Project Forking

Forking a project to your own namespace is useful if you have no write access to the project you want to contribute to. If you do have write access or can request it, we recommend working together in the same repository since it is simpler.

The Git forking workflow is a popular approach used for managing code contributions in distributed version control systems like GitLab. It is particularly effective in environments where multiple contributors work on a project independently, often in open-source settings.

Benefits of the Forking Workflow

- **Decentralization:** Contributors work on their own forks, reducing the risk of affecting the main repository.
- **Collaboration:** Merge requests (also called Pull Requests) facilitate a clear review and collaboration process.
- **Contributor Control:** Each contributor manages their own repository, branches, and workflow, allowing them to work independently.

This workflow is especially common in open-source projects but is also used in private or corporate settings where developers need a structured process for proposing changes.

"Fork" is not a Git operation — it just means you have made a copy of an existing repository and are doing new development on your copy.

Forking Workflow for Contributors

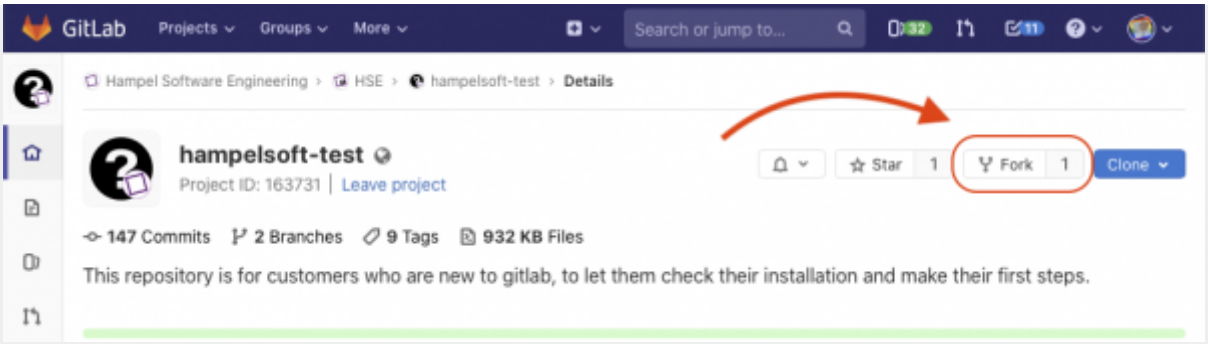
You are the `contributor`. The guy who created the original repo is the `maintainer`.

1. Forking the Repository:

The workflow begins by forking the main (upstream) repository to your own Git account. This creates a personal copy of the repository, which allows you to freely experiment without affecting the main project.

When working with GitLab, forking a project is a two-step process:

1. Click on the fork button between the star and clone buttons on the project's home page.



2. Once you do that, you'll be presented with a screen where you can choose the namespace to fork to. Only namespaces (groups and your own namespace) where you have write access to, will be shown. Click on the namespace to create your fork there.

If you're trying to fork a repository on `code.hampel-soft.com` and receive an error message saying you cannot create projects in your own namespace, please see our GitLab FAQ for more details.

2. Cloning the Fork:

After forking, you clone your repository to your local machine. This gives you a working directory where you can make changes to the codebase.

```
# clone own private repo (this is remote 'origin')
git clone https://user@gitlab.com/user/repo.git
```

You can also connect your local clone of the repo with the original repository of the maintainer by adding a second remote:

```
# add the public repo as a remote called 'upstream'
git remote add upstream https://user@gitlab.com/maintainer/repo.git
```

3. Creating a Feature Branch:

Before making any changes, it's a good practice to create a new branch that isolates your feature or bug fix. This branch is separate from the main branch of both your fork and the upstream repository.

```
git checkout -b feature-branch
```

4. Making Changes and Committing:

If you added the maintainer's repo as a second remote, you can check for changes and bring them into your local clone.

```
# keep my private repo up to date with changes from public repo
git pull upstream master
```

You can now make the necessary changes to the code, add them to the staging area, and commit them to your feature branch.

```
# Stage changes
git add .
# Edit some code
git commit -a -m "this is my change"
```

5. Pushing Changes to Your Fork:

Once you've committed your changes, you push the branch to your remote fork on GitHub (or another Git hosting service).

```
# publish my changes to my private repo
git push origin feature-branch
```

6. Creating a Merge Request:

With the changes pushed to your fork, the next step is to create a merge request (MR) to the original repository. The MR allows the maintainers of the upstream repository to review your changes and decide whether to merge them into the main codebase.

7. Review, Feedback, and Merging:

The repository maintainers will review your MR, possibly provide feedback, and request changes. After addressing any feedback, they can merge the changes into the upstream repository.

```
# update my private repo from the public repo after merging my changes
git pull upstream master
```

Forking Workflow for Maintainers

You are the `maintainer`. The guy who created the MR is the `contributor`.

When reviewing and merging a merge request (MR) in a Git forking workflow, maintainers follow several important steps to ensure that the proposed changes are well-integrated into the main repository.

1. Review the Merge Request

1.1 Merge Request Access via GitHub/Bitbucket UI

When a contributor submits a merge request (MR), their code changes are automatically made accessible to the maintainer within the platform's interface (GitHub, GitLab, Bitbucket, etc.). The maintainer can see:

- A diff view that highlights the changes made in the code (line-by-line comparisons).
- A list of commits made on the contributor's feature branch.
- The full branch history showing what has changed, when, and why.

From here, the maintainer can review all files and see a summary of added, modified, or deleted code.

1.2 Locally Accessing Contributor’s Code

If the maintainer prefers to examine the code changes in a local environment, they can pull the contributor’s branch from the contributor’s forked repository and test it locally. Here’s how:

i. Add the Contributor’s Fork as a Remote: First, the maintainer needs to add the contributor's fork as a remote in their local repository. They use the git remote add command to point to the fork’s URL.

```
git remote add contributor https://gitlab.com/user/repo.git
```

ii. Fetch the Branch: Once the contributor’s repository is added, the maintainer can fetch the branch associated with the merge request:

```
git fetch contributor feature-branch
```

iii. Checkout the Branch Locally: The maintainer can then check out the branch locally to explore and test the changes:

```
git checkout feature-branch
```

This allows the maintainer to run tests, inspect the code, or even make changes locally.

1.3 Merging Changes Temporarily (Without Official Merge)

To test how the contributor’s changes integrate with the latest version of the upstream (main) repository, the maintainer can perform a local merge without pushing it to the remote. This helps to identify merge conflicts or issues before the official merge.

i. Fetch the Latest Upstream Changes:

```
git fetch origin
```

ii. Merge Contributor’s Branch:

```
git merge contributor/feature-branch
```

If there are merge conflicts or issues, the maintainer can resolve them locally, test the code, and provide feedback to the contributor.

1.4 Accessing Code via a Fork on the Platform

On platforms like GitHub, GitLab, or Bitbucket, the maintainer can also directly access and browse the code through the contributor’s fork via the web interface. They can:

- Browse the files in the contributor’s fork repository.
- Check different branches of the fork.
- Look at the commit history to see how the feature evolved.

This method allows maintainers to quickly assess the contributor’s overall code structure, branch management, and history.

2. Providing Feedback

- Request Changes: If there are issues, maintainers will provide detailed feedback through the MR comments. This may involve requesting changes related to functionality, optimization, or code readability.
- Open Discussion: The merge request becomes a forum for discussion between the maintainer and the contributor. Both can exchange ideas, ask for clarifications, or suggest improvements.
- Iterative Improvement: The contributor then updates their code based on the feedback, and pushes the changes to the same branch. GitLab automatically updates the MR with these new commits.

3. Approving the Merge Request

- Final Check: After the contributor has made the necessary adjustments, the maintainer performs a final review. If everything is satisfactory, the merge request is approved.
- Automated Checks: Many teams use automated workflows that perform final checks, such as verifying if the branch can be merged without conflicts, or if additional tests are required before merging.

4. Merging the Merge Request

- Choose a Merge Strategy:
    - Merge Commit: This is the default behavior, where all commits from the branch are combined into a single merge commit in the main branch. This preserves the full history of changes.
    - Squash and Merge: This option squashes all the commits into a single commit before merging, making the history cleaner, especially when a feature contains many small or insignificant commits.

    - Rebase and Merge: This strategy replays the changes from the branch on top of the main branch. It results in a linear history but can make resolving conflicts harder.
2. Conflict Resolution: If there are merge conflicts, the maintainer must resolve them before merging. GitLab provides an in-browser tool for this, or it can be done locally.
3. Closing the Merge Request: Once the MR is successfully merged, it is closed. Some maintainers may also delete the contributor's feature branch if it is no longer needed.

5. Post-Merge Actions

- Follow-Up Testing: Even after merging, some projects may have additional automated tests or manual quality assurance steps.
- Communication: Maintainers typically leave a comment acknowledging the contributor's efforts and may encourage them to continue contributing to the project.
- Documentation and Changelog: If the feature is significant, maintainers often update documentation or project release notes to reflect the new changes.

Tools and Best Practices

- Automated Tools: Continuous Integration (CI) systems (like GitLab CI) are commonly used to automate testing and validation during MR reviews. These tools automatically check if the code passes tests and adheres to code quality metrics.
- Branch Protection Rules: Maintainers can set up branch protection rules to ensure that the main branch cannot be modified without passing all tests and reviews.
- Contributor Guidelines: It’s common for open-source projects to include contributing guidelines (e.g., CONTRIBUTING.md), which describe the expectations and process for submitting a merge request.

By following these steps, maintainers ensure that the contribution process is smooth and that the codebase remains stable and maintainable over time.

Other Resources

- https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow
- https://reflectoring.io/github-fork-and-pull/
- https://blog.seibert-media.net/blog/2014/04/24/git-workflows-der-forking-workflow-teil-1/
- https://blog.seibert-media.net/blog/2014/04/25/git-workflows-der-forking-workflow-teil-2/