Project VIABLE—Git guidelines

Commit messages

This section is somewhat optional—ignore it if you would like, but it's recommended for the sake of consistency and clarity.

A commit message should consist of a single sentence, capitalized, but without a period.

✗ Has period and is not capitalized

```
remove nasal demons.
```

✓ No period and capitalized

```
Remove nasal demons
```

It should also be written in the imperative mood, as if instructing Git to execute the content of the message.

✗ Not imperative

```
Fixed problem
```

✓ Imperative

```
Fix problem
```

This is to keep consistent with the commit messages automatically generated by Git. E.g., if you merge a branch `A` into a branch `B`, the default message generated by Git is `Merge branch 'A' into B`.

Commit messages should usually be only around 50–70 characters in length. But in some cases, you may have made a change for reasons that are not apparent just from the code and would like to further explain *why* you made the changes. In this case, you should make use of the commit

message body; this can be done by putting a blank line in the commit message, followed by the body text. The command `git commit` without the `-m` option will open your editor—set with the `core.editor` git config option—and allow you to easily write a multiline commit description.

For example, consider a commit message for a commit that adds bounds checking to prevent a buffer overflow. The following commit message is too long, making it harder to immediately tell *what* it changes:

✗ Very long commit message

```
Add bounds checking to a buffer array that stores user
input to prevent a buffer overflow that was a large
security vulnerability
```

On the other hand, this commit message has a short description of the change, followed by a body with more detail:

✓ Short commit summary with a detailed body

```
Add bounds checking to input buffer

Since users tend not to create items with names longer
than 8192 characters, this is *typically* not a
problem. But someone could theoretically send a message
longer than that and cause a buffer overflow. Given
that we have our stack set to be executable, this could
easily lead to remote code execution.
```

In some cases, you may wish to express this kind of thing in a code comment; only do this if the comment is still relevant in the current state of the code.

✗ Describing the reason for a change in a code comment

```
var x := get_value()
# There used to be a line here that would check the
# type of `x`, but it was unnecessary, so I removed it.
```

✓ Describing the reason for a change in the commit body

```
Remove unnecessary type check

`get_value()` is already guaranteed to return a value
of type `Value`, so it's not necessary to check it.
```

Branch types

`main` The `main` branch represents the "canonical" state of the project. I.e., if someone—in particular, a user rather than a developer—wants to try out VIABLE, then they should be able to clone the `main` branch and get it running out of the box.

`dev` The `dev` branch is the working state of the repository from the developer's perspective. The current state of `dev` isn't guaranteed to be fully working, but the changes there have to be approved, so it's unlikely to be fully broken unless the current sprint inherently necessitates that the repository be broken.

`topic/*` Topic branches should be given a name in the form `topic/<name>`, where `<name>` is a name describing the purpose of the branch. If the branch is clearly associated with a specific issue, then you may use GitHub's "create a branch" feature to create the branch; when doing this, prepend `topic/` to the name of the branch. For example, if the default branch name is `123-fix-stuff`, then rename it to `topic/123-fix-stuff`. You may also choose anything else for `<name>` that makes sense to you as long as it doesn't overlap with a topic branch that currently exists.

Part of the reason for doing this is that it ensures all topic branches are listed *below* the `dev` branch on GitHub, but it also makes it easy to write commands that automatically pick out topic branches. For example, if you want to fetch all topic branches, you can use the command `git fetch origin 'refs/heads/topic/*:refs/remotes/origin/topic/*'`.

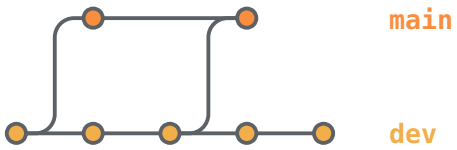
Topic branches are short-lived branches intended to be merged into `dev` via pull request. Once a topic branch is merged, it should be deleted—typically, this will be done with the "delete branch" button in the pull

request page once it's closed. The short-lived nature of these branches means that their name isn't *that* important; it's best to try not to use a name that has been used before, but it's not a big deal if you do.

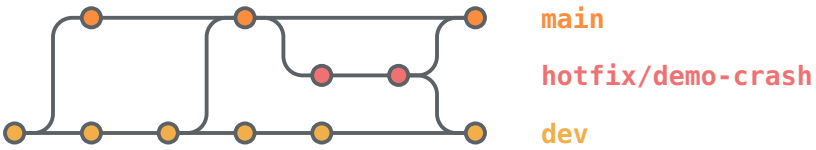
`hotfix/*` The name of a hotfix branch is in the form `hotfix/<name>`, where `<name>` describes the thing being fixed. These follow the same naming rules as topic branches, except prepended with `hotfix/` instead of `topic/`. See above in the `topic/*` section for an explanation.

If an outstanding critical bug is found in the `main` branch and it needs to be fixed before the next time `dev` can be merged in, then a hotfix branch can be created to address it. This should then be merged into both `main` *and* `dev` via pull request. Once it has been merged into both, it should be deleted.

For example, let's imagine we plan to demo the project in a week, and we've just finished implementing a feature, so we decide to merge `dev` into `main`. Over the next few days, we continue working on some new features in the `dev` branch, so the history looks like this:



But a day before the demo, we realize that it crashes when running it on the demo platform! Then we can fix that on a hotfix branch. We create a branch off of `main` called `hotfix/demo-crash`. Once we've managed to fix the crash in the hotfix branch, we create two pull requests: one into `main` and one into `dev`. Once those pull requests have been merged, the history looks like this:



The `main` branch

As much as is reasonably possible, `main` should *always* point to a commit that works correctly. This doesn't necessarily mean that it should have

zero bugs, but there should be no “compile”-time errors, no runtime errors during normal usage, and no tests failing.

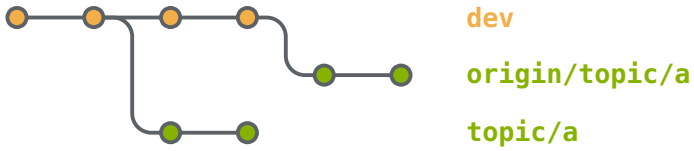
If we have reached the end of a sprint, then we will discuss *as a team* whether the current state of `dev` is in good enough shape to be merged into `main`. If we agree, then one of us will create a pull request from `dev` to `main` and merge it in.

Updating topic branches

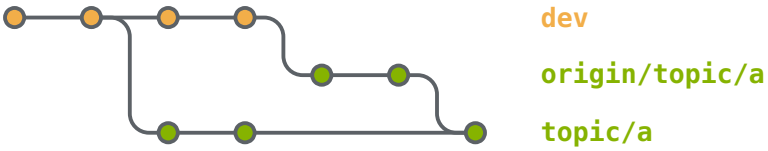
Let's say you have checked out the topic branch `topic/a`, which is being worked on by someone else. When pulling these changes, always use the command `git pull --rebase`. Alternatively, if you set the config option `pull.rebase` to `true`, then `git pull` by itself will automatically use the rebase strategy.

Let's say that `topic/a` originally looked like this:

In order to keep their branch up to date, they rebased `topic/a` on top of `dev`. If you were to fetch these changes without updating your local branch, then your local repository would look like this:



By default, `git pull` will fetch these changes and then merge `origin/topic/a` into `topic/a`, which will produce something that looks like this:



This is undesirable, since it results in an extra merge commit with two different versions of the same branch as parents. On the other hand, using `git pull --rebase` will give the history:

which is much cleaner.

Incorporating changes from `dev`

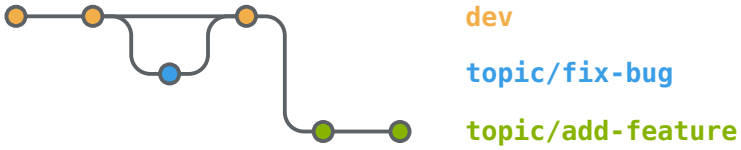
Let's say you want to add some feature, so you create a branch from `dev` called `topic/add-feature`. At the same time, someone else fixes a bug in the branch `topic/fix-bug`:

While you're still working on `topic/add-feature`, `topic/fix-bug` is merged into `dev`:

You realize that you need the bug fix to keep working on the feature. To include the changes, do the following if possible:

- Pull the new changes into `dev`, then switch back to `topic/add-feature`.
- Rebase `topic/add-feature` onto `dev` with the command `git rebase dev`.
- If this resulted in a merge conflict, try to resolve the conflict. If you feel that you have resolved it, use `git rebase --continue` to continue the rebase process. If you can't find a good way to resolve it, use `git rebase --abort` to stop trying to rebase.
- If `topic/add-feature` already has an upstream branch, you will need to use the command `git push --force-with-lease` the next time you push changes to `topic/add-feature`.

Once the rebase is complete, the branches will look like this:



The benefit of rebasing is that it is practically invisible. It appears as if `topic/add-feature` was directly branched from the most recent commit in `dev` and doesn't create noisy merge commits.

Making issues

If you notice any problem or have any idea for a change you think would be good, make an issue. Don't hesitate to do this *even if you're unsure of whether it's a real problem or that we would ever address it*. When making an issue, please make sure to do the following:

- Give it a title that clearly describes what the problem is. The exact formatting of this isn't that important, as long as it's clear.
- Use the description to give any additional context or to mention any ideas you already have for solving the problem.
- Add any relevant labels—this makes it much easier to quickly look through the issue list for certain types of issues.
- Add the issue to the correct section of the project board. This can either be done through the “Project” section on the issue page or by directly creating the issue using the “Add item” button on the project board. If the issue is something that you're sure is relevant to the current sprint, add it to the “Todo” column; otherwise, add it to the “Backlog” column.

If you notice that an issue made by someone else doesn't have labels or isn't on the project board, you are welcome to add those yourself.

Assigning yourself work

If you have a particular thing you want to work on, first look in the “In Progress” column of the project board to make sure that nobody else is

already working on it. If not, look in the “Todo” column and see if someone has already created an issue for it. Otherwise, you may create the issue yourself. Please follow the guidelines in the “Making issues” section.

Once you have the issue, add yourself as an assignee. If you're planning on making changes that will solve multiple issues at once, then assign yourself to *all* such issues. If you plan to fix only *part* of an issue, then in addition to assigning yourself to the issue, also make a comment on the issue page explaining what part you're doing. You may assign yourself to an issue that already has assignees as long as your work doesn't overlap with other peoples' work and you make it clear what you're working on.

Topic branches

Once you have assigned yourself work, create a topic branch. If your work corresponds with exactly one issue, then you can create the topic branch using the “Create a branch” button on the issue page, or you can make the branch by hand. Otherwise, you must create the branch by hand. Either way, please follow the guidelines for topic branches as specified in the “Branch types” section.

While working on your topic branch, please consider regularly incorporating changes from `dev`. This reduces the likelihood of running into merge conflicts and will prevent you from fixing stuff that has already been fixed.

Making pull requests

When you feel that your topic branch is done and have pushed all of your changes upstream, create a pull request from the topic branch into `dev`.

A good rule of thumb for the title and description is to write them as if they were the summary and body of a commit message. I.e., the title should be in the imperative mood and explain the overall changes made by the pull request, and the description should give further explanation if necessary.

For each issue that is *completely* resolved by the pull request, add a line in the description in the form `Resolves #<n>`, where `<n>` is the ID of

the issue. For example, if your pull request resolves issue #123, then the description should include `Resolves #123`. Since `dev` isn't the default branch, these won't be resolved automatically when the pull request is merged, but it *does* make it much easier to see what issues should be closed when the pull request is merged.

Add at least one person as a reviewer for your pull request. If you have anything you're unsure of or you want reviewers to test something specific, then include those details in the pull request description or in a comment.

Reviewing and merging pull requests

If you can, try to regularly look through and review existing pull requests. If you're fairly confident that a pull request is correct, then leave an approval on it. If you're confident that something is incorrect, leave a review requesting changes with a comment explaining what you think needs to be changed. If you want to reference specific lines in your review, then you can click on the "+" button on the line in the diff viewer and drag to select more lines.

Before it can be merged, a pull request must be approved by at least one person and must not have any requested changes. Once those requirements are met, anybody can merge the pull request. Note that this means you can merge a pull request immediately after approving it.

After successfully merging a pull request, delete the topic branch using the "Delete branch" button on the pull request page.