MergeBoard

Git Workflow Cheat Sheet

Version 1.0

Initializing a Repository

1. Initialize your git folder.

   Fetch an existing git repository.

   ```
   git clone <url> [folder]
   ```

   OR

   Create an empty git repository.

   ```
   git init [folder]
   ```

2. Change into the folder of the repository.

   ```
   cd <folder>
   ```

Making Changes

New features should be developed in a separate feature branch which is later merged back into the development branch (usually named `master` or `main`).

1. Switch to the development branch.

   ```
   git checkout <main/master/...>
   ```

2. Get latest changes made by others.

   ```
   git pull
   ```

3. Create a branch with a custom name.

   ```
   git checkout -b <feature_branch>
   ```

4. Make your changes

Committing Changes

After making changes to the code you need to split them into commits. You can achieve this by repeating the following process.

1. Check which files you have changed.

   ```
   git status
   ```

2. Restore files you didn't intend to change.

   ```
   git restore <path>
   ```

3. Stage all changes that belong into a single commit using these commands:

   Add/stage complete files.

   ```
   git add <file/path>
   ```

   OR

   Interactively decide which part of a file you want to stage.

   ```
   git add -p [file]
   ```

   OR

   Stage all changes in existing files.

   ```
   git add -u
   ```

4. Double check your staged changes.

   ```
   git diff --staged [options]
   ```

`-w`	Ignore whitespace changes
`--word-diff`	Compare words, not lines

5. Unstage accidentally staged files (or all).

   ```
   git reset [path]
   ```

6. Commit your staged changes.

   Commit with a one-line message.

   ```
   git commit -m <message>
   ```

   OR

   Type the message in an editor.

   ```
   git commit
   ```

Best Practices

- A commit should do one thing. Split unrelated changes into multiple ones.
- Complex features can be developed first and later split into multiple commits by using `git add -p`.
- Commit messages should sum up the change in the first line. The following lines can be used for additional context.

Preparing a Merge/Pull Request

1. Verify that all changes were committed.

   ```
   git status
   ```

2. Double check your commit messages.

   ```
   git log
   ```

3. Push your changes to the remote server.

   ```
   git push -u origin <feature_branch>
   ```

4. Create a new pull / merge request (depends on the used DevOps software).

Best Practices

- Use the merge / pull request description to provide all the necessary information to understand your changes, e.g. a link to your task/documentation or a reasoning why you chose this approach.

Reviewing Code

The things to look out for in a code review depend on your project. The following tips can be used as a starting point.

Tips

- Does the code solve the task? Did the author understand the problem?
- Is the code difficult to understand?
- Would you solve the problem in the same way? Why not?
- Is the implementation maintainable?
- Can you spot any shortcuts, missing validations, edge cases, ...?
- Always ask if you don't understand something.

Fix Your Latest Commit

1. Change the code as desired/requested.
2. Stage all intended changes. Check Committing Changes for more info.

   ```
   git add -p
   ```

3. Double check your staged changes.

   ```
   git diff --staged [options]
   ```

`-w`	Ignore whitespace changes
`--word-diff`	Compare words, not lines

4. Add the staged changes to the latest commit.

   ```
   git commit --amend
   ```

5. Send your modified commit to the server.

   ```
   git push -f
   ```

Fix Any Commit(s) - Rebase

A rebase workflow can be used to make larger changes to commits or change the number of commits.

1. Start an interactive rebase with the development branch as base.

   ```
   git rebase -i <development_branch>
   ```

   Replace the word `pickup` in a line to modify the commit - or delete the line if you want to drop the commit.

`edit`	Edit the content of a commit
`reword`	Change commit message only
`squash`	Merge content into previous commit

2. Git pauses the rebase for each commit marked for editing. Follow the steps of Fix Your Latest Commit to modify the commit. Afterwards continue the rebase.

   ```
   git rebase --continue
   ```

3. In case of a mistake, abort the process and return to the state before rebasing.

   ```
   git rebase --abort
   ```

4. Send your modified commit to the server.

   ```
   git push -f
   ```

Tips

- In case of a conflict: Resolve the conflicts in the code, stage your changes and continue the rebase **without committing them**. Git will do this automatically for you.

Fix Any Commit(s) - Fixup

A fixup based workflow is a faster alternative to rebases if the changes are small and don't conflict with each other. See mergeboard.com/fixup.

1. Modify the code to address the feedback.
2. Stage all changes related to one faulty commit.

   ```
   git add -p
   ```

3. Double check your staged changes.

   ```
   git diff --staged [options]
   ```

`-w`	Ignore whitespace changes
`--word-diff`	Compare words, not lines

4. Create a fixup commit for it.

   ```
   git commit --fixup <commit_hash>
   ```

   The `commit_hash` needs to point to the faulty commit you want to edit.

5. Repeat the previous step to create a fixup commit for each faulty commit.
6. Move the changes from the fixup commits back into the faulty commits.

   ```
   git rebase -i --autosquash
   ```

   You don't have to change anything, just close the editor. Git knows what to do.

7. Verify that your git history doesn't contain any fixup commits any more.

   ```
   git log
   ```

8. Send your modified commit to the server.

   ```
   git push -f
   ```

Test Your Commits

It is easy to introduce bugs when editing the git history. You may therefore want to test each commit afterwards.

1. Run the tests for each commit and stop when encountering an error.

   ```
   git rebase --exec '<test_cmd>' <development_branch>
   ```

   `test_cmd` depends on your build system. Any command returning an exit code of zero on success and a non-zero value on failure can be used, e.g. `make test`.

Stashing Changes

If you want to temporarily preserve changes without committing them, use `git stash`.

1. Remove changes from your files and preserve them in the stash.

   ```
   git stash save [message]
   ```

2. List your stashed changes.

   ```
   git stash list
   ```

3. View info about a stashed change.

   ```
   git stash show [options] [stash@{X}]
   ```

`-p`	Show code changes instead of file statistics

4. Apply the specified (or last) stashed changes back to your files.

   ```
   git stash apply [stash@{X}]
   ```

5. Get rid of the stashed changes if you don't need them any longer.

   Forget the specified (or last) stash.

   ```
   git stash drop [stash@{X}]
   ```

   OR

   Forget all stashed changes.

   ```
   git stash clear
   ```

Working With Branches

- List branches

  ```
  git branch [options]
  ```

`-a`	List local & remote branches
`-r`	List remote branches only
`-v`	Show last commit for each branch

- Create a branch without switching to it

  ```
  git branch <branch-name> [start-point]
  ```

- Create a branch and switch to it

  ```
  git branch -b <branch-name> [start-point]
  ```

- Switch to a branch

  ```
  git checkout <branch-name>
  ```

- Switch to your last branch

  ```
  git checkout -
  ```

- Delete a branch

  ```
  git branch -d <branch-name>
  ```

Working With Tags

- List tags

  ```
  git tag [options]
  ```

`-n`	Show tag annotations

- Create a tag

  ```
  git tag [options] <tag-name>
  ```

`-m <message>`	Add annotation message

- Delete a tag

  ```
  git tag -d <tag-name>
  ```

Do You Need More Help?

Check out our workshops at mergeboard.com/workshops