KTH Royal Institute of Technology

SEMINAR 2 - 29 March 2017

Simone Stefani - sstefani@kth.se

WHAT IS THIS SEMINAR ABOUT

Branching

Merging and rebasing

Git team workflows

Pull requests and forks

WHAT IS THIS SEMINAR NOT ABOUT

How to setup Git on your computer

Git/GitHub Clients with GUI

Comparison of workflows

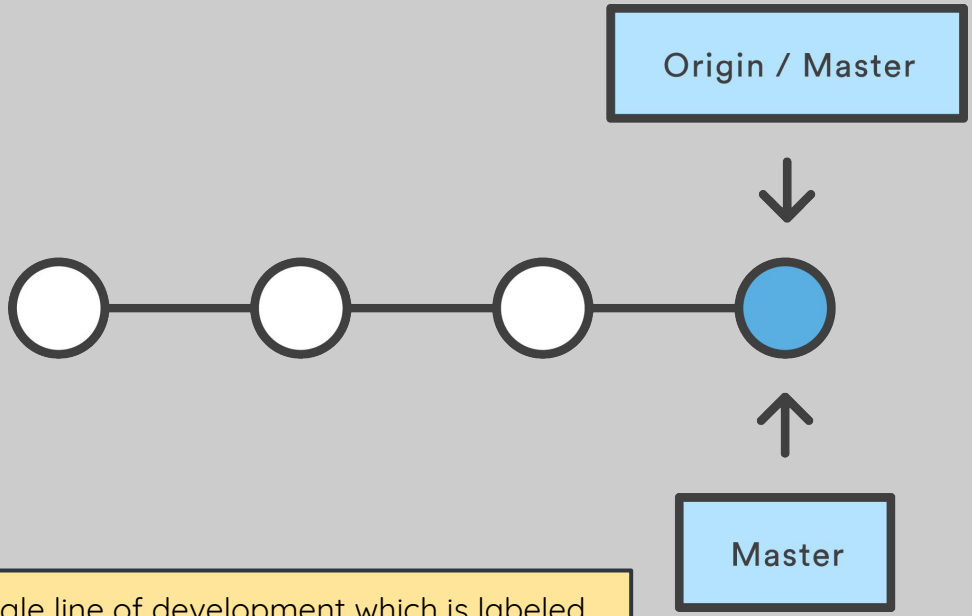
The most important lesson:

If you don’t know which is the best next move ask your peers for help.

You’ll learn more and have less problems

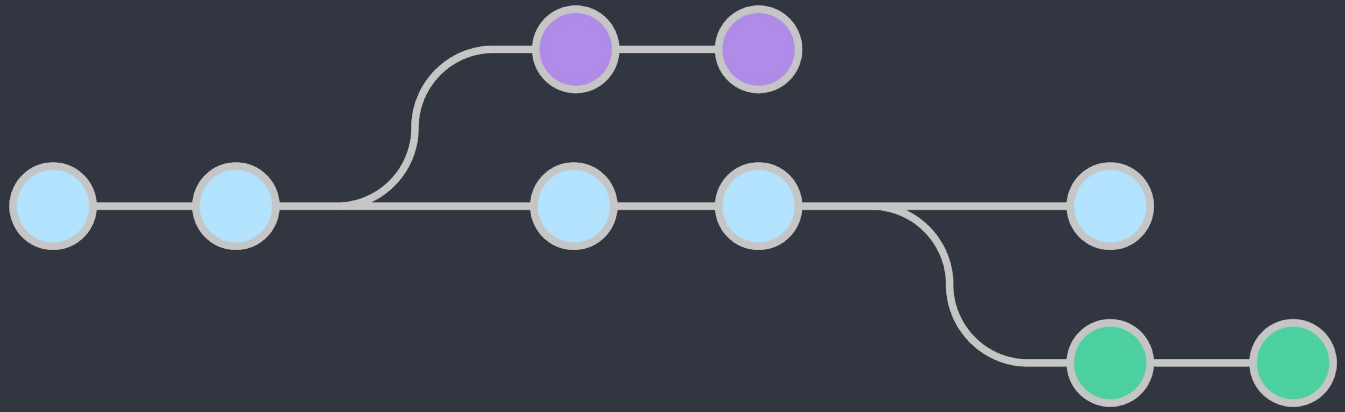
PART 3
The Branches

A quick reminder

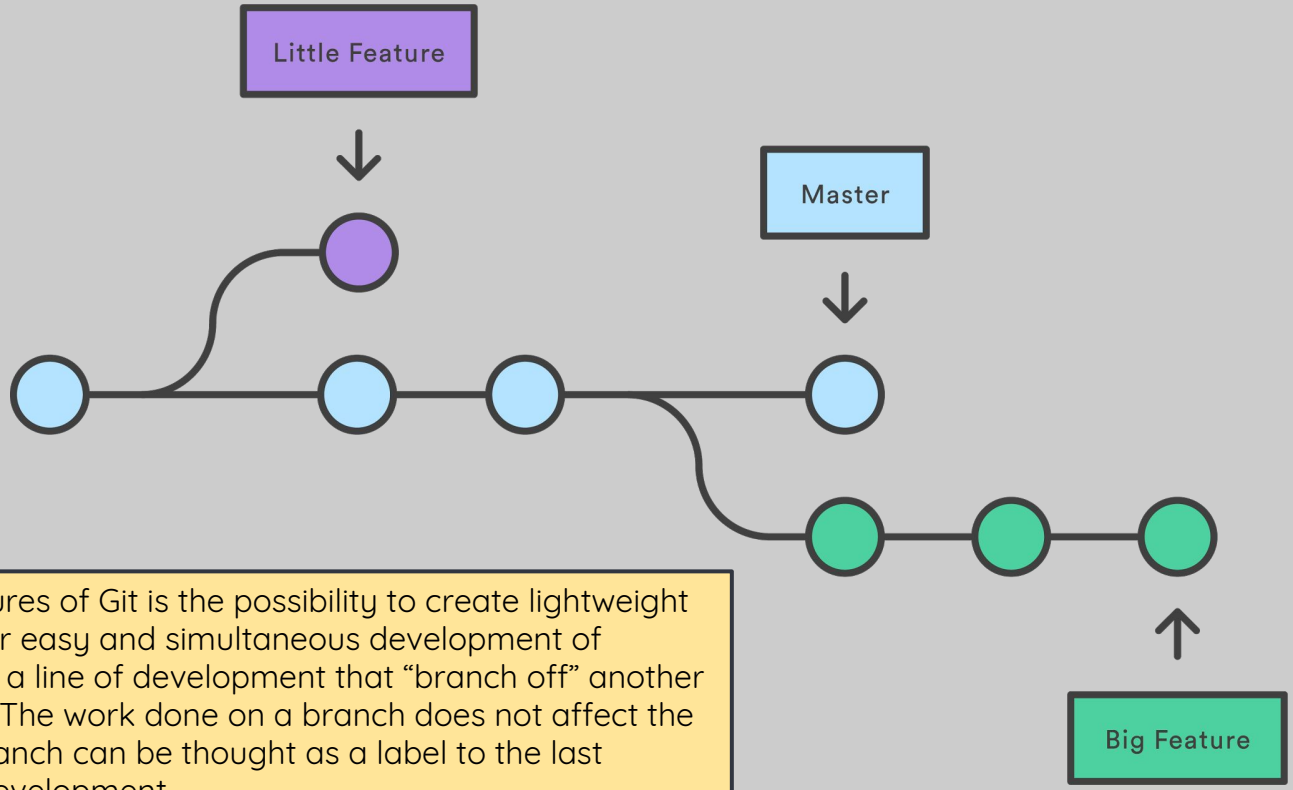


We left PART 1 & 2 with single line of development which is labeled `master`. This represents the story of the project as a chain of snapshots (commits). It also may have been pushed to a remote hosting service as a remote named `origin`.

The best idea



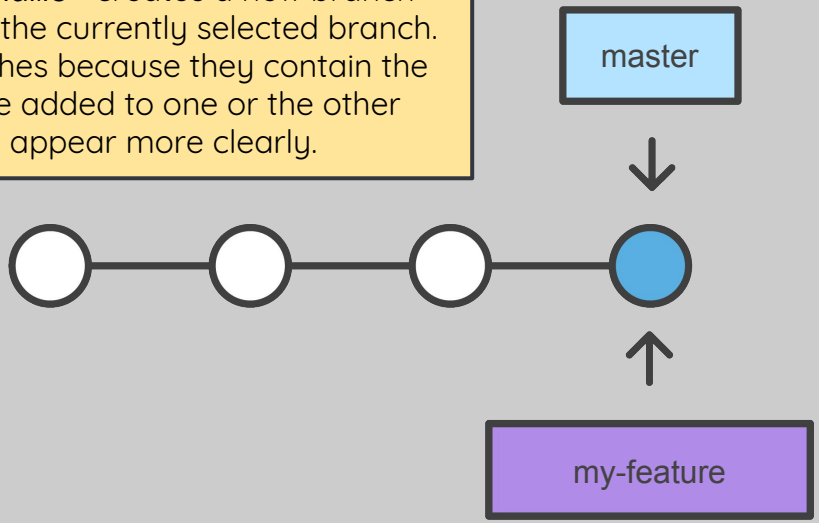
Many lines of development



One of the best features of Git is the possibility to create lightweight branches allowing for easy and simultaneous development of features. A **branch** is a line of development that “branch off” another line of development. The work done on a branch does not affect the other branches. A branch can be thought as a label to the last commit in a line of development.

Create a branch

The command `git branch <branch-name>` creates a new branch with the given name that detaches from the currently selected branch. Initially is hard to visualise the two branches because they contain the same commits. As soon new commits are added to one or the other branch, the two lines of development will appear more clearly.



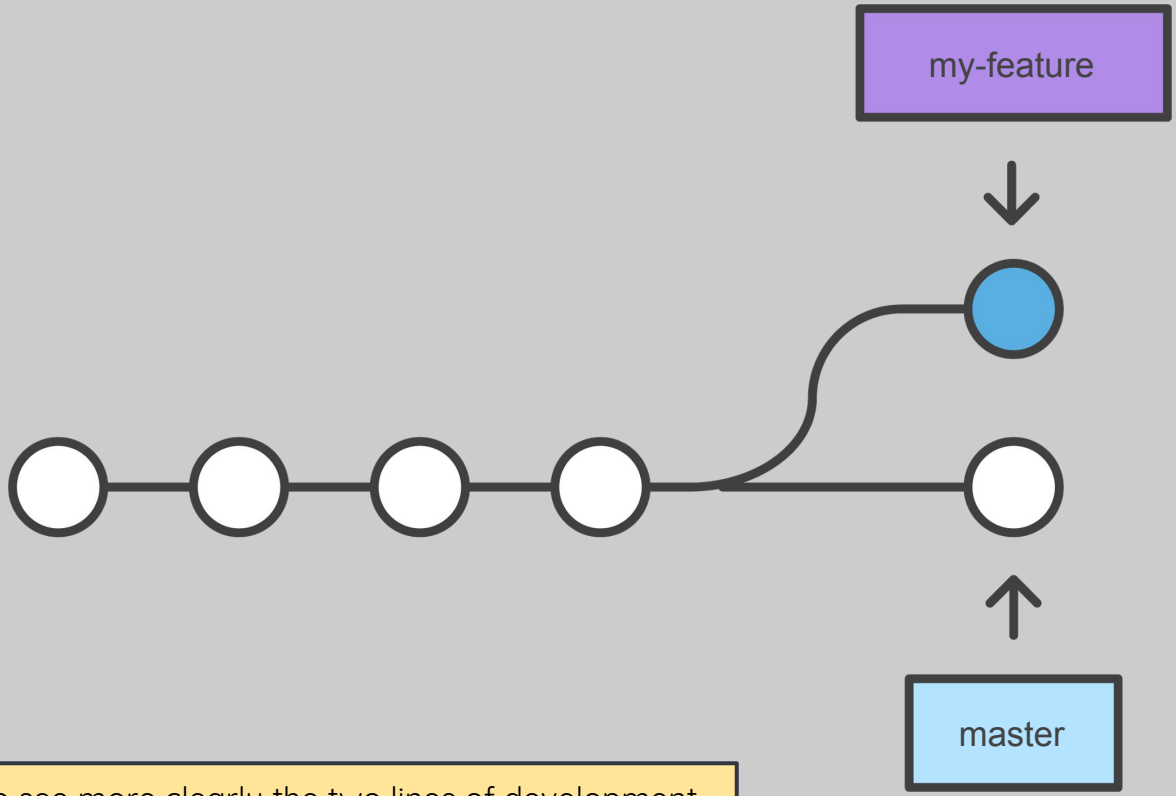

```
$ git branch my-feature
```

Make a commit on `master`

Switch to `feature-branch`

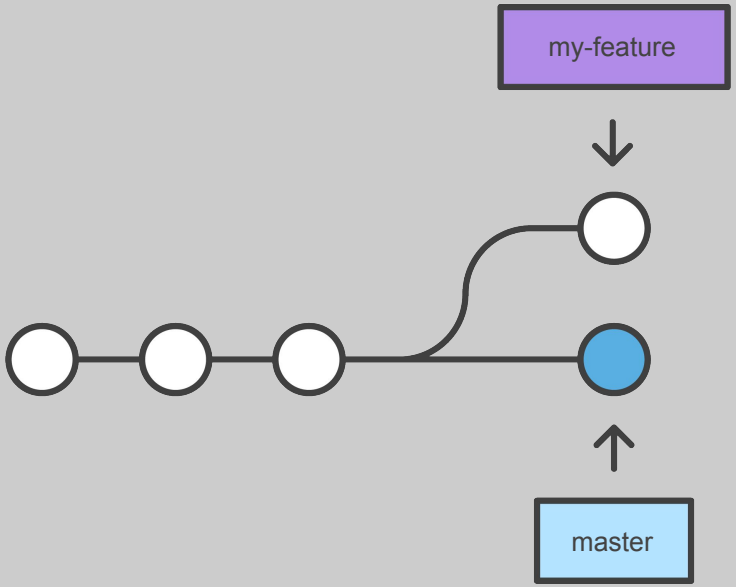
Make a commit on `feature-branch`

Create two different commits, one on each branch.



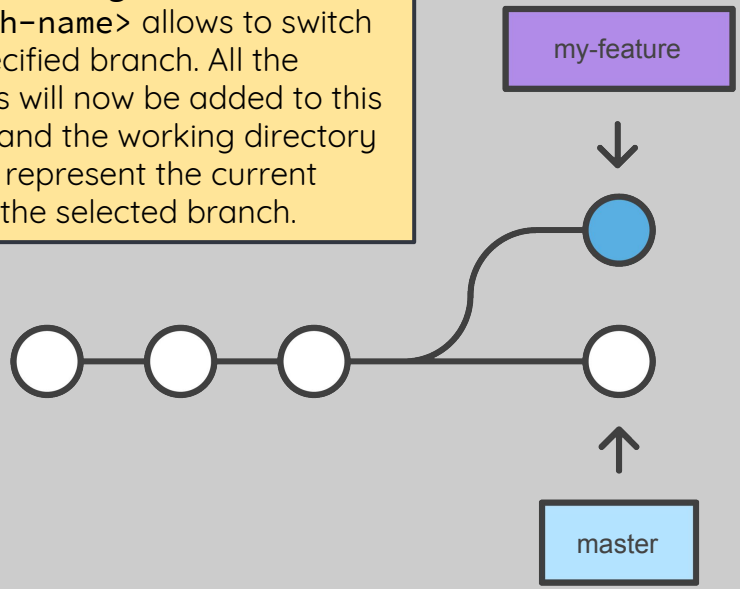
Now it is possible to see more clearly the two lines of development represented by the branches `master` and `my-feature`.
The blue circle represents the selected branch.

Checking Out Master



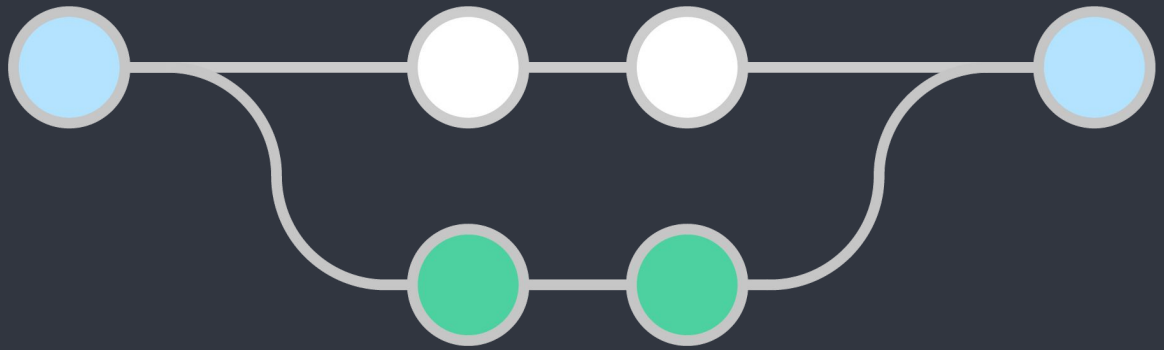
Checking Out Some Feature

The command `git checkout <branch-name>` allows to switch to a specified branch. All the commits will now be added to this branch and the working directory is set to represent the current state in the selected branch.




```
$ git checkout my-feature
```

Merging and rebasing



Merge a specific branch into the current branch

```
$ git merge <branch>
```

The command `git merge <branch-name>` allows to integrate the commits on the specified branch into the current branch. This may happen, for example, when a feature is completed on a branch and needs to be integrated in the main project.

It comes in two flavours:

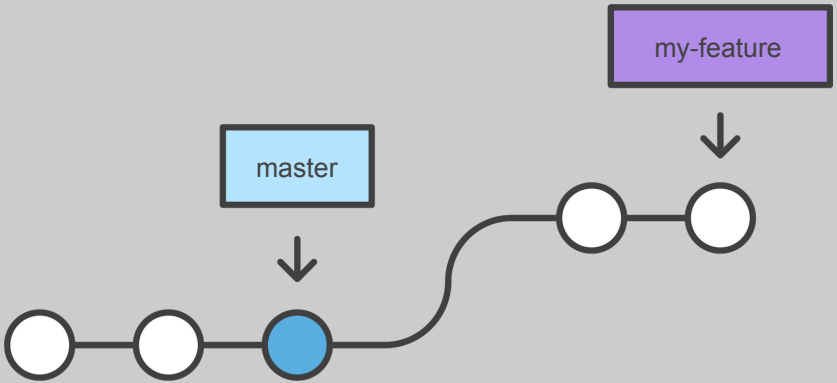
1. Fast-forward merge

2. Three-way merge

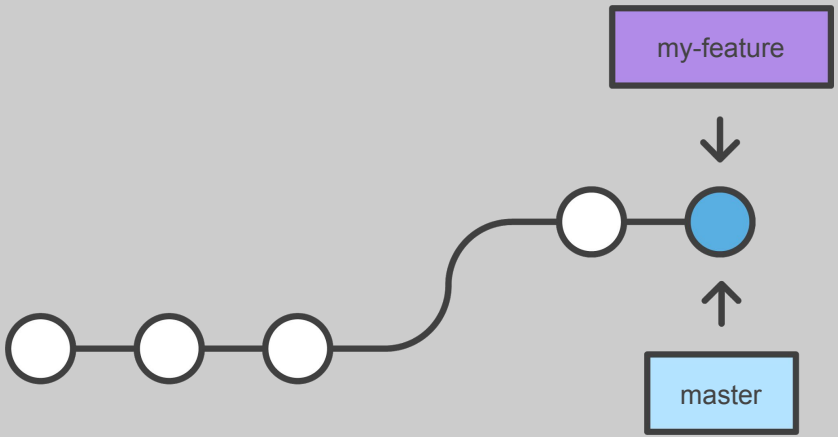
At this point is important to consider how the merge of commits may happen. Git will choose between two strategies: fast-forward and three-way merge.

Fast-forward merge

Before Merging

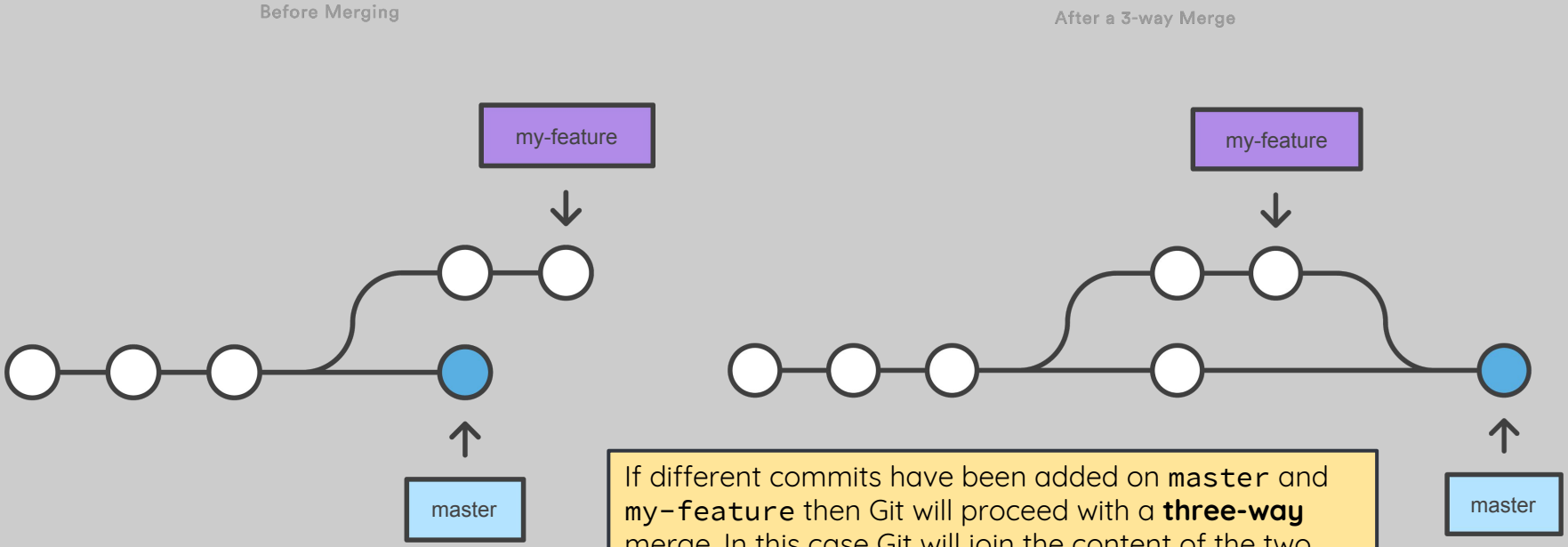


After a Fast-Forward Merge



Let's consider the case when `my-feature` needs to be merged into `master`. A **fast-forward** merge will happen when commits have been added on `my-feature` but not on `master`. In this case the merge consist in chaining all the new commits from `my-feature` to the last commit of `master`. Now the two branches look like the same and contain the same commits.

Three-way merge



If different commits have been added on `master` and `my-feature` then Git will proceed with a **three-way** merge. In this case Git will join the content of the two branches and make a new commit containing all the changes introduced in the two lines of development.

A conflictual story

Merge conflicts may happen in case of a three-way merge

It may happen that two different commits on different branches change the same file in the same place. In this case Git is not able to tell which version should be kept and it throws a **merge CONFLICT**. The developer needs to manually inspect the file and decide how the final version should look like. Then commit to complete the merge.

```
$ git merge my-feature

Auto-merging hobbit.txt
CONFLICT (content): merge conflict in hobbit.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merge conflicts resolution is an art

Conflicts will happen:
get used to them!

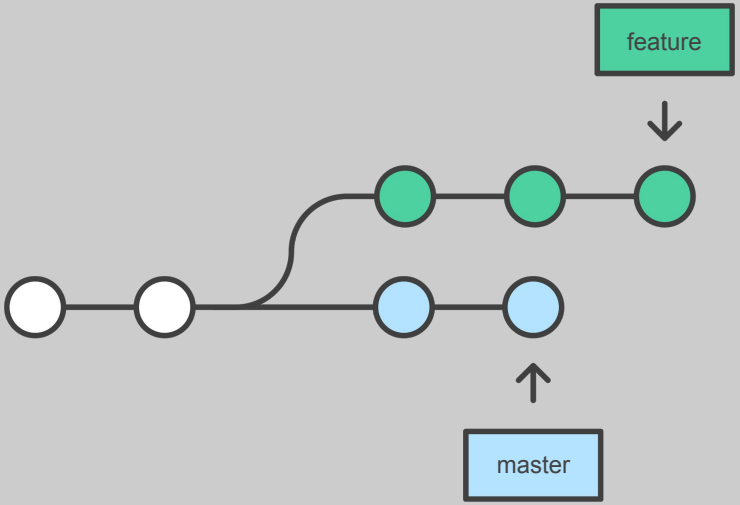
There are many resources to learn how to resolve merge conflicts but the best way it's probably doing it once with someone more experienced. This is one of the situations where it's clearly important to reach for help.

A strategy to avoid conflicts:

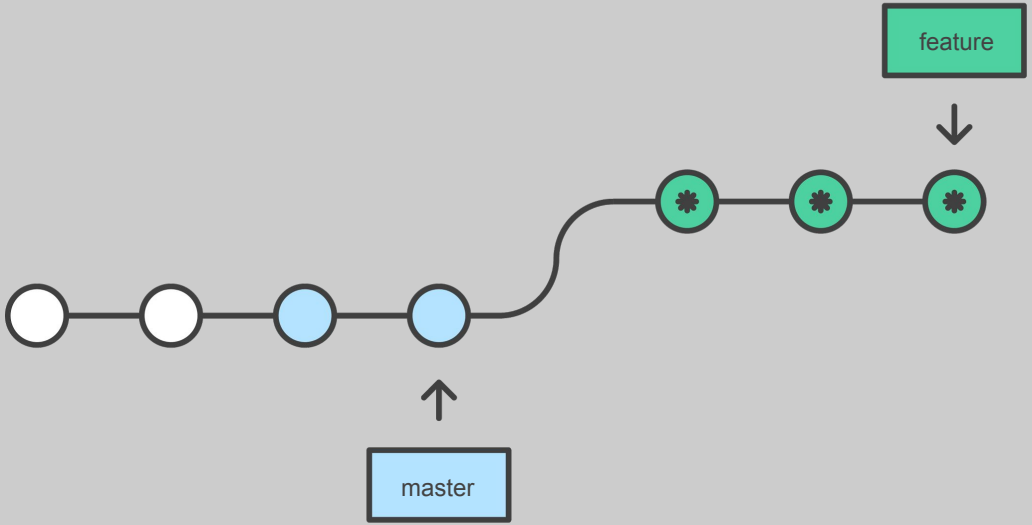
```
$ git checkout feature
  $ git rebase master
```

The command `git rebase <target-branch>` allows to copy the commits of the current branch and link them to the last commit of the target-branch. In this way if a merge is executed right after the rebase, Git will use the fast-forward strategy.

A forked commit history



Rebasing the feature branch onto master



Brand New Commit

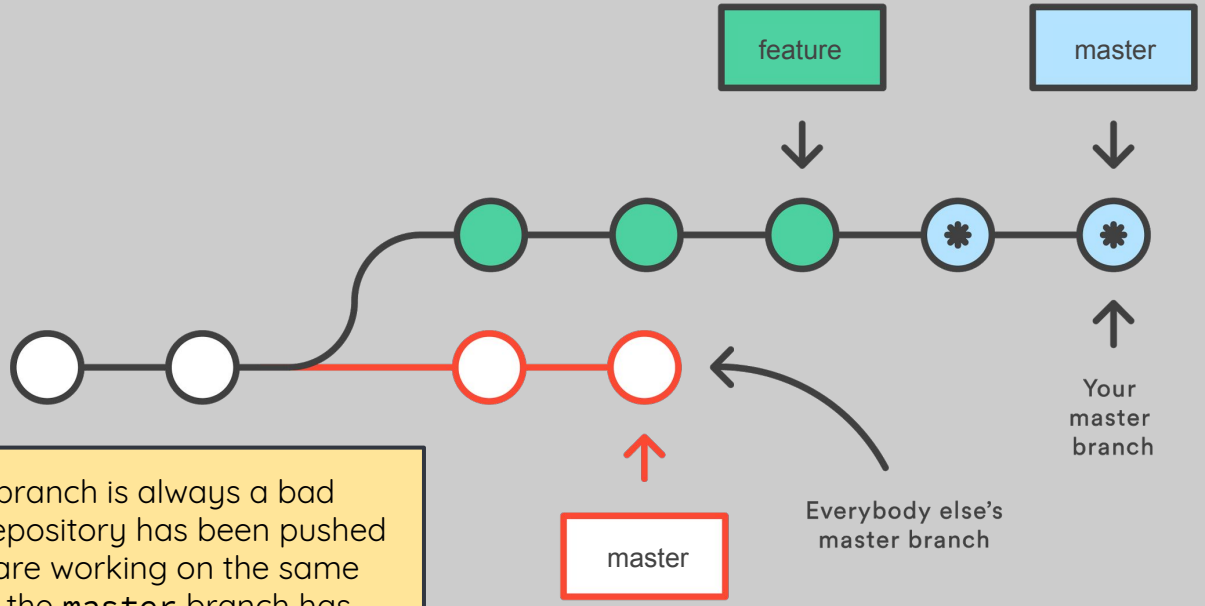
After the executing `git rebase master`, the `feature` branch is “mounted” on top of the last commit in the `master` branch. Observe that the commits in the `feature` branch after the rebase are actually copies of the original ones. Now executing a fast-forward merge results to be a trivial operation.

The golden rule of rebasing:

Never use `rebase` on public branches

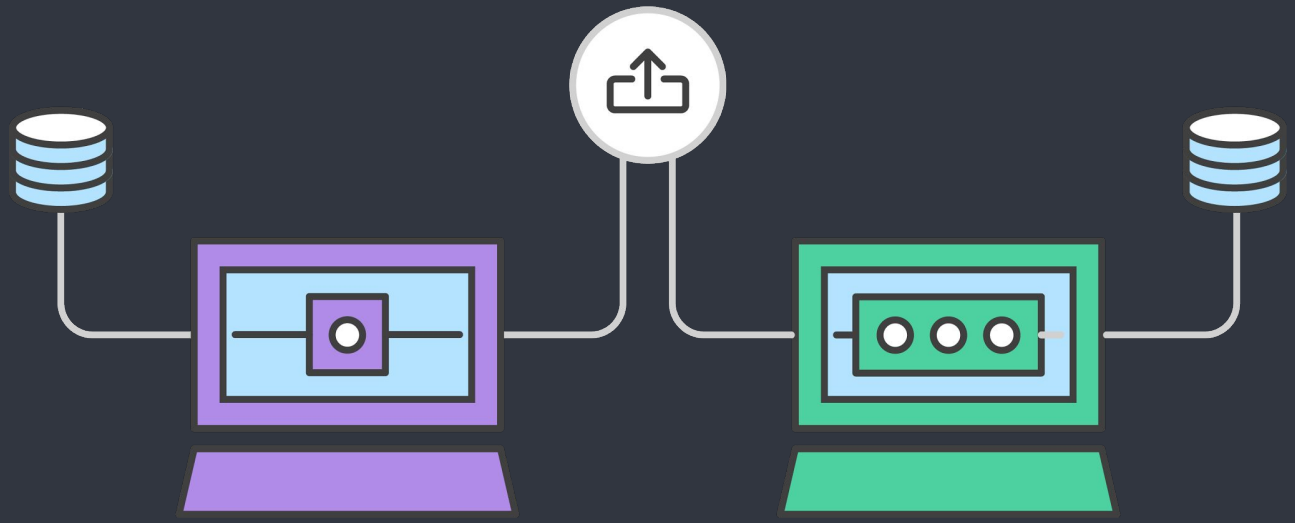
Rebase should only be used with branches in the local repository. It should never be used with public branches because it would result in re-written history and would create problems to other developers.

Rebasing the master branch



Rebasing the `master` branch is always a bad idea especially if the repository has been pushed and other developers are working on the same "history". In the picture the `master` branch has been rebased on top of `feature` but all the other developers are building their work on top of it. This will generate a large number of conflicts and loss of informations.

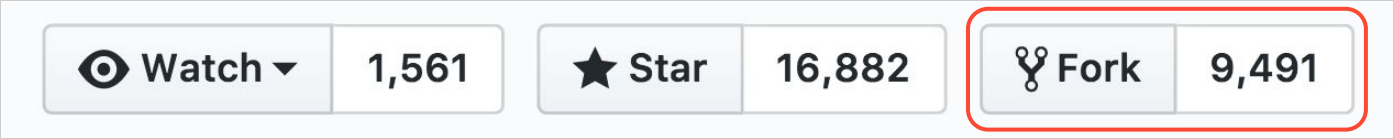
Fork & Pull Requests



Fork a repository

Create a copy of someone else repository linking back to the source

Where do I fork?



If you are on GitHub visiting someone else repository and you press the button **Fork**, then GitHub will make a copy of that repository and save it on your account. The two repositories are independent from each other but yours will remember that it was originally copied from the other.

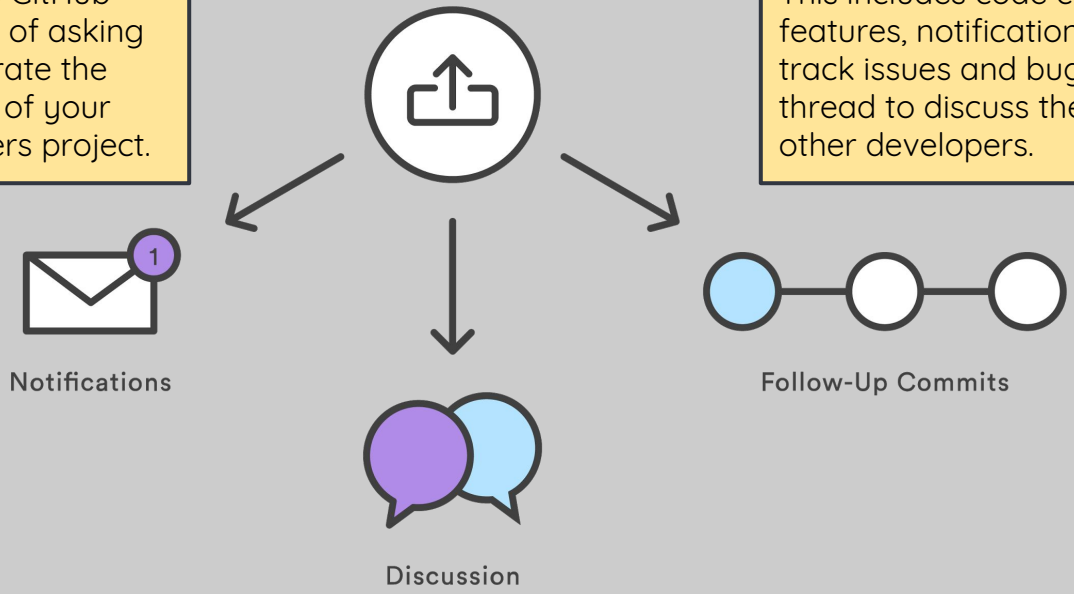
From GitHub!

And the repository will be copied to my GitHub account

Pull request

A **pull request** has nothing to do with the `pull` command. It is an operation that can be triggered in the GitHub interface with the final goal of asking another developer to integrate the work from a branch of one of your projects into a branch of hers project.

GitHub will then open an interface to ease the code integration procedure. This includes code comparison features, notifications, a system to track issues and bugs and a comment thread to discuss the operation with other developers.

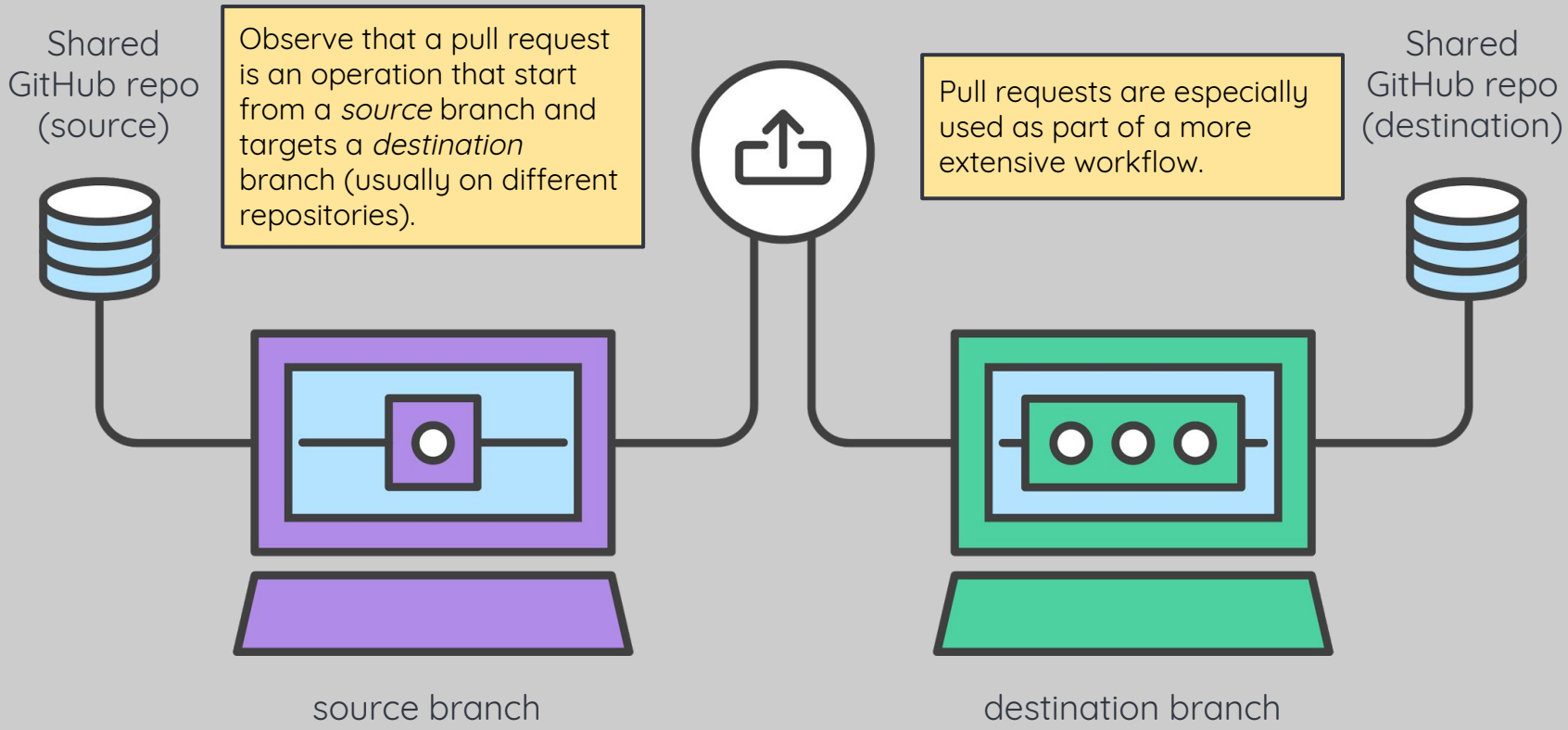


Start a discussion with the team about a feature and notify them about any progress

Pull request: in words

1. A developer creates the feature in a dedicated branch in the local repo
2. The developer pushes the branch to a his public GitHub repository
3. The developer files a pull request via GitHub
4. The rest of the team reviews the code, discusses it, and alters it
5. The project maintainer merges the feature into the official repository and closes the pull request

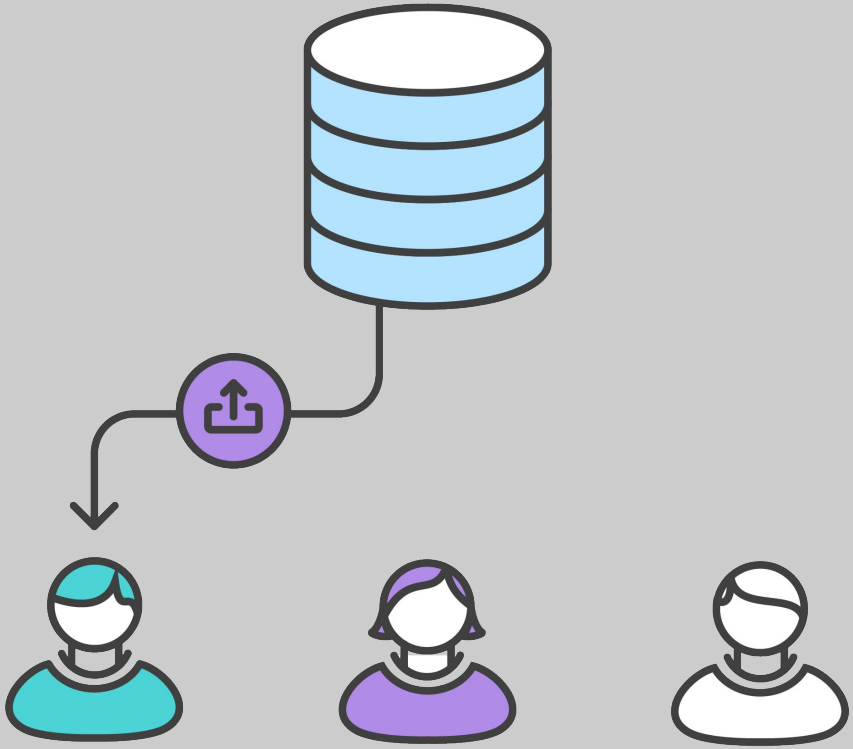
Pull request: in picture



PART 4
Team Workflows

When working in a team it is important to establish a common routine to manage Git actions

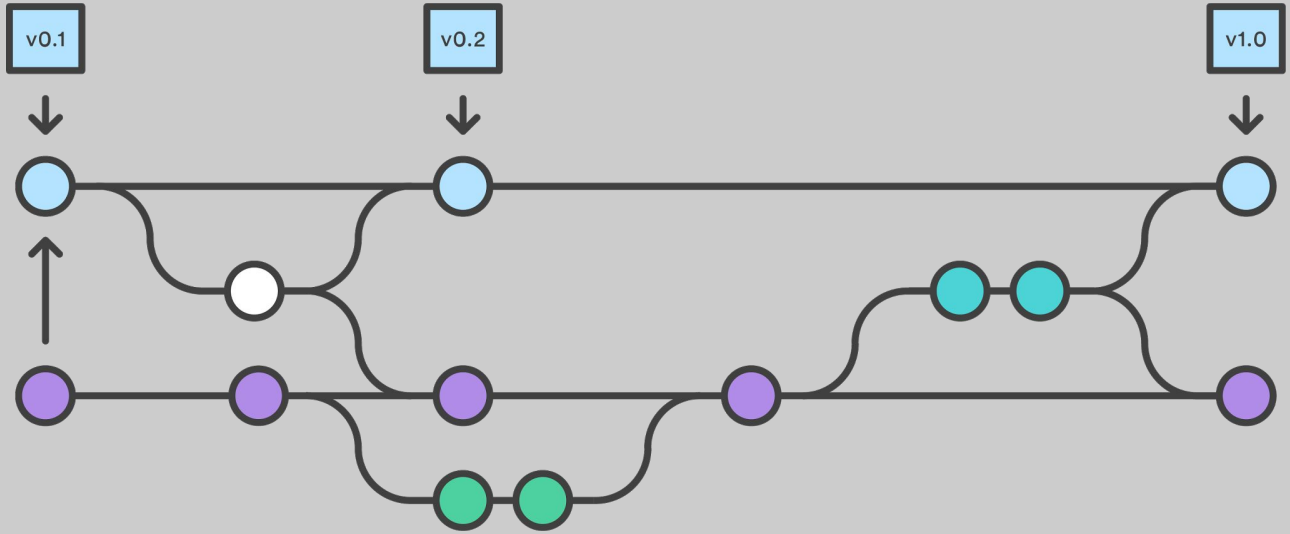
Workflows



There is no such thing as a “right” workflow.

Choose something that fits your team’s needs

Gitflow



Gitflow is a workflow developed by the developer Vincent Driessen as a “successful Git branching model” shaped after several years of working with Git in teams. The original post can be found at: http://nvie.com/posts/a-successful-git-branching-model/
This model relies on heavy use of branches and it is especially useful in small teams.

A) Set of rules to identify and manage branches

B) Set of actions to work in a team on a single project

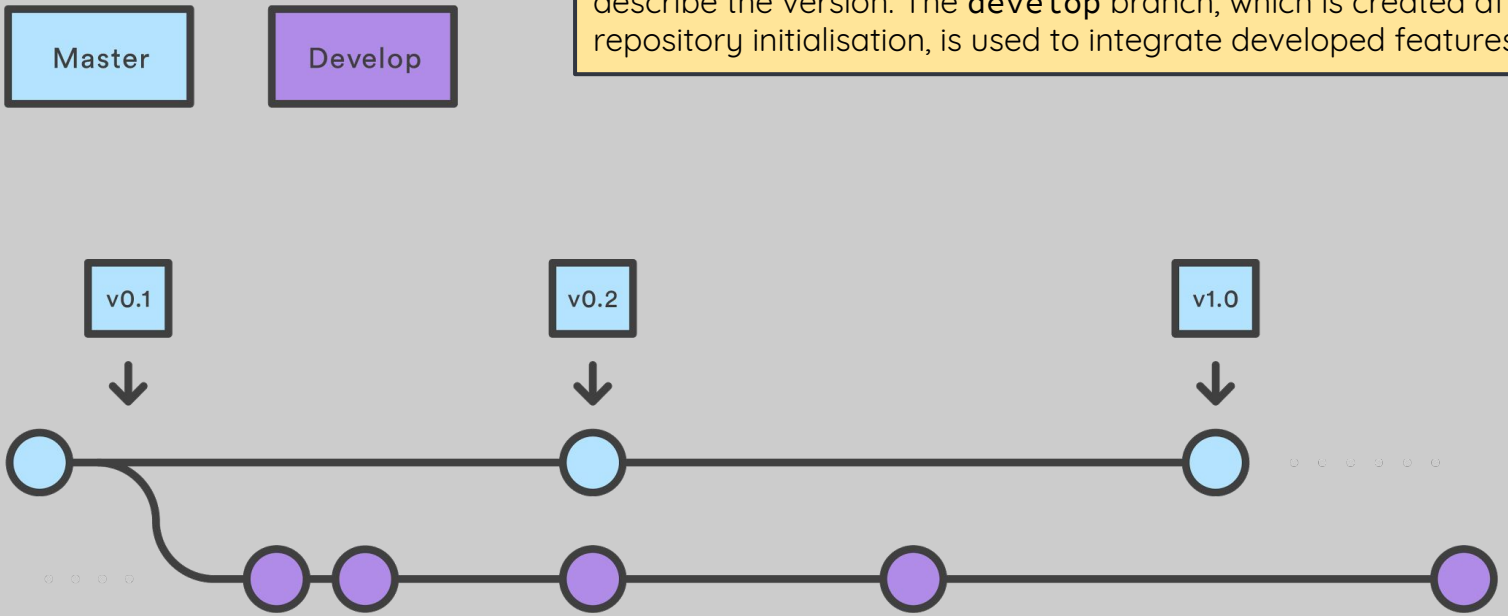
Branches conventions

`master`: should always represent the current state of the project in production

Production is the technical word to describe the “live” version of a project. If the project is a website or web service it can refer to the version running on a live server. If the project is an app it is the released version (possibly published on some online store)

`develop`: originating from the `master` branch, this is the place where development happens

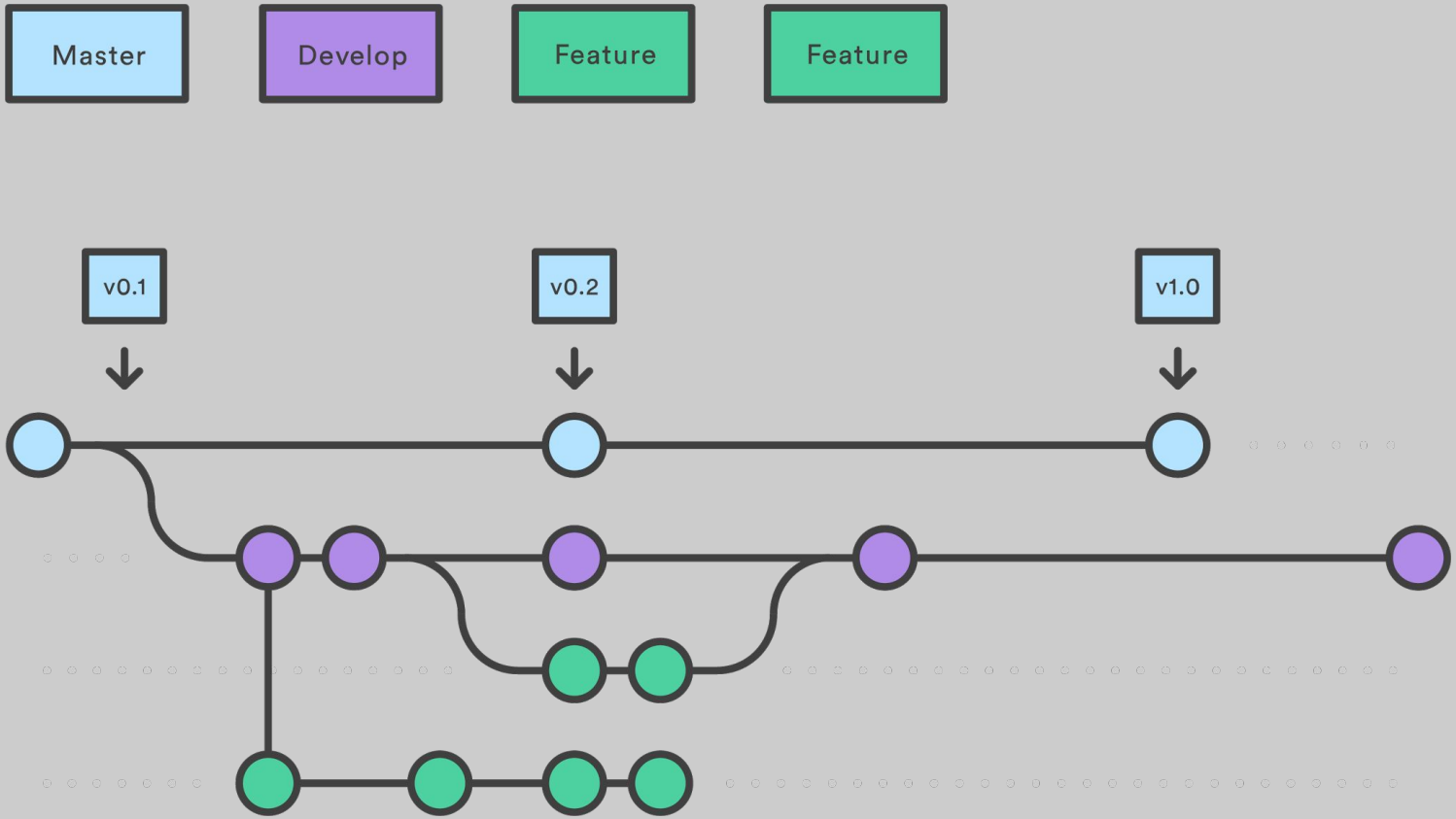
The `master` branch corresponds to the release history, always reflecting the production state. The commits may be have a **tag** do describe the version. The `develop` branch, which is created after the repository initialisation, is used to integrate developed features.



`feature`: used to add new features to the project

- Name convention: `feature/<name>`
- Originates from: **`develop`**
- Merged into: **`develop`**

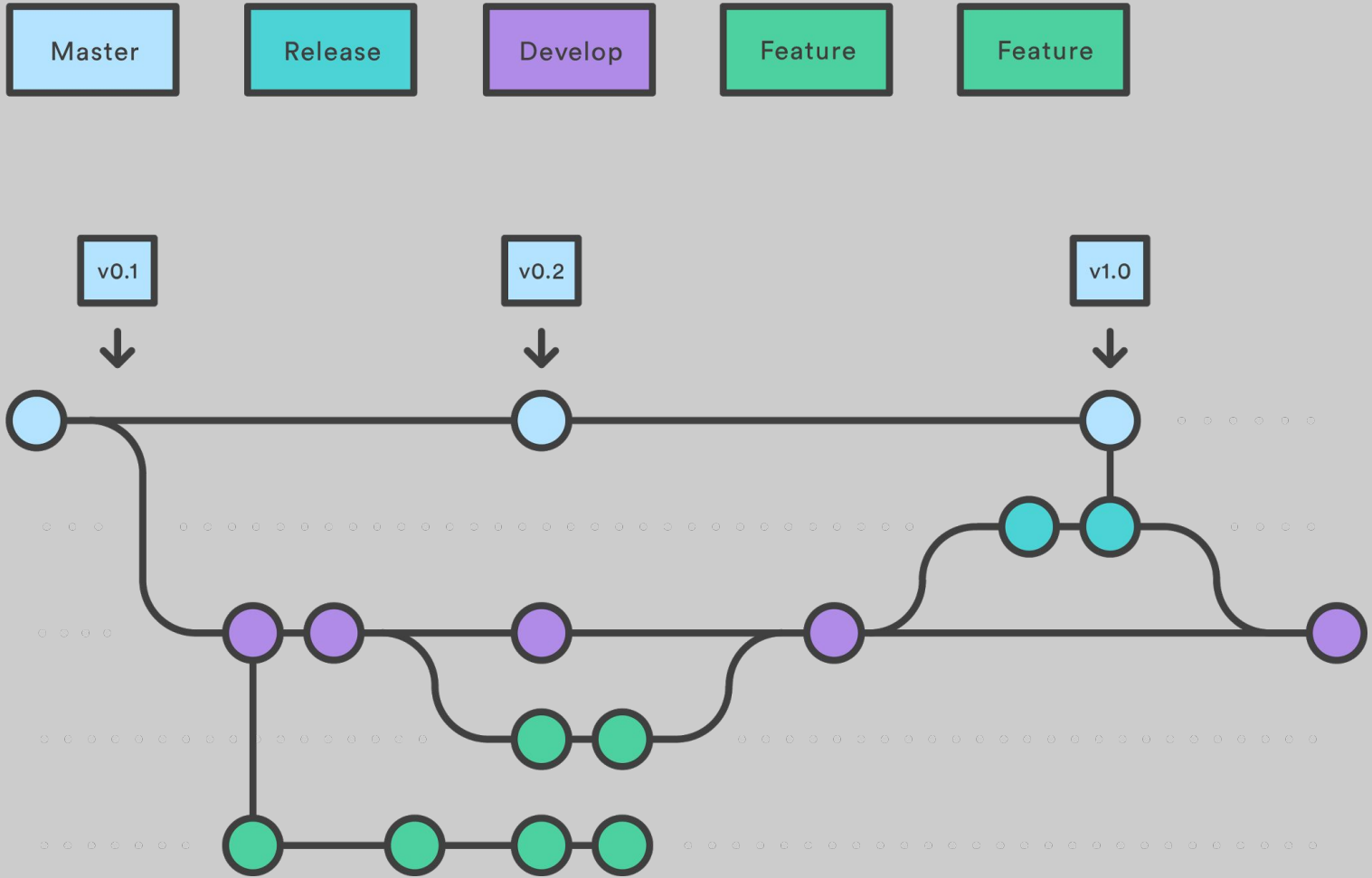
Every new feature should be developed inside its own `feature/<name>` branch. When the feature is ready it can be merged into **`develop`**. No feature should be merged directly into `master`.


Master
Develop
Feature
Feature
v0.1
v0.2
v1.0

`release`: used to prepare the code for a release while development on the **`develop`** branch continues

- Name convention: `release/<version>`
- Originates from: **`develop`**
- Merged into: first **`master`** then **`develop`**

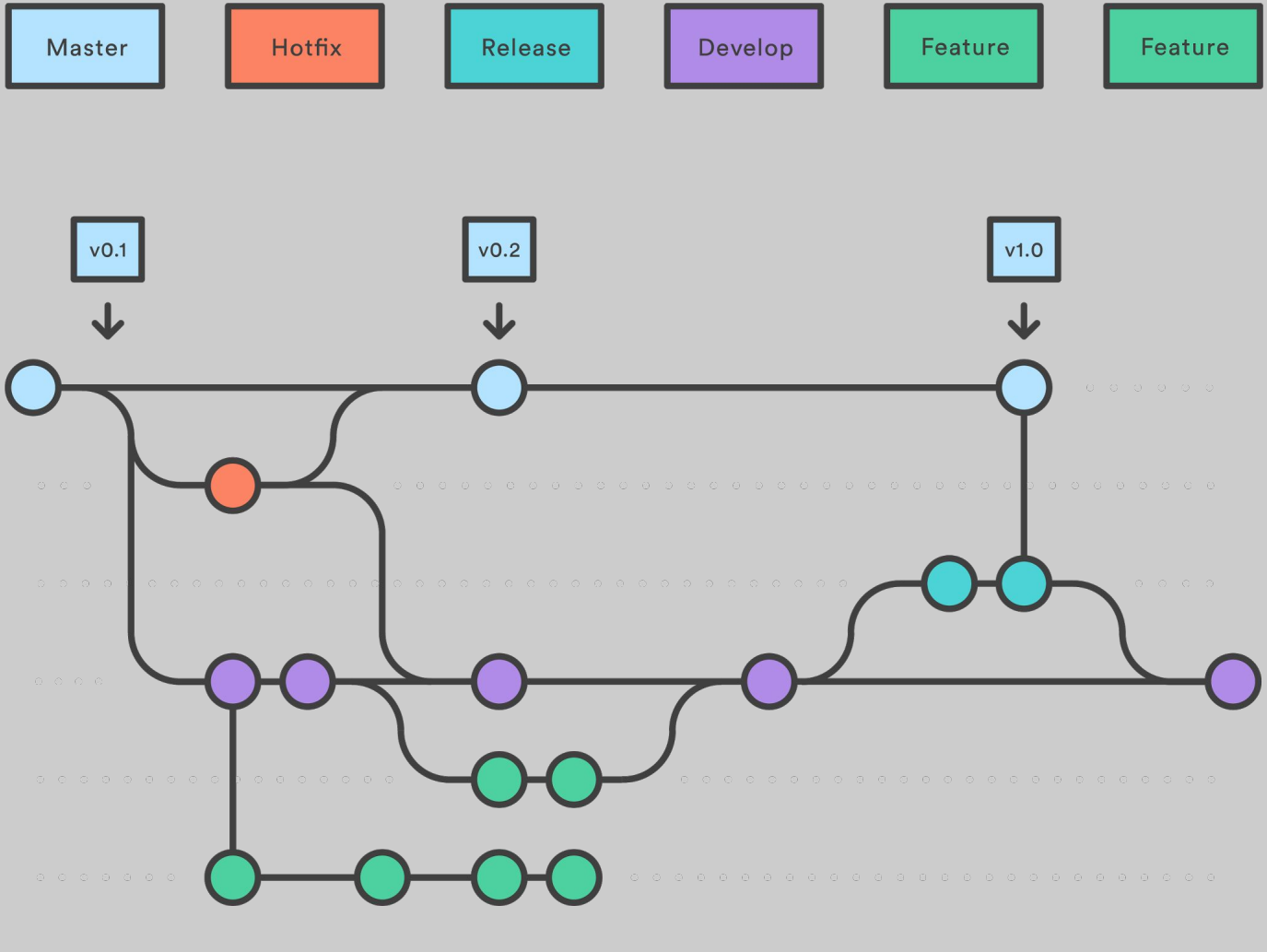
The **`release`** branch is not necessary in every project but can be useful when the development team implements a release workflow. The **`release`** branch allows to freeze the software in a stable condition (allowing only further testing and bug fix) while the development may continue on the **`develop`** branch.


Master
Release
Develop
Feature
Feature
v0.1
v0.2
v1.0

`hotfix`: used for severe bugfixes that prevent production from running properly

- Name convention: `hotfix/<name>`
- Originates from: `master`
- Merged into: `master` and `develop`

The `hotfix` branch is used only when a bug is found in the production code and allows a quick fix that is then integrated both into `master` and `develop`.


Master
Hotfix
Release
Develop
Feature
Feature
v0.1
v0.2
v1.0

The big picture

A repository can be created, for example on GitHub, by a team member who has the role of project maintainer. This repository is defined as **main** (or central) by a team convention (all repositories are equal for Git).

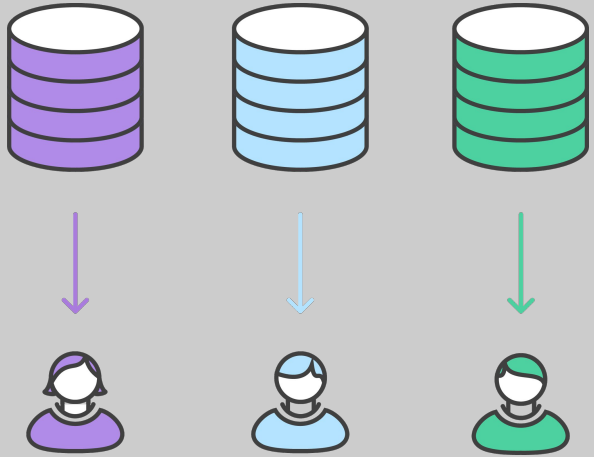
Maintainer creates main repository

Add all configuration files,
README and .gitignore

Using the built-in functionality of GitHub, the other team members **fork** the central repository thus making a copy of it on their accounts.

Developers fork the main repository

Every developer in the team **clones** her own fork of the repository in order to have a copy on the local machine. When cloning the repository, Git establishes a link setting the developer's own GitHub repository as a *remote* with name `origin`. The developer should also add another remote called `upstream` to link to the main repository (on the maintainer's GitHub account).



Developers clone their forked repositories

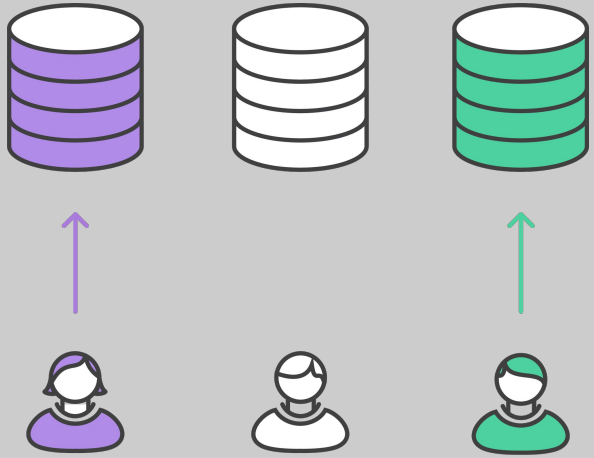
`origin`: developer's own fork
`upstream`: the main repository

Developers work on their features

Make local commits

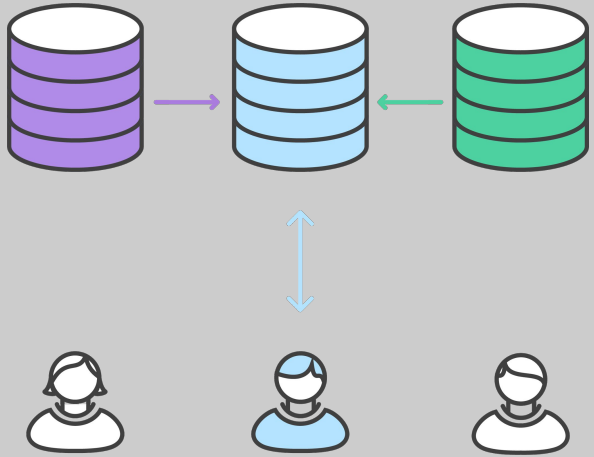
Developers can work independently on their machines making commits in the local repository. For this purpose feature branches should be used.

The developers then push their work (contained in feature branches) from the local repositories to their remote forks using the `remote/origin`.



Developers publish their features

Push to their forks (`remote/origin`)



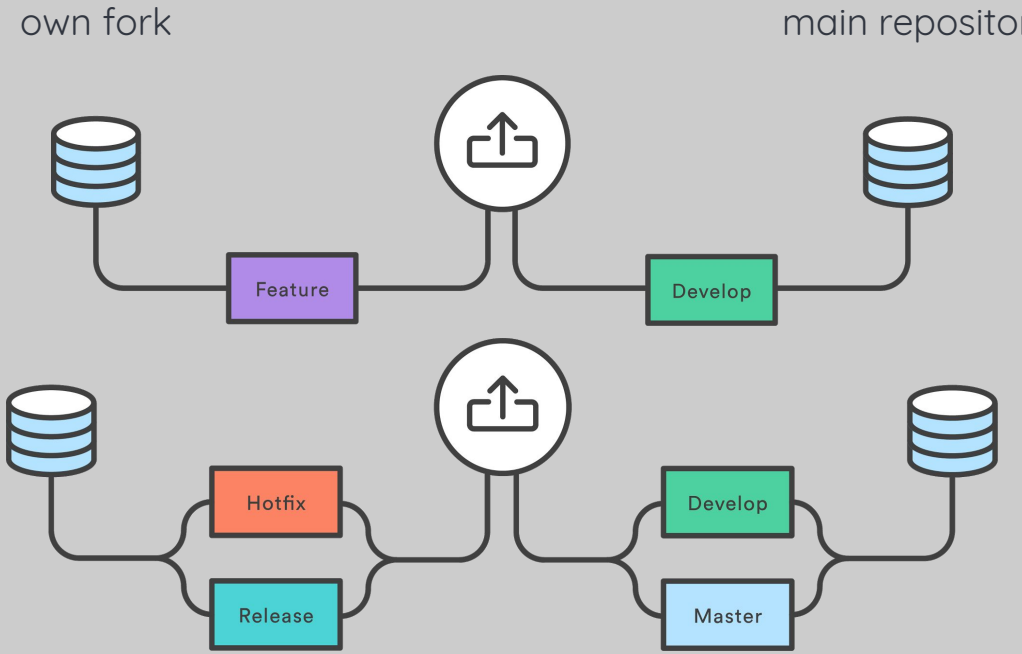
The project maintainer has the task of integrating the work of the developers into the main repository. In order to do this, **each developer opens a pull request from the feature branch on her own fork to the maintainer's develop branch** (on the main repository). After the integration is discussed in the pull request thread and all the possible conflicts are solved, the maintainer merges the developers' work into the main repository.

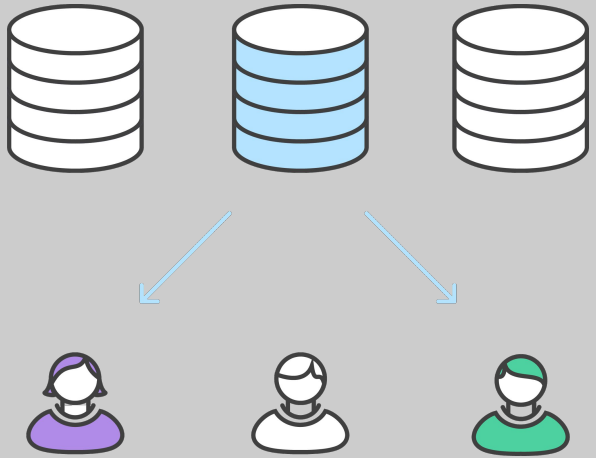
The project maintainer integrates their features

Developers use pull requests

Developers use pull request to ask the maintainer to integrate their features

Possible types of pull requests. The most common is from the developer's **`feature`** branch to the maintainer's **`develop`** branch. In case of **`hotfix`** and **`release`** branches the pull request should target both **`develop`** and **`master`** in the maintainer's main repository.





The last step is to synchronize the developers' repositories with the main repository. This is needed because only the maintainer's repository reflects the current state of the project after a merge, with the new feature integrated in the `develop` branch. In order to do this, each developer execute a `git pull` command (on the local machine) from the main remote repository, available as `remote/upstream`. Finally the developer can execute a push to `remote/origin` to keep her own local and remote repositories synchronized.

Developers synchronize with the main repository

Pull from `remote/upstream`

Learn more about *gitflow* at:

www.github.com/eschmar/gitflow

Credits

All the graphics in this presentation are from Atlassian *Getting Right Git* guide reachable at:

www.atlassian.com/git