BIG LITTLE BOOK ON

GIT

GITLAB, GITHUB, GITOPS
DEVOPS & CICD

U V OMOS

KNOWLEDGE REFRESH IN MINUTES NOT HOURS

Big Little Book on Git

U V Omos

Introduction

Hi there and welcome to the Big Little Book on Git. This book was written with both the new IT recruit and the seasoned IT professional in mind, but with a view to aid fast recall of information without having to lift a four-hundred-page tome! As well as condensing detailed technical facts, it is accompanied with useful hints and diagrams to help the concepts and understanding of the ideas stand out. Handy for those work or interview situations where there’s just not enough time to filter out the pertinent information from all that text. Take care and happy reading.

Git

Git is a Distributed Version Control System (DVCS) that can be used to manage code and other assets both locally or remotely. It avoids the issues associated with centralised version control systems, as developers have the ability to save branches of their repositories locally containing ***full*** change history and tracking. This avoids the inflexibility of a Centralised Version Control System (CVCS), which only saves the ***latest*** file snapshots locally and not the full project repository. A CVCS only stores all of the project repositories centrally, so a loss of data centrally means it cannot be recovered. This is where a DVCS had the advantage.

It is a content-addressable filesystem. Git gives developers the ability to write and update code in any programming language as well as documentation such as markdown diagrams and README documents in collaboration with colleagues. Changes are logged and can be tracked and regressed. ***Each change has a hash in SHA-1 format*** for verification and integrity-checking amongst other purposes.

Developed by the same team behind Linux, Git started out in 2005 when Linux's developers had to find an alternative to Bitkeeper (the version control system they were using then). This is because the relationship between them broke down and Bitkeeper started charging a licence fee for using their software.

It has gone on to now be amongst the most used platforms in its field, with GUI-based interfaces such as GitHub and Gitlab using it in the background.

It uses the concept of ***snapshots*** whereas other systems such as Subversion and Bazaar use the concept of differences and file-based changes (also known as delta-based). Git works by saving a picture of what all the files look like at a commit and storing a reference to this picture or snapshot. Unchanged files are not stored again as a link to the previous identical file stored is used. So, Git looks at this data as ***a stream of snapshots***. Summarising, Git uses a snapshot stream while other systems use file differences.

Data Integrity

All changes in Git are checksummed before they are stored and can be referenced by this checksum. So, all changes are known by Git and this is a core feature of the application. Therefore, data cannot be lost in transit or be altered without Git being aware of it. The mechanism used for this is the SHA-1 hash. A SHA-1 hash is a 40-character string of hexadecimal characters (0-9, a-f) and in Git this is calculated based on the file or directory contents. Git generally adds data to its database. All changes can typically be changed back to the previous version.

Git States

There are a number of states that Git files can be in and these are as follows.

- Modified
- Staged
- Committed

These are described below.

Modified

These are changed but not committed to the database yet. So, the files as they currently are i.e. without any new changes are considered modified files.

Staged

These are changed files in the current version that are marked for addition to the next commit snapshot. So, such files are those updated and included using the ‘git add’ command.

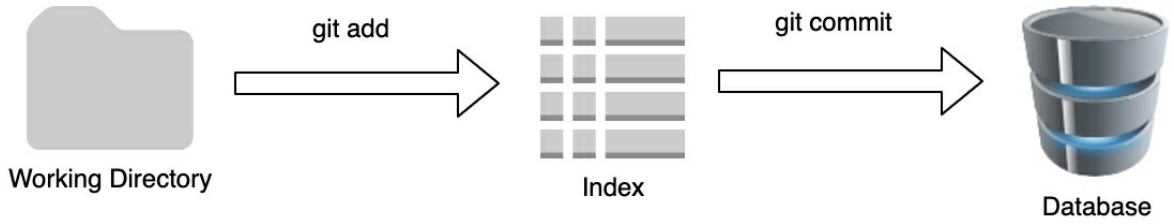
Committed

Committed changes are those that are now saved to the local database. These are any changes documents that are now ready for distribution by using the ‘git commit’ command.

Git Structure

Git has the following sections.

- Working Directory
- Index or Staging Area
- Object Database or .git



Basically, Git has a **mutable** Index and **immutable** Database. These are both linked to a Working Directory and are described below.

Working Directory

This is the modified state version of the repository that is checked out. These files are pulled out of the compressed Git directory database and saved on disk for modification locally.

Index or Staging Area

This is where the changes are initially saved in the staged state without being committed yet. It is also referred to as the Index.

Object Database or .git

Changes saved in the Staging Area are saved in the committed state. This is the local repository that gets pushed to the remote repository and is also called the Database.

This append-only database contains object information.

Git Workflow

The Staging Area or Index connects the Working Directory and the .git Database. Each of these sections works in a sequence that can be summarized below.

Working Directory –> (git add) -> Index –> (git commit) -> Object Database

The workflow would be as follows. Modify the file in the Working Directory (Working Tree). Stage just the changes needed using the **git add** command. Add those changes to the database using the **git commit** command. Store these changes in the remote repository using the **git push** command.

Referencing

A Git reference is a file that contains a Git commit SHA-1 hash. **All Git objects must be referenced.** Any non-referenced objects are automatically cleaned up by the Garbage Collector (this can also be done manually using the command line). An object can reference another object or reference. Git references can be created, deleted, updated or read from the reference database. There are a number of reference types as shown in the list that follows.

- Heads: this is a local object reference
- Remotes: this is an object in a remote location
- Stash: this is an uncommitted object
- Meta: this is a bare repository configuration
- Tags: these are the branch references described previously

As mentioned previously a branch is just a pointer or reference to the head of a line of work.

You can view references in the .git/refs/heads folder.

```
cd .git/refs/heads
ls
```

Let’s assume this showed the following output.

newdev

This is the name of the branch existing in the local repository. Let's use the git cat-file command to view the details of this branch.

git cat-file -p newdev

tree 528116019182f841f3a11f43938ef9fe1e685c39
parent 1ad98ed7c6338fcb265710806c36a3b893a3fea5
author Administrator <automation@vsphere.local> 1595769866 +0100
committer Administrator <automation@vsphere.local> 1595769866 +0100

updating newsites.yml with copy module

The SHA-1 hash of this branch's most recent commit can also be viewed with the cat command.

cat newdev

This would show the SHA-1 hash of the newdev commit similar to that shown below.

f0f6d77e96c26fc8a4ab30de55960c7a5a22fa04

This SHA-1 hash can then be used to view the latest commit history using the following command.

git log f0f6d77e96c26fc8a4ab30de55960c7a5a22fa04
commit f0f6d77e96c26fc8a4ab30de55960c7a5a22fa04 (HEAD -> newdev, origin/newdev)
Author: Administrator <automation@vsphere.local>
Date: Sun Jul 26 14:24:26 2020 +0100

    updating newsites.yml with copy module

commit 1ad98ed7c6338fcb265710806c36a3b893a3fea5 (origin/master)
Merge: 6911ca36 e8b2a223
Author: Administrator <admin@example.com>
Date: Sun Jul 26 12:12:31 2020 +0000

Merge branch 'newdev' into 'master'

updating newsites.yml with templates module

See merge request user/project!932

Database Objects

These are the objects stored in the immutable Database. These objects cannot be changed and any file changes are stored as new snapshots referenced by the particular commit. Git objects all have SHA-1 hashes of their contents for identification. These hashes are computed to obtain an object name and the first two characters of the hash used as ID.

The .git Database itself has the following objects.

- Blob
- Tree
- Commit
- Tag
- Packfile

These are described further below.

Blob

This is a binary large object and is a representation of a file that doesn't have a file name or a time stamp or metadata. The blob's hash is used as its name. These are saved when a file is stored in the repository and then added using the 'git add' command. These are stored in the .git/objects folder. They can be manually created using the following command.

git hash-object -w file.txt

This shows output similar to this (i.e. the SHA-1 hash of the file).

a58424524802425ad678968697679af183247890

This will save the contents of file.txt as a 40 character SHA-1 hash in the .git/objects folder. Note that the first two characters are the hash's folder and the other 38 characters the file as shown below.

echo 'this is a test' > file.txt
git hash-object -w file.txt

cd .git/objects
find . -type f

This shows output similar to this.

./a5/8424524802425ad678968697679af183247890

Note that the first two characters are the folder and the next 38 characters are the file.

You can actually view the contents of this file as follows.

git cat-file -p a58424524802425ad678968697679af183247890

This shows the following content of the file saved earlier.

this is a test

You could actually update this file's content to create and entirely new SHA-1 have of it. Remember, this is Git's function as a content-addressable version control system i.e. to save different file versions.

echo 'this is an update' > file.txt
git hash-object -w file.txt
cd .git/objects
find . -type f

This shows output similar to this.

./a5/8424524802425ad678968697679af183247890
./d7/586949469adcf486934dcaf768397483958375

This confirms there are now currently two hashes for the current file in the .git/objects folder.

Let's look at the contents of this new hash with the following command.

git cat-file -p d7586949469adcf486934dcaf768397483958375

This shows the following output.

this is an update

Typically, you won't use the git hash-object file and its use in these examples is just for illustration. These hash-generation actions are done in the background when you run the 'git add' command to save files to the repository Staging Area before they are committed.

Tree

This is analogous to a directory and contains files names with type bits that have references to the described blobs or tree objects that are files, symbolic links or directory contents.

This can be summarized as follows.

Tree contents: file list – blob reference – symbolic link / directory contents

Trees are also saved as 40-character SHA-1 hashes. Rather than having text content each tree contains the SHA-1 hashes of the files saved in the directory it references.

Git stores its content in a way that is similar to the Unix filesystem i.e. the use of blobs is similar to that of inodes used in the OS. A tree contains one or more blobs or trees itself. Each tree contains these hashes with their mode, type (of blob or tree) and filename similar to the following (using the command shown).

git cat-file -p master^{tree}

100644 blob a58424524802425ad678968697679af183247890 file.txt
100644 blob d7586949469adcf486934dcaf768397483958375 file.txt

A note on the file mode meanings.

- 100644 - normal file
- 100755 - executable
- 120000 - symbolic link
- 040000 - directory

Note that the same file has different hashes indicating different file versions.

Let's assume the tree itself was saved with the hash.

b831d9fb5961694a196e5bc1071e9f6f67f2cbd5

You can verify it as a tree with the following command (output shown as well).

git cat-file -t b831d9fb5961694a196e5bc1071e9f6f67f2cbd5
tree

You could view the tree contents (i.e. its list of files and folder as SHA-1 hashes) using the following command.

git cat-file -p b831d9fb5961694a196e5bc1071e9f6f67f2cbd5

100644 blob b4b1442c89d5776c1c10f7b920ecaf9a2d672f8d READM
100644 blob 5f1330b7b93c5dcaf43c120f945fe1d181032e28 instruct
100644 blob ebbd528ec97c1f9f89a67cf71f0d6dc1026dd5a8app.py
100644 blob 830e55561f0f9c1d0abd93edd3f027ca6ea6ec9ftest.pl
040000 tree f23125a2bc09774781d2413f0a8bd08a93270f50 lib
040000 tree 67ecbbf87af884d8b182b47cb8a885d208ddba9d var
040000 tree f 25fa46a7308f4938494850c4b581a4c4cb274ae2 www
040000 tree f e4a3ba29d6027d4312d4b5f02f6f85c50cdc0b98 opt

This shows that the tree itself contains four files and four folders. You can actually use a plumbing command (i.e. not for normal use) to create a tree as shown below.

git update-index –add –cacheinfo 100644
b831d9fb5961694a196e5bc1071e9f6f67f2cbd5

Note that this command is updating the Index or Staging Area i.e. this is a pre-commit staging command.

Commit

A commit connects tree objects in a history and contain the tree object's

name, timestamp, log message and commit objects. A commit object has the following contents.

- Commit SHA-1
- Author information
- Configuration, timestamp
- Blank line
- Commit message

An example of this from the ‘git log’ command output is shown below.

git log

commit 5cf7cef181cfaa7f00e2227eb098ab985a45f92f
Author: U V Omos <victor@gridlockaz.com>
Date: Wed May 29 13:23:16 2019 +0100

29052019 updated the add.py htmlchanges class

There could be a number of commits shown so you could filter down to a specific commit using the first few SHA-1 characters as follows.

git log –stat 5cf7ce

Note the use of the first six characters of the same SHA-1 hash for the earlier git log output. This would just show this commit’s output with changes as shown below using the command.

commit 5cf7cef181cfaa7f00e2227eb098ab985a45f92f
Author: U V Omos <victor@gridlockaz.com>
Date: Wed May 29 13:23:16 2019 +0100

29052019 updated the add.py htmlchanges class

controllers/template.pl | 251 ++++++++++++++++++---
controllers/crgenfind.pl | 2 -
controllers/generate.pl | 11 +-
controllers/ raiseticket.pl | 9 +-
data/sh-int.yml | 16 ++

docs/CRGen.txt | 133 +++++++++--------
models/gitloop.py | 14 +-
templates/password.yml | 2 +-

(git log output is truncated for illustrative purposes)

91 files changed, 1386 insertions(+), 571 deletions(-)

Note that this command will show all commit changes in the history of this particular tree i.e. there could be a few commit objects listed, each with a SHA-1 hash, author and so on.

Each committed object is zlib compressed with a header that uses the commit or tree as its content.

Tags

Tagging is a way of marking the changes made on a repository. These are similar to branch references but are fixed to a particular commit and can show specific sections in the history of a repository. Tags can be alpha-numeric and depend on the versioning used by your project. For instance, some projects might use ***Semantic Versioning or Semver*** to version their code. This entails the use of a MAJOR.MINOR.PATCH methodology with regards to the management of code. E.g. the first major release would be 1.0.0. The first patch of this major release would be 1.0.1 and so on.

Two types of tagging are available in Git as follows.

- Lightweight
- Annotated

These are described below.

Lightweight

Lightweight tags contain very little information. The actually only contain a checksum for the tag that can be used to verify the integrity of the repository.

A lightweight tag can be created with the following command.

git update-ref refs/tags/v1.0.1 b4b1442c89d5776c1c10f7b920ecaf9a2d672f8d

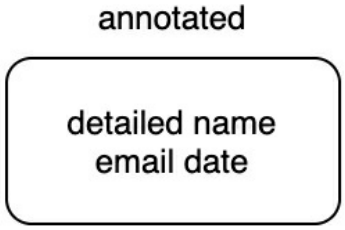
So, a lightweight tag ***is a reference that doesn't move*** and will stay fixed to its SHA-1 commit hash as configured.

This can be summarized as follows.

Tag → Commit

Annotated

Annotated tags contain a lot more information such as the name, email address and date of the actual tag.



Annotated tagging must be used for any versions of code merged to the master branch. This ensures the use of additional information as well as

checksums available in lightweight tagging (which does not include other information such as tag-specific author name and date information).

The following command creates an annotated tag testing-1.0.1 (using the -a option).

git tag -a testing-1.0.1 5f1330b7b93c5dcaf43c120f945fe1d181032e28 -m 'file tag'

Let's view the created tag.

cat .git/refs/tags/testing-1.0.1

This will show output similar to the following on screen.

31b3744ce818f807d3ad592f2851f31c3e43c39d

Let's now view the reference shown.

git cat-file -p 31b3744ce818f807d3ad592f2851f31c3e43c39d

This will show output similar to the following on screen.

object 5f1330b7b93c5dcaf43c120f945fe1d181032e28
type commit
tag testing-1.0.1
tagger Gridlockaz <info@gridlockaz.com> Sat May 23 16:48:58 2009 -0700

file tag

Note that the object SHA-1 is that used in the previous git tag command. So, the git cat-file printed out the tagged object.

This can be summarized as follows.

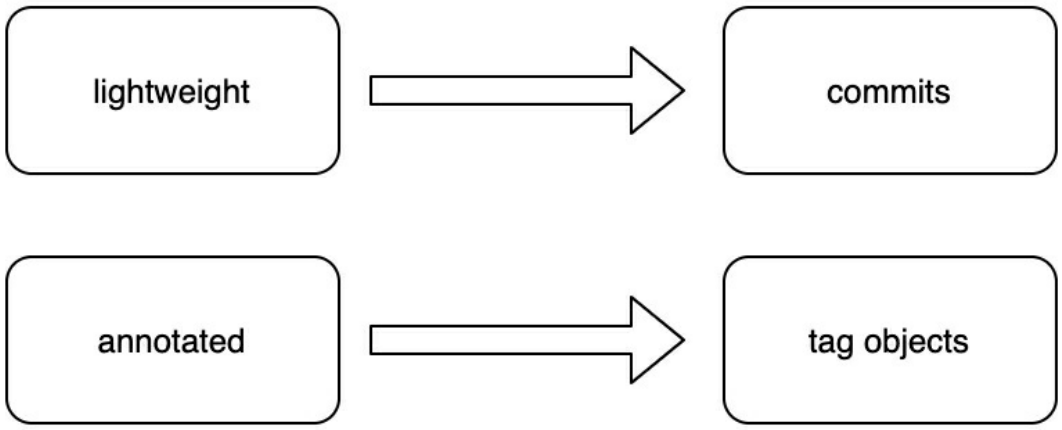
Reference → Created Tag Object (Contains Commit & Other Information) → Commit

A tag is a reference to a specific commit

It is advisable to check out a tagged repository to a branch, thus avoiding it

being in a 'detached HEAD' state (i.e. its changes cannot be merged back to the code's repository, as the new commit won't belong to a branch as will therefore be unreachable).

In summary, lightweight tags point to commits whilst annotated tags point to a tag object.



Packfile

This is a zlib compression of objects making it more efficient to transfer data over networks.

Plumbing Commands

These are low level commands that can be used to administer Git.

The standard commands used in Git are called porcelain commands.

The git init command creates the .git directory. The contents of this directory can be viewed as follows.

cd .git
ls

These contents are as follows or similar to the following depending on the Git version.

COMMIT_EDITMSG
FETCH_HEAD
HEAD
branches
config
description
hooks
index
info
logs
objects - stores the database contents
packed-refs
refs – this contains the commit pointers

Reference Database

Git stores labels called references or refs to indicate commit locations for indexing purposes. These are stored in the ref database and are as follows.

- Heads(branches)
- HEAD
- Tags

References are labels indicating commit locations

These are described further below.

Heads(branches)

These are specific named references that are automatically advanced to the new commit when a commit is made on top of them.

A branch is a Git reference that stores a new commit SHA-1 hash

HEAD

A reserved head that is checked against the working tree to create a commit.

HEAD is a reference to the latest commit reference and can be viewed using the following command.

cat .git/HEAD
ref: refs/heads/newdev

Changing the active branch using a command such as git checkout would change this reference as shown below.

git checkout newprod

Running the previous command would change the reference as follows.

cat .git/HEAD
ref: refs/heads/newprod

There are instances where a HEAD can point to a commit not a branch. This is known as a **detached HEAD** state. This is usually done if work needs to be done on a commit that is not at the tip of a branch. Let's say that the current branch is newdev. But you want to work on a commit tagged as testing-1.0.0 which is two commits behind newdev.

testing-1.0. 0 → testing-1.0.1 → newdev-0.0.1 → newdev **(HEAD)**

So, you would check out this commit as follows.

git checkout testing-1.0.0

This now puts the HEAD at testing-1.0.0.

testing-1.0.0 **(HEAD)** → testing-1.0.1 → newdev-0.0.1 → newdev

This is now in a **detached HEAD** state.

Tag

As described in a previous section, a tag object has references to other objects. As a container it can store metadata of other objects.

Git Branching

Branching is amongst the main features of Git and makes it powerful as a Version Control Software (VCS) platform. Git doesn't store its branches as differences but rather as snapshots. These snapshots have pointers. Git stores commits as objects. This can be summarized as follows.

Object > Pointer > Snapshot > Name, Email of Commit Author & Snapshot Pointers

Each file committed to the repository computes a SHA-1 hash known as a checksum and stores that file version in the repository as a **blob**.

A git branch is simply a pointer or reference to the head of a line of work.

The checksum is now added to the staging area.

All of this is done using the git commit command as shown in the following illustration.

git commit -m "adding html index file"

This checksums each directory and stores the repository as a **tree**, which has all the **metadata** and a pointer to the root so snapshots can be recreated if required.

The checksum is a means of assuring the integrity of all of the repository's contents.

So, five types of objects are present in the repository as follows.
1 x **Tree** showing contents and filenames of blobs
1 x **Commit** with pointer to root tree and commit metadata
Blobs (files)

A Git branch is basically a movable pointer to the repository's commits.

It is a file containing the 40-character SHA-1 hash of the commit it is pointing to.

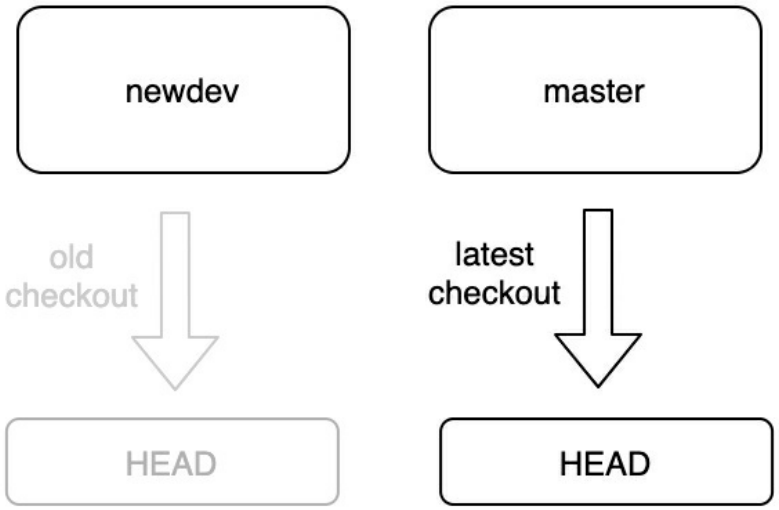
So, master is just a pointer to the latest commit and this pointer moves forward automatically. It is actually the same as any other branch and is just created by default using git init.

Branch <> Pointer

Creating a new branch is therefore creating a new pointer.

If the branch is checked out as well, HEAD now points to it.

HEAD moves with the checkout



Commands

You can review the current branches using the following command.

git branch

This will show a list of the current repository branches.

The currently checked out branch will be denoted with *

The last commit can be viewed with the following command

git branch -v

View those currently merged into the main branch as follows

git branch —merged

Any listed could be deleted if you have finished working on them

View branches that haven’t been merged yet with the following command

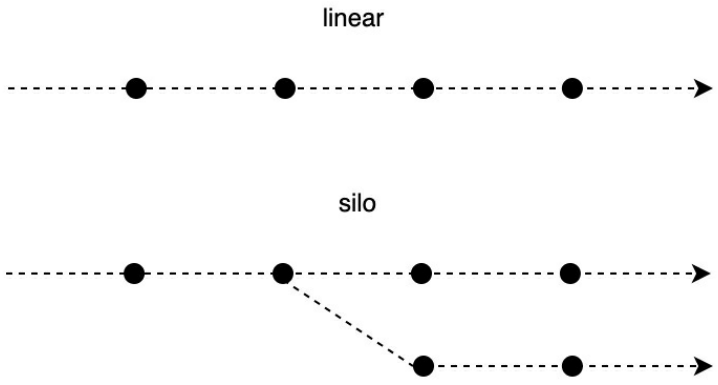
git branch —no-merged

These branches cannot be deleted until they have been merged into the folders

Branching Workflows

The following are ways of viewing the branching.

- Linear: all shown in a straight line
- Silo: branches diverge into lanes



Note that the above examples are for a local implementation of git by a single developer. If you are working with other developers, the distributed Git implementation is necessary. This is so that each developer’s changes are not overwritten by others. This is done by ensuring that only developers who have used Git to fetch the latest copy of the code can merge their changes into the repository.

So, the following process is recommended when working in a distributed Git environment.

1. Fetch
2. Create Branch
3. Code
4. Merge
5. Delete Branch

git fetch
git checkbout -b master

Update code

git add .

git -m commit "add a suitable commit message"

(As an alternative the *git -am "commit message"* command could be used for both the add and the commit commands).

git push origin master

This pushes the new changes to the master branch on the server.

Login to the server and using the GUI, request a Merge (which can be set to delete the newdev branch).

Typically, the official repository will require approval by a maintainer before accepting merges.

Including approvals, the process would be more like the following.

1. Fetch
2. Create Branch
3. Code
4. Push To Developer's Copy
5. Developer Request Pull By Maintainer
6. Code Pulled By Maintainer

7. Delete Branch

git fetch
git checkbout -b newdev

Update code

git add .

git -m commit "add a suitable commit message"

(As an alternative the *git -am "commit message"* command could be used for both the add and the commit commands).

git push origin newdev

This creates a new branch on the git server

Login to the server and using the GUI, request a Merge ***on approval by a suitable repository Maintainer*** (which can be set to delete the newdev branch).

Types of Git Branching (Workflows)

There are different types of branching or workflows used with the various Git platforms. These can be summarized as follows. A brief note on terminology. Merge Request (Gitlab) and Pull Request (GitHub, Bitbucket) mean the same thing. Also, not all pull requests need merging into branches. For instance, it is typical for developers to submit a merge or pull request without assigning it to a colleague (but mentioning them in the comment e.g. using the @user format so the user receives a notification), implying that this is more for review than actually changing the repository.

Git Flow

Summary: feature/release/hotfix branch > develop > master

This is the classic setup as originated by Git. It uses a develop branch as an

intermediary between developer owned feature, release or hotfix branches. All changes must be merged to the develop branch and this is tested in the test environment. Changes can only be merged to master on success. Master is the only branch that can be deployed to any live environment.

1. Developers update their local branches
2. These branches are committed/merged to the Develop branch
3. The Develop branch is deployed in test environment
4. Only successful branches' owners can send a Pull/Merge Request for addition to Master
5. Pull/Merge Request is granted

GitHub Flow

Summary: named branch > master

This advocates the deployment of the feature branch as proven before it is merged to the master branch, ensuring a regression path to master if the feature does not work.

The master branch must ***always*** be deployable with GitHub Flow so ***only*** commit approved and successful changes to it as master is the main branch.

So, the process would be as follows.

1. Pull Master branch from remote repository
2. Create local branch e.g. Patch_1.0.1
3. Write code for Patch_1.0.1
4. Developer sends Patch_1.0.1 Pull Request to Maintainer
5. Patch_1.0.1 pulled to remote repository by Maintainer
6. **Patch_1.0.1 is deployed to required environment**
7. **If Patch_1.0.1 is successful merge it to Master**
8. Delete Patch_1.0.1 if not required further

Gitlab Flow

Summary: master > pre-prod > prod

1. Maintain master, pre-production and production branches
2. Write code, commit to the master branch and deploy in a test environment
3. On success commit to the pre-production branch and deploy in pre-production
4. On success send a PR requesting merge to the production branch and deploy in production

Managing A Local Git Repository

The following sections outline the steps to add and manage code in a local Git repository.

Initialising the Local Git Repository

This can be done with the following command.

sudo git init

This creates a hidden folder called **.git** in the current working directory and makes it ready to accept Git-based actions.

Configuring the Local Setup

The local setup requires SSH connectivity to the remote repository. So, items such as credentials email accounts must be configured using relevant Git CLI commands. It is advisable to configure SSH to connect with the Git remote repository so that the implementation recognises the relevant key(s) to use instead of you having to select them manually. This is shown in the following steps.

Run the following command that add the SSH key.

git config core.sshCommand "ssh -i sshkey.pem approved-key -F /dev/null"

Set the credentials as follows.

CURRENTNAME=IT

CURRENTEMAIL=info@gridlockaz.com
sudo git config --global user.name "$CURRENTNAME"
sudo git config --global user.email $CURRENTEMAIL

Adding the Remote Git Repository

Run the following commands.

This adds a new repository.

sudo git remote add origin https://github.com/gridlockaz /chat.git

You can set up the current repository to **cache the next authentication login** so the credentials don't need to keep being supplied during login.

sudo git config credential.helper store

Add all of the current files in the repository to the next commit (obviously only the changed files will be acted upon by any git actions).

*sudo git add **

Commit and push the changes to the master branch.

sudo git commit -m "Initial commit"
sudo git push -u origin master

The following shows the steps to add either an SSL or an SSH link as the repository, depending on the CLI authentication type required.

*sudo git add **
sudo git commit -m "Added new javascript html configuration"
sudo git push -u origin master
sudo git remote add origin ssh://git@github.com/gridlockaz/chat.git
(or sudo git remote add origin https://github.com/gridlockaz/chat.git)

If for any reason you need to change the repository, simply remove the current repository with the following command.

sudo git remote remove origin

Then add the new repository using the commands shown previously in this section.

Create a New Local Branch

First get the most up to date master from git.

git pull origin master

Then create the new branch using the git command's -b option.

git checkout -b gridbranch

Check Out, Review and Merge the Local Branch

Fetch and check out the branch for this merge request.

git fetch origin
git checkout -b "develop" "origin/develop"

Review the changes locally.

Merge the branch and fix any conflicts that come up using the no fast-forward option.

git checkout origin/master
git merge --no-ff develop

Push the result of the merge to GitLab

git push origin master

Alternative Merging From a Remote Repository

This can be done as follows.

Clone the required repository.

git clone test@gitlab.git
git fetch origin

Update code.

git merge origin/master # in case other developers have made changes since the repository was cloned

Test code.

git commit -am “update the index.html file”
git push origin master

At this stage you could run any particular manual or automated steps to verify that the code works. For full on CI implementation, you could use Gitlab’s runners as described later on in this document.

Updating a Local Branch

Always ensure that the latest version of code is local before creating a new branch by running the git pull command as follows.

git pull

Then create and switch to the required branch as follows. Let’s use the name *gridbranch* for this at present.

git checkout -b gridbranch

Push this branch online.

git push origin gridbranch

Check you are in the right branch as follows and when ready push upstream, adding the -u setting to set upstream.

View the branches and look for the asterisked branch as that is the current version of code being developed.

git branch -a

The output is as follows.

** approval_messages*

master
master_clean

Add the remote version of the branch. Let’s call this remote version gridremote.

git remote add gridremote gridbranch

Push the locally committed changes to the remote branch.

git push gridremote https://github.com/gridlockaz/chat.git

Retrieve the newest version of code from the now updated remote version.

git fetch gridremote

Then you need to apply to merge changes if your branch is derived from develop you need to do :

git merge gridremote/develop

Delete the local filesystem branch.

git branch -d gridbranch

Confirm deletion of the local branch.

git branch -D gridbranch

Delete the remote version.

git push origin :gridbranch

Updating a Local Branch: Alternative Method

A branch can also be updated as follows.

Get the latest code version.

git fetch

Look for the relevant branch e.g. check if the branch hotfix-a is listed in the following command's output.

git branch

Check out this branch.

git checkout hotfix-a

Make the necessary code changes and then add all files.

git add .

Then commit the changes.

git commit -m "adding code with the hotfix-a branch"

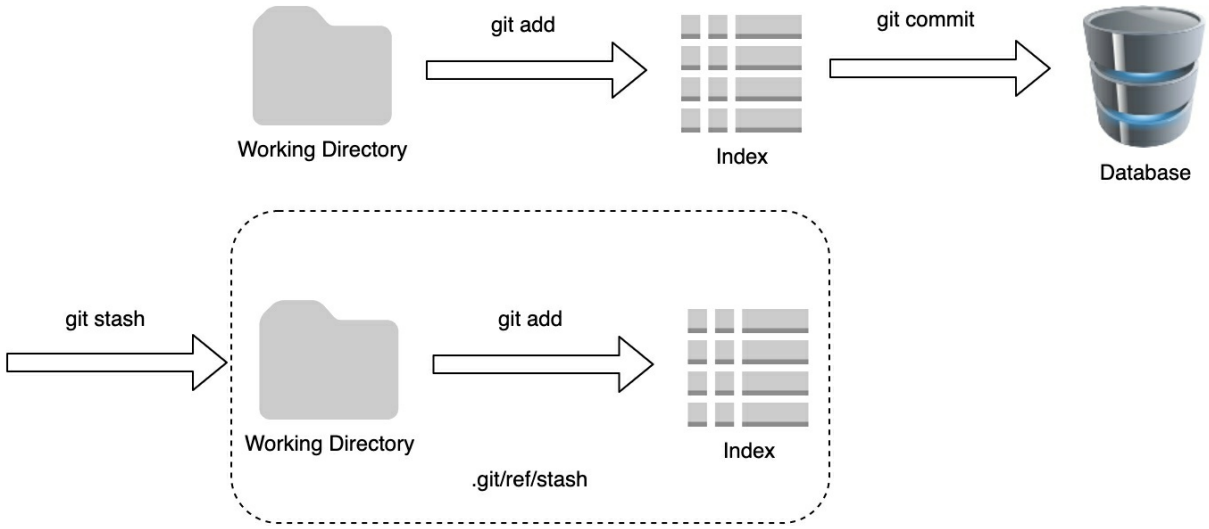
Push the changes to the hotfix-a branch.

git push origin hotfix-a

These changes will now appear as a merge request on Git server's web page.

Using the Git Stash Command

This command is used to save changes but not to the working directory. They are saved to a dirty working directory and these changes can be referenced in the .git/ref/stash. This command basically saves the working folder and index information without it getting in the way of the current working directory.



This shows commands such as the following.

git status
git stash save - keep a temp cache of files not yet committed
git stash pop - commit those files in the temp cache
git stash pop --index - show the current unsaved files index
git stash list - shows a list of your caches
git stash show - shows the current caches
git stash apply
git diff - compare revisions

Committing Changes

A commit checks any changes into the repository. Before committing these changes, run the following command to check for any whitespace issues.

git diff —check

Don’t make a lot of changes per commit and ensure your commit messages are succinct and clear. Avoid vague commit messages like ‘bug fixed’ or ‘fix item’. Use imperative language i.e. ‘Fix networking commands issue’ not ‘Networking commands issue fixed’. If you are using any ticketing system such as Service Now or Azure DevOps, use a relevant reference number such as the Service Now Incident Number of ADO Task Number in the commit

message. An example of this is shown below.

git commit -m "Task 12345 – add SMTP class to sending.py"

Updating the Commit Message

It might be necessary to update the commit message if it needs to change for any reason. You can amend a commit comment using the --amend switch as shown below.

git commit -m "Ticket 54321 – Added TextUpdate class to incidentmgmt.py"

[master 63f74b7] "Ticket 54321 – Added TextUpdate class to incidentmgmt.py"
1 file changed, 4 insertions(+)

Let's assume that the Ticket Number in the commit message was incorrect and should be 12125. The following would update it.

git commit --amend -m "Ticket 12125 – Added TextUpdate class to incidentmgmt.py"

[master 951bf2f] "Ticket 12125 – Added TextUpdate class to incidentmgmt.py"

Date: Tue May 22 20:41:27 2018 -0600
1 file changed, 4 insertions(+)

The output shows that the Ticket Number is now 12125.

Git HEAD and MASTER Configuration

Git HEAD can be moved from the MASTER position to an earlier position if necessary, to make an edit to an earlier version of a file and then restored later.

git config --global diff.tool meld
git config --global difftool.prompt false

The command with the prompt option ensures that the prompt is now shown

each time the tool starts.

This can be quite annoying as it does it for every file in a diff, one at a time.

git difftool HEAD^ HEAD

Git Rebase

A rebase operation is similar to a merge, but it can produce a much cleaner history. When you rebase, Git will find the common ancestor between your current branch and the specified branch. It will then take all of the changes after that common ancestor from your branch and “replay” them on top of the other branch. The result will look like you did all of your changes after the other branch.

Aliases

Git doesn’t predict and/or auto-complete commands as you type them on the CLI. So, you might want to create command aliases for convenience.

This can be done using the **git config --global setting** as follows with the equivalent comments shown in the comments.

git config --global alias.co checkout # ‘git co’ runs ‘git checkout’
git config --global alias.br branch # ‘git br’ runs ‘git branch’
git config --global alias.ci commit # ‘git ci’ runs ‘git commit’
git config --global alias.st status# ‘git st’ runs ‘git status’

Variants of the command syntax can be used. For example, this command alias shows the last commit.

git config --global alias.last 'log -1 HEAD' # ‘git last’ runs ‘git log -1 HEAD’

Whist this command resets the HEAD.

git config --global alias.unstage 'reset HEAD --' # ‘git unstage’ runs ‘git reset HEAD --’

Summarily, it is the same as the following command.

git reset HEAD – fileA

Overwrite Local Files Without Need for a Merge

Run the following commands.

sudo git fetch --all
sudo git reset --hard origin/master

Now carry out the following steps.

- Create the Git repository folder
- cd to that folder
- create the repo
- Link the local repo to the online repo.
- Pull from the current repository
- Commit the current code

These steps are shown below.

echo "# googlechatbox" >> README.md
sudo git pull origin master
sudo git add README.md
sudo git commit -m "first commit"
sudo git remote add origin https://github.com/gridlockaz/chat.git
sudo git push -u origin master

Resolving Git Conflicts with Git Mergetool

There could be instances of merge conflicts arising if two or more developers work on the same item of code simultaneously.

The following command can be used to show the conflict in a graphical format.

git mergetool

Another tool that is useful in resolving merge conflicts is opendiff, which can be downloaded on *nix systems. This tool is out of this book's scope.

A Summarised Git Strategy

Check out the master and create a hotfix branch.

git checkout master
git checkout -b hotfix

Work on the hotfix branch as required.

Test it.

Merge hotfix with the main branch.

git checkout master
git merge hotfix

The hotfix is now merged with the master using the ‘fast-forward’ process.

This means that the pointer was simply moved forward as the new branch is on the same path of the git history as the old branch i.e. there is no divergent work that requires merging to the current repository.

If it wasn’t on the same path, a ‘recursive’ merge would be used.

You can now delete the hotfix branch with the following command.

git branch -d hotfix

Merge conflicts can occur if the same part of a file was amended in different branches.

So, care is necessary when multiple branches have been checked out and look for any merge conflicts in the following command’s output.

git status

Gitlab High Availability (HA)

Introduction

Gitlab is an open source web-based implementation of a DVCS. It uses Git in the background to give developers the ability to work on a distributed repository from a common platform. Repositories are pushed to and pulled from it as has been shown throughout this book. This is done from a web-based, GUI actions rather than on the command line. Code can also be updated directly in the browser instead of from the CLI. This chapter details the design of a resilient Gitlab implementation. The example in this book ensures that any of the multiple read-only secondary Gitlab instances housing the Postgres database can be promoted to primary status in the event of a failover.

The following setup is accurate at the time of writing, but do check on the latest requirements on the Gitlab website before carrying out any implementations.

Requirements

As of this document's writing, Gitlab Geo requires the following components as a minimum requirement.

1. Ruby MRI 2.6
2. Go 1.12
3. Git 2.21.x if using Git 11.11 or higher
4. Node.js 8.10.0 if using Gitlab 11.8 or higher
5. 8GB RAM per 100 users as a minimum
6. 2 cores per 100 users as a minimum
7. PostgreSQL 9.6 as the lowest version

It will work with an operating system that supports OpenSSH 6.9+ (needed for fast lookup of authorized SSH keys in the database). Currently, Geo uses OpenSSH 6.9+ (for fast authorised SSH key lookup in the database), which runs on CentOS 7.4+ and this is the OS recommended. This and other requirements are listed below.

1. CentOS 7.4+ (or Ubuntu 16.04+)
2. PostgreSQL 9.6+ with FDW support and Streaming Replication
3. Git 2.9+
4. All servers must use the same Gitlab version.
5. Gitlab EE 10 Premium or higher software and relevant licencing

Node Features

As part of its high availability requirement, Gitlab Geo is implemented with a primary node and a secondary as the minimum setup for this requirement. This section details a few of the characteristics of the primary and secondary nodes.

Primary Node

This is a read-write Gitlab instance. This requires the standard installation as would be done for a single Gitlab instance. The configuration changes to make this a primary Geo server will be described later on in the installation section for the secondary node as these steps are linked.

Secondary Node

This is a read-only node that receives all of its data from the primary node. These devices get user data for logins using the API and replicate repositories, LFS objects and attachments using HTTPS and JWT. Git is used as the transport mechanism and Geo uses OpenSSH and gitlab-shell for SSH and gitlab-workhorse for HTTPS. The secondary nodes uses two databases as follows.

- Read-only: streams data from the main database
- Internal replication: stores replication information for use by the secondary node

The second database is also called the Geo Tracking database. There is also a program called the log cursor that runs on the secondary node.

Installation Summary

The installation steps can be summarised as follows.

1. Install Gitlab EE on the primary node using the normal single server installation process.
2. Install Gitlab EE on the secondary node using the normal single server installation process, but don’t log in to it after installation.
3. Upload the Gitlab Premium or higher licence to the primary node to unlock Geo’s features and start configuring it as the primary node.
4. Set up primary read-write (RW) to secondary read-only (RO) database replication.
5. Configure SSH fast lookup on the database server on both the primary and secondary nodes.
6. Assign nodes as primary or secondary nodes in Gitlab.
7. Configure a secondary LDAP server as secondary node.

Installing a Gitlab Server as a Primary Node

This requires the same steps as is done for a standalone server.

A typical Ansible playbook installation is shown in this folder's ansible-gitcode.yml.

Edit /etc/gitlab/gitlab.rb and add the following:

Enable Geo primary role

roles ['geo_primary_role']

Add a unique identifier for node

gitlab_rails['geo_node_name'] = '<node_name_here>'

Disable automatic migrations

gitlab_rails['auto_migrate'] = false

Configure primary role PostgreSQL roles

roles ['geo_primary_role', 'postgres_role']

Adding a Secondary Geo Node

Use the same Gitlab installation steps shown in the previous sections to install the secondary node before carrying on with the following steps.

This includes steps such as configuring the primary to secondary node replication so that all features and settings are present on both or all devices in the architecture. This entails synchronising the /etc/gitlab/gitlab-secrets.json and OpenSSH keys from the primary nodes to the secondary nodes.

Login to the primary node using SSH and run the following command.

sudo cat /etc/gitlab/gitlab-secrets.json

This will show the secrets that need replication in JSON format.

SSH into the secondary node and then login as the root user with the following command.

sudo -i

Back up the current secrets as follows.

mv /etc/gitlab/gitlab-secrets.json /etc/gitlab/gitlab-secrets.json.date +%F

Copy */etc/gitlab/gitlab-secrets.json* from the secondary node to the primary node (this could be done with a copy and paste or other preferred method).

sudo editor /etc/gitlab/gitlab-secrets.json

(or sudo cat /etc/gitlab/gitlab-secrets.json)

Save the output of this command to the same file location on the primary node and ensure the file permissions are correct as follows.

chown root:root /etc/gitlab/gitlab-secrets.json

chmod 0600 /etc/gitlab/gitlab-secrets.json

Reconfigure the secondary node as follows and the effect of these changes will be implemented.

gitlab-ctl reconfigure gitlab-ctl restart

Gitlab Geo-specific Configuration Steps

The following sections show the steps necessary to implement the setup specifically for the Gitlab Geo functionality.

Back Up the SSH Keys

find /etc/ssh -iname ssh_host_* -exec cp {} {}.backup.date +%F ;

Copy the keys to the secondary nodes.

scp root@<primary_node_fqdn>:/etc/ssh/ssh_host__key /etc/ssh

Extract files and set the correct permissions on the secondary node.

*sudo tar --transform 's/.///g' -zcvf ~/geo-host-key.tar.gz /etc/ssh/ssh_host__key**

Run Secondary Node Code

scp <user_with_sudo>@<primary_node_fqdn>:geo-host-key.tar.gz . tar zxvf ~/geo-host-key.tar.gz -C /etc/ssh

On your secondary node, ensure the file permissions are correct:

chown root:root /etc/ssh/ssh_host__key chmod 0600 /etc/ssh/ssh_host__key

Verify Keys

This should be done on all nodes.

for file in /etc/ssh/ssh_host_*_key; do ssh-keygen -lf $file; done

The output should be similar to the following output.

1024 SHA256:FEZX2jQa2bcsd/fn/uxBzxhKdx4Imc4raXrHwsbtP0M root@serverhostname (DSA) 256 SHA256:uw98R35Uf+fYEQ/UnJD9Br4NXUFPv7JAUln5uHlgSeY root@serverhostname (ECDSA) 256 SHA256:sqOUWcraZQKd89y/QQv/iynPTOGQxcOTIXU/LsoPmnM root@serverhostname (ED25519) 2048 SHA256:qwa+rgir2Oy86QI+PZi/QVR+MSmrdrpsuH7YyKknC+s root@serverhostname (RSA)

Verify the Private Keys

On each of the primary and secondary nodes, verify the keys using the following commands by ensuring that they have the same fingerprint.

for file in /etc/ssh/ssh_host_*_key; do ssh-keygen -lf $file; done

for file in /etc/ssh/ssh_host_*_key.pub; do ssh-keygen -lf $file; done

Restart SSH on the Secondary Node

Run the following command from the CLI to restart SSH on the current node being configured.

service sshd reload

1. Configure a tracking database on the secondary nodes.
2. Start Gitlab on all of the secondary nodes. Ensure that the /etc/gitlab/gitlab.rb has been configured so the device acts as a secondary node.
3. Ensure that the primary server has the secondary node configurations in Admin Area > Geo then /admin/geo/nodes.

The unique identifier for the Geo node.

gitlab_rails['geo_node_name'] = '<node_name_here>'

Reconfigure the Secondary Node

Add a unique identifier in /etc/gitlab/gitlab.rb as follows.

gitlab_rails['geo_node_name'] = '<node_name_here>'

Run the following command for the change to take effect.

gitlab-ctl reconfigure

On the primary node go to Admin Area > Geo (/admin/geo/nodes).

Click New node > Add secondary node

Then add the <node_name_here> used in the earlier gitlab_rails['geo_node_name'] /etc/gitlab/gitlab.rb setting to the Name field.

Add the external_url setting from */etc/gitlab/gitlab.rb* to the URL field.

It is very important that this matches exactly what is in the /etc/gitlab/gitlab.rb file on the secondary nodes.

Don’t tick the ‘This is a primary node’ checkbox.

If necessary, select any groups or storage shards that should be replicated by the secondary node or leave blank to replicate all – check if this is required by design.

Then click on the Add node button.

The secondary node addition is now done on the primary node.

Confirm Firewall Rules are Implemented

The following ports must be allowed in both directions between the primary and secondary nodes for correct functioning.

- 80 HTTP

- 443 HTTPS
- 22 SSH
- 5432 Postgres

Confirm that these ports have been allowed between the tenant instances housing the nodes by the relevant Security team.

Verify Geo Functionality is Working

The following commands can then be used to verify Geo is working correctly on both the primary and secondary nodes.

gitlab-rake gitlab:geo:check

If necessary, restart the Gitlab services on any relevant nodes using the following command.

gitlab-ctl restart

Gitlab Runner

A Gitlab runner is a physical or virtual device used to carry out CI/CD steps on behalf of a systems administration. If you have used systems like Jenkins then you will be used to the concept of pipelines as autonomous units of code capable of automatically running commands on local or remote systems towards the achievement of a specific deployment goal. Runners are the Gitlab-native version of that. You can link any physical or virtual machine to your Gitlab server and then issue a sequence of commands triggered by a script called .gitlab-ci.yml containing all the stages necessary to achieve the CI/CD goal.

This .gitlab-ci.yml script is sometimes referred to as the pipeline itself as it contains the pipeline commands. These commands are executed in sequence set by the configuration.

Gitlab > (.gitlab-ci.yml script) > Gitlab runner > commands issued

For instance, there could be a requirement to run an Ansible script that installs 10 CentOS servers in a Kubernetes cluster. You could then write a .gitlab-ci.yml script that would spin up a Docker container, install Ansible on it, download the Ansible playbooks from your repository and run the Ansible playbooks which would install the Kubernetes containers in their required pods, deployments and so on.

This would all be done automatically whenever a change in the .gitlab-ci.yml file is detected or on you manually starting the pipeline.

Installation

A runner is a device used for testing code updates. These can be single or group runners and can be on-premises or in the cloud such as GCP, AWS or Azure.

Log on the CLI of the device that will be the runner, then run the following commands to install the gitlab-runner binary.

The following is a list of typical variable settings required during the installation.

```
# REFERENCE https://docs.gitlab.com/runner/install/
# configuring the environment variable before the register command
# export CI_SERVER_URL=http://gitlab.example.com
# export RUNNER_NAME=my-runner
# export REGISTRATION_TOKEN=my-registration-token
# export REGISTER_NON_INTERACTIVE=true
# gitlab-runner register
```

The steps are as follows.

Download the Gitlab Runner package shell script

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh | sudo bash
```

Install the Gitlab Runner package using the relevant installer (below is using Yum found on CentOS and RHEL Linux platforms).

yum install gitlab-runner

Start the service and enable it.

systemctl start gitlab-runner

systemctl enable gitlab-runner

You can supply the Gitlab server URL and the registration token it supplied during the installation or these can be added during the registration with the following command.

gitlab-runner register --name my-runner --url {{ CI_URL }} --registration-token {{ CI_TOKEN }}"

The CI_URL value is that of the Gitlab server and the CI_TOKEN value is that for this particular runner being set up.

You will also be asked for a tag to use for it during the setup so that this runner can be used only for specifically tagged jobs if necessary.

The setup will also be asked which type of shell is preferred for usage by the runner. The options include shell and VirtualBox and you can select any preferred option.

Note that the CLI setup might not indicate that registration was successful but you can confirm this on the Gitlab server.

The configuration files location is as follows for systems installed as root.

/etc/gitlab-runner/config.toml

Or for systems where the runner is being launched aby a non-root administrator.

~/.gitlab-runner/config.toml

Configuration

A GitLab runner is a device that carries out the implementation of any scripts updated in that repository. It needs a .gitlab-ci.yml file in the root folder of the git repository whose code it must implement. The following are steps to set up GitLab automation and runners.

This is part of you CI pipeline or Continuous Integration pipeline.

Add a .gitlab-ci.yml file to the repository root directory.

This file should consist of at least one of the following stages.

- Build
- Test
- Deploy

Any stages not used are skipped.

This pipeline will appear in the CI/CD > Pipelines page in the Gitlab browser.

If the pipeline runs ok a green check will appear next to the relevant commit.

Note that the .gitlab-ci.yml file uses YAML syntax and therefore requires field value pairs.

Save this file to GitLab repository as follows.

git add .gitlab-ci.yml
git commit -m “adding .gitlab-ci.yml”
git push origin master

Then visit the Pipelines page, the pipeline will be there in a pending state.

If using a mirrored repository that GitLab pulls code from, you might need to enable pipeline triggered by going to Settings > Repository > Pull form a remote repository > Trigger pipelines for mirror updates.

The Commits page will also show a pause icon next to the relevant commit.

A runner runs the repository's .gitlab-ci.yml scripts. It can be a physical or virtual machine, VPS, Docker container or cluster. GitLab communicates with the runner using an API so there must be network access between these devices. Runner can be project specific (only used by a specific Gitlab project) or shared (by different projects on the Gitlab server).

View the current runners in Settings > CI/CD.

Runner Setup in Gitlab

Set up a Gitlab Runner as follows.

Go to Settings > CI/CD and click Expand on the Runners section of the page.

You will now be presented with options to install a specific or shared runner. Let's set up a specific runner for now.

Click on the link to get the Gitlab runner instructions.

Pipeline Setup in Gitlab

This requires the addition of a .gitlab-ci.yml pipeline file to the root of the repository that contains the code being automatically implemented.

- Go to the Gitlab page > CI/CD menu > Pipelines option.
- Select the .gitlab-ci.yml file and review its stages.
- Click on the Run Pipeline option that appears on screen.
- Check that the runner is reachable on 443 or 80 required for this.
- Review the Pipeline Status column to determine whether or not it ran successfully.
- The pipeline stages link to a CLI view of its running which can be reviewed further.

Introduction to CICD Methodology

CICD stands for Continuous Integration/Continuous Deployment. There is

also Continuous Delivery which is slightly different.

These concepts can be summarised as follows.

- Continuous Integration: build and test only
- Continuous Delivery: build, test and manually deploy
- Continuous Deployment: build, test and automatically deploy

Continuous integration is the process of applying changes to code bases at a non-intrusive and considerable frequency. The emphasis must be on small, incremental changes being delivered by multiple contributors and developers.

Thus, the key phrase to emphasize this is ‘little changes done often’. If you are only applying changes every other month or the changes are consistently quite large, you are not practicing CI. There are obviously exceptions, i.e. a particular feature might merit considerable changes to code but these must be exceptions and not the norm.

Small changes mean regression (if needed) is in turn a small and manageable task. This means such changes can be regressed quickly and with minimal impact. It also means that the changes lost as a result are also minimal and deltas between pre and post change code are small, both from the developer and user point of view.

A typical CICD process is shown below.

- Developer checks out a dev branch
- Developer applies changes to dev branch
- Developer tests these changes locally
- Developer checks in the successfully tested changes
- Reviewer reviews the checked-in branch
- Reviewer accepts the new changes
- Git runners automatically test the code

Gitlab-CI is Gitlab’s implementation of CI/CD by the use of a pipeline. .gitlab-ci.yml is the Gitlab CI pipeline file and is used to declaratively set ***the CICD build, test and automatic deploy*** commands that will be carried out by it. A .gitlab-ci.yml file is saved in the root of the repository containing test

and implementation commands. These commands are initially launched on a test instance known as a runner before being copied over and installed on production devices. This fulfils ***the automatic build and test*** function of ***CI***. The structure of the file is typically as follows.

```
job1:
  script: "execute-script-for-job1"

job2:
  script: "execute-script-for-job2"
```

Each of the jobs show will be executed in sequence or according to programmatic settings. These settings will be discussed later on in this book.

GitOps

GitOps is a DevOps methodology posited by Weaveworks (https://www.weave.works). It proposes the automated continuous deployment of cloud-based devices using available tools such as Gitlab. Weaveworks started GitOps as a means of managing Kubernetes clusters. It uses Git as the single source of truth and uses declarative statements to implement this for the infrastructure and applications. It is specifically used in Kubernetes deployments and is considered by some as a DevOps subset, but some consider it entirely distinct from DevOps. It can be applied or at least some of its principles can be applied to any infrastructure as code (IaC) environment.

It works by keeping the application deployment state as a declarative configuration, typically in a YAML file stored in either an application or environment repository. Any changes in the YAML configuration triggers an automatic implementation using a Git pipeline. The configuration itself consists of a number of ***descriptors***. A descriptor is simply a key value pair detailing the current version or state of the application to use. Below is a summary of the repository structure required.

- Application Repository: containing application configurations and binaries

- Environment Repository: containing only the application descriptor files

Note that these repositories can be in the same or different locations. Same location setup is used by the Push-based deployment, whilst different locations used by Pull-based. A key feature of GitOps is that it gives development teams the freedom to choose deployment tools that suit them and the current deployment requirements.

The foundation of Gitlab or other repository system does not tie development teams to a particular technology.

These will be described a bit later in the document.

Principles

It is worth noting the following principles applied in GitOps.

- Declarative description of the entire system is compulsory
- All changes are driven by and version controlled using Git
- Automatic changes only and manual CLI changes frowned upon
- Software agents keep track of changes and add self-healing

A declarative system is defined by facts rather than instructions.

This shows the actual ***desired state*** i.e. the state in which the system intends to exist based on the code in the repository.

This achieves a single source of truth in the Git repository.

Benefits

There are numerous benefits and some of these are listed below.

- Faster deployments, as a simple configuration change triggers deployment
- Control as admins now use Git as the sole change tool without local logins
- Full change audit trail using Git history

- Version control is a lot clearer as descriptors can use Semantic Versioning
- Developer familiarity with tools used so improved experience
- Improved stability based on clear auditing
- Standardisation of code deployments as engineers use the same models
- Increased security by avoiding local, unplanned changes

Process

A summary of a typical process is as follows.

- Developers update code and issue a pull request (PR) on a repository
- The maintainer(s) of the repository approve the PR
- The code changes are committed
- Agents check Gitlab repositories for any change detect the latest commit
- Agents push these changes to the execution nodes
- Execution nodes implement the changes

The committed changes could be anything from a simple change in the number of nodes in a cluster to the change of its service settings e.g. Istio sidecar or other connectivity settings.

Push v Pull Based Deployments

There are different types of GitOps pipeline deployments that can be used to automate and ensure that desired state is maintained in accordance with declarative code stored. As mentioned previously there are two types of GitOps deployments known as Push-based and Pull-based deployments. These are described as follows.

Push-based Deployment

This is the triggering of a pipeline based on changes in an environment repository's or image registry configuration. The change is pushed to the environment repository by the application repository and it triggers the

deployment. All changes are logged to Git and therefore appear in its history. This is summarised in the following flow for clarity.

App Repo change > Build Pipeline > Env Repo change > Deploy Pipeline > Env Updated

Note that this change will only occur if there is a change in the application repository. Any environment changes do not lead to changes.

Push-based deployments tend to pose security risks by divulging credential information outside of the automation cluster.

Pull-based Deployment

This uses the concept of an automated operator tasked with continuously checking the environment repository for changes. This takes over the Deployment Pipeline role and triggers a build on noticing any changes in the environment repository or image registry if used. It can also regress changes in the environment that are not in the environment repository. Again, all changes are logged to Git and therefore appear in its history. So, a change in the image version stored in a Docker repository would cause a pull implementation when it is detected by the automated scheduler. Credentials are not divulged in such deployments.

Push-based	Pull-based
Requires ‘god-mode’ as pipeline has authentication details	Does not require ‘god-mode’ so possible seen as more secure
Can only implement changes	Can implement and regress changes
Credentials are outside the environment cluster	Credentials are kept inside the environment cluster
Config status in same repo as the applications	Config status in different repo to applications

GitOps leans in the direction of Pull-based Deployments.

Pipeline Types

There are different types of pipelines and these are discussed below.

- Basic
- Parent-child
- Direct Acyclic Graphs (DAG)

Basic

Parent-child

This is a parent pipeline.

```
stages:
  - triggers

trigger_a:
  stage: triggers
  trigger:
    include: a/.gitlab-ci.yml
  rules:
    - changes:
      - a/*

trigger_b:
  stage: triggers
  trigger:
    include: b/.gitlab-ci.yml
  rules:
    - changes:
      - b/*
```

This is its child pipeline.

```
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something."

test_a:
  stage: test
```

```
  needs: [build_a]
  script:
    - echo "This job tests something."

deploy_a:
  stage: deploy
  needs: [test_a]
  script:
    - echo "This job deploys something."
```

This is the additional pipeline. Note that it uses the DAG needs: keyword.

```
stages:
  - build
  - test
  - deploy

image: alpine

build_b:
  stage: build
  script:
    - echo "This job builds something else."

test_b:
  stage: test
  needs: [build_b]
  script:
    - echo "This job tests something else."

deploy_b:
  stage: deploy
  needs: [test_b]
  script:
    - echo "This job deploys something else."
```

The needs keyword effectively means that the stage referred to in the needs keyword only requires that particular stage and none of the others for the current stage to run.

E.g.

Linux-setup:

- *Code here*

Linux-prechecks:

- *Code here*

Linux-build:
  needs: linux-setup

linux-test:
  needs: linux-prechecks

This means that the linux-build stage only needs for linux-setup to complete before it will run. It doesn't need linux-prechecks to complete before it launches.

The job referred to must exist for the pipeline to run.

Direct Acyclic Graphs (DAG)

This creates relationships between jobs facilitating fast pipeline deployment. It does this by allowing certain pipelines to skip others in earlier stages. A DAG will look at the jobs used to build the application and will automatically determine relationships between them to determine the order in which they should be implemented. This is all done by DAG itself without any need for user intervention. Each stage will then be execution as soon as possible rather than waiting for other stage completion.

DAG uses the needs: keyword, which enables the running of non-sequential pipelines.

```
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something quickly."

build_b:
  stage: build
  script:
```

```
    - echo "This job builds something else slowly."

test_a:
  stage: test
  needs: [build_a]
  script:
    - echo "This test job will start as soon as build_a finishes."
    - echo "It will not wait for build_b, or other jobs in the build stage, to finish."

test_b:
  stage: test
  needs: [build_b]
  script:
    - echo "This test job will start as soon as build_b finishes."
    - echo "It will not wait for other jobs in the build stage to finish."

deploy_a:
  stage: deploy
  needs: [test_a]
  script:
    - echo "Since build_a and test_a run quickly, this deploy job can run much earlier."
    - echo "It does not need to wait for build_b or test_b."

deploy_b:
  stage: deploy
  needs: [test_b]
  script:
    - echo "Since build_b and test_b run slowly, this deploy job will run much later
```

It can also be used with the ***parallel*** keyword, which allows a number of jobs to be implemented simultaneously.

Includes

Includes give the developer a means of directly importing another CI file in a different location into the running pipeline dynamically.

Included files can be local or remote. An example is shown below.

```
include:
  - 'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'
  - '/templates/.after-script-template.yml'
```

The keywords can be set differently.

```
include:
  - 'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'
  - '/templates/.after-script-template.yml'
  - template: Auto-DevOps.gitlab-ci.yml
  - project: 'my-group/my-project'
    ref: master
    file: '/templates/.gitlab-ci-template.yml'
```

Note that the inheritance only applies to keys and not to values.

So, if the key server: is found in both the included yaml and the recipient yaml, the recipient's value takes precedence.

Includes can be nested, with each layer being overwritten by its precedent. They can also be added to before_script and after_script templates.

```
before_script:
  - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
  - gem install bundler --no-document
  - bundle install --jobs $(nproc)  "${FLAGS[@]}"
```

This will then be referenced in .gitlab-ci.yml as shown below.

```
include: 'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'

rspec:
  script:
    - bundle exec rspec
```

Values will be overwritten by the new values.

```
variables:
  POSTGRES_USER: user
  POSTGRES_PASSWORD: testing_password
  POSTGRES_DB: $CI_ENVIRONMENT_SLUG

production:
  stage: production
  script:
    - install_dependencies
    - deploy
  environment:
    name: production
    url: https://$CI_PROJECT_PATH_SLUG.$KUBE_INGRESS_BASE_DOMAIN
  only:
    - master
```

The Postgres credentials in the above example will be overwritten by those below in line with these specifics.

```
include: 'https://company.com/autodevops-template.yml'

image: alpine:latest

variables:
  POSTGRES_USER: root
  POSTGRES_PASSWORD: secure_password

stages:
  - build
  - test
  - production

production:
  environment:
    url: https://domain.com
```

Conclusion

That’s it for now folks. Hopefully this has been a useful ‘introduction’ or ‘reminder’ of some key concepts and will continue to be useful to you as a serious IT professional. We are constantly striving to enhance the Big Little Book series so let us know if there are any topics you would like to see in future editions of this book. That’s it for now, let us know if there’s anything you would like added to the next edition of this book by sending an email to info@gridlockaz.com.

Thanks for reading and wishing you all the best in your career pursuits.

Take care.

U V Omos