Commit queue

> *Stability: 1 - Experimental*

tl;dr: You can land pull requests by adding the `commit-queue` label to it.

Commit Queue is an experimental feature for the project which simplifies the landing process by automating it via GitHub Actions. With it, collaborators can land pull requests by adding the `commit-queue` label to a PR. All checks will run via node-core-utils, and if the pull request is ready to land, the Action will rebase it and push to master.

This document gives an overview of how the Commit Queue works, as well as implementation details, reasoning for design choices, and current limitations.

Overview

From a high-level, the Commit Queue works as follow:

1. Collaborators will add `commit-queue` label to pull requests ready to land
2. Every five minutes the queue will do the following for each pull request with the label:
    1. Check if the PR also has a `request-ci` label (if it has, skip this PR since it's pending a CI run)
    2. Check if the last Jenkins CI is finished running (if it is not, skip this PR)
    3. Remove the `commit-queue` label
    4. Run `git node land <pr> --oneCommitMax`
    5. If it fails:
        1. Abort `git node land` session
        2. Add `commit-queue-failed` label to the PR
        3. Leave a comment on the PR with the output from `git node land`
        4. Skip next steps, go to next PR in the queue
    6. If it succeeds:
        1. Push the changes to nodejs/node
        2. Leave a comment on the PR with `Landed in ...`
        3. Close the PR
        4. Go to next PR in the queue

To make the Commit Queue squash all the commits of a pull request into the first one, add the `commit-queue-squash` label. To make the Commit Queue land a pull request containing several commits, add the `commit-queue-rebase` label. When using this option, make sure that all commits are self-contained, meaning every commit should pass all tests.

Current limitations

The Commit Queue feature is still in early stages, and as such it might not work for more complex pull requests. These are the currently known limitations of the commit queue:

1. All commits in a pull request must either be following commit message guidelines or be a valid `fixup!` commit that will be correctly handled by the `--autosquash` option
2. A CI must've ran and succeeded since the last change on the PR
3. A collaborator must have approved the PR since the last change
4. Only Jenkins CI and GitHub Actions are checked (V8 CI and CITGM are ignored)

Implementation

The action will run on scheduler events every five minutes. Five minutes is the smallest number accepted by the scheduler. The scheduler is not guaranteed to run every five minutes, it might take longer between runs.

Using the scheduler is preferable over using pull_request_target for two reasons:

1. if two Commit Queue Actions execution overlap, there's a high-risk that the last one to finish will fail because the local branch will be out of sync with the remote after the first Action pushes. `issue_comment` event has the same limitation.
2. `pull_request_target` will only run if the Action exists on the base commit of a pull request, and it will run the Action version present on that commit, meaning we wouldn't be able to use it for already opened PRs without rebasing them first.

`node-core-utils` is configured with a personal token and a Jenkins token from @nodejs-github-bot. `octokit/graphql-action` is used to fetch all pull requests with the `commit-queue` label. The output is a JSON payload, so `jq` is used to turn that into a list of PR ids we can pass as arguments to `commit-queue.sh`.

> *The personal token only needs permission for public repositories and to read profiles, we can use the GITHUB_TOKEN for write operations. Jenkins token is required to check CI status.*

`commit-queue.sh` receives the following positional arguments:

1. The repository owner
2. The repository name
3. The Action GITHUB_TOKEN
4. Every positional argument starting at this one will be a pull request ID of a pull request with commit-queue set.

The script will iterate over the pull requests. `ncu-ci` is used to check if the last CI is still pending, and calls to the GitHub API are used to check if the PR is waiting for CI to start (`request-ci` label). The PR is skipped if CI is pending. No other CI validation is done here since `git node land` will fail if the last CI failed.

The script removes the `commit-queue` label. It then runs `git node land`, forwarding stdout and stderr to a file. If any errors happen, `git node land --abort` is run, and then a `commit-queue-failed` label is added to the PR, as well as a comment with the output of `git node land`.

If no errors happen during `git node land`, the script will use the `GITHUB_TOKEN` to push the changes to `master`, and then will leave a `Landed in ...` comment in the PR, and then will close it. Iteration continues until all PRs have done the steps above.

Reverting broken commits

Reverting broken commits is done manually by collaborators, just like when commits are landed manually via `git node land`. An easy way to revert is a good feature for the project, but is not explicitly required for the Commit Queue to work because the Action lands PRs just like collaborators do today. If once we start using the Commit Queue we notice that the number of required reverts increases drastically, we can pause the queue until a Revert Queue is implemented, but until then we can enable the Commit Queue and then work on a Revert Queue as a follow-up.