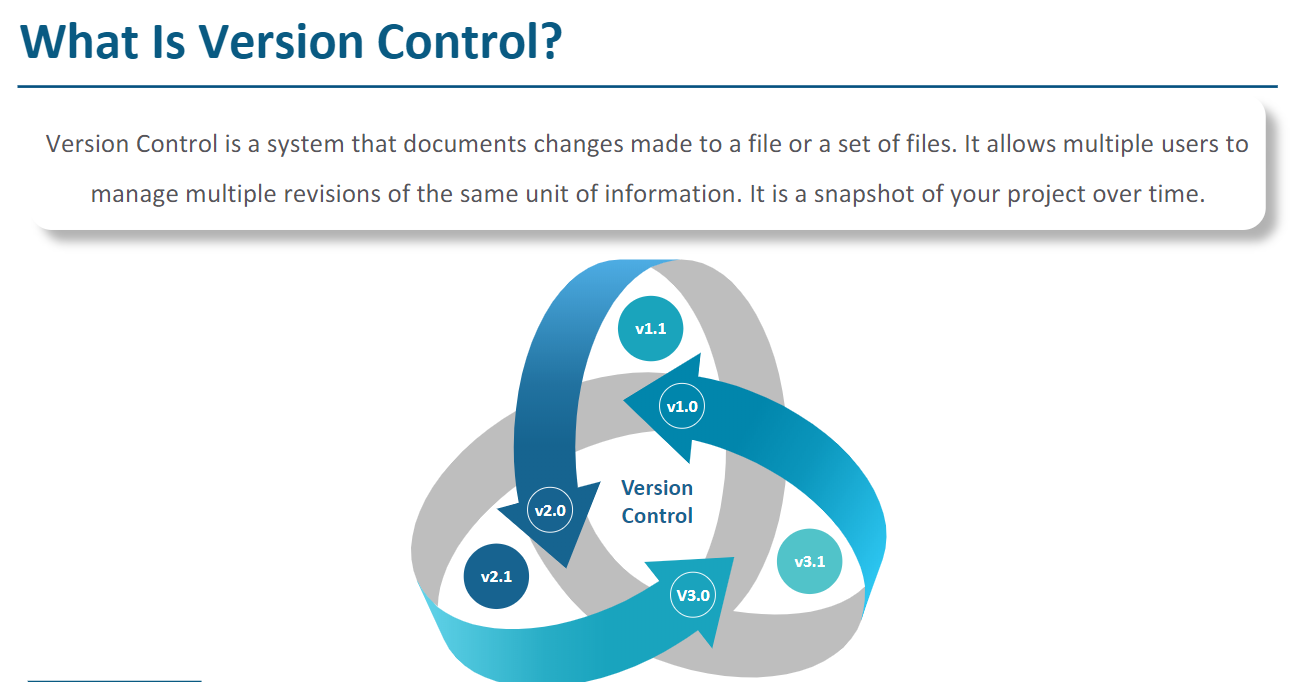
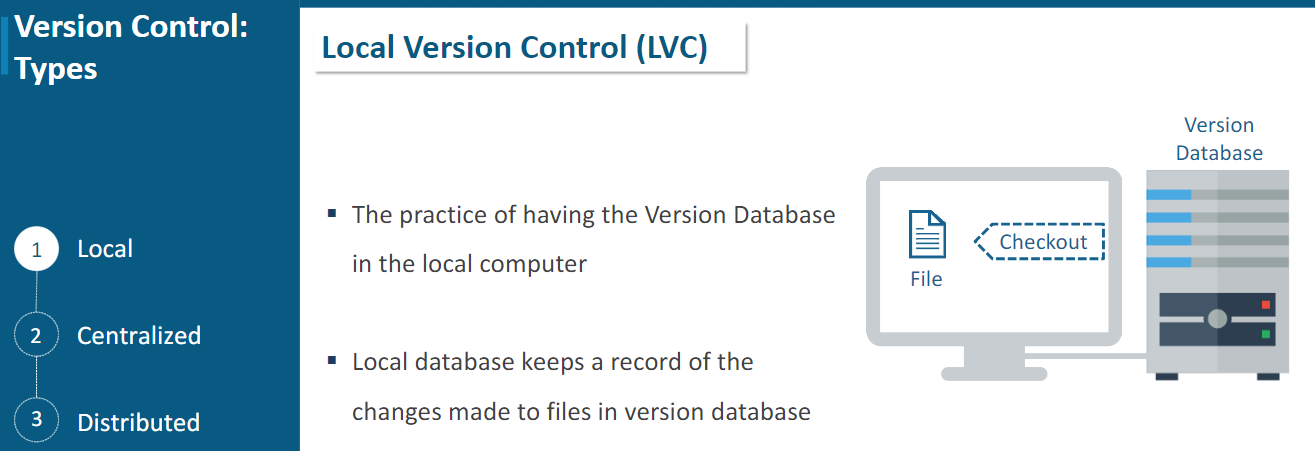
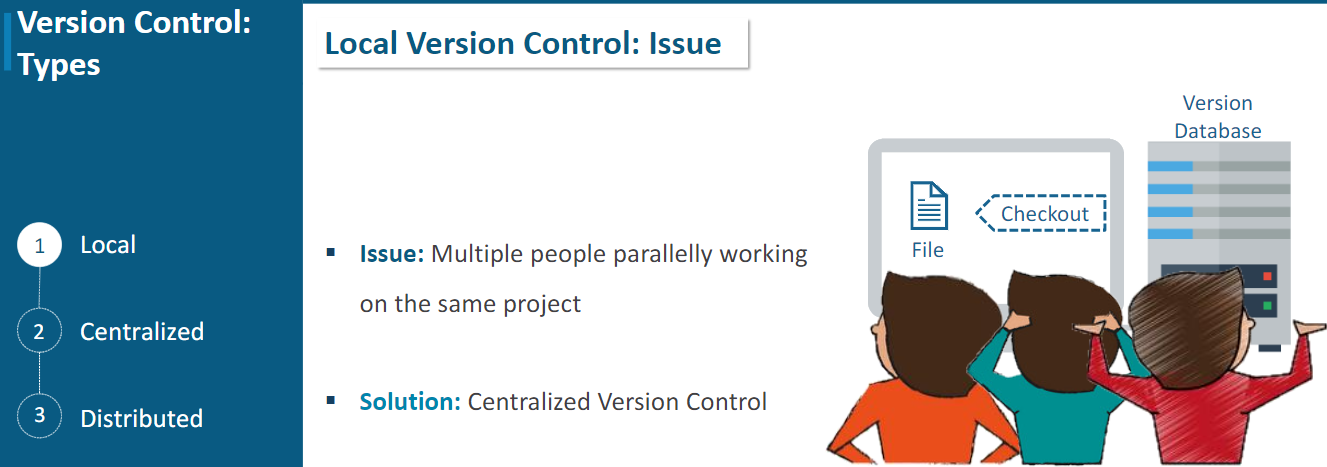
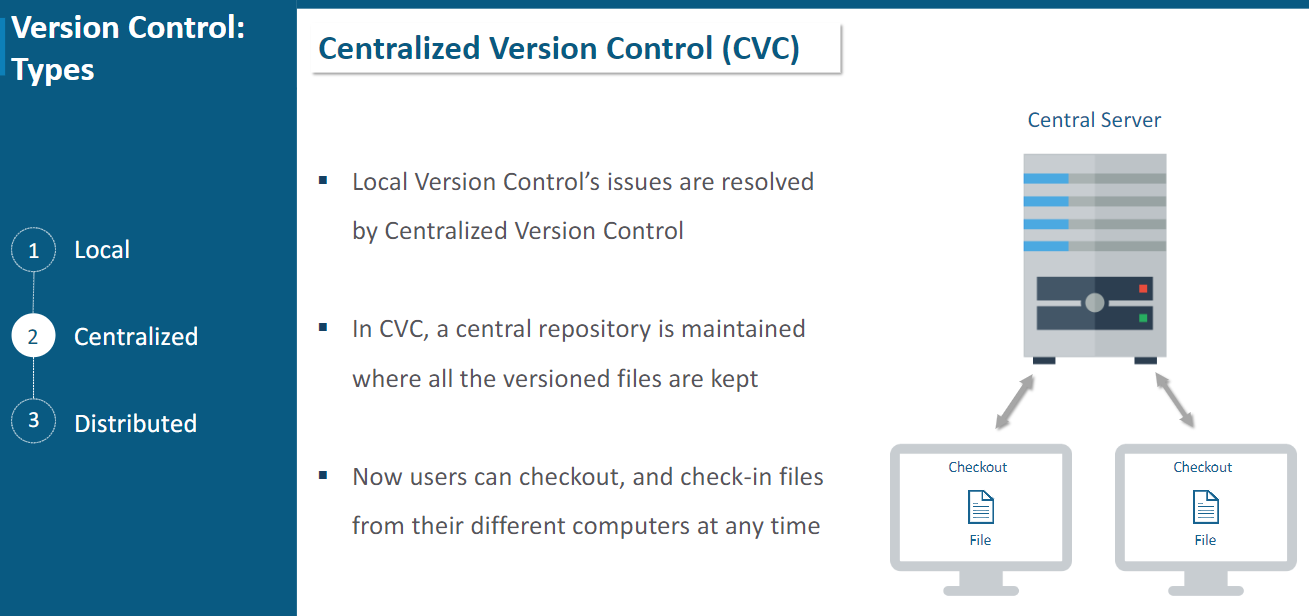
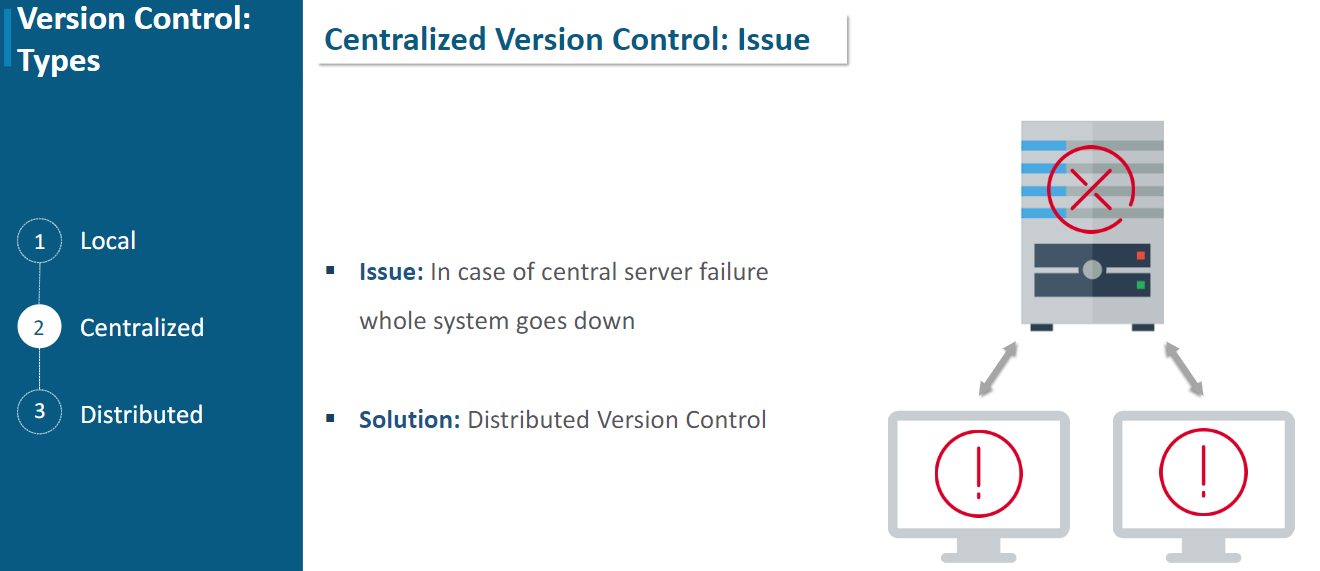


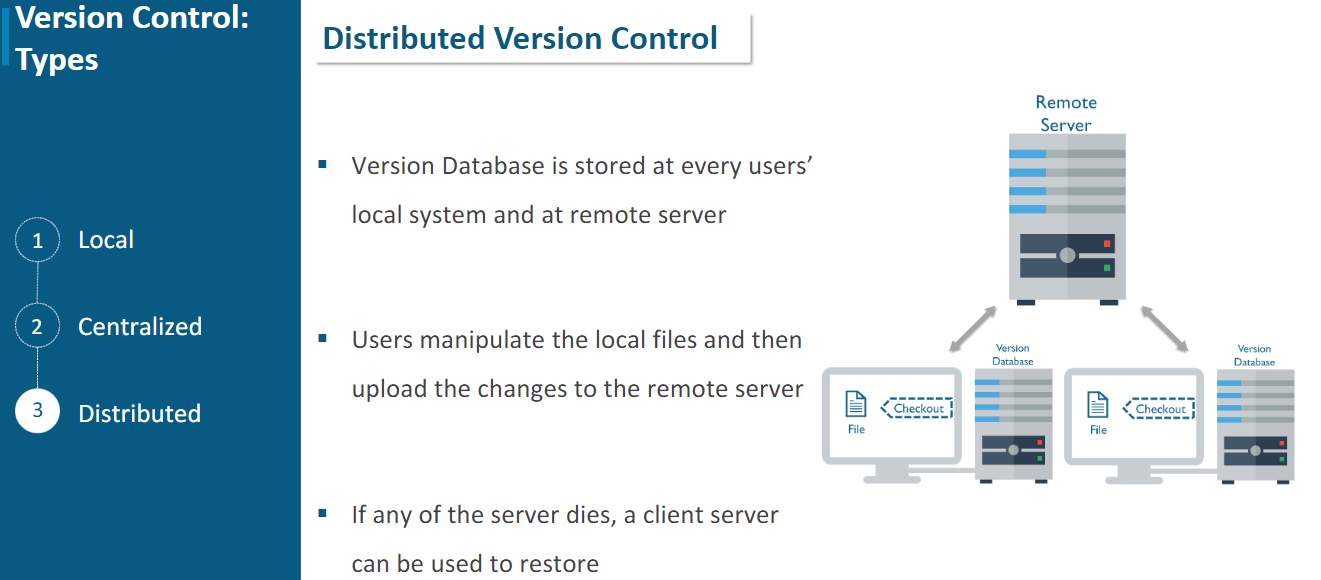
****

****

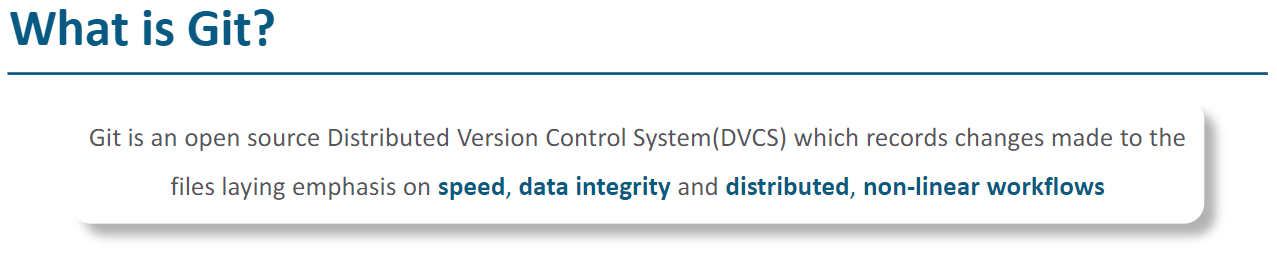
****

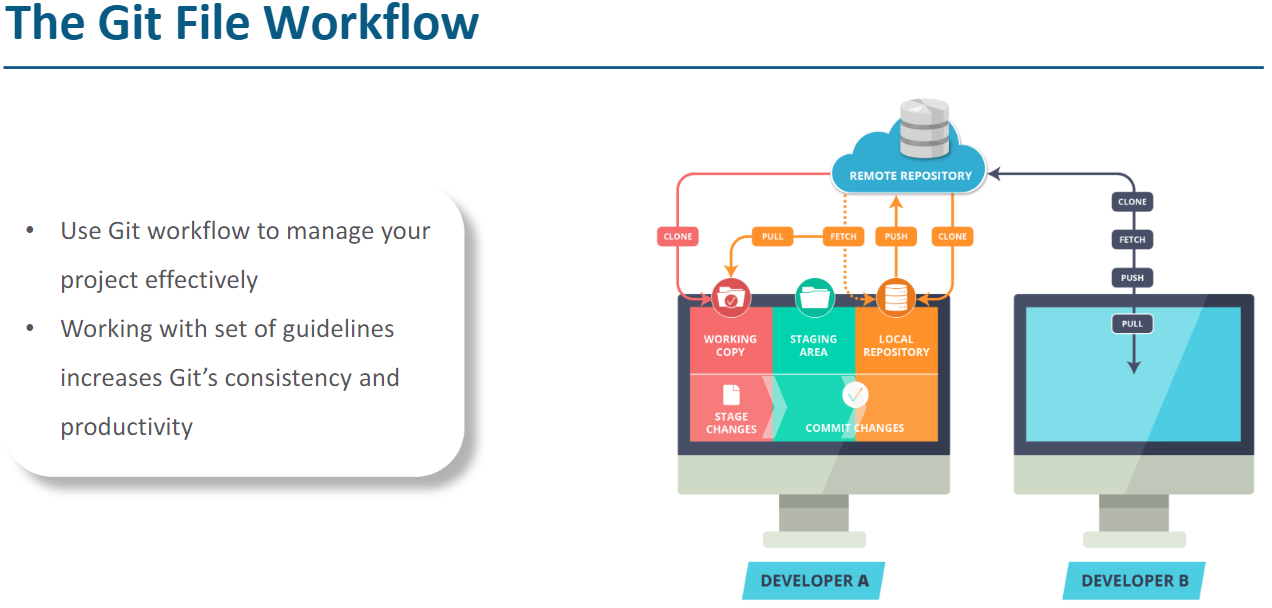
****

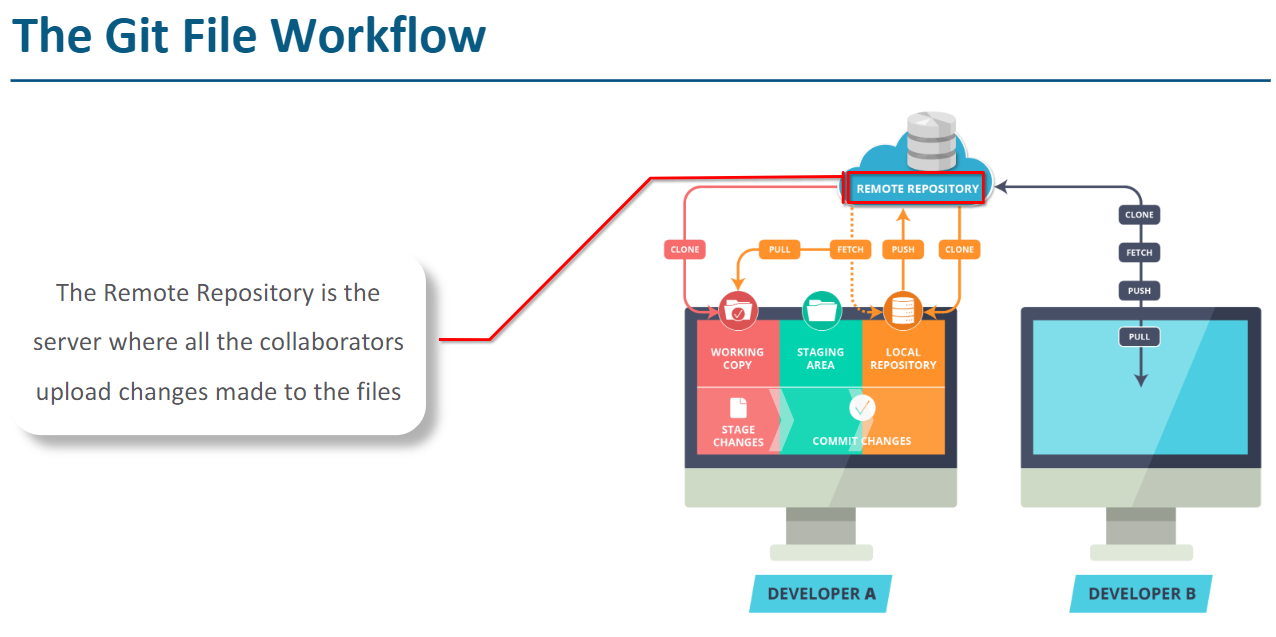
****

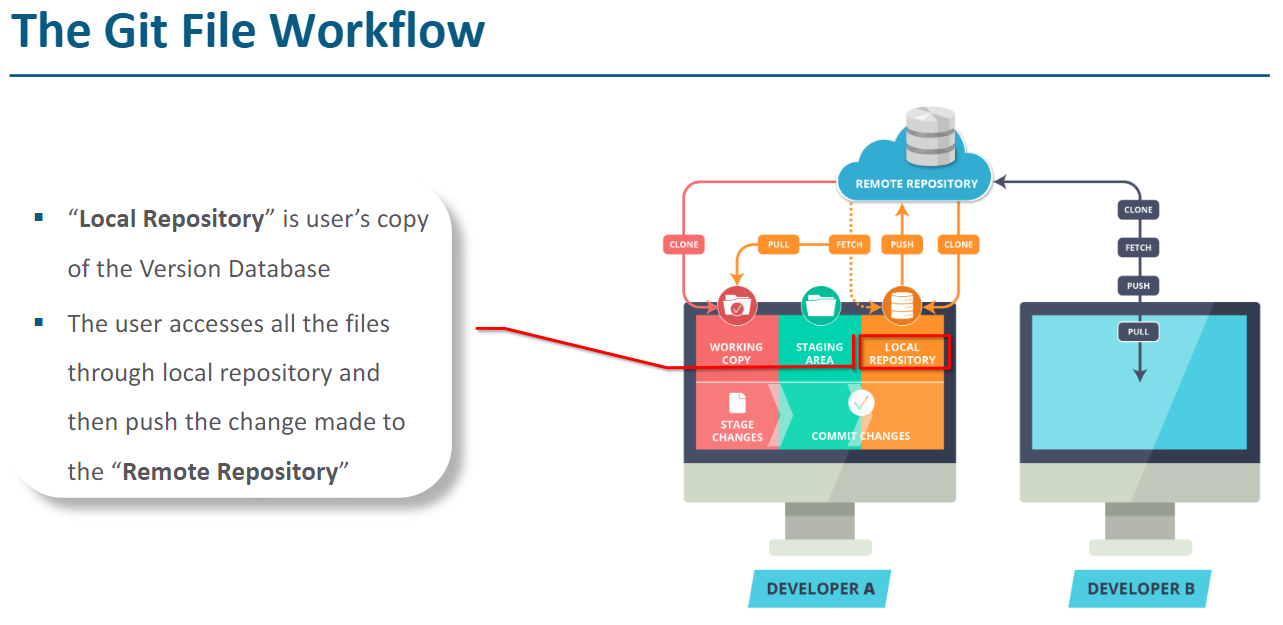
****

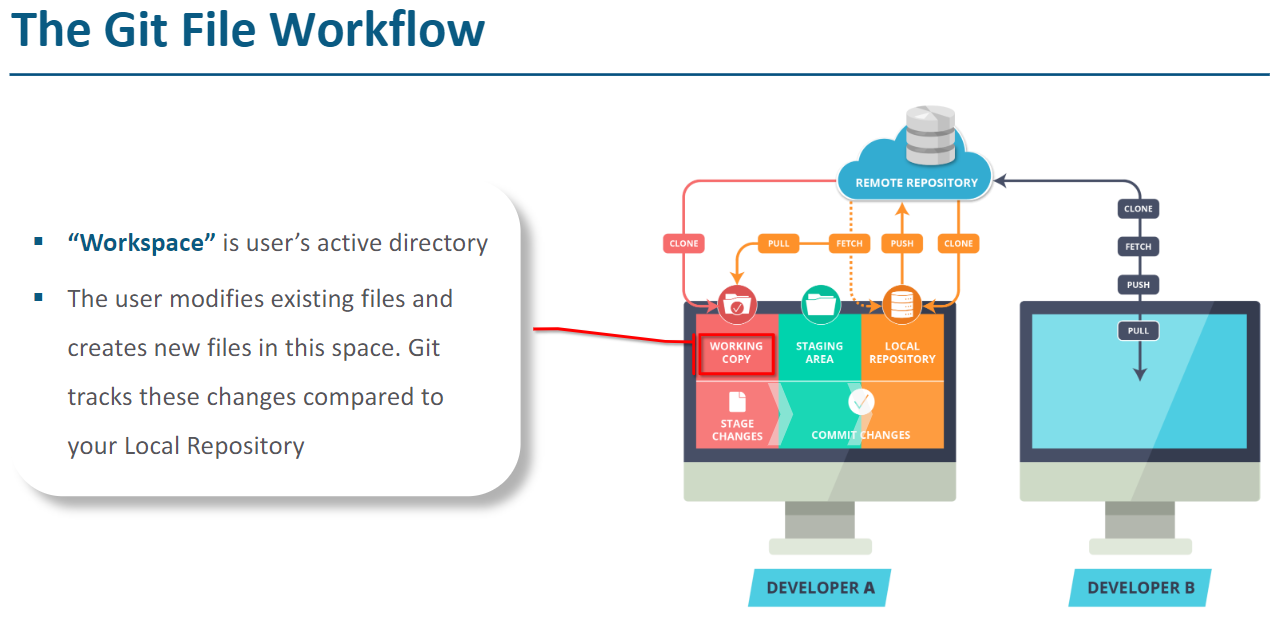
****

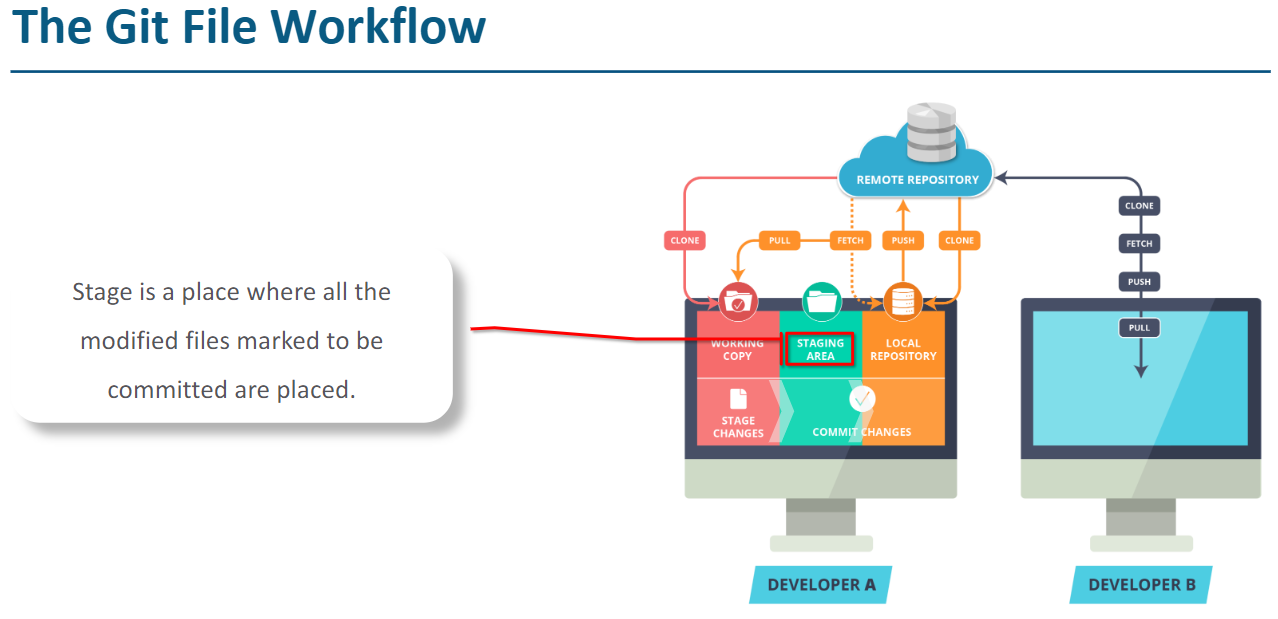
****

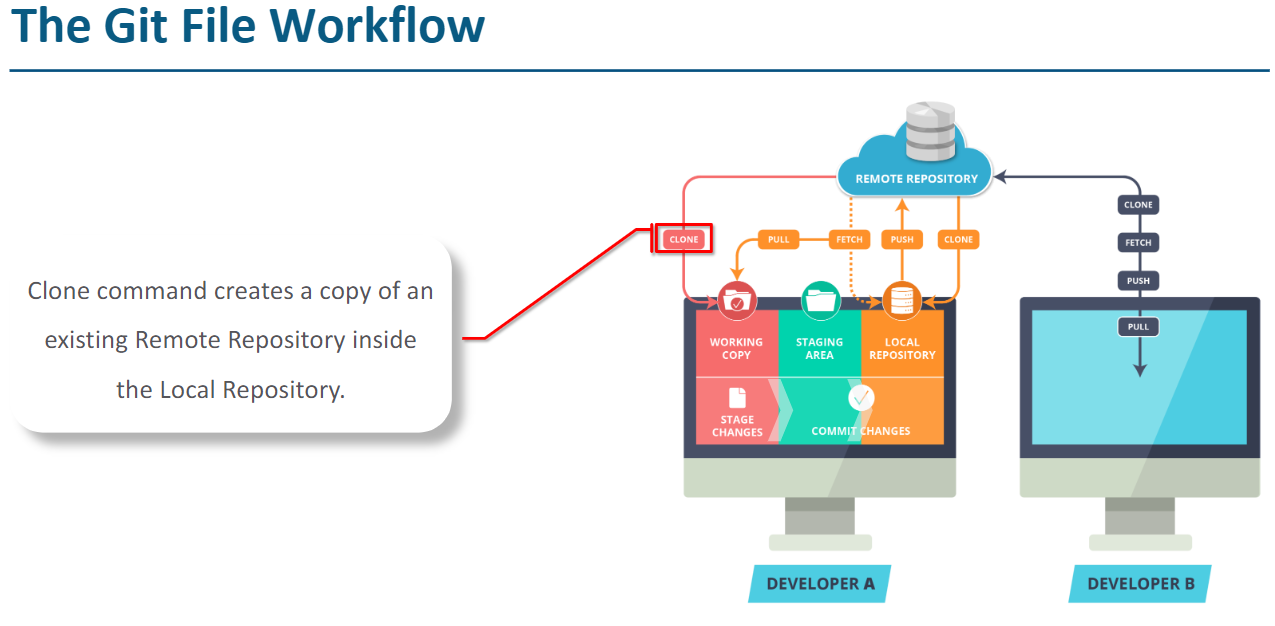
****

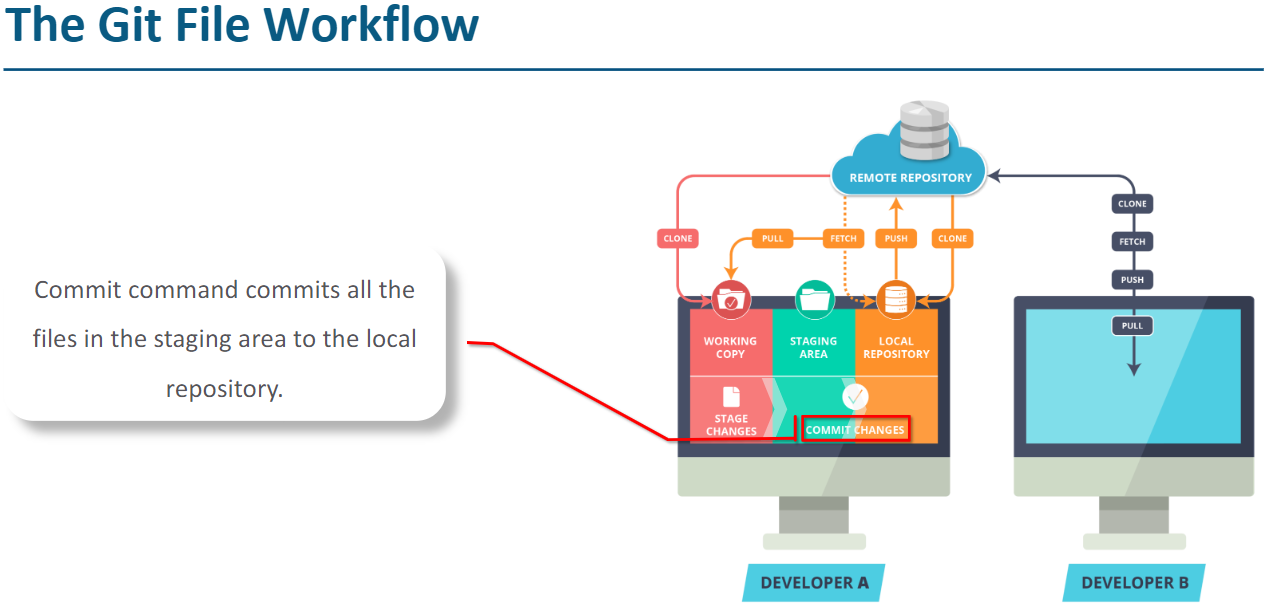
****

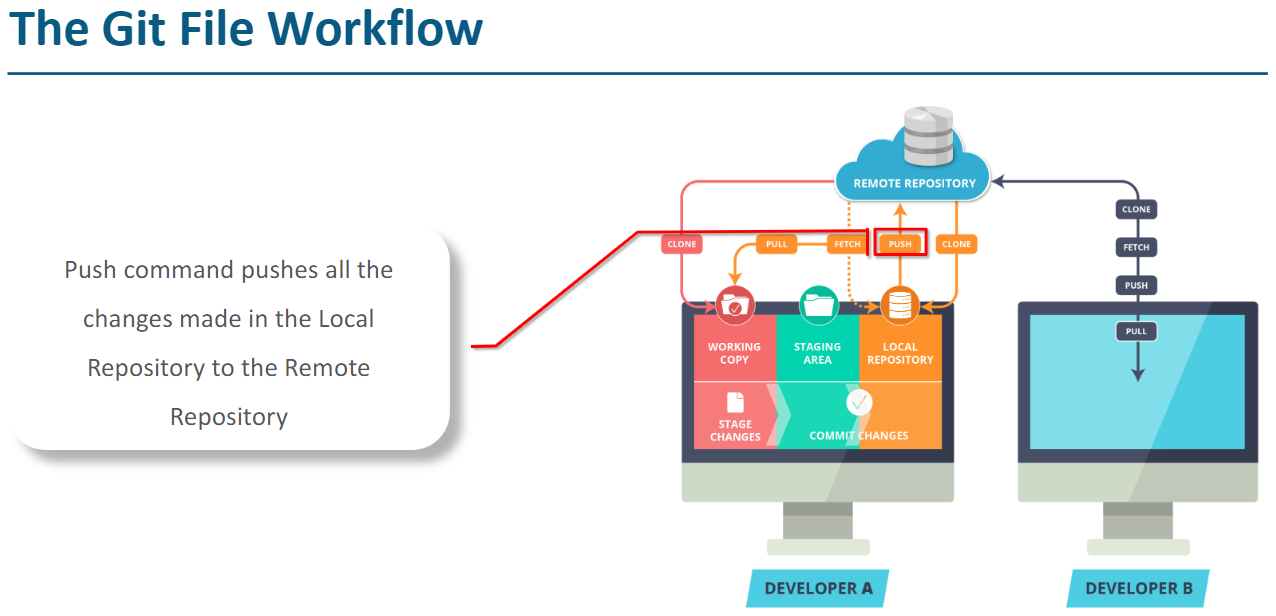
****

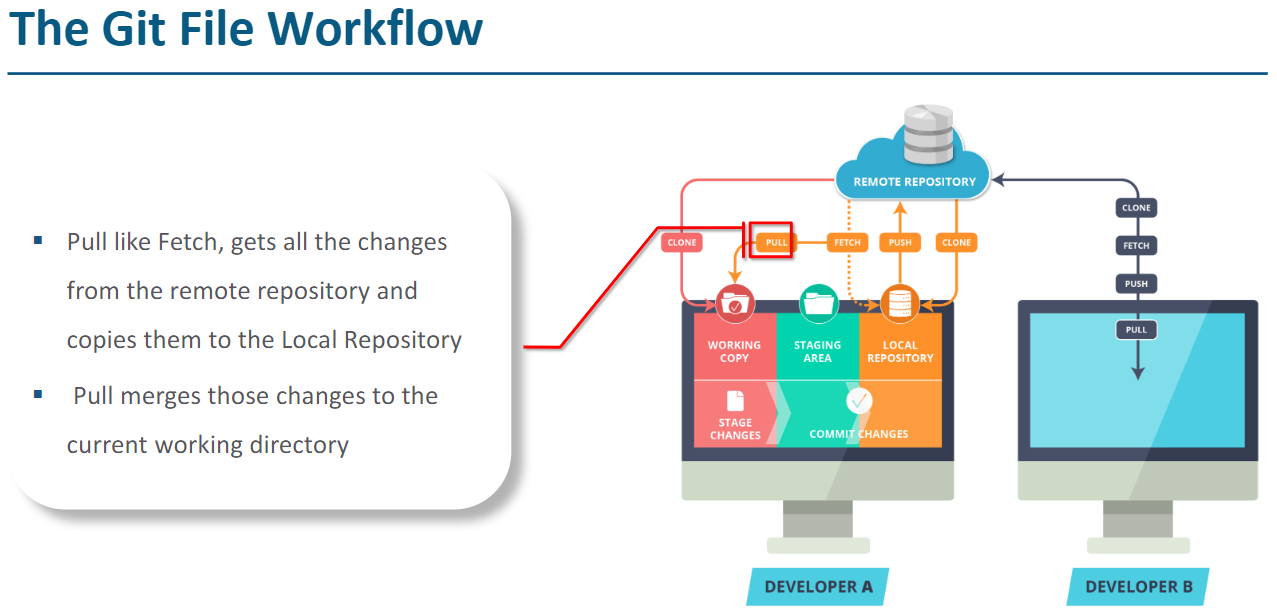
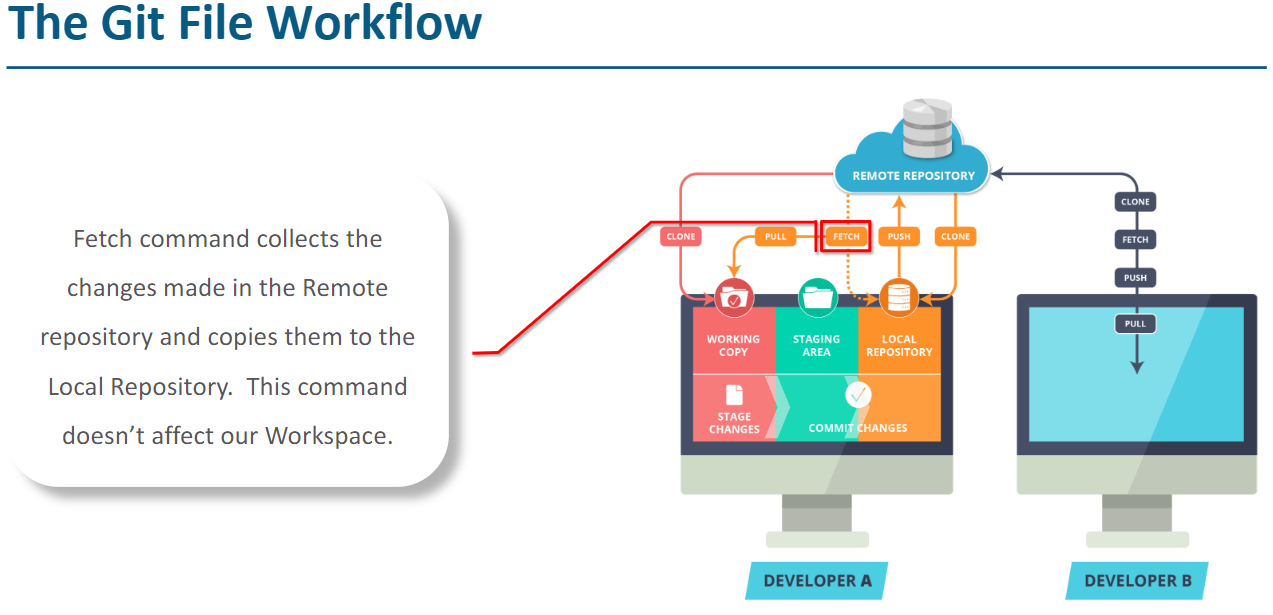
****

****

****

****

****

****

[**Installing GIT on Linux**](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git#Installing-on-Linux)

**$ sudo yum install git-all**

**$ sudo apt-get install git-all**

**$ git - -version**

**Or**

**$ git –v**

1. **Initial Configuration**

**$ git config - -global user.name "Firstname Lastname"**

**$ git config - -global user.email** [**myname@mycompany.com**](mailto:myname@mycompany.com)

**Checking Your Settings**

**$ git config - -list**

**You may see keys more than once, because Git reads the same key from different files**

**(/etc/gitconfig and ~/.gitconfig, for example). In this case, Git uses the last value for each unique key it sees.**

**You can also check what Git thinks a specific key’s value is by typing git config <key>:**

**$ git config user.name**

**John Doe**

**2. Creating a Git Repository**

**Initializing a Repository in an Existing Directory**

**$ git init**

**Cloning and Existing Repository**

**$ git clone https://github.com/libgit2/libgit2 (http protocol)**

**$ git clone user@server:path/to/repo.git (ssh protocol)**

**That creates a directory named “libgit2”, initializes a .git directory inside it, pulls down all the data for that repository,**

**and checks out a working copy of the latest version. If you go into the new libgit2directory, you’ll see the project files in there, ready to be worked on or used.**

**3. GIT Workflow (Local)**

**Once you have cloned or initialized a new GIT project, begin changing files as needed. There is no locking of files or a traditional VCS checkout concept. Simply begin editing files in a progression towards a committable state.**

**Adding (staging)**

**New files must be ‘staged’ with the add command as follows:**

**$ git add index.html**

**$ git add javascript/**

**$ git add \*.js**

**Committing**

**Once all desired files are added and staged, a commit command saves the pending additions to the local repository.**

**The default text $EDITOR will be opened for entry of the commit message.**

**$ git commit**

**$ git commit –m “Your commit message”**

**Status**

**To check the current status of a project’s local directories and files, such as modified, new, deleted or**

**Untracked files, invoke the following command:**

**$ git status**

**4. GIT Workflow (Remote Collaboration)**

**Remotes**

**Full addresses of your configured remotes can be viewed with:**

**$ git remote -v**

**To add a new remote, type:**

**$ git remote add <remote name> <remote address>**

**git fetch**

**$ git fetch <remote>**

**Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.**

**$ git fetch <remote> <branch>**

**Same as the above command, but only fetch the specified branch.**

**To approve the changes and merge them into your local master branch with the following**

**commands:**

**$ git checkout master**

**$ git log origin/master**

**Then we can use git merge origin/master**

**$ git merge origin/master**

**The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.**

**Git Pull**

**Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with git fetch followed by git merge, but git pull rolls this into a single command.**

**$ git pull <remote>**

**Above command will fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as:**

**$ git fetch <remote>**

**followed by**

**$ git merge origin/<current-branch>**

**Note: You can think of “git pull” as Git’s version of “svn update”**

**Git Push**

**Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.**

**$ git push <remote> <branch>**

**Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.**

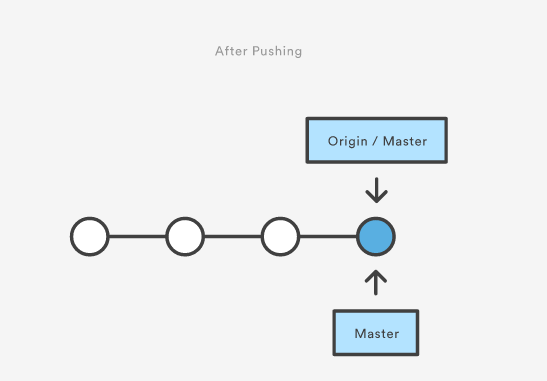
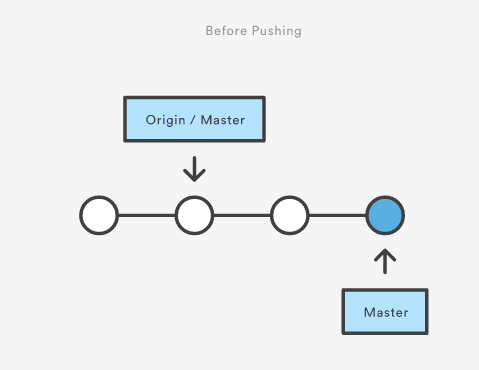
**$ git push <remote> --force**

**Same as the above command but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you’re sure you know what you’re doing.**

**$ git push <remote> --all**

**Push all your local branches to the specified remote.**

**The most common use case for git push is to publish your local changes to a central repository. After you’ve accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.**

****

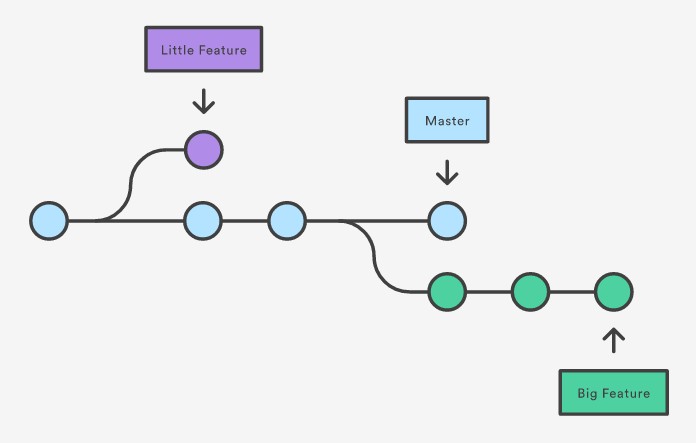
**The above diagram shows what happens when your local master has progressed past the central repository’s master and you publish changes by running git push origin master. Notice how git push is essentially the same as running git merge master from inside the remote repository.**

**Branch**

**A branch represents an independent line of development. You can think of them as a way to request a brand-new working directory, staging area, and project history.**

**New commits are recorded in the history for the current branch, which results in a fork in the history of the project.**

**The git branch command lets you create, list, rename, and delete branches. It doesn’t let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.**

****

**$ git branch**

**List all the branches in your repository.**

**$ git branch <branch>**

**Create a new branch called <branch>. This does not check out the new branch.**

**$ git branch -d <branch>**

**Delete the specified branch. This is a “safe” operation in that Git prevents you from**

**deleting the branch if it has unmerged changes.**

**$ git branch -D <branch>**

**Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all the commits associated with a particular line of development.**

**$ git branch -m <branch>**

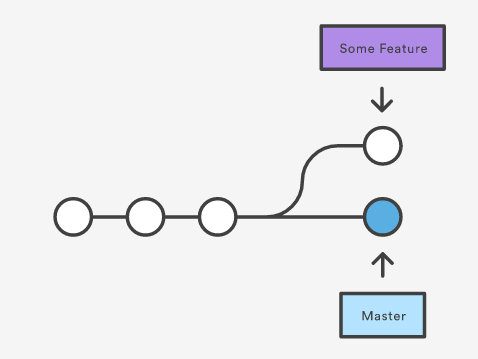
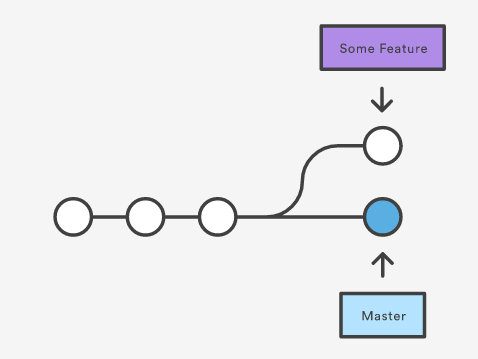
**Rename the current branch to <branch>.**

**Checkout**

**git checkout works hand-in-hand with git branch. When you want to start a new**

**feature, you create a branch with git branch, then check it out with git checkout.**

**You can work on multiple features in a single repository by switching between them with git checkout.**

**$ git checkout master $ git checkout some feature**

**$ git checkout <existing-branch>**

**Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.**

**$ git checkout -b <new-branch>**

**Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new- branch>**

**$ git checkout -b <new-branch> <existing-branch>**

**Same as the above invocation, but base the new branch off of <existing- branch> instead of the current branch.**

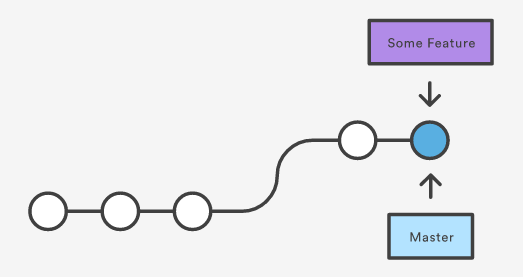
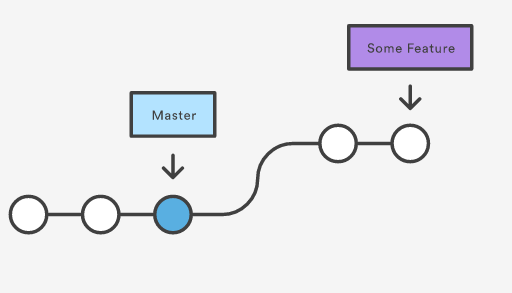
**Git merge**

**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>**

**Merge the specified branch into the current branch.**

**Before merging After merging**

**5. Viewing the Commit History**

**One of the more helpful options is -p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries:**

**$ git log -p -2**

**You can also use a series of summarizing options with git log. For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:**

**$ git log --stat**

**strongly urge you to try out all the options to understand each sub command for git log and what it is capable of.**

**6. Limiting Log Output**

**$ git log --since=2. weeks**

**This command works with lots of formats – you can specify a specific date like |"2008-01-15"|, or a relative date such as |"2 years 1 day 3 minutes ago".**

**Removing File from Repository**

**$ git rm <file name>**

**Git rm command delete file from repository as well as user’s system.**

**To remove file from git repository but not from system.**

**Git rm - - cached <File Name>**

**An error shows up of you try to delete staged file.**

**You can forced remove staged file using below command.**

**$ git rm -f <file name>**

**Git ignoring files**

**You can create a .gitignore file and add all the untracked files you want Git to ignore.**

**Config files:**

**/etc/gitconfig → all users, repositories (--system)**

**~/.gitconfig → one user, all repo (--global)**

**[repo]/.git/config → specific to repository (default)**

**=========================================**

**GIT Topics/commands**

**=========================================**

**Git installation (sudo yum install git or sudo apt-get install git)**

**git config**

**git init**

**git add (stage or track the files)**

**git commit**

**git status**

**git pull**

**git push**

**git log**

**git log --oneline (git reflog)**

**git diff**

**git branch**

**git checkout**

**git ignore**

**Merge conflicts**

**git tag**

**git checkout tagname/commitid (to go back to check a particular tag or commit)**

**Branching concepts**

**git branch newbranch (to create new branch)**

**git merge branchname (to merge with current branch)**

**[git pull = git fetch + git merge]**

**git remote**

**git rebase (https://www.atlassian.com/git/tutorials/merging-vs-rebasing)**

**git revert HEAD (git revert HEAD~3)**

**git reset --hard commitid (Example: git reset --hard 544d354)**

**https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting**

**git stash**

**=========================================**

**"Commits are cheap in Git. Do it as often as possible"**

**=========================================**

**Centralized VCS examples: SVN, Accurev, TFS, Perforce CVS**

**Distributed VCS examples: GIT, Mercurial**

**=========================================**

**Why VCS/SCM? VCS (Version Control system) SCM (Software configuration management)**

**1. Versioning**

**2. Collaboration**

**3. Due diligence**

**=========================================**

**DVCS vs CVCS (distributed VCS vs centralized VCS)**

**1. users can work offline**

**2. Multiple developers can contribute at the same time.**

**3. You don't need a dedicated server.**

**Branching and Merging**