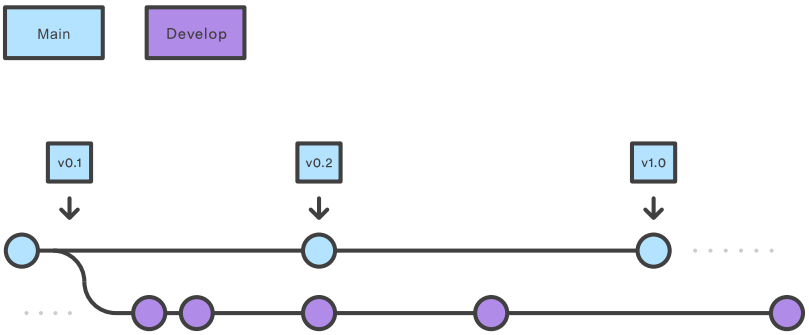
Software workflow and release guidelines

Document version	1
Author	Ing. Enrico Gambini

Version history:

Version n°	Date	Responsible	Description
1	06/06/2023	Enrico Gambini	First issue

Repository workflow (Gitflow):



Develop and main branches

Instead of a single `main` branch, this workflow uses two branches to record the history of the project. The `main` branch stores the official release history, and the `develop` branch serves as an integration branch for features. It's also convenient to tag all commits in the `main` branch with a version number.

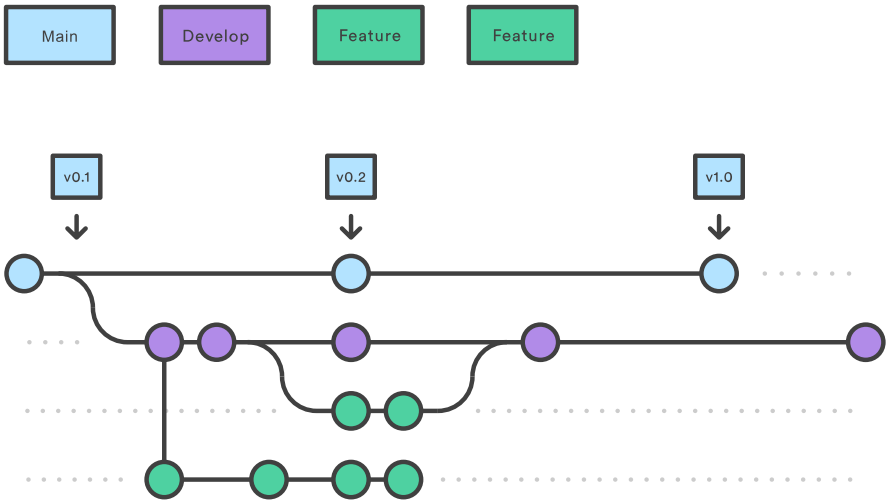
The first step is to complement the default `main` with a `develop` branch. A simple way to do this is for one developer to create an empty `develop` branch locally and push it to the server:

```
git branch develop
git push -u origin develop
```

This branch will contain the complete history of the project, whereas `main` will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for `develop`.

Feature branch

Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of `main`, `feature` branches use `develop` as their parent branch. When a feature is complete, it gets merged back into develop. Features should never interact directly with `main`.



`Feature` branches are generally created off to the latest `develop` branch.

Creating a feature branch

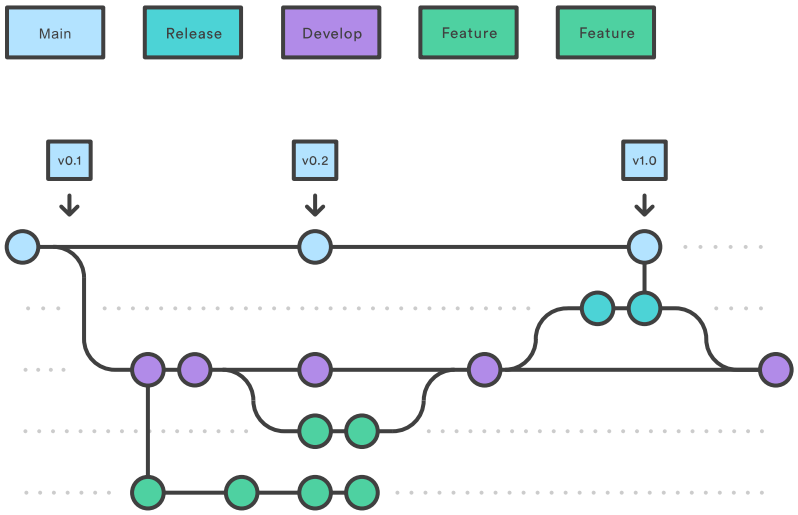
```
git checkout develop
git checkout -b feature/feature_branch
```

Finishing a feature branch

When you're done with the development work on the feature, the next step is to merge the `feature_branch` into `develop`.

```
git checkout develop
git merge feature/feature_branch
```

Release branches



Once `develop` has acquired enough features for a release (or a predetermined release date is approaching), you fork a `release` branch off of `develop`. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the `release` branch gets merged into `main` and tagged with a version number. In addition, it should be merged back into `develop`, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making `release` branches is another straightforward branching operation. Like `feature` branches, `release` branches are based on the `develop` branch. A new `release` branch can be created using the following methods.

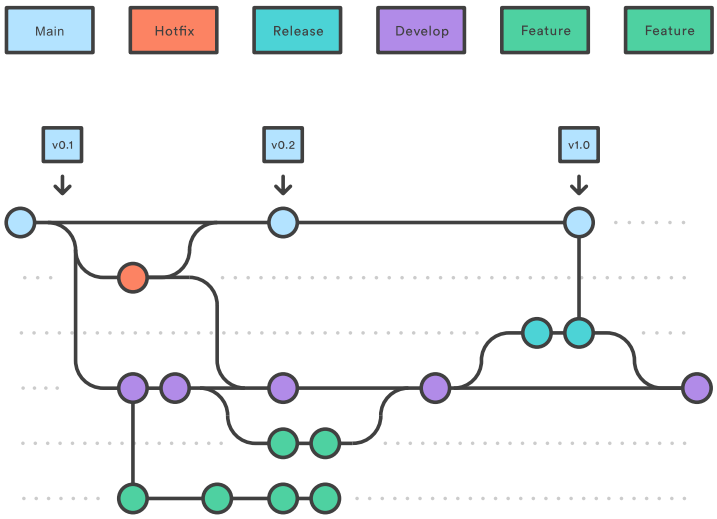
```
git checkout develop
git checkout -b release/0.1.0
```

Once the release is ready to ship, it will get merged it into `main` and `develop`, then the `release` branch will be deleted. It's important to merge back into `develop` because critical updates may have been added to the `release` branch and they need to be accessible to new features. If your you need code review, this would be an ideal place for a pull request.

To finish a `release` branch, use the following methods:

```
git checkout main
git merge release/0.1.0
```

Hotfix branches



Maintenance or "`hotfix`" branches are used to quickly patch production releases. `Hotfix` branches are a lot like `release` branches and `feature` branches except they're based on `main` instead of `develop`. This is the only branch that should fork directly off of `main`. As soon as the fix is complete, it should be merged into both `main` and `develop` (or the current `release` branch), and `main` should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad-hoc `release` branches that work directly with `main`. A `hotfix` branch can be created using the following methods:

```
git checkout main
git checkout -b hotfix/hotfix_branch
```

Similar to finishing a `release` branch, a `hotfix` branch gets merged into both `main` and `develop`.

```
git checkout main
git merge hotfix/hotfix_branch
git checkout develop
git merge hotfix/hotfix_branch
git branch -D hotfix/hotfix_branch
```

Example

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a `main` branch.

```
git checkout main
git checkout -b develop
git checkout -b feature_branch
# work happens on feature branch
git checkout develop
git merge feature_branch
git checkout main
git merge develop
git branch -d feature_branch
```

In addition to the `feature` and `release` flow, a `hotfix` example is as follows:

```
git checkout main
git checkout -b hotfix_branch
# work is done commits are added to the hotfix_branch
git checkout develop
git merge hotfix_branch
git checkout main
git merge hotfix_branch
```

Summary

Some key takeaways to know about Gitflow are:

- The workflow is great for a release-based software workflow.
- Gitflow offers a dedicated channel for hotfixes to production.

The overall flow of Gitflow is:

- A `develop` branch is created from `main`
- A `release` branch is created from `develop`
- `Feature` branches are created from `develop`
- When a `feature` is complete it is merged into the `develop` branch
- When the `release` branch is done it is merged into `develop`and `main`
- If an issue in `main` is detected a `hotfix` branch is created from `main`
- Once the `hotfix` is complete it is merged to both `develop`and `main`

Release version numbering

Semantic versioning like Major.Minor.Patch sequence. When a major, minor, or patch update is made, the corresponding number is increased.

- Major version changes are related to incompatible API changes. E.g. when communication protocol between boards or cloud is changed.
- Minor version changes are related to adding new functionality in a backward-compatible manner.
- Patch version changes are related to bug fixes which are also backward compatible.

The version number is increased by one digit, depending on the type of changes made in the new versioning.
For example, software version 1.3.1 means that the software is in its first major version, with three additional functionalities added, and one patch fix implemented. When a developer fixes a bug in that version, the next version release is named version 1.3.2.
More than one bug could be grouped and resolved in a single release, especially if bugs are low priority.