DEVOPS – CI/CD

23-11-2020 to 27-11-2020

Venkatesh Reddy Madduri && ShankarPrasad

Agenda

- Introduction
- **Git & GitHub**
- Continuous Integration using Jenkins
- Introduction to Docker
- Kubernetes
- Ansible
- Continuous Monitoring using Nagios
- DevOps Project

What is Git

Git is a free, opensource distributed version control system tool designed to handle everything from small to very large projects with speed and efficiency.

It was created by Linus Torvalds in 2005 to develop Linux Kernel.

Git has the functionality, performance, security and flexibility that most teams and individual developers need.

It also serves as an important distributed version-control.

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software. Before you go ahead, check out this video on GIT which will give you better in-sight.

Purpose of Git & Version Control System

What is the purpose of Git?

Git is primarily used to manage your project, comprising a set of code/text files that may change.

But before we go further, let us take a step back to learn all about Version Control Systems (VCS) and how Git came into existence.

Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

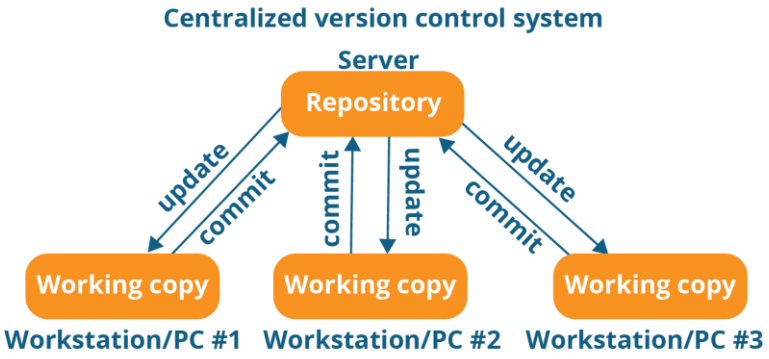
There are two types of VCS:

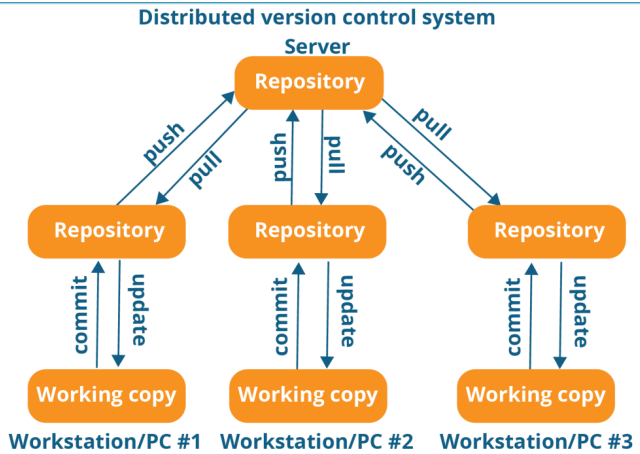
Centralized Version Control System (CVCS)
Distributed Version Control System (DVCS)

Centralized/Distributed VCS

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.

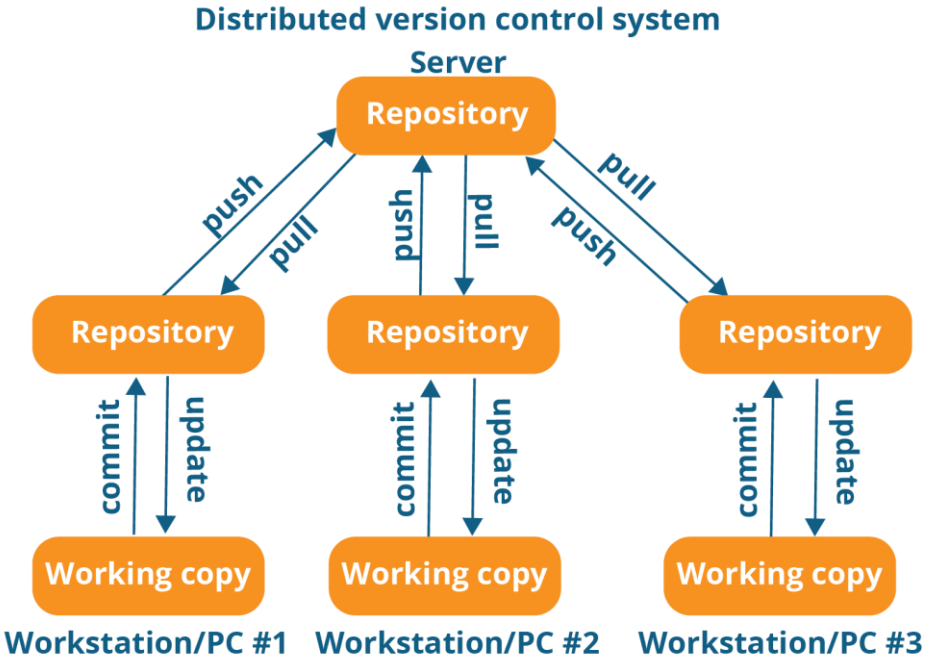
Distributed VCS These systems do not necessarily rely on a central server to store all the versions of a project file. In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.





Distributed VCS

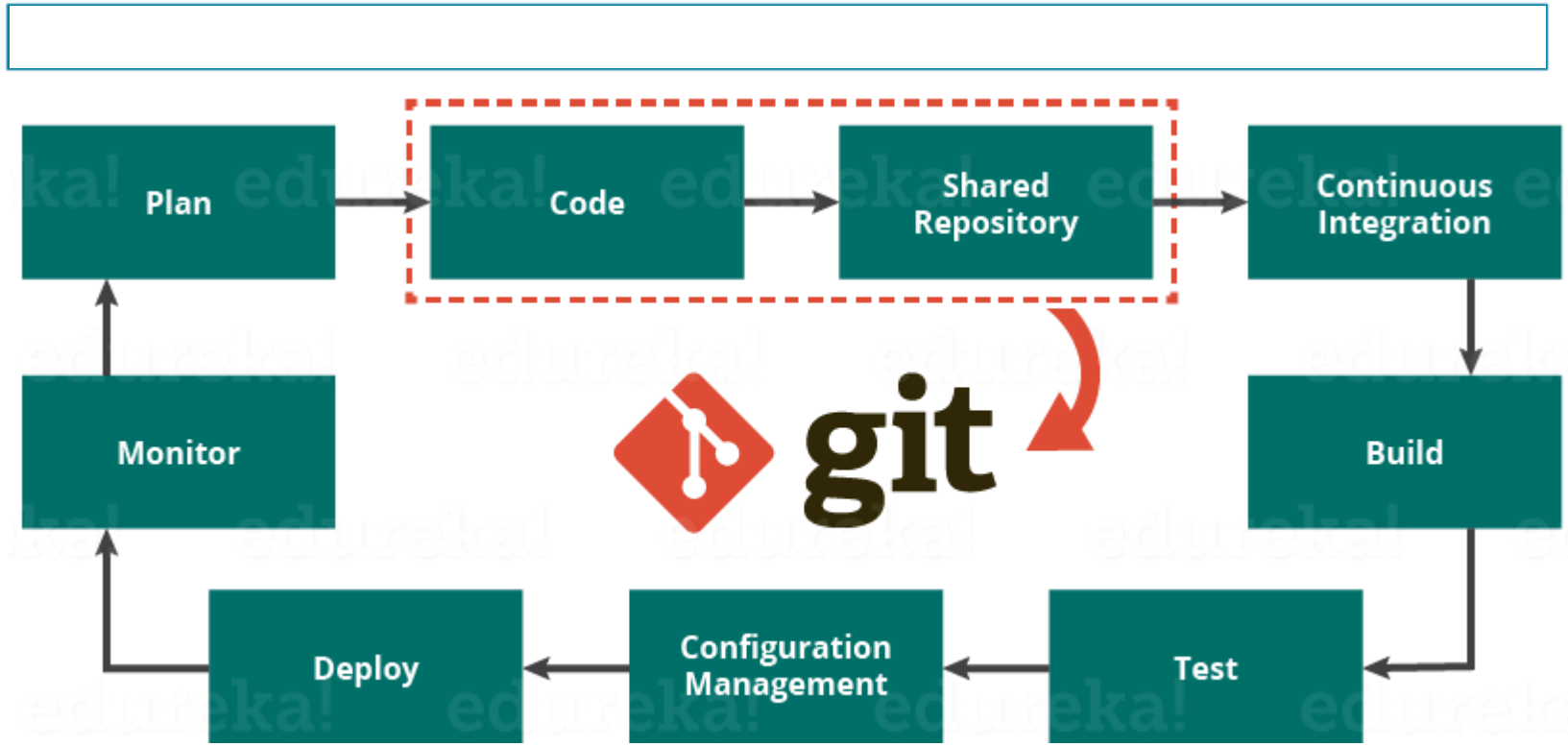
These systems do not necessarily rely on a central server to store all the versions of a project file. In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.



Features Of Git

Free and open source -
Speed -
Scalable - The number of collaborators increase Git can easily handle this change.
Reliable - On the events of a system crash, the lost data can be recovered from any of the local repositories
Easy Branching -
Distributed development - Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch
Compatibility with existing systems or protocol - Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

Role Of Git In DevOps



Git - Installation

sudo su –

git - - version

yum install git

Create a account in https://github.com/

```
[root@vlmazjuly199 ~]#
[root@vlmazjuly199 ~]# yum install git
Jenkins-stable                                                                                                  6.0 kB/s | 2.9 kB     00:00
Red Hat Enterprise Linux 8 for x86_64 - BaseOS - Extended Update Support from RHUI (RPMs)                        13 kB/s | 2.4 kB     00:00
Red Hat Enterprise Linux 8 for x86_64 - AppStream - Extended Update Support from RHUI (RPMs)                     6.9 kB/s | 2.8 kB     00:00
Microsoft Azure RPMs for RHEL8 Extended Update Support                                                           47 kB/s | 2.1 kB     00:00
Dependencies resolved.
================================================================================================================================================
 Package                         Architecture             Version                              Repository                                  Size
================================================================================================================================================
Installing:
 git                             x86_64                   2.18.4-1.el8_1                       rhel-8-for-x86_64-appstream-eus-rhui-rpms   187 k
Installing dependencies:
 perl-Error                      noarch                   1:0.17025-2.el8                      rhel-8-for-x86_64-appstream-eus-rhui-rpms    46 k
 perl-Git                        noarch                   2.18.4-1.el8_1                       rhel-8-for-x86_64-appstream-eus-rhui-rpms    77 k
 git-core                        x86_64                   2.18.4-1.el8_1                       rhel-8-for-x86_64-appstream-eus-rhui-rpms   5.0 M
 git-core-doc                    noarch                   2.18.4-1.el8_1                       rhel-8-for-x86_64-appstream-eus-rhui-rpms   2.3 M

Transaction Summary
================================================================================================================================================
Install  5 Packages

Total download size: 7.6 M
Installed size: 42 M
Is this ok [y/N]:
```

Git - Installation

```
Total download size: 7.6 M
Installed size: 42 M
Is this ok [y/N]: y
Downloading Packages:
(1/5): perl-Error-0.17025-2.el8.noarch.rpm                                  97 kB/s |  46 kB     00:00
(2/5): perl-Git-2.18.4-1.el8_1.noarch.rpm                                  153 kB/s |  77 kB     00:00
(3/5): git-2.18.4-1.el8_1.x86_64.rpm                                       367 kB/s | 187 kB     00:00
(4/5): git-core-doc-2.18.4-1.el8_1.noarch.rpm                               15 MB/s | 2.3 MB     00:00
(5/5): git-core-2.18.4-1.el8_1.x86_64.rpm                                   25 MB/s | 5.0 MB     00:00
--------------------------------------------------------------------------------------------------------------------------------------------------------
Total                                                                       11 MB/s | 7.6 MB     00:00
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :                                                                                                                              1/1
  Installing       : git-core-2.18.4-1.el8_1.x86_64                                                                                               1/5
  Installing       : git-core-doc-2.18.4-1.el8_1.noarch                                                                                           2/5
  Installing       : perl-Error-1:0.17025-2.el8.noarch                                                                                            3/5
  Installing       : git-2.18.4-1.el8_1.x86_64                                                                                                    4/5
  Installing       : perl-Git-2.18.4-1.el8_1.noarch                                                                                               5/5
  Running scriptlet: perl-Git-2.18.4-1.el8_1.noarch                                                                                               5/5
  Verifying        : perl-Error-1:0.17025-2.el8.noarch                                                                                            1/5
  Verifying        : perl-Git-2.18.4-1.el8_1.noarch                                                                                               2/5
  Verifying        : git-2.18.4-1.el8_1.x86_64                                                                                                    3/5
  Verifying        : git-core-2.18.4-1.el8_1.x86_64                                                                                               4/5
  Verifying        : git-core-doc-2.18.4-1.el8_1.noarch                                                                                           5/5
Installed products updated.

Installed:
  git-2.18.4-1.el8_1.x86_64   perl-Error-1:0.17025-2.el8.noarch   perl-Git-2.18.4-1.el8_1.noarch   git-core-2.18.4-1.el8_1.x86_64   git-core-doc-2.18.4-1.el8_1.noarch

Complete!
[root@vlmazjuly199 ~]#
```

Git - Commands

sudo su –

git - - version

yum install git

Create a account in https://github.com/

git config --global user.name shankarnj
git config --global user.email shankarnj@yahoo.com

Git - Commands

```
yum install git

git config --list

git config --global user.name shankarnj
git config --global user.email shankarnj@yahoo.com

git config --global --unset user.name
git config --global --unset user.email

git commit File1.txt

git commit -m "Pushed Updated File1.txt"
```

Git - Commands

```
git log

---- Check the current branch
git branch



---- add branch project1_b1
git branch project1_b1

---- add branch project1_b2
git branch project1_b2

---- add branch "project1_b3" and switch to "project1_b3"
git checkout -b project1_b3

---- create file_3.txt in b3 branch
echo 'New file in project1_b3 branch' > file_3.txt
```

Git - Commands

---- add file_3.txt in b3 branch at local repository
git add .

---- Commit the changes to local repository at branch project1_b3

git commit -m "commit the file file_3.txt in project1_b3 branch"

-- Switch to master

git checkout master

-- Verify the contents of master branch, you should see only master branch files

ls -ltr

Git - Commands

git remote add origin https://github.com/shankarnj/Project1.git

git remote -v

--To push the current branch and set the remote as upstream, use

    git push origin master

Git - Commands

```
 git push origin project1_b3

--Create a file "File3.txt"

git add File3.txt

git push origin master

git push origin master

git config --get remote.origin.url
```

Git - SSh

Many Git servers authenticate using SSH public keys. In order to provide a public key, each user in your system must generate one if they don't already have one. This process is similar across all operating systems. For more information, please check below link.
https://confluence.atlassian.com/bitbucketserver045/using-bitbucket-server/controlling-access-to-code/using-ssh-keys-to-secure-git-operations/creating-ssh-keys

```
$ ssh-keygen -o
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 schacon@mylaptop.local
```

Get the key

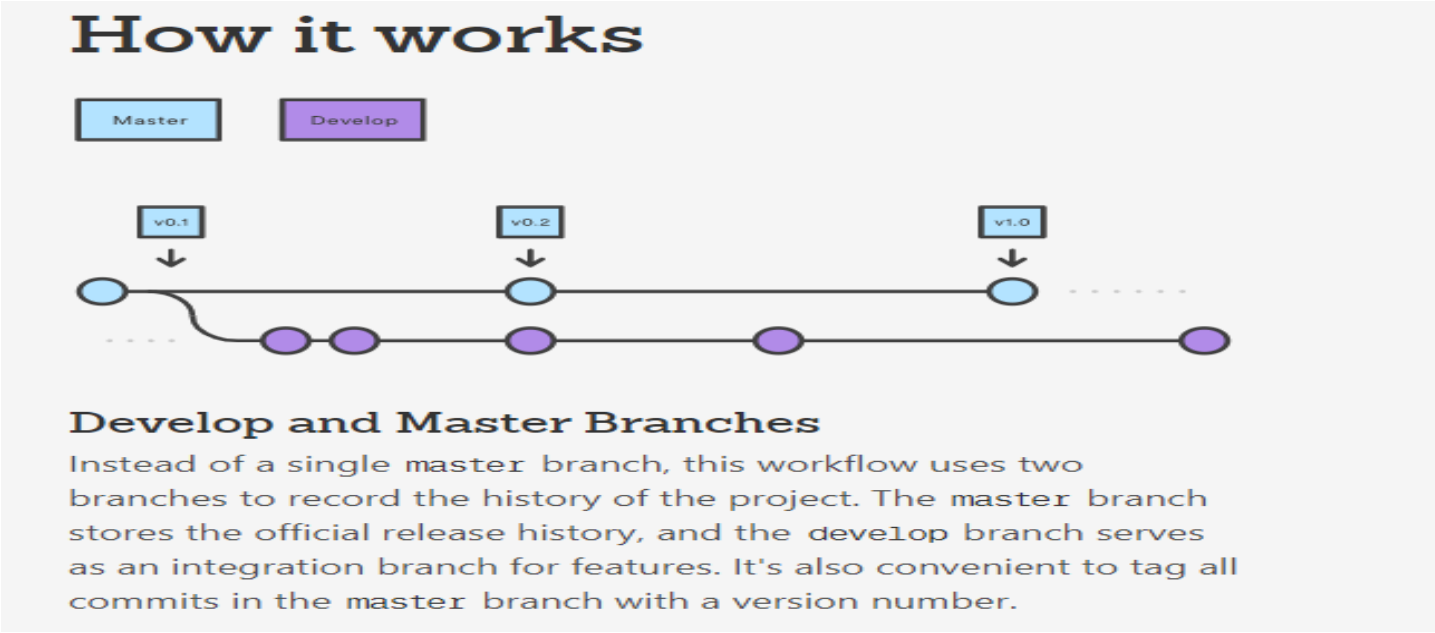
```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Git - SSh

1. Login into github.com
2. Click Profile and Settings
3. Click Add SSH and GPG Keys and the Key.
4. Try add new file and try to push and pull with your project.

Git Flow

Gitflow Workflow is a Git workflow that helps with continuous software development and implementing DevOps practices. It was first published and made popular by Vincent Driessen at nvie. The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.



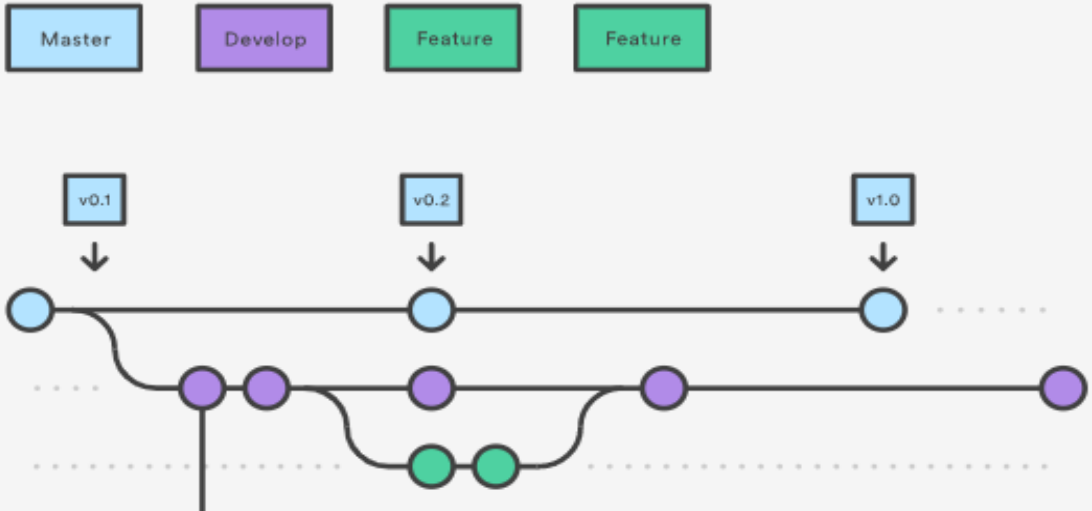
Git Flow

Gitflow Workflow is a Git workflow that helps with continuous software development and implementing DevOps practices. It was first published and made popular by Vincent Driessen at nvie. The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.

```
$ git flow init

Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created r
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []

$ git branch
* develop
  master
```

Feature Branches

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 `master`, `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 `master`.



Creating a feature branch

Without the git-flow extensions:

```
git checkout develop
git checkout -b feature_branch
```

When using the git-flow extension:

```
git flow feature start feature_branch
```

Continue your work and use Git like you normally would.

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`.

Without the git-flow extensions:

```
git checkout develop
git merge feature_branch
```

Using the git-flow extensions:

```
git flow feature finish feature_branch
```

Releasing branch

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 `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 (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.

Without the git-flow extensions:

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

When using the git-flow extensions:

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```

Once the release is ready to ship, it will get merged it into `master` 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 organization stresses code review, this would be an ideal place for a pull request.

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

Without the git-flow extensions:

```
git checkout master
git merge release/0.1.0
```

Or with the git-flow extension:

```
git flow release finish '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 master instead of develop. 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. A hotfix branch can be created using the following methods:

Hotfix Branches

Without the git-flow extensions:

```
git checkout master
git checkout -b hotfix_branch
```

When using the git-flow extensions:

```
$ git flow hotfix start hotfix_branch
```

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

```
git checkout master
git merge hotfix_branch
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

FIS

Hotfix Branches

Without the git-flow extensions:

```
git checkout master
git checkout -b hotfix_branch
```

When using the git-flow extensions:

```
$ git flow hotfix start hotfix_branch
```

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

```
git checkout master
git merge hotfix_branch
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

Hotfix Branches

Example

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

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

FIS

Git rebase and merge operations

Thank you