In DevOps, **Change Management** refers to controlling, tracking, and managing changes to code, infrastructure, or processes in a way that ensures stability while still enabling fast delivery. Here's an overview of how change management works in a **DevOps team**:

**🔄 What is Change Management in DevOps?**

It’s the process of managing changes across environments (development, testing, production) to minimize risk and downtime.

**👥 Who is in the Change Management Team?**

The team typically involves:

| **Role** | **Responsibility** |
| --- | --- |
| **Change Manager** | Approves and coordinates all changes; ensures risk is assessed and documented. |
| **DevOps Engineer** | Implements and tests changes in infrastructure, CI/CD pipelines, and environments. |
| **Developer** | Writes and updates code that will be part of the change. |
| **Tester/QA** | Tests the change in staging/pre-production environments. |
| **Product Owner** | Validates the need and timing of the change from a business perspective. |
| **Release Manager** | Oversees the release calendar and coordinates deployments. |

**📋 Typical DevOps Change Management Process**

1. **Request for Change (RFC)**
   * A developer or product owner submits a change request (via Jira, ServiceNow, etc.).
2. **Impact Assessment**
   * DevOps engineers evaluate risks, downtime, and dependencies.
3. **Approval**
   * Change Manager reviews and either approves or rejects the change.
4. **Implementation**
   * DevOps team deploys change using automated CI/CD tools (e.g., Jenkins, GitHub Actions).
5. **Testing & Validation**
   * QA team tests in staging or production with rollback plans ready.
6. **Monitoring**
   * Post-deployment, monitoring tools (e.g., Prometheus, ELK) track success/failure.
7. **Closure**
   * If successful, the change is marked as completed and documented.

**✅ Tools Commonly Used**

* **Change Tracking**: Jira, ServiceNow
* **CI/CD & Automation**: Jenkins, GitHub Actions, GitLab CI/CD
* **Monitoring**: Prometheus, Datadog, Grafana
* **Communication**: Slack, Microsoft Teams

Git Tagging :

**🔖 What is Tagging in Git?**

**Tagging** in Git means putting a **label** on a specific commit to mark it as important — like a version or release.

**🧒 Like a Kid’s Story:**

Imagine you're building a LEGO house.  
One day, you finish a really cool version. So you take a photo and write on it:  
**"Version 1.0 – My Best LEGO House!"**

That’s **tagging**.

**✅ In Git:**

* You run: git tag v1.0
* It creates a **named snapshot** of that moment in your code.
* Later, you (or anyone else) can go back to it by name:  
  git checkout v1.0

**📌 Why Use Tags?**

* To mark **releases** (like v1.0, v2.0)
* To help developers know which version is deployed or stable
* So you can go back to that exact code anytime

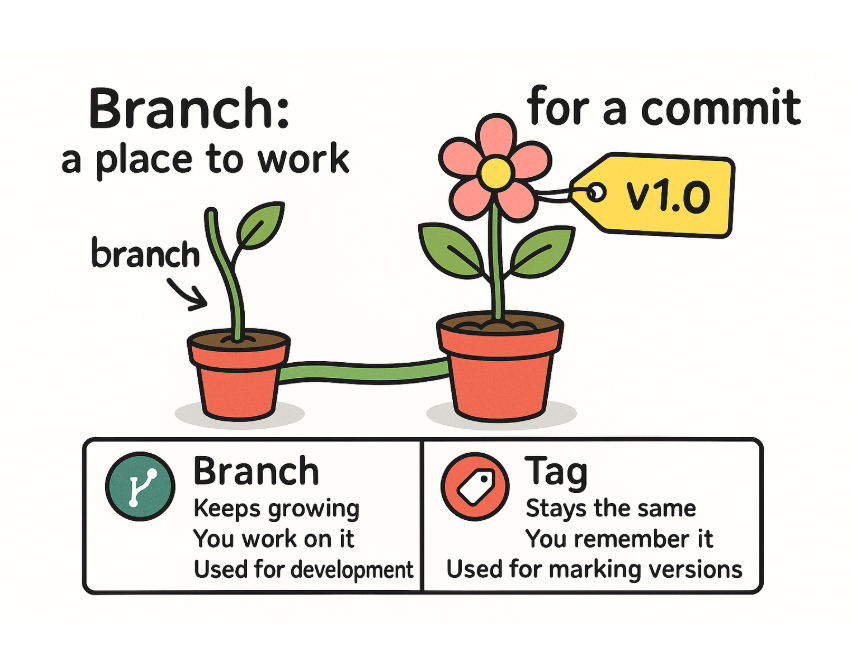
### ✅ ****Interview Answer – What is Tagging in Git?****

**"Tagging in Git is a way to mark specific points in the repository’s history as important, usually for releases. Unlike branches, tags are immutable and do not change over time. Developers commonly use tags to mark version releases, such as v1.0, v2.1, etc. There are two types of tags in Git: lightweight and annotated. Lightweight tags are just references to a commit, while annotated tags store additional metadata like the tagger’s name, email, date, and message.**

**Tags help in identifying stable code snapshots and are very useful for version control, deployment, and release management. For example, when we want to deploy a particular version to production, we can use a tag to ensure we’re deploying the exact intended code.**

* **We can create N no of branches while developing a Project**

**Difference between Branch and Tag  
**

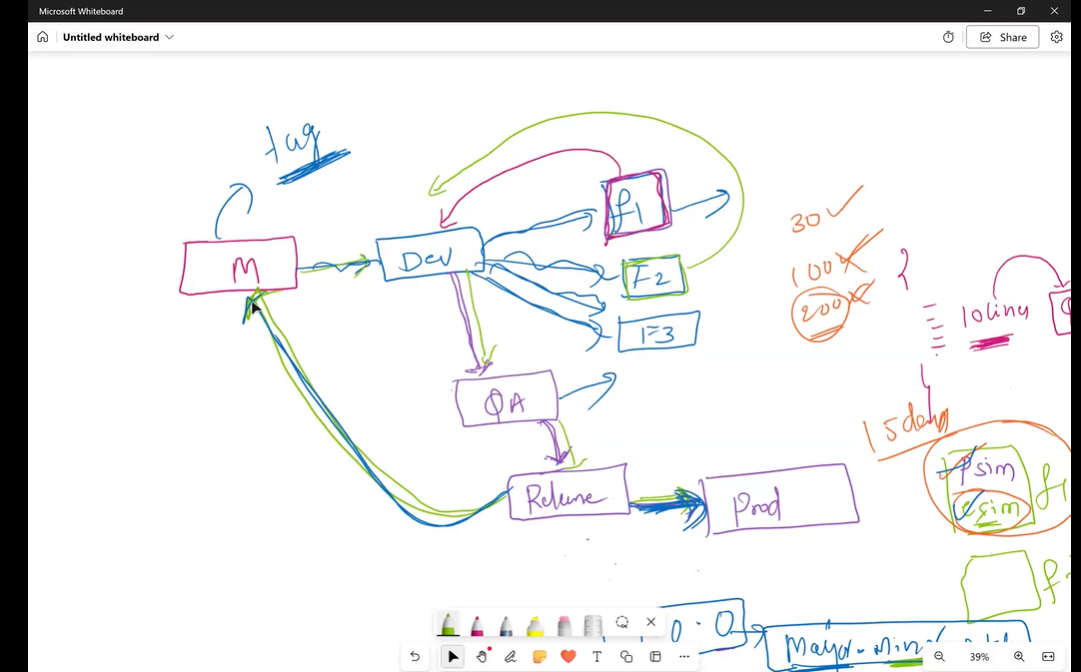
****

**To delete an un commited branch we have to use the command  
git branch –D <BN>**

**For deleting a committed branch we have to use git branch –d <bn>**

**\*\* If we checkout to other branch without comitting the updated file the other branch file code will be automatically updated and auto merge without executing merge command .**

**Example :**

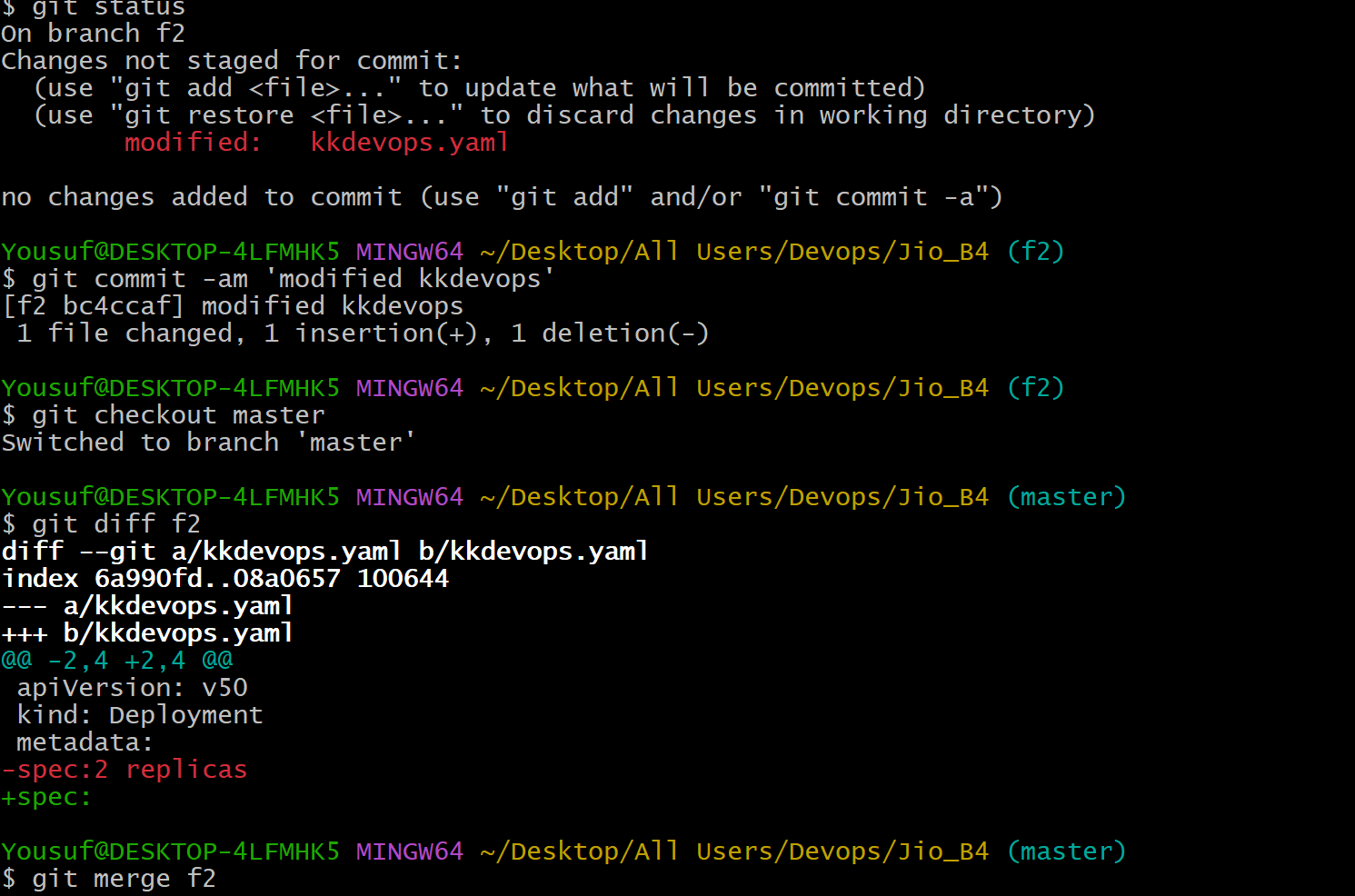
****

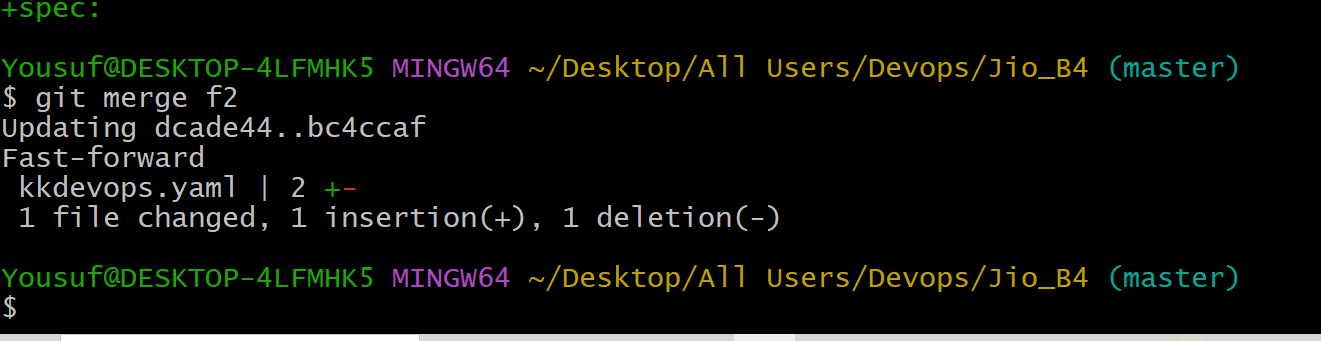
**In the below F2 branch we have updated kkdevops.yaml and then commited the updated file and then merge in to master branch.**

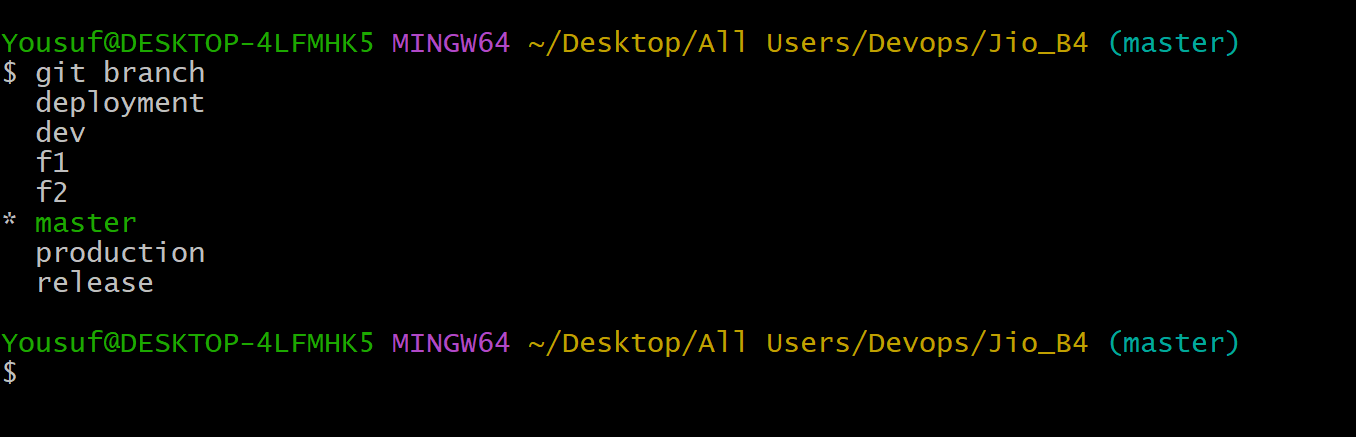
**git checkout master**

**git diff f2**

**git merge f2**

****

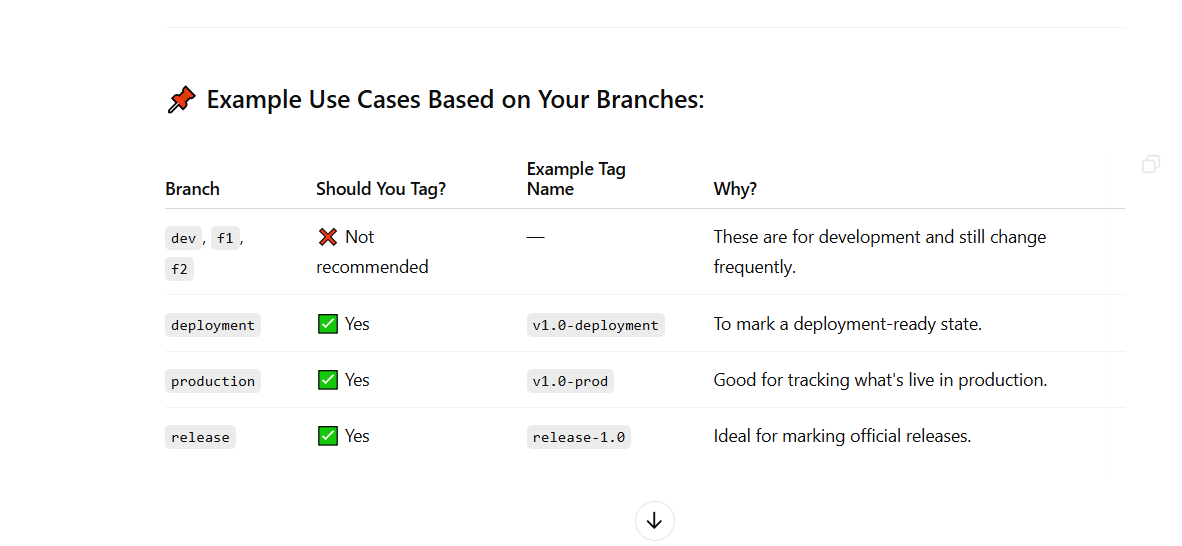
****

****

When to Create a Tag:

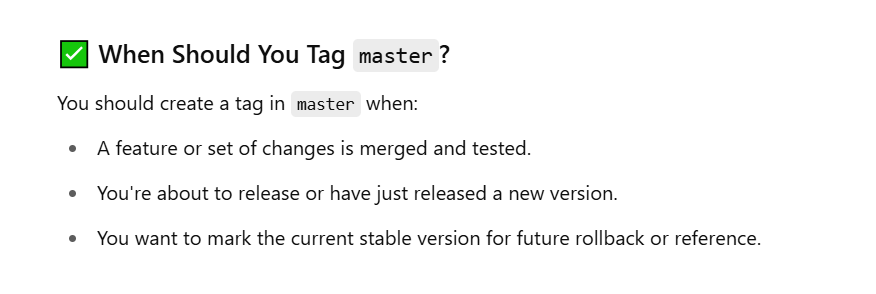
You create a tag when:

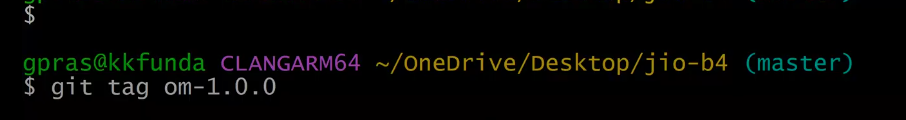
* A branch has reached a stable point (e.g., release-ready or deployed version).
* You want to mark a specific commit for future reference (like v1.0, production-release, etc.).



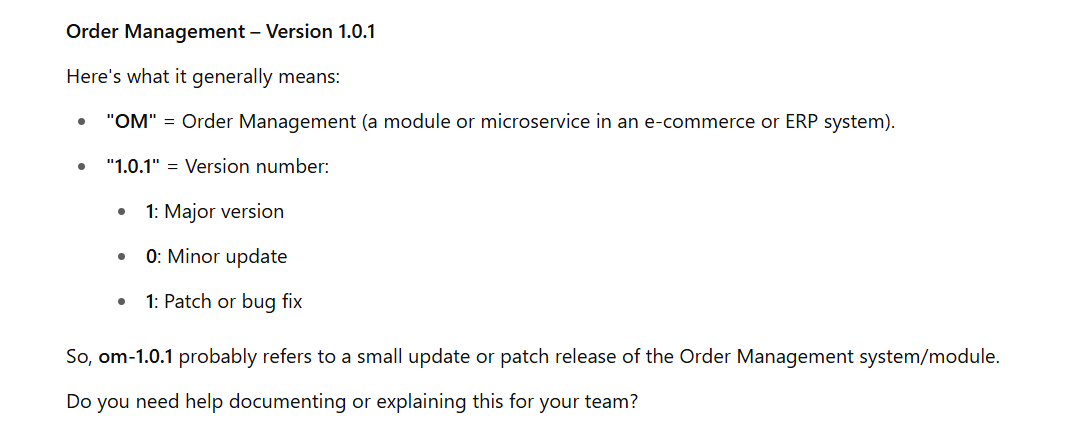
Let’s create the tag in master branch

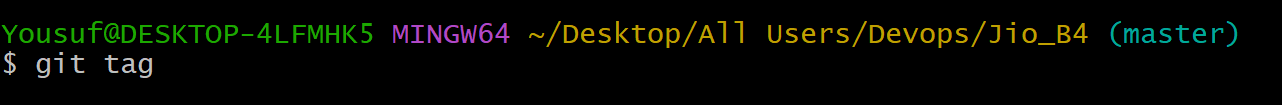
Creating a tag on the **master branch** is a very common and good practice — especially when you're marking a stable point like a release, deployment, or milestone.

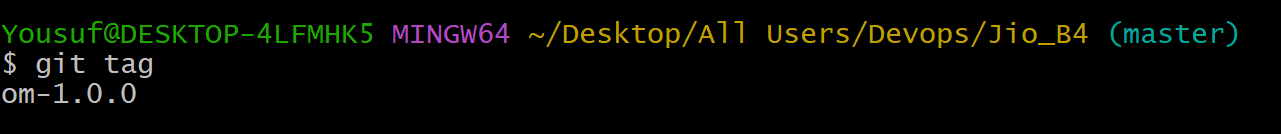


Command to create a tag  
 git tag om-1.0.0 

Om means (order management)

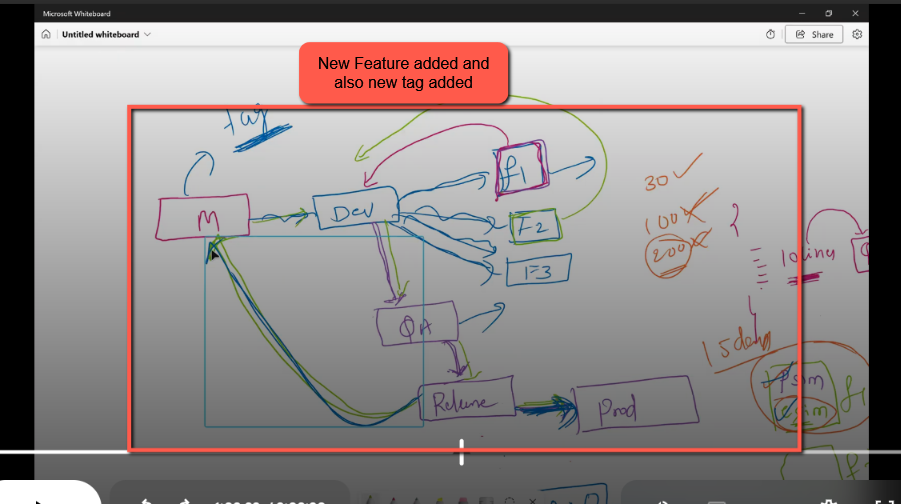


To check the list of tags execute the command git tag



Let’s Assume that a new feature developed or added in all the branches Master-dev—Qa-release-prod

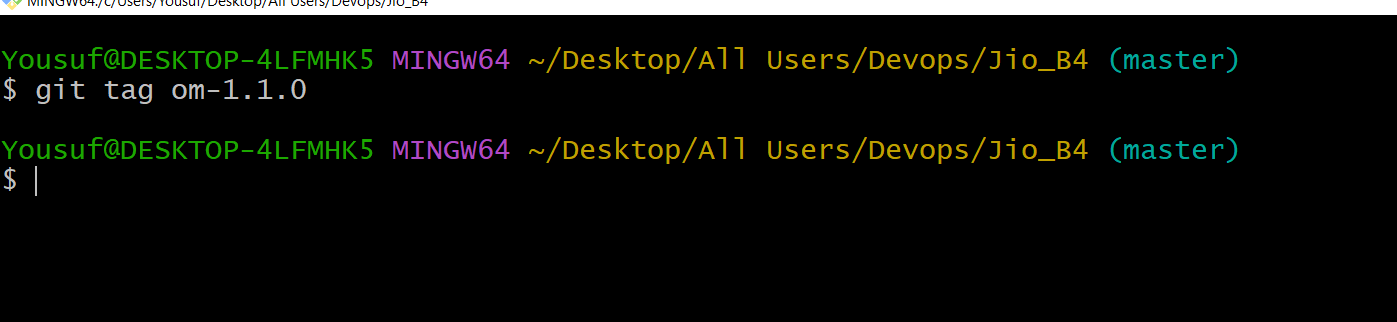
**And create a new tag in master for the new feature added in the all the branches**

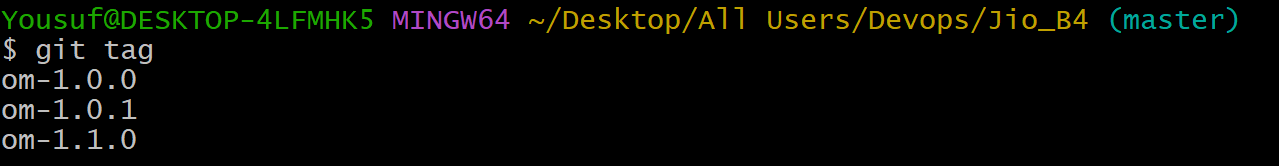


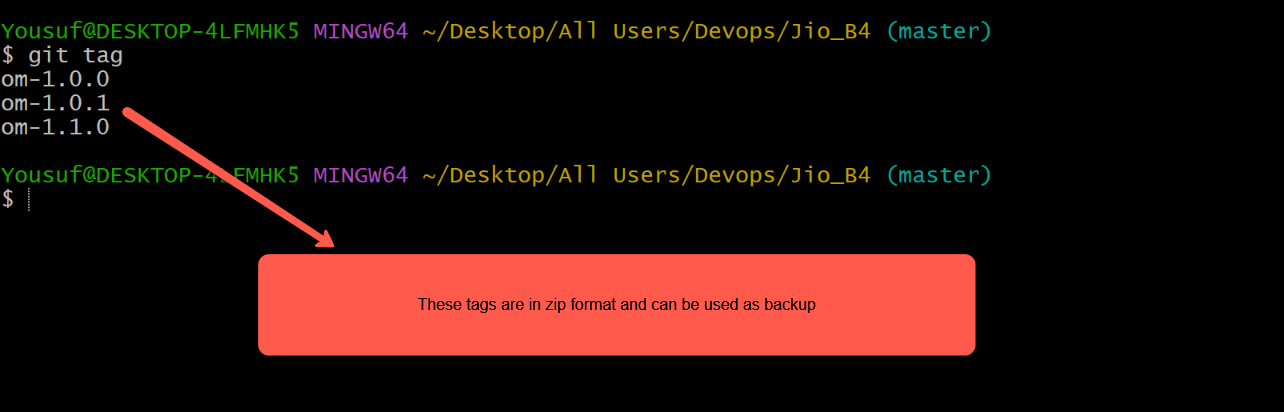
**Another New Tag “git tag om-1.0.1”**



Create one more tag with one new minor release  
git tag om-1.1.0



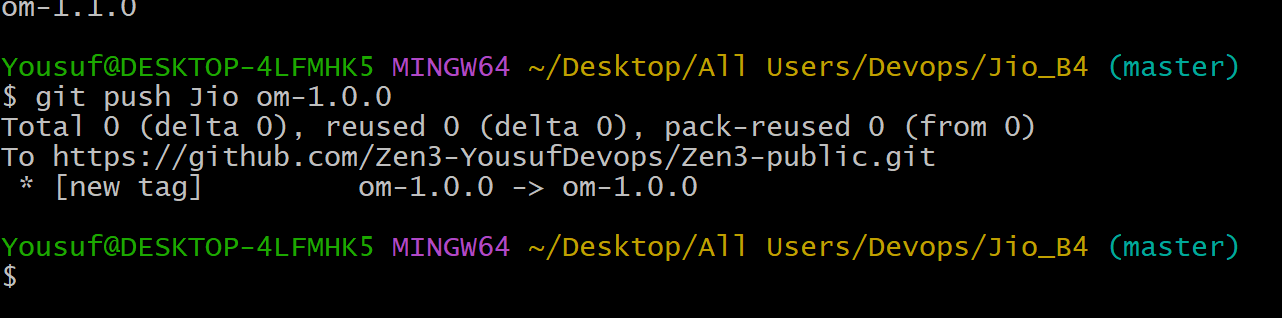
Total Three tags has been created 



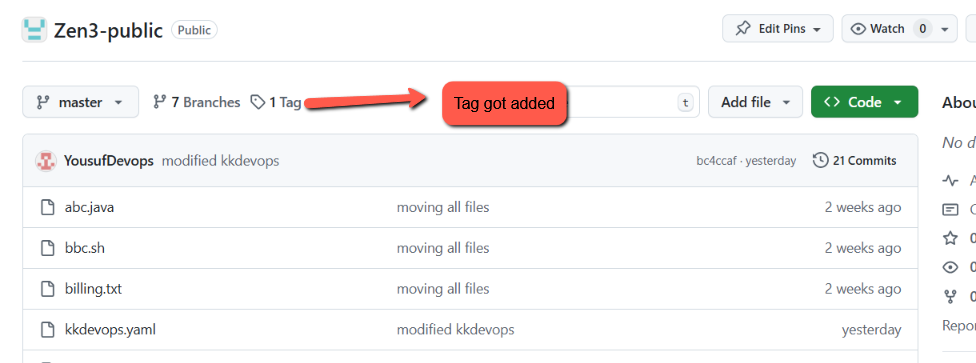
Let’s push these Tags in to Github Remote Repositoryy

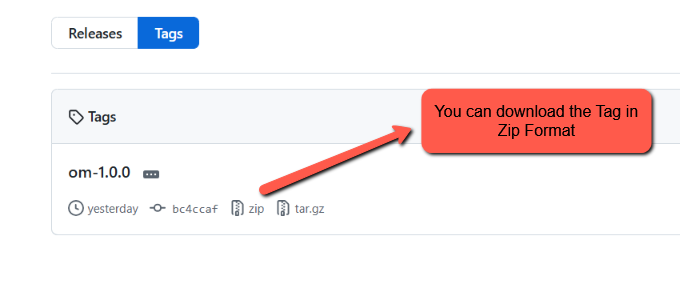
Let’s try to push the first tag in Git Repository  
  


Press Enter



Successfully Tag is moved to Github Remote Repository .So go and check the tag in Github Repository

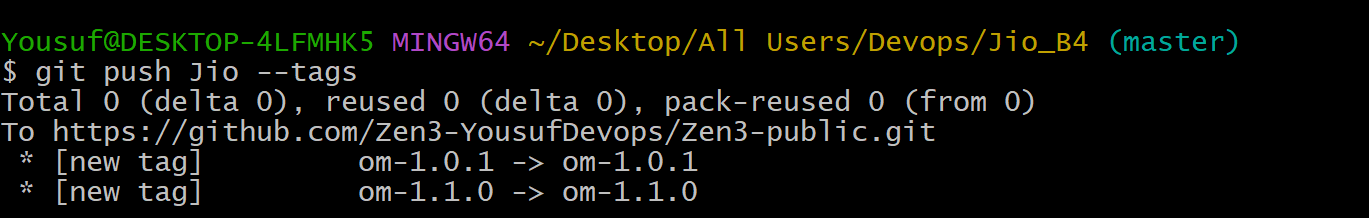
Hurray!! Tag is added in GitHub Repo  


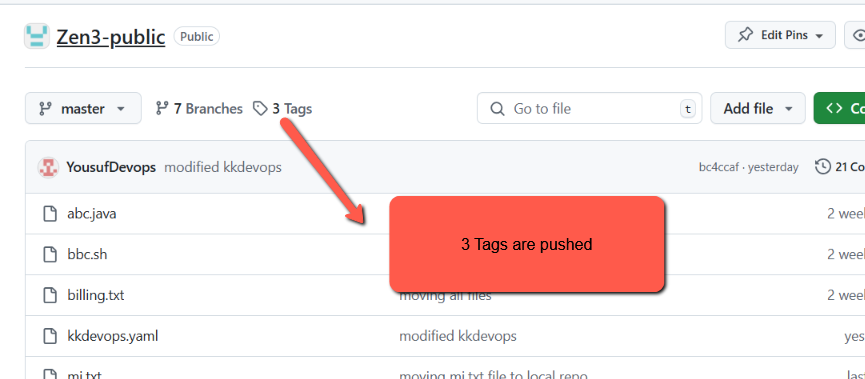


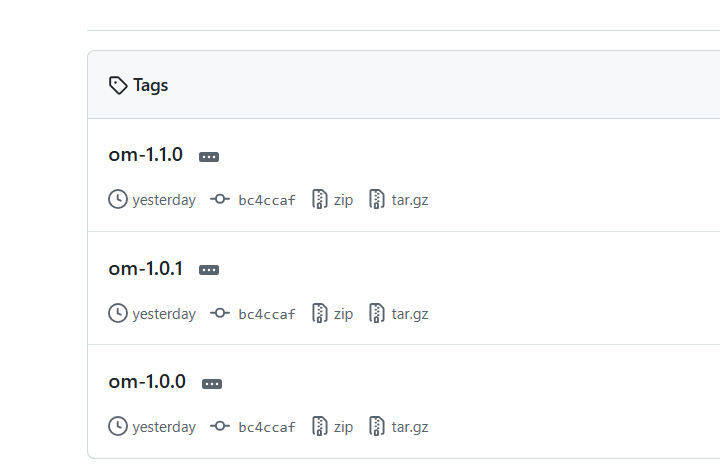
These Tag is useful for Backup of the Files

To push all the Tags to Github Repo execute the below command





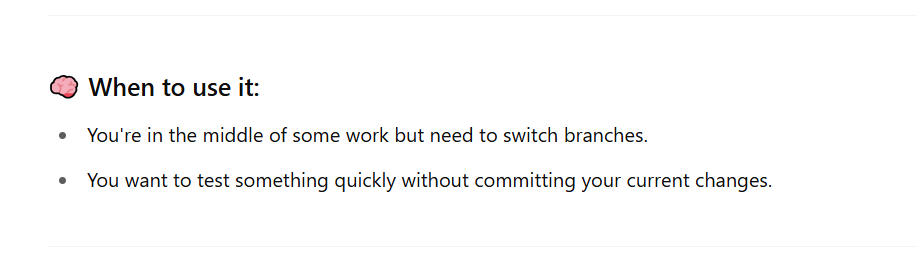
Succesfully pushed all 3 tags in GitHub Repo

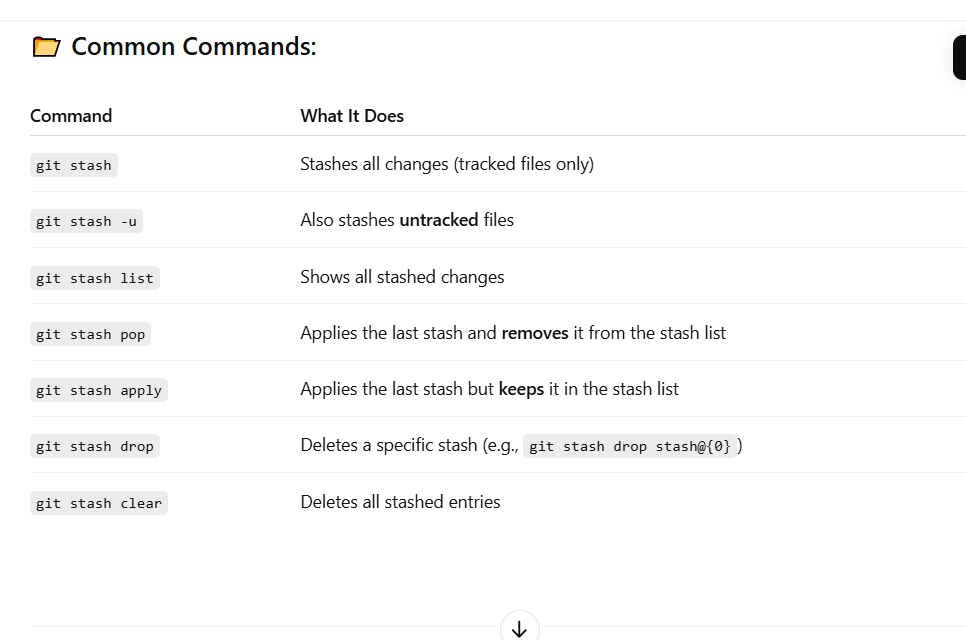


We can create tags in via Github UI too.

**Git Stash Command** : The git stash command is used to **temporarily save your uncommitted changes** (both staged and unstaged) without committing them, so you can work on something else and come back to those changes later.

git stash command saves your changes and reverts your working directory to match the last commit.





# 🧠 Git Stash Cheat Sheet

## Why Use git stash?

Git stash is useful when you're in the middle of work but need to switch branches, test something, or resolve an urgent issue—without committing incomplete changes.

## ✅ When to Use Stash

- You have local changes and need to switch branches  
- You’re not ready to commit but don’t want to lose work  
- You need to pull remote changes and avoid conflicts  
- You’re fixing an urgent bug on another branch

## ❌ When You Don’t Need Stash

- Your file doesn’t conflict with other branches  
- You’re not switching branches or pulling  
- You’re ready to commit your changes

## 🔥 Example Case: Switching with Conflicts

You're on 'dev' branch editing main.txt and want to switch to 'master', but main.txt doesn't exist in master. Git will stop you with an error:

You're on dev branch editing main.txt, and main.txt exists in master too — but with different content.  
Git will stop you with an error because switching would **overwrite your uncommitted changes** in main.txt.

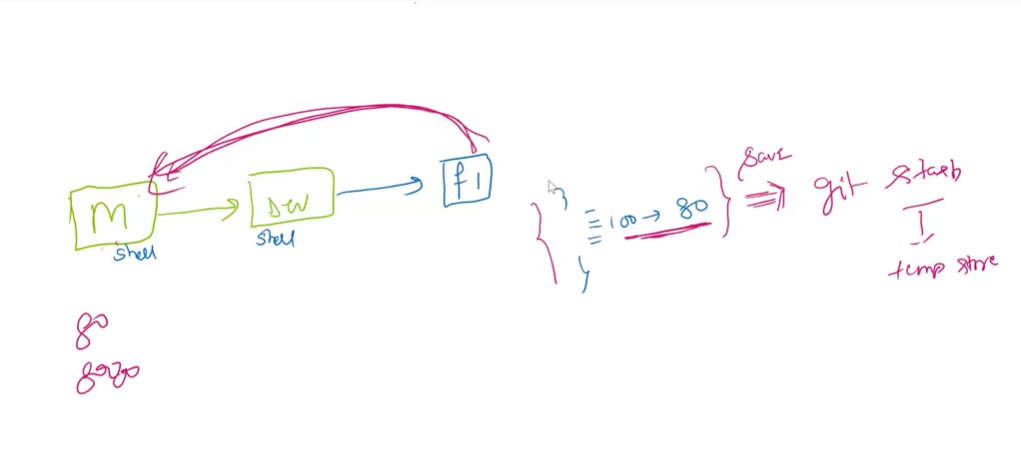
error: Your local changes to the following files would be overwritten by checkout:  
 main.txt  
Please commit your changes or stash them before you switch branches.

## 💻 Useful Commands

Temporarily save changes:  
 git stash  
  
Apply the latest stash:  
 git stash pop  
  
List stashes:  
 git stash list  
  
Apply specific stash:  
 git stash apply stash@{1}

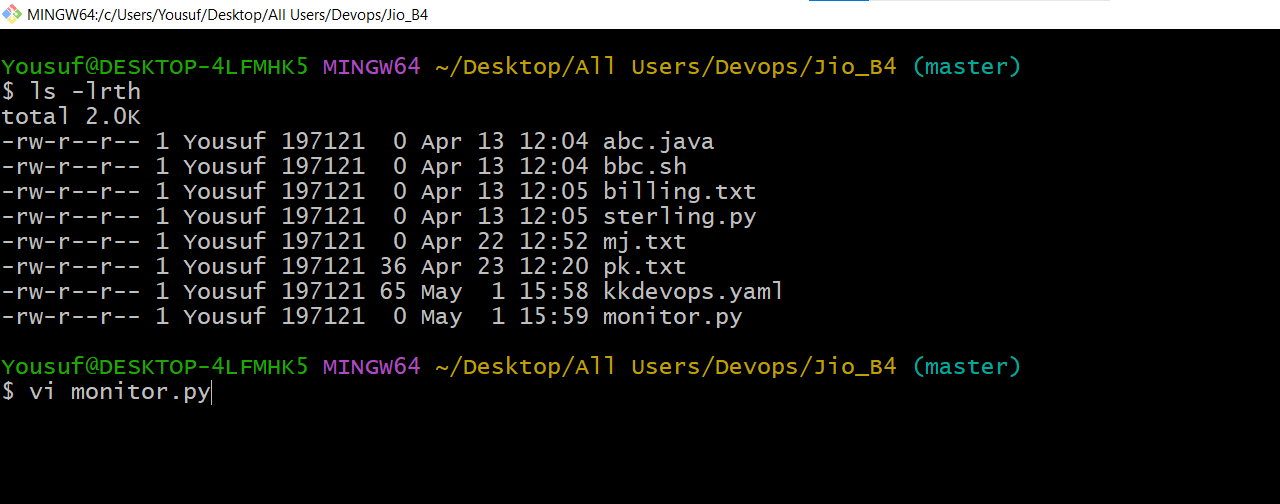
First save the code whenever we trying to switching to another branch and use the command **git stash** after executing this command then only you can switch from one branch to another branch .

After completing the the other task you can switch to the left over code branch which we we left in the mid .

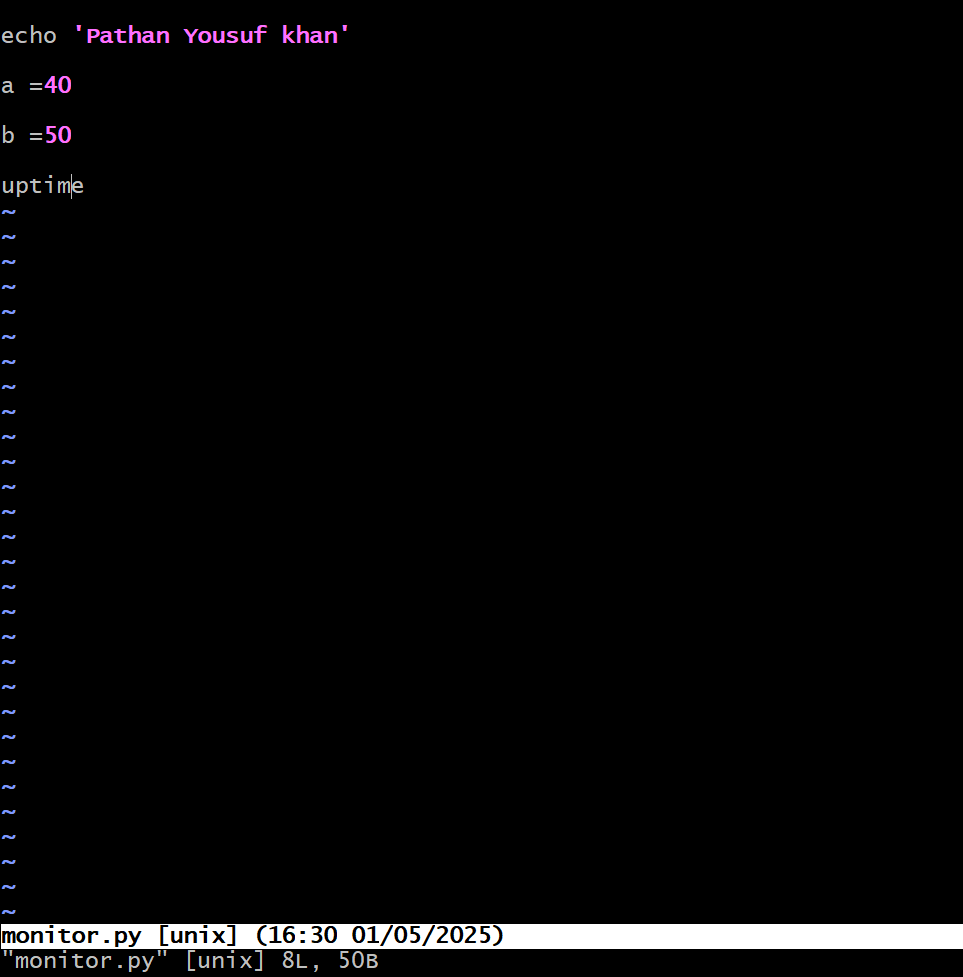


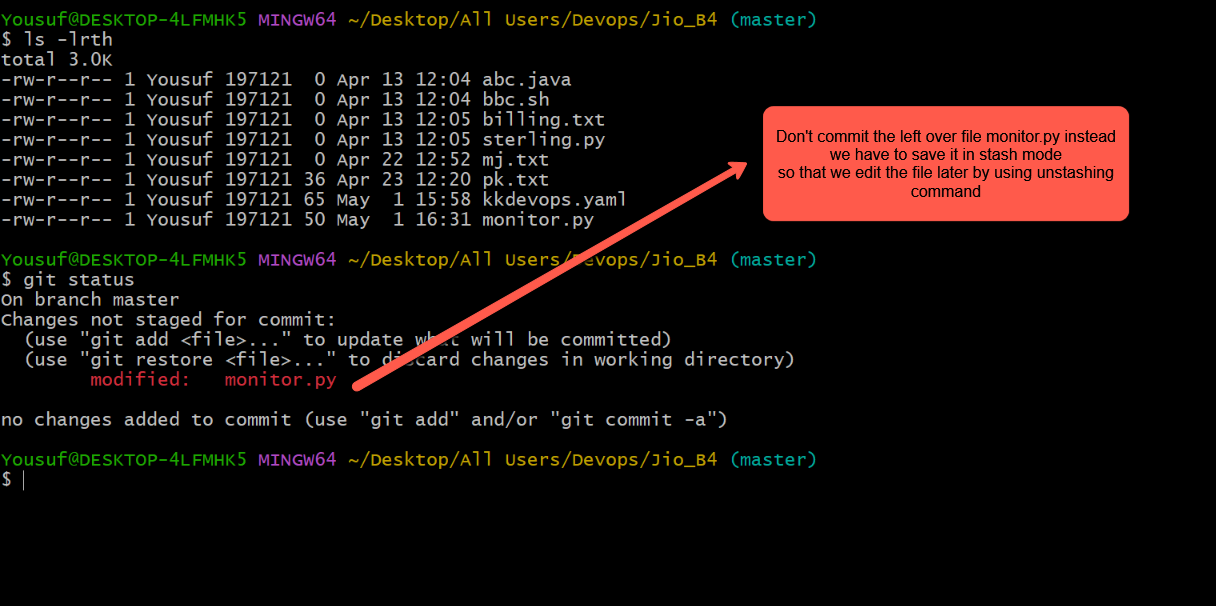
Let’s see how to use the **git stash** command through the below example

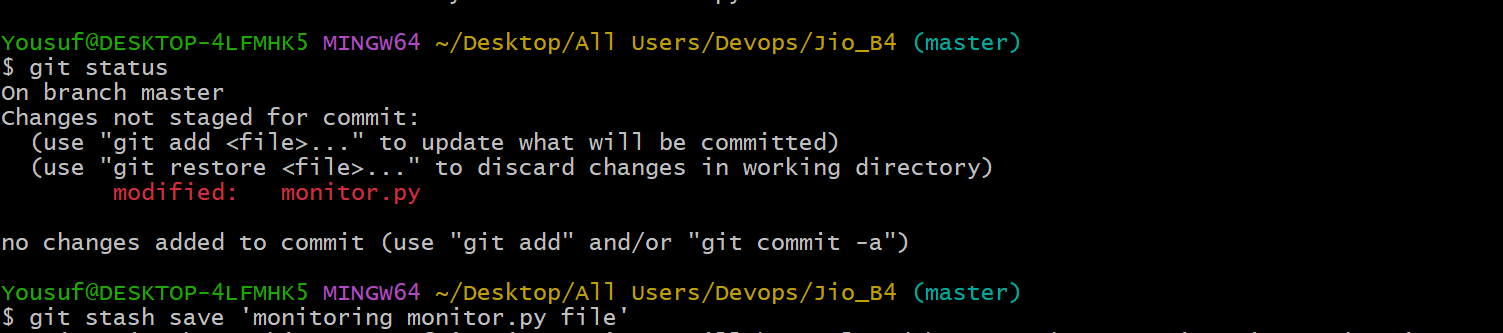
Let’s create a Monitor.py file in master branch and add some data in to it while working on it we get a call from manager to switch to another file in master branch and ask us to make changes so before moving to other file we have to stash the current file (ie) monitor.py file

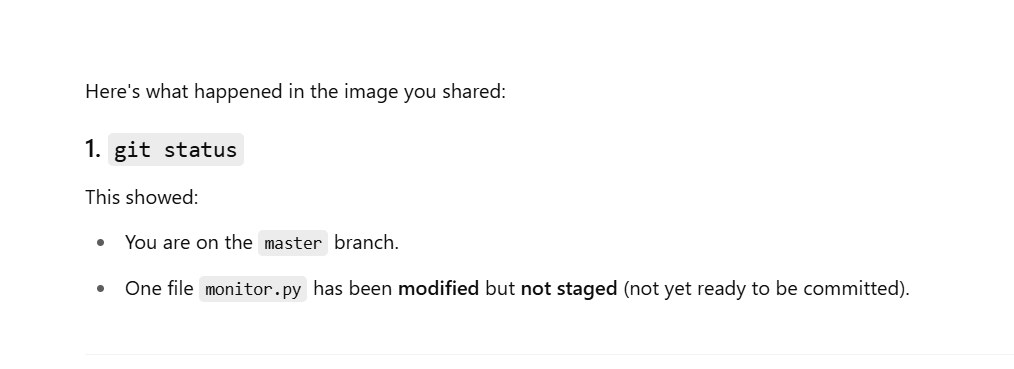


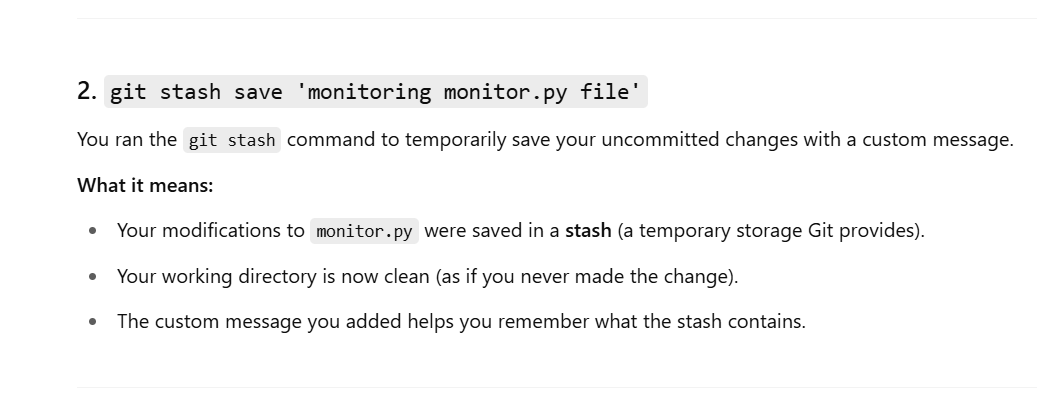
Added some data

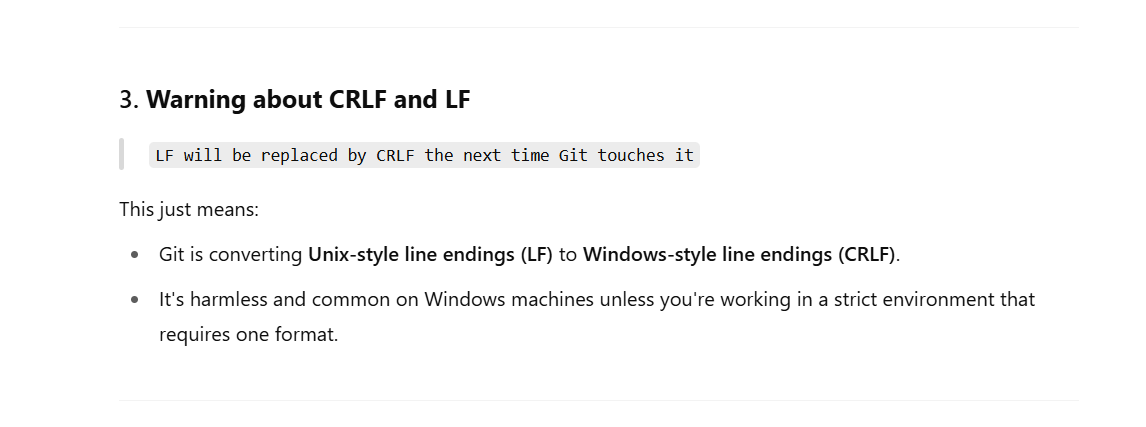








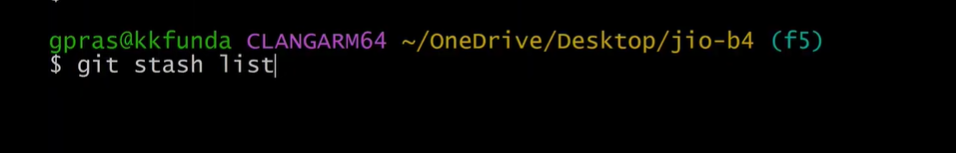


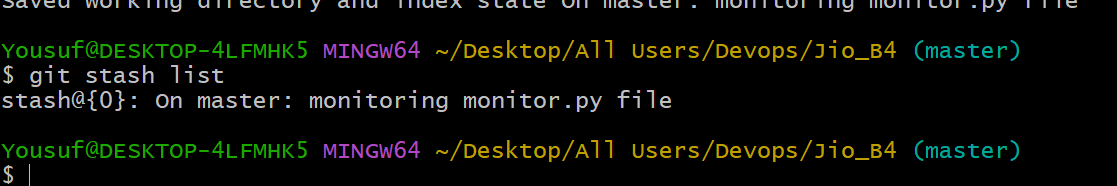


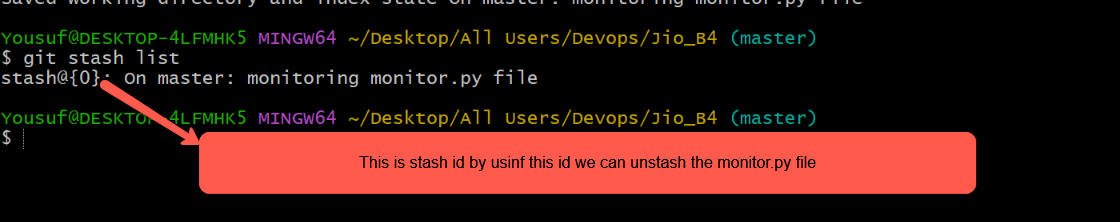
You can bring it back by using the command “git stash apply stash id”

You can find the stash id by using the command git stash list

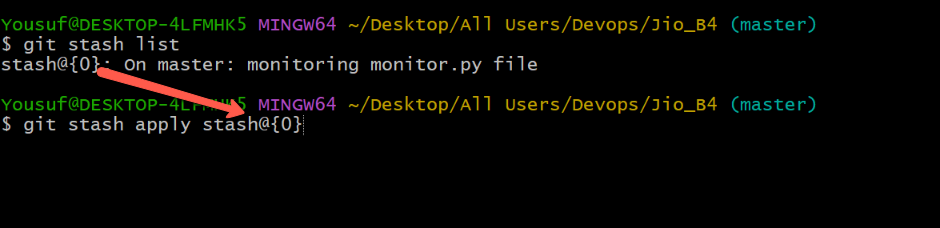
To check the stashed files we have to execute the command ‘git stash list’

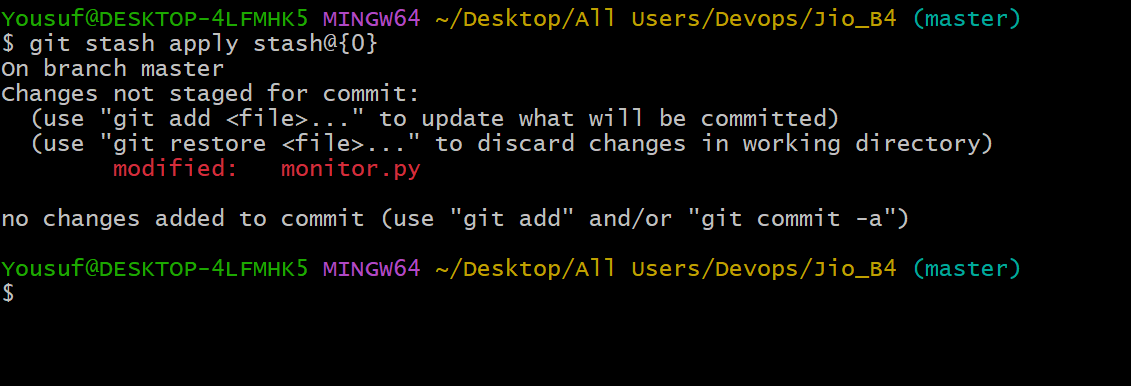






Let’s unstash the stashed file using the following command

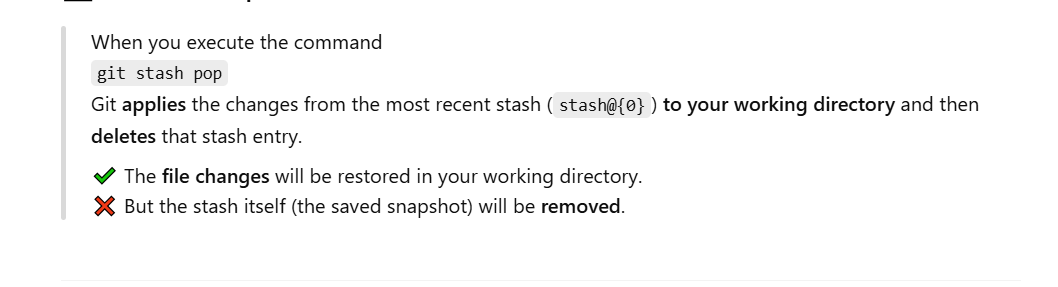
Git stash apply stash ID  


After executing the above command we can see the modified monitor.py file in the below image  


Now you can open the file monitor.py and continue editing the file

git stash pop stash@{0} --🡪 (Recent stash file )

This stash pop command



To **delete all stash entries** in Git, use the following command  
git stash clear

To delete a specific stash we have to execute the below stash command  
git stash drop stash@{0}

To restore restore recently edited file we have to use the command   
git restore file name

Session Ended