you track file changes in. You can have any number of repos on your computer, each stored in their own folder.
- Each Git repo on your system is independent, so changes saved in one Git repo don't affect the contents of another.
- A Git repo contains every version of every file saved in the repo. Git saves these files very efficiently, so having a large number of versions doesn't mean that it uses a lot of disk space. Storing each version of your files helps Git merge code better and makes working with multiple versions of your code quick and easy



Benefits of Git

- Development with local copy.
- Faster releases because branches allow simultaneous development.
- Strong community support being open source.
- Built-in integration in many IDE. (eg: Visual Studio Code and VS.NET)
- Setup Branch policies to ensure pull requests are reviewed.

Objections to using Git

- I can overwrite history
- I have large files (binaries)
- I have a very large repo
- I don't want to use GitHub (because its by MS)
- There's a steep learning curve

What is GitHub?

GitHub is a hosting service for **Git** repositories. **GitHub** is the service for projects that use **Git**.

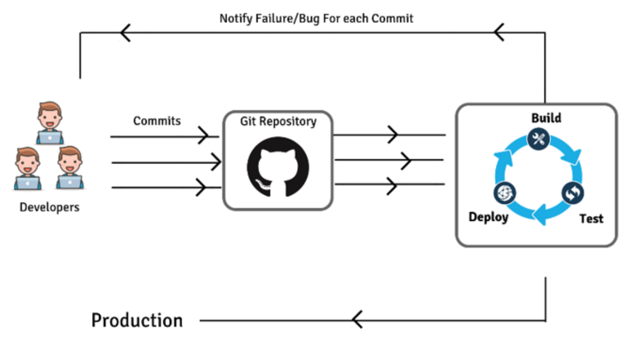
While Git is a command line tool, GitHub provides a Web-based graphical interface.

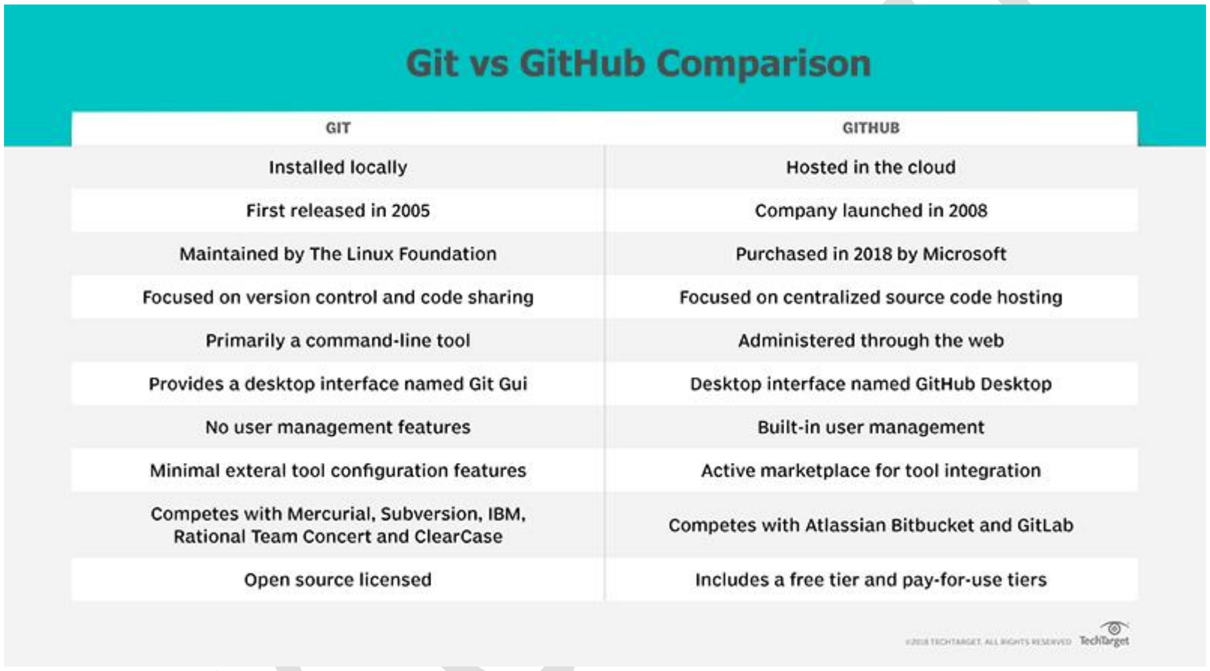
GitHub also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.

GitHub essentially manages:

- Repositories
- Branches
- Commits
- Pull Requests
- Git (the version control software GitHub is built on)

The flagship functionality of GitHub is "**forking**" – copying a repository from one user's account to another. This enables you to take a project that you don't have write access to and modify it under your own account. If you make changes you'd like to share, you can send a notification called a "pull request" to the original owner. That user can then, with a click of a button, merge the changes found in your repo with the original repo.





Git vs GitHub Comparison

GIT	GITHUB
Installed locally	Hosted in the cloud
First released in 2005	Company launched in 2008
Maintained by The Linux Foundation	Purchased in 2018 by Microsoft
Focused on version control and code sharing	Focused on centralized source code hosting
Primarily a command-line tool	Administered through the web
Provides a desktop interface named Git Gui	Desktop interface named GitHub Desktop
No user management features	Built-in user management
Minimal exteral tool configuration features	Active marketplace for tool integration
Competes with Mercurial, Subversion, IBM, Rational Team Concert and ClearCase	Competes with Atlassian Bitbucket and GitLab
Open source licensed	Includes a free tier and pay-for-use tiers

©2018 TECHTARGET. ALL RIGHTS RESERVED TechTarget

Note: You don't *have* to use a remote service like GitHub or Azure Repos if all you want is version control - local git is just fine for that.

Azure Repos:

Azure Repos in Azure DevOps is equivalent to GitHub. It's a set of version control tools that you can use to manage your code.

Azure Repos provides two types of version control:

- **Git**
  - Distributed source control system
  - Each developer has a copy of the source repository on their dev machine
- **TFVC**
  - Centralized source control system

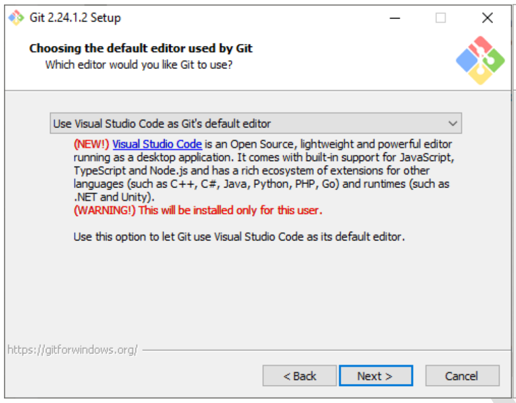
- Team members have **only one version of each file** on their dev machines.
- In the *Server workspaces* model, before making changes, team members publicly check out files.
- In the *Local workspaces* model, each team member takes a copy of the latest version of the codebase with them and works offline as needed.

Download and install Git

Git Command line package (https://git-scm.com/downloads)

Download and run the latest Git for Windows installer, which includes the **Git Credential Manager** for Windows.

Make sure to enable the Git Credential Manager installation option.


Git 2.24.1.2 Setup

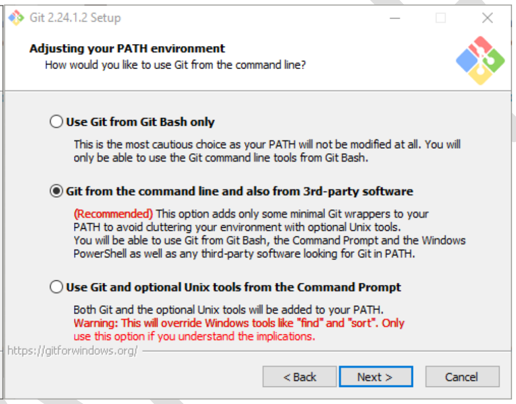
Choosing the default editor used by Git
Which editor would you like Git to use?

Use Visual Studio Code as Git's default editor

(NEW!) Visual Studio Code is an Open Source, lightweight and powerful editor running as a desktop application. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, Go) and runtimes (such as .NET and Unity).
(WARNING!) This will be installed only for this user.

Use this option to let Git use Visual Studio Code as its default editor.

https://gitforwindows.org/


Git 2.24.1.2 Setup

Adjusting your PATH environment
How would you like to use Git from the command line?

[ ] **Use Git from Git Bash only**

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

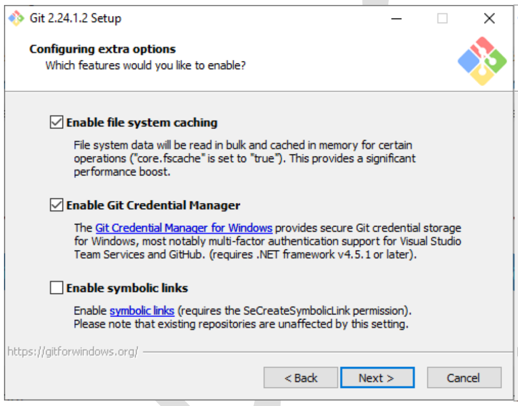
[✓ **Git from the command line and also from 3rd-party software**]

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

[ ] **Use Git and optional Unix tools from the Command Prompt**

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

https://gitforwindows.org/


Git 2.24.1.2 Setup

Configuring extra options
Which features would you like to enable?

[✓] **Enable file system caching**

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

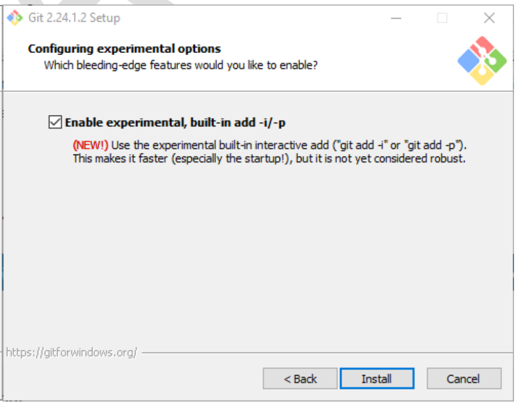
[✓] **Enable Git Credential Manager**

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

[ ] **Enable symbolic links**

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

https://gitforwindows.org/


Git 2.24.1.2 Setup

Configuring experimental options
Which bleeding-edge features would you like to enable?

[✓] **Enable experimental, built-in add -i/-p**

(NEW!) Use the experimental built-in interactive add ("git add -i" or "git add -p"). This makes it faster (especially the startup!), but it is not yet considered robust.

https://gitforwindows.org/

Authenticating to Your Git Repos:

When you first connect to a Git repo (Azure Repo) the **credential manager prompts** for your Microsoft Account or Azure Active Directory credentials. Once authenticated, the credential manager creates and caches a **personal access token** for future connections to the repo.

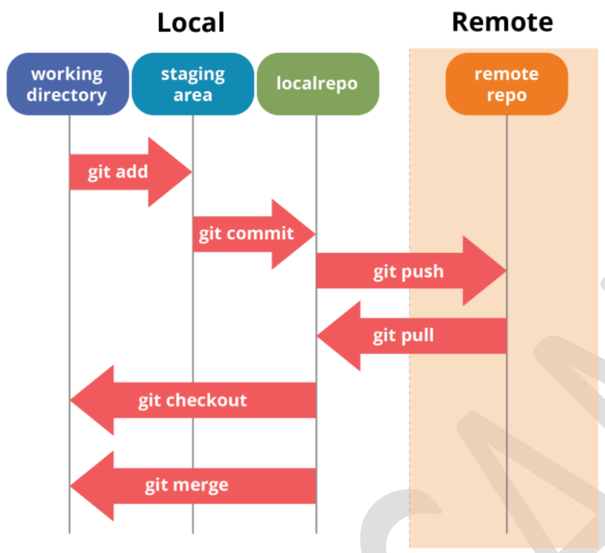
- Git Credential Manager simplifies authentication with your Azure DevOps Services/TFS Git repos

- Supports MFA and two-factor authentication with GitHub repositories
- You can also configure your IDE with a Personal Access Token or SSH to connect with your repos.

Git Commands Documentation on Local System: C:\Program Files\Git\mingw64\share\doc\git-doc

Getting Started with Git Commands

Git Data Transport Commands



Solution (.sln)

- ➔ Project (.csproj) → Restore → Build → EXE/DLL → Publish
    - CS Files
    - Config Files
    - XML Files
    - HTML / CSS / JS

1. Create the ASP.NET Core Web application with Unit Test project.

```
dotnet new sln -o HelloWorldApp
cd HelloWorldApp
dotnet new mvc -n HelloWorldApp.Web
dotnet sln HelloWorldApp.sln add HelloWorldApp.Web/HelloWorldApp.Web.csproj
```

2. OPTIONAL - Now the build the created project: Type the following code in Terminal Window (Ctrl + ~)

```
dotnet dev-certs https --trust
dotnet build
cd HelloWorldApp.Web
dotnet .\bin\Debug\netcoreapp3.1\HelloWorldApp.Web.dll
cd..
```

It will restore all the packages and build the project.

Note: The project is not added to the Git yet. Let's start adding to Git.

3. First step is initializing the Git repository.

```
git init
```

This creates a hidden folder .git in the current directory

4. Add **.gitignore** file as below to the root directory where git hidden folder is present.

```
Code .
In .gitignore add following:
obj
bin
```

Note that the .gitignore file is now added to the current directory.

5. Now let's check the status of git repository.

```
git status
```

There are no commits till now, that will be reflected in the output

Stage your changes

Git does not automatically add changed files to the snapshot when you create a commit. You must first stage your changes to let Git know which updates you want to add to the next commit. Staging lets you to selectively add files to a commit while excluding changes made in other files.

6. Now add all the files to the staging

```
git add .
```

Save with commits

Commits include the following information:

a. A snapshot of the files saved in the commit. Git snapshots the contents of all files in your repo at the time of the commit—this makes switching versions very fast and helps Git merge changes.

b. A reference to the parent commit(s). Commits with multiple parents occur when branches are merged together.

c. A short and to the point message describing the changes in the commit. You enter this message when you create the commit.

7. Now commit all the changes to the master branch (default branch) in local git repository

OPTIONAL: git config user.name sandeep

OPTIONAL: git config user.email sandeep@dss.com

git commit -m "First commit"

-a is used for staging all files before committing. Commit any files you've added with git add, and also commit any files you've changed since then

The commit command runs with the -m flag, which allows you to pass a message through the command line. If you don't provide this, Git will open up an editor in the terminal so you can enter a commit message.

Updating to Azure DevOps Repository

Once you are ready to share your code, get the clone URL for the repository you want to connect to and then set up a remote relationship (in this case, origin) so your repo can push changes to a shared repo.

If the Local Repository is already Existing:

8. **DevOps Portal** → Project → Repos → Copy the Commands from second section as below

git remote add origin https://DemoOrg@dev.azure.com/ DemoOrg /HelloWorldDemo/_git/HelloWorldDemo

git push -u origin --all

9. Execute the above commands on your local machine.

It prompts for sign in. Sign in with the DevOps credentials.

If New Local Repository has to be created. This creates a local copy of project.

10. Open another command window on your local machine

If the local folder is Empty then use the following command to clone the remote repo to local.

git clone <url>

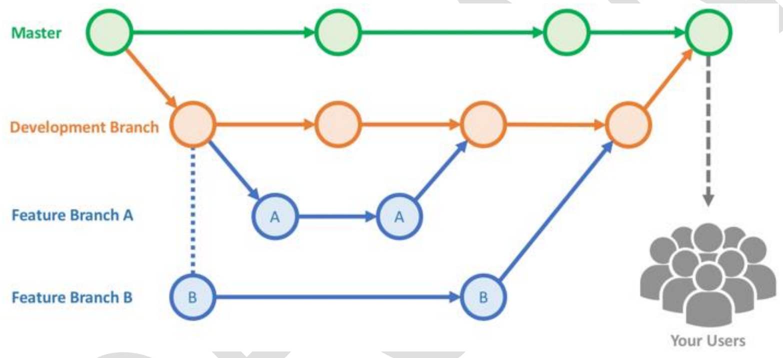
url should be copied from Azure DevOps portal eg:https:// DemoOrg @dev.azure.com/DemoOrg /HelloWorldDemo /_git/HelloWorldDemo.

In windows use "**doskey /history"** = To get the list of commands from history.

Working with Branches

Git branches aren't much more than a small reference that keeps an exact history of commits, so they are very cheap to create. Committing changes to a branch will not affect other branches, and you can share branches with others without having to merge the changes into the main project. Create new branches to isolate changes for a feature or a bug fix from your master branch and other work.

After you feel that your code is ready to be **merged** into the master branch in the main repository that's shared by all developers, you create what's called **a *pull request***. When you create a pull request, you're telling the other developers that you have code ready to review and you want it merged into the master branch. When your pull request is approved, it becomes part of the master codebase.



Some suggestions for naming your feature branches:

1. features/feature-name
2. features/feature-area/feature-name
3. users/username/description
4. users/username/workitem
5. bugfix/description
6. hotfix/description

Merging Branches

Merging is Git's way of putting a forked history back together again.

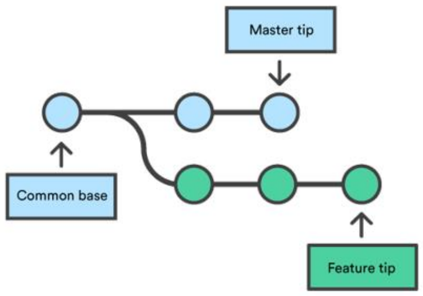
The ***git merge*** command lets you take the independent lines of development created by git branch and integrate them into a single branch.

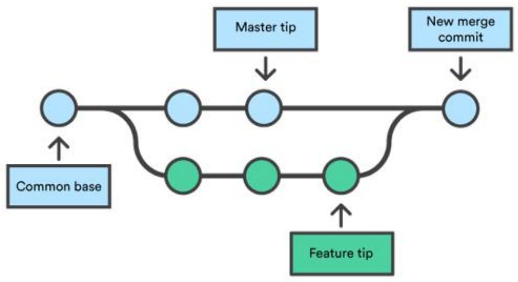
git merge <branch name>

where <branch name> is the name of the branch that will be merged into the receiving branch.

Note that merge merges the target branch into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that ***git merge*** is often used in conjunction with ***git checkout*** for selecting the current branch

Say we have a new branch feature that is based off the master branch. We now want to merge this feature branch into master.





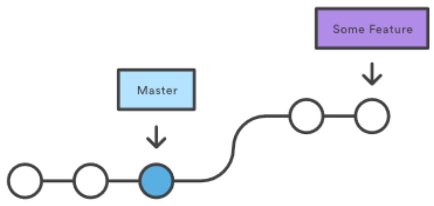
Git will determine the merge algorithm automatically (discussed below).

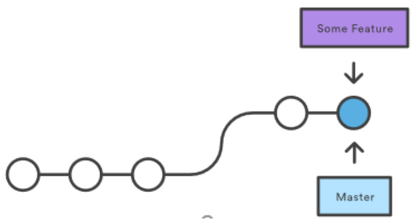
Two Git Merging Algorithms

Fast Forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.

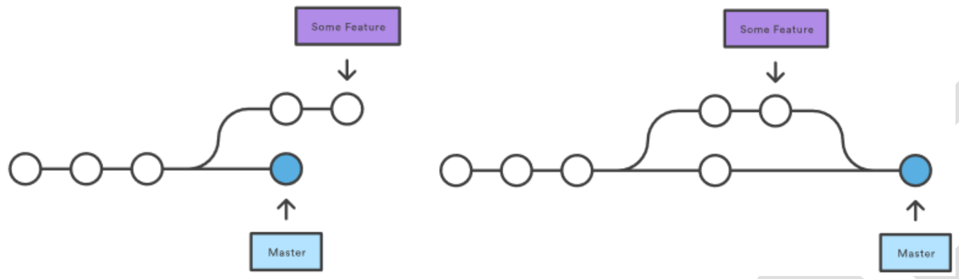
For example, a fast forward merge of some-feature into master would look something like the following:





3-way merge

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.



Merging the branch changes with Master branch

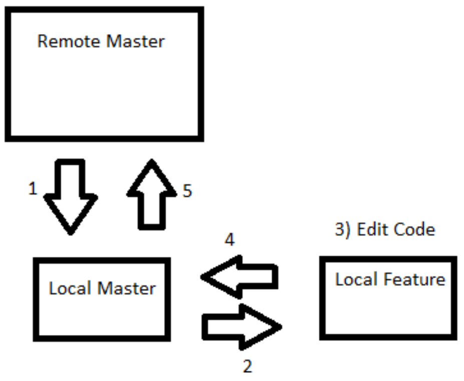
While you were busy working on your feature, there might have been changes made to the remote master branch. Before you **create a pull request**, it's common practice to get the latest from the remote master branch.

pull = fetch + merge (it is a combined command that does a fetch and then a merge)

a) **fetch** downloads the changes from your remote repo but do not apply them to your code.

b) **merge** applies changes taken from fetch to a branch on your local repo.

Working with Local Branch and Pushing Changes to Remote Master:



1. git pull origin master
2. git branch feature1 #

   git checkout feature1
3. Edit the Code in Index.html

```
git add .
git commit -m "feature1"
```

4. **git checkout master**

   git merge feature1

5. git push origin master

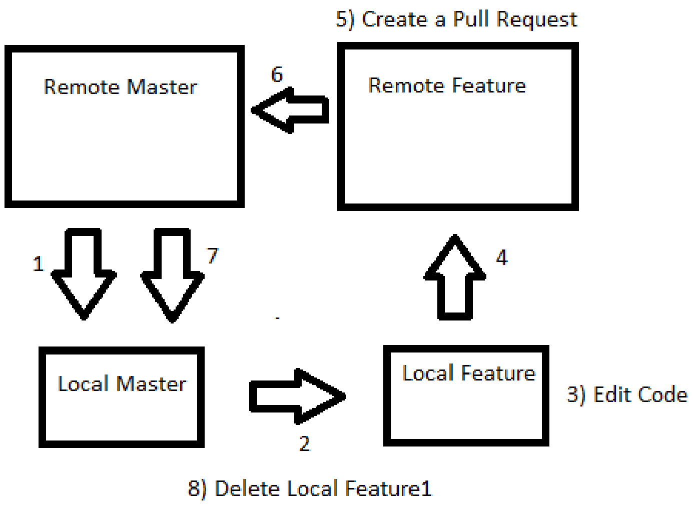
6. **git log** #Press 'q' to quit

Create a Pull Request (PR) and Commit Changes to Master

Pull requests let your team give feedback on changes in feature branches before merging the code into the master branch. Reviewers can step through the proposed changes, leave comments, and vote to approve or reject the code. Azure DevOps provides a rich experience for creating, reviewing, and approving pull requests.
Pull requests will be issued from feature branch to master branch.

Workflow

1. Create a Feature Branch
2. Checkout to Feature Branch
3. Develop the feature (Edit the files)
4. Stage and Commit changes to Local Feature Branch
5. Push the Commit to Remote Feature Branch
6. Create a Pull Request (Pull from Remote Feature to Remote Master)
7. Pull Request should [Approved] or Rejected.
8. On my local - Pull from the Remote master so that Local master is updated...
9. Delete the Local feature and Remote Feature...



Step1: Create a feature branch – Edit the code and update the feature branch in remote repo

1. git pull origin master
2. git branch feature/edit-home-title

   git checkout feature/edit-home-title
3. Edit the code

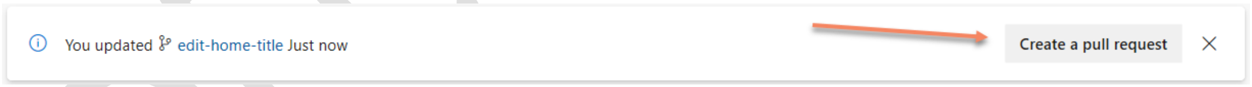
   git add .

   git commit -m "Some Comment"
4. git push origin feature/edit-home-title

Step2: Creating Pull requests in Azure DevOps portal

5. GO to the Azure repos and navigate to feature branch "**edit-home-title"**

   Click on "Create a Pull Request"



   Provide title and description and click on **Create**
6. Now login as the reviewer (email id given while creating pull request)

   Now navigate to Pull requests from Left side menu and click on displayed Pull request.

   1. Navigate to Files tab and review the code changes…

   Now click on the pull requests and check the code changes and we can approve it or reject it. The suggestions can be made in discussion board. Once we are done. We can approve the request.

   Once approved. We can **complete** the merger in the by clicking **Complete → Provide the required details → Complete merge**

Note that the branch edit-home-title is deleted and changes are now reflected in master branch.

Step3: Fetch the latest code from the remote repository and merges it into your local repository and delete the local feature branch

Login as Approver and → Go to Pull Request → Commit the PR

7. git checkout master
   git pull origin master
8. git branch -d feature/edit-home-title
9. git branch #Displays only master branch

ADD A RULE TO REQUIRE A REVIEW:

Branch policies help teams protect their important branches of development. Policies enforce your team's code quality and change management standards.

1. Select the Branch → Select . . . icon → Branch Policies
2. Opens configure your policies in the **Policies** page.
3. Check "**Require a minimum number of reviewers"**
4. Minimum number of reviewers = 1
5. Select **Save changes** to apply your new policy configuration.

Note: If a policy is created on a branch, direct push to that branch is not allowed.

Test the Rule

1. git checkout master
   git pull origin master
   git checkout -b bugfix/home-page-typo
2. Edit the Index.cshtml
   git add .
   git commit -m "Fixed home page content"
   git push origin bugfix/home-page-content
3. **Go to DevOps and → Create a pull request**
4. You can see that a human review is required before you can merge the change.
5. To merge the pull request, select **Merge pull request**.
6. Select the **Use your Administrator privileges to merge this pull request** check box, and then select **Confirm merge**.

Recovering a Deleted Branch

- **git reflog** and find the SHA1 for the commit at the tip of your deleted branch,
- git checkout -b <branch> <sha>

Working with Merge Conflicts

Example to demo AUTO RESOLVE if merge conflict

Open Command Window1: Create Local Git and also Push the same to remote repo.

D:\Demo\>**git config user.name user1**

D:\Demo\>**git config user.email user1@dss.com**

D:\Demo\>md User1

D:\Demo\>cd User1

D:\Demo\User1>git init

D:\Demo\User1>**git remote add origin <URL>**

D:\Demo\User1>Notepad demo.txt

```
This is line1
This is line2
```

D:\Demo\User1>git add .

D:\Demo\User1>git commit -m "Initial Commit"

D:\Demo\User1>git push origin master

Open Command Window2: Clone the repo and add a new line and push to remote

D:\Demo\>md User2

D:\Demo\>cd User2

D:\Demo\User1>**git clone <URL>**

D:\Demo\User1>**git pull origin master**

D:\Demo\>**git config user.name user2**

D:\Demo\>**git config user.email user2@dss.com**

D:\Demo\User2>Notepad demo.txt (Add a new line of text)

```
This is line1
This is line2
This is by User2
```

D:\Demo\User2>git add .

D:\Demo\User2>git commit -m "User2 Commit"

D:\Demo\User2>git push origin master

Example to demo MANUAL RESOLVE if merge conflict

Open Command Window1: Add a New line (different from User2). Auto resolves merge conflict

D:\Demo\User1>Notepad demo.txt (Add a new line of text)

```
This is by User1
This is line1
This is line2
```

D:\Demo\User1>git add .

D:\Demo\User1>git commit -m "User1 Commit"

D:\Demo\User1>git push origin master

Note that this will get rejected.

D:\Demo\User1>git pull **origin master**

Resolves Merge Conflict Automatically and does fast forward merge

D:\Demo\User1>Type Demo.txt

Note that it will auto merge the changes as they are in different lines…

```
This is by User1
This is Line1
This is Line2
This is by User2
```

D:\Demo\User1>git push origin

Switch to Command Window2: Edit the same line (First Line) which was updated by user1 without fetching the changes of User1

```
This is line by User2
This is Line1
This is Line2
This is by User2
```

D:\Demo\User2>git add .

D:\Demo\User2>

D:\Demo\User2>git push origin master

Merge Conflict will be reported…

D:\Demo\User2>git pull origin master

Merge Conflict will be reported again because auto merge will not work

Resolve the conflict manually by editing the file…

D:\Demo\User2>git add .

D:\Demo\User2>git commit -m "Update"

D:\Demo\User2>git push origin master

Undo Changes using reset or revert

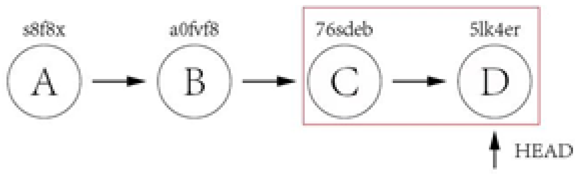
About Command: git reset

Reset your local branch to a previous commit.

git **reset** --hard HEAD
git **reset** --hard <SHA-ID-OF-COMMIT>

1. The --hard part of the command tells Git to reset the files to the state of the previous commit and **discard any staged changes**.
2. The HEAD argument tells Git to reset the local repository to the most recent commit. If you want to reset the repo to a different commit, provide the ID instead of HEAD.

Assuming we have below few commits.

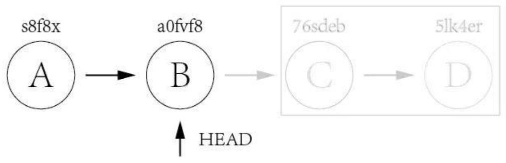


Commit A and B are working commits, but commit C and D are bad commits. Now we want to rollback to commit B and drop commit C and D. Currently HEAD is pointing to commit D *5lk4er*, we just need to point HEAD to commit B *a0fvf8* to achieve what we want.

It's easy to use git reset command.

```
git reset --hard a0fvf8
```

After executing above command, the HEAD will point to commit B.



But now the remote origin still has HEAD point to commit D, if we directly use **git push** to push the changes, it will not update the remote repo, we need to add a **-f** option to force pushing the changes.

```
git push -f origin master
```

The drawback of this method is that all the commits after HEAD will be gone once the reset is done. In case one day we found that some of the commits ate good ones and want to keep them, it is too late. Because of this, many companies forbid to use this method to rollback changes.

About Command: git revert

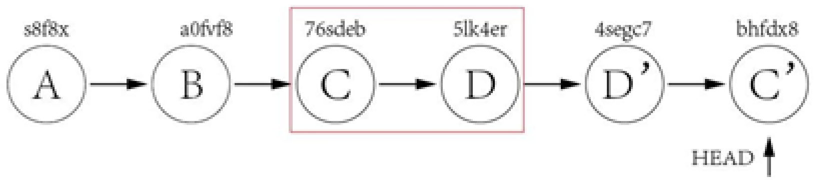
The use of **git revert** is to create a new commit which reverts a previous commit. The HEAD will point to the new reverting commit.

For the example of git reset above, what we need to do is just reverting commit D and then reverting commit C.

```
git revert 5lk4er

git revert 76sdeb
```

Now it creates two new commit D' and C',



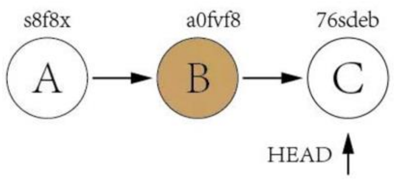
In above example, we have only two commits to revert, so we can revert one by one. But what if there are lots of commits to revert? We can revert a range indeed.

```
git revert OLDER_COMMIT^..NEWER_COMMIT
```

This method would not have the disadvantage of **git reset**, it would point HEAD to newly created reverting commit and it is ok to directly push the changes to remote without using the **-f** option.

Special Case

Now let's take a look at a more difficult example. Assuming we have three commits but the bad commit is the second commit.



It's not a good idea to use **git reset** to rollback the commit B since we need to keep commit C as it is a good commit. Now we can revert commit C and B and then use **cherry-pick** to commit C again.

Execute the following commands

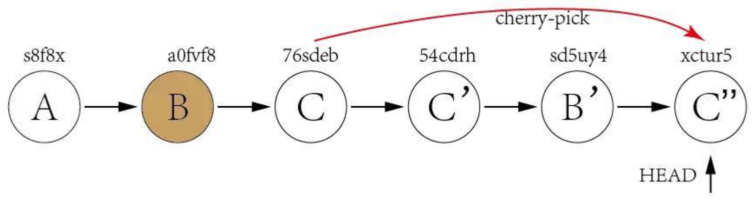
a) git **revert** 76sdeb

Head is at 54cdrh

b) git **revert** a0fvf8

Head is at sd5uy4

c) git **cherry-pick** 76sdeb



From above explanation, we can find out that the biggest difference between **git reset** and **git revert** is that git reset will reset the state of the branch to a previous state by dropping all the changes post the desired commit while git revert will reset to a previous state by creating new reverting commits and keep the original commits. It's recommended to use git revert instead of git reset in enterprise environment.

To temporarily Store uncommitted changes and revert to last commit:

- git stash

To apply the changes to working directory

- git stash pop

Branch Permissions (Security)

Set up permissions to control who can read and update the code in a branch on your Git repo. You can set permissions for individual users and groups, and inherit and override permissions as needed from your repo permissions.

Select the Branch → Select . . . icon → Branch Security

Permission	**Description**
Contribute	Users with this permission can push new commits to the branch and lock the branch.
Edit Policies	Can edit branch policies.

Bypass policies when completing pull requests	Users with this permission are exempt from the branch policy set for the branch when completing pull requests and can opt-in to override the policies by checking **Override branch policies and enable merge** when completing a PR.
Bypass policies when pushing	Users with this permission can push to a branch that has branch policies enabled. Note that when a user with this permission makes a push that would override branch policy, the push automatically bypasses branch policy with no opt-in step or warning.
Force Push (Rewrite History and Delete Branches)	Can force push to a branch, which can rewrite history. This permission is also required to delete a branch.
Manage Permissions	Can set permissions for the branch.
Remove Others' Locks	Can remove locks set on branches by other users.

Note: Git doesn't support to set security permissions at file level.

Branching Workflow Types

When evaluating a workflow for your team, it's most important that you consider your team's culture. You want the workflow to enhance the effectiveness of your team and not be a burden that limits productivity.

Some things to consider when evaluating a Git workflow are:

- Does this workflow scale with team size?
- Is it easy to undo mistakes and errors with this workflow?
- Does this workflow impose any new unnecessary cognitive overhead to the team?

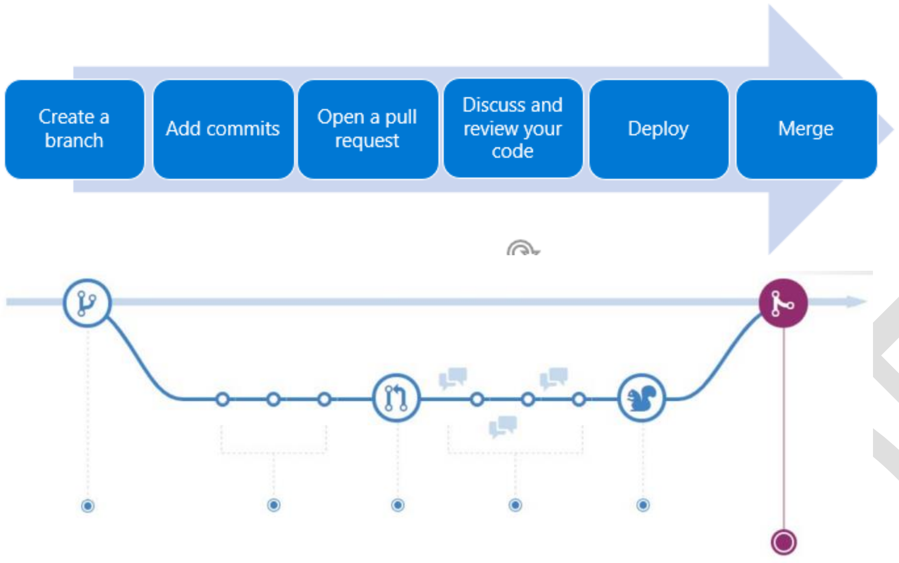
Common branch workflows

1. Trunk Based Development / Feature branching
2. Gitflow branching
3. Forking Workflow

Trunk Based Development / Feature Branch Workflow

- All feature development should take place in a dedicated feature branch instead of the master branch.
- Encapsulating feature development leverages pull requests, which are a way to initiate discussions around a branch.
- Share a feature with others without touching any official code.

- Trunk-based development is a logical extension of Centralized Workflow.



Let's cover the principles of what is being proposed:

The master branch:

- The master branch is the only way to release anything to production.
- The master branch should always be in a ready-to-release state.
- Protect the master branch with branch policies.
- Any changes to the master branch flow through pull requests only.
- Tag all releases in the master branch with Git tags.

The feature branch:

- Use feature branches for all new features and bug fixes.
- Use feature flags to manage long-running feature branches.
- Changes from feature branches to the master only flow through pull requests.
- Name your feature to reflect their purpose.

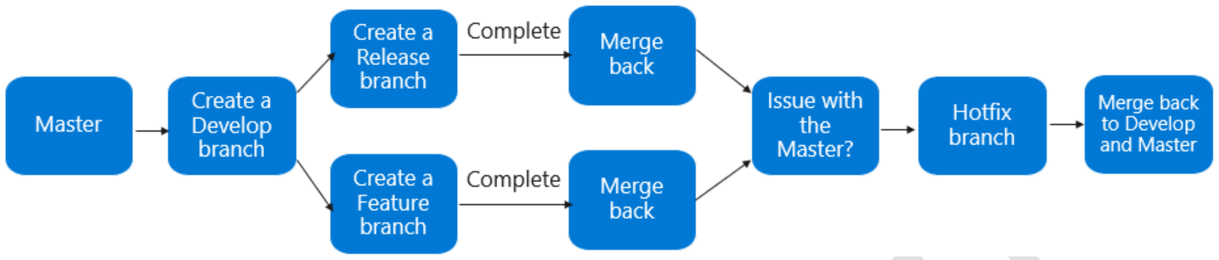
Pull requests:

- Review and merge code with pull requests.
- Automate what you inspect and validate as part of pull requests.
- Track pull request completion duration and set goals to reduce the time it takes.

GitFlow Branch Workflow:

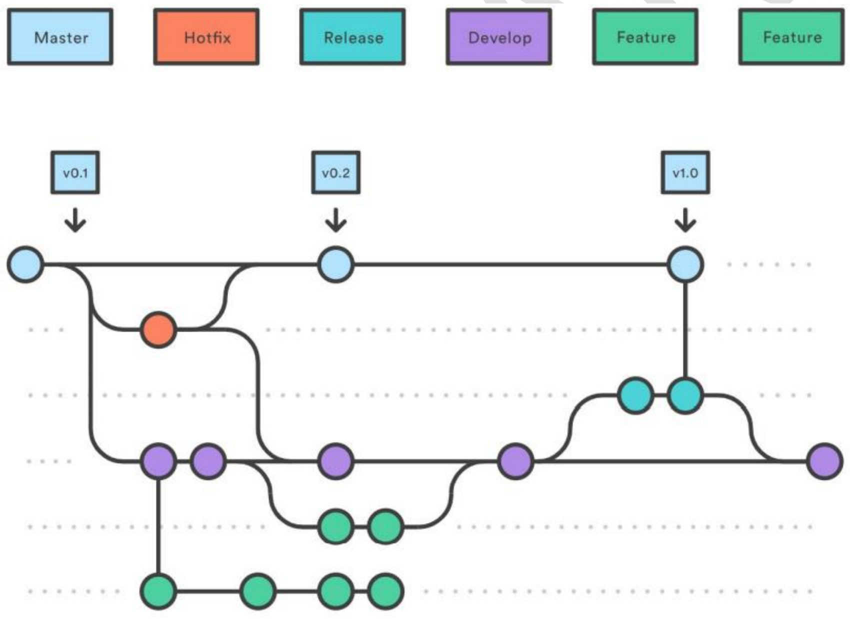
- GitFlow is great for a release-based software workflow.
- This provides a robust framework for managing larger projects.

- GitFlow offers a dedicated channel for hotfixes to production.



The overall flow of Gitflow is:

- A develop branch is created from master
- A release branch is created from develop
- Feature branches are created from develop
- When a feature is complete it is merged into the develop branch through a Pull request.
- When the release branch is done it is merged into develop and master
- If an issue in master is detected a hotfix branch is created from master
- Once the hotfix is complete it is merged to both develop and master



Note: Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. Once the release is ready to ship, it will get merged it into master and develop, then the release branch will be deleted.

Recommendation of Code Map to Environment:

- Develop Code -> Dev Environment
- Release Code -> QA + UAT Environment
- Master Code -> Prod Environment

Forking Branch Workflow:

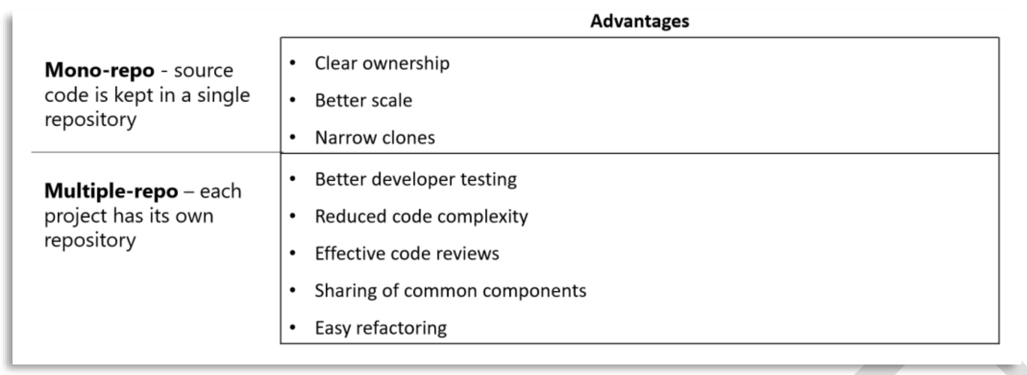
Forked repositories use the standard **git clone** command.

- Forking Branch workflow gives every developer their **own server-side repository**.
- Most often seen in public open source projects.
- Contributions can be integrated without the need for everybody to push to a single central repository.
- Typically follows a branching model based on the GitFlow Workflow.

The following is a step-by-step example of this workflow.

1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone. (git remote add upstream <official url>
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.
8. The developer opens a pull request from the new branch to the 'official' repository.
9. The pull request gets approved for merge and is merged into the original server-side repository.

Mono vs Multiple Repo



	Advantages
Mono-repo - source code is kept in a single repository	• Clear ownership • Better scale • Narrow clones
Multiple-repo – each project has its own repository	• Better developer testing • Reduced code complexity • Effective code reviews • Sharing of common components • Easy refactoring

Summary of Git Commands

Git task	Notes	Git commands
Tell Git who you are	Configure the author name and email address to be used with your commits.	git config --global user.name "Sam Smith" git config --global user.email sam@example.com
Create a new local repository		git init
Check out a repository	Create a working copy of a local repository:	git clone /path/to/repository
	For a remote server, use:	git clone username@host:/path/to/repository
Add files	Add one or more files to staging (index):	git add <filename> git add *
Commit	Commit changes to head (but not yet to the remote repository):	git commit -m "Commit message"
	Commit any files you've added with git add, and also commit any files you've changed:	git commit -a
Push	Send changes to the master branch of your remote repository:	git push origin master
	Push a specific branch to your remote repository	git push origin <branch_name>
	Push all branches to your remote repository	git push --all origin

Status	List the files you've changed and those you still need to add or commit:	git status
Stash	Takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.	git stash
	Popping your stash removes the changes from your stash and reapplies them to your working copy.	git stash pop
	You can reapply the changes to your working copy and keep them in your stash with git stash apply:	git stash apply
Connect to a remote repository	If you haven't connected your local repository to a remote server, add the server to be able to push to it:	git remote add origin <server>
	List all currently configured remote repositories:	git remote -v
Branches	Create a new branch and switch to it:	git checkout -b <branchname>
	Switch from one branch to another:	git checkout <branchname>
	List all the branches in your repo, and also tell you what branch you're currently in:	git branch
	Delete the feature branch:	git branch -d <branchname>
Update from the remote repository	Fetch and merge changes on the remote server to your working directory:	git pull
	To merge a different branch into your active branch:	git merge <branchname>
	View all the merge conflicts: View the conflicts against the base file: Preview changes, before merging:	git diff git diff --base <filename> git diff <sourcebranch> <targetbranch>
	After you have manually resolved any conflicts, you mark the changed file:	git add <filename>
Tags	You can use tagging to mark a significant changeset, such as a release:	git tag 1.0.0 <commitID>

	CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using:	git log
	Push all tags to remote repository:	git push --tags origin
Undo local changes	If you mess up, you can replace the changes in your working tree with the last content in head: Changes already added to the index, as well as new files, will be kept.	git checkout -- <filename>
	Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this:	git fetch origin git reset --hard origin/master