Git Tutorial

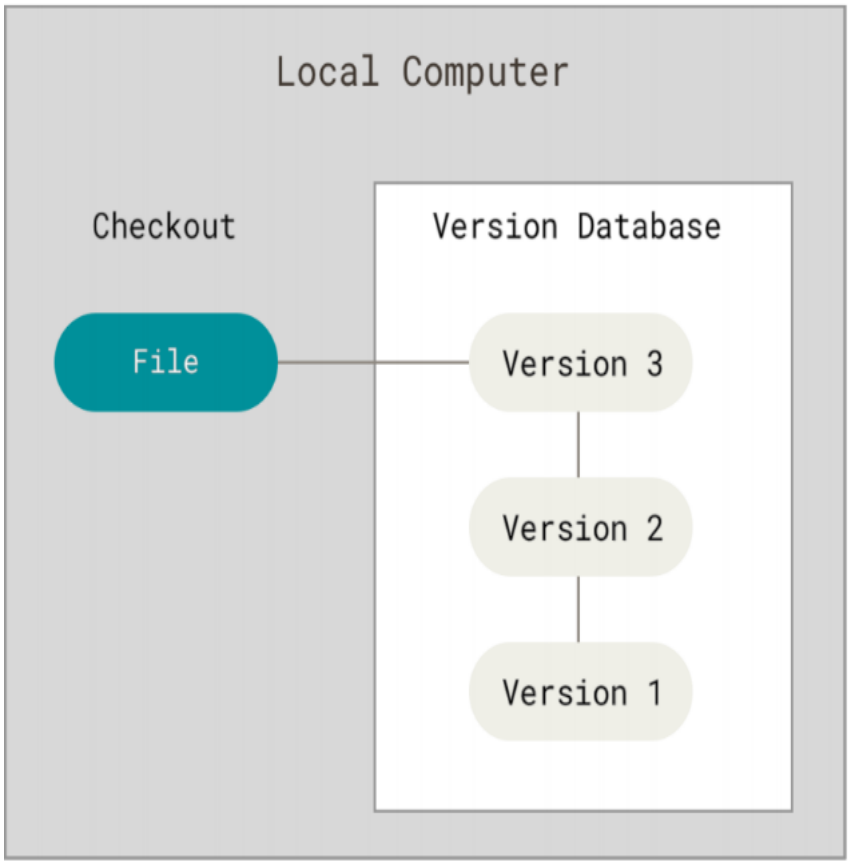
How to git for fun & profit

Thomas Dost

March 28, 2022

What is git

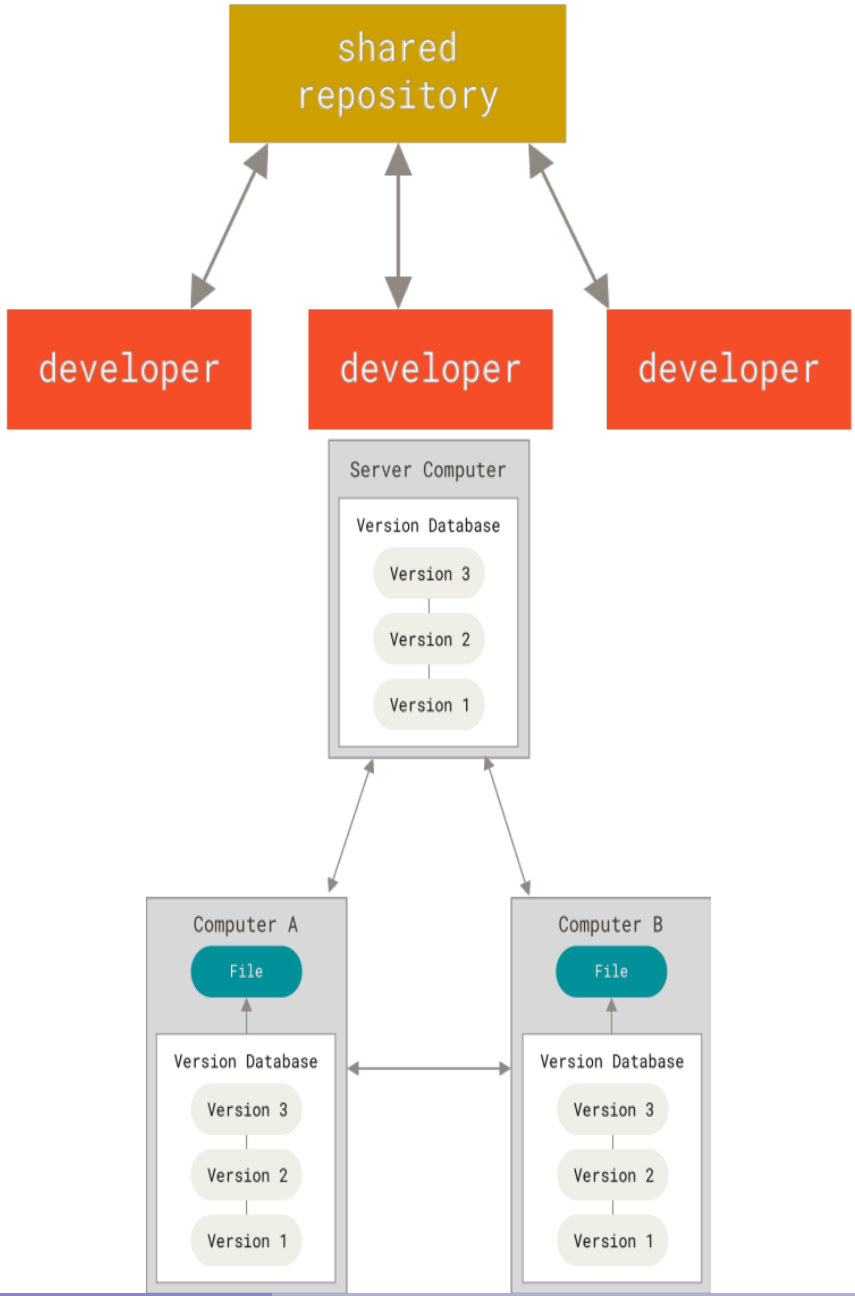
1. version control system
2. a version control sytem is a system that records changes to a set of files over time so that you can recall specific versions later



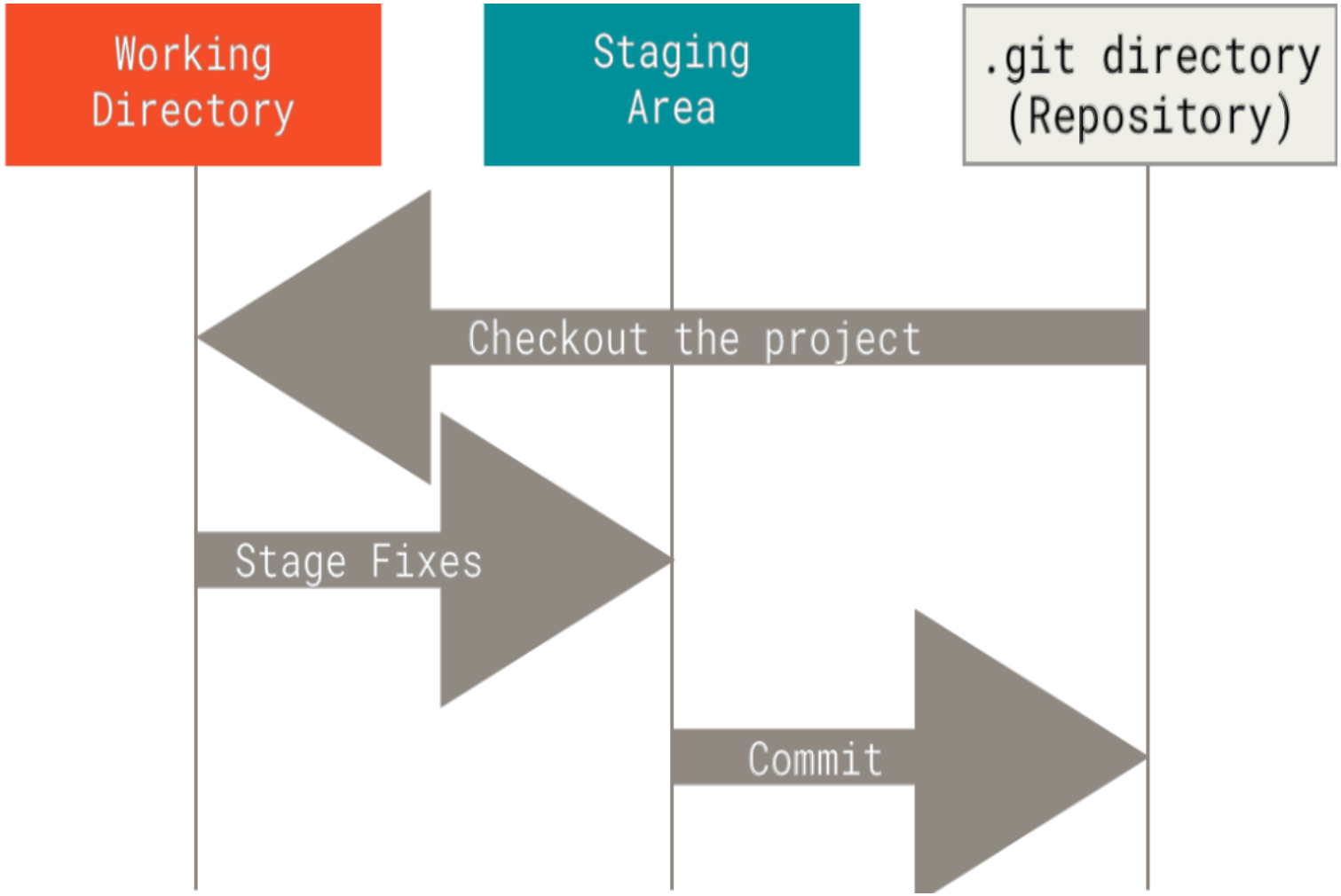
Centralized version control vs distributed version control

1. all files are stored on a central server
2. each developer can checkout a specific file, which makes other developers unable to edit it as well

1. each developer has local copy of all the files
2. most operations are local, i.e. each developer can work idependently of the the others
3. everything you do in git is checksumed, that means its impossible to change anything without Git knowing about it



Three stages of git



Setting up git

Prerequisite: Some version of git should be installed(in my case it's 2.30.2)

Git config paths

- /.gitconfig
- /.config/git/config
- .git/config

Show current config

```
1  git config --list --show-origin
2  git config  user.name "Thomas Dost"
3  git config  user.email ThomasDost@example.com
4  git config  core.editor vim
```

Getting help

```
1  git help <command>
```

Making a new repository

New repository

```
1  git init
2  git init <name>
```

Cloning an existing repository

```
1  git clone <url>
```

Making a new repository

New repository

```
1  git init
2  git init <name>
```

Cloning an existing repository

```
1  git clone <url>
```

Adding files to the staging area

```
1  git add main.py
```

Making a new repository

New repository

```
1  git init
2  git init <name>
```

Cloning an existing repository

```
1  git clone <url>
```

Adding files to the staging area

```
1  git add main.py
```

Committing changes

```
1  git commit main.py
```

Making a new repository

New repository

```
1  git init
2  git init <name>
```

Cloning an existing repository

```
1  git clone <url>
```

Adding files to the staging area

```
1  git add main.py
```

Committing changes

```
1  git commit main.py
```

Amending/Verbose

```
1  git commit -a main.py #amend a commit
2  git commit -v main.py #show difference
```

Interactive adding

```
1  git add -p
```

- -y stage the chunk
- -n ignore the chunk
- -s split into smaller chunks
- -e edit the chunk
- -q exit

Look back in time(git log)

```
1  git log
```

- –graph
- –oneline
- –decorate

Look back in time(git log)

```
1  git diff
```

- –staged also diff files which are already staged

```
1  git difftool
```

Checking out older commits

```
1  git checkout <commit-id>
```

Reverting changes/Cleaning up

revert unstaged changes

```
1  git restore <file>
```

reset all staged changes

```
1  git reset HEAD
```

revert last commit

```
1  git revert HEAD
```

Remove directories

```
1  git rm
```

Remove all files not under gits control

```
1  git clean
```

git revert

reverts and already committed change by inverting it and appending a new comit

```
1  git revert <commit-hash>
```

- -n, does not create a new commit, instead stages the changes

git reset

Three primary modes of invocation, which correspond to gits internal management stages. Mostly used to remove files from the staging area

- –soft: the commit tree(HEAD)
- –mixed: the staging index
- –hard: the working directory

```
1  git reset # remove all files from staging area
2  git reset <file> # removes <file> from statging area
3  git reset --hard
```

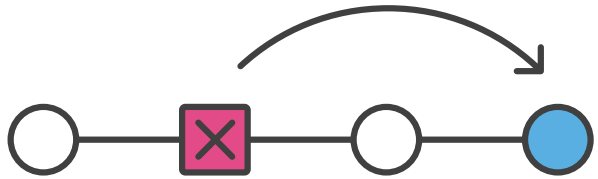
–hard undoes all uncommited changes

git revert vs. git reset

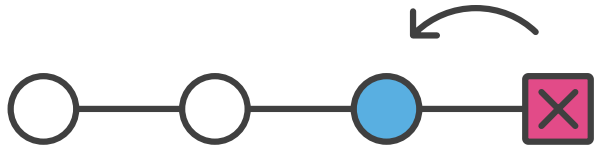
reverting does not change the history i.e. its safe.

git revert is able to target arbitrary commits in the history, while reset only works backwards from the current commit

Reverting



Resetting



Branching 1

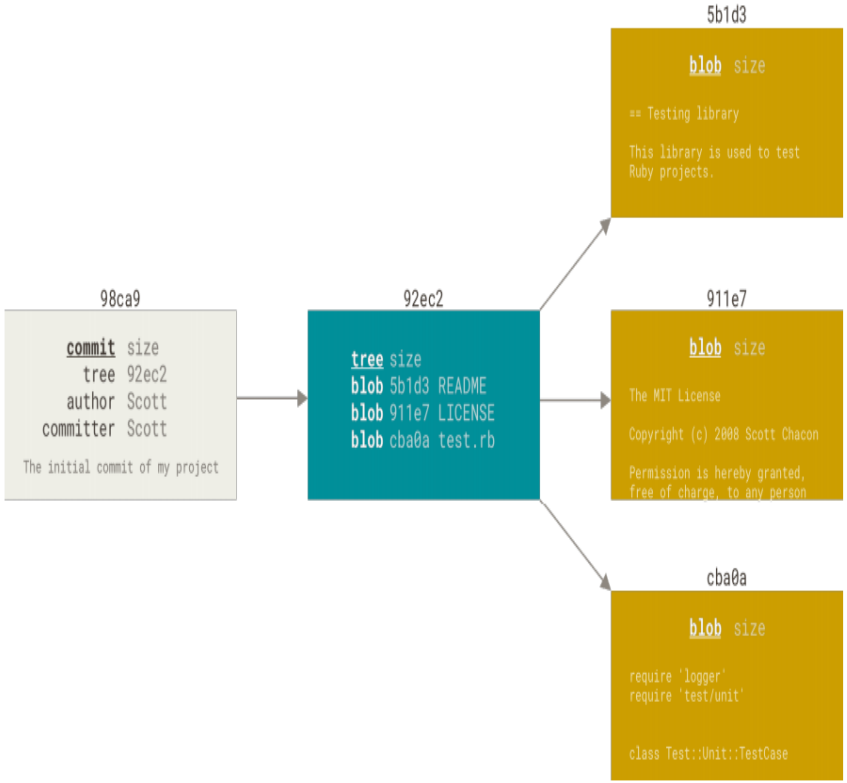
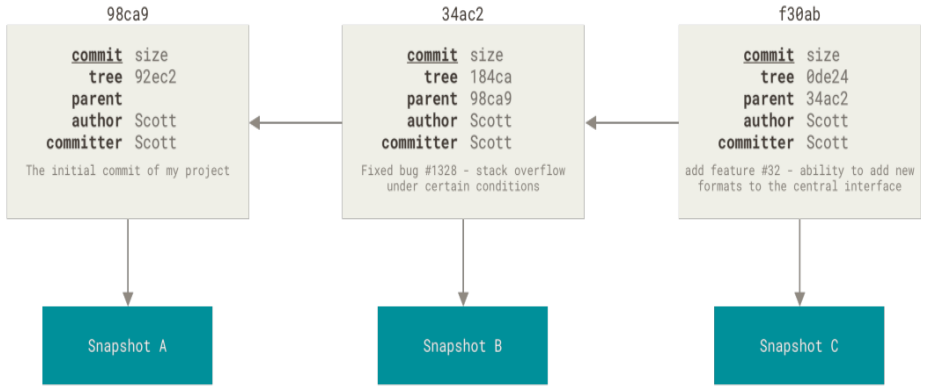
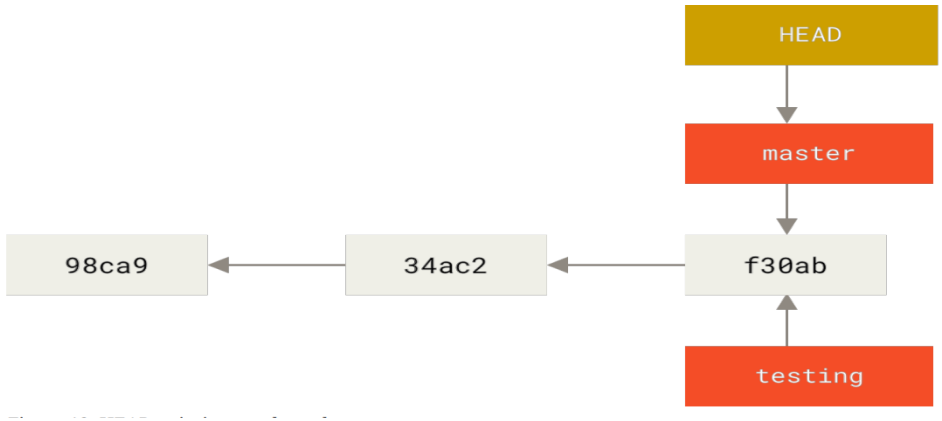


Figure 9. A commit and its tree

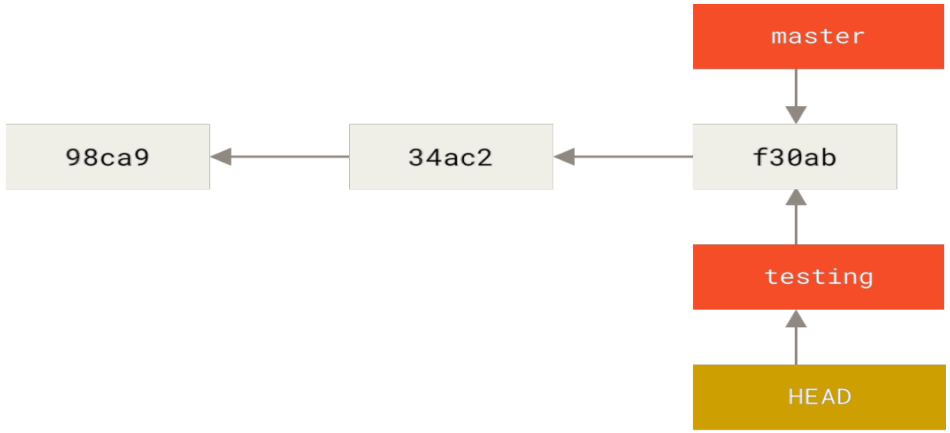


Creating a branch

```
1 git branch <branch-name> # create branch
```




```
1 git checkout <branch-name> #switch to
    branch
2 git checkout -b <branch-name> #create and
    switch
```



Merging

```
1  git checkout main
2  git branch -d feature/user #safe delete
3
```

Collaboration

We now know how to

1. execute basic git commands
2. work with branches

But everything we did was local, so how do we work online or with other people via git?

```
1  git push # upload local changes
2  git pull # download changes and merge into branches
```

Merge conflicts

What happens when we make conflicting changes?

.gitignore

Used to specify specific files, file endings which git should ignore, also possible to ignore whole folders

git tag

```
1  git tag # list all tags
2  git tag -a <tag-name> -m "<message>" # annotaged tag
3  git tag <tag-name> # lightweight tag
4
5  git tag <tag-name> <commit-id> # add tag to earlier commit
6  git push --tags # by defaults tags are not pushed
7  git tag -d <name> # delete tag
8  git checkout <tag-name> #go to specific commit by tag name
```

Stashing

What happens when you in the middle of changing sth., and for example, a colleague wants you to look at the stuff he just did.

```
1  git stash #
2  git stash pop
```

Git & RStudio

Versioning

no standard way of when/what to commit commit semantically similiar units

SemVer Prerequisite: A version number in the form of MAJOR.MINIOR.PATCH Change

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backwards compatible manner
3. PATCH version when you make backwards compatible bug fixes

All of those, of course, warrant a commit but are not really applicable outside of software engineering
There is a another workflow based on rebase, which enables you to basically just commit everything

git rebase

Combines older/multiple commits into one base commit.

Alleviates the issue of what/when to commit

Do NOT use in public repositories/already published changes. There are workflows to mitigate this risk

```
1  git rebase -i
```

git difftools

Tools

1. GitKraken(Partially free)
2. GitHub Desktop(free)
3. MagitEmacs(free)
4. SourceTree(free)

Things not covered

1. git lfs: git large file storage
2. git merge in-depth: understanding what merge does exactly is pretty usefull when working with it
3. git rebase: "alternative" to merge, allows a linear commit history
4. git rebase: Changing history
5. git rebase vs merge: rebase allows a cleaner history
6. git workflows: automatic running of tests and deployment i.e. publishing to pip or stuff like that
7. forking: how copy other open source project and contribute to them -¿ leads to pull requests
8. git blame: who/what was modified on a specific file

Things not covered

9. git hooks: run user scripts at specific git events(server/client side)
10. git cherrypick
11. git submodules: incorporate external code
    https://www.atlassian.com/git/tutorials/git-submodule
12. git subtrees: alternative to git submodules
13. git reflog: Allows you to visit commits, which are not referenced by any brach anymore, for example after a squash
14. git bisect: "automatic" bug finding, requires tests
15. git alias: Make "shortcuts" for git commands

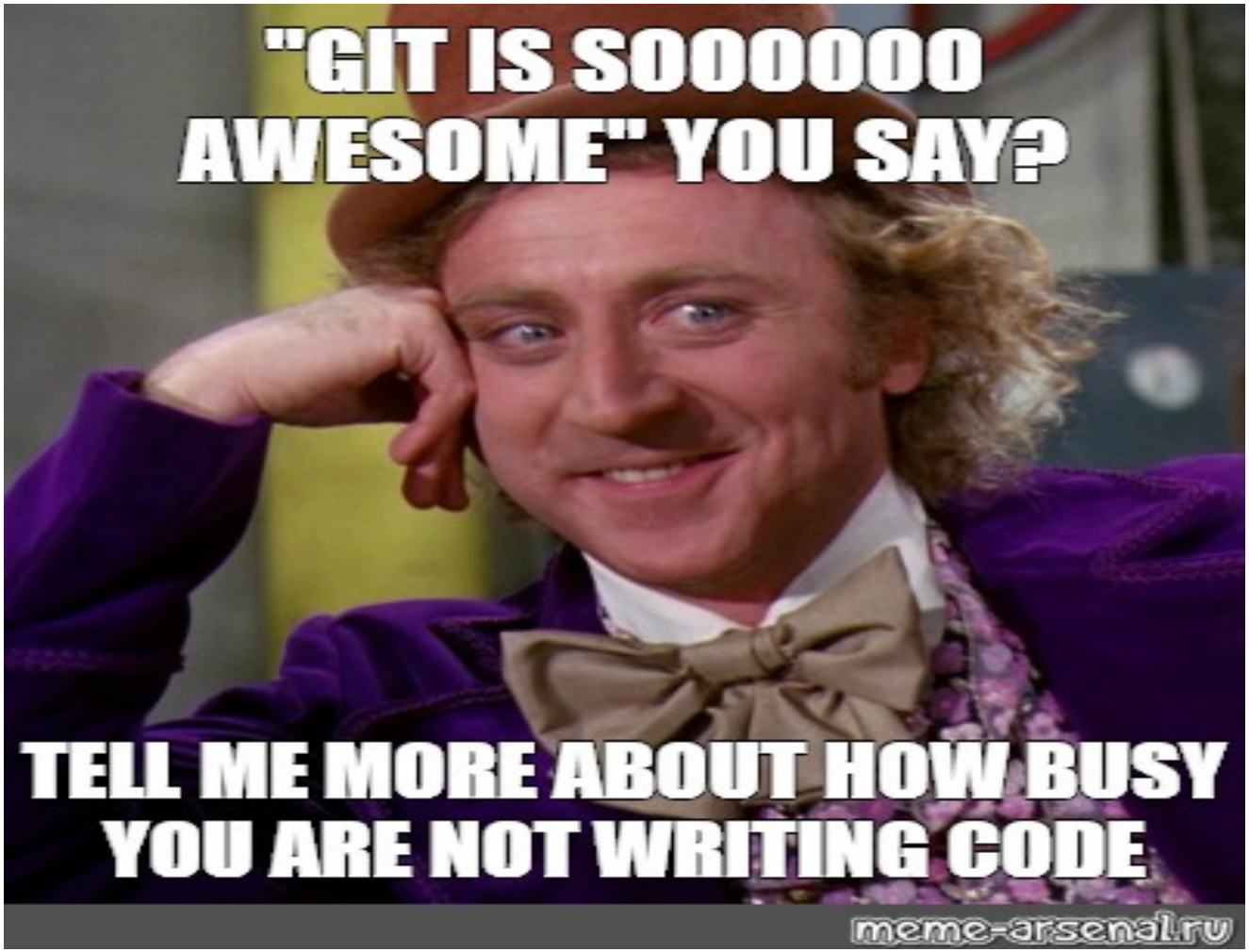
Warning

A lot of the commands which change the history have to be used with caution if you work on a public repo. with other developers, please read up/think about what happens when your change local history referenced by other developers and try to push those changes, but that is out of the scope for this presentation.

References

1. https://git-scm.com/site(All material under MIT)
2. ProGit(All material under CreativeCommons CC-BY-NC-SA)
3. Atlassian tutorials(Just stolen :'), no license given)
4. https://semver.org/lang/en/

Questions?


"GIT IS SOOOOOO AWESOME" YOU SAY?
TELL ME MORE ABOUT HOW BUSY YOU ARE NOT WRITING CODE
meme-arsenal.ru