# GIT 101 @ Sysfore

[GIT 101 @ Sysfore 1](#_Toc482346257)

[Git CLI Tool Installation 2](#_Toc482346258)

[Git Work Flow Diagram 2](#_Toc482346259)

[Setting up a new Repository 2](#_Toc482346260)

[Cloning an existing Repository 3](#_Toc482346261)

[Configuring GIT CLI 4](#_Toc482346262)

[Staging Code Changes 4](#_Toc482346263)

[Commiting staged files to project history 6](#_Toc482346264)

[Switch Context with Stashing 7](#_Toc482346265)

[Ignored Files - .gitignore 14](#_Toc482346266)

[Inspecting staged and untracked files 15](#_Toc482346267)

[Inspecting project history 15](#_Toc482346268)

[Collaboration @ Git Hub - Syncing 20](#_Toc482346269)

[Pushing into remote repo 21](#_Toc482346270)

[Fetching from remote repo 21](#_Toc482346271)

[Merging Upstream changes 22](#_Toc482346272)

[Creating Branches 24](#_Toc482346273)

[Checking out from branch 27](#_Toc482346274)

[Integrating into single branch 29](#_Toc482346275)

[Undoing changes 33](#_Toc482346276)

[Reverting changes 37](#_Toc482346277)

[Resetting changes 39](#_Toc482346278)

[Permanent purging 40](#_Toc482346279)

[Rewriting history 41](#_Toc482346280)

[Rebasing branch 43](#_Toc482346281)

[Rebasing branch - interactive 48](#_Toc482346282)

[Tracking history 52](#_Toc482346283)

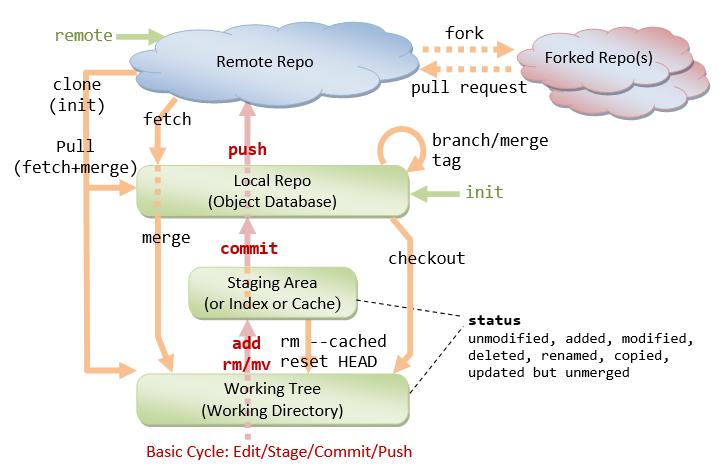
[Versioning with tags 53](#_Toc482346284)

[.gitignore 55](#_Toc482346285)

# Git CLI Tool Installation

* Download & install git bash from <https://git-scm.com/downloads>

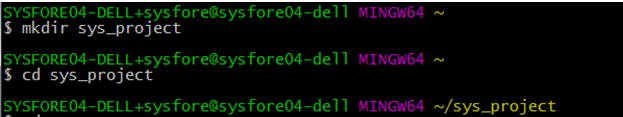
# Git Work Flow Diagram



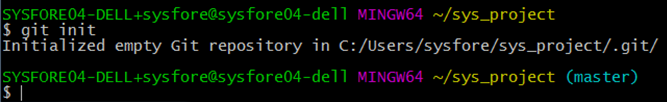
# Setting up a new Repository

Create local repository using **mkdir <repositoryname>**

Change the path to newly created repository using **cd <repositoryname>**

****

Transform the newly created directory into local repository using **git init**

****

.git folder will be created to store git revisions

Add all responsive , native project folders and files under the newly created local repository. This stages them for the first commit.

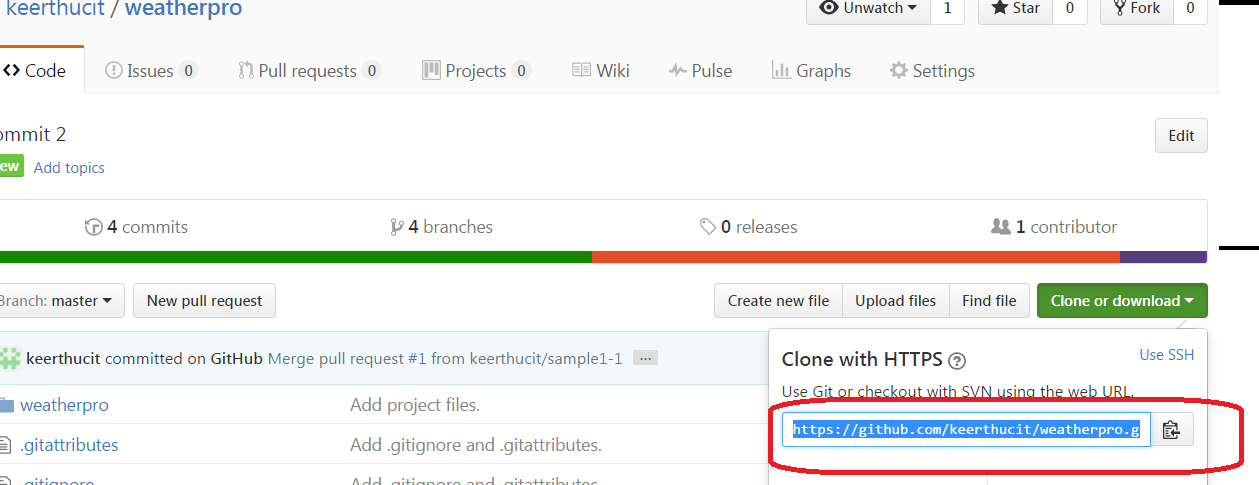
Go to the repository path created in windows or Linux system and copy the project files to this repository.

# Cloning an existing Repository

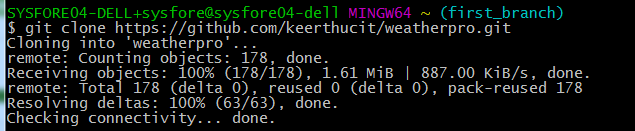
Clones an existing Git repository to workstation.

Clone the repository located at **<repo>** onto the local machine. The original repository can be located on the local filesystem or on a remote machine .

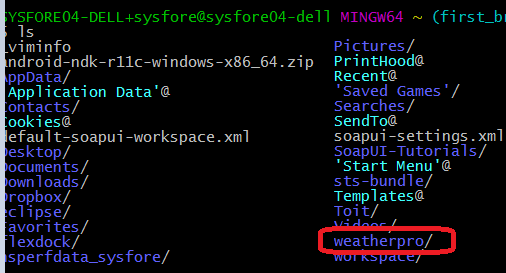
Copy the url of repository to be cloned.



Using GIT CLI clone the repository -**git clone https://github.com/keerthucit/weatherpro.git**



Find the new folder created as local repository similar to the remote.



\*\*\* Like [**git init**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init)**,** cloning is generally a one-time operation—once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository.

# Configuring GIT CLI

**git config user.name keerthanaBathrinath**

Configure with the author name to be used for all commits in the current repository. Use the **--global**flag to set configuration options for the current user.

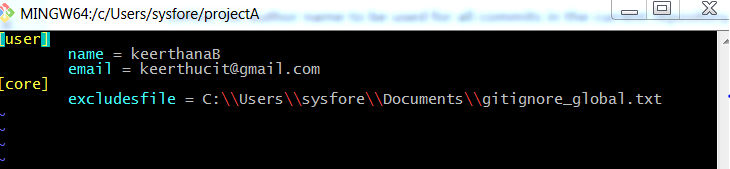


**git config --global user.email** [**keerthana.b@sysfore.com**](mailto:keerthana.b@sysfore.com)Define the author email to be used for all commits by the current user.



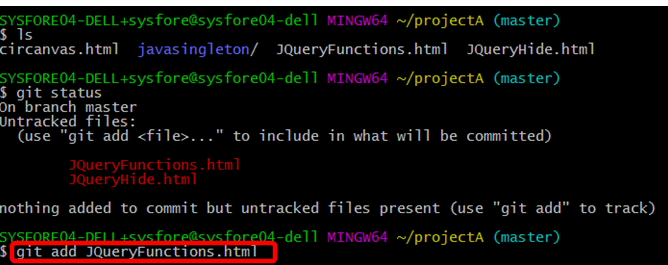
**git config --global -- edit** Open the global configuration file in a text editor for manual editing.



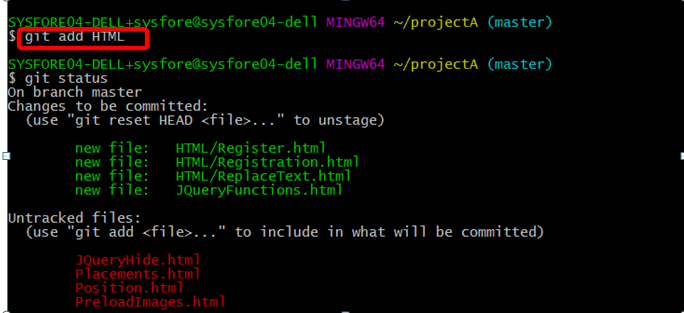


# Staging Code Changes

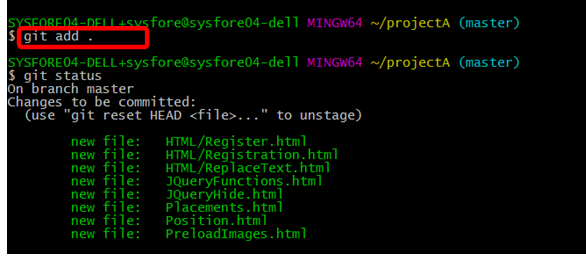
The **git add <file>** command adds a change in the working directory to the staging area.**git add**doesn't really affect the repository in any significant way—changes are not actually recorded until  [**git commit**](https://www.atlassian.com/git/tutorials/saving-changes/git-commit)is run.With the command [**git status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status)view the state of the working directory and the staging area.



**git add <directory>** to Stage all changes in <directory> for the next commit.

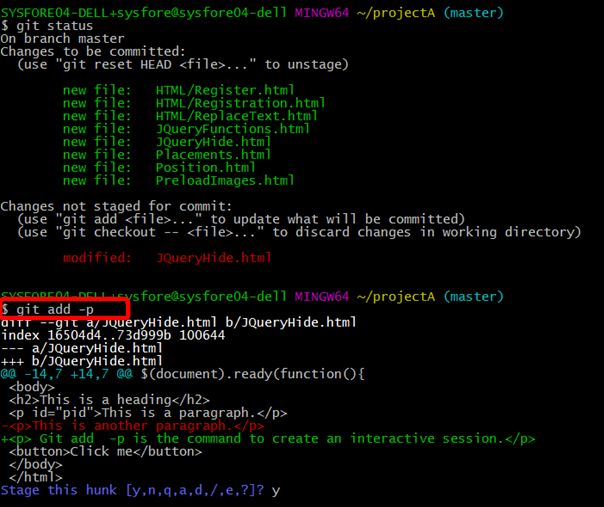


Use **git add .**to add all the files in the current working directory to the staging area.



**git add –p** Begin an interactive staging session to choose portions of a file to add to the next commit. This will present with a chunk of changes and prompt for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

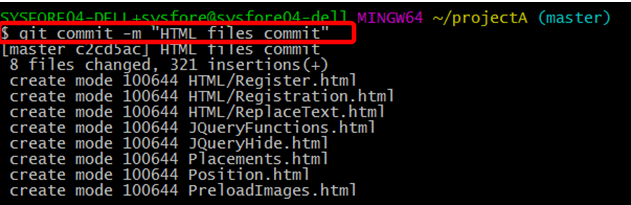
The scenario below is possible only when staged files have undergone some changes and if you want to choose between the updations done use -p option with git add to decide.



# Commiting staged files to project history

**git commit** command commits the staged snapshot to the project history. Committed snapshots are in the local repository , can be thought of as “safe” versions of a project , Git will never change them unless explicity asked to. Along with **git add**, this is one of the most important Git commands.

**git commit -m "HTML files Commit"**



# Switch Context with Stashing

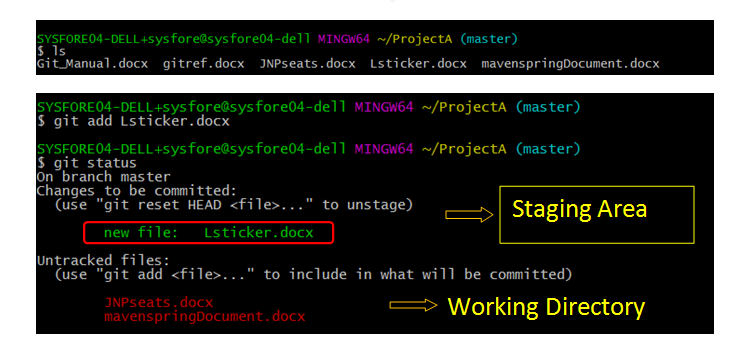
git stash temporarily shelves (or stashes) changes made to working copy so you can work on something else, and then come back and re -apply them later on. Stashing is handy to quickly switch context and work on something else, but if we're mid-way through a code change and aren't quite ready to commit.

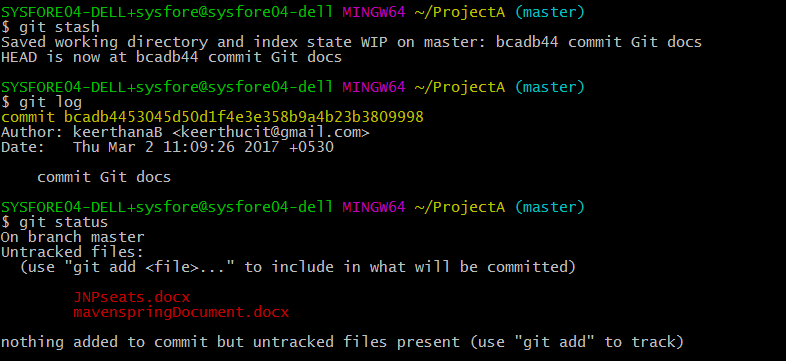
* Stashing Work
* [Re-applying stashed changes](https://www.atlassian.com/git/tutorials/git-stash#re-applying-your-stashed-changes)
* [Stashing untracked or ignored files](https://www.atlassian.com/git/tutorials/git-stash#stashing-untracked-or-ignored)
* [Managing multiple stashes](https://www.atlassian.com/git/tutorials/git-stash#managing-multiple-stashes)
* [Viewing stash diffs](https://www.atlassian.com/git/tutorials/git-stash#viewing-stash-diffs)
* [Partial stashes](https://www.atlassian.com/git/tutorials/git-stash#partial-stashes)
* [Creating a branch from stash](https://www.atlassian.com/git/tutorials/git-stash#creating-a-branch-from-your-stash)
* [Cleaning up stash](https://www.atlassian.com/git/tutorials/git-stash#cleaning-up-your-stash)
* [How git stash works](https://www.atlassian.com/git/tutorials/git-stash#how-git-stash-works)

**Stashing Work**

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

**projectA** repository created locally got the files listed using is command.

**git status** command lists the status of working directory and staging area but does not show any information regarding the committed project history.

Now apply git stash and check the git status ,

Check with ls command for the files that are stashed from staging area is not shown.



**lsticker.docx** is not listed after stash.

**git log** command lists commit history – single commit is been done shown with id bcad.....998

After git stash file Lsticker.docx is stashed from the staging area.

Compare the output of git status prior stashing and after stashing.

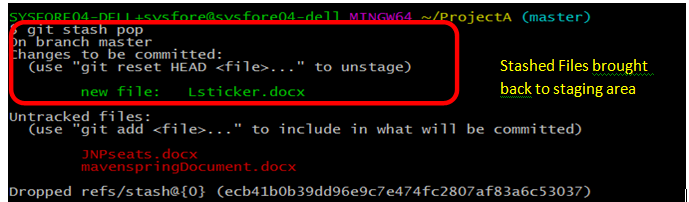
**git status** command does not list the staged area information after **git stash.**

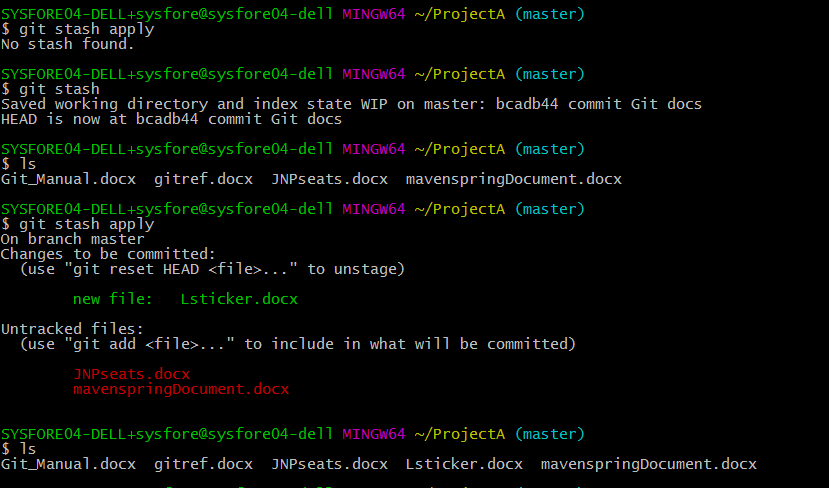
At this point it is free to make changes, create new commits, switch branches, and perform any other Git operations. Come back and re-apply stash when ready.

Note that the stash is local to Git repository . Stashes are not transferred to the server when we perform push.

[**Re-applying stashed changes**](https://www.atlassian.com/git/tutorials/git-stash#re-applying-your-stashed-changes)

Reapply previously stashed changes with **git stash pop**:



Popping your stash removes the changes from your stash and reapplies them to working copy.Alternatively, you can reapply the changes to your working copy and keep them in your stash with **git stash apply**:

This is useful if you want to apply the same stashed changes to multiple branches.

With the basics of stashing, there is one caveat with **git stash**  is that need to be aware of , by default Git **won't**stash changes made to untracked or ignored files.

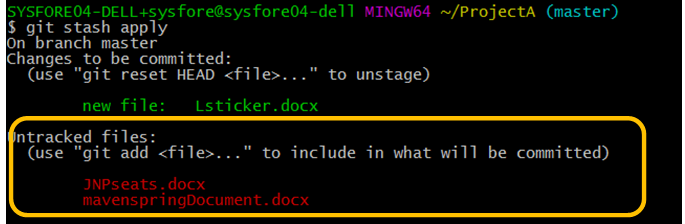
[**Stashing untracked or ignored files**](https://www.atlassian.com/git/tutorials/git-stash#stashing-untracked-or-ignored)

By default, running **git stash** will stash:

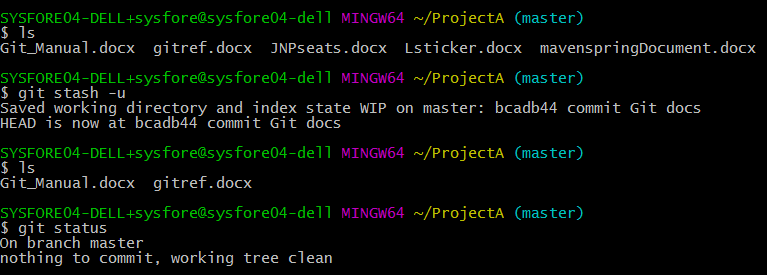
* changes that have been added to your index (staged changes)
* changes made to files that are currently tracked by Git (unstaged changes)

But it will **not** stash:

* new files in your working copy that have not yet been staged
* files that have been [**ignored**](https://www.atlassian.com/git/tutorials/gitignore)

So if we add a third file to our example above, but don't stage it (i.e. we don't run **git add), git stash** won't stash it. 

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files:



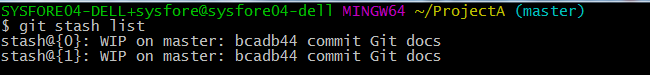
You can find the files not staged or untracked files **, JNpseats.docx and mavenspringDocument.docx is** stashed and not shown with ls command. After applying –u option with **git stash.**

Also by passing the -a option (or --all) when running git stash, include changes to [ignored](https://www.atlassian.com/git/tutorials/gitignore) files.

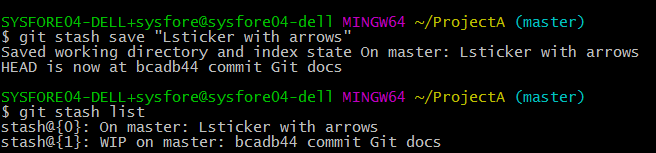
[**Managing multiple stashes**](https://www.atlassian.com/git/tutorials/git-stash#managing-multiple-stashes)

Git is not limited to a single stash. Git stash  can be run several times to create multiple stashes, and then use git stash list to view them. By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from. After a while it can be difficult to remember what each stash contains:

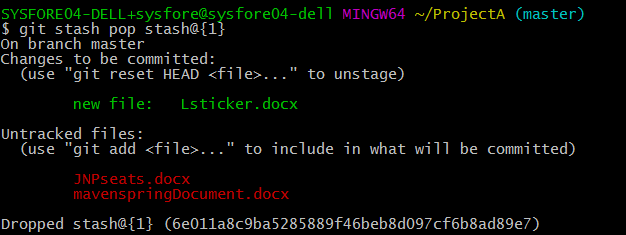
**git stash list** commandshelps to inspect the modifications stashed away by this command.



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

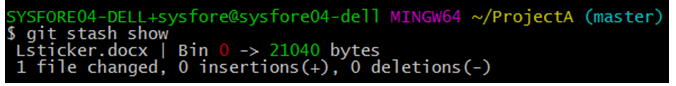


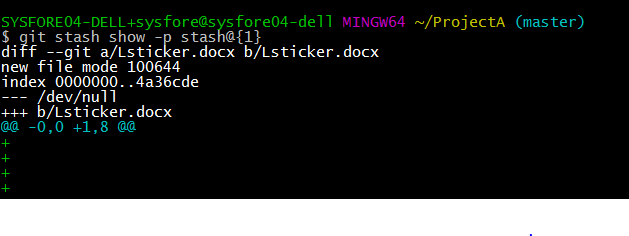
By default, **git stash pop** will re-apply the most recently created stash: **stash@{0}.**So,choose which stash to re-apply by passing its identifier as the last argument,



[**Viewing stash diffs**](https://www.atlassian.com/git/tutorials/git-stash#viewing-stash-diffs)

can view a summary of a stash with **git stash show**. Show <stash> Show the changes recorded in the stash as a diff between the stashed state and its original parent. When no <stash> is given, shows the latest one .

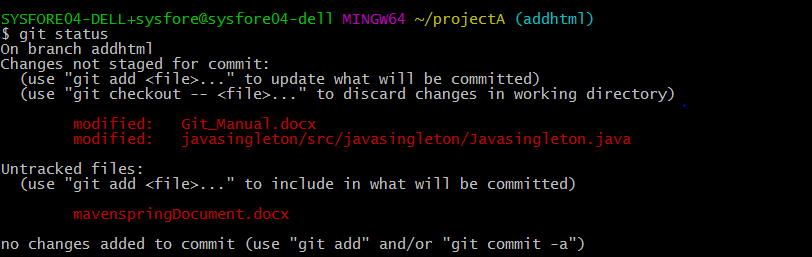


By default, the command shows the diffstat, but it will accept any format known to **git diff** (e.g., git stash show -p stash@{1} to view the second most recent stash in patch form). 

[**Partial stashes**](https://www.atlassian.com/git/tutorials/git-stash#partial-stashes)

We can choose to stash just a single file, a collection of files, or individual changes from within files. If you pass the -p option (or --patch) to git stash, it will iterate through each changed "hunk" in your working copy and ask whether you wish to stash it:

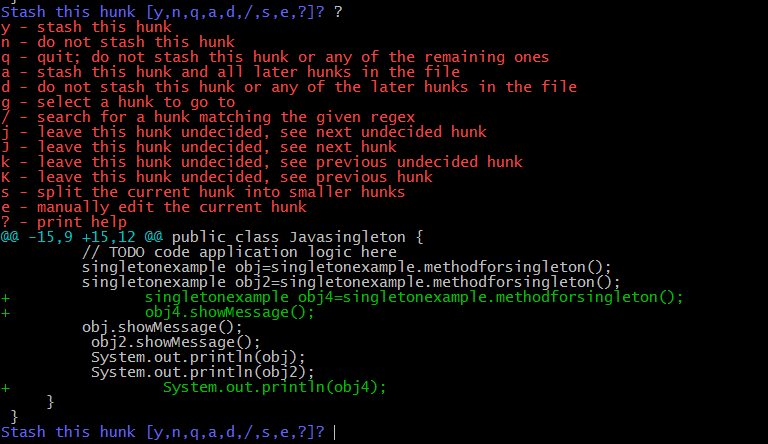
Edit javasingleton.java file and check with **git status**



Now apply **git stash -p**





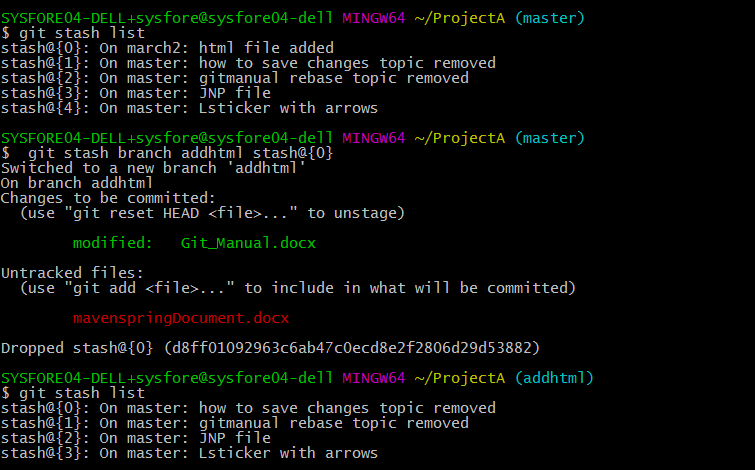
We can hit ? for a full list of hunk commands. Commonly useful ones are:

There is no explicit "abort" command, but hitting CTRL-C(SIGINT) will abort the stash process.

[**Creating a branch from stash**](https://www.atlassian.com/git/tutorials/git-stash#creating-a-branch-from-your-stash)

If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to:

**git stash branch addhtml stash@{0}**



This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it.

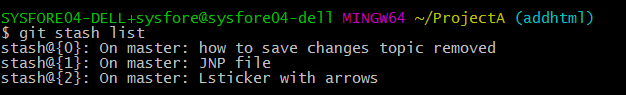
[**Cleaning up stash**](https://www.atlassian.com/git/tutorials/git-stash#cleaning-up-your-stash)

If you decide you no longer need a particular stash, you can delete it with **git stash drop**:

**git stash drop stash@{1}**

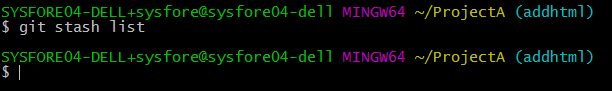


Or you can delete all of your stashes with:



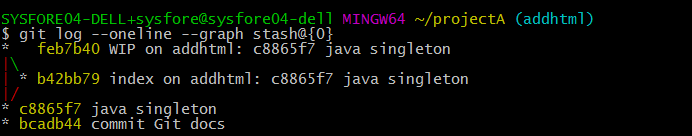
**git stash clear**





[**How git stash works**](https://www.atlassian.com/git/tutorials/git-stash#how-git-stash-works)

Stashes are actually encoded in your repository as commit objects. The special ref at .git/refs/stash points to your most recently created stash, and previously created stashes are referenced by the stash ref's reflog. So, why we refer to stashes by stash@{n}: Actually we are referring to the nth reflog entry for the stash ref. Since a stash is just a commit, you can inspect it with **git log**: **git log –oneline –graph stash@{0}**



# Ignored Files - .gitignore

Git sees every file in your working copy as one of three things:

1. tracked - a file which has been previously staged or committed;
2. untracked - a file which *has not* been staged or committed; or
3. ignored - a file which Git has been explicitly told to ignore.

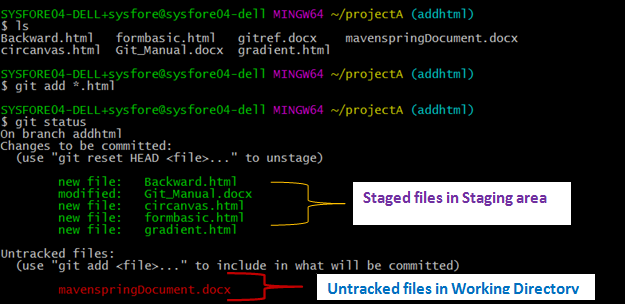
Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

* dependency caches, such as the contents of /node\_modules or /packages
* compiled code, such as .o, .pyc, and .class files
* build output directories, such as /bin, /out, or /target
* files generated at runtime, such as .log, .lock, or .tmp
* hidden system files, such as .DS\_Store or Thumbs.db
* personal IDE config files, such as .idea/workspace.xml

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. .gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

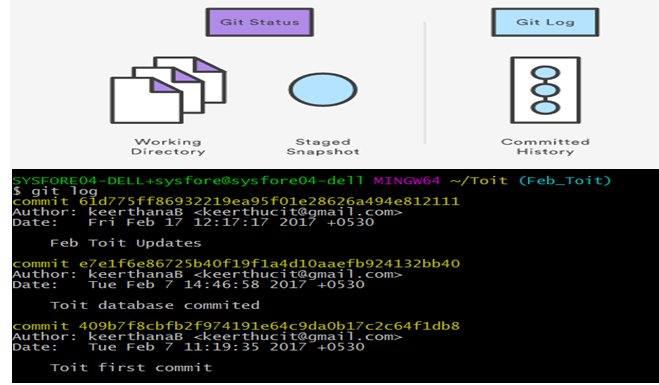
# Inspecting staged and untracked files

The **git status** command displays the state of the working directory and the staging area. It lets to see the changes have been staged, which haven’t, and which files aren’t being tracked by Git. Status output does not show any information regarding the committed project history for which we need to use [git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log) .



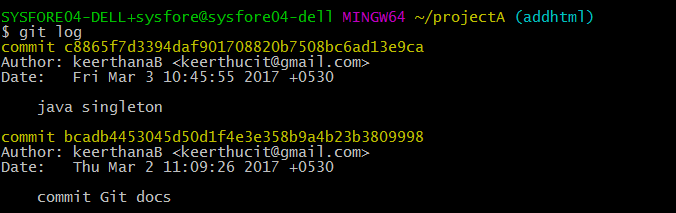
It simply shows what is going on with **git add** and **git commit**.

# Inspecting project history

The **git log** command displays committed snapshots. It lists the project history, filter it, and search for specific changes. While **git status** lets us to inspect the working directory and the staging area **, git** **log**only operates on the committed history.

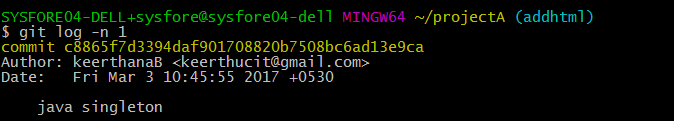
**git log**

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use **Space** to scroll and **q** to exit.



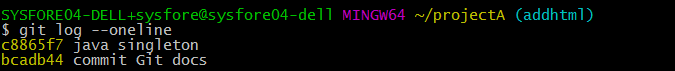
**git log –n <limit>**

Limit the number of commits by **<limit>.** For example, **git log -n 1** will display only 1 commits.



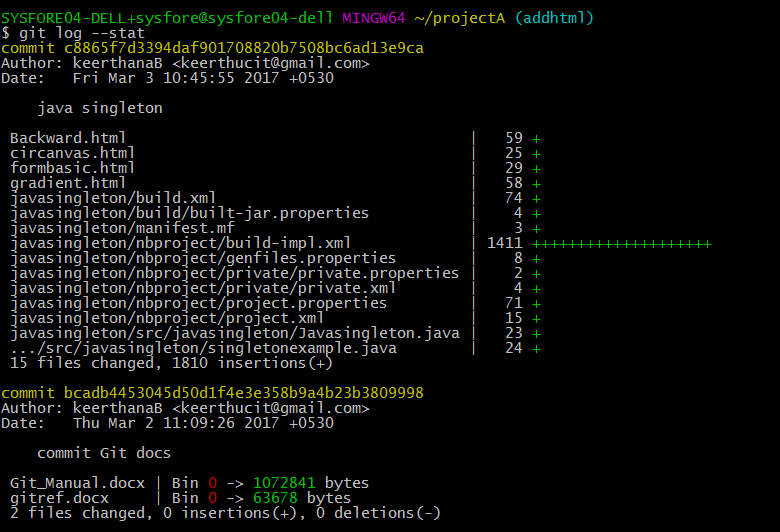
**git log --oneline**

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.



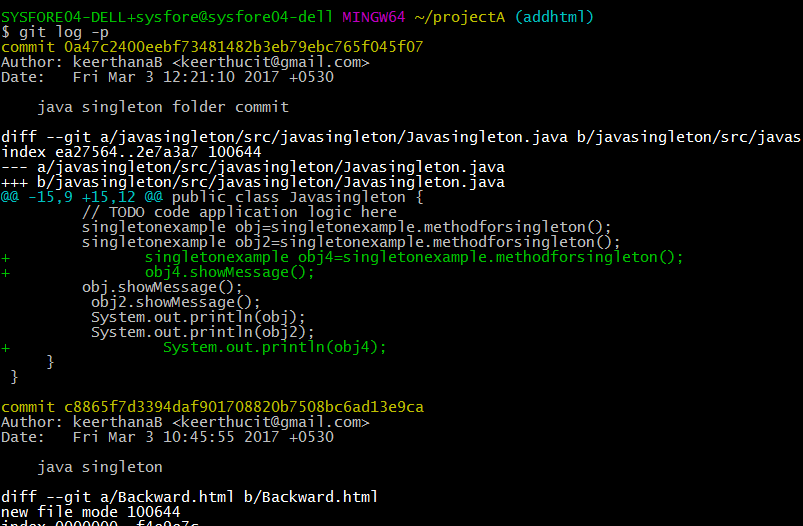
**git log --stat**

Along with the ordinary **git log** information, include which files were altered and the relative number of lines that were added or deleted from each of them.



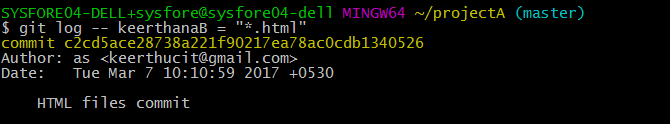
**git log -p**

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.



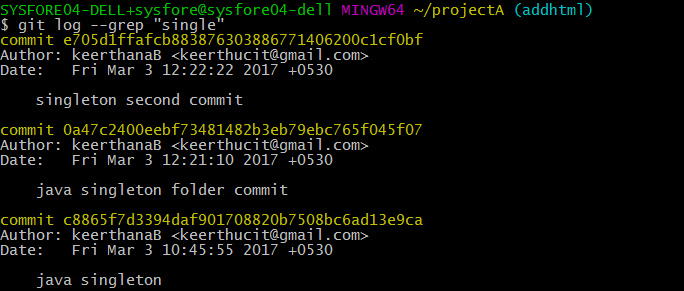
**git log --author=”<pattern>”**

Search for commits by a particular author. The **<pattern**> argument can be a plain string or a regular expression.



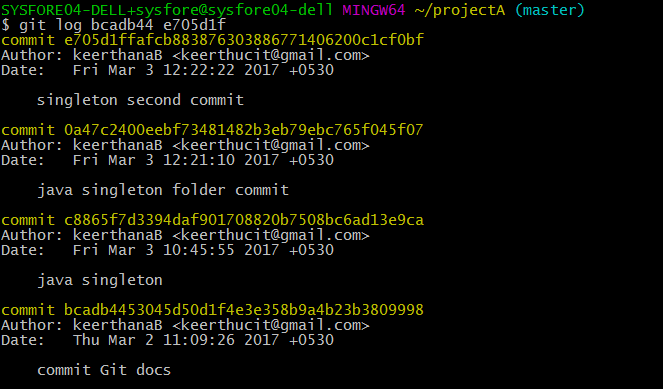
**git log --grep “<pattern>”**

Search for commits with a commit message that matches <**pattern>**, which can be a plain string or a regular expression.



**git log <since> . . <until>**

Show only commits that occur between **<since>**and**<until>.** Both arguments can be either a commit ID, a branch name, **HEAD,** or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).



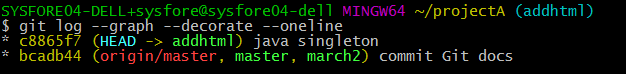
**git log <file>**

Only display commits that include the specified file. This is an easy way to see the history of a particular file.



**git log –graph –decorate --oneline**

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

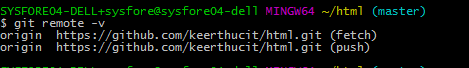


# Collaboration @ Git Hub - Syncing

Manage connections with other repositories, publish local history by "pushing" branches to other repositories, and check what others have contributed by "pulling" branches into your local repository. We can list the remote connections to other repositories using **git remote**

**git remote -v**

Lists the URL of each connection.



**git remote add <name> <url>**

Create a new connection to a remote repository. After adding a remote, you’ll be able to use <name> as a convenient shortcut for <url> in other Git commands.



In the Command prompt, [add the URL for the remote repository](https://help.github.com/articles/adding-a-remote) where your local repository will be pushed. The **git remote** command is an easier way to pass URLs to these "sharing" commands.

**The origin Remote**

In addition to origin, it’s often convenient to have a connection to our teammates’ repositories. For example, if it is a publicly accessible repository on https://github.com/keerthucit/ProjectA.git, add a connection as follows:

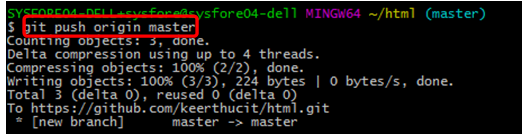
**git remote add projectA** [**https://github.com/keerthucit/projectA.git**](https://github.com/keerthucit/projectA.git)



Having this kind of access to individual developers’ repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

# Pushing into remote repo

**git push <remote> <branch>** command is to transfer commits from local repository to a remote repo. **git push <origin> <master>** use origin in place of remote and master if there is no branching.



**git push <remote> <branchname>** use origin in place of remote and branch name if branch exists.



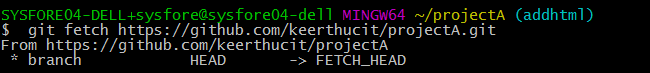
This creates a local branch in the destination repository.

# Fetching from remote repo

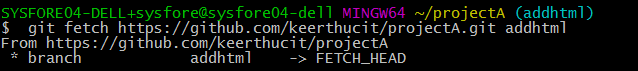
The git fetch command imports commits from a remote repository into local repo. The resulting commits are stored as remote branches instead of the normal local branches that we’ve been working with. This helps to review changes before integrating them into our copy of the project.

**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.



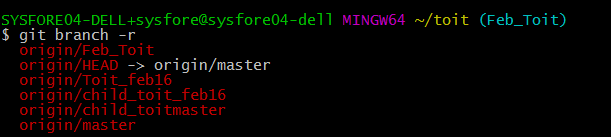
**git fetch <remote> <branch>**



Fetching is to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no effect on our local development work. This makes fetching a safe way to review commits before integrating them with your local repository

**Remote Branches**

Remote branches are just like local branches, represent commits from somebody else’s repository. Check out a remote branch just like a local one, but this puts in a detached **HEAD** state just like checking out an old commit .To view remote branches pass the **-r** flag to the **git branch** command. Remote branches are prefixed by the remote they belong to so that you don’t mix them up with local branches.The code snippet shows the branches after fetching from the origin remote:

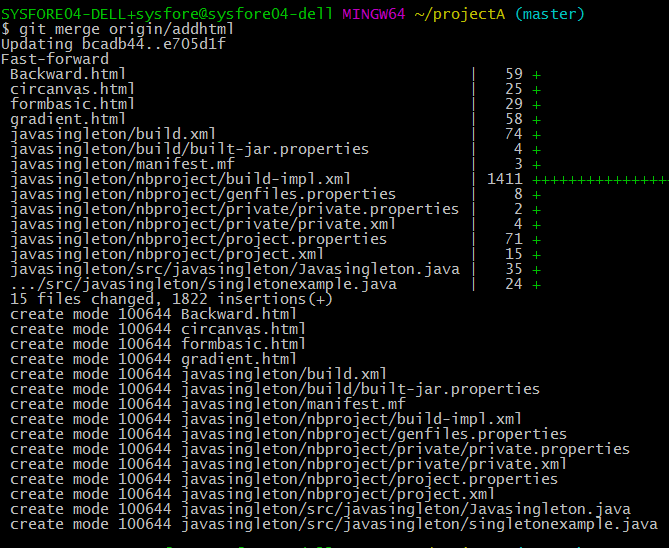


# Merging Upstream changes

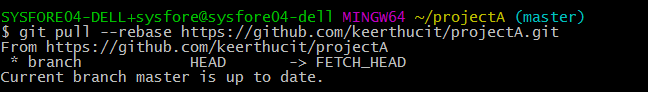
Merging upstream changes into our local repository is a common task in Git-based collaboration workflows. We can do this with [**git fetch**](https://www.atlassian.com/git/tutorials/syncing/git-fetch) followed by [**git merge**](https://www.atlassian.com/git/tutorials/using-branches/git-merge), but **git pull** rolls this into a single command **git pull <remote>**

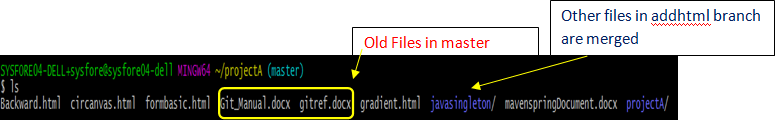
Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as **git fetch <remote>** followed by **git merge origin/<current-branch>.**





**git pull --rebase <remote>** - **git pull --rebase https://github.com/keerthucit/projectA.git**





Master branch is merged and updated with the branch addhtml.

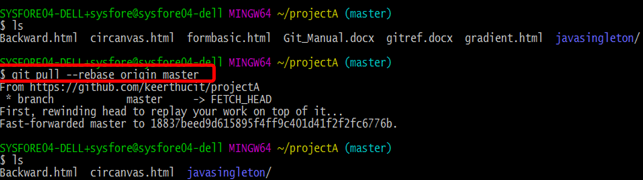
Same as the above command, but instead of using **git merge** to integrate the remote branch with the local one, use **git rebase.**

**Pulling via Rebase**

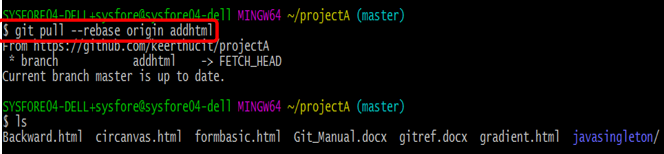
The example is to synchronize with the central repository's master branch using the **--rebase** option to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since this simply moves our local changes onto the top of what everybody else has already contributed.

**git checkout master**

**git pull - -rebase origin <branchname> -**Here, **git pull - -rebase origin <master>**

****

To rebase to a particular branch add that branch name ,**git pull - -rebase origin <addhtml>**

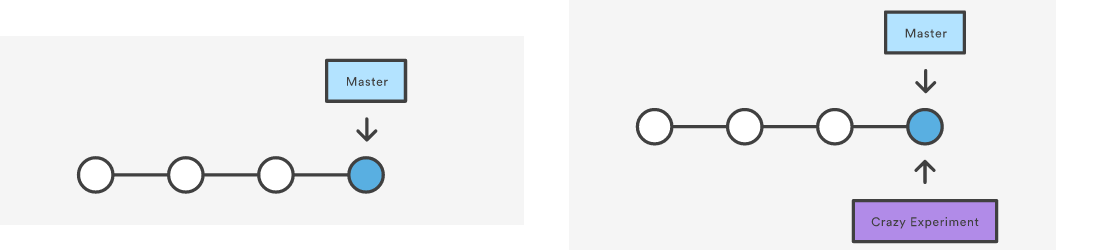
****

# Creating Branches

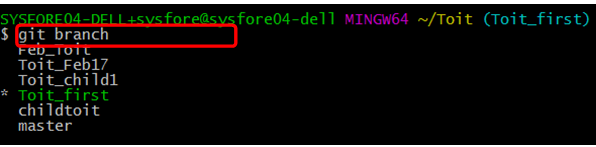
Branches are just *pointers* to commits. When creating branch, Git creates a new pointer—it doesn’t change the repository in any other way.

Start with a repository that looks like the figure in left and then create a branch using the following command , **git branch crazy-experiment**

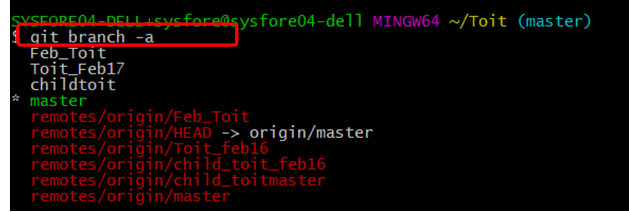
The repository history remains unchanged. All we get is a new pointer to the current commit:



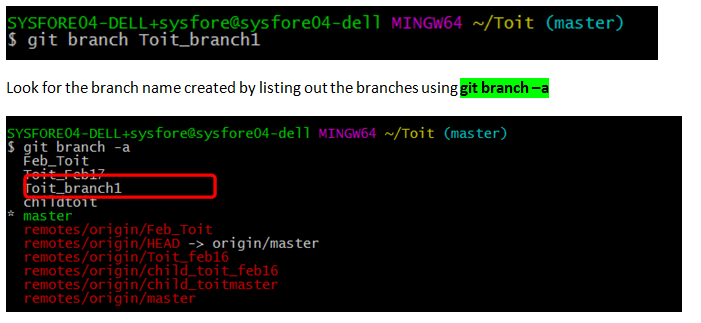
**git branch** List all of the branches in local repository.



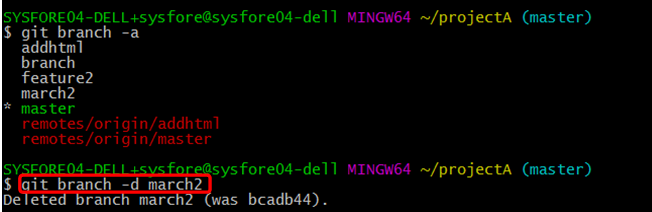
**git branch –a** List all of the branches in local as well as remote repository.



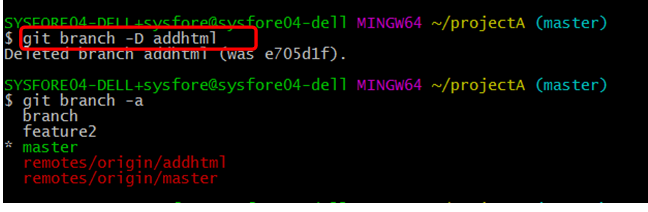
**git branch <branch>** Create a new branch called <branch>. This does *not* check out the new branch.

****

**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.

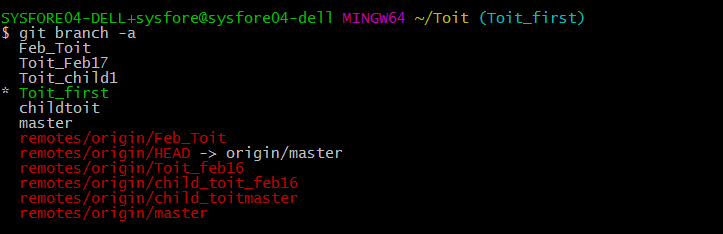


**git branch -D <branch>** Force delete the specified branch, even if it has unmerged changes. This is the command to permanently throw away all of the commits associated with a particular line of development. To remove a **remote branch** from the server, git push origin –delete <branch name > after git branch –D <branch name>

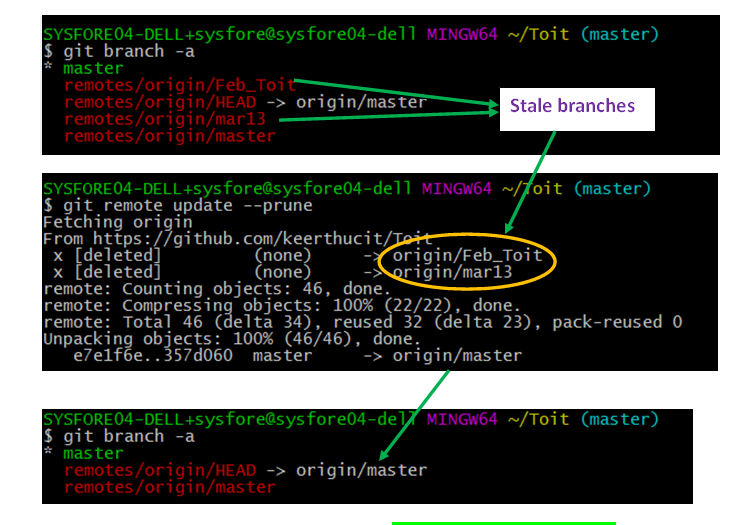


**git branch –m <branch>** Rename the current branch to <branch> after checkout



List out the branches using **git branch –a** and look for the renamed branch

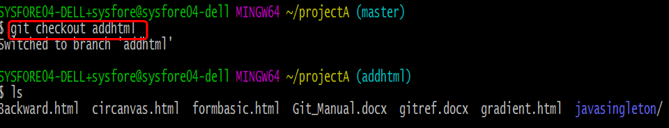
To remove all stale branches, **git remote update –prune**

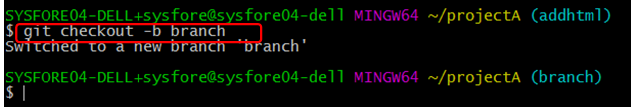
****To remove that particular remote-tracking branch, git branch -d -r origin/branchname

# Checking out from branch

The **git checkout**  command to check out branches is similar in that the working directory is updated to match the selected branch/revision. However, new changes are saved in the project history

**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>.  **

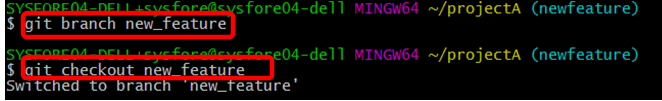
**git checkout -b <new-branch> <existing-branch>** is same as the above invocation, but base the new branch off of <existing-branch> instead of the current branch.

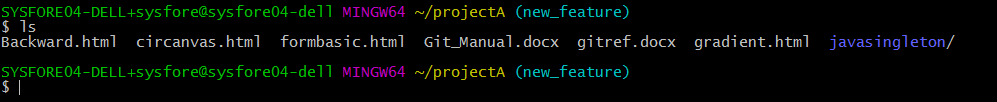
**git checkout** works hand-in-hand with **git branch**. To start a new feature, create a branch with **git branch**, then check it out with **git checkout**. You can work on multiple features in a single repository by switching between them with **git checkout**

The following example demonstrates the basic Git branching process. When you want to start working on a new feature, create a dedicated branch and switch into it:

**git branch new-feature -** Creates branch and needs to checkout that branch using checkout command

**git checkout new-feature**

****



Now we can commit new snapshots just like we’ve seen in previous modules:

**# Edit some files**

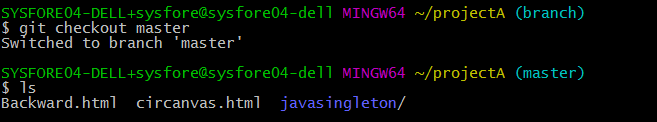
**git add <file>**

**git commit –m “started work on a new feature”**

**#Repeat**

All of these are recorded in **new-feature**, which is completely isolated from **master.** We can add as many commits here as necessary without worrying about the rest of our branches. When it’s time to get back to “official” code base, simply check out the **master**branch:

**git checkout master**



This shows the state of the repository before started new feature. From here, options are there to merge in the completed feature, branch off a brand new, unrelated feature, or do some work with the stable version of our project.

# Integrating into single branch

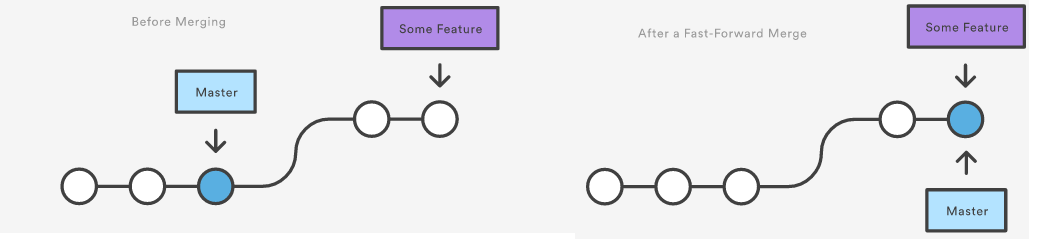
The **git merge**command takes the independent lines of development created by [**git branch**](https://www.atlassian.com/git/tutorials/using-branches/git-branch) and integrate them into a single branch.

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**](https://www.atlassian.com/git/tutorials/using-branches/git-checkout)for selecting the current branch and **git branch -d** for deleting the obsolete target branch.

**git merge <branch>**

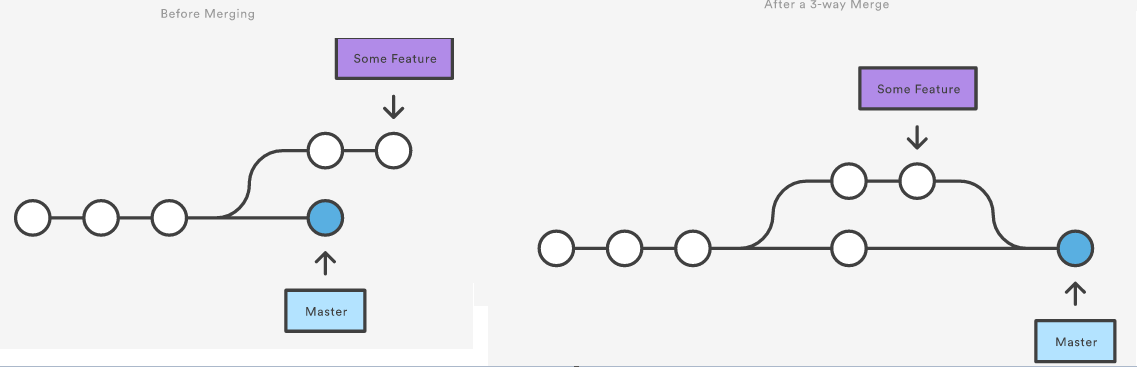
Merge the specified branch into the current branch, but always generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in our repository.

Once you’ve finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of our repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. 



However, a fast-forward merge is not possible if the branches are 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, a dedicated commit to tie together the two histories.Git uses **three** commits to generate the merge commit: the two branch tips and their common ancestor.



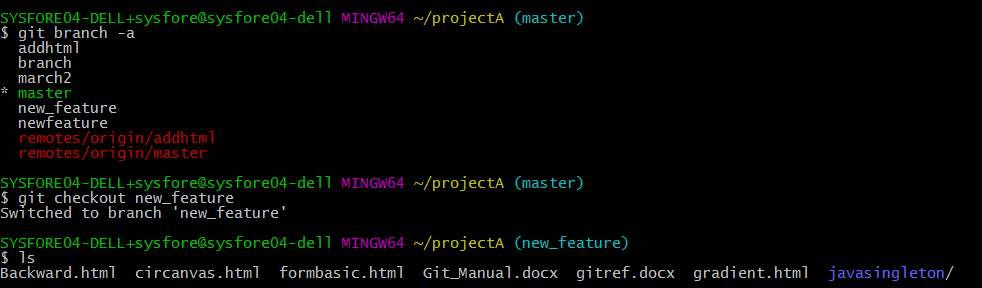
**Resolving Conflicts**

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

Merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

**Fast-Forward Merge Example**

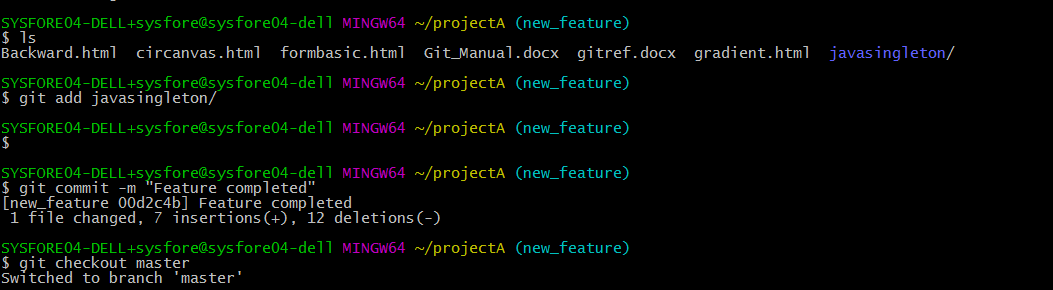
The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.



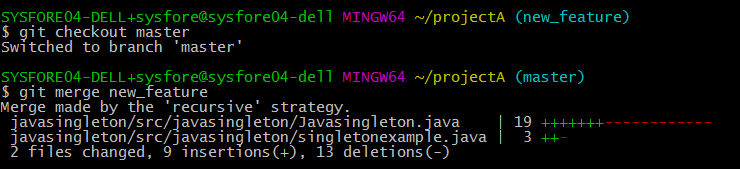
Edited the file under javasingleton folder ,staged and commited

# Commit 1- “new Feature started”

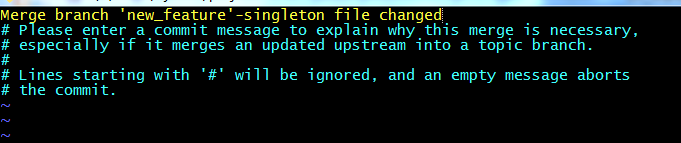
# Commit 2- “Feature Completed”



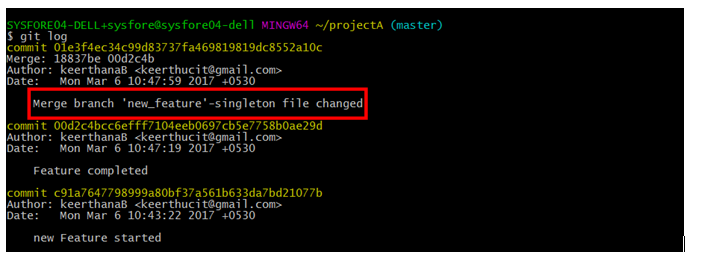
Now ,checkout master and perform fastforward merge using git merge command.



Git merge command prompts for commit message with editor as shown:

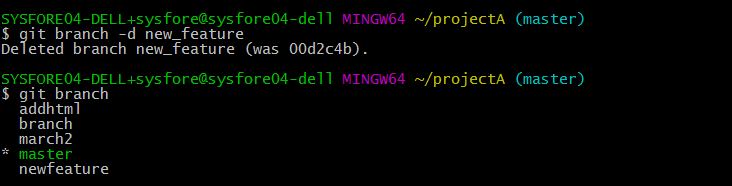


Git log shows the merge commit.

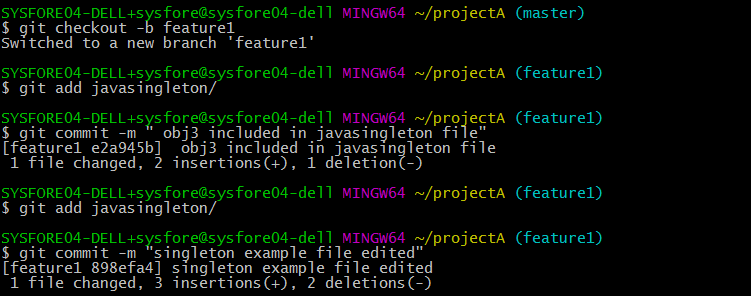


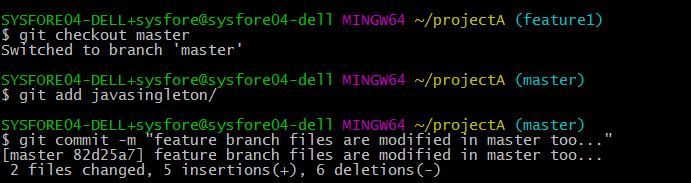
This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Now git need not maintain the branch **new-feature** . Use **git branch –d** to remove the branch. Since **new-feature** is now accessible from the master branch.

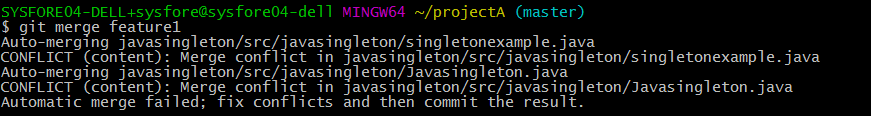


**3-Way Merge**

Create and switch to feature1 branch with **git checkout –b feature1**.Edit and stage the singleton files using **git add** and commit using **git commit –m “<message”>** 

Switch to master and add new features,stage and commit the changes.

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move **master**up to **feature1** without backtracking.



For most workflows, **feature1** would be a much larger feature that took a long time to develop, which would be why new commits would appear on **master**in the meantime. If feature branch was actually as small as the one in the above example, better can be rebased it onto **master**and can do a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

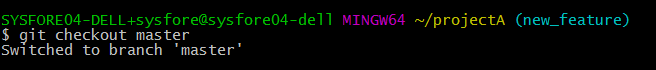
# Undoing changes

The **git checkout** command serves three distinct functions: checking out files, checking out commits, and checking out branches.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

**git checkout master**

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the “current” state of the project.



As a continuation of example under 3-way merge ,use **git log –oneline** and checkout the commit files



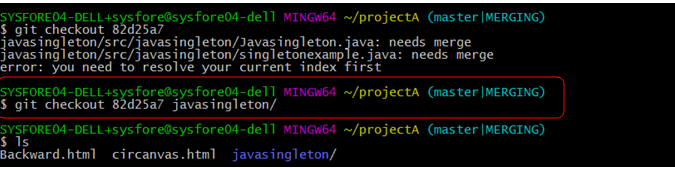
**git checkout <commit>**

Update all files in the working directory to match the specified commit. We can use either a commit hash or a tag as the **<commit>**argument. This will put you in a detached HEAD state.

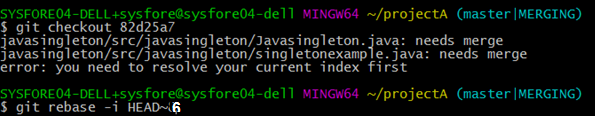
**git checkout <commit> <file>**

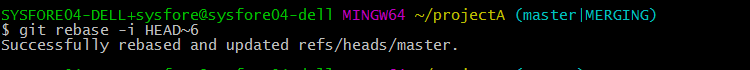
Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

Either edit the files by **git checkout <commit> <file>**

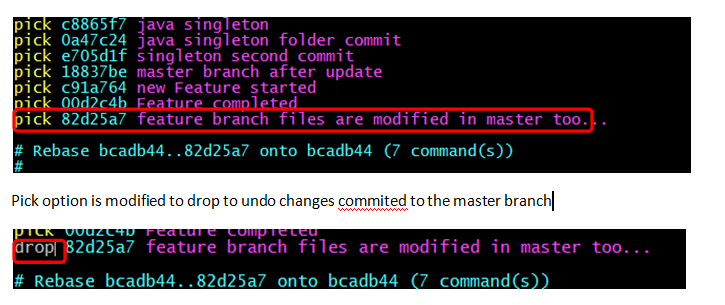


Or perform rebase and apply changes and set the file compatible for merging.





Git rebase interactive command will open the editor ,to apply changes.

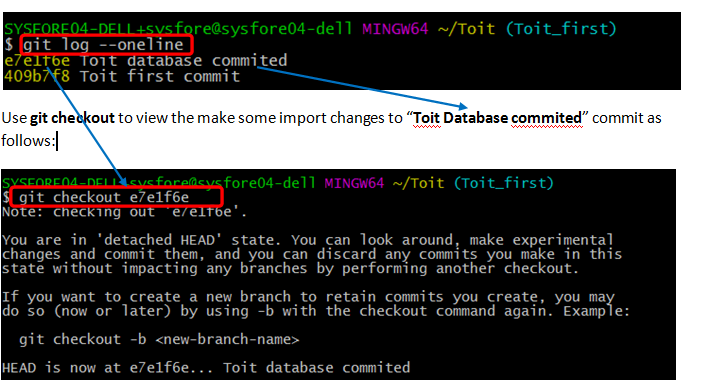


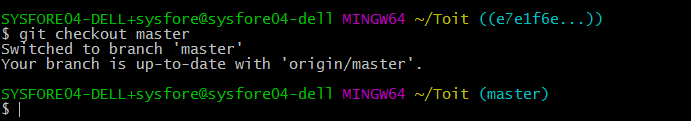
**git checkout** is an easy way to “load” any of the saved snapshots onto development machine. The HEAD usually points to master or some other local branch, but when we check out a previous commit, **HEAD** no longer points to a branch—it points directly to a commit. This is called a “**detached HEAD**” state.

**Viewing an Old Revision**

To take a look at the state of the project before starting our work we need to find the ID of the revision.

**git log - - oneline -** List theproject history

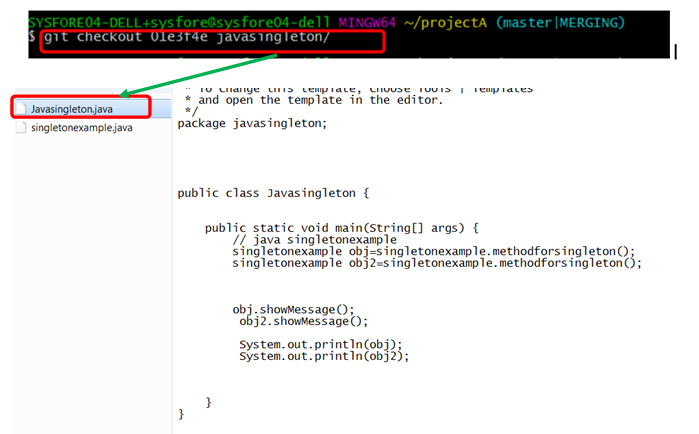
****

This makes working directory match the exact state of the **e7e1f6e** commit. We can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project.  Now, get back to the “current” state of your project with **git checkout master** 

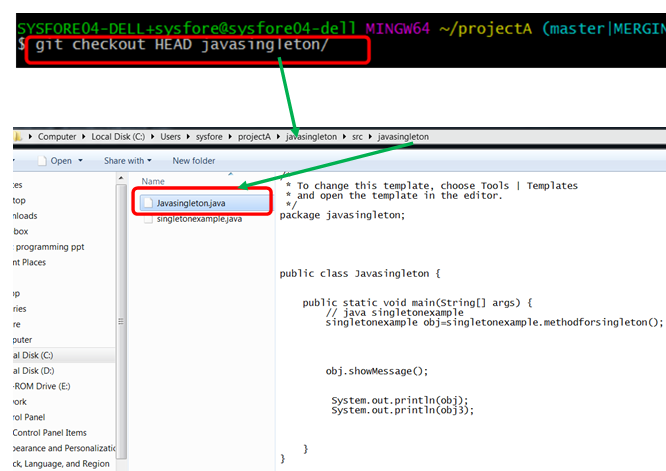
After getting back to the master branch, use either **git revert** or **git reset** to undo any undesired changes.

**Checking Out a File**

To check a single file or files from a folder, use **git checkout** to fetch an old version of it , use the following command: **git checkout <commit id> <folder/filename>**

****

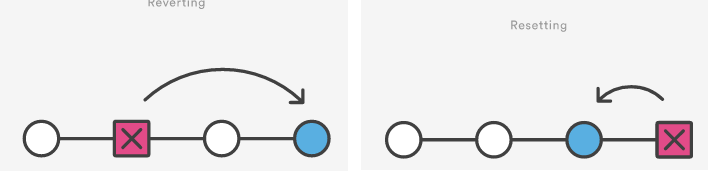
Remember, unlike checking out a commit, this does affect the current state of our project. The old file revision will show up as a “Change to be committed,” giving the opportunity to revert back to the previous version of the file. If we don’t want to keep the old version, check out the most recent version with the following: **git checkout HEAD <folder/filename>**

****

# Reverting changes

**git revert <commit>** Generate a new commit that undoes all of the changes introduced in **<commit>,** then apply it to the current branch. Instead of removing the commit from the project history appends a new commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

**Reverting vs. Resetting**

**git revert** undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. 

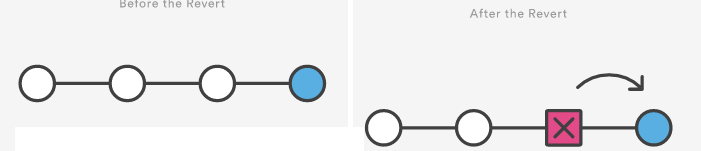
Reverting has two important advantages over resetting.

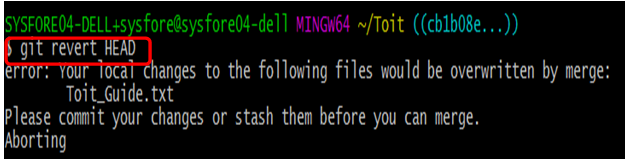
First, it doesn’t change the project history, which makes it a “safe.

Second, **git revert** is able to target an individual commit at an arbitrary point in the history, whereas **git reset** can only work backwards from the current commit. For example, if you wanted to undo an old commit with **git reset**, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution.

**Example**

The following example is a simple demonstration of git revert. It commits a snapshot, then immediately undoes it with a revert.



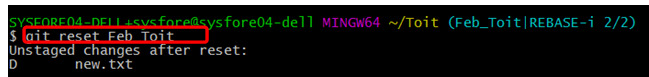
Note that the 4th commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 3rd and 5th commits represent the exact same code base, and the 4th commit is still in our history just in case we want to go back to it down the way.

# Resetting changes

If **git revert** is a “safe” way to undo changes, **git reset** is the dangerous method. Like [**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-checkout)**,  git reset** is a versatile command with many configurations, used to remove committed snapshots, u undo changes in the staging area and the working directory. In either case, it should only be used to undo local changes—we should never reset snapshots that have been shared with other developers.

**git reset <file>** Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

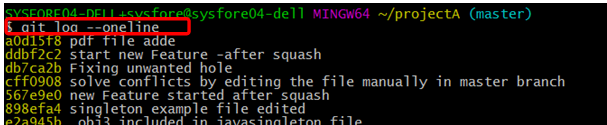
**git reset** Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, gives the opportunity to re-build the staged snapshot from scratch.

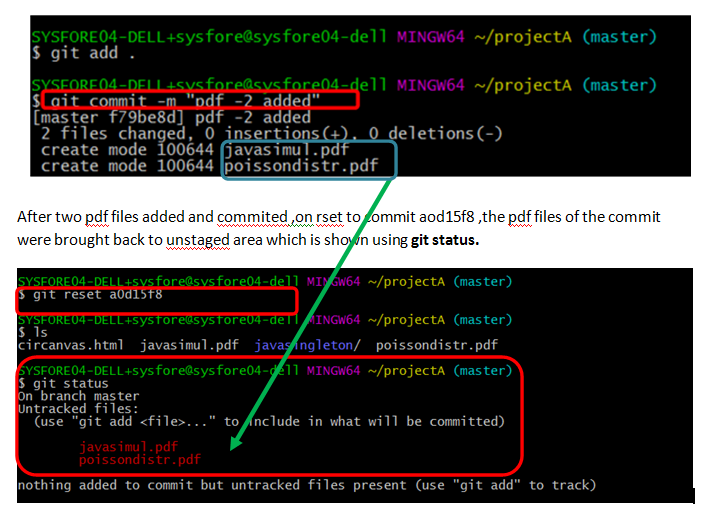


**git reset - - hard** Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the **--hard** flag tells Git to overwrite all changes in the working directory, too.



**git reset <commit>** Move the current branch tip backward to **<commit>,** reset the staging area to match, but leave the working directory alone. All changes made since **<commit>** will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.



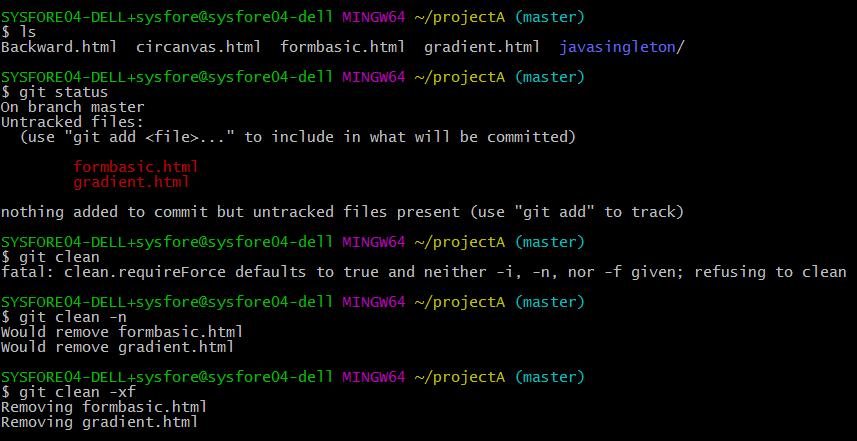


**git reset - - hard <commit>**

Move the current branch tip backward to **<commit>** and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after **<commit>**, as well.

# Permanent purging

The **git clean** command removes untracked files from your working directory. This is really more of a convenience command, since it’s trivial to see which files are untracked with **git status** and remove them manually. Like an ordinary **rm** command,  **git clean** is **not**undoable, so make sure before deleting the untracked files.

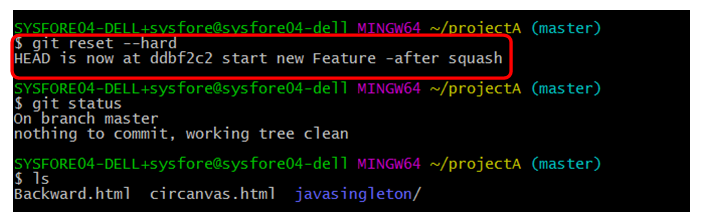


**git clean –n** Perform a “dry run” of git clean. This will show you which files are going to be removed without actually doing it.

**git clean –df** Remove untracked files and untracked directories from the current directory.

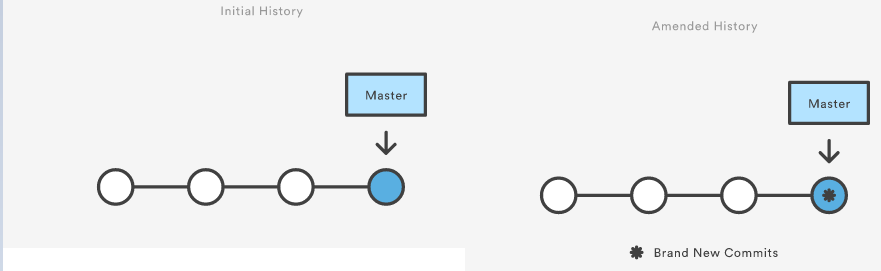
**git clean –xf** Remove untracked files from the current directory as well as any files that Git usually ignores.

The **git clean** command is often executed in conjunction with **git reset --hard**. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.



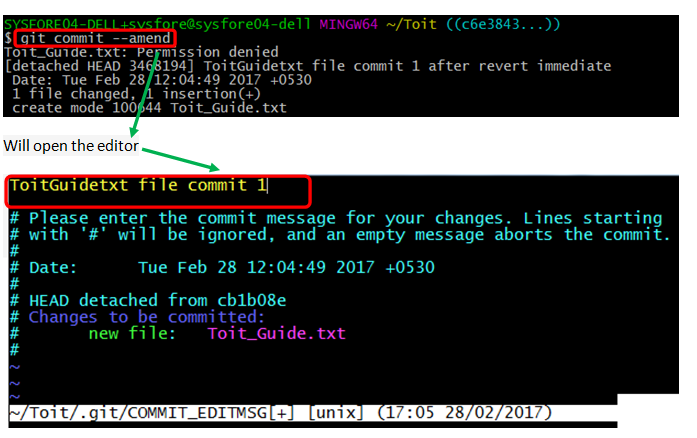
# Rewriting history

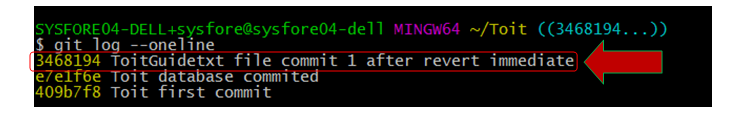
The git commit --amend command is a convenient way to fix up the most recent commit. It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot.



But, amending doesn’t just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram above. It’s important to keep this in mind when working with public repositories.

**git commit – amend** will open the editor like below to combine the staged changes with the previous commit and replace the previous commit with the resulting snapshot. Running this when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.



Check with **git log - -oneline**

# Rebasing branch

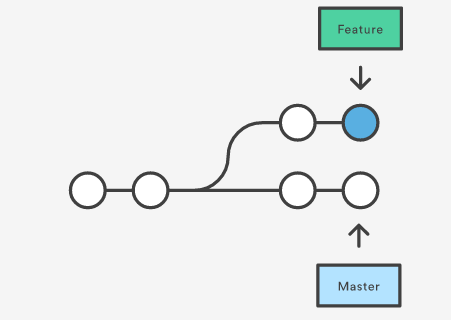
Rebasing is the process of moving a branch to a new base commit. The general process can be visualized as the following:



Git accomplishes this by creating new commits and applying them to the specified base—it’s literally rewriting our project history. It’s very important to understand that, even though the branch looks the same, it’s composed of entirely new commits.

**git rebase <base>** Rebase the current branch onto <base>, which can be any kind of commit reference (an ID, a branch name, a tag, or a relative reference to HEAD).

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature:



You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto master facilitates a fast-forward merge.



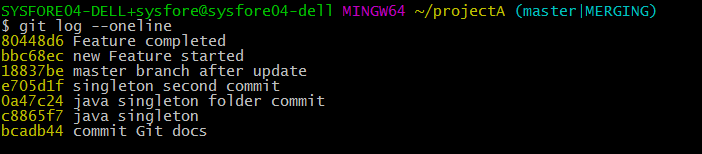
Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with **git merge** results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on what everybody has already done.”

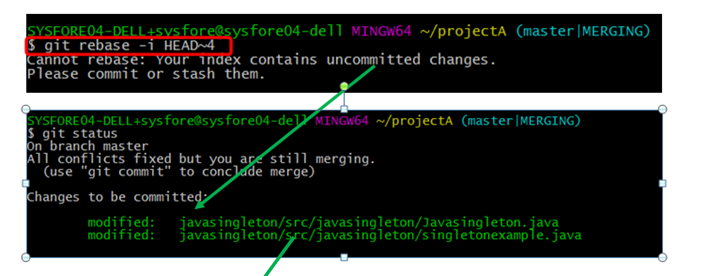
**Don’t Rebase Public History**

With **git commit --amend** and **git reset,** never rebase commits that have been pushed to a public repository. The rebase would replace the old commits with new ones, and it would look like that part of your project history abruptly vanished.

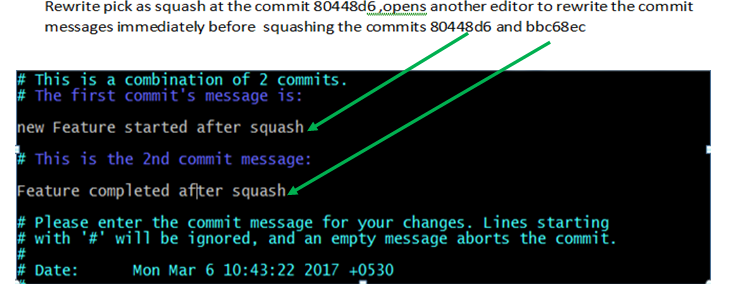
**Examples**

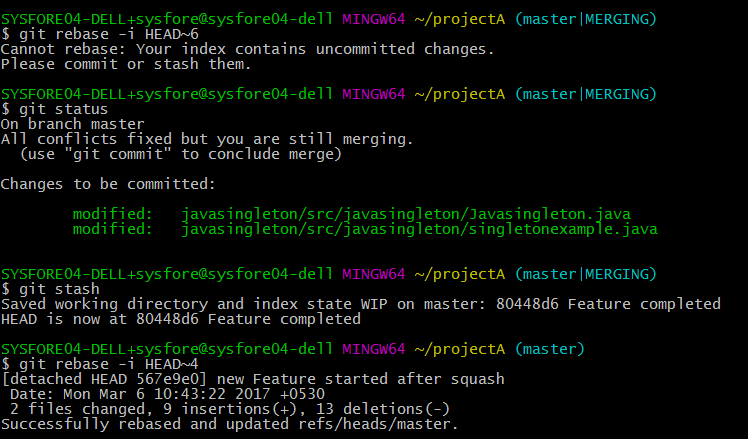
The example below combines git rebase with git merge to maintain a linear project history. This is a quick and easy way to ensure that your merges will be fast-forwarded.





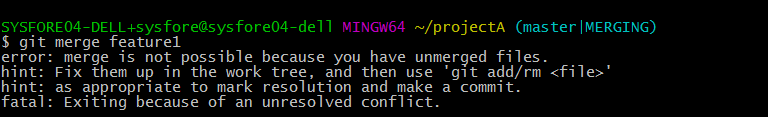


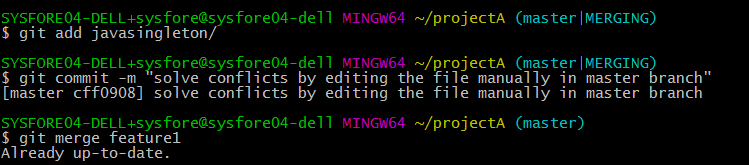


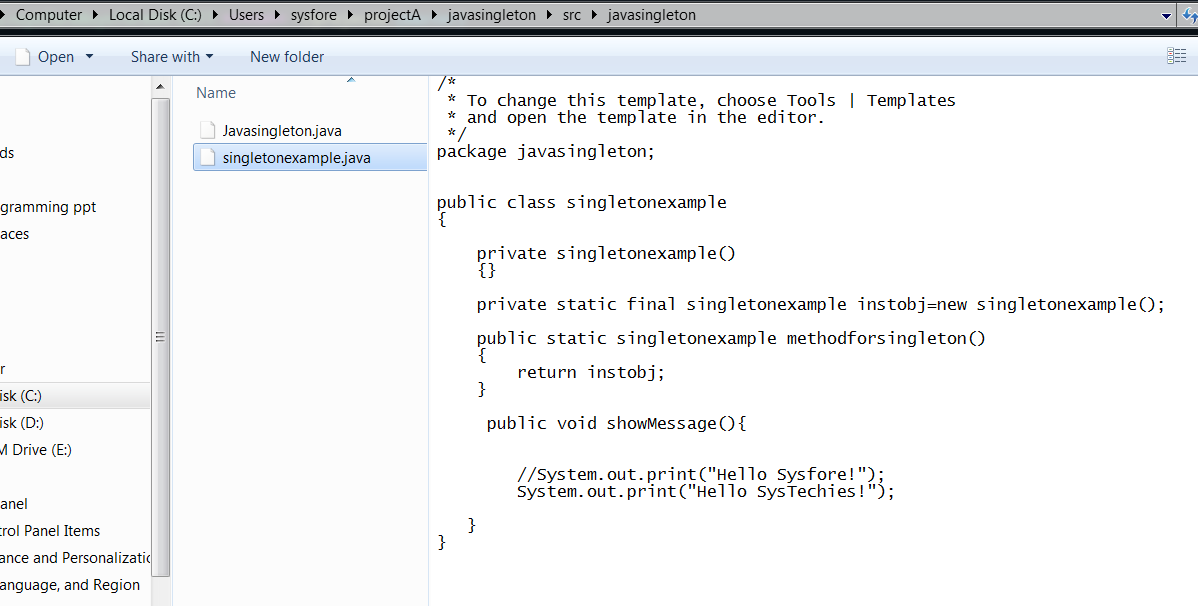


In the middle of our feature, we realize there’s a hole in our project, Solve the conflicts ,by editing the files and then apply merge commit. Before solving conflicts we cannot apply merge or jump to checkout another feature.

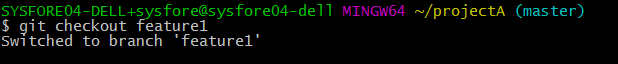


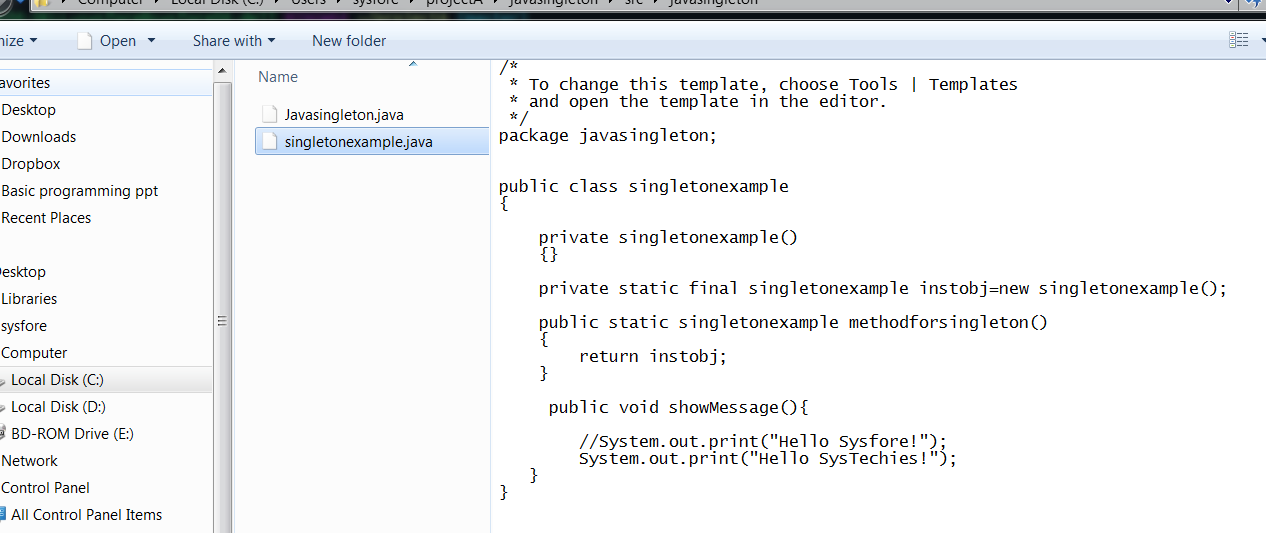


Edit the files, solve the conflict manually . Then stage the files,commit it in the project history and then merge feature branch with master.



Checkout feature1 branch files and check for update.

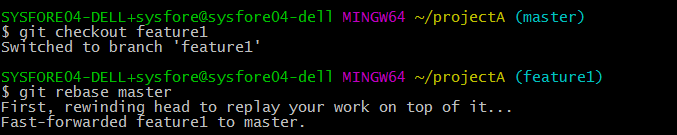




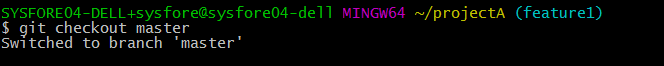
Now ,the branch can be deleted after fastforward merge.

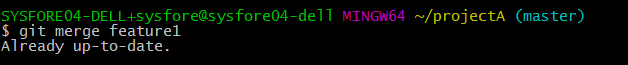


After merging the feature1 into master, we have a forked project history. Instead of a plain git merge, we can integrate the feature1 branch with a rebase to maintain a linear history:



This moves new-feature to the tip of master, which lets us do a standard fast-forward merge from master:

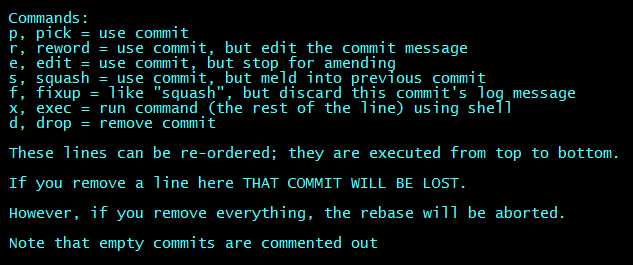




# Rebasing branch - interactive

Running git rebase with the -i flag begins an interactive rebasing session , gives the opportunity to alter individual commits in the process. This lets to clean up history by removing, splitting, and altering an existing series of commits. It’s like git commit --amend on steroids.

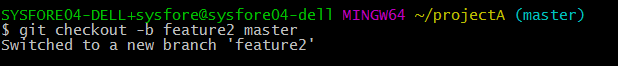
**git rebase –i <base>**



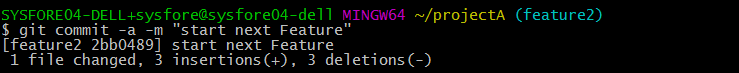
This gives the opportunity to squash insignificant commits, delete obsolete ones, and make sure everything else is in order before committing to the “official” project history.

**Example**

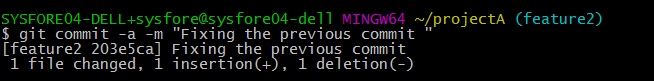
Start another feature branch feature2 using  **git checkout –b <branchname> <master branch>**



Edit singleton java file, stage the changes and commiting altogether with **git –a –m “<commit message>”**



Edit example file , stage the changes and commit again



Switch to master branch and edit some files, stage and commit the changes.



The last command will open an editor populated with the two commits from new-feature, along with some instructions:

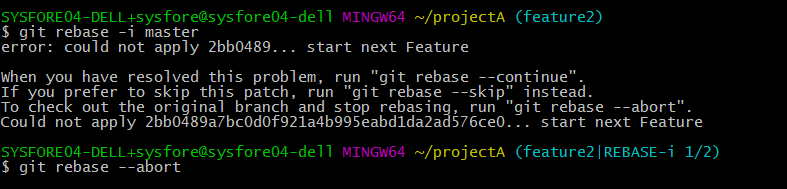


We can change the pick commands before each commit to determine how it gets moved during the rebase.We just combine the two commits with a squash command:



Use **git rebase –abort**, to skip rebase ,squash if any conflicts faced because of squash and then solve the conflict.

Edit the files , fix conflicts and then squash the commits by using git rebase –i master .

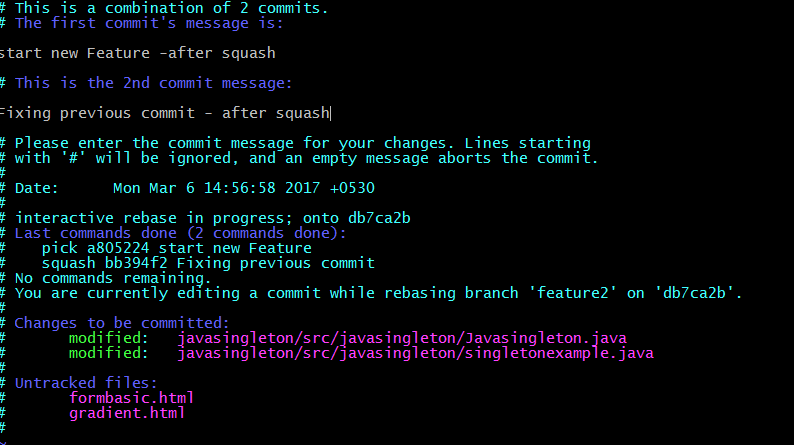


It opens the interactive editor , 

Edit **pick** into **squash** to meld with previous commit.

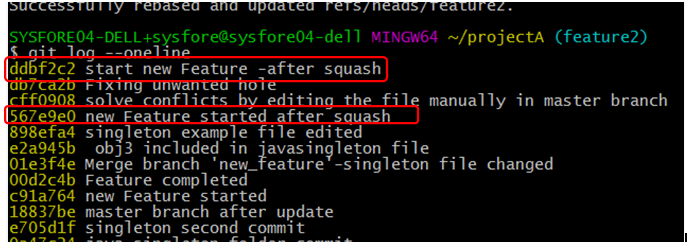


Save and close the editor to begin rebase. This will open another editor asking for the commit message for the combined snapshot. After defining the commit message, the rebase is complete and we can see the squashed commit in your git log output. This entire process can be visualized as follows:

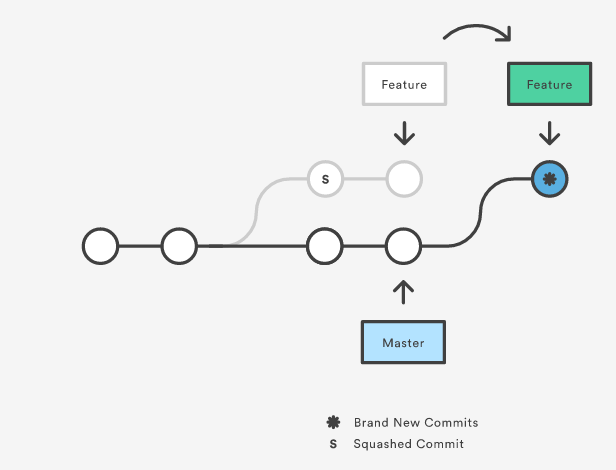


After closing the editor rebase will be completed successful and change of commit message can be checked using **git log --oneline**

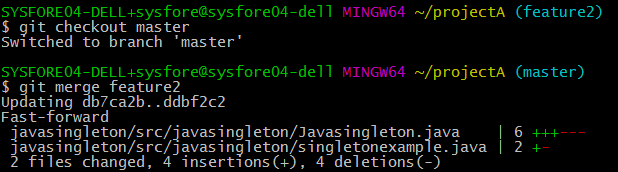




Note that the squashed commit has a different ID than either of the original commits, which tells us that it is indeed a brand new commit.



Finally, a fast-forward merge is done to integrate the polished feature branch into the main code base using **git checkout < master branch>** and **git merge <branch name>**



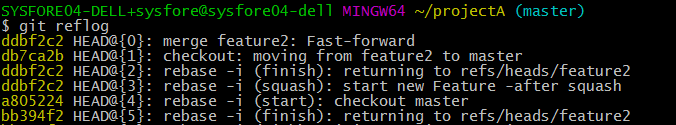
# Tracking history

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary.

**git reflog** Show the reflog for the local repository.

**git reflog - -relative-date** Show the reflog with relative date information (e.g. 2 weeks ago).

Every time the current HEAD gets updated (by switching branches, pulling in new changes, rewriting history or simply by adding new commits) a new entry will be added to the reflog.



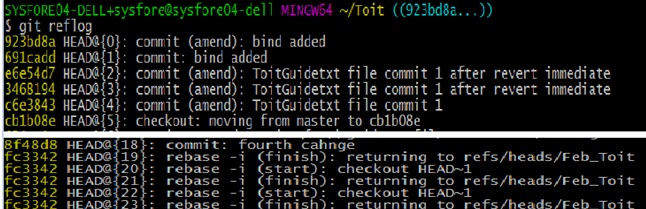
The reflog above shows the latest activity as represented at the top labeled HEAD@{0}.

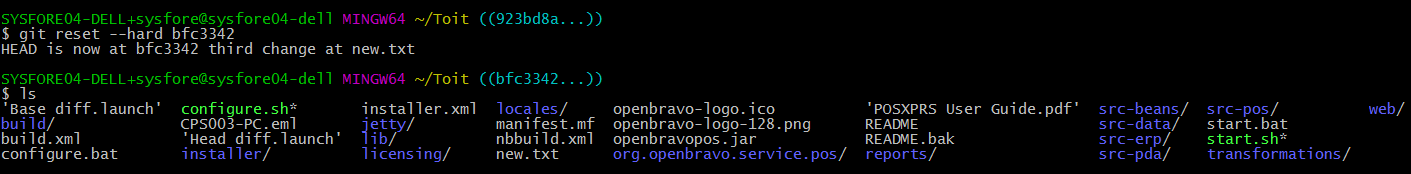
A checkout from feature 2 branch to the master and back represented with label HEAD@{1}

Using [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) it is then possible to change master back to the commit it was before. This provides a safety net in case history was accidentially changed.

It's important to note that the reflog only provides a safety net if changes have been commited to your local repository and that it only tracks movements.

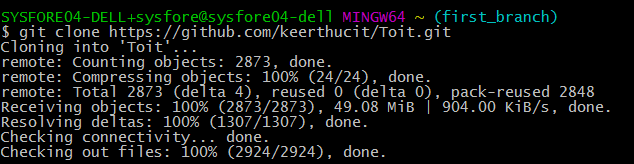
**git reset --hard bfc3342** The below scenario shows reflog and reset to commit **bfc3342**

****

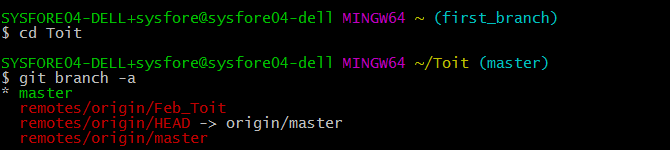


# Versioning with tags

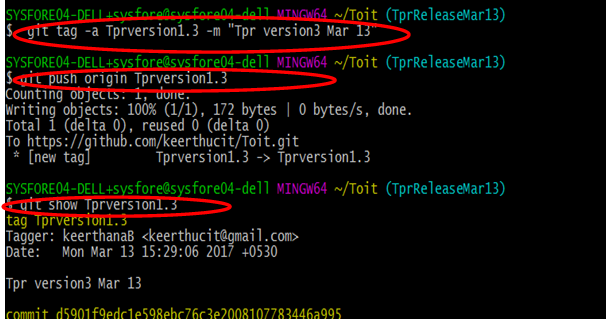
Checkout the code from github



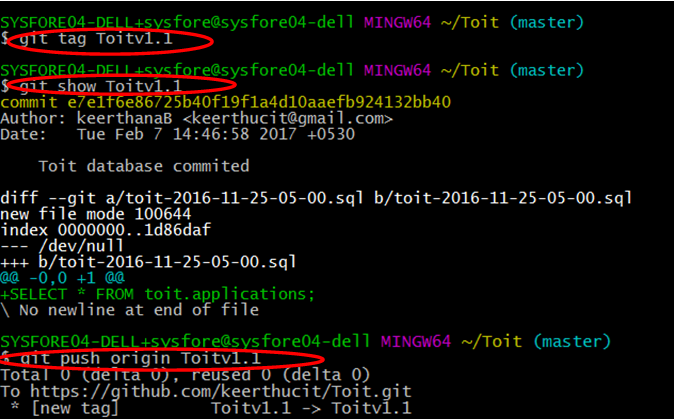
Check the master branch and other branches cloned to the local repo



Create an annotated tag by specifying –a option and –m to specify tag message which will be stored with the tag.



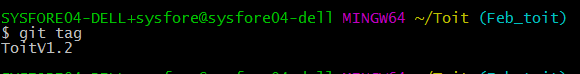
Create a lightweight tag to maintain versioning of your commit in the master branch and Push it to the remote using git push origin.



Checkout branch Feb\_Toit to create next version using tags .

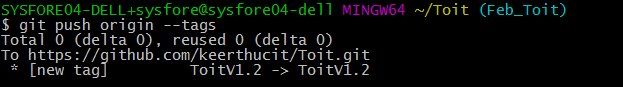


Create Toitv1.2 tag version for the branch Feb\_Toit commits



Check the same with show command

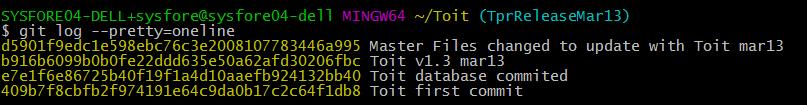
We can push all the tag versioning at a time using --tags option with git push origin



To checkout a tag version we need to create a new branch



To list out the commits with messages use git log - - pretty=oneline



# .gitignore

Git sees every file in your working copy as one of three things:

* tracked - a file which has been previously staged or committed;
* untracked - a file which *has not* been staged or committed; or
* ignored - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

dependency caches, such as the contents of /node\_modules or /packages

compiled code, such as .o, .pyc, and .class files

build output directories, such as /bin, /out, or /target

files generated at runtime, such as .log, .lock, or .tmp

hidden system files, such as .DS\_Store or Thumbs.db

personal IDE config files, such as .idea/workspace.xml

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. .gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

you can use # to include comments in your .gitignore file:

# ignore all logs

\*.log

You can use \ to escape .gitignore pattern characters if you have files or directories containing them:

# ignore the file literally named foo[01].txt

foo\[01\].txt