Git workflow guideline

Current guideline should help us to achieve a more clean versioning codebase and streamline the process of releasing daily changes.

These guidelines assume that we have the following setup:

* Application’s repository has a release branch named *main*. This branch contains periodic releases that should be stable enough. No broken or work in progress code should be pushed here.
* For unstable versions of the application we have a second branch, that is derived from main, called *development*. This branch can be temporarily unstable and also some work in progress code can be pushed (thus not strongly recommended). This should usually be the branch where people are integrating their feature code once they finished.

# 1. New feature development

## 1.1 Create new branch for each feature

One thing that we may ask though is how do we know which tasks are new features. As a rule of thumb, a task should have the following characteristic in order to be considered a new feature:

* The new code is adding new standalone functionality to the application. If the new version of the application can be tested without any additional work, this means we added a new feature.
  + Example: we created some config file for the application, but there is no code that’s using that file. So in this state we cannot test the whole config feature until other coding tasks are done. If we want to commit just the config file, we’ll not create a special branch only for this thing. Instead, we should create a branch for the config feature, commit the file there, continue working and making other commits until we have the whole feature working, then merge everything into the main branch.
  + Bugfixes should also not be part of new features. They should be commited directly to the *development* branch (or dedicated branches if they are more complex)
  + Refactorings should also not be part of new features, unless they change some of the application’s user facing behavior. Behind the curtain refactorings should be committed to *development*, while those that change user behavior in application should be treated as new features.

## Git workflow:

1. Checkout *development* branch:
   * *git checkout development*
2. Pull the latest changes
   * *git pull*
3. Create new branch:
   * *git checkout -b new\_branch\_name*

### Branch naming convention

Usually all development code should have a task assigned, which has an ID. In this case it would be a good practice to name the branch as follows:

* *TaskID\_summarized\_task\_name* (E.g.: *PTE\_120\_implement\_config*)

## 1.2. Perform all commits on feature branch

During development of the new feature, all commits should be performed on feature branch. This should include fixes for bugs that are found during testing phase, or other refactorings resulted from code review or other sources.

## 1.3. Commits should have descriptive messages

As a rule of thumb, it’s a good practice for each commit to contain at least 1 row description of what has been implemented/changed/fixed. However, this is only minimal, as for consistent commits it would be good to have at least a few rows of description (let’s say 3 to 5 rows would be a good reference, or more if it’s needed).

## 1.4. Create merge requests

After development of the new feature is done, the changes can be pushed on the server:

* *git push -u origin <branch>* (where branch should have the same name as the local one)

Then we should create a merge request on the server with:

* Source branch: *TaskID\_summarized\_task\_name* (feature branch)
* Destination/Target branch: *development*

## 1.5. Perform code review

On the previously created merge request someone can review the changes and provide feedback.

**Note**: Changes suggested on code-review/test should be committed on the local feature branch and pushed to the remote one, until feature is ready to be shipped.

At the end of a code review, the reviewer should give the ship greenlight by giving a ***Thumb Up*** on the merge page.

Now the code should be ready for shipping.

## 1.6. Shipping code (merge into *development*)

Naturally, the next step that should be performed is merge into development branch. This is a pretty important step, as now the new code will be integrated with other features that most probably were developed in parallel by other developers. As such, here are some good practice steps that should keep us from getting into trouble:

1. **Try to keep the feature size as small as possible**. Big features require extensive time to develop. Meanwhile, the other developers may push a lot of changes on development branch, which naturally would generate a lot of conflicts during merge of your new feature.
2. **Perform merge with squash option**:
   * *git checkout development*
   * *git pull*
   * *git merge --squash TaskID\_summarized\_task\_name* (feature branch)

This merge should compress all commits from the feature branch into a single one, and allow to create a new message for that commit, which can summarize what has been implemented in that feature.

1. **Resolve merge conflicts carefully**
2. **Test the application after merge**. Implemented feature should be carefully tested after merge in order to validate that nothing got broken after integration. Another good practice is to run some smoke tests on the core features to validate they’re working.

If during step 4 some bugs are discovered, they should be fixed either in the merge commit, or in subsequent commits, but on development branch this time. No need to go back to the feature branch, as this could complicate future merges.

## 1.7. New feature workflow summary

1. **Create new branch:**
   * *git checkout development*
   * *git pull*
   * *git checkout -b TaskID\_summarized\_task\_name*
2. **Code, code, code...and commit on local branch.**
3. **Push changes remote for merge request:**
   * *git push -u origin TaskID\_summarized\_task\_name*
4. **Create Merge Request:**
   * Source branch: *TaskID\_summarized\_task\_name* (feature branch)
   * Destination/Target branch: *development*
5. **Perform Test/Code Review**
   * Commit all after test/code review changes to *TaskID\_summarized\_task\_name* and push them on server
6. **Perform merge with squash option:**
   * *git checkout development*
   * *git pull*
   * *git merge --squash TaskID\_summarized\_task\_name* (feature branch)

# 2. Bugfixing

This section refers to those bugs that are fixed after a feature has been merged to development/master branch. Bugs that are discovered during test/code review phase while developing a new feature should be committed to that feature branch before merge.

There could be 2 types of defects discovered after a new feature delivery:

## 2.1 Easy to fix bugs

Usually these fixes require small changes, like a few lines of code in 1-2 files. Thus, there is no need to create different branch for them, they could be easily committed directly to *development* branch.

**Steps**:

1. **Pull latest changes**
   * *git checkout development*
   * *git pull*
2. **Do necessary fixes and commit**
   * *git commit -m “[PTE-TaskID] Summarized Task Name: Description of the performed changes…” (****P****iano-****T****iles* ***E****GC -> PTE)*
3. **Update from remote in case there are new changes**
   * *git pull --rebase*

**Note**: Be careful as there could be conflicts. They should be resolved in the same way as when performing a merge operation.

1. **Push changes on remote**
   * *git push*

## 2.2 Complex bugs

These are normally issues that require multiple changes and sometimes could also include redesign or refactoring. In this case it’s advisable to create separate branch:

1. **Create new branch:**
   1. *git checkout development*
   2. *git pull*
   3. *git checkout -b TaskID\_summarized\_task\_name*
2. **Perform fixes and commit on local branch.**
3. **Push changes remote for merge request:**
   1. *git push -u origin TaskID\_summarized\_task\_name*
4. **Create Merge Request:**
   1. Source branch: *TaskID\_summarized\_task\_name* (feature branch)
   2. Destination/Target branch: *development*
5. **Perform Test**
   1. Commit all after test changes to *TaskID\_summarized\_task\_name* and push them on server
6. **Perform merge with squash option:**
   1. *git checkout development*
   2. *git pull*
   3. *git merge --squash TaskID\_summarized\_task\_name* (bugfix branch)

# 3. Refactoring

This operation should basically be similar with bugfixing. Small refactorings could easily be worked and committed on development branch, while for more complex ones it’s advisable to work on separate branch.

For detailed workflows we could use those from section 2.