

|  |
| --- |
|  |

**Git Guidelines and Best Practices**

# Table of Contents

1. INTRODUCTION……………………………………………………………………………………………………………………………1

2. BEST PRACTICE S OF GIT…………………………………………………………………………………………………………..2

# 

# Introduction

GIT is distributed version control system (DVCS) . This document provides Best Practices of GIT .

# BEST PRACTICES OF GIT

Following are the guidelines for developers:

1. **Branching :**

While creating a new branch, use proper conventions. Name of the branch should be proper and meaningful because it encloses all metadata about the branch.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |  | Here are some branch naming conventions :   1. Use grouping tokens (words) at the beginning of your branch names. 2. Define and use short lead tokens to differentiate branches in a way that is meaningful to your workflow. 3. Use slashes to separate parts of your branch names. 4. Do not use bare numbers as leading parts. 5. Avoid long descriptive names for long-lived branches. |

There are four types of branches :

* Master
* Release
* Develop
* Feature

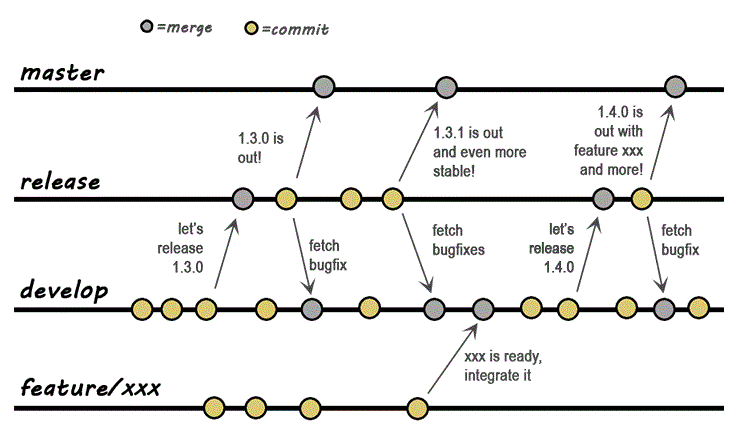
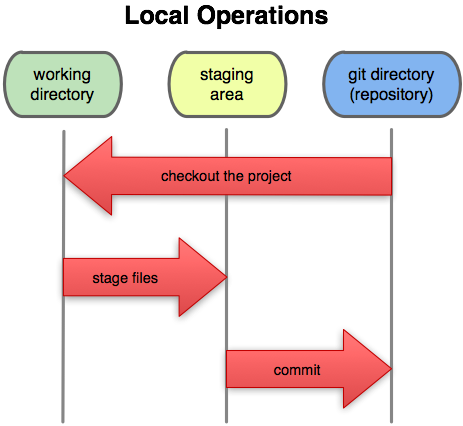


Fig: git branching

Branching is important so that we don’t make changes directly into master branch, it is best practice to take branch from master branch on local developer’s system and make changes in that branch once it is reviewed and merge that code in master.



**So staging by command “git add .” or ”git add --all” at regular interval is best practice**

**3. Commit and Commit messages :**

Git only takes full responsibility for your data when code is committed. If we fail to commit then we can run into trouble.

Git commit should contain JIRA issue-id.

Do regular commit with proper commit message. Good commit message serves at least three important purposes :

* To speed up the reviewing process.
* To help us write a good release note.
* To find out why a particular change was made to the code or why a specific feature was added.

“Commit Early And Often” comes under best practice.

Here are some best practices to be followed while committing in git.

* Do not assume the reviewer understands what the original problem was.
* Do not assume the reviewer has access to external web services/site.
* Do not assume the code is self-evident/self-documenting.
* Describe *why* a change is being made.
* Read the commit message to see if it hints at improved code structure.
* Ensure sufficient information to decide whether to review.
* The first commit line is the most important.
* Describe any limitations of the current code.
* Do not include patch set-specific comments.

**4. Always review code before pushing it to the master(main) branch :**

Review the changes made by the developer before pushing in main branch. Reviewing is important for improving both the overall quality of software and the developer’s skills.

Objectives of review are as follows :

* Defect-free, well-documented software.
* Software that compiles with enterprise coding standards.
* Teaching and sharing knowledge between developers.
* Other objectives are maintainability, security, consistent end-user documentation, adequate comments in code, complete unit tests, and scalability.

**5. Do choose a proper workflow :**

Best practice for using git is to pick the style of workflow that suits your project.

For initial stage of project or shorter projects, it is better to use branch workflow.

For larger projects and which includes more developers , it is advisable to go with Git workflow.

**6. Delete old branches as soon as they are merged** .

**7. Leave a clean commit history by pushing your code daily to git repository**

**8. Never delete unmerged remote branches**

**git merge -s ours obsolete-branch**

This will merge obsolete-branch into the current branch, but completely discarding the changes in the obsolete

branch.

I usually make it clear in the commit message for the merge that the branch is being discarded instead of a

true merge.

**git merge -s ours --edit obsolete-branch**

If the old changes are ever needed for reference or to be resurrected, it's as easy as checking out the last commit

on the merged branch and creating a new branch pointing to it.

**9. Making Tag(tagging) :**

Tagging in git is a great way to denote specific release version of code, or perhaps a way to refer to exactly one commit in history for some reason. It is best practice to tag stable builds

It contain the name, like v#.#.#, and a message too for better understanding. It follows this naming convention.

Where first digit is release number and next digits are it’s sub versions as per the changes.

**10. Merging :**

When branch reaches perfection, it’s ready to be merged. Merging is important at regular interval So that conflict arise during merge can be solved at early stage.

FIX merge Conflicts as follows

* Consult with the authors of the changes .
* Consult with the Team Lead
* Use a merge tool
* Communication between developers is must to solve merge conflict .