KIP Intern Git Assignment
Group 12
Topic:- **Add**

Git Add

The `git add` command adds new or changed files in your working directory to the Git staging area. `git add` is an important command - without it, no `git commit` would ever do anything. Sometimes, `git add` can have a reputation for being an unnecessary step in development. But in reality, `git add` is an important and powerful tool. `git add` allows you to shape history without changing how you work.

When do we use `git add`?

As we're working, we change and save a file or multiple files. Then, before we commit, we must **`git add`.** This step allows us to choose what we are going to commit. Commits should be logical, atomic units of change - but not everyone works that way. Maybe we are making changes to files that *aren't* logical or atomic units of change. **`git add`** allows us to systematically shape our commits and our history anyway.

What Does Git Add Do?

`git add [filename]` selects that file, and moves it to the staging area, marking it for inclusion in the next commit. We can select all files, a directory, specific files, or even specific parts of a file for staging and commit.

This means if we `git add` a deleted file the *deletion* is staged for commit. The language of "add" when we're actually "deleting" can be confusing. If we think or use **`git stage`** in place of **`git add`**, the reality of what is happening may be more clear.

`git add` **and** **`git commit`** go together hand in hand. They don't work when they aren't used together. And, they both work best when used thinking of their joint functionality.

Git Add Commands:-

-A

--all

--no-ignore-removal

Update the index not only where the working tree has a file matching <pathspec> but also where the index already has an entry. This adds, modifies, and removes index entries to match the working tree.

If no <pathspec> is given when -A option is used, all files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

--no-all

--ignore-removal

Update the index by adding new files that are unknown to the index and files modified in the working tree, but ignore files that have been removed from the working tree. This option is a no-op when no <pathspec> is used.

This option is primarily to help users who are used to older versions of Git, whose "git add <pathspec>…" was a synonym for "git add --no-all <pathspec>…", i.e. ignored removed files.

-N

--intent-to-add

Record only the fact that the path will be added later. An entry for the path is placed in the index with no content. This is useful for, among other things, showing

the unstaged content of such files with `git diff` and committing them with `git commit -a`.

--refresh

Don't add the file(s), but only refresh their stat() information in the index.

Interactive Mode

When the command enters the interactive mode, it shows the output of the status subcommand, and then goes into its interactive command loop.

The command loop shows the list of subcommands available, and gives a prompt "What now> ". In general, when the prompt ends with a single >, you can pick only one of the choices given and type return, like this:

```
*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> 1
```

We also could say **s** or **sta** or **status** above as long as the choice is unique.

The main command loop has 6 subcommands (plus help and quit).

status

This shows the change between HEAD and index (i.e. what will be committed if you say git commit), and between index and working tree files (i.e. what you could stage further before git commit using git add) for each path. A sample output looks like this:

```
           staged     unstaged path
1:       binary      nothing foo.png
2:     +403/-35        +1/-1 git-add--interactive.perl
```

To remove a selection, prefix the input with - like this:

```
Update>> -2
```

After making the selection, answer with an empty line to stage the contents of working tree files for selected paths in the index.

revert

This has a very similar UI to **update**, and the staged information for selected paths is reverted to that of the HEAD version. Reverting new paths makes them untracked.

add untracked

This has a very similar UI to **update** and **revert** and lets you add untracked paths to the index.

patch

This lets you choose one path out of a **status** like selection. After choosing the path, it presents the diff between the index and the working tree file and asks you if you want to stage the change of each hunk. You can select one of the following options and type return:

```
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

After deciding the fate for all hunks, if there is any hunk that was chosen, the index is updated with the selected hunks.

We can omit having to type return here, by setting the configuration variable **interactive.singlekey** to **true.**

diff
This lets you review what will be committed (i.e. between HEAD and index).

Editing Patch

Invoking **git add -e** or selecting **e** from the interactive hunk selector will open a patch in your editor; after the editor exits, the result is applied to the index. You are free to make arbitrary changes to the patch, but note that some changes may have confusing results, or even result in a patch that cannot be applied. If you want to abort the operation entirely (i.e., stage nothing new in the index), simply delete

all lines of the patch. The list below describes some common things you may see in a patch, and which editing operations make sense on them.

modified content
Modified content is represented by "-" lines (removing the old content) followed by "+" lines (adding the replacement content). You can prevent staging the modification by converting "-" lines to " ", and removing "+" lines. Beware that modifying only half of the pair is likely to introduce confusing changes to the index.

There are also several operations that should be avoided entirely, as they will make the patch impossible to apply:

- adding context (" ") or removal ("-") lines
- deleting context or removal lines
- modifying the contents of context or removal lines

Examples of `git add`

`git add` usually fits into the workflow in the following steps:

1. Create a branch: **`git branch update-readme`**
2. Checkout to that branch: **`git checkout update-readme`**
3. Change a file or files
4. Save the file or files
5. Add the files or segments of code that should be included in the next commit: **`git add README.md`**
6. Commit the changes: **`git commit -m "update the README to include links to contributing guide"`**

7. Push the changes to the remote branch: **`git push -u origin update-readme`**

But, **`git add`** could also be used like:

1. Create a branch: **`git branch update-readme`**
2. Checkout to that branch: **`git checkout update-readme`**
3. Change a file or files
4. Save the file or files
5. Add only one file, or one part of the changed file: **`git add README.md`**
6. Commit the first set of changes: **`git commit -m "update the README to include links to contributing guide"`**
7. Add another file, or another part of the changed file: **`git add CONTRIBUTING.md`**
8. Commit the second set of changes: **`git commit -m "create the contributing guide"`**
9. (Repeat as necessary)
10. Push the changes to the remote branch: **`git push -u origin update-readme`**

Related Terms

- **`git status`:** Always a good idea, this command shows you what branch you're on, what files are in the working or staging directory, and any other important information.
- **`git checkout [branch-name]`:** Switches to the specified branch and updates the working directory.
- **`git commit -m "descriptive message"`:** Records file snapshots permanently in the version history.
- **`git push`:** Uploads all local branch commits to the remote.

Documented by:-

Utkarsh Ambastha

Emp ID: 1713