10
Gitflow Workflow Written by Jawwad Ahmad

Gitflow is a workflow that Vincent Driessen introduced in his 2010 blog post, A successful Git branching model.
At its core, Gitflow is a specialized version of the branching workflow. It introduces a few different types of branches with well-defined rules on how code can flow between them.
Vincent posted a ten-year update titled “Note of reflection” on March 5th, 2020, at the beginning of his original blog post. In his note, he recommends that you should consider whether this is the right workflow for you.
He notes that Gitflow is great for versioned software, but a simpler approach might work better in today’s age of continuous deployment. He ends his note with the words: “Decide for yourself.”
In this chapter, you’ll learn about the rules that make up Gitflow, as well as the reasons behind them. This will allow you to decide if Gitflow is right for you.

When to use Gitflow

Gitflow is a good fit if you’re building software that’s explicitly versioned, especially if you need to support multiple versions of your software at the same time. For example, you might release a 2.0 version of a desktop app that’s a paid upgrade, but still want to continue releasing minor bug fix updates for the 1.0 version.
Gitflow is also a good fit if your project has a regular release cycle. Its release branch workflow allows you to test and stabilize your release while normal day-to-day development continues on your main development branch.
Managing larger projects is easier with Gitflow since it has a well-defined set of rules for the movement of code between branches.
Gitflow is less ideal in scenarios that favor a continuous deployment model, such as web development. In these situations, Gitflow’s release workflow might add unnecessary extra overhead.

Chapter roadmap

In this chapter, you’ll first get a quick introduction to the basic concepts in Gitflow. You’ll learn about the different long-lived and short-lived branches and the rules for how to create and merge them.
You’ll then install the git-flow extensions, which aren’t necessary to use the Gitflow workflow itself, but make it easier to adopt. The term **git-flow** will be used to refer to the extensions, and **Gitflow** will be used to refer to the workflow itself.
Once installed, you’ll learn how to use the git-flow extension commands to create and merge the various types of branches Gitflow uses.

Types of Gitflow branches

Gitflow uses two long-lived branches: **main** (or **master**) and **develop** and three main *types* of short-lived branches: **feature**, **release** and **hotfix**. While you never delete long-lived branches, you delete short-lived branches once you merge them into a long-lived branch.
There are well-defined rules about how and when to create short-lived branches, and rules for merging them back into the long-lived branches. You’ll learn about these rules in a bit. But first, you’ll learn about the purpose of the various branch types.

Long-lived branches

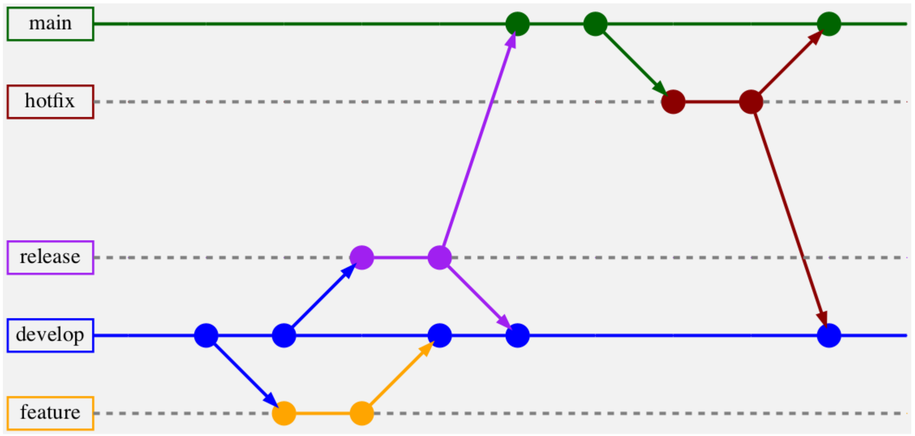
Git itself uses a single long-lived branch, typically named **main** or **master**. Gitflow introduces the concept of an additional long-lived *production* branch.
Instead of introducing a new name for this production branch, Gitflow repurposes the **main** branch for this role. This means that the **main** branch can now *only* contain code that’s been released to production, or that will be released to production as a result of it being merged to **main**.
Consequently, Gitflow adds a **develop** branch for the role that the **main** branch had previously played, i.e., for normal day-to-day development.
Code can only be added to, and moved between the long-lived branches using short-lived branches, which you’ll learn about next.

Short-lived branches

The three main types of short-lived branches are **feature**, **release**, and **hotfix**.
Feature branches: Just as in the typical branching workflow, you do all your new development here. You create a **feature** branch when you add new functionality to your app, such as a new settings screen or improvements to the login flow.
Release branches: Use these to prepare and test code on the **develop** branch for a production release. If you find any bugs, you fix them on the **release** branch. These branches are also suitable for tasks like updating release notes and versions.
Hotfix branches: These are used to fix bugs already in production.
There’s also a less commonly-used type of branch known as a **support** branch. You use these only when you need to support previously released versions of your software.

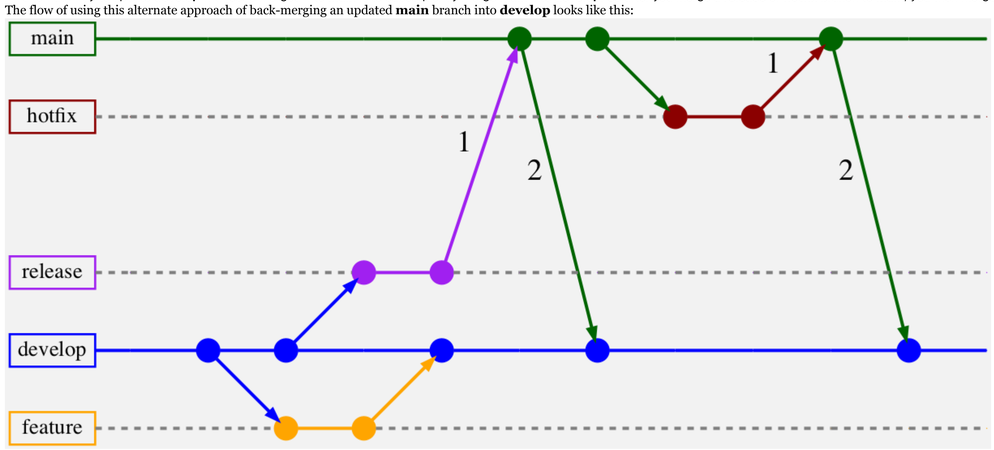
Rules for creating and merging branches

Here are the rules for creating and merging branches:
Feature branches are created only from **develop** and are only merged into **develop** since that’s the branch you use for normal day-to-day development.
Release branches are created only from **develop** and are merged into both **main** and **develop**. The additional merge to **develop** ensures that any updates or bugfixes you commit on the **release** branch make their way back into **develop**.
Hotfix branches are created only from **main** and are merged into both **main** and **develop**. This ensures that bugs you fix in production are also fixed on develop.
Here’s an image that displays an example of the branching and merging flow:



Gitflow's branching and merging flow for feature, release and hotfix branches

Solid lines represent long-lived branches, dashed lines represent short-lived branches, and arrows represent the branch creation and merging flow.
Here are a few alternate ways of thinking about the branch creation and merging rules, which may help you grasp the flow better:
The only branch you can create from **main** is the **hotfix** branch since it's only meant to fix bugs in production. You can't create **hotfix** branches from **develop** because merging them would also deploy any additional features committed to **develop** since the last release.
Any code merged to **main** that isn't already in **develop** needs to be added to **develop**. This is why, when you merge the **release** and **hotfix** branches to **main**, you also need to merge them into **develop**.
An alternate way to update **develop** with code merged into **main** is to subsequently merge **main** into **develop**. So after you merge a **release** or **hotfix** branch to **main**, you then merge **main** into **develop**. Think of this as **back-merging** main into develop.
The flow of using this alternate approach of back-merging an updated **main** branch into **develop** looks like this:



An alternate flow for hotfix and release that back-merges main into develop

Both approaches are acceptable, but there's a slight benefit to the back-merging approach, which you'll learn about later.
Next, you'll install the git-flow extensions and start playing with the various types of short-lived branches.

Installing git-flow

The git-flow extensions are a library of Git subcommands that make it easier to adopt the Gitflow workflow. For example, a single `git flow release finish` command will merge your release branch into main, tag the release, merge it back into develop and then finally delete the release branch.
Essentially, the git-flow extensions run sequences of Git commands that encode specific Gitflow workflows.
You have two options for installing the git-flow extensions. The first is from the creator of Gitflow, Vincent Driessen; you can find it at https://github.com/nvie/gitflow.
Unfortunately, Vincent Driessen no longer maintains this project, so installing this version isn't recommended. Its last release was in 2011, and you'll install that outdated version if you simply run `brew install git-flow` on macOS, so don't do that!
The recommended version to install is the AVH version by Peter van der Does, which is available at https://github.com/petervanderdoes/gitflow-avh, and is available on Homebrew as **git-flow-avh**.
Note: If you've already installed the default `git-flow`, you can uninstall it via `brew uninstall git-flow`. Alternatively, if you've installed both, you can overwrite the older version with `brew link --overwrite git-flow-avh`.
If you need to install Homebrew, see https://brew.sh. To install git-flow-avh on Windows, see https://github.com/petervanderdoes/gitflow-avh/wiki/Installing-on-Windows.
Run the following to install the AVH version of git-flow:

```
brew install git-flow-avh
```

To verify the version you've installed, run the following:

```
git flow version
```

You should see **1.12.3 (AVH Edition)**. If you see **0.4.1** instead, you have the original, unmaintained version installed. See the note above on how to uninstall it.
If you have trouble installing git-flow, you can also use the manual Git commands that git-flow would run, which will also be provided in the chapter.
Next, you'll configure the starter project to use git-flow.

Initializing git-flow

Unzip **repos.zip** from the **starter** folder for this chapter. You'll only be working within the **alex/checklists** repository, so the **beth** and **chad** directories from previous chapters aren't included.
You'll see the following unzipped directories within **starter**:

```
starter
└── repos
    ├── alex
    │   └── checklists
    └── checklists.git
```

As in the previous chapter, the local **checklists.git** directory is configured as the remote origin for the **alex/checklists** repository. This means when you run `git push` from **alex/checklists**, it will push to the **checklists.git** directory.
Open a terminal window and **cd** to the **alex/checklists** directory in **starter/repos**.

```
cd path/to/projects/starter/repos/alex/checklists
```

Before you start using git-flow, you'll need to initialize a few configuration settings.
Run the following command to initialize a git-flow configuration for this repository:

```
git flow init
```

Press **Enter** to accept each of the defaults. However, for the **Version tag prefix?** question, use a lowercase **v** since this is a very common convention.
You'll see the following:

```
Which branch should be used for bringing forth production releases?
   - main
```

```
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? [] v
Hooks and filters directory? [...]
```

If you accidentally miss the question, you can use `git flow init -f` to re-initialize it. Alternatively, you can manually edit the **.git/config** file by changing the **versiontag =** line to **versiontag = v** and save it.
Gitflow uses branch prefixes to differentiate between different types of branches. These prefixes will automatically be added when using the various git-flow **start** commands to create the specified type of branch.
Note: You may notice there is also a **bugfix** type of branch. This type of branch wasn't present in Vincent's implementation of git-flow or in his original workflow. This was added in git-flow-avh for fixing bugs on **develop**. It can be thought of as equivalent to a **feature** branch, but with an alternate prefix that more clearly indicates that the branch is for a bugfix instead of a feature.
Running `git flow init` will also create a **develop** branch from the **main** branch if one doesn't exist already.
You're now ready to add a new feature using Gitflow, which you'll do next.

Creating and merging a feature branch

Gitflow uses feature branches for work on new features, just as the branching workflow does. Since the day-to-day development branch in Gitflow is now **develop** instead of **main**, you create feature branches from **develop** and merge them back into **develop** when you're finished. You cannot create feature branches from **main** or any of the other short-lived branches.
Make sure you're still in the **checklists** folder and create a **feature** branch by running the following command:

```
git flow feature start update-h1-color
```

This is equivalent to running the following command from the develop branch:

```
git checkout -b feature/update-h1-color  # equivalent to above
```

You'll see the following **Summary of actions** that confirms what the command did:

```
Summary of actions:
- A new branch 'feature/update-h1-color' was created, based on 'develop'
- You are now on branch 'feature/update-h1-color'
```

Now, you'll make a minor change and commit this feature. Open **style.css** and, on the second line, change the color of the **h1** tag from **navy** to **blue**.

```
 h1 {
-    color: navy;
+    color: blue;
 }
```

Run the following command to commit the change:

```
git commit -am "Updated h1 color from navy to blue"
```

Now that you've completed the feature, you'll merge the **feature/update-h1-color** branch back into **develop** using git-flow.
While the previous `git flow feature start` command didn't save much typing over the actual checkout command, the **finish** version of this command does a bit more.
In a single command, it will perform the following three actions:

```
1. Checkout the develop branch           # git checkout develop
2. Merge the feature branch  # git merge feature/update-h1-color
3. Delete the branch     # git branch -d feature/update-h1-color
```

Run the following command to finish the feature:

```
git flow feature finish
```

You'll see the following **Summary of actions** for it:

```
Summary of actions:
- The feature branch 'feature/update-h1-color' was merged into 'develop'
- Feature branch 'feature/update-h1-color' has been locally deleted
- You are now on branch 'develop'
```

You saved some typing, and that's nice. But the individual commands aren't difficult to remember since they're common used in Git. Why bother, then?
The main benefit is that the git-flow extensions automatically enforce Gitflow's rules for you and prevent mistakes. For example, they'll prevent you from accidentally creating a feature branch from main or accidentally merging a completed feature branch into main.
When you're working on release and hotfix branches, the `git flow finish` commands will save you even more typing (and remembering) since you need to merge them into both develop and main. This is the perfect segue to learn about them in the next section!

Creating and merging a release branch

Release branches are where you prepare code for an upcoming release. They let you run tests and implement fixes while the day-to-day development continues on the development branch.
Since release branches hold release code under development, you create them from the **develop** branch and merge them into **main**. You also merge them back into **develop** so that any additional bug fixes and updates committed to the release branch make it back into the **develop** branch.
You generally name release branches using a version number, then use that same version number to tag the release.
The AVH version of git-flow includes several improvements over the original, including a `--showcommands` option, which shows you the Git commands executed while running a git-flow command.
Run the following to create a new release branch and to see the command it uses:

```
git flow release start 1.0.0 --showcommands
```

Ignore the first `git config --local ...` line, which git-flow uses internally for tracking things. On the second line, you'll see:

```
git checkout -b release/1.0.0 develop
```

The extra `develop` argument listed at the end is the start point (or the base) of the `release/1.0.0` branch. It's equivalent to running `git checkout develop` and then `git checkout -b release/1.0.0`.
Now, you'll make an additional update to the release branch. There aren't any bug fixes to make, but you'll add a new VERSION file.
Run the following to add the new VERSION file and commit it:

```
echo '1.0.0' > VERSION
git add VERSION
git commit -m "Adding VERSION file for initial release"
```

Now the release branch is complete. It's time to merge into main to deploy it.
You'll also want to bring the commit that adds the VERSION file back into the develop branch. Recall that there are two ways to accomplish this: You can either merge `release/1.0.0` into `develop`, or you can *back-merge* `main` into `develop`.
For your next step, you'll use the back-merge approach, which the AVH version of `git-flow` uses by default.
You'll do this by typing a single git-flow command shortly.
However, if you were to manually perform these actions, the commands would be:

```
## NOTE: Do not execute these! You'll be using git-flow for this.

# Merge release into main
git checkout main
git merge --no-ff release/1.0.0

# Tag the release
git tag -a v1.0.0

# Merge main back to develop
git checkout develop
git merge --no-ff main

# Delete the branch
git branch -d release 1.0.0
```

Note: To merge main back into develop, git-flow uses a reference to the tag instead of using main. The result is the same since both the tag and main resolve to the latest commit on main. However, using the tag as a reference results in a more specific commit message since it includes the version.
With git-flow, run the following command:

```
git flow release finish --showcommands
```

Now, you'll need to save three commit messages. Type **:wq** to accept the default message for merging `release/1.0.0` into `main`. Next, for the tag message, type **Tag for 1.0.0 release** and save it with **Esc**, **:wq**.
Before accepting the last message, note that it says: **Merge tag 'v1.0.0' into develop**. If you had used `main`, it would have said: **Merge branch 'main' into develop**. It's nice to see the specific version that was merged in the actual merge message.
Type **:wq** once more to accept the default message for the merge.
Here's the last part of the output, with the third line confirming the back-merge of the tag:

```
Summary of actions:
- Release branch 'release/1.0.0' has been merged into 'main'
```

```
- The release was tagged 'v1.0.0'
- Release tag 'v1.0.0' has been back-merged into 'develop'
- Release branch 'release/1.0.0' has been locally deleted
- You are now on branch 'develop'
```

This is also reflected in the commands you ran:

```
git checkout main
...
git merge --no-ff release/1.0.0
...
git tag -a v1.0.0
git checkout develop
...
git merge --no-ff v1.0.0  # instead of: git merge --no-ff main
...
git branch -d release/1.0.0
```

As a final verification, you'll check that both main and the v1.0.0 tag point to the same commit. Run the following commands to get the latest commit for the tag and for main:

```
git -P log --oneline -1 v1.0.0
git -P log --oneline -1 main
```

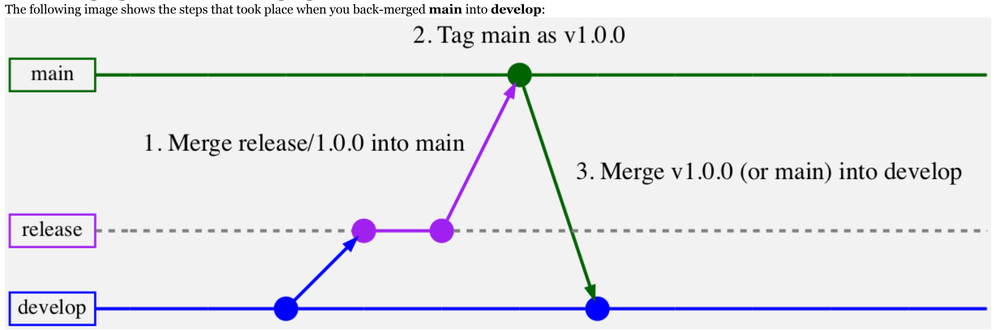
Note: The `-P` or `--no-pager` option disables the pager, so you don't need to press **q** to quit it, and just prints the output directly to the console.
You'll see the same message and commit hash for both, something similar to:

```
b01405c (tag: v1.0.0, main) Merge branch 'release/1.0.0'
```

Next, you'll learn about some of the differences between back-merging main into develop versus merging the release branch into develop.

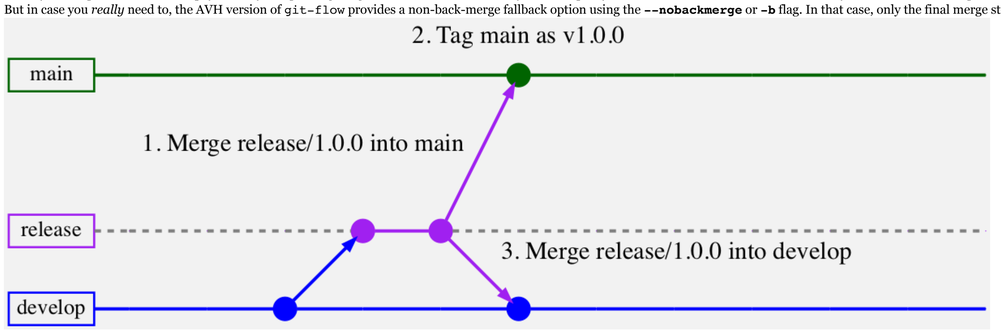
Back-merging main versus merging release

The following image shows the steps that took place when you back-merged **main** into **develop**:



Back-merging main into develop

When you merge main into develop, you really merge *everything* on main that wasn't already in develop. If you have additional commits on main that you *never* want to merge into develop — which you really shouldn't — then you can't use this strategy.
But in case you *really* need to, the AVH version of `git-flow` provides a non-back-merge fallback option using the **`--nobackmerge`** or **`-b`** flag. In that case, only the final merge step will be different, which you can see in the following image:



Merging the release branch into develop

One additional point to note is that with the `--nobackmerge` option, the tagged commit on main is not an ancestor of the commit that's merged into develop. This can create issues with commands like `git describe`, which finds the most recent tag reachable from a commit.
With the back-merge option, the commit tagged with `v1.0.0` is a parent of the merged commit on develop. This means `git describe` will be able to report the most recent tag as being `v1.0.0`, even when run from the develop branch.
With the `--nobackmerge` option, however, the tagged commit is not an ancestor of the merged commit on develop. `git describe` would only be able to find the `v1.0.0` tag when you run it from the **main** branch, but not from the **develop** branch.
In the next section, you'll finish a hotfix branch using `--nobackmerge` to see this difference with the output of `git describe`.

Creating and merging a hotfix branch

Since **hotfix** branches are used to fix bugs in production, you must create them from the **main** branch.
Even though the bug is also likely to be in the develop branch, you don't want to branch from develop to fix it since deploying that would prematurely deploy any additional code committed to develop since the last release.
You need to merge **hotfix** branches to both **main** and **develop** (or via back-merge from main). Like release branches, you name them with a version number, which git-flow will also use to tag the merge to main.
In a sense, hotfix branches are almost exactly like release branches, except they're created from **main** instead of **develop**.
So it turns out that changing the color from **navy** to **blue** was a mistake and should have instead been changed to **midnightblue**. This is an urgent fix that needs to be deployed immediately, and cannot wait to be included in the next release!
Run the following git-flow command to start a hotfix branch:

```
git flow hotfix start 1.0.1 --showcommands
```

You'll see that the command that actually executes is simply the following:

```
git checkout -b hotfix/1.0.1 main
```

Note that you didn't have to make sure you were on a specific branch before running the command. Are you starting to see the advantages of Gitflow? :]
Now, open **style.css** and change the color of the **h1** tag from **blue** to **midnightblue**:

```
 h1 {
-    color: blue;
+    color: midnightblue;
 }
```

Run the following command to commit the change:

```
git commit -am "Updated h1 color from blue to midnightblue"
```

Additionally, update the version number in the **VERSION** file to **1.0.1**. You can either edit the file manually or simply run the following command:

```
echo '1.0.1' > VERSION
```

Then commit the version number update:

```
git commit -am "Updated VERSION to 1.0.1"
```

Now, you could run `git flow hotfix finish`, or simply `git flow finish`, to merge the hotfix branch you're on. However, this time you'll use the classic behavior of merging the **hotfix** branch into **develop** by using the `--nobackmerge` option.

```
git flow finish --nobackmerge --showcommands
```

Again, type **:wq** to accept the initial message for the merge to main and add **Tag for 1.0.1 release** for the tag message. Type **:wq** and then type **:wq** one final time to accept the message for the merge of the `hotfix/1.0.1` branch to develop.
You'll see the following **Summary of actions**:

```
Summary of actions:
- Hotfix branch 'hotfix/1.0.1' has been merged into 'main'
- The hotfix was tagged 'v1.0.1'
- Hotfix branch 'hotfix/1.0.1' has been merged into 'develop'
- Hotfix branch 'hotfix/1.0.1' has been locally deleted
- You are now on branch 'develop'
```

Without using the `--nobackmerge` option, the third line would have said:

```
- Hotfix tag 'v1.0.1' has been back-merged into 'develop'
```

And you'll see this reflected in the commands as well:

```
git checkout main
...
git merge --no-ff hotfix/1.0.1
...
git tag -a v1.0.1
git checkout develop
...
git merge --no-ff hotfix/v1.0.1  # not: git merge --no-ff v1.0.1
...
git branch -d hotfix/1.0.1
```

That's all there is to Gitflow! You've successfully used git-flow to adopt the Gitflow workflow. With it, you've created and merged a feature branch, a release branch, and a hotfix branch.
Now you'll cover some final details about how `git describe` won't be accurate when using the classic approach with the `--nobackmerge` option. You'll also learn a bit about using help with `git-flow`, and then you'll be done with this chapter!

Using git describe

The `git describe` command shows you the most recent tag that's accessible from a commit. If the tag is on the current commit, `git describe` will show the tag itself. On the other hand, if the tag is on one of the ancestors, it will also show the number of additional commits and the commit hash in the following format:

```
{tag}-{number-of-additional-commits-from-tag}-g{commit hash}
```

Run **`git describe develop`** and you'll see something like the following, just with a different hash:

```
v1.0.0-4-g19c9939
```

Now run **`git log --oneline -5 develop`** to get the five latest commits on develop:

```
19c9939 (HEAD -> develop) Merge branch 'hotfix/1.0.1' into develop
d7bd387 Updated VERSION to 1.0.1
e4648d7 Updated h1 color from blue to midnightblue
10bc940 Merge tag 'v1.0.0' into develop
b01405c (tag: v1.0.0) Merge branch 'release/1.0.0'
```

In the output of `git describe`, **v1.0.0** is the tag, **4** is the number of additional commits after the tag, and **827ddd8** is the commit hash of the commit you ran `git describe` on — that is, **develop**.
This is misleading since the current version is 1.0.1, and the `git log` command above shows that the commit that updates the **VERSION** file to 1.0.1 is included in the develop branch.
The reason it even shows `v1.0.0` is because you back-merged the `v1.0.0` tag from main to develop, so develop can access it. If it hadn't been back-merged, it would just show an error.
You can replicate the error by running `git describe origin/main` since there are no tags accessible from the `origin/main` commit:

```
fatal: No tags can describe 'c0623652f3f7979f664918689fca42e9...
```

Now, run `git describe main`, and you'll see the following:

```
v1.0.1
```

When the reference you used points to the same commit as the tag, running `git describe` prints the tag without the additional info.
Note: You can suppress the additional info by using the **`--abbrev=0`** option. In that case, running `git describe develop --abbrev=0` would just show `v1.0.0`.
So one benefit of using the back-merge strategy to merge **main** into **develop** is that any tags on the **main** branch will also be accessible from the **develop** branch. That means you can run a `git describe` while on the **develop** branch to show the tag for the latest release.

Thus far, you've only used the most commonly used commands from `git-flow`. Next, you'll learn how to explore the various commands `git-flow` provides and the different options that can be used with each command.

Exploring the git-flow library

The `git-flow` library includes a few additional commands that can be helpful, such as `delete` for deleting a type of branch or `publish` for pushing it to the remote.
To see all subcommands run **`git flow help`**. You'll see the following:

```
$ git flow help
usage: git flow <subcommand>

Available subcommands are:
   init      Initialize a new git repo with support for the b...
   feature   Manage your feature branches.
   bugfix    Manage your bugfix branches.
   release   Manage your release branches.
   hotfix    Manage your hotfix branches.
   support   Manage your support branches.
   version   Shows version information.
   config    Manage your git-flow configuration.
   log       Show log deviating from base branch.

Try 'git flow <subcommand> help' for details.
```

Next, use **help** with a specific type of subcommand. Run **`git flow release help`** to see the types of subcommands available for release branches:

```
$ git flow release help
usage: git flow release [list]
   or: git flow release start
   or: git flow release finish
   or: git flow release publish
   or: git flow release track
   or: git flow release delete

    Manage your release branches.

    For more specific help type the command followed by --help
```

Next, use `--help` or `-h` with the specific type of sub-subcommand to see a description of what it does. You need the dashes; otherwise, it would use **help** as the branch name. For example, run **`git flow release publish --help`** and you'll see the following:

```
$ git flow release publish --help
usage: git flow release publish [-h] <name>

    Publish the release branch <name> on origin

    -h, --help            Show this help
    --showcommands        Show git commands while executing them
```

And you're done! You've not only learned about the Gitflow workflow but also how to use and explore the various commands in the `git-flow` library.

Key points

The main branch serves as the production branch.
The develop branch is for normal day-to-day development.
Feature branches are used for new feature development.
Release branches are used to test, stabilize and deploy a release to production.
Hotfix branches are used to fix bugs you've already released to production.
Feature branches are created from develop and merged back into to develop.
Release branches are created from develop and merged into both main and develop.
Hotfix branches are created from main and are also merged into both main and develop.
Install the newer AVH version of git-flow with `brew install git-flow-avh`.

That's Gitflow for you! In the next chapter, you'll learn about the Forking Workflow.

11. Forking Workflow 9. Feature Branch Workflow

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.