Backport guidelines

Context

In theory, Tiki master/trunk should be kept stable enough for dogfooding. In practice, that would be risky. And thus, many community members prefer to use stable versions, and backport fixes.

Backports take time, and can introduce regressions, which eats us time. Maybe that time is better invested in fixing more bugs in master?

And the more the stable branch and master diverge, the more difficult and risky it becomes. On the other hand, the sooner fixes and enhancements are in a stable branch, the sooner they are available to end users.

So what is the right balance? We have some high-level guidelines at Where to commit. This page will help with more explanations.

The Tiki26 release cycle is exceptional in that

1. We increased the requirements from PHP 7.4 to 8.1
2. We started Using GlitchTip as part of the Tiki development process
3. We added PHPStan and Rector
4. In May 2023 (the month before branching 26x), we broke the record for the number of monthly commits.

As of 2023-06-06

- We need to branch soon
- In the coming months, we'll have hundreds of commits from the reports from GlitchTip and PHPStan.

Types of commits

Changes to the database structure

In general, these should not be backported. We've have too many problems in the past. Please see: Database Schema Upgrade

Fatal error

Should be backported

Bug

Major: Can be backported.
Minor: Do you really need to backport? The next version will soon be upon us.

New feature

If self-contained, low risk and important to a specific project (client, dogfood for tiki.org, etc.), can be backported (when not in a Freeze and Slush period)

Warnings

With default error reporting settings, warnings are generally only visible to admins. Fix these in the stable branch if you are nearby - not essential for the `.0` release, but nice to have *one day*.

Deprecation warnings

- What do we do about a deprecation warning to PHP 8.2? (We are running PHP 8.1 for many sites, but we are hoping it works OK in PHP 8.2)
  - Nice to have *one day*
- What do we do about a deprecation warning but that we don't know in which PHP version it will stop working?
  - Act casual, hope no one notices

Merging and cherry-picking

- I propose that we allow [FIX] *(etc)* commits directly into 26.x and then use `git merge` back into trunk/master to make sure all changes are contained in future versions. Cherry-picking from trunk to 26.x should still be ok and shouldn't create conflicts if done cleanly.
  - **@Jonny Bradley**: Would you envisage something like the script we had for Semi-automatic merging period or each committer is responsible to track?
    - I am worried about omissions and merge conflicts. To keep things simpler, and reduce odds of merge conflicts maybe everyone could make an effort to focus on 26x and any 27x destined work could stay in GitLab draft merge requests until 26.0 is released? And thus, most/all merge conflict resolution would fall on 27x, where we have plenty of time.
- This is slightly different to the past couple of releases *(always commit in trunk first then cherry-pick)* and a little more like how we did it in `svn` days, and should allow for easier and quicker testing of fixes on real/staging sites running on 26.x before 26.0 release.
  - **@Jonny Bradley**: For Pre-dogfood servers for Tiki 26 release process, we'll have a fairly easy way to test, but I agree that for the community at large, it's easier to just focus on 26.
- I'd rather stay with commit-to-master and then backport to previous stable/supported branches back to whatever is needed. Otherwise, we risk loosing updates in upcoming versions if someone forgets to cherry-pick to master.
  - Fair enough, but wouldn't doing `git merge` periodically from 26.x to master catch anything that was missed?
- I feel (strongly) the we should keep cherry-picking

1. On the other hand, I don't care which direction we cherry pick to. It's more logical to do user bug fixes on 26.x since one can relatively use the same database to test in master, while the reverse is not true. As long as it's a rule that you are not done until you merge a change back into master, or wrote in the commit message that it's unnecessary or unwanted in master.
2. I do not see how merging back could work, especially since the master branch is in the middle of refactoring. Say someone fixes a bug in master (commit A). Then cherry picks in 26.x to backport, but have to do changes (say a variable or function name) because master diverged (this isn't a merge conflict). He does commit B in 26.x (say "Use old variable name for X). He is done, everything is fine. The another developper comes along and fixes an unrelated bug directly in 26.x (commit C). He then merges in master. Commit B gets pulled along, and master is now broken, in a completely unerelated place.
3. We use a linear history process for master (we use rebase on merge requests by default). I do not know of any way git can keep track that commit B above is still commit B when "merging" into master. Cherry-pick doesn't have that problem as far as I know.
4. Halting merging merge request into master means (in practice) halting almost all fundamental work in master (everything that affects a lot of files), as if the changes are not merged regularly, not only will there be more and bigger conflicts once they do merge, but there is a lot more chance of the merge being incorrect. For example (not that it is especially likely example, just an easy one to understand), if master renames a global variable, and a merge request newly uses a global variable (and that is frequent, prefs...), the code won't work once merged, even if there is no merge conflict.
   - 2023-06-12: Victor, Jonny, Roberto and Marc discussed: We can try this commit-to-26x process for this version, and evaluate after. (TODO update Where To Commit)

Related

- Git cherry-pick
- Where To Commit