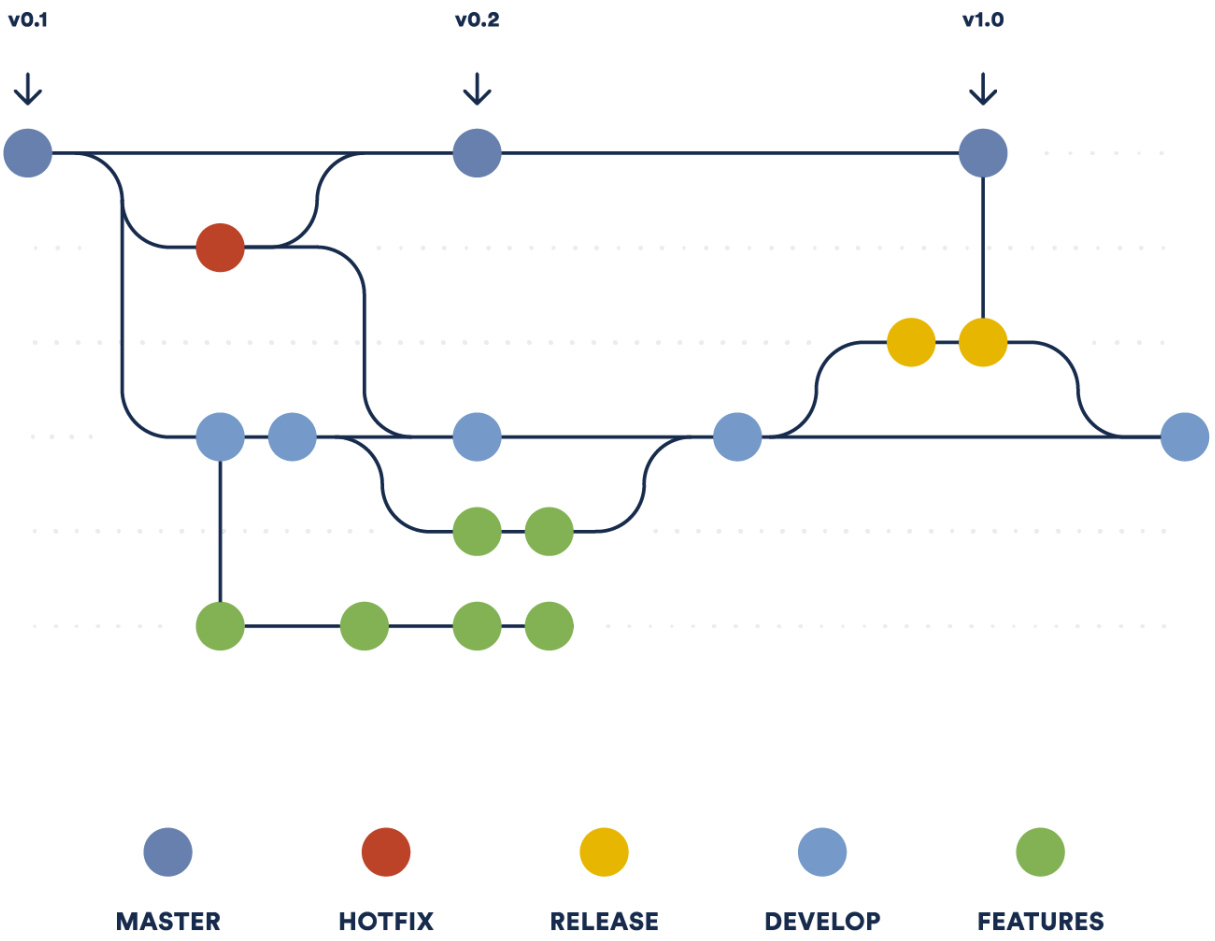
GitFlow-based branch management



Context

Why is GitFlow important?

- Once we are clear about what it is and why **Git** is fundamental, the next step is to lay the foundations of a good **workflow** in the use of this tool.

Definition of GitFlow

- **GitFlow** is a branch management model for **Git** that uses long-running feature branches and multiple main branches.
- It can be used for projects that have a scheduled release cycle, as well as for DevOps practices of continuous integration and continuous delivery.
- **GitFlow is a strict branching model designed around project releases.**

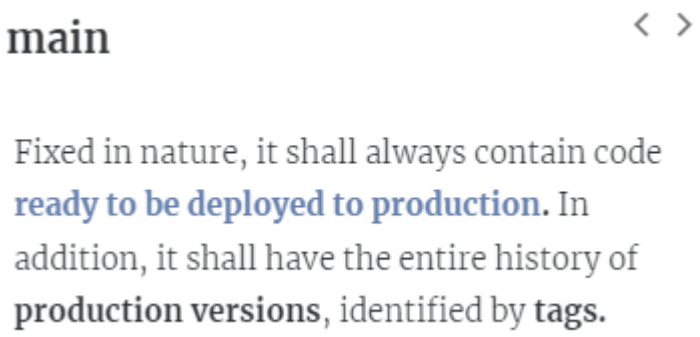
Branches

- In **Git**, **a branch is a version** of the project code you are working on. These branches help to maintain order in version control and to manipulate the code safely.

Types of branches

GitFlow defines 2 types of branches:

- **Fixed type**, which are responsible for recording the project history with an indefinite lifetime.
- **Auxiliary types**; as many of these can be created as necessary.



hotfix

Auxiliary in nature, it is used for **production errors.**

Nomenclature is:
hotfix- <BugIdentifier>

It is generated from the production version stored in the ***main*** branch.

Once the bug is fixed, this branch must be integrated with the ***main*** and ***develop*** branches, and then deleted.

release

Auxiliary in nature, it contains **versions of candidate code** to be stable.
Nomenclature is:
release- <VersionIdentifier>

Born from the ***develop*** branch for integration into the ***main*** **branch,** once it is determined that branch ***release*** is going to production.

develop

Fixed type, it is born from the ***main*** branch and contains the **latest changes** developed for the **next version of the software**.
It does not necessarily contain full functionality, but allows for compilation at any time.

features

Auxiliary in nature, it shall contain a new **development or upgrade**.
Nomenclature is:
feature- <Development Identifier>

It is often integrated with the ***develop*** **branch.**

bug

Auxiliary in nature, it is used to **resolve bugs in the production release.**

The nomenclature is:
bug-<VersionIdentifier>

Born from the ***release*** branch if functionality bugs are detected. Once the bug is fixed, it is promoted to both the ***release*** branch and the ***develop*** **branch,** in order not to keep the bug in the next developments.

Roles

Developer

- **Generates and updates the code** in the *feature*, *bug* and *hotfix* branches locally for subsequent remote versioning.
- He is mostly responsible for **opening the *merge request*** from the *feature*, *bug* and *hotfix* branch for integrating the code in *develop*.
- He is in charge of **keeping the local area "clean"**, determining when the *feature*, *bug* and *hotfix* auxiliary branches should be created and removed, respecting the team´s working arrangements.

Release Manager

- **Reviews and accepts** candidate code via *merge request* to *develop* branches raised by developers.
- **Resolves low-impact conflicts** after checking the *merge request* raised.
- **Identifies the versions of the solution** (*major* and *minor*) as well as gaining certainty of the deployments in the different environments.
- **Keeps the remote area "clean"**, determining the times when the *bug*, *hotfix* and *release* auxiliary branches should be removed in accordance with the team's working arrangements.
- **Performs secondary *merge request*** to replicate integrations to lower branches.

Branching flow

Below we will look at examples of versioning flows based on **GitFlow** for:

1. Creating a new *feature*.
2. Launching a *release*.
3. Fixing a *production* error.

Creating a new feature

Creation of feature branch in local

- According to the **GitFlow** flow, do not code in a **fixed** branch to create or modify a feature.
- The *feature* branch is used for this. It is the responsibility of the developer to create this new local branch from the *develop* branch.

Creation of feature branch in origin

- Once the local branch is generated, the developer performs a *push* to *origin* of the *feature* branch created, allowing other team members to work on the same functionality.
- In environments where automated continuous integration processes are in place (with **Jenkins**, for example), the compilation and execution of code quality processes will begin.

Work on feature

- Once the remote branch is created, the developer can make all the necessary modifications to his local ***feature*** branch.
- When the developer is happy with his modifications and wishes to upload his changes, it is very important that he performs a ***pull*** of the remote ***feature*** branch to get all possible changes that are available.
- In this way, the necessary *merge* can be performed locally before running the *push* cleanly to the remote ***feature*** branch.

At the end of the development

- Once it is determined that the development of the functionality is complete, ***merge* to the *develop*** branch.
- To do this, the developer must ensure that all possible changes that have occurred in the *develop* branch are in the *feature* branch.
- A *pull* from *develop* to *feature* to resolve possible **conflicts in local must always take place**.
- Once these are resolved, *push* to the **remote *feature*** branch. The developer finally makes a *merge request*.

Acceptance of merge requests

- The person responsible for accepting or rejecting the *merge request* is the ***Release Manager***.

- If the change is accepted, and it is determined that all features at this point are in *develop* and are *release candidates*,
- The ***Release Manager*** can create a remote *release* branch that freezes the code for testing in some environment, e.g., pre-production.

Moving to production

- Once the version of the *release* branch has been scanned and tested in the pre-production environment and everything is correct,
- The ***Release Manager*** can make the *merge request* from the remote *release* **branch to the *main*** branch.

Quick considerations about creating features process

- The *feature* branch always originates from the *develop* branch created by the ***developer***.
- Before performing a *push*, always perform a *pull* to ensure that possible conflicts are resolved locally before uploading to remote.
- The creation of the *merge request* to *develop* is the responsibility of the ***developer.***
- The ***Release Manager*** is responsible for accepting the *merge requests* raised.
- **Fixed-type** main and develop branches are never modified directly, and an auxiliary branch must always be generated to make modifications.

Workflow for a new release

Creating the release branch

- The ***Relase Manager*** has created a *release* branch from the production candidate features from *develop*. In environments with CI/CD automation.
- The build, test and deployment processes will be executed and the new version will be available for testing.

Detection of a functional failure

- When a client detects a bug, the ***developer*** must create a *bug* branch in their local environment from the *release* branch.

- It is then important to *push* this branch to the remote repository, allowing other team members to work on the same bug if necessary.

Working and promoting corrections in a functional failure

- Once the remote branch is created, the ***developer*** can make all the necessary modifications to his local ***bug*** branch.
- When he is happy with the fixes and wishes to upload his changes, it is very important that a *pull* of the **remote *bug*** branch is performed to get all possible changes that are available from other *developers*.
- This way he can perform the necessary *merge* locally before *pushing* cleanly **to the remote *bug*** branch.

At the end of the bit correction

- Once it is determined that the bug fix is complete, *merge* to the *release* branch.
- To do this, the ***developer*** must ensure that all possible changes that have occurred in the *release* branch are in the ***bug branch***.
- Always *pull* from *release* to *bug* to resolve local conflicts, and then *push* to the ***remote bug*** branch so that the developer can proceed with the *merge request*

Acceptance and propagation of the Merge Request

The person responsible for accepting or rejecting the *merge request* is the ***Release Manager***.

If it is accepted and the correction is determined to be correct and proper, the ***Release Manager*** propagates the correction to both the **develop branch** via another *merge request* and the ***release branch***.

Moving to production

Once the version of the ***release* branch** has been scanned and tested in the pre-production environment and everything is correct,

The ***Release Manager*** can issue the *merge request* from the ***release* remote branch** to the **main branch**.

Quick considerations about release process

- The ***bug branch*** always originates from the **branch *release*** and is created by the ***developer***.
- Before performing a *push*, always perform a *pull* to ensure that potential conflicts are resolved locally before promoting to remote.
- The ***developer*** is responsible for creating to *merge request* to release.
- The ***Release Manager*** is responsible for accepting the *merge requests* raised and propagating them to both *develop*, *release* and *main*.

Workflow to fixing a *production* error

- When an error is detected in production, **it must be resolved as soon as possible**.
- For production bugs, the ***developer*** must generate a **hotfix *branch* locally** from the ***main branch***, and then *push* the branch to the remote repository.

Work and promotion of a production error

- Once the remote branch has been created, the ***developer*** can make all the necessary modifications to his **local *hotfix* branch.**
- When the ***developer*** is done with the fix and wants to upload his changes, it is very important that a *pull* of the **remote *hotfix* branch** is performed to get all possible changes that are available from other developers.
- This way the necessary *merge* can be performed locally before *pushing* cleanly to the **remote *hotfix* branch**.

Upon completion of the error correction

- Once the bug fix is complete, *merge* into the **branch *main***.
- To do this, the ***developer*** must ensure that all possible changes that have occurred in the ***main* branch** are in the ***hotfix* branch** he is working on.
- A *pull* from *main* to *hotfix* to resolve local conflicts is performed, and then a *push* to the **remote *hotfix* branch**, so that the ***developer*** makes a *merge request* to *main*.

Acceptance and propagation of the Merge Request

- The ***Release Manager*** is responsible for accepting or rejecting the *merge request*.
- If accepted and the correction is determined to be correct, the ***Release Manager*** propagates the correction to both the ***release***, ***develop*** and ***main*** **branches** via another *merge request*.

Quick considerations to fix a production bug

- The ***hotfix*** **branch** always originates from the ***main*** **branch** and is created by the *developer*.
- Before performing a *push*, a *pull* should always be performed to ensure that potential conflicts are resolved locally before promoting to the remote repository.
- The creation of the *merge request* to *main* is the responsibility of the ***developer***.
- The ***Release Manager*** is responsible for accepting the *merge request* raised and propagating it to both *develop*, *release* and *main*.

What are tags?

- ***Tags*** are strings that point to a specific *commit*.
- A ***tag*** is a *string* that identifies a specific and important point in the history of a repository.
- **Git** has the ability to tag specific points in history. This functionality is typically used to tag release versions, for example: **v1.0.0.0**, **v1.0.1**,and so on.

Git uses two main types of tags:

1. Annotated.
2. Lightweight.

Annotated

Annotated tags are stored in the **Git** database as integer objects. They contain:

- Name of **tagger**
- **E-mail**
- **Date**

- An **associated message**

To view the detail of a previously created tag, use the ***git show*** command followed by the tag name.

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Lightweight

A **lightweight tag** is much like a branch that does not change; it is simply a pointer to a specific ***commit.*** They are used if for some reason you want a temporary tag or if you don't need additional information.

Lightweight tags contain:

- The **commit**
- **Author**
- **Date of creation**

To view the detail of a previously created tag, run the ***git show*** command, followed by the tag name.

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number
```

Checkout for a tag

In **Git**, it is possible to check out a tag.

In case a tag is created in an unwanted *commit* we can proceed as follows:

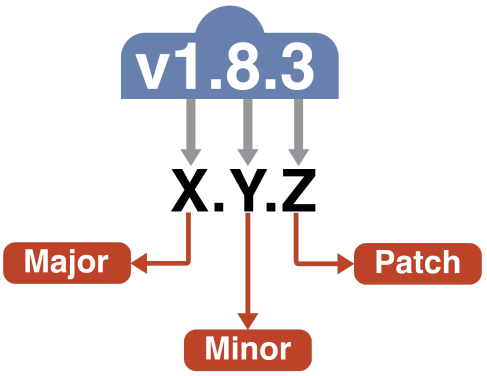
1. It can be moved by creating the tag again and **Git** will notify that it already exists.
2. A second option is to delete the tag and create it again at the correct commit.

As a special case, if you want to make changes to the working directory from a repository version that matches a tag, you must create a new branch from the tag with the command:**git checkout -b [name_branch] [tag]**

Versioning tag nomenclature

The versioning in **GitFlow** is based on the three-digit X.Y.Z. semantic versioning nomenclature.

Each digit indicates what the changes in the new versions look like as follows.



A nomenclatura de tags no Git Flow segue um padrão semântico conhecido como SemVer (Semantic Versioning). Esse padrão é amplamente utilizado para gerenciar as versões de um software de maneira clara e previsível. Ele define como as versões devem ser incrementadas com base nas mudanças feitas no código, garantindo assim compatibilidade e facilitando a gestão de dependências. A nomenclatura básica do SemVer é `MAJOR.MINOR.PATCH`, onde:

- **MAJOR**: Indica uma versão que faz mudanças incompatíveis com a API anterior. Essencialmente, quando você faz uma alteração que não é compatível com versões anteriores, incrementa-se a versão maior. Isso pode incluir alterações substanciais na funcionalidade, remoção de funcionalidades antigas, ou qualquer outra mudança que obrigue os usuários a alterar a forma como interagem com seu software.
- **MINOR**: Refere-se a uma versão que adiciona funcionalidades de maneira compatível com versões anteriores. Quando você adiciona novas funcionalidades que não quebram a compatibilidade com versões anteriores do software, você incrementa a versão menor. Essas mudanças devem ser substanciais o suficiente para merecer um novo número de versão, mas não tão disruptivas a ponto de necessitar de uma nova versão maior.
- **PATCH**: Usado para lançamentos que fazem correções de bugs compatíveis com versões anteriores. Se você está corrigindo bugs, melhorando a performance, ou fazendo pequenas melhorias que não afetam a forma como os usuários interagem com o software, você

incrementa a versão de correção. Essas alterações são geralmente pequenas mas importantes para a manutenção do software.

No contexto do Git Flow, esses princípios de versionamento são aplicados da seguinte maneira:

- **main/master**: É a branch principal onde o código está em estado de produção, ou seja, pronto para ser entregue ou já entregue. As versões aqui devem ser marcadas com tags seguindo a nomenclatura SemVer, refletindo as mudanças feitas desde a última versão. Por exemplo, se você está lançando uma nova funcionalidade que muda a maneira como os usuários interagem com seu produto, você pode criar uma tag `2.0.0` para indicar uma nova versão maior.
- **hotfix**: Branches de hotfix são usadas para fazer correções rápidas em produção. Essas correções são geralmente urgentes e precisam ser feitas fora do ciclo normal de desenvolvimento. Depois de um hotfix ser aplicado e mesclado de volta à branch principal, a versão deve ser incrementada seguindo a regra do PATCH. Por exemplo, se a versão atual em produção é `2.0.0` e você faz um hotfix para corrigir um bug crítico, você pode marcar a nova versão como `2.0.1`.

Manual and automated versioning

The process of versioning and tagging is a difficult thing to manage in a busy day-to-day life and automating it is very useful. Below we will see, step by step, each of these cases:

1. Manual versioning
2. Automatic versioning done with **Jenkins**

Manual versioning

- Two fixed ***master*** and ***develop*** branches, with an initial 2.**5**.0 version in the *master* branch.
- Subsequently, a copy is made to the *develop* branch and we start working on an upgrade of a functionality and, therefore, the digit **Y** is modified, which corresponds to a ***minor*** version, i.e. the version would become:
- 2.**6**.0-**SNAPSHOT**
- Note: SNAPSHOT = work in progress, not stable and for testing purposes.

Version in feature

- Subsequently, we will create a ***feature*** branch, which inherits the same version of *develop*; in our case it will be:
- **2.6.0-SNAPSHOT**