|  |  |
| --- | --- |
| **Guidelines FOR**  **Using GITLAB for Configuration Management**  Prepared By:  KPIT Technologies Ltd.  Delivery Excellence Group, PES |  |
| CONFIG ID : KPITPGAPA PA 476  VERSION : 1.0  DATE : 29-Jan-2021  Prepared For:  KPIT |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | |
|  |  | | |
|  |  |  |  |
|  |
|  |
|  |  | | |

Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ver. | Change Description | Prepared/  Modified By | Approved By | Date of Approval |
| 1.0 | Initial Document | Sagar Patankar  Jawad Ali | Samir Kulkarni  Shashidarshan Pai | 29-Jan-2021 |
|  |  |  |  |  |

TABLE OF CONTENTS

[1 INTRODUCTION 4](#_Toc62841097)

[1.1 Reference 4](#_Toc62841098)

[2 GITLAB CONFIGURATION MANAGEMENT TOOL 5](#_Toc62841099)

[2.1 GITLAB Repository 5](#_Toc62841100)

[2.2 Local Copy 5](#_Toc62841101)

[2.3 Using Git Commands 5](#_Toc62841102)

[2.4 GIT and GITLAB 6](#_Toc62841103)

[2.5 Using GITLAB through Web Browser 6](#_Toc62841104)

[2.6 GITLAB Flow 6](#_Toc62841105)

[2.7 Tortoise GIT 7](#_Toc62841106)

[2.8 GIT Project Structure 12](#_Toc62841107)

[3 GIT BEST PRACTICES 13](#_Toc62841108)

[3.1 GIT Repositories 13](#_Toc62841109)

[3.2 GIT Branching 13](#_Toc62841110)

[3.3 GIT Commits 14](#_Toc62841111)

[3.4 GIT Feature Commits 15](#_Toc62841112)

[3.5 Roles and Permissions 15](#_Toc62841113)

[3.6 Responsibilities 15](#_Toc62841114)

# INTRODUCTION

This document provides guidelines to be followed for storing products, project code and artifacts developed at KPIT using GITLAB Server Configuration Management system.

All products shall follow the naming conventions and folder structure as mentioned in this document.

## Reference

Following documents are referred while preparing these guidelines.

1. <https://kquest.kpit.com/kquest/projects/git/wiki>
2. <https://docs.gitlab.com/ce/gitlab-basics/README.html>

# GITLAB CONFIGURATION MANAGEMENT TOOL

GITLAB has community and commercial license. GITLAB manages files and directories over time. A tree of files is placed into a central repository. The repository is like file server, except that it maintains every change to the files and directories. This allows user to recover older versions of data or examine the history of how the data/files changed.

Some of the features that GITLAB provides:

* Directory versioning
* True version history
* Atomic commits
* Versioned metadata
* Choice of network layers
* Consistent data handling
* Efficient feature branching, merging and tagging
* CI/CD
* Analytics
* Issue tracking

## GITLAB Repository

GITLAB Repository is created on server and will be accessible from clients though a URL. The repository stores information in the form of a filesystem tree (a typical hierarchy of files and directories). Any numbers of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. When user commits the code, a version/snapshot is created in the local repository. Remote repository: A remote repository is hosted on a remote machine. GITLAB can access its repository across networks, which allows it to be used by user on different computers.

A Git repository is the .git folder inside a project. This repository tracks all changes made to files in your project, building a history over time.

## Local Copy

A GITLAB working copy also referred to as local copy is clone is an ordinary directory tree on the local system, containing a collection of files. A local copy also contains some extra files, created and maintained by GITLAB, to help it carry out the commands under .git folder. Local copy contains a subdirectory named *.git*, also known as the working copy administrative directory. The files in each administrative directory help GITLAB recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work.

User can create Git repository in one of two ways:

* Use a local directory that is currently not under version control, and turn it into a Git repository, or
* *Clone* an existing Git repository from GITLAB server

## Using Git Commands

To use Git, developers use specific commands to copy, create, change, and combine code. These commands can be executed directly from the command line or by using an application like TortoiseGIT. Here are some common commands for using Git:

* **git init** : initializes a brand-new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.
* **git clone** : creates a local copy of a project that already exists remotely. The clone includes all the project’s files, history, and branches.
* **git add** : stages a change. Git tracks changes to a developer’s codebase, but it’s necessary to stage and take a snapshot of the changes to include them in the project’s history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project’s history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.
* **git commit** : saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that’s been staged with git add will become a part of the snapshot with git commit.
* **git status** : shows the status of changes as untracked, modified, or staged.
* **git branch** : shows the branches being worked on locally.
* **git merge** : merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the master branch for deployment.
* **git pull** : updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.
* **git push** : updates the remote repository with any commits made locally to a branch.

## GIT and GITLAB

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. GitLab: Open source self-hosted Git management. GitLab offers git repository management, code reviews, CI/CD, issue tracking, activity feeds and wikis

## Using GITLAB through Web Browser

GITLAB web server makes GITLAB repositories available to clients via extension to HTTPS 1.1.

This result in a standardized robust system that can be accessed over a network from different geographic locations by using the built-in support for authentication and encryption based on SSL. To view the latest versions of all the stored files in the repository, a web browser can be used as a client and access to repository can done using HTTPS. Using the URL of the repository, the browser displays the latest versions of all the files that are stored in the repository.

However, to store files in the repository, one needs a client which retrieves the files from server using *clone* method. Once such client is “TortoiseGIT for Windows” which gets integrated with Windows explorer

## GITLAB Flow

Gitlab flow is very simple to understand and implement. It uses common git commands.

* **Create a branch** : Topic branches created from the canonical deployment branch (usually master) allow teams to contribute to many parallel efforts. Short-lived topic branches keep teams focused and results in quick ships.
* **Add commits** : Snapshots of development efforts within a branch create safe, revertible points in the project’s history.
* **Open a merge request** : Merge requests publicize a project’s ongoing efforts and set the tone for a transparent development process.
* **Discuss and review code** : Teams participate in code reviews by commenting, testing, and reviewing open pull requests. Code review is at the core of an open and participatory culture.
* **Merge** : Upon clicking merge, GitHub automatically performs the equivalent of a local ‘git merge’ operation. Git/GITLAB also keeps the entire branch development history on the merged pull request.
* **Deploy** : Teams can choose the best release cycles or incorporate continuous integration tools and operate with the assurance that code on the deployment branch has gone through a robust workflow.

## Tortoise GIT

TortoiseGIT is a client for the GIT based version control system.

Some of the features that TortoiseGIT provides:

* Shell Integration
* Icon overlays
* Easy access to GIT or GITLAB commands

TortoiseGIT is installed and it gets integrated with the Windows Shell. The required menus will be available in the Context menu of Windows Explorer as shown below. TortoiseGit supports user by regular tasks, such as committing, showing logs, difference between two versions, creating branches ,tags, creating patches and most of the commands supported by GIT or GITLAB repository

To work with GIT or GITLAB using TortoiseGIT, following operations can be performed.

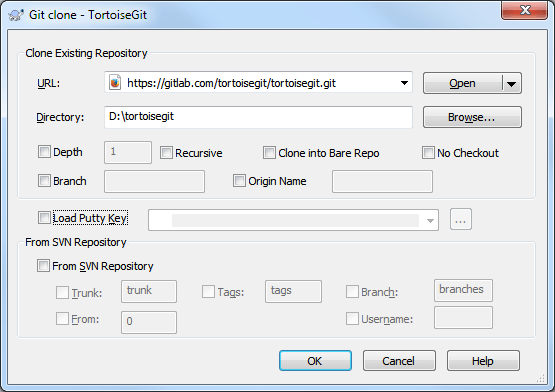


### Create Working Copy

To create a clone of a repository, right click on the system and select ‘Git Clone…” option.

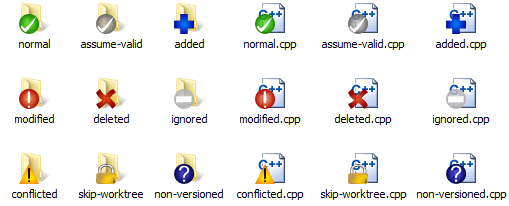


In the textbox “URL of repository”, give the path of repository where GIT or GITLAB repository exists. Refer GIT Repository for details.



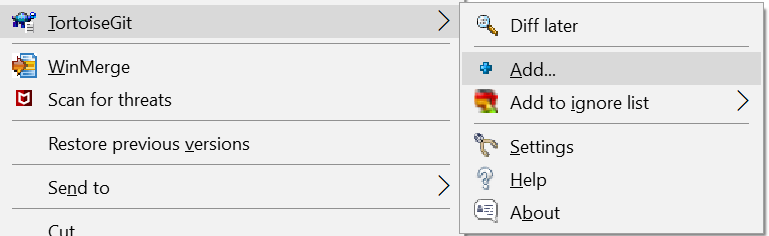
In the textbox “Directory” provide the path of local system where the local working copy shall reside.

After the clone is completed, contents of the repository will be copied to the given path. The Overlay icons provided by TortoiseGIT shows the status of the checked-out files/products.

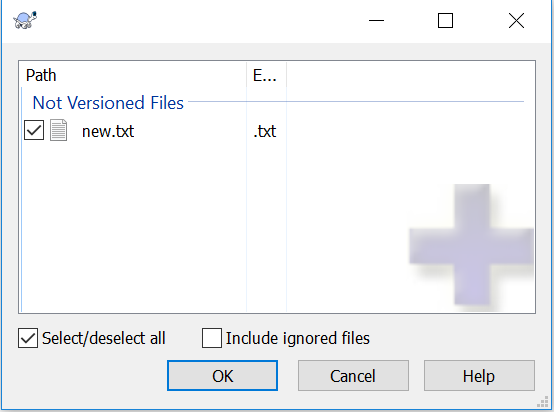


### Adding Contents

New files/folders can be added in the repository as we do in local file system. After adding the files, they need to be added to the SVN. Right click on the file (or blank space) and select TortoiseSVN 🡪 Add



Following window, appears



Select the files added newly and press ‘OK’.

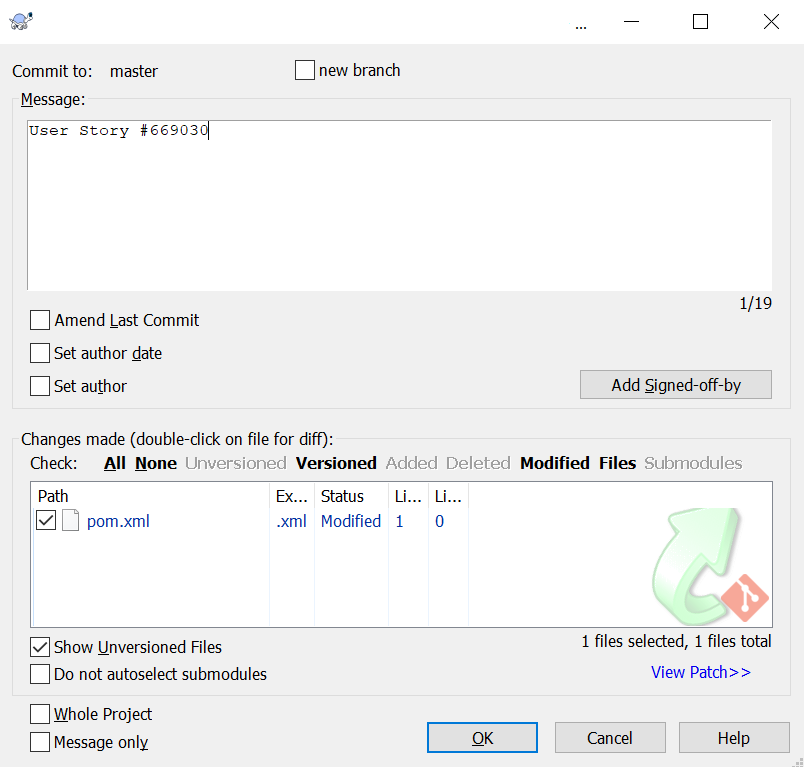
Overlapped icon after adding the file will look as below



### Committing and Pushing Changes

Changes can be made to the files in working copy as we do with any other files. Then later commit the changes back to the repository by performing an “GIT Commit”. Right click on the file/blank space and select “GIT Commit”. Enter the requirement reference/User story reference/Bug number /change description and relevant project user story/task/bug/feature number being implemented in the “Message” box and press OK.

**Note**: GIT Commit is committing to local repository. For adding/updating change to remote repository, user must use ***Push*** command

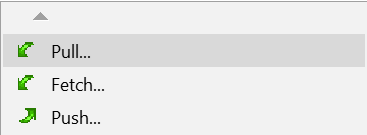


Overlapped icon after committing will look as below



### Updating Working Copy

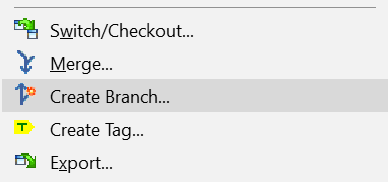
To reflect the changes being made to the repository by other developers into the working copy, “Pull” command must be used.



### Branching

Files/folders can be branched out to start a different version tree and implement changes. To branch a file/folder, “Create Branch” command has to be used.

Select the folder in your working copy which you want to copy to a branch, and then select the command TortoiseGIT 🡪 Create Branch

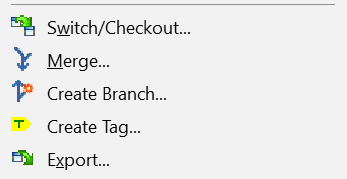


### Merging

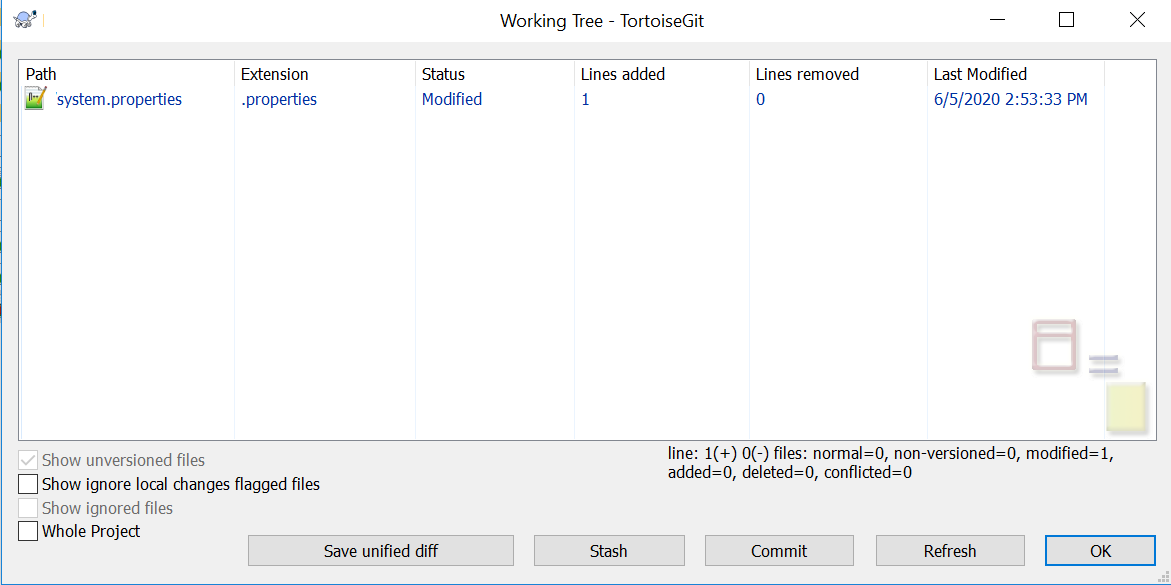
Changes made on different versions of the files/folders can be merged using “Merge” menu command.

To merge revisions, you need to go to working copy of the repository in which you want to receive the changes, and then select TortoiseGIT 🡪 Merge... from the context menu

GIT Merge command can be used to merge two different revisions of the same object or two different revisions of two different objects. The result of the merge will be stored in a different location where the merged objects are required to receive.



While merging, if there are any conflicts between the files being merged, GIT highlights those files as conflicted. In such cases, continue with the merge and then resolve the conflicts (to tell which changes to be available in the result) manually using “Edit Conflicts”. Then commit the files.



Overlay icon for conflicted files will be as shown below.

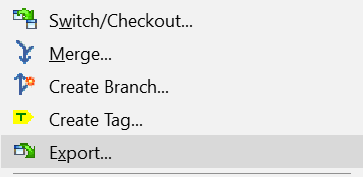


Note: It is recommended to use GIT merge feature for ASCII Files

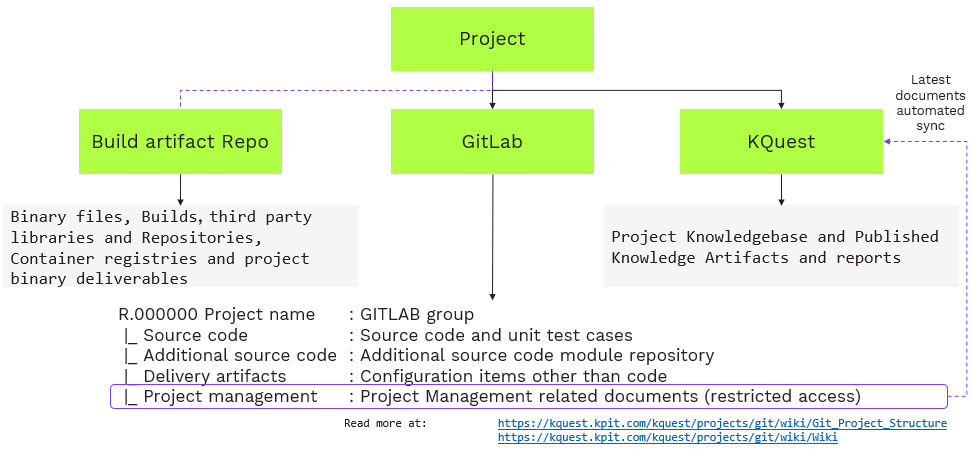
### Export

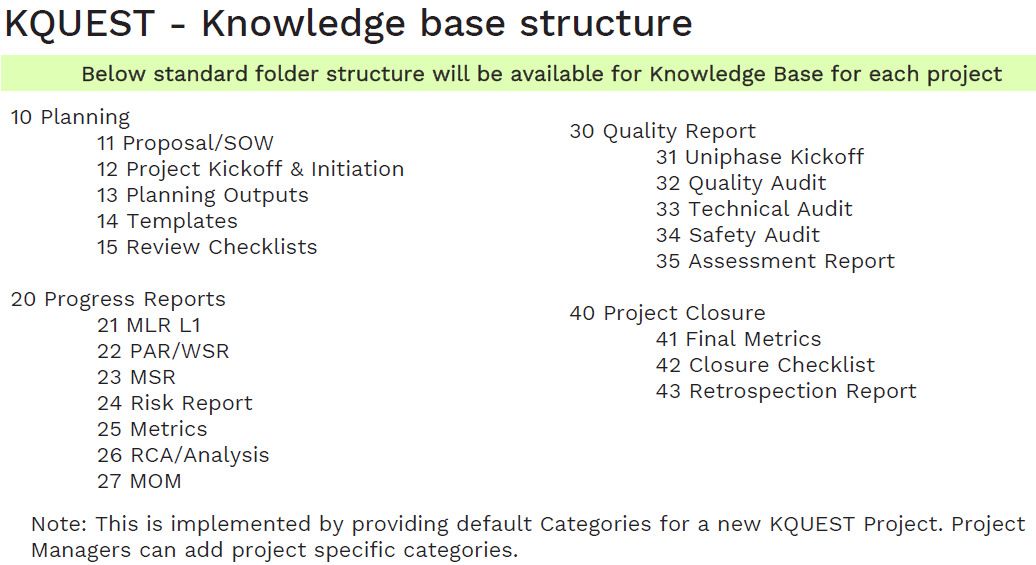
“Export” option in context menu can be used during releases to transfer files from specific branch to external folder which will ensure the complete transfer of files/products to be released in terms of number of files and its size. This will help in not missing any product during releases and it will not leave any trace of revision control (.git directories) in destination folder.

The working copy can be exported directly from the repository too. Use the Repository Browser to navigate to the relevant subtree in the repository, then use **Context Menu** **→** **Export**.



## GIT Project Structure





# GIT BEST PRACTICES

## GIT Repositories

Repositories are critical for code organization and access control

**Conceptual or functionality group per repository**

This could be name such as one per product, program, or library. Dividing code or artefacts at a later stage is difficult and might lead to rewriting of the public history or missing history. Structuring the repository or related repositories at early stage is important. ​The repository name should be unique to your account. Choose a descriptive name based on the product or project. Check Copyright and trademarks to ensure proper naming. As far as naming conventions, separate words with - (hyphen) as typing hyphen is easier than typing \_ (underscore)​

**Read and Write access control is at the repository level​​**

User has access to the entire repository, all branches, history. For fine grained access compartmentalize read/write access by structuring the code base into different repositories

**Shared files into repository required by multiple projects​**

Shared repository promotes code reuse and is highly recommended

**Avoid repositories for large binary files​​​**

Git sync activities become challenging with large binary files. Large repositories can be slow and cloning and branching operations become slow. Use Artifactory for storing binary files

## GIT Branching

Branching is one of the most powerful features. Branches are the perfect tool to differentiate different lines of development. Project teams should use feature branching extensively in the development workflows: for new features, bug fixes  
​  
GIT repository comprises of key branches - main and development

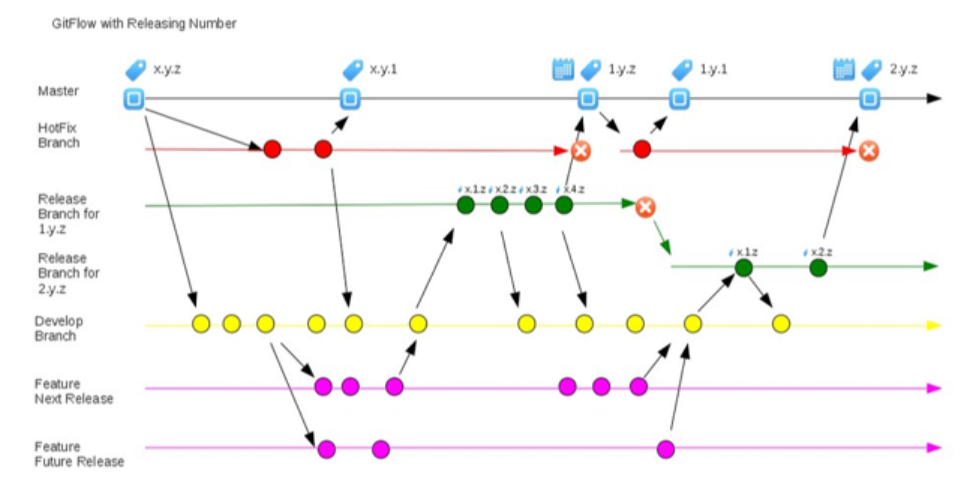
* ​The master branch at origin should be familiar to every Git user
* origin/master to be the main branch where the source code of “master” always reflects a production-ready state

Project/Product team can have additional types of branches  
**Feature branches**

* Feature branches are used to develop new features for the upcoming or a distant future release.
* They are merged and deleted after merge to the development branch or release branch  
  **Release branches**
* Release branches support preparation of a new production release  
  **Hotfix branches​**
* Hotfix branches are like release branches for a new production release.
* They arise from the necessity to act immediately upon an undesired state of a live production version
* Hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version

**Feature branch naming**  
Use a consistent naming convention for your feature branches to identify the work done in the branch. Some suggestions for naming your feature branches:

* bugfix/description
* features/feature-name
* features/feature-area/feature-name
* hotfix/description



## GIT Commits

**Commit related changes​​​**

A commit should comprise of related changes only

For example, fixing two different bugs should have two separate commits.

Small commits make it easier for reviewers and other developers to understand the changes and roll them back in case something goes wrong

**Author**

* GIT Commits should be performed by a recognized author with valid organization email id only.
* Committer commits code using the wrong email address and as a result commit has an unrecognized author.
* Unrecognized authors make it difficult to track the author for the specific code commits.
* Ensure GIT client is configured with the correct email address and linked to GITLAB user login name.  
  ​

**Comment**

* ​GIT Commit comments should have reference to User story, Task, Bug fix, Feature ID.
* Use the specific reference to the tracker ID in the comment such as Bug #634377
* Commit message should reflect user’s intention and not just the contents of the commit.
* Commit message should explain reason for the changes.
* An example of a good commit message is: “Combine templates to reduce duplicate code in the user views.”
* The words “change,” “improve,” “fix,” and “refactor” do not add much information to a commit message.
* For example, “Improve XML generation” could be better written as “Bug #634377 Properly escape special characters in XML generation.”

## GIT Feature Commits

Each feature should reside in its own branch, which can be pushed to the remote repository for backup/collaboration.

* Do not use branching off from master, feature branches should use development as their parent branch.
* On completion of feature, it should be merged into development.
* Features should never interact directly with master branch

## Roles and Permissions

For each repository, following roles with given permissions shall be created.

|  |  |  |
| --- | --- | --- |
| **#** | **Role** | **Permissions** |
| 1 | Project Configuration Manager | Read/Write permission for entire repository |
| 2 | Project Team Member | Read permission for “master” and “tags” and read/write permission for “branches” |

## Responsibilities

Following are the responsibilities of each role mentioned above.

1. *Project Configuration Manager* shall create (Refer: Adding Contents) different folders for each product in the “branches” and as well as in “master” after a uniphase kick-off or SCR Approval.
2. *Project Team Members* shall create a branch (Refer: Branching) of the product from “master” into a new version (folder) in “branches” after a Uniphase kick-off or after approval of SCR(s). If it is SCR, mention the SCR number in the message panel, if it is first version, then mention ‘first version’ in the message panel.
3. *Project Team Members* shall create different folders for different versions in the “branches”.
4. If more than one *Project Team Members* are assigned for same product-version, every Team member shall create his/her own branch folder. Naming convention for each Team member’s folder shall be “Ver x.x yyyy” where x.x represents version and yyyy represents unique field to distinguish between different Team member’s folders.
5. *Project Team Members* shall perform daily “Commits” (Refer: Committing Changes) to the changes happening in the “branches”.
6. For every commit, details shall be provided in the message panel, to represent the operation being performed.

For Example, “Screen is completed for BD 001 1.1 and product is ready for stores” represents that the Ver 1.1 of product identified by BD 001 is ready to be merged back to master.

1. Following table provides guidelines for typing message while committing changes into branches.

|  |  |  |
| --- | --- | --- |
| **#** | **Stage** | **Message** |
| 1 | Creating branch for first version | Creating branch for <Config ID> to create the initial version. |
| 2 | Creating branch for subsequent versions | Creating branch for <Config ID> to create version <new Ver> as per SCR\_XXX |
| 3 | Daily Changes | <Changes specific message> |
| 4 | Products ready for Technical Review | <Config ID>, <new Ver> is ready for technical review |
| 5 | Technical Review is completed | <Config ID>, <new Ver> is reworked for technical review comments. |
| 6 | Products ready for Screen | <Config ID>, <new Ver> is ready for screen |
| 7 | Products ready for Store | <Config ID>, <new Ver> is ready for store. |

Note: Based on project needs, message text mentioned above can be changed.

1. Before sending the products to review/screen, if more than one folder belonging to different team members exists in the branches. *Project Configuration Manager* shall merge the folders of different team members in Branches (in any one team member’s folder). If any conflicts exist, then the same shall be resolved (discuss with the team and project leader if required) before sending the products to review/screen.
2. After completion of screen, *Project Team Members* shall send the SAR and TR Reports (if any) to Project Configuration Manager and inform *Project Configuration Manger* to move the products to “master”.
3. *Project Configuration Manager* shall store SAR and TR reports in “Reports” folder of trunk and shall perform the checks on the product(s) as per CMCheck.txt file which is available in common repository. Filled in CMCheck.txt shall be stored in “Config ID” folder of trunk.
4. *Project Configuration Manager* shall then merge (Refer: Merging) the changes of “branches” and store into the “master” as per the folder structure.

Note: There will not be different folders for different versions in “master”.

1. The product is treated baselined once it is stored in “master”.
2. *Project Configuration Manager* shall branch (Refer: Branching) the files and folders to be released along with supporting files like “Readme.txt” & “Release Checklist” into “tags” Before releasing to customer. Creating “tags” is not mandatory.anding Stores"pening in the " orm daily "for different versions, instealders.