GIT & Gerrit

Agenda - Day1

- Introduction to GIT - Overview, Installation, folder structure
- Basic GIT Commands - Stage, commit, add, remove, Unstage, Untrack
- Git ignore
- git alias, log, amend commit, checkout
- GIT revert, reset
- Branching & Merging & Conflict Resolution
- git Stash

Will do Gerrit Installation by end of day today

Agenda - Day2

- git rebase
- Rebase interactive
- Tags
- Cherry Pick
- Patch
- GIT Hooks
- Gerrit Introduction - Overview, pratical usage
- Create Gerrit Project, setup Users, Clone project, etc.

Agenda - Day3

- Run Gerrit Scenarios for Merge, Rebase, Cherrypick, Abandon, Revert
- Use of Topic in Gerrit Code Review
- Gerrit Project Access

Centralized VCS

SVN, CVS, TFS

Vs

Decentralized VCS

GIT, Mercurial, Cogito

Git Client

Vs

Git Server

Installation

Configure GIT repository

git config -l

git config - -global user.name ‘parvez’

git config - -global user.email ‘parvez@booleanminds.com

git config - -global user.email - -replace-all ‘parvez@booleanminds.com

GIT Config

Global Level - All repositories for an account on a machine. ~/.gitconfig or

~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.

System Level - All repositories, all account on a machine. /etc/gitconfig file:

Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

Repository Level - For a particular repository. config file in the Git directory (that

is, .git/config) of whatever repository you're currently using: Specific to that single repository.

Git Folder Structure

CONFIG - Contains settings for this repository.

HEAD - The current ref that you're looking at. In most cases it's probably refs/heads/master

HOOKS - This contains scripts that are executed at certain times when working with Git, such as after a commit or before a rebase.

INDEX - The staging area with meta-data such as timestamps, file names and also SHAs of the files that are already wrapped up by Git.

Git Folder Structure

INFO - Relatively uninteresting except for the exclude file that lives inside of it.

LOGS - Contains history for different branches.

OBJECTS - Git's internal warehouse of blobs, all indexed by SHAs.

ORIG_HEAD - When doing a merge, this is the SHA of the branch you're merging into.

REFS - the master copy of all refs that live in your repository, be they for stashes, tags, remote tracking branches, or local branches.

DESCRIPTION - If you're using gitweb or firing up git instaweb, this will show up when you view your repository or the list of all versioned repositories.

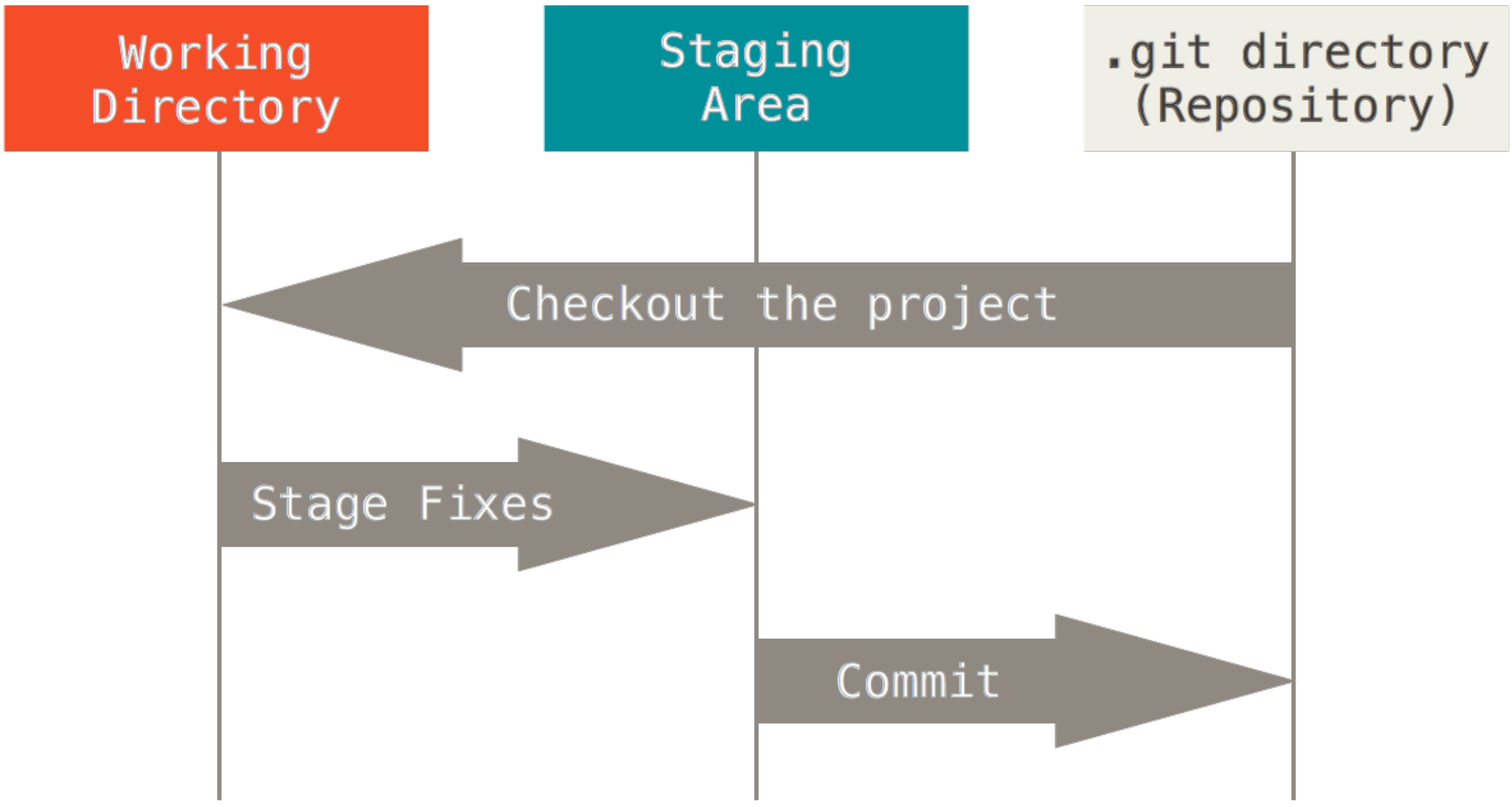
GIT Objects

BLOB - A blob generally stores the contents of a file.

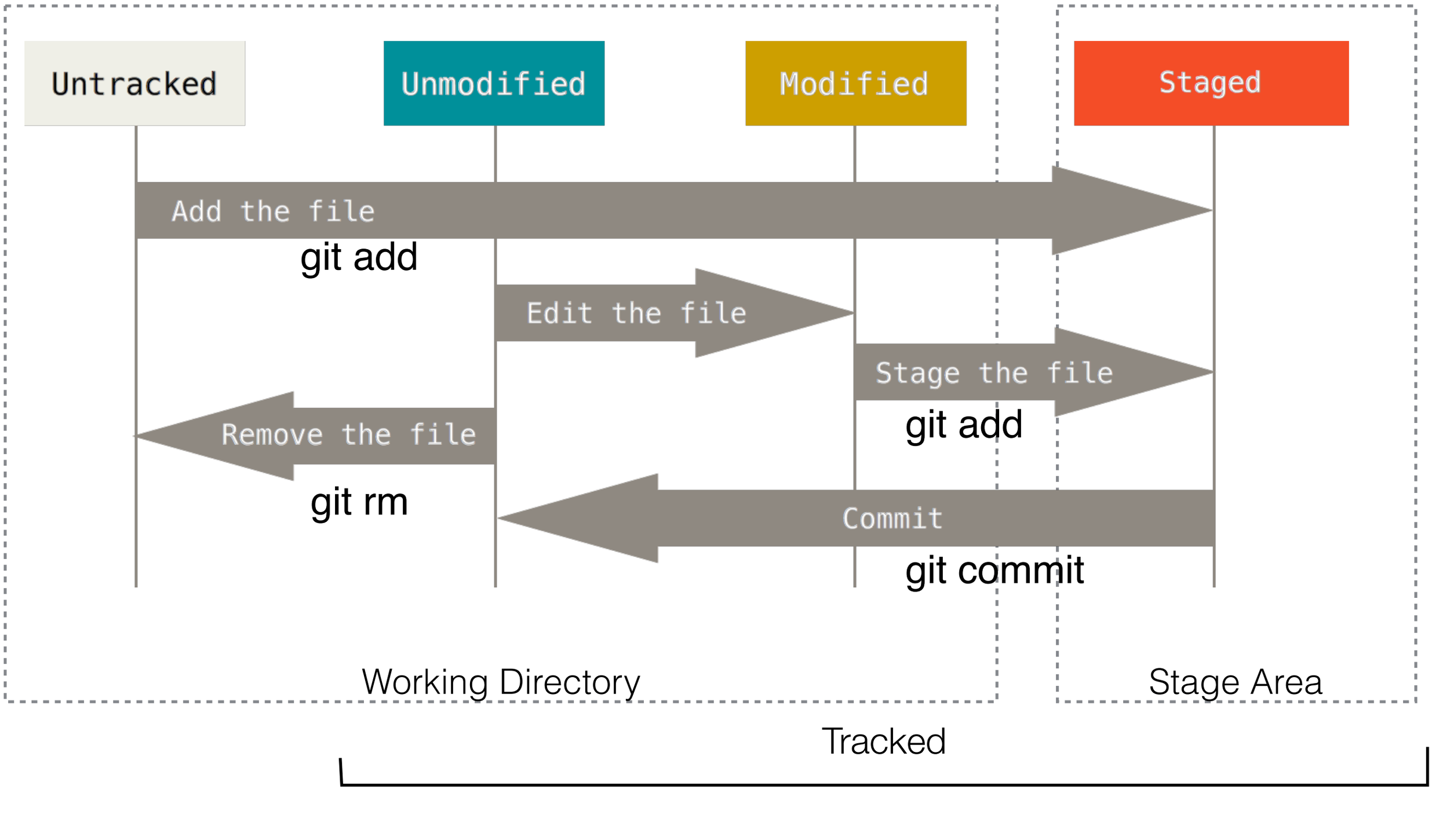
TREE - is basically like a directory - it references a bunch of other trees and/or blobs

COMMIT - Points to a single tree, marking it as what the project looked like at a certain point in time. It contains meta-information about that point in time, such as a timestamp, the author of the changes since the last commit, a pointer to the previous commit(s), etc.

Git repository local

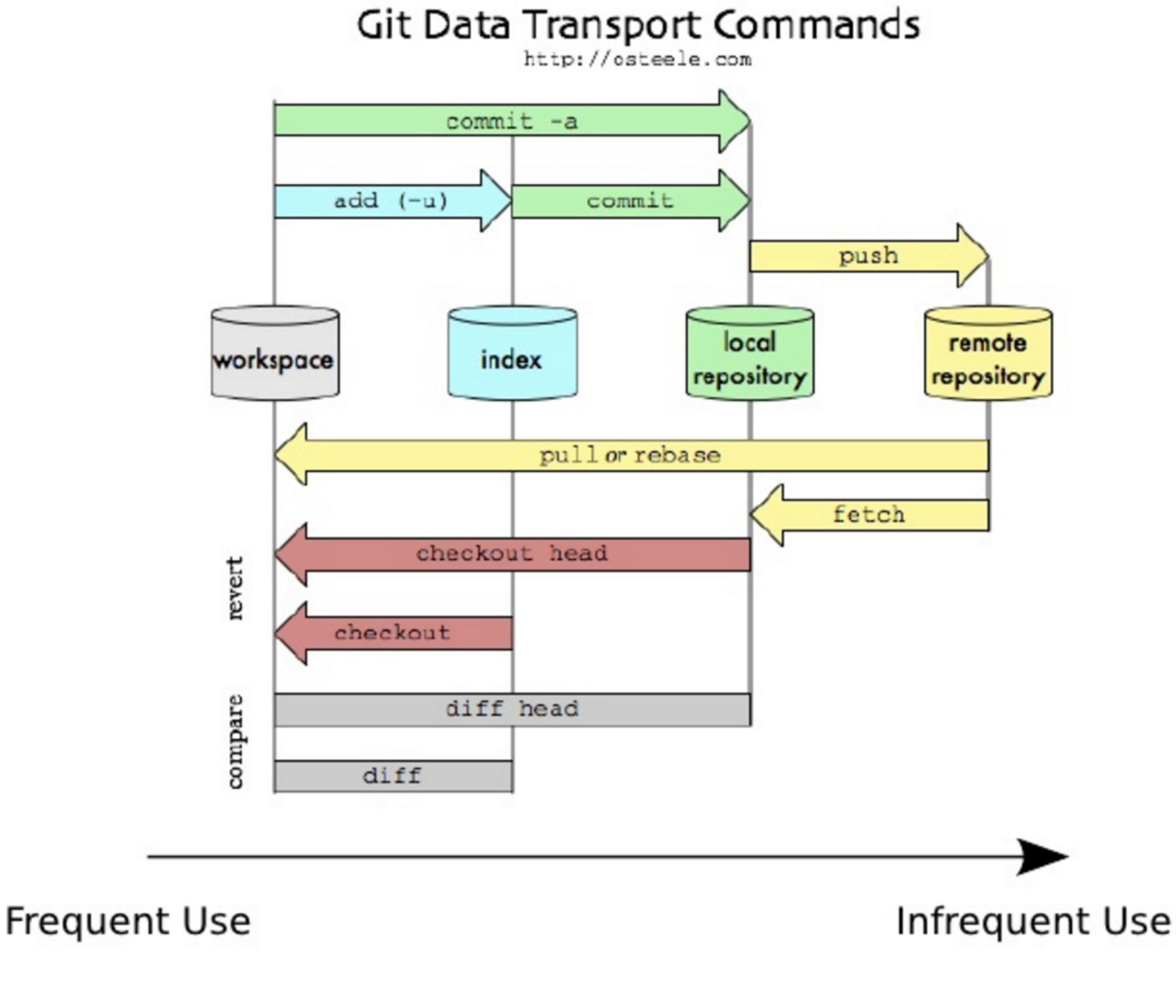


File status


Untracked
Unmodified
Modified
Staged
Add the file
git add
Edit the file
Stage the file
git add
Remove the file
git rm
Commit
git commit
Working Directory
Stage Area
Tracked

Exercise

1. Open Git bash
2. Configure - git config —global user.name ‘parvez’
3. Configure - git config —global user.email ‘parvez@abc.com’
4. Create folder - mkdir repo1 and cd inside it
5. Create a file - touch index.txt
6. Create repo - git init
7. Check the commit status - git status
8. stage(add) a file - git add index.txt
9. Look at .git/object folder
10. Commit the file - git commit -m ‘commit message’
11. Look at .git/object folder
12. Check the commit status - git status
13. Modify the file again and repeat the above steps


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout head
revert
checkout
compare
diff head
diff
Frequent Use
Infrequent Use

Viewing old files

- Contents of a file at a particular commit
  - git show <commitish>:<path to file>
- Contents of a directory
  - git show <commitish>:<directory>

Pointing Fingers

- git blame <file>
  - show who committed each line
- git blame <commit ID> <file>
  - show the line history before that commit

Unstaging Changes

- Revert to the last commit
    - git reset - -hard HEAD
- Remove all added changes from the index
    - git reset - -mixed HEAD
- Remove some files added to index
    - git add -i and chose revert or
    - git reset HEAD filename(s)

Staging Area

- git add <filename>
- git add *.txt
- git add .
- git reset <file> - to remove staged contents
- git diff —staged - Shows difference between staged and last commit
- git diff - differences not yet staged

GIT LOG

- git shortlog -s -n : To get the number of commits
- git log -p
- git log - -oneline
- git log - -graph
- git log - -stat: Show statistics for files modified in each commit
- git log - -shortstat: Display only the changed/insertions/deletions line from the --stat command.git
- git log - -name-only: Show the list of files modified after the commit information.
- git log - -name-status: Show the list of files affected with added/modified/deleted information as well.
- git log --pretty=oneline
- git log --pretty=format:"%h - %an, %ar : %s”
  - %H-Commit hash, %h-Abbreviated Commit hash
  - %T- Tree hash, %t-Abbreviated tree hash
  - %P-Parent hashes, %p-Abb.. parent hashes
  - %an-Author name, %ae-Author email, %ad-Author date, %ar-Author date relative
  - %cn-committer name, %ce-Committer email, %cd-Committer date, %cr-Relatives
  - %s: Subject (commit message)

Git rm

- **Command**
  - git rm - Remove files from the working tree and from the index
  - -q, --quiet do not list removed files
  - --cached only remove from the index (to untrack)
  - -n, --dry-run dry run
  - -f, --force override the up-to-date check
  - -r allow recursive removal
  - --ignore-unmatch exit with a zero status even if nothing matched

GIT Clean

- **Command**
- clean - Remove untracked file from working directory
  - git clean -n - Lists items to be cleaned.
  - git clean -f -n - Show what will be deleted with the -n option:
  - git clean -f -d - Also removes directories
  - git clean -f -X - Also removes ignored files
  - git clean -f -x - Removes both ignored and non-ignored files
  - git clean -i

.gitignore

Gitignore - To ignore files to be tracked

- git check-ignore -v * - To list ignored files

Steps

- Create .gitignore file
- Add contents to be ignored
- Stage and Commit the file

.gitignore

*.log - Ignore all files with .log extension

.log

/config/databases.yml

/log/* - Ignore log directory and everything under it

/data/sql/*

/lib/filter/base/*

/lib/form/base/*

/lib/model/map/*

/lib/model/om/**

git diff

- **git diff –staged** - Shows difference between staged and last commit
- **git diff**: Show differences between your working directory and the index.
- **git diff –cached**: Show differences between the index and the most recent commit.
- **git diff HEAD:** Show the differences between your working directory and the most recent commit.

Undoing Things

git commit – –amend – To amend last commit message

git reset HEAD <filename> – To unstage

git checkout – – <filename> – to discard changes from the

working directory

git alias

- git config —global alias.co commit
- git config —global alias.last ‘log -1 HEAD’
- git config --global alias.unstage 'reset HEAD —'
- git config —global alias.lol ‘log —pretty —oneline —decorate —graph —all’

git checkout

git checkout <commit>

git checkout <commit> <file>

git checkout <branch>

- git checkout -b <newbranch>
- git checkout -B <newbranch> - new branch is created if it doesn’t exists, otherwise it is reset

git checkout

Command	Scope	Common use cases
`git reset`	Commit-level	Discard commits in a private branch or throw away uncommited changes
`git reset`	File-level	Unstage a file
`git checkout`	Commit-level	Switch between branches or inspect old snapshots
`git checkout`	File-level	Discard changes in the working directory
`git revert`	Commit-level	Undo commits in a public branch
`git revert`	File-level	(N/A)

git revert

git revert

- git revert HEAD~1
- git revert - -no-edit
- git revert - -abort
- git revert - -continue

git reset

- --**soft** – The staged snapshot and working directory are not altered in any way.
- --**mixed** – The staged snapshot is updated to match the specified commit, but the working directory is not affected. This is the default option.
- --**hard** – The staged snapshot and the working directory are both updated to match the specified commit.

GIT Branch

git branch - To list branches available in repository

git branch <newbranch> - To create a new branch

git checkout <branchname> - To switch to a branch

git merge <branchname> - To merge the branch

Bitbucket Flow

- Create branch on bitbucket
- git pull origin - To pull the branch
- git checkout -b <branchname> <remotebranchname>
- Make some change and commit..
- git push origin <branchname>

{Go to bitbucket Admin and create a user}

{Go to repository settings and assign permission to the user in repository}

- Create pull request
- Add a reviewer
- Start another browser and login with another user and Review and comment and approve
- Come back to original user - Merge

Git Workflow

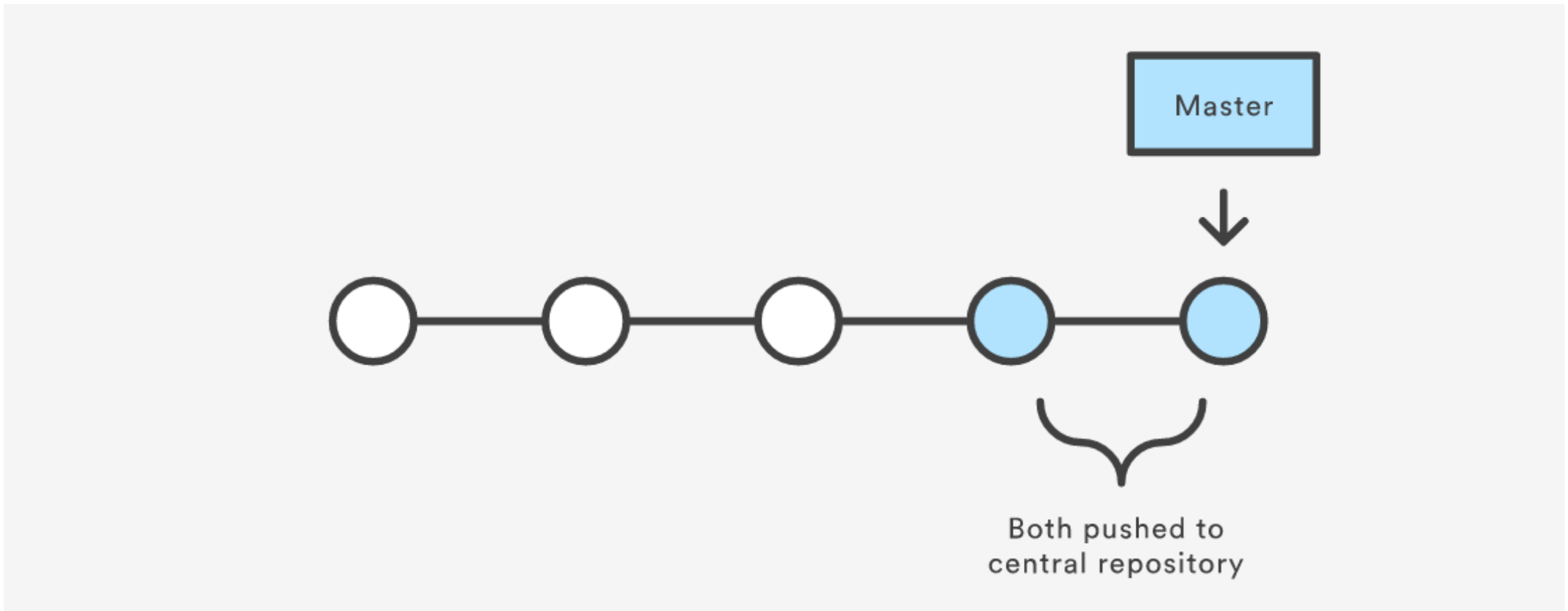
Centralized Workflow

Feature Branch Workflow

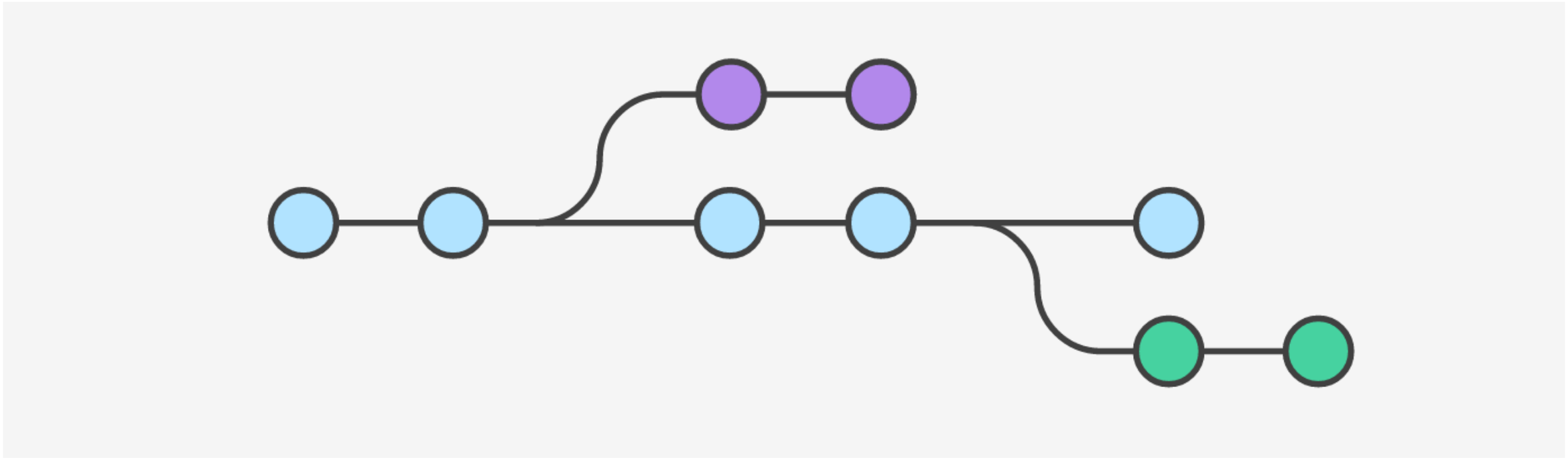
Gitflow workflow

Forking workflow

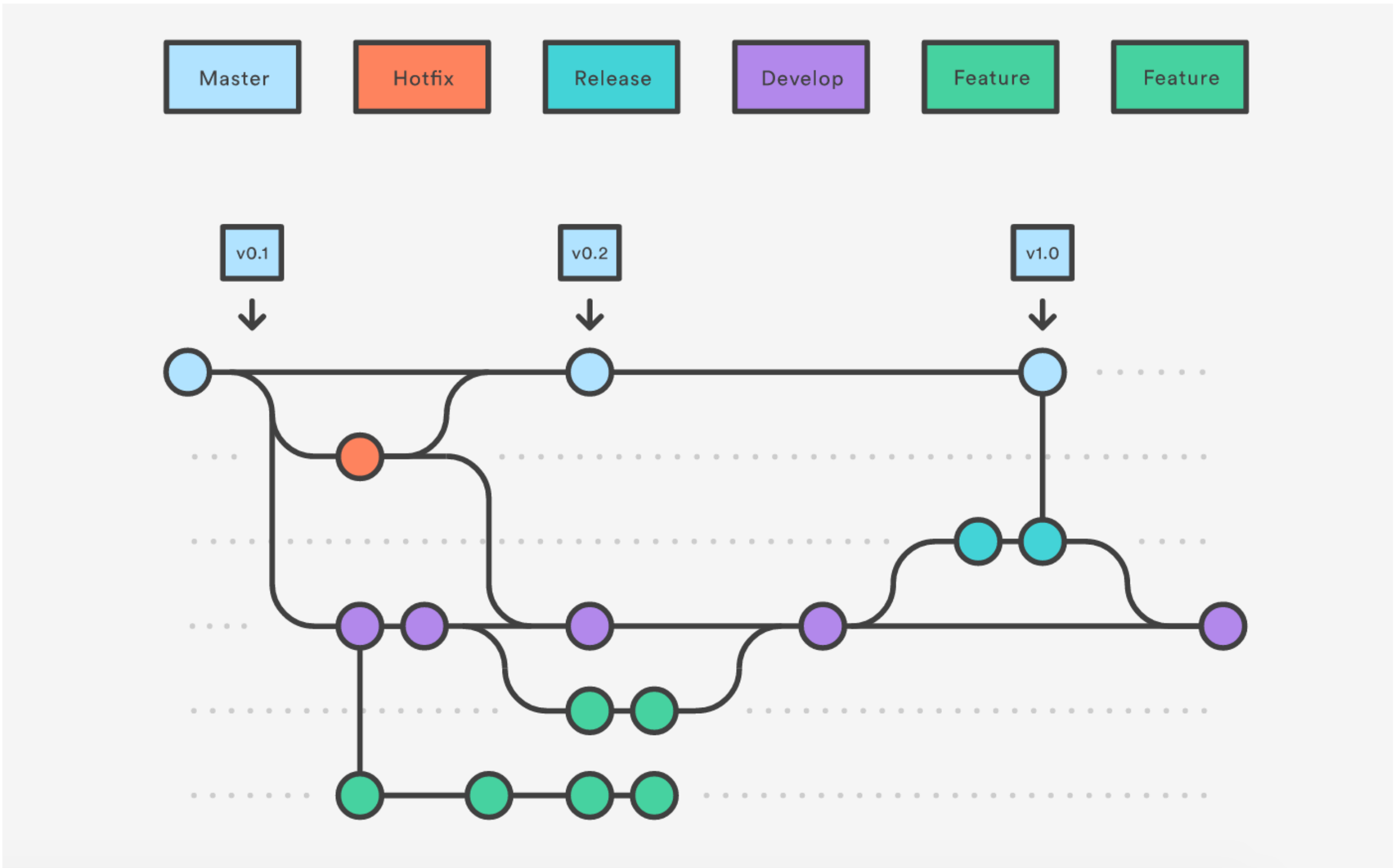
centralized Workflow



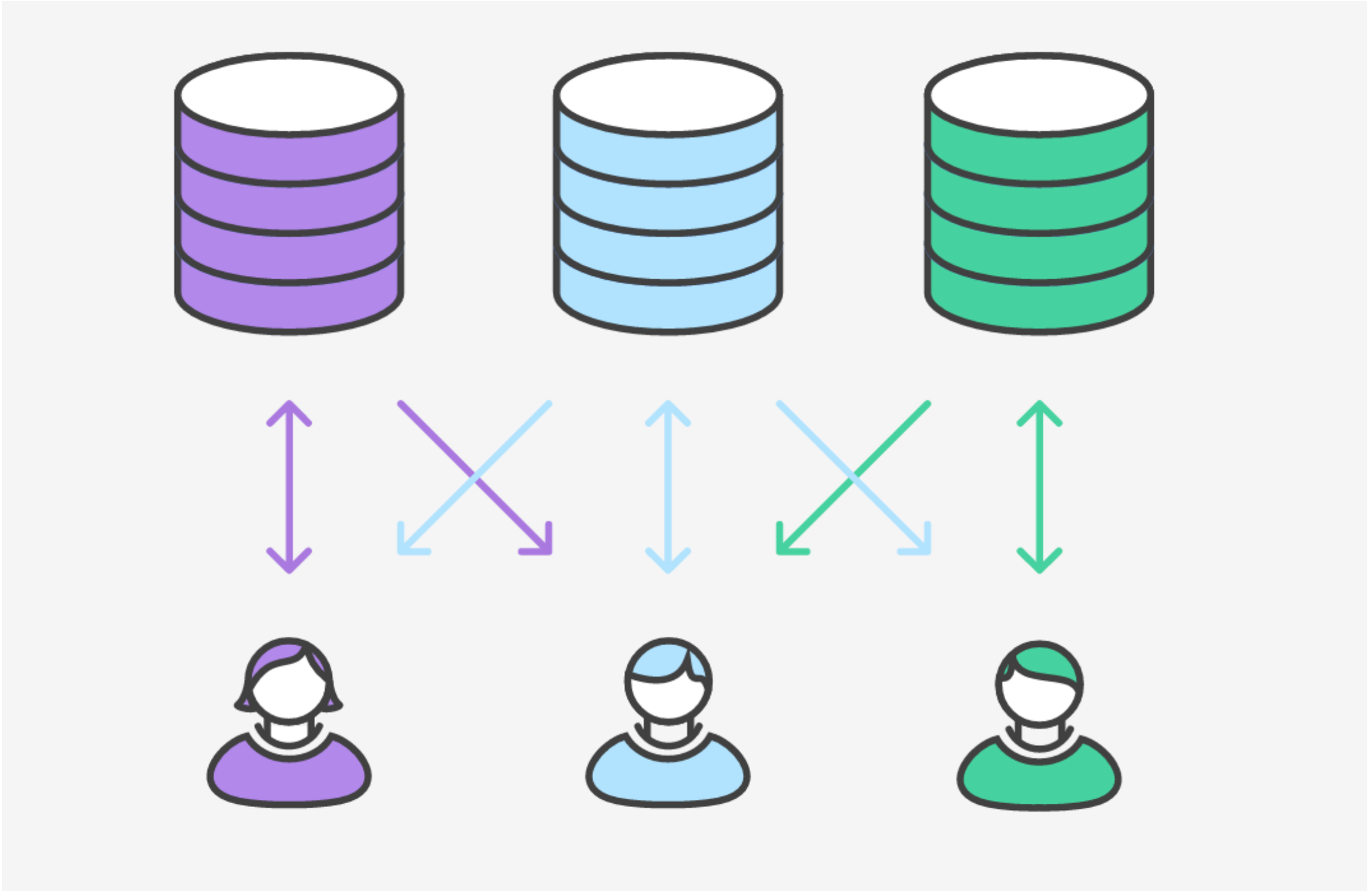
Feature Workflow



Git flow workflow



Forking Workflow



Branching and merging

git branch <new branch>

git checkout -b <new branch>

git branch -d <existing branch> - Delete merged branch

git branch -D <existing branch> - Delete Unmerged branch

git branch -m <existing branch>

Git merge

- Fast Forward merge
- Recursive Merge with no conflict. Adding a new file
- Merge with conflict

Steps

- git status - Ensure working directory is clean on master
- git branch feature2 (to create a feature2 branch)
- git checkout feature2 (to checkout and work on feature2 branch)
- Create new file.. git add and git commit (on feature2)
- git checkout master
- change existing file and commit
- Git merge feature2

Working with Remote

Scenarios

- Create local repository and push to remote
- Clone repository

GITHUB

User Account

Organization

Teams & Permissions

Collaborators

Pricing and Installation

git rebase

If you are using git pull and want to make --rebase the default, you can set the pull.rebase config value with something like **git config --global pull.rebase true.**

- Create branch1 from master
- Make change on master and commit
- Checkout to branch1
- Make change on branch1 and commit
- Rebase (git rebase master)
- In case of conflict, it will give error message of Automerge fail
- Manually resolve the conflict
- Git add
- git rebase —continue (this will give message Applying commit in branch3
- ———This completes the rebase———-

- Now fast forward merge will be seamless

git rebase interactive

- pick
- reword - Reword commit messages
- squash - which melds the commit into the previous one
- fixup - which acts like “squash”, but discards this commit’s message

git stash

git stash and git stash apply

git cherrypick

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
            \
             76cada - 62ecb3 - b886a0 [feature]
```

Used to merge only specific commits from another branch into your current branch

git cherry-pick 62ecb3 : 62ecb3 is now applied to the master branch and commited (as a new commit) in master.

Cherrypick Range of Comments

- You'd want commit a and b of branch in master.
- Branch has 3 commits (a, b and c)
- git checkout -b newbranch b
- git rebase --onto master a^

Steps to Practice - Synchronizing with Remote

- Git pull origin master
- change file on local and commit (dont push)
- go to another folder (outside repo)
- clone the repo
- make change same file and line and commit and push
- come back to original repo location
- try to push - it will reject
- git fetch origin master
- git rebase origin/master (this will give conflicting file)
- open file, resolve conflict..
- git add .
- git rebase - -continue
- git push origin master (it will be successful)

Create demo project and repo1 on bitbucket

on local machine: git remote add origin <url>

git push -u origin master

Steps to Learn - Pull Request

- Create branch in Stash
- Pull branch - git pull <branchname>
- git checkout <branchname>
- change files.. commit
- git push origin <branchname>
- On Stash - Click on Pull Request and Add Pull Request
- Provide Source as branchname, and destination as Master
- Add Reviewers
- Once review is accepted.. Click on Merge button to merge the branch to master

git patch

git format-patch HEAD~ (git format-patch -1) - Create patch of last one commit

git format-patch master --stdout > fix_empty_poster.patch

- This will create a new file fix_empty_poster.patch with all changes from the current (fix_empty_poster) against master.

git format-patch COMMIT_REFERENCE

Apply Patch

Git provides two commands to apply patches git amand git apply, respectively. Git apply modifies the local files without creating commit, while git am modifies the file and creates commit as well.

git apply --stat fix_empty_poster.patch

- Take a look at what changes are in the patch.

git apply name.patch

git am *.patch

git am --signoff < fix_empty_poster.patch

git merge

-Xignore-all-space or -Xignore-space-change - Ignores whitecaps

git merge -Xours branch1

git merge -Xtheirs branch1

Advanced

GIT Hooks

Git hooks are scripts that run automatically every time a particular event occurs in a Git repository.

Common use cases for Git hooks include encouraging a commit policy, implementing continuous integration workflows etc.

The built-in scripts are mostly shell and PERL scripts, but you can use any scripting language like Python etc

Hooks are local to any given Git repository, and they are not copied over to the new repository when you run git clone. And, since hooks are local, they can be altered by anybody with access to the repository.

Hooks

pre-commit
prepare-commit-msg
commit-msg
post-commit Email/SMS team members of a new commit.
pre-rebase
post-checkout
post-merge
pre-receive
applypatch-msg
pre-applypatch
post-applypatch
update
post-receive
post-update
pre-receive - Enforce project coding standards.
pre-auto-gc
post-rewrite
pre-push

Hooks

- **pre-commit -** The pre-commit script is executed every time you run git commit before Git asks the developer for a commit message or generates a commit object. You can use this hook to inspect the snapshot that is about to be committed.
- For example,
    - You may want to run some automated 3rd party tests that make sure the commit doesn’t break any existing functionality.
    - Check commit message for spelling errors
    - Check whitespace errors

Hooks

- **pre-commit -**This script aborts the commit if it finds any **whitespace errors**, as defined by the git diff-index command (trailing whitespace, lines with only whitespace, and a space followed by a tab inside the initial indent of a line are considered errors by default).

```
#!/bin/sh

# Check if this is the initial commit
if git rev-parse --verify HEAD >/dev/null 2>&1
then
    echo "pre-commit: About to create a new commit..."
    against=HEAD
else
    echo "pre-commit: About to create the first commit..."
    against=4b825dc642cb6eb9a060e54bf8d69288fbee4904
fi

# Use git diff-index to check for whitespace errors
echo "pre-commit: Testing for whitespace errors..."
if ! git diff-index --check --cached $against
then
    echo "pre-commit: Aborting commit due to whitespace errors"
    exit 1
else
    echo "pre-commit: No whitespace errors :)"
    exit 0
fi
```

Hooks

- **prepare-commit-msg -** The prepare-commit-msg hook is called after the pre-commit hook to populate the text editor with a commit message. This is a good place to alter the automatically generated commit messages for squashed or merged commits.

```
- #!/usr/bin/env python
- import sys, os, re
- from subprocess import check_output

- # Collect the parameters
- commit_msg_filepath = sys.argv[1]
- if len(sys.argv) > 2:
-     commit_type = sys.argv[2]
- else:
-     commit_type = ''
- if len(sys.argv) > 3:
-     commit_hash = sys.argv[3]
- else:
-     commit_hash = ''

- print "prepare-commit-msg: File: %s\nType: %s\nHash: %s" % (commit_msg_filepath, commit_type, commit_hash)

- # Figure out which branch we're on
- branch = check_output(['git', 'symbolic-ref', '--short', 'HEAD']).strip()
- print "prepare-commit-msg: On branch '%s'" % branch

- # Populate the commit message with the issue #, if there is one
- if branch.startswith('issue-'):
-     print "prepare-commit-msg: Oh hey, it's an issue branch."
-     result = re.match('issue-(.*)', branch)
-     issue_number = result.group(1)

-     with open(commit_msg_filepath, 'r+') as f:
-         content = f.read()
-         f.seek(0, 0)
-         f.write("ISSUE-%s %s" % (issue_number, content))
```

echo "# Please include a useful commit message!" > $1

Hooks

- **commit-msg -** The commit-msg hook is much like the prepare-commit-msg hook, but it's called after the user enters a commit message. This is an appropriate place to warn developers that their message doesn't adhere to your team's standards.

For example, the following script checks to make sure that the user didn't delete the ISSUE-[#] string that was automatically generated by the prepare-commit-msg hook in the previous section.

```
#!/usr/bin/env python

import sys, os, re
from subprocess import check_output

# Collect the parameters
commit_msg_filepath = sys.argv[1]

# Figure out which branch we're on
branch = check_output(['git', 'symbolic-ref', '--short', 'HEAD']).strip()
print "commit-msg: On branch '%s'" % branch

# Check the commit message if we're on an issue branch
if branch.startswith('issue-'):
    print "commit-msg: Oh hey, it's an issue branch."
    result = re.match('issue-(.*)', branch)
    issue_number = result.group(1)
    required_message = "ISSUE-%s" % issue_number

    with open(commit_msg_filepath, 'r') as f:
        content = f.read()
        if not content.startswith(required_message):
            print "commit-msg: ERROR! The commit message must start with '%s'" % required_message
            sys.exit(1)
```

Hooks

- **post-Commit -** The post-commit hook is called immediately after the commit-msg hook.

For example, if you want to email your boss every time you commit a snapshot (probably not the best idea for most workflows), you could add the following post-commit hook.

```
#!/usr/bin/env python

import smtplib
from email.mime.text import MIMEText
from subprocess import check_output

# Get the git log --stat entry of the new commit
log = check_output(['git', 'log', '-1', '--stat', 'HEAD'])

# Create a plaintext email message
msg = MIMEText("Look, I'm actually doing some work:\n\n%s" % log)

msg['Subject'] = 'Git post-commit hook notification'
msg['From'] = 'parvezmisarwala@gmail.com'
msg['To'] = ‘parvez.mslc@gmail.com'

# Send the message
SMTP_SERVER = 'smtp.gmail.com'
SMTP_PORT = 587

session = smtplib.SMTP(SMTP_SERVER, SMTP_PORT)
session.ehlo()
session.starttls()
session.ehlo()
session.login(msg['From'], 'secretPassword')

session.sendmail(msg['From'], msg['To'], msg.as_string())
session.quit()
```

Git Submodule

To Create submodule

- git submodule add ../subrepo1
- git submodule init
- git submodule update

To update

- CD into repo2 - git pull ../../repo2 (fetches all commits of repo2 in repo1)

Git Subtree

git subtree add --prefix .vim/bundle/subrepo <URI> master

git subtree pull --prefix .vim/bundle/subrepo <URI> master

Git Subtree

To add a subtree

git remote add rack_remote git@github.com:schacon/rack.git
git fetch rack_remote
git checkout -b rack_branch rack_remote/master
git read-tree --prefix=rack/ -u rack_branch

To Update

git checkout rack_branch
git pull
git checkout master
git merge --squash -s subtree --no-commit rack_branch

git diff-tree -p rack_branch

Splitting Repository

- Turn a folder within a repository into a brand new repository
- git clone https://github.com/USERNAME/REPOSITORY-NAME
- cd REPOSITORY-NAME
- git filter-branch --prune-empty --subdirectory-filter FOLDER-NAME BRANCH-NAME

Splitting Repository

- git clone git@bitbucket.org:tutorials/splitpractice.git freshrepo
- bigdir lildir lildir2
- git remote rm origin
- git filter-branch --index-filter 'git rm --cached -r lildir lildir2' --

  —all
- git remote add origin https://tutorials@bitbucket.org/tutorials/

  freshrepo.git
- git push origin master

To Amend Last commit

To Amend Last commit

-One method

  -git rebase —interactive ‘31f73c0^’

  -choose edit instead of default pick

  -make changes and do git add .

  -git rebase —continue

  -(This will update the current commit, but commit ID will be new)

-Second method

  -make changes and do git add .

  -git commit —all —amend

git Perforce Migration

git-p4

These commands require the Perforce Helix command line client

- mkdir git-p4-area
- vi .p4config
- P4PORT=perforce:1666
- P4USER=zig
- P4CLIENT=zig-git-p4
- p4 login
- mkdir demo and cd demo
- git-p4 clone //depot@all . —verbose
- git remote add origin git@github.com:org/demo.git
- git push origin master

GIT Fusion

- Install Git Fusion
- Set up the correct views of your data, including the branching structure
- Use any Git client to clone from Git Fusion
- Push your repo into Bitbucket

Plugin Development

Pre-requisites

- Atlassian Plugin SDK
- Maven 3.0

Steps

- atlas-create-stash-plugin
    - com.atlassian.stash.plugin.demoplugin
    - demo-plugin

- atlas-create-stash-plugin-module
    - 8
    - democlass

Bitbucket Admin

User & Groups
External User Directory
Global Permissions
Setting up mail server
Integration bitbucket with Jira
Connecting bitbucket with external database
Setting up SSH

Backup Bitbucket

To Backup
Download and install bitbucket-backup-client-3.3.1
Configure backup-config.properties file
cd <path/to/backup-config.properties file>
java -jar <path/to/bitbucket-backup-client.jar>

To Restore
java -Dbitbucket.home="path/to/bitbucket/home" -jar bitbucket-restore-client.jar /path/to/original/backup/file