1. **What is the difference between Git and SVN?**

Ans:

|  |  |
| --- | --- |
| GitHub | SVN |
| GitHub is a distributed version control platform. | SVN is a centralized version control platform. |
| It uses multiple repositories for accessing and maintenance of code. | SVN does not have any centralized repository for code maintenance. |
| It is available offline means you can continue working even if the connection is lost. | In SVN, the connection has to be there for code commitment. |
| Faster commit in GitHub because you work on local repositories. | Slower than GitHub as you commit the code to the central repository. |
| In GitHub, even if a single point failure occurs, we can still commit the changes as the developer is using it in the local repository until it gets fixed. | In SVN, if node failure happens then, some code breaks the build flow, and the developers can’t commit the work. |
| In this, you have a single repository called the branch where you keep the original/modified code. | In SVN, there is an additional repository called trunk, along with a branch where the final developed code is stored. |
| The content in GitHub is stored as metadata. | SVN stores files of content. |
| The cloning feature is available on GitHub. | The cloning feature is not available in SVN. |
| Branching and merging support is available in GitHub. | Merging support is not available in SVN. |
| GitHub requires you to check out the repository as a unit. | SVN allows you to check out branches and their sub-trees. |
| Storing large binary files can slow down the performance in GitHub. | In SVN, the latest changes are only checked out, so it takes less time than GitHub. |

**2. Name a few Git commands and explain their usage?**

Ans

**git config**

**Usage:**git config -global user.name “[name]”

**Usage:**git config -global user.email “[email address]”

This command sets the author name and email address respectively to be used with your commits.

**git init**

**Usage: git init [repository name]**

This command is used to start a new repository.

**git clone**

**Usage: git clone [url]**

This command is used to obtain a repository from an existing URL.

**git add**

**Usage:**git add [file]

This command adds one or more files to the staging area.





**git commit**

**Usage: git commit -m “[ Type in the commit message]”**

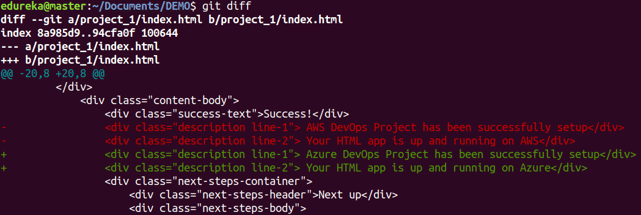
This command records or snapshots the file permanently in the version history.



**git diff**

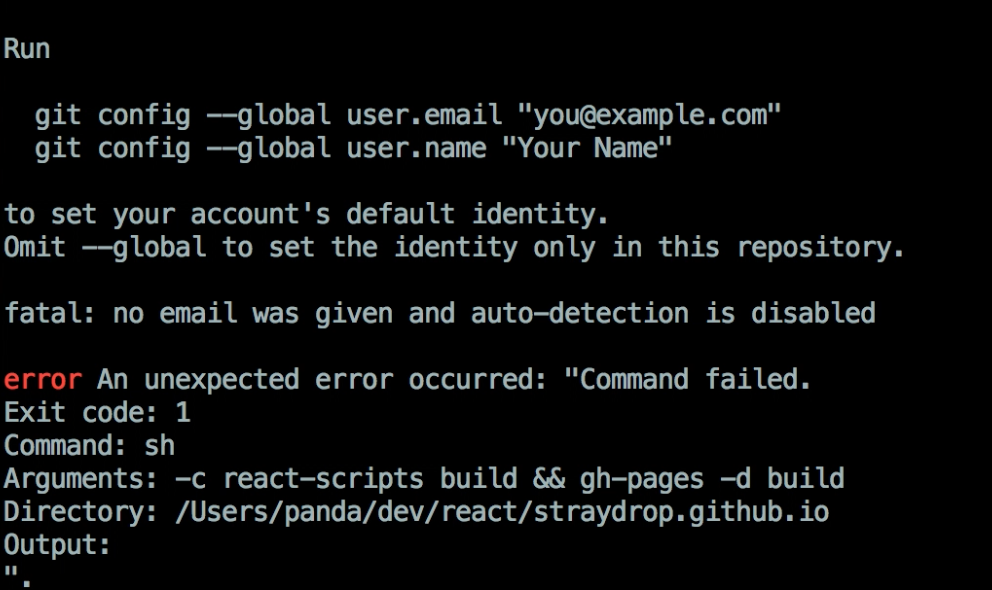
**Usage: git diff**

This command shows the file differences which are not yet staged.



1. **What is the function of ‘git config’?**

Ans: **git config**  
**Usage:**git config -global user.name “[name]”  
**Usage:**git config -global user.email “[email address]”

This command sets the author name and email address respectively to be used with your commits\

1. **Explain the different points when a merge can enter a conflicted stage. What is the difference between fork, branch, and clone?**

Ans: Types of Git Merge Conflicts

There are two points when a merge can enter a conflicted state:

**1. Starting the Merge Process**

If there are changes in the working directory’s stage area for the current project, merging won’t start.

In this case, conflicts happen due to pending changes that need to be stabilized using different Git commands.

**2. During the Merge Process**

The failure during the merge process indicates that there is a conflict between the local branch and the branch being merged.

In this case, Git resolves as much as possible, but there are things that have to be resolved manually in the conflicted files.

We will now go over resolving merge conflicts in Git.

**A clone** is simply a copy of a repository. On the surface, its result is equivalent to svn checkout, where you download source code from some other repository. The difference between centralized VCS like Subversion and DVCSs like Git is that in Git, when you clone, you are actually copying the entire source repository, including all the history and branches. You now have a new repository on your machine and any commits you make go into that repository. Nobody will see any changes until you push those commits to another repository (or the original one) or until someone pulls commits from your repository, if it is publicly accessible.

**A branch** is something that is within a repository. Conceptually, it represents a thread of development. You usually have a master branch, but you may also have a branch where you are working on some feature xyz, and another one to fix bug abc. When you have checked out a branch, any commits you make will stay on that branch and not be shared with other branches until you merge them with or rebase them onto the branch in question. Of course, Git seems a little weird when it comes to branches until you look at the underlying model of how branches are implemented. Rather than explain it myself (I've already said too much, methinks), I'll link to the "computer science" explanation of how Git models branches and commits, taken from the Git website:

**A fork** isn't a Git concept really, it's more a political/social idea. That is, if some people aren't happy with the way a project is going, they can take the source code and work on it themselves separate from the original developers. That would be considered a fork. Git makes forking easy because everyone already has their own "master" copy of the source code, so it's as simple as cutting ties with the original project developers and doesn't require exporting history from a shared repository like you might have to do with SVN.

1. **What is the difference between rebasing and merge in Git?**

When you **rebase** your branch onto their branch, you tell Git to make it look as though you checked out their branch cleanly, then did all your work starting from there. That makes a clean, conceptually simple package of changes that someone can review. You can repeat this process again when there are new changes on their branch, and you will always end up with a clean set of changes "on the tip" of their branch.

When you **merge** their branch into your branch, you tie the two branch histories together at this point. If you do this again later with more changes, you begin to create an interleaved thread of histories: some of their changes, some of my changes, some of their changes. Some people find this messy or undesirable.