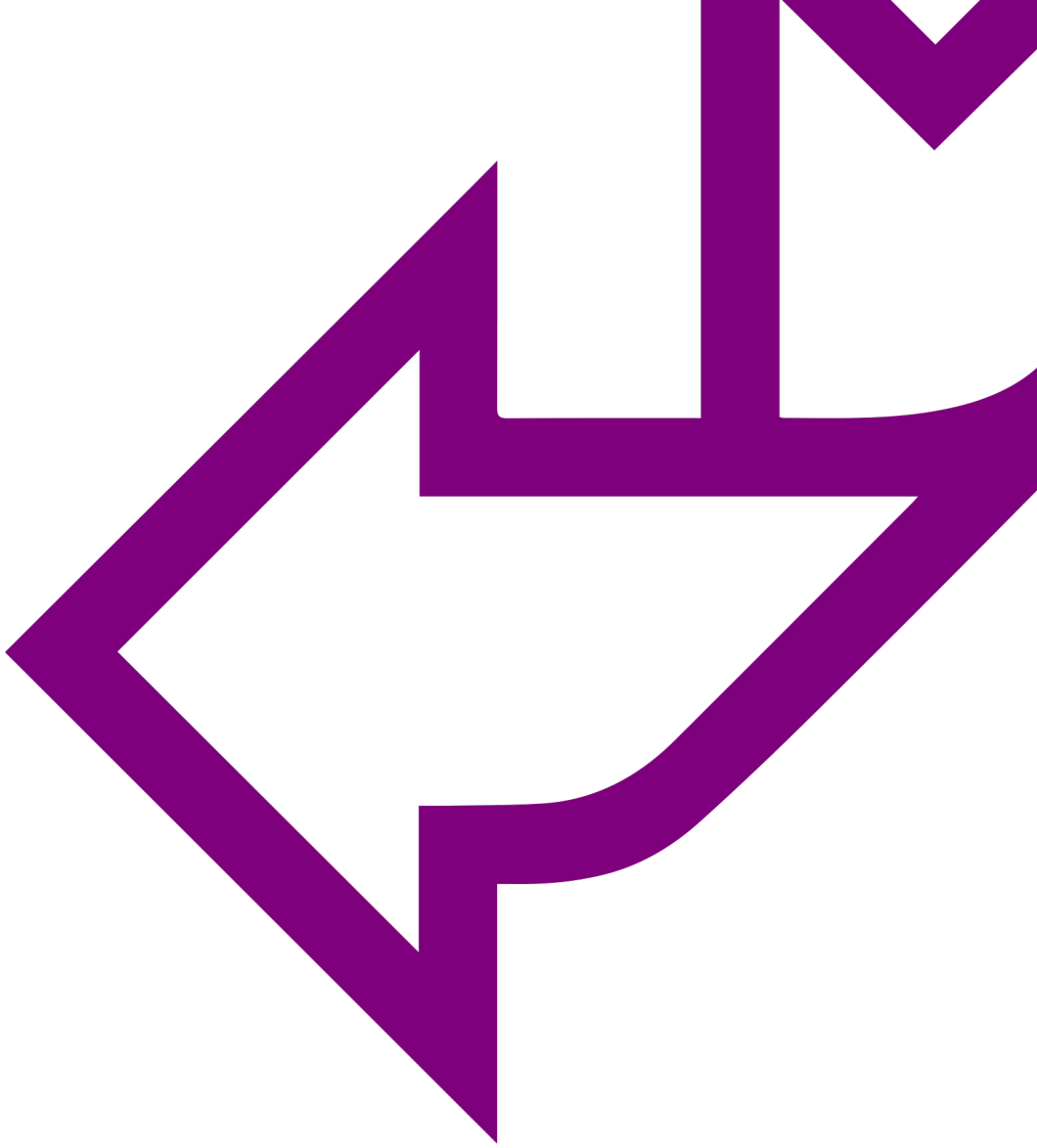
POWERING POTENTIAL



Merging and Reverting

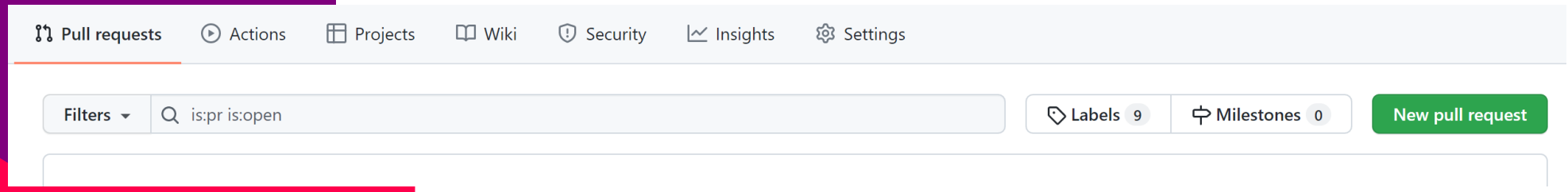
Module 5 - Version Control with GIT

PULL REQUESTS

Pull Requests (PRs) are used to change, review, and merge code in a Git repository.

You can create PRs from branches in the upstream repository or from branches in a forked repository.

Team members can review the PRs and provide feedback on the changes.



MERGING

Joining the history of two or more branches through **git merge** incorporates changes into the current branch. This command is used by **git pull** to incorporate changes from a different repository, as well as to merge changes from one branch into another.

If we assume that we have a history like this, and the current branch is **main**:

```
      E---F---G issue
     /
A---B---C---D main
```

Executing **git merge *issue*** would add changes E, F and G into **main** and result in a new commit.

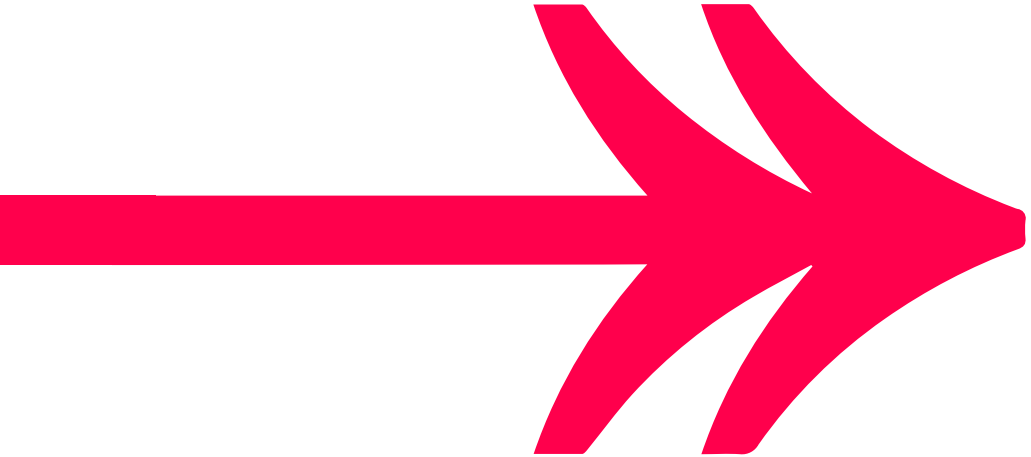
This new commit would result in the following history:

```
      E---F---G issue
     /         \
A---B---C---D---H main
```

Exercise 4: Branching and Merging

- Create a pull request
- Merge develop into main
- See the changes on the main branch

Merge conflicts



Merge conflicts happen when more than one person has edited a file, and the line numbers that were edited are the same. It can also happen if someone deleted a file that another person was working on.

This conflict only affects the person performing the merge, and the rest of the team won't be affected by it.

If a merge conflict happens, *Git* will automatically halt the merge process and mark the file, or files, that are conflicting. It is then up to the developer to resolve the conflicts by editing the files and then committing the edited versions to the repo.

Ignoring Files

We often do not want to track certain files in our repositories. This could be because they are large or unnecessary (e.g. compiled code, build output directories or dependency caches). In order to make sure that Git does not track them, we need to create a **.gitignore** file in our repository.

This file is a list of intentionally untracked files that Git should ignore.

.gitignore

```
1   ####################
2   # compiled source #
3   ####################
4   *.com
5   *.class
6   *.dll
7   *.exe
8   *.pdb
9   *.dll.config
10  *.cache
11  *.suo
12
13  # Directories #
14  ################
15  bin/
16  obj/
```

TAGGING

Tags are refs that point to specific points in Git history.

Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0)

```
git tag v1.0
```

This creates a lightweight tag

```
git tag -a v1.5 -m "Version 1.5"
```

This creates an annotated tag which includes more metadata such as the tagger's name and a tagging message.

To list tags:

```
git tag
```

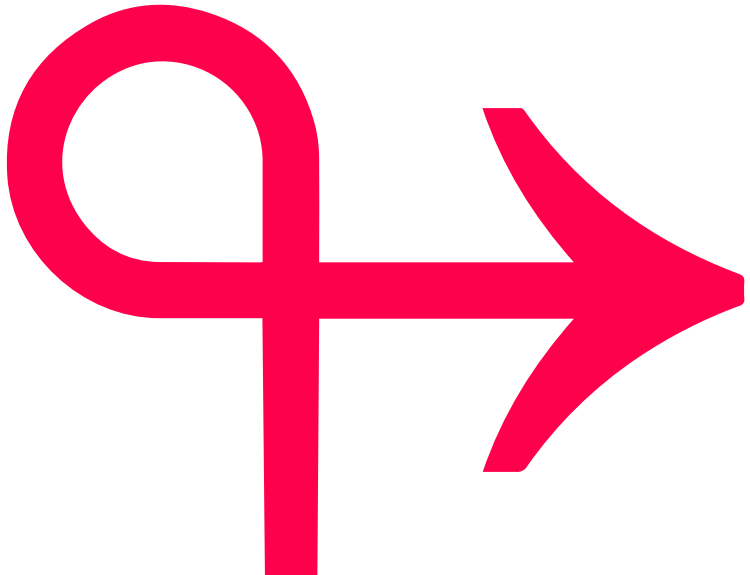
FORKING

In layperson's terms, forking is creating a copy of an existing repository under your account.

Forking a repository allows you to freely experiment with the existing code base, without the fear of breaking the project.

Forking also gives you the ability to contribute to a project, by adding additional functionality.

Scenarios for Forking



Imagine you are using someone's project and you find a bug. Usually, you would raise an issue for this; however, forking means you would be able to attempt a fix yourself.

The steps would be:

1. Fork the repository
2. Create a new branch
3. Fix the issue
4. Submit a pull request to the owner of the original project
5. If the owner approves your fix, your work should then be merged into the original repository.

Your idea

If a project is under public license that allows you to freely use it, you could use the project as a starting point for your own project. For example, if you found a web application that you liked, and it was under the public license, you could fork the project and add any additional functionality. This would then be yours to freely use or distribute.

EXERCISE 5: MERGE CONFLICTS

- Create a new repo
- Add a file to the main/master branch and edit the file
- Make a new branch and edit the same file
- Attempt to merge the branches
- Resolve the merge conflict

Reviewing the history of a repository

The best way to view commit history is by using the following command:

```
git log
```

This command will show you the history of all commits for the current branch, with the output including: date, commit message and the SHA-1 identifying hash.

There are also additional flags that make this command easier to use, such as:

```
git log --oneline
```

Using the --oneline flag simplifies the output into one line per commit.

```
git log --branches=*
```

By using the --branches=* flag, we are making Git return the history of all commits for all the branches in that repository.

To get the data for a specific branch, the log command would need to be used like this:

```
# git log [BRANCH_NAME]
git log main
```

Reverting to a previous commit with revert

Let's assume that your git log --oneline looks similar to this:

```
 875f31e (HEAD -> main) fourth commit
 483856a third commit
 2dd011d second commit
 bcabb84 first
```

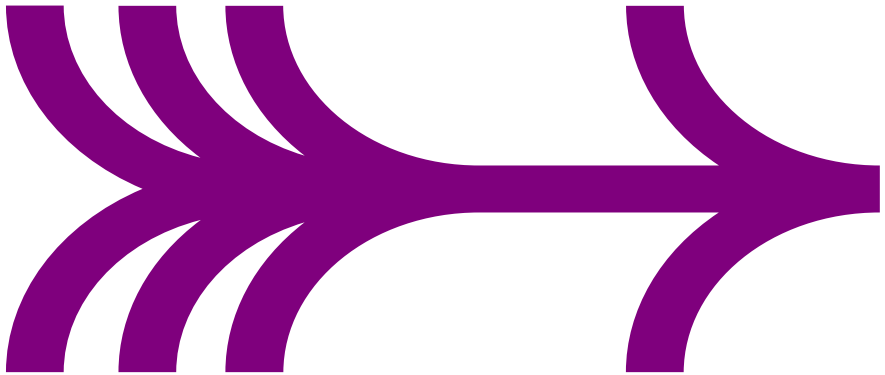
If we execute:

```
git revert HEAD
```

Git will create a new commit, which will do the opposite of the previous commit (for example, if you added a piece of code you didn't need, the revert would create a commit deleting this piece of code). You can also use revert to go back to a specific SHA-1 (483856a for example).

The history will still contain the fourth commit, but it's changes will be undone. Using revert allows us to use the same branch and is considered a better solution for reverting than git reset.

Reverting with reset



Instead of using git revert in this situation above, we could have used:

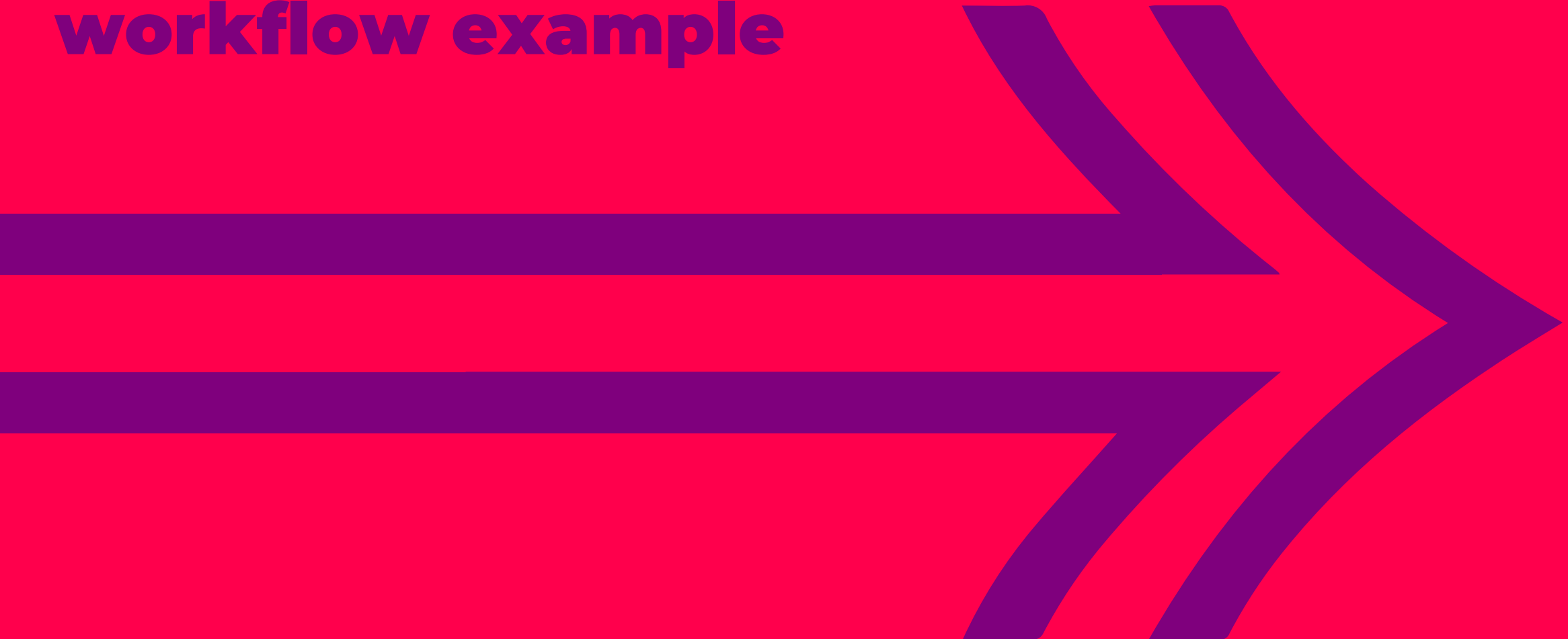
```
git reset --hard 483856a
```

This command will return the state to the selected commit (483856a third commit).

The difference between this and **git revert** is that now the Git history will no longer contain the fourth commit, and work would continue as if the fourth commit never happened.

However, not having the commit reflected in the commit history can cause complications when working with a shared remote repository; if the reset happened to a commit that is already shared with others, and we tried to push some changes afterwards, Git would throw an error; this is because it would think that our local Git history isn't up to date. In these scenarios, it's more appropriate to use the revert strategy.

Git Branching workflow example



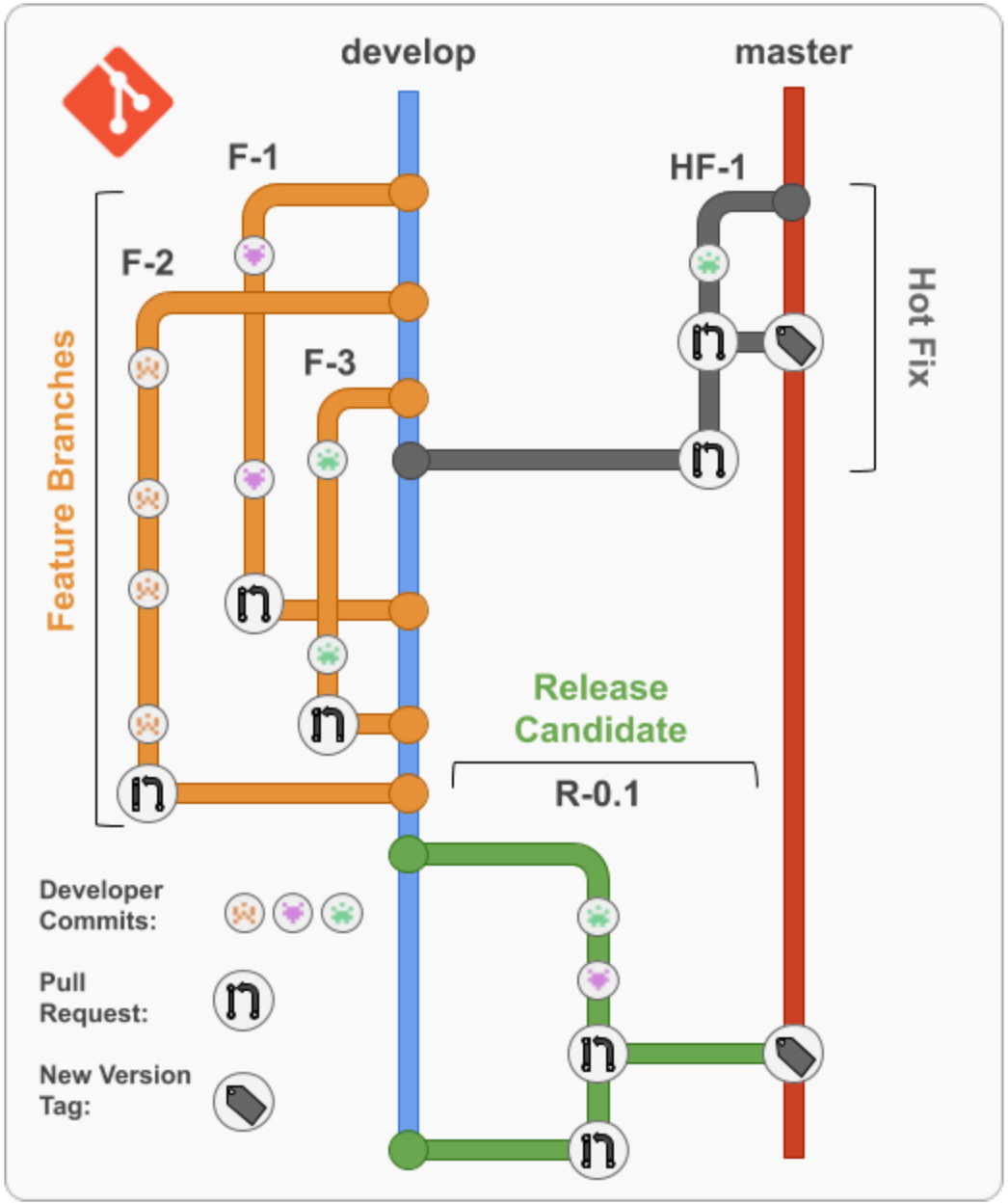
GIT BRANCHING WORKFLOW EXAMPLE

This is an example of a workflow using Git.

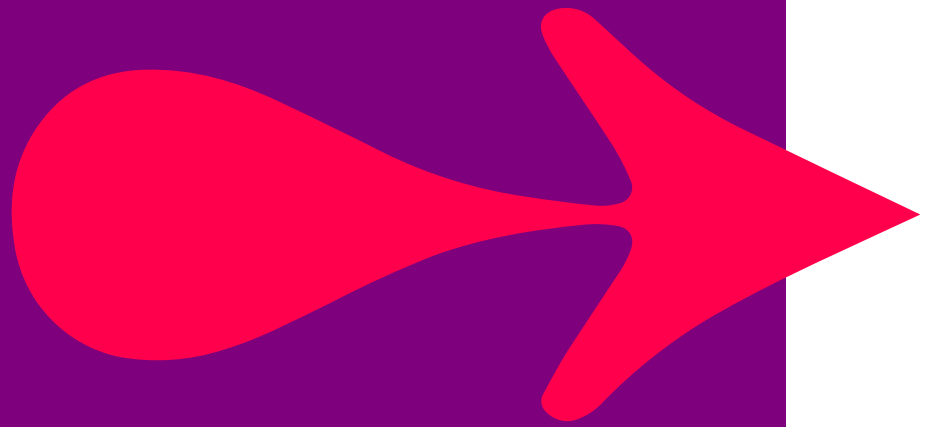
The two main branches that exist are the develop and main branches.

New Application Features
Conventionally, new features are to be created in feature branches from the develop branch, and, once the feature has been developed, the code can be reviewed by a peer in a Pull Request and deployed to a test environment for Integration or User Acceptance Testing.

If all testing and reviews pass, the Pull Request can be approved and merged into the develop branch.


develop
master
F-1
F-2
F-3
HF-1
Feature Branches
Hot Fix
Release Candidate
R-0.1
Developer Commits:
Pull Request:
New Version Tag:

RELEASES



When there is a release approaching, a release candidate branch can be made.

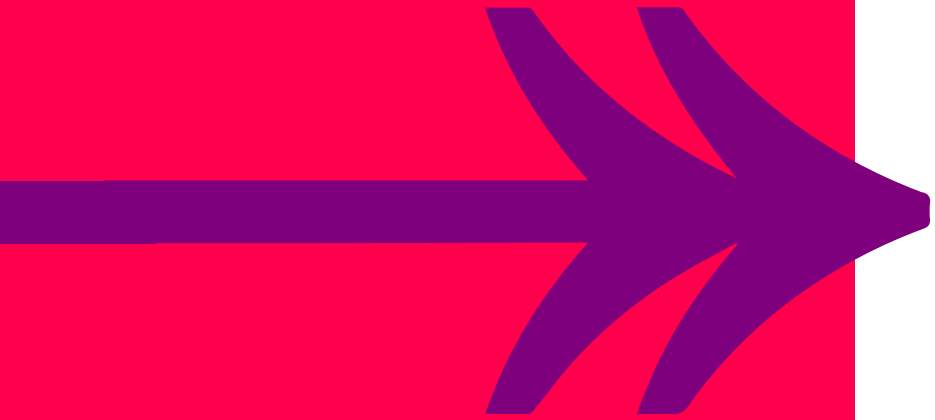
On this new branch, further testing of all the new features working together can take place, and more commits can be made on this branch to amend any issues.

Once testing has passed and the release has been signed off by the individual accountable for releases, the release candidate branch can be merged into **master/main** for the release to be deployed to a production environment.

Once merged into the **master/main** branch, the code can be *tagged* or marked as a release on the Git service that you are using.

Changes must also be merged back into the develop branch, so that any changes that were made to the release candidate will also be included in future releases.

HOTFIXES



Preventing hotfixes is one of the main reasons for designing a workflow such as this.

Hotfixes should be prevented where possible, but they are sometimes necessary.

A hotfix can be conducted by creating a hotfix branch from the **master/main** branch and applying the changes on that branch; before merging back into the **master/main** branch all the changes should, of course, be tested and reviewed to avoid even more hotfixes!

Once merged into the **master/main** branch, the changes must also be merged back into the develop branch; this will keep any important changes the hotfix made. The code in the **master/main** branch should be *tagged*.

Exercise 6: Reverting in Git

- Initialise a new git repository
- Create some new files
- Stage and commit the files
- Revert one of the commits
- Check the logs

END OF SESSION