

|  |
| --- |
|  |

**Git Flow Implementation**

Document Control

|  |  |
| --- | --- |
| **Title**: | **Git Flow Implementation** |

|  |  |  |
| --- | --- | --- |
| **Revision history**: | | |
| ***Version No.*** | ***Release Date*** | ***Author*** |
| 0.1 | 05-July-2016 | Saikrishna Gogula |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| **Significant changes**: | |
| ***Version No.*** | ***Details*** |
| 0.1 | Initial Draft Version |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| **Availability**: | |
| ***Copy*** | ***Location*** |
| Soft copy |  |
| Hard copy |  |

|  |  |
| --- | --- |
| **General**: | |
| **Approved by**: |  |
| **Released by**: |  |

|  |
| --- |
| **Distribution list**: |

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approver/Reviewer** |
| Chandra Allaka | Strategy Architect | Reviewer |
| Daniel Lane | Chief Engineer | Approver |
| Mikkal Sveum | Portfolio Testing Infrastructure Manager | Approver |
| Sunil Kothyala | Senior Manager | Approver |

**Table of Contents**

Table of Contents

[1 Overview 5](#_Toc455500867)

[2 Purpose 6](#_Toc455500868)

[3 Git-Flow Setup 7](#_Toc455500869)

[4 Git Flow Workflow 7](#_Toc455500870)

[5 Implementation 8](#_Toc455500871)

[5.1 Feature Branches 9](#_Toc455500872)

[5.2 Release Branches 9](#_Toc455500873)

[5.3 Maintenance Branches 10](#_Toc455500874)

[5.4 Support Development 11](#_Toc455500875)

[6 Advantages and Disadvantages 13](#_Toc455500876)

# Overview

Git-Flow is a Workflow for Development practice proposed by Vincent Driessen which has since been implemented in a number of variations. Git-Flow is a workflow for branching that can be used for doing development work.

# Purpose

The purpose of this document is to provide details of Git-Flow implementation.

# Git-Flow Setup

1. Download the Git-Flow installer with below command:

wget -q – <http://github.com/nvie/gitflow/raw/develop/contrib/gitflow-installer.sh> –no-check-certificate

Above command will download the shell script to install Git-Flow.

1. Execute the below command to make the script executable:

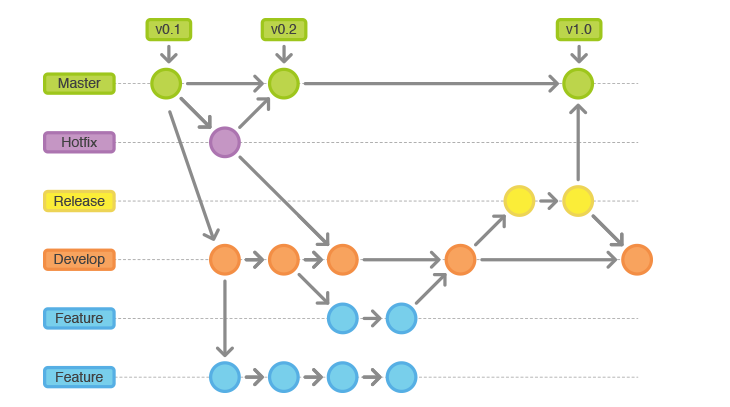
**sudo chmod a+x** gitflow-installer.sh

1. Execute the below command to install Git-Flow:

sudo ./gitflow-installer.sh

# Git Flow Workflow

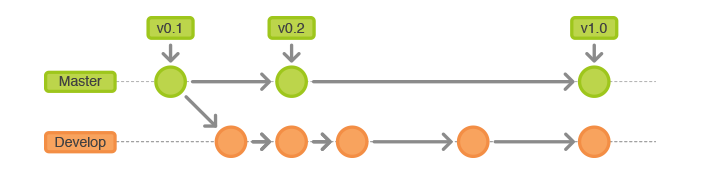
The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Please find the below flow diagram:



It assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases.

# Implementation

In Git-Flow everything is broken down into branches.  When you do new feature development, you create a new branch off the develop branch.  If you’re working on a hotfix, then you branch off master and if you’re busy with release hardening then you branch of the develop branch. Please find the below diagram which shows two main branches of Git-Flow:



In Git-Flow there are two main branches that you should try and keeps as clean as possible and always in a releasable state.  The one is master which can be seen as your released code base i.e. what you currently have running in Production.  On the other hand you have the develop branch which can be seen as the integration branch of code that consists of completed features and hotfixes that are stable and that should consist of complete unit test suites.  Your inter-team as well as support developers will integrate in this branch and you Continuous Integration server will also run against this branch to do your daily builds, run your complete test suites as well as generate test reports such as code coverage and release notes from.

The first step is to complement the default master 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 using below commands:

git branch develop

git push -u origin develop

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

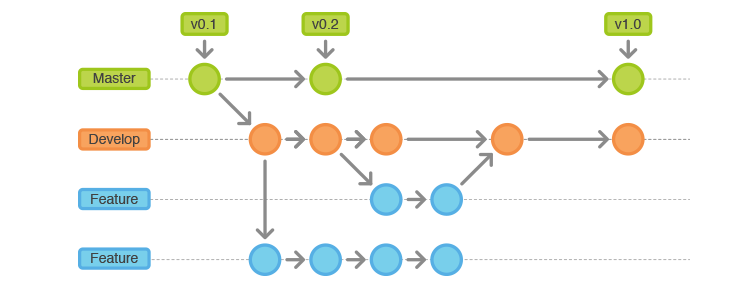
git clone ssh://user@host/path/to/repo.git

git checkout -b develop origin/develop

Everybody now has a local copy of the historical branches set up.

## Feature Branches

Each new feature should reside in its own branch, which can be [pushed to the central repository for backup/collaboration](https://www.atlassian.com/git/tutorials/syncing/git-push).



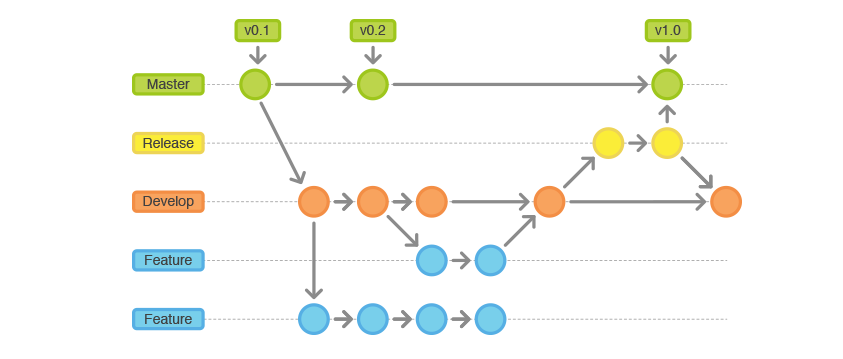
But, instead of branching off of master, feature branches use develop as their parent branch. When a feature is complete, it gets [merged back into develop](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with master.

You need to create a separate branch for feature If you are working on a new feature, Instead of basing it on master, they should both base their feature branches on develop which can be done using below command:

git checkout -b some-feature develop

## 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 these point-only bug fixes, documentation generation, and other release-oriented tasks should go in this branch.



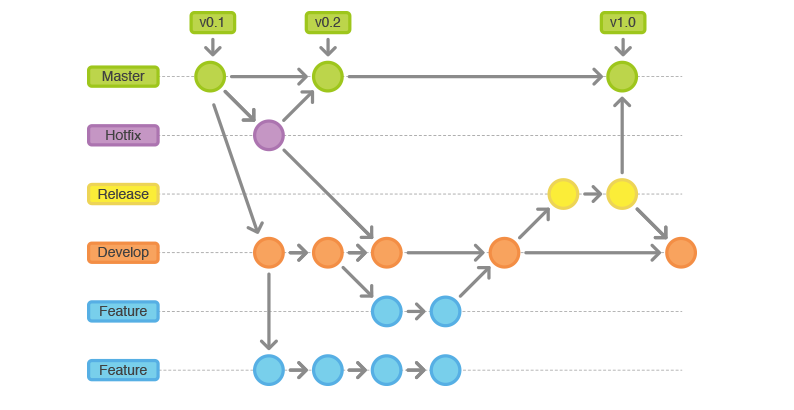
Once it’s ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated. Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development.

You need to create new branch to encapsulate the release preparations once you are done with feature and ready to ship. This step is also where the release’s version number is established which can be done using below command:

git checkout -b release-0.1.0 develop

## Maintenance Branches

Maintenance or “hotfix” branches are used to quickly patch production releases. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master 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 master.

If end-user opens a ticket complaining about a bug in the current release. To address the bug, you need to create a maintenance branch off of master, fixe the issue with as many commits as necessary, then merge it directly back into master which can be done using the below commands:

git checkout -b hotfix-0.1.1 master

# Optional: Bump version number, commit

# Fix the bug, commit

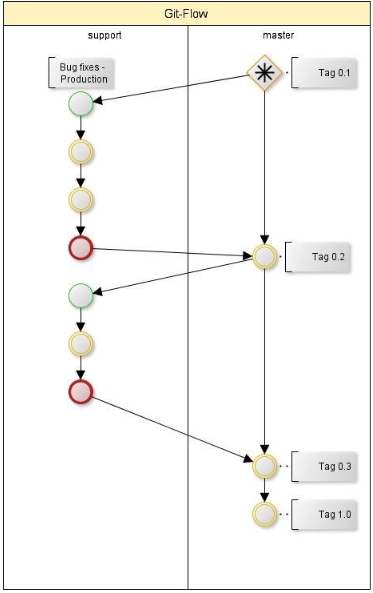
git checkout master

git merge --no-ff hotfix-0.1.1

git push

## Support Development

This support branch is in fact a special kind of branch that can be created of a specific Tag in the master branch. So, the idea is that you’ll treat it the same as a hotfix branch where you’ll fix issue and then merge it back into master only.



It’s a bit strange to only merge back into the master branch and not the develop branch, but there is a good reason.  If your most recent release is on version 10.2 and you’re still doing Support to some big client on version 4.3 then you may not want to merge the changes you’re making with the latest develop branch seeing that the code base and architecture might have changed completely.  In these cases you’ll do the fix on this special branch, push it through the UAT process and then do a Release of the software to that specific client after tagging it in the same manner as you would a hotfix.

You need to create the new branch from specific tag of a master branch and merge the changes only to the master branch using the below command:

git checkout -b support-0.1.1 master

# Optional: Bump version number, commit

# Fix the bug, commit

git checkout master

git merge --no-ff support-0.1.1

git push

# Advantages and Disadvantages

**Advantages:**

* **Parallel Development**: One of the great things about GitFlow is that it makes parallel development very easy, by isolating new development from finished work. New development (such as features and non-emergency bug fixes) is done in **feature branches**, and is only merged back into main body of code when the developer is happy that the code is ready for release.
* **Collaboration:** Feature branches also make it easier for two or more developers to collaborate on the same feature, because each feature branch is a sandbox where the only changes are the changes necessary to get the new feature working. That makes it very easy to see and follow what each collaborator is doing.
* **Release Staging Area:** As new development is completed, it gets merged back into the **develop branch**, which is a staging area for all completed features that haven’t yet been released. So when the next release is branched off of **develop**, it will automatically contain all of the new stuff that has been finished.
* **Support for Emergency Fixes:** GitFlow supports **hotfix branches** - branches made from a tagged release. You can use these to make an emergency change, safe in the knowledge that the hotfix will only contain your emergency fix. There’s no risk that you’ll accidentally merge in new development at the same time.
* **Multiple versions in production:** If your code is having multiple versions in production (i.e. typical software products like Operating Systems, Office Packages, Custom applications, etc.) you may use git-flow. Main reason is that you need to continuously support previous versions in production while developing the next version.

**Disadvantages:**

* Complex to roll-back individual features once they have been released, forced to quickly create and deploy a “hotfix”.
* Large merges into master making it difficult to visualize how a project is changing overtime using only the git history. Becomes necessary to maintain additional documentation.
* Can feed into a perception that long manual testing cycles and/or release processes always result in fewer bugs, which is a fallacy.