Git –version

Git log --oneline *this reads the logs in short form*

Git log --oneline –-reverse  *this arranges the log in timely order.*

git status -s *how to get short status*

git diff –-staged *for comparing file to files*

Git config -–global user.name “ibidun oluwatosin”

git config --global user.email [ibiduntosin@yahoo.com](mailto:ibiduntosin@yahoo.com)

git restore –-staged (this restore to previous version of a staged file)

git clean (it totally remove specified file)

git config --global -e *opening editor*

git log  *this used to check logs*

git config -help  *this gives command help*

using touch; touch newFile.txt *only creates*.

using echo; echo > newFile.txt *only creates.*

using cat; cat > newFile.txt *creates and can start appending to file.*

using vim; vim newFile.txt *creates and can start editing the file*.

git add newFile.txt

git commit -m "added new file"

echo "Put any content here" >> Newfile.txt *adds content into the file created*

*Save with* escape, **:wq ENTER.**

*To stage all files use* **'git add .** **'**

git config -h *gives help summary.*

Blobs = files

Trees = directories

**or create a new repository on the command line**

echo "# ubuntu" >> README.md

git init

git add README.md

Git status *shows staged file to be pushed or pull*

git commit -m "first commit"

git branch -M main

git remote add origin git@github.com:TeeConcept/ubuntu.git

git push -u origin main

**…or push an existing repository from the command line**

git remote add origin git@github.com:TeeConcept/ubuntu.git

git branch -M main

git push -u origin main

Git branch <new branch name> to add new branch.

Git checkout <branch name> to switch branches

git clone <https://github.com/josephifekwe97/Prestmit_ui_test.git> *forking public repo.*

**Gitflow workflow**

Gitflow is a legacy Git workflow that was originally a disruptive and novel strategy for managing Git branches. Gitflow has fallen in popularity in favor of [trunk-based workflows](https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development), which are now considered best practices for modern continuous software development and [DevOps](https://www.atlassian.com/devops/what-is-devops) practices. Gitflow also can be challenging to use with [CI/CD](https://www.atlassian.com/continuous-delivery). This post details Gitflow for historical purposes.

**What is Gitflow?**

Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. It was first published and made popular by [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). Compared to trunk-based development, Gitflow has numerous, longer-lived branches and larger commits. Under this model, developers create a feature branch and delay merging it to the main trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge and have a higher risk of deviating from the trunk branch. They can also introduce conflicting updates.

Gitflow can be used for projects that have a scheduled release cycle and for the [DevOps best practice](https://www.atlassian.com/devops/what-is-devops/devops-best-practices) of [continuous delivery](https://www.atlassian.com/continuous-delivery). This workflow doesn’t add any new concepts or commands beyond what’s required for the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow). Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

[Read tutorial](https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud)

**How it works**

**Develop and main branches**

Instead of a single main branch, this workflow uses two branches to record the history of the project. The main branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the main branch with a version number.

The first step is to complement the default main with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

git branch develop

git push -u origin develop

This branch will contain the complete history of the project, whereas main will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop.

When using the git-flow extension library, executing git flow init on an existing repo will create the develop branch:

$ git flow init

Initialized empty Git repository in ~/project/.git/

No branches exist yet. Base branches must be created now.

Branch name for production releases: [main]

Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?

Feature branches? [feature/]

Release branches? [release/]

Hotfix branches? [hotfix/]

Support branches? [support/]

Version tag prefix? []

$ git branch

\* develop

main

**Feature branches**

**Step 1. Create the repository**

Each new feature should reside in its own branch, which can be [pushed to the central repository](https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of main, feature branches use develop as their parent branch. When a feature is complete, it gets [merged back into develop](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with main.

Note that feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow workflow doesn’t stop there.

Feature branches are generally created off to the latest develop branch.

**Creating a feature branch**

Without the git-flow extensions:

git checkout develop

git checkout -b feature\_branch

When using the git-flow extension:

git flow feature start feature\_branch

Continue your work and use Git like you normally would.

**Finishing a feature branch**

When you’re done with the development work on the feature, the next step is to merge the feature\_branch into develop.

Without the git-flow extensions:

git checkout develop

git merge feature\_branch

Using the git-flow extensions:

git flow feature finish feature\_branch

**Release branches**

Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into main and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation. Like feature branches, release branches are based on the develop branch. A new release branch can be created using the following methods.

Without the git-flow extensions:

git checkout develop

git checkout -b release/0.1.0

When using the git-flow extensions:

$ git flow release start 0.1.0

Switched to a new branch 'release/0.1.0'

Once the release is ready to ship, it will get merged it into main and develop, then the release branch will be deleted. It’s important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

Without the git-flow extensions:

git checkout main

git merge release/0.1.0

Or with the git-flow extension:

git flow release finish '0.1.0'

**Hotfix branches**

Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on main instead of develop. This is the only branch that should fork directly off of main. As soon as the fix is complete, it should be merged into both main and develop (or the current release branch), and main should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with main. A hotfix branch can be created using the following methods:

Without the git-flow extensions:

git checkout main

git checkout -b hotfix\_branch

When using the git-flow extensions:

$ git flow hotfix start hotfix\_branch

Similar to finishing a release branch, a hotfix branch gets merged into both main and develop.

git checkout main

git merge hotfix\_branch

git checkout develop

git merge hotfix\_branch

git branch -D hotfix\_branch

$ git flow hotfix finish hotfix\_branch

**Example**

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a main branch.

git checkout main

git checkout -b develop

git checkout -b feature\_branch

# work happens on feature branch

git checkout develop

git merge feature\_branch

git checkout main

git merge develop

git branch -d feature\_branch

In addition to the feature and release flow, a hotfix example is as follows:

git checkout main

git checkout -b hotfix\_branch

# work is done commits are added to the hotfix\_branch

git checkout develop

git merge hotfix\_branch

git checkout main

git merge hotfix\_branch

**Summary**

Here we discussed the Gitflow Workflow. Gitflow is one of many styles of [Git workflows](https://www.atlassian.com/git/tutorials/comparing-workflows) you and your team can utilize.

Some key takeaways to know about Gitflow are:

* The workflow is great for a release-based software workflow.
* Gitflow offers a dedicated channel for hotfixes to production.

The overall flow of Gitflow is:

1. A develop branch is created from main

2. A release branch is created from develop

3. Feature branches are created from develop

4. When a feature is complete it is merged into the develop branch

5. When the release branch is done it is merged into develop and main

6. If an issue in main is detected a hotfix branch is created from main

7. Once the hotfix is complete it is merged to both develop and main

Next, learn about the [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) or visit our [workflow comparison page](https://www.atlassian.com/git/tutorials/comparing-workflows).

1. **History and Traceability**: Version control systems track changes made to the codebase over time. This historical record allows developers to understand the evolution of the project, review past changes, and identify when and why specific modifications were made.
2. **Collaboration**: VCS enables multiple developers to work on the same codebase simultaneously. It facilitates collaboration by allowing developers to work on different branches of the codebase, merge their changes together, and resolve conflicts that may arise during the integration process.
3. **Concurrency Management**: In collaborative environments, multiple developers may attempt to modify the same files concurrently. Version control systems handle concurrent access to files by providing mechanisms for merging changes made by different developers and resolving conflicts that occur when changes overlap.
4. **Backup and Recovery**: Version control systems serve as a backup mechanism for the codebase. By storing all versions of the code, including previous iterations and changes, they provide a safety net in case of accidental deletions, data corruption, or other unforeseen events.
5. **Code Reviews**: VCS facilitates code reviews by allowing developers to review changes made by their peers before integrating them into the main codebase. Code review processes help improve code quality, identify potential issues or bugs early on, and share knowledge among team members.
6. **Reproducibility**: With version control, it's possible to recreate any past version of the codebase at any point in time. This reproducibility is valuable for debugging purposes, testing, and ensuring that previous versions of the software can be reliably deployed if needed.
7. **Branching and Experimentation**: Version control systems support branching, which enables developers to create separate lines of development for new features, experiments, or bug fixes without affecting the main codebase. Branching allows for parallel development efforts and provides a mechanism for isolating changes until they are ready to be merged into the main codebase.

**DOCKER**

Commands:

build Build an image from a Dockerfile

history Show the history of an image

import Import the contents from a tarball to create a filesystem image

inspect Display detailed information on one or more images

load Load an image from a tar archive or STDIN

ls List images

prune Remove unused images

pull Download an image from a registry

push Upload an image to a registry

rm Remove one or more images

save Save one or more images to a tar archive (streamed to STDOUT by default)

tag Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE

*docker run -d container\_name* this runs container without holding on to the terminal

PORT MAPPING

docker run -p localPort:containerPort container

docker run -p 8080:8080 -p 50000:50000 jenkins/jenkins

LOGS

Container Logs Container Id/Name

DOCKER COMPOSE

version: '3.8'

services:

web:

image: nginx:latest

ports:

- "8080:80"

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

restart: always

db:

image: mysql:latest

environment:

MYSQL\_ROOT\_PASSWORD: example

MYSQL\_DATABASE: mydatabase

MYSQL\_USER: myuser

MYSQL\_PASSWORD: mypassword

volumes:

- db\_data:/var/lib/mysql

restart: always

volumes:

db\_data: