

Git and Github Usage

 freeletics.engineering/2017/05/15/git-and-github-usage.html

Using Git and GitHub in your Android team

At Freeletics we use Git and GitHub on a daily basis as vital tools for all our crucial projects. To help the entire engineering team contribute fast and effectively, we abide by a set of rules that we partly developed by ourselves, and partly imported from best practices in software development. Here are a few that you might find useful in co-ordinating your own workflow among your engineering team.

Branching

Main Branches

- *master* - reflects the current app store version
- *develop* - reflects the latest stable development changes

Naming Branches

A branch name should consist of the ticket number (if available) and a meaningful name. A dash should be used as a separator between words. Branch names should be lower case only.

Examples of good naming:

- *jira-764-invite-friends-screen*
- *jira-152-wear-popup*

Prefixes

Some of the branches can contain prefixes like:

- *release* - release branches (e.g. *release/3.4.1*)
- *hotfix* - when it's a fix on production (e.g. *hotfix/4.0.1*)

Those prefixes will trigger a release job on our CI server Jenkins, with each push to the branch.

Branch Owners

If you are the creator of a branch, you are also the owner. You are responsible for taking care of it over its entire lifetime. Responsibilities include:

- Rebasing on a regular basis (make it a habit every morning so you don't have any merge issues later on).
- Make sure that any Localization changes are kept up-to-date, especially in release branches.
- Deleting the branch when it's no longer needed.

Force-Pushing Branches

- Only force-push a branch if you are the owner.
- All other cases need the involvement of all developers contributing to that branch *#communicate*.
- The branches *develop* and the *master* should never be force-pushed. For additional safety they are also

set as protected branches on GitHub.

Credits

The whole system of Git branching is based on the widely used model of [Vincent Driessen](#).

Commit Messages

1. Use a short meaningful title (max. 80 characters)
2. Use the description for detailed explanations
3. Use International English
4. Use present tense in imperative form (example: Add translations for Invite Friends screen).
5. Use sentence case
6. The ticket ID should be mentioned at the end of the description (separate line), not the title. Such meta data will work with the integration of the Bug Tracker system.
7. Squash commits when it makes sense
8. Commits should be atomic

Example of good commit message:

```
Add translations for Invite Friends  
screen
```

```
JIRA-713
```

Pull Requests

Pull Requests and code reviews should ideally be just a quick sanity check of your work. Remember this at all times and the process will be as hassle-free as it can get, for all developers that are involved. We use GitHub for reviewing Pull Requests, so some of the items belong to GitHub features and could be missing in other code review tools.

Why are we doing Pull Requests?

- Eliminating bugs at an early stage
- Improving code quality
- Increasing knowledge about the project
- Learning from each other
- Making sure that our definition of Done is met
- Not about naming and shaming < this is crucial

Rules

- Shorter Pull Requests = faster integration
- Avoid bulk commits by issuing small, atomic commits
- Quality means sticking to the same process for every single Pull Request (use templates).

Before

- Get your architecture and design of the feature and fix challenges by the team before you even start coding. This avoids long discussions later on, conversations that may involve further Pull Requests.
- Choose the correct target branch
- Remember to use the [Boy Scout Rule](#) whenever you can.

Submission

- Assign and request a reviewer. Assignment shows in the overview of the Pull Request. An unassigned Pull Request means that it's free to pick up.
- Before assigning, review the Pull Request yourself (check all points on the template).
- Communicate assignment to your team mate. Don't rely on GitHub notifications *#communicate* in a separate channel (e.g. Slack)

Think of this before doing a Pull Request on develop: The feature or fix should be production-ready, so can be part of the next release.

Labels

In order to go through the list of Pull Request (PR), we have a different group of labels:

1. Channel

- *beta* – the PR should be merged to the current beta (next release) branch
- *hotfix* – the PR should be merged as a hotfix to hotfix branch

2. Status

- *wip* (work in progress) - the PR is not ready to be reviewed in detail but the creator already wants feedback
- *ready* – the PR is ready for review and can be picked up
- *blocked* – the PR is blocked and cannot be reviewed at the moment

3. Complexity. Defines the complexity of the PR in order to evaluate the PR faster

- *low*
- *medium*
- *high* - If complexity is high, it alarms that probably something is wrong in the PR and creator should revise it.

4. Team. Defines to which feature team the PR belongs. It helps to split visually PRs by different logical groups.



- *training*
- *community*
- *monetization*
- *etc.*

Keep the wip Pull Requests to a minimum, especially if they don't at this stage need to be visible. This will give a better overview and saves the build power on Jenkins for PRs that actually need it. Remember that ideally you get your concept, architecture or class design challenged before you start implementing

Reviewing

- As a reviewer, communicate when you start and are done reviewing
- As a creator, don't push while someone else is reviewing *#efficiency*
- Stay friendly and constructive during code reviews (a good source on the right tone)
- If discussions get too long, take it offline (and add the results in the PR)
- If you find an inspiring piece of code by a colleague, share it with the team
- When you are done with review, submit it (either as approved or as requiring changes)
- Unassign yourself and assign PR back to creator

Merging

- Approval is done by , or  or :shipit: or by submission review as accepted (using GitHub review functionality).
- Merging should only be done by the creator of the Pull Request.
- Do not merge without review by other team mates (or while any comments remain open).
- All items in the Pull Request template should be checked off.
- Must be tested on a device
- Code must not be merged before CI builds successful
- Do not merge if unit or integration tests are failing
- Use a clear commit message
- We decided to use GitHub's squash and commit functionality to have a linear history and to avoid having all commits from the branch. But feel free to use another option.
- Delete the merged Pull Request branch.

Pull Request Templates

Pull Requests are inspected by different team members, allowing for an increasing amount of test criteria to be covered. The lists below create an important guide for developers to systematically test. For your daily work, stick to the Pull Request templates. Using a ~~Strikethrough~~, cross out any items that might not be relevant for the specific Pull Request.

Example of the PR Template:

Description

- Added tracking events for Wear Popup

To test

- Go to Debug Panel
- Choose item Popups
- Select Open Wear Popup
- Wear Popup should appear
- Check logs for tracking events

Links to relevant tickets

- JIRA-199

Definition of Done

- [x] Ticket number in commit message
- [x] No warnings added (Sonarqube, Lint)
- [x] Tests added
- [x] All (unit) tests pass
- [x] Test coverage should not drop
- [x] Tested on device
- [x] No regressions (no crashes or critical bugs)
- [] ~~Reviewed by designer/PM~~
- [x] Changelog updated
- [] ~~Strings added to mobile-localizations repo in <_insert here="" the="" link="" to="" pr="" in="" mobile-localizations="" repo_="">~~

Automation

All that's mentioned above will always be supported by as much automation as possible. For instance, like using linters to reduce the amount of attention needed to focus on style violations all the time. We always challenge ourselves to see what else can be automated.

Resources

1. <https://github.com/thoughtbot/guides/tree/master/code-review>
2. <http://kevinlondon.com/2015/05/05/code-review-best-practices.html>
3. http://blogs.atlassian.com/2010/03/code_review_in_agile_teams_part_ii/
4. <https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/>
5. <http://nvie.com/posts/a-successful-git-branching-model/>
6. http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule